FISCO BCOS Documentation

发布 ***v2.7.1***

fisco-dev

2020 年 **12** 月 **29** 日

平台介绍

1 FISCO BCOS 区块链 3

2 版本信息 7

3 搭建第一个区块链网络 21

4 开发第一个区块链应用 31

5 高阶教程 57

6 关键概念 93

7 区块链网络搭建 101

8 区块链应用开发 203

9 问题排查 441

10 开发部署工具(build_chain.sh) 449

11 命令行交互控制台 459

12 WeBASE管理平台 549

13 区块链浏览器 553

14 运维部署工具 573

15 数据治理通用组件 637

16 国密使用 643

17 证书相关 645

18 JDK版本 647

19 整体架构 649

20 模块设计 655

21 JSON-RPC API 743

22 深入浅出FISCO BCOS 781

23 社区 1317

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。

- 了解FISCO BCOS区块链
- FISCO BCOS 2.X新特性
- 搭建第一个区块链网络
- 开发第一个区块链应用
- 关键概念
- 区块链网络搭建
- 区块链应用开发
- FISCO BCOS Java SDK
- 问题排查
- 整体架构
- 共识算法
- 交易并行
- 分布式存储
- 预编译合约
- 更多设计文档
- 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

CHAPTER 1

FISCO BCOS 区块链

标签：FISCO BCOS介绍 关键特性 开发工具

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
- 基于Web3SDK的交互式的命令行工具console
- Interactive command line tool console based on Java SDK
- 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。
- 区块链数据治理组件解决方案：稳定、高效、安全的区块链数据治理组件解决方案，分别从底层数据存储层、智能合约数据解析层和应用层三个方面，提供了区块链数据挖掘、裁剪、扩容、可信存储、抽取、分析、审计、对账、监管等数据治理方面的关键能力。
- 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。
- FISCO BCOS区块链工具箱：与WeBase/Remix/VSCode/ChainIDE等IDE协同工作，提升开发体验与开发效率。
- Ansible for FISCO BCOS 自动化生成企业级部署文件：本项目由为区块链开源项目 FISCO-BCOS 提供了自动化生成企业级配置文件的 ansible playbook。2 群组 3 机构 6 节点的环境，可以在 30 秒内（除下载时间）生成配置，极大简化了部署难度，避免了手工配置容易发生的错误。

CHAPTER 2

版本信息

标签：版本特性 `Release Note`

2.1 FISCO BCOS 2.x 新特性

2.1.1 群组架构

群组架构是FISCO BCOS 2.0众多新特性中的主线，创造灵感来源于人人都熟悉的群聊模式——群的建立非常灵活，几个人就可以快速拉个主题群进行交流。同一个人可以参与到自己感兴趣的多个群里，并行地收发信息。现有的群也可以继续增加成员。

采用群组架构的网络中，根据业务场景的不同，可存在多个不同的账本，区块链节点可以根据业务关系选择群组加入，参与到对应账本的数据共享和共识过程中。该架构的特点是：

- 各群组独立执行共识流程，由群组内参与者决定如何进行共识，一个群组内的共识不受其他群组影响，各群组拥有独立的账本，维护自己的交易事务和数据，使得各群组之间解除耦合独立运作，可以达成更好的隐私隔离；
- 机构的节点只需部署一次，通过群组设置即可参与到不同的多方协作业务中，或将一个业务按用户、时间等维度分到各群组，群组架构可快速地平行扩展，在扩大了业务规模同时，极大简化了运维复杂度，降低管理成本。

更多的群组介绍，请参考 群组架构设计文档 和 群组使用教程

2.1.2 分布式存储

FISCO BCOS 2.0新增了对分布式数据存储的支持，节点可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。该方案有以下优点：

- 支持多种存储引擎，选用高可用的分布式存储系统，可以支持数据简便快速地扩容；
- 将计算和数据隔离，节点故障不会导致数据异常；
- 数据在远端存储，数据可以在更安全的隔离区存储，这在很多场景中非常有意义；
- 分布式存储不仅支持Key-Value形式，还支持SQL方式，使得业务开发更为简便；

- 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；
- 优化了数据存储的结构，更节约存储空间。

同时，2.0版本仍然兼容1.0版本的本地存储模式。更多关于存储介绍，请参考 分布式存储操作手册

2.1.3 并行计算模型

2.0版本中新增了合约交易的并行处理机制，进一步提升了合约的并发吞吐量。

1.0版本以及大部分业界传统区块链平台，交易是被打包成一个区块，在一个区块中交易顺序串行执行的。 2.0版本基于预编译合约，实现一套并行交易处理模型，基于这个模型可以自定义交易互斥变量。在区块执行过程中，系统将会根据交易互斥变量自动构建交易依赖关系图——DAG，基于DAG并行执行交易，最好情况下性能可提升数倍（取决于CPU核数）。

更多并行计算模型的介绍，请参考并行交易的 设计文档 和 使用手册。

2.1.4 预编译合约

FISCO BCOS 2.0提供预编译合约框架，支持采用C++编写合约，其优势是合约调用响应更快，运行速度更高，消耗资源更少，更易于并行计算，极大提升整个系统的效率。FISCO BCOS内置了多个系统级的合约，提供准入控制、权限管理、系统配置、CRUD式的数据存取等功能，这些功能天然集成在底层平台里，无需手动部署。

FISCO BCOS提供标准化接口和示例，帮助用户进行二次开发，便于用户编写高性能的业务合约，并方便地部署到FISCO BCOS里运行。预编译合约框架兼容EVM引擎，形成了“双引擎”架构，熟悉EVM引擎的用户可以选择将Solidity合约和预编译合约结合，在满足业务逻辑的同时获得巨大的效率提升。

另外，还有类似CRUD操作等也由预编译合约实现，更多预编译合约的介绍，请参考 预编译设计文档 和 预编译合约开发文档

2.1.5 CRUD接口

FISCO BCOS 2.0新增符合CRUD接口的合约接口规范，简化了将主流的面向SQL设计的商业应用迁移到区块链上的成本。其好处显而易见：

- 与传统业务开发模式类似，降低了合约开发学习成本；
- 合约只需关心核心逻辑，存储与计算分离，方便合约升级；
- CRUD底层逻辑基于预编译合约实现，数据存储采用分布式存储，效率更高；

同时，2.0版本仍然兼容1.0版本的合约，更多关于CRUD接口的介绍，请参考 使用CRUD接口 。

2.1.6 控制台

FISCO BCOS 2.0新增控制台，作为FISCO BCOS 2.0的交互式客户端工具。

控制台安装简单便捷，简单配置后即可和链节点进行通信，拥有丰富的命令和良好的交互体验，用户可以通过控制台查询区块链状态、读取和修改配置、管理区块链节点、部署并调用合约。控制台给用户管理、开发、运维区块链带来了巨大的便利，降低了操作繁琐性和使用门槛。

相比于传统的nodejs等脚本工具，控制台安装简单、使用体验更好。详细请查看 基于Java SDK的控制台使用手册 和 基于Web3SDK的控制台使用手册。

2.1.7 虚拟机

2.0版本引入了最新的以太坊虚拟机版本，支持Solidity 0.5版本。同时，引入了EVMC扩展框架，支持扩展不同虚拟机引擎。 底层内部集成支持interpreter虚拟机，未来可扩展支持WASM/JIT等虚拟机。

更多关于虚拟机的介绍，请参考 虚拟机设计文档

2.1.8 密钥管理服务

2.0版本对落盘加密进行了重塑升级，开启落盘加密功能时，依赖KeyManager服务进行密钥管理，安全性更强。

KeyManager在Github开源发布，节点与KeyManager的交互协议是开放的，支持机构设计实现符合自身密钥管理规范的KeyManager服务，比如采用硬件加密机技术。 该部分更详细的文档请参考 使用文档 和 设计文档

2.1.9 准入控制

2.0版本对准入机制进行了重塑升级，包括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。

采用新的权限控制体系，基于表进行访问权限的设计，另外还支持CA黑名单机制，可以实现对作恶/故障节点的屏蔽。 详情请查看 准入机制设计文档

2.1.10 异步事件

2.0版本同时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。

2.1.11 模块重塑

2.0版本对核心模块进行升级重塑，进行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。

2.2 更多版本

FISCO BCOS 2.x Releases

- FISCO BCOS v2.7.0 [release]
- FISCO BCOS v2.6.0 [release]
- FISCO BCOS v2.5.0 [release]
- FISCO BCOS v2.4.0 [release]
- FISCO BCOS v2.3.0 [release]
- FISCO BCOS v2.2.0 [release]
- FISCO BCOS v2.1.0 [release]
- FISCO BCOS v2.0.0 [release]
- FISCO BCOS v2.0.0-rc3 [release]
- FISCO BCOS v2.0.0-rc2 [release]
- FISCO BCOS v2.0.0-rc1 [release]

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

2.2.1 v2.7.0

标签：`v2.7.0` 版本特性

v2.6.x升级到v2.7.0

- **兼容升级**：直接替换v2.6.x节点的二进制为 v2.7.0二进制，升级后的版本修复v2.6.x中的bug，并支持取消事件订阅、查询节点订阅的topics信息、批量返回交易回执等功能，且节点交易和区块相关的RPC接口返回了交易和区块的所有字段内容、节点返回的回执信息中带有交易执行的错误信息
- **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v2.7.0所有新特性
- v2.7.0 Release Note

变更描述

新增

- 合约生命周期管理提供revokeManager功能，可通过v2.7.0控制台的revokeContractStatusManager命令使用
- 新增委员权限投票的查询接口，可通过v2.7.0控制台的queryVotesOfThreshold和queryVotesOfMember使用
- 支持取消事件订阅
- 添加批量返回回执的方法getBatchReceiptsByBlockNumberAndRange和getBatchReceiptsByBlockHashAndRange
- 添加getNodeInfo接口，返回节点订阅的topics信息
- Table合约增加address类型的EQ condition查询

更改

- MySQL存储模式支持MySQL 8.0
- 重构交易和区块相关的RPC接口，返回交易和区块的所有字段内容
- 交易推送的回执中带有交易执行结果的报错信息
- P2P模块添加读写idle检查的逻辑

修复

- 解决gcc9和clang12编译报错的问题
- Precompiled合约报错信息写入到回执中，给出更明确的报错提示
- 优化落盘加密，解决落盘加密模式下，存储空间占用大的问题
- 修复MacOS系统下CachedStorage清理缓存导致节点宕机的问题
- 修复MySQL存储模式下，CNS合约名过长导致区块提交异常的问题

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v2.6.x方法

当节点采用兼容模式从v2.6.x升级至v2.7.0后，可直接通过将节点二进制替换回v2.6.x完成回滚。

2.2.2 v2.6.0

标签：`v2.6.0` 版本特性

---

v2.5.x升级到v2.6.0

- **兼容升级**：直接替换v2.5.x节点的二进制为 v2.6.0二进制，新增支持国产麒麟操作系统、solidity 0.6、IPv6、节点时间对齐、区块执行超时时间配置等功能，升级后的版本修复v2.5.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v2.6.0所有新特性
- v2.6.0 Release Note

变更描述

新增

- 新增对国产麒麟操作系统V10（Kunpeng-920/ARM aarch64）的支持
- 新增对Solidity 0.6的支持
- 新增节点间时间对齐机制
- 新增对IPv6的支持
- 新增群组SDK白名单机制，默认关闭
- 新增获取区块头的接口`getBlockheaderByNumber/getBlockHeaderByHash`
- 新增支持配置区块执行超时时间

更改

- rPBFT共识算法引入VRF随机数算法，进一步保证共识算法的安全性
- evm引擎由aleth-interpreter替换为evmone
- 升级EVMC接口版本为7.3.0
- 源码编译要求GCC版本不小于5.4.0
- 支持AMOP消息路由到节点自身
- MySQL存储模式下区块等数据使用二进制存储并启动压缩

修复

- 修复因为交易广播标记缓存没有清理干净导致的轻微内存泄露问题
- 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
- 在MySQL存储模式下，Solidity合约写接口中调用Table的select接口时，偶现的Gas计算不一致问题
- 用户自己生成的错误证书导致coredump问题
- 在Solidity中使用汇编调用以太坊预编译合约0x1-0x9，对错误输入没有返回0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v2.5.x方法

当节点采用兼容模式从v2.5.x升级至v2.6.0后，可直接通过将节点二进制替换回v2.5.x完成回滚。

2.2.3 v2.5.0

标签：`v2.5.0` 版本特性

---

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

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。

2.2.4 v2.4.0

标签： `v2.4.0` 版本特性

---

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

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。

2.2.5 v2.3.0

标签： `v2.3.0` 版本特性

---

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

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。

2.2.6 v2.2.0

标签：`v2.2.0 版本特性`

---

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

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。

2.2.7 v2.1.0

标签：`v2.1.0` 版本特性

---

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

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。

2.2.8 v2.0.0

标签：`v2.0.0` 版本特性

---

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

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。

2.2.9 v2.0.0-rc3

标签：`v2.0.0-rc3` 版本特性

---

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

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

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。

2.2.10 v2.0.0-rc2

标签：`v2.0.0-rc2` 版本特性

---

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

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

兼容性说明

2.2.11 v2.0.0-rc1

标签：`v2.0.0-rc1` 版本特性

---

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

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。

兼容性说明

CHAPTER 3

搭建第一个区块链网络

标签：搭建区块链网络 区块链教程 `HelloWorld` 控制台调用合约

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。

注解:

- 搭建全链路国密版本的链，请参考这里 。

3.1 1. 搭建单群组FISCO BCOS联盟链

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。
- 使用docker模式搭建 ，供有丰富docker经验和容器化部署需求的用户参考。

3.1.1 第一步. 安装依赖

开发部署工具 `build_chain.sh`脚本依赖于`openssl`, `curl`，根据您使用的操作系统，使用以下命令安装依赖。

安装macOS依赖

```
brew install openssl curl
```

安装ubuntu依赖

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

安装centos依赖

```
sudo yum install -y openssl openssl-devel
```

3.1.2 第二步. 创建操作目录, 下载安装脚本

```
## 创建操作目录
cd ~ && mkdir -p fisco && cd fisco

## 下载脚本
curl -#LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.7.1/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/manual/build_chain.sh && chmod u+x build_chain.sh*

3.1.3 第三步. 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。

注解:

- 国密版本请执行 `bash build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545 -g -G`
- 其中-g表示生成国密配置，-G表示使用国密SSL连接
- web3sdk已经支持国密SSL，如果使用web3sdk建议带上-G选项使用国密SSL

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

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
```

3.1.4 第四步. 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
```

3.1.5 第五步. 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
```

(continues on next page)

(续上页)

```
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
```

3.1.6 第六步. 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连接信息，从输出可以看出node0与另外3个节点有连接。

```
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|2020-12-22 17:24:43.729402|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=2e133146...
info|2020-12-22 17:24:47.740603|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=eb199760...
```

3.2 2. 配置及使用控制台

重要:

- 控制台1.x 系列基于 Web3SDK 实现，控制台2.6之后 基于 Java SDK 实现，最新版本控制台基于 Java SDK 实现
- 2.6及其以上版本控制台使用文档请 参考这里 ， 1.x版本控制台使用文档请 参考这里
- 可通过命令 ./start.sh --version 查看当前控制台版本
- 基于 Web3SDK 开发应用时将 solidity 代码转换为 java 代码时，必须使用 1.x 版本控制台，具体请参考 这里

在控制台链接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。2.6版本控制台指令详细介绍参考这里，1.x版本控制台指令详细介绍参考这里。

3.2.1 第一步. 准备依赖

- 安装java （推荐使用java 14）.

```
# ubuntu系统安装java
sudo apt install -y default-jdk

#centos系统安装java
sudo yum install -y java java-devel
```

- 获取控制台并回到fisco目录

```
cd ~/fisco && curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.
↪7.1/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channel端口。

```
# 最新版本控制台使用如下命令拷贝配置文件
cp -n console/conf/config-example.toml console/conf/config.toml
```

- 配置控制台证书

注解:

使用1.x版本控制台时：

- 搭建国密版时，如果使用国密SSL请执行 `cp nodes/127.0.0.1/sdk/gm/* console/conf/`
- 搭建国密版时，请修改 applicationContext.xml 中 encryptType 修改为1

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

3.2.2 第二步. 启动并使用控制台

- 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/config.toml中节点端口配置是否正确

```
=====================================================================================================
Welcome to FISCO BCOS console(2.6.0)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________  ______   ______   ______   ______          ______   ______   ______␣
↪   ______
|        \|      \ /      \ /      \ /      \        |      \ /      \ /      ␣
↪\ /      \
| $$$$$$$$ \$$$$$$|  $$$$$$\|  $$$$$$\|  $$$$$$\      | $$$$$$$\|  $$$$$$\|  $$$$$
↪$\|  $$$$$$\
| $$__       | $$  | $$___\$$| $$   \$$| $$  | $$      | $$__/ $$| $$   \$$| $$  | $
↪$| $$___\$$
| $$  \      | $$   \$$     \ | $$       | $$  | $$      | $$    $$| $$       | $$  | $
↪$ \$$     \
```

(continues on next page)

(续上页)

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

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

若1.x控制台启动失败，参考 Web3SDK启动失败场景

- 用控制台获取信息

```
# 获取客户端版本
[group:1]> getNodeVersion
ClientVersion{
    version='2.6.0',
    supportedVersion='2.6.0',
    chainId='1',
    buildTime='20200819 15:47:59',
    buildType='Darwin/appleclang/RelWithDebInfo',
    gitBranch='HEAD',
    gitCommitHash='e4a5ef2ef64d1943fccc4ebc61467a91779fb1c0'
}
# 获取节点信息
[group:1]> getPeers
[
    PeerInfo{
        nodeID=
↪'c1bd77e188cd0783256ee06838020f24a697f9af785438403d3620967a4a3612e3abc4bbe986d1e9dddf62d4236bff
↪',
        ipAndPort='127.0.0.1:30302',
        agency='agency',
        topic=[

        ],
        node='node2'
    },
    PeerInfo{
        nodeID=
↪'7f27f5d67f104eacf689790f09313e4343e7887a1a7b79c31cd151be33c7c8dd57c895a66086c3c8e0b54d2fa49340
↪',
        ipAndPort='127.0.0.1:57266',
        agency='agency',
        topic=[
            _block_notify_1
        ],
        node='node1'
    },
    PeerInfo{
        nodeID=
↪'862f26d9681ed4c12681bf81a50d0b8c66dd5b6ee7b0b42a4af12bb37b1ad2442f7dcfe8dac4e737ce9fa46aa94d90
↪',
        ipAndPort='127.0.0.1:30303',
        agency='agency',
        topic=[

        ],
        node='node3'
```

(continues on next page)

(续上页)

```
    }
]

[group:1]>
```

3.3 3. 部署及调用HelloWorld合约

3.3.1 第一步. 编写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() {
        name = "Hello, World!";
    }

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

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

3.3.2 第二步. 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
transaction hash:␣
↪0xd0305411e36d2ca9c1a4df93e761c820f0a464367b8feb9e3fa40b0f68eb23fa
contract address:0xb3c223fc0bf6646959f254ac4e4a7e355b50a344
```

3.3.3 第三步. 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
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    "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"
transaction hash:␣
↪0x7e742c44091e0d6e4e1df666d957d123116622ab90b718699ce50f54ed791f6e
-------------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully
-------------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Event logs
Event: {}

# 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
```

(continues on next page)

(续上页)

```
# 调用get接口获取name变量，检查设置是否生效
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 get
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,FISCO BCOS"
]
-------------------------------------------------------------------------------------------
↪----------

# 退出控制台
[group:1]> quit
```

CHAPTER 4

开发第一个区块链应用

标签：开发第一个应用 合约开发 区块链应用 教程

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Java SDK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。

如果您还未搭建区块链网络，或未下载控制台，请先走完教程搭建第一个区块链网络，再回到本教程。

4.1 1. 了解应用需求

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。本示例中，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. 设计与开发智能合约

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。

4.2.1 第一步. 设计智能合约

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)
```

4.2.2 第二步. 开发源码

根据我们第一步的存储和接口设计，创建一个Asset的智能合约，实现注册、转账、查询功能，并引入一个叫Table的系统合约，这个合约提供了CRUD接口。

```
# 进入console/contracts目录
cd ~/fisco/console/contracts/solidity
# 创建Asset.sol合约文件
vi Asset.sol

# 将Assert.sol合约内容写入。
# 并键入wq保存退出。
```

Asset.sol的内容如下：

```
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) {
```

(continues on next page)

(续上页)

```
        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
        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
```

(continues on next page)

(续上页)

```
            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
            -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;
```

(continues on next page)

(续上页)

```
        }

        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;
        }

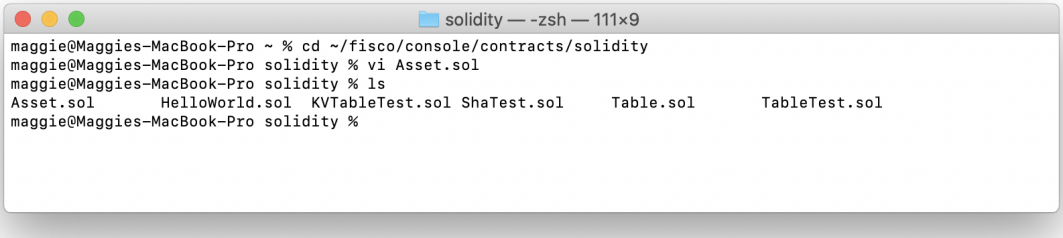
        Entry entry1 = table.newEntry();
        entry1.set("account", to_account);
        entry1.set("asset_value", int256(to_asset_value + amount));
        // 更新接收账户
        table.update(to_account, entry1, table.newCondition());

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

        return ret_code;
    }
}
```

Asset.sol所引用的Table.sol已在~/fisco/console/contracts/solidity目录下。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。

运行ls命令，确保Assert.sol和Table.sol在目录~/fisco/console/contracts/solidity下。



4.3 3. 编译智能合约

.sol的智能合约需要编译成ABI和BIN文件才能部署至区块链网络上。有了这两个文件即可凭借Java SDK进行合约部署和调用。但这种调用方式相对繁琐，需要用户根据合约ABI来传参和解析结果。为此，控制台提供的编译工具不仅可以编译出ABI和BIN文件，还可以自动生成一个与编译的智能合约同名的合约Java类。这个Java类是根据ABI生成的，帮助用户解析好了参数，提供同名的方法。当应用需要部署和调用合约时，可以调用该合约类的对应方法，传入指定参数即可。使用这个合约Java类来开发应用，可以极大简化用户的代码。

```
# 创建工作目录~/fisco
mkdir -p ~/fisco
# 下载控制台
```

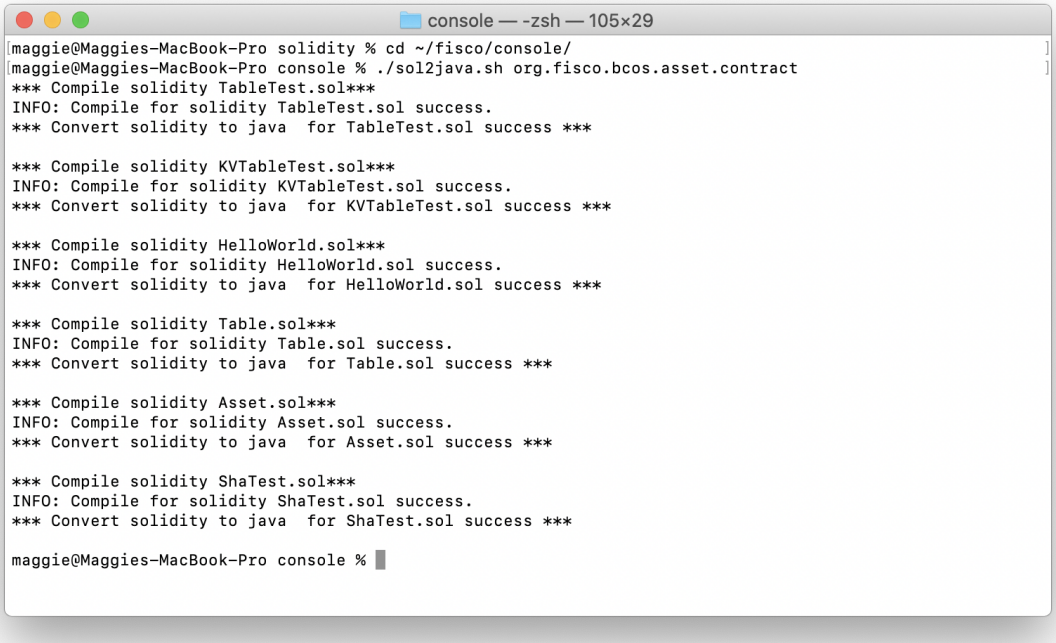
(continues on next page)

(续上页)

```
cd ~/fisco && curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.
↪7.1/download_console.sh && bash download_console.sh

# 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。

```
# 其它无关文件省略
|-- 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 TransactionReceipt transfer(String from_account, String to_account,␣
↪BigInteger amount);
    // Asset.sol合约 register接口生成
    public TransactionReceipt register(String account, BigInteger asset_value);
    // Asset.sol合约 select接口生成
    public Tuple2<BigInteger, BigInteger> select(String account) throws␣
↪ContractException;

    // 加载Asset合约地址，生成Asset对象
    public static Asset load(String contractAddress, Client client, CryptoKeyPair␣
↪credential);

    // 部署Assert.sol合约，生成Asset对象
    public static Asset deploy(Client client, CryptoKeyPair credential) throws␣
↪ContractException;
}
```

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。

4.4 4. 创建区块链应用项目

4.4.1 第一步. 安装环境

首先，我们需要安装JDK以及集成开发环境

- Java：JDK 14 （JDK1.8 至JDK 14都支持）

  首先，在官网上下载JDK14并安装

  然后，修改环境变量

```
# 确认您当前的java版本
$ java -version
# 确认您的java路径
$ ls Library/Java/JavaVirtualMachines
# 返回
# jdk-14.0.2.jdk

# 如果使用的是bash
$ vim .bash_profile
# 在文件中加入JAVA_HOME的路径
# export JAVA_HOME = Library/Java/JavaVirtualMachines/jdk-14.0.2.jdk/Contents/
↪Home
$ source .bash_profile

# 如果使用的是zash
$ vim .zashrc
# 在文件中加入JAVA_HOME的路径
# export JAVA_HOME = Library/Java/JavaVirtualMachines/jdk-14.0.2.jdk/Contents/
↪Home
$ source .zashrc

# 确认您的java版本
$ java -version
# 返回
# java version "14.0.2" 2020-07-14
```

(continues on next page)

(续上页)

```
# Java(TM) SE Runtime Environment (build 14.0.2+12-46)
# Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing)
```

- IDE：IntelliJ IDE.

  进入IntelliJ IDE官网，下载并安装社区版IntelliJ IDE

4.4.2 第二步. 创建一个Java工程

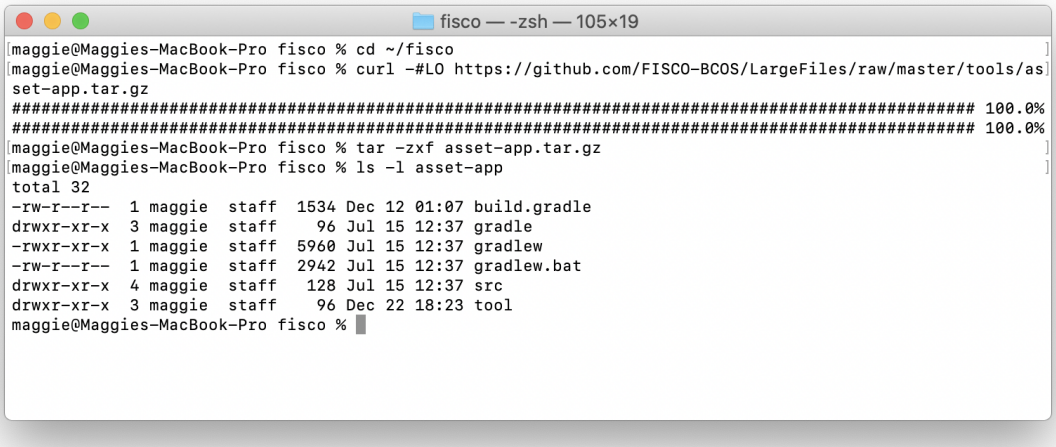
在IntelliJ IDE中创建一个gradle项目，勾选Gradle和Java，并输入工程名`assert-app`。

注意：该项目的源码可以用以下方法获得并参考。（此步骤为非必须步骤）

```
$ cd ~/fisco
$ 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
$ 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，请尝试将‘199.232.28.133 raw.githubusercontent.com‘追加到‘/etc/hosts‘中，或者请尝试 *curl -#LO https://osp-1257653870.cos.ap-guangzhou.myqcloud.com/FISCO-BCOS/FISCO-BCOS/tools/asset-app.tar.gz*



4.4.3 第三步. 引入FISCO BCOS Java SDK

在build.gradle文件中的`dependencies`下加入对FISCO BCOS Java SDK的引用。

```
repositories {
    mavenCentral()
    maven {
        url "http://maven.aliyun.com/nexus/content/groups/public/"
```

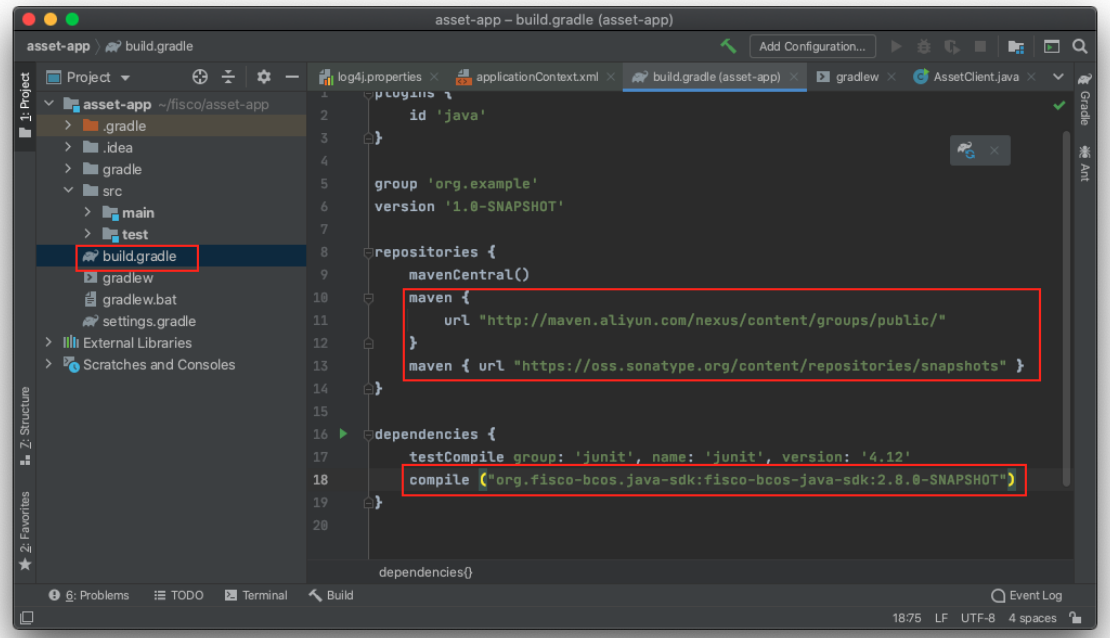
(continues on next page)

(续上页)

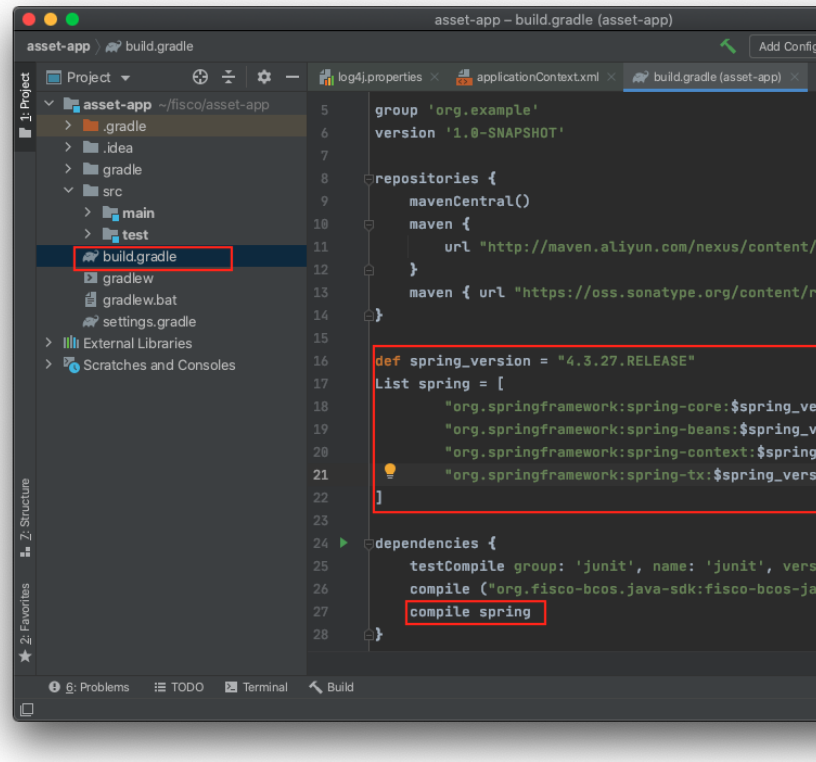
```
    }
    maven { url "https://oss.sonatype.org/content/repositories/snapshots" }
}

-   引入Java SDK jar包

```java
testCompile group: 'junit', name: 'junit', version: '4.12'
compile ('org.fisco-bcos.java-sdk:fisco-bcos-java-sdk:2.7.1')
```


```

4.4.4 第四步. 配置SDK证书

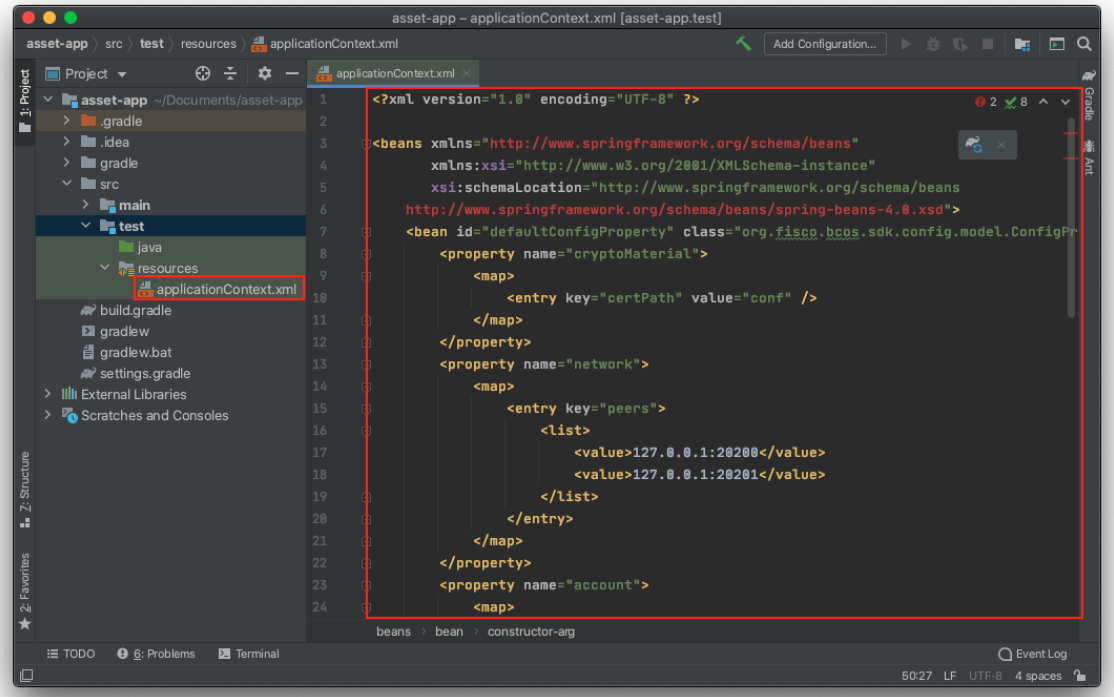


修改build.gradle文件，引入Spring框架。

```
def spring_version = "4.3.27.RELEASE"
List spring = [
        "org.springframework:spring-core:$spring_version",
        "org.springframework:spring-beans:$spring_version",
        "org.springframework:spring-context:$spring_version",
        "org.springframework:spring-tx:$spring_version",
]

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    compile ("org.fisco-bcos.java-sdk:fisco-bcos-java-sdk:2.8.0-SNAPSHOT")
    compile spring
}
```

在asset-app/test/resources目录下创建配置文件applicationContext.xml，写入配置内容。各配置项的内容可参考Java SDK配置说明，该配置说明以toml配置文件为例，本例中的配置项与该配置项相对应。



applicationContext.xml的内容如下：

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">
    <bean id="defaultConfigProperty" class="org.fisco.bcos.sdk.config.model.
↪ConfigProperty">
        <property name="cryptoMaterial">
            <map>
                <entry key="certPath" value="conf" />
            </map>
        </property>
        <property name="network">
            <map>
                <entry key="peers">
                    <list>
                        <value>127.0.0.1:20200</value>
                        <value>127.0.0.1:20201</value>
                    </list>
                </entry>
            </map>
        </property>
        <property name="account">
            <map>
                <entry key="keyStoreDir" value="account" />
                <entry key="accountAddress" value="" />
                <entry key="accountFileFormat" value="pem" />
                <entry key="password" value="" />
                <entry key="accountFilePath" value="" />
            </map>
        </property>
        <property name="threadPool">
```

(continues on next page)

(续上页)

```
            <map>
                <entry key="channelProcessorThreadSize" value="16" />
                <entry key="receiptProcessorThreadSize" value="16" />
                <entry key="maxBlockingQueueSize" value="102400" />
            </map>
        </property>
    </bean>

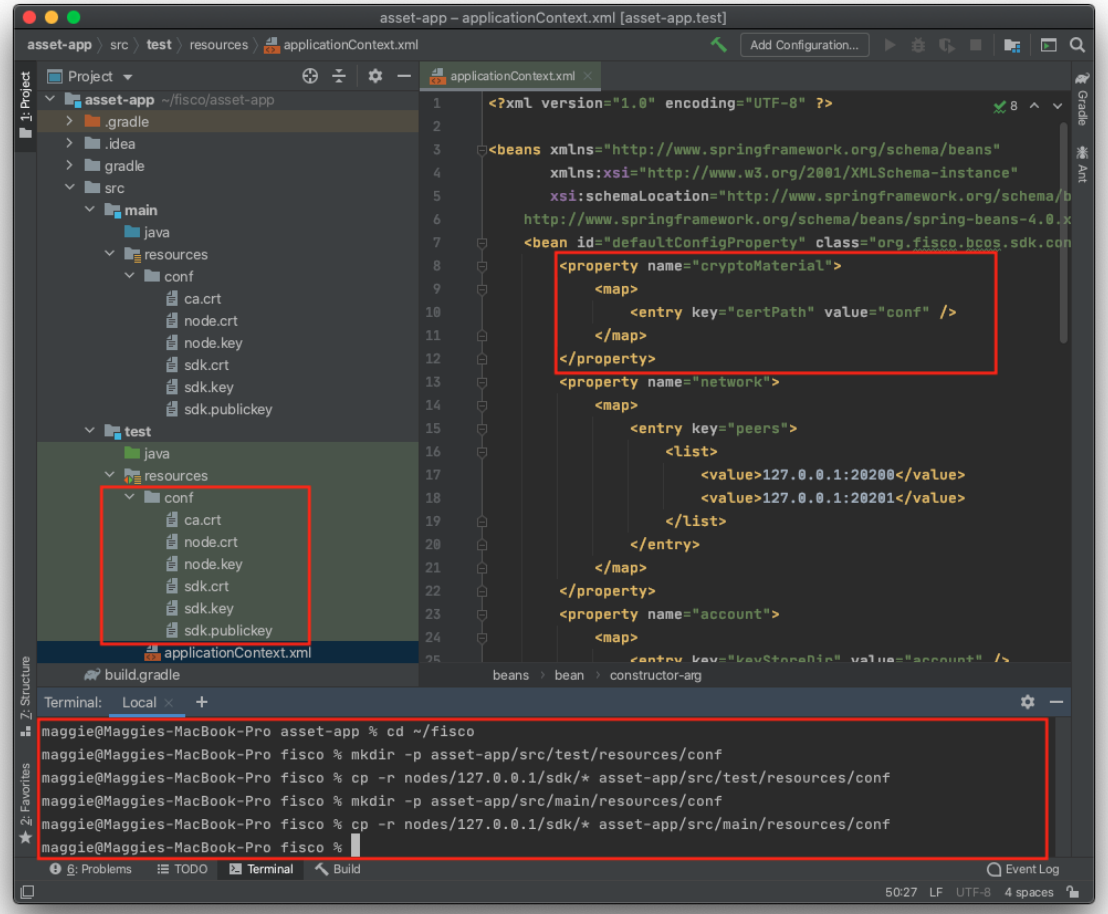
    <bean id="defaultConfigOption" class="org.fisco.bcos.sdk.config.ConfigOption">
        <constructor-arg name="configProperty">
            <ref bean="defaultConfigProperty"/>
        </constructor-arg>
    </bean>

    <bean id="bcosSDK" class="org.fisco.bcos.sdk.BcosSDK">
        <constructor-arg name="configOption">
            <ref bean="defaultConfigOption"/>
        </constructor-arg>
    </bean>
</beans>
```

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的network属性下的peers配置选项，配置所连接节点的IP:channel_listen_port。

在以上配置文件中，我们指定了证书存放的位certPath的值为conf。接下来我们需要把SDK用于连接节点的证书放到指定的conf目录下。

```
# 假设我们将asset-app放在~/fisco目录下 进入~/fisco目录
$ cd ~/fisco
# 创建放置证书的文件夹
$ mkdir -p asset-app/src/test/resources/conf
# 拷贝节点证书到项目的资源目录
$ cp -r nodes/127.0.0.1/sdk/* asset-app/src/test/resources/conf
# 若在IDE直接运行，拷贝证书到resources路径
$ mkdir -p asset-app/src/main/resources/conf
$ cp -r nodes/127.0.0.1/sdk/* asset-app/src/main/resources/conf
```

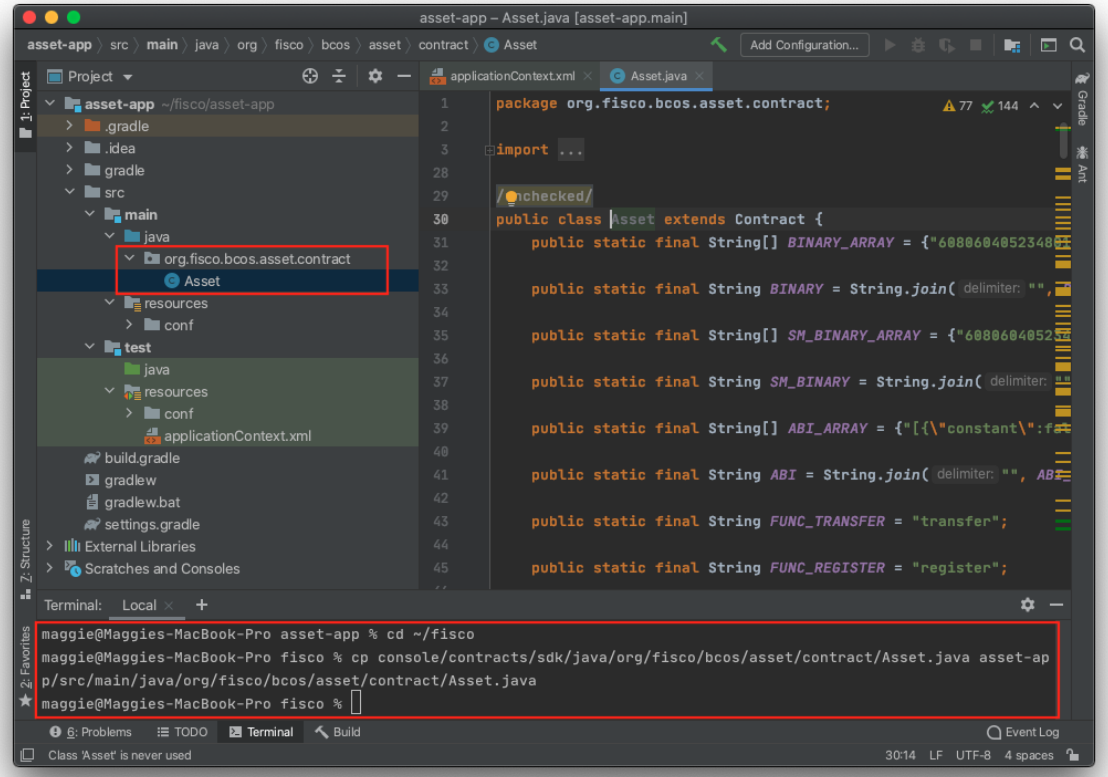


4.5 5. 业务逻辑开发

我们已经介绍了如何在自己的项目中引入以及配置Java SDK，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。

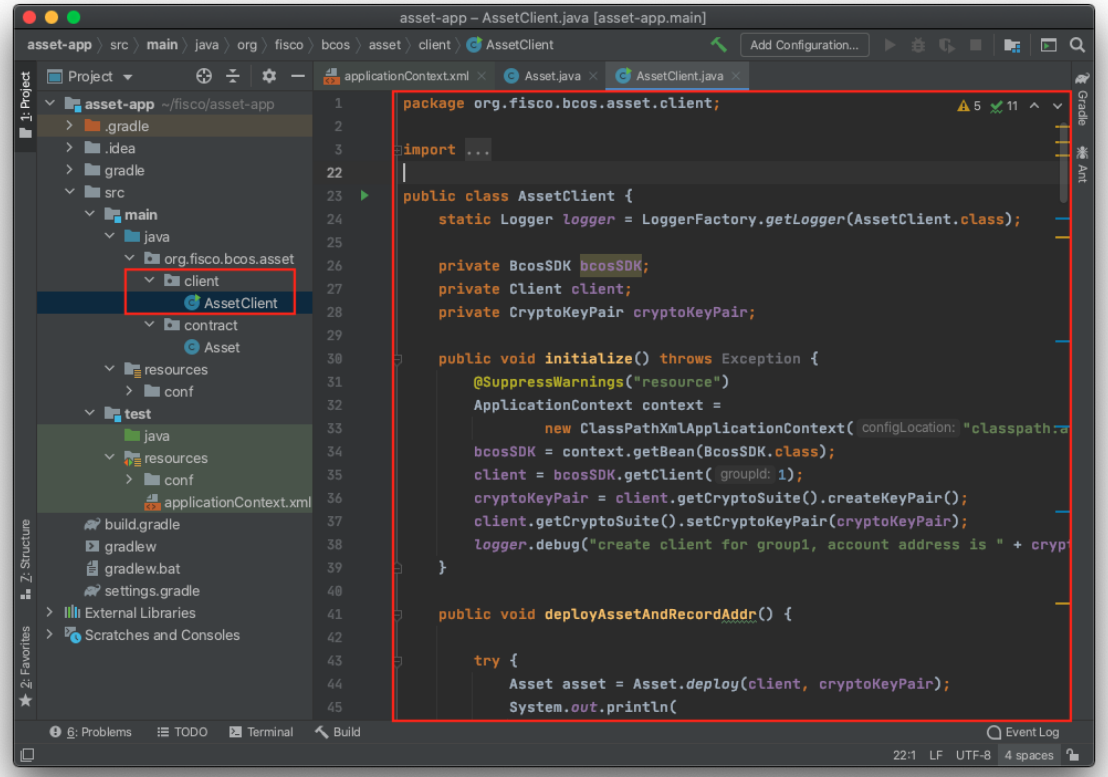
4.5.1 第一步.将3编译好的Java合约引入项目中

```
cd ~/fisco
# 将编译好的合约Java类引入项目中。
cp console/contracts/sdk/java/org/fisco/bcos/asset/contract/Asset.java asset-app/
↪src/main/java/org/fisco/bcos/asset/contract/Asset.java
```



4.5.2 第二步.开发业务逻辑

在路径/src/main/java/org/fisco/bcos/asset/client目录下，创建AssetClient.java类，通过调用Asset.java实现对合约的部署与调用



AssetClient.java 代码如下：

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

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.math.BigInteger;
import java.util.List;
import java.util.Properties;
import org.fisco.bcos.asset.contract.Asset;
import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.abi.datatypes.generated.tuples.generated.Tuple2;
import org.fisco.bcos.sdk.client.Client;
import org.fisco.bcos.sdk.crypto.keypair.CryptoKeyPair;
import org.fisco.bcos.sdk.model.TransactionReceipt;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;

public class AssetClient {
    static Logger logger = LoggerFactory.getLogger(AssetClient.class);

    private BcosSDK bcosSDK;
    private Client client;
    private CryptoKeyPair cryptoKeyPair;

    public void initialize() throws Exception {
```

(continues on next page)

(续上页)

```
        @SuppressWarnings("resource")
        ApplicationContext context =
                new ClassPathXmlApplicationContext("classpath:applicationContext.
↪xml");
        bcosSDK = context.getBean(BcosSDK.class);
        client = bcosSDK.getClient(1);
        cryptoKeyPair = client.getCryptoSuite().createKeyPair();
        client.getCryptoSuite().setCryptoKeyPair(cryptoKeyPair);
        logger.debug("create client for group1, account address is " +␣
↪cryptoKeyPair.getAddress());
    }

    public void deployAssetAndRecordAddr() {

        try {
            Asset asset = Asset.deploy(client, cryptoKeyPair);
            System.out.println(
                    " deploy Asset success, contract address is " + asset.
↪getContractAddress());

            recordAssetAddr(asset.getContractAddress());
        } catch (Exception e) {
            // TODO Auto-generated catch block
            // e.printStackTrace();
            System.out.println(" deploy Asset contract failed, error message is  "␣
↪+ e.getMessage());
        }
    }

    public void recordAssetAddr(String address) throws FileNotFoundException,␣
↪IOException {
        Properties prop = new Properties();
        prop.setProperty("address", address);
        final Resource contractResource = new ClassPathResource("contract.
↪properties");
        FileOutputStream fileOutputStream = new FileOutputStream(contractResource.
↪getFile());
        prop.store(fileOutputStream, "contract address");
    }

    public String loadAssetAddr() throws Exception {
        // load Asset contact address from contract.properties
        Properties prop = new Properties();
        final Resource contractResource = new ClassPathResource("contract.
↪properties");
        prop.load(contractResource.getInputStream());

        String contractAddress = prop.getProperty("address");
        if (contractAddress == null || contractAddress.trim().equals("")) {
            throw new Exception(" load Asset contract address failed, please␣
↪deploy it first. ");
        }
        logger.info(" load Asset address from contract.properties, address is {}",␣
↪contractAddress);
        return contractAddress;
    }

    public void queryAssetAmount(String assetAccount) {
        try {
            String contractAddress = loadAssetAddr();
            Asset asset = Asset.load(contractAddress, client, cryptoKeyPair);
```

(continues on next page)

(续上页)

```
            Tuple2<BigInteger, BigInteger> result = asset.select(assetAccount);
            if (result.getValue1().compareTo(new BigInteger("0")) == 0) {
                System.out.printf(" asset account %s, value %s \n", assetAccount,␣
↪result.getValue2());
            } else {
                System.out.printf(" %s asset account is not exist \n",␣
↪assetAccount);
            }
        } catch (Exception e) {
            // TODO Auto-generated catch block
            // e.printStackTrace();
            logger.error(" queryAssetAmount exception, error message is {}", e.
↪getMessage());

            System.out.printf(" query asset account failed, error message is %s\n",
↪ e.getMessage());
        }
    }

    public void registerAssetAccount(String assetAccount, BigInteger amount) {
        try {
            String contractAddress = loadAssetAddr();

            Asset asset = Asset.load(contractAddress, client, cryptoKeyPair);
            TransactionReceipt receipt = asset.register(assetAccount, amount);
            List<Asset.RegisterEventEventResponse> response = asset.
↪getRegisterEventEvents(receipt);
            if (!response.isEmpty()) {
                if (response.get(0).ret.compareTo(new BigInteger("0")) == 0) {
                    System.out.printf(
                            " register asset account success => asset: %s, value:
↪%s \n", assetAccount, amount);
                } else {
                    System.out.printf(
                            " register asset account failed, ret code is %s \n",␣
↪response.get(0).ret.toString());
                }
            } else {
                System.out.println(" event log not found, maybe transaction not␣
↪exec. ");
            }
        } catch (Exception e) {
            // TODO Auto-generated catch block
            // e.printStackTrace();

            logger.error(" registerAssetAccount exception, error message is {}", e.
↪getMessage());
            System.out.printf(" register asset account failed, error message is
↪%s\n", e.getMessage());
        }
    }

    public void transferAsset(String fromAssetAccount, String toAssetAccount,␣
↪BigInteger amount) {
        try {
            String contractAddress = loadAssetAddr();
            Asset asset = Asset.load(contractAddress, client, cryptoKeyPair);
            TransactionReceipt receipt = asset.transfer(fromAssetAccount,␣
↪toAssetAccount, amount);
            List<Asset.TransferEventEventResponse> response = asset.
↪getTransferEventEvents(receipt);
```

(continues on next page)

(续上页)

```
            if (!response.isEmpty()) {
                if (response.get(0).ret.compareTo(new BigInteger("0")) == 0) {
                    System.out.printf(
                            " transfer success => from_asset: %s, to_asset: %s,␣
↪amount: %s \n",
                            fromAssetAccount, toAssetAccount, amount);
                } else {
                    System.out.printf(
                            " transfer asset account failed, ret code is %s \n",␣
↪response.get(0).ret.toString());
                }
            } else {
                System.out.println(" event log not found, maybe transaction not␣
↪exec. ");
            }
        } catch (Exception e) {
            // TODO Auto-generated catch block
            // e.printStackTrace();

            logger.error(" registerAssetAccount exception, error message is {}", e.
↪getMessage());
            System.out.printf(" register asset account failed, error message is
↪%s\n", e.getMessage());
        }
    }

    public static void Usage() {
        System.out.println(" Usage:");
        System.out.println(
                "\t java -cp conf/:lib/*:apps/* org.fisco.bcos.asset.client.
↪AssetClient deploy");
        System.out.println(
                "\t java -cp conf/:lib/*:apps/* org.fisco.bcos.asset.client.
↪AssetClient query account");
        System.out.println(
                "\t java -cp conf/:lib/*:apps/* org.fisco.bcos.asset.client.
↪AssetClient register account value");
        System.out.println(
                "\t java -cp conf/:lib/*:apps/* org.fisco.bcos.asset.client.
↪AssetClient transfer from_account to_account amount");
        System.exit(0);
    }

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            Usage();
        }

        AssetClient client = new AssetClient();
        client.initialize();

        switch (args[0]) {
            case "deploy":
                client.deployAssetAndRecordAddr();
                break;
            case "query":
                if (args.length < 2) {
                    Usage();
                }
                client.queryAssetAmount(args[1]);
                break;
```

(continues on next page)

(续上页)

```
            case "register":
                if (args.length < 3) {
                    Usage();
                }
                client.registerAssetAccount(args[1], new BigInteger(args[2]));
                break;
            case "transfer":
                if (args.length < 4) {
                    Usage();
                }
                client.transferAsset(args[1], args[2], new BigInteger(args[3]));
                break;
            default:
            {
                Usage();
            }
        }
        System.exit(0);
    }

}
```

让我们通过AssetClient这个例子，来了解FISCO BCOS Java SDK的调用：

- 初始化

初始化代码的主要功能为构造Client与CryptoKeyPair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
// 初始化BcosSDK
@SuppressWarnings("resource")
ApplicationContext context =
        new ClassPathXmlApplicationContext("classpath:applicationContext.xml");
bcosSDK = context.getBean(BcosSDK.class);
// 初始化可向群组1发交易的Client
client = bcosSDK.getClient(1);
// 随机生成发送交易的公私钥对
cryptoKeyPair = client.getCryptoSuite().createKeyPair();
client.getCryptoSuite().setCryptoKeyPair(cryptoKeyPair);
logger.debug("create client for group1, account address is " + cryptoKeyPair.
↪getAddress());
```

- 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(client, cryptoKeyPair);
// 加载合约地址
Asset asset = Asset.load(contractAddress, client, cryptoKeyPair);
```

- 接口调用

使用合约对象调用对应的接口，处理返回结果。

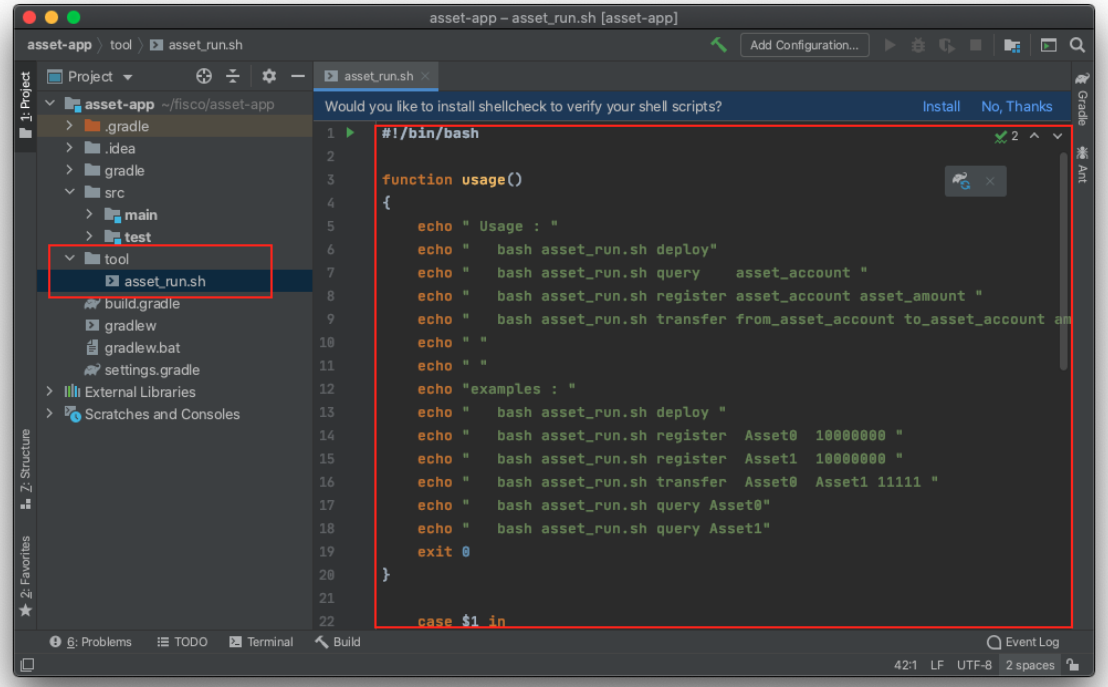
```
// select接口调用
 Tuple2<BigInteger, BigInteger> result = asset.select(assetAccount);
// register接口调用
TransactionReceipt receipt = asset.register(assetAccount, amount);
```

(continues on next page)

(续上页)

```
// transfer接口
TransactionReceipt receipt = asset.transfer(fromAssetAccount, toAssetAccount,␣
↪amount);
```

在asset-app/tools目录下添加一个调用AssetClient的脚本asset_run.sh。




```
#!/bin/bash

function usage()
{
    echo " Usage : "
    echo "   bash asset_run.sh deploy"
    echo "   bash asset_run.sh query    asset_account "
    echo "   bash asset_run.sh register asset_account asset_amount "
    echo "   bash asset_run.sh transfer from_asset_account to_asset_account amount
↪"
    echo " "
    echo " "
    echo "examples : "
    echo "   bash asset_run.sh deploy "
    echo "   bash asset_run.sh register  Asset0  10000000 "
    echo "   bash asset_run.sh register  Asset1  10000000 "
    echo "   bash asset_run.sh transfer  Asset0  Asset1 11111 "
    echo "   bash asset_run.sh query Asset0"
    echo "   bash asset_run.sh query Asset1"
    exit 0
}

    case $1 in
    deploy)
            [ $# -lt 1 ] && { usage; }
            ;;
    register)
            [ $# -lt 3 ] && { usage; }
```

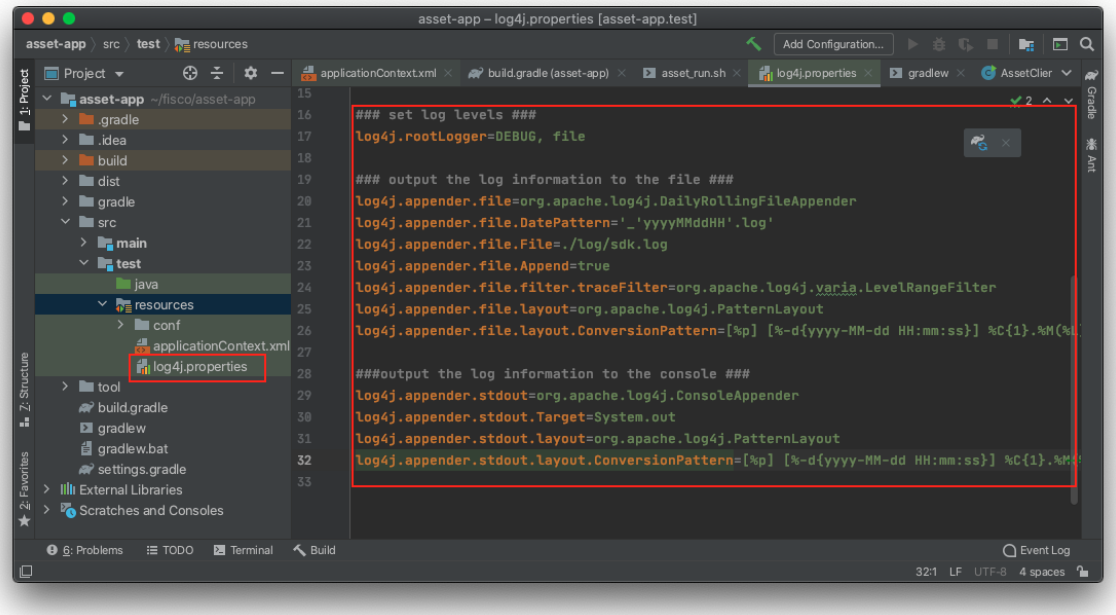
(continues on next page)

(续上页)

```
        ;;
    transfer)
        [ $# -lt 4 ] && { usage; }
        ;;
    query)
        [ $# -lt 2 ] && { usage; }
        ;;
    *)
        usage
        ;;
    esac

    java -Djdk.tls.namedGroups="secp256k1" -cp 'apps/*:conf/:lib/*' org.fisco.bcos.
↪asset.client.AssetClient $@
```

接着，配置好log。在asset-app/test/resources目录下创建log4j.properties



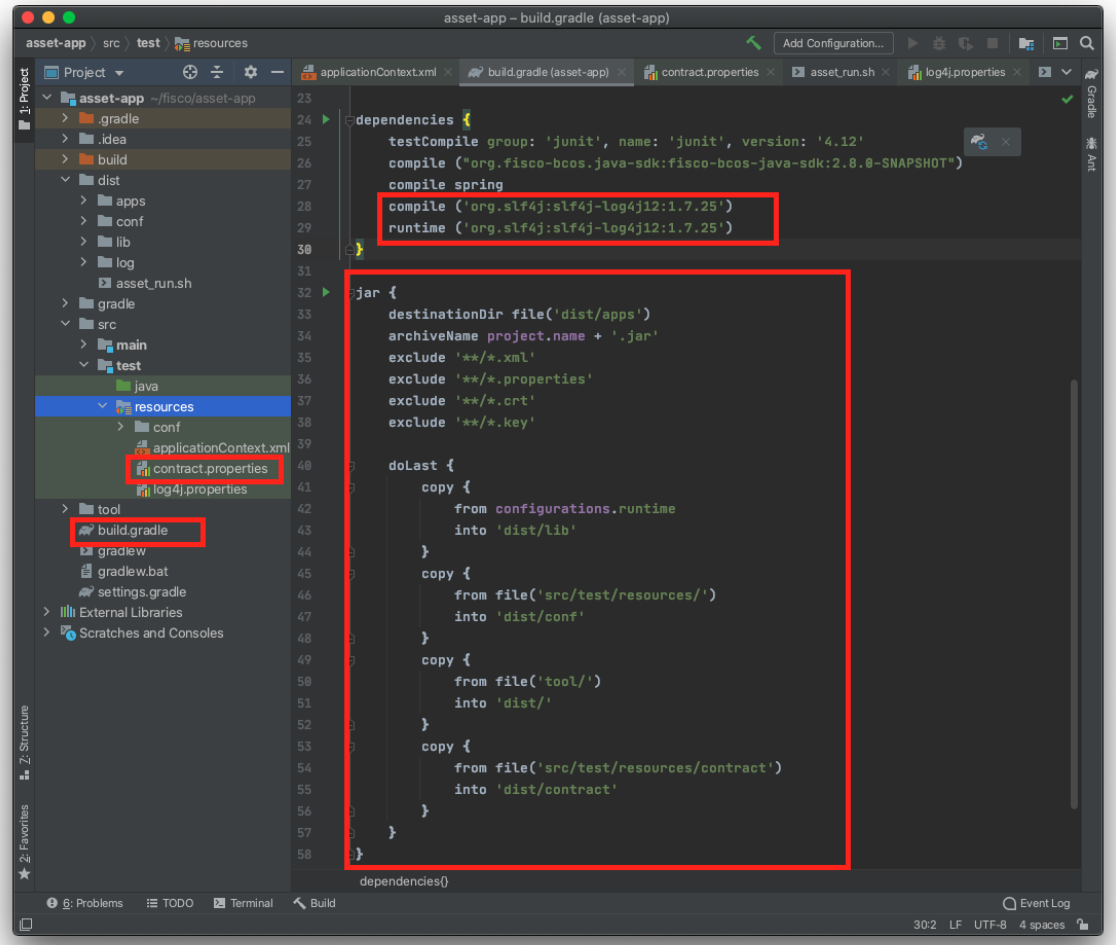

```
### set log levels ###
log4j.rootLogger=DEBUG, file

### output the log information to the file ###
log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
log4j.appender.file.DatePattern='_'yyyyMMddHH'.log'
log4j.appender.file.File=./log/sdk.log
log4j.appender.file.Append=true
log4j.appender.file.filter.traceFilter=org.apache.log4j.varia.LevelRangeFilter
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p] [%-d{yyyy-MM-dd HH:mm:ss}] %C{1}.
↪%M(%L) | %m%n

###output the log information to the console ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%p] [%-d{yyyy-MM-dd HH:mm:ss}] %C
↪{1}.%M(%L) | %m%n
```

接着，通过配置gradle中的Jar命令，指定复制和编译任务。并引入日志库，在asset-app/test/

resources目录下，创建一个空的contract.properties文件，用于应用在运行时存放合约地址。




```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    compile ("org.fisco-bcos.java-sdk:fisco-bcos-java-sdk:2.8.0-SNAPSHOT")
    compile spring
    compile ('org.slf4j:slf4j-log4j12:1.7.25')
    runtime ('org.slf4j:slf4j-log4j12:1.7.25')
}
jar {
    destinationDir file('dist/apps')
    archiveName project.name + '.jar'
    exclude '**/*.xml'
    exclude '**/*.properties'
    exclude '**/*.crt'
    exclude '**/*.key'

    doLast {
        copy {
            from configurations.runtime
            into 'dist/lib'
        }
        copy {
            from file('src/test/resources/')
            into 'dist/conf'
        }
        copy {
```

(continues on next page)

(续上页)

```
            from file('tool/')
            into 'dist/'
        }
        copy {
            from file('src/test/resources/contract')
            into 'dist/contract'
        }
    }
}
```

至此，我们已经完成了这个应用的开发。最后，我们得到的assert-app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
|   |   |-- resources
|   |        |-- conf
|   |               |-- ca.crt
|   |               |-- node.crt
|   |               |-- node.key
|   |               |-- sdk.crt
|   |               |-- sdk.key
|   |               |-- sdk.publickey
|   |        |-- 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
|   |        |-- contract //存放solidity约文件
|   |                |-- Asset.sol
|   |                |-- Table.sol
|   |-- test
|       |-- resources // 存放代码资源文件
|           |-- conf
|                  |-- ca.crt
|                  |-- node.crt
|                  |-- node.key
|                  |-- sdk.crt
|                  |-- sdk.key
|                  |-- sdk.publickey
|           |-- 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
|           |-- contract //存放solidity约文件
|                   |-- Asset.sol
|                   |-- Table.sol
|
|-- tool
```

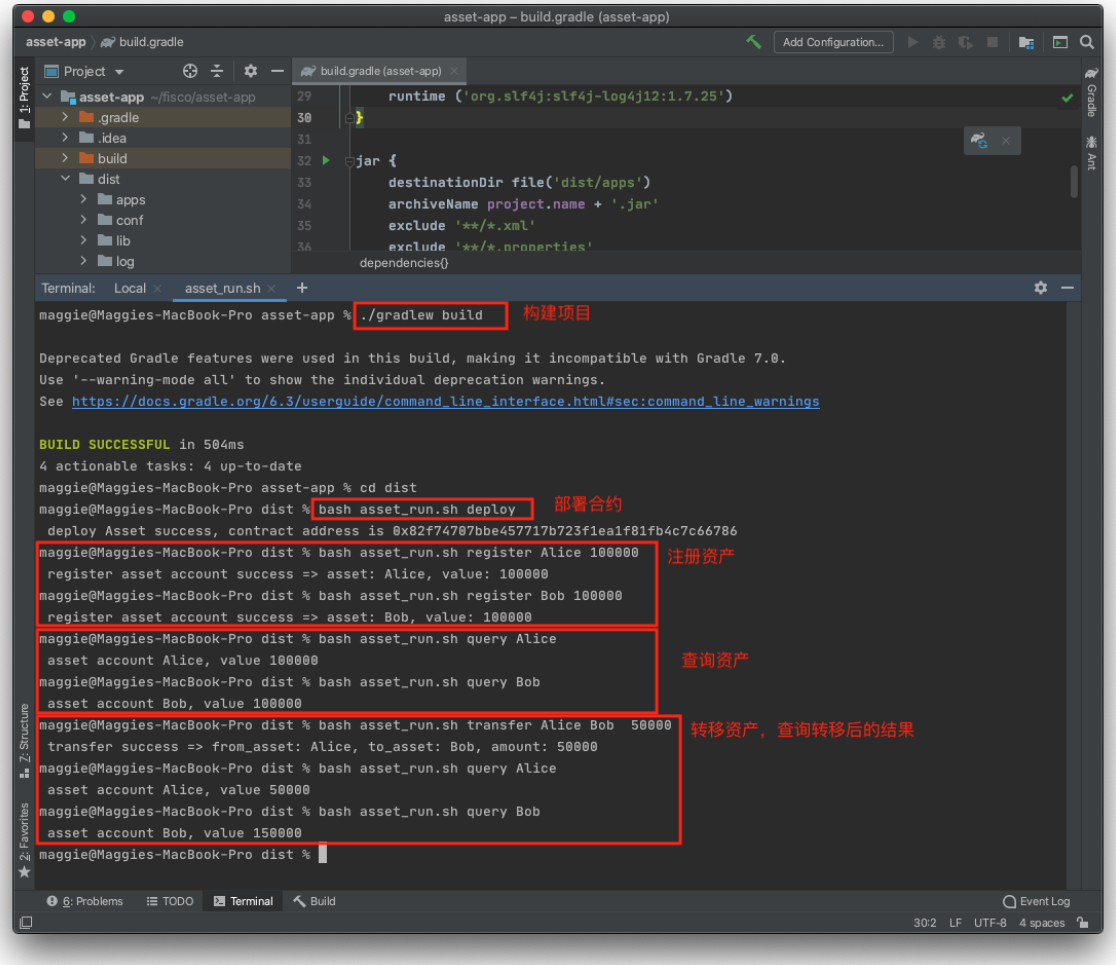
(continues on next page)

(续上页)

```
    |-- asset_run.sh // 项目运行脚本
```

4.6 6. 运行应用

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 ~/fisco/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
```

(continues on next page)

(续上页)

- 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

高阶教程

标签：教程

5.1 搭建多机区块链网络

标签：搭建多机区块链网络 区块链教程 `HelloWorld` 控制台调用合约

搭建第一个区块链网络章节详细介绍了如何搭建单机4节点区块链网络，本章以搭建多机4节点区块链网络为例，详细介绍如何在多台机器上部署FISCO BCOS。

5.1.1 1. 搭建多机4节点区块链网络

本节详细介绍如何基于开发部署工具build_chain搭建多机4节点区块链系统，生产环境建议使用运维部署工具进行区块链搭建和部署操作。

本教程，假设四台物理机器的IP分别为`196.168.0.1`, `196.168.0.3`, `196.168.0.4`和`196.168.0.2`，每台机器部署一个区块链节点。

注解:

- 请确保每台机器的 `30300`, `20200`, `8545` 端口没有被占用。
- 请确保每台机器开通了 `30300`, `20200` 和 `22` 端口的网络访问权限
- 请确保生成区块链节点配置的机器可以访问外网(用于下载建链脚本)

第一步. 生成区块链节点配置

类似于单机区块链节点的部署，部署多机区块链系统之前，首先要生成节点配置，这里假设统一在IP为`196.168.0.1`机器上生成节点配置，具体操作流程如下：

创建操作路径，下载开发部署工具build_chain

```
# 创建操作路径~/fisco
mkdir -p ~/fisco && cd ~/fisco

# 下载开发部署工具build_chain
curl -#LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.7.1/build_
↪chain.sh && chmod u+x build_chain.sh
```

生成区块链网络配置文件

```
# 这里所有区块链节点均属于agencyA，并仅启动了群组1
# (注：下面的ip信息需要根据真实的机器IP填写)
cat >> ipconf << EOF
196.168.0.1 agencyA 1
196.168.0.3 agencyA 1
196.168.0.4 agencyA 1
196.168.0.2 agencyA 1
EOF
```

基于配置文件生成区块链节点配置

```
bash build_chain.sh -f ipconf -p 30300,20200,8545
```

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 and certificates ...
Processing IP=196.168.0.1 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.3 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.4 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.2 Total=1 Agency=agencyA Groups=1
============================================================
Generating configuration files ...
Processing IP=196.168.0.1 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.3 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.4 Total=1 Agency=agencyA Groups=1
Processing IP=196.168.0.2 Total=1 Agency=agencyA Groups=1
============================================================
Group:1 has 4 nodes
============================================================
[INFO] FISCO-BCOS Path : bin/fisco-bcos
[INFO] IP List File    : ipconf
[INFO] Start Port      : 30300 20200 8545
[INFO] Server IP       : 196.168.0.1 196.168.0.3 196.168.0.4 196.168.0.2
[INFO] Output Dir      : /home/ubuntu/fisco/nodes
[INFO] CA Path         : /home/ubuntu/nodes/cert/
============================================================
[INFO] Execute the download_console.sh script in directory named by IP to get␣
↪FISCO-BCOS console.
e.g.  bash /home/ubuntu/fisco/nodes/196.168.0.1/download_console.sh -f
============================================================
[INFO] All completed. Files in /root/fisco/nodes
```

至此，成功生成了多机4节点配置，每台机器的区块链节点配置均位于nodes文件夹下，如下：

```
$ ls nodes/
196.168.0.1  196.168.0.2  196.168.0.3  196.168.0.4  cert
```

第二步. 拷贝区块链节点配置

生成区块链节点配置后，需要将每个节点配置拷贝到对应机器上，可通过scp命令执行拷贝，具体如下：

```
# 为每台机器创建操作目录~/fisco
# (注：这里使用fisco用户进行操作；实际操作时，可使用自己的账户进行类似操作，IP也需要替换成自己的机器IP)
ssh fisco@196.168.0.1 "mkdir -p ~/fisco"
ssh fisco@196.168.0.2 "mkdir -p ~/fisco"
ssh fisco@196.168.0.3 "mkdir -p ~/fisco"
ssh fisco@196.168.0.4 "mkdir -p ~/fisco"

# 拷贝节点配置
# 拷贝节点配置到196.168.0.1的~/fisco路径
scp -r nodes/196.168.0.1/ fisco@196.168.0.1:~/fisco/196.168.0.1
# 拷贝节点配置到196.168.0.2的~/fisco路径
scp -r nodes/196.168.0.2/ fisco@196.168.0.2:~/fisco/196.168.0.2
# 拷贝节点配置到196.168.0.3的~/fisco路径
scp -r nodes/196.168.0.3/ fisco@196.168.0.3:~/fisco/196.168.0.3
# 拷贝节点配置到196.168.0.4的~/fisco路径
scp -r nodes/196.168.0.4/ fisco@196.168.0.4:~/fisco/196.168.0.4
```

第三步. 启动多机4节点区块链系统

区块链节点配置拷贝成功后，需要启动所有节点，可通过某台机器发起ssh操作远程启动区块链节点，也可登录上所有物理机后，在对应的物理机上启动区块链节点。

方法一: 远程启动区块链节点

这里同样从196.168.0.1发起节点启动命令，具体如下：

```
# (注：这里使用fisco用户进行操作；实际操作时，可使用自己的账户进行类似操作，IP也需要替换成自己的机器IP)
# 启动196.168.0.1机器上部署的区块链节点
$ ssh fisco@196.168.0.1 "bash ~/fisco/196.168.0.1/start_all.sh"
try to start node0
 node0 start successfully

# 启动196.168.0.2机器上部署的区块链节点
$ ssh fisco@196.168.0.2 "bash ~/fisco/196.168.0.2/start_all.sh"
try to start node0
 node0 start successfully

# 启动196.168.0.3机器上部署的区块链节点
$ ssh fisco@196.168.0.3 "bash ~/fisco/196.168.0.3/start_all.sh"
try to start node0
 node0 start successfully

# 启动196.168.0.4机器上部署的区块链节点
$ ssh fisco@196.168.0.4 "bash ~/fisco/196.168.0.4/start_all.sh"
try to start node0
 node0 start successfully
```

方法二: 直接登录机器启动区块链节点

```
# (注：这里使用fisco用户进行操作；实际操作时，可使用自己的账户进行类似操作，IP也需要替换成自己的机器IP)
# 登录196.168.0.1并启动区块链节点
$ ssh fisco@196.168.0.1
$ bash ~/fisco/196.168.0.1/start_all.sh
```

(continues on next page)

(续上页)

```
# 登录196.168.0.2并启动区块链节点
$ ssh fisco@196.168.0.2
$ bash ~/fisco/196.168.0.2/start_all.sh

# 登录196.168.0.3并启动区块链节点
$ ssh fisco@196.168.0.3
$ bash ~/fisco/196.168.0.3/start_all.sh

# 登录196.168.0.4并启动区块链节点
$ ssh fisco@196.168.0.4
$ bash ~/fisco/196.168.0.4/start_all.sh
```

至此，已经搭建了一条多机4节点区块链系统，接下来需要检查区块链节点是否正常工作。

第四步. 检查区块链节点

检查进程是否启动成功

登录每台机器，执行如下命令判断进程是否启动成功：

```
ps aux | grep fisco | grep -v grep
```

正常情况，每台机器都会有类似下面的输出；

```
fisco     29306  0.8  0.1 747008 31488 ?        Sl   17:08   0:05 /home/ubuntu/
↪fisco/196.168.0.1/node0/../fisco-bcos -c config.ini
```

若某些机器没有类似以上的输出，请检查机器的30300, 20200, 8545端口是否被占用。

检查网络连接是否正常

登录每台机器，执行如下命令判断节点网络连接是否正常：

```
tail -f ~/fisco/*/node0/log/* |grep -i connected
```

正常情况会不停地输出连接信息，从输出可以看出该节点与其他机器节点连接正常。

```
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
```

检查区块链共识是否正常

登录每台机器，进入操作目录，执行如下命令判断节点共识是否正常：

```
tail -f ~/fisco/*/node0/log/* |grep -i +++
```

正常情况会不停输出++++Generating seal，表示共识正常。

```
info|2020-12-22 17:24:43.729402|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=2e133146...
info|2020-12-22 17:24:47.740603|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=eb199760...
```

若以上检查均正常，说明多机4节点区块链系统部署成功，可通过控制台对其发起交易。

5.1.2 2. 配置和使用控制台

本章介绍如何为多机4节点区块链系统配置控制台，并使用控制台对多机区块链系统发起交易。

第一步. 准备依赖

- 安装java （推荐使用java 14）.

```
# ubuntu系统安装java
sudo apt install -y default-jdk

#centos系统安装java
sudo yum install -y java java-devel
```

第二步. 下载并配置控制台

下载控制台

```
# 创建操作目录
mkdir -p ~/fisco && cd ~/fisco

curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/download_
↪console.sh && bash download_console.sh
```

配置控制台: 拷贝SDK证书

这里选择从196.168.0.1拷贝SDK证书，具体操作如下:

```
# 从196.168.0.1拷贝证书到conf目录下
$ scp 196.168.0.1:~/fisco/196.168.0.1/sdk/* ~/fisco/console/conf
```

配置控制台: 修改控制台配置

```
# 拷贝控制台配置
$ cp -n ~/fisco/console/conf/config-example.toml ~/fisco/console/conf/config.toml

# 修改控制台连接信息(操作中，控制台连接的IP信息请根据实际情况填写)
sed -i 's/peers=\["127.0.0.1:20200", "127.0.0.1:20201"\]/peers=["196.168.0.1:20200
↪", "196.168.0.2:20200", "196.168.0.3:20200", "196.168.0.4:20200"]/g' ~/fisco/
↪console/conf/config.toml
```

第三步. 启动并使用控制台

启动控制台

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

控制台启动成功后，会输出如下信息:

```
=====================================================================================================
Welcome to FISCO BCOS console(2.7.0)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______  ______  ______      ________  ______  ______  ______
|        |      \/      \ /      \ /      \    |        \ /      \ /      \ /      ␣
↪\
| $$$$$$$$\$$$$|  $$$$$$|  $$$$$$|  $$$$$$\   | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__     | $$ | $$___\$| $$   \$| $$  | $$   | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \    | $$  \$$     \| $$     | $$  | $$   | $$    $| $$     | $$  | $$\$$    ␣
↪\
| $$$$$    | $$  _\$$$$$$| $$   __| $$  | $$   | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
```

(continues on next page)

(续上页)

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

若控制台启动失败，请参考控制台FAQ和连接问题进行排查。

使用控制台发送交易

```
# 获取节点信息
[group:1]> getNodeVersion
ClientVersion{
    version='2.6.0',
    supportedVersion='2.6.0',
    chainId='1',
    buildTime='20200819 15:47:59',
    buildType='Darwin/appleclang/RelWithDebInfo',
    gitBranch='HEAD',
    gitCommitHash='e4a5ef2ef64d1943fccc4ebc61467a91779fb1c0'
}

# 获取当前块高
[group:1]> getBlockNumber
0

# 部署和调用HelloWorld合约
[group:1]> deploy HelloWorld
transaction hash:␣
↪0x0e1f192c97a625471597a3483ec230604ff4bb3d22826d118e606ba64bdb0bf9
contract address: 0x302b771c6e7525dc16f945510c210bae0976ad64

[group:1]> call HelloWorld 0x302b771c6e7525dc16f945510c210bae0976ad64 get
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,World!"
]
-------------------------------------------------------------------------------------------
↪----------

# 获取当前块高(块高增加1)
[group:1]> getBlockNumber
1

# 修改HelloWorld合约状态变量的值
[group:1]> call HelloWorld 0x302b771c6e7525dc16f945510c210bae0976ad64 set "hello,␣
↪fisco"
transaction hash:␣
↪0xaa318a24b8a049778c7167bc2631eedadaeaba326a6cf17c79772036a0edc239
-------------------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
transaction status: 0x0
description: transaction executed successfully
-------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
Return value: []
-------------------------------------------------------------------------------------
↪----------
Event logs
Event: {}

# 获取HelloWorld合约状态变量的值
[group:1]>  call HelloWorld 0x302b771c6e7525dc16f945510c210bae0976ad64 get
-------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------
↪----------
Return values:
[
    "hello,fisco"
]
-------------------------------------------------------------------------------------
↪----------

# 获取当前块高
[group:1]> getBlockNumber
2
```

5.2 扩容一个新节点

标签：节点管理 扩容群组 游离节点 新增节点 共识节点

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。

- 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。
- 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。

将指定节点分别转换成共识节点、观察者节点、游离节点，相关操作命令如下：

- 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

下面结合具体操作案例详细阐述群组如何扩容一个新节点。扩容操作分两个阶段， 分别为**为节点生成证书并启动**、**将节点加入群组**。

本节假设用户已经参照搭建第一个区块链网络搭建了一条4节点的联盟链，接下来的操作将生成一个新的节点，然后将节点加入群组1。

如果是使用运维部署工具，请参考这里进行扩容操作。

5.2.1 1. 为节点生成证书并启动

每个节点都需要有一套证书来与链上的其他节点建立连接，扩容一个新节点，首先需要为其签发证书。

为新节点生成私钥证书

接下来的操作都在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node4/conf中会存在机构agency新签发的证书和私钥
# 成功会输出 All completed 提示
bash gen_node_cert.sh -c ../cert/agency -o node4
```

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 node4 -g ../gmcert/agency/
```

准备节点配置文件

1. 拷贝node0/config.ini、node0/start.sh和node0/stop.sh到node4目录;

```
cp node0/config.ini node0/start.sh node0/stop.sh node4/
```

1. 修改node4/config.ini。对于[rpc]模块，修改channel_listen_port=20204和jsonrpc_listen_port
   对于[p2p]模块，修改listen_port=30304并在node.中增加自身节点信息;

```
$ vim node4/config.ini
[rpc]
    ;rpc listen ip
    listen_ip=127.0.0.1
    ;channelserver listen port
    channel_listen_port=20204
    ;jsonrpc listen port
    jsonrpc_listen_port=8549
[p2p]
    ;p2p listen ip
    listen_ip=0.0.0.0
    ;p2p listen port
    listen_port=30304
    ;nodes to connect
```

(continues on next page)

(续上页)

```
    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. 节点3拷贝节点1的node0/conf/group.1.genesis（内含**群组节点初始列表**）和node0/conf/group.1.ini到node4/conf目录下，不需改动；

```
cp node0/conf/group.1.genesis node0/conf/group.1.ini node4/conf/
```

1. 执行node4/start.sh启动节点；

```
bash node4/start.sh
```

1. 确认node4与其他节点连接已经建立，加入网络操作完成。

```
tail -f node4/log/log*  | grep "connected count"
```

```
# 以下日志表明节点node4与其他4个节点建立了连接
info|2020-12-22 20:44:36.113611|[P2P][Service] heartBeat,connected count=4
info|2020-12-22 20:44:46.117942|[P2P][Service] heartBeat,connected count=4
info|2020-12-22 20:44:56.120799|[P2P][Service] heartBeat,connected count=4
```

5.2.2 2. 节点加入群组

获取node4的nodeid

```
cat node4/conf/node.nodeid
```

得到类似下面的字符串就是nodeid，nodeid是节点公钥的16进制表示，国密请执行cat node4/conf/gmnode.nodeid

```
94ae60f93ef9a25a93666e0149b7b4cb0e044a61b7dcd1b00096f2bdb17d1c6853fc81a24e037c9d07803fcaf78f768de
```

使用控制台将node4加入群组1

1. 使用addObserver将node4作为观察节点加入群组1

```
[group:1]> getObserverList
[]

[group:1]> addObserver␣
↪94ae60f93ef9a25a93666e0149b7b4cb0e044a61b7dcd1b00096f2bdb17d1c6853fc81a24e037c9d07803fcaf78f768
{
    "code":1,
    "msg":"Success"
}

[group:1]> getObserverList
[
␣
↪94ae60f93ef9a25a93666e0149b7b4cb0e044a61b7dcd1b00096f2bdb17d1c6853fc81a24e037c9d07803fcaf78f768
]
```

1. 使用addObserver将node4作为观察节点加入群组1

```
[group:1]> getSealerList
[
    "6c41f7e138051a13a220cb186e934398e37700295ff355b87f113704996b3e03750100e16653cda18b5f954d3b7b08d
    "7404cdf7f34f038aba90059ff25dc5f05f538010c55e98976aea6bc954910f34f15a255869751c8fe564bdb0fa1eee8
    "a7b856e5b59072c809ea963fa45ede72f7d37561afff989fbede6cd61a40137e2146db205434788e61b89a57f08c61
    "e5ea1e18717418a57f115bf1cea5168250f86e5b77f74dd15d0c4bf3758ca37002059ba2e54131296d1646a62be5faf
]

[group:1]> addSealer 94ae60f93ef9a25a93666e0149b7b4cb0e044a61b7dcd1b00096f2bdb17d1c6853fc81a24e037c9d07803fcaf78f768
{
    "code":1,
    "msg":"Success"
}

[group:1]> getSealerList
[
    "6c41f7e138051a13a220cb186e934398e37700295ff355b87f113704996b3e03750100e16653cda18b5f954d3b7b08d
    "7404cdf7f34f038aba90059ff25dc5f05f538010c55e98976aea6bc954910f34f15a255869751c8fe564bdb0fa1eee8
    "a7b856e5b59072c809ea963fa45ede72f7d37561afff989fbede6cd61a40137e2146db205434788e61b89a57f08c61
    "e5ea1e18717418a57f115bf1cea5168250f86e5b77f74dd15d0c4bf3758ca37002059ba2e54131296d1646a62be5faf
    "94ae60f93ef9a25a93666e0149b7b4cb0e044a61b7dcd1b00096f2bdb17d1c6853fc81a24e037c9d07803fcaf78f768
]
```

更多操作请参考节点管理

5.3 FISCO BCOS 源码编译

标签：依赖下载 macOS ARM

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。请务必保证机器能够访问githubusercontent.com，否则可能会因为网络不通阻塞编译！

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

5.3.1 1. 克隆代码

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

5.3.2 2. 安装依赖

- Ubuntu

推荐Ubuntu 16.04以上版本，源码编译时需要先安装gcc等工具。

```
sudo apt install -y g++ libssl-dev openssl cmake git build-essential autoconf␣
↪texinfo flex patch bison libgmp-dev zlib1g-dev automake libtool
```

- CentOS

推荐使用CentOS7以上版本。

```
sudo yum install -y epel-release centos-release-scl
sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git flex patch bison gmp-
↪static devtoolset-7
```

- 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
```

5.3.3 3. 编译

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

- Ubuntu

推荐使用Ubuntu 18.04以上版本。

```
cd FISCO-BCOS
mkdir -p build && cd build
cmake ..
# 高性能机器可添加-j4使用4核加速编译
make -j2
```

- CentOS

推荐使用CentOS7以上版本。

```
cd FISCO-BCOS
mkdir -p build && cd build
source /opt/rh/devtoolset-7/enable   # CentOS请执行此命令，其他系统不需要
cmake3 ..
# 高性能机器可添加-j4使用4核加速编译
make  -j2
```

- macOS

推荐xcode11以上版本。

```
cd FISCO-BCOS
mkdir -p build && cd build
cmake ..
make -j2
```

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/
- 如果编译过程中长时间没有相应，一般是不能访问githubusercontent.com导致依赖无法下载，请使用代理或修改hosts使得机器能够访问githubusercontent.com

5.3.4 4. 编译选项介绍

- 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。
- ARCH_NATIVE，默认off，编译时根据本地CPU指令优化以获得更好的性能，在ARM架构上编译可以使用此选项。GCC9以上版本编译暂未适配完成，可以通过打开此编译选项避过问题，完成编译。
- 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。
- CMAKE_BUILD_TYPE，默认RelWithDebInfo，编译类型，如要编译Release版本，通过`cmake -DCMAKE_BUILD_TYPE=Release ..`设置
- DEBUG，默认off，调试模式，编译后会打印将要提交的数据，性能大幅降低，仅用于开发查问题。
- 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
cmake ..
make doc
```

5.3.5 5. ARM平台

FISCO BCOS支持在ARM平台上运行，在基于ARM的平台上编译时，需要打开`ARCH_NATIVE`编译选项。编译指令如下:

```
cmake -DARCH_NATIVE=on ..
make
```

出于指令集和兼容性的考虑，FISCO BCOS发布时不提供ARM平台上预编译的二进制程序，用户如果需要在ARM平台上运行需要在对应机器源码编译，然后使用build_chain的-e选项搭建联盟链网络。例如

```
bash build_chain.sh -l 127.0.0.1:4 -e bin/fisco-bcos
```

5.4 Caliper压力测试指南

标签: 压力测试 `Caliper`

注解: 目前FISCO BCOS适配的Caliper版本为0.2.0，请在部署Caliper运行环境时确保Caliper的版本为0.2.0，如在部署或使用过程中遇到任何问题，请优先参考 https://github.com/FISCO-BCOS/FISCO-BCOS/issues/1248 中的解决方案进行排查。

5.4.1 1. 环境要求

第一步. 配置基本环境

- 部署Caliper的计算机需要有外网权限;
- 操作系统版本需要满足以下要求：Ubuntu >= 16.04、CentOS >= 7或MacOS >= 10.14;
- 部署Caliper的计算机需要安装有以下软件：python 2.7、make、g++、gcc及git。

第二步. 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
```

第三步. 部署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
```

第四步. 安装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
```

5.4.2 2. Caliper部署

第一步. 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
```

第二步. 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"

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。

第三步. 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。

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
```

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
```

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
```

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。

5.4.3 3. 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。

第一步. 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。

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
```

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
```

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。

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"
                },
```

(continues on next page)

(续上页)

```
                {
                    "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"
```

(continues on next page)

(续上页)

```
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。

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
```

(continues on next page)

(续上页)

```
    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
  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。

5.5 存储加密

标签：存储安全 存储加密 落盘加密

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

5.5.1 第一步. 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。

5.5.2 第二步. 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。

5.5.3 第三步. 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 8150 123xyz
```

启动成功，打印日志

```
[1546501342949][TRACE][Load]key-manager started,port=8150
```

5.5.4 第四步. 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 8150 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=8150
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=8150
cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea
```

5.5.5 第五步. 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 8150 ../../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

5.5.6 第六步. 节点运行

直接启动节点即可

```
cd nodes/127.0.0.1/node0/
./start.sh
```

5.5.7 第七步. 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"
}
```

5.6 使用CFCA证书部署节点

标签：CFCA证书 获取证书 开发手册

使用前建议阅读证书说明

5.6.1 购买前注意事项

- 普通版FISCO BCOS节点使用的节点证书算法为EC secp256k1曲线
- 国密版FISCO BCOS节点使用的节点证书算法为SM2
- 用户向CFCA购买前请确认签发算法是否正确
- 购买前请确认证书用途、节点信息已经填写正确
- 使用前请确认已经安装openssl 1.0.2k以上版本
- cfca申请的ca.crt--node.crt二级证书结构与fisco bcos自带工具生成的三级证书结构互相兼容，用户可直接使用cfca签发的二级证书作为节点或sdk的证书使用

建议用户结合白名单机制一起使用

第一步. 获取节点证书

根据CFCA的要求，生成节点私钥node.key和节点证书node.crt

生成的私钥会存储在node.key中，生成后私钥示例如下：

```
-----BEGIN PRIVATE KEY-----
MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgZQE8JAJfs97BAj3mJbme
jSyNG+5kClhuAmXpZ1aI5VyhRANCAASLs7td5X1aDPLynH9HjruPLlovJYx1nIWu
E9mB8iTehZ++qd4b6YWXZoAizCgjvXIRIPEXOSkNaVSzJG7whmgb
-----END PRIVATE KEY-----
```

注意，节点私钥需要私密存储，部署区块链网络时需要用到私钥，每个节点都需要独立的私钥

根据CFCA的要求，使用节点私钥node.key填写信息模板，采用如下方式生成证书请求文件

```
openssl req -new -key node.key -out node.csr
```

将此步生成的证书请求文件node.csr发送给CFCA，得到PEM格式的证书node.crt，同时，CFCA会给你返回一个验证的根证书ca.crt

最终得到的节点证书node.crt示例如下：

```
-----BEGIN CERTIFICATE-----
MIIC/zCCAeegAwIBAgIJALk+dh2WPTueMA0GCSqGSIb3DQEBCwUAMDUxDjAMBgNV
BAMMBWNoYWluMRMwEQYDVQQKDApmaXNjby1iY29zMQ4wDAYDVQQLDAVjaGFpbjAe
Fw0xOTExMDEwODM0MDZaFw0yOTEwMjkwODM0MDZaMDcxDzANBgNVBAMMBmFnZW5j
eTETMBEGA1UECgwKZmlzY28tYmNvczEPMA0GA1UECwwGYWdlbmN5MIIBIjANBgkq
hkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAp4AkR1wfCodSiUraqb0zR/QLMGjpzuz5
r+s78Ao9G9oLq4q8NiGNlPOC+vrQZPtFlTzf8Ax+0DaC6L33JHIDTH6GCE/qDS0W
b1sBCZtB02opqHcKjzgRDlL7ITwS5o7wtEMm3bp1ade7rZbYavlBQzSQ4aBgwDt7
8YX8XkqDun4KYDL6EGPq7xDTKpkE81hCU1L4huXdo4HXO16JeabP8J7EEsIfdtUZ
prl/e+QmvgKs7HThjA61OT2spSAFFNh+q48ZbGuEUF2iRQA47wnk/H9zI5phMSt5
DqQNP4D1w5DM7poDsOhc8GpM7gDOmnhC5gmiomBSYiZRmB3jSUQWHwIDAQABoxAw
DjAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IBAQA9GSvkYPHljWtKSpzV
RqszxyqfpxpH+X1z2LXHrNhE1mVSmcfU74rmu1dOlCOEJcOFTe/XTVUMgB14PWuh
Bpigh/pDMEP3eMLNlKuJhH2SxPmTJTKtAkAg2uu8Q6nr7UJA58ja9PA8clT4YNDO
XnI4r/cEF8qoNBw2Gna8+ENuqlD8IbLMt4JbX0zlmjIL6sJnfRZ8SuOqPANSvJiz
/6j8yys1QX7b66MkGHoUt6mnMzncmO0BCjXOu0DUC4kx57EKLX5jaB3Wm6jJogDE
hOGRKcNXMkct51ISRTJ7Yrn+mP0ELiACDOpM/dbNsqfyxwxQMcmZe2gNHgZcpI1L
C7/4
-----END CERTIFICATE-----
```

第二步. 部署FISCO BCOS节点

部署前需要先获取所有节点的证书，并使用FISCO BCOS企业级部署工具，完成下载安装的操作，操作过程与文档类似。

本节中，我们将部署如下图所示的节点网络：

机器环境

每个节点的IP，端口号为如下：

重要： 针对云服务器中的vps服务器，RPC监听地址需要写网卡中的真实地址(如内网地址或127.0.0.1)，可能与用户登录的ssh服务器不一致。

- 获取generator

```
cd ~/ && git clone https://github.com/FISCO-BCOS/generator.git
```

- 安装依赖

```
cd ~/generator && bash ./scripts/install.sh
```

第三步. 部署证书并启动节点

我们假设用户上述过程中生成了4个节点证书，分别放置为~/cert/node_127.0.0.1_30300，~/cert/node_127.0.0.1_30301，~/cert/node_127.0.0.1_30302，~/cert/node_127.0.0.1_30303路径下，每个目录包含以下文件

```
node_127.0.0.1_30300 # 节点证书存放文件夹，名称必须为node_ip_port
├── cert_127.0.0.1_30300.crt # 节点证书，名称必须为cert_ip_port.crt
├── node.key # 节点私钥，名称必须为node.key
```

- 拷贝节点证书路径

```
cp -r ~/cert/node_127.0.0.1_30300 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30301 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30302 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30303 ./meta
```

- 生成群组创世区块

将上一步所有节点的节点证书node.crt拷贝至meta文件夹下

```
cp ./meta/node_127.0.0.1_30300/cert_127.0.0.1_30300.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30301/cert_127.0.0.1_30301.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30302/cert_127.0.0.1_30302.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30303/cert_127.0.0.1_30303.crt ./meta
```

- 填写group_genesis.ini，教程中采用默认ip
- 生成群组创世区块

```
./generator --create_group_genesis ./group
```

生成的group.1.genesis即为群组创世区块

- 修改节点配置文件

```
```bash
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
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
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546

[node2]
p2p_ip=127.0.0.1
```

(continues on next page)
```

(续上页)

```
rpc_ip=127.0.0.1
p2p_listen_port=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node3]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

- 生成节点

```
echo "" >> ./meta/peers.txt
```

```
./generator --build_install_package ./meta/peers.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
├── node_127.0.0.1_30302 # 127.0.0.1服务器 端口号30300的节点配置文件夹
├── node_127.0.0.1_30303
├── 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
```

后续操作与使用企业级部署工具相同

注解： 国密CFCA证书申请和使用流程与上述过程类似，需额外向CFCA申请证书生成小工具，或使用FISCO BCOS的tassl小工具生成对应的证书请求文件，可以参考开发部署工具，国密节点生成过程使用tassl小工具[build_chain.sh](../manual/build_chain.md)

5.7 使用MySQL存储引擎

标签：分布式存储 MySQL CRUD

本章介绍使用FISCO BCOS搭建基于MySQL存储引擎的分布式存储区块链网络，通过在单机上部署一条基于MySQL分布式存储的4节点FISCO BCOS联盟链，帮助用户掌握FISCO BCOS分布式存储区块链网络的部署流程，请参考这里获取**系统和硬件要求**信息。

5.7.1 1. 安装和配置MySQL

由于FISCO BCOS目前支持的分布式数据库是MySQL，因此在搭建分布式存储区块链节点之前，须先安装和配置MySQL服务。

1.1 安装MySQL

本节介绍如何在ubuntu和centos系统中安装MySQL.

ubuntu系统

安装MySQL

执行下面命令在ubuntu系统中安装MySQL:

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

启动MySQL

MySQL安装成功后，执行下述命令启动MySQL服务，并设置root账户密码:

```
# 启动MySQL服务
sudo service mysql start

# 设置root账户密码
mysql -uroot -p
```

centos系统

安装MySQL 执行下面命令在centos系统中安装MySQL:

```
sudo yum install mariadb*
```

启动MySQL

MySQL安装成功后，执行下面命令启动服务，并设置root账户密码:

```
# 启动MySQL服务
service mariadb start

# 设置root账户密码 (这里将密码设置为123456)
mysql -uroot -p
mysql> set password for root@localhost = password('123456');
```

1.2 配置MySQL

修改MySQL配置文件

```
sudo bash -c 'cat >> /etc/mysql/my.cnf << EOF
max_allowed_packet = 1024M
sql_mode =STRICT_TRANS_TABLES
ssl=0
default_authentication_plugin = mysql_native_password
EOF'
```

重启MySQL服务

```
# ubuntu系统
sudo service mysql restart

# centos系统
sudo service mariadb restart
```

登录MySQL客户端，验证参数是否生效

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)

# 查看default_authentication_plugin的值
MariaDB [(none)]> show variables like 'default_authentication_plugin';
+-------------------------------+-----------------------+
| Variable_name                 | Value                 |
+-------------------------------+-----------------------+
| default_authentication_plugin | mysql_native_password |
+-------------------------------+-----------------------+
1 row in set (0.01 sec)
```

5.7.2 2. 搭建基于MySQL存储的区块链节点

2.1 安装依赖

ubuntu系统

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

centos系统

```
sudo yum install -y openssl openssl-devel curl
```

2.2 创建操作目录, 下载安装脚本

```
## 创建操作目录
mkdir -p ~/fisco && cd ~/fisco

## 下载脚本
curl -#LO https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.7.1/build_
↪chain.sh && chmod u+x build_chain.sh
```

2.3 搭建单机4节点分布式存储区块链网络

注解： 请确保机器的 30300~30303, 20200~20203, 8545~8548 端口没有被占用。

生成单机4节点区块链节点配置

```
# 进入操作目录
cd ~/fisco

# 搭建单机4节点区块链系统
bash build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545
```

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

```
================================================================
Generating CA key...
================================================================
Generating keys and certificates ...
Processing IP=127.0.0.1 Total=4 Agency=agency Groups=1
================================================================
Generating configuration files ...
Processing IP=127.0.0.1 Total=4 Agency=agency Groups=1
================================================================
[INFO] Start Port      : 30300 20200 8545
[INFO] Server IP       : 127.0.0.1:4
[INFO] Output Dir      : /home/ubuntu/fisco/nodes
[INFO] CA Path         : /home/ubuntu/fisco/nodes/cert/
================================================================
[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 -f
================================================================
[INFO] All completed. Files in /home/ubuntu/fisco/nodes
```

修改存储配置为MySQL

通过群组配置文件group.[group_id].ini的storage配置项可配置MySQL，详细可参考这里. 本教程中，所有节点均以root用户名连接同一个本机MySQL数据库，真实业务场景中，可按需修改数据库相关配置(包括MySQL的IP和端口，连接MySQL的用户名和密码等)。

```
# 修改存储类型为mysql
sed -i 's/type=rocksdb/type=mysql/g' ~/fisco/nodes/127.0.0.1/node*/conf/group.1.ini

# 配置数据库用户名和密码 (本教程中，所有节点均以root的用户名连接同一个数据库，root用户密码
为123456)
sed -i 's/db_username=/db_username=root/g' ~/fisco/nodes/127.0.0.1/node*/conf/
↪group.1.ini
sed -i 's/db_passwd=/db_passwd=123456/g' ~/fisco/nodes/127.0.0.1/node*/conf/group.
↪1.ini

# -----配置每个区块链节点在MySQL中创建的库名-----
# 配置node0的数据库名称为db_node0
sed -i 's/db_name=/db_name=db_node0/g' ~/fisco/nodes/127.0.0.1/node0/conf/group.1.
↪ini

# 配置node1的数据库名称为db_node1
sed -i 's/db_name=/db_name=db_node1/g' ~/fisco/nodes/127.0.0.1/node1/conf/group.1.
↪ini

# 配置node2的数据库名称为db_node2
```

(continues on next page)

(续上页)

```
sed -i 's/db_name=/db_name=db_node2/g' ~/fisco/nodes/127.0.0.1/node2/conf/group.1.
↪ini

# 配置node3的数据库名称为db_node3
sed -i 's/db_name=/db_name=db_node3/g' ~/fisco/nodes/127.0.0.1/node3/conf/group.1.
↪ini
```

启动区块链节点

```
# 进入操作目录
cd ~/fisco

# 启动所有节点
bash nodes/127.0.0.1/start_all.sh
```

启动成功后会输出如下日志：

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

2.4 检查区块链网络

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
```

检查网络连接是否正常

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连接信息，从输出可以看出node0与另外3个节点有连接。

```
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
```

检查区块链共识是否正常

```
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|2020-12-22 17:24:43.729402|[g:1][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=2e133146...
info|2020-12-22 17:24:47.740603|[g:1][CONSENSUS][SEALER]++++++++++++++++
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=eb199760...
```

至此，已经搭建好一个基于MySQL的分布式存储区块链网络。

5.7.3 3. 使用控制台发起CRUD操作

为了直观显示区块链状态数据在MySQL中的存储形式，本章通过控制台发起CRUD操作，并通过MySQL客户端查询状态数据信息。 正式开始本操作前，请先参考**搭建第一个区块链网络**中的配置和使用控制台安装控制台。

创建表t_demo

```
# 创建表t_demo，包含(name, item_id, item_name)三个字段，其中name是主key
[group:1]> create table t_demo(name varchar, item_id varchar, item_name varchar,
↪primary key(name))
Create 't_demo' Ok.

# 查看t_demo表的信息
[group:1]> desc t_demo
[
    {
        "key_field":"name",
        "value_field":"item_id,item_name"
    }
]
```

向表中插入数据

```
# 向t_demo表中插入一条数据记录，主key是fruit，item_id为1，item_name为apple1
[group:1]> insert into t_demo (name, item_id, item_name) values (fruit, 1, apple1)
Insert OK:
1 row affected.
```

查询表中数据

```
# 查询主key为fruit的所有数据记录
[group:1]> select * from t_demo where name = fruit
{name=fruit, item_id=1, item_name=apple1}
1 row in set.
```

通过MySQL客户端查询t_demo在数据库中的情况

```
# 连接db_node0数据库
mysql -uroot -p123456 -A db_node0

# 查看数据库中的表
show tables;
MariaDB> show tables;
+-----------------------+
| Tables_in_db_node0    |
+-----------------------+
| _sys_block_2_nonces_  |
| _sys_cns_             |
| _sys_config_          |
| _sys_consensus_       |
| _sys_current_state_   |
| _sys_hash_2_block_    |
```

(continues on next page)

(续上页)

```
| _sys_hash_2_header_   |
| _sys_number_2_hash_   |
| _sys_table_access_    |
| _sys_tables_          |
| _sys_tx_hash_2_block_ |
| u_t_demo              |
+-----------------------+

# 查询用户表t_demo的存储信息
MariaDB> select * from u_t_demo;
+--------+-------+----------+-------+---------+-----------+
| _id_   | _num_ | _status_ | name  | item_id | item_name |
+--------+-------+----------+-------+---------+-----------+
| 100025 |     2 |        0 | fruit | 1       | apple1    |
+--------+-------+----------+-------+---------+-----------+
1 row in set (0.00 sec)
```

5.8 国内镜像和CDN加速

标签：CDN加速 国内镜像 gitee镜像 访问GitHub慢

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。

5.8.1 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release中提供对应的二进制程序和部署工具，同时在CDN也会提供同样的二进制程序。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-macOS.tar.gz： 对应macOS系统的二进制程序
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

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`

5.8.2 FISCO BCOS文档镜像

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。

5.8.3 FISCO BCOS配套工具获取

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，如果下载失败则自动尝试从官网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
```

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。相应的macOS请下载https://gitee.com/FISCO-BCOS/LargeFiles/blob/master/tools/tassl_mac.tar.gz，ARM下请下载https://gitee.com/FISCO-BCOS/LargeFiles/blob/master/tools/tassl-aarch64.tar.gz

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或https://osp-1257653870.cos.ap-guangzhou.myqcloud.com/FISCO-BCOS/FISCO-BCOS/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或https://osp-1257653870.cos.ap-guangzhou.myqcloud.com/FISCO-BCOS/FISCO-BCOS/tools/get_gm_account.sh

5.8.4 举例：使用国内镜像建链

本节以搭建2.7.1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/v2.7.1/build_
↪chain.sh
```

如果下载失败请尝试curl -#LO https://gitee.com/FISCO-BCOS/FISCO-BCOS/raw/master/manual/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2.7.1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 2.7.1
```

5.8.5 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 centos-release-scl
sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git flex patch bison gmp-
↪static devtoolset-7
```

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。v2.5.0以上版本由于使用evmone作为虚拟机引擎，需要从GitHub下载部分依赖，需要保证机器能正常访问`githubusercontent.com`。

https://gitee.com/FISCO-BCOS/LargeFiles/tree/master/libs

编译源码

1. 创建编译目录

```
cd FISCO-BCOS && mkdir build && cd build
```

1. 使用gcc7

```
source /opt/rh/devtoolset-7/enable
```

1. 编译

```
cmake3 ..
make -j2
```

高阶教程

- 搭建多机区块链网络
- 扩容一个新节点
- FISCO BCOS源码编译
- 使用MySQL存储账本
- 国内镜像和CDN加速
- 使用CFCA证书部署节点
- 配置落盘加密
- 使用Caliper进行性能压测

博客

- 了解智能合约概念
  - 智能合约初探：概念与演变
- 快速上手智能合约开发
  - FISCO BCOS智能合约开发快速入门
  - 如何优雅地编写智能合约
  - FISCO BCOS 控制台详解，飞一般的区块链体验
  - FISCO BCOS的外部账户生成
  - FISCO BCOS的CNS合约命名服务
  - 智能合约测试的技巧与实践
- 预编译智能合约开发
  - 预编译合约开发指南
  - FISCO BCOS CRUD使用指南
- Solidity合约开发
  - 深入浅出Solidity
  - 智能合约编写之Solidity的基础特性
  - 智能合约编写之Solidity的高级特性
  - 智能合约编写之Solidity的设计模式
  - 智能合约编写之Solidity的编程攻略
  - Meetup回顾|16大技巧快速实现高水准智能合约
  - Meetup回顾|存证&积分场景合约设计技巧与实例代码解析

CHAPTER 6

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。

6.1 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。

6.1.1 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。

6.1.2 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。

6.1.3 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。

6.2 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。

6.2.1 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。

6.2.2 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。

6.2.3 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。

CHAPTER 7

区块链网络搭建

标签：搭建区块链网络 索引 开发手册 区块链搭建

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。

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

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

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
- 极端异常下的共识恢复应急方案
  - 极端异常下的共识恢复应急方案
- 海量数据治理
  - 海量数据治理使用说明

重要:

- **核心特性**
  - 多群组部署
  - 分布式存储

7.1 硬件和系统要求

标签：硬件要求 操作系统 开发手册 内存要求

7.1.1 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

7.1.2 操作系统

- CentOS 7.2+
- Ubuntu 16.04+
- macOS 10.14+

7.2 获取FISCO BCOS可执行程序

标签：可执行程序 开发手册 预编译程序 源码编译 编译教程

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。

7.2.1 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**。

7.2.2 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。

7.2.3 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 18.04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 zlib1g-dev
```

- CentOS

推荐使用CentOS7以上版本。

```
$ sudo yum install -y epel-release centos-release-scl
$ sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git flex patch bison␣
↪gmp-static devtoolset-7
```

- 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
$ source /opt/rh/devtoolset-7/enable   # CentOS请执行此命令，其他系统不需要
# CentOS请使用cmake3
$ cmake ..
# 高性能机器可添加-j4使用4核加速编译
# macOS 编译出现 "ld: warning: direct access" 提示时，可以忽略
$ 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

- 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。
- ARCH_NATIVE，默认off，编译时根据本地CPU指令优化以获得更好的性能，在ARM架构上编译可以使用此选项。GCC9以上版本编译暂未适配完成，可以通过打开此编译选项避过问题，完成编译。

- 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。
- CMAKE_BUILD_TYPE，默认RelWithDebInfo，编译类型，如要编译Release版本，通过`cmake -DCMAKE_BUILD_TYPE=Release ..`设置
- DEBUG，默认off，调试模式，编译后会打印将要提交的数据，性能大幅降低，仅用于开发查问题。
- 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
```

7.3 获取证书

标签：获取证书 开发手册 x509证书 CFCA证书 CA SDK证书 证书检测脚本

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：

使用第三方证书部署节点的说明，可以参考使用CFCA证书部署节点

7.3.1 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

7.3.2 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

7.3.3 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
```

(continues on next page)

(续上页)

```
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。

7.3.4 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

7.4 使用CFCA证书部署节点

标签：CFCA证书 获取证书 开发手册

使用前建议阅读证书说明

7.4.1 购买前注意事项

- 普通版FISCO BCOS节点使用的节点证书算法为EC secp256k1曲线
- 国密版FISCO BCOS节点使用的节点证书算法为SM2
- 用户向CFCA购买前请确认签发算法是否正确

- 购买前请确认证书用途、节点信息已经填写正确
- 使用前请确认已经安装openssl 1.0.2k以上版本
- cfca申请的ca.crt--node.crt二级证书结构与fisco bcos自带工具生成的三级证书结构互相兼容，用户可直接使用cfca签发的二级证书作为节点或sdk的证书使用

建议用户结合白名单机制一起使用

7.4.2 本地生成证书请求文件

普通证书操作流程

根据CFCA的要求，生成节点私钥node.key和节点证书node.crt

生成的私钥会存储在node.key中，生成后私钥示例如下：

```
-----BEGIN PRIVATE KEY-----
MIGEAgEAMBAGByqGSM49AgEGBSuBBAAKBG0wawIBAQQgZQE8JAJfs97BAj3mJbme
jSyNG+5kClhuAmXpZ1aI5VyhRANCAASLs7td5X1aDPLynH9HjruPLlovJYx1nIWu
E9mB8iTehZ++qd4b6YWXZoAizCgjvXIRIPEXOSkNaVSzJG7whmgb
-----END PRIVATE KEY-----
```

注意，节点私钥需要私密存储，部署区块链网络时需要用到私钥，每个节点都需要独立的私钥

根据CFCA的要求，使用节点私钥node.key填写信息模板，采用如下方式生成证书请求文件

```
openssl req -new -key node.key -out node.csr
```

将此步生成的证书请求文件node.csr发送给CFCA，得到PEM格式的证书node.crt，同时，CFCA会给你返回一个验证的根证书ca.crt

最终得到的节点证书node.crt示例如下：

```
-----BEGIN CERTIFICATE-----
MIIC/zCCAeegAwIBAgIJALk+dh2WPTueMA0GCSqGSIb3DQEBCwUAMDUxDjAMBgNV
BAMMBWNoYWluMRMwEQYDVQQKDApmaXNjby1iY29zMQ4wDAYDVQQLDAVjaGFpbjAe
Fw0xOTExMDEwODM0MDZaFw0yOTEwMjkwODM0MDZaMDcxDzANBgNVBAMMBmFnZW5j
eTETMBEGA1UECgwKZmlzY28tYmNvczEPMA0GA1UECwwGYWdlbmN5MIIBIjANBgkq
hkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAp4AkR1wfCodSiUraqb0zR/QLMGjpzuz5
r+s78Ao9G9oLq4q8NiGNlPOC+vrQZPtFlTzf8Ax+0DaC6L33JHIDTH6GCE/qDS0W
b1sBCZtB02opqHcKjzgRDlL7ITwS5o7wtEMm3bp1ade7rZbYavlBQzSQ4aBgwDt7
8YX8XkqDun4KYDL6EGPq7xDTKpkE81hCU1L4huXdo4HXO16JeabP8J7EEsIfdtUZ
prl/e+QmvgKs7HThjA61OT2spSAFFNh+q48ZbGuEUF2iRQA47wnk/H9zI5phMSt5
DqQNP4D1w5DM7poDsOhc8GpM7gDOmnhC5gmiomBSYiZRmB3jSUQWHwIDAQABoxAw
DjAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBCwUAA4IBAQA9GSvkYPHljWtKSpzV
Rqszxyqfpxp H+X1z2LXHrNhE1mVSmcfU74rmu1dOlCOEJcOFTe/XTVUMgB14PWuh
Bpigh/pDMEP3eMLNlKuJhH2SxPmTJTKtAkAg2uu8Q6nr7UJA58ja9PA8clT4YNDO
XnI4r/cEF8qoNBw2Gna8+ENuqlD8IbLMt4JbX0zlmjIL6sJnfRZ8SuOqPANSvJiz
/6j8yys1QX7b66MkGHoUt6mnMzncmO0BCjXOu0DUC4kx57EKLX5jaB3Wm6jJogDE
hOGRKcNXMkct51ISRTJ7Yrn+mP0ELiACDOpM/dbNsqfyxwxQMcmZe2gNHgZcpI1L
C7/4
-----END CERTIFICATE-----
```

7.4.3 开始部署FISCO BCOS节点

部署前需要先获取所有节点的证书，并使用FISCO BCOS企业级部署工具，完成下载安装的操作，操作过程与文档类似。

本节中，我们将部署如下图所示的节点网络：

机器环境

每个节点的IP，端口号为如下：

重要： 针对云服务器中的vps服务器，RPC监听地址需要写网卡中的真实地址(如内网地址或127.0.0.1)，可能与用户登录的ssh服务器不一致。

- 获取generator

```
cd ~/ && git clone https://github.com/FISCO-BCOS/generator.git
```

- 安装依赖

```
cd ~/generator && bash ./scripts/install.sh
```

普通版部署教程

我们假设用户上述过程中生成了4个节点证书，分别放置为~/cert/node_127.0.0.1_30300，~/cert/node_127.0.0.1_30301，~/cert/node_127.0.0.1_30302，~/cert/node_127.0.0.1_30303路径下，每个目录包含以下文件

```
node_127.0.0.1_30300 # 节点证书存放文件夹，名称必须为node_ip_port
├── cert_127.0.0.1_30300.crt # 节点证书，名称必须为cert_ip_port.crt
├── node.key # 节点私钥，名称必须为node.key
```

- 拷贝节点证书路径

```
cp -r ~/cert/node_127.0.0.1_30300 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30301 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30302 ./meta
```

```
cp -r ~/cert/node_127.0.0.1_30303 ./meta
```

- 生成群组创世区块

将上一步所有节点的节点证书node.crt拷贝至meta文件夹下

```
cp ./meta/node_127.0.0.1_30300/cert_127.0.0.1_30300.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30301/cert_127.0.0.1_30301.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30302/cert_127.0.0.1_30302.crt ./meta
```

```
cp ./meta/node_127.0.0.1_30303/cert_127.0.0.1_30303.crt ./meta
```

- 填写group_genesis.ini，教程中采用默认ip
- 生成群组创世区块

```
./generator --create_group_genesis ./group
```

生成的group.1.genesis即为群组创世区块

- 修改节点配置文件

```bash
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
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
p2p_listen_port=30301
channel_listen_port=20201
jsonrpc_listen_port=8546

[node2]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
p2p_listen_port=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node3]
p2p_ip=127.0.0.1
rpc_ip=127.0.0.1
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

- 生成节点

```
echo "" >> ./meta/peers.txt
```

```
./generator --build_install_package ./meta/peers.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
├── node_127.0.0.1_30302 # 127.0.0.1服务器 端口号30300的节点配置文件夹
├── node_127.0.0.1_30303
├── scripts # 节点的相关工具脚本
```

(continues on next page)

(续上页)

```
├── 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
```

后续操作与使用企业级部署工具相同

7.4.4 如何使用国密CFCA证书

国密CFCA证书申请和使用流程与上述过程类似，需额外向CFCA申请证书生成小工具，或使用FISCO BCOS的tassl小工具生成对应的证书请求文件，可以参考开发部署工具，国密节点生成过程使用tassl小工具build_chain.sh

7.5 配置CA黑白名单

标签：CA黑白名单 开发手册 拒绝连接

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》。

7.5.1 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
```

7.5.2 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
```

7.5.3 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）

7.5.4 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
f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf5636e
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
↪",
      "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",
      "Node": "node2",
      "NodeID":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf56
↪",
      "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":
↪"f306eb1066ceb9d46e3b77d2833a1bde2a9899cfc4d0433d64b01d03e79927aa60a40507c5739591b8122ee609cf56
↪",
      "Topic": []
    },
    {
      "Agency": "agency",
      "IPAndPort": "127.0.0.1:30301",
      "Node": "node1",
      "NodeID":
↪"7718df20f0f7e27fdab97b3d69deebb6e289b07eb7799c7ba92fe2f43d2efb4c1250dd1f11fa5b5ce687c8283d6503
↪",
      "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
```

(continues on next page)

(续上页)

```
[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": []
    }
  ]
}
```

7.6 节点配置

标签：节点配置 开发手册 config.ini 端口配置 日志级别 配置

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。

7.6.1 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
- v2.6.0 版本开始，RPC 支持 ipv4 和 ipv6

RPC配置示例如下：

```
# ipv4
[rpc]
    channel_listen_ip=0.0.0.0
    jsonrpc_listen_ip=127.0.0.1
    channel_listen_port=30301
    jsonrpc_listen_port=30302

# ipv6
[rpc]
    channel_listen_ip=::1
    jsonrpc_listen_ip=::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
- v2.6.0 版本开始，P2P 支持 ipv4 和 ipv6

- `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。
- v2.6.0 版本开始，P2P 支持 ipv4 和 ipv6

P2P配置示例如下：

```
# ipv4
[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

# ipv6
[p2p]
    listen_ip=::1
    listen_port=30300
    node.0=[::1]:30300
    node.1=[::1]:30304
    node.2=[::1]:30308
    node.3=[::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
    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=8150
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
```

7.6.2 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；
- consensus_timeout：PBFT共识过程中，每个区块执行的超时时间，默认为3s，单位为秒，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
    ;共识过程中区块最长执行时间，默认为3秒
    consensus_timeout=3
    ; 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
```

7.6.3 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

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`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`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 / Scalable, RocksDB is recommended
    type=RocksDB
    max_capacity=256
    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
```

可选配置：SDK白名单配置

为了实现sdk到群组的访问控制，FISCO BCOS v2.6.0引入了群组级的SDK白名单访问控制机制，配置位于`group.{group_id}.ini`的`[sdk_allowlist]`，默认关闭，群组级别SDK白名单机制请参考这里。

重要： FISCO BCOS v2.6.0默认关闭SDK到群组的白名单访问控制功能，即默认情况下sdk与所有群组均可通信，若要开启sdk与群组间基于白名单的访问控制功能，需要将 *;public_key.0* 等配置项前面的分号去掉

- `public_key.0`、`public_key.1`、...、`public_key.i`：配置允许与该群组进行通信的SDK公钥公钥列表。

SDK白名单配置示例如下：

```
[sdk_allowlist]
; When sdk_allowlist is empty, all SDKs can connect to this node
; when sdk_allowlist is not empty, only the SDK in the allowlist can connect to␣
↪this node
; public_key.0 should be nodeid, nodeid's length is 128
public_key.
↪0=b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a995
```

7.6.4 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
```

(continues on next page)

(续上页)

```
# 获取区块执行超时时间
[group:1]> getSystemConfigByKey consensus_timeout
3

# 设置区块执行超时时间为5s
[group:1]> setSystemConfigByKey consensus_timeout 5
{
    "code":0,
    "msg":"success"
}
```

7.7 使用国密算法

标签：国密算法 国密证书 国密版FISCO BCOS 国密配置

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。

7.7.1 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/v2.7.1/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 搭建国密版本的链
# -G 设置`chain.sm_crypto_channel=true`。确认sdk支持的情况下（web3sdk v2.5.0+），可以指
定-G参数，连接也使用国密SSL
$ ./build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545 -g -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
================================================================
```

(continues on next page)

(续上页)

```
[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。

7.7.2 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，也支持国密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
```

重要： 国密环境下统一使用了国密商用密码相关算法，在solidity中使用keccak256/sha3指令，虚拟机中实际执行sm3算法！

7.7.3 国密版SDK使用

详细操作参考SDK文档。

7.7.4 国密版控制台配置

1.x版本控制台需要配置国密选项，详情操作参考配置国密版控制台。

7.7.5 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，2.6及其以上版本控制台不需要额外配置国密选项，1.x版本控制台的配置方法请参考控制台操作手册。

7.7.6 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 8150 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 8150 nodes/127.0.0.1/node0/conf/origin_cert/
↪node.key ed157f4588b86d61a2e1745efe71e6ea
```

7.8 多群组部署

标签：多群组 加入群组 节点扩容 新增节点

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。

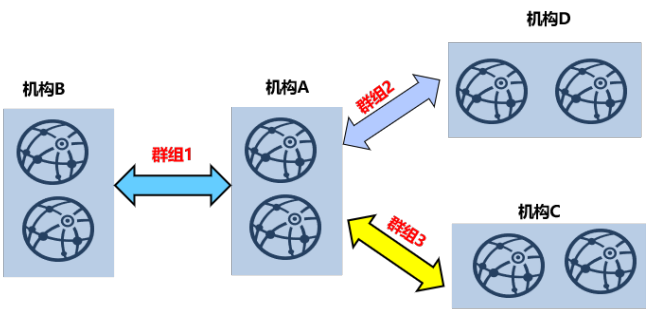
重要：

- 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

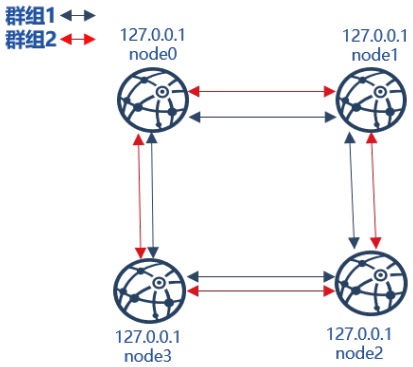
7.8.1 星形拓扑和并行多组

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。

7.8.2 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
```

7.8.3 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/v2.7.1/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Java SDK连接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。2.6及其以上版本控制台使用手册请参考这里, 1.x版本控制台使用手册请参考这里。

```
#回到fisco目录
$ cd ~/fisco

# 获取控制台
$ curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/
↪download_console.sh && bash download_console.sh

# 进入控制台操作目录
$ cd console

# 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
```

(continues on next page)

(续上页)

```
/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

# 拷贝控制台配置
$ cp ~/fisco/console/conf/config-example.toml ~/fisco/console/conf/config.toml
```

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(2.6.1)!
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...
```

(continues on next page)

(续上页)

```
# 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
transaction hash:␣
↪0xd0305411e36d2ca9c1a4df93e761c820f0a464367b8feb9e3fa40b0f68eb23fa
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
transaction hash:␣
↪0xd0305411e36d2ca9c1a4df93e761c820f0a464367b8feb9e3fa40b0f68eb23fa
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
```

(continues on next page)

(续上页)

```
# 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
↪

↪227c600c2e52d8ec37aa9f8de8db016ddc1c8a30bb77ec7608b99ee2233480d4c06337d2461e24c26617b6fd53acfa6
↪

↪7a50b646fcd9ac7dd0b87299f79ccaa2a4b3af875bd0947221ba6dec1c1ba4add7f7f690c95cf3e796296cf4adc989f
↪

↪8b2c4204982d2a2937261e648c20fe80d256dfb47bda27b420e76697897b0b0ebb42c140b4e8bf0f27dfee64c946039
```

(continues on next page)

(续上页)

```
]
# 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......

# 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
```

7.8.4 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/v2.7.1/
↪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
```

(continues on next page)

(续上页)

```
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
$ 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/v2.7.1/
↪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

# 拷贝控制台配置
$ cp ~/fisco/console/conf/config-example.toml ~/fisco/console/conf/config.toml

# 修改控制台连接节点的端口为20100和20101
# linux系统使用如下命令:
$ sed -i 's/127.0.0.1:20200/127.0.0.1:21000/g' ~/fisco/console/conf/config.toml
$ sed -i 's/127.0.0.1:20201/127.0.0.1:21001/g' ~/fisco/console/conf/config.toml

# mac系统使用如下命令:
$ sed -i .bkp 's/127.0.0.1:20200/127.0.0.1:21000/g' ~/fisco/console/conf/config.
↪toml
$ sed -i .bkp 's/127.0.0.1:20201/127.0.0.1:21001/g' ~/fisco/console/conf/config.
↪toml
```

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(2.6.0)!
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
transaction hash:␣
↪0xd0305411e36d2ca9c1a4df93e761c820f0a464367b8feb9e3fa40b0f68eb23fa
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
transaction hash:␣
↪0xd0305411e36d2ca9c1a4df93e761c820f0a464367b8feb9e3fa40b0f68eb23fa
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/
```

(continues on next page)

(续上页)

```
# 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
```

7.9 使用分布式存储

标签：分布式存储 MySQL

7.9.1 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');
```

7.9.2 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
ssl=0
default_authentication_plugin = mysql_native_password
```

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)

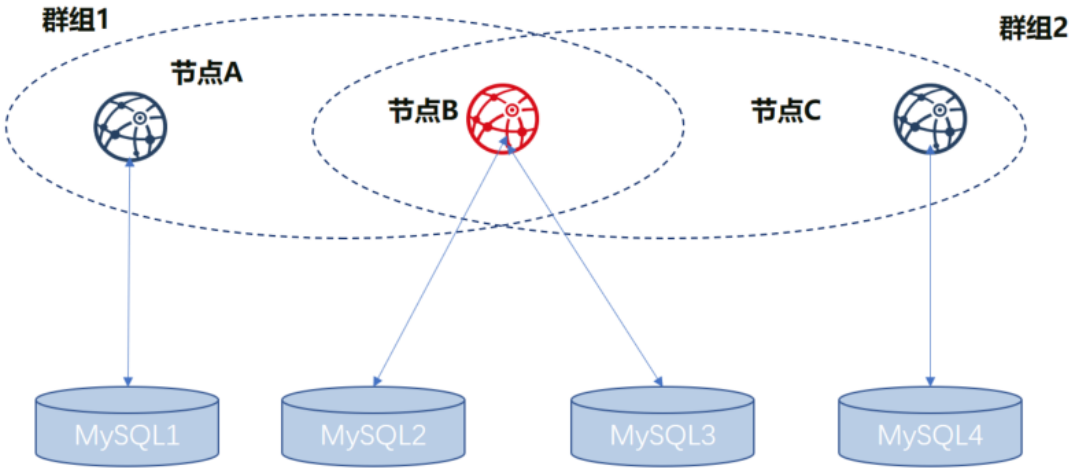
#执行下面命令，查看sql_mode的值
MariaDB [(none)]>  show variables like 'sql_mode%';
+---------------+----------------------+
| Variable_name | Value                |
+---------------+----------------------+
| sql_mode      | STRICT_TRANS_TABLES |
+---------------+----------------------+
1 row in set (0.00 sec)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

7.9.3 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/v2.7.1/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/v2.7.1/download_
 ↪console.sh && bash download_console.sh
cp -n console/conf/config-example.toml console/conf/config.toml
cp nodes/127.0.0.1/sdk/* console/conf/
```

修改配置文件

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

```
[network]
peers=["127.0.0.1:20300", "127.0.0.1:20301"]    # The peer list to connect
```

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                                      |
+------------------------------------------------------------+
| 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)
```

7.10 设置SDK白名单

标签：SDK访问控制 SDK白名单 SDK白配置

FISCO BCOS 2.0开始支持多群组，但没有控制SDK对多群组的访问权限，SDK只要可以与节点建立连接，就可以访问节点的所有群组，会带来安全风险。

FISCO BCOS v2.6.0引入了群组级别的SDK白名单机制，控制SDK对群组的访问权限，并提供脚本支持SDK白名单列表的动态加载，进一步提升区块链系统的安全性。

注解:

- 当配置项中的SDK白名单列表数目为0时，节点没有开启SDK白名单控制功能，任意SDK均可访问该群组；
- SDK白名单是节点级别的访问控制机制，开启该功能的节点基于本地配置的白名单列表控制SDK对本节点群组的访问权限

- SDK白名单机制控制SDK对节点所有群组级接口的访问权限

7.10.1 配置方法

群组级别的SDK白名单配置选项位于`group.{group_id}.ini`，具体可参考 这里

注解： 每个群组的SDK白名单配置位于 *group.{group_id}.ini* 配置文件的 *[sdk_allowlist]* 配置项中，详细可参考 这里

7.10.2 获取SDK公钥

将sdk加入到白名单列表前，首先需要获取SDK的公钥，用于设置`group.*.ini`的`public_key`，各个版本获取SDK公钥的方法如下：

新搭建的链

新搭建的链生成的SDK证书自带SDK私钥信息，非国密版为`sdk.publickey`，国密版为`gmsdk.publickey`

```
# 设证书已拷贝到SDK，则进入SDK目录，执行如下命令(设sdk位于~/fisco目录)
$ cd ~/fisco/web3sdk

# 获取国密版SDK公钥
$ cat dist/conf/sdk.publickey

# 获取非国密版SDK公钥
$ cat dist/conf/gmsdk.publickey
```

旧链

旧链需要使用openssl或tassl命令生成`sdk.publickey`(国密版是`gmsdk.publickey`)，并从生成的文件中加载公钥，具体如下：

非国密

```
# 进入SDK目录，执行如下命令：
$ openssl ec -in dist/conf/sdk.key -text -noout 2> /dev/null | sed -n '7,11p' | tr␣
↪-d ": \n" | awk '{print substr($0,3);}' | cat > dist/conf/sdk.publickey
# 获取SDK公钥
$ cat dist/conf/sdk.publickey
```

国密链

```
# 注：必须保证~/.fisco/tassl存在
$ ~/.fisco/tassl ec -in dist/conf/gmsdk.key -text -noout 2> /dev/null | sed -n '7,
↪11p' | sed 's/://g' | tr "\n" " " | sed 's/ //g' | awk '{print substr($0,3);}' ␣
↪| cat > dist/conf/gmsdk.publickey
# 获取SDK公钥
$ cat dist/conf/gmsdk.publickey
```

7.10.3 动态修改SDK白名单列表

为方便用户修改SDK白名单列表，每个节点的scripts目录下提供了reload_sdk_allowlist.sh脚本用于重新加载SDK白名单列表。

注解: 旧链节点无 *reload_sdk_allowlist.sh* 脚本，可通过命令 *curl -#LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/reload_sdk_allowlist.sh* 下载该脚本。

7.10.4 示例

本节以将控制台加入和删除白名单列表为例，详细展示SDK白名单机制的使用方法。

搭建区块链并拷贝证书到控制台

请参考安装

获取控制台公钥信息

```
# 进入到控制台目录
$ cd ~/fisco/console/

# 通过sdk.publickey获取控制台公钥信息
$ cat conf/sdk.publickey
ebf5535c92f7116310ed9e0f9fc9bfc66a607415d4fa444d91f528485eff61b15e40a70bc5d73f0441d3959efbc7718c2
```

开启SDK白名单机制

将某控制台的公钥添加到node0的group.[group_id].ini配置文件白名单列表中：

```
[sdk_allowlist]
public_key.
 ↪0=b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a995
```

重新加载SDK白名单列表

```
$ bash node0/scripts/reload_sdk_allowlist.sh
 [INFO] node0 is trying to reload sdk allowlist. Check log for more information.

# 热加载SDK白名单列表成功后，节点输出了如下日志：
parseSDKAllowList,
 ↪sdkAllowList=[b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf98
 ↪enableSDKAllowListControl=true
```

控制台访问节点

注解: 由于SDK白名单是节点级别的访问控制机制，为了展示 *node0* 对SDK的访问控制功能，控制台仅连接 *node0*

由于node0没有配置控制台对群组的访问权限，部署合约的返回结果如下：

```
# 控制台部署HelloWorld合约
[group:1]> deploy HelloWorld
sendRawTransaction The SDK is not allowed to access this group
```

添加控制台到SDK白名单列表

将控制台配置到node0的白名单列表中：

```
[sdk_allowlist]
public_key.
↪0=b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a995
public_key.
↪1=ebf5535c92f7116310ed9e0f9fc9bfc66a607415d4fa444d91f528485eff61b15e40a70bc5d73f0441d3959efbc771
```

重新加载SDK白名单列表：

```
$ bash node0/scripts/reload_sdk_allowlist.sh
 [INFO] node0 is trying to reload sdk allowlist. Check log for more information.

# 热加载SDK白名单列表成功后，节点输出了如下日志：
parseSDKAllowList,
↪sdkAllowList=[b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf98
↪ebf5535c92f7116310ed9e0f9fc9bfc66a607415d4fa444d91f528485eff61b15e40a70bc5d73f0441d3959efbc7718
↪enableSDKAllowListControl=true
```

控制台访问节点

node0加入控制台到白名单列表后，控制台可正常部署合约，如下：

```
[group:1]> deploy HelloWorld
contract address: 0xcd4ccd96c86fe8e4f27b056c0fdb7eb4ca201f0f
```

7.11 存储加密

标签：存储安全 存储加密 落盘加密

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

7.11.1 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。

7.11.2 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。

7.11.3 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 8150 123xyz
```

启动成功，打印日志

```
[1546501342949][TRACE][Load]key-manager started,port=8150
```

7.11.4 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 8150 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=8150
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=8150
cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea
```

7.11.5 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 8150 ../../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

7.11.6 节点运行

直接启动节点即可

```
cd nodes/127.0.0.1/node0/
./start.sh
```

7.11.7 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"
}
```

7.12 组员节点管理

标签：节点管理 退出群组 游离节点 观察者节点 共识节点

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。

7.12.1 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␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969
# 查询共识节点列表
[group:1]> getSealerList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff
]

# 将指定节点转换为观察者节点
[group:1]> addObserver␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969

# 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␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969

# 查询节点列表
[group:1]> getNodeIDList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff
]
```

(continues on next page)

(续上页)

```
[group:1]> getSealerList
[]
[group:1]> getObserverList
[]
```

7.12.2 操作案例

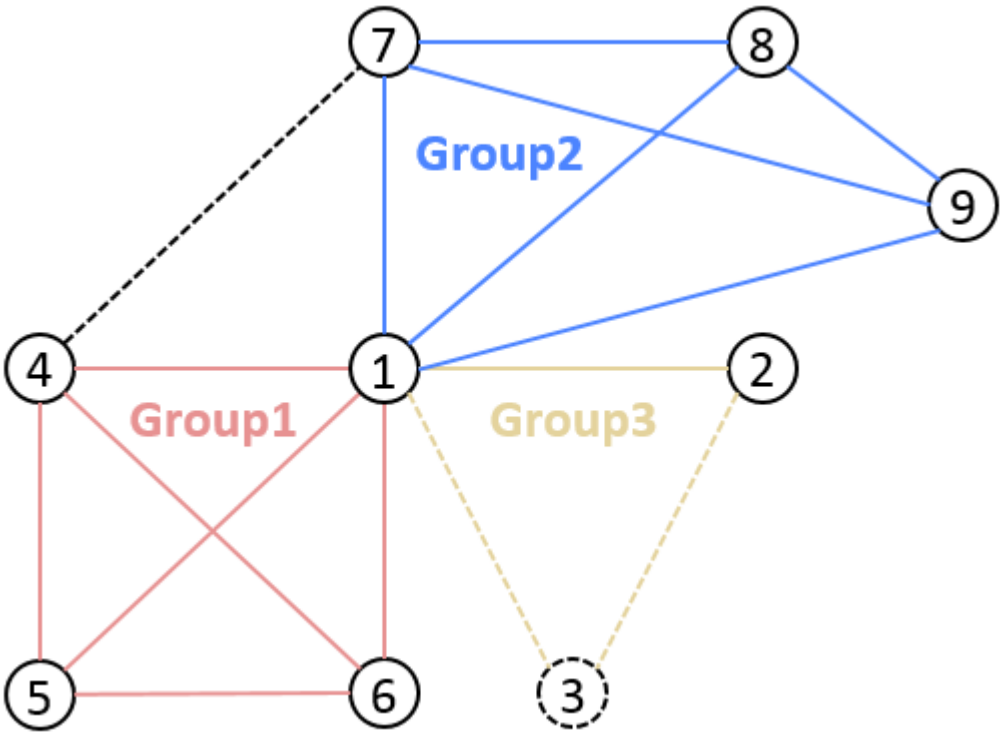
下面结合具体操作案例详细阐述群组扩容操作及节点退网操作。扩容操作分两个阶段， 分别为**将节点加入网络**、**将节点加入群组**。退网操作也分为两个阶段，为**将节点退出群组**、**将节点退出网络**。

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（如果是国密节点，使用␣
↪-g 参数）
$ ./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。

7.13 账户权限控制

标签：账户权限 增删委员 修改链配置项 委员账号 运维账号

7.13.1 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            ␣
↪          |
| 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的控制台中操作，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。

- 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            ␣
↪          |
-------------------------------------------------------------------------------
↪----------
```

- 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            ␣
↪          |
-------------------------------------------------------------------------------
↪----------
```

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。

- 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            ␣
↪          |
-------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

- 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
```

- 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%。

- 账户1添加账户2为委员

```
[group:1]> grantCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
--------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num        ␣
↪          |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                      1            ␣
↪          |
| 0x0b6f526d797425540ea70becd7adac7d50f4a7c0  |                      9            ␣
↪          |
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                     12            ␣
↪          |
--------------------------------------------------------------------------------
↪----------
```

- 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%
```

- 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。

- 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            ␣
↪          |
--------------------------------------------------------------------------------
↪----------
```

- 使用运维账号部署HelloWorld

```
[group:1]> deploy HelloWorld
contract address: 0xac1e28ad93e0b7f9108fa1167a8a06585f663726
```

- 使用账号1部署HelloWorld失败

```
[group:1]> deploy HelloWorld
permission denied
```

- 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.
```

7.13.2 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
```

(continues on next page)

(续上页)

```
----------------------------------------------------------------------------------------
↪----------
|                    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
↪
```

(continues on next page)

(续上页)

```
↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078c
↪

↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2ac
↪

↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9c
]
```

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␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d951
{
    "code":0,
    "msg":"success"
}

[group:1]> getObserverList
[

↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d951
]

[group:1]> getSealerList
[

↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078c
↪

↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2ac
↪

↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9c
]
```

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␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d951
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> getSealerList
[

↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078c
↪

↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2ac
↪

↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9c
```

(continues on next page)

(续上页)

```
]

[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
|
| 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
-------------------------------------------------------------------------------------
↪----------
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。

7.14 极端异常下的共识恢复应急方案

标签：共识恢复 存活性

重要:

- 使用此方案之前，请确保您对区块链共识、存储、以及区块链无法篡改特性有深刻理解。该方案所涉及的操作都是高危操作，仅用于发生极端异常情况导致区块链网络无法正常运行时的紧急修复。切勿在其它场合使用，使用不当可能导致数据异常甚至系统异常，请明确知悉，谨慎使用。确定使用该恢复方案之前，建议先对已有数据进行备份。
- 此方案并不会篡改区块链数据，详细请阅读文章 《如何解释"我篡改了区块链"这个问题》

背景：FISCO BCOS 支持的 PBFT 共识算法 需要区块链网络中至少存在 `2*f+1` 个共识节点正常工作（假定总节点数为 `3*f+1` ）才能维持网络的良性运转。但是实际生产环境中往往会出现各种特殊情况，如网络脑裂、节点网络中断、节点硬件崩溃，从而导致网络中节点数量少于 `2*f+1`，这时网络将无法对交易和区块达成共识，网络陷入瘫痪。

目标：处理 FISCO BCOS 区块链网络中由于节点数量不足 `2*f+1` 而引起的 PBFT 网络共识异常问题。

解决方案：手动关闭正常共识节点，修改正常共识节点数据库 `_sys_consensus_` 表中的异常共识节点信息，将异常共识节点的 type 字段值从 sealer 修改为 observer，也就是手动将异常共识节点修改为观察者节点，然后启动正常共识节点。

FISCO BCOS 支持不同的数据库存储模式，不同的数据库模式在操作和存储方面存在差异。接下来分别讲解 Mysql 和 RocksDB 数据库存储模式在面对 PBFT 共识节点不足的情况下可采取的处理方案。

7.14.1 Mysql存储模式

处理步骤：假设网络中存在A、B、C、D四个共识节点，C、D节点由于某些原因发生崩溃，导致网络无法正常共识打包出块。

1. **关闭节点A和节点B**：为了防止操作数据库给正在运行的网络造成影响，建议先关闭节点A和节点B；
2. **手动修改节点A和节点B的数据库**：确保A、B、C、D节点块高一致的前提下修改节点A和节点B数据库中 `_sys_consensus_` 表，将节点C和节点D的 type 字段值从 sealer 修改为 observer；
3. **启动节点A和节点B**：重启节点之后，网络中共识节点只有节点A和节点B，满足 PBFT 共识条件，网络可正常共识打包出块；
4. **手动修改节点C和节点D的数据库**：修改节点C和节点D中数据库 `_sys_consensus_` 表，将节点C和节点D的 type 字段值从 sealer 修改为 observer；
5. **启动节点C和节点D**：节点C和节点D作为观察者节点，可以顺利同步区块，但无法参与网络共识；
6. **添加共识节点**：通过控制台调用 addSealer 命令，将节点C和节点D转换为共识节点。

注解： 该解决方案会引入一个新的问题：当对现有网络进行共识节点扩容时，新加入的共识节点在同步区块的过程中会在某一个区块高度出现同步失败的情况，此时需要手动将新节点数据库 *_sys_consensus_* 表中节点C和节点D的 type 字段值从 sealer 修改为 observer，然后重启新加入的共识节

点。重启后共识节点将继续同步区块，当同步的区块高度追上网络中最新的区块高度时，新加入的共识节点方可参与网络共识。

关闭节点A和节点B

网络在运行的过程中直接修改数据库表中字段值是非常危险的，所以先关闭节点A和节点B。

```
# 关闭节点A
bash ./node0/stop.sh

# 关闭节点B
bash ./node1/stop.sh
```

手动修改节点A和节点B的数据库

在A、B、C、D节点拥有相同的最新区块高度前提下，为了可以让由节点A和节点B组成的网络可以继续共识，需要手动修改节点A和节点B数据库中 _sys_consensus_ 表的 type 字段，将该字段值从 sealer 修改为 observer。

1. 查看节点C和节点D的 node_id

```
# 节点C编号
> cat ./node2/conf/node.nodeid
da8688893fcf5b4194e4da1b79e77e81be9cb740fb373e0f59bf3d28a349e7fba9b79ec36e05715fcbe7dbb68cae661d45

# 节点D编号
> cat ./node3/conf/node.nodeid
17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e30bfd82b84
```

1. 修改节点A和节点B数据库表 _sys_consensus_

```
# 登录 mysql
> mysql -u username -p password  # 根据实际情况填写 username 和 password

# 选择数据库
> use database-name # 根据实际情况填写 database-name

# 查看 _sys_consensus_ 表信息
> select * from _sys_consensus_;
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
| _id_   | _num_ | _status_ | name | type     | node_id
↪                                                                                  ⌟
↪             | enable_num |
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
| 100004 |     0 |        0 | node | sealer   |⌟
↪060a3dc076c57e42dabb9a54d5d0460dae97f1d7f63a5f4d24f4c126f7ee5d5c2f0ced34f7a661b59fe10962f26dd2f
↪| 0          |
| 100005 |     0 |        0 | node | sealer |⌟
↪17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e30bfd82b
↪| 0          |
| 100006 |     0 |        0 | node | sealer   |⌟
↪205d9fdeb3e3598bb6e50b4f54dab64841187cb3f8fa7f1eac71dd45fda07b81d3615857891fcd8c7cdcbb2374d8a5d
↪| 0          |
| 100007 |     6 |        0 | node | sealer   |⌟
↪da8688893fcf5b4194e4da1b79e77e81be9cb740fb373e0f59bf3d28a349e7fba9b79ec36e05715fcbe7dbb68cae661
↪| 6          |
```

(continues on next page)

(续上页)

```
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
4 rows in set (0.00 sec)

# 将节点C和节点D的 type 字段修改为 observer
> update _sys_consensus_ set type="observer" where _id_="100005";
> update _sys_consensus_ set type="observer" where _id_="100007";

# 再次查看 _sys_consensus_  表信息
> select * from _sys_consensus_;
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
| _id_   | _num_ | _status_ | name | type     | node_id
↪                                                                                  ⌟ ⌟
↪              | enable_num |
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
| 100004 |     0 |        0 | node | sealer   |⌟
↪060a3dc076c57e42dabb9a54d5d0460dae97f1d7f63a5f4d24f4c126f7ee5d5c2f0ced34f7a661b59fe10962f26dd2f
↪| 0           |
| 100005 |     0 |        0 | node | observer |⌟
↪17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e30bfd82b
↪| 0           |
| 100006 |     0 |        0 | node | sealer   |⌟
↪205d9fdeb3e3598bb6e50b4f54dab64841187cb3f8fa7f1eac71dd45fda07b81d3615857891fcd8c7cdcbb2374d8a5d
↪| 0           |
| 100007 |     6 |        0 | node | observer   |⌟
↪da8688893fcf5b4194e4da1b79e77e81be9cb740fb373e0f59bf3d28a349e7fba9b79ec36e05715fcbe7dbb68cae661
↪| 6           |
+--------+-------+----------+------+----------+-----------------------------------
↪-----------------------------------------------------------------------------------
↪-------------+------------+
4 rows in set (0.00 sec)

# 继续修改另外一个节点的数据库表
```

启动节点A和节点B

启动节点A和节点B，从数据库中读取新的节点配置信息。通过查看节点A的日志发现网络共识已恢复正常。

```
# 启动节点A
bash ./node0/start.sh

# 启动节点B
bash ./node1/start.sh
```

手动修改节点C和节点D的数据库

如果此时将节点C和节点D启动，因为节点C和节点D数据库表中记录的共识节点包括A、B、C、D，所以节点C和节点D无法同步网络中共识产生的新区块，也无法参与到新区块的共识中。为此，用户需手动修改节点C和节点D数据库中 _sys_consensus_ 表的字段，将节点C和节点D的 type 字段值从 sealer 修改为 observer。

修改步骤同 **手动修改节点A和节点B的数据库** 小节，不重复占用篇幅。

启动节点C和节点D

再次启动节点C和节点D之后，节点C和节点D作为观察者节点可顺利同步区块，但是无法参与区块共识。

```
# 启动节点C
bash ./node2/start.sh

# 启动节点D
bash ./node3/start.sh
```

注解：特殊情况处理：节点C和节点D启动之后，此时网络已经可以正常共识出块，共识节点为A和B。假如节点A开启了 **binary_log=true** 配置项用于备份数据，某一个时刻节点A的主数据发生异常损坏，节点A需要从备份数据中进行数据恢复，在数据恢复完成之后，节点A需要再次手动修改数据库 *_sys_consensus_* 表中节点C和节点D的 type 字段值，从 sealer 修改为 observer，然后重启节点A。同理，如果节点B、C、D遇到这种情况，也需要进行相同的操作。

添加共识节点

现在节点C和节点D是观察者节点，只能同步区块却无法参与区块共识。为此，可通过在控制台执行 addSealer 命令将节点C和节点D转换为共识节点。关于节点角色转换可参考：组员管理。

```
# 转换节点C为共识节点
[group:1]> addSealer␣
↪da8688893fcf5b4194e4da1b79e77e81be9cb740fb373e0f59bf3d28a349e7fba9b79ec36e05715fcbe7dbb68cae661c
{
    "code":0,
    "msg":"success"
}

# 转换节点D为共识节点
[group:1]> addSealer␣
↪17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e30bfd82b8
{
    "code":0,
    "msg":"success"
}
```

7.14.2 RocksDB存储模式

处理步骤：假设网络中存在A、B、C、D四个共识节点，C、D节点由于某些原因发生崩溃，导致网络无法正常共识打包出块。

1. **获取rocksdb-storage工具**：编译 FISCO BCOS源码 获取 rocksdb-storage 工具，使用该工具查询、修改 rocksdb 数据库中信息；
2. **关闭节点A和节点B**：为了防止操作数据库给正在运行的网络造成影响，建议先关闭节点A和节点B；
3. **手动修改节点A和节点B的数据库**：确保A、B、C、D节点块高一致的前提下修改节点A和节点B数据库中 _sys_consensus_ 表，将节点C和节点D的 type 字段值从 sealer 修改为 observer；
4. **启动节点A和节点B**：启动节点之后，网络中共识节点只有节点A和节点B，满足 PBFT 共识条件，网络可正常共识打包出块；
5. **手动修改节点C和节点D的数据库**：修改节点C和节点D中数据库 _sys_consensus_ 表，将节点C和节点D的 type 字段值从 sealer 修改为 observer；

6. **启动节点C和节点D**：节点C和节点D作为观察者节点，可以顺利同步区块，但无法参与网络共识；
7. **添加共识节点**：通过控制台调用 addSealer 命令，将节点C和节点D转换为共识节点。

注解： **该解决方案会引入一个新的问题**：当对现有网络进行共识节点扩容时，新加入的共识节点在同步区块的过程中会在某一个区块高度出现同步失败的情况，此时需要手动将新节点数据库 *_sys_consensus_* 表中节点C和节点D的 type 字段值从 sealer 修改为 observer，然后重启新加入的共识节点。重启后共识节点将继续同步区块，当同步的区块高度追上网络中最新的区块高度时，新加入的共识节点方可参与网络共识。

获取rocksdb-storage工具

FISCO BCOS 提供查询、修改 rocksdb 数据库信息的 rocksdb-storage 工具。该工具需要手动编译 FISCO BCOS源码 获取，编译时通过 `cmake -DTOOL=on ..` 打开工具开关，编译成功后 rocksdb-storage 工具位于 `FISCO-BCOS/build/bin/`。详细编译步骤可参考：编译。

```
#  编译成功后将 rocksdb-storage 工具移动到节点主目录，如 ~/fisco/nodes/127.0.0.1/
cp ./bin/rocksdb-storage ~/fisco/nodes/127.0.0.1/
```

有关 rocksdb-storage 工具的使用说明可以参考本页第三章：rocksdb-storage 工具

关闭节点A和节点B

网络在运行的过程中直接修改数据库表中字段值是非常危险的，所以先关闭节点A和节点B。

```
# 关闭节点A
bash ./node0/stop.sh

# 关闭节点B
bash ./node1/stop.sh
```

手动修改节点A和节点B的数据库

在A、B、C、D节点拥有相同的最新区块高度前提下，为了可以让由节点A和节点B组成的网络可以继续共识，需要使用 rocksdb-storage 工具手动修改节点A和节点B数据库中 _sys_consensus_ 表的 type 字段，将该字段值从 sealer 修改为 observer。

1. 查看节点C和节点D的 node_id

```
# 节点C编号
> cat ./node2/conf/node.nodeid
700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816bb11cc

# 节点D编号
> cat ./node3/conf/node.nodeid
b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc743ed0
```

1. 修改节点A和节点B数据库表 _sys_consensus_

```
# 使用 rocksdb-storage 工具查询节点A数据库 _sys_consensus_ 表数据
> ./rocksdb-storage -p ./node0/data/group1/block/RocksDB -s _sys_consensus_ node
DB path : ./node0/data/group1/block/RocksDB
select [_sys_consensus_,node] || params num : 2
================ open Table [_sys_consensus_] success! key node
***0 [ enable_num=0 ][ name=node ][ node_
↪id=700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816b
↪][ type=sealer ]
```

(continues on next page)

(续上页)

```
***1 [ enable_num=0 ][ name=node ][ node_
↪id=b64c6d23f479e4740c09662bcbf721c265abf34d511644d92344a20bb7c661de33d72cf56d3ba0d3eb4cb553cd49
↪][ type=sealer ]
***2 [ enable_num=0 ][ name=node ][ node_
↪id=b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc7
↪][ type=sealer ]
***3 [ enable_num=0 ][ name=node ][ node_
↪id=e794f3666fb0c9d36e8f5e9a99d49bc15304a8332f3945c260bfdafd66dc04b5a6c1397e8900a835cce54b58b94d
↪][ type=sealer ]

# 修改节点A数据库 _sys_consensus_ 表中节点C的 type 字段值
> ./rocksdb-storage -p ./node0/data/group1/block/RocksDB -u _sys_consensus_ node␣
↪node_id␣
↪700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816bb11
↪type observer
DB path : ./node0/data/group1/block/RocksDB
update [_sys_consensus_,node,node_id,
↪700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816bb11
↪type,observer] || params num : 6
open Table [_sys_consensus_] success!
condition is [node_
↪id=700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816b
update [type:observer]

# 修改节点A数据库 _sys_consensus_ 表中节点D的 type 字段值
> ./rocksdb-storage -p ./node0/data/group1/block/RocksDB -u _sys_consensus_ node␣
↪node_id␣
↪b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc743e
↪type observer
DB path : ./node0/data/group1/block/RocksDB
update [_sys_consensus_,node,node_id,
↪b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc743e
↪type,observer] || params num : 6
open Table [_sys_consensus_] success!
condition is [node_
↪id=b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc7
update [type:observer]

# 再次查询节点A数据库 _sys_consensus_ 表数据
> ./rocksdb-storage -p ./node0/data/group1/block/RocksDB -s _sys_consensus_ node
DB path : ./node0/data/group1/block/RocksDB
select [_sys_consensus_,node] || params num : 2
================ open Table [_sys_consensus_] success! key node
***0 [ enable_num=0 ][ name=node ][ node_
↪id=700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816b
↪][ type=observer ]
***1 [ enable_num=0 ][ name=node ][ node_
↪id=b64c6d23f479e4740c09662bcbf721c265abf34d511644d92344a20bb7c661de33d72cf56d3ba0d3eb4cb553cd49
↪][ type=sealer ]
***2 [ enable_num=0 ][ name=node ][ node_
↪id=b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc7
↪][ type=observer ]
***3 [ enable_num=0 ][ name=node ][ node_
↪id=e794f3666fb0c9d36e8f5e9a99d49bc15304a8332f3945c260bfdafd66dc04b5a6c1397e8900a835cce54b58b94d
↪][ type=sealer ]

# 继续修改节点B的数据库表
```

启动节点A和节点B

启动节点A和节点B，从数据库中读取新的节点配置信息。启动成功后，通过查看节点A的日志发现网络共识已恢复正常。

```
# 启动节点A
bash ./node0/start.sh

# 启动节点B
bash ./node1/start.sh
```

手动修改节点C和节点D的数据库

如果此时将节点C和节点D启动，因为节点C和节点D数据库表中记录的共识节点包括A、B、C、D，所以节点C和节点D无法同步网络中共识产生的新区块，也无法参与到新区块的共识中。为此，用户需手动修改节点C和节点D数据库中 _sys_consensus_ 表的字段，将节点C和节点D的 type 字段值从 sealer 修改为 observer。

修改步骤同 **手动修改节点A和节点B的数据库** 小节，不重复占用篇幅。

启动节点C和节点D

启动节点C和节点D之后，节点C和节点D作为观察者节点可顺利同步区块，但是无法参与区块共识。

```
# 启动节点C
bash ./node2/start.sh

# 启动节点D
bash ./node3/start.sh
```

注解：特殊情况处理：节点C和节点D启动之后，此时网络已经可以正常共识出块，共识节点为A和B。假如节点A开启了 **binary_log=true** 配置项用于备份数据，某一个时刻节点A的主数据发生异常损坏，节点A需要从备份数据中进行数据恢复，在数据恢复完成之后，节点A需要再次手动修改数据库 *_sys_consensus_* 表中节点C和节点D的 type 字段值，从 sealer 修改为 observer，然后重启节点A。同理，如果节点B、C、D遇到这种情况，也需要进行相同的操作。

添加共识节点

现在节点C和节点D是观察者节点，只能同步区块却无法参与区块共识。为此，可通过在控制台执行 addSealer 命令将节点C和节点D转换为共识节点。关于节点角色转换可参考：组员管理。

```
# 转换节点C为共识节点
[group:1]> addSealer␣
↪700076fb7e70c073dd04858f1030cfe51b482e6ec80e20cdcfa33335091f930b67620dd5b1627607195b1512816bb11c
{
    "code":0,
    "msg":"success"
}

# 转换节点D为共识节点
[group:1]> addSealer␣
↪b88921fc39d90190fa908b24cbbd8282d823984e24f56a01f622c87c2d0f255e02eec03dbf427e29a56180b8dbc743ec
{
    "code":0,
    "msg":"success"
}
```

7.14.3 rocksdb-storage 工具

FISCO BCOS 提供 rocksdb-storage 工具帮助用户对 rocksdb 数据库进行常规 CRUD 操作，此外 rocksdb-storage 工具也支持手动创建数据表。

获取途径：手动编译 FISCO BCOS源码。编译时通过 `cmake -DTOOL=on ..` 打开工具开关，编译成功后 rocksdb-storage 工具位于 `FISCO-BCOS/build/bin/`。详细编译步骤可参考：编译。

帮助

```
Usage:
      -h [ --help ]               help of rocksdb reader
      -c [ --createTable ] arg    [TableName] [KeyField] [ValueField]
      -p [ --path ] arg (=data/)  [RocksDB path]
      -s [ --select ] arg         [TableName] [priKey]
      -u [ --update ] arg         [TableName] [priKey] [keyEQ] [valueEQ] [Key]
                                  [NewValue]
      -i [ --insert ] arg         [TableName] [priKey] [Key]:[Value],...,[Key]:[Valu
                                  e]
      -r [ --remove ] arg         [TableName] [priKey]
      -e [ --encrypt ] arg        [encryptKey] [SMCrypto]
e.g
        ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s _
↪sys_consensus_ node
```

选项介绍

-c选项[Optional]

同 `--createTable` 选项，用于创建数据库表。参数包括：

- TableName：表名；
- KeyField：主键字段名；
- ValueField：其余字段名。如果有多个字段名则以符号 `,` 进行分隔。

-p选项

同 `--path` 选项，用于指定节点 rocksdb 数据文件所在存储目录。rocksdb 默认文件路径为 `./node*/data/group*/block/RocksDB/`。

-s选项[Optional]

同 `--select` 选项，用于查询数据库表中数据项记录。参数包括：

- TableName：表名；
- priKey：主键字段值；

注解： priKey 是 KeyFiled 的具体值。比如 _sys_consensus_ 系统表的主键是 name，主键值为 node。详细请参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/design/storage/storage.html。

-u选项[Optional]

同 `--update` 选项，用于更新数据库表中数据项记录。参数包括：

- TableName：表名；
- priKey：主键字段值；
- keyEQ：key 值，用于指定非主键字段名；
- valueEQ：value 值，用于指定非主键字段值，和 keyEQ 组合使用；
- Key：key 值，待修改的字段名；
- NewValue：value 值，待修改值。

注解： keyEQ 和 valueEQ 组合在一起的作用可以类比为 mysql 中的 where 条件。

-i选项[Optional]

同 `--insert` 选项，用于新增数据库表中数据项记录。参数包括：

- TableName：表名；
- priKey：主键字段值，同时也是待插入的数据；
- `Key:Value`：以符号 `,` 连接的多个待插入数据。

-r选项[Optional]

同 `--remove` 选项，用于删除数据库表中数据项记录。参数包括：

- TableName：表名；
- priKey：主键字段值，此处用于指定待删除的数据项。

-e选项[Optional]

同 `--encrypt` 选项，用于标识已开启落盘加密的节点，有关落盘加密可 参考。参数包括：

- encryptKey：dataKey；
- SMCrypto：是否开启国密。

使用举例

注解： 使用 rocksdb-storage 工具操作网络中开启落盘加密节点的数据库时，需要额外指定 *-e* 选项，同时建议将 -e 选项放置在命令行最后。

1. 搭建单群组四节点区块链网络，可参考：安装。搭建完之后需要将所有节点关闭，rocksdb 数据库无法同时被打开两次。
2. 编译 FISCO BCOS 源码，获取 rocksdb-storage 工具，移动工具至 `~/fisco/` 文件夹目录
3. **查询数据库表中数据项记录**：使用 `-s` 参数查看 `_sys_consensus_` 系统表中的节点数据。关于该系统表的说明可参考：AMDB

```
# -p 指定 rocksdb 文件夹，-s 指定 TableName 和 priKey
# 非国密落盘加密 ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/
↪RocksDB -s _sys_consensus_ node -e 123456
# 国密落盘加密 ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/
↪RocksDB -s _sys_consensus_ node -e 123456 SMCrypto
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s _
↪sys_consensus_ node

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
select [_sys_consensus_,node] || params num : 2
================ open Table [_sys_consensus_] success! key node
***0 [ enable_num=0 ][ name=node ][ node_
↪id=5ea630a8cad6681c9e54a30009caf2afd3fe75a01a46375a5c9ad947d42b3bae8830cd9dbf67ea5d3cccb3eb
↪][ type=sealer ]
***1 [ enable_num=0 ][ name=node ][ node_
↪id=6a6bfde992a8bc7b5fa77a5dfc5512df588b7ea8229ae48925929f77762f40e9bdecef420f5af7b752d9335f
↪][ type=sealer ]
***2 [ enable_num=0 ][ name=node ][ node_
↪id=6b5612f7206ef9559f1e33efc7f427bc4ab2e3a4674be4c7939f2ea49cb23dab341ab8f6f4b1586b38822e51
↪][ type=sealer ]
***3 [ enable_num=0 ][ name=node ][ node_
↪id=8e30de9cf192ee644b735d8a4765c7160a3227fa398e040eafa52cd834fd04260b5b960a6dc72c1a3ed8cbb7
↪][ type=sealer ]
```

4. **更新数据库表中数据项记录**：使用 -u 参数修改 _sys_consensus_ 系统表中的节点数据

```
# -p 指定 rocksdb 文件夹，-u 指定 TableName、priKey、keyEQ、valueEQ、Key 和␣
↪NewValue
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -u _
↪sys_consensus_ node node_id␣
↪5ea630a8cad6681c9e54a30009caf2afd3fe75a01a46375a5c9ad947d42b3bae8830cd9dbf67ea5d3cccb3eb02c
↪type observer

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
update [_sys_consensus_,node,node_id,
↪5ea630a8cad6681c9e54a30009caf2afd3fe75a01a46375a5c9ad947d42b3bae8830cd9dbf67ea5d3cccb3eb02c
↪type,observer] || params num : 6
open Table [_sys_consensus_] success!
condition is [node_
↪id=5ea630a8cad6681c9e54a30009caf2afd3fe75a01a46375a5c9ad947d42b3bae8830cd9dbf67ea5d3cccb3eb
update [type:observer]

# 查看修改后的 _sys_consensus_ 系统表
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s _
↪sys_consensus_ node

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
select [_sys_consensus_,node] || params num : 2
================ open Table [_sys_consensus_] success! key node
***0 [ enable_num=0 ][ name=node ][ node_
↪id=5ea630a8cad6681c9e54a30009caf2afd3fe75a01a46375a5c9ad947d42b3bae8830cd9dbf67ea5d3cccb3eb
↪][ type=observer ]
***1 [ enable_num=0 ][ name=node ][ node_
↪id=6a6bfde992a8bc7b5fa77a5dfc5512df588b7ea8229ae48925929f77762f40e9bdecef420f5af7b752d9335f
↪][ type=sealer ]
***2 [ enable_num=0 ][ name=node ][ node_
↪id=6b5612f7206ef9559f1e33efc7f427bc4ab2e3a4674be4c7939f2ea49cb23dab341ab8f6f4b1586b38822e51
↪][ type=sealer ]
***3 [ enable_num=0 ][ name=node ][ node_
↪id=8e30de9cf192ee644b735d8a4765c7160a3227fa398e040eafa52cd834fd04260b5b960a6dc72c1a3ed8cbb7
↪][ type=sealer ]
```

5. **新增数据库表中数据项记录**：使用 -i 参数新增 _sys_consensus_ 系统表中的节点数据

```
# -p 指定 rocksdb 文件夹，-i 指定 TableName、priKey 和 [Key]:[Value],...,
↪[Key]:[Value]，注意 priKey 为 peer
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -i _
↪sys_consensus_ peer node_
↪id:17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e3
↪type:sealer,enable_num:0

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
insert [_sys_consensus_,peer,node_
↪id:17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e3
↪type:sealer,enable_num:0] || params num : 3
open Table [_sys_consensus_] success!

# 查看新增后的 _sys_consensus_ 系统表，注意 priKey 为 peer
>  ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s _
↪sys_consensus_ peer

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
select [_sys_consensus_,peer] || params num : 2
================ open Table [_sys_consensus_] success! key peer
***0 [ enable_num=0 ][ name=peer ][ node_
↪id=17f4ad3e71418be3e074c897c73fdd07a3a69a20524476dfbd4dc18fc6284714acf58d753bbc651549f7b5e3
↪][ type=sealer ]
```

6. **删除数据库表中数据项记录**：使用 -r 参数删除 _sys_consensus_ 系统表中的节点数据

```
# -p 指定 rocksdb 文件夹，-r 指定 TableName 和 priKey，注意 priKey 为 peer
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -r _
↪sys_consensus_ peer

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
remove [_sys_consensus_,peer] || params num : 2
open Table [_sys_consensus_] success!

# 查看删除后的 _sys_consensus_ 系统表，注意 priKey 为 peer
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s _
↪sys_consensus_ peer

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
select [_sys_consensus_,peer] || params num : 2
================ open Table [_sys_consensus_] success! key peer is empty!
```

7. **创建数据库表**：使用 -c 参数为数据库创建新的数据表 user，包括 username、age 和 sex 三个字段，其中 username 是主键

```
# -p 指定 rocksdb 文件夹，-c 指定 TableName、KeyField 和 ValueField
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -c␣
↪user username age,sex

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
createTable [user,username,age,sex] || params num : 3
KeyField:[username]
ValueField:[age,sex]
createTable [user] success!

# 插入数据
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -i␣
↪user Bob age:20,sex:male

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
insert [user,Bob,age:20,sex:male] || params num : 3
open Table [user] success!
```

(continues on next page)

(续上页)

```
# 查看新插入的数据
> ./rocksdb-storage -p ./nodes/127.0.0.1/node0/data/group1/block/RocksDB -s␣
↪user Bob

DB path : ./nodes/127.0.0.1/node0/data/group1/block/RocksDB
select [user,Bob] || params num : 2
================ open Table [user] success! key Bob
***0 [ age=20 ][ sex=male ][ username=Bob ]
```

7.15 Caliper压力测试

标签：压测 Caliper

7.15.1 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
```

7.15.2 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。

7.16 查看日志

标签：日志格式 日志关键词 问题排查 查看日志

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

7.16.1 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

7.16.2 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

7.16.3 日志模块关键字

FISCO BCOS日志中核心模块关键字如下：

CHAPTER 8

区块链应用开发

标签：应用开发 索引 开发手册 区块链应用

外部调用

- 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
- 使用预编译合约
  - 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
- 隐私保护
  - 预编译合约支持同态加密、群/环签名验证

FISCO BCOS SDK

- FISCO BCOS Java SDK
  - (推荐使用)稳定、功能强大、无内置控制台、支持FISCO BCOS 2.0+

- **FISCO BCOS Web3SDK**
  - 旧版Java SDK、支持FISCO BCOS 2.0+
- **FISCO BCOS NodeJS SDK**
  - 简单轻便、有内置控制台
- **FISCO BCOS Python SDK**
  - 简单轻便、有内置控制台
- **FISCO BCOS Go SDK**
  - 简单轻便、有内置控制台

重要:

- **核心特性**
  - 并行合约

8.1 创建和使用账户

标签: 创建账户 国密账户 密钥文件

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。

8.1.1 账户的创建

使用脚本创建账户

国密生成账户脚本`get_gm_account.sh`与非国密`get_account.sh`选项和使用方式一致，请参考操作即可，不再赘述。

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://osp-1257653870.cos.ap-guangzhou.myqcloud.com/FISCO-BCOS/FISCO-BCOS/tools/tassl-1.0.2/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。

8.1.2 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。

8.1.3 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，生成的的账户私钥文件在accounts目录下，控制台加载私钥时需要指定私钥文件。 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());
```

8.1.4 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
```

8.2 智能合约开发

标签：Solidity 智能合约 合约开发 预编译合约

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。
- 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。

8.2.1 Solidity合约开发

- WeBASE合约IDE
- Solidity官方文档
- Remix在线IDE

重要： 国密环境下统一使用了国密商用密码相关算法，在solidity中使用keccak256/sha3指令，虚拟机中实际执行sm3算法！

8.2.2 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 (uint256);
    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 {
    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);
    }
```

(continues on next page)

(续上页)

```
    //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。

8.2.3 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
```

(continues on next page)

(续上页)

```
    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);
    // 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 <0.6.11;
pragma experimental ABIEncoderV2;
```

(continues on next page)

(续上页)

```
import "./Table.sol";

contract TableTest {
    event CreateResult(int256 count);
    event InsertResult(int256 count);
    event UpdateResult(int256 count);
    event RemoveResult(int256 count);

    TableFactory tableFactory;
    string constant TABLE_NAME = "t_test";
    constructor() public {
        tableFactory = TableFactory(0x1001); //The fixed address is 0x1001 for␣
↪TableFactory
        // the parameters of createTable are tableName,keyField,"vlaueFiled1,
↪vlaueFiled2,vlaueFiled3,..."
        tableFactory.createTable(TABLE_NAME, "name", "item_id,item_name");
    }

    //select records
    function select(string memory name)
    public
    view
    returns (string[] memory, int256[] memory, string[] memory)
    {
        Table table = tableFactory.openTable(TABLE_NAME);

        Condition condition = table.newCondition();

        Entries entries = table.select(name, condition);
        string[] memory user_name_bytes_list = new string[](
            uint256(entries.size())
        );
        int256[] memory item_id_list = new int256[](uint256(entries.size()));
        string[] memory item_name_bytes_list = new string[](
            uint256(entries.size())
        );

        for (int256 i = 0; i < entries.size(); ++i) {
            Entry entry = entries.get(i);

            user_name_bytes_list[uint256(i)] = entry.getString("name");
            item_id_list[uint256(i)] = entry.getInt("item_id");
            item_name_bytes_list[uint256(i)] = entry.getString("item_name");
        }

        return (user_name_bytes_list, item_id_list, item_name_bytes_list);
    }
    //insert records
    function insert(string memory name, int256 item_id, string memory item_name)
    public
    returns (int256)
    {
        Table table = tableFactory.openTable(TABLE_NAME);

        Entry entry = table.newEntry();
        entry.set("name", name);
        entry.set("item_id", item_id);
        entry.set("item_name", item_name);

        int256 count = table.insert(name, entry);
        emit InsertResult(count);
```

(continues on next page)

(续上页)

```
        return count;
    }
    //update records
    function update(string memory name, int256 item_id, string memory item_name)
    public
    returns (int256)
    {
        Table table = tableFactory.openTable(TABLE_NAME);

        Entry entry = table.newEntry();
        entry.set("item_name", item_name);

        Condition condition = table.newCondition();
        condition.EQ("name", name);
        condition.EQ("item_id", item_id);

        int256 count = table.update(name, entry, condition);
        emit UpdateResult(count);

        return count;
    }
    //remove records
    function remove(string memory name, int256 item_id) public returns (int256) {
        Table table = tableFactory.openTable(TABLE_NAME);

        Condition condition = table.newCondition();
        condition.EQ("name", name);
        condition.EQ("item_id", item_id);

        int256 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。

8.2.4 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,
        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)
    {
        // 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])
```

(continues on next page)

(续上页)

```
{
    // 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
// 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;
```

(continues on next page)

(续上页)

```
    // 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(),
                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");
        }
```

(continues on next page)

(续上页)

```
        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 ../manual/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]>
```

8.3 使用预编译合约

标签：预编译合约 高性能合约

FISCO BCOS 2.0受以太坊内置合约启发，实现了一套预编译合约框架。未来，我们还会尝试将现有的典型业务场景抽象，开发成预编译合约模板，作为底层提供的基础能力，帮助用户更快的更方便的在业务中使用FISCO BCOS。

8.3.1 预编译合约的好处

可访问分布式存储接口：基于这套框架，用户可以访问本地DB存储状态，实现自己需要的任何逻辑。

更好的性能表现：由于实现是C++代码，会编译在底层中，不需要进入EVM执行，可以有更好的性能。

无需学习Solidity语言即可上手：基于FISCO BCOS预编译合约框架，开发者可以使用C++开发自己的预编译合约，快速实现需要的业务逻辑，而不需要学习Solidity语言。

并行模型大幅提升处理能力：我们在2.0版本中基于预编译合约和DAG实现了合约的并行执行，用户只需要指定接口冲突域，底层会自动根据冲突域构建交易依赖关系图，根据依赖关系尽可能并行执行交易，从而使得交易处理能力大幅提升。

8.3.2 预编译合约与以太坊内置合约的对比

上述说到，FISCO BCOS 预编译合约受以太坊内置合约启发，但实现原理却是大不相同的。

以太坊通过内置合约来避免EVM中复杂计算的代价，以太坊当前使用内置合约实现了8个函数（如下表所示）。可以看到，以太坊内置合约占用了0x1-0x8这8个地址，每个内置合约实际上就是一个本地函数的调用，只能用于状态无关的计算使用。

地址	函数
0x1	ecrecover
0x2	sha256hash
0x3	ripemd160hash
0x4	dataCopy
0x5	bigModExp
0x6	bn256Add
0x7	bn256ScalarMul
0x8	bn256Pairing

用户在Solidity中使用内置合约，需要借助call这个操作，依次输入下列参数

call(gasLimit, to, value, inputOffset, inputSize, outputOffset, outputSize)

包括内置合约地址、输入参数偏移、输入参数大小、输出参数偏移和输出参数大小，对用户而言，这不是一件简单的事情。

而FISCO BCOS的预编译合约框架，支持复杂的参数类型，支持通过AMDB读取和存储数据。每个预编译合约地址固定，合约内可以实现多个接口，所实现接口的调用方式与原生Solidity完全相同。

下图是比较直观的对比：

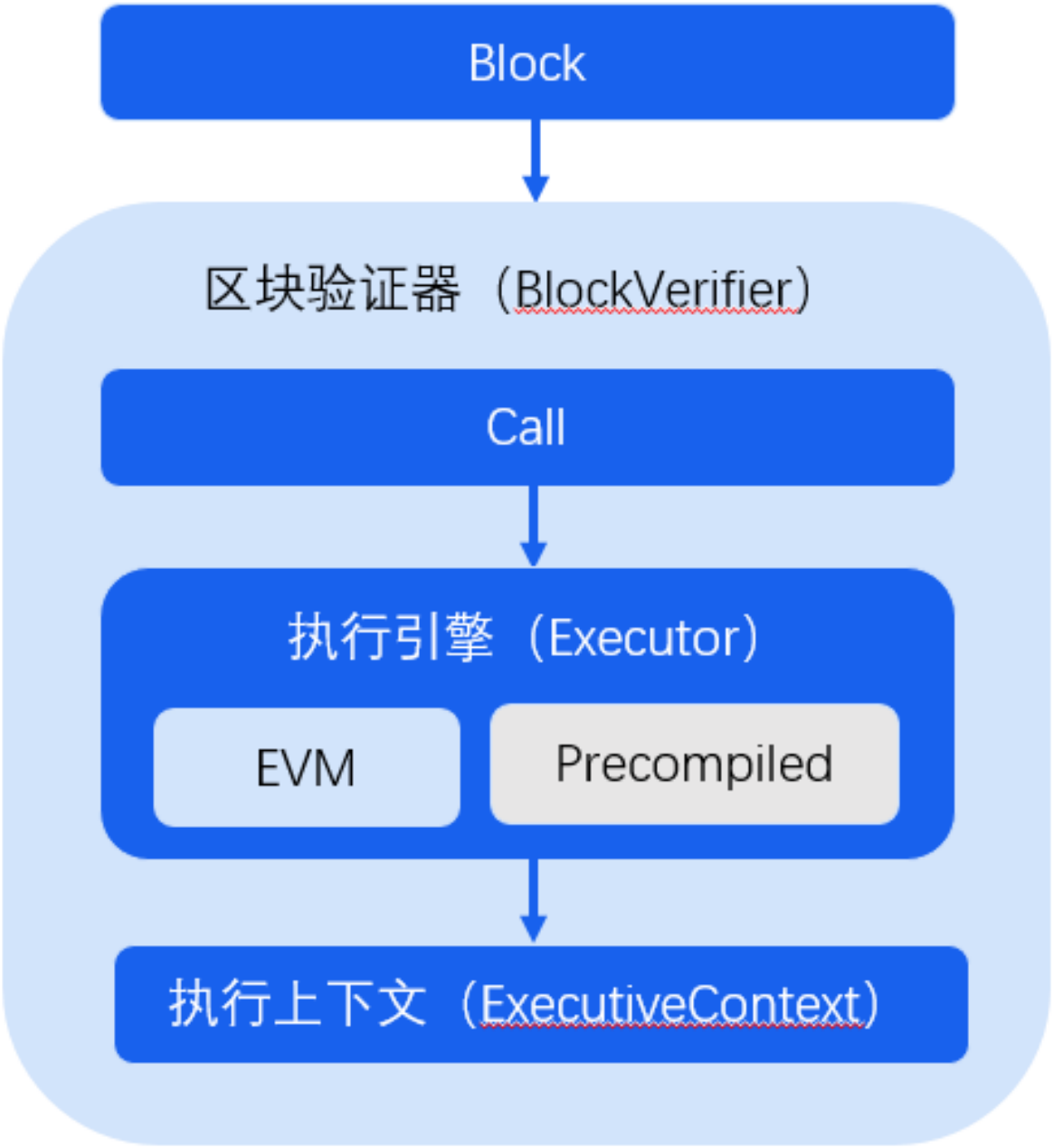
功能特性	FISCO BCOS 预编译合约	以太坊 内置合约
Solidity内调用	√	√
native执行	√	√
通过交易调用	√	×
访问存储	√	×
保存状态	√	×
复杂类型参数	√	×
动态创建临时预编译合约对象	√	×
单个预编译合约多接口	√	×

注：√代表支持，× 代表不支持

8.3.3 FISCO BCOS预编译合约架构

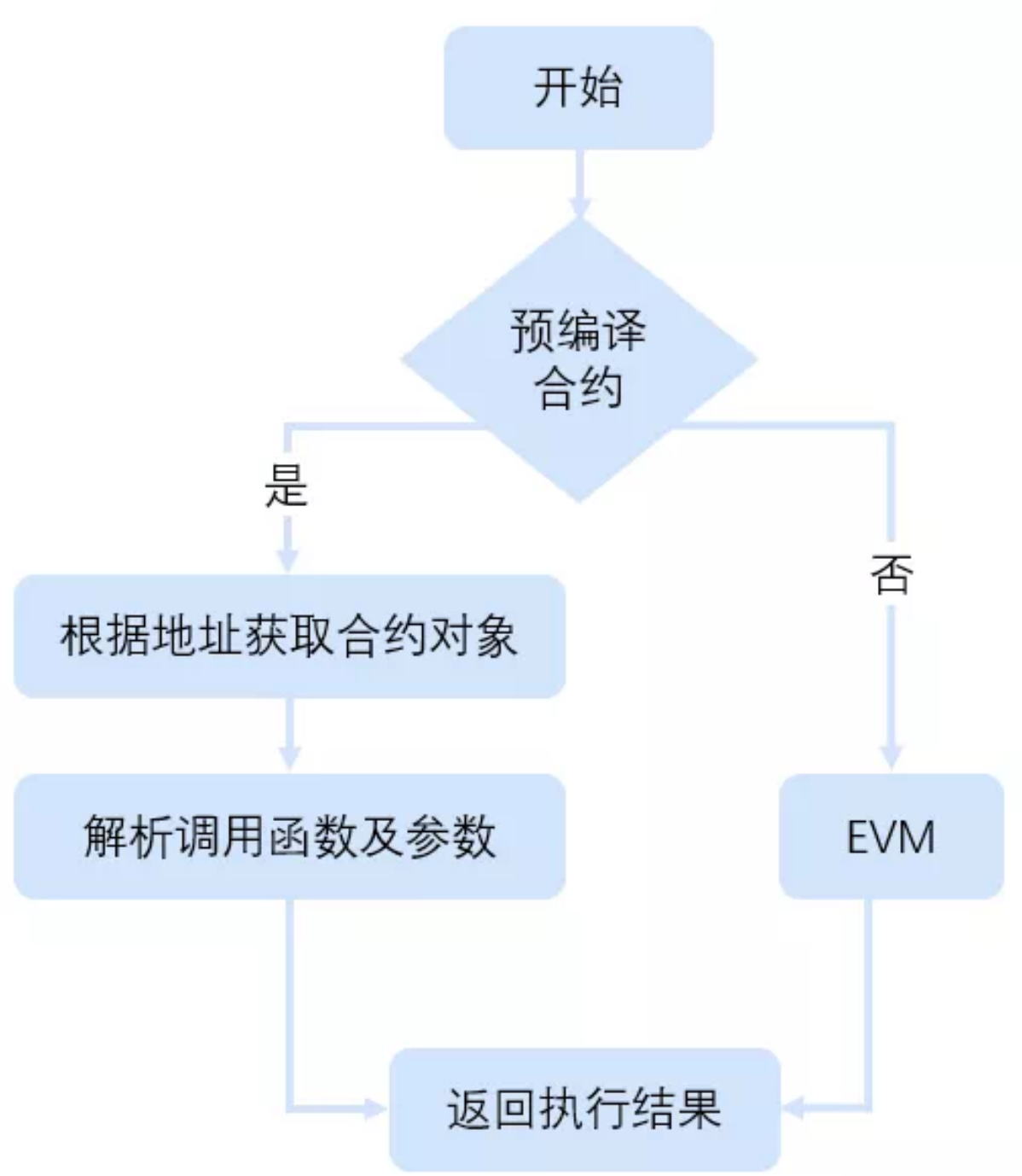
通过这一小节，你可以清楚了解预编译合约模块在FISCO BCOS中的位置，以及预编译合约的执行流程。

如下图所示，预编译合约会被区块执行引擎所调用，区块验证器通过区块执行引擎来执行区块，执行引擎执行区块时，会根据被调用合约的地址，来判断使用EVM还是预编译合约引擎。



当被调用的合约地址是EVM合约时，执行引擎会创建并执行EVM来执行交易；当被调用合约地址是已注册的预编译合约地址时，执行引擎通过调用地址对应的预编译合约接口来执行交易。

预编译合约执行流程如下图所示：



执行引擎首先根据预编译合约地址拿到合约对象，然后通过调用合约对象的call接口来获取执行结果。call接口中的操作主要包括：

1. 根据调用参数解析出被调用的接口
2. 根据ABI编码解析传入的参数
3. 执行被调用的合约接口
4. 将执行结果ABI编码并返回

所以，开发者如果要开发预编译合约，只需要实现其预编译合约的call接口和在执行引擎中注册所实现合约的地址即可。

8.3.4 预编译合约在FISCO BCOS 2.0中的应用

- **联盟链治理**：节点管理、系统配置管理、权限管理、CNS实现等系统合约。通过预编译合约形式实现这些功能，方便用户实现对联盟链的治理。
- **扩展Solidity能力**：KVTable合约接口、Table合约接口。使得Solidity合约中数据可以存储在FISCO的表结构中，数据逻辑分离，更容易实现合约逻辑升级。
- **支持Solidity并行**：Solidity并行合约接口，借助ParallelConfig预编译合约，使得Solidity合约接口并行成为可能。
- **提升SDK易用性，降低开发门槛**：基于CRUD预编译合约，SDK可实现CRUDService，提供类似传统数据库增删改查接口。
- **提供密码学算法**：基于预编译合约提供群签名校验、环签名校验、同态加等密码学算法功能

FISCO BCOS当前系统合约及地址如下表：

8.3.5 预编译合约接口描述与SDK支持

SystemConfigPrecompiled-0x1000

接口声明

```
pragma solidity ^0.4.24;

contract SystemConfigPrecompiled
{
    function setValueByKey(string key, string value) public returns(int256);
}
```

setValueByKey说明

- key表示配置项名称，当前支持的参数有tx_count_limit,tx_gas_limit,rpbft_epoch_sealer_num,rpbft
- value表示对应配置项的值，其中tx_count_limit默认值为1000，不可设置为负数，tx_gas_limit默认值为300000000，不可设置为负数。

SDK支持

- Java SDK
- Nodejs SDK
- Python SDK
- Go SDK

TableFactoryPrecompiled-0x1001

接口声明

完整solidity请参考这里

```
pragma solidity>=0.4.24 <0.6.11;

contract TableFactory {
    function openTable(string tableName) public view returns (Table); //open table
    function createTable(string tableName, string keyField, string valueFields)␣
↪public returns (int256); //create table
}
```

openTable说明

- 根据入参Table返回Solidity中的Table对象

createTable说明

- 根据入参创建表，其中valueFields可以有多个字段，使用英文逗号分割。
- 表名允许字母、数字、下划线，表名不超48字符
- keyField不能以下划线开始，允许字母、数字、下划线，总长度不能超过64字符
- valueField不能以下划线开始，允许字母、数字、下划线，单字段名不超过64字符， valueFields总长度不超过1024
- valueFields与keyField不能存在重复字段

CRUDPrecompiled-0x1002

接口声明

```
pragma solidity ^0.4.24;

contract CRUDPrecompiled {
    function insert(string tableName, string key, string entry, string) public␣
↪returns (int256);
    function remove(string tableName, string key, string condition, string) public␣
↪returns (int256);
    function select(string tableName, string key, string condition, string)
        public
        view
        returns (string);
    function update(
        string tableName,
        string key,
        string entry,
        string condition,
        string
    ) public returns (int256);
    function desc(string tableName) public view returns (string, string);
}
```

接口说明

- Insert接口往指定的Table的key中插入一条记录，entry是json编码，需要在sdk处理
- Remove接口删除指定Table中key下的一条记录
- Update接口，根据condition和entry更新对应Table中key下的记录

- Select接口根据condition查询对应Table中key下的记录
- Desc接口，查询对应Table的字段名

SDK支持

- Java SDK
- Nodejs SDK
- Python SDK
- Go SDK

ConsensusPrecompiled-0x1003

接口声明

```
pragma solidity ^0.4.24;

contract ConsensusPrecompiled {
    function addSealer(string) public returns (int256);
    function addObserver(string) public returns (int256);
    function remove(string) public returns (int256);
}
```

接口说明

- addSealer添加一个共识节点，参数是新节点公钥的16进制表示
- addObserver添加一个观察节点或将已经存在的共识节点身份改为观察节点
- Remove删除某个节点，如果是最后一个共识节点则不允许删除
- 数据存放在_sys_consensus_表中

SDK支持

- Java SDK
- Nodejs SDK
- Python SDK
- Go SDK

CNSPrecompiled-0x1004

接口声明

```
pragma solidity ^0.4.24;

contract CNSPrecompiled
{
    function insert(string name, string version, string addr, string abi) public␣
↪returns(uint256);
    function selectByName(string name) public constant returns(string);
```

(continues on next page)

(续上页)

```
    function selectByNameAndVersion(string name, string version) public constant␣
↪returns(string);
    function getContractAddress(string name, string version) public constant␣
↪returns(address);
}
```

接口说明

- Insert插入了合约版本、地址和abi
- selectByName返回该合约所有版本的版本、地址、abi的json
- selectByNameAndVersion根据合约名和版本号返回对应地址、abi的json
- getContractAddress根据合约名和版本号返回合约地址
- version不超128字符，address不超256字符，abi不超16MB

SDK支持

- Java SDK
- Nodejs SDK
- Python SDK
- Go SDK

PermissionPrecompiled-0x1005

接口声明

```
pragma solidity ^0.4.24;

contract PermissionPrecompiled {
    function insert(string table_name, string addr) public returns (int256);
    function remove(string table_name, string addr) public returns (int256);
    function queryByName(string table_name) public view returns (string);

    function grantWrite(address contractAddr, address user)
        public
        returns (int256);
    function revokeWrite(address contractAddr, address user)
        public
        returns (int256);
    function queryPermission(address contractAddr) public view returns (string);
}
```

接口说明

- Insert为某个表添加一条写权限。可以对同一个表插入多个账户有写权限。
- Remove删除某个表的一个账户的写权限。
- queryByName返回有表写权限的地址等信息的json，sdk需要解析

SDK支持

- Java SDK
- Nodejs SDK
- Python SDK
- Go SDK

ParallelConfigPrecompiled-0x1006

接口声明

```
pragma solidity ^0.4.24;

contract ParallelConfigPrecompiled {
    function registerParallelFunctionInternal(address, string, uint256)
        public
        returns (int256);
    function unregisterParallelFunctionInternal(address, string)
        public
        returns (int256);
}
```

接口说明

- registerParallelFunctionInternal注册合约的并行接口信息，参数为合约地址，并行函数签名、互斥参数个数。并行函数的互斥参数必须放在不互斥参数之前
- unregisterParallelFunctionInternal删除某个函数的并行设置

ContractLifeCyclePrecompiled-0x1007

接口声明

```
pragma solidity ^0.4.24;

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(int,string);
    function listManager(address addr) public constant returns(int,address[]);
}
```

接口说明

- Kill将某个合约置为删除状态，不可恢复
- Freeze冻结合约
- Unfree解冻
- queryStatus查询合约状态

SDK支持

- Java SDK
- Go SDK

ChainGovernancePrecompiled-0x1008

接口声明

```
pragma solidity ^0.4.24;


contract ChainGovernancePrecompiled {
    function grantCommitteeMember(address user) public returns (int256);

    function revokeCommitteeMember(address user) public returns (int256);

    function listCommitteeMembers() public view returns (string);

    function queryCommitteeMemberWeight(address user)
        public
        view
        returns (bool, int256);

    function updateCommitteeMemberWeight(address user, int256 weight)
        public
        returns (int256);
    function queryVotesOfMember(address member) public view returns (string);
    function queryVotesOfThreshold() public view returns (string);
    // threshold [0,100)
    function updateThreshold(int256 threshold) public returns (int256);

    function queryThreshold() public view returns (int256);

    function grantOperator(address user) public returns (int256);

    function revokeOperator(address user) public returns (int256);

    function listOperators() public view returns (string);

    // account life cycle
    function freezeAccount(address account) public returns (int256);

    function unfreezeAccount(address account) public returns (int256);

    function getAccountStatus(address account) public view returns (string);
}
```

SDK支持

- Java SDK
- Go SDK

KVTableFactoryPrecompiled-0x1010

接口声明

```
pragma solidity>=0.4.24 <0.6.11;
//one record
contract Entry {
    function getInt(string memory) public view returns (int256) {}
    function getUInt(string memory) public view returns (uint256) {}
    function getAddress(string memory) public view returns (address) {}
    function getBytes64(string memory) public view returns (bytes1[64] memory) {}
    function getBytes32(string memory) public view returns (bytes32) {}
    function getString(string memory) public view returns (string memory) {}

    function set(string memory, int256) public {}
    function set(string memory, uint256) public {}
    function set(string memory, string memory) public {}
    function set(string memory, address) public {}
}

contract KVTableFactory {
    function openTable(string memory) public view returns (KVTable) {}
    function createTable(string memory, string memory, string memory) public␣
↪returns (int256) {}
}

//KVTable per permiary key has only one Entry
contract KVTable {
    function get(string memory) public view returns (bool, Entry) {}
    function set(string memory, Entry) public returns (int256) {}
    function newEntry() public view returns (Entry) {}
}
```

接口说明

- openTable 根据tableName返回一个solidity中的Table对象
- createTable 创建表，参数分别是表名、主键列名、以逗号分隔的其他列名。
- createTable 表名允许字母、数字、下划线，表名不超48字符
- keyField不能以下划线开始，允许字母、数字、下划线，总长度不能超过64字符
- valueField不能以下划线开始，允许字母、数字、下划线，单字段名不超过64字符， valueFields总长度不超过1024
- valueFields与keyField不能存在重复字段

8.4 Java SDK

标签： `java-sdk` 区块链应用

Java SDK 提供了访问 FISCO BCOS 节点的Java API，支持节点状态查询、部署和调用合约等功能，基于Java SDK可开发区块链应用，目前支持FISCO BCOS 2.0+。

主要特性

- 提供 合约编译 ， 将Solidity合约文件转换成Java合约文件

- 提供 Java SDK API,提供访问FISCO BCOS JSON-RPC 的功能，并支持预编译（Precompiled）合约调用
- 提供 自定义构造和发送交易功能
- 提供 AMOP 功能
- 支持 事件推送
- 支持 ABI解析
- 提供 账户管理 接口

8.4.1 快速入门

标签：`java-sdk` 引入Java SDK

---

1. 安装环境

- Java：JDK 14 （JDK1.8 至JDK 14都支持）

  首先，在官网上下载JDK14并安装

  然后，修改环境变量

```
# 确认您当前的java版本
$ java -version
# 确认您的java路径
$ ls Library/Java/JavaVirtualMachines
# 返回
# jdk-14.0.2.jdk

# 如果使用的是bash
$ vim .bash_profile
# 在文件中加入JAVA_HOME的路径
# export JAVA_HOME = Library/Java/JavaVirtualMachines/jdk-14.0.2.jdk/Contents/
↪Home
$ source .bash_profile

# 如果使用的是zash
$ vim .zashrc
# 在文件中加入JAVA_HOME的路径
# export JAVA_HOME = Library/Java/JavaVirtualMachines/jdk-14.0.2.jdk/Contents/
↪Home
$ source .zashrc

# 确认您的java版本
$ java -version
# 返回
# java version "14.0.2" 2020-07-14
# Java(TM) SE Runtime Environment (build 14.0.2+12-46)
# Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing)
```

- IDE：IntelliJ IDE.

  进入IntelliJ IDE官网，下载并安装社区版IntelliJ IDE

2. 搭建一条FISCO BCOS链

请参考FISCO BCOS安装搭建。

3. 开发智能合约应用

第一步. 创建一个Gradle应用

在IntelliJ IDE中创建一个gradle项目。勾选Gradle和Java

第二步. 引入Java SDK

在build.gradle中引入Java SDK

```
compile ('org.fisco-bcos.java-sdk:fisco-bcos-java-sdk:2.7.1')
```

如果您使用maven 通过以下方法引入Java SDK

```
<dependency>
    <groupId>org.fisco-bcos.java-sdk</groupId>
    <artifactId>fisco-bcos-java-sdk</artifactId>
    <version>2.7.1</version>
</dependency>
```

第三步. 配置SDK证书

参考java sdk证书配置。

注解:

- 大部分场景仅需要配置 *certPath* 配置项即可，其他配置项不需额外配置；
- SDK证书获取：若参考 安装 搭建区块链，则参考 这里 将 *nodes/${ip}/sdk/* 目录下的证书拷贝到 *certPath* 指定的路径；若区块链节点参考 运维部署工具 搭建，则参考 这里 将 *generator/meta* 文件夹下的SDK证书拷贝到 *certPath* 指定路径，‘certPath‘默认为‘conf‘目录；
- SDK与节点间SSL连接方式，可通过节点配置项 *sm_crypto_channel* 判断，该配置项详细说明请参考 FISCO BCOS配置文件与配置项说明 .

将SDK证书拷贝到java sdk的示例如下(这里假设SDK证书位于~/fisco/nodes/127.0.0.1/sdk目录):

```
# 假设SDK证书位于~/fisco/nodes/127.0.0.1/sdk/目录
mkdir -p conf && cp -r ~/fisco/nodes/127.0.0.1/sdk/* conf
```

第四步. 准备智能合约

控制台console和java-sdk-demo均提供了工具，可以将solidity合约生成出调用该合约java工具类。本例中使用console做为例子，使用java-sdk-demo的例子请看第6章“附录一. 使用java-sdk-demo给智能合约生成调用它的Java工具类” **首先，下载控制台。**

```
$ mkdir -p ~/fisco && cd ~/fisco
# 获取控制台
$ curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/
↪download_console.sh && bash download_console.sh
$ cd ~/fisco/console
```

然后，将您要用到的Solidity智能合约放入~/fisco/console/contracts/solidity的目录。本次我们用console中的HelloWorld.sol作为例子。保证HelloWorld.sol在指定的目录下。

```
# 当前目录~/fisco/console
$ ls contracts/solidity
```

得到返回

```
HelloWorld.sol        KVTableTest.sol        ShaTest.sol        Table.
↪sol        TableTest.sol
```

接着，生成调用该智能合约的java类

```
# 使用sol2java.sh将contracts/solidity下的所有合约编译产生bin,abi,java工具类。
# 当前目录~/fisco/console
$ bash sol2java.sh org.com.fisco
# 以上命令中参数"org.com.fisco"是指定产生的java类所属的包名。
# ./sol2java.sh [packageName] [solidityFilePath] [javaCodeOutputDir]
```

sol2java.sh的用法可以参照第4章 "附录二. sol2java.sh脚本的使用方法"。

得到返回

```
*** Compile solidity TableTest.sol***
INFO: Compile for solidity TableTest.sol success.
*** Convert solidity to java  for TableTest.sol success ***

*** Compile solidity KVTableTest.sol***
INFO: Compile for solidity KVTableTest.sol success.
*** Convert solidity to java  for KVTableTest.sol success ***

*** Compile solidity HelloWorld.sol***
INFO: Compile for solidity HelloWorld.sol success.
*** Convert solidity to java  for HelloWorld.sol success ***

*** Compile solidity Table.sol***
INFO: Compile for solidity Table.sol success.
*** Convert solidity to java  for Table.sol success ***

*** Compile solidity ShaTest.sol***
INFO: Compile for solidity ShaTest.sol success.
*** Convert solidity to java  for ShaTest.sol success ***
```

查看编译结果

```
$ ls contracts/sdk/java/org/com/fisco
# 得到返回
# HelloWorld.java                KVTableTest.java        ShaTest.
↪java                Table.java                TableTest.java
```

最后, 将编译得到的HelloWorld.java放入应用中。注意：在应用中所放的位置要与我们设定的包名相同。

(操作示范请看如下gif动图，动画总共有2分40秒，请耐心等待观看，请勿点击图片，如果点击图片将从头开始播放。)

第五步. 创建配置文件

在项目中创建配置文件config.toml, 可参照配置向导进行配置，也可以参照config-example.toml,

通过xml配置请参照第4章“附录三. 使用xml配置进行配置”。

第六步. 使用Java SDK部署和调用智能合约

以使用java sdk调用群组1的getBlockNumber接口获取群组1最新块高，并向群组1部署和调用HelloWorld合约为例，对应的示例代码如下:

```
public class BcosSDKTest
{
    // 获取配置文件路径
    public final String configFile = BcosSDKTest.class.getClassLoader().
↪getResource("config-example.toml").getPath();
     public void testClient() throws ConfigException {
         // 初始化BcosSDK
        BcosSDK sdk =  BcosSDK.build(configFile);
        // 为群组1初始化client
        Client client = sdk.getClient(Integer.valueOf(1));

        // 获取群组1的块高
        BlockNumber blockNumber = client.getBlockNumber();

        // 向群组1部署HelloWorld合约
        CryptoKeyPair cryptoKeyPair = client.getCryptoSuite().getCryptoKeyPair();
        HelloWorld helloWorld = HelloWorld.deploy(client, cryptoKeyPair);

        // 调用HelloWorld合约的get接口
        String getValue = helloWorld.get();

        // 调用HelloWorld合约的set接口
        TransactionReceipt receipt = helloWorld.set("Hello, fisco");
     }
}
```

4. 附录

附录一. 使用java-sdk-demo给智能合约生成调用它的Java工具类

```
$ mkdir -p ~/fisco && cd ~/fisco
# 获取java-sdk代码
$ git clone https://github.com/FISCO-BCOS/java-sdk-demo
$ cd java-sdk-demo
# 编译
$ ./gradlew clean build -x test
# 进入sdk-demo/dist目录，创建合约存放目录
$ cd dist && mkdir -p contracts/solidity
# 将需要转换为java代码的sol文件拷贝到~/fisco/java-sdk/dist/contracts/solidity路径下
# 转换sol, 其中${packageName}是生成的java代码包路径
# 生成的java代码位于 ~/fisco/java-sdk/dist/contracts/sdk/java目录下
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.codegen.DemoSolcToJava ${
↪packageName}
```

附录二. sol2java.sh脚本的使用方法

控制台v2.6+提供了sol2java.sh脚本可将solidity转换为java代码, sol2java.sh使用方法如下:

```
$ bash sol2java.sh -h
# Compile Solidity Tool
./sol2java.sh [packageName] [solidityFilePath] [javaCodeOutputDir]
          packageName:
                  the package name of the generated Java class file
          solidityFilePath:
                  (optional) the solidity file path or the directory where␣
↪solidity files located, default: contracts/solidity
          javaCodeOutputDir:
                  (optional) the directory where the generated Java files located,␣
↪default: contracts/sdk/java
```

参数如下:

- packageName: 生成Java文件的包名
- solidityFilePath: (可选)solidity文件的路径，支持文件路径和目录路径两种方式，参数为目录时将目录下所有的solidity文件进行编译转换。默认目录为contracts/solidity。
- javaCodeOutputDir: (可选)生成Java文件的目录，默认生成在contracts/sdk/java目录

附录三. 使用xml配置进行配置

为了适配更多场景，java sdk支持使用xml初始化BcosSDK, xml配置示例请参考java sdk源码的applicationContext-sample.xml, 配置项的含义参考配置说明.

通过xml配置文件初始化BcosSDK之前，需要先引入spring。

通过gradle引入spring如下:

```
def spring_version = "4.3.27.RELEASE"
List spring = [
        "org.springframework:spring-core:$spring_version",
        "org.springframework:spring-beans:$spring_version",
        "org.springframework:spring-context:$spring_version",
        "org.springframework:spring-tx:$spring_version",
]
compile spring
```

通过maven引入spring如下:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.3.27.RELEASE</version>

    <groupId>org.springframework</groupId>
    <artifactId>spring-beans</artifactId>
    <version>4.3.27.RELEASE</version>

    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.27.RELEASE</version>

    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
```

(continues on next page)

(续上页)

```
    <version>4.3.27.RELEASE</version>
</dependency>
```

使用applicationContext-sample初始化BcosSDK如下：

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("classpath:applicationContext-sample.xml");
BcosSDK sdk = context.getBean(BcosSDK.class);
```

8.4.2 配置说明

标签：java-sdk 配置

Java sdk主要包括五个配置选项，分别是

- 证书配置（必须）
- 网络连接配置 （必须）
- AMOP配置（非必须）
- 账户配置（非必须，不配置则使用默认配置值）
- 线程池配置（非必须，不配置则使用默认配置值）

1. 快速配置

配置步骤：

1. 在您的应用的主目录下新建一个conf目录。
2. 从节点nodes/${ip}/sdk/ 目录下的证书拷贝到新建的conf目录。
3. 将配置文件config-example.toml, 存放在应用的主目录下。
   - config-example.toml可以在java-sdk的源文件以下位置找到：src/test/resources/config-example.toml
   - 本文的“3. 配置示例” 部分也可以看到config-example.toml的内容。
4. 修改config-example.toml中节点的IP和端口，与您要连接的节点所匹配。

   ```
   [network]
   peers=["127.0.0.1:20200", "127.0.0.1:20201"]
   ```

5. 在您的应用中使用该配置文件初始化Java SDK，您就可以开发区块链应用了。

   ```
   String configFile = "config-example.toml";
   BcosSDK sdk =  BcosSDK.build(configFile);
   ```

关于如何在业务项目中引入以及配置Java SDK的进一步详细说明，可参考例子。对一项目，完成Java SDK引入、配置文件及证书放置、编译后，项目的dist目录结构如下：

```
├── lib
│   ├── fisco-bcos-java-sdk-2.7.0.jar
│   └── XXXXX.jar
├── conf
│   ├── applicationContext.xml
│   ├── conf
```

(continues on next page)

(续上页)

```
|   |   ├── node.crt
|   |   ├── ca.crt
|   |   ├── sdk.publickey
|   |   ├── sdk.key
|   |   ├── node.key
|   |   └── sdk.crt
|   └── log4j.properties
├── apps
|   └── XXXX.jar
└── other folders
```

2. 配置解读

Java sdk主要包括五个配置选项，分别是

- 证书配置（必须）
- 网络连接配置 （必须）
- AMOP配置（非必须）
- 账户配置（非必须，不配置则使用默认配置值）
- 线程池配置（非必须，不配置则使用默认配置值）

证书配置

基于安全考虑，java sdk与节点间采用SSL加密通信，目前同时支持非国密SSL连接以及国密SSL连接，`[cryptoMaterial]`配置SSL连接的证书信息，具体包括如下配置项：

- `certPath`: 证书存放路径，默认是`conf`目录；
- `caCert`: CA证书路径，默认注释该配置项，该配置项注释时，当SDK与节点间采用非国密SSL连接时，默认的CA证书路径为`${certPath}/ca.crt`，当SDK与节点采用国密SSL连接时，默认的CA证书路径为`${certPath}/gm/gmca.crt`；开启该配置项时，从配置指定的路径加载CA证书；
- `sslCert`: SDK证书路径，默认注释该配置项，该配置项注释时，当SDK与节点间采用非国密SSL连接时，从`${certPath}/sdk.crt`加载SDK证书，当SDK与节点间采用国密SSL连接时，从`${certPath}/gm/gmsdk.crt`加载SDK证书；开启该配置选项时，从配置指定的路径加载SDK证书；
- `sslKey`: SDK私钥路径，默认注释该配置项，当该配置项注释时，当SDK与节点间采用非国密SSL连接时，从`${certPath}/sdk.key`加载SDK私钥，SDK与节点采用国密SSL连接时，从`${certPaht}/gm/gmsdk.key`加载SDK私钥；开启该配置项时，直接从配置项指定的路径加载SDK私钥；
- `enSslCert`: 国密SSL加密证书路径，仅当SDK与节点间采用国密SSL连接时，需要配置该配置项，默认从`${certPath}/gm/gmensdk.crt`加载国密SSL加密证书；当开启该配置项时，从配置项指定的路径加载国密SSL加密证书；
- `enSslKey`: 国密SSL加密私钥路径，仅当SDK与节点间采用国密SSL连接时，需配置该配置项，默认从`${certPath}/gm/gmensdk.key`加载国密SSL加密私钥；当卡其该配置项时，从配置项指定的路径加载国密SSL加密私钥。

注解:

- 大部分场景仅需要配置 *certPath* 配置项即可，其他配置项不需额外配置；

- SDK证书获取：若参考 安装 搭建区块链，则参考 这里 将 *nodes/${ip}/sdk/* 目录下的证书拷贝到 *certPath* 指定的路径；若区块链节点参考 运维部署工具 搭建，则参考 这里 将 *generator/meta* 文件夹下的SDK证书拷贝到 ‘certPath‘指定路径；
- SDK与节点间SSL连接方式，可通过节点配置项 *sm_crypto_channel* 判断，该配置项详细说明请参考 FISCO BCOS配置文件与配置项说明 .

SDK证书配置示例如下：

```
[cryptoMaterial]

certPath = "conf"                               # The certification path

# The following configurations take the certPath by default if commented
# caCert = "conf/ca.crt"                        # CA cert file path
                                                # If connect to the GM node, default␣
↪CA cert path is ${certPath}/gm/gmca.crt

# sslCert = "conf/sdk.crt"                      # SSL cert file path
                                                # If connect to the GM node, the␣
↪default SDK cert path is ${certPath}/gm/gmsdk.crt

# sslKey = "conf/sdk.key"                       # SSL key file path
                                                # If connect to the GM node, the␣
↪default SDK privateKey path is ${certPath}/gm/gmsdk.key

# enSslCert = "conf/gm/gmensdk.crt"             # GM encryption cert file path
                                                # default load the GM SSL encryption␣
↪cert from ${certPath}/gm/gmensdk.crt

# enSslKey = "conf/gm/gmensdk.key"              # GM ssl cert file path
                                                # default load the GM SSL encryption␣
↪privateKey from ${certPath}/gm/gmensdk.key
```

网络连接配置

SDK与FISCO BCOS节点通信，必须配置SDK连接的节点的`IP`和`Port`，`[network]`配置了java sdk连接的节点信息，具体包括如下配置项：

- peers：SDK连接的节点的`IP:Port`信息，可配置多个连接。

注解： 节点与网络之间的连接信息 SDK与节点间通过 *ChannelServer* 进行通信，SDK需要连接 *ChannelServer* 的监听端口，该端口可通过节点 *config.ini* 的 *rpc.channel_listen_port* 获取，具体请参考 这里

SDK与节点间的网络配置示例如下：

```
[network]
peers=["127.0.0.1:20200", "127.0.0.1:20201"]    # The peer list to connect
```

AMOP配置

AMOP支持私有话题的功能，配置文件中提供了`AMOP`相关配置项于`[[amop]]`中。

私有话题订阅配置

AMOP私有话题订阅者需要配置私钥用于进行私有话题认证，具体配置项包括：

- topicName: 私有话题名称；
- privateKey: 私有话题订阅者的私钥路径，用于证明订阅房身份信息；
- password: 访问私钥文件的口令。

AMOP订阅私有话题的配置项示例如下：

```
# Configure a private topic as a topic subscriber.
[[amop]]
topicName = "PrivateTopic"
privateKey = "conf/amop/consumer_private_key.p12"          # Your private key that␣
↪used to subscriber verification.
password = "123456"
```

私有话题消息发布配置

AMOP私有话题认证成功后，消息发布方可向订阅方发送私有话题消息，发布私有话题消息的配置包括：

- topicName: 私有话题名称；
- publicKeys: 消息订阅方的公钥列表。

AMOP发布私有话题消息的配置示例如下：

```
# Configure a private topic as a topic message sender.
[[amop]]
topicName = "PrivateTopic"
publicKeys = [ "conf/amop/consumer_public_key_1.pem" ]     # Public keys of the␣
↪nodes that you want to send AMOP message of this topic to.
```

账户配置

账户配置主要用于设置SDK向节点发交易的账户信息，SDK初始化client时，默认读取[account]配置项加载账户信息，具体如下：

- keyStoreDir: 加载/保存账户文件的路径，默认为account；
- accountFileFormat: 账户文件格式，默认为pem，目前仅支持pem和p12，pem格式的账户文件不需要口令加载，加载p12格式的账户文件时需要口令；
- accountAddress: 加载的账户地址，默认为空
- accountFilePath: 加载的账户文件路径，默认注释该配置项，注释该配置项时，当SDK连接非国密区块链时，默认从${keyStoreDir}/ecdsa/${accountAddress}.${accountFileFormat}路径加载账户文件，当SDK连接国密区块链时，默认从${keyStoreDir}/gm/${accountAddress}.${accountFileFormat}路径加载账户；当开启该配置项时，从该配置项指定的目录加载账户；
- password: 加载p12类型账户文件的口令。

注解： 当没有配置 *accountAddress* 和 *accountFilePath* 时，SDK会生成随机的账户向节点交易，生成的账户信息均保存在 *keyStoreDir* 配置项指定的目录下：当SDK连接节点是非国密节点时，生成的临时账户以 *pem* 的格式保存在 *${keyStoreDir}/ecdsa/* 目录下；当SDK连接的节点时国密节点时，生成的临时账户以 *p12* 的格式保存在 *${keyStoreDir}/gm* 目录下

账户配置文件示例如下：

```
[account]
keyStoreDir = "account"          # The directory to load/store the account file,␣
↪default is "account"
# accountFilePath = ""           # The account file path (default load from the␣
↪path specified by the keyStoreDir)
accountFileFormat = "pem"        # The storage format of account file (Default is
↪"pem", "p12" as an option)

# accountAddress = ""            # The transactions sending account address
                                 # Default is a randomly generated account
                                 # The randomly generated account is stored in the␣
↪path specified by the keyStoreDir

# password = ""                  # The password used to load the account file
```

线程池配置

为了方便业务根据机器实际负载调整SDK的处理线程，java sdk将其线程配置项暴露在配置中，[threadPool]是线程池相关配置，具体包括：

- channelProcessorThreadSize: 处理网络回调的线程数目，默认注释该配置项，注释该配置项时，其默认值为机器的CPU数目；开启该配置项时，根据配置的值创建处理网络回调的线程数目；
- receiptProcessorThreadSize: 接收交易的线程数目，默认注释该配置项，注释该配置项时，默认值为机器的CPU数目；开启该配置项时，根据配置的值创建接收交易的线程数目；
- maxBlockingQueueSize: 线程池队列等待被处理的最大任务数目，默认为102400。

注解： 大多数场景下，不需要手工配置线程池配置；压测场景下，可将 *maxBlockingQueueSize* 配置大一些。

线程池配置示例如下：

```
[threadPool]
# channelProcessorThreadSize = "16"          # The size of the thread pool to␣
↪process channel callback
                                             # Default is the number of cpu cores

# receiptProcessorThreadSize = "16"          # The size of the thread pool to␣
↪process transaction receipt notification
                                             # Default is the number of cpu cores

maxBlockingQueueSize = "102400"              # The max blocking queue size of the␣
↪thread pool
```

3. 配置示例

config-example.toml

```
[cryptoMaterial]

certPath = "conf"                            # The certification path

# The following configurations take the certPath by default if commented
# caCert = "conf/ca.crt"                     # CA cert file path
                                             # If connect to the GM node, default␣
↪CA cert path is ${certPath}/gm/gmca.crt
```

(continues on next page)

(续上页)

```
# sslCert = "conf/sdk.crt"                    # SSL cert file path
                                              # If connect to the GM node, the␣
↪default SDK cert path is ${certPath}/gm/gmsdk.crt

# sslKey = "conf/sdk.key"                     # SSL key file path
                                              # If connect to the GM node, the␣
↪default SDK privateKey path is ${certPath}/gm/gmsdk.key

# enSslCert = "conf/gm/gmensdk.crt"           # GM encryption cert file path
                                              # default load the GM SSL encryption␣
↪cert from ${certPath}/gm/gmensdk.crt

# enSslKey = "conf/gm/gmensdk.key"            # GM ssl cert file path
                                              # default load the GM SSL encryption␣
↪privateKey from ${certPath}/gm/gmensdk.key

[network]
peers=["127.0.0.1:20200", "127.0.0.1:20201"]     # The peer list to connect

# AMOP configuration
# You can use following two methods to configure as a private topic message sender␣
↪or subscriber.
# Usually, the public key and private key is generated by subscriber.
# Message sender receive public key from topic subscriber then make configuration.
# But, please do not config as both the message sender and the subscriber of one␣
↪private topic, or you may send the message to yourself.

# Configure a private topic as a topic message sender.
# [[amop]]
# topicName = "PrivateTopic"
# publicKeys = [ "conf/amop/consumer_public_key_1.pem" ]    # Public keys of the␣
↪nodes that you want to send AMOP message of this topic to.

# Configure a private topic as a topic subscriber.
# [[amop]]
# topicName = "PrivateTopic"
# privateKey = "conf/amop/consumer_private_key.p12"         # Your private key␣
↪that used to subscriber verification.
# password = "123456"


[account]
keyStoreDir = "account"          # The directory to load/store the account file,␣
↪default is "account"
# accountFilePath = ""           # The account file path (default load from the␣
↪path specified by the keyStoreDir)
accountFileFormat = "pem"        # The storage format of account file (Default is␣
↪"pem", "p12" as an option)

# accountAddress = ""            # The transactions sending account address
                                 # Default is a randomly generated account
                                 # The randomly generated account is stored in the␣
↪path specified by the keyStoreDir

# password = ""                  # The password used to load the account file

[threadPool]
# channelProcessorThreadSize = "16"         # The size of the thread pool to␣
↪process channel callback
                                            # Default is the number of cpu cores
```

(continues on next page)

(续上页)

```
# receiptProcessorThreadSize = "16"          # The size of the thread pool to
↪process transaction receipt notification
                                             # Default is the number of cpu cores

maxBlockingQueueSize = "102400"              # The max blocking queue size of the
↪thread pool
```

8.4.3 交易构造与发送

标签：java-sdk 发送交易

注解： java sdk同时支持将 *solidity* 转换为 *java* 文件后，调用相应的 *java* 方法部署和调用合约，也支持构造交易的方式部署和调用合约，这里主要展示交易构造与发送，前者的使用方法请参考 这里

---

1. 准备abi和binary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合约文件编译生成Java文件和abi、binary文件，具体使用方式参考这里。

通过运行sol2java.sh脚本，生成的abi和binary文件分别位于contracts/sdk/abi、contracts/sdk/bin目录下（其中，国密版本编译产生的文件位于contracts/sdk/abi/sm和contracts/sdk/bin/sm文件夹下）。可将文件复制到项目的目录下，例如src/main/resources/abi和src/main/resources/bin。

为了便于演示，我们使用了以下HelloWorld的合约。

```
pragma solidity ^0.4.25;

contract HelloWorld{
    string public name;
    constructor() public{
       name = "Hello, World!";
    }

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

2. 部署并调用合约

Java SDK提供了基于abi和binary文件来直接部署和调用合约的方式。可以使用AssembleTransactionProcessor来完成合约操作。

部署合约

部署合约调用了deployByContractLoader方法，传入合约名和构造函数的参数，上链部署合约，并获得TransactionResponse的结果。

```
    // 初始化BcosSDK对象
    BcosSDK sdk = new BcosSDK(configFile);
    // 获取Client对象，此处传入的群组ID为1
    Client client = sdk.getClient(Integer.valueOf(1));
    // 构造AssembleTransactionProcessor对象，需要传入client对象，CryptoKeyPair对象
和abi、binary文件存放的路径。abi和binary文件需要在上一步复制到定义的文件夹中。
    CryptoKeyPair keyPair = client.getCryptoSuite().createKeyPair();
    AssembleTransactionProcessor transactionProcessor =␣
↪TransactionProcessorFactory.createAssembleTransactionProcessor(client, keyPair,
↪"src/main/resources/abi/", "src/main/resources/bin/");
    // 部署HelloWorld合约。第一个参数为合约名称，第二个参数为合约构造函数的列表，是List
↪<Object>类型。
    TransactionResponse response = transactionProcessor.deployByContractLoader(
↪"HelloWorld", new ArrayList<>());
```

TransactionResponse的数据结构如下：

- returnCode: 返回的响应码。其中0为成功。
- returnMessages: 返回的错误信息。
- TransactionReceipt：上链返回的交易回执。
- ContractAddress: 部署或调用的合约地址。
- values: 如果调用的函数存在返回值，则返回解析后的交易返回值，返回Json格式的字符串。
- events: 如果有触发日志记录，则返回解析后的日志返回值，返回Json格式的字符串。
- receiptMessages: 返回解析后的交易回执信息。

例如，部署HelloWorld合约的返回结果：

```
{
  "returnCode": 0,
  "returnMessage": "Success",
  "transactionReceipt": {
    "transactionHash":
↪"0xcfdfb78be52b232afdee717826f9516af98fb2d67ee743da4b78e8c22172112b",
    "transactionIndex": "0x0",
    "root": "0xf6503b5f1a319dbd2c938d7e371b89441c238271bbaabf3d650112017158a658",
    "blockNumber": "0x14b",
    "blockHash":
↪"0x817e0aaaba5448a6ac62fc2531be793cf6de9fed70c73ed8837082b7fcf74881",
    "from": "0xcaa405b5dd47e7f28e6b862c198c15e923000c0b",
    "to": "0x0000000000000000000000000000000000000000",
    "gasUsed": "0x44683",
    "contractAddress": "0x9dbaf42da05a0148d2ca9905870a91085c23ce71",
    "logs": [],
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "status": "0x0",
    "input":
↪"0x6080604052348015610010576000080fd5b506040805190810160405280600d81526020017f48656c6c6f2c20576f
↪",
    "output": "0x",
    "txProof": null,
    "receiptProof": null,
    "message": null,
    "statusOK": true
  },
  "contractAddress": "0x9dbaf42da05a0148d2ca9905870a91085c23ce71",
  "values": null,
  "events": "{}",
```

(continues on next page)

(续上页)

```
  "receiptMessages": "Success"
}
```

调用合约

假如只调用合约，而不部署合约，那么就不需要复制binary文件，且在构造时无需传入binary文件的路径，例如构造方法的最后一个参数可传入空字符串。

```
    AssembleTransactionProcessor transactionProcessor =␣
↪TransactionProcessorFactory.createAssembleTransactionProcessor(client, keyPair,
↪"src/main/resources/abi/", "");
```

合约调用又可以被区分为『交易』和『查询』。被view修饰符修饰的方法一般称为“查询”，而未被修饰的才会称为“交易”。以下是“交易”和“查询”更详细的区别。

交易

调用合约交易使用了sendTransactionAndGetResponseByContractLoader来调用合约交易，此处展示了如何调用HelloWorld中的set函数。

```
    // 创建调用交易函数的参数，此处为传入一个参数
    List<Object> params = new ArrayList<>();
    params.add("test");
    // 调用HelloWorld合约，合约地址为helloWorldAddress， 调用函数名为『set』，函数参数类型
为params
    TransactionResponse transactionResponse = transactionProcessor.
↪sendTransactionAndGetResponseByContractLoader("HelloWorld", helloWorldAddrss,
↪"set", params);
```

例如，调用HelloWorld合约的返回如下：

```
{
  "returnCode": 0,
  "returnMessage": "Success",
  "transactionReceipt": {
    "transactionHash":
↪"0x46b510dfff02c327432362db76555c75211aab17b3b75221137132123773fe8f",
    "transactionIndex": "0x0",
    "root": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "blockNumber": "0x156",
    "blockHash":
↪"0x2d1ef0246bc0484e0fecd2a0b6671c19b69c86ac9e464b162ff4a932908151a0",
    "from": "0x77a5933b5af032a313fde655d8290c134aeeb0d5",
    "to": "0x975810f096e8d8b2daa9ee399f5ce809c0a12f1b",
    "gasUsed": "0x6032",
    "contractAddress": "0x0000000000000000000000000000000000000000",
    "logs": [],
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)

```
    "message": null,
    "statusOK": true
  },
  "contractAddress": "0x0000000000000000000000000000000000000000",
  "values": "[]",
  "events": "{}",
  "receiptMessages": "Success"
}
```

查询合约

查询合约使用了sendCallByContractLoader来查询合约，此处展示了如何调用HelloWorld中的name函数来进行查询。

```
    // 查询HelloWorld合约的『name』函数，合约地址为helloWorldAddress，参数为空
    CallResponse callResponse = transactionProcessor.sendCallByContractLoader(
↪"HelloWorld", helloWorldAddrss, "name", new ArrayList<>());
```

查询函数返回如下：

```
{
  "returnCode": 0,
  "returnMessage": "Success",
  "values": "[\"test\"]"
}
```

3. 详细API功能介绍

AssembleTransactionProcessor支持自定义参数发送交易，详细的API功能如下。

- **public void deployOnly(String abi, String bin, List<Object> params)**：传入合约abi、bin和构造函数参数来部署合约，不接收回执结果。
- **public TransactionResponse deployAndGetResponse(String abi, String bin, List<Object> params)**：传入合约abi、bin和构造函数参数来部署合约，接收回执结果
- **public TransactionResponse deployAndGetResponseWithStringParams(String abi, String bin, List<String> params)**：传入合约abi和String类型的List作为构造函数参数来部署合约，接收TransactionResponse结果。
- **public void deployAsync(String abi, String bin, List<Object> params, TransactionCallback callback)**：传入合约abi、构造好的合约构造函数和callback来异步部署合约
- **public CompletableFuture<TransactionReceipt> deployAsync(String abi, String bin, List<Object> params)**：传入合约abi、bin和构造函数参数来部署合约，接收CompletableFuture封装的回执结果
- **public TransactionResponse deployByContractLoader(String contractName, List<Object> params)**：传入合约名和构造好的合约构造函数，接收TransactionResponse结果。
- **public void deployByContractLoaderAsync(String contractName, List<Object> args, TransactionCallback callback)**：传入合约名和构造好的合约构造函数以及callback，来异步部署合约
- **public TransactionReceipt sendTransactionAndGetReceiptByContractLoader(String contractName, String contractAddress, String functionName, List<Object> params)**：传入调用合约名称、合约地址、函数名和函数参数，接收交易回执
- **public TransactionResponse sendTransactionAndGetResponse(String to, String abi, String functionName, List<Object> params)**：传入调用合约地址、合约abi、函数名和函数参数，接收TransactionResponse结果

- **public TransactionResponse sendTransactionWithStringParamsAndGetResponse(String to, String abi, String functionName, List<String> params)**：传入调用合约地址、合约abi、函数名和String类型List的函数参数，接收TransactionResponse结果
- **public void sendTransactionAsync(String to, String abi, String functionName, List<Object> params, TransactionCallback callback)**：传入调用合约地址、合约abi、函数名、函数参数、callback，异步发送交易。
- **public void sendTransactionAndGetReceiptByContractLoaderAsync(String contractName,String contractAddress, String functionName, List<Object> args, TransactionCallback callback)**：传入调用合约名、合约地址、函数名、函数参数、callback，异步发送交易。
- **public TransactionResponse sendTransactionAndGetResponseByContractLoader(String contractName, String contractAddress, String functionName, List<Object> funcParams)**：传入调用合约名、合约地址、函数名、函数参数，并接收TransactionResponse结果。
- **public CallResponse sendCallByContractLoader(String contractName, String contractAddress, String functionName, List<Object> params)**：传入调用合约名、合约地址、函数名、函数参数，并接收CallResponse结果。
- **public CallResponse sendCall(String from, String to, String abi, String functionName, List<Object> args)**：传入调用者地址、合约地址、合约abi、函数名、函数参数，并接收CallResponse结果。
- **public CallResponse sendCall(CallRequest callRequest)**：传入CallRequest，并接收CallResponse结果。
- **public CallResponse sendCallWithStringParams(String from, String to, String abi, String functionName, List<String> paramsList):** 传入调用者地址、合约地址、合约abi、函数名、String类型List的函数参数，并接收CallResponse结果。

4. 扩展：使用接口签名的方式发送交易

此外，对于特殊的场景，可以通过接口签名的方式DIY拼装交易和发送交易。

例如上述HelloWorld智能合约定义的set方法的签名为 “set(string)”

构造接口签名

```
    ABICodec abiCodec = new ABICodec(client.getCryptoSuite());
    String setMethodSignature = "set(string)";
    String abiEncoded = abiCodec.encodeMethodByInterface(setMethodSignature, new␣
↪Object[]{new String("Hello World")});
```

构造TransactionProcessor

TransactionProcessor同样可使用TransactionProcessorFactory来构造。

```
    // ......
    TransactionProcessor transactionProcessor = TransactionProcessorFactory.
↪createTransactionProcessor(client, keyPair);
```

发送交易

```
    // ......
    TransactionReceipt transactionReceipt = transactionProcessor.
↪sendTransactionAndGetReceipt(contractAddress, abiEncoded, keyPair);
```

8.4.4 交易回执解析

标签：java-sdk 回执解析 日志解析

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把交易回执解析为json数据。

1. 构造TransactionDecoderInterface

创建一个TransactionDecoderService对象。

```
// 初始化SDK
BcosSDK sdk =  BcosSDK.build(configFile);
// 发送群组ID1
Client client = sdk.getClient(Integer.valueOf(1));
// 获取当前群组对应的密码学接口
CryptoSuite cryptoSuite = client.getCryptoSuite();
// 构造TransactionDecoderService实例，传入是否密钥类型参数。
TransactionDecoderInterface decoder = new TransactionDecoderService(cryptoSuite);
```

TransactionDecoderInterface 主要包括以下功能： // abi在合约生成的java客户端文件夹下,以HelloWorld.sol为例,为HelloWorld.abi中的json字符串。

- **public TransactionResponse decodeReceiptWithValues(String abi, String functionName, TransactionReceipt receipt)**：解析不带函数返回值的交易回执。
- **public TransactionResponse decodeReceiptWithoutValues(String abi, TransactionReceipt transactionReceipt)**：解析带函数返回值的交易回执。
- **public Map<String, List<List<Object>>>> decodeEvents(String abi, List<Logs> logs)**：解析交易日志。
- **public TransactionResponse decodeReceiptStatus(TransactionReceipt receipt)**：解析回执的状态和报错信息等。

2. 解析带返回值的交易

传入合约的abi文件，调用函数的名称，以及交易回执，解析交易结果。

```
TransactionResponse transactionResponse = decoder.decodeReceiptWithValues(abi,
↪"incrementUint256", transactionReceipt);
```

解析结果示例：

对应的solidity代码：

```
function incrementUint256(uint256 v) public returns(uint256){
    _uint256V = v + 1 ;
    emit LogIncrement(msg.sender, v);
    return _uint256V;
}
```

以上函数定义中，有函数返回值，也触发了event调用。我们的传入值v为1. 解析交易执行返回的TransactionReceipt以后，对应的结果如下：


```
{
  "returnCode": 2,
  "returnMessage": "Success",
  "transactionReceipt": {
    "transactionHash":
↪"0x433c41e0bdd5420f07186eb33d47aac9cf4bbfff040d27213f17ad739096f19b",
    "transactionIndex": "0x0",
    "root": "0x3094859967ad37882b450ffa97dc15ad9d1d69cab8d3ff8212f6c200185f7bae",
    "blockNumber": "0x20a",
    "blockHash":
↪"0x02143583560dd1bc0c226f0d2a7bd947170c322ef5be203da3308fdd36fde87e",
    "from": "0x7c8000530ae01adb3f8f77e7096b335eef83172f",
    "to": "0xa90bec2f9957eed99c6856172f0c58a5cb2a46fd",
    "gasUsed": "0xab6a",
    "contractAddress": "0x0000000000000000000000000000000000000000",
    "logs": [
      {
        "address": "0xa90bec2f9957eed99c6856172f0c58a5cb2a46fd",
        "topics": [
          "0xaca9a02cfe513f3f88c54a860469369849c8fa0a2119a8d1f3f75c67ac0c9547"
        ],
        "data":
↪"0x0000000000000000000000007c8000530ae01adb3f8f77e7096b335eef83172f0000000000000000000000000000000
↪",
        "blockNumber": null
      }
    ],
    "logsBloom":
↪"0x00000000000000000000000000000000000000000
```

3. 解析无返回值的交易

传入合约的abi文件和交易回执，解析交易结果。

```
TransactionResponse transactionResponseWithoutValues = decoder.
↪decodeReceiptWithoutValues(abi, transactionReceipt);
```

解析结果示例：

还是以上节调用incrementUint256函数为例，我们还是解析此交易回执，但不解析函数返回值，返回的结果如下：

```
{
  "returnCode": 2,
  "returnMessage": "Success",
  "transactionReceipt": {
    "transactionHash":
↪"0x433c41e0bdd5420f07186eb33d47aac9cf4bbfff040d27213f17ad739096f19b",
    "transactionIndex": "0x0",
    "root": "0x3094859967ad37882b450ffa97dc15ad9d1d69cab8d3ff8212f6c200185f7bae",
    "blockNumber": "0x20a",
    "blockHash":
↪"0x02143583560dd1bc0c226f0d2a7bd947170c322ef5be203da3308fdd36fde87e",
    "from": "0x7c8000530ae01adb3f8f77e7096b335eef83172f",
    "to": "0xa90bec2f9957eed99c6856172f0c58a5cb2a46fd",
    "gasUsed": "0xab6a",
    "contractAddress": "0x0000000000000000000000000000000000000000",
    "logs": [
      {
        "address": "0xa90bec2f9957eed99c6856172f0c58a5cb2a46fd",
        "topics": [
          "0xaca9a02cfe513f3f88c54a860469369849c8fa0a2119a8d1f3f75c67ac0c9547"
        ],
        "data":
↪"0x0000000000000000000000007c8000530ae01adb3f8f77e7096b335eef83172f0000000000000000000000000000000
↪",
        "blockNumber": null
      }
    ],
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "status": "0x0",
    "input":
↪"0x7c1bf3c50000000000000000000000000000000000000000000000000000000000000001",
    "output": "0x0000000000000000000000000000000000000000000000000000000000000002",
    "txProof": null,
    "receiptProof": null,
    "message": "Success",
    "statusOK": true
  },
  "contractAddress": "0x0000000000000000000000000000000000000000",
  "values": null,
  "events": "{\"LogIncrement\":[\"0x7c8000530ae01adb3f8f77e7096b335eef83172f\",1]}
↪",
  "receiptMessages": "Success",
  "valuesList": null,
  "eventResultMap": {
    "LogIncrement": [
      [
```

(continues on next page)

(续上页)

```
        "0x7c8000530ae01adb3f8f77e7096b335eef83172f",
        1
      ]
    ]
  }
}
```

4. 解析日志

只解析调用函数过程中触发的日志。传入合约的abi文件和交易回执的logs，解析交易结果；返回事件名和事件List的Map。

```
Map<String, List<List<Object>>>> events = decoder.decodeEvents(abi,␣
↪transactionReceipt.getLogs());
```

解析结果示例：

还是以上节调用incrementUint256函数为例，现在演示只解析事件，返回的结果如下：

```
{
  "LogIncrement": [
    [
      "0x7c8000530ae01adb3f8f77e7096b335eef83172f",
      1
    ]
  ]
}
```

5. 解析回执的错误信息

传入交易回执，解析返回数据，并解析为TransactionResponse对象。

```
TransactionResponse transactionResponse = decoder.
↪decodeReceiptStatus(transactionReceipt);
```

解析结果示例：

对应的solidity代码：

```
function setBytesMapping(bytes[] bytesArray) public returns (bool) {
    require(bytesArray.length>1, "Bytes array is less than 2");
    _bytesMapping[bytesArray[0]] = bytesArray;
    return true;
}
```

以下函数执行中，交易执行失败，在执行require语句后报错。解析交易执行返回的TransactionReceipt以后，对应的结果如下：

```
{
  "returnCode": 22,
  "returnMessage": "Bytes array is less than 2",
  "transactionReceipt": null,
  "contractAddress": null,
  "values": null,
```

(continues on next page)

(续上页)

```
  "events": null,
  "receiptMessages": "Bytes array is less than 2",
  "valuesList": null,
  "eventResultMap": null
}
```

8.4.5 AMOP 功能

标签: java-sdk AMOP 链上信使

Java SDK支持链上信使协议AMOP（Advanced Messages Onchain Protocol），用户可以通过AMOP协议与其它机构互传消息。

1. 配置方法

AMOP有两种话题模式普通话题和私有话题。任何一个订阅了某普通话题的订阅者都能收到该话题相关的推送消息。但在某些情况下，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话题，这时就需要使用AMOP私有话题了。

AMOP私有话题的特别之处在于，SDK间需要进行了身份认证，认证通过的订阅者才可以收到该话题的消息。身份认证的原理是，首先由发送方生成一个随机数，订阅方用私钥对随机数签名，发送方用所配置的公钥验证这个签名来确定对方是否是自己指定的订阅方。因此，一个成功的私有话题通道的建立需要（1）消息发送者需要设置指定的订阅者的公钥；（2）订阅方也需要设置能证明自己身份的私钥。

当用户需要订阅私有话题，或者作为消息发布者配置一个私有话题时，可用配置文件进行配置。但AMOP的配置不是必须项，私有话题的订阅和设置，还可以通过调用AMOP的接口实现。以下是AMOP的配置示例，是test/resource/config-example.toml配置文件中的一部分。

```
# AMOP configuration
# You can use following two methods to configure as a private topic message sender
↪or subscriber.
# Usually, the public key and private key is generated by subscriber.
# Message sender receive public key from topic subscriber then make configuration.
# But, please do not config as both the message sender and the subscriber of one
↪private topic, or you may send the message to yourself.

# Configure a private topic as a topic message sender.
# [[amop]]
# topicName = "PrivateTopic"
# publicKeys = [ "conf/amop/consumer_public_key_1.pem" ]    # Public keys of the
↪nodes that you want to send AMOP message of this topic to.

# Configure a private topic as a topic subscriber.
# [[amop]]
# topicName = "PrivateTopic"
# privateKey = "conf/amop/consumer_private_key.p12"         # Your private key
↪that used to subscriber verification.
# password = "123456"
```

配置详解:

1. 配置一个私有话题（作为消息发布者）
    - 需要在配置文件中新建一个[[amop]]节点。
    - 并配置话题名称topicName = "PrivateTopic"
    - 消息订阅方的公钥列表publicKeys = ["conf/amop/consumer_public_key_1.pem"]，即您想指定的接收对象的公钥，这个公钥必须与某个订阅方的私钥所匹配。

```
[[amop]]
topicName = "PrivateTopic"
publicKeys = [ "conf/amop/consumer_public_key_1.pem" ]
```

2. 订阅一个私有话题（作为订阅者）
   - 需要在配置文件中新建一个`[[amop]]`节点。
   - 并配置话题名称`topicName = "PrivateTopic"`
   - 证明订阅方身份的私钥`privateKey = "conf/amop/consumer_private_key.p12"`
   - 该私钥的密码`password = "123456"`

```
[[amop]]
topicName = "PrivateTopic"
privateKey = "conf/amop/consumer_private_key.p12"
password = "123456"
```

2. 接口说明

AMOP模块的接口类可参考文件java-sdk中的`sdk-amop/src/main/org/fisco/bcos/sdk/amop/Amop.java`文件，其中主要包含以下几个接口：

subscribeTopic

订阅一个普通话题

参数：

- topicName: 话题名称。类型：`String`。
- callback: 处理该话题消息的函数，当收到该话题相关消息时，会被调用。类型：`AmopCallback`。

例子：

```
// 初始化java SDK，获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 定义一个Callback，重写receiveAmopMsg方法，定义收到消息后的处理流程。
AmopCallback cb = new AmopCallback() {
  @Override
  public byte[] receiveAmopMsg(AmopMsgIn msg) {
    // 你可以在这里写收到消息后的处理逻辑。
    System.out.println("Received msg, content:" + new String(msg.getContent()));
    return "Yes, I received.".getBytes();
  }
};
// 订阅话题
amop.subscribeTopic("MyTopic", cb);
```

subscribePrivateTopic

订阅一个私有话题

参数：

- topicName: 话题名称。类型：`String`。

- privateKeyManager：证明订阅者身份的私钥信息。类型：KeyManager。
- callback：处理该话题消息的函数，当收到该话题相关消息时，会被调用。类型：AmopCallback。

例子：

```
// 初始化java SDK，获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 定义一个Callback，重写receiveAmopMsg方法，定义收到消息后的处理流程。
AmopCallback cb = new AmopCallback() {
  @Override
  public byte[] receiveAmopMsg(AmopMsgIn msg) {
    // 你可以在这里写收到消息后的处理逻辑。
    System.out.println("Received msg, content:" + new String(msg.getContent()));
    return "Yes, I received.".getBytes();
  }
};

// 加载私钥
KeyManager km = new P12Manager("private_key.p12", "12s230");

// 订阅话题
amop.subscribePrivateTopics("MyPrivateTopic", km, cb);
```

publishPrivateTopic

作为消息发送者设置一个私有话题。

参数：

- topicName: 话题名称。类型：String。
- publicKeyManagers：指定订阅者的公钥列表。类型：List<KeyManager>。

例子：

```
// 初始化java SDK，获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 加载指定订阅者的私钥列表
List<KeyManager> publicKeyList = new ArrayList<>();
KeyManager pubKey1 = new PEMManager("target_subscriber1_public_key.pem");
KeyManager pubKey2 = new PEMManager("target_subscriber2_public_key.pem");
publicKeyList.add(pubKey1);
publicKeyList.add(pubKey2);

// 设置一个私有话题
amop.publishPrivateTopic("MyPrivateTopic", publicKeyList);
```

unsubscribeTopic

取消一个话题的订阅。

参数：

- topicName: 话题名称。类型：String。

例子：

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 订阅"MyTopic", 过程省略

// 取消订阅
amop.unsubscribeTopic("MyTopic");
```

sendAmopMsg

以单播的方式发送AMOP消息

参数:

- content: 消息内容。类型：AmopMsgOut。
- callback: 回调函数。类型：ResponseCallback。

例子:

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 构造内容
AmopMsgOut out = new AmopMsgOut();
out.setTopic("MyTopic");
out.setType(TopicType.NORMAL_TOPIC);
out.setContent("some content here".getBytes());
out.setTimeout(6000);

// 构造Callback
ResponseCallback cb =
  new ResponseCallback() {
  @Override
  public void onResponse(Response response) {
                // 你可以在这里写收到回复的处理逻辑。
    System.out.println(
      "Get response, { errorCode:"
      + response.getErrorCode()
      + " error:"
      + response.getErrorMessage()
      + " seq:"
      + response.getMessageID()
      + " content:"
      + new String(response.getContentBytes())
      + " }");
  }
};

// 发送消息
amop.sendAmopMsg(out, cb);
```

broadcastAmopMsg

以广播的方式发送AMOP消息

参数:

- content: 消息内容。类型：AmopMsgOut。

例子：

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 构造内容
AmopMsgOut out = new AmopMsgOut();
out.setTopic("MyTopic");
out.setType(TopicType.NORMAL_TOPIC);
out.setContent(content.getBytes());
out.setTimeout(6000);

// 发送消息
amop.broadcastAmopMsg(out);
```

getSubTopics

获取SDK当前订阅的话题

参数：

无

返回值：

- 订阅的话题集合。类型：`Set<String>`。

例子：

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

Set<String> topics = amop.getSubTopics();
```

start

启动amop功能，SDK初始化后默认启动。

例子：

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

amop.stop()

// 在需要的时候
amop.start()
```

stop

停止AMOP功能。

例子：

```
// 初始化java SDK, 获得Amop对象
BcosSDK sdk = new BcosSDK("config-example.toml");
Amop amop = sdk.getAmop();

// 停止AMOP
amop.stop()
```

3. 示例

更多的示例请看java-sdk-demo项目源码`java-sdk-demo/src/main/java/org/fisco/bcos/sdk/demo/amop/tool`下的代码示范。

- 普通话题代码示例：
  - AmopPublisher 输入： 话题名（TopicName），是否多播（isBroadcast, true 为多播，false为单拨），内容（Content），发送的数量（Count） 功能：发送普通话题消息、广播普通话题消息
  - AmopPublisherFile 输入： 话题名（TopicName），是否多播（isBroadcast），文件名（FileName），发送的数量（Count） 功能：发送普通话题文件、广播普通话题文件
  - AmopSubscriber 输入：话题名（TopicName） 默认：订阅了一个名为“Test”的普通话题 功能：订阅一个普通话题
- 私有话题代码示例：
  - AmopSubscriberPrivate

    输入：话题名（TopicName），私钥文件名（Filename），密码（password） 默认：订阅了一个名为“PrivTest”的私有话题 功能：订阅一个私有话题
  - AmopPublisherPrivate

    输入： 话题名（TopicName），公钥文件名1（Filename），公钥文件名2（Filename），是否多播（isBroadcast），内容（Content），发送的数量（Count） 功能：配置私有话题发送，发送私有话题消息，广播私有话题消息
  - AmopPublisherPrivateFile

    输入：话题名（TopicName），公钥文件名1（Filename），公钥文件名2（Filename，如无则输入null），是否多播（isBroadcast），文件名（FileName），发送的数量（Count） 测试功能：发送私有话题文件、广播普通话题文件

示例讲解

- 配置。

  Java SDK支持动态订阅话题。也可以在配置文件中配置固定的私有话题。

  订阅者配置例子：java-sdk-demo项目源码`java-sdk-demo/src/resources/amop/config-subscriber-for-test.toml`

```
[network]
# The peer list to connect
peers=["127.0.0.1:20202", "127.0.0.1:20203"]

# Configure a private topic as a topic message sender.
[[amop]]
topicName = "privTopic"
# Your private key that used to subscriber verification.
privateKey = "conf/amop/consumer_private_key.p12"
password = "123456"
```

  注意，订阅方通过这种方法配置的话题，需要在程序中设定一个默认的回调函数，否则，接收消息时会因为找不到回调函数而报错。

发送者配置例子：java-sdk-demo项目源码java-sdk-demo/src/resources/amop/config-publisher-for-test.toml

```
[network]
# The peer list to connect
peers=["127.0.0.1:20200", "127.0.0.1:20201"]

# Configure a "need verify AMOP topic" as a topic message sender.
[[amop]]
topicName = "privTopic"
# Public keys of the nodes that you want to send AMOP message of this topic to.
publicKeys = [ "conf/amop/consumer_public_key_1.pem"]
```

发送者所配置的公钥是从订阅者那里获取的，与订阅者的私钥是成对的。这样发送者可以通过私有话题"privTopic"向订阅者发送消息。

- 公有话题订阅和发送

订阅者代码例子：src/main/java/org/fisco/bcos/sdk/demo/amop/tool/AmopSubscriber.java

```
package org.fisco.bcos.sdk.demo.amop.tool;

import java.net.URL;
import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.amop.Amop;
import org.fisco.bcos.sdk.amop.AmopCallback;

public class AmopSubscriber {

    public static void main(String[] args) throws Exception {
        URL configUrl =
                AmopSubscriber.class
                        .getClassLoader()
                        .getResource("amop/config-subscriber-for-test.toml");
        if (args.length < 1) {
            System.out.println("Param: topic");
            return;
        }
        String topic = args[0];
        // Construct a BcosSDK instance
        BcosSDK sdk = BcosSDK.build(configUrl.getPath());

        // Get the amop module instance
        Amop amop = sdk.getAmop();

        // Set callback
        AmopCallback cb = new DemoAmopCallback();
        // Set a default callback
        amop.setCallback(cb);
        // Subscriber a normal topic
        amop.subscribeTopic(topic, cb);
        System.out.println("Start test");
    }
}
```

回调函数例子: src/main/java/org/fisco/bcos/sdk/demo/amop/tool/DemoAmopCallback.java

```
package org.fisco.bcos.sdk.demo.amop.tool;

import static org.fisco.bcos.sdk.utils.ByteUtils.byteArrayToInt;
```

(continues on next page)

(续上页)

```
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;
import org.fisco.bcos.sdk.amop.AmopCallback;
import org.fisco.bcos.sdk.amop.topic.AmopMsgIn;
import org.fisco.bcos.sdk.model.MsgType;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class DemoAmopCallback extends AmopCallback {
    private static Logger logger = LoggerFactory.getLogger(DemoAmopCallback.
↪class);

    @Override
    public byte[] receiveAmopMsg(AmopMsgIn msg) {
        if (msg.getContent().length > 8) {
            byte[] content = msg.getContent();
            byte[] byteflag = subbytes(content, 0, 4);
            int flag = byteArrayToInt(byteflag);
            if (flag == -128) {
                // Received a file.
                byte[] bytelength = subbytes(content, 4, 4);
                int length = byteArrayToInt(bytelength);
                byte[] bytefilename = subbytes(content, 8, length);
                String filename = new String(bytefilename);
                System.out.println(
                        "Step 2:Receive file, filename length:"
                                + length
                                + " filename binary:"
                                + Arrays.toString(bytefilename)
                                + " filename:"
                                + filename);

                int contentlength = content.length - 8 - filename.length();
                byte[] fileContent = subbytes(content, 8 + filename.length(),␣
↪contentlength);
                getFileFromBytes(fileContent, filename);
                System.out.println("|---save file:" + filename + " success");
                byte[] responseData = "Yes, I received!".getBytes();
                if (msg.getType() == (short) MsgType.AMOP_REQUEST.getType()) {
                    System.out.println("|---response:" + new␣
↪String(responseData));
                }
                return responseData;
            }
        }


        byte[] responseData = "Yes, I received!".getBytes();
        // Print receive amop message
        System.out.println(
                "Step 2:Receive msg, topic:"
                        + msg.getTopic()
                        + " content:"
                        + new String(msg.getContent()));
        if (msg.getType() == (short) MsgType.AMOP_REQUEST.getType()) {
            System.out.println("|---response:" + new String(responseData));
        }
        // Response to the message sender
```

(continues on next page)

(续上页)

```
        return responseData;
    }

    public static byte[] subbytes(byte[] src, int begin, int count) {
        byte[] bs = new byte[count];
        System.arraycopy(src, begin, bs, 0, count);
        return bs;
    }

    public static void getFileFromBytes(byte[] b, String outputFile) {
        File ret = null;
        BufferedOutputStream stream = null;
        FileOutputStream fstream = null;
        try {
            ret = new File(outputFile);
            fstream = new FileOutputStream(ret);
            stream = new BufferedOutputStream(fstream);
            stream.write(b);
        } catch (Exception e) {
            logger.error(" write exception, message: {}", e.getMessage());
        } finally {
            if (stream != null) {
                try {
                    stream.close();
                } catch (IOException e) {
                    logger.error(" close exception, message: {}", e.
↪getMessage());
                }
            }

            if (fstream != null) {
                try {
                    fstream.close();
                } catch (IOException e) {
                    logger.error(" close exception, message: {}", e.
↪getMessage());
                }
            }
        }
    }
}
```

发送方使用例子：src/main/java/org/fisco/bcos/sdk/demo/amop/tool/AmopPublisher.java

```
package org.fisco.bcos.sdk.demo.amop.tool;

import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.amop.Amop;
import org.fisco.bcos.sdk.amop.AmopMsgOut;
import org.fisco.bcos.sdk.amop.topic.TopicType;

public class AmopPublisher {
    private static final int parameterNum = 4;
    private static String publisherFile =
            AmopPublisher.class
                    .getClassLoader()
                    .getResource("amop/config-publisher-for-test.toml")
                    .getPath();

    /**
```

(continues on next page)

(续上页)

```
     * @param args topicName, isBroadcast, Content(Content you want to send␣
↪out), Count(how many msg
     *     you want to send out)
     * @throws Exception AMOP publish exceptioned
     */
    public static void main(String[] args) throws Exception {
        if (args.length < parameterNum) {
            System.out.println("param: target topic total number of request");
            return;
        }
        String topicName = args[0];
        Boolean isBroadcast = Boolean.valueOf(args[1]);
        String content = args[2];
        Integer count = Integer.parseInt(args[3]);
        BcosSDK sdk = BcosSDK.build(publisherFile);
        Amop amop = sdk.getAmop();

        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);
            AmopMsgOut out = new AmopMsgOut();
            out.setType(TopicType.NORMAL_TOPIC);
            out.setContent(content.getBytes());
            out.setTimeout(6000);
            out.setTopic(topicName);
            DemoAmopResponseCallback cb = new DemoAmopResponseCallback();
            if (isBroadcast) {
                    // send out amop message by broad cast
                amop.broadcastAmopMsg(out);
                System.out.println(
                        "Step 1: Send out msg by broadcast, topic:"
                                + out.getTopic()
                                + " content:"
                                + new String(out.getContent()));
            } else {
                    // send out amop message
                amop.sendAmopMsg(out, cb);
                System.out.println(
                        "Step 1: Send out msg, topic:"
                                + out.getTopic()
                                + " content:"
                                + new String(out.getContent()));
            }
        }
    }
}
```

发送方接收回包的函数例子：`src/main/java/org/fisco/bcos/sdk/demo/amop/tool/DemoAmopResponseCallback.java`

```
package org.fisco.bcos.sdk.demo.amop.tool;
```

(continues on next page)

(续上页)

```
import org.fisco.bcos.sdk.amop.AmopResponse;
import org.fisco.bcos.sdk.amop.AmopResponseCallback;

public class DemoAmopResponseCallback extends AmopResponseCallback {

    @Override
    public void onResponse(AmopResponse response) {
            // 当出现102超时错误时，打印该错误
        if (response.getErrorCode() == 102) {
            System.out.println(
                    "Step 3: Timeout, maybe your file is too large or your␣
↪gave a short timeout.");
        } else {
                                // 收到正常的回包
            if (response.getAmopMsgIn() != null) {
                System.out.println(
                        "Step 3:Get response, { errorCode:"
                                + response.getErrorCode()
                                + " error:"
                                + response.getErrorMessage()
                                + " seq:"
                                + response.getMessageID()
                                + " content:"
                                + new String(response.getAmopMsgIn().
↪getContent())
                                + " }");
            } else {
              // 收到其它错误
                System.out.println(
                        "Step 3:Get response, { errorCode:"
                                + response.getErrorCode()
                                + " error:"
                                + response.getErrorMessage()
                                + " seq:"
                                + response.getMessageID());
            }
        }
    }
}
```

- 私有话题的发送和订阅:

  订阅一个私有话题例子：`src/main/java/org/fisco/bcos/sdk/demo/amop/tool/AmopSubscriberPrivate.java`

```
package org.fisco.bcos.sdk.demo.amop.tool;

import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.amop.Amop;
import org.fisco.bcos.sdk.amop.AmopCallback;
import org.fisco.bcos.sdk.crypto.keystore.KeyTool;
import org.fisco.bcos.sdk.crypto.keystore.P12KeyStore;
import org.fisco.bcos.sdk.crypto.keystore.PEMKeyStore;

public class AmopSubscriberPrivate {
    private static String subscriberConfigFile =
            AmopSubscriberPrivate.class
                    .getClassLoader()
                    .getResource("amop/config-subscriber-for-test.toml")
                    .getPath();
```

(continues on next page)

(续上页)

```
    /**
     * @param args topic, privateKeyFile, password(Option)
     * @throws Exception AMOP exceptioned
     */
    public static void main(String[] args) throws Exception {
        if (args.length < 2) {
            System.out.println("Param: topic, privateKeyFile, password");
            return;
        }
        String topic = args[0];
        String privateKeyFile = args[1];
        BcosSDK sdk = BcosSDK.build(subscriberConfigFile);
        Amop amop = sdk.getAmop();
        AmopCallback cb = new DemoAmopCallback();

        System.out.println("Start test");
        amop.setCallback(cb);

        // Read a private key file
        KeyTool km;
        if (privateKeyFile.endsWith("p12")) {
            String password = args[2];
            km = new P12KeyStore(privateKeyFile, password);
        } else {
            km = new PEMKeyStore(privateKeyFile);
        }

        // Subscriber a private topic.
        amop.subscribePrivateTopics(topic, km, cb);
    }
}
```

发送私有话题消息例子:

```
package org.fisco.bcos.sdk.demo.amop.tool;

import java.util.ArrayList;
import java.util.List;
import org.fisco.bcos.sdk.BcosSDK;
import org.fisco.bcos.sdk.amop.Amop;
import org.fisco.bcos.sdk.amop.AmopMsgOut;
import org.fisco.bcos.sdk.amop.topic.TopicType;
import org.fisco.bcos.sdk.crypto.keystore.KeyTool;
import org.fisco.bcos.sdk.crypto.keystore.PEMKeyStore;

public class AmopPublisherPrivate {
    private static final int parameterNum = 6;
    private static String publisherFile =
            AmopPublisherPrivate.class
                    .getClassLoader()
                    .getResource("amop/config-publisher-for-test.toml")
                    .getPath();

    /**
     * @param args topicName, pubKey1, pubKey2, isBroadcast: true/false,
↪content, count. if only one
     *     public key please fill pubKey2 with null
     * @throws Exception AMOP exceptioned
     */
    public static void main(String[] args) throws Exception {
```

(continues on next page)

(续上页)

```
        if (args.length < parameterNum) {
            System.out.println(
                    "param: opicName, pubKey1, pubKey2, isBroadcast: true/
↪false, content, count");
            return;
        }
        String topicName = args[0];
        String pubkey1 = args[1];
        String pubkey2 = args[2];
        Boolean isBroadcast = Boolean.valueOf(args[3]);
        String content = args[4];
        Integer count = Integer.parseInt(args[5]);
        BcosSDK sdk = BcosSDK.build(publisherFile);
        Amop amop = sdk.getAmop();

        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(
↪"======================================================================");
        System.out.println("set up private topic");
        List<KeyTool> keyToolList = new ArrayList<>();

        // Read public key files.
        KeyTool keyTool = new PEMKeyStore(pubkey1);
        keyToolList.add(keyTool);
        if (!pubkey2.equals("null")) {
            KeyTool keyTool1 = new PEMKeyStore(pubkey2);
            keyToolList.add(keyTool1);
        }
        // Publish a private topic
        amop.publishPrivateTopic(topicName, keyToolList);
        System.out.println("wait until finish private topic verify");
        System.out.println("3s ...");
        Thread.sleep(1000);
        System.out.println("2s ...");
        Thread.sleep(1000);
        System.out.println("1s ...");
        Thread.sleep(1000);

        for (Integer i = 0; i < count; ++i) {
            Thread.sleep(2000);
            AmopMsgOut out = new AmopMsgOut();
            // It is a private topic.
            out.setType(TopicType.PRIVATE_TOPIC);
            out.setContent(content.getBytes());
            out.setTimeout(6000);
            out.setTopic(topicName);
            DemoAmopResponseCallback cb = new DemoAmopResponseCallback();
            if (isBroadcast) {
                // Send out message by broadcast
                amop.broadcastAmopMsg(out);
                System.out.println(
                        "Step 1: Send out msg by broadcast, topic:"
                                + out.getTopic()
                                + " content:"
```

(continues on next page)

(续上页)

```
                                + new String(out.getContent()));
            } else {
                // Send out amop message
                amop.sendAmopMsg(out, cb);
                System.out.println(
                        "Step 1: Send out msg, topic:"
                                + out.getTopic()
                                + " content:"
                                + new String(out.getContent()));
            }
        }
    }
}
```

接下来，可以根据下一节的方法来试用这些AMOP的Demo。

4. 快速试用AMOP

第一步：下载项目

```
mkdir -p ~/fisco && cd ~/fisco
# 获取java-sdk代码
git clone https://github.com/FISCO-BCOS/java-sdk-demo
cd java-sdk-demo
# 构建项目
bash gradlew build
```

第二步：搭建FISCO BCOS区块链网络

方法一：如果您使用的操作系统为Linux或Darwin，可以用将以下脚本保存到java-sdk的目录下，命名为initEnv.sh，并运行该脚本。

initEnv.sh

```
#!/bin/bash
download_tassl()
{
  mkdir -p ~/.fisco/
  if [ "$(uname)" == "Darwin" ];then
    curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/tassl_mac.
↪tar.gz
    mv tassl_mac.tar.gz ~/.fisco/tassl.tar.gz
  else
    curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/tassl.tar.gz
    mv tassl.tar.gz ~/.fisco/tassl.tar.gz
  fi
  tar -xvf ~/.fisco/tassl.tar.gz
}

build_node()
{
  local node_type="${1}"
  if [ "${node_type}" == "sm" ];then
      ./build_chain.sh -l 127.0.0.1:4 -g
      sed_cmd=$(get_sed_cmd)
      $sed_cmd 's/sm_crypto_channel=false/sm_crypto_channel=true/g' nodes/127.0.0.
↪1/node*/config.ini
```

(continues on next page)

(续上页)

```
  else
      ./build_chain.sh -l 127.0.0.1:4
  fi
  ./nodes/127.0.0.1/fisco-bcos -v
  ./nodes/127.0.0.1/start_all.sh
}
download_build_chain()
{
  tag=$(curl -sS "https://gitee.com/api/v5/repos/FISCO-BCOS/FISCO-BCOS/tags" |␣
↪grep -oe "\"name\":\"v[2-9]*\.[0-9]*\.[0-9]*\"" | cut -d \" -f 4 | sort -V |␣
↪tail -n 1)
  LOG_INFO "--- current tag: $tag"
  curl -LO "https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/${tag}/
↪build_chain.sh" && chmod u+x build_chain.sh
}
prepare_environment()
{
  ## prepare resources for amop demo
  bash gradlew build -x test
  cp -r nodes/127.0.0.1/sdk/* sdk-demo/dist/conf
}

download_tassl
./gradlew build -x test
download_build_chain
build_node
prepare_environment
```

运行该文件：

```
# 更改文件权限
chmod 777 initEnv.sh
# 运行initEnv.sh文件
./initEnv.sh
```

运行完成后，终端显示4个节点已经启动，demo环境就已经准备好了。

```
FISCO-BCOS Version : 2.6.0
Build Time         : 20200814 09:04:17
Build Type         : Darwin/appleclang/RelWithDebInfo
Git Branch         : HEAD
Git Commit Hash    : e4a5ef2ef64d1943fccc4ebc61467a91779fb1c0
try to start node0
try to start node1
try to start node2
try to start node3
 node1 start successfully
 node3 start successfully
 node0 start successfully
 node2 start successfully
```

方法二：您也可以根据指引搭建FISCO BCOS区块链网络。然后进行以下操作

```
# 当前目录为java-sdk,构建项目
gradlew.bat build -x test
```

第三步：配置

- 复制证书：将你搭建FISCO BCOS网络节点nodes/${ip}/sdk/ 目录下的证书复制到java-sdk-demo/dist/conf目录下。

- 修改配置：如果您采用的是方法一搭建的网络，则无需修改配置。如果您采用方法二搭建区块链，需要修改订阅者配置文件java-sdk-demo/dist/conf/amop/config-subscriber-for-test.toml，和发送者配置文件java-sdk-demo/dist/conf/amop/config-publisher-for-test.toml，修改配置文件中的节点信息。注意：订阅者和发送者最好不连相同节点，如果连接了相同节点，则会被认为是同一个组织下的成员，私有话题无需认证即可通讯。

第四步：运行Demo

公有话题Demo

新打开一个终端，下载java-sdk-demo的代码，并build。

```
cd ~/fisco
# 获取java-sdk-demo代码
git clone https://github.com/FISCO-BCOS/java-sdk-demo
cd java-sdk-demo

# build项目
bash gradlew build
```

运行订阅者：

```
# 进入java-sdk-demo/dist目录
cd dist
# 使用第三节中所描述的工具
# 我们订阅名为"testTopic"的话题
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.AmopSubscriber␣
↪testTopic
```

订阅方的终端输出

```
Start test
```

然后，运行发送者Demo

单播消息：

```
# 调用AmopPublisher发送AMOP消息
# 话题名：testTopic，是否广播：false(即使用单播)，内容：Tell you something， 发送次数：2次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.AmopPublisher␣
↪testTopic false "Tell you something" 2
```

终端输出：

```
3s ...
2s ...
1s ...
start test
============================================================
Step 1: Send out msg, topic:testTopic content:Tell you something
Step 3:Get response, { errorCode:0 error:null seq:3fa95b760f7f48ddbdf1216a48f361e0␣
↪content:Yes, I received! }
Step 1: Send out msg, topic:testTopic content:Tell you something
Step 3:Get response, { errorCode:0 error:null seq:2bc754b1d8844445a4cc2af226fbaa58␣
↪content:Yes, I received! }
```

同时，返回到话题订阅者的终端，发现终端输出：

```
Step 2:Receive msg, topic:testTopic content:Tell you something
|---response:Yes, I received!
Step 2:Receive msg, topic:testTopic content:Tell you something
|---response:Yes, I received!
```

广播消息:

```
# 调用AmopPublisher发送AMOP消息
# 话题名：testTopic，是否广播：false(即使用单播)，内容：Tell you something， 发送次数：1次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.AmopPublisher␣
↪testTopic true "Tell you something" 1
```

终端的输出

```
3s ...
2s ...
1s ...
start test
================================================================
Step 1: Send out msg by broadcast, topic:testTopic content:Tell you something
```

同时，返回到话题订阅者的终端，发现终端输出：

```
Start test
Step 2:Receive msg, topic:testTopic content:Tell you something
Step 2:Receive msg, topic:testTopic content:Tell you something
Step 2:Receive msg, topic:testTopic content:Tell you something
Step 2:Receive msg, topic:testTopic content:Tell you something
```

注意：

1. 广播消息没有回包。
2. 接收方可能收到多条重复的广播信息。比如，上述例子中，网络中总共有4个节点，发送发连接节点1和2，接收方连接3和4。因此，广播的时候存在四条路径[发送方 -> 节点1 -> 节点3 -> 接收方]，[发送方 -> 节点1 -> 节点4 -> 接收方]，[发送方 -> 节点2 -> 节点3 -> 接收方]，[发送方 -> 节点2 -> 节点4 -> 接收方]，则接收方SDK收到了4条信息。

发送文件:

```
# 调用AmopPublisherFile发送AMOP消息文件
# 话题名：testTopic，是否广播：false(即使用单播)，文件：dist/conf/ca.crt， 发送次数：1次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.AmopPublisherFile␣
↪testTopic false ../docs/images/FISCO_BCOS_Logo.svg 1
```

终端输出

```
3s ...
2s ...
1s ...
start test
================================================================
Step 1: Send out msg, topic:testTopic content: file ../docs/images/FISCO_BCOS_Logo.
↪svg
Step 3:Get response, { errorCode:0 error:null seq:6e6a1e23d7ca47a0a1904bcb0a151f51␣
↪content:Yes, I received! }
```

订阅者终端输出

```
Start test
Step 2:Receive file, filename length:34 filename binary:[46, 46, 47, 100, 111, 99,␣
↪115, 47, 105, 109, 97, 103, 101, 115, 47, 70, 73, 83, 67, 79, 95, 66, 67, 79, 83,
↪ 95, 76, 111, 103, 111, 46, 115, 118, 103] filename:../docs/images/FISCO_BCOS_
↪Logo.svg
```

(continues on next page)

(续上页)

```
|---save file:../docs/images/FISCO_BCOS_Logo.svg success
|---response:Yes, I received!
```

私有话题Demo

运行订阅者:

```
# 话题名：PrivateTopic，私钥文件：conf/amop/consumer_private_key.p12， 文件密码：123456
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.
↪AmopSubscriberPrivate PrivateTopic conf/amop/consumer_private_key.p12 123456
```

终端输出

```
Start test
```

单播私有话题消息:

```
# 话题名：PrivateTopic，公钥1:conf/amop/consumer_public_key_1.pem，公钥2:null， 是否广
播：false(即使用单播)，内容：Tell you something， 发送次数：2次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.
↪AmopPublisherPrivate PrivateTopic conf/amop/consumer_public_key_1.pem null false
↪"Tell you something" 2
```

广播消息:

```
# 话题名：PrivateTopic，公钥1:conf/amop/consumer_public_key_1.pem，公钥2:null， 是否广
播：true，内容：Tell you something， 发送次数：1次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.
↪AmopPublisherPrivate PrivateTopic conf/amop/consumer_public_key_1.pem null true
↪"Tell you something" 1
```

发送文件:

```
# 调用AmopPublisherPrivateFile发送AMOP消息文件
# 话题名：PrivateTopic，公钥1:conf/amop/consumer_public_key_1.pem，公钥2:null， 是否广
播：false(即使用单播)，文件：../docs/images/FISCO_BCOS_Logo.svg， 发送次数：1次
java -cp "apps/*:lib/*:conf/" org.fisco.bcos.sdk.demo.amop.tool.
↪AmopPublisherPrivateFile PrivateTopic conf/amop/consumer_public_key_1.pem null␣
↪false ../docs/images/FISCO_BCOS_Logo.svg 1
```

5. 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。

8.4.6 合约事件推送

标签： java-sdk 事件订阅 Event

1. 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。

2. 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

客户端向节点发送事件推送的注册请求:

```
// 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 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

3. Java SDK教程

注册接口

Java SDK中org.fisco.bcos.sdk.eventsub类提供合约事件的注册接口，用户可以调用subscribeEvent向节点发送注册请求，并设置回调函数。

```
  public String subscribeEvent(EventLogParams params, EventCallback callback);
```

params注册参数

事件回调请求注册的参数：

```
public class EventLogParams {
    private String fromBlock;
    private String toBlock;
    private List<String> addresses;
    private List<Object> topics;
}
```

callback回调对象

```
public interface EventCallback {
    void onReceiveLog(int status, List<EventLog> logs);
}
```

- status 回调返回状态：

```
    0        : 正常推送，此时logs为节点推送的事件日志
    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
```

(continues on next page)

(续上页)

```
    -41006  : 客户端无订阅权限
    -41007  : 事件尚未注册，取消订阅失败
    42000   : 其他错误
```

- logs表示回调的Event Log对象列表，status为0有效。默认值null，可以在子类中通过org.fisco.bcos.sdk.abi.ABICodec解析以下EventLog对象的data字段。

```
// EventLog 对象
public class EventLog {
  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;
}
```

- 实现回调对象

Java SDK对回调类EventCallback无默认实现，用户可以通过继承EventCallback类，重写onReceiveLog接口，实现自己的回调逻辑处理。

```
class SubscribeCallback implements EventCallback {
    public void onReceiveLog(int status, List<EventLog> logs) {
        // ADD CODE
    }
}
```

注意：onReceiveLog接口多次回调的logs有重复的可能性，可以根据EventLog对象中的blockNumber，transactionIndex，logIndex进行去重

topic工具

org.fisco.bcos.sdk.abi.TopicTools提供将各种类型参数转换为对应topic的工具，用户设置EventLogParams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)
}
```

4. 示例

这里以Asset合约的TransferEvent为例说明，给出合约事件推送的一些场景供用户参考。

```
contract Asset {
    event TransferEvent(int256 ret, string indexed from_account, string indexed to_
↪account, uint256 indexed amount);
    event TransferAccountEvent(string,string);

    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);

        // TransferAccountEvent 保存账号
        TransferAccountEvent(from_account, to_account);
    }
}
```

- 场景1：将链上所有/最新的事件回调至客户端

```
        // 其他初始化逻辑，省略

        // 参数设置
        EventLogParams params = new EventLogParams();

        // 全部Event fromBlock设置为"1"
        params.setFromBlock("1");

        // 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>());

        // 注册事件
        EventCallback callback = new EventCallback();
        String registerId = eventSubscribe.subscribeEvent(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
        EventLogParams params = new EventLogParams();

        // 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)"));
```

(continues on next page)

(续上页)

```
        params.setTopics(topics);

        // 注册事件
        EventCallback callback = new EventCallback();
        String registerId = eventSubscribe.subscribeEvent(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";`

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";

        // 设置参数
        EventLogParams params = new EventLogParams();

        // 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);

        // 注册事件
        EventCallback callback = new EventCallback();
        String registerId = eventSubscribe.subscribeEvent(params, 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";`

```
        // 其他初始化逻辑，省略

        // 设置参数
        EventLogParams params = new EventLogParams();

        // 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);
```

(continues on next page)

(续上页)

```
        // 注册事件
        EventCallback callback = new EventCallback();
        String registerId = eventSubscribe.subscribeEvent(params, 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"`

```
        // 其他初始化逻辑，省略

        String addr = "0x06922a844c542df030a2a2be8f835892db99f324";
        String fromAccount = "account";

        // 参数
        EventLogParams params = new EventLogParams();

        // 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);

        // 注册事件
        EventCallback callback = new EventCallback();
        String registerId = eventSubscribe.subscribeEvent(params, callback);
```

4. 解析例子

以Asset合约为例，描述合约部署、调用、注册事件及解析节点推送事件内容的实现。注意：对增加了indexed属性的事件参数，均不进行解码，在相应位置上直接记录，其余非indexed属性的事件参数将进行解码。

```
        String contractAddress = "";
        try {
            AssembleTransactionProcessor manager =
                    TransactionProcessorFactory.createAssembleTransactionProcessor(
                            client, client.getCryptoSuite().createKeyPair(),␣
↪abiFile, binFile);
            // deploy
            TransactionResponse response = manager.deployByContractLoader("Asset",␣
↪Lists.newArrayList());
            if (!response.getTransactionReceipt().getStatus().equals("0x0")) {
                return;
            }
            contractAddress = response.getContractAddress();
```

(continues on next page)

(续上页)

```
            // call function with event
            List<Object> paramsSetValues = new ArrayList<Object>();
            paramsSetValues.add("Alice");
            paramsSetValues.add("Bob");
            paramsSetValues.add(new BigInteger("100"));
            TransactionResponse transactionResponse =
                    manager.sendTransactionAndGetResponse(
                            contractAddress, abi, "transfer", paramsSetValues);
            logger.info("transaction response : " + JsonUtils.
↪toJson(transactionResponse));
        } catch (Exception e) {
            logger.error("exception:", e);
        }

        // subscribe event
        EventLogParams eventLogParams = new EventLogParams();
        eventLogParams.setFromBlock("latest");
        eventLogParams.setToBlock("latest");
        eventLogParams.setAddresses(new ArrayList<>());
        ArrayList<Object> topics = new ArrayList<>();
        ArrayList<Object> topicIdx0 = new ArrayList<>();
        CryptoSuite invalidCryptoSuite =
                new CryptoSuite(client.getCryptoSuite().getCryptoTypeConfig());
        TopicTools topicTools = new TopicTools(invalidCryptoSuite);
        topicIdx0.add(topicTools.stringToTopic("TransferEvent(int256,string,string,
↪uint256)"));
        topicIdx0.add(topicTools.stringToTopic("TransferAccountEvent(string,string)
↪"));
        eventLogParams.setTopics(topics);

        class SubscribeCallback implements EventCallback {
            public transient Semaphore semaphore = new Semaphore(1, true);

            SubscribeCallback() {
                try {
                    semaphore.acquire(1);
                } catch (InterruptedException e) {
                    logger.error("error :", e);
                    Thread.currentThread().interrupt();
                }
            }

            @Override
            public void onReceiveLog(int status, List<EventLog> logs) {
                Assert.assertEquals(status, 0);
                String str = "status in onReceiveLog : " + status;
                logger.debug(str);
                semaphore.release();

                // decode event
                if (logs != null) {
                    for (EventLog log : logs) {
                        logger.debug(
                                " blockNumber:"
                                        + log.getBlockNumber()
                                        + ",txIndex:"
                                        + log.getTransactionIndex()
                                        + " data:"
                                        + log.getData());
                        ABICodec abiCodec = new ABICodec(client.getCryptoSuite());
                        try {
```

(continues on next page)

(续上页)

```
                            List<Object> list = abiCodec.decodeEvent(abi,
↪"TransferEvent", log);
                            logger.debug("decode event log content, " + list);
                            // list = [0,␣
↪0x81376b9868b292a46a1c486d344e427a3088657fda629b5f4a647822d329cd6a,␣
↪0x28cac318a86c8a0a6a9156c2dba2c8c2363677ba0514ef616592d81557e679b6,␣
↪0x0000000000000000000000000000000000000000000000000000000000000064]
                            // 后三个事件参数均为indexed属性
                            Assert.assertEquals(4, list.size());
                        } catch (ABICodecException e) {
                            logger.error("decode event log error, " + e.
↪getMessage());
                        }
                        try {
                            List<Object> list = abiCodec.decodeEvent(abi,
↪"TransferAccountEvent", log);
                            logger.debug("decode event log content, " + list);
                            // list = [Alice, Bob]
                            Assert.assertEquals(2, list.size());
                        } catch (ABICodecException e) {
                            logger.error("decode event log error, " + e.
↪getMessage());
                        }
                    }
                }
            }
        }

        SubscribeCallback subscribeEventCallback1 = new SubscribeCallback();
        String registerId =
                eventSubscribe.subscribeEvent(eventLogParams,␣
↪subscribeEventCallback1);
        try {
            subscribeEventCallback1.semaphore.acquire(1);
            subscribeEventCallback1.semaphore.release();
            logger.info("subscribe successful, registerId is " + registerId);
        } catch (InterruptedException e) {
            logger.error("system error:", e);
            Thread.currentThread().interrupt();
        }
```

8.4.7 ABI解析

标签：`java-sdk` `abi`

ABI（Application Binary Interface）定义了调用智能合约函数以及合约之间函数调用的消息编码格式。它作为智能合约函数调用的接口说明，其内容包括了合约的函数签名、参数编码、返回结果编码等。

在Java SDK中，`org.fisco.bcos.sdk.abi.ABICodec`类提供了编码交易的输出（`data`的字段）、解析交易返回值及解析合约事件推送内容的功能。

这里以`Add.sol`合约为例，给出`ABICodec`的使用参考。

```
pragma solidity>=0.4.24 <0.6.11;

contract Add {

    uint256 private _n;
```

(continues on next page)

(续上页)

```
    event LogAdd(uint256 base, uint256 e);

    constructor() public {
        _n = 100;
    }

    function get() public view returns (uint256 n) {
        return _n;
    }

    function add(uint256 e) public returns (uint256 n) {
        emit LogAdd(_n, e);
        _n = _n + e;
        return _n;
    }
}
```

调用add(uint256)接口的交易回执内容如下，重点关注input、output和logs字段:

```
{
  // 省略 ...
  "input":
↪"0x1003e2d20000000000000000000000000000000000000000000000000000000000000000000000000000003c",
  "output":"0x00000000000000000000000000000000000000000000000000000000000000a0",
  "logs":[
      {
        // 省略 ...
        "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000064000000000000000000000000000000000
↪",
        // 省略 ...
      }
  ],
  // 省略 ...
}
```

1. 构造交易input

交易的input由两部分组成，函数选择器及调用该函数所需参数的编码。其中input的前四个字节数据（如"0x1003e2d2"）指定了要调用的函数选择器，函数选择器的计算方式为函数声明（去除空格，即add(uint256)）的哈希，取前4个字节。input的剩余部分为输入参数根据ABI编码之后的结果（如"000000000000000000000000000000000000000000000000000000000000003c"为参数"60"编码之后的结果）。

根据函数指定方式及参数输入格式的不同，ABICodec分别提供了以下接口计算交易的data。

```
  // 函数名 + Object格式的参数列表
  String encodeMethod(String ABI, String methodName, List<Object> params);
  // 函数声明 + Object格式的参数列表
  String encodeMethodByInterface(String methodInterface, List<Object> params)
  // 函数签名 + Object格式的参数列表
  String encodeMethodById(String ABI, String methodId, List<Object> params);
  // 函数名 + String格式的参数列表
  String encodeMethodFromString(String ABI, String methodName, List<String>␣
↪params);
  // 函数声明 + String格式的参数列表
  String encodeMethodByInterfaceFromString(String methodInterface, List<String>␣
↪params);
  // 函数签名 + String格式的参数列表
```

(continues on next page)

(续上页)

```
  String encodeMethodByIdFromString(String ABI, String methodId, List<String>␣
↪params);
```

以下以encodeMethod为例举例说明使用方法，其他接口的使用方法类似。

```
  // 构造参数列表
  List<Object> argsObjects = new ArrayList<Object>();
  argsObjects.add(new BigInteger("60"));
  BcosSDK sdk =  BcosSDK.build(configFile);
  Client client = sdk.getClient(Integer.valueOf(1));
  ABICodec abiCodec = new ABICodec(client.getCryptoSuite());
  String abi = ""; // 合约ABI编码，省略
  try {
    String encoded = abiCodec.encodeMethod(abi, "add", argsObjects));
    logger.info("encode method result, " + encoded);
    // encoded =
↪"0x1003e2d2000000000000000000000000000000000000000000000000000000000000003c"
  } catch (ABICodecException e) {
    logger.info("encode method error, " + e.getMessage());
  }
```

2. 解析交易返回值

根据函数指定方式及返回值类型的不同，ABICodec分别提供了以下接口解析函数返回值。

```
  // 函数名 + Object格式的返回列表
  List<Object> decodeMethod(String ABI, String methodName, String output)
  // 函数声明 + Object格式的返回列表
  List<Object> decodeMethodByInterface(String ABI, String methodInterface, String␣
↪output)
  // 函数签名 + Object格式的返回列表
  List<Object> decodeMethodById(String ABI, String methodId, String output)
  // 函数名 + String格式的返回列表
  List<String> decodeMethodToString(String ABI, String methodName, String output)
  // 函数声明 + String格式的返回列表
  List<String> decodeMethodByInterfaceToString(String ABI, String methodInterface,␣
↪String output)
  // 函数签名 + String格式的返回列表
  List<String> decodeMethodByIdToString(String ABI, String methodId, String output)
```

上述接口参数中的output为交易回执中的output字段（"0x000000000000000000000000000000000000000000000000000000000000
接口的使用方法可参考构造交易input的接口用法。

3. 解析合约事件推送内容

根据事件指定方式及解析结果类型的不同，ABICodec分别提供了以下接口解析事件内容。

```
  // 事件名 + Object格式的解析结果列表
  List<Object> decodeEvent(String ABI, String eventName, String output)
  // 事件声明 + Object格式的解析结果列表
  List<Object> decodeEventByInterface(String ABI, String eventSignature, String␣
↪output)
  // 事件签名/Topic + Object格式的解析结果列表
  List<Object> decodeEventByTopic(String ABI, String eventTopic, String output)
  // 事件名 + String格式的解析结果列表
  List<String> decodeEventToString(String ABI, String eventName, String output)
  // 事件声明 + String格式的解析结果列表
  List<String> decodeEventByInterfaceToString(String ABI, String eventSignature,␣
↪String output)
```

(continues on next page)

(续上页)

```
  // 事件签名/Topic + String格式的解析结果列表
  List<String> decodeEventByTopicToString(String ABI, String eventTopic, String␣
↪output)
```

对于事件推送，Java SDK需用户可以通过继承EventCallback类，重写onReceiveLog接口，实现自己对回调的处理逻辑。以下例子使用decodeEvent对推送的事件内容进行解析。其他接口的使用方法类似。

```
class SubscribeCallback implements EventCallback {
  public void onReceiveLog(int status, List<EventLog> logs) {
  if (logs != null) {
    String abi = ""; // 合约ABI编码，省略
    for (EventLog log : logs) {
      ABICodec abiCodec = new ABICodec(client.getCryptoSuite()); // client初始化，省
略
      try {
        List<Object> list = abiCodec.decodeEvent(abi, "LogAdd", log.getData());
        logger.debug("decode event log content, " + list);
        // list.size() = 2
      } catch (ABICodecException e) {
        logger.error("decode event log error, " + e.getMessage());
      }
    }
  }
}
```

8.4.8 远程调用接口

标签：java-sdk Client

Java SDK为区块链应用开发者提供了Java API接口，按照功能，Java API可分为如下几类：

- Client: 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；
- Precompiled: 提供调用FISCO BCOS 2.0+ Precompiled合约(预编译合约)的接口，主要包括ConsensusService、SystemConfigService、PermissionService、ChainGovernanceService、T

注解:

- Client接口声明位于 *Client.java* 文件中
- Client是群组维度的对象，可参考 快速入门 初始化Client，初始化Client时，必须传入群组ID

---

sendRawTransaction

发送交易到区块链节点。

参数

- signedTransactionData：签名后的交易

返回值

- SendTransaction: 节点收到交易后，回复给SDK的回包，包括交易哈希信息。

sendRawTransactionAsync

交易发布异步接口, 收到节点的响应之后，调用指定的callback。

参数

- signedTransactionData: 签名后的交易字符串;
- callback: SDK收到节点的回包后，调用的回调函数。

返回值

- 无

sendRawTransactionAndGetProof

向区块链节点发送交易，且声明在交易推送时交易回执中包含交易Merkle证明和交易回执Merkle证明。

参数

- signedTransactionData: 签名后的交易。

返回值

- SendTransaction: 节点收到交易后，回复给SDK的回包，包括交易哈希信息。

sendRawTransactionAndGetProofAsync

`sendRawTransactionAndGetProof`异步接口，向节点发送交易，并在收到节点回报时，调用回调函数。

参数

- signedTransactionData: SDK发送到节点的签名后交易;
- callback: SDK收到节点回包后，调用的回调函数。

返回值

- 无

sendRawTransactionAndGetReceipt

交易发送同步接口，发送交易并获取交易回执。

参数

- signedTransactionData: 组装好的交易字符串，包含签名信息。

返回值

- TransactionReceipt: 交易回执。

sendRawTransactionAndGetReceiptAsync

交易发送异步接口，发送交易，并在收到交易回执时调用指定的回调函数。

参数

- signedTransactionData: 组装好的交易字符串，包含签名信息;
- callback: 收到交易回执后调用的回调函数。

返回值

- 无。

sendRawTransactionAndGetReceiptWithProof

交易发送同步接口，发送交易，并获取带有交易Merkle证明和交易回执Merkle证明的交易回执。

参数

- signedTransactionData: 组装好的交易字符串，包含签名信息;

返回值

- TransactionReceipt: 带有交易Merkle证明和交易回执Merkle证明的交易回执。

sendRawTransactionAndGetReceiptWithProofAsync

交易发送异步接口，发送交易，并在收到交易回执时，调用指定的回调函数(交易回执中包含交易Merkle证明和交易回执的Merkle证明)。

参数

- signedTransactionData: 组装好的交易字符串，包含签名信息;
- callback: 收到交易回执后调用的回调函数。

返回值

- 无。

call

向节点发送请求，调用合约常量接口。

参数

- transaction: 合约调用信息，包含合约地址、合约调用者以及调用的合约接口和参数的abi编码

返回值

- Call: 合约常量接口的返回结果，包括当前块高、接口执行状态信息以及接口执行结果

callAsync

合约常量接口异步调用，接收到节点返回的合约接口执行结果后，执行指定的回调函数

参数

- transaction: 合约调用信息，包含合约地址、合约调用者以及调用的接口和参数信息;
- callback: 回调函数。

返回值

- 无

getBlockNumber

获取Client对象对应的群组最新块高。

参数

- 无

返回值

- BlockNumber: Client对象对应的群组最新区块高度。

getCode

查询指定合约地址对应的合约代码信息。

参数

- address: 合约地址。

返回值

- Code: 合约地址对应的合约代码。

getTotalTransactionCount

获取Client对应群组的交易统计信息，包括上链的交易数、上链失败的交易数目。

参数

- 无

返回值

- TotalTransactionCount: 交易统计信息，包括：
  - txSum: 上链的交易总量
  - blockNumber: 群组的当前区块高度
  - failedTxSum: 上链执行异常的交易总量

getBlockByHash

根据区块哈希获取区块信息。

参数

- blockHash: 区块哈希；
- returnFullTransactionObjects: true/false，表明获取的区块信息中是否包含完整的交易信息；
  - true: 节点返回的区块中包含完整的交易信息；
  - false: 节点返回的区块中仅包含交易哈希。

返回值

- BcosBlock: 查询获取的区块信息。

getBlockByNumber

根据区块高度获取区块信息。

参数

- blockNumber: 区块高度；
- returnFullTransactionObjects: true/false，表明获取的区块信息中是否包含完整的交易信息；
  - true: 节点返回的区块中包含完整的交易信息；
  - false: 节点返回的区块中仅包含交易哈希。

返回值

- BcosBlock: 查询获取的区块信息

getBlockHashByNumber

根据区块高度获取区块哈希。

参数

- blockNumber: 区块高度。

返回值

- BlockHash: 指定区块高度对应的区块哈希

getBlockHeaderByHash

根据区块哈希获取区块头信息。

参数

- blockHash: 区块哈希;
- returnSignatureList: true/false，表明返回的区块头中是否附带签名列表信息
  - true: 返回的区块头中带有区块签名列表信息;
  - false: 返回的区块头中不带区块签名列表信息。

返回值

- BcosBlockHeader: 指定区块哈希对应的区块头。

getBlockHeaderByNumber

根据区块高度获取区块头信息。

参数

- blockHash: 区块高度;
- returnSignatureList: true/false，表明返回的区块头中是否附带签名列表信息
  - true: 返回的区块头中带有区块签名列表信息;
  - false: 返回的区块头中不带区块签名列表信息。

返回值

- BcosBlockHeader: 指定区块高度对应的区块头。

getTransactionByHash

根据交易哈希获取交易信息。

参数

- transactionHash: 交易哈希

返回值

- BcosTransaction: 指定哈希对应的交易信息。

getTransactionByHashWithProof

根据交易哈希获取交易信息，交易信息中带有交易Merkle证明。

参数

- transactionHash: 交易哈希

返回值

- BcosTransaction: 指定哈希对应的交易信息。

getTransactionByBlockNumberAndIndex

根据区块高度和交易索引获取交易信息。

参数

- blockNumber: 交易所在的区块高度;
- transactionIndex: 交易索引。

返回值

- BcosTransaction: 指定区块高度和交易索引对应的交易信息。

getTransactionByBlockHashAndIndex

根据区块哈希和交易索引获取交易信息。

参数

- blockHash: 交易所在的区块哈希;
- transactionIndex:交易索引。

返回值

- BcosTransaction: 指定区块哈希和交易索引对应的交易信息。

getTransactionReceipt

根据交易哈希获取交易回执信息。

参数

- transactionHash: 交易哈希

返回值

- BcosTransactionReceipt: 交易哈希对应的回执信息。

getTransactionReceiptByHashWithProof

根据交易哈希获取交易回执信息，回执中带有Merkle证明。

参数

- transactionHash: 交易哈希。

返回值

- TransactionReceiptWithProof: 带有Merkle证明的交易回执信息。

getPendingTransaction

获取交易池内待处理的交易列表。

参数

- 无

返回值

- PendingTransactions: 交易池内未处理的交易列表。

getPendingTxSize

获取交易池内未处理的交易数目。

参数

- 无

返回值

- PendingTxSize: 交易池内未处理的交易数目。

getBlockLimit

获取Client对应群组的BlockLimit，BlockLimit主要用于交易防重。

参数

- 无

返回值

- BigInteger: 群组的BlockLimit。 /**
  - Get cached block height
  -
  - @return block number */ BigInteger getBlockLimit();

generateGroup

为指定节点动态创建一个新群组。

参数

- groupId: 创建的群组ID;
- timestamp: 新创建的群组创世块时间戳;
- enableFreeStorage: 新创建群组是否使用`enable_free_storage`的gas计算模式，推荐设置为`false`;
- nodeList: 新创建群组的共识节点NodeID列表;
- peerIpPort: 创建该群组的节点`IP:Port`信息。

返回值

- GenerateGroup: 群组创建结果。

startGroup

启动指定节点的指定群组。

参数

- groupId: 需要启动的群组ID;
- peerIpPort: 启动指定群组的节点`IP:Port`信息。

返回值

- StartGroup: 群组启动状态。

stopGroup

停止指定节点的指定群组。

参数

- groupId: 需要停止的群组ID;
- peerIpPort: 群组所在的节点`IP:Port`信息。

返回值

- StopGroup: 被停止的群组状态。

removeGroup

删除指定节点的指定群组。

参数

- groupId: 被删除的群组信息;
- peerIpPort: 群组所在的节点`IP:Port`信息。

返回值

- RemoveGroup: 被删除的群组状态。

recoverGroup

恢复指定节点被删除的群组。

参数

- groupId: 需要恢复的群组ID;
- peerIpPort: 群组所在的节点`IP:Port`信息。

返回值

- RecoverGroup: 被恢复的群组状态。

queryGroupStatus

查询指定群组的状态。

参数

- groupId: 被查询的群组ID;
- peerIpPort: 群组状态查询发送到的目标节点信息，包括`IP:Port`信息。

返回值

- QueryGroupStatus: 被查询的群组状态。

getGroupList

获取指定节点的群组列表。

参数

- peerIpPort: 被查询的节点的`IP:Port`。

返回值

- GroupList: 指定节点的群组列表。

getGroupPeers

获取指定节点指定群组连接的节点列表。

参数

- peerIpPort: 被查询的节点的`IP:Port`。

返回值

- GroupPeers: 指定群组连接的节点列表。

getPeers

获取指定节点的网络连接信息。

参数

- endpoint: 被查询的节点的`IP:Port`。

返回值

- Peers: 指定节点的网络连接信息。

getNodeIDList

获取指定节点连接的节点列表。

参数

- endpoint: 被查询的节点的`IP:Port`。

返回值

- NodeIDList: 指定节点连接的节点列表。

getObserverList

获取Client对应群组的观察节点列表。

参数

- 无

返回值

- ObserverList: 观察节点列表。

getSealerList

获取Client对应群组的共识节点列表。

参数

- 无

返回值

- SealerList: 共识节点列表。

getPbftView

节点使用PBFT共识算法时，获取PBFT视图信息。

参数

- 无

返回值

- PbftView: PBFT视图信息。

getNodeVersion

获取节点版本信息。

参数

- ipAndPort: 请求发送的目标节点，包括`IP:Port`信息。

返回值

- NodeVersion: 查询获取的节点版本信息。

getNodeInfo

获取节点的NodeID，Topic等信息。

参数

- ipAndPort: 请求发送的目标节点，包括`IP:Port`信息。

返回值

- NodeInfo: 查询获取的节点版本信息。

getConsensusStatus

获取节点共识状态。

参数

- 无

返回值

- ConsensusStatus: 节点共识状态。

getSystemConfigByKey

根据指定配置关键字获取系统配置项的值。

参数

- key: 系统配置项，目前包括`tx_count_limit`, `tx_gas_limit`, `rpbft_epoch_sealer_num`, `rpbft_epoch_block_num`和`consensus_timeout`.

返回值

- SystemConfig: 系统配置项的值。

getSyncStatus

获取节点同步状态。

参数

- 无

返回值

- SyncStatus: 区块链节点同步状态。

ConsensusService

addSealer

将指定节点添加为共识节点。

参数

- nodeId: 被添加为共识节点的node ID.

返回值

- RetCode: 共识节点添加结果。

注解： 为了保证新节点加入不影响共识，即将被添加为共识节点的节点必须与群组内其他节点建立P2P网络连接，否则其无法被添加为共识节点。

---

addObserver

将指定节点添加为观察节点。

参数

- nodeId: 被添加为观察节点的node ID.

返回值

- RetCode: 观察节点添加结果。

removeNode

将指定节点移出群组。

参数

- nodeId: 被移出群组的节点的node ID.

返回值

- RetCode: 节点被移出群组的执行结果。

SystemConfigService

setValueByKey

设置指定系统配置项的值。

参数

- key: 配置项，目前支持`tx_count_limit`, `tx_gas_limit`, `rpbft_epoch_block_num`, `rpbft_epoch_sealer_num`和`consensus_time`;
- value: 系统配置项被设置的值。

返回值

- RetCode: 系统配置项设置结果。

PermissionService

grantPermission

为指定用户添加对指定表的写权限。

参数

- tableName: 授权访问的表名;
- userAddress: 被授权对指定表写操作权限的账户地址。

返回值

- RetCode: 授权结果。

revokePermission

撤销指定用户对指定表的写权限。

参数

- tableName: 撤销写访问权限的表名;
- userAddress: 被撤销对指定表写访问权限的账户地址。

返回值

- RetCode: 权限撤销结果。

queryPermission

查询对指定合约有权限的账户信息。

参数

- contractAddress: 被查询的合约地址。

返回值

- List: 拥有对指定合约写权限的账户信息列表。

grantWrite

授权指定用户对指定合约的写权限。

参数

- contractAddress: 授权写访问的合约地址;
- userAddress: 被授权对指定合约写访问权限的账户地址。

返回值

- RetCode: 授权结果。

revokeWrite

撤销指定用户对指定合约的写访问权限。

参数

- contractAddress: 撤销写访问权限的合约地址;
- userAddress: 被撤销对指定合约写访问权限的账户地址。

返回值

- RetCode: 合约写访问权限撤销结果。

queryPermissionByTableName

查询对指定表有写权限的账户地址信息。

参数

- tableName: 查询权限信息的表名。

返回值

- List: 拥有对指定表写访问权限的账户信息列表。

grantDeployAndCreateManager

授权指定用户部署和调用合约的权限。

参数

- userAddress: 被授权部署和调用合约权限的账户地址。

返回值

- RetCode: 授权结果。

revokeDeployAndCreateManager

撤销指定账户部署和调用合约的权限。

参数

- userAddress: 被撤销部署和调用合约权限的账户地址。

返回值

- RetCode: 指定用户合约部署和调用权限的撤销结果。

listDeployAndCreateManager

列出拥有部署和调用合约权限的账户信息。

参数

- 无

返回值

- 拥有部署和调用合约权限的账户信息列表。

grantPermissionManager

授予指定用户权限控制的权限，被授权的账户可配置普通账户的访问权限。

参数

- userAddress: 被授予权限控制权限的账户地址。

返回值

- RetCode: 授权结果

revokePermissionManager

撤销指定用户权限控制的权限，被撤销的账户不可配置普通账户的对合约、表的访问权限。

参数

- userAddress: 被撤销权限的账户地址。

返回值

- RetCode: 权限撤销结果。

listPermissionManager

列出拥有权限配置权力的账户信息。

参数

- 无

返回值

- List: 拥有权限控制权限的账户信息列表。

grantNodeManager

授权指定用户操作系统共识表的权限，被授权用户可增加共识节点、增加观察者节点、删除节点。

参数

- userAddress: 被授权的账户地址。

返回值

- RetCode: 授权结果。

revokeNodeManager

撤销指定用户对系统共识表的写访问权限，撤销权限的账户无法进行增加共识节点、增加观察者节点、删除节点等操作。

参数

- userAddress: 被撤销的账户地址。

返回值

- RetCode: 系统够共识表写访问权限撤销结果。

listNodeManager

列出拥有增加共识节点、增加观察者节点、删除节点等操作权限的账户信息。

参数

- 无

返回值

- List: 有增加共识节点、增加观察者节点、删除节点等操作权限的账户信息。

grantCNSManager

授权指定账户操作CNS表的权限。

参数

- userAddress: 被授权账户地址。

返回值

- RetCode: 授权结果。

revokeCNSManager

撤销指定账户对CNS表的写权限。

参数

- userAddress: 被撤销的账户地址。

返回值

- RetCode: 撤销结果。

listCNSManager

列出对CNS表有写权限的账户信息。

参数

- 无

返回值

- List: 对CNS表有写权限的账户列表。

grantSysConfigManager

授权指定用户设置系统配置项的权限。

参数

- userAddress: 被授权账户地址。

返回值

- RetCode: 授权结果。

revokeSysConfigManager

撤销指定用户设置系统配置项的权限。

参数

- userAddress: 被撤销账户地址。

返回值

- RetCode: 权限撤销结果。

listSysConfigManager

列出有修改系统配置项权限的账户信息。

参数

- 无

返回值

- List: 有修改系统配置项权限的账户信息。

ChainGovernanceService

grantCommitteeMember

授权指定用户为链管理员

参数

- userAddress: 被授权为链管理员的账户地址。

返回值

- RetCode: 授权结果。

revokeCommitteeMember

撤销指定链管理员为普通账户。

参数

- userAddress: 被撤销的链管理员账户地址。

返回值

- RetCode: 撤销结果。

listCommitteeMembers

列出当前链管理员信息。

参数

- 无

返回值

- List: 所有链管理员的信息。

updateCommitteeMemberWeight

修改指定链管理员的投票权重。

参数

- userAddress: 更新投票权重的链管理员账户地址;
- weight: 更新后的投票权重。

返回值

- RetCode: 连管理员投票权重更新结果。

updateThreshold

修改投票生效阈值。

参数

- threshold: 更新后的投票生效阈值。

返回值

- RetCode: 投票生效阈值更新结果。

queryThreshold

查询当前的投票生效阈值。

参数

- 无

返回值

- BigInteger: 当前投票生效阈值。

queryCommitteeMemberWeight

查询指定链管理员的投票权重。

参数

- userAddress: 被查询投票权重的链管理员账户地址。

返回值

- BigInteger: 查询到的指定链管理员投票权重。

grantOperator

将指定账户授权为运维管理员。

参数

- userAddress: 被授权为运维管理员的账户地址。

返回值

- RetCode: 授权结果。

revokeOperator

撤销指定运维管理员的运维权限。

参数

- userAddress: 被撤销的运维管理员账户地址。

返回值

- RetCode: 运维权限撤销结果。

listOperators

列出当前群组的运维管理员信息。

参数

- 无

返回值

- List: 运维管理员列表。

queryVotesOfMember

查询指定账户被选举为委员账户的投票情况

参数

- `account`: 被查询的账户地址

返回值

- `String`: 返回该账户被选举为委员账户的投票情况；若没有任何委员选举该账户，则返回null字符串

queryVotesOfThreshold

查询updateThreshold的投票情况

参数

- 无

返回值

- String: updateThreshold的投票情况，当没有任何委员更新threshold时，返回null字符串。

freezeAccount

冻结指定账户，被冻结的账户不能部署和调用合约。

参数

- userAddress: 被冻结的账户地址。

返回值

- RetCode: 账户冻结结果。

unfreezeAccount

解冻被冻结的账户，解冻后的账户可以部署和调用合约。

参数

- userAddress: 解冻的账户地址。

返回值

- RetCode: 账户解冻结果。

getAccountStatus

获取指定账户的状态。

参数

- userAddress: 账户地址。

返回值

- String: 指定账户的状态。

TableCRUDService

createTable

创建用户表。

参数

- tableName: 创建的用户表名;
- keyFieldName: 用户表的主key名;
- valueFields: 用户表的fields.

返回值

- RetCode: 用户表创建结果。

insert

向指定用户表中插入一条记录。

参数

- tableName: 需要插入记录的表名;
- key: 主key被设置的值;

- fieldNameToValue: 每个field到其对应值的映射。

返回值

- RetCode: 记录是否插入成功。

update

更新指定用户表中，将指定主key对应的记录更新为传入的记录。

参数

- tableName: 用户表名;
- key: 需要更新的记录对应的主key值;
- fieldNameToValue: 更新后的记录;
- condition: 记录更新条件。

返回值

- RetCode: 记录是否更新成功。

remove

删除用户表指定记录。

参数

- tableName: 用户表名;
- key: 被删除的记录主key值;
- condition: 被删除记录的匹配条件。

返回值

- RetCode: 记录删除结果。

select

查询用户表指定记录。

参数

- tableName: 被查询的用户表名;
- key: 被查询的主key值;
- condition: 查询条件。

返回值

- List<Map<String, String>>: 查询结果。

desc

获取指定用户表的描述信息。

参数

- tableName: 被查询的用户表名。

返回值

- List<Map<String, String>>: 用户表描述信息，记录了PrecompiledConstant.KEY_NAME到主key的映射，以及PrecompiledConstant.FIELD_NAME到所有field的映射，field之间用逗号分隔开。

asyncInsert

insert的异步接口，向指定表插入指定记录，并在接收到节点的回执后，调用指定回调函数。

参数

- tableName: 需要插入记录的表名;
- key: 主key被设置的值;
- fieldNameToValue: 每个field到其对应值的映射;
- callback: 回调函数。

返回值

- 无

asyncUpdate

update的异步接口，更新指定记录，并在接收到节点的回执后，调用指定回调函数。

参数

- tableName: 用户表名;
- key: 需要更新的记录对应的主key值;
- fieldNameToValue: 更新后的记录;
- condition: 记录更新条件;
- callback: 回调函数。

返回值

- 无

asyncRemove

remove的异步接口，删除指定记录，并在接收到节点的回执后，调用指定回调函数。

参数

- tableName: 用户表名;
- key: 被删除的记录主key值;
- condition: 被删除记录的匹配条件;
- callback: 回调函数。

返回值

- 无

ContractLifeCycleService

freeze

冻结指定合约。

参数

- contractAddress: 被冻结的合约地址。

返回值

- RetCode: 合约冻结结果。

unfreeze

解冻指定合约。

参数

- contractAddress: 被解冻的合约地址。

返回值

- RetCode: 合约解冻结果。

grantManager

给指定账户授权冻结/解冻指定合约的权限。

参数

- contractAddress: 指定的合约地址;
- userAddress: 被授权对指定合约冻结/解冻权限的账户地址。

返回值

- RetCode: 合约生命周期管理权限授权结果。

getContractStatus

获取指定合约的状态。

参数

- contractAddress: 合约地址。

返回值

- String: 合约状态。

listManager

获取对某合约有管理权限的账户地址。

参数

- contractAddress: 合约地址。

返回值

- List: 对指定合约有冻结/解冻权限的账户信息。

CNSService

registerCNS

为指定合约注册CNS。

参数

- contractName: 合约名;
- contractVersion: 注册的合约版本;
- contractAddress: 合约地址;
- abiData: 注册合约的abi;

返回值

- RetCode: CNS注册结果。

selectByName

根据合约名查询合约CNS信息。

参数

- contractName: 需要查询CNS信息的合约名;

返回值

- List: 查询到的CNS信息列表。

selectByNameAndVersion

根据合约名和合约版本查询CNS信息。

参数

- contractName: 合约名;
- contractVersion: 合约版本。

返回值

- List: 查询到的CNS信息。

getContractAddress

根据合约名和合约版本获取合约地址。

参数

- contractName: 合约名;
- contractVersion: 合约版本。

返回值

- String: 查询到的合约地址。

getBatchReceiptsByBlockNumberAndRange

根据区块高度和交易范围，批量返回群组内的交易回执信息。

参数

- blockNumber: 请求获取的回执信息所在的区块高度;
- from: 需要获取的回执起始索引;
- count: 需要批量获取的回执数目，当设置为-1时，返回区块内所有回执信息;

返回值

- BcosTransactionReceiptsDecoder: 包含压缩编码信息的交易回执数据，调用decodeTransactionReceiptsInfo方法可将压缩的交易回执转换为TransactionReceiptsInfo对象，并可通过该对象获取交易回执具体信息

示例:

```
    // 获取最新区块高度的所有交易回执信息(cient初始化过程省略，详细可以参考快速入门)
   BcosTransactionReceiptsDecoder bcosTransactionReceiptsDecoder =
                    client.getBatchReceiptsByBlockNumberAndRange(
                            client.getBlockNumber().getBlockNumber(), "0", "-1");
    // 解码交易回执信息
    BcosTransactionReceiptsInfo.TransactionReceiptsInfo receiptsInfo =␣
↪bcosTransactionReceiptsDecoder.decodeTransactionReceiptsInfo();
    // 获取回执所在的区块信息
    BcosTransactionReceiptsInfo.BlockInfo blockInfo = receiptsInfo.getBlockInfo();
    // 获取交易回执列表
    List<TransactionReceipt> receiptList = receiptsInfo.getTransactionReceipts();
```

getBatchReceiptsByBlockHashAndRange

根据区块哈希和交易范围，批量返回群组内的交易回执信息。

参数

- blockHash: 请求获取的回执信息所在的区块哈希;
- from: 需要获取的回执起始索引;
- count: 需要批量获取的回执数目，当设置为-1时，返回区块内所有回执信息;

返回值

- BcosTransactionReceiptsDecoder: 包含压缩编码信息的交易回执数据，调用decodeTransactionReceiptsInfo方法可将压缩的交易回执转换为TransactionReceiptsInfo对象，并可通过该对象获取交易回执具体信息

示例:

```
    // 获取最新区块的所有交易回执信息(cient初始化过程省略，详细可以参考快速入门)
    BcosTransactionReceiptsDecoder bcosTransactionReceiptsDecoder =
                    client.getBatchReceiptsByBlockHashAndRange(
                            client.getBlockHashByNumber(client.getBlockNumber().
↪getBlockNumber())
                                    .getBlockHashByNumber(),
                            "0",
                            "-1");
    // 解码交易回执信息
```

(continues on next page)

(续上页)

```
    BcosTransactionReceiptsInfo.TransactionReceiptsInfo receiptsInfo =␣
↪bcosTransactionReceiptsDecoder.decodeTransactionReceiptsInfo();
    // 获取回执所在的区块信息
    BcosTransactionReceiptsInfo.BlockInfo blockInfo = receiptsInfo.getBlockInfo();
    // 获取交易回执列表
    List<TransactionReceipt> receiptList = receiptsInfo.getTransactionReceipts();
```

8.4.9 账户管理

标签: java-sdk 设置账户

---

java sdk提供账户管理接口，支持以下功能:

- **账户加载**: 从指定路径加载账户，同时支持pem和p12格式的账户文件加载
- **账户生成**: 随机生成账户公私钥对
- **账户保存**: 保存账户信息
- **提供加载p12和pem文件解析的接口**

注解: java sdk提供了接口级别的账户生成方法，工具级别的账户生成脚本请参考 get_account.sh脚本 .

---

1. 账户加载

java sdk的org.fisco.bcos.sdk.crypto.CryptoSuite提供账户加载功能，默认从配置文件的[account]配置项加载交易发送账户，具体请参考这里.

从指定pem账户文件加载交易发送账户的示例如下(client初始化方法请参考快速入门):

```
// 从pemAccountFilePath指定路径加载pem账户文件，并将其设置为交易发送账户
public void loadPemAccount(Client client, String pemAccountFilePath)
{
    // 通过client获取CryptoSuite对象
    CryptoSuite cryptoSuite = client.getCryptoSuite();
    // 加载pem账户文件
    cryptoSuite.loadAccount("pem", pemAccountFilePath, null);
}
```

从指定的p12账户文件加载交易发送账户的示例如下:

```
// 从p12AccountFilePath指定的路径加载p12账户文件，并将其设置为交易发送账户
public void loadP12Account(Client client, String p12AccountFilePath, String␣
↪password)
{
    // 通过client获取CryptoSuite对象
    CryptoSuite cryptoSuite = client.getCryptoSuite();
    // 加载pem账户文件
    cryptoSuite.loadAccount("p12", p12AccountFilePath, password);
}
```

2. 账户生成

java sdk的org.fisco.bcos.sdk.crypto.CryptoSuite提供了账户生成功能。

随机生成非国密账户示例如下:

```
// 创建非国密类型的CryptoSuite
CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
// 随机生成非国密公私钥对
CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
// 获取账户地址
String accountAddress = cryptoKeyPair.getAddress();
```

随机生成国密账户示例如下：

```
// 创建国密类型的CryptoSuite
CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
// 随机生成国密公私钥对
CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
// 获取账户地址
String accountAddress = cryptoKeyPair.getAddress();
```

3. 账户保存

当没有自定义加载账户，也没有通过配置文件配置账户信息时(账户配置请参考这里)，java sdk会随机生成账户发送交易，java sdk的org.fisco.bcos.sdk.crypto.CryptoSuite提供账户保存功能，可将随机生成的账户保存在指定路径。

以pem的格式保存账户文件到指定路径的示例如下：

```
// 将随机生成的账户信息保存在pemFilePath指定的路径
public void saveAccountWithPem(CryptoKeyPair cryptoKeyPair, String pemFilePath)
{
    // 以pem的格式保存账户文件到pemFilePath路径
    cryptoKeyPair.storeKeyPairWithPem(pemFilePath);
}

// 将随机生成的账户信息保存在账户配置${keyStoreDir}指定的目录下
public void saveAccountWithPemToKeyStoreDir(CryptoKeyPair cryptoKeyPair)
{
    // 账户文件名为${accountAddress}.pem
    cryptoKeyPair.storeKeyPairWithPemFormat();
}
```

以p12的格式保存账户文件到指定路径的示例如下：

```
public void saveAccountWithP12(CryptoKeyPair cryptoKeyPair, String p12FilePath,␣
↪String password)
{
    // 以p12的格式将账户文件保存到p12FilePath路径
    cryptoKeyPair.storeKeyPairWithP12(p12FilePath, password);
}

// 将随机生成的账户信息保存在账户配置${keyStoreDir}指定的目录下
public void saveAccountWithP12ToKeyStoreDir(CryptoKeyPair cryptoKeyPair, String␣
↪password)
{
    // 账户文件名为${accountAddress}.p12
    cryptoKeyPair.storeKeyPairWithP12Format(password);
}
```

java sdk随机生成的账户信息可通过如下方法获取：

```
public CryptoKeyPair getCreatedCryptoKeyPair(Client client)
{
    return client.getCryptoSuite().getCryptoKeyPair();
}
```

4. p12和pem文件解析接口

java sdk的org.fisco.bcos.sdk.crypto.keystore.KeyManager提供p12和pem文件解析接口。

从指定文件加载pem文件示例如下：

```
public KeyManager loadPem(String pemFilePath)
{
    return new PEMManager(pemFilePath);
}
```

从指定文件加载pem文件示例如下：

```
public KeyManger loadP12(String p12FilePath, String password)
{
    return new P12Manager(p12FilePath, password);
}
```

pem和p12解析后生成的KeyManager对象提供了访问公私钥信息的接口如下：

```
// 获取公私钥对信息
public KeyPair getKeyPair();
```

此外，org.fisco.bcos.sdk.crypto.CryptoSuite也提供了将java.security.KeyPair类型的公私钥信息转换为CryptoKeyPair的功能，示例如下：

```
// KeyManager中维护的是非国密公私钥信息
public CryptoKeyPair loadKeyManager(KeyManager keyManager)
{
    KeyPair keyPair = keyManager.getKeyPair();
    CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
    return cryptoSuite.createKeyPair(keyPair);
}

// KeyManager中维护的是国密公私钥信息
public CryptoKeyPair loadGMKeyManager(KeyManager GMKeyManager)
{
    KeyPair keyPair = GMKeyManager.getKeyPair();
    CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
    return cryptoSuite.createKeyPair(keyPair);
}
```

8.4.10 密码学模块

标签：java-sdk Crypto

---

java sdk提供了可访问所有密码学相关接口的CryptoSuite，CryptoSuite会根据传入的cryptoType(目前支持CryptoType.ECDSA_TYPE和CryptoType.SM_TYPE，前者用在非国密链中，后者用于国密链中)初始化密码学相关的套件。

java sdk目前支持以下功能:

- **计算哈希**: 支持sm3和keccak256两种哈希算法，一般国密采用前者，非国密采用后者;
- **签名/验签** : 支持sm2和secp256k1两种签名和验签方法，一半国密采用前者，非国密采用后者。

创建CryptoSuite

java sdk目前支持创建非国密、国密类型的CryptoSuite。

创建非国密类型的CryptoSuite的示例如下：

```
    public CryptoSuite createECDSACryptoSuite()
    {
        return new CryptoSuite(CryptoType.ECDSA_TYPE);
    }
```

创建国密类型的CryptoSuite的示例如下：

```
  public CryptoSuite createSMCryptoSuite()
    {
        return new CryptoSuite(CryptoType.SM_TYPE);
    }
```

哈希接口

初始化密码学套件CryptoSuite后，用户可直接使用创建的CryptoSuite调用哈希接口，也可自定义创建指定的哈希类，并调用哈希算法。

使用CryptoSuite调用哈希接口

```
    /// 调用keccak256哈希算法
    public String calculateHashWithkeccak256(String data)
    {
        // 创建非国密的CryptoSuite
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 使用cryptoSuite调用hash算法，返回十六进制哈希字符串
        return cryptoSuite.hash(data);
    }

    public byte[] calculateHashWithkeccak256(byte[] data)
    {
         // 创建非国密的CryptoSuite
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 使用cryptoSuite调用hash算法
        // 返回二进制数组，可通过Hex.toHexString(result)将其转换成十六进制字符串
        byte[] result = cryptoSuite.hash(data);
        return result;
    }

    /// 调用sm3哈希算法
    public String calculateHashWithSM3(String data)
    {
        // 创建国密的CryptoSuite
        CryptoSuite SMcryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 使用SMcryptoSuite调用hash算法，返回十六进制哈希字符串
        return SMcryptoSuite.hash(data);
    }

    public byte[] calculateHashWithSM3(byte[] data)
    {
         // 创建非国密的CryptoSuite
        CryptoSuite SMcryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 使用SMcryptoSuite调用hash算法
        // 返回二进制数组，可通过Hex.toHexString(result)将其转换成十六进制字符串
        byte[] result = SMcryptoSuite.hash(data);
        return result;
    }
```

创建指定方法的哈希对象调用哈希接口

```
    /// 调用keccak256哈希算法
    public String calculateHashWithkeccak256(String data)
    {
        // 创建keccak256对应的对象
        Keccak256 hasher = new Keccak256();
        // 返回十六进制哈希字符串
        return hasher.hash(data);
    }

    public byte[] calculateHashWithkeccak256(byte[] data)
    {
        // 创建keccak256对应的对象
        Keccak256 hasher = new Keccak256();
        // 返回二进制数组，可通过Hex.toHexString将其转换为十六进制字符串
        return hasher.hash(data);
    }

    /// 调用sm3哈希算法
    public String calculateHashWithSM3(String data)
    {
        // 创建sm3对应的对象
        SM3Hash hasher = new SM3Hash();
        // 返回十六进制哈希字符串
        return hasher.hash(data);
    }

    public byte[] calculateHashWithSM3(byte[] data)
    {
        // 创建sm3对应的对象
        SM3Hash hasher = new SM3Hash();
        // 返回二进制数组，可通过Hex.toHexString将其转换为十六进制字符串
        return hasher.hash(data);
    }
```

签名/验签接口

初始化密码学套件CryptoSuite后，用户可直接使用创建的CryptoSuite调用签名和验签接口，也可自定义创建指定的签名验签对象，调用签名和验签接口。

注解： 签名/验签接口传入的明文数据必须是哈希，生成指定明文的签名前，须计算其哈希，并将哈希结果作为签名原文传入接口生成签名

使用CryptoSuite调用哈希接口

非国密签名/验签接口调用示例如下：

```
    /// 生成secp256k1签名
    public ECDSASignatureResult  generateSigantureWithSecp256k1(String data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 生成CryptoKeyPair
        CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
        // 计算传入数据的哈希(keccak256哈希算法)
        String hashData = cryptoSuite.hash(data);
```

(continues on next page)

(续上页)

```
        // 生成签名
        return (ECDSASignatureResult)(cryptoSuite.sign(hashData, cryptoKeyPair));
    }

    // 当入参为byte[]
    public ECDSASignatureResult generateSigantureWithSecp256k1(byte[] data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 生成CryptoKeyPair
        CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
        // 计算传入数据的哈希(keccak256哈希算法)
        byte[] hashData = cryptoSuite.hash(data);
        // 生成签名
        return (ECDSASignatureResult)(cryptoSuite.sign(hashData, cryptoKeyPair));
    }

    /// 验证签名
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪keyPair, String data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 计算data的哈希(keccak256k1哈希算法)
        String hashData = cryptoSuite.hash(data);
        // 验证签名
        return cryptoSuite.verify(keyPair.getHexPublicKey(), hashData,␣
↪signatureResult.convertToString());
    }
    // 入参为byte[]
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪keyPair, byte[]ß data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.ECDSA_TYPE);
        // 计算data的哈希(keccak256k1哈希算法)
        byte[] hashData = cryptoSuite.hash(data);
        // 验证签名
        return cryptoSuite.verify(keyPair.getHexPublicKey(), hashData,␣
↪signatureResult.convertToString());
    }
```

类似地，国密签名/验签接口调用示例如下：

```
    /// 生成sm2签名
    public SM2SignatureResult  generateSigantureWithSM2(String data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 生成CryptoKeyPair
        CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
        // 计算传入数据的哈希(sm3哈希算法)
        String hashData = cryptoSuite.hash(data);
        // 生成签名
        return (SM2SignatureResult)(cryptoSuite.sign(hashData, cryptoKeyPair));
    }

    // 当入参为byte[]
    public SM2SignatureResult generateSigantureWithSM2(byte[] data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 生成CryptoKeyPair
        CryptoKeyPair cryptoKeyPair = cryptoSuite.createKeyPair();
        // 计算传入数据的哈希(sm3哈希算法)
        byte[] hashData = cryptoSuite.hash(data);
```

(continues on next page)

(续上页)

```
        // 生成签名
        return (SM2SignatureResult)(cryptoSuite.sign(hashData, cryptoKeyPair));
    }

    /// 验证签名
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪keyPair, String data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 计算data的哈希(sm3哈希算法)
        String hashData = cryptoSuite.hash(data);
        // 验证签名
        return cryptoSuite.verify(keyPair.getHexPublicKey(), hashData,␣
↪signatureResult.convertToString());
    }
    // 入参为byte[]
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪keyPair, byte[]ß data)
    {
        CryptoSuite cryptoSuite = new CryptoSuite(CryptoType.SM_TYPE);
        // 计算data的哈希(sm3哈希算法)
        byte[] hashData = cryptoSuite.hash(data);
        // 验证签名
        return cryptoSuite.verify(keyPair.getHexPublicKey(),  Hex.
↪toHexString(hashData), signatureResult.convertToString());
    }
```

创建指定方法的签名验签对象调用签名验签接口

非国密签名/验签接口调用示例如下(签名密钥对的生成可参考这里):

```
    // 生成secp256k1签名
    public ECDSASignatureResult generateSignatureWithSecp256k1(CryptoKeyPair␣
↪ecdsaKeyPair, String data)
    {
        // 生成secp256k1签名对象
        ECDSASignature signer = new ECDSASignature();
        // 计算data的哈希(keccak256)
        Keccak256 hasher = new Keccak256();
        String hashData = hasher.hash(data);
        return (ECDSASignatureResult)signer.sign(hashData, ecdsaKeyPair);
    }

    // data类型为byte[]
    public ECDSASignatureResult generateSignatureWithSecp256k1(CryptoKeyPair␣
↪ecdsaKeyPair, byte[] data)
    {
        // 生成secp256k1签名对象
        ECDSASignature signer = new ECDSASignature();
        // 计算data的哈希(keccak256)
        Keccak256 hasher = new Keccak256();
        byte[] hashData = hasher.hash(data);
        return (ECDSASignatureResult)signer.sign(hashData, ecdsaKeyPair);
    }

    /// 验证secp256k1签名
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪ecdsaKeyPair, String data)
    {
```

(continues on next page)

(续上页)

```
        // 生成secp256k1验签对象
        ECDSASignature verifier = new ECDSASignature();
        // 计算data的哈希(keccak256)
        Keccak256 hasher = new Keccak256();
        String hashData = hasher.hash(data);
        // 验证签名
        return verifier.verify(ecdsaKeyPair.getHexPublicKey(), hashData,␣
↪signatureResult.convertToString());
    }
    // 入参为byte[]
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪ecdsaKeyPair, byte[] data)
    {
        // 生成secp256k1验签对象
        ECDSASignature verifier = new ECDSASignature();
        // 计算data的哈希
        Keccak256 hasher = new Keccak256();
        byte[] hashData = hasher.hash(data);
        // 验证签名
        return verifier.verify(ecdsaKeyPair.getHexPublicKey(), Hex.
↪toHexString(hashData), signatureResult.convertToString());
    }
```

类似地，国密签名/验签接口调用示例如下：

```
    // 生成sm2签名
    public SM2SignatureResult generateSignatureWithSM2(CryptoKeyPair sm2KeyPair,␣
↪String data)
    {
        // 生成sm2签名对象
        SM2Signature signer = new SM2Signature();
        // 计算data的哈希(sm3)
        SM3Hash hasher = new SM3Hash();
        String hashData = hasher.hash(data);
        return (SM2SignatureResult)signer.sign(hashData, sm2KeyPair);
    }

    // data类型为byte[]
    public SM2SignatureResult generateSignatureWithSecp256k1(CryptoKeyPair␣
↪sm2KeyPair, byte[] data)
    {
        // 生成sm2签名对象
        SM2Signature signer = new SM2Signature();
        // 计算data的哈希(sm3)
        SM3Hash hasher = new SM3Hash();
        byte[] hashData = hasher.hash(data);
        return (SM2SignatureResult)signer.sign(hashData, sm2KeyPair);
    }

    /// 验证sm2签名
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪sm2KeyPair, String data)
    {
        // 生成sm2验签对象
        SM2Signature verifier = new SM2Signature();
        // 计算data的哈希
        SM3Hash hasher = new SM3Hash();
        String hashData = hasher.hash(data);
        // 验证签名
        return verifier.verify(sm2KeyPair.getHexPublicKey(), hashData,␣
↪signatureResult.convertToString());
```

(continues on next page)

(续上页)

```
    }
    // 入参为byte[]
    public boolean verifySignature(SignatureResult signatureResult, CryptoKeyPair␣
↪sm2KeyPair, byte[] data)
    {
        // 生成sm2验签对象
        SM2Signature verifier = new SM2Signature();
        // 计算data的哈希
        SM3Hash hasher = new SM3Hash();
        byte[] hashData = hasher.hash(data);
        // 验证签名
        return verifier.verify(sm2KeyPair.getHexPublicKey(), Hex.
↪toHexString(hashData), signatureResult.convertToString());
    }
```

签名结果类型转换

java sdk提供了将签名结果SignatureResult转换为字符串，以及从字符串中构造签名对象SignatureResult的功能，示例如下:

```
    // 将签名结果转换为字符串
    public String covertSignatureResultToString(SignatureResult signatureResult)
    {
        return signatureResult.convertToString();
    }

    // 从字符串中构造非国密签名对象
    public ECDSASignatureResult covertStringToECDSASignatureResult(String␣
↪signatureString)
    {
        return new ECDSASignatureResult(signatureString);
    }

    // 从签名字符串中构造国密签名对象
    public SM2SignatureResult covertStringToECDSASignatureResult(CryptoKeyPair␣
↪smKeyPair, String signatureString)
    {
        return new SM2SignatureResult(smKeyPair.getHexPublicKey(),␣
↪signatureString);
    }
```

8.5 Web3SDK

标签: Web3SDK

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
- 支持通过国密SSL与节点通信

8.5.1 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。

8.5.2 Java应用引入SDK

通过gradle或maven引入SDK到java应用

gradle:

```
compile ('org.fisco-bcos:web3sdk:2.6.1')
```

maven:

```
<dependency>
    <groupId>org.fisco-bcos</groupId>
    <artifactId>web3sdk</artifactId>
    <version>2.6.1</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。

8.5.3 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
- FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错。

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
    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" />
```

(continues on next page)

(续上页)

```
                <!-- 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>

</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)
    - 使用国密SSL，在国密区块链环境基础上，SDK需要打开encryptType开关，然后配置国密SSL证书
- 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
    * `caCert`SL连接根证书路径
    * `sslCert`SDK证书路径
    * `sslKey`SDK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
    * `gmSslCert`SSL连接签名证书路径
    * `gmSslKey`SSL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
        - 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。

8.5.4 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();
    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");
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");
    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
```

(continues on next page)

(续上页)

```
    //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。

8.5.5 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。

8.5.6 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);
```

(continues on next page)

(续上页)

```
        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 =
↪"0x406d373b0000000000000000000000000000000000000000000000000000000000000001b207fffffffffffffffffffff
↪";
String jsonResult = txDecodeSampleDecoder.decodeInputReturnJson(input);
InputAndOutputResult objectResult = txDecodeSampleDecoder.
↪decodeInputReturnObject(input);
```

(continues on next page)

(续上页)

```
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"
    }
  ]
}

object =>
InputAndOutputResult[
  function=echo(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes),
  methodID=0x406d373b,
  result=[
    ResultEntity[
      name=_u,
```

(continues on next page)

(续上页)

```
      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)
String input =␣
↪"0x406d373b0000000000000000000000000000000000000000000000000001b207fffffffffffffffffff
↪

String output =
↪""0x0000000000000000000000000000000000000000000000000001b207fffffffffffffffffffffffff
↪";

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[
  function=echo(uint256,
  int256,
  bool,
  address,
  bytes32,
  string,
  bytes),
  methodID=0x406d373b,
  result=[
    ResultEntity[
      name=,
      type=uint256,
      data=111111
    ],
    ResultEntity[
      name=,
```

(continues on next page)

(续上页)

```
      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());

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
      },
```

(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": "sadfljkjkljkl",
        "indexed": false
      }
    ]
  ],
  "Event2(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",
```


(continues on next page)

(续上页)

```
        "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
      ],
      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=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[
        name=_s,
        type=string,
        data=FISCO-BCOS
      ],
      ResultEntity[
        name=_bs,
        type=bytes,
        data=nices
      ]
    ]
  ]
}
```

8.5.7 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
```

(continues on next page)

(续上页)

```
    -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)
```

(continues on next page)

(续上页)

```
    // 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<>();
```

(continues on next page)

(续上页)

```
    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>());

        // 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");
```

(continues on next page)

(续上页)

```
        // 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"
```

(continues on next page)

(续上页)

```
        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);
```

8.5.8 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。

8.6 Node.js SDK

标签：Node.JS SDK

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

8.6.1 快速安装

标签：安装Node.js 命令行工具

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]
```

8.6.2 配置说明

标签：Node.JS SDK 配置

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 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。

8.6.3 Node.js API

标签：java-sdk Client

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

8.7 Python SDK

标签：`Python API`

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

8.7.1 快速安装

标签：`Python API` 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
- FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错

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
```

8.7.2 配置说明

标签：`Python SDK 证书配置`

`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
    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
```

8.7.3 Python API

标签: Python API Client

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

8.7.4 控制台

标签: Python SDK PythonSDK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
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
        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
     >> 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
```

(continues on next page)

(续上页)

```
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
```

(continues on next page)

(续上页)

```
     >> 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
```

(continues on next page)

(续上页)

```
    >> 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
↪",
    "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",
```

(continues on next page)

(续上页)

```
    "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
↪"
        }
    ],
    "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":
↪"12ce3fc76bc3253ba9be25dc3adb8b75df392583b8f2813f4c623cff258980c8c2c73f384ce6f37dca7261ea0a9fb2
↪",
    "Topic": []
}
     >> {
    "Agency": "agency",
```

(continues on next page)

(续上页)

```
    "IPAndPort": "127.0.0.1:30302",
    "Node": "node2",
    "NodeID":
↪"6bd07f2f8180ac9d56b40ff9977ba528a4f65e83d4ca95a0537e12f6638f78848e0765cbee0cb2b5f581d7eb5027d1
↪",
    "Topic": []
}
     >> {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30303",
    "Node": "node3",
    "NodeID":
↪"3ad90ae5a10b8d88c9936492a37f564884e82b176e91f5e2e2f75a347be87212aac148ee7fa2060be8a790eaa3d44a
↪",
    "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']
```

(continues on next page)

(续上页)

```
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",
```

(continues on next page)

(续上页)

```
    "extraData": [
        "0x312d623837383336366653363303733613533326539636263343739373864
        ... 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",
```

(continues on next page)

(续上页)

```
    "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",
    "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",
```

(continues on next page)

(续上页)

```
    "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
```

8.7.5 开发样例

标签：`Python API` 快速安装

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
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)
```

(continues on next page)

(续上页)

```
#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")
print("call getall result:",res)
print("demo_tx,total req {}".format(client.request_counter))
client.finish()
```

8.8 Go SDK

标签：go-sdk Go SDK

Go SDK 提供了访问 FISCO BCOS 节点的Go API，支持节点状态查询、部署和调用合约等功能，基于Go SDK可快速开发区块链应用，目前支持 FISCO BCOS 2.2.0+

主要特性

- 提供调用FISCO BCOS JSON-RPC 的Go API
- 提供合约编译，将Solidity合约文件编译成abi和bin文件，然后再转换成Go合约文件
- 提供部署及调用go合约文件的GO API
- 提供调用预编译（Precompiled）合约的Go API
- 提供与FISCO BCOS节点通信的 Channel协议，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

8.8.1 环境和配置文件

标签：`go-sdk` `环境配置`

环境

- Go开发环境
  - Golang >= 1.13.6
  - 项目采用 go module 进行包管理，可参考 Using Go Modules
  - 如果您没有部署过Go环境，可参考 官方文档
- 基本开发组件
  - Git（Windows、Linux及MacOS需要）
  - Git bash（仅Windows需要）

配置文件

Go SDK 的配置文件为一个 TOML 文件，主要包括**网络配置**、**账户配置**以及**链配置**。配置文件 config.toml 示例如下：

```
[Network]
Type="channel"
CAFile="ca.crt"
Cert="sdk.crt"
Key="sdk.key"
[[Network.Connection]]
NodeURL="127.0.0.1:20200"
GroupID=1
# [[Network.Connection]]
# NodeURL="127.0.0.1:20200"
# GroupID=2

[Account]
KeyFile=".ci/0x83309d045a19c44dc3722d15a6abd472f95866ac.pem"

[Chain]
ChainID=1
SMCrypto=false
```

网络配置

网络配置主要用于设置 **网络连接模式**、**证书文件** 和待连接的 **节点信息**，支持设置多个节点。

- `Type`：是Go SDK与区块链节点建立连接的模式，支持channel和rpc两种方式；
  - `channel`：使用ssl协议建立连接，需要提供ca.crt、sdk.crt、sdk.key证书；
  - `rpc`：使用http协议建立连接，不需要提供证书；
- `CAfile`：CA根证书文件路径，用于验证待连接节点的合法性；
- `Cert`：SDK证书文件路径，用于待连接节点验证SDK的合法性；
- `Key`：SDK私钥文件路径，Cert证书对应的私钥，用于加解密和签名；
- `NodeURL`：待连接节点的URL地址，由IP和port两部分组成；
- `GroupID`：待连接节点所属的群组ID。

账户配置

- `KeyFile`：外部账户的私钥文件路径，目前只支持pem格式的私钥文件。国密和非国密账户脚本可从get_account.sh和get_gm_account.sh下载，使用方式可参考账户管理。此外，Go SDK代码也支持生成账号，参考这里了解更多。

链配置

- `ChainID`：待连接节点所属的链ID，可通过查看节点config.ini配置文件中chain.id配置项获得；
- `SMCrypto`：待连接节点所属链使用的签名算法，ture表示使用国密SM2，false表示使用普通ECDSA。

8.8.2 Go API

标签：`go-sdk` `AMOP`

Go SDK为区块链应用开发者提供了Go API接口，以服务的形式供外部调用。按照功能，Go API可以分为如下几类：

- **client**：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 基于表的权限控制、CNS、系统属性配置、节点类型配置、用户表 CRUD、基于角色的权限控制、合约生命周期权限控制等功能。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；
  - **ChainGovernanceService**：提供基于角色的权限控制支持；
  - **ContractLifeCycleService**：提供合约生命周期权限控制支持。

Client

位置：go-sdk/client/go_client.go

功能：访问FISCO BCOS 2.0+节点JSON-RPC

注解:

- 如果用户试图尝试使用一个 sdk 连接多个群组，可以利用 APIHandler 中暴露的接口，详细内容可阅读源码 go-sdk

---

PermissionService

位置：go-sdk/precompiled/permission

功能：提供对基于表的权限控制支持

CNSService

位置：go-sdk/precompiled/cns

功能：提供对CNS的支持

SystemConfigService

位置：go-sdk/precompiled/config

功能：提供修改系统配置项的功能，目前支持的配置项有tx_count_limit、tx_gas_limit、rpbft_epoch_sealer_num 和 rpbft_epoch_block_num

ConsensusService

位置：go-sdk/precompiled/consensus

功能：提供对节点类型配置的支持

CRUDService

位置：go-sdk/precompiled/crud

功能：提供对CRUD(增删改查)操作的支持

ChainGovernanceService

位置：go-sdk/precompiled/chaingovernance

功能：提供对基于角色的权限控制支持

ContractLifeCycleService

位置：go-sdk/precompiled/contractlifecycle

功能：提供对合约生命周期操作的支持

8.8.3 控制台

标签：go-sdk Go SDK控制台

Go SDK 通过 console.go 实现了一个简单的控制台，支持区块链和节点信息查询。

使用

- 拉取代码并编译

```
# 拉取代码
git clone https://github.com/FISCO-BCOS/go-sdk.git
# 切换目录
cd go-sdk
# 编译 cmd/console.go
go build cmd/console.go
```

- 搭建FISCO BCOS 2.2以上版本节点，请 参考
- config.toml 配置文件默认使用 channel 连接模式，请拷贝对应的 ca.crt、sdk.crt 和 sdk.key 证书至 go-sdk 目录
- FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错

getBlockByHash

根据区块哈希获取区块信息:

```
./console getBlockByHash [blockHash] [true/false]
```

参数包括:

- blockHash：区块 hash 值;
- true/false：true 会返回区块中所有交易的详细内容，false 只会返回区块中所有交易的 hash 值，默认为 true。

```
> ./console getBlockByHash␣
↪0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848 true

Block:
{
  "extraData": [],
  "gasLimit": "0x0",
  "gasUsed": "0x0",
  "hash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
  "logsBloom":
↪"0x00000000000
```

(续上页)

```
↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",

↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",

↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪"
  ],
  "stateRoot": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7
↪",
  "timestamp": "0x172fe19b855",
  "transactions": [
    {
      "blockHash":
↪"0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
      "blockNumber": "0x3",
      "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
      "gas": "0x11e1a300",
      "gasPrice": "0x11e1a300",
      "hash": "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
      "input":
↪"0x2800efc0000000000
```

(续上页)

```
↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪"
  ],
  "stateRoot": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7
↪",
  "timestamp": "0x172fe19b855",
  "transactions": [
    "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434"
  ],
  "transactionsRoot":
↪"0xcf057dc481d7a97700e93a1ea65f331c3cfee2fee80e3bb80c30748e4988fe9d"
}
```

getBlockByNumber

根据区块高度获取区块信息:

```
./console getBlockByNumber [blockNumber] [true/false]
```

参数包括:

- blockNumber：区块高度;
- true/false：true 会返回区块中所有交易的详细内容，false 只会返回区块中所有交易的 hash 值，默认为 true。

```
> ./console getBlockByNumber 3 true

Block:
{
  "dbHash": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7",
  "extraData": [],
  "gasLimit": "0x0",
  "gasUsed": "0x0",
  "hash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
  "logsBloom":
↪"0x00000000000
```

(续上页)

```
  "timestamp": "0x172fe19b855",
  "transactions": [
    {
      "blockHash":
↪"0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
      "blockNumber": "0x3",
      "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
      "gas": "0x11e1a300",
      "gasPrice": "0x11e1a300",
      "hash": "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
      "input":
↪"0x2800efc0000000000000000000000000000000000000000000000000000000000000000200000000000000000000000
↪",
      "nonce": "0x359fbc4677e4f4ca87a96a31372b1194f03ba200db94a18ad0b30f2e858ac32",
      "to": "0x0000000000000000000000000000000000001003",
      "transactionIndex": "0x0",
      "value": "0x0"
    }
  ],
  "transactionsRoot":
↪"0xcf057dc481d7a97700e93a1ea65f331c3cfee2fee80e3bb80c30748e4988fe9d"
}

> ./console getBlockByNumber 3 false

Block:
{
  "dbHash": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7",
  "extraData": [],
  "gasLimit": "0x0",
  "gasUsed": "0x0",
  "hash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
  "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
  "number": "0x3",
  "parentHash": "0x57016509418eb81c0353b1252a364383fcfc5c71035c8a01d24e785ac6e2ce4a
↪",
  "receiptsRoot":
↪"0x4e430ca6474d7013a819a7c602497b7bfdfa14a1197a1edc35444b756cf7e6fc",
  "sealer": "0x1",
  "sealerList": [

↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
↪",

↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",

↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",

↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪"
  ],
  "stateRoot": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7
↪",
  "timestamp": "0x172fe19b855",
  "transactions": [
    "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434"
  ],
```

(continues on next page)

(续上页)

```
  "transactionsRoot":
↪"0xcf057dc481d7a97700e93a1ea65f331c3cfee2fee80e3bb80c30748e4988fe9d"
}
```

getBlockHashByNumber

根据区块高度获取区块哈希：

```
./console getBlockHashByNumber [blockNumber]
```

参数包括：

- blockNumber：区块高度。

```
> ./console getBlockHashByNumber 3

Block Hash:
"0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848"
```

getBlockNumber

获取最新块高：

```
> ./console getBlockNumber

blocknumber:
    hex: "0x3"
decimal:  3
```

getClientVersion

获取区块链节点版本信息：

```
> ./console getClientVersion

Client Version:
{
  "Build Time": "20200610 15:42:05",
  "Build Type": "Linux/g++/RelWithDebInfo",
  "Chain Id": "1",
  "FISCO-BCOS Version": "2.5.0",
  "Git Branch": "master",
  "Git Commit Hash": "b0978f773ca1dbb499a4343b9fb3a12c40b8fc97",
  "Supported Version": "2.4.0"
}
```

getCode

根据合约地址查询合约数据：

```
./console getCode [contract address]
```

参数包括：

- contract address：合约地址。

```
> ./console getCode 0x65474dbd4f08170bc2dc30f9ae32f8e2206b15a6

Contract Code:

↪"0x6080604052600436106100
↪"
```

getConsensusStatus

获取区块链节点共识状态

```
> ./console getConsensusStatus

Consensus Status:
[
  {
    "accountType": 1,
    "allowFutureBlocks": true,
    "cfgErr": false,
    "connectedNodes": 3,
    "consensusedBlockNumber": 4,
    "currentView": 80,
    "groupId": 1,
    "highestblockHash":
↪"0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
    "highestblockNumber": 3,
    "leaderFailed": false,
    "max_faulty_leader": 0,
    "nodeId":
↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪",
    "nodeNum": 3,
    "node_index": 2,
    "omitEmptyBlock": true,
    "protocolId": 65544,
    "sealer.0":
↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",
    "sealer.1":
↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",
    "sealer.2":
↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪",
    "toView": 80
  },
  [
    {
      "nodeId":
↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",
      "view": 78
    },
    {
      "nodeId":
↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",
      "view": 79
    },
    {
```

(continues on next page)

(续上页)

```
      "nodeId":
↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪",
      "view": 80
    }
  ]
]
```

getGroupID

获取配置文件中指定的连接节点群组ID:

```
> ./console getGroupID

Group ID:
1
```

getGroupList

获取节点所属群组的群组ID列表:

```
> ./console getGroupList

Group ID List:
[
  1
]
```

getGroupPeers

获取指定群组的共识节点和观察节点列表:

```
> ./console getGroupPeers

Peers:
[

↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
↪",

↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",

↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",

↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪"
]
```

getNodeIDList

获取节点及其连接节点的列表:

```
> ./console getNodeIDList

Node ID list:
[

 ↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
 ↪",

 ↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
 ↪",

 ↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
 ↪",

 ↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
 ↪"
]
```

getObserverList

获取观察者节点列表：

```
> ./console getObserverList

Observer List:
[

 ↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
 ↪"
]
```

getPbftView

获取PBFT视图：

```
> ./console getPbftView

PBFT view:
"0x30"
```

getPeers

获取区块链节点的连接信息：

```
> ./console getPeers

Peers:
[
  {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:51808",
    "Node": "node3",
    "NodeID":
 ↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
 ↪",
    "Topic": []
  },
```

(continues on next page)

(续上页)

```
  {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30302",
    "Node": "node2",
    "NodeID":
↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
↪",
    "Topic": []
  },
  {
    "Agency": "agency",
    "IPAndPort": "127.0.0.1:30301",
    "Node": "node1",
    "NodeID":
↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",
    "Topic": []
  }
]
```

getPendingTransactions

获取交易池内所有未上链的交易：

```
> ./console getPendingTransactions

Pending Transactions:
[]
```

getPendingTxSize

获取交易池内未上链的交易数目：

```
> ./console getPendingTxSize

Pending Transactions Count:
    hex: "0x0"
decimal:  0
```

getSealerList

获取共识节点列表：

```
>  ./console getSealerList

Sealer List:
[

↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪",

↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪",

↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪"
]
```

getSyncStatus

获取区块链节点同步状态:

```
> ./console getSyncStatus

Synchronization Status:
{
  "blockNumber": 3,
  "genesisHash": "65e9c13da61b1f47564ccc6498260d739ce3dfd3366accc539a64412c1ef8e88
↪",
  "isSyncing": false,
  "knownHighestNumber": 3,
  "knownLatestHash":
↪"ce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
  "latestHash": "ce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
  "nodeId":
↪"955ab783c6adc7a5f817773a5fbe32ecd9310f9392406eb1f9fa3d2b21539577a70c933158c1bd3a0bf183d5498bf5
↪",
  "peers": [
    {
      "blockNumber": 3,
      "genesisHash":
↪"65e9c13da61b1f47564ccc6498260d739ce3dfd3366accc539a64412c1ef8e88",
      "latestHash":
↪"ce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
      "nodeId":
↪"42ae2ae3950a2933b6f576dc946b90b242b7f9a2a8d45aae53c1a1664cd582bd759fa69e4b52f7a453dce702878b9e
↪"
    },
    {
      "blockNumber": 3,
      "genesisHash":
↪"65e9c13da61b1f47564ccc6498260d739ce3dfd3366accc539a64412c1ef8e88",
      "latestHash":
↪"ce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
      "nodeId":
↪"8b5e90815966004e807803aba5f003bc271d0b0aa82805c85764b21187bd504f79ec46eaf1e60752956af174a927d7
↪"
    },
    {
      "blockNumber": 3,
      "genesisHash":
↪"65e9c13da61b1f47564ccc6498260d739ce3dfd3366accc539a64412c1ef8e88",
      "latestHash":
↪"ce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
      "nodeId":
↪"95381c1d22d10ad73171f0d34ec1f2e5809f47ee76264aeed4bb0daaf594bfb5da89f6a65ff4a056952f66a6a99fc9
↪"
    }
  ],
  "protocolId": 65545,
  "txPoolSize": "0"
}
```

getSystemConfigByKey

根据关键字获取区块链系统配置:

```
./console getSystemConfigByKey [tx_count_limit/tx_gas_limit]
```

参数包括：

- tx_count_limit/tx_gas_limit：单个区块中交易数量限制/单笔交易中可消耗的 gas 限制

```
> ./console getSystemConfigByKey tx_count_limit
Result:
"1000"

> ./console getSystemConfigByKey tx_gas_limit
Result:
"300000000"
```

getTotalTransactionCount

获取指定群组的上链交易数目：

```
> ./console getTotalTransactionCount

Latest Statistics on Transaction and Block Height:
{
  "blockNumber": "0x5",
  "failedTxSum": "0x0",
  "txSum": "0x5"
}
```

getTransactionByBlockHashAndIndex

根据交易所属的区块哈希、交易索引获取交易信息：

```
./console getTransactionByBlockHashAndIndex [blockHash] [transactionIndex]
```

参数包括：

- blockHash：区块 hash 值；
- transactionIndex：交易索引值，注意需要转换为16进制。

```
> ./console getTransactionByBlockHashAndIndex␣
↪0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848 0x0

Transaction:
{
  "blockHash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848
↪",
  "blockNumber": "0x3",
  "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
  "gas": "0x11e1a300",
  "gasPrice": "0x11e1a300",
  "hash": "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
  "input":
↪"0x2800efc0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000
↪",
  "nonce": "0x359fbc4677e4f4ca87a96a31372b1194f03ba200db94a18ad0b30f2e858ac32",
  "to": "0x0000000000000000000000000000000000001003",
  "transactionIndex": "0x0",
  "value": "0x0"
}
```

getTransactionByBlockNumberAndIndex

根据交易所属的区块高度、交易索引获取交易信息：

```
./console getTransactionByBlockNumberAndIndex [blockNumber] [transactionIndex]
```

参数包括：

- blockNumber：区块高度；
- transactionIndex：交易索引值，注意需要转换为16进制。

```
> ./console getTransactionByBlockNumberAndIndex 3 0x0

Transaction:
{
  "blockHash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848
↪",
  "blockNumber": "0x3",
  "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
  "gas": "0x11e1a300",
  "gasPrice": "0x11e1a300",
  "hash": "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
  "input":
↪"0x2800efc0000000000000000000000000000000000000000000000000000000000000000200000000000000000000000
↪",
  "nonce": "0x359fbc4677e4f4ca87a96a31372b1194f03ba200db94a18ad0b30f2e858ac32",
  "to": "0x0000000000000000000000000000000000001003",
  "transactionIndex": "0x0",
  "value": "0x0"
}
```

getTransactionByHash

根据交易哈希获取交易信息：

```
./console getTransactionByHash [transactionHash]
```

参数包括：

- transactionHash：交易 hash 值。

```
> ./console getTransactionByHash␣
↪0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434

Transaction:
{
  "blockHash": "0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848
↪",
  "blockNumber": "0x3",
  "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
  "gas": "0x11e1a300",
  "gasPrice": "0x11e1a300",
  "hash": "0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
  "input":
↪"0x2800efc0000000000000000000000000000000000000000000000000000000000000000200000000000000000000000
↪",
  "nonce": "0x359fbc4677e4f4ca87a96a31372b1194f03ba200db94a18ad0b30f2e858ac32",
  "to": "0x0000000000000000000000000000000000001003",
  "transactionIndex": "0x0",
  "value": "0x0"
}
```

getTransactionReceipt

根据交易哈希获取交易回执:

```
./console getTransactionReceipt [transactionHash]
```

参数包括:

- transactionHash：交易 hash 值。

```
> ./console getTransactionReceipt␣
↪0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434

Transaction Receipt:
{
        "transactionHash":
↪"0x5518df7c2063efeb6481c35c4c58f378fac5f476c023c2019b9b01d221478434",
        "transactionIndex": "0x0",
        "blockHash":
↪"0xce28a18b54ee72450c403968f705253a59c87a22801a88cc642ae800bb8b4848",
        "blockNumber": "0x3",
        "gasUsed": "0x765b",
        "contractAddress": "0x0000000000000000000000000000000000000000",
        "root": "0x2d23b1248fe53a1769db06af5c0e99261678643f405f058cfa89193592d13fa7
↪",
        "status": "0x0",
        "from": "0x4ca29e9e8cb79c863c04f83827ab540315f25e67",
        "to": "0x0000000000000000000000000000000000001003",
        "input":
↪"0x2800efc0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000
↪",
        "output":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
        "logs": [],
        "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

grantUserTableManager

根据用户表名和外部账户地址设置权限信息

```
> ./console grantUserTableManager [tableName] [accountAddress]
```

参数包括：

- tableName：表名
- accountAddress：外部账户地址

```
> ./console grantUserTableManager t_test 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064d
success
```

revokeUserTableManager

根据用户表名和外部账户地址撤销权限信息

```
> ./console revokeUserTableManager [tableName] [accountAddress]
```

参数包括：

- tableName：表名
- accountAddress：外部账户地址

```
> ./console revokeUserTableManager t_test␣
↪0xFbb18d54e9Ee57529cda8c7c52242EFE879f064d
success
```

listUserTableManager

根据用户表名查询设置的权限记录列表

```
./console listUserTableManager [tableName]
```

参数包括：

- tableName：表名

```
> ./console listUserTableManager t_test
{"user table managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064d",
↪"enable_num":"11","table_name":"u_t_test"}]}
```

grantContractWritePermission

根据合约地址和外部账户地址设置合约写权限信息

```
> ./console grantContractWritePermission [contractAddress] [accountAddress]
```

参数包括：

- contractAddress：合约地址
- accountAddress：外部账户地址

```
> ./console grantContractWritePermission␣
↪0x0a68F060B46e0d8f969383D260c34105EA13a9dd␣
↪0xFbb18d54e9Ee57579cda8c7c52242EFE879f064a
success
```

revokeContractWritePermission

根据合约地址和外部账户地址去除合约写权限信息

```
> ./console revokeContractWritePermission [contractAddress] [accountAddress]
```

参数包括：

- contractAddress：合约地址
- accountAddress：外部账户地址

```
> ./console revokeContractWritePermission␣
↪0x0a68F060B46e0d8f969383D260c34105EA13a9dd␣
↪0xFbb18d54e9Ee57579cda8c7c52242EFE879f064a
success
```

listSysConfigManager

根据合约地址查询拥有合约写权限的记录列表

```
./console listSysConfigManager [contractAddress]
```

参数包括：

- contractAddress：合约地址

```
> ./console listContractWritePermission 0x0a68F060B46e0d8f969383D260c34105EA13a9dd
{"managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064f","enable_num":
↪"32","table_name":"c_0a68f060b46e0d8f969383d260c34105ea13a9dd"},{"address":
↪"0xfbb18d54e9ee57579cda8c7c52242efe879f064a","enable_num":"38","table_name":"c_
↪0a68f060b46e0d8f969383d260c34105ea13a9dd"}]}
```

grantDeployAndCreateManager

增加外部账户地址部署合约和创建用户表的权限

```
./console grantDeployAndCreateManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console grantDeployAndCreateManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

revokeDeployAndCreateManager

移除外部账户地址部署合约和创建用户表的权限

```
./console revokeDeployAndCreateManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console revokeDeployAndCreateManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

listDeployAndCreateManager

查询拥有部署合约和创建用户表权限的记录列表

```
> ./console listDeployAndCreateManager
{"managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064f","enable_num":
↪"15","table_name":"_sys_tables_"}]}
```

grantPermissionManager

授予外部账户地址链管理员权限，链管理员可以使用权限分配功能。该命令只支持 FISCO BCOS 2.4.0 和以下版本。

```
./console grantPermissionManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console grantPermissionManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

revokePermissionManager

撤销链外部账户地址链管理员权限

```
./console revokePermissionManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console revokePermissionManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

listPermissionManager

查询拥有链管理权限的记录列表

```
> ./console listPermissionManager
{"managers":[{"address":"0x83309d045a19c44dc3722d15a6abd472f95866ac","enable_num":
↪"24","table_name":"_sys_table_access_"}]}
```

grantNodeManager

增加外部账户地址的节点管理权限

```
./console grantNodeManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console grantNodeManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

revokeNodeManager

移除外部账户地址的节点管理权限

```
./console revokeNodeManager [accountAddress]
```

参数包括:

- accountAddress：外部账户地址

```
> ./console revokeNodeManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

listNodeManager

查询拥有节点管理的权限记录列表

```
> ./console listNodeManager
{"managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064f","enable_num":
↪"18","table_name":"_sys_consensus_"}]}
```

grantCNSManager

增加外部账户地址使用CNS的权限

```
./console grantCNSManager [accountAddress]
```

参数包括:

- accountAddress：外部账户地址

```
> ./console grantCNSManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

revokeCNSManager

移除外部账户地址使用CNS的权限

```
./console revokeCNSManager [accountAddress]
```

参数包括:

- accountAddress：外部账户地址

```
> ./console revokeCNSManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

listCNSManager

查询拥有使用CNS权限的记录列表

```
> ./console listCNSManager
{"managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064f","enable_num":
↪"19","table_name":"_sys_cns_"}]}
```

grantSysConfigManager

增加外部账户地址的系统参数管理权限

```
./console grantSysConfigManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console grantSysConfigManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

revokeSysConfigManager

移除外部账户地址的系统参数管理权限

```
./console revokeSysConfigManager [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console revokeSysConfigManager 0xFbb18d54e9Ee57529cda8c7c52242EFE879f064F
success
```

listSysConfigManager

查询拥有系统参数管理的权限记录列表

```
> ./console listSysConfigManager
{"managers":[{"address":"0xfbb18d54e9ee57529cda8c7c52242efe879f064f","enable_num":
↪"21","table_name":"_sys_config_"}]}
```

queryCNS

根据合约名和版本号查询CNS信息

```
./console queryCNS [name] [version]
```

参数包括：

- name：合约名（必须）
- version：版本号（可选）

```
> ./console queryCNS store 5.0
name: store, version: 5.0, address: 0x0626918C51A1F36c7ad4354BB1197460A533a2B9

> ./console queryCNS store
name: store, version: 5.0, address: 0x0626918C51A1F36c7ad4354BB1197460A533a2B9
```

getAddressByContractNameAndVersion

根据合约名和合约版本号查询合约地址

```
./console getAddressByContractNameAndVersion [name] [version]
```

参数包括：

- name：合约名
- version：版本号

```
> ./console getAddressByContractNameAndVersion store 5.0
0626918c51a1f36c7ad4354bb1197460a533a2b9
```

addObserver

根据节点NodeID设置对应节点为观察者节点

```
./console addObserver [Node ID]
```

参数包括：

- Node ID：节点 ID。

```
> ./console addObserver␣
↪58108297d9b545dc6e9a7ee4fea539c7886ced0c4cfeb33acd16ad23158247901d7d45dfbacc2fe97e38afaf163a4608
success
```

addSealer

根据节点NodeID设置对应节点为共识节点

```
./console addSealer [Node ID]
```

参数包括：

- Node ID：节点 ID。

```
> ./console addSealer␣
↪58108297d9b545dc6e9a7ee4fea539c7886ced0c4cfeb33acd16ad23158247901d7d45dfbacc2fe97e38afaf163a4608
success
```

removeNode

根据节点NodeID设置对应节点为游离节点

```
./console removeNode [Node ID]
```

参数包括：

- Node ID：节点 ID。

```
> ./console removeNode␣
↪58108297d9b545dc6e9a7ee4fea539c7886ced0c4cfeb33acd16ad23158247901d7d45dfbacc2fe97e38afaf163a4608
success
```

grantCommitteeMember

根据外部账户地址新增委员会成员

```
./console grantCommitteeMember [accountAddress]
```

参数包括：

- accountAddress：外部账户地址。

```
> ./console grantCommitteeMember 0x83309d045a19c44dc3722d15a6abd472f95866ac
success
```

revokeCommitteeMember

根据外部账户地址撤销委员会成员

```
./console revokeCommitteeMember [accountAddress]
```

参数包括：

- accountAddress：外部账户地址。

```
> ./console revokeCommitteeMember 0x83309d045a19c44dc3722d15a6abd472f95866ac
success
```

listCommitteeMembers

查询委员会成员列表

```
> ./console listCommitteeMembers
{"committee_members":[{"address":"0x83309d045a19c44dc3722d15a6abd472f95866ac",
↪"enable_num":"24"}]}
```

queryCommitteeMemberWeight

根据外部账户地址查询委员会成员投票权值

```
./console queryCommitteeMemberWeight [accountAddress]
```

参数包括：

- accountAddress：外部账户地址。

```
> ./console queryCommitteeMemberWeight 0x83309d045a19c44dc3722d15a6abd472f95866ac
success, the weight 0x83309d045a19c44dc3722d15a6abd472f95866ac is 1
```

updateCommitteeMemberWeight

根据外部账户地址更新委员会成员投票权值

```
./console updateCommitteeMemberWeight [accountAddress] [weight]
```

参数包括：

- accountAddress：外部账户地址
- weight：权值，大于 0 的整数

```
> ./console updateCommitteeMemberWeight 0x83309d045a19c44dc3722d15a6abd472f95866ac␣
↪2
success
```

queryThreshold

查询委员会全体委员投票时总票数生效的阈值

```
> ./console queryThreshold
success, the effective threshold of voting is 50
```

updateThreshold

更新委员会全体委员投票时票数占比生效的阈值

```
./console updateThreshold [threshold]
```

参数包括：

- threshold：阈值，支持范围 [0, 99)。

```
> ./console updateThreshold 60
success
```

grantOperator

根据外部账户地址新增运维权限

```
./console grantOperator [accountAddress]
```

参数包括：

- accountAddress：外部账户地址。

```
> ./console grantOperator 0x112fb844934c794a9e425dd6b4e57eff1b519f17
success
```

revokeOperator

根据外部账户地址撤销运维权限

```
./console revokeOperator [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console revokeOperator 0x112fb844934c794a9e425dd6b4e57eff1b519f17
success
```

listOperators

查询全体运维成员列表

```
> ./console listOperators
{"operators":[{"address":"0x112fb844934c794a9e425dd6b4e57eff1b519f17","enable_num":
↪"32"}]}
```

freezeAccount

根据外部账户地址冻结账户，该外部账号需要是部署过合约的账号

```
./console freezeAccount [accountAddress]
```

参数包括：

- accountAddress：外部账户地址。

```
> ./console freezeAccount 0x112fb844934c794a9e425dd6b4e57eff1b519f17
success
```

unfreezeAccount

根据外部账户地址解冻合账户

```
./console unfreezeAccount [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console unfreezeAccount 0x112fb844934c794a9e425dd6b4e57eff1b519f17
success
```

getAccountStatus

根据外部账户地址查询账户状态

```
./console getAccountStatus [accountAddress]
```

参数包括：

- accountAddress：外部账户地址

```
> ./console getAccountStatus 0x112fb844934c794a9e425dd6b4e57eff1b519f17
The account has been frozen. You can use this account after unfreezing it.
```

freezeContract

根据合约地址冻结合约

```
./console freezeContract [contract address]
```

参数包括：

- contract address：合约地址

```
> ./console freezeContract 0x54Fb7aAAF3D2d6663E3472d641b7fB54cB246Ff0
success
```

unfreezeContract

根据合约地址解冻合约

```
./console unfreezeContract [contract address]
```

参数包括：

- contract address：合约地址

```
> ./console unfreezeContract 0x54Fb7aAAF3D2d6663E3472d641b7fB54cB246Ff0
success
```

grantContractStatusManager

根据合约地址和外部账户地址授予账户合约管理权限

```
./console grantContractStatusManager [contract address] [accountAddress]
```

参数包括：

- contract address：合约地址
- accountAddress：外部账户地址

```
> ./console grantContractStatusManager 0x54Fb7aAAF3D2d6663E3472d641b7fB54cB246Ff0␣
↪0xae66fbe9ee2b5007e245d98bf7cf9904cc61e394
success
```

getContractStatus

根据合约地址查询合约状态

```
./console getContractStatus [contract address]
```

参数包括：

- contract address：合约地址

```
> ./console getContractStatus 0x54Fb7aAAF3D2d6663E3472d641b7fB54cB246Ff0
The contract is available.
```

ListManager

查询拥有合约地址管理权限的外部账号

```
./console ListManager [contract address]
```

参数包括：

- contract address：合约地址

```
> ./console ListManager 0x54Fb7aAAF3D2d6663E3472d641b7fB54cB246Ff0
{"managers":["0x112fb844934c794a9e425dd6b4e57eff1b519f17",
↪"0x83309d045a19c44dc3722d15a6abd472f95866ac",
↪"0xae66fbe9ee2b5007e245d98bf7cf9904cc61e394"]}
```

8.8.4 合约开发样例

标签：go-sdk 合约开发

非国密样例

本开发样例使用标准单群组四节点区块链网络结构，搭建请参考：安装。

在利用SDK进行项目开发时，对智能合约进行操作需要利用go-sdk的abigen工具将Solidity智能合约转换为Go文件代码。整体上主要包含六个流程：

- 准备需要编译的智能合约
- 配置好相应版本的solc编译器
- 构建go-sdk的合约编译工具abigen
- 编译生成go文件
- 准备建立ssl连接需要的证书
- 使用生成的go文件进行合约部署、调用

HelloWorld样例

准备HelloWorld.sol合约文件

```
# 该指令在go-sdk目录中执行
mkdir helloworld && cd helloworld
```

在 go-sdk 主目录中新建 helloworld 文件夹，在该文件夹中创建 HelloWorld.sol 合约。该合约提供两个接口，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 <0.6.11;

contract HelloWorld {
    string value;

    constructor() public {
        value = "Hello, World!";
    }

    function get() public view returns (string memory) {
        return value;
    }

    function set(string v) public {
        value = v;
    }
}
```

安装solc编译器

该编译器用于将 sol 合约文件编译成 abi 和 bin 文件，目前FISCO BCOS提供的solc编译器有0.4.25/0.5.2/0.6.10，每个版本有国密和非国密两种。

```
# 该指令在helloworld文件夹中执行
bash ../tools/download_solc.sh -v 0.4.25
```

构建go-sdk的代码生成工具abigen

该工具用于将 abi 和 bin 文件转换为 go 文件

```
# 该指令在helloworld文件夹中执行，编译生成abigen工具
go build ../cmd/abigen
```

编译生成go文件

先利用solc编译合约文件HelloWorld.sol，生成abi和bin文件

```
# 该指令在helloworld文件夹中执行
./solc-0.4.25 --bin --abi -o ./ ./HelloWorld.sol
```

helloworld目录下会生成HelloWorld.bin和HelloWorld.abi。此时利用abigen工具将HelloWorld.bin和HelloWorld.abi转换成HelloWorld.go：

```
# 该指令在helloworld文件夹中执行
./abigen --bin ./HelloWorld.bin --abi ./HelloWorld.abi --pkg helloworld --type␣
↪HelloWorld --out ./HelloWorld.go
```

最后helloworld文件夹下面存在以下6个文件：

```
HelloWorld.abi、HelloWorld.bin、HelloWorld.go、HelloWorld.sol、solc-0.4.25、abigen
```

准备建立ssl连接需要的证书

使用build_chain.sh脚本搭建区块链时会在./nodes/127.0.0.1/sdk文件夹中生成sdk证书、私钥以及ca证书，需要将这三个文件拷贝至`config.toml`中配置的位置。

部署合约

在helloworld文件夹中创建cmd文件夹，在cmd文件夹中创建main.go文件，main.go的内容如下，在该文件中调用HelloWorld.go部署智能合约

```
package main

import (
    "fmt"
    "log"

    "github.com/FISCO-BCOS/go-sdk/client"
    "github.com/FISCO-BCOS/go-sdk/conf"
    "github.com/FISCO-BCOS/go-sdk/helloworld" // import helloworld
)

func main(){
    configs, err := conf.ParseConfigFile("config.toml")
    if err != nil {
        log.Fatal(err)
    }
    config := &configs[0]

    client, err := client.Dial(config)
    if err != nil {
        log.Fatal(err)
    }
    address, tx, instance, err := helloworld.DeployHelloWorld(client.
↪GetTransactOpts(), client) // deploy contract
    if err != nil {
```

(continues on next page)

(续上页)

```
        log.Fatal(err)
    }
    fmt.Println("contract address: ", address.Hex())  // the address should be␣
↪saved
    fmt.Println("transaction hash: ", tx.Hash().Hex())
    _ = instance
}
```

构建并执行。

```
# 该指令在go-sdk目录中执行
go run helloworld/cmd/main.go
```

注解:

- 合约地址需要手动保存，调用合约接口时使用

调用合约get/set接口

在contract文件夹中创建helloworld_get.go文件，调用合约get接口，获取智能合约中name变量存储的值

```
package main

import (
    "fmt"
    "log"

    "github.com/FISCO-BCOS/go-sdk/client"
    "github.com/FISCO-BCOS/go-sdk/conf"
    "github.com/FISCO-BCOS/go-sdk/helloworld"
    "github.com/ethereum/go-ethereum/common"
)

func main() {
    configs, err := conf.ParseConfigFile("config.toml")
    if err != nil {
        log.Fatal(err)
    }
    config := &configs[0]
    client, err := client.Dial(config)
    if err != nil {
        log.Fatal(err)
    }

    // load the contract
    contractAddress := common.HexToAddress("contract address in hex") //␣
↪0x481D3A1dcD72cD618Ea768b3FbF69D78B46995b0
    instance, err := helloworld.NewHelloWorld(contractAddress, client)
    if err != nil {
        log.Fatal(err)
    }

    helloworldSession := &helloworld.HelloWorldSession{Contract: instance,␣
↪CallOpts: *client.GetCallOpts(), TransactOpts: *client.GetTransactOpts()}

    value, err := helloworldSession.Get()    // call Get API
    if err != nil {
        log.Fatal(err)
```

(continues on next page)

(续上页)

```
    }
    fmt.Println("value :", value)

    value = "Hello, FISCO BCOS"
    tx, receipt, err := helloworldSession.Set(value)  // call set API
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("tx sent: %s\n", tx.Hash().Hex())
    fmt.Printf("transaction hash of receipt: %s\n", receipt.GetTransactionHash())
}
```

KVTableTest样例

准备Table.sol合约文件

在 go-sdk 主目录中新建 kvtabletest 文件夹，拷贝 Table.sol 合约。

```
# 创建 kvtabletest 文件夹，该指令在go-sdk目录中执行
mkdir kvtabletest && cd kvtabletest

# 拷贝 Table.sol KVTableTest 合约
cp ../.ci/Table/Table.sol ../.ci/Table/KVTableTest.sol ./
```

准备KVTableTest.sol合约文件

该合约调用 Table 合约，实现创建用户表 t_kvtest，并对 t_kvtest 表进行读写。

```
pragma solidity>=0.4.24 <0.6.11;

import "./Table.sol";

contract KVTableTest {
    event SetResult(int256 count);

    KVTableFactory tableFactory;
    string constant TABLE_NAME = "t_kvtest";

    constructor() public {
        //The fixed address is 0x1010 for KVTableFactory
        tableFactory = KVTableFactory(0x1010);
        // the parameters of createTable are tableName,keyField,"vlaueFiled1,
↪vlaueFiled2,vlaueFiled3,..."
        tableFactory.createTable(TABLE_NAME, "id", "item_price,item_name");
    }

    //get record
    function get(string memory id) public view returns (bool, int256, string␣
↪memory) {
        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) {
```

(continues on next page)

(续上页)

```
            item_price = entry.getInt("item_price");
            item_name = entry.getString("item_name");
        }
        return (ok, item_price, item_name);
    }

    //set record
    function set(string memory id, int256 item_price, string memory 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;
    }
}
```

准备环境与合约编译

下面的操作都在 kvtabletest 文件夹中执行

```
bash ../tools/download_solc.sh -v 0.5.2

./solc-0.5.2 --bin --abi -o ./ ./KVTableTest.sol
```

编译生成 go 文件

先利用 solc 编译合约文件 KVTableTest.sol，生成 abi 和 bin 文件

```
go run ../cmd/abigen --bin ./KVTableTest.bin --abi ./KVTableTest.abi --pkg␣
↪kvtabletest --type KVTableTest --out ./KVTableTest.go
```

最后 kvtabletest 文件夹下面存在以下5个文件和其它若干文件：

```
KVTableTest.abi、KVTableTest.bin、KVTableTest.go、KVTableTest.sol、solc-0.5.2
```

部署合约

在 kvtabletest 文件夹中创建 cmd 文件夹，在 cmd 文件夹中创建 kvtabletest_main.go 文件，调用 KVTableTest.go 部署智能合约。合约将创建 t_kvtest 表，该表用于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。使用build_chain.sh脚本搭建区块链时会在./nodes/127.0.0.1/sdk文件夹中生成sdk证书、私钥以及ca证书，需要将这三个文件拷贝至`config.toml`中配置的位置。

```
package main

import (
    "fmt"
```

(continues on next page)

(续上页)

```
    "log"

    "github.com/FISCO-BCOS/go-sdk/client"
    "github.com/FISCO-BCOS/go-sdk/conf"
    kvtable "github.com/FISCO-BCOS/go-sdk/kvtabletest" // import kvtabletest
)

func main(){
    configs, err := conf.ParseConfigFile("config.toml")
    if err != nil {
        log.Fatal(err)
    }
    config := &configs[0]

    client, err := client.Dial(config)
    if err != nil {
        log.Fatal(err)
    }
    address, tx, instance, err := kvtable.DeployKVTableTest(client.
↪GetTransactOpts(), client)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("contract address: ", address.Hex())  // the address should be␣
↪saved
    fmt.Println("transaction hash: ", tx.Hash().Hex())
    _ = instance
}
```

```
# 该指令在go-sdk目录中执行
go run kvtabletest/cmd/kvtabletest_main.go
```

注解:

- 合约地址需要手动保存，调用合约接口时使用

调用合约set/get接口

在 contract 文件夹中新建 kvtabletest_set.go 文件，该文件调用合约 set 接口，向 t_kvtest 表中插入一条数据：id="100010001001"、item_name="Laptop"、item_price=6000。然后调用get接口查询数据。

```
package main

import (
        "fmt"
        "log"
        "math/big"
        "strings"

        "github.com/FISCO-BCOS/go-sdk/abi"
        "github.com/FISCO-BCOS/go-sdk/client"
        "github.com/FISCO-BCOS/go-sdk/conf"
        kvtable "github.com/FISCO-BCOS/go-sdk/kvtabletest"
        "github.com/ethereum/go-ethereum/common"
)

func main() {
```

(continues on next page)

(续上页)

```
        configs, err := conf.ParseConfigFile("config.toml")
        if err != nil {
                log.Fatal(err)
        }
        config := &configs[0]

        client, err := client.Dial(config)
        if err != nil {
                log.Fatal(err)
        }

        // load the contract
        contractAddress := common.HexToAddress(contract address in hex string) //␣
↪deploy contract to get address
        instance, err := kvtable.NewKVTableTest(contractAddress, client)
        if err != nil {
                log.Fatal(err)
        }

        kvtabletestSession := &kvtable.KVTableTestSession{Contract: instance,␣
↪CallOpts: *client.GetCallOpts(), TransactOpts: *client.GetTransactOpts()}

        id := "100010001001"
        item_name := "Laptop"
        item_price := big.NewInt(6000)
        tx, receipt, err := kvtabletestSession.Set(id, item_price, item_name) //␣
↪call set API
        if err != nil {
                log.Fatal(err)
        }
        fmt.Printf("tx sent: %s\n", tx.Hash().Hex())
        // 解析abi
        kvtableTestABI, err := abi.JSON(strings.NewReader(kvtable.KVTableTestABI))
        if err != nil {
                fmt.Printf("parse abi failed, err: %v\n", err)
                return
        }
        // kvtableTestABI 解析返回值
        ret := big.NewInt(0)
        err = kvtableTestABI.Unpack(&ret, "set", common.FromHex(receipt.Output))
        if err != nil {
                fmt.Printf("parse return value failed, err: %v\n", err)
                return
        }

    fmt.Printf("seted lines: %v\n", ret.String())

        success, item_price, item_name, err := kvtabletestSession.Get(id) // call␣
↪get API
        if err != nil {
                log.Fatal(err)
        }
        if !success {
                log.Fatalf("id: %v is not found \n", id)
        }
        fmt.Printf("id: %v, item_price: %v, item_name: %v \n", id, item_price,␣
↪item_name)
}
```

异步接口使用样例

异步合约开发指的是调用编译生成的 go 文件中提供的异步接口部署合约、修改数据，可以极大提高交易并发量。以 KVTableTest 为例，编译生成 go 文件 的步骤 同上

异步部署、调用KVTableTest合约

利用生成的异步接口部署和调用合约。通过注入 handler 函数，处理交易回执，获取交易回执中的 output。

```
package main

import (
    "encoding/hex"
    "fmt"
    "log"
    "math/big"
    "strings"

    "github.com/ethereum/go-ethereum/common"

    "github.com/FISCO-BCOS/go-sdk/abi"
    "github.com/FISCO-BCOS/go-sdk/client"
    "github.com/FISCO-BCOS/go-sdk/conf"
    "github.com/FISCO-BCOS/go-sdk/core/types"
    kvtable "github.com/FISCO-BCOS/go-sdk/examples" // import kvtabletest
)

var (
    channel         = make(chan int, 0)
    contractAddress common.Address
)

func deployContractHandler(receipt *types.Receipt, err error) {
    if err != nil {
        fmt.Printf("%v\n", err)
        return
    }
    fmt.Println("contract address: ", receipt.ContractAddress.Hex()) // the␣
↪address should be saved
    contractAddress = receipt.ContractAddress
    channel <- 0
}

func invokeSetHandler(receipt *types.Receipt, err error) {
    if err != nil {
        fmt.Printf("%v\n", err)
        return
    }
    setedLines, err := parseOutput(kvtable.KVTableTestABI, "set", receipt)
    if err != nil {
        log.Fatalf("error when transfer string to int: %v\n", err)
    }
    fmt.Printf("seted lines: %v\n", setedLines.Int64())
    channel <- 0
}

func main() {
    configs, err := conf.ParseConfigFile("config.toml")
    if err != nil {
        log.Fatal(err)
```

(continues on next page)

(续上页)

```
    }
    config := &configs[0]

    // AsyncDeploy
    fmt.Println("-------------------starting deploy contract----------------------
↪")
    client, err := client.Dial(config)
    if err != nil {
        log.Fatal(err)
    }
    tx, err := kvtable.AsyncDeployKVTableTest(client.GetTransactOpts(),␣
↪deployContractHandler, client)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("transaction hash: ", tx.Hash().Hex())
    <-channel

    // invoke AsyncSet to insert info
    fmt.Println("\n-------------------starting invoke Set to insert info-----------
↪------------")
    instance, err := kvtable.NewKVTableTest(contractAddress, client)
    if err != nil {
        log.Fatal(err)
    }
    kvtabletestSession := &kvtable.KVTableTestSession{Contract: instance,␣
↪CallOpts: *client.GetCallOpts(), TransactOpts: *client.GetTransactOpts()}
    id := "100010001001"
    item_name := "Laptop"
    item_price := big.NewInt(6000)
    tx, err = kvtabletestSession.AsyncSet(invokeSetHandler, id, item_price, item_
↪name) // call set API
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("tx sent: %s\n", tx.Hash().Hex())
    <-channel

    // invoke Get to query info
    fmt.Println("\n-------------------starting invoke Get to query info------------
↪-----------")
    bool, item_price, item_name, err := kvtabletestSession.Get(id) // call get API
    if err != nil {
        log.Fatal(err)
    }
    if !bool {
        log.Fatalf("id: %v is not found \n", id)
    }
    fmt.Printf("id: %v, item_price: %v, item_name: %v \n", id, item_price, item_
↪name)
}

func parseOutput(abiStr, name string, receipt *types.Receipt) (*big.Int, error) {
    parsed, err := abi.JSON(strings.NewReader(abiStr))
    if err != nil {
        fmt.Printf("parse ABI failed, err: %v", err)
    }
    var ret *big.Int
    b, err := hex.DecodeString(receipt.Output[2:])
    if err != nil {
        return nil, fmt.Errorf("decode receipt.Output[2:] failed, err: %v", err)
```

(continues on next page)

(续上页)

```
    }
    err = parsed.Unpack(&ret, name, b)
    if err != nil {
        return nil, fmt.Errorf("unpack %v failed, err: %v", name, err)
    }
    return ret, nil
}
```

异步部署、调用HelloWorld合约

```
package main

import (
    "fmt"
    "log"

    "github.com/FISCO-BCOS/go-sdk/client"
    "github.com/FISCO-BCOS/go-sdk/conf"
    "github.com/FISCO-BCOS/go-sdk/helloworld"
    "github.com/ethereum/go-ethereum/common"
    "github.com/FISCO-BCOS/go-sdk/core/types"
)

func main() {
        configs, err := conf.ParseConfigFile("config.toml")
        if err != nil {
                log.Fatalf("ParseConfigFile failed, err: %v", err)
        }
        client, err := client.Dial(&configs[0])
        if err != nil {
                fmt.Printf("Dial Client failed, err:%v", err)
                return
        }
        var contractAddress common.Address
        var channel = make(chan int, 0)
        tx, err := helloworld.AsyncDeployHelloWorld(client.GetTransactOpts(),␣
↪func(receipt *types.Receipt, err error) {
                if err != nil {
                        fmt.Printf("%v\n", err)
                        return
                }
                fmt.Println("contract address: ", receipt.ContractAddress.Hex()) //
↪ the address should be saved
                contractAddress = receipt.ContractAddress
                channel <- 0
        }, client)
        fmt.Println("transaction hash: ", tx.Hash().Hex())
        <-channel
        instance, err := helloworld.NewHelloWorld(contractAddress, client)
        if err != nil {
                log.Fatal(err)
        }
        if err != nil {
                fmt.Printf("Deploy failed, err:%v", err)
                return
        }
        hello := &helloworld.HelloWorldSession{Contract: instance, CallOpts:␣
↪*client.GetCallOpts(), TransactOpts: *client.GetTransactOpts()}
        ret, err := hello.Get()
```

(continues on next page)

(续上页)

```
        if err != nil {
                fmt.Printf("hello.Get() failed: %v", err)
                return
        }
        fmt.Printf("Get: %s\n", ret)
        tx, err = hello.AsyncSet(func(receipt *types.Receipt, err error) {
                if err != nil {
                        fmt.Printf("hello.AsyncSet failed: %v\n", err)
                        return
                }
                if receipt.Status != 0 {
                        fmt.Printf("hello.AsyncSet failed: %v\n", receipt.
↪GetErrorMessage())
                }
                channel <- 0
        }, "fisco")
        <-channel
        ret, err = hello.Get()
        if err != nil {
                fmt.Printf("hello.Get() failed: %v", err)
                return
        }
        fmt.Printf("Get: %s\n", ret)
}
```

国密样例

使用国密特性的开发流程和非国密大致相同，不同点在于以下几部分：

- 搭建的 FISCO BCOS 区块链网络需要开启国密特性，可参考：国密支持
- go-sdk 的 config.toml 配置文件中 KeyFile 配置项，需要将非国密私钥替换为国密私钥
- go-sdk 的 config.toml 配置文件中 SMCrypto 配置项，需要修改为 true
- 安装 solc 编译器时需要添加 **-g** 选项，替换为国密版本
- 使用 abigen 工具将 bin 和 abi 转换为 go 文件时，需要添加参数 **–smcrypto=true**

HelloWorld样例

准备HelloWorld.sol合约文件

在 go-sdk 主目录中新建 helloworld 文件夹，在该文件夹中创建 HelloWorld.sol 合约。该合约提供两个接口，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 <0.6.11;

contract HelloWorld {
    string name;

    constructor() public {
        name = "Hello, World!";
    }

    function get() public view returns (string memory) {
        return name;
    }
```

(continues on next page)

(续上页)

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

安装国密solc编译器

该编译器用于将 sol 合约文件编译成 abi 和 bin 文件

```
# 该指令在helloworld文件夹中执行
bash ../tools/download_solc.sh -v 0.4.25 -g
```

构建go-sdk的代码生成工具abigen

该工具用于将 abi 和 bin 文件转换为 go 文件

```
# 该指令在helloworld文件夹中执行，编译生成abigen工具
go build ../cmd/abigen
```

编译生成go文件

先利用solc编译合约文件HelloWorld.sol，生成abi和bin文件

```
# 该指令在helloworld文件夹中执行
./solc-0.4.25-gm --bin --abi -o ./ ./HelloWorld.sol
```

helloworld目录下会生成HelloWorld.bin和HelloWorld.abi。此时利用abigen工具将HelloWorld.bin和HelloWorld.abi转换成HelloWorld.go：

```
# 该指令在helloworld文件夹中执行
./abigen --bin ./HelloWorld.bin --abi ./HelloWorld.abi --pkg helloworld --type␣
↪HelloWorld --out ./HelloWorld.go --smcrypto=true
```

- 接下来的步骤同非国密，不占用多余篇幅

8.8.5 AMOP 使用案例

标签：`go-sdk` `AMOP`

AMOP（Advanced Messages Onchain Protocol）即链上信使协议，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。

进一步了解 AMOP，请参考：链上信使协议。案例源码，请参考：go-sdk

初始化：

- 搭建单群组四节点区块链网络，可参考：安装。

单播案例

单播 指的是节点从监听相同 Topic 的多个订阅者中随机抽取一个订阅者转发消息，流程详细可参考 单播时序图

- 启动 AMOP 消息订阅者：

```
# go run examples/amop/sub/subscriber.go [endpoint] [topic]
> go run examples/amop/sub/subscriber.go 127.0.0.1:20201 hello

  Subscriber success
  2020/08/11 21:21:50 received: hello, FISCO BCOS, I am unique broadcast␣
↪publisher! 0
  2020/08/11 21:21:52 received: hello, FISCO BCOS, I am unique broadcast␣
↪publisher! 1
  2020/08/11 21:21:54 received: hello, FISCO BCOS, I am unique broadcast␣
↪publisher! 2
  2020/08/11 21:21:56 received: hello, FISCO BCOS, I am unique broadcast␣
↪publisher! 3
```

- 运行 AMOP 消息发布者：

```
# go run examples/amop/unicast_pub/publisher.go [endpoint] [topic]
> go run examples/amop/unicast_pub/publisher.go 127.0.0.1:20200 hello

  2020/08/11 21:21:50 publish message: hello, FISCO BCOS, I am unique␣
↪broadcast publisher! 0
  2020/08/11 21:21:52 publish message: hello, FISCO BCOS, I am unique␣
↪broadcast publisher! 1
  2020/08/11 21:21:54 publish message: hello, FISCO BCOS, I am unique␣
↪broadcast publisher! 2
  2020/08/11 21:21:56 publish message: hello, FISCO BCOS, I am unique␣
↪broadcast publisher! 3
```

多播案例

多播 指的是节点向监听相同 Topic 的所有订阅者转发消息。只要网络正常，即使没有监听 Topic 的订阅者，消息发布者也会收到节点消息推送成功的响应包，流程详细可参考 多播时序图

- 启动 AMOP 消息订阅者：

```
# go run examples/amop/sub/subscriber.go [endpoint] [topic]
> go run examples/amop/sub/subscriber.go 127.0.0.1:20201 hello

  Subscriber success
  2020/08/11 21:23:54 received: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 0
  2020/08/11 21:23:56 received: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 1
  2020/08/11 21:23:58 received: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 2
  2020/08/11 21:24:00 received: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 3
```

- 运行 AMOP 消息发布者：

```
# go run examples/amop/multicast_pub/publisher.go [endpoint] [topic]
> go run examples/amop/multicast_pub/publisher.go 127.0.0.1:20200 hello

  2020/08/11 21:23:54 publish message: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 0
  2020/08/11 21:23:56 publish message: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 1
  2020/08/11 21:23:58 publish message: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 2
  2020/08/11 21:24:00 publish message: hello, FISCO BCOS, I am multi broadcast␣
↪publisher! 3
```

身份验证单播案例

常规场景中，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。针对此类场景，FISCO BCOS 推出了 topic 认证功能。 认证功能是指对于特定的topic消息，只允许通过认证的接收者接收消息。详细请参考：Topic认证功能

- 启动 AMOP 消息订阅者：

```
# go run examples/amop_auth/sub/subscriber.go [endpoint] [topic]
> go run examples/amop_auth/sub/subscriber.go 127.0.0.1:20201 hello

Subscriber success
2020/08/27 15:59:33 received: Hi, FISCO BCOS! 0
2020/08/27 15:59:35 received: Hi, FISCO BCOS! 1
2020/08/27 15:59:37 received: Hi, FISCO BCOS! 2
2020/08/27 15:59:39 received: Hi, FISCO BCOS! 3
```

- 运行 AMOP 消息发布者：

```
# go run examples/amop_auth/unicast_pub/publisher.go [endpoint] [topic]
> go run examples/amop_auth/unicast_pub/publisher.go 127.0.0.1:20200 hello

publish topic success
2020/08/27 15:59:33 publish message: Hi, FISCO BCOS! 0
2020/08/27 15:59:35 publish message: Hi, FISCO BCOS! 1
2020/08/27 15:59:37 publish message: Hi, FISCO BCOS! 2
2020/08/27 15:59:39 publish message: Hi, FISCO BCOS! 3
```

身份验证多播案例

同理，FISCO BCOS 支持带身份验证的消息多播功能

- 启动 AMOP 消息订阅者：

```
# go run examples/amop_auth/sub/subscriber.go [endpoint] [topic]
> go run examples/amop_auth/sub/subscriber.go 127.0.0.1:20201 hello

Subscriber success
2020/08/27 16:02:39 received: Hi, FISCO BCOS! 1
2020/08/27 16:02:41 received: Hi, FISCO BCOS! 2
2020/08/27 16:02:43 received: Hi, FISCO BCOS! 3
2020/08/27 16:02:45 received: Hi, FISCO BCOS! 4
```

- 运行 AMOP 消息发布者：

```
# go run examples/amop_auth/multicast_pub/publisher.go [endpoint] [topic]
> go run examples/amop_auth/multicast_pub/publisher.go 127.0.0.1:20200 hello

publish topic success
2020/08/27 16:02:37 publish message: Hi, FISCO BCOS! 0
2020/08/27 16:02:39 publish message: Hi, FISCO BCOS! 1
2020/08/27 16:02:41 publish message: Hi, FISCO BCOS! 2
2020/08/27 16:02:43 publish message: Hi, FISCO BCOS! 3
2020/08/27 16:02:45 publish message: Hi, FISCO BCOS! 4
```

8.9 使用链上信使协议

标签：AMOP 链上信使 私有话题 认证流程

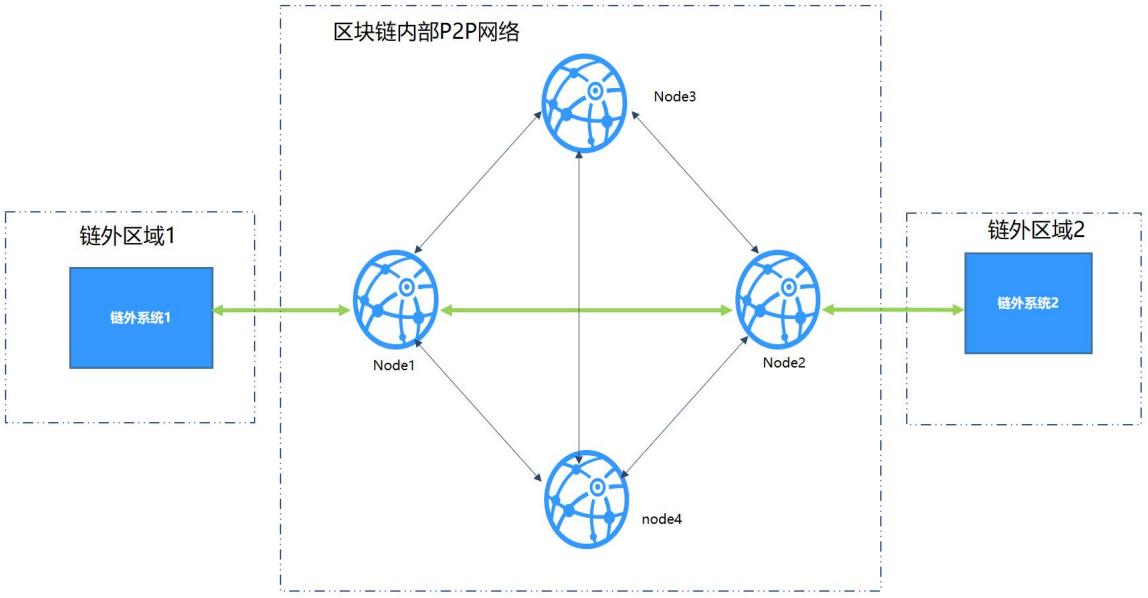
8.9.1 介绍

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。

AMOP功能使用请参照Java SDK AMOP.

8.9.2 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。

8.9.3 普通话题

AMOP的消息收发基于话题（Topic）订阅机制，订阅者首先订阅一个话题，发送者往该话题发送消息，订阅者即可收到。

发送方式和内容

总共支持两种传播方式：

- **单播**：仅随机发送给一个指定话题的订阅者。
- **多播**：发送给所有指定话题的订阅者。

发送内容：

- **文本消息**
- **文件**

8.9.4 私有话题

在普通的配置下，任何一个订阅了某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，需要使用私有话题功能。

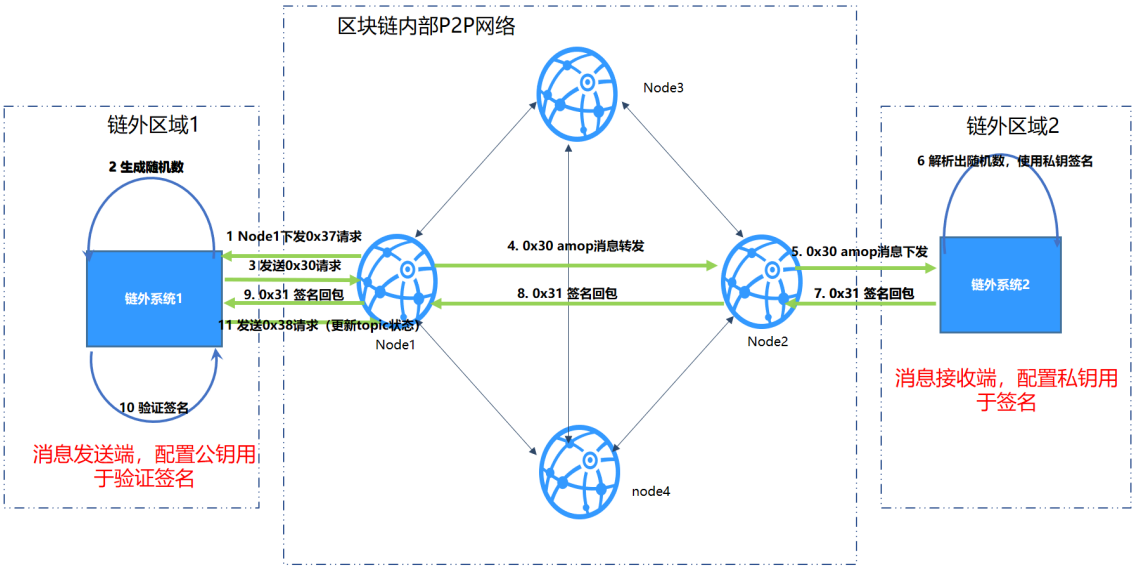
私有话题：对于特定的话题，发送者配置好期望的接受者的公钥，仅该公钥所对应的订阅者才能接收该私有话题的消息。

私有话题功能是自FISCO BCOS 2.1.0 之后新增的。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。

私有话题的认证流程

假定链外系统1是话题消息发送者（消息发送端），链外系统2是话题订阅者（消息接收端）。私有话题的认证流程如下：



- 1：链外系统2连接Node2,宣称订阅T1,Node2将T1加入到话题列表，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，发现不一致，则从Node2获取最新的话题列表，并将话题列表更新到p2p的话题列表，对于还没认证的私有话题，状态置为待认证。Node1遍历列表。针对每一个待认证的私有话题,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发起私有话题认证流程。
  - 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。

发送方式和内容

同样的私有话题支持单播和多播，发送文本和文件。

8.9.5 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。

8.10 开发并行合约

标签：并行合约 合约开发 智能合约

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。

8.10.1 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。

8.10.2 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
    {
        // 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
```

(continues on next page)

(续上页)

```
}

// 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(2.6及其以上版本控制台使用手册请参考这里，1.x版本控制台使用手册请参考这里)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
{
```

(continues on next page)

(续上页)

```
    // 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))
        {
            // 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。

8.10.3 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。

8.11 合约隐私保护

标签：隐私合约 隐私保护 合约开发 同态加密 环签名

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。

注解:

1. FISCO BCOS 2.3.0+ 支持同态加密、群签名和环签名
2. FISCO BCOS 2.3.0， 2.4.0以及2.4.1，需要手动编译二进制启用隐私保护模块
3. FISCO BCOS 2.5.0+默认启用隐私保护模块

8.11.1 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：

$$C_1 = Encrypt(m_1, pk)$$
$$C_2 = Encrypt(m_2, pk)$$
$$C_3 = Hom_{f()}(C_1, C_2, pk)$$
$$Decrypt(C_3, sk) = f(m_1, m_2)$$

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。

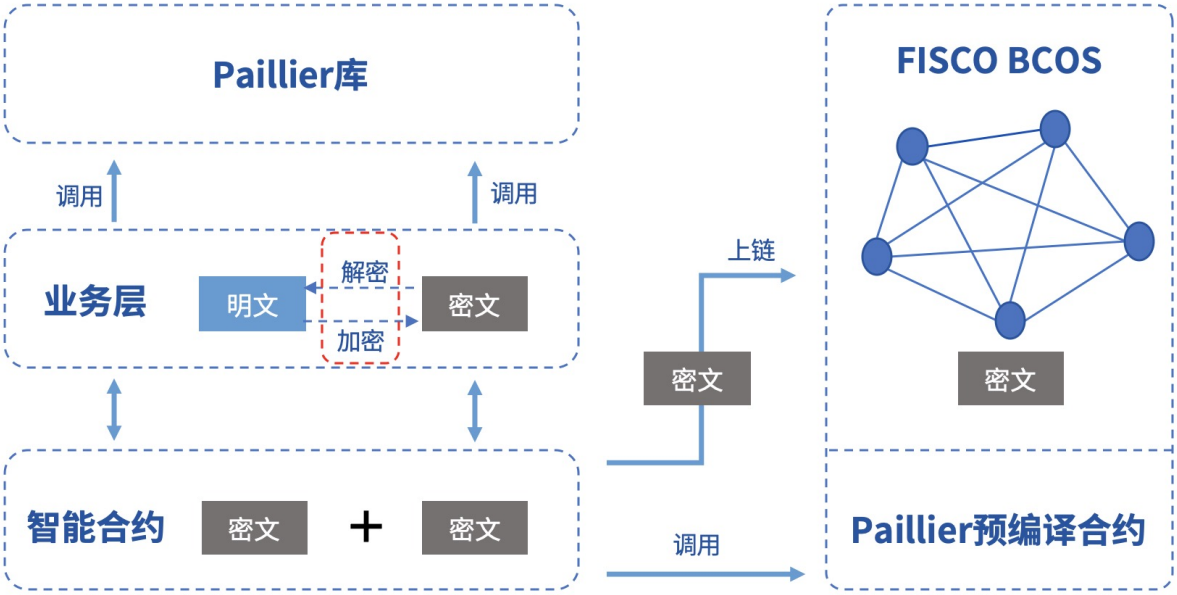
功能组件

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。

8.11.2 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。

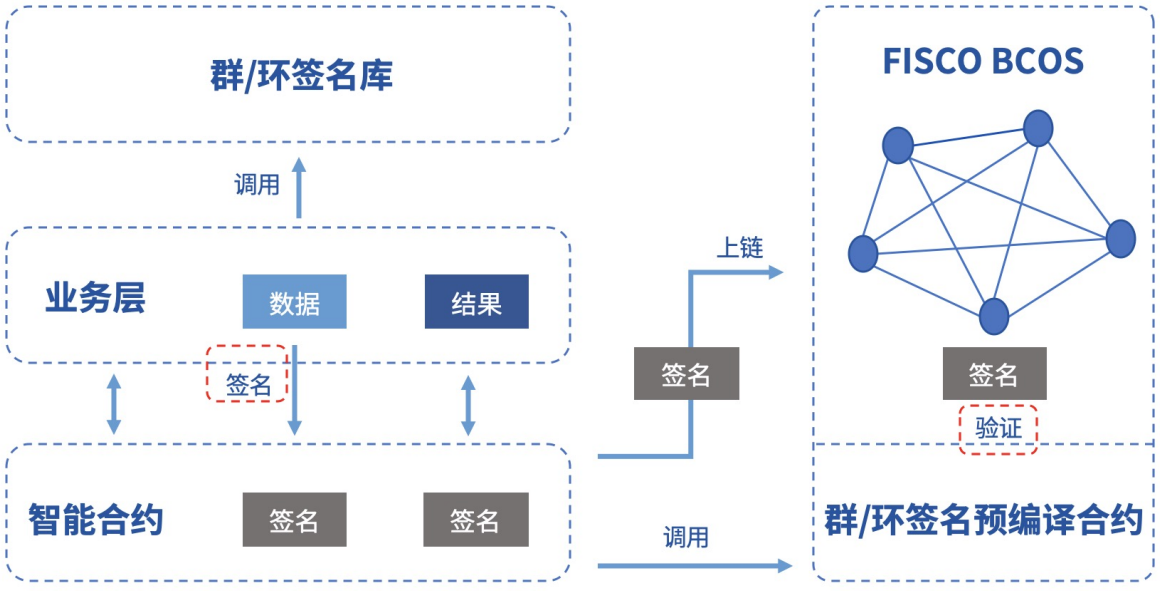
功能组件

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

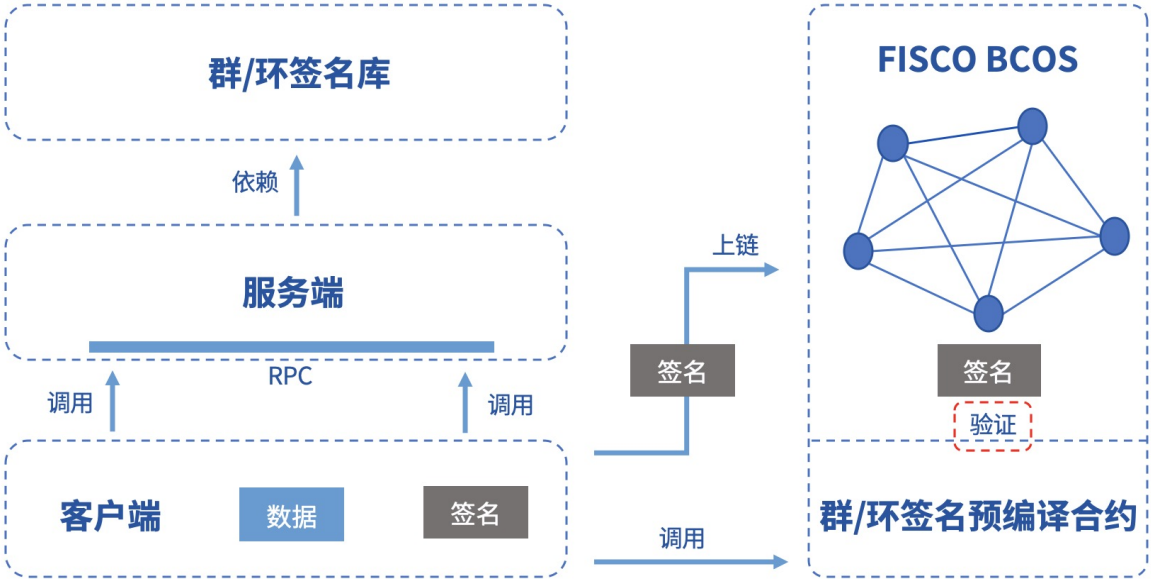
群/环签名由于其天然的匿名性，在需要对参与者身份进行隐匿的场景中有广泛的应用前景，例如匿名投票、匿名竞拍、匿名拍卖等等，甚至在区块链UTXO模型中可用于实现匿名转账。同时，由于群签名具备可追踪性，可以用于需要监管介入的场景，监管方作为群主或者委托群主揭露签名者身份。

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。



8.11.3 启用方法

FISCO BCOS隐私保护模块是通过预编译合约实现的，默认被启用。使用源码编译请参考这里.

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 ../manual/build_chain.sh -l 127.0.0.1:4 -e bin/fisco-bcos
```

8.11.4 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);
}
```

8.11.5 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 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。

8.11.6 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() {
```

(continues on next page)

(续上页)

```
        // 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。

CHAPTER 9

问题排查

标签： 问题排查

9.1 版本问题

标签： 问题排查 版本问题 版本不正确 Java版本 版本兼容性

9.1.1 FISCO BCOS版本疑问

问题:FISCO BCOS 2.0版本与之前版本有哪些变化？ 答案:请参考这里。

问题：国密和普通版本的区别有哪些？ 答案： 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。

问题：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？ 答案： 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。

问题：是否支持从1.3或1.5升级到2.0版本？ 答案： 不支持。

9.1.2 Java版本不正确（Java SDK）

Java SDK要求JDK版本大于等于1.8，推荐使用OpenJDK/OracleJDK 15以下版本的JDK，不同系统的Java安装请参考这里

9.2 连接失败

标签：连接失败 问题排查 Failed to connect to nodes 握手失败

9.2.1 Failed to connect to nodes

若SDK连接节点失败，抛出错误`Failed to connect to nodes: [Connection refused:`，可从以下思路排查:

第一步.检查节点进程

通过`ps aux | grep fisco-bcos`命令检查节点进程是否启动

第二步.检查节点监听IP

若SDK与节点处于不同机器，检查节点的channel服务的监听端口`channel_listen_ip`(位于SDK直连节点的config.ini配置文件中)是否为`0.0.0.0`或外网IP(**注:** 此外网IP非云服务器的虚拟IP，必须是机器的网卡IP)

第三步.检查SDK的连接配置

检查SDK配置的节点连接是否正确，Web3SDK的节点连接配置参考这里；Java SDK的节点连接配置参考这里

第四步.检查SDK与节点之间连通性

下载telnet工具，使用命令`telnet ${nodeIP}:${channel_listen_port}`检查SDK与节点之间是否可连通，其中`${nodeIP}`为节点IP，`${channel_listen_ip}`为节点Channel服务的监听端口，可通过节点目录下的`config.ini`配置文件中`rpc.channel_listen_port`配置选项获取，若SDK与节点之间不连通，请检查是否开启了防火墙/安全组策略，需要在防火墙/安全组中开放FISCO BCOS节点所使用的channel端口。

9.2.2 Java SDK握手失败

若Java SDK连接节点时，显示握手失败，请按以下步骤排查:

第一步.检查是否拷贝证书

若没有配置证书，请参考Java SDK快速入门配置SDK证书

第二步.检查SDK与直连节点是否处于相同机构

FISCO BCOS v2.5.0版本起，限制SDK只能连接本机构的节点，请确保SDK证书与直连节点证书属于相同机构

第三步.检查JDK版本

请确认JDK版本不小于JDK 1.8且不大于JDK 14。

第四步.检查证书配置是否正确

检查证书是否拷贝到默认的配置路径，请参考这里.

第五步.检查是否Jar包冲突

Jar包冲突会报如下几个错误。

- SSL_MODE_ENABLE_FALSE_START **问题描述** 基于**Java SDK 2.6.1-rc1**开发应用时，已经按照Java SDK快速入门配置了证书，启动报错如下:

```
java.lang.NoSuchFieldError: SSL_MODE_ENABLE_FALSE_START
    at io.netty.handler.ssl.ReferenceCountedOpenSslEngine.
↪(ReferenceCountedOpenSslEngine.java:355) ~[netty-all-4.1.52.Final.jar:4.1.52.
↪Final]
    at io.netty.handler.ssl.OpenSslEngine.(OpenSslEngine.java:32) ~[netty-all-
↪4.1.52.Final.jar:4.1.52.Final]
```

问题分析: 应用中引入了4.1.52版本的netty，使用了较高版本的`tcnative`，与Java SDK 2.6.1-rc1中的tcnative版本冲突，导致Java SDK初始化失败 **解决方法:** 升级Java SDK 2.6.1-rc1到Java SDK 2.6.1或者最新，请参考引入Java SDK.

- PrivateKey type not supported PKCS#8 **问题描述** 应用中同时Java SDK 2.6.1和Web3SDK 2.6.0(或Web3SDK 2.6.1)，已经按照Java SDK快速入门配置了证书，SDK启动报错如下：

```
create BcosSDK failed, error info: init channel network error: SSL context␣
↪init failed, please make sure your cert and key files are properly␣
↪configured. error info: PrivateKey type not supported PKCS#8
```

问题分析： Java SDK 2.6.1使用了较高版本的netty和tcnative，Web3SDK 2.6.0与Web3SDK 2.6.1使用的netty和tcnative版本较低，两者同时使用时jar包冲突，导致SDK启动报错。 **解决方法** 由于Java SDK和Web3SDK包含相同的功能，但Java SDK功能更全且会持续迭代，因此不建议同时使用Web3SDK与Java SDK，**建议使用Java SDK替代Web3SDK**.

  - **方法一：** 将Web3SDK升级到`2.6.2`版本，使其与Java SDK基于同样版本的netty与tcnative
  - **方法二：** 去除对Web3SDK的依赖，仅依赖Java SDK

9.2.3 web3SDK握手失败

若Web3SDK连接节点时，显示握手失败，请按以下步骤排查:

第一步.检查证书配置

若没有配置证书，请参考Web3SDK证书配置配置SDK证书

第二步.检查SDK与直连节点是否处于相同机构

FISCO BCOS v2.5.0版本起，限制SDK只能连接本机构的节点，请确保SDK证书与直连节点证书属于相同机构

第三步.检查JDK版本

Web3SDK要求JDK版本大于等于1.8，推荐使用OracleJDK。 注：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环境安装请参考Java环境配置

第四步.针对Web3SDK常见的异常启动问题进行排查

- 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。

- Failed to connect to nodes: [ssl handshake failed:/127.0.0.1:20200], PKIX path validation failed: java.security.cert.CertPathValidatorException:Algorithm constraints check failed on disabled algorithm: secp256k1 SDK依赖的Java禁用了`secp256k1`曲线，解决方法包括：

  - 使用1.8以上的OracleJDK，启动SDK时，加上`-Djdk.tls.namedGroups="secp256k1"`参数启用`secp256k1`曲线
  - 升级到最新的`Web3SDK 2.6.2`版本

- java.lang.NoSuchFieldError: SSL_MODE_ENABLE_FALSE_START

  请参考web3sdk issue #711

9.3 证书问题

标签：证书问题 问题排查 证书位置 证书过期

9.3.1 证书放置位置出错（Java SDK）

问题描述

将证书放置于`resources`目录，SDK连接节点时，报出如下错误：

```
org.fisco.bcos.sdk.channel.ChannelImp    : init channel network error, Not␣
↪providing all the certificates to connect to the node! Please provide the␣
↪certificates to connect with the block-chain.
```

问题分析：

Java SDK默认的证书配置路径是`conf`，将证书放置与`resources`目录下但没有修改证书配置，导致SDK与节点建连时无法读取证书

解决方法：

- **方法一**：在项目根目录下新建`conf`文件夹，并将SDK证书拷贝到该目录，详细参考Java SDK快速入门章节的证书配置操作
- **方法二**：修改证书配置选项`certPath`，使其指向当前证书路径，证书配置选项说明参考这里
- **方法三**: 使用`>= 2.7.1`版本的Java SDK

9.3.2 证书过期

问题描述

节点或SDK使用的OpenSSL证书过期了，如何续期?

解决方法:

证书续期操作可以参考证书续期操作

9.3.3 证书验证失败（运维部署工具）

问题描述

使用运维部署工具下载命令提示certificate verify failed

解决方法:

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
```

9.3.4 证书配置错误（1.x国密版控制台）

问题描述

搭建国密版本节点，并将config.ini中的chain.sm_crypto_channel配置为true，且参考安装教程与配置国密版控制台拷贝了证书、配置了国密开关(将applicationContext.xml中的encryptType设置为1)，启动节点报错如下:

```
Failed to init the console!  Failed to connect to nodes: [ ssl handshake failed:/
↪127.0.0.1:20200]The reasons for failure may be:
        1. the configured certificate is not the same set of certificates as the␣
↪node's certificate;
        2. the configured certificate is not issued by the same authority as the␣
↪node's certificate.
        Please refer to https://fisco-bcos-documentation.readthedocs.io/zh_CN/
↪latest/docs/sdk/java_sdk.html#id24
```

问题分析

节点的chain.sm_crypto_channel=true时，SDK与节点之间应采用国密SSL的方式通信，控制台应该加载国密证书，默认的applicationContext.xml从如下配置中加载国密证书:

```
        <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" />
```

生成的SDK国密证书位于sdk/gm子目录下，直接拷贝SDK证书到console/conf文件夹下，会找不到gmca.crt、gmensdk.crt等证书，当控制台配置路径下不存在这些证书时，控制台会采用非国密SSL的方式连接节点，握手失败。

解决方案

当使用国密SSL连接节点时候，须将国密SDK证书拷贝到console/conf子文件夹，操作如下:

```
cp console/conf/gm/* console/conf
```

注:当节点的chain.sm_crypto_channel=false时，须保证console/conf子文件夹下不存在gmca.crt、gmensdk.crt等国密证书与私钥，否则控制台会以国密SSL的方式初始化证书，抛出类似的错误，若出现该问题，可通过删除国密SDK证书，执行rm -rf console/conf/gm*的方式来解决

9.4 控制台报错

标签：控制台报错 问题排查

9.4.1 控制台疑问

问题：控制台指令是否区分大小写

答案： 区分大小写，命令是完全匹配，可采用tab补全命令。

问题：控制台是否支持群组切换功能

答案： 可以切换群组。

- **通过switch命令切换群组：** 控制台提供了switch命令支持群组切换，可通过switch [groupID]([groupID]是切换到的群组ID)，详细请参考这里
- **启动控制台指定需要访问的群组ID:** 使用./start [groupID]([groupID]是切换到的群组ID)启动控制台

问题：游离节点是否可以同步group数据

答案： 游离节点（非群组节点）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。

9.4.2 合约调用回滚

问题描述

版本号不大于1.0.10国密版控制台调用合约时，报错The execution of the contract rolled back，具体如下:

```
[group:1]> deploy HelloWorld
contract address: 0x8396a8793ce537bed6d894e456ea1ba70a449447

[group:1]> call HelloWorld 0x8396a8793ce537bed6d894e456ea1ba70a449447 get
The execution of the contract rolled back.
```

问题分析

版本号不大于1.0.10国密版控制台调用合约时，须将solc包替换为国密版本，否则部署到节点的交易字节码为非国密字节码，国密节点使用非国密字节码执行交易时，交易执行出错，导致交易回滚。

解决方案

方法一: 升级到最新版本控制台，可参考配置和使用控制台 **方法二：** 将console/lib目录下的solcJ-all-0.4.25.jar替换为国密版本的solc，具体操作步骤如下:

```
# 获取国密版solc(须保证安装了curl)
# 若curl的时候报错: curl: (7) Failed to connect to raw.githubusercontent.com port␣
↪443: Connection refused
# 可以修改/etc/hosts文件，追加如下配置:
# 199.232.28.133 raw.githubusercontent.com
curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/solcj/solcJ-all-
↪0.4.25-gm.jar
```

(continues on next page)

(续上页)

```
# 替换非国密的`solc`为国密的solc
cp solcJ-all-0.4.25-gm.jar lib/ && rm -rf lib/solcJ-all-0.4.25.jar

# 重新启动控制台，部署并调用合约
bash start.sh
```

9.4.3 节点加入共识列表报错

问题分析

为保证新加入共识节点不影响共识，要求新加入节点必须处于运行状态，且与其他共识节点建立网络连接，当新加入的节点未启动或没有与共识节点建立连接时，会报出`nodeID is not in network`的错误。

解决方案

参考这里，为新加入节点配置所有其他共识节点的连接信息，启动新加入节点，重新通过控制台发送`addSealer`命令，将新节点加入到共识节点列表。

9.4.4 删除节点报错

问题描述

删除节点报错`nodeID is not in group peers`

问题分析

正常报错，当被删除的节点不属于群组时，会报出该错误，可通过getGroupPeers命令确定群组是否在该群组中.

9.5 运维部署工具无法使用

标签：没有网络 找不到`pip` 运维部署工具

9.5.1 没有网络

问题描述

在没有网络的情况下，如何使用运维部署工具?

解决方法

运维部署工具依赖configparser等模块，建议在可以连接外网的情况下配置运维部署工具，并生成节点配置文件夹，推送至内网服务器。

9.5.2 找不到pip

问题描述

运维部署工具使用时出现找不到pip的错误。

解决方法

运维部署工具依赖python pip，需要使用以下命令安装:

```
$ bash ./scripts/install.sh
```

9.5.3 找不到configparse

问题描述

运维部署工具使用时出现找不到configparse模块的错误。

```
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
```

解决方法

系统缺少python configparser模块，请按照以下命令安装：

```
  $ pip install configparser
```

CHAPTER 10

开发部署工具(build_chain.sh)

标签：build_chain 搭建区块链网络

重要： 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。

10.1 功能介绍

- 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。脚本的源码位于这里。
- 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

10.2 使用方法

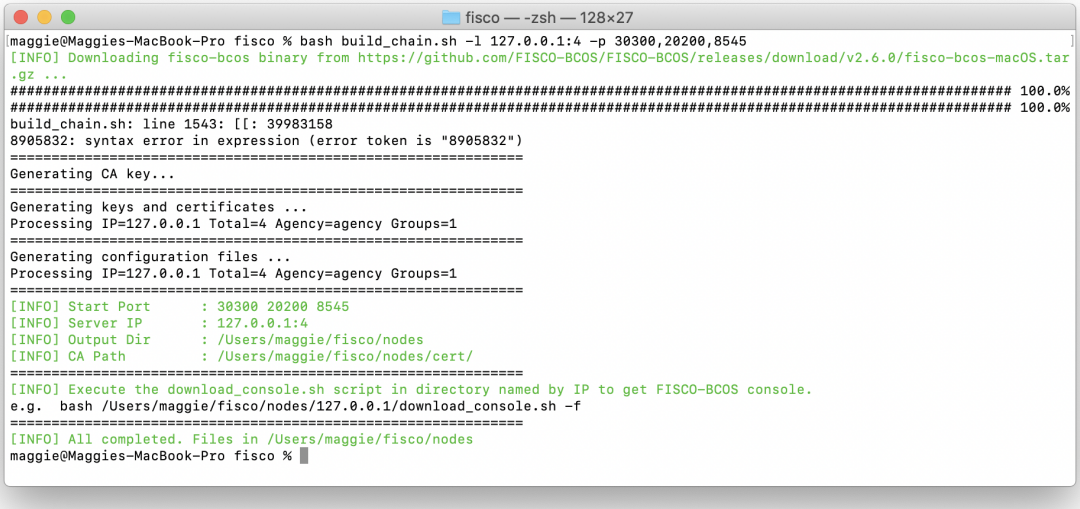
```
Usage:
    -l <IP list>                        [Required] "ip1:nodeNum1,ip2:nodeNum2" e.g:
↪"192.168.0.1:2,192.168.0.2:3"
    -f <IP list file>                   [Optional] split by line, every line␣
↪should be "ip:nodeNum agencyName groupList p2p_port,channel_port,jsonrpc_port".␣
↪eg "127.0.0.1:4 agency1 1,2 30300,20200,8545"
    -v <FISCO-BCOS binary version>      Default is the latest v${default_version}
```

(continues on next page)

(续上页)

```
    -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
    -q <List FISCO-BCOS releases>       List FISCO-BCOS released versions
    -i <Host ip>                        Default 127.0.0.1. If set -i, listen 0.0.0.
↪0
    -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
    -6 <Use ipv6>                       Default no. If set -6, treat IP as IPv6
    -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
```

比如：使用build_chain.sh部署4个本地节点。



10.2.1 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。

10.2.2 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
```

10.2.3 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下载最新的二进制程序。

```
# 从GitHub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
```

10.2.4 o选项[Optional]

指定生成的配置所在的目录。

10.2.5 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
```

10.2.6 q选项[Optional]

列出FISCO BCOS已经发布的所有版本号。

10.2.7 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。

10.2.8 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
```

10.2.9 s选项[Optional]

有参数选项，参数为db名，目前支持RocksDB、mysql、Scalable。默认使用rocks。

- 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。
- 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。

10.2.10 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。

10.2.11 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
```

10.2.12 g选项[Optional]

无参数选项，设置该选项时，搭建国密版本的FISCO BCOS。**确认sdk支持的情况下（web3sdk v2.5.0+），可以指定-g -G参数，连接也使用国密SSL**，-G设置chain.sm_crypto_channel=true

```
bash build_chain.sh -l 127.0.0.1:2 -g -G
```

10.2.13 z选项[Optional]

无参数选项，设置该选项时，生成节点的tar包。

10.2.14 t选项[Optional]

该选项用于指定生成证书时的证书配置文件。

10.2.15 6选项[Optional]

该选项表示使用IPv6模式，监听::

10.2.16 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。

10.2.17 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。

10.2.18 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。

10.2.19 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。**确认sdk支持的情况下（web3sdk v2.5.0+），可以指定-g -G参数，连接也使用国密SSL**

10.2.20 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。

10.2.21 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

10.3 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
│   ├── 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，FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错
│   │   ├── 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
```

10.4 使用工具脚本

介绍由build_chain.sh生成的脚本。

10.4.1 start_all.sh

启动当前目录下的所有节点。

10.4.2 stop_all.sh

停止当前目录下的所有节点。

10.4.3 download_console.sh

下载console的脚本。

- v选项支持下载指定版本的console
- f选项自动配置下载的console的证书和端口

10.4.4 download_bin.sh

用于下载fisco-bcos二进制程序，选项如下。

```
Usage:
    -v <Version>            Download binary of spectfic version, default latest
    -b <Branch>             Download binary of spectfic branch
    -o <Output Dir>         Default ./bin
    -l                      List FISCO-BCOS released versions
    -m                      Download mini binary, only works with -b option
    -h Help
e.g
    ./download_bin.sh -v 2.7.1
```

10.5 使用教程

10.5.1 无外网条件的搭建单群组区块链网络

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. 针对某些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...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
```

(continues on next page)

(续上页)

```
============================================================
[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
```

10.5.2 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=30304
    ;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1，2.6版本控制台指令详细介绍参考这里，1.x版本控制台指令详细介绍参考这里，`nodeID`可以通过命令`cat newNode/conf/node.nodeid`来获取
3. 检查连接和共识

10.5.3 为机构生成新的SDK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newSDK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 newSDK -s
```

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 newSDK -g ../gmcert/agency/ -s
```

10.5.4 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/
```

10.5.5 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。

CHAPTER 11

命令行交互控制台

标签：控制台

命令行交互控制台 (简称“控制台”) 是FISCO BCOS 2.0重要的交互式客户端工具，它通过 Java SDK 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。

重要:

- 控制台`2.6+` 基于 Java SDK 实现, 控制台`1.x` 系列基于 Web3SDK 实现，本教程针对 **2.6及其以上版本控制台**，1.x及其以上版本控制台使用文档请 参考这里
- 可通过命令 `./start.sh --version` 查看当前控制台版本
- 基于 Java SDK 开发应用过程中将 `solidity` 代码转换为 `java` 代码时，必须使用 `2.6+` 版本控制台，具体请参考 这里

11.1 下载和使用控制台

标签：下载控制台

`download_console.sh`脚本提供获取所有版本的控制台的功能，默认获取2.6+版本的控制台，可通过如下命令获取最新的`download_console.sh`脚本：

```
mkdir -p ~/fisco && cd ~/fisco && curl -#LO https://github.com/FISCO-BCOS/console/
↪releases/download/v2.7.1/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*

11.1.1 帮助

```
 bash download_console.sh -h

Usage:
    -c <console version>   Specify the downloaded console version, download the␣
↪latest version of the console by default
    -v <solc version>      Download the console with specific solc version,␣
↪default is 0.4, 0.5 and 0.6 are supported
    -h Help
e.g
    download_console.sh -v 0.6
```

11.1.2 选项介绍

download_console.sh脚本提供了-c和-v选项，分别用于指定下载的控制台版本号，以及控制台依赖的solidity编译器版本号。

-c选项

指定下载的控制台版本号，默认拉取最新版本控制台(2.6+版本)，若基于Web3SDK开发应用，需下载1.x版本的控制台将solidity代码转换为java代码，可通过该选项指定下载的控制台版本，下载1.2.0版本控制台的示例如下：

```
# 下载1.2.0版本的控制台
$ bash download_console.sh -c 1.2.0
```

-v选项

控制台1.1.0版本开始支持使用0.6版本的solidity编译器(**FISCO BCOS底层需要升级到v2.5.0及其以上版本**)，-v选项指定下载的控制台基于的solidity编译器版本，目前支持0.4, 0.5和0.6三个版本的solidity编译器，默认使用0.4.25版本编译器。 下载支持solidity 0.6版本的控制台示例如下：

```
# 下载支持0.6.0版本solidity编译器的控制台
$ bash download_console.sh -v 0.6
```

11.2 控制台2.6+版本

标签：控制台2.6

重要:

- 控制台2.6+ 基于 Java SDK 实现, 控制台1.x 系列基于 Web3SDK 实现，本教程针对 **2.6及其以上版本控制台**，1.x及其以上版本控制台使用文档请 参考这里
- 可通过命令 ./start.sh --version 查看当前控制台版本
- 基于 Java SDK 开发应用过程中将 solidity 代码转换为 java 代码时，必须使用 2.6+ 版本控制台，具体请参考 这里

控制台是FISCO BCOS 2.0重要的交互式客户端工具，它通过Java SDK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。

11.2.1 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/v2.7.1/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
│   ├── config-example.toml # 配置文件
│   ├── group-generate-config.toml # 创建群组的配置文件，具体可参考命
令genrateGroupFromFile
│   └── 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
|-- get_gm_account.sh # 账户生成脚本，国密版
|-- sol2java.sh # solidity合约文件编译为java合约文件的开发工具脚本
```

注意：默认下载的控制台内置`0.4.25`版本的`solidity`编译器，用户需要编译`0.5`或者`0.6`版本的合约时，可以通过下列命令获取内置对应编译器版本的控制台

```
# 0.5
curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/download_
↪console.sh && bash download_console.sh -v 0.5
# 0.6
curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/download_
↪console.sh && bash download_console.sh -v 0.6
```

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config-example.toml文件重命名为config.toml文件。配置config.toml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config.toml配置不用修改。**
  - FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错。

配置示例文件如下：

```
[cryptoMaterial]

certPath = "conf"                               # The certification path

# The following configurations take the certPath by default if commented
# caCert = "conf/ca.crt"                        # CA cert file path
                                                # If connect to the GM node, default␣
↪CA cert path is ${certPath}/gm/gmca.crt

# sslCert = "conf/sdk.crt"                      # SSL cert file path
                                                # If connect to the GM node, the␣
↪default SDK cert path is ${certPath}/gm/gmsdk.crt

# sslKey = "conf/sdk.key"                       # SSL key file path
                                                # If connect to the GM node, the␣
↪default SDK privateKey path is ${certPath}/gm/gmsdk.key

# enSslCert = "conf/gm/gmensdk.crt"             # GM encryption cert file path
                                                # default load the GM SSL encryption␣
↪cert from ${certPath}/gm/gmensdk.crt

# enSslKey = "conf/gm/gmensdk.key"              # GM ssl cert file path
                                                # default load the GM SSL encryption␣
↪privateKey from ${certPath}/gm/gmensdk.key

[network]
peers=["127.0.0.1:20200", "127.0.0.1:20201"]    # The peer list to connect

# Configure a private topic as a topic message sender.
# [[amop]]
# topicName = "PrivateTopic1"
# publicKeys = [ "conf/amop/consumer_public_key_1.pem" ]    # Public keys of the␣
↪nodes that you want to send AMOP message of this topic to.

# Configure a private topic as a topic subscriber.
# [[amop]]
# topicName = "PrivateTopic2"
# privateKey = "conf/amop/consumer_private_key.p12"         # Your private key␣
↪that used to subscriber verification.
# password = "123456"

[account]
keyStoreDir = "account"          # The directory to load/store the account file,␣
↪default is "account"
# accountFilePath = ""           # The account file path (default load from the␣
↪path specified by the keyStoreDir)
accountFileFormat = "pem"        # The storage format of account file (Default is
↪"pem", "p12" as an option)
```

(continues on next page)

(续上页)

```
# accountAddress = ""           # The transactions sending account address
                                # Default is a randomly generated account
                                # The randomly generated account is stored in the␣
↪path specified by the keyStoreDir

# password = ""                 # The password used to load the account file

[threadPool]
# channelProcessorThreadSize = "16"         # The size of the thread pool to␣
↪process channel callback
                                            # Default is the number of cpu cores

# receiptProcessorThreadSize = "16"         # The size of the thread pool to␣
↪process transaction receipt notification
                                            # Default is the number of cpu cores

maxBlockingQueueSize = "102400"             # The max blocking queue size of the␣
↪thread pool
```

配置项详细说明参考这里。

重要: 控制台说明

- 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。

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。

```
$ bash sol2java.sh -h
# Compile Solidity Tool
./sol2java.sh [packageName] [solidityFilePath] [javaCodeOutputDir]
          packageName:
                  the package name of the generated Java class file
          solidityFilePath:
                  (optional) the solidity file path or the directory where␣
↪solidity files located, default: contracts/solidity
          javaCodeOutputDir:
                  (optional) the directory where the generated Java files located,␣
↪default: contracts/sdk/java
```

参数

- packageName: 生成Java文件的包名
- solidityFilePath: (可选)solidity文件的路径，支持文件路径和目录路径两种方式，参数为目录时将目录下所有的solidity文件进行编译转换。默认目录为contracts/solidity。
- javaCodeOutputDir: (可选)生成Java文件的目录，默认生成在contracts/sdk/java目录。

使用

```
$ cd ~/fisco/console
$ ./sol2java.sh org.com.fisco # 指定java包名
```

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。

重要： Java合约文件说明

- 控制台自1.1.0版本起，生成的Java合约文件国密、非国密环境均可以运行，国密与非国密环境生成一份合约代码即可

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
========================================================================================
Welcome to FISCO BCOS console(2.6.0)!
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: 2.6.0
```

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

使用控制台配置文件指定的默认群组号启动。

```
./start.sh
```

注意: 控制台启动未指定私钥账户时，会尝试从accounts目录下加载一个可用的私钥账户用于发送交易，加载失败则会创建一个新的PEM格式的账户文件，将其保存在accounts目录下。

指定群组号启动

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

控制台命令结构

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。

常用命令

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

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

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。

11.2.2 控制台命令列表

help

输入help或者h，查看控制台所有的命令。

```
[group:1]> help
* help([-h, -help, --h, --H, --help, -H, h])  Provide help information
* addObserver                                 Add an observer node
* addSealer                                   Add a sealer node
* call                                        Call a contract by a function and␣
↪parameters
* callByCNS                                   Call a contract by a function and␣
↪parameters by CNS
* create                                      Create table by sql
* delete                                      Remove records by sql
* deploy                                      Deploy a contract on blockchain
* deployByCNS                                 Deploy a contract on blockchain by CNS
* desc                                        Description table information
* quit([quit, q, exit])                       Quit console
* freezeAccount                               Freeze the account
* freezeContract                              Freeze the contract
* generateGroup                               Generate a group for the specified node
* generateGroupFromFile                       Generate group according to the␣
↪specified file
* getAccountStatus                            GetAccountStatus of the account
* getAvailableConnections                     Get the connection information of the␣
↪nodes connected with the sdk
* getBatchReceiptsByBlockHashAndRange         Get batched transaction receipts␣
↪according to block hash and the transaction range
* getBatchReceiptsByBlockNumberAndRange       Get batched transaction receipts␣
↪according to block number and the transaction range
* getBlockByHash                              Query information about a block by hash
* getBlockByNumber                            Query information about a block by␣
↪number
* getBlockHashByNumber                        Query block hash by block number
* getBlockHeaderByHash                        Query information about a block header␣
↪by hash
* getBlockHeaderByNumber                      Query information about a block header␣
↪by block number
* getBlockNumber                              Query the number of most recent block
* getCode                                     Query code at a given address
* getConsensusStatus                          Query consensus status
* getContractStatus                           Get the status of the contract
* getCryptoType                               Get the current crypto type
* getCurrentAccount                           Get the current account info
* getDeployLog                                Query the log of deployed contracts
* getGroupConnections                         Get the node information of the group␣
↪connected to the SDK
```

(continues on next page)

(续上页)

```
* getGroupList                                Query group list
* getGroupPeers                               Query nodeId list for sealer and␣
↪observer nodes
* getNodeIDList                              Query nodeId list for all connected␣
↪nodes
* getNodeInfo                                Query the specified node information.
* getNodeVersion                             Query the current node version
* getObserverList                            Query nodeId list for observer nodes.
* getPbftView                                Query the pbft view of node
* 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
* 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
* getTransactionByHashWithProof              Query the transaction and transaction␣
↪proof by transaction hash
* getTransactionReceipt                      Query the receipt of a transaction by␣
↪transaction hash
* getTransactionReceiptByHashWithProof       Query the receipt and transaction␣
↪receipt proof by transaction hash
* grantCNSManager                            Grant permission for CNS by address
* grantCommitteeMember                       Grant the account committee member
* grantContractStatusManager                 Grant contract authorization to the␣
↪user
* grantContractWritePermission               Grant the account the contract write␣
↪permission.
* grantDeployAndCreateManager                Grant permission for deploy contract␣
↪and create user table by address
* grantNodeManager                           Grant permission for node␣
↪configuration by address
* grantOperator                              Grant the account operator
* grantSysConfigManager                      Grant permission for system␣
↪configuration by address
* grantUserTableManager                      Grant permission for user table by␣
↪table name and address
* insert                                     Insert records by sql
* listAbi                                    List functions and events info of the␣
↪contract.
* listAccount                                List the current saved account list
* listCNSManager                             Query permission information for CNS
* listCommitteeMembers                       List all committee members
* listContractStatusManager                  List the authorization of the contract
* listContractWritePermission                Query the account list which have␣
↪write permission of the contract.
* listDeployAndCreateManager                 Query permission information for␣
↪deploy contract and create user table
* listDeployContractAddress                  List the contractAddress for the␣
↪specified contract
* listNodeManager                            Query permission information for node␣
↪configuration
* listOperators                              List all operators
* listSysConfigManager                       Query permission information for␣
↪system configuration
```

(continues on next page)

(续上页)

```
* listUserTableManager                    Query permission for user table␣
↪information
* loadAccount                             Load account for the transaction␣
↪signature
* newAccount                              Create account
* queryCNS                                Query CNS information by contract name␣
↪and contract version
* queryCommitteeMemberWeight              Query the committee member weight
* queryGroupStatus                        Query the status of the specified␣
↪group of the specified node
* queryThreshold                          Query the threshold
* queryVotesOfMember                      Query votes of a committee member.
* queryVotesOfThreshold                   Query votes of updateThreshold␣
↪operation
* recoverGroup                            Recover the specified group of the␣
↪specified node
* registerCNS                             RegisterCNS information for the given␣
↪contract
* removeGroup                             Remove the specified group of the␣
↪specified node
* removeNode                              Remove a node
* revokeCNSManager                        Revoke permission for CNS by address
* revokeCommitteeMember                   Revoke the account from committee␣
↪member
* revokeContractStatusManager             Revoke contract authorization to the␣
↪user
* revokeContractWritePermission           Revoke the account the contract write␣
↪permission
* revokeDeployAndCreateManager            Revoke permission for deploy contract␣
↪and create user table by address
* revokeNodeManager                       Revoke permission for node␣
↪configuration by address
* revokeOperator                          Revoke the operator
* revokeSysConfigManager                  Revoke permission for system␣
↪configuration by address
* revokeUserTableManager                  Revoke permission for user table by␣
↪table name and address
* switch([s])                             Switch to a specific group by group ID
* select                                  Select records by sql
* setSystemConfigByKey                    Set a system config value by key
* startGroup                              Start the specified group of the␣
↪specified node
* stopGroup                               Stop the specified group of the␣
↪specified node
* unfreezeAccount                         Unfreeze the account
* unfreezeContract                        Unfreeze the contract
* update                                  Update records by sql
* updateCommitteeMemberWeight             Update the committee member weight
* updateThreshold                         Update the threshold
-------------------------------------------------------------------------------
↪----------
```

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]
```

(continues on next page)

(续上页)

```
* 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]>
```

newAccount

创建新的发送交易的账户，默认会以PEM格式将账户保存在account目录下。

```
# 控制台连接非国密区块链时，账户文件自动保存在`account/ecdsa`目录
# 控制台连接国密区块链时，账户文件自动保存在`accout/gm`目录下
[group:1]> newAccount
AccountPath: account/ecdsa/0x6fad87071f790c3234108f41b76bb99874a6d813.pem
newAccount: 0x6fad87071f790c3234108f41b76bb99874a6d813
AccountType: ecdsa

$ ls -al account/ecdsa/0x6fad87071f790c3234108f41b76bb99874a6d813.pem
$ -rw-r--r--  1 octopus  staff  258  9 30 16:34 account/ecdsa/
↪0x6fad87071f790c3234108f41b76bb99874a6d813.pem
```

loadAccount

加载PEM或者P12格式的私钥文件，加载的私钥可以用于发送交易签名。 参数:

- 私钥文件路径: 支持相对路径、绝对路径和默认路径三种方式。用户账户地址时，默认从config.toml的账户配置选项keyStoreDir加载账户，keyStoreDir配置项请参考这里。
- 账户格式: 可选，加载的账户文件类型，支持pem与p12，默认为pem。

```
[group:1]> loadAccount 0x6fad87071f790c3234108f41b76bb99874a6d813
Load account 0x6fad87071f790c3234108f41b76bb99874a6d813 success!
```

listAccount

查看当前加载的所有账户信息

```
[group:1]> listAccount
0x6fad87071f790c3234108f41b76bb99874a6d813(current account) <=
0x726d9f31cf44debf80b08a7e759fa98b360b0736
```

注意：带有<=后缀标记的为当前用于发送交易的私钥账户，可以使用loadAccount进行切换

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
[
    "0c0bbd25152d40969d3d3cee3431fa28287e07cff2330df3258782d3008b876d146ddab97eab42796495bfbb281591fe
    "10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada75836e
    "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
[
    "41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd43462
    "87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9ea
    "29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5d9
    "d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b75
]
```

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
ConsensusInfo{
    baseConsensusInfo=BasicConsensusInfo{
        nodeNum='4',
        nodeIndex='3',
        maxFaultyNodeNum='1',
        sealerList=[
            ␣
↪11e1be251ca08bb44f36fdeedfaeca40894ff80dfd80084607a75509edeaf2a9c6fee914f1e9efda571611cf4575a15
↪
            ␣
↪78a313b426c3de3267d72b53c044fa9fe70c2a27a00af7fea4a549a7d65210ed90512fc92b6194c14766366d434235c
↪
            ␣
↪95b7ff064f91de76598f90bc059bec1834f0d9eeb0d05e1086d49af1f9c2f321062d011ee8b0df7644bd54c4f9ca3d8
↪
            ␣
↪b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a99584
        ],
        consensusedBlockNumber='1',
        highestblockNumber='0',
        groupId='1',
        protocolId='65544',
        accountType='1',
        cfgErr='false',
        omitEmptyBlock='true',
        nodeId=
↪'b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a9958
↪',
        allowFutureBlocks='true',
        connectedNodes='3',
        currentView='1735',
        toView='1735',
        leaderFailed='false',
        highestblockHash=
↪'0x4f6394763c33c1709e5a72b202ad4d7a3b8152de3dc698cef6f675ecdaf20a3b'
    },
    viewInfos=[
        ViewInfo{
            nodeId=
↪'11e1be251ca08bb44f36fdeedfaeca40894ff80dfd80084607a75509edeaf2a9c6fee914f1e9efda571611cf4575a15
↪',
            view='1732'
        },
        ViewInfo{
            nodeId=
↪'78a313b426c3de3267d72b53c044fa9fe70c2a27a00af7fea4a549a7d65210ed90512fc92b6194c14766366d434235c
↪',
            view='1733'
        },
        ViewInfo{
            nodeId=
↪'95b7ff064f91de76598f90bc059bec1834f0d9eeb0d05e1086d49af1f9c2f321062d011ee8b0df7644bd54c4f9ca3d8
↪',
            view='1734'
        },
        ViewInfo{
            nodeId=
↪'b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a9958
↪',
```

(continues on next page)

(续上页)

```
            view='1735'
        }
    ]
}
```

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
[
    PeerInfo{
        nodeID=
↪'78a313b426c3de3267d72b53c044fa9fe70c2a27a00af7fea4a549a7d65210ed90512fc92b6194c14766366d434235
↪',
        ipAndPort='127.0.0.1:54647',
        agency='agency',
        topic=[

        ],
        node='node2'
    },
    PeerInfo{
        nodeID=
↪'95b7ff064f91de76598f90bc059bec1834f0d9eeb0d05e1086d49af1f9c2f321062d011ee8b0df7644bd54c4f9ca3d
↪',
        ipAndPort='127.0.0.1:30303',
        agency='agency',
        topic=[

        ],
        node='node3'
    },
    PeerInfo{
        nodeID=
↪'b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a9958
↪',
        ipAndPort='127.0.0.1:30300',
        agency='agency',
        topic=[
            _block_notify_1
        ],
        node='node0'
    }
]
```

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
↪
```

(continues on next page)

(续上页)

```
↪ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54b
↪
↪0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪
↪2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
]
```

getGroupList

运行getGroupList，查看群组列表:

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

getBlockHeaderByHash

运行getBlockHeaderByHash，根据区块哈希查询区块头信息。 参数:

- 区块哈希：0x开头的区块哈希值
- 签名列表标志：默认为false，即：区块头信息中不显示区块签名列表信息，设置为true，则显示区块签名列表。

```
[group:1]> getBlockHeaderByHash
↪0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d
{
    "dbHash":"0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)

```
    "stateRoot":"0x0000000000000000000000000000000000000000000000000000000000000000
↪",
    "timestamp":"0x173ad8703d6",
    "transactionsRoot":
↪"0xb563f70188512a085b5607cac0c35480336a566de736c83410a062c9acc785ad"
}
```

getBlockHeaderByNumber

运行getBlockHeaderByNumber，根据区块高度查询区块头信息。 参数：

- 区块高度
- 签名列表标志：默认为false，即：区块头信息中不显示区块签名列表信息，设置为true，则显示区块签名列表。

```
[group:1]> getBlockHeaderByNumber 1 true
{
    "dbHash":"0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)

```
        },
        {
            "index":"0x1",
            "signature":
↪"0xea3c27c2a1486c7942c41c4dc8f15fbf9a668aff2ca40f00701d73fa659a14317d45d74372d69d43ced8e81f789e4
↪"
        }
    ],
    "stateRoot":"0x0000000000000000000000000000000000000000000000000000000000000000
↪",
    "timestamp":"0x173ad8703d6",
    "transactionsRoot":
↪"0xb563f70188512a085b5607cac0c35480336a566de736c83410a062c9acc785ad"
}
```

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":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(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":
↪"0x00000000000
```

```
[group:1]> getTransactionByHash␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
```

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。

```
[group:1]> getTransactionByBlockHashAndIndex␣
↪0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b 0
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
```

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。

```
[group:1]> getTransactionByBlockNumberAndIndex 8 0
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
```

(continues on next page)

(续上页)

```
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "nonce":"0x1aec6e447da49b9a140bf39a91a4d75fd19ea77f7dc38ccf940d8d510d78bd0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionIndex":"0x0",
    "value":"0x0"
}
```

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。

```
[group:1]> getTransactionReceipt␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gasUsed":"0x94f5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "logs":[
        {
            "address":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0xc57b01fa77f41df77eaab79a0e2623fab2e7ae3e9530d9b1cab225ad65f2b7ce
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x0000000000000080000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0x42fc572759fd568bd590f46011784be2a2d53f0c",
    "transactionHash":
↪"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "transactionIndex":"0x0"
}
```

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

部署合约。(默认提供HelloWorld合约和TableTest.sol进行示例使用) 参数：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`，比如：HelloWorld。

```
# 部署HelloWorld合约，默认路径
[group:1]> deploy HelloWorld
contract address:0xc0ce097a5757e2b6e189aa70c7d55770ace47767

# 部署HelloWorld合约，相对路径
[group:1]> deploy contracts/solidity/HelloWorld.sol
contract address:0xd653139b9abffc3fe07573e7bacdfd35210b5576

# 部署HelloWorld合约，绝对路径
[group:1]> deploy ~/fisco/console/contracts/solidity/HelloWorld.sol
contract address:0x85517d3070309a89357c829e4b9e2d23ee01d881
```

注：

- 部署用户编写的合约，可以将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。

listAbi

显示合约接口和Event列表 参数：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`，比如：TableTest。
- 合约名：(可选)合约名称，默认情况下使用合约文件名作为合约名参数

```
[group:1]> listAbi TableTest
Method list:
 name                 |    constant   |    methodId    |    signature
  ------------------------------------------------------------------
 remove               |    false      |    0x0fe1160f  |    remove(string,int256)
 update               |    false      |    0x49cc36b5  |    update(string,int256,
↪string)
 select               |    true       |    0x5b325d78  |    select(string)
 insert               |    false      |    0xe020d464  |    insert(string,int256,
↪string)

Event list:
 name                 |   topic                                                 ␣
↪              signature
  ------------------------------------------------------------------
 remove               |  ␣
↪0x0fe1160f9655e87c29e76aca1cab34fb2a644d375da7a900c7076bad17cad26b  |  ␣
↪remove(string,int256)
 update               |  ␣
↪0x49cc36b56a9320d20b2d9a1938a972c849191bceb97500bfd38fa8a590dac73a  |  ␣
↪update(string,int256,string)
 select               |  ␣
↪0x5b325d7821528d3b52d0cc7a83e1ecef0438f763796770201020ac8b8813ac0a  |  ␣
↪select(string)
 insert               |  ␣
↪0xe020d464e502c11b54a7e37e568c78f0fcd360213eb5f4ac0a25a17733fc19f7  |  ␣
↪insert(string,int256,string)
```

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：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 合约地址: 部署合约获取的地址。
- 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；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 0x175b16a1299c7af3e2e49b97e68a44734257a35e get
----------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
----------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,World!"
]
----------------------------------------------------------------------------------------
↪----------

# 调用HelloWorld的set接口设置name字符串
[group:1]> call HelloWorld 0x175b16a1299c7af3e2e49b97e68a44734257a35e set "Hello,␣
↪FISCO BCOS"
transaction hash:␣
↪0x54b7bc73e3b57f684a6b49d2fad41bd8decac55ce021d24a1f298269e56f1ce1
----------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully
----------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
----------------------------------------------------------------------------------------
↪----------
Event logs
Event: {}

# 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 0x175b16a1299c7af3e2e49b97e68a44734257a35e get
----------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
----------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,FISCO BCOS"
]
```

(continues on next page)

(续上页)

```
-------------------------------------------------------------------------------------------
↪----------


# 调用TableTest的insert接口插入记录，字段为name, item_id, item_name
[group:1]> call TableTest 0x5f248ad7e917cddc5a4d408cf18169d19c0990e5 insert "fruit
↪" 1 "apple"
transaction hash:␣
↪0x64bfab495dc1f50c58d219b331df5a47577aa8afc16be926260238a9b0ec0fbb
-------------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully
-------------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Event logs
Event: {"InsertResult":[1]}

# 调用TableTest的select接口查询记录
[group:1]> [group:1]> call TableTest 0x5f248ad7e917cddc5a4d408cf18169d19c0990e5␣
↪select "fruit"
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    [
        "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：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 合约版本号：部署的合约版本号。

```
# 部署HelloWorld合约1.0版
[group:1]> deployByCNS HelloWorld 1.0
transaction hash:␣
↪0xb994d8e510f147815bdf9838fda542e553c2fe981177ee7a97a0686b9619cfbb
contract address: 0xf485b9ccfffa668f4d7fac37c81c0cd63408188c
{
    "code":1,
    "msg":"Success"
}

# 部署HelloWorld合约2.0版
[group:1]> deployByCNS HelloWorld 2.0
transaction hash:␣
↪0x3132f73e5af72fce44c2e07185a04f554ac06ddd94119dcf798980695b0890a0
contract address: 0xf68a1aabfad336847e109c33ca471b192c93c0a9
{
    "code":1,
    "msg":"Success"
}

# 部署TableTest合约
[group:1]> deployByCNS TableTest 1.0
transaction hash:␣
↪0x288ccc6e166e43f5cd3bd563e00af48988e641196aaea57a59f0ab256e23c84e
contract address: 0x0fe221339e50c39aaefddfc3a9a26b4aeff23c63
{
    "code":1,
    "msg":"Success"
}
```

注：

- 部署用户编写的合约，可以将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。

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
--------------------------------------------------------------------------------
↪----------
|                    version                    |                    address          ␣
↪          |
|                      1.0                      |␣
↪0xf485b9ccfffa668f4d7fac37c81c0cd63408188c  |
|                      2.0                      |␣
↪0xf68a1aabfad336847e109c33ca471b192c93c0a9  |
--------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
[group:1]> queryCNS HelloWorld 1.0
--------------------------------------------------------------------------------------
↪----------
|                    version                    |                   address
↪          |
|                      1.0                      |
↪0xf485b9ccfffa668f4d7fac37c81c0cd63408188c  |
--------------------------------------------------------------------------------------
↪----------
```

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。当省略合约版本号时，例如HelloWorld，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:
↪0x7ad2f34a13bbc2272d2d52fa46915e6f03136a6960d84a23478bee3c37e76ad6
--------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully
--------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
--------------------------------------------------------------------------------------
↪----------
Event logs
Event: {}

# 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:
↪0x41f8decbe137905824321da6c28a19d957e902cfad1e8078d797bc649c078d3e
--------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully
--------------------------------------------------------------------------------------
↪----------
Output
Receipt message: Success
Return message: Success
--------------------------------------------------------------------------------------
↪----------
Event logs
Event: {}

# 调用HelloWorld合约1.0版，通过get接口获取name字符串
```

(continues on next page)

(续上页)

```
[group:1]> callByCNS HelloWorld:1.0 get
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,CNS"
]
-------------------------------------------------------------------------------------------
↪----------

# 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
-------------------------------------------------------------------------------------------
↪----------
Return code: 0
description: transaction executed successfully
Return message: Success
-------------------------------------------------------------------------------------------
↪----------
Return values:
[
    "Hello,CNS2"
]
-------------------------------------------------------------------------------------------
↪----------
```

registerCNS

注册合约至CNS。

参数:

- 合约路径: 合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 合约地址: 注册合约地址
- 合约版本号: 注册合约版本号

```
[group:1]> registerCNS HelloWorld 0xf19a7ec01f0b1adb16a033f0a30fb321ec6edcbf v1.0.0
{
    "code":1,
    "msg":"success"
}
```

addSealer

运行addSealer，将节点添加为共识节点。 参数:

- 节点nodeId

```
[group:1]> addSealer␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
```

(continues on next page)

(续上页)

```
        "code":1,
        "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":1,
        "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":1,
        "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、rpbft_epoch_block_num和consensus_tim
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
- consensus_timeout：PBFT共识过程中，每个区块执行的超时时间，默认为3s，单位为秒

参数:

- key
- value

```
[group:1]> setSystemConfigByKey tx_count_limit 100
{
        "code":1,
        "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
```

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":1,
        "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":1,
        "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":1,
        "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":1,
        "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":1,
        "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
↪          |
```

(continues on next page)

(续上页)

```
| 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":1,
        "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":1,
        "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":1,
        "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":1,
        "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":1,
        "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":1,
        "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
↪          |
| 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":1,
        "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":1,
    "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":1,
    "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":1,
    "msg":"success"
}
```

revokeContractStatusManager

运行revokeContractStatusManager，用于撤销指定权限账号对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]> revokeContractStatusManager 0x30d2a17b6819f0d77f26dd3a9711ae75c291f7f1␣
↪0x965ebffc38b309fa706b809017f360d4f6de909a
{
    "code":1,
    "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":1,
    "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":1,
    "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":1,
    "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":1,
    "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":1,
    "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":1,
    "msg":"success"
}
```

listOperators

查询有运维权限的账号。

```
[group:1]> listOperators
--------------------------------------------------------------------------------------
↪----------
|                    address                     |                 enable_num
↪          |
```

(continues on next page)

(续上页)

```
| 0x283f5b859e34f7fd2cf136c07579dcc72423b1b2  |                          1
↪          |
--------------------------------------------------------------------------------
↪----------
```

queryVotesOfThreshold

查询updateThreshold的投票情况:

```
[group:1]> queryVotesOfThreshold
The votes of the updateThreshold operation : {"0.100000":[{"block_limit":"10002",
↪"origin":"0x2eb1be0f52c0d00f9594a021240ea7fb027d7485"}]}
```

queryVotesOfMember

查询指定账户被选举为委员的投票情况，若没有任何委员投票，则返回null:

- 账户地址: 被查询的账户地址

```
[group:1]> queryVotesOfMember 0xc398d318662aa19487c405a45267ecd60115adec
queried account: 0xc398d318662aa19487c405a45267ecd60115adec
votes:{"grant":[{"block_limit":"10003","origin":
↪"0x2eb1be0f52c0d00f9594a021240ea7fb027d7485"}]}
```

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":1,
    "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":1,
    "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.
```

listDeployContractAddress

列出指定合约名部署的所有合约地址， 列出部署指定合约产生的合约地址列表，参数：

- contractNameOrPath: 合约名或合约绝对路径，用于指定合约;
- recordNumber: 显示的合约地址列表长度，默认为20

```
# 获取部署HelloWorld合约产生的合约地址列表
[group:1]> listDeployContractAddress HelloWorld
0xe157185434183b276b9e5af7d0315a9829171281  2020-10-13 15:35:29
0x136b042e1fc480b03e1e5b075cbdfa52f5851a23  2020-10-13 15:35:22
0xd7d0b221bc984a20aa6b0fc098dad89888378e3a  2020-10-13 15:34:14
0x0fe221339e50c39aaefddfc3a9a26b4aeff23c63  2020-10-13 15:16:20
0x5f248ad7e917cddc5a4d408cf18169d19c0990e5  2020-10-13 12:28:23
0xf027fd91a51bd4844f17600c01e943058fc27482  2020-10-12 18:28:50
0x682b51f3c7f9a605fa2026b09fd2635a6fa562d9  2020-10-11 23:22:28
0xf68a1aabfad336847e109c33ca471b192c93c0a9  2020-10-11 22:53:07
0xf485b9ccfffa668f4d7fac37c81c0cd63408188c  2020-10-11 22:52:26
0x175b16a1299c7af3e2e49b97e68a44734257a35e  2020-10-11 22:49:25
0x7c6dc94e4e146cb13eb03dc98d2b96ac79ef5e67  2020-10-11 22:46:35


# 限制显示的合约地址列表长度为2
[group:1]> listDeployContractAddress HelloWorld 2
0xe157185434183b276b9e5af7d0315a9829171281  2020-10-13 15:35:29
0x136b042e1fc480b03e1e5b075cbdfa52f5851a23  2020-10-13 15:35:22
```

getCurrentAccount

获取当前账户地址。

```
[group:1]> getCurrentAccount
0x6fad87071f790c3234108f41b76bb99874a6d813
```

getCryptoType

获取当前控制台所连接的节点账本类型、SSL协议类型。

注解:

- 非国密账本类型为 ECDSA ， 国密类型账本类型为 SM
- OpenSSL协议类型为 ECDSA ， 国密SSL协议类型为 SM

```
[group:1]> getCryptoType
ledger crypto type: ECDSA
ssl crypto type: ECDSA
```

getAvailableConnections

获取SDK连接的节点连接信息。

```
[group:1]> getAvailableConnections
[
    127.0.0.1:20200,
    127.0.0.1:20201
]
```

getGroupConnections

从SDK连接的节点列表中，筛选出启动控制台当前登录群组的节点列表信息。

```
[group:1]> getGroupConnections
[
    127.0.0.1:20200,
    127.0.0.1:20201
]
```

generateGroup

为指定节点动态创建一个新群组，参数：

- endPoint: 接收创建新群组请求的区块链节点IP:Port信息，SDK连接的所有节点IP:Port信息可通过命令getAvailableConnections获取；
- groupId: 新创建的群组ID；
- timestamp: 新创建群组的创世块时间戳，可通过命令echo $(($(date '+%s')*1000))获取时间戳；
- sealerList: 新创建群组的共识节点列表，多个共识节点ID之间用空格分割。

为监听在本机20200端口的区块链节点创建新群组2的示例如下：

```
# 获取时间戳
$ echo $(($(date '+%s')*1000))
1590586645000
```

```
[group:1]> generateGroup  127.0.0.1:20200 2 1590586645000␣
↪b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a99584
GroupStatus{
    code='0x1',
    message='Group 2 generated successfully',
    status='null'
}
```

generateGroupFromFile

通过新群组配置文件为指定节点列表创建新群组，配置文件指定了需要创建群组节点列表、新群组的共识列表以及创世块时间戳，群组配置示例group-generate-config.toml如下：

```
# The peers to generate the group
[groupPeers]
peers=["127.0.0.1:20200", "127.0.0.1:20201"]

# The consensus configuration of the generated group
```

(continues on next page)

(续上页)

```
[consensus]
# The sealerList
sealerList=[
↪"b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a9958
↪",
↪"11e1be251ca08bb44f36fdeedfaeca40894ff80dfd80084607a75509edeaf2a9c6fee914f1e9efda571611cf4575a1
↪"]

[genesis]
# The genesis timestamp, It is recommended to set to the current utcTime, which␣
↪must be greater than 0
timestamp = "1590586645000"
```

generateGroupFromFile命令的参数包括：

- groupConfigFilePath：群组配置文件路径，控制台conf/group-generate-config.toml是提供的群组配置文件模板，用户可根据实际场景拷贝、修改该配置模板，并加载修改后的群组配置文件；
- groupId：新创建的群组ID。

```
[group:1]> generateGroupFromFile conf/group-generate-config.toml 3
* Result of 127.0.0.1:20200:
GroupStatus{
    code='0x0',
    message='Group 3 generated successfully',
    status='null'
}
* Result of 127.0.0.1:20201:
GroupStatus{
    code='0x0',
    message='Group 3 generated successfully',
    status='null'
}
```

startGroup

为指定节点启动群组，参数：

- endPoint：接收启动群组请求的区块链节点IP:Port信息，SDK连接的所有节点IP:Port信息可通过命令getAvailableConnections获取；
- groupId：启动的群组ID。

启动群组2的控制台命令示例如下：

```
# 获取127.0.0.1:20200当前群组列表
[group:1]> getGroupList 127.0.0.1:20200
[1]
[group:1]> startGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x0',
    message='Group 2 started successfully',
    status='null'
}
# 127.0.0.1:20200的群组2启动成功后，群组列表中新增群组2
[group:1]> getGroupList 127.0.0.1:20200
[1, 2]
```

stopGroup

为指定节点停止群组，参数：

- endPoint: 接收停止群组请求的区块链节点IP:Port信息，SDK连接的所有节点IP:Port信息可通过命令getAvailableConnections获取；
- groupId：停止的群组ID。

停止群组2的控制台命令示例如下：

```
# 获取127.0.0.1:20200当前群组列表
[group:1]> getGroupList 127.0.0.1:20200
[1, 2]
[group:1]> stopGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x0',
    message='Group 2 stopped successfully',
    status='null'
}
# 127.0.0.1:20200的群组2停止成功后，群组列表中群组2被移出
[group:1]> getGroupList 127.0.0.1:20200
[1]
```

removeGroup

为指定节点删除群组，参数：

- endPoint: 接收删除群组请求的区块链节点IP:Port信息，SDK连接的所有节点IP:Port信息可通过命令getAvailableConnections获取；
- groupId：删除的群组ID。

删除群组2的控制台命令示例如下：

```
# 删除127.0.0.1:20200的群组2
[group:1]> removeGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x0',
    message='Group 2 deleted successfully',
    status='null'
}
# 删除127.0.0.1:20200的群组2后，尝试启动被删除的群组，启动失败
[group:1]> startGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x5',
    message='Group 2 has been deleted',
    status='null'
}
[group:1]> getGroupList 127.0.0.1:20200
[1]
```

recoverGroup

为指定节点恢复指定群组，参数：

- endPoint: 接收群组恢复请求的区块链节点IP:Port信息，SDK连接的所有节点IP:Port信息可通过命令getAvailableConnections获取；
- groupId：恢复的群组ID。

```
# 获取127.0.0.1:20200的当前群组列表
[group:1]> getGroupList 127.0.0.1:20200
[1]
# 恢复127.0.0.1:20200的群组2
[group:1]> recoverGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x0',
    message='Group 2 recovered successfully',
    status='null'
}
# 启动127.0.0.1:20200的群组2
[group:1]> startGroup 127.0.0.1:20200 2
GroupStatus{
    code='0x0',
    message='Group 2 started successfully',
    status='null'
}
# 获取127.0.0.1:20200的当前群组列表, 新增了群组2
[group:1]> getGroupList 127.0.0.1:20200
[1, 2]
```

getBatchReceiptsByBlockNumberAndRange

指定块高和交易回执范围，获取指定区块高度、指定范围的交易回执:

- `blockNumber`: 请求获取的交易回执所在的区块高度
- `from`: 请求的批量回执的起始索引，可不填，默认值为0
- `count`: 请求的批量回执数目，可不填，默认值为-1，设置为-1时，返回区块内所有交易回执

```
[group:1]> getBatchReceiptsByBlockNumberAndRange 1
TransactionReceiptsInfo{
    blockInfo=BlockInfo{
        receiptRoot=
↪'0x67182babfe1500a8ec442a8b9548e7d0d912af4943c3d549bdf9ed0c76fe8c11',
        blockNumber='0x1',
        blockHash=
↪'0x5eb495f6fa457dbcaf6323630a257a65a7085e01087421b7191d8efec69da0c0',
        receiptsCount='0x1'
    },
    transactionReceipts=[
        TransactionReceipt{
            transactionHash=
↪'0xa3ce50e3f03d3282e21248172efd1345b9eb15b281791b499f2e6c7bbe464667',
            transactionIndex='0x0',
            root='null',
            blockNumber='null',
            blockHash='null',
            from='0x2eb1be0f52c0d00f9594a021240ea7fb027d7485',
            to='0x0000000000000000000000000000000000001008',
            gasUsed='0x589b',
            contractAddress='0x0000000000000000000000000000000000000000',
            logs=[

            ],
            logsBloom='null',
            status='0x0',
            statusMsg='null',
            input='null',
            output=
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
```

(continues on next page)

(续上页)

```
            txProof=null,
            receiptProof=null
        }
    ]
}
```

getBatchReceiptsByBlockHashAndRange

指定区块哈希和交易回执范围，获取指定区块哈希、指定范围的交易回执:

- blockHash: 请求获取的交易回执所在的区块哈希
- from: 请求的批量回执的起始索引，可不填，默认值为0
- count: 请求的批量回执数目，可不填，默认值为-1，设置为-1时，返回区块内所有交易回执

```
[group:1]> getBatchReceiptsByBlockHashAndRange␣
↪0x5eb495f6fa457dbcaf6323630a257a65a7085e01087421b7191d8efec69da0c0
TransactionReceiptsInfo{
    blockInfo=BlockInfo{
        receiptRoot=
↪'0x67182babfe1500a8ec442a8b9548e7d0d912af4943c3d549bdf9ed0c76fe8c11',
        blockNumber='0x1',
        blockHash=
↪'0x5eb495f6fa457dbcaf6323630a257a65a7085e01087421b7191d8efec69da0c0',
        receiptsCount='0x1'
    },
    transactionReceipts=[
        TransactionReceipt{
            transactionHash=
↪'0xa3ce50e3f03d3282e21248172efd1345b9eb15b281791b499f2e6c7bbe464667',
            transactionIndex='0x0',
            root='null',
            blockNumber='null',
            blockHash='null',
            from='0x2eb1be0f52c0d00f9594a021240ea7fb027d7485',
            to='0x0000000000000000000000000000000000001008',
            gasUsed='0x589b',
            contractAddress='0x0000000000000000000000000000000000000000',
            logs=[

            ],
            logsBloom='null',
            status='0x0',
            statusMsg='null',
            input='null',
            output=
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
            txProof=null,
            receiptProof=null
        }
    ]
}
```

11.3 控制台1.x版本

标签：控制台1

重要:

- 控制台2.6+ 基于 Java SDK 实现，控制台1.x 系列基于 Web3SDK 实现，本教程针对 **1.x版本控制台**，2.6及其以上版本控制台使用文档请 参考这里
- 可通过命令 ./start.sh --version 查看当前控制台版本
- 基于 Web3SDK 开发应用过程中将 solidity 代码转换为 java 代码时，必须使用 1.x 版本控制台，具体请参考 这里

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。

11.3.1 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/v2.7.1/download_
↪console.sh && bash download_console.sh -c 1.2.0
```

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 -c 1.2.0*

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
|-- get_gm_account.sh # 账户生成脚本，国密版
|-- sol2java.sh # solidity合约文件编译为java合约文件的开发工具脚本
```

注意：默认下载的控制台内置0.4.25版本的solidity编译器，用户需要编译0.5或者0.6版本的合约时，可以通过下列命令获取内置对应编译器版本的控制台

```
# 0.5
curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v1.1.0/download_
↪console.sh && bash download_console.sh -v 0.5
# 0.6
curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v1.1.0/download_
↪console.sh && bash download_console.sh -v 0.6
```

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。**
  - FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建链报错。

```
<?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>
```

(continues on next page)

(续上页)

```
        <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。**
  - FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建链报错。
- 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>
```

重要： 控制台编译工具重要说明

- 控制台自1.1.0版本起，移除对solcJ-all-0.x.x.jar、solcJ-all-0.x.x-gm.jar的依赖，控制台使用新的合约编译工具，新编译工具上传至maven仓库进行管理，不再需要进行替换文件操作
- 新编译工具支持0.4.25、0.5.2、0.6.10三个版本，与同版本的solidity编译器对应
- 控制台默认配置0.4.25版本编译工具，用户可以修改build.gradle配置的版本号重新编译，也可以通过download_console.sh脚本指定-v参数，下载配置对应编译器版本的控制台
- 新的编译工具同时支持国密、非国密编译功能，控制台国密或者非国密环境运行时，不再需要solcJ国密与非国密版本的替换

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。

```
$ bash sol2java.sh -h
# Compile Solidity Tool
./sol2java.sh [packageName] [solidityFilePath] [javaCodeOutputDir]
          packageName:
                  the package name of the generated Java class file
          solidityFilePath:
                  (optional) the solidity file path or the directory where␣
↪solidity files located, default: contracts/solidity
          javaCodeOutputDir:
                  (optional) the directory where the generated Java files located,␣
↪default: contracts/sdk/java
```

参数

- packageName: 生成Java文件的包名
- solidityFilePath: (可选)solidity文件的路径，支持文件路径和目录路径两种方式，参数为目录时将目录下所有的solidity文件进行编译转换。默认目录为contracts/solidity。
- javaCodeOutputDir: (可选)生成Java文件的目录，默认生成在contracts/sdk/java目录。

使用

```
$ cd ~/fisco/console
$ ./sol2java.sh org.com.fisco # 指定java包名
```

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。

重要： Java合约文件说明

- 控制台自1.1.0版本起，生成的Java合约文件国密、非国密环境均可以运行，国密与非国密环境生成一份合约代码即可

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
================================================================================================
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: 1.2.0
```

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

使用控制台配置文件指定的默认群组号启动。

```
./start.sh
```

注意: 控制台启动未指定私钥账户时，会尝试从accounts目录下加载一个可用的私钥账户用于发送交易，加载失败则会创建一个新的PEM格式的账户文件，将其保存在accounts目录下。

指定群组号启动

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

控制台命令结构

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。

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

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

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。

11.3.2 控制台命令列表

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.
getBlockHeaderByHash                    Query information about a block header by␣
↪hash.
getBlockHeaderByNumber                  Query information about a block header by␣
↪block number.
getBlockByHash                          Query information about a block by hash.
getBlockByNumber                        Query information about a block by block␣
↪number.
```

(continues on next page)

(续上页)

```
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.
getNodeInfo                             Query the specified node information.
getObserverList                         Query nodeId list for observer nodes.
getPbftView                             Query the pbft view of node.
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.
```

(continues on next page)

(续上页)

```
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.
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。

newAccount

创建新的发送交易的账户，默认会以PEM格式将账户保存在accounts目录下。

```
[group:1]> newAccount
 new account successfully, account␣
↪address:0x4cb7d6c013d9c7fa4ec75a3df3d0fddf39674c14

# 私钥文件自动保存在accounts目录下
$ ls -al accounts/0x4cb7d6c013d9c7fa4ec75a3df3d0fddf39674c14.pem
$ -rw-r--r--  1 octopus  staff  258  9 30 16:34 accounts/
↪0x4cb7d6c013d9c7fa4ec75a3df3d0fddf39674c14.pem
```

loadAccount

加载PEM或者P12格式的私钥文件，加载的私钥可以用于发送交易签名。 参数:

- 私钥文件路径: 支持相对路径、绝对路径和默认路径三种方式。用户输入文件名时，会从默认目录获取文件，默认目录为: accounts。
- 账户密码: (可选)P12私钥文件的密码。

```
[group:1]> loadAccount 0xa0f749a6eb735d578b81239c1661c726c4f05d0e.pem
 load 0xa0f749a6eb735d578b81239c1661c726c4f05d0e.pem successfully, account␣
↪address: 0xa0f749a6eb735d578b81239c1661c726c4f05d0e
```

注意：加载的私钥需要使用switchAccount才可以用与发送交易，也可以使用listAccount查看当前加载的所有私钥

switchAccount

切换发送交易的私钥账户。 参数:

- 账户地址

```
[group:1]> switchAccount 0xa0f749a6eb735d578b81239c1661c726c4f05d0e
switch to account: 0xa0f749a6eb735d578b81239c1661c726c4f05d0e successfully.
```

listAccount

查看当前加载的所有账户信息

```
[group:1]> listAccount
 account list:
         0xa0f749a6eb735d578b81239c1661c726c4f05d0e <=
         0x4cb7d6c013d9c7fa4ec75a3df3d0fddf39674c14
```

注意：带有<=后缀标记的为当前用于发送交易的私钥账户，可以使用switchAccount进行切换

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":
↪"8718579e9a6fee647b3d7404d59d66749862aeddef22e6b5abaafe1af6fc128fc33ed5a9a105abddab51e12004c6bfe
↪",
                "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)

```
↪0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca8
↪
↪2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a0
]
```

getGroupList

运行getGroupList，查看群组列表:

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

getBlockHeaderByHash

运行getBlockHeaderByHash，根据区块哈希查询区块头信息。 参数:

- 区块哈希：0x开头的区块哈希值
- 签名列表标志：默认为false，即：区块头信息中不显示区块签名列表信息，设置为true，则显示区块签名列表。

```
[group:1]> getBlockHeaderByHash␣
↪0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d
{
    "dbHash":"0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)

```
    "transactionsRoot":
↪"0xb563f70188512a085b5607cac0c35480336a566de736c83410a062c9acc785ad"
}
```

getBlockHeaderByNumber

运行getBlockHeaderByNumber，根据区块高度查询区块头信息。 参数：

- 区块高度
- 签名列表标志：默认为false，即：区块头信息中不显示区块签名列表信息，设置为true，则显示区块签名列表。

```
[group:1]> getBlockHeaderByNumber 1 true
{
    "dbHash":"0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":1,
    "parentHash":
↪"0x4f6394763c33c1709e5a72b202ad4d7a3b8152de3dc698cef6f675ecdaf20a3b",
    "receiptsRoot":
↪"0x69a04fa6073e4fc0947bac7ee6990e788d1e2c5ec0fe6c2436d0892e7f3c09d2",
    "sealer":"0x2",
    "sealerList":[

↪"11e1be251ca08bb44f36fdeedfaeca40894ff80dfd80084607a75509edeaf2a9c6fee914f1e9efda571611cf4575a15
↪",

↪"78a313b426c3de3267d72b53c044fa9fe70c2a27a00af7fea4a549a7d65210ed90512fc92b6194c14766366d434235
↪",

↪"95b7ff064f91de76598f90bc059bec1834f0d9eeb0d05e1086d49af1f9c2f321062d011ee8b0df7644bd54c4f9ca3d8
↪",

↪"b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a99584
↪"
    ],
    "signatureList":[
        {
            "index":"0x3",
            "signature":
↪"0xb5b41e49c0b2bf758322ecb5c86dc3a3a0f9b98891b5bbf50c8613a241f05f595ce40d0bb212b6faa32e985467548
↪"
        },
        {
            "index":"0x0",
            "signature":
↪"0x411cb93f816549eba82c3bf8c03fa637036dcdee65667b541d0da06a6eaea80d16e6ca52bf1b08f77b59a834bffb
↪"
        },
        {
            "index":"0x1",
```

(continues on next page)

(续上页)

```
            "signature":
↪"0xea3c27c2a1486c7942c41c4dc8f15fbf9a668aff2ca40f00701d73fa659a14317d45d74372d69d43ced8e81f789e
↪"
        }
    ],
    "stateRoot":"0x0000000000000000000000000000000000000000000000000000000000000000
↪",
    "timestamp":"0x173ad8703d6",
    "transactionsRoot":
↪"0xb563f70188512a085b5607cac0c35480336a566de736c83410a062c9acc785ad"
}
```

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":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)


```
{
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bc
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        {
            "blockHash":
↪"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
            "blockNumber":"0x1",
            "from":"0x7234c32327795e4e612164e3442cfae0d445b9ad",
            "gas":"0x1c9c380",
            "gasPrice":"0x1",
            "hash":
↪"0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d",
            "input":
↪"0x60806040523480156100105760080fd5b506040805190810160405280600d81526020017f48656c6c6f2c20576f
↪",
            "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":
↪"0x00000000000
```

```
[group:1]> getTransactionByHash␣
↪0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "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":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000000
↪",
    "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
{
    "blockHash":"0xe4e1293837013f547ad7f443a8ff20a4e32a060b9cac56c41462255603548b7b
↪",
```

(continues on next page)

(续上页)

```
    "blockNumber":"0x8",
    "from":"0xf0d2115e52b0533e367447f700bfbf2ed35ff6fc",
    "gas":"0x11e1a300",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
    "input":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "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":
↪"0xebf3b24f0000000000000000000000000000000000000000000000000000000000000060000000000000000000000
↪",
    "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",
    "gasPrice":"0x11e1a300",
    "hash":"0x1dfc67c51f5cc93b033fc80e5e9feb049c575a58b863483aa4d04f530a2c87d5",
```

(continues on next page)

(续上页)

```
    "input":
↪"0xebf3b24f0000000000
```

(续上页)

```
            "topics":[
                "0xc57b01fa77f41df77eaab79a0e2623fab2e7ae3e9530d9b1cab225ad65f2b7ce
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x00000000000000800000000000000
```

(续上页)

```
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

部署合约。(默认提供HelloWorld合约和TableTest.sol进行示例使用) 参数:

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`，比如：HelloWorld。

```
# 部署HelloWorld合约，默认路径
[group:1]> deploy HelloWorld.sol
contract address:0xc0ce097a5757e2b6e189aa70c7d55770ace47767

# 部署HelloWorld合约，相对路径
[group:1]> deploy contracts/solidity/HelloWorld.sol
contract address:0xd653139b9abffc3fe07573e7bacdfd35210b5576

# 部署HelloWorld合约，绝对路径
[group:1]> deploy /root/fisco/console/contracts/solidity/HelloWorld.sol
contract address:0x85517d3070309a89357c829e4b9e2d23ee01d881
```

注：

- 部署用户编写的合约，可以将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。

listAbi

显示合约接口和Event列表 参数：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`，比如：TableTest。
- 合约名：(可选)合约名称，默认情况下使用合约文件名作为合约名参数

```
[group:1]> listAbi TableTest
Method list:
 name                |    constant  |    methodId    |    signature
  ---------------------------------------------------------------
 remove              |    false     |    0x0fe1160f  |    remove(string,int256)
 update              |    false     |    0x49cc36b5  |    update(string,int256,
↪string)
 select              |    true      |    0x5b325d78  |    select(string)
 insert              |    false     |    0xe020d464  |    insert(string,int256,
↪string)

Event list:
 name                |   topic                                                    ␣
↪              signature
  ---------------------------------------------------------------
 remove              |  ␣
↪0x0fe1160f9655e87c29e76aca1cab34fb2a644d375da7a900c7076bad17cad26b  |  ␣
↪remove(string,int256)
 update              |  ␣
↪0x49cc36b56a9320d20b2d9a1938a972c849191bceb97500bfd38fa8a590dac73a  |  ␣
↪update(string,int256,string)
 select              |  ␣
↪0x5b325d7821528d3b52d0cc7a83e1ecef0438f763796770201020ac8b8813ac0a  |  ␣
↪select(string)
 insert              |  ␣
↪0xe020d464e502c11b54a7e37e568c78f0fcd360213eb5f4ac0a25a17733fc19f7  |  ␣
↪insert(string,int256,string)
```

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：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 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!

# 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
----------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully

# 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
----------------------------------------------------------------------------------
↪----------
transaction status: 0x0
```

(continues on next page)

(续上页)

```
description: transaction executed successfully
---------------------------------------------------------------------------------------
↪----------

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：

- 合约路径：合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 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
---------------------------------------------------------------------------------------
↪----------
register contract to cns:
{
    "code":0,
    "msg":"success"
}
# 部署HelloWorld合约2.0版
[group:1]> deployByCNS HelloWorld.sol 2.0
contract address:0x07625453fb4a6459cbf14f5aa4d574cae0f17d92
---------------------------------------------------------------------------------------
↪----------
register contract to cns:
{
    "code":0,
    "msg":"success"
}
# 部署TableTest合约
[group:1]> deployByCNS TableTest.sol 1.0
contract address:0x0b33d383e8e93c7c8083963a4ac4a58b214684a8
---------------------------------------------------------------------------------------
↪----------
register contract to cns:
{
    "code":0,
```

(continues on next page)

(续上页)

```
    "msg":"success"
}
```

注：

- 部署用户编写的合约，可以将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。

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。当省略合约版本号时，例如HelloWorld，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
----------------------------------------------------------------------------------------
↪----------
```

(continues on next page)

(续上页)

```
transaction status: 0x0
description: transaction executed successfully

# 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
-------------------------------------------------------------------------------------
↪----------
transaction status: 0x0
description: transaction executed successfully

# 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

# 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
```

registerCNS

注册合约至CNS。

参数:

- 合约路径: 合约文件的路径，支持相对路径、绝对路径和默认路径三种方式。用户输入为文件名时，从默认目录获取文件，默认目录为: `contracts/solidity`。
- 合约地址: 注册合约地址
- 合约版本号: 注册合约版本号(长度不能超过40)

```
[group:1]> registerCNS HelloWorld 0xf19a7ec01f0b1adb16a033f0a30fb321ec6edcbf v1.0.0
{
    "code":0,
    "msg":"success"
}
```

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, rpbft_epoch_block_num和consensus_timeout。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
- consensus_timeout：PBFT共识过程中，每个区块执行的超时时间，默认为3s，单位为秒

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          ␣
↪          |
| 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
```

(continues on next page)

(续上页)

```
{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.
```

11.3.3 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
```

CHAPTER 12

WeBASE管理平台

标签：WeBASE 中间件平台 节点管理 系统监控 系统管理

微众银行开源的自研区块链中间件平台——WeBASE(WeBank Blockchain Application Software Extension)是区块链应用和FISCO BCOS节点之间搭建的中间件平台。WeBASE屏蔽了区块链底层的复杂度，降低区块链使用的门槛，大幅提高区块链应用的开发效率，包含节点前置、节点管理、交易链路，数据导出，Web管理平台等子系统。用户可以根据业务所需，选择子系统进行部署，可以进一步体验丰富交互的体验、可视化智能合约开发环境IDE。

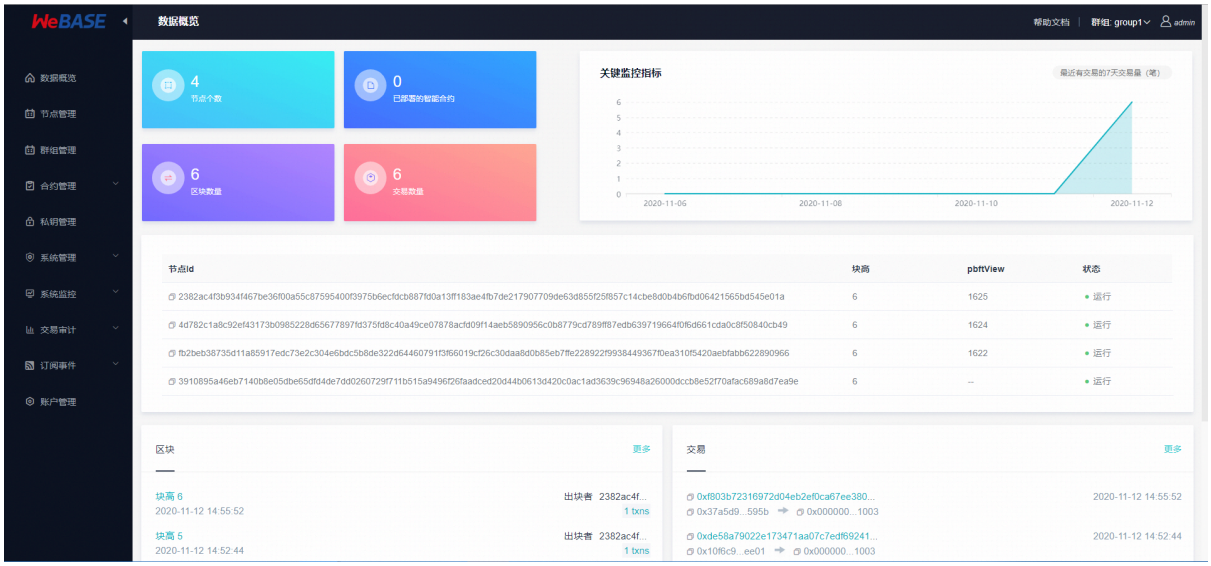
WeBASE管理平台是由四个WeBASE子系统组成的一套管理FISCO-BCOS联盟链的工具集。详细了解可以参考《WeBASE管理平台使用手册》 。

12.1 主要功能

1. 区块链数据概览
2. 节点管理
3. 合约管理
4. 私钥管理
5. 系统管理
6. 系统监控
7. 交易审计
8. 订阅事件
9. 账号管理
10. 群组管理

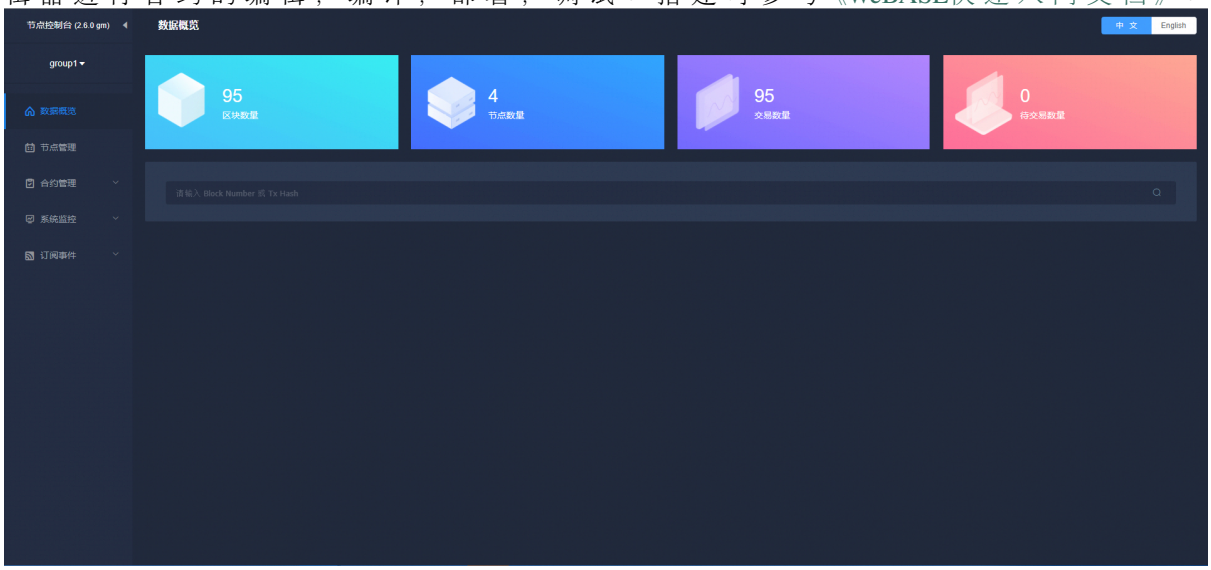
12.2 WeBASE管理平台搭建

搭建可参考《WeBASE一键部署文档》 。



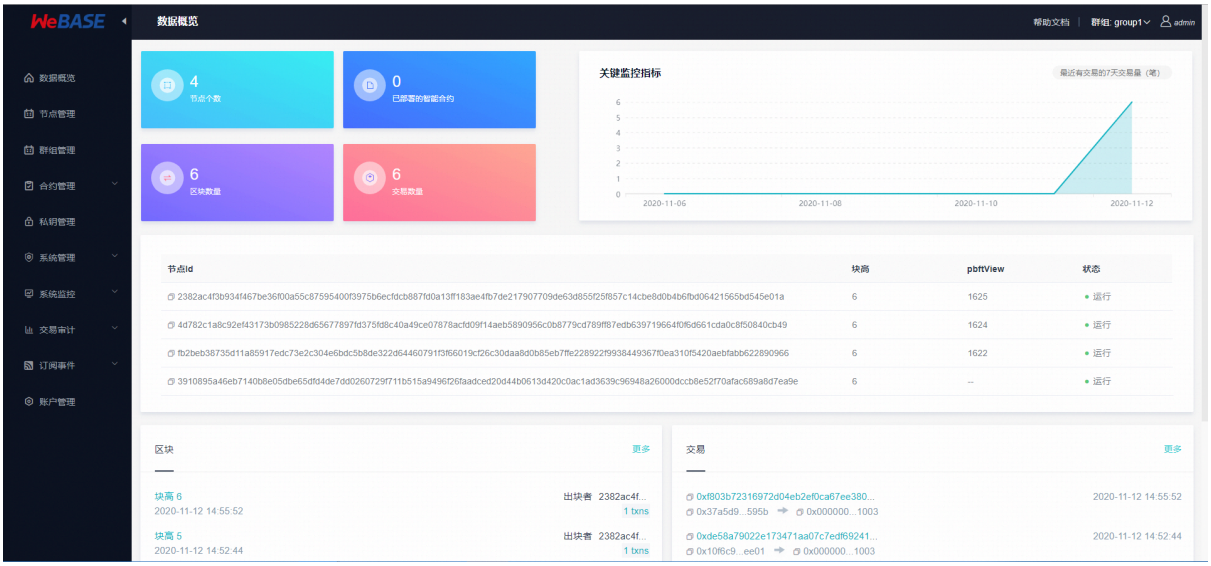
12.2.1 WeBASE快速入门

开发者只需要搭建节点和节点前置服务(WeBASE-Front)，就可通过WeBASE-Front的合约编辑器进行合约的编辑，编译，部署，调试。搭建可参考《WeBASE快速入门文档》。



12.2.2 WeBASE管理台

通过WeBASE一键脚本，可以搭建一个WeBASE的基础环境，可以方便用户体验WeBASE核心功能如：区块浏览，节点查看，合约IDE，系统管理，节点监控，交易审计，私钥管理。搭建可参考《WeBASE一键部署文



档》。

12.2.3 WeBASE其他

WeBASE其他组件可以参考《WeBASE文档》

CHAPTER 13

区块链浏览器

标签：区块链浏览器

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。

13.1 使用介绍

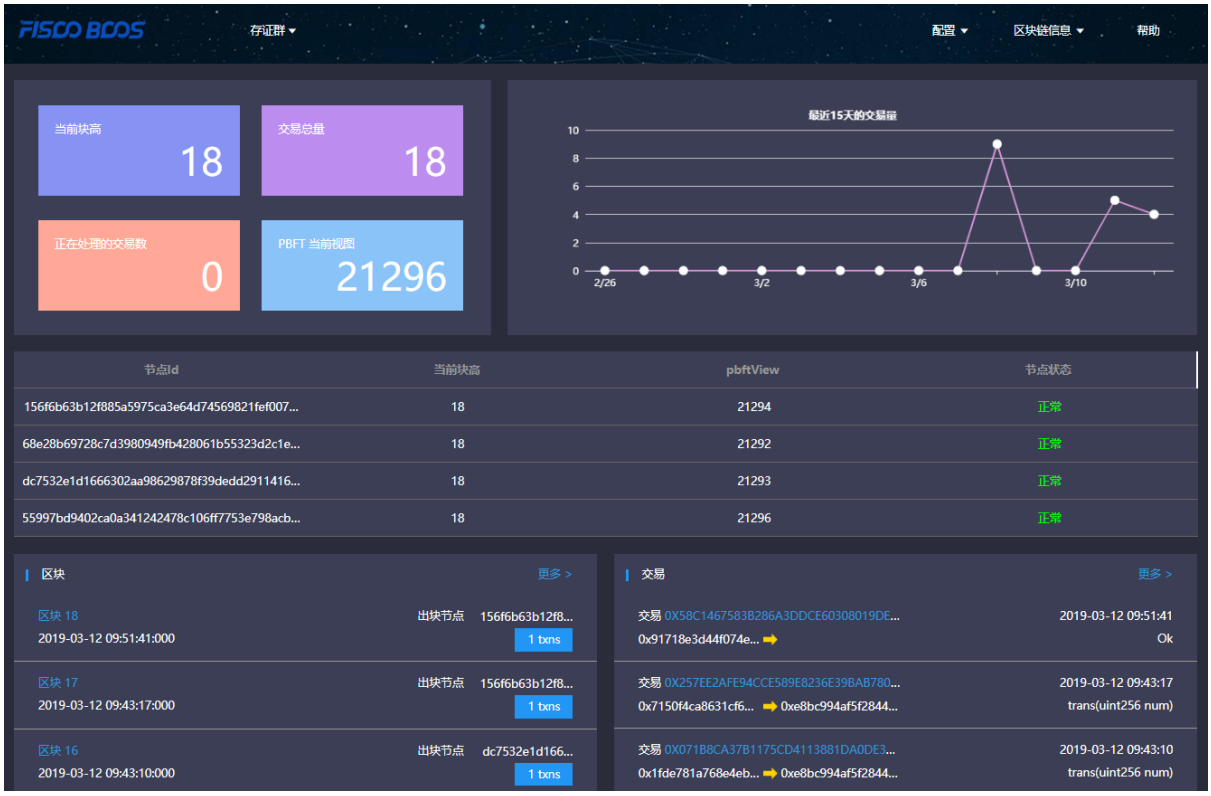
标签：区块链浏览器 图形化

13.1.1 1. 浏览器主要用途

1.1 基本介绍

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。

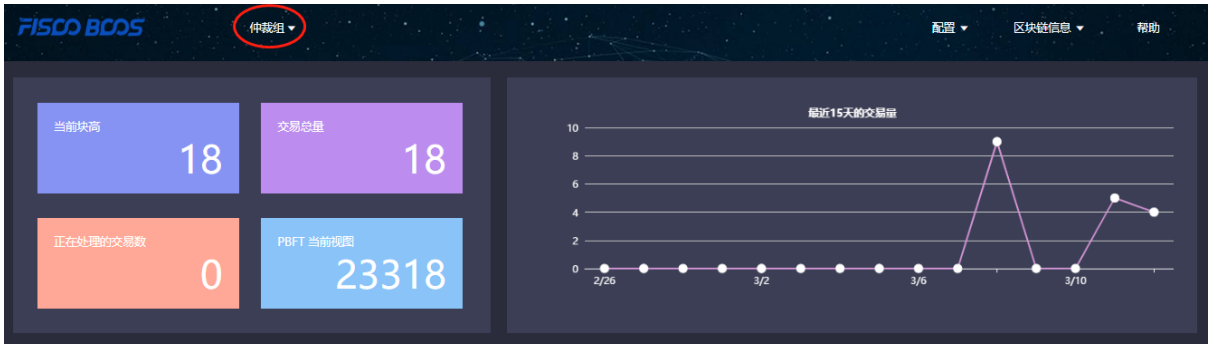


1.2 主要功能模块

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。

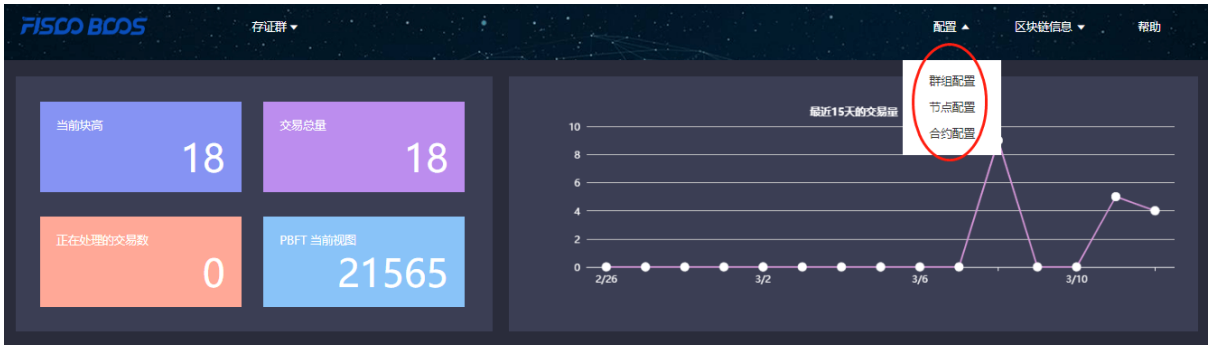
1.2.1. 群组切换模块

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。



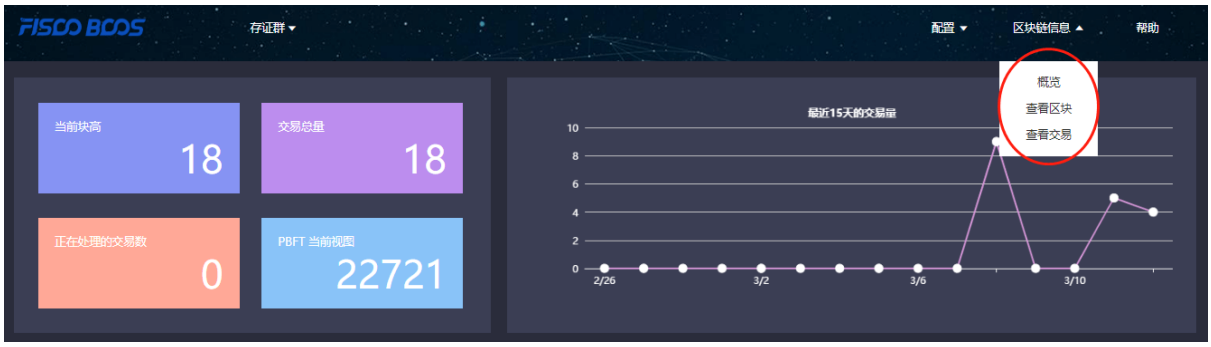
1.2.2. 配置模块

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。



13.1.2 2. 使用前提

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。

13.1.3 3. 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。

3.1. 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。

3.1. 手动搭建

后台服务搭建

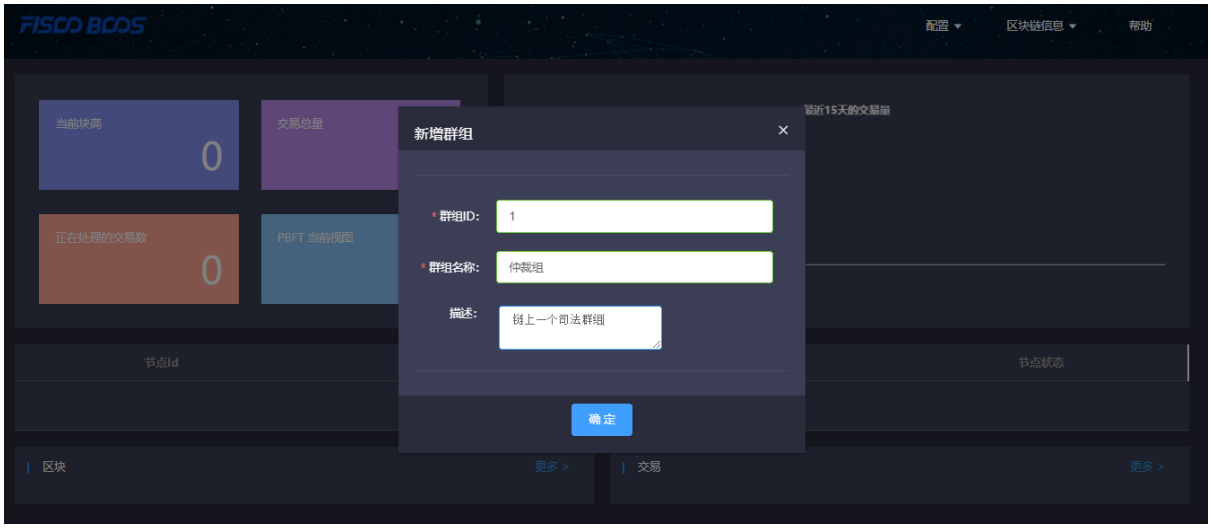
区块链浏览器后台服务使用Spring Boot的JAVA后台服务，具体搭建流程参见安装文档。

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。

13.1.4 4. 初始化环境

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。

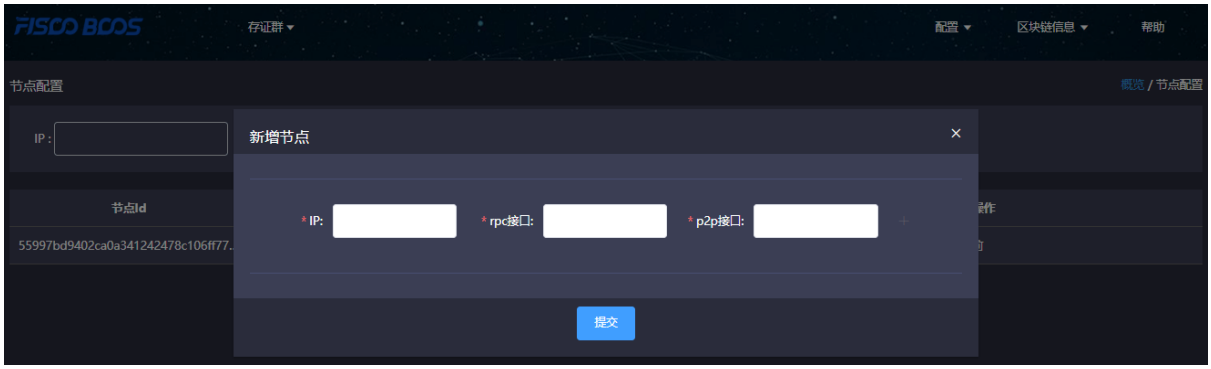
群组ID需要和区块链群组ID保持一致。 群组ID有多种查看方式:

- rpc接口获取
- 控制台命令: 2.6及其以上版本控制台使用手册请参考这里, 1.x版本控制台使用手册请参考这里。

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。

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。

4.3. 添加合约

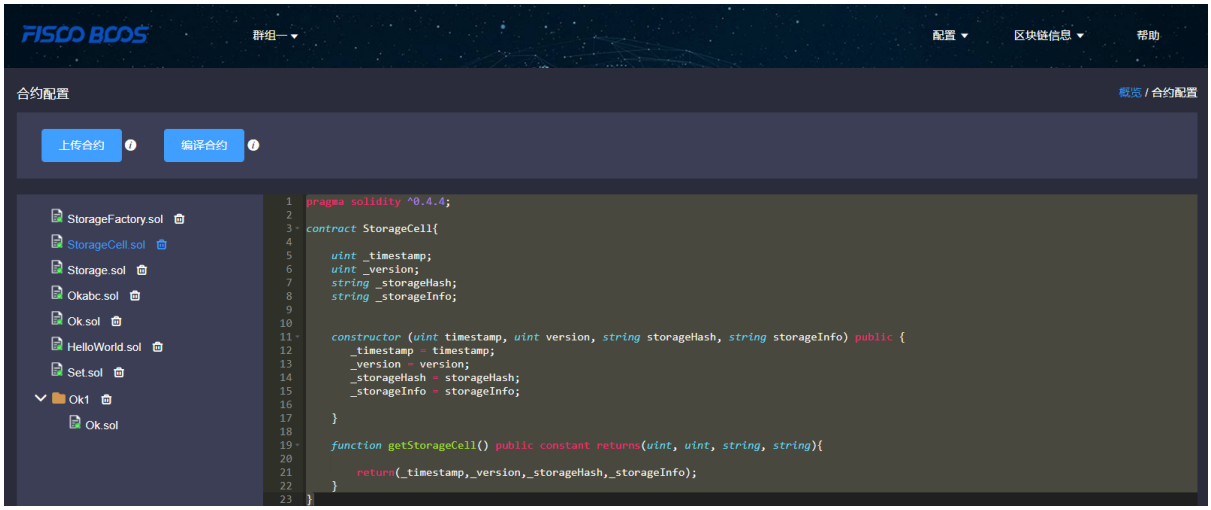
本浏览器版本提供合约解析的功能。此功能需要用户把本群组使用的所有合约进行导入。本版本支持用zip包上传一级目录，用于解决同名合约的问题。

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



13.1.5 5. 界面展示

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

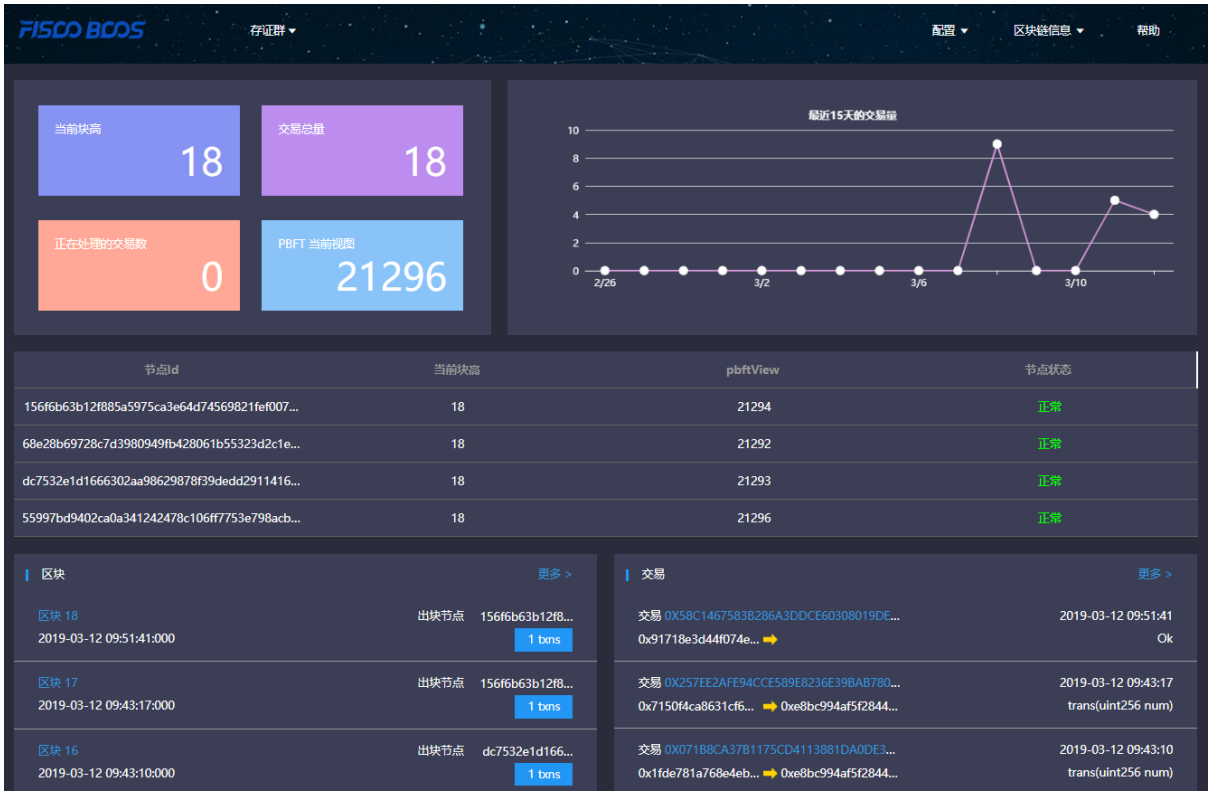
节点概览展示了当前群组内各个节点的ID，当前快高，pbftView，和节点状态。

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。



5.2、区块信息浏览

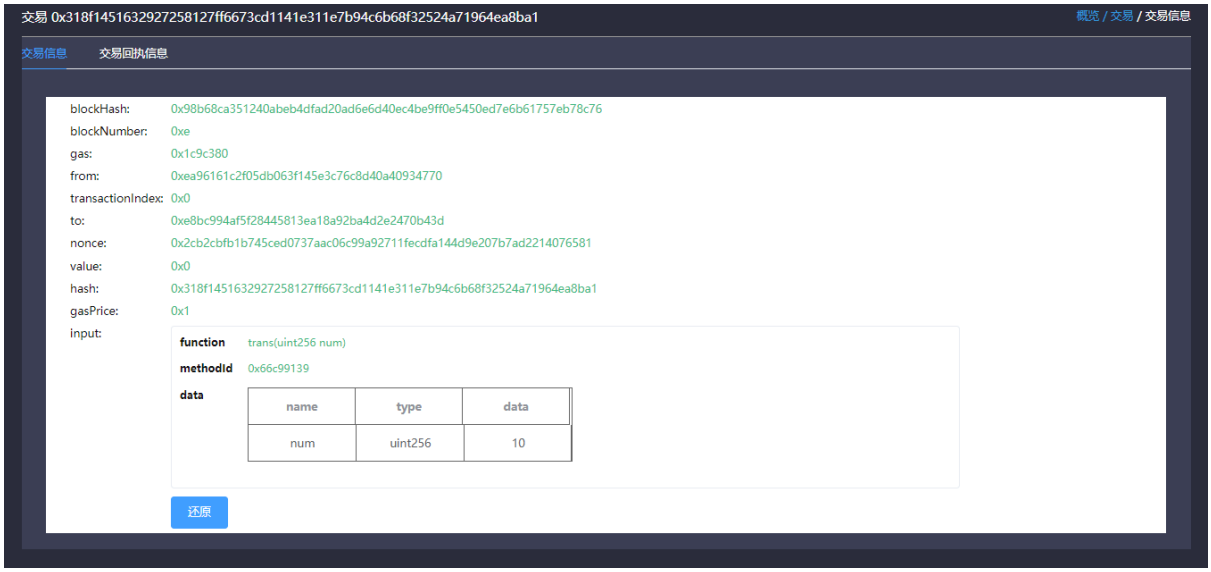
区块信息浏览主要包括区块列表页面和区块详情页面。

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。

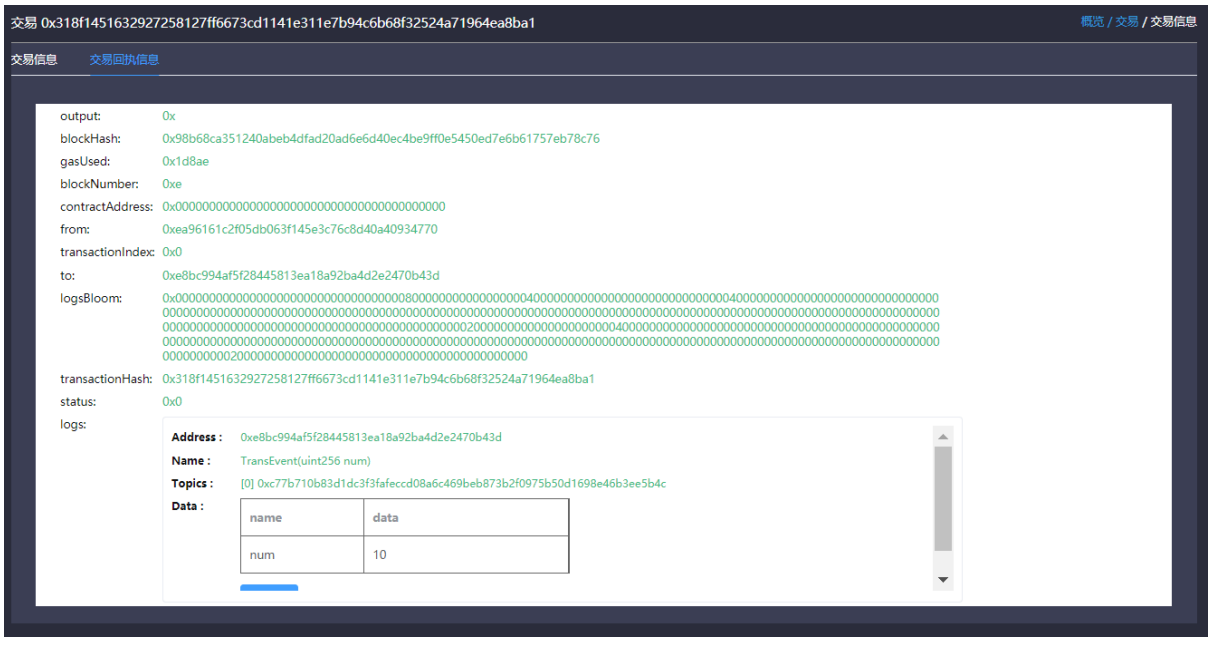
5.3.1、交易解析

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。



13.2 一键部署

标签：区块链浏览器 部署

本文档试用于v2.1.0及之后版本一键部署。v2.1.0之前版本一键部署可参考文档。

13.2.1 1.前提条件

检查环境

平台要求

推荐使用CentOS 7.2+, Ubuntu 16.04及以上版本, 一键部署脚本将自动安装`openssl, curl, wget, git, nginx`相关依赖项。

其余系统可能导致安装依赖失败，可自行安装`openssl, curl, wget, git, nginx`依赖项后重试

检查Java

推荐JDK8-JDK13版本：

```
java -version
```

- Java推荐使用*OpenJDK*，建议从OpenJDK网站 自行下载。
- **注意：需要配置root用户的java_home**

检查mysql

MySQL-5.6或以上版本：

```
mysql --version
```

- Mysql安装部署可参考数据库部署

检查Python

使用Python3.5或以上版本：

```
python3 --version
```

- Python3安装部署可参考*Python*部署

PyMySQL部署（Python3.5+）

Python3.5及以上版本，需安装PyMysql依赖包：

- CentOS

```
sudo yum -y install python36-pip
sudo pip3 install PyMySQL
```

- Ubuntu

```
sudo apt-get install -y python3-pip
sudo pip3 install PyMySQL
```

CentOS或Ubuntu不支持pip命令的话，可以使用以下方式：

```
  git clone https://github.com/PyMySQL/PyMySQL
  cd PyMySQL/
  python3 setup.py install
```

13.2.2 2.拉取安装脚本

获取部署安装包：

```
wget https://osp-1257653870.cos.ap-guangzhou.myqcloud.com/FISCO-BCOS/fisco-bcos-
↪browser/releases/download/v2.2.2/browser-deploy.zip
```

解压安装包：

```
unzip browser-deploy.zip
```

进入目录：

```
cd browser-deploy
```

13.2.3 3.修改配置（没有变化的可以不修改）

① 可以使用以下命令修改，也可以直接修改文件（vi common.properties）

② 数据库需要提前安装（数据库安装请参看 数据库部署）

③ 服务端口不能小于1024

```
数据库IP：sed -i "s/127.0.0.1/${your_db_ip}/g" common.properties
数据库端口：sed -i "s/3306/${your_db_port}/g" common.properties
数据库用户名：sed -i "s/dbUsername/${your_db_account}/g" common.properties
数据库密码：sed -i "s/dbPassword/${your_db_password}/g" common.properties
数据库名称：sed -i "s/db_browser/${your_db_name}/g" common.properties

前端服务端口：sed -i "s/5100/${your_web_port}/g" common.properties
后端服务端口：sed -i "s/5101/${your_server_port}/g" common.properties

例子（将数据库IP由127.0.0.1改为0.0.0.0）：sed -i "s/127.0.0.1/0.0.0.0/g" application.
↪yml
```

13.2.4 4.部署

部署所有服务：

```
python3 deploy.py installAll
```

停止所有服务：

```
python3 deploy.py stopAll
```

启动所有服务：

```
python3 deploy.py startAll
```

单独启停命令和说明可查看帮助：

```
python3 deploy.py help
```

备注： 部署过程出现问题可以查看 常见问题

13.2.5 5.访问

例如：在浏览器输入以下访问地址，IP为部署服务器IP，端口为前端服务端口

```
http://127.0.0.1:5100/
```

13.2.6 6.日志路径

```
部署日志：log/
后端日志：server/log/
前端日志：web/log/
```

13.2.7 7.附录

7.1 Java环境部署

此处给出OpenJDK安装简单步骤，供快速查阅。更详细的步骤，请参考官网。

① 安装包下载

从官网下载对应版本的java安装包，并解压到服务器相关目录

```
mkdir /software
tar -zxvf openjdkXXX.tar.gz /software/
```

② 配置环境变量

- 修改/etc/profile

```
sudo vi /etc/profile
```

- 在/etc/profile末尾添加以下信息

```
JAVA_HOME=/software/jdk-11
PATH=$PATH:$JAVA_HOME/bin
CLASSPATH==.:$JAVA_HOME/lib
export JAVA_HOME CLASSPATH PATH
```

- 重载/etc/profile

```
source /etc/profile
```

③ 查看版本

```
java -version
```

7.2. 数据库部署

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

① 安装MariaDB

- 安装命令

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

- 启停

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

- 设置开机启动

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

- 初始化

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

② 授权访问和添加用户

- 使用root用户登录，密码为初始化设置的密码

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

- 授权root用户远程访问

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

- 创建test用户并授权本地访问

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

安全温馨提示:

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

③ 测试连接和创建数据库

- 登录数据库

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

- 创建数据库

```
mysql > create database db_browser;
```

7.3. Python部署

python版本要求使用python3.x, 推荐使用python3.5及以上版本

- CentOS

```
sudo yum install -y python36
sudo yum install -y python36-pip
```

- Ubuntu

```
// 添加仓库，回车继续
sudo add-apt-repository ppa:deadsnakes/ppa
// 安装python 3.6
sudo apt-get install -y python3.6
sudo apt-get install -y python3-pip
```

13.2.8 8.常见问题

8.1 数据库安装后登录报错

腾讯云centos mysql安装完成后，登录报错：Access denied for user 'root'@'localhost'

① 编辑 /etc/my.cnf，在[mysqld] 部分最后添加一行

```
skip-grant-tables
```

② 保存后重启mysql

```
service mysqld restart
```

③ 输入以下命令，回车后输入密码再回车登录mysql

```
mysql -uroot -p mysql
```

8.2. Python命令出错

- SyntaxError报错

```
  File "deploy.py", line 62
    print helpMsg
                ^
SyntaxError: Missing parentheses in call to "print". Did you mean print(helpMsg)?
```

- 找不到fallback关键字

```
File "/home/ubuntu/webase-deploy/comm/utils.py", line 127, in getCommProperties
    value = cf.get('common', paramsKey,fallback=None)
TypeError: get() got an unexpected keyword argument 'fallback'
```

答：检查*Python*版本，推荐使用python3.5及以上版本

8.3 使用Python3时找不到pymysql

```
Traceback (most recent call last):
...
ImportError: No module named 'pymysql'
```

答：需要安装PyMySQL，安装请参看 *pymysql*

8.4 部署时数据库访问报错

```
...
checking database connection
Traceback (most recent call last):
  File "/data/temp/browser/fisco-bcos-browser/deploy/comm/mysql.py", line 21, in␣
↪dbConnect
    conn = mdb.connect(host=mysql_ip, port=mysql_port, user=mysql_user,␣
↪passwd=mysql_password, charset='utf8')
OperationalError: (1045, "Access denied for user 'root'@'localhost' (using␣
↪password: YES)")
```

答：确认数据库用户名和密码

8.5 Server启动失败

答：1: 请检查是否设置了JAVA_HOME。 2: 检查gradle版本是否在5.0以上。 3: 检查jdk版本，如果使用的是jdk9或者更高版本，需要在项目启动时加入 –add-modules java.xml.bind 后缀，但是这种方法只能在JDK9或者10去使用。修改位置：server/start.sh文件，start()函数，nohub $JAVA_HOME/bin/java 语句之后加上 –add-modules java.xml.bind

8.6 server启动成功，但提示启动失败

答：将openjdk换成oracle jdk，因为openjdk中缺少一些组件。

13.3 浏览器后端服务

标签：后端服务 区块链浏览器

13.3.1 1.功能说明

本工程是区块链浏览器的后端服务，功能是解析节点数据储存数据库，向前端提供数据接口，页面展示。

13.3.2 2.前提条件

13.3.3 3.部署说明

3.1 拉取代码

执行命令：

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

```
cd fisco-bcos-browser
```

3.2 编译代码

（1）进入目录：

```
cd server/fisco-bcos-browser
```

（2）执行构建命令：

```
gradle build
```

构建完成后，会在目录中生成已编译的代码目录dist。

3.3 修改配置

（1）dist目录中提供了一份配置模板dist/conf_template，**供拷贝参考**：

```
需要根据配置模板生成一份实际配置dist/conf。初次安装可直接拷贝。
例如：cp dist/conf_template dist/conf -r
```

（2）进入目录：

```
cd dist/conf
```

（3）修改服务配置（没变化可以不修改）：

数据库服务器，和数据库需要提前准备，创建方法可以参照附录。

```
修改当前服务端口：sed -i "s/5101/${your_server_port}/g" application.yml
修改数据库IP：sed -i "s/127.0.0.1/${your_db_ip}/g" application.yml
修改数据库用户名：sed -i "s/dbUsername/${your_db_account}/g" application.yml
修改数据库密码：sed -i "s/dbPassword/${your_db_password}/g" application.yml
修改数据库名称：sed -i "s/db_browser/${your_db_name}/g" application.yml

例子（将数据库IP由127.0.0.1改为0.0.0.0）：sed -i "s/127.0.0.1/0.0.0.0/g" application.
↪yml
```

温馨提示：

1. 实际生产中建议将编译后的安装包（dist目录）放到服务部署目录。例如/data/app/fisco-bcos-browser

3.4 服务启停

进入到已编译的代码根目录：

```
cd dist
```

```
启动：sh start.sh
停止：sh stop.sh
检查：sh status.sh
```

3.5 查看日志

进入到已编译的代码根目录：

```
cd dist
```

查看

```
tail -f log/fisco-bcos-browser.log
```

13.3.4 4.问题排查

4.1 启停失败

如果脚本执行出现问题，尝试以下操作：

```
chmod +x *.sh
```

4.2 gradle build失败

```
gradle build
```

执行后，出现下面错误。**请检查gradle版本，需要使用5.0以上版本。**

```
Could not find method annotationProcessor() for arguments [org.
↪projectlombok:lombok:1.18.2] on object of type org.gradle.api.internal.artifacts.
↪dsl.dependencies.DefaultDependencyHandler.
```

13.3.5 5.附录

5.1 Java环境部署

此处给出简单步骤，供快速查阅。更详细的步骤，请参考官网。

（1）从官网下载对应版本的java安装包，并解压到相应目录

```
mkdir /software
tar -zxvf jdkXXX.tar.gz /software/
```

（2）配置环境变量

```
export JAVA_HOME=/software/jdk1.8.0_121
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

5.2 gradle环境部署

此处给出简单步骤，供快速查阅。更详细的步骤，请参考官网。

（1）从官网下载对应5.0以上版本的gradle安装包，并解压到相应目录

```
mkdir /software/
unzip -d /software/ gradle-XXX.zip
```

（2）配置环境变量

```
export GRADLE_HOME=/software/gradle-XXX
export PATH=$GRADLE_HOME/bin:$PATH
```

5.3 数据库部署

此处以Centos/Fedora为例。

（1）切换到root

```
sudo -s
```

（2）安装mysql

```
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*
```

（3）启动mysql

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

（4）初始化数据库用户

初次登录

```
mysql -u root
```

给root设置密码和授权远程访问

```
mysql > SET PASSWORD FOR 'root'@'localhost' = PASSWORD('123456');
mysql > GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '123456' WITH␣
↪GRANT OPTION;
```

安全温馨提示：

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

授权test用户本地访问数据库

```
mysql > create user 'test'@'localhost' identified by '123456';
```

（5）测试连接

另开一个ssh测试本地用户test是否可以登录数据库

```
mysql -utest -p123456 -h 127.0.0.1 -P 3306
```

登陆成功后，执行以下sql语句，若出现错误，则用户授权不成功

```
mysql > show databases;
mysql > use test;
```

（6）创建数据库

登录数据库

```
mysql -utest -p123456 -h 127.0.0.1 -P 3306
```

创建数据库

```
mysql > create database db_browser;
```

5.3.1 常见错误

5.3.1.1 腾讯云centos mysql安装完成后，登陆报错：Access denied for user 'root'@'localhost'

1. 编辑 /etc/my.cnf ，在[mysqld] 部分最后添加一行

```
skip-grant-tables
```

1. 保存后重启mysql

```
service mysqld restart
```

1. 输入以下命令，回车后输入密码再回车登录Mysql

```
mysql -uroot -p mysql
```

13.4 浏览器前端

标签：前端 区块链浏览器

本项目是FISCO BCOS前端项目，使用框架vue-cli。
兼容浏览器IE9及以上，360浏览器兼容版（IE9内核），360浏览器极速版，chrome浏览器。

13.4.1 1.功能

(1)、主要功能是区块链概览，查看区块，查看交易，节点配置以及群组切换。
(2)、支持群组切换，需要配置群组和节点。
(3)、上传并编译发送交易的合约后，可以查看交易的inputs和event解码数据。
(4)、区块链概览，查看区块，查看交易和节点配置页面每10s执行一轮请求。

13.4.2 2.部署

2.1 依赖环境

nginx安装请参考附录

2.2 拉取代码

执行命令：(前后端同机部署时，只需要拉取一次)

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

```
cd fisco-bcos-browser
```

然后将其中./web/fisco-bcos-browser-front/目录中的dist目录放到/data/app/web目录下。
提示： 目录可以自定义指定，只需要nginx配置文件步骤2保持一致即可。

2.3 修改nginx配置

在./web/fisco-bcos-browser-front/doc文件下有nginx配置文件，直接可以拿来替换安装的nginx的配置文件nginx.conf;

注意： 如果按照附录安装的nginx，配置文件路径在/usr/local/nginx/conf/nginx.conf 。

然后修改nginx.conf;

1. 修改前端服务的ip地址和端口。
2. 修改前端文件的路径,直接指向已拉取代码的dist目录。并且如果有需要请修改nginx的user配置，换成对应的user用户（有dist目录访问权限的用户）
3. 修改后端服务(fisco-bcos-browser)的ip和端口，注意'/api'不要修改。

```
#user nobody
```

```
    server {
            listen       5100 default_server;    #步骤1、前端nginx监听端口
            server_name  192.168.0.1;            #步骤1、前端地址，可配置为域名
            location / {
                    root    /data/app/web/dist;    #步骤2、前端文件路径
                    index  index.html index.htm;
                    try_files $uri $uri/ /index.html =404;
                }

            # Load configuration files for the default server block.
            include /etc/nginx/default.d/*.conf;

            location /api {
                    proxy_pass     http://192.168.0.1:5101/;     #步骤3、后端服务(fisco-
↪bcos-browser server)地址及端口
                                proxy_set_
↪header                 Host                                    $host;
                    proxy_set_header                X-Real-
↪IP                         $remote_addr;
                    proxy_set_header                X-Forwarded-For
↪$proxy_add_x_forwarded_for;
            }
            }
```

2.4 启动nginx

(1)、启动nginx。 启动命令:

```
/usr/local/nginx/sbin/nginx
```

启动报错重点排查:

1. 日志路径是否正确（error.log和access.log）
2. nginx有没有添加用户权限。

(2)、打开页面，页面url是nginx配置的前端端口和前端ip。 例如:上面配置文件的url为http:192.168.0.1:5100

(3)、打开页面后，配置群组（群组ip是搭链的群组ip），配置节点（该群组下节点），然后就可以查看具体数据了。

13.4.3 3.附录

3.1 安装nginx（可参考网络教程）

3.1.1 下载nginx依赖

在安装nginx前首先要确认系统中安装了gcc.pcre-devel.zlib-devel、openssl-devel。如果没有，请执行命令

```
yum -y install gcc pcre-devel zlib-devel openssl openssl-devel
```

执行命令时注意权限问题，如遇到，请加上sudo

3.1.2 下载nginx

nginx下载地址：https://nginx.org/download/（下载最新稳定版本即可） 或者使用命令：

```
wget http://nginx.org/download/nginx-1.10.2.tar.gz  (版本号可换)
```

将下载的包移动到/usr/local/下

3.1.3 安装nginx

3.1.3.1 解压

```
tar -zxvf nginx-1.10.2.tar.gz
```

3.1.3.2 进入nginx目录

```
cd nginx-1.10.2
```

3.1.3.3 配置

```
./configure --prefix=/usr/local/nginx
```

3.1.3.4 make

```
make
make install
```

3.1.3.5 测试是否安装成功

使用命令：

```
/usr/local/nginx/sbin/nginx -t
```

正常情况的信息输出：

```
nginx: the configuration file /usr/local/nginx/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/nginx/conf/nginx.conf test is successful
```

3.1.3.6 nginx几个常见命令

```
/usr/local/nginx/sbin/nginx -s reload              # 重新载入配置文件
/usr/local/nginx/sbin/nginx -s reopen              # 重启 Nginx
/usr/local/nginx/sbin/nginx -s stop                # 停止 Nginx
ps -ef | grep nginx                                # 查看nginx进程
```

CHAPTER 14

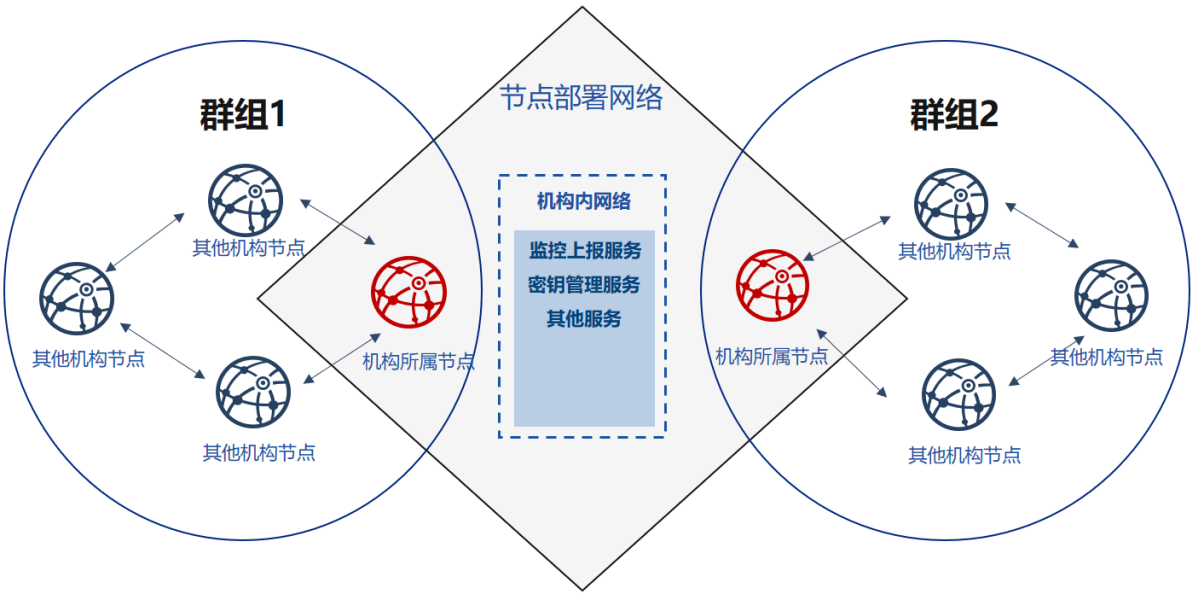
运维部署工具

标签：运维部署工具 管理工具 监控工具

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。

14.1 一键部署

标签：运维部署工具 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。

14.1.1 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，如果网络较差，可以尝试通过其他方式下载fisco-bcos，或使用--cdn选项，并将下载好的二进制放置于meta文件夹下

```
./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
```

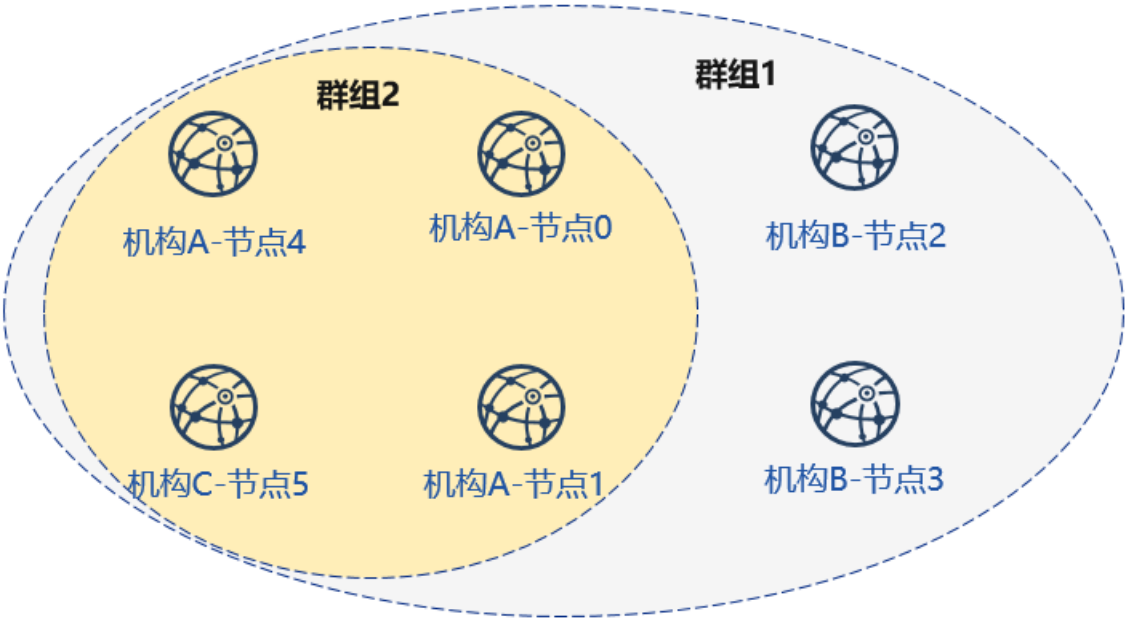
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉meta文件夹下的二进制即可。

14.1.2 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

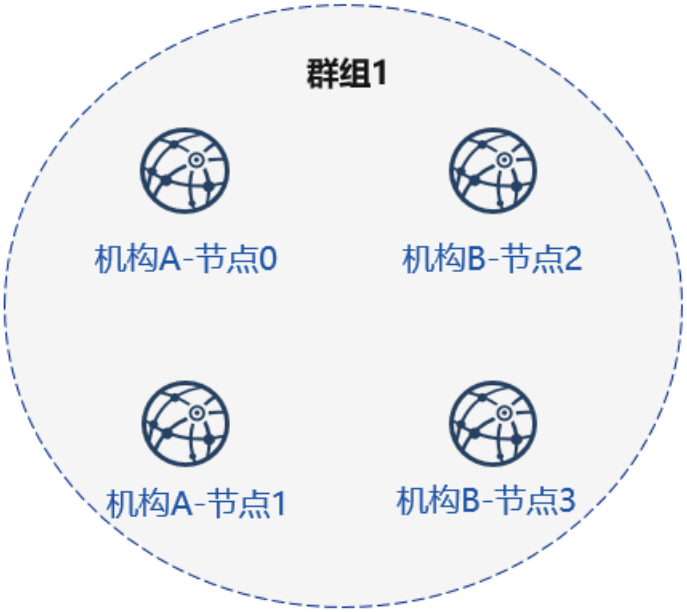
每个节点的IP，端口号为如下：

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

14.1.3 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：

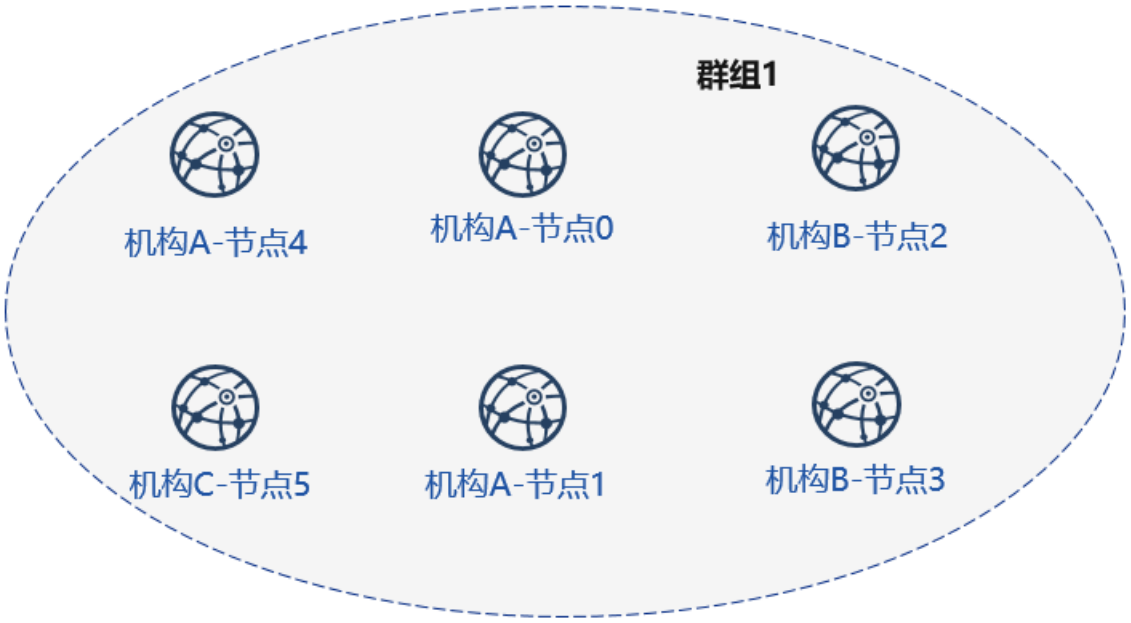
```
tail -f ~/generator/tmp_one_click/agency*/node/node*/log/log*  | grep +++
```

```
# 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...
```

14.1.4 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
5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa9ae
```

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：

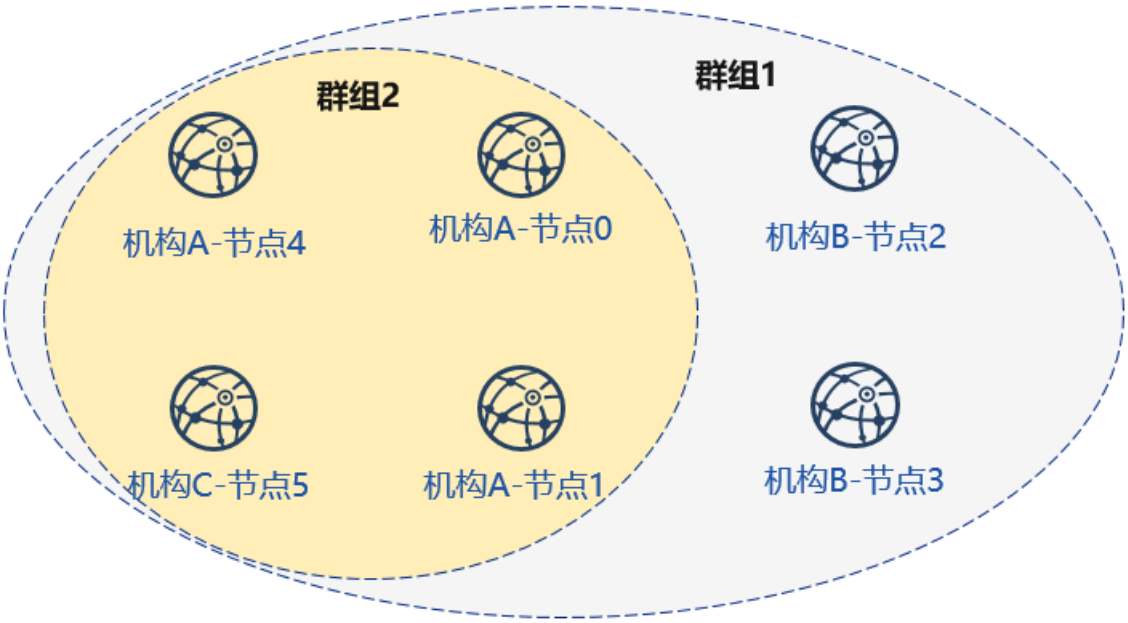
```
exit
```

至此，我们完成了新增节点至现有群组的操作。

14.1.5 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*

14.1.6 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

14.2 使用运维部署工具

标签：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。

本文为在本地服务器部署的操作手册，多机部署，可以参考这里

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。

14.2.1 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，如果网络较差，可以尝试通过其他方式下载`fisco-bcos`，或使用`--cdn`选项，并将下载好的二进制放置于`meta`文件夹下

```
./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
```

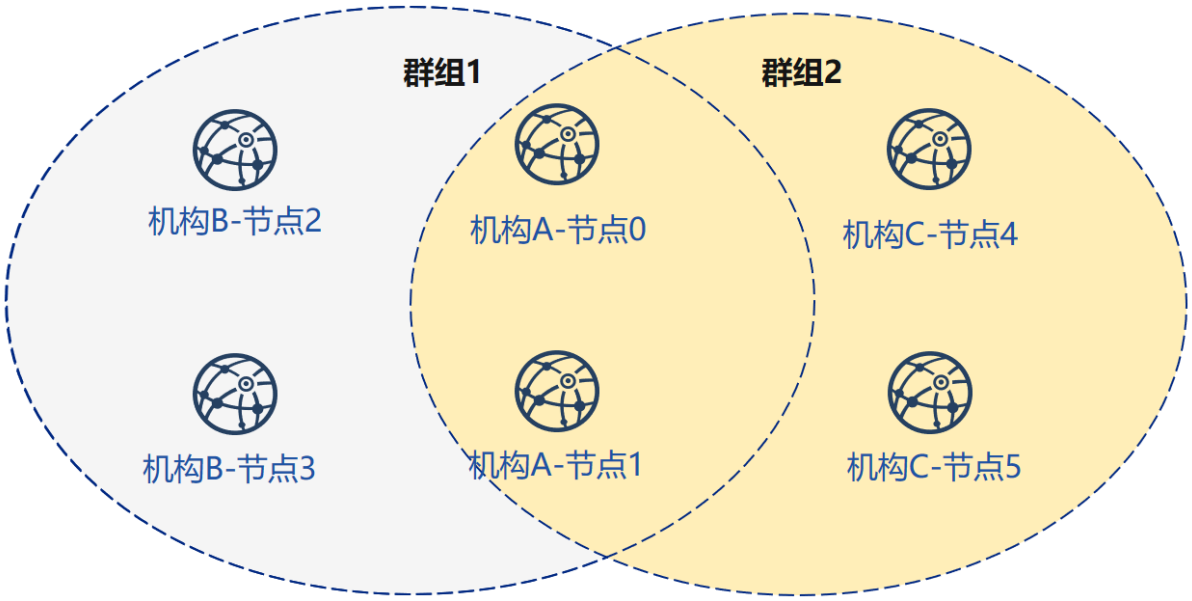
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉`meta`文件夹下的二进制即可。

14.2.2 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

14.2.3 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
```

14.2.4 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连接信息，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)

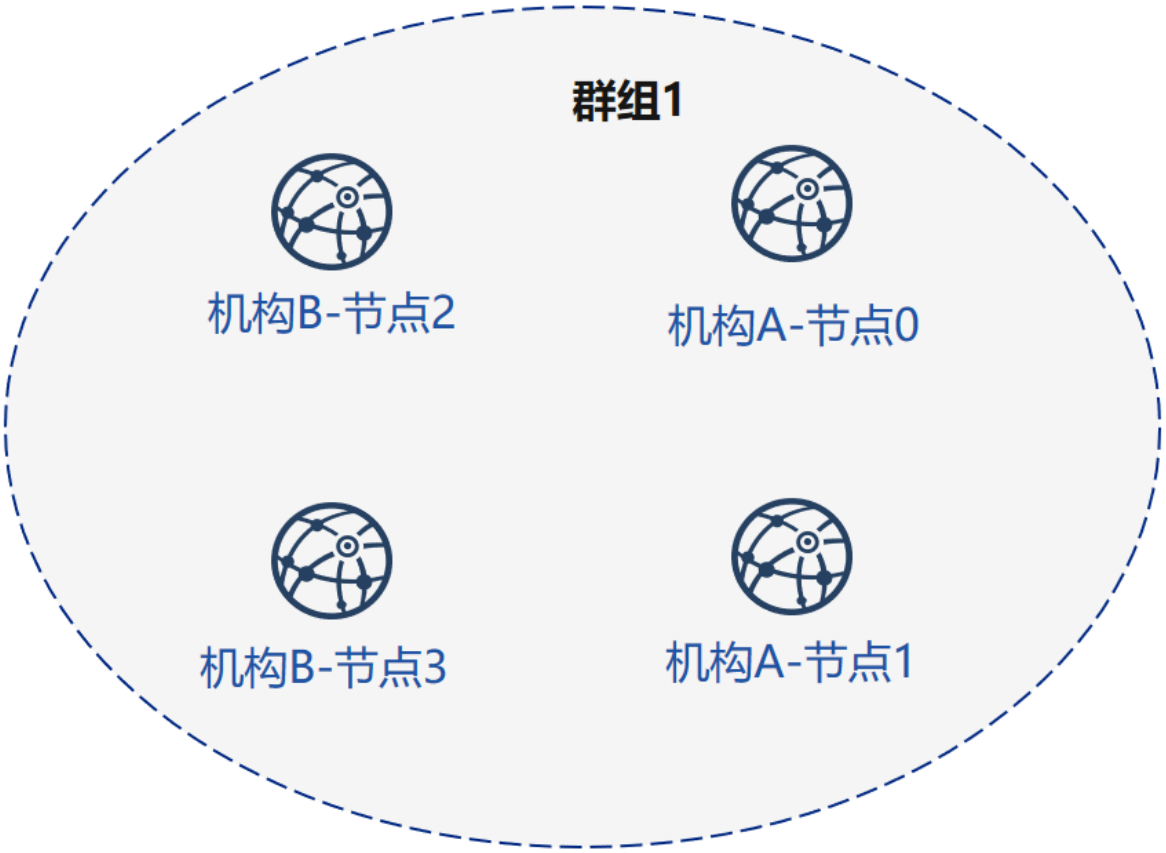
```
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
```

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:



14.2.5 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/
```

14.2.6 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：

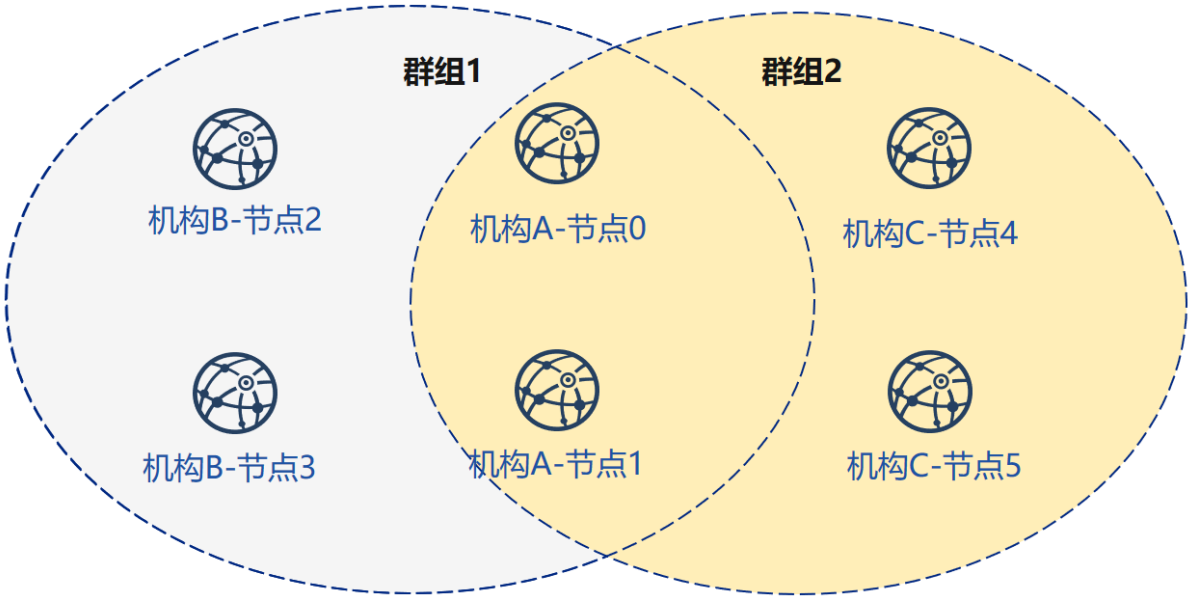
在~/generator-C目录下执行下述命令

```
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：



14.2.7 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

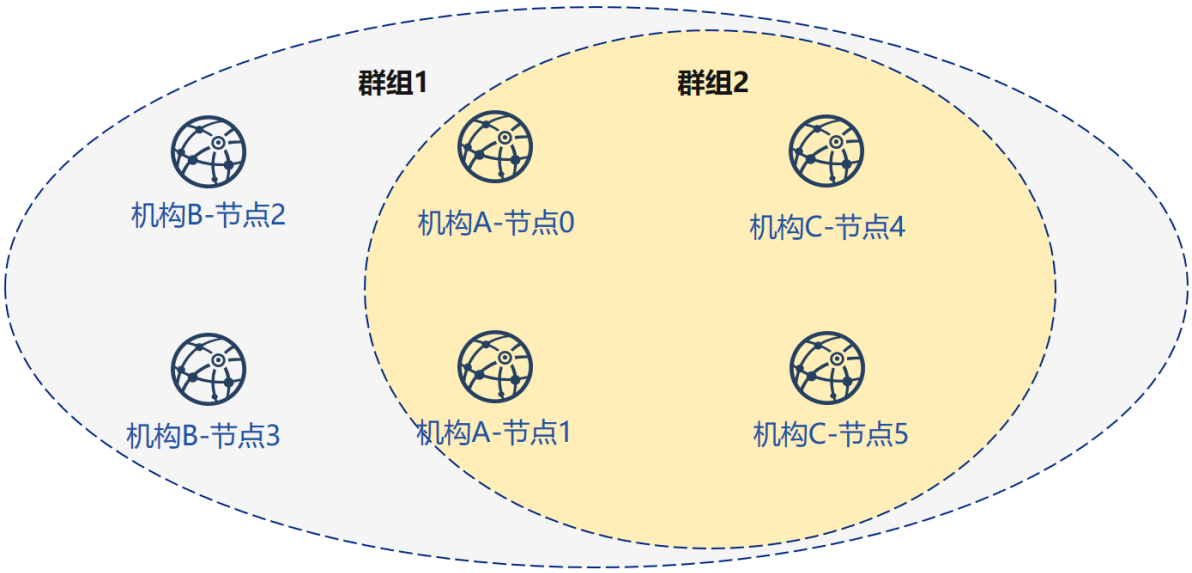
在~/generator-C目录下执行下述命令

```
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
```



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

14.2.8 扩展教程2–机构A扩容节点加入群组1

本节中，我们将会为机构A新增节点6，IP、端口号如下：

在上述过程中，机构A已经生成了自己所属的机构证书及私钥，并且拥有了群组1的创世区块，扩容节点需要进行的操作如下。

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：

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=30306
```

(continues on next page)

(续上页)

```
channel_listen_port=20206
jsonrpc_listen_port=8551
EOF
```

机构A生成节点证书

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_new
```

查看生成文件:

```
ls ./agencyA_node_info_new
```

```
# 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_30306.crt peers.txt
```

机构A生成新增节点

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连接信息，多个机构组网的情况下需要将其合并。

合并当前节点的peers.txt

```
cat ./agencyA_node_info/peers.txt >> ./meta/peersB.txt
```

生成新增节点:

```
./generator --build_install_package ./meta/peersB.txt ./nodeA_new
```

查看生成节点配置文件夹:

```
ls ./nodeA_new
```

```
# 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_30306 # 127.0.0.1服务器 端口号30306的节点配置文件夹
├── 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_new/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
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
fisco  15673     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA_new/node_127.0.0.
↪1_30306/fisco-bcos -c config.ini
```

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
```

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:

如果已经配置过控制台，请忽略此步

```
./generator --download_console ./ --cdn
```

查看机构A节点6信息

机构A使用控制台加入节点6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-A/nodeA_new/node_127.0.0.1_30306/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
degca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c9715
```

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:

添加共识节点的过程类似，参考addObserver命令

```
addSealer␣
↪degca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
```

```
# 命令解释
# 执行成功会提示success
$ [group:1]> addSealer␣
↪degca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

退出控制台：

```
exit
```

查看机构A新增节点

在~/generator-A目录下执行下述命令

```
cd ~/generator-A
```

查看节点log内group1信息:

```
cat nodeA_new/node_127.0.0.1_30306/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。

如果使用该教程遇到问题，请查看FAQ

14.3 部署国密区块链

标签：部署区块链 国密区块链 运维部署工具

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。

14.3.1 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
```

获取节点二进制

拉取最新fisco-bcos二进制文件到meta中，如果网络较差，可以尝试通过其他方式下载`fisco-bcos`，或使用`--cdn`选项，并将下载好的二进制放置于`meta`文件夹下

```
./generator --download_fisco ./meta
```

检查二进制版本

2.5 及以前版本，若成功，输出 FISCO-BCOS Version : x.x.x-x

2.6 版本之后，直接输出 FISCO-BCOS Version : x.x.x-x

```
./meta/fisco-bcos -v
```

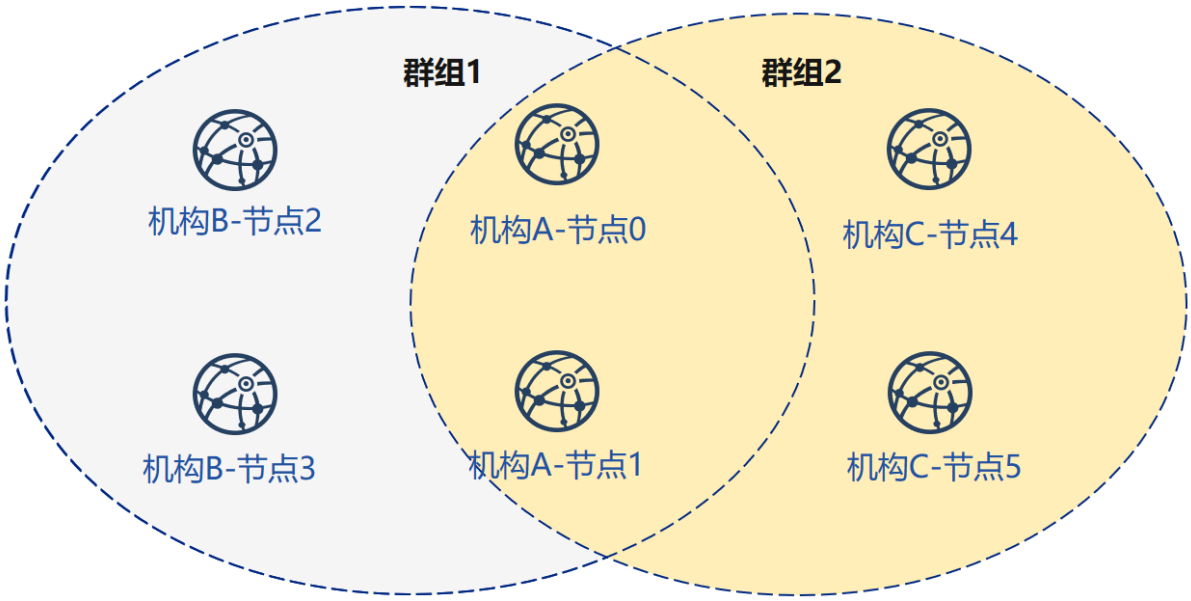
PS：源码编译节点二进制的用户，只需要用编译出来的二进制替换掉`meta`文件夹下的二进制即可。

14.3.2 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

14.3.3 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
```

14.3.4 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
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连接信息，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
```

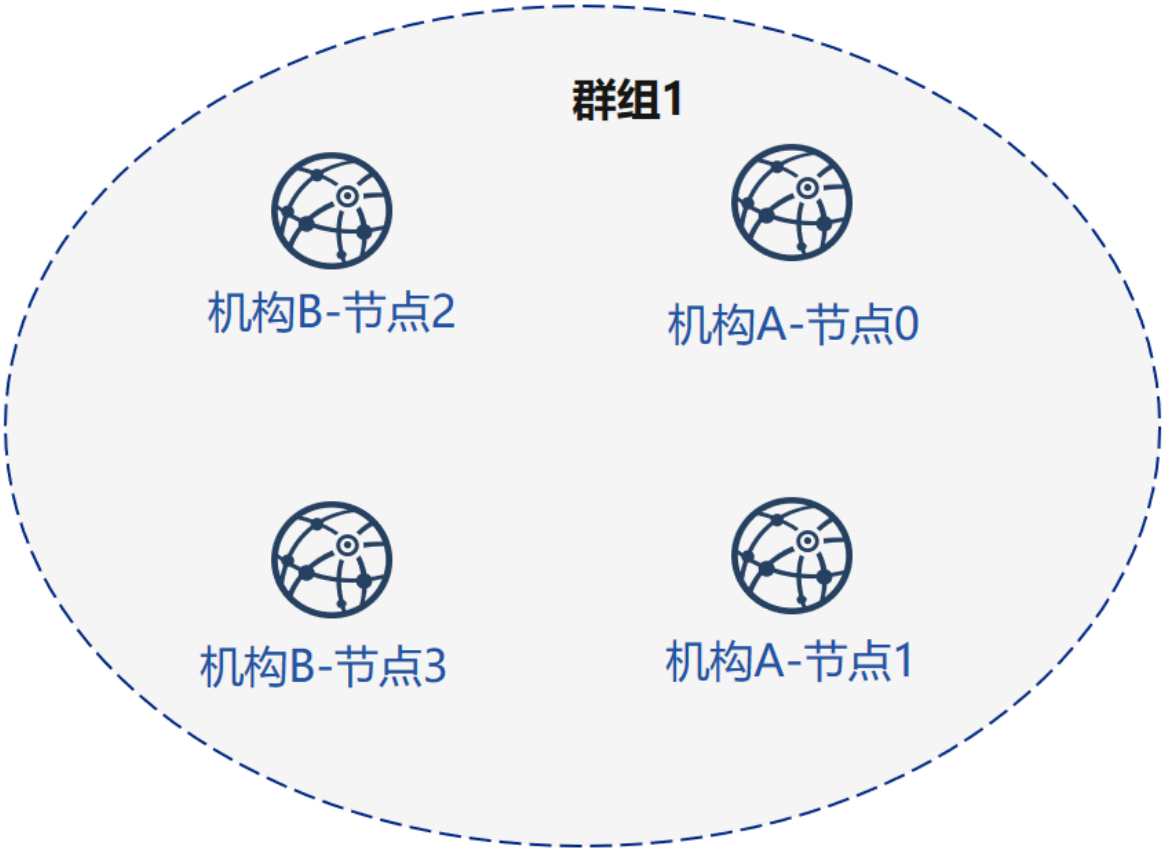
```
# 命令解释
# 可以看到如下所示的进程
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30300/fisco-bcos -c config.ini
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪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:



14.3.5 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/
```

14.3.6 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：

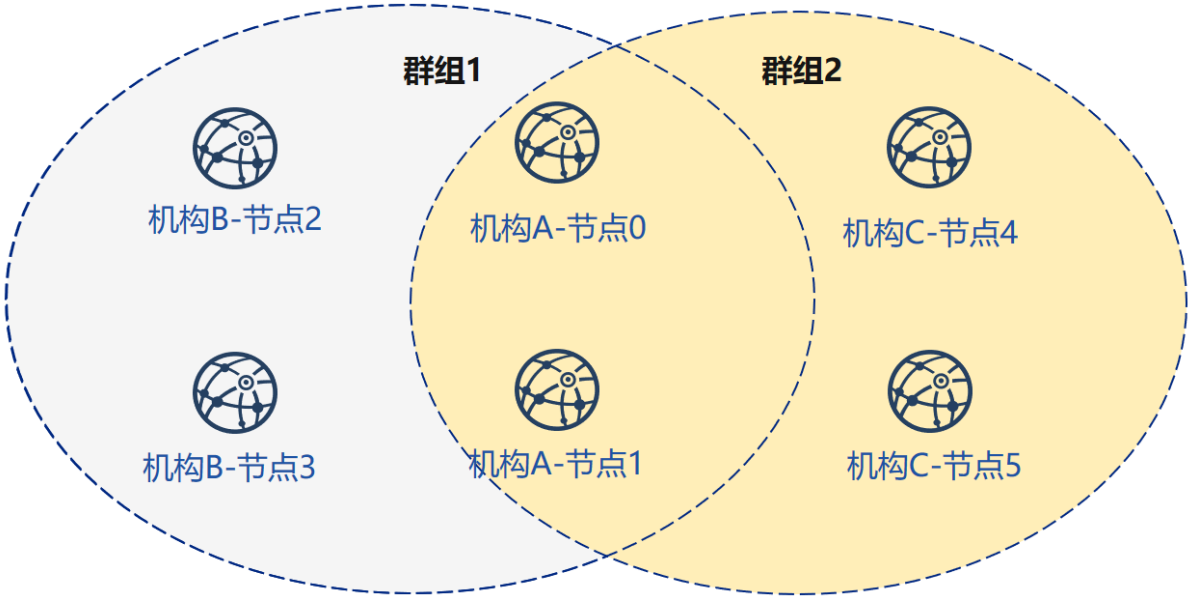
在~/generator-C目录下执行下述命令

```
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：



14.3.7 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 (**默认下载2.6.0+版本控制台不需要额外配**)：

```
./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

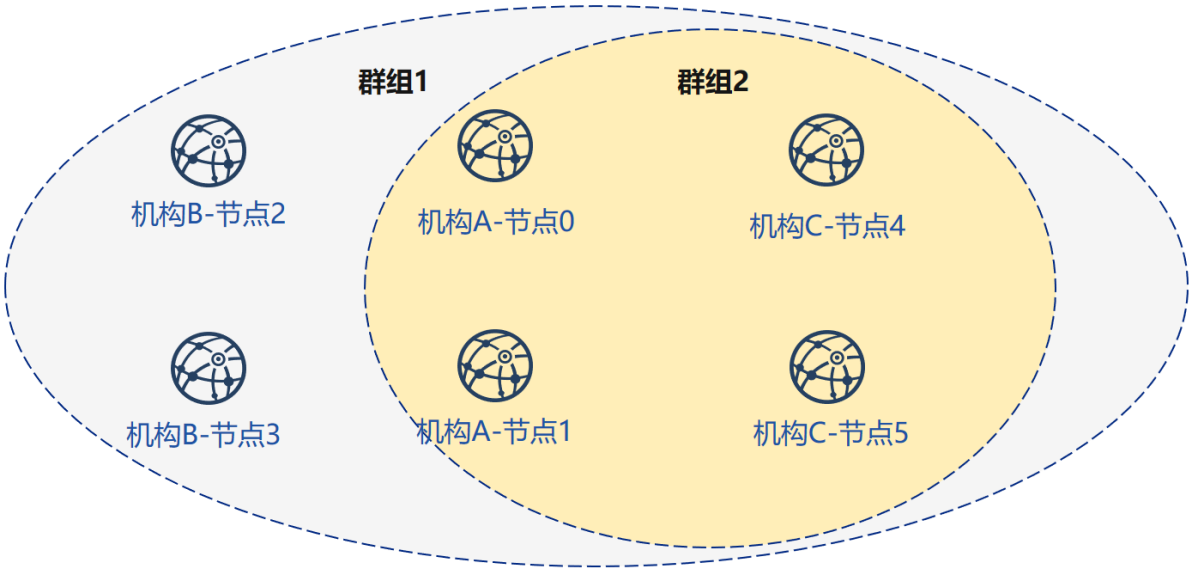
```
cd ~/generator-C
```

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
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

14.4 下载安装

标签：运维部署工具 下载安装

14.4.1 环境依赖

FISCO BCOS generator依赖如下：

14.4.2 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
```

14.4.3 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。

14.5 配置方法

标签：运维部署工具 配置

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。

14.5.1 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。

14.5.2 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。

14.5.3 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。

14.5.4 模板文件夹tpl

generator的模板文件夹如下图所示：

```
├── config.toml  # 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
```

(continues on next page)

(续上页)

```
├── 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

14.5.5 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

14.6 操作手册

标签：运维部署工具 命令

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：

14.6.1 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。

14.6.2 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

14.6.3 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。

14.6.4 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。

14.6.5 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。

14.6.6 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。

FISCO-BCOS 2.5及之后的版本，添加了SDK只能连本机构节点的限制，操作时需确认拷贝证书的路径，否则建联报错。

14.6.7 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

14.6.8 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

14.6.9 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

14.6.10 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

14.6.11 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

14.6.12 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

14.6.13 download_console

使用–download_console可以指定的目录下下载并配置控制台，其支持--console_version选项指定下载的控制台版本，默认下载最新版本控制台; 也支持--cdn命令从cdn下载控制台，推荐国内用户使用--cdn命令从cdn下载。

使用示例:

```
$./generator --download_console ./meta
```

或

```
$./generator --download_console ./meta --cdn
```

指定下载1.2.0版本控制台的使用示例如下:

```
$./generator --download_console ./meta --console_version 1.2.0 --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

14.6.14 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。

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

14.6.15 version (-v)

使用–version命令查看当前部署工具的版本号。

```
$ ./generator --version
```

14.6.16 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]
```

14.6.17 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

14.6.18 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
```

14.6.19 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。

14.6.20 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 15

数据治理通用组件

标签：WeBankBlockchain-Data 数据治理 通用组件 数据导出 数据仓库 数据对账

15.1 什么是数据治理通用组件

数据治理通用组件的全名是WeBankBlockchain-Data数据治理通用组件，它是一套稳定、高效、安全的区块链数据治理组件解决方案，可无缝适配FISCO BCOS区块链底层平台。
它由数据导出组件(Data-Export)、数据仓库组件(Data-Stash)、数据对账组件(Data-Reconcile)这三款相互独立、可插拔、可灵活组装的组件所组成，开箱即用，灵活便捷，易于二次开发。

这三个组件分别从底层数据存储层、智能合约数据解析层和应用层三个方面，提供了区块链数据挖掘、裁剪、扩容、可信存储、抽取、分析、审计、对账、监管等数据治理方面的关键能力。
WeBankBlockchain-Data已在金融、公益、农牧产品溯源、司法存证、零售等多个行业落地和使用。

15.2 设计目标

在区块链底层和区块链应用之间，横亘着一条区块链技术、业务和产品的鸿沟，存在着区块链数据扩容难度大、链上数据难以查询和分析、典型场景缺乏通用化产品或组件支持等多种挑战。

社区也经常会有朋友问到： 区块链节点服务器的磁盘快满了，该怎么办？ 如何批量查询智能合约里的数据？ 我想查询某个账号发过的所有交易怎么查？ 微众银行区块链对账方案是怎么做的，有没有通用的解决方案？ ……

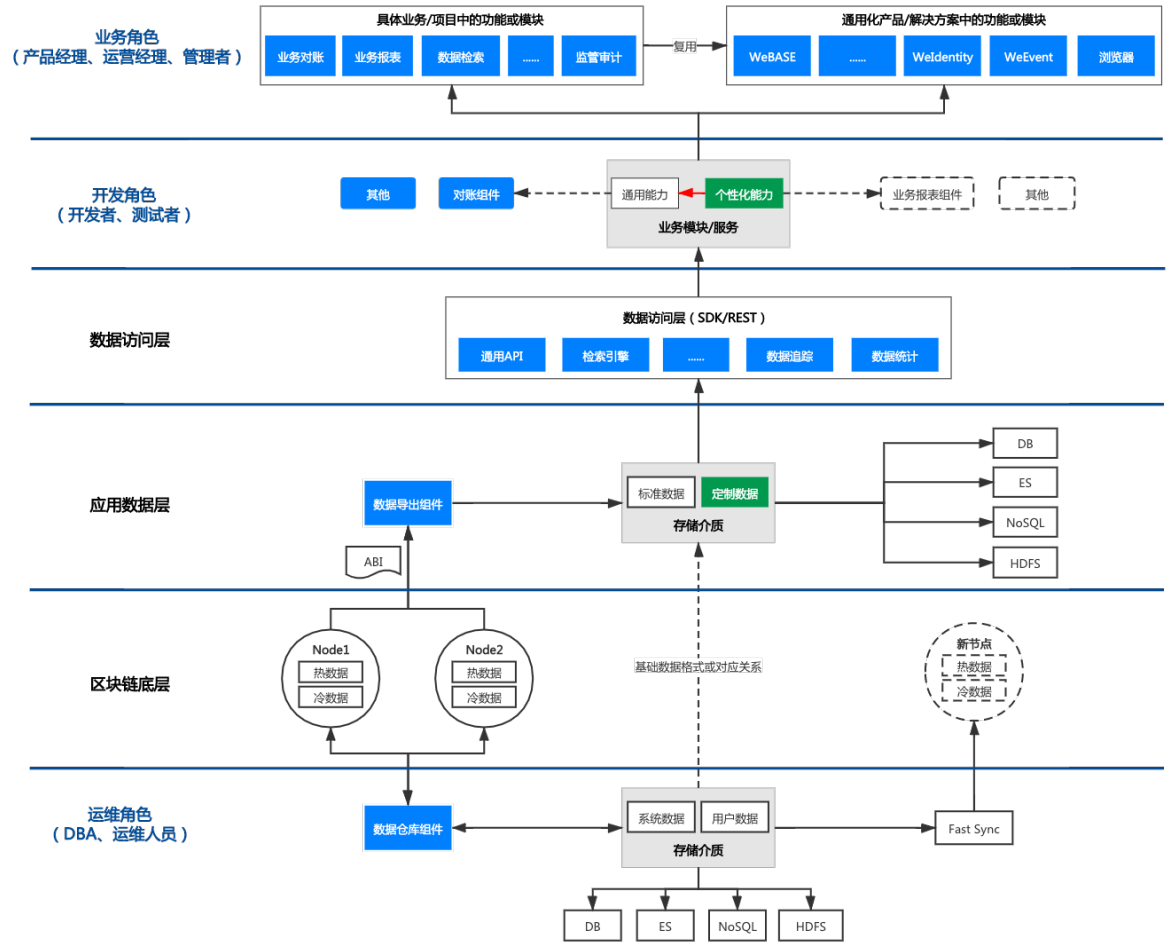
为什么强大的区块链还有这些问题呢?

首先，随着区块链数据的“爆炸式”增长，链上已经积累了亿计的交易、几T的数据，节点服务器逐渐无法满足交易数据的存储需求，单纯对节点存储空间进行扩容不仅开发成本高、硬件成本高，而且在数据扩容过程中因技术要求较高，易引发系统性风险，且不能一劳永逸地解决问题。另一方面，大量交易冷数据不仅浪费空间，而且会影响区块链节点出块和执行交易的性能。

其次，区块链由于特定的链式存储结构，链上数据只能通过智能合约接口的方式获取和调用，不仅执行效率低下，且随着链上数据的增加，其查询和计算性能逐级衰减，无法满足大数据分析和复杂查询的诉求，比如需要检索已经部署在链上的所有合约。基于特定场景的数据导出解决方案，由于智能合约存在较大差异，需要针对智能合约做特定开发，成本很高，不能复用。

最后，基于区块链的可信数据缺乏通用的产品和可复用的组件，部分场景之间存在类似的需求，比如业务对账、区块链浏览器、业务分析、监管审计等。不同项目之间存在着大量重复的开发，费时费力，同时区块链应用的开发者完成工作目标需要经历陡峭的学习曲线，还可能在开发、测试中引入各类风险。

WeBankBlockchain-Data 从底层数据存储层、智能合约数据解析层和应用层等多个层次入手，提供区块链数据挖掘、裁剪、扩容、可信存储、抽取、分析、审计、对账、监管等数据治理方面的关键能力，满足数据治理全流程开发场景的需求，如下图所示：



区块链数据经过区块链共识节点的多方共识，一旦生成就不会被修改。

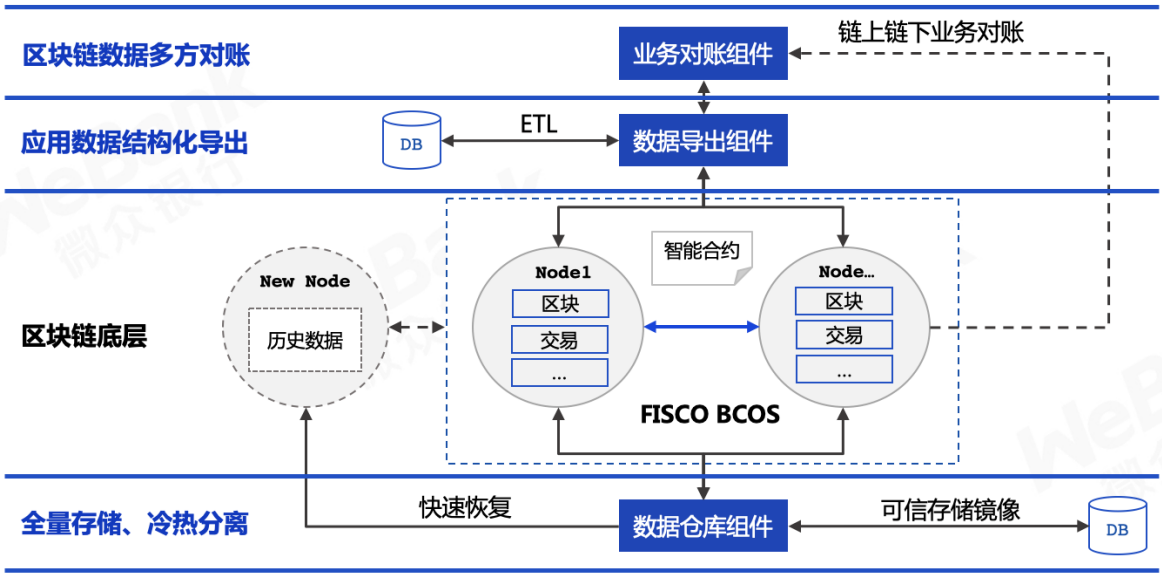
在运维层，区块链的历史区块数据可被数据仓库组件完整或部分导出到本地。作为可信存储镜像，导出的数据仅在本地有效，被修改不会影响链上共识。我们建议使用方建立管理方法，限制对本地数据的修改。

在应用数据层，数据导出组件支持导出源数据、初步解析和基于合约解析的多维数据。所有参与者均可部署自己的导出服务，作为本地查询或分析的可信数据源。

在业务层，业务对账组件支持机构内外部对链上链下业务数据进行对账。

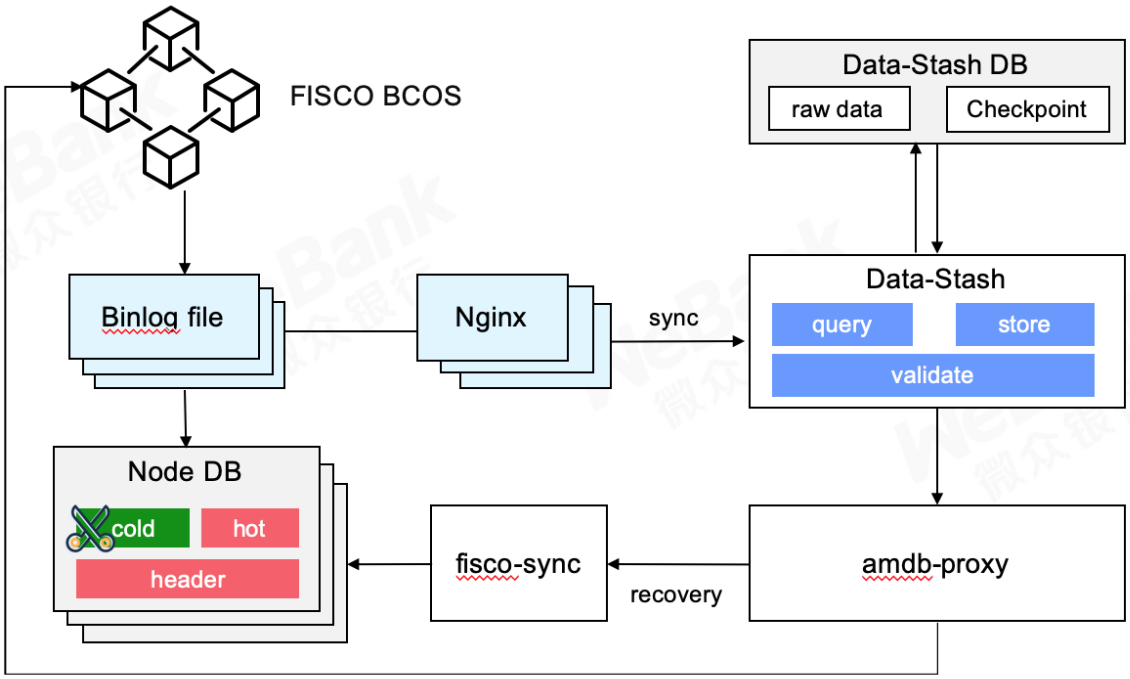
15.3 组件简介

目前，WeBankBlockchain-Data由数据仓库组件(Data-Stash)、数据导出组件(Data-Export)、数据对账组件(Data-Reconcile)三个相互独立、可插拔、可灵活组装的组件构成，后续会根据业务和场景诉求提供更多的功能和解决方案子组件。



15.3.1 WeBankBlockchain-Data-Stash 数据仓库组件

提供FISCO BCOS节点数据扩容、备份和裁剪的能力。 可基于binlog协议同步区块链底层节点数据，支持断点续传，数据可信验证，并提供快速同步机制。

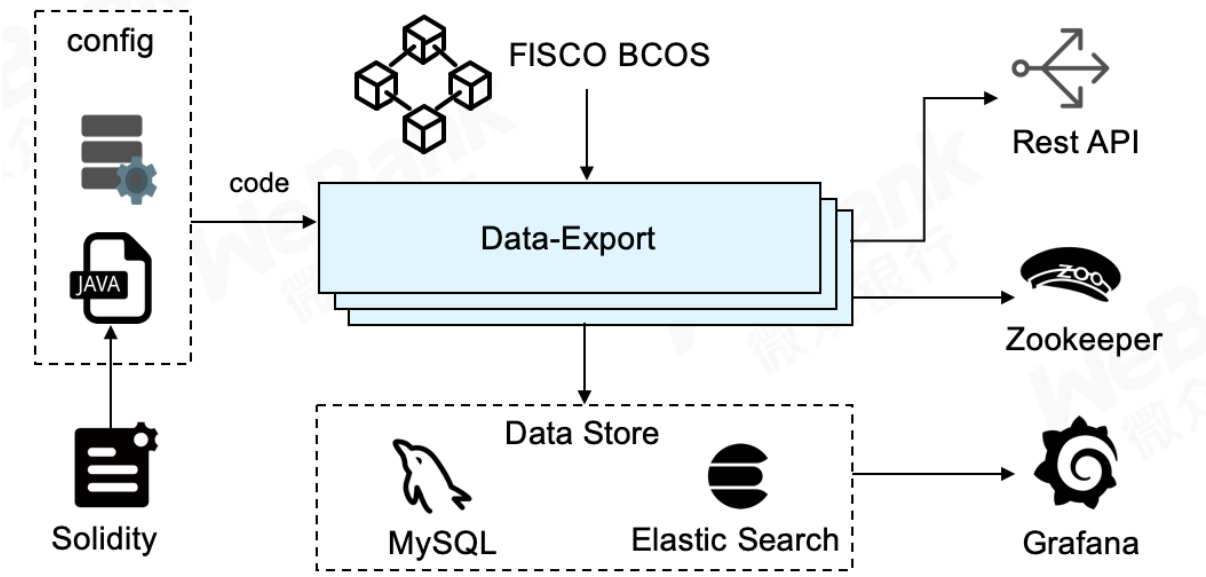


请参考

- Github地址
- Gitee地址
- 文档
- 快速开始

15.3.2 WeBankBlockchain-Data-Export 数据导出组件

支持将链上数据导出到MySQL等结构化存储中，解决区块链数据复杂查询、分析和处理的问题。 只需简单配置、无需开发、即可实时导出个性化的业务数据，实现将裸数据转化为标准化、结构化、有序化、可视化的高价值数据。

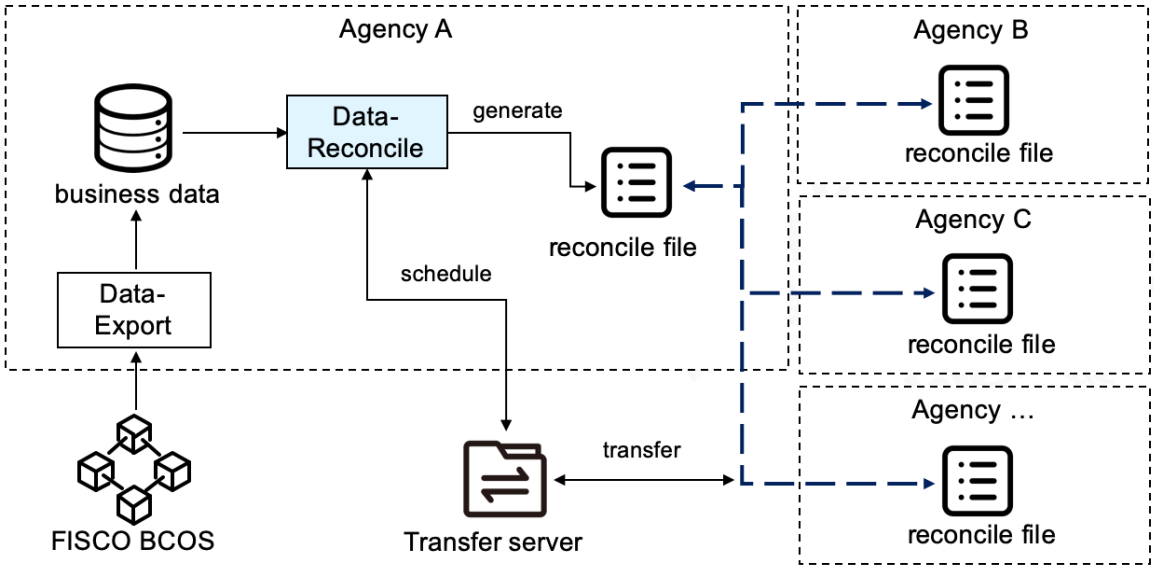


请参考

- Github地址
- Gitee地址
- 文档
- 快速开始

15.3.3 WeBankBlockchain-Data-Reconcile 数据对账组件

提供区块链数据的对账解决方案。 灵活配置、无需开发，支持自定义对账数据和对账格式，支持定时对账和触发对账，对账处理模块可插拔可扩展。



请参考

- Github地址
- Gitee地址
- 文档
- 快速开始

15.4 使用场景

企业级区块链应用存在多元化角色参与，诸如业务角色、运营人员、开发角色、运维角色等。针对区块链数据，每一个特定的角色都有着不同的数据治理诉求。WeBankBlockchain-Data分别从区块链底层节点数据维护、应用数据处理和业务数据应用三个维度，抽象、设计了对应的组件来满足不同角色的对数据治理的需求。

15.4.1 场景1：节点数据维护

数据仓库组件Data-Stash是一款针对区块链节点数据处理的轻量化、高安全、高可用组件，主要面向运维人员和开发人员。

数据备份：Data-Stash可以通过Binlog协议对区块链节点数据进行准实时全量备份，区块链节点可以根据实际情况进行冷热数据裁剪和分离，在确保数据安全可信的基础上，解决节点扩容问题，降低开发和硬件成本。在解决节点扩容问题的同时，可以使得节点“轻装上阵”，不仅能够减少节点空间的开销，而且能够有效提升节点执行交易的性能。

数据同步：对于加入区块链网络的新节点，可以通过Data-Stash，在Fisco Sync工具的配合下，快速同步区块链网络的数据，确保节点以最快的速度参与到区块链网络的“工作”中，降低新节点因等待数据同步而造成的时间浪费。

15.4.2 场景2：应用数据处理

数据导出组件Data-Export 提供了导出的区块链标准数据、自动基于智能合约代码智能分析而生成的定制数据，存储到MySQL和ElasticSearch等存储介质中，主要面向开发人员。

复杂查询与分析：现有区块链对查询功能不太友好，且链上计算非常宝贵，Data-Export支持将链上存储的区块链数据导出到链下的分布式存储系统中。开发者可以基于已导出的区块链系统基础数据，智能合

约部署合约账户、事件和函数等数据，进行二次开发，定制复杂查询和数据分析的逻辑，快速实现业务需求。例如，开发者可对交易明细根据业务逻辑进行统计和关联查询分析，开发各类反洗钱和审计监管报表等等。

区块链数据可视化：Data-Export会自动生成Grafana的配置文件，无需开发，即可实现区块链数据可视化。区块链数据可视化不仅能够作为区块链数据大盘、数据查看、运营分析的工具，同时也可以运用在应用开发、调试、测试阶段，以可见即可得的方式提升研发体验和效率。此外，Data-Export还提供了Restful API供外部系统集成。运维人员可以通过Grafana实时监控业务系统的状态，业务人员可以在集成后的业务后台系统上获得该业务的实时进展。

区块链中间件平台WeBASE的数据导出子系统已经整合了Data-Export，同时，Data-Export也可以独立与区块链底层集成，以灵活地支持业务需求，迄今已在数十个生产系统中稳定、安全运行。

如今，Data-Export作为区块链数据治理的关键组件，以开源形式发布，由社区伙伴通力完善，以适应更多的使用场景、打造更多的功能。

15.4.3 场景3：业务数据应用

在业务层，数据对账是区块链交易系统中最常见的场景之一。基于数个区块链DAPP应用的开发和实践经验，我们封装和开发了数据对账组件Data-Reconcile，提供基于区块链智能合约账本的通用化数据对账解决方案，并提供了一套可动态延展的对账框架，支持定制化开发，主要面向开发人员，为业务人员提供服务。

企业内部对账：Data-Reconcile支持企业内部系统之间的对账，例如区块链链上数据与链下业务系统之间的对账。开发人员可以利用Data-Reconcile快速进行二次开发，将业务系统数据和链上数据进行核算比对，保证了企业内部业务系统数据的可靠和运行安全。

企业间对账：Data-Reconcile可以帮助开发者快速构建跨机构间的对账应用系统。例如，在结算时，A企业定期将自身业务系统交易数据导出为对账文件，发送至与文件存储中心。B企业可借助Data-Reconcile定期拉取A企业对账文件，配合Data-Export，与企业内部的链上数据进行对账处理。Data-Reconcile在保证对账结果可信的同时提升了对账的效率，可实现准实时对账。

综上所述，WeBankBlockchain-Data 是一套稳定、高效、安全的立体化区块链数据治理解决方案，旨在通过提供一系列相互独立、可插拔、可灵活组装的组件，应对和处理区块链海量数据的问题，为用户带来更加便捷、简单、低成本和轻量化的用户体验，从而推动区块链数据治理的发展。

CHAPTER 16

国密使用

标签：国密算法

16.1 1. 系统设计

- 国密支持

16.2 2. 使用方法

区块链网络搭建

- 使用国密算法
- 使用CFCA证书部署节点 > 如何使用国密CFCA证书
- 存储加密
- 运维部署工具 > 部署国密区块链
- 运维部署工具 > 国密操作相关

区块链应用开发

- Web3SDK > SDK国密功能使用
- Python SDK > 国密账户配置

16.3 3. 博客

- 深入浅出FISCO BCOS > 国密特性
- 深入浅出FISCO BCOS > FISCO BCOS国密版部署示例

CHAPTER 17

证书相关

标签：证书

17.1 1. 系统设计

- 节点准入管理介绍
- CA黑白名单介绍

17.2 2. 使用方法

区块链网络搭建

- 获取证书
- 使用CFCA证书部署节点
- 配置CA黑白名单
- 开发部署工具 > 为新节点生成私钥证书
- 运维部署工具 > 操作手册 > 检验证书有效期

区块链应用开发

- 开发第一个区块链应用 > 证书配置
- 控制台2.6+版本 > 证书配置
- 控制台1.x版本 > 证书配置
- Java SDK > 快速开始 > 证书配置
- Java SDK > 配置说明 > 证书配置
- Web3SDK > 配置SDK > 证书配置
- Go SDK > 配置文件
- Node.js SDK > 证书配置

- Python SDK > 配置说明 > 证书配置

17.3 3. 问题排查

- 证书问题
- 连接失败

17.4 4. 博客

- FISCO BCOS上使用第三方CA证书底层节点部署实操
- FISCO BCOS证书说明
- FISCO BCOS的TLS1.2握手流程

CHAPTER 18

JDK版本

标签：JDK

18.1 1. JDK版本要求

- 命令行交互控制台：
  - 控制台2.6+：JDK1.8 至 JDK14
  - 控制台1.x：大于等于1.8，推荐使用OracleJDK。
- Java SDK: JDK1.8 至 JDK14
- Web3SDK: 大于等于1.8，推荐使用OracleJDK。
- WeBASE: JDK 1.8 至JDK 14
- 区块链浏览器：JDK8或以上版本

18.2 2. 安装方法

- Java SDK > 快速入门 > 安装环境

18.3 3. 问题排查

- 连接失败 > 检查JDK版本

CHAPTER 19

整体架构

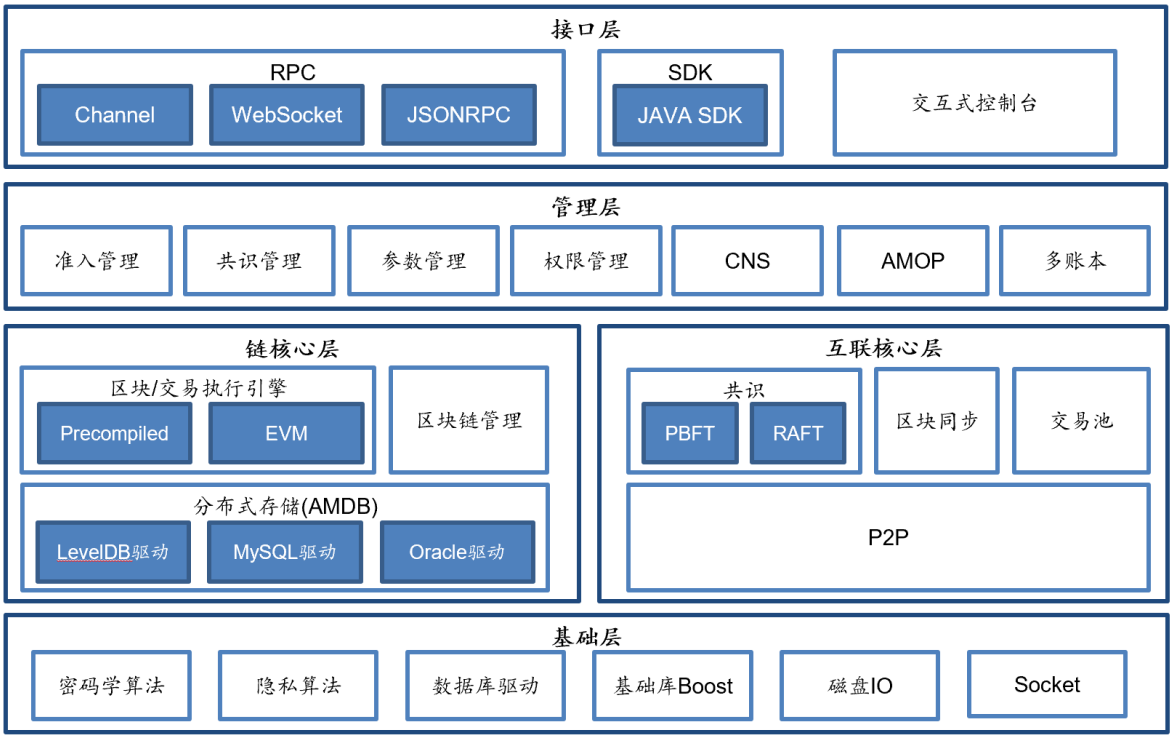
标签：架构 强扩展性 模块设计

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)。



19.1 群组架构

标签：群组架构 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。

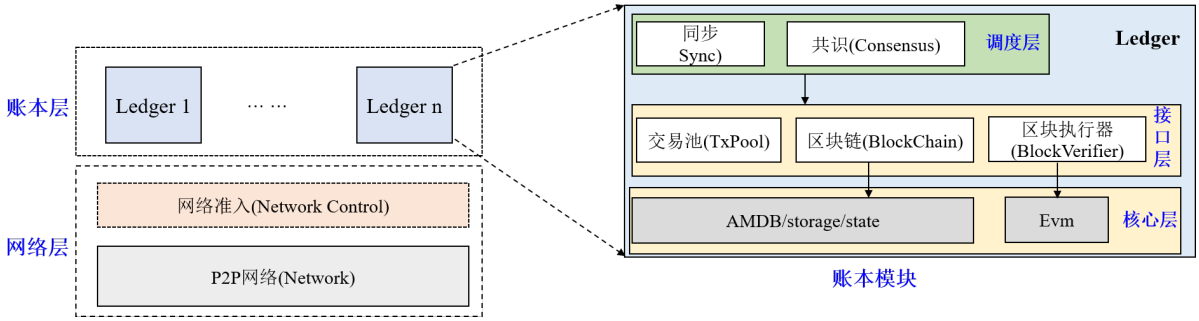
注解： 举个例子:

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。

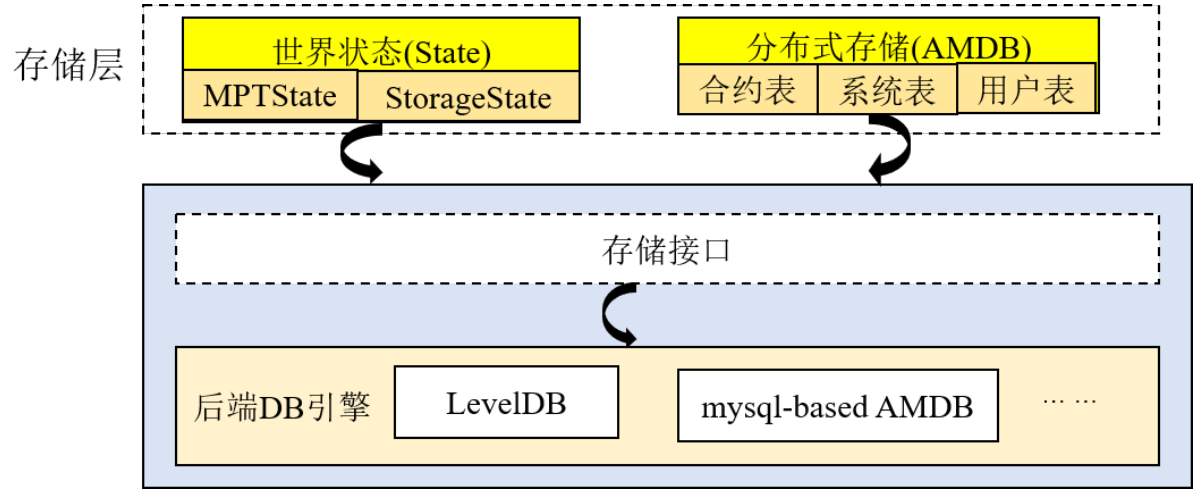


群组间数据隔离，每个群组独立运行各自的共识算法，不同群组可使用不同的共识算法。每个账本模块自底向上主要包括核心层、接口层和调度层三层，这三层相互协作，FISCO BCOS可保证单个群组独立健壮地运行。

19.1.1 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。



19.1.2 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。

19.1.3 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。

19.2 交易流

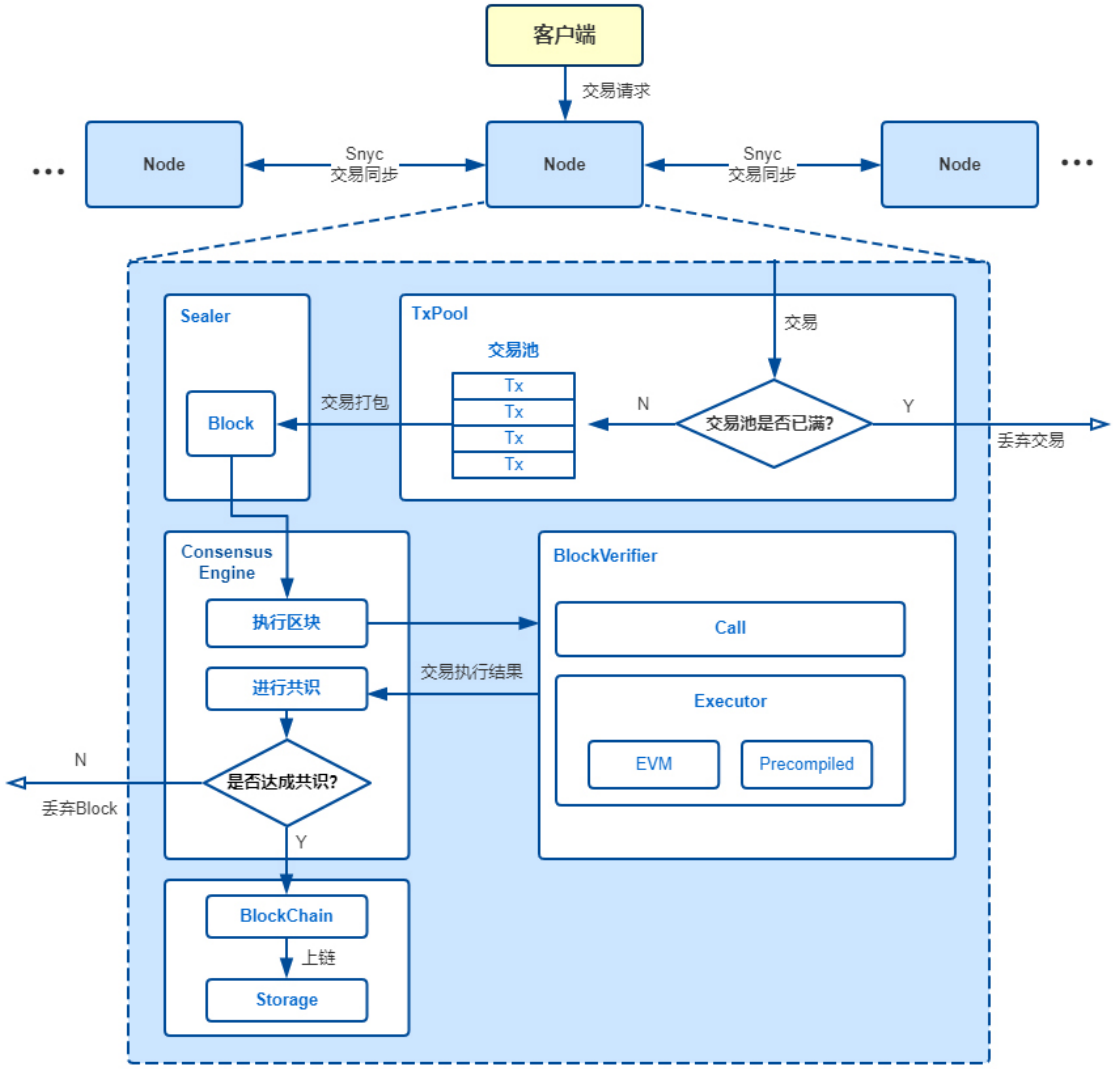
标签：交易流 架构

19.2.1 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。

19.2.2 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

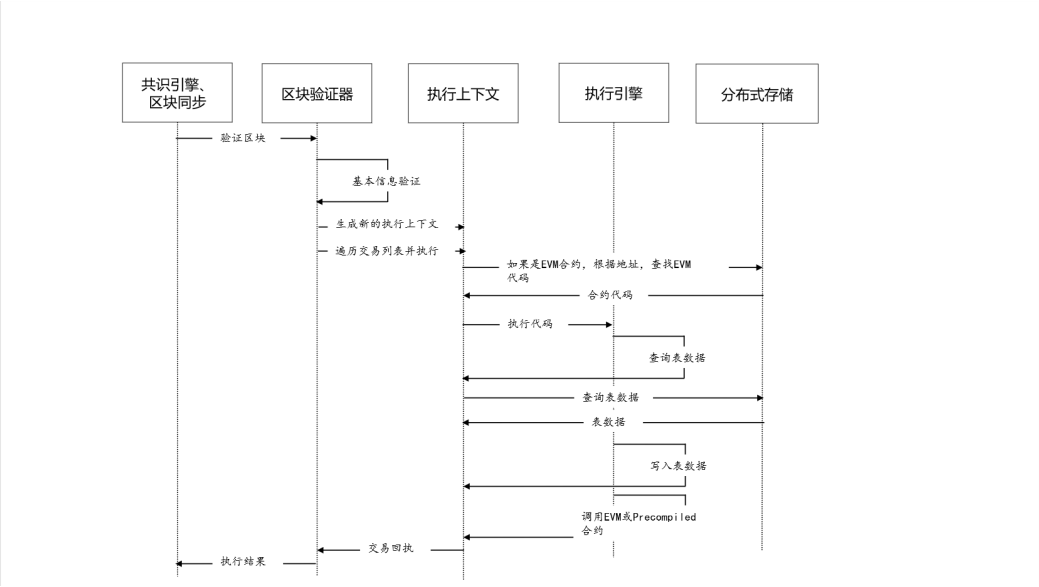
主要关系如下：

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。

19.2.3 3 方案流程

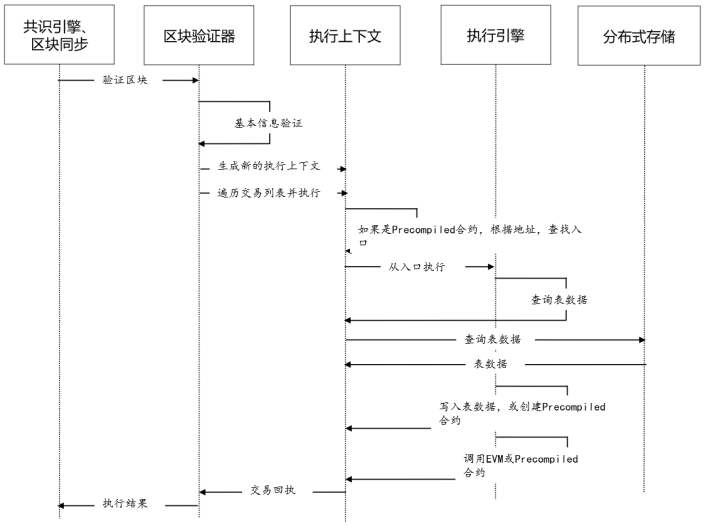
3.1 合约执行流程

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：


共识引擎、
区块同步
区块验证器
执行上下文
执行引擎
分布式存储
验证区块
基本信息验证
生成新的执行上下文
遍历交易列表并执行
如果是Precompiled合约，根据地址，查找入口
从入口执行
查询表数据
查询表数据
表数据
写入表数据，或创建Precompiled合约
调用EVM或Precompiled合约
交易回执
执行结果

CHAPTER 20

模块设计

标签：模块设计 设计方案

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。

20.1 共识算法

标签：共识算法 设计方案 可扩展的共识框架

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

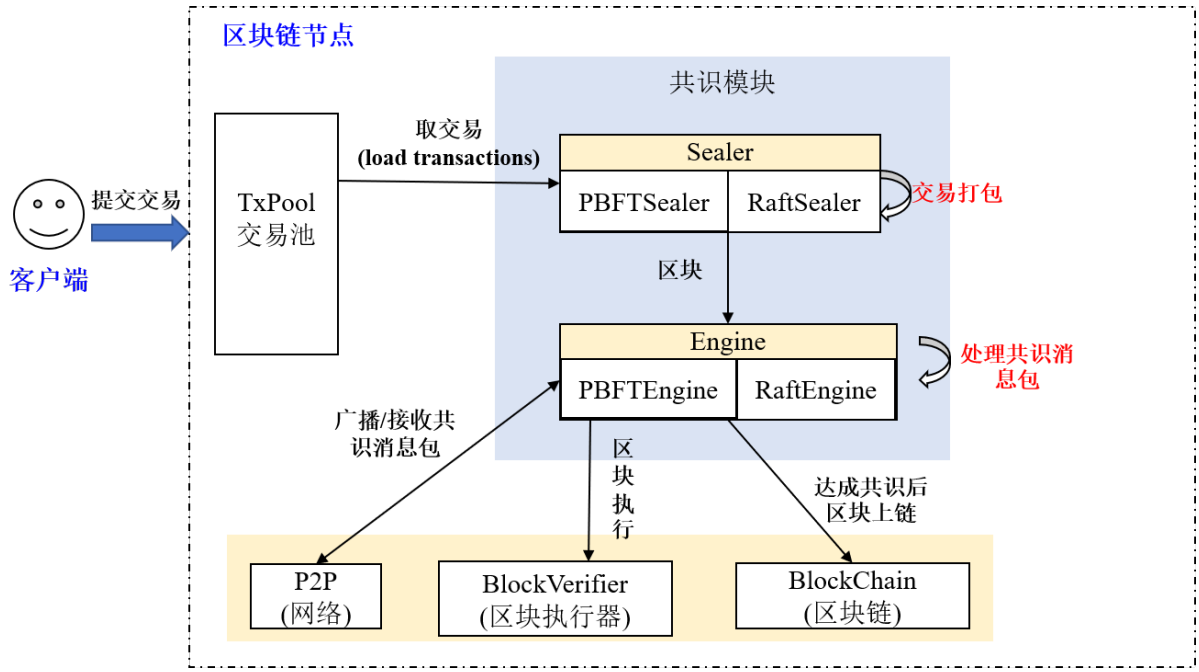
FISCO BCOS基于多群组架构实现了插件化的共识算法，不同群组可运行不同的共识算法，组与组之间的共识过程互不影响，FISCO BCOS目前支持PBFT(Practical Byzantine Fault Tolerance)和Raft(Replication and Fault Tolerant)两种共识算法：

- **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。

20.1.1 框架

标签：共识算法 设计方案

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。

20.1.2 PBFT基础流程

标签：PBFT 共识算法 设计方案

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

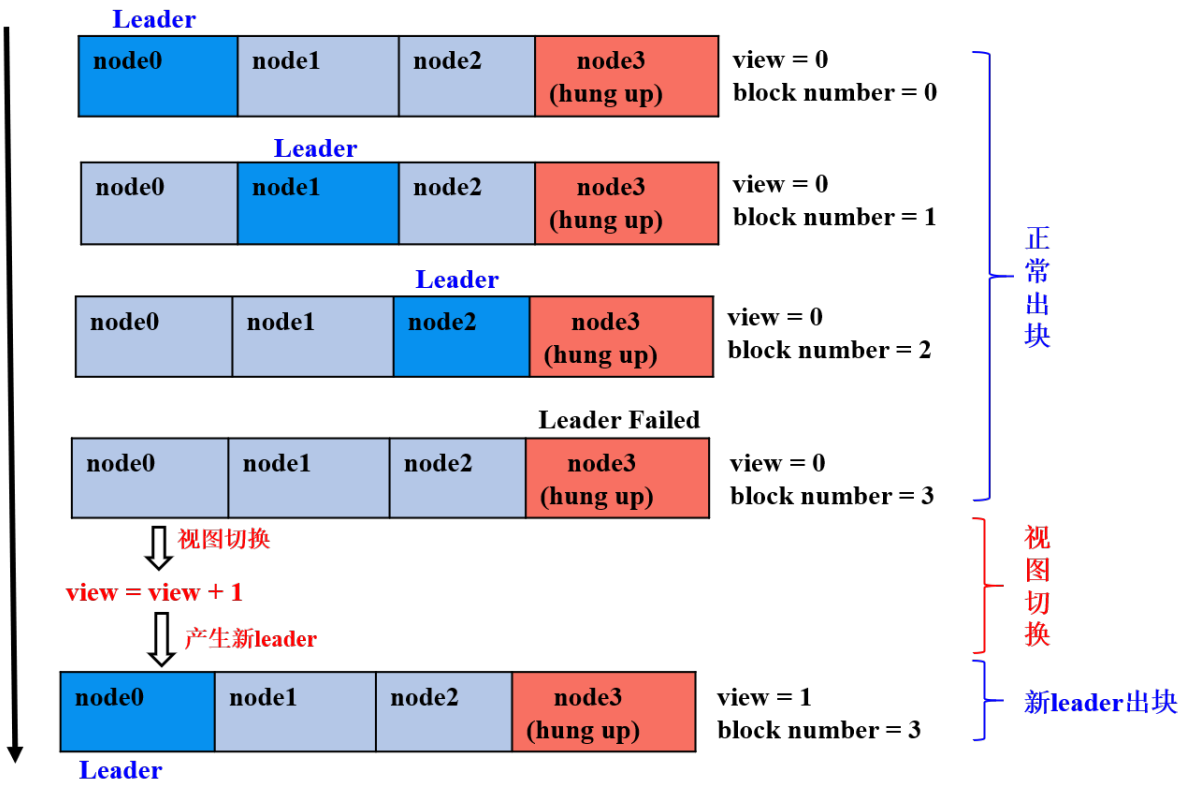
1.3 视图(view)

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：

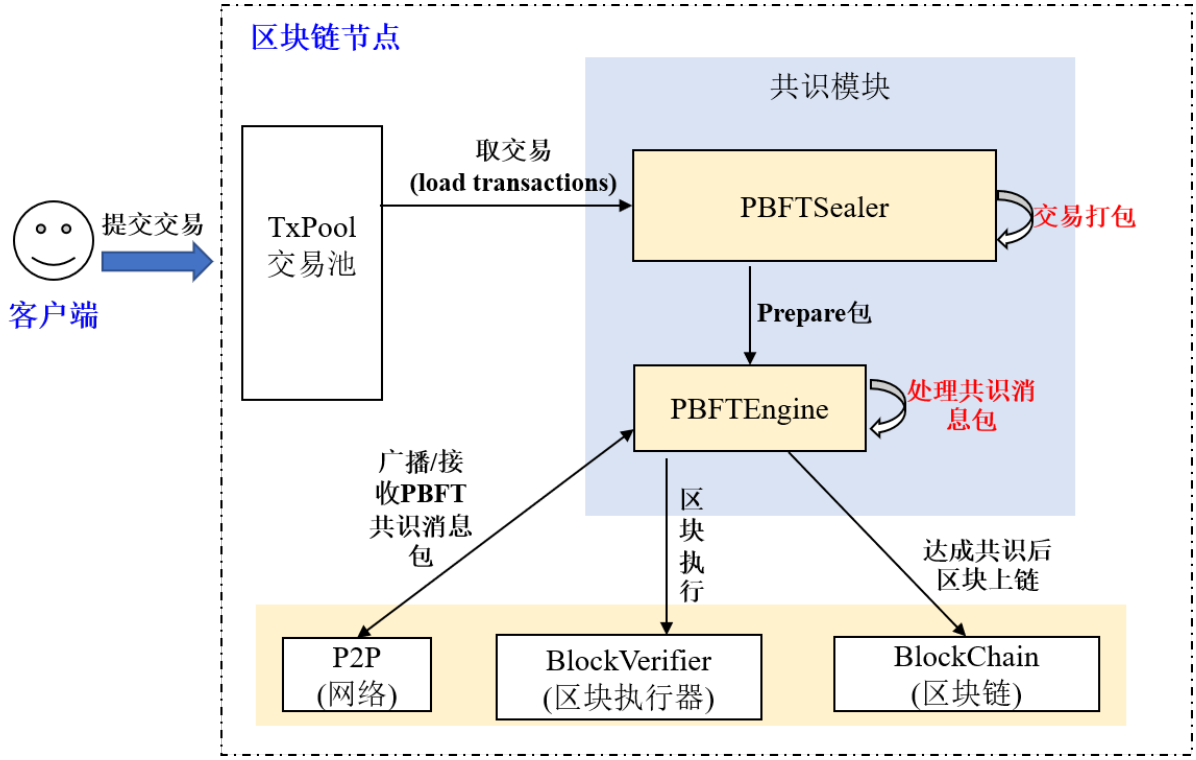
- **PrepareReqPacket**：包含区块的请求包，由leader产生并向所有Replica节点广播，Replica节点收到Prepare包后，验证PrepareReq签名、执行区块并缓存区块执行结果，达到防止拜占庭节点作恶、保证区块执行结果的最终确定性的目的;
- **SignReqPacket**：带有区块执行结果的签名请求，由收到Prepare包并执行完区块的共识节点产生，SignReq请求带有执行后区块的hash以及该hash的签名，分别记为SignReq.block_hash和SignReq.sig，节点将SignReq广播到所有其他共识节点后，其他节点对SignReq(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:



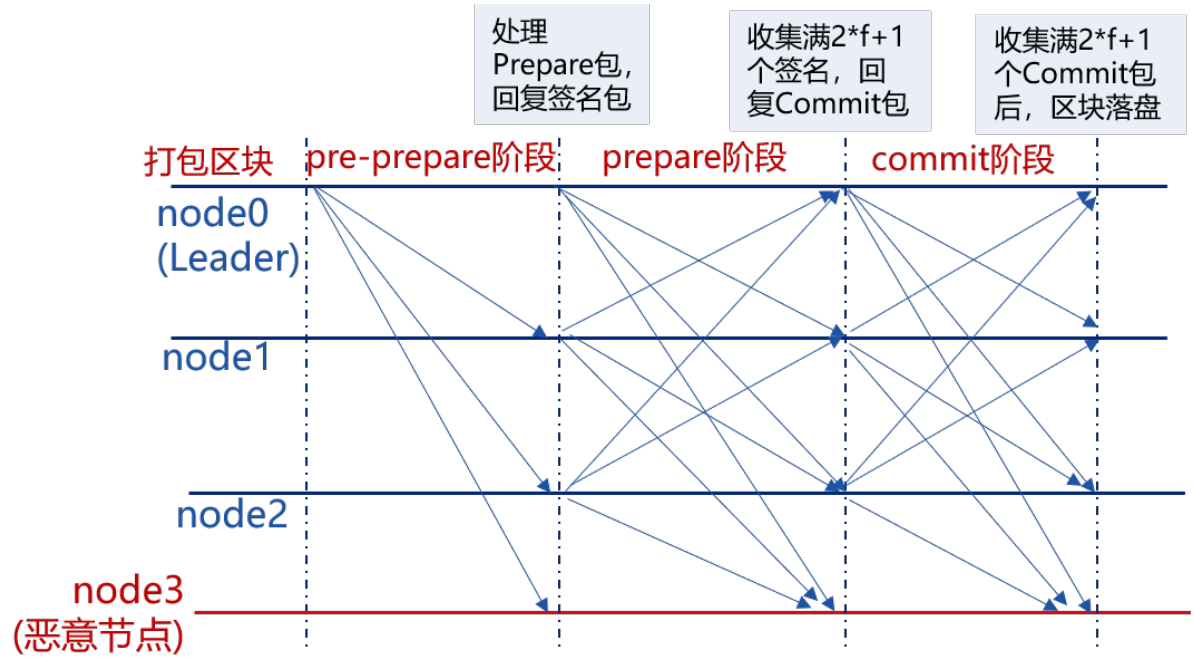
PBFT共识主要包括两个线程:

- 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。

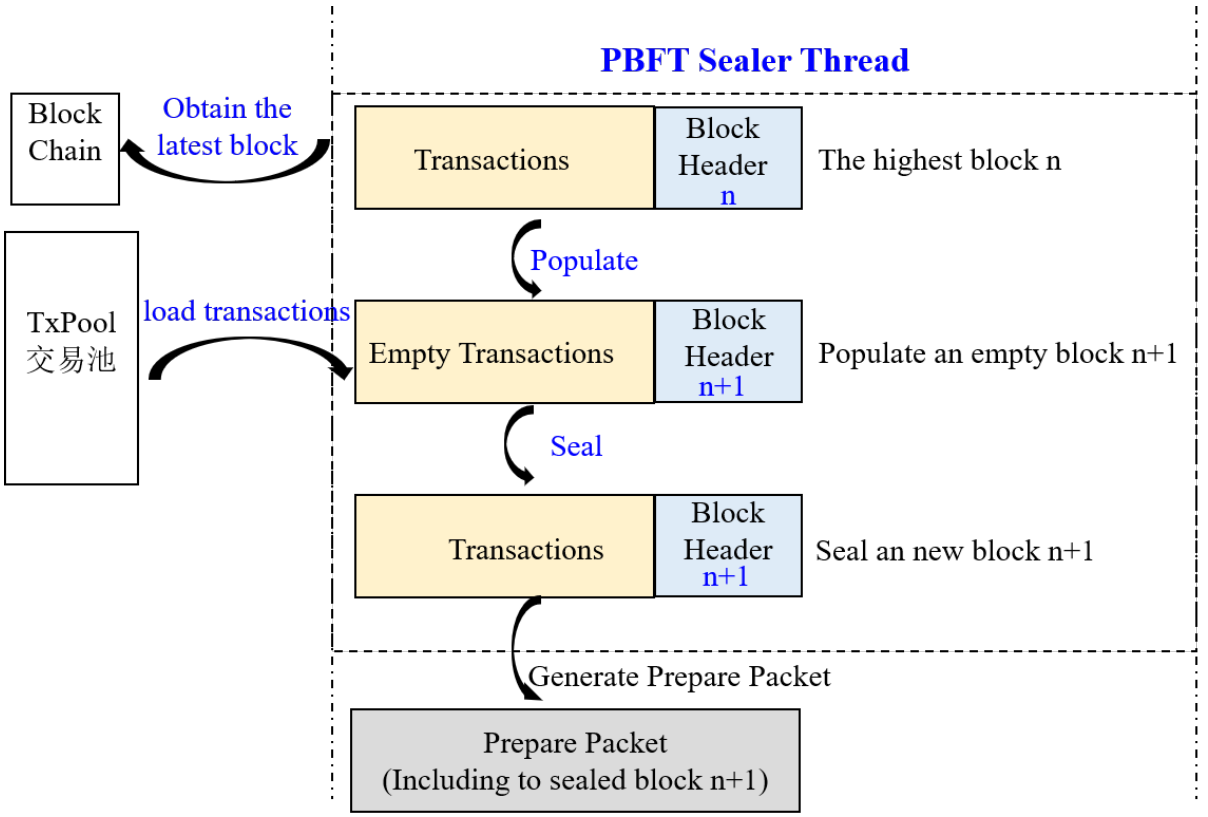


下图详细介绍了PBFT各个阶段的具体流程：

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。

20.1.3 PBFT网络优化

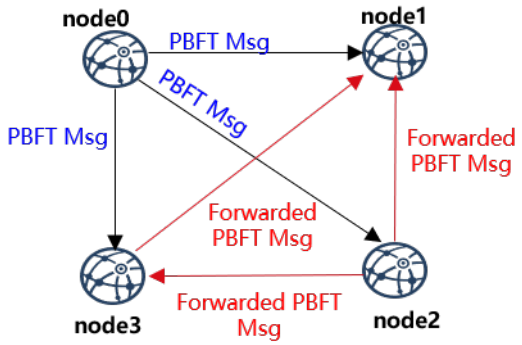
标签：PBFT 共识算法 网络优化 设计方案

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

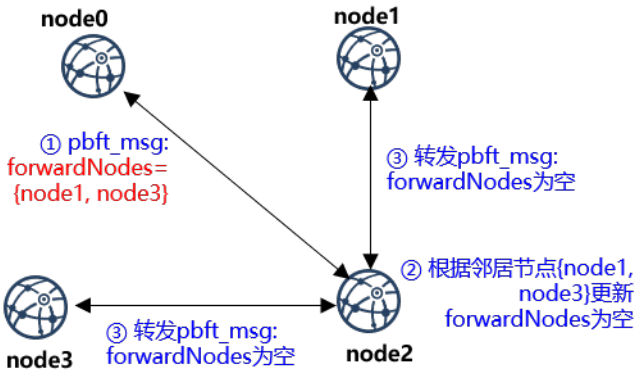
为了保证节点断连情况下共识消息包能到达所有节点，FISCO BCOS PBFT共识模块采用了消息转发机制，优化前的消息转发机制如下：

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：



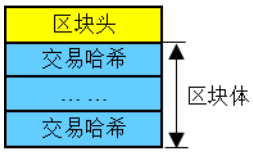
- node0向{node1, node2, node3}发送PBFT消息，发现{node1, node3}不在连接列表内，则将PBFT消息msg的forwardNodes字段设置为{node1, node3}，并将其转发给node2；
- node2收到node1的PBFT消息后，判断forwardNodes字段不为空，则遍历邻居节点列表{node1, node3}，并将邻居节点从forwardNodes中移除；
- node2向node1和node3转发更新后的PBFT消息msg；
- node1和node3收到msg后，判断forwardNodes字段为空，认为该消息已经到达了所有节点，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。

20.1.4 Raft

标签：Raft 共识算法 设计方案

---

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:

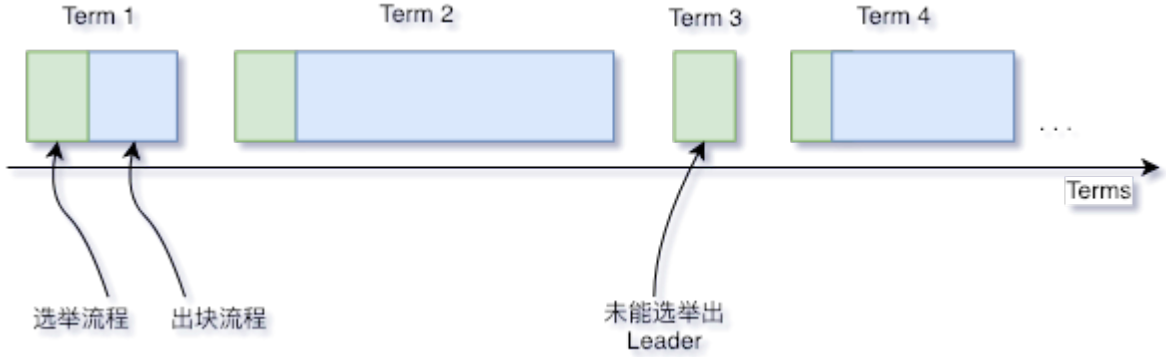
- **Leader**：主要负责与外界交互，由Follower节点选举而来，在每一次共识过程中有且仅有一个Leader节点，由Leader全权负责从交易池中取出交易、打包交易组成区块并将区块上链；
- **Follower**：以Leader节点为准进行同步，并在Leader节点失效时举行选举以选出新的Leader节点；
- **Candidate**：Follower节点在竞选Leader时拥有的临时身份。

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：

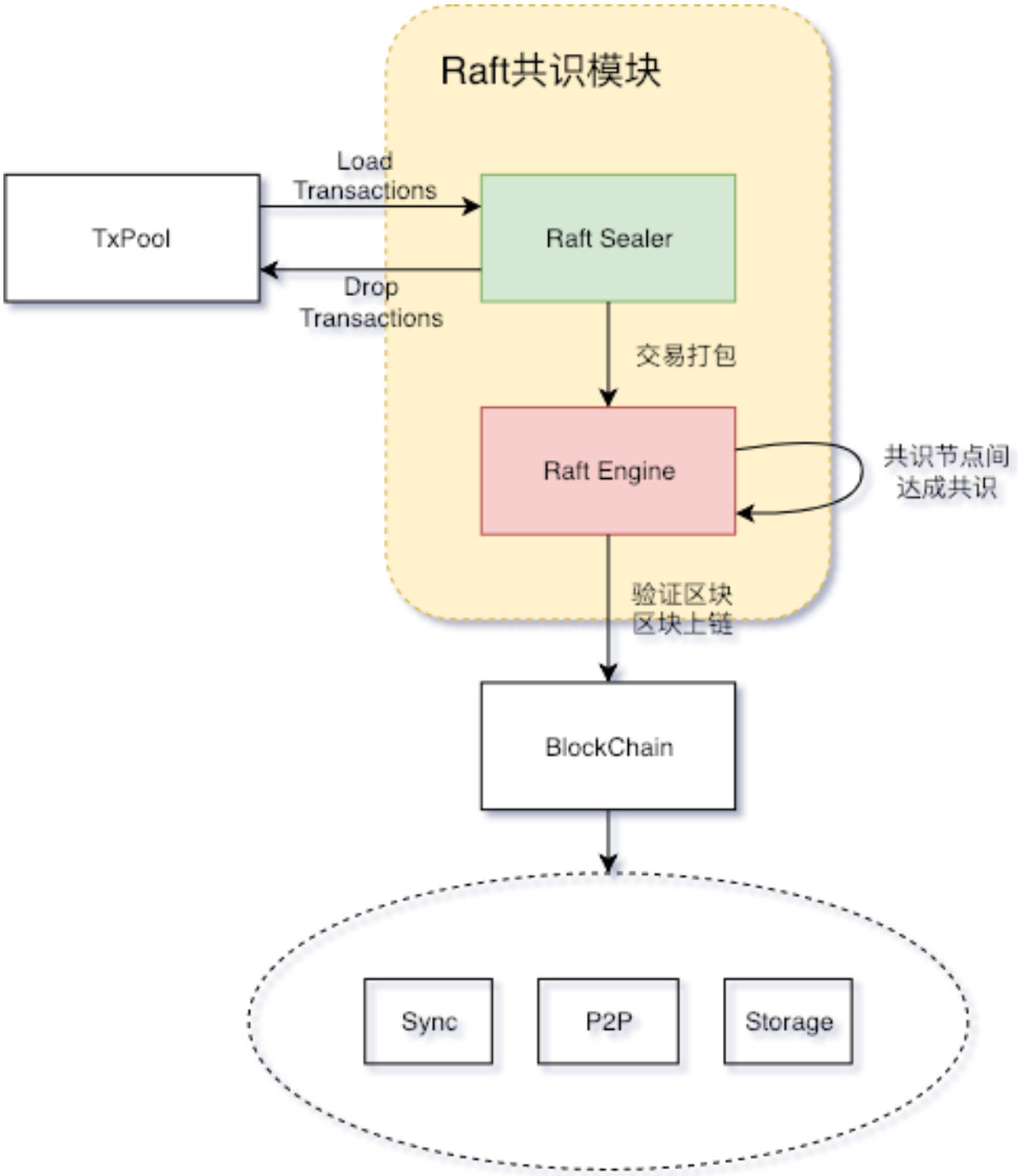
- **VoteReq**：投票请求，由Candidate节点主动发出，用于向网络中其他节点请求投票以竞选Leader；
- **VoteResp**：投票响应，在节点收到投票请求后，用于对投票请求进行响应，响应内容为同意或拒绝该投票请求；
- **Heartbeat**：心跳，由Leader节点主动周期发出，其作用有两个：(1) 用于维护Leader节点身份，只要Leader能够一直正常发送心跳且被其他节点响应，Leader身份就不会发生变化；(2) 区块数据复制，当Leader节点成功打包一个区块后，会将区块数据编码至心跳中以将区块进行广播，其他节点在收到该心跳后会解码出区块数据并将区块放入自己的缓冲区中；
- **HeartbeatResp**：心跳响应，在节点收到心跳后，用于对心跳进行响应，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：

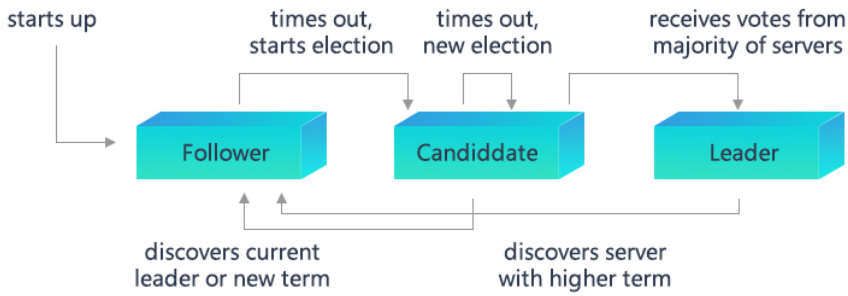


- Raft Sealer：负责从交易池取出交易并打包成区块，并发送至Raft Engine进行共识。区块上链后，Raft Sealer负责从交易池中删除已上链交易；
- Raft Engine：负责在共识节点进行共识，将达成共识的区块上链。

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。

20.1.5 rPBFT

标签：`rPBFT` `共识算法`

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。

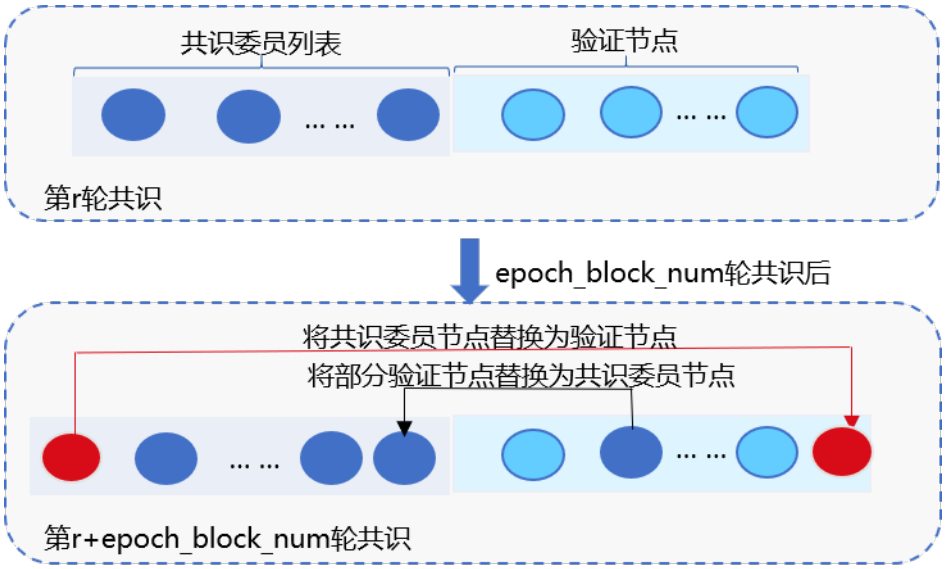
综上所述，FISCO BCOS v2.3.0提出了rPBFT共识算法，旨在保留BFT类共识算法高性能、高吞吐量、高一致性、安全性的同时，尽量减少节点规模对共识算法的影响。

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：

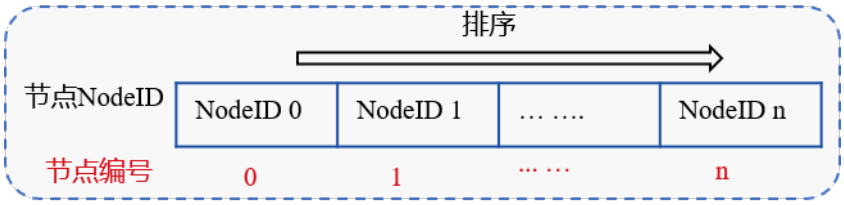
- `epoch_sealer_num`：每轮共识过程中参与共识的节点数目，可通过控制台发交易方式动态配置该参数
- `epoch_block_num`：共识节点替换周期，为防止选取的共识节点联合作恶，rPBFT每出epoch_block_num个区块，会替换一个共识节点，可通过控制台发交易的方式动态配置该参数

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。

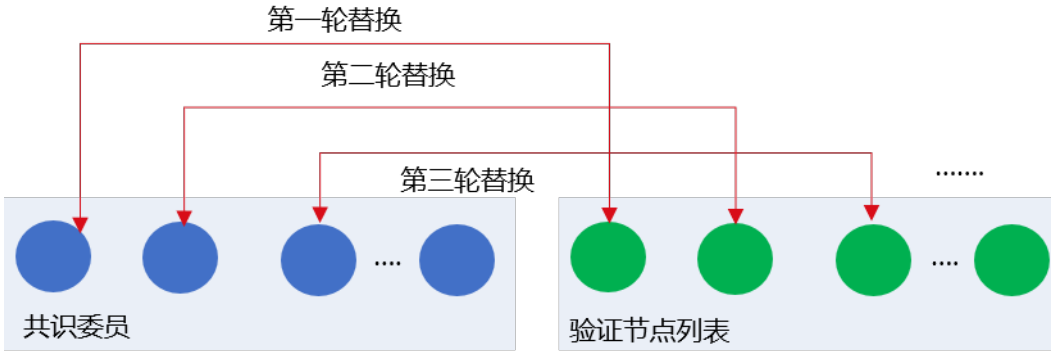
共识委员节点运行PBFT共识算法

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**

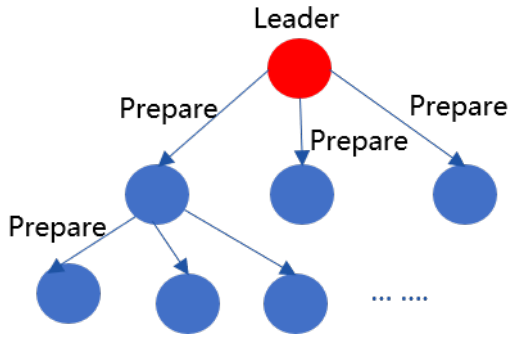
rPBFT算法分析

- 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:

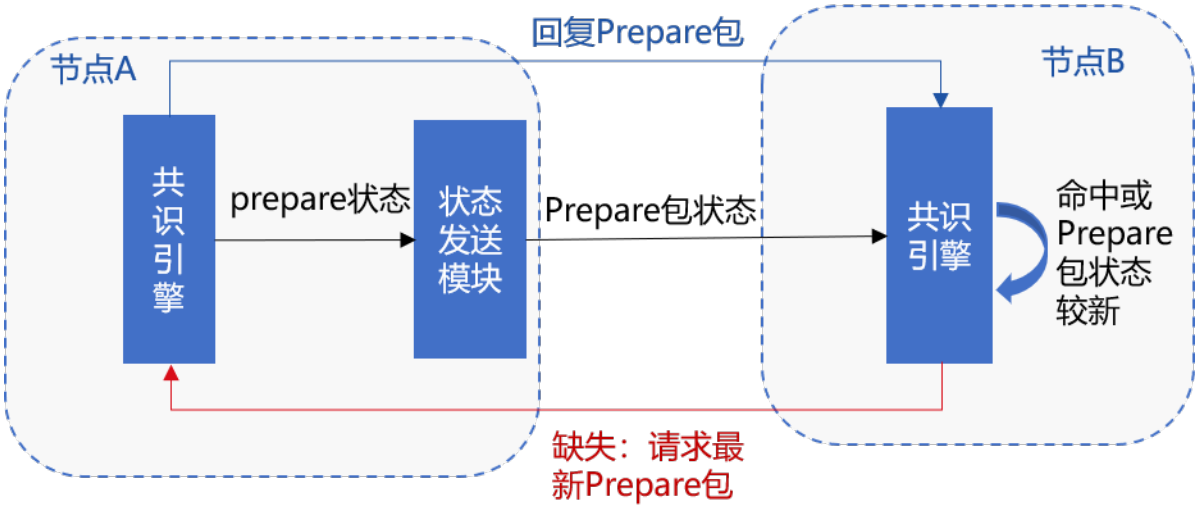
- 传播速度比gossip快，无冗余消息包
- 分而治之，每个节点出带宽为O(1)，可扩展性强

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

20.2 对等网络

标签：对等网络 P2P模块 点对点网络 状态同步 AMOP

20.2.1 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。

20.2.2 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

20.2.3 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

20.2.4 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

20.2.5 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

20.2.6 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

20.2.7 AMOP 消息转发流程

单播时序图

注解:

- 单播是指如果有多个 Subscriber 订阅同一个 topic，节点则随机选择一个 Subscriber 推送消息
- 消息发布者和消息订阅者需要选择同一个 topic
- Subscriber 接收到消息之后的回包是由 sdk 自动发送，不需要用户自己处理，该回包仅仅表示 Subscriber 成功收到消息
- 如果 Publisher 在推送消息之前，没有对应的订阅者，那么 Publisher 将会收到错误码 *100*，表示网络中没有可用节点

多播时序图

注解:

- 多播是指如果有多个 Subscriber 订阅同一个 topic，节点则向所有的 Subscriber 推送消息
- 只要网络正常，即使没有 Subsciber 接收消息， Publisher 也可以收到节点消息推送成功的响应包

带身份验证的单播时序图

注解:

- 身份验证： Publisher 在推送消息的时候，只给满足身份条件的订阅者推送消息
- Publisher 拥有公钥 P，同时监听 #!$TopicNeedVerify_hello 和 #!$PushChannel_#!$TopicNeedVerify_hello 两个 topic
- Subscriber1 拥有私钥 S，同时监听 #!$TopicNeedVerify_hello 和#!$VerifyChannel_#!$TopicNeedVerify_hello_{uuid} 两个 topic
- 时序图中所有的 !$ 前都缺少符号 #， mermaid 不支持该符号转义
- 节点0 给 Publisher 发送的 0x37 消息包中除了 topic：#!$TopicNeedVerify_hello，还有 topicForCert：#!$VerifyChannel_#!$TopicNeedVerify_hello_{uuid} 以及 NodeID
- Publisher 收到 0x37 消息后回包，包中使用的 topic 为：#!$VerifyChannel_#!$VerifyChannel_#!$TopicNeedVerify_hello_
- 身份验证通过之后，推送消息使用的 topic 为 !$TopicNeedVerify_hello
- 带身份验证的多播在验证身份的流程上同带身份验证的单播一致。不同点在于身份验证通过之后，单播推送消息是一对一，而多播推送消息是一对多

通信协议使用的数据结构:

- 0x37 消息类型中 content 数据结构：

```
{"nodeId":
↪"2bb4562f4f4b69e2c5156510da4beddfca548403eb7cea49bd56daed46de31e4d78a44fdfa051170c64f0233db
↪","topic":"#!$TopicNeedVerify_hello","topicForCert":"#!$VerifyChannel_#!
↪$TopicNeedVerify_hello_92be6ce4dbd311eaae5a983b8fda4e0e"}
```

- 随机数单播中的 content 数据结构：

```
{"randValue":"14131f50ef730219d48e1f9c441db871c","topic":"hello"}
```

- 随机数签名回包中 content 数据结构：

```
{"signature":"vU/Vzqn4MiP0nO1xN+M5TOk/YcyFtY/
↪TLrgU38jFdooN66r3TzNVKEBpkNId8gCuAeNpNPCo8vmTV3dcs/Xj1AE="}
```

- 0x38 消息类型中 content 数据结构：

```
{"checkResult":0,"nodeId":
↪"2bb4562f4f4b69e2c5156510da4beddfca548403eb7cea49bd56daed46de31e4d78a44fdfa051170c64f0233db
↪","topic":"#!$TopicNeedVerify_hello"}
```

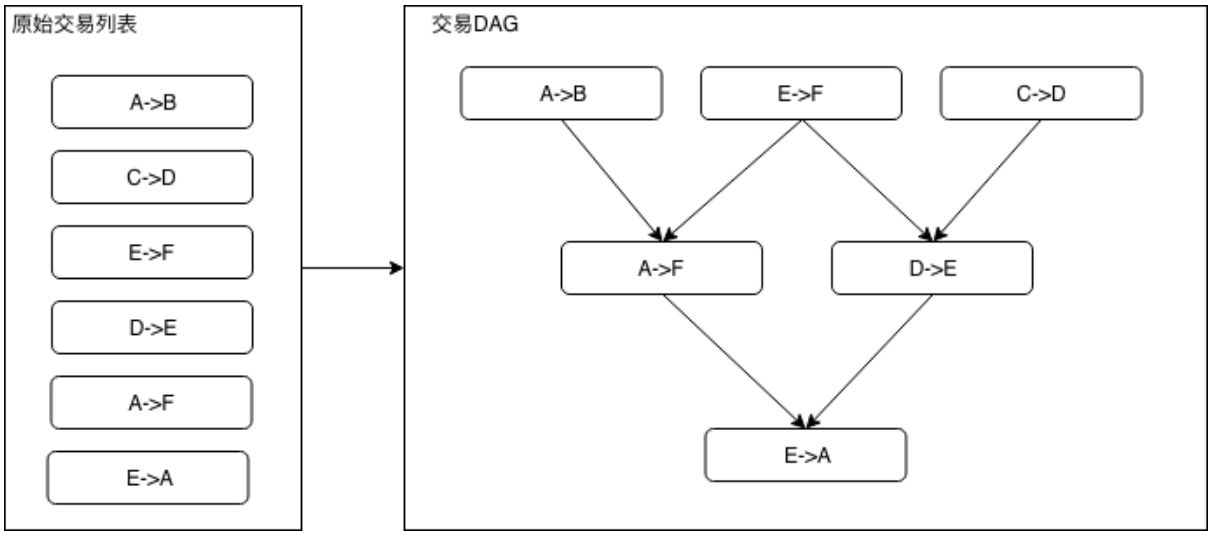
20.3 交易并行

标签：交易并行 DAG

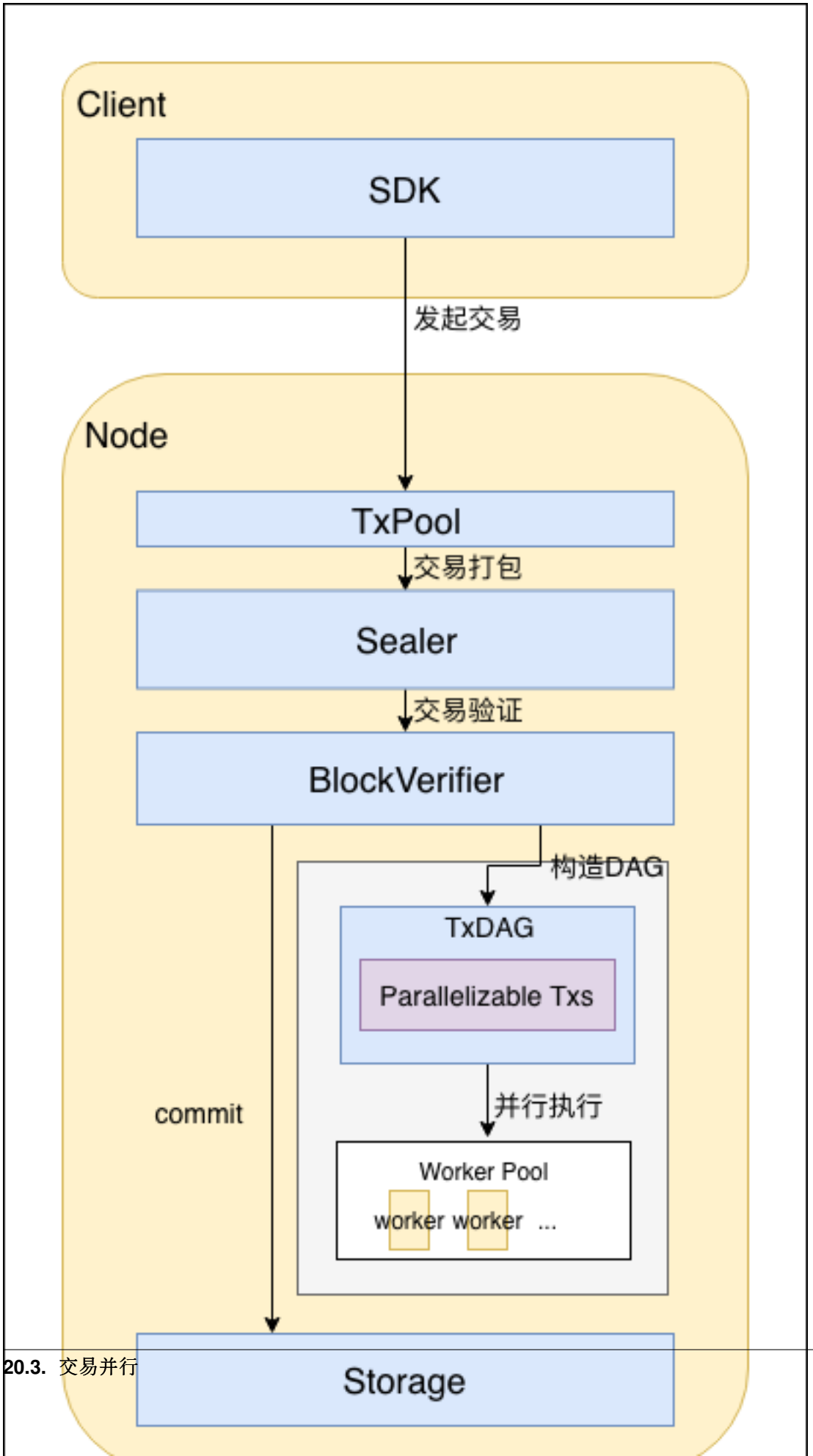
20.3.1 1 名词解释

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。



20.3.2 2 模块架构

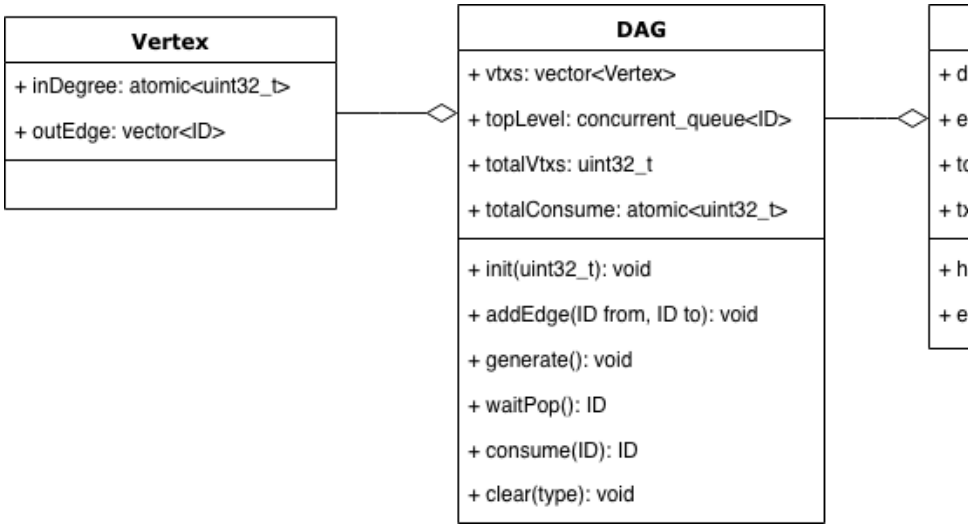

Client
SDK
发起交易
Node
TxPool
交易打包
Sealer
交易验证
BlockVerifier
构造DAG
TxDAG
Parallelizable Txs
并行执行
commit
Worker Pool
worker worker ...
Storage

- 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。

20.3.3 3 重要流程

3.1 交易DAG构建

3.1.1 DAG数据结构



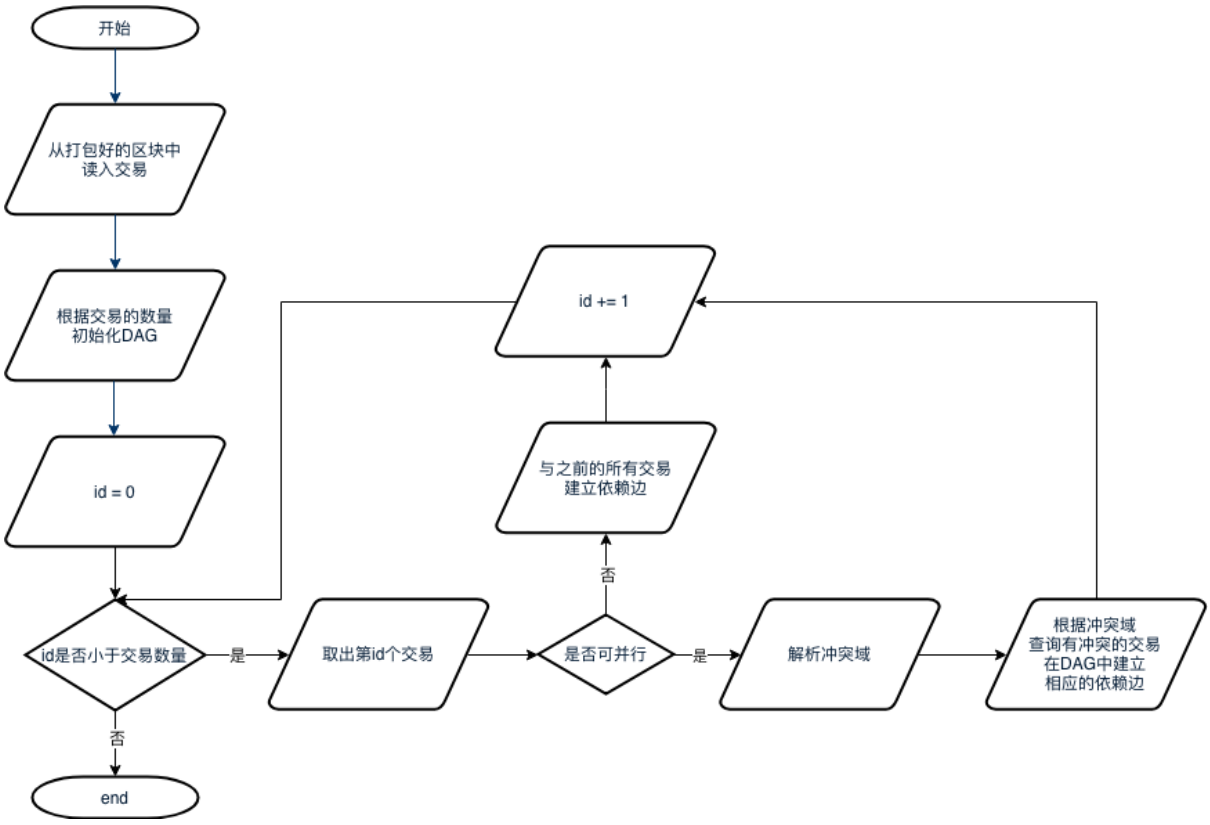
方案中所用到的DAG数据结构如下：
其中：

- 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

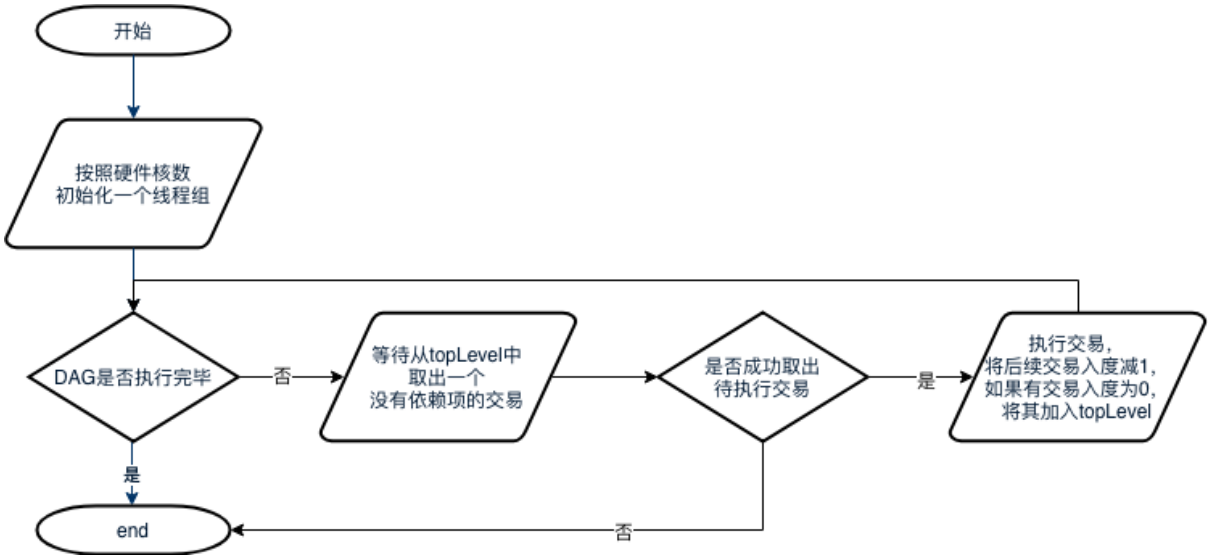
流程如下：



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。

20.4 数据同步

标签：数据同步 交易同步 区块同步

本节介绍节点间的同步机制以及同步优化策略。

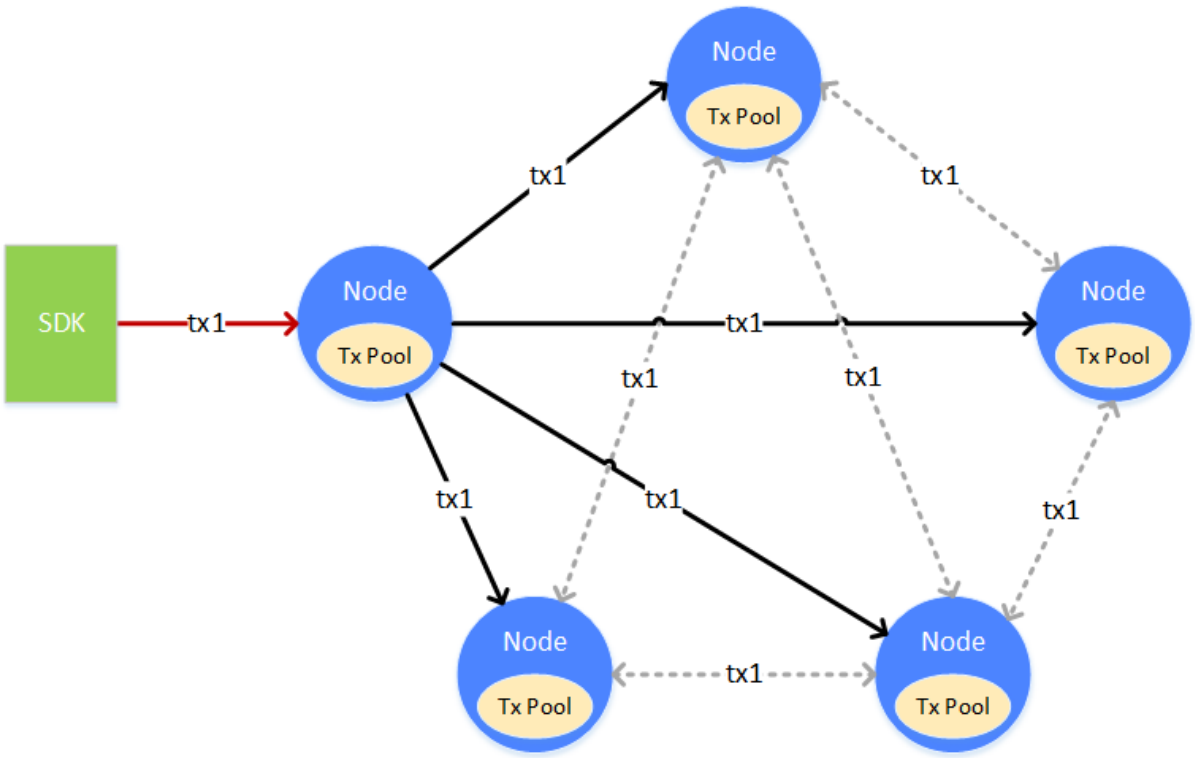
20.4.1 同步基础流程

标签：数据同步 广播策略 状态同步

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

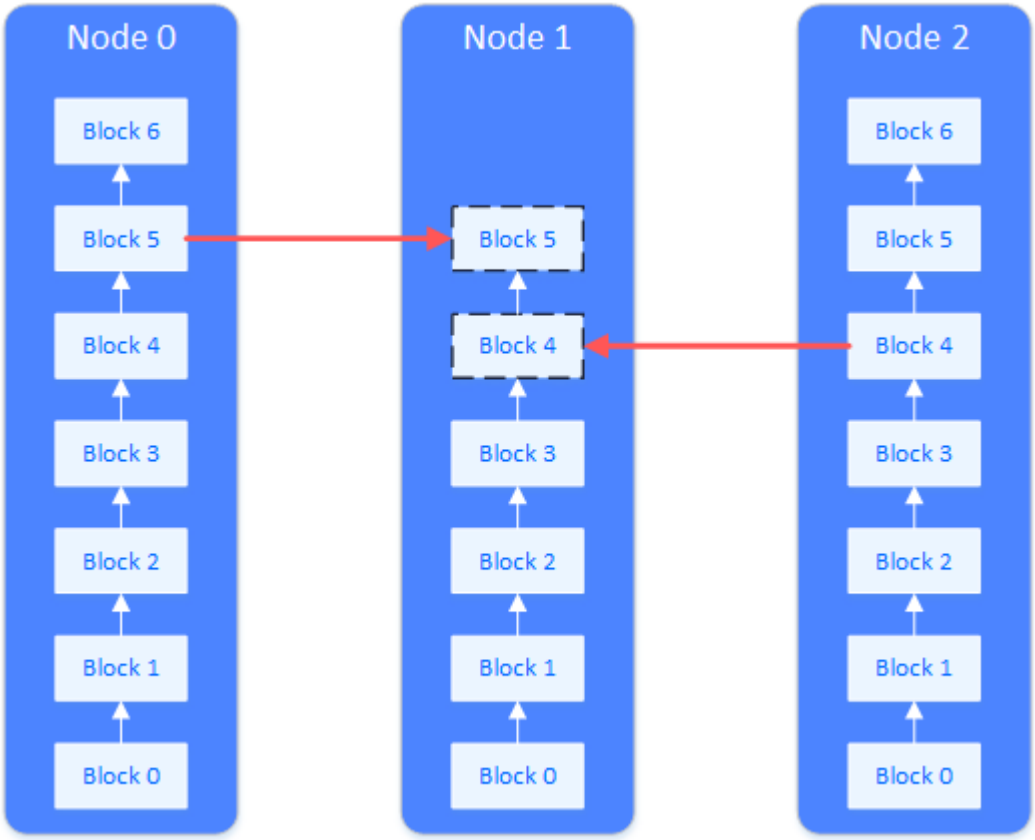
如果每个节点都没有限制的转发/广播收到的交易，带宽将被占满，出现交易广播雪崩的问题。为了避免交易广播的雪崩，FISCO BCOS根据经验，选择了较为精巧的交易广播策略。在尽可能保证交易可达性的前提下，尽量的减少重复的交易广播。

- 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。

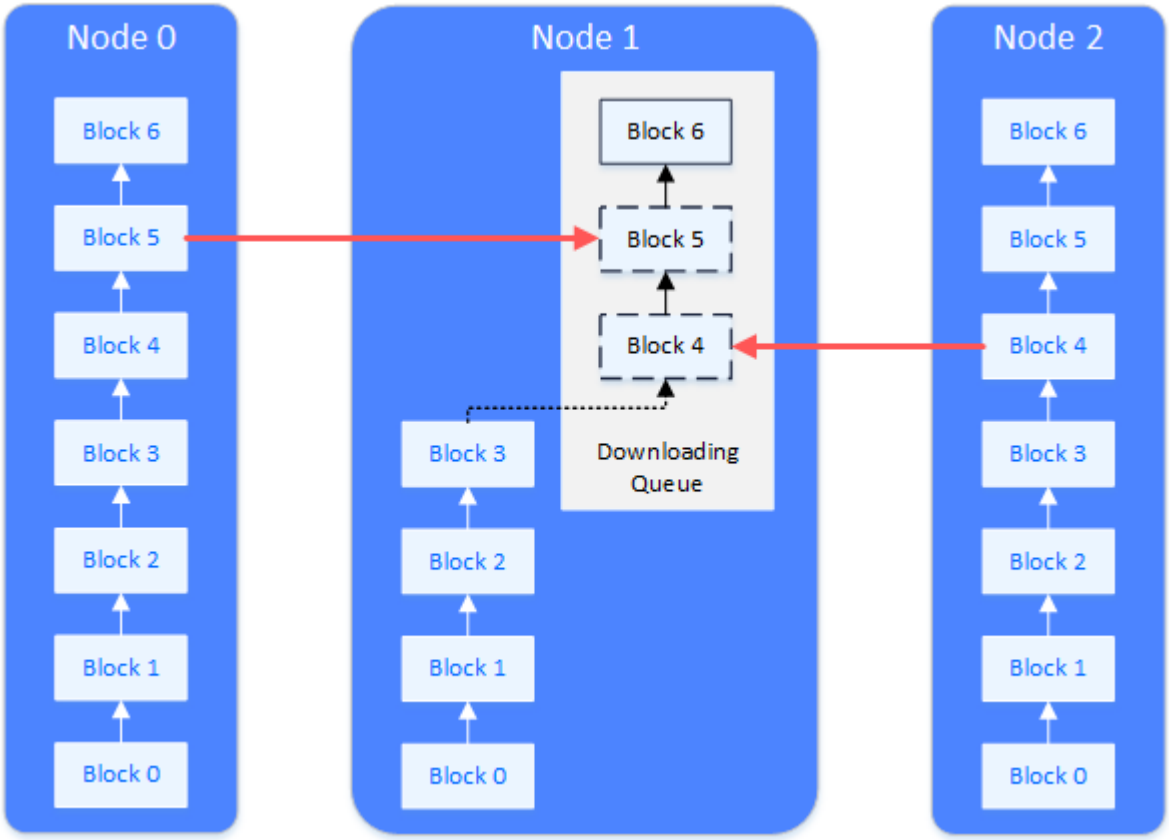


在一个全新的节点加入到区块链上，或一个已经断网的节点恢复了网络时，此节点的区块落后于其它节点，状态不是最新的。此时就需要进行状态同步。如图，需要状态同步的节点（Node 1），会主动向其它节点请求下载区块。整个下载的过程会将下载的负载分散到多个节点上。

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

20.4.2 区块同步优化

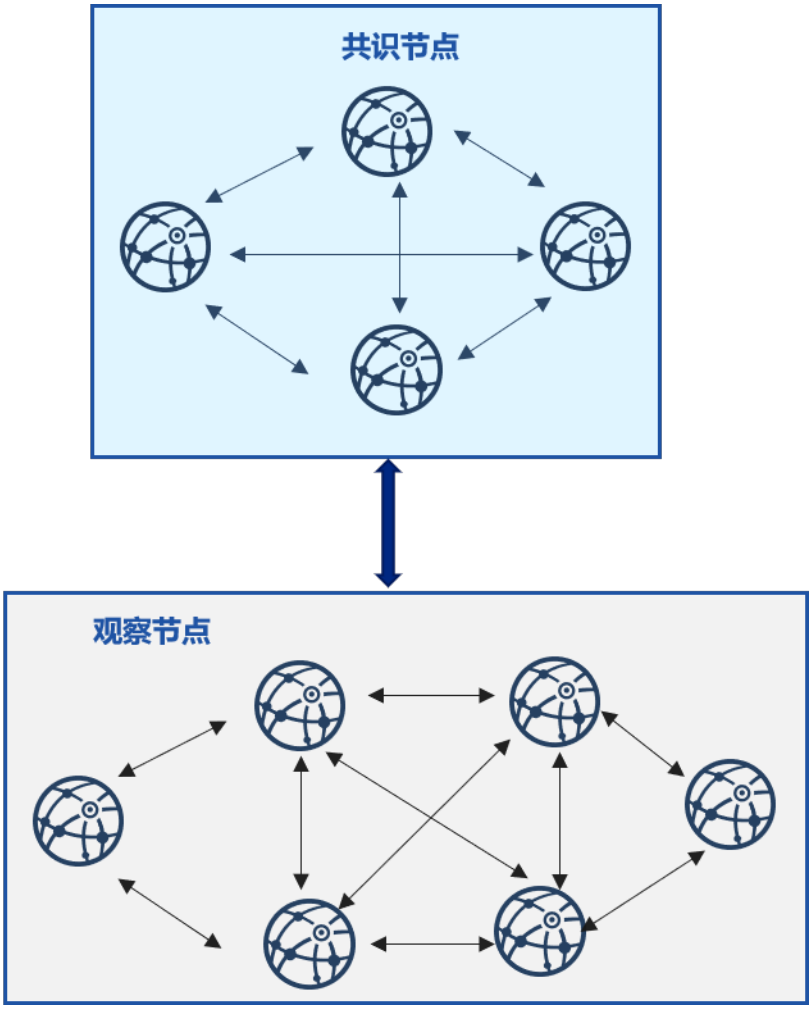
标签：数据同步 同步优化

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

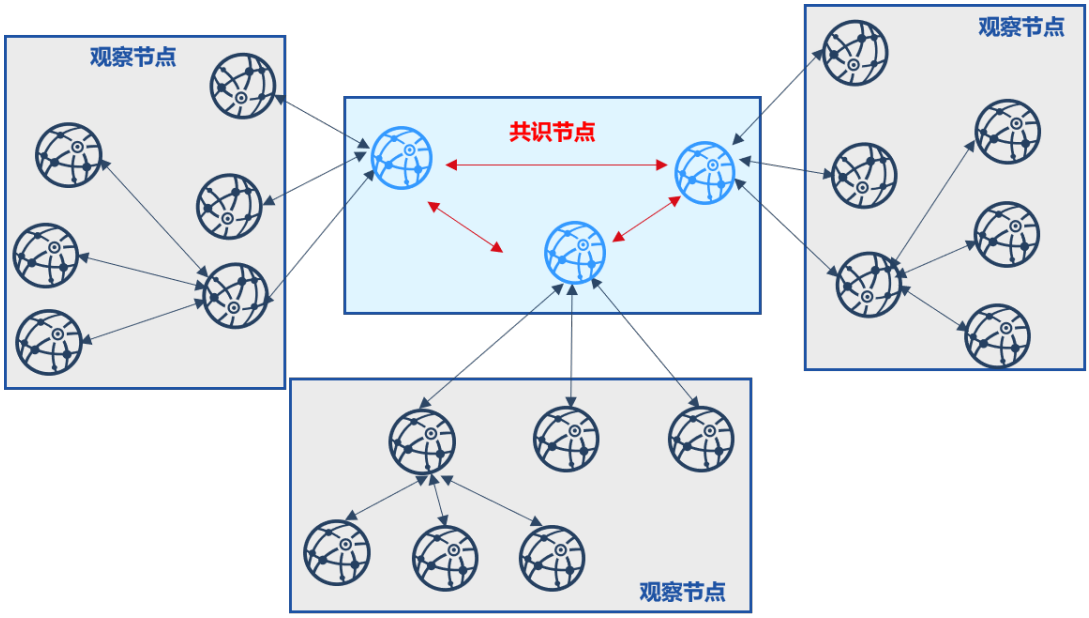
考虑到目前使用BFT类共识算法的区块链网络复杂度较高、不具有无限可扩展性，因此大部分业务架构仅有部分节点作为共识节点，其他节点均作为观察节点(不参与共识，但拥有区块链全量数据)，如下图所示。



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：



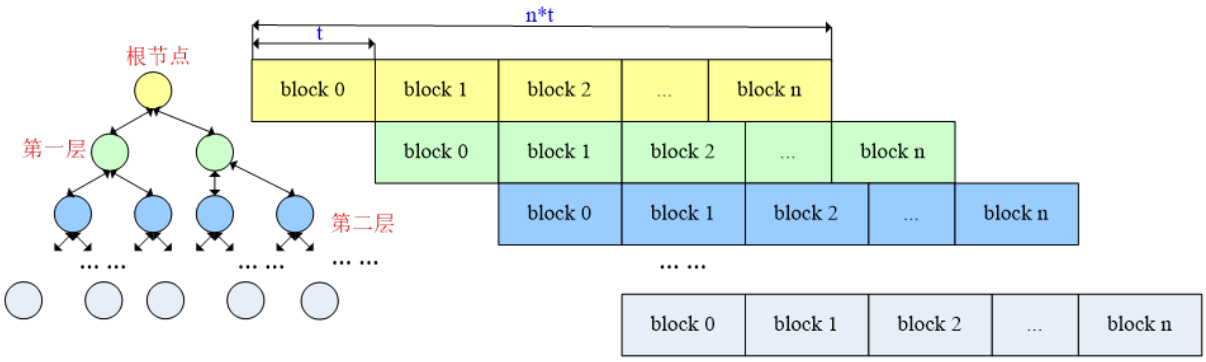
该策略将观察节点分摊给每个共识节点，并以共识节点为顶点构造一颗三叉树，共识节点出块后，优先向其子观察节点发送最新区块状态，子观察节点同步最新区块后，优先向自己的子节点发送最新区块状态，以此类推。采用了区块状态树状广播策略后，每个节点仅将最新区块状态发送给子节点，设区块大小为`block_size`，树的宽度为`w`，则用于区块同步的网络带宽均为`(block_size * w)`，与区块链系统的节点总数无关，具有可扩展性。上图所示的共识节点采用区块状态树状广播后，出带宽降低了2倍。

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。

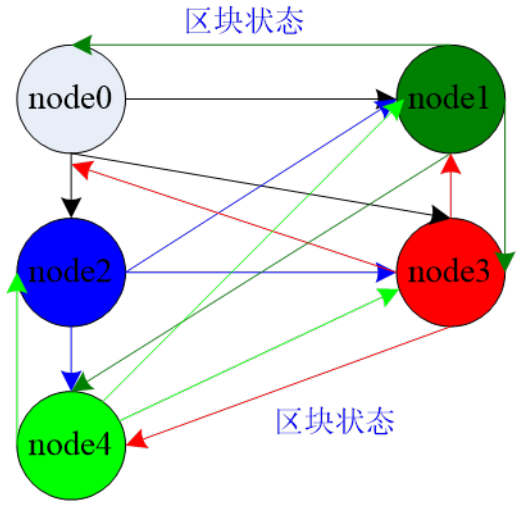
下图展示了各层节点提交`n`个区块的时延，设树的深度为`d`，每个区块提交时延为`t`，则根节点(共识节点)提交`n`个区块的时延为`n*t`，第一层节点(观察者节点)同步并提交区块的时延为`n*t + t`，第二层节点(观察者节点)同步并提交区块的时延为`n*t + 2*t`，叶子节点同步并提交区块的时延为`n*t + d * t`，与共识节点的时延差为`d*t`，`n`远大于`d`时，这个时延几乎可以忽略，因此该策略对观察者节点TPS的影响非常小。



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。



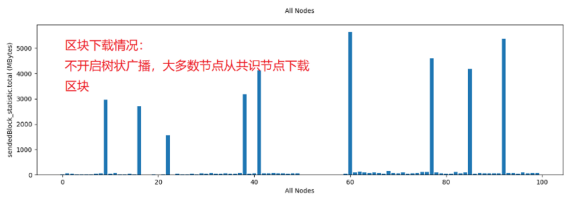
上图展示了各个节点如何使用gossip协议定期同步区块状态：

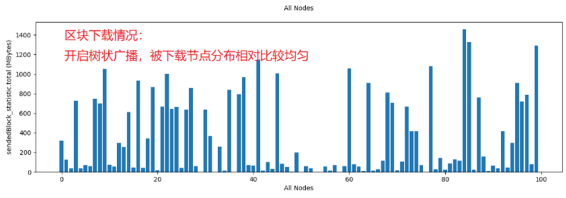
- 各个区块链节点每2s随机选择三个邻居节点广播当前区块状态，包括{区块高度，区块哈希}
- 节点收到这些区块状态包后，更新本地缓存的各个节点区块状态到最新
- 若某节点区块高度高于本节点区块高度，该节点会向拥有更高区块的节点同步区块

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：



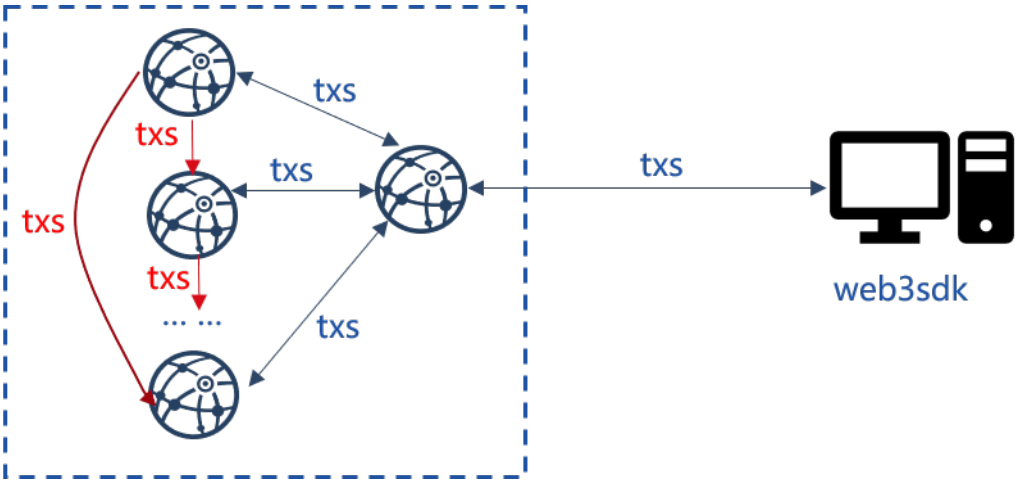


整个区块链网络中包含10个共识节点，90个观察者节点，树的度设置为2。优化前，观察者节点主要从10个共识节点下载区块，共识节点的出流量可达到5000MB；优化后，部分下载流量分摊到了观察者节点，节点由区块下载带来的流量开销降低到了1400MB，降低了3倍多，基本接近最优(最优的情况是优化前峰值出带宽是优化后峰值出带宽的4.5倍，由于gossip协议导致的区块随机拉取情况的存在，无法达到最优，只能接近最优)。

20.4.3 交易同步优化

标签：数据同步 同步优化 交易同步

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：



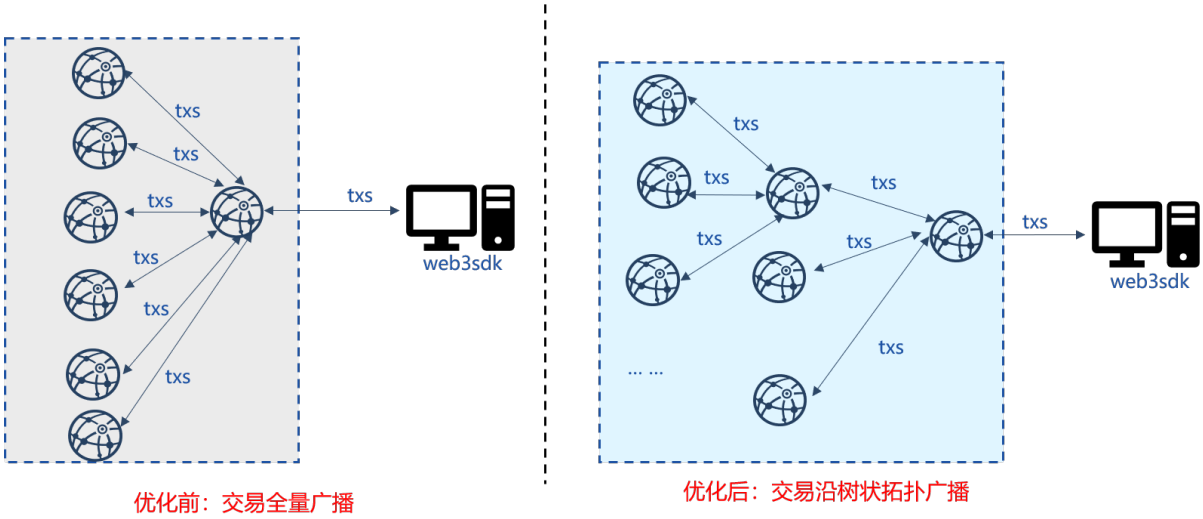
显然，SDK直连节点的出带宽与区块链节点总数成正比，随着区块链系统节点数的增加，该节点必然成为整个系统的瓶颈。

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：

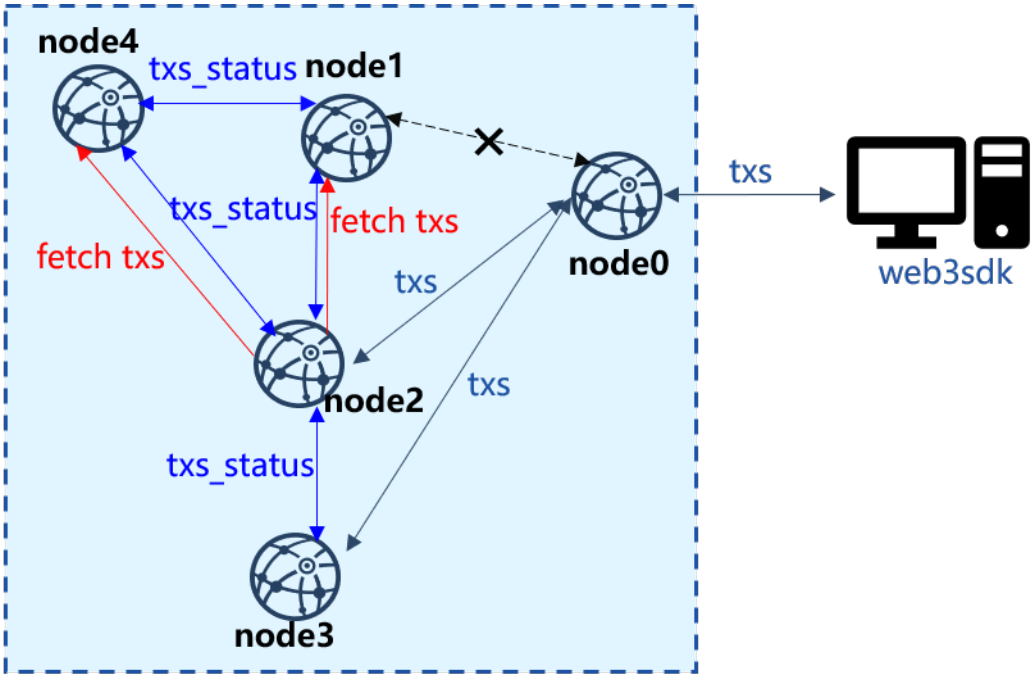


- 优化前：节点收到SDK的交易后，全量广播给其他节点；
- 优化后：节点收到SDK的交易后，将其发送给子节点，子节点收到交易后，继续将其发送给自身的子节点。

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。

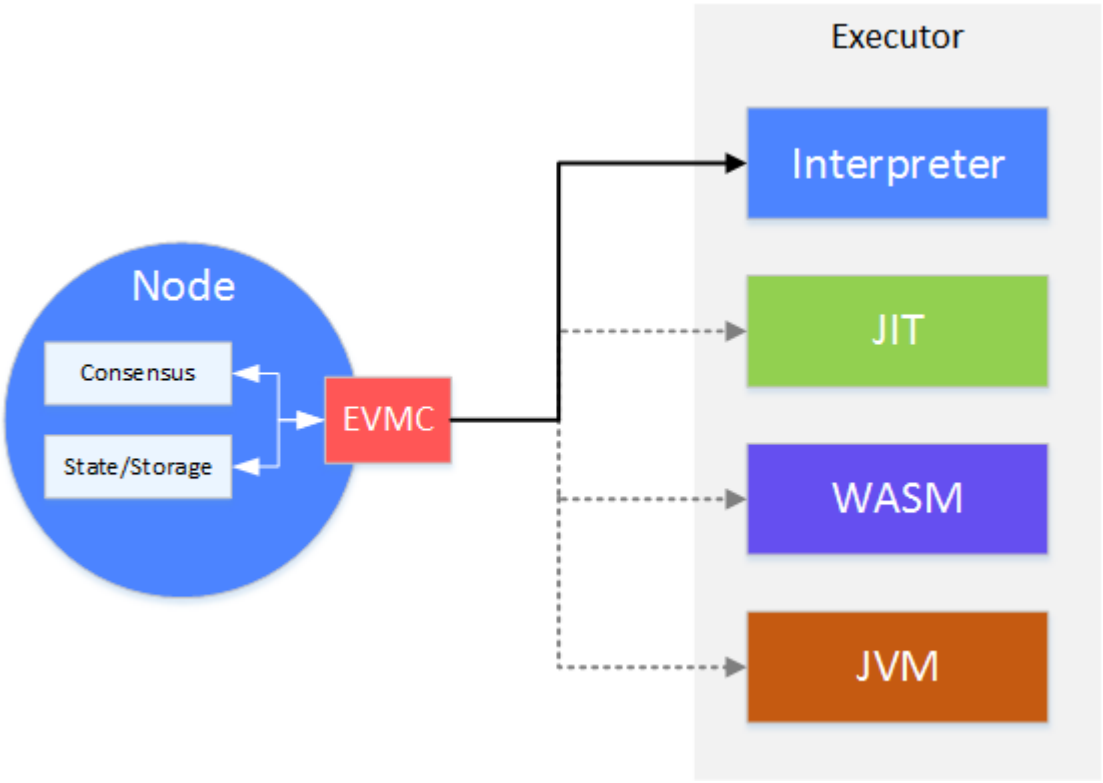
20.5 智能合约

标签： 智能合约 虚拟机

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。

20.5.1 EVM 以太坊虚拟机

标签：EVM 智能合约 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。

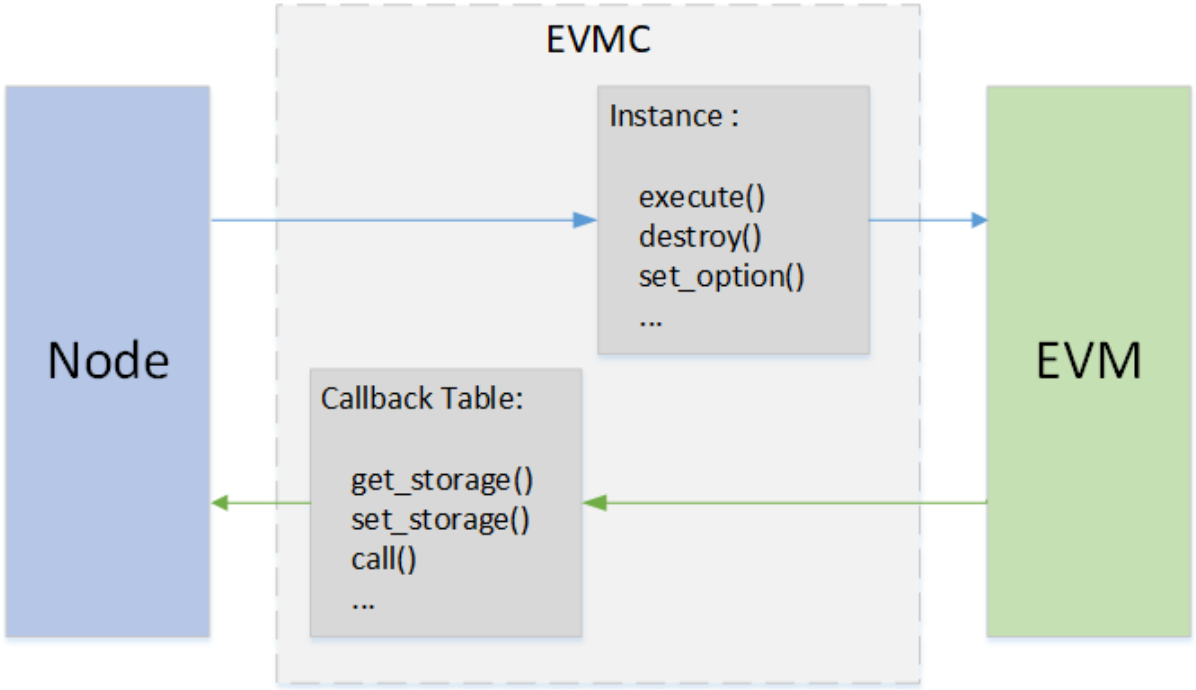
传统的solidity虚拟机，在以太坊中称为interpreter，下文主要解释interpreter的实现。

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.
```

(continues on next page)

(续上页)

```
    m_returnData.clear();

    bytesRef output;
    if (caseCallSetup(msg, output))
    {
        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。

20.5.2 预编译合约

标签：预编译合约 智能合约 `Precompiled`

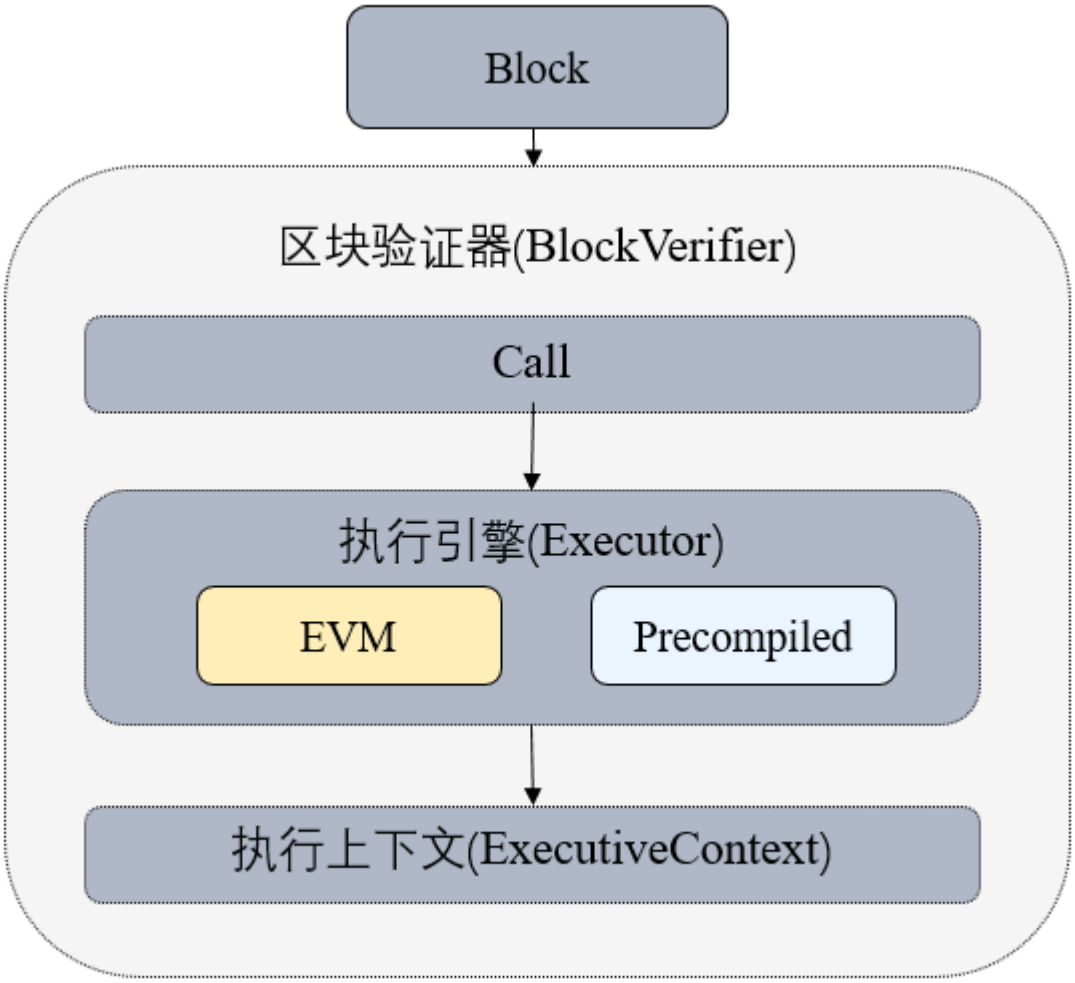
预编译合约提供一种使用C++编写合约的方法，合约逻辑与数据分离，相比于solidity合约具有更好的性能，可以通过修改底层代码实现合约升级。

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`。

20.5.3 Gas

标签：`Gas` 智能合约 虚拟机

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。

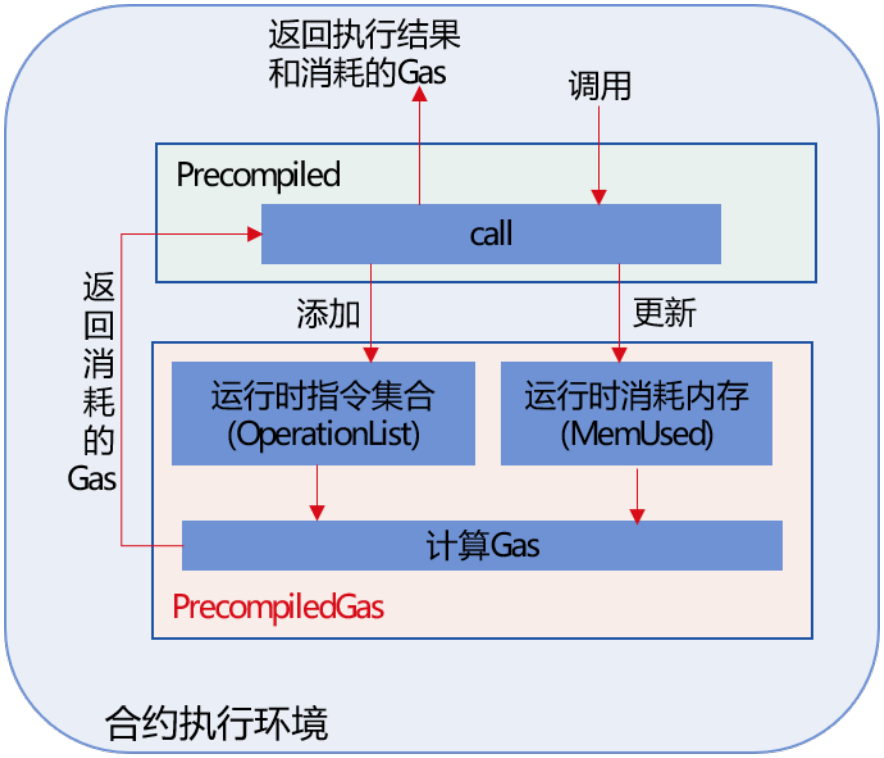
此外，EVM原始的Gas机制中，交易的主要Gas消耗来源于存储，考虑到联盟链场景更关注CPU和内存消耗，FISCO BCOS v2.4.0调整了存储Gas，引入`Free Storage` Gas衡量模式，提升CPU和内存在交易Gas消耗中的占比。

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：

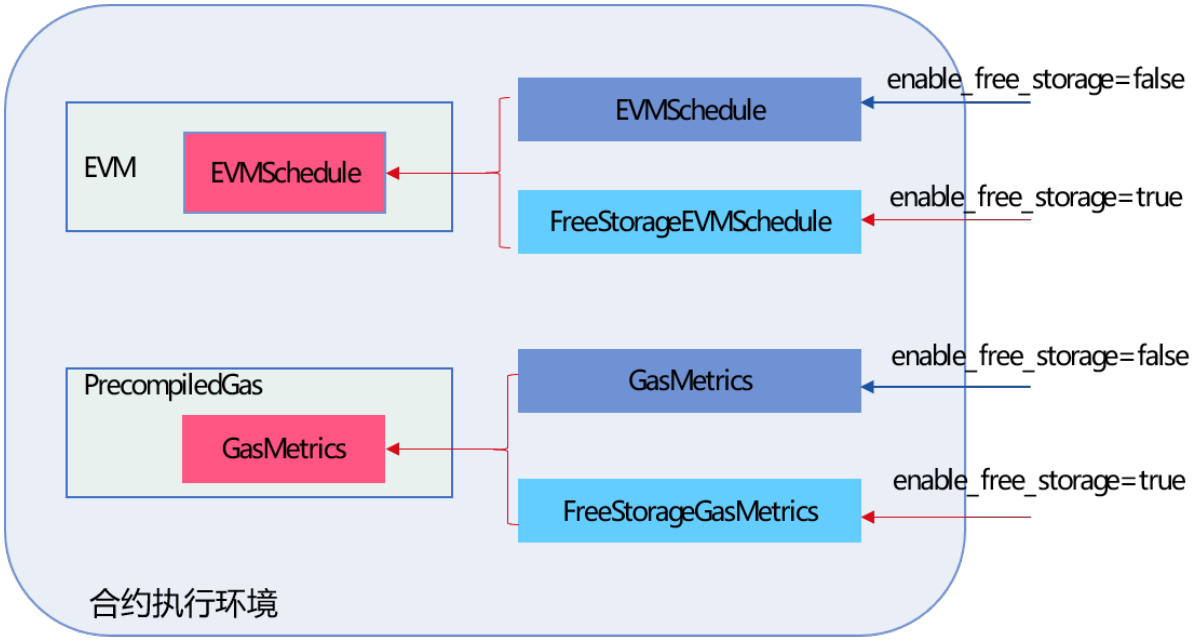
EVM Gas衡量标准插件化

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

20.6 存储模块

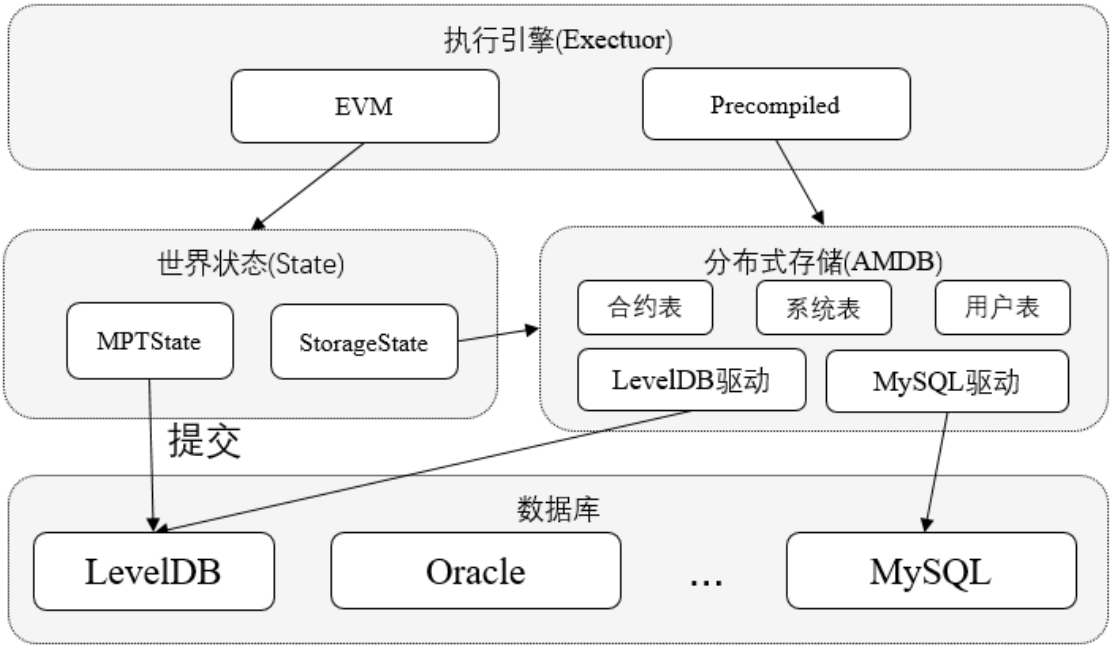
标签：存储模块 `MPT` 分布式存储 区块数据 状态数据

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。



20.6.1 AMDB

标签：存储模块 分布式存储 合约数据 状态数据

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

20.6.2 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。

20.6.3 MPT State

标签：存储模块 `MPT` 状态数据

MPT State是以太坊上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

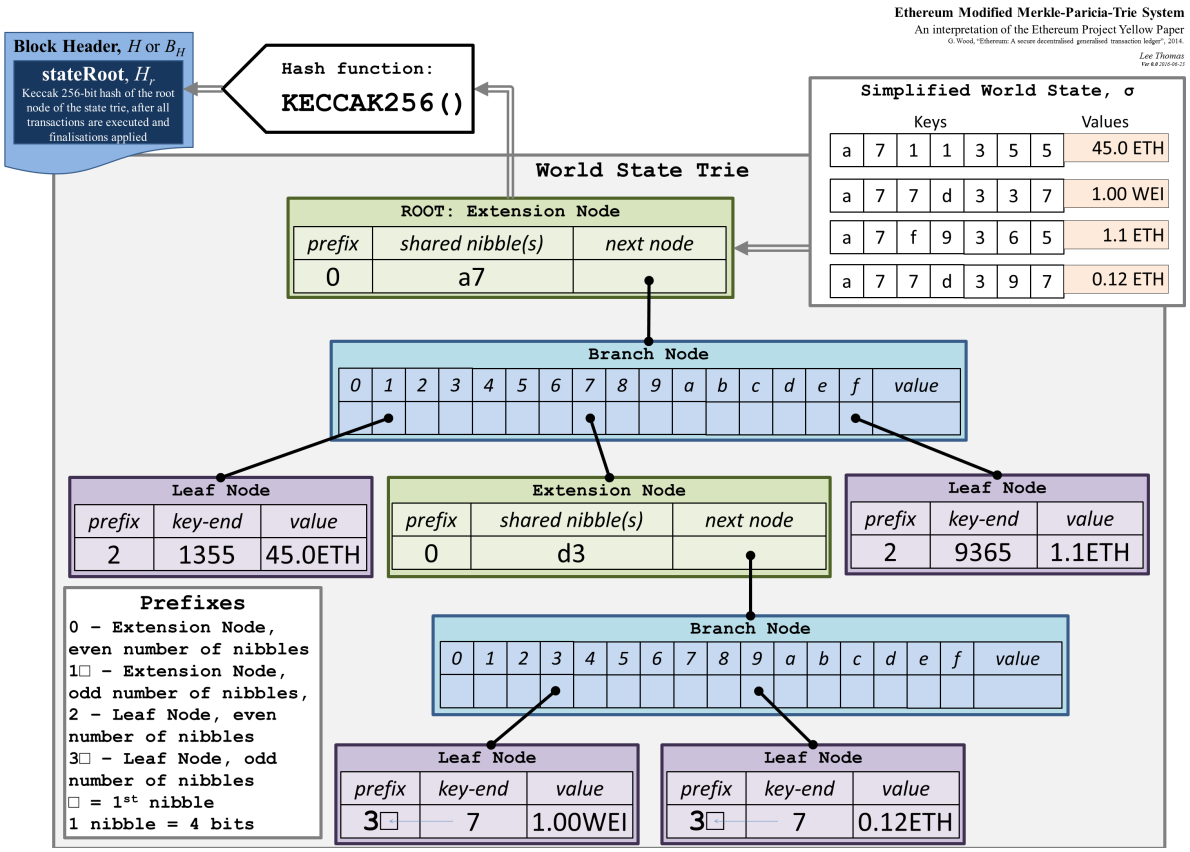
MPT(Merkle Patricia Trie)，是一种用hash索引数据的前缀树。

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）



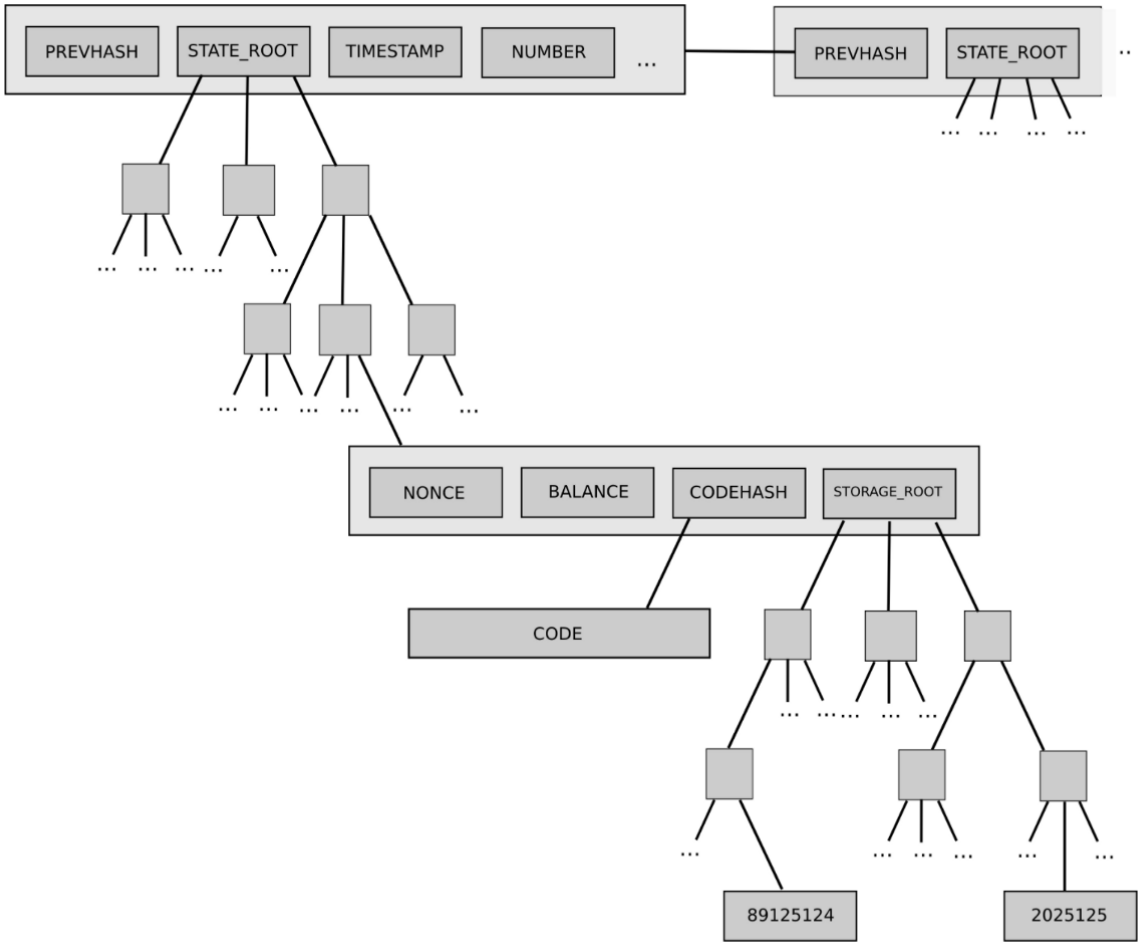
状态 State

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。

20.7 安全控制

标签：安全控制 网络安全 存储安全 黑白名单 权限控制

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。

20.7.1 节点准入管理介绍

标签：安全控制 访问控制 节点准入

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》。

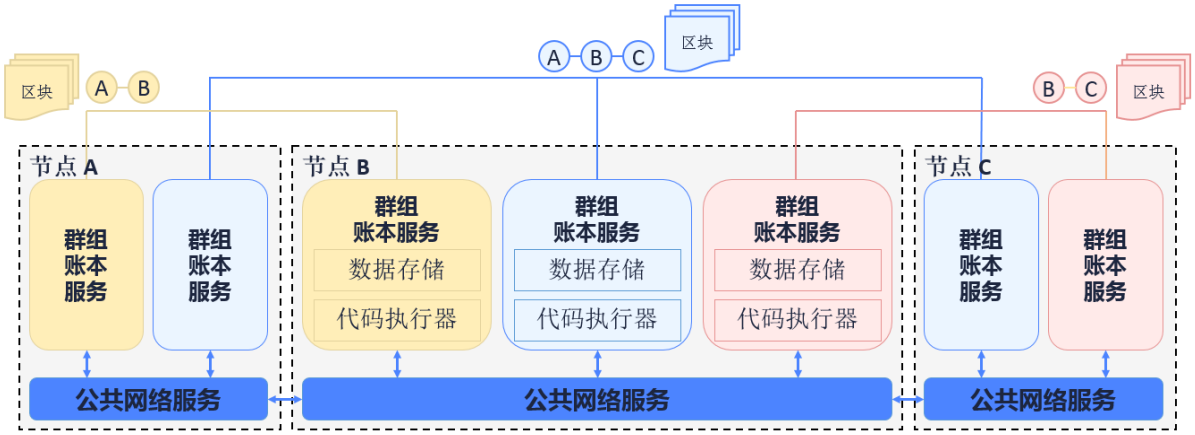
概述

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。

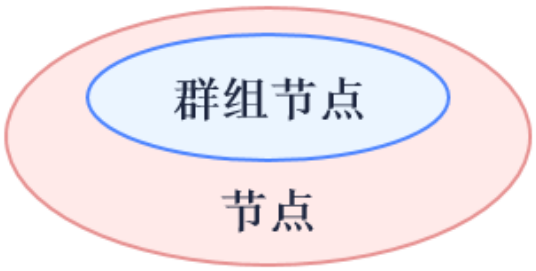
名词解释

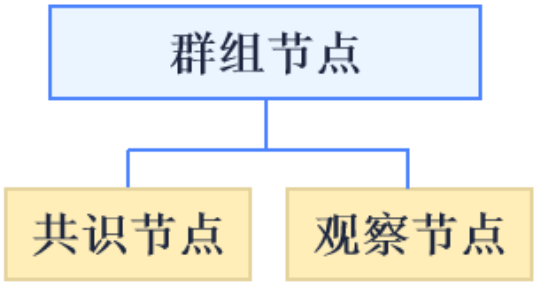
节点类型

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：



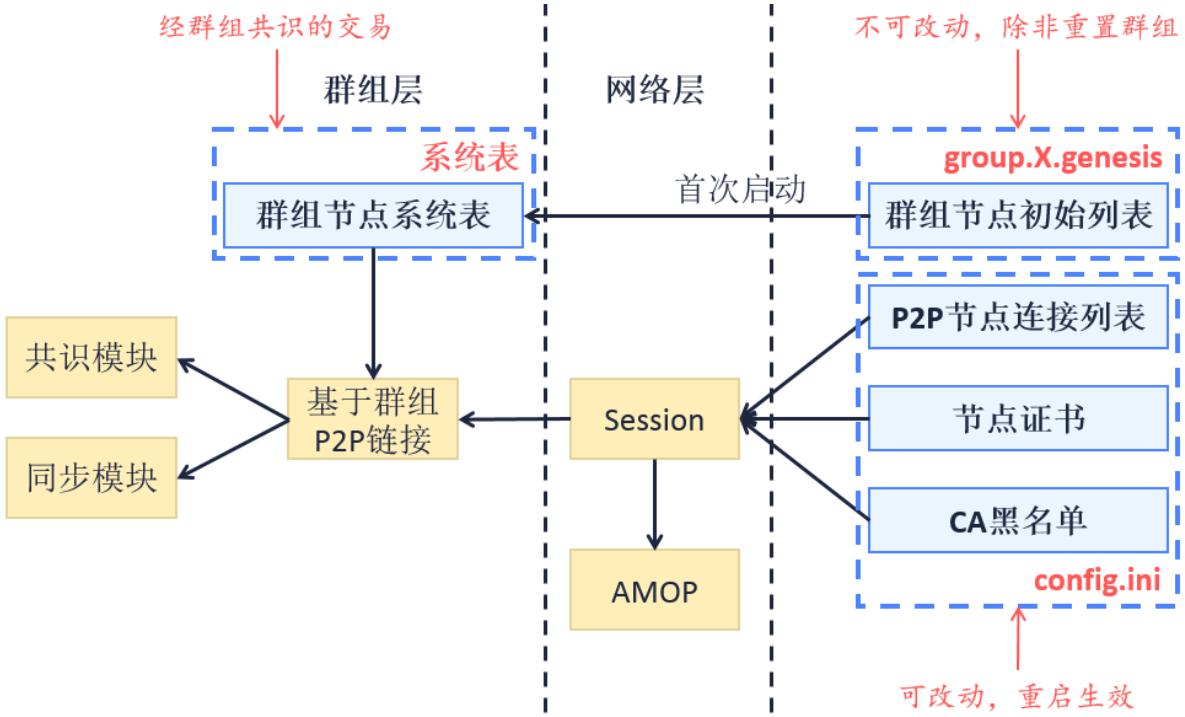


配置类型

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**。

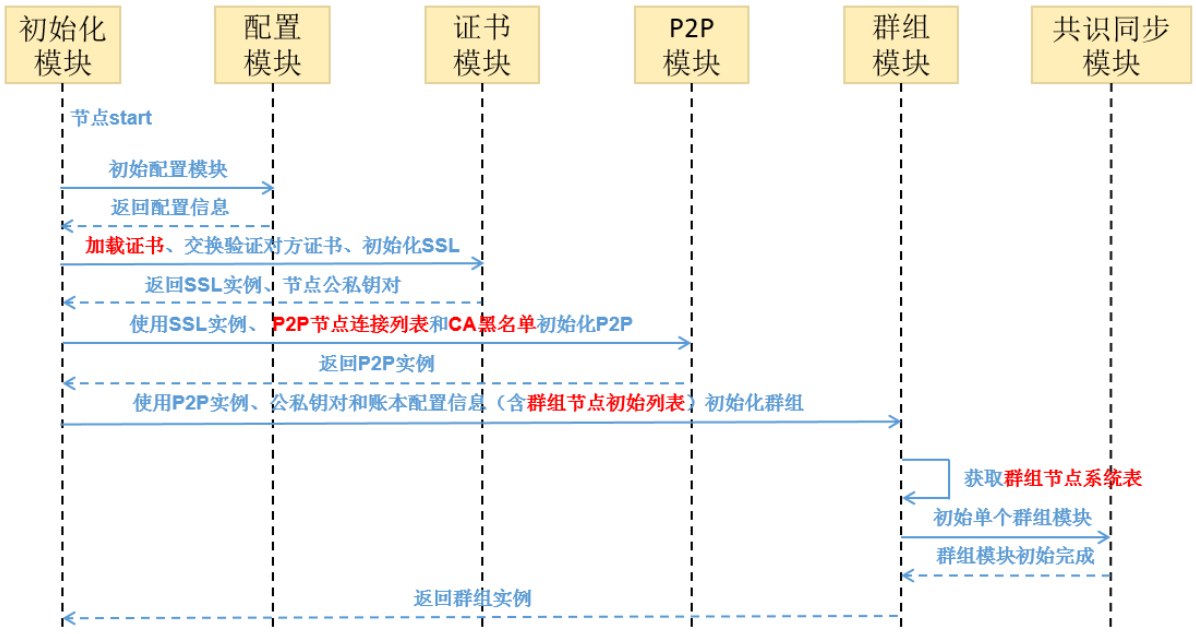
模块架构



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。

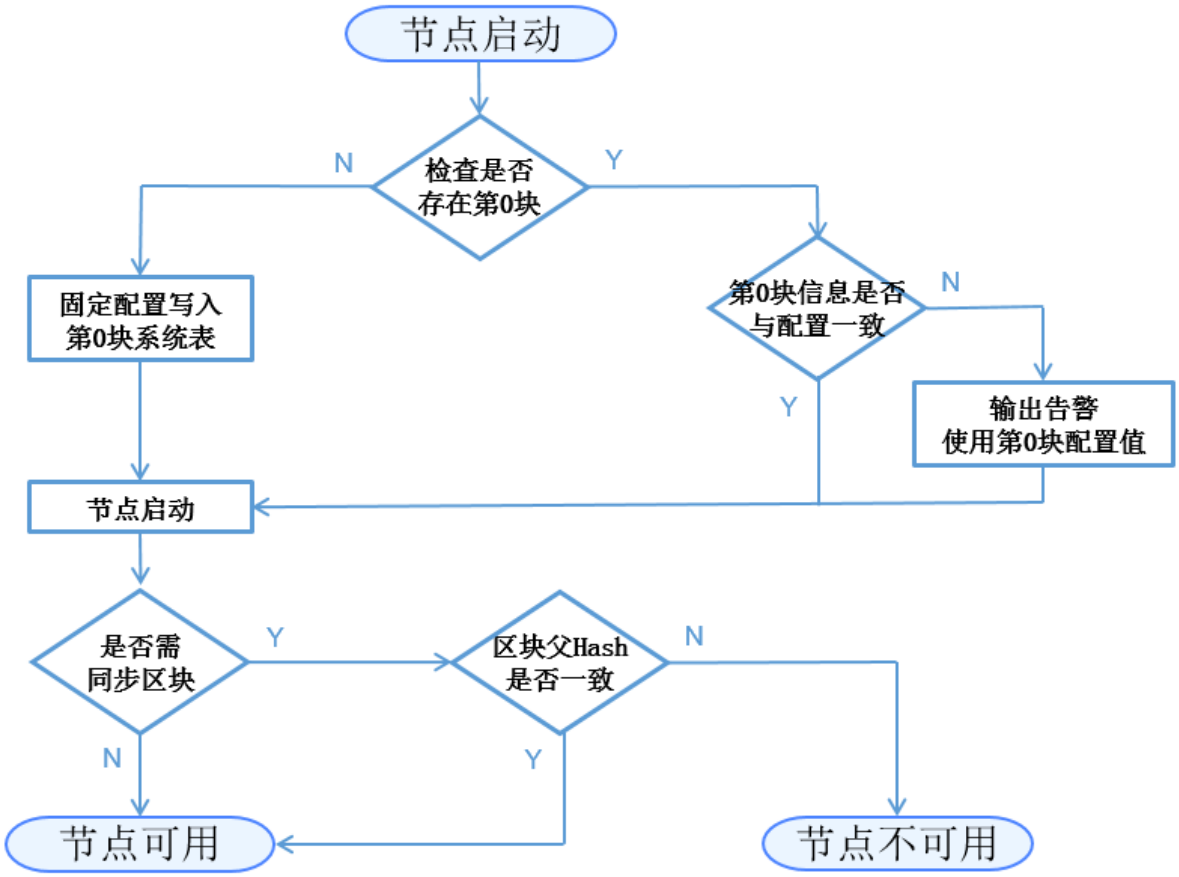
核心流程

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

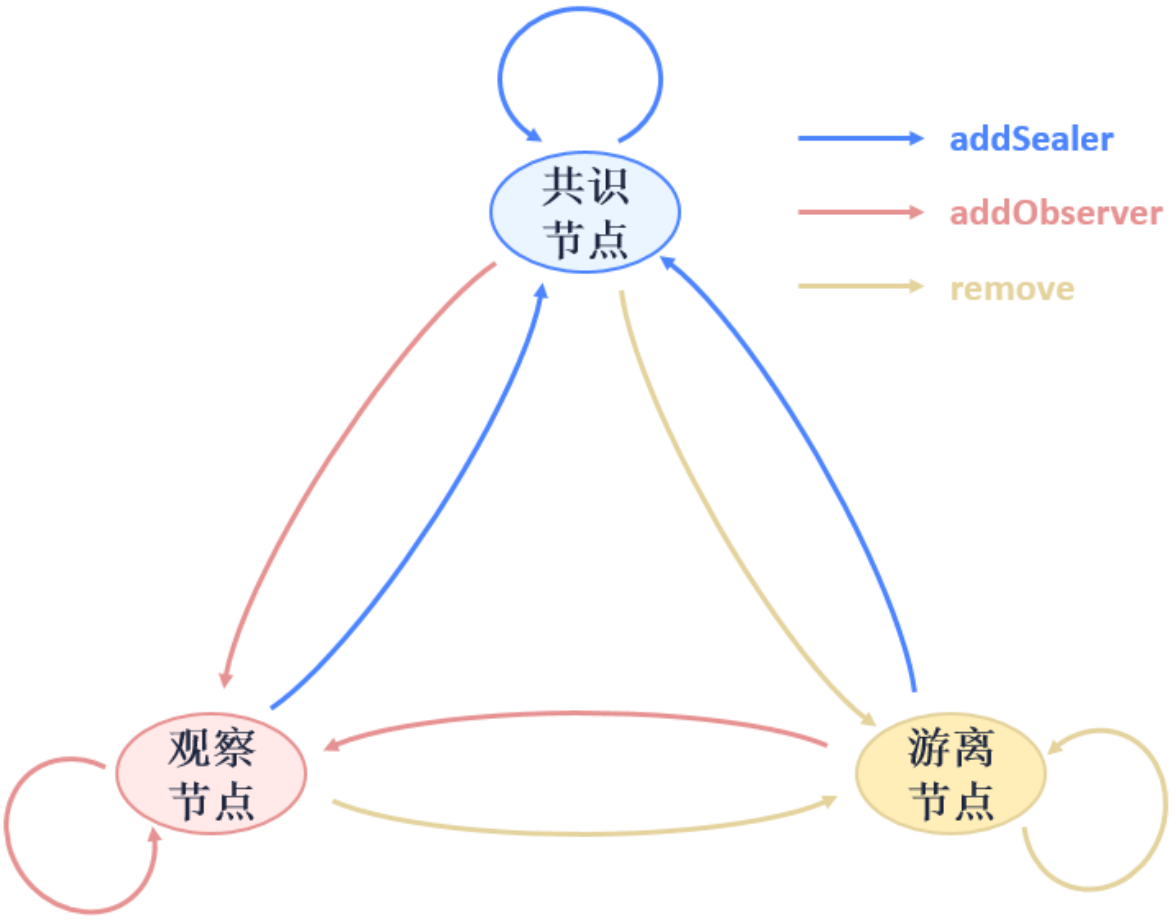
SSL认证用于确定节点之间是否许可加入某条链。一条链上的节点均信任可信的第三方（节点证书的颁发者）。

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。

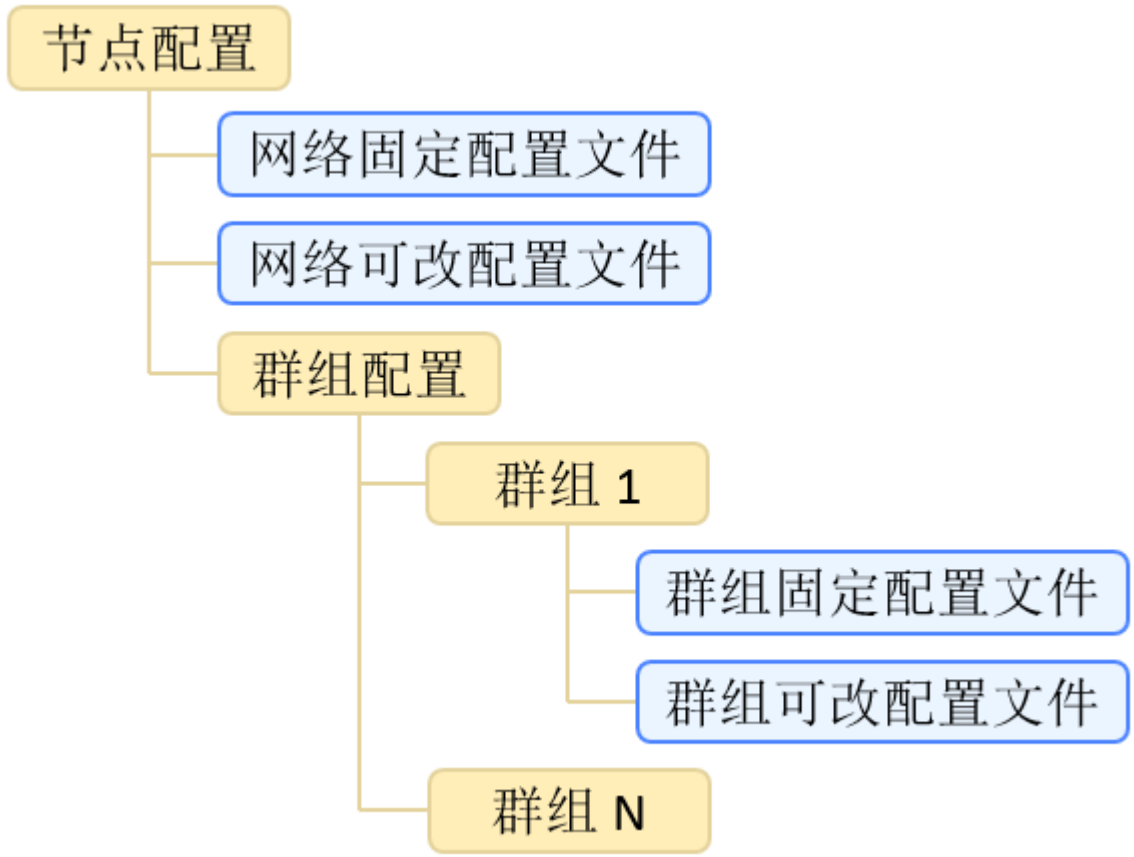
节点相关类型及其转换操作

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。

20.7.2 CA黑白名单介绍

标签：安全控制 网络安全 黑白名单

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

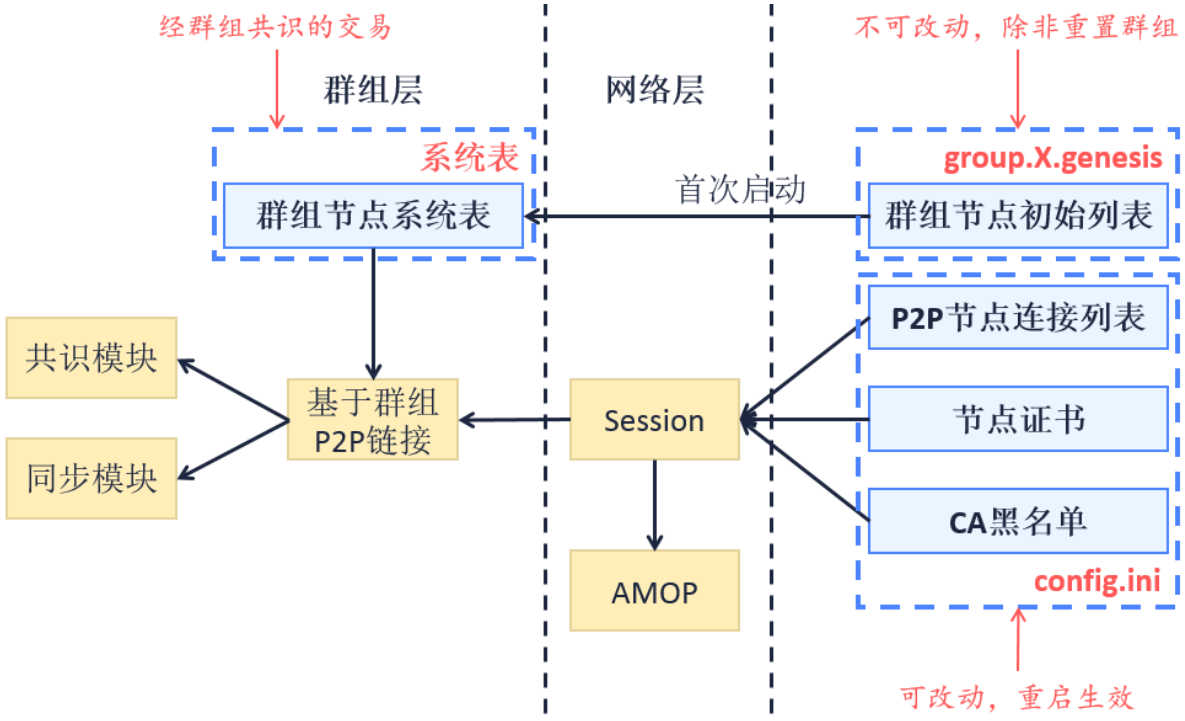
- 别称**证书接受列表**（certificate whitelist，简称CAL）。CA白名单基于`config.ini`文件中`[certificate_whitelist]`配置的NodeID进行判断，拒绝除白名单外所有节点发起的连接。

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
```

20.7.3 基于角色的权限控制

标签：安全控制 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

20.7.4 权限控制

标签：安全控制 访问控制 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。

20.8 远程过程调用（RPC）

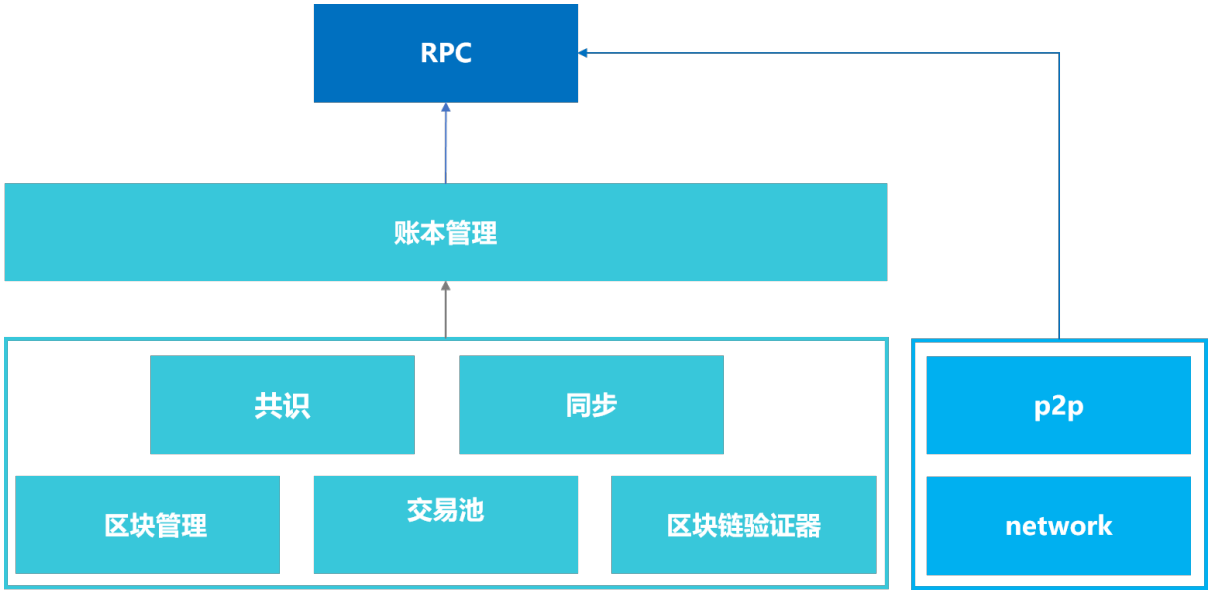
标签：远程过程调用 RPC

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。

20.8.1 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。

20.8.2 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。

20.8.3 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

20.8.4 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。

20.8.5 5 RPC接口列表

参考RPC API文档

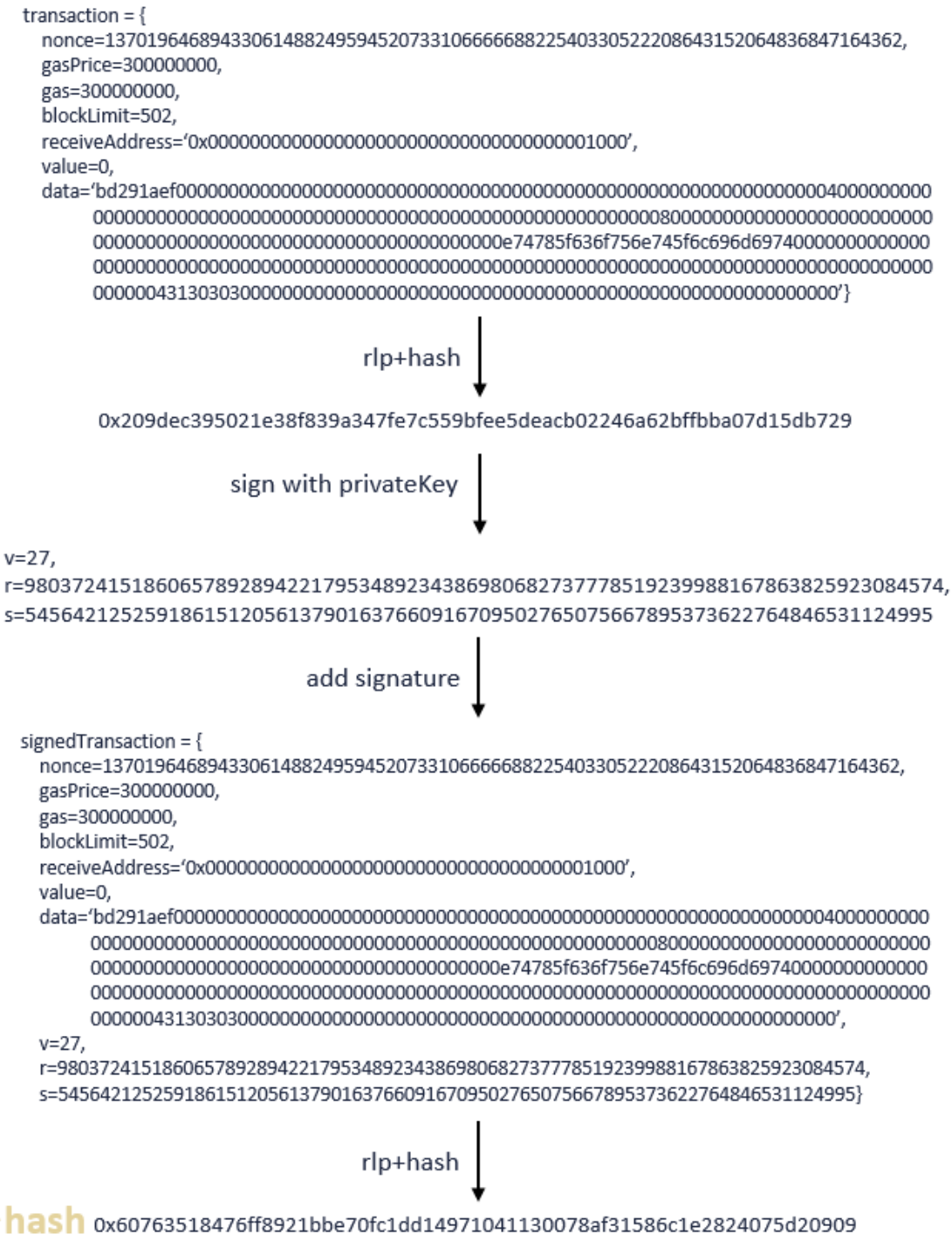
20.9 数据结构与编码协议

标签：数据结构 RLP 区块结构 消息包 交易结构

20.9.1 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：

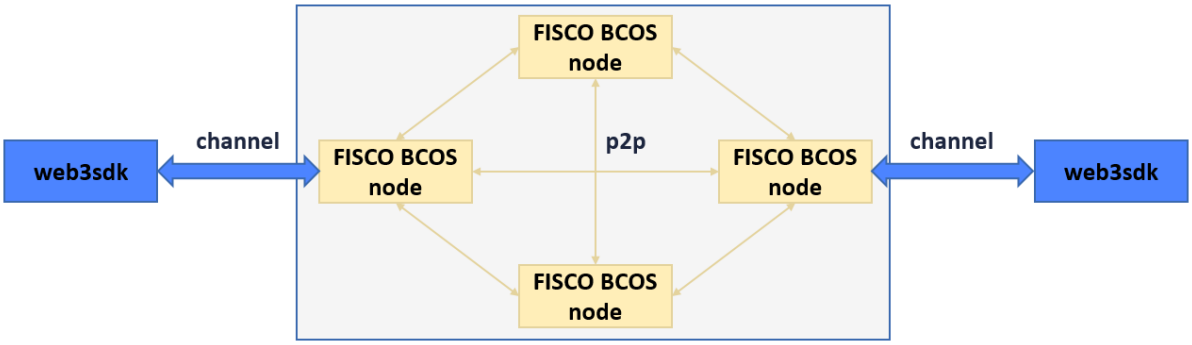
RC1的hashWith字段（也称交易hash/交易唯一标识）的生成流程如下：


transaction = {
nonce=137019646894330614882495945207331066666882254033052220864315206483684716436 2,
gasPrice=300000000,
gas=300000000,
blockLimit=502,
receiveAddress='0x0000000000000000000000000000000000001000',
value=0,
data='bd291aef0000000000000000000000000000000000000000000000000000000000000040000000000000000000000000000000000000000000000000000000000000008000000000000000000000000000000000000000000000000000000000000000e74785f636f756e745f6c696d6974000000000000000000000000000000000000000000000000000000000

20.9.3 交易收据

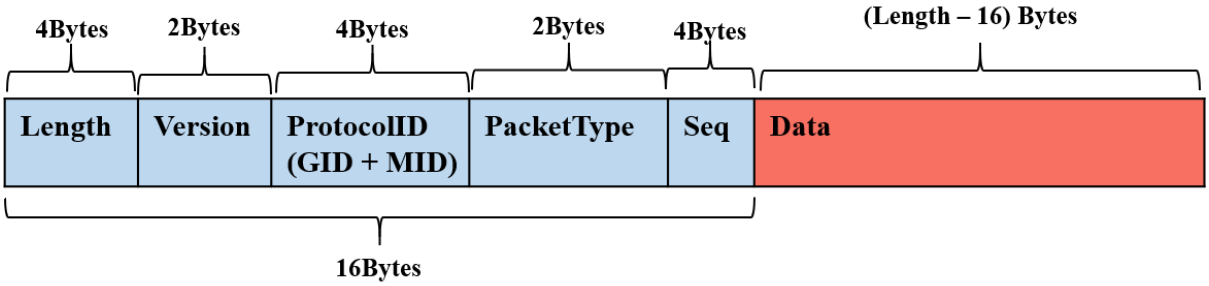
20.9.4 网络传输协议

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包结构，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

20.10 其他特性

标签：设计方案 特性 区块结构 消息包 交易结构

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。

20.10.1 CNS方案

标签：合约命名服务 CNS表结构

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。

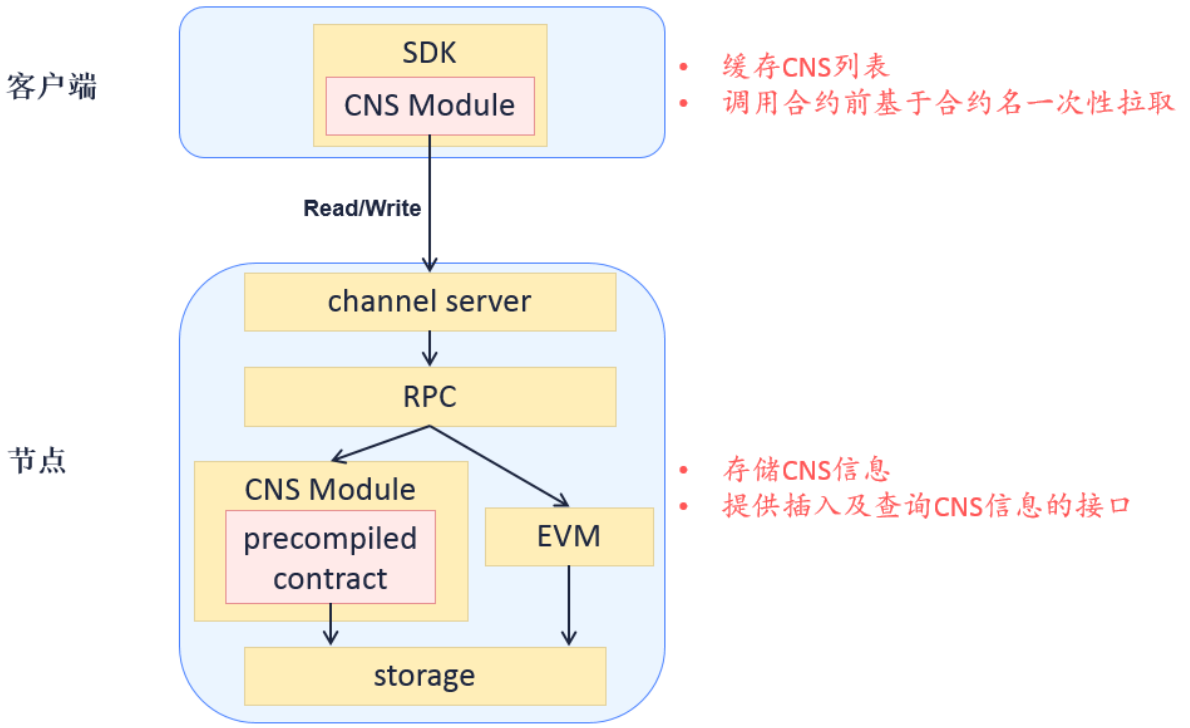
对标ENS

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：

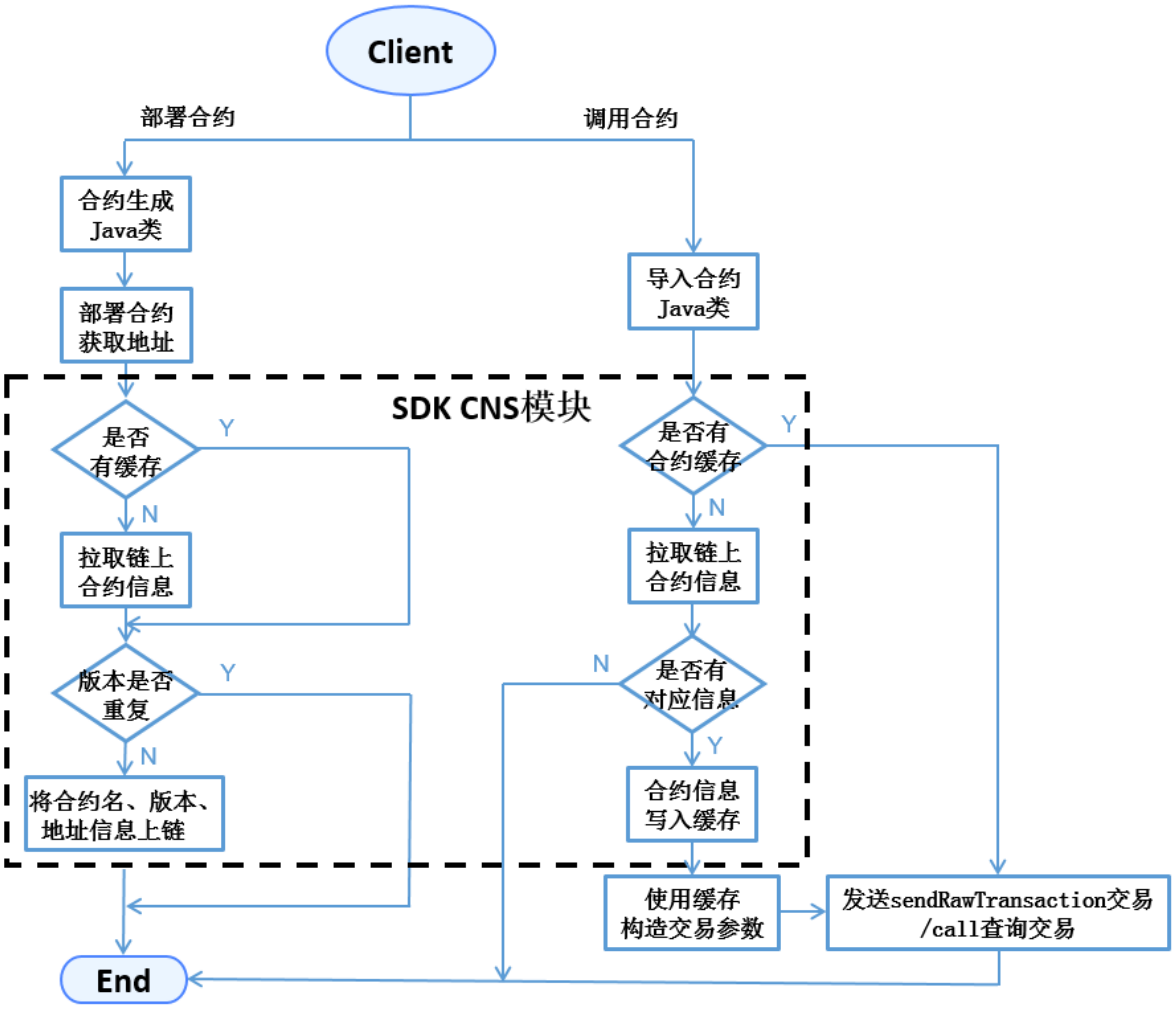
- ENS映射的地址类型包括合约地址及钱包地址，CNS可支持，当地址类型为钱包地址时合约abi为空。
- ENS有竞拍功能，CNS不需支持。
- ENS支持多级域名，CNS不需支持。

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，2.6及其以上版本控制台使用手册请参考这里, 1.x版本控制台使用手册请参考这里。

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

20.10.2 国密支持方案

标签：国密算法 `SSL` `SM`

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：

20.10.3 落盘加密

标签：落盘加密 数据加密 数据安全

背景介绍

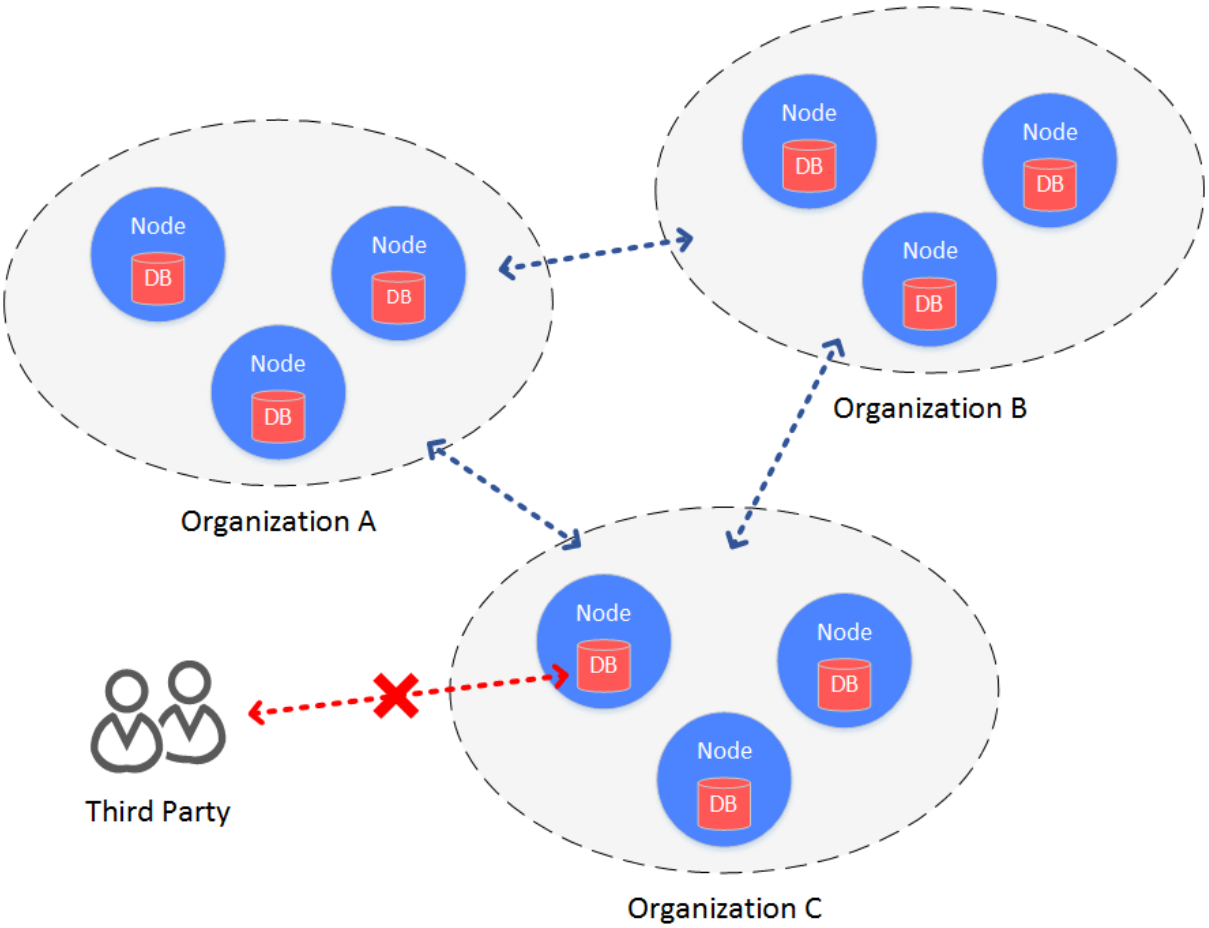
在联盟链的架构中，机构和机构之间搭建一条区块链，数据在联盟链的各个机构内是可见的。

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

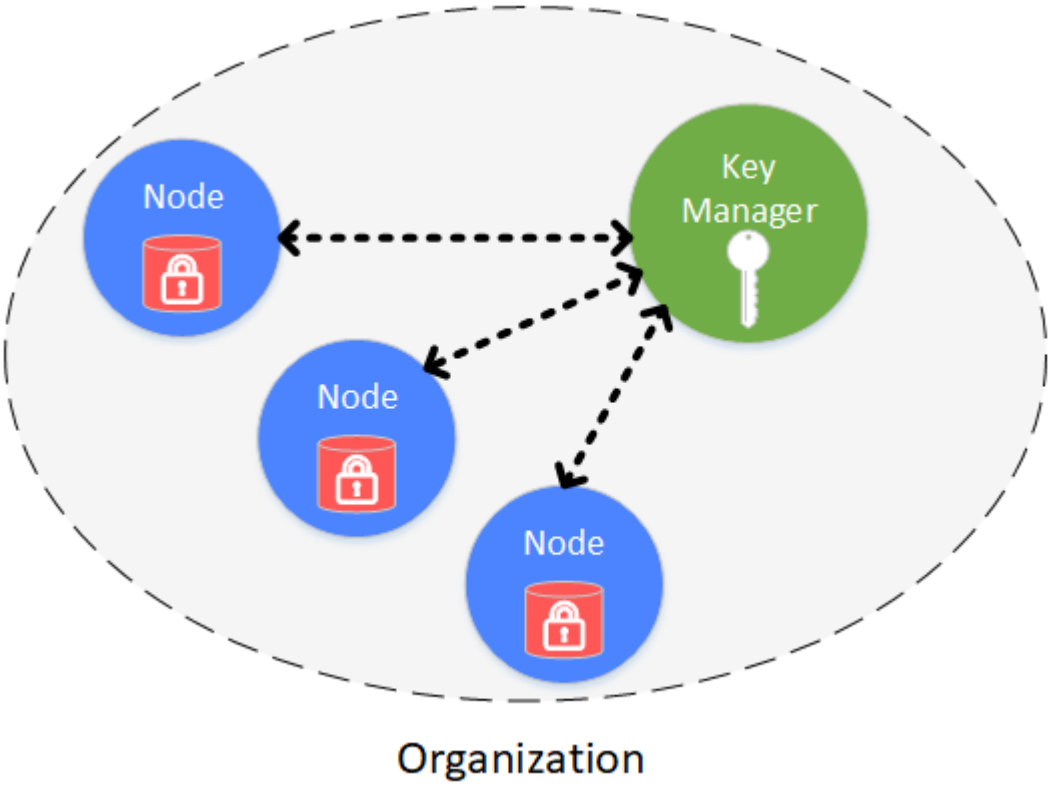
对于链上通信数据的访问控制，FISCO BCOS是通过节点证书和SSL来完成。此处主要介绍的是节点存储数据的访问控制，即落盘加密。



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。

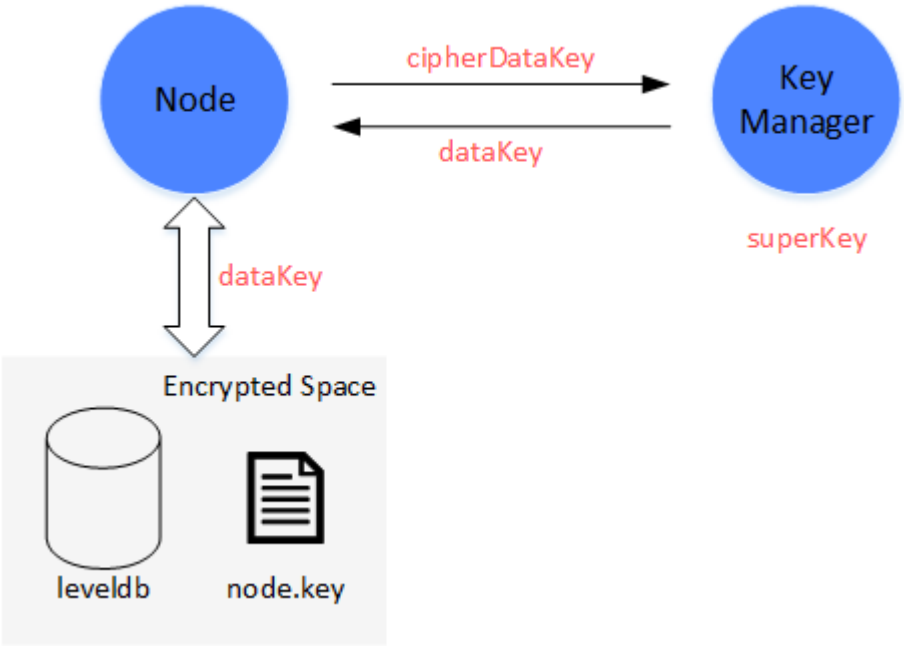
节点

- 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

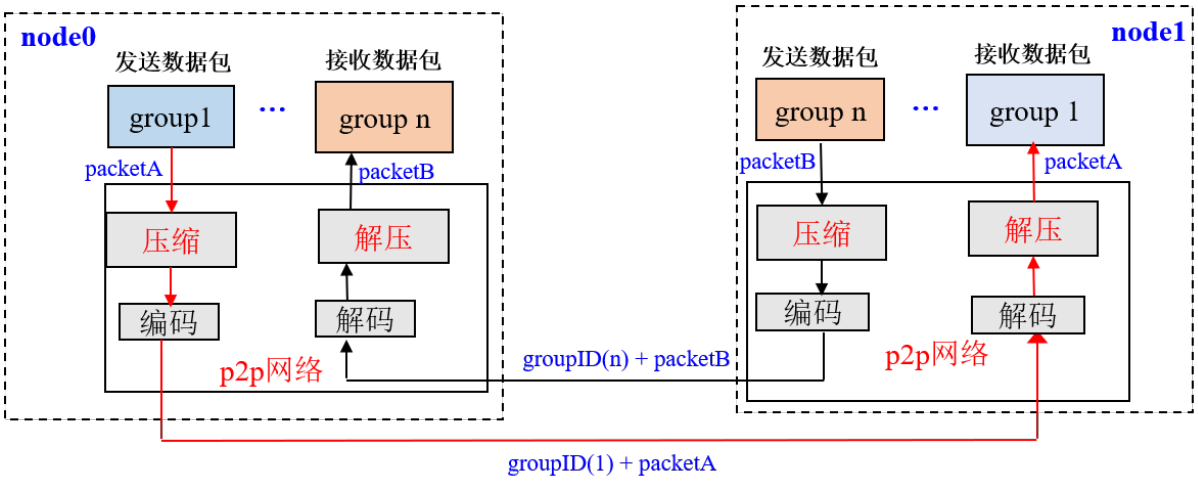
20.10.4 网络压缩

标签：网络压缩 数据压缩

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：

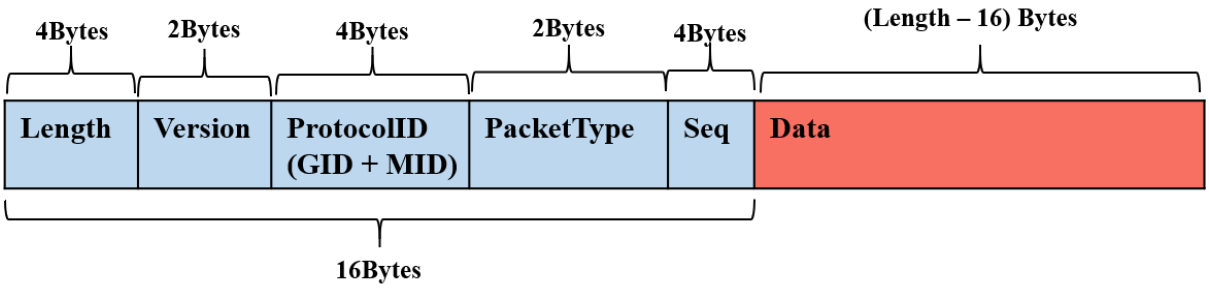
- **发送端压缩数据包**：群组层通过P2P层发送数据时，若数据包大小超过1KB，则压缩数据包后，将其发送到目标节点；
- **接收端解压数据包**：节点收到数据包后，首判断收到的数据包是否被压缩，若数据包是压缩后的数据包，则将其解压后传递给指定群组，否则直接将数据传递给对应群组。

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：

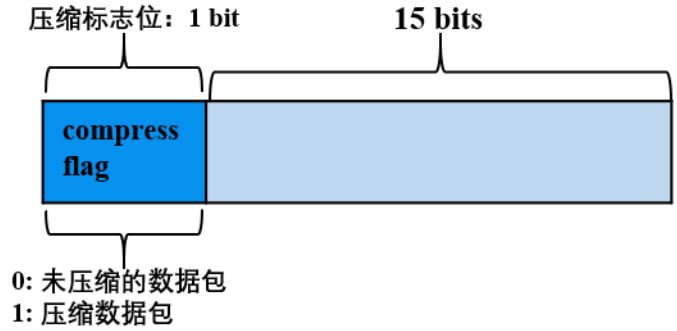


网络数据包主要包括包头和数据两部分，包头占了16个字节，各个字段含义如下：

- 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。

20.10.5 合约管理

标签：合约生命周期 智能合约

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。

20.10.6 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。

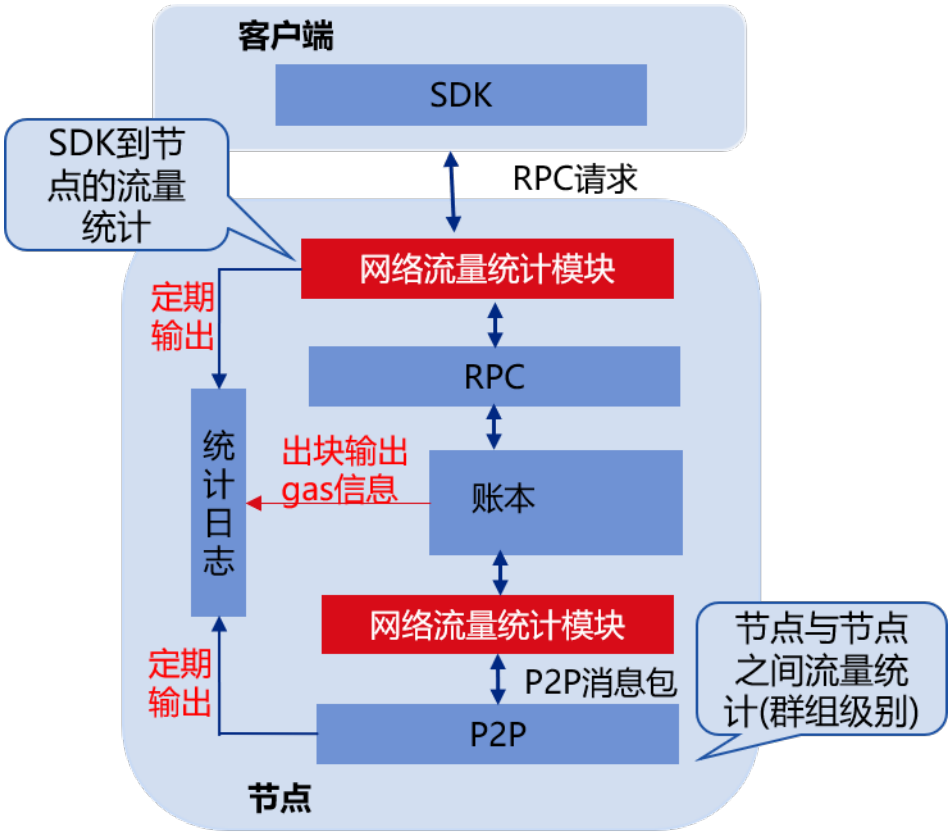
20.10.7 网络流量和Gas统计

标签：网络流量 `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

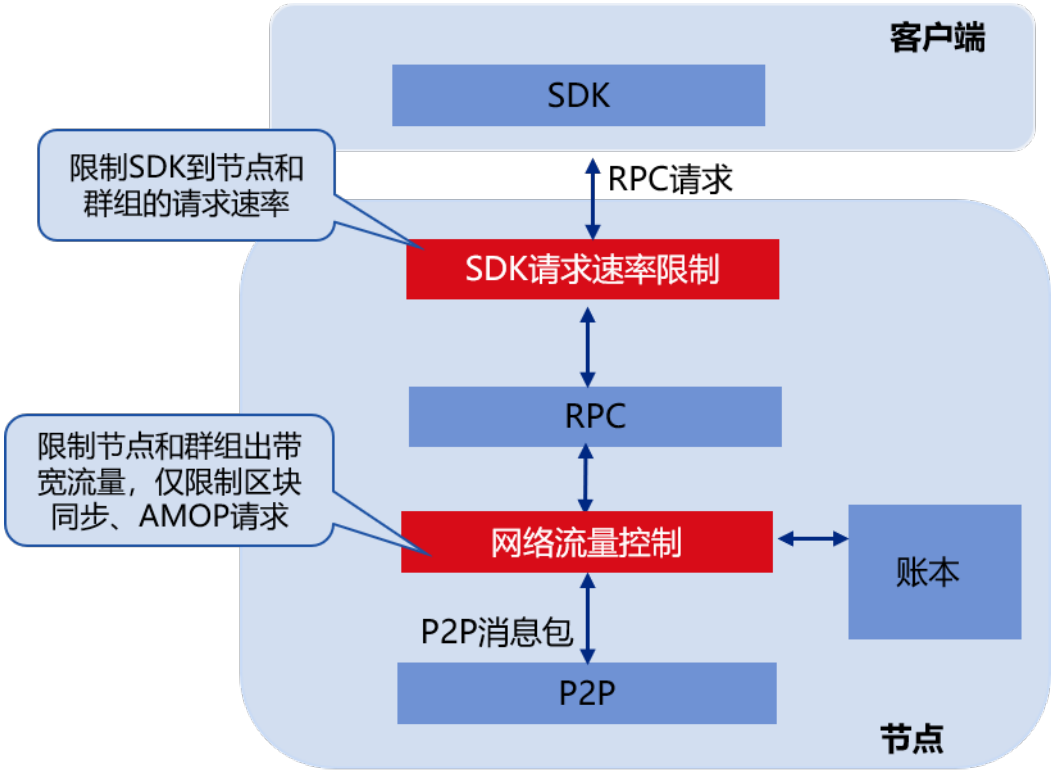
20.10.8 流量控制

标签：流量控制 速率限制

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 21

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。

21.1 getClientVersion

返回节点的版本信息

21.1.1 参数

无

21.1.2 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"
  }
}
```

21.2 getBlockNumber

返回节点指定群组内的最新区块高度

21.2.1 参数

- groupID: unsigned int - 群组ID

21.2.2 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"
}
```

21.3 getPbftView

返回节点所在指定群组内的最新PBFT视图

21.3.1 参数

- groupID: unsigned int - 群组ID

21.3.2 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"
}
```

21.4 getSealerList

返回指定群组内的共识节点列表

21.4.1 参数

- `groupID: unsigned int` - 群组ID

21.4.2 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
↪"
    ]
}
```

21.5 getObserverList

返回指定群组内的观察节点列表

21.5.1 参数

- groupID: unsigned int - 群组ID

21.5.2 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
↪"
    ]
}
```

21.6 getConsensusStatus

返回指定群组内的共识状态信息

21.6.1 参数

- groupID: unsigned int - 群组ID

21.6.2 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
    }
  ]
}
```

21.7 getSyncStatus

返回指定群组内的同步状态信息

21.7.1 参数

- groupID: unsigned int - 群组ID

21.7.2 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"
  }
}
```

21.8 getPeers

返回已连接的p2p节点信息

21.8.1 参数

- `groupID`: `unsigned int` - 群组ID

21.8.2 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": [ ]
        }
    ]
}
```

21.9 getGroupPeers

返回指定群组内的共识节点和观察节点列表

21.9.1 参数

- `groupID: unsigned int` - 群组ID

21.9.2 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
↪"
    ]
}
```

21.10 getNodeIDList

返回节点本身和已连接的p2p节点列表

21.10.1 参数

- `groupID: unsigned int` - 群组ID

21.10.2 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
↪"
    ]
}
```

21.11 getGroupList

返回节点所属群组的群组ID列表

21.11.1 参数

无

21.11.2 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]
}
```

21.12 getBlockByHash

返回根据区块哈希查询的区块信息

21.12.1 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)

21.12.2 返回值

- `object` - 区块信息，字段如下：
  - `dbHash`: `string` - 记录交易数据变更的哈希
  - `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
  - `receiptsRoot`: `string` - 区块内所有交易回执的merkle根
  - `sealer`: `string` - 共识节点序号
  - `sealerList`: `array` - 共识节点列表
  - `signatureList`: `string` - PBFT共识的签名列表
  - `stateRoot`: `string` - 状态根哈希
  - `timestamp`: `string` - 时间戳，单位毫秒
  - `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）

  - transactionsRoot: string - 区块内所有交易的merkle根

- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",true],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "dbHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number": "0x1",
    "parentHash":
↪"0x249f59e00beac8424a7821c4750fdd70c128f4ce795afbab53f345e9fce95d1a",
    "receiptsRoot":
↪"0x69a04fa6073e4fc0947bac7ee6990e788d1e2c5ec0fe6c2436d0892e7f3c09d2",
    "sealer": "0x0",
    "sealerList": [

↪"4ca3a91a4937355dba6a2e5fe76141479a1fc44e9caa86750092dab64e0b8382f6b8476749c2d2de414350a5449162
↪"
    ],
    "signatureList": [
      {
        "index": "0x0",
        "signature":
↪"0x4602135870d9a4846e2536d4a48e831825a5d95768dd0d4f08544a0bd4c2af41242dec1751a05c07d7572027f8d6
↪"
      }
    ],
    "stateRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp": "0x175ac38cf10",
    "transactions": [
      {
        "blockHash":
↪"0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",
        "blockLimit": "0x100",
        "blockNumber": "0x1",
        "chainId": "0x1",
        "extraData": "0x",
        "from": "0x57c7be32cbfb3bfed4fddc87efcc735b4e945fb3",
        "gas": "0x2faf080",
        "gasPrice": "0xa",
        "groupId": "0x1",
        "hash": "0x3961fac263d8e640b148ddcfafd71d2069e93a006abc937c32fb16cfa96e661d
↪",
        "input":
↪"0x4ed3885e0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000
↪",
        "nonce": "0x3eb675ec791c2d19858c91d0046821c27d815e2e9c151604912205000002968
↪",
```

(continues on next page)

(续上页)

```
        "signature": {
          "r": "0x9edf7c0cb63645442aff11323916d51ec5440de979950747c0189f338afdcefd
↪",
          "s": "0x2f3473184513c6a3516e066ea98b7cfb55a79481c9db98e658dd016c37f03dcf
↪",
          "signature":
↪"0x9edf7c0cb63645442aff11323916d51ec5440de979950747c0189f338afdcefd2f3473184513c6a3516e066ea98b
↪",
          "v": "0x0"
        },
        "to": "0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744",
        "transactionIndex": "0x0",
        "value": "0x0"
      }
    ],
    "transactionsRoot":
↪"0xb880b08df3b43a9ffc334d7a526522b33e004ef95403d61d76454b6085b9b2f1"
  }
}

// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",false],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "dbHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

21.13 getBlockByNumber

返回根据区块高度查询的区块信息

21.13.1 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)

21.13.2 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

21.14 getBlockHeaderByHash

根据区块哈希获取区块头信息

21.14.1 参数

- `groupID`: `unsigned int` - 群组ID
- `blockHash`: `string` - 区块哈希
- `includeSignatures`: `bool` - 包含签名列表标志(true显示签名列表)，如果为true，则会带上下面例子中的`signatureList`字段。

21.14.2 返回值

```
//Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockHeaderByHash","params":[1,
↪"0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",true],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "dbHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0x99576e7567d258bd6426ddaf953ec0c953778b2f09a078423103c6555aa4362d",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000
```

(续上页)


```
    "number": "0x1",
    "parentHash":
↪"0x4f6394763c33c1709e5a72b202ad4d7a3b8152de3dc698cef6f675ecdaf20a3b",
    "receiptsRoot":
↪"0x69a04fa6073e4fc0947bac7ee6990e788d1e2c5ec0fe6c2436d0892e7f3c09d2",
    "sealer": "0x2",
    "sealerList": [

↪"11e1be251ca08bb44f36fdeedfaeca40894ff80dfd80084607a75509edeaf2a9c6fee914f1e9efda571611cf4575a1
↪",

↪"78a313b426c3de3267d72b53c044fa9fe70c2a27a00af7fea4a549a7d65210ed90512fc92b6194c14766366d434235
↪",

↪"95b7ff064f91de76598f90bc059bec1834f0d9eeb0d05e1086d49af1f9c2f321062d011ee8b0df7644bd54c4f9ca3d
↪",

↪"b8acb51b9fe84f88d670646be36f31c52e67544ce56faf3dc8ea4cf1b0ebff0864c6b218fdcd9cf9891ebd414a9958
↪"
    ],
    "signatureList": [
      {
        "index": "0x2",
        "signature":
↪"0xae098aabc63a53b8dcb57da9a87f13aebf231bfe1704da88f125cee6b4b30ee0609d0720a97bed1900b96bc3e7a6
↪"
      },
      {
        "index": "0x0",
        "signature":
↪"0x411cb93f816549eba82c3bf8c03fa637036dcdee65667b541d0da06a6eaea80d16e6ca52bf1b08f77b59a834bffb
↪"
      },
      {
        "index": "0x3",
        "signature":
↪"0xb5b41e49c0b2bf758322ecb5c86dc3a3a0f9b98891b5bbf50c8613a241f05f595ce40d0bb212b6faa32e985467548
↪"
      }
    ],
    "stateRoot":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp": "0x173ad8703d6",
    "transactionsRoot":
↪"0xb563f70188512a085b5607cac0c35480336a566de736c83410a062c9acc785ad"
  }
}
```


21.15 getBlockHeaderByNumber

返回根据区块高度查询的区块头

21.15.1 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)

- includeSignatures: bool - 包含签名列表标志(true显示签名列表)

21.15.2 返回值

- 示例

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockHeaderByNumber","params
↪":[1,"0x0",true],"id":1}' http://127.0.0.1:8545 |jq
```

Result见getBlockHeaderByHash

21.16 getBlockHashByNumber

返回根据区块高度查询的区块哈希

21.16.1 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)

21.16.2 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"
}
```

21.17 getTransactionByHash

返回根据交易哈希查询的交易信息

21.17.1 参数

- groupID: unsigned int - 群组ID
- transactionHash: string - 交易哈希

21.17.2 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
  - blockLimit: string - 交易的blockLimit，用于交易防重
  - chainId: string - 交易所在的链ID
  - extraData: string - 交易内的extraData
  - groupId: string - 交易所在的群组ID
  - 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
  - signature: 交易签名，包含r, s, v以及序列化的交易签名signature
  - 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,
↪"0x3961fac263d8e640b148ddcfafd71d2069e93a006abc937c32fb16cfa96e661d"],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
 {
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "blockHash":
↪"0xfa639d1454362a8cdfcab1ca1948a5defaf7048b28f67e80780ab1e24e8f8c59",
    "blockLimit": "0x100",
    "blockNumber": "0x1",
    "chainId": "0x1",
    "extraData": "0x",
    "from": "0x57c7be32cbfb3bfed4fddc87efcc735b4e945fb3",
    "gas": "0x2faf080",
    "gasPrice": "0xa",
    "groupId": "0x1",
    "hash": "0x3961fac263d8e640b148ddcfafd71d2069e93a006abc937c32fb16cfa96e661d",
    "input":
↪"0x4ed3885e0000000000000000000000000000000000000000000000000000000000000020000000000000000000000
↪",
    "nonce": "0x3eb675ec791c2d19858c91d0046821c27d815e2e9c151604912205000002968",
    "signature": {
      "r": "0x9edf7c0cb63645442aff11323916d51ec5440de979950747c0189f338afdcefd",
      "s": "0x2f3473184513c6a3516e066ea98b7cfb55a79481c9db98e658dd016c37f03dcf",
      "signature":
↪"0x9edf7c0cb63645442aff11323916d51ec5440de979950747c0189f338afdcefd2f3473184513c6a3516e066ea98b
↪",
```

(continues on next page)

(续上页)

```
      "v": "0x0"
    },
    "to": "0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744",
    "transactionIndex": "0x0",
    "value": "0x0"
  }
}
```

21.18 getTransactionByBlockHashAndIndex

返回根据区块哈希和交易序号查询的交易信息

21.18.1 参数

- `groupID`: `unsigned int` - 群组ID
- `blockHash`: `string` - 区块哈希
- `transactionIndex`: `string` - 交易序号

21.18.2 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

21.19 getTransactionByBlockNumberAndIndex

返回根据区块高度和交易序号查询的交易信息

21.19.1 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

21.19.2 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

21.20 getTransactionReceipt

返回根据交易哈希查询的交易回执信息

21.20.1 参数

- groupID: unsigned int - 群组ID
- transactionHash: string - 交易哈希

21.20.2 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",
```

(continues on next page)

(续上页)

```
        "blockNumber": "0xa",
        "contractAddress": "0x0000000000000000000000000000000000000000",
        "from": "0xcdcce60801c0a2e6bb534322c32ae528b9dec8d2",
        "gasUsed": "0x1fb8d",
        "input":
↪"0xb602109a0000000000000000000000000000000000000000000000000000000000000080000000000000000000000
↪",
        "logs": [ ],
        "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
        "output":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        "root":"0x38723a2e5e8a17aa7950dc008209944e898f69a7bd10a23c839d341e935fd5ca
↪",
        "status": "0x0",
        "to": "0x15538acd403ac1b2ff09083c70d04856b8c0bdfd",
        "transactionHash":
↪"0x708b5781b62166bd86e543217be6cd954fd815fd192b9a124ee9327580df8f3f",
        "transactionIndex": "0x0"
    }
}
```

21.21 getPendingTransactions

返回待打包的交易信息

21.21.1 参数

- `groupID`: `unsigned int` - 群组ID

21.21.2 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
{
```

(continues on next page)

(续上页)

```
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        [
            {
                "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
                "gas": "0x9184e729fff",
                "gasPrice": "0x174876e7ff",
                "hash":
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f",
                "input":
↪"0x48f85bce000000000000000000000000000000000000000000000000000000000000001bf5bd8a9e7ba8b936ea70
↪",
                "nonce":
↪"0x65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f",
                "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
                "value": "0x0"
            }
        ]
    }
}
```

21.22 getPendingTxSize

返回待打包的交易数量

21.22.1 参数

- groupID: unsigned int - 群组ID

21.22.2 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"
}
```

21.23 getCode

返回根据合约地址查询的合约数据

21.23.1 参数

- groupID: unsigned int - 群组ID
- address: string - 合约地址

21.23.2 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,
    "jsonrpc": "2.0",
    "result":
↪"0x60606040523415600b57fe5b5b60928061001a6000396000f300606060405260000357c0100000000000000000000000
↪"
}
```

21.24 getTotalTransactionCount

返回当前交易总数和区块高度

21.24.1 参数

- groupID: unsigned int - 群组ID

21.24.2 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"
```

(continues on next page)

(续上页)

```
    }
}
```

21.25 getSystemConfigByKey

返回根据key值查询的value值

21.25.1 参数

- `groupID`: `unsigned int` - 群组ID
- `key`: `string` - 支持tx_count_limit和tx_gas_limit

21.25.2 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"
}
```

21.26 call

执行一个可以立即获得结果的请求，无需区块链共识

21.26.1 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

21.26.2 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",
        "status": "0x0"
    }
}
```

21.27 sendRawTransaction

执行一个签名的交易，需要区块链共识

21.27.1 参数

- groupID: unsigned int - 群组ID
- rlp: string - 签名的交易数据

21.27.2 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,
↪"f8d3a003922ee720bb7445e3a914d8ab8f507d1a647296d563100e49548d83fd98865c8411e1a3008411e1a300820
↪"],"id":1}' http://127.0.0.1:8545 |jq

// RC2 Result
{
```

(continues on next page)

(续上页)

```
    "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
```

21.28 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

21.28.1 参数

- `groupID`: `unsigned int` - 群组ID
- `rlp`: `string` - 签名的交易数据

21.28.2 返回值

- `string` - 交易哈希
- 示例：同`sendRawTransaction`，参考这里

21.29 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

21.29.1 参数

- `groupID`: `unsigned int` - 群组ID
- `transactionHash`: `string` - 交易哈希

21.29.2 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":
↪"0x8a42ebe900000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪",
      "nonce": "0x208f6fd78d48aad370df51c6fdf866f8ab022de765c2959841ff2e81bfd9af9",
      "to": "0xd6c8a04b8826b0a37c6d4aa0eaa8644d8e35b79f",
      "transactionIndex": "0x32",
      "value": "0x0"
    },
    "txProof": [
      {
        "left": [
          "30f0abfcf4ca152815548620e33d21fd0feaa7c78867791c751e57cb5aa38248c2",
          "31a864156ca9841da8176738bb981d5da9102d9703746039b3e5407fa987e5183e"
        ],
        "right": [
          "33d8078d7e71df3544f8845a9db35aa35b2638e8468a321423152e64b9004367b4",
          "34343a4bce325ec8f6cf48517588830cd15f69b60a05598b78b03c3656d1fbf2f5",
```

(continues on next page)

(续上页)


```
            "35ac231554047ce77c0b31cd1c469f1f39ebe23404fa8ff6cc7819ad83e2c029e7",
            "361f6c588e650323e03afe6460dd89a9c061583e0d62c117ba64729d2c9d79317c",
            "377606f79f3e08b1ba3759eceada7fde3584f01822467855aa6356652f2499c738",
            "386722fe270659232c5572ba54ce23b474c85d8b709e7c08e85230afb1c155fe18",
            "39a9441d668e5e09a5619c365577c8c31365f44a984bde04300d4dbba190330c0b",
            "3a78a8c288120cbe612c24a33cce2731dd3a8fe6927d9ee25cb2350dba08a541f5",
            "3bd9b67256e201b5736f6081f39f83bcb917261144384570bdbb8766586c3bb417",
            "3c3158e5a82a1ac1ed41c4fd78d5be06bf79327f60b094895b886e7aae57cff375",
            "3de9a4d98c5ae658ffe764fbfa81edfdd4774e01b35ccb42beacb67064a5457863",
            "3e525e60c0f7eb935125f1156a692eb455ab4038c6b16390ce30937b0d1b314298",
            "3f1600afe67dec2d21582b8c7b76a15e569371d736d7bfc7a96c0327d280b91dfc"
          ]
        },
        {
          "left": [
            "3577673b86ad4d594d86941d731f17d1515f4669483aed091d49f279d677cb19",
            "75603bfea5b44df4c41fbb99268364641896334f006af3a3f67edaa4b26477ca",
            "1339d43c526f0f34d8a0f4fb3bb47b716fdfde8d35697be5992e0888e4d794c9"
          ],
          "right": [
            "63c8e574fb2ef52e995427a8acaa72c27073dd8e37736add8dbf36be4f609ecb",
            "e65353d911d6cc8ead3fad53ab24cab69a1e31df8397517b124f578ba908558d"
          ]
        },
        {
          "left": [],
          "right": []
        }
      ]
    }
}
```


21.30 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

21.30.1 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",
          "3b0039ce903e275170640f3a464ce2e1adc2a7caee41267c1954693650740324 01",
          "3ca53017502028a0cb5bbf6c47c4779f365138da6910ffcfebf9591b45b89abd48",
          "3de04fc8766a344bb73d3fe6360c61d036e2eeedfd9ecdb86a0498d7849ed591f0",
          "3e2fc73ee22c4986111423dd20e8db317a313c9df29fa5aa3090f27097ecc4e1a9",
          "3fa7d31ad5c6e7bba3f99f9efc03ed8dd97cb1504003c34ad6bde5a662481f00a0"
        ]
      },
      {
        "left": [
          "cd46118c0e99be585ffcf50423630348dbc486e54e9d9293a6a8754020a68a92",
          "3be78209b3e3c83af3668ec3192b5bf232531323ef66b66de80a11f386270132",
          "bd3a11d74a3fd79b1e1ea17e45b76eda4d25f6a5ec7fc5f067ea0d086b1ce70f"
        ],
        "right": [
          "6a6cefef8b48e455287a8c8694b06f4f7cb7950017ab048d6e6bdd8029f9f8c9",
          "0a27c5ee02e618d919d228e6a754dc201d299c91c9e4420a48783bb6fcd09be5"
        ]
      },
      {
        "left": [],
        "right": []
      }
    ],
```

(continues on next page)

(续上页)

```
    "transactionReceipt": {
      "blockHash":
↪"0xcd31b05e466bce99460b1ed70d6069fdfbb15e6eef84e9b9e4534358edb3899a",
      "blockNumber": "0x5",
      "contractAddress": "0x0000000000000000000000000000000000000000",
      "from": "0x148947262ec5e21739fe3a931c29e8b84ee34a0f",
      "gasUsed": "0x21dc1b",
      "input":
↪"0x8a42ebe900000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪",
      "logs": [],
      "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
      "output": "0x",
      "root": "0xc3b4185963c78a4ca8eb90240e5cd95371d7217a9ce2bfa1149d53f79c73afbb",
      "status": "0x0",
      "to": "0xd6c8a04b8826b0a37c6d4aa0eaa8644d8e35b79f",
      "transactionHash":
↪"0xd2c12e211315ef09dbad53407bc820d062780232841534954f9c23ab11d8ab4c",
      "transactionIndex": "0x32"
    }
  }
}
```

21.31 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

21.31.1 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

21.31.2 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":"generateGroup","params":[2, {
↪"timestamp":"1585214879000","sealers":[
↪"70f18c055d366615e86df99f91b6d3f16f07d66293b203b73498442c0366d2c8ff7a21bb56923d9d81b1c291625188
↪",
↪"dde37f534885f08db914566efeb03183d59363a4be972bbcdde25c37f0b350e1980a7de4fdc4aaf956b931aab00b73
↪",
↪"d41672b29b3b1bfe6cad563d0f0b2a2735865b27918307b85085f892043a63f681ac8799243e920f7bb144b111d854
↪",
↪"7ba2717f81f38e7371ccdbe173751f051b86819f709e940957664dbde[illegible]d73741ba42afc
↪"],"enable_free_storage":true}],"id":1}' http://127.0.0.1:8545 | jq
```

(continues on next page)

(续上页)

```
// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 generated successfully"
  }
}
```

21.32 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

21.32.1 参数

- `groupID: unsigned int` - 群组ID

21.32.2 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":"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"
  }
}
```

21.33 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

21.33.1 参数

- `groupID: unsigned int` - 群组ID

21.33.2 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 -X POST --data '{"jsonrpc":"2.0","method":"stopGroup","params":[2],"id":1}'␣
↪http://127.0.0.1:8545 | jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "Group 2 stopped successfully"
  }
}
```

21.34 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

21.34.1 参数

- groupID: unsigned int - 群组ID

21.34.2 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
- 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"
  }
}
```

21.35 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

21.35.1 参数

- groupID: unsigned int - 群组ID

21.35.2 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"
  }
}
```

21.36 queryGroupStatus

根据群组ID查询相应群组的状态

21.36.1 参数

- groupID: unsigned int - 群组ID

21.36.2 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,
  "jsonrpc": "2.0",
  "result": {
    "code": "0x0",
    "message": "",
    "status": "STOPPED"
  }
}
```

21.37 getNodeInfo

获取请求节点的NodeID，Topic等信息。

21.37.1 参数

- 无

21.37.2 返回值

- NodeInfo: 获取节点信息，包括节点所在的机构、节点名、节点NodeID以及节点订阅的Topic
- 示例:

```
curl -X POST --data '{"jsonrpc":"2.0","method":"getNodeInfo","params":[],"id":1}'␣
↪http://127.0.0.1:8545 |jq
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "Agency": "agency",
    "IPAndPort": "0.0.0.0:30300",
    "Node": "node1",
    "NodeID":
↪"2263a586cba1c1b8419f0dd9e623c7db79b2ba9127367aa854b1493e218efaa2ef58b08de1b36defdeb1d407449014
↪",
    "Topic": []
  }
}
```

21.38 getBatchReceiptsByBlockNumberAndRange

根据区块高度和交易范围，批量返回指定群组的交易回执信息。

21.38.1 参数

- groupID: 群组ID;
- blockNumber: 请求获取的回执信息所在的区块高度;
- from: 需要获取的回执起始索引;
- count: 需要批量获取的回执数目，当设置为-1时，返回区块内所有回执信息;
- compressFlag: 压缩标志，当设置为false时，以明文的形式返回批量交易回执信息; 当设置为true时, 以zlib格式压缩批量交易回执，并将压缩后的回执信息以Base64编码的格式返回。

21.38.2 返回值

当compressFlag为true，返回明文形式的批量交易回执信息; 当compressFlag为false时，返回zlib压缩后的Base64编码的批量回执信息，返回的具体字段如下:

- blockHash: 回执所在的区块哈希
- blockNumber: 回执所在的区块高度
- receiptRoot: 回执所在的区块的receiptRoot
- receiptsCount: 区块中包含的回执数目

每条回执信息包含的字段如下:

- contractAddress: 回执对应的合约地址
- from: 交易回执对应的外部账户信息
- gasUsed: gas信息
- logs: 回执中包含的event log信息
- output: 交易执行结果
- status: 回执状态
- to: 目标账户地址
- transactionHash: 产生回执的交易哈希
- transactionIndex: 产生回执的交易在区块中的索引

示例:

```
# 获取块高为1的区块所有回执明文信息
curl -X POST --data '{"jsonrpc":"2.0","method":
↪"getBatchReceiptsByBlockNumberAndRange","params":[1,"0x1","0","-1",false],"id":1}
↪' http://127.0.0.1:8545 |jq
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "blockInfo": {
      "blockHash":
↪"0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2",
      "blockNumber": "0x1",
      "receiptRoot":
↪"0x69a04fa6073e4fc0947bac7ee6990e788d1e2c5ec0fe6c2436d0892e7f3c09d2",
      "receiptsCount": "0x1"
    },
    "transactionReceipts": [
      {
        "contractAddress": "0x0000000000000000000000000000000000000000",
```

(continues on next page)

(续上页)

```
        "from": "0xb8e3901e6f5f842499fd537a7ac7151e546863ad",
        "gasUsed": "0x5798",
        "logs": [],
        "output":
↪"0x08c379a0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000
↪",
        "status": "0x1a",
        "to": "0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744",
        "transactionHash":
↪"0xc6ec15fd1e4d696e66d7fbef6064bda3ed012bcb7744d09903ee289df65f7d53",
        "transactionIndex": "0x0"
      }
    ]
  }
}

# 获取区块高度为1的压缩编码后的所有回执信息
curl -X POST --data '{"jsonrpc":"2.0","method":
↪"getBatchReceiptsByBlockNumberAndRange","params":[1,"0x1","0","-1",true],"id":1}
↪' http://127.0.0.1:8545 |jq
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": "eJylUrFu3DAM3fsZmm+gRImUbiu6NEuHAJ2KDLJEpkETK7B9QIDD/
↪XsZX4AgQIcCEWCLj+YjafKd3fQ42p+bWYc7nq/ge11/
↪u6ODlyaaQ8XmhZmh1iiYc9XUGKRwFc9ckTy3BAGqQAqTRCipa+9YanCHa8Ifp6dJlj2lN98iTR6et9sxtt1HpULUSsAoURu
↪Yw5tu3CHf8dXZtzPapbV97X2RddwL857FquoynnTNlwQJeSJPmGGIp2hNyZWvfJy8pUias3Tj3df25St9pi
↪TUPNhwJ2q5dM2cpFArCpqmoByj+7D/d1GTNJ/
↪UNBU7FRKizjqJElCcekXp4MPUJrYUHUyBKBJeK1BStu1+THszd3m5Ls5d7i5f/gLIg98l"
}
```

21.39 getBatchReceiptsByBlockHashAndRange

根据区块哈希和交易范围，批量返回指定群组的交易回执信息。

21.39.1 参数

- `groupID`: 群组ID;
- `blockHash`: 包含需要获取的回执信息的区块的哈希;
- `from`: 需要获取的回执起始索引;
- `count`: 需要批量获取的回执数目，当设置为-1时，返回区块内所有回执信息;
- `compressFlag`: 压缩标志，当设置为false时，以明文的形式返回批量交易回执信息; 当设置为true时, 以zlib格式压缩批量交易回执，并将压缩后的回执信息以Base64编码的格式返回。

21.39.2 返回值

当`compressFlag`为true，返回明文形式的批量交易回执信息; 当`compressFlag`为false时，返回zlib压缩后的Base64编码的批量回执信息，返回的具体字段如下:

- `blockHash`: 回执所在的区块哈希
- `blockNumber`: 回执所在的区块高度
- `receiptRoot`: 回执所在的区块的receiptRoot

- `receiptsCount`: 区块中包含的回执数目

每条回执信息包含的字段如下:

- contractAddress: 回执对应的合约地址
- from: 交易回执对应的外部账户信息
- gasUsed: gas信息
- logs: 回执中包含的event log信息
- output: 交易执行结果
- status: 回执状态
- to: 目标账户地址
- transactionHash: 产生回执的交易哈希
- transactionIndex: 产生回执的交易在区块中的索引

示例:

```
# 获取区块哈希为0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2的区
块所有回执明文信息
 curl -X POST --data '{"jsonrpc":"2.0","method":
↪"getBatchReceiptsByBlockHashAndRange","params":[1,
↪"0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2","0","1",
↪false],"id":1}' http://127.0.0.1:8545 |jq
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "blockInfo": {
      "blockHash":
↪"0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2",
      "blockNumber": "0x1",
      "receiptRoot":
↪"0x69a04fa6073e4fc0947bac7ee6990e788d1e2c5ec0fe6c2436d0892e7f3c09d2",
      "receiptsCount": "0x1"
    },
    "transactionReceipts": [
      {
        "contractAddress": "0x0000000000000000000000000000000000000000",
        "from": "0xb8e3901e6f5f842499fd537a7ac7151e546863ad",
        "gasUsed": "0x5798",
        "logs": [],
        "output":
↪"0x08c379a0000000000000000000000000000000000000000000000000000000000000002000000000000000000000000
↪",
        "status": "0x1a",
        "to": "0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744",
        "transactionHash":
↪"0xc6ec15fd1e4d696e66d7fbef6064bda3ed012bcb7744d09903ee289df65f7d53",
        "transactionIndex": "0x0"
      }
    ]
  }
}

# 获取区块哈希为0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2的压
缩编码后的所有回执信息
curl -X POST --data '{"jsonrpc":"2.0","method":"getBatchReceiptsByBlockHashAndRange
↪","params":[1,"0xcef82a3c1e7770aa4e388af5c70e97ae177a3617c5020ae052be4095dfdd39a2
↪","0","1",true],"id":1}' http://127.0.0.1:8545 |jq
{
```

(continues on next page)

(续上页)

```
  "id": 1,
  "jsonrpc": "2.0",
  "result": "eJylUrFu3DAM3fsZmm+gRImUbiu6NEuHAJ2KDLJEpkETK7B9QIDD/
↪XsZX4AgQIcCEWCLj+YjafKd3fQ42p+bWYc7nq/ge11/
↪u6ODlyaaQ8XmhZmh1iiYc9XUGKRwFc9ckTy3BAGqQAqTRCipa+9YanCHa8Ifp6dJlj2lN98iTR6et9sxtt1HpULUSsAoURuU
↪Yw5tu3CHf8dXZtzPapbV97X2RddwL857FquoynnTNlwQJeSJPmGGIp2hNyZWvfJy8pUias3Tj3df25St9piUs21+O4f23n7u
↪TUPNhwJ2q5dM2cpFArCpqmoByj+7D/d1GTNJ/
↪UNBU7FRKizjqJElCcekXp4MPUJrYUHUyBKBJeK1BStu1+THszd3m5Ls5d7i5f/gLIg98l"
}
```

21.40 错误码描述

21.40.1 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：

21.40.2 交易回执状态

21.40.3 Precompiled Service API 错误码

21.40.4 动态群组管理 API 状态码

CHAPTER 22

深入浅出FISCO BCOS

22.1 介绍

FISCO BCOS开源社区已沉淀120+篇技术文章，这些文章涉猎理论、开发以及应用案例等方面，有助于开发者快速上手、全面掌握FISCO BCOS，依据平台关键特性，社区对这些文章进行整理编排成《深入浅出FISCO BCOS》，供各位浏览阅读。

《深入浅出FISCO BCOS》分为六大章节，由社区全体成员共创完成并不断优化，社区开发者可以在GitHub上提交PR贡献文章，分享FISCO BCOS开发经验与使用心得。

22.2 区块链的概念与原理

22.2.1 亲朋好友都能看懂的区块链

作者：张开翔 | FISCO BCOS 首席架构师

据新华社10月25日晚消息，中共中央政治局10月24日下午就区块链技术发展现状和趋势进行第十八次集体学习。中共中央总书记习近平在主持学习时强调，区块链技术的集成应用在新的技术革新和产业变革中起着重要作用。

随后“什么是区块链”、“区块链有什么用”、“怎么掌握区块链”这几个问题响不绝耳。这里尽量用通俗易懂的语句回答这些问题，期望即使是从没接触过区块链的人也能搞懂这项已经成为国家级现象的神奇技术。

什么是区块链?

专业解释里，一般都会带上“分布式网络”、“密码学”、“共识算法”、“智能合约”这些有点晦涩的术语，这里打算举一个例子。

要理解区块链，首先要接受一个设定：有个账本，要记录的是一群人之间的公共账目（比如班费、物业费、公益捐款的收支），这个账本由大家一起来记账。选一个人，在账本的某一页，一行一行的记录明细，当一页记满后，大家都去核对账目，正确的话，大家签字认可这一页的所有账目。记满一页后，再选另一个人开始记接下来的一页。现在有意思的事情来了，新的一页首先要把上一页的一些摘要特征（比如页码、余额、人数、条数什么的...）抄写下来放在页首以供对照，免得前一页被改了或丢了还无

据可查。然后，再一条条记账，记满一页后，核对、签名确认...依次反复。这样，账本的一页和一页之间就形成了“证据链”。

更重要的是，账本上已经签名确认的每一页，所有人都会一字不漏地复制一份，放到自己家里，以免少数人篡改、污损、丢失账目。这样，账本的每一页是一个“区块(Block)”，一页一页之间形成前后连贯的证据链，每个人之间构成了多点的网络，这就是“区块链”的概要原型了。这里面的关键点在于，这个账本一定是一群人的账，并不是一个人的账。如果是一个人的账，自己拿个小本本记就好了，为什么要整这么麻烦。就是因为这一群人互相之间并不完全信任，记账过程可能出纰漏，所以必须用这么繁琐的步骤，让大家平等参与，一起保证账目的准确和公允性，产生的结果大家保存，永远不会丢也不会错。

上面的举例看起来步骤繁琐，还要求每个人都一字不差地复刻和保存，那么就需要技术手段来帮助大家记账。这又回到技术环节，随着计算机科技的发展，无论是网络、密码学、数据结构、分布式算法等技术都趋于成熟，区块链领域把这些技术组合在一起，一揽子完成了从记账到分发到验证到保存的整个过程。运用技术服务整个群体，这就是神奇的区块链技术。

区块链是比特币吗?

首先，可以肯定地回答：不是。比特币确实是早期的一种区块链技术的应用，它提供了一种价格波动不定的虚拟代币，之后还有许多“山寨币”效仿比特币运作，这些虚拟代币风险都很高，不被法律法规所容。现在国家提倡的“区块链”，已经和虚拟代币明确划清界限，更强调研究核心技术，加固系统安全，控制业务风险，也更强调区块链技术和具体实体经济融合，如合规合法的金融业务、政务、智慧城市、农业工业等，用技术去解决实际的问题，发掘创新场景。区块链并不是发币割韭菜的工具，不是资金盘的幌子，而是越来越成熟和规范的创新技术，即将像互联网、APP等常见的技术服务一样，潜移默化地影响大家的日常生活。

区块链只是一种技术吗?

如果只是谈技术，可能还没参透区块链的魅力。上面说到，区块链的记录一定是多人参与的，那么大家为什么要花精力参与到记账这个麻烦的事情里，为什么要出成本买本子和笔去记账？这就牵涉到“多方协作”了。现代社会很多事情是很难一个人完成的，必须大家一起合作才能达到1+1>2的效应，但是人多了，账就复杂，可能出现类似“信息不对称”之类的问题，那么就需要把大家组织成联盟，共同记账，共享数据，使一切都发生在阳光下，这就消除了“信息不对称”。这样的协作环境，是可信任和高效的，每个人的利益都得到了保护，营商环境得以良性发展。这就是更重要的“区块链思维”，让更多人参与进来按规则运作：诚实做事将获得应有的收益，如果作弊自然就会路人皆知，千夫所指。技术就是帮助实现这种模式的基础。

区块链能做什么?

综合区块链技术和区块链思维，只要是牵涉到多方参与的，有着复杂的账目来往和数据共享要求的场景，都可以考虑使用区块链。区块链不但能记账，也能记录人和物的信息，并通过共同对信息进行验证，固化下来，而变得可信。

在**传统金融场景**中，金融机构每天都需要进行大量的对账清算工作，如果交易发生在区块链上，那么就可以起到“交易即对账”的作用，极大提升运营效率。更进一步，在供应链、跨境支付、票据等典型的金融业务中，都可以用区块链在合作伙伴之间构建公众账本，海量的资金和资产在链上记录、核验、交易，账本可信、准确、高效，可以极大扩大金融业务规模、提升运作效率、降低成本和风险，能更好地解决小微企业融资难、银行风控难、监管接入难等一系列问题。另外，合规且标准化数字资产也可以在区块链链上定义、流转、承兑，可以构建创新的商业模式。

在**司法存证领域**，如合同的签署过程被全部记录在区块链上，并且由包括司法机构在内的参与方共同见证，那么发生纠纷时，司法机构就可以一键从链上提取证据进行核验，证明合同从诞生到取证时都没有修改过。由于链上有司法机构参与，这样的证据已经具备一定的司法效力，大大减少了司法成本。

在**政务服务**方面，区块链用于身份认证，可以让人们的身份证明在一处验证，处处可用。用区块链连接多个部门，就能做到“办事时数据多跑腿，用户少跑腿”，并且“证明我是我，我妈是我妈”的事情就不复存在，还能保护用户数据隐私。

此外，区块链技术还可以被运用到版权、物业、物联网、智慧城市、新能源、文娱、人才交流等海量领域。正确地将区块链技术和区块链思维，与国计民生相关的事务深入融合，能够大幅提升智能化、精准化水平，使得产业之间互联互通，保障生产要素在区域内有序高效流动。

点击查看区块链应用案例精编（附高清PDF全本下载）

为什么国家这么重视区块链?

区块链有巨大的发展潜力，本身蕴含着极密集的技术含量，对技术有足够研究和把控，才能使自身发展不受外界影响和制约。区块链技术将会被用到与国计民生有关的广泛场景里，牵涉到许多人的财务和个人信息，甚至是金融行业、政务民生等关键领域的重要信息，重视技术和运作的安全可控，才能保护财务安全、信息安全，乃至社会安全。所以，当下国家给我们提出了明确的要求："注意区块链技术发展现状和趋势，提高运用和管理区块链技术能力"，这样才能更好地建设网络强国、发展数字经济、助力经济社会发展。

我国的区块链发展现状如何?

我国产业界在区块链领域投入很早、力度很大，诸多大型金融企业、互联网公司、科技公司都在进行区块链研究，在过去几年中，已经逐步解决或接近解决了区块链领域的一系列核心问题，如性能、安全、易用性、合规等，在这个过程中也积累了大量的知识产权和专利。

我们重点发展的区块链领域是"联盟链"，联盟链和匿名的、有虚拟代币的"公有链"（诸如比特币、以太坊等）有极大不同，联盟链不发币不挖矿，摈弃了野蛮运作的弊端，能更规范、合法的运作，可有效服务于实体经济。

由微众银行、深圳市金融科技协会等国内机构共同发起成立的区块链联盟——金链盟，掌握多项核心技术，于2017年开源发布了以FISCO BCOS为代表的区块链底层平台和系列解决方案。这一系列开源项目都安全可控、性能优异、免费且易用，致力于服务实体经济，在大量关系国计民生的产业中已广泛落地，技术和模式得到了大量实际案例的验证。

另外值得一提的是，FISCO BCOS采用技术开源的路径加速行业发展，孵化拓展了国内最大和最活跃的行业生态社区，在推进中重视人才培养，在高校、社会、产业机构中培养了大量的区块链专业人才，帮助区块链技术和产业发展加速人才方面的突破。

我该如何切入区块链领域?

如果您是一位**对区块链感兴趣的非工科人士**，那么可以多关注这个领域里正规和权威的公众号、主流媒体，多了解区块链相关的新闻动态、产业趋势，建立正确的区块链概念，消除来自虚拟币和资金盘的噪声，并逐步将诸如"多方对等协作"、"公开透明"等"区块链思维"对照到当前的工作生活中来。

如果您是一位**工科、信息学方面的学生**，那么建议您在学校打好学业基础，比如数学、算法和数据结构、概率论、博弈论、密码学等，掌握一两门主要的计算机语言，向经验丰富的教授和老师请教，积极参与学校的区块链社团活动，也可以参加到FISCO BCOS区块链开源技术社区和学校联合开展的课程中来，三天上手区块链应用。

如果您是**商科、文科背景**，可以重点关注分布式商业的趋势，善用人文、经管、经济理论和博弈论方面的经验和知识，应对多方协作的种种挑战，用创新思维开拓可服务实体经济、人民生活的新场景。

如果您是**在IT界工作的专业人士**，需要区块链技术的加持，欢迎您加入FISCO BCOS区块链开源技术社区。这里有活跃的微信群讨论技术和产业问题，有公众号定期推送技术解析文章、活动通知等。更重要的是，您可以获得全面开源且免费的区块链解决方案，从区块链底层平台到身份、物联网、图形化工具、云服务应有尽有，有丰富的文档和专业的线上线下课程帮助您快速学习，从入门到精通。区块链技术可以连接多项技术，包括人工智能、物联网、大数据、金融科技等，技术界人士和区块链的结合，将大有可为。

如果您已经是**硬核区块链专业人士**，且愿意参与到开源社区里来，欢迎您关注FISCO BCOS区块链开源技术社区，互通有无，共同深入研究核心技术，为开源社区添砖加瓦，进行代码优化、文档编著等，一起实现更多的功能，共同打造最好用的区块链技术平台。

如果您已经面对各种实际场景，**希望用区块链解决您的具体问题**，FISCO BCOS区块链开源技术社区有诸多的金融机构、科技公司和产业专家，可以与您一起整合资源，进行业务合作，完成系统建设和业务运营。

区块链已经触达当下，更将影响未来，和一个具备无限潜力的趋势共同成长，现在，就是最好的时节。

22.2.2 反常理，反直觉，区块链是怎样的一种“分布式系统”

作者：张开翔 | FISCO BCOS 首席架构师

我们经常看到“区块链是分布式系统”的说法，并推论出区块链先天具备分布式系统的优势，仿佛作为分布式系统，规模就该足够大，数据就该足够分散。 事实上，典型区块链有很多特征和常见的分布式系统不同，甚至是相悖的，为此，区块链曾被戏称为“最慢的分布式数据库”。 **其实区块链之所以难以理解，其中一个原因是其设计哲学的“反常理、反直觉”**。笔者本人曾多年在互联网海量服务领域里踩坑，然后转向区块链领域深入研究，也经历过一阵子的观念切换期。

本文不打算全面讲述分布式系统原理和历史，那能写几本书。这里打算从常见的、被人广泛认知的互联网分布式系统出发，聊聊“分布式系统”和区块链有什么异同，对技术和设计的要求有哪些路径分支。

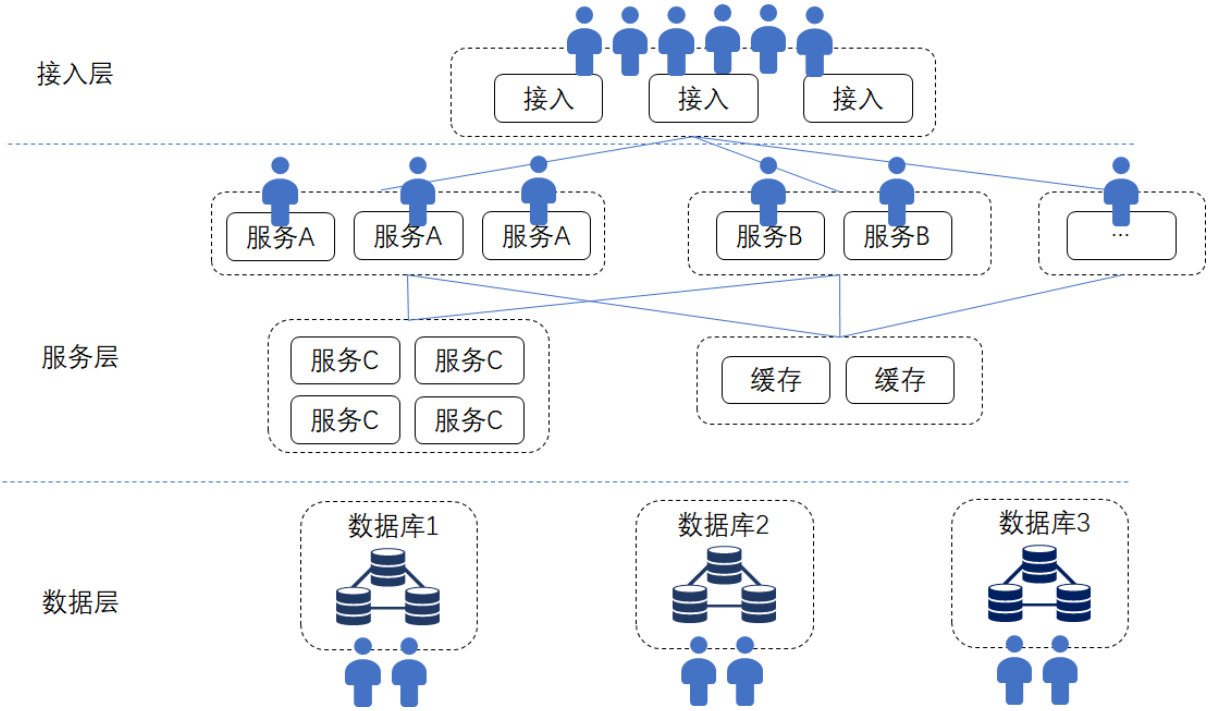
经典的分布式著作《分布式系统概念与设计》中，对“分布式系统”给出的定义其实很笼统：**分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。**

简而言之，只要不是运行在一台机器上或一个进程里的系统，都可以是分布式系统。 比如，大型网站、APP、全民IM社交软件的支撑系统，基于X86体系的新型金融基础设施，都可以是分布式系统。 至于这个分布式系统是属于一个机构的，还是属于多个不同机构维护的，并不是这个定义的重点。

互联网海量服务系统之道

提示：已经比较熟悉互联网海量设计的读者，可以直接跳过本节

如果要求一个互联网服务的架构师，在面对山呼海啸的请求量、保存无限增长的数据的同时，提供良好的用户体验，保证延时很短的响应表现，这位架构师脑子里蹦出来的通常是“分层设计”、“冷热分离”、“平行扩展”、“并行处理”、“分库分表”等等关键字。 互联网海量服务的架构通常是分层的，比如接入层、服务层、数据层等。在接入层根据各种策略进行负载均衡和灵活的路由分发，服务可以按功能分组。比如： 用户要改个人资料，请求发给资料服务； 用户浏览网页，请求查询服务吐数据； 用户要发帖了，转给内容服务； 用户要下单，该交易服务器接手了，每个服务只处理一部分业务逻辑，互相之间通过远程接口互相调用，且本身通常是“无状态”的，不受自身和其他服务处理状态的牵制，这种高内聚和松耦合的设计，利于针对不同服务的特性和负载，进行特定优化和灵活部署； 最后，数据存储是“分片”的，俗称“分库分表”，根据用户标识、业务类型等规则，将数据分摊在不同的数据库实例中，每个数据库实例上只有“部分”数据，仅仅是数据的“子集”。



以UGC、社交、富文本、视频内容为主的互联网服务，对“交易事务性”要求并不那么强，其“分布式”更多的是追求将计算和数据均匀分散在多台服务器上，利用更多的CPU、更大的带宽、更大的内存、更大的磁盘空间来处理请求。一组服务器不够了，立刻增加一组，平行扩容和多活特性表现得淋漓尽致。整个体系会有一些主备和冗余，更多是为了满足服务质量、可用性、备份方面的要求。电商以及一些和金融、支付相关的流程会强调ACID事务性，采用的分布式一致性算法，如Raft、Paxos等，主要是追求多模块之间的最终一致性和系统稳定性。

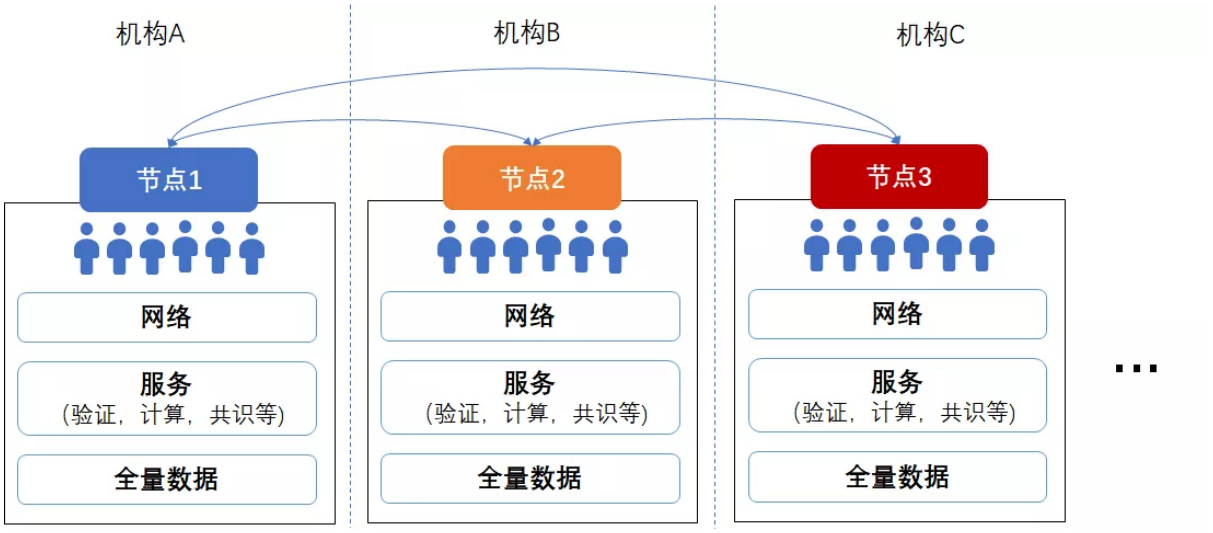
毕竟在同一个机构里的系统，对抗欺诈的要求并不会太苛刻，而且可以根据特定的业务流程将交易分类，将单个事务控制在有限的范围内（如只有用户向商户单向转账），事务和事务之间进行清晰地隔离，这样也比较容易进行平行扩展。

如上图所示，一个典型的多层结构的互联网服务，模块众多，计算和存储分布都比较均匀，哪个模块成为瓶颈，就去增加哪个模块的硬件资源，进行业务路由配置和必要的数据迁移就是了。

像分布式数据库、大数据集群、或者BT网络、CDN这些“分布式系统”，大多也是遵循“**将用户和数据分散到不同的物理设备上**”这个原则，达到简单的“堆硬件”即可平行扩展的效果。硬件越多，技术上的“规模效应”越显著，即存储量更大、计算能力更强、总带宽越多，能服务的用户就越多。最重要的是，整个服务大部分是在同一个机构里，由同一个公司的团队维护，服务和服务之间并不需要解决“信任”的问题，默认信任其他服务的接口给出的结果，简单校验后立刻继续处理，整个系统就是要“快”，要“稳”。如果有一些合作类的接口暴露给外部公司，只需要确保接口层的性能、安全、稳定，以及满足合作规则即可。

“带着锁链跳舞”的区块链

再看区块链，首先**整个网络并不是只有一个机构参与，节点都可能归属不同的机构**，大家一起共同维护网络，共同对交易达成共识。区块链的使命是突破机构边界，解决机构之间的信任和博弈问题，如果只在一个机构内“家养”一个区块链系统，固然可以运作，但收效未必最佳。区块链体系里，为了保证数据可追溯和可验证，避免少数节点篡改数据影响全网，**每个节点保存全部数据**，而不是像互联网服务这样“分库分表”。



让我们算一下，如果有一千万个用户，在互联网系统里分成10个数据库实例保存，则每个数据库实例只保存100万用户的数据。 即使每个数据库实例都是“一主两备”，那么总的数据开销也只是原始数据的3倍。 而经典的区块链设计是，每个节点都如数保存一千万用户的所有数据，且全网节点越多，保存的数据副本就越多。 如全网有100个节点，则全网总的数据存储开销是原始数据的100倍；如有200个节点，全网总的数据存储则达到原始数据的200倍……以此类推，挺触目惊心。 极端情况下，如果把整个地球的数据都上链，链上有1万个节点，那么相当于我们有了1万个地球……然而我们并没有这么多资源吧。

区块链的数据结构本身也是很有意思的一个环节。区块之间是链式关系，新的区块必须基于旧的区块生成。 智能合约生成状态数据是全局性的，常常用类似帕特里夏树、merkle树这种复杂的数据结构进行维护，以便全网追溯、验证和计算。 数据环环相扣，仿佛“铁板一块”，使得按冷热特征、时间和用户等维度切分数据有相当的挑战性，既要使数据可以分布到不同的存储里，又要保证可以快速访问，保证盘根错节的追溯验证。工程难度很大。

交易层面，在一个互相没有强信任关系的网络里，且处理的是大概率和“价值”有关系的交易，**区块链尤其强调交易的全局事务性，保证全网一致性，且为了避免有作恶节点伪造假数据，所有的交易都会在共识机制和网络同步驱动下，在所有的节点上进行排序、运算和校验。** 在支持智能合约的区块链上，每个智能合约被交易调用时，都会在所有节点上把合约代码跑一遍，以确保在每个节点上生成数据的过程是公认的、运行结果是一致的。

可见，区块链并不能把计算量分摊到不同的服务器上，所有的节点都是“复读机”，这就意味着增加节点，也不会增加全网并行计算能力，就算是全网有一万台计算机，速度也跟只有一台是一样的。 如果把这种区块链称为“世界计算机”，可算是全世界步骤最统一、冗余度最高、速度最慢的世界计算机了。

区块链系统的核心是共识算法。

POW挖矿是矿工用算力解一个难题，以争到记账权，再进行记账，并期盼自己的记账结果成为最长链。 POW算法是出了名的又慢又耗电； POS权益类共识，由一批“富翁”抵押资产以获得记账权，轮流记账，不需要算力竞争，性能表现好一些，但依旧是有轮流打包出块然后全网最终确认的过程； PBFT（实用性拜占庭）共识，需要记账者多次往返交互，大多数人参与到多阶段的确认，才能达到最终状态。 总的来说，和互联网服务追求的“无状态”不同，共识服务是“有状态”的，每种共识算法都需要参照网络、数据、记账者参与度、链当前状态（如“区块高度”和“共识阶段”）等等多维度的信息进行决策。

记账节点越多，协作成本越高，尤其是典型的PBFT算法，在节点增多时，共识会越来越慢，交易延迟会越来越高。 如果让节点内部也“集群化”，采用“多层分布式架构”，是否可以实现类似互联网海量服务系统这样的可平行扩展呢？比如，将虚拟机改为多实例，并行计算交易，这样就可以解决计算瓶颈问题。 但计算越快，单位时间可以产生的数据越多，这样也要求硬盘存储更多的数据，而硬盘的容量和IO速度是有限的。 同时，计算快了，节点也会向网络广播更多的数据，节点公网互联的带宽通常不会太大，延时一般也比较明显（几十毫秒起）。 于是，节点和节点之间交互又会命中网络速度的瓶颈，共识时延和数据同步时间变长，节点们就像一串绑在绳子上的蚂蚱，谁也蹦不高。由于存储和网络这些硬条件的天花板存在，集群化的计算再快也没有用了。 最后，节点集群化确实可以使以节点为单位的服务处理能力有一定上升，同时也会带来的架构上的额外复杂性和部署运维成本。

还是引用一个数钱的例子： 有一大堆钱让十个人数，有两种方法。 方法一，可以把钱分成十份，每个

人数一份，这样很快就数完了，人越多，数得越快，但是如果里面有人数错了，或者甚至偷钱，那就有问题了。 于是，为了资金安全，改成方法二，让一个人整理出一叠钱，先数一遍，然后让其他九个人也数一遍，每个人都验算记账后，再换个人整理出下一叠钱，重复上述节奏去数，这样结果肯定是不会错的，且能得到大家的公认。但参与数钱的人增加，并不会加速计数，反而有可能因为人多手杂导致更慢。 方法一就是常见的互联网分布式系统的做法，方法二是区块链。 可见，目的不同，导致设计哲学、系统结构、最终效果都不同。

如何把“好钢用到刀刃上”

综上所述，区块链这种“分布式系统”，存储成本和节点数同比线性增加，而计算效率不升反降，使整个系统显得“贵”和“重”，这和互联网服务的“轻快灵”相背而驰。最要命的是，难以通过增加硬件、带宽、节点数来显著提升并行处理能力和存储量。 但区块链的“网络规模效应”并不体现在硬件和计算上，而是体现在因为“共识”和“信任”上。 区块链通过复杂的算法和博弈，构建了一种可信网络，使得更多人愿意参与到网络里，共同贡献数据和维护网络，体现“协作共赢”这个价值效应。 这样也给我们一个重要启示：既然区块链的计算和存储成本是很高的，其目的是为了达到共识，那么，我们应该让区块链干最该干的事情。

1、哪些数据可以“上链”?

只有多方要在交易过程中用到的、必须共享的关键数据要上链，比如公共账本的账目。

视频、文件、图片、大规模的业务数据，可以生成摘要与交易数据关联，其本体通过其他渠道（如FTP、分布式文件系统等）进行交换。 比如下棋，只需要把棋局结果放到链上，或者把每一步的数据算个摘要放到链上，并不需要每一步都记录到链上（除非认为这也是非常关键的信息）。 毕竟每一笔放到“链上”的数据，都会占用所有节点的硬盘。

2、合约里写什么逻辑?

应该是**写多方共同参与、协作记账、必须全局共识的关键逻辑，而不是牵涉密集计算的逻辑**。 比如，进行复杂查询或建模分析，可以把链上的数据导出来放在链下去做，而不是写在合约里。要清楚，你写的每一行代码，再不是只在自己的服务上跑一次了，而是会在链上所有参与者的节点上跑起来，多写一行代码就会多消耗大家的一点CPU。 所以，区块链上会有类似“Gas上限”这样的机制，来控制合约的代码规模。新一代的合约引擎，更是考虑只提供有限的、可以定制化的商业规则实现，而不是完全的开放式编程。 作为开发者，是面向“自己的电脑”编程，还是面向“大家的电脑”编程，这就是互联网海量服务系统和区块链最大的不同。 开发者必须切换思维模式，切忌滥用区块链上宝贵的计算、存储、网络资源，避免有意或无意的“公地悲剧”，而是精打细算，从全局权衡，找出协作模式和数据共享里的“最大公约数”，把好钢用到刀刃上。

性能也不是什么大问题

区块链的规模化、并发能力，依旧是业界非常关注的研究方向。互联网海量分布式系统的一些思想，对区块链的优化也有很重要的参考意义，包括平行扩展、分库分表、冷热分离、服务集群、负载均衡等等。 现在我们在研究区块链优化时，常常有一种感觉，就是把之前在互联网业里解决经典“C1000K问题（应对百万级的并发量）”的过程再来一遍。 技术优化方案百花齐放，诸如FISCO BCOS的多群组和并行多链架构，基于DAG的交易并发模型，以及行业热议的交易分片、Layer1/2多层网络、链外通道如闪电网络等。 因为要满足区块链苛刻的信任和安全要求，实施这些方案成为“带着锁链跳舞”的艰辛工作。在不远的未来，在力求保证“信任”、“一致性”、“事务性”、“安全”等大前提下，区块链系统也可以具备可观的可扩展性，突破或逼近“不可能三角”的极限并不是梦。

目前，在保证金融级业务的正确性、稳定性的前提下，FISCO BCOS已经做到在PBFT共识、16个节点规模、采用智能合约实现业务逻辑的条件下，达到单链2万多的TPS（硬件条件比较理想的压测环境）。 如果开启多群组、多链跨链、点对点通道等模式，更有显著的平行扩展效果，满足当前的业务需求已经没什么问题了，只要有足够的硬件资源投入，进行合理的调度，百万千万TPS也不是梦。

最后总结一下，我们认为区块链是一种特殊的“分布式系统”，要透彻地理解区块链，应该清晰地、就事论事地与其他类型分布式系统进行区分，理清各自的本质和设计哲学，避免混淆和迷惑。 说到底，“分

布式系统”本身就是一门博大精深的学科，包罗万象，并不能从一而论，而是有太多的精彩和技术路径需要去学习和发掘。

22.2.3 说信任区块链时究竟在信任什么?

作者：张开翔 | FISCO BCOS 首席架构师

当前，“区块链，信任的机器（Blockchain：The trust machine）”已经成为了一句口号，紧接着就是“去中心化、群体共识、不可篡改、高一致性、安全和保护隐私”等一系列听起来很厉害的术语。究竟区块链具有多大的魔力能让人如此信任，或者说，我们在说“**信**”的时候究竟信的是什么。

信息，指身份、资产、价格、地理位置等自然属性和行为信息，它并不是先天可信任的，因为信息散乱、不完整，可能虚假，甚至可能会有人利用信息的不对称性牟利。

把信息整理成结构化数据，通过数据校验的方式，保证其在传播中可保持完整性、全网一致性、可追溯性，不会被恶意篡改；通过冗余存储的方式，保证其公开、共享、可访问，保证数据一直有效。那么，这信息本身就可以被“信任”，从而成为大家的“公共知识”，成为全网参与者都认可的“**最大公约数**”。

如果信息体现着价值，且这些价值被大家认知、认可，能被量化，具有可交易的等价物属性，或可能随着时间增值，甚至得到司法背书承认，这些信息才具有商业意义上的“信用”。

好比我们认识一个人，但不代表我们信任他。然而这个人一贯表现不错，在社群里言行一致，渐渐地获得了大家的信任。这时的信任依旧不等于信用，除非这个人拥有可观的资产，或者其个人历史上有盈利和偿还的能力，未来也大概率能持续持有资产和承兑债务，那么这个人才具备了“**信用**”。

区块链体系基于算法而不是人治，有望通过其独特的分布式架构、加密算法、数据结构、共识机制等，把信息固化成大家的**信任**锚点；有望通过技术手段把各种现实世界的资源转换成可兑付的数字资产，并展开一系列多方商业协作的活动，这就是所谓的 “信息到信任到信用”，甚至于因为区块链这个黑科技的、行之有效且难以理解的玄妙，这个“信”字仿佛升华成了“**信仰**”。

那么我们说信区块链时，信的是什么呢?

信密码学算法

区块链是用算法达成信任的，其中最重要的算法之一，就是密码学。区块链中最基本的密码学应用是HASH摘要、对称加密和非对称加密算法，以及相关的签名验签算法。

HASH算法的旧版本已经被证明可破解而被抛弃了，目前在用的SHA256等算法依旧坚不可破。HASH算法的特性是把一堆数据单向生成一段定长的数据，基本不会发生碰撞，可起到原始数据的“指纹”作用，其单向性不可逆，推不出原始数据，具有一定的抗量子性，是能隐藏原始数据又能在必要时提供校验凭据的最佳方式。

数字签名一般基于公私钥体系，用私钥签名，公钥验签或者反之。数字签名源自密码学的牢靠性，使得不可能有人能伪造别人的私钥签名，所以一个拥有私钥的人可以通过数字签名，对他的资产签名确权，或者在双方交易时，采用对手方的公钥发起交易，将资产转移给对方，对方用自己的私钥才能验签解开，以获得所有权。

AES、RSA、ECC椭圆曲线等几种对称和非对称算法广泛地用于数据加解密、安全通信等场景，其安全级别取决于算法本身和密钥长度，当AES使用128~512位密钥，RSA/ECC采用1024甚至2048位密钥时，

其保护的数据理论上需要普通计算机上亿年的计算时间才能暴力破解。这些算法在商业、科学、军事领域都经受了考验。

加密学领域里还有同态加密、零知识证明、环签名群签名、格密码等新的方向，目前都处于从理论发展到工程的阶段，都在功能、安全强度、效率方面快速优化中，已经可以看到落地使用的可能性了。同时我们也意识到，密码学通常需要经过长期的发展、验证，稳定后才能获得广泛认可，要么实践中经历了大量考验，要么经过权威机构的审核和认证，才能在生产领域大放异彩。密码学里的理论到工程，常常有很长的时间周期。

加密算法的一个基本哲学是**计算成本**，当一个算法保护的资产价值，远低于攻破该算法所需的成本时，就是安全的。但如果用一个算法保护一个无价之宝，自然就会有人不计成本地去攻击获利，所以，密码学的安全，也是辩证的、需要量化的。

随着量子计算机等学说的兴起，经典密码学可能会经受一些挑战，但量子计算机的理论完善和工程实现还有待时日，目前来看，基本上我们可以近乎"无条件"相信区块链里已经采用的密码学算法，同时，区块链领域的实践者也在陆续引入各种抗量子的密码学算法，这是一场持续的博弈。

信数据

区块链的数据结构，无非是区块+链。新区块将自己的区块高度、交易列表，和上一个区块的HASH，共同再生成一个HASH做为新区块的标识，如此循环，形成了一个环环相扣的数据链。这个链条里的任何一个字节甚至一个Bit被修改，都会因为HASH算法的特性被校验发现。

同时，区块数据被广播给全网所有参与者，参与者越多，规模效应越强。少数人即使强行修改、删剪自己的区块数据，也很容易被其他人校验出异常并不予采纳，只有多数人认可的数据得以留存和流传。也就是说，数据是大家人盯人的形态盯着的，且存在多份副本，一旦落地，只要链还在，数据就可以永远留存。

基于容易验证的链式数据结构、群体冗余保存、共同鉴证，区块链数据是"难以篡改"的，所有人拿到的数据也都是一致的，信息公开透明，公共知识得以彰显和固化。

从另一个角度看，数据达到信任，但能否达到"信用"还要看数据的价值，也就是数据本身携带的信息，是否能代表有价的资产、有用的信息，诸如身份、交易关系、交易行为、大数据等等，都能代表一定的商业价值。这些数据如果分享出来，足以构建完整的商业基础。

但如果是在过于强调隐私的场景里，大家愿意分享的信息本来就很少，那样就很难达到信用的"最大公约数"。然而，在当前的商业环境里，信息隔离和隐私保护是硬诉求，信息共享和隐私保护成了严峻的矛和盾的关系，除非整个商业关系和商业逻辑出现了革新。

所以，隐私保护相关的研究被大量关注，诸如"多方安全计算"、"零知识证明"的理论大行其道。理论上讲，确实可以做到公布很少的信息就能做到可验证，但其复杂性和计算开销，又是工程层面要去解决的事情了。

信博弈论

区块链中最玄妙的部分是"共识算法"。共识算法的定义是在一个群体中，用一种机制协调大家共同或轮流记账，得出无争议的、唯一性的结果，且保证这个机制可以持续下去。

换句话说，大家一起维护一个账本，选择谁做为记账者？凭什么相信记账者的动作是正确的？怎么防止记账者作恶？如果记账者正确记账如何得到激励？共识机制完整地回答了这些问题。

共识的逻辑是发生在线上的，但实际上，背后是现实世界的竞争博弈。

POW（工作量证明）采用算力去竞争记账者的席位和获得记账者的奖励。现实生活中，为了构建具有竞争力的算力工厂，矿工通常需要研发或购买大量的新型号矿机，运输到有稳定和便宜电力供应地区，消耗大量的电费、网费以及其他运营费用，在被监管时又得举家搬迁，浪迹全球，实际上投入了大量（现实世界的）资金、精力以及背着巨大的风险。如果要在POW竞争中获得稳定可观的收益，投入的资金动辄以亿计，并不亚于办一家企业。

POS和DPOS用权益证明代替了算力消耗，看起来是环保多了。而代表权益的token，除了创始团队自己发行的之外，"矿工们"一般需要通过币币兑换，或者直接法币购买数字币的方式获得，即使是币币兑换，掏出来的币也常常是采用法币购买的，或者至少这些权益都能以法币进行标价，这其实也是现实世界里的财富注入和背书。

然而，和现实的商业关系对比，POW和POS等共识并没有法律和监管机制兜底，也容易受不断变化的博弈形势所影响，比如社群的规模、矿工的更迭、核心技术运营团队的变化。慢慢地，本来有钱和有能力的人，或许会更加有钱和有权力，去中心化的网络可能逐渐变成了卡特尔组织，矿工和技术社区的瓜葛也会不停地掀起波澜，造成分叉、回滚、价格倾轧、对韭当割等现象。

总的来说，人们还是信任区块链上的"自治"，在这种分布式自治里，单个事件（如一笔交易）具有"概率性",同时全网又追求"最终一致性"（公共账本的一致）。这种短期的概率性和长期的确定性，一定程度上可以达成动态的"**纳什均衡**"，支撑起链的生态，给人演化出一种玄妙的"信仰"感。

另一方面，联盟链的记账者一般是机构级的角色。联盟链要求记账者身份可知，参与者们经过许可才能接入网络，他们之间是一种**合作博弈**。联盟链引入了现实世界里的身份信息作为**信用背书**，如工商注册信息、商业声誉、承兑信用、周转资金，或者行业地位、执业牌照、法律身份等，参与者在链上的一切行为均可审计、追查，也让相关的监管部门在必要时可以有的放矢，精准惩戒，强制执行，具有很高的威慑力。

在这种环境里，联盟链的参与者一起协作维护网络，共享必要的信息，在平等透明、安全可信的网络里开展交易，只需要防止少量记账者的恶意操作风险，避免系统上的可用性风险。因引入了现实世界里必要的信任背书，即使联盟链业务逻辑非常复杂，而信任模型却更直观。

所以，所谓的共识机制，背后依旧是现实世界里财力物力的竞争和信用背书，以及相应行之有效的激励和惩戒机制。

天下并没有免费的午餐，也没有平白无故的爱恨。"信"一个记账者，是信他在现实世界里所投入的成本、付出的代价，以及考虑到整个机制有震慑他的惩罚，相信记账者为了持续的收益和增值，不会无故破坏这个网络。

信智能合约

智能合约是由多产的跨领域法律学者尼克·萨博（Nick Szabo）提出来的。他在发表于自己的网站的几篇文章中提到了智能合约的理念，定义如下：

"一个智能合约是一套以数字形式定义的承诺（promises），包括合约参与方可以在上面执行这些承诺的协议"。

简单地说，可以理解为纸质合约的电子版，用代码实现，无差别地运行在区块链网络的每一个节点上，在共识的作用下执行既定的合约规则。

智能合约一般基于一个特制的虚拟机，使用沙盒模式运行，屏蔽掉可能导致不一致性的一些功能。比如获取系统时间这个操作，在不同的机器上，时钟都可能不同，这就可能导致依赖时间的业务逻辑出现问题。再比如随机数，以及外部文件系统、外部网站输入等，这些都可能导致虚拟机执行结果不同，都会被虚拟机沙盒环境隔离。

如果要采用java语言写合约，要么裁减掉jdk里的相关函数（系统时间、随机数、网络、文件等），要么放到一个有严密权限管控和隔离设定的docker里运行。或者干脆设计一门新的语言，如以太坊的Solidity，只实现特定的指令。又或者放弃掉一些"智能"特性，用简单的堆栈指令序列完成关键的验证判断逻辑。

所以，在区块链上执行智能合约，基于沙盒机制控制，凭借区块链的共识算法，达到全网一致、难以篡改、不可否认等特性，运行结果输出就是全网认可的一份合同，江湖人称"Code is Law"。

然而，只要是代码，就一定有出现bug或漏洞的概率，可能来自底层虚拟机和网络漏洞，更多的可能来自逻辑实现。随便搜一下"智能合约 安全 漏洞"，就有一堆搜索结果，包括溢出、重入、权限错误等，甚至就是低级错误。近年来，这些漏洞已经造成各种资产损失，最著名的是DAO项目代码漏洞、Parity的多签钱包漏洞、某互联网公司的代币交易过程溢出归零……

技术文章可以参考：https://paper.seebug.org/601/

目前，行业里对智能合约的安全也是各出奇招，包括安全公司和白帽子审查、形式化证明、众测等，对安全问题会有一定地改善。如果再出问题，要么是黑客太厉害，或者只能抓程序员祭天了：）

所以，信智能合约，是有条件的，是要信经过严格测试、长时间稳定运行、万一出错还有办法补救（而不是绝望的只能等分叉大招）的合约。联盟链里的智能合约一般是经过严格测试的，上线时会执行灰度验证流程，运营中监控运行过程，且根据治理规则设计事后追责、补救（冲正，调账，冻结...）等措施，还是比较可信的。

信中间人（？）

注意本小节标题打了问号，区块链推崇“去中心或多中心，去中介或弱中介”的运作模式，但是由于目前发展尚未完善，很多场景实际上还是引入了中介，如币币兑换通常需要经过交易所，尤其是中心化的交易所。其交易原理是要求用户把资产存入交易所的帐户里，交易时其实是在交易所的数据库里进行记账，只有在存币或提币时，才会和区块链网络发生交互。

交易所的信任模型和区块链某种程度上是脱钩了，这时，交易所本身的资质，运营方的技术能力、安全防护能力、资产信用和承兑能力，才是用户最需要关心的。一旦交易所出了问题，比如跑路、破产、暗盘操作、监守自盗，基本上散户就只能做韭菜了。

多的不说，参见著名的“门头沟事件”:https://baike.baidu.com/item/Mt.Gox/3611884

所以，相信一个托管者，是一个见仁见智的事情，只是现行的模式里，类似交易所这样的角色还在某些区域运作着。2018年，全球虚拟数字资产交易所有1万多家，其中多少能做到高规格的安全，运营规范，干净……那就看情况了。

最后提一点：联盟链默认是没有公链那种虚拟数字资产交易所的。

区块链领域的细节还有很多，以上先罗列主要的几个点，信任技术，信任共识机制，信任规模化的社群博弈，超过了信任“人”。“人”是一种不确定因素，你可以信任一个你很熟悉很老铁的人，也可以信任一大群有共同理念且有完善机制协作的人，但你不能信任某一小撮居心叵的人，要不分分钟变成韭菜：）

总结一下，在区块链世界中，人们可以建立以下基本的信心：

- 我持有的资产和信息，只有我能动用或披露
- 我可以按公允的规则参与交易，分享信息，转入转出资产
- 别人给我转过来的资产一定是有效的，不会被重复花费而失效
- 一旦交易完成，就是板上钉钉的事情
- 一切已经发生的事情都可验证，可追溯
- 违反规则的人会损失更大
- 维护网络的人付出了劳动会有恰当的回报，整个模式可持续

基于这些信心和信任，在合法合规的前提下，人们给网络注入各种资产，开展互补互利、规则透明、公开公平公正的商业行为，将会是一种理想的状态。

22.2.4 区块链世界里不能信什么?

作者：张开翔 | FISCO BCOS 首席架构师

上一篇分享了“信任区块链时究竟在信任什么？”（还没看的童鞋，点击标题可直达），这次换个角度，漫步月之暗面，谈谈在**区块链系统和业务设计**时，不信任什么。

先讲结论： 几乎什么都不能信！建立Don’t Trust,Just Verify的理念，才是通往区块链世界的正确态度。——By我随口说的

不信任其他节点

区块链节点和其他节点会建立P2P通信，共同组成网络，传递区块、交易、共识信令等各种信息。其他节点可能是由不同的机构、不同的人持有，持有节点的人可能是善意，也可能是恶意。即使在善意假设时，节点运行存活的健康度也会受运维水平和资源影响，比如处于一个不稳定的网络里，会偶尔挂掉，会抽风乱发消息，或者硬盘满等原因导致数据存储失败，以及出现其他可能的故障。在恶意假设时，要预设其他节点可能会骗自己或伤害自己，比如传递过来错误的协议包，或者用诡异的指令寻找漏洞进行攻击，或者发起高频垃圾请求，频繁连接然后断开，又或者海量连接占用资源等。

所以节点应该是把自己看成在黑暗丛林里孤身求生存的猎人，必须有“独立自主”、“自给自足”的态度，摆出“不相信其他任何节点”的姿势保护自己。在节点准入时，需要采用证书技术来认证节点身份；在连

接控制上，拒绝有异常的连接；采用频率控制对连接次数、请求量等做限制；在协议包格式和指令正确性等方面做验证。自己发出去的信息，不应暴露自己的私有信息，也不期望其他节点一定会给出立刻和正确的响应，必须采用异步处理和校验容错的设计。

节点和客户端互相不信任

客户端，指在区块链网络外，向区块链发起请求的模块，如业务使用的java sdk、钱包客户端等。客户端和节点通过网络端口通信。如果客户端掌握在不受控的人手里，有可能会向节点发起大量的请求，或发送一堆垃圾信息，使节点疲于应对，甚至巧妙地构建漏洞攻击信息，试图越权访问，窃取信息或使节点出错。

同时，从客户端的角度看，节点有可能不响应或响应缓慢，或者返回错误的数据，包括格式错误、状态错误、表示收妥但其实不处理等，甚至别有用心的人会设置一个“假”节点和客户端通信，欺骗客户端。节点做出这些与期望不符的反应，可能使客户端运行出错，功能受损。

为提升节点和客户端的互信，可以为双方分配数字证书，必须通过证书进行双向握手，客户端经过私钥签名才能对节点发起交易类请求，节点应对客户端进行权限控制，拒绝高危的接口调用，不要轻易开放节点管理接口、系统配置接口等。双方对每次通信的数据格式、数据有效性都进行严密校验。双方在交互时也应该进行频率控制，异步处理，对每一个交互进行结果校验，不能预设对方正确处理，必须获取交易回执和处理结果进行确认。

当认为只和一个节点通信并不能保证安全时，客户端可以采用“f+1查询”的思路，尽可能多地和几个节点通信。如果当前链的共识安全模型是“3f+1”，那么，如果从f+1个节点读到的信息是一致的，结果是可以确认的。

不信任区块高度

区块高度是一个非常关键的信息，代表整个链当前的状态。向区块链发送交易、节点间进行共识、对区块和状态的校验等操作都会依赖区块高度。

某个节点在断网或处理速度缓慢时，其区块高度有可能落后于整个链，又或者某个节点恶意伪造数据时，其高度又可能超过整个链。在链出现分叉时，如某一个分叉上的区块高度被另一个分叉超越，落后的分叉就会变得毫无意义。即使在正常的情况下，节点依旧有可能间歇性地落后于整个链一到几个区块，然后在一定时间内才可能追上最新高度。

如在PBFT共识模型里，总数2/3以上节点在同一个高度时，全链就有机会达成共识继续出块。余下的1/3的节点有可能和参与共识的节点高度不同，这时意味着从这个节点读取到的数据，并不是全网最新的数据，只能代表链在该高度时的一个快照。

业务逻辑可以把区块高度做为一个参考值，基于高度做一些判定逻辑，在确定性共识（如PBFT）的链上，采用f+1查询等方法确认链的最新高度，在可能分叉的链上，需要参考“6个区块确认”的逻辑，审慎选取可信的区块高度。

不信任交易数据

交易（Transaction）代表一方向另一方发起了一个事务请求，交易可能导致资产的转移、改变帐户状态或系统配置，区块链系统通过共识后确认交易，使相关的事务生效。交易必须带上发送者的数字签名，交易里所有数据字段都必须包含在签名里，未经签名的字段存在被伪造的可能，不予采信。交易数据在网络上广播时，可以被其他人读取，如交易数据里包含隐私数据，发送者则必须对数据进行脱敏或加密保护。交易可能因为网络原因被重发，或者被其他人保存下来刻意再次发送，造成交易的“重放”，所以区块链系统必须对交易进行防重，避免出现“双花”。

不信任状态数据

区块链的状态（State）数据是由智能合约运行后生成的，理想情况下，每个节点的合约引擎一致、输入一致、规则一致，那么输出的状态就应该一致。但不同的节点可能安装了不同的软件版本，或者合约引擎的沙盒机制不够严密引入了不确定性因素，甚至被侵入、篡改，或者存在其他莫名其妙的bug，都可能导致合约运行输出结果不一致，那么一致性和事务性就无法得到保障。

状态的校验是成本很高的事情，典型的校验方法是使用MPT（Merkle Patricia Tree）树，把所有状态都塞到树里管理起来。MPT树可以把所有的状态归结为一个Merkleroot Hash，节点之间在共识过程中确认交易运行后生成的状态树Merkleroot，确保状态一致。

这棵树结构复杂，数据量大，消耗不少的计算和存储资源，很容易就成为了性能瓶颈。所以对状态的校验需要有更快、更简单，且又稳妥的方案，如结合版本验证、增量Hash验证等算法，辅以数据缓存，可减少重复计算和优化IO次数，能在保证一致性、正确性的同时，有效地提升验证效率。

不信任私钥持有者

采用私钥对交易以及其他关键操作进行签名，再使用公钥验签，是区块链上最基础的验证逻辑。只要私钥被正确使用，这个逻辑是安全的。

但私钥仅仅是一段数据，只依赖私钥则用户是匿名的。在联盟链面对的场景里，需要使用许可型的身份，首先通过KYC、尽调、权威认证等现实世界的验证方式确认身份，然后将身份和公钥绑定并公示，或者结合PKI体系的数字证书发放公私钥，这样私钥对应的身份是可知、可信、可控的。

私钥可能会因丢失、泄漏而被他人盗用，或者因被遗忘导致资产损失。所以在私钥的保存上，需要考虑采用周全的保护方案，如加密存储、TEE环境、密码卡、USBkey、软硬加密机等方案。在私钥的管理上，则需要考虑密钥丢失后如何安全的重置、找回。

加强版的私钥使用思路有几个，比如使用多签、门限签名等方式，每次交易时必须用多个私钥进行签名，私钥可以保管在不同的地方，安全性高，但技术方案和使用体验复杂。

还有一种是交易私钥和管理私钥分离。交易私钥用于管理资产，管理私钥用于管理个人资料，交易私钥可以被管理私钥重置，管理私钥本身则通过门限、分片等算法，分开存储保管，以备重置或找回。

不信任其他链

在跨链的场景里，每条链有自己的资产、共识，链之间的安全模型变得非常复杂，比如一条链上的记账者串通造假，或者链出现了分叉、区块高度回滚，这时如果链外的其他模块和链有不够严谨的交互，都会造成数据不一致或资产损失。如果不同的链采用的还是不一样的平台架构，那么在工程上会更加复杂。

跨链、侧链目前依旧是业界在研究和逐步实现的课题，主要目的是解决链和链之间的通信，进行资产锁定和资产交换，保证整个过程的全局一致性、交易事务性，以及抗欺诈。从A链往B链转移一个资产，必须要确保A链上的资产被锁定或销毁，且B链上一定能增加对应的一笔资产，在双方可能分别出现分叉、回滚的时间窗里，要有机制确保双向的资产安全。

在现有跨链的方案里，存在中继、链间HUB等方式，这些系统的设计本身也要达到高度可信可靠的标准，安全等级应不低于甚至高于所对接的链，同样也应采用多中心、群体共识的体系设计，整体复杂度可算是链的N次方了。

不信任网络层

区块链节点需要和其他节点发生通信，所以必须在网络上暴露自己的通信端口，如果通过公网通信，那么相当于在公网上暴露了自己，很容易遭到类似渗透、DDOS这样的网络攻击。节点必须在网络层保护自己，包括在网关上设置IP黑白名单、设置端口策略、进行DDOS流量防护，且对网络流量、网络状态进行监测，如果突发网络流量或连接数暴增，说不定，就是被人当肉鸡或者正在脱库进行时了。非必要端口，切忌对公网开放，如用于做管理监控的RPC端口，只能对机构内部开放，在进行网络策略设定之前，一定要慎之又慎。

不信任代码

“Code is law”确实是一句响亮的口号，但是在程序员头发掉光之前，他写的代码都可能有bug，只是看写bug快还是修bug快而已。

无论是底层的代码还是智能合约代码，都可能存在技术性或逻辑性的坑，但凡代码产生的数据和指令行为，都需要另一段代码对其进行严格地校验，代码本身也需要进行静态和动态扫描，包括采用形式化证

明等技术进行全面地审核验证，以检测可能的逻辑错误、安全漏洞或是否有信息泄露。前段时间有一份公布到github上的某酒店系统的代码，居然包括了mysql的连接用户名密码，且数据库端口居然是向公网开放的，这种坑简直不可想象。

开放出去的开源代码，固然可以被人审查、反馈以提升安全性，也可能被人翻找漏洞、随意修改，甚至恶意埋雷。但总的来说，开源还是利大于弊。在开源社区中，开发者会向项目提交PR（Pull Request）。审核PR是很关键也很繁重的工作，值得安排专家并分配大量时间去做审核。有开源项目的老司机透露，其项目核心模块的PR的审核时间长达经年，否则"加了个功能引入两个bug"那真是得不偿失，更别说如果被植入漏洞埋雷了。

不信任记账者

共识的流程大致可以抽象为，选出记账者，记账者发布区块，其他节点校验和确认。公链里记账可以用"挖矿"的方式进行（如比特币），矿工用大量的算力代价为它自己的诚信背书，又或者是用大量的资产权益抵押获得记账权（Pos和DPos等共识）。在联盟链常用的PBFT/Raft等算法里，记账者列表可以是随机或轮换产生，记账者给出提案，其他投票人多步提交，收集投票。按少数服从多数的原则，一般是2/3以上共识节点同意，共识才能达成。

从系统可用性角度看，记账者有可能出错、崩溃，或者运行缓慢，影响整个链的出块。又或者记账者可以只收录手续费高的交易，抛弃一些交易，导致有些交易总是不能达成。有的记账者还可以凭借算力或暗箱运作，进行"预挖"或者"扣块攻击"，破坏博弈关系……

记账者故障或作恶，超越了共识的安全阈值的话，将直接伤害整条链的价值基础。根据不同的记账模式，记账者需要设计不同的容错、校验、抗欺诈算法，执行激励和惩罚机制，在运行过程中定期检查记账者的健康度，对于无力记账或者作恶的记账节点，全网不接受他们的记账结果，并对其进行惩戒，甚至是踢出网络。

罗列起来还有很多，包括合约、证书、同步等等，每一个模块都有自己的功用和风险点，简直罄竹难书。总之，区块链做为分布式的多方协作的体系，接入了形形色色参与者，整个体系绝不是单个开发者或运营者所能单点把控，"善意推测"在这个领域已经不尽适用，整个世界步步惊心，处处冷箭，只能通过周密的算法和繁杂的流程维系共识和安全，简而言之，没有经过验证的信息，一个字节都不能相信。

比起单一环境里的软件设计，区块链领域的设计思路确实存在颠覆性，开发者要从"做功能，只容错，不防骗"的思维模式里跳出来，带着"怀疑一切"的态度进行设计。

开发者在面向区块链领域时，不能只是思考怎么实现一个功能，而更要去思考整个流程会不会有出错，会不会被人篡改数据、发掘漏洞、攻击系统、欺诈其他参与者。要换位思考自己所实现的功能，会被别人用什么方式使用，在不同的环境会有什么表现，可能造成什么后果。任何收到的信息，任何流程输入、输出，都必须经过严格地校验才能采信，开发者能做到这一点，才算是打开了区块链新世界的大门，才能在连续剧里至少活到第二集。

分布式算法、对称非对称加密、HASH、证书、安全和隐私等技术在区块链领域大行其道，都是为了在保护信息的同时，给信息加上一层又一层的证明和可验证因子，这使得整个系统变得复杂、繁琐，但这是值得的，因为这样才能共同验证，构建"安全"和"信任"。

以上，写给准备跳坑，或已经在坑里的程序员。共勉。

22.2.5 区块链的速度困境："贵"在信任，"慢"得其所

作者：张开翔 | FISCO BCOS 首席架构师

举个例子

数钱，比如数一个亿（是不是好刺激~）

1、如果一个人数，慢，但好在专注，全力以赴，在可见的时间内可以数完。这叫单线程密集计算。

2、如果N个人一起数，每人平分，分头同时数，最后汇总总数，所用时间基本上是第一种情况的1/N，参与的人越多，所需时间就越少，TPS就越高。这叫并行计算和MapReduce。

3、如果N个人一起数，但由于这N个人互相不信任，得彼此盯着，首先抽签选一个人，这个人捡出一叠钱（比如一万块一叠）数一遍，打上封条，签名盖章，然后给另外几个人一起同时重新数一遍，数好的人都签名盖章，这叠钱才算点好了。然后再抽签换个人检出下一叠来数，如此循环。因为一个人数钱时别人只是盯着，而且一个人数完且打上封条和签名的一叠钱，其他人要重复数一遍再签名确认，那么可想而知，这种方式肯定是最慢的。**这就叫区块链。**

但换个角度，方式1，一个人数有可能会数错，这个人有可能生病或休假，导致没有人干活，更坏的结果是，这个人可能调换假币或者私藏一部分钱，报一个错的总数。

方式2，N个人中会有一定比例数错，也可能其中一个人休假或者怠工，导致最终结果出不来，更可能因为人多手杂，出现部分人偷钱、换假钱、报假数……

方式3，很慢，但是很安全，因为所有人都会盯着全过程进行验算，所以肯定不会数错。如果其中有人掉线，可以换人捡出新的一叠钱继续数，工作不会中断。所有数过的钱上面都有封条和签名，不会被做手脚，万一出错了也可以找到责任人进行追责。这种情况下，资金安全是完全得到保障的，除非所有的参与者都串通一气了。该模式下，参与的人越多，资金安全性就越高。

所以，区块链方案致力追求的是，在缺乏互相信任的分布式网络环境下，实现交易的安全性、公允性，达成数据的高度一致性，防篡改、防作恶、可追溯，付出的代价之一就是性能。

最著名的比特币网络，平均每秒只能处理5~7笔交易，10分钟出1个块，达到交易的最终确定性需要6个块也就是1个小时，且出块过程相当损耗算力（POW挖矿）。号称"全球计算机"的以太坊，每秒能处理的交易数也仅是2位数的量级，十几秒出1个块。以太坊目前也是采用损耗算力的共识机制POW挖矿，会逐步迁移到POS共识机制。这两个网络在粉丝们爆炸性地进行交易时，可能会陷入拥堵状态，大量的交易发出后，一两天甚至更长的时间才会被打包确认。

但在资金安全就是命的场景下，有些事情是"必须"的，所以，即使慢，还是会考虑选择区块链。

区块链为什么慢

分布式系统里有一个著名的理论叫CAP理论：2000年，Eric Brewer教授提出一个猜想：一致性、可用性和分区容错性三者，无法在分布式系统中被同时满足，并且最多只能满足其中两个。

CAP的大致解释

Consistency(一致性)：数据一致更新，所有数据变动都是同步的

Availability(可用性)：好的响应性能

Partition tolerance(分区容错性)： 可靠性

这个理论虽然有一些争议，但从工程实践中看，和光速理论一样，可以无限逼近极致但是难以突破。区块链系统能把一致性和可靠性做到极致，但是"好的响应性能"方面一直有点被人诟病。

我们面向的"联盟链"领域，因为在准入标准，系统架构、参与节点数、共识机制等方面都和公链不同，其性能表现远高于公有链，但是目前几个主流的区块链平台，在常规PC级服务器硬件上实测，TPS一般是在千级的样子，交易延迟一般在1秒到10秒这个级别。（听说TPS十几万级和百万级千万级区块链已经做出来了？好吧，期待）

笔者曾在大型互联网公司工作多年，在海量服务领域，面对C10K问题（concurrent 10000 connection，万级并发）已经有轻车熟路的解决方案，对一般的电商业务或内容浏览服务，普通pc级服务器单机达到几万TPS，且平均延时在500毫秒以内，飞一般的体验已经是常态，毕竟互联网产品卡一下说不定就会导致用户流失。对于快速增长的互联网项目，通过平行扩容、弹性扩容、立体扩容的方式，几乎能无底线、无上限地面对山呼海啸的海量流量。

相比而言，区块链的性能比互联网服务慢，而且难以扩容，根因还是在其"用计算换信任"的设计思路上。

具体哪里慢呢?

从"古典"区块链的系统内部来看

1、为了安全防篡改防泄密可追溯，引入了加密算法来处理交易数据，增加了CPU计算开销，包括HASH、对称加密、椭圆曲线或RSA等算法的非对称加密、数据签名和验签、CA证书校验，甚至是目

前还慢到令人发指的同态加密、零知识证明等。在数据格式上，区块链的数据结构本身包含了各种签名、HASH等交易外的校验性数据，数据打包解包、传输、校验等处理起来较为繁琐。

对比互联网服务，也会有数据加密和协议打包解包的步骤，但是越精简越好，优化到了极致，如无必要，绝不增加累赘的计算负担。

2、为了保证交易事务性，交易是串行进行的，而且是彻底的串行，先对交易排序，然后用单线程执行智能合约，以避免乱序执行导致的事务混乱、数据冲突等。即使在一个服务器有多核的CPU，操作系统支持多线程多进程，以及网络中有多个节点、多台服务器的前提下，所有交易也是有条不紊地、严格地按单线程在每台计算机上单核地进行运算，这个时候多核CPU其他的核可能完全是空闲的。

而互联网服务则是能用多少服务器的多少个核，采用全异步处理、多进程、多线程、协程、缓存、优化IOWAIT等等，一定会把硬件计算能力跑满。

3、为了保证网络的整体可用性，区块链采用了P2P网络架构以及类似Gossip的传输模式，所有的区块和交易数据，都会无差别地向网络广播，接收到的节点继续接力传播，这种模式可以使数据尽可能地传达给网络中的所有人，即使这些人在不同的区域或子网里。代价是传输冗余度高，会占用较多的带宽，且传播的到达时间不确定，可能很快，也可能很慢（中转次数很多）。

对比互联网服务，除非出错重传，否则网络传输一定是最精简的，用有限的带宽来承载海量的数据，且传输路径会争取最优，点对点传输。

4、为了支持智能合约特性，类似以太坊等区块链解决方案，为了实现沙盒特性，保证运行环境的安全和屏蔽不一致性因素，其智能合约引擎要么是解释型的EVM，或者是采用docker封装的计算单元，智能合约核心引擎的启动速度，指令执行速度，都没有达到最高水平，消耗的内存资源也没有达到最优。

而用常规计算机语言如C++、JAVA、go、rust语言直接实现海量互联网服务，在这方面常常没有限制。

5、为了达到可容易校验防篡改的效果，除了第一条提到的，区块数据结构里携带数据较多之外，针对交易输入和输出，会采用类似merkle树、帕特里夏（Patricia）树等复杂的树状结构，通过层层计算得到数据证明，供后续流程快速校验。树的细节这里不展开，可以通过网络上的资料来学习其机制。

基本上，生成和维护这种树的过程是非常非常非常非常繁琐的，既占用CPU的计算量，又占用存储量，使用了树后，整体有效数据承载量（即客户端发起的交易数据和实际存储下来的最终数据对比）急剧下降到百分之几，极端情况下，可能接受了10m的交易数据后，在区块链磁盘上可能实际需要几百兆的数据维护开销），因为存储量的几何级数增加，对IO性能要求也会更高。

互联网服务因为基本不考虑分布式互验互信的问题，很少有使用这种树的证明结构，了不起算下MD5和HASH做为协议校验位。

6、为了达到全网一致性和公信力，在区块链中所有的区块和交易数据，都会通过共识机制框架驱动，在网络上广播出去，由所有的节点运行多步复杂的验算和表决，大多数节点认可的数据，才会落地确认。

在网络上增加新的节点，并不会增加系统容量和提升处理速度，这一点彻底颠覆了“性能不足硬件补”的常规互联网系统思维，其根因是区块链中所有节点都在做重复的验算以及生成自己的数据存储，并不复用其他节点数据，且节点计算能力参差不齐，甚至会使最终确认的速度变慢。

在区块链系统中增加节点，只会增加可容错性和网络的公信力，而不会增强性能表现，使得在同一个链中，平行扩展的可能性基本缺失了。

而互联网服务大多是无状态的，数据可缓存可复用，请求和返回之间的步骤相对简单，容易进行平行扩展，可以快速调度更多的资源参与服务，拥有无限的弹性。

7、因为区块数据结构和共识机制特性，导致交易到了区块链之后，会先排序，然后加入到区块里，以区块为单位，一小批一小批数据的进行共识确认，而不是收到一个交易立刻进行共识确认，比如：每个区块包含1000个交易，每3秒共识确认一次，这个时候交易有可能需要1~3秒的时间才能被确认。

更坏的情况是，交易一直在排队，而没有被打包进区块（因为队列拥堵），导致确认时延更长。这种交易时延一般远大于互联网服务500ms响应的标准。所以区块链其实并不适合直接用于追求快速响应的实时交易场景，行业通常说的“提高交易效率”是把最终清结算的时间都算在内的，比如把T+1长达一两天的对账或清计算时延，缩短到几十秒或几分钟，成为一个“准实时”的体验。

综上所述，区块链系统天生就背着几座大山，包括单机内部计算开销和存储较大，背着串行计算的原罪，网络结构复杂冗余度高，区块打包共识的节奏导致时延较长，而在可扩展性上又难以直接增加硬件来平行扩容，导致scale up和scale out两方面，都存在明显瓶颈。

Scale Out（等同scale horizontally）：横向扩展，向外扩展，如：向原有系统添加一组独立的新机器，用更多的机器来增加服务容量

Scale Up（等同Scale vertically）：纵向扩展，向上扩展，如向原有的机器添加CPU、内存，在机器内部增加处理能力

直面区块链的速度困境，FISCO BCOS的开发者发挥"愚公移山"的精神，努力优化。经过一段时间的努力，已经移山倒海，修出了一条又一条高速通道，使区块链找到了迈向极速时代的路子（详见下篇），这就是我们系列文章要深入解析的内容。

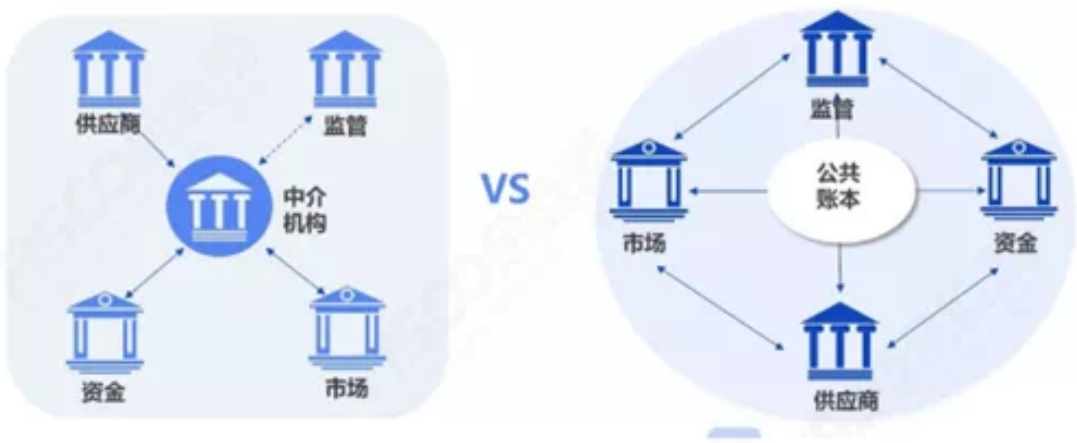
22.2.6 区块链上的交易真的是"点对点"吗?

作者：张开翔 | FISCO BCOS 首席架构师

区块链上的交易是点对点的，不需要中心实体来维护交易。这是很多为区块链摇旗呐喊的文章常用的开篇语，斩钉截铁，毋庸置疑，让人有种"**一入区块链，宛如面对面**"的错觉。

字面上的点对点和面对面

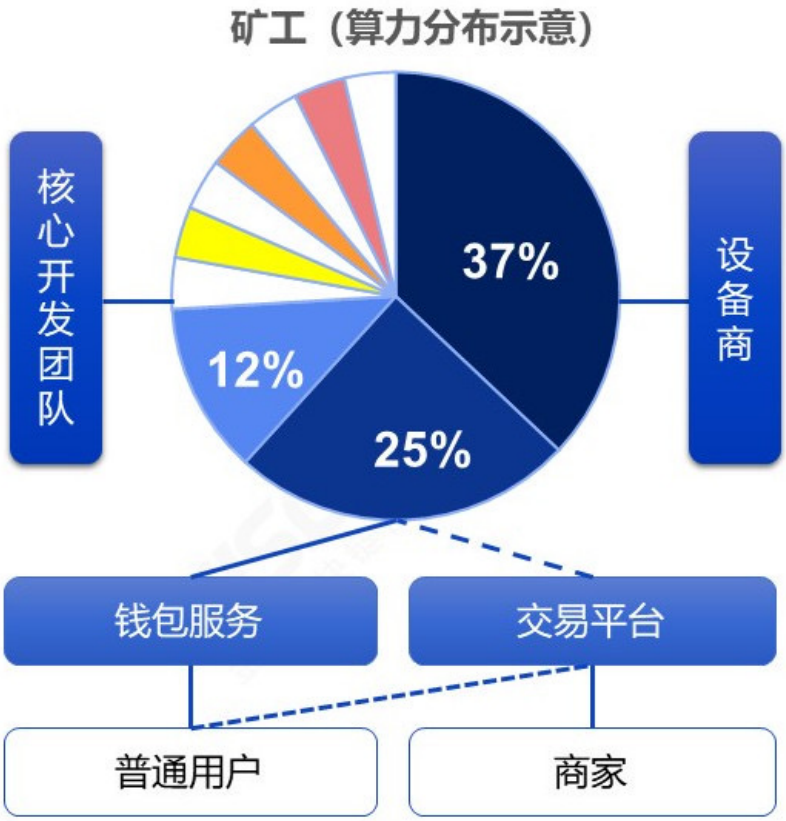
其实这句话有一定的迷惑性，如果精确地将"**中心实体**"定义为类似当下的一些强中介（这里不讨论这种模式的好坏），貌似说得通，毕竟区块链上没有物理上隔断各参与方的中介存在。



星型结构和公共账本互连模式对比

但如果到此为止，很容易给人“区块链上的交易就是‘点对点’了”的误导，而模糊了下述事实：实际上任何区块链系统都需要**共识节点，或者称为“记账者”或“矿工”**来打包和确认交易。即使是plasma、闪电/雷电网络等方案，也只是让参与者在特定通道上，直接或间接地交换账务信息流，最终结算或者纠纷处理，还是得回到主链上进行共识完成流程。

“矿工”这种存在，从理念上可以说是“去中心化”的，因为矿工不止一个，所以“没有中心实体”这句话看起来对，但矿工的“去中心化”事实上是一种**不确定性**，公链上的矿工已经出现了明显的集中趋势，或者类似“**21个长老**”这种被去中心化信徒诟病的卡特尔组织已经萌芽，其去中心化程度和安全性，某种程度上取决于“社区”博弈或某个大神的运作。



围绕着“矿工”的网络生态

而在效率和体验上，“古典”去中心化记账的服务质量很难说是可靠的，一旦出现热点交易状况，那么长达几天的网络堵塞、交易延迟、交易手续费飞涨都是常见的事情，一笔几美金的价值转换，手续费可能

要上百美金。同时，做为普通小散韭菜用户，基本上没有机会去投诉或推动你期望的服务质量提升，连客服渠道都是找不到的，毕竟是“去中心化”嘛。

如果用户是使用区块链钱包或客户端交易，那么更别提“点对点”了，非轻客户端类型的钱包客户端，实际上是把交易发给了服务器，服务器来代理，将交易提交到区块链网络上。这还算是好的，只要钱包服务商不闹妖就问题不大。

如果是交易所的客户端，那很多情况就是在交易所的**业务数据库**里记一笔，连区块链网络都不碰。在缺乏监管、运营水平又参差不齐的“交易所”行业里，这种模式既“中心化”，又面临严峻的道德和安全风险，而“去中心化交易所”则是把中介角色交给了链上智能合约和矿工，理论上更加透明，但在效率、流程方面，依旧说不上是“点对点”。

如果用户干脆自己安装节点，直接加入到网络里，又是什么情况呢?

物理上，因为连接模型的约束（常见的Gossip模型只连接少数“邻居”节点），仅仅从网络层面看，你和交易对手大概率也是不会直连的，而是经过多次转发后，双方才能“碰头”，这是P2P网络的基本特性，并不是“字面上”的“点对点”。如果转发链路上有什么幺蛾子，那也是不确定性的。

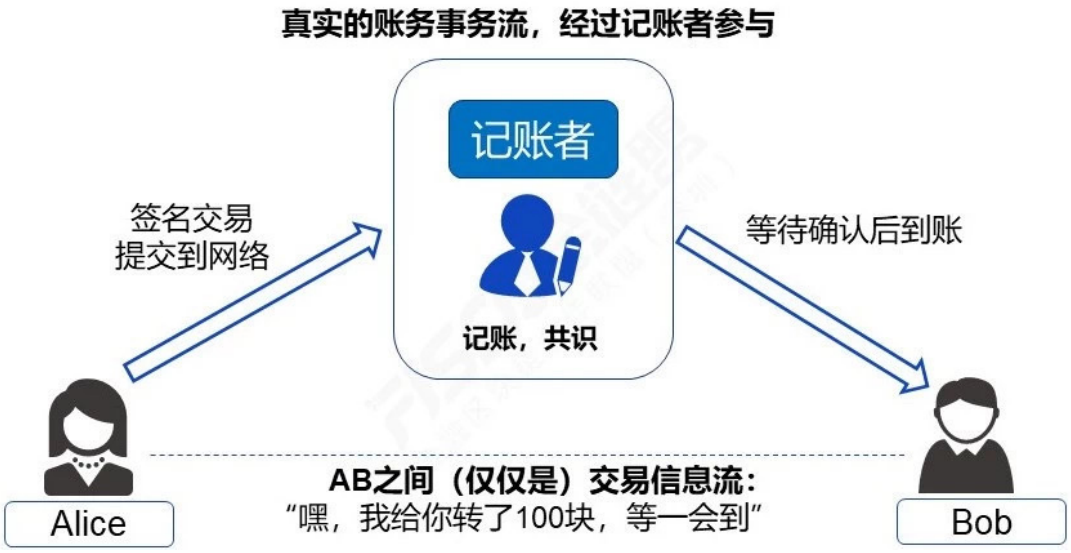
一般来说，反复广播转发的网络模型里，在严谨地选择路由、验证连接邻居的前提下，信息到达率还是可以保障的。这里的“点对点”，事实上是广义的桥接型多跳连接（雷电/闪电网络也类似），这种模型不可避免的有一定的传输冗余和延迟率。事实上，被我们熟知的“P2P下载网络”，如果要达到飞一般的速度体验，是依赖普通节点之外的中心化服务器来加速的。

话说回来，就算物理网络建立了点对点连接，交易事务层面上还是要等待记账者（矿工）收拢交易打包，共识确认和同步区块，一笔交易才算完成了，这个过程中“物理直连”并不涉及经济模型方面的意义。

综上所述，所谓的“点对点”更像一句口号，其真实含义需要穿透表象来理解。对普通用户来说，他们理解的最朴素的“点对点”，是**面对面地把钱塞到对方手里**，一次握手即完成交易。而用户把交易发出去，无论下一步是到银行、支付公司、还是“分布式账本”，对普通用户来说，从本质上讲都是一种**委托交易**。

一手交钱一手交货

字面上的“点对点”交易



事实上的分布式账本

在这种格局下，“**面对面**”是不存在的，私密一定程度上也是不存在的。信息不给到记账者，怎么记账？零知识、环签名等重武器并不是每条链都配备的，追求全面隐私保护的链，计算和架构都会更加复杂，我们会另外分析。

最后，想想当下移动支付各种顺畅体验，银行大额转账秒级短信通知，区块链字面上所谓的“**点对点**”，真是说得美好，而现实骨感呢。

所以，如果读文章时，看到以吹嘘区块链是“点对点”这个slogan为立论基准，建议读者先理清楚他这里的“点对点”究竟说的是什么语境下的“点对点”。我心目中的分布式账本，真正的意义并不在于“点对点”，而是“可靠可信可控”，通过博弈和协作，多方共同实现的价值网络，让每个参与角色在其中都有其存在感。

当区块链系统能达到或接近现有中心化交易系统的速度和交互体验，包括**7\24高可用的多活多方记账、秒级延迟确认、快速达成最终一致性、并发处理能力又足够高、交易成本足够低、对公对私都有明确的QoS约定和兜底容灾容错方案**，而且区块链先天的分布式架构体系上有群体规模效应，可防作恶、防篡改，可监管，那么才算是**“多中心”信任背书、透明协作**的靠谱分布式交易系统。不喊口号、实干为主的联盟链目标，更贴近这个方向。

22.2.7 怎样让你的区块链更加安全可控

作者：张开翔 | FISCO BCOS 首席架构师

随着技术和产业的发展，区块链会被用到实体行业的广泛场景，以及金融业、政务、工业等关键领域。我们更需要重视技术和运作的安全可控，以保护信息安全、财务安全，乃至社会安全。

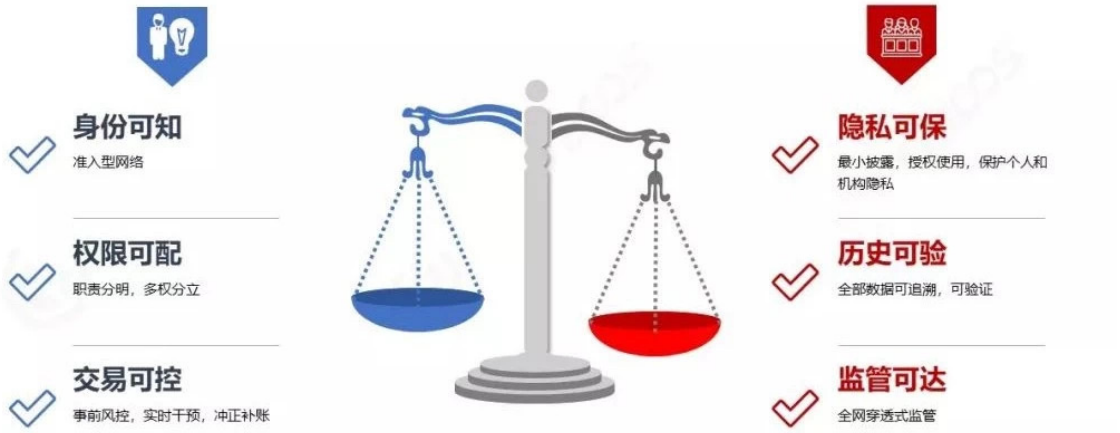
安全稳定的技术路径

对比任意驰骋的“越野车”，联盟链更像安全可靠的“商旅车”。正所谓“道路千万条，交规第一条”，出行首先要选择对的路线，也要遵守交规；其次，安全的重要性毋庸置疑，车子硬件达到高安全标准，驾驶员审慎专业驾驶，才能保证乘客的人身和财务安全；最后，车子本身也得可靠和耐用，不能因故障频发而给旅途制造各种麻烦。在这些基础上，再去追求为乘客创造舒适的环境、友好的体验，并以强劲的性能，高效率地将乘客送到目的地。同理，联盟链的技术路径兼顾稳定和创新，优先保障“合规、安全、稳定”这三个指标，而后再追求易用友好、高性能，这两者也是很重要的加分项。

具体而言，**联盟链合规的第一要务是去代币**，尤其是金融业，绝不应参与虚拟代币的运作。在安全方面，对网络、存储、业务等进行安全加固，保证链上多方共同执行，避免出现木桶的短板。在稳定性方面，严格检测系统质量，遵循“两地多中心”等容灾容错要求，建立可快速反应的监控体系，都是必要的基础工作。

可信可控的几个要点

从系统功能看，区块链是多方参与、重视协作、用算法传递和加固“信任”的系统，针对可信和可控的要求有以下一些要点：



身份可知

拒绝匿名，任何加入这个系统的人或者节点，都需要经过KYC（Know Your Client）环节，依托PKI体系(公开密钥基础建设，Public Key Infrastructure)等技术，向网络认证自己。

权限可配

针对不同的角色，分配不同的系统权限，避免越权访问，或者出现“既当裁判又当运动员”的情况。

交易可控

交易调用的是智能合约，智能合约是用代码写成的，目前“Code is law”依旧是个理想值，交易可控的要点是，事先检查合约逻辑，执行时避免异常和错误，对不符合预期的指令进行干预，在出错后要能冲正、补账、纠正。

隐私可保

个人信息、财务情况、交易明细等都属于个人隐私。机构也有自己的商业隐私，如风控规则、重大交易细节等。在多方共享数据的同时，还需要保护隐私不会泄露。

历史可验

区块链先天是一种“只增不删”的数据库。所有历史交易以及产生的数据都可以记录在区块链上，需要妥善管理链上海量数据，对导出到链外的数据，也要有方法和链上对照，确保数据是完整的、可验证的。

监管可达

许多行业都有对应的监管部门，金融更是强监管行业。区块链系统要做到的是，为监管者提供相关功能接口、访问权限，并提供完整的、可校验的数据以供审计分析。

联盟链角色和治理

联盟链治理模式目前也是正在探讨的热点，总的来说，联盟链是由联盟一起建设、使用和管理的，下面将按这三个维度角色进行梳理归纳。

- 职责分明，各司其职；DO分离，建/管/用/分离；
- 避免“又当裁判又当运动员”

运营委员会可以由联盟里的部分或全部机构一起组成，其本身是“多中心化”的，并不是独家管理。运营委员会共同对联盟链的各种事务进行讨论和决策，包括联盟链的准入审批、规则制定和执行、日常运营等。比如，目前有的联盟链会把“部署合约”的权限收敛到委员会，这样可以保障链上交易规则的有效性和合理性。

监管方则独立于运营委员会之外，可以由主管部门、权威机构、行业协会等角色承担，运营委员会制定的规则，本身应该符合监管要求。

在链上进行交易、参与共识记账、负责清结算等事务的角色，都归纳为链的“使用者”。他们遵循运营委员会制定的规则加入联盟链，用自己的私钥对区块和交易签名，基于链上智能合约和共识机制驱动，高效的完成协作。

开发者负责研究和开发联盟链的底层平台，根据业务需求构建应用系统和编写智能合约，这些内容经过运营委员会审核之后，再由运维团队发布上线。

运维团队还负责配置软硬件环境的参数，关注监控告警，进行版本升级，长期维护系统的稳定性和可持续性。

可见，“建设”和“管理”的角色，都不会直接参与到业务交易流程中，而“使用”角色基于健壮高效的平台进行协作，并接受运营委员会和监管方的管理监督。同时，无论对哪一个角色，都会进行权限控制和操作审计，如果账号丢失、私钥泄露，也能通过链上治理的方式进行找回和重置。这样的治理体系是职责清晰、各司其职、安全可控的，既能保证链上治理的合理性、有效性、安全性，也能保护参与者的权益，使得大家可以高效协作。

贯通全程的安全、可控、合规

从业务流程上看，区块链上的业务参与方众多、流程复杂、资产的生命周期漫长，应该确保端到端的业务可控，否则到了承兑结算时，才发现资产是无价值的，那就难以挽回了。

首先，来自实体世界的人、物、资产，要经过严格鉴定才能上链。 这个环节还是要依靠有公信力的权威机构把关，如中国公民的身份来自公安认证，房产的产权则需要国土局认证，以此类推。还可以依托物联网、基因技术、人工智能分析等技术，对数据进行交叉比对、深度分析，提升鉴别效率和准确度。 资产发行上链后，要检查这笔资产的权责是否明确，定价是否公允，资产的交易规则是否公开透明，这个步骤需要由多方共同完成并达成共识。 对刚上链的资产设置一个“预发布”的状态，链上多方检验后才能使资产“生效”，同时结合链下措施，锁定资产防“双花”，不失为一种好的实践。

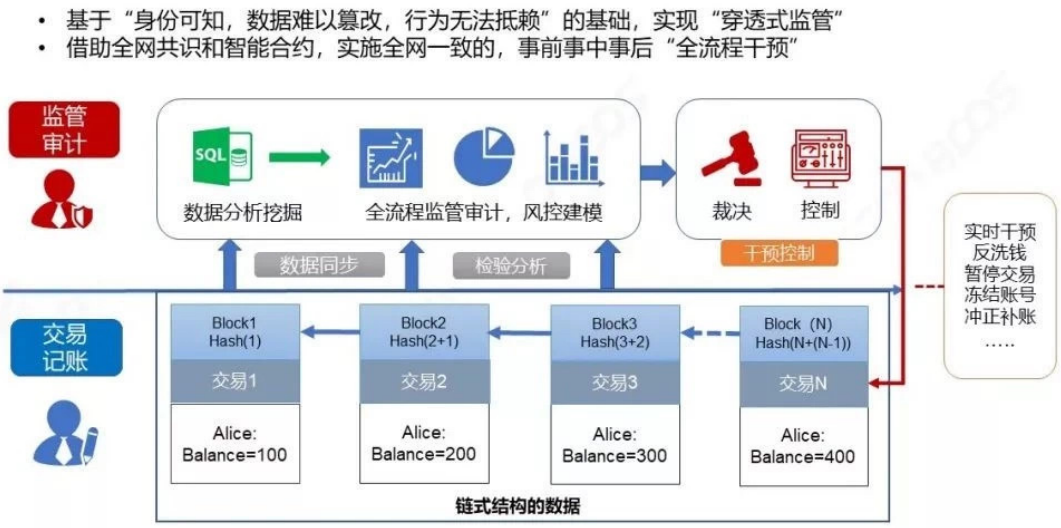
然后，资产可以进入“点对点交易”的高效流转环节，交易过程可以采用多方签名、拜占庭容错等机制，避免少数人作恶。

同时，设置关键检测点，对交易进行实时监测，如发现可能的异常，如高频交易、异常出价、违反交易规则的行为等，则由权限的管理者调用管理接口及时制止和纠正。

当交易正常完成，即进入了清结算和承兑环节，在没有原生代币的联盟链上，需要稳妥的对接链下支付系统，以及物理世界的执行体系等，以保证承兑和执行。 即使前面的环节都顺利完成，因为各种复杂的因素，业务还是有小概率会出现错误或纠纷，那么监管审计和司法仲裁就可以发挥作用了，目前基于区块链实现业务的存证、司法仲裁也是很常见的方式。 可见，端到端地进行设计，并重视链上链下的结合，有助于完整、闭环地管理资产生命周期。

可监管，可干预

关于“区块链业务难以监管”的说法，其实说的是匿名和跨监管主体运作的公链。 部分公链参与者是匿名的，其网络遍布全世界，链上发行虚拟代币且没有稳妥的治理方式，常常只能靠硬分叉治理，使得难以定位链上行为人、难以套用对应的法律法规、也无法针对某一个节点实施监管。



而具备一系列可信可控特性的联盟链，因为身份可知，使得参与者在链上的行为具有确定性，诚实合规的行为可以在业务合作中获得回报，而恶意不轨的行为则无法否认抵赖。区块链特有的链式数据结构具有很强的自校验性，数据难以篡改，可以全程追溯。监管机构可以安装节点接入业务链，或用数据同步的方式获得区块链上的数据，得到完整无错漏的数据集合。

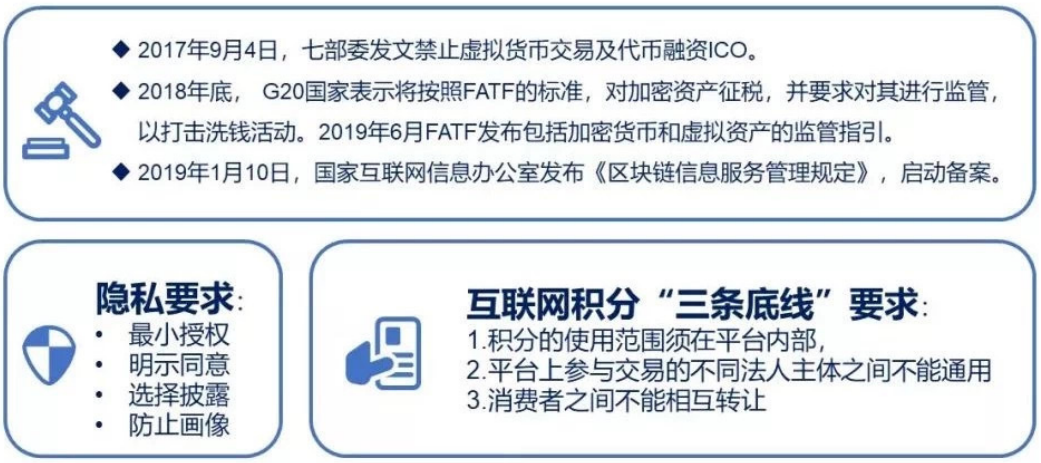
然后，将这些数据纳入大数据挖掘分析，可以实现端到端的、跨机构和交易全流程的监管审计。同时，进行风控建模，预判业务风险，以便识别和裁决恶意行为。

联盟链通过权限配置和业务智能合约的设计，可以实现细粒度的控制功能，如将合约的发布权限收拢到委员会或监管方。合约在运行时置入监管规则检查，发现违规交易立刻终止或回滚，针对用户和节点制定黑名单，采取冻结违规账号、踢掉恶意节点、进行冲正补账等措施。区块链具有全网事务一致性的特点，所有节点基于共识机制协同工作，监管措施可以一点接入，全网生效，这为全网协同的稳健运营、反洗钱等提供了行之有效的可操作性，可以达成“穿透式监管”的效果。

所以，为区块链加入可信可控特性的联盟链，和公链相比，可以呈现截然不同的可监管效果。

技术中立，风控优先

作为区块链和金融从业者，我们在研究技术之余，也应该关注政策的走向。2017年七部委发文禁止“虚拟货币以及代币融资”以来，国际国内都相继出台一系列的规定，对“冒区块链之名行违规之实”的行为明确说不。



“技术中立”和“风控优先”可说是非常经典的总结，这两个词来自工信部给人大代表的答复函，摘录如下：“银保监会督促指导银行机构坚持‘技术中立’和‘风控优先’原则，在厘清金融业务本质特征的基础上，遵循同等的业务规则和风险管理要求，积极推动区块链等金融科技在银行业的应用，采取有效措施防控区块链技术风险。”不同行业都有各自的本质规律和基本守则，如互联网积分，虽然看起来不是“钱”，但其实在业务上积分是有清晰的会计核算规则的，监管部门也对积分的运营提出了“三条底线”要求（如上图），并不会因为用了区块链，运作上就可以突破这三条底线。

再如，有关部门已经发布的“隐私保护标准”，以及欧洲的GDPR等条例，定义了清晰的隐私保护要求，技术运用和管理规范，让隐私保护有法可依。无论采用什么技术，最终都要回到业务本质上。技术运用的初衷，应该是使业务更加健康和可持续发展，并严控和降低其中的风险，而不是带来新的风险，尤其是金融业，有句话说的很好：**“金融不只是经营资金，更多是在经营风险。”**

总结一下，在区块链技术和应用探索的过程中，我们一直遵循几个基本原则：

技术过硬：深耕关键核心技术，持续创新的同时追求安全可控。

广泛开源：开源社区本身有茂盛的生命力，而且还能通过开放源代码，获得社区协助，提升安全性。

善良规范：当我们把技术用于实体领域时，就有可能牵涉国计民生，影响人们身家性命，所以，有着善良的意图和规范合规运作，能使世界因为创新而变得更好一点。

FISCO BCOS开源实现参考：

《FISCO BCOS权限控制一览》

FISCO BCOS安全控制

行业规范参考：

《商业银行数据中心监管指引》

《商业银行信息科技风险管理指引》

《中华人民共和国密码法》

《工信部对十三届全国人大二次会议第1394号建议的答复》

《工信部隐私保护规范》

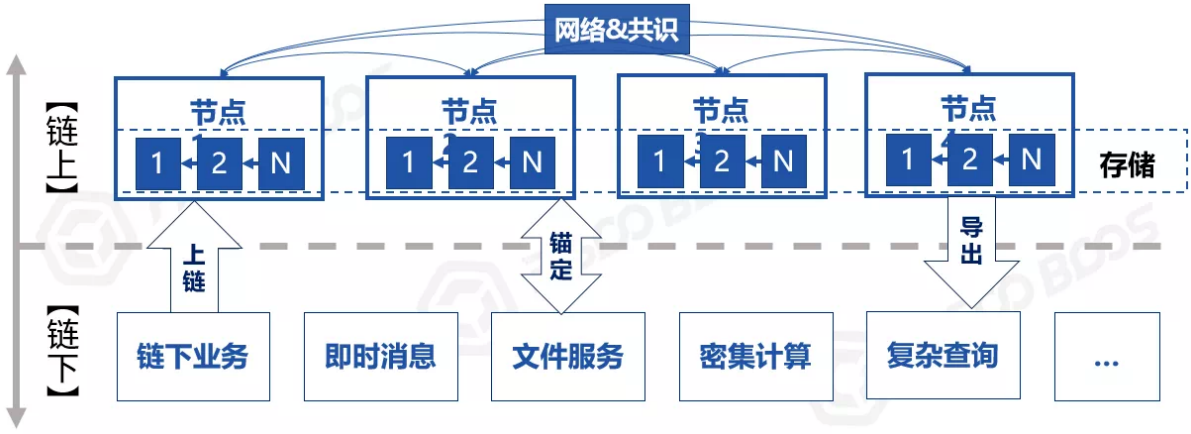
《WeDPR隐私保护白皮书》

22.2.8 一文说清“链上”和“链下”

作者：张开翔 | FISCO BCOS 首席架构师

什么是“上链”？什么数据和逻辑应该“上链”？文件能不能上链？链上能不能批量查数据？“链下”又是什么？“链上”、“链下”诸多问题，一文说清。

什么是“链上”和“链下”



区块“链”的链，包含“**数据链**”和“**节点链**”。数据链指用链式结构组织区块数据，构成数据校验和追溯的链条；“节点链”指多个节点通过网络连接在一起，互相共享信息，其中的共识节点则联合执行共识算法，产生并确认区块。

交易“上链”的简要过程如下：

1. 记账者们收录交易，按链式数据结构打包成“区块”。
2. 共识算法驱动大家验证新区块里的交易，确保计算出一致的结果。
3. 数据被广播到所有节点，稳妥存储下来，每个节点都会存储一个完整的数据副本。

交易一旦“上链”，则意味着得到完整执行，达成了“分布式事务性”。简单地说，就像一段话经过集体核准后在公告板上公示于众，一字不错不少，永久可见且无法涂改。“上链”意味着“**共识**”和“**存储**”，两者缺一不可。交易不经过共识，则不能保证一致性和正确性，无法被链上所有参与者接受；共识后的数据不被多方存储，意味着数据有可能丢失或被单方篡改，更谈不上冗余可用。除此之外，如果仅仅是调用接口查询一下，没有改变任何链上数据，也不需要进行共识确认，则不算“上链”。

或者，某个业务服务本身和区块链并不直接相关，或其业务流程无需参与共识，所生成的数据也不写入节点存储，那么这个业务服务称为“**链下服务**”，无论它是否和区块链节点共同部署在一台服务器，甚至和节点进程编译在一起。当这个业务服务调用区块链的接口发送交易，且交易完成“共识”和“存储”后，才称为“上链”；如果这个交易没有按预期被打包处理，那么可以叫“**上链失败**”。

事实上，几乎所有的区块链系统，尤其是和实体经济、现实世界结合的区块链应用，都需要链上链下协同，用“混合架构“来实现，系统本身就包含丰富的技术生态。

*注1：交易（transaction）是区块链里的通用术语，泛指发往区块链，会改动链上数据和状态的一段指令和数据

*注2：本节描述的是简要的模型，在多层链、分片模型里，流程会更加复杂，事务划分更细，但“共识”和“存储”才叫上链的基本原则不变

交易之轻和“上链”之重

目前区块链底层平台逐步趋于成熟，性能和成本已经不是什么大问题，只是以下几个开销是因“分布式多方协作”而先天存在的：

- **共识开销**：主流共识算法里，PoW（工作量证明，也就是挖矿）消耗电力；PoS（权益证明）要抵押资产获得记账权；PBFT（联盟链常用的拜占庭容错算法）记账者要完成多次往返投票，流程步骤繁杂。
- **计算开销**：除了加解密、协议解析等计算之外，在支持智能合约的区块链上，为了验证合约的执行结果，所有节点都会无差别地执行合约代码，牵一发而动全身。
- **网络开销**：与节点数呈指数级比例，节点越多，网络传播次数越多，带宽和流量开销越大，如果数据包过大，就更雪上加霜。

- **存储开销**：和节点数成正比，所有的链上数据，都会写入所有节点的硬盘，在一个有100个节点的链上，就变成了100份副本，如果有1000个节点，那就是1000份。

也许有人会说："这就是'信任'的成本，值得的！"我同意。只是理想无法脱离现实，毕竟硬件资源总是有限的。 想象一下，如果每个交易都是一个复杂科学计算任务，那么每个节点CPU和内存会跑满；如果每个交易都包含一个大大的图片或视频，那么全网的带宽，以及各节点存储很快被塞爆；如果大家都敞开来滥用"链上"资源，"公地悲剧"就不可避免。 调用API发个交易是很容易的，而链上的开销就像房间里的大象，难以视而不见。作为开发者，需要正视"**交易之轻和链上之重**"，积极"上链"的同时减少不必要的开销，找到平衡之道。

*注1：常规联盟链节点参考配置：8核/16G内存/10m外网带宽/4T硬盘，不考虑"矿机"和其他特种配置。土豪随意，俗话说"钱能解决的问题都不是问题，问题是..."

*注2：本节暂未讨论"局部/分片共识"，也不探讨"平行扩容"的情况，默认假定全网参与共识和存储

让"链上"归链上，"链下"归链下

开销只是成本问题，而本质上，应该让区块链干自己最该干的事情。链上聚焦多方协作，尽快达成共识，营造或传递信任，将好钢用到刀刃上；那些非全局性的、无需多方共识的、数据量大的、计算繁杂的...通通放到链下实现，一个好汉三个帮。

如何进行切割？在业务层面，识别多方协作事务和数据共享中"最大公约数"，抓住要点痛点，四两拨千斤；在技术上，合理设计多层架构，扬长避短、因地制宜地运用多种技术，避免拿着锤子看什么都是钉子、一招打天下的思维。

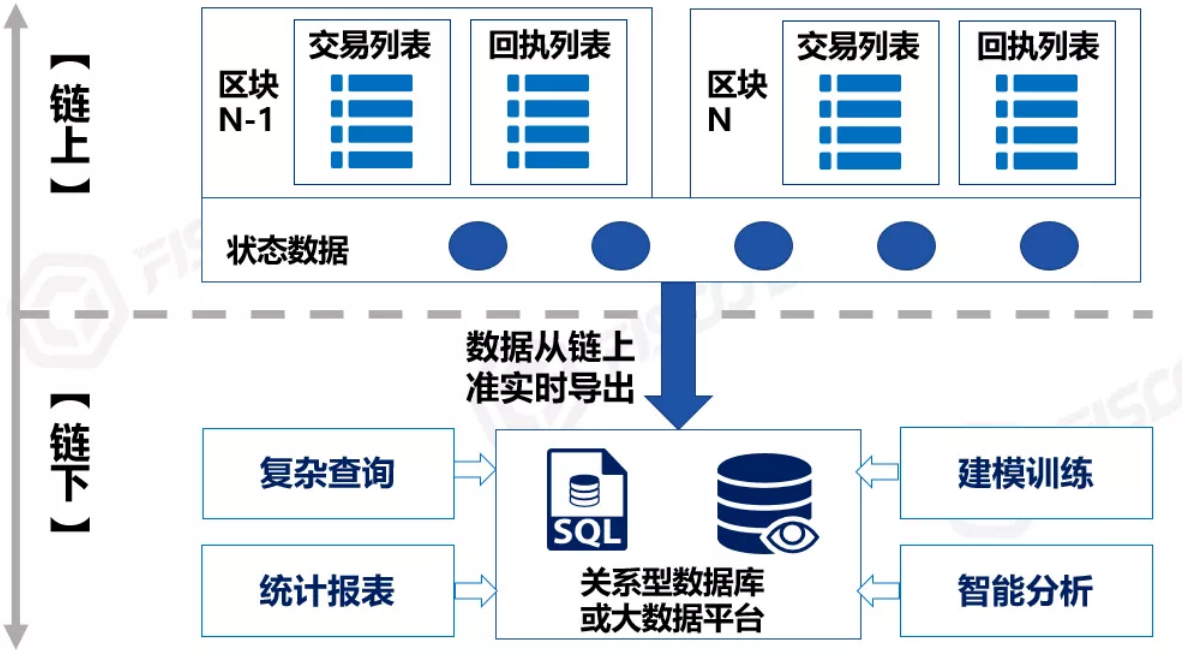
为避免过于抽象，下面给出几个例子。

*注：每个例子其实都有大量的细节，考虑篇幅，这里做概要介绍，聚焦链上链下的区别和有机结合

文件能不能上链?

这是个非常高频的问题，经常被问到。这里的文件一般指图像、视频、PDF等，也可以泛指大体量的数据集，上链可信分享的目的，是使接受者可以验证文件的完整性、正确性。 常见的场景里，文件共享一般是局部的、点对点的，而不是广播给所有人，让区块链无差别地保存海量数据，会不堪重负。所以，合理的做法是计算文件的数字指纹（MD5或HASH），并与其他一些可选信息一起上链，如作者、持有人签名、访问地址等，单个上链信息并不多。 文件本身则保存在私有的文件服务器、云文件存储、或者IPFS系统里，这些专业方案更适合维护海量文件和大尺寸文件，容量更高、成本更低。**注意，如果文件的安全级别到了"一个字节都不能泄露给无关人等"的程度，那么应慎用IPFS这种分布式存储的方案，优选私有存储方式。** 需要分享文件给指定的朋友时，可以走专用传输通道点对点的发送文件，或者授权朋友到指定的URL下载，可以和区块链的P2P网络隔离，不占用区块链带宽。朋友获得文件后，计算文件的MD5、HASH，和链上对应的信息进行比对，验证数字签名，确保收到了正确且完整的文件。 这种方案，文件在链上"确权"、"锚定"和"寻址"，明文在链下传输并与链上互验，无论是成本、效率、还是隐私安全都取得了平衡。

怎么批量查询和分析数据?



对区块链上的数据进行分析是自然的需求，比如“某个账户参与哪些业务流程、完成了多少笔交易、成功率如何”，“某个记账节点在一段时间内参与了多少次区块记账、是否及时、有否作弊”，这些逻辑会牵涉到时间范围、区块高度、交易收发双方、合约地址、事件日志、状态数据等维度。 目前区块链底层平台一般是采用“Key-Value”的存储结构，其优势是读写效率极高，但难以支持复杂查询。 其次，复杂查询逻辑一般是在区块生成后进行，时效性略低，且并不需要进行多方共识，有一定的“离线”性。 最后，数据一旦“上链”，就不会改变，且只增不减，数据本身有明显特征（如区块高度、互相关联的HASH值、数字签名等）可以检验数据的完整性和正确性，在链上还是链下处理并无区别，任何拥有完整数据的节点都能支持独立的复杂查询。

于是，我们可以将数据完整地从链上导出，包括从创世块开始到最新的所有区块、所有交易流水和回执、所有交易产生的事件、状态数据等，通通写入链外的关系型数据库（如MySQL）或大数据平台，构建链上数据的“镜像”，然后可以采用这些引擎强大的索引模型、关联分析、建模训练、并行任务能力，灵活全面地对数据进行查询分析。 区块链浏览器、运营管理平台、监控平台、监管审计等系统，都会采用这种策略，链上出块，链下及时ETL入库，进行本地化地分析处理后，如需要和链上进行交互，再通过接口发送交易上链即可。

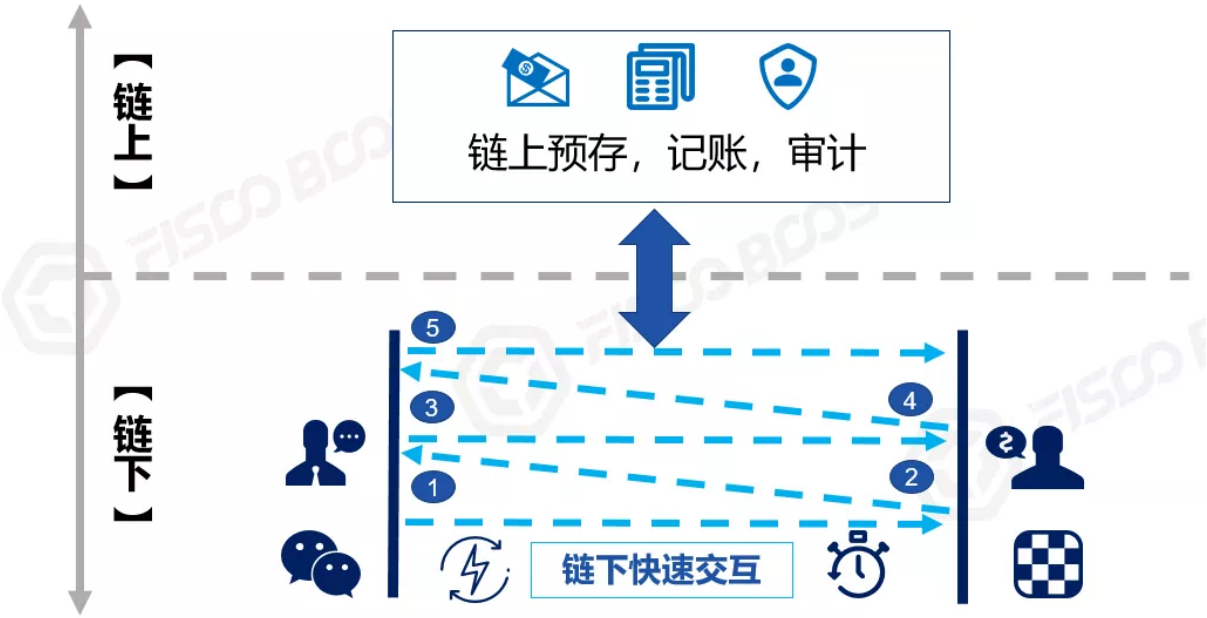
复杂逻辑和计算

和复杂查询略有不同，复杂逻辑指交易流程中关系复杂、流程繁杂的部分。 如上所述，链上的智能合约会在所有节点上运行，如果智能合约写得过于复杂，或者包含其实不需要全网共识的多余逻辑，全网就会承担不必要的开销。极端的例子是，合约里写了个超级大的数据遍历逻辑（甚至是死循环），那么全网所有节点都会陷入这个遍历中，吭哧吭哧跑半天，甚至被拖死。

除了用类似GAS机制来控制逻辑的长度外，在允许的GAS范围内，我们推荐智能合约的设计尽量精简，单个合约接口里包含的代码在百行以上就算是比较复杂的了，可以考虑是否将一部分拆解出去。 拆解的边界因不同业务而异，颇为考验对业务的熟悉程度。开发者要对业务进行庖丁解牛式地分层分模块解耦，仅将业务流程中牵涉多方协作、需要共识、共享和公示的部分放到链上，使得合约只包含“必须”“铁定”要在链上运行的逻辑，合约逻辑“小而美”。

一般来说，多方见证的线上协同、公共账本管理、一定要分享给全体的关键数据（或数据的HASH）都是可以放到链上的，但相关的一些前置或后续的检验、核算、对账等逻辑可以适当拆解到链下。 一些和密集计算有关的逻辑，宜尽量将其在链下实现，如复杂的加解密算法，可以设计成链下生成证明链上快速验证的逻辑；如果业务流程中牵涉对各种数据的遍历、排序和统计，则在链下建立索引，链上仅进行Key-Value的精准读写。 其实，现在但凡看到合约里有用到mapping或array，我都会强迫症地想想能不能把这部分放链下服务去，个人比较欣赏“**胖链下**”和“**瘦链上**”的设计取向。 强调一下，精简链上合约逻辑，并不全是因为合约引擎的效率问题，合约引擎已经越来越快了。核心原因还是在发挥区块链最大功效的同时，避免“公地悲剧”。开发者拿出计算和存储成本最小的合约，有着“**如无必要勿增实体**”的奥卡姆剃刀式美感，更是对链上所有参与者表达尊重和负责任的态度。

即时消息：快速协商和响应



受队列调度、共识算法、网络广播等因素约束，“上链”的过程多少都会有一点延时。采用工作量证明共识的链，时延在十几秒到10分钟，采用DPOS、PBFT的共识，时延可缩短到秒级，此外，如果遇到网络波动、交易拥挤等特殊情况，时延表现会有抖动。 总的来说，对照毫秒或百毫秒级响应的瞬时交互，“上链”会显得些许“迟钝”。比如去超市买瓶水，支付后肯定不能站在那里等十几秒到十分钟，链出块确认后才走吧（略尴尬）。

对类似场景，宜结合链上预存和链外支付，在链下的点对点通道实现高频、快速、低延时的交易，链下确保收妥和响应，最后将双方的账户余额、交易凭据汇总到链上，在链上完成妥善记账。著名的“闪电网络”就类似这种模式。

另外，有些商业场景会先进行多轮的订单撮合、竞价拍卖或讨价还价。一般来说，这些操作是发生在局部的交易对手方之间，未必需要全网共识，所以也可以通过链下通道完成，最后将双方的订单（包含双方磋商结果、数字签名等信息）发送到链上，完成交易事务即可。

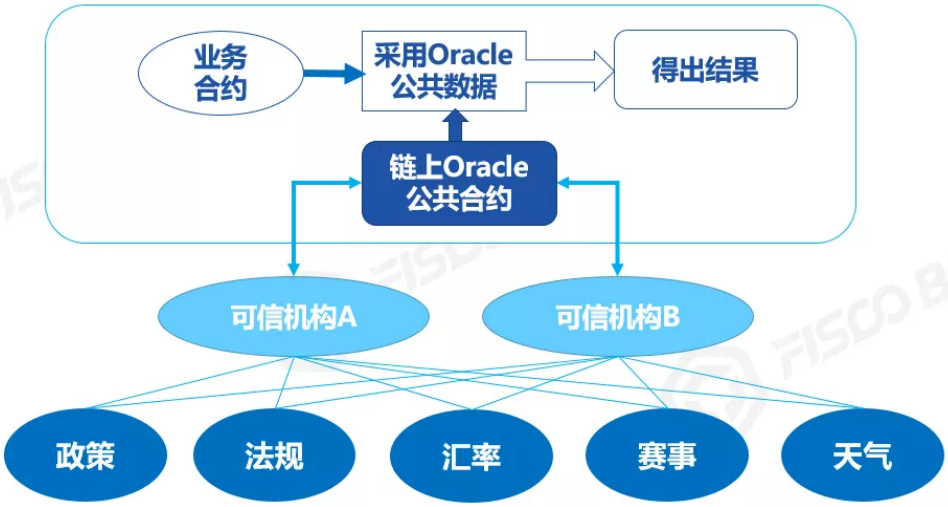
举个下快棋的例子，棋手的每一步棋并不需要实时上链，双方只管啪啪地下，裁判和观众只管围观，在棋局结束时，比如总共下了一百手，那么将这一百手的记录汇总起来，连同输赢结果上链，以便记录战绩分配奖金。如果要复盘棋局详情（如视频），可以参考上文提及的链下文件存储模式，用专用的服务器或分布式存储实现。

针对类似需求，在FISCO BCOS底层平台中，提供了AMOP（链上信使协议），利用已经搭建起来的区块链网络，在全网范围实现点对点、实时、安全的通信。基于AMOP，可以支持即时消息、快速协商、事件通知、交换秘密、构建私有交易等，推荐。

*注：【AMOP】详情可参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/certificate_list.html

链下信息如何可信上链?

先看一个典型问题：“**智能合约运行中要使用链外信息，怎么办？**”比如，链上有个世界杯决赛竞猜游戏，但世界杯不可能在链上踢吧；或者需要参考今天的天气，天气显然不是链上原生信息，应该从气象局获取；在跨境业务中，可能用到法定汇率，而汇率一定是来自权威机构的，不能在链上凭空生成。这时候就要用到“**预言机(Oracle)**”，由一个或多个链下可信机构将球赛、天气、汇率等信息写到链上的公共合约，其他合约统一使用这份经过共识确认的可信信息，不会出现歧义。考虑到安全和效率，预言机（Oracle）会有多种具体做法，实现起来相当有趣。



更进一步的灵魂拷问是：**“**如何保证上链的数据是真实的**？”**坦率地说，区块链并不能从根本上保证链下数据的可信性，只能保证信息一旦上链，就是全网一致且难以篡改的。而区块链跟实体经济结合时，势必要面对“如何可信上链”这个问题。如资产相关应用，除了进行人员管理之外，还要“**四流合一**”，即“信息流、商流、物流、资金流”互相匹配和交叉印证，会使业务流程更加可信。这些“流”常常发生在链下现实世界，要把控它们，可能会用到物联网（传感器、摄像头等）、人工智能（模式识别、联邦学习等）、大数据分析、可信机构背书等多种技术和方式，这已经远远超出了区块链的范围。

所以，本节的命题其实是：**区块链如何和数字世界里的技术广泛结合，更好地发挥自身多方协作、营造信任的作用**。随着数字世界的发展、尤其“**新基建**”的强力推动，我们相信广泛的数字化能在保护隐私的前提下，降低信息采集和校验的成本，采集的数据会越来越丰富。如在使用、转移、回收实体物资时，及时采集监测，甚至是多方、多路、多维度立体化的采集监控，并上链进行共识、公示、锚定，链上链下交叉验证，这样就可以逐渐逼近“**物理世界可信上链**”的效果，逻辑会更严密，更具有公信力，数据和价值流通会更可靠，协作的摩擦更低。

“链上”还是“链下”治理?

“治理”即制定行业联盟和业务运作规则，确保规则的执行，处理异常事件，奖励和惩戒参与者等。以理想化的标准，似乎应该实现链上治理，通过代码决策、制定和执行规则，出错时系统具有“自修复”的“超能力”。实际上，完备的链上治理过于复杂，实现起来很有挑战性，尤其在需要达成现实世界法律法规的执行力时，纯链上的治理往往力不从心。 再多想一步：如完全依赖代码，万一代码本身有BUG、或者要“改需求”呢？链下的决策者、开发者如何发现和介入？ 所以，“Code is Law”还是个理想化的目标，链下治理不可或缺。

联盟链参与者们组成管理委员会，在现实世界里进行民主集中制的讨论和决策，共同制定规则，采用多签、工作流的方式一起发起治理动作，调用区块链接口上链。 在链上，包括区块链底层平台和智能合约在内，都会内置一系列的决策和控制点，如支持多方投票决策，具备从业务层穿透到底层的准入和权限控制能力，可修改业务和节点的参数，能应对异常情况的重置账户，对错账进行冲正调账等等。 治理动作和结果经过共识确认，在链上全网生效，公开透明，接受广泛监督，彰显其合理性和公正性。必要时还可以引入监管方和司法仲裁。 反过来，联盟链上的数据，具备身份可知、难以篡改、无法否认且可全程追溯等特点，可为链下治理决策提供完备的数据基础，也便于为链下实际执行提供可信的凭据。所以，链上和链下有机结合，有助于设计完备、可控、可持续的治理机制。

如何做到“上” “下”自如

或许有人会说：“这链上链下什么的太复杂了，我就想用区块链！”我认为这个说法很对。说到底，用户就想要一条趁手的“链”。作为开发者，我们要打造灵活的、插件化的系统架构，实现各种能力，什么数据导出、文件存储和传输、密集计算、数据采集和异步上链、治理监管、一键部署……按需取舍后，打包起来开箱即用，实际上提供了“**基于区块链的一系列能力**”。 最终呈现的“链”，除了节点之外，还有区块链浏览器、管理台、监控和审计系统、业务模板、APP/小程序等一系列交互入口，用户只需动动鼠标，点点页面，调调接口，一站式体验到一个完整的区块链应用。用户会觉得：“这就是区块链”，无需再分“链上”和“链下”，浑然一体。

说到这里，推荐一个我认为非常棒的设计：分布式身份标识（DID）。 DID是一套涵盖了分布式身份管理、可信数据交换的规范。权威机构为用户完成KYC，颁发凭据。用户将身份标识的摘要公布到链上，而将自己隐私数据存在链下（这一点非常重要）。 使用时，用户采用“**明确授权**”和“**选择性披露**”的策略，仅需出示少量的信息或加密证明，与链上数据进行对照校验，即可证明用户凭据和数据可信性，达成了“数据多跑路，用户少跑腿”、保护了用户隐私的可喜效果。 这种设计很好地将链上链下结合起来，逻辑闭环自洽，并不因为数据存在链下，就削弱了链上的功效，反而使得链的授信模型更为重要。DID规范定义了语义清晰、层次分明的数据结构，以及通用的交互协议。开源项目WeIdentity完整地实现了DID协议，并提供丰富的周边支撑工具和服务，值得参考。

*注：【WeIdentity】详情可见：https://fintech.webank.com/weidentity

结语

链漫漫其修远兮，吾将“上下”而求索。在未来，“可信的”区块链将越来越多地和人们日常生活、实体经济联动，步入寻常百姓家。作为从业者，保持开放的心态，积极而创新地将区块链与更多技术结合，无论运作于链上还是链下，只要能解决问题、创造价值，就是一条好链。

22.2.9 如何解释“我篡改了区块链”这个问题

作者：张开翔 | FISCO BCOS 首席架构师

区块链数据“**全局一致**”、“**难以篡改**”这两个特性已经广为人知，是区块链营造“信任”的基石。为了达到这两个效果，区块链的共识、同步、校验等技术细节足可大书特书，而本文要从“**我篡改了区块链数据**”讲起。

“我篡改了区块链数据”

FISCO BCOS开源联盟链社区现在相当活跃，每天都会产生大量的讨论，大家也会饶有兴趣地研究和挑战区块链如何做到“难以篡改”。我们注意到，尤其在FISCO BCOS支持MySQL数据库作为数据存储引擎后，隔一阵子就有同学在群里问：“我手动修改了我节点连接的数据库里某个状态数据，**这是不是就是篡改了区块链数据呢**？”

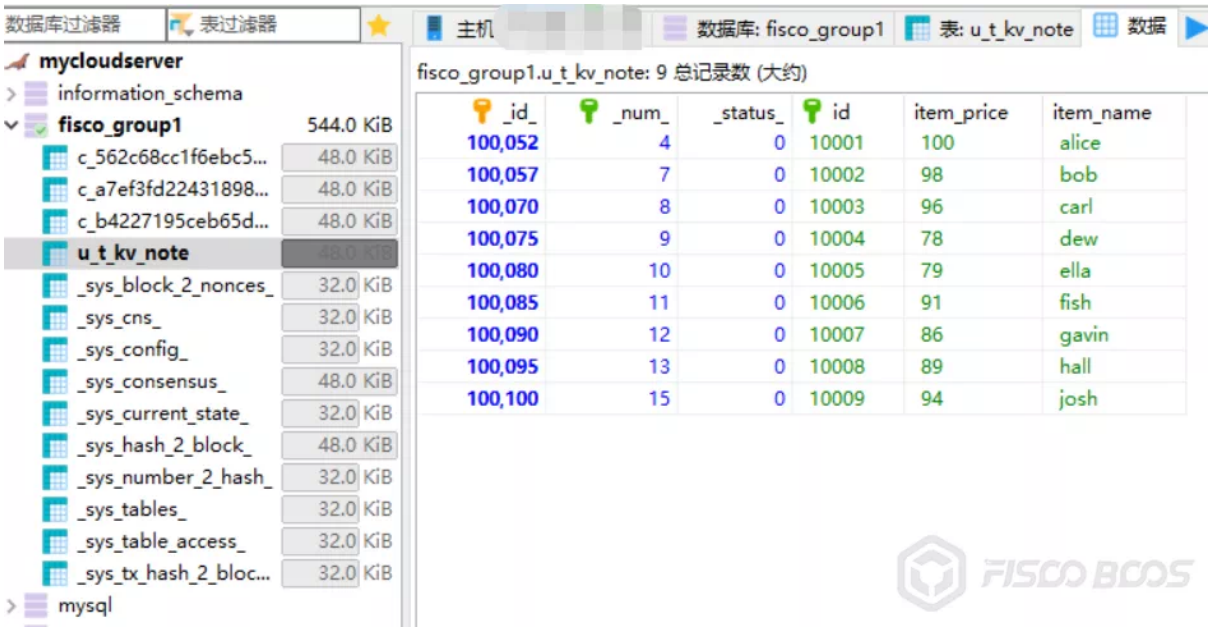
直观地举个例，如链上有个智能合约，管理特定资产余额，在数据库合约表里，经过共识的Alice的余额本来是**100**，这时有人打开MySQL客户端，找到那个合约对应的Table，把Alice的余额update成**10000**。

这时他表示：“你看，我调用合约的查询接口，查出来Alice的余额确实是10000，这就不对了嘛，而且，链还在出块，**根本不防篡改嘛**！”。

初步分析和解答

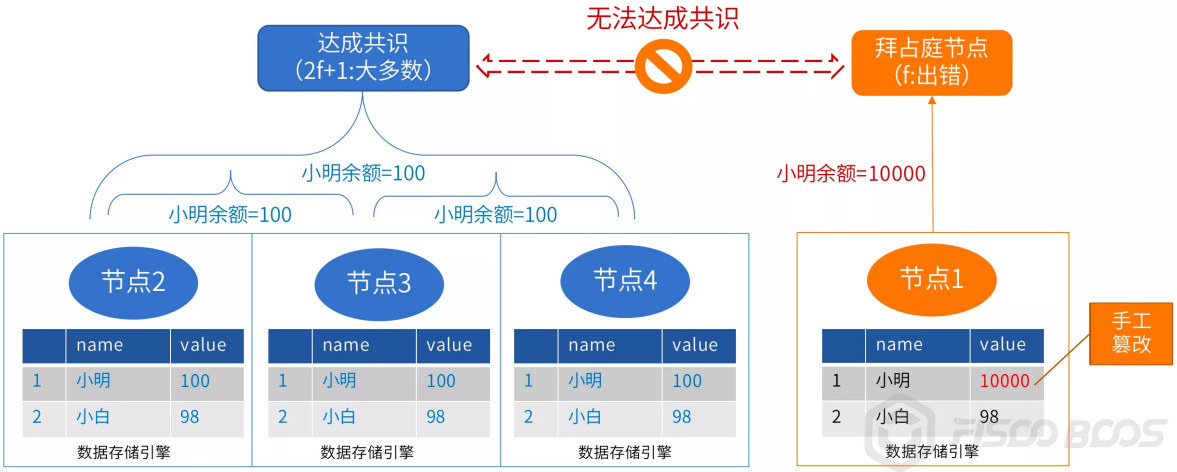
为何这类问题最近多起来了？我们分析了下，猜想主要是由于MySQL数据库用户基础良好，体系比较成熟，给用户提供友好的命令行或图形化交互工具，FISCO BCOS提供了一种Table风格的合约开发模式，表结构设计得清晰直观，对于用户来说，一方面理解和管理起来更容易了，另一方面顺手update甚至delete一下都是小意思。

下图仅为示例数据，采用KVTable合约方式，创建了名为t_kv_node的合约表，系统自动加了“u_”前缀，可见，这个表结构和数据一目了然。



而之前只采用LevelDB或RocksDB作为存储引擎，这两个文件型数据库交互工具比较少，在用户面前的存在感不强，操作相对晦涩，主要通过API编程访问，数据用肉眼难以辨别的Hash key寻址，动手修改数据库的case就少了一些（但也不是不可能）。

所以，热点问题浮出水面，前提是用户可以更方便地修改底层数据了，而不是这个问题之前不存在。这时我们会建议用户试一下，针对Alice的余额，发起一个交易，比如给Alice充值，或者让Alice转账操作，这时，修改过数据的节点将无法参与共识。因为该节点上算出来的Alice的余额和其他节点结果不同，其他节点依旧按100的余额进行计算，而不是**10000**，显然结果是对不齐的。



复习下PBFT的容错模型：定义“f”为可容错节点数，网络中共识节点总数应等于或多于3f+1。即链上有4个共识节点时，可容错的f=1，共识节点总数为7个时，f=2，以此类推。

如果未修改过数据的节点数满足PBFT要求的2f+1的数量，链依旧可以出块。但被修改过的节点，一旦有交易涉及脏数据，就像踩到了雷一样，从此再无法与链共识、同步，相当于被抛弃了。这种节点可以称为“拜占庭节点”，即作恶或出错的节点，具备节点准入控制能力的联盟链甚至会将拜占庭节点隔离出去。

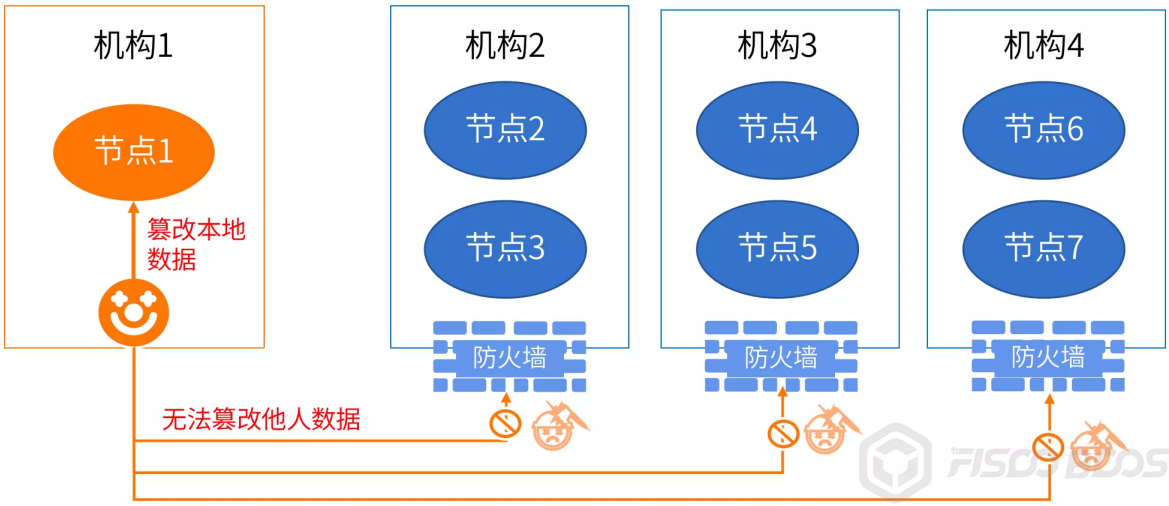
还有一种可能性是，手动修改了数据库里的数据，但节点内存里还刚好缓存了一份副本，并没有被修改，所以通过节点对这个数据的查询、交易还是正常的，甚至会用正确的结果把数据库里被篡改过的数据覆盖掉，不过这是概率性事件，取决于缓存的大小和当时包含的数据项。

*注：对于采用PoW或其他共识机制的链来说，容错模型有所不同，但在容错范围内的少数节点被篡改，也不会影响链的共识。

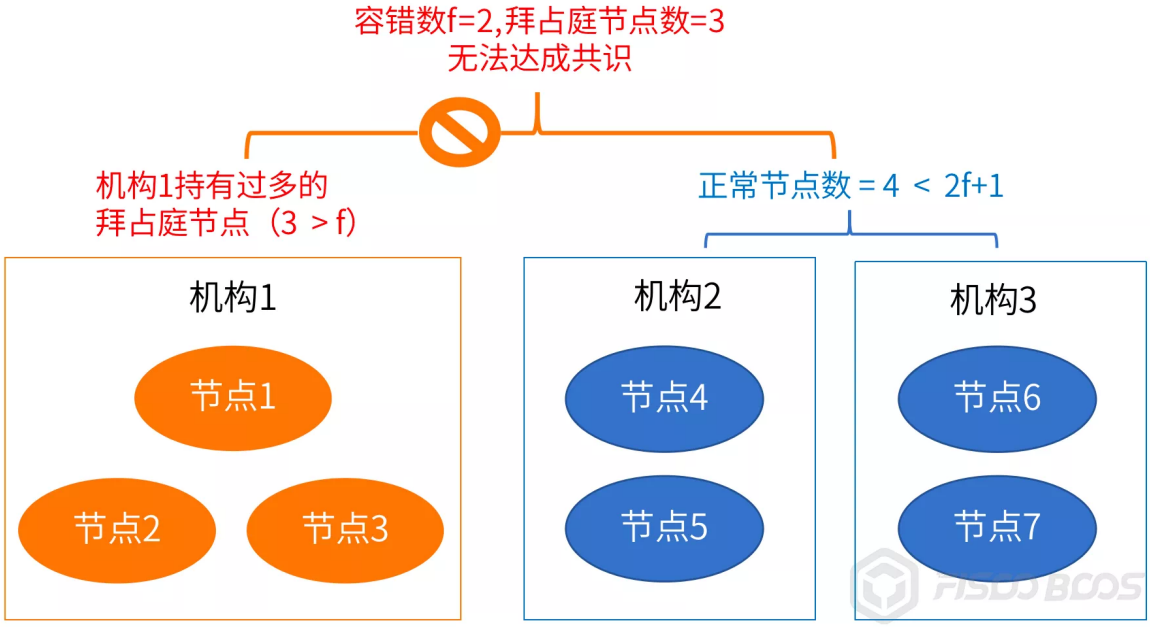
“能否篡改整个联盟链”

有的同学可能会继续刨根问底：“**那我多修改几个节点的数据是不是就篡改了**？”，一般提出这个问题的同学是面向他自己部署的开发测试环境，所有节点都在他手上，所以可以随便改。

在真实的联盟链环境上，节点分别掌握在不同机构手里，要修改，首先得侵入他人的网络、获得服务器和数据库权限、发起修改再全身而退。事实上，在注重安全防护的商业化环境里，这是非常艰难、几乎不可能做到的事情。



从机构粒度来看，单个机构掌握的节点数，应该低于共识算法可容错的数量。比如，链上总共有7个共识节点，那么单个机构掌握的共识节点不应多于2个，这样可以避免机构内部强行修改自己掌握的节点数据，或一个机构的所有节点都意外出错、掉线（比如机房光纤都被挖断了），导致链无法出块。



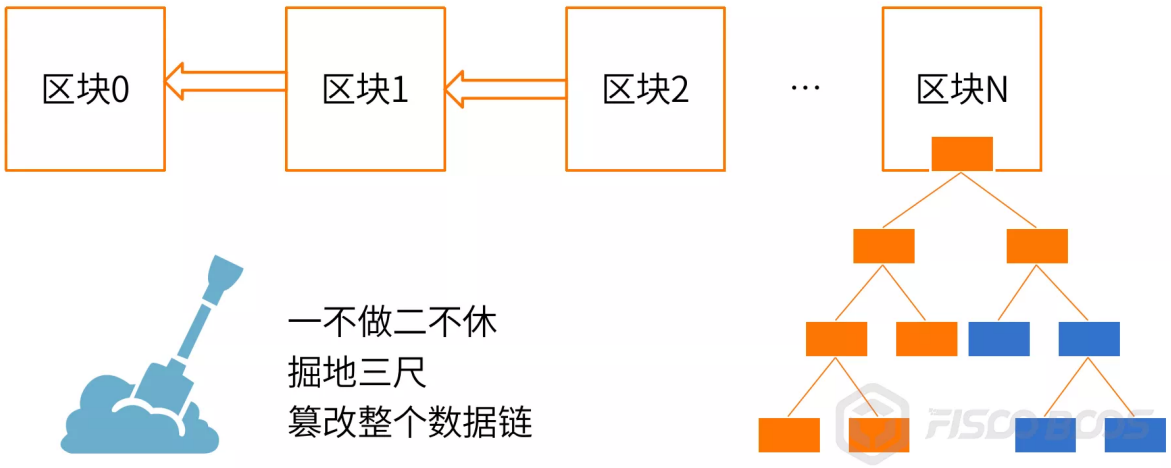
真的没有办法防“本地篡改”吗?

考虑区块链数据本地验证的机制，比如区块之间的Hash关系、状态的Merkle树结构、共识节点的签名等，按数据的互验关系顺藤摸瓜进行检测，似乎有一定概率可以本地检测出数据异常。

但进一步想，对某个数据的查询，区块链的本地校验范围是有限的，一般不会超出单个区块或者一棵merkle树，所以如果篡改者比较熟悉区块链数据的结构和本地校验逻辑，也可以顺着数据校验关系，从状态值开始，把merkle树、区块Hash等关键数据全部改掉。

甚至更彻底地，从创世块开始，把所有的区块、系统配置（对于PoW，可以修改挖矿难度以加速出块）、PBFT的共识者列表等等，都按他的逻辑改一遍，这样这条本地数据链依旧是校验自洽的，只是无法和其他节点共识了。

这种改法，听起来需要不少力气活，但对于一个有决心、有能力的篡改者来说，改改本地数据这个事情其实并不难，难的只是去改别的机构数据而已。



到了这个份上，就相当于一个人铁了心要“骗自己”，那神仙都没有办法了。一旦把本地数据修改的权限交给了不适当的人，最坏情况下，整条链没有一个字节是对的。

但是，本地数据再错，也只会影响自己，影响不了别人，一旦和其他节点进行共识，就会被发现，甚至被惩戒，整个效果会有一点掩耳盗铃的意思。

“为什么区块链不拦住我篡改数据？”

再进一步，那位同学又会问:“**为什么区块链不能立刻发现、并且阻止我篡改数据？也许我只是无意手误呢**”。坦率说，这有点对区块链期望过高了。区块链系统并非无所不能，也不会包办一切，区块链并不会阻止用户对自己的服务器、软件、数据库等施加操作，就像法律不能也不应去阻止你打碎家里的杯子一样。

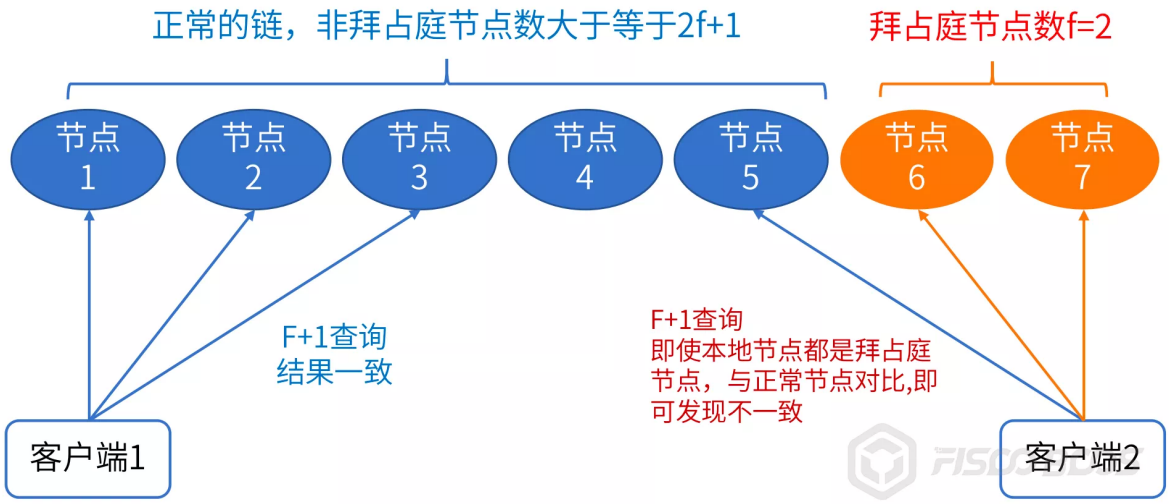
本质上，区块链的一致性、难以篡改性是面向“**全局**”的，是由多方博弈和协作达成的，当链上交易牵涉错误数据时，共识机制可以检测并拒绝已被篡改的数据，保证链上剩余的大多数健康节点继续共识出块。而节点本地不参与共识的数据，共识机制鞭长莫及。

那么，区块链为什么不能主动检测和纠正错误，保证每个节点上的数据一致性？首先，链上的数据非常庞杂，用户直接登入数据库手动修改少量数据，区块链节点并不知道哪一条数据被修改了，无法触发检查。

如果区块链系统定期巡检所有数据，并将所有数据和其他节点进行比对，可想而知，这样做的话，网络、磁盘和计算开销会非常大。关键是，这并不解决问题，因为从数据被篡改后到检测出来的时间窗里，哪怕脏数据只存在了几十毫秒，但这时如果不幸有应用来查询数据，依旧会得到篡改后的结果。对要求苛刻的业务来说，事后检测未必是最佳选项，因为有可能已经造成了业务损失，届时能做的最多就是告警和冲正了。

当然，也可以结合数据库的操作监控、binlog等辅助机制，加速响应速度和检测效率。方法还是有的，如上所述，只是性价比较低，也不彻底解决问题，只有对数据修改极其敏感，且业务上接受延时发现和修订的特定场景，才会考虑将其作为补救措施。我们把这部分归类到运营管理工具里，根据场景需求来实现。

还有一种方法，可以部分解决查询问题：**f+1查询**。即查询数据时，无论是查区块数据，还是合约的状态数据，不妨多查几个节点，查询节点数多于 f 即可。如链上有个7个节点时，f=2，用户查询自己节点之外，继续发出网络请求去查询其他机构的2个节点，共查询3个节点，如果得到的数据都是一致的，则表示数据一定是正确的，反之，一定是这3个节点里出了问题。



但是，要执行f+1查询，前提是其他机构开了查询接口权限，让你连接上去查询。在很多安全防护严密的联盟链上，一般只打开节点之间P2P互联的网络端口，不会轻易给其他机构提供数据查询权限。再则，在网络上发起多次查询，其异步性、时效性、成功率和性能表现都会带来更多变数。

综上所述，对节点本地的数据，就像打地鼠，冒头的（发出交易参与共识，或进行f+1查询），区块链全局共识和容错机制能发现，没有冒头仅蹲在用户硬盘里的，只能用户自己负责了。

结语

区块链通过网络博弈、多方校验实现了全网的容错防作恶，而区块链同步给到各节点的本地数据，需要用户自行妥善管理保存。

从信任传递来看，首先用户得“信自己”，如果连自己都无法相信，说明系统和数据管理有漏洞，莫说是修改数据了，在本地系统的整个链路上，包括区块链软件、SDK、业务服务都有可能出错和篡改作假，这样的环境有何信任可言?

节点持有者必须非常审慎，首先不要手痒或手误去改数据，然后关键是要建立周全的制度，包括管理策略和技术防护，比如，主机访问控制、数据库登录和操作权限控制、操作审计、日志审计等，以避免本机构内有人越权访问监守自盗，或者被外部渗透。

万一数据出错，区块链比中心化系统好一点的就是，还有可能通过与其他节点互相校验检测出来，这时则应该进行告警、查证、补正和追责，以及在有条件和有必要的前提下，善用f+1查询方法，给查询操作加一点点保险。

另外，建议定期备份节点数据到安全的离线设备上，这样无论是出现意外还是人为的数据问题，依旧可以从冷数据里快速恢复，保证一定的RTO（复原时间目标）和RPO（恢复点目标）。

而区块链的健壮性在于，无论单个角色怎么折腾自己的节点和数据，对全局是没有影响的，只有修改者自己受损。任凭窝里翻天覆地，链上依旧云淡风轻，其“**全局一致**”、“**难以篡改**”的定律依旧成立，链仍然是信任的锚点，这就是区块链的魅力。

22.2.10 联盟链要对区块链做减法

作者：梁永甫 | 区块链架构师

当前联盟链平台在公有链的基础上，对共识机制做调整，去掉公有属性，使得准入可控、效率提升，这实际上是对区块链在做减法。但这是否足够呢?

我们在应用中很快发现，联盟链的使用比较受限制，一条链基本上只能做一项业务，而且成员的扩展会面临很多问题，这个扩展并不是在技术上增加一个节点，而是在业务上增加一个参与方。

问题是新增加的成员能看哪些数据，不能看哪些数据，有时候他可能只需要了解其中很少数据就可以了，大部分数据不需让他看到。试问当前的联盟链平台怎么解决? 可能会说，再建一条链，然后做跨链。

跨链也是现在大家重点在做的事情，就是因为看到了这样的问题。但是我们为什么不能从另一个角度思考，区块链本身是否就可以做到，毕竟使用跨链就意味着承认区块链在创建新的信息孤岛。只是以前“孤”在一个地方，现在“孤”在一个圈子， 这其实并没有解决让信息顺畅流通和共享的问题。

中本聪创建比特币的时候想到了区块链这种天才的解决方案，但这并不意味着它是唯一的解决方案。

既然是解决方案，那它要解决的是什么问题？明确了问题，然后再思考下去也许能找到新的解决方法呢。或者对区块链做一些改良让他更适合当下的商业需求。

为什么说要对区块链做减法？因为现在区块链设计得实在是太极致了，对不可篡改、可追溯有极致要求，在其他方面就会有所牺牲，比如隐私保护、性能。所以需要做减法，降低对这两个特性的支持，来提升对其他需求的满足度， 比如去掉历史状态数据，去掉merkle树。

那么，区块链的哪些方面还可以做减法呢？

现在全局唯一链这种形式是否也有改良的可能呢？为什么要全局唯一的链？为什么区块链网络里面不可以有多条链？每条链只跟踪一个事件的过程是否可行？在链产生的过程中增加一些防篡改机制，甚至是不用增加，因为联盟链不用像公有链那样完全自治，完全去中心化。

联盟链是商业链，是在当下商业规则下运行的，如果有些商业规则已经解决的问题，我们是否还要在技术层面去解决呢？如果有政府机构+第三方商业在里面背书，我们还需要花很大精力去解决防篡改的问题吗？

我们讲区块链的时候都是在讲区块链网络，我一直的观点是区块链有两个重要的属性，一个是存储，一个是网络。然而现在大部分的平台这两者之间是没有界限的，区块链网络就是要存储数据，或者每个节点都是全量数据。

那么存储和网络的功能是否可以更明确的区分出来呢？网络做成一个开放的网络，任何机构都可以接入，但网络节点并不存储数据，不感知数据，只转发数据，并通过应用层协议，对数据传输过程进行背书。数据节点才会存储数据，但也并不是存储网络中传输的所有数据，每个数据可以通过网络路由到需要存储的数据节点。

结合上面提到的减掉全局唯一链，是否可以构建一个开放的区块链网络，参与机构可以任意自由的选择数据的传输路径和存储节点？从而形成一个开放性的可信存储网络。只有这样，区块链才能作为一个基础设施存在，就像5G一样

技术对某一方面过于追求，往往会使得其他方面有所缺失。联盟链对区块链做减法的同时，其实也是在其他方面做加法，不让区块链做一个太偏科的学生。

22.3 区块链学习必读

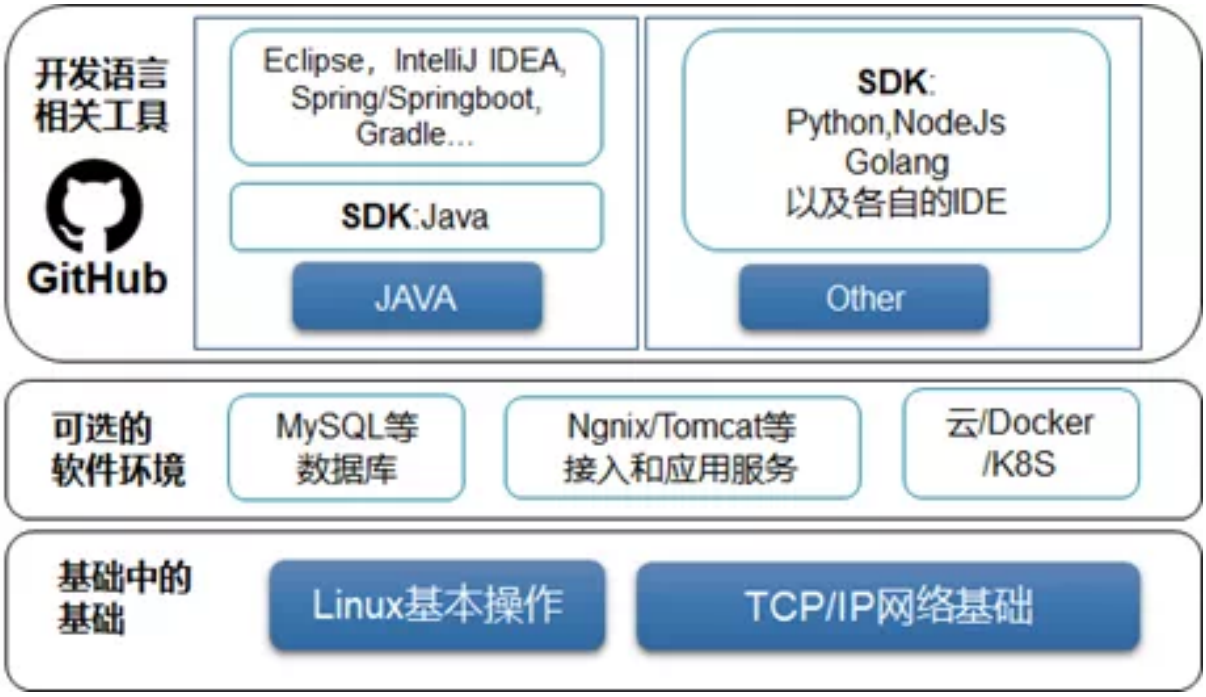
22.3.1 新人必读：区块链实用型技能树

作者：张开翔 | FISCO BCOS 首席架构师

随着新一波的区块链热潮，许多同学怀着巨大的热情进入了这个领域，同时也会遇到不少疑惑，区块链开发需要哪些知识？怎么学习？从哪里学习？遇到问题怎么办？本文将试图给区块链领域新人一个快速实用的指引。

一、基本IT技能

区块链堪称“黑科技”，本身具有大量的技术元素，有志于从技术角度切入区块链的人，应该具备或掌握基本的IT技能，达到至少是常规级别“程序员”或“系统管理员”的技能水平。



首先需要熟练的Linux操作系统知识。

大多数区块链系统是可以跑在Linux上的，包括CentOS和Ubuntu等，你至少要会一些基本的Linux操作指令，比如ls查看目录、ps或top查看进程、find查找文件、netstat查看网络、ulimit检查系统参数限制、df/du查看磁盘空间、用apt/yum安装软件等等，如果这些基本命令都不掌握，在Linux上操作肯定是举步维艰的。

这方面的书和资料都很多，相信一星期就能上手。另外，善于Linux的man指令，可以获得每个命令的详细帮助。如果学会写shell脚本，那更如虎添翼，可以把大量的繁琐操作给自动化了。

要有清晰的网络概念。

区块链本来是分布式系统，节点之间一定是通过网络相连的，只是跑起来的话，不需要多高深的网络知识，只需要了解什么是TCP/IP；公网、内网、本地地址的区别；端口如何配置；节点和节点、SDK和节点之间的互联是否会被防火墙和网络策略挡住；采用ifconfig、telnet、ping、netstat等命令检查网络信息和进行探测、定位网络问题。一般来说，Linux书籍也都会介绍这部分内容。

区块链周边的支持，如浏览器、中间件、业务应用，会依赖一些第三方基础软件，如MySQL/MariaDB数据库、Nginx服务、Tomcat服务等，至少懂得怎么去安装指定版本的软件，掌握修改这些软件的配置文件并使之生效的基本操作，了解各款软件的密码、权限配置和网络安全策略，以保护自身安全。

如果是基于云、docker或者k8s等容器环境构建，需要了解使用的服务商或容器的功能、性能、配置方式，包括对资源的分配：CPU、内存、带宽、存储等，以及安全和权限的配置、网络策略配置、运维方式，达到轻松分发构建的同时，还能保持其稳定性和可用性。

各种云服务商和容器解决方案都有周全的文档和客服服务渠道，可以帮助用户顺畅地使用。

到编程语言阶段，可以根据自己的学习路径，选择不同的语言。

如果是使用Java语言，那就应该熟练掌握Eclipse、IntelliJ IDEA等集成IDE，熟悉Gradle为主的工程管理软件，熟悉Spring、Springboot等java的基础开发组件，熟悉在IDE或命令行下对资源路径如ApplicationContext等路径的定义，或许还有myBatis等流行的组件，这些都可以在java相关的社区和网站找到资料和书籍。

在熟练使用Java语言的情况下，采用Java SDK接入到区块链，跑起一个Demo Sample，将是非常轻松写意的事情。

如果是采用其他语言，我们也提供了Python、Node.js、Golang等语言的区块链SDK。

不同的语言，其安装包有不同的稳定版本，会采用不同的环境和依赖安装配置方法，会有不同的IDE和调试方法，就不在本文一一罗列，相信学习和使用语言这件事本身，于程序员已经是最基本的技能了。

最后，作为在开源世界里冲浪的玩家，“全球最大同性交友网站”——github一定是要上的了。

注册github账号，掌握git版本管理工具的基本操作，clone和pull开源软件代码，提交issue，commit自己的修改，给开源项目提交pull request，再顺手点个star，激情而有范儿，在开源世界里留下你的姓名。

二、区块链领域的基础知识栈

以下部分的知识和区块链或区块链某一个平台更加相关，从底到上依次如下：

HASH（哈希算法）、签名、证书

严格来说，这并不是区块链领域的专有知识，只是必须具备的基础知识，包括SHA3/SHA256/RIPEMD160等摘要算法，以及这些算法和“区块链地址”的关系，基于公私钥的数字签名和验证方法，数字证书的概念和格式，比如X.509证书，以及保存证书/公私钥的文件格式，如PEM文件、keystore文件等。

基础应用密码学

基础应用密码学其实范围很广，作为入门者，至少要了解对称和非对称加密的常见算法，如AES对称加密，RSA、ECDSA椭圆曲线等非对称加密算法，以及这些算法在签名验签、数据加密、通信协商和保护方面的作用。如果要使用国密，那么需要了解SM2~SM9一系列算法的概念和使用。

分布式网络结构

区块链是先天的“分布式网络系统”，节点和节点通过网络的P2P端口互连，客户端、SDK通过RPC/Channel端口互连，首先要保证网络之间是互通的，监听的地址和端口是对的，端口是开放的，防火墙和网络策略是正确的，用于安全连接的证书已经到位，才能保证区块链的“通则不痛”。

这也要求使用者具备基本的网络知识、网络工具，同时了解区块链特有的节点类型（共识节点、观察节点、轻节点等）、互连方式（点对点双向连接、JSON RPC的HTTP短连接、Channel长连接等）。详情点击参考《FISCO BCOS网络端口讲解》。

智能合约

智能合约可说是应用开发者直面区块链的一道大门，入得此门，精彩无穷。流行的智能合约语言是Solidity语言，这门源自以太坊，从诞生开始就是为区块链而来的。

Solidity语言更新活跃、文档完备，具有良好的一致性和事务性，功能足够实现中型的商业应用。

当然，它在实时调试、第三库支持、运行速度等方面还比不上成熟的语言，如果开发者想要用C++等语言编写智能合约，那就要对区块链上的计算范式进行深入了解，避免写出无法共识的智能合约来，一般是建议有深入的了解后再采用Solidity之外的其他语言编写合约。

要掌握Solidity合约，当然是通读文档，并动手尝试。具体参考以下文：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/smart_contract.html

ABI接口原理

在采用EVM作为虚拟机的区块链上，EVM执行的是Solidity语言的合约。合约编译会生成后缀名为ABI的文件，其实里面就是该合约接口定义的JSON文本，可以用文本查看器查阅，了解你写的合约如何翻译成ABI里的接口，接口返回类型，参数列表，参数类型等，只要有合约的ABI文件，就可以调用区块链SDK的接口，解析这个合约相关的交易、返回值、回执等。

区块数据结构

区块（Block）有区块头和区块体。区块体有交易列表，交易列表里的每个交易（Transaction或Tx）有发起方、目标地址、调用方法和参数，以及发送者签名。交易的结果会生成一个“回执（Receipt）”，回执里包含被调用方法的返回值、运行过程生成的EventLog等……

了解这些，基本上就掌握了区块链数据的脉络，还可以继续深究数据结构里的merkle root以及对应的merkle tree是如何生成的，有什么作用（如用于SPV：Simplified PaymentVerification）。具体参考以下文档：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/design/protocol_description.html

RPC接口

这里把区块链节点暴露的功能接口统称为“RPC接口”。查看链上数据，包括区块、交易、回执、系统信息、配置信息，向链上发起交易，以调用智能合约、修改系统配置等，或者通过AMOP协议发送消息、监听事件，都是通过RPC接口。

几十个RPC接口建议一一走读，或善用搜索，以发现自己想要的接口。

接口通信采用的协议可能是JSON RPC，或者是FISCO BCOS独创的Channel协议，SDK基本上已经对接口和协议进行了良好的包装，也可以在深入理解ABI和RLP等编码模式前提下自行开发接口客户端。具体参考以下文档：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/api.html

准入和权限模型

联盟链强调安全可控，节点准入是第一步，在链初始化后，其他节点或者SDK配置了相应的证书，才能接入到既有的联盟链上。

链上的角色用权限模型控制，包括管理员权限、发布合约的权限、创建表的权限、参数配置权限等，以避免角色之间操作混淆，某些角色既当运动员又当裁判员。

初学者需要仔细阅读区块链平台提供的技术文档了解原理，遵循操作手册的步骤进行操作。具体参考以下文档：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/permission_control.html

数据存储模型

区块链节点会采用文件数据库（LevelDB或RocksDB），或者关系型数据库如MySQL保存数据，所以，链上是真的有“数据库”的。

写入数据库的数据包括区块、交易、回执、合约产生的状态数据等，是否写入“调用合约产生的历史数据”根据不同的平台而定，FISCO BCOS默认只保存最新的状态值，可以选择性地将修改记录写入“回执”或“历史表”里进行追踪。

FISCO BCOS还提供方案，将历史数据导出到链下数据库进行关联分析。具体参考以下文档：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/design/storage/index.html

共识机制原理

联盟链通常采用插件化共识机制实现，FISCO BCOS提供PBFT和RAFT两种高效共识算法，而不会采用“挖矿”这些高耗能低效率的共识。

共识机制是区块链的灵魂，对共识机制进行深入学习，才可以深入理解区块链通过多方协作、达成高度一致性、支持交易事务性、防篡改防作恶的功效。具体参考以下文档：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/design/consensus/index.html

区块链的知识包罗万象，更深层次的知识还有分布式系统理论、博弈论、前沿密码学、经济学、社会学等，掌握以上的基础知识，再深入学习，举一反三，用场景去验证和探索创新式应用，方可发挥技术的潜力，感受分布式商业的魅力。

三、做一个怎样的学习者

在这个过程中，希望学习者做到：

读文档的耐心

我们的开源项目文档足有20万字以上的篇幅，公众号里还有大量的技术解析和科普文章，这都是程序员们在coding之外，用尽自己仅有的语文储备，码出的海量文字，是一笔巨大的技术财富，涵盖了相关开源项目的方方面面。如果能通读，或者记住文档结构和标题，需要时快速打开，足以解惑且深入。

搜资料的能力

文档、公众号都有搜索功能，当想起和开源社区有关的问题时，可以随手用关键字搜索，一般都能找到答案。如果有语言不详之处，可以向开源项目团队提出，或者根据自己的理解进行补充。通用的知识点，如操作系统、网络等，通过公网搜索引擎，一般都能找到答案。

排查环境和依赖问题的能力

开源软件牵涉的系统环境、第三方软件、软件的版本等常常有错综复杂的依赖关系，太高或太低的版本都可能会有一些问题，请注意阅读项目文档对软硬件环境和依赖的描述，保证自己的环境符合要求，并善用配置管理工具、软件安装工具获取和设置合适的版本。

调试能力

如上所述，Solidity语言的调试工具完善程度尚未达到完美，但可以善用合约的返回值、EventLog等方式，通过WeBASE、控制台等趁手的工具进行调试，并查阅Solidity文档，了解问题可能出在哪里。

区块链节点的日志开启debug级别后，也会打印详细的信息，可以查阅运行日志，获取运行信息和可能的错误信息，将这些信息与自己所做的操作比如发交易的流程结合起来进行分析，提高调试效率。

同时，目前的开源软件通常会在屏幕上打印错误原因和解决问题的提示，仔细查看操作反馈，大概率能了解错误原因和解决方案。

代码阅读能力

开源软件的最大效能是把代码毫无遗漏的摊到了开发者和学习者面前，了解代码结构，查阅代码里的关键流程，用关键字去搜索代码里的对应实现，都可以深入系统细节，挖掘设计思想，定位问题，寻找优化方法。一个好学且硬核的程序员，足可通过代码，和世界对话。

问问题的方式方法

“一个好问题，比答案还重要”。我们的社区非常活跃，大家都很热情地答复和解决问题。我们鼓励在社区里公开提出问题，一方面使大家都可以分享问题，找到解决方案，另一方面提问者也可以得到更多人的帮助。同时，希望提问者提出问题时，一次性描述详尽，把相关的操作步骤、系统环境、软件版本、出错提示以及希望得到的解决方案都提出来。

如果是通用性的问题，可以先搜索再提问，有利于培养独立解决问题的能力。希望提问者能向社区反馈更深层次的问题，以帮助社区快速优化。对很多典型问题，社区也积累了一些行之有效的解决方案，我们会整理和公布出来，以便查阅。

从新人到老鸟的路也许漫漫，如果能参考这篇小文的一些方法，可以少踩许多坑，多写一些应用。Enjoy blockchain，社区与你共同进步。

资源链接：

各项开源组件索引

FISCO BCOS开源文档

新人请注重阅读：

关键概念

以及使用手册

FISCO BCOS公众号的开发教程仓库

《麻雀虽小五脏俱全 | 从Python-SDK谈谈FISCO BCOS多语言SDK》，此文详细讲解了从客户端的应用出发，去理解区块链接口的方法。

Solidity智能合约（中文）（注意选择对应版本）

Solidity智能合约（英文）注意选择对应版本）

《如何优雅地写智能合约》

《WeBASE数据导出：助力区块链大数据场景快速落地》，区块链数据导出到链外，采用海量数据存储。

图形化的区块链，WeBase文档

推荐书目：

《鸟哥的linux私房菜》（系列）

《UNIX网络编程》（系列）

《java核心技术》（系列）

《Springboot实战》

《Spring实战》

22.3.2 排除万难，从入门到精通区块链

作者：张开翔 | FISCO BCOS 首席架构师

目前越来越多的人已经进入或准备进入区块链领域，过程中不免抱着各样的疑虑和问题。想起自己之前用几年时间，从“略懂”区块链到all in，同样也经历着类似的心路历程，这个领域确实是有一些门槛的，但万事开头难，摸索路上还远不止八十一难，这里梳理几个概要性的困难和感悟，谨作分享。

方向之难

我是谁”、“我在哪”、“我要去哪儿”，一切都源于这么一个哲学三问。区块链是什么？区块链究竟能做什么？为什么区块链这么火？不用区块链行不行？这些问题都充满了终极拷问的意味。

其实很难彻底回答这些问题，因为，并没有标准答案。所有创新的前沿的事物大抵如此，在质疑和动荡中发展，在黑暗和荒芜中摸索，精华和糟粕齐飞，绿洲与韭菜一色，直到引爆点迸发。如果因为充满疑虑而左右摇摆，或者裹足不前，那么在从事相关工作研究的过程，体验会很糟糕，结果也不会好。

分享一点个人体会：区块链领域从一开始就吸引了全世界无数的聪明人，不乏极客、学者、大师, 他们进行了大量的技术和社会实践。这个领域蕴含着数学、计算机、密码学、博弈论、经济学、社会学等学科的精华，这是一个智力飞扬、思想激荡的世界。目前，整个行业更是获得前所未有的关注，包括政府、各行业巨头都在关注，大量注意力和资源持续涌入，区块链迎来了“最好的时节”。

在持续进化、结构多元的现代社会里，分布式商业的理念已经成为现实。人和人、机构和机构之间会产生更多的联系和协作，信息和价值在新型网络模式中快速流转，作为分布式技术代表之一的区块链，很有机会成为新一代基础设施和创新的据点。

所以，方向并不是问题。即使你不把区块链当做“信仰”，只看这个充满魅力的技术本身，以及区块链与实体经济深度融合的机遇，还是可以给我们带来信心。

概念之难

在哲学三问中，“区块链是什么”是最晦涩的问题，区块、交易、账户、共识、智能合约、双花...这都是什么？！我自己在刚接触区块链时，也有一种被颠覆了认知的感觉。有一些介绍区块链的文章，往往着眼在区块链的社会和经济效能，从价值理念讲起，这些固然是必要的，但俗话说“科学要定性更要定量”，作为工程技术人员，我们更应该关注的是，区块链里的知识点、基本原理，进而澄清术语、把握架构、处理逻辑和程序流程。

前面提到，区块链蕴含着大量学科的精华，同时，行业也有一句俗语：“**区块链并没有发明什么新技术，都是成熟技术的组合**”。

- 这些成熟的技术包括数据结构，诸如链表、树、图、过滤器等，是大学里数据结构的基础知识；

- 基础密码学，包括HASH和对称非对称加密、数字签名等等，已经是几十年的经典技术，而事关隐私保护等领域的新一代密码学，更给密码学专业人士开拓了一片广阔的发挥空间；
- 分布式网络和系统这门学科本身包罗万象，覆盖了海量服务的知识范畴，许多从事过海量互联网服务技术的同学，都会对区块链的P2P网络和共识算法、并行计算模型、事务性一致性原理如数家珍；
- 博弈论和激励相容是协作方面的知识点，是"区块链思维"的重要组成部分，工科学生可能需要翻翻书，社科经管背景的同学估计会似曾相识吧；
- 至于智能合约，比如solidity语言、WebAssembly等，很少听说吗？其实这些语言以及编程模式学习曲线未必有javascript高，有几年程序基础的同学基本上一周就可以上手，写出顺溜的智能合约来。

区块链让人觉得认知困难，是因为它就像个"筐"，什么都可以往里装，牵涉的技术繁杂，组合方式却和常规技术套路不同。所以学习者一定程度上先要放空自己，避免让自己在原有领域的思维定势干扰了学习，在丰富自己知识面的同时，接受区块链的"群体共识"、"防止篡改"、"不可否认"、"高度一致性"等神奇逻辑，然后潜心进去看每一个独立的概念时，并不会觉得高不可攀。

突破概念之难的要点，是排除来自各种渠道的噪声，有些信息似是而非，或者是各说各话，把同一件事用N种话术讲出来，混淆了定义，模糊了本质，于事无补的同时还带来更多疑问。靠谱的方法是着重阅读权威媒体的正规内容，关注一些主流区块链项目的文档库，认真地、全面地通读技术文档，然后找一个自己感兴趣的领域（如共识算法），结合自身经验知识进行对照研究。

同时，也可以加入活跃的开源社区和技术圈子，和有经验的人多讨论，勇于把问题抛出来，挨个术语，挨个流程讨论透彻。我们在研究区块链的初期，团队经常就一个概念的定义咬文嚼字地争论很久，最后愉快地达成共识时，大家都觉得神清气爽。

在概念阶段，切忌求全责备，不要变成资料收集机，一口吃不成胖子，基于靠谱的学习资料，澄清基本概念，在实践中去验证和发掘新的概念，建立发现问题、解决问题的方法论，慢慢就能举一反三，说不定某一天就能醍醐灌顶了。

上手之难

好吧，哲学和概念问题终于不会阻碍我们学习的脚步了，那么怎么继续"二十一天入门到精通"之路。作为技术人员，遇到新的技术平台、软件体系、编程语言...那当然是："不要怂，就是干！"

几年前，我们刚开始研究区块链时，通读过几个国外流行的开源区块链项目代码，并搭建测试网进行试验剖析，分析如何让这些平台能在复杂的金融业务上用起来。当时有个困惑，如果直接基于底层平台开发应用，当需要实现更多功能时，是不是就得直接修改底层平台代码。

而当看到"智能合约"这个东西时，思路一下就打开了：采用智能合约作为中间层，在合约里编写业务逻辑，并为调用者定义清晰的功能接口，这样，业务可以很好和底层解耦，而底层平台则定位成强大的引擎，通过架构的解耦，使得整个开发过程变得清晰合理、轻松愉快，感觉就像是从"C/S"架构到"B/S"架构的演变，多年互联网海量服务开发的手感又回来了。

另外，我们觉得当时的开源项目主要是公链形态，在安全可控以及合规方面考虑没有那么周全，并不适合金融场景使用。

那么，没有趁手的平台，就想办法造一个。从此，深耕底层技术、迭代应用验证的漫漫之路开启了。这个过程也和开源社区一众伙伴建立了紧密的合作，正是"来自开源并回馈开源"，经过开源工作组几年的打磨，FISCO BCOS已经是一个全面开源、安全可控、高速稳定、易用友好的金融级底层技术平台，面向金融和广大的产业领域，提供丰富的功能和各种操作工具。丰富全面的文档和便捷易用的体验，可以帮助开发者从快速入门到精通，整体的技术门槛和开发成本变得前所未有的低。

有了基础的底层平台之后，下载、安装、配置、运行、阅读使用手册、写hello world和业务应用、Debug和分析日志...都是step by step的工作了。

我们目标是，用户在几分钟内，用一键安装、docker、云服务等方式顺畅无错地搭建出自己的区块链网络，在一周内通过学习就可以写出完整的智能合约，基于支持多款语言的SDK（Java、Node.js、Python、Go...还在增补中）实现业务逻辑，将业务发布上线，保持稳定运行。

为了这个目标，我们一直在持续优化使用文档、开发手册，以及部署和运维工具。众所周知，“码农”们喜欢写代码，而写注释和文档就比较痛苦了，为了向社区交出一份漂亮的作业，大家倾尽了有生以来的语文水平，一次又一次修订，硬是写出了数十万字的文档库。

同时，开源社区推出了一系列的线下线上沙龙、培训,用社区的方式，进行广泛交流和技术支持。在多次的现场学习和黑客松大赛中，我们欣喜地看到,开发者用两三天时间，就能基于FISCO BCOS实现他们精巧的项目设计，而且有开发者将其中和开源项目相关的优化贡献到Github上。

到这个程度，即使是对没有区块链研发经验的开发人员,快速上手已经没有什么问题，即便区块链底层还像一个黑盒子有待探索，但就像在电脑上安装App、使用mysql、tomcat之类的软件一样，足可以用起来，感受区块链的魅力了。

深入之难

对技术人员来说，探索技术的内涵永无止境：参与到区块链底层开发，实现大型的区块链应用，为区块链生态增加更多有用的特性、工具，对现有功能性能进行极致优化，这些都是进入“深水区”的路线。

之前提到，区块链系统知识点和框架涉猎广泛，无论是知识面还是深度都有相当的规模。量化一下的话，可以套用一万小时理论：如果每天学习工作8-10小时，一个月足以上手，一年可得小成，两年轻车熟路，三年可成老司机...但老司机要行的前路依旧漫漫。我们希望通过持续的科普、交流、实践，去缩短这个过程，但毕竟学习就是一种最基本的“工作量证明”，并没有什么其他捷径。

学习方法，首先是大量的泛读，每天早上一睁眼到晚上，都可以看到持续更新的行业新闻、公众号文章、技术大咖博客、邮件组讨论组、开源项目.....阅读的过程或许会遇到不同观点的碰撞，需要去伪存真，在心态开放的同时，也保持自己的立场和方向。

然后是深入的精读，先选定一两个感兴趣的方向，研读诸如密码学、分布式理论等方面的一些经典论文。FISCO BCOS的核心共识算法使用的PBFT和RAFT算法，是基于对原版论文的研究解读，有了深入的理解再去做的实现和优化。区块链里广泛用到密码学原理，场景和逻辑多变，其原理有可能来自某一篇“顶会”论文。精读深度原理剖析文章和学术论文，基于扎实的理论，才能根据自己的需要进行发挥，创造性地解决工程问题。

其实我们的在线文档已经有数十万字的规模，各种信息应有尽有，技术社区会定期对热点知识进行解读，只要读者认真阅读在线技术文档，接受公众号的体贴推送，并动手进行更多的实践，随着时间推移，必能深度理解区块链的技术原理，参透架构设计的来龙去脉，建立起巩固的知识体系。

最后，精读的对象，自然也包括源代码，毕竟“Talk is cheap, show me the code”，区块链开源项目代码大多是几万到几十万行的级别，阅读代码是达成庖丁解牛水准最直接的方法。在研究区块链的历程里，我们有许多长夜漫漫review代码的日子，当读到一眯眼，眼前都是代码在飞，各种接口和对象翩翩起舞，既优雅又有规律，脉络清晰，那种愉悦简直难以名状。这种体验，之前有，之后也会继续有。

如果已经深入到了这个程度，领域门槛已经基本越过，考验的就是开发者的脑力和体力了吧。

持续之难

在过去几年因为鱼目混珠、政策法规、技术障碍等等一系列情况，区块链会遇到市场波动、应用落地延缓等挑战。未来如何，虽然没有预言家告诉我们，但现在大家已经看到了趋势。这又回到了第一个问题：“方向”，明确清晰的方向，不但能回答“要不要进入这个领域”，同样也能回答“要不要坚持下去”。我们一直在分布式商业模式中开拓场景，为公众提供优质的服务，为行业提供完备和好用的开源技术，这是从开始到现在，乃至未来坚持的方向，从未改变。

再具体一点，如果我们已经用区块链部署上线了业务系统，那么影响系统生命周期和持续性的问题还有：可运维性、可升级性、兼容性、数据容量、业务性能容量等等。

多次和社区朋友交流的过程，他们会提起一些问题，例如，新版本是否能兼容旧版本？随着业务发展，越来越多的数据是否可以迁移和重用？这都是用户的真实声音。我们所构建的平台，一定要走可持续发展的路线，重视软件体系的兼容性，有合理的版本发布节奏，以及周全的数据迁移维护策略，可以更好地保护社区用户利益，也使得用户愿意长期与社区共同发展。

另外，区块链领域还在高速发展中，各种新技术、新思潮、新模式、新政策还在层出不穷，这个领域集结了世界上大量的聪明人，他们不但聪明而且努力，从来都不闲着。于是在这个领域工作，每天都会有新的知识、新的刺激，这一方面是一种幸运，另一方面，也会让人极其焦虑。

如何去消化这么海量的信息，如何去探索和掌握前沿的知识，如何更好满足高速发展带来的用户需求和新挑战，如何做出卓有成效的突破性创新，这真的是一个创新和焦虑并存的世界，让人欲罢不能。

作为从业者，必须持续进行大量阅读，在各种信息流里过滤和吸收，不断地归纳/总结/思考/开拓；每一个需求和用户ISSUE反馈都是一个小目标，每发布一个新版本都会是下一个版本的新起点。区块链的世界和其他技术领域没有什么不同，都必须保持敏锐和奔跑，保持好奇和谦逊，不断学习和实践，修订短板寻求突破，将成果分享给社区。正如系统需要极大的弹性，人也需要极大的韧性。共勉。

推荐链接

《亲朋好友都能看懂的区块链》

FISCO BCOS开源项目github地址

关键概念阅读

《FISCO BCOS零基础入门，五步轻松构建应用》

深入区块链底层平台FISCO BCOS

22.3.3 带你读源码：四大视角多维走读区块链源码

作者：李辉忠 | FISCO BCOS 高级架构师

引子

区块链作为「新基建」的重要组成部分，越来越受技术爱好者关注。区块链极客信奉“code is law”，相信通过代码可以构筑一个可信的世界。

而作为一门综合学科技术，区块链建立在数学、密码学、计算机原理、分布式网络和博弈论等众多基础学科之上，底层代码动辄数十万行，如果没有摸清门道，要完全掌握这些代码是极具挑战的。

本文希望给读者一个走读区块链源码的方法，让读者面对区块链底层项目时可以从容地说出“show me the code”。

基础知识储备

区块链是一门综合学科，涉及多个专业领域，涵括多方面的基础知识，在深度研究区块链之前需要做一定广度的知识储备。注意，这里说的是广度，并非深度，也就是说你只需要大概知道这些基础知识的基本原理与作用即可。

- 密码学相关：理解哈希、对称加密、非对称加密以及数字签名的基本原理和作用；
- 计算机操作系统相关：理解多进程、多线程、互斥、并行等相关概念和作用；
- 数据结构相关：理解队列、堆栈、树等基本数据结构和使用场景；
- 计算机网络相关：理解TCP/IP、心跳包、消息流等基本概念；
- 数据库相关：理解数据库基本概念，了解KV数据库的基本原理；
- 计算机原理相关：理解程序编译、解析、执行和字节码、虚拟机等概念；
- 分布式系统相关：理解点对点网络、分布式一致性、CAP等相关概念和基本原理；
- 程序开发相关：掌握相关的编程语言、构建工具等，理解项目构建基本流程。

多维走读

在储备了相关的基础知识之后，你就可以打开一份真正的区块链底层代码了，一般通过git clone可以快速下载到项目代码。

但是，面对数十万行的代码，该从何看起呢?

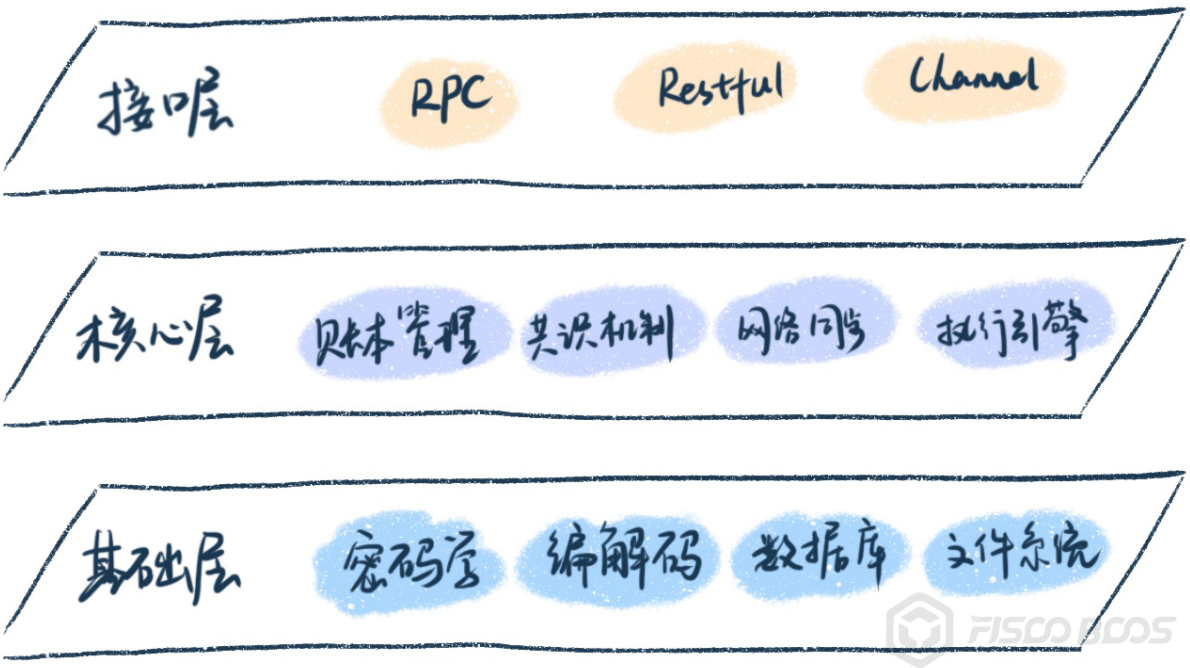
一个优秀的区块链底层项目，必然有一份优秀的工程代码，这份代码有其合理的组织结构与纹理逻辑。走读代码应效仿庖丁解牛，先摸清区块链的基本结构和逻辑，再开始走读，可以达到事半功倍的效果。

本文推荐要从四个不同视角进行走读，站在自己的需求角度出发去看代码，而不要被巨量的代码所左右。这四个角度为功能视角、系统视角、用户视角和开发视角，分别从逻辑层面、运行层面、使用层面和开发层面厘清代码架构和关键算法。

功能视角

在深入一份区块链底层代码之前，首先要通过其官网、技术文档、github wiki等渠道获取项目设计文档，了解其基本功能设计。

一般每个项目都会提供核心功能列表、总体架构图、功能模块图等介绍文档，通过这些介绍可以掌握项目基本功能。即使你真的找不到也不打紧，大部分区块链底层项目在功能设计层面的差异较小，核心功能模块也大致相同。



以FISCO BCOS为例，基础层代码如下：

基础层	libdevcore	提供数据编解码、数据库、文件系统、线程池、锁等常用工具库
	libdevcrypto	提供国密与非国密的哈希、对称加密、非对称加密算法
	libethcore	提供基本的交易、回执和区块定义，以及公共基础类等

核心层核心代码如下：

账本管理	libblockchain	管理区块、交易、区块链状态等信息的读写
	libstorage	数据落盘存储模块，通过Storage和MemoryTable框架管理
	libstoragestate	采用storage模式的状态存储实现
	libmptstate	采用mpt模式的状态存储实现
共识机制	libconsensus	插件化共识框架，Sealer封装打包流程，ConsensuEngineBase封装共识框架
	libconsensus/pbft	Pbft共识引擎
	libconsensus/raft	Raft共识引擎
	libconsensus/rotating_pbft	rPBFT共识引擎
网络同步	libnetwork	网络框架，实现节点互联和消息收发逻辑，核心逻辑在Host
	libp2p	P2P消息发送与处理的实现
	libsync	区块、交易的同步逻辑，由SyncMaster管理
交易执行	libblockverifier	一个区块的执行逻辑，包括DAG并行实现
	libexecutive	一笔交易的执行逻辑，包括合约部署
	libevm	Evm虚拟机的接口
	libinterpreter	Evm虚拟机的解释执行版实现
	libprecompiled	预编译合约的逻辑

接口层核心代码如下：

接口层	librpc	RPC接口定义与实现
	libchannelserver	Channel接口定义与实现

从功能视角出发，先定位核心功能模块的代码位置，再仔细深入各个功能代码，从单个功能模块内，也可继续递归采用功能视角拆分法，广度遍历直至了解全貌。

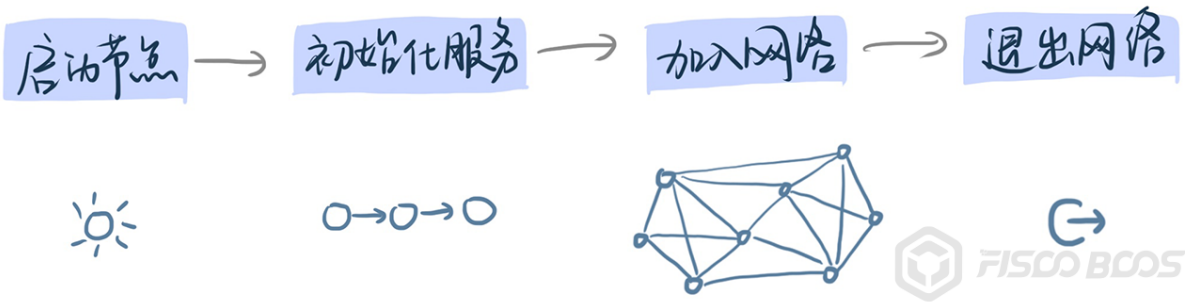
系统视角

系统视角从整个区块链网络运行角度，关注区块链节点全生命周期所参与的系统行为。

关注点包括从敲下启动节点的命令开始，节点经历了哪些初始化环节，之后又是如何与其他节点建立点对点网络，以及完成分布式协作的。

由于不同区块链在部署架构上略有差异，系统运行方式也有所不同，但万变不离其宗，系统视角来看，每个区块链系统都要经历节点初始化、建立点对点网络、完成分布式交互的过程。

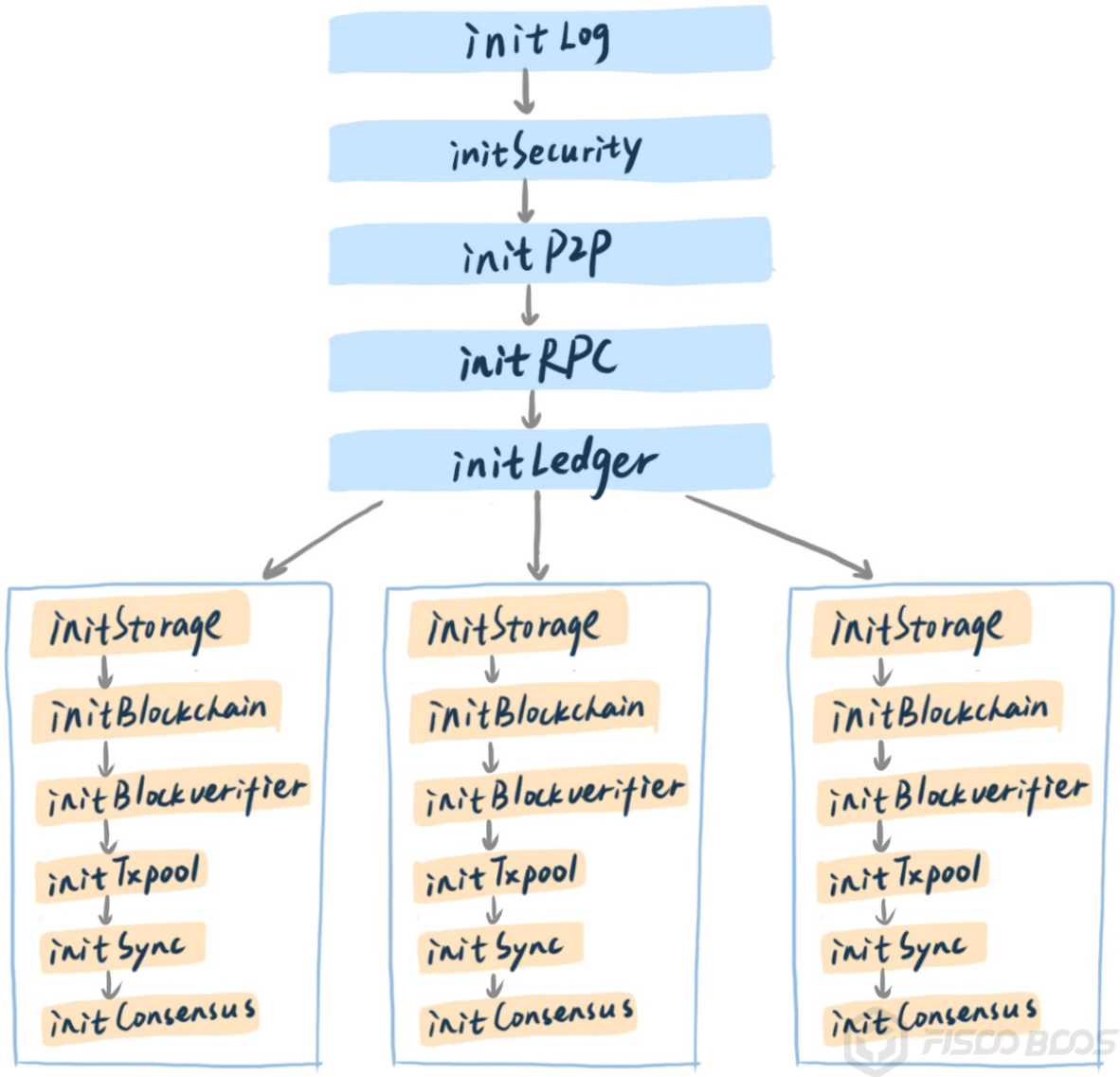
从系统视角看区块链，首先要关注初始化工作。以FISCO BCOS为例，区块链节点启动从main函数入口进入，通过libinitializer模块初始化并启动各模块，启动顺序如下：



通过启动顺序可以知道FISCO BCOS的一个重要特性——支持多群组账本，每个群组是一个独立的Ledger模块，每个Ledger具有独立的存储、同步、共识处理功能。

完成初始化工作同时，系统将会启动若干线程（或者进程、协程，原理类似），这些线程包括网络监听、共识、消息同步等，可以结合代码分析与系统命令查看运行节点配合确定有哪些关键线程，搞清楚关键线程的工作机制就可以基本掌握区块链系统运行机制。

以FISCO BCOS为例，节点启动之后的关键线程以及他们之间的关系如下：



初始化完成之后，网络模块的Host线程将根据配置列表，主动与其他节点建立连接，并且持续监听来自其他节点的连接；Sync线程开始相互发送区块高度，发现高度低于其他节点则开启下载逻辑；RPC与Channel线程等待客户端发送请求，将收到的交易塞入txpool；Sealer线程从txpool获取交易，Consensus线程则开始处理共识消息包。

如此，整个区块链系统有条不紊地运转，完成客户端请求与分布式协作。

用户视角

用户视角关注操作接口和交易生命周期，关注访问区块链的接口和协议设计、编解码方式、核心数据结构、错误码规范等，还会关注如何发送一笔交易到链上，交易在链上又经历了哪些处理流程，直到达成全网共识。

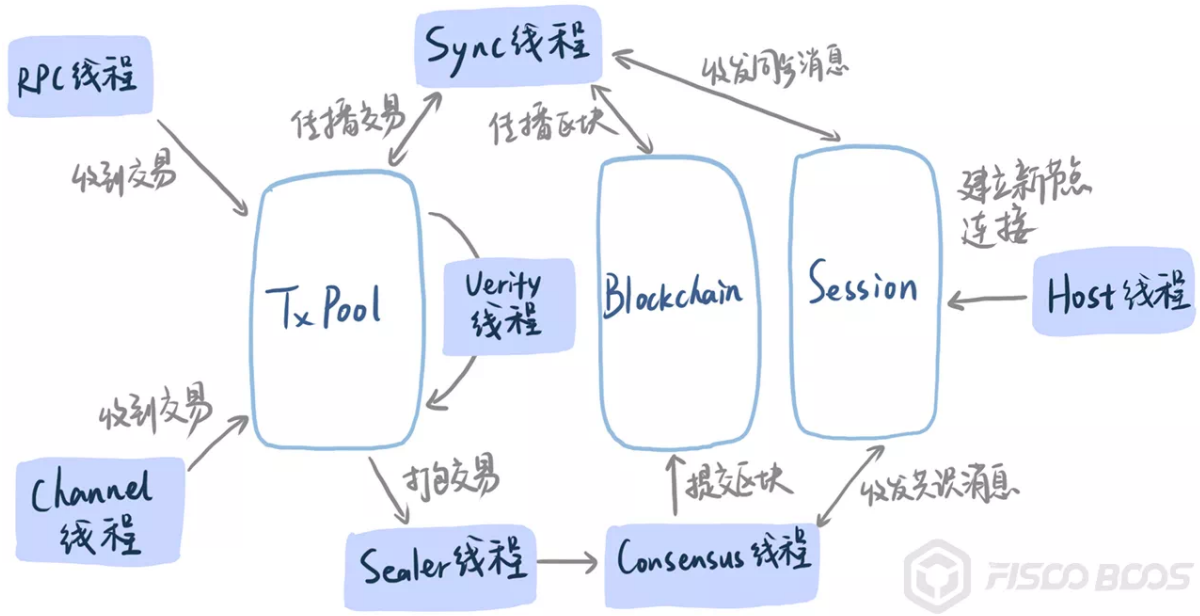
一般区块链底层项目都会给出交互协议的说明文档，通常实现包括JsonRPC、gRPC、Restful等不同类型的交互协议。

不同项目的交互接口会有所不同，但大都会包含发送交易、部署合约、调用合约、查看区块、查看交易以及回执、查看区块链状态等接口。不同项目的数据编码也会有所不同，有些采用Json，有些采用protobuf等。

当从技术文档中了解清楚交互协议、接口、编解码和错误码等设计细节之后，接下来最重要的是通过发送交易、部署合约、调用合约这些关键接口，对代码进行抽丝剥茧，贯穿交易整个生命周期，从而搞清

楚区块链底层最核心的逻辑。

以FISCO BCOS为例，通过多个模块相互协作，完成交易整个生命周期的处理：

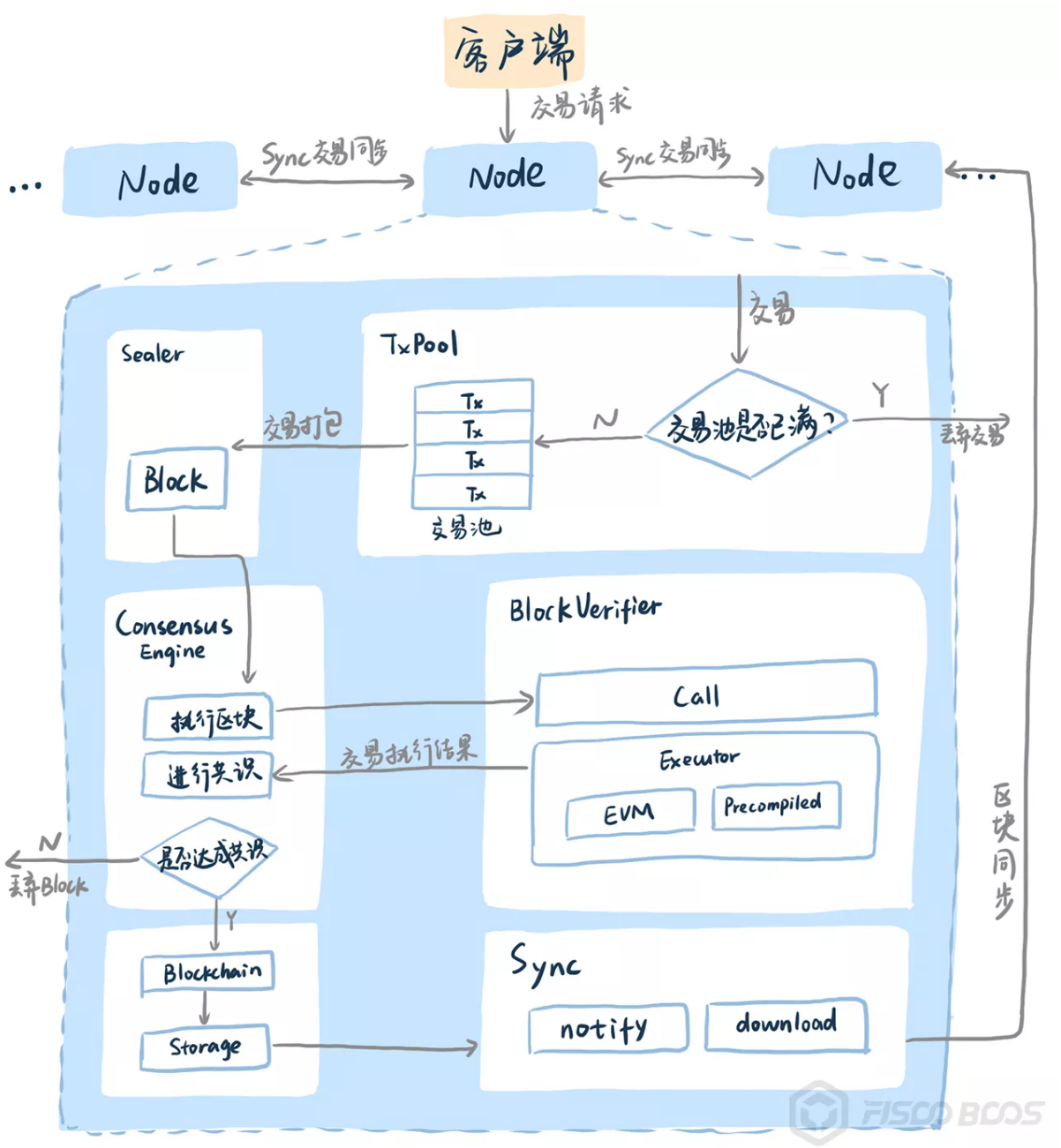


开发视角

开发视角关注的是整个代码工程，包括第三方依赖，源码模块之间的相互关系，单元测试框架和测试用例，编译和构建方式，持续集成和benchmark，以及如何参与社区源码贡献等等。

不同语言都有相应推荐的编译构建方式以及单测框架，通常在区块链项目源码目录可以快速定位到第三方依赖库，比如以cmake构建的C++项目有CmakeLists.txt文件，go项目有go.mod文件，rust项目有cargo.toml文件等。

以FISCO BCOS为例，从CmakeLists.txt可以看到依赖库包括：



项目核心源码包括fisco-bcos程序入口代码，以及libxxx的各模块代码，根据模块的名字可以快速识别其对应功能，这里也体现了一个项目源码质量的高低，高质量的代码应该是“代码即注释”。

单元测试代码在test目录，采用boost的单元测试框架，子目录unittests中单测代码与源码目录一一对应，非常容易找到源码对应的单元测试代码。

构建和持续集成工具代码在tools目录，子目录ci中维护了多个不同场景的持续集成用例，在github提交的每一个pr（pull request）都会触发这些持续集成用例，当且仅当每个用例成功通过方可允许合入pr。

关于FISCO BCOS的代码规范和贡献方式，在CODING_STYLE.md和CONTRIBUTING.md文件中有详细描述，鼓励社区用户积极参与贡献。

总结

区块链涉及领域和知识较多，需要深入源码细节，才能真正完全掌握区块链核心技术。所谓“重剑无锋，大巧不工”，掌握源码走读的基本方法论，才能在巨量代码前，面不改色心不跳。

本文提出从功能、系统、用户和开发四个不同视角进行区块链底层代码走读的方法，一般来说，依次选择不同视角进行走读是比较推荐的方式，也可以根据个人喜好和能力模型选择视角顺序。

最后，本文所举示例皆为FISCO BCOS，但这套走读方法可以适用于任何其他区块链底层项目，希望本文对你有所帮助。

22.4 FISCO BCOS的原理和特性

22.4.1 整体架构

架构模型：一体两翼多引擎

群组架构：支持链内动态扩展多群组

FISCO BCOS 2.0原理解析篇1： 群组架构的设计

作者：陈宇杰 | FISCO BCOS 核心开发者

为了方便企业、开发者更深入理解FISCO BCOS 2.0诸多新特性，更快速地运用FISCO BCOS搭建联盟链应用，我们启动了FISCO BCOS 2.0系列剖析的计划。在后续的推送中，我们将陆续推出《FISCO BCOS 2.0原理解析》、《FISCO BCOS 2.0使用教程》、《FISCO BCOS 2.0源码分析》等文章系列，抽丝剥茧地将FISCO BCOS 2.0进行全面拆解。

本文是原理解析系列第一篇，介绍FISCO BCOS 2.0众多新特性中的主线——群组架构。主要包括群组架构的整体架构设计，群组架构包括哪些组件，各组件的主要功能以及组件间的交互。

设计目标

理解群组架构的设计目标，可以从人人都熟悉的群聊模式说起。

灵活扩展：保证业务接入和扩展像拉群聊天一样方便

群的建立非常灵活，几个人就可以快速拉个主题群进行交流。同一个人可以参与到自己感兴趣的多个群里，并行地收发信息。现有的群也可以继续增加成员。

看回群组架构，采用群组架构的网络中，根据业务场景的不同，可存在多个不同的账本，区块链节点可以根据业务关系选择群组加入，参与到对应账本的数据共享和共识过程中。群组架构具有良好的扩展性，一个机构一旦参与到这样的联盟链里，有机会灵活快速地丰富业务场景和扩大业务规模，而系统的运维复杂度和管理成本也线性下降。

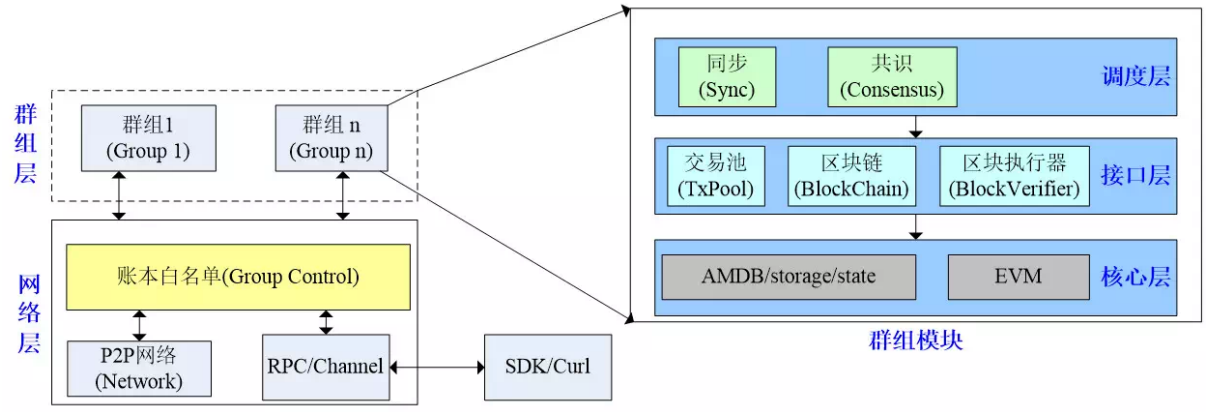
隐私保护：各群组之间解除耦合独立运作

回想一下群聊场景：群聊用户都在你的通信录中，都是经过验证才添加的，且不在群里的用户看不到群聊信息。这与联盟链准入机制不谋而合，所有参与者的机构身份可知。

另一方面，群组架构中各群组独立执行共识流程，各组独立维护自己的交易事务和数据，不受其他群组影响。这样的好处是，可以使得各群组之间解除耦合独立运作，从而达成更好的隐私隔离。在跨群组之间的消息互通，则会带上验证信息，是可信和可追溯的。

架构设计

架构设计全景

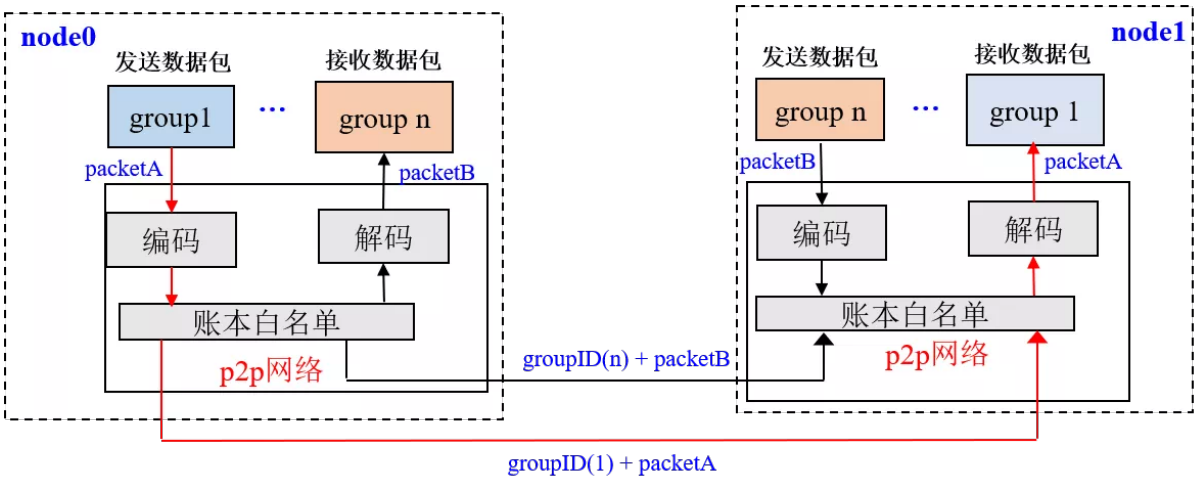


▲ 图为群组架构设计全景

如上图所示，群组架构自底向下主要划分为网络层、群组层，网络层主要负责区块链节点间通信，群组层主要负责处理群组内交易，每个群组均运行着一个独立的账本。

网络层

群组架构中，所有群组共享P2P网络，群组层传递给网络层的数据包中含有群组ID信息，接收节点根据数据包中的群组ID，将收到的数据包传递给目标节点的相应群组。为了做到群组间通信数据隔离，群组架构引入了**账本白名单**机制，下图展示了群组架构下群组间收发消息的流程：



账本白名单

每个群组均持有一个账本白名单，用于维护该群组的节点列表。为了保证账本白名单群组内一致性，仅可通过发交易共识的方式修改账本白名单。

发包流程

以node0的第一组向node1的第一组发送消息packetA为例：

(1) group1将消息packetA传递到网络层；

(2) 网络层模块对packetA进行编码，在packetA包头加上本群组ID，记为{groupID(1) + packetA}；

(3) 网络层访问账本白名单，判断node0是否是group1的节点，若非group1节点，则拒绝数据包；若是group1节点，则将编码后的包发送给目标节点node1。

收包流程

node1接收到node0 group1的数据包{groupID(1) + packetA}后：

(1) 网络层访问账本白名单，判断源节点node0是否是group1节点，若非group1节点，则拒绝数据包，否则将数据包传递给解码模块；

(2) 解码模块从数据包中解码出group ID=1和数据包packetA，将数据包packetA发送到group1。

通过账本白名单，可以有效地防止群组节点获取其他群组通信消息，保障了群组网络通信的隐私性。

群组层

群组层是群组架构的核心。为了实现群组间账本数据的隔离，每个群组持有单独的账本模块。群组层自下向上一次分为核心层、接口层和调度层：核心层提供底层存储和交易执行接口；接口层是访问核心层的接口；调度层包括同步和共识模块，负责处理交易、同步交易和区块。

核心层

主要包括存储(AMDB/storage/state)和执行(EVM)两大模块。存储负责从底层数据库中存储或读取群组账本的区块数据、区块执行结果、区块信息以及系统表等。执行(EVM)模块主要负责执行交易。

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。

模块1：交易池(TxPool)

交易池是客户端与调度层交互的接口，负责从客户端或者其他节点收到的新交易，共识模块会从中取出交易打包处理，同步模块从中取出新交易进行广播。

模块2：区块链(BlockChain)

区块链模块是核心层与调度层交互的接口，是调度层访问底层存储和执行模块的唯一入口，调度层可通过该模块提交新区块和区块执行结果，查询历史区块等信息。区块链模块也是RPC模块与核心层的接口，RPC模块通过区块链模块可获取区块、块高以及交易执行结果等信息。

模块3：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。

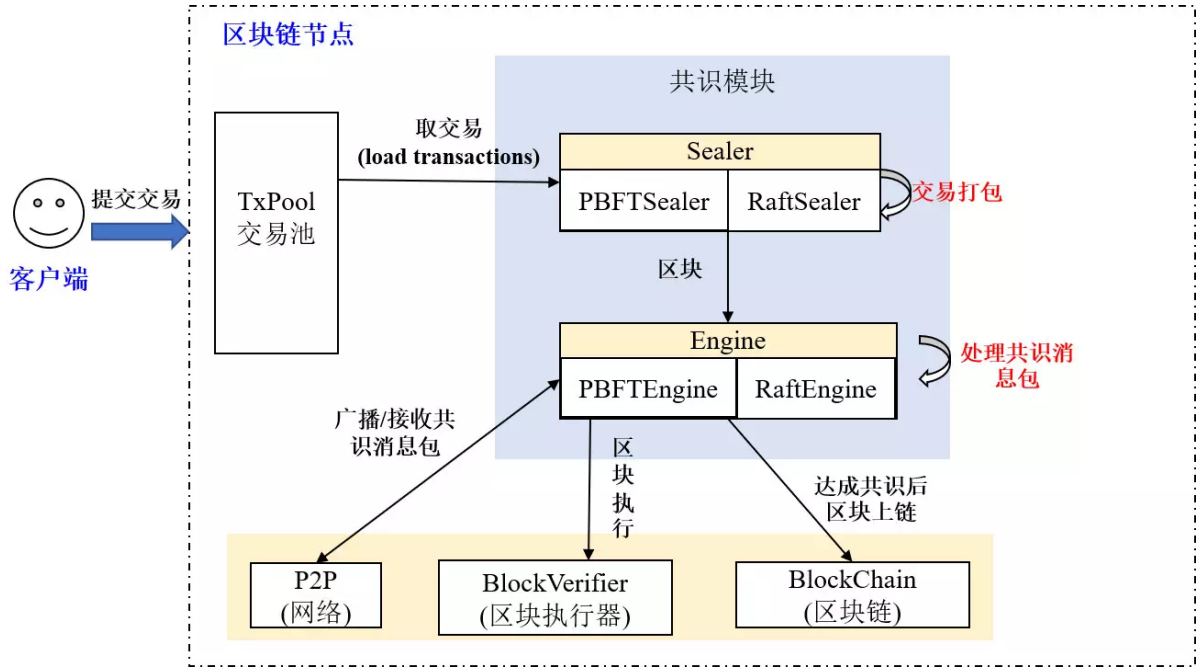
调度层

调度层包括共识模块(Consensus)和同步模块(Sync)。

模块1：共识(Consensus)模块

共识模块主要负责执行客户端提交的交易，并对交易执行结果达成共识。

如下图，共识模块包括打包(Sealer)线程和共识(Engine)线程，Sealer线程负责从交易池获取未执行交易，并将交易打包成区块；Engine线程负责对区块执行结果进行共识，目前主要支持PBFT和Raft共识算法。



共识模块主要流程包括：

(1) 客户端提交的交易缓存到TxPool后，会唤醒共识节点的Sealer线程，Sealer线程从交易池中获取最新交易，并基于当前最高区块，打包产生一个新区块blockI；

(2) Sealer线程将打包产生的新区块blockI传递到Engine线程，用于共识；

(3) Engine线程收到新区块blockI后，启动共识流程，共识过程中，会调用区块执行器BlockVerifier执行区块blockI内的每笔交易，并对执行结果达成一致共识；

(4) 若共识成功，则调用BlockChain将新区块blockI以及区块执行结果等提交到底层数据库；

(5) 新区块blockI上链成功后，触发交易池删除以上链的所有交易，并将交易执行结果以回调的形式推送到客户端。

模块2：同步(Sync)模块

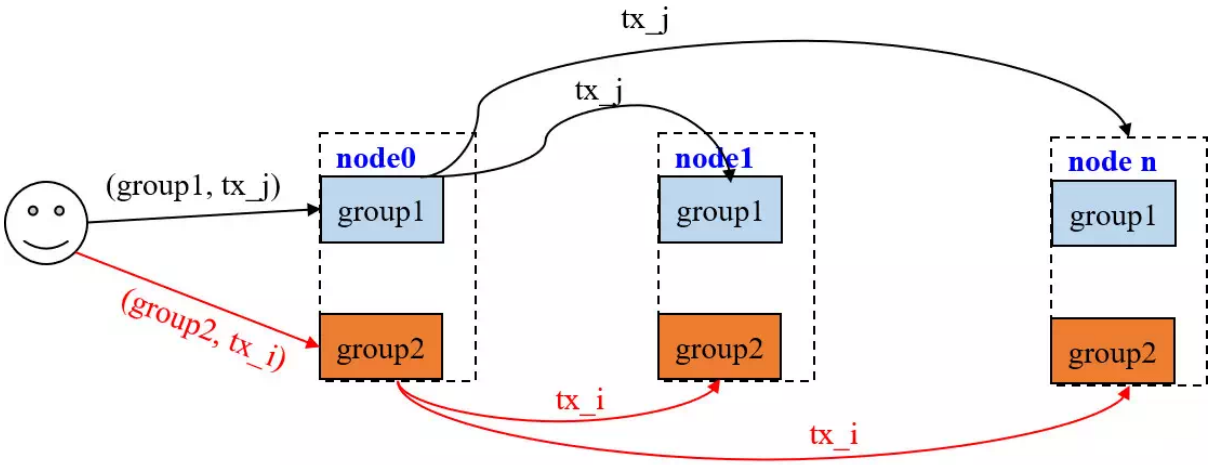
考虑到共识过程中，需要尽可能保证每个群组节点拥有全量的交易，FISCO BCOS 2.0引入了同步模块来保证客户端的交易尽可能发送到每个共识节点。

同步模块主要包括交易同步和区块同步：

交易同步

客户端通过RPC向指定群组交易池提交新交易时，会唤醒相应群组同步模块的交易同步线程，该线程将所有新收到的交易广播到其他群组节点，其他群组节点将最新交易插入到交易池，保证每个群组节点拥有全量的交易。

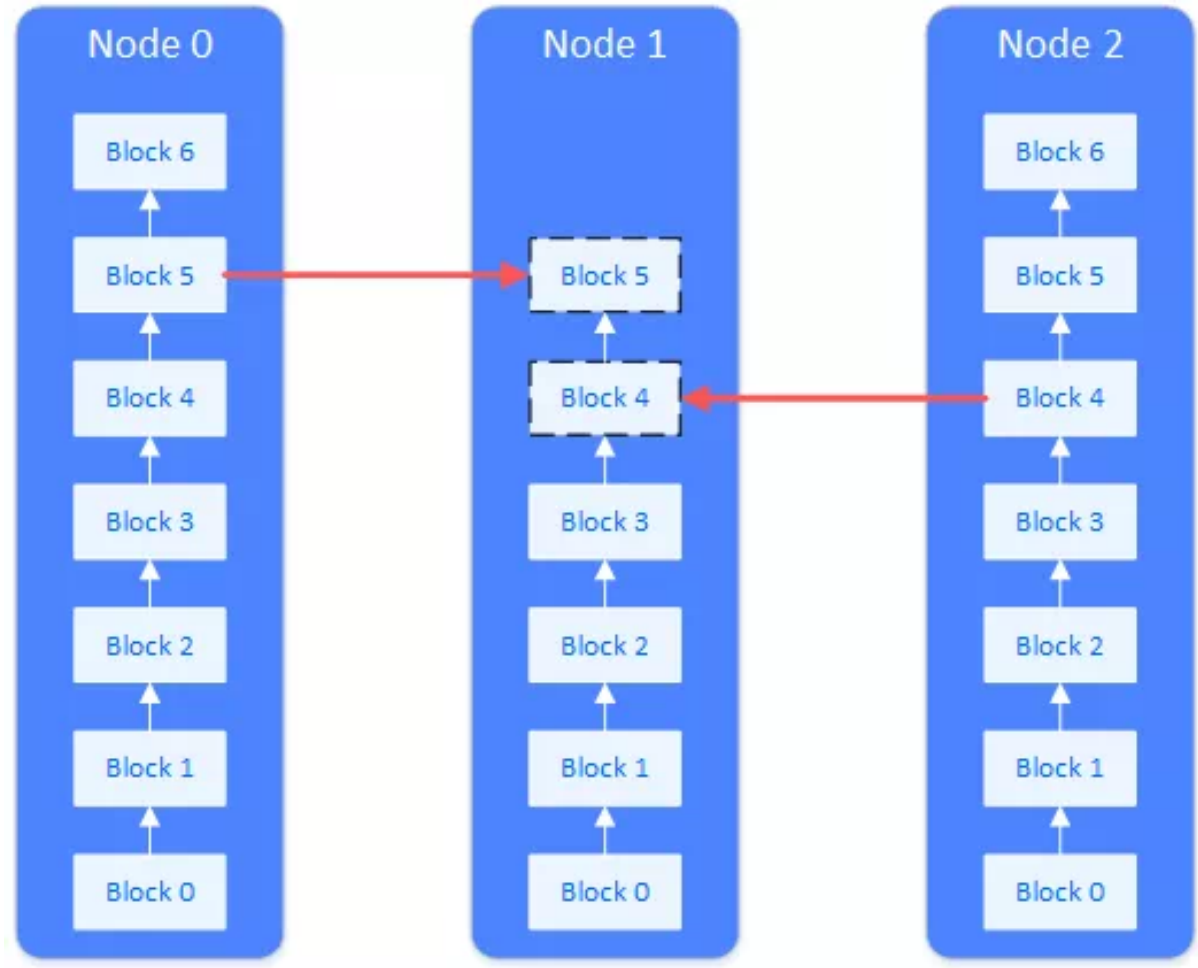
如下图，客户端将交易tx_j发送到group1, tx_i发送到group2后，交易同步线程会将tx_i广播到所有群组节点的group1，将tx_j广播到所有群组节点的group2。



区块同步

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。

以三节点区块链系统为例，区块同步流程如下图：



(1) Node0, Node1和Node2的区块同步线程定期广播最新区块高度信息；

(2) Node1收到Node0和Node2的最新区块高度后，发现自身块高3低于Node0和Node2最新快高6；

(3)本着负载均衡的原则， Node1向Node2请求第4个区块，向Node0请求区块5和区块6；

(4) Node0和Node2接收到Node1的区块请求后，分别将第{5,6}和第{4}个区块返回给Node1；

(5) Node1按照区块高度执行第4、5、6个区块，并将最新区块按次序提交到底层存储。

下篇预告：群组架构的使用教程

下一篇文章，我会以搭建群组区块链为例，向大家提供群组架构的实操课程，敬请持续锁定FISCO BCOS开源社区。

FISCO BCOS 2.0使用教程篇1： 群组架构实操演练

作者：陈宇杰 | FISCO BCOS 核心开发者

如果说，上篇是带你潜入团队的大脑，看清群组架构诞生的由来和架构设计（还没看的伙伴可以点标题直接进入：群组架构的设计。

那么下篇，即本文，视线则聚焦到团队的双手，看看群组架构如何在跳动的十指中轻快飞舞。

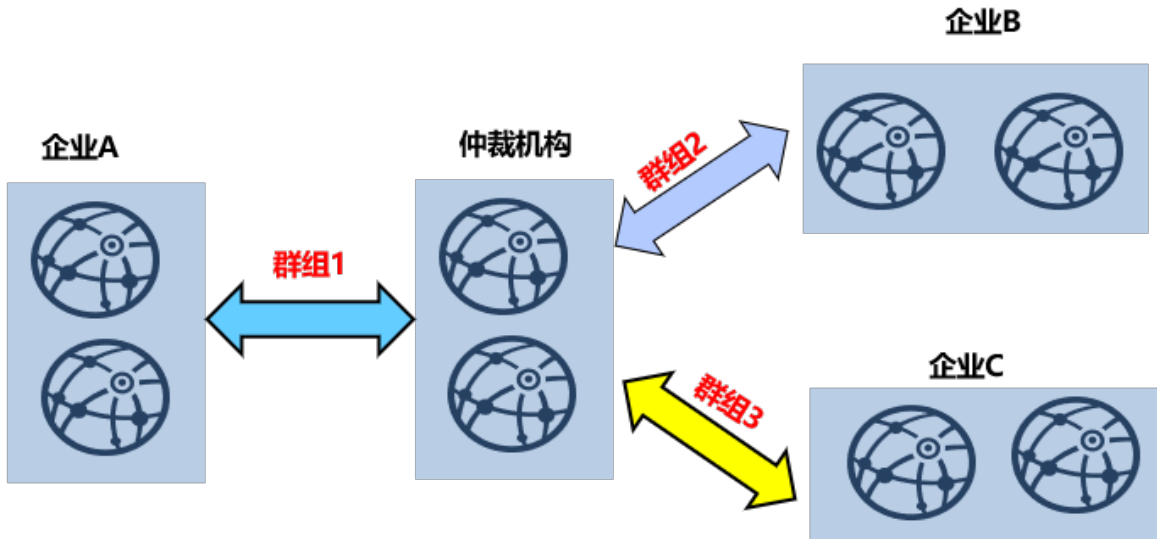
本文是高能实操攻略,全程硬核干货，我将**以搭建仲裁链为例**，并演示如何向该链发送交易。

课程知识点：

- 使用build_chain创建多群组区块链安装包
- 如何启动区块链节点、查看节点共识状态和出块状态
- 搭建控制台，向多个群组部署合约

仲裁链组织结构

下图是一个仲裁链示例：



企业A、企业B和企业C分别和仲裁机构合作，采用区块链搭建仲裁服务。在群组架构下，搭链方式为：**仲裁机构**配置两个节点，分别加入三个群组；**企业A**配置两个节点，加入群组1；**企业B**配置两个节点，加入群组2；**企业C**配置两个节点，加入群组3。

仲裁链组网详情

上节介绍了仲裁链组织结构，这里在一台机器的环境下模拟仲裁链组网环境。仿真的组网环境如下：

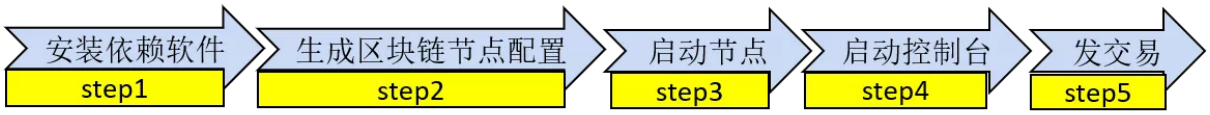
- **仲裁机构**：包括两个节点，节点IP均为127.0.0.1，同时属于群组1，群组2和群组3
- **企业A**：包括两个节点，节点IP均为127.0.0.1，仅属于群组1
- **企业B**：包括两个节点，节点IP均为127.0.0.1，仅属于群组2
- **企业C**：包括两个节点，节点IP均为127.0.0.1，仅属于群组3

温馨提示：

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。本例中仲裁机构节点归属于所有群组，负载较高，建议单独部署于性能较好的机器。

仲裁链搭建关键流程

如下图所示，使用FISCO BCOS 2.0快速建链脚本搭建仲裁链(以及所有其他区块链系统)主要包括五个步骤:



- step1：安装依赖软件，主要是openssl和build_chain.sh脚本
- step2: 使用build_chain.sh生成区块链节点配置
- step3: 启动所有机构区块链节点
- step4: 启动控制台
- step5: 使用控制台发送交易

下面我将就这五个步骤详细叙述构建仲裁链的关键流程。

安装依赖软件

搭建FISCO BCOS 2.0区块链节点需要准备如下依赖软件：

- openssl：FISCO BCOS 2.0的网络协议依赖openssl
- build_chain.sh脚本：主要用于构建区块链节点配置，可从https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/manual/build_chain.sh下载

生成区块链节点配置

FISCO BCOS 2.0提供的build_chain.sh可快速生成区块链节点配置，按照【仲裁链组网详情】介绍的节点组织结构，先生成区块链配置文件ip_list:

```
#ip_list文件内容格式：[ip]:[节点数] [机构名] [所属群组列表]
$ cat > ipconf << EOF
127.0.0.1:2 arbitrator 1,2,3
127.0.0.1:2 agencyA 1
127.0.0.1:2 agencyB 2
127.0.0.1:2 agencyC 3
EOF
```

调用build_chain.sh脚本构建仿真的本机仲裁链：

```
$ bash build_chain.sh -f ipconf -p 30300,20200,8545
```

区块链节点配置成功后，会看到[INFO] All completed.的输出。

启动节点

生成区块链节点后，需要启动所有节点，节点提供start_all.sh和stop_all.sh脚本启动和停止节点。

```
# 启动节点
$ bash start_all.sh

# 查看节点进程
$ ps aux | grep fisco-bcos
```

不发交易时，共识正常的节点会输出+++日志，使用tail -f node*/log/* | grep “++”查看各节点是否共识正常。

启动控制台

控制台是用户与FISCO BCOS 2.0区块链节点交互的重要工具，实现查询区块链状态、部署调用合约等功能，能够快速获取用户到所需要信息。

启动控制台前需获取并配置控制台：

- **获取控制台**：从https://github.com/FISCO-BCOS/console/releases/download/v1.0.0/console.tar.gz下载控制台
- **配置控制台：**主要拷贝证书、配置conf/applicationContext.xml所连接节点的IP和端口信息，控制台关键配置如下：

```
    <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" />  群组ID
                    <property name="connectionsStr">
                        <list>
                            <value>127.0.0.1:20200</value>
                            <value>127.0.0.1:20201</value>
                        </list>
                    </property> 控制台所连群组区块节点的IP和channel_port
                </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" />  群组ID
        <property name="allChannelConnections" ref="groupChannelConnectionsConfig"></property>
    </bean>

</beans>
```

当然，控制台也支持连接多个群组，并提供了switch命令来切换群组，连接多个群组时，需要在groupChannelConnectionsConfig bean id中配置多个连接，分别连接到对应群组的区块链节点。

注：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。

使用start.sh脚本启动控制台，控制台启动成功会输出如下界面：

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

=====================================================================================
>
```

向群组发交易

控制台提供了deploy HelloWorld指令向节点发交易，发完交易后，区块链节点块高会增加

```
# ... 向group1发交易...
$ [group:1]> deploy HelloWorld
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
[group:2]deploy Helloworld
...
```

总结

本文介绍了搭建仲裁链的关键过程，FISCO BCOS 2.0的操作文档step by step介绍了如何部署多群组区块链，详细流程可以参考https://fisco-bcos-documentation.readthedocs.io/zh_CN/release-2.0/docs/tutorial/group_use_cases.html

分布式存储：支持海量数据存储

FISCO BCOS 2.0原理解析篇2： 分布式存储架构设计

作者：莫楠 | FISCO BCOS 高级架构师

FISCO BCOS 2.0新增对分布式数据存储的支持，克服了本地化数据存储的诸多限制。

在FISCO BCOS 1.0中，节点采用MPT数据结构，通过LevelDB将数据存储于本地，这种模式受限于本地磁盘大小，当业务量增大时数据会急剧膨胀，要进行数据迁移也非常复杂，给数据存储带来较大的成本和维护难度。

为了突破性能的瓶颈，我们在FISCO BCOS 2.0中，对底层的存储进行了重新设计，实现了分布式存储，使用不同于MPT的方式来实现追溯，带来了性能上的提升。

先夸夸分布式存储方案的优点：

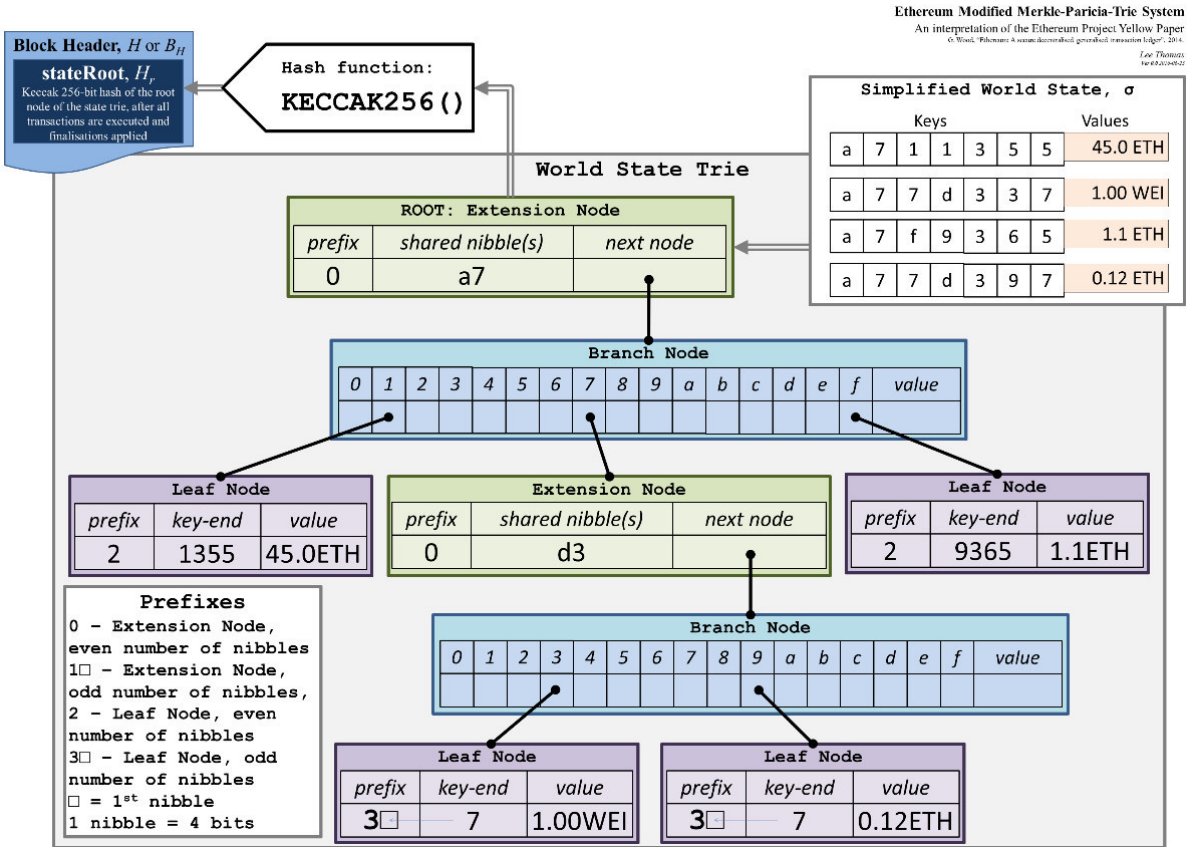
- 支持多种存储引擎，选用高可用的分布式存储系统，可以支持数据简便快速地扩容；
- 将计算和数据隔离，节点故障不会导致数据异常；
- 数据在远端存储，数据可以在更安全的隔离区存储，这在很多场景中非常有意义；
- 分布式存储不仅支持Key-Value形式，还支持SQL方式，使得业务开发更为简便；
- 世界状态的存储从原来的MPT存储结构转为分布式存储，避免了世界状态急剧膨胀导致性能下降的问题；
- 优化了数据存储的结构，更节约存储空间。

从MPT存储到分布式存储

MPT存储

MPT(Merkle Paricia Trie)，来自以太坊，对外接口为Key-Value，使用前缀树来存储数据，是FISCO BCOS 1.0的存储模式。

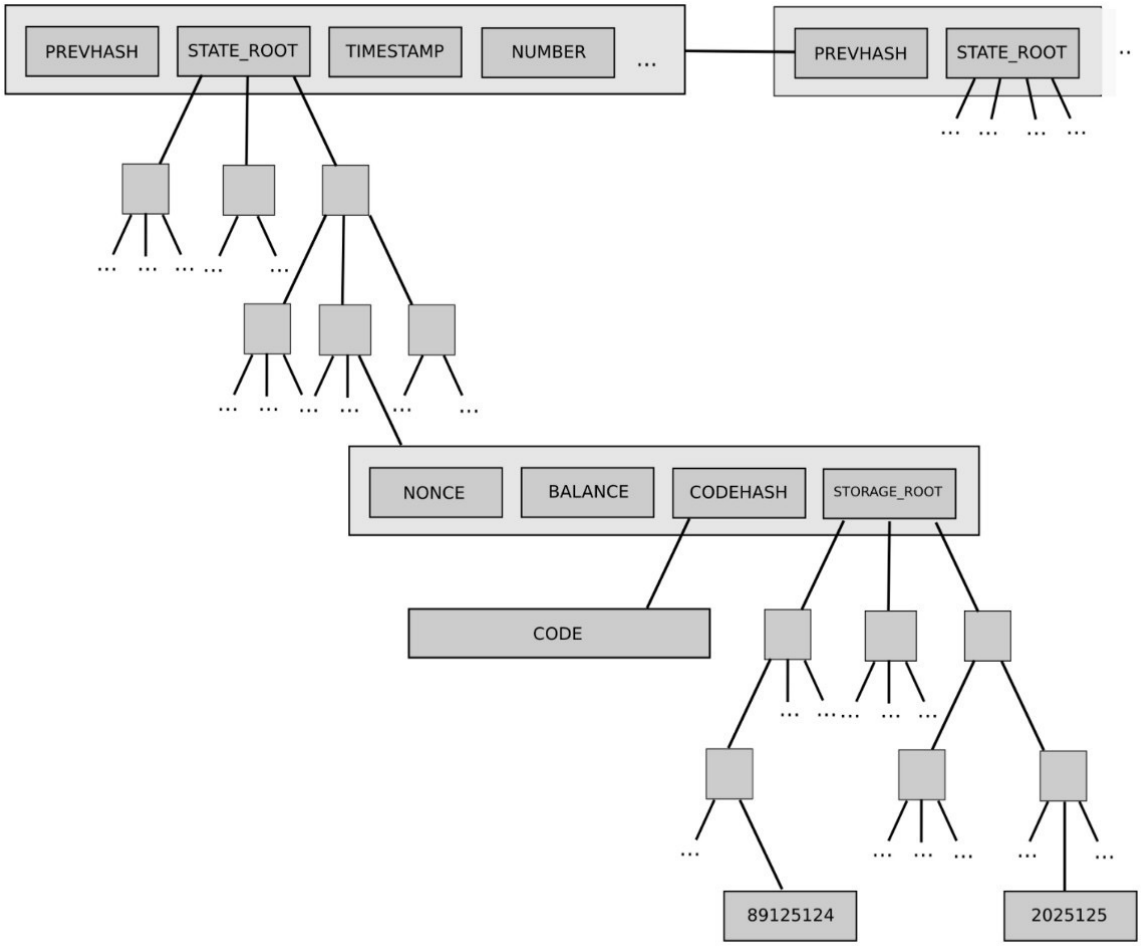
MPT是前缀树结构，树中的每个叶子节点允许有最多16个子叶子节点，叶子节点有一个HASH字段，由该叶子的所有子叶子节点HASH运算得出。树根有唯一的HASH值，标识整棵树的HASH。



图片来自以太坊白皮书

以太坊的全局状态数据，保存在MPT树中，状态数据由账号组成。账号在MPT中是一个叶子节点，账号数据包括Nonce、Balance、CodeHash和StorageRoot。任意一个账号字段发生变化时，会导致该账号所在的叶子的HASH发生变化，从该叶子直到顶部的所有叶子的HASH都会变化，最后导致顶部的StateRoot变化。

由此可见，任何账户的任意字段变化，都会导致StateRoot的变化，StateRoot能唯一标识以太坊的全局状态。

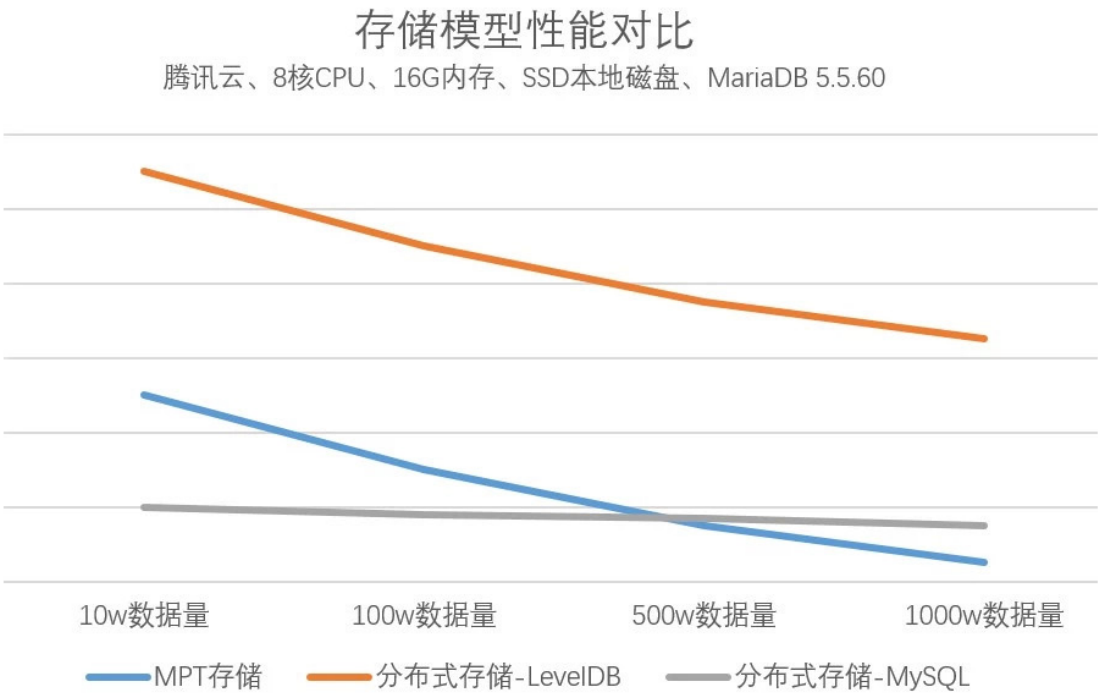


图片来自以太坊白皮书

MPT可以实现轻客户端和数据追溯，通过StateRoot可以查询到区块的状态。MPT带来了大量HASH的计算，打散了底层数据存储的连续性。在性能方面，MPT State存在着天然的劣势。可以说，MPT State追求极致的可证明性和可追溯性，对性能和可扩展性做了一定的妥协。

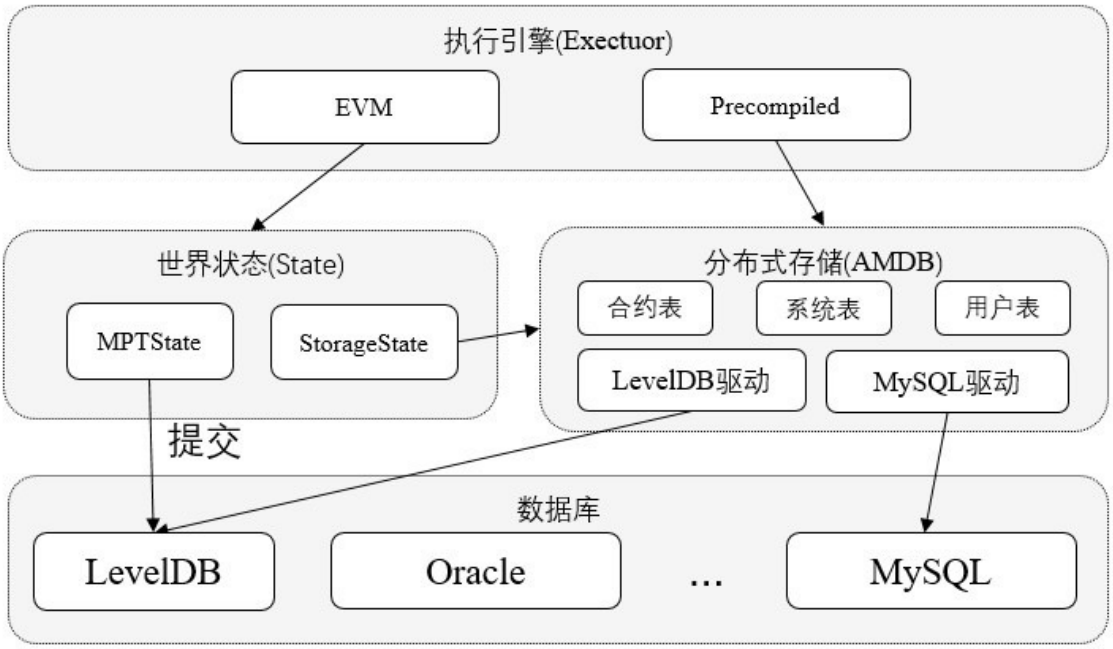
分布式存储

FISCO BCOS 2.0在保持存储接口兼容性的同时，引入了高扩展性、高吞吐量、高可用、高性能的分布式存储。**分布式存储(Advanced Mass Database，AMDB)**：重新抽象了区块链的底层存储模型，实现了类SQL的抽象存储接口，支持多种后端数据库，包括KV数据库和关系型数据库。 引入了分布式存储后，数据读写请求不经过MPT，直接访问存储，结合缓存机制，存储性能相比基于MPT的存储有大幅提升。MPT数据结构仍然保留，仅做为可选方案。



分布式存储支持MySQL等关系型数据库，支持MySQL集群、分库分表等平行扩展方式，理论上存储容量无限。

分布式存储架构



State层（State）

抽象了智能合约的存储访问接口，由EVM调用，分为StorageState和MPTState。StorageState为分布式存储的适配层，MPTState为老的MPT适配层，FISCO BCOS默认使用StorageState。

分布式存储层（Table）

抽象了分布式存储的类SQL接口，由State层和Precompiled调用。分布式存储层抽象了存储的增删改查接口，把区块链的核心数据分类存储到不同的表中。

驱动层（Storage）

实现具体的数据库访问逻辑，包括LevelDB和MySQL。

分布式存储名词解释

Table

存储表中的所有数据。Table中存储分布式存储主key到对应Entries的映射，可以基于分布式存储主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。分布式存储的主Key与Mysql中的主key不同，分布式存储主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拥有自己的分布式存储主key，不同Entry允许拥有相同的分布式存储主key。

Condition

Table中的“删改查”接口可传入条件，支持“等于”“大于”“小于”等过滤逻辑，接口根据条件对数据进行筛选后进行相应操作，返回结果数据。如果条件为空，则不做任何筛选。

举例

以某公司员工领用物资登记表为例，解释上述名词

Name*	item_id	item_name
Alice	1001001	laptop
Alice	1001002	screen
Bob	1002001	macbook
Chris	1003001	PC

- 表中Name是分布式存储主key。
- 表中的每一行为一个Entry。一共有4个Entry，每个Entry有三个字段。
- 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，查接口需要指定分布式存储主key和条件，设置查询的分布式存储主key为Alice，条件为price > 40，会查询出Entry1。

分布式存储表分类

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。

表名	存储数据说明
_sys_tables_	存储所有表的结构，以表名为主键
_sys_consensus_	存储共识节点和观察节点的列表
_sys_table_access_	存储每个表的具有写权限的外部账户地址
_sys_cns_	存储CNS映射关系
_sys_config_	存储需要共识的群组配置项
_sys_current_state_	存储最新的状态
_sys_tx_hash_2_block_	存储交易hash到区块号的映射
_sys_number_2_hash_	存储区块号到区块头hash的16进制表示的映射
_sys_hash_2_block_	存储hash到序列化的区块数据
_sys_block_2_nonces_	存储区块中交易的nonces

用户表

用户CRUD合约所创建的表，以_user_为表名，底层自动添加_user_前缀。

StorageState账户表

contract_data+Address+_作为表名。表中存储外部账户相关信息。表结构如下：

key*	value
balance	账户余额（可选）
nonce	账户计数
code	智能合约代码（可选）
codeHash	智能合约代码HASH（可选）
alive	账户是否有效

总结

FISCO BCOS发布至今，历经大量真实业务实践。分布式存储在持续改进的过程中，总结出适合于金融业务、高性能、高可用性和高可扩展的存储模型，架构愈发稳定成熟，未来分布式存储将继续作为区块链系统的基石，支持区块链系统的发展。

下篇文章，我会提供分布式存储的体验流程，敬请持续锁定FISCO BCOS开源社区。

系列精选

FISCO BCOS 2.0发布：（附新特性解读）

原理解析

群组架构的设计：使企业间建立多方协作的商业关系像拉群聊天一样简便。

使用教程

群组架构实操演练：以搭建仲裁链为例，并演示如何向该链发送交易。

FISCO BCOS 2.0使用教程： 分布式存储体验

作者：莫楠 | FISCO BCOS 高级架构师

《分布式存储架构设计》一文发布后，社区成员对技术内核及使用非常关注。团队和社区热心小伙伴、业内专家针对分布式存储，进行了一箩筐地讨论。在此，跟大家分享交流心得，或有助你更好地理解和使用分布式存储：

- FISCO BCOS 2.0的分布式存储采用库表风格，CRUD操作符合业务习惯。
- 不用合约存储变量模式，解构了合约和数据的内嵌式耦合，合约升级更容易。
- 存储访问引擎逻辑和数据结构更直观，容易适配各种存储引擎，扩展空间大。

- 数据本身行列式存储，没有MPT树那般盘根错节的关系，更容易打快照和切割迁移。
- 表加主键的结构索引数据，存取效率高，并发访问更容易。
- 存储开销更少，容量模型和交易数、状态数线性相关，更容易预测业务容量，对海量服务非常有意义。
- 细节方面，弱化状态MPT，但保留了交易和回执MPT，依旧可支持轻客户端，采用过程证明和存在证明，而不依赖易变的状态，不影响实现跨链。
- 状态由增量HASH检验，每块交易产生的状态集会全网严格检验以保证一致性。
- 一开始是面向SQL类型构建的，可支持MySQL和Oracle等引擎，然后适配到NoSQL类型，如LevelDB等。后续还会适配更多高速和海量存储引擎，在【单次io延迟/并发效率/容量扩展】这个三角关系中，探索出最优解。

分布式存储虽说是个大工程（团队几个快枪手撸了小一年才敢拿出来见人），但使用非常简单，本文就讲讲分布式存储的体验流程。初步接触用户，建议先从上篇入手（点标题可直接跳转）→ 分布式存储架构设计

配置分布式存储

分布式存储支持多种存储引擎，根据业务需求和部署环境灵活选择，可以配置为不同的存储引擎。

区块链的区块、交易等基础数据采用库表结构保存，状态数据的存储方式可配为库表结构或MPT，满足不同场景的需求。

分布式存储的配置项位于群组的配置文件中，各个群组可以使用单独的存储策略，群组配置文件位于区块链节点中名为conf/group.[群组号].genesis的路径下，如group.1.genesis，一旦群组启动，该群组的分布式存储的相关配置不能再改变。

分布式存储配置项示例如下：

[storage]

type=LevelDB：分布式存储的DB引擎类型，支持"LevelDB"和"External"（rc2版本）

[state]

type=storage：state类型，目前支持storage state和MPT state，默认为storage state

推荐使用 storage state， 除非必须使用MPT来追溯全局历史状态，不建议使用MPT State。

使用CRUD智能合约开发

分布式存储提供了专用的CRUD接口，支持合约直接访问底层的存储表。

访问CRUD需要引用分布式存储专用的智能合约Table.sol接口，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。

引用Table.sol

```
import "./Table.sol";
```

Table.sol的接口包括：

- createTable //创建表
- select(string, Condition) //查询数据
- insert(string, Entry) //插入数据
- update(string, Entry, Condition) //更新数据
- remove(string, Condition) //删除数据

每个接口的用法如下：

创建表

```
// TableFactory的地址固定为0x1001
TableFactory tf = TableFactory(0x1001);

// 创建t_test表，表的key_field为name，value_field为item_id,item_name
// key_field表示分布式存储主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");
```

查询数据

```
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);
```

插入数据

```
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);
```

更新数据

```
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);
```

删除数据

```
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);
```

PS

存储架构的优化是个基础工程，也是个大工程。实现的转变，其实是架构世界观的一种进化，影响会

比看到的功能点更深远。此二文所讲述的，仅是分布式存储的冰山一角。更多原理和使用案例，请参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/smart_contract.html

系列精选

FISCO BCOS 2.0发布：（附新特性解读）

原理解析

群组架构的设计：使企业间建立多方协作的商业关系像拉群聊天一样简便。

使用教程

群组架构实操演练：以搭建仲裁链为例，并演示如何向该链发送交易。

并行计算：支持块内交易并行执行

区块链性能腾飞：基于DAG的并行交易执行引擎

作者：李陈希 | FISCO BCOS 核心开发者

在区块链世界中，交易是组成事务的基本单元。交易吞吐量很大程度上能限制或拓宽区块链业务的适用场景，愈高的吞吐量，意味着区块链能够支持愈广的适用范围和愈大的用户规模。当前，反映交易吞吐量的TPS（Transaction per Second，每秒交易数量）是评估性能的热点指标。为了提高TPS，业界提出了层出不穷的优化方案，殊途同归，各种优化手段的最终聚焦点，均是尽可能提高交易的并行处理能力，降低交易全流程的处理时间。

在多核处理器架构已经成为主流的今天，利用并行化技术充分挖掘CPU潜力是行之有效的方案。FISCO BCOS 2.0 中设计了一种基于DAG模型的并行交易执行器（PTE，Parallel Transaction Executor）。

PTE能充分发挥多核处理器优势，使区块中的交易能够尽可能并行执行；同时对用户提供简单友好的编程接口，使用户不必关心繁琐的并行实现细节。基准测试程序的实验结果表明：相较于传统的串行交易执行方案，理想状况下4核处理器上运行的PTE能够实现约200%~300%的性能提升，且计算方面的提升跟核数成正比，核数越多性能越高。

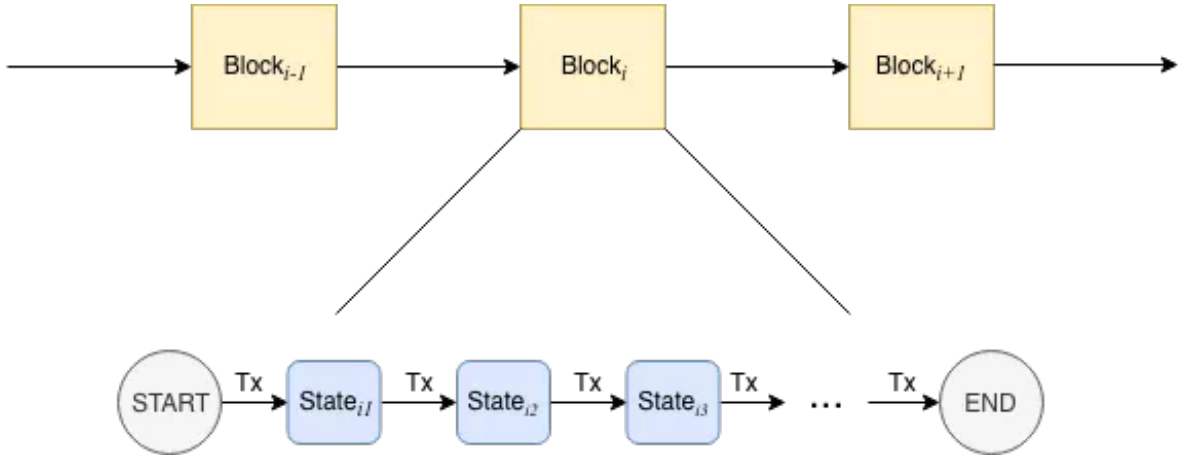
PTE为助力FISCO BCOS性能腾飞奠定了坚实基础，本文将全面介绍PTE的设计思路及实现方案，主要包括以下内容：

- **背景**：传统方案的性能瓶颈与DAG并行模型的介绍
- **设计思路**：PTE应用到FISCO BCOS中时遇到的问题以及解决方案
- **架构设计**：应用PTE后FISCO BCOS的架构及核心流程
- **核心算法**：介绍主要用到的数据结构与主要算法
- **性能测评**：分别给出PTE的性能与可扩展性测试结果

背景

FISCO BCOS交易处理模块可以被抽象为一个基于交易的状态机。在FISCO BCOS中，『状态』即是指区块链中所有账户的状态，而『基于交易』即是指FISCO BCOS将交易作为状态迁移函数，并根据交易内容从旧的状态更新为新的状态。FISCO BCOS从创世块状态开始，不断收集网络上发生的交易并打包为区块，并在所有参与共识的节点间执行区块中的交易。当一个区块内的交易在多个共识节点上执行完成且状态一致，则我们称在该块上达成了共识，并将该区块永久记录在区块链中。

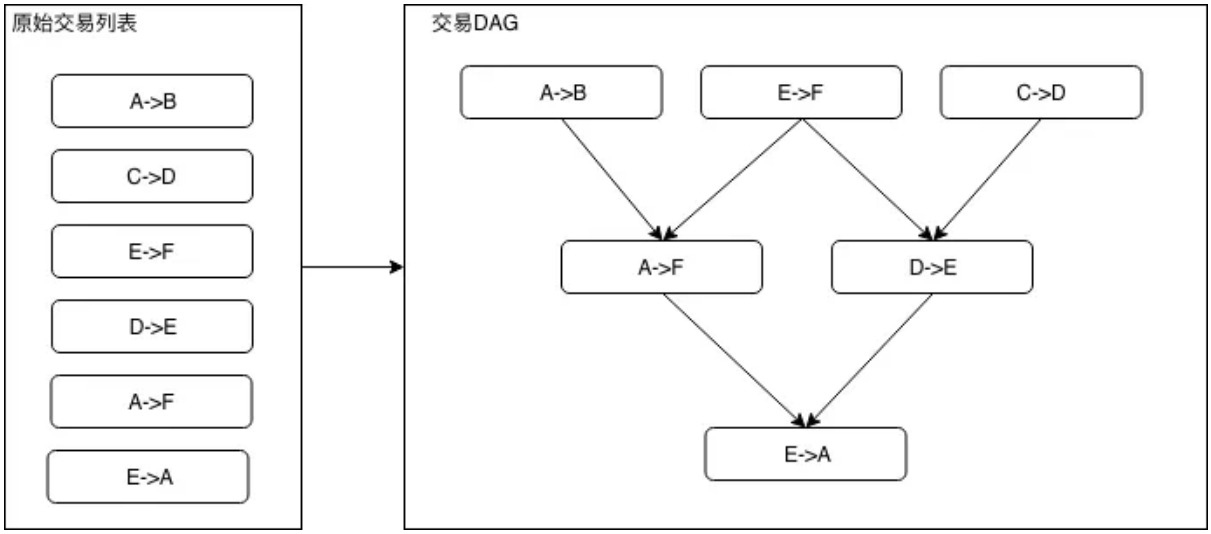
从上述区块链的打包→共识→存储过程中可以看到，执行区块中的所有交易是区块上链的必经之路。传统交易执行方案是：执行单元从待共识的区块逐条读出交易，执行完每一笔交易后，状态机都会迁移至下一个状态，直到所有交易都被串行执行完成，如下图所示：



显而易见，这种交易执行方式对性能并不友好。即使两笔交易没有交集，也只能按照先后顺序依次执行。就交易间的关系而言，既然一维的『线』结构有这般痛点，那何不把目光投向二维的『图』结构呢?

在实际应用中，根据每笔交易执行时需要使用的互斥资源（互斥意味着对资源的排他性使用，比如在上述转账问题互斥资源中，指的就是各个账户的余额状态），我们可以组织出一张交易依赖关系图，为防止交易依赖关系在图中成环，我们可以规定交易列表中牵涉到相同的互斥资源，且排序靠后的交易，必须等待靠前的交易完成后才被执行，由此得到的输出便是一张反映交易依赖关系的有向无环图，即交易DAG。

如下图所示，左侧的6笔转账交易可以组织为右侧的DAG形式:



在交易DAG中，入度为0的交易是没有任何依赖项、可以被立即投入运行的就绪交易。当就绪交易的数量大于1时，就绪交易可以被分散至多个CPU核心上并行执行。当一笔交易执行完，依赖于该交易的所有交易的入度减1，随着交易不断被执行，就绪交易也源源不断被产生。在极限情况下，假如构造出的交易DAG层数为1（即所有交易均是没有依赖项的独立交易），则交易整体执行速度的提升倍数将直接取决于处理器的核心数量n，此时若n大于区块内的交易数，则区块内所有交易的执行时间与单笔交易执行的时间相同。

理论上拥有如此让人无法拒绝的优美特性的交易DAG模型，该如何应用至FISCO BCOS中?

设计思路

要应用交易DAG模型，我们面临的首要问题便是：对于同一个区块，如何确保所有节点执行完后能够达到同一状态，这是一个关乎到区块链能否正常出块的关键问题。

FISCO BCOS采用验证(state root, transaction root, receipt root)三元组是否相等的方式，来判断状态是否达成一致。transaction root是根据区块内的所有交易算出的一个哈希值，只要所有共识节点处理的区

块数据相同，则transaction root必定相同，这点比较容易保证，因此重点在于如何保证交易执行后生成的state和receipt root也相同。

众所周知，对于在不同CPU核心上并行执行的指令，指令间的执行顺序无法提前预测，并行执行的交易也存在同样情况。在传统的交易执行方案中，每执行一笔交易，state root便发生一次变迁，同时将变迁后的state root写入交易回执中，所有交易执行完后，最终的state root就代表了当前区块链的状态，同时再根据所有交易回执计算出一个receipt root。

可以看出，在传统的执行方案中，state root扮演着一个类似全局共享变量的角色。当交易被并行且乱序执行后，传统计算state root的方式显然不再适用，这是因为在不同的机器上，交易的执行顺序一般不同，此时无法保证最后的state root能够一致，同理，receipt root也无法保证一致。

在FISCO BCOS中，我们采用的解决方案是先执行交易，将每笔交易对状态的改变历史记录下来，待所有交易执行完后，再根据这些历史记录再算出一个state root，同时，交易回执中的state root，也全部变为所有交易执行完后最终的state root，由此就可以保证即使并行执行交易，最后共识节点仍然能够达成一致。

搞定状态问题后，下一个问题便是：如何判断两笔交易之间是否存在依赖关系?

若两笔交易本来无依赖关系但被判定为有，则会导致不必要的性能损失；反之，如果这两笔交易会改写同一个账户的状态却被并行执行了，则该账户最后的状态可能是不确定的。因此，依赖关系的判定是影响性能甚至能决定区块链能否正常工作的重要问题。

在简单的转账交易中，我们可以根据转账的发送者和接受者的地址，来判断两笔交易是否有依赖关系，比如如下3笔转账交易：A→B，C→D，D→E，可以很容易看出，交易D→E依赖于交易C→D的结果，但是交易A→B和其他两笔交易没有什么关系，因此可以并行执行。

这种分析在只支持简单转账的区块链中是正确的，但是一旦放到图灵完备、运行智能合约的区块链中，则可能不那么准确，因为我们无法准确知道用户编写的转账合约中到底有什么操作，可能出现的情况是：A->B的交易看似与C、D的账户状态无关，但是在用户的底层实现中，A是特殊账户，通过A账户每转出每一笔钱必须要先从C账户中扣除一定手续费。在这种场景下，3笔交易均有关联，则它们之间无法使用并行的方式执行，若还按照先前的依赖分析方法对交易进行划分，则必定会掉坑。

我们能否做到根据用户的合约内容自动推导出交易中实际存在哪些依赖项？答案是不太靠谱。我们很难去追踪用户合约中到底操作了什么数据，即使做到也需要花费不小的成本，这和我们优化性能的目标相去甚远。

综上，我们决定在FISCO BCOS中，将交易依赖关系的指定工作交给更熟悉合约内容的开发者。具体地说，交易依赖的互斥资源可以由一组字符串表示，FISCO BCOS暴露接口给到开发者，开发者以字符串形式定义交易依赖的资源，告知链上执行器，执行器则会根据开发者指定的交易依赖项，自动将区块中的所有交易排列为交易DAG。比如在简单转账合约中，开发者仅需指定每笔转账交易的依赖项是｛发送者地址+接收者地址｝。进一步地，如开发者在转账逻辑中引入了另一个第三方地址，那么依赖项就需要定义为｛发送者地址+接受者地址+第三方地址｝了。

这种方式实现起来较为直观简单，也比较通用，适用于所有智能合约，但也相应增加了开发者肩上的责任，开发者在指定交易依赖项时必须十分小心，如果依赖项没有写正确，后果无法预料。指定依赖项的相关接口会在后续文章中给出使用教程，本文暂且假定所有谈论到的交易依赖项都是明确无误的。

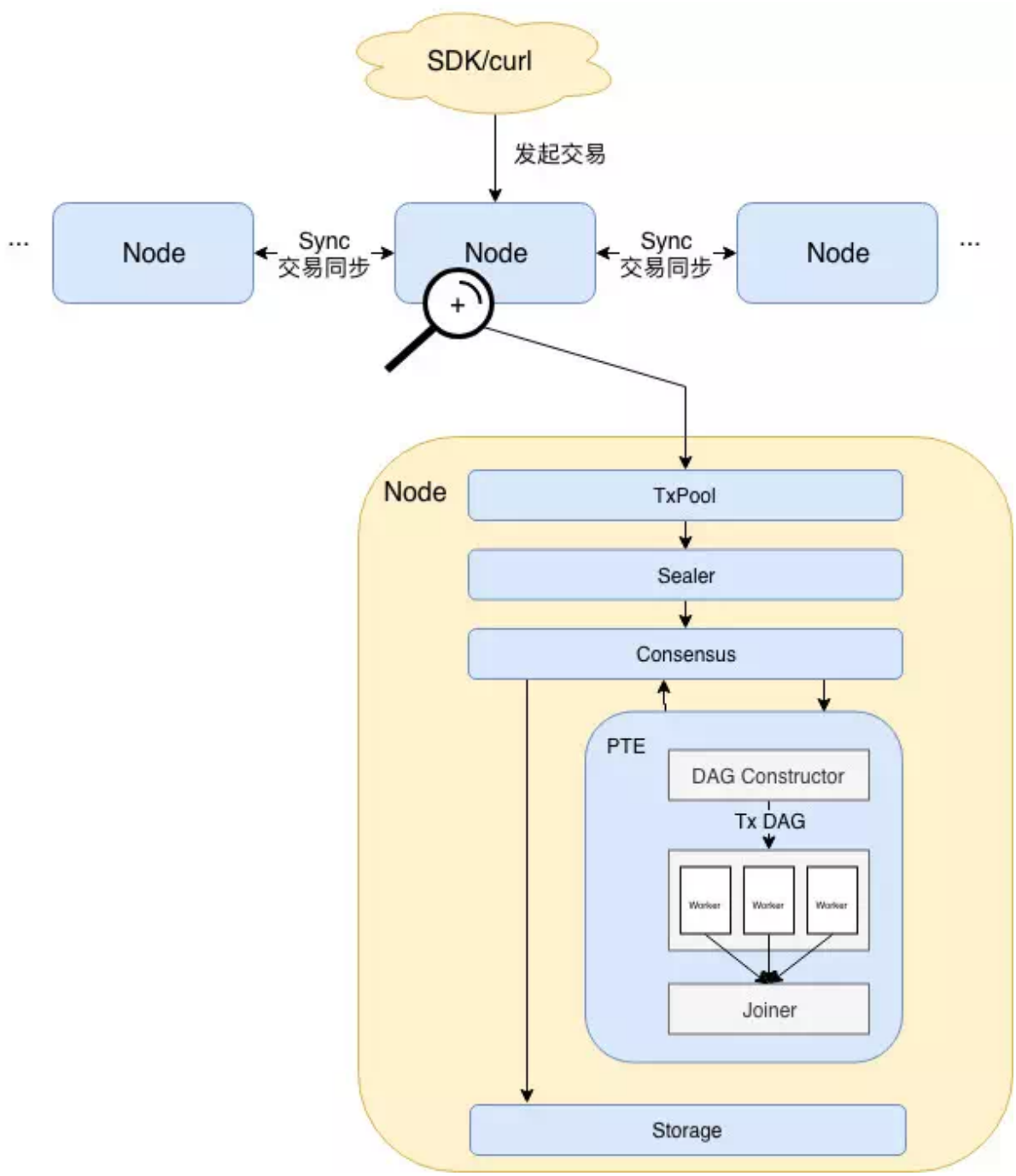
解决完上面两个比较重要的问题后，还剩下一些较为细节的工程问题：比如并行交易能否和非并行交易混合到一起执行？怎么保证资源字符串的全局唯一性?

答案也不复杂，前者可通过将非并行交易作为屏障（barrier）插入到交易DAG中——即我们认为，它即依赖于它的所有前序交易，同时又被它的所有后序交易依赖——来实现；后者可以通过在开发者指定的交易依赖项中，加入标识合约的特殊标志位解决。由于这些问题并不影响PTE的根本设计，本文暂不展开。

万事俱备，带着全新交易执行引擎PTE的FISCO BCOS已经呼之欲出。

架构设计

搭载PTE的FISCO BCOS架构图：



整个架构的核心流程如下：

用户通过SDK等客户端将交易发送至节点，此处的交易既可以是可并行执行的交易，也可以是不能并行执行的交易。随后交易在节点间同步，同时拥有打包权的节点调用打包器（Sealer），从交易池（Tx Pool）中取出一定量交易并将其打包成一个区块。此后，区块被发送至共识单元（Consensus）准备进行节点间共识。

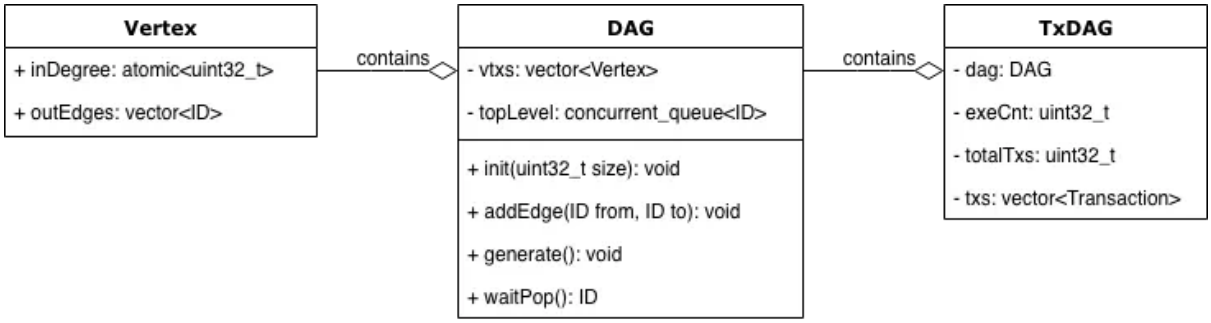
共识前需要执行区块中的交易，此处便是PTE施展威力之处。从架构图中可以看到，PTE首先按序读取区块中的交易，并输入到DAG构造器（DAG Constructor）中，DAG构造器会根据每笔交易的依赖项，构造出一个包含所有交易的交易DAG，PTE随后唤醒工作线程池，使用多个线程并行执行交易DAG。汇合器（Joiner）负责挂起主线程，直到工作线程池中所有线程将DAG执行完毕，此时Joiner负责根据各个交易对状态的修改记录计算state root及receipt root，并将执行结果返回至上层调用者。

在交易执行完成后，若各个节点状态一致，则达成共识，区块随即写入底层存储（Storage），被永久记录于区块链上。

核心算法

1.交易DAG的数据结构

交易DAG的数据结构如下图所示：



Vertex类为最基础里的类型，在交易DAG中，每一个Vertex实例都表征一笔交易。Vertex类包含：

- **inDegree**：表示该顶点的入度
- **outEdges**：用于存储该节点的出边信息，即所有出边所连顶点的ID列表

DAG类用于对DAG的顶点与边关系进行封装，并提供操作DAG的接口，其包含：

- **vtxs**：Vertex数组
- **topLevel**：包含所有入度为0的顶点的队列，由于在执行过程中topLevel会动态变化且会被多个线程访问，因此其需要一个能够支持线程安全访问的容器
- **void init(int32_t size)接口**：根据传入的size初始化一个包含相应数量顶点的DAG结构
- **addEdge(ID from, ID to)接口**：用于在顶点from和顶点to之间建立边关系，具体地说，将顶点to的ID加入顶点from的outEdges中
- **void generate()接口**：当所有的边关系录入完毕后，调用该方法以初始化topLevel成员
- **ID waitPop()接口**：从topLevel中获取一个入度为0的顶点ID

TxDAG类是DAG类更上一层的封装，是DAG与交易之间建立联系的桥梁，其包含：

- **dag**：持有的DAG类实例
- **exeCnt**：已执行过的交易总数
- **totalTxs**：交易总数
- **txs**：区块中的交易列表

2. 交易DAG的构造流程

DAG构造器在构造交易DAG时，会首先将totalTxs成员的值设置为区块中的交易总数，并依据交易总数对dag对象进行初始化，dag会在vtxs中为每笔交易生成一个位置关系一一对应的顶点实例。随后，初始化一个空的资源映射表criticalFields，并按序逐个扫描每笔交易。

对于某笔交易tx，DAG构造器会在其解析出该交易的所有依赖项，对于每个依赖项均会去criticalFields中查询，如果对于某个依赖项d，有前序交易也依赖于该依赖项，则在这两笔交易间建边，并更新criticalFields中d的映射项为tx的ID。

交易DAG构造流程的伪代码如下所示：

```
criticalFields ← map<string, ID>();
totalTxs ← txs.size();
dag.init(txs.size());
for id ← 0 to txs.size() by 1 do
```

(continues on next page)

(续上页)

```
  tx ← txs[id];
  dependencies ← 解析出tx的依赖项;
  for d in dependencies do
    if d in criticalFields then
        dag.addEdge(id, criticalFields[d]);
      end
    criticalFields[d] = id;
    end
  end
end
dag.generate();
```

3.交易DAG的执行流程

PTE在被创建时，会根据配置生成一个用于执行交易DAG的工作线程池，线程池的大小默认等于CPU的逻辑核心数，此线程池的生命周期与PTE的生命周期相同。工作线程会不断调用dag对象的waitPop方法以取出入度为0的就绪交易并执行，执行后该交易的所有后序依赖任务的入度减1，若有交易的入度被减至0，则将该交易加入到topLevel中。循环上述过程，直到交易DAG执行完毕。

交易DAG执行流程的伪代码如下所示：

```
while exeCnt < totalTxs do
  id ← dag.waitPop();
  tx ← txs[id];
  执行tx;
  exeCnt ← exeCnt + 1;
  for txID in dag.vtxs[id].outEdges do
    dag.vtxs[txID].inDegree ← dag.vtxs[txID].inDegree - 1;
    if dag.vtxs[txID].inDegree == 0 then
      dag.topLevel.push(txID)
    end
  end
end
```

性能测评

我们选用了2个基准测试程序，用以测试PTE给FISCO BCOS的性能带来了怎样的变化，它们分别是基于预编译框架实现的转账合约和基于Solidity语言编写的转账合约，两份合约代码的路径分别为：

FISCO-BCOS/libprecompiled/extension/DagTransferPrecompiled.cpp

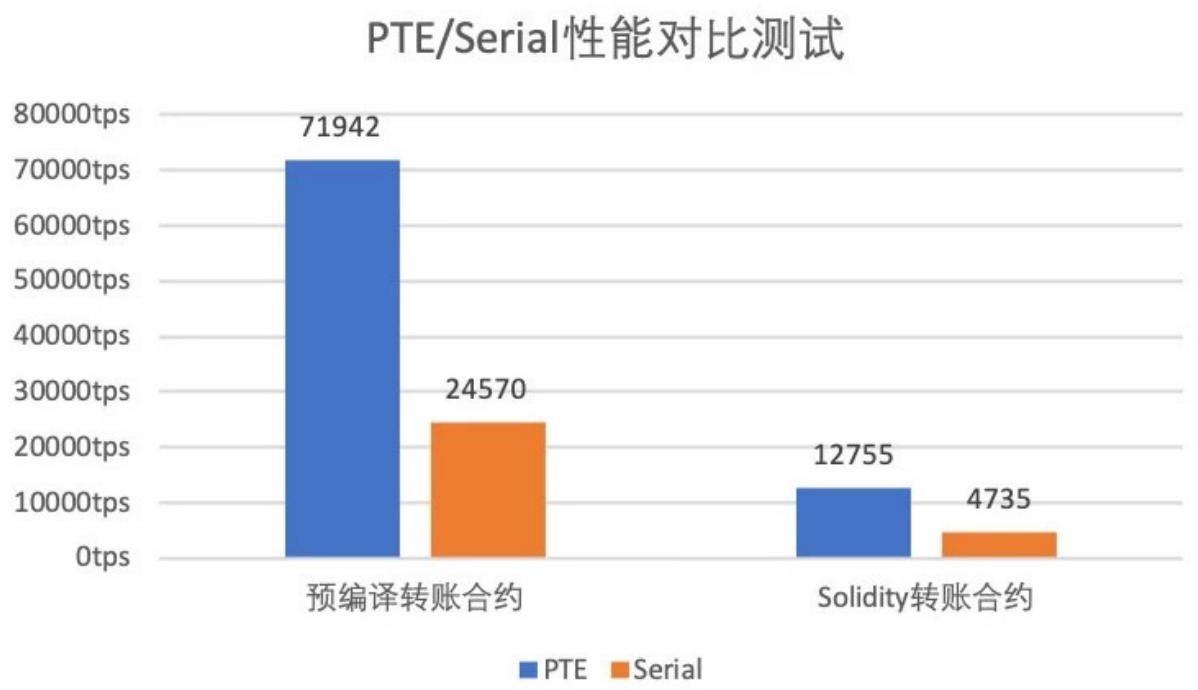
web3sdk/src/test/resources/contract/ParallelOk.sol

我们使用一条单节点链进行测试，因为我们主要关注PTE的交易处理性能，因此并不考虑网络、存储的延迟带来的影响。

测试环境的基本硬件信息如下表所示：

CPU	Intel(R) Xeon(R) Gold 61xx CPU@2.0GHz * 4
内存	8GB DRAM
硬盘	160GB 7200rpm
网络带宽	1Gbps

1.性能测试



性能测试部分，我们主要测试PTE和串行交易执行方式（Serial）在各个测试程序下的交易处理能力。可以看到，相对于串行执行方式，PTE从左至右分别实现了2.91和2.69倍的加速比。无论是对于预编译合约还是Solidity合约，PTE均有着不俗的性能表现。

2.可扩展性测试

可扩展性测试部分，我们主要测试PTE在不同CPU核心数下的交易处理能力，使用的基准测试程序是基于预编译框架实现的转账合约。可以看到，随着核数增加，PTE的交易吞吐量呈近似线性递增。但是同

时也能看到，随着核数在增加，性能增长的幅度在放缓，这是因为随着核数增加线程间调度及同步的开销也会增大。

写在最后

从列表到DAG，PTE赋予了FISCO BCOS新的进化。更高的TPS会将FISCO BCOS带至更加广阔的舞台。予以长袖，FISCO BCOS必能善舞！

让木桶没有短板，FISCO BCOS全面推进并行化改造

作者：李陈希 | FISCO BCOS 核心开发者

背景

PTE（Parallel Transaction Executor，一种基于DAG模型的并行交易执行器）的引入，使FISCO BCOS具备了并行执行交易的能力，显著提升了节点交易处理的效率。对这个阶段性结果，我们并不满足，继续深入挖掘发现，FISCO BCOS的整体TPS仍有较大提升空间。用木桶打个比方：如果参与节点的交易处理所有模块构成木桶，交易执行只是组成整个木桶的一块木板，根据短板理论，一只木桶能盛多少水取决于桶壁上最矮的那块，同理，**FISCO BCOS的性能也由速度最慢的组件决定**。尽管PTE取得了理论上极高的性能容量，但是FISCO BCOS的整体性能仍然会被其他模块较慢的交易处理速度所掣肘。**为了能够最大化利用计算资源以进一步提高交易处理能力，在FISCO BCOS中全面推进并行化改造势在必行。**

数据分析

根据并行程序设计的『分析→分解→设计→验证』四步走原则，首先需定位出系统中仍存在的性能瓶颈的精确位置，才能更深入地对任务进行分解，并设计相应的并行化策略。**使用自顶向下分析法，我们将交易处理流程分为四个模块进行性能分析**，这四个模块分别是：

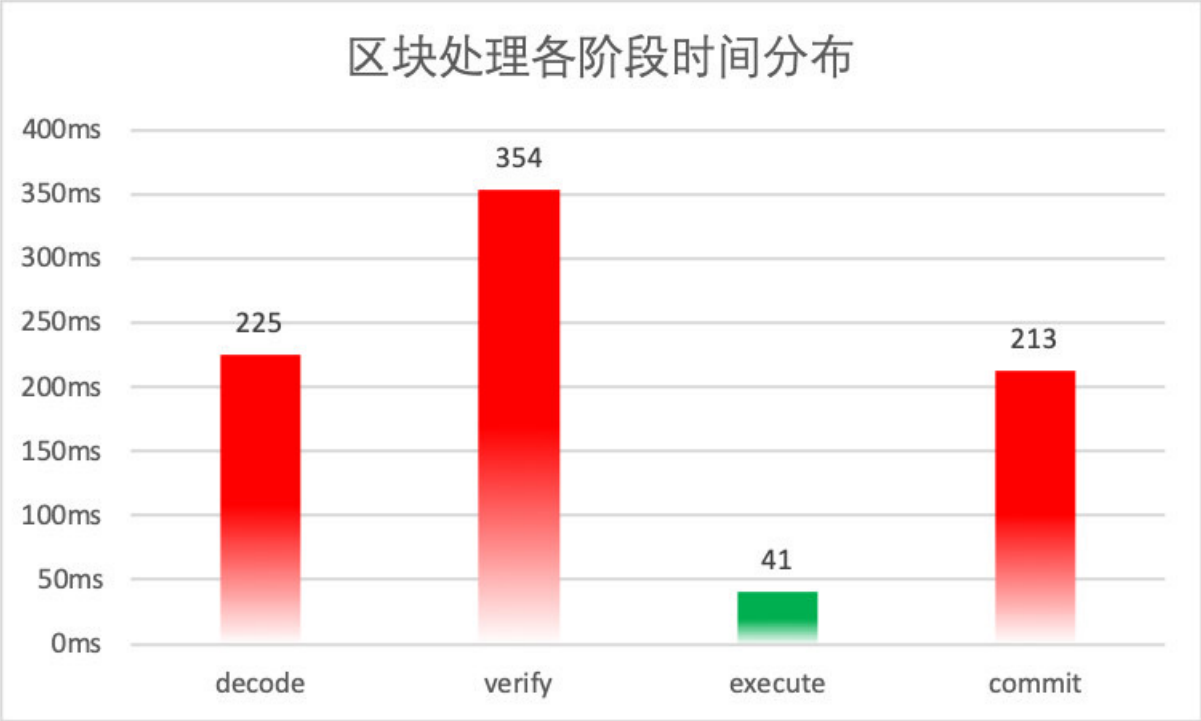
区块解码（decode）：区块在节点间共识或同步时需要从一个节点发送至另一个节点，这个过程中，区块以RLP编码的形式在网络间传输。节点收到区块编码后，需要先进行解码，将区块还原为内存中的二进制对象，然后才能做进一步处理。

交易验签（verify）：交易在发送之前由发送者进行签名，签名得到的数据可以分为(v, r, s)三部分，验签的主要工作便是在收到交易或交易执行前，从(v, r, s)数据中还原出交易发送者的公钥，以验证交易发送者的身份。

交易执行（execute）：执行区块中的所有交易，更新区块链状态。

数据落盘（commit）：区块执行完成后，需要将区块及相关数据写入磁盘中，进行持久化保存。

以包含2500笔预编译转账合约交易的区块为测试对象，在我们的测试环境中，各阶段的平均耗时分布如下图所示：



从图中可以看出，2500笔交易的执行时间已经被缩短到了50毫秒以内，可以证明PTE对FISCO BCOS交易执行阶段的优化是行之有效的。但图中也暴露出了非常明显的问题：其他阶段的用时远远高于交易执行的用时，导致交易执行带来的性能优势被严重抵消，PTE无法发挥出其应有的价值。

早在1967年，计算机体系结构领域的元老Amdahl提出的以他名字命名的定律，便已经向我们阐明了衡量处理器并行计算后效率提升能力的经验法则：

$$SpeedUp \leq \frac{W_S + W_p}{W_S + \frac{W_p}{N}}$$

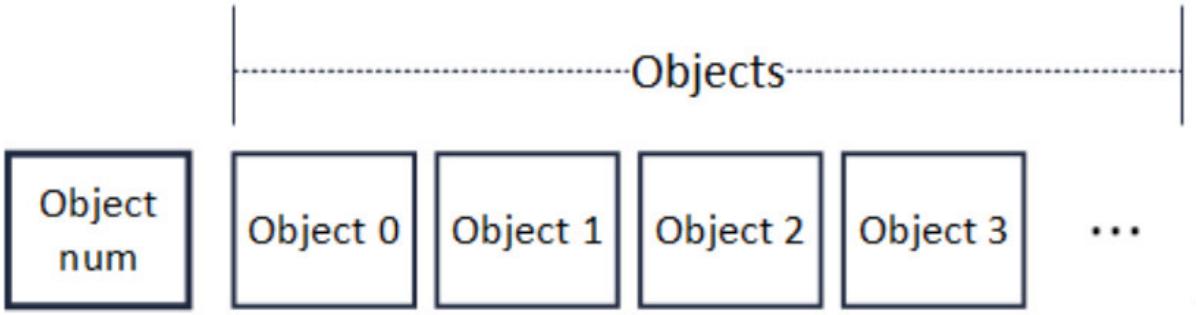
其中，SpeedUp为加速比，Ws是程序的串行分量，Wp是程序中的并行分量，N为CPU数量。可以看出，在工作总量恒定的情况下，可并行部分代码占比越多，系统的整体性能越高。我们需要把思维从线性模型中抽离出来，继续细分整个处理流程，找出执行时间最长的程序热点，对这些代码段进行并行化从而将所有瓶颈逐个击破，这才是使通过并行化获得最大性能提升的最好办法。

根因拆解

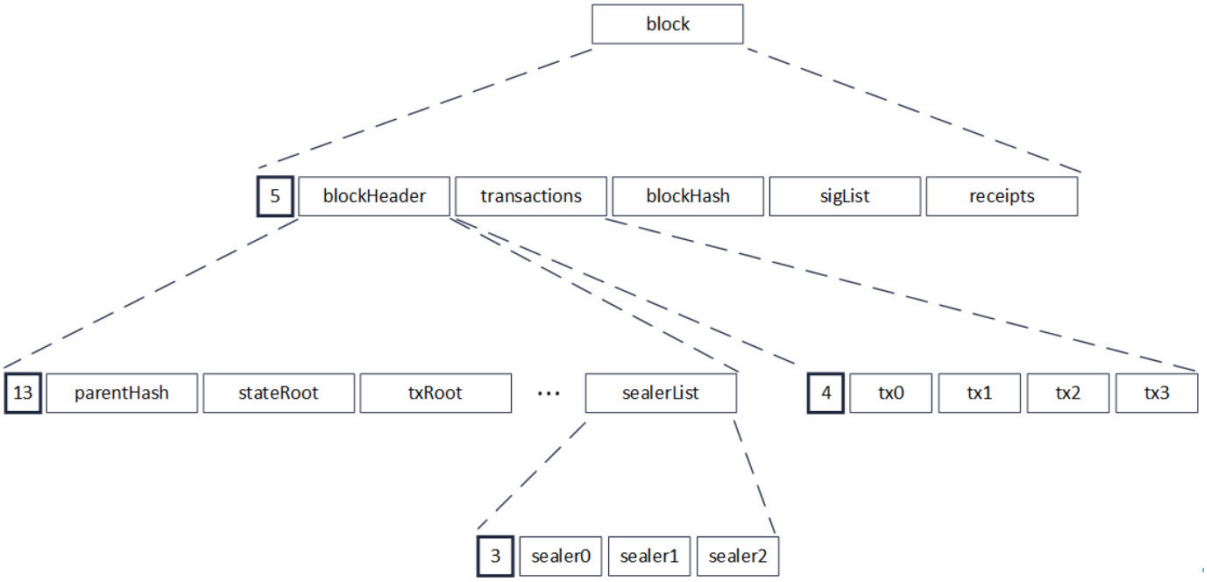
1.串行的区块解码

区块解码主要性能问题出在RLP编码方法本身。RLP全称是递归的长度前缀编码，是一种用长度作为前缀标明编码对象中元素个数的编码方法。如下图所示，RLP编码的开头即是此编码中的对象个数（Object num）。在个数后，是相应个数的对象（Object）。递归地，每个对象，也是RLP编码，其格式也与下图相同。

需要特别注意的是，在RLP编码中。每个Object的字节大小是不固定的，Object num只表示Object的个数，不表示Object的字节长度。



RLP通过一种长度前缀与递归结合的方式，理论上可编码任意个数的对象。下图是一个区块的RLP编码，在对区块进行编码时，先递归至最底层，对多个sealer进行编码，多个sealer被编码并加上长度前缀后，编码成为一串RLP编码（sealerList），此编码又作为一个对象，被编入上层的一串RLP编码（blockHeader）中。此后层层递归，最后编码成为区块的RLP编码。由于RLP编码是递归的，在编码前，无法获知编码后的长度。



解码时，由于RLP编码中每个对象的长度不确定，且RLP编码只记录了对象的个数，没记录对象的字节长度，若要获取其中的一个编码对象，必须递归解码其前序的所有对象，在解码前序的对象后，才能访问到需要访问的编码对象的字节位置。例如在上图中，若需要访问区块中的第0笔交易，即tx0，必须先将blockHeader解码，而blockHeader的解码，需要再次递归，把parentHash，stateRoot直至sealerList都解码出来。

解码区块最重要的目的是解码出包含在区块中的交易，而交易的编码都是互相独立的，但在RLP特殊的编码方式下，解码一笔交易的必要条件是解码出上一笔交易，交易的解码任务之间环环相扣，形成了一种链式的依赖关系。需要指出的是，这种解码方式并不是RLP的缺陷，RLP的设计目标之一本就是尽量减少空间占用，充分利用好每一个字节，虽然编解码变得低效了些，但编码的紧凑度却是有目共睹，因此这种编码方式本质上还是一种时间换空间的权衡结果。

由于历史原因，FISCO BCOS中使用了RLP编码作为多处信息交换协议，贸然换用其他并行化友好的序列化方案可能会带来较大的开发负担。基于这一考虑，我们决定在原有的RLP编解码方案稍作修改，通过为每个被编码的元素添加额外的位置偏移信息，便可以做到并行解码RLP的同时不会改动大量原有代码。

2.交易验签&数据落盘开销大

通过对交易验签和数据落盘部分的代码进行拆解，我们发现两者的主要功能都集中在一个耗时巨大的for循环。交易验签负责按序取出交易，然后从交易的签名数据中取出(v, r, s)数据，并从中还原出交易发送者的公钥，其中，还原公钥这一步，由于涉及密码学算法，因此耗时不少；数据落盘负责从缓存中

逐个取出交易相关数据，将其编码为JSON字符串后写入磁盘，由于JSON编码过程本身效率比较低，因此也是性能损失的重灾区。

两者代码分别如下所示：

```
// 交易验签
for(int i = 0; i < transactions.size(); ++i)
{
  tx = transactions[i];
  v, r, s = tx.getSignature();
  publicKey = recover(v, r, s);  // 从(v, r, s)中复原出发送者公钥
  ...
}
```

```
// 数据落盘
for(int i = 0; i < datas.size(); ++i)
{
  data = datas[i];
  jsonStr = jsonEncode(data);  // 将数据编码为JSON字符串进行存储
  db.commit(jsonStr);
  ...
}
```

两个过程共有的特点是，它们均是将同样的操作应用到数据结构中不同的部分，对于这种类型的问题，可以直接使用数据级并行进行改造。所谓数据级并行，即是将数据作为划分对象，通过将数据划分为大小近似相等的片段，通过在多个线程上对不同的数据片段上进行操作，达到并行处理数据集的目的。

数据级并行唯一的附加要求是任务之间彼此独立，毫无疑问，在FISCO BCOS的实现中，交易验签和数据落盘均满足这一要求。

优化实践

1.区块解码并行化

改造过程中，我们在系统中使用的普通RLP编码的基础上，加入了offset字段，用以索引每个Object的位置。如下图所示，改造后编码格式的开头，仍然是对象的个数（Object num），但是在个数字段后，是一个记录对象偏移量的数组（Offsets）。

数组中的每个元素有着固定的长度。因此要读取某个Offset的值，只需向访问数组一样，根据Offset的序号直接索引便可以进行随机访问。在Offsets后，是与RLP编码相同的对象列表。相应序号的Offset，指向相应序号的对象的RLP编码字节位置。因此，任意解码一个对象，只需要根据对象的序号，找到其偏移量，再根据偏移量，就可定位到相应对象的RLP编码字节位置。

编码流程也进行了重新设计。流程本身仍然基于递归的思路，对于输入的对象数组，首先将对象数组的大小编码在输出编码的开头处，若数组大小超过1，则按序逐个取出待编码对象并缓存其递归编码，并在Offsets数组中记录该对象的偏移位置，待数组遍历完后，将缓存的对象编码第一次性取出并附加至输出编码末尾；若数组大小为1，则递归对其编码并写入输出编码的末尾，结束递归。

编码流程的伪代码如下：

```
Rlps = RLP();  // Output，初始时为空
void encode(objs)  //Input: objs = 待编码对象的数组
{
  offset = 0;
  codes = [];

  objNum = objs.size()
  Rlps.push(objNum)
  if objNum > 1
  {
    for obj in objs
    {
      rlp = encode(obj);  // 递归调用编码方法
      Rlps.push(offset);
      offset += rlp.size();
      codes.add(rlp);  // 缓存递归编码的结果
    }
    for x in codes
    {
      Rlps.push(x);
    }
  }
  else
  {
    rlp = encode(objs[0]);
    Rlps.push(rlp);
  }
}
```

偏移量的引入使解码模块能够对元素编码进行随机访问。Offsets的数组范围可以在多个线程间均摊，从而每个线程可以并行访问对象数组的不同部分，分别进行解码。由于是只读访问，这种并行方式是线程安全的，仅需最后再对输出进行汇总即可。

解码流程的伪代码如下：

```
Objs decode(RLP Rlps)
{
  objNum = Rlps.objNum;   // 获取对象个数
  outputs = []  // 输出的对象数组
  if objNum > 1
  {
    parallel for i = 0 to objNum
    {
      offset = Rlps.offsets[i];
      code = Rlps.objs[offset];
      x = decode(code);
      outputs.add(x);   // 有序插入outputs
    }
  }
  else
  {
    outputs.add(decode(Rlps.objs[0]));
  }
  return outputs;
}
```

2.交易验签 & 数据落盘并行化

对于数据级并行，业内已有多种成熟的多线程编程模型。虽然Pthread这类显式的多线程编程模型能够提供对线程进行更精细的控制，但是需要我们对线程通信、同步拥有娴熟的驾驭技巧。实现的复杂度越高，犯错的几率越大，日后代码维护的难度也相应增加。我们的主要目标仅仅对密集型循环进行并行

化，因此在满足需求的前提下，Keep It Simple & Stupid才是我们的编码原则，因此我们使用隐式的编程模型来达成我们的目的。

经过再三权衡，我们在市面上众多隐式多线程编程模型中，选择了来自Intel的线程构建块（Thread Building Blocks，TBB）开源库。在数据级并行方面，TBB算是老手，TBB运行时系统不仅屏蔽了底层工作线程的实现细节，还能够根据任务量自动在处理器间平衡工作负载，从而充分利用底层CPU资源。

使用TBB后，交易验签和数据落盘的代码如下所示：

```
// 并行交易验签
tbb::parallel_for(tbb::blocked_range<size_t>(0, transactions.size()),
[&](const tbb::blocked_range<size_t>& _range)
{
  for(int i = _range.begin(); i != _range.end(); ++i)
  {
    tx = transactions[i];
    v, r, s = tx.getSignature();
    publicKey = recover(v, r, s);  // 从(v, r, s)中复原出发送者公钥
    ...
  }
});
```

```
// 并行数据落盘
tbb::parallel_for(tbb::blocked_range<size_t>(0, transactions.size()),
[&](const tbb::blocked_range<size_t>& _range)
{
  for(int i = _range.begin(); i != _range.end(); ++i)
  {
    data = datas[i];
    jsonStr = jsonEncode(data);  // 将数据编码为JSON字符串进行存储
    db.commit(jsonStr);
    ...
  }
});
```

可以看到，除了使用TBB提供的tbb::parallel_for进行并行循环和tbb::blocked_range引用数据分片外，循环体内的代码几乎没有任何变化，接近C++原生语法正是TBB的特点。TBB提供了抽象层级较高的并行接口，如parallel_for、parallel_for_each这类泛型并行算法，从而使得改造能够较为容易地进行。同时，TBB不依赖任何语言或编译器，只要有能支持ISO C++标准的编译器，便有TBB的用武之地。

当然，使用TBB并不是完全没有额外负担，比如线程间安全还是需要开发人员的仔细分析来保证，但TBB考虑周到，提供了一套方便的工具来辅助我们解决线程间互斥的问题，如原子变量、线程局部存储和并行容器等，这些并行工具同样被广泛地应用在FISCO BCOS中，为FISCO BCOS的稳定运行保驾护航。

写在最后

经过一套并行优化的组合拳，FISCO BCOS的性能表现更上层楼。压力测试的结果表明，FISCO BCOS的交易处理能力，相较于并行化改造之前，成功提升了1.74倍，基本达到了这个环节的预期效果。

但是我们也深深明白，性能优化之路漫漫，木桶最短的一板总是交替出现，并行之道在于，通过反复的分析、拆解、量化和优化，使得各模块互相配合齐头并进，整个系统达到优雅的平衡，而最优解总是在“跳一跳”才能够得着的地方。

FISCO BCOS可并行合约开发框架（附实操教程）

作者：石翔 | FISCO BCOS 核心开发者

本专题系列文章追到现在，也许你会想问，FISCO BCOS的并行到底怎么用？作为专题的完结篇，本文就来揭晓“庐山真面目”，并教你上手使用FISCO BCOS的并行特性！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。

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。互斥情况如下：

交易	互斥对象	交集	是否互斥
transfer(A, B) 和 transfer(A, C)	[A, B] 和 [A, C]	[A]	互斥，不可并行执行
transfer(A, B) 和 transfer(B, C)	[A, B] 和 [B, C]	[B]	互斥，不可并行执行
transfer(A, C) 和 transfer(B, C)	[A, C] 和 [B, C]	[C]	互斥，不可并行执行
transfer(A, B) 和 transfer(A, B)	[A, B] 和 [A, B]	[A, B]	互斥，不可并行执行
transfer(A, B) 和 transfer(C, D)	[A, B] 和 [C, D]	无	无互斥，可并行执行

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。

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合约的并行框架

编写并行的solidity合约，开发流程与开发普通solidity合约流程相同。在此基础上，只需将ParallelContract 作为需要并行的合约基类，并调用registerParallelFunction()，注册可以并行的接口即可。

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
{
        // 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); // 冲突参数:␣
↪string string
        registerParallelFunction("set(string,uint256)", 1); // 冲突参数: 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：

step1 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;
}
```

step2 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

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;
}
```

step3 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
```

(continues on next page)

(续上页)

```
    registerParallelFunction("transfer(string,string,uint256)", 2); // transfer接
口，前2个是互斥参数
    registerParallelFunction("set(string,uint256)", 1); // transfer接口，前1个四互斥参
数
}

// 注销并行合约接口
function disableParallel() public
{
    unregisterParallelFunction("transfer(string,string,uint256)");
    unregisterParallelFunction("set(string,uint256)");
}
```

step4 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，将在下文的例子中给出。

预编译合约的并行框架

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。

step1 将合约定义成支持并行

```
bool isParallelPrecompiled() override { return true; }
```

step2 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))
        {
            // 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
}
```

step3 编译，重启节点

手动编译节点的方法，参考FISCO BCOS技术文档。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。

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

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，让交易均匀发送到每个节点上，才能让链接收足够多的交易

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。

step1 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。

step2 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。
```

step3 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  : 0.04400000000000004%
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)。

step4 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（3 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仅做示例使用，请勿直接运用于生产环境。

step1 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。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 add 10000 2500 user
```

step2 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。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 transfer 100000 4000 user 2
```

step3 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:
```

(continues on next page)

(续上页)

```
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
```

可以看出，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)。

step4 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。

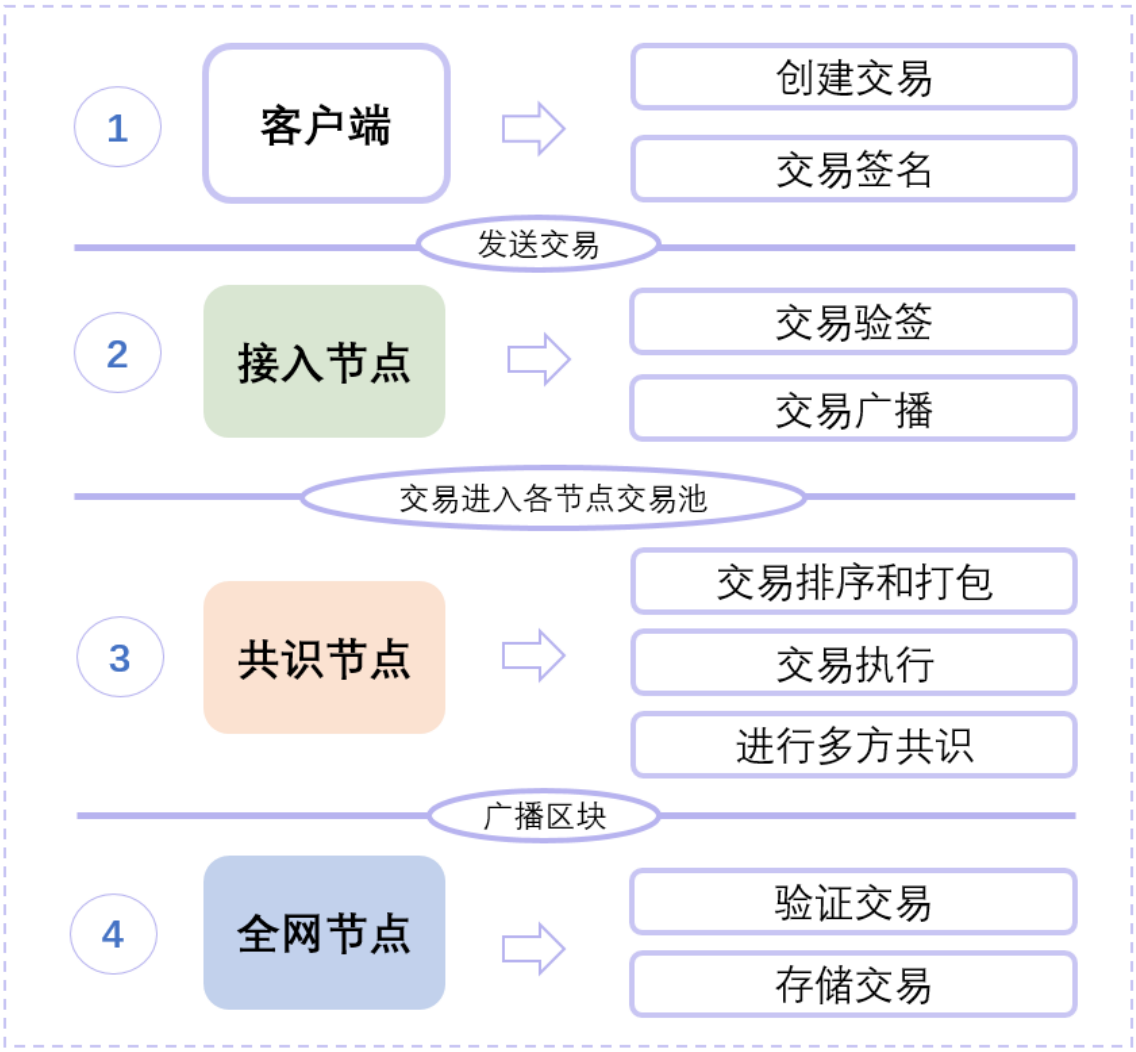
如您在部署过程中遇到阻碍或有问题需要咨询，可以进入FISCO BCOS官方技术交流群寻求解答。
（进群请长按下方二维码识别添加小助手）

FISCO BCOS中交易的一生

作者：李陈希 | FISCO BCOS 核心开发者

交易——区块链系统的核心，负责记录区块链上发生的一切。区块链引入智能合约后，交易便超脱『价值转移』的原始定义，其更加精准的定义应该是区块链中一次事务的数字记录。无论大小事务，都需要交易的参与。

交易的一生，贯穿下图所示的各个阶段。本文将梳理交易的整个流转过程，一窥FISCO BCOS交易完整生命周期。

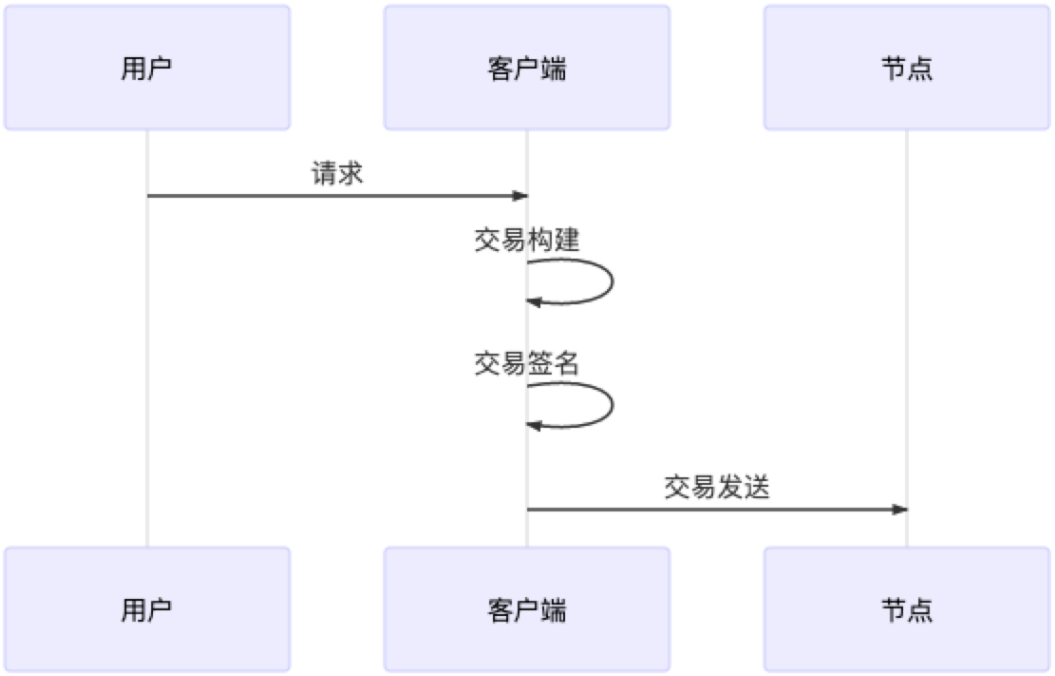


交易生成

用户的请求给到客户端后，客户端会构建出一笔有效交易，交易中包括以下关键信息：

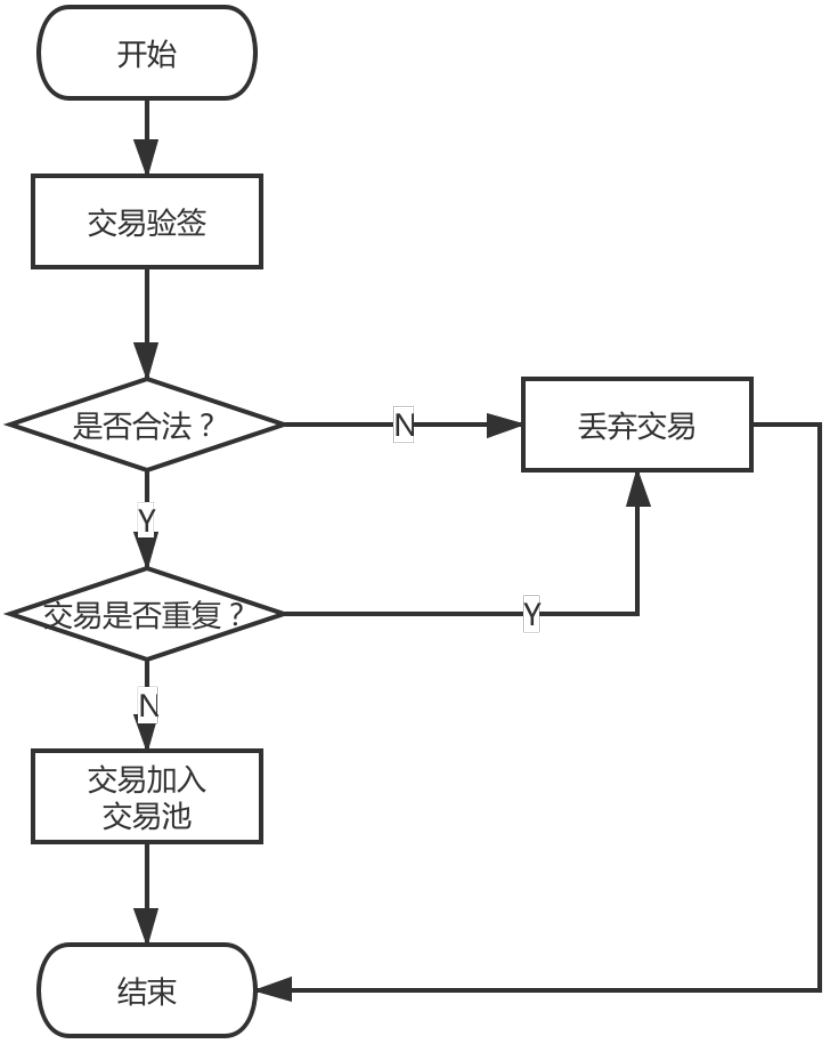
1. 发送地址：即用户自己的账户，用于表明交易来自何处。
2. 接收地址：FISCO BCOS中的交易分为两类，一类是部署合约的交易，一类是调用合约的交易。前者，由于交易并没有特定的接收对象，因此规定这类交易的接收地址固定为0x0；后者，则需要将交易的接收地址置为链上合约的地址。
3. 交易相关的数据：一笔交易往往需要一些用户提供的输入来执行用户期望的操作，这些输入会以二进制的形式被编码到交易中。
4. 交易签名：为了表明交易确实是由自己发送，用户会向SDK提供私钥来让客户端对交易进行签名，其中私钥和用户账户是一一对应的关系。

之后，区块链客户端会再向交易填充一些必要的字段，如用于防交易重放的交易ID及blockLimit。交易的具体结构和字段含义可以参考编码协议文档，交易构造完成后，客户端随后便通过Channel或RPC信道将交易发送给节点。



交易池

区块链交易被发送到节点后，节点会通过验证交易签名的方式来验证一笔交易是否合法。若一笔交易合法，则节点会进一步检查该交易是否重复出现过，若从未出现过，则将交易加入交易池缓存起来。若交易不合法或交易重复出现，则将直接丢弃交易。



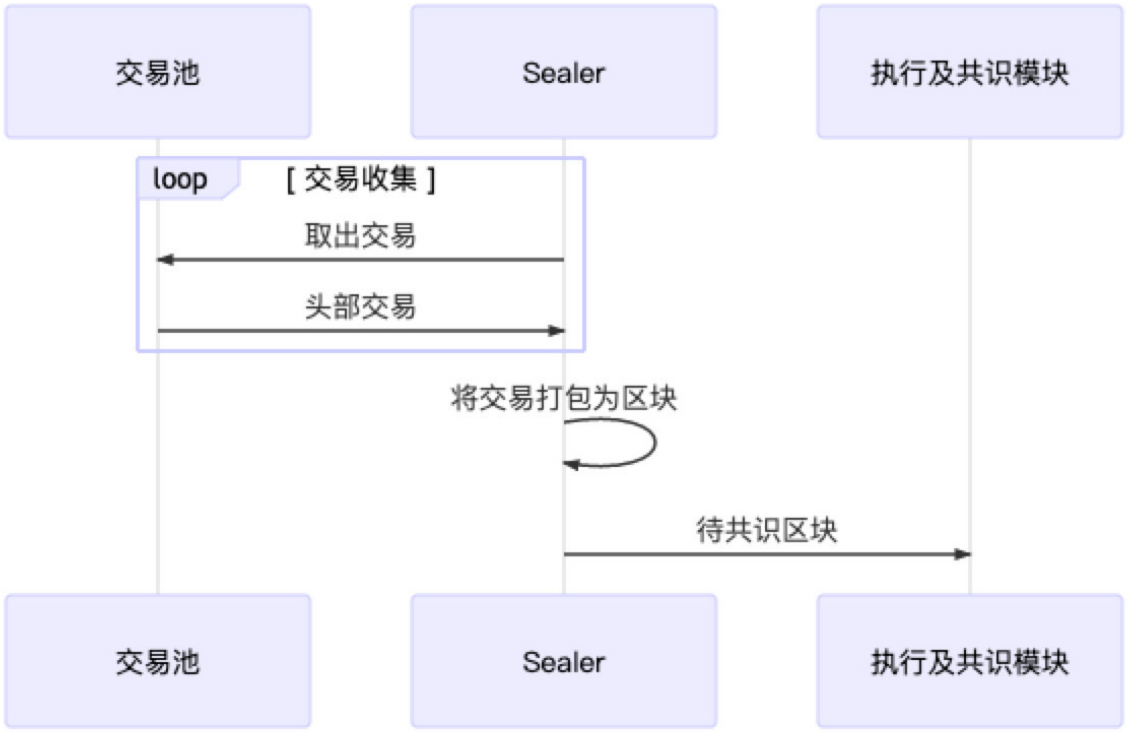
交易广播

节点在收到交易后，除了将交易缓存在交易池外，节点还会将交易广播至该节点已知的其他节点。

为了能让交易尽可能到达所有节点，其他收到广播过来的交易节点，也会根据一些精巧的策略选择一些节点，将交易再一次进行广播，比如：对于从其他节点转发过来的交易，节点只会随机选择25%的节点再次广播，因为这种情况一般意味着交易已经开始在网络中被节点接力传递，缩减广播的规模有助于避免因网络中冗余的交易太多而出现的广播风暴问题。

交易打包

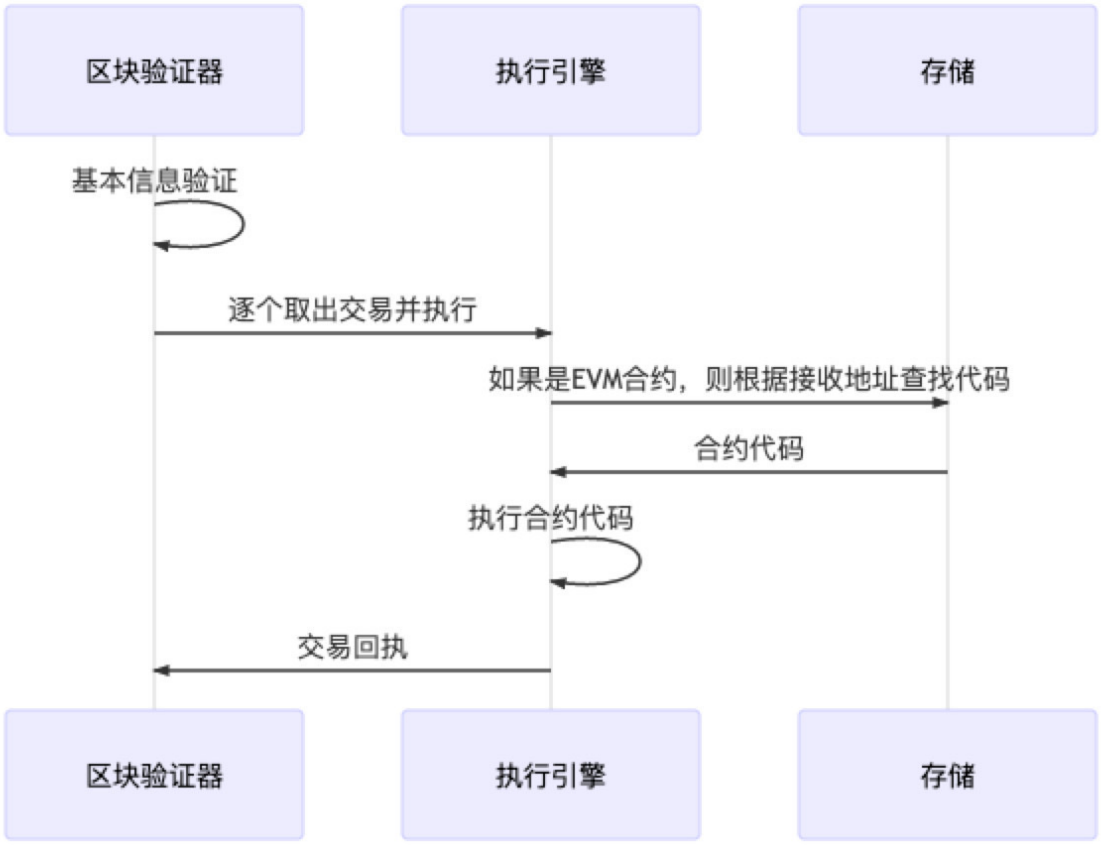
为了提高交易处理效率，同时也为了确定交易之后的执行顺序保证事务性，当交易池中有交易时，Sealer线程负责从交易池中按照先进先出的顺序取出一定数量的交易，组装成待共识区块，随后待共识区块会被发往各个节点进行处理。



交易执行

节点在收到区块后，会调用区块验证器把交易从区块中逐一拿出来执行。如果是预编译合约代码，验证器中的执行引擎会直接调用相应的C++功能，否则执行引擎就会把交易交给EVM（以太坊虚拟机）执行。

交易可能会执行成功，也可能因为逻辑错误或Gas不足等原因执行失败。交易执行的结果和状态会封装在交易回执中返回。



交易共识

区块链要求节点间就区块的执行结果达成一致才能出块。FISCO BCOS中一般采用PBFT算法保证整个系统的一致性，其大概流程是：各个节点先独立执行相同的区块，随后节点间交换各自的执行结果，如果发现超过2/3的节点都得出了相同的执行结果，那说明这个区块在大多数节点上取得了一致，节点便会开始出块。

交易落盘

在共识出块后，节点需要将区块中的交易及执行结果写入硬盘永久保存，并更新区块高度与区块哈希的映射表等内容，然后节点会从交易池中剔除已落盘的交易，以开始新一轮的出块流程。用户可以通过交易哈希等信息，在链上的历史数据中查询自己感兴趣的交易数据及回执信息。

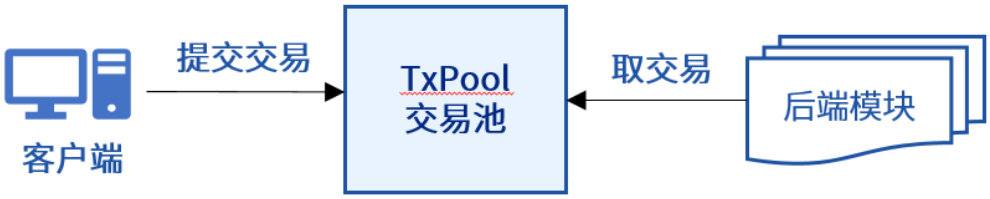
FISCO BCOS中交易池及其优化策略

作者：陈宇杰 | FISCO BCOS 核心开发者

作者语

FISCO BCOS区块链系统中，交易上链之前，均存储在交易池中。交易池是区块链小能手，一方面担任质检员的职务，将所有非法交易拒之门外；一方面担任供应商的职责，向共识模块输送合法交易；还负责向客户端推送上链通知。可以说，FISCO BCOS区块链系统的交易池异常忙碌，其性能会直接影响区块链系统性能。本文就带领大家揭开交易池面纱，了解交易池多重身份，并一起领会FISCO BCOS区块链系统中的交易池如何在多重角色间游刃有余。

初识交易池

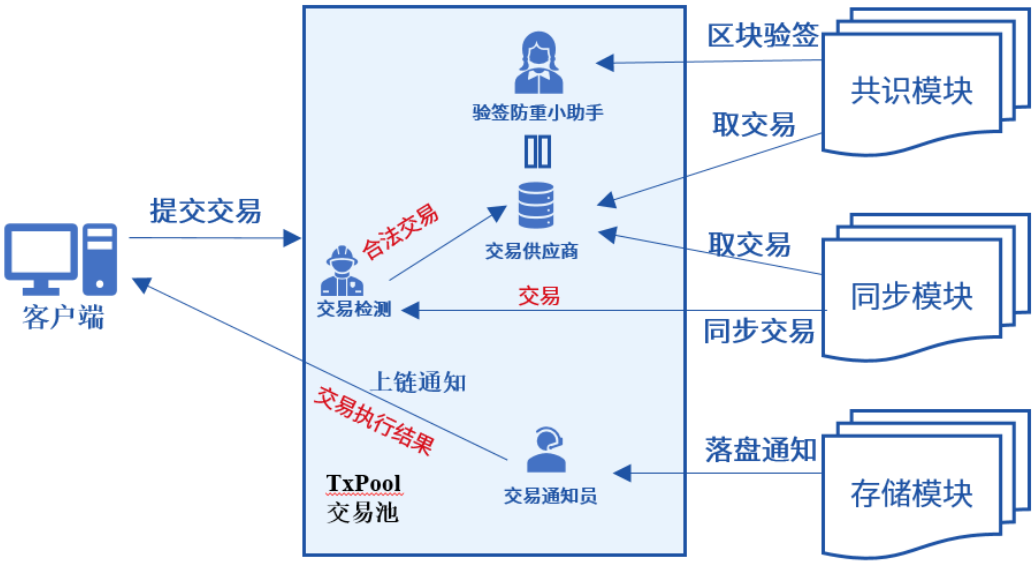


如上图所示，FISCO BCOS区块链系统中，接收存储客户端发送的交易是交易池基本职责，这些交易是共识模块打包交易和同步模块广播交易的“原材料”。交易池需要保证这些交易“原材料”的质量，对交易合法性进行验证。当然，为了防止DOS攻击，FISCO BCOS对交易池容量进行了限制，当交易池内交易数目超过容量限制后，会拒绝客户端发送的新交易。

交易池的重要性

FISCO BCOS区块链系统中，交易池作为关键系统模块，同时肩负着和SDK以及后端多个模块的交互，本节就以交易池的多重职责为切入点，和大家一起看看交易池有多忙。

交易池的四重职责



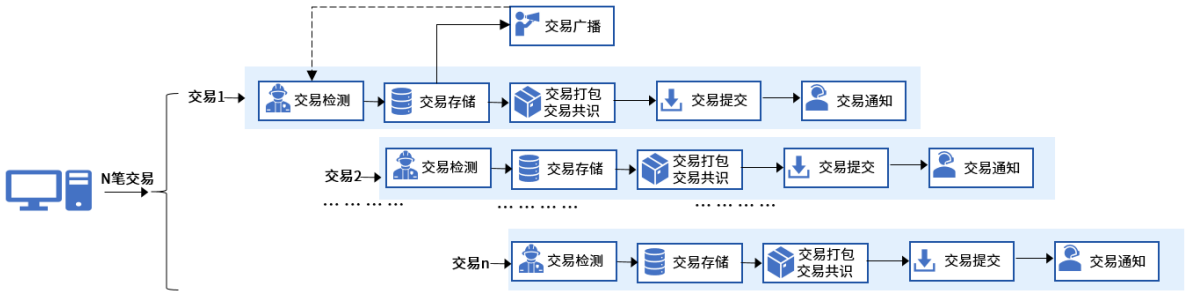
上图展示了交易从客户端发出到上链整个生命周期里，交易池所扮演的多重角色：

- **交易质量检测员**：交易放入交易池前，检测交易的有效性，有效的交易必须满足：①签名有效；② 非重复交易；③ 非已上链交易。
- **交易供应商**：存储合法交易，为后端模块提供交易“原材料”。
- **验签防重小助手**：为共识模块提供区块验签接口，仅验证在交易池内未命中交易，提升共识区块验证效率。
- **交易上链通知员**：交易上链成功后，将交易执行结果通知到客户端。

交易池作为区块链核心模块，身兼四职，每笔交易处理流程中，均需与后端三个模块、内部四个模块以及客户端进行多达八次交互，确实是异常忙碌。

交易池的作用

下面就以区块链节点交易处理生命周期为例，详细了解交易池各重角色在其中所起的作用。



如上图，客户端发送的交易到节点会被流水线式处理，每条流水均需有五个处理流程：

- **交易检测**：客户端发送的交易发送到节点后，首先要经过交易池的质量检测员的检验，交易池仅会把签名有效、不重复、未上链的交易放入交易池内。
- **交易存储**：交易过了“质量检测”后，被存储交易池内，此时交易池身负“供应商”的角色，共识模块从交易池内获取新交易用于打包成区块；同步模块从交易池内获取新增交易，广播给所有其他共识节点。
- **交易打包&&交易共识**：共识模块从交易池内获取合法交易，打包成新区块，并广播到所有其他共识节点，其他共识节点收到打包的新区块后，为了保证区块的合法性，会验证区块内每笔交易签名。考虑到交易验签是非常耗时的操作，且新区块内交易有极大概率在其他节点交易池内命中，为了提升共识验证效率，交易验签防重小助手此时派上了用场，它会仅验证新区块中未在本地交易池命中的交易签名。
- **交易提交**：交易共识达成后，会调用存储模块，将交易及其执行结果提交到区块链数据库。
- **交易通知**：交易上链成功后，交易池的上链通知员将交易执行结果通知给客户端。

在交易从发出到上链的整个生命周期里，每个过程都有交易池的参与，因此交易池对于整个区块链系统非常重要，交易池每个处理过程都直接影响了区块链系统性能。

交易池优化

通过前面介绍，我们了解到FISCO BCOS区块链系统的交易池异常忙碌，并且直接影响了区块链系统性能，本节就来详细说说交易池的优化历程和优化方法。

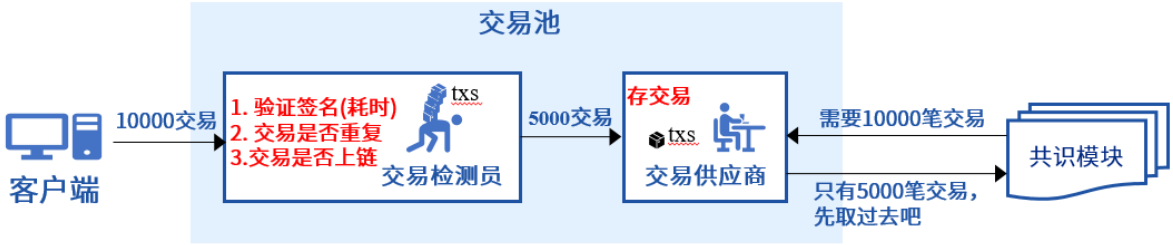
优化交易处理流水线效率

通过上面交易处理流水线示意图可看出，交易池参与了交易处理的每个流程，因此交易池每个处理流程都对系统性能影响很大。FISCO BCOS区块链系统采用拆分且并行执行交易验证任务、交易异步通知策略来优化交易流水线处理效率。

优化交易验证效率

FISCO BCOS rc2引入并行交易后，FISCO BCOS开发者们发现压测过程中共识模块每个区块经常无法打满交易，偶尔还会出现共识模块出空块、等待交易池提供新交易的现象。排查发现，交易池作为交易检测员任务太重，既要验证交易签名，又要检查交易是否重复、是否已上链，导致向交易供应商提供交易效率非常低，经常出现交易供不应求的情况，严重影响了区块链系统TPS。

下图描述了这种供不应求的现象：



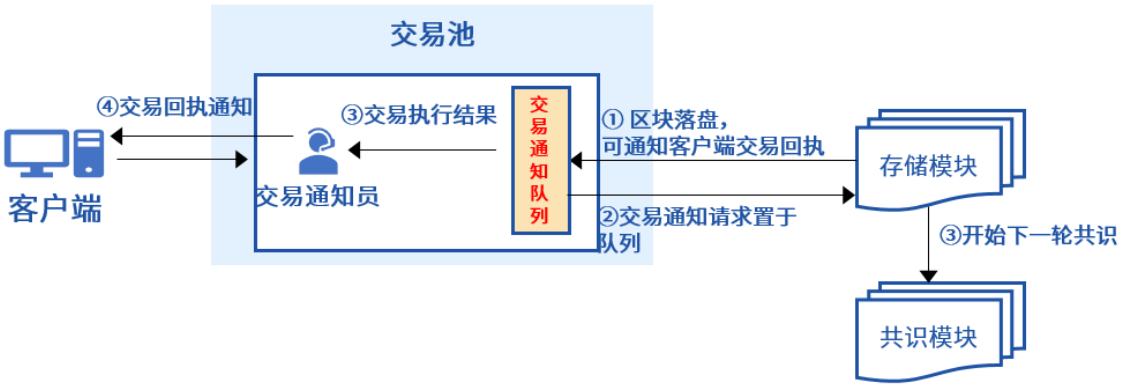
为了打破交易池供不应求的困局，优化交易流水线处理效率，FISCO BCOS区块链系统引入专门的交易验签模块，并将“交易检测员”的“验证签名”职责分担给了这个新模块，且为了进一步提升交易存储效率，交易验签模块并行对交易进行验证。优化交易处理流水线后，“交易检测员”的工作负担轻了很多，交易供应商完全能满足共识模块的交易需求，而且还留有部分存货。

优化处理流水线后，”交易检测员”的重活被”人力充足”的”验签模块”分担了，系统性能显著提升：采用并行交易压测，FISCO BCOS区块链系统性能突破了1W。

交易异步通知

通过前面的介绍，大家了解到交易池还承担着交易通知的职责，”交易通知员”也是个忙碌的角色，它需要在收到区块落盘信号后，将所有上链交易通知给客户端，当且仅当共识模块确认上一轮共识的上链交易都会被通知到后，共识模块才会开始下一轮共识，交易同步推送无疑会拖慢共识流程。为了进一步优化流水线处理效率，FISCO BCOS区块链系统采用了交易异步通知策略：存储模块将交易通知结果放置于交易通知队列后直接返回，共识模块直接开始下一轮共识流程，与此同时，交易通知模块将交易执行结果依次返回给客户端。

如下图所示：



采用交易异步通知策略后，交易通知不会阻塞共识流程，大约能提升10%的系统性能。

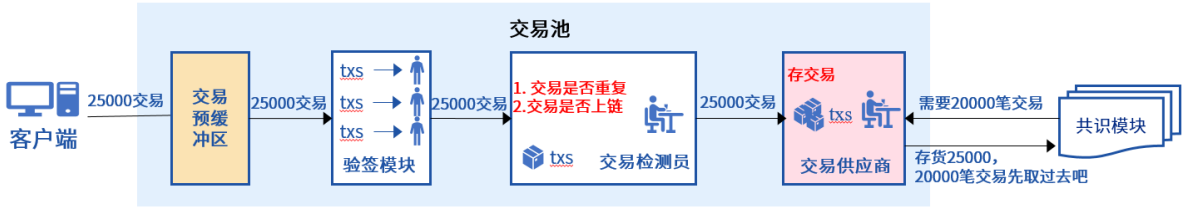
双缓存队列

FISCO BCOS 2.1之后，FISCO BCOS团队认真统计了每个区块的处理时间，觉得系统性能还有继续上升的空间，于是决定继续优化性能，进一步提升FISCO BCOS区块链系统的处理能力。

当存储模块和执行模块性能优化到极致后，可是最终的压测结果总是不符合预期。经过排查发现交易池又出现了供不应求的情况，只是这种供不应求是客户端引起的，客户端发送交易后，大量线程阻塞在等待交易验证通过，返回交易哈希，无法空出更多线程发送新交易。

为了提升节点对客户端的响应速度，从而提升客户端交易发送速率，FISCO BCOS区块链系统在“交易供应商”持有的交易储存队列基础上，引入了交易预缓冲区，存放客户端发送到节点的交易，并直接对客户端进行响应。

交易预缓冲区会持续将缓存的交易送到”验签模块”和”交易检测员”处进行验证，验证通过的交易最终会被放入到真正的交易队列供交易供应商调度，如下图所示：



这种双缓存队列机制，大大提升了交易池对客户端的响应速度，客户端也可以持续腾出线程继续发送新交易，优化后，客户端发送性能更高，交易池供大于求。

小结

交易池很忙，FISCO BCOS区块链系统中，交易池更忙，它被用来验证交易、存储交易、防止交易重复验签、向客户端推送交易执行结果等。交易池很重要，FISCO BCOS区块链系统中，交易池更重要，任务繁重的”交易检测员”会大大降低交易插入速率，导致交易供不应求；没有”交易预缓冲区”的交易池会阻塞客户端交易发送线程，降低客户端交易发送速率；交易同步推送，会损耗10%左右的系统性能……

在性能优化的道路上，交易池性能优化一直会被排在重要的位置。

22.4.2 FISCO BCOS性能

峰值TPS：2万+ TPS（PBFT）

交易确认时延：秒级

FISCO BCOS的速度与激情：性能优化方案最全解密

作者：石翔 | FISCO BCOS 核心开发者

上篇文章说到，区块链的速度困境是“贵”在信任，“慢”得其所，说到底，根因还是在其“用计算换信任”的设计思路上。业内普遍赞誉区块链是信任的机器，为了实现信任，区块链不得不做很多复杂而繁琐的操作，同步、验签、执行、共识等，都是区块链中必不可少的环节。

这就像行车时的“交规”，时刻在告诉我们开发者，为了安全，请按规定速度行驶！然而，社区里大家还是有着共同的心声：真的太慢了！

那么，能不能对这台信任的机器来一次装备升级，让它既安全又快速呢？经过团队深入的探索和实践，我们打通了多条迈向极速时代的路子。回看整个过程，恰如打造了一台性能卓越的汽车。

- 高功率发动机**基于DAG的交易并行执行引擎**
- 燃料输送装置**分布式存储**

- 前排与后座**共识与同步的流程优化**
- 传动装置**全方位并行处理**
- 氢燃料**预编译合约**
- 监控仪表**全面的性能分析工具**
- 专属的方向盘**可并行合约开发框架**

高功率发动机：基于DAG的交易并行执行引擎——极尽所能让交易并行执行

传统的交易执行引擎，采用串行的方式执行交易，交易只能被一条条依次执行。一个区块中无论有多少交易，都需要一条条依次执行完。这就好比一个低功率的发动机，纵使给它装上巨无霸油箱，仍然无法输出强大的动力。1个气缸不够，改成4个气缸，8个气缸，行不行?

FISCO BCOS实现了一种交易并行执行引擎（PTE），能够让一个区块内的多个交易同时被执行。若机器有4个核，最大限度能支持4笔交易同时执行，如果有8个核，则能支持8笔交易同时执行。当然，在“交规”管控下，并行执行的正确性需要得到保证，也就是说，并行执行的结果和串行执行的结果需要一致。为了保证并行执行的一致性，FISCO BCOS的交易并行执行引擎（PTE）引入了DAG（有向无环图）这个数据结构。

执行引擎在执行区块中的交易之前，会根据交易相互间的互斥关系，自动构建交易间的依赖关系。这种依赖关系是一种DAG，在引擎执行时，会根据DAG让可并行的交易并行执行。这样一来，交易执行的一致性得以保证，交易执行的吞吐量也得到数量级的提升。

燃料输送装置：分布式存储——为引擎提供足够的燃料

传统的区块链存储模式，是一棵参天的MPT树。区块链上所有的数据，都汇聚到这棵树上来。对数据的每一次写或读，都是一次从树枝到树根（或者从树根到树枝）的漫长旅行。随着链上的数据越来越多，树也越来越高，树枝到树根的路程会变得越来越长。更麻烦的是，虽然树枝有很多个，但是树根只有一个。对海量链上数据的写或读，就像千军万马抢过独木桥一样悲壮，惨烈程度可想而知。所以传统的区块链，选择了一个个来，一个数据一个数据地读，一条交易一条交易地执行。形象地说，就是用一根输油管为引擎输送燃料。

这样肯定不行！我们需要多个输油管为引擎输送燃料！这一次，FISCO BCOS不是粗暴地为引擎接上多个输油管（MPT树），因为用输油管输油（用MPT存数据）实在是太慢了。我们干脆抛弃输油管，直接把引擎泡进油箱里！这样的比喻也许欠妥当，但理解了FISCO BCOS的执行引擎和存储设计，相信你会和我有一样的感慨。

我们抛弃MPT树，采用“表”的方式组织数据。执行引擎读写数据，无需再对MPT树进行树根到树枝的遍历，直接在“表”上读写。这样一来，每一条数据的读写，都不依赖于一个全局的操作，可以分开独立进行。这就为交易并行执行引擎（PTE）提供了并发数据读写的基础，类似于泡在油箱里的发动机，汽油直接流入气缸，谁也不共用谁的输油管。

分布式存储详细解析请点击：分布式存储架构设计

前排与后座：共识与同步的流程优化——不搞平均主义，先富带动后富

在区块链节点中，同步模块和共识模块，是一对形影不离的双胞胎，有时相互帮助，有时也为争夺资源大打出手。在以往的设计中，同步模块和共识模块并没有优先级的区分。好比坐车，谁坐前排，谁坐后排，没个规定，导致这对双胞胎经常在争夺先后顺序上浪费大量的时间。

一切从实际出发，先富带动后富！

共识模块负责主导整个区块链出块的节奏，应让共识模块先行。而同步模块，理应扮演好配合的角色，辅佐共识模块更快出块。基于此思想，FISCO BCOS对共识与同步的流程进行了优化：

- 第一，将同步模块中交易验签的操作，从P2P的回调线程中剥离出来，让共识模块能够更加顺畅地收到共识消息，以便更快进行共识。

- 第二，对交易验签进行去重，并对交易的二进制进行缓存。一笔交易只进行一次验签和解码，为共识模块中区块的执行腾出更多的CPU资源。
- 第三，优化同步流程，在交易同步之前，尽可能地让同步模块跑在共识模块之前，从而使得同步模块优先把交易写入交易池中，优先进行解码和验签，让共识模块拿到交易时，免去解码和验签的过程，更快进入区块打包阶段。

总而言之，言而总之。一切的目的，都是为共识的流程服务，让其更快更顺畅地打包、执行、共识、出块。

传动装置：全方位并行处理——让功率有效地输出

若不搭配合适的传动装置，再高功率的引擎也无法将功率有效输出。签名验证、编解码、数据落盘，是区块链中除开交易以外，其他耗时占用较高的部分。在以往的设计中，签名验证、编解码、数据落盘，都是串行执行的。就算交易被并行执行，这台信任机器的性能，也受制于这三个环节的性能。

这三个环节的性能问题一日不绝，性能永无抬头之日！那就给高功率发动机配上一个高性能传动装置，释放出它的威力来。

FISCO BCOS引入了并行容器，让数据的读写天然支持并发访问。在此基础上，对于交易的验签，直接让交易的验签并行执行，交易与交易间的验签流程互不影响；对于编解码，改造了RLP的编码格式，使原来只能按顺序读写的RLP格式支持并行的编解码；对于区块落盘，对状态的改变进行并行编码。

不仅如此，FISCO BCOS在可并行之处都进行了并行处理，让系统CPU资源得到最大化利用。交易不仅在进入合约引擎时能并行执行，在诸如签名验证、编解码、数据落盘等环节中也都是被并行处理的。强大的发动机，配合上高性能的传动装置，果然效果显著啊！

氢燃料：预编译合约 ——高效率的轻量级合约框架

众所周知，区块链上跑的是智能合约，智能合约用solidity语言编写。solidity合约部署到链上，烧掉Gas，得到结果。但是，有没有想过换一种燃料，一种成本更低却又让车跑得更快的燃料?

且看FISCO BCOS自研的“氢燃料”——预编译合约！

FISCO BCOS为机构提供了一种高性能、定制化、轻量级合约框架。机构可按照自身业务需求，将自己实现的预编译合约内置于FISCO BCOS节点中。预编译合约用C++编写，其性能高于solidity引擎，而且启动速度更快、指令更精简、内存使用更少。正如“氢燃料”一般，成本更低，热值更高，让汽车跑得更快！当然，提取“氢燃料”需要下一点小功夫，预编译合约的实现相对复杂，门槛比较高。了解预编译合约请点击：预编译合约架构设计

监控仪表：多维度性能分析工具——给人全局在握的踏实感

FISCO BCOS在开发过程中使用了大量的性能分析工具，就像汽车上安装了诸多指数清晰的监控仪表。我们采用了主流的性能分析工具，如perf，systemtap，对程序的热点、锁、内存等进行了分析，还根据区块链的程序流程特点，开发了定制化的性能分析工具，以便在共识，区块验证、存储模块和流程中更好地评估数据。工具能够对程序中各个阶段的时间占比、时间变化进行分析。有了可靠的量化工具，开发者在做每一处优化时，都能做到心中有数。

专属的方向盘：可并行合约开发框架——给开发者流畅的操作体验

一切准备就绪，上车！坐上驾驶位，你手中掌控的，将是FISCO BCOS为你提供的专属方向盘——可并行合约开发框架！如何合理操作这台机器，全靠这个方向盘。“手握”并行合约开发框架，在开发并行合约时，合约开发者无需关心具体的底层逻辑，而是将更多注意力集中在自己的合约逻辑上。当合约部署成功后，并行合约会被底层代码自动识别，自动并行地执行！

现在，终于开上了车。没过瘾？没关系，接下来的几篇，请接好，是真正的硬核干货！我们将在下一篇文章中，系统介绍FISCO BCOS中基于DAG模型的并行交易执行器（PTE），敬请期待~

区块链性能腾飞：基于DAG的并行交易执行引擎

作者：李陈希 | FISCO BCOS 核心开发者

在区块链世界中，交易是组成事务的基本单元。交易吞吐量很大程度上能限制或拓宽区块链业务的适用场景，愈高的吞吐量，意味着区块链能够支持愈广的适用范围和愈大的用户规模。当前，反映交易吞吐量的TPS（Transaction per Second，每秒交易数量）是评估性能的热点指标。为了提高TPS，业界提出了层出不穷的优化方案，殊途同归，各种优化手段的最终聚焦点，均是尽可能提高交易的并行处理能力，降低交易全流程的处理时间。

在多核处理器架构已经成为主流的今天，利用并行化技术充分挖掘CPU潜力是行之有效的方案。FISCO BCOS 2.0 中设计了一种基于DAG模型的并行交易执行器（PTE，Parallel Transaction Executor）。

PTE能充分发挥多核处理器优势，使区块中的交易能够尽可能并行执行；同时对用户提供简单友好的编程接口，使用户不必关心繁琐的并行实现细节。基准测试程序的实验结果表明：相较于传统的串行交易执行方案，理想状况下4核处理器上运行的PTE能够实现约200%~300%的性能提升，且计算方面的提升跟核数成正比，核数越多性能越高。

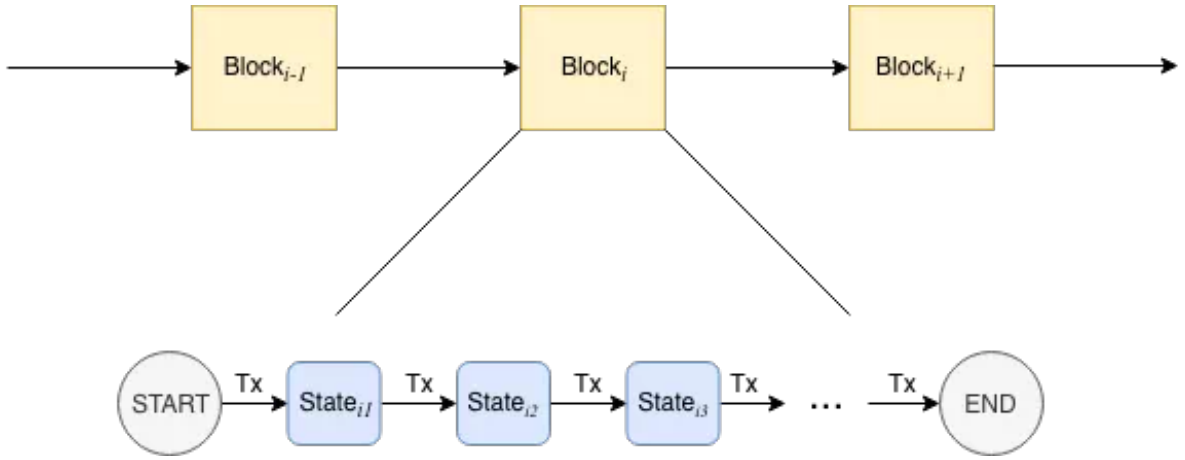
PTE为助力FISCO BCOS性能腾飞奠定了坚实基础，本文将全面介绍PTE的设计思路及实现方案，主要包括以下内容：

- **背景**：传统方案的性能瓶颈与DAG并行模型的介绍
- **设计思路**：PTE应用到FISCO BCOS中时遇到的问题以及解决方案
- **架构设计**：应用PTE后FISCO BCOS的架构及核心流程
- **核心算法**：介绍主要用到的数据结构与主要算法
- **性能测评**：分别给出PTE的性能与可扩展性测试结果

背景

FISCO BCOS交易处理模块可以被抽象为一个基于交易的状态机。在FISCO BCOS中，『状态』即是指区块链中所有账户的状态，而『基于交易』即是指FISCO BCOS将交易作为状态迁移函数，并根据交易内容从旧的状态更新为新的状态。FISCO BCOS从创世块状态开始，不断收集网络上发生的交易并打包为区块，并在所有参与共识的节点间执行区块中的交易。当一个区块内的交易在多个共识节点上执行完成且状态一致，则我们称在该块上达成了共识，并将该区块永久记录在区块链中。

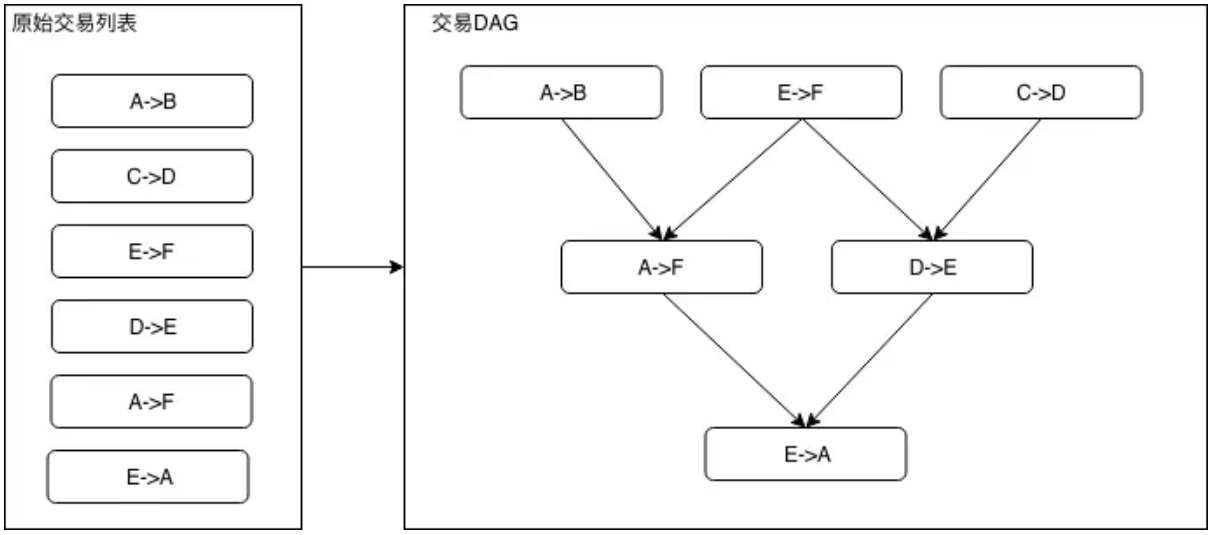
从上述区块链的打包→共识→存储过程中可以看到，执行区块中的所有交易是区块上链的必经之路。传统交易执行方案是：执行单元从待共识的区块逐条读出交易，执行完每一笔交易后，状态机都会迁移至下一个状态，直到所有交易都被串行执行完成，如下图所示：



显而易见，这种交易执行方式对性能并不友好。即使两笔交易没有交集，也只能按照先后顺序依次执行。就交易间的关系而言，既然一维的『线』结构有这般痛点，那何不把目光投向二维的『图』结构呢?

在实际应用中，根据每笔交易执行时需要使用的互斥资源（互斥意味着对资源的排他性使用，比如在上述转账问题互斥资源中，指的就是各个账户的余额状态），我们可以组织出一张交易依赖关系图，为防止交易依赖关系在图中成环，我们可以规定交易列表中牵涉到相同的互斥资源，且排序靠后的交易，必须等待靠前的交易完成后才被执行，由此得到的输出便是一张反映交易依赖关系的有向无环图，即交易DAG。

如下图所示，左侧的6笔转账交易可以组织为右侧的DAG形式：



在交易DAG中，入度为0的交易是没有任何依赖项、可以被立即投入运行的就绪交易。当就绪交易的数量大于1时，就绪交易可以被分散至多个CPU核心上并行执行。当一笔交易执行完，依赖于该交易的所有交易的入度减1，随着交易不断被执行，就绪交易也源源不断被产生。在极限情况下，假如构造出的交易DAG层数为1（即所有交易均是没有依赖项的独立交易），则交易整体执行速度的提升倍数将直接取决于处理器的核心数量n，此时若n大于区块内的交易数，则区块内所有交易的执行时间与单笔交易执行的时间相同。

理论上拥有如此让人无法拒绝的优美特性的交易DAG模型，该如何应用至FISCO BCOS中？

设计思路

要应用交易DAG模型，我们面临的首要问题便是：对于同一个区块，如何确保所有节点执行完后能够达到同一状态，这是一个关乎到区块链能否正常出块的关键问题。

FISCO BCOS采用验证(state root, transaction root, receipt root)三元组是否相等的方式，来判断状态是否达成一致。transaction root是根据区块内的所有交易算出的一个哈希值，只要所有共识节点处理的区块数据相同，则transaction root必定相同，这点比较容易保证，因此重点在于如何保证交易执行后生成的state和receipt root也相同。

众所周知，对于在不同CPU核心上并行执行的指令，指令间的执行顺序无法提前预测，并行执行的交易也存在同样情况。在传统的交易执行方案中，每执行一笔交易，state root便发生一次变迁，同时将变迁后的state root写入交易回执中，所有交易执行完后，最终的state root就代表了当前区块链的状态，同时再根据所有交易回执计算出一个receipt root。

可以看出，在传统的执行方案中，state root扮演着一个类似全局共享变量的角色。当交易被并行且乱序执行后，传统计算state root的方式显然不再适用，这是因为在不同的机器上，交易的执行顺序一般不同，此时无法保证最后的state root能够一致，同理，receipt root也无法保证一致。

在FISCO BCOS中，我们采用的解决方案是先执行交易，将每笔交易对状态的改变历史记录下来，待所有交易执行完后，再根据这些历史记录再算出一个state root，同时，交易回执中的state root，也全部变为所有交易执行完后最终的state root，由此就可以保证即使并行执行交易，最后共识节点仍然能够达成一致。

搞定状态问题后，下一个问题便是：如何判断两笔交易之间是否存在依赖关系？

若两笔交易本来无依赖关系但被判定为有，则会导致不必要的性能损失；反之，如果这两笔交易会改写同一个账户的状态却被并行执行了，则该账户最后的状态可能是不确定的。因此，依赖关系的判定是影响性能甚至能决定区块链能否正常工作的重要问题。

在简单的转账交易中，我们可以根据转账的发送者和接受者的地址，来判断两笔交易是否有依赖关系，比如如下3笔转账交易：A→B，C→D，D→E，可以很容易看出，交易D→E依赖于交易C→D的结果，但是交易A→B和其他两笔交易没有什么关系，因此可以并行执行。

这种分析在只支持简单转账的区块链中是正确的，但是一旦放到图灵完备、运行智能合约的区块链中，则可能不那么准确，因为我们无法准确知道用户编写的转账合约中到底有什么操作，可能出现的情况是：A->B的交易看似与C、D的账户状态无关，但是在用户的底层实现中，A是特殊账户，通过A账户每转出每一笔钱必须要先从C账户中扣除一定手续费。在这种场景下，3笔交易均有关联，则它们之间无法使用并行的方式执行，若还按照先前的依赖分析方法对交易进行划分，则必定会掉坑。

我们能否做到根据用户的合约内容自动推导出交易中实际存在哪些依赖项？答案是不太靠谱。我们很难去追踪用户合约中到底操作了什么数据，即使做到也需要花费不小的成本，这和我们优化性能的目标相去甚远。

综上，我们决定在FISCO BCOS中，将交易依赖关系的指定工作交给更熟悉合约内容的开发者。具体地说，交易依赖的互斥资源可以由一组字符串表示，FISCO BCOS暴露接口给到开发者，开发者以字符串形式定义交易依赖的资源，告知链上执行器，执行器则会根据开发者指定的交易依赖项，自动将区块中的所有交易排列为交易DAG。比如在简单转账合约中，开发者仅需指定每笔转账交易的依赖项是｛发送者地址+接收者地址｝。进一步地，如开发者在转账逻辑中引入了另一个第三方地址，那么依赖项就需要定义为｛发送者地址+接受者地址+第三方地址｝了。

这种方式实现起来较为直观简单，也比较通用，适用于所有智能合约，但也相应增加了开发者肩上的责任，开发者在指定交易依赖项时必须十分小心，如果依赖项没有写正确，后果无法预料。指定依赖项的相关接口会在后续文章中给出使用教程，本文暂且假定所有谈论到的交易依赖项都是明确无误的。

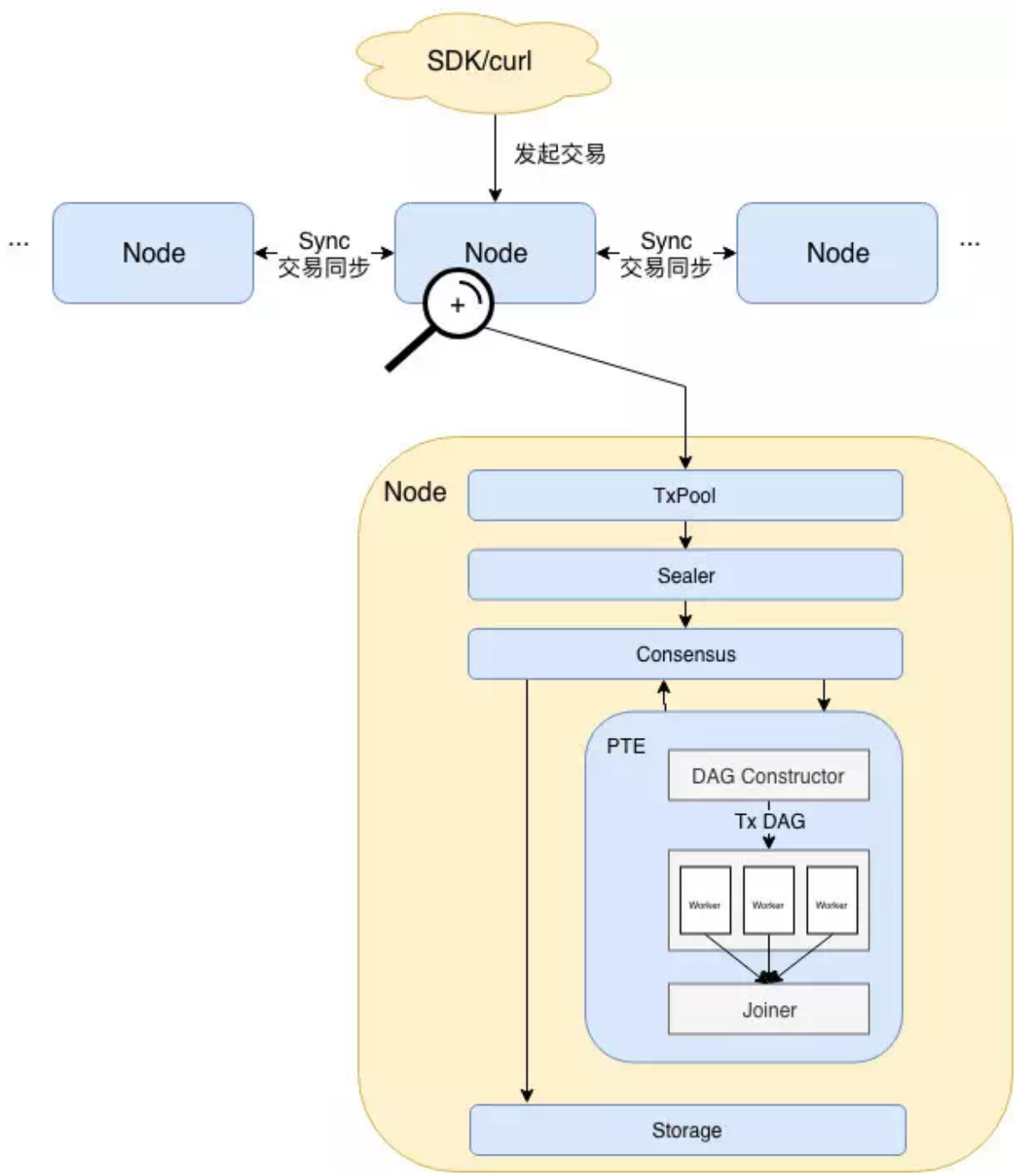
解决完上面两个比较重要的问题后，还剩下一些较为细节的工程问题：比如并行交易能否和非并行交易混合到一起执行？怎么保证资源字符串的全局唯一性？

答案也不复杂，前者可通过将非并行交易作为屏障（barrier）插入到交易DAG中——即我们认为，它即依赖于它的所有前序交易，同时又被它的所有后序交易依赖——来实现；后者可以通过在开发者指定的交易依赖项中，加入标识合约的特殊标志位解决。由于这些问题并不影响PTE的根本设计，本文暂不展开。

万事俱备，带着全新交易执行引擎PTE的FISCO BCOS已经呼之欲出。

架构设计

搭载PTE的FISCO BCOS架构图：



整个架构的核心流程如下：

用户通过SDK等客户端将交易发送至节点，此处的交易既可以是可并行执行的交易，也可以是不能并行执行的交易。随后交易在节点间同步，同时拥有打包权的节点调用打包器（Sealer），从交易池（Tx Pool）中取出一定量交易并将其打包成一个区块。此后，区块被发送至共识单元（Consensus）准备进行节点间共识。

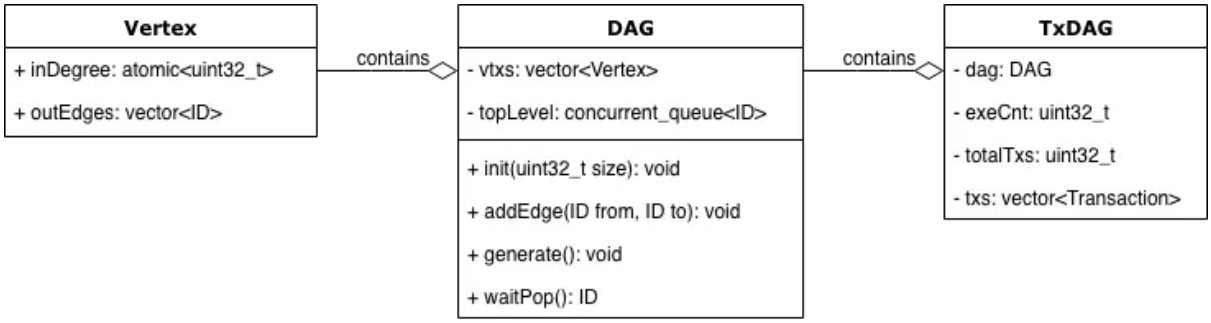
共识前需要执行区块中的交易，此处便是PTE施展威力之处。从架构图中可以看到，PTE首先按序读取区块中的交易，并输入到DAG构造器（DAG Constructor）中，DAG构造器会根据每笔交易的依赖项，构造出一个包含所有交易的交易DAG，PTE随后唤醒工作线程池，使用多个线程并行执行交易DAG。汇合器（Joiner）负责挂起主线程，直到工作线程池中所有线程将DAG执行完毕，此时Joiner负责根据各个交易对状态的修改记录计算state root及receipt root，并将执行结果返回至上层调用者。

在交易执行完成后，若各个节点状态一致，则达成共识，区块随即写入底层存储（Storage），被永久记录于区块链上。

核心算法

1.交易DAG的数据结构

交易DAG的数据结构如下图所示：



Vertex类为最基础里的类型，在交易DAG中，每一个Vertex实例都表征一笔交易。Vertex类包含：

- **inDegree**：表示该顶点的入度
- **outEdges**：用于存储该节点的出边信息，即所有出边所连顶点的ID列表

DAG类用于对DAG的顶点与边关系进行封装，并提供操作DAG的接口，其包含：

- **vtxs**：Vertex数组
- **topLevel**：包含所有入度为0的顶点的队列，由于在执行过程中topLevel会动态变化且会被多个线程访问，因此其需要一个能够支持线程安全访问的容器
- **void init(int32_t size)接口**：根据传入的size初始化一个包含相应数量顶点的DAG结构
- **addEdge(ID from, ID to)接口**：用于在顶点from和顶点to之间建立边关系，具体地说，将顶点to的ID加入顶点from的outEdges中
- **void generate()接口**：当所有的边关系录入完毕后，调用该方法以初始化topLevel成员
- **ID waitPop()接口**：从topLevel中获取一个入度为0的顶点ID

TxDAG类是DAG类更上一层的封装，是DAG与交易之间建立联系的桥梁，其包含：

- **dag**：持有的DAG类实例
- **exeCnt**：已执行过的交易总数
- **totalTxs**：交易总数
- **txs**：区块中的交易列表

2. 交易DAG的构造流程

DAG构造器在构造交易DAG时，会首先将totalTxs成员的值设置为区块中的交易总数，并依据交易总数对dag对象进行初始化，dag会在vtxs中为每笔交易生成一个位置关系一一对应的顶点实例。随后，初始化一个空的资源映射表criticalFields，并按序逐个扫描每笔交易。

对于某笔交易tx，DAG构造器会在其解析出该交易的所有依赖项，对于每个依赖项均会去criticalFields中查询，如果对于某个依赖项d，有前序交易也依赖于该依赖项，则在这两笔交易间建边，并更新criticalFields中d的映射项为tx的ID。

交易DAG构造流程的伪代码如下所示：

```
criticalFields ← map<string, ID>();
totalTxs ← txs.size();
dag.init(txs.size());
for id ← 0 to txs.size() by 1 do
```

(continues on next page)

(续上页)

```
  tx ← txs[id];
  dependencies ← 解析出tx的依赖项;
  for d in dependencies do
    if d in criticalFields then
        dag.addEdge(id, criticalFields[d]);
      end
    criticalFields[d] = id;
    end
  end
end
dag.generate();
```

3.交易DAG的执行流程

PTE在被创建时，会根据配置生成一个用于执行交易DAG的工作线程池，线程池的大小默认等于CPU的逻辑核心数，此线程池的生命周期与PTE的生命周期相同。工作线程会不断调用dag对象的waitPop方法以取出入度为0的就绪交易并执行，执行后该交易的所有后序依赖任务的入度减1，若有交易的入度被减至0，则将该交易加入到topLevel中。循环上述过程，直到交易DAG执行完毕。

交易DAG执行流程的伪代码如下所示：

```
while exeCnt < totalTxs do
  id ← dag.waitPop();
  tx ← txs[id];
  执行tx;
  exeCnt ← exeCnt + 1;
  for txID in dag.vtxs[id].outEdges do
    dag.vtxs[txID].inDegree ← dag.vtxs[txID].inDegree - 1;
    if dag.vtxs[txID].inDegree == 0 then
      dag.topLevel.push(txID)
    end
  end
end
```

性能测评

我们选用了2个基准测试程序，用以测试PTE给FISCO BCOS的性能带来了怎样的变化，它们分别是基于预编译框架实现的转账合约和基于Solidity语言编写的转账合约，两份合约代码的路径分别为：

FISCO-BCOS/libprecompiled/extension/DagTransferPrecompiled.cpp

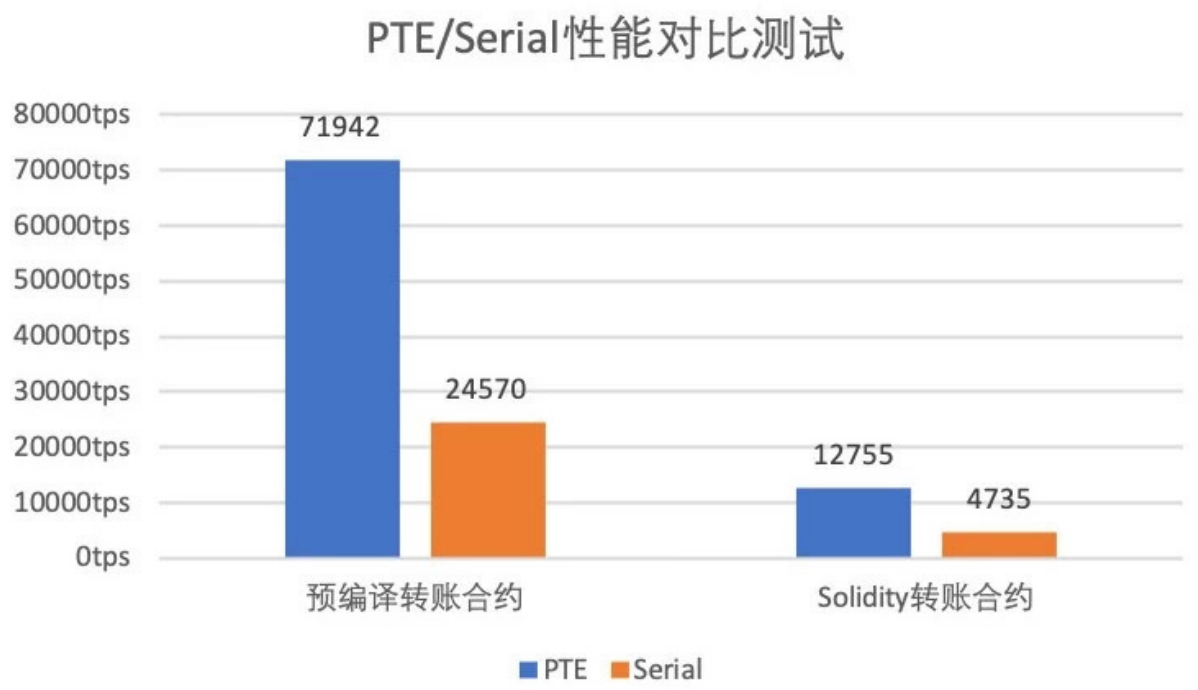
web3sdk/src/test/resources/contract/ParallelOk.sol

我们使用一条单节点链进行测试，因为我们主要关注PTE的交易处理性能，因此并不考虑网络、存储的延迟带来的影响。

测试环境的基本硬件信息如下表所示：

CPU	Intel(R) Xeon(R) Gold 61xx CPU@2.0GHz * 4
内存	8GB DRAM
硬盘	160GB 7200rpm
网络带宽	1Gbps

1.性能测试



性能测试部分，我们主要测试PTE和串行交易执行方式（Serial）在各个测试程序下的交易处理能力。可以看到，相对于串行执行方式，PTE从左至右分别实现了2.91和2.69倍的加速比。无论是对于预编译合约还是Solidity合约，PTE均有着不俗的性能表现。

2.可扩展性测试

可扩展性测试部分，我们主要测试PTE在不同CPU核心数下的交易处理能力，使用的基准测试程序是基于预编译框架实现的转账合约。可以看到，随着核数增加，PTE的交易吞吐量呈近似线性递增。但是同

时也能看到，随着核数在增加，性能增长的幅度在放缓，这是因为随着核数增加线程间调度及同步的开销也会增大。

写在最后

从列表到DAG，PTE赋予了FISCO BCOS新的进化。更高的TPS会将FISCO BCOS带至更加广阔的舞台。予以长袖，FISCO BCOS必能善舞！

《新摩登时代》：卓别林演绎共识与同步流程优化

作者：石翔 | FISCO BCOS 核心开发者

共识与同步的流程优化，是FISCO BCOS性能优化迈开的第一步。**仅依靠这一流程优化，就给系统TPS带来可观的1.75倍提升。但这不是目的，其目的在于确定了共识的主导地位，排除了同步给共识带来的性能影响，让之后的性能分析更好地聚焦在共识流程中。**

基础讲解

在卓别林的电影《摩登时代》里，卓别林扮演一个工人，日复一日地重复着拧螺丝的动作：提起扳手，找到零件，对准螺丝，拧紧，再提起扳手，再找到下一个零件，再对准螺丝，再拧紧..... 在FISCO BCOS的设计里，共识和同步也在干着这样重复的事情。

共识与同步是个啥?

共识与同步，是FISCO BCOS节点中的两个核心流程。它们相互配合，实现了区块链的核心功能：生产出一条在每个节点上都一致的区块链。在FISCO BCOS节点的实现里，共识和同步的实体，我们称为共识模块和同步模块。

- **共识模块**：负责生产区块，让节点产生的区块都是一模一样的
- **同步模块**：负责广播交易，让用户发出的交易尽可能地到达每个节点

共识与同步在干啥?

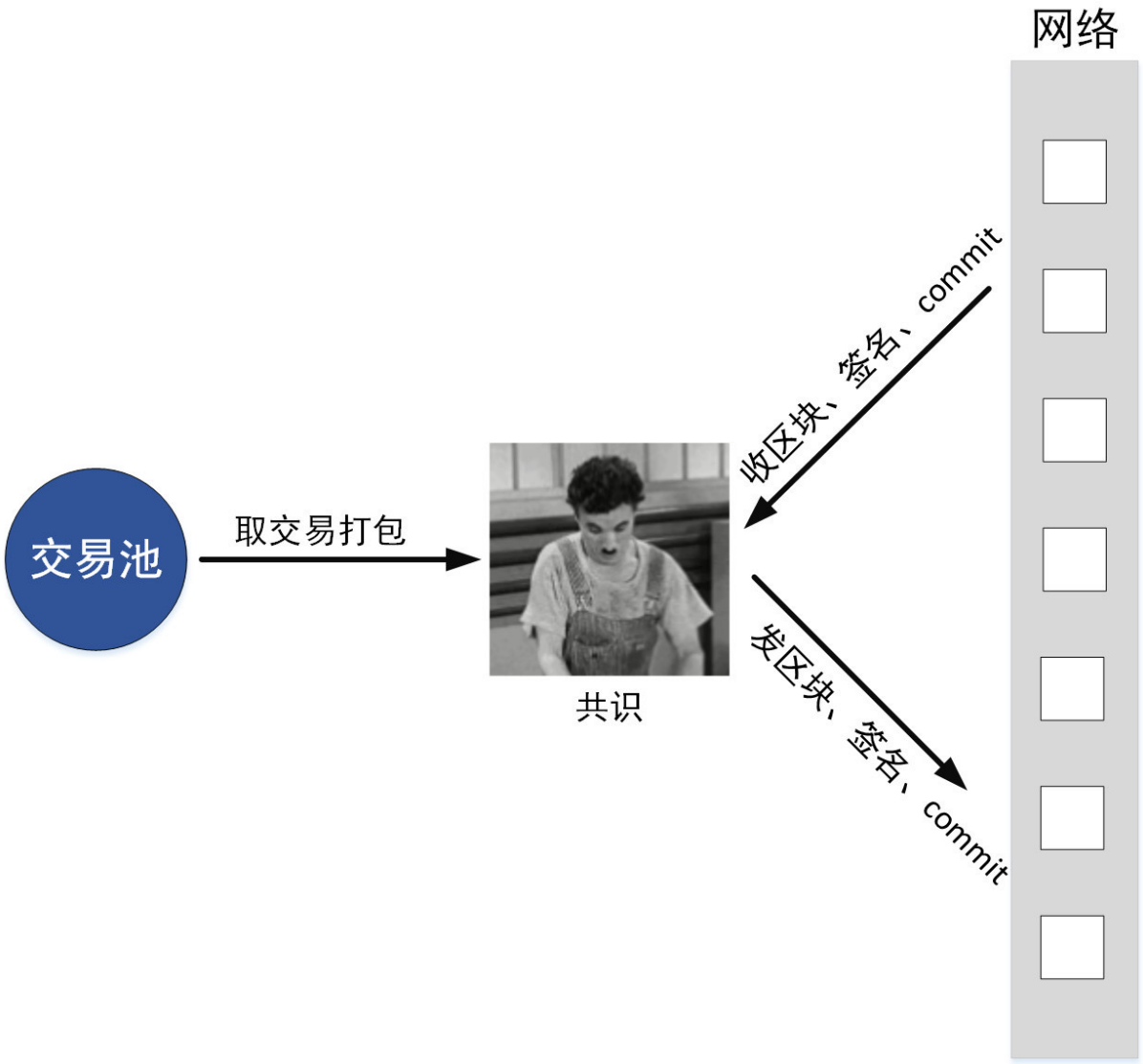
我们来看看共识模块和同步模块的工作环境：

- **交易池**：节点中缓存未被处理交易的容器
- **网络模块**：接收其它节点发来的消息包，也可向其它节点发送消息包 那么，共识模块与同步模块在干啥?

共识模块

不断进行共识消息的处理和发送，让所有节点上的区块达到一致，此处以PBFT的共识为例。

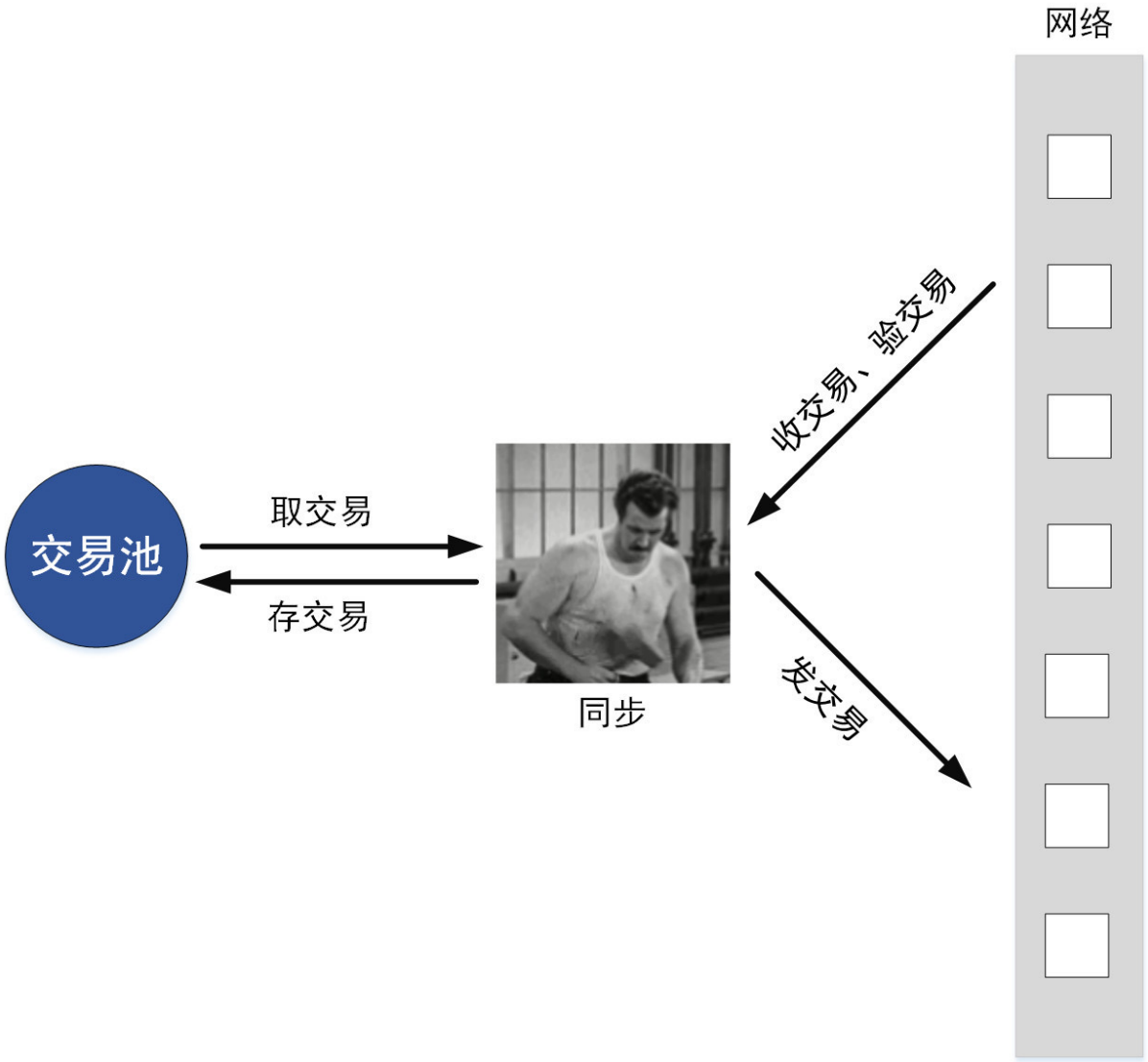
1. **打包区块**：从交易池拿取出交易，打包成区块广播出去，或处理从网络模块拿到的其它节点的区块
2. **执行区块**：解码区块，验证区块，执行区块，将区块的执行结果签个名广播出去
3. **收集签名**：收集其它节点执行结果的签名，如果收集到的签名达到一定数量，就向其它节点广播“commit消息”
4. **收集commit**：收集其它节点的commit消息，当收集到的commit消息达到一定数量，说明区块已经一致，可以落盘了
5. **落盘**：把区块连接到现有区块链的末端，形成区块链，存储到DB中



同步模块

不断进行交易的收发，让每一笔交易尽可能地到达每个节点。

1. **取交易**：从交易池中取出未发送的交易
2. **发交易**：把未发送的交易广播给其它节点
3. **收交易**：从网络模块收取其它节点的交易
4. **验交易**：对交易进行解码和验签
5. **存交易**：把验签通过的交易存到交易池中



问题与优化

卓别林和他的伙伴各司其职，井然有序，看似非常和谐。可当工厂落后的生产力跟不上旺盛的市场需求，即便是卓别林这种熟练工，加班加点也干不完。这个时候，卓别林不得不开始思考自己和伙伴在生产关系上的问题。

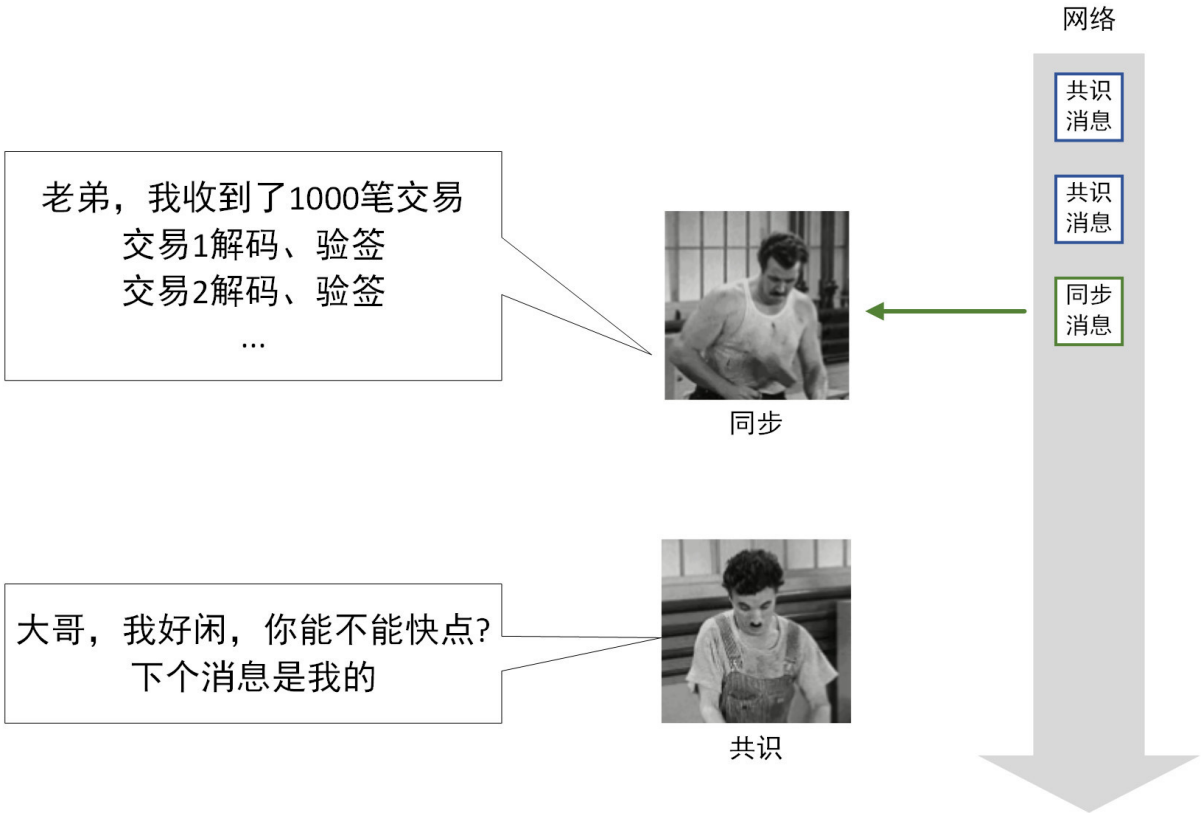
在以往的设计中，共识模块和同步模块并没有优先级的区分，导致它们在争夺资源时浪费了大量的时间。同时，共识模块和同步模块中还有很多重复的操作，也浪费了时间。因此，应该将共识模块和同步模块的执行流程一并考虑，优化流程，提高效率。在经过详细分析和缜密验证后，FISCO BCOS将共识模块和同步模块流程进行了优化。优化基于以下思想:

共识模块负责主导整个区块链出块的节奏，应让共识模块先行。而同步模块，理应扮演好配合的角色，辅佐共识模块更快出块。

基于上述思想，我们来看看其中几个问题的优化方法。

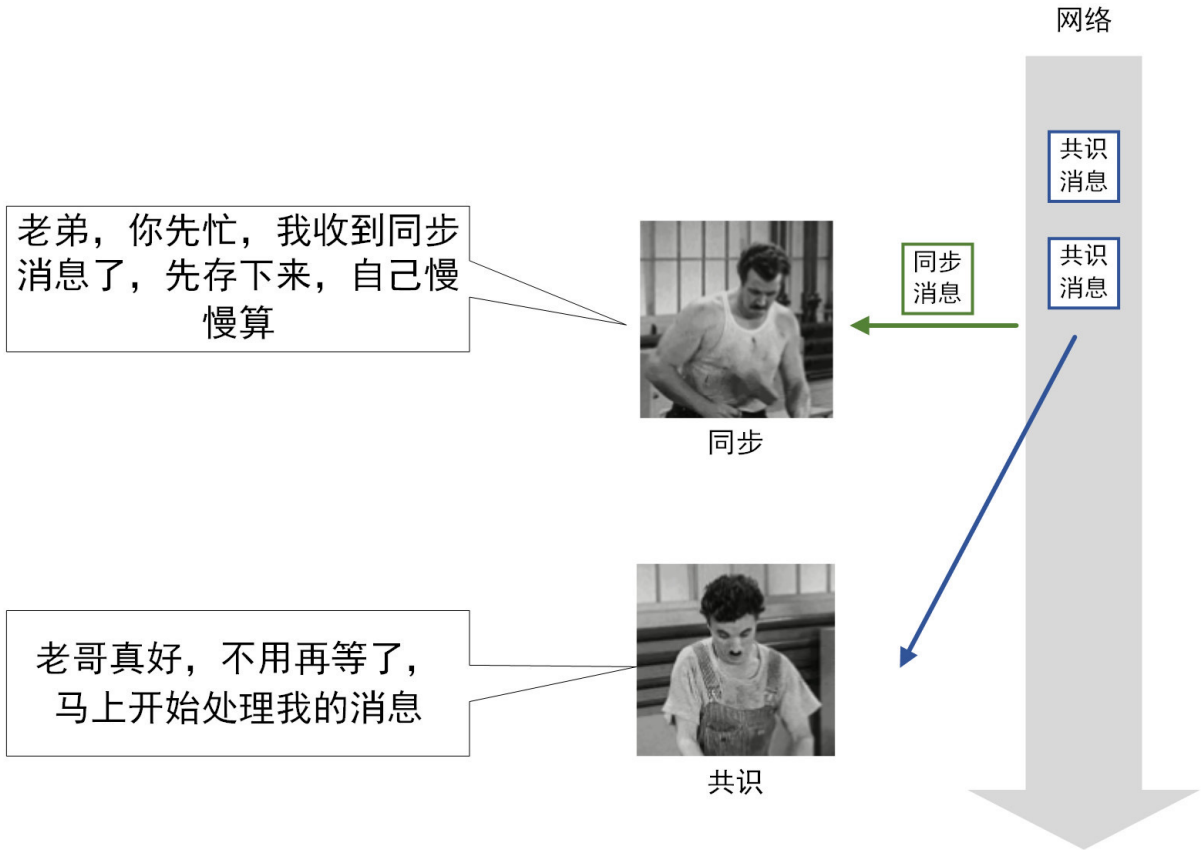
问题1：工作阻塞

共识模块与同步模块都是从网络模块中获取消息包，再根据相应消息包进行下一步操作。但是，由于网络回调线程数量的限制，同步模块在处理消息包时，占用了网络的回调线程，导致共识模块无法及时处理其它节点发来的共识消息，共识流程被阻塞。



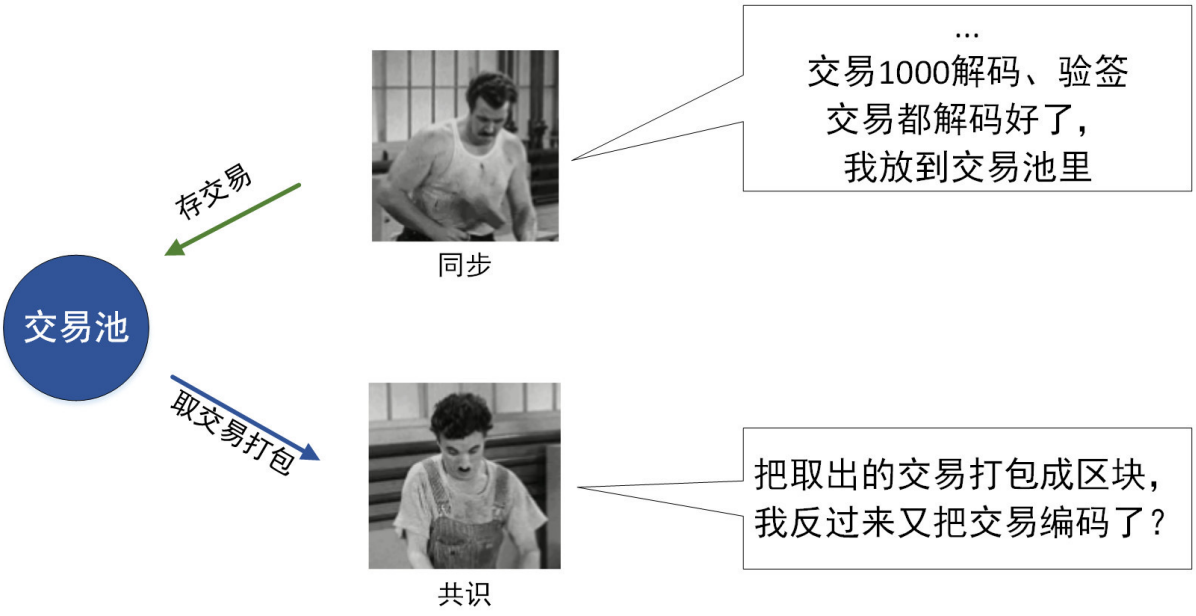
如何解决？——将同步消息的处理操作从网络回调线程中剥离

基于共识模块先行的思想，应让共识模块更及时地收到共识消息，不能让同步模块占用网络回调线程太久。因此，同步模块在拿到消息时，不是直接在回调线程中对交易进行解码和验签，而是将同步消息包缓存起来，用另外一个线程“私下”慢慢处理。这样一来，同步消息的处理不会长时间占用网络回调线程，能让共识消息响应得更快。



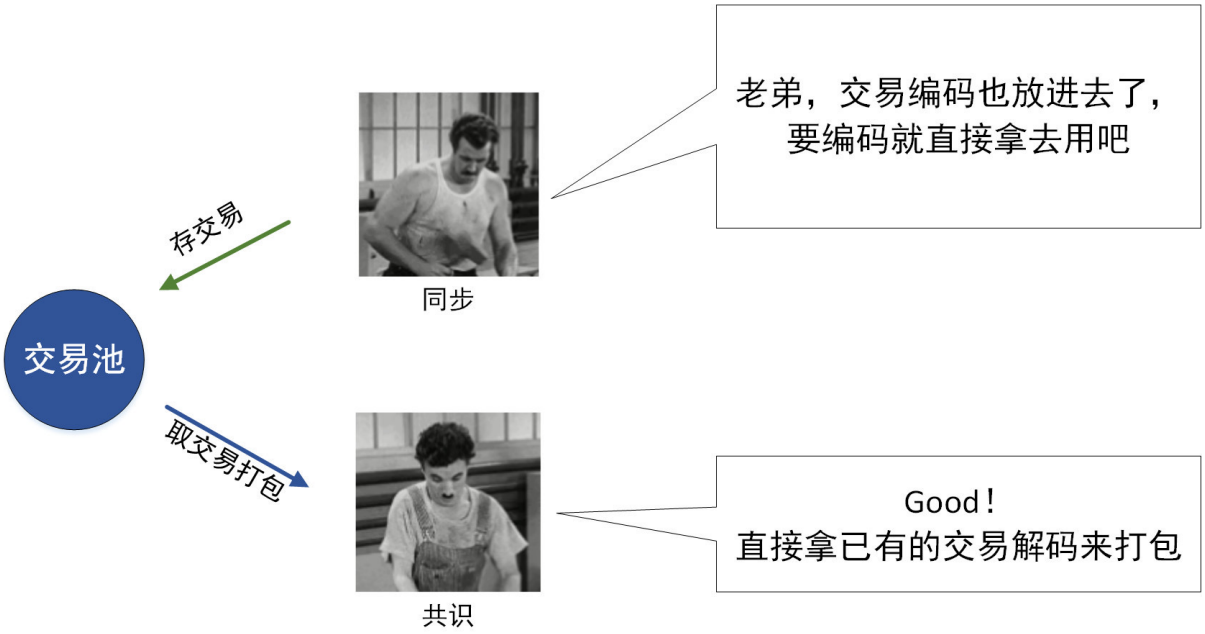
问题2：编解码冗余

同步模块收到同步消息中的交易，是经过编码的，同步模块需将其解码成节点代码中的数据结构，再存入交易池中。共识模块打包区块时，把交易从交易池中取出，将交易进行编码，打包成区块，再将区块发送出去。在这个过程中，交易先被解码，又被编码，操作存在冗余。



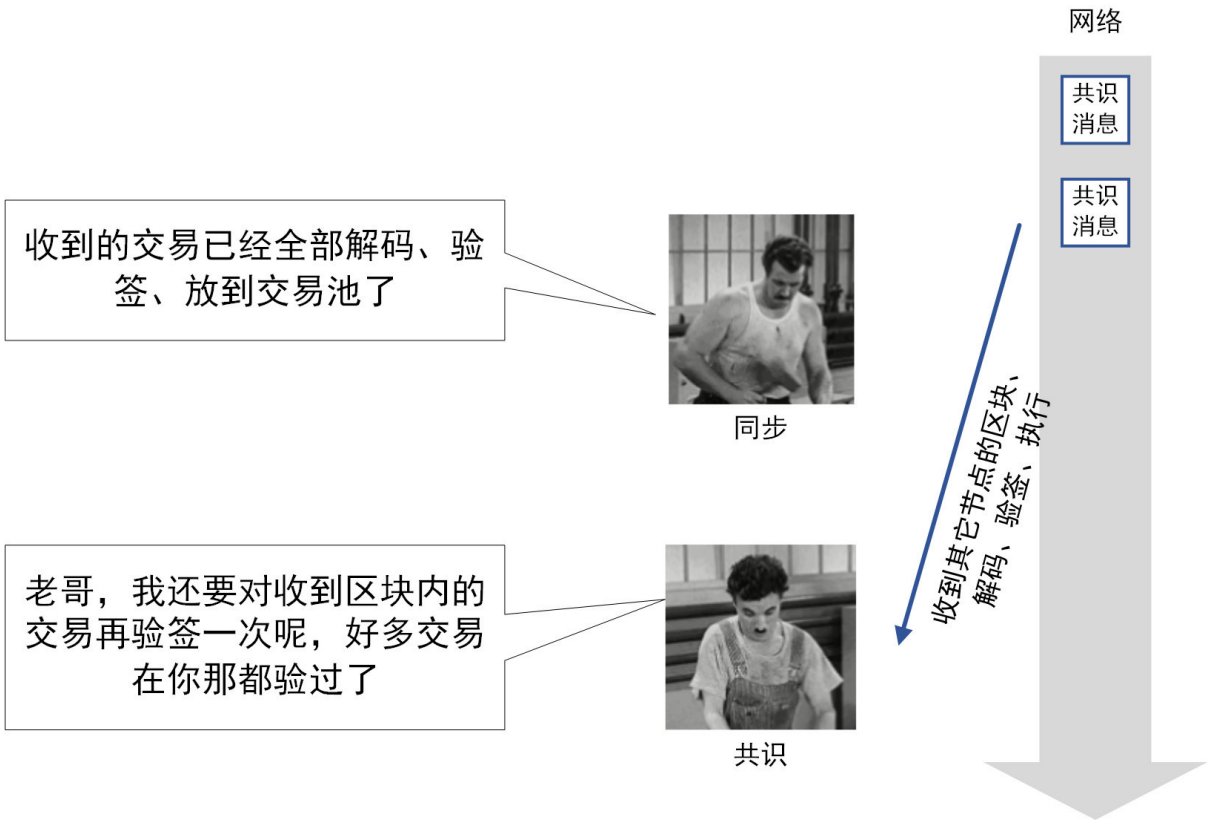
如何解决？——交易编码缓存

共识优先级高于同步，应尽量减少共识模块中不必要的操作。因此，在同步模块存交易时，一并将交易的编码存入交易池。共识模块取交易时，直接从交易池中拿出编码好的交易，免去了编码操作。



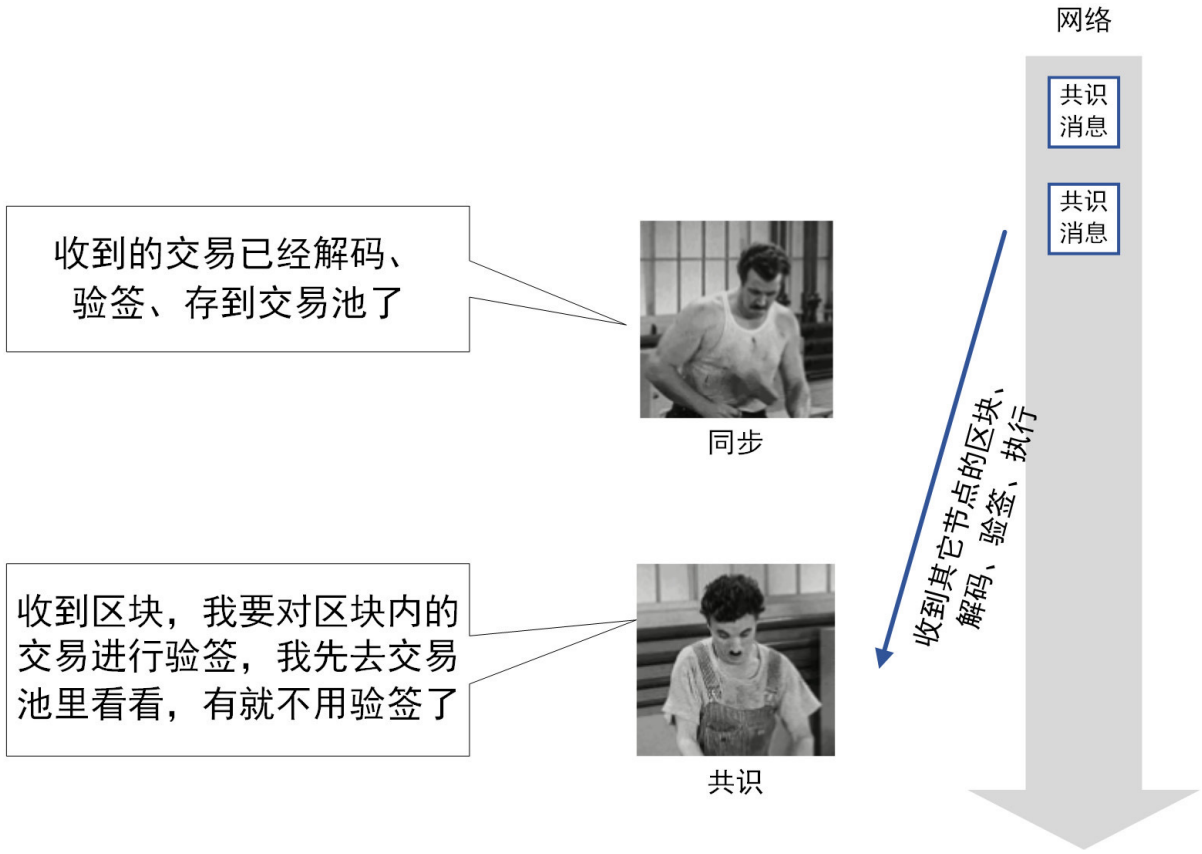
问题3：重复验签

同步模块在收到交易后，需对交易的签名进行验证（简称“验签”），共识模块在收到区块后，也需要对区块中的交易进行验签。同步模块和共识模块所验签的交易，有很大概率是重复的。验签是一个非常耗时的操作，每进行一次额外的验签，都会消耗大量的时间。



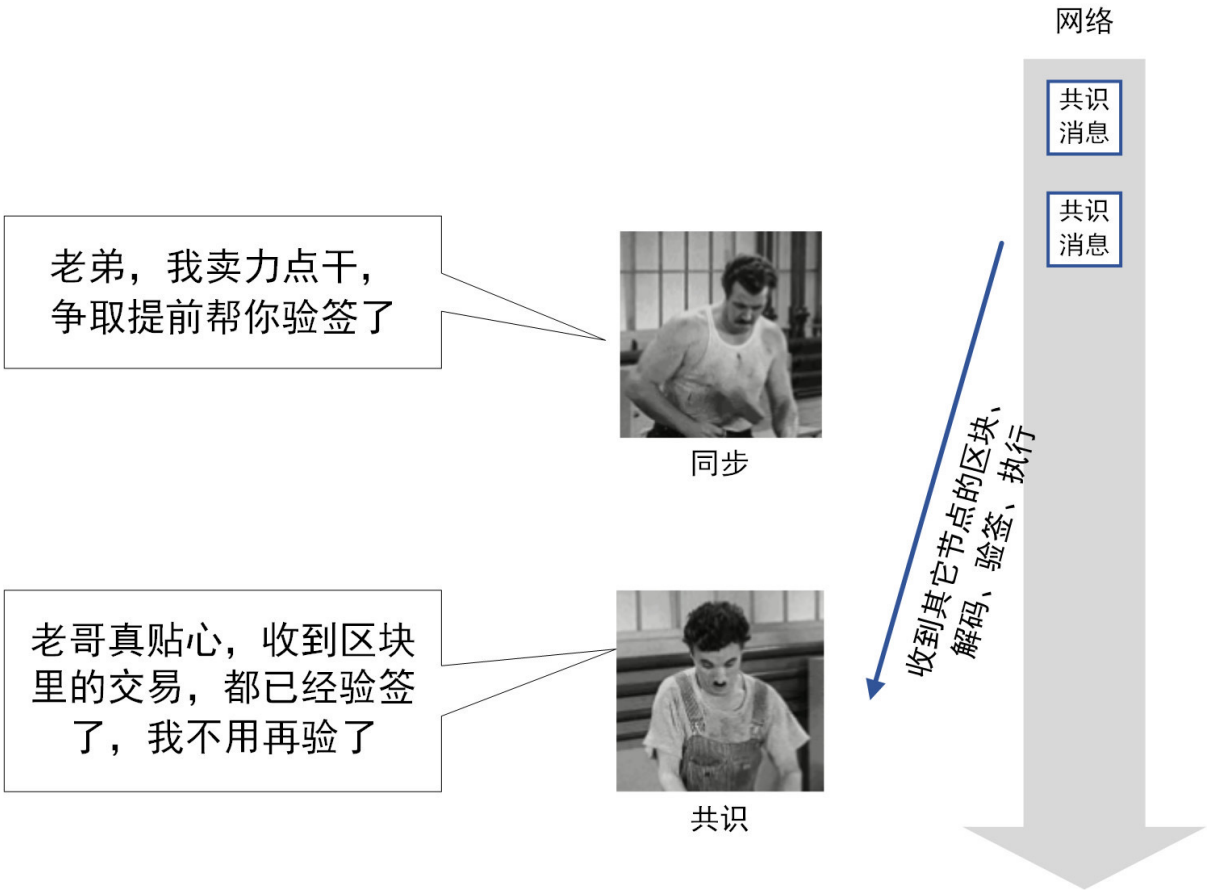
如何解决？——验签去重

无论是同步模块还是共识模块，在验签前，都去交易池里查询该笔交易是否存在。如果存在，就省略验签操作。如此一来，一笔交易只验签一次，减少了不必要的验签开销。



解决方案可否更优？——尽量让同步验签，减少共识模块验签的次数

仍然是共识模块优先的思想，尽量减少共识模块验签的操作。因此，同步模块必须比共识模块跑得快，在共识模块处理一笔交易前，同步模块先拿到交易，优先对交易验签。



FISCO BCOS在此处对同步模块采取的策略是：**对交易进行全量的广播**。

在一个打包节点拿到交易时，其它节点的同步模块也收到了相应的交易。在其它节点收到打包节点发过去的区块时，区块中所包含的交易早已被同步模块验签后写入交易池中。同时，为了让同步模块在相同操作的处理速度上不低于共识模块，同步模块的交易编解码，也采用了和共识模块一样的"并行编解码"和"交易编码缓存"。

结果如何?

共识与同步的流程优化，一定程度上也提高了交易处理的TPS。经测试，交易处理的TPS提高至原来的1.75倍！更重要的是，通过流程优化，确定了共识的主导地位，排除了同步给共识带来的性能影响，让之后的性能分析更好地聚焦在共识流程中！

消除了阻塞，消除了编码冗余，消除了重复验签，卓别林和他的伙伴工作得更轻松，更顺畅了！

下篇文章，我们将集中阐述并行优化，让可并行的操作都并行起来！敬请期待《**全方位的并行处理**》。

让木桶没有短板，FISCO BCOS全面推进并行化改造

作者：李陈希 | FISCO BCOS 核心开发者

背景

PTE（Parallel Transaction Executor，一种基于DAG模型的并行交易执行器）的引入，使FISCO BCOS具备了并行执行交易的能力，显著提升了节点交易处理的效率。对这个阶段性结果，我们并不满足，继续深入挖掘发现，FISCO BCOS的整体TPS仍有较大提升空间。用木桶打个比方：如果参与节点的交易处理所有模块构成木桶，交易执行只是组成整个木桶的一块木板，根据短板理论，一只木桶能盛多少水取决于桶壁上最矮的那块，同理，**FISCO BCOS的性能也由速度最慢的组件决定**。尽管PTE取得了理论

上极高的性能容量，但是FISCO BCOS的整体性能仍然会被其他模块较慢的交易处理速度所掣肘。**为了能够最大化利用计算资源以进一步提高交易处理能力，在FISCO BCOS中全面推进并行化改造势在必行。**

数据分析

根据并行程序设计的『分析→分解→设计→验证』四步走原则，首先需定位出系统中仍存在的性能瓶颈的精确位置，才能更深入地对任务进行分解，并设计相应的并行化策略。**使用自顶向下分析法，我们将交易处理流程分为四个模块进行性能分析**，这四个模块分别是：

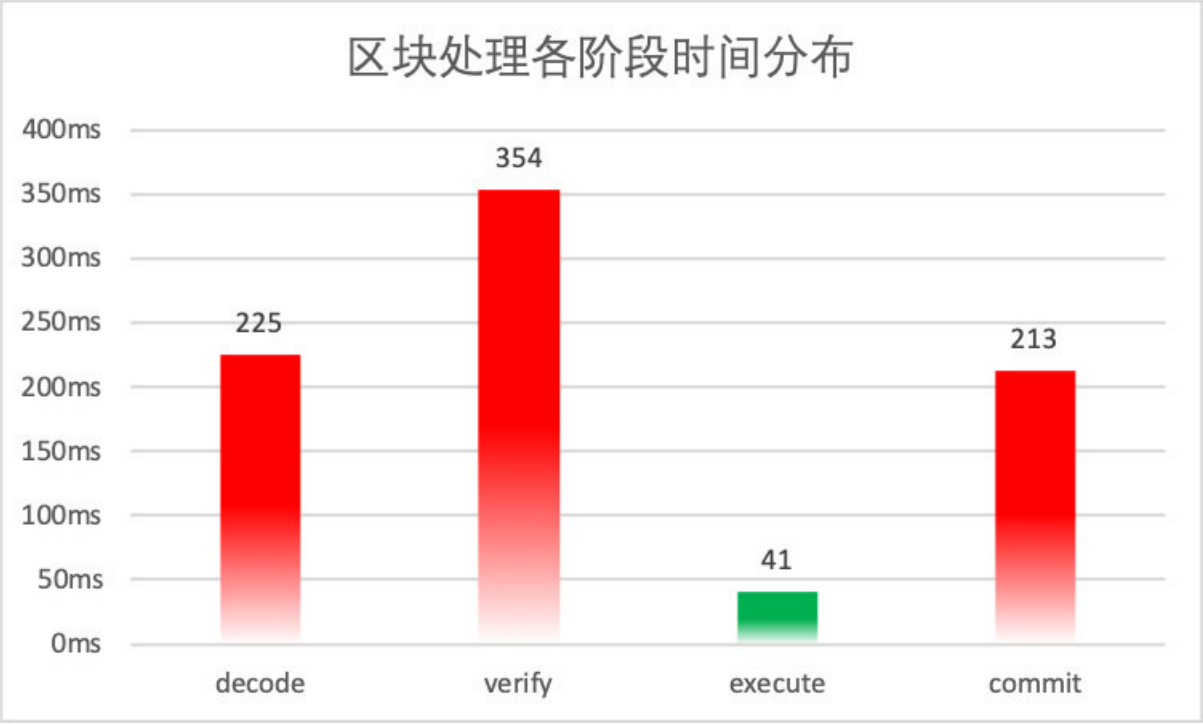
区块解码（decode）：区块在节点间共识或同步时需要从一个节点发送至另一个节点，这个过程中，区块以RLP编码的形式在网络间传输。节点收到区块编码后，需要先进行解码，将区块还原为内存中的二进制对象，然后才能做进一步处理。

交易验签（verify）：交易在发送之前由发送者进行签名，签名得到的数据可以分为(v, r, s)三部分，验签的主要工作便是在收到交易或交易执行前，从(v, r, s)数据中还原出交易发送者的公钥，以验证交易发送者的身份。

交易执行（execute）：执行区块中的所有交易，更新区块链状态。

数据落盘（commit）：区块执行完成后，需要将区块及相关数据写入磁盘中，进行持久化保存。

以包含2500笔预编译转账合约交易的区块为测试对象，在我们的测试环境中，各阶段的平均耗时分布如下图所示：



从图中可以看出，2500笔交易的执行时间已经被缩短到了50毫秒以内，可以证明PTE对FISCO BCOS交易执行阶段的优化是行之有效的。但图中也暴露出了非常明显的问题：其他阶段的用时远远高于交易执行的用时，导致交易执行带来的性能优势被严重抵消，PTE无法发挥出其应有的价值。

早在1967年，计算机体系结构领域的元老Amdahl提出的以他名字命名的定律，便已经向我们阐明了衡量处理器并行计算后效率提升能力的经验法则：

$$SpeedUp \leq \frac{W_S + W_p}{W_S + \frac{W_p}{N}}$$

其中，SpeedUp为加速比，Ws是程序的串行分量，Wp是程序中的并行分量，N为CPU数量。可以看出，在工作总量恒定的情况下，可并行部分代码占比越多，系统的整体性能越高。我们需要把思维从线性模

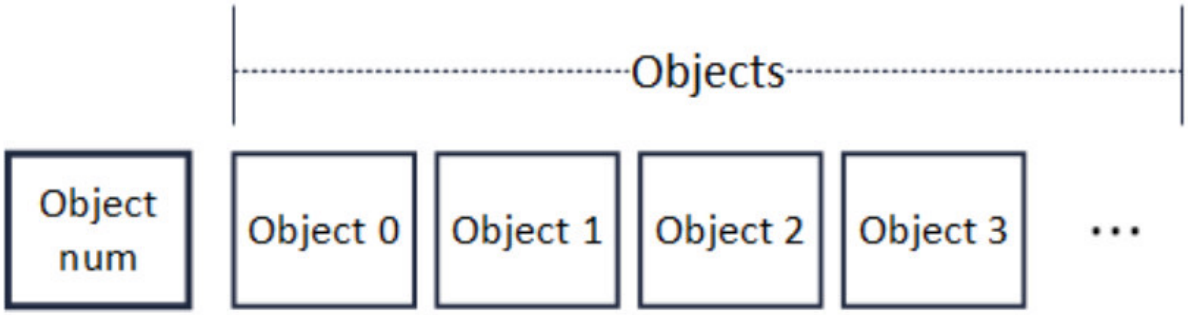
型中抽离出来，继续细分整个处理流程，找出执行时间最长的程序热点，对这些代码段进行并行化从而将所有瓶颈逐个击破，这才是使通过并行化获得最大性能提升的最好办法。

根因拆解

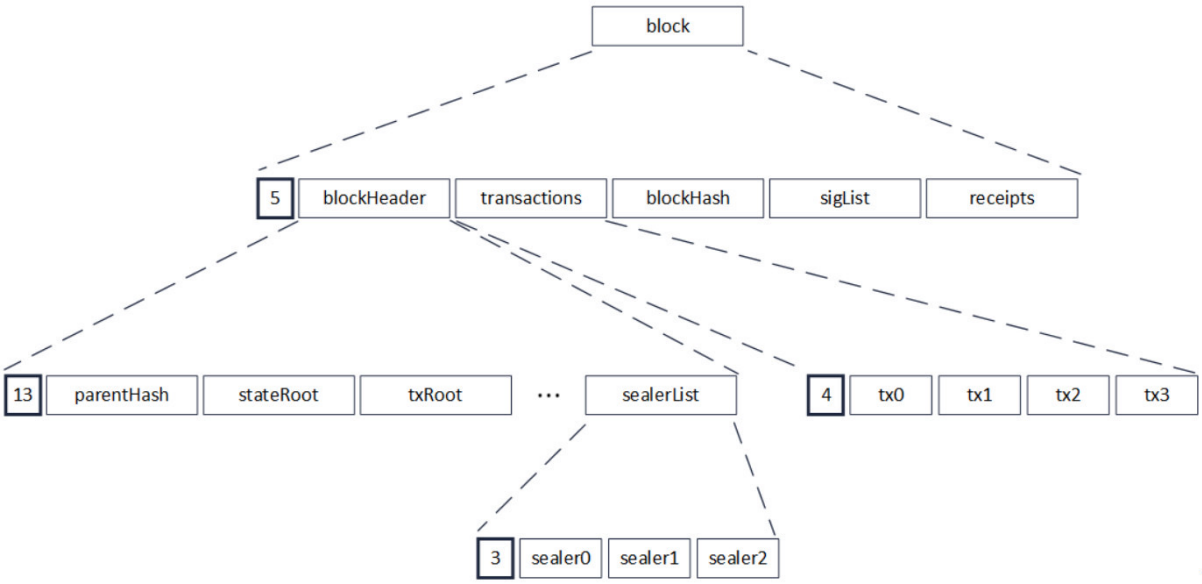
1.串行的区块解码

区块解码主要性能问题出在RLP编码方法本身。RLP全称是递归的长度前缀编码，是一种用长度作为前缀标明编码对象中元素个数的编码方法。如下图所示，RLP编码的开头即是此编码中的对象个数（Object num）。在个数后，是相应个数的对象（Object）。递归地，每个对象，也是RLP编码，其格式也与下图相同。

需要特别注意的是，在RLP编码中。每个Object的字节大小是不固定的，Object num只表示Object的个数，不表示Object的字节长度。



RLP通过一种长度前缀与递归结合的方式，理论上可编码任意个数的对象。下图是一个区块的RLP编码，在对区块进行编码时，先递归至最底层，对多个sealer进行编码，多个sealer被编码并加上长度前缀后，编码成为一串RLP编码（sealerList），此编码又作为一个对象，被编入上层的一串RLP编码（blockHeader）中。此后层层递归，最后编码成为区块的RLP编码。由于RLP编码是递归的，在编码前，无法获知编码后的长度。



解码时，由于RLP编码中每个对象的长度不确定，且RLP编码只记录了对象的个数，没记录对象的字节长度，若要获取其中的一个编码对象，必须递归解码其前序的所有对象，在解码前序的对象后，才能访问到需要访问的编码对象的字节位置。例如在上图中，若需要访问区块中的第0笔交易，即tx0，必须先将blockHeader解码，而blockHeader的解码，需要再次递归，把parentHash，stateRoot直至sealerList都解码出来。

解码区块最重要的目的是解码出包含在区块中的交易，而交易的编码都是互相独立的，但在RLP特殊的编码方式下，解码一笔交易的必要条件是解码出上一笔交易，交易的解码任务之间环环相扣，形成了一种链式的依赖关系。需要指出的是，这种解码方式并不是RLP的缺陷，RLP的设计目标之一本就是尽量减少空间占用，充分利用好每一个字节，虽然编解码变得低效了些，但编码的紧凑度却是有目共睹，因此这种编码方式本质上还是一种时间换空间的权衡结果。

由于历史原因，FISCO BCOS中使用了RLP编码作为多处信息交换协议，贸然换用其他并行化友好的序列化方案可能会带来较大的开发负担。基于这一考虑，我们决定在原有的RLP编解码方案稍作修改，通过为每个被编码的元素添加额外的位置偏移信息，便可以做到并行解码RLP的同时不会改动大量原有代码。

2.交易验签&数据落盘开销大

通过对交易验签和数据落盘部分的代码进行拆解，我们发现两者的主要功能都集中在一个耗时巨大的for循环。交易验签负责按序取出交易，然后从交易的签名数据中取出(v, r, s)数据，并从中还原出交易发送者的公钥，其中，还原公钥这一步，由于涉及密码学算法，因此耗时不少；数据落盘负责从缓存中逐个取出交易相关数据，将其编码为JSON字符串后写入磁盘，由于JSON编码过程本身效率比较低，因此也是性能损失的重灾区。

两者代码分别如下所示：

```
// 交易验签
for(int i = 0; i < transactions.size(); ++i)
{
  tx = transactions[i];
  v, r, s = tx.getSignature();
  publicKey = recover(v, r, s);  // 从(v, r, s)中复原出发送者公钥
  ...
}
```

```
// 数据落盘
for(int i = 0; i < datas.size(); ++i)
{
  data = datas[i];
  jsonStr = jsonEncode(data);  // 将数据编码为JSON字符串进行存储
  db.commit(jsonStr);
  ...
}
```

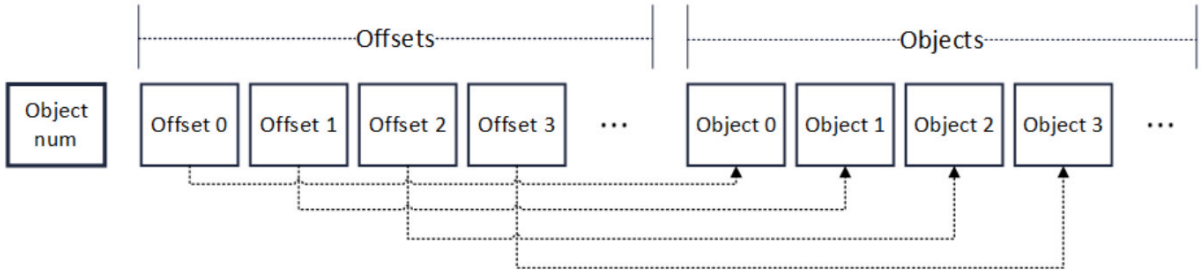
两个过程共有的特点是，它们均是将同样的操作应用到数据结构中不同的部分，对于这种类型的问题，可以直接使用数据级并行进行改造。所谓数据级并行，即是将数据作为划分对象，通过将数据划分为大小近似相等的片段，通过在多个线程上对不同的数据片段上进行操作，达到并行处理数据集的目的。

数据级并行唯一的附加要求是任务之间彼此独立，毫无疑问，在FISCO BCOS的实现中，交易验签和数据落盘均满足这一要求。

优化实践

1.区块解码并行化

改造过程中，我们在系统中使用的普通RLP编码的基础上，加入了offset字段，用以索引每个Object的位置。如下图所示，改造后编码格式的开头，仍然是对象的个数（Object num），但是在个数字段后，是一个记录对象偏移量的数组（Offsets）。



数组中的每个元素有着固定的长度。因此要读取某个Offset的值，只需向访问数组一样，根据Offset的序号直接索引便可以进行随机访问。在Offsets后，是与RLP编码相同的对象列表。相应序号的Offset，指向相应序号的对象的RLP编码字节位置。因此，任意解码一个对象，只需要根据对象的序号，找到其偏移量，再根据偏移量，就可定位到相应对象的RLP编码字节位置。

编码流程也进行了重新设计。流程本身仍然基于递归的思路，对于输入的对象数组，首先将对象数组的大小编码在输出编码的开头处，若数组大小超过1，则按序逐个取出待编码对象并缓存其递归编码，并在Offsets数组中记录该对象的偏移位置，待数组遍历完后，将缓存的对象编码第一次性取出并附加至输出编码末尾；若数组大小为1，则递归对其编码并写入输出编码的末尾，结束递归。

编码流程的伪代码如下：

```
Rlps = RLP();  // Output，初始时为空
void encode(objs)  //Input: objs = 待编码对象的数组
{
  offset = 0;
  codes = [];

  objNum = objs.size()
  Rlps.push(objNum)
  if objNum > 1
  {
    for obj in objs
    {
      rlp = encode(obj);  // 递归调用编码方法
      Rlps.push(offset);
      offset += rlp.size();
      codes.add(rlp);  // 缓存递归编码的结果
    }
    for x in codes
    {
      Rlps.push(x);
    }
  }
  else
  {
    rlp = encode(objs[0]);
    Rlps.push(rlp);
  }
}
```

偏移量的引入使解码模块能够对元素编码进行随机访问。Offsets的数组范围可以在多个线程间均摊，从而每个线程可以并行访问对象数组的不同部分，分别进行解码。由于是只读访问，这种并行方式是线程安全的，仅需最后再对输出进行汇总即可。

解码流程的伪代码如下：

```
Objs decode(RLP Rlps)
{
  objNum = Rlps.objNum;    // 获取对象个数
  outputs = []  // 输出的对象数组
  if objNum > 1
```

(continues on next page)

(续上页)

```
  {
    parallel for i = 0 to objNum
    {
      offset = Rlps.offsets[i];
      code = Rlps.objs[offset];
      x = decode(code);
      outputs.add(x);    // 有序插入outputs
    }
  }
  else
  {
    outputs.add(decode(Rlps.objs[0]));
  }
  return outputs;
}
```

2.交易验签 & 数据落盘并行化

对于数据级并行，业内已有多种成熟的多线程编程模型。虽然Pthread这类显式的多线程编程模型能够提供对线程进行更精细的控制，但是需要我们对线程通信、同步拥有娴熟的驾驭技巧。实现的复杂度越高，犯错的几率越大，日后代码维护的难度也相应增加。我们的主要目标仅仅对密集型循环进行并行化，因此在满足需求的前提下，Keep It Simple & Stupid才是我们的编码原则，因此我们使用隐式的编程模型来达成我们的目的。

经过再三权衡，我们在市面上众多隐式多线程编程模型中，选择了来自Intel的线程构建块（Thread Building Blocks，TBB）开源库。在数据级并行方面，TBB算是老手，TBB运行时系统不仅屏蔽了底层工作线程的实现细节，还能够根据任务量自动在处理器间平衡工作负载，从而充分利用底层CPU资源。

使用TBB后，交易验签和数据落盘的代码如下所示：

```
// 并行交易验签
tbb::parallel_for(tbb::blocked_range<size_t>(0, transactions.size()),
[&](const tbb::blocked_range<size_t>& _range)
{
  for(int i = _range.begin(); i != _range.end(); ++i)
  {
    tx = transactions[i];
    v, r, s = tx.getSignature();
    publicKey = recover(v, r, s);  // 从(v, r, s)中复原出发送者公钥
    ...
  }
});
```

```
// 并行数据落盘
tbb::parallel_for(tbb::blocked_range<size_t>(0, transactions.size()),
[&](const tbb::blocked_range<size_t>& _range)
{
  for(int i = _range.begin(); i != _range.end(); ++i)
  {
    data = datas[i];
    jsonStr = jsonEncode(data);  // 将数据编码为JSON字符串进行存储
    db.commit(jsonStr);
    ...
  }
});
```

可以看到，除了使用TBB提供的tbb::parallel_for进行并行循环和tbb::blocked_range引用数据分片外，循环体内的代码几乎没有任何变化，接近C++原生语法正是TBB的特点。TBB提供了抽象层级较高的并行接口，如parallel_for、parallel_for_each这类泛型并行算法，从而使得改造能够较为容易地进行。同

时，TBB不依赖任何语言或编译器，只要有能支持ISO C++标准的编译器，便有TBB的用武之地。

当然，使用TBB并不是完全没有额外负担，比如线程间安全还是需要开发人员的仔细分析来保证，但TBB考虑周到，提供了一套方便的工具来辅助我们解决线程间互斥的问题，如原子变量、线程局部存储和并行容器等，这些并行工具同样被广泛地应用在FISCO BCOS中，为FISCO BCOS的稳定运行保驾护航。

写在最后

经过一套并行优化的组合拳，FISCO BCOS的性能表现更上层楼。压力测试的结果表明，FISCO BCOS的交易处理能力，相较于并行化改造之前，成功提升了1.74倍，基本达到了这个环节的预期效果。

但是我们也深深明白，性能优化之路漫漫，木桶最短的一板总是交替出现，并行之道在于，通过反复的分析、拆解、量化和优化，使得各模块互相配合齐头并进，整个系统达到优雅的平衡，而最优解总是在“跳一跳”才能够得着的地方。

FISCO BCOS性能优化——工具篇

作者：李陈希 | FISCO BCOS 核心开发者

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. ——Donald Knuth

『过早的优化是万恶之源』

说出这句话的计算机科学先驱Donald Knuth并不是反对优化，而是强调要对系统中的关键位置进行优化。假设一个for循环耗时0.01秒，即使使用循环展开等各种奇技淫巧将其性能提升100倍，把耗时降到0.00001秒，对于用户而言，也基本无法感知到。对性能问题进行量化测试之前，在代码层面进行各种炫技式优化，可能不仅提升不了性能，反而会增加代码维护难度或引入更多错误。

『没有任何证据支撑的优化是万恶之源』

在对系统施展优化措施前，一定要对系统进行详尽的性能测试，从而找出真正的性能瓶颈。奋战在FISCO BCOS性能优化的前线上，我们对如何使用性能测试工具来精确定位性能热点这件事积累了些许经验心得。本文将我们在优化过程中使用到的工具进行了整理汇总，以飨读者。

1.Poor Man's Profiler

穷人的分析器，简称PMP。尽管名字有些让人摸不着头脑，但人家真的是一种正经的性能分析手段，甚至有自己的官方网站https://poormansprofiler.org/。PMP的原理是Stack Sampling，通过调用第三方调试器（比如gdb），反复获取进程中每个线程的堆栈信息，PMP便可以得到目标进程的热点分布。

第一步，获取一定数量的线程堆栈快照：

```
pid=$(pidof fisco-bcos)
num=10
for x in $(seq 1 $(num))
  do
    gdb -ex "set pagination 0" -ex "thread apply all bt" -batch -p $pid
    sleep 0.5
done
```

第二步，从快照中取出函数调用栈信息，按照调用频率排序：

```
awk '
  BEGIN { s = ""; }
  /^Thread/ { print s; s = ""; }
  /^\#/ { if (s != "" ) { s = s "," $4} else { s = $4 } }
```

(continues on next page)

(续上页)

```
  END { print s }' | \
sort | uniq -c | sort -r -n -k
```

最后得到输出，如下图所示：

从输出中可以观察到哪些线程的哪些函数被频繁采样，进而可按图索骥找出可能存在的瓶颈。上述寥寥数行shell脚本便是PMP全部精华之所在。极度简单易用是PMP的最大卖点，除了依赖一个随处可见的调试器外，PMP不需要安装任何组件，正如PMP作者在介绍中所言：『**尽管存在更高级的分析技术，但毫无例外它们安装起来都太麻烦了……Poor man doesn't have time. Poor man needs food.**。

PMP的缺点也比较明显：gdb的启动非常耗时，限制了PMP的采样频率不能太高，因此一些重要的函数调用事件可能会被遗漏，从而导致最后的profile结果不够精确。但是在某些特殊场合，PMP还是能发挥作用的，比如在一些中文技术博客中，就有开发人员提到使用PMP成功定位出了线上生产环境中的死锁问题，PMP作者也称这项技术在Facebook、Intel等大厂中有所应用。不管怎样，这种闪烁着程序员小智慧又带点小幽默的技术，值得一瞥。

2.perf

perf的全称是Performance Event，在2.6.31版本后的Linux内核中均有集成，是Linux自带的强力性能分析工具，使用现代处理器中的特殊硬件PMU（Performance Monitor Unit，性能监视单元）和内核性能计数器统计性能数据。perf的工作方式是对运行中的进程按一定频率进行中断采样，获取当前执行的函数名及调用栈。如果大部分的采样点都落在同一个函数上，则表明该函数执行的时间较长或该函数被频繁调用，可能存在性能问题。

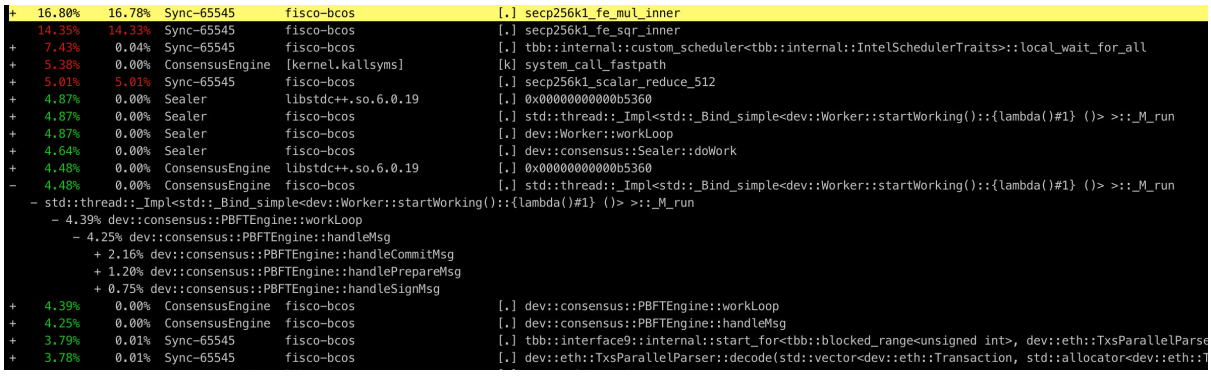
使用perf需要首先对目标进程进行采样：

```
$ sudo perf record -F 1000 -p `pidof fisco-bcos` -g -- sleep 60
```

在上述命令中， 我们使用perf record指定记录性能的统计数据；使用-F指定采样的频率为1000Hz，即一秒钟采样1000次；使用-p指定要采样的进程ID（既fisco-bcos的进程ID），我们可以直接通过pidof命令得到；使用-g表示记录调用栈信息；使用sleep指定采样持续时间为60秒。待采样完成后，perf会将采集到的性能数据写入当前目录下的perf.data文件中。

```
$ perf report -n
```

上述命令会读取perf.data并统计每个调用栈的百分比，且按照从高到低的顺序排列，如下图所示：



信息已足够丰富，但可读性仍然不太友好。尽管示例中perf的用法较为简单，但实际上perf能做的远不止于此。配合其他工具，perf采样出的数据能够以更加直观清晰的方式展现在我们面前，这便是我们接下来要介绍的性能分析神器——火焰图。

3.火焰图

火焰图，即Flame Graph，藉由系统性能大牛 Brendan Gregg提出的动态追踪技术而发扬光大，主要用于将性能分析工具生成的数据进行可视化处理，方便开发人员一眼就能定位到性能问题所在。火焰图的使用较为简单，我们仅需将一系列工具从github上下载下来，置于本地任一目录即可：

```
wget https://github.com/brendangregg/FlameGraph/archive/master.zipunzip master.zip
```

3.1CPU火焰图

当我们发现FISCO BCOS性能较低时，直觉上会想弄清楚到底是哪一部分的代码拖慢了整体速度，CPU是我们的首要考察对象。

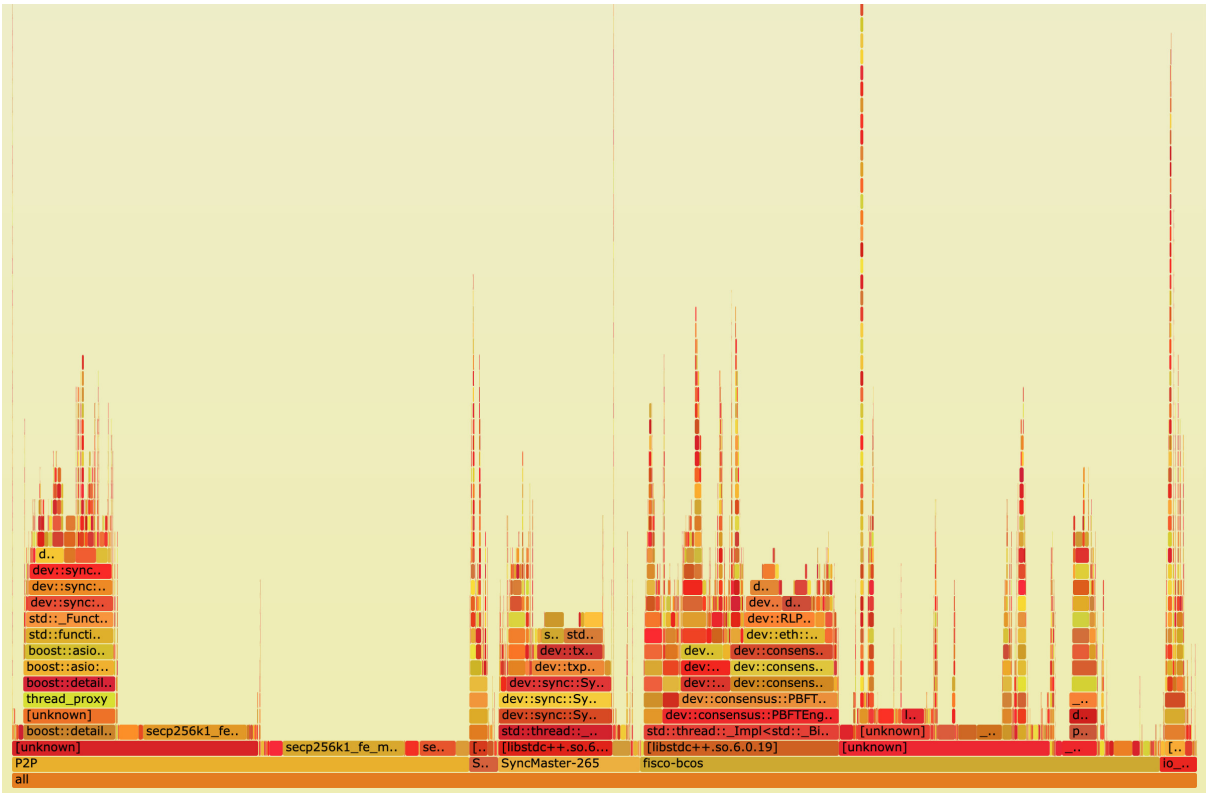
首先使用perf对FISCO BCOS进程进行性能采样：

```
sudo perf record  -F 10000 -p `pidof fisco-bcos` -g -- sleep 60
# 对采样数据文件进行解析生成堆栈信息
sudo perf script > cpu.unfold
```

生成了采样数据文件后，接下来调用火焰图工具生成火焰图：

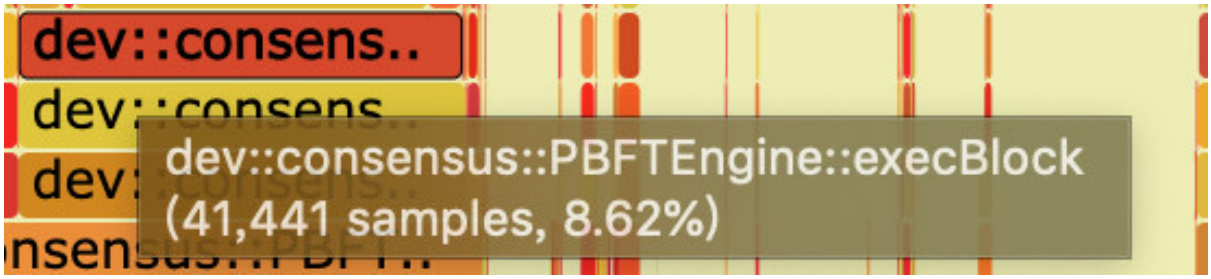
```
# 对perf.unfold进行符号折叠
sudo ./stackcollapse-perf.pl cpu.unfold > cpu.folded
# 生成SVG格式的火焰图
sudo ./flamegraph.pl  cpu.folded > cpu.svg
```

最后输出一个SVG格式图片，用来展示CPU的调用栈，如下图所示：

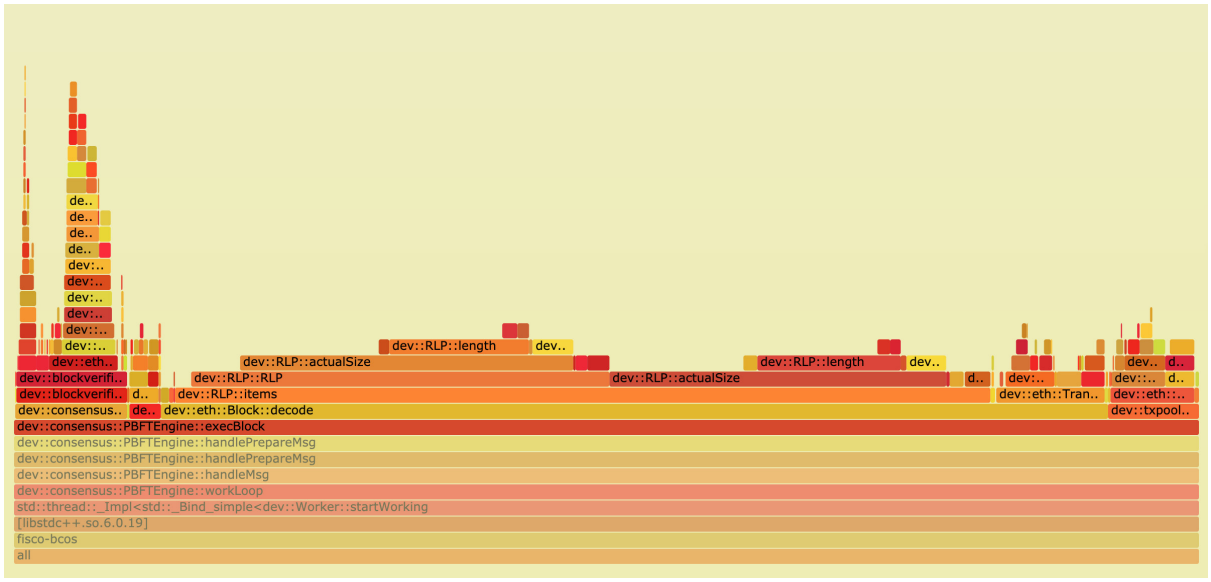


纵轴表示调用栈。每一层都是一个函数，也是其上一层的父函数，最顶部就是采样时正在执行的函数，调用栈越深，火焰就越高。**横轴表示抽样数**。注意，并不是表示执行时间。若一个函数的宽度越宽，则表示它被抽到的次数越多，所有调用栈会在汇总后，按字母序列排列在横轴上。

火焰图使用了SVG格式，可交互性大大提高。在浏览器中打开时，火焰的每一层都会标注函数名，当鼠标悬浮其上，会显示完整的函数名、被抽样次数和占总抽样字数的百分比，如下：



点击某一层时，火焰图会水平放大，该层会占据所有宽度，并显示详细信息，点击左上角的『Reset Zoom』即可还原。下图展示了PBFT模块在执行区块时，各个函数的抽样数占比：



从图中可以看出，在执行区块时，主要开销在交易的解码中，这是由于传统的RLP编码在解码时，RLP编码中每个对象的长度不确定，且RLP编码只记录了对象的个数，没记录对象的字节长度，若要获取其中的一个编码对象，必须递归解码其前序的所有对象。

因此，RLP编码的解码过程是一个串行的过程，当区块中交易数量较大时，这一部分的开销将变得十分巨大。对此，我们提出了一种并行解码RLP编码的优化方案，具体实现细节可以参考上一篇文章《FISCO BCOS中的并行化实践》。

有了火焰图，能够非常方便地查看CPU的大部分时间开销都消耗在何处，进而也能针对性进行优化了。

3.2 Off-CPU火焰图

在实现FISCO BCOS的并行执行交易功能时，我们发现有一个令人困惑的现象：有时即使交易量非常大，区块的负载已经打满，但是通过top命令观察到CPU的利用率仍然比较低，通常4核CPU的利用率不足200%。在排除了交易间存在依赖关系的可能后，推测CPU可能陷入了I/O或锁等待中，因此需要确定CPU到底在什么地方等待。

使用perf，我们可以轻松地了解系统中任何进程的睡眠过程，其原理是利用perf static tracer抓取进程的调度事件，并通过perf inject对这些事件进行合并，最终得到诱发进程睡眠的调用流程以及睡眠时间。

我们要通过perf分别记录sched:sched_stat_sleep、sched:sched_switch、sched:sched_process_exit三种事件，这三种事件分别表示进程主动放弃 CPU 而进入睡眠的等待事件、进程由于I/O和锁等待等原因被调度器切换而进入睡眠的等待事件、进程的退出事件。

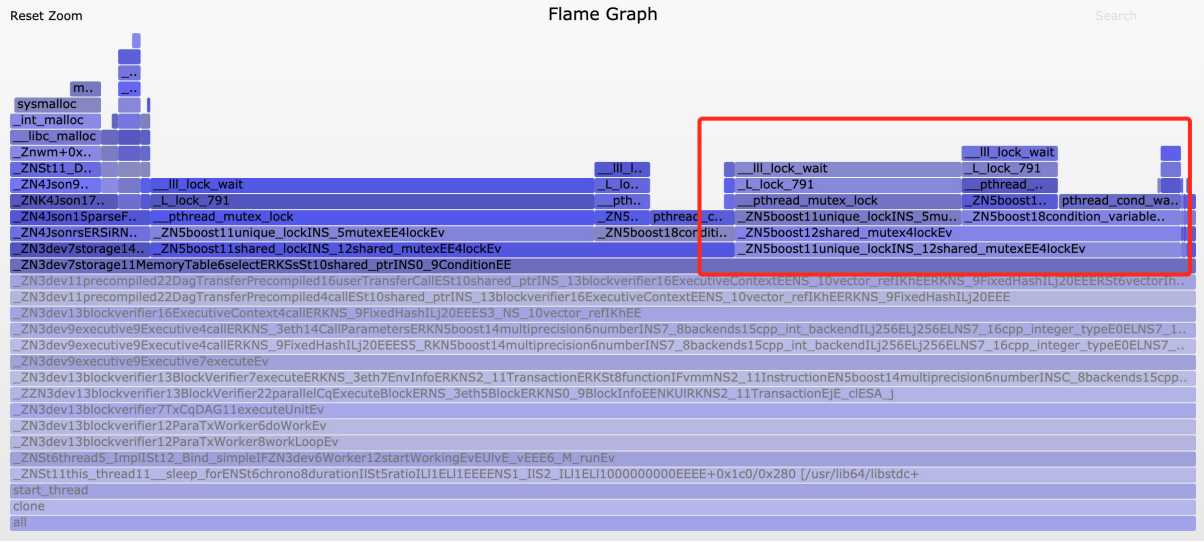
```
perf record -e sched:sched_stat_sleep -e sched:sched_switch \
-e sched:sched_process_exit -p `pidof fisco-bcos` -g \
-o perf.data.raw sleep 60
perf inject -v -s -i perf.data.raw -o perf.data
# 生成Off-CPU火焰图
perf script -f comm,pid,tid,cpu,time,period,event,ip,sym,dso,trace | awk '
    NF > 4 { exec = $1; period_ms = int($5 / 1000000) }
    NF > 1 && NF <= 4 && period_ms > 0 { print $2 }
    NF < 2 && period_ms > 0 { printf "%s\n%d\n\n", exec, period_ms }' | \
./stackcollapse.pl | \
./flamegraph.pl --countname=ms --title="Off-CPU Time Flame Graph" --colors=io >␣
↪offcpu.svg
```

在较新的Ubuntu或CentOS系统中，上述命令可能会失效，出于性能考虑，这些系统并不支持记录调度事件。好在我们可以选择另一种profile工具——OpenResty的SystemTap，来替代perf帮助我们收集进程调度器的性能数据。我们在CentOS下使用SystemTap时，只需要安装一些依赖kenerl debuginfo即可使用。

```
wget https://raw.githubusercontent.com/openresty/openresty-systemtap-toolkit/
↪master/sample-bt-off-cpu
chmod +x sample-bt-off-cpu

./sample-bt-off-cpu -t 60 -p `pidof fisco-bcos` -u > out.stap
./stackcollapse-stap.pl out.stap > out.folded
./flamegraph.pl --colors=io out.folded > offcpu.svg
```

得到的Off-CPU火焰图如下图所示：



展开执行交易的核心函数后，位于火焰图中右侧的一堆lock_wait很快引起了我们的注意。分析过它们的调用栈后，我们发现这些lock_wait的根源，来自于我们在程序中有大量打印debug日志的行为。

在早期开发阶段，我们为了方便调试，添加了很多日志代码，后续也没有删除。虽然我们在测试过程中将日志等级设置得较高，但这些日志相关的代码仍会产生运行时开销，如访问日志等级状态来决定是否打印日志等。由于这些状态需要线程间互斥访问，因此导致线程由于竞争资源而陷入饥饿。

我们将这些日志代码删除后，执行交易时4核CPU的利用率瞬间升至300%+，考虑到线程间调度和同步的开销，这个利用率已属于正常范围。这次调试的经历也提醒了我们，在追求高性能的并行代码中输出日志一定要谨慎，避免由于不必要的日志而引入无谓的性能损失。

3.3 内存火焰图

在FISCO BCOS早期测试阶段，我们采取的测试方式是反复执行同一区块，再计算执行一个区块平均耗时，我们发现，第一次执行区块的耗时会远远高于后续执行区块的耗时。从表象上看，这似乎是在第一

次执行区块时，程序在某处分配了缓存，然而我们并不知道具体是在何处分配的缓存，因此我们着手研究了内存火焰图。

内存火焰图是一种非侵入式的旁路分析方法，相较于模拟运行进行内存分析的Valgrid和统计heap使用情况的TC Malloc，内存火焰图可以在获取目标进程的内存分配情况的同时不干扰程序的运行。

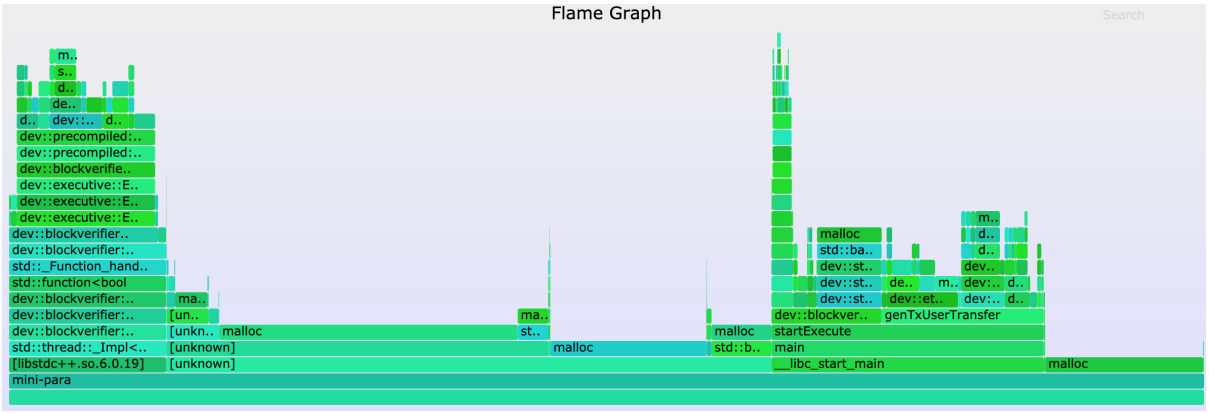
制作内存火焰图，首先需要向perf动态添加探针以监控标准库的malloc行为，并采样捕捉正在进行内存申请/释放的函数的调用堆栈：

```
perf record -e probe_libc:malloc -F 1000 -p `pidof fisco-bcos` -g -- sleep 60
```

然后绘制内存火焰图：

```
perf script > memory.perf
./stackcollapse-perf.pl memory.perf > memory.folded
./flamegraph.pl  --colors=mem memory.folded > memory.svg
```

得到的火焰图如下图所示：



我们起初猜想，这块未知的缓存可能位于LevelDB的数据库连接模块或JSON解码模块中，但通过比对第一次执行区块和后续执行区块的内存火焰图，我们发现各个模块中malloc采样数目的比例大致相同，因此很快便将这些猜想否定掉了。直到结合Off-CPU火焰图观察，我们才注意到第一次执行区块时调用sysmalloc的次数异常之高。联想到malloc会在首次被调用时进行内存预分配的特性，我们猜想第一次执行区块耗时较多可能就是由此造成的。

为验证猜想，我们将malloc的预分配空间上限调低：

```
export MALLOC_ARENA_MAX=1
```

然后再次进行测试并绘制Off-CPU火焰图，发现虽然性能有所降低，但是第一次执行区块的耗时和sysmalloc调用次数，基本无异于之后执行的区块。据此，我们基本可以断定这种有趣的现象是由于malloc的内存预分配行为导致。

当然，这种行为是操作系统为了提高程序整体性能而引入的，我们无需对其进行干涉，况且第一个区块的执行速度较慢，对用户体验几乎也不会造成负面影响，但是再小的性能问题也是问题，作为开发人员我们应当刨根问底，做到知其然且知其所以然。

虽然这次Memory火焰图并没有帮我们直接定位到问题的本质原因，但通过直观的数据比对，我们能够方便地排除错误的原因猜想，减少了大量的试错成本。面对复杂的内存问题，不仅需要有敏锐的嗅觉，更需要Memory火焰图这类好帮手。

4.DIY工具

尽管已经有如此多优秀的分析工具，帮助我们在性能优化前进的道路上披荆斩棘，但强大的功能有时也会赶不上性能问题的多变性，这种时候就需要我们结合自身的需求，自给自足地开发分析工具。

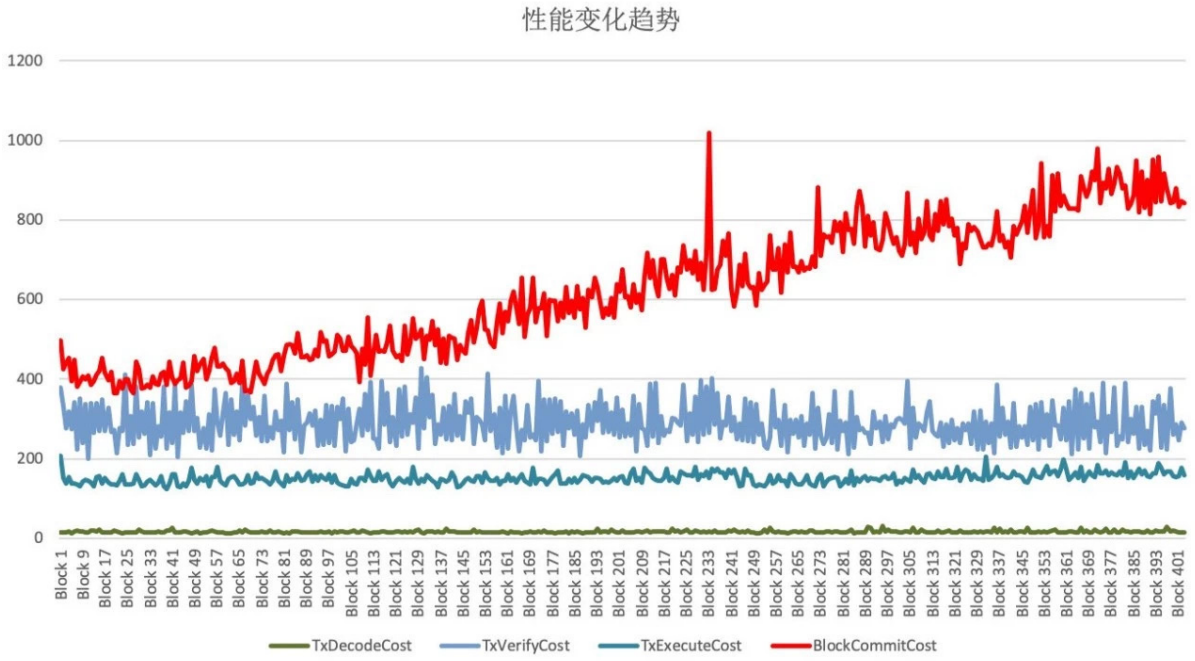
在进行FISCO BCOS的稳定性测试时，我们发现随着测试时间的增长，FISCO BCOS节点的性能呈现衰减趋势，我们需要得到所有模块的性能趋势变化图，以排查出导致性能衰减的元凶，但现有的性能分析

工具基本无法快速、便捷地实现这一需求，因此我们选择另寻他路。

首先，我们在代码中插入大量的桩点，这些桩点用于测量我们感兴趣的代码段的执行耗时，并将其附加上特殊的标识符记录于日志中：

```
auto startTime = utcTime();
/*
...code to be measured...
*/
auto endTime = utcTime();
auto elapsedTime = endTime - startTime;
LOG(DEBUG) << MESSAGE("<identifier>timeCost: ") \
  << MESSAGE(to_string(elspasedTime));
```

当节点性能已经开始明显下降后，我们将其日志导出，使用自己编写的Python脚本将日志以区块为单位进行分割，随后读取每个区块在执行时产生的桩点日志，并解析出各个阶段的耗时，然后由脚本汇总到一张大的Excel表格中，最后再直接利用Excel自带的图表功能，绘制出所有模块的性能趋势变化图，如下图所示：



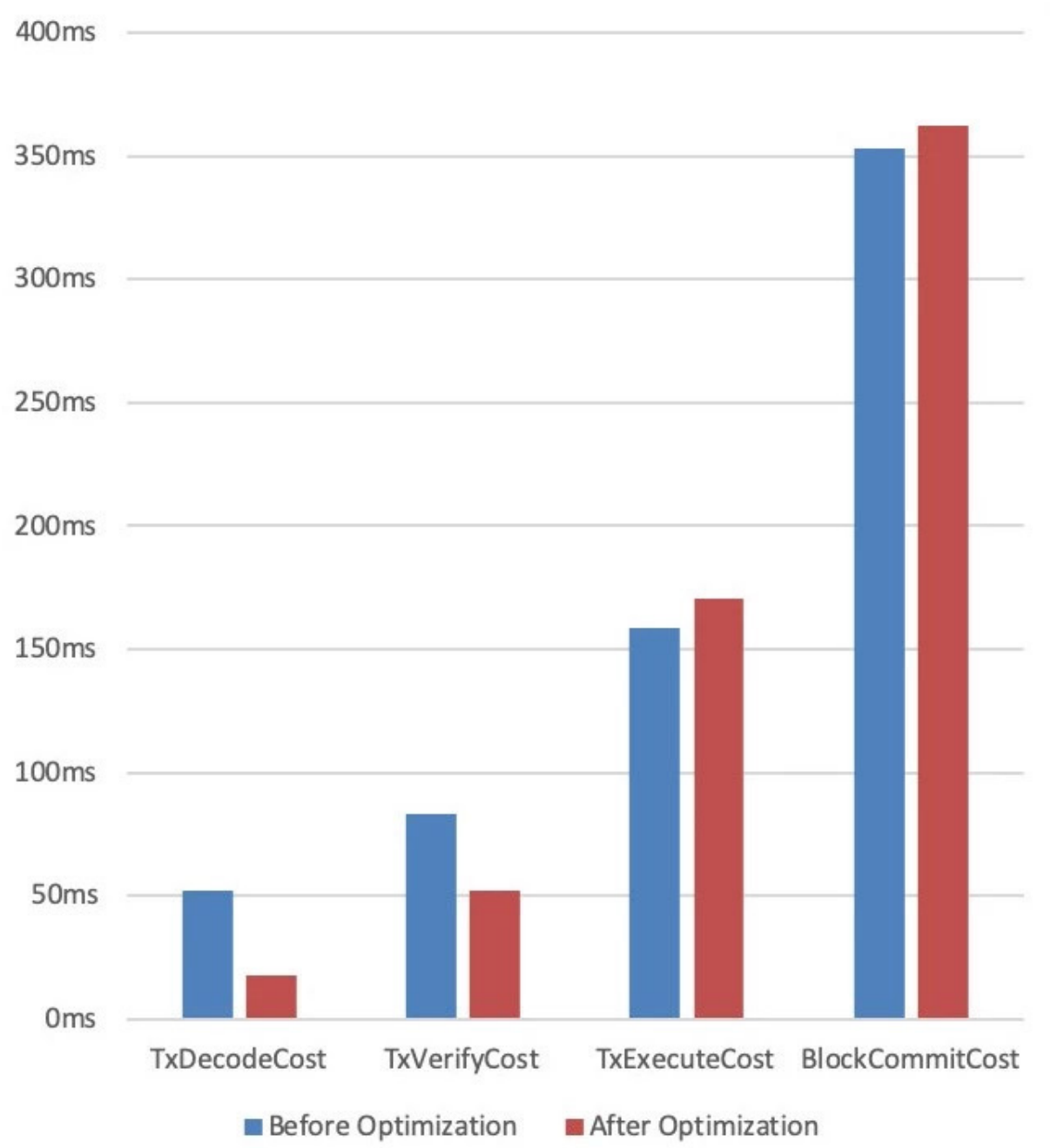
其中，横坐标为区块高度，纵坐标为执行耗时（ms），不同颜色曲线代表了不同模块的性能变化。

从图中可以看出，只有由红色曲线代表的区块落盘模块的执行耗时明显地随着数据库中数据量的增大而迅速增加，由此可以判断节点性能衰减问题的根源就出在区块落盘模块中。使用同样的方式，对区块落盘模块的各个函数进一步剖析，我们发现节点在向数据库提交新的区块数据时，调用的是LevelDB的update方法，并非insert方法。

两者的区别是，由于LevelDB以K-V的形式存储数据，update方法在写入数据前会进行select操作，因为待update的数据可能在数据库中已存在，需要先按Key查询出Value的数据结构才能进行修改，而查询的耗时与数据量成正比，insert方法则完全不需要这一步。由于我们写入的是全新的数据，因此查询这一步是不必要的，只需改变数据写入的方式，节点性能衰减的问题便迎刃而解。

相同的工具稍微变换一下用法，就能衍生出其他的用途，比如：将两批桩点性能数据放入同一张Excel表格中，便能够通过柱状图工具清晰地展现两次测试结果的性能变化。

下图展示的是我们在优化交易解码及验签流程时，优化前后性能柱状对比图：



从图中可以看出，交易解码和验签流程优化后的耗时的确比优化前有所降低。借由柱状对比图，我们能够轻松地检查优化手段是否行之有效，这一点在性能优化的过程中起到了重要的指导作用。

综上可见，DIY工具并不一定需要有多复杂，但一定可以最快地满足我们的定制化需求。

FISCO BCOS可并行合约开发框架（附实操教程）

作者：石翔 | FISCO BCOS 核心开发者

本专题系列文章追到现在，也许你会想问，FISCO BCOS的并行到底怎么用？作为专题的完结篇，本文就来揭晓"庐山真面目"，并教你上手使用FISCO BCOS的并行特性！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。

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。互斥情况如下：

交易	互斥对象	交集	是否互斥
transfer(A, B) 和 transfer(A, C)	[A, B] 和 [A, C]	[A]	互斥，不可并行执行
transfer(A, B) 和 transfer(B, C)	[A, B] 和 [B, C]	[B]	互斥，不可并行执行
transfer(A, C) 和 transfer(B, C)	[A, C] 和 [B, C]	[C]	互斥，不可并行执行
transfer(A, B) 和 transfer(A, B)	[A, B] 和 [A, B]	[A, B]	互斥，不可并行执行
transfer(A, B) 和 transfer(C, D)	[A, B] 和 [C, D]	无	无互斥，可并行执行

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。

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合约的并行框架

编写并行的solidity合约，开发流程与开发普通solidity合约流程相同。在此基础上，只需将ParallelContract 作为需要并行的合约基类，并调用registerParallelFunction()，注册可以并行的接口即可。

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;
```

(continues on next page)

(续上页)

```
    function transfer(string from, string to, uint256 num) public
{
        // 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); // 冲突参数:␣
↪string string
        registerParallelFunction("set(string,uint256)", 1); // 冲突参数: 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：

step1 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;
}
```

step2 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

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;
}
```

step3 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); // transfer接口，前1个四互斥参
数
```

(continues on next page)

(续上页)

```
}

// 注销并行合约接口
function disableParallel() public
{
    unregisterParallelFunction("transfer(string,string,uint256)");
    unregisterParallelFunction("set(string,uint256)");
}
```

step4 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，将在下文的例子中给出。

预编译合约的并行框架

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。

step1 将合约定义成支持并行

```
bool isParallelPrecompiled() override { return true; }
```

step2 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))
        {
            // 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
}
```

step3 编译，重启节点

手动编译节点的方法，参考FISCO BCOS技术文档。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。

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

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，让交易均匀发送到每个节点上，才能让链接收足够多的交易

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。

step1 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。

step2 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。
```

step3 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)。

step4 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（3 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仅做示例使用，请勿直接运用于生产环境。

step1 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。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 add 10000 2500 user
```

step2 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。

```
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 transfer 100000 4000 user 2
```

step3 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%
```

(continues on next page)

(续上页)

```
2000 < time           : 74323  : 92.90375%
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3143。执行结果校验后，无任何错误(verify_failed count is 0)。

step4 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。

如您在部署过程中遇到阻碍或有问题需要咨询，可以进入FISCO BCOS官方技术交流群寻求解答。（进群请长按下方二维码识别添加小助手）

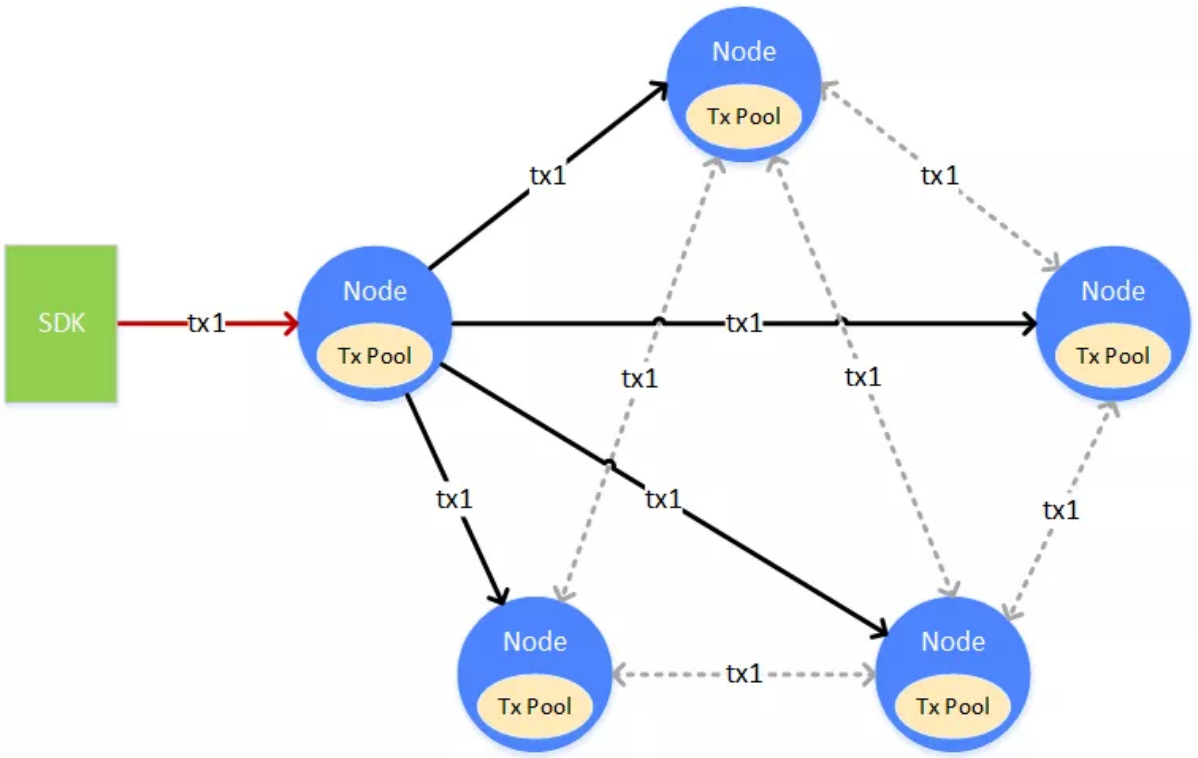
区块链的同步及其性能优化方法

作者：石翔 | FISCO BCOS 核心开发者

同步，是区块链中非常重要的流程，从功能上分为“交易同步”和“状态同步”。交易同步在交易提交时执行，优先保证交易能发往所有的节点，被打包处理。状态同步发生在某个节点发现自己区块高度落后于全网时，通过状态同步快速追回到全网最高的高度，这样作为共识节点可以参与到最新的共识过程中，而非共识节点能获取到最新的区块数据，以进行存储和验证。

交易同步

交易同步，是让区块链上的交易尽可能到达所有节点，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到达所有节点，收到广播交易的节点，会根据自身网络拓扑和网络流量策略，选择一到多个相邻节点，进行接力广播。

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设计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

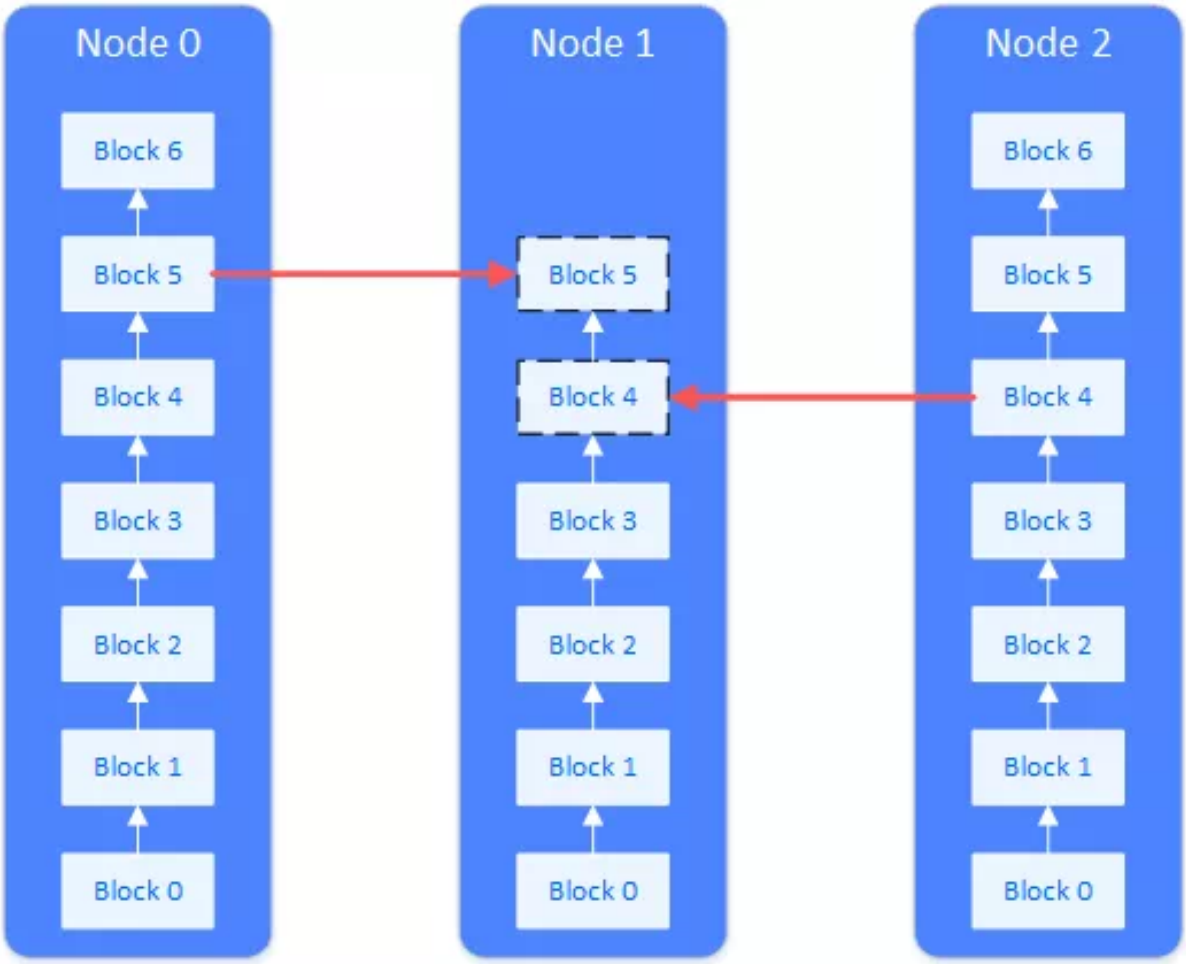
通过上述策略，能够尽量让交易到达所有的节点，交易会尽快被打包、共识、确认，让交易能够更快得到执行的结果。

广播策略已经在复杂网络里尽量追求了网络最终到达率，但也会有极小的概率，出现某交易在一定时间窗内无法到达某节点的情况。当交易未到达某个节点，只会使得交易被确认的时间变长，不会影响交易的正确性，也不会漏处理交易，因为有广播机制，网络里还有更多的节点有机会继续处理这个交易。

区块同步

区块同步，能让区块链节点的数据状态保持在最新。

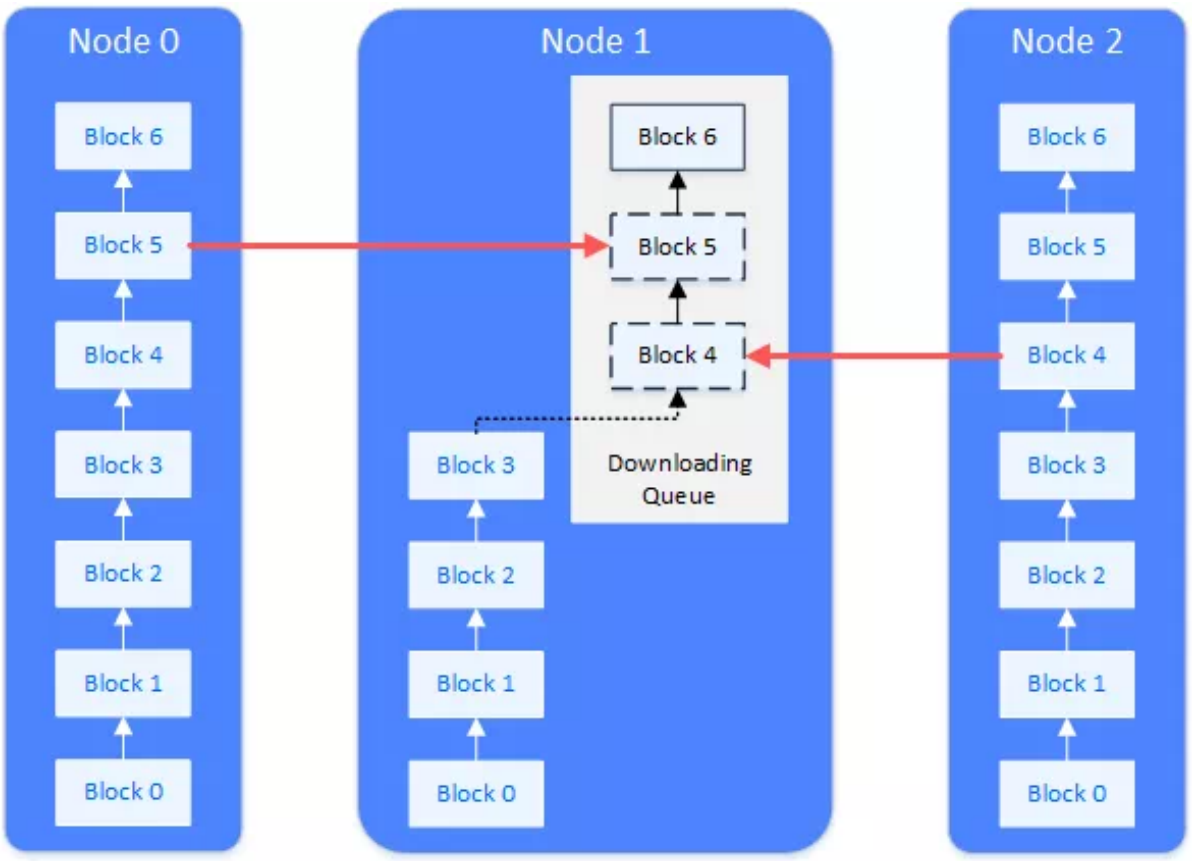
区块链状态的新旧，最重要的标识之一就是区块高度，而区块里包含了链上的历史交易，若一个节点的块高和全网最高块高对齐，则此节点有机会回溯历史交易，以获得区块链的最新状态，只有拥有最新状态的节点，才能参与到共识中去，进行下一个新区块的共识。



当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，此节点的区块高度落后于其它节点，其状态不是最新的。此时就需要进行区块同步。如上图，需要区块同步的节点（Node 1），会主动向其它节点请求下载区块。整个下载过程会将网络请求负载分散到多个节点上。

区块同步与下载队列

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/响应”的方式完成，进入下载流程的节点，会随机挑选满足要求的节点，发送需要下载的区块高度区间。收到下载请求的节点，会根据请求的内容，回应相应的区块。



收到回应区块的节点，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。下载下来的新区块，会不断插入到下载队列中，按高度排序。排序好的区块，会依次被节点执行和验证。验证通过后，更新本地的数据状态，让块高增加，直至更新到最新块，块高达到最高。

性能优化

对同步的性能优化，能有效提升系统效率。FISCO BCOS在这方面做了很多，下面选取了一些关键的优化点。

编码缓存

在交易广播中，需要将交易编码成二进制数据发送给其它节点，其它节点在收到交易二进制数据后，需要解码成程序可识别的数据结构。当交易量很大时，编解码成为交易广播的性能瓶颈。FISCO BCOS将交易的二进制编码进行缓存，当要发送交易时，直接从缓存中取出二进制发送，减少编解码的频率，增加交易广播的速率。

负载均衡

区块落后的节点会通过请求的方式从其它节点下载区块。其它节点在收到请求后，会将相应区间的区块发送给落后的节点。在区块落后很多的情况下，FISCO BCOS节点会将下载区间均匀切分，向不同的节点发起请求，把下载负载分散到不同的节点上，避免单一被请求节点因承载大量的数据访问请求而影响其运行性能。

回调剥离

在FISCO BCOS节点中，有多个回调线程处理网络上收到的包。当网络流量很大时，处理网络包的线程处理不过来，会将网络包放到缓冲队列中。网络上的包主要为同步包和共识包，共识包优先级更高，直接影响出块速度。为了不影响共识包的处理，FISCO BCOS将同步包的处理逻辑从网络回调线程中剥离出来，交给另外的独立线程，和共识包的处理解耦和并行了。

验签去重

同步模块在收到交易时，需要对交易进行验签。共识模块收到区块后，从区块中取出交易，也需要对交易进行验签。虽然是同一笔交易，却在同步和共识中都进行了验签。然而验签非常耗时，大大影响了交易执行的TPS。FISCO BCOS在交易执行时做了去重逻辑，无论是同步还是共识，在验签前先查询验签记录，若此交易已验签，则直接从记录中获取验签结果，保证同一笔交易只验签一次。同时，FISCO BCOS让同步尽可能地在共识前验签，让共识尽可能直接获取验签结果，减少验签在共识中的耗时。共识得到了加速，链的TPS性能表现相应得到提升。

总结

共识与同步是区块链中必不可少的环节。共识打头阵，同步打辅助。同步流程使得整个区块链网络所有节点都达到数据一致性，保证数据全网可验证。同时，在不影响共识的前提下，为共识提前准备好所需的数据，让共识跑得更快更稳定。

相关阅读

- 卓别林演绎共识与同步流程优化
- 同步模块文档

「群聊互动」

Q 大雪无痕：负载均衡同步时如果同步到一个支链上的块，怎么办？另外，出现分叉时，支链保留多久后被丢弃?

A 石翔：FISCO BCOS 用的共识算法是pbft，不会分叉哈。不会分叉的共识算法带来很多好处，没有了回滚，同步可以负载均衡了，mpt树可以去了，mpt改成了表结构的存储，表结构间没有数据冲突，就可以并行执行交易了，所以联盟链可以很快。

FISCO BCOS同步模块的优化策略

作者：陈宇杰 | FISCO BCOS 核心开发者

作者语

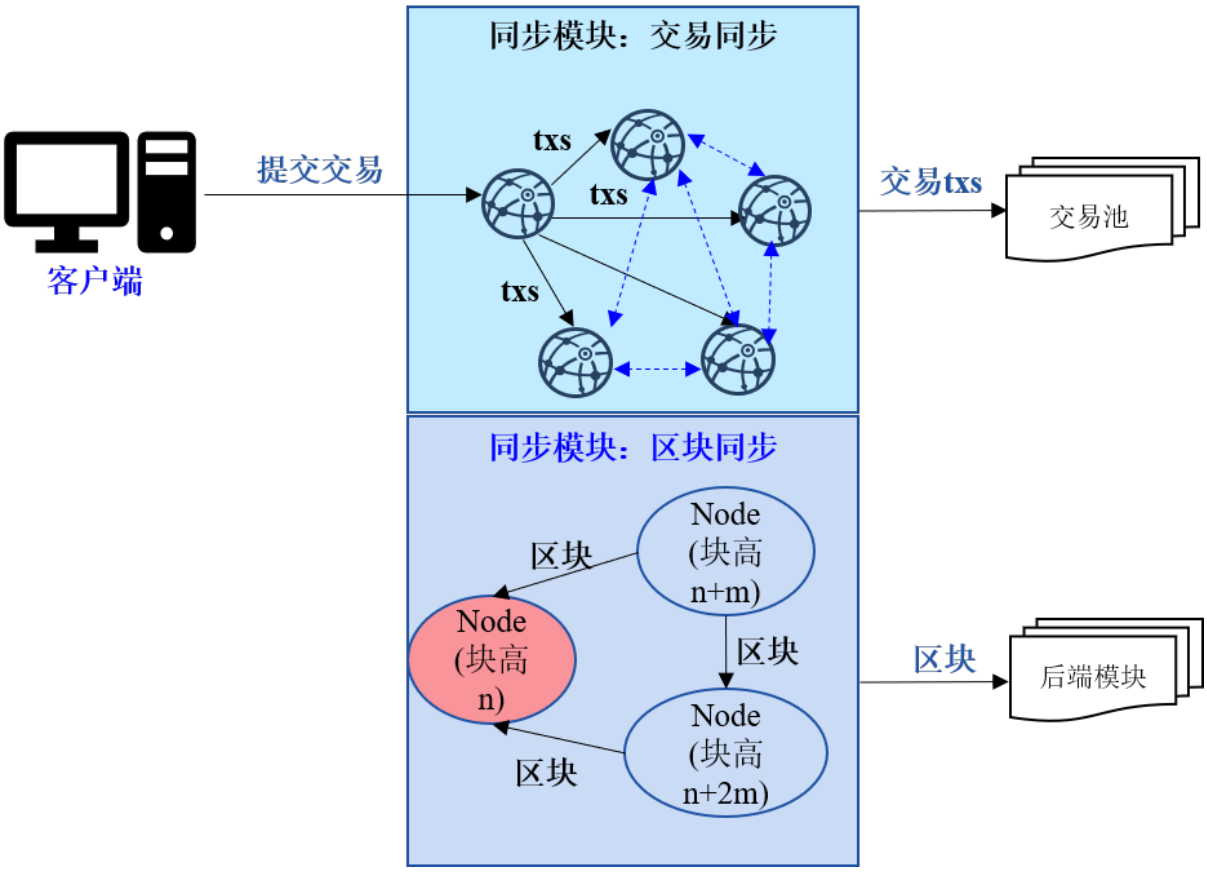
在FISCO BCOS区块链系统中，同步模块身担重任。既是交易传输小能手，将客户端发送的交易输送给所有节点交易池，源源不断地为共识模块提供打包区块的“原材料”；又是“困难户拯救者”，将最新区块同步到块高落后的“困难户”节点，使其能正常参与共识。

当然，由于同步模块的职责大多与网络有关，因此它也是系统“带宽消耗大户”，会导致区块链部分节点带宽负载高。为此，FISCO BCOS开发者们也设计了一系列策略来优化这个“带宽消耗大户”，使其能更优雅地服务于系统。

本文将详细介绍FISCO BCOS同步模块的优化之路。

初识同步模块

交易同步和区块同步是FISCO BCOS区块链系统同步模块的主要职责，这两个职责都和网络相关。



如上图所示，交易同步负责将客户端发出的交易发送给所有其他节点，为共识模块提供用于打包区块的交易。为保证交易能到达所有节点，交易同步主要包括交易广播和交易转发两个部分：

- **交易广播**：客户端首先将交易发送给客户端直连节点，该节点将收到的交易广播给所有其他节点；
- **交易转发**：为保证网络断连情况下交易能到达所有节点，收到广播交易的节点，随机选取25%的节点转发收到的交易。

区块同步则负责拯救块高落后的"困难户"，将最新块高同步到块高落后的节点。节点区块高度低于其他节点时，会主动向区块高度更高的节点拉取新区块。

同步模块的不合理网络使用姿势

前面章节提到同步模块是"带宽消耗大户"，下面就来细说这个"带宽消耗大户"的不合理网络使用姿势。

交易同步时，客户端直连节点网络负载高

考虑到gossip协议消息传递速度慢，联盟链场景一般采用节点网络全互联的方式来提升网络效率。为保障客户端发出的交易能快速到达所有节点，客户端直连节点收到交易后，会将交易广播给所有节点。由于区块链节点外网带宽有限，随着节点规模增加，客户端直连节点必然因网络负载高而成为系统瓶颈。

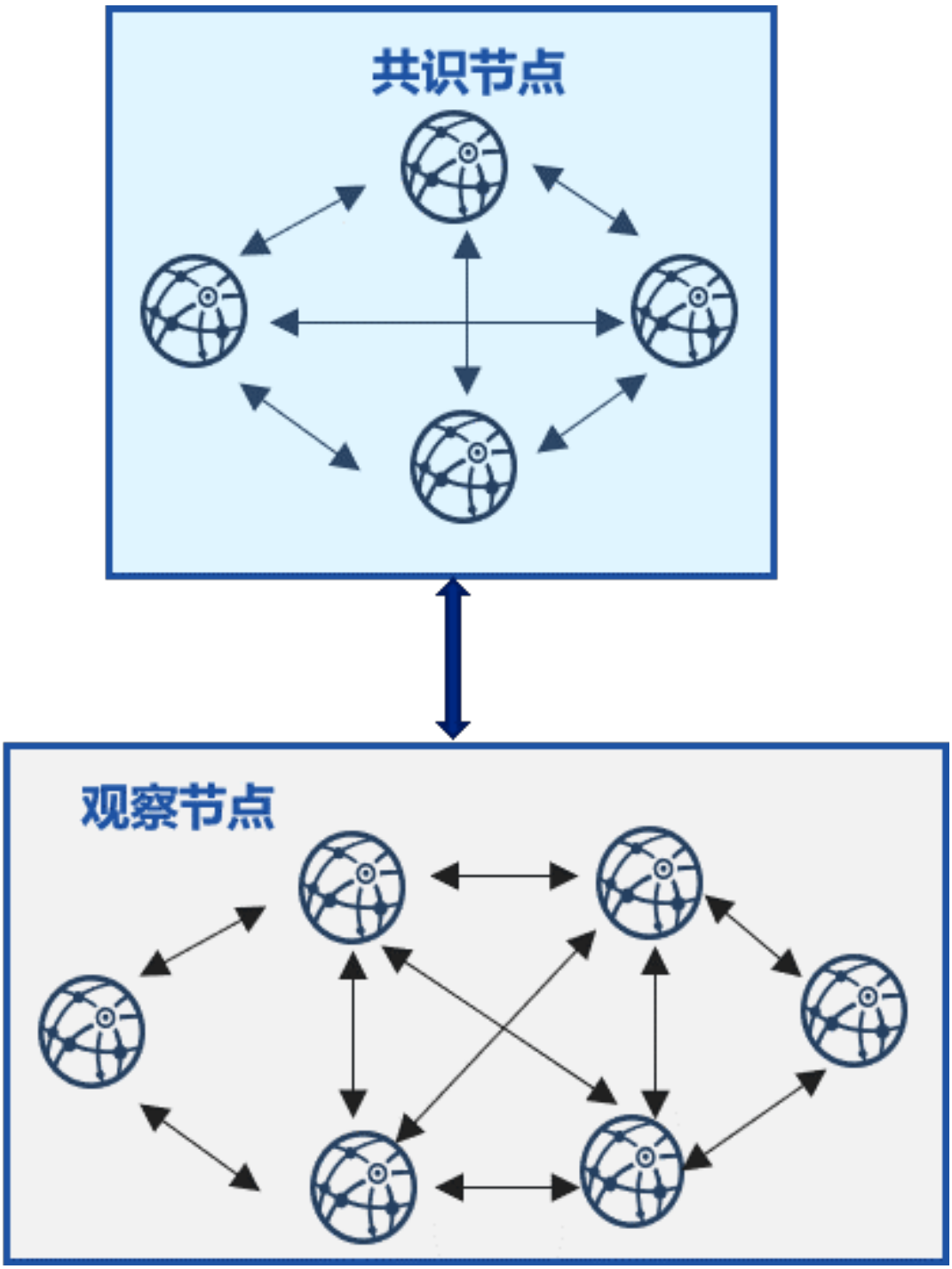
交易转发时，网络利用效率低

为保障在部分节点网络断连的情况下，交易仍然能到达所有节点，同步模块引入了交易转发机制。节点收到交易后，会随机选取若干节点广播收到的交易。

网络全连的情况下，这样会导致部分节点频繁收到重复的数据包，且节点数目越多，因交易转发带来的冗余消息包就越多，无疑造成巨大的网络带宽浪费。

区块同步时，部分节点网络负载高，导致节点规模不可扩展

考虑到目前使用的BFT或CFT类共识算法的区块链复杂度较高、不具有无限可扩展性，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。



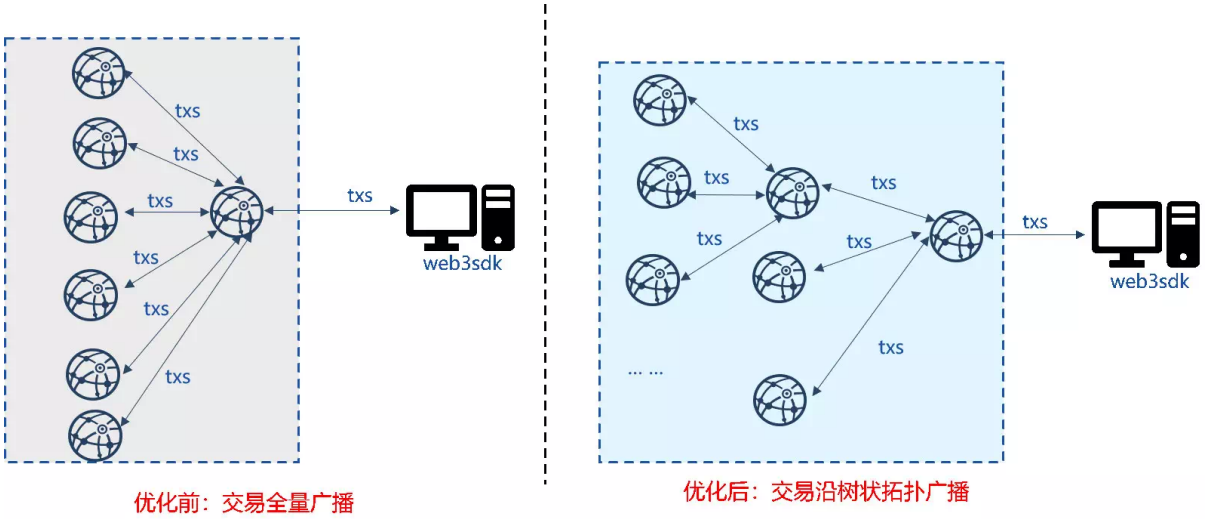
在这种架构中，大部分观察节点随机从拥有最新区块的共识节点下载区块。在包含n个共识节点、m个观察节点的区块链系统中，设每个区块大小为block_size，理想情况下(即负载均衡)，每共识一个区块，每个共识节点需要向m/n个观察节点发送区块，共识节点出带宽大约是(m/n)*block_size*；设网络带宽是*bandwidth*，则每个共识节点最多可向*(bandwidth/block_size)*个节点同步区块，即区块链最大节点规模是*(n*bandwidth/block_size)。在公网带宽bandwidth较小、区块较大的情况下，能容纳的节点数有限，因此随机的区块同步策略不具有可扩展性。

同步模块的优化策略

为提升系统带宽使用效率和系统的可扩展性，FISCO BCOS开发者们提出了一系列优化策略来“纠正”同步模块不合理的网络使用姿势，使其能更优雅、更高效地服务于FISCO BCOS区块链系统。

策略一：交易树状广播

为降低客户端直连节点交易广播带来的网络压力，FISCO BCOS秉着负载均衡的原则，设计了交易树状广播策略，将客户端直连节点的压力分摊给其下属子节点，下图展示了优化前后七节点区块链系统的交易广播拓扑：

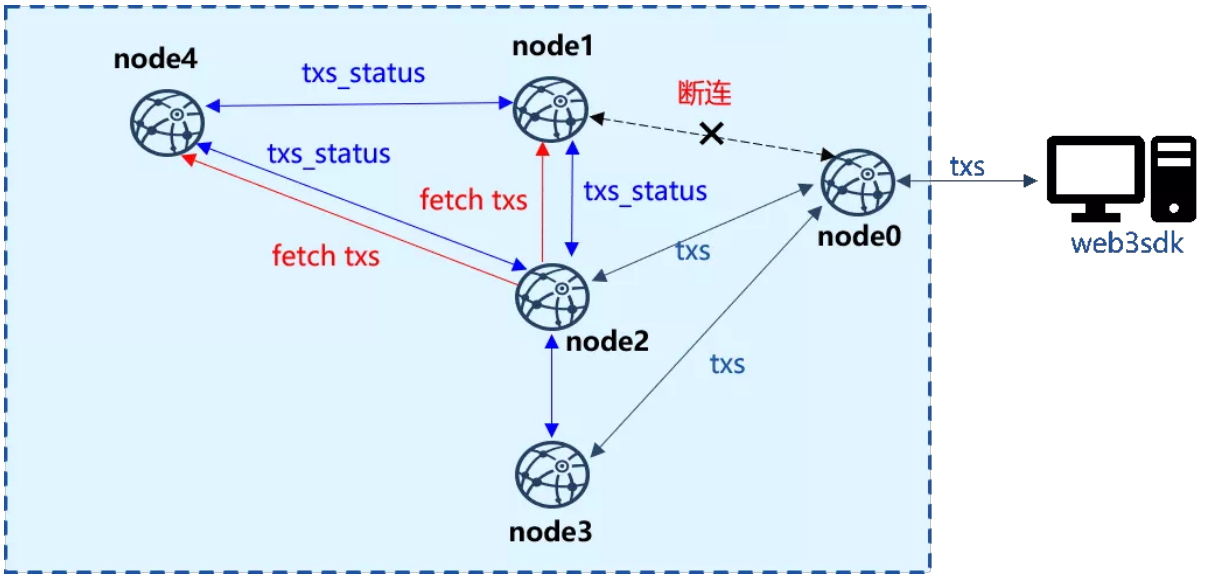


- **优化前：**节点收到客户端交易后，全量广播给其他节点；
- **优化后：**节点收到客户端交易后，将其发送给子节点，子节点收到交易后，继续将交易转发给自身的子节点。

采用交易树状广播后，上图所示的客户端直连节点将部分网络负载分摊给子节点，带宽负载降低为原先的一半，达到了负载均衡的目标。并且，由于所有节点广播交易的出带宽仅与树状拓扑的宽度有关，交易树状广播策略具有可扩展性。此外，相较于基于gossip的交易广播机制，树状广播策略提升了交易广播速率的同时，降低了网络中冗余的消息包数量。

策略二：基于状态包的交易转发优化

为消除交易转发带来的带宽消耗，提升网络效率，FISCO BCOS提出了基于状态包的交易转发策略，优化后的交易转发模块不直接转发交易，仅转发交易状态。节点可根据收到的交易状态以及交易池内已有的交易获取缺失的交易，并直接向相应节点拉取缺失交易。



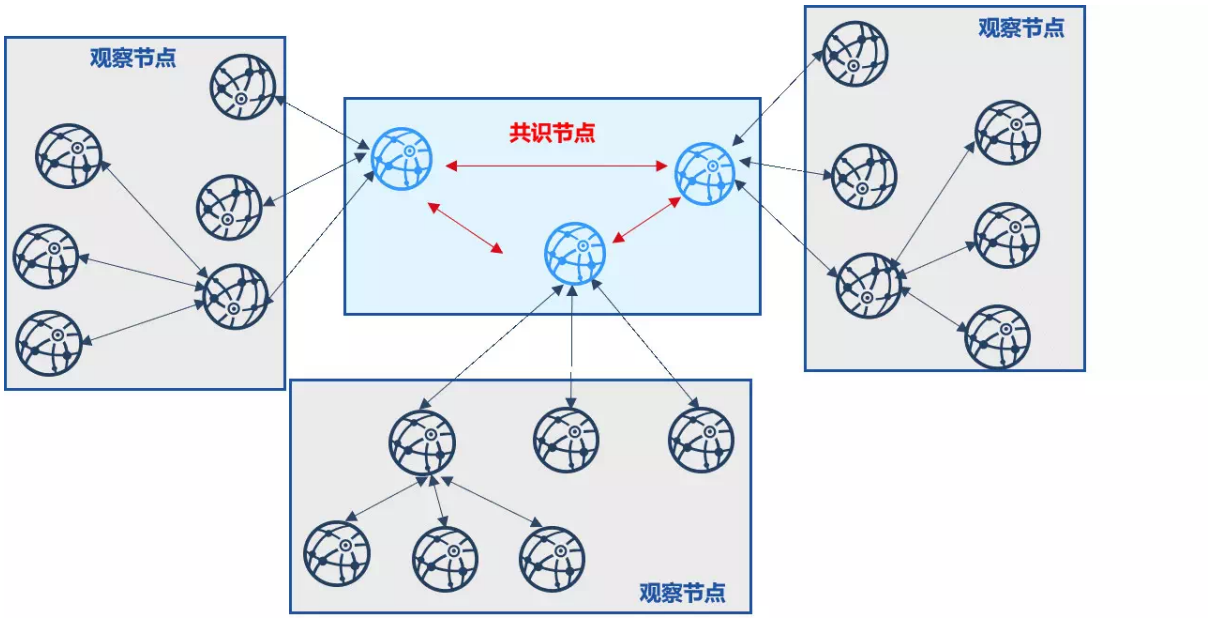
上图中，客户端直连node0，但node0与node1、node4断连，此时node0仅能将交易广播

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请求交易。在全连的网络环境中，所有节点交易状态基本一致，节点间交易请求较少，相较于直接转发交易的策略，大大降低了转发冗余交易引起的带宽浪费。

策略三：区块同步可扩展性优化

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，提升区块链系统区块同步的可扩展性，FISCO BCOS设计并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，每个节点优先将最新区块状态发送给子节点，子节点优先向父节点同步最新块，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。

此外，考虑到树状拓扑中，节点断连可能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，使得树状拓扑中的断连节点也能从其邻居节点处同步最新区块，保障树状区块状态广播的健壮性。

小结

同步模块是交易传输小能手，也是“困难户拯救者”，更是”带宽消耗大户”，这个”带宽消耗大户”无论是在执行同步交易的任务，还是在履行同步区块的职责，都存在浪费带宽、过度使用部分节点带宽的嫌疑。

FISCO BCOS开发者采用一系列优化策略规范了同步模块的带宽使用姿势，尽量减少同步模块冗余消息包的同时，将高负载节点的带宽压力分摊给下属子节点，提升了区块链系统的可扩展性，使得优化后的同步模块能更优雅、更高效、更健壮地服务于区块链系统。

记一次CachedStorage中死锁的调试经历

作者：李陈希 | FISCO BCOS 核心开发者

在整合FISCO BCOS非国密单测与国密单测的工作中，我们发现CachedStorage的单测偶然会陷入卡死的状态，且可在本地持续复现。复现方式为循环执行CachedStorage单测200次左右，便会发生一次所有线

程均陷入等待状态、单测无法继续执行的情况，我们怀疑在CachedStroage中发生了死锁，故对此进行调试。

Debug思路

中医治病讲究望闻问切，调试bug同样需要遵循寻找线索、合理推断、验证解决的思路。

观察线程栈

在死锁发生时，使用/usr/bin/sample工具（mac平台环境下）将所有的线程的栈打印出来，观察各线程的工作状态。从所有线程的线程栈中观察到有一个线程（此处称为T1）卡在CachedStorage.cpp的第698行的touchCache函数中，点击参考具体的代码实现。

```
691         RWMutexScoped lockCache(m_cachesMutex, false);
692
693         auto result = m_caches.insert(std::make_pair(cacheKey, cache));
694         if (!result.second && cache != result.first->second)
695         {
696             cache = result.first->second;
697             cacheLock.reset();
698             cacheLock = std::make_shared<Cache::RWScoped>(*(cache->mutex()), write);
699         }
```

从代码片段中可以看到，T1在第691行已经获得了m_cachesMutex的读锁：代码RWMutexScoped(some_rw_mutex, false)的意思是获取某个读写锁的读锁；相应地，代码RWMutexScoped(some_rw_mutex, true)的意思是获取某个读写锁的写锁，这里的RWMutex是一个Spin Lock。

随后在第698行处尝试获取某个cache的写锁。除了T1，还有另外一个线程（此处称为T2）卡在CachedStorage.cpp的第691行的touchCache函数中：

```
681     auto cacheLock = std::make_shared<Cache::RWScoped>(*(cache->mutex()), write);
682     if (inserted)
683     {
684         hit = false;
685
686         cache->setKey(key);
687         cache->setTableInfo(tableInfo);
688     }
689     else
690     {
691         RWMutexScoped lockCache(m_cachesMutex, false);
```

从代码片段中可以看到，T2在第681行已经获得了某个cache的写锁，随后在第691行处尝试获取m_cachesMutex的读锁。继续观察后还发现若干线程卡在CachedStorage.cpp第673行的touchCache函数中：

```
671     bool inserted = false;
672     {
673         RWMutexScoped lockCache(m_cachesMutex, false);
674 
675         auto result = m_caches.insert(std::make_pair(cacheKey, cache));
676 
677         cache = result.first->second;
678         inserted = result.second;
679     }
```

最后还有一个Cache清理线程（此处称为T3）卡在CachedStorage.cpp的第734行的removeCache函数中：

```
731 void CachedStorage::removeCache(const std::string& table, const std::string& key)
732 {
733     auto cacheKey = table + "_" + key;
734     RWMutexScoped lockCache(m_cachesMutex, true);
```

从代码片段中可以看到，这些线程均没有持任何锁资源，只是在单纯地尝试获取m_cachesMutex的写锁。

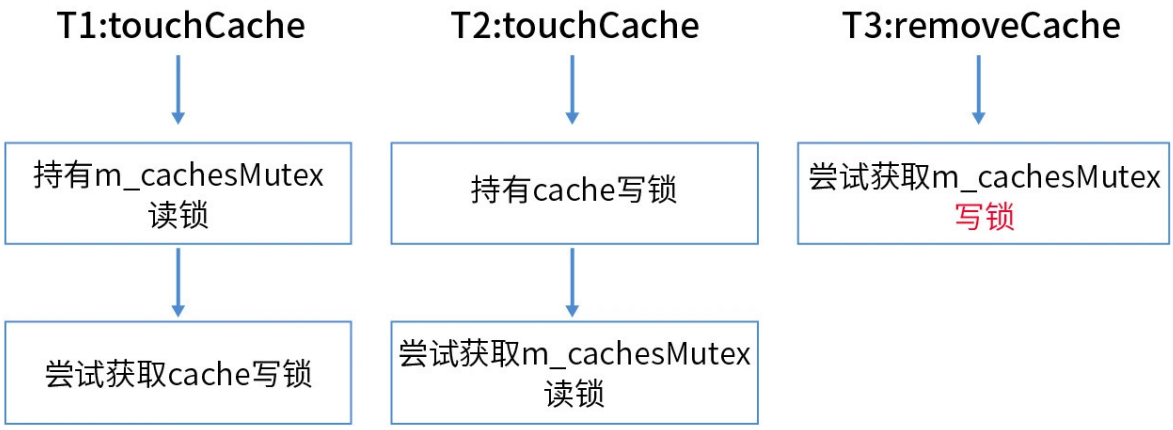
读写饥饿问题

初期分析问题时，最诡谲的莫过于：在T1已经获取到m_cachesMutex读锁的情况下，其他同样试图获取m_cachesMutex读锁的线程竟然无法获取到。但是看到T3线程此时正努力尝试获取m_cachesMutex写锁，联想到读写锁饥饿问题，我们认为其他线程获取不到读锁的问题根源很可能就在T3。

所谓读写锁饥饿问题是指，在多线程共用一个读写锁的环境中，如果设定只要有读线程获取读锁，后续想获取读锁的读线程都能共享此读锁，则可能导致想获取写锁的写线程永远无法获得执行机会（因为读写锁一直被其他读线程抢占）。为了解决饥饿问题，部分读写锁会在某些情况下提高写线程的优先级，即由写线程先占用写锁，而其他读线程只能在写线程后乖乖排队直到写线程将读写锁释放出来。

在上述问题中，T1已经获取了m_cachesMutex的读锁，若此时T3恰好获得时间片并执行到CachedStorage.cpp的第734行，会因获取不到m_cachesMutex的写锁而卡住，随后其他线程也开始执行并到了获取m_cachesMutex读锁的代码行。若读写防饥饿策略真的存在，那这些线程（包括T2）的确会在获取读锁阶段卡住，进而导致T2无法释放cache锁，从而T1无法获取到cache锁，此时所有线程均会陷入等待中。

在这个前提下，似乎一切都能解释得通。上述流程的时序图如下所示：



(1) T1获取到m_cachesMutex读锁

(2) T2获取到cache写锁

(3) T3此时强势插入，试图获取m_cachesMutex写锁，由于T1已经持有m_cachesMutex读锁，T3陷入等待

(4) T1尝试获取cache写锁，由于cache写锁被被T2持有，T1陷入等待

(5) T2试图获取m_cachesMutex读锁，推测为了避免写线程陷入饥饿，tbb读写锁的读写优先级调度机制此时会让写线程（T3）之后的读线程（T2）无法获取到锁，直到写线程释放锁，因此T2陷入等待

(6) dead lock

我们找到了TBB中Spin RW Lock的实现代码，如下图所示：

获取写锁：

```
//! Acquire write lock on the given mutex.
bool spin_rw_mutex_v3::internal_acquire_writer()
{
    ITT_NOTIFY(sync_prepare, this);
    for( internal::atomic_backoff backoff;;backoff.pause() ){
        state_t s = const_cast<volatile state_t&>(state); // ensure reloading
        if( !(s & BUSY) ) { // no readers, no writers
            if( CAS(state, WRITER, s)==s )
                break; // successfully stored writer flag
            backoff.reset(); // we could be very close to complete op.
        } else if( !(s & WRITER_PENDING) ) { // no pending writers
            __TBB_AtomicOR(&state, WRITER_PENDING);
        }
    }
    ITT_NOTIFY(sync_acquired, this);
    return false;
}
```

获取读锁：

```
//! Acquire read lock on given mutex.
void spin_rw_mutex_v3::internal_acquire_reader()
{
    ITT_NOTIFY(sync_prepare, this);
    for( internal::atomic_backoff b;;b.pause() ){
        state_t s = const_cast<volatile state_t&>(state); // ensure reloading
        if( !(s & (WRITER|WRITER_PENDING)) ) { // no writer or write requests
            state_t t = (state_t)__TBB_FetchAndAddW( &state, (intptr_t) ONE_READER );
            if( !( t&WRITER ))
                break; // successfully stored increased number of readers
            // writer got there first, undo the increment
            __TBB_FetchAndAddW( &state, -(intptr_t)ONE_READER );
        }
    }

    ITT_NOTIFY(sync_acquired, this);
    __TBB_ASSERT( state & READERS, "invalid state of a read lock: no readers" );
}

//! Upgrade reader to become a writer
```

在获取写锁的代码中，可以看到写线程如果没有获取到写锁，会置一个WRITER_PENDING标志位，表明此时正有写线程在等待读写锁的释放，其他线程请勿打扰。

而获取的读锁代码中，也可以看到，如果读线程发现锁上被置了WRITER_PENDING标志位，就会老实地循环等待，让写线程优先去获取读写锁。这里读写锁的行为完美符合之前对读写锁防饥饿策略的推测，至此真相大白。既然找到了问题起因，那解决起来就容易多了。在CachedStorage的设计中，Cache清理线程优先级很低，调用频率也不高（约1次/秒）,因此给予它高读写锁优先级是不合理的，故将removeCache函数获取m_cachesMutex写锁方式做如下修改：

```
//! Acquire read lock on given mutex.
void spin_rw_mutex_v3::internal_acquire_reader()
{
    ITT_NOTIFY(sync_prepare, this);
    for( internal::atomic_backoff b;;b.pause() ){
        state_t s = const_cast<volatile state_t&>(state); // ensure reloading
        if( !(s & (WRITER|WRITER_PENDING)) ) { // no writer or write requests
            state_t t = (state_t)__TBB_FetchAndAddW( &state, (intptr_t) ONE_READER );
            if( !( t&WRITER ))
                break; // successfully stored increased number of readers
            // writer got there first, undo the increment
            __TBB_FetchAndAddW( &state, -(intptr_t)ONE_READER );
        }
    }

    ITT_NOTIFY(sync_acquired, this);
    __TBB_ASSERT( state & READERS, "invalid state of a read lock: no readers" );
}

//! Upgrade reader to become a writer
```

修改后，获取写锁方式跟获取读锁类似：每次获取写锁时，先try_acquire，如果没获取到就放弃本轮时间片下次再尝试，直到获取到写锁为止，此时写线程不会再去置WRITER_PENDING标志位，从而能够不影响其他读线程的正常执行。

相关代码已提交至2.5版本中，该版本将很快与大家见面，敬请期待。

实现效果

修改前循环执行CachedStorage单测200次左右便会发生死锁；修改后循环执行2000+次仍未发生死锁，且各个线程均能有条不紊地工作。

经验总结

从这次调试过程中，总结了一些经验与大家分享。

首先，分析死锁问题最有效的仍然是“两步走”方法，即通过pstack、sample、gdb等工具看线程栈，推测导致发生死锁的线程执行时序。这里的第二步需要多发挥一点想象力。以往的死锁问题往往是两个线程间交互所导致，教科书上也多以两个线程来讲解死锁四要素，但在上述问题中，由于读写锁的特殊性质，需要三个线程按照特殊时序交互才可以引发死锁，算是较为少见的情况。

其次，『只要有线程获取到读锁，那其他想获取读锁的线程一定也能获取读锁』的思维定势是有问题的。至少在上面的问题中，防饥饿策略的存在导致排在写线程后的读线程无法获取读锁。但本文的结论并非放之四海而皆准，要不要防饥饿、怎么防饥饿在各个多线程库的实现中有着不同的取舍。有的文章提到过某些库的实现就是遵循『读线程绝对优先』规则，那这些库就不会遇到这类问题，所以仍然需要具体问题具体分析。

FISCO BCOS流量控制实现

作者：陈宇杰 | FISCO BCOS 核心开发者

引言

区块链系统作为分布式系统，面对大数据量突发请求场景，暴涨的请求容易引起区块链服务或接口不可用，严重时可能导致整个区块链系统陷入雪崩状态。

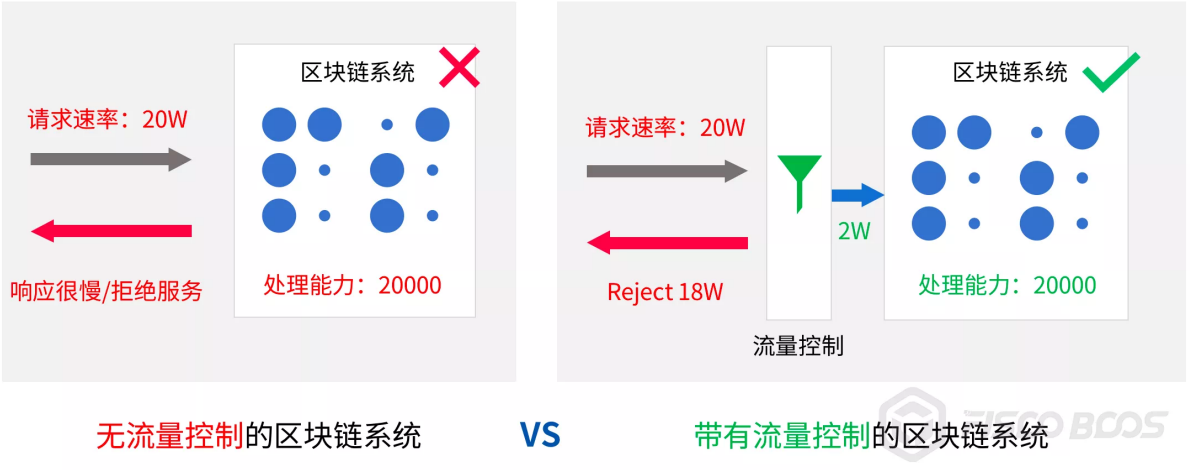
为了提供更稳定可靠、柔性可用的服务，FISCO BCOS v2.5版本引入了流量控制功能，从节点和群组两个维度进行限流，一方面，面对大数据量突发请求时对区块链系统进行保护，保证系统能正常运行，提升系统可用性；另一方面降低区块链节点间、群组间的资源干扰，提升区块链系统的服务质量。

为什么引入流量控制

FISCO BCOS引入流量控制，旨在：

- 应对大数据量突发请求
- 降低区块链节点间、群组间的资源干扰
- 降低模块间的相互影响

应对大数据量突发请求



上图对比了**无流量控制功能**和**带有流量控制功能**的区块链系统面对大数据量突发请求时的处理情况。

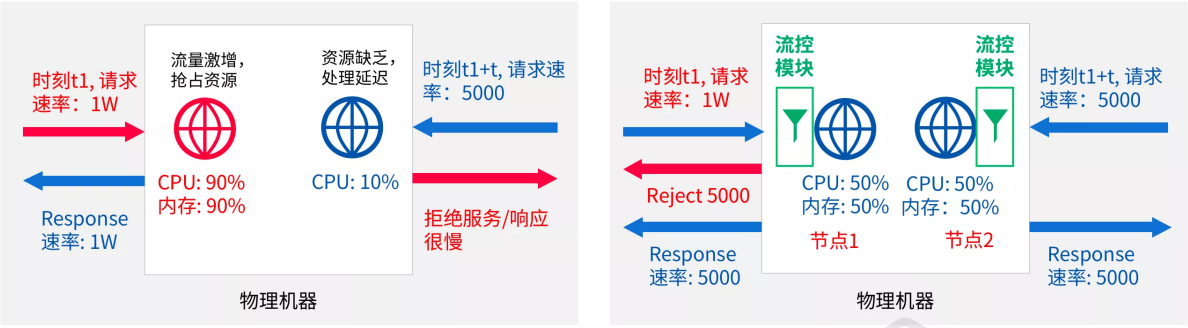
假设该区块链系统处理能力为2W，当业务以20W请求速率访问区块链节点时：

- **无流量控制**的场景下，系统对业务请求**照单全收**，导致内部积压的请求数目越来越多，区块链节点响应速度越来越慢，若业务持续以高于系统处理能力的速率发起请求，最终整个系统可能会陷入雪崩状态，无法响应任何业务请求。
- 加入了**流量控制功能**后，流量控制模块会**根据****系统处理能力过滤业务请求**。在业务请求速率超出系统处理能力时，流量控制模块会拒绝剩余的处理请求，使系统维持"收支平衡"的健康状

态；并将请求过载的信息返回给业务，业务可根据该信息自适应地调整请求速率，对区块链系统进行保护。

简而言之，引入流量控制模块就是给区块链系统加上一层安全保护罩，让系统在接收大数据量突发请求的场景下可以健壮工作，正常响应业务请求。

降低区块链节点间/群组间资源干扰



无流量控制的区块链系统 VS 带有流量控制的区块链系统

注：图中两个节点属于两条不同的链，接入了两个不同服务

如上图，当多个区块链节点部署于同一台机器时，会出现资源竞争的问题，某些节点占用过多系统资源会影响到其他节点的正常服务。

- t1时刻，业务1持续以1W的请求速率请求左边节点，该节点流量激增，系统接收并处理请求后，使用了90%的CPU
- 经过t时间间隔，业务2以5000的请求速率请求右边节点，该节点资源匮乏，只能抢占到10%的CPU，响应速度很慢

上述场景中，左边节点因占用过多系统资源影响了右边节点的服务质量。引入流量控制后，可限制每个节点接收请求的速率，控制每个区块链节点的资源占用，避免因区块链节点资源竞争导致的服务质量下降或服务不可用问题。

仍以上图为例：

- t1时刻，业务1持续以1W的请求速率请求节点1，节点1流量控制模块根据配置的请求阈值拒绝多余的请求(这里设阈值为5000)，机器CPU占用率维持在50%
- 业务1收到"流量过载"的响应后，可将其请求速率调整到5000
- 经过t时间间隔，业务2以5000的请求速率请求节点2，此时机器还剩余50%的CPU，足以处理5000个请求，业务2的请求得到正常响应

类似于一台机器上运行多个区块链节点时会发生资源竞争，多群组架构下，群组间也存在资源竞争，某个群组占用过多资源同样会影响到其他群组的服务质量，采用群组级别的流量控制是解决群组间资源竞争的良方。

降低模块间相互影响

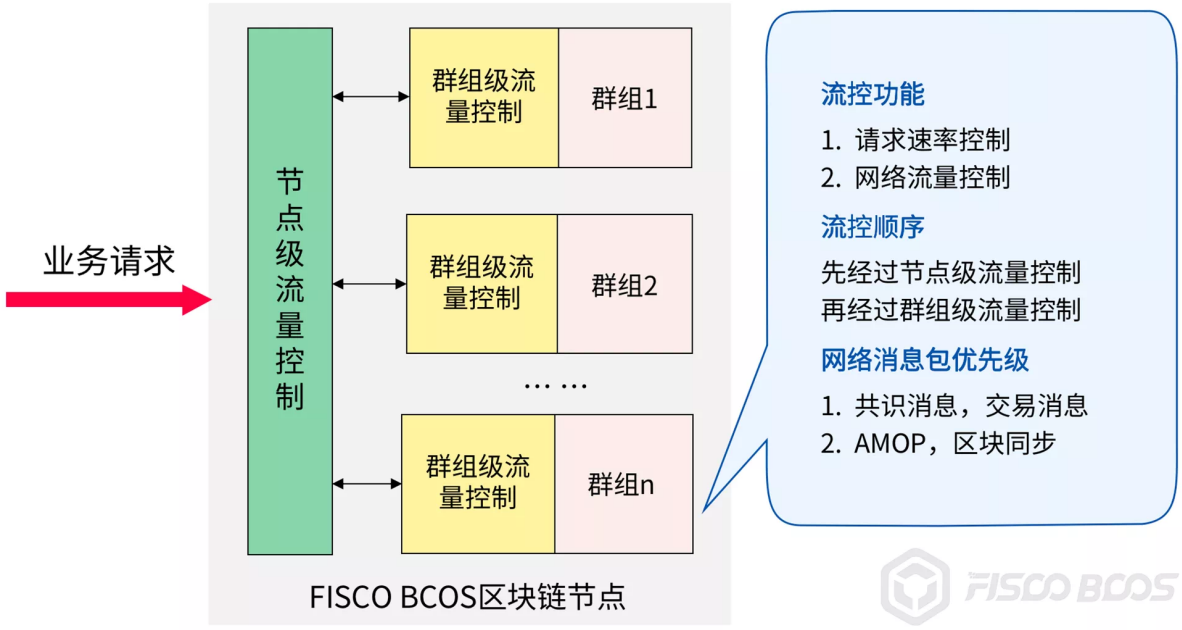
同一个节点或群组内的不同模块，也存在资源竞争问题，主要是网络资源竞争，存在网络资源竞争的模块包括：

- 共识模块
- 交易同步模块
- 区块同步模块
- AMOP模块

其中共识模块、交易同步模块是决定区块链系统服务质量的关键模块，其他模块过多占用网络资源，会影响这些关键模块，进而影响系统可用性。FISCO BCOS实现了模块级别的流量控制，通过控制非关键的网络流量，优先保证关键模块服务质量，提升系统健壮性。

流量控制的功能

FISCO BCOS从节点和群组两个维度实现了业务到节点的请求速率限制和模块粒度的网络流量限制。前者限制业务到节点的请求速率，以应对大数据量突发请求，保证区块链节点的柔性服务；后者通过限制区块同步、AMOP等非关键模块的网络流量，优先保证共识、交易同步等关键模块的性能和稳定性。



- **节点级别请求速率限制**：限制业务到节点的总请求速率，当请求速率超过指定阈值后，节点会拒绝业务请求，避免节点过载，防止过多的请求导致节点异常；控制节点资源使用量，降低区块链节点之间的资源竞争
- **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设置阈值后，节点收到区块同步请求后会暂缓发送区块、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

群组维度上，主要功能包括：

- **群组级别请求速率限制**：限制业务到群组的请求速率，当请求速率超过阈值后，群组会拒绝业务请求，该功能可在大数据量突发请求的场景下保护区块链节点，控制群组资源使用量，降低群组间的资源竞争
- **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设置阈值后，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

当节点和群组都开启请求速率限制时：

节点收到业务发送的请求包时，首先调用节点级别请求速率限制模块判断是否接收该请求，如请求被接收，则进入群组级别请求速率限制模块，通过该模块检查后的请求才会被转发到相应群组，进行处理。

当节点和群组都开启网络流量控制功能时：

1、节点收到客户端AMOP请求，首先调用节点级流量控制模块判断是否接收该AMOP请求

2、当某个群组收到其他节点对应群组的区块请求后，群组在回复区块之前，需要：

- 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阈值
- 调用群组级流量控制模块，判断群组出带宽是否超过设置阈值，当且仅当节点级和群组级平均出带宽均未超过设置阈值时，该群组才会回复区块请求

如何使用流量控制功能

流量控制配置分别位于config.ini和group.i.ini配置文件的[flow_control]配置项中，分别对应为节点级别流量控制配置和群组级别流量控制。这里向大家介绍如何启用、关闭、配置流量控制。

节点级流量控制

节点级别的网络流量控制配置项均位于config.ini配置文件中，主要包括：

请求速率限制

节点级别的请求速率限制位于配置项[flow_control].limit_req中，用于限制业务每秒到节点的最大请求数目，当请求数目超过设置阈值时，请求会被拒绝。**该配置项默认关闭**，若要开启，请将limit_req配置项前面的;去掉。

打开请求速率限制并设计节点每秒可接受2000个业务请求的示例如下：

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

网络流量限制

- [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**，若要开启，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
```

群组级流量控制

群组级别的网络流量控制配置项均位于group.i.ini配置文件中，主要包括：

请求速率限制

群组i的请求速率限制位于group.i.ini的配置项[flow_control].limit_req中，限制业务每秒到群组的最大请求数目，当请求数目超过配置项的值时，请求会被拒绝。该配置项默认关闭，若要开启，请将limit_req配置项前面的;去掉。

打开请求速率限制并配置群组每秒可接受1000个业务请求的示例如下：

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

群组内网络流量限制

[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**，若要开启，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
```

总结

随着区块链技术的发展，越来越多应用部署于区块链系统中，对区块链系统服务质量的要求也日渐提升，区块链系统的柔性可用、稳定健壮变得更加重要。

FISCO BCOS v2.5引入流量控制功能，是FISCO BCOS对区块链柔性服务探索的重要一步。

社区将持续打磨，优化区块链系统服务质量，希望未来能为海量业务场景提供更好的、高可用的柔性服务。如何做好流量控制的同时，又不影响原本系统性能？敬请关注社区后续文章，为您详解流量控制策略的具体实现原理。欢迎大家共同探讨交流，积极反馈使用的体验与改进建议。

22.4.3 共识算法

共识框架：可插拔设计

共识算法：PBFT、Raft、Rpbft

FISCO BCOS共识优化之路

作者：陈宇杰 | FISCO BCOS 核心开发者

作者语

原来的PBFT共识算法在区块打包、交易验签、区块执行、空块处理等方面有持续优化空间，为了让PBFT算法更快更稳，FISCO BCOS做了一系列优化，包括：

- 打包和共识并发进行；
- 不重复验签交易；
- 引入DAG并行交易执行框架，可并行执行区块内交易；
- 空块快速触发视图切换，并切换Leader，不落盘空块，消除空块落盘存储开销的同时，有效防止了节点作恶；
- 解决了节点宕机后，无法快速追上其他节点视图的问题，保证了系统的可用性。

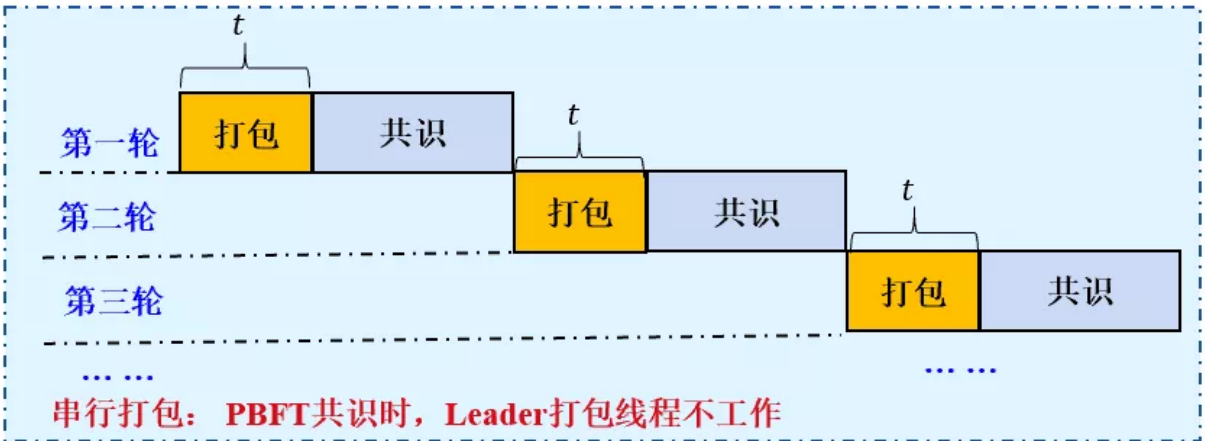
本文分别从性能、存储和可用性三个方面，向大家细说FISCO BCOS的共识优化方案。

性能优化

考虑到**Leader轮流串行打包交易**、**交易验签速度慢**以及**区块执行速度慢**，都是导致性能问题的主要因素，FISCO BCOS做了如下优化：

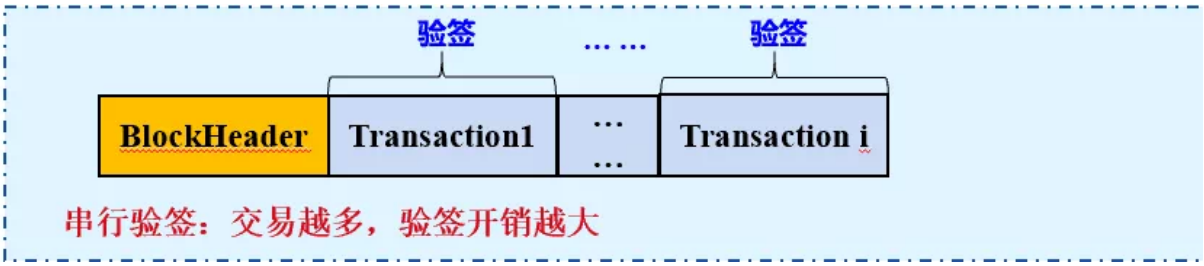
打包和共识并发执行

PBFT共识算法在每一轮共识，都包括**打包阶段**和**共识阶段**，Leader打包新区块时，所有共识节点都处于等待Prepae包的状态中，无法进入共识阶段；共识节点处于共识阶段时，Leader的打包线程不工作，但打包区块和共识是两个独立互斥的过程，可以并发执行。



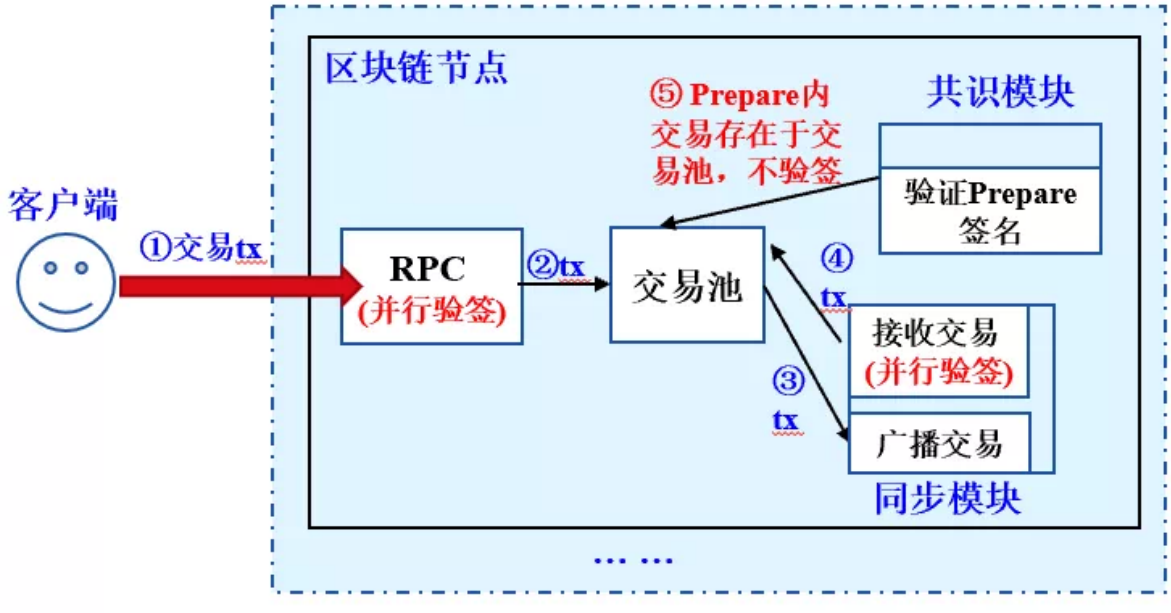
设打包阶段的时间开销为t, 共识阶段时间开销为u，n轮共识的时间开销为n(t+u)；但若下一轮共识的Leader参与共识阶段的同时，也提前打包区块，并在下一轮共识时，广播已经打包好的区块，则可将共识时间开销缩短为nu+t，时间开销降低了(n-1)*t，可以有效提升PBFT共识算法性能。

避免交易重复验签



共识节点收到Leader发送的Prepare包后，会从中取出区块，并验证区块内每笔交易签名的有效性，但交易验签是很耗时的操作，会增加PBFT Prepare阶段的时间开销，降低性能。

考虑到交易插入到交易池的时候，会进行一次验签，如下图所示，FISCO BCOS系统做了防止交易重复验签的优化，下面结合整个交易流的处理流程，详细说明FISCO BCOS防止交易重复验签的处理流程：



1. RPC接收客户端发送的交易后，进行交易验签；
2. 交易验证通过后，被插入到交易池，同步模块广播交易；
3. 其他节点的同步模块收到其他节点的交易后，并进行验签交易，并将有效交易插入到交易池；
4. 共识模块收到Prepare包后，解出Prepare包内区块，判断区块内交易是否在交易池内，仅验证不包含在交易池内的交易签名。

经过上述优化后，将Prepare请求内包含10000笔交易的区块解码和验签时间，由2s降低为200ms，大大减少了Prepare阶段的时间开销。

区块并行执行

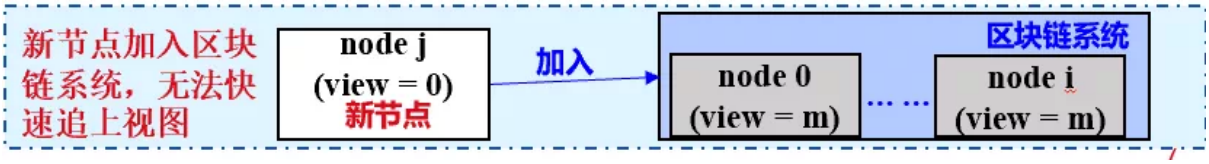
区块执行是PBFT共识算法的主要时间开销之一，没有做任何并行优化的情况下，PBFT共识算法几乎无法就一个包含上万笔交易的区块达成共识。

为了提升区块链系统TPS，FISCO BCOS系统开发了基于DAG的交易并行执行引擎，并引入了可并行的合约开发框架，支持并行执行交易，达到了上万的TPS。具体可参考这里：《区块链性能腾飞：基于DAG的并行交易执行引擎》。

存储优化

为保障系统正常运行、确认Leader可用、防止Leader故意作恶，基于PBFT共识算法的区块链系统在没有交易时，会产生空块，并就空块达成共识。虽然空块共识是必要的，但考虑到当前区块链系统的QPS不大，落盘空块会耗费存储空间、降低硬盘利用效率(可存储的交易数)。FISCO BCOS基于PBFT共识算法，实现了高效的空块处理方法，保证空块参与PBFT共识流程的同时，不落盘空块，提升了磁盘利用效率。详细方案可参考这里：《FISCO BCOS的PBFT空块处理》。

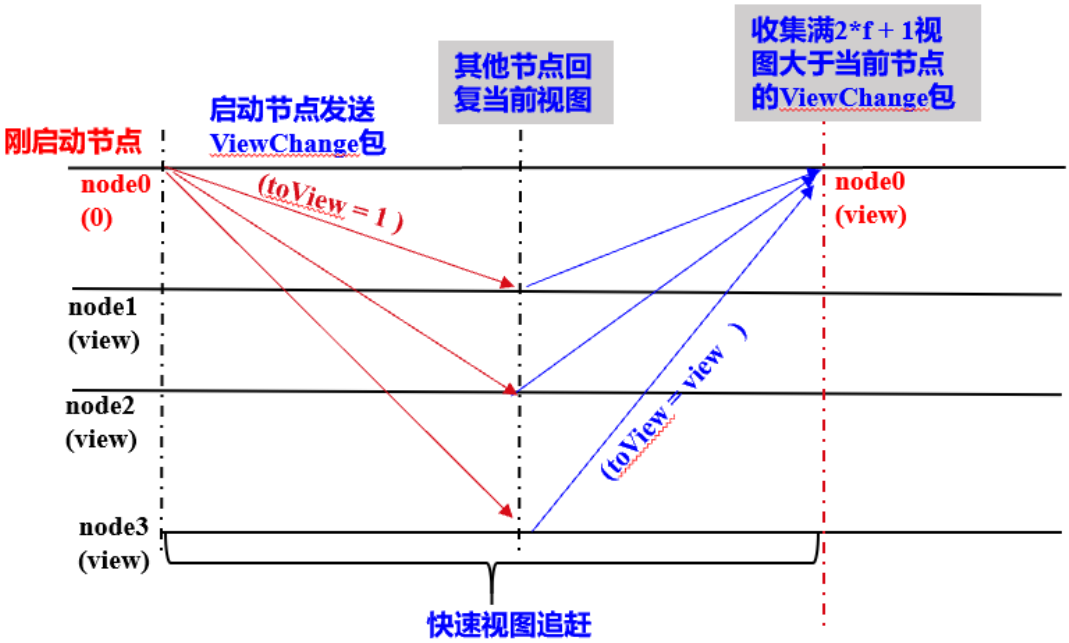
可用性优化



刚启动的节点或者新节点加入区块链网络时，若不能立即和其他节点视图达成一致，会影响系统容错性。

- case1：4节点区块链系统，node0宕机，剩余三个节点容错节点数为0；node0重启若无法快速追上其他节点视图，系统可容错节点数仍然为0，且node0处于共识异常状态中。
- case2：2节点区块链系统正常运行，新加入节点node2，若node2无法快速追上其他节点视图，系统会由于1个节点异常(新加入节点)而处于共识异常状态中。

针对上面问题，FISCO BCOS PBFT共识算法引入了快速视图追赶机制，刚启动节点向所有其他共识节点发送视图切换包，其他节点收到包后，向其回复最新视图，从而使得刚启动节点可快速和其他共识节点达成一致视图，系统在加入了新节点后也不会共识异常。



如上图所示，核心流程如下：

- 刚启动节点向所有其他节点广播视图切换请求包ViewChange，请求包内的视图ViewChange.toView为1；
- 其他节点收到toView远小于当前节点视图的ViewChange请求后，回复包含当前视图(view)的ViewChange包；
- 刚启动节点收集满2*f+1个ViewChange包后，切换到和其他共识节点一致的视图。

总结

以上详述了FISCO BCOS在共识算法上的优化策略，FISCO BCOS使用系统的方法使得PBFT算法性能和存储效率更高，可用性更强。当然，除了前面提到的问题外，PBFT算法在网络复杂度上也有持续优化空间，FISCO BCOS开发团队也在积极调研最新的共识算法及共识算法优化策略，并寻求大规模节点共识的解决方案，敬请期待。

FISCO BCOS的PBFT空块处理

作者：陈宇杰 | FISCO BCOS 核心开发者

作者语

为了保障系统正常运行、确认Leader可用、防止Leader故意作恶，基于PBFT共识算法的区块链系统(如Algorand) 在没有交易时，会产生空块。

在常见的区块链网络里，记账者通常根据算法持续出块，目的是为了保障系统正常运行、防止作恶等，即使区块里不包含交易，空区块也会被共识确认和落盘存储。

虽然对空块进行共识有一定作用，但空块落盘会耗费存储空间、降低硬盘利用率 (可存储的交易数)，并一定程度上影响面向区块数据的回溯和检索效率。

所以，FISCO BCOS基于PBFT共识算法，实现了高效的空块处理方法，保证每个区块都参与PBFT共识流程的同时，不落盘空块，提升了磁盘利用效率，也确保了系统的安全性、健壮性。

名词解释

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。

视图(view)

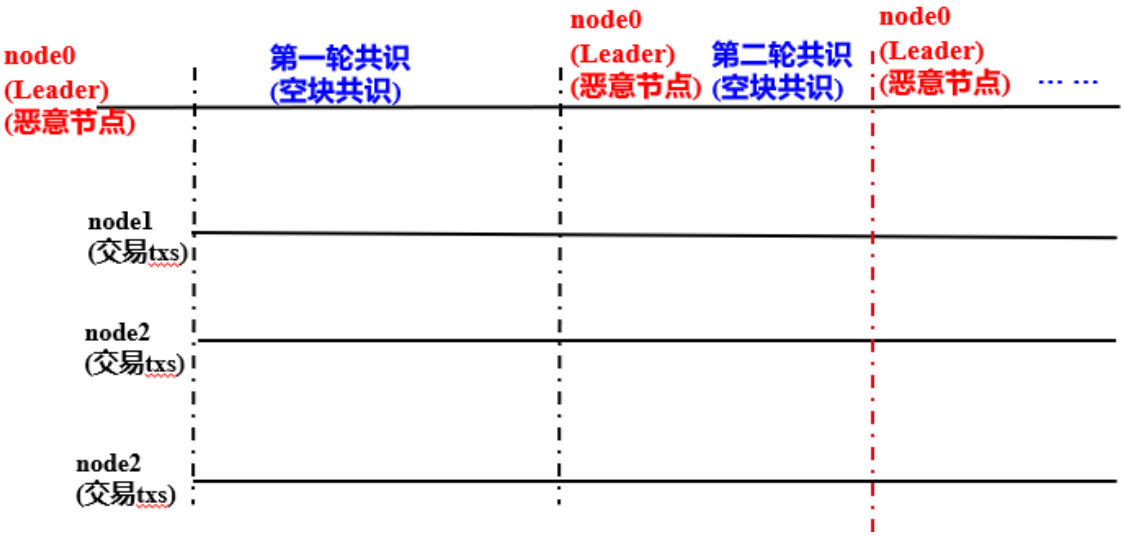
PBFT共识算法使用**视图view记录每个节点的共识阶段，相同视图节点维护相同的Leader和Replica节点列表**。当Leader出现故障，会发生视图切换，并根据新视图选出新Leader。

FISCO BCOS系统中，Leader计算公式如下：

```
leader_idx = (view + block_number) % node_num
```

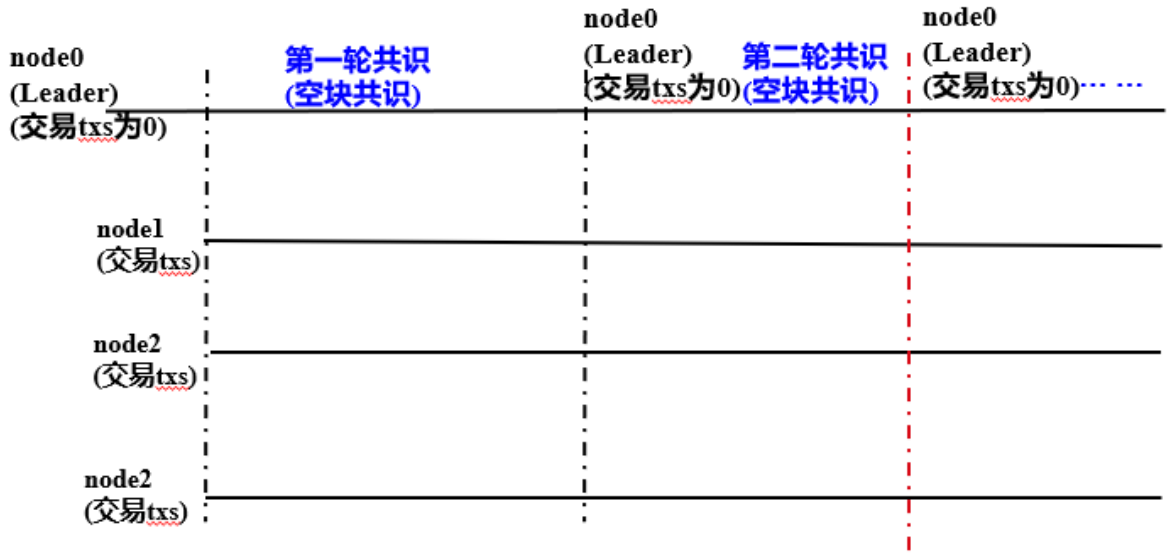
为什么共识空块须切换Leader

防止Leader故意作恶



如上图，node0是不可信的当前Leader，若共识空块后不切换Leader，该节点一直向其他节点广播空块，使得Leader一直是node0，导致系统无法处理正常交易。共识空块后，切换Leader，可切换Leader为可信节点，有效防止Leader作恶问题。

防止切换到无交易Leader导致的系统异常



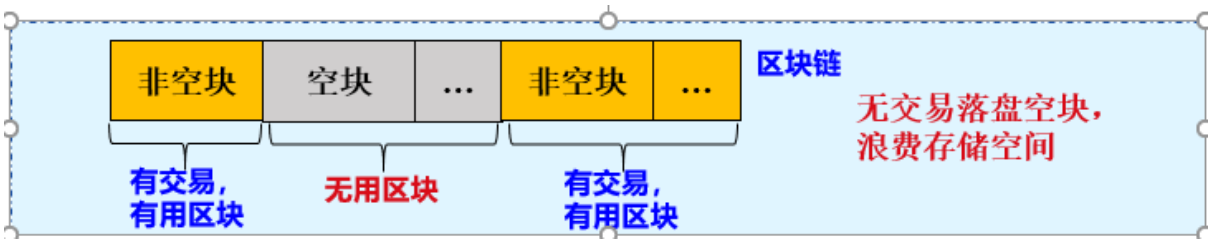
如上图，node0是可信的当前Leader，但其交易池内可打包的交易数为0，共识空块后不切换Leader会导致该节点一直向其他节点广播空块，系统Leader一直是node0，无法处理正常交易。共识空块后，切换Leader，可切换Leader为有交易的节点，保障了系统正常运行。

落盘空块存在的问题

浪费存储空间

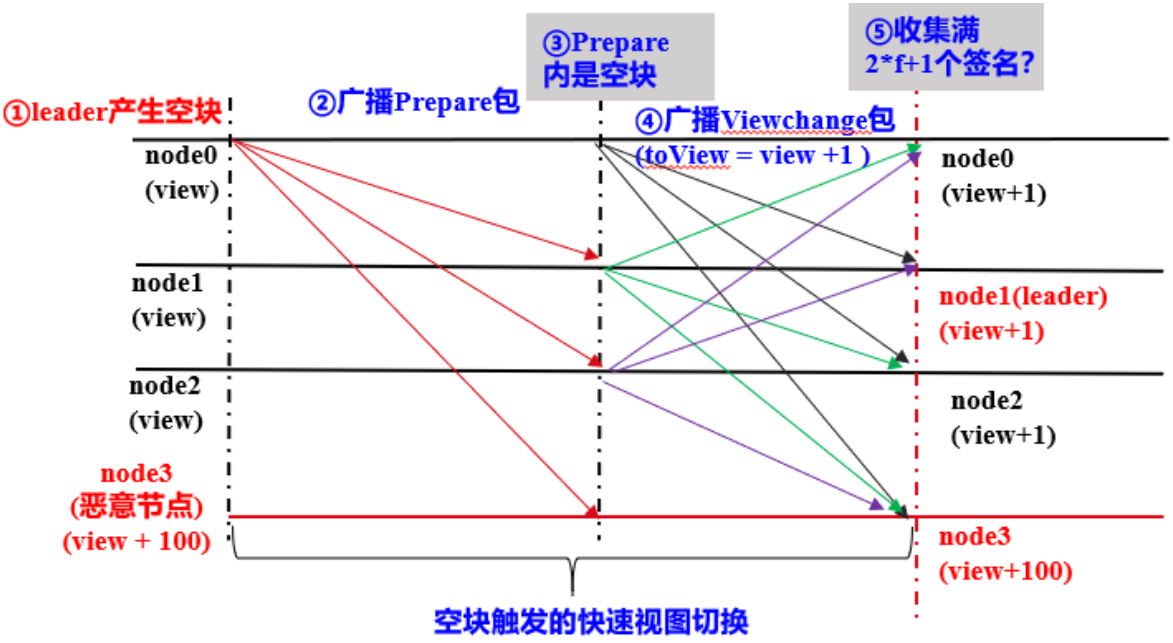
有些业务在一天中有忙闲的时段之分，比如，在半夜里，可能有大段的时间并没有用户在交易，这个时候如果持续出块，会持续有空块生成。

例：某区块链系统1s出一个块，1天有50%的时间没有交易，每个空块大小为1KB，若这些空块都落盘，则一天空块占据的磁盘空间为：3600s/h * 24h * 50% * 1KB ≈ 43.2MB，1年空块占据约为15.7GB磁盘空间。假设平均每个交易大小为1KB，这15.7GB磁盘空间可用于存储15.7GB/1KB=15,700笔交易。



FISCO BCOS PBFT空块处理方案

如下图所示，FISCO BCOS PBFT共识算法，通过空块触发快速视图切换的方法，达到切换Leader目的，同时不落盘空块。



核心流程

结合上图，下面介绍FISCO BCOS PBFT空块处理算法的主要流程：

1. Leader(node0)节点在指定时间间隔(当前是1秒)没有打包到交易，则基于最高块构造一个空块，内含0笔交易；
2. Leader将空块封装在Prepare包内，并广播给所有其他共识节点；
3. 其他共识节点收到Prepare包后，取出区块，若区块是空块，则将节点即将切换的视图toView设置为当前视图加一，并相互间广播视图切换请求viewchange_request，viewchange_request中的视图为toView，即视图增加当前view；
4. 共识节点收集视图切换包：节点收集满n- f (n是共识节点数，至少为3*f+1；f是系统可容错的节点数目) 个来自不同节点、视图与节点toView值一致的视图切换请求后，触发视图切换，将当前视图view切换为toView；
5. 由于视图切换，下一轮共识的Leader切换为下一个节点(即node1)。

总结

综上可知，FISCO BCOS PBFT共识算法，通过空块触发快速视图切换，并切换Leader优化了空块处理流程，解决了共识空块不切换Leader导致的系统异常，实现了空块不落盘存储，提升了磁盘利用效率，加快了数据追溯的效率，减少了数据分析的复杂度。

我们为什么这样设计RPBFT?

作者：陈宇杰 | FISCO BCOS 核心开发者

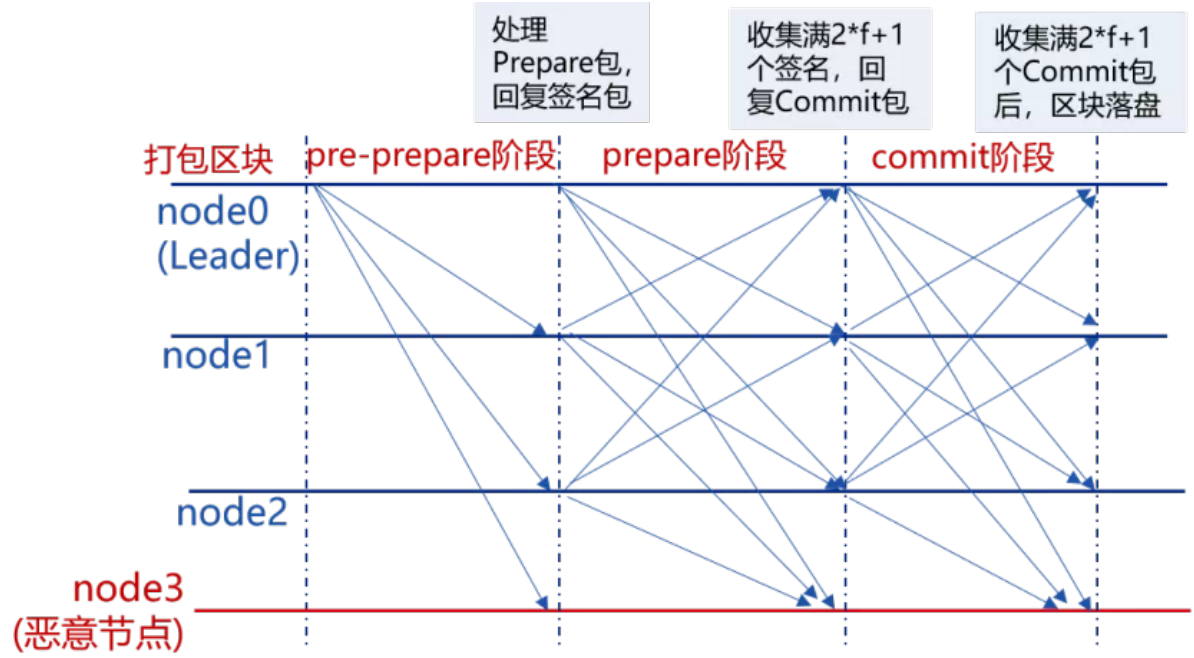
前言

共识模块是区块链系统的引擎，在保证各区块链节点数据一致性方面起到至关重要的作用。FISCO BCOS引入可插拔的共识引擎后，同时支持PBFT和Raft两种共识算法，相较于采用POW共识算法的区块链系统，交易确认时延更低、吞吐量更高，可满足当前大部分联盟链系统的性能需求。

PBFT共识算法更因其可容忍拜占庭错误而天然适用于区块链系统。可PBFT共识算法也存在可扩展性低的问题。FISCO BCOS团队从2019年开始一直致力于研究新的共识算法，以同时保障区块链系统的性能和可扩展性。FISCO BCOS 2.3版本中发布的RPBFT共识算法为研究成果之一。本文将详细介绍RPBFT共识算法的设计目的与其中的技术实现。

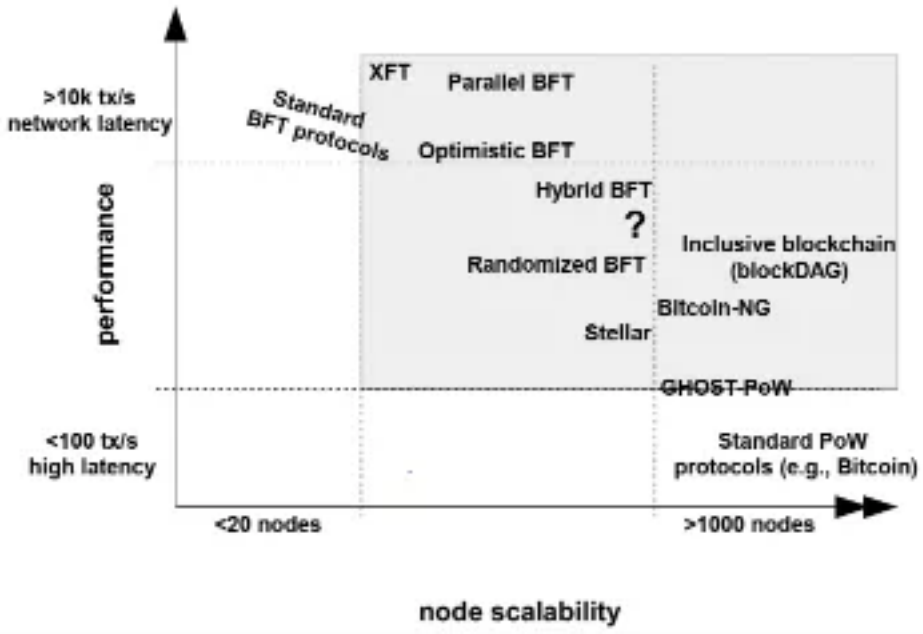
PBFT共识算法挑战

介绍RPBFT共识算法之前，先来看看PBFT共识算法的挑战，以及学术界的对应解决方案。PBFT共识算法起源于上世纪。1999年，Miguel Castro(卡斯特罗)和Barbara Liskov(利斯科夫)提出PBFT共识算法，将BFT算法复杂度从指数级降到多项式级，使得PBFT共识算法就可应用于实际系统中。不同于POW共识算法，PBFT基于分布式一致性原理保障分布式系统最终一致性。由于摒弃了算力，PBFT共识算法性能更高、交易确认时延更低，加之基于密码学技术的防作恶机制，天然适用于联盟区块链系统。可PBFT共识算法也有"软肋"，从其三阶段共识流程得以一窥。

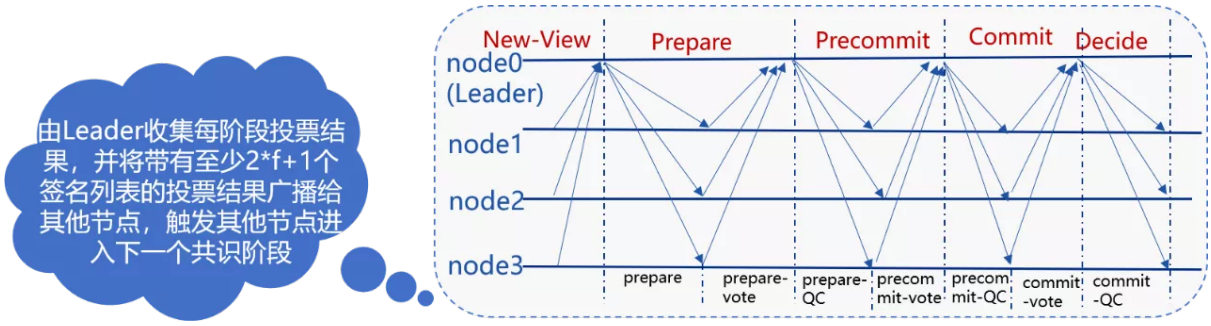


从上图可看出，PBFT共识流程中，节点之间需要相互广播共识消息包，且网络复杂度与节点数目的平方成正比，严重限制了PBFT的可扩展性。

下图整理自IBM研究员Marko调研，反映了采用不同共识算法的区块链系统节点规模与交易延迟之间的关系：



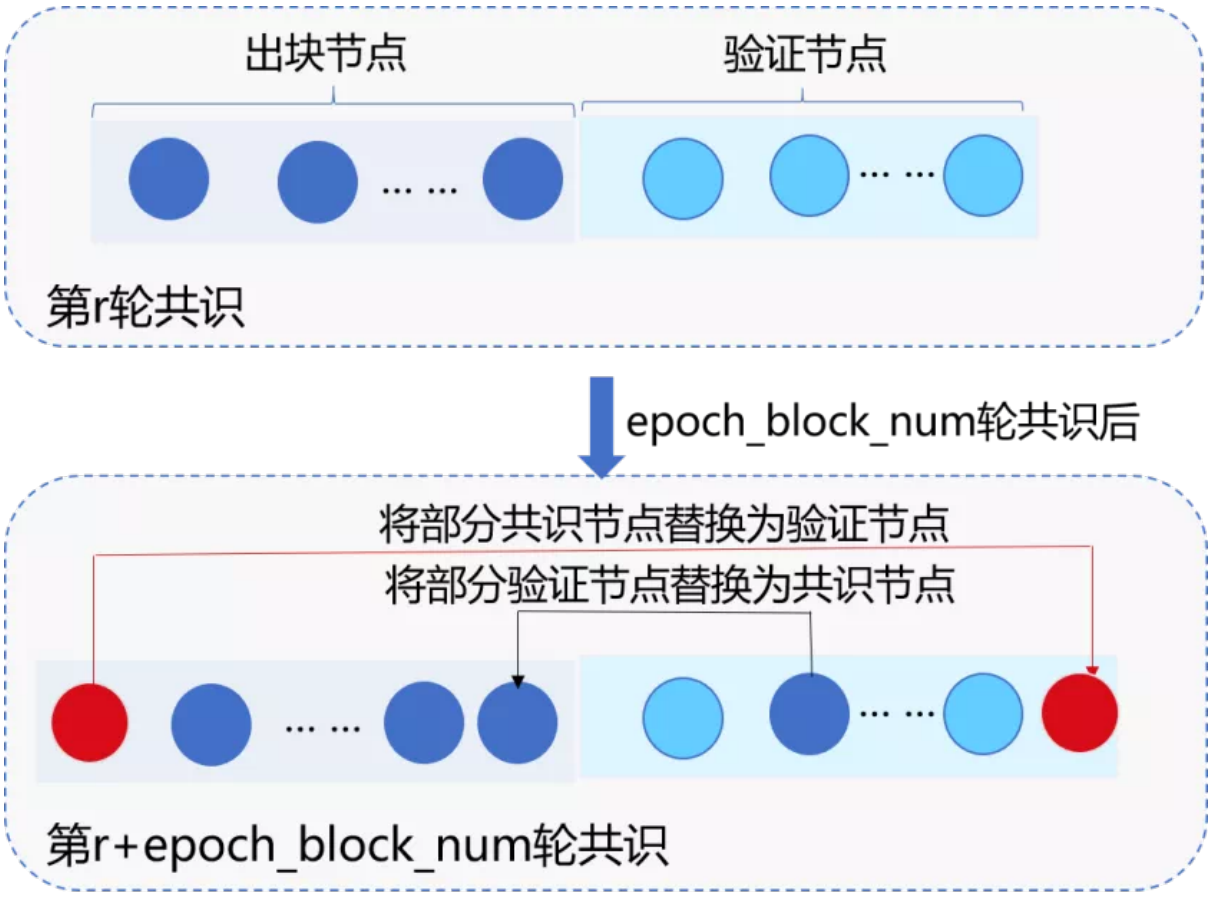
从中可看出，BFT类共识算法性能很高，但其支撑的节点规模最多不超过1000。2019年，HotStuff成为各区块链平台争相研究的共识算法，相对于PBFT，HotStuff具备算法简单、网络复杂度与节点规模成线性关系等多种优势。下图是HotStuff的核心流程：



由于HotStuff复杂度仍然与节点规模成正比，没法从根本上解决共识算法可扩展性问题，且HotStuff每个阶段都依赖于Leader收集、广播消息包，Leader会成为每轮共识的瓶颈。基于以上调研，FISCO BCOS团队先后实现了PBFT分组共识算法、HotStuff共识算法。但随着节点规模的增加，这些共识算法的性能、吞吐率逐渐下降。因此，我们开始探索一种不会因节点数量增加而导致区块链系统性能快速线性下降的共识机制，RPBFT共识算法在这种情况下应运而生。

RPBFT共识算法核心思想

RPBFT共识算法的目标是在保障区块链系统性能、安全性的前提下，将共识算法网络复杂度与共识节点规模解耦，提升区块链系统的可扩展性。为实现这个目标，FISCO BCOS团队参考DPOS思路，在大节点规模下，随机选取部分节点作为“共识委员节点”参与每轮PBFT共识，由于共识委员节点数目固定、与节点规模无关，因此RPBFT共识算法可扩展性更强。此外，为了保障系统安全、防止共识委员节点联合作恶，RPBFT算法会周期性地替换共识委员节点，如下图所示



RPBFT共识算法实现方案

RPBFT算法主要包括2个系统参数:

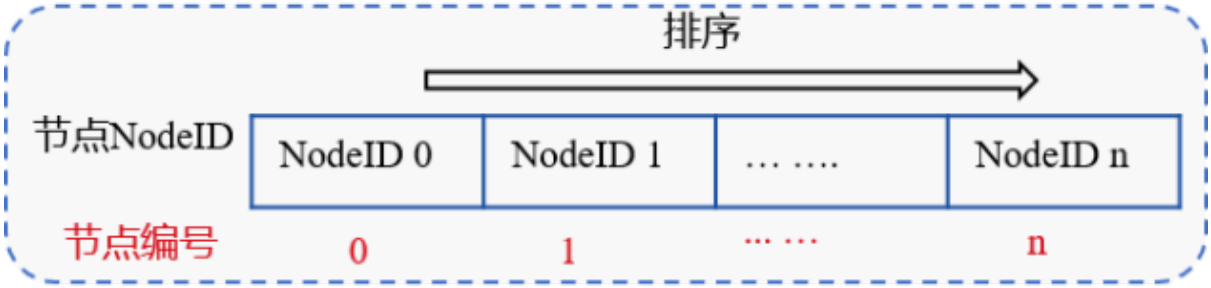
- epoch_sealer_num：每轮共识过程中参与共识的节点数目，可通过控制台发交易方式动态配置该参数。
- epoch_block_num: 共识节点替换周期，为防止选取的共识节点联合作恶，RPBFT每出epoch_block_num个区块，会替换若干个共识委员节点，可通过控制台发交易的方式动态配置该参数。

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的最新生效块高，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:

RPBFT共识算法核心算法流程如下:

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。

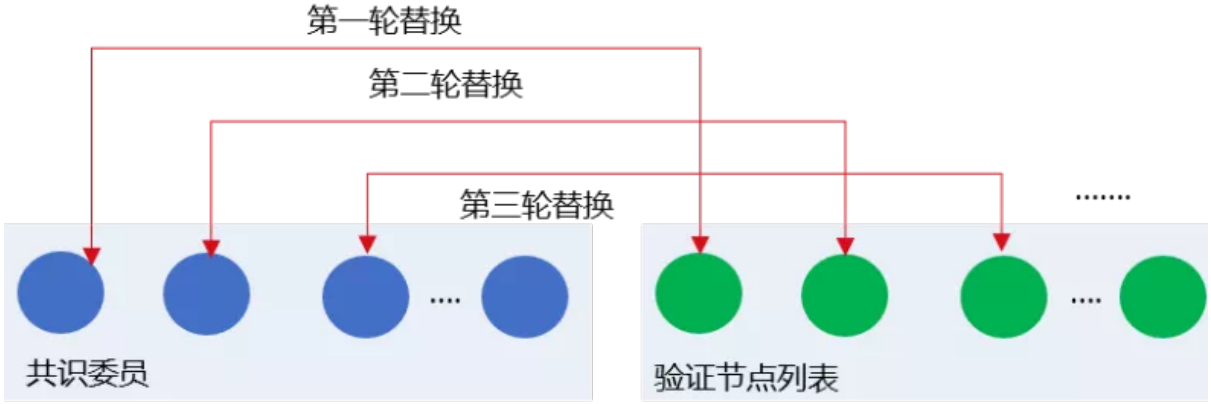
共识委员节点运行PBFT共识算法

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验证节点共识产生的区块，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节点的签名
- 校验区块执行结果：本地执行结果须与共识委员产生的区块执行结果一致

动态替换共识委员节点列表

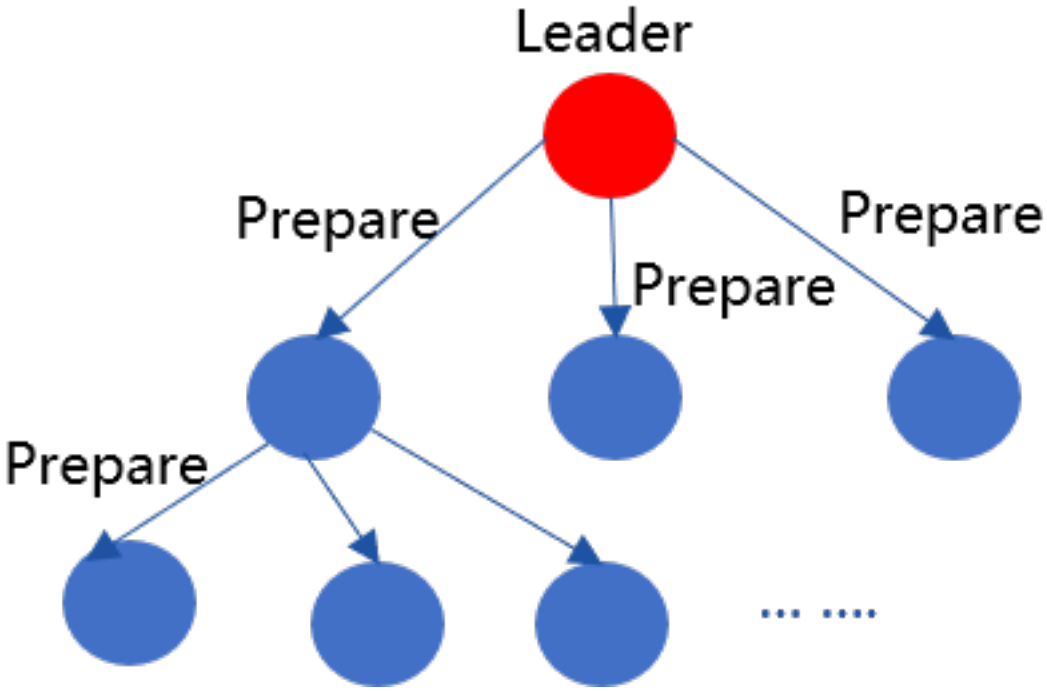
为保障系统安全性，RPBFT算法每出epoch_block_num个区块后，会在共识委员列表中剔除若干个节点，并加入若干个验证节点，下面以每epoch_block_num个区块剔除一个节点为例：



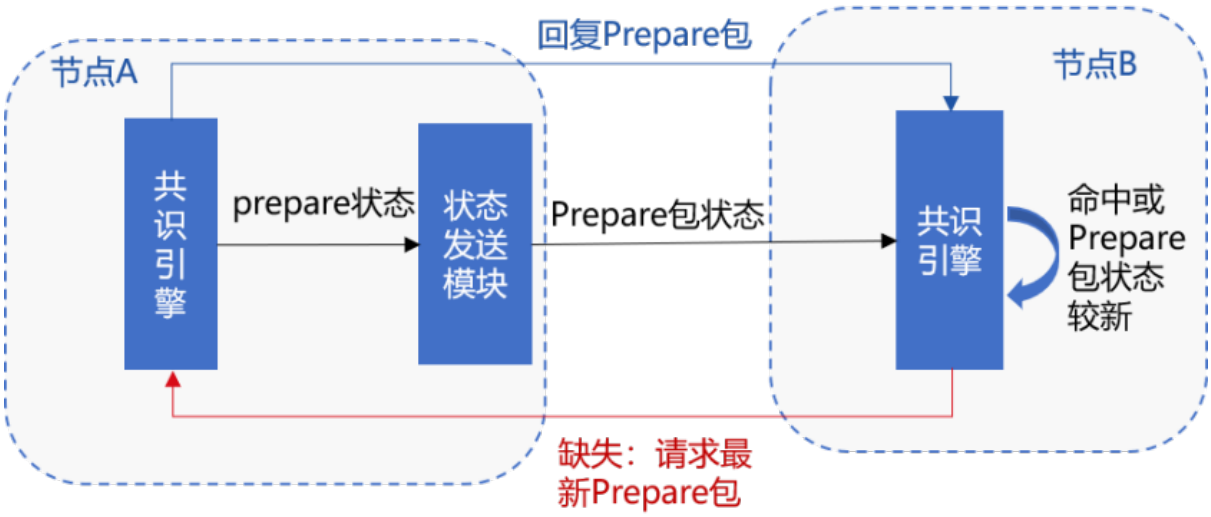
在目前的RPBFT算法实现中，共识委员列表节点被轮流替换为验证节点，设当前有序的共识委员会节点列表为CommitteeSealersList，共识节点总数为N，则共识epoch_block_num个区块后，会将CommitteeSealersList[0]剔除出共识委员列表，并加入索引为(CommitteeSealersList[0].IDX + epoch_sealer_num) % N的验证节点。第i轮替换周期，将CommitteeSealersList[i % epoch_sealer_num]剔除出共识委员列表，加入索引为(CommitteeSealersList[i%epoch_sealer_num].IDX + epoch_sealer_num) % N的验证节点。

RPBFT网络优化

考虑到Prepare数据包较大，占网络开销大头，为了进一步提升RPBFT共识算法可扩展性，我们在FISCO BCOS 2.3引入了Prepare包广播优化。将Leader因广播Prepare包产生的出带宽流量分摊给其下属子节点，即：Leader产生Prepare包后，沿着树状拓扑将数据包传播给其他节点，如下图所示：



为保证在节点断连情况下开启树状广播时，Prepare包仍能到达每个节点，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。主要判断依据包括：

   (1) prepareStatus.blockNumber是否大于当前块高

   (2) prepareStatus.blockNumber是否大于localPrepare.blockNumber

   (3) 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。

区块落盘后，为降低多个验证节点、观察节点向共识委员节点同步区块时，共识委员节点的网络出带宽对网络可扩展性的影响，RPBFT采用了区块状态树状广播策略，详细可参考FISCO BCOS同步模块的优化策略。

RPBFT算法优化展望

FISCO BCOS 2.3初步实现了RPBFT共识算法，消除了节点规模对共识算法复杂度的影响。但目前实现的RPBFT共识算法，仍有待改进的地方，诸如：共识委员节点选取替换规则比较简单等。未来计划引入VRF可验证随机数算法来实现私密、随机、非交互式的共识委员节点选取方法，欢迎大家体验并反馈意见。

小结

本文介绍了BFT类算法的挑战以及FISCO BCOS团队在共识算法领域探索的初步成果。分布式系统共识是大而复杂的领域，FISCO BCOS 2.3发布的RPBFT算法目前仅解耦了节点规模对网络复杂度的影响，是实现高安全、可扩展共识算法的第一步，未来还会引入VRF算法来保证共识委员节点选取的安全性，敬请期待。

22.4.4 关于存储

存储设计：支持KV和SQL 引擎类型：支持leveldb、rocksdb、mysql CRUD接口：提供CRUD接口访问链上数据

FISCO BCOS基于表的存储结构

作者：尹强文 | FISCO BCOS 核心开发者

FISCO BCOS底层的存储数据结构，并没有采用传统的MPT存储结构，而是用了基于表结构的方式。一方面，避免了世界状态急剧膨胀而导致的性能下降问题；另一方面，表结构能够兼容各种存储引擎，使得业务开发更加方便。

FISCO BCOS表的分类

FISCO BCOS每张表都有一个主key字段，1个或者多个value字段，表分为系统表（以_sys_开头），用户表（以_user_开头）和StorageState账户表（以_contract_data_开头）。表结构的所有记录，都会有_id_,_status_,_num_,_hash_内置字段。用户表和StorageState账户表key字段的类型是varchar(255)，value的类型为mediumtext。

系统表

系统表默认存在，由节点进程或者amdb-proxy进程启动时，保证系统表的创建，每张表的说明如下。

- ***sys_tables***

存储所有表的结构，每一张表都会在这张表中有一条记录，记录表的结构，包括表的key和field字段。表结构如下：

以表名为_sys_tx_hash_2_block_为例，_sys_tables_中各个字段数据为：

```
table_name=_sys_tx_hash_2_block_key_field=hashvalue_field=value,index
```

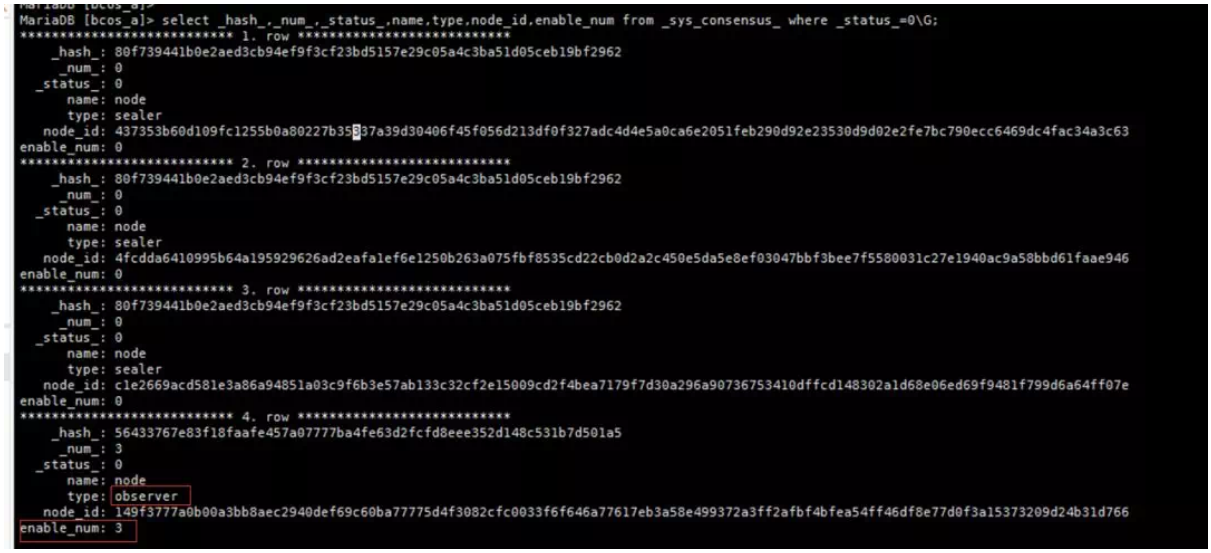
底层创建表和读取结构表，是基于_sys_tables_这张表，从创建好的_sys_tx_hash_2_block_表中可以看到，这张表包含3个字段，分别是主key字段hash，value字段有2个分别是value和index。

- ***sys_consensus***

存储共识节点和观察节点的列表。表结构如下：

例如，某个链包括4个节点，初始化都是共识节点，可以看到四个节点都是sealer（共识节点），生效块高都是0，如图：

通过控制台将`149f3777a0...`这个节点移除并加入到观察者列表，查询_sys_consensus_表的数据，发现该条记录的type已经修改为observer，生效块高已经修改为3。如图：



- ***sys_current_state***

存储当前区块链最新的状态，每次有区块数据存储，这个表都会去更新信息，包括当前分配出去的自增id，当前块高，交易失败数，交易总数。表结构如下：

存储的信息如下：

- ***sys_config***

存储需要共识的群组配置项，表结构同_sys_current_state_，当前配置了2个数值项，分别是一个区块包含的最大交易数及gas值。写入创世块的时候，会从group.[groupid].genesis文件中读

取consensus.max_trans_num和tx.gas_limit两个配置项并写入表中。存储的信息如下：

- ***sys_table_access***

存储含有写入权限的外部账户地址。表结构如下：

默认这个表是没有数据的，表示所有外部账户都有读写权限，通过控制台使用`grantDeployAndCreateManager`命令为某个账号授权，会在`_sys_table_access_`这张表中新增一条Entry。

```
MariaDB [bcos_a]> select * from _sys_table_access_ where _status_=0\G;
*************************** 1. row ***************************
      _id_: 243
    _hash_: 94b4b88414c55d899e12907c9b86e18c845e510da0a607a2770bd3cea290ab4a
     _num_: 39
  _status_: 0
table_name: _sys_tables_
   address: 0x6a7eca49048302082a963198f719b31a4909cc69
enable_num: 39
1 row in set (0.00 sec)
```

同时可以看到，除了被授权的外部账号可以部署合约之外，其他账户部署合约会提示没有权限。

```
[darren@VM_105_58_centos ~/fisco_bcos/dist_kzt]$ ./start.sh 1
================================================================================================
Welcome to FISCO BCOS console(1.0.4)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________  ______  ______  ______  ______      _______  ______  ______  ______
|        \/      \/      \/      \/      \    |       \/      \/      \/      \
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\   | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$   | $$__/ $| $$   \$| $$  | $| $$___\$$
| $$  \     | $$  \$$    \| $$      | $$  | $$   | $$    $| $$      | $$  | $$\$$    \
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$   | $$$$$$$| $$   __| $$  | $$_\$$$$$$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$   | $$__/ $| $$__/  | $$__/ $|  \__| $$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$   | $$    $$\$$    $$\$$    $$\$$    $$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$     \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$$

================================================================================================
[group:1]> deploy TableTest
{
    "code":-50000,
    "msg":"permission denied"
}
```

- ***sys_number_2_hash***

存储区块号到区块hash映射，可以根据区块号映射到区块hash值。表结构如下：

- ***sys_hash_2_block***

存储hash到序列化的区块数据映射，可以根据区块hash值映射到区块值。表结构如下：

- ***sys_block_2_nonces***

存储区块中交易的nonces，可以根据区块号，映射到该区块生成时所用到的nonces值。表结构如下：

- ***sys_tx_hash_2_block***

存储交易hash到区块号的映射，表结构如下：

一个区块可能包括多个交易，因此，区块hash和交易hash是一对多的关系，所以一个区块会在这个表里生成多条数据。

- ***sys_cns***

存储合约名到合约地址的映射，表结构如下：

用CNS部署的合约，可以通过合约名去调用，具体方式是根据合约名，找到包括多个版本号的合约地址列表，筛选出版本号正确的合约地址，再使用`_contract_data_+Address+_`作为表名，去查询code的值，执行合约代码。

例如，通过CNS部署的TableTest合约，可以在_sys_cns_表中查询到如下数据：

```
MariaDB [bcos_a]> select * from _sys_cns_\G;
*************************** 1. row ***************************
     _id_: 157
   _hash_: f63838dd4b5bf4078960bdf3eb0a3ce86ff30b7f9ab594b99e091714ed9e36f7
    _num_: 23
_status_: 0
    name: TableTest
 version: 2.0
 address: 0x2e8e5d7f11699537464a1c03acbc8e8ca79b0506
     abi:
```

用户表

用户调用CRUD接口所创建的表，以_user_<TableName>为表名，底层自动添加_user_前缀。

表名和表结构由合约决定，例如创建表的合约代码为：

```
TableFactory tf = TableFactory(0x1001); int count = tf.createTable("t_test", "name
↪", "item_id,item_name"); return count;
```

创建出来的表名为_user_t_test，包括3个字段，分别是主key字段name，类型为varchar(255)；value包含两个字段，分别是item_id，item_name；如图：

```
MariaDB [bcos_a]> desc _user_t_test;
+-----------+------------------+------+-----+---------+----------------+
| Field     | Type             | Null | Key | Default | Extra          |
+-----------+------------------+------+-----+---------+----------------+
| _id_      | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| _hash_    | varchar(128)     | NO   |     | NULL    |                |
| _num_     | int(11)          | NO   | MUL | NULL    |                |
| _status_  | int(11)          | NO   |     | NULL    |                |
| name      | varchar(255)     | YES  | MUL |         |                |
| item_id   | mediumtext       | YES  |     | NULL    |                |
| item_name | mediumtext       | YES  |     | NULL    |                |
+-----------+------------------+------+-----+---------+----------------+
```

StorageState账户表

_contract_data_+Address+_作为表名。表中存储外部账户相关的信息。存储信息如下：

以部署TableTest合约为例，deploy TableTest会返回一个地址，如图：

```
[group:1]> deploy TableTest
contract address: 0xa582f529ff55e6ca2ada7ad3bab3b97e1c7013f2
```

同时，可以看到数据库中生成了一个表名为_contract_data_a582f529ff55e6ca2ada7ad3bab3b97e1c7013f2_表，存储信息如下（和上述表述一致）：

```
MariaDB [bcos_a]> select `key` from _contract_data_a582f529ff55e6ca2ada7ad3bab3b97e1c7013f2_;
+----------+
| key      |
+----------+
| alive    |
| balance  |
| code     |
| codeHash |
| nonce    |
+----------+
5 rows in set (0.00 sec)
```

总结

基于表的存储方式，抽象了区块链的底层存储模型，实现了类SQL的抽象存储接口，支持多种后端数据库。引入基于表的存储方式后，数据读写请求不经过MPT，直接访问存储，结合缓存机制，存储性能相比基于MPT的存储有大幅提升。MPT数据结构作为可选方案仍然保留。

「群提问」

Q：腾龙(何直群)：table 和 智能合约内部存储有优缺点对比吗?

A：Wheat：table是类似传统数据库使用方式的设计，可以让写业务逻辑的开发比较容易上手。数据基于table存储，管理也更方便。相比合约mpt格式存储数据，table格式性能也更高。

Q：王先生：有点疑惑，表存储不就和传统的数据库存储一样了吗，那区块链还有什么用?

A：尹强文：使用什么存储结构，本质上不会改变区块链具有去中心化、不可篡改、不可逆、匿名等特性。只是使用基于表的存储，有一些优势，一是数据基于表存储，管理更方便。相比合约mpt格式存储数据，表格式性能也更高，同时表结构能够兼容各种存储引擎，使得业务开发更加方便。

区块链数据是存在链上还是数据库里?

作者：莫楠 | FISCO BCOS 高级架构师

在回答这个问题之前，首先要理清“**区块链数据**”和“**链上数据**”的概念。

区块链数据

“区块链数据”广义上包括区块链的**区块数据**和区块链的**状态数据**：

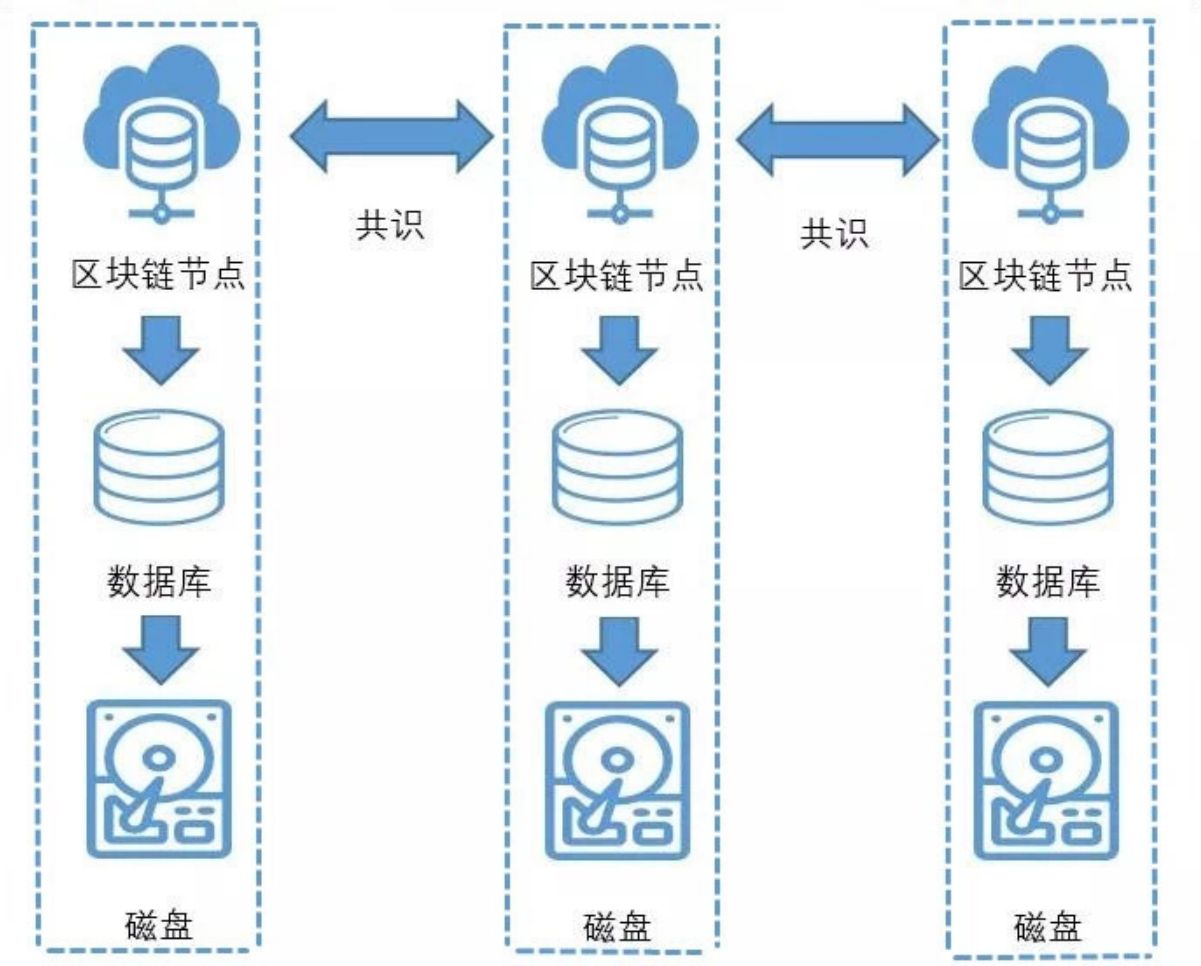
- 区块数据记录了区块链上发生的每一笔交易，譬如小明给小王转账了50元、小王充值了20元等类似这样的交易数据；
- 状态数据记录了区块链上每个账户或智能合约的当前状态，比如小明当前的余额是50元、小王当前的余额是100元。

无论区块数据还是状态数据，它们都是由区块链节点使用和存储的。区块链节点是一个程序，运行在我们的个人电脑、虚拟机或服务器上。多个分布在不同电脑或服务器上的区块链节点，通过网络互相连接，组成了完整的区块链网络。

区块链节点通常会把区块链数据存储在个人电脑、虚拟机或服务器上，存储区块链数据最常见的介质，就是磁盘。

区块链节点不会直接访问磁盘，它们会通过特定的数据库，如LevelDB、RocksDB或MySQL等单机或分布式数据库来操作数据。相比于直接操作磁盘，数据库抽象了特定的数据访问模型，对区块链节点更为友好。

因此，当我们说：“区块链数据保存在数据库”时，可以认为区块链节点将区块链数据保存在MySQL（或其它数据库），MySQL将区块链数据保存在磁盘。



数据库有**独立式**与**嵌入式**之分：

- 独立式数据库，如MySQL、Oracle是通常理解的数据库，独立式数据库作为独立的进程运行，需要单独部署和启停。独立式数据库可以与区块链节点部署在同一台服务器，或者部署在不同的服务器，还支持分布式、集群化的部署。无论何种部署方式，独立式数据库都是区块链节点的存储组件，隶属于区块链节点，与区块链网络无关。
- 嵌入式数据库如LevelDB、RocksDB，它们以动态依赖库或静态依赖库的方式，与区块链节点整合在同一个进程中，同时启停，用户不会明显感受到它们的存在。

链上数据

区块链数据的区块数据和状态数据并不是凭空产生的。区块数据中的交易，是由区块链的用户生成，用户把交易发送到区块链节点，区块链节点将多个交易打包进区块，区块会在区块链网络上广播和共识，区块链网络对区块达成共识后，认同区块中的交易，将交易的执行结果保存到状态数据中。

假设区块链原本的状态数据是：小明当前的余额是50元、小王当前的余额是100元，那么执行了“小明给小王转账了50元”的交易后，状态数据会发生变化，小明当前的余额会变为0元，小王当前的余额变为150元。

区块需要进行区块链共识，状态数据是通过执行区块中的交易生成的，这两类数据都直接或间接跟区块链共识有关系，可以将其称为“链上数据”。那么，“链上数据”的明确定义，就是：链上数据是直接或间接由区块链共识产生的数据。

回到最初的问题

很显然，“链上数据”和“数据库”不是同一个层面的概念，“区块链数据是存在链上还是存在数据库？”这个问题不成立，区块链数据无论是存储在LevelDB、RocksDB、MySQL数据库或直接存储在磁盘，只要是直接或间接由区块链共识产生，都可以视为链上数据。

FISCO BCOS的链上数据

FISCO BCOS的区块链数据，默认是通过RocksDB保存在磁盘中。如果希望把数据保存到MySQL数据库，可以先自行部署一个MySQL数据库，然后修改区块链节点下的群组配置文件，群组配置文件通常位于区块链节点的配置目录下：conf/group.1.ini

```
[storage]
    type=mysql
    db_ip=127.0.0.1
    db_port=3306
    db_username=root
    db_name=db_Group1_A
    db_passwd=******
```

其中：

- type为区块链节点的存储类型，配置为mysql，表示使用MySQL来存储区块链数据；
- db_ip为MySQL数据库的IP地址，如果部署在本机，就是127.0.0.1；
- db_port为MySQL数据库的端口，默认为3306；
- db_username是MySQL数据库的登陆用户名；
- db_name是MySQL数据库中用于存储区块链数据的数据库名，无需先行创建；
- db_passwd是MySQL数据库的登陆密码。

其它未提及的配置项，可保留默认值不修改，完成这些信息的填写以后，确保数据库运行正常，然后重启区块链节点，区块链节点就会将区块链数据保存到MySQL数据库中。FISCO BCOS的区块链，无论是保存在RocksDB还是MySQL中，都可视为链上数据。使用MySQL，可以方便地查看链上数据的大小、结构等信息，如区块的大小、账户的大小等等。

总结

FISCO BCOS提供了灵活的数据存储机制，对于追求便利与性能的场景，可以使用默认的RocksDB；对于偏重审计和治理的场景，可以使用MySQL，满足不同的需求。

关于FISCO BCOS存储，请参考FISCO BCOS分布式存储文档

「群聊互动」

【Q】 **什么的小世界**：区块链上的数据都是增加在上面的，不能删除是么，那么长时间使用，会不会节点的效率不断下降?

【A】 **中大刘忠楠**：节点header存储的交易、状态数据，是限定长度的根哈希值。

【A】**莫楠**：区块链确实会随着使用不断增长，但区块链的数据访问模型通常是kv的，kv模型的查询效率受数据量的影响很小，因此不会明显影响性能。

【Q】 **什么的小世界**：如果在区块链网络之中增加一个节点，该节点会从其他节点通过广播自动拷贝数据过来是么?

【A】**莫楠**：在区块链网络中增加一个节点，这个节点会自动同步其它节点的数据。

为什么从LevelDB切换到RocksDB?

作者：白兴强 | FISCO BCOS 核心开发者

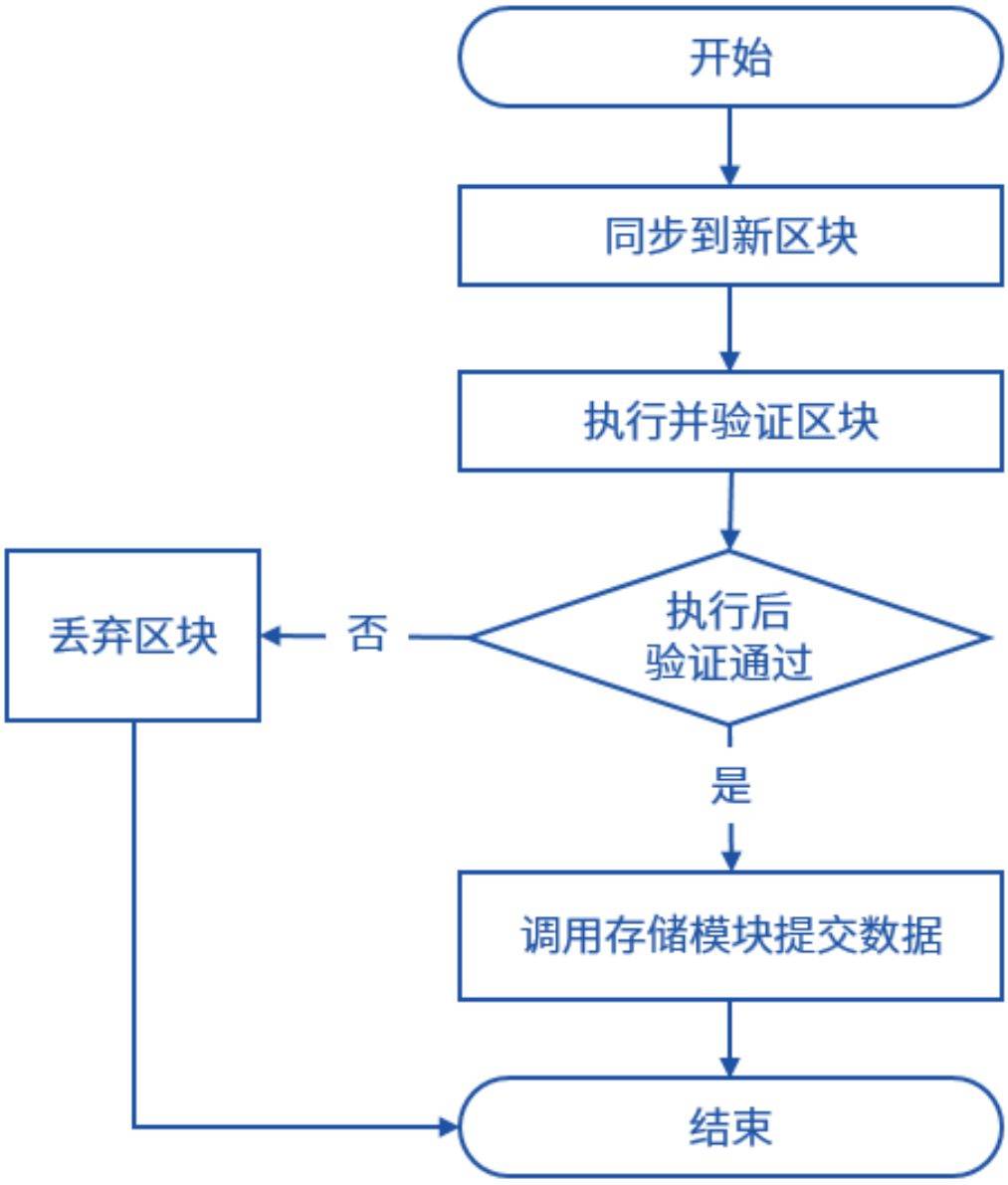
存储模块是区块链底层平台中的核心之一，负责将区块链中所有需要持久化的数据存储到磁盘上。一个优秀的区块链底层平台，必然要有一个强大的存储模块支持。FISCO BCOS存储模块经过多次重构和优化，为性能突破提供了有力支撑。目前，FISCO BCOS单链TPS达到2万+，且支持平行多链的并行扩展。

2.0.0-rc3版本以前，FISCO BCOS支持使用LevelDB和MySQL作为数据存储引擎，rc3之后开始将嵌入式存储引擎从LevelDB切换到RocksDB。为什么要做切换？切换RocksDB之后能带来什么？本文将带大家一起回顾我们作出这个决定时的考虑。

FISCO BCOS存储模块概览

数据提交流程

FISCO BCOS中需要存储的数据可以分为两部分，一部分是经过共识的链上数据，包括交易、收据、区块和合约数据等；另一部分是各节点维持区块链运行所需的数据，包括当前块高、链上交易数和一些交易区块相关的索引信息。区块链上的新区块来自于同步模块和共识模块。以同步模块为例，当拿到新的区块之后，同步模块会调用BlockVerifier模块执行和验证区块，如果验证通过则调用BlockChain模块将区块和执行区块产生的数据提交给存储模块，由存储模块负责将数据序列化写入数据库。



存储模块概览

数据提交到存储模块之后，是一种抽象的表结构，存储模块首先将提交的数据加入缓存层，以提高查询性能。完成缓存的更新之后，需要提交的数据会加入提交队列，由缓存层负责异步提交到适配层，如果关闭了缓存设置，则同步提交到适配层。

适配层需要把提交的数据从FISCO BCOS的抽象表结构组织形式转换为后端对应存储的组织形式，对于MySQL这种关系型数据库，则直接将存储模块的表结构对应到数据库表结构即可，例如_sys_config_这个表在MySQL中如下图所示。

_id_	_num_	_status_	key	value	enable_num
100002	0	0	tx_count_limit	1000	0
100003	0	0	tx_gas_limit	300000000	0

对于RocksDB或LevelDB这种KV的存储模式，将表名和插入时设置的主key拼起来作为数据库的KEY，对应的数据则序列化为VALUE。对应于_sys_config_这个表，以及tx_conut_limit这个主key的数据，其在KV数据库中的KEY为_sys_config__tx_conut_limit，VALUE为对应的数据序列化后的字符串。

为什么选择RocksDB?

FISCO BCOS从1.0版本开始就使用LevelDB作为底层数据存储引擎，在使用过程中我们也碰到一些小问题，例如内存占用高、文件描述符超限导致进程被干掉、节点被kill后可能导致的DB损坏等。

重构2.0版本时，为了更高的性能，我们需要一个更优秀的存储引擎，这个存储引擎应该满足下面这些条件：

1. 开源且有持续地维护；
2. 读写性能要比LevelDB更高；
3. 嵌入式KV数据库，能够支持大数据量场景下的读写；
4. 与LevelDB类似的接口，降低迁移成本。

基于上述条件，RocksDB进入了我们的视野。

RocksDB fork自LevelDB，开源且由fackbook维护，相比于LevelDB有较明显的性能提升，保持了与LevelDB一致的接口，极低的迁移成本。从资料上看非常符合我们的需求。

LevelDB与RocksDB性能对比

下面的测试数据是在一台4 vCPU E5-26xx 2.4GHz 8G 500GB腾讯云硬盘的机器上获取的，由FISCO BCOS核心开发者尹强文提供。

该测试KEY的长度为16字节，VALUE的长度为100字节，压缩算法使用Snappy，其他参数使用默认值，在1千万条数据和1亿条数据的情况下，可看出LevelDB和RocksDB的性能对比：在两种数据量下，各个场景RocksDB都取得了不比LevelDB差或者更好的表现。

	rocksdb_io	rocksdb_耗时（s）	leveldb_io	leveldb_耗时（s）
fillseq	18.4 MB/s	60.28	13.0 MB/s	85.31
fillsync	0.2 MB/s	4453.93	0.2 MB/s	4668.53
fillrandom	11.9 MB/s	92.72	8.0 MB/s	139.05
overwrite	10.0 MB/s	110.91	7.7 MB/s	143.68
readrandom	7.41MB/s	149.77	7.0MB/s	158.03
readseq	285.5 MB/s	3.88	129.1 MB/s	8.57
readreverse	208.8 MB/s	5.3	77.4 MB/s	14.29
readrandom	10.5 MB/s	66.42	7.051 MB/s	70.51
readseq	472.6 MB/s	2.34	153.2 MB/s	7.22
readreverse	359.6 MB/s	3.08	88.4 MB/s	12.52

	rocksdb_io	rocks_耗时（s）	leveldb_io	leveldb_耗时
fillseq	17.0 MB/s	652.6	14.9 MB/s	741.3
fillrandom	7.5 MB/s	1484.2	2.0 MB/s	5648.1
overwrite	5.5 MB/s	2014.1	1.5 MB/s	73651.1
readrandom	5.2 MB/s	1352.6		超过3h没跑出来
readseq	425.9 MB/s	26	30.4 MB/s	363.8
readreverse	320.8 MB/s	24.5	26.4 MB/s	418.3
readrandom	5.6 MB/s	1240.3		超过3h没跑出来
readseq	492.7 MB/s	22.5	31.5 MB/s	350.1
readreverse	367.6 MB/s	30.1	30.5 MB/s	362.1

FISCO BCOS中使用RocksDB

在RocksDB的官方wiki上有一个页面叫做Features Not in LevelDB，这个页面中描述了RocksDB中所有新增的功能，例如对列族的支持，允许我们在逻辑上对数据库分区，对backup和checkpoint的支持，支持备份到HDFS，两种compaction方式允许用户在读放大、写放大、空间放大之间取舍，自带统计模块便于调优，支持了ZSTD等更新的压缩算法等。

官方wiki中也提到RocksDB为提升性能所做的优化，包括多线程Compaction、多线程memtable插入、降低DB锁的持有时间、写锁的优化、跳表搜索时更少的比较操作等。官方文档中指出，在插入key是有序的场景下，RocksDB使用多线程Compaction，使得RocksDB的性能大幅度高于LevelDB。

FISCO BCOS使用RocksDB时只使用了默认参数和与LevelDB兼容的读写接口，并没有做进一步的参数调优，RocksDB在官方文档中有指出，默认参数已经可以达到很好的性能，更进一步的调参并不能带来大幅的性能提升，而用户的业务场景是多种多样的，针对业务场景优化的参数修改对于FISCO BCOS不一定合适。

日后，随着对RocksDB的深入了解，如果发现更优的参数设置，我们也会采用。

总结

为什么换RocksDB，其实就一句话，RocksDB性能更高！凡是能够让FISCO BCOS更加优秀的事情，我们都愿意去做。最近，FISCO BCOS发布了v2.2.0版本，在性能方面作了进一步的优化，每一次的性能提升都是FISCO BCOS的开发者不停死磕的结果，这种死磕我们会一直进行下去，希望社区的同学们也一起参与进来，初学者点个star，进一步了解后可以提一些修复PR或者issue，大家一起让FISCO BCOS更加优秀！

参考链接

Features Not in LevelDB

RocksDB官方wiki

Benchmarking LevelDB vs. Rocks DB vs. HyperLevel DB vs. LMDB Performance for Influx DB

FISCO BCOS CRUD使用指南

作者：廖飞强 | FISCO BCOS 核心开发者

本文将介绍 FISCO BCOS的CRUD功能，帮助开发者更高效便捷地开发区块链应用。

为什么设计CRUD功能?

在FISCO BCOS 1.0中，节点采用MPT数据结构，通过LevelDB将数据存储于本地，这种模式受限于本地磁盘大小，当业务量增大时数据会急剧膨胀，要进行数据迁移也非常复杂，给数据存储带来较大成本和维护难度。

为了突破容量和性能瓶颈，FISCO BCOS 2.0针对底层存储进行了重新设计，实现了分布式存储，带来了容量和性能上的提升。得益于分布式存储采用了库表结构，FISCO BCOS 2.0设计一套CRUD(Create增加、Read读取、Update更新和Delete删除)，让接口更加顺其自然。CRUD面向库表的开发方式符合业务开发习惯，同时也为业务开发提供了另外一种选择(以往只能用Solidity合约)，从而让区块链应用开发更加便利。

CRUD有哪些优势?

CRUD的核心设计思想是提供面向SQL编程的区块链应用开发规范。其好处显而易见，主要体现在两升两降。

提升开发区块链应用的效率

CRUD类似传统业务SQL编程开发模式，大大降低了合约开发难度。开发者将合约当做数据库的存储过程，将区块链数据的读写操作转换为面向表的读写操作，简单易用，极大提升了开发区块链应用的效率。

提升区块链应用的性能

CRUD底层逻辑基于预编译合约实现，其数据存储采用分布式存储，读写数据产生的交易不再进入缓慢的EVM虚拟机执行，而是由高速的预编译合约引擎执行，因此提高了合约的数据读写效率，使得基于CRUD开发的区块链应用具有更高的性能。

降低合约维护和升级的复杂度

CRUD合约的逻辑与存储分离，这样开发者就只需要关心核心业务逻辑，数据不再与特定合约绑定，更方便合约升级。当合约逻辑需要变更时，可以部署新合约，新合约可以读写旧合约创建的表和数据。

降低面向SQL业务的迁移成本

传统商业应用很大一部分通过数据库表的方式管理数据，通过CRUD合约可以将面向SQL设计的商业应用非常顺滑的迁移到区块链上，降低业务的迁移成本。

CRUD如何使用?

介绍了CRUD的两升两降优势，想必大家非常关注CRUD具体如何使用? 其实，CRUD简单易用，目前可以用两种方式使用CRUD功能，分别是CRUD合约和SDK CRUD Service接口。

CRUD合约

开发者与区块链交互的媒介主要是智能合约，CRUD功能自然会集成在智能合约。集成方式非常轻，Solidity合约只需要引入FISCO BCOS官方提供的Table.sol抽象接口合约文件即可。Table.sol包含分布式存储专用的智能合约接口，其接口实现在区块链节点，可以创建表，并对表进行增删改查操作。引入了该抽象接口的合约称为CRUD合约，以区别于没有引用该接口的Solidity合约。

Table.sol抽象接口合约文件包括以下抽象合约接口，下面分别进行介绍。

TableFactory合约

用于创建和打开表，其固定合约地址为0x1001，接口如下：

Entry合约

Entry代表记录对象，一个Entry对象代表一行记录，其接口如下：

Entries合约

Entries是记录集合对象，Entries用于存放Entry对象，其接口如下：

Condition合约

查询、更新和删除记录时指定的过滤条件对象，其接口如下：

Table合约

用于对表进行增删改查操作的对象，其接口如下：

有了以上对Table.sol抽象接口合约的了解，现在可以正式进行CRUD合约的开发。以官方提供的CRUD合约TableTest.sol为示例，介绍其开发的三步曲：

第一步：引入Table.sol

将Table.sol合约放入TableTest.sol同级目录，并在TableTest.sol合约文件中引入Table.sol，其代码如下：

```
import "./Table.sol";
```

第二步：创建表

在TableTest.sol合约文件中，创建表的核心代码如下：

```
// 创建TableFactory对象，其在区块链上的固定地址是0x1001
TableFactory tf =TableFactory(0x1001);
// 创建t_test表，表的主键名为name，其他字段名为item_id和item_name
int count =tf.createTable("t_test", "name","item_id,item_name");
```

注：

- createTable执行原理：createTable执行成功之后，将会在区块链系统表_sys_tables_(区块链启动会自动创建该表，专门记录区块链中所有表的信息)中插入t_test的表信息，即表名，主键名和其他字段名，但并没有正式创建该表。当对t_test表进行增删改查操作时，会首先判断t_test表是否存在，若不存在，则会查询_sys_tables_表获取t_test表的信息，如果查询有t_test表信息，则创建该表，否则执行失败。t_test表存在，则继续执行增删改查操作。
- 这一步是可选操作：比如新合约只是读写旧合约创建的表，则不需创建表这步操作。

第三步：针对表进行CRUD操作

在TableTest.sol合约文件中，插入记录核心代码如下：

```
TableFactory tf =TableFactory(0x1001);
// 打开创建的t_test表
Table table =tf.openTable("t_test");
// 创建Entry对象，即创建一条空记录
Entry entry = table.newEntry();
// 设置Entry对象，即设置记录的字段值
entry.set("name", name);
entry.set("item_id",item_id);
entry.set("item_name",item_name);
// 调用Table的insert方法插入记录
// 返回插入影响的记录数，值为1则插入成功，否则插入失败
int count = table.insert(name,entry);
```

查询记录核心代码如下：

```
TableFactory tf =TableFactory(0x1001);
Table table =tf.openTable("t_test");
// 创建Condition对象，条件为空表示不过滤，也可以根据需要使用条件过滤
Condition condition =table.newCondition();
// 调用Table的select方法查询记录，获得记录集合
Entries entries =table.select(name, condition);
// 获取记录集合的大小
int size = entries.size();
bytes32[] memoryuser_name_bytes_list = new bytes32[](uint256(size));
int[] memory item_id_list = newint[](uint256(size));
bytes32[] memoryitem_name_bytes_list = new bytes32[](uint256(size));
// 遍历记录集合
// 将记录的三个字段值分别存放到三个数组中，方便方法返回查询的数据
for(int i = 0; i< size; ++i) {
// 根据索引获取记录集合中的记录
```

(continues on next page)

(续上页)

```
Entry entry = entries.get(i);
// 根据记录的字段名查询字段值
// 注意字段值的类型不同需要选择相应的get方法
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");
}
```

更新记录核心代码如下：

```
TableFactory tf =TableFactory(0x1001);
Table table =tf.openTable("t_test");
Entry entry = table.newEntry();
entry.set("item_name",item_name);
Condition condition =table.newCondition();
// 设置过滤条件
condition.EQ("name",name);
condition.EQ("item_id",item_id);
// 调用Table的update方法更新记录
// 返回更新影响的记录数，值大于0则更新成功，否则更新失败
int count = table.update(name, entry,condition);
```

删除记录核心代码如下：

```
TableFactory tf =TableFactory(0x1001);
Table table =tf.openTable("t_test");
Condition condition =table.newCondition();
condition.EQ("name",name);
condition.EQ("item_id",item_id);
// 调用Table的remove方法删除记录
// 返回删除影响的记录数，值大于0则删除成功，否则删除失败
int count = table.remove(name,condition);
```

SDKCRUD Service接口

通过CRUD合约，我们可以看到只要合约方法涉及到数据读写操作，则首先打开对应的表，然后调用读写相关接口就可以进行区块链数据的读写。同时，我们注意到，CRUD合约的开发方式还是离不开编写合约、编译合约、部署合约，最后才能调用合约实现相关功能。那么有没有更方便，更简洁的方式呢？比如开发者合约都不用写，不用部署合约，就可以读写区块链上的数据。答案显而易见，我们已经实现了这样的极致易用需求。

FISCO BCOS SDK提供CRUD Service数据上链接口，这些接口实现的原理是调用区块链内置的一个预编译的CRUD合约，专门负责对用户表进行增删改查操作。Java SDKCRUD Service实现在（Python SDK和Nodejs SDK类似）org.fisco.bcos.web3j.precompile.crud.CRUDService类，其接口如下：

以上接口覆盖了表的创建、查看和增删改查操作。用户只需要调用SDK的接口就能完成相关操作，具体示例以下地址查看。

其中调用写接口会产生与调用CRUD合约接口等效的交易，需要共识节点共识一致后才会落盘存储。值得关注的是，利用CRUD Service接口，FISCO BCOS控制台针对每个接口实现了易用的sql语句命令，欢迎访问以下地址体验。

两种CRUD使用方式的比较

可能用户有这样的疑问，CRUDService接口如此简洁易用，那么是不是我们就只依靠这组接口完成区块链业务呢？其实也不尽然，相比于CRUD合约，其局限主要在两点：

- 数据访问范围受限：CRUDService接口做的事情非常专一，其本质是一组针对区块链用户表数据的读写接口。因此对于不是用户表数据，这组接口无能为力。例如Solidity合约的状态变量，并没

有存储在一个用户创建的表中，通过这组接口就查询不到状态变量的数据。然而CRUD合约本质是Solidity合约，只是多引入了一个Table.sol合约文件，使其有了CRUD功能。通过CRUD合约就可以操作Solidity状态数据和用户表数据。

- 业务处理能力受限：很多区块链业务不仅仅只是利用区块链上链数据，而是需要设计相关合约逻辑，满足相关条件和验证的前提下才可以向区块链提交数据，仅利用CRUD Service接口就难以胜任这样的区块链业务，因此CRUD合约必不可少。

有哪些Effective CRUD最佳实践?

CRUD功能虽然简单易用，但也需要清晰的知道如何高效的使用CRUD。

条款1：绝大多数情况下请选择CRUD合约开发区块链应用

区块链应用开发中，如果确定仅仅只是上链数据且数据只采用用户表存储，不依赖合约执行相关业务逻辑，那么考虑使用CRUD Service接口。否则，推荐使用CRUD合约。另外，为了方便开发和调试，可以随时利用CRUD Service的读接口（select和desc接口）或者利用控制台对应的相关命令进行数据查询。

条款2：Table.sol文件内容不可修改

Table.sol文件定义的是CRUD功能相关的抽象接口，每个接口区块链均有对应的具体实现。如果用户修改该文件，会导致交易执行失败或功能异常等问题。

条款3：CRUD用户表的主键不唯一

用户习惯于关系型数据库中的主键字段是唯一的，因此默认CRUD用户表中的主键也是唯一的，其实并不是。插入多条记录时，可以指定相同的主键。主键不唯一是由于CRUD表中存储的是主键到对应Entries的映射，进而基于主键进行增删改查。因此，调用CRUD接口(插入、更新、查询和删除记录)时，均需要传入主键值(不是主键名)。另外，主键值过长，会影响读写效率，建议主键值不要设置太长。

条款4：使用CRUD合约时，表字段过多，使用数组或struct封装字段参数

由于Solidity合约语言的限制，一个合约方法中的局部变量个数不得超过16个，否则会出现”Stacktoo deep, try removing local variables”编译错误。因此对于表字段比较多的情况，可以考虑将多个字段参数使用数组或struct封装，以减少局部变量个数，使其满足编译条件。

条款5：使用CRUD合约时，方法中如果涉及表操作，均需要先打开表

TableFactory抽象合约有openTable(string)方法，该方法返回打开的表对象。我们可能会将该表对象设置为合约的全局变量，方便下次操作表时直接使用。注意这是错误操作，因为每次打开表时，其获取的表对象注册在一个临时地址上，在下一个区块之后，该地址不再有效，因此不能将表对象设置为全局变量使用，而是当方法需要操作表之前，均需要打开表。另外，由于TableFactory在区块链中注册的是固定地址0x1001，因此可以将TableFactory对象设置为全局变量，不必每次均创建该对象。

条款6：使用CRUD合约，可以采用表和状态变量混合存储数据

CRUD合约是具备CRUD功能的Solidity合约，因此Solidity之前的状态变量存储方式依然可以使用。当存储一个变量值时，可能直接使用一个变量存储比较方便，因此可以既定义几个状态变量，同时也创建几个用户表进行数据存储。对于比较结构化的数据，推荐使用表的存储方式。

条款7：采用权限控制管理用户表

CRUD合约的逻辑与用户表数据分离，因此通过合约接口的限制已控制不住用户表的写操作（包括插入、更新和删除操作）。为了避免任何账户均可以写用户表，推荐使用权限控制来管理用户表（点击参考权限控制具体使用），指定特定账户才有权限写用户表。

22.4.5 网络协议

节点间通信：P2P协议

客户端与节点通信：JsonRPC，Channel协议

消息订阅服务：AMOP协议

FISCO BCOS网络端口讲解

作者：张开翔 | FISCO BCOS 首席架构师

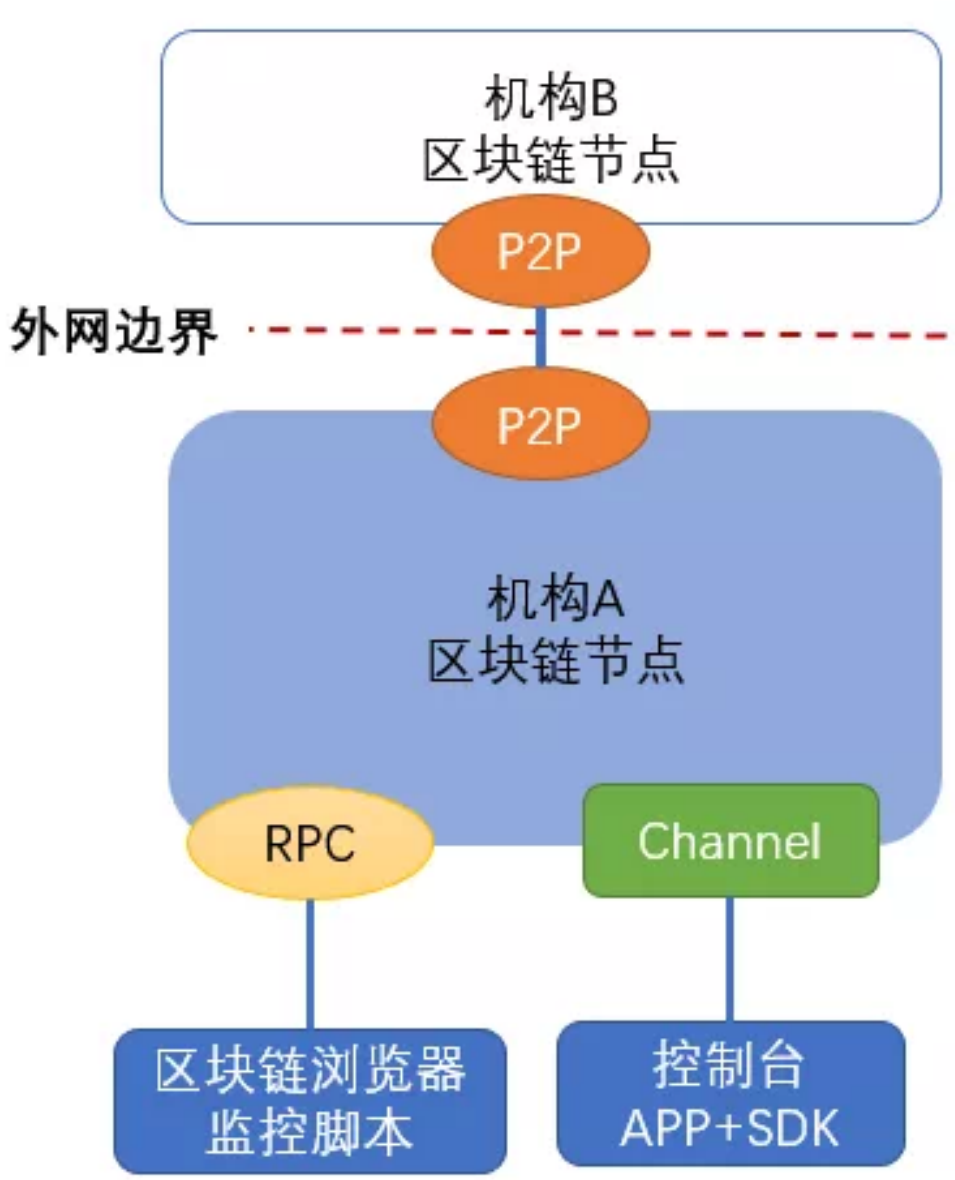
作者语

区块链网络由多个互相连接的节点构成，每个节点又与客户端浏览器监控工具等相连；理清各种网络端口的存在，达成网络畅通的同时又保证安全是建立区块链网络的基础。 同时，在搭链的过程中有一些热点问题，比如为什么节点开了这么多端口？或者为什么网络不通？节点无法连接？没有共识出块？正所谓"通则不痛"，**网络畅通才能链接一切**。

本期文章讲的就是网络端口互通这点事，作者从FISCO BCOS的网络端口、FISCO BCOS 2.0的典型网络配置、设计网络安全组的一些策略等角度进行了解析。

FISCO BCOS 2.0网络的三类端口

FISCO BCOS 2.0的网络包括P2P端口、RPC端口、Channel端口。



1.P2P端口

P2P端口，用于区块链节点之间的互联，包括机构内的多个节点，以及多机构间节点和节点的互联。如果其他节点在机构外，那么这个连接要监听公网地址，或者监听内网，且由连接公网的网关（如nginx）转发网络连接。 节点之间的连接会由联盟链的准入机制控制，节点之间连接依赖节点证书验证，以排除未经许可的危险连接。这个链路上的数据通过SSL方式加密，采用高强度密钥，可以有效的保护通信安全。

P2P网络详细设计

网络安全和准入控制

2.Channel端口

Channel端口，控制台和客户端SDK连接Channel端口，互相之间要通过证书认证，只有经过认证的客户端才能向节点发起请求，通信数据也是采用SSL方式加密。Channel端口使用了TCP的长连接，用心跳包检测和保持存活，通信效率较高，支持AMOP协议的点对点通信，功能相当灵活强大。 Channel端口应只监听内网IP地址，供机构内其他的应用服务器通过SDK连接，不应监听外网地址或接受公网的连接，以免发生不必要的安全的问题，也不要只监听本地地址（127.0.0.1或localhost），否则其他应用服务器将无法连接到节点上。

SDK文档

AMOP协议

3. RPC端口

RPC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。

RPC端口接受JSON-RPC格式的请求，格式比较直观清晰，采用CURL、JavaScript、Python、Go等语言都可以组装JSON格式的请求，发送到节点来处理。当然发送交易时，要在客户端实现交易签名。要注意的是，RPC连接没有做证书验证，且网络传输默认是明文的，安全性相对不高，建议只监听内网端口，用于监控、运营管理，状态查询等内部的工作流程上。目前监控脚本，区块链浏览器连接的是RPC端口。

RPC端口文档

一个FISCO BCOS 2.0的典型网络配置

FISCO BCOS 2.0一个典型网络配置如下所示，可以看到RPC和Channel端口共用一个IP，P2P连接单独监听一个IP，即一个区块链节点使用2个IP和3个端口。

节点下的config.ini文件：

```
[rpc]listen_ip=127.0.0.1channel_listen_port=20200jsonrpc_listen_
↪port=8545[p2p]listen_ip=0.0.0.0listen_port=30300
```

几种计算机的典型网络地址

1、特殊地址：0.0.0.0，表示监听本机所有的地址，包括本地、内网、公网（如有）地址，也就是全面放开，来者不拒。除非为了方便且确信安全，一般不应监听这个地址。

2、本机地址：127.0.0.1（有的配置可以写成localhost），只有同在本机上的其他进程才能连接到这个地址，其他机器一律连不过来。FISCO BCOS有的示例脚本为了安全和简易起见，默认写的是这个地址，包括build_chain脚本默认配置等。用户有时会发现其他机器运行客户端程序连不过来，大概率是这个原因，或者也可以检查下网络策略是否开通了互联，建议可以用系统的 telnet 【ip】【port】命令来先快速检测下是否能联通。

3、内网地址：通常192.168.xxx.xxx, 172.xxx.xxx.xxx，10.xxx.xxx.xxx开头的地址是内网地址，如监听这个地址，则只有同一局域网的其他机器可以访问它。

4、外网地址：暴露在互联网上的公网地址，或者可以从机构外部网络访问的地址，总之是外部服务器能连接的就是外部地址。

设计网络安全组的一些策略

在不同的网络拓扑上，可能牵涉这样的情况：服务器虽然可以访问外网，但是是由网关、路由器、NAT转发的，这时就需要了解具体的网络结构，进行配置了。如监听一个内网地址，把这个内

网地址和监听的端口配置到转发器上，同样也可以接收来自外网的连接。

在网络安全方面，需要仔细的设计网络安全组策略，IP和端口黑白名单，精确的进行双向的连接控制。包括不限于以下策略：

- **1、设置外部IP白名单**，只有这些外部IP（一般是建立了联盟的其他机构）能连接过来；
- **2、设置IP黑名单**，拒绝某些特定IP的连接，而不用等它连接到节点才进行准入控制判断；
- **3、控制RPC端口**，（如8545端口）只对本机开放，其他内网外网服务器都连不到这个端口；
- **4、控制Channel端口**，只对某一个内网网段或某几个IP开放，把自己的应用部署到开放的网段或IP对应的服务器上，内网其他应用不能访问区块链节点；
- **5、但凡有外网端口**，建议设置防DDoS的措施，避免频繁连接、海量并发连接攻击。

总结

三种网络端口的IP地址、端口、作用、安全考量

名称	IP地址	端口	作用	安全考量
P2P端口	监听外网IP或可从外网转发请求的内网IP	默认30300，同机部署更多节点依次递增（下同）	节点之间的互联	准入控制，加密通信，IP黑白名单，防DDoS
RPC端口	内网IP	默认8545	本地管理监控	切勿对外网公开此端口
Channel端口	内网IP（同RPC端口）	默认20200	SDK、控制台连接	通过证书和节点互相校验，加密通信，不建议对公网提供此端口

群友问答

Q：节点间采用P2P通讯，同时采用SSL认证，每个节点如何获取和验证链接节点的公钥证书、预置根证书和证书链？节点跟其他节点通讯时，哪方做Server，哪方做Client？

A：在创建链的时候，就分配了链的根证书；每个节点跟其他节点通讯时，互为Server Client。

Q：外网IP建议布防DDoS攻击，这是一种标准的中心化的防御逻辑。区块链的实现逻辑是去中心化，攻击单个节点是否变得没有意义？机构能不能布置更多的节点？

A：关于DDoS的问题，对联盟链来说，一个机构一般部署两个节点，要是被攻击可能是会影响该机构的业务，但不至于影响全网。机构可以布置多个节点，比如4个、5个或者10个。

感谢参与本次话题讨论的小伙伴们！开源社区，因你们更美！

欢迎更多朋友加入FISCO BCOS官方技术交流群，参与更多话题交流。（进群请扫描下方二维码添加小助手【FISCOBCOS010】）

FISCO BCOS的网络压缩功能如何？

作者：陈宇杰 | FISCO BCOS 核心开发者

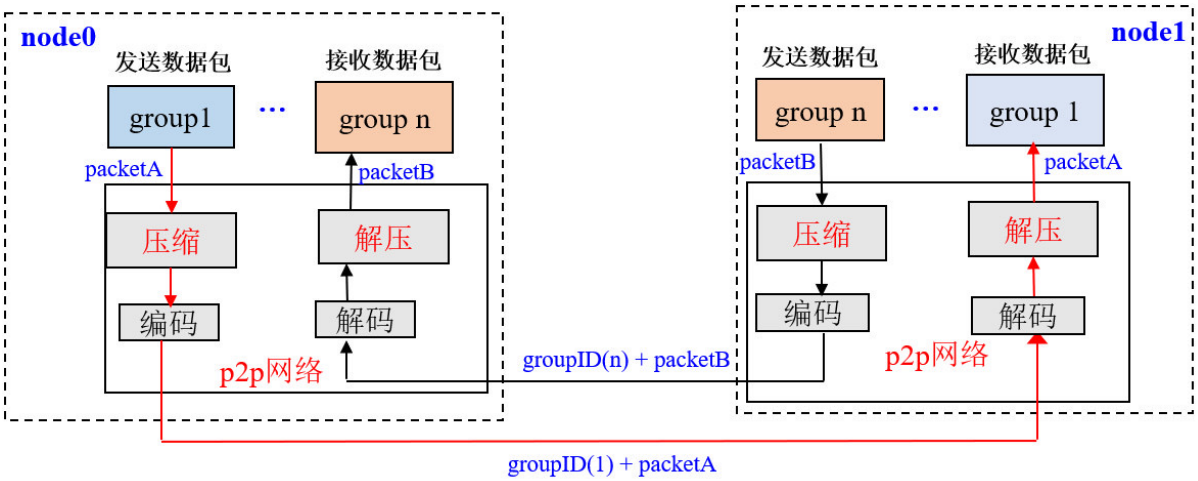
作者语

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。

本期文章讲的就是FISCO BCOS的网络压缩功能，作者从FISCO BCOS的系统框架、核心实现、处理流程、测试结果等角度进行了解析。

Part 1. 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：

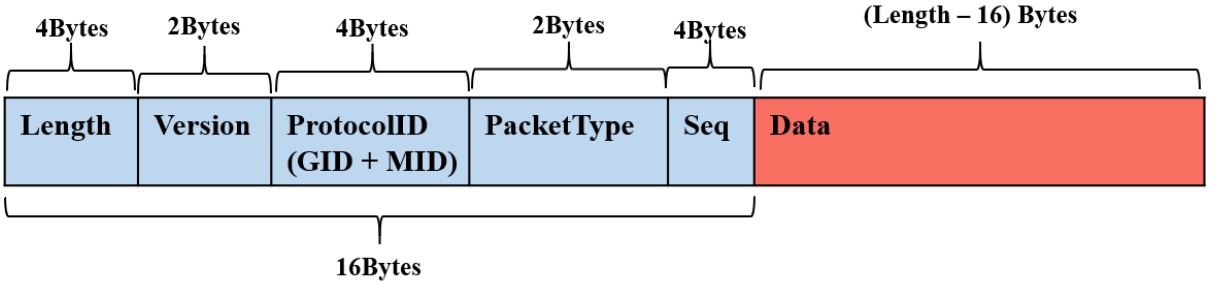
- **发送端压缩数据包**：群组层通过P2P层发送数据时，若数据包大小超过1KB，则压缩数据包后，将其发送到目标节点；
- **接收端解压数据包**：节点收到数据包后，首先判断收到的数据包是否被压缩，若数据包是压缩后的数据包，则将其解压后传递给指定群组，否则直接将数据传递给对应群组。

Part 2. 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。

数据压缩标记位

FISCO BCOS的网络数据包结构如下图：



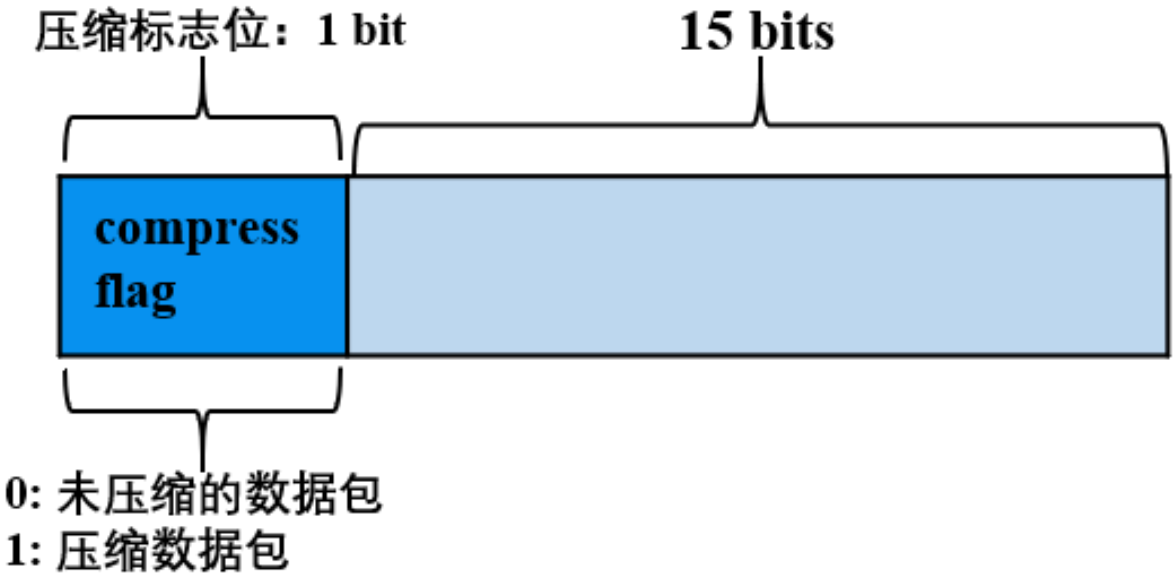
网络数据包主要包括包头和数据两部分，包头占了16个字节，各个字段含义如下：

- 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。

Part 3. 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。

Part 4. 配置与兼容

配置说明

- 开启压缩：2.0.0-rc2及其以上版本支持网络压缩功能，配置`config.ini`的`[p2p].enable_compresss=true`
- 关闭压缩：`config.ini`的`[p2p].enable_compresss=false`

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，目前仅**release-2.0.0-rc2及其以上版本**有网络压缩功能

Part 5. 测试结果

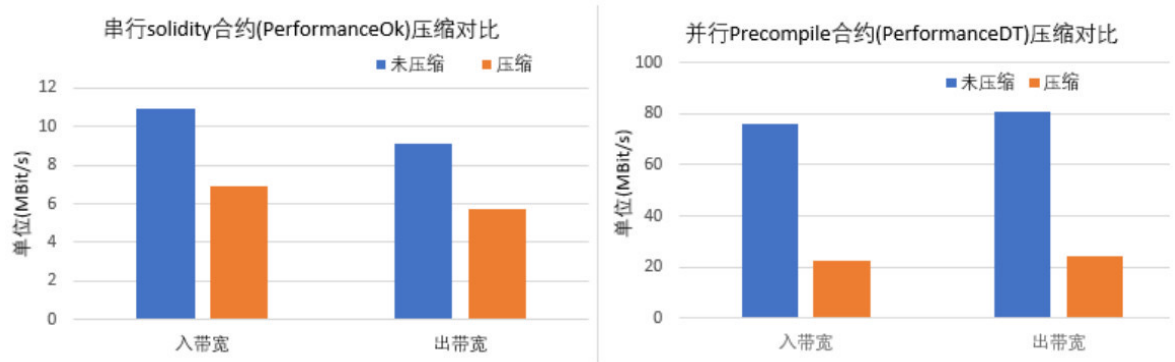
为测试网络压缩效果，分别在**内网和外网**环境下，以**同样的压测程序和QPS**压测**开启网络压缩和没开启网络压缩**的四节点区块链，测试结果如下。

通过测试结果可看出：

- **内网环境下**：**开启压缩对区块链系统性能影响不大**，运行**串行solidity压测合约**时，网络带宽消耗降低为未开压缩时的**三分之二**；运行**并行precompile压测合约**，网络带宽消耗降低到未开压缩时的**三分之一**；
- **外网环境下**：开启压缩可提升区块链系统性能

图一：带宽对比

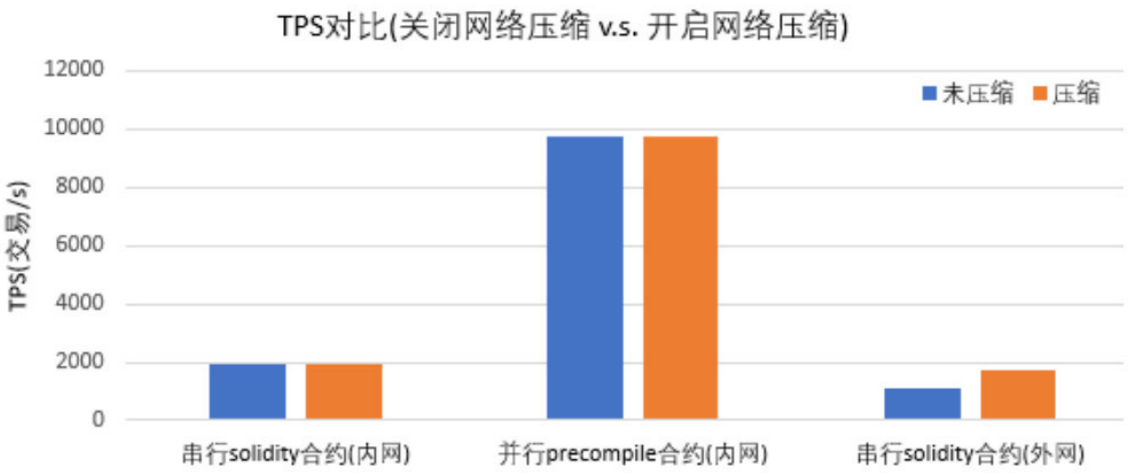
（关闭压缩和开启压缩情况下，压测并行solidity合约和串行Precompile合约）



通过图一可看出，执行**串行solidity合约**，开启压缩可**节省三分之一带宽**；执行**并行Precompile合约可节省三分 之二带宽**。

图二：TPS对比

(内网和外网环境下，关闭压缩和开启压缩情况下TPS)



通过图二可看出，**内网环境下，开启压缩对区块链系统性能影响不大**；外网环境下，因为在有限带宽限制下，开启 压缩可处理更多交易，区块链性能提升了约三分之一。

图三：详细数据

内网环境测试结果

外网环境测试结果

群友问答

@名：测试带宽是用什么软件测的呢?

@陈宇杰：当时测试带宽的时候，是独占的机器，直接用的nload，当然，在多进程环境下，还可以用nethogs等。

@elikong：提两个问题：

1、为啥选snappy，有没有做过压缩性能分析对比，包括压缩率，cpu时间，典型消息等方面。

2、内网情况压缩前后带宽变化很大，但tps提升不明显的原因是什么?

@陈宇杰

1、是有前期调研的，当时调研了各种压缩库的压缩比、压缩和解压速度、license等。初选的是lz4和snappy，并实现了同时支持两种库压缩算法的版本，并且进行了压测，压测结果显示两种库测试结果相差不大。由于我们的系统里已经集成了snappy，为了避免引入额外的库，所以最终选用了snappy 。

2、内网情况下，性能瓶颈是CPU(包括交易执行速度、验签性能等)、IO等，网络不是瓶颈，因此即使开启了压缩，节省了网络资源，对性能影响也不大。当然这也表明压缩、解压本身对性能损耗不大；外网环境下，网络是瓶颈，这个时候大部分时间是耗费在网络上的，开启压缩，节省了很多网络带宽，使得在相同时间内，节点间可传输更多数据包，因而性能有提升。

FISCO BCOS的AMOP使用介绍

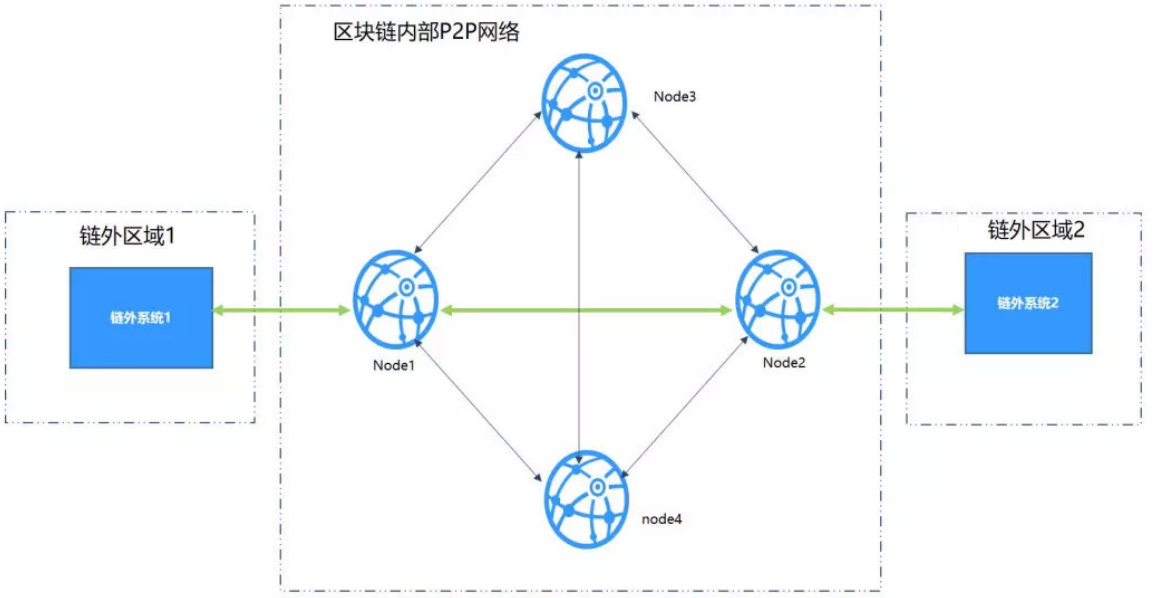
作者：尹强文 | FISCO BCOS 核心开发者

AMOP简介：链上信使协议AMOP（Advanced Messages Onchain Protocol）旨在为联盟链各个机构提供一个安全高效的消息传输信道，支持跨机构之间，点对点地实时消息通信，为链外系统之间的交互提供

标准化接口，AMOP基于SSL通信加密，确保消息无法被窃听，消息收发均有异常重传、超时检测和路径规划机制，确保消息传输的可靠性。

逻辑架构

AMOP使用了FISCO BCOS的P2P底层通信，逻辑架构如下：



各区域概述如下：

- **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。
- **区块链内部P2P网络区域**：此区域为逻辑区域，部署各机构的区块链节点，区块链节点也可部署在机构内部。

核心实现

AMOP的消息收发基于**Sub-Pub订阅机制**，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。

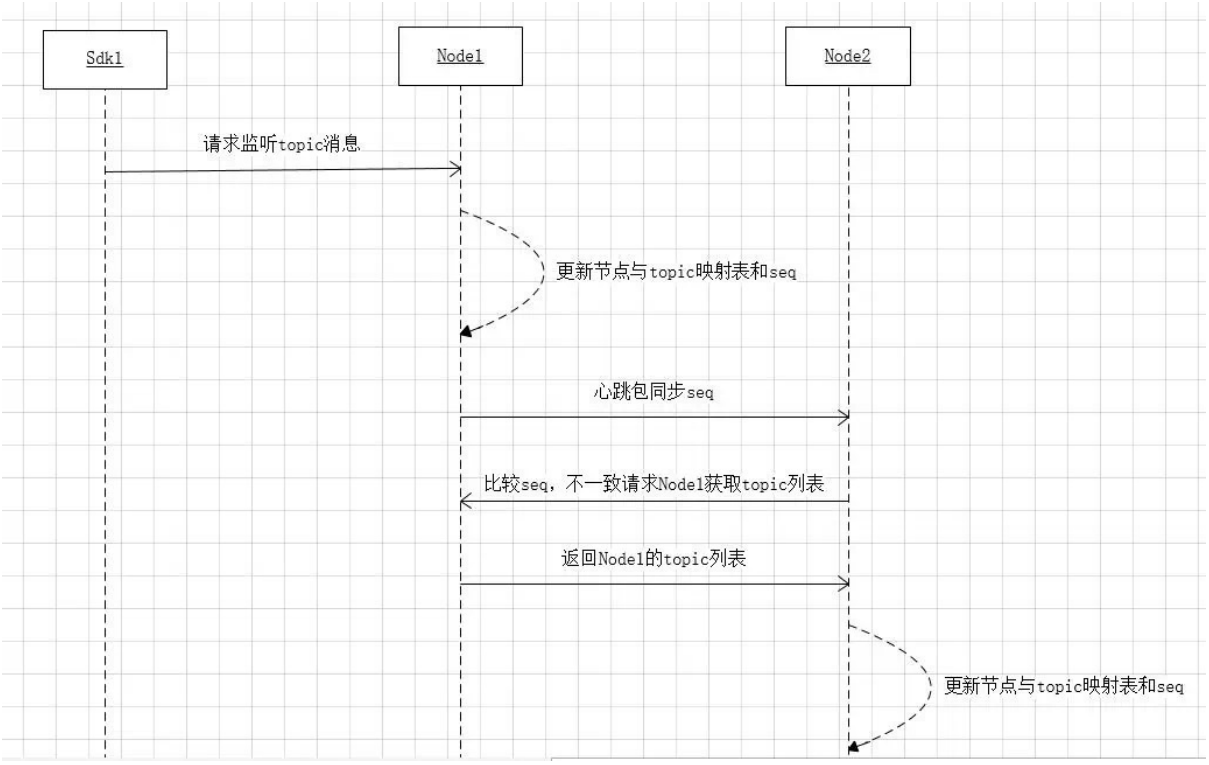
AMOP支持同一个区块链网络中有多个Topic收发消息，支持任意数量的服务端和客户端，当有多个服务端关注同一个Topic时，该Topic的消息将随机下发到其中一个可用的服务端。

AMOP包括两个流程：

1. 服务端设置Topic，用以监听该Topic的消息；
2. 客户端往该Topic发送消息。

下面以一个例子来说明内部实现，如下图所示，有2个SDK分别为SDK1、SDK2，2个节点分别是Node1、Node2。SDK1连接Node1设置Topic T1，SDK2连接Node2发送Topic为T1的消息。

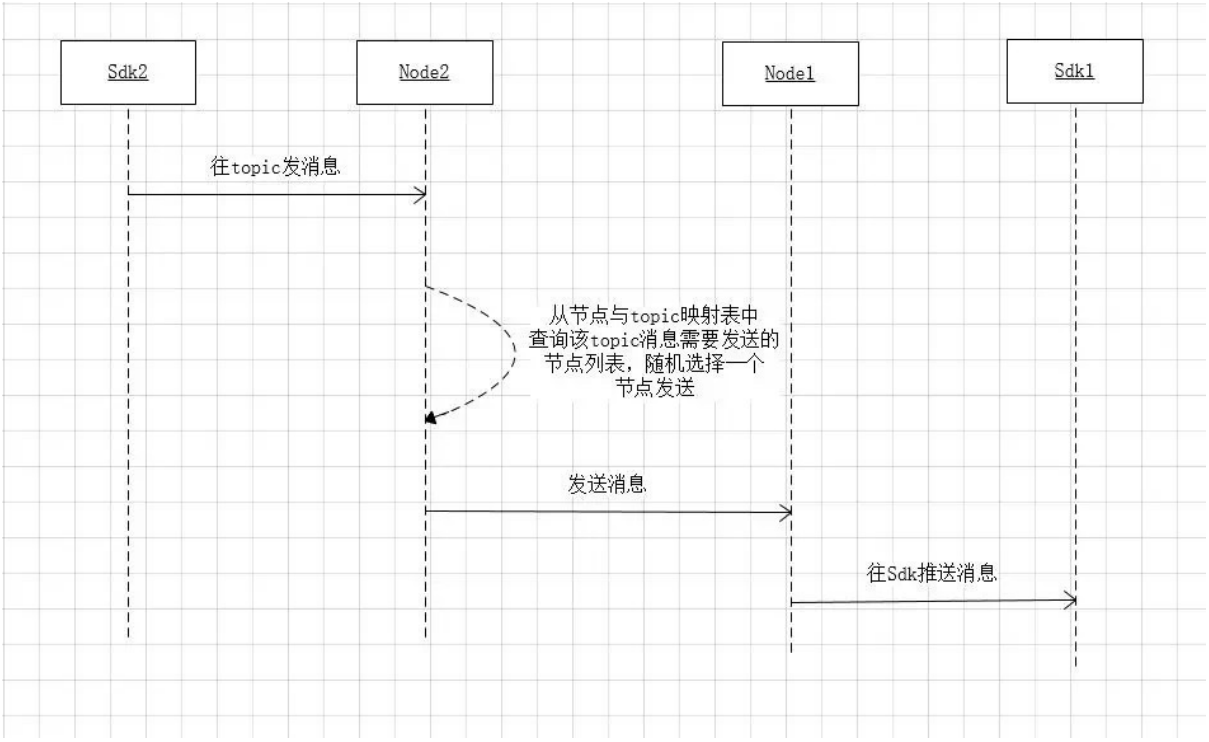
1. 服务端设置Topic，用以监听该Topic的消息的时序



主要流程如下：

1. SDK1发送监听某个Topic消息的请求，到其直连的节点Node1，Node1里维护了节点与Topic列表的映射关系，该映射关系用于消息路由，是一个map结构，key是NodeId，value是一个set，set存储的是该NodeId可以接收消息的Topic列表。
2. Node1新增一个Topic之后，节点会更新节点与Topic映射表。
3. Node1更新seq：seq主要用于保证各节点映射表一致，新增一个Topic之后，本节点的seq会加1，节点之间的心跳包会将这个值带上发送到其他节点，其他节点（Node2）接收到心跳包之后对比参数里的seq与本节点的seq，如果不一致会向源节点(Node1)请求节点与Topic列表的映射关系，将最新的映射关系更新到本节点并更新seq。这样就保证了全局映射关系的一致。

2. 客户端往该Topic发送消息时序



- SDK2向Node2上发送消息。
- Node2从节点与Topic列表的映射关系中查到该Topic可以发往的节点列表，从中随机选择一个节点Node1发送。
- Node1节点收到消息之后推送给SDK1。

配置文件配置

AMOP无需任何额外配置，以下为Web3Sdk的参考配置，详细请参考文档。

```
<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>127.0.0.1:20000</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="agencyName" value="fisco" />
        <property name="allChannelConnections" ref="groupChannelConnectionsConfig"></property>
</bean>
```

不同SDK的配置文件差异在于连接的节点地址不同，以文章开头处的逻辑架构图为例，假定Node1的监听地址是127.0.0.1:20200，Node2的监听地址是127.0.0.1:20201，那么SDK1配置为127.0.0.1:20200，SDK2配置127.0.0.1:20201。

测试

按上述说明配置好后，用户指定一个Topic，执行以下两个命令可以进行测试。

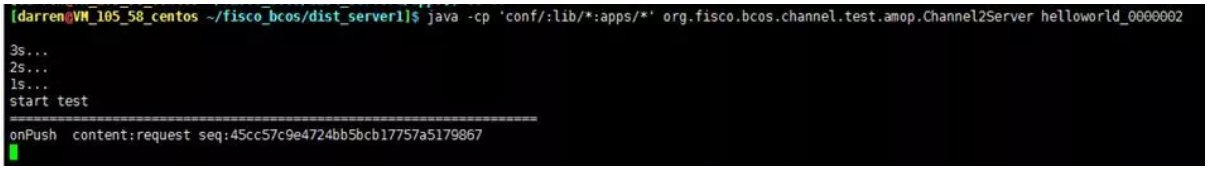
启动amop服务端

```
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]
```

客户端和服务器执行后得如下效果：



常见错误码及问题定位

- **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。

未来计划

未来将继续丰富AMOP功能，支持二进制传输、消息多播协议以及Topic认证机制等，也欢迎大家使用AMOP，并提出优化建议。

22.4.6 智能合约

合约引擎：支持Solidity和预编译合约

引擎特点：图灵完备，沙盒运行

版本控制：基于CNS支持多版本合约

灰度升级：支持多版本合约共存、灰度升级

生命周期管理：支持合约和账户的冻结、解冻

FISCO BCOS智能合约开发快速入门

作者：张龙 | FISCO BCOS 核心开发者

目前，FISCO BCOS平台支持Solidity和Precompiled两种类型的智能合约，同时，提供交互式控制台工具（Console）,方便开发者与链进行交互，部署、调用智能合约。 为了让大家快速上手智能合约，FISCO BCOS推出了智能合约系列教程，本文将带大家快速入门，运用FISCO BCOS开发部署一个简单的智能合约。

智能合约简介

众所周知，智能合约的出现，使区块链不仅能处理简单的转账功能，还能实现复杂的业务逻辑，极大地推动了区块链技术发展，加速应用落地。

目前，在众多区块链平台中，大多数集成了以太坊虚拟机，并使用Solidity作为智能合约开发语言。作为一门面向合约的高级编程语言，Solidity借鉴了C++、Python和JavaScript等语言的设计，使用静态类型，不仅支持基础/复杂数据类型操作、逻辑操作，同时提供高级语言的相关特性，比如继承、重载、库和用户自定义类型等。

作为最大最活跃的国产开源联盟链社区，FISCO BCOS无缝支持Solidity合约，并提供从开发、编译、部署到调用的全链路工具和完整解决方案，使智能合约和区块链应用开发变得简单。 除此之外，基于大量探索和实践，FISCO BCOS不仅支持Solidity合约，还支持Precompiled合约，并在用户层提供CRUD合约接口。面向库表开发的CRUD合约不仅更符合用户开发习惯，进一步降低合约开发难度，提升性能，使区块链应用满足高并发场景的诉求。

智能合约分类

FISCO BCOS平台支持两种类型的智能合约：Solidity合约和Precompiled合约。

Solidity合约

Solidity合约运行在EVM上，EVM为以太坊虚拟机，采用哈佛架构，指令、数据和栈完全分离。 在智能合约运行期间，首先创建一个沙盒环境（EVM实例），沙盒环境与外部环境完全隔离，无法访问网络、文件系统其它进程，智能合约在EVM内只允许进行有限的操作。交易执行时，EVM通过获取合约的opcode，将opcode转化为对应的EVM指令，并按照指令进行执行。

从应用落地的数量来看，Solidity合约使用最为广泛，几乎所有区块链平台都支持，但Solidity也有很多缺点。如下：

- 合约在EVM中串行执行，性能较差；
- 跨合约调用会新建EVM，内存开销较大；
- 合约变量和数据存在MPT数中，不便于合约升级；
- 逻辑和数据耦合，不便于存储扩容。

Precompiled合约

Precompiled合约即预编译合约。预编译合约通过Precompiled引擎执行，采用C++编写合约逻辑，合约编译集成进FISCO BCOS底层节点。 调用合约不进EVM，可并行执行，突破EVM性能瓶颈；提供标准开发框架，只需继承基类，实现call接口即可；适合于逻辑相对确定、追求高并发的场景；数据存在表中，与合约分离，可升级合约逻辑。

当然，预编译合约的使用有一定的门槛。如下：

- 对于数据的存储，需要创建FISCO BCOS特有的表结构；
- 编写合约时需继承Precompiled类，然后实现Call接口函数；
- 完成合约开发后，需在底层为预编译合约注册地址；
- 编写完成合约后，需要重新编译FISCO BCOS源码。

为了屏蔽预编译合约在开发和使用中的门槛，FISCO BCOS基于预编译合约和分布式存储设计了CRUD合约接口。用户在编写Solidity合约时，只需要引入抽象合约接口文件Table.sol，便可使用CRUD功能，用户不需要关心底层的具体实现。

智能合约开发

本节将基于全球英文认证考试成绩管理作为场景，基于FISCO BCOS平台对智能合约进行开发。全球认证考试包括GRE、TOEFL、IELTS等。为了简化合约逻辑，所有成绩统一由考试管理中心发布和管理，学生可以根据自己的账号（地址）查询自己的考试成绩。

Solidity合约开发

在 Solidity 中，合约类似于面向对象编程语言中的类。Solidity合约有自身的代码结构，由几个部分组成，如下所示。

- 状态变量：状态变量是永久存储在合约中的值
- 构造函数：用于部署并初始化合约
- 事件：事件是能方便地调用以太坊虚拟机日志功能的接口
- 修饰器：函数修饰器可以用来改变函数的行为，比如自动检查，类似Spring的AOP
- 函数：函数是合约中代码的可执行单元

创建合约

首先创建一个名为StudentScoreBySol的合约，用于管理学生的成绩。如下代码所示，开头需要引入合约版本。

```
pragma solidity ^0.4.25;

contract StudentScoreBySol {}
```

定义状态变量

状态变量用于存储和管理学生的考试成绩。在当前场景中定义两个变量，其中_owner为合约的创建者，即考试管理中心；_scores用于存储学生成绩，为一个嵌套mapping类型，第一个mapping的key(address)为学生的账户（私钥对应的地址），value也为一个mapping类型，对应为每一科的成绩，在第二个mapping中key（bytes32）为科目名称，如GRE、TOEFL等，value(uint8)为成绩。如下所示。

```
//状态变量:用于存储合约创建者和学生成绩
address private _owner;
mapping(address => mapping(bytes32 => uint8)) private _scores;
```

定义事件

定义一个事件setScoreEvent，用于跟踪对学生成绩的新增/修改操作，可以在业务层面对事件进行监听。事件的定义是可选的，如果没有定义也没关系，在业务层面可以根据方法的返回值去判断交易是否成功，但无法做到更精细的问题定位。

```
//事件：用户跟踪监听对学生成绩的修改
event setScoreEvent(address studentId, bytes32 courseName, uint8 score);
```

定义修饰器

智能合约中的修饰器（Modifier）类似面向对象编程中的AOP，满足条件才会执行后续操作。如下所示，修饰器要求必须是合约的Owner才能进行后续操作，其中Owner为考试管理中心。

```
//修饰器：用于对函数的操作权限验证
modifier onlyOwner {
    require(_owner == msg.sender, "Auth: only owner is authorized.");
    _;
}
```

定义构造方法

构造方法用于实例化合约，在当前构造方法中，指定Owner为合约的部署者。

```
//构造方法：用于初始化合约
constructor () public {
    _owner = msg.sender;
}
```

编写函数

当前合约中，定义两个函数，setScore函数用于新增/修改/删除（score置为0）学生成绩，并使用了onlyOwner修饰器，只有Owner(考试管理中心）才能操作，并通过setScoreEvent事件跟踪每次操作内容。getScore方法用于成绩查询，其中view修饰符表示不会修改状态变量。

```
//函数：用于新增/修改学生成绩
function setScore(address studentId, bytes32 courseName, uint8 score) public onlyOwner returns(bool){
    _scores[studentId][courseName] = score;
    emit setScoreEvent(studentId, courseName, score);
    return true;
}
//函数：用于学生查询成绩
function getScore(bytes32 courseName) public view returns(uint8){
    return _scores[msg.sender][courseName];
}
```

Solidity合约完整代码如下所示。基于Solidity语言的合约开发，看似简单，但需要对Solidity编程语言深入学习，才能编写出高可用的合约，具备一定学习成本。 通过FISCO BCOS开源社区推出的智能合约专题，开发者可了解更多运用Solidity编写智能合约的方法与技巧，关注“FISCO BCOS开源社区”公众号可获取专题。 更多细节可参考Solidity官方文档：https://solidity-cn.readthedocs.io/zh/develop/

```
pragma solidity ^0.4.25;

contract StudentScoreBySol {

    //状态变量:用于存储合约创建者和学生成绩
    address private _owner;
    mapping(address => mapping(bytes32 => uint8)) private _scores;

    //事件: 用于跟踪监听对学生成绩的修改
    event setScoreEvent(address studentId, bytes32 courseName, uint8 score);

    //修饰器: 用于对函数的操作权限验证
    modifier onlyOwner {
        require(_owner == msg.sender, "Auth: only owner is authorized.");
        _;
    }
    //构造方法: 用于初始化合约
    constructor () public {
        _owner = msg.sender;
    }
    //函数: 用于新增/修改学生成绩
    function setScore(address studentId, bytes32 courseName, uint8 score) public onlyOwner returns(bool){
        _scores[studentId][courseName] = score;
        emit setScoreEvent(studentId, courseName, score);
        return true;
    }
    //函数: 用于学生查询成绩
    function getScore(bytes32 courseName) public view returns(uint8){
        return _scores[msg.sender][courseName];
    }
}
```

CRUD合约开发

CRUD合约是CRUD功能的核心，用户可以直接在合约中引用CRUD接口文件Table.sol，并在Solidity合约中调用CRUD接口。CRUD合约的开发完全遵从数据库的操作习惯，更容易理解和上手。

CRUD合约更多开发细节可参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/smart_contract.htm

创建合约

CRUD创建的合约和Solidity创建的没有太大区别，唯一区别在于需要引入CRUD接口文件Table.sol，如下所示。修饰器和构造方法的作用与Solidity合约相同。

```
pragma solidity ^0.4.25;

import "./Table.sol";

contract StudentScoreByCRUD {

    address private _owner;

    modifier onlyOwner {
        require(_owner == msg.sender, "Auth: only owner is authorized.");
        _;
    }

    constructor () public {
        _owner = msg.sender;
    }
}
```

事件定义

在Solidity合约中可以通过setScore完成成绩的新增/修改/删除，但在CRUD合约中，需要借助CRUD接口文件的不同接口，通过不同函数实现，所以需要针对不同的函数定义不同时间，如下所示。

```
event createEvent(address owner, string tableName);
event insertEvent(address studentId, string courseName, int score);
event updateEvent(address studentId, string courseName, int score);
event removeEvent(address studentId, string courseName);
```

- createEvent：用于跟踪创建表操作；
- insertEvent：用于跟踪插入成绩操作；
- updateEvent：用于跟踪更新成绩操作；
- removeEvent：用于跟踪删除成绩操作。

创建表函数

CRUD合约实现业务功能，首先需要像数据库操作一样，创建一张表，用于存放数据。 FISCO BCOS底层提供了TableFactory合约，该合约的地址固定为0x1001，可以通过TableFactory对象提供的方法对表进行创建(createTable)和打开(openTable)，如下所示。 createTable接口返回值为0时，说明创建成功。需要注意的是，为了让创建的表可被多个合约共享访问，表名必须是群组内全局可见且唯一的，无法在同一条链上的同一个群组中，创建多个名称相同的表。

```
//创建成绩表
function create() public onlyOwner returns(int){
    TableFactory tf = TableFactory(0x1001);
    int count = tf.createTable("stu_score", "student_id", "course_name, score");
    emit createEvent(msg.sender, "stu_score");
    return count;
}
```

新增成绩函数

对表进行操作时，首先需要通过TableFactory打开对应的表，获得表对象，然后通过表对象的insert方法进行数据插入。在插入数据前，首先需要构建一个Entry对象实例，代码如下所示。

```
// 插入成绩操作
function insert(address studentId, string courseName, int score) public onlyOwner returns(int) {
    TableFactory tf = TableFactory(0x1001);
    Table table = tf.openTable("stu_score");

    string memory stuIdStr = addressToString(studentId);
    Entry entry = table.newEntry();
    entry.set("student_id", stuIdStr);
    entry.set("course_name", courseName);
    entry.set("score", score);

    int count = table.insert(stuIdStr, entry);
    emit insertEvent(studentId, courseName, score);
    return count;
}
```

需要注意的是，Table接口合约的insert、remove、update和select函数返回值类似数据库，均为受影响的记录行数，且接口中key的类型为string。 而在当前场景中，学生的studentId为address类型，所以需要在函数内部将address类型转化为string类型，代码如下。

```
function addressToString(address addr) private pure returns(string){
    //Convert addr to bytes
    bytes20 value = bytes20(uint160(addr));
    bytes memory strBytes = new bytes(42);
    //Encode hex prefix
    strBytes[0] = '0';
    strBytes[1] = 'x';
    //Encode bytes usig hex encoding
    for(uint i=0;i<20;i++){
        uint8 byteValue = uint8(value[i]);
        strBytes[2 + (i<<1)] = encode(byteValue >> 4);
        strBytes[3 + (i<<1)] = encode(byteValue & 0x0f);
    }
    return string(strBytes);
}

function encode(uint8 num) private pure returns(byte){
    //0-9 -> 0-9
    if(num >= 0 && num <= 9){
        return byte(num + 48);
    }
    //10-15 -> a-f
    return byte(num + 87);
}
```

更新成绩函数

更新成绩操作步骤包括：通过TableFactory对象打开表，然后，像数据库一样构造筛选条件。 在CRUD合约接口中，提供了Condition对象，该对象提供了诸如大于、等于、小于等一系列条件方法。构造完成条件对象后，可以调用table对象的udpdate接口，完成更新操作，代码如下。

```
//更新成绩操作
function update(address studentId, string courseName, int newScore) public onlyOwner returns(int) {
    TableFactory tf = TableFactory(0x1001);
    Table table = tf.openTable("stu_score");

    Entry entry = table.newEntry();
    entry.set("score", newScore);

    string memory stuIdStr = addressToString(studentId);
    Condition condition = table.newCondition();
    condition.EQ("student_id", stuIdStr);
    condition.EQ("course_name", courseName);

    int count = table.update(stuIdStr, entry, condition);
    emit updateEvent(studentId, courseName, newScore);
    return count;
}
```

删除成绩操作

删除操作和更新操作类似，需要调用table.remove接口完成，不再赘述，代码如下。

```
//删除成绩操作
function remove(address studentId, string courseName) public onlyOwner returns(int){
    TableFactory tf = TableFactory(0x1001);
    Table table = tf.openTable("stu_score");

    string memory stuIdStr = addressToString(studentId);
    Condition condition = table.newCondition();
    condition.EQ("student_id", stuIdStr);
    condition.EQ("course_name", courseName);

    int count = table.remove(stuIdStr, condition);
    emit removeEvent(studentId, courseName);
    return count;
}
```

查询成绩操作

查询成绩操作很简单，需要调用table.select接口完成，代码如下。

```
// 查询成绩操作
function select(string courseName) public view returns(int){
    TableFactory tf = TableFactory(0x1001);
    Table table = tf.openTable("stu_score");

    string memory stuIdStr = addressToString(msg.sender);
    Condition condition = table.newCondition();
    condition.EQ("student_id", stuIdStr);
    condition.EQ("course_name", courseName);

    Entries entries = table.select(stuIdStr, condition);

    if(entries.size() == 0) return 0;
    return entries.get(0).getInt("score");
}
```

至此基于CRUD的合约开发完成。

从当前场景的代码行数来看，CRUD合约比较复杂，Solidity合约相对简单。但这只是一个幻觉，实际情况可能并非如此。而且CRUD合约的开发更符合开发者习惯，没有多余的学习成本，更容易理解和上手。

合约部署及调用

开发完智能合约后，需要对合约进行编译和部署，然后才能调用。FISCO BCOS平台提供了交互式控制台工具（Console）,可以非常方便地与链进行交互。下面将以上述智能合约为例，采用控制台工具进行部署和调用。 控制台安装及使用可参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/console/console.html

准备工作

在对合约进行部署和调用前需要做三件事情。首先复制编写好的合约到控制台目录console/contracts/solidity下。如下图所示。

```
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console/contracts/solidity$ ls
StudentScoreByCRUD.sol  StudentScoreBySol.sol  Table.sol
```

其次对合约进行编译，可以采用控制台目录下的sol2java.sh脚本对合约进行编译，编译结束后会在console/contracts/sdk目录下生成如下文件夹，如下图所示。

```
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console$ ls
accounts  apps  conf  contracts  deploylog.txt  get_account.sh  get_gm_account.sh  lib  log  replace_solc_jar.sh  sol2java.sh  start.sh
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console$ bash sol2java.sh com

Compile solidity contract files to java contract files successfully!
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console$ ls contracts/sdk/
abi  bin  java
```

其中abi中存放合约的ABI，bin存放合约的二级制编码文件(BINARY)，JAVA文件夹中为对应的JAVA合约，便于通过SDK和链进行交互。 需要注意的是对CRUD合约编译时。需要将CRUD接口合约Table.sol一并放入console/contracts/solidity目录下，不然会报错。 最后，对合约进行部署时，依赖外部账户，所以首先需要生成账户。在控制台中提供了账户生成工具get_account.sh，运行该脚本会在console/accounts目录下生成账户。

我们利用账户生成工具生成两个账户。一个为考试管理中心的账户，用于部署和新增/修改/删除学生成绩；一个为学生账户，用于查看考试成绩。如下所示。

```
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console/accounts$ ls
0x216176e40ded68df4345a75274318f7001030ae3.pem         0x76e47590efa1875f22a8f7bbadf980d1015475bd.pem
0x216176e40ded68df4345a75274318f7001030ae3.public.pem  0x76e47590efa1875f22a8f7bbadf980d1015475bd.public.pem
```

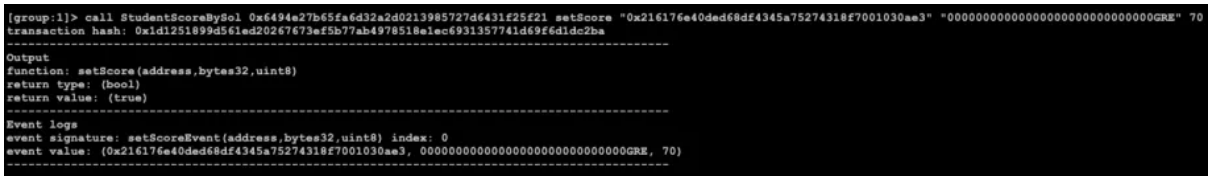
Solidity合约部署和调用

首先采用考试管理中心账户启动控制台，如下图所示，即表示控制台启动成功。

然后通过deploy命令对合约进行部署，合约部署成功之后会返回合约地址，如下图所示。

```
[group:1]> deploy StudentScoreBySol
contract address: 0x6494e27b65fa6d32a2d0213985727d6431f25f21
```

合约部署完成之后，可以通过控制台的call命令调用合约函数。如下图所示，新增学生的GRE成绩为70（修改和删除均可通过调用setScore方法进行操作），函数返回值为true，则表示交易成功。call命令的具体用法可以通过call -h 查看。



采用学生账户启动控制台，通过getScore函数查看成绩，如下图所示，返回值为70，说明没有问题。也可以使用学生账户调用setScore方法，会报错，打印没有权限，不再赘述。

CRUD合约部署和调用

CRUD合约的部署和调用和Solidity合约没有区别，这里同样采用控制台进行。

首先采用考试管理中心账户启动控制台，并对StudentScoreByCRUD合约进行部署。如下图所示。

合约部署完成之后，调用create函数创建stu_score表，如下所示，返回结果为0，说明创建成功。

```
[group:1]> call StudentScoreByCRUD 0x386bd131b0a73bf4a0cdb25bdc6b745b304c477b create
transaction hash: 0xbc58a155fa4c693eb9cc82f93d05a58615e80bd98f9108f572a22748bc4cff45
-----------------------------------------------------------------------------------------
Output
function: create()
return type: (int256)
return value: (0)
-----------------------------------------------------------------------------------------
Event logs
event signature: createEvent(address,string) index: 0
event value: (0x76e47590efa1875f22a8f7bbadf980d1015475bd, stu_score)
-----------------------------------------------------------------------------------------
```

创建好表之后，就可以调用相关接口对学生成绩进行操作了，首先调用insert函数新增学生成绩。如下所示，返回结果为1，说明插入成绩成功。

```
[group:1]> call StudentScoreByCRUD 0x386bd131b0a73bf4a0cdb25bdc6b745b304c477b insert "0x216176e40ded68df4345a75274318f7001030ae3" "GRE" 70
transaction hash: 0x43032408f1941e1a1c5149c170d529734a200d64873bed63c3740a3e207b130c
-----------------------------------------------------------------------------------------
Output
function: insert(address,string,int256)
return type: (int256)
return value: (1)
-----------------------------------------------------------------------------------------
Event logs
event signature: insertEvent(address,string,int256) index: 0
event value: (0x216176e40ded68df4345a75274318f7001030ae3, GRE, 70)
-----------------------------------------------------------------------------------------
```

成绩插入成功之后，关闭当前控制台，用学生账户登录控制台，并调用select函数查询成绩，如下图所示，返回70，说明查询成功。剩余函数测试可自行完成，不再赘述。

```
[group:1]> exit
work@instance-wgqad8dy:~/fisco/2.2/nodes/127.0.0.1/console_grayson/console$ ./start.sh 1 -pem accounts/0x216176e40ded68df4345a75274318f7001030ae3.pem
=====================================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 _______ _______ _______ _______ _______     _______ _______ _______ _______
|       |       \/       \ /       \ /       \    |       \ /       \ /       \ /       \
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\
| $$__     | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\$$
| $$  \    | $$  \$$    \| $$     | $$  | $$    | $$    $| $$     | $$  | $$\$$    \
| $$$$$    | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$$\
| $$      _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__| $$
| $$     |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$    $$
 \$$      \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$$

=====================================================================================
[group:1]> call StudentScoreByCRUD 0x386bd131b0a73bf4a0cdb25bdc6b745b304c477b select "GRE"
70
```

结束语

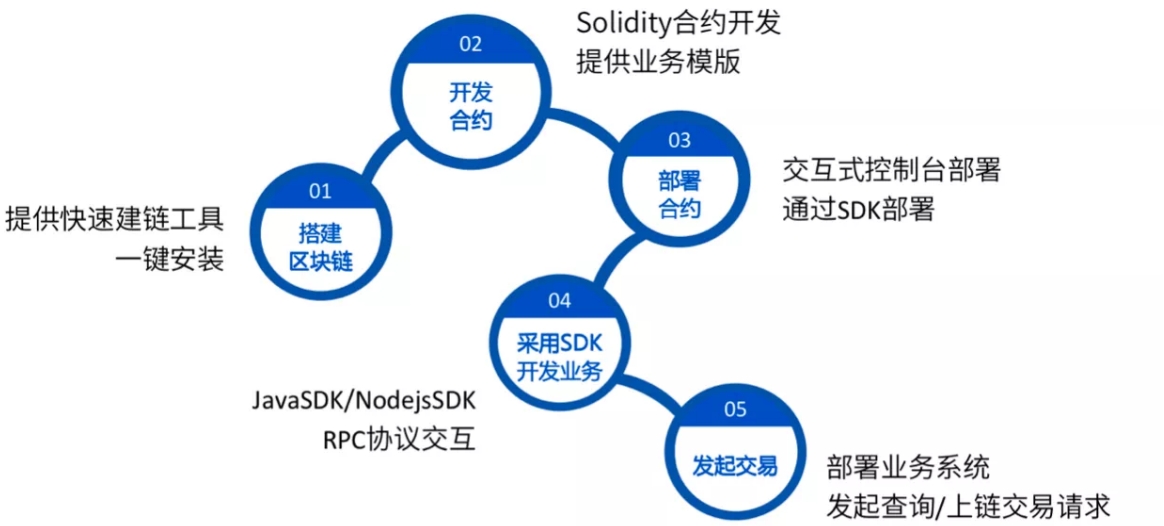
本文重点介绍了FISCO BCOS平台的智能合约开发。在FISCO BCOS平台中，既可以采用原生Solidity语言开发智能合约，也可以使用预编译合约模式开发合约。Solidity合约性能差、学习成本高；预编译合约，采用预编译引擎，支持并行计算，性能更高，同时支持存储扩容等。 但由于预编译合约使用存在一定的门槛，基于此，FISCO BCOS平台开发了CRUD合约接口，用户不需要关心底层的实现逻辑，只需要引入Table.sol合约接口文件，通过调用相关接口即可完成合约的开发。

16大技巧快速实现高水准智能合约

作者：张龙｜FISCO BCOS 核心开发者

什么是智能合约?

智能合约是指把合同/协议条款以代码的形式电子化地放到区块链网络上，通过调用相关条款的接口实现执行，也可以把它当成是一种可自动执行的程序片段。作为区块链的参与者，智能合约既可以接收和存储价值，也可以向外发送信息和价值。 在区块链中，智能合约非常重要，其贯穿了区块链应用开发全流程。

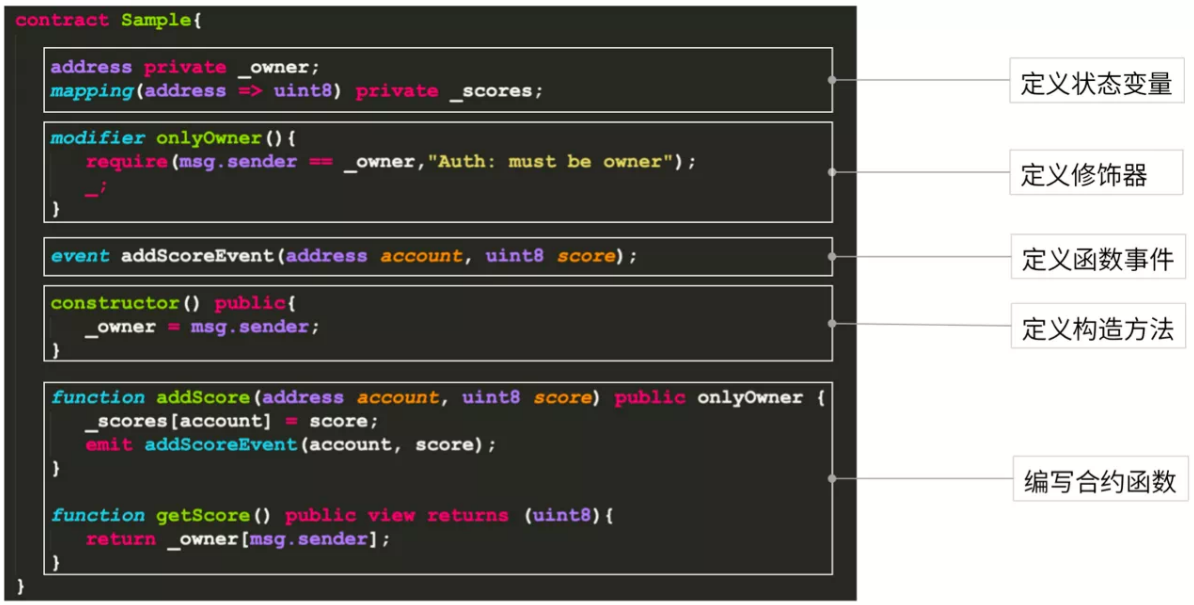


从另一个角度，我们再来看下智能合约在交易执行过程中的重要性。 首先，在交易构建中，我们会进行合约部署与函数调用，合约部署依赖于智能合约的二进制编码。函数调用则依赖智能合约的ABI，这里的ABI是在智能合约编译阶段产生。 交易签名是对构建好的交易进行签名，然后将其在网络上进行交易广播与交易打包。 在交易执行前，需要完成合约部署，把智能合约的二进制编码存放到区块链网络上。交易执行阶段，也会回调获取整个智能合约的二进制编码，根据构建的交易解析相应二进制片段，二进制片段对应交易的执行指令集，交易会按照指令集执行。

由此可见，智能合约在整个交易流程中也非常重要。

FISCO BCOS平台的智能合约类型

FISCO BCOS平台目前主要支持两种智能合约类型：Solidity智能合约与预编译智能合约。 预编译合约主要用于区块链底层平台，比如系统合约的实现。在应用开发层面，我们推荐使用Solidity合约和基于CRUD接口的预编译合约。 两者最大的区别在于，Solidity合约采用的是EVM引擎，预编译合约以及基于CRUD合约接口开发的预编译合约都是采用预编译引擎。这两个引擎在执行效力上有非常大的差异，后续会给大家详细介绍。 Solidity合约源自以太坊，现在许多平台都支持该合约。与其他开发语言类似，写Solidity合约需要先写合约名称，再去定义其状态变量，这里的状态变量类似java语言的成员变量，区别之处在于定义修饰符，修饰符用于条件或权限校验。



在这之后，是定义函数事件，函数事件重点记录了方法调用过程中的执行情况，以方便对智能合约的交易执行环节做业务层面的监控。定义构造方法的用法和创建类的实例化对象一样。 最后，我们会对定义的状态变量做一些操作或者业务上的处理，需要再去编写一些合约函数。以上就是Solidity合约的结构。Solidity合约的优点在于覆盖用户与应用范围广泛，其功能强大，且经过多年发展，已逐步成熟稳定。

但相比原生合约，Solidity语言对开发者而言具备一定学习门槛；同时，在执行过程中需要使用EVM，性能有局限，EVM对象对内存开销较大；最后，智能合约的数据和逻辑是相对耦合的，不便于合约升级和存储扩容。 针对Solidity合约的缺点，FISCO BCOS设计了预编译合约。

预编译合约也有一些不足，比如，分配一些固定的合约地址、对底层源码进行编译。为了解决这些问题，我们开发了CRUD合约接口，开发过程中用户只要继承Table合约，通过引入抽象的接口文件Table.sol就可以开发基于CRUD接口的预编译合约。

基于CRUD接口的预编译合约本质上与Solidity合约并没有特别多差异。主要区别有三个：

1. 需要引入Table.sol合约接口。
2. 进行上链交易时，首先用接口提供的功能创建一张表，只有这样数据和逻辑才能做到分离。
3. 对合约相关状态变量的操作过程中，借助Table合约相关接口操作合约数据。比如，通过Table.insert接入口进行insert操作。

基于CRUD接口的预编译合约优点非常明显：

1. 类似于数据库的操作，面向接口的编程，降低学习门槛与成本。
2. 底层通过预约引擎执行，可并行执行，所以其性能非常高。
3. 底层以表的形式存储数据，做到数据和逻辑分离，便于合约升级和存储扩容。

但是其也有一些不足。

1. 基于FISCO BCOS平台，无法跨平台。
2. 适用一些业务逻辑简单的场景，例如存证业务。

16个小技巧快速写好智能合约

在智能合约开发过程中，我们往往面对三座大山。

1. 合约安全。安全是智能合约、区块链应用的基础和命脉。在整个区块链发展历史中出现过多次因为智能合约漏洞给用户、平台带来重大损失的事件。
2. 合约性能。性能是衡量区块链应用可用性的重要指标，决定系统的负载能力和用户体验。
3. 可扩展性。可扩展性是智能合约及区块链应用系统应对业务变更和升级的有效手段，可以确保系统升级的时效性和成本。

针对这三座大山，我们整理了16大技巧，帮助大家快速上手智能合约开发。

合约安全篇

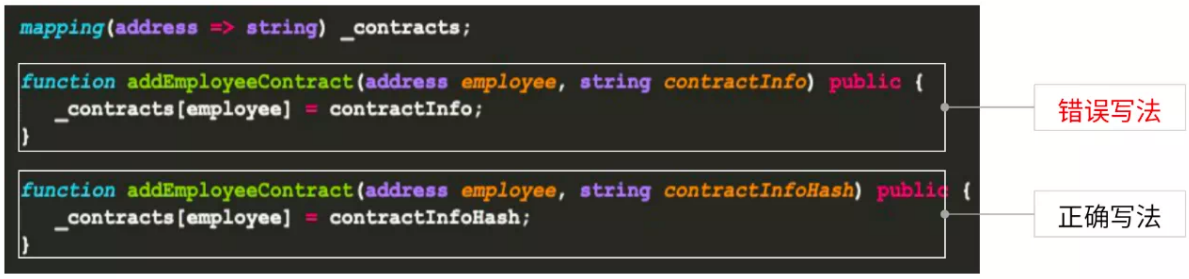
这里，我们总结了几个智能合约安全问题。

- 程序错误：初始化方式错误，变量隐藏造成混用
- 检查不足：权限和边界检查不足
- 逻辑缺陷：出块者可操纵、重入攻击
- 恶意合约：骗取tx.origin， RTLO字符攻击

针对智能合约开发过程中的安全问题，给大家几个建议。

技巧1：做好隐私数据的加密上链

智能合约上的数据是完全透明，所以需要一些数据隐私保护方案来确保数据安全。比如说通过哈希、同态加密或者零知识证明的方式对链上数据进行加密处理。



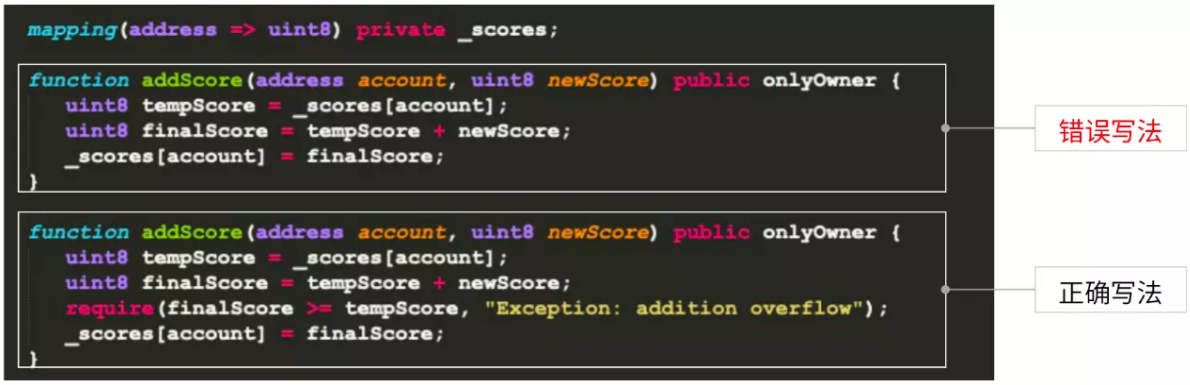
这份合约里的两个方法都很简单，就是添加一个雇员。在上面的写法中直接把合同相关信息写入，而下面的方法把合同哈希写入。为了确保用户隐私对合同进行加密处理。我们推荐第二种写法。

技巧2：合理的设置状态变量和函数的可见范围

这里有两个modifyScore，一个使用的是public，另一个使用的是internal。两者区别在于，下面的Test合约里继承了Base合约，在testFunction里利用onlyOwner调用modifyScore。 如果直接采用public修饰符，在这里并不起作用，它不会去校验onlyOwner，因为public函数方法是对外暴露的，用户无需通过testFunction去调用onlyOwner，而是直接调用。所以一定要注意状态变量和函数的可见范围。

技巧3：函数权限及变量边界检查

关于变量边界检查这里举一个例子：



我们要给学生成绩进行加分，假设其类型是uint8，在Solidity里uint8按数据换算是0到255的数据区间，如果不做校验将两个相加，可能会造成finalScore直接溢出，导致结果不正确。 所以我们在加法或者类似的变量操作之后，需要进行合法性的校验。如果在这里加一个require条件判断，就可以及时检查，确保业务逻辑的正确性。

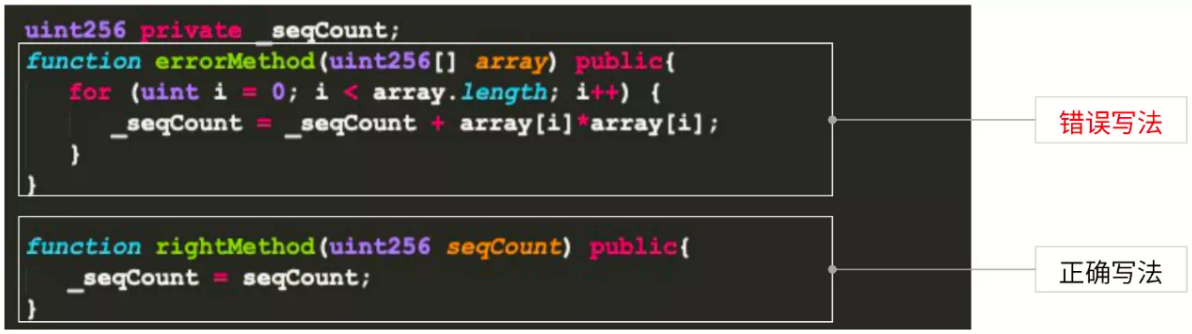
技巧4：学会利用安全工具

运用Securify，Mythx，Slither等工具对智能合约进行安全扫描，这样的工具非常多，有一些是完全免费的，大家可以了解并尝试使用自己感兴趣的工具。其他智能合约安全实现技巧这里就不过多赘述。

合约性能篇

保证智能合约的性能非常关键，如果系统的性能达不到要求，将影响整个系统的可用性。 而性能取决于代码运行期间对机器资源的占用。机器资源主要包括CPU、内存、网络等。 与中心化系统不同，区块链在共识阶段需要每个节点对每笔交易进行执行，每个节点的机器配置可能都不一样，而最短板的机器会影响整个区块链网络的性能。所以在一定的机器配置下可以通过智能合约节省一些资源消耗。 这里我们也给到大家一些优化建议。

技巧5：减少不必要的计算和验证逻辑，减少CPU开销



这两个方法都是对一个数组进行平方求和，在errorMethod里对每个数进行平方计算。 下面的方法没有计算而直接输出结果，由于大量复杂计算会对区块链网络以及性能造成很大影响，建议像这样的复杂计算和验证逻辑无需在区块链智能合约里出现，而放到链下或者业务系统中实现。

技巧6：减少不必要的数据，减少内存、网络和存储的开销

这里定义了两个公司的状态变量，在Company1中companyInduction变量存放公司的简介。这在Company2里被注释掉了。这是因为公司简介对链上交易逻辑没有多大影响，但从性能角度，其占据大量区块链网络节点内存、网络和存储开销，从而对整个网络性能造成极大压力。 所以建议大家只需把跟业务关联的核心数据、轻量化数据上链。

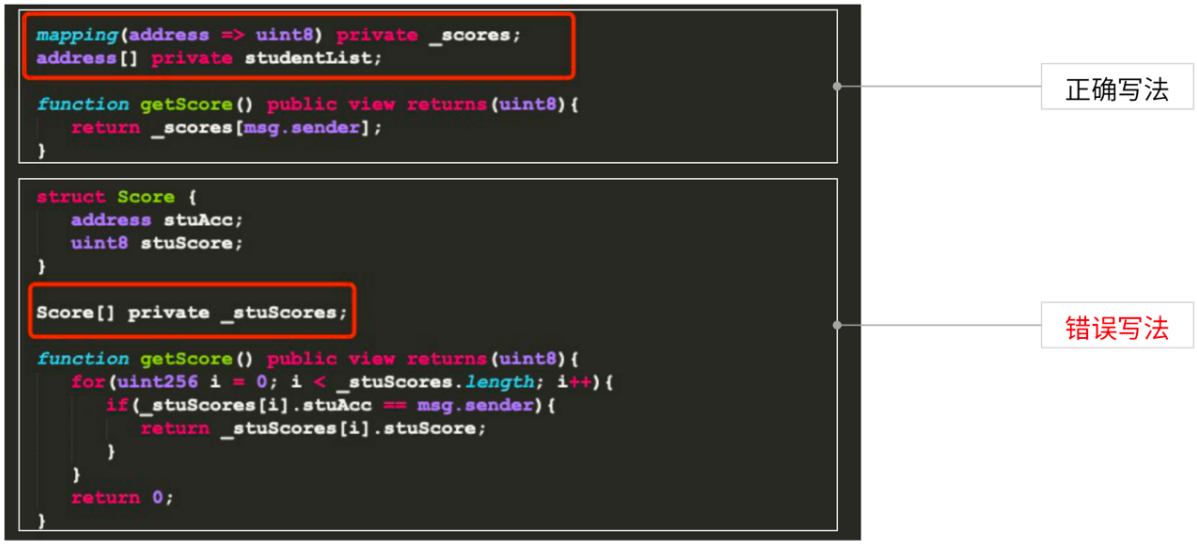
技巧7：利用不同的数据组装形式，减少跨合约调用

跨合约调用中，区块链网络节点会重新建一个EVM，这个过程需要消耗时间，对内存开销与区块链网络性能影响都非常大。 因此，推荐灵活使用不同的数据组装形式，例如，使用结构体避免跨合约调用，从而节省区块链节点内存、网络以及时间开销。

技巧8：通过智能合约提供的高级特性，比如继承，减少跨合约调用

上面两个例子，一个是通过合约引入直接调用，另一个是通过继承去调用合约里getName。在智能合约里的继承指的是在编译阶段，把父合约代码全部Copy到子合约中进行编译。也就是说，在最终合约里，父合约融入到了子合约里。当调用父合约时，就不是跨合约调用。

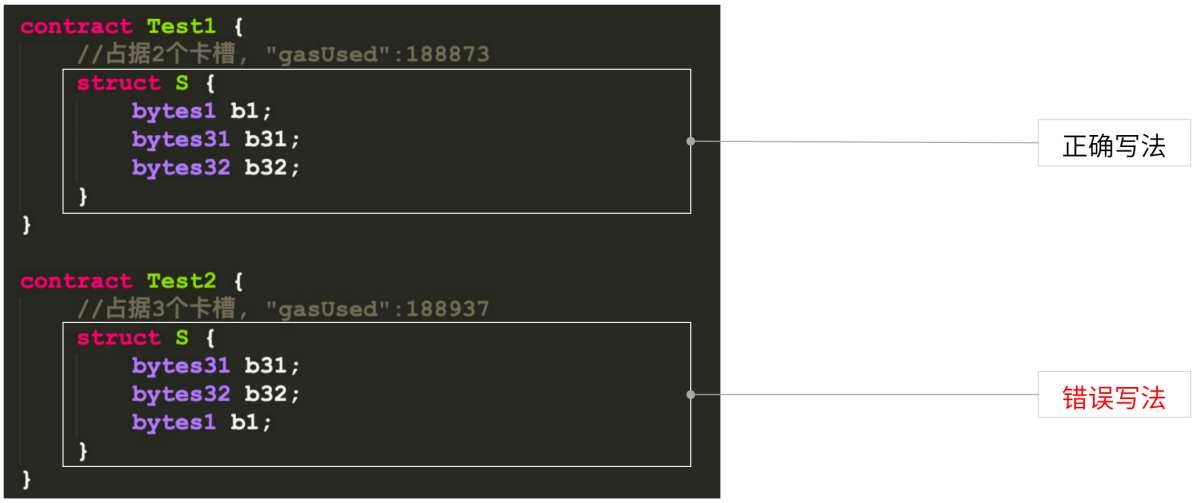
技巧9：改变数据类型，学会用空间换时间



上述例子通过mapping避免了数组的使用，提升查询时的性能。但是，按照以往经验，mapping所占用空间是数组的4倍，是否使用还要看业务具体需求：如果出于性能考虑，可以通过mapping改变数据类型，提升智能合约的执行效率。

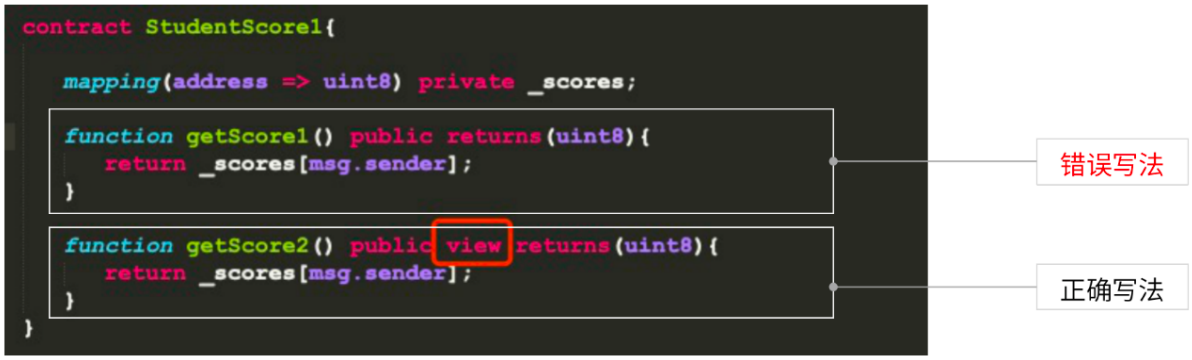
技巧10：紧凑状态变量打包，减少内存和存储开销

什么是紧凑状态变量？智能合约在EVM中的执行基于栈，栈里有相应卡槽，每个卡槽是32位左右。如果不注意变量顺序，会占用更多卡槽，消耗更多资源。下面的gasUsed就是计算机资源的开销。



在上面示例中，如果定义了一个bytes1、bytes31、bytes32，同样占据的是64个byte，这里因为bytes1和bytes31在同一位置，EVM自动打入一个卡槽。而错误写法里，EVM放到了两个不同卡槽。因此上面的结构体占据两个卡槽，下面的占据三个卡槽，它们使用资源的情况也不同，在实际操作中，需要注意这些细节。

技巧11：注意函数修饰符，减少不必要的执行



函数修饰符一般包括pure、view等，如果不加这些修饰符，区块链网络会自动把智能合约理解为一笔交易。按照以太坊黄皮书定义，查询操作为调用，对状态数据产生变更的就可以理解为交易。交易需要经过执行、共识过程，调用则不用。在智能合约中，如果设置了view修饰符，就是一次调用，不需要执行共识、进入EVM，而是直接查询本地节点数据，因此性能会得到很大提升。

可拓展性篇

在智能合约开发过程中，上链部署之后，对智能合约的升级是非常复杂的事情。智能合约可扩展性的价值就在于以下几点：

- 最小化成本（时间+人力）进行业务升级
- 作为系统异常的应急处理方式
- 便于其他人的接手和维护

以下一些技巧，可帮助大家提升智能合约可扩展性。

技巧12：逻辑和数据的分离：采用经典三层架构

```
contract ScoreManager{

    mapping(address => uint8) private _scores;

    function setScore(address account, uint8 score) public{
        _scores[account] = score;
    }
}
```

```
contract Score {
    uint8 private _score; //get
    uint8 private _version; //get
    constructor(uint8 score, uint8 version) public {
        _score = score;
        _version = version;
    }
}

contract DataMapping{
    mapping(address => address) private _dataMap;
    function setData(address account, address score) public {
        _dataMap[account] = score;
    }
    function getData(address account) public view returns(address){
        return _dataMap[account];
    }
}

contract ScoreManager{
    DataMapping private _dataMap;
    function addScore(address account, uint8 score) public {
        Score scoreAddress = new Score(score, 1);
        _dataMap.setData(account, scoreAddress);
    }
}
```

左边的智能合约通过setScore对成绩做管理，但如果要对学生成绩增加其他属性，那整个合约需要重新定义、部署，导致之前链上的数据没法使用。 采用三层架构就能解决这个问题。 首先，我们把数据单独放在Score合约里，通过datamap管理，然后通过ScoreManager对数据操作。这就是经典三层架构，确保逻辑和数据分离。 如果Score里要新增其他字段，比如studentid，我们只需更新Score合约与ScoreManager合约，就可做到兼容。由于datamap里的数据完全不能变，我们只需在Manage合约里对不同数据实体进行不同逻辑处理就可以确保合约的可扩展性。

技巧13：抽象通用逻辑

```
contract AccountManager{
    address public _owner;
    constructor() public {
        _owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only authorized.");
        _;
    }
    mapping(address => bytes32) _accounts;

    function addAccount(bytes32 account, bytes32 accName) onlyOwner public {
        _accounts[account] = accName;
    }
}

contract CompanyManager{
    address public _owner;
    constructor() public {
        _owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only authorized.");
        _;
    }
    mapping(address => bytes32) _companies;

    function addCompany(bytes32 company, bytes32 comName) onlyOwner public {
        _companies[company] = comName;
    }
}
```

```
contract BasicAuth {
    address public _owner;
    constructor() public {
        _owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only authorized.");
        _;
    }
}

contract AccountManager is BasicAuth {
    mapping(address => bytes32) _accounts;

    function addAccount(bytes32 account, bytes32 accName) onlyOwner public {
        _accounts[account] = accName;
    }
}

contract CompanyManager is BasicAuth{
    mapping(address => bytes32) _companies;

    function addCompany(bytes32 company, bytes32 comName) onlyOwner public {
        _companies[company] = comName;
    }
}
```

在左边合约里，每个合约都有onlyOwner修饰符，如果10个合约同时使用这个修饰符，维护成本将非常高。因此，建议大家重构出Base合约，在具体业务合约中继承，就可以进行复用了。这就是抽象通用逻辑实现合约的可扩展性。当下次合约升级时，只需修改Base合约即可。

技巧14：模块化编程：单一职责模式

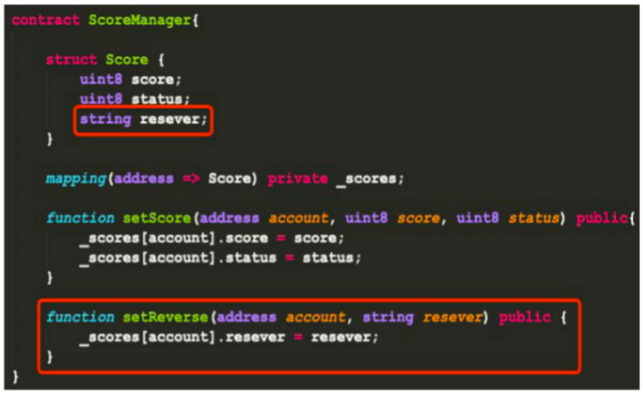
单一职责模式是一种编码规范。

左上的Rolemanager里包括两种角色account和company，也包括对两者的操作，其中显然违背了单一职责模式。一旦修改对账户的操作，对公司的操作也会随之改变，造成较大影响。 通过右下的方式把不同实体的操作分离。当修改account操作时，company操作不会受影响，从而减少了智能合约的维护成本。

技巧15：尽量复用成熟库

尽量复用成熟库的第一个好处是提升智能合约开发效率；第二个是减少智能合约编写中的漏洞，毕竟成熟库是经过前人大量业务实践总结出来的，其安全性有保证。

技巧16：适当预留空闲字段



在上面的ScoreManager合约里，Score结构体包括两个字段score本身与status成绩状态。如果需要增加studentid或其他备注信息，则需重新部署升级智能合约。因此，可以通过添加resever的保留字段的方式提升其可用性。但其实这样的方式也会对安全和性能造成影响。

今天主要跟大家分享了FISCO BCOS的两种智能合约。同时，针对智能合约安全、性能以及可扩展性，为大家提供了相应的开发技巧。智能合约的开发过程是安全、性能、可扩展性三者的博弈。开发者应根据实际业务需要选择适用的技巧与方案。

Q&A

Q： 合约升级重新部署，怎么让数据可以重用？两个合约地址，我的SDK层需要修改适配。

A： 合约通过三层模式进行逻辑和数据分离后，如果需要合约升级，需要在数据处理层对不同的合约数据做不同的处理，SDK层面也需要进行适配。

Q： 请问在uint256的话，在EVM栈中是占几个卡槽呢?

A： 占用一个卡槽。

Q： 智能合约能模糊查询吗？溯源如何处理?

A： 在区块链上模糊查询、历史数据查询等是不建议做的，因为区块链不适合做大数据处理。目前我们提供数据导出工具，已开源，可以助力业务快速进行大数据处理。

Q：请问智能合约中怎么安全的处理百分比呢?

A：智能合约没有小数类型,在上链前可以按照精度乘100或1000，链下处理在除一下就可以了.

Q：能否绑定智能合约在某个节点上运行？同一个智能合约在某个节点上可否开多个实例?

A：不行的，智能合约的部署及交易的执行，都会进行全网共识，在所有共识节点执行，并在所有节点落盘（或同步账本）。智能合约部署相当于创建一个智能合约实例，多次部署会产生多个实例。

存证&积分场景合约设计技巧与实例代码解析

作者：毛嘉宇 | FISCO BCOS 核心开发者

场景1：区块链+存证的权限合约编写

电子数据存证是记录“用户身份验证-数据创建-存储-传输”全过程的方式，应用一系列安全技术全方位确保电子数据的真实性、完整性、安全性，在司法上具备完整的法律效力。

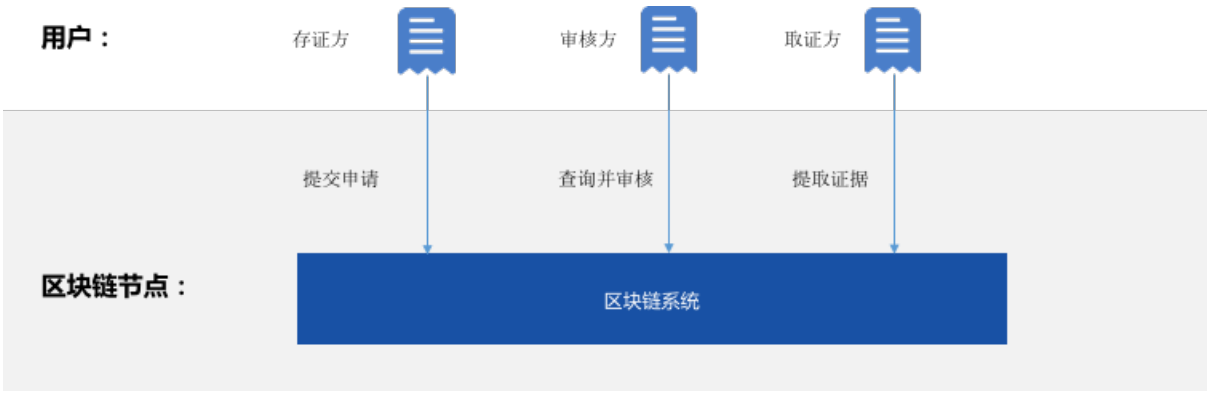
区块链技术的下列特点有助于降低成本、提升效率，并且确保存证数据的安全。

- 完善的防篡改机制：使用区块链技术保全证据，进一步加强了证据不可篡改性；
- 证据效力得到机构认可：司法机构作为链上节点，对链数据参与认可和签名，事后可从链上确认数据的真实有效性；
- 服务持续有效：数据被多方共识上链后，即使有部分共识方退出也不会造成数据的丢失或失效。

存证场景简要业务流程

在存证场景中可以抽象出三类典型用户：**存证方、审核方和取证方**。

- 存证方提交需要存证的申请。
- 审核方基于内容，对存证数据进行审核和签名确认。实际业务场景中，审核方可能会涉及投票和多方审核的多签过程。
- 当存证上链后，取证方可随时查询存证者地址、时间戳和审核详情等相关信息进行核验。



存证场景权限合约实例讲解

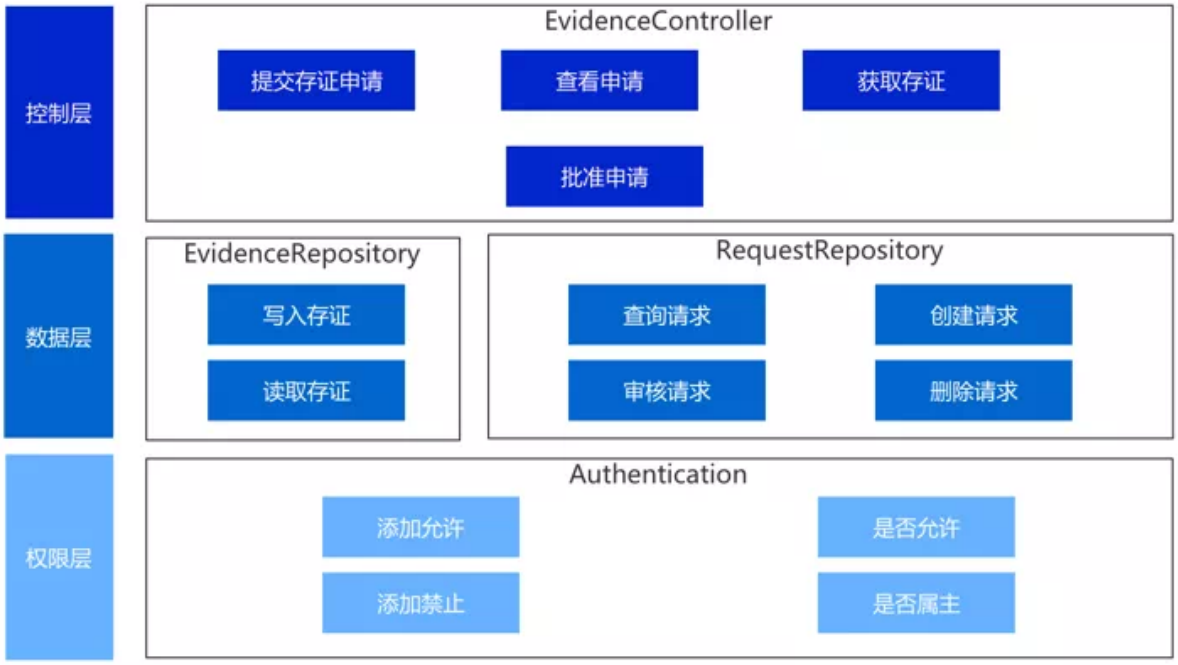
我们以存证场景中权限合约进行讲解。

存证合约概要设计

首先将逻辑和数据层分离。因为Solidity智能合约语言没有独立数据层，为便于合约后续扩展、升级，需要将逻辑和数据层分离，体现在下图里就是将数据层和控制层区分开。

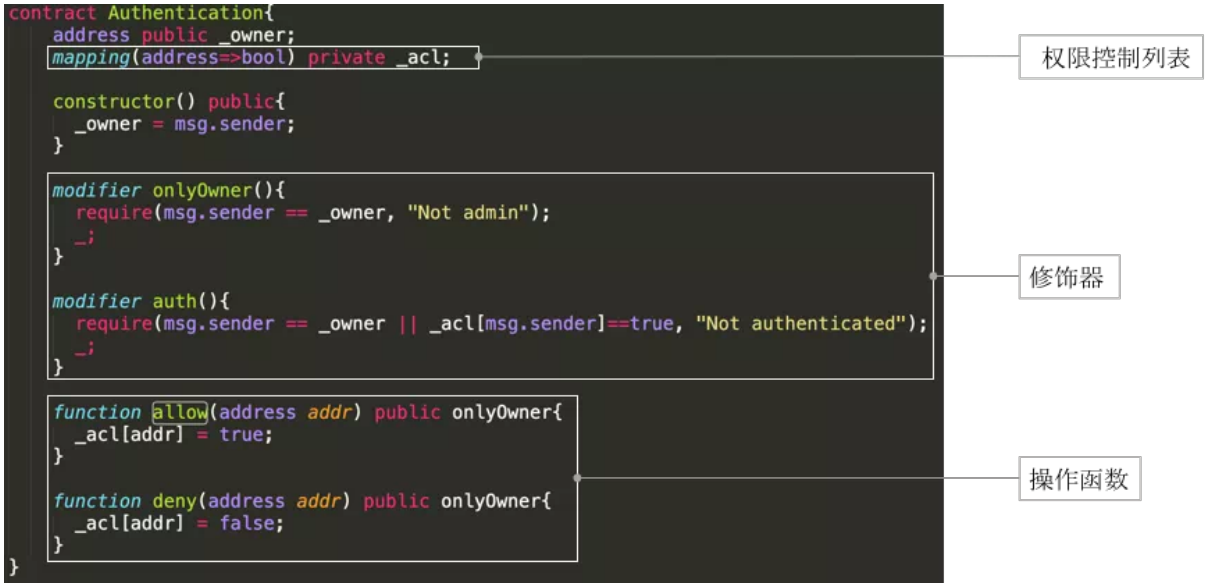
其次引入权限层。在一条联盟链上所有节点可以自由访问链上数据，智能合约提供了一种修饰器机制，可控制合约给指定授权用户访问，依据合约单一职责原理，将这一层抽象出来。

同时，我们需要控制数据层的权限，防止将写入数据层的接口权限开放给所有人，因此需要依赖并引入权限合约。



权限合约实例讲解

权限合约比较简单，不需要依赖其他合约，在很多合约开发中存在需求，可供复用。




```
contract Authentication{
    address public _owner;
    mapping(address=>bool) private _acl;

    constructor() public{
      _owner = msg.sender;
    }

    modifier onlyOwner(){
      require(msg.sender == _owner, "Not admin");
      _;
    }

    modifier auth(){
      require(msg.sender == _owner || _acl[msg.sender]==true, "Not authenticated");
      _;
    }

    function allow(address addr) public onlyOwner{
      _acl[addr] = true;
    }

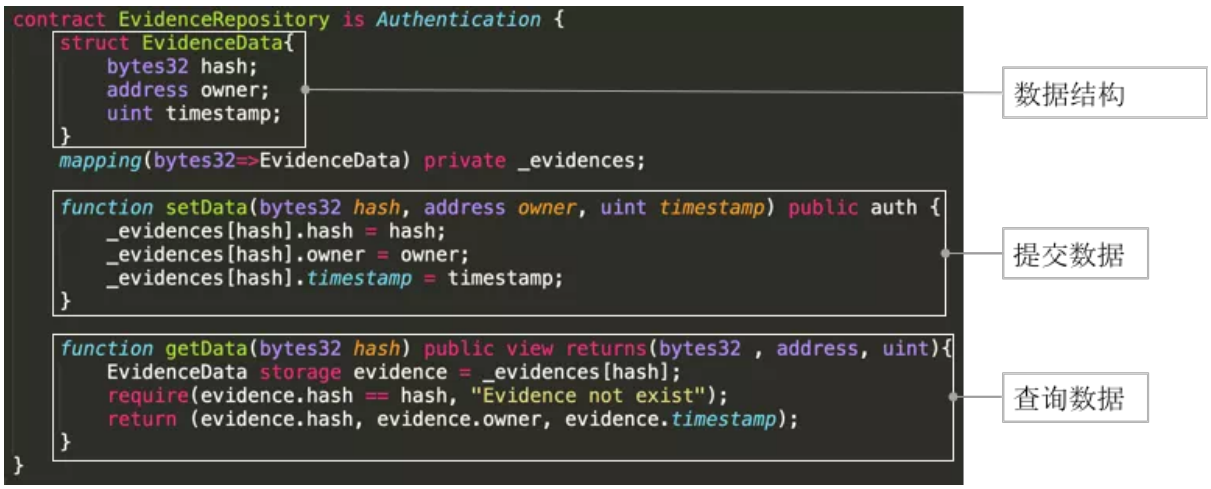
    function deny(address addr) public onlyOwner{
      _acl[addr] = false;
    }
}
```

Authentication合约里首先定义了两个成员变量，合约属主owner和权限控制映射列表acl(access control list)。

- **owner**：在合约构造时owner会自动被赋值为msg.sender，也就是合约的调用者，修饰器onlyOwner可以用来判断后续调用者是不是当初合约的创建者。
- **acl**：acl的变量是一个address到bool类型的mapping，我们看下allow和deny函数，allow将传入参数的地址所映射bool值设置为true，deny设置为false，设置完后可以通过auth修饰器来判断合约访问者权限。

存证数据

下图展示的是证据数据层，存证数据合约的代码。




```
contract EvidenceRepository is Authentication {
    struct EvidenceData{
        bytes32 hash;
        address owner;
        uint timestamp;
    }
    mapping(bytes32=>EvidenceData) private _evidences;

    function setData(bytes32 hash, address owner, uint timestamp) public auth {
        _evidences[hash].hash = hash;
        _evidences[hash].owner = owner;
        _evidences[hash].timestamp = timestamp;
    }

    function getData(bytes32 hash) public view returns(bytes32 , address, uint){
        EvidenceData storage evidence = _evidences[hash];
        require(evidence.hash == hash, "Evidence not exist");
        return (evidence.hash, evidence.owner, evidence.timestamp);
    }
}
```

EvidenceRepository是存证数据仓库，它继承了权限合约，权限合约里的方法和修饰器可在存证仓库的合约被使用。

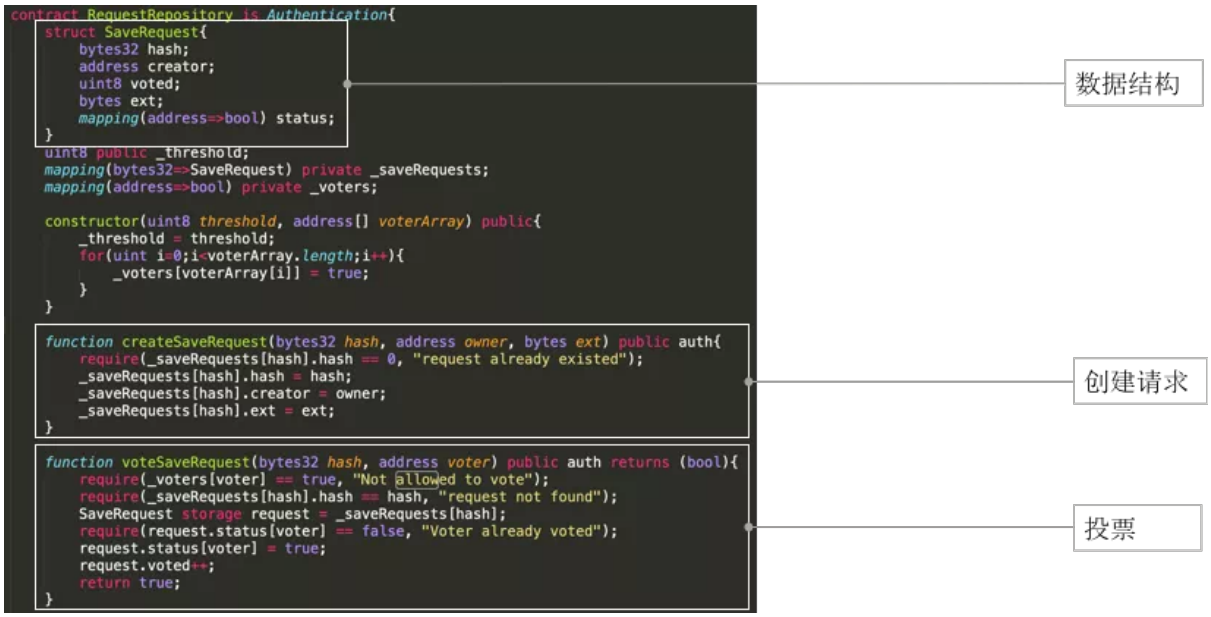
- 存证数据合约定义了一个struct的结构体EvidenceData，用来存放存证数据。为了简化模型，我们只定义了三个最核心的数据字段：存证数据Hash值、提交者地址和存入时间戳，在实际业务存证场景中大家可以根据所需字段进行扩展。
- mapping的变量，就是byte32到结构体的映射关系，实际上是以存证hash做为主键映射这个结构体的数据。key是hash，值就是上面的结构体，通过hash值可以检索查询和保存存证数据。
- 定义的函数中只确定了最核心的setData和getData函数。请注意，该合约本身继承了Authentication合约，所以在setData里能够使用auth的修饰器，控制只有被授权用户才能访问，防

止智能合约被部署到链上以后遭到恶意攻击或调用。getData函数根据传入的哈希把存入数据整体查询出来并返回。

可以看出，所有存证数据都被保存到数据合约里。这样可以起到统一存储、统一管理的效果。当然，这不一定是最优方案。 在业务场景中，如果合约拥有海量存证数据，则可能成为性能瓶颈，采用分拆设计方案会更加合理。

请求数据

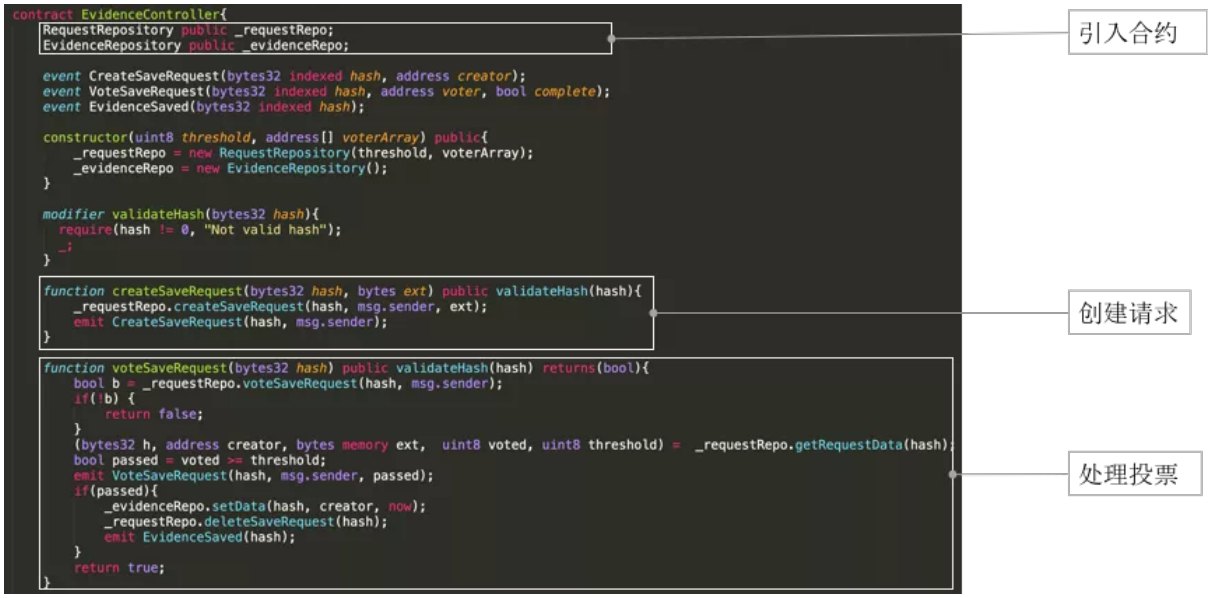
存证方开始提交存证数据并不会直接被写入到存证仓库中，而是经过审核方签名完成后才会真正提交，审核方可以为多方。



理解合约用途后，我们仔细看这个合约。

- 请求数据的合约结构和存证数据一致。第一部分是数据结构，SaveRequest是一个结构体，里面定义了一个存证请求详细的数据结构，包括最核心的存证hash、提交者、已经审核通过的票数。
- ext是一个描述说明字段，status记录了已经投票签名通过的审核方，threshold是投票阈值，voters是审核方地址列表，saveRequests定义了请求hash和请求本身的映射。类似存证数据合约，在请求数据合约中，所有请求数据也都存储在单个数据仓库合约里。
- 最后来看两个核心的函数：创建请求和投票审核。两个函数内都有auth修饰符控制权限。创建请求时，函数内部会使用require语句检查请求是否已经存在；投票时，函数内部会使用require语句检查审核者是否已经投票，请求本身是否存在，审核者是否合法。如果检查通过，审核投票票数加一，标记审核者已签名。

控制器



控制器引入了两个数据仓库合约，我们只需调用controller就可以完成所有用户接口的交互；它的构造函数参数变量包含了请求合约构造所需的参数：审核者列表及投票阈值，这个构造函数会自动构造和创建合约。

controller定义了两个方法，一个是创建存证请求，另一个是审核人根据请求进行投票。 创建请求函数较为简单，会直接调用请求数据仓库合约里的创建请求函数。 处理投票函数相对复杂。在验证hash数据非空后，会调用审核接口，如果审核成功，会触发检查当前请求审核通过数是否超过阈值，一旦超过，就自动保存到存证数据合约，同时，删除该请求。

此外，这个合约里还定义了三个event事件，有以下作用：

- 记录事件定义的参数，存储到区块链交易日志中，提供廉价存储。
- 提供一种回调机制，在事件执行成功后，由节点向注册监听的SDK发送回调通知，触发回调函数被执行。
- 提供一个过滤器，支持参数检索和过滤。

例如，createSaveRequest日志记录了hash和调用地址。如果配合SDK，我们可以实现对这个特定事件监听，并自动触发自定义回调函数。

存证合约示例小结

以上就是一个完整的存证场景权限合约demo。为了便于理解，我们并没有把例子设计得面面俱到，希望大家能更好地从中理解到demo的设计思想：

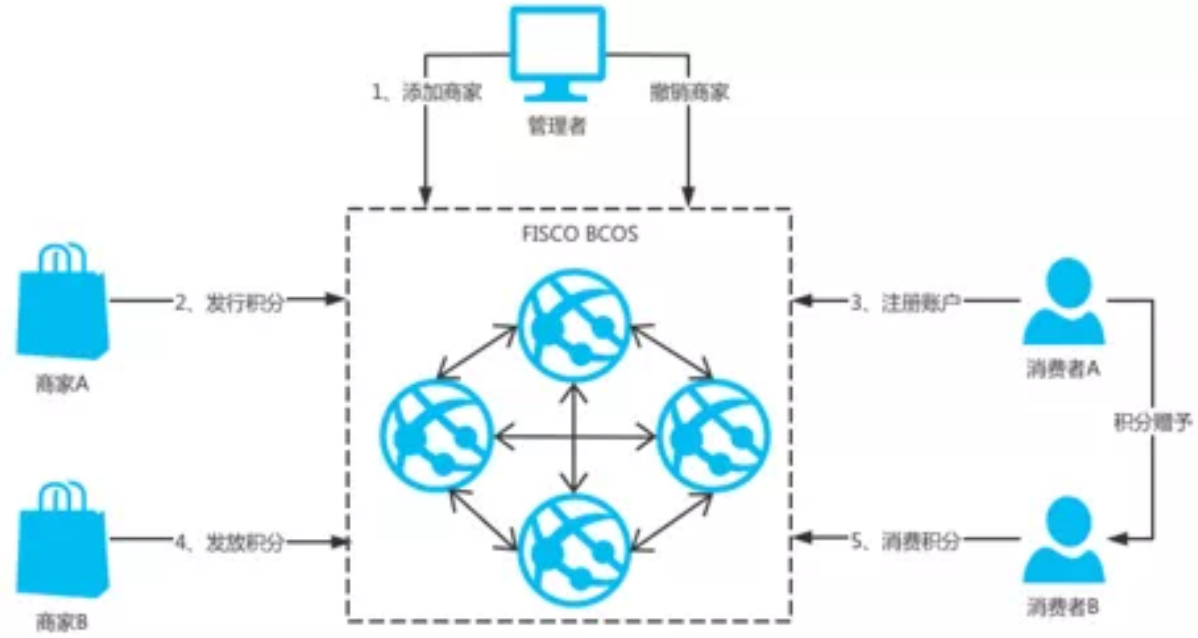
- 数据与逻辑分离；
- 建议采取自底向上开发，先开发依赖最少的部分，模块化、分层化的设计与实现；
- 关注权限控制与检查，避免出现一些非授权访问；
- 定义统一和清晰的接口；
- 存证数据hash上链。

场景2：区块链+积分合约实例讲解

下面介绍另一个智能合约典型应用场景——积分场景。

区块链技术可以如何解决积分场景中的这些痛点呢?

- 增加品牌曝光度：多个机构组成积分联盟，积分可有效通兑，实现客户资源引流，提升营销效果。
- 保证积分安全性：所有积分的生成和流转过程保存到链上，防止商户篡改和抵赖。
- 提升用户体验：不同商户和用户之间实现积分流转、互通，更加便利。



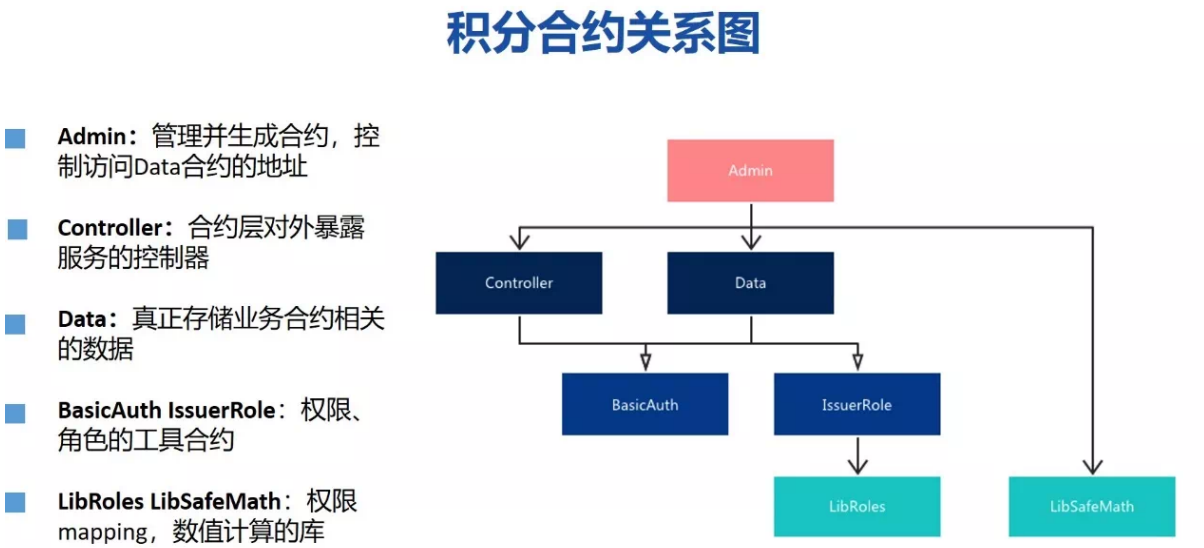
图：典型积分业务场景示例

一种思路是基于区块链技术，多个商家组成积分联盟，实现积分通存通兑、客户资源相互引流等。我们抽象了一个管理者，管理者部署和管理合约，商家有发行积分、拉入其他商家、撤销发行者身份的权限；消费者有开户、销户、消费积分和积分转账的权限。

积分场景合约实例讲解

先来做积分合约的概要设计。在存证合约中，我们引入了数据和逻辑分离的思想；在积分合约中，我们将引入管理、数据和逻辑分离的思想。 为什么要增加一层管理合约呢? 原有的两层结构中，控制合约会自动创建数据合约，而数据层合约中写死了属主是控制合约。 引入了管理合约后，就实现了类似控制反转的效果，控制合约和数据合约都由管理合约来创建；同时，管理合约还可以随时设置数据合约中控制合约的地址。这样，控制合约就可以随时实现平滑地业务逻辑升级；将管理合约分离出来，还有利于链

上权限治理。 此外，我们还将常用的权限、角色功能抽象为合约，并抽象了权限mapping和数据计算的库。



下面我们将看下积分合约具体的代码实现。

合约库——安全计算

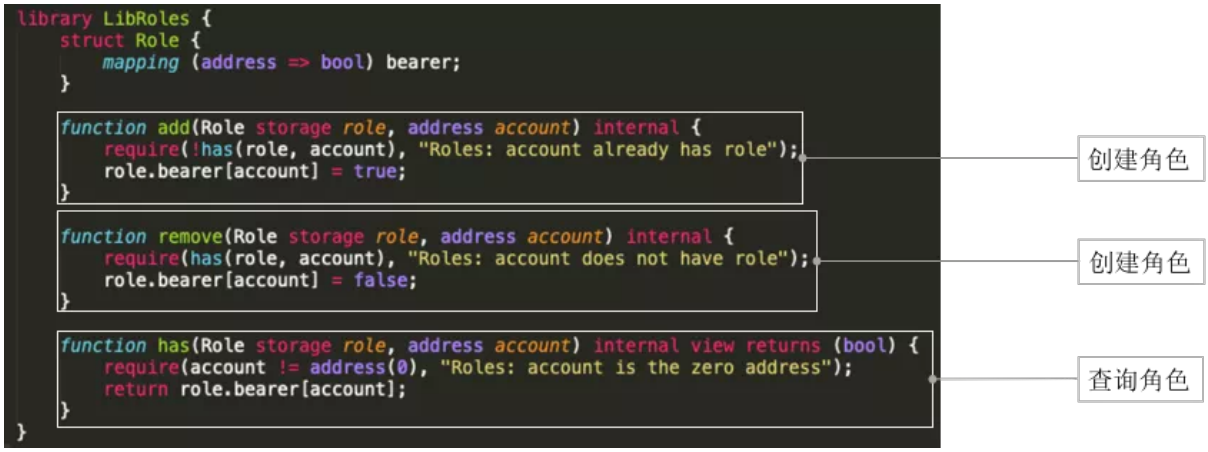
```
library LibSafeMath {
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;
        return c;
    }
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c =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

是否溢出

安全计算在Solidity中非常重要。涉及到数值计算部分，可优先考虑采用成熟开源的库。分享一个小技巧，由于链上资源非常宝贵，建议大家使用库的时候可以裁剪掉冗余的代码，节省资源。 安全计算的库，会在执行后重新检查数值，避免出现溢出，规避攻击。

库——角色管理

角色管理的库提供了创建角色、删除角色、查询角色的功能。这里有个基础的mapping bearer，是address到bool的映射，在mapping中维护role身份。



基础权限合约

在BasicAuth的基础权限合约中，我们提供了对属主的判断。

```
contract BasicAuth {
    address public _owner;

    constructor() public {
        _owner = msg.sender;
    }

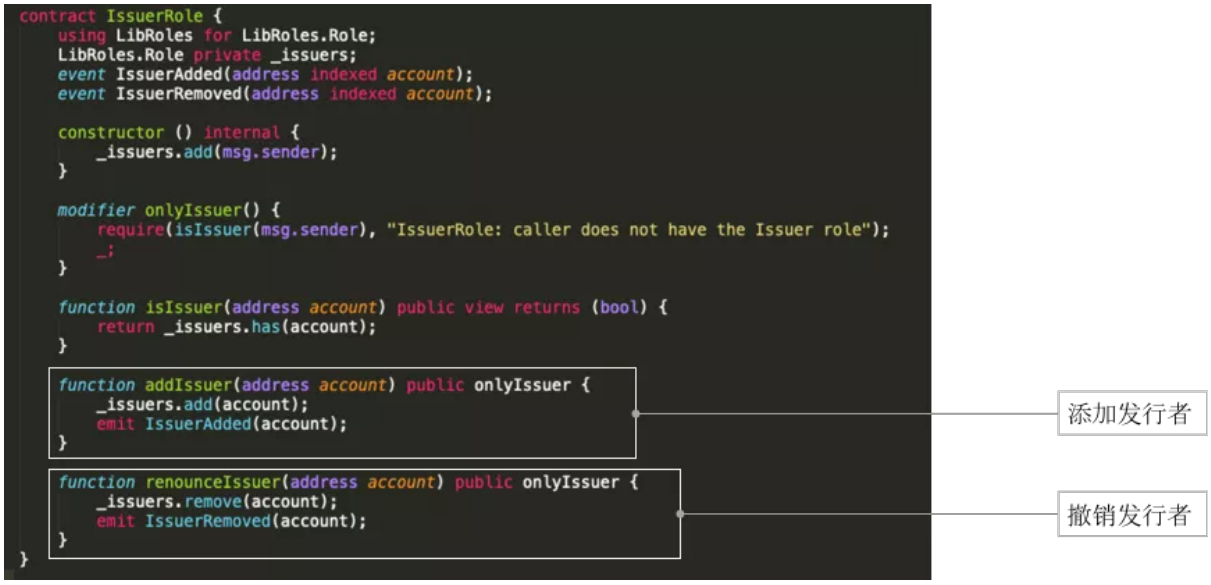
    modifier onlyOwner() {
        require(auth(msg.sender), "Only owner!");
        _;
    }

    function setOwner(address owner) public onlyOwner
    {
        _owner = owner;
    }

    function auth(address src) public view returns (bool) {
        if (src == address(this)) {
            return true;
        } else if (src == _owner) {
            return true;
        } else {
            return false;
        }
    }
}
```

发行者合约

发行者合约依赖上面LibRole的合约。为了简化规则，易于理解，我们这样定义：发行者允许添加新的发行者，也可以撤销自己发行者身份。有了发行者角色后，我们就可以发行积分了。



积分数据合约

现在进入到积分合约主体——admin-controller-data三层架构。 首先介绍积分数据合约。它会将所有用户积分，以及角色信息都存到积分数据合约里。

- balances维护了每个用户的余额；
- accounts维护已经注册的账户；
- totalAmount系统里是总共发行的积分；
- description注释或备注信息
- latestVersion实现访问权限控制；
- upgradeVersion的方法，有授权用户可调用这个方法来升级合约，这里由Admin合约来发起并调用；
- setBalance方法，设置某个账户的余额必须通过onlyLatestVersion检查，只有有权限的owner才能调用这个数据合约。

管理合约

管理合约作用是创建所有的合约。constructor更新在数据合约中所持有的版本号，一旦controller需要升级，只需调用下这个方法即可。

控制合约

最后介绍控制合约。由于controller代码较长，这里只展示最典型的两个函数。 balance查余额调的就是数据查余额的接口。积分的消费通过transfer实现，其中会有很多修饰器来检查账户是否已经注册，是否有效等，此外采用了智能合约事件来输出和打印日志。

```
function balance(address addr) public view returns (uint256) {
    return _rewardPointData.getBalance(addr);
}

function transfer(address toAddress, uint256 value)
  accountExist(msg.sender) accountExist(toAddress) checkAccount(toAddress)
    public returns(bool b, uint256 balanceOfFrom, uint256 balanceOfTo) {
        uint256 balance1 = _rewardPointData.getBalance(msg.sender);
        balanceOfFrom = balance1.sub(value);
        _rewardPointData.setBalance(msg.sender, balanceOfFrom);
        uint256 balance2 = _rewardPointData.getBalance(toAddress);
        balanceOfTo = balance2.add(value);
        _rewardPointData.setBalance(toAddress, balanceOfTo);
        emit LogSend(msg.sender, toAddress, value);
        b = true;
}
```

积分合约示例小结

小结一下积分场景demo中的一些设计思想:

- 引用三层架构：使用数据、逻辑、管理来管理合约;
- 抽象了库、封装了常用的合约来复用;
- 必须时刻关注合约安全，比如合约数值计算是否正确，权限是否合适;
- 以合约为单位，对职责进行抽象，尽可能实现合约职责单一。

如何写出高质量设计说明文档

文档编写之『术』

『术』主要就是编写文档时结构上所需要素，这里分为了三类：

- 业务背景：首先编写文档时不要预设他人能理解所有的专业术语，其次要清楚地介绍业务存在的痛点，区块链能否解决这些痛点。
- 技术方案设计：需清晰地说明基础业务需求（如参与者、场景、活动），设计思路概要，详细的合约职责、函数、参数等。
- 使用说明：实际使用场景、上手指南和使用手册等。

文档编写之『道』

有了术，就能确保文档内容结构的完整性，文档就有了骨架和血肉；但是，『道』才是文档的灵魂和精髓所在，这里整理了5个关注点：

- 亮点：展示出本方案的独特性（创意/设计/功能/规范/文档）；
- 痛点：说清楚问题和解决方法；
- 重点：代码、注释、文档编写要条理清晰、可读性强，围绕问题和解决方案展开，不要为了炫技而炫技；
- 难点：基于智能合约特性，权衡规范、效率和安全性；
- 要点：以解释作为基本导向，不要预设别人能够理解所有业务和技术术语。

本文主要分享了存证和积分两个典型应用场景合约设计思路和实例代码解析，为大家总结了相关的开发技巧和文档写作技巧。 在智能合约开发过程中，开发者需根据实际业务需求选择适用的技巧与方案。正所谓『兵无常势，水无常形』，没有最优的设计，只有最合适的设计。

Q&A

Q：智能合约中如何获取之前的数据?

A：主要有两种方法：1.在智能合约中定义需要查询历史数据的函数，通过合约查询接口来查询。2.使用WeBASE-Collect-Bee数据导出组件将链上数据导出到链下数据库中，可以查询所有数据。

Q：合约中的变量定义了private 属性，也可以被链上公开获取到吗?

A：链上所有数据都是公开的，即便定义了private属性的变量，也可以通过技术手段获取。

Q：在智能合约内可不可以调用其他外部接口?

A：智能合约中可以调用其他合约接口，但是不能访问外部接口。

Q：如果积分在一定时间内会减少，这种积分场景怎么实现?

A：首先要定义一个积分减少的业务规则；其次要实现这种场景在技术上是可行的，例如可以设计一个积分销毁函数，调用这个函数后，可以减少指定账户积分并同时减少总积分数量。最后，具体的实现逻辑和方式取决于业务规则。另外，需要注意的是，智能合约不支持类似定时脚本机制，需要外部调用才能触发。

Q：在编写智能合约时一定要按照三层结构进行编写吗?

A：不一定，取决于具体的业务场景，需要从实际业务场景中分析利弊，选择合适的方案。一般场景下我们推荐分层，这样更加灵活，更利于合约升级和维护。

Q：怎么通过智能合约实现同一群组内不同用户数据读写权限管控?

A：数据一旦上链后，对链上所有参与者来说都是公开透明的，因此，在链上实现同一群组内不同用户数据读写权限的管控本身不可行。但我们可以通过在数据上链前对数据本身进行加密来实现类似效果。

Q：存证中的哈希值是合同或者发票的哈希值吗?

A：这个规则取决于具体应用需求，可以直接是文件的哈希或者是拼装了其他信息再算一个哈希值，需要看具体场景需求。

Q：如果上链的数据错误，怎么处理?

A：数据一旦上链后，就不能被篡改和被物理删除；但可以设计一种合约逻辑删除机制，例如，在特定的数据合约中添加一个状态字段来标记数据是否被删除。

预编译合约

FISCO BCOS 2.0原理解析：预编译合约架构设计

作者：白兴强 | FISCO BCOS 核心开发者

FISCO BCOS 2.0提出了一套预编译合约框架，允许用户使用C++来写智能合约。由于不进入EVM执行，预编译合约可以获得更高的性能，适用于合约逻辑简单但调用频繁，或者合约逻辑固定而计算量大的场景。

本文介绍预编译合约的起源和实现，主要包括以下几个方面：

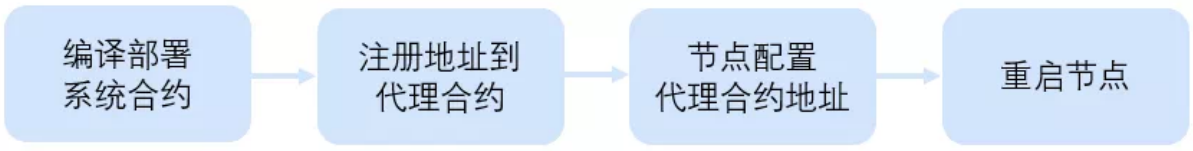
- Solidity合约的使用及遇到的难题；
- FISCO BCOS 2.0新增预编译合约，其架构设计和执行流程流程；
- 在部分特定场景中，为什么预编译合约比Solidity更优秀；
- 预编译合约在FISCO BCOS 2.0版本中的使用。

Solidity合约的使用与不足

在FISCO BCOS平台中使用Solidity合约，一般需经过以下五个步骤。在开发完Solidity合约之后，必须要将编译后的合约部署到底层平台，根据平台返回的地址才能够调用合约的接口。

Solidity合约的优点是完全与以太坊兼容，开发资源丰富且比较通用，但是Solidity合约也存在虚拟机执行性能低、代价高和开发复杂的问题。尤其是对于联盟链治理的场景，一些参数需要链上所有节点保持一致，非常适合使用合约管理，但如果使用Solidity实现，会导部署步骤非！常！复！杂！

FISCO-BCOS 1.3版本使用Solidity实现了一套系统合约，使用一个代理合约管理其他系统合约。其部署过程如下图所示：



部署完系统合约后，需要将系统合约地址配置在代理合约中，然后将代理合约地址配置在节点配置文件并重启，才能调用这一套系统治理的合约，并且之后的节点扩容也需要基于创世节点的配置操作，才能保持一致。

FISCO BCOS 2.0 新增预编译合约

FISCO BCOS 2.0受以太坊内置合约启发，实现了一套预编译合约框架。未来，我们还会尝试将现有的典型业务场景抽象，开发成预编译合约模板，作为底层提供的基础能力，帮助用户更快的更方便的在业务

中使用FISCO BCOS。

预编译合约的好处

可访问分布式存储接口：基于这套框架，用户可以访问本地DB存储状态，实现自己需要的任何逻辑。

更好的性能表现：由于实现是C++代码，会编译在底层中，不需要进入EVM执行，可以有更好的性能。

无需学习Solidity语言即可上手：基于FISCO BCOS预编译合约框架，开发者可以使用C++开发自己的预编译合约，快速实现需要的业务逻辑，而不需要学习Solidity语言。

并行模型大幅提升处理能力：我们在2.0版本中基于预编译合约和DAG实现了合约的并行执行，用户只需要指定接口冲突域，底层会自动根据冲突域构建交易依赖关系图，根据依赖关系尽可能并行执行交易，从而使得交易处理能力大幅提升。

预编译合约与以太坊内置合约的对比

上述说到，FISCO BCOS 预编译合约受以太坊内置合约启发，但实现原理却是大不相同的。

以太坊通过内置合约来避免EVM中复杂计算的代价，以太坊当前使用内置合约实现了8个函数（如下表所示）。可以看到，以太坊内置合约占用了0x1-0x8这8个地址，每个内置合约实际上就是一个本地函数的调用，只能用于状态无关的计算使用。

地址	函数
0x1	ecrecover
0x2	sha256hash
0x3	ripemd160hash
0x4	dataCopy
0x5	bigModExp
0x6	bn256Add
0x7	bn256ScalarMul
0x8	bn256Pairing

用户在Solidity中使用内置合约，需要借助call这个操作，依次输入下列参数

call(gasLimit, to, value, inputOffset, inputSize, outputOffset, outputSize)

包括内置合约地址、输入参数偏移、输入参数大小、输出参数偏移和输出参数大小，对用户而言，这不是一件简单的事情。

而FISCO BCOS的预编译合约框架，支持复杂的参数类型，支持通过AMDB读取和存储数据。每个预编译合约地址固定，合约内可以实现多个接口，所实现接口的调用方式与原生Solidity完全相同。

下图是比较直观的对比：

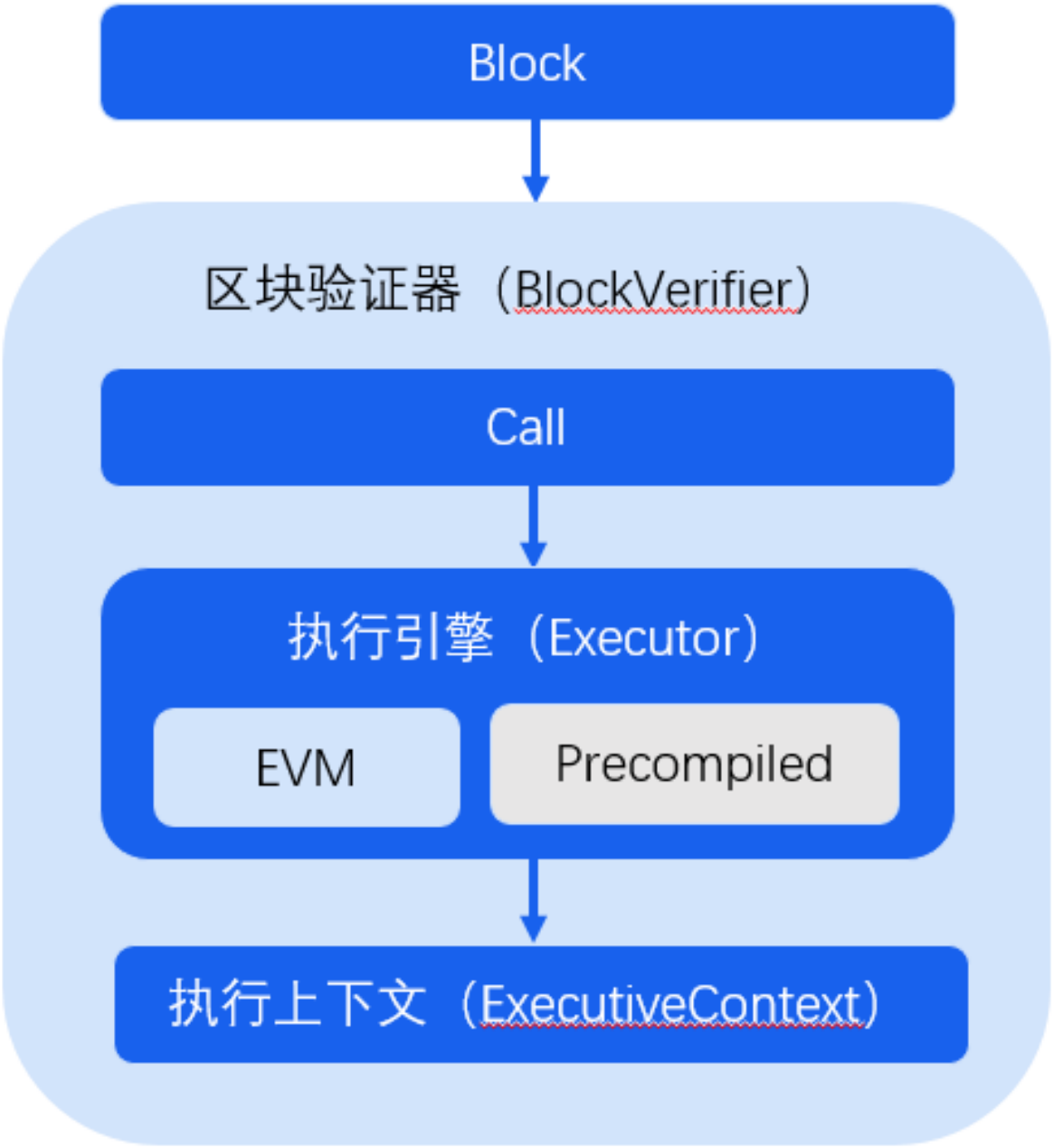
功能特性	FISCO BCOS 预编译合约	以太坊 内置合约
Solidity内调用	√	√
native执行	√	√
通过交易调用	√	×
访问存储	√	×
保存状态	√	×
复杂类型参数	√	×
动态创建临时预编译合约对象	√	×
单个预编译合约多接口	√	×

注：√代表支持，× 代表不支持

FISCO BCOS预编译合约架构

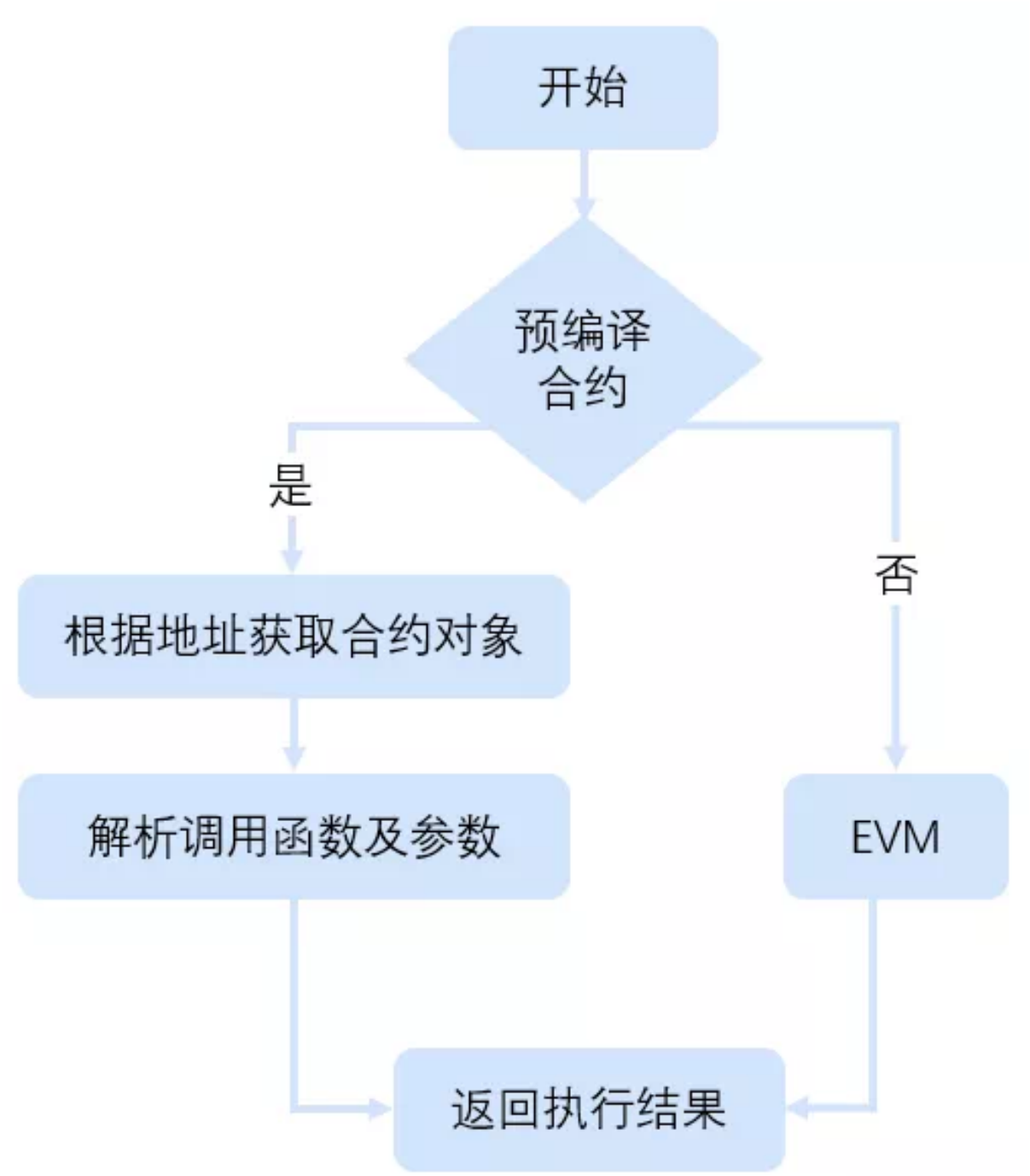
通过这一小节，你可以清楚了解预编译合约模块在FISCO BCOS中的位置，以及预编译合约的执行流程。

如下图所示，预编译合约会被区块执行引擎所调用，区块验证器通过区块执行引擎来执行区块，执行引擎执行区块时，会根据被调用合约的地址，来判断使用EVM还是预编译合约引擎。



当被调用的合约地址是EVM合约时，执行引擎会创建并执行EVM来执行交易；当被调用合约地址是已注册的预编译合约地址时，执行引擎通过调用地址对应的预编译合约接口来执行交易。

预编译合约执行流程如下图所示：



执行引擎首先根据预编译合约地址拿到合约对象，然后通过调用合约对象的call接口来获取执行结果。call接口中的操作主要包括：

1. 根据调用参数解析出被调用的接口
2. 根据ABI编码解析传入的参数
3. 执行被调用的合约接口
4. 将执行结果ABI编码并返回

所以，开发者如果要开发预编译合约，只需要实现其预编译合约的call接口和在执行引擎中注册所实现合约的地址即可。

预编译合约在FISCO BCOS 2.0中的应用

系统合约

FISCO BCOS 2.0 基于预编译合约实现了一套系统合约，使用系统合约来管理需要共识的链配置，包括对群组内节点的加入、删除、节点身份的转换、CNS服务的管理、链权限的管理、CRUD合约的使用等。

FISCO BCOS当前系统合约及地址如下表：

地址	合约	说明
0x1000	SystemConfigPrecompiled	实现对群组系统参数配置管理
0x1001	TableFactoryPrecompiled	CRUD合约写法使用的Table对象管理
0x1002	CRUDPrecompiled	CRUD接口实现，开发中
0x1003	ConsensusPrecompiled	群组节点及节点身份管理
0x1004	CNSPrecompiled	CNS管理
0x1005	PermissionPrecompiled	权限控制管理

CRUD合约支持

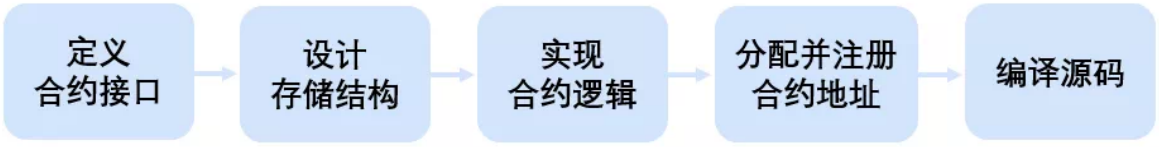
FISCO BCOS 2.0基于预编译合约实现了AMDB存储对应的预编译合约，使得用户能够在Solidity中访问AMDB存储，也就是FISCO BCOS 2.0的CRUD合约写法。通过这种方式，用户可以将合约数据存储在底层的AMDB存储中，使得合约逻辑与数据分离，一方面提高合约处理性能，一方面使得升级合约逻辑更加方便。

预编译合约极速开发指南（附完整步骤+实操模板）

作者：白兴强 | FISCO BCOS 核心开发者

上篇文章，我们亮出了FISCO BCOS预编译合约的架构设计，该框架具备地址固定、无需部署、本地执行性能更高等诸多优点。因预编译合约的使用方式与普通Solidity合约使用方式完全相同，该框架能做到在不改变客户端开发者体验的情况下，获得极高的运行速度，这对逻辑相对确定、追求高速度和并发能力的场景来说，可谓是屠龙刀一样的存在。

今天，我将以HelloWorld合约为例，为大家介绍如何使用预编译合约版本的HelloWorld。注意，本章内容需要你具备一定的C++开发经验，且详细阅读了《FISCO BCOS 2.0原理解析：分布式存储架构设计》。下图所示5个步骤是开发预编译合约的必经之路，我将按步骤实现HelloWorld预编译合约，然后分别使用控制台、Solidity合约两种方式来调用HelloWorld预编译合约。



HelloWorld预编译合约开发

先来看一下我们想要实现的HelloWorld合约的Solidity版本。Solidity版本的HelloWorld，有一个成员name用于存储数据，两个接口get(),set(string)分别用于读取和设置该成员变量。

```
pragma solidity ^0.4.24;

contract HelloWorld{
    string name;
    constructor() public {
       name = "Hello, World!";
    }
    function get() public constant returns(string){
        return name;
    }
    function set(string n) public{
      name = n;
    }
}
```

step1 定义HelloWorld接口

Solidity的接口调用都会被封装为一笔交易，其中，调用只读接口的交易不会被打包进区块，而写接口交易会被打包进区块中。由于底层需要根据交易数据中的ABI编码来判断调用的接口并解析参数，所以需要先把接口定义出来。预编译合约的ABI接口规则与Solidity完全相同，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，这个合约称为预编译合约的**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 HelloWorldPrecompiled{
    function get() public constant returns(string);
    function set(string n);
}
```

step2 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)。这在之前的文章分布式存储架构设计有介绍。如果合约中不涉及变量存储，可以忽略该步骤。对于HelloWorld，我们设计如下的表。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。

key	value
hello_key	Hello World!

step3 实现合约逻辑

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类， 重载call函数，在call函数中实现各个接口的调用行为。

```
virtual bytes call(std::shared_ptr<ExecutiveContext> _context,
    bytesConstRef _param, Address const& _origin) = 0;
```

call函数有三个参数，_ context保存交易执行的上下文，_ param是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，_origin是交易发送者，用于权限控制。 接下来，我

们在源码**FISCO-BCOS/libprecompiled/extension**目录下实现HelloWorldPrecompiled类，重载call函数，实现get()/set(string)两个接口。

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
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";
```

在call函数中添加打开表的逻辑：

```
// 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)
    {// 创建表失败，返回错误码
    }
}
```

区分调用接口：

```

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。

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。

HelloWorldPrecompiled实现：

考虑手机上的阅读体验，我们分块介绍call接口内部实现并省略部分错误处理逻辑，详细代码实现可以参考FISCO BCOS 2.0文档使用手册->智能合约开发->预编译合约开发。

```
bytes HelloWorldPrecompiled::call(dev::blockverifier::ExecutiveContext::Ptr _
↪context,
    bytesConstRef _param, Address const& _origin)
{
    // 解析函数接口
    uint32_t func = getParamFunc(_param);
    // 解析函数参数
    bytesConstRef data = getParamData(_param);
    bytes out;
    dev::eth::ContractABI abi;

    // 打开_ext_hello_world_表，省略
........
```

get()接口实现

```
// 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);
    }
```

set接口实现

```
else if (func == name2Selector[HELLO_WORLD_METHOD_SET])
    {  // set(string) 接口调用 略，请参考前文链接
        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())
        {  // update
```

(continues on next page)

(续上页)

```
            count = table->update(HELLOWORLD_KEY_FIELD_NAME, entry, table->
↪newCondition(),
                                  std::make_shared<AccessOptions>(_origin));
        }
        else
        {  // insert
            count = table->insert(HELLOWORLD_KEY_FIELD_NAME, entry,
                                  std::make_shared<AccessOptions>(_origin));
        }
        if (count == storage::CODE_NO_AUTHORIZED)
        { // 没有表操作权限
        }
        // 返回错误码
        out = abi.abiIn("", u256(count));
    }
    else
    {  // 参数错误，未知的接口调用
        out = abi.abiIn("", u256(CODE_UNKNOW_FUNCTION_CALL));
    }
    return out;
}
```

step4 分配并注册合约地址

FSICO BCOS 2.0执行交易时，根据合约地址区分是不是预编译合约，所以开发完预编译合约后，需要在底层注册为预编译合约注册地址。2.0版本地址空间划分如下：

地址用途	地址范围
以太坊内置合约	0x0001-0x0008
保留地址	0x0008-0x0fff
FISCO BCOS预编译合约	0x1000-0x5000
用户预编译合约	0x5001-0xffff
CRUD临时合约	0x10000+
Solidity	其他

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**。

开发者需要修改FISCO-BCOS/cmake/templates/UserPrecompiled.h.in文件，在registerUserPrecompiled函数中注册HelloWorldPrecompiled合约的地址(**要求v2.0.0-rc2以上版本**)，如下注册HelloWorldPrecompiled合约：

```
void ExecutiveContextFactory::registerUserPrecompiled(ExecutiveContext::Ptr␣
↪context)
{
    // 用户预编译合约地址范围 [0x5001,0xffff]
```

(continues on next page)

(续上页)

```
    context->setAddress2Precompiled(Address(0x5001), std::make_shared
↪<precompiled::HelloWorldPrecompiled>());
}
```

step5 编译源码

参考FISCO BCOS 2.0使用手册->获取可执行程序->源码编译。需要注意的是，实现的HelloWorldPrecompiled.cpp和HelloWorldPrecompiled.h需要放置于FISCO-BCOS/libprecompiled/extension目录下。

HelloWorld预编译合约调用

使用控制台调用HelloWorld预编译合约

在控制台solidity/contracts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]>
```

在Solidity中调用HelloWorld预编译合约

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solidity/contracts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();
```

(continues on next page)

(续上页)

```
    }
    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]>
```

到这里，就可以恭喜你顺滑地完成了HelloWorld预编译合约的开发，其他预编译合约的开发流程道理相通。

Solidity合约

智能合约初探：概念与演变

作者：储雨知 | FISCO BCOS 核心开发者

前言

自2009年比特币开启区块链时代以来，近10年里，随着技术与生态的发展，基于区块链的分布式应用（dapp）呈现出井喷的趋势，而支撑着dapp的底层技术就是“区块链+智能合约”。 智能合约与区块链的结合，普遍被认为是区块链世界中一次里程碑式的升级。第一个结合了区块链与智能合约技术的平台–以太坊的诞生，被认为是开启了”区块链2.0”时代。

什么是智能合约

1996年，Nick Szabo在文章《Smart Contracts: Building Blocks For Digital Markets》中提出了智能合约的概念。 所谓“合约”，就是条文、合同一类的东西，里面记录了发生的条件与对应执行的条款，以支持确权等操作；所谓”智能”，就意味着自动化、可编程。

所以，智能合约就是可编程的合同，也可以理解为一段自动执行的条文合同，在计算机中，就是一段自动执行的程序片段。它更易于合约保存，并且由确定的算法运行，给定输入，就得到对应的输出，极大保障了合约的执行力。 以自动售货机做类比，可以帮助我们更好地理解智能合约的核心特征。 当使用者选择好要购买的货物并完成支付，出货逻辑就会被触发，用户就能得到想要的货物，而这个过程不需要人工介入，节省了售卖货物的人力成本。如果要破坏这个合约，就得物理破坏售卖机。像POS刷卡机、EDI（电子数据交换）等，也可作此种类比。

智能合约与区块链

智能合约在上世纪被提出，而区块链2009年才诞生，就定义而言，智能合约与区块链关系不大。 那为什么在这10年中，智能合约与区块链却产生了如此紧密的关联？因为区块链可以保证智能合约的不可篡改，不仅合约内容不可篡改，每次调用记录亦不可篡改。 智能合约产生价值的最基本前提是有一个强有力的底层介质用于储存，让其不可被物理破坏。

然而，智能合约的本体是一份代码，非常容易被篡改，如何为其提供强力的存储介质就成了问题。这正好是区块链擅长解决的——通过比特币的实践，证明了区块链可以在分布式环境下让电子记录不可被篡改。 与此同时，智能合约也在反哺着区块链，它极大地扩展了区块链的业务场景。

与智能合约结合后，区块链不再服务于单一的货币支付，可以延申到生活中的方方面面。丰富的应用场景也对区块链的能力产生了新的挑战。

区块链2.0：以太坊的诞生

2009年诞生的比特币，运用区块链等技术来保证生态，开创了区块链1.0时代。 用户可以通过脚本代码来定制一些内容，例如如何解锁一笔资金。这些脚本代码会随着交易一起保存，从而享有不可篡改的特质，并且是确定性的。所以从某种角度来说，这些脚本也可看作智能合约。可是它们并不好用。 首先，这些脚本代码不是图灵完备的，这限制了实现的功能；其次，开发门槛较高，编写复杂逻辑的体验会很差，好比用JVM字节码来写程序。

2013年，一个青年V神提出了以太坊，其核心是通过世界状态对区块链数据进行更新和验证。以太坊与比特币最大的不同在于可通过智能合约执行复杂的逻辑操作。 在以太坊上，智能合约的语言是Solidity，它是图灵完备且较为上层的语言，极大地扩展了智能合约的能力范畴，降低了智能合约编写难度。 正因为此，以太坊的诞生，也标志着区块链2.0时代开启。随后，智能合约技术逐步渗透了溯源、存证、供应链等多个业务场景。

智能合约的现状与前景

从编程角度而言，智能合约就是一段代码。相比常规代码，智能合约具有许多差别与限制，例如:

- 单线程执行
- 代码执行会消耗资源，不能超出资源限制
- 目前难以获取链外数据，例如取得天气信息、比赛结果等
- 其他限制，如TPS

这些特点使得目前智能合约生态以链上资源的治理为核心。就像以太坊上各式各样的ERC标准与治理方案；EOS上有各种资源模型，比如CPU、RAM、兼经济模型、Rex、Bancor协议等。 显然，就目前的生态而言，智能合约对现实世界的影响力有限。 但事物总是在发展的。目前，已有许多致力于突破这些限制的研究，典型的有Oracle（谕言机，但常被称为预言机），它允许智能合约和链外进行交互，这样就能大大提高智能合约的使用场景，彷佛一台电脑通上了网；再比如那些突破链自身性能瓶颈的尝试，例如支付通道、跨链、plasma、rollup，它们都从不同角度打破安全与性能的枷锁。 毋庸置疑，智能合约将扮演着越来越重要的角色，将来随着以太坊2.0的落地，也许会开启新一个区块链时代。

智能合约技术

以太坊采用了Solidity作为智能合约语言，Solidity 是一门为实现智能合约而创建的高级编程语言，能在允许以太坊程序的节点上运行。该语言吸收了C++、JavaScript的一些特性，例如它是静态类型语言，支持继承、库等。 除了Solidity，每个平台的智能合约技术也有所不同，接下来将从公有链、联盟链作为切入，介绍其他平台所采用的技术。

公有链

首先，不妨先认识三大公链的智能合约技术。

链	合约语言	底层技术	图灵完备	备注
比特币	脚本语言	脚本语言	否	未被官宣为智能合约，但具备某些特征
以太坊	Solidity	EVM	是	最活跃的合约语言之一
EOS	C++/..	WASM	是	WASM是底层字节码，理论支持多种上级语言

联盟链

除了公链，联盟链也是重要的区块链类型。比之公链，联盟链共识的复杂度被大大缩减，因此具有更高的执行效率。 联盟链受企业级机构青睐，一般而言，相关机构之间会形成联盟，通过联盟链来共享数据。联盟链可覆盖供应链金融、司法存证、溯源等多种场景，未来还会与IOT、AI等技术结合。 在当今联盟链生态中，除去采用chaincode的Fabric，大部分平台都采用Solidity作为智能合约语言，FISCO BCOS即是如此。

时下，Solidity可谓占据了智能合约的C位，掌握Solidity是学习智能合约和区块链的重要一环。后面系列也将对如何用Solidity编写、运行以及测试智能合约作深入介绍解析，敬请关注。 除了Solidity，WebAssembly、Libra的Move等一些智能合约语言也在发展中，可以保持关注。

智能合约运行分析

下面将分享一个基础的智能合约：HelloWorld。

```
pragma solidity 0.4.22;

contract HelloWorld{

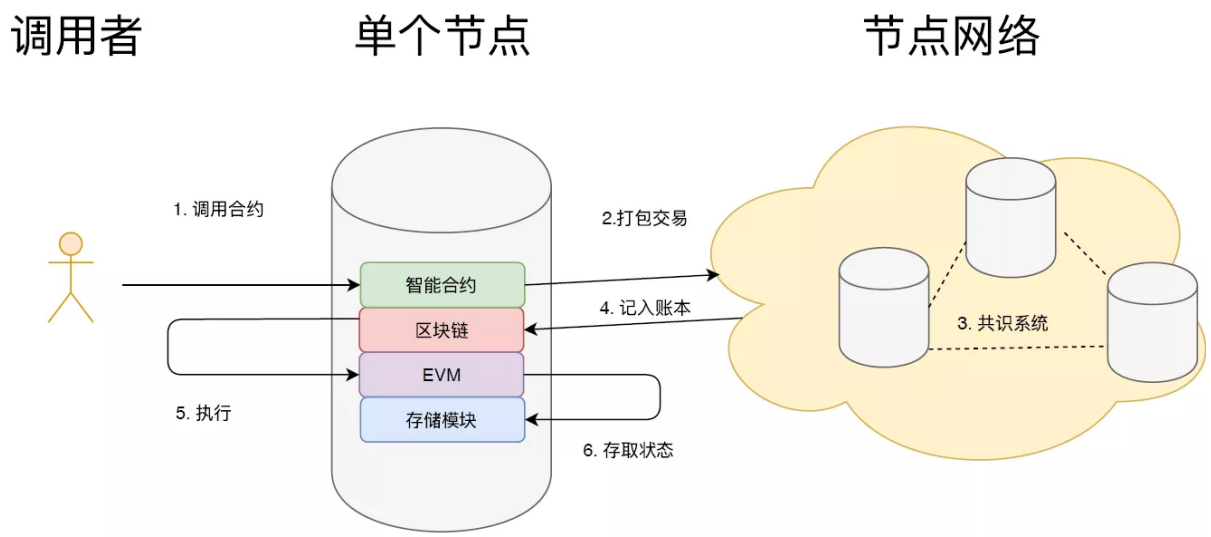
    uint private _num;
    function store(uint256 num) public {
        _num = num;
    }

    function get() public view returns (uint256){
        return _num;
    }
}
```

这段Solidity代码的功能是存取_num字段。该字段被称为“状态变量”，会由区块链持久存储。 用户可以将这段代码部署在以太坊或类似的区块链上，部署成功就意味着该智能合约不可再被修改，只要底层区块链不被销毁，这段合约就一直存在。任何人都可通过“合约地址”来调用该合约接口，每次调用信息都会被记录在链上。 在讲解这段代码如何运行之前，我们先回顾下传统java程序的运行方式。

首先，用户编译完java代码后，会以字节码的形式保存在磁盘上；然后用户会调用程序，这由JVM来托管执行；程序执行期间可能会通过日志来记录调用参数，也可能会和磁盘进行IO。 Solidity的执行与此

类似。不同的是介质由硬盘换成了区块链，由单机变为分布式。 代码部署后，以字节码的形式存储在每一个节点上。当用户要求调用某个函数时，调用请求将会被囊括在交易中，并被打包到某个区块上，一旦全网对该区块形成共识，就意味着调用是合法的。 接下来，EVM会来调用字节码，它负责存取底层的状态变量，好比传统编程的IO。



光从代码来看，合约开发似乎不过如此，单个合约只需要围绕着字段进行操作，对于很多简单业务而言，不过是CRUD而已。 但其复杂性也恰恰在于此，合约在区块链环境上执行，是不可修改的。 所以如果出现了bug，就必须部署新的合约，这对于合约的可维护性提出了挑战。并且，一旦业务复杂起来，容易出现安全漏洞，导致链上资产损失。同时，还要考虑完成代码编写、逻辑执行、数据存储的成本问题。 综上所述，写合约不难，但写好合约，却需要一定功底。

结语

本文介绍了智能合约的概念与历史演变。 智能合约是上世纪提出的技术，在区块链浪潮下焕发出了新的生机。反之，智能合约广泛的应用场景，又极大的促进了区块链发展。

学习智能合约，推荐选择Solidity语言，因为其具备一些传统语言的特征，同时，执行环境完全基于区块链，实际业务开发的体验与以往的编程体验会有所差别。 读者可以尝试基于FISCO BCOS快速搭建区块链环境，部署一个最简单的合约，熟悉部署、调用方式，再进一步走入Solidity的世界。

如何优雅地编写智能合约

作者：张龙 | FISCO BCOS 核心开发者

写在开头

众所周知，智能合约的出现，使得区块链不仅能够处理简单的转账功能，还能实现复杂的业务逻辑处理，其核心在于账户模型。

目前在众多区块链平台中，大多数集成了以太坊虚拟机，并使用Solidity作为智能合约的开发语言。Solidity语言不仅支持基础/复杂数据类型操作、逻辑操作，同时提供高级语言的相关特性，比如继承、重载等。

除此之外，Solidity语言还内置很多常用方法，比如成套的加密算法接口，使得数据加解密非常简单；提供事件Event，便于跟踪交易的执行状态，为业务的逻辑处理、监控和运维提供便利。

然而，我们在编写智能合约代码的时候，还是会碰到各种问题，这些问题包括：代码bug、可扩展性、可维护性、业务互操作的友好性等。同时，Solidity语言还不完善、需要执行在EVM上、语言本身及执行环境也会给我们带来一些坑。

基于此，我们结合之前的项目和经验进行梳理，希望将之前碰到的问题总结下来，为后续的开发提供借鉴依据。

⊙ 注：智能合约安全不在本篇文章讨论范畴，文中智能合约代码为0.4版本写法。

Solidity常见问题

EVM栈溢出

EVM的栈深度为1024，但是EVM指令集最多访问深度为16，这给智能合约的编写带来很多限制，常见的报错为：stack overflows。

这个报错出现在智能合约编译阶段。我们知道EVM的栈用于存储临时变量或者局部变量，比如函数的参数或者函数内部的变量。优化一般也是从这两个方面出发。

下述代码片段可能存在栈溢出问题：

```
//如果课程超过14个，那么参数超过16个，则溢出
function addStudentScores(
  bytes32 studentId,
  bytes32 studentName,
  uint8 chineseScore;
  uint8 englishScore;
  ...
  uint8 mathScore
)
  public
    returns (bool)
{
    //TODO
}
```

函数参数和局部变量不能超过16个，一般建议不超过10个。参数过多会出现的问题：

1. 容易栈溢出；
2. 编写代码费劲容易出错；
3. 不利于业务理解和维护；
4. 不便于扩展。

常规做法是尽量减少函数参数，碰到实在无法减少的情况，建议采用数组。局部变量和函数参数类似，定义过多也会导致栈溢出，可以通过拼接数组，减少变量个数，一般会和数组入参结合使用，上述代码片段优化后如下所示：

```
function addStudentScores(
  bytes32[] studentInfo,
    uint8[] scores
)
  public
    returns (bool)
{
    //TODO
}
```

BINARY字段超长

智能合约通过JAVA编译器编译后会生成对应的JAVA合约，在JAVA合约中有一个重要的常量字段BINARY，该字段为智能合约的编码，即合约代码。合约代码用于合约部署时签名，每一次合约的变更对应的BINARY都会不一样。

在编写智能合约时，如果单个智能合约代码很长，经过编译后的BINARY字段会很大。在JAVA合约中，BINARY字段用String类型存储，String类型的最大长度为65534，如果智能合约代码过多，会导致BINARY字段的长度超过String类型的最大长度，导致String类型溢出，从而报错。

解决方案也非常简单：

1. 尽可能复用代码，比如某些判断在不同的方法中多次出现，可以抽取出来，这样也便于后续的维护；
2. 合约拆分，将一个合约拆分成为多个合约，一般出现String越界，基本上可以说明合约设计不合理。

慎用string类型

string类型是一个比较特殊的动态字节数组，无法直接和定长数组进行转化，其解析和数组转化也非常复杂。

除此之外，string类型浪费空间、非常昂贵（消耗大量gas），且不能进行合约间传递（新的实验性ABI编译器除外），所以建议用bytes代替，特殊场景例外，比如未知长度字节数组或预留字段。

⊙ **备注**：string类型可以通过在合约中添加新的实验性ABI编译器（如下代码）进行合约间传递。

```
pragma experimental ABIEncoderV2;
```

智能合约编写

分层设计

网上多数智能合约的例子，比如著名的ERC20等，通常做法是写在一个智能合约文件中，这种写法本身没有什么问题，但面临复杂的业务，这种写法无可避免地会出现：

1. 代码全部写在一个文件中，这个文件就非常大，不便于查看和理解，修改容易出错；
2. 不便于多人协作和维护，尤其是业务发生变动或代码出现漏洞时，需要重新升级部署合约，导致之前的合约作废，相关业务数据或资产也就没有了。

那么，有没有一种方法可以使得智能合约升级又不影响原有账户（地址）？

先给答案：没有！（基于底层的分布式存储的CRUD除外，目前FISCO BCOS 2.0支持分布式存储，可直接通过CRUD操作数据库进行合约升级。）

但是！没有并不意味着不能升级，智能合约升级之后最大的问题是数据，所以只要保证数据完整就可以了。

举个例子：我们需要对学生信息上链，常规写法如下所示：

```
contract Students {
  struct StudentInfo {
        uint32 _studentId;
        bytes32 _studentName;
    }
    mapping (uint32 => StudentInfo) private _studentMapping;
    function addStudent(uint32 studentId, bytes32 studentName) public returns(bool)
↪{
      //TODO:
    }
}
```

这种写法，代码全部在一个智能合约中，如果现有的智能合约已经不能满足业务诉求，比如类型为uint32字段需升级为为uint64，或者合约中添加一个新的字段，比如sex，那这个智能合约就没有用了，需要重新部署。但因为重新部署，合约地址变了，无法访问到之前的数据。

一种做法是对合约进行分层，将业务逻辑和数据分离，如下所示：

```
contract StudentController {
  mapping (uint32 => address) private _studentMapping;
    function addStudent(uint32 studentId, bytes32 studentName) public returns(bool)
↪{
      //TODO:
    }
}
contract Student {
  uint32 _studentId;
  bytes32 _studentName;
    //uint8 sex;
}
```

这种写法使得逻辑和数据分离，当需要新增一个性别sex字段时，原始数据可以编写两个StudentController合约，通过版本区分，新的Student采用新的逻辑，需要业务层面做兼容性处理，其最大的问题是对于原有数据的交互性操作，需要跨合约完成，非常不方便，比如查询所有学生信息。

我们再次进行分层，多出一个map层，专门用于合约数据管理，即使业务逻辑层和数据层都出现问题，也没有关系，只需要重新编写业务逻辑层和数据层，并对原有数据进行特殊处理就可以做到兼容。不过，这种做法需要提前在数据合约中做好版本控制（version），针对不同的数据，采用不同的逻辑。

这种做法最大的好处是数据全部保存在StudentMap中，数据合约和逻辑合约的变更都不会影响到数据，且在后续的升级中，可以通过一个controller合约做到对新老数据的兼容，如下所示：

```
contract StudentController {
  mapping (uint32 => address) private _studentMapping;
  constructor(address studentMapping) public {
      _studentMapping = studentMapping;
    }
    function addStudent(uint version, uint32 studentId, bytes32 studentName, uint8␣
↪sex) public returns(bool){
      if(version == 1){
            //TODO
        }else if(version == 2){
            //TODO
        }
    }
}
contract StudentMap {
  mapping (uint32 => address) private _studentMapping;
    function getStudentMap() public constant returns(address){
      return _studentMapping;
    }
}
contract Student {
  uint8 version;
  uint32 _studentId;
  bytes32 _studentName;
    //uint8 sex;
}
```

统一接口

智能合约尽管具备很多高级语言的特性，但是本身还是存在很多限制。对于业务的精准处理，需要采用Event事件进行跟踪，对于不同的合约和方法，可以编写不同的Event事件，如下：

PS：你也可以采用require的方式进行处理，不过require方式不支持动态变量，每个require处理后需要填入特定的报错内容，在SDK层面耦合性太重，且不便于扩展。

```
contract StudentController {
  //other code
    event addStudentSuccessEvent(...); //省略参数，下同
    event addStudentFailEvent(...);

    function addStudent(bytes32 studentId, bytes32 studentName) public␣
↪returns(bool){
      if(add success){
          addStudentSuccessEvent(...);
            return true;
        }else {
          addStudentFailEvent(...);
            return false;
        }
    }
}
```

这种做法也没有问题，不过我们需要编写大量的Event事件，增加了智能合约的复杂性。如果每次新增加一个方法或者处理逻辑，我们都需要编写一个专门的事件进行追踪，代码侵入性太强，容易出错。

除此之外，基于智能合约的SDK开发，对于每一个交易（方法）由于Event事件不同，需要编写大量的不可复用的代码，解析Event事件。这种写法，对于代码的理解和维护性都是非常差的。要解决这个问题，我们只需要编写一个基合约CommonLib，如下所示：

```
contract CommonLib {
  //tx code
  bytes32 constant public ADD_STUDENT = "1";
    bytes32 constant public MODIFY_STUDENT_NAME = "2";

    //return code
    bytes32 constant public STUDENT_EXIST = "1001";
    bytes32 constant public STUDENT_NOT_EXIST = "1002";
    bytes32 constant public TX_SUCCESS = "0000";

  event commonEvent(bytes id, bytes32 txCode, bytes32 rtnCode);
}

contract StudentController is CommonLib {
     function addStudent(bytes32 studentId, bytes32 studenntName) public␣
↪returns(bool) {
      //process add student
        if(add success){
          commonEvent(studentId, ADD_STUDENT, TX_SUCCESS);
            return true;
        }else {
          commonEvent(studentId, ADD_STUDENT, STUDENT_EXIST);
            return false;
        }
    }
    function modifyStudentName(bytes32 studentId, bytes32 studentName) public␣
↪returns(bool){
      //TODO:
    }
}
```

当新增一个modifyStudentName方法或其他合约时，原有的做法是根据方法可能出现的情况定义多个Event事件，然后在SDK中针对不同的Event编写解析方法，工作量很大。现在只需要在CommonLib中定义一对常量即可，SDK的代码可以完全复用，几乎没有任何新增的工作。

⊙ 注：在上述例子中，commonEvent包含三个参数，其中txCode为交易类型，即调用的哪个交易方法，rtnCode为返回代码，表示在执行txCode所代表的交易方法时出现什么情况，这两个参数是必须的。在commonEvent中还有一个Id字段，用于关联业务字段studentId，在具体的项目中，关联的业务字段可以自行定义和调整。

代码细节

代码细节能体验一个coder的能力和职业操守。在业务比较赶的情况下，经常会忽略代码细节，同时代码细节（风格）因人而异。对于一个多人协作的项目，统一的代码风格、代码规范，能极大提升研发效率、降低研发及维护成本、降低代码错误率。

命名规范

智能合约命名并没有一个标准，不过团队内部可以按照一个行业共识的规范执行。经过实战，推荐以下风格（不强制），如下代码块。

1. 合约命名：采用驼峰命名、首字母大写、且能表达对应的业务含义；
2. 方法命名：采用驼峰命名、首字母小写、且能表达对应的业务含义；
3. 事件命名：采用驼峰命名、首字母小写、且能表达对应的业务含义，以Event结尾；
4. 合约变量：采用驼峰命名、以_开头，首字母小写、且能表达对应的业务含义；
5. 方法入参：采用驼峰命名、首字母小写、且能表达对应的业务含义；
6. 方法出参：建议只写出参类型，无需命名，特殊情况例外；
7. 事件参数：同方法入参；
8. 局部变量：同方法入参。

```
contract Student {
  bytes32 _studentId;
    bytes32 _studentName;
  event setStudentNameEvent(bytes32 studentId, bytes32 studentName);
  function setStudentName(bytes32 studentName) public returns(bool){}
    //other code
}
```

条件判断

在智能合约中，可以通过逻辑控制进行条件判断，比如if语句，也可以采用solidity语言提供的内置方法，比如require等。

两者在执行时存在一些差异，一般情况下，使用require没有问题，但是require不支持传参，如果业务需要在异常情况下给出明确的异常提示，则推荐使用if语句结合Event使用，如下。

```
event commonEvent(bytes id, bytes32 txCode, bytes32 rtnCode);
//require(!_studentMapping.studentExist(studentId),"student does not exist");
if(_studentMapping.studentExist(studentId)){
  commonEvent(studentId, ADD_STUDENT, STUDENT_EXIST);
  return false;
}
```

常量及注释

在智能合约中，常量和其他编程语言一样，需要采用大写加下划线方式命名，且命名需具备业务含义，同时需要采用constant关键词修饰，建议放置在合约开头。

常量也需要区分，对外接口常量采用public修饰，放置在基合约中。业务相关常量采用private修饰，放置在具体的业务逻辑合约中。如下所示：

```
contract CommonLib {
    //tx code
  bytes32 constant public ADD_STUDENT = "1";
    bytes32 constant public MODIFY_STUDENT_NAME = "2";
    ...
}

contract StudentController is CommonLib {
  /** student status */
    bytes32 constant private STUDENT_REGISTERED = "A";
    bytes32 constant private STUDENT_CANCELED = "C";

    //other code
}
```

智能合约的注释同大部分编程语言，没有很严格的要求。对于一些特殊字段、常量、数组中的每个变量及特定逻辑，需进行说明，方法及Event可以使用/** comments */，特定字段及逻辑说明可采用//。如下所示：

```
/**
* stundent controller
*/
contract StudentController {
  /** add student */
    function addStudent(
      //[0]-seqNo;[1]-studentId;[2]-studentName;
      bytes32[3] studentInfos
)
      public returns(bool)
{
      //TODO:
    }
}
```

兜底方案

在智能合约设计过程中，谁都无法保证自己的代码一定满足业务诉求，因为业务的变动是绝对的。同时，谁也无法保证业务及操作人员一定不会犯错，比如业务对某些字段未做校验导致链上出现非法数据，或者因为业务操作人员手误、恶意操作等，导致链上出现错误数据。

区块链系统不像其他传统系统，可以通过手动修改库或文件对数据进行修正，区块链必须通过交易对数据进行修正。

针对业务变更，在编写智能合约时可以适当增加一些保留字段，用于后续可能存在的业务变更。一般定义为一个通用化的数据类型比较合适，比如string，一方面string类型存储容量大，另一方面几乎啥都可以存。

我们可以在SDK层面通过数据处理将扩展数据存入string字段，在使用时提供相应的数据处理反向操作解析数据，比如在Student合约中，新增reserved字段，如下所示。当前阶段，reserved没有任何作用，在智能合约中为空。

```
contract Student {
  //other code
    string _reserved;

    function getReserved() constant public returns(string){
    return _reserved;
  }

  function setReserved(string reserved) onlyOwner public returns(bool){
```

(continues on next page)

(续上页)

```
    _reserved = reserved;
    return true;
  }
}
```

针对手误或者非法操作导致的数据错误，务必预留相关的接口，以便在紧急情况下可以不修改合约，而通过更新SDK对链上数据进行修复（SDK中可以先不实现）。比如针对Student合约中的owner字段，添加set操作。

```
contract Student {
  //other code;
    address _owner;
  function setOwner(address owner) onlyOwner public returns(bool){
    _owner = owner;
    return true;
  }
}
```

需要特别注意的是，对于预留字段和预留方法，必须确保其操作权限，防止引入更多问题。同时预留字段和预留方法都是一种非正常情况下的设计，具备超前意识，但一定要避免过度设计，这样会导致智能合约的存储空间非常浪费，同时预留方法使用不当会给业务的安全性带来隐患。

写在最后

区块链应用的开发涉及很多方面，智能合约是核心，本篇给出了开发智能合约过程中的一些建议和优化方法，但并不是完整和完美的，且本质上无法杜绝bug的出现，但通过优化方法，可以让代码变得更加健壮和易维护，从这点上来讲，已具备业界的基本良心要求了。

深入浅出Solidity

作者：石翔 | FISCO BCOS 核心开发者

在只有比特币的年代，区块链能够实现简单的价值产生和转移，但却未出现更多的商业模式。以太坊给区块链带来了维度的提升，基于区块链的应用渐趋丰富，区块链的各种商业模式加速涌现。这其中很重要的原因，是以太坊给区块链带来了一套图灵完备的编程语言。

区块链的主要功能，是实现了多方的共识。在比特币中，需要共识的操作是固定的，是非图灵完备的。其共识的仅仅是价值所有者的改变。但以太坊上，开发者可以自己编写需要共识的逻辑，以太坊通过智能合约语言Solidity，实现了共识逻辑的自定义。

Solidity 介绍

Solidity语言和Java存在着些许相似之处。在众多编程语言中，Java是发展较为成熟的。Java代码在Java虚拟机（JVM）执行。JVM屏蔽掉了操作系统的差异，使得Java成为一个跨平台的语言。一套Java代码可在Windows、Linux、Mac上通用，而不需要关心操作系统的差异。

Solidity与Java类似。代码写好后，都需要通过编译器将代码转换成二进制，在Java中，编译器是Javac，而对于Solidity，是solc。生成后的二进制代码，会放到虚拟机里执行。Java代码在Java虚拟机（JVM）中执行，在Solidity中，是一个区块链上的虚拟机EVM。

Solidity与Java的不同之处在于，Solidity是服务于区块链的语言，代码在区块链上执行。EVM是区块链上的一个执行器。每个区块链节点都有一个EVM。Solidity在EVM中被执行后，EVM对区块链的数据进行了改变。这些数据的改变交由共识算法去共识。同时，Solidity的操作仅限于EVM内部，不能访问外部不确定系统或数据，如系统时钟，网络文件系统等。

Solidity的设计目的，是给区块链提供一套统一的逻辑，让相同的代码跑在区块链的每个节点上，借助共识算法，让区块链的数据以统一的方式进行改变，达到全局一致的结果。

Solidity 实现细节

以此处的Demo合约为例，合约中有一个全局变量m，并有一个函数add()，实现给全局变量m增加x数值的功能。

```
pragma solidity ^0.4.25;

contract Demo {
    int m = 0;

    function add(int x) public {
        m = m + x;
    }
}
```

通过合约编译器solc，可将此合约编译成二进制。二进制的每个字（8 bit），表示一个EVM的操作码（OPCODE）。Demo合约编译出的二进制及其相应的OPCODE如下，实现了完整的Demo合约的功能，包括对合约的装载、合约接口的调用和异常处理的逻辑。其中，标红部分是add()方法的实现。

Binary

```
608060405260008055348015601357600080fd5b5060a88061002260000396000f30060806040
5260043610603f576000357c0100000000000000000000000000000000000000000000000000000000
000000900463ffffffff16806387db03b7146044575b600080fd5b348015604f57600080fd5b506
06c60048036038101908080359060200190929190505050606e565b005b8060005401600081
905550505600a165627a7a7230582034cde309944b55c8ff48aa2ac34fa292ce468222f1beff97c
9c945186a49b4cd0029
```

OPCODE

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x0 DUP1 SSTORE CALLVALUE DUP1 ISZERO PUSH1 0x13 JUMPI
PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0xA8 DUP1 PUSH2 0x22 PUSH1 0x0 CODECOPY PUSH1 0x0
RETURN STOP PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1 0x0
CALLDATALOAD PUSH29 0x100000000000000000000000000000000000000000000000000000000 SWAP1
DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x87DB03B7 EQ PUSH1 0x44 JUMPI JUMPDEST PUSH1 0x0 DUP1
REVERT JUMPDEST CALLVALUE DUP1 ISZERO PUSH1 0x4F JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP
PUSH1 0x6C PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1 DUP1 DUP1 CALLDATALOAD SWAP1
PUSH1 0x20 ADD SWAP1 SWAP3 SWAP2 SWAP1 POP POP POP PUSH1 0x6E JUMP JUMPDEST STOP
JUMPDEST DUP1 PUSH1 0x0 SLOAD ADD PUSH1 0x0 DUP2 SWAP1 SSTORE POP POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 CALLVALUE 0xcd 0xe3 MULMOD SWAP5 0x4b SSTORE 0xc8 SELFDESTRUCT
0x48 0xaa 0x2a 0xc3 0x4f LOG2 SWAP3 0xce 0x46 DUP3 0x22 CALL 0xbe SELFDESTRUCT SWAP8 0xc9 0xc9
GASLIMIT XOR PUSH11 0x49B4CD0029000000000000
```

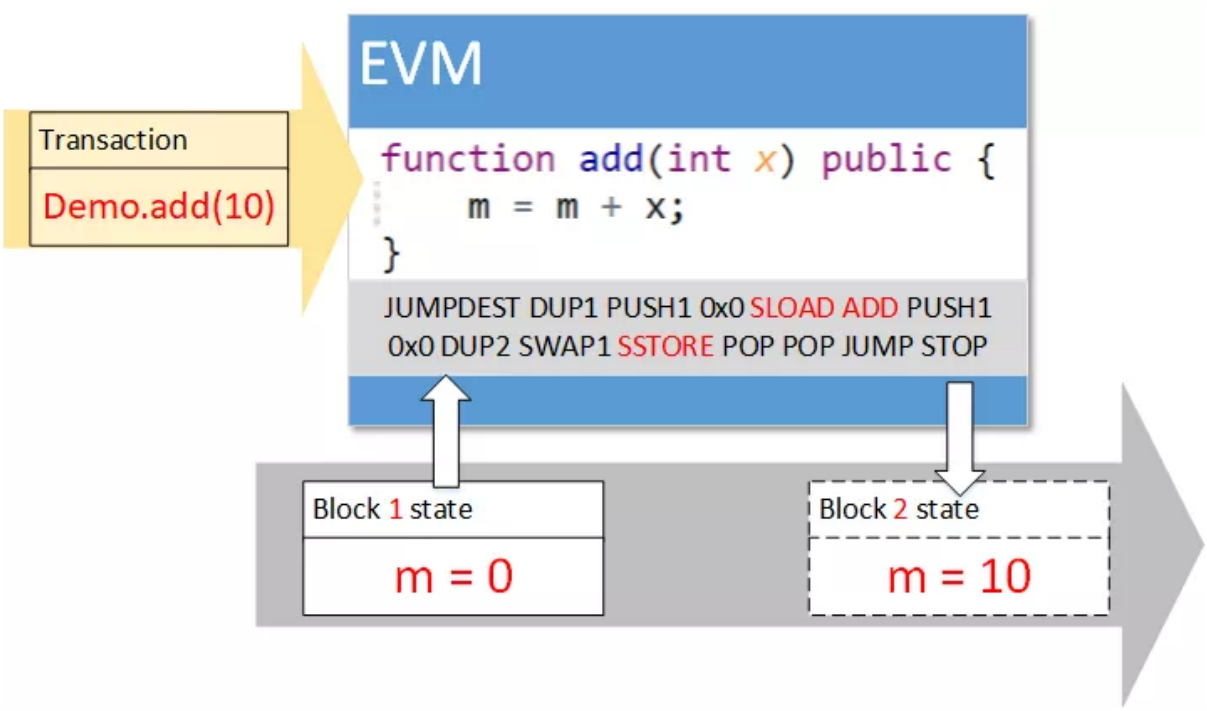
将add()函数的OPCODE的**标红部分**摘取出来，可看到其具体的实现思想与汇编代码相同，是一种基于堆栈式的操作。其中的SLOAD将区块链上指定位置的数据读入堆栈顶部，ADD实现将堆栈顶端的两个数据相加，SSTORE再将相加后放在堆栈顶部的结果写入区块链下一个区块的数据中，为下个区块的共识做准备。

```
JUMPDEST                    function add(int x) public {\r...
DUP1                        x
PUSH 0                      m
SLOAD                       m
ADD                         m + x
PUSH 0                      m
DUP2                        m = m + x
SWAP1                       m = m + x
SSTORE                      m = m + x
POP                         m = m + x
POP                         function add(int x) public {\r...
JUMP [out]                          function add(int x) public {\r...
```

在合约二进制被部署到区块链上后，通过发送交易调用合约里的方法。节点根据交易将合约代码装载入EVM中，并根据交易的传参执行合约上相应的函数add()。

EVM执行合约代码，从区块链上读入当前区块的数据，进行相加操作，并将结果写入下一个区块（等待共识的区块）对应的状态数据中。

此后，共识算法将待执行的区块共识落盘，区块高度增加，区块链上的数据完成更新。



上述步骤可见，Solidity的实现与当今已有的做法有着很多相似之处。编译，用的是传统的套路，将代码转换成虚拟机可执行的二进制；执行，也是与传统方式相同，借助堆栈作为缓冲区执行二进制代码。

Solidity 局限与改进

Solidity由于是第一个大规模应用的智能合约语言，存在着一些有待改进的地方。

Solidity不够灵活。Solidity语言受到自身堆栈深度的限制，函数传参和局部参数的个数总和不能超过16个。要实现一些比较复杂的函数难免有些鸡肋。Solidity是一种强类型的语言，但其类型转换较为麻烦。将整型转换成字符串时需要转换成二进制再拼接。在字符串的操作上，缺少一些方便的函数。

Solidity的性能较差。在执行上，OPCODE的执行是一种用程序模拟的汇编执行器，而不是直接使用CPU的资源。在存储上，Solidity的底层存储单位是32字节（256 bits），对硬盘的读写要求较高，浪费了大量的存储资源。

针对上述两点，FISCO BCOS提供了一种用C++写合约方式：预编译合约。开发者可以用C++编写智能合约逻辑，并将其内置在节点中。

预编译合约的调用方法与Solidity合约相同，通过合约地址即可直接调用。FISCO BCOS提供了参数解析，将调用的参数解析成C++可识别的格式。

预编译合约突破了Solidity语言的限制，借助强大的C++语言，可以灵活的实现各种逻辑，灵活性大大提高。同时，C++的性能优势也得到了很好的利用，通过预编译合约编写的逻辑，相比于Solidity语言来说，性能得到提升。

智能合约编写之Solidity的基础特性

作者：储雨知 | FISCO BCOS 核心开发者

如前篇介绍，目前大部分的联盟链平台，包括FISCO BCOS，都采用Solidity作为智能合约开发语言，因此熟悉并上手Solidity十分必要。作为一门面向区块链平台设计的图灵完备的编程语言，Solidity支持函数调用、修饰符、重载、事件、继承等多种特性，在区块链社区中，拥有广泛的影响力和踊跃的社区支持。但对于刚接触区块链的人而言，Solidity是一门陌生的语言。智能合约编写阶段将从Solidity基础特性、高级特性、设计模式以及编程攻略分别展开，带读者认识Solidity并掌握其运用，更好地进行智能合约开发。本篇将围绕Solidity的基础特性，带大家上手开发一个最基本的智能合约。

智能合约代码结构

任何编程语言都有其规范的代码结构，用于表达在一个代码文件中如何组织和编写代码，Solidity也一样。本节，我们将通过一个简单的合约示例，来了解智能合约的代码结构。

```
pragma solidity ^0.4.25;
contract Sample{

    //State variables
    address private _admin;
    uint private _state;

    //Modifier
    modifier onlyAdmin(){
        require(msg.sender == _admin, "You are not admin");
        _;
    }

    //Events
    event SetState(uint value);

    //Constructor
    constructor() public{
        _admin = msg.sender;
    }

    //Functions
    function setState(uint value) public onlyAdmin{
        _state = value;
        emit SetState(value);
    }

    function getValue() public view returns (uint){
        return _state;
    }

}
```

上面这段程序包括了以下功能:

- 通过构造函数来部署合约
- 通过setValue函数设置合约状态
- 通过getValue函数查询合约状态

整个合约主要分为以下几个构成部分:

- **状态变量** - _admin, _state，这些变量会被永久保存，也可以被函数修改
- **构造函数** - 用于部署并初始化合约
- **事件** - SetState, 功能类似日志，记录了一个事件的发生
- **修饰符** - onlyAdmin, 用于给函数加一层”外衣”
- **函数** - setState, getState，用于读写状态变量

下面将逐一介绍上述构成部分。

状态变量

状态变量是合约的骨髓，它记录了合约的业务信息。用户可以通过函数来修改这些状态变量，这些修改也会被包含到交易中；交易经过区块链网络确认后，修改即为生效。

```
uint private _state;
```

状态变量的声明方式为：[类型] [访问修饰符-可选] [字段名]

构造函数

构造函数用于初始化合约，它允许用户传入一些基本的数据，写入到状态变量中。在上述例子中，设置了_admin字段，作为后面演示其他功能的前提。

```
constructor() public{
    _admin = msg.sender;
}
```

和java不同的是，构造函数不支持重载，只能指定一个构造函数。

函数

函数被用来读写状态变量。对变量的修改将会被包含在交易中，经区块链网络确认后才生效。生效后，修改会被永久的保存在区块链账本中。函数签名定义了函数名、输入输出参数、访问修饰符、自定义修饰符。

```
function setState(uint value) public onlyAdmin;
```

函数还可以返回多个返回值：

```
function functionSample() public view returns(uint, uint){
    return (1,2);
}
```

在本合约中，还有一个配备了view修饰符的函数。这个view表示了该函数不会修改任何状态变量。与view类似的还有修饰符pure，其表明该函数是纯函数，连状态变量都不用读，函数的运行仅仅依赖于参数。

```
function add(uint a, uint b) public pure returns(uint){
    return a+b;
}
```

如果在view函数中尝试修改状态变量，或者在pure函数中访问状态变量，编译器均会报错。

事件

事件类似于日志，会被记录到区块链中，客户端可以通过web3订阅这些事件。

定义事件

```
event SetState(uint value);
```

构造事件

```
emit SetState(value);
```

这里有几点需要注意：

- 事件的名称可以任意指定，不一定要和函数名挂钩，但推荐两者挂钩，以便清晰地表达发生的事情.
- 构造事件时，也可不写emit，但因为事件和函数无论是名称还是参数都高度相关，这样操作很容易笔误将事件写成函数调用，因此不推荐。

```
function setState(uint value) public onlyAdmin{
    _state = value;
    //emit SetState(value);
    //这样写也可以，但不推荐，因为很容易笔误写成setState
    SetState(value);
}
```

- Solidity编程风格应采用一定的规范。查看编程风格

修饰符

修饰符是合约中非常重要的一环。它挂在函数声明上，为函数提供一些额外的功能，例如检查、清理等工作。在本例中，修饰符onlyAdmin要求函数调用前，需要先检测函数的调用者是否为函数部署时设定的那个管理员(即合约的部署人)。

```
//Modifer
modifier onlyAdmin(){
    require(msg.sender == _admin, "You are not admin");
    _;
}

...
//Functions
function setState(uint value) public onlyAdmin{
    ...
}
```

值得注意的是，定义在修饰符中的下划线“_”，表示函数的调用，指代的是开发者用修饰符修饰的函数。在本例中，表达的是setState函数调用的意思。

智能合约的运行

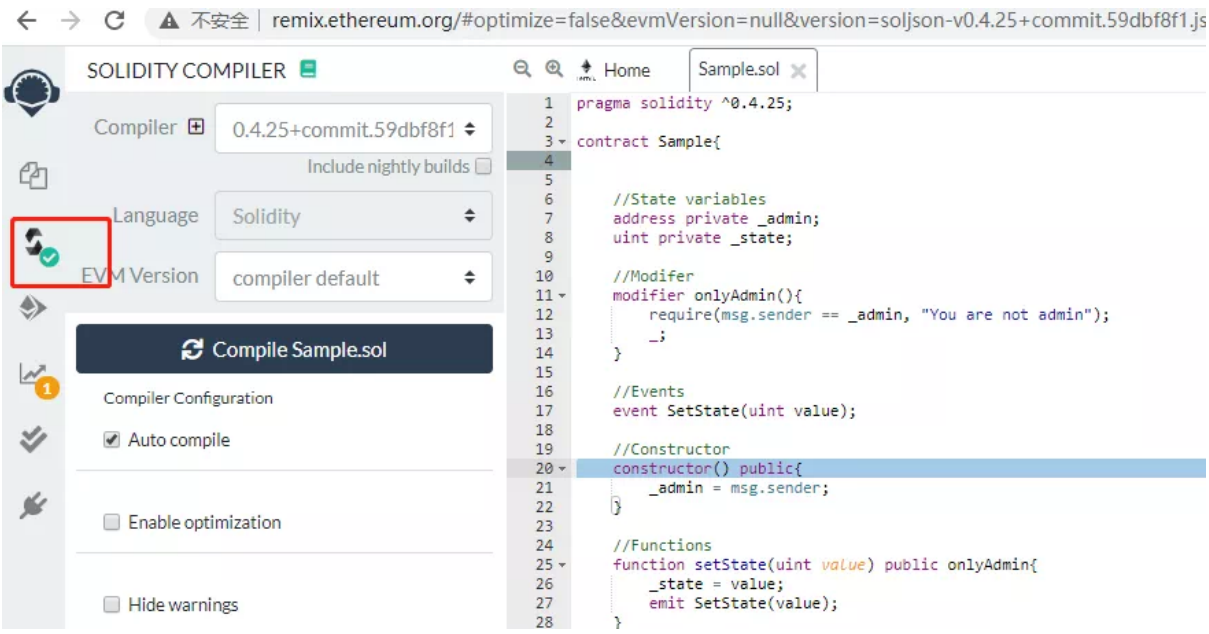
了解了上述的智能合约示例的结构，就可以直接上手运行，运行合约的方式有多种，大家可以任意采取其中一种:

- 方法一：可以使用FISCO BCOS控制台的方式来部署合约
- 方法二：使用FISCO BCOS开源项目WeBASE提供的在线ide WEBASE-front运行
- 方法三：通过在线ide remix来进行合约的部署与运行, remix的地址

本例中使用remix作为运行示例。

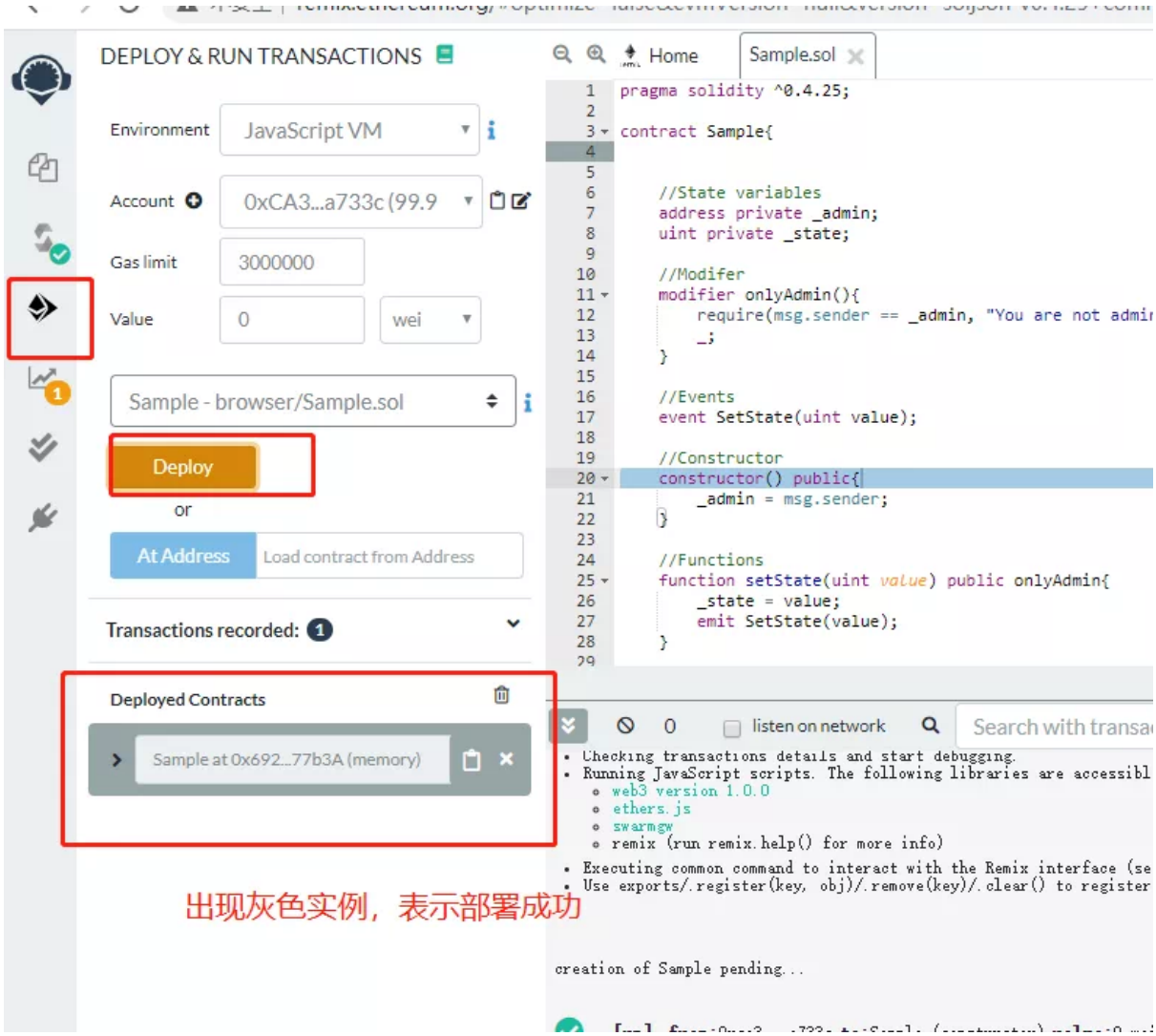
编译

首先，在remix的文件ide中键入代码后，通过编译按钮来编译。成功后会在按钮上出现一个绿色对勾:



部署

编译成功后就可进行部署环节，部署成功后会出现合约实例。



setState

合约部署后，我们来调用setState(4)。在执行成功后，会产生一条交易收据，里面包含了交易的执行信息。

在这里，用户可以看到交易执行状态(status)、交易执行人(from)、交易输入输出(decoded input, decoded output)、交易开销(execution cost)以及交易日志(logs)。在logs中，我们看到SetState事件被抛出，里面的参数也记录了事件传入的值4。如果我们换一个账户来执行，那么调用会失败，因为onlyAdmin修饰符会阻止用户调用。

getState

调用getState后，可以直接看到所得到的值为4，正好是我们先前setState所传入的值：

Solidity数据类型

在前文的示例中，我们用到了uint等数据类型。由于Solidity类型设计比较特殊，这里也会简单介绍一下Solidity的数据类型。

整型系列

Solidity提供了一组数据类型来表示整数, 包含无符号整数与有符号整数。每类整数还可根据长度细分，具体细分类型如下。

定长bytes系列

Solidity提供了bytes1到bytes32的类型，它们是固定长度的字节数组。用户可以读取定长bytes的内容。

```
    function bytesSample() public{

        bytes32 barray;
        //Initialize baarray
        //read brray[0]
        byte b = barray[0];
    }
```

并且，可以将整数类型转换为bytes。

```
    uint256 s = 1;
    bytes32 b = bytes32(s);
```

这里有一个关键细节，Solidity采取大端序编码，高地址存的是整数的小端。例如，b[0]是低地址端，它存整数的高端，所以值为0；取b[31]才是1。

```
    function bytesSample() public pure returns(byte, byte){

        uint256 value = 1;
        bytes32 b = bytes32(value);
        //Should be (0, 1)
        return (b[0], b[31]);
    }
```

变长bytes

从上文中，读者可了解定长byte数组。此外，Solidity还提供了一个变长byte数组：bytes。使用方式类似数组，后文会有介绍。

string

Solidity提供的string，本质是一串经UTF-8编码的字节数组，它兼容于变长bytes类型。目前Solidity对string的支持不佳，也没有字符的概念。用户可以将string转成bytes。

```
    function stringSample() public view returns(bytes){
        string memory str = "abc";
        bytes memory b = bytes(str);
        //0x616263
        return b;
    }
```

要注意的是，当将string转换成bytes时，数据内容本身不会被拷贝，如上文中，str和b变量指向的都是同一个字符串abc。

address

address表示账户地址，它由私钥间接生成，是一个20字节的数据。同样，它也可以被转换为bytes20。

```
    function addressSample() public view returns(bytes20){

        address me = msg.sender;
        bytes20 b = bytes20(me);
        return b;
    }
```

mapping

mapping表示映射, 是极其重要的数据结构。它与java中的映射存在如下几点差别：

- 它无法迭代keys，因为它只保存键的哈希，而不保存键值，如果想迭代，可以用开源的可迭代哈希类库
- 如果一个key未被保存在mapping中，一样可以正常读取到对应value，只是value是空值（字节全为0）。所以它也不需要put、get等操作，用户直接去操作它即可。

```
contract Sample{

    mapping(uint=>string) private values;

    function mappingSample() public view returns(bytes20){
        //put a key value pair
        values[10] = "hello";

        //read value
        string value = values[10];

    }

}
```

数组

如果数组是状态变量，那么支持push等操作：

```
contract Sample{

    string[] private arr;

    function arraySample() public view {
        arr.push("Hello");
        uint len = arr.length;//should be 1
        string value = arr[0];//should be Hello
      }

}
```

数组也可以以局部变量的方式使用，但稍有不同：

```
function arraySample() public view returns(uint){
        //create an empty array of length 2
        uint[] memory p = new uint[](2);
        p[3] = 1;//THIS WILL THROW EXCEPTION
        return p.length;
}
```

struct

Solidity允许开发者自定义结构对象。结构体既可以作为状态变量存储，也可以在函数中作为局部变量存在。

```
struct Person{
        uint age;
        string name;
    }

    Person private _person;

    function structExample() {
        Person memory p = Person(1, "alice");
        _person = p;
    }
```

本节中只介绍了比较常见的数据类型，更完整的列表可参考Solidity官方网站。

全局变量

示例合约代码的构造函数中，包含msg.sender。它属于全局变量。在智能合约中，全局变量或全局方法可用于获取和当前区块、交易相关的一些基本信息，如块高、块时间、合约调用者等。

比较常用的全局变量是msg变量，表示调用上下文，常见的全局变量有以下几种：

- **msg.sender**：合约的直接调用者。

  由于是直接调用者，所以当处于 用户A->合约1->合约2 调用链下，若在合约2内使用msg.sender，得到的会是合约1的地址。如果想获取用户A，可以用tx.origin.

- **tx.origin**：交易的"始作俑者"，整个调用链的起点。
- **msg.calldata**：包含完整的调用信息，包括函数标识、参数等。calldata的前4字节就是函数标识，与msg.sig相同。
- **msg.sig**：msg.calldata的前4字节，用于标识函数。
- **block.number**：表示当前所在的区块高度。
- **now**：表示当前的时间戳。也可以用block.timestamp表示。

这里只列出了部分常见全局变量，请参考完整版本。

结语

本文以一个简单的示例合约作为引入，介绍了运用Solidity开发智能合约的基本知识。读者可以尝试运行该合约，感受智能合约的开发。若想更深入学习智能合约示例，推荐官方网站示例供读者学习，也可关注本专题后续系列文章。在官网的示例中，提供了投票、竞拍、微支付通道等多个案例，这些案例贴近实际生活，是很好的学习资料。

智能合约编写之Solidity的高级特性

作者：毛嘉宇 | FISCO BCOS 核心开发者

前言

FISCO BCOS使用了Solidity语言进行智能合约开发。Solidity是一门面向区块链平台设计、图灵完备的编程语言，支持函数调用、修饰器、重载，事件、继承和库等多种高级语言的特性。在本系列前两篇文章

中，介绍了智能合约的概念与Solidity的基础特性。本文将介绍Solidity的一些高级特性，帮助读者快速入门，编写高质量、可复用的Solidity代码。

合理控制函数和变量的类型

基于最少知道原则（Least Knowledge Principle）中经典面向对象编程原则，一个对象应该对其他对象保持最少的了解。优秀的Solidity编程实践也应符合这一原则：每个合约都清晰、合理地定义函数的可见性，暴露最少的信息给外部，做好对内部函数可见性的管理。同时，正确地修饰函数和变量的类型，可给合约内部数据提供不同级别的保护，以防止程序中非预期的操作导致数据产生错误；还能提升代码的可读性与质量，减少误解和bug；更有利于优化合约执行的成本，提升链上资源的使用效率。

守住函数操作的大门：函数可见性

Solidity有两种函数调用方式：

- 内部调用：又被称为『消息调用』。常见的有合约内部函数、父合约的函数以及库函数的调用。（例如，假设A合约中存在f函数，则在A合约内部，其他函数调用f函数的调用方式为f()。）
- 外部调用：又被称为『EVM调用』。一般为跨合约的函数调用。在同一合约内部，也可以产生外部调用。（例如，假设A合约中存在f函数，则在B合约内可通过使用A.f()调用。在A合约内部，可以用this.f()来调用。）。

函数可以被指定为 external ， public ， internal 或者 private标识符来修饰。

基于以上表格，我们可以得出函数的可见性 public > external > internal > private。另外，如果函数不使用上述类型标识符，那么默认情况下函数类型为 public。

综上所述，我们可以总结一下以上标识符的不同使用场景：

- public，公有函数，系统默认。通常用于修饰**可对外暴露的函数，且该函数可能同时被内部调用。**
- external，外部函数，推荐**只向外部暴露**的函数使用。当函数的某个参数非常大时，如果显式地将函数标记为external，可以强制将函数存储的位置设置为calldata，这会节约函数执行时所需存储或计算资源。
- internal，内部函数，推荐所有合约内**不对合约外暴露**的函数使用，可以避免因权限暴露被攻击的风险。
- private，私有函数，在极少数严格保护合约函数**不对合约外部开放且不可被继承**的场景下使用。

不过，需要注意的是，无论用何种标识符，即使是private，整个函数执行的过程和数据是对所有节点可见，其他节点可以验证和重放任意的历史函数。实际上，整个智能合约所有的数据对区块链的参与节点来说都是透明的。

刚接触区块链的用户常会误解，在区块链上可以通过权限控制操作来控制和保护上链数据的隐私。这是一种错误的观点。事实上，在区块链业务数据未做特殊加密的前提下，区块链同一账本内的所有数据经过共识后落盘到所有节点上，链上数据是全局公开且相同的，智能合约只能控制和保护合约数据的执行权限。如何正确地选择函数修饰符是合约编程实践中的『必修课』，只有掌握此节真谛方可自如地控制合约函数访问权限，提升合约安全性。

对外暴露最少的必要信息：变量的可见性

与函数一样，对于状态变量，也需要注意可见性修饰符。状态变量的修饰符默认是internal，不能设置为external。此外，当状态变量被修饰为public，编译器会生成一个与该状态变量同名的函数。具体可参考以下示例：

```
pragma solidity ^0.4.0;

contract TestContract {
```

(continues on next page)

(续上页)

```
    uint public year = 2020;
}

contract Caller {
    TestContract c = new TestContract();
    function f() public {
        uint local = c.year();
        //expected to be 2020
    }
}
```

这个机制有点像Java语言里lombok库所提供的@Getter注解，默认为一个POJO类变量生成get函数，大大简化了某些合约代码的书写。同样，变量的可见性也需要被合理地修饰，不该公开的变量果断用private修饰，使合约代码更符合『最少知道』的设计原则。

精确地将函数分类：函数的类型

函数可以被声明为pure、view，两者的作用可见下图。

那么，什么是读取或修改状态呢？简单来说，两个状态就是读取或修改了账本相关的数据。

在FISCO BCOS中，读取状态可能是：

1. 读取状态变量。
2. 访问 block，tx， msg 中任意成员 （除 msg.sig 和 msg.data 之外）。
3. 调用任何未标记为 pure 的函数。
4. 使用包含某些操作码的内联汇编。

而修改状态可能是：

1. 修改状态变量。
2. 产生事件。
3. 创建其它合约。
4. 使用 selfdestruct。
5. 调用任何没有标记为 view 或者 pure 的函数。
6. 使用底层调用。
7. 使用包含特定操作码的内联汇编。

需要注意的是，在某些版本编译器中，并没有对这两个关键字进行强制的语法检查。推荐尽可能使用pure和view来声明函数，例如将没有读取或修改任何状态的库函数声明为pure，这样既提升了代码可读性，也使其更赏心悦目，何乐而不为?

编译时就确定的值：状态常量

所谓的状态常量是指被声明为constant的状态变量。一旦某个状态变量被声明为constant，那么该变量值只能为编译时确定的值，无法被修改。编译器一般会在编译状态计算出此变量实际值，不会给变量预留储存空间。所以，constant只支持修饰值类型和字符串。状态常量一般用于定义含义明确的业务常量值。

面向切片编程：函数修饰器（modifier）

Solidity提供了强大的改变函数行为的语法：函数修饰器（modifier）。一旦某个函数加上了修饰器，修饰器内定义的代码就可以作为该函数的装饰被执行，类似其他高级语言中装饰器的概念。这样说起来很抽象，让我们来看一个具体的例子：

```
pragma solidity ^0.4.11;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;

    // 修饰器所修饰的函数体会被插入到特殊符号 _; 的位置。
    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    // 使用onlyOwner修饰器所修饰，执行changeOwner函数前需要首先执行onlyOwner"_;"前的语句。
    function changeOwner(address _owner) public onlyOwner {
        owner = _owner;
    }
}
```

如上所示，定义onlyOwner修饰器后，在修饰器内，require语句要求msg.sender必须等于owner。后面的"_;"表示所修饰函数中的代码。

所以，代码实际执行顺序变成了：

1. 执行onlyOwner修饰器的语句，先执行require语句。（执行第9行）
2. 执行changeOwner函数的语句。（执行第15行）

由于changeOwner函数加上了onlyOwner的修饰，故只有当msg.sender是owner才能成功调用此函数，否则会报错回滚。同时，修饰器还能传入参数，例如上述的修饰器也可写成：

```
modifier onlyOwner(address sender) {
    require(sender == owner);
    _;
}

function changeOwner(address _owner) public onlyOwner(msg.sender) {
        owner = _owner;
}
```

同一个函数可有多个修饰器，中间以空格间隔，修饰器依次检查执行。此外，修饰器还可以被继承和重写。由于其所提供的强大功能，修饰器也常被用来实现权限控制、输入检查、日志记录等。比如，我们可以定义一个跟踪函数执行的修饰器：

```
event LogStartMethod();
event LogEndMethod();

modifier logMethod {
    emit LogStartMethod();
    _;
    emit LogEndMethod();
}
```

这样，任何用logMethod修饰器来修饰的函数都可记录其函数执行前后的日志，实现日志环绕效果。如果你已经习惯了使用Spring框架的AOP，也可以试试用modifier实现一个简单的AOP功能。

modifier最常见的打开方式是通过提供函数的校验器。在实践中，合约代码的一些检查语句常会被抽象并定义为一个modifier，如上述例子中的onlyOwner就是个最经典的权限校验器。这样一来，连检查的逻辑也能被快速复用，用户也不用再为智能合约里到处都是参数检查或其他校验类代码而苦恼。

可以debug的日志：合约里的事件（Event）

介绍完函数和变量，我们来聊聊Solidity其中一个较为独有的高级特性——事件机制。

事件允许我们方便地使用 EVM 的日志基础设施，而Solidity的事件有以下作用：

1. 记录事件定义的参数，存储到区块链交易的日志中，提供廉价的存储。
2. 提供一种回调机制，在事件执行成功后，由节点向注册监听的SDK发送回调通知，触发回调函数被执行。
3. 提供一个过滤器，支持参数的检索和过滤。

事件的使用方法非常简单，两步即可玩转。

- 第一步，使用关键字『event』来定义一个事件。建议事件的命名以特定前缀开始或以特定后缀结束，这样更便于和函数区分，在本文中我们将统一以『Log』前缀来命名事件。下面，我们用『event』来定义一个函数调用跟踪的事件：

```
event LogCallTrace(address indexed from, address indexed to, bool result);
```

事件在合约中可被继承。当他们被调用时，会将参数存储到交易的日志中。这些日志被保存到区块链中，与地址相关联。在上述例子中，用indexed标记参数被搜索，否则，这些参数被存储到日志的数据中，无法被搜索。

- 第二步，在对应的函数内触发定义事件。调用事件的时候，在事件名前加上『emit』关键字：

```
function f() public {
    emit LogCallTrace(msg.sender, this, true);
}
```

这样，当函数体被执行的时候，会触发执行LogCallTrace。

最后，在FISCO BCOS的Java SDK中，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。更多细节可以参考合约事件推送功能文档。在SDK中，可以根据事件的indexed属性，根据特定值进行搜索。合约事件推送功能文档：

不过，日志和事件无法被直接访问，甚至在创建的合约中也无法被直接访问。但好消息是日志的定义和声明非常利于在『事后』进行追溯和导出。例如，我们可以在合约的编写中，定义和埋入足够的事件，通过WeBASE的数据导出子系统我们可以将所有日志导出到MySQL等数据库中。这特别适用于生成对账文件、生成报表、复杂业务的OLTP查询等场景。此外，WeBASE提供了一个专用的代码生成子系统帮助分析具体的业务合约，自动生成相应的代码。

- WeBASE的数据导出子系统
- 代码生成子系统

在Solidity中，事件是一个非常有用的机制，如果说智能合约开发最大的难点是debug，那善用事件机制可以让你快速制伏Solidity开发。

面向对象之重载

重载是指合约具有多个不同参数的同名函数。对于调用者来说，可使用相同函数名来调用功能相同，但参数不同的多个函数。在某些场景下，这种操作可使代码更清晰、易于理解，相信有一定编程经验的读者对此一定深有体会。下面将展示一个典型的重载语法：

```
pragma solidity ^0.4.25;

contract Test {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }
```

(continues on next page)

(续上页)

```
    function f(uint _in, bytes32 _key) public pure returns (uint out) {
        out = 2;
    }
}
```

需要注意的是，每个合约只有一个构造函数，这也意味着合约的构造函数是不支持重载的。我们可以想像一个没有重载的世界，程序员一定绞尽脑汁、想方设法给函数起名，大家的头发可能又要多掉几根。

面向对象之继承

Solidity使用『is』作为继承关键字。因此，以下这段代码表示的是，合约B继承了合约A：

```
pragma solidity ^0.4.25;

contract A {
}

contract B is A {
}
```

而继承的合约B可以访问被继承合约A的所有非private函数和状态变量。在Solidity中，继承的底层实现原理为：当一个合约从多个合约继承时，在区块链上只有一个合约被创建，所有基类合约的代码被复制到创建的合约中。相比于C++或Java等语言的继承机制，Solidity的继承机制有点类似于Python，支持多重继承机制。因此，Solidity中可以使用一个合约来继承多个合约。在某些高级语言中，比如Java，出于安全性和可靠性的考虑，只支持单重继承，通过使用接口机制来实现多重继承。对于大多数场景而言，单继承的机制就可以满足需求了。多继承会带来很多复杂的技术问题，例如所谓的『钻石继承』等，建议在实践中尽可能规避复杂的多继承。继承简化了人们对抽象合约模型的认识和描述，清晰体现了相关合约间的层次结构关系，并且提供软件复用功能。这样，能避免代码和数据冗余，增加程序的重用性。

面向对象之抽象类和接口

根据依赖倒置原则，智能合约应该尽可能地面向接口编程，而不依赖具体实现细节。Solidity支持抽象合约和接口的机制。如果一个合约，存在未实现的方法，那么它就是抽象合约。例如：

```
pragma solidity ^0.4.25;

contract Vehicle {
    //抽象方法
    function brand() public returns (bytes32);
}
```

抽象合约无法被成功编译，但可以被继承。接口使用关键字interface，上面的抽象也可以被定义为一个接口。

```
pragma solidity ^0.4.25;

interface Vehicle {
    //抽象方法
    function brand() public returns (bytes32);
}
```

接口类似于抽象合约，但不能实现任何函数，同时，还有进一步的限制：

1. 无法继承其他合约或接口。
2. 无法定义构造函数。
3. 无法定义变量。

4. 无法定义结构体
5. 无法定义枚举。

合适地使用接口或抽象合约有助于增强合约设计的可扩展性。但是，由于区块链EVM上计算和存储资源的限制，切忌过度设计，这也是从高级语言技术栈转到Solidity开发的老司机常常会陷入的天坑。

避免重复造轮子：库(Library)

在软件开发中，很多经典原则可以提升软件的质量，其中最为经典的就是尽可能复用久经考验、反复打磨、严格测试的高质量代码。此外，复用成熟的库代码还可以提升代码的可读性、可维护性，甚至是可扩展性。

和所有主流语言一样，Solidity也提供了库（Library）的机制。Solidity的库有以下基本特点：

1. 用户可以像使用合约一样使用关键词library来创建合约。
2. 库既不能继承也不能被继承。
3. 库的internal函数对调用者都是可见的。
4. 库是无状态的，无法定义状态变量，但是可以访问和修改调用合约所明确提供的状态变量。

接下来，我们来看一个简单的例子，以下是FISCO BCOS社区中一个LibSafeMath的代码库。我们对此进行了精简，只保留了加法的功能：

```
pragma solidity ^0.4.25;

library LibSafeMath {
  /**
  * @dev Adds two numbers, throws on overflow.
  */
  function add(uint256 a, uint256 b) internal returns (uint256 c) {
    c = a + b;
    assert(c >= a);
    return c;
  }
}
```

我们只需在合约中import库的文件，然后使用L.f()的方式来调用函数，（例如LibSafeMath.add(a,b)）。接下来，我们编写调用这个库的测试合约，合约内容如下：

```
pragma solidity ^0.4.25;

import "./LibSafeMath.sol";

contract TestAdd {

  function testAdd(uint256 a, uint256 b) external returns (uint256 c) {
    c = LibSafeMath.add(a,b);
  }
}
```

在FISCO BCOS控制台中，我们可以测试合约的结果（控制台的介绍文章详见FISCO BCOS 控制台详解，飞一般的区块链体验），运行结果如下：

```
=================================================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______   ______   ______      ________   ______   ______   ______
|        |      \/      \ /      \ /      \     |        \ /      \ /      \ /      ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
```

(continues on next page)

(续上页)

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

```
================================================================================================
[group:1]> deploy TestAdd
contract address: 0xe2af1fd7ecd91eb7e0b16b5c754515b775b25fd2

[group:1]> call TestAdd 0xe2af1fd7ecd91eb7e0b16b5c754515b775b25fd2 testAdd 2000 20
transaction hash:␣
↪0x136ce66603aa6e7fd9e4750fcf25302b13171abba8c6b2109e6dd28111777d54
------------------------------------------------------------------------------------------
↪----------
Output
function: testAdd(uint256,uint256)
return type: (uint256)
return value: (2020)
------------------------------------------------------------------------------------------
↪----------

[group:1]>
```

通过以上示例，我们可清晰了解在Solidity中应如何使用库。类似Python，在某些场景下，指令『using A for B;』可用于附加库函数（从库 A）到任何类型（B）。这些函数将接收到调用它们的对象作为第一个参数（像 Python 的 self 变量）。这个功能使库的使用更加简单、直观。

例如，我们对代码进行如下简单修改：

```
pragma solidity ^0.4.25;

import "./LibSafeMath.sol";

contract TestAdd {
  // 添加using ... for ... 语句，库 LibSafeMath 中的函数被附加在uint256的类型上
  using LibSafeMath for uint256;

  function testAdd(uint256 a, uint256 b) external returns (uint256 c) {
        //c = LibSafeMath.add(a,b);
        c = a.add(b);
        //对象a直接被作为add方法的首个参数传入。
  }
}
```

验证一下结果依然是正确的。

```
================================================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 _________ _______  _______  _______  _______       ________  _______  _______  _______
|         |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
```

(continues on next page)

(续上页)

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

================================================================================================
[group:1]> deploy TestAdd
contract address: 0xf82c19709a9057d8e32c19c23e891b29b708c01a

[group:1]> call TestAdd 0xf82c19709a9057d8e32c19c23e891b29b708c01a testAdd 2000 20
transaction hash:␣
↪0xcc44a80784404831d8522dde2a8855606924696957503491eb47174c9dbf5793
-------------------------------------------------------------------------------------------
↪----------
Output
function: testAdd(uint256,uint256)
return type: (uint256)
return value: (2020)
-------------------------------------------------------------------------------------------
↪----------

[group:1]>
```

更好地使用Solidity library有助于开发者更好地复用代码。除了Solidity社区提供的大量开源、高质量的代码库外，FISCO BCOS社区也计划推出全新的Solidity代码库，开放给社区用户，敬请期待。当然，你也可以自己动手，编写可复用的代码库组件，并分享到社区。

总结

本文介绍了Solidity合约编写的若干高级语法特性，旨在抛砖引玉，帮助读者快速沉浸到Solidity编程世界。编写高质量、可复用的Solidity代码的诀窍在于：多看社区优秀的代码，多动手实践编码，多总结并不断进化。期待更多朋友在社区里分享Solidity的宝贵经验和精彩故事，have fun :)

智能合约编写之 Solidity的设计模式

作者：储雨知 | FISCO BCOS 核心开发者

前言

随着区块链技术发展，越来越多的企业与个人开始将区块链与自身业务相结合。区块链所具有的独特优势，例如，数据公开透明、不可篡改，可以为业务带来便利。但与此同时，也存在一些隐患。数据的公开透明，意味着任何人都可以读取；不可篡改，意味着信息一旦上链就无法删除，甚至合约代码都无法被更改。除此之外，合约的公开性、回调机制，每一个特点都可被利用，作为攻击手法，稍有不慎，轻则合约形同虚设，重则要面临企业机密泄露的风险。所以，在业务合约上链前，需要预先对合约的安全性、可维护性等方面作充分考虑。幸运的是，通过近些年Solidity语言的大量实践，开发者们不断提炼和总结，已经形成了一些"设计模式"，来指导应对日常开发常见的问题。

智能合约设计模式概述

2019年，IEEE收录了维也纳大学一篇题为《Design Patterns For Smart Contracts In the Ethereum Ecosystem》的论文。这篇论文分析了那些火热的Solidity开源项目，结合以往的研究成果，整理出了18种设计模式。这些设计模式涵盖了安全性、可维护性、生命周期管理、鉴权等多个方面。

接下来，本文将从这18种设计模式中选择最为通用常见的进行介绍，这些设计模式在实际开发经历中得到了大量检验。

安全性(Security)

智能合约编写，首要考虑的就是安全性问题。在区块链世界中，恶意代码数不胜数。如果你的合约包含了跨合约调用，就要特别当心，要确认外部调用是否可信，尤其当其逻辑不为你所掌控的时候。如果缺乏防人之心，那些"居心叵测"的外部代码就可能将你的合约破坏殆尽。比如，外部调用可通过恶意回调，使代码被反复执行，从而破坏合约状态，这种攻击手法就是著名的Reentrance Attack（重入攻击）。这里，先引入一个重入攻击的小实验，以便让读者了解为什么外部调用可能导致合约被破坏，同时帮助更好地理解即将介绍的两种提升合约安全性的设计模式。

关于重入攻击，这里举个精简的例子。AddService合约是一个简单的计数器，每个外部合约可以调用AddService合约的addByOne来将字段_count加一，同时通过require来强制要求每个外部合约最多只能调用一次该函数。这样，_count字段就精确的反应出AddService被多少合约调用过。在addByOne函数的末尾，AddService会调用外部合约的回调函数notify。AddService的代码如下：

```
contract AddService{

    uint private _count;
    mapping(address=>bool) private _adders;

    function addByOne() public {
        //强制要求每个地址只能调用一次
        require(_adders[msg.sender] == false, "You have added already");
        //计数
        _count++;
        //调用账户的回调函数
        AdderInterface adder = AdderInterface(msg.sender);
        adder.notify();
        //将地址加入已调用集合
        _adders[msg.sender] = true;
    }
}

contract AdderInterface{
    function notify() public;
}
```

如果AddService如此部署，恶意攻击者可以轻易控制AddService中的_count数目，使该计数器完全失效。攻击者只需要部署一个合约BadAdder，就可通过它来调用AddService，就可以达到攻击效果。BadAdder合约如下：

```
contract BadAdder is AdderInterface{

    AddService private _addService = //...;
    uint private _calls;

    //回调
    function notify() public{
        if(_calls > 5){
            return;
        }
```

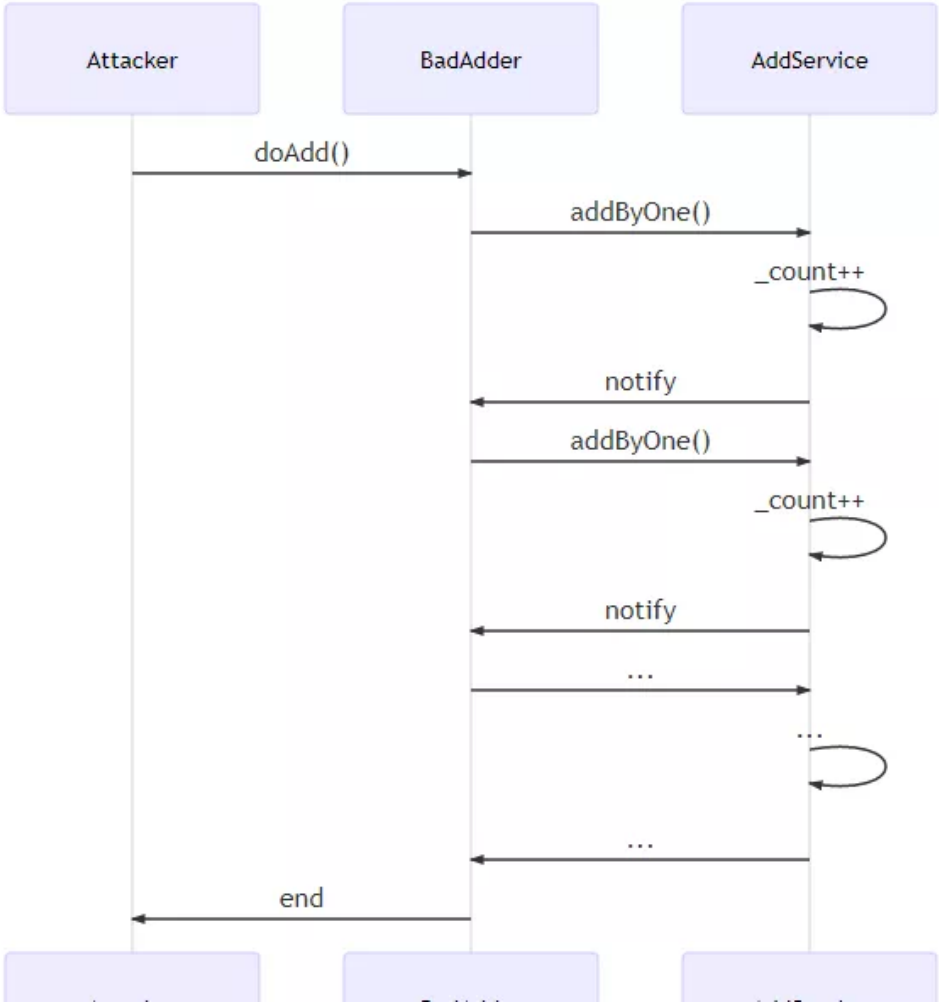
(continues on next page)

(续上页)

```
        _calls++;
        //Attention !!!!!!
        _addService.addByOne();
    }

    function doAdd() public{
        _addService.addByOne();
    }
}
```

BadAdder在回调函数notify中，反过来继续调用AddService，由于AddService糟糕的代码设计，require条件检测语句被轻松绕过，攻击者可以直击_count字段，使其被任意地重复添加。攻击过程的时序图如下：



在这个例子中，AddService难以获知调用者的回调逻辑，但依然轻信了这个外部调用，而攻击者利用了AddService糟糕的代码编排，导致悲剧的发生。本例子中去除了实际的业务意义，攻击后果仅仅是_count值失真。真正的重入攻击，可对业务造成严重后果。比如在统计投票数目是，投票数会被改得面目全非。打铁还需自身硬，如果想屏蔽这类攻击，合约需要遵循良好的编码模式，下面将介绍两个可有效解除此类攻击的设计模式。

Checks-Effects-Interaction - 保证状态完整，再做外部调用

该模式是编码风格约束，可有效避免重放攻击。通常情况下，一个函数可能包含三个部分：

- Checks：参数验证

- Effects：修改合约状态
- Interaction：外部交互

这个模式要求合约按照Checks-Effects-Interaction的顺序来组织代码。它的好处在于进行外部调用之前，Checks-Effects已完成合约自身状态所有相关工作，使得状态完整、逻辑自洽，这样外部调用就无法利用不完整的状态进行攻击了。回顾前文的AddService合约，并没有遵循这个规则，在自身状态没有更新完的情况下去调用了外部代码，外部代码自然可以横插一刀，让_adders[msg.sender]=true永久不被调用，从而使require语句失效。我们以checks-effects-interaction的角度审阅原来的代码：

```
    //Checks
    require(_adders[msg.sender] == false, "You have added already");
    //Effects
    _count++;
    //Interaction
    AdderInterface adder = AdderInterface(msg.sender);
    adder.notify();
    //Effects
    _adders[msg.sender] = true;
```

只要稍微调整顺序，满足Checks-Effects-Interaction模式，悲剧就得以避免：

```
    //Checks
    require(_adders[msg.sender] == false, "You have added already");
    //Effects
    _count++;
    _adders[msg.sender] = true;
    //Interaction
    AdderInterface adder = AdderInterface(msg.sender);
    adder.notify();
```

由于_adders映射已经修改完毕，当恶意攻击者想递归地调用addByOne，require这道防线就会起到作用，将恶意调用拦截在外。虽然该模式并非解决重入攻击的唯一方式，但依然推荐开发者遵循。

Mutex - 禁止递归

Mutex模式也是解决重入攻击的有效方式。它通过提供一个简单的修饰符来防止函数被递归调用：

```
contract Mutex {
    bool locked;
    modifier noReentrancy() {
        //防止递归
        require(!locked, "Reentrancy detected");
        locked = true;
        _;
        locked = false;
    }

    //调用该函数将会抛出Reentrancy detected错误
    function some() public noReentrancy{
        some();
    }
}
```

在这个例子中，调用some函数前会先运行noReentrancy修饰符，将locked变量赋值为true。如果此时又递归地调用了some，修饰符的逻辑会再次激活，由于此时的locked属性已为true，修饰符的第一行代码会抛出错误。

可维护性（Maintenance）

在区块链中，合约一旦部署，就无法更改。当合约出现了bug，通常要面对以下问题：

1. 合约上已有的业务数据怎么处理?
2. 怎么尽可能减少升级影响范围，让其余功能不受影响?
3. 依赖它的其他合约该怎么办?

回顾面向对象编程，其核心思想是将变化的事物和不变的事物相分离，以阻隔变化在系统中的传播。所以，设计良好的代码通常都组织得高度模块化、高内聚低耦合。利用这个经典的思想可解决上面的问题。

Data segregation - 数据与逻辑相分离

了解该设计模式之前，先看看下面这个合约代码：

```
contract Computer{

    uint private _data;

    function setData(uint data) public {
        _data = data;
    }

    function compute() public view returns(uint){
        return _data * 10;
    }
}
```

此合约包含两个能力，一个是存储数据(setData函数)，另一个是运用数据进行计算(compute函数）。如果合约部署一段时间后，发现compute写错了，比如不应是乘以10，而要乘以20，就会引出前文如何升级合约的问题。这时，可以部署一个新合约，并尝试将已有数据迁移到新的合约上，但这是一个很重的操作，一方面要编写迁移工具的代码，另一方面原先的数据完全作废，空占着宝贵的节点存储资源。

所以，预先在编程时进行模块化十分必要。如果我们将"数据"看成不变的事物，将"逻辑"看成可能改变的事物，就可以完美避开上述问题。Data Segregation（意为数据分离）模式很好地实现了这一想法。该模式要求一个业务合约和一个数据合约：数据合约只管数据存取，这部分是稳定的；而业务合约则通过数据合约来完成逻辑操作。

结合前面的例子，我们将数据读写操作专门转移到一个合约DataRepository中：

```
contract DataRepository{

    uint private _data;

    function setData(uint data) public {
        _data = data;
    }

    function getData() public view returns(uint){
        return _data;
    }
}
```

计算功能被单独放入一个业务合约中：

```
contract Computer{
    DataRepository private _dataRepository;
    constructor(address addr){
        _dataRepository =DataRepository(addr);
```

(continues on next page)

(续上页)

```
    }

    //业务代码
    function compute() public view returns(uint){
        return _dataRepository.getData() * 10;
    }
}
```

这样，只要数据合约是稳定的，业务合约的升级就很轻量化了。比如，当我要把Computer换成ComputerV2时，原先的数据依然可以被复用。

Satellite - 分解合约功能

一个复杂的合约通常由许多功能构成，如果这些功能全部耦合在一个合约中，当某一个功能需要更新时，就不得不去部署整个合约，正常的功能都会受到波及。Satellite模式运用单一职责原则解决上述问题，提倡将合约子功能放到子合约里，每个子合约（也称为卫星合约）只对应一个功能。当某个子功能需要修改，只要创建新的子合约，并将其地址更新到主合约里即可，其余功能不受影响。

举个简单的例子，下面这个合约的setVariable功能是将输入数据进行计算（compute函数），并将计算结果存入合约状态_variable：

```
contract Base {
    uint public _variable;

    function setVariable(uint data) public {
        _variable = compute(data);
    }

    //计算
    function compute(uint a) internal returns(uint){
        return a * 10;
    }
}
```

如果部署后，发现compute函数写错，希望乘以的系数是20，就要重新部署整个合约。但如果一开始按照Satellite模式操作，则只需部署相应的子合约。

首先，我们先将compute函数剥离到一个单独的卫星合约中去：

```
contract Satellite {
    function compute(uint a) public returns(uint){
        return a * 10;
    }
}
```

然后，主合约依赖该子合约完成setVariable：

```
contract Base {
    uint public _variable;

    function setVariable(uint data) public {
        _variable = _satellite.compute(data);
    }

     Satellite _satellite;
    //更新子合约（卫星合约）
    function updateSatellite(address addr) public {
        _satellite = Satellite(addr);
    }
}
```

这样，当我们需要修改compute函数时，只需部署这样一个新合约，并将它的地址传入到Base.updateSatellite即可：

```
contract Satellite2{
    function compute(uint a) public returns(uint){
        return a * 20;
    }
}
```

Contract Registry - 跟踪最新合约

在Satellite模式中，如果一个主合约依赖子合约，在子合约升级时，主合约需要更新对子合约的地址引用，这通过updateXXX来完成，例如前文的updateSatellite函数。这类接口属于维护性接口，与实际业务无关，过多暴露此类接口会影响主合约美观，让调用者的体验大打折扣。Contract Registry设计模式优雅地解决了这个问题。在该设计模式下，会有一个专门的合约Registry跟踪子合约的每次升级情况，主合约可通过查询此Registyr合约取得最新的子合约地址。卫星合约重新部署后，新地址通过Registry.update函数来更新。

```
contract Registry{

    address _current;
    address[] _previous;

    //子合约升级了，就通过update函数更新地址
    function update(address newAddress) public{
        if(newAddress != _current){
            _previous.push(_current);
            _current = newAddress;
        }
    }

    function getCurrent() public view returns(address){
        return _current;
    }
}
```

主合约依赖于Registry获取最新的卫星合约地址。

```
contract Base {
    uint public _variable;

    function setVariable(uint data) public {
        Satellite satellite = Satellite(_registry.getCurrent());
        _variable = satellite.compute(data);
    }

    Registry private _registry = //...;
}
```

Contract Relay - 代理调用最新合约

该设计模式所解决问题与Contract Registry一样，即主合约无需暴露维护性接口就可调用最新子合约。该模式下，存在一个代理合约，和子合约享有相同接口，负责将主合约的调用请求传递给真正的子合约。卫星合约重新部署后，新地址通过SatelliteProxy.update函数来更新。

```
contract SatelliteProxy{
    address _current;
    function compute(uint a) public returns(uint){
```

(continues on next page)

(续上页)

```
        Satellite satellite = Satellite(_current);
        return satellite.compute(a);
    }

    //子合约升级了，就通过update函数更新地址
    function update(address newAddress) public{
        if(newAddress != _current){
            _current = newAddress;
        }
    }
}


contract Satellite {
    function compute(uint a) public returns(uint){
        return a * 10;
    }
}
```

主合约依赖于SatelliteProxy：

```
contract Base {
    uint public _variable;

    function setVariable(uint data) public {
        _variable = _proxy.compute(data);
    }
    SatelliteProxy private _proxy = //...;
}
```

生命周期（Lifecycle）

在默认情况下，一个合约的生命周期近乎无限——除非赖以生存的区块链被消灭。但很多时候，用户希望缩短合约的生命周期。这一节将介绍两个简单模式提前终结合约生命。

Mortal - 允许合约自毁

字节码中有一个selfdestruct指令，用于销毁合约。所以只需要暴露出自毁接口即可：

```
contract Mortal{

    //自毁
    function destroy() public{
        selfdestruct(msg.sender);
    }
}
```

Automatic Deprecation - 允许合约自动停止服务

如果你希望一个合约在指定期限后停止服务，而不需要人工介入，可以使用Automatic Deprecation模式。

```
contract AutoDeprecated{

    uint private _deadline;
```

(continues on next page)

(续上页)

```
    function setDeadline(uint time) public {
        _deadline = time;
    }

    modifier notExpired(){
        require(now <= _deadline);
        _;
    }

    function service() public notExpired{
        //some code
    }
}
```

当用户调用service，notExpired修饰符会先进行日期检测，这样，一旦过了特定时间，调用就会因过期而被拦截在notExpired层。

权限（Authorization）

前文中有许多管理性接口，这些接口如果任何人都可调用，会造成严重后果，例如上文中的自毁函数，假设任何人都能访问，其严重性不言而喻。所以，一套保证只有特定账户能够访问的权限控制设计模式显得尤为重要。

Ownership

对于权限的管控，可以采用Ownership模式。该模式保证了只有合约的拥有者才能调用某些函数。首先需要有一个Owned合约：

```
contract Owned{

    address public _owner;

    constructor() {
        _owner = msg.sender;
    }

    modifier onlyOwner(){
        require(_owner == msg.sender);
        _;
    }
}
```

如果一个业务合约，希望某个函数只由拥有者调用，该怎么办呢？如下：

```
contract Biz is Owned{
    function manage() public onlyOwner{
    }
}
```

这样，当调用manage函数时，onlyOwner修饰符就会先运行并检测调用者是否与合约拥有者一致，从而将无授权的调用拦截在外。

行为控制（Action And Control）

这类模式一般针对具体场景使用，这节将主要介绍基于隐私的编码模式和与链外数据交互的设计模式。

Commit - Reveal - 延迟秘密泄露

链上数据都是公开透明的，一旦某些隐私数据上链，任何人都可看到，并且再也无法撤回。Commit And Reveal模式允许用户将要保护的数据转换为不可识别数据，比如一串哈希值，直到某个时刻再揭示哈希值的含义，展露真正的原值。以投票场景举例，假设需要在所有参与者都完成投票后再揭示投票内容，以防这期间参与者受票数影响。我们可以看看，在这个场景下所用到的具体代码：

```
contract CommitReveal {

    struct Commit {
        string choice;
        string secret;
        uint status;
    }

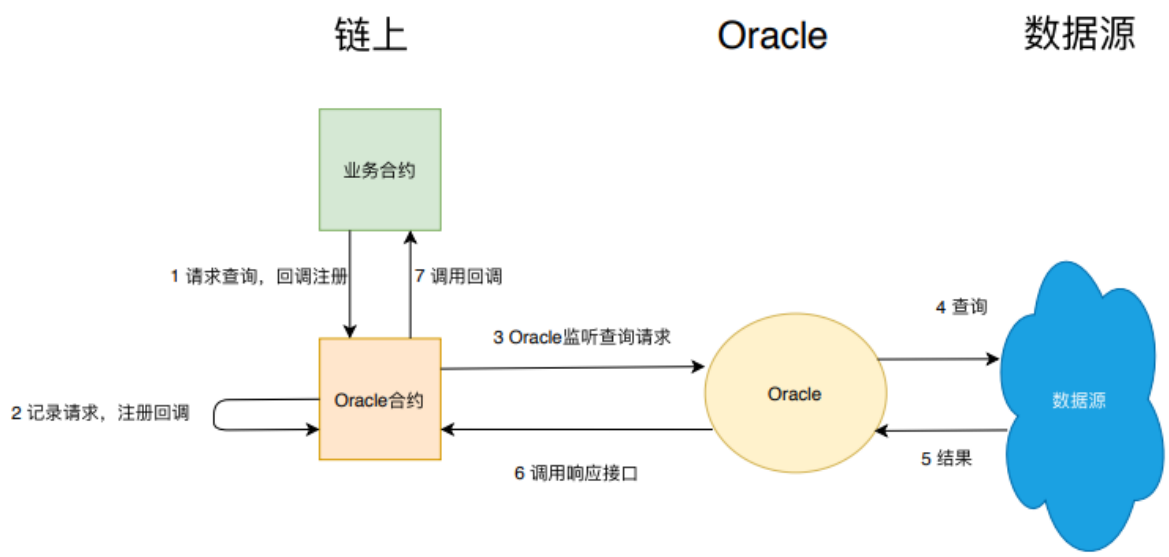
    mapping(address => mapping(bytes32 => Commit)) public userCommits;
    event LogCommit(bytes32, address);
    event LogReveal(bytes32, address, string, string);

    function commit(bytes32 commit) public {
        Commit storage userCommit = userCommits[msg.sender][commit];
        require(userCommit.status == 0);
        userCommit.status = 1; // committed
        emit LogCommit(commit, msg.sender);
    }

    function reveal(string choice, string secret, bytes32 commit) public {
        Commit storage userCommit = userCommits[msg.sender][commit];
        require(userCommit.status == 1);
        require(commit == keccak256(choice, secret));
        userCommit.choice = choice;
        userCommit.secret = secret;
        userCommit.status = 2;
        emit LogReveal(commit, msg.sender, choice, secret);
    }
}
```

Oracle - 读取链外数据

目前，链上的智能合约生态相对封闭，无法获取链外数据，影响了智能合约的应用范围。链外数据可极大扩展智能合约的使用范围，比如在保险业中，如果智能合约可读取到现实发生的意外事件，就可自动执行理赔。获取外部数据会通过名为Oracle的链外数据层来执行。当业务方的合约尝试获取外部数据时，会先将查询请求存入到某个Oracle专用合约内；Oracle会监听该合约，读取到这个查询请求后，执行查询，并调用业务合约响应接口使合约获取结果。



下面定义了一个Oracle合约：

```
contract Oracle {
    address oracleSource = 0x123; // known source

    struct Request {
        bytes data;
        function(bytes memory) external callback;
}

    Request[] requests;
    event NewRequest(uint);
    modifier onlyByOracle() {
        require(msg.sender == oracleSource); _;
    }

    function query(bytes data, function(bytes memory) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    //回调函数，由Oracle调用
    function reply(uint requestID, bytes response) public onlyByOracle() {
        requests[requestID].callback(response);
    }
}
```

业务方合约与Oracle合约进行交互：

```
contract BizContract {
    Oracle _oracle;

    constructor(address oracle){
        _oracle = Oracle(oracle);
    }

    modifier onlyByOracle() {
        require(msg.sender == address(_oracle));
```

(continues on next page)

(续上页)

```
        _;
    }

    function updateExchangeRate() {
        _oracle.query("USD", this.oracleResponse);
    }

    //回调函数，用于读取响应
    function oracleResponse(bytes response) onlyByOracle {
    // use the data
    }
}
```

总结

本文的介绍涵盖了安全性、可维护性等多种设计模式，其中，有些偏原则性，如Security和Maintenance设计模式；有些是偏实践，例如Authrization，Action And Control。这些设计模式，尤其实践类，并不能涵盖所有场景。随着对实际业务的深入探索，会遇到越来越多的特定场景与问题，开发者可对这些模式提炼、升华，以沉淀出针对某类问题的设计模式。上述设计模式是程序员的有力武器，掌握它们可应对许多已知场景，但更应掌握提炼设计模式的方法，这样才能从容应对未知领域，这个过程离不开对业务的深入探索，对软件工程原则的深入理解。

智能合约编写之Solidity的编程攻略

作者：毛嘉宇 | FISCO BCOS 核心开发者

前 言

作为一名搬砖多年的资深码农，刚开始接触Solidity便感觉无从下手：昂贵的计算和存储资源、简陋的语法特性、令人抓狂的debug体验、近乎贫瘠的类库支持、一言不合就插入汇编语句……让人不禁怀疑，这都已经过了9012年了，居然还有这种反人类的语言？对于习惯使用各类日益“傻瓜化”的类库和自动化高级框架的码农而言，学习Solidity的过程就是一场一言难尽的劝退之旅。但随着对区块链底层技术的深入学习，大家会慢慢理解作为运行在“The World Machine”上的Solidity语言，必须要严格遵循的设计原则以及权衡后必须付出的代价。正如黑客帝国中那句著名的slogan：“Welcome to the dessert of the real”，在恶劣艰苦的环境面前，最重要的是学会如何适应环境、保存自身并快速进化。本文总结了一些Solidity编程的攻略，期待各位读者不吝分享交流，达到抛砖引玉之效。

上链的原则

“如无必要，勿增实体”。

基于区块链技术及智能合约发展现状，数据的上链需遵循以下原则：

- 需要分布式协作的重要数据才上链，非必需数据不上链；
- 敏感数据脱敏或加密后上链（视数据保密程度选择符合隐私保护安全等级要求的加密算法）；
- 链上验证，链下授权。

在使用区块链时，开发者不需要将所有业务和数据都放到链上。相反，“好钢用在刀刃上”，智能合约更适合被用在分布式协作的业务场景中。

精简函数变量

如果在智能合约中定义了复杂的逻辑，特别是合约内定义了复杂的函数入参、变量和返回值，就会在编译的时候碰到以下错误：

```
Compiler error: Stack too deep, try removing local variables.
```

这也是社区中的高频技术问题之一。造成这个问题的原因就是EVM所设计用于最大的栈深度为16。所有的计算都在一个栈内执行，对栈的访问只限于其顶端，限制方式为：允许拷贝最顶端16个元素中的一个到栈顶，或者将栈顶元素和下面16个元素中的一个交换。所有其他操作都只能取最顶的几个元素，运算后，把结果压入栈顶。当然可以把栈上的元素放到存储或内存中。但无法只访问栈上指定深度的那个元素，除非先从栈顶移除其他元素。如果一个合约中，入参、返回值、内部变量的大小超过了16个，显然就超出了栈的最大深度。因此，我们可以使用结构体或数组来封装入参或返回值，达到减少栈顶元素使用的目的，从而避免此错误。例如以下代码，通过使用bytes数组来封装了原本16个bytes变量。

```
function doBiz(bytes[] paras) public {
        require(paras.length >= 16);
        // do something
}
```

保证参数和行为符合预期

心怀“Code is law”的远大理想，极客们设计和创造了区块链的智能合约。在联盟链中，不同的参与者可以使用智能合约来定义和书写一部分业务或交互的逻辑，以完成部分社会或商业活动。

相比于传统软件开发，智能合约对函数参数和行为的安全性要求更为严格。在联盟链中提供了身份实名和CA证书等机制，可以有效定位和监管所有参与者。不过，智能合约缺乏对漏洞和攻击的事前干预机制。正所谓字字珠玑，如果不严谨地检查智能合约输入参数或行为，有可能会触发一些意想不到的bug。

因此，在编写智能合约时，一定要注意对合约参数和行为的检查，尤其是那些对外部开放的合约函数。Solidity提供了require、revert、assert等关键字来进行异常的检测和处理。一旦检测并发现错误，整个函数调用会被回滚，所有状态修改都会被回退，就像从未调用过函数一样。以下分别使用了三个关键字，实现了相同的语义。

```
require(_data == data, "require data is valid");

if(_data != data) { revert("require data is valid"); }

assert(_data == data);
```

不过，这三个关键字一般适用于不同的使用场景：

- require：最常用的检测关键字，用来验证输入参数和调用函数结果是否合法。
- revert：适用在某个分支判断的场景下。
- assert: 检查结果是否正确、合法，一般用于函数结尾。

在一个合约的函数中，可以使用函数修饰器来抽象部分参数和条件的检查。在函数体内，可以对运行状态使用if-else等判断语句进行检查，对异常的分支使用revert回退。在函数运行结束前，可以使用assert对执行结果或中间状态进行断言检查。在实践中，推荐使用require关键字，并将条件检查移到函数修饰器中去；这样可以让函数的职责更为单一，更专注到业务逻辑中。同时，函数修饰器等条件代码也更容易被复用，合约也会更加安全、层次化。

在本文中，我们以一个水果店库存管理系统为例，设计一个水果超市的合约。这个合约只包含了对店内所有水果品类和库存数量的管理，setFruitStock函数提供了对应水果库存设置的函数。在这个合约中，我们需要检查传入的参数，即水果名称不能为空。

```
pragma solidity ^0.4.25;

contract FruitStore {
    mapping(bytes => uint) _fruitStock;
    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    function setFruitStock(bytes fruitName, uint stock) validFruitName(fruitName)␣
↪external {
        _fruitStock[fruitName] = stock;
    }
}
```

如上所述，我们添加了函数执行前的参数检查的函数修饰器。同理，通过使用函数执行前和函数执行后检查的函数修饰器，可以保证智能合约更加安全、清晰。智能合约的编写需要设置严格的前置和后置函数检查，来保证其安全性。

严控函数的执行权限

如果说智能合约的参数和行为检测提供了静态的合约安全措施，那么合约权限控制的模式则提供了动态访问行为的控制。由于智能合约是发布到区块链上，所有数据和函数对所有参与者都是公开透明的，任一节点参与者都可发起交易，无法保证合约的隐私。因此，合约发布者必须对函数设计严格的访问限制机制。Solidity提供了函数可见性修饰符、修饰器等语法，灵活地使用这些语法，可帮助构建起合法授权、受控调用的智能合约系统。还是以刚才的水果合约为例。现在getStock提供了查询具体水果库存数量的函数。

```
pragma solidity ^0.4.25;

contract FruitStore {
    mapping(bytes => uint) _fruitStock;
    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    function getStock(bytes fruit) external view returns(uint) {
        return _fruitStock[fruit];
    }
    function setFruitStock(bytes fruitName, uint stock) validFruitName(fruitName)␣
↪external {
        _fruitStock[fruitName] = stock;
    }
}
```

水果店老板将这个合约发布到了链上。但是，发布之后，setFruitStock函数可被任何其他联盟链的参与者调用。虽然联盟链的参与者是实名认证且可事后追责；但一旦有恶意攻击者对水果店发起攻击，调用setFruitStock函数就能任意修改水果库存，甚至将所有水果库存清零，这将对水果店正常经营管理产生严重后果。因此，设置某些预防和授权的措施很必要：对于修改库存的函数setFruitStock，可在函数执行前对调用者进行鉴权。类似的，这些检查可能会被多个修改数据的函数复用，使用一个onlyOwner的修饰器就可以抽象此检查。owner字段代表了合约的所有者，会在合约构造函数中被初始化。使用public修饰getter查询函数，就可以通过_owner()函数查询合约的所有者。

```
contract FruitStore {
    address public  _owner;
    mapping(bytes => uint) _fruitStock;

    constructor() public {
        _owner = msg.sender;
    }
```

(continues on next page)

(续上页)

```
    modifier validFruitName(bytes fruitName) {
        require(fruitName.length > 0, "fruite name is invalid!");
        _;
    }
    // 鉴权函数修饰器
    modifier onlyOwner() {
        require(msg.sender == _owner, "Auth: only owner is authorized.");
        _;
    }
    function getStock(bytes fruit) external view returns(uint) {
        return _fruitStock[fruit];
    }
    // 添加了onlyOwner修饰器
    function setFruitStock(bytes fruitName, uint stock)
        onlyOwner validFruitName(fruitName) external {
        _fruitStock[fruitName] = stock;
    }
}
```

这样一来，我们可以将相应的函数调用权限检查封装到修饰器中，智能合约会自动发起对调用者身份验证检查，并且只允许合约部署者来调用setFruitStock函数，以此保证合约函数向指定调用者开放。

抽象通用的业务逻辑

分析上述FruitStore合约，我们发现合约里似乎混入了奇怪的东西。参考单一职责的编程原则，水果店库存管理合约多了上述函数功能检查的逻辑，使合约无法将所有代码专注在自身业务逻辑中。对此，我们可以抽象出可复用的功能，利用Solidity的继承机制继承最终抽象的合约。基于上述FruitStore合约，可抽象出一个BasicAuth合约，此合约包含之前onlyOwner的修饰器和相关功能接口。

```
contract BasicAuth {
    address public _owner;

    constructor() public {
        _owner = msg.sender;
    }

    function setOwner(address owner)
        public
        onlyOwner
{
        _owner = owner;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only owner is authorized.");
        _;
    }
}
```

FruitStore可以复用这个修饰器，并将合约代码收敛到自身业务逻辑中。

```
import "./BasicAuth.sol";

contract FruitStore is BasicAuth {
    mapping(bytes => uint) _fruitStock;

    function setFruitStock(bytes fruitName, uint stock)
        onlyOwner validFruitName(fruitName) external {
```

(continues on next page)

(续上页)

```
        _fruitStock[fruitName] = stock;
    }
}
```

这样一来，FruitStore的逻辑被大大简化，合约代码更精简、聚焦和清晰。

预防私钥的丢失

在区块链中调用合约函数的方式有两种：内部调用和外部调用。出于隐私保护和权限控制，业务合约会定义一个合约所有者。假设用户A部署了FruitStore合约，那上述合约owner就是部署者A的外部账户地址。这个地址由外部账户的私钥计算生成。但是，在现实世界中，私钥泄露、丢失的现象比比皆是。一个商用区块链DAPP需要严肃考虑私钥的替换和重置等问题。这个问题最为简单直观的解决方法是添加一个备用私钥。这个备用私钥可支持权限合约修改owner的操作，代码如下：

```
ontract BasicAuth {
    address public  _owner;
    address public _bakOwner;

    constructor(address bakOwner) public {
        _owner = msg.sender;
        _bakOwner = bakOwner;
    }

    function setOwner(address owner)
        public
        canSetOwner
{
        _owner = owner;
    }

    function setBakOwner(address owner)
        public
        canSetOwner
{
        _bakOwner = owner;
    }

    // ...

    modifier isAuthorized() {
        require(msg.sender == _owner || msg.sender == _bakOwner, "BasicAuth: only␣
↪owner or back owner is authorized.");
        _;
    }
}
```

这样，当发现私钥丢失或泄露时，我们可以使用备用外部账户调用setOwner重置账号，恢复、保障业务正常运行。

面向接口编程

上述私钥备份理念值得推崇，不过其具体实现方式存在一定局限性，在很多业务场景下，显得过于简单粗暴。对于实际的商业场景，私钥的备份和保存需要考虑的维度和因素要复杂得多，对应密钥备份策略也更多元化。以水果店为例，有的连锁水果店可能希望通过品牌总部来管理私钥，也有的可能通过社交关系重置帐号，还有的可能会绑定一个社交平台的管理帐号.....面向接口编程，而不依赖具体的实现细节，可以有效规避这个问题。例如，我们利用接口功能首先定义一个判断权限的抽象接口：

```
contract Authority {
    function canCall(
        address src, address dst, bytes4 sig
    ) public view returns (bool);
}
```

这个canCall函数涵盖了函数调用者地址、目标调用合约的地址和函数签名，函数返回一个bool的结果。这包含了合约鉴权所有必要的参数。我们可进一步修改之前的权限管理合约，并在合约中依赖Authority接口，当鉴权时，修饰器会调用接口中的抽象方法：

```
contract BasicAuth {
    Authority  public  _authority;

    function setAuthority(Authority authority)
        public
        auth
    {
        _authority = authority;
    }

    modifier isAuthorized() {
        require(auth(msg.sender, msg.sig), "BasicAuth: only owner or back owner is␣
↪authorized.");
        _;
    }

    function auth(address src, bytes4 sig) public view returns (bool) {
        if (src == address(this)) {
            return true;
        } else if (src == _owner) {
            return true;
        } else if (_authority == Authority(0)) {
            return false;
        } else {
            return _authority.canCall(src, this, sig);
        }
    }
}
```

这样，我们只需要灵活定义实现了canCall接口的合约，在合约的canCall方法中定义具体判断逻辑。而业务合约，例如FruitStore继承BasicAuth合约，在创建时只要传入具体的实现合约，就可以实现不同判断逻辑。

合理预留事件

迄今为止，我们已实现强大灵活的权限管理机制，只有预先授权的外部账户才能修改合约owner属性。不过，仅通过上述合约代码，我们无法记录和查询修改、调用函数的历史记录和明细信息。而这样的需求在实际业务场景中比比皆是。比如，FruitStore水果店需要通过查询历史库存修改记录，计算出不同季节的畅销与滞销水果。

一种方法是依托链下维护独立的台账机制。不过，这种方法存在很多问题：保持链下台账和链上记录一致的成本开销非常高；同时，智能合约面向链上所有参与者开放，一旦其他参与者调用了合约函数，相关交易信息就存在不能同步的风险。针对此类场景，Solidity提供了event语法。event不仅具备可供SDK监听回调的机制，还能用较低的gas成本将事件参数等信息完整记录、保存到区块中。FISCO BCOS社区中，也有WEBASE-Collect-Bee这样的工具，在事后实现区块历史事件信息的完整导出。

WEBASE-Collect-Bee工具参考

基于上述权限管理合约，我们可以定义相应的修改权限事件，其他事件以此类推。

```
event LogSetAuthority (Authority indexed authority, address indexed from);
}
```

接下来，可以调用相应的事件：

```
function setAuthority(Authority authority)
        public
        auth
{
        _authority = authority;
        emit LogSetAuthority(authority, msg.sender);
    }
```

当setAuthority函数被调用时，会同时触发LogSetAuthority，将事件中定义的Authority合约地址以及调用者地址记录到区块链交易回执中。当通过控制台调用setAuthority方法时，对应事件LogSetAuthority也会被打印出来。基于WEBASE-Collect-Bee，我们可以导出所有该函数的历史信息到数据库中。也可基于WEBASE-Collect-Bee进行二次开发，实现复杂的数据查询、大数据分析和数据可视化等功能。

遵循安全编程规范

每一门语言都有其相应的编码规范，我们需要尽可能严格地遵循Solidity官方编程风格指南，使代码更利于阅读、理解和维护，有效地减少合约的bug数量。Solidity官方编程风格指南参考。除了编程规范，业界也总结了很多安全编程指南，例如重入漏洞、数据结构溢出、随机数误区、构造函数失控、为初始化的存储指针等等。重视和防范此类风险，采用业界推荐的安全编程规范至关重要，例如Solidity官方安全编程指南。同时，在合约发布上线后，还需要注意关注、订阅Solidity社区内安全组织或机构发布的各类安全漏洞、攻击手法，一旦出现问题，及时做到亡羊补牢。

对于重要的智能合约，有必要引入审计。现有的审计包括了人工审计、机器审计等方法，通过代码分析、规则验证、语义验证和形式化验证等方法保证合约安全性。虽然本文通篇都在强调，模块化和重用被严格审查并广泛验证的智能合约是最佳的实践策略。但在实际开发过程，这种假设过于理想化，每个项目或多或少都会引入新的代码，甚至从零开始。不过，我们仍然可以视代码的复用程度进行审计分级，显式地标注出引用的代码，将审计和检查的重点放在新代码上，以节省审计成本。

最后，“前事不忘后事之师”，我们需要不断总结和学习前人的最佳实践，动态和可持续地提升编码工程水平，并不断应用到具体实践中。

积累和复用成熟的代码

前文面向接口编程中的思想可降低代码耦合，使合约更容易扩展、利于维护。在遵循这条规则之外，还有另外一条忠告：尽可能地复用现有代码库。智能合约发布后难以修改或撤回，而且发布到公开透明的区块链环境上，就意味着一旦出现bug造成的损失和风险更甚于传统软件。因此，复用一些更好更安全的轮子远胜过重新造轮子。在开源社区中，已经存在大量的业务合约和库可供使用，例如OpenZeppelin等优秀的库。如果在开源世界和过去团队的代码库里找不到合适的可复用代码，建议在编写新代码时尽可能地测试和完善代码设计。此外，还要定期分析和审查历史合约代码，将其模板化，以便于扩展和复用。

例如，针对上面的BasicAuth，参考防火墙经典的ACL(Access Control List)设计，我们可以进一步地继承和扩展BasicAuth，抽象出ACL合约控制的实现。

```
contract AclGuard is BasicAuth {
    bytes4 constant public ANY_SIG = bytes4(uint(-1));
    address constant public ANY_ADDRESS = address(bytes20(uint(-1)));
    mapping (address => mapping (address => mapping (bytes4 => bool))) _acl;

    function canCall(
        address src, address dst, bytes4 sig
) public view returns (bool) {
        return _acl[src][dst][sig]
```

(continues on next page)

(续上页)

```
            || _acl[src][dst][ANY_SIG]
            || _acl[src][ANY_ADDRESS][sig]
            || _acl[src][ANY_ADDRESS][ANY_SIG]
            || _acl[ANY_ADDRESS][dst][sig]
            || _acl[ANY_ADDRESS][dst][ANY_SIG]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][sig]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][ANY_SIG];
    }

    function permit(address src, address dst, bytes4 sig) public onlyAuthorized {
        _acl[src][dst][sig] = true;
        emit LogPermit(src, dst, sig);
    }

    function forbid(address src, address dst, bytes4 sig) public onlyAuthorized {
        _acl[src][dst][sig] = false;
        emit LogForbid(src, dst, sig);
    }

    function permit(address src, address dst, string sig) external {
        permit(src, dst, bytes4(keccak256(sig)));
    }

    function forbid(address src, address dst, string sig) external {
        forbid(src, dst, bytes4(keccak256(sig)));
    }

    function permitAny(address src, address dst) external {
        permit(src, dst, ANY_SIG);
    }

    function forbidAny(address src, address dst) external {
        forbid(src, dst, ANY_SIG);
    }
}
```

在这个合约里，有调用者地址、被调用合约地址和函数签名三个主要参数。通过配置ACL的访问策略，可以精确地定义和控制函数访问行为及权限。合约内置了ANY的常量，匹配任意函数，使访问粒度的控制更加便捷。这个模板合约实现了强大灵活的功能，足以满足所有类似权限控制场景的需求。

提升存储和计算的效率

迄今为止，在上述的推演过程中，更多的是对智能合约编程做加法。但相比传统软件环境，智能合约上的存储和计算资源更加宝贵。因此，如何对合约做减法也是用好Solidity的必修课程之一。

选取合适的变量类型

显式的问题可通过EVM编译器检测出来并报错；但大量的性能问题可能被隐藏在代码的细节中。Solidity提供了非常多精确的基础类型，这与传统的编程语言大相径庭。下面有几个关于Solidity基础类型的小技巧。在C语言中，可以用short\int\long按需定义整数类型，而到了Solidity，不仅区分int和uint，甚至还能定义uint的长度，比如uint8是一个字节，uint256是32个字节。这种设计告诫我们，能用uint8搞定的，绝对不要用uint16！几乎所有Solidity的基本类型，都能在声明时指定其大小。开发者一定要有效利用这一语法特性，编写代码时只要满足需求就尽可能选取小的变量类型。数据类型bytes32可存放 32 个（原始）字节，但除非数据是bytes32或bytes16这类定长的数据类型，否则更推荐使用长度可以变化的bytes。bytes类似byte[]，但在外部函数中会自动压缩打包，更节省空间。如果变量内容是英文的，不需要采用UTF-8编码，在这里，推荐bytes而不是string。string默认采用UTF-8编码，所以相同字符串的存储成本会高很多。

紧凑状态变量打包

除了尽可能使用较小的数据类型来定义变量，有的时候，变量的排列顺序也非常重要，可能会影响到程序执行和存储效率。其中根本原因还是EVM，不管是EVM存储插槽（Storage Slot）还是栈，每个元素长度是一个字（256位，32字节）。分配存储时，所有变量（除了映射和动态数组等非静态类型）都会按声明顺序从位置0开始依次写下。在处理状态变量和结构体成员变量时，EVM会将多个元素打包到一个存储插槽中，从而将多个读或写合并到一次对存储的操作中。值得注意的是，使用小于32 字节的元素时，合约的gas使用量可能高于使用32字节元素时。这是因为EVM每次会操作32个字节，所以如果元素比32字节小，必须使用更多的操作才能将其大小缩减到所需。这也解释了Solidity中最常见的数据类型，例如int，uint，byte32，为何都刚好占用32个字节。所以，当合约或结构体声明多个状态变量时，能否合理地组合安排多个存储状态变量和结构体成员变量，使之占用更少的存储位置就十分重要。例如，在以下两个合约中，经过实际测试，Test1合约比Test2合约占用更少的存储和计算资源。

```
contract Test1 {
    //占据2个slot, "gasUsed":188873
    struct S {
        bytes1 b1;
        bytes31 b31;
        bytes32 b32;
    }
    S s;
    function f() public {
        S memory tmp = S("a","b","c");
        s = tmp;
    }
}

contract Test2 {
    //占据1个slot, "gasUsed":188937
    struct S {
        bytes31 b31;
        bytes32 b32;
        bytes1 b1;
    }
    // ......
}
```

优化查询接口

查询接口的优化点很多，比如一定要在只负责查询的函数声明中添加view修饰符，否则查询函数会被当成交易打包并发送到共识队列，被全网执行并被记录在区块中；这将大大增加区块链的负担，占用宝贵的链上资源。再如，不要在智能合约中添加复杂的查询逻辑，因为任何复杂查询代码都会使整个合约变得更长更复杂。读者可使用上文提及的WeBASE数据导出组件，将链上数据导出到数据库中，在链下进行查询和分析。

缩减合约binary长度

开发者编写的Solidity代码会被编译为binary code，而部署智能合约的过程实际上就是通过一个transaction将binary code存储在链上，并取得专属于该合约的地址。缩减binary code的长度可节省网络传输、共识打包数据存储的开销。例如，在典型的存证业务场景中，每次客户存证都会新建一个存证合约，因此，应当尽可能地缩减binary code的长度。常见思路是裁剪不必要的逻辑，删掉冗余代码。特别是在复用代码时，可能引入一些非刚需代码。以上文ACL合约为例，支持控制合约函数粒度的权限。

```
function canCall(
        address src, address dst, bytes4 sig
    ) public view returns (bool) {
        return _acl[src][dst][sig]
```

(continues on next page)

(续上页)

```
            || _acl[src][dst][ANY_SIG]
            || _acl[src][ANY_ADDRESS][sig]
            || _acl[src][ANY_ADDRESS][ANY_SIG]
            || _acl[ANY_ADDRESS][dst][sig]
            || _acl[ANY_ADDRESS][dst][ANY_SIG]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][sig]
            || _acl[ANY_ADDRESS][ANY_ADDRESS][ANY_SIG];
    }
```

但在具体业务场景中，只需要控制合约访问者即可，通过删除相应代码，进一步简化使用逻辑。这样一来，对应合约的binary code长度会大大缩小。

```
function canCall(
        address src, address dst
) public view returns (bool) {
        return _acl[src][dst]
            || _acl[src][ANY_ADDRESS]
            || _acl[ANY_ADDRESS][dst];
    }
```

另一种缩减binary code的思路是采用更紧凑的写法。经实测，采取如上短路原则的判断语句，其binary长度会比采用if-else语法的更短。同样，采用if-else的结构，也会比if-if-if的结构生成更短的binary code。最后，在对binary code长度有极致要求的场景中，应当尽可能避免在合约中新建合约，这会显著增加binary的长度。例如，某个合约中有如下的构造函数:

```
constructor() public {
        // 在构造器内新建一个新对象
        _a = new A();
}
```

我们可以采用在链下构造A对象，并基于address传输和固定校验的方式，来规避这一问题。

```
constructor(address a) public {
        A _a = A(a);
        require(_a._owner == address(this));
}
```

当然，这样也可能会使合约交互方式变得复杂。但其提供了有效缩短binary code长度的捷径，需要在具体业务场景中做权衡取舍。

保证合约可升级

经典的三层结构

通过前文方式，我们尽最大努力保持合约设计的灵活性；翻箱倒柜复用了轮子；也对发布合约进行全方位、无死角的测试。除此之外，随着业务需求变化，我们还将面临一个问题：如何保证合约平滑、顺利的升级？作为一门高级编程语言，Solidity支持运行一些复杂控制和计算逻辑，也支持存储智能合约运行后的状态和业务数据。不同于WEB开发等场景的应用-数据库分层架构，Solidity语言甚至没有抽象出一层独立的数据存储结构，数据都被保存到了合约中。但是，一旦合约需要升级，这种模式就会出现瓶颈。

在Solidity中，一旦合约部署发布后，其代码就无法被修改，只能通过发布新合约去改动代码。假如数据存储在老合约，就会出现所谓的“孤儿数据”问题，新合约将丢失之前运行的历史业务数据。这种情况，开发者可以考虑将老合约数据迁移到新合约中，但此操作至少存在两个问题:

1. 迁移数据会加重区块链的负担，产生资源浪费和消耗，甚至引入安全问题;
2. 牵一发而动全身，会引入额外的迁移数据逻辑，增加合约复杂度。

一种更合理的方式是抽象一层独立的合约存储层。这个存储层只提供合约读写的最基本方法，而不包含任何业务逻辑。在这种模式中，存在三种合约角色：

- 数据合约：在合约中保存数据，并提供数据的操作接口。
- 管理合约：设置控制权限，保证只有控制合约才有权限修改数据合约。
- 控制合约：真正需要对数据发起操作的合约。

具体的代码示例如下：

数据合约：

```
contract FruitStore is BasicAuth {
    address _latestVersion;
    mapping(bytes => uint) _fruitStock;

    modifier onlyLatestVersion() {
       require(msg.sender == _latestVersion);
        _;
    }

    function upgradeVersion(address newVersion) public {
        require(msg.sender == _owner);
        _latestVersion = newVersion;
    }

    function setFruitStock(bytes fruit, uint stock) onlyLatestVersion external {
        _fruitStock[fruit] = stock;
    }
}
```

管理合约：

```
contract Admin is BasicAuth {
    function upgradeContract(FruitStore fruitStore, address newController)␣
↪isAuthorized external {
        fruitStore.upgradeVersion(newController);
    }
}
```

控制合约：

```
contract FruitStoreController is BasicAuth {
    function upgradeStock(bytes fruit, uint stock) isAuthorized external {
        fruitStore.setFruitStock(fruit, stock);
    }
}
```

一旦函数的控制逻辑需要变更，开发者只需修改FruitStoreController控制合约逻辑，部署一个新合约，然后使用管理合约Admin修改新的合约地址参数就可轻松完成合约升级。这种方法可消除合约升级中因业务控制逻辑改变而导致的数据迁移隐患。但天下没有免费的午餐，这种操作需要在可扩展性和复杂性之间需要做基本的权衡。首先，数据和逻辑的分离降低了运行性能。其次，进一步封装增加了程序复杂度。最后，越是复杂的合约越会增加潜在攻击面，简单的合约比复杂的合约更安全。

通用数据结构

到目前为止，还存在一个问题，假如数据合约中的数据结构本身需要升级怎么办?

例如，在FruitStore中，原本只保存了库存信息，现在由于水果销售店生意发展壮大，一共开了十家分店，需要记录每家分店、每种水果的库存和售出信息。在这种情况下，一种解决方案是采用外部关联管理方式：创建一个新的ChainStore合约，在这个合约中创建一个mapping，建立分店名和FruitStore的关系。

此外，不同分店需要创建一个FruitStore的合约。为了记录新增的售出信息等数据，我们还需要新建一个合约来管理。假如在FruitStore中可预设一些不同类型的reserved字段，可帮助规避新建售出信息合约的开销，仍然复用FruitStore合约。但这种方式在最开始会增加存储开销。一种更好的思路是抽象一层更为底层和通用的存储结构。代码如下：

```
contract commonDB  is BasicAuth {
    mapping(bytes => uint) _uintMapping;

    function getUint(bytes key) external view returns(uint) {
        return _uintMapping[key];
    }

    function setUint(bytes key, uint value) isAuthorized onlyLatestVersion␣
↪external {
        _uintMapping[key] = value;
    }

}
```

类似的，我们可加入所有数据类型变量，帮助commonDB应对和满足不同的数据类型存储需求。相应的控制合约可修改如下：

```
contract FruitStoreControllerV2 is BasicAuth {
    function upgradeStock(bytes32 storeName, bytes32 fruit, uint stock)
        isAuthorized external {
        commonDB.setUint(sha256(storeName, fruit), stock);
        uint result = commonDB.getUint(sha256(storeName, fruit));
    }
}
```

使用以上存储的设计模式，可显著提升合约数据存储灵活性，保证合约可升级。众所周知，Solidity既不支持数据库，使用代码作为存储entity，也无法提供更改schema的灵活性。但是，通过这种KV设计，可以使存储本身获得强大的可扩展性。总之，**没有一个策略是完美的，优秀的架构师善于权衡**。智能合约设计者需要充分了解各种方案的利弊，并基于实际情况选择合适的设计方案。

总结

文至于此，希望激起读者对在Solidity世界生存与进化的兴趣。“若有完美，必有谎言”，软件开发的世界没有银弹。本文行文过程就是从最简单的合约逐步完善和进化的过程。在Solidity编程世界中，生存与进化都离不开三个关键词：安全、可复用、高效。生命不息，进化不止。短短一篇小文难以穷尽所有生存进化之术，希望这三个关键词能帮助大家在Solidity的世界里翱翔畅游，并不断书写辉煌的故事和传说：）

智能合约编写之Solidity运行原理

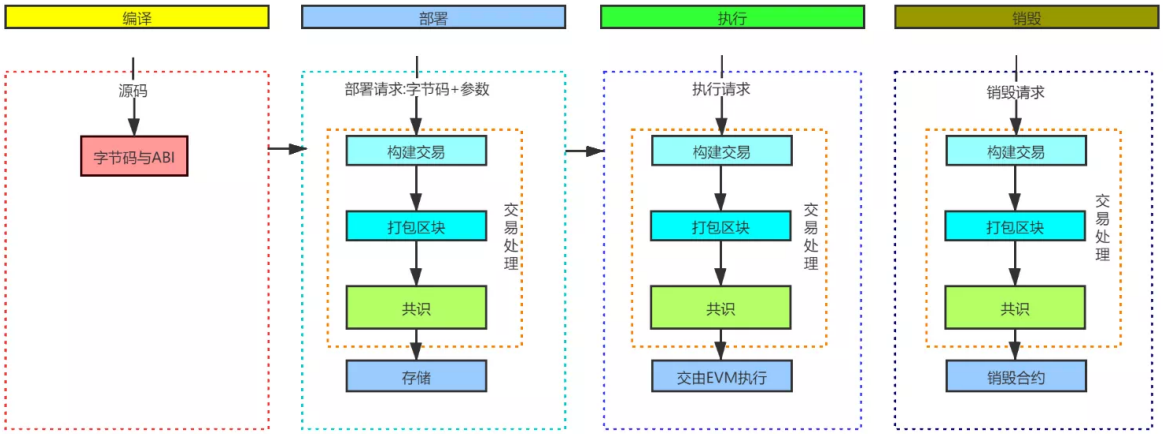
作者：储雨知 | FISCO BCOS 核心开发者

引 言

作为一门面向智能合约的语言，Solidity与其他经典语言既有差异也有相似之处。一方面，服务于区块链的属性使其与其他语言存在差异。例如，合约的部署与调用均要经过区块链网络确认；执行成本需要被严格控制，以防止恶意代码消耗节点资源。另一方面，身为编程语言，Solidity的实现并未脱离经典语言，比如Solidity中包含类似栈、堆的设计，采用栈式虚拟机来进行字节码处理。本系列前几篇文章介绍了如何开发Solidity程序，为了让读者知其然更知其所以然，本文将进一步介绍Solidity的内部运行原理，聚焦于Solidity程序的生命周期和EVM工作机制。

Solidity的生命周期

与其他语言一样，Solidity的代码生命周期离不开编译、部署、执行、销毁这四个阶段。下图整理展现了Solidity程序的完整生命周期：



经编译后，Solidity文件会生成字节码。这是一种类似jvm字节码的代码。部署时，字节码与构造参数会被构建成交易，这笔交易会被打包到区块中，经由网络共识过程，最后在各区块链节点上构建合约，并将合约地址返还用户。当用户准备调用该合约上的函数时，调用请求同样也会经历交易、区块、共识的过程，最终在各节点上由EVM虚拟机来执行。

下面是一个示例程序，我们通过remix探索它的生命周期。

```
pragma solidity ^0.4.25;

contract Demo{
    uint private _state;
    constructor(uint state){
        _state = state;
    }
    function set(uint state) public {
        _state = state;
    }
}
```

编译

源代码编译完后，可以通过ByteCode按钮得到它的二进制：

```
608060405234801561001057600080fd5b506040516020806100ed833981018060405281019080805190602001909291905
```

还可以得到对应的字节码（OpCode）：

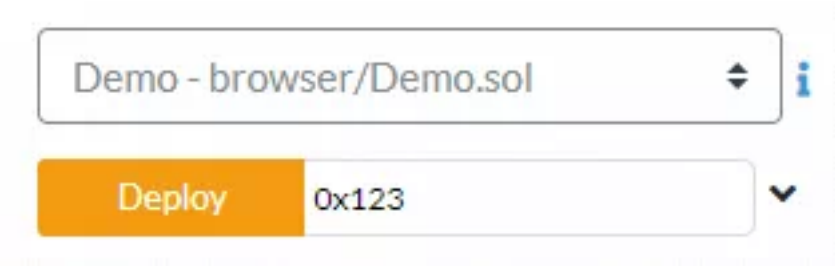
```
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH2 0x10 JUMPI PUSH1 0x0 DUP1␣
↪REVERT JUMPDEST POP PUSH1 0x40 MLOAD PUSH1 0x20 DUP1 PUSH2 0xED DUP4 CODECOPY␣
↪DUP2 ADD DUP1 PUSH1 0x40 MSTORE DUP2 ADD SWAP1 DUP1 DUP1 MLOAD SWAP1 PUSH1 0x20␣
↪ADD SWAP1 SWAP3 SWAP2 SWAP1 POP POP POP DUP1 PUSH1 0x0 DUP2 SWAP1 SSTORE POP POP␣
↪PUSH1 0xA4 DUP1 PUSH2 0x49 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1 0x80␣
↪PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1 0x0␣
↪CALLDATALOAD PUSH29 0x100000000000000000000000000000000000000000000000000000000␣
↪SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x60FE47B1 EQ PUSH1 0x44 JUMPI␣
↪JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST CALLVALUE DUP1 ISZERO PUSH1 0x4F JUMPI␣
↪PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x6C PUSH1 0x4 DUP1 CALLDATASIZE SUB␣
↪DUP2 ADD SWAP1 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1 SWAP3 SWAP2␣
↪SWAP1 POP POP POP PUSH1 0x6E JUMP JUMPDEST STOP JUMPDEST DUP1 PUSH1 0x0 DUP2␣
↪SWAP1 SSTORE POP POP JUMP STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 0x4e 0xd9 MOD␣
↪DIFFICULTY 0x4c 0xc4 0xc9 0xaa 0xbd XOR EXTCODECOPY MSTORE 0xb2 0xd4 DUP7 0xdf␣
↪0xc5 0xde 0xa9 DUP1 SLT PUSH1 0xC3 CALLDATACOPY XOR 0x5d 0xad KECCAK256 0xe1␣
↪0x1f DUP2 SHL STOP 0x29
```

其中下述指令集为set函数对应的代码，后面会解释set函数如何运行。

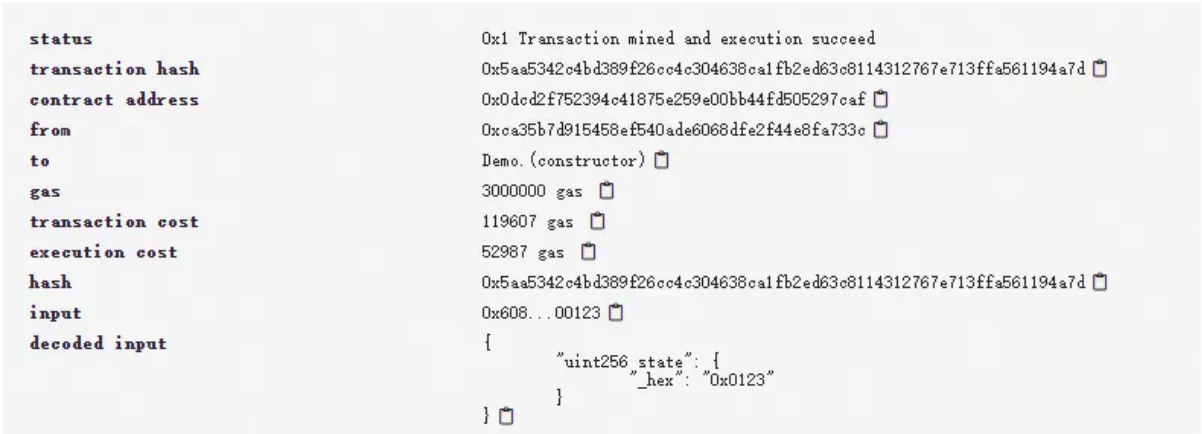
```
JUMPDEST DUP1 PUSH1 0x0 DUP2 SWAP1 SSTORE POP POP JUMP STOP
```

部署

编译完后，即可在remix上对代码进行部署，构造参数传入0x123:



部署成功后，可得到一条交易回执:



点开input，可以看到具体的交易输入数据:

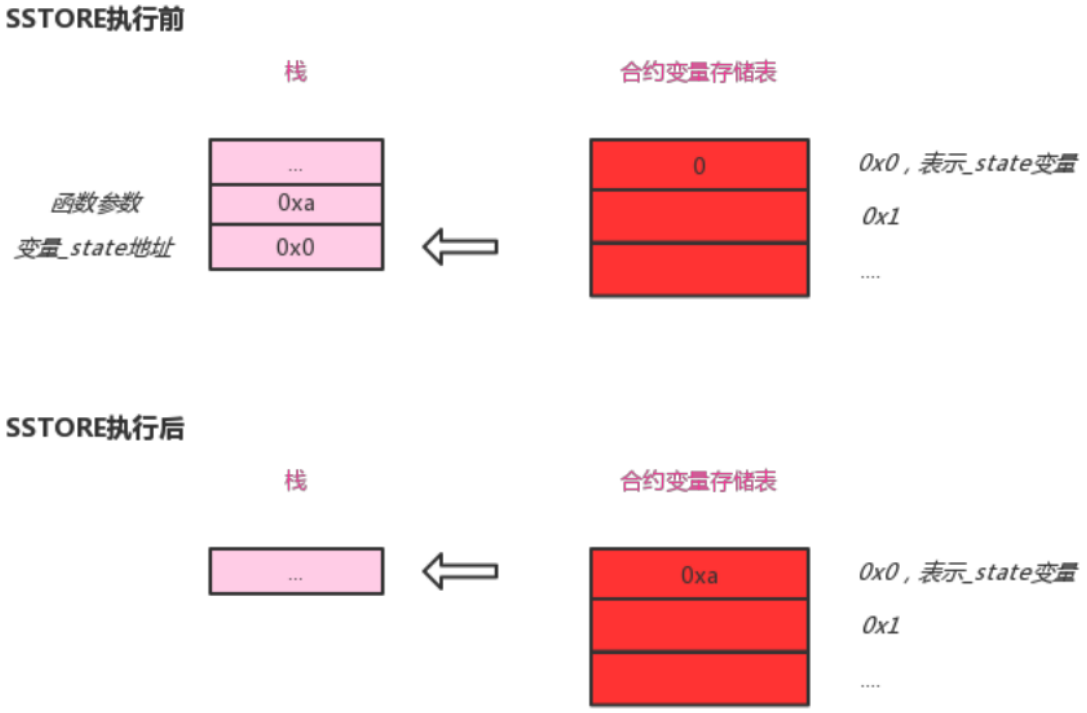
```
0x6080604052348015610010576000 80fd5b506040516020806100ed83398
1018060405281019080805190602001909291905050508060008190555050
60a4806100496000396000f3006080604052600436106 03f576000357c0100
0000000000000000000000000000000000000000000000000000000900463f
ffffff16806360fe47b1146044575b600080fd5b348015604f57600080fd5b50
606c6004803603810190808035906020019092919050505060 6e565b005b8
0600081905550505600a165627a7a723058204ed906444cc4c9aabd183c52
b2d486dfc5dea9801260c337185dad20e11f811b0029000000000000000000
0000000000000000000000000000000000000000000123
```

上面这段数据中，标黄的部分正好是前文中的合约二进制；而标紫的部分，则对应了传入的构造参数0x123。这些都表明，合约部署以交易作为介质。结合区块链交易知识，我们可以还原出整个部署过程：

- 客户端将部署请求(合约二进制，构造参数)作为交易的输入数据，以此构造出一笔交易
- 交易经过rlp编码，然后由发送者进行私钥签名
- 已签名的交易被推送到区块链上的节点
- 区块链节点验证交易后，存入交易池
- 轮到该节点出块时，打包交易构建区块，广播给其他节点
- 其他节点验证区块并取得共识。不同区块链可能采用不同共识算法，FISCO BCOS中采用PBFT取得共识，这要求经历三阶段提交（pre-prepare，prepare, commit）
- 节点执行交易，结果就是智能合约Demo被创建，状态字段_state的存储空间被分配，并被初始化为0x123

执行

根据是否带有修饰符view，我们可将函数分为两类：调用与交易。由于在编译期就确定了调用不会引起合约状态的变更，故对于这类函数调用，节点直接提供查询即可，无需与其他区块链节点确认。而由于交易可能引起状态变更，故会在网络间确认。下面将以用户调用了set(0x10)为假设，看看具体的运行过程。首先，函数set没有配置view/pure修饰符，这意味着其可能更改合约状态。所以这个调用信息会被放入一笔交易，经由交易编码、交易签名、交易推送、交易池缓存、打包出块、网络共识等过程，最终被交由各节点的EVM执行。在EVM中，由SSTORE字节码将参数0xa存储到合约字段_state中。该字节码先从栈上拿到状态字段_state的地址与新值0xa，随后完成实际存储。下图展示了运行过程：



这里仅粗略介绍了set(0xa)是如何运行，下节将进一步展开介绍EVM的工作机制以及数据存储机制。

销毁

由于合约上链后就无法篡改，所以合约生命可持续到底层区块链被彻底关停。若要手动销毁合约，可通过字节码selfdestruct。销毁合约也需要进行交易确认，在此不多作赘述。

EVM原理

在前文中，我们介绍了Solidity程序的运行原理。经过交易确认后，最终由EVM执行字节码。对EVM，上文只是一笔带过，这一节将具体介绍其工作机制。

运行原理

EVM是栈式虚拟机，其核心特征就是所有操作数都会被存储在栈上。下面我们将通过一段简单的Solidity语句代码看看其运行原理：

```
uint a = 1;
uint b = 2;
uint c = a + b;
```

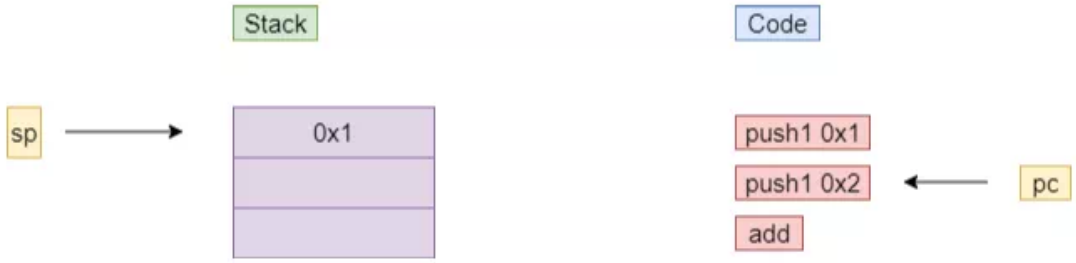
这段代码经过编译后，得到的字节码如下：

```
PUSH1 0x1
PUSH1 0x2
ADD
```

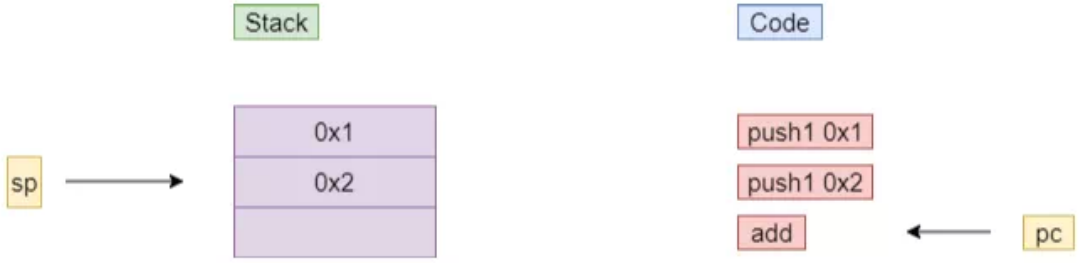
为了读者更好了解其概念，这里精简为上述3条语句，但实际的字节码可能更复杂，且会掺杂SWAP和DUP之类的语句。我们可以看到，在上述代码中，包含两个指令：PUSH1和ADD，它们的含义如下：

- PUSH1：将数据压入栈顶。
- ADD：POP两个栈顶元素，将它们相加，并压回栈顶。

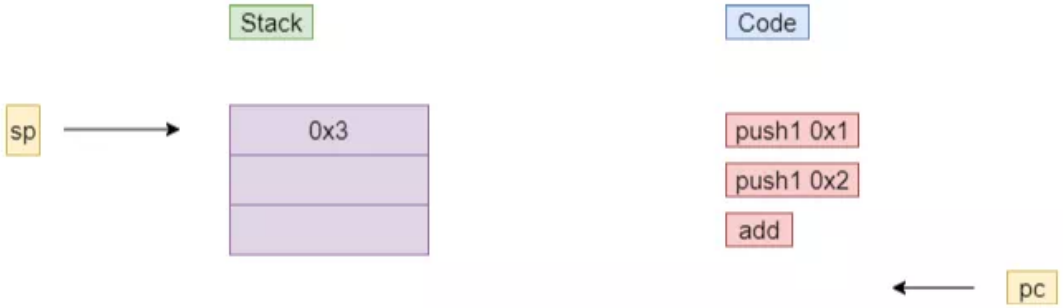
这里用半动画的方式解释其执行过程。下图中，sp表示栈顶指针，pc表示程序计数器。当执行完push1 0x1后，pc和sp均往下移：



类似地，执行push1 0x2后，pc和sp状态如下：



最后，当add执行完后，栈顶的两个操作数都被弹出作为add指令的输入，两者的和则会被压入栈：



存储探究

在开发过程中，我们常会遇到令人迷惑的memory修饰符；阅读开源代码时，也会看到各种直接针对内存进行的assembly操作。不了解存储机制的开发者遇到这些情况就会一头雾水，所以，这节将探究EVM的存储原理。在前文《智能合约编写之Solidity的基础特性》中我们介绍过，一段Solidity代码，通常会涉及到局部变量、合约状态变量。而这些变量的存储方式存在差别，下面代码表明了变量与存储方式之间的关系。

```
contract Demo{
    //状态存储
    uint private _state;

    function set(uint state) public {
        //栈存储
```

(continues on next page)

(续上页)

```
        uint i = 0;
        //内存存储
        string memory str = "aaa";
    }
}
```

栈

栈用于存储字节码指令的操作数。在Solidity中，局部变量若是整型、定长字节数组等类型，就会随着指令的运行入栈、出栈。例如，在下面这条简单的语句中，变量值1会被读出，通过PUSH操作压入栈顶：

```
uint i = 1;
```

对于这类变量，无法强行改变它们的存储方式，如果在它们之前放置memory修饰符，编译会报错。

内存

内存类似java中的堆，它用于储存"对象"。在Solidity编程中，如果一个局部变量属于变长字节数组、字符串、结构体等类型，其通常会被memory修饰符修饰，以表明存储在内存中。

本节中，我们将以字符串为例，分析内存如何存储这些对象。

1. 对象存储结构

下面将用assembly语句对复杂对象的存储方式进行分析。assembly语句用于调用字节码操作。mload指令将被用于对这些字节码进行调用。mload(p)表示从地址p读取32字节的数据。开发者可将对象变量看作指针直接传入mload。在下面代码中，经过mload调用，data变量保存了字符串str在内存中的前32字节。

```
string memory str = "aaa";
bytes32 data;
assembly{
    data := mload(str)
}
```

掌握mload，即可用此分析string变量是如何存储的。下面的代码将揭示字符串数据的存储方式：

```
function strStorage() public view returns(bytes32, bytes32){
    string memory str = "你好";
    bytes32 data;
    bytes32 data2;
    assembly{
        data := mload(str)
        data2 := mload(add(str, 0x20))
    }
    return (data, data2);
}
```

data变量表示str的0~31字节，data2表示str的32~63字节。运行strStorage函数的结果如下：

```
0: bytes32: 0x0000000000000000000000000000000000000000000000000000000000000006
1: bytes32: 0xe4bda0e5a5bd0000000000000000000000000000000000000000000000000000
```

可以看到，第一个数据字得到的值为6，正好是字符串"你好"经UTF-8编码后的字节数。第二个数据字则保存的是"你好"本身的UTF-8编码。熟练掌握了字符串的存储格式之后，我们就可以运用assembly修改、拷贝、拼接字符串。读者可搜索Solidity的字符串库，了解如何实现string的concat。

2. 内存分配方式

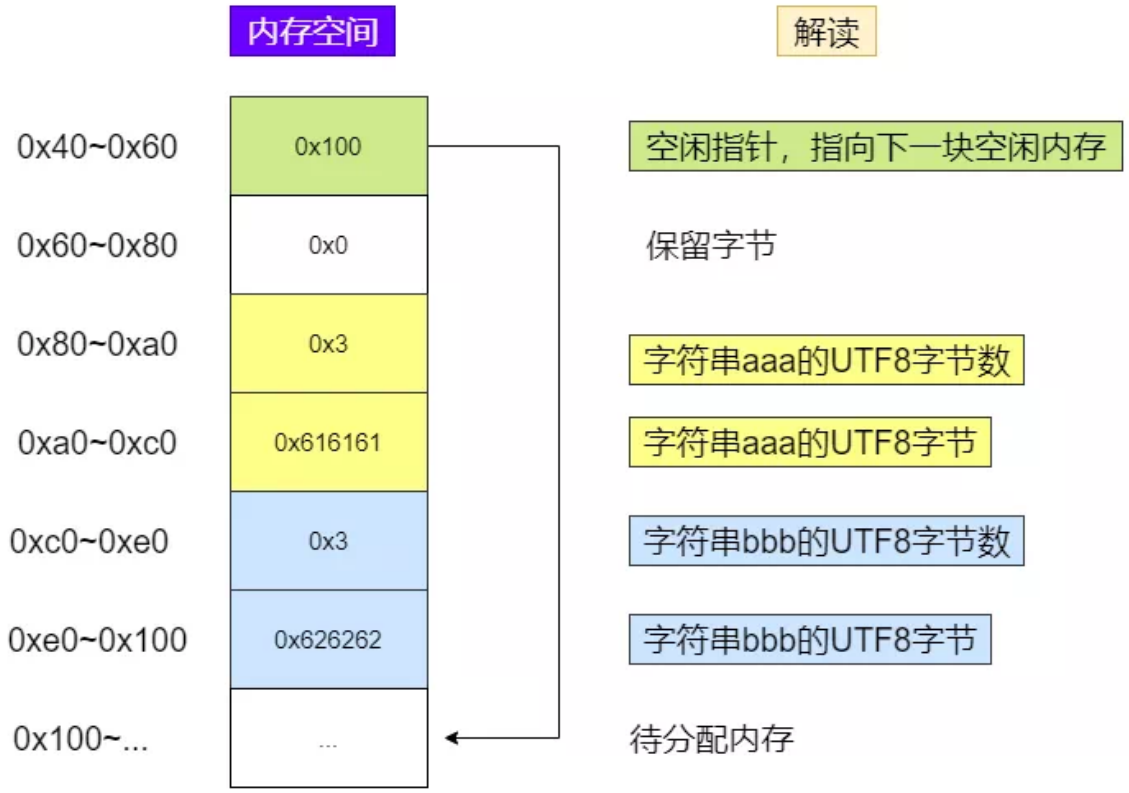
既然内存用于存储对象，就必然涉及到内存分配方式。memory的分配方式非常简单，就是顺序分配。下面我们将分配两个对象，并查看它们的地址：

```
function memAlloc() public view returns(bytes32, bytes32){
    string memory str = "aaa";
    string memory str2 = "bbb";
    bytes32 p1;
    bytes32 p2;
    assembly{
        p1 := str
        p2 := str2
    }
    return (p1, p2);
}
```

运行此函数后，返回结果将包含两个数据字：

```
0: bytes32: 0x0000000000000000000000000000000000000000000000000000000000000080
1: bytes32: 0x00000000000000000000000000000000000000000000000000000000000000c0
```

这说明，第一个字符串str1的起始地址是0x80，第二个字符串str2的起始地址是0xc0，之间64字节，正好是str1本身占据的空间。此时的内存布局如下，其中一格表示32字节（一个数据字，EVM采用32字节作为一个数据字，而非4字节）：



- 0x40~0x60：空闲指针，保存可用地址，本例中是0x100，说明新的对象将从0x100处分配。可以用mload(0x40)获取到新对象的分配地址。
- 0x80~0xc0：对象分配的起始地址。这里分配了字符串aaa
- 0xc0~0x100：分配了字符串bbb
- 0x100~...：因为是顺序分配，新的对象将会分配到这里。

状态存储

顾名思义，状态存储用于存储合约的状态字段。从模型而言，存储由多个32字节的存储槽构成。在前文中，我们介绍了Demo合约的set函数，里面0x0表示的是状态变量_state的存储槽。所有固定长度变量会依序放到这组存储槽中。对于mapping和数组，存储会更复杂，其自身会占据1槽，所包含数据则会按相应规则占据其他槽，比如mapping中，数据项的存储槽位由键值k、mapping自身槽位p经keccak计算得来。从实现而言，不同的链可能采用不同实现，比较经典的是以太坊所采用的MPT树。由于MPT树性能、扩展性等问题，FISCO BCOS放弃了这一结构，而采用了分布式存储，通过rocksdb或mysql来存储状态数据，使存储的性能、可扩展性得到提高。

结语

本文介绍了Solidity的运行原理，运行原理总结如下。首先，Solidity源码会被编译为字节码，部署时，字节码会以交易为载体在网络间确认，并在节点上形成合约；合约函数调用，如果是交易类型，会经过网络确认，最终由EVM执行。EVM是栈式虚拟机，它会读取合约的字节码并执行。在执行过程中，会与栈、内存、合约存储进行交互。其中，栈用于存储普通的局部变量，这些局部变量就是字节码的操作数；内存用于存储对象，采用length+body进行存储，顺序分配方式进行内存分配；状态存储用于存储状态变量。理解Solidity的运行方式及其背后原理，是成为Solidity编程高手必经之路。

智能合约测试的技巧与实践

作者：毛嘉宇 | FISCO BCOS 核心开发者

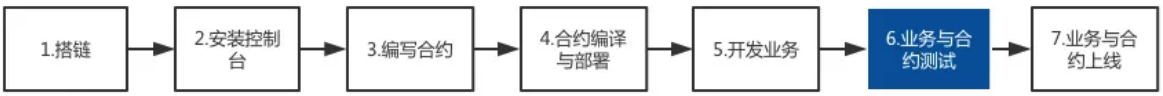
前言

区块链的发展伴随着信息安全的话题。在Solidity不长的发展历程中，发生过不止一次骇人听闻、影响深远的安全攻击事件，给一些机构和组织造成不可挽回的损失。『祸乱生于疏忽，测试先于交付』，如果这些缺陷和漏洞在测试环节被发现，就能有效避免损失。测试是智能合约开发与交付过程中至关重要的一环。其能有效检验实际结果是否符合设计预期，有助识别错误、查缺补漏。同时，高质量、可复用的测试，还有助于提升整体开发效率。前文介绍了Solidity的前世今生、语法特性、设计模式、编程攻略和底层原理。作为系列终篇，本文将重点介绍和分享Solidity的测试场景、方法和实践。

前置准备

进入测试前，需要完成以下步骤：链的搭建、控制台安装、智能合约开发、智能合约编译与部署、利用Java等SDK开发一个应用。

详细准备工作可参考《FISCO BCOS零基础入门，五步轻松构建应用》和FISCO BCOS官方文档，此处不再赘述。FISCO BCOS官方文档地址。



测试环境

FISCO BCOS提供控制台、WeBASE-Front和SDK代码测试，这三种环境适用于不同测试场景：

- 控制台：提供命令行交互界面，通过在控制台创建合约和输入调用、查询指令，来进行简单调试。适用于非常简单的合约。
- WeBASE-Front：提供可视化交互界面以及简易的IDE环境。适用于业务逻辑并不复杂的合约，更推荐开发人员进行一些调试。

- SDK：例如集成Java Web3sdk，创建一个Java项目，并编写应用和测试代码。适用于对智能合约质量要求较高、要求测试案例可复用、业务逻辑复杂或要求持续集成的场景。

控制台测试

FISCO BCOS 2.0及以上版本提供了方便易用的命令行终端、『开箱即用』的区块链利器，详情可参考《FISCO BCOS 控制台详解，飞一般的区块链体验》。接下来将通过一个合约示例来讲解如何运用控制台进行测试。首先，我们写一个HelloWorld合约：

```
pragma solidity ^0.4.25;

contract HelloWorld{
    string public name;
    constructor() public{
       name = "Hello, World!";
    }

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

然后，将这个合约命名为HelloWorld.sol，并保存到控制台所在的contracts/solidity/路径下。随后，执行./start.sh 启动控制台并部署合约：

```
=====================================================================================================
Welcome to FISCO BCOS console(1.0.8)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______   _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       \
↪\
| $$$$$$$$\$$$$$$|  $$$$$$$|  $$$$$$$|  $$$$$$$\    | $$$$$$$$|  $$$$$$$|  $$$$$$$|  $$$$$
↪$$\
| $$__      | $$ | $$___\$$| $$    \$$| $$  | $$    | $$__/ $$| $$    \$$| $$  | $$| $$___\
↪$$
| $$  \     | $$  \$$    \ | $$      | $$  | $$    | $$    $$| $$      | $$  | $$\$$    \
↪\
| $$$$$     | $$  _\$$$$$$\| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$$\
| $$       _| $$_|  \__| $$| $$__/  | $$__/ $$    | $$__/ $$| $$__/  | $$__/ $$|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

=====================================================================================================
[group:1]> deploy HelloWorld
contract address: 0x34e95689e05255d160fb96437a11ba97bb31809f

[group:1]>
```

合约被成功部署后就可以开始测试了。我们先打印这个合约中name的值，然后修改为新的值，最后重新查询name中的值。

```
[group:1]> call HelloWorld 0x34e95689e05255d160fb96437a11ba97bb31809f name
Hello, World!

[group:1]> call HelloWorld 0x34e95689e05255d160fb96437a11ba97bb31809f set "Hello,␣
↪test!"
transaction hash:␣
↪0x72c8a95b651fb5d12c44b69624d5213d58af1509f00920757fce94d019b5eae8
```

(continues on next page)

(续上页)

```
[group:1]> call HelloWorld 0x34e95689e05255d160fb96437a11ba97bb31809f name
Hello, test!

[group:1]>
```

上述例子演示了如何使用控制台部署与调试合约。控制台设计简洁优雅，使用体验如丝般顺滑。不过，在处理一些复杂场景，例如需要切换外部账户，或通过可视化界面进行操作时，WeBASE-Front当仁不让地扛起大旗。

WeBASE-Front测试

WeBASE-Front为开发者提供了运行核心信息的可视化操作、Solidity开发的IDE环境以及私钥管理功能，让大家更便捷地开启区块链之旅。关于WeBASE-Front的介绍可参考《WeBASE节点前置组件功能解析》与《安装部署说明》。

接下来，我们演示一个需要切换外部账户的测试案例，以下是合约代码：

```
pragma solidity ^0.4.25;

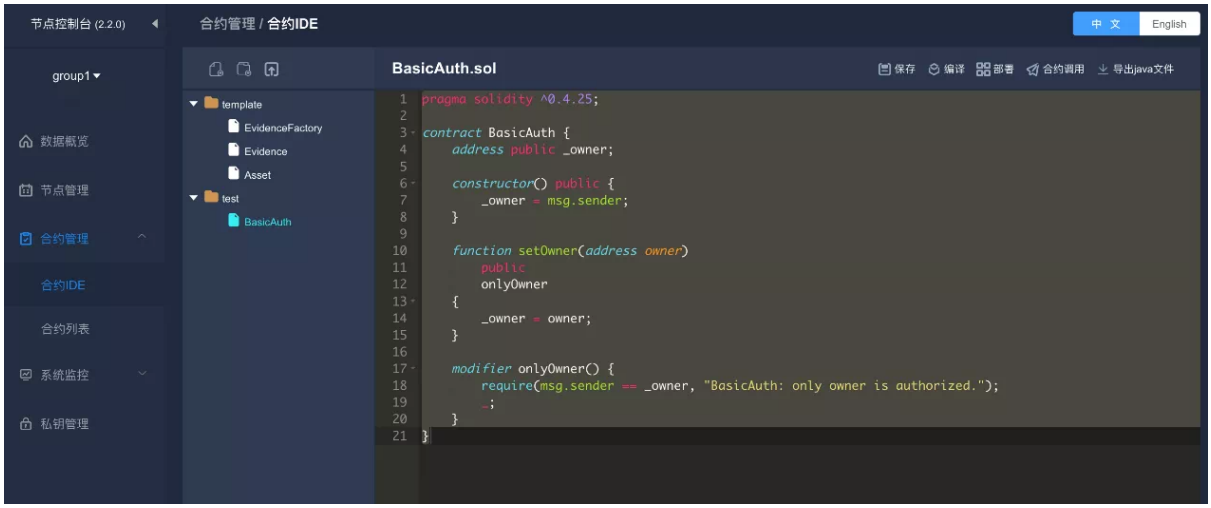
contract BasicAuth {
    address public _owner;

    constructor() public {
        _owner = msg.sender;
    }

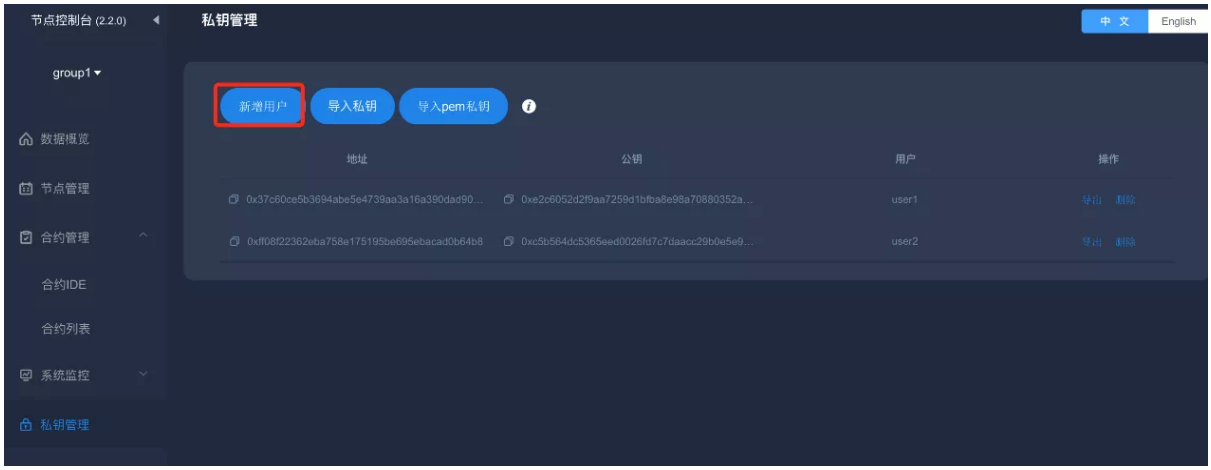
    function setOwner(address owner)
        public
        onlyOwner
{
        _owner = owner;
    }

    modifier onlyOwner() {
        require(msg.sender == _owner, "BasicAuth: only owner is authorized.");
        _;
    }
}
```

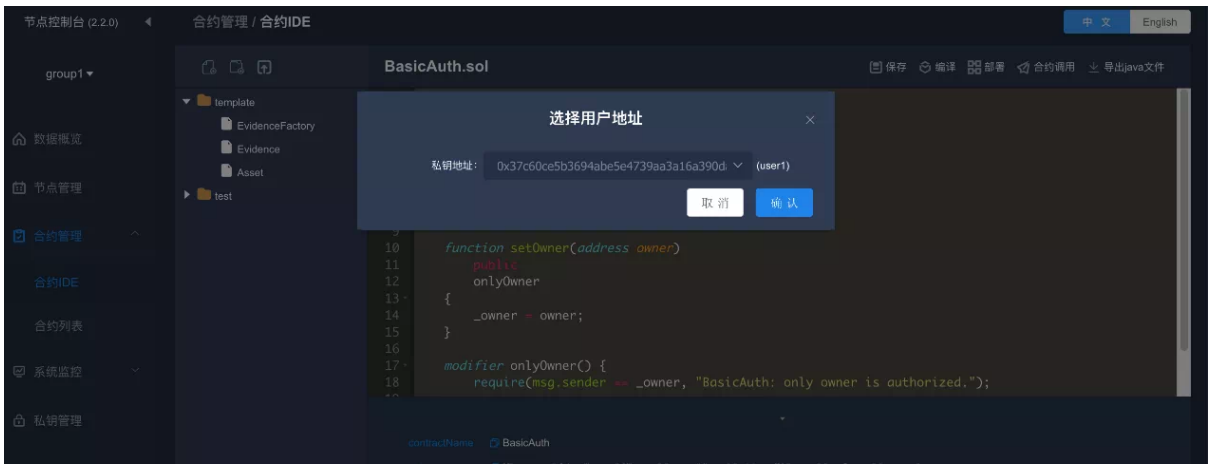
在这个示例中，合约owner被自动赋值为合约部署者。修饰器onlyOwner决定了setOwner函数只能由_owner用户来发起。在合约管理-合约IDE中，创建test文件夹，并复制合约代码：



然后，点击私钥管理-新增用户，创建user1和user2两个用户。



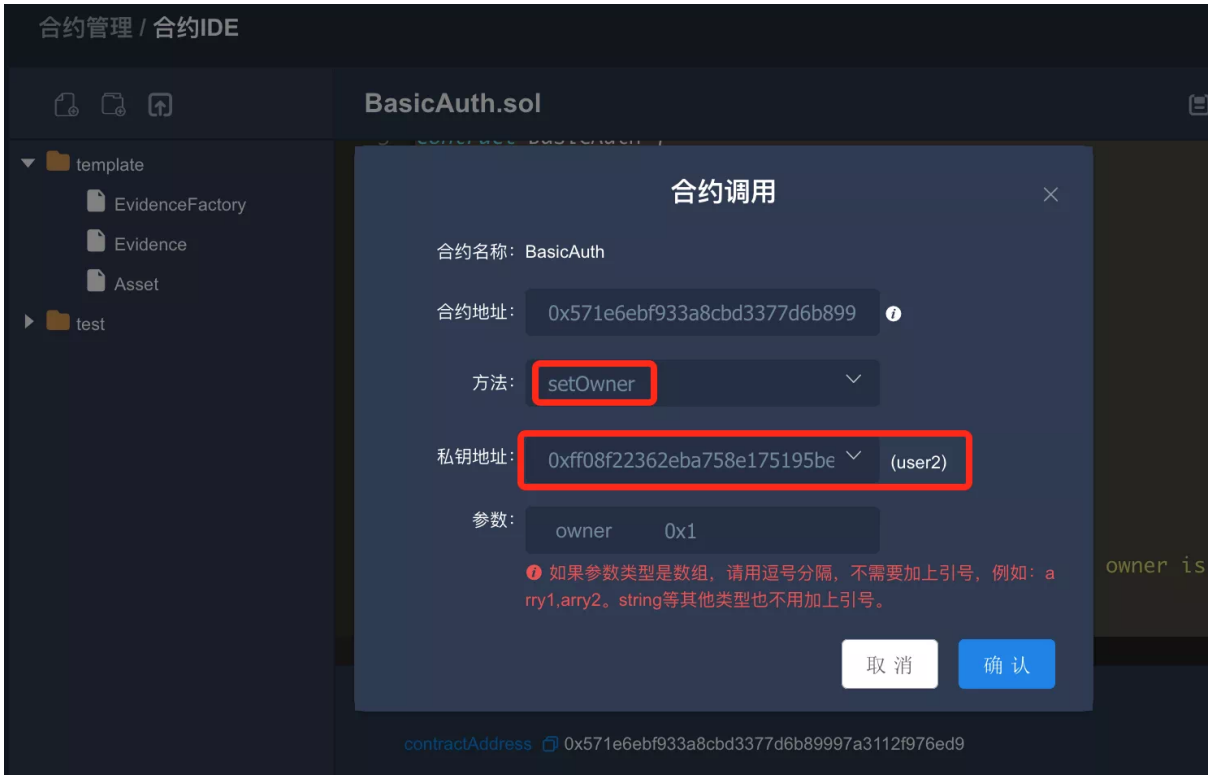
这时，选择部署合约，会弹出用户地址窗口选项，选择user1。



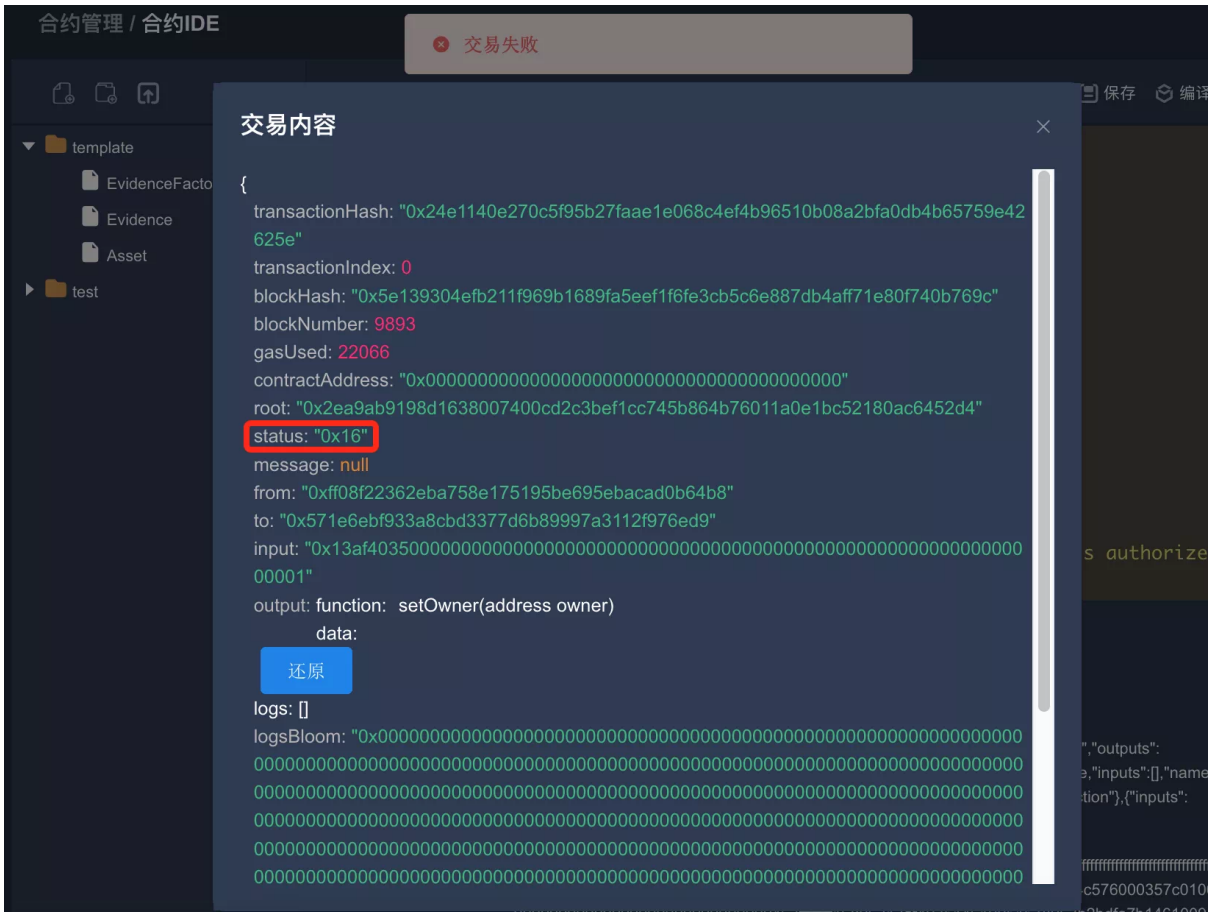
合约部署成功后，会展示合约地址、合约名、abi、合约binary等系列信息：

点击合约调用，弹出调用窗口，『方法』下拉框里包含了所有可调用的函数列表，选择_owner函数，点击确认，就可返回该合约的_owner变量值。

现在，我们来测试setOwner()函数。如上所述，本合约的_owner是user1，通过切换用户user2来调用这个函数，预期结果是调用失败。我们选择setOwner方法，选择私钥地址为user2：



正如预期，这个函数的调用失败了：


合约管理 / 合约IDE
交易失败
交易内容
{
transactionHash: "0x24e1140e270c5f95b27faae1e068c4ef4b96510b08a2bfa0db4b65759e42625e"
transactionIndex: 0
blockHash: "0x5e139304efb211f969b1689fa5eef1f6fe3cb5c6e887db4aff71e80f740b769c"
blockNumber: 9893
gasUsed: 22066
contractAddress: "0x0000000000000000000000000000000000000000"
root: "0x2ea9ab9198d1638007400cd2c3bef1cc745b864b76011a0e1bc52180ac6452d4"
status: "0x16"
message: null
from: "0xff08f22362eba758e175195be695ebacad0b64b8"
to: "0x571e6ebf933a8cbd3377d6b89997a3112f976ed9"
input: "0x13af4035000000000000000000000000000000000000000000000000000000000000001"
output: function: setOwner(address owner)
data:
还原
logs: []
logsBloom: "0x0000000000000000000000000000000000000000000000000000000000

SDK测试

在系统测试中，需要遵循经典的AIR实践原则：

- Automatic：测试应该被全自动执行，这也是实现持续集成的前提。
- Independent：测试案例彼此保持独立，不存在相互依赖和调用。
- Repeatable：测试案例必须可被复用。可以跨越软硬件环境，被重复执行。

要满足以上甚至更多测试实践原则，使用控制台或WeBASE-Front的方式就显得有些力不从心，而集成SDK并编写测试代码的方式更值得推荐。这种方式虽然前期花费时间较长、成本较高；但测试后期能大大降低重复的工作量，显著提升整体测试效率。这也符合IT公司现行的软件测试实践规范。一般这部分代码由公司开发测试人员或质量保证（QA）工程师编写。但在很多公司中，这部分代码由开发人员编写。良好的测试代码，能提升代码质量，降低代码重构难度，提高开发效率。

FISCO BCOS提供了多语言SDK，如Java、Python、Node.js等，其中最为成熟与常用的是Java版Web3SDK。《手把手教你在Eclipse中使用JavaSDK》详细介绍了在Eclipse创建新工程以及将已经提供的示例项目导入Eclipse。在Java开发实践中，Springboot使用更为深入人心，FISCO BCOS也提供了相应使用案例，相关配置文档可以参考：《Spring Boot项目配置》，通过spring-boot-starter开发者可快速下载示例工程，并导入偏好的IDE中。

配置工程链基本环境后，下面将以上述spring-boot-starter工程为例，介绍具体测试步骤和要点。

1. 编写合约：HelloWorld合约。
2. 编译智能合约，将其转为Java文件，从上述WeBASE-Front中，也能编译并导出Java文件。
3. 将编译生成的Java文件导入工程中，如HelloWorld.java。
4. 基于上述文件，调用合约功能和接口，编写相关测试案例，例如：ContractTest。
5. 基于Spring提供的gradle插件，我们可以通过"./gradlew test"命令来运行所有测试案例。
6. 如果需要持续集成，可以在配置和初始化FISCO BCOS后，将步骤5命令加入自动化脚本中。

接下来，我们将具体分析核心测试代码ContractTest：

```
@Test
public void deployAndCallHelloWorld() throws Exception {
    // deploy contract
    HelloWorld helloWorld = HelloWorld
            .deploy(web3j, credentials, new StaticGasProvider(GasConstants.GAS_
↪PRICE, GasConstants.GAS_LIMIT))
            .send();
    Assert.assertNotNull(helloWorld);
    // call set function
    helloWorld.set("Hello, World!").send();
    // call get function
    String result = helloWorld.get().send();
    Assert.assertTrue("Hello, World!".equals(result));
 }
```

- 第4行中，HelloWorld合约被部署。为符合独立性原则，建议在每个测试案例中部署独立的合约，以避免测试案例执行顺序对正常测试造成干扰。需要模拟合约依赖性的情况除外。
- 第9行和第11行，分别调用了set和get。为符合可重复性原则，该测试案例必须设计成幂等，即在任意的软硬件环境下，测试案例预期结果都是一致的。
- 第7行和第12行，使用了Junit框架提供的断言方法，用来判断智能合约执行结果是否符合预期。

值得一提的是，在Java SDK中，任意交易上链后，会获得一个回执TransactionReceipt对象，该对象中包含了返回状态和错误信息（如果交易成功，则message为null），可通过这个状态判断交易是否正常，例如：

```
TransactionReceipt tr = helloWorld.set("Hello, World!").send();
Assert.assertEquals(tr.getMessage(), "0x0", tr.getStatus());
```

以上基于Springboot提供的测试特性，实现了合约测试，如果需要了解更多细节，请参考Spring的相关文档。

测试类型

如传统软件，智能合约的测试同样可分为功能测试、非功能测试、安全性测试和回归测试，下面将对这些测试作逐一介绍。

功能测试

功能测试包括但不限于单元测试、集成测试、冒烟测试、用户验收测试。除用户验收测试外，其他测试都可通过开发或测试人员编写代码实现。智能合约测试重要目的之一是检测合约代码正确性，在预定输入值前提下，检查输出值是否符合预期。

上文我们介绍了控制台、WeBASE-Front和SDK三种测试环境。在一些逻辑复杂的智能合约中，测试难点之一在于构造测试案例。在这种场景下，使用智能合约可以更好地模拟和构造测试数据，直接使用Solidity编写智能合约更为原生、友好。

最后，测试并不游离于智能合约开发之外，而是其中重要的一环，测试也需遵循依赖关系原则，即开发者在进行开发时需要考虑智能合约的『可测试性』。例如，假设测试代码完全使用SDK编写，那智能合约的修改就可能导致测试代码需要做出对应变更，这会对测试效果造成影响，提升测试成本。基于软件设计『不可依赖多变部分』的原则，可测试性同样不能依赖于多变部分。

为了解决上述问题，我们在智能合约层引入测试代码。这些代码仅被设计为测试组件，不会发布到线上环境，以此将测试案例改动影响解耦并封装到智能合约层。测试合约作为独立组件，支持开发和测试环节。

测试组件中首先可以抽象和定义一些测试的工具合约，例如：

```
pragma solidity 0.4.25;

library LibAssert {
    event LogTest(bool indexed result, string message);

     function equal(uint8 a, uint8 b, string message) internal returns (bool␣
↪result) {
        result = (a == b);
        log(result, message);
    }

    function notEqual(uint8 a, uint8 b, string message) internal returns (bool␣
↪result) {
        result = (a != b);
        log(result, message);
    }

    function log(bool result, string message) internal {
        if(result)
            emit LogTest(true, "");
        else
            emit LogTest(false, message);
    }
}
```

这是一个最简单的测试合约库，提供了equal和notEqual的方法来判断uint8变量，开发者可基于此工具扩展自己的测试工具合约。

其次，依托于工具合约，我们可以编写独立的测试合约案例。

下面将以上述Uint8MathTest为例，演示如何使用此工具合约，首先，我们定义一个TestDemo合约：

```
pragma solidity ^0.4.25;
import "./Uint8MathTest.sol";
import "./LibAssert.sol";
contract TestDemo {
  using LibAssert for uint8;
  function testAdd() external returns (bool result) {
    uint8 a = 1;
    uint8 b = 1;
    Uint8MathTest math = new Uint8MathTest();
    uint8 c = math.add(a,b);
    uint8 expectedValue = 2;
    result = expectedValue.equal(c, "Not equal");
  }
}
```

然后，在控制台中运行这个测试案例：

```
[group:1]> deploy TestDemo
contract address: 0xd931b41c70d2ff7b54eb9b2453072f9b1a4dc05c

[group:1]> call TestDemo 0xd931b41c70d2ff7b54eb9b2453072f9b1a4dc05c testAdd
transaction hash:␣
↪0xe569e5d8eae1b949a0ffe27a60f0b4c8bd839f108648f9b18879833c11e94ee4
-------------------------------------------------------------------------------------------
↪----------
Output
function: testAdd()
return type: (bool)
return value: (true)
-------------------------------------------------------------------------------------------
↪----------
Event logs
-------------------------------------------------------------------------------------------
↪----------
```

除了依赖自定义的智能合约测试代码，也可以用智能合约本身来编写测试案例。在SDK层，我们只需实现TestDemo中的测试函数代码。即使未来测试逻辑改动，也无需改动SDK代码，从而保证了测试代码的健壮性。合约测试代码组件需贯彻智能合约在整体设计中所遵循的设计原则。

非功能测试

非功能测试主要包括性能测试、容量测试、可用性测试等。由于智能合约运行于FISCO BCOS的底层节点，容量测试和可用性测试更多与底层节点相关，因此，对用户而言，非功能测试的重点就在性能测试上。

虽然，我们可以使用系列通用性能测试工具对智能合约进行测试，但是，实际压测过程中会存在一些痛点，例如：

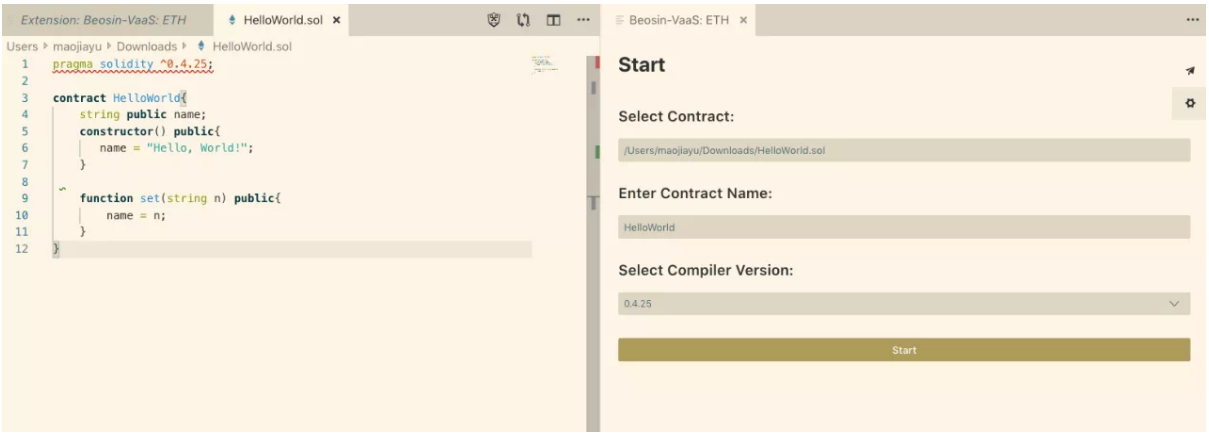
- 针对具体合约场景测试，存在大量且重复的测试代码，费时费力；
- 性能指标缺乏统一度量，无法横向对比；
- 展示结果不够直观。

为了解决以上痛点，FISCO BCOS适配了区块链测试专业工具Caliper，让用户可以优雅地进行性能测试。更多细节和内容请参考：《性能测试工具Caliper在FISCO BCOS平台中的实践》。

安全性测试

智能合约上线前需要进行严格的安全性测试。安全性测试的方式包括：公开智能合约并发布悬赏、聘请专门的智能合约安全机构对合约进行检测和评估、使用智能合约专用的工具进行审计等。大家可以视自

身合约的重要性及逻辑复杂程度选择对应的安全性测试级别。对于个人开发者或非重大业务的智能合约，选择免费的智能合约工具检测即可。下面以VS Code为例，演示如何使用智能合约安全插件对合约进行安全性检测。打开VS Code，在其插件库中搜索Beosin-VaaS: ETH，选择安装。随后，打开智能合约文件，右键选择Beosin-VaaS:ETH选项，在弹出的窗口中选择当前合约代码版本。安装完成后，会自动弹出如下界面：



点击start即可对所选合约进行安全检测。以下是对HelloWorld合约的检测结果：

```
start compile!
compile over!
start deploy!
deploy over!
start execute function name: set(string)
execute function name : set(string) end!
No problem was found in the contract
---end---
```

当我们对Uint8MathTest合约进行检测时，会发现报错：

```
####### Compile Problem #######
file: /Users/maojiayu/Downloads/Uint8MathTest.sol
title: Compile Problem
description:  Function state mutability can be restricted to pure
code:   function add(uint8 a, uint8 b) public returns (uint8 c) {
warningType: Info
line: 5
```

通过报错信息，我们可以发现该函数既没读取也没使用区块链上的状态，符合pure修饰符的使用条件，建议使用『pure』关键字来修饰。安全性测试是合约测试过程中不可或缺的一环，需高度重视。在严格的工程实践中，合约上生产前必须由相关的测试负责人出具安全性检测报告。此外，开发人员可以收集和总结Solidity安全编程规则和漏洞风险提示，并动态更新维护。在交付测试前，开发人员可组织代码审查和走查，依据总结，集中、逐项进行风险排查。安全性测试是检测和评价智能合约是否安全的必备环节，但不能将合约安全性完全寄望于测试。更重要的是，在设计、开发和测试的各个阶段，都需要用户时刻保持安全意识，建立和培养安全编码习惯。合约发布上线后，也不能放松警惕，时刻关注业界最新的安全警告，并定期、动态对所有代码进行检测和扫描。

回归测试

回归测试通常包括执行持续集成的自动化测试案例以及合约上线前的人工测试。上述SDK以及智能合约测试案例可有效覆盖测试案例所执行的路径和范围。自动化的回归测试有助于快速检测并发现问题，节省大量重复人力，保证合约经历系列修改和测试后，不偏离主干功能。同样，开发者可使用Jenkins、Travis等工具进行构建和测试。除了执行个别自动化测试无法执行的案例外，人工测试更多地是确认之前修改的代码是否符合预期，智能合约的运行是否符合当初设计。此外，在合约最终发布前，人工测试还起到审核检查作用，故而一定要认真对待。

测试要点

无论何种类型的系统测试，均需特别注意以下测试要点。

- 注意边界值的测试，例如数字溢出、特殊取值、循环边界值等。
- 注意检查智能合约的实现是否符合预期，其运行逻辑和流程是否与设计一致。
- 除了正常流程外，还需要模拟和测试在各种异常环境，甚至是极端环境下，智能合约运作是否正常，能否达到预期的处理结果。
- 围绕智能合约不变的业务逻辑，忽略变化值，进行对应测试。

边界值

边界值测试是一种常见的软件测试方法。据统计，绝大多数的bug发生在输入或输出范围的边界上，而非其内部。我们来演示一个典型的数值溢出测试案例。如下简单的合约，实现了两个uint8数值加法，并返回了结果：

```
pragma solidity 0.4.25;

contract Uint8MathTest {
    function add(uint8 a, uint8 b) public returns (uint8 c) {
        c = a + b;
    }
}
```

这时，我们需要设计一个边界值的测试案例。已知uint8的取值范围在[0,28)即[0,256)之间，以此设计一个溢出的测试案例：（a=255, b=1），测试结果如下：

```
[group:1]> deploy Uint8MathTest
contract address: 0xd8a765995c58eb8da103bdcc2c033c0acb81e373

[group:1]> call Uint8MathTest 0xd8a765995c58eb8da103bdcc2c033c0acb81e373 add 255 1
transaction hash:␣
↪0x0583185839bc52870ac57cdfd00c0c18840c2674d718b4cc3cb7bc1ef4c173e0
-------------------------------------------------------------------------------------------
↪----------
Output
function: add(uint8,uint8)
return type: (uint8)
return value: (0)
-------------------------------------------------------------------------------------------
↪----------
```

可以看到最后结果变为0，而不是预期的256，显然，这是因为结果发生了溢出。当然，在使用边界值测试方法取值时，并非只取有效边界值，还应包含无效边界值。边界值测试可帮助查出更多错误和缺陷。

是否符合预期

在建筑公司交付房屋后，开发商会依据设计标准检验房屋建造是否合格。同样，除了检查合约执行结果是否正确外，在测试中还需检查合约交互、操作、数据流向、性能表现等各方面实现是否符合设计。举个例子，在金融业场景中，我们会使用Solidity开发具体链上业务逻辑。其中，不可避免与金额打交道，就需要测试对应资产规模是否符合设计标准。

在需求和设计阶段，我们会预估运行在链上的资产规模并作设计要求，例如要求某个供应链金融区块链业务系统能在千亿资产规模下正常运行。假设在智能合约中，定义金额的字段是uint32，可以计算其最大能表示的值为232即不到43亿。虽然在智能合约发布上线后大多数时间内，该问题不会造成任何影响；但是，一旦业务快速增长，当资产规模超过43亿后，就会出现严重故障。这类问题非常隐蔽，因此，需要更加全面、仔细的测试来提早发现和定位。

异常流程

在实际业务场景中，即便经过系列测试保证智能合约运行符合绝大多数场景需求，也可能因为疏忽或遗漏，导致在特定异常流程下，出现问题，在智能合约发布后造成了重大影响。故而，开发者需要特别重视在测试范围和内容中对异常流程的搜寻和覆盖，尽可能测试覆盖到所有的异常流程。

变与不变

使用Solidity测试与其他语言的显著不同还在于合约中的很多交易和值无法复现。正如『人的一生无法两次踏入同一条河流』，随着私钥、证书、环境等改变，其交易Hash值、区块Hash、合约地址等也会随之不同，这也给测试带来一些客观困难。另一方面，Solidity对于一致性有着严苛要求，例如对EVM执行指令语义的一致性、对外部随机变量的『排斥』等。掌握合约测试的关键在于抓住其中不变的方面。

测试技巧

作为区块链上专用编程语言，相比其他高级语言，Solidity有非常多限制。例如，缺乏成熟的测试工具支持，无法在运行时进行debug以及查看EVM虚拟机内的状态与数据。同时，Solidity缺乏独立的数据抽象层，导致无法像传统应用那样直接通过连接数据库来查看合约内详细的变量状态，只能通过在合约内增加查询调用来『曲线救国』。 当然，我们也可以采取一些技巧，尽可能规避以上问题。

如何展示合约更多内部变量?

编写Solidity代码与其他高级语言最大差别是无法进行debug。这个问题解决办法之一是打印和展示更多内部变量。

问题场景：

还是上述的HelloWorld合约，现在我们无法在set函数执行时获取原有的name值，只能通过name()函数来查询。有没有办法在set函数执行时，记录和打印出name原来的值呢?

解决方案：

定义一个event，在set函数内部进行调用。例如：

```
pragma solidity ^0.4.25;

contract HelloWorld{
    string public name;
    event LogSet(string oldValue, string newValue, address sender);
    constructor() public{
        name = "Hello, World!";
    }

    function set(string n) public{
      emit LogSet(n, name, msg.sender);
      name = n;
    }
}
```

然后，在控制台中执行，可以看到刚才定义的LogSet事件被打印出来了。我们在事件中还定义打印了msg.sender的地址，如下图所示，对应地址也被打印出来。

```
[group:1]> deploy HelloWorld
contract address: 0x3f1be1c2d885a67d3590958454daa02159e89909

[group:1]> call HelloWorld 0x3f1be1c2d885a67d3590958454daa02159e89909 set "Hello, event!"
transaction hash: 0x919678483ba0b13645701af620ddb1b54ee264d90e016af040947627c6868175
Event logs
event signature: LogSet(string,string,address) index: 0
event value: (Hello, event!, Hello, World!, 0x720c9fa40f5dec1fc182c685475169167f76e93a)
```

Event相当于提供了log日志的简单机制，当出现无法解决且需要展示合约内部状态的情况，Event提供了合适的方法和机制。

如何获得链上全量数据模型?

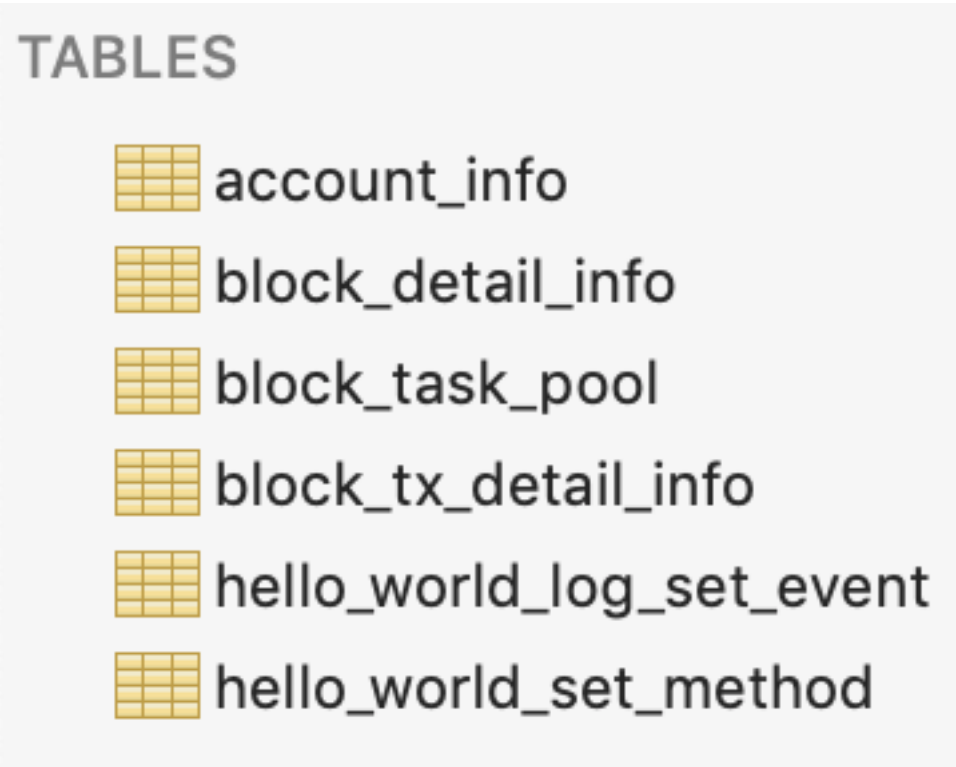
问题场景：

都说Solidity是图灵完备的语言，EVM是设计简洁的状态机，如果想要获取状态机内部每个状态和交易的『世界状态』蛛丝马迹，有什么好的解决方案?

解决方案：

Solidity缺乏独立的可对外访问的数据层，无法直接获取每笔交易或每个状态的详情。不过，我们可以通过WeBASE数据导出组件来导出所有链上数据。WeBASE-Collect-Bee可以导出区块链上的基础数据，如当前块高、交易总量等。如果正确配置了FISCO BCOS上运行的所有合约，WeBASE-Collect-Bee可以导出区块链上合约的业务数据，包括event、构造函数、合约地址、执行函数的信息等。数据导出组件WeBASE-Collect-Bee的目的在于降低获取区块链数据的开发门槛，提升研发效率。研发人员几乎无需编写任何代码，只进行简单配置，就可以把数据导出到Mysql数据库。

还是以上面的HelloWorld为例，参考安装文档：WeBASE-Collect-Bee快速部署。安装过程略过不表，导出以后会自动在数据库中生成如下的表：



数据库表的功能如下：

- account_info: 记录了所有在链上部署的合约信息，包括了合约名、合约地址、块高、上链时间等。
- blcok_detail_info: 记录了每个区块的信息，包括区块hash、交易数量、块高、上链时间等。
- block_task_pool：数据导出的任务表，记录了任务详情和状态信息。
- block_tx_detail_info：记录了每笔交易的信息，包括了交易的合约名称、函数名称、交易发送者地址、交易接收者地址、交易hash等。
- hello_world_log_set_event：数据导出会自动生成每个event的数据库表，命名规则为合约名+事件名。上面例子中定义的event所自动生成的表，包括了event中定义的所有变量和交易信息等。
- hello_world_set_method：数据导出会自动生成不同transaction的数据库表，命名规则为合约名+函数名。上面例子中定义的set函数表，包含了函数的入参、返回码和交易hash等。

hello_world_log_set_event显示了所有HelloWorld合约中，LogSetEvent事件记录的信息。

pk_id	bl...	block_timestamp	depot_updatet...	event_contract_address	_new_value	_old_value	_sender	tx_hash
1	2	2020-04-01...	2020-04-01...	0x3f1be1c2d885a67d3...	Hello, World!	Hello, event!	0x720c9fa40f5dec1fc182c6...	0x919678483ba0b13645701af62...
2	3	2020-04-01...	2020-04-01...	0x3f1be1c2d885a67d3...	Hello, event!	Bye!	0x720c9fa40f5dec1fc182c6...	0x0766db0e2c53443156582525...

hello_world_set_method显示了历史调用的所有函数信息。

pk_id	bl...	block_timestamp	contract_address	depot_updatet...	method_status	_n	tx_hash
2	2	2020-04-01...	0x3f1be1c2d885a67d3590958454daa02159e89909	2020-04-01...	0x0	Hello, event!	0x919678483ba0b13645701af62...
1	3	2020-04-01...	0x3f1be1c2d885a67d3590958454daa02159e89909	2020-04-01...	0x0	Bye!	0x0766db0e2c53443156582525...

有了上述的数据库导出状态与信息，使用区块链瞬间变得容易，测试也更加得心应手、如虎添翼。数据库会如实地导出和记录区块链上所有操作，让一切链上数据都尽在掌握中！

总结

优质的测试环节能够提升智能合约编写的质量，杜绝因为合约编写漏洞而造成的重大损失，还能使合约开发曲线趋缓，有助于提升智能合约开发效率。『翻过高山，正遇江海。行过雪原，恰逢花期。』只有经历了艰苦、严谨、细心、反复的测试周期，才能迎接智能合约发布的曙光。至此，智能合约系列也行至尾声。本教程从智能合约的概念引入，介绍了Solidity的基本语言特性、设计模式、编程攻略，再到深入到EVM核心、Solidity测试。短短几篇小文难以穷尽Solidity编程所有细节，期待社区的开发者，一起参与和分享更多知识和经验，共同建设更好的FISCO BCOS社区。

参考资料

- Spring Boot项目配置参考
- spring-boot-starter
- HelloWorld合约
- 转为Java文件
- HelloWorld.java
- ContractTest
- WeBASE-Collect-Bee快速部署

其他

FISCO BCOS的CNS合约命名服务

作者：陈宇 | FISCO BCOS 核心开发者

CNS为何而生?

FISCO BCOS原有的调用智能合约流程为：

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实现对合约的调用。

从上述合约调用流程可知，业务方调用合约之前，必须获取合约abi以及合约地址address，这是业界调用合约的通用方法。

但通过后续的用户调研，我们收集到业务方提出的如下建议：

1. 对于较长的合约abi字符串，需提供位置进行存储，从而代替业务方的自行存储；
2. 对于20字节的合约地址魔数，其丢失后将导致合约不可访问，需减少业务方的记忆成本；
3. 合约重新部署后，相关多个业务方可快速无感知地更新合约地址；
4. 易于对合约进行版本管理。

为给业务方提供更加良好的智能合约调用体验，FISCO BCOS提出CNS合约命名服务方案。

CNS如何实现?

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，方便业务方通过记忆简单的合约名称来实现对链上合约的调用。为方便业务方调用合约，SDK封装了CNS方式调用合约的接口，接口内部实现了合约地址的查找，业务方对此无感知。

信息记录

CNS记录的内容包括：合约名称、合约版本、合约地址和合约abi。其中合约abi指合约的接口说明，描述了合约字段名称、字段类型、方法名称、参数名称、参数类型、方法返回值类型。上述CNS信息以系统表的方式进行存储，账本内各节点一致，但各账本独立。CNS表定义如下：

Field	Type	IsNull	Key	Description
name	string	No	PRI	合约名称，逻辑上name和version为联合主键
version	string	No		合约版本，最大长度为40字节
address	string	No		合约地址
abi	string	YES		合约abi
id	string	No	UNI	分布式存储字段，该记录的唯一标识
status	string	No		分布式存储字段，“0”可用“1”删除
num	string	No		分布式存储字段，该记录插入/最后更新的块高

接口描述

SDK与区块链节点间的接口以合约的方式提供。该CNS合约以预编译合约的方式进行逻辑实现，声明如下接口：

```
pragma solidity ^0.4.2;
contract CNS
{
    // CNS信息上链
    function insert(string name, string version, string addr, string abi) public␣
↪returns(uint256);
    // 查询返回表中该合约所有不同version的记录，JSON格式
    function selectByName(string name) public constant returns(string);
    // 查询返回表中该合约该版本的唯一地址
    function selectByNameAndVersion(string name, string version) public constant␣
↪returns(string);
}
```

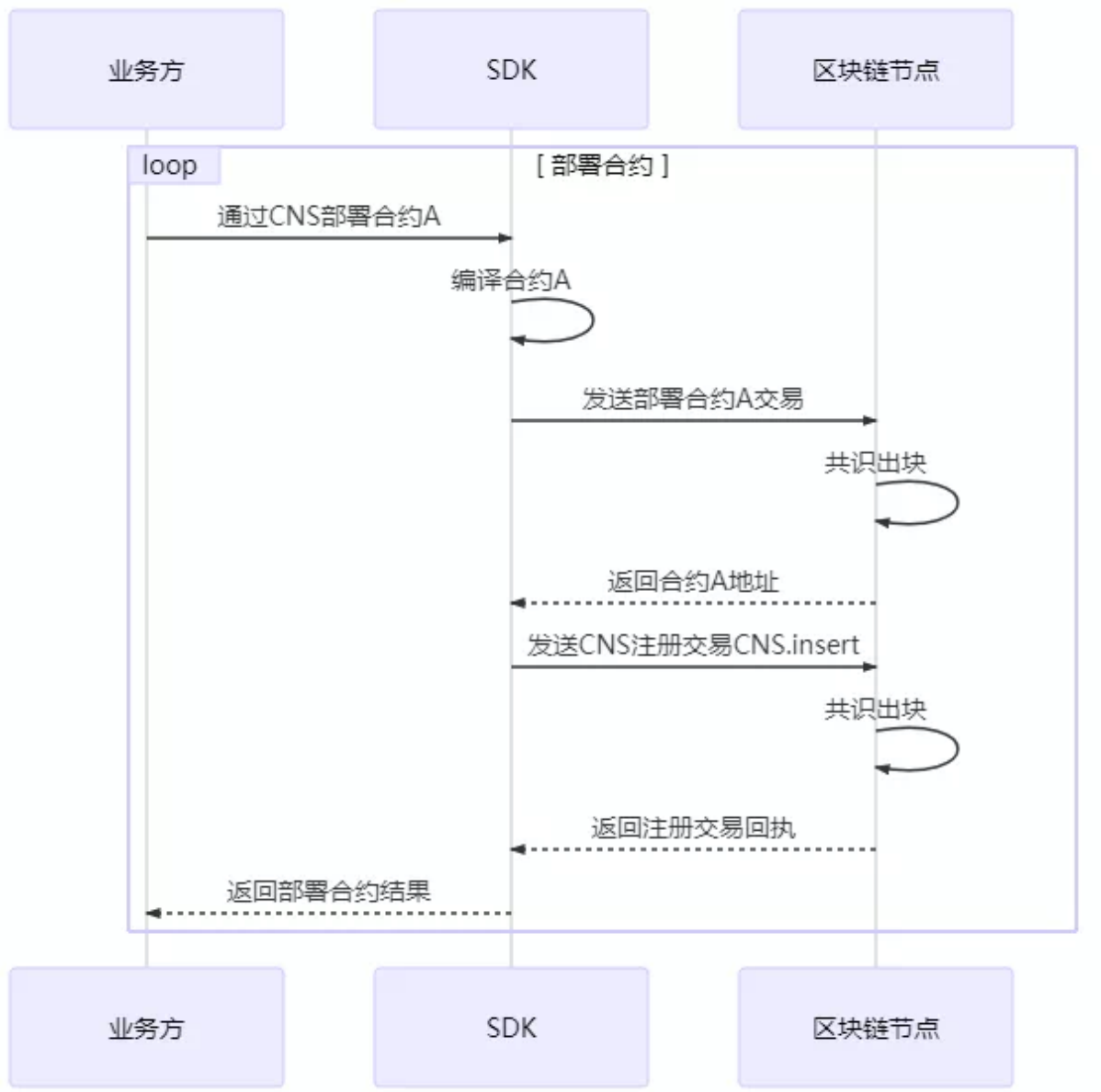
SDK提供预编译合约对应的CnsService类，实现对CNS的支持。CnsService供业务方进行调用，可以配置及查询CNS信息，其API如下：

- `String registerCns(String name, String version, String address, String abi)`：根据合约名称、合约版本、合约地址和合约abi注册CNS信息。
- `String getAddressByContractNameAndVersion(String contractNameAndVersion)`：根据合约名称和合约版本（合约名称和合约版本用英文冒号连接）查询合约地址。若缺失合约版本，默认使用合约最新版本。
- `List<CnsInfo> queryCnsByName(String name)`：根据合约名称查询CNS信息。
- `List<CnsInfo> queryCnsByNameAndVersion(String name, String version)`：根据合约名称和合约版本查询CNS信息。

交互流程

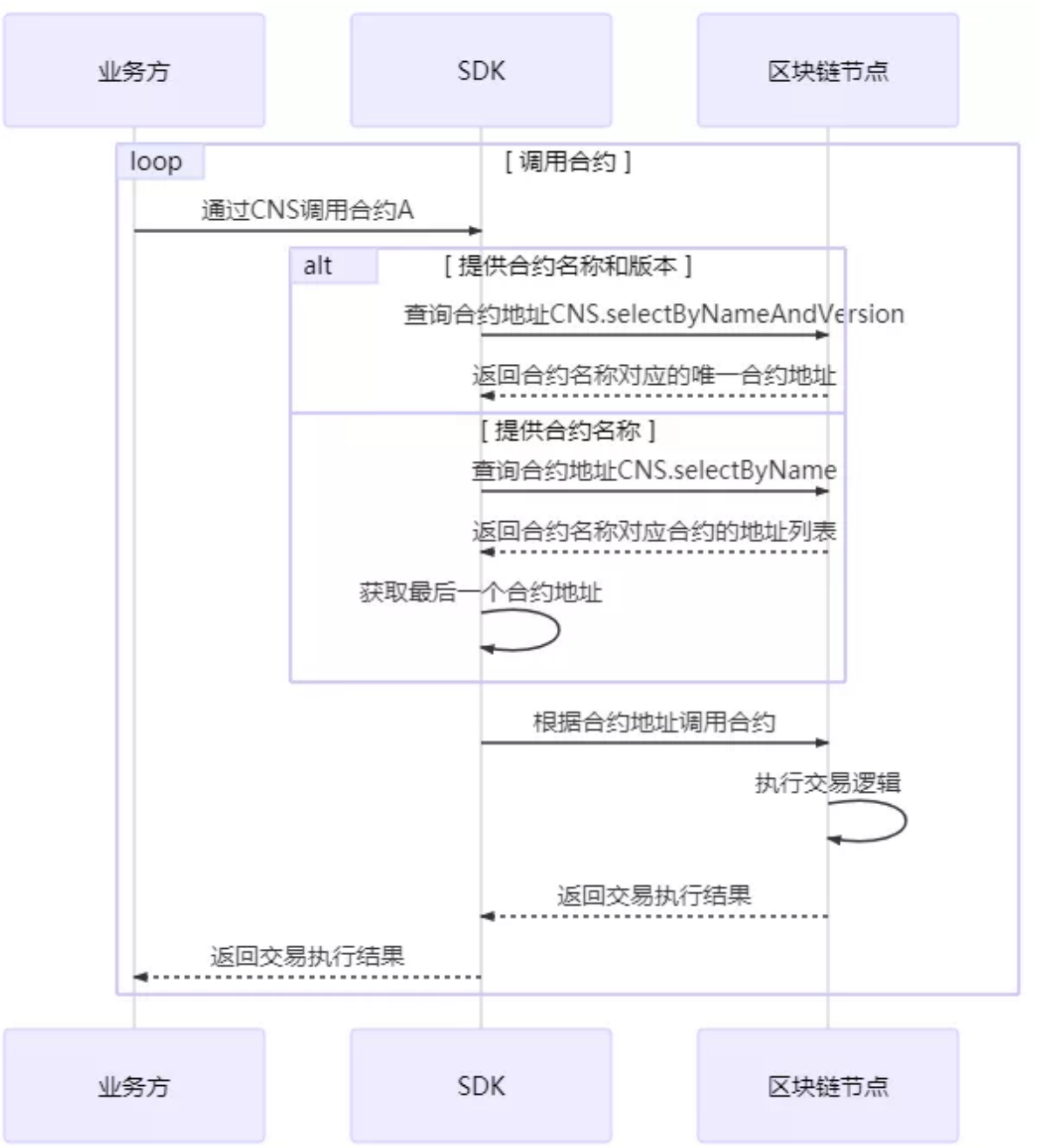
部署合约

业务方通过CNS部署合约的过程包括两个步骤，两个步骤均由SDK执行。第一个是发送部署合约的交易上链；第二个是通过发送上链交易，实现合约名称和合约地址的关联。



调用合约

SDK接收到业务方基于CNS调用合约的请求时，先行查询获取该合约名称对应的合约地址，再基于合约地址进行合约的调用。



CNS使用演示

我们以调用CnsService的控制台为例，描述CNS相关的注册、调用和查询功能。

deployByCNS

运行deployByCNS，采用部署合约。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：部署的合约版本。

示例：

```
# 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
```

queryCNS

运行queryCNS，根据合约名称和合约版本（可选参数），查询CNS表记录信息。

参数：

- 合约名称：部署的合约名称。
- 合约版本：部署的合约版本（可选）。

示例：

```
[group:1]> queryCNS HelloWorld.sol
---------------------------------------------------------------------------------------------
|                  version                  |                  address                  |
|                    1.0                    | 0x3554a56ea2905f366c345bd44fa374757fb4696a  |
|                    2.0                    | 0x07625453fb4a6459cbf14f5aa4d574cae0f17d92  |
---------------------------------------------------------------------------------------------

[group:1]> queryCNS HelloWorld 1.0
---------------------------------------------------------------------------------------------
|                  version                  |                  address                  |
|                    1.0                    | 0x3554a56ea2905f366c345bd44fa374757fb4696a  |
---------------------------------------------------------------------------------------------
```

callByCNS

运行callByCNS，采用CNS调用合约，即用合约名称直接调用合约。

参数：

- 合约名称与合约版本：合约名称与版本用英文冒号分隔，例如HelloWorld:1.0或HelloWorld.sol:1.0。

  当省略合约版本时，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。

示例：

```
# 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

# 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
```

总结

FISCO BCOS通过CNS简化了业务方调用合约的方式，也方便了业务方对合约进行管理及升级操作。同时，CNS专注于实现地址映射功能。CNS所映射的地址类型除合约地址外，还可以映射账号地址。当CNS映射账号地址时，合约abi内容为空。

FISCO BCOS的外部账户生成

作者：白兴强 | FISCO BCOS 核心开发者

账户是什么

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，得到的地址字符串被用作该账户的账户名，即账户地址。而仅有用户知晓的私钥则对应着传统认证模型中的密码。这类有私钥的账户也常被称为外部账户或账户。

FISCO BCOS中部署到链上的智能合约在底层存储中也对应一个账户，我们称这类账户为合约账户。与外部账户的区别在于，合约账户的地址是部署时确定，根据部署者的账户地址及其账户中的信息计算得出，并且合约账户没有私钥。

本文将主要介绍外部账户的生成，不讨论合约账户相关内容，关于生成的外部账户的使用方式，请参考FISCO BCOS各SDK的说明文档。

账户的使用场景

在FISCO BCOS中，账户有以下使用场景：

- SDK需要持有外部账户私钥，使用外部账户私钥对交易签名。区块链系统中，每一次对合约写接口的调用都是一笔交易，而每笔交易需要用账户的私钥签名。
- 权限控制需要外部账户的地址。FISCO BCOS权限控制模型，根据交易发送者的外部账户地址，判断是否有写入数据的权限。
- 合约账户地址唯一的标识区块链上的合约。每个合约部署后，底层节点会为其生成合约地址，调用合约接口时，需要提供合约地址。

外部账户的生成

出于方便，下文中提及外部账户，均简称为账户。FISCO BCOS提供了get_account.sh脚本和Web3SDK接口用以创建账户，同时，console和Web3SDK也支持加载创建的账户私钥，用于交易签名。用户可将账户私钥，存储为PEM或PKCS12格式的文件。其中，PEM格式使用明文存储私钥，而PKCS12格式使用用户提供的口令加密存储私钥，详情可以参考这里。

使用get_account.sh脚本生成账户（操作）

- **获取脚本**

```
curl -LO https://media.githubusercontent.com/media/FISCO-BCOS/LargeFiles/master/
↪tools/get_account.sh && chmod u+x get_account.sh && bash get_account.sh -h
```

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
```

- **生成PEM格式存储的账户私钥**

```
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
```

- **生成PKCS12格式存储的账户私钥**

```
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
```

使用Java-SDK接口生成账户

有时我们需要在代码中生成新的账户，这个时候就需要借助Java-SDK(项目名为web3SDK)提供的接口。

如下所示，Java-SDK提供生成账户、计算账户地址和获取公钥等功能，与get_account.sh脚本相比更多了对国密账户生成的支持。

```
import org.fisco.bcos.web3j.crypto.EncryptType
import org.fisco.bcos.web3j.crypto.Credentials
import org.fisco.bcos.web3j.crypto.gm.GenCredential

//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;
```

(continues on next page)

(续上页)

```
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);
```

上述接口可以直接在Java业务代码中使用，同时Java-SDK也提供了加载PEM格式或PKCS12格式存储的私钥的功能，详情请参考这里。

账户地址计算方法

FISCO BCOS的账户地址由ECDSA公钥计算得来，对ECDSA公钥计算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。

下面简要演示账户地址计算步骤:

- 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
```

- 根据私钥计算公钥，然后使用公钥计算对应的账户地址。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ecprivkey.pem对应的账户地址。

```
dcc703c0e500b653ca82273b7bfad8045d85a470
```

总结

本文简要介绍了FISCO BCOS外部账户的定义、生成以及账户地址的计算方法。未来，我们也会开放更多好用的配套组件，来帮助开发者更方便安全地管理账户。

浅谈合约ABI

作者：王章 | FISCO BCOS 核心开发者

引子

当调用合约接口时，可以向区块链发送一笔交易，并获取交易的回执，交易回执保存交易的输入参数、输出、Event log、执行状态等信息。交易回执示例如下图所示:

```
{
  "blockHash":"0xf1a2c2f521ca6033e204c8d417f88abcf604d3add273dd1ec933695c5f30c1c0",
  "blockNumber":"0x5",
  "contractAddress":"0x0000000000000000000000000000000000000000",
  "from":"0x043b617102f206b7590bcfaa111f014916dfcc57",
  "gasUsed":"0x10620",
  "input":"0x9b80b050000000000000000000000000000000000000000000000000000000000000006000000000
  0000000000000000000000000000000000000000000000000a000000000000000000000000000000000000000
  0000000000000000000000271000000000000000000000000000000000000000000000000000000000000000000
  05416c6963650000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000003426f6200000000000000000000000000000000000000000000000000
  000000000000",
  "output":"0x0000000000000000000000000000000000000000000000000000000000000000",
  "logs":[
    {
      "address":"0xb827cbe163a77681c857835bf63a83473f98fdf8",
      "data":"0x0000000000000000000000000000000000000000000000000000000000000000",
      "topics":[
        "0x8f6b9fa4d4bf04c7c1c3242d4a5c59ba22525b6761cf89e44becb27c606154bd",
        "0x81376b9868b292a46a1c486d344e427a3088657fda629b5f4a647822d329cd6a",
        "0x28cac318a86c8a0a6a9156c2dba2c8c2363677ba0514ef616592d81557e679b6",
        "0x0000000000000000000000000000000000000000000000000000000000002710"
      ]
    }
  ],
  "root":"0x6738e4cda837c39ad6dc31f5a85f25a620fa525990541d26cf60321d3bb33e5b",
  "status":"0x0",
  "to":"0xb827cbe163a77681c857835bf63a83473f98fdf8",
  "transactionHash":"0x67d9e0ca430b7dabb16460eb26b1280fb78f8c0a61d651565b1124678e3be1d9",
  "transactionIndex":"0x0"
}
```


交易回执详情如下。 在交易回执中，input、output字段可以分别表示交易的输入参数及EVM执行交易后的返回值，这些字段是接口的入参或者返回值根据合约的ABI编码生成的。

什么是合约ABI

“合约ABI是以太坊生态系统中与合约交互的标准方式，不论是外部客户端与合约的交互还是合约与合约之间的交互。”上述是以太坊官方文档给出的定义，更通俗的理解，包含两方面内容:

1. ABI是合约接口的说明。
2. ABI定义与合约进行交互数据编码规则。

下面我们将从这两方面对ABI进行说明。

ABI接口说明

ABI是合约接口的说明，内容包括合约的接口列表、接口名称、参数名称、参数类型、返回类型等。这些信息以JSON格式保存，可以在solidity文件编译时由合约编译器生成，详情请参考。

这里以Asset.sol合约为例:

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

    function select(string account) public constant returns(int256, uint256);
    function register(string account, uint256 asset_value) public returns(int256);
    function transfer(string from_account, string to_account, uint256 amount) public returns(int256);
...其他省略...
}
```

Asset Contract ABI：

```
[
  {
    "constant": true,
    "inputs": [
      {
        "name": "account",
        "type": "string"
      }
    ],
    "name": "select",
    "outputs": [
      {
        "name": "",
        "type": "int256"
      },
      {
        "name": "",
        "type": "uint256"
      }
    ],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "from_account",
        "type": "string"
      },
      {
        "name": "to_account",
        "type": "string"
      },
      {
        "name": "amount",
        "type": "uint256"
      }
    ],
    "name": "transfer",
    "outputs": [
      {
        "name": "",
        "type": "int256"
      }
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "constant": false,
    "inputs": [
      {
        "name": "account",
        "type": "string"
      },
      {
        "name": "asset_value",
```

(continues on next page)

(续上页)

```
        "type": "uint256"
      }
    ],
    "name": "register",
    "outputs": [
      {
        "name": "",
        "type": "int256"
      }
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "inputs": [

    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "constructor"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": false,
        "name": "ret",
        "type": "int256"
      },
      {
        "indexed": true,
        "name": "account",
        "type": "string"
      },
      {
        "indexed": true,
        "name": "asset_value",
        "type": "uint256"
      }
    ],
    "name": "RegisterEvent",
    "type": "event"
  },
  {
    "anonymous": false,
    "inputs": [
      {
        "indexed": false,
        "name": "ret",
        "type": "int256"
      },
      {
        "indexed": true,
        "name": "from_account",
        "type": "string"
      },
      {
        "indexed": true,
        "name": "to_account",
        "type": "string"
```

(continues on next page)

(续上页)

```
      },
      {
        "indexed": true,
        "name": "amount",
        "type": "uint256"
      }
    ],
    "name": "TransferEvent",
    "type": "event"
  }
]
```

可以看到ABI是一个JSON的对象数组，包含接口与Event的信息。Asset合约的transfer接口以及其ABI如下：

接口：

```
function transfer(string from_account, string to_account, uint256 amount)
public returns(int256)
```

接口ABI:

ABI编码

假定用户需要调用Asset合约的transfer接口，已知条件如下。

Asset合约地址：

```
0x1386bf8e0138e821994140503ee214a9019eb0ec
```

transfer接口定义：

```
function transfer(string from_account, string to_account, uint256 amount)
public returns(int256);
```

用户参数：

```
String fromAccount = "Alice";
String toAccount = "Bob"
BigInteger amount = 10000;
```

用户如何将这些参数传递给最终执行交易的EVM，使EVM能够知道用户调用的接口为transfer接口，并且EVM能够正确读取用户输入的参数？EVM的返回值用户又该如何使用？

这是ABI的另一个作用，定义了数据的编码格式。

这里以引子中交易回执的input字段为例来分析交易的输入编码：

```
"input":  "0x9b80b050000000000000000000000000000000000000000000000000000000000000600000
```

input数据可以分为函数选择器和参数编码两部分。

1. 函数选择器（Function Selector）

用来指定调用的函数，函数签名Keccak哈希的前四个字节，EVM根据函数选择器来判断用户调用的是合约的哪个接口。

在transfer接口调用中：

```
bytes4(sha3("transfer(string,string,uint256)")) = 0x9b80b050
```

2. 参数编码

参数的编码(解码同样适用)需要结合ABI描述信息的内容，根据ABI描述信息中接口的类型列表对参数进行编码。

transfer类型列表：


```
[
    {
        "name":"from_account",
        "type":"string"
    },
    {
        "name":"to_account",
        "type":"string"
    },
    {
        "name":"amount",
        "type":"uint256"
    }
]
```


参数列表：

```
String fromAccount = "Alice";
String toAccount = "Bob"
BigInteger amount = 10000;
```

编码流程：

```
enc(string("Alice"),string("Bob"),uint256(10000)) = "0000000000000000000000000000000000000000
```

将Function selector与参数编码两部分合并，最终得到input。

合约ABI的优势与局限

为什么需要合约ABI

从ABI的定义就可以看出，ABI是与合约交互的标准形式，相当于定义访问合约接口协议规范，统一了合约与合约、不同平台的客户端与合约之间的交互形式。

合约ABI的局限

下面来谈谈合约ABI编码的一些局限：

- ABI编码本身的规则很复杂，这增加了用户实现的难度，不过除了个别ABI库的作者外，普通用户并不需要自己实现。

- ABI的编码会对所有的数据编码强制32字节对齐，最终这些编码数据都需要随交易进行持久化，浪费了很多的存储空间。
- 升级困难：ABI添加新的类型支持甚至是新的规则时，所有平台的实现都需要升级，这些新的特性在有的平台上不一定容易支持。比如：ABIEncoderV2到目前为止，各个库的支持仍然不是很完善。

总结

本文介绍了合约ABI的概念，ABI的JSON描述信息以及ABI编解码，并且最后分析了ABI编解码的优势与局限，让用户对合约ABI有一个初步的了解认识。大家如果有更深入的需求，可以查看ABI的官方文档。

参考资料

- Asset.sol源码
- 合约ABI JSON格式
- Function Selector
- ABI编码

22.4.7 密码算法和协议

国密算法 支持

国密SSL 支持

哈希算法：Keccak256、SM3

对称加密算法：AES、SM4

非对称加密算法：ECDSA、SM2

非对称加密椭圆曲线：secp256k1、sm2p256v1

FISCO BCOS的国密特性

作者：李昊轩 | FISCO BCOS 核心开发者

> 安全是信任的保障，算法则是安全的基础，节点通信信道的建立、签名的生成、数据的加密等，都需要用到密码学的相应算法。FISCO BCOS采用国密算法，实现安全可控的区块链架构。

本期文章讲解国密算法的相关概念，以及国密算法在区块链中的应用。

区块链中的密码学

在区块链中，一般使用密码学的场景有以下几种：

1. 数据哈希算法

哈希函数是一类单向函数，作用是将任意长度的消息转换为定长的输出值，具有单向性、无碰撞性、确定性、不可逆的性质。 在区块链中，哈希函数用于将消息压缩成定长的输出，以及保证数据真实性，确保数据未被修改。

2. 数据加密解密算法

加解密算法主要分为对称加密和非对称加密两种：

- 对称加密具有速度快、效率高、加密强度高等特点，使用时需要提前协商密钥，主要对大规模数据进行加密，如FISCO BCOS的节点数据落盘时进行的加密。
- 非对称加密具有无需协商密钥的特点，相较于对称加密计算效率较低，存在中间人攻击等缺陷，主要用于密钥协商的过程。

针对不同的需求，两者可以互相配合，组合使用。

3. 消息签名的生成和验证

在区块链中，需要对消息进行签名，用于消息防篡改和身份验证。如节点共识过程中，需要对其他节点的身份进行验证，节点需要对链上交易数据进行验证等。

4. 握手建立流程

前面我们讲过**节点TLS的握手过程**（点击直达），其中需要使用到密码组件和数字证书，都需要使用到相应的密码学算法。

FISCO BCOS的国密算法

国密算法由国家密码局发布，包含SM1\ SM2\ SM3\ SM4\等，为我国自主研发的密码算法标准。 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。

1. 节点TLS握手中采用国密SSL算法；
2. 交易签名生成，验证过程采用国密SM2算法；
3. 数据加密过程中采用国密SM4算法；
4. 数据摘要算法采用国密SM3算法。

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：

国密版与标准版FISCO BCOS在数据结构上的差异如下：

开启国密特性

需要使用国密版FISCO BCOS特性时，有以下几点需要注意：

1. 搭建国密版FISCO BCOS区块链

搭建国密版FISCO BCOS的区块链方式主要有2种：

(1). 使用buildchain.sh脚本搭建

buildchain.sh脚本的-g为国密编译选项，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，操作方式：

```
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
./build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545 -g
```

执行成功后，会搭建本地四节点的国密版FISCO BCOS联盟链。

(2). 使用企业级部署工具搭建

企业级部署工具generator的-g命令为国密相关操作，用户需要在生成相关证书，下载二进制等操作的过程中附带-g选项，操作方式：

参考教程：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/tutorial/enterprise_quick_start.html

```
./generator --download_fisco ./meta -g
./generator --generate_chain_certificate ./dir_chain_ca -g
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyB -g
...
```

以下操作与教程中类似

```
./generator --build_install_package ./meta/peersB.txt ./nodeA -g
```

2. 使用国密版sdk/控制台

国密版sdk和控制台的操作命令与普通版相同，Web3SDK和1.x版本控制台在启用国密特性时需要做如下操作：

- (1). 将applicationContext.xml中的encryptType设置为1；
- (2). 加载私钥时需要加载国密版的私钥；
- (3). 控制台还需要获取国密版jar包。

注解： java sdk和2.6.0+版本的控制台启用国密算法时不需要额外配置

---

3. 开启国密版落盘加密功能

国密版落盘加密需要同时对节点conf/gmnode.key和conf/origin_cert/node.key进行加密，其余操作与普通版相同。

总结

FISCO BCOS中的主要算法特性对比如下：

「用户提问」

Q：**王刚+云飞微联网+珠海**：请问solidity编译器也需要用到密码学算法吗？

A：**李昊轩**：solidity中的abi会用到哈希，需要底层和合约编译器都使用相同的SM3算法。

Q：**陈小军-江南科友-广州**：请问一下国密TLS协议套件是自己实现的吗，还是用了开源的？

A：**李昊轩**：国密TLS我们采用的是TASSL开源库。

Q：**腾龙(何直群)**：如果使用国密的节点，因为签名算法都换了，是不是 RPC SDK 都会不一样了?

A：**李昊轩**：是的， sdk也需要开启国密特性。

Q：**KHJ**：目前落盘加密的私钥是如何处理的?

A：**meng**：要自己保存，可以参考这里的说明

FISCO BCOS国密版部署示例

作者：刘海锋 | FISCO BCOS开源社区贡献者

随着区块链的底层技术被不断关注，越来越多的项目开始使用区块链来固化数据。近期因为工作需要，对比了多家国内的区块链产品，实际部署了几个区块链系统后，最终选择了FISCO BCOS。在实际部署的过程中，我总结出了这一套可供实际项目使用的部署流程，供区块链爱好者参考。

设计覆盖实际项目使用情况的一套场景

在决定使用FISCO BCOS时，为了尽可能覆盖到实际项目使用的情况，我设计了一套场景，里面包含了大多数实际情况，并且遇坑填坑，最终按照设计的场景部署成功了。以下是我设计的场景：

1、假设多家机构组成联盟，每家机构都派出代表进驻联盟委员会(在本次测试组网部署中，假设有ABCD四家机构)。

2、联盟决定使用国密版FISCO BCOS组建联盟链。

3、联盟委员会确定联盟链组网模式(主要确认节点数量、节点分布、群组构成情况和信息交换方式等)。

本次测试，使用同一服务器的不同端口模拟不同节点，使用拷贝指令模拟信息交换方式(实际部署可通过U盘、邮件、网络消息传输等联盟成员信任的方式传递)，组网模式假设如下：

> **机构A**
>
> > **节点0** 127.0.0.1:30300 **节点1** 127.0.0.1:30301 **节点2** 127.0.0.1:30302
>
> **机构B**
>
> > **节点3** 127.0.0.1:30303
>
> **机构C**
>
> > **节点4** 127.0.0.1:30304
>
> **机构D**
>
> > 暂时不参与联盟链的组建，没有节点。
>
> 机构A所有节点与机构B所有节点组成群组1，使用pbft共识机制。
>
> 机构B所有节点与机构C所有节点组成群组2，使用raft共识机制。
>
> 联盟链所有节点都参与共识 。

4、联盟委员会提供联盟链的链证书(自建或向CA公司申请，本次测试使用FISCO BCOS提供的企业级部署工具自建)。

5、联盟的每家机构向联盟委员会申请对应机构的机构证书(本次测试使用FISCO BCOS提供的企业级部署工具自建)。

6、联盟每个机构，根据联盟委员会确定的联盟链组网模式，使用FISCO BCOS提供的企业级部署工具，生成机构下属的节点部署程序，并按组网模式修改对应节点部署程序的配置文件。

7、联盟所有机构部署并启动下属节点程序，组成联盟链。

8、组网变动：

- 机构A认为自己下属节点太多，消耗资源，准备从群组1移除节点2变为游离节点，不参与共识。同时为了提高业务处理效率，将节点1设置为观察节点，不参与共识。
- 机构D想加入群组2，并添加机构D下属节点：

> **节点5** 127.0.0.1:30305

- 机构D和机构B想用节点3和节点5组成新群组：群组3。

9、机构A和机构D向联盟委员会提出组网变动申请。

10、联盟委员会同意机构A和机构D的组网变动申请。

11、机构A开始组网变动操作：

- 群组1的机构A或B根据联盟委员会同意的机构A组网变动申请，使用控制台或SDK发送节点2退出和节点1转换为观察节点的指令，完成机构A的组网变动请求。

12、机构D开始组网变动操作：

- 第5步中，机构D已向联盟委员会申请机构D的机构证书，现在，机构D使用FISCO BCOS提供的企业级部署工具，生成机构D下属的节点部署程序，根据新的组网模式，修改节点部署程序的配置文件，启动节点，准备加入联盟链。
- 机构D向群组2的机构B或C发送节点5入网请求，等待机构B或C使用控制台或SDK发送节点5加入群组2共识的指令。
- 机构B或C根据联盟委员会同意的机构D组网变动申请，使用控制台或sdk发送节点5加入群组2共识的指令。
- 机构D和机构B根据联盟委员会同意的群组3组网变动申请，组建群组3。

13、结束。

部署联盟链准备工作

FISCO BCOS为实际项目中部署和使用联盟链提供了企业级部署工具，里面包含了部署联盟链所需的各种操作。联盟委员会可以使用企业级部署工具自建生成链证书并为联盟成员机构颁发自建的机构证书，联盟成员机构可以使用企业级部署工具生成群组的创世区块和机构下属的节点部署程序。

本次测试模拟联盟链的实际搭建过程，测试部署目录(Linux)为：/usr/local/rc3-test-BCOS/，以下简称为目录rc3-test-BCOS。

- 环境要求：python 2.7+/3.6+，openssl 1.0.2k+。
- 获取企业级部署工具：git clone https://github.com/FISCO-BCOS/generator.git，得到企业级部署工具根目录generator。
- 将generator上传(或直接clone)为/usr/local/rc3-test-BCOS/generator/，以下简称为目录generator。
- 保证generator的操作权限：chmod -R 777 /usr/local/rc3-test-BCOS/generator/
- 安装企业级部署工具(一定要保证安装正确)：

```
      cd generator
./scripts/install.sh

#查看版本:
./generator -v
> INFO | v1.0.0-rc3
```

- 下载国密版FISCO BCOS：

```
      cd generator
./generator --download_fisco ./meta -g
```

-g参数表示国密，去掉-g则下载的是普通版FISCO BCOS，执行完该命令，国密版FISCO BCOS会下载到generator/meta/下。

- 查看国密FISCO BCOS版本：

```
cd generator
./meta/fisco-bcos -v
```

```
> FISCO-BCOS Version : 2.0.0-rc3 gm
> Build Time              : 20190613 15:24:38
> Build Type              : Linux/clang/RelWithDebInfo
> Git Branch              : master
> Git Commit Hash      : a43952c544aa8252f7ac965e310148c099510410
```

可以看到，本次测试部署使用的是v1.0.0-rc3版本的企业级部署工具，2.0.0-rc3 gm版本的FISCO BCOS(似乎与rc2版本的部署流程相同)。

证书说明

FISCO BCOS国密版的链证书和机构证书需要两套，分别为国密证书和普通版证书。

- 证书链：

```
> 链证书----机构证书----节点证书
>
> 链证书----机构证书----sdk证书
```

- 证书支持自建、CA公司申请，企业级部署工具底层使用openssl工具自建生成，如果要用CA公司申请的证书，可直接把证书文件的名称改为规定的名称即可。
- 链证书普通版ca.crt，对应的私钥ca.key。链证书国密版gmca.crt，对应的私钥gmca.key。
- 机构证书普通版agency.crt，对应的私钥agency.key，机构证书国密版gmagency.crt，对应的私钥gmagency.key。
- 说明：链证书私钥需要联盟委员会保管，机构证书的私钥由各机构自行保管，证书及私钥的意义这里不做延伸。
- 使用FISCO BCOS企业级部署工具时，所有需要证书的操作都需要将链证书和机构证书(.crt文件)放置在generator/meta/下。

部署联盟链详细步骤（附每一步代码）

经过尝试，我按以下操作顺序执行，成功部署了一个符合上面场景描述的联盟链。由于步骤十分详细，而且含有大量代码，所以单独放在此处链接里：《手把手教你实操部署FISCO BCOS联盟链》（温馨提示：此文附每一步代码演示，复制链接到PC端打开体验为佳。）

关键步骤总结

1. 联盟委员会及相关成员确定组网模式；
2. 相关的各种证书的颁发；
3. 准备FISCO BCOS企业级部署工具；
4. 根据组网模式，各机构收集所要组建群组的下属节点的p2p连接信息；
5. 各群组生成并分发创世区块(文件)；
6. 各机构生成机构下属加点的节点部署程序；
7. 各机构根据组网模式，修改下属节点部署程序中的配置文件，启动联盟链。

写在最后

至此，这套可供实际项目使用的部署流程就搭建完成了。我在刚开始部署时，会觉得整个系统的部署很复杂，多次部署之后，觉得流程挺合理挺清楚的，因为在实际场景中不是一个人在操作，是整个联盟各机构一起在协同，按联盟链该有的组织形式去部署其实很快，多次部署中我也总结了一些经验：

- 首先是多看官方技术文档，梳理出了大致流程。
- 其次就是解决证书的问题，这个问题对我来说如鲠在喉，在实际业务中，证书必须是具有法律效力，由CA公司颁发的。我认为证书的生成，可以考虑从企业级部署工具中分离出来，作为搭建测试链的一个工具。当然，在实际业务中，如果不重视数据的法律效应，也可以只把证书当做一个加密的公私钥对看待。
- FISCO BCOS目前处于快速迭代开发的阶段，推出的一些功能都很好，比如区块链浏览器。

- 适配支持FISCO BCOS底层平台的区块链中间件平台WeBASE，功能全面，上手快速，节约大量时间都很好。

整体而言FISCO BCOS体验很不错，期待看到FISCO BCOS更多的突破。

一个数字引发的探索——ECDSA解析

作者：李辉忠 | FISCO BCOS 高级架构师

FISCO BCOS交易签名算法基于ECDSA原理进行设计，ECDSA也是比特币和以太坊采用的交易签名算法。本文介绍ECDSA及椭圆曲线加密（ECC）相关知识、ECDSA的Recover机制和实现方式、FISCO BCOS交易签名和验签的底层原理。内容偏硬（shu）核（xue），欢迎对密码学原理、区块链底层原理感兴趣的开发者一起交流。

故事开始

故事要从以太坊中一个神奇的魔数开始说起。

APPENDIX F. SIGNING TRANSACTIONS

Transactions are signed using recoverable ECDSA signatures. This method utilises the SECP-256k1 curve as described by Courtois et al. [2014], and is implemented similarly to as described by Gura et al. [2004] on p. 9 of 15, para. 3.

It is assumed that the sender has a valid private key p_r, which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range $[1, \mathbf{secp256k1n} - 1]$.

We assume the existence of functions ECDSAPUBKEY, ECDSASIGN and ECDSARECOVER. These are formally defined in the literature, e.g. by Johnson et al. [2001].

$$\mathbf{ECDSAPUBKEY}(p_r \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \quad (277)$$

$$\mathbf{ECDSASIGN}(e \in \mathbb{B}_{32}, p_r \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \quad (278)$$

$$\mathbf{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \quad (279)$$

Where p_u is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each $< 2^{256}$), p_r is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range) and e is the hash of the transaction, $h(T)$. It is assumed that v is the 'recovery identifier'. The recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value; this value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid. The value 27 represents an even y value and 28 represents an odd y value.

We declare that an ECDSA signature is invalid unless all the following conditions are true[5]:

$$0 < r < \mathbf{secp256k1n} \quad (280)$$

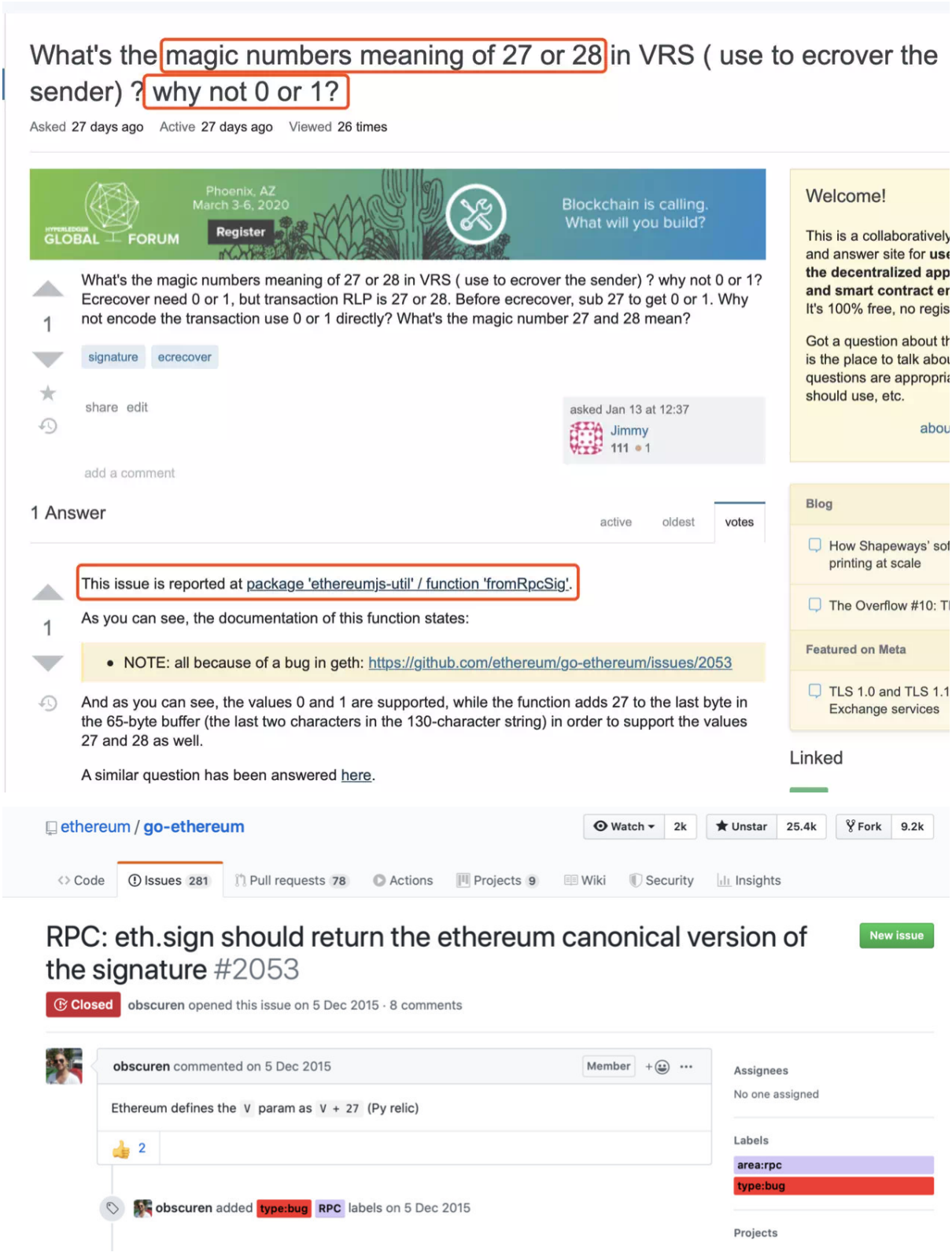
$$0 < s < \mathbf{secp256k1n} \div 2 + 1 \quad (281)$$

$$v \in \{27, 28\} \quad (282)$$

以太坊黄皮书中，关于交易签名的阐述讲到两个特殊的数「27，28」，实际上是从「0，1」通过加了一个27演变得到「27，28」，所以本质上是一个特殊的数27。这个特殊的数字27代表了什么含义呢？一次侦探之旅开始了...

这像是一个bug

搜索发现此前已有许多关于该问题的讨论，其中，Stack Exchange的一篇帖子指出这是一个设计bug。以太坊源码github上，也有一个相关issue，该issue被打上了「type:bug」的标签。



Stack Exchange帖子中有一个链接给出了修复该Bug的代码，请看下面截图（红框）。在注释说明和代码可见，fromRpcSig函数对27这个魔数进行了特殊处理。从RPC过来的签名中，v值如果小于27（可能是0-3），则直接加上27作为新v值，fromRpcSig函数通过这个方式兼容ECDSA原始v值（也就是recoveryID）和以太坊v值。

```
66  /**
67   * Convert signature format of the `eth_sign` RPC method to signature parameters
68   * NOTE: all because of a bug in geth: https://github.com/ethereum/go-ethereum/issues/2053
69   */
70  export const fromRpcSig = function(sig: string): ECDSASignature {
71    const buf: Buffer = toBuffer(sig)
72
73    // NOTE: with potential introduction of chainId this might need to be updated
74    if (buf.length !== 65) {
75      throw new Error('Invalid signature length')
76    }
77
78    let v = buf[64]
79    // support both versions of `eth_sign` responses
80    if (v < 27) {
81      v += 27
82    }
83
84    return {
85      v: v,
86      r: buf.slice(0, 32),
87      s: buf.slice(32, 64),
88    }
89  }
```

这真是以太坊设计的一个bug吗？回到刚才那个fromRpcSig的源代码文件，详细看其各接口实现，我们发现有这样一行代码「v: chainId ? recovery + (chainId * 2 + 35) : recovery + 27」，这行为v赋值的代码透露了三个信息，分别是魔数27、魔数35和ChainID。

```
12  /**
13   * Returns the ECDSA signature of a message hash.
14   */
15  export const ecsign = function(
16    msgHash: Buffer,
17    privateKey: Buffer,
18    chainId?: number,
19  ): ECDSASignature {
20    const sig = secp256k1.sign(msgHash, privateKey)
21    const recovery: number = sig.recovery
22
23    const ret = {
24      r: sig.signature.slice(0, 32),
25      s: sig.signature.slice(32, 64),
26      v: chainId ? recovery + (chainId * 2 + 35) : recovery + 27,
27    }
28
29    return ret
30  }
```

于是，疑问更多了，魔数35是什么？ChainID又是什么？

这不像是一个Bug

带着这些疑问，再一次查阅相关设计材料，我们看到，以太坊EIP155中描述了有关ChainID的设计。基于以太坊源码构建的网络，实际运行的链有很多，为了防止一条链的交易被提交上链到另一条链，造成重放攻击，引入了ChainID的设计，在块高2,675,000的位置进行分叉实现。

eip	title	author	type	category	status	created
155	Simple replay attack protection	Vitalik Buterin (@vbuterin)	Standards Track	Core	Final	2016-10-14

Hard fork

Spurious Dragon

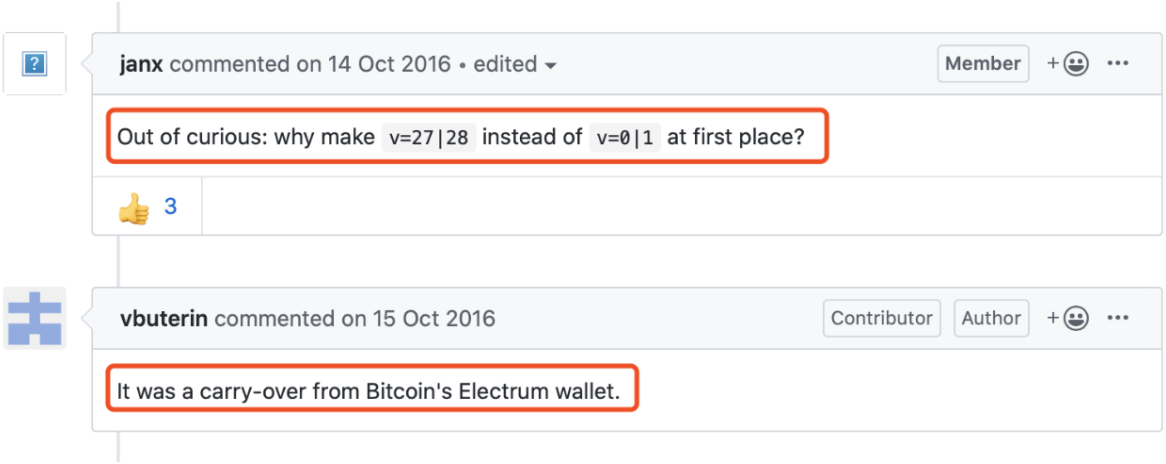
Parameters

- `FORK_BLKNUM` : 2,675,000
- `CHAIN_ID` : 1 (main net)

Specification

If `block.number >= FORK_BLKNUM` and `v = CHAIN_ID * 2 + 35` or `v = CHAIN_ID * 2 + 36`, then when computing the hash of a transaction for purposes of signing or recovering, instead of hashing only the first six elements (i.e. nonce, gasprice, startgas, to, value, data), hash nine elements, with `v` replaced by `CHAIN_ID`, `r = 0` and `s = 0`. The currently existing signature scheme using `v = 27` and `v = 28` remains valid and continues to operate under the same rules as it does now.

明白了ChainID的作用，另一个疑问又产生了——以太坊中，有NetworkID来区分不同网络，为什么还需要ChainID？这要从NetworkID和ChainID的作用范围来解释。NetworkID主要在网络层面进行链的隔离，节点在建立相互连接的时候需要交换NetworkID，拥有一致的NetworkID才能完成握手连接。ChainID是交易层面，防止不同网络的交易被交叉重复攻击。以太坊（ETH）和经典以太坊（ETC）的主网NetworkID都是1，需要通过 ChainID机制才能防止交易在ETH和ETC网络之间交叉重放，ETH主网的ChainID是1，ETC主网的ChainID是61。说到这里其实还是没有搞清楚为什么是27，为什么是35？我们在EIP github的Issue#155中看到Jan和Buterin的交流记录，看来27是来自比特币的产物。



顺藤摸瓜，打开electrum的github，我们在electrum/electrum/ecc.py中找到如下代码

```
451     def sign_message(self, message: bytes, is_compressed: bool, algo=lambda x: sha256d(msg_magic(x))) -> bytes:
452         def bruteforce_recid(sig_string):
453             for recid in range(4):
454                 sig65 = construct_sig65(sig_string, recid, is_compressed)
455                 try:
456                     self.verify_message_for_address(sig65, message, algo)
457                     return sig65, recid
458                 except Exception as e:
459                     continue
460             else:
461                 raise Exception("error: cannot sign message. no recid fits..")
462
463         message = to_bytes(message, 'utf8')
464         msg_hash = algo(message)
465         sig_string = self.sign(msg_hash,
466                                sigencode=sig_string_from_r_and_s,
467                                sigdecode=get_r_and_s_from_sig_string)
468         sig65, recid = bruteforce_recid(sig_string)
469         return sig65
```

```
496 def construct_sig65(sig_string: bytes, recid: int, is_compressed: bool) -> bytes:
497     comp = 4 if is_compressed else 0
498     return bytes([27 + recid + comp]) + sig_string
```

```
286     def verify_message_for_address(self, sig65: bytes, message: bytes, algo=lambda x: sha256d(msg_magic(x))) -> None:
287         assert_bytes(message)
288         h = algo(message)
289         public_key, compressed = self.from_signature65(sig65, h)
290         # check public key
291         if public_key != self:
292             raise Exception("Bad signature")
293         # check message
294         self.verify_message_hash(sig65[1:], h)
```

```
209     def from_signature65(cls, sig: bytes, msg_hash: bytes):
210         if len(sig) != 65:
211             raise Exception("Wrong encoding")
212         nV = sig[0]
213         if nV < 27 or nV >= 35:
214             raise Exception("Bad encoding")
215         if nV >= 31:
216             compressed = True
217             nV -= 4
218         else:
219             compressed = False
220         recid = nV - 27
221         return cls.from_sig_string(sig[1:], recid, msg_hash), compressed
```

从代码中可见，electrum在签名时，为原本只有0-3之间的recid（recoveryID），加上了27，还有一个压缩标记，如果有压缩则再加上4，recid的值范围在27-34。至此可知，27和35大概来源于此，以太坊继承比特币的设计，在比特币源码bitcoin/src/key.cpp的CKey::SignCompact函数中也确定了该实现方式，但是比特币为什么如此设计，仍未可知。

ECDSA才是“bug”

故事到这里，我们对以太坊代码中那个魔数27的前世今生有大概了解，但这仅仅是故事的开端，由此引发我们进一步思考一个问题：recoveryID是什么？为了解释清楚这个问题，我们需要从ECDSA算法着手，从数学角度理解其背后的原理。ECDSA是FISCO BCOS采用的交易签名算法，由此我们会发现，ECDSA算法有一种Recover机制，它才是真正“bug”级别的功能。

ECDSA（Elliptic Curve Digital Signature Algorithm）是基于椭圆曲线的数字签名算法。数字签名算法是采用公私钥体系实现类似写在纸上的普通签名，用于鉴别数字信息的方法，常见的数字签名算法包括DSA、RSA和ECDSA等。椭圆曲线密码（ECC）是基于椭圆曲线数学的公钥加密算法，建立在椭圆曲线离散对数困难问题之上，常用的协议有ECDH、ECDSA和ECIES等。椭圆曲线的参数可以有多种配置

方式，也就存在多种不同的曲线，例如secp256k1、secp256r1、Curve25519等，不同曲线的安全性存在一些区别，在SafeCurves中有相关对比描述。

ECDSA算法主要包括以下四个关键功能:

产生密钥GenKey

- 选择一条椭圆曲线E_P(a,b)，选择基点G，G的阶数为n
- 选择随机数d ∈ n为私钥，计算公钥Q = dG

签名算法Sign

- 对消息m使用消息摘要算法，得到z=hash(m)
- 生成随机数k ∈ n，计算点(x, y)=kG
- 取r=x mod n，若r=0则重新选择随机数k
- 计算s = k^1(z+rd) mod n，若s=0则重新选择随机数k
- 上述(r,s)即为ECDSA签名

验证算法Verify

使用公钥Q和消息m，对签名(r,s)进行验证。

- 验证r,s ∈ n
- 计算z = hash(m)
- 计算u_1 =zs^1 mod n和u_2 = rs^1 mod n
- 计算(x, y) = u1G+u2Q mod n
- 判断r == x，若相等则签名验证成功

恢复算法Recover

已知消息m和签名(r,s)，恢复计算出公钥Q。

- 验证r, s ∈ n
- 计算R=(x, y)，其中x=r,r+n,r+2n...，代入椭圆曲线方程计算获得R
- 计算z = hash(m)
- 计算u_1 = zr^1 mod n和u_2 = sr^1 mod n
- 计算公钥Q= (x', y')=u_1G+u_2R

为了回答recoveryID的问题，我们重点关注「恢复算法Recover」。在计算R的步骤可以看到，存在多个x的取值可能性，导致存在多个R的可能性，因此计算得到的Q也存在多个可能的结果，需要通过和已知的公钥对比，确定哪一个Q是正确的。如果遍历x的所有可能都未找到正确的Q，说明该消息和签名是不对应的，或者是一个未知的公钥。

为了确定正确的Q，需要遍历x的所有可能取值，跑多轮Recover算法，这个时间开销是比较大的。**为了提高Recover的时间效率，采用空间换时间的思路，在签名中增加一个v值，用于快速确定x，避免遍历查找试探，这个v值就是recoveryID。**

在区块链系统中，客户端对每笔交易进行签名，节点对交易签名进行验证。如果采用「验证算法Verify」，那节点必须首先知道签发该交易所对应的公钥，因此需要在每笔交易中携带公钥，这

需要消耗很大带宽和存储。如果采用「恢复算法Recover」，并且在生成的签名中携带recoveryID，就可以快速恢复出签发该交易对应的公钥，根据公钥计算出用户地址，然后在用户地址空间执行相应操作。

这里潜藏了一个区块链设计哲学，区块链上的资源（资产、合约）都是归属某个用户的，如果能够构造出符合该用户地址的签名，等同于掌握了该用户的私钥，因此节点无需事先确定用户公钥，仅从签名恢复出公钥，进而计算出用户地址，就可以执行这个用户地址空间的相应操作。**FISCO BCOS基于这个原理设计实现了交易签名和验签**。

recoveryID的计算

关于JavaSDK性能优化的文章（记一次JavaSDK性能从8000提升至30000的过程）中提到一个关键优化点——recoveryID的计算，这里仔细展开讨论。

ECDSA签名（r,s），其中r是椭圆曲线上一个点kG (x, y)对应的x mod n，相当于签名信息中只留下了X轴坐标相关的值，丢弃了Y轴相关的值。在「恢复算法Recover」中尝试找回Y轴对应的值构造R，进而恢复出公钥。

由于r = x mod n，因此r,r+n,r+2n...都可能是合法的原始x值，不同的椭圆曲线存在不同数量这样合法的x值，FISCO BCOS采用的secp256k1曲线存在两个可能r, r+n。

每一个X轴坐标对应两个可能的Y坐标，因此FISCO BCOS中具备四种可能的R，(r, y) (r, -y) (r+n, y') (r+n, -y')。但是，对于一个r值存在两个X轴坐标的概率极低，低到几乎可以忽略，以太坊中就忽略了这两种小概率事件。

那这个小概率事件的概率具体有多小呢？这要从secp256k1曲线的参数说起，通常描述一个椭圆曲线的点（x,y）的时候，x和y的值是 mod p 的结果，p是曲线的参数，它是一个大素数，之前提到的n也是曲线的参数，等于这条曲线上点的数量（曲线上点的数量为n*h，h也是曲线参数，该曲线h=1），在secp256k1中，n和p的值非常接近，具体可见下图。

2.4.1 Recommended Parameters secp256k1

The elliptic curve domain parameters over $\mathbb{F}_p$ associated with a Koblitz curve secp256k1 are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field $\mathbb{F}_p$ is defined by:

$$p = \texttt{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F}$$
$$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

The curve E: $y^2 = x^3 + ax + b$ over $\mathbb{F}_p$ is defined by:

$$a = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000}$$
$$b = \texttt{00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007}$$

The base point G in compressed form is:

$$G = \texttt{02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798}$$

and in uncompressed form is:

$$G = \texttt{04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8}$$

Finally the order n of G and the cofactor are:

$$n = \texttt{FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$
$$h = \texttt{01}$$

由于r = x mod n，x是mod p的结果，r是mod n的结果，x值的范围是[0, p-1]，r值的范围是[0, n-1]。如果r+n也是曲线上的点，则r的值必须小于p-n，概率为 (p-n) / p，大约为3.73*10^-39，这个概率是非常小的。

基于签名结果（r, s）和签名过程中生成的随机点（x, y）的y值，recoveryID的计算方式如下：

1. id = y & 1; //「签名算法Sign」中kG点的y坐标，根据奇偶性设置id值，因为y是mod p的结果，其奇偶性与坐标轴的正负性是完全对应的
2. id |= (x != r ? 2 : 0); // 小概率事件，如前文解释
3. if (s > n / 2) id = id ^ 1; // 签名计算得出的s如果大于n/2就会取n-s作为s值，因此这里做相应转换，这两个转换是同时发生的

JavaSDK性能优化的文章就是基于这个计算公式，将遍历探寻recoveryID改为计算获得，大幅提升了性能。

后话

从一个神奇的数字开始，查阅相关资料，了解设计原理，进而闯入ECDSA的世界，在一堆数学公式中迷茫、游荡，问题一个接着一个。一开始雾里看花，似懂非懂，靠着处女座的洁癖精神，总算把心中疑问一一化解。精妙绝伦的密码协议，高深莫测的数学理论，做一个区块链码农，要学习的东西还很多。唯有苦其心志，劳其筋骨，善待每一个疑点，不放过每一处细节。总会有一天，那时拨开云雾见天日，守得云开见月明。

参考资料

- Ethereum yellow paper
- Stack Exchange
- ISSUE
- EIP155
- EIP155 Issue
- ChainID
- ECC wiki
- ECDSA wiki
- Curve25519 wiki
- SafeCurves
- ECC SEC1
- ECC SEC2

一场椭圆曲线的寻根问祖之旅

作者：李辉忠 | FISCO BCOS 高级架构师

本文介绍密码学中常见的椭圆曲线以及他们之间的关系，介绍不同标准体系的命名规则，尝试描述椭圆曲线之间的家族演义关系。文章试图讲清椭圆曲线相关概念和功能，不涉及复杂的数学证明和推理，欢迎感兴趣的同学阅读。笔者主要参考Wikipedia和相关组织网站的信息进行整理，不排除出现纰漏的可能，欢迎专家批评指正。

一个可能你没关心过的问题

在《一个数字引发的探索——ECDSA解析》中提到的椭圆曲线secp256k1，它有一些特性，可以快速计算出recoveryID。这个secp256k1为什么如此命名？不怕各位笑话，我在弄清楚它之前，经常拼写错，写成sec256pk1，seck256p1等。

咬文嚼字secp256k1

搞清楚secp256k1的命名含义其实很简单，搜索引擎可以快速为你定位到答案，它出自一个密码协议标准，每一个字母和数字都代表着特定含义，我们来逐一解析。

1、密码协议标准

第一部分是「sec」，sec是Standards for Efficient Cryptography 的简称，是SECG发布的一种密码学协议标准。SECG发布的「SEC 1」和「SEC 2」两个关于椭圆曲线的协议标准，在「SEC 2」中有详细说明secp256k1以及其他曲线的参数定义。除了「sec」，还有众多其他关于椭圆曲线的协议标准，从「SafeCurve」中可以看到有下列不同类型的标准。

Introduction

There are several different standards covering selection of curves for use in elliptic-curve cryptography (ECC):

- ANSI X9.62 (1999).
- IEEE P1363 (2000).
- SEC 2 (2000).
- NIST FIPS 186-2 (2000).
- ANSI X9.63 (2001).
- Brainpool (2005).
- NSA Suite B (2005).
- ANSSI FRP256V1 (2011).

「SafeCurve」此处较久没有更新，有些标准已经更新了多次，例如NIST关于数字签名的标准 FIPS 186目前在用的是第四版，第五版也在起草中了，从「NIST」官网中可见。NIST是美国的国家标准技术研究所（National Institute of Standards and Technology），因此，NIST的标准也是美国标准。

FIPS	186-5	**Digital Signature Standard (DSS)** Download: FIPS 186-5 (Draft) (DOI); Local Download	Draft	10/31/2019
FIPS	186-4	**Digital Signature Standard (DSS)** Download: FIPS 186-4 (DOI); Local Download; Comments received on FIPS 186-4 (Dec. 2015); Request for Comments on FIPS 186-4 (Oct. 2015); Press Release (07-23-2013); Proposed Change Notice for FIPS 186-3 (Apr. 2012); Request for Comments on Proposed Change Notice (Apr. 2012)	Final	7/19/2013

「NIST FIPS 186-4」标准中定义了若干椭圆曲线标准，例如NIST P-256、NIST P-384等，其中开头NIST也代表密码协议标准的名字。后续描述都是围绕这两个标准来解析。

2、有限域

第二部分是「p」，p表示该椭圆曲线是基于素数有限域Fp。有限域是离散数学中的概念，此处不做展开，简单来说，它是一个由有限数量元素组成的集合，元素之间可以进行加法和乘法计算，具备一些独特的属性。密码学中使用椭圆曲线都是基于有限域的，除了素数有限域Fp之外，还有另一种特征为2的有限域F2m（因格式问题，2m应为2的m次方，下同），Fp的大小（元素个数）为p，F2m的大小为2m。基于Fp的椭圆曲线为：

基于F2m的椭圆曲线为：

$$y^2 + xy = x^3 + ax^2 + b \text{ in } \mathbb{F}_{2^m}$$

在「SEC 2」中还定义了sect163k1、sect163r1等曲线，其中，t表示的是该曲线基于F2m。在「NIST FIPS 186-4」中定了P-256、B-163等曲线，P-表示基于Fp，B-表示基于F2m。

3、有限域大小

每个椭圆曲线E都有若干关键参数，包括阶为n的基点G和系数h等，其中，n为一个大素数，n*h为椭圆曲线上点的数量。为了计算效率考虑，h通常设置为1、2或4。通俗地讲，如果椭圆曲线上的点数量越多，那么这条椭圆曲线的安全度就越高，因此n的取值是影响曲线安全的关键。椭圆曲线又都是基于有限域的，曲线上的点都是有限域中的元素，因此，有限域大小决定了曲线安全度。第三部分「256」就是有限域大小的表现形式，还有更多其他如192、224、384等，在「NIST FIPS 186-4」中有个表格展现了Fp 和F2m 两个域的各种不同大小配置。

Table D-1: Bit Lengths of the Underlying Fields of the Recommended Curves

Bit Length of n	Prime Field	Binary Field
161 – 223	$\mathbf{len}(p) = 192$	$m = 163$
224 – 255	$\mathbf{len}(p) = 224$	$m = 233$
256 – 383	$\mathbf{len}(p) = 256$	$m = 283$
384 – 511	$\mathbf{len}(p) = 384$	$m = 409$
≥ 512	$\mathbf{len}(p) = 521$	$m = 571$

SEC标准在这块的设置和NIST标准类似，我们会看到p系列的曲线有p192、p224、p256（secp256k1就是其中一种）、p384和p521，t/B系列有t163/B-163、t233/B-233等。

4、Koblitz Curve

第四部分「k」表示该曲线是Koblitz Curve，从「SEC 2」中可以看到还有此处标记为r的曲线（如secp256r1），r表示该曲线是伪随机曲线Pesudo-Random Curve。Koblitz Curve命名源自数学家「Neal Koblitz」，它是一种特殊的曲线，它的一些参数是精心挑选设置的。Koblitz Curve具有自同态的性质，可以通过优化大幅提升计算效率。相比之下，Pesudo-Random Curve的对应参数是通过随机种子计算出来的，有标准的检验算法可以检测所有参数是随机种子产生而来。对应「**2、有限域**」中的两个椭圆曲线，Koblitz Curve分别简化为

$$y^2 = x^3 + b$$

$$y^2 + xy = x^3 + ax^2 + 1,\ a = \{0, 1\}$$

例如，secp256k1对应的曲线b=7，即曲线表示为

$$y^2 = x^3 + 7$$

在「NIST FIPS 186-4」中Koblitz Curve曲线以「K-」标记开头，分别有K-163、K-233等。

5、末位标记

到了第五部分「1」，这是代表在前4个条件下提供了多种推荐参数设置，在SEC标准中大部分该位都是1，即只提供一种推荐参数，sect163r2是一个例外。下面把SEC和NIST两个标准推荐的曲线分别列一下，二者有较大部分是相同的参数设置。

SEC	NIST
secp192r1	P−192
secp224r1	P−224
secp256r1	P−256
secp384r1	P−384
secp521r1	P−521
secp192k1	
secp224k1	
secp256k1	
sect163k1	K−163
sect163r1	
sect163r2	B−163
sect233k1	K−233
sect233r1	B−233
sect239k1	
sect283k1	K−283
sect283r1	B−283
sect409k1	K−409
sect409r1	B−409
sect571k1	K−571
sect571r1	B−571

上述表格中，同一行中SEC和NIST都出现的，两个曲线虽然名字不同，但参数完全相同，也就是说其实一样的。橙色底纹的几个SEC曲线没有对应的NIST曲线，因此SEC标准包含的曲线比NIST多一些，本文开头提到的secp256k1就是SEC单独存在的。说到这里，不得不提一个正经八卦。据说，NIST推荐的Pesudo-Random Curve，也就是P和B系列，并没有公布随机数挑选规则，外界存在一种疑虑，可能NSA（美国国家安全局）掌握了后门，能够轻易破解这些密码协议。有兴趣的同学可以搜索「Dual_EC_DRBG后门」，更大的八卦是据说中本聪选择secp256k1作为比特币签名算法的曲线，而没有选择更常用的secp256r1，也是因为这个潜藏的风险。

椭圆曲线族谱

调研发现，「STD」记录了比「SafeCurve」更为详细的标准和曲线，感觉这可以算是椭圆曲线族谱了。翻阅该网站记录的所有曲线，发现绝大部分还是基于「（2）有限域」中的曲线，推荐的参数不同而已。但是，在「other」中存在几种例外，E-222采用Edward Curve，Curve25519采用Montgomery Curve，Ed448采用Twisted Edward Curve。

Edward Curve是什么？Montgomery Curve又是怎样的？Edward与Twisted Edward Curve又有什么关系？上述问题再一次触碰到我的知识盲区，所以接下来只好以截图为主，内容源自Wikipedia，如果觉得看着有点晕，可直接跳过看结论。「Edward Curve」定义如下：

Definition [edit]

The equation of an Edwards curve over a field *K* which does not have characteristic 2 is:

$$x^2 + y^2 = 1 + dx^2y^2$$

for some scalar $d \in K \setminus \{0, 1\}$. Also the following form with parameters *c* and *d* is called an Edwards curve:

$$x^2 + y^2 = c^2(1 + dx^2y^2)$$

where $c, d \in K$ with $cd(1 - c^4 \cdot d) \neq 0$.

「Montgomery Curve」定义如下：

Definition [edit]

A Montgomery curve over a field K is defined by the equation

$$M_{A,B} : By^2 = x^3 + Ax^2 + x$$

for certain $A, B \in K$ and with $B(A^2 - 4) \neq 0$.

「Twisted Edward Curve」定义如下：

Definition [edit]

Each twisted Edwards curve is a twist of an Edwards curve. A twisted Edwards curve $E_{E_{a,d}}$ over a field $\mathbb{K}$ which have $\mathrm{char}(\mathbb{K}) \neq 2$ is an affine plane curve defined by the equation:

$$E_{E_{a,d}} : ax^2 + y^2 = 1 + dx^2y^2$$

where a, d are distinct non-zero elements of $\mathbb{K}$. The special case $a = 1$ is *untwisted*, because the curve reduces to an ordinary Edwards curve.

Every twisted Edwards curve is birationally equivalent to an elliptic curve in Montgomery form and vice versa.[2]

「Curve25519」定义如下：

Mathematical properties [edit]

The curve used is $y^2 = x^3 + 486662x^2 + x$, a Montgomery curve, over the prime field defined by the prime number $2^{255} - 19$, and it uses the base point $x = 9$. This point generates a cyclic subgroup whose order is the prime $2^{252} + 27742317777372353535851937790883648493$ and is of index 8. Using a prime order subgroup prevents mounting a Pohlig–Hellman algorithm attack.[4]

The protocol uses compressed elliptic point (only *X* coordinates), so it allows efficient use of the Montgomery ladder for ECDH, using only *XZ* coordinates.[5]

Curve25519 is constructed such that it avoids many potential implementation pitfalls.[6] By design, it is immune to timing attacks and it accepts any 32-byte string as a valid public key and does not require validating that a given point belongs to the curve, or is generated by the base point.

The curve is birationally equivalent to a twisted Edwards curve used in Ed25519[7][8] signature scheme.[9]

「Ed25519」的定义如下：

Ed25519 [edit]

Ed25519 is the EdDSA signature scheme using SHA-512 (SHA-2) and Curve25519[2] where

- $q = 2^{255} - 19$,
- $E/\mathbb{F}_q$ is the twisted Edwards curve

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2,$$

- $\ell = 2^{252} + 27742317777372353535851937790883648493$ and $c = 3$
- B is the unique point in $E(\mathbb{F}_q)$ whose y coordinate is $4/5$ and whose x coordinate is positive.
  "positive" is defined in terms of bit-encoding:
  - "positive" coordinates are even coordinates (least significant bit is cleared)
  - "negative" coordinates are odd coordinates (least significant bit is set)
- H is SHA-512, with $b = 256$.

The curve $E(\mathbb{F}_q)$ is birationally equivalent to the Montgomery curve known as Curve25519. The equivalence is[2][7]

$$x = \frac{u}{v}\sqrt{-486664}, \quad y = \frac{u-1}{u+1}.$$

根据Wikipedia，大概可以整理出这么几个信息：

1. Edward Curve是Twisted Edward Curve中的一种
2. Twisted Edward Curve和Montgomery Curve可以互相转换
3. Edward Curve和Montgomery Curve这两种曲线都具有特殊属性，例如能够为计算加速
4. Curve25519是一种曲线，Ed25519是一种签名算法
5. Curve25519又是精选的Montgomery Curve，具有更高的计算效率
6. Curve25519和Ed25519采用的曲线是一致的，一个是Montgomery表现形式，一个是Twisted Edward Curve表现形式

7. 25519的取名来自于该曲线的有限域参数p=2255- 19

在阅读Wikipedia的过程中发现一个名字「Weierstrass equation」，原来它才是这些曲线的鼻祖，在一个域k上的任何一个平面曲线，都可以表示成Weierstrass equation。

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \quad a_1, a_2, a_3, a_4, a_6 \in k.$$

不难发现，前面提到过的各个公式都是Weierstrass equation的一种演化版本（Twisted Edward Curve看起来不是，但是它可以转换到Montgomery Curve，本质上也一样）到此，为椭圆曲线寻根问到祖，并且从「STD」也获知了椭圆曲线家族的族谱。

「STD」中罗列了多个其他标准，例如Brainpool曲线系列、BN曲线系列、MNT曲线系列等，这些系列的背后都代表了一种独特的曲线生成哲学，或是为了提供可验证的随机数，或是为了提供满足Paring的特性，或是为了提高抗攻击的能力等等，每一份精心选择的参数都是一群数学家们巧夺天工的设计。

后话

古有拆文解字，参透汉字玄机，道破人生天机；而今咬文嚼字，摸清椭圆原理，揭开曲线家谱。始于名字，解码secp256k1，厘清标准；终于名字，问祖weierstrass，致敬大神。

通过了解椭圆曲线之间的内在关系，对其设计有了更多一点的理解。知道的更多了，不知道的也更多了，那些特殊曲线的数学原理是什么？为什么具有更高的计算效率？性能能提升多少？... 又是一个深夜，拥抱最新的收获，夹杂更多的困惑。电脑中正好在播放“把太细的神经割掉，会不会比较睡得着...”

参考资料

- SEC1
- SEC2
- NIST
- STD
- SafeCurves
- Koblitz Curves
- Weierstrass Equation
- Montgomery Curve wiki
- Twisted Edward Curve wiki
- Edward Curve wiki
- ECDSA wiki
- Curve25519 wiki
- Curve25519 paper
- Ed25519 paper

22.4.8 安全控制

存储安全：支持落盘数据加密存储

FISCO BCOS的落盘加密

作者：石翔 | FISCO BCOS 核心开发者

区块链部署是多方参与的，为了简化多方协作环境的搭建，通常会使用公有云部署区块链。机构将自己的节点部署到云上，让服务与云上的节点进行交互，实现多方协作。在这个架构中，机构内部的安全性是很高的，尤其是在金融机构。 虽然通过网络的隔离机制，将节点限制在“内网”中，通过网络不能轻易地盗取数据，但是数据全部托管在云上，由于所有参与者都会保存一份数据，在网络和系统安全措施有疏漏或操作不当等极端情况下，可能出现某一份数据被越权访问的情况。 为了防止数据盘被攻破或者盗取，避免数据泄露情况的发生，FISCO BCOS引入了“落盘加密”的功能。

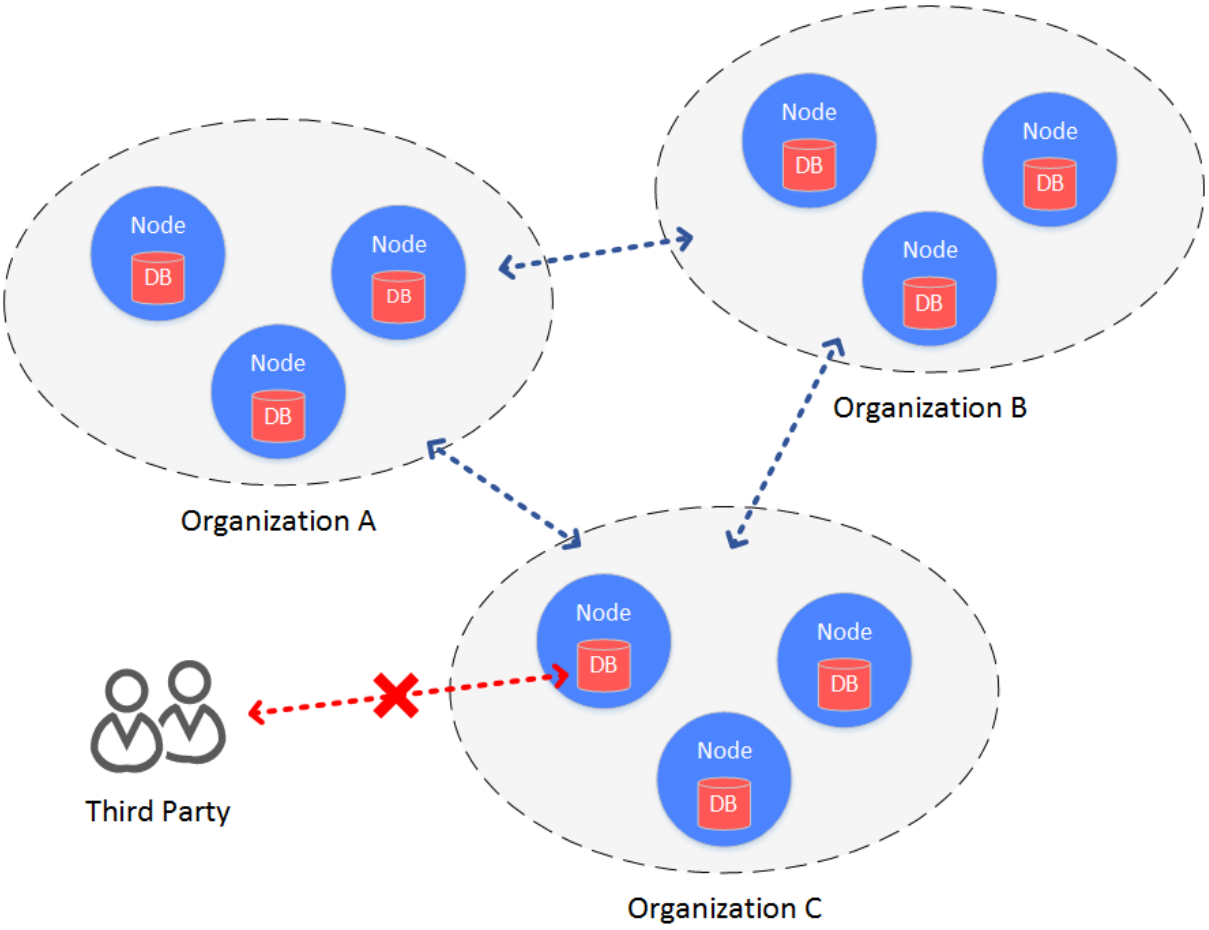
背景架构

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

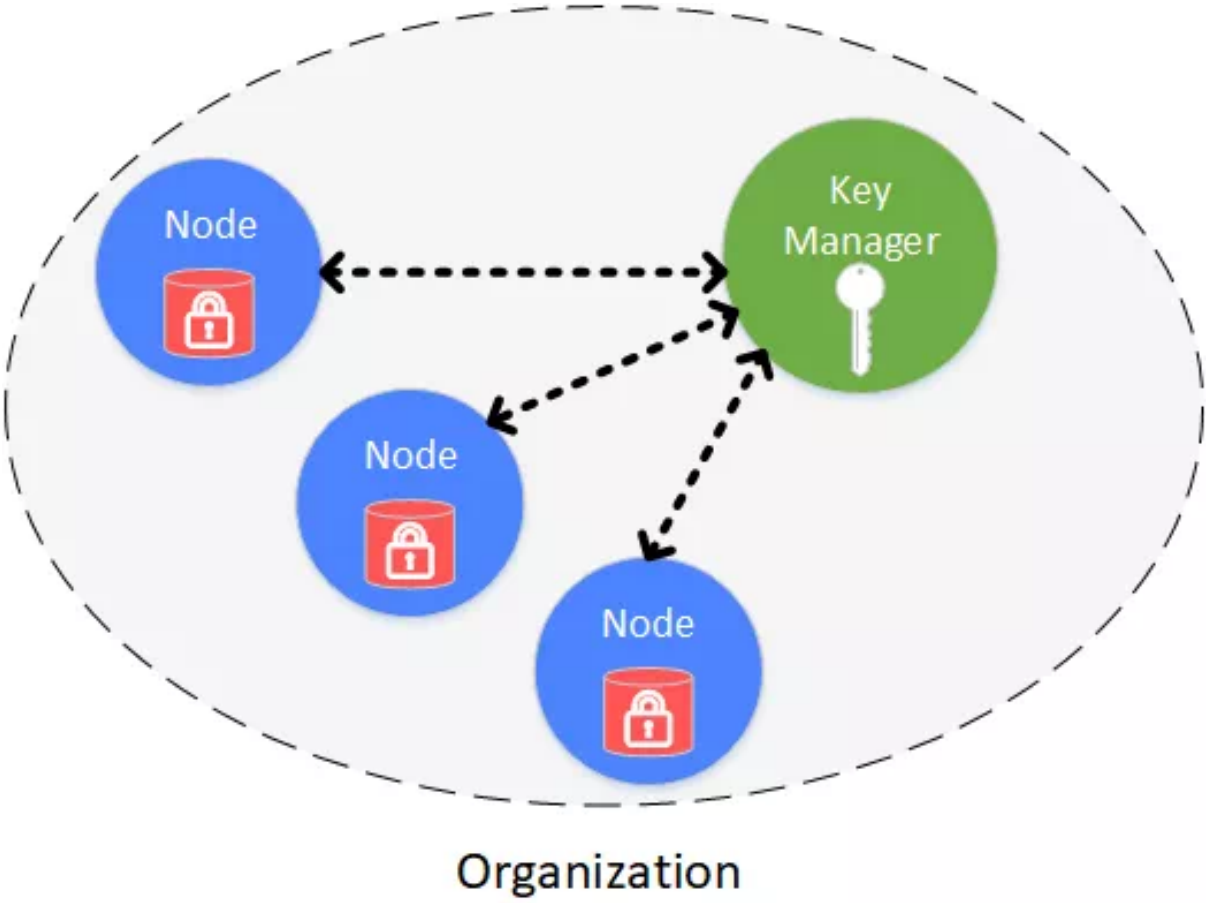
对于链上通信数据的访问控制，FISCO BCOS是通过节点证书和SSL来完成。此处主要介绍节点存储数据的访问控制，即落盘加密。



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部署在机构内网里，是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：rocksdb或leveldb
- 节点私钥：node.key，gmnode.key（国密）

实现原理

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。

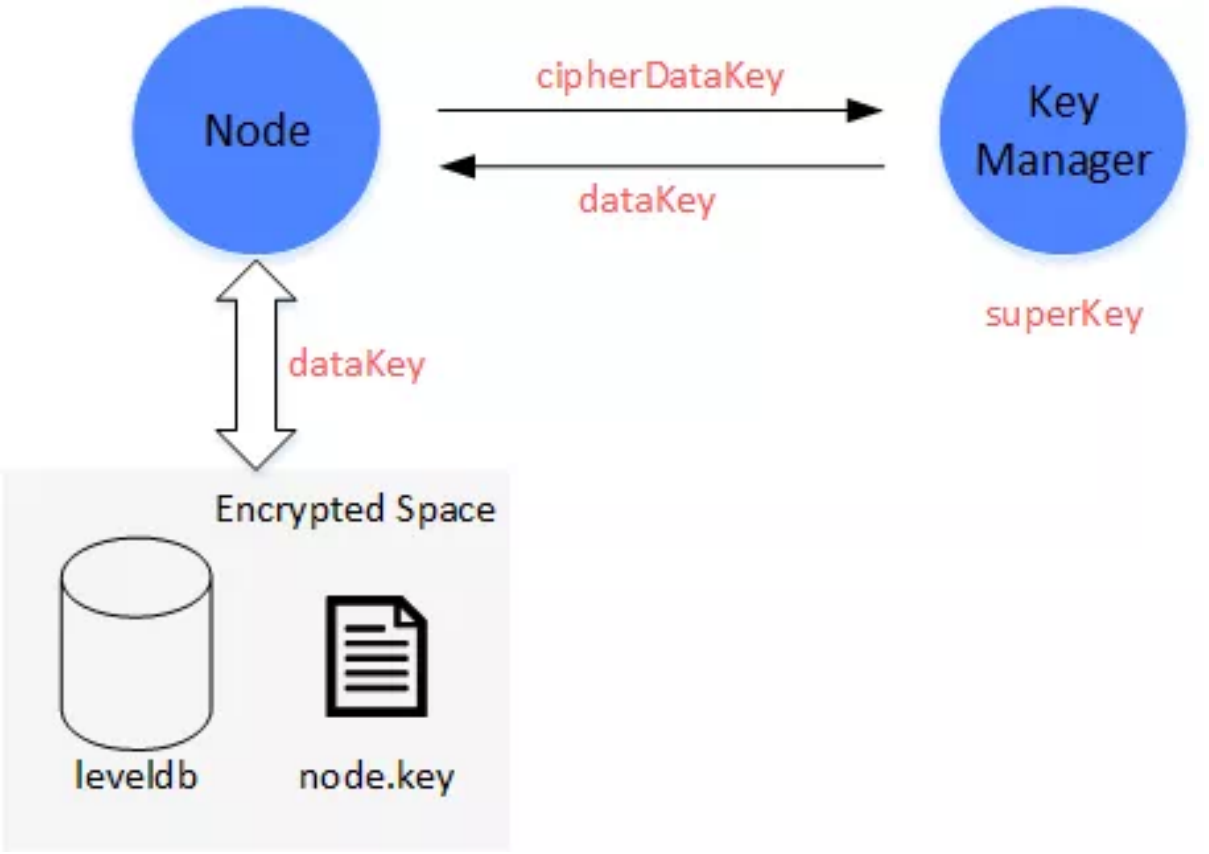
节点

- 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。

- Key Manager在节点启动时必须在线，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，机构外的外网无法访问Key Manager。



相关操作

Key Manager操作

在每个机构上启动一个key-manger程序，用如下命令启动，指定Key Manager：

```
# 参数：端口，superkey
./key-manager 31443 123xyz
```

节点操作

（1）配置新节点dataKey

```
# 参数：Key Manager IP，端口，dataKey
bash key-manager/scripts/gen_data_secure_key.sh 127.0.0.1 31443 12345
```

得到cipherDataKey，脚本自动打印出落盘加密所需要的ini配置(如下)。此时得到节点的cipherDataKey：cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea。

```
[storage_security]
enable=true
key_manager_ip=127.0.0.1
key_manager_port=31443
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。

（2）加密新节点的私钥

执行脚本，加密节点私钥：

```
# 参数：ip port 节点私钥文件 cipherDataKey
bash key-manager/scripts/encrypt_node_key.sh 127.0.0.1 31443
nodes/127.0.0.1/node_127.0.0.1_0/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。

注意：

国密版比非国密版需要多加密一个文件：

- 非国密版：conf/node.key
- 国密版：conf/gmnode.key和conf/origin_cert/node.key

（3）启动节点

直接启动节点即可。如果这个节点所在的硬盘被带出机房（内网），将无法访问Key Manager。节点拿不到自己的dataKey，无法解密硬盘数据，也无法解密自己的私钥，因而无法启动。

注意事项

- Key Manager是一个demo版本，目前superKey是通过命令行在启动的时候指定，在实际应用中，需要根据安全要求，自己定制加载superKey的方式，比如使用加密机进行管理。
- 落盘加密的配置只针对新生成的节点，节点一旦启动，将无法转换成带落盘加密的节点。
- 国密版的落盘加密比非国密版多加密一个私钥。

参考链接

Key Manager源码

通信安全：支持全流程SSL

FISCO BCOS的TLS1.2握手流程

作者：李昊轩 | FISCO BCOS 核心开发者

写在前面

之前我们说过，互联网上应该质疑一切，因此采用传统明文方式发送数据包无疑是非常不安全的。因此https协议中要求需要加密，那么加密前需要做什么，本文将对握手建立前，通信双方的具体流程进行分析。

SSL通信过程是TLS1.2 **[1]** 的内容。最终目的是安全的传输数据包。SSL通信的核心是通过证书验证身份，之后通过"握手"交互生成一个本次通信的非对称会话主密钥。然后在本轮通信过程中，通过这个会话主密钥对数据包进行加解密，并在网络上只传递密文。

本文将以FISCO BCOS节点通信双向认证为例，讲解通信双方如何加载、使用、验证证书，并如何生成会话主密钥。

Part 1 主密钥设置

网络压缩主要在P2P网络层实现，系统框架如下： TLS中的主密钥是一个对称密钥，即client端和server端使用的密钥是相通的，握手的过程也就是双方通过交互一些随机数，来完成主密钥的设置。

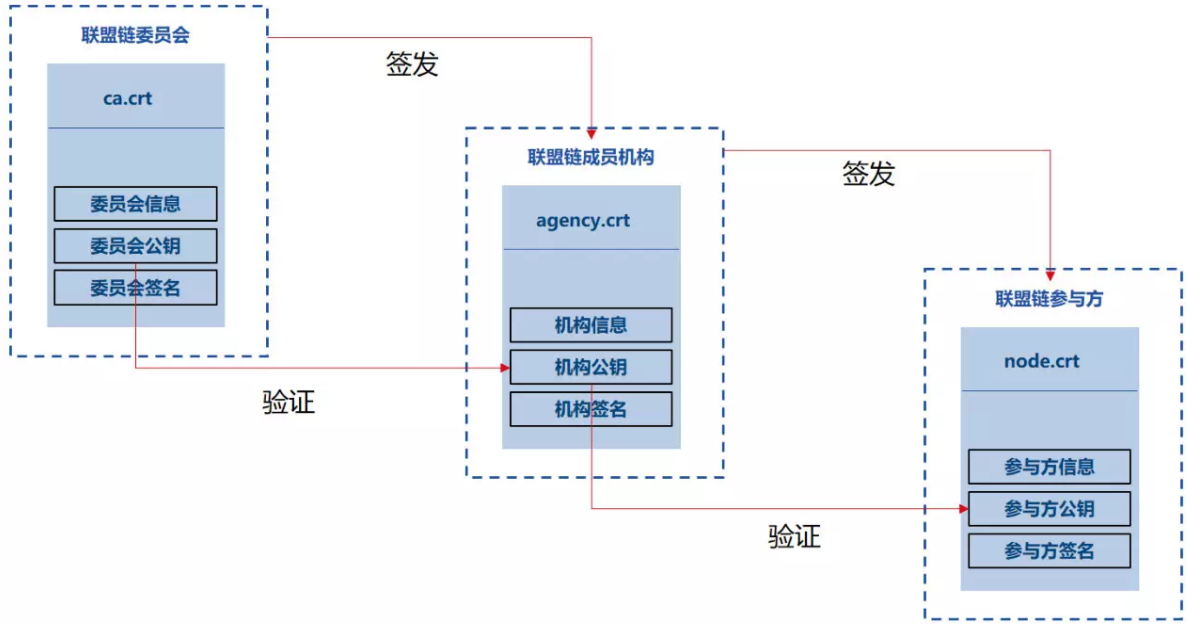
我们以DH密钥交换算法x协商主密钥为例，设攻击者为Eve，Eve可以获取他们传输的中间值：

1. Alice和Bob先协商公平一个大素数p，和生成元g，Eve可以获取p和g；
2. Alice选择随机整数a ∈ Zp，计算A=g^a modp，将A发送给Bob，Eve可以获取A
3. Bob选择随机整数b ∈ Zp，计算B=g^b modp，将B发送给Alice，Eve可以获取B
4. Alice计算S=B^a = g^(ab) modp
5. Bob计算S=A^b = g^(ab) modp

通过上述过程，Alice和Bob协商出了一个密钥S，虽然Eve获取了中间值A，B，p，g，根据离散对数问题，Eve无法获取S具体的值。

Part 2 证书验证

之前的文章中我们讲到 证书的签发流程（点击阅读），在本节中将以三级证书为例，讲解证书的验证过程。TLS握手中的证书验证，即通过加载双方预先都信任的根证书，从而认证对方的身份。



▲感谢社区核心开发者黎连文贡献的图

- 在程序启动时，会首先加载本地的ca.crt和node.crt；
- 当节点验证对方的证书时，会首先使用对方node.crt中的公钥验证附带的签名，当验证通过时，可以确认对应节点拥有当前node.crt对应的node.key；
- 之后节点会使用agency.crt中的信息验证node.crt是否为合法的agency签发；
- 最后节点会使用本地加载ca.crt中的信息验证agency.crt是否为联盟链委员会签发；
- 当两者都验证通过时，表示节点接收到的node.crt是由本地加载的ca.crt签发的。

Part 3 TLS握手流程

下图是我们采用对两台服务器进行抓包得到了TLS握手流程：

```
TCP          74 58606 → 30370 [SYN] Seq=0 Win=29200 Len=0 MSS=1424 SACK_PERM=1 TSval=3015179707 TSecr=0 WS=
TCP          66 58606 → 30370 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3015179707 TSecr=1424423810
TLSv1.2     236 Client Hello
TCP          66 58606 → 30370 [ACK] Seq=171 Ack=1413 Win=32128 Len=0 TSval=3015179712 TSecr=1424423814
TCP          66 58606 → 30370 [ACK] Seq=171 Ack=1645 Win=34944 Len=0 TSval=3015179712 TSecr=1424423814
TLSv1.2    1478 Certificate
TLSv1.2     228 Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message
TLSv1.2     112 Application Data
TCP          66 58606 → 30370 [ACK] Seq=1791 Ack=2493 Win=37760 Len=0 TSval=3015179765 TSecr=1424423827
TCP          66 58606 → 30370 [ACK] Seq=1791 Ack=2608 Win=37760 Len=0 TSval=3015179841 TSecr=1424423943
TCP          66 42280 → 36000 [ACK] Seq=253 Ack=469 Win=1349 Len=0 TSval=3015179924 TSecr=1424424027
TLSv1.2     920 Application Data
```

▲Client端

```
TCP          66 30370 → 58606 [ACK] Seq=1 Ack=171 Win=30080 Len=0 TSval=1424423813 TSecr=3015179710
TLSv1.2    1478 Server Hello
TLSv1.2     298 Certificate, Server Key Exchange, Certificate Request, Server Hello Done
TCP          66 30370 → 58606 [ACK] Seq=1645 Ack=1745 Win=35712 Len=0 TSval=1424423825 TSecr=3015179722
TLSv1.2     868 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
TLSv1.2     112 Application Data
```

▲server端

在流程中我们一共看到了6个数据包，分别为：

- client向server发送的client hello
- server向client发送的server hello，发送server证书，进行参数协商
- client向server发送的client证书，进行参数协商
- server向client发送的结束标志，握手完成

(1) client hello

在这步中，client向server发起建连请求，并告知server以下信息：

```
{
client端支持的TLS协议
支持xxx加密算法
密钥协商过程需要的随机数
}
```

```
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 165
  ▼ Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 161
      Version: TLS 1.2 (0x0303)
    ▶ Random: a9050ddcfdefc866e114612c7465984a644ec696e8863653…
      Session ID Length: 0
      Cipher Suites Length: 56
    ▶ Cipher Suites (28 suites)
      Compression Methods Length: 1
    ▶ Compression Methods (1 method)
      Extensions Length: 64
    ▶ Extension: ec_point_formats (len=4)
    ▶ Extension: supported_groups (len=4)
    ▶ Extension: session_ticket (len=0)
    ▶ Extension: encrypt_then_mac (len=0)
    ▶ Extension: extended_master_secret (len=0)
    ▶ Extension: signature_algorithms (len=32)
```

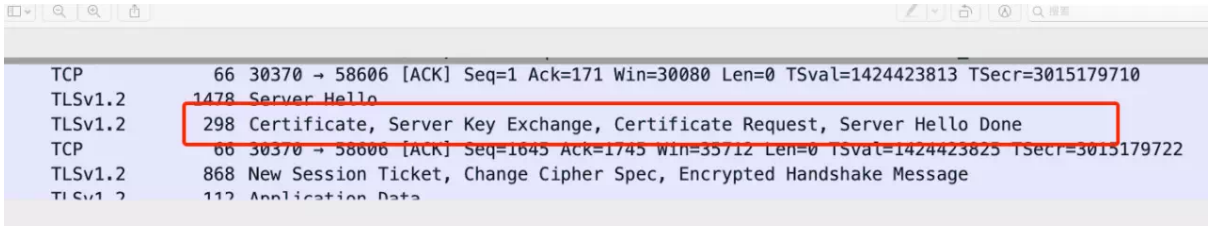
(2) server hello

```
▶ Frame 14: 298 bytes on wire (2384 bits), 298 bytes captured (2384 bits)
▶ Ethernet II, Src: fe:ee:5f:9f:e7:8e (fe:ee:5f:9f:e7:8e), Dst: RealtekU_84:66:ff (52:54:00:84:66:ff)
▶ Internet Protocol Version 4, Src: 10.107.105.81, Dst: 10.107.105.58
▶ Transmission Control Protocol, Src Port: 30370, Dst Port: 58606, Seq: 1413, Ack: 171, Len: 232
▶ [2 Reassembled TCP Segments (1365 bytes): #13(1342), #14(23)]
▼ Transport Layer Security
  ▶ TLSv1.2 Record Layer: Handshake Protocol: Certificate
▼ Transport Layer Security
  ▶ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
  ▶ TLSv1.2 Record Layer: Handshake Protocol: Certificate Request
  ▶ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
```

server端接收到client的请求后，对其进行回应，包括：

```
{
我本次使用的TLS协议
这是我的随机数
本次会话我们的session id
本次我们使用的加密算法
}
```

(3) server communicate



如图所示，共包含四个部分:

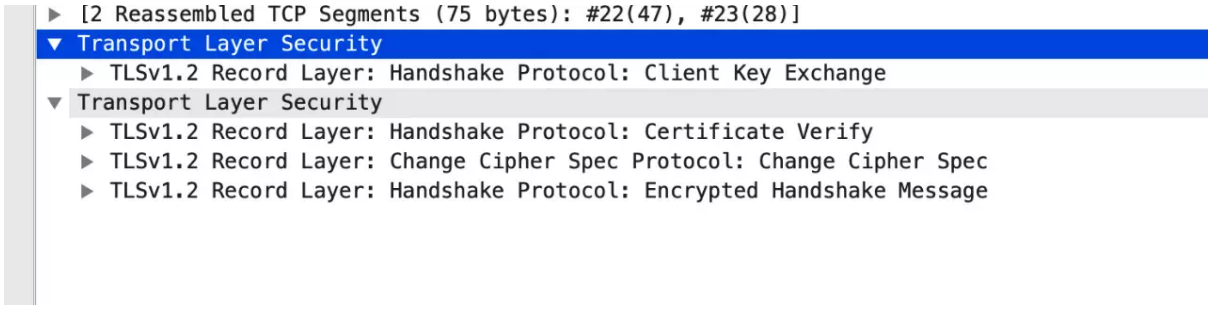
```
{
server端证书certificate
密钥交换参数server key exchange
验证client端的请求 certificate request
server端 发送完毕 server hello done
}
```

(2)(3)两步的操作可以总结为：server端对client端做出了“我们使用xx算法协商会话主密钥，这是我的证书，这是我的协商参数，请把你的证书发给我”的回应。

(4) client certificate

client端收到server的certificate request后，会将自己的证书发送给对方。

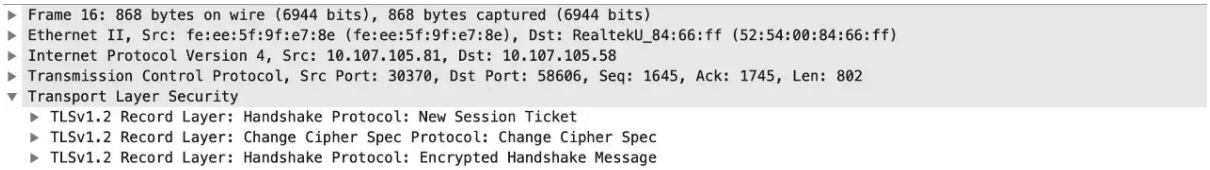
(5) client communicate



在这步中，client会先使用本地加载的ca.crt 验证server证书，之后进行参数协商操作，数据包中内容有:

```
{
client 密钥交换参数
client 对server端证书的验证结果
使用会话主密钥的密文传输内容
client对(1)(2)(3)过程的签名
}
```

(6) server handshake established



server端接收到数据包后，会开始使用会话主密钥对数据包进行加密传输。

参考资料

【1】TLS(Transport Layer Security)：安全传输层协议用于在两个通信应用程序之间提供保密性和数据完整性。该协议由两层组成： TLS 记录协议（TLS Record）和 TLS 握手协议（TLS Handshake）。

【2】Discrete Logarithm Problem

证书管理：支持证书颁发、撤销、更新

FISCO BCOS证书说明

作者：李昊轩 | FISCO BCOS 核心开发者

联盟链中，链上的多方参与是一种协作关系；联盟链向授权的组织或机构开放，有准入机制。在准入机制中，证书是各参与方互相认证身份的重要凭证；可以说，**证书机制是联盟链网络安全的基石**。

Part1：FISCO BCOS 证书结构

FISCO BCOS网络采用面向CA的准入机制，使用x509协议的证书格式，支持任意多级的证书结构，保障信息保密性、认证性、完整性及不可抵赖性。根据现有业务场景，FISCO BCOS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。

▲ 图：x509协议的证书格式

证书内容包括了证书版本号、序列号、证书的签名算法、消息摘要算法等生成信息；同时包括了证书的颁发者、有效期、使用者、公钥信息、SSL通信需要用到的密码套件等信息。节点通过加载证书，在接受数据包时，根据证书规定的密码套件和其消息字段，对数据包中携带的证书进行验证。

Part2：角色定义

FISCO BCOS的证书结构中有 4 种角色，分别是联盟链委员会、联盟链成员机构、联盟链参与方（节点和SDK）。

1. 联盟链委员会

联盟链委员会拥有联盟链的根证书ca.crt和私钥ca.key，通过使用ca.key对联盟链成员机构签发机构证书，负责完成联盟链成员机构的准入、剔除等操作。

2. 联盟链成员机构

联盟链成员机构为经过联盟链委员会许可，加入到联盟链的机构。联盟链成员机构拥有机构私钥agency.key和经过根私钥ca.key签发的机构证书agency.crt。联盟链成员机构可以通过机构私钥签发节点证书，从而配置本机构的节点和SDK。

3. 联盟链参与方

联盟链参与方可以通过运行节点或SDK联盟链交互，拥有与其他节点进行通信的节点证书node.crt和私钥node.key。联盟链参与方运行节点或SDK时，需要加载根证书ca.crt和相应的节点证书node.crt和私钥node.key；与其他成员进行网络通信时，使用预先加载的证书进行身份认证。

Part3：证书生成流程

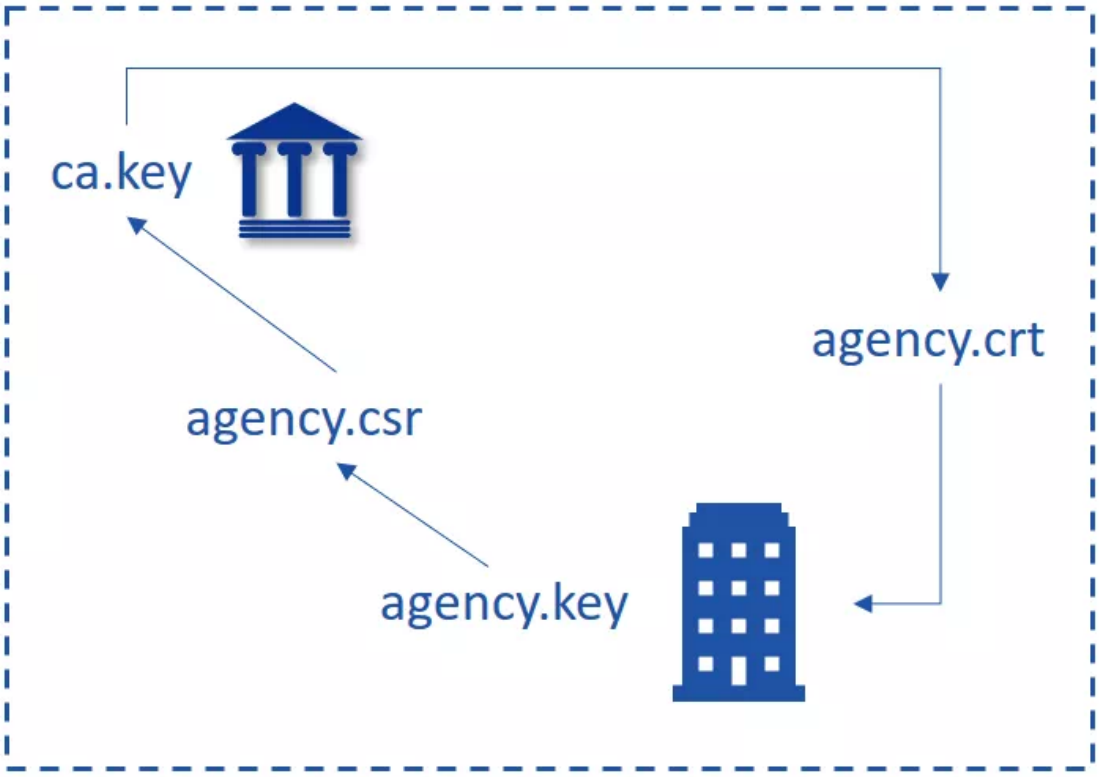
1. 联盟链委员会初始化根证书ca.crt

- 本地生成私钥ca.key；自签生成根证书ca.crt。

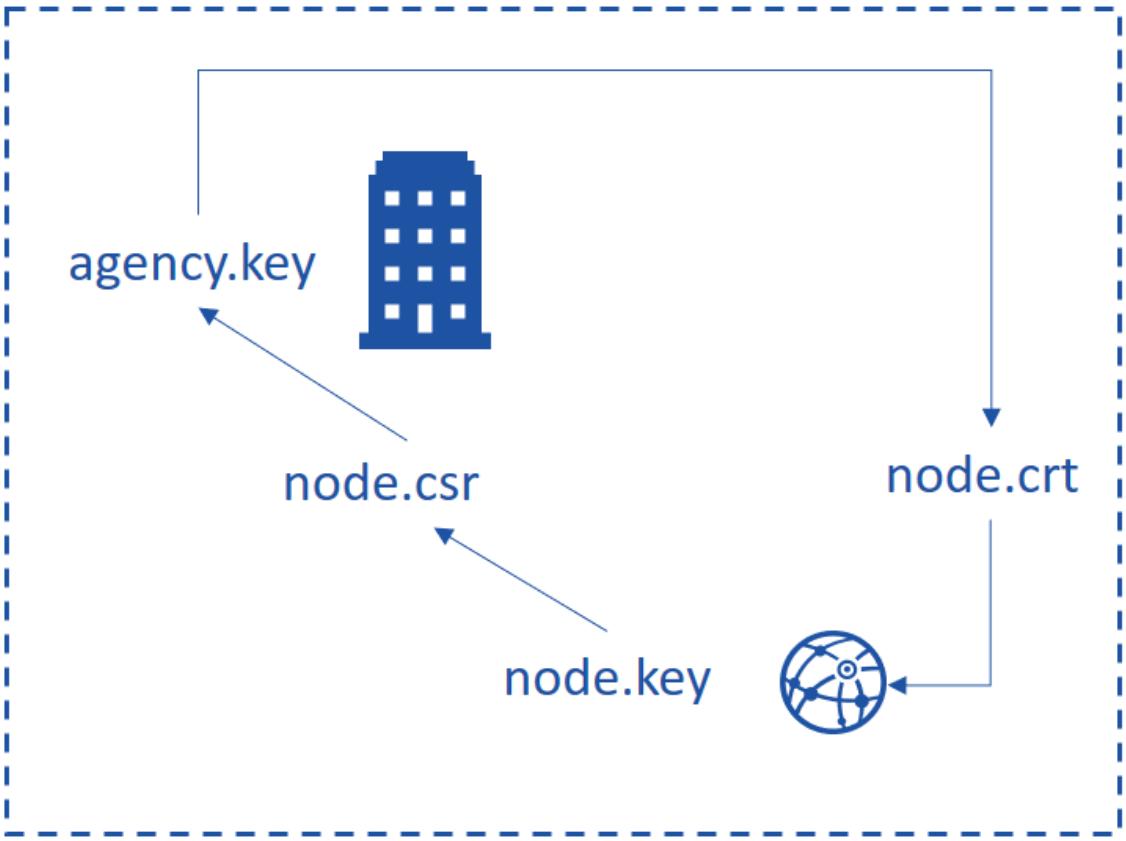
2.联盟链成员机构获取机构证书agency.crt

- 本地生成私钥agency.key；

- 由本地私钥生成证书请求文件agency.csr；
- 将证书请求文件agency.csr发送至联盟链委员会；
- 联盟链委员会使用ca.key对证书请求文件agency.csr进行签发，得到联盟链成员机构证书agency.crt；
- 联盟链委员会将联盟链成员机构证书agency.crt发送至对应成员。



3.节点/SDK获取证书node.crt



- 本地生成私钥node.key；
- 由本地私钥生成证书请求文件node.csr；
- 将证书请求文件node.csr发送至联盟链成员机构；
- 联盟链成员机构使用agency.key对证书请求文件node.csr进行签发，得到节点/SDK证书node.crt；
- 联盟链成员机构将节点证书node.crt发送至对应实体。

算法类型	标准版FISCO BCOS
签名	ECDSA (公钥长度: 512 bits, 私钥长度: 256 bits)
哈希	SHA3 (哈希串长度: 256 bits)
对称加解密	AES (加密秘钥长度: 256 bits)
交易长度	520bits(其中标识符8bits,签名长度512bits)

▲ 图：节点私钥相关说明

Part4：证书生成实例

以生成联盟链成员机构机构证书为例：

1.机构首先在本地使用机构私钥agency.key生成证书请求文件agency.csr；

```
openssl req -new -sha256 -subj "/CN=$name/O=fisco-bcos/OU=agency" -key ./agency.
↪key -config ./cert.cnf -out ./agency.csr
```

2.联盟链委员会根据证书请求文件生成机构证书agency.crt；

```
openssl x509 -req -days 3650 -sha256 -CA ./ca.crt -CAkey ./ca.key -CAcreateserial -
↪in ./agency.csr -out ./agency.crt  -extensions v4_req -extfile ./cert.cnf
```

上述过程中的cert.cnf为证书信息配置项，可以采用默认设置。

Part5：小结及参考文档

本文介绍了FISCO BCOS采用的相关证书说明和其对应的层级架构；在后续文章中，我们会具体讲解SSL握手过程中如何使用这些数字证书。

参考资料

- 证书说明
- 权限控制
- 节点准入

FISCO BCOS上使用第三方CA证书底层节点部署实操

作者：林宣名 | FISCO BCOS开源社区贡献者

CA证书怎么生成？节点相互验证证书时会交叉验证吗？对于社区常遇到的此类问题，分享一些个人使用第三方CA证书部署底层节点的经验，希望可以给大家一些借鉴与参考。

为什么要对第三方CA证书进行改造？

首先，说明一下我进行第三方CA证书改造的背景和原因：

- 社区内经常有人会问到第三方CA证书的改造问题，个人感觉这是大家都关注的要点。
- 在我们的一些项目中，业务方指定要使用第三方CA证书，实际的生产需求也要求我们进行CA改造。
- 在司法领域区块链存证场景中，需要由具有电子认证许可证书机构出具的认证证书才能作为电子认证。

鉴于以上三点，我觉得大家对如何进行第三方CA证书改造都很关注。

FISCO BCOS技术文档中提供了CFCA证书改造的案例，但在一些细节上还待完善，因此我想写一篇教程，结合生产环境改造、第三方CA配合、合规性、技术实现等内容具体说明，看看能否对其他社区用户有所帮助。

除了司法领域存证，还有哪些场景需要由第三方CA机构参与?

基本上用到CA证书的区块链场景都有可能用到第三方CA证书，是否采用第三方CA证书主要考虑：

- 联盟链是否需要第三方CA机构背后的相关资质。
- 联盟链中，参与方对节点准入管理、以及后续控管是否需要第三方CA机构作为公正机构来签发证书，防止自建CA体系中存在任意签发证书导致节点作恶等问题。

两级证书模式下，为什么需要配置白名单列表？如果不配置会有什么问题?

两级证书模式下使用第三方提供CA证书作为链证书，如果不配置白名单，只要是CA证书签发节点证书都可以连接到这条链上，配置白名单可以实现准入拦截。

实操步骤教学

接下来，我们来看看使用第三方CA证书部署底层节点的具体实操步骤。

此次改造要点为：

- FISCO BCOS的底层CA缺省提供的是三级模式，链证书–>机构证书–>节点证书；
- 现实环境中CA方虽然可以提供三级签发的证书，但在一些场景下有合规风险；
- 目前我们采用的做法是剔除机构证书，从链证书–>节点证书的签发，其中链证书由CA方提供的CA.crt，配合白名单机制使用完成基础底层节点部署。

环境准备

1. 两台测试服务器：118.25.208.8、132.232.115.126
2. 操作系统为ubuntu:18.04
3. openssl 工具使用ubuntu 18.04自带 openssl 1.1.1
4. 选用普通版FISCO BCOS 2.5.0 版本，节点使用的节点证书算法为EC secp256k1曲线
5. 结合【白名单机制】一起使用

   https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/certificate_list.html#id2

备注：测试过程中节点私钥和请求证书文件统一管理，但在生产环境中节点私钥应由各机构管理员进行生成，提交给CA方，私钥各自留存。

基础证书准备

生成基础节点私钥和节点证书请求文件

采用openssl 工具，要求1.0.2版本以上，生成对应的节点私钥和节点证书请求文件、以及对应的node.nodeid（nodeid 是公钥的十六进制表示）。（说明：以下每个节点第四步生成node.nodeid 中的node.key，都是对应节点的cert_IP_port.key 修改的，该操作是底层要求的。）

- 生产节点1 node_118.25.208.8_30300 相关文件

```
（1）生成cert_118.25.208.8_30300.param: openssl ecparam -out   cert_118.25.208.8_30300.param
-name secp256k1
（2）生成cert_118.25.208.8_30300.key: openssl genpkey -paramfile cert_118.25.208.8_30300.para
m -out  cert_118.25.208.8_30300.key 2> /dev/null
（3）生成cert_118.25.208.8_30300.csr: openssl req -new -sha256 -subj "/CN=fj/O=Hashchain/OU=H
ash" -key  cert_118.25.208.8_30300.key  -out  cert_118.25.208.8_30300.csr
（4）生成对应的node.nodeid:openssl ec -in node.key -text -noout 2> /dev/null | sed -n '7,11p'
| tr -d ": \n" | awk '{print substr($0,3);}' | cat >node.nodeid
```

- 生产节点2 node_118.25.208.8_30301 相关文件

```
（1）生成cert_118.25.208.8_30301.param: openssl ecparam -out   cert_118.25.208.8_30301.param
-name secp256k1
（2）生成cert_118.25.208.8_30301.key: openssl genpkey -paramfile cert_118.25.208.8_30301.para
m -out  cert_118.25.208.8_30301.key 2> /dev/null
（3）生成cert_118.25.208.8_30301.csr: openssl req -new -sha256 -subj "/CN=fj/O=Hashchain/OU=f
c" -key  cert_118.25.208.8_30301.key  -out  cert_118.25.208.8_30301.csr
（4）生成对应的node.nodeid:openssl ec -in node.key -text -noout 2> /dev/null | sed -n '7,11p'
| tr -d ": \n" | awk '{print substr($0,3);}' | cat >node.nodeid
```

- 生产节点3 node_132.232.115.126_30300 相关文件

```
（1）生成cert_132.232.115.126_30300.param: openssl ecparam -out    cert_132.232.115.126_3030
0.param -name secp256k1
（2）生成cert_132.232.115.126_30300.key: openssl genpkey -paramfile  cert_132.232.115.126_303
00.param -out  cert_132.232.115.126_30300.key 2> /dev/null
（3）生成cert_132.232.115.126_30300.csr: openssl req -new -sha256 -subj "/CN=fj/O=Hashchain/O
U=gz" -key  cert_132.232.115.126_30300.key  -out  cert_132.232.115.126_30300.csr
（4）生成对应的node.nodeid:openssl ec -in node.key -text -noout 2> /dev/null | sed -n '7,11p'
| tr -d ": \n" | awk '{print substr($0,3);}' | cat >node.nodeid
```

- 生产节点4 node_132.232.115.126_30301 相关文件

```
（1）生成cert_132.232.115.126_30300.param: openssl ecparam -out    cert_132.232.115.126_3030
0.param -name secp256k1
（2）生成cert_132.232.115.126_30300.key: openssl genpkey -paramfile  cert_132.232.115.126_303
00.param -out  cert_132.232.115.126_30300.key 2> /dev/null
（3）生成cert_132.232.115.126_30300.csr: openssl req -new -sha256 -subj "/CN=fj/O=Hashchain/O
U=gz" -key  cert_132.232.115.126_30300.key  -out  cert_132.232.115.126_30300.csr
（4）生成对应的node.nodeid:openssl ec -in node.key -text -noout 2> /dev/null | sed -n '7,11p'
| tr -d ": \n" | awk '{print substr($0,3);}' | cat >node.nodeid
```

说明：FISCO BCOS V2.5版本中，采用了私钥及EC secp 256k1曲线算法。

CA方进行节点证书签发

提交各个节点的node.csr 文件给CA方，CA方返回一张CA.crt 证书作为链证书，返回四张pem格式的节点证书。（说明：FISCO BCOS中，CA方返回证书的模式为：root -> node -issuer,节点证书中糅合issuer证书内容。）

建链

- step 1： 下载国内镜像，cd ~/ && git clone https://gitee.com/FISCO-BCOS/generator.git

- step 2 ： 完成安装，cd ~/generator && bash ./scripts/install.sh完成安装，如果输出 usage: generator xxx，则表示安装成功
- step 3 ： 获取节点二进制，拉取最新fisco-bcos二进制文件到meta中（国内cdn），如果输出 FISCO-BCOS Version : x.x.x-x，则表示成功
- step 4 ： 机构分配

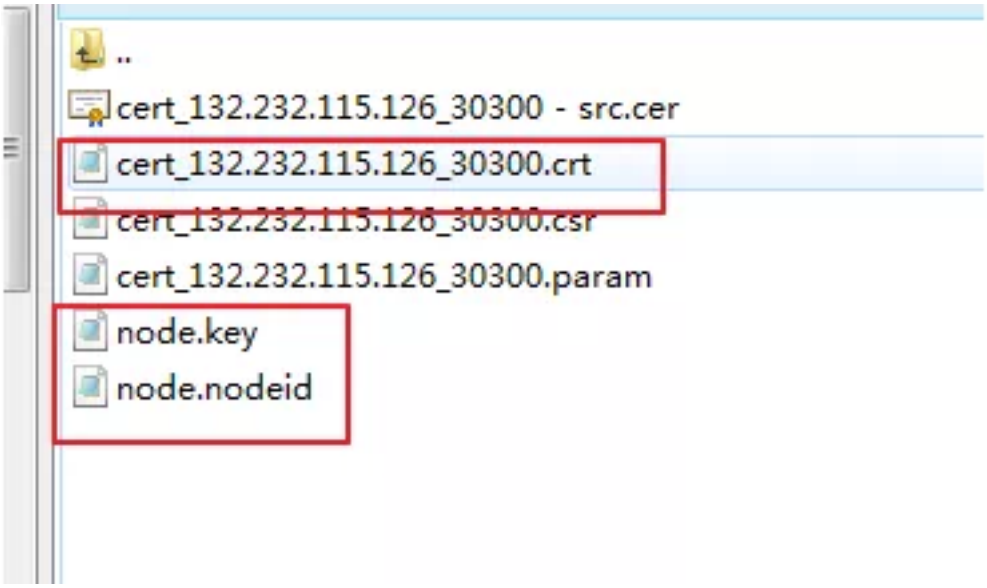
  选用 118.25.208.8 所属机构作为机构A，并由机构A负责创世区块生成

  选用 132.232.115.126 所属机构作为机构B
- step 5 ： 将CA方提供CA.crt 证书作为链证书

  在机构A所属目录手动创建dir_chain_ca目录，并将CA.crt 放到dir_chain_ca目录中
- step 6 ： 在机构A和机构B的meta目录下进行节点证书迁移

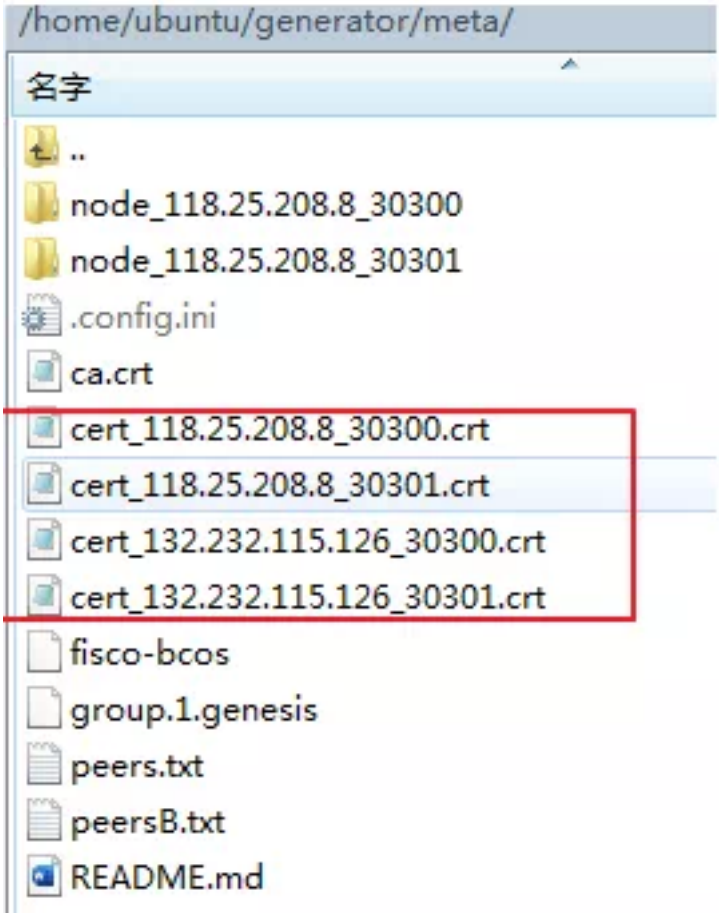
  在meta目录中，手动创建对应的节点目录，其中机构A为：node_118.25.208.8_30300、node_118.25.208.8_30301，机构B为：node_132.232.115.126_30300、node_132.232.115.126_30301

  每个目录需要存放对应的节点证书和节点私钥、节点Id，将CA方生成的节点证书、以及最初准备的节点id、节点私钥等文件统一分发至对应的节点目录，详细如图：



- step 7：机构A收集所有节点证书

  在机构A的meta目录下，收集对应的节点证书，用于后续生成创世区块。如图所示：



- step 8：手动配置机构A修改conf文件夹下的group_genesis.ini，生成创世区块
- step 9：修改机构A、机构B的conf目录下的node_deployment.ini；其中p2p 地址为外网地址、rpc、channel 地址为内网地址
- step 10：在机构meta目录下手动创建peers.txt 文件

  机构A中创建 peers.txt、peersB.txt,机构B 创建 peers.txt、peersA.txt其中以机构A为例，peers.txt 内容如下：

```
118.25.208.8:30300
118.25.208.8:30301
peersB.txt 内容如下:
132.232.115.126:30300
132.232.115.126:30301
```

- step 11：机构A和机构B中生成节点,在机构A的generator中执行命令./generator –build_install_package ./meta/peersB.txt ./nodeA生成机构A的对应节点；在机构B的generator中执行命令./generator –build_install_package ./meta/peersA.txt ./nodeB生成机构B的对应节点
- step 12：运行两个机构的节点：bash ./nodeA/start_all.sh和bash ./nodeB/start_all.sh；共识状态正常则如图所示：

```
ubuntu@VM-16-3-ubuntu:~/generator$ tail -f ./node*/node*/log/log* | grep +++
info|2020-07-14 13:59:58.452310|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=2511ab42...
info|2020-07-14 15:59:58.813077|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=2,tx=0,nodeIdx=0,hash=ab819cbe...
info|2020-07-14 16:59:58.043195|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=2,tx=0,nodeIdx=0,hash=2a3c7cf0...
info|2020-07-14 14:59:58.834800|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=2,hash=786c4595...
info|2020-07-14 17:24:58.310263|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=2,tx=0,nodeIdx=2,hash=34c63d93...
info|2020-07-14 17:25:00.317284|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=2,tx=0,nodeIdx=0,hash=678f7aee...
info|2020-07-14 17:25:02.325213|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=2,tx=0,nodeIdx=2,hash=e25ac71b...
```

- step 13：控制台部署并进行合约部署测试

  针对机构A和机构B对应的控制台操作结果进行比对，两者数据一致，确保共识正常

- step 14：在对应节点的config.ini 配置白名单

至此，我们就完成了第三方CA证书结合底层节点部署的改造。从流程上看，主要是在链证书–>机构证书–>节点证书的生成流程改变了，以及需要在meta目录下进行手动创建peers.txt文件和节点目录等。

结缘FISCO BCOS开源社区

说起我与FISCO BCOS开源社区的结缘也源于CA证书，在一次政企项目对接中，业主方要求区块链底层需适配国密，并使用业主方指定的CA证书。

早期我们团队使用的是其他区块链底层，无法直接适配国密，且改造难度大、周期长、成本高；考虑到后续国内不少项目会涉及到适配国密和CA改造等，我们迫切需要一套完备的区块链底层支持上述需求。通过圈内朋友介绍，了解到FISCO BCOS，在进行深入技术调研、可行性分析后最终选择了FISCO BCOS。

FISCO BCOS开源社区中创设了开放交流的氛围，欢迎大家在社区与我交流讨论。

关于本期分享内容的更多详情，可通过元磁之力进一步了解。

元磁之力论坛是由林宜名和他的团队为FISCO BCOS开源社区贡献的用户交流平台，主要用于分享和学习FISCO BCOS及相关的技术知识。感谢大家为社区做出的各类贡献，您的每一次参与都将成为社区成长的动力！

权限控制：支持细粒度权限控制

FISCO BCOS权限控制一览

作者：张开翔 | FISCO BCOS 首席架构师

作者语

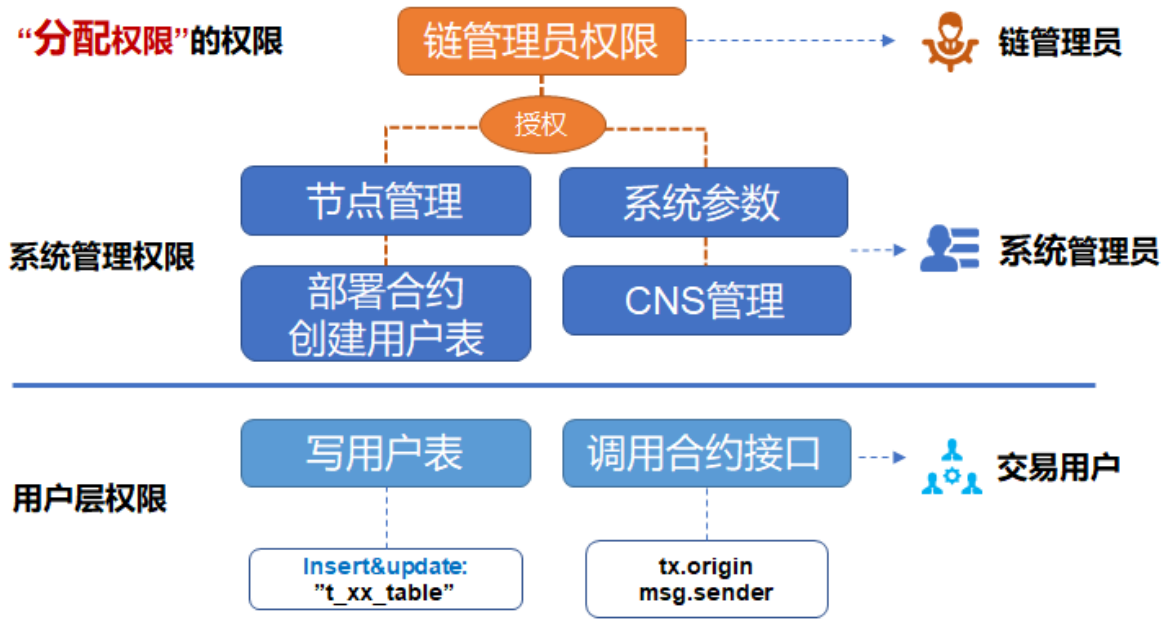
在多方参与的联盟链上，各方分工协作的同时也要做到**职责分明、各司其职**。链的管理者没有必要“既当裁判又当运动员”的参与业务交易，只参与交易的用户也不必很操心智能合约的开发部署。同时，”DO分离”（开发和运维分离）已经是业界成熟的做法，越权操作会带来风险，最终可能破坏信誉和造成资产损失。

清晰、易用、周全的**权限控制能力**，无论是对于信息安全，还是为了完善联盟治理，都至关重要。

本期文章讲的就是FISCO BCOS权限控制这点事，作者从FISCO BCOS的权限分类、典型的联盟链角色设计、权限控制操作基本步骤等角度进行了解析。

FISCO BCOS的权限分类

FISCO BCOS在链刚建立起来的时候，为了方便快速地进行开发和体验，默认并没有做任何权限控制。但如果这条链是用来提供企业级服务，则从一开始就要设计权限控制的策略并落实之。 FISCO BCOS的权限分类：



1. 链管理员权限

即**分配权限的权限**，如定义账户A作为链管理员，A可以为账户B、C、D分配权限；可以设置多个管理员，如果不设置管理员，任何账户都可以无差别修改各种权限。

2. 系统管理权限

目前包括4种:

- 节点管理权限（增删共识节点或观察节点）
- 系统参数修改权限
- 修改CNS合约命名权限
- 能否部署合约和建表的权限

其中部署合约和建表是“二合一”的控制项，在使用CRUD合约时，我们建议部署合约的时候一起把合约里用到的表建了（写在合约的构造函数里），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。

3. 用户表权限

以用户表为粒度，控制某些账户能否**改写某个用户表**，以避免用户表被他人意外修改，这个权限要依赖FISCO BCOS的CRUD合约写法。另外，**读用户表**不受权限控制；如果要进行数据的隐私控制，需要引入数据加密、零知识等技术。

4.合约接口权限

一个合约可以包括多个接口，由于合约里的逻辑和业务密切相关，接口粒度的权限控制由开发者实现，开发者可对msg.sender或 tx.orgin进行判断，决定是否允许本次调用继续处理。

FISCO BCOS的控制台提供了一系列控制权限的命令，可以很方便的为用户**授予（grant）、取消(revoke)、查看(list)**各种权限，可以参见控制台的文档。

联盟链中典型的权限管理角色设计

在联盟链里，不同的角色各司其职，分工协作，典型的角色设计可参照：

1. 链管理员

通常是参与链的多方共同选出一个委员会，一个或多个机构可获得管理员权限，以进行人员管理和权限分配。链管理员并不负责任节点管理，修改系统参数，部署合约等系统管理操作。

2. 系统管理员

指定的业务运营人员或系统运维人员，按需分配各种权限，负责日常的链上管理，包括节点增删，系统参数修改等。由链管理员根据大家约定的治理规则来分配权限，比如只允许指定的账户部署合约，则给他们设定合约部署权限，这样其他账户则不能随意部署合约了。

3.交易用户

用户向区块链发送业务交易请求，业务交易主要是调用合约和读写用户表，可以根据业务逻辑，结合用户表权限和合约接口权限来进行灵活的控制。

4. 监管方

给监管方分配哪些系统和用户表权限，可参考具体的监管规则，如监管方只读所有数据，则不用设定特殊的权限。

管理不同角色的帐户又是另一个复杂的问题，既要清晰区分，又要便于使用，还要保证帐号安全；万一帐号丢失后要支持找回，帐号泄漏后重置，以后会另文介绍。

权限控制操作基本步骤

step1

首先采用控制台grantPermissionManager命令设置一个或多个链管理员角色。 **Tips**：登陆控制台时，命令行里输入的是私钥明文，而使用grantPermissionManager命令输入的是“账户地址”，要注意区分。如：

用户私钥：3bed914595c159cbce70ec5fb6aff3d6797e0c5ee5a7a9224a21cae8932d84a4

对应地址：0xf1585b8d0e08a0a00fff662e24d67ba95a438256

那么登陆控制台的命令行是：

```
./start.sh 1 3bed914595c159cbce70ec5fb6aff3d6797e0c5ee5a7a9224a21cae8932d84a4
```

分配管理员权限的命令行是：

```
grantPermissionManager 0xf1585b8d0e08a0a00fff662e24d67ba95a438256
```

当这个账户得到了链管理员权限后，退出当前控制台或切到另一个终端窗口，用这个账户的私钥登陆一次，即可作为链管理员进行后续的操作了。

Tips：一定要记住管理员地址和私钥的对应关系，否则一旦设置管理员权限，只有管理员可以为其他账户分配权限,其他账户的设置操作都会报告没有权限。

step2

用链管理员账户登陆控制台，根据管理策略，依次给其他系统管理员账户分配节点管理权限，系统参数修改权限，CNS权限，部署合约和建表权限等。然后用一个有相应权限的系统管理员账户私钥登陆控制台，如有部署和建表权限的账号，以便进行下一步。

step3

开发者编写CRUD的合约时，一旦合约在链上创建了用户表，则可以采用控制台的grantUserTableManager命令，由有权限的系统管理员指定哪些账号可以创建表和增删改该表的数据，命令行是

```
grantUserTableManager   t_order  0xf1585b8d0e08a0a00fff662e24d67ba95a438256
```

授权0xf1585b8d0e08a0a00fff662e24d67ba95a438256这个账号可以操作**t_order**表，而其他账号不行，如要读写该表，继续添加权限，对一个表有读写权限的账号可以有多个。

step4

对solidity合约里的某个接口，可以参考这样的代码进行控制：

```
                function testFunction() public returns(int256)
{
        require(msg.sender == tx.origin); //这一行的效果是禁止了合约调合约
         if(msg.sender != address(0x156dff526b422b17c4f576e6c0b243179eaa8407) )  //
↪这里为示例，直接写了账户地址明文，实际上开发时可以灵活处理。
        { return -1;    }  //如果调用者和预设的有权限的调用者不同，则返回
    }
```

msg.sender是当前合约调用者的地址，可以是用户也可以是另一个合约（合约调合约时）；tx.origin则是本次交易的真正发起用户，给交易签名的用户。

小结及参考资料

本文介绍FISCOBCOS在基础层面上提供的一些接口和能力，权限控制的合理性和周密程度最终会取决于使用者，关于不同链的场景化治理和安全控制，可以继续深入探讨，以得出最佳实践。

参考资料

- P2P网络详细设计
- 权限控制设计
- 控制台的文档
- 权限控制实操介绍
- CRUD合约开发介绍
- CRUD合约应用示例

群友问答

@李然：请问几个问题：

1）管理员分配表权限，这个操作是不是在每个节点都可以进行?

2）创建角色，这个操作是不是每个节点都可以操作?

如果以上两个问题答案是肯定的，请问是不是只要有了一个节点的超级权限，就可以修改整个网络的数据?

@光路：是肯定的，建立链之前就先协商好链管理员角色由哪一个或几个账号担任，链建立后第一时间进行角色分配，具体可以参见FISCO BCOS权限控制相关文档。

感谢参与本次话题讨论的小伙伴们！开源社区，因你们更美！

FISCO BCOS 角色权限模型的实现

作者：白兴强 | FISCO BCOS 核心开发者

引言

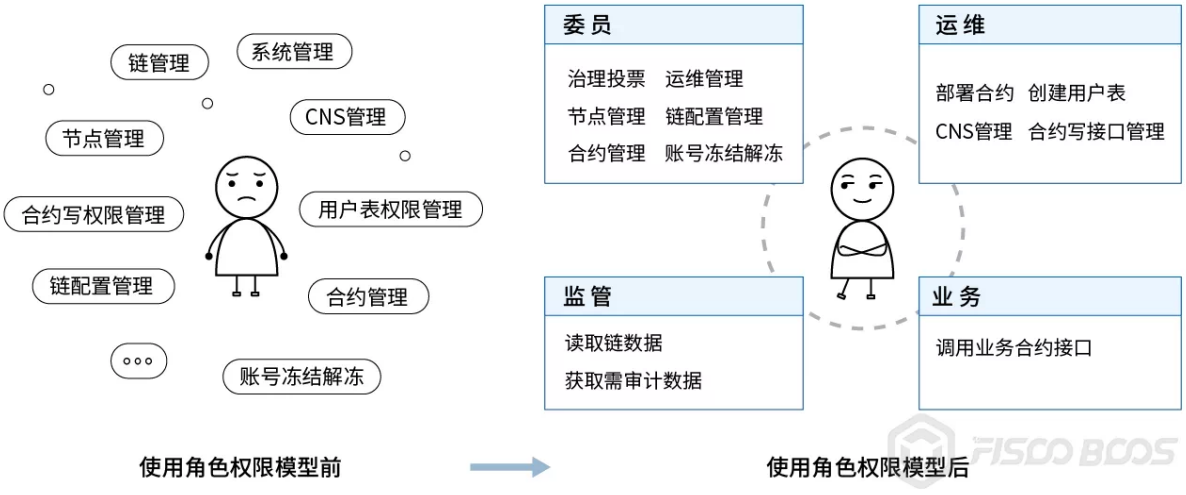
FISCO BCOS的权限控制是通过控制账号对系统中表的写权限来实现的。这种权限控制模型非常灵活且强大，用户几乎可以控制任意一项权限，例如，通过控制权限表的写权限管理分配权限；通过控制系统合约所对应表的写权限管理链配置、节点身份管理、合约部署、用户表创建等；通过控制合约表的写权限管理合约写接口的调用。

然而，绝对完美是不存在的。强大灵活的权限控制也带来较高的学习成本：用户需要理解每个权限项控制的内容以及如何设置，了解链管理员和系统管理员的区别……大量的概念和操作，对用户要求极高。

为了降低使用难度，提升用户体验，FISCO BCOS v2.5对此功能进行了优化，新增了基于角色的权限控制。把不同的权限统归到不同角色，用户根据账号所属角色即可判断其所拥有的权限。同时v2.5基于角色引入了链上治理投票模型，使治理操作更加方便。

什么是角色权限模型?

使用角色权限模型后，用户只需记住角色，而角色对应的权限不言自明，例如，治理委员会委员拥有链治理相关权限，这极大降低用户理解难度与学习成本。



角色对应的权限

区块链上的参与者，可根据角色分为治理方、运维方、监管方和业务方。为避免既当裁判又当运动员，治理方、运维方应权责分离，角色互斥。

- 治理方：角色称为治理委员会委员，简称委员，负责区块链治理。
- 运维方：负责区块链运维，该角色由委员添加。
- 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记录以及需要审计的数据。

各角色所对应权限具体如下表所示。

角色	权限
治理委员会委员 （投票默认阈值为0.5）	投票增加、删除委员
	投票修改委员权重
	投票修改生效阈值
	增加、删除节点（观察/共识）
	修改链配置项
	冻结、解冻合约
	增加、删除运维
	管理用户表的权限
运维	部署合约
	创建用户表
	管理CNS
	冻结、解冻本账号部署的合约
	管理本账号部署合约的写接口
业务	调用运维赋权的业务合约

角色权限实现的细节

本小节将简单介绍委员、运维和业务角色的权限实现细节以及背后原理，以便更好理解与使用角色权限功能。

链初始无委员账号，当存在至少一个委员账号时，委员拥有的权限开始受到控制。联盟链实际应用中多个参与方的技术实力可能并不相同，从实际应用场景出发，我们引入了链上治理投票模型，所有治理操作需要有效投票数/委员数>生效阈值才能生效，用户通过新增的链治理预编译合约可以实现委员的增删、权重修改、投票生效阈值修改等操作。

投票模型有几处值得注意：

- 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
- 投票设置过期时间，根据块高，blockLimit的10倍，固定不可改

运维角色的新增与撤销必须由委员角色操作。链初始无运维账号，当存在至少一个运维账号时，运维拥有的权限开始受到控制。业务账号可以调用链上查询接口与运维指定合约的写接口。

兼容性说明

目前，角色权限模型基于对系统中各类表的写权限控制。我们做了最大努力与之前版本使用体验保持一致，但为了权限控制的完整严谨，对FISCO BCOS v2.5新建链，控制台grantPermissionManager指令不再有效，原PermissionManager的权限归属于委员角色。对于v2.5之前的链，该指令仍然有效。

如何使用角色权限?

本节将以“委员增删”和“运维增删”为例进行简要的实操演示，文档含括了更丰富的角色权限相关操作，欢迎移步查看。

增删委员

使用控制台v1.0.10以上版本中自带的get_account.sh脚本，生成3个如下账号，接下来的操作以这3个账号为例演示。配置好控制台后，使用控制台的-pem选项分别加载3个私钥启动3个控制台。

```
# 账号10x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a.pem# 账
号20x85961172229aec21694d742a5bd577bedffcfec3.pem# 账
号30x0b6f526d797425540ea70becd7adac7d50f4a7c0.pem
```

添加账号1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 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a
{
    "code":0,
    "msg":"success"
}
[group:1]> listCommitteeMembers
-------------------------------------------------------------------------------------------
|                   address                    |                enable_num                |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a   |                    1                     |
-------------------------------------------------------------------------------------------
```

使用账号1添加账号2为委员

此处由于只有账号1是委员，所以使用账号1投票后，满足阈值判断立刻生效。

```
[group:1]> grantCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
-------------------------------------------------------------------------------------------
|                   address                   |                enable_num                 |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                     1                     |
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                     2                     |
-------------------------------------------------------------------------------------------
```

撤销账号2的委员权限

此时系统中有账号1和账号2两个委员，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。

账号1投票撤销账号2的委员权限，如下图：

```
[group:1]> revokeCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
-------------------------------------------------------------------------------------------
|                   address                   |                enable_num                 |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                     1                     |
| 0x85961172229aec21694d742a5bd577bedffcfec3  |                     2                     |
-------------------------------------------------------------------------------------------
```

账号2操作投票撤销账号2的委员权限，如下图：

```
[group:1]> revokeCommitteeMember 0x85961172229aec21694d742a5bd577bedffcfec3
{
    "code":0,
    "msg":"success"
}

[group:1]> listCommitteeMembers
-------------------------------------------------------------------------------------------
|                   address                   |                enable_num                 |
| 0x61d88abf7ce4a7f8479cff9cc1422bef2dac9b9a  |                     1                     |
-------------------------------------------------------------------------------------------
```

增删运维

委员可以添加与撤销运维角色，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服务等。

使用账号1添加账号3为运维

```
[group:1]> grantOperator 0x0b6f526d797425540ea70becd7adac7d50f4a7c0
{
    "code":0,
    "msg":"success"
}

[group:1]> listOperators
-------------------------------------------------------------------------------------------
|                  address                   |                enable_num                 |
| 0x0b6f526d797425540ea70becd7adac7d50f4a7c0 |                    15                     |
-------------------------------------------------------------------------------------------
```

使用账号3部署HelloWorld

账号3是运维角色，可以部署合约，具体操作如下图：

```
[group:1]> deploy HelloWorld
contract address: 0xac1e28ad93e0b7f9108fa1167a8a06585f663726
```

使用账号1部署HelloWorld

账号1是委员，不具有部署合约的权限，部署合约失败，如下图：

```
[group:1]> deploy HelloWorld
permission denied
```

使用账号1撤销账号3的运维权限

账号1是委员可以撤销运维，如下图：

```
[group:1]> revokeOperator 0x0b6f526d797425540ea70becd7adac7d50f4a7c0
{
    "code":0,
    "msg":"success"
}

[group:1]> listOperators
Empty set.
```

总结

作为联盟链的重要特性，权限控制需要做到灵活且强大，但如何在此基础上达成良好的用户体验，需要不断地改进优化。FISCO BCOS v2.5前的权限控制实现了灵活且强大的功能，但同时，社区也收到不少反馈，认为权限控制的使用理解门槛过高。通过角色权限，我们希望能在保持原有功能的同时，降低学

习门槛、提升大家的使用体验。整合优化权限控制的工作仍在进行，希望未来能实现从底层到应用全覆盖的权限控制解决方案。欢迎大家共同探讨交流，积极反馈使用体验与改进建议。

22.4.9 隐私保护

物理隔离：群组间数据隔离 隐私保护协议：支持群签名、环签名、同态加密 场景化隐私保护机制：基于WeDPR支持隐匿支付、匿名投票、匿名竞拍、选择性披露等场景

- FISCO BCOS隐私特性：群/环签名技术实现
- 链上密文参与计算？同态加密尽显神通 | FISCO BCOS隐私特性
- 隐私保护专题

22.4.10 跨链协议

SPV：提供获取SPV证明的接口 跨链协议：基于WeCross支持同构、异构跨链

WeCross跨链专题

22.5 FISCO BCOS的部署运维与实战工具

22.5.1 FISCO BCOS零基础入门，五步轻松构建应用

作者：李辉忠 | FISCO BCOS 高级架构师

本文面向零区块链基础入门 FISCO BCOS的开发者，以高纯度、超浓缩的极简方式，分享如何快速基于FISCO BCOS 构建你的第一个DAPP应用。

社区经常有人会问：FISCO BCOS项目有10W+行源代码，10W+字说明文档，几十个子项目，我该如何下手，怎样入门？ 莫慌，五步入门宝典已备好！！！

第一步：构建一条FISCO BCOS的链

安装文档里给了一箩筐事无巨细的介绍，但此文就是要简明扼要地告诉你：**三步就能召唤FISCO BCOS！**

(请先在home目录创建fisco目录，接下来都在这个目录操作)

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

你可以通过执行*ps -ef | grep -v grep | grep fisco-bcos*查看到四个节点已经在运行。

```
501 55515     1   0  3:00下午 ttys000    0:00.54 /Users/wheatli/fisco/nodes/127.0.0.1/node0/../fisco-bcos -c config.ini
501 55517     1   0  3:00下午 ttys000    0:00.56 /Users/wheatli/fisco/nodes/127.0.0.1/node3/../fisco-bcos -c config.ini
501 55518     1   0  3:00下午 ttys000    0:00.52 /Users/wheatli/fisco/nodes/127.0.0.1/node2/../fisco-bcos -c config.ini
501 55520     1   0  3:00下午 ttys000    0:00.53 /Users/wheatli/fisco/nodes/127.0.0.1/node1/../fisco-bcos -c config.ini
```

当然，如果你还是需要阅读详细文档，请参考【安装】：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/installation.html#fisco-bcos

第二步：安装一个交互式控制台

控制台是一个可以交互式访问区块链，进行区块链数据读写请求的工具。无需太多解释，**四步完成控制台安装**：

```
$ curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/
↪download_console.sh && bash download_console.sh -c 1.2.0
$ cp -n console/conf/applicationContext-sample.xml console/conf/applicationContext.
↪xml
$ cp nodes/127.0.0.1/sdk/* console/conf/
$ cd console && ./start.sh
```

此时，你已经进入控制台界面，可以通过help查看命令列表，通过getPeers获取节点连接列表，通过exit或quit命令退出控制台。

同时，控制台内置了一个HelloWorld合约，可以直接调用deploy HelloWorld进行部署，然后调用call HelloWorld进行访问。

第三步：编写一个Solidity合约

教程文档依然是循序渐进地指导，但其实！遵循业务合约编写**三部曲：****存储设计->接口设计->逻辑实现，**就足以顺利完成业务合约。

如果你还是习惯阅读详细文档，请参考【教程】：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/tutorial/sdk_application.html

以文档中的资产转移应用为例，支持链上用户的资产登记、查询、转移功能：

- **存储设计：基于分布式存储，设计存储表结构**
- **接口设计：基于业务需求，设计合约接口**

```
// 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)
```

(continues on next page)

(续上页)

```
// 资产转移
function transfer(string from_asset_account, string to_asset_account, uint256␣
↪amount) public returns(int256)
```

- **逻辑实现：基于CRUD接口，实现业务逻辑**

asset.sol合约：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/tutorial/sdk_application.html

第四步：合约编译与部署

Solidity合约需要通过编译器转换成机器（虚拟机）可执行的二进制，这些二进制是一系列OpCode的组合，虚拟机将解析执行这些OpCode实现合约业务逻辑。

编译之后的合约需要通过工具部署到区块链（写入区块链账本），之后才可以根据合约接口描述文件（ABI）进行调用访问。

额，老毛病犯了，又唠唠叨叨讲原理，还是讲讲**如何无脑一键完成合约编译与部署吧**：

参考说明文档【控制台】的deploy命令：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/console/console.htm

将Assert.sol放置在console/solidity/contract目录，在控制台执行deploy Assert.sol即可完成合约的编译以及部署。

```
=====================================================================================
Welcome to FISCO BCOS console(1.0.2)!
Type 'help' or 'h' for help. Type 'quit' or 'q' or 'exit' to quit console.
 _______ ______  ______   ______   ______       _______  ______  ______  ______
|       |      \/      \ /      \ /      \     |       \ /      \ /      \ /      \
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\
| $$__     | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\$$
| $$  \    | $$  \$$    \| $$     | $$  | $$    | $$    $| $$     | $$  | $$\$$    \
| $$$$$    | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$$\
| $$      _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__| $$
| $$     |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$    $$
 \$$      \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$$

=====================================================================================
[group:1]> deploy Asset.sol
contract address:0x0b9ce0c6c4a85816bb328815d6befd7aa56119e8
```

第五步：开发业务

继续假设采用Java开发业务，当然也假设你熟悉eclipse、gradle、spring等常用工具。

1、建立一个Gradle Java工程asset-client，通过IntelliJ IDEA或者Eclipse；

2、编译build.gradle，增加maven库依赖；

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

dependencies增加：compile ('org.fisco-bcos:web3sdk:2.0.4'){exclude group: 'org.ethereum'}

3、将第二步中控制台配置目录（console/conf/）的相关配置文件（applicationContext.xml，log.properties，ca.crt，node.crt，node.key）拷贝到asset-client项目的main/resource目录；

4、将第四步中一键编译生成的java文件（console/solidity/java/*)拷贝到asset-client项目的main/java目录；

5、在main/java目录新建AssetClient类，Asset.java已经实现了有deploy，load，select，register，transfer接口，通过调用这些接口实现业务相关逻辑。

具体代码可参考示例工程：https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/asset-app.tar.gz

到这里，你已经完成第一个基于FISCO BCOS的应用开发！如你对开发流程有疑问或优化建议，可以通过公众号进入技术交流群和我们一起探讨。

22.5.2 WeBASE

FISCO BCOS迎来区块链中间件平台WeBASE，应用落地提速

随着区块链技术的发展，越来越多的开发者基于稳定和高效的区块链底层平台，结合智能合约和链上接口，开发各种丰富的应用，随之而来的是对区块链系统的易用性、应用开发速度、业务组件丰富性产生了更多的需求。

直面底层平台，“裸写”智能合约和底层代码，在技术上是可行的，而进一步提供“所见即所得”、“开箱即用”这种无门槛体验的动力，则来自社区开发者的反馈，比如：

- 缺乏好用的智能合约开发工具，合约的开发和调试效率不高，难以便捷地管理链上各节点配置信息，观察其运行状态。
- 区块链上的区块、交易、回执等数据的呈现方式不够友好，对链上的海量数据难以进行灵活和多维度的分析。
- 针对参与到业务的各账号及其进行的交易行为，需要通用的审计工具，以便及时发现和杜绝异常。

为响应开源社区里持续的需求，同时将自身长期探索的成果开放分享， FISCO BCOS开源工作组成员单位——微众银行，为社区贡献一条从区块链底层通往应用落地的高速通道。

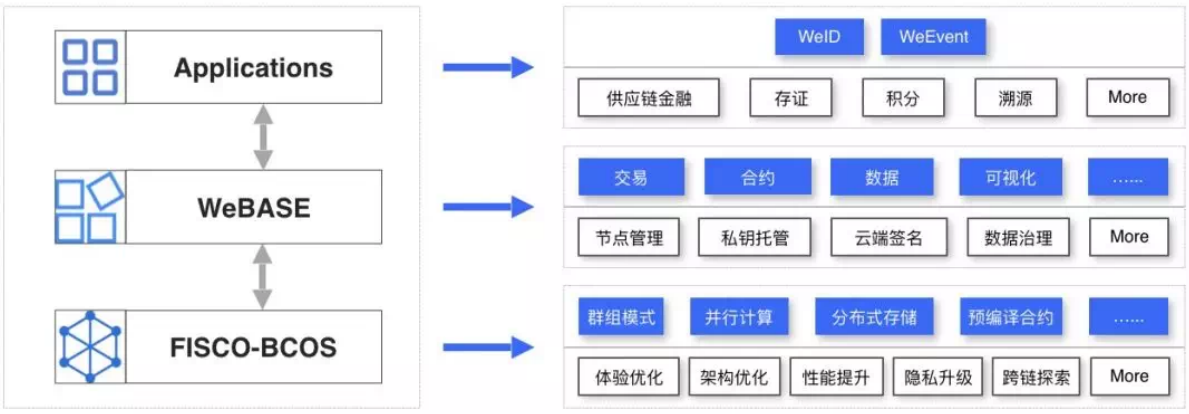
7月2日，微众银行正式开源自研的区块链中间件平台——WeBASE，该平台适配支持FISCO BCOS底层平台，面向多种对象，如开发者、运营者，并根据不同的场景，包括开发、调试、部署、审计等，打造丰富的功能组件和实用工具，提供友好的、可视化的可操作环境。

基于FISCO BCOS底层平台，部署WeBASE，可以简化区块链应用开发流程，大大降低企业搭建区块链应用，以及进行运营分析的时间成本、人力成本，使开发者可得心应手的操控区块链网络，更专注投入到应用开发和业务落地中。

WeBASE代码仓库地址：https://github.com/WeBankFinTech/WeBASE

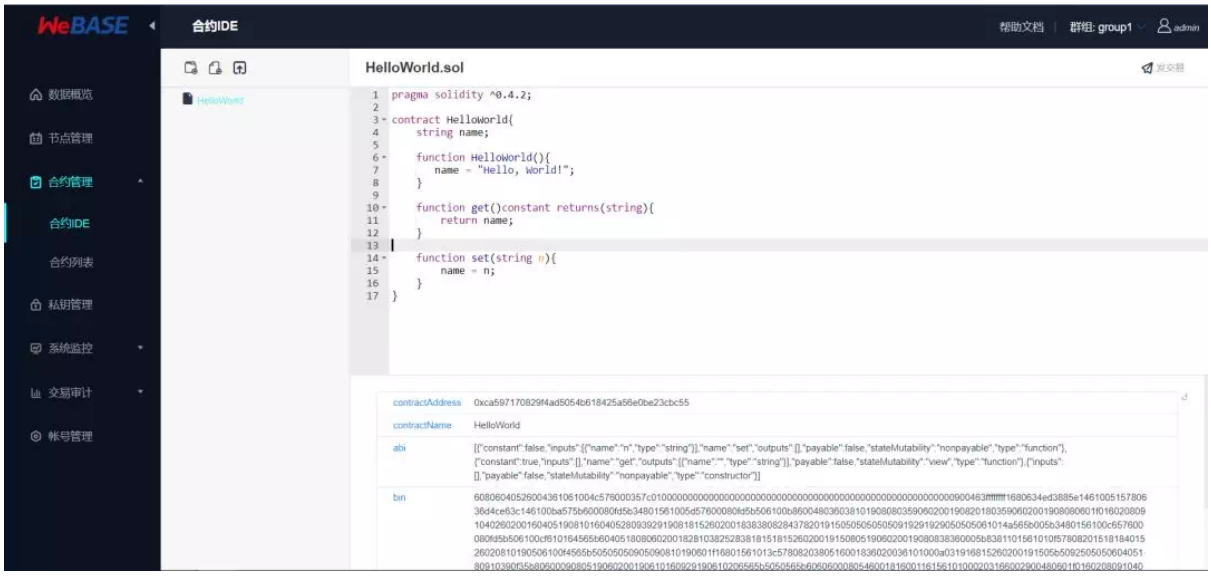
WeBASE简介

WeBASE（WeBank Blockchain Application Software Extension） 是在区块链应用和FISCO BCOS节点之间搭建的中间件平台，如下图所示，开发者可在区块链节点上层部署WeBASE的可交互模块如浏览器、管理台和其他工具等，也可以基于WeBASE内置的组件和API，进行应用的开发。

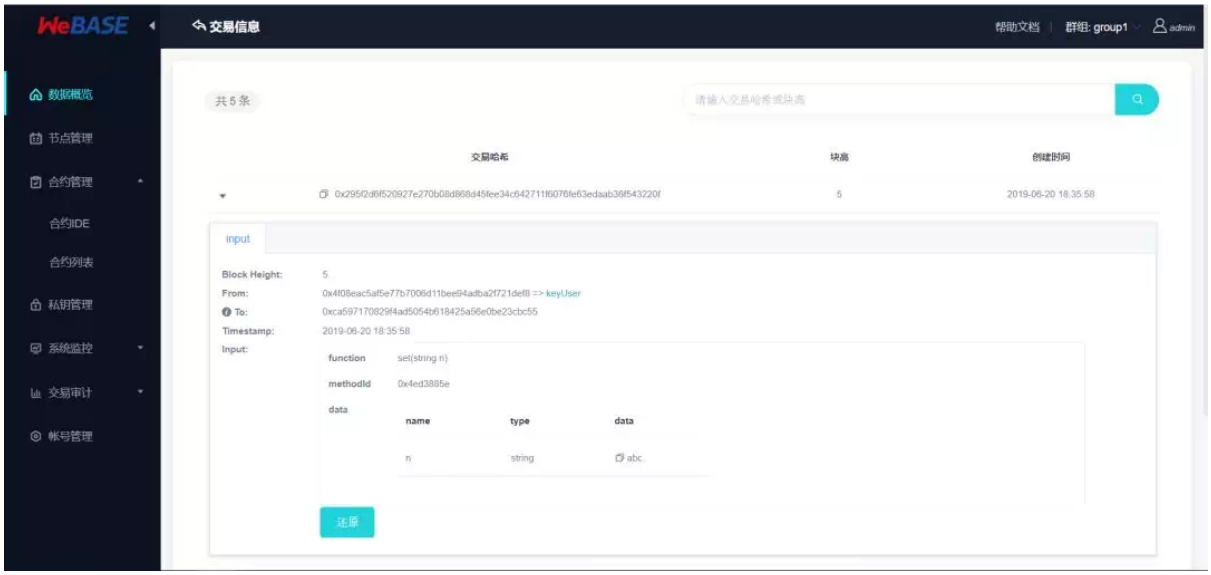


WeBASE的“**简约而不简单**”，体现在以下几个方面：

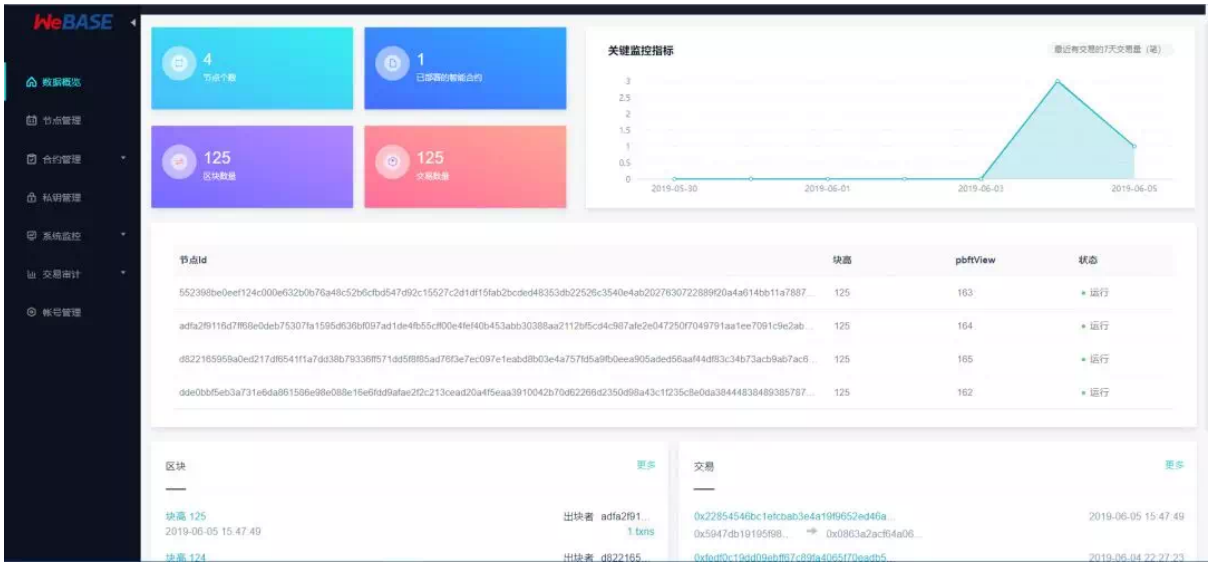
一、提供友好的智能合约开发平台，支持在线编译、调试、测试、部署智能合约，具有高效的编辑环境，率先支持Solidity智能合约语言。



二、在SDK基础上封装Restful风格的API接口，Restful接口更直观，有良好的可扩展性，可轻松适配多种编程语言。通过接口可对交易数据进行编解码，可在包括网页、手机终端等设备上全方位、多维度展示链上数据细节。



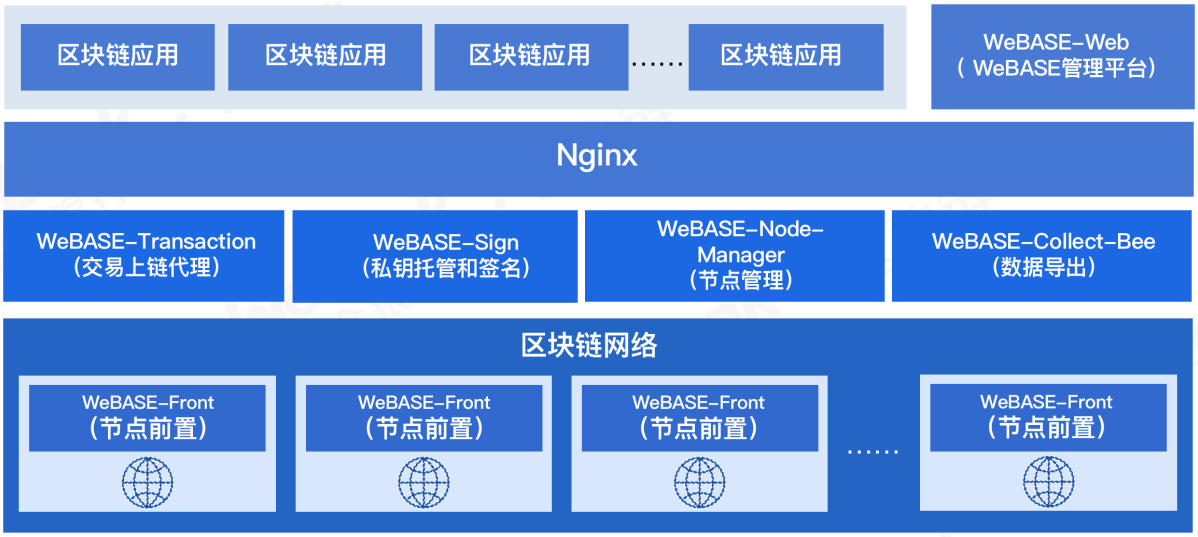
三、区块链管理平台是运营管理员的首选工作台，可查看链上的数据统计、每个区块的详情、各个节点多维度的统计数据，对节点的健康度做到全方位监控。



四、数据导出组件可配置式的导出链上数据到关系型数据库、大数据处理等系统，以便对链上数据进行多元化处理，如进行数据挖掘，构建商业模型等。

WeBASE的整体架构和设计原则

WeBASE的完整架构如下图所示：



WeBASE的设计理念是一个子系统解决一个问题，无需部署所有子系统就能跑起来，因此在设计之初就遵循如下原则：

按需部署：WeBASE抽象对应用开发通用共性进行抽象，形成各类服务组件，如业务接入、私钥管理、交易队列、合约开发、数据导出、审计等，开发者根据需要部署所需组件。

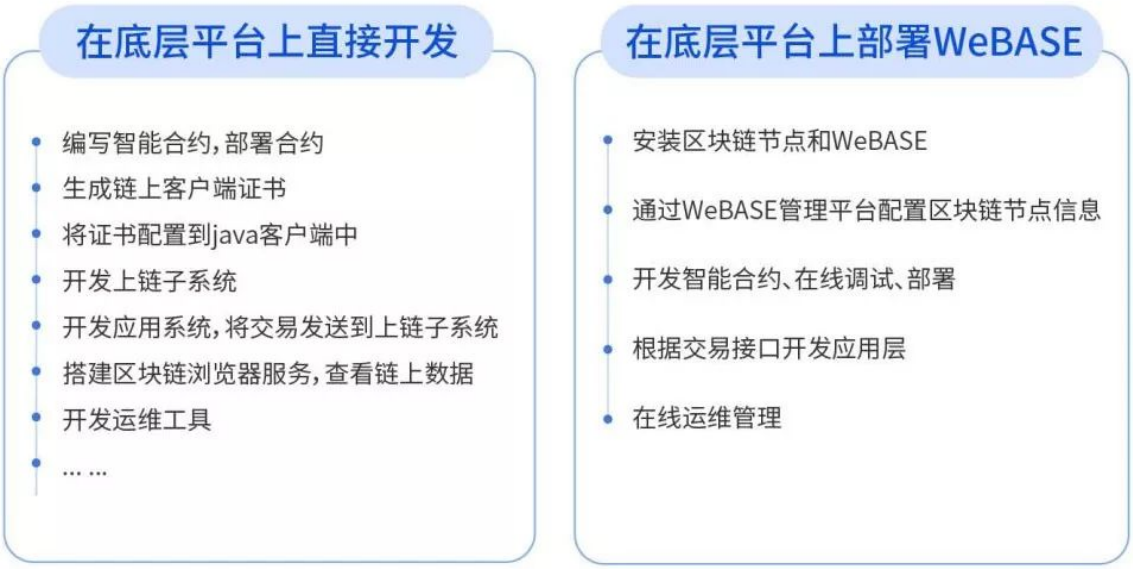
微服务架构：WeBASE采用微服务架构，基于spring-boot框架，提供Restful风格接口。

零耦合：WeBASE所有子系统独立存在，均可独立部署，独立运行，面向不同的场景提供服务，避免出现“全家桶”式的冗余负担。

可定制：前端体验往往带有开发者自身的业务表现，如不同的样式、不同的交互风格、不同的品牌表现等，因此WeBASE采用前后端分离的设计，后端接口保持稳定和可扩展，前端页面则由开发者自由定制。

基于WeBASE的应用开发流程

基于WeBASE的应用开发流程有全新的体验，下图直观比对两种开发流程的不同：



显然，基于WeBASE的应用开发，流程大大简化。智能合约开发工具、完善的数据可视化平台、简易的交易上链方式，降低了开发门槛，使得开发效率大幅提升。而且对于应用上线后的交易审计、数据导出、立体监控等方面的管理，WeBASE为其提供一系列完善的组件，能有效避免开发者和企业重复破荒造路。

WeBASE的下一步

今天，**WeBASE开源只是一小步，未来，WeBASE将有计划地开放更多功能**：

- 提供更多直面商业领域的、经过业务验证的组件，便于集成到有共通模式的应用中，建立区块链应用开发最佳实践和标准架构；
- 提供各类行业解决方案和参考实现；
- 为云厂商提供更友好的接入方式。

社区开发者体验

- **网络技术总监 林冬艺** 广州市品高软件股份有限公司

  品高软件是一家云计算公司，正在探索云计算与区块链的融合创新。WeBASE具有友好的界面设计、便捷的管理以及云生态的良好融合等特点，使得区块链上云变得更轻松，我们得以更专注于区块链与云计算的业务场景。

- **架构师 魏巍** 宝付网络科技（上海）有限公司

  WeBASE操作简便，社区文档非常完善，涵盖企业所需的各类组件，可插拔式的架构设计，根据所需组合使用，帮助我们解决区块链技术的后顾之忧，缩短业务数据化进程，快速实现业务价值。

- **CTO 金兆康** 杭州亦笔科技有限公司

  WeBASE的联盟治理、前置管理功能使得管理区块链就像管理聊天群一样简单。社区的及时响应、功能的快速迭代，处处体现出国内顶级区块链服务提供商的实力。

- **区块链架构师 孙耀普** 全链通有限公司

WeBASE各系统模块都提供了丰富的部署文档，明确的部署步骤，按照部署文档就可以很快部署和使用，系统的UI设计和布局也非常合理。

注册和登录功能，满足了客户关于区块链信息查询权限管理需求，减少了开发的工作量；系统集成了FISCO BCOS的java接口，提供了丰富的调用接口，减轻了开发工作量，降低了区块链的使用难度。

系统提供智能合约的编辑和编译和部署功能，使智能合约开发更加便捷。

- **创始人 张泷** 聚滴成海(广州)信息技术有限公司

  聚滴成海是一家专注区块链深层次挖掘的中间件提供服务商，我们需要直接对客户提供封装好的接口，来降低他们的开发工作量。由于我们产品性质的特殊，需要兼容多种生态、包容多种共识机制，因而我们的开发工作量非常庞大！

  但是WeBASE将各种接口已经封装的非常完善，只需稍作暴露就可以快速连接！希望WeBASE可以继续坚持对开源社区的支持，我们也愿意为之不断付出和贡献。

漫谈WeBASE之交易两三事

作者：刘明臻 | FISCO BCOS 核心开发者

2019年7月2日，区块链中间件平台WeBASE正式对外开源，大家首先会想到的就是：WeBASE是什么，有什么用？WeBASE是WeBank Blockchain Application Software Extension的简写，是在区块链应用和FISCO BCOS节点之间搭建的一套通用组件。开发这套通用组件的目的是为了屏蔽区块链底层的复杂度，降低开发者的门槛，提高区块链应用的开发效率。WeBASE主要包含：节点前置、节点管理、交易链路，数据导出，Web管理平台等子系统。其完整部署架构如下图：

详细了解WeBASE请点击前往：《FISCO BCOS迎来区块链中间件平台WeBASE，应用落地提速》。我们计划推送WeBASE系列文章，与你一道体验WeBASE的“简约而不简单”。本文是系列文章第一篇《漫谈WeBASE之交易两三事》，打算谈谈WeBASE在交易相关方面做的一些工作。

交易的概念、基本数据结构及过程

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,以求达到维护合约的生命周期，管理资产，进行价值交换等目标。交易的基本数据结构包括发送者，接受者，交易数据等。

一个完整的交易过程可分为交易构建并**链上执行**、**交易展示**、**交易审计**三步：

- 首先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）；然后链收到交易并交由多个节点机制处理，执行相关的智能合约代码，生成交易指定的状态数据；最后将交易打包到区块里，和状态数据一起落盘存储。落盘交易即为被确认，被确认的交易通常被认为具备了事务性和一致性。
- 随着交易确认，相应的还会有交易回执（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中生成的信息，如：结果码、日志、消耗的gas量等。用户可以使用交易hash查询交易，交易回执展示给用户。
- 随时间推移，链上交易越来越多，需要对链上交易进行分析，辅助监管和审计链上的行为，确保链的合理、合规运行。

大家对交易有了初步的了解，接下来我们就从交易的三步曲中，详细探讨WeBASE在其中充当的角色和发挥的作用。

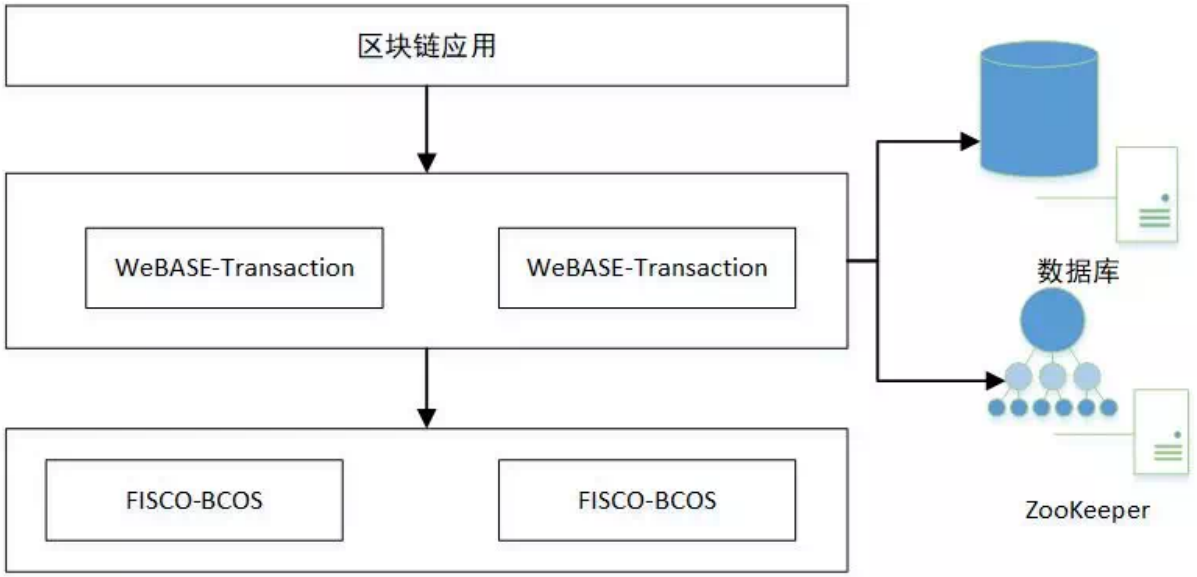
交易上链

通过WeBASE有多种发送交易上链的方式,比较常见的有WeBASE管理平台和交易上链代理子系统等两种，前者由WeBAS管理平台提供一个合约IDE，可以部署和调用合约接口，这里就不赘述了。今天我们重点介绍下**交易上链代理子系统（WeBASE-Transaction）**。

目前，区块链开发者主要面临以下几个痛点：

- 区块链学习成本高–不想关注区块链细节，想用传统的方式调用区块链服务。
- 现实业务的峰值可能超过区块链的处理能力–需要一套缓存系统来削峰平谷。

为了解决这些痛点，WeBASE-Transaction应运而生。WeBASE-Transaction是从很多区块链存证类型项目中总结出来的一套服务系统，以帮助大家快速搭建区块链应用，其功能主要是接收无状态交易请求，缓存到数据库中，再异步上链，服务支持分布式任务，多活部署，异地容灾。部署架构如下图：



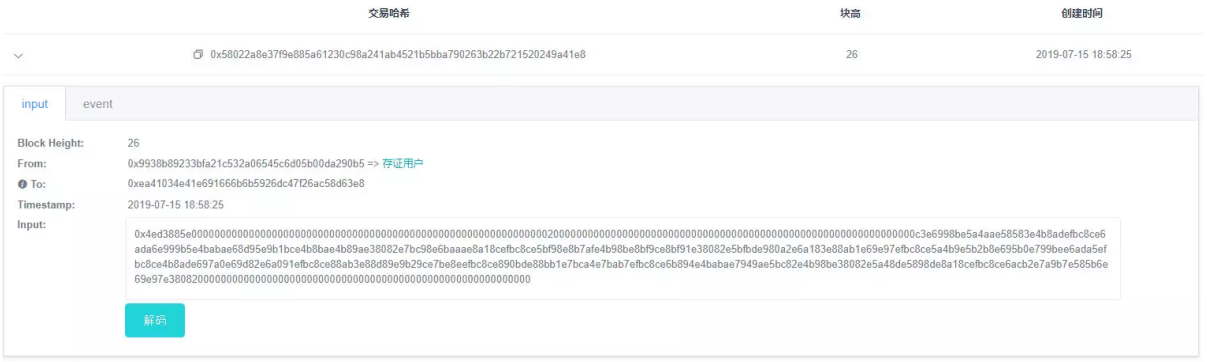
从部署结构图可以看到，WeBASE-Transaction具有以下几大特性：

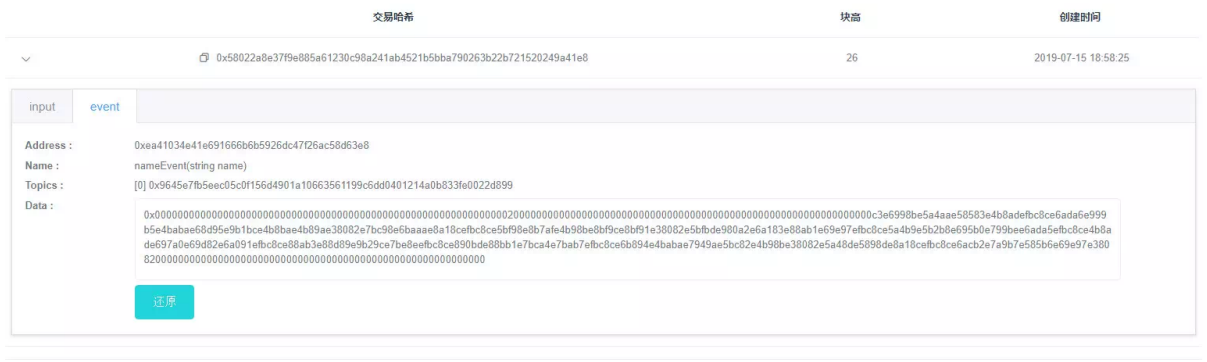
- **异步上链，过载保护**：区块链请求先缓存数据库，削峰平谷，由服务使用合理速度异步上链。
- **多活部署，分布式任务**：使用Zookeeper做分布式任务协调，将上链分为多个分布式任务，多活部署，异地容灾。
- **错误重试，实时对齐**：服务自动查验上链状态，错误重试，达到数据库和链上状态的实时对齐。

交易展示

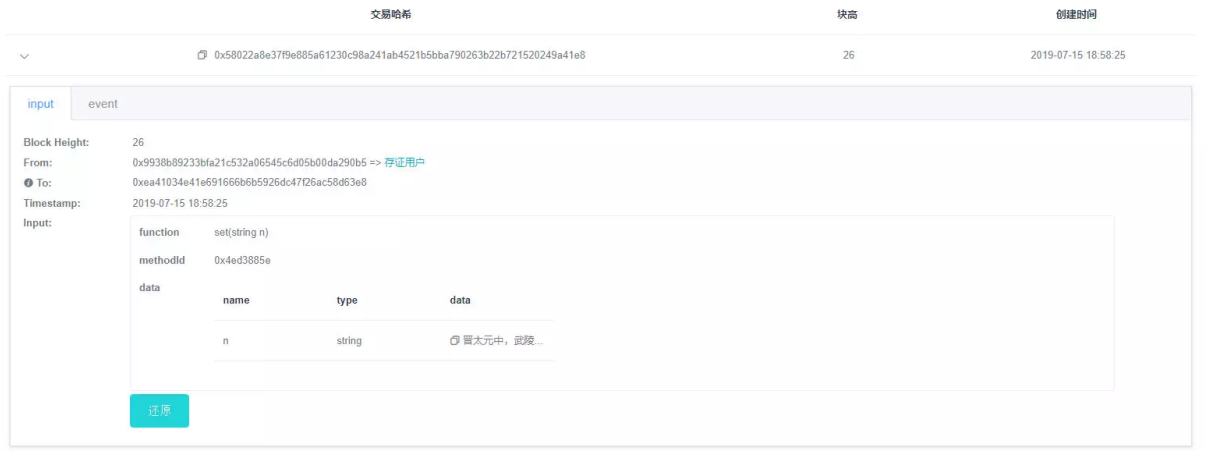
交易上链代表着终于把数据写入区块链了。兴冲冲地跑去给老板汇报：我们数据上链啦！老板听了既高兴，也很好奇。区块链这么高深，写入区块链的数据如何展示出来？区块链交易长啥样？

此时你可以搭建一套WeBASE管理平台。搭建方法可以查看《安装部署文档》。使用方法可以查看《WeBASE管理平台使用手册》。搭建完成后，打开WeBASE管理平台，你会发现交易是这样的：发送者、接受者、交易输入数据等。交易执行后，会产生交易回执。交易回执里面有event，记录了交易执行过程中的事件日志。





老板看后第一感觉，肯定是五味杂陈。此时你轻点“解码”按钮，把交易从一串“0x的天书”翻译成人类能读懂的模样。哪个用户通过哪个合约方法上链的，一目了然。老板看后必投来赞许的目光。此时你想必会有一种普天之下，舍我其谁的感觉。

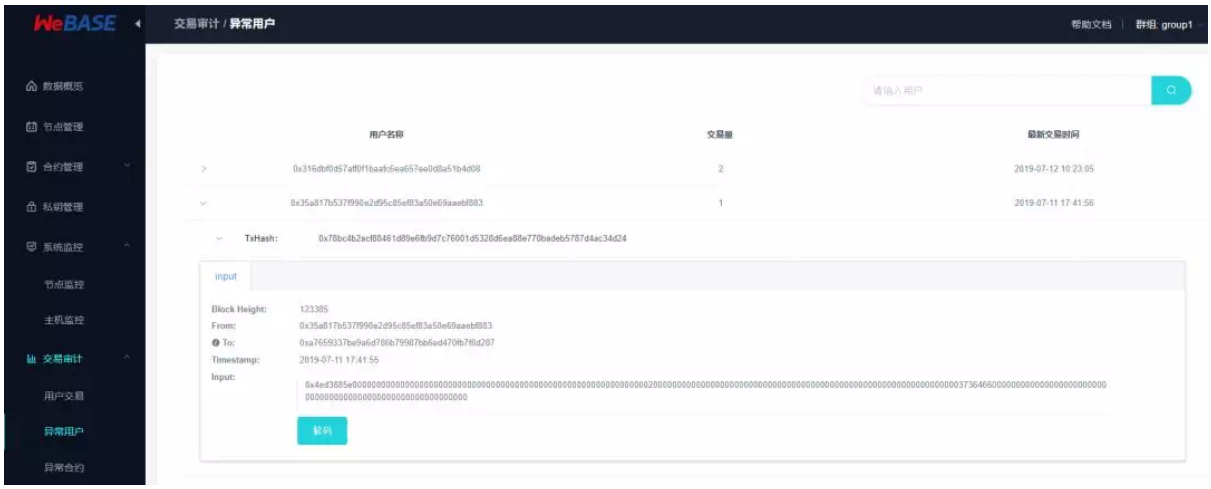


交易审计

交易上链了，也展示了，但这还不够。联盟链中各个机构，按照联盟链委员会制定的规章，在链上共享和流转数据。这些规章往往是字面的，大家是否遵守，缺乏监管和审计。因此为了规范大家的使用方式，避免链的计算资源和存储资源被某些机构滥用，急需一套服务来辅助监管和审计链上的行为。WeBASE管理平台提供的交易审计，就是一套辅助监管和审计链上行为的服务。它结合区块链数据、WeBASE管理平台的私钥管理和合约管理三者的数据，以区块链数据为原材料，以私钥管理和合约管理为依据，做成的一个综合性的数据分析功能。

交易审计主要指标：

交易审计界面如下



WeBASE管理平台提供可视化的去中心化合约部署、交易监控、审计功能，方便识别异常的用户、异常的合约，还有异常的交易量，为联盟链治理提供依据。

回归初心

到这里，我们可以看到一条交易，从组装发到区块链节点，然后节点执行落盘成交易回执，交易和state，再到交易展示，交易事后审计和监管。WeBASE参与了其中的组装上链、交易展示、交易审计，基本上可以说WeBASE参与了交易的全过程。

在组装上链过程中提供了交易上链代理子系统，其有效屏蔽了区块链底层的复杂度，降低了开发者的门槛，助力开发者高速搭建联盟链应用。在交易展示过程中，WeBASE搭建了可视化的管理平台，对交易进行了立体化展示，方便开发者实时监控和查看交易数据。在交易事后审计和监管过程中，还提供了综合的数据分析功能，辅助监管和审计链上的行为，高效治理联盟链。

链接指引

- 区块链底层平台FISCO BCOS代码仓库
- FISCO BCOS技术文档
- WeBASE代码仓库
- WeBASE技术文档

在WeBASE的动物园漫步

作者：毛嘉宇 | FISCO BCOS 核心开发者

作者语：开源软件，是大师心目中商业化大潮之外的伊甸园；是极客们聚集出没的大教堂和集市；是技术爱好者挚爱的隽永诗篇；但在我眼里，开源软件的世界就是一座动物园。

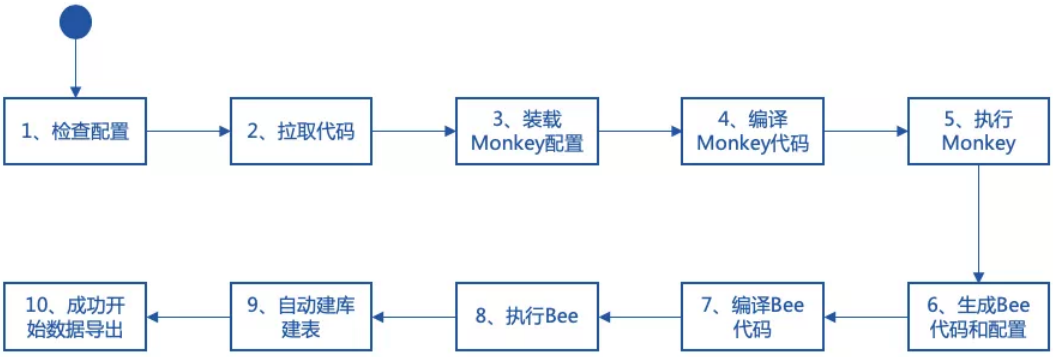
开源机构酷爱拿动物来命名软件或制作LOGO，这些名字早已似春雨般深入人心。比如一脸严肃，身手敏捷，酷似狮子的Tomcat。还有Python，两条肥胖蠢萌的蟒蛇形象火遍了全世界。而Linux用一个叫Tux的企鹅作为吉祥物，Tux 现在已经开始拍游戏视频、广告了，甚至还有个叫Gown的女朋友。开源社区动物园里的风景异彩纷呈，受此启发，我们将WeBASE中的两个数据导出相关组件分别取名为Monkey和Bee。

本文就为您介绍神通广大的Monkey和专注如一的Bee。**WeBASE是在区块链应用和FISCO BCOS节点之间搭建的中间件平台**，对技术和业务架构中的共性进行抽象，形成通用化且体验友好的组件，简化区块链应用开发流程。

WeBASE-Codegen-Monkey之美猴王

WeBASE-Codegen-Monkey（以下简称Monkey）是WeBASE数据导出工具的代码生成组件。可帮助用户自动生成数据导出组件，更进一步地提升研发效率，帮助开发者减负。我们开发的Monkey，其命名灵感来自于美猴王——齐天大圣孙悟空。

Monkey提供可执行的shell脚本：generate_bee.sh。只需简单的配置，同时按要求提供智能合约文件，便可自动生成WeBASE-Collect-Bee。在执行完脚本后，Monkey会自动退出，仿佛化生变幻的美猴王，腾云而来，驾雾而去。Monkey的启动脚本，会自动下载代码、装载合约、构建生成业务代码、编译数据导出组件代码、启动数据导出的应用。generate_bee.sh的主要执行步骤如下图：



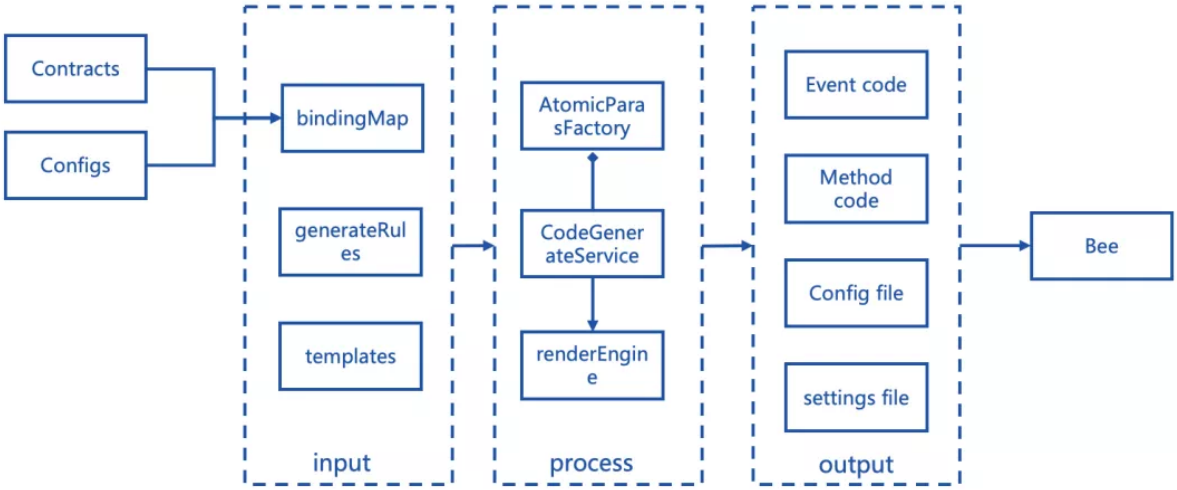
其中，步骤3，脚本会自动将开发者的配置文件，放置到指定的配置路径。步骤5，Monkey在启动后，会根据次序，生成日志、函数代码、配置和设置文件，并放置到Bee所需的路径上。步骤8，Bee在启动时，可随心所欲地加载所需的代码和配置文件。开发者也可以在预先约定的路径下，获得所需的各类脚本或参数的文件。

generate_bee.sh的执行过程，就如同《西游记》所述：『见他凶猛（数据导出定制化开发需求复杂），即使身外身法（读取环境和合约的配置），拔一把毫毛（下载代码、装配配置和编译Monkey代码），丢在口中嚼碎（运行monkey代码，自动生成Bee的配置和代码），望空喷去（编译Bee的代码），叫一

声'变！'（开始执行Bee的代码）即变做三二百个小猴，周围攒簇（自动建库建表并成功开始导出数据）。』

美猴王有七十二变的神通，Monkey有自动生成所需代码的功能，真正做到让数据导出系统开箱可用Monkey生成的代码等文件，可分为四大类：

1. 基于合约和配置生成的具体日志解析的代码；
2. 基于合约和配置生成的具体函数解析的代码；
3. 基于配置生成的配置文件；
4. 基于合约和配置生成的设置文件。



上图中，不同类型的Paras会生成不同的代码文件。其中：

- EventGenerateParas：包含了日志（Event）相关代码和脚本的生成。

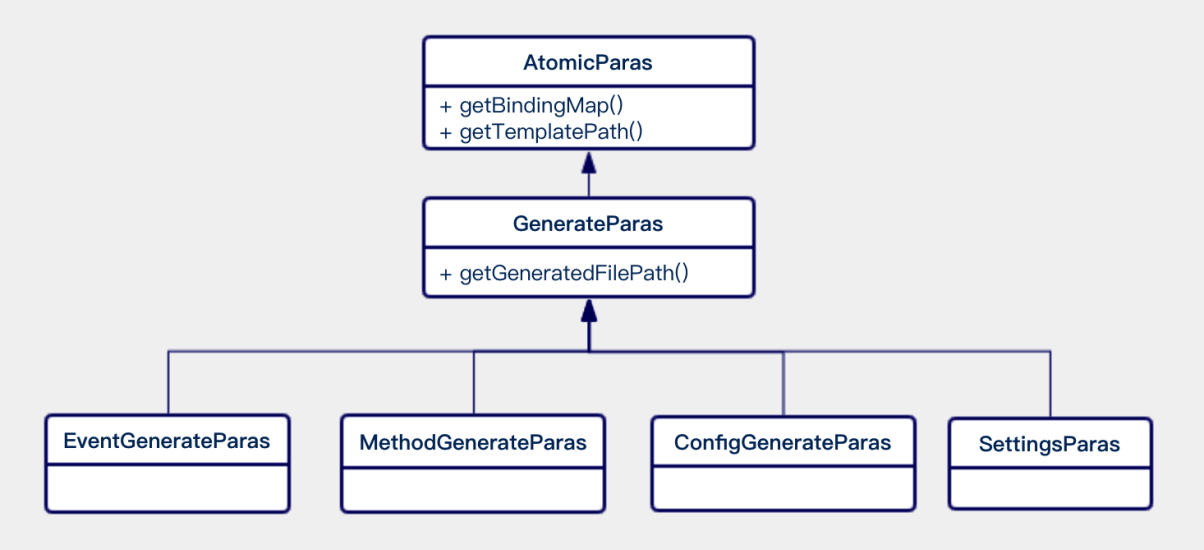
  如合约文件中解析的日志entity和repository的定义、以及每个不同日志解析的代码、日志的BO类和日志的数据库建表语句。

- MethodGenerateParas：包含了函数（Method）相关代码和脚本的生成。

  如合约文件中解析的函数hibernate entity和repository的定义、以及每个不同函数解析的代码、函数的BO类和函数的数据库建表语句。

- ConfigGenerateParas: 包含了Bee工程的数据库建表脚本和数据库配置文件。
- SettingsParas：包含了grafana dashboard和table_panel的json导入设置文件。

如上所述，为了开发者方便易用，一次配置。Monkey除了生成执行代码外，相关的区块链软件和数据导出组件的通用配置，也被放到了Monkey系统的配置中，并自动传递到Bee的配置文件中。Monkey系统集成了Beetl作为模板引擎，无论多么复杂的合约，多么繁琐的配置，都能化繁为简，快速生成。Monkey通过载入开发者编译后的solidy对应的Java文件，利用反射技术，获取合约文件的Class信息。



Monkey启动后，会创建一个代码生成的模板引擎，上述的AtomicParas则是引擎启动所必须的原料。如上图，通过这三个核心的调用步骤:

1. 引擎通过调用getTemplateFilePath()方法，获得渲染模板的文件路径;
2. 通过getBindingMap()方法获得渲染所需的所有参数;
3. 通过getGeneratedFilePath()获得生成文件的具体路径。

代码生成引擎便可飞速启动，执行渲染，最终完成所需文件的自动生成。

当然，美猴王的传奇不会终结，Monkey系统提供了很多强大而灵活的配置，来满足拥有独特个性化需求的开发者。美猴王的征途也不会仅仅局限于数据导出，未来还将会涉足更多的领域，为大家带来更多的惊喜……

WeBASE-Collect-Bee之蜜蜂

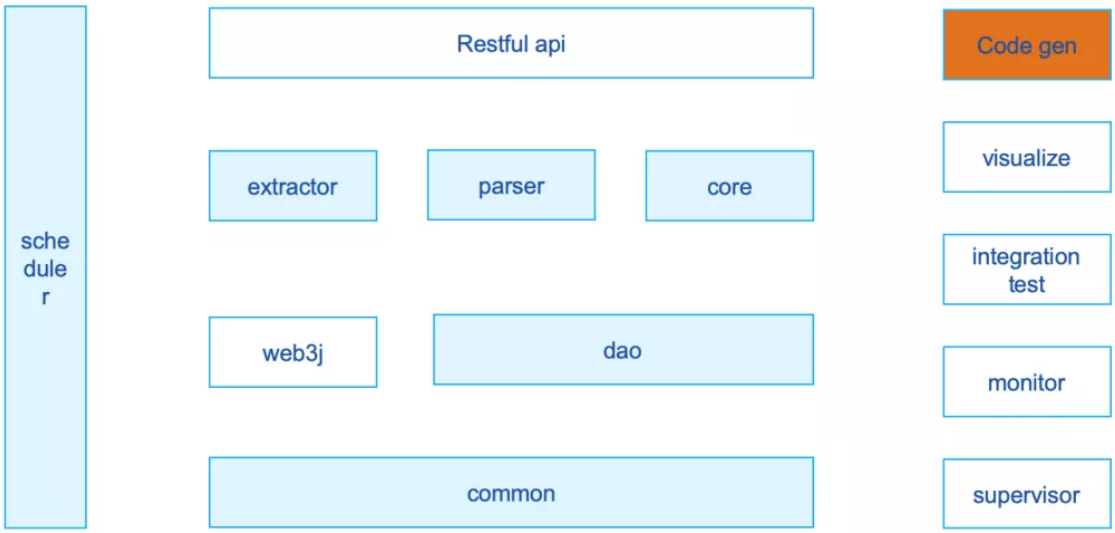
WeBASE-Collect-Bee（以下简称Bee） 是WeBASE中一个基于FISCO BCOS平台的数据导出组件，支持将区块链的数据导出到Mysql等数据库中。

为何取名『Bee』呢? 古往今来，文人墨客留下了诸多歌咏蜜蜂的诗文：穿花度柳飞如箭， 粘絮寻香似落星。小小微躯能负重， 器器薄翅会乘风。——吴承恩

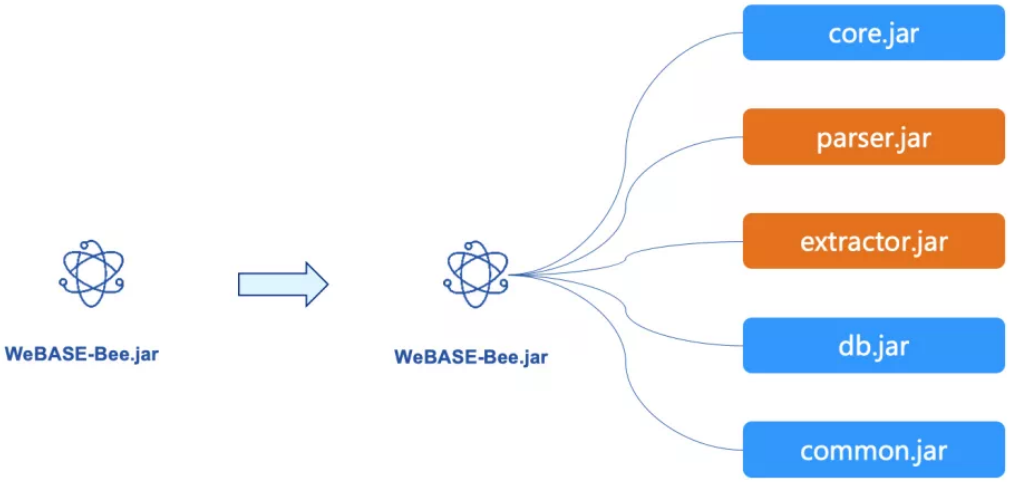
Bee系统恰如蜂群一般，专注如一，辛勤劳作。服务不停，数据导出的任务就不会停止。小蜜蜂们在区块链的花丛之中，来回穿行，在区块之间，寻找香甜、漂浮的业务数据，将海量的区块链数据，稳定高效地导出到存储之中，供开发者执行分析、计算和查询。小小微躯的蜜蜂是轻盈的，契合了Bee系统设计『轻』的气质。

在Monkey的协助下，遵循『契约优于配置』的原则，开发者只需修改极少的配置，辅以搭链后自带的证书和开发编译后的合约文件，执行脚本后就可以快速获得已打包的可执行Jar包，甚至可以直接启动运行。蜜蜂的轻盈还体现在，系统地提取和抽象了大量的配置项。按照开发者的个性化需求，灵活配置，按需而动。

系统架构图如下所示，除了核心的模块，Bee系统的模块都是可插拔的。例如是否需要引入可视化数据分析、集成测试界面、监控和Supervisor进程管理等增强功能模块。



为了让蜜蜂器器薄翅能够舞动地更加轻盈，我们在最新的V1.1.0中进行重构，将原有的单工程按照功能粒度进一步拆分。这样，开发者既可以选择直接部署一个可执行的Jar包。还可以按需引入Jar包，将指定功能模块嵌入到自己的工程中。上述的模块均基于Springboot2开发，支持个性化配置。



如上图所示，各个Jar的功能正如其名：

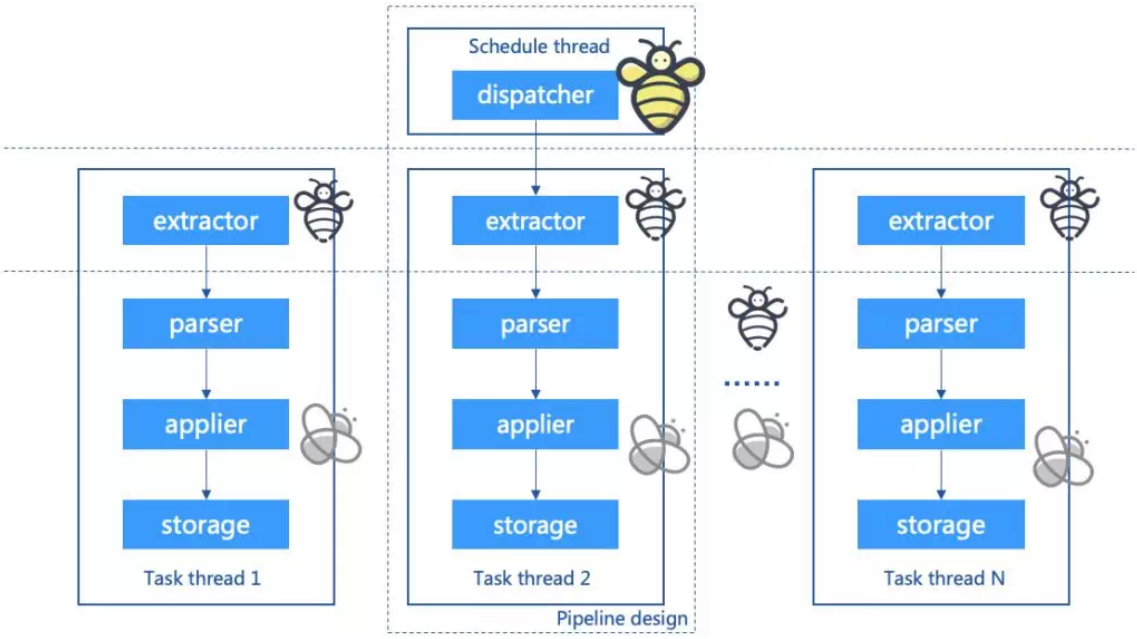
- core：打包为一个可执行的BootJar包，内置开箱即用的Restful API，使得开发者更加便捷地获得数据导出的服务。同时，core自身也是一个示例，向开发者展示了，如何组合利用各个功能模块的Jar包，来开发一个功能完整的数据导出系统。
- parser：封装了区块解析的功能。入口的方法签名可以传入一个区块Block，获得完整解析后的BlockInfoBO的Bean对象，其中包含了解析后的数据。
- extractor：进一步封装了web3j的SDK的逻辑，可以便利地调用上链功能和获取区块链的数据。
- db：封装了数据存储的逻辑，理论上支持各类主流的数据库，实际严格测试支持将数据存储到Mysql 5.6及以上社区版。
- common：封装了一些公用的数据结构、工具类和公共参数等。

蜜蜂的微躯虽然轻盈，可是蜂群的协作却可以负重。

为了应对海量数据的存储，例如单表数据超过1KW条，单库容量超过1T的场景，Bee 集成了sharding-jdbc，支持可配置的多数据源存储、读写分离和分库分表。

为了加快数据导出的速度，Bee系统进行了多轮的性能优化，大幅提升了单机部署下数据导出的效率。同时，依托集成Elastic-Job后，获得分布式任务协调服务的能力，Bee支持实例多活部署，横向扩展，灵活伸缩。

蜜蜂的工作是三型分工，Bee系统的任务执行者也分为三类：任务的调度者、区块的采集者和区块的处理者。



如上图所示，具体的三类任务执行者的说明如下：

蜂王（Dispatcher）：

通过分布式协调服务，保证只有唯一的一个线程运行，负责检测区块链当前的块高和已拉取任务的明细，准备任务，检测区块的分叉状态，检测任务的超时状态和错误状态，并基于预先定义的策略执行重拉等。准备任务是指Dispatcher会基于区块的维度在数据库进行维护：为每个区块创建一条状态记录，并插入到block_task_pool的系统表中。

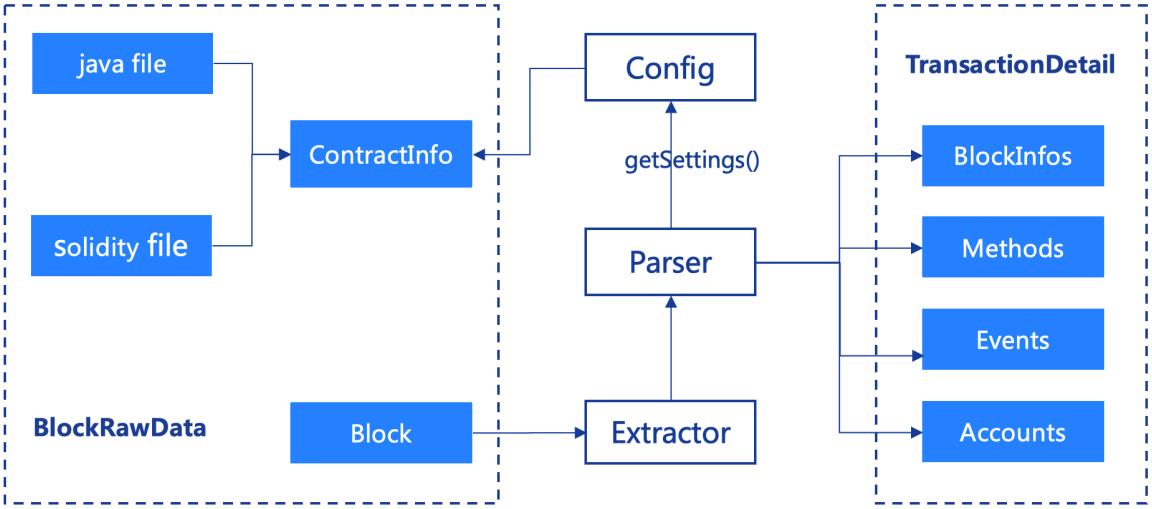
雄蜂（Extractor）：

与Depot共用相同的线程，线程的总数可以通过配置文件来指定，但任务执行的分片由分布式协调服务自动调度。Extractor执行的任务职责是：从Dispatcher维护的block_task_pool系统表中拉取对应分片序号的区块任务，然后从区块链下载指定区块高度的Block数据，并修改block_task_pool系统表的任务执行状态。

工蜂（Depot）：

与Extractor共用相同的线程，线程的总数可通过配置文件来指定，但任务执行的分片由分布式协调服务自动调度。Depot可以直接从Extractor获取到Block内容，然后执行数据导出的任务，先后执行区块解析、内容转换处理、存储数据库，最后修改block_task_pool系统表的任务执行状态。

这样设计的优点是隔离了不同区块在不同线程间切换的数据交换开销，大大提升了处理的效率。同时，可以通过增加部署实例、增加处理线程集合的数量，来提升数据导出的速度。如果把内容转换和内容存储比作花蜜的酿造和存储，那么区块解析就似花粉的采集——这项使工蜂日夜辛劳的工作任务，最能诠释『穿花度柳飞如箭， 粘絮寻香似落星』。



如上图所示，区块解析的结果共分为四类：区块基本信息、账户信息、函数信息、日志信息。以下为解析的简要思路：

1、合约加载：解析器在执行前，会加载合约的BIN、ABI和配置信息，并计算所有合约函数的MethodId。

2、账户解析：根据Block可以获取区块的合约地址，根据合约地址获取runcode，再通过预先加载的合约BIN信息，识别账户的类型，最终解析出账户。

3、函数解析：如果函数为构造函数，如1所述，可以解析出函数的值。

如果函数非构造函数，则transaction中的to字段即为合约地址。如2所述，根据合约地址可以获得函数所属的合约。再根据transaction中附带的input和预先加载的methodId进行比对，可获得函数名。

在精准定位了函数所属合约和函数名以后，会自动触发transactionHandler，并执行对应的解析工作。

4、事件解析:基于步骤2已解析的transactionHash与合约名的映射，可以获取合约名称。根据特定的合约名称，会自动触发该合约下的eventHandler，并执行对应的解析工作。

5、区块解析:根据获得的Block对象的属性解析，获得区块的概要信息。

所有的这些解析步骤，相同的区块都由一个工蜂来负责完成，每个工蜂为一个线程。这些线程通过线程集合，由分布式协调服务来统一分配和管理，从而达到『飞如箭』和『似落星』的效果。Bee系统严格遵循了『技能专一』的原则，致力于为FISCO BCOS提供数据导出服务，因为专一，所以极致。

总结

WeBASE的社区还在蓬勃发展中，欢迎来到WeBASE的动物园漫步，和我们一起发现和探索更多有趣的故事，期待未来动物园的故事里有你。

链接指引

- 区块链底层平台FISCO BCOS代码仓库
- FISCO BCOS技术文档
- WeBASE代码仓库
- WeBASE技术文档

WeBASE数据导出：助力区块链大数据场景快速落地

作者：张龙 | FISCO BCOS 核心开发者

随着区块链技术的快速发展，各类应用如雨后春笋般涌现，链上数据呈指数级增长，基于区块链的大数据场景成为兵家必争之地，数据分析也成为刚需。

然而由于链上数据的存储特点，只能通过接口方式从链上直接获取，这种方式需要针对智能合约接口编写大量的代码，成本很高；同时从链上读取数据，除过网络开销，还需要进行解码操作，甚至遍历MPT树等，读取性能差；更重要的是，不能直接在链上进行全局数据处理，从而无法满足大数据场景的需求，比如复杂查询、大数据挖掘和分析等。

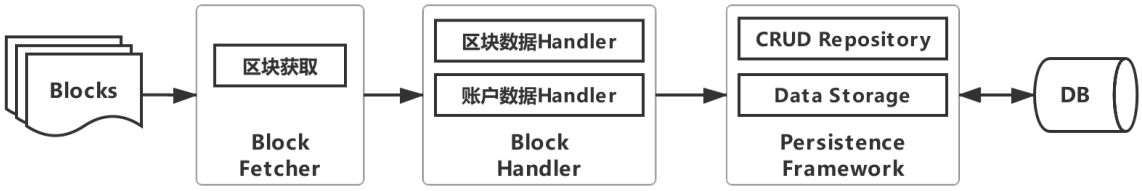
针对用户的需求，我们致力于提供一种自动化、智能化的数据导出解决方案，并通过不断迭代完善，满足用户基于数据导出的各种诉求，助力区块链大数据场景快速落地。本文将从用户需求出发，逐层揭开**WeBASE数据导出**组件功能、特性和架构演进的神秘面纱。

WeBASE是在区块链应用和FISCO BCOS节点之间搭建的中间件平台，对技术和业务架构中的共性进行抽象，形成通用化且体验友好的组件，简化区块链应用开发流程。

用户说：链上获取数据性能差，不便于大数据处理，便有了WeBASE-Collect-Bee

对于用户而言，希望有原始数据支持大数据处理，但由于区块链特有的链式存储结构及出于安全性而设计的编解码操作及状态树等，使得从链上读取数据性能很差，于是我们设计了WeBASE-Collect-Bee 。

WeBASE-Collect-Bee初步架构如下图所示，其目的在于降低获取区块数据的开发门槛，提升数据获取效率，支持大数据处理，用户只需要进行简单配置，就可以把区块数据导出到指定的存储介质上，比如支持复杂的关系查询、大数据挖掘的数据库或大数据平台上。



WeBASE-Collect-Bee包括三个模块：区块获取模块、区块处理模块和持久化模块。

- 区块获取模块：根据区块ID获取相应的区块；
- 区块数据处理：解析区块数据、区块交易数据；
- 账户数据处理：解析区块链账户数据。

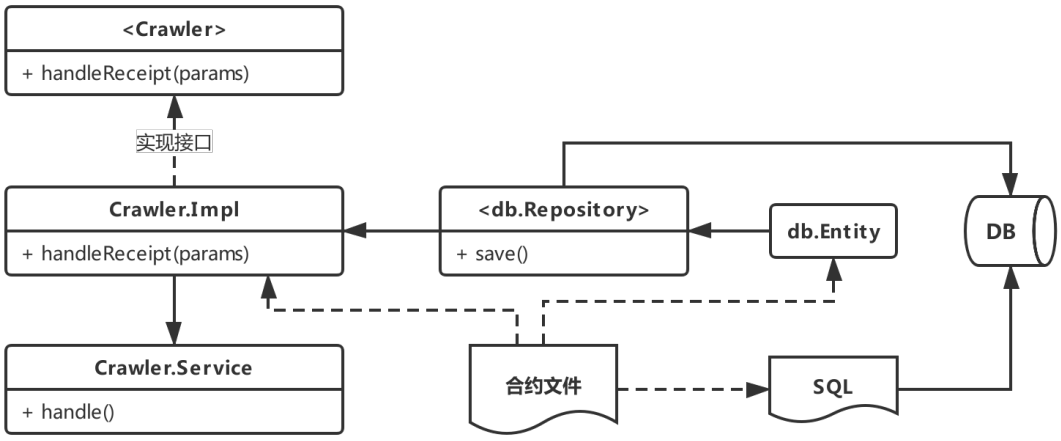
用户只需要提供链的相关配置及数据库配置，就可以一键导出链上的数据，后续可以采用SQL的方式对数据库中的数据进行操作，同时只要确保WeBASE-Collect-Bee服务正常运行，数据库可以几乎实时同步链上的数据。

用户说：业务数据获取工作量大，且不便于维护和复用，便有了WeBASE-Codegen-Monkey

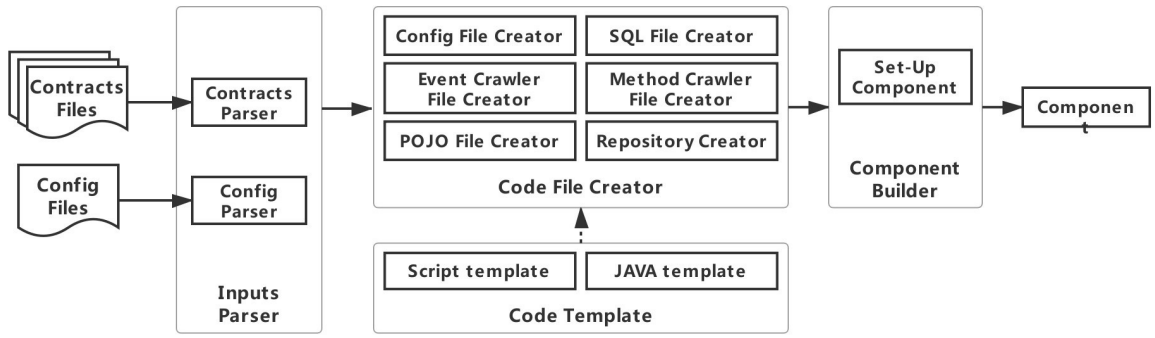
只有区块数据是远远不够的，用户更关心业务数据，即交易数据。交易数据通过调用智能合约方法上链，同时为了便于查看交易执行情况，交易中存在大量的log数据，即event数据，event数据对业务分析至关重要。

要想获取交易数据和event数据，必须针对区块链上的每一笔交易和event，根据智能合约进行解析，核心模块至少包括：交易/event和数据解析、数据库访问接口、POJO、SQL几个模块。

如下图所示，假设我们的业务包含2个智能合约、每个智能合约包含2个接口和2个event，每个模块都需要独立编写代码，那么至少需要编写32个代码文件或脚本，工作量相当大，维护复杂，且不能复用。



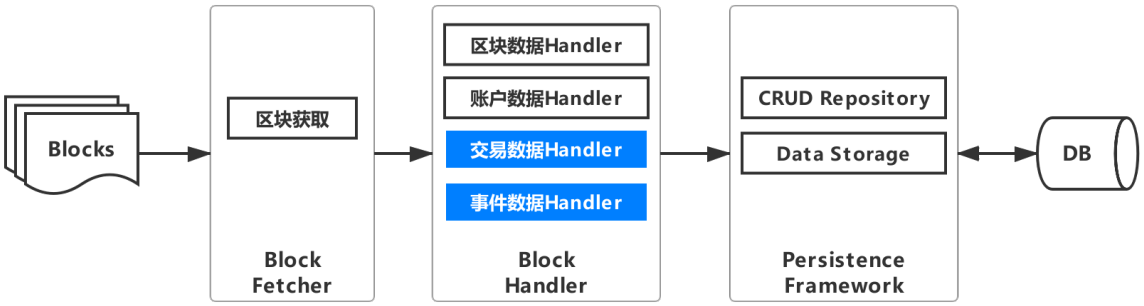
基于此，我们设计了WeBASE-Codegen-Monkey 。WeBASE-Codegen-Monkey用于生成解析和存储交易/event数据的所有核心代码。用户无需编写任何代码，只需要提供智能合约文件，WeBASE-Codegen-Monkey会自动解析合约，生成获取交易/event数据的所有代码文件，并自动和WeBASE-Collect-Bee组装成为一个独立的服务。WeBASE-Codegen-Monkey架构如下图所示。



WeBASE-Codegen-Monkey包括合约解析模块、代码模板模块、代码生成模块、组件组装模块。

- 合约解析模块：解析智能合约文件，获取合约中的交易和event对象；
- 代码模板模块：用于生成获取交易/event数据的代码模板；
- 代码生成模块：根据获取的交易和event对象，填充代码模板，生成代码文件；
- 组件组装模块：用于将生成的代码和WeBASE-Collect-Bee组装成一个独立的服务。

由于新增了交易/event数据的获取，相应的WeBASE-Collect-Bee架构演进如下，新增交易数据处理模块和事件（event）数据处理模块。用户只需要提供智能合约文件，就可以获取几乎链上的所有数据。



从用户使用的角度来讲，只需要将链的证书文件和智能合约文件放在指定的目录下，然后进行节点和数据库配置，并设置智能合约的包名即可。

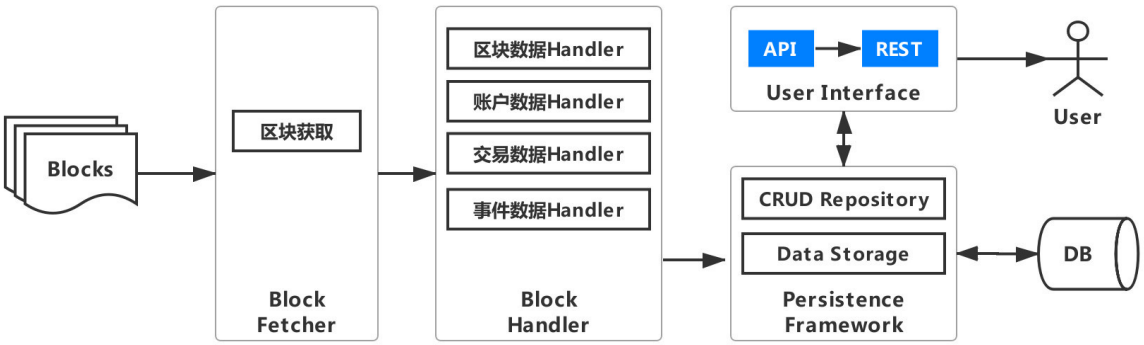
```
#### 节点的IP及通讯端口、组号。NODE_NAME可以是任意字符和数字的组合
system.nodeStr=[NODE_NAME]@[IP]:[PORT]
system.groupId=[GROUP_ID]

#### 数据库的信息，暂时只支持mysql；serverTimezone 用来设置时区
system.dbUrl=jdbc:mysql://[IP]:[PORT]/[database]?useSSL=false&serverTimezone=GMT
↪%2b8&useUnicode=true&characterEncoding=UTF-8
system.dbUser=[user_name]
system.dbPassword=[password]

# 合约Java文件的包名
monitor.contractPackName=[编译Solidity合约时指定的包名]
```

用户说：有了数据，要使用还需要开发数据访问接口，不方便，便有了用户接口

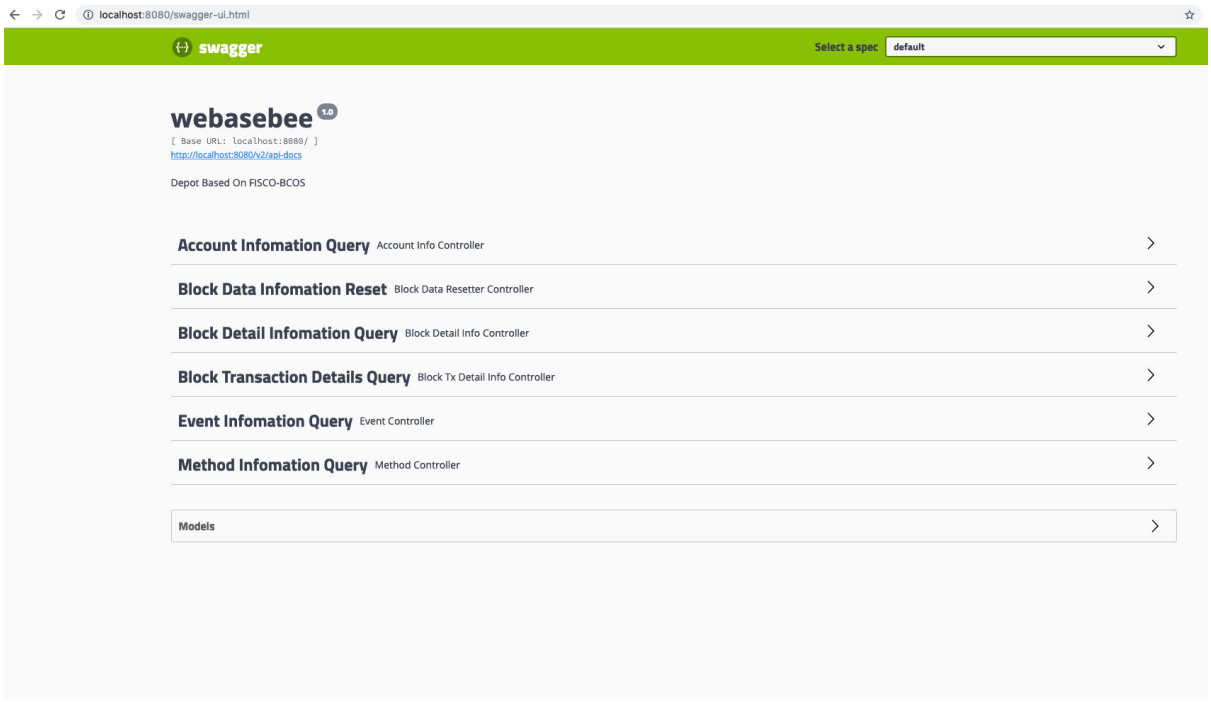
对于用户而言，尽管我们已经将用户关心的所有数据导出到DB中，且每一张表的名称和交易方法/event名称对应，字段名称直观易懂，但如果用户想在自己的系统中使用数据，还需要编写大量的数据库访问接口。基于此，WeBASE-Collect-Bee中增加了**用户接口模块**，如下图所示。



用户接口模块提供两种数据访问方式，一种是API方式，支持用户系统本地调用；另一种为REST方式，可通过http方式访问，降低了业务的耦合性，支持跨平台调用。

用户接口按照数据类型分成四类接口：区块数据接口、账户数据接口、交易数据接口和事件数据接口。其中，每类接口中都支持基于块高、hash或账户的查询，同样也支持基于时间、特定字段的复杂查询。用户接口使得用户在使用数据导出组件时，可以和自身系统零成本对接。

除此之外，为了便于用户进行可视化数据验证和查看，数据导出组件集成了Swagger插件，当用户完成数据导出服务部署后，可在浏览器中输入http://your_ip:port/swagger-ui.html，查看所有用户接口，并输入查询条件，进行可视化查询，如下图所示。



用户说：有了数据和查询接口，但是不够直观，老板看不懂，便有了Grafana集成

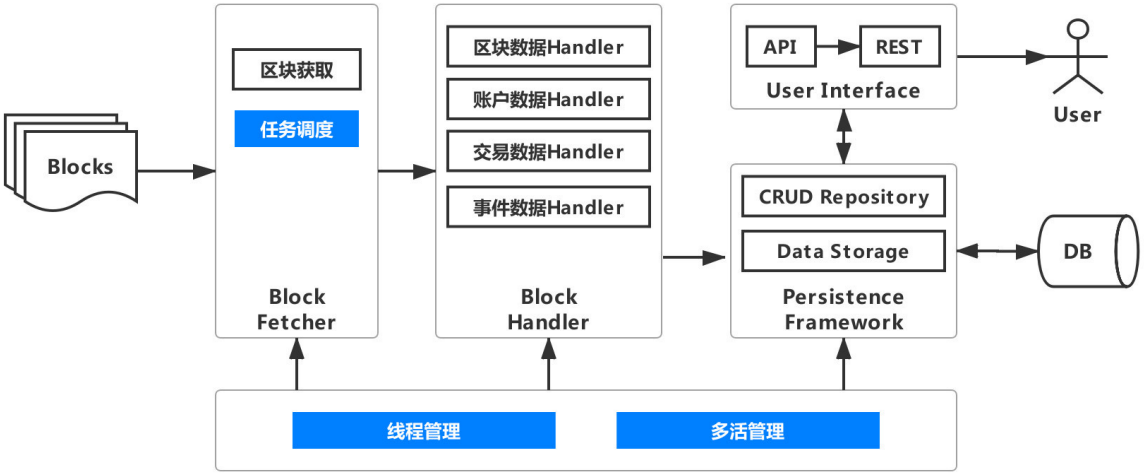
为了更加实时、可视化的展示区块链数据，满足产品、运营等非技术人员的需求，基于轻量化考虑，最终我们选择了可视化数据插件Grafana。

然而Grafana展示数据需要针对每一张表数据编写Dashboard模板，学习和编写模板成本很高。不过不用担心，WeBASE-Code-Monkey会自动生成Grafana脚本。用户只需要安装Grafana并进行数据源配置，然后导入生成的Dashboard模板脚本即可，实现数据可视化可在1分钟内完成，如下图所示。

用户说：一个服务导出数据太慢，服务挂了怎么办，便有了多线程处理和分布式部署

对于数据导出服务，一旦链的性能很高，超过单机数据导出的TPS，那么DB中永远获取不到最新的数据，且数据会越来越旧，显然无法满足业务对数据的诉求。同时单机处理的风险在于系统稳定性很差，

一旦单机服务挂掉，就无法获取最新数据，且无法使用用户接口进行交互。因此我们引入了多线程处理和分布式部署，架构演进如下图所示。



线程管理

线程管理相对简单，只需要关闭多活开关，开启单节点任务模式，同时对独立线程处理的区块数进行设置即可。如下，系统默认开启4个线程，进行区块抓取和处理。

```
#### 当此参数为false时，进入单节点任务模式
system.multiLiving=false
#### 多线程下载的分片数量，当完成该分片所有的下载任务后，才会统一更新下载进度。
system.crawlBatchUnit=100
```

多活管理

为了进一步提升数据导出效率，确保系统稳定性和容错性，我们集成了Elastic-Job，支持分布式部署、任务分片、弹性扩容、并行调度和定制化流程任务。数据导出组件在分布式环境下，首先通过一个SimpleJob对区块进行抓取，然后通过DataflowJob对区块进行处理。

考虑到使用Elastic-Job的成本，系统会自动生成任务分片和执行策略的所有配置，除过几个必要配置，用户不需要做任何事情就可以完成多活配置和部署。必要配置如下。

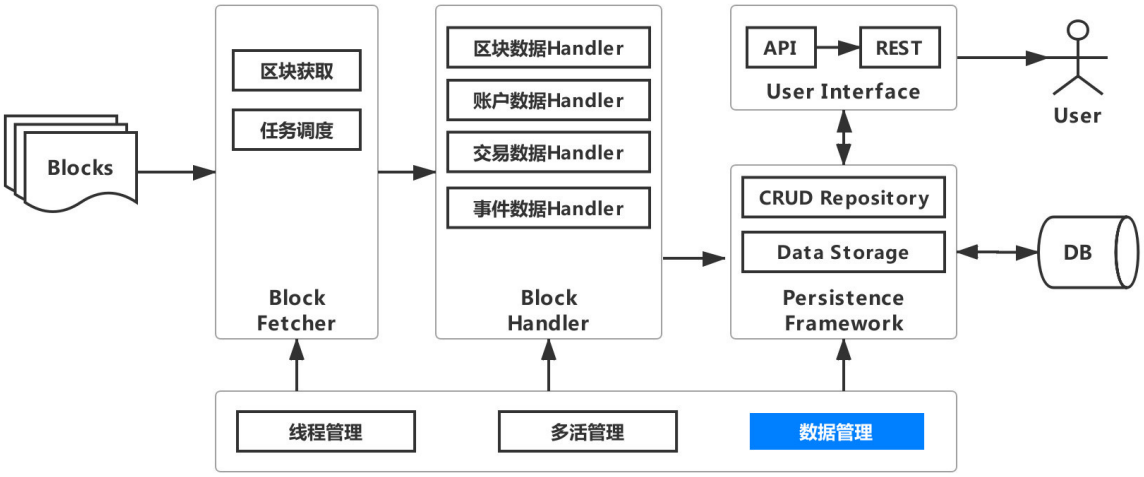
```
#### 当此参数为true时，进入多节点任务模式
system.multiLiving=true

#### zookeeper配置信息，ip和端口
regcenter.serverList=ip:port
#### zookeeper的命名空间
regcenter.namespace=namespace
```

用户说：导出的数据量太大，查询和存储性能跟不上，容易崩，便有了分库分表

当区块链上存在海量的数据时，导出到单个数据库或单个业务表会对运维造成巨大的压力，造成数据库性能的衰减。一般来讲，单一数据库实例的数据阈值在1TB之内，单一数据库表的数据阈值在10G以内，是比较合理的范围。

如果数据量超过此阈值，建议对数据进行分片。将同一张表内的数据拆分到多个或同个数据库的多张表中，即分库分表。数据导出引入数据管理模块，架构演进如下图所示。



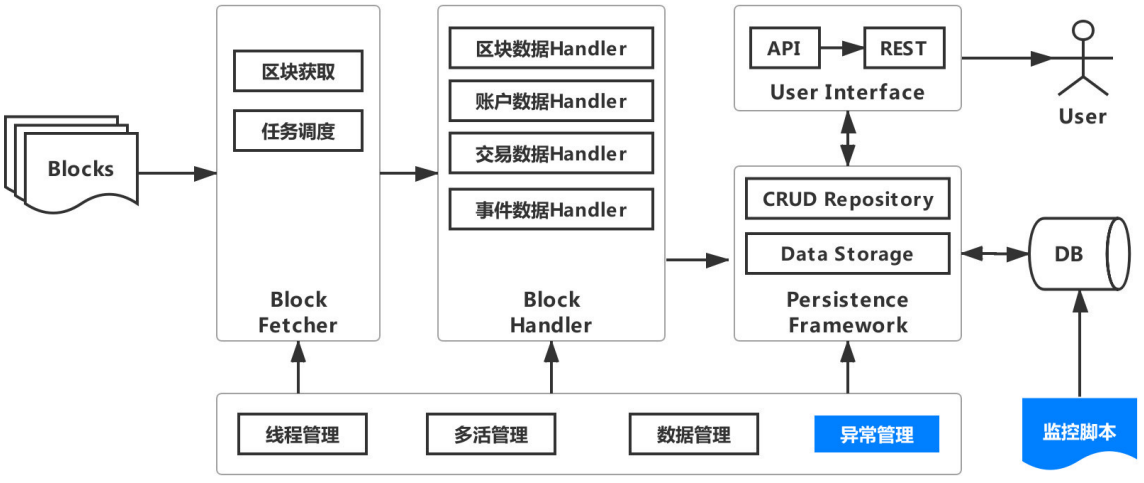
数据管理模块集成Sharding-JDBC，支持分库分表和读写分离。用户只需要设置分片数，系统会自动生成分片策略配置。如果需要支持读写分离，可以在生成的WeBASE-Collect-Bee中进行配置，可参考数据导出高级配置。在用户接口层，用户可以无感使用同一套接口，感觉是在同一个库或表中操作一样。

```
#### 交易和event数据分片配置
system.contractName.[methodName or eventName].shardingNO=XXX
#### 区块和账户数据分片配置
system.sys.[sysTableName].shardingNO
```

用户说：链上出现临时分叉或服务异常，DB数据不一致，便有了异常处理和监控报警

数据导出服务旨在导出链上数据。在确保性能、稳定性、可扩展性的前提下，如果选择了非高度一致性的共识机制，链上会有一定概率出现临时分叉，导致数据库出现脏数据；或者由于网络/服务异常等原因，导致数据导出服务无法导出链上最新数据。

为了确保数据正确性、数据一致性，数据导出组件新增异常管理模块和监控脚本，至此，数据导出组件功能已非常强大，完整架构如下图所示。



异常处理

异常处理主要为了校验导入DB数据的正确性，如果选择了非高度一致性的共识机制，链上会有一定概率出现临时分叉，可能导致DB中的数据和链上的数据不一致。

基于6个区块后链上分叉概率趋近于0的理论，每次导出数据之前，异常管理模块会针对最近6个区块进行hash校验，如果一致，则继续导出。如果不一致，则回滚大于等于异常区块高度的所有区块数据，然后重新拉取和导入。

监控报警

即使有分布式部署保证数据导出服务的稳定性，但无法保证链上数据是否真正的导出。监控脚本monitor.sh用于监控链上数据是否真正的导出。主要基于两个维度：

- 一段时间内，DB中的数据开始落后于链上数据，直到达到某个阈值；
- 链上块高增加，一段时间内，DB中块高不变。

用户可以根据实际情况自行配置，如下所示。

```
#### 数据导出的进度落后于链高度的报警阈值，输出报警日志
threshold=20
#### 如当前块高增长，但完成导出的区块数量增长小于等于1，则输出报警日志
warn_number=1
```

用户你尽管说

用户说：链上历史数据没有价值，全量抓取浪费资源，只想要最近的数据。

数据导出组件支持从特定时间或特定块高拉取数据，只需要进行简单配置即可，如下。

```
#### 设置数据导出的起始高度
system.startBlockHeight=XXXX
#### 设置数据导出的起始时间
system.startDate=XXXX
```

用户说：我只想要部分交易/event的数据，或者只要交易/event中的部分字段数据。

在实际场景中，用户可能并不需要全量数据，只需要特定交易/event的数据，或者特定交易/event中的特定字段数据，数据导出支持个性化导出，可按如下配置。

```
#### 设置特定交易或event数据是否导出，默认全部导出
monitor.[contractName].[methodName/eventName].generated=on/off
#### 忽略特定合约特定event的特定字段不进行抓取
monitor.[contractName].[methodName/eventName].ignoreParam=XXX,XXX
```

用户说：链上数据增长的不快，线程一直空转浪费机器资源。

数据导出组件可以灵活配置数据抓取任务的频率，可通过如下配置项进行修改。

```
#### 所有method和event的抓取频率，默认5秒轮询一次
monitor.default.frequency=5
```

用户说：交易/event的个别字段采用string类型，数据库默认255，会导致入库报错。

数据导出对于智能合约中的bytes和string类型默认采用VARCHAR(255)，这种设计是为了节省数据库资源，能满足绝大部分场景，但同时也存在个别字段超过255长度，导致数据不一致。因此数据导出组件针对超长字段提供配置，可有效合理利用数据库存储空间，避免资源浪费。

```
#### 配置特定合约中特定交易/event的特定字段在数据库中的长度
length.[contractName].[methodName or eventName].[paraName]=1024
```

用户说：导出原始数据不是我想要的，比如某个账户的最新余额，而不是所有变更历史。

数据导出组件会导出链上的全量数据，包含所有的历史数据。针对用户的特定数据需求，数据导出组件支持本机编译，复制执行包到其他服务器上运行，在执行generate_bee.sh后，会生成完整的WeBASE-Collect-Bee服务工程，用户可基于工程源码进行二次开发，修改导入数据的策略。

用户说：整体服务功能很强大，但是我只想把部分功能集成到自己的系统中。

数据导出服务考虑到模块耦合性，将整个WeBASE-Collect-Bee拆分成为区块获取、数据解析，数据库操作和公共模块，用户可按照自己的需求直接使用独立的模块。

用户不仅仅可以说

对于任何一个方案，没有最完美的，只有最适合的。数据导出组件致力于解决区块链大数据场景问题，不仅可以适用于具体的业务诉求，对于一个区块链项目而言，还可以作为贯穿开发、测试和运营的利器，提升研发、测试和运营的效率。

对于用户的需求，我们的一贯态度是：欢迎说，但也不怕说，必要时你必须得说。面对志同道合的小伙伴，我们也早已给出答案：聊，随时恭候；来，双手欢迎。

除了积极参与区块链领域的技术开源，我们也致力于和广大用户共建开源生态，用户不仅仅可以说，还可以加入我们，这里不仅有前沿的黑科技，还有诗和远方！

链接指引

- 区块链底层平台FISCO BCOS代码仓库
- FISCO BCOS技术文档
- 数据导出代码仓库
- 数据导出技术文档

WeBASE节点前置组件功能解析

作者：何硕彦 | FISCO BCOS 核心开发者

FISCO BCOS搭链脚本已经让开发者有极致的搭链体验，怎样能快速搭建一个区块链可视化界面，以便跟区块链交互，WeBASE-Front就是可以最快满足这个期望的组件。WeBASE-Front为开发者提供区块链交互最小功能的子集，安装轻量便捷，无需安装任何第三方组件。在搭完节点之后搭建WeBASE-Front，可以在浏览器打开界面，快速开启区块链体验之旅。

WeBASE-Front还提供了很多友好且实用的功能：

- 在WeBASE-Front首页界面，开发者可以查看到区块链的区块信息、交易信息、群组信息、节点数量、节点状态等，**对区块链网络的核心信息一目了然**。
- WeBASE-Front提供了合约开发IDE，方便开发者在上面编写调试智能合约，**快速开发自己的区块链应用**。
- WeBASE-Front集成了Web3SDK，并对Web3SDK接口进行了封装，开发者可通过HTTP请求，调用WeBASE-Front接口和区块链节点进行交互。这种方式屏蔽了SDK语言的限制，**任意语言的开发者都可以通过调用WeBASE-Front的接口与区块链交互**。

当然，WeBASE-Front的功能不限于此。作为WeBASE家族成员之一，该组件配合WeBASE-Node-Manager和WeBASE-Web一起使用，作为节点前置，对区块链网络的全方位监控，实现企业级区块链监控功能。

功能介绍

WeBASE-Front主要有以下五大功能：

一、数据概览

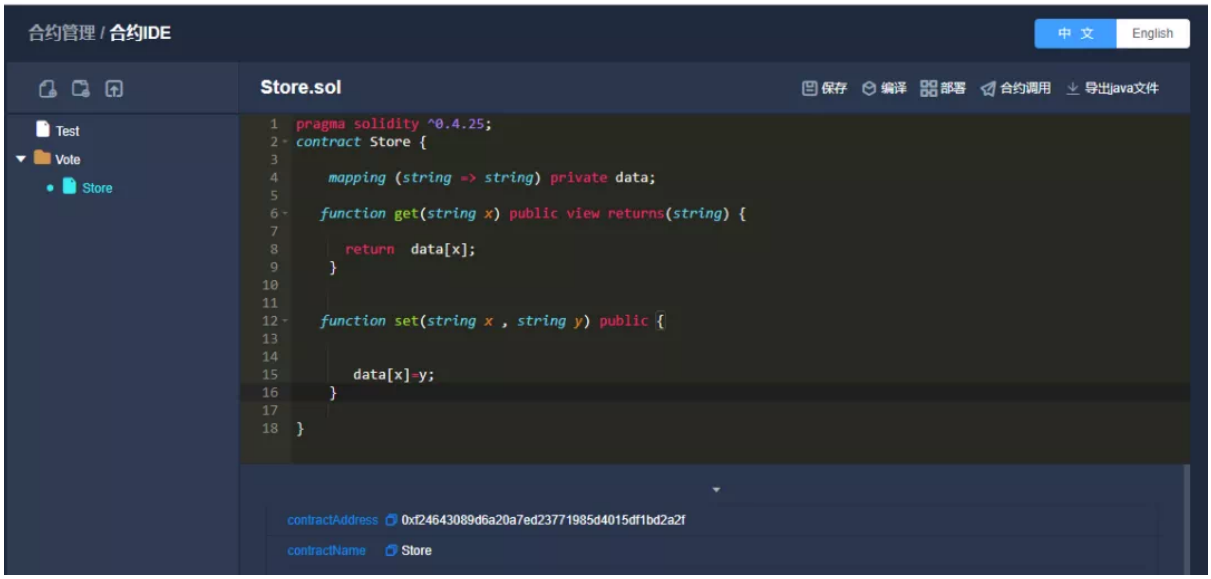
显示开发者关心的常用数据：**区块块高、节点数量、交易总数、待交易数量，**并支持区块信息和交易信息的查询。

二、节点管理

显示区块链网络的**节点数量**、**节点ID**、**块高**、**pbftview和节点运行状态**，并支持区块链群组的动态切换。

三、合约管理

此为WeBASE-Front的核心功能，开发者可以在上面**编写**、**编译**、**调试合约**，以及**一键生成合约的JAVA类**，已部署合约会存在H2嵌入式数据库里，可以在合约列表里查询历史合约。



四、系统监控

系统监控分为两方面监控：

1. 节点服务器的性能指标监控，包括**CPU**、**硬盘**、**内存**、**上行带宽**、**下行带宽**；
2. 区块链节点的性能指标监控，包括**区块高度**、**pbftview**、**待打包交易数**。

节点长久运行后，可以通过此功能查看服务器的性能。

五、私钥管理

生成椭圆曲线公私钥对，支持**导入导出私钥**，并支持给地址取别名，方便地址记忆。部署合约前需要先创建公私钥对。

技术剖析

WeBASE-Front是基于FISCO BCOS提供的**spring-boot-starter**（请参考文末链接）项目的一个开发实例。**Web3SDK接口封装、动态切换群组、部署调用合约（无需生成JAVA类）、公私钥对产生，**这些通用功能请参考WeBASE-Front代码，开发者可以借鉴并编写自己的springboot应用。

为了方便安装和使用，WeBASE-Front采用了轻量级的**H2嵌入式数据库**，后端使用SSH框架，使用JPA访问数据库；前端采用**VUE**框架开发，前端资源内置在后端springboot服务中，无需安装配置nginx和mysql这些步骤，直接启动JAVA服务即可访问界面。

产生的公私钥和已部署的合约都会存储在H2数据库中，方便查询历史记录。性能监控功能采用的是**sigar**数据收集组件。采集的数据也会存在H2数据库中，但只保存最近一周的监控数据。

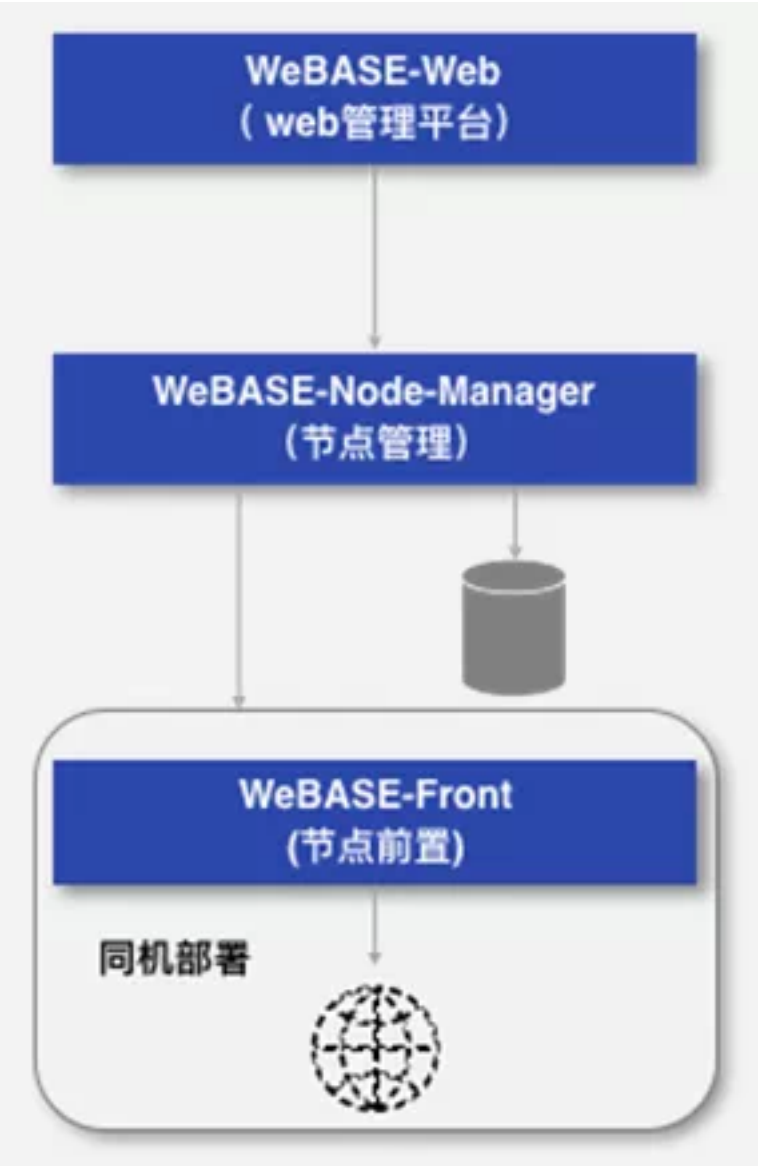
部署方式

作为节点前置，WeBASE-Front需要跟节点同机部署。一台机器部署多个节点时，建议只部署一个WeBASE-Front服务即可。

WeBASE-Front部署方式有以下三种:

1. 单独部署作为独立控制台使用，并配有界面，部署简单快捷，只需下载WeBASE-Front应用，替换节点证书即可启动。推荐初学者和开发者使用此种部署方式，使用此组件查询区块链的相关信息和开发调试智能合约。（安装请参考文末链接）
2. 方式1中，WeBASE-Front作为可视化控制台使用，私钥默认加密存储在H2数据库中，如果需要更安全的私钥保护方案，可以结合WeBASE-Sign服务一起部署使用，私钥保存在WeBASE-Sign中，WeBASE-Sign服务负责对交易数据进行签名，提供更安全的私钥保护方案。

   此方式在方式1的基础上再部署WeBASE-Sign服务，如果对私钥安全性要求较高，请采用此种部署方式。（WeBASE-Sign服务请参考文末链接）
3. 结合WeBASE-Node-Manager和WeBASE-Web服务一起部署使用，这里WeBASE-Front只作为节点前置，多个节点前置统一由WeBASE-Node-Manager管理，WeBASE-Node-Manager有一套spring security实现的鉴权登录系统，并会拉取链上的区块信息和交易信息，存储在Mysql数据库中。生产环境下建议使用此种方式，架构图如下所示。（WeBASE安装部署请参考文末链接）



总结

WeBASE-Front作为一个搭建便捷、功能强大的区块链组件，既可以独立使用，作为可视化控制台，方便开发者与区块链交互，也可以配合WeBASE-Node-Manager和WeBASE-Web一起使用，实现生产环境的区块链监控功能。

WeBASE-Front还在持续优化开发中，未来会增加越来越丰富的功能，如联盟链增删节点的系统管理功能、交易解析功能等。当然，持续迭代升级中也会保持它的易用性和便捷性。欢迎社区朋友提PR和ISSUE，一起参与优化。

链接指引

- Spring-boot-starter
- Sigar数据收集组件
- WeBASE-Front单独安装部署说明
- WeBASE-Sign服务

- WeBASE安装部署

22.5.3 一键搭链脚本工具

FISCO BCOS快速建链实现之路

作者：白兴强 | FISCO BCOS 核心开发者

跟很多开发者一样，团队刚开始搭链时，也经历过迷之困惑的阶段：安装哪个版本，怎么编译这么久还容易出错？配几个节点，都用什么IP端口？证书从哪里来，放到哪里去？怎么验证我的链确实搭起来了？...

相信从FISCO BCOS1.X版本上手的工程师们，内心都有一座小火山，面对超长的文档、超多的操作步骤.....每次版本部署，耗费大量时间，工程师头上都快能摊熟一枚鸡蛋。而数据表明，一个软件如果15分钟还使用不起来，用户就会流失。 为了浇灭大家心中的小火山，一起愉快地玩耍区块链，FISCO BCOS的易用性优化势在必行。团队的第一个目标是让开发者**在5分钟内搭起开发测试链**，这就需要一个《哈利•波特》里召唤术一样的命令，我们把它称为**build_chain**。 本文将聊聊build_chain脚本的诞生记，以及当前脚本能够提供的帮助。

build_chain脚本的诞生

首先是去掉编译步骤，源码编译不但需要安装下载依赖，还需要配置开发环境，即便这两步一切顺利，编译过程中还可能因为内存不足而失败，更何况下载依赖经常受到网速影响导致下载失败。于是我们提供了预编译的二进制发布包，让用户跳过冗长的编译阶段。 马上我们又发现了新的问题，即使针对不同平台都提供二进制发布包，用户的环境是千变万化的，预编译程序所依赖的动态库的安装又成了问题。于是我们想到了提供静态编译的二进制发布包，兼容多种Linux64位操作系统，不依赖任何其他库，省时又省力。为了实现静态编译，我们不惜重新实现部分功能，以去掉对不提供.a的外部库的依赖。 接下来我们尝试减少部署步骤，减轻对用户的压力。

配置项太多太灵活，我们优化配置，所有配置都提供合适的默认值，删除不需要灵活定制的配置项。json格式的配置文件阅读不够直观，手工修改容易因为格式问题导致错误，我们替换为更清晰的ini文件。 系统合约手动部署太麻烦，我们借助预编译合约实现内置系统合约来管理链上配置。 手工搭建和工具脚本搭建的节点目录结构不统一，我们整理文档，统一工具创建的目录结构，提供辅助脚本。

经过上述这些优化，我们认为可以有更轻量级的部署方式，可以尝试通过一个脚本来完成部署过程中的所有事情。 相比于大而全的部署工具，脚本更轻更快；而相比于手工部署的繁琐，脚本能够更简单。这样，build_chain脚本就诞生了。

build_chain脚本提供的帮助

这个脚本能够完成环境检查、参数解析、FISCO BCOS二进制发布包下载、公私钥证书生成、配置文件生成和工具脚本生成等功能，支持MacOS、Linux 64bit、docker模式和国密版本搭建。 然而实际使用后，我们发现在家里的网络条件下，脚本下载二进制发布包需要很长时间，导致5分钟内无法完成搭起一条FISCO BCOS的链。 为了能达成5分钟建链的目标，我们又新增了CDN支持，在哪怕网络条件不是很好的情况下，也可以顺畅地在5分钟内完成建链。心中的小火山熄灭了。

具体而言，build_chain脚本能够提供的帮助包括以下几个：

环境检查

build_chain脚本需要使用OpenSSL来生成节点所需要使用到的相关证书文件，而FISCO BCOS 2.0要求使用OpenSSL 1.0.2以上版本。只有找到符合要求的程序版本，脚本才能继续执行。 需要注意的是，MacOS自带的是LibreSSL，所以需要用户使用brew install OpenSSL安装OpenSSL。

解析参数

build_chain脚本支持很多自定义参数，例如-p指定节点使用的端口范围、-f搭建指定配置的网络、-g搭建国密版本、-v指定FISCO BCOS程序版本号、-o指定输出路径等，可参考详情

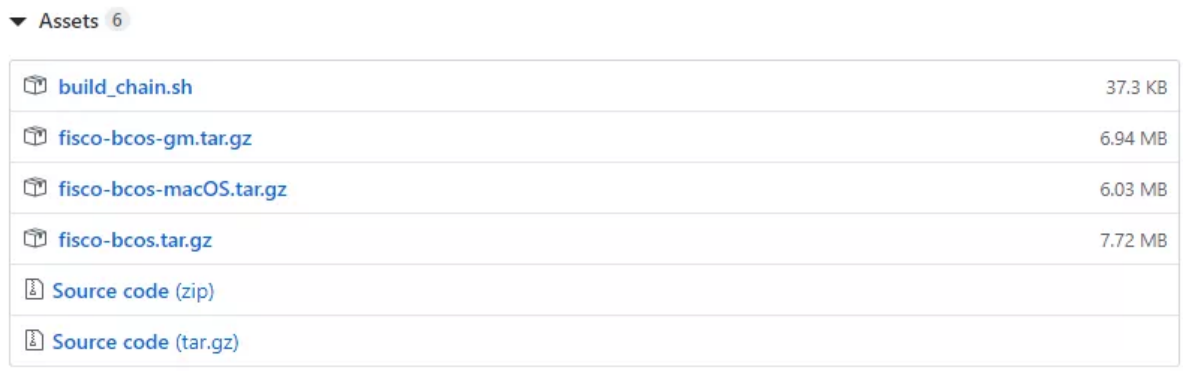
```
Usage:
    -l <IP list>                    [Required] "ip1:nodeNum1,ip2:nodeNum2" e.g:"192.168.0.1:2,192.168.0.2:3"
    -f <IP list file>               [Optional] split by line, every line should be "ip:nodeNum agencyName groupList". eg "127.0.0.1:4 agency1 1,2"
    -e <FISCO-BCOS binary path>     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 / external, rocksdb is recommended
    -d <docker mode>                Default off. If set -d, build with docker
    -c <Consensus Algorithm>        Default PBFT. If set -c, use Raft
    -m <MPT State type>             Default storageState. if set -m, use mpt state
    -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
    -T <Enable debug log>           Default off. If set -T, enable debug log
    -F <Disable log auto flush>     Default on. If set -F, disable log auto flush
    -h Help
```

获取FISCO BCOS可执行程序

FISCO BCOS提供标准版和国密版本的预编译可执行程序，可以在大部分x64的Linux机器上运行。另外，为方便开发人员调试，同时提供了MacOS版本的可执行程序。

- build_chain脚本会根据操作系统和是否国密下载对应的可执行程序。
- 当从GitHub下载可执行程序较慢时，会自动切换到CDN下载。如下图可以看到fisco-bcos.tar.gz只有7.72M。
- 当不使用-v选项指定版本时，脚本会自动拉去GitHub上FISCO BCOS发布的最新版本，使用-v选项时，则下载指定版本的可执行程序。

对除官方明确的Ubuntu 16.04+和CentOS 7.2+以外的平台，上生产时建议使用源码编译获得的可执行程序，然后通过-b选项和-f选项搭建区块链网络。



生成私钥证书

FISCO BCOS支持证书链，默认使用三级证书链结构，使用自签CA证书作为链的根证书，使用CA签发的机构证书用于区分机构，然后使用机构私钥签发节点所使用的证书。 节点的conf目录下会有ca.crt、node.key、node.crt三个文件，节点使用这三个文件建立双向SSL链接，使用node.key在共识过程中为区块签名。 如果是国密版本，则脚本会下载TaSSL工具，生成国密版本的证书文件。

生成配置文件和工具脚本

build_chain脚本中内置了FISCO BCOS节点所需要使用的配置文件模板，根据用户指定的参数修改，生成节点所使用的配置文件(可查看配置文件的介绍)。

同时，为方便用户启动和停止节点，节点目录下还会生成start.sh和stop.sh（可查看节点目录结构说明）。

总结

下面来总结一下，FISCO BCOS团队提升部署速度、实现快速建链的几个要诀：

- 1．提供静态编译的二进制发布包，兼容多种操作系统，让用户跳过冗长的编译阶段。
- 2．简化配置，极大限度采用能保证最大成功率的默认参数，最小化用户需要关注的信息，用户只需关注少量网络配置。
- 3．标准化目录结构，无论是用一键搭链、企业级搭链、手工搭链...生成的目标都是一样的，降低了差异化带来的复杂度。
- 4．巧用脚本，build chain脚本可自动串起从准备环境到启动所有链节点一系列常规操作步骤，自动处理各种可能的小异常，让整个过程显得行云流水。
- 5．优化依赖库地址、网络速度等，极大减少用户的等待消耗，谈笑间，链已经搭好。

对于想通过手工搭建进一步学习的同学和生产环境使用，建议使用我们提供的企业级部署工具generator。在性能和易用性方面，做再多努力都是值得的，我们会继续努力优化，非常欢迎各位社区的参与者提优化建议和bug。

参考链接

企业级部署工具generator：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/enterprise_tools/index.html

22.5.4 交互式控制台Console

FISCO BCOS 控制台详解，飞一般的区块链体验

作者：廖飞强 | FISCO BCOS 核心开发者

我们对Linux系统中的命令行终端已经很熟悉了，通过shell终端可以顺畅地使用Linux。类似地，FISCO BCOS联盟链中，也有这样的命令行终端，称为控制台。控制台是开发者探索区块链世界的助推器，其提供的各种功能有助于跨越横在区块链入门到精通之间的座座高山，带来“开箱即用”的顺滑体验。

Why: 为什么要做控制台？

体验环境影响用户从“入门到放弃”，或者“入门到精通”。

当学习一个新技术或产品时，除了读文档之外，尝试上手操作，获得第一手体验也尤为重要。如果体验环境配置复杂、操作繁琐，很可能导致用户从“入门到放弃”；而体验环境简单易配置、功能丰富，则会为用户迅速打开一扇新世界的大门，加速用户从入门到精通。

选择何种形式承载这种极速且友好的体验方式呢？如果我们有一个控制台，只要输入单条命令或代码，然后按下回车键就可以返回结果显示在用户面前。这种 “开箱即用”的效果，正是我们所期待的体验方式。

可以为FISCO BCOS实现极速体验的控制台吗?

FISCO BCOS 1.3版本其实已经具备快速体验的功能，由两部分组成，分别是ethconsole和Node.js工具。其中，ethconsole可以查询链上信息，包括节点、区块和交易信息；Node.js工具提供了部署和调用合约的模板js文件，协助用户实现合约的部署与调用。

但是这种体验方式还不够友好，那一版的ethconsole只能查询非常有限的链上信息，并不能发送交易和管理区块链，功能比较单一；通过Node.js工具需要手工编写部署和调用合约的模板js文件，操作较为繁琐，体验和ethconsole割裂，不那么顺畅。

因此，FISCO BCOS 2.0版本规划时，重点设计了FISCO BCOS 2.0控制台，目标是做一个易用、友好、且功能强大的全新控制台，给FISCO BCOS提供一种极速体验方式。

What: 控制台实现了哪些功能?

控制台每一个功能的实现都源于一个个朴素的操作，一切按实际需求出发，对用户有价值的功能，一一实现。

需求1：区块链是什么，在哪里？看得见吗?

实现查询区块链相关的一系列命令，让区块链看得见摸得着！例如查询区块高度、区块、交易、节点等等，并且根据不同参数，提供不同的查询方式以满足不同条件下的查询需求。

值得关注的是，对于交易和交易回执信息查询命令（getTransactionByHash，getTransactionReceipt等），提供按ABI定义解析详细数据的功能，让交易的输入、输出和event日志信息以解码的方式呈现，而不再是满屏十六进制的天文数字。

需求2：部署和调用合约是使用区块链最核心的诉求，控制台可以直接部署和调用合约吗?

必须可以。控制台推出之前，部署和调用合约有两种选择：一种是借助Node.js工具，即写Node.js客户端部署和调用合约；一种是使用Java SDK, 写Java 客户端部署和调用合约。这两种方式都很强大，但并不是为极速体验设计的，用户需要编写合约之外的部署和调用代码。

因此，控制台实现的效果是，用户写好合约，放入指定路径，在控制台输入一个命令（deploy）即可完成部署，再用call指令就能调用合约接口，不需要其他任何额外的工作（例如将solidity合约转为java代码，编写部署和调用合约的客户端代码等）。deploy命令部署合约之后，会显示一个合约地址，考虑到后续调用合约时还会用到这个合约地址，控制台本地会记录部署的合约地址，并提供getDeployLog命令查看部署过的合约地址列表信息。

另外，FISCO BCOS区块链提供CNS功能，即合约命令服务功能。链上可以记录部署的合约名、版本号和对应的合约部署地址；部署合约时，指定合约名和版本号；调用合约时，指定合约名和版本号（若不指定，使用最近部署的合约版本号）。

这是一种更高级的部署和调用合约方式，也是推荐的部署和调用合约方式。因此控制台实现了利用CNS部署合约命令deployByCNS和利用CNS调用合约命令callByCNS。值得关注的是，为了开发者便于查看处理信息和调试合约，控制台会自动解析合约的输出和event日志信息。

需求3：FISCO BCOS 2.0支持多群组，控制台可以在线切换群组吗?

控制台登录之后，其命令提示符前面显示当前群组号。通过提供switch命令在线切换多群组，即不需要退出控制台就能完成无缝切换。切换之后，命令提示符前面的群组号自动更新，而后可以在切换后的群组中发送命令。

需求4：控制台可以管理区块链吗?

FISCO BCOS 2.0 提供节点管理、系统参数管理、权限管理功能，控制台均提供对应的命令进行操作，方便用户通过简单的命令轻松管理区块链。 其中，节点管理的命令是addSealer(添加共识节点)、addObserver(添加观察节点)、removeNode(移除群组中的节点)；系统参数管理的命令是setSystemConfigByKey(设置系统参数)；权限管理有一系列的命令，管理区块链系统相关功能的操作权限，其具体是以grant开头的赋予权限命令，以revoke开头的撤销权限命令，以及以list开头的查询权限命令。

具体使用参考这里

需求5：可以不用写CRUD合约，操作区块链中的用户表吗?

FISCO BCOS 2.0提供分布式存储，其核心在于表存储设计。CRUD合约是面向表存储的一种合约写法，合约的数据存储在用户表中，合约接口面向表进行增删改查操作。 为了让用户不用写CRUD合约，就可体验分布式存储功能，控制台提供类似mysql语句的形式，提供创建表（create）、查看表（desc）和表的增删改查（insert, delete, update, select）命令。 创建表和增删改命令均是发送一条需要区块链节点共识的交易，与用户编写CRUD合约操作表等效。

需求6：控制台支持国密方式发生交易吗?

控制台提供修改配置文件的国密开关，并下载国密版合约编译器以及替换，这就成为了国密版控制台。因此，当区块链节点是国密版本时，控制台可以连接国密节点并支持部署和调用国密版合约，发送国密版交易。

需求7：...等你来提?

欢迎社区用户积极提意见、建议以及需求（issue或微信社群的方式），同时也可以直接提pull request到官方console仓库，修改和新增你们需要的功能，甚至可以fork源码然后单独定制个人或组织的控制台，开源社区的共建和共享依靠社区用户广泛而积极的参与。

Where: 控制台的价值在哪里

现在，用户只要启动控制台，就可以查询丰富的链上信息，快速部署和调用合约，轻松管理区块链，这些特性给用户带来什么价值呢?

- **面向初学人员**：不用搭建复杂的开发环境，只要简单的节点ip和端口配置，非常轻量级，就可以极速体验区块链功能。当启动控制台后便呈现出一个炫酷的FISCO BCOS标志图案时，会顿时感觉自己成功地与FISCO BCOS产生了第一次连接，从而信心倍增地继续探索区块链。

  配置和启动控制台请参考这里

- **面向开发人员**：开发人员可以利用控制台部署和调试合约，当写好合约后，可以在控制台先部署，采用call指令调用验证合约逻辑，观察合约运行过程和结果。如果业务端采用Java开发应用，可以利用控制台的合约编译工具将Solidity合约一键编译为Java客户端代码文件，供客户端的java工程调用。

- **面向测试和运维人员**：可以在搭建区块链环境后，利用控制台查看链状态，操作区块链配置，测试或检查相关区块链功能。

总之，控制台已成为FISCO BCOS的一个极速体验的窗口，一件强有力的利器，为用户持续带来实际价值。

注：

1、控制台已是FISCO BCOS SDK的一个标配功能，目前已提供Java版控制台（已单独成立了一个console仓库），Python版控制台和Node.js控制台。上文主要是针对Java版控制台进行的介绍，其他控制台相关功能大体相同。

2、控制台文档列表：

Java版控制台使用文档

Python版控制台使用文档

Node.js版控制台使用文档

22.5.5 多语言SDK

麻雀虽小五脏俱全 | 从Python-SDK谈谈FISCO BCOS多语言SDK

作者：张开翔 | FISCO BCOS 首席架构师

FISCO BCOS 2.0从发布起就自带官方控制台，经过社区持续使用打磨，已经足够强大、完善、友好。

社区还有用各种开发语言开发的区块链应用，为满足开发者方便管理区块链节点的需求，目前Python-SDK和Nodejs-SDK已经上架，go语言版本已经在路上。

本文以笔者最熟悉的Python-SDK为例，分享一些SDK开发的点滴，涵盖应用开发流程、协议编解码、网络通信和安全事项等。

FISCO BCOS自带快速搭建特性，五分钟一键搭链后，开发者只需连上区块链节点，写合约、发交易。

控制台和SDK的定位，是帮助用户快速访问区块链，开发测试智能合约，实现业务逻辑。根据“**奥卡姆剃刀**”原则，设计哲学应尽量轻、模块化、浅层次，不引入多余的功能，不给用户和二次开发者造成额外负担。

客户端控制台和SDK就像一辆操控好、配置精的快车，供开发者和用户驾驭，轻松惬意，尽情驰骋于区块链应用之路。

控制台体验

首先结合从准备环境到调用合约的全流程，体验下控制台，命令行交互界面风格如下：

1. 准备环境

在开始之前，请先**通读使用手册和开发文档（非常重要！链接在文末）**，根据文档介绍，step by step初始化环境，安装依赖库，目前Python-SDK支持linux/mac/windows操作系统。

为了连上区块链节点，需要修改本地配置文件，填写区块链节点的对应网络端口，如果选择Channel协议，则需要配置相应的客户端证书。

2. 在线体验

配置好网络后，可以运行console的get系列命令。试下手感，与FISCO BCOS亲密接触。确认链在正常工作，常用的指令有getNodeVersion、getBlockNumber、getPeers、getSyncStatus等，可以用console的usage或help命令了解所有支持的指令。

3. 创建账户

创建一个新的帐户，即代表自己身份的公私钥对，发送交易时会采用私钥签名。创建的帐户用keystore格式保存在本地文件系统里，用户可以设定密码保护这个文件，注意记住创建帐户时使用的密码。

控制台提供的帐户相关命令是**newaccount, showaccount**（参数为账户名和密码）。如果要使用刚创建的新帐户为交易签名，记得把它配置到client_config.py文件的相应位置。

另外，如果账户信息需要高等级保护，则可以进行二次开发。将其放入加密机、TEE等安全区，以及开发秘钥分片、助记词等方案。

4.编写合约

编写一个智能合约，或者参照SDK里自带的智能合约例子修改定制，实现自己的业务逻辑。本文重点关注solidity智能合约，FISCO BCOS还有一种"预编译合约"，采用C++开发，需要和FISCO BCOS底层代码联合编译。

5. 编译部署

对合约进行编译，获得合约的ABI接口文件和BIN二进制代码文件。Python-SDK里有bcos_solc.py文件可帮助开发者简化编译器配置和调用，同时，只要正确配置了合约路径和编译器路径信息，直接运行控制台的部署或调用合约接口指令，也会尝试自动去编译合约，操作体验相当行云流水。

独立部署合约的话，可使用控制台的deploy指令，部署指令成功后会得到新的合约地址。参考命令是./console.py deploy SimpleInfo save，其中SimpleInfo是合约名（不需要带后缀），最后的"**save**"为可选，如果指定了"**save**"，则将合约新地址记录到本地文件里，以便后续使用。

6. 调用合约：

用call或sendtx命令，指定合约名、合约地址、方法名、对应的参数，调用链上合约。

参考命令./console.py sendtx SimpleInfo last setbalance 100，即选择SimpleInfo合约，指向其最近部署成功的地址（用"**last**"指代，可以省掉复制粘贴合约地址的繁琐操作），调用setbalance接口，传入参数100。

交易在链上共识完成后，控制台会自动打印交易回执里的方法返回码、交易Event log信息列表等，供用户查看，如果错误，则打印异常信息。

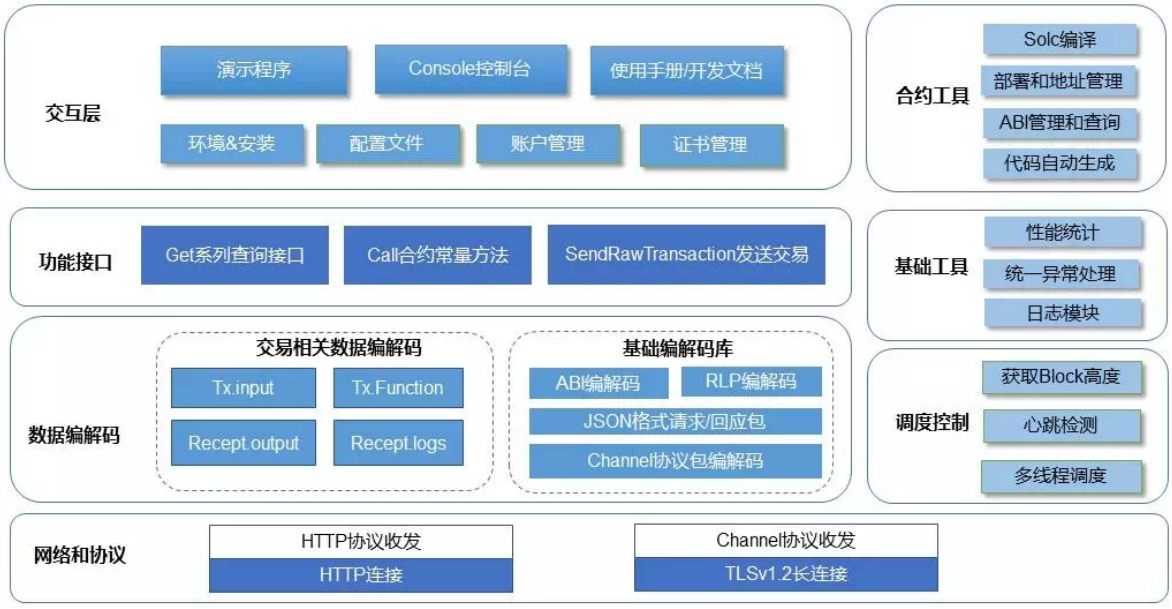
如果一切正常，到此即可基本走通区块链应用之路。

值得一提的是，FISCO BCOS几个语言版本的控制台，都支持按Tab键提示指令和自动完成，帮助使用者流畅无错地操作，提升用户体验。

再进一步，如果希望有丰富多彩的、可视化交互式页面体验，不妨使用WeBASE中间件平台。

深入了解（Dive Deeper）

整个SDK的模块组合如下，可谓是麻雀虽小五脏俱全。



功能接口

支撑控制台等交互模块的，是已经封装完备、开箱即用的功能接口API，包括：

1. get系列

诸多的“get”开头的接口，用于获取链上的各种信息，包括区块、交易、回执、状态、系统信息等等。虽然几十个get接口，但其实现逻辑基本一致，都是指定命令字和参数列表，请求和处理回应，实现起来也很快。

2. call

对应合约的常量方法。所谓常量方法，是指合约里对应代码不修改状态，该请求不会全网广播，仅在指定节点上运行。

3. sendRawTransaction

构建一个交易，用账户私钥签名，发送到链上，这种交易会被广播，进行共识处理，生成的状态数据会被全网确认。部署新合约这个操作，实际上也是一种交易，只是不需要指定目标合约地址。

相关的是**sendRawTransactionGetReceipt**，名字很长，在**sendRawTransaction**基础上增加了获取回执的流程**，**用于简化从发交易到获取回执的闭环流程。

4.更多

针对FISCO BCOS的全局系统配置、节点管理、CNS、权限等系统级功能的API，其原理是读写链上的系统合约，详细指令列表见文末。 开发者可以参考控制台和client/bcosclient.py等代码，进行二次开发，实现更多更酷炫的功能。另外，SDK里内置了一系列的开发库和小工具，帮助管理帐户、输出日志、统一异常处理、简单的性能和耗时统计等。

合约开发相关

围绕着合约开发，Python-SDK实现了合约编译部署、合约地址本地化管理、ABI接口文件的管理，支持代码自动生成（参考codegen.py），一个命令行即可生成供业务端直接使用的代码，如

python codegen.py contracts/SimpleInfo.abi。

solidity合约编译后的ABI文件是个好东西。ABI 全称是 **Application Binary Interface**（应用程序二进制接口），里面详细描述了合约的接口信息，包括方法名、参数列表和类型、方法类型（常量方法，还是交易方法），以及Event log格式定义等等。

对ABI的管理，参见client/datatype_parser.py，加载和解析ABI文件（默认为JSON格式），根据方法名、方法4字节签名、方法类型等维度，灵活查询方法列表和方法定义，并针对方法定义、输入数据等进行编码解码，解析交易返回值、Event logs等。

有ABI定义在手，对合约的操控简直是可以随心所欲，开发者读懂了ABI描述，基本就能全面理解一个合约的输入输出，和合约毫无障碍地对话，这种“**面向远程接口编程**”的思想，很类似WSDL、IDL、ACE、ProtoBuffer和gRPC等经典软件设计。

事实上，整个SDK中最繁琐的是ABI编解码部分，为了兼容EVM，FISCO BCOS在交易处理时沿用了ABI编码，以及兼容RLP协议。

ABI、RLP制定了严格的规范，对基础数据类型、数组和变长数据、函数方法、参数列表等都有特定的编解码方式，否则组件之间无法通信，数据无法解析，虚拟机“不认识”所输入的交易，则不能执行合约。

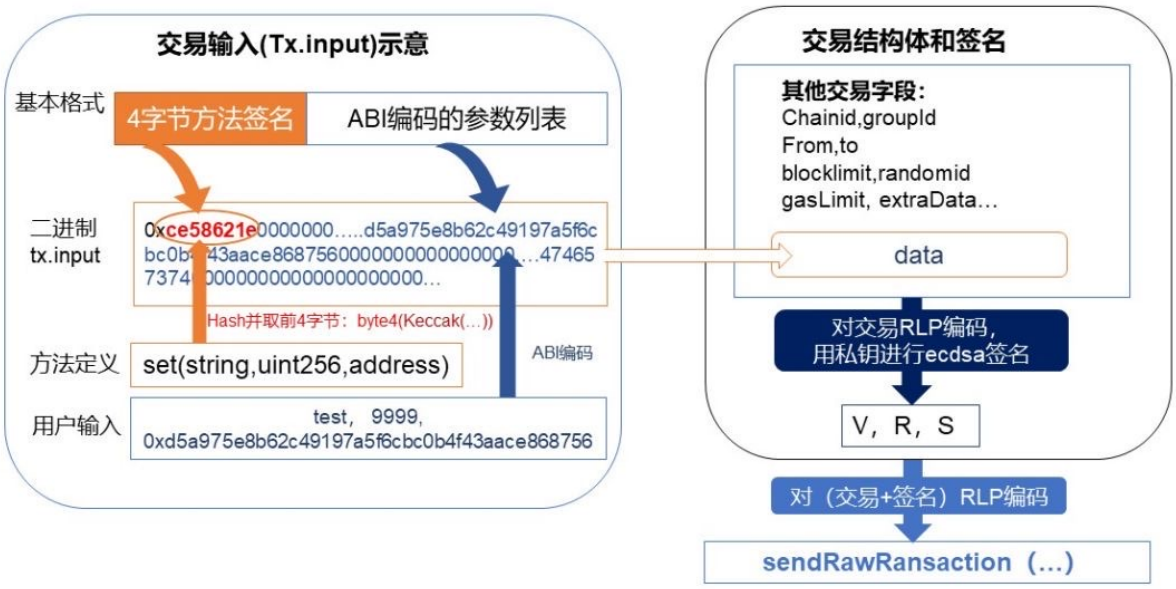
如果自行手写这里的编解码，即使是熟手也得花不少时间，还要能保证测试通过、保持版本兼容，所幸github上已经有eth-abi、eth-utils、rlp等一系列开源项目（多为MIT宽松许可协议），可以引入这些项目且根据具体的需要进行修订（保留原作者声明和版权开源许可），能节约不少工作量，向这些项目作者们致谢，开源就是爽！

交易数据结构相关

在搞定了基础编解码之外，还需要实现 FISCO BCOS交易结构，重点注意支持并行处理交易增加的randomid、blocklimit字段，为支持群组特性增加的fiscoChainId和groupId字段，在交易的receipt里增加的交易output等。

其中，交易的blocklimit定义为“交易生命周期，该交易最晚被处理的块高”，SDK需要定期到链上查询当前块高，以确定当前交易的生命周期（比如，此交易允许在后续一百个区块内被处理）。

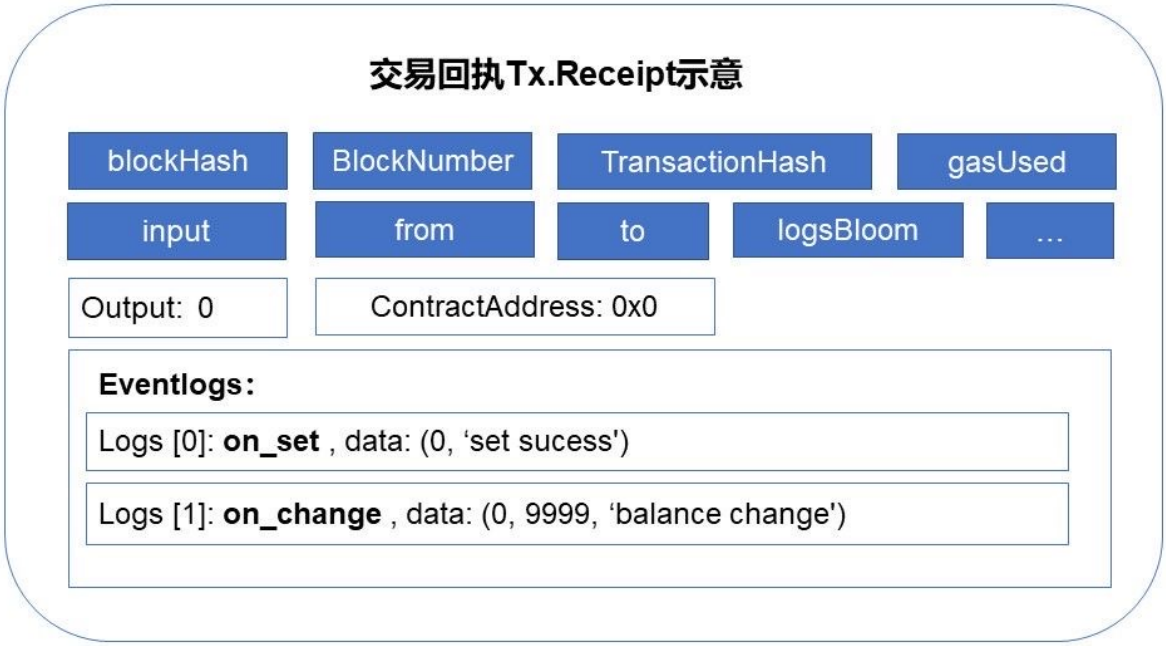
对于开发者来说，清晰理解交易的输入（tx.input）、交易回执（tx.receipt）、交易输出(tx.output)是非常重要的。



交易调用合约里的某一个方法时，首先将方法名字和参数类型列表组合，如**’set(string,uint256,address)’**，对这一段文本进行Keccak-256 (SHA-3)计算，并截取前4个字节做为“方法签名”（signature），然后对传入的参数，根据类型定义依次进行ABI编码，并和”方法签名”拼接一串二进制数据，做为交易的输入数据。

和交易结构体的其他字段（from、to、groupid、randomid等）一起再进行RLP编码，并用帐户私钥进行签名，得到一段二进制请求数据，由sendRawTransaction发往节点，节点收到后，立刻返回交易Hash给到客户端。

交易在链上被网络共识确认，处理完成后，通过getTransactionReceipt接口（传入之前获得的交易Hash），可以获得交易处理的详细结果。



在交易回执中，以下几个字段尤为关键：

1. contractAddress

仅在部署合约交易时有效，表示新合约的地址。

2. output

对应方法的return值，可用于判断业务逻辑的最终处理结果（取决于合约的写法）。

3. Logs

如果在合约代码里，写了一些Event log， 则receipt的logs字段里可以解码出详细的信息。Event log可用于帮助客户端监听、跟踪交易的处理结果，甚至可以帮助开发者调试合约执行过程，相当于在合约里打调试日志。当然，在合约正式发布时，应清除调试的Event log，只保留必要的log，避免冗余信息存到链上。

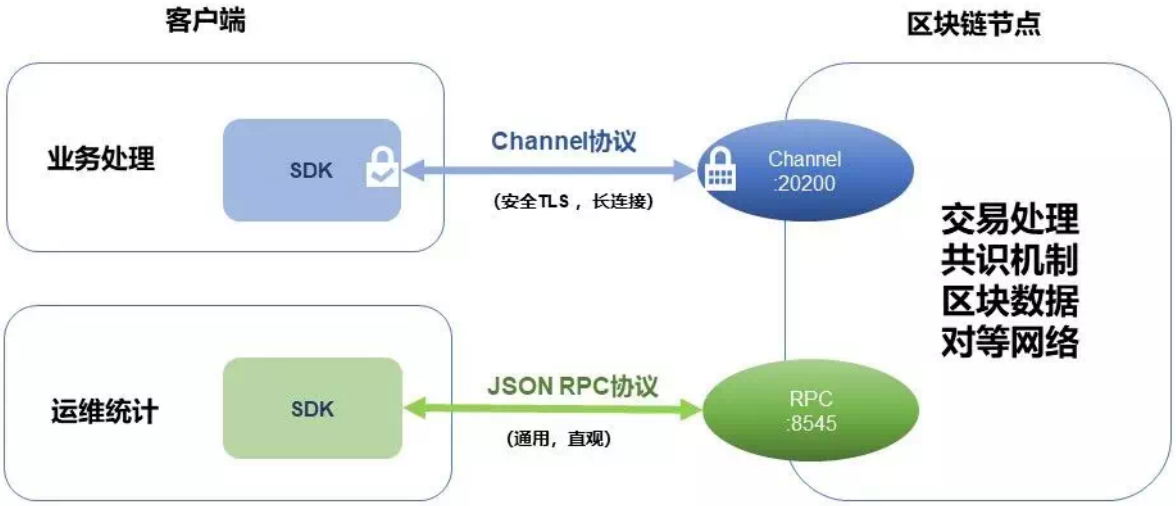
Python-SDK客户端里内置了解析“方法签名”（根据4字节的signature，找到对应的方法定义）、交易input/output、receipt.logs等字段的方法。

在使用控制台命令行时，只要是查询交易和回执的指令，在命令行后面附带合约名（前提是使用者知道这个交易调用的是什么合约），也可以自动解析出相关的数据来，例如：./console.py getTransactionReceipt 0x79b98dbb56d2eea289f756e212d5b6e5c08960beaa8ea8331740fdcfaa8dcab1 SimpleInfo，最后这个“SimpleInfo”为可选合约名，不需要带后缀，要求在contracts/目录下有SimpleInfo.sol文件。

这个贴心小设计，可以帮助开发者直观探秘区块链交易的脉络，对各种信息一目了然，不会迷失在天书一样的十六进制字符海洋里。

网络协议

最后聊聊FISCO BCOS的两种网络协议：JSON RPC和Channel长连接。



JSON RPC连接没有证书验证和通信加密，建议在本身安全可信的环境里使用，比如本机或内网，一般用于运维管理和统计分析场合。

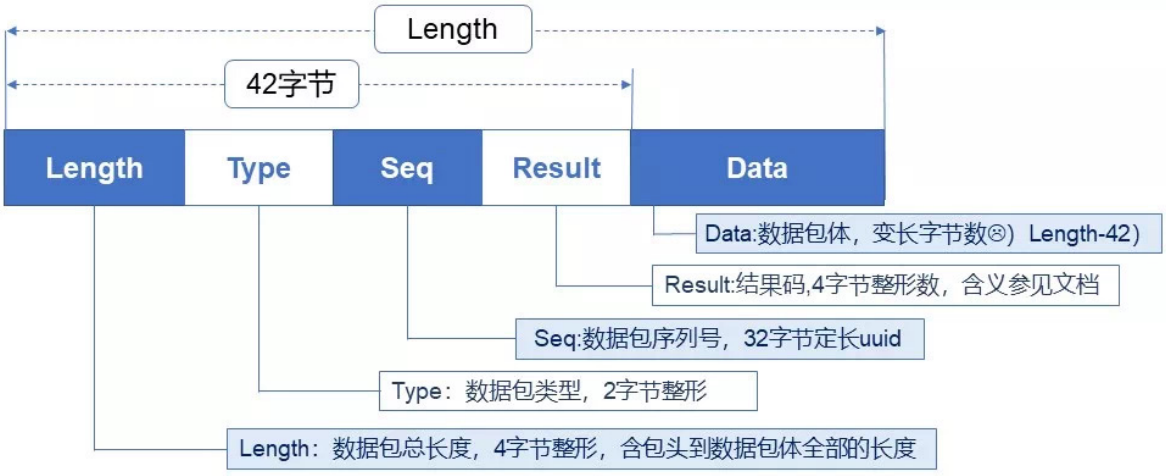
JSON RPC的格式相当简单通用，各种语言库都内置了JSON编解码以及HTTP请求协议实现，一般不需要自行开发，甚至可以采用curl、telnet等工具进行收发，如：

```
// 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"    }
```

Channel协议是FISCO BCOS独有的协议，Channel协议的特点是安全高效，支持双向实时通信，可用于远程调用乃至公网的通信。

如果使用Channel长连接方式，则需要从区块链节点上获取SDK证书，放置到SDK项目的对应路径下，证书详见文末。

这个协议的数据包格式示意图如下，是一种TLV(Tag/Length/Value)风格的扩展实现：



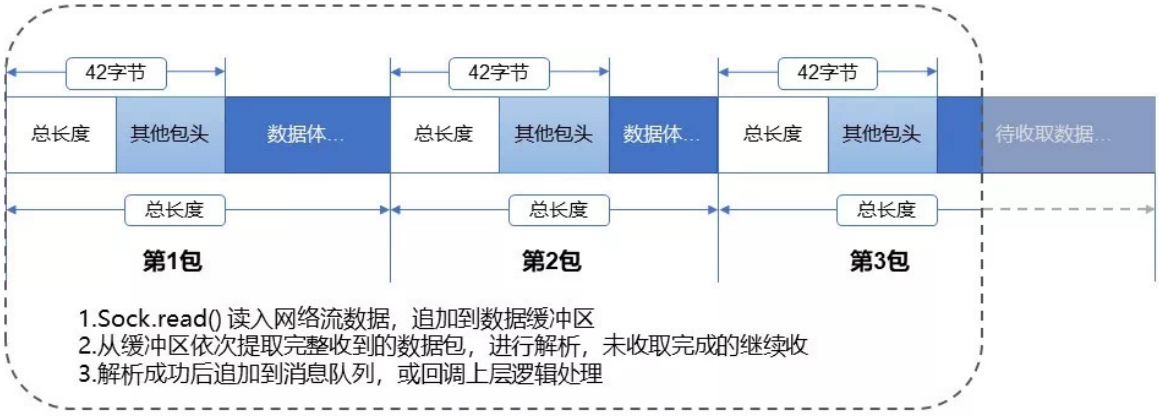
格式说明：

1. 所有整形数编码都是网络序，大端（Big endian）；
2. Length实际上包含了从第一个字段到最后一个字段（data）的整个数据包的长度；

3. 包头（Length+Type+Seq+Result）为定长，为（4+2+32+4) = 42字节；
4. 数据体的实际长度根据具体内容而变，字节数为Length-42字节。

Channel长连接通信和数据收发的要点如下：

- 1. 采用TLSv1.2安全传输，SDK和节点之间需要加载证书，用证书握手、验证后才能建立长连接。
- 1. 长连接用心跳包维护，需要定期发起心跳包。
- 1. 数据按包为单位，编码成流数据传输，那么在收发数据时，需要持续从socket流里获取数据，按照数据包的格式，判断长度是否合法，数据是否收全，是否能正确的解析，对“部分收取”的数据，要保留在接受缓冲区里，待收取完成后再进行解析，不能丢弃，否则可能导致解析错误。



- 1. Channel协议支持双向通信，SDK可以主动请求节点，节点也可能往SDK推送消息，如区块链系统通知、数据更新通知、AMOP跨机构消息等。
- 1. 设计异步的、队列化、回调式消息处理机制，根据消息的序列号、指令类型、状态码等维度，正确处理消息。Python-SDK用了多线程以及Promise库，以尽量高速优雅地处理各种消息。

对socke流数据编程有一定经验的开发者，理解这个协议和实现它并不会很难。对Channel协议实现，数据包解析参见client/channelpack.py，通信和数据收发参见client/channelhandler.py。

总结

Python-SDK的开发始于今年6月中旬，写出第一个可用版本只花了一个星期，然后雕琢用户交互细节，以及进行代码优化、文档完善，并进行多轮测试保证质量，团队其他同学实现Nodejs版本SDK的用时也差不多。 总的来说，在有一些基础代码参考的前提下，开发一个FISCO BCOS 特定语言版本SDK，还是挺敏捷写意的事情，一点儿也不难，Just for fun。 在各语言版本SDK开发和迭代过程中，FISCO BCOS团队和社区开发者一直保持沟通交流，纳入优质pull request，在体验中持续优化。

欢迎社区开发者根据自身使用场景的实际情况，继续完善现有SDK，或贡献更多语言类型的FISCO BCOS SDK，帮助更多开发者顺畅地走在区块链之路上。 最后，感谢杰哥、安总、小白、wheat等同学，以及多位社区开发者对Python-SDK的重要贡献。

参考资料

- FISCO BCOS官方控制台
- Python-SDK
- Nodejs-SDK

- FISCO BCOS安装
- 合约开发教程
- WeBASE
- ABI
- RLP
- 交易数据结构
- RPC原理
- Channel协议定义
- SDK证书
- AMOP跨机构

手把手教你在Eclipse中使用JavaSDK

作者：王章 | FISCO BCOS 核心开发者

Eclipse是当前主流的Java IDE之一，本文将进行上手实操，指导如何在Eclipse中创建FISCO BCOS的JavaSDK应用。

本文首先介绍如何在Eclipse创建新的工程，在工程中引入JavaSDK依赖，通过配置与区块链进行交互，最后通过获取区块链的块高验证创建的项目。同时，本文也会介绍实际开发中更加便捷的使用方式，将已经提供的示例项目导入Eclipse。

注意：

文章项目以Gradle方式进行构建，请确认以下环境正确安装：1、Java 1.8或者以上版本；2、

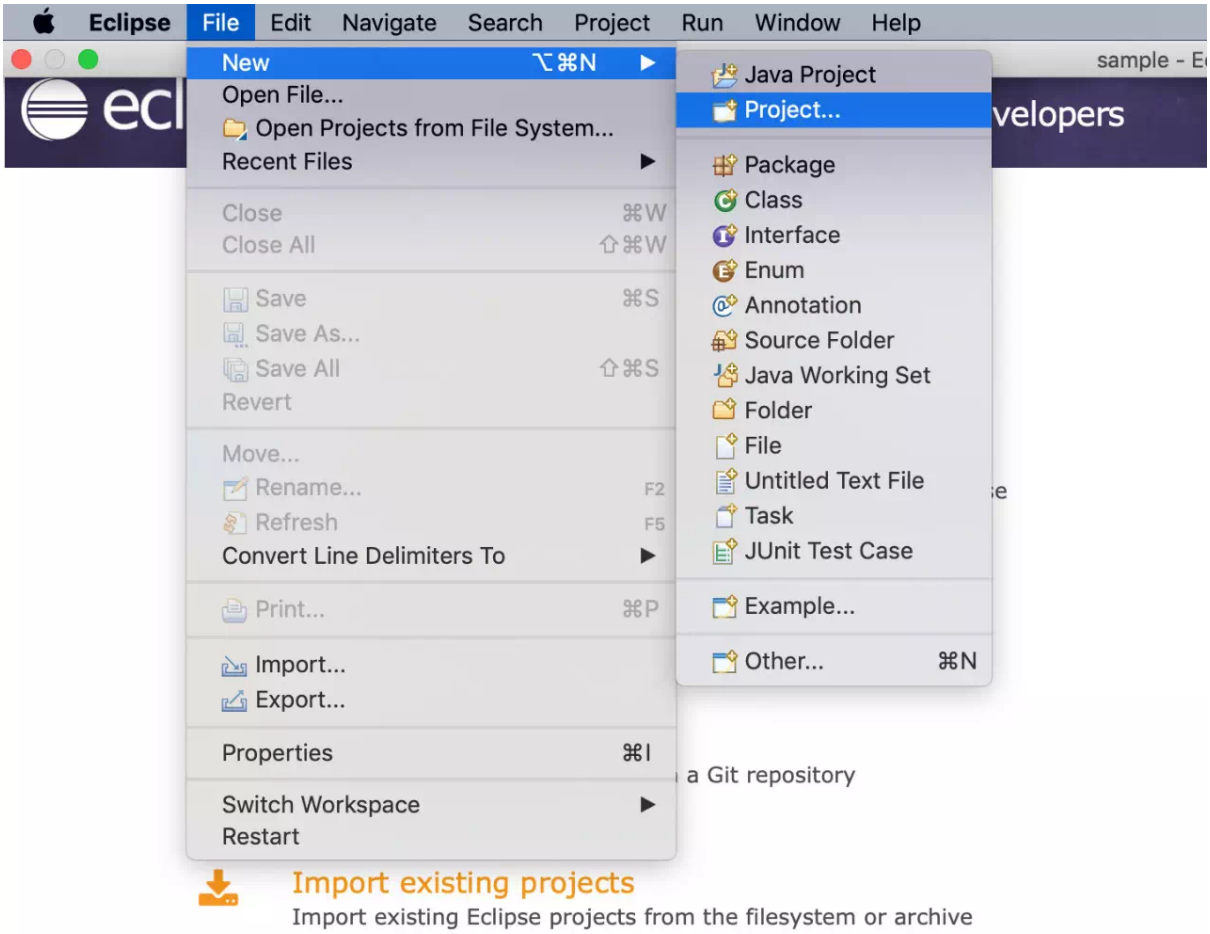
前置条件：

参考安装文档：完成FISCO BCOS区块链的搭建，功能测试需要与区块链节点进行交互。

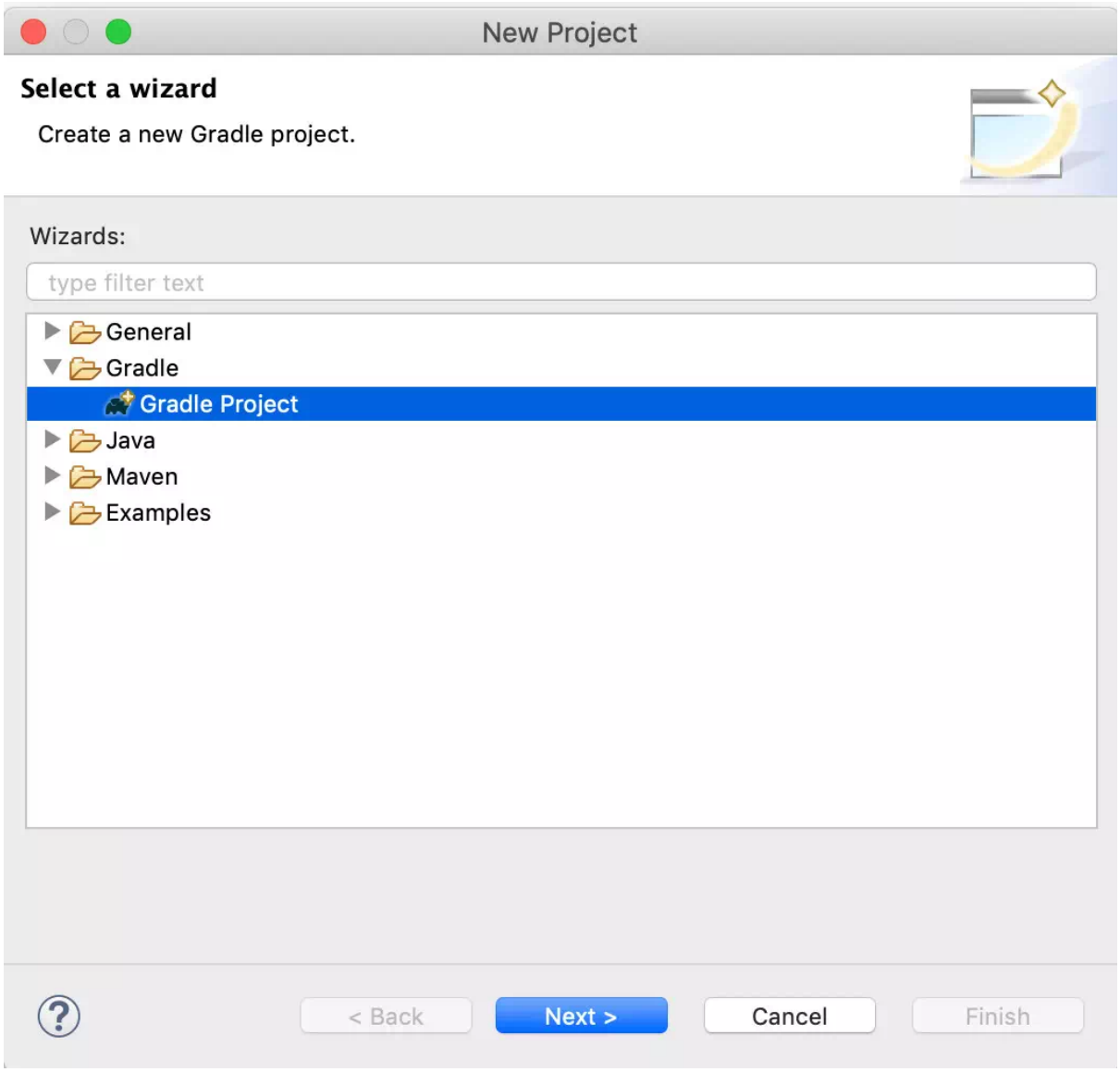
Eclipse 创建新工程

创建Gradle工程

打开Eclipse，右键选择New下Project选项，如下图所示：



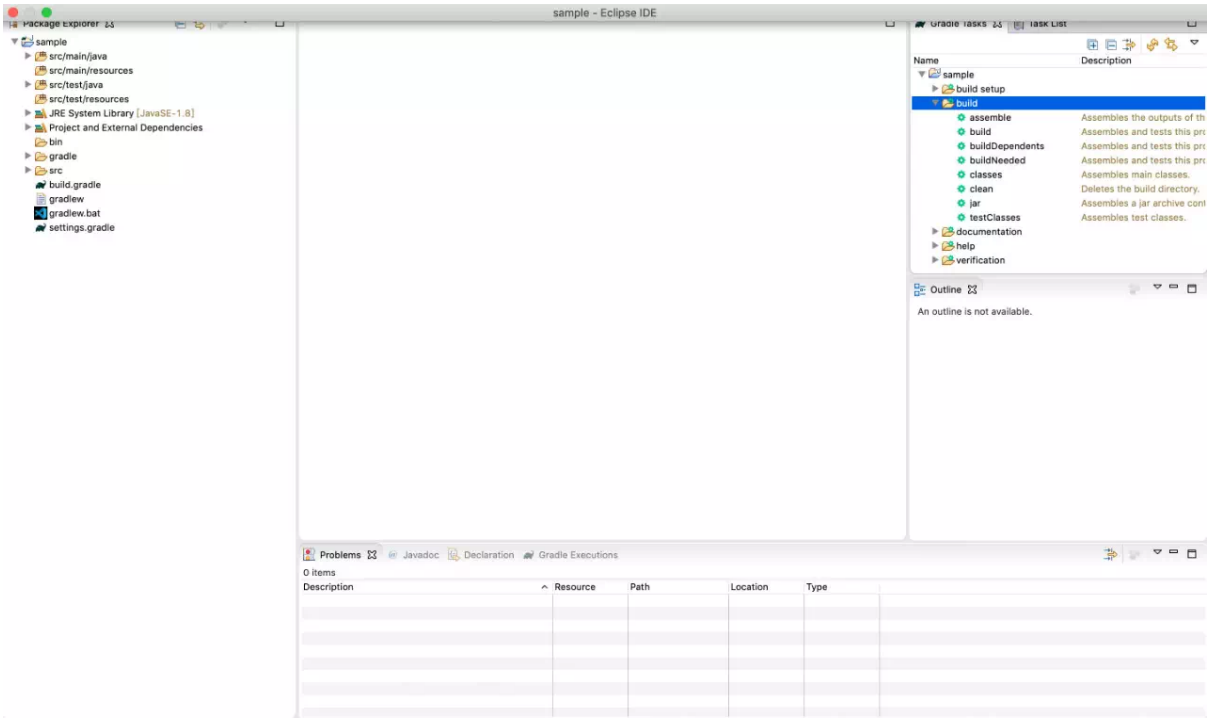
在弹出的新建New Project界面，选择Gradle => Gradle Project，点击Next：



再次点击Next，在接下来的界面输入项目名称，这里项目名为sample。

docs/articles/4_tools/44_sdk/../../../../images/articles/use_javasdk_in_eclipse/IMG_563

点击Finish，然后等待项目初始化。



项目初始化之后，一个新的Gradle工程已经创建完成。接下来，我们需要修改项目的build.gradle文件，引入JavaSDK的maven库以及依赖版本。

项目引入JavaSDK依赖

打开build.gradle文件，在repositories添加需要的maven库:

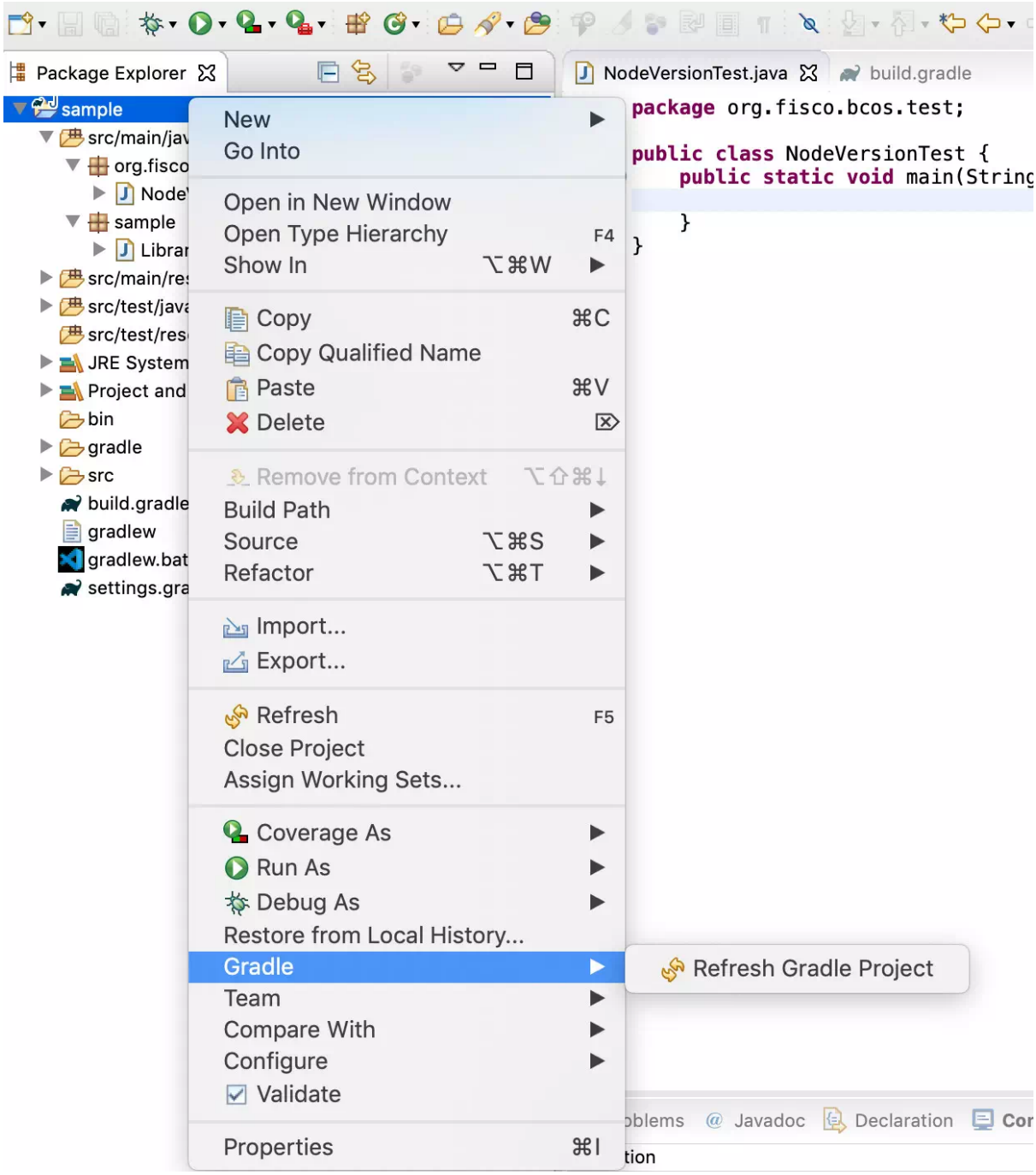
```
repositories   {
    // 其他maven库
    // solidity编译器maven库
    maven { url"https://dl.bintray.com/ethereum/maven/" }
    // JavaSDK snapshot版本 maven库
    maven{ url "https://oss.sonatype.org/content/repositories/snapshots"}
    // JavaSDK maven库
    maven { url"http://maven.aliyun.com/nexus/content/groups/public/" }
     // 通用maven库
     mavenCentral()
    }
```

在dependencies添加JavaSDK的依赖版本：

```
dependencies {
        // 其他依赖

        // 日志依赖
        compile ('org.slf4j:slf4j-log4j12:1.7.25')
        // JavaSDK依赖, 2.1.0版本
        compile ("org.fisco-bcos:web3sdk:2.1.0")
    }
```

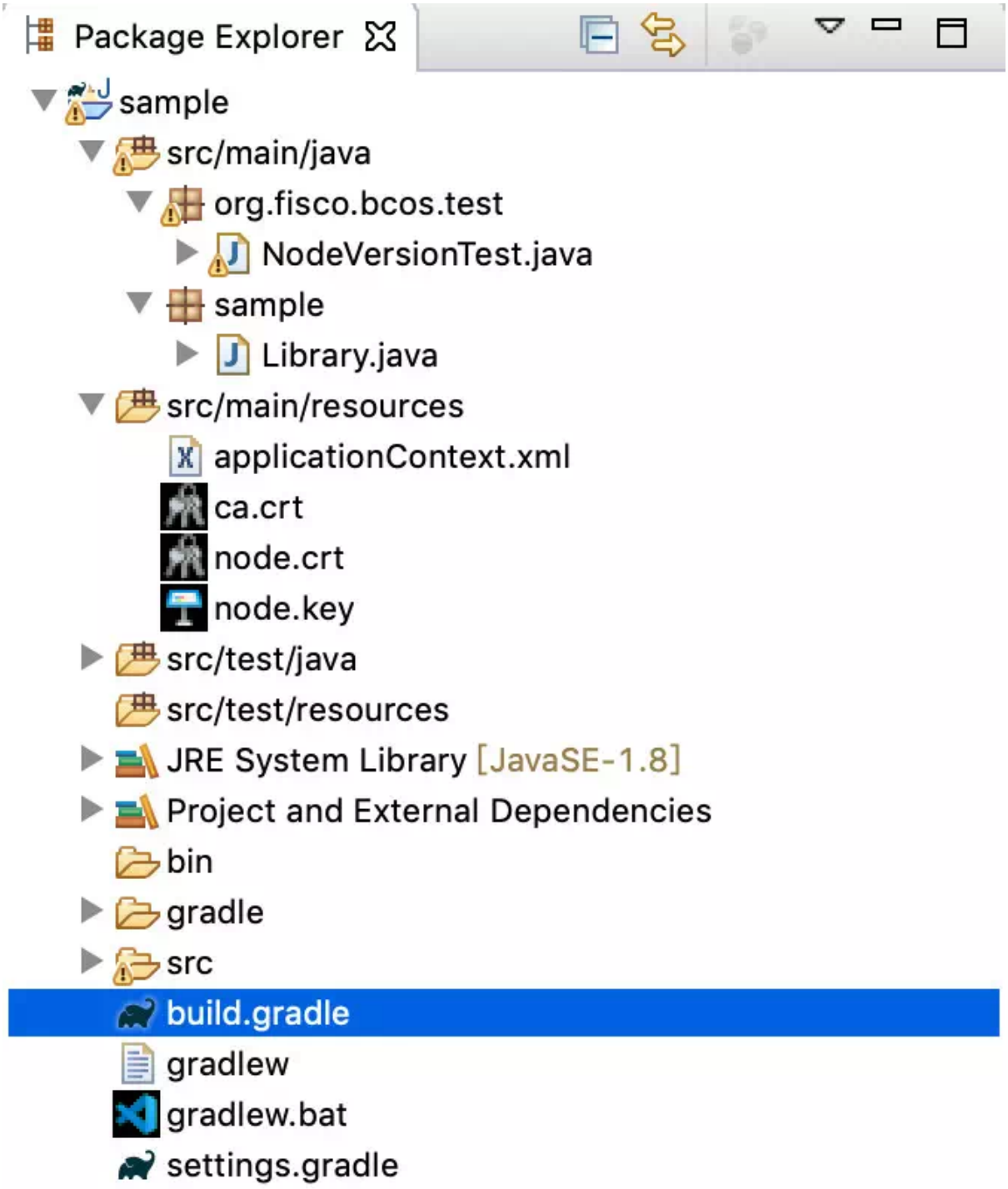
然后右键点击项目名：Gradle=> Refresh Gradle Project刷新项目。



注意：刷新项目可能需要从远程maven库下载项目依赖jar的包，请确保网络畅通，下载过程会需要一些时间。

证书与配置文件

JavaSDK服务运行需要依赖applicationContext.xml配置文件以及证书。查看配置参考。将配置文件applicationContext.xml，证书文件ca.crt sdk.crt sdk.key拷贝到项目src/main/resources目录下。

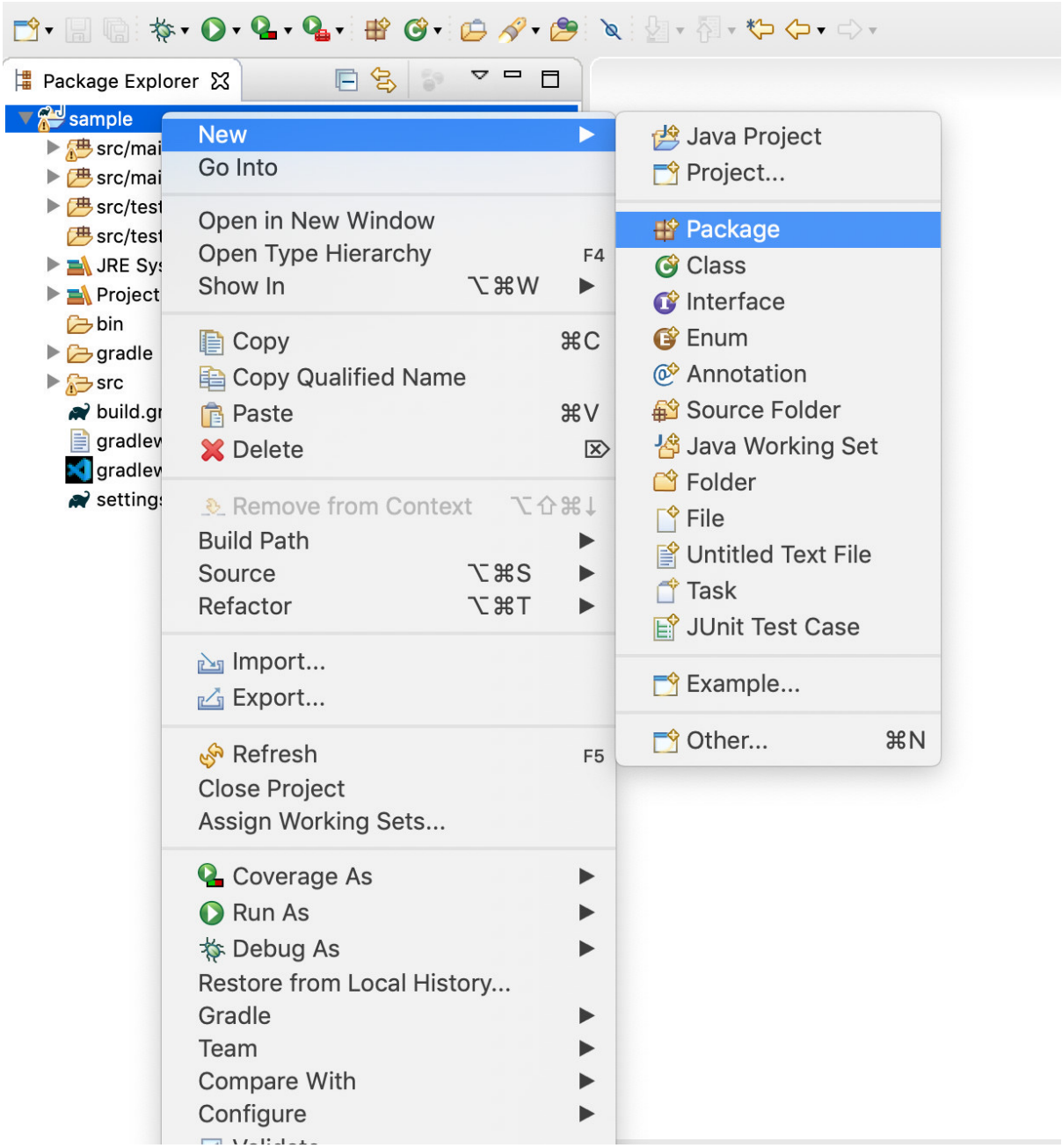


创建运行测试类

到此，我们已经完成了新项目的创建，引入并完成了JavaSDK的配置，现在我们来测试项目的功能。

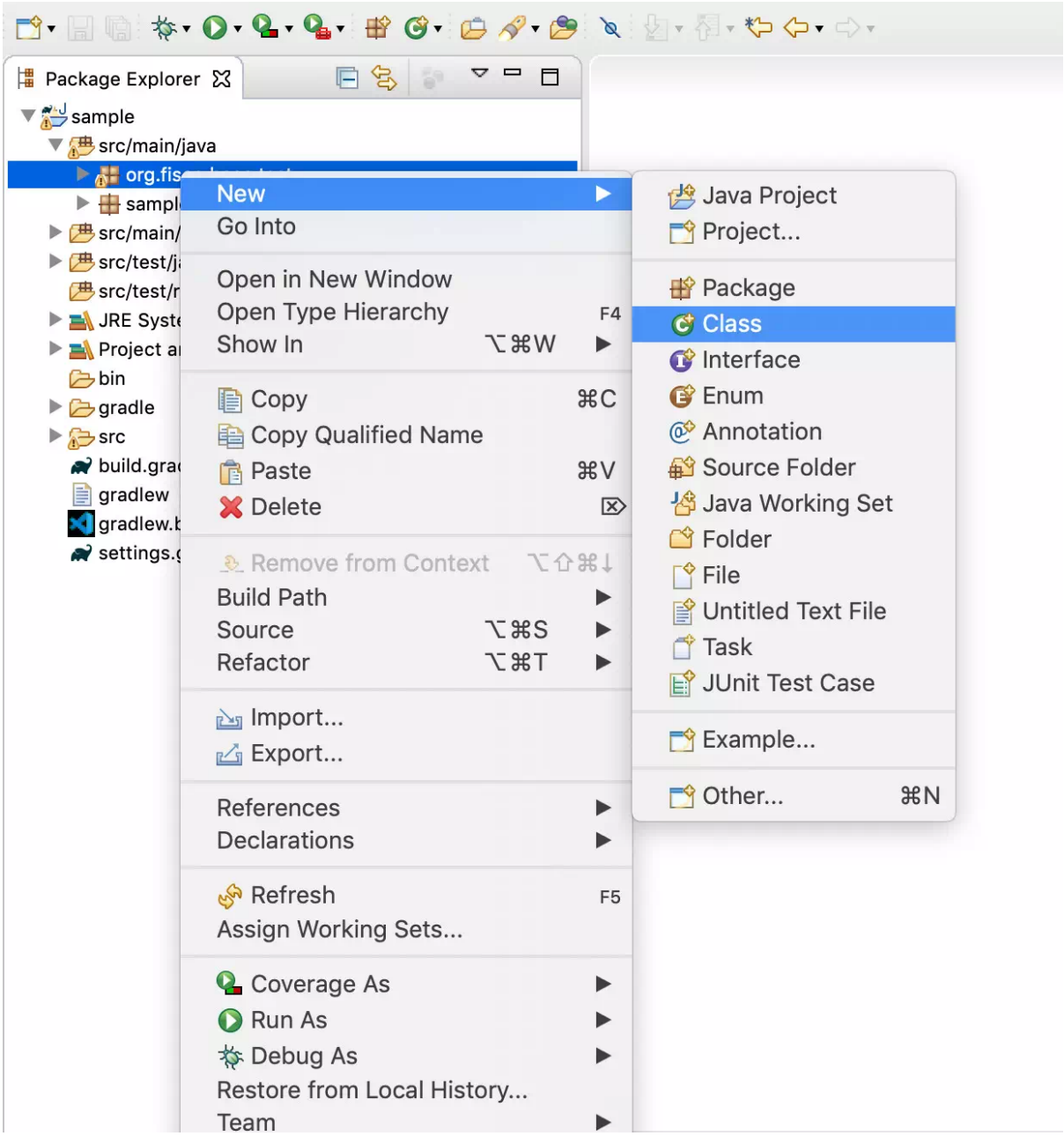
创建类包package

右键点击项目名称选择New => Package然后输入包名，这里使用org.fisco.bcos.test。

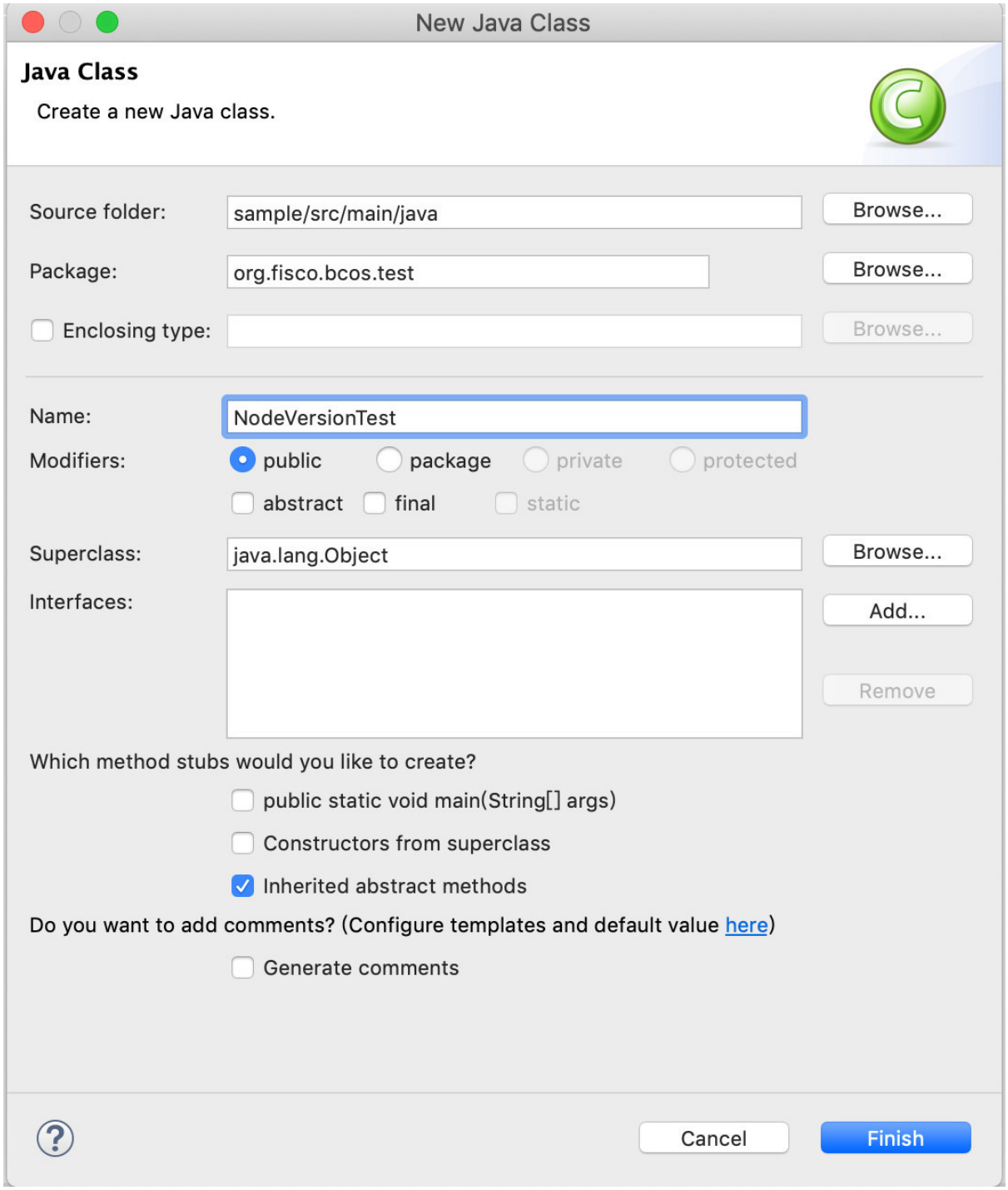


创建测试类

右键点击类包名选择 'New => Class'。



输入类名，点击Finish，这里类名使用NodeVersionTest。



测试类的功能为调用getBlockNumber接口获取当前的块高：

```
package org.fisco.bcos.test;

import org.fisco.bcos.channel.client.Service;
import org.fisco.bcos.web3j.protocol.Web3j;
import org.fisco.bcos.web3j.protocol.channel.ChannelEthereumService;
import org.fisco.bcos.web3j.protocol.core.methods.response.BlockNumber;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class NodeVersionTest {
 public static void main(String[] args) throws Exception {
```

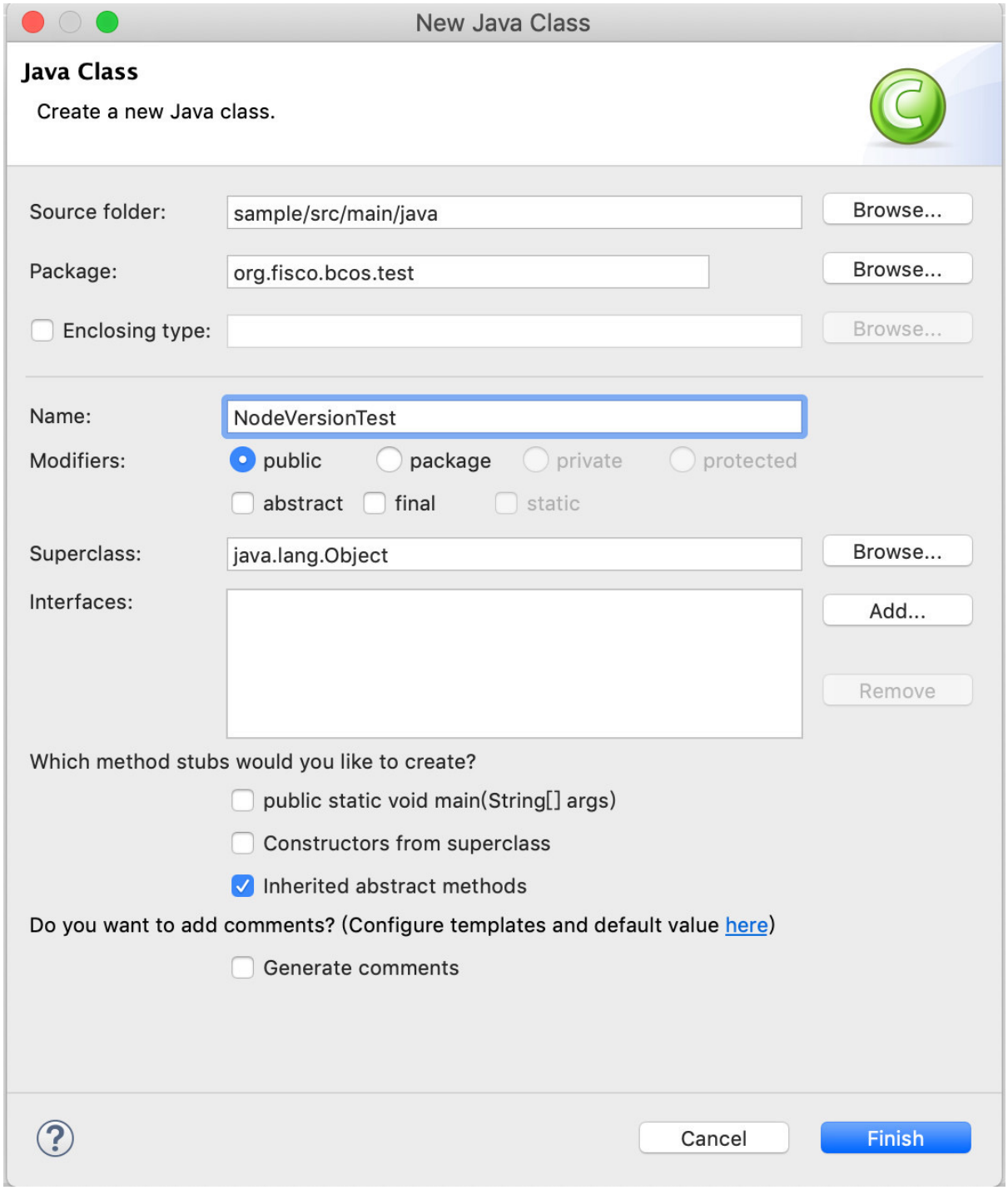
(continues on next page)

(续上页)

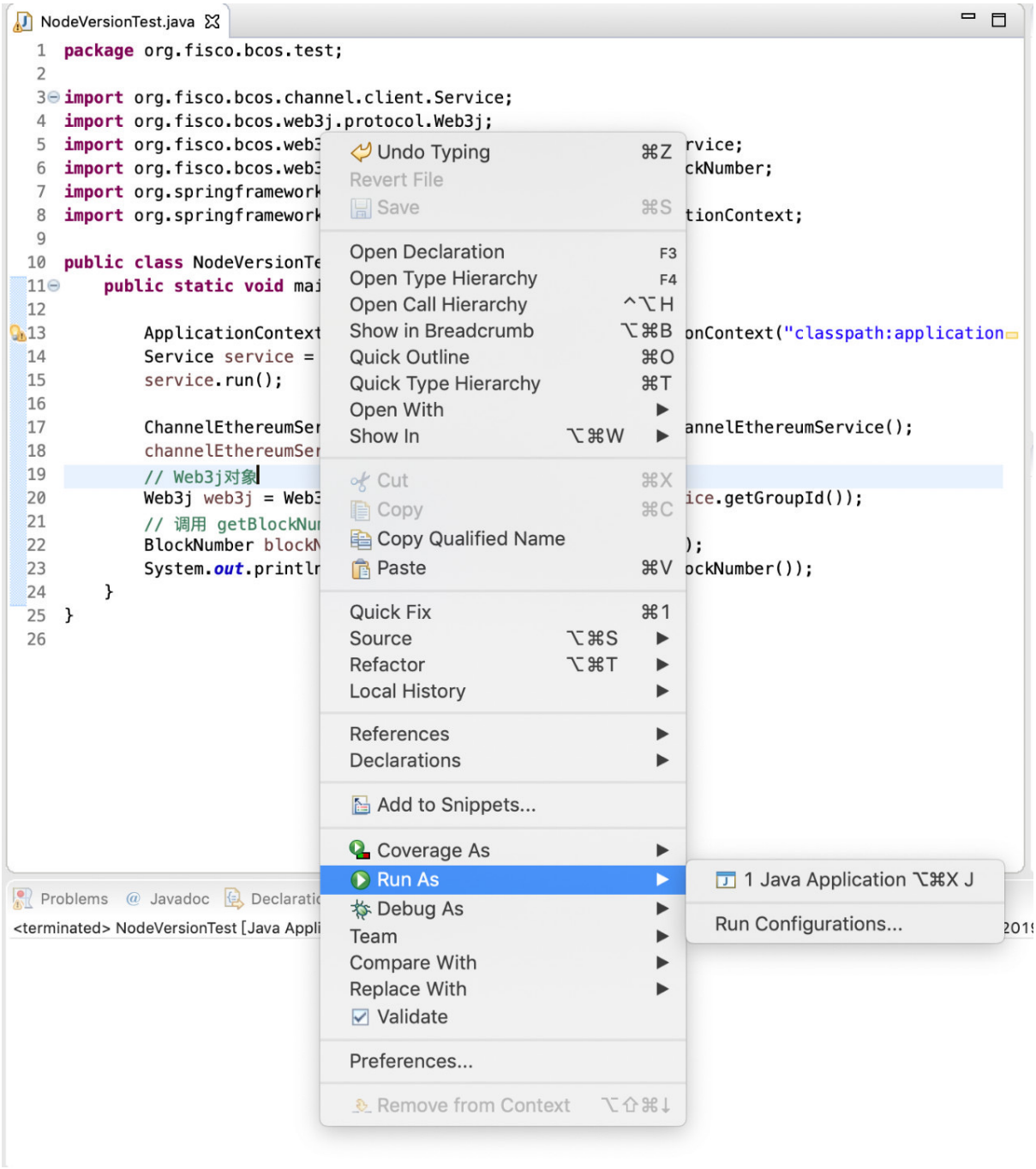
```
  ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
  Service service = context.getBean(Service.class);
  service.run();

  ChannelEthereumService channelEthereumService = new ChannelEthereumService();
  channelEthereumService.setChannelService(service);
  // Web3j对象
  Web3j web3j = Web3j.build(channelEthereumService, service.getGroupId());
  // 调用 getBlockNumber 接口获取
  BlockNumber blockNumber = web3j.getBlockNumber().send();
  System.out.println("BlockNumber: " + blockNumber.getBlockNumber());
  System.exit(-1);
 }
}
```

右键点击NodeVersionTest.java文件选择Run As => Java Application运行测试类。


New Java Class
Java Class
Create a new Java class.
Source folder:
sample/src/main/java
Browse...
Package:
org.fisco.bcos.test
Browse...
Enclosing type:
Browse...
Name:
NodeVersionTest
Modifiers:
public
package
private
protected
abstract
final
static
Superclass:
java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to create?
public static void main(String[] args)
Constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Cancel
Finish

运行结果:



Eclipse导入项目

上面流程可以看出创建新项目的过程需要比较多的配置流程，为了便于用户使用，我们提供已有的示例项目asset-app，用户可以将其快速导入Eclipse，在示例基础上快速修改开发自己的应用。请参考asset-app项目详情。

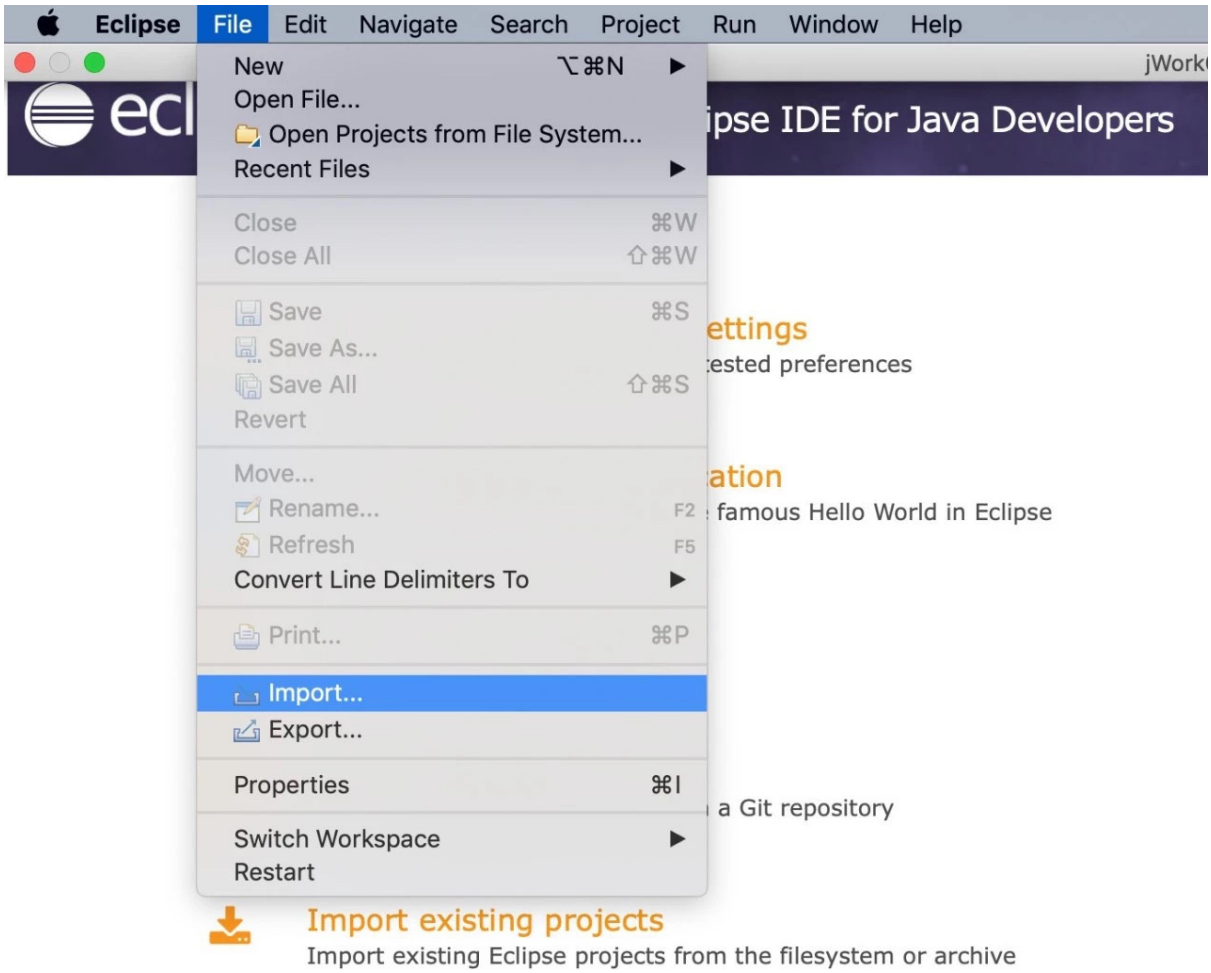
下载asset-app项目工程

下载asset-app，下载之后本地解压至指定路径。

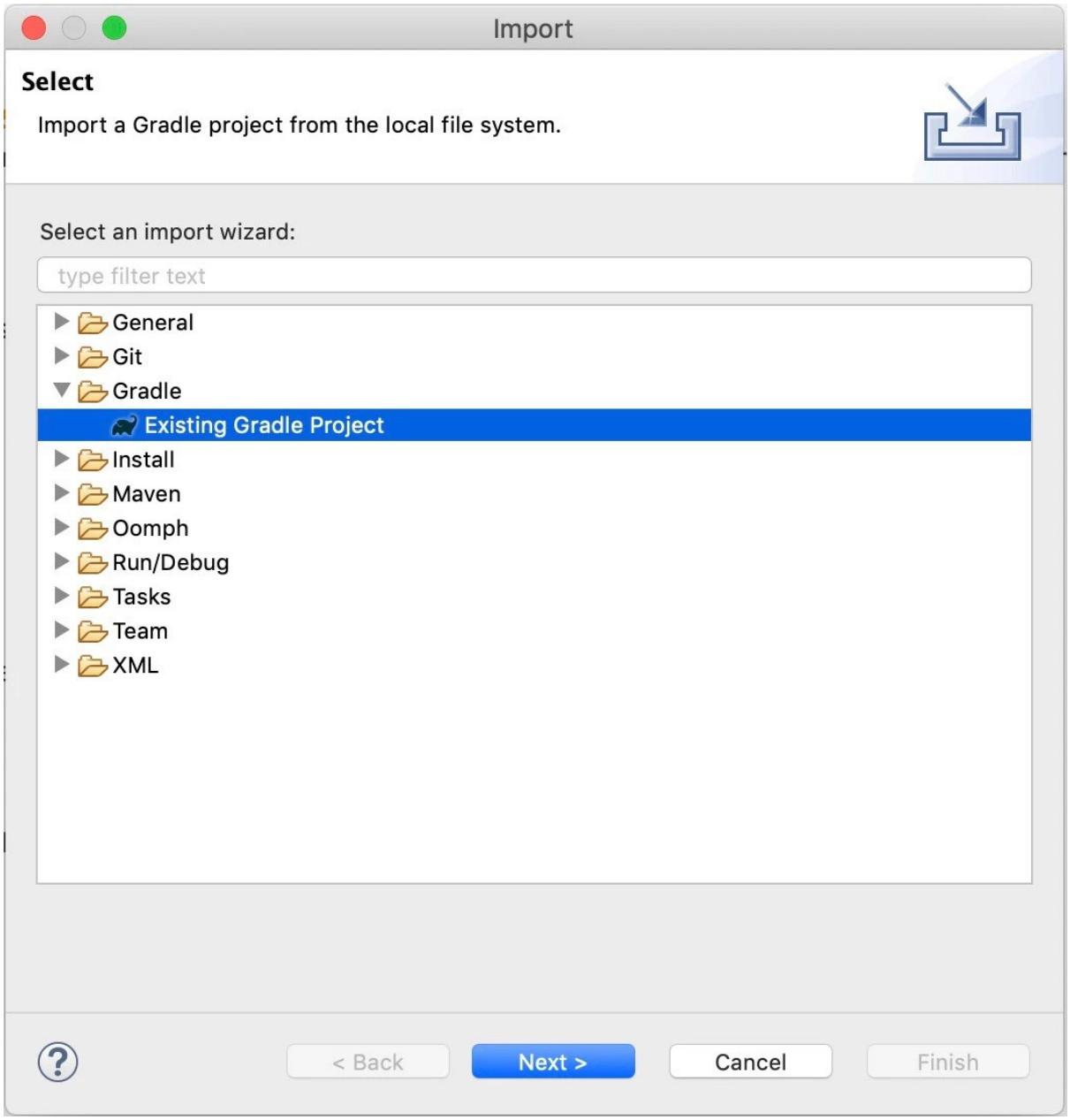
导入工程

打开Eclipse选择: File => Import => Gradle => Existing Gradle Project。

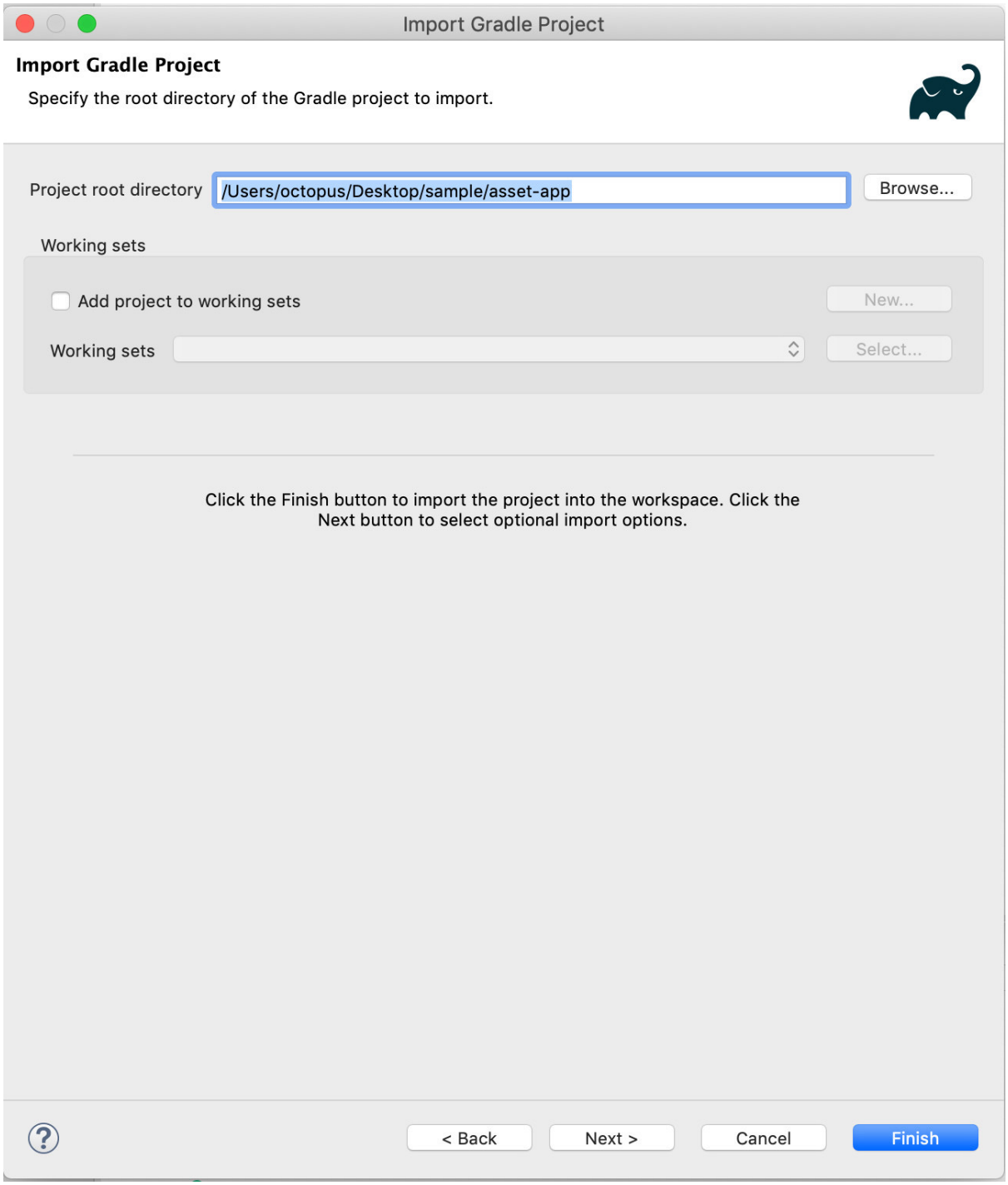
```
log4j:WARN No appenders could be found for logger (org.springframework.core.env.StandardEnvironment).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
BlockNumber: 1157
```



点击Next，选择asset-app路径，点击Finish等待项目加载。



项目加载完成之后，右键点击项目名：Gradle=> Refresh Gradle Project刷新项目。



ok! asset-app项目已经被正常加载。

一文快速入门FISCO BCOS Node.js SDK

作者：李陈希 | FISCO BCOS 核心开发者

SDK一般是指为了方便开发者为系统开发应用而提供的API、开发辅助工具以及文档的集合。面对不同的开发者群体，FISCO BCOS已经相继推出了面向企业级应用开发者的Java SDK（功能丰富、稳定），以及面向个人开发者的Python SDK（上手迅速、轻便）。在实际推广过程中，我们发现有部分开发者习惯使用JavaScript构建应用前端，倘若此时能有一个支持使用JavaScript进行后端开发的SDK，那么前后端的语言壁垒能够进一步被打破——开发者只需了解JavaScript，便能完成整个FISCO BCOS应用程序的前后端开发。在此背景下，FISCO BCOS Node.js SDK诞生了。本文将会介绍Node.js SDK的设计及使用。

一、SDK的设计

Node.js SDK遵循分层设计原则，层与层之间界限清晰、功能明确，下图为Node.js SDK的架构示意图：

图中自底向上依次为：

- **基础层**：提供网络通信、交易构造及签名、合约编译等基础功能；
- **API层**：基于基础层提供的功能，对FISCO BCOS常用功能进行了进一步地封装，暴露出API用于给上层的应用层调用。这些API覆盖基本的JSON RPC功能（查询链状态、部署合约等）及预编译合约功能（权限管理、CNS服务等）；
- **应用层**：使用Node.js SDK进行二次开发的应用程序均属于此层，如Node.js SDK自带的CLI（Command-Line Interface，命令行界面）链管理工具。

二、SDK的安装

2.1 环境准备

Node.js SDK依赖下列软件：

- 8.10.0或以上的版本的Node.js，5.6.0或以上的版本的NPM；
- Python2、g++及make。Solidity编译器solc需要编译后才能使用，编译所需基础软件需要用户提供，其中Python2用于运行构建工具node-gyp，g++和make则是用于编译solc。对于没有g++和make的Windows用户，可以安装 windows-build-tools再进行构建。

2.2 安装SDK

- 从https://github.com/FISCO-BCOS/nodejs-sdk下载sdk；
- 进入nodejs-sdk目录；
- 执行npm i。Node.js SDK使用lerna管理依赖项，此步用于安装lerna；
- 执行npm repoclean。初次安装时无需执行该命令，但如果出现了安装到一半被打断等情况，建议先执行该命令以清除所有依赖项；
- 执行npm run bootstrap，此命令会安装SDK的所有依赖项。

上述所有命令执行完后，API层的目录结构如下图所示：

对于应用开发者，可以重点关注 nodejs-sdk/packages/api/web3j 及 nodejs-sdk/packages/api/precompiled目录，SDK提供的所有API均位于这两个目录下的模块中，应用开发过程中通过require语句导入相应模块即可。

三、使用SDK进行开发

3.1 初始化

使用Node.js SDK的所有功能前，必须对SDK进行配置，配置以.json格式配置文件的形式提供。主要包括以下配置项：

- **privateKey**：FISCO BCOS基于公私钥体系，每个账户（公钥）均有一个对应的私钥，SDK需要使用该私钥对交易进行签名；
- **nodes**：SDK连接的节点，可以有多个，当节点数大于1时，SDK的每次请求都会随机从nodes调出一个节点发送；
- **authentication**：SDK使用Channel协议与节点进行通信，Channel使用SSL安全传输协议传输数据，在双方建立通信前，需要进行必要的身份验证。因此这个配置项中需要指明SDK私钥文件、证书文件、CA根证书文件的路径，这些文件通常是在生成链的阶段自动生成；
- **groupID**：FISCO BCOS使用多群组架构，同一个节点可以隶属于多个群组，因此需要指明SDK需要连接的链的群组ID；
- **timeout**：受限于网络环境，SDK的请求有可能超时，因此可能会导致调用SDK接口的程序陷入无尽等待，因此需要指定一个超时时间，SDK在请求超时后强制返回并向调用者返回错误。

开发者需要在初始化阶段将配置文件载入Configuration对象中，Configuration对象是全局唯一的且被所有模块共享。

3.2 调用示例

以一个简单的获取当前块高的程序为例，调用步骤如下：

1、初始化Configuration对象，将配置文件路径传入Configuration对象的setConfig函数：

```
const Configuration = require('./nodejs-sdk/packages/api/common/configuration').
↪Configuration;
Configuration.setConfig(path.join(__dirname, args.config));
```

2、获取当前块高的API位于Web3jService中，构造一个该对象：

```
const Web3jService = require('./nodejs-sdk/packages/api').Web3jService;
let web3jService = new Web3jService();
```

3、调用Web3jService的getBlockNumber接口，获取返回值并在控制台中输出：

```
web3jService.getBlockNumber().then(blockNumber => {
        console.log(blockNumber)
    });
```

需要注意的是，上述代码中使用了Promise.protoype.then方法。Promise正如其名字所示，封装了一个异步的操作，并且“承诺”在这个操作结束后，一定会调用用户在then或者catch中指定的回调函数。

由于Node.js天然支持异步的特性，Promise的概念在Node.js SDK处处存在（细心的读者可能已经注意到，Node.js SDK基础层中负责Channel通信的模块叫做channelPromise）。Node.js SDK的API调用约定是：所有的API调用时均返回Promise对象，开发者需要使用await或then...catch...方法才获取到调用结果。因此开发者在调用API时需要小心，如果直接使用了API的返回值，会很容易导致bug。

四、使用CLI工具

Node.js SDK除了提供API外，还提供了一个小巧的CLI工具供用户直接在命令行中对链进行操作，同时CLI工具也是一个展示如何使用Node.js SDK进行二次开发的示例。CLI工具位于packages/cli目

录下，若需要使用则需要进入该目录并执行./cli.js脚本，使用之前同样需要进行配置，配置文件位于packages/cli/conf/config.json文件中。以下给出了几个使用示例：

1、查看所连节点的版本：

2、获取当前块高：

3、部署合约，部署之前合约需要放置于packages/cli/contracts目录中。

4、调用HelloWorld合约的get方法：

五、Node.js SDK的未来需要你

当前，Node.js SDK还在成长，在某些地方仍然需要进一步打磨，比如需要CLI工具能够解析SQL语句，或者SDK的性能需要优化……秉承开源的精神，我们相信社区的能量能够将Node.js SDK 变得更加方便易用，欢迎广大开发者踊跃在issue或PR中贡献自己的idea和力量！

Python SDK的前世今生

作者：陈宇杰 | FISCO BCOS 核心开发者

今年6月，在已有Java SDK的基础上，FISCO BCOS相继推出了Python SDK和Go SDK，不少开发者表示，这两个组件极大降低了开发区块链应用的难度，让更多开发者可以参与到FISCO BCOS的开源共建中。今天，本文就来科普一下Python SDK的前世今生，看看这个简单易上手又发挥着极大作用的组件是如何实现功能的。

Why: 为什么要做Python SDK

众所周知，Python语言简单明了，几乎接近自然语言，可以快速上手；Java语言虽然功能强大，但语法、组件相对复杂，一般会在企业级应用中使用。为了融合两者优势，FISCO BCOS首席架构师张开翔经过两周集中开发，成就了Python SDK的第一个版本。这个版本在今年7月份社区的黑客松比赛中发

挥了极大作用，帮助参赛者更好聚焦在开发项目本身。在高强度的黑客松比赛中，参赛者只有36小时去完成区块链相关项目。在和选手交流中，我们发现大多数极客开发者均是从比特币、以太坊等公链开始入门，对Java语言不是很熟悉，反而精通Python、Go、Node.JS。当时团队主推Java SDK、控制台和WeBASE等基于Java的区块链开发工具，可是短短36小时，要求不熟悉Java的极客开发者基于这些工具完成开发，确实有些强人所难。在这种情况下，已经初见雏形Python SDK就成了开发者的救命良药。最终，很多参赛项目就是基于Python SDK完成的。

What: Python SDK实现了哪些功能

Python SDK经过不同版本的演进优化，逐步实现丰富的功能。

基础版本Python SDK

第一版的Python SDK主要基于Windows平台开发，实现了访问FISCO BCOS链RPC服务的接口，通过Python SDK接口，用户可以获取区块链的基本信息(如区块高度、区块、交易、交易回执等)，还可以部署和调用合约。但该版本的SDK不支持Channel协议，因此无法保证SDK与节点间通信安全；且Python SDK无法接收FISCO BCOS区块链节点推送的回执、区块高度信息，因此发送交易后，必须轮询节点获取最新信息。

支持Channel协议的Python SDK

为了提升SDK与节点通信的安全性，Python SDK在基础版本上，实现了Channel协议。支持Channel协议后的SDK在发送交易后，不需轮询节点获取交易执行结果，直接从节点接收执行结果的推送即可，这样SDK发送交易后，不会阻塞在状态轮询，实现了异步发送交易的功能。

支持Precompile合约调用

为了突破区块链虚拟机性能瓶颈，大幅度提升交易处理吞吐量，FISCO BCOS实现了预编译合约框架，现有的预编译合约包括命名服务合约、共识合约、系统配置合约、权限控制合约、CRUD等，为方便用户使用这些预编译合约，Python SDK提供了访问Precompile合约的接口。

Python SDK控制台

以上功能ready后，Python SDK已经可以实现与FISCO BCOS链的所有交互，但用户仍无法通过Python SDK直观感受FISCO BCOS链，于是在实现以上所有接口的基础上，Python SDK集成了控制台console，console通过命令去调用以上接口，用户可通过命令行获取区块链节点信息、部署和调用合约。

Python SDK多平台支持

众所周知，Python是跨平台语言，支持Windows、MacOS、 CentOS、Ubuntu等各种开发平台。而Python SDK是基于Windows系统开发的基础版本，因此部署在MacOS、CentOS系统还有一些问题，比如说：各平台依赖包要求不一致、符合Python SDK版本的Python安装方式有差异、合约编译器安装方式不相同等。

以上这些问题，会使部分开发者的精力在安装基本环境上消耗殆尽，为了解决这个问题，Python SDK提供了部署脚本`init_env.sh`，用户可使用该脚本，一键初始化Python SDK环境。

脚本`init_env.sh`的主要功能包括：

- 若运行Python SDK的系统Python版本小于3.6.0，安装3.7.3版本的Python虚拟环境python-sdk；
- 安装v0.4.25版本的solidity编译器。

How: Python SDK如何实现

上节介绍了Python SDK的主要功能，这节就来说说Python SDK的实现。

罗马不是一日建成的，Python SDK在开发过程中参考了以太坊客户端部分模块，包括：

- ABI编解码模块
- RLP编解码模块
- 账户和密钥生成模块

实现Channel协议

Python SDK在以上模块的基础上，实现了支持FISCO BCOS Channel协议的RPC接口和发送交易接口。Channel消息包类型定义实现于模块`client.channelpack.ChannelPack`，主要消息包类型包括:

- 0x12: RPC类型消息
- 0x13: 节点心跳消息包
- 0x30: AMOP请求包
- 0x31: 失败的AMOP消息的包体
- 0x32: SDK topics
- 0x35: AMOP消息包包体
- 0x1000: 交易上链通知(包含交易回执)
- 0x1001: 区块高度推送包

Channel协议的实现位于`client.channel.handler.ChannelHandler`模块，为了支持异步发送交易，Python SDK引入了`pymitter`和`Promise`组件，用于管理异步事件，该模块发送交易并获取交易回执的主要工作流程如下：

- 1. Python SDK调用`sendRawTransactionAndGetReceipt`接口，对交易进行编码和签名，并将编码好的数据作为`ChannelHandler`模块发送socket请求包的参数；
- 1. ChannelHandler接收到1的数据后，将数据包的uuid注册到异步事件队列后，将消息包放入发送缓冲区发送并发送；
- 1. 节点处理完交易后，向SDK发送类型为0x1001的交易上链通知；
- 1. SDK端ChannelHandler收到消息类型为0x1001的回包后，取出消息包的uuid，并触发该uuid对应的异步事件，将交易执行结果返回给上层应用。

实现控制台

Python SDK控制台实现位于`console`模块，主要调用了一下模块接口实现与区块链的交互：

- `client.bcosclient`：Python SDK基本接口类，提供了访问FISCO BCOS RPC服务的接口、部署和调用合约的接口；
- `console_utils.precompile`：提供了访问Precompile预编译合约的功能；
- `console_utils.rpc_console`：封装了`client.bcosclient`，实现命令行调用RPC服务功能。

Python SDK使用展示

了解了Python SDK的由来、功能和实现，下面从控制台角度直观感受下Python SDK:

获取节点版本

命令行终端输入./console.py getNodeVersion可获取节点版本：

部署HelloWorld合约

Python SDK内置HelloWorld合约，命令行终端输入./console.py deploy HelloWorld可部署HelloWorld合约，若用户需要部署自编写合约，则需将该合约放置于contracts子目录，并使用命令./console.py deploy 【合约名】来部署合约。下图是HelloWorld合约部署输出，从中可看出，合约地址为: 0xbbe16a7054c0f1d3b71f4efdb51b9e40974ad651

调用HelloWorld合约

Python SDK控制台通过sendtx子命令发送交易，使用call子命令调用合约常量接口。HelloWorld合约代码如下，主要包括set和get两个接口，前者用于设置合约局部变量name，后者是常量接口，获取局部变量name的当前值。

```
pragma solidity^0.4.24;
contract HelloWorld{
  string name;
  constructor() public{
      name ="Hello, World!";
  }
# get interface
   function get() constant public returns(string){
      return name;
  }
# set interface
   function set(string n) public{
      name = n;
  }
}
```

调用HelloWorld set接口

由于set接口改变了合约状态，因此使用sendtx子命令调用，sendtx用法如下：

```
./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

使用`./console.py sendtx HelloWorld 0xbbe16a7054c0f1d3b71f4efdb51b9e40974ad651 set "Hello,Fisco"`命令可将`HelloWorld`合约的`name`成员变量设置为"Hello, Fisco"，输入如下：

调用HelloWorld get接口

`HelloWorld`的get是常量接口，因此使用`call`子命令调用，call用法如下：

```
./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

使用./console.py call HelloWorld 0xbbe16a7054c0f1d3b71f4efdb51b9e40974ad651 get获取HelloWorld合约name成员变量的最新值：

小结

本文介绍了Python SDK的前世今生，包括Python SDK的由来、功能和实现，可参考Python SDK详细使用方法。当然，目前Python SDK还处于不断完善过程，欢迎社区爱好者关注相关issue，为Python SDK的优化贡献宝贵PR。

记一次JavaSDK性能从8000提升至30000的过程

作者：李辉忠 | FISCO BCOS 高级架构师

缘起

FISCO BCOS在中国信通院可信区块链测评中达到2万+ TPS的交易处理能力，在同类产品中处于领先水平。此次测试标的是底层平台，目的是压测得出底层平台的性能上限，主要评估目标是底层平台的交易处理能力。

交易构造是由客户端（集成了SDK）完成，客户端通常可以非常容易实现平行扩展。SDK要完成交易构造的过程，实现交易组包、编码、签名、发送等一系列操作，这些过程本身是无状态的，客户端可以通过多线程的方式进行扩展，一个客户端性能达到瓶颈，可以增加更多客户端进行扩展。

虽然"堆机器"的平行扩展方式可以解决发送端的性能问题，但是机器本身就是珍贵的资源，进一步优化算法效率，提高资源利用率，将大有裨益。于是，我们打算先测试一下JavaSDK目前的性能，主要评测生成交易的性能。生成交易包括交易组包、参数编码、交易编码、交易签名等过程，其中，交易签名是最重要的环节，测试数据如下：

- 8核机器，测试本地生成50W笔交易的耗时
- 完全并行：每秒生成**8498笔**交易，平均每笔交易耗时0.12毫秒
- 完全串行：每秒生成1504笔交易，平均每笔交易耗时0.66毫秒

对比C++实现交易签名，这个性能实属不高。根据以往经验判断，这里大有优化空间啊！于是团队开始了一次JavaSDK的性能优化之路。

过程

关于性能优化，社区做过多次分享，可以补充阅读下列文章：

- FISCO BCOS共识优化之路
- 区块链的同步及其性能优化方法
- FISCO BCOS中交易池及其优化策略
- FISCO BCOS性能优化——工具篇
- FISCO BCOS的速度与激情：性能优化方案最全解密
- 让木桶没有短板，FISCO BCOS全面推进并行化改造

这些性能优化的过程中，感触最深的有两句话：『过早的优化是万恶之源』『没有任何证据支撑的优化是万恶之源』

优化要靠数据说话，而获取数据需要靠有效的分析工具，所以此次JavaSDK的性能优化，首要任务就是确定采用什么性能分析工具。

工具：发现java自带的分析工具挺管用

采用HPROF跑了一次，得到下面这样的数据报告：

数据显示，热点在很底层的库函数。这个结论并不符合预期，说明 SDK本身代码不是热点，性能优化比较难入手。

幸运的是，很快好消息就来了：采用java自带的另一个工具jvisualvm，较好地可视化输出性能分析数据，同时也很直观地显示出JavaSDK本身存在热点。

分析：找到一个“伪热点”

通过jvisualvm工具分析出来，最大的热点是在生成随机数，这让我等有些吃惊。

上面分析的是国密版本，对于非国密如何呢？跑了一遍，热点依旧可见在createTransaction，那就试试uuid吧。

但是，打脸，来得是那叫一个快，UUID的实现也是基于SecureRandom。

群聊出现好长时间的静默.....那就研究一下SecureRandom吧，看看为什么它会这么慢，也了解Java随机数的一些相关背景知识。终于，下面这个知识点让我们重拾兴奋。（https://zhuanlan.zhihu.com/p/72697237）

我那句“采用ThreadLocalRandom替换SecureRandom计算Nonce”还未敲字出去，新一轮打脸已经开始了。

这个方向的讨论为本次性能优化之旅豁开了另一扇思考之门~

讨论很有道理，再仔细看数据，从数据层面也印证了这个想法。

实验测试证明这是一个正确的方向，线程数降下来，这个热点就消失了。随机数那里出现热点是因为压测并发线程数开启太多，太多线程抢占资源导致随机数获取较慢。

再分析：找到令人震惊的热点

经过第一轮分析，算是找到一个“伪热点”，但性能提升依旧收效甚微。革命尚未成功，同志还需努力！降低线程数再跑一次性能分析，得到性能数据如下：

乍一看，热点分布又是很底层的基础库操作，焦虑又弥漫心头...伴着凌晨的静谧，群聊再一次陷入了无言的沉默.....直到，我又抛出一个疑问。

章鱼哥（王章）秒回，团队的激情苏醒了~为他这个回复速度，也为这个回复结果！

签名算法的实现，居然是先签名然后再做验签，而验签就是为了获得一个叫recoveryID的值（关于ECDSA的recovery原理，将在另一篇详细展开）。这里讲述一下我们为什么会兴奋。recoveryID设置的目的是为了给未来使用者可以从签名快速恢复出公钥，如果没有这个recoveryID，恢复公钥就需要遍历4种可能性，然而，这里的实现方式在生成签名的时候是通过遍历查找来计算出recoveryID。这种做法完全没有减小实际开销，只是采用"乾坤大挪移"把开销转移到签名环节了而已。实际上，recoveryID是有更快的计算方式可得到的，下一节可见。【这部分代码继承自web3j，之前没有深入考究其实现方式，当前web3j还仍然是这种实现】。出于老码农的本能，发现症状就想要第一时间先搞清楚问题影响边界在哪里，于是就有了这样的分析和尝试：

同样老司机的章鱼哥快速给出了可喜的结论：

到这里，心里有底了！至少把recoverFromSignature干掉是能够翻倍提升性能的，至于怎么干掉，额...再说呗。出于好奇，想再看看此时的性能数据会怎样（幸亏好奇了一下）。

重新调整姿势（换成采用ThreadLocalRandom生成nonce，以及线程并发数降低为10的版本），再来一发，再次获得惊喜的数据。

这种情况下，性能分布也比较均匀了，没有明显的热点。

凌晨一点，总算收获满意的成果，心情是棒棒的...一秒之后，脑回路蹦出来个问题"国密的会怎样呢?明天再说吧（国密的优化会在另一篇再详细讲）。

填坑：recoveryID的计算方法

由于对Java密码学算法库（bc-java）不够熟悉，此处也遇到不少坑，最终通过继承密码算法库，将更多所需参数暴露返回给上层，总算实现了Java版本的recoveryID计算。

```
// Now we have to work backwards to figure out the recId needed to recover the␣
↪signature.
        ECPoint ecPoint = sig.p;
        BigInteger affineXCoordValue = ecPoint.normalize().getAffineXCoord().
↪toBigInteger();
        BigInteger affineYCoordValue = ecPoint.normalize().getAffineYCoord().
↪toBigInteger();

        int recId = affineYCoordValue.and(BigInteger.ONE).intValue();
        recId |= (affineXCoordValue.compareTo(sig.r) != 0 ? 2 : 0);
        if (sig.s.compareTo(halfCurveN) > 0) {
            sig.s = Sign.CURVE.getN().subtract(sig.s);
            recId = recId ^ 1;
        }
```

关于ECDSA算法的recover机制和recoveryID生成原理，将在后续的推送中详细展开，敬请期待。

后话

每一次做性能优化，都是一次很爽的体验，少不了熬夜，但永不缺激情。对代码进行抽丝剥茧，经历反复多次的发现瓶颈、无情打脸、重拾信心过程，最后到达柳暗花明。 再次用那两句话结尾：过早的优化是万恶之源，没有任何数据支撑的优化是万恶之源。共勉！

杂谈Java合约代码

作者：王章 | FISCO BCOS 核心开发者

FISCO BCOS提供了多种语言的SDK，包括Go、NodeJS、Python和Java。Java SDK不同于其他语言SDK，调用合约时，用户需要使用合约编译工具将Solidity合约源码生成对应的Java代码。这个由合约编译工具生成的与Solidity合约同名的Java代码，我们俗称Java合约代码，本文将介绍如何生成以及使用这份代码。

如何生成Java合约代码

合约编译工具可将Solidity源码生成对应的Java代码，具体使用方式可参考以下链接。在说明合约编译工具如何生成Java合约代码之前，先介绍合约ABI的概念。ABI的定义为：

```
The Contract Application Binary Interface (ABI) is the standard way to interact␣
↪with contracts in the Ethereum ecosystem, both from outside the blockchain and␣
↪for contract-to-contract interaction.
```

ABI是以太坊生态系统中与合约交互的标准方式，包括了外部客户端与合约的交互、合约与合约之间的交互。更通俗的理解，ABI是合约接口的具体描述，包含合约的接口列表、接口名称、参数名称、参数类型、返回类型等等。这种描述通常为JSON格式，见ABI格式详情。在EVM生态中，Solidity编译器可以生成合约ABI信息。合约编译工具生成Java代码时，编译Solidity合约生成ABI信息，并解析ABI文件，根据ABI文件的描述内容，确定合约包含的接口列表、各个接口的输入参数名称/类型列表、返回类型等信息。根据这些信息，合约编译工具为生成的Java合约合约类生成接口。具体可参考以下例子。

```
// 示例合约  HelloWorld.sol
pragma Solidity ^0.4.25;

contract HelloWorld {
    string name;
    function HelloWorld(){
       name = "Hello, World!";
    }

    function get()constant returns(string){
        return name;
    }
    function set(string n){
        name = n;
    }
}
```

HelloWorld合约ABI：

```
[{"constant":false,"inputs":[{"name":"n","type":"string"}],"name":"set","outputs
↪":[],"payable":false,"type":"function","stateMutability":"nonpayable"},{"constant
↪":true,"inputs":[],"name":"get","outputs":[{"name":"","type":"string"}],"payable
↪":false,"type":"function","stateMutability":"view"},{"inputs":[],"type":
↪"constructor","payable":true,"stateMutability":"payable"}]
```

上述代码包含三个接口的描述信息：set、get、constructor(默认构造函数，无参数不处理)。关于set与get接口的生成可参考如下图示。

set接口

docs/articles/4_tools/44_sdk/../../../../images/articles/talking_about_java-contract-co

get接口

docs/articles/4_tools/44_sdk/../../../../images/articles/talking_about_java-contract-co

最终生成的HelloWorld.java类：

```
// HelloWorld.java
// 注意：这里省略了无关内容，只保留了核心接口
public class HelloWorld {
```

(continues on next page)

(续上页)

```
    public RemoteCall<TransactionReceipt> set(String n);
    public RemoteCall<String> get();
}
```

HelloWorld.java类中set、get分别是对HelloWorld合约get、set调用的封装。从上述介绍中可以看出，合约编译工具通过编译获取合约ABI信息，通过解析ABI内容获取合约接口描述信息，为Java类生成对应接口。

面向Java对象

了解了如何生成Java合约代码，接下来将会说明如何通过生成的接口实现对合约的调用。这里仍然以HelloWorld合约说明如何调用接口:

```
HelloWorld helloWorld;     // 初始化HelloWorld对象，省略
TransactionReceipt receipt = helloWorld.set("HelloWorld").send();     // 调用set接口
```

Java SDK这种调用合约的姿势可以总结为：面向Java对象操作合约。在这种方式下，用户只需使用合约编译工具生成Java合约类，对合约的所有操作基于构造的Java合约对象，不需要再关注合约ABI，发送接受细节、交易打包编码、结果返回的解码等屏蔽的细节。具体调用合约方式请参考Java SDK教程。值得一提的是，在特定场景下，面向对象的Java调用方式显然无法满足需求，比如：无法预先生成Java合约代码，或者，交易签名与交易构造服务需要分离。在这些场景下，使用类似nodejs/python sdk的姿势会更灵活。但最灵活的还是，用户自己关心交易编解码、打包、签名、发送、收回包、解码的整体流程。

总结

Java SDK的使用，帮助用户屏蔽编码、签名、发送、接收、解码等流程的细节，通过前期合约ABI转换生成Java合约代码，可以做到一次转义多处使用。但是也有用户希望能够掌握交易发送的全过程，或者是在特定场景下要实现交易发送各个流程的解耦。这种情况下，NodeJS、Python、Go版本的客户端对这些细节的支持更为完善，Java SDK也会逐步开放各个模块的接口，敬请期待。

FISCO BCOS即拆即用的“Python区块链盒子”

作者：汤会枫 | FISCO BCOS开源社区贡献者

作者为Python开发者，将FISCO BCOS Python SDK封装成镜像，以提升配置环境的速度，提高易用性。

前言

作为一个 Python 开发者，我一直想通过 Python 去了解区块链。机缘巧合下，在 2019 年一次开源年会上，我接触并加入了 FISCO BCOS 开源社区，此后，一直用课余时间琢磨 FISCO BCOS 的 Python-SDK。

在配置环境的时候，我耗费了一些时间，因此也萌生了将整个框架封装成一个 docker 镜像的想法，我为它取名“Python 区块链盒子”，就像 Minecraft 的“工作台”一样， 可以提升配置环境的速度，提高易用性。有了这样的想法，我便开始利用课余的时间着手去编写 Dockerfile。

我把搭建好的 docker 镜像分享给周围的同学体验，“即拆即用”的特性反响很好，大家不会因为环境配置难而退怯，能像Minecraft里的Steve一样，放下“工作台”就可以制造出一堆有用的工具出来。

什么是“Python 区块链盒子”?

在回答这个问题之前，先来了解一下 Python-SDK。这是由 FISCO BCOS 开源，帮助开发者运用 Python 语言开发区块链应用的组件。既然是通过 Python 语言开发，我相信它将有持续不断的生命力。

Python 的阅读难度相对比较低，特别方便学生、初学者通过 Python-SDK 去了解、学习区块链。可尝试按以下环境要求安装搭建。

- Python 环境：python 3.6.3, 3.7.x
- FISCO BCOS 节点：请参考 FISCO BCOS 安装搭建

关于 Python-SDK，可点击参考 FISCO BCOS 团队推出的开发教程。

- 《麻雀虽小五脏俱全 | 从 Python-SDK 谈谈 FISCO BCOS 多语言 SDK》
- 《Python-SDK 的前世今生》

“Python 区块链盒子”相当于将配置好的 Python-SDK 和部署好的区块链打包成一个包裹，类似于将一个大房子整理成一辆房车。通过这种方式，用户无需关注环境配置问题，可减少部署时间——不用一分钟就能获得镜像，方便开发者快速上手使用，便于自动化运维。你也可以尝试将这个服务和 JenKins 结合，进一步优化运维过程。“Python 区块链盒子”可以当成 Python 区块链开发的工作台，一个现拆现用的工具箱，而且可以保证开发环境的整洁，最重要的是它很轻，就像 Minecraft 里的工作台。

只要你有一台计算机，就可以随时随地打开这个工具箱。安装工具箱的过程也变成了享受：只需一行代码就能上手直接研究开发，不必过多关注环境配置的繁杂，能节省大量时间，解放双手。

获得并运行“Python 区块链盒子”

可以通过以下部署来获得这个“盒子”。

```
docker run -it -p 20200:20200 --name python_sdk fiscoorg/playground:python_sdk
```

进入容器后，首先需要启动节点，启动节点后就可以“食用”了。在这个过程中，请注意不要使用 sh。

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

随后，在/python_sdk 中进行调试，调试过程中的小贴士：console.py 添加了 tab 自动补全。

```
# 查看 SDK 使用方法
./console.py usage
```

(continues on next page)

(续上页)

```
# 获取节点版本
./console.py getNodeVersion
```

```
bash-5.0# ./console.py getNodeVersion

INFO >> user input : ['getNodeVersion']
INFO >> getNodeVersion >> { "Build Time": "20190923 13:22:09", "Build Type":
↪"Linux/clang/Release", "Chain Id": "1", "FISCO-BCOS Version": "2.1.0", "Git␣
↪Branch": "HEAD", "Git Commit Hash": "cb68124d4fbf3df563a57dfff5f0c6eedc1419cc",
↪"Supported Version": "2.1.0" }
```

完成这些之后就成功了，前前后后，相当于在几秒内就搭建了一个开源的区块链框架。你可以把自己的合约放到/python-sdk/contracts 中，更多可以参考 Python-SDK 的使用方法。这个即拆即用的"区块链盒子"对于要使用 Python 来开发区块链应用或者学习区块链的开发者很有帮助。开发者可以通过调用/python-sdk/client 中的函数使用 ./console.py 命令行并与盒子中正在运行的区块链进行交互。

下面将会以 Python-Flask 开发作为例子，实现调用 HelloWorld 合约的功能。

- step1 进入容器

```
docker run -it -p 20200:20200 -p 80:80 --name flask_web fiscoorg/playground:python_
↪sdk
```

- step2 启动节点

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

- step3 部署 HelloWorld 合约

/python-sdk/contract 下存放了 HelloWorld.sol，可以直接使用这个合约进行测试。首先，查看 HelloWorld.sol 合约的内容。

```
pragma solidity ^0.4.24;

contract HelloWorld{
   string name;

   constructor() public{
      name = "Hello, World!";
  }

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

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

可以通过 get 接口返回字符串 name 的值，通过 set 更新 name 的值。有 Solidity 编程基础的开发者还可以自己定制合约内容。然后，就可以部署 HelloWorld 合约。

```
$ ./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:
```

(continues on next page)

(续上页)

```
{
   "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
```

完成后可得到 HelloWord 合约部署的地址，通过这个地址调用函数接口。

- step4 在/python-sdk 文件夹下编辑 app.py

```
$ vi app.py

# -*- coding:utf-8 -*-
from client.common import transaction_common
from flask import Flask, request
from jinja2 import escape

app = Flask(__name__)

tx_client = transaction_common.TransactionCommon(
↪"0x2d1c577e41809453c50e7e5c3f57d06f3cdd90ce","contracts","HelloWorld")
# tx_client = transaction_common.Transaction_Common("地址","合约路径","合约名称")

@app.route('/')
def index():
    '''
   调用 get 接口，获得 HelloWorld 的字符串
   '''
    new_str = request.args.get('new_str')
    # 通过 new_str 的值更新 HelloWorld 的字符串
    if new_str:
        tx_client.send_transaction_getReceipt("set",(new_str,))
        # 若 new_str 不为空则更新字符串
    return escape(str(tx_client.call_and_decode("get")))

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=80)
```

- step5 安装 app.py 依赖/运行程序

```
pip install flaskpython app.py
```

这样就通过 Python-Flask 框架实现了调用 HelloWorld 合约，通过调用 get 接口查看字符串，调用 set 接口更新字符串。FISCO BCOS 的 Python-SDK 十分适合像我这样的学生党或者初学者去研究和了解区块链技术。十分期待更多开发者参与其中，用它搭建出更多有趣好玩的开源项目。Dockerfile 地址请参考文末，最近我也会对其做一些更新，提升其操作易用性，最新操作手册和消息将在GitHub上发布，欢迎大家关注。点击参考更多Python Demo。

3. 提交 pr 体验

关于如何在 FISCO BCOS 中提交 pr 可点击参考开源社区整理的内容，这里不多做说明。详情请见《揭秘 FISCO BCOS 开源项目开发协作》

我想分享一下个人提交 pr 的体验，整个过程既新奇又有趣。萌生“区块链盒子”的 idea 之后，我很快就编写了第一个 Dockerfile，随即就提交了 pr。很快社区的小哥哥石翔就回复了我，刚开始我以为是国外友人，就一直用” Poor English “和他交流。他十分欢迎我的pr，还不断给我点赞和支持，他不但解决了许多在配置过程中的问题，还时不时分享一些小故事鼓励我。在合并我的pr过程中，十分感谢团队的热情帮助，及时向我提出review意见，并细心给我介绍需要添加的内容功能，这让我充分感受到 FISCO BCOS 社区和睦共处、互帮互助的氛围。

目前，“Python 区块链盒子”还有一些有待优化的地方，例如，每次启动一个容器都需要手动操作，启动节点的条件是–>机器暴露所需端口，默认是 20200, 8045, 30300。后期可能会考虑对默认启动节点的功能进行优化，添加数据卷，以及优化容器大小，使 fiscoorg/playground:python_sdk 更精致。如果你有不错的优化想法，欢迎提交 pr，来一次有趣新奇的 pr 体验

参考链接

- Dockerfile

22.5.6 其他工具

FISCO BCOS 交易解析工具指南

作者：廖飞强 | FISCO BCOS 核心开发者

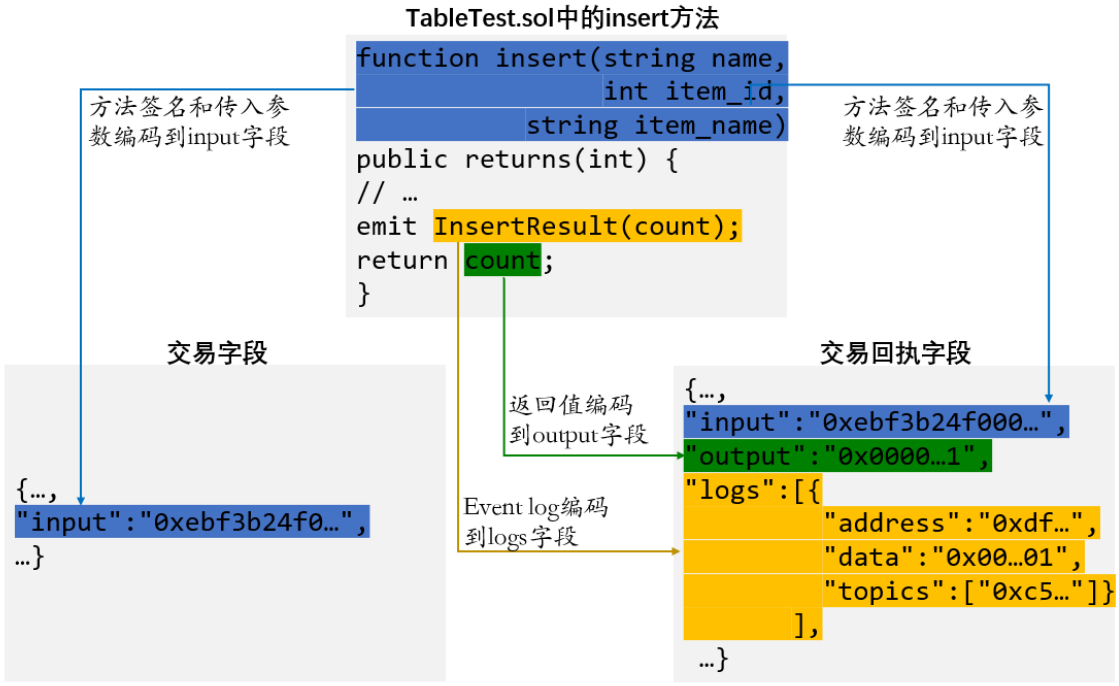
> 本文将介绍 FISCO BCOS的交易解析工具，帮助开发者简单快捷地解析交易和交易回执中的input、output、 logs字段，助力区块链应用开发。

社区用户经常询问：FISCO BCOS的智能合约支持发送交易后，直接获取返回值吗？交易和交易回执中的input、output和logs字段存的是什么，挺好奇，可解吗？怎么解？

现在，让FISCO BCOS交易解析工具来揭开这神秘的面纱！

What: 解析什么？

解析工具解析的是交易和交易回执中的三个重要字段，分别是input、output和logs。这个三个字段分别代表什么，与智能合约有什么关系？下面请出一张图进行分析。



图中为了突出重点，只显示TableTest.sol合约中涉及到交易解析字段的关键代码（TableTest.sol合约是控制台提供的示例合约，用于创建示例表t_test，并提供增删改查方法。完整合约代码可以到控制台目录contracts/solidity/下查阅，或直接通过文档查阅，请参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/manual/smart_contract.html#crud）。

交易和交易回执字段，同样只突出要解析的input、output和logs字段，其他字段省略。其中交易信息包含input字段，交易回执信息包含input、output和logs字段**（注：**FISCO BCOS 2.0.0及以上版本返回的交易回执包含input字段**）。

从图中可以看到，蓝色部分是insert方法的签名，方法签名部分和调用该方法传入的参数，将编码到交易和交易回执的input字段（十六进制字符串）。

绿色部分是方法的返回值，这部分将编码到交易回执的output字段（十六进制字符串）。这里可以解答用户的一个疑问，**即FISCO BCOS的智能合约支持发送交易后获取返回值，该返回值将会编码保存在交易回执的output字段，并且利用交易解析工具可以解析返回值**。

橙色部分是方法调用的event，可以记录event log信息，这部分将编码到交易回执的logs字段（其中address为调用的合约地址，data是event log数据的十六进制编码，topic是event签名的十六进制编码）。

由此可知，合约方法的输入，输出以及event log均编码到了交易和交易回执对应的input、output和logs字段。要知道一个交易或交易回执是调用了合约的哪个方法，输入、输出和event log分别是什么数据，只要解析这个三个字段即可，这正是交易解析工具要解决的问题！

How: 如何使用?

交易解析是web3sdk 2.0.4及以上版本提供的功能。其使用非常简单，三步曲如下：

step 1: 引入web3sdk 2.0.5依赖

Gradle 项目:

compile (‘org.fisco-bcos:web3sdk:2.0.5’)

Maven 项目:

org.fisco-bcos

web3sdk

2.0.5

step 2: 创建交易解析对象TransactionDecoder

使用TransactionDecoderFactory工厂类创建交易解析对象TransactionDecoder，提供两种方式：

1. TransactionDecoder buildTransactionDecoder(String abi, String bin)，传入的参数分别是合约的abi和bin字符串（bin字符串暂不使用，可以直接传入空字符串）。
2. TransactionDecoder buildTransactionDecoder(String contractName)，传入合约名称。需要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。

注意：创建交易解析对象前，请确定解析该交易对应的合约（即该交易是由调用该合约产生的），可以直接提供solidity合约或者用户自行编译，然后传入合约的abi，两种方式均可创建交易解析对象。

step 3: 调用交易解析对象进行解析任务

TransactionDecoder对input，output和logs的解析结果均分别提供返回java对象和json字符串（java对象的json字符串形式）的方法。详细设计文档请参考：https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/sdk/sdk.html#id11。

java对象方便服务端处理数据，json字符串方便客户端处理数据。

交易解析对象的方法列表如下：

其中InputAndOutputResult类，ResultEntity类和EventResultEntity类的结构如下：

```
public class InputAndOutputResult {
   private String function;              // 方法签名
   private String methodID;              // 方法签名编码的字符串，用于方法选择器
   private List<ResultEntity> result;    // 数据列表
 }
public class ResultEntity {
   private String name;    // 字段名称
   private String type;    // 字段类型
   private Object data;    // 字段值
 }
public class EventResultEntity extends ResultEntity {
   private boolean indexed;  // indexed标志位，true表示event字段使用了indexed关键字修饰
 }
```

根据交易对象，可以获取其input字段；根据交易回执对象，可以获取其input、output和logs字段。调用交易解析对象对应的方法即可解析相关字段。

注意：如果解析FISCO BCOS 2.0.0之前的版本(即rc1、rc2、rc3版本)返回的交易回执中的output字段，因为解析output字段的方法需要传入input字段，但交易回执中缺少input字段，则可以根据交易回执中的hash字段，查询一次web3j对象的getTransactionByHash方法获取交易对象，然后从交易对象获取input字段，即可解析output字段。

解析示例：

下面示例解析调用TableTest合约的insert方法，输入参数为(fruit, 1 ,apple)，返回值为1（代表新增1条记录）时产生的交易和交易回执。解析其input、outout和logs字段，其解析结果如下：

由解析结果可知，根据TableTest.sol合约的abi和交易回执中的input、output以及logs字段，通过交易解析工具可以解析出其调用的合约方法名、参数类型、参数值、返回类型、返回值以及event log数据。这就是我们期待的交易解析效果！

Where: 使用场景在哪里?

是英雄，一定有用武之地！交易解析工具入场的地方包括如下场景：

- **控制台**：控制台1.0.4版本已使用交易解析工具解析查询的交易、交易回执以及调用合约时的相关字段。具体用法
- **区块链浏览器、WeBASE管理平台**：用于解码交易和交易回执中的字段，便于交易详情展示。
- **基于web3sdk的区块链应用**：比较重要的意义是用于获取合约方法的返回值，以往一般对于发送交易的方法，习惯用event记录数据，而方法的返回值处于闲置状态。现在可以使用返回值，并利用交易解析工具解析返回值，助力业务开发。

总之，哪里需要交易字段解析，哪里就可以调用交易解析工具！

22.5.7 压测工具

性能压测工具Caliper在FISCO BCOS平台中的实践

作者：李陈希 | FISCO BCOS 核心开发者

Hyperledger Caliper在发布的0.2.0版本中，正式宣布支持FISCO BCOS平台。

关于如何使用Caliper对FISCO BCOS进行压测，开发小哥已经呕心沥血对如何部署Caliper及如何自定义测试用例进行了全面总结，欢迎猛戳FISCO BCOS官方文档学习如何使用：

本文将深入解析Caliper，以便大家更好使用Caliper压测FISCO BCOS。

为何要适配Caliper?

对于区块链开发者和用户而言，性能是评价一个区块链平台重要的考量条件。一直以来，FISCO BCOS都是通过Java SDK中自带的压力测试程序及压测脚本来对FISCO BCOS进行持续的性能追踪。

虽然这些压测手段已经完全能够胜任FISCO BCOS底层开发人员评估性能的需求，但是当压测需求从底层开发人员向需要对自定义的合约及场景进行测试的外部用户延伸时，传统的压测方法则会有些力不从心，需要改善的地方主要集中在以下几点。

扩展测试场景的方式需要精简

目前压测程序直接支持的测试场景是具有代表性的转账场景及增删改查数据表的模拟，如果用户想对自定义的合约及场景进行测试，则需要按照模板自行编写测试程序。 SDK自身已经提供了丰富的API来帮助用户编写这类程序，但是仍然需要用户自己处理合约编译及API转换、压力线程池等细节。 尽管一个用户可能只会有一个测试需求点，但是千人千面，当不同用户的需求累积起来时，重复的工作量就变得相当可观。

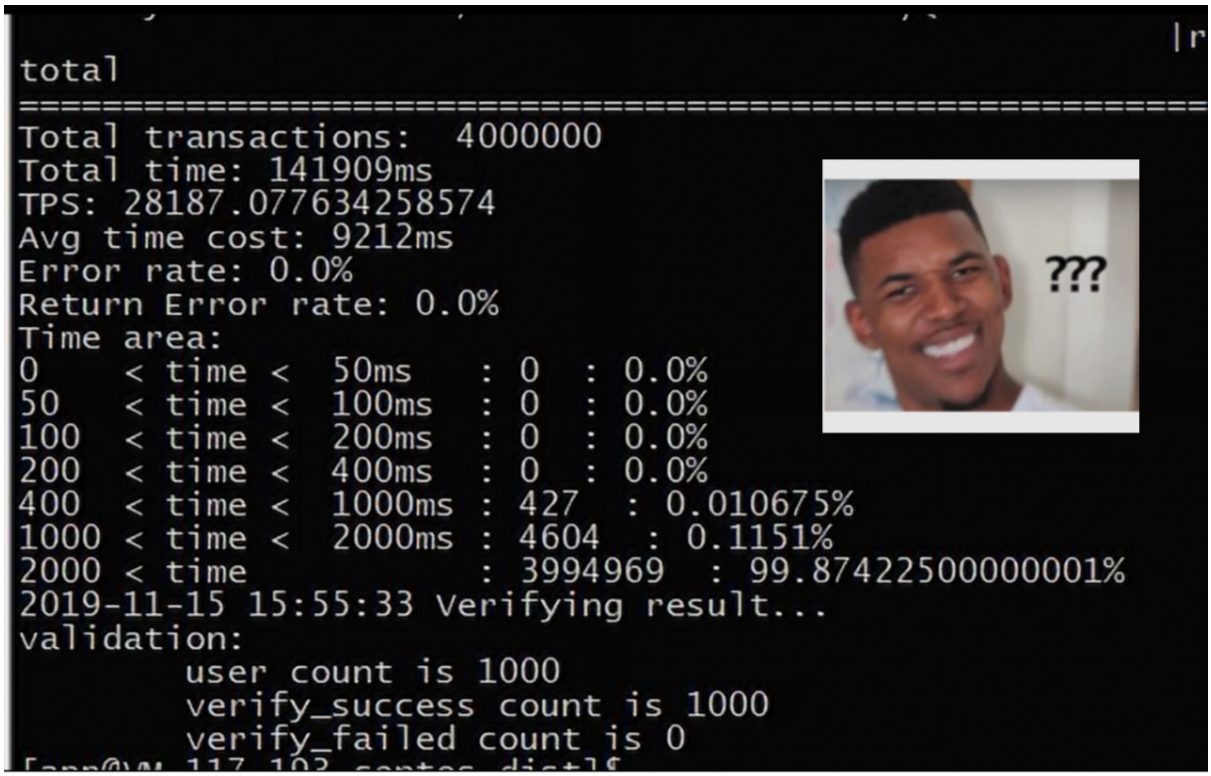
因此，我们希望有一套测试框架帮助我们处理这些琐碎的细节，从而使我们的精力能更加聚焦在构造压测场景本身。

性能指标的定义需要统一

对于性能指标如何计算的问题，FISCO BCOS内部已经形成了一套规范的计算方法，但是当用户自行编写测试程序时，由于测试程序本身的灵活性，用户可以自己定义性能指标该如何计算。 以交易处理能力指标TPS为例，有的用户可能会觉得交易的本地签名时间不应该归到远端节点处理交易的耗时中，也有用户可能会认为总交易量/总时间的计算方式不够精确，而倾向于使用在压测过程中多次采样最后计算平均值的方式。 可以看出，灵活性带来的负面效应是导致各方缺少性能的统一度量衡。 **因此，我们希望能有一套自带标准性能指标定义的测试体系，且最好用户无法直接干预性能指标的计算流程。**

结果的展示方式需要优化

基于命令行的压测程序在编写时更多是面向FISCO BCOS的底层开发人员，因此在使用方式上也显得较为『底层』。当外部用户想使用Java SDK中自带的压力测试程进行测试时，可能会看到如下测试结果:

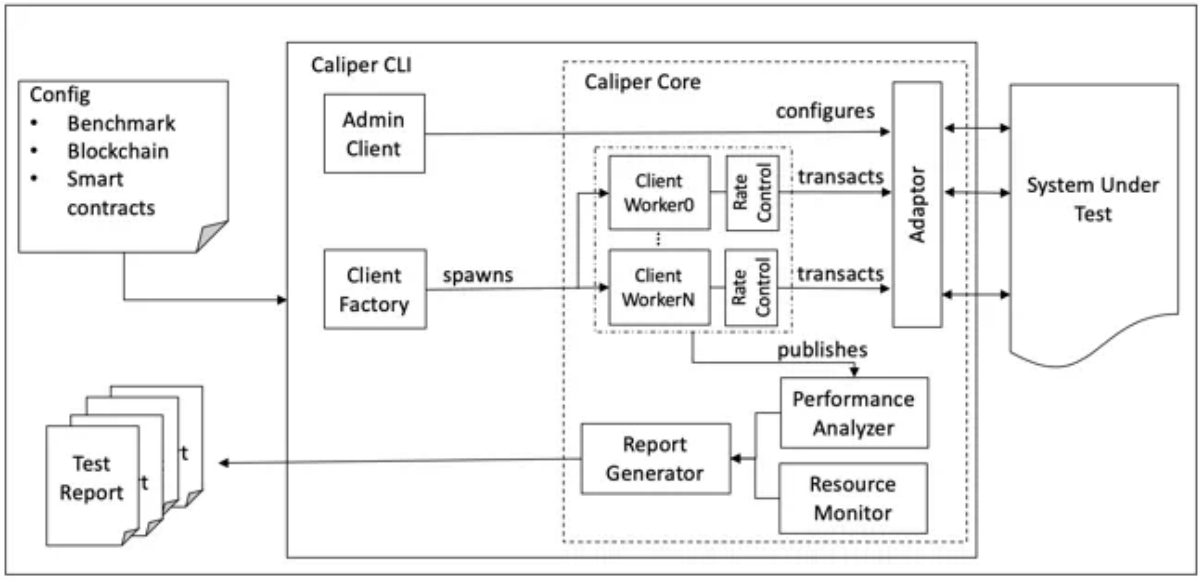


尽管理解每项统计数据的含义并不难，但是如果是向外界进行展示和介绍的话，仍然会显得有一些笨拙和不便。 **因此，我们希望有一个测试工具能在测试完成后输出直观的测试报告，方便用户理解和传播** 。秉承『不要重复造轮子』的精神，我们将目光投向了开源社区，以期能有现成的工具解决FISCO BCOS测试工具的痛点。在经过充分调研后，我们发现了Hyperledger Caliper项目。

Caliper是一个通用区块链性能测试工具。”Caliper”一词的原意就是标尺，Caliper旨在为区块链平台的测试提供一个公共的基准线。Caliper完全开源，因此用户不需要担心由于测试工具不开源导致无法对压测结果进行验证的问题。 同时，Hyperledger项目下设了性能及可扩展性工作组（PWSG），专门负责对各种性能指标（TPS、延迟、资源利用率等）进行形式化、规范化的定义，Caliper在设计也采用这一套性能指标体系并内嵌进了框架中。 Caliper能够方便地对接多种区块链平台并屏蔽了底层细节，用户只需要负责设计具体的测试流程，即可获取Caliper输出的可视化性能测试报告。可以看出，拥有这些特点的Caliper，能恰好满足FISCO BCOS对压测工具的需求。

FISCO BCOS对Caliper框架的适配工作随即展开，回顾来看，整体工作量并不算繁重，最主要的时间开销反倒是花在开发小哥因为不熟悉Node.js（Caliper主要使用Node.js进行开发）而学习了一段时间，这也从侧面也印证了Caliper易于集成的特点。

Caliper长啥样?



Caliper的架构图如上图所示。在Caliper中，Caliper CLI负责为内部的Caliper Core（接口及核心层）提供方便易用的命令行工具。接口及核心层包含区块链适配接口、资源监控模块、性能分析模块及报告生成模块。

区块链适配API

包含诸如在后端区块链上部署智能合约、调用合约、从账本查询状态等操作的接口，这些接口主要由区块链支配器提供。每个区块链适配器使用相应的区块链SDK或RESTful API来实现这些接口，Caliper也正是通过这些适配器提供的接口实现将区块链系统集成进Caliper框架中，目前除FISCO BCOS外，Caliper还支持Fabric、Iroha等区块链系统。

资源监控模块

提供启动/停止监视器和获取后端区块链系统资源消耗状态的支持，资源监控的范围包括CPU、内存、网络IO等。目前Caliper提供两种监视器，一种是监视本地/远程docker容器，另一种则是监控本地进程。

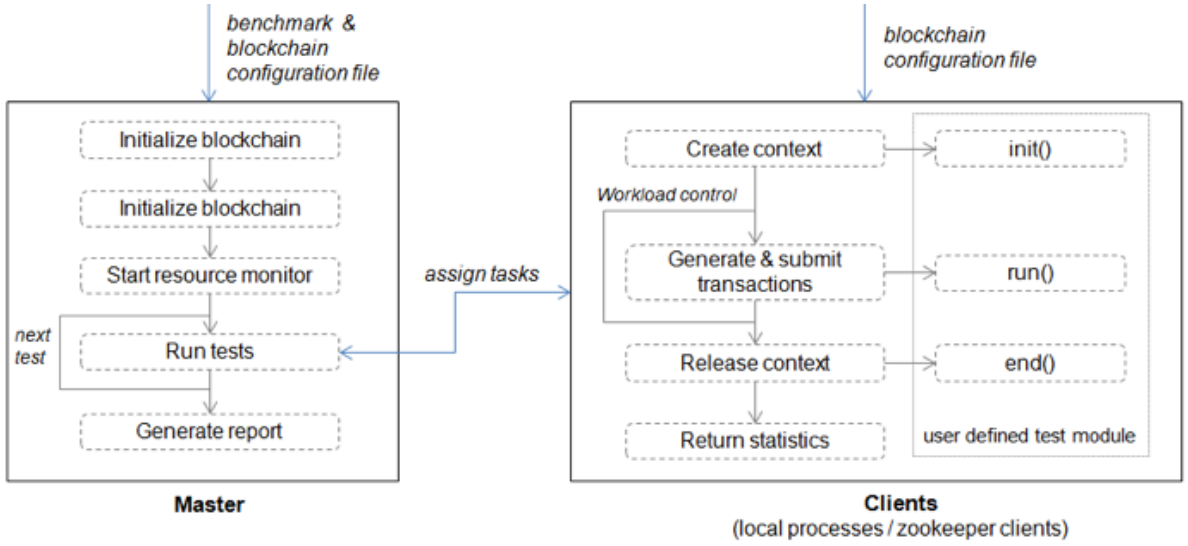
性能分析模块

提供读取预定义性能统计信息（包括TPS、延迟、成功交易数等）和打印基准测试结果等操作的支持。在调用区块链适配接口时，每个交易的关键指标（如创建交易的时间、交易提交时间、交易返回结果等）都会被记录下来，并用于生成最终的预定义性能指标统计信息。

报告生成模块

主要负责对从性能分析模块获取到的性能数据进行美化加工，生成HTML格式测试报告。 Caliper的上层便是应用层，负责对区块链系统实施测试。每次测试都需要设置对应的测试配置文件以及定义后端区块链网络信息的测试参数。基于这些配置，Caliper便可以完成区块链系统的性能测试。 Caliper预置了一个默认的基准测试引擎以帮助测试人员快速理解框架并实施自己的测试，下一节将介绍如何使用基准测试引擎。当然，测试人员也可以不使用测试框架，直接使用区块链适配API完成自有区块链系统的测试。

测试流程



Caliper默认的基准测试引擎采用Master/Client模式来实施测试流程。

整个测试流程由Master进程负责驱动，包含以下三个阶段。

- **准备阶段**：在此阶段，Master进程使用区块链配置文件创建并初始化内部区块链对象，按照配置中指定的参数部署智能合约，并启动监控对象以监控后端区块链系统的资源消耗；
- **测试阶段**：在此阶段，Master进程根据配置文件执行测试，将根据定义的负载生成任务并将其分配给客户端子进程。最后将存储各个客户端返回的性能统计信息以供后续分析。
- **报告阶段**：分析每个测试轮次的所有Client进程的统计数据，并自动生成HTML格式报告。

Client进程主要负责与后端区块链系统进行具体的交互。在Local模式下，Master进程使用Node.js集群模块启动多个本地Client（子进程）来执行实际的测试工作。

由于Node.js本质上是单线程的，因此本地Client子进程集群用来提高Client在多核机器上的性能。在实际使用中，Client子进程的数量越多（在CPU核数能够支持的情况下），Caliper的交易发送及处理能力越高。

在此模式下，总工作负载被平均分配给子进程，每个子进程相当于区块链客户端，子进程拥有临时生成的上下文，可以独立地后端区块链系统交互。子进程上下文通常包含客户端的标识和加密信息，在测试结束后上下文将被自动释放，这些细节并不需要用户关心。

Client进程在第一轮测试时启动，并在完成所有测试后被销毁。

图中最右侧的用户自定义测试模块用于实现交易生成和上链的功能。通过这种方式，测试人员可以实现自己的测试逻辑并将其与基准测试引擎集成。测试模块主要实现3个函数，所有这些函数都应该返回一个Promise对象：

- **init**：将在每个测试轮次开始时由Client调用。所需参数包括当前区块链对象、上下文以及从基准配置文件中读取的用户定义的参数。在该函数内可以保存区块链对象和上下文供以后使用，其他初始化工作也可在此处实现；
- **run**：使用Caliper的区块链适配API处生成交易并将交易上链。Client将根据工作负载重复调用此函数；
- **end**：用于在每轮测试结束时调用，任何结束时需要清理环境的工作都在此处执行。

Caliper使得压测FISCO BCOS的方式变得优雅，同时，FISCO BCOS在适配Caliper的过程中也对Caliper的一些bug和性能问题进行了修复和改进。

Caliper目前仍然在不断地进化中，后续还会陆续添加友好的GUI界面、分布式测试框架、Prometheus监控系统等功能，FISCO BCOS也会持续迭代优化测试工具，以满足用户性能测试的需求。

22.5.8 运维支持

FISCO BCOS权限控制一览

作者：张开翔 | FISCO BCOS 首席架构师

作者语

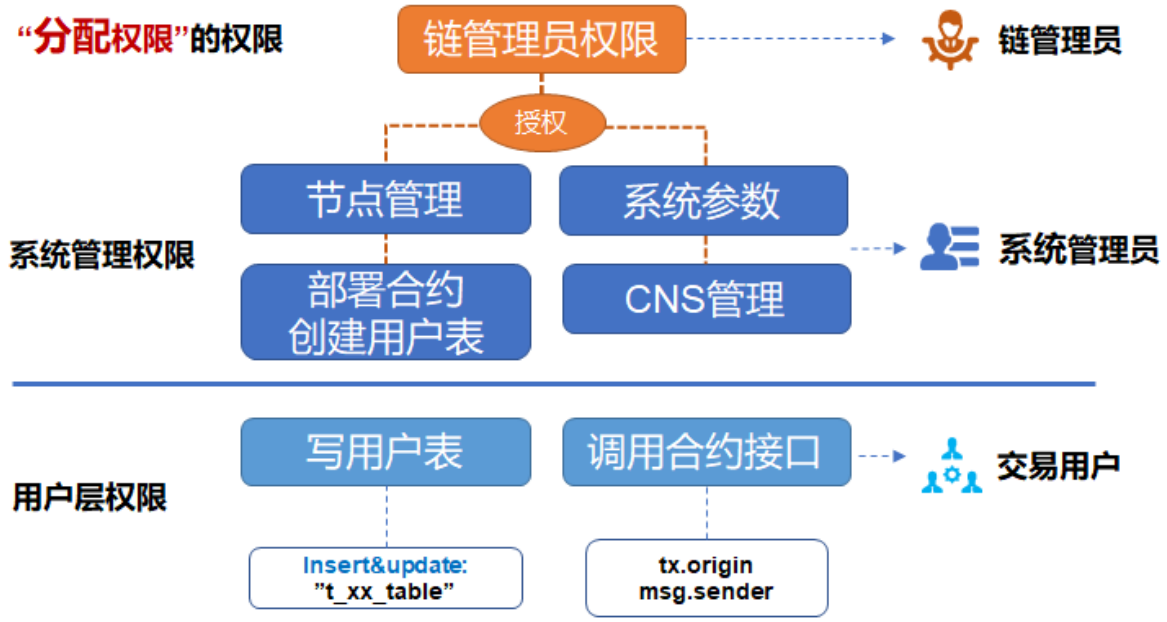
在多方参与的联盟链上，各方分工协作的同时也要做到**职责分明、各司其职**。链的管理者没有必要“既当裁判又当运动员”的参与业务交易，只参与交易的用户也不必很操心智能合约的开发部署。同时，”DO分离”（开发和运维分离）已经是业界成熟的做法，越权操作会带来风险，最终可能破坏信誉和造成资产损失。

清晰、易用、周全的**权限控制能力**，无论是对于信息安全，还是为了完善联盟治理，都至关重要。

本期文章讲的就是FISCO BCOS权限控制这点事，作者从FISCO BCOS的权限分类、典型的联盟链角色设计、权限控制操作基本步骤等角度进行了解析。

FISCO BCOS的权限分类

FISCO BCOS在链刚建立起来的时候，为了方便快速地进行开发和体验，默认并没有做任何权限控制。但如果这条链是用来提供企业级服务，则从一开始就要设计权限控制的策略并落实之。 FISCO BCOS的权限分类：



1. 链管理员权限

即**分配权限的权限**，如定义账户A作为链管理员，A可以为账户B、C、D分配权限；可以设置多个管理员，如果不设置管理员，任何账户都可以无差别修改各种权限。

2. 系统管理权限

目前包括4种：

- 节点管理权限（增删共识节点或观察节点）
- 系统参数修改权限
- 修改CNS合约命名权限

- 能否部署合约和建表的权限

其中部署合约和建表是“二合一”的控制项，在使用CRUD合约时，我们建议部署合约的时候一起把合约里用到的表建了（写在合约的构造函数里），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。

3. 用户表权限

以用户表为粒度，控制某些账户能否**改写某个用户表**，以避免用户表被他人意外修改，这个权限要依赖FISCO BCOS的CRUD合约写法。另外，**读用户表**不受权限控制；如果要进行数据的隐私控制，需要引入数据加密、零知识等技术。

4.合约接口权限

一个合约可以包括多个接口，由于合约里的逻辑和业务密切相关，接口粒度的权限控制由开发者实现，开发者可对msg.sender或 tx.orgin进行判断，决定是否允许本次调用继续处理。

FISCO BCOS的控制台提供了一系列控制权限的命令，可以很方便的为用户**授予（grant）、取消(revoke)、查看(list)**各种权限，可以参见控制台的文档。

联盟链中典型的权限管理角色设计

在联盟链里，不同的角色各司其职，分工协作，典型的角色设计可参照：

1. 链管理员

通常是参与链的多方共同选出一个委员会，一个或多个机构可获得管理员权限，以进行人员管理和权限分配。链管理员并不负责任节点管理，修改系统参数，部署合约等系统管理操作。

2. 系统管理员

指定的业务运营人员或系统运维人员，按需分配各种权限，负责日常的链上管理，包括节点增删，系统参数修改等。由链管理员根据大家约定的治理规则来分配权限，比如只允许指定的账户部署合约，则给他们设定合约部署权限，这样其他账户则不能随意部署合约了。

3.交易用户

用户向区块链发送业务交易请求，业务交易主要是调用合约和读写用户表，可以根据业务逻辑，结合用户表权限和合约接口权限来进行灵活的控制。

4. 监管方

给监管方分配哪些系统和用户表权限，可参考具体的监管规则，如监管方只读所有数据，则不用设定特殊的权限。

管理不同角色的帐户又是另一个复杂的问题，既要清晰区分，又要便于使用，还要保证帐号安全；万一帐号丢失后要支持找回，帐号泄漏后重置，以后会另文介绍。

权限控制操作基本步骤

step1

首先采用控制台grantPermissionManager命令设置一个或多个链管理员角色。 **Tips**：登陆控制台时，命令行里输入的是私钥明文，而使用grantPermissionManager命令输入的是"账户地址"，要注意区分。如：

用户私钥：3bed914595c159cbce70ec5fb6aff3d6797e0c5ee5a7a9224a21cae8932d84a4

对应地址：0xf1585b8d0e08a0a00fff662e24d67ba95a438256

那么登陆控制台的命令行是：

```
./start.sh 1 3bed914595c159cbce70ec5fb6aff3d6797e0c5ee5a7a9224a21cae8932d84a4
```

分配管理员权限的命令行是：

```
grantPermissionManager 0xf1585b8d0e08a0a00fff662e24d67ba95a438256
```

当这个账户得到了链管理员权限后，退出当前控制台或切到另一个终端窗口，用这个账户的私钥登陆一次，即可作为链管理员进行后续的操作了。

Tips：一定要记住管理员地址和私钥的对应关系，否则一旦设置管理员权限，只有管理员可以为其他账户分配权限,其他账户的设置操作都会报告没有权限。

step2

用链管理员账户登陆控制台，根据管理策略，依次给其他系统管理员账户分配节点管理权限，系统参数修改权限，CNS权限，部署合约和建表权限等。然后用一个有相应权限的系统管理员账户私钥登陆控制台，如有部署和建表权限的账号，以便进行下一步。

step3

开发者编写CRUD的合约时，一旦合约在链上创建了用户表，则可以采用控制台的grantUserTableManager命令，由有权限的系统管理员指定哪些账号可以创建表和增删改该表的数据，命令行是

```
grantUserTableManager   t_order  0xf1585b8d0e08a0a00fff662e24d67ba95a438256
```

授权0xf1585b8d0e08a0a00fff662e24d67ba95a438256这个账号可以操作**t_order**表，而其他账号不行，如要读写该表，继续添加权限，对一个表有读写权限的账号可以有多个。

step4

对solidity合约里的某个接口，可以参考这样的代码进行控制：

```
                function testFunction() public returns(int256)
{
        require(msg.sender == tx.origin); //这一行的效果是禁止了合约调合约
         if(msg.sender != address(0x156dff526b422b17c4f576e6c0b243179eaa8407) )  //
↪这里为示例，直接写了账户地址明文，实际上开发时可以灵活处理。
        { return -1;    }  //如果调用者和预设的有权限的调用者不同，则返回
    }
```

msg.sender是当前合约调用者的地址，可以是用户也可以是另一个合约（合约调合约时）；tx.origin则是本次交易的真正发起用户，给交易签名的用户。

小结及参考资料

本文介绍FISCOBCOS在基础层面上提供的一些接口和能力，权限控制的合理性和周密程度最终会取决于使用者，关于不同链的场景化治理和安全控制，可以继续深入探讨，以得出最佳实践。

参考资料

- P2P网络详细设计
- 权限控制设计
- 控制台的文档
- 权限控制实操介绍
- CRUD合约开发介绍
- CRUD合约应用示例

群友问答

@李然：请问几个问题：

1）管理员分配表权限，这个操作是不是在每个节点都可以进行?

2）创建角色，这个操作是不是每个节点都可以操作?

如果以上两个问题答案是肯定的，请问是不是只要有了一个节点的超级权限，就可以修改整个网络的数据?

@光路：是肯定的，建立链之前就先协商好链管理员角色由哪一个或几个账号担任，链建立后第一时间进行角色分配，具体可以参见FISCO BCOS权限控制相关文档。

感谢参与本次话题讨论的小伙伴们！开源社区，因你们更美！

22.6 如何参与社区开发协作

22.6.1 揭秘FISCO BCOS开源项目开发协作

作者：石翔 | FISCO BCOS 核心开发者

作者语

从FISCO BCOS 2.0的第一个版本开发至今，在各位程序猿（媛）的猛烈输出下，FISCO BCOS的节点代码已经达到了11万多行，并且代码仍在快速迭代着。

如此大的代码输入量，对于开发流程来说，是一个巨大的考验。如何保证代码质量？如何做到井然有序呢?

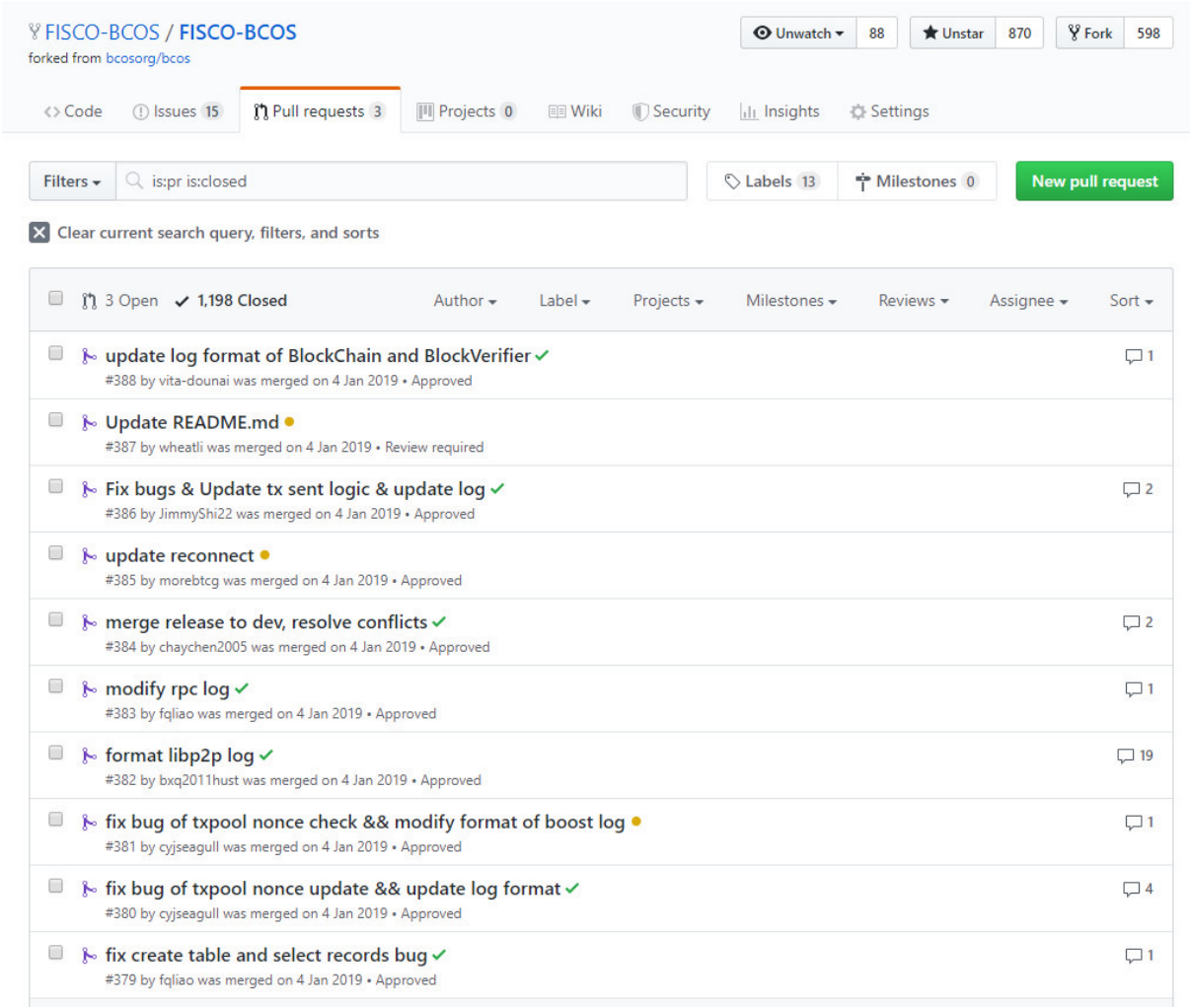
本文将为大家揭秘提交PR的流程，看看在这个过程中，程序猿（媛）是如何进行开发协作的。

PR是什么?

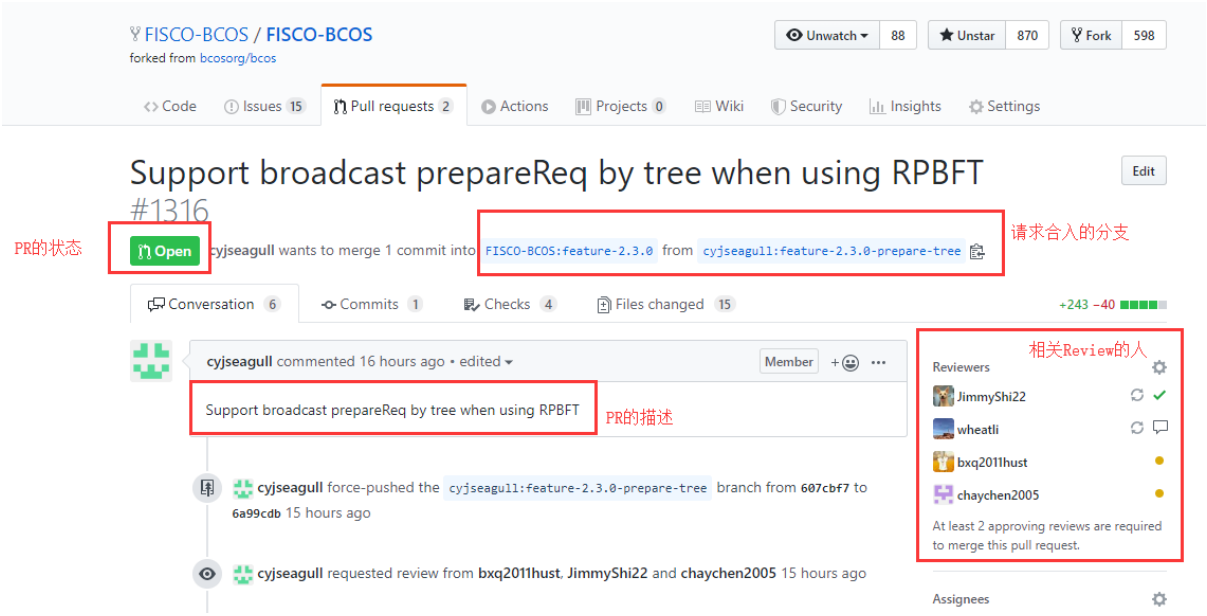
每个人各自在本地开发自己的代码，当代码准备好后，会向FISCO BCOS主仓库提交一个“请求单”，请求主仓库拉取开发好的代码进行合入，此“请求单”就是PR（Pull Request，拉取代码请求）。

在PR中，其它开发者会Review代码，CI（持续集成工具）会对代码的规范和正确性进行初步的检查。当PR达到要求后，就可合入了。

当我们打开FISCO BCOS的PR列表，我们可以看到很多大家的PR记录。有经验老道的老司机（wheatli）在掌控全局，有曾经都想为他生猴子的廖老师（fqliao）在精益求精，有程序媛小姐姐（cyjseagull）在救死扶伤，还有初来乍到的爱猫人士（vita-dounai）在小试牛刀。

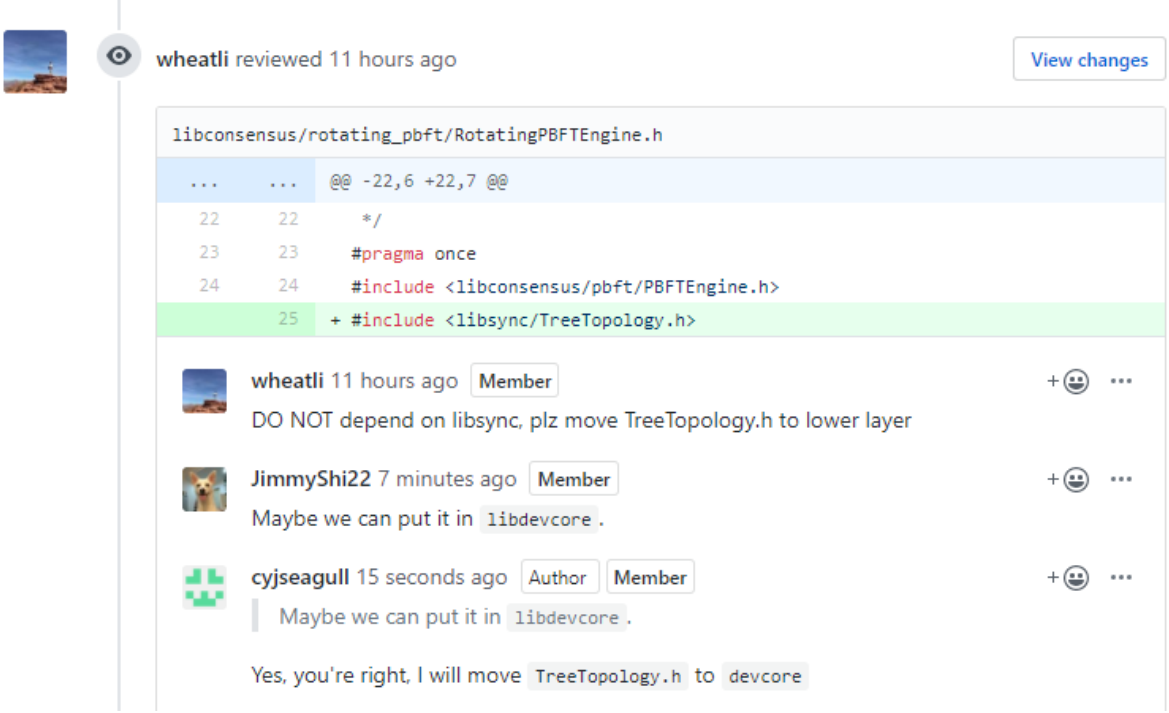


下面我们来看看程序媛小姐姐cyjseagull提的一个PR。这个PR处在Open的状态，表示正在审核中。她请求将开发好的代码合入到feature-2.3.0分支。JimmyShi22、wheatli等小哥哥正在Review，同意了，后面会打钩。



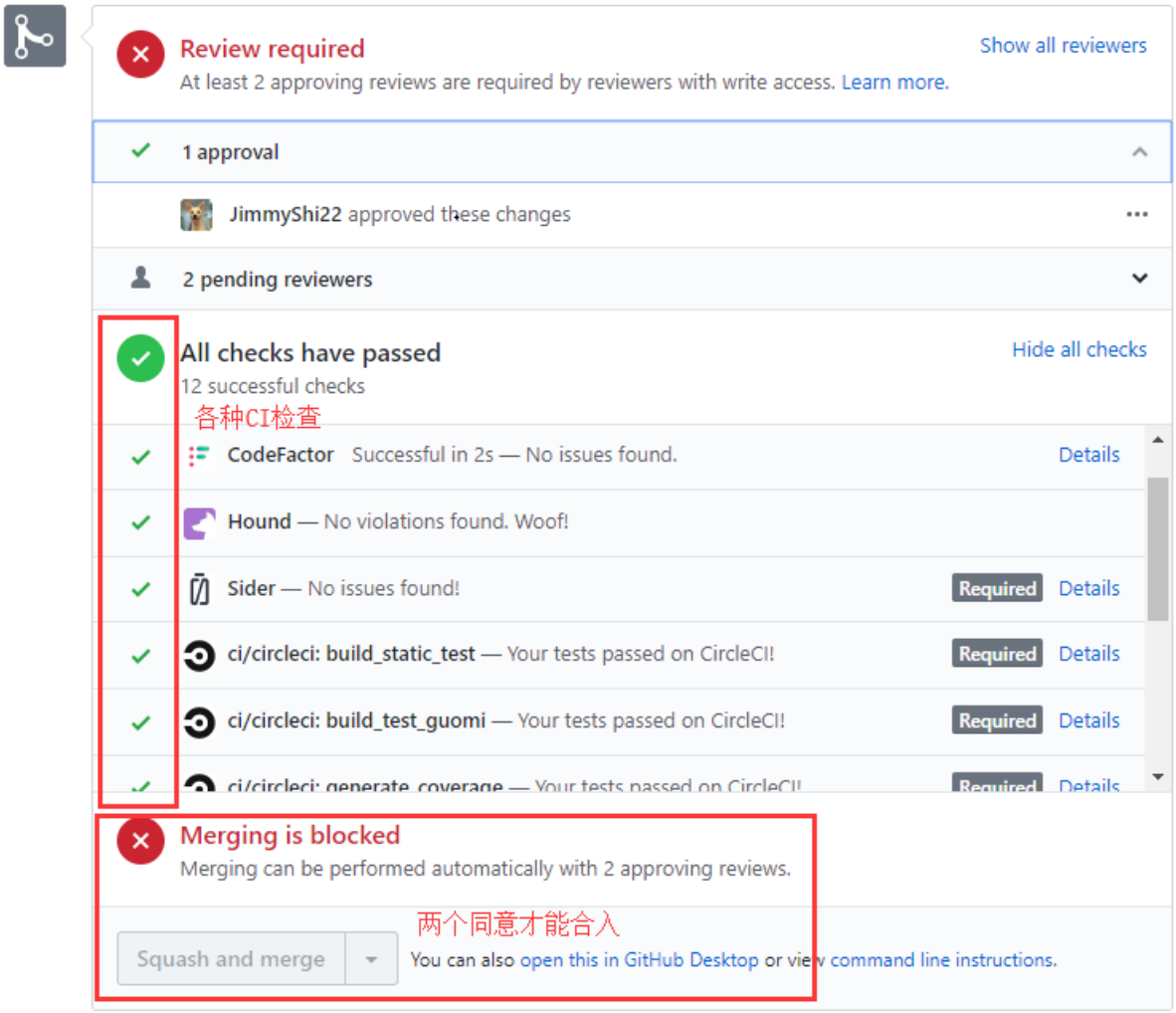
我们向下继续看这个PR，cyjseagull小姐姐的代码遭受到了小哥哥们的挑战，给出了一些Review的意见。共识模块和同步模块彼此独立，不会有依赖。她开发的节点树状拓扑逻辑TreeTopology.h应该放到更底层的模块中去。小姐姐愉快答应了。

通常来说，一个PR需要根据Review反复的修改后才能合入。



此外，PR还会被各种CI（集成测试）检查，cyjseagull小姐姐很稳，通过了所有的检查。但是，一个PR需要至少两个Review的人同意才能合入。

这个PR还缺一个人的同意，合入的按钮是灰的，不能点击。正常情况下，如果同意的人数足够，就可以点击合入按钮，代码合入到相关的分支上。

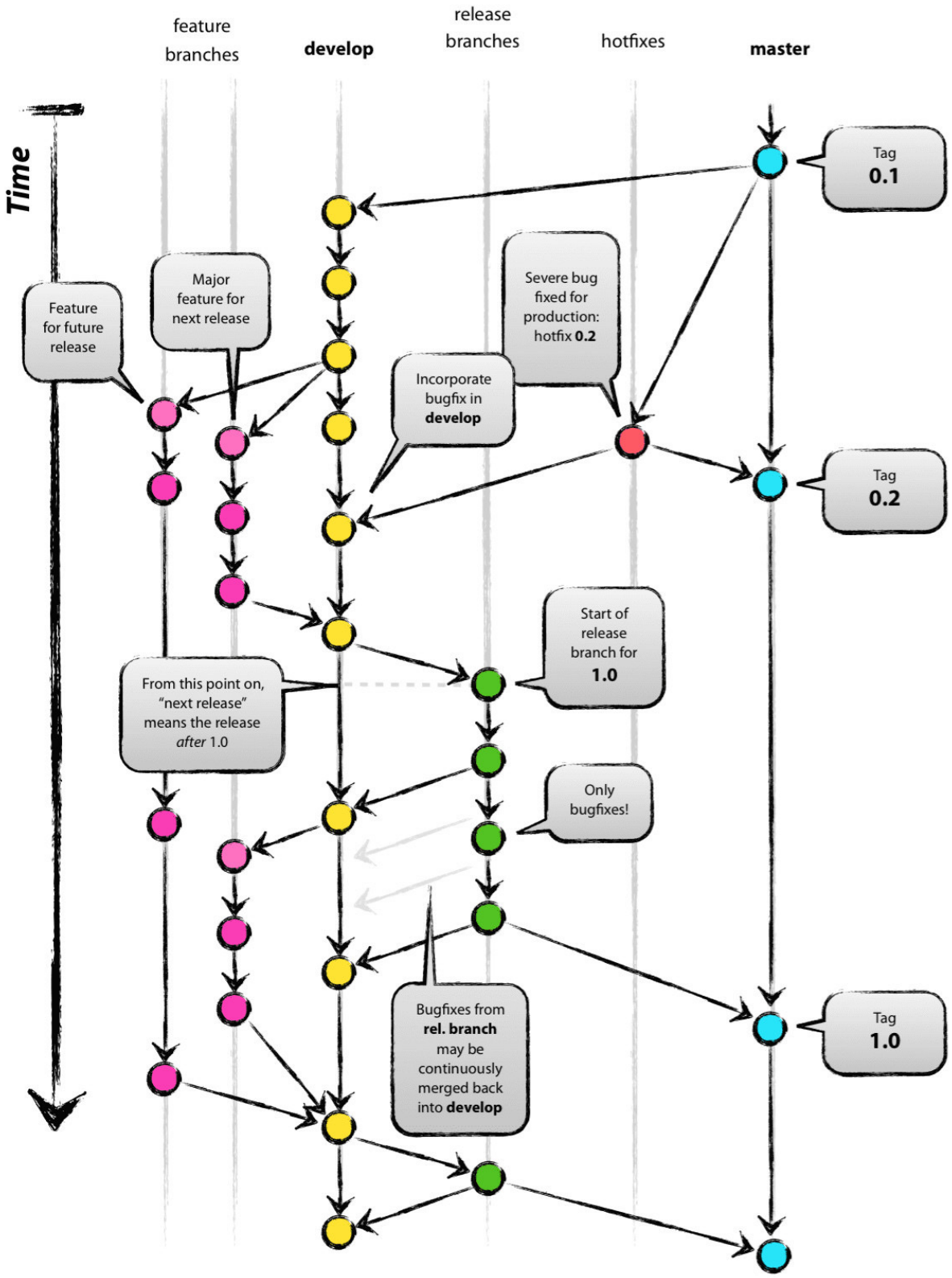


FISCO BCOS分支策略

在明白了PR的概念之后，我们有两个问题需要解决：

- 多人同时开发，如何做到互不影响的情况下进行代码整合呢?
- 合入的代码，还需要进一步的人工测试才能发布，测试在什么阶段介入，能够更有效地进行测试，同时不影响其它人的开发呢?

FISCO BCOS采用了经典的分支策略Gitflow去管理整个开发、测试和发布的流程，下面我们来看看Gitflow。



在Gitflow中，有5种类型的分支：master、develop、feature、release、hotfix。不同分支具有不同的功能。FISCO BCOS代码的开发、测试和发布的阶段，则对应了上述的分支。

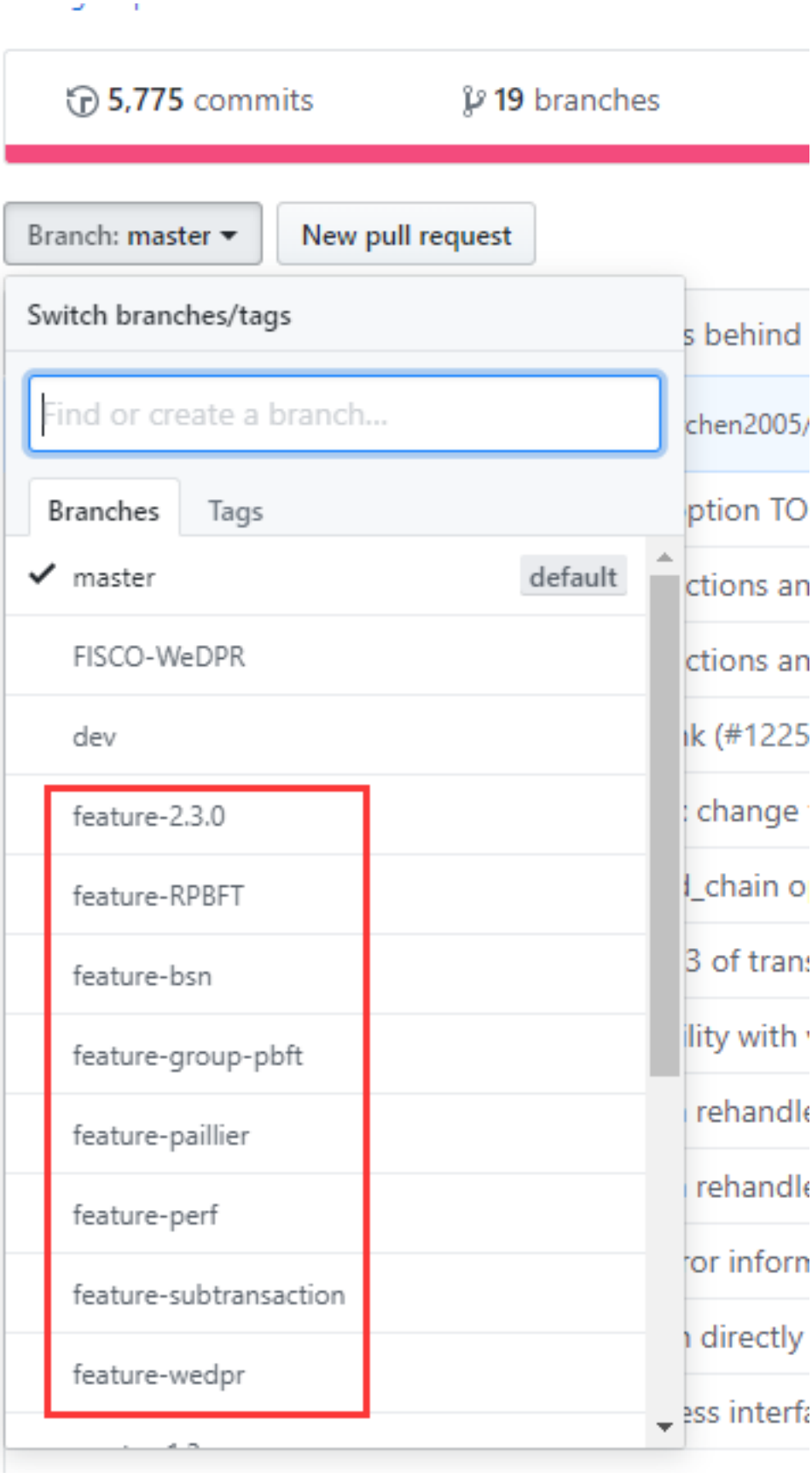
feature分支

FISCO BCOS的代码开发，是基于一个个特性（feature）的。

多个人同时进行开发，是基于自己的feature-xxx分支进行。在FISCO BCOS的主仓库上，我们可以看到有很多这些特性分支，每个特性分支归属于一个（或多个）程序猿（媛）。

他们平时在自己本地的仓库写代码，写好后以PR的形式提交到各自的feature-xxx分支上。上节中cyjseagull小姐姐提交的PR正是处在这个状态。

当feature-xxx分支开发好后，测试介入，进行“特性测试”。测试期间修的bug同样以PR的方式提交到此feature分支。特性测试的目的，是为了保证此特性的功能正确。


5,775 commits
19 branches
Branch: master
New pull request
Switch branches/tags
Find or create a branch...
Branches
Tags
master
default
FISCO-WeDPR
dev
feature-2.3.0
feature-RPBFT
feature-bsn
feature-group-pbft
feature-paillier
feature-perf
feature-subtransaction
feature-wedpr

develop分支

develop分支（在FISCO BCOS中为dev），是用来合并多个feature分支的分支。

当“特性测试”测试通过后，feature-xxx分支就可合入dev分支了。

合并过程同样以PR的方式进行。多个feature同时合入dev分支时，需按照顺序依次合入。先合入dev的feature分支会给后合入的feature分支带来冲突。后合入的feature分支需要解决冲突后再合入dev。

release分支

当我们积累了一些开发好的特性后，需要将代码发布出去。此时从dev分支拉取出release-xxx分支，进行“发布测试”。

在feature分支合并入dev分支时，只能保证特性的完整，但不能保证特性与特性间的影响。当多个特性合并入dev分支后，需要在发布前做最后的总体测试。此时发现的bug直接以PR的方式合入release分支。

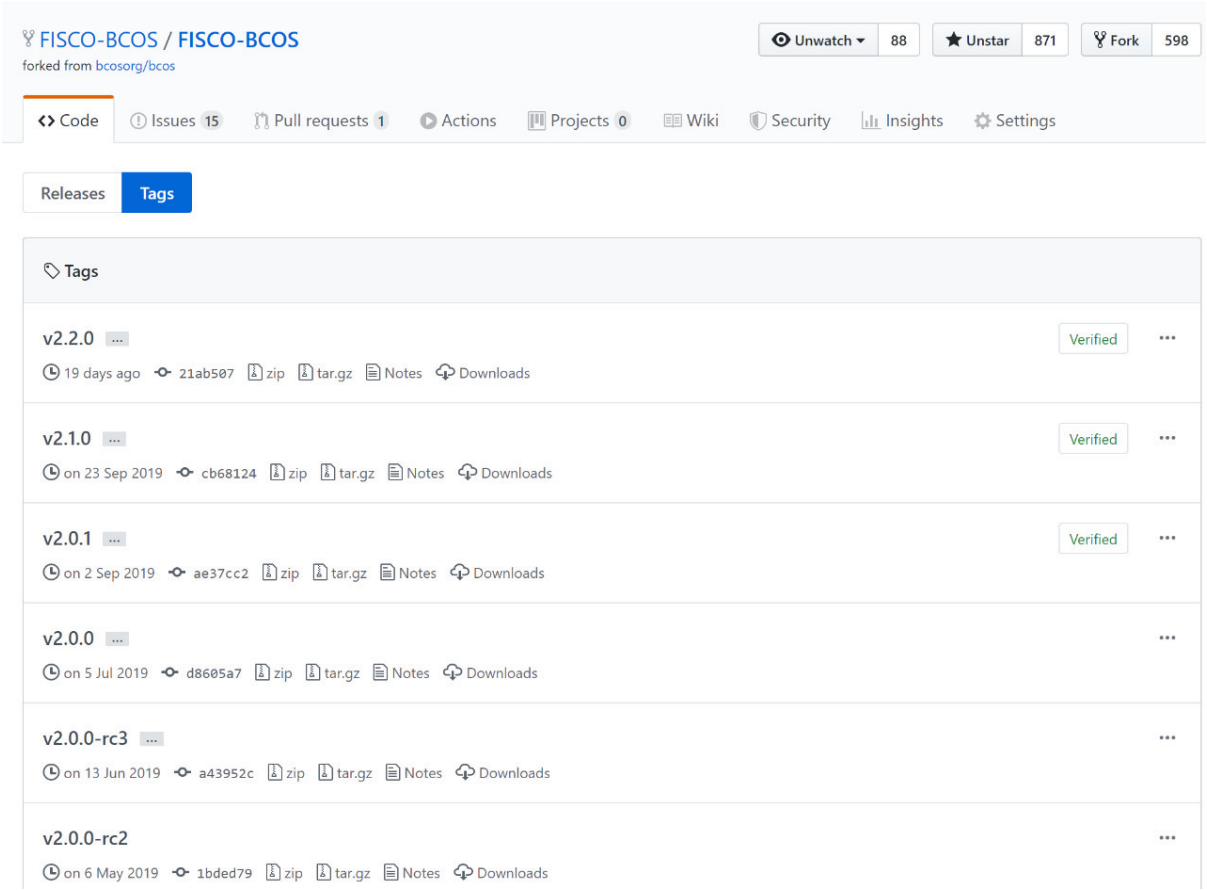
通过此种方式，release分支在进行测试时，不会影响到其它开发者在feature分支上的开发，也不影响feature往dev分支的合入。

master分支

master是主分支，向外提供可用的代码。

当“发布测试”完成后，即可将releaes分支合入master分支。

release分支同样以PR的方式合入master分支。同时，release分支也会合入dev分支。当合入master分支后，根据新的master分支打tag。发布完成！最终发布的版本以tag为准，代码可直接从tag上下载。



hotfix分支

当代码发布后，若出现改动较小的bug或进行小的优化，从master分支上拉出一个hotfix-xxx分支，在其上快速修复。

修复并测试完成后，同时合入master和dev分支。master打小版本的tag。若出现改动范围较大的bug，根据目前项目的发布状态，在feature或release分支上进行修复。

FISCO BCOS分支策略

了解了PR与分支策略，接下来就到提PR的阶段。

- 如果你只是想修改小bug，进行小优化，那么可以直接PR到master分支。
- 如果你是要针对某一特性进行开发，可以与社区沟通方案后，从dev上拉取出自己的feature-xxx分支，就可以开始撸了！然后用PR来提交代码。为了避免Review时大改，需尽量多的提PR来表明自己的思路。PR不要求功能完全可用，只需要最后开发完成时feature分支可用即可。

提PR的具体步骤也可参考《如何为FISCO BCOS做贡献》

欢迎大家积极提PR，一起让FISCO BCOS变得更好。

参考资料

FISCO BCOS仓库

cyjseagull小姐姐的PR

git-flow

22.7 FISCO BCOS应用实践

22.7.1 区块链已革新哪些行业？附应用案例下载

据新华社10月25日晚消息，中共中央政治局10月24日下午就区块链技术发展现状和趋势进行第十八次集体学习。中共中央总书记习近平在主持学习时强调，区块链技术的集成应用在新的技术革新和产业变革中起着重要作用。**我们要把区块链作为核心技术自主创新的重要突破口，明确主攻方向，加大投入力度，着力攻克一批关键核心技术，加快推动区块链技术和产业创新发展。**

报道指出，区块链技术应用已延伸到数字金融、物联网、智能制造、供应链管理、数字资产交易等多个领域。

作为集实践大成的开源底层平台，FISCO BCOS自2017年开源以来，已汇集上万名社区成员、超1000家企业及机构共同参与区块链产业生态构建，在各行各业广泛落地成熟应用案例，包括政务、金融、公益、医疗、教育、交通、版权、商品溯源、供应链、招聘、农业、社交、游戏等。

我们甄选其中较为典型的应用场景出品区块链应用案例精编，带您迅速了解区块链的行业应用现状和前景。

[FISCO BCOS开源社区】公众号后台回复“案例”，可下载高清全本。

《FISCO BCOS案例精编》全文

02 FISCO BCOS

应用领域

03 FISCO BCOS

代表案例

1 文化版权
人民版权平台
版权存证交易平台

2 司法服务
仲裁链 | JustSign-电子合同存证链
可信电子固证平台 | 八戒保全
券商存证链 | 电子借据平台

3 政务服务
澳门智慧城市建设
自然资源登记-不动产
政务诚信管理

4 物联网
智能家居
农业溯源

5 金融行业
网贷机构良性退出投票表决
机构间对账
供应链金融
旅游金融

6 智慧社区
物业管理

7 公益事业
互助急救联盟链

8 人才招聘
乐寻坊-区块链人才活动平台
家政服务诚信证明平台

9 游戏娱乐
网易星球
乐寻坊

01 降低侵权举证成本

基于区块链技术的版权存证，透明、可信、可验证、不可篡改，能够快捷高效保障海量内容的原创版权。

02 助力版权交易

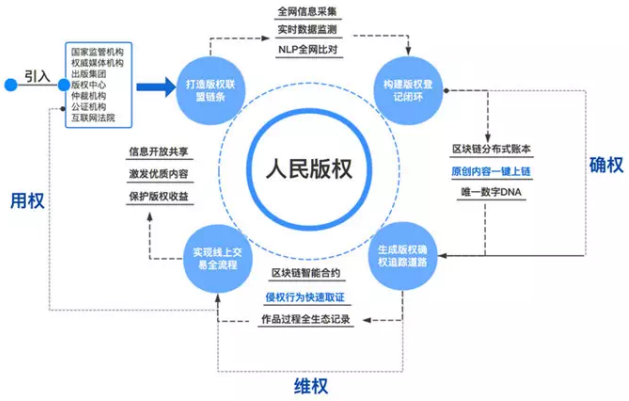
在创作者、传播平台和消费者之间，保障版权确权的真实性、授权规则的准确性、版权定价透明性及支付保证。

03 保障优质内容获益

内容付费层面，助力优质作品获得收益，转载产生的收益亦可被原创者获得。

02 FISCO BCOS

解决方案



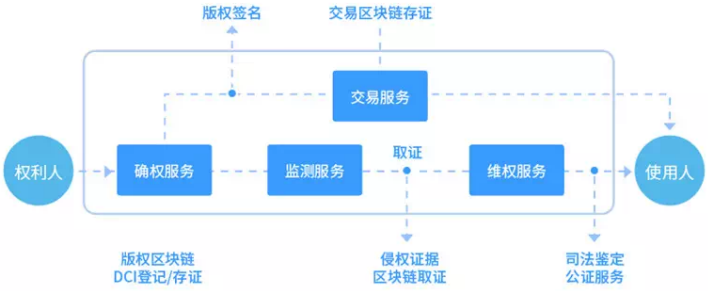
人民网舆情数据中心

人民版权-一站式版权保护管理平台

- 基于FISCO BCOS区块链底层技术构建新闻版权联盟链，实现数字版权确权、监测、侵权取证、诉讼的全流程线上化和自动化
- 通过作者姓名、登记时间、作品名称、作品核心摘要信息生成唯一对应的数字指纹DNA
- 基于区块链的『实体身份标识』及『可信数据交换』解决方案——WeIdentity，对作者身份进行唯一标识及存证相关数据
- 链外全网数据监测，自动发现侵权行为并取证上链

03 FISCO BCOS

解决方案



安妮股份

版权家-版权存证交易平台

- **创作即确权：**版权家可信存证：授时中心授时、CA身份签名，记录在区块链，可在创作完成时立即证明权属。
- **使用即授权：**结合智能合约打造更自动化、智能化的版权交易解决方案，可以解决版权内容权属、分发和获利分成的问题，让正版作品的交易更正规、更方便。
- **发现即维权：**使用大数据发现和辨别侵权行为，使用区块链来固定侵权证据，为维权提供依据。

01 证据不可篡改

在传统存证方案基础上进一步加强了证据的不可篡改性。

02 证据核实可追溯

为后续的证据核实、纠纷解决、裁决送达提供了可信、可追溯、可证明的技术保障。

03 提升效率降低成本

有助于仲裁机构快速完成证据的核实、解决纠纷，降低仲裁过程中人力物力和时间成本，提升司法效率，降低仲裁成本。

02 FISCO BCOS 解决方案

- **完善的防篡改机制：**
  使用区块链技术保全证据，进一步加强了证据的不可篡改性。
- **证据效力得到机构认可：**
  司法机构为作链上节点，对上链数据参与认可和签名，事后可从链上确认数据的真实有效性。
- **服务持续有效：**
  数据被多方共识上链后，即使有部分共识方退出也不会造成数据的丢失或失效。

03 FISCO BCOS 实践案例

仲裁链

- 亦笔科技联合微众银行基于FISCO BCOS打造仲裁链，且引入金融、存证、公证、仲裁、法院等机构作为链上节点，满足证据真实性、合法性、关联性等要求，实现证据及审判的标准化。
- 仲裁链使得传统长达数月的仲裁流程缩短到7天左右，而原本动辄成千上万的仲裁费，也得以降低至几百元。

JustSign-电子合同存证链

- 云之道联手四方精创基于FISCO BCOS开发电子合同缔约和存证系统，其独创的JustKey白盒密码算法实现“手机即U盾”，解决了传统CA兼容性受限、在移动端无法保护密钥安全以及数据集中存储易遭攻击等问题。
- 该系统适用所有需要电子缔约的场景，已在金融行业落地。

可信电子固证平台

- 深圳市电子商务安全证书管理有限公司基于FISCO BCOS建设集成认证、存证、鉴证、取证于一体的存证固证服务平台，打通存证司法通道。典型应用场景包括：电子合同和电子数据防篡改、防抵赖；知识产权原创证明；保护企业设计成果及商业机密；收集合法权益受侵犯的证据。
- 粤港澳大湾区科技基础设施试点项目，承载大湾区对区块链应用落地的期望。

04
FISCO BCOS
实践案例
八戒保全
baoquan.zbj.com
深圳证券通信有限公司
SHENZHEN SECURITIES COMMUNICATION CO.,LTD.
信用算力
Credit Force
八戒保全
• 八戒保全是猪八戒网基于FISCO BCOS搭建的存证保全平台，将保全发起方、保全行为、保全数据全流程存储在区块链。
• 目前八戒保全的业务已经覆盖电子合同、数字版权等领域。
券商存证链
• 2019年1月，深证通联合微众银行等机构，基于FISCO BCOS发起存证服务，实现全网同步，数据不被篡改，留痕信息不可抵赖。
• 长城证券是首个试点理财存证业务的券商机构。
电子借据平台
• 由信用算力基于FISCO BCOS，把用户的线上借款凭证存储在区块链上，将借贷关系实时上链，保证借贷信息的不可篡改和可溯源性，并且实现金融可监管。

政务服务

智慧城市建设

澳门智慧城市之证书电子化

01 电子化证书管理

支持多终端、多场景使用，降低机构管理成本，提升用户使用体验。

02 信息跨机构交互

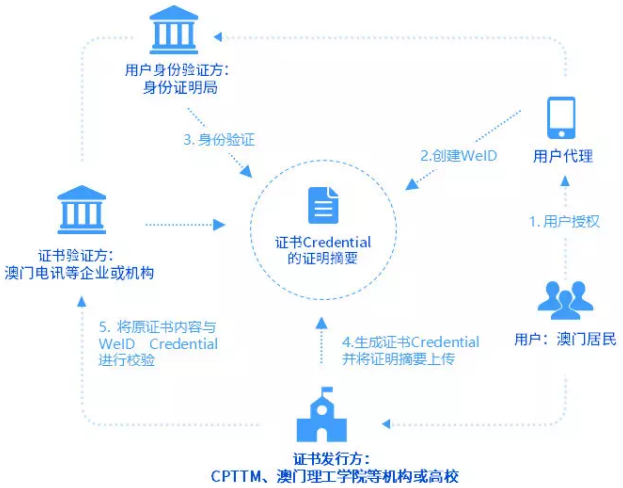
机构使用数据需得到用户授权且授权凭证链上存储，基于区块链的数据传输实现了安全的信息跨机构交互，同时保护用户隐私。

03 信息真实性验证

通过链上ID及摘要信息的验真、增信，确保用户身份及证书数据真实有效，解决机构间信任问题。

02 FISCO BCOS

解决方案



× WeBank

澳门智慧城市建设之证书电子化

- 2019年2月27日，微众银行与澳门政府设立的澳门科学技术发展基金签署合作协议，以区块链等创新技术作为主要支点，推进澳门特区的智慧城市建设与未来型城市发展，提升粤港澳大湾区的科创能力。
- 首个项目基于“WeIdentity”实体身份认证及可信数据交换解决方案展开，实现安全高效的跨机构身份认证和数据合作。
- 项目使用FISCO BCOS区块链底层平台。

02 自然资源登记

01 FISCO BCOS

不动产资源登记

01

保障数据存储安全

利用区块链的安全技术存储系统中的业务数据，保障数据安全；同时，保证改动可追溯。

02

降低验真门槛

将不动产数据向公众开放（不等于公开），让需求方能便利地验证数据，不仅提升内外部查询效率，更大价值在于消除信息不对称，提升造假成本，可有效防范异常使用、欺诈等现象发生。

03

操作便捷高效

登记、汇交一次完成，监管能实时介入，减少和杜绝人为因素干扰。避免传统汇交模式的因涉及人员广，汇交操作繁复，操作系统多样而导致的数据汇交错误或者汇交失败。

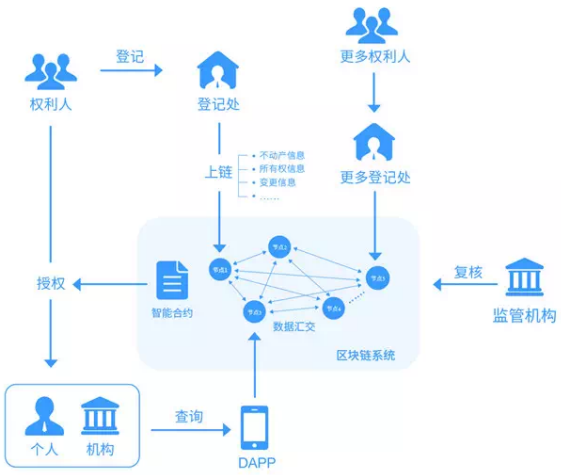
04

优化管理模式

优化管理模式，将监管和执行分离，让监管岗位能实时看到执行层的登记结果，并复核结果，从而最大化不动产登记效率。

02 FISCO BCOS

解决方案



链动时代

链入信任 升级未来

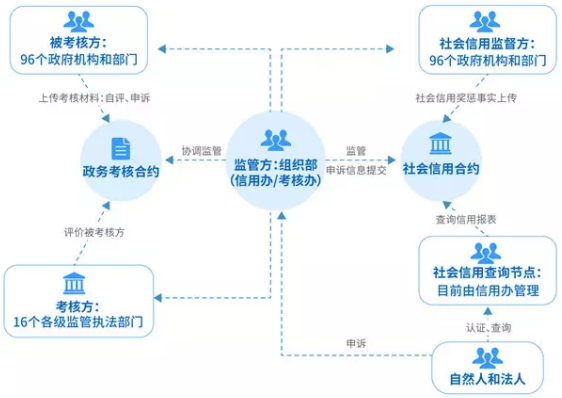
区块链不动产登记系统

- 武汉链动时代科技有限公司基于FISCO BCOS搭建了区块链不动产登记系统。
- 利用区块链技术存储不动产数据，保障数据安全，并实时汇交。对入链数据和交易情况进行实时监控，支持数据的共享查询。

03
政务诚信管理
01
FISCO BCOS
荣成市政务诚信管理
01
信用数据电子化
居民、政务等信用相关数据上链统一保存，安全可靠，查找使用效率高。
02
信息跨部门交互
基于区块链的数据传输共享，实现了安全的信息交互，提升工作效率。
03
数据可追溯
信用数据的上传、修改历史一目了然，不可篡改。
04
构建信用社会
信用数据真实可查、安全可信，助力社会信用体系建设，构建诚信社会。

02
FISCO BCOS

解决方案



× 观海数据

区块链政务诚信管理系统

- 山东观海数据技术有限公司基于FISCO BCOS搭建山东省荣成市区块链政务诚信管理系统，建设信用信息征集、评价、披露和应用于一体的制度机制。
- 利用区块链技术，将居民、企业、部门单位等纳入信用管理系统，实现数据的公开化、透明化，建立起组织和个人的“信用档案”和“诚信名片”。

物联网

01 智能家居

01 FISCO BCOS 智能家居跨平台联盟

中心化

- 各厂商间数据、协议相对独立，信息不能互通
- 控制指令只能由代表用户的中心发起，树状到达智能终端
- 硬件性能要求高，终端厂商面临设备制造成本的挑战

多中心化

- 构建多中心化的核心架构，实现设备互联和信息共享
- 通过智能合约约束数据采集信息，保护用户隐私
- 通过多层同步，解决多信任域环境下共识效率低的问题

02
FISCO BCOS

解决方案

CHANGHONG长虹

跨平台互联方案

- 长虹信息安全灯塔实验室提出应用FISCO BCOS的跨平台互联方案，将各个设备平台、监管机构、检测机构和可信第三方组成一个联盟链，通过物联网护照机制，构建多中心化的核心架构，实现设备互联和数据共享。
- 在这个联盟链的基础之上，通过一系列的合约组合，保证设备联动过程中各个环节安全可靠执行，且公平体现各方的意志，保障各方权益。

02

农业溯源

01 FISCO BCOS

畜牧养殖综合服务

01 可信溯源 安全监管

依托于物联网（IoT），采集猪牛羊等牲畜在养殖环节的特征信息，结合时间戳，在区块链进行多维存证，实现养殖环节的安全监管；与生产、加工、物流等环节打通数据通道，实现食品的全流程溯源。

02 远程放牧 节省成本

通过防拆卸、电子围栏、运动轨迹、位置定位等物联网应用技术，牧主能够及时发现并找到牲口，减少和避免牧民、畜牧企业的偷盗丢失损失。

03 在线投保 理赔溯源

投保保险的猪牛羊若发生意外死亡，结合运动轨迹跟踪信息及拆除报警功能，清晰判定牲畜为意外死亡而非人为骗保，有效避免保险理赔纠纷。解决牧民不给保，险企不敢保的矛盾。

02 FISCO BCOS

解决方案

全链通 iallchain

畜牧区块链溯源保险业务平台

- 全链通基于FISCO BCOS上线畜牧养殖综合服务平台，通过智能物联网终端设备实时采集畜牧数据，并将畜牧数据上传至区块链溯源平台完成存证，同时结合大数据处理中心分析，支撑智慧养殖溯源与畜牧保险业务。
- **应用场景：**可信畜牧/种植产业食品安全溯源服务、区块链畜牧保险链上签约保单流转、畜牧养殖贷款溯源、基于区块链的数据共享服务、智慧养殖服务等。

01 网贷机构良性退出

01 FISCO BCOS

网贷机构良性退出投票

01 助力监管 规范金融市场

为有效稳定金融市场，化解P2P网贷机构引发的金融风险，2019年3月，深圳市互联网金融协会在金融局的指导下发布《深圳市网贷机构良性退出指引》，要求决策事项在网上投票表决。网贷机构良性退出投票平台是《深圳市网络借贷信息中介机构良性退出指引》的配套技术支持平台。

02 有效决策 提升公信力

搭建独立的投票验证服务，任何人都可以通过此服务，来验证投票系统的数据和区块链上存证的数据，是否一致，从而从每个人的角度监督整个投票过程，解决P2P网贷机构良性退出面临的涉众决策难问题，增强投票结果的公信力。

03 打击伪冒 保护隐私

利用人脸识别和数字证书认证方式确认出借人身份，并将投票人的身份标识记载上区块链，既可防止投票过程中存在他人伪冒的情况，又避免了将投票人的全部敏感信息上链，在保护用户隐私的同时，避免了投票冒充、抵赖、篡改的情况发生，确保了投票结果的公正性。

02 FISCO BCOS

解决方案

深圳市互联网金融协会 & WeBank

网贷机构良性退出投票表决平台

- 平台在深圳市、区两级地方金融监管部门的指导和推动下推出，系全国首例P2P网贷机构良性退出统一投票表决平台，解决了网贷机构良性退出面临的涉众决策难问题，拥有完全自主知识产权。
- 截至2019年12月，平台已在深圳市27家网贷机构投入应用，导入出借人39万余人，待收本金214亿余元，为防范化解金融风险做出了积极贡献。（数据来源：深圳商报2020年1月6日报道《深圳地方金融监管局：已推动203家网贷机构完全退出》）

机构间对账

机构间对账

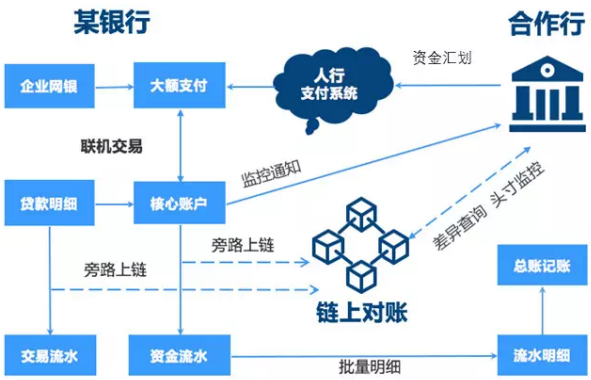
VS

- 每日凌晨批量文件对账
- 整个流程需要在T+1或T+2才能完成
- 文件存在受损、丢失的可能

- 秒级即可确认一批交易
- 每日数万个区块(取决出块时间)
- 每个区块包含多个流水明细记录
- 每批记录都能送达且保证一致性

02 FISCO BCOS

解决方案



WeBank

机构间对账平台

- 2016年8月，微众银行联合合作行推出国内首个在生产环境投产运行的多金融机构间的区块链应用，零故障运行至今。
- 目前已接入3家合作行，在生产环境中运行的交易记录笔数超2000万笔。

03 供应链金融

01 FISCO BCOS

供应链金融

01 助力中小微企业融资

基于区块链的供应链金融可帮助中小微企业实现有效的信用评估和风险控制机制，助力其完成企业融资。

02 降低融资成本

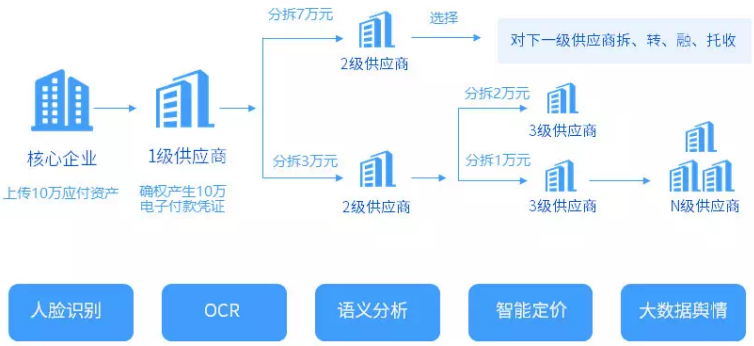
不同参与者使用一致的数据来源，保证了供应链信息的可追溯性，实现供应链透明化。降低行业风险和风控成本。极大简化多方认证和核实的流程，节省了大量的时间和人力物力成本。

03 支持信用资产拆分

资产和信用虚拟化数字化，并支持拆分，能覆盖更多场景，实现供应链线上资产高效流转。

02 FISCO BCOS

解决方案



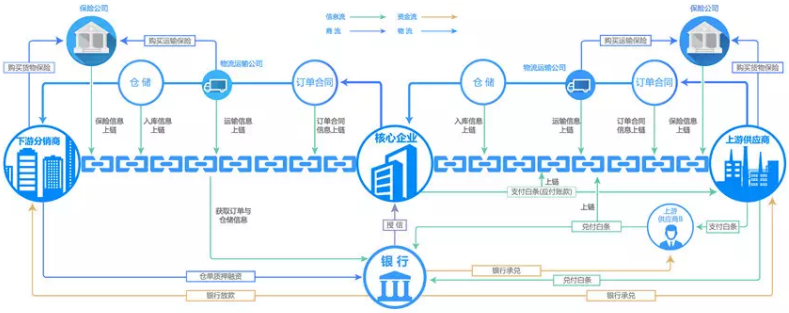
WeBank

供应链金融服务平台

- 微众银行基于FISCO BCOS区块链技术，提出一种新型的纯线上供应链金融服务模式，以核心企业为中心，以实际贸易背景为基础，为核心企业上下游提供融资金融服务。
- 平台依托区块链、大数据、云计算、人工智能等一系列新兴金融科技技术与产业经济的深度融合，通过核心企业汇聚1~N级供应商，引入合作银行等外部金融机构，共同构建及维护完整的线上供应链金融生态圈。

03
FISCO BCOS

解决方案



四方精创
FORMS SYNTRON

供应链金融平台

- 四方精创设计了基于FISCO BCOS的供应链金融平台，利用区块链去中心化、不可篡改、不可抵赖、可追溯的特性，在供应链生态企业间实现信用传递。
- 取得成效：核心企业减少了供应链资金占用，提升现金流；中小企业解决了融资难问题，能更高效，低成本融资；银行能通过区块链的价值信息降低贷款风险和风控成本，获得更多优质资产。

04 旅游金融

01 FISCO BCOS

旅游金融

01

助力机构跨地域服务和发展

区块链技术助力银行机构和旅游企业打造“旅游+金融”业务新模式，充分发挥各自优势，突破地域和资源限制，提供更丰富的产品和服务。

02

机构间实现资源共享

合作机构之间产品资源、流量资源、营销活动、征信信息等共享，实现效益最大化，同时降低风控成本。

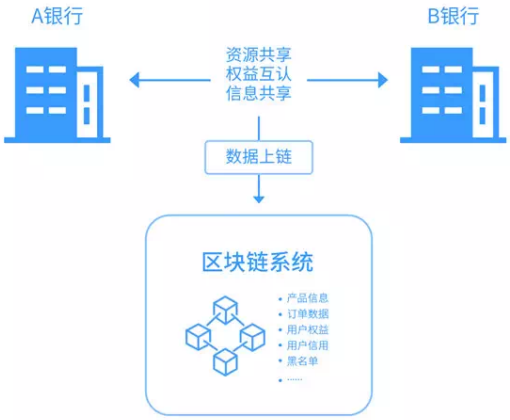
03

降低合作互信成本

通过区块链，实现各参与方之间信息透明，降低信任成本。

02 FISCO BCOS

解决方案



TFCB
城商行旅游金融联盟
Tourism Finance Union of City Commercial Banks

旅游金融联盟平台

- 深圳优讯基于FISCO BCOS搭建的面向联盟成员间使用的开放式客户服务平台，已于2018年6月底上线。
- 成员机构之间共享产品、流量、征信和金融资源，整合旅游产业链资源，实现跨地域的旅游金融协作。

智慧社区

01
FISCO BCOS
物业管理
01
信息共享和财务共管
实现了业主、物业公司、监管部门、银行等多方信息共享和财务共管。
02
降低投票门槛
通过嵌入各类物业服务系统触达业主，降低投票门槛，提升投票覆盖率，从而使电子投票真正能具备替代纸质投票，提升决策效率的功效。
03
社区公共事务决策
为物业管理提供了全新的解决方案，让业主可以参与社区公共事务的决策，促进了业主和物业之间的顺畅协作。
04
社区信息透明安全
发起请款投票，业主投票，资金使用等全流程都在区块链上记录，安全、透明；更换物业，更换账户，更换业委会，都不影响信息的记录和追溯。

02
FISCO BCOS
解决方案
监管部门
银行
腾讯海纳
业委会
物业
APP
小程序
公众号
社区联盟链
投票结果
物业存管账户
应用
发起请款投票
投票信息
业主
认证
个人身份核查
业主身份核查
公安机关
政府监管部门
社区公共收益管理
业委会选主
维修金使用投票
社区公共决策
腾讯海纳
智慧社区开放平台
区块链业主决策系统
• 腾讯海纳是腾讯推出的智慧社区综合解决方案，为了解决物业管理领域的痛点，腾讯海纳基于FISCO BCOS开发了一套区块链业主决策系统，用于社区公共事务的决策以及公共资金的管理。
• 目前，腾讯海纳及各地合作伙伴，正在逐步落地这些场景，已经和多个城市的房管局建立联系，并初步确立了合作意向。

公益事业

01 FISCO BCOS

公益事业

01

公益行为透明可追溯

依托区块链不可篡改、可追溯等特性，实现公益行为、善款流向的透明化，让每一个爱心都清晰可查。

02

联盟机构间价值互联

基于去中心化的数据存储方式，链上公益机构有权限查询开放的数据，且机构间可在数据上相互背书；通过技术对接，联盟机构间可实现价值互联和资源共享。

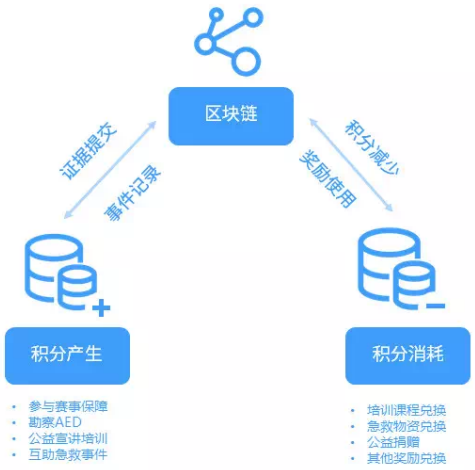
03

保护被捐助人和捐助人隐私

区块链加密技术的使用，很好地保护了被捐助人和捐助人的隐私。只有持有项目私钥人才可以看到项目有关的信息，其它人无法获取项目的详细内容。

02 FISCO BCOS

解决方案



第一反应®

互助急救联盟链

- 第一反应®将现有业务体系和系统与FISCO BCOS的区块链底层技术相结合，使得公益行为及其证据、积分、奖励过程以区块链的方式进行记录，确保公正、透明、不可篡改。
- 项目将赛事保障、公益急救培训、社区互助急救、AED勘察等社会互助急救产品的证据、用户行为、积分奖励记录上链，实现公益行为的透明化、可追溯化和价值化，并且对接其他联盟企业，实现链上公益积分的流转。

人才招聘

01
FISCO BCOS
人才招聘
01
招聘信息真实可信
利用区块链多中心化特性，避免招聘活动过于依赖中心化的中介，信息透明可信，避免欺诈错漏的空间。
02
企业高效低成本获取人才
基于区块链不可篡改及可溯源的特性，忠实记录人才用户信息，让企业或社团以更高效率、更低成本获取优秀人才。
03
保障用户隐私安全
通过密码学技术，人才用户可真正掌控个人数据所有权，保障隐私安全。

02 FISCO BCOS

解决方案

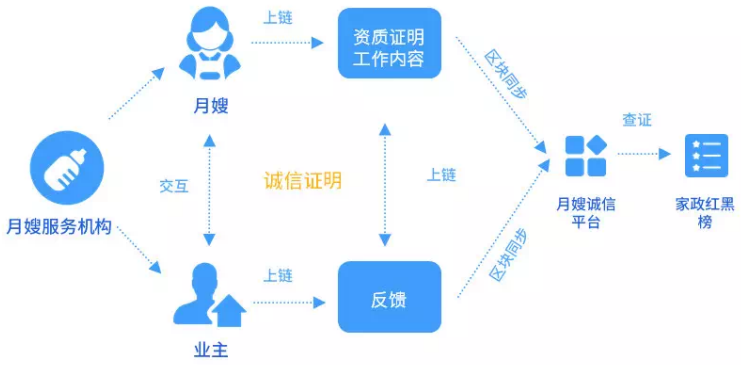
乐寻坊

区块链人才活动平台

- 乐寻坊使用FISCO BCOS作为底层区块链，与企业及社团合作，以联盟链形式实现价值互动的人才活动平台。
- 通过使用区块链密码学技术，人才用户可真正掌控个人数据所有权，隐私安全有保障。

03 FISCO BCOS

解决方案



栽诚科技

家政服务诚信证明平台

- 栽诚科技基于FISCO BCOS打造诚信证明平台，为合作企业提供有据可查、不可篡改的互联网母婴、月嫂诚信证明。
- 通过区块链技术解决家政行业痛点，提高家政从业人员的职业化水平，促进家政服务提质扩容。

01 FISCO BCOS 竞猜游戏

01 提升竞猜环境透明度

传统中心化的竞猜模式，因游戏规则不透明、运作不透明，存在被干预和操纵的空间，区块链能够通过其不可篡改特性和智能合约机制实现竞猜预测规则和信息本身的透明可信。

02 提高效率降低成本

区块链通过智能合约自动化执行以及区块链开奖即结算的特点，避免过于依赖中心化机构，极大地降低了运营和风控成本。

03 保障隐私安全

参与者存在较强的匿名需求。基于区块链技术的竞猜，可通过密码学算法将身份加密，能够实现可被监管的身份匿名。

案例征集

如果您有应用案例或应用计划，敬请通过公众号联系我们，FISCO BCOS开源社区将给予资源、技术等全方位支持。

关于我们

FISCO BCOS是首个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，为各行各业开展区块链应用提供可靠、免费的基础设施，源码在github完全开放：https://github.com/fisco-bcos，欢迎下载体验。

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，于2017年12月正式对外开源，工作组成员包括博彦科技、华为、深证通、神州数码、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。

22.7.2 FISCO BCOS助力深圳破解网贷良性退出难题

读特记者 沈勇

FISCO BCOS区块链底层开源平台在金融领域再添重磅应用！落地网贷机构投票系统，有效解决网贷平台良性退出过程中涉众决策难的问题。该系统今年上半年在深圳率先落地，有望推广至全国。

以下是深圳成功应用的报道，内容整理自读特新闻。原标题：《深圳成功应用区块链技术破解网贷良性退出难题》

虽然区块链技术热度方兴，而深圳已经成功应用于网贷机构良性退出工作中。记者昨日从深圳地方金融监管局获悉，今年上半年，深圳率先全国推出以区块链技术为核心的P2P网贷机构良性退出统一投票表决系统，至今已在全市20家网贷机构投入应用，导入出借人8万余人，待收本金102亿余元。

投票系统让万千出借人有了“表决权”

面对网贷机构退出的兑付方案，普通出借人如何表达自己的意见和诉求？今年上半年，在深圳市、区两级地方金融监管部门的指导和推动下，深圳率先全国推出P2P网贷机构良性退出统一投票表决系统（下称“投票系统”）。

该系统是《深圳市网络借贷信息中介机构良性退出指引》（下称《良性退出指引》）的配套技术支持平台，旨在有效解决平台良性退出过程中涉众决策难的问题，为网贷良性退出工作打响了“开山炮”。

“P2P网贷机构良性退出面临的首要难题是涉众决策难。”深圳市互联网金融协会秘书长曾光介绍，网贷机构管理的资产权属属于出借人，但出借人数量庞大，分布在全国各地，结构复杂诉求不一，平台与出借人沟通成本高、出借人对平台不信任。

为提供涉众决策统一规则，深圳市互联网金融协会率先推出《良性退出指引》，创新设置出借人监督委员会、“三分之二加双过半”的表决规则等一系列良性退出规则要求。

在《良性退出指引》基础上，再度推出“网贷机构投票系统”，使涉及出借人重大利益事项的投票表决成为可能。“在以前的出借人表决实际操作中，往往通过微信群、QQ群、社交网站的投票软件，少数代表的现场会等形式进行投票，其合法性和公正性倍受质疑，执行过程也常有反复。”曾光介绍，在此情况下，新的投票系统为出借人提供投票表决服务，致力于实现公平、公开、透明的投票表决环境。出借人监督委员会负责对相关事项进行审核监督，网贷机构授权协会提供信息披露及投票表决服务。

记者了解到，目前深圳市已经有20家网贷机构使用投票系统。曾光介绍，“使用投票系统的大部分网贷机构都按照《退出指引》要求，积极推动了第一次重大事项投票表决，确认退出流程，选举监委会成员并对清退组和监委会授权。”另外部分平台已通过投票系统完成第二轮投票表决事项，各平台的良性退出工作进展有序。

区块链技术好钢用在刀刃上

宝剑锋从磨砺出，区块链技术的应用，正是“网贷机构投票系统”破解涉众决策难的刀锋。据悉，投票系统是基于FISCO BCOS区块链开源底层平台搭建，同时引入人工智能、生物识别、数字认证等多项国内领先技术，拥有完全自主知识产权。

FISCO BCOS区块链开源底层平台由金融区块链合作联盟（深圳）（简称“金链盟”）开源工作组牵头推出。FISCO BCOS代码已于2017年完全开源并持续迭代更新，目前其开源社区拥有上万名社区成员，超过500家企业参与，成功落地超过60个生产环境应用案例，已经发展成为最大最活跃的国产联盟链开源生态圈。

网贷机构投票系统在设计中成功运用了该平台的技术成果，并融合多项其他前沿技术，集中打造了纯线上、智能化、政府和金融机构共同参与、具备司法仲裁效力的多功能投票决策系统。

首先，投票系统引入智能语音交互机器人和线上人脸识别技术，搭建投票人身份标识链，记录投票人身份标识。一方面，通过利用机器人与出借人进行语音交互，确认债权信息并留存语音记录；另一方面，利用人脸识别和数字证书认证方式确认出借人身份，并将投票人的身份标识记载上区块链，既可防止投票过程中存在他人伪冒的情况，又避免了将投票人的全部敏感信息上链，在保护用户隐私的同时，确保了投票结果的公信力。

其次，投票系统充分利用区块链技术防篡改、去中心化的技术优势，搭建司法存证仲裁链。通过将所有投票记录和结果的数据摘要在链上进行记录，同样在保障用户隐私情况下，使得每笔投票结果准确被记录且无法被篡改；同时，每笔存证均有权威仲裁机构参与链上司法存证，从法律角度确保了投票结果的证据保存，最终保障了全过程投票数据的司法有效和真实可信。

最后，投票系统还搭建了独立的投票验证服务，任何人都可以通过此服务，来验证投票系统的数据和区块链上存证的数据，是否一致，从而从每个人的角度监督整个投票过程，增强投票结果的公信力。

22.7.3 “人民版权”存证原创新闻超200万篇 | FISCO BCOS应用案例

FISCO BCOS开源社区公众号回复【案例】，下载FISCO BCOS最新案例精编

2019年7月，基于FISCO BCOS区块链底层技术研发的一站式版权管理平台“人民版权”正式上线。至2020年第一季度，该平台在版权原创存证、侵权监测、司法维权方面取得一系列成果：

“人民版权”已为超过200万篇新闻稿件进行版权存证；可自动识别的新闻数超过1亿条，相当于3年的新闻总量；全网监测数据量日均近300万条, 全年总监测量超过10亿条。

目前，“人民版权”已经通过国家网信办境内区块链信息服务备案。未来，平台将持续通过区块链、人工智能等创新技术与知识产权保护深度融合，对抗侵权问题，助力构建版权保护新生态。

区块链助力版权保护 存证新闻报道200万篇

近年来，近六成原创媒体作者遭遇过内容侵权，原创稿件被侵权频次高达每篇作品3.64次。网络的虚拟性使得侵权行为难以被及时发现和识别；网络的即时性和裂变传播，令侵权的发展更为迅速，为版权溯源、取证维权也带来了困难。

“人民版权”利用区块链技术的完整性、可追溯性、不可篡改性等特点，综合应用基于区块链的分布式身份解决方案WeIdentity，实现了数字作品链上信息追溯和全网数据监测在内的版权保护全流程管理。

自上线以来，“人民版权”已为2,030,890篇原创新闻报道进行了版权存证保护。全网实时对比监测稿件量达日均2,871,903篇，全年总监测量超过10亿条。识别采集媒体总量达9,988,781家，覆盖了将近全部的电子报刊、网络媒体以及主流客户端。海量信息的实时监测，为快速识别侵权转载并取证上链提供了前提。

首创梯度化司法服务 低成本完成侵权诉讼

“人民版权”首创“梯度化司法综合服务”。面向数字版权维权需求，“人民版权”提供了一套创新司法服务，形成了包含互联网法院、公证、司法鉴定、律师、仲裁为一体的权威司法梯度化服务体系，为版权保护在数权时代的司法维权奠定坚实基础。

此外，与传统的维权模式相比，“人民版权”采用电子证据管理、线上调解、互联网法院诉讼等新模式，以智能电子化代替传统人工模式降低成本，用传统版权服务1/2的价格便可完成确权、维权全流程，帮助用户以最小成本、最高效率维权，提升版权维权的司法效率。

今年初，“人民版权”还正式接入了北京互联网法院“天平链”电子证据平台，成为首个实现版权存证、侵权监测、线上版权交易、司法维权全链条打通的媒体版权平台。

媒体年交易额可达百万 打造线上最大版权交易平台

“人民版权”平台将版权交易环节引入线上。版权交易中心利用区块链技术对内容生产传播全流程的记录，能够进一步帮助媒体实现在信息分发链条上的价值再分配。据悉，线上引入交易环节，媒体单位版权交易年均收益额预估可达七百万元，市场效益可观。

目前，“人民版权”平台已经完成北京歌华有线电视网络股份有限公司、山东数字出版传媒有限公司、微众银行等五大一级节点接入。2019年10月，“北京云·融媒体”成为第一家接入“人民版权”的省级融媒体平台。同时，人民在线/人民网舆情数据中心发起的“人民版权联盟”已有100多家党媒正在对接中。

“人民版权”运用区块链、大数据与AI 识别等创新技术，即将实现视频版权保护功能。目前，“人民版权”正在与西部国家版权交易中心、北方国家版权交易中心、中广电传媒有限公司等进行视频版权方面的合作，携手共促视频版权生态的良性发展。

文章来源：人民网

- FISCO BCOS代码仓库
- WeIdentity代码仓库

22.7.4 BSN首批“官方指定区块链应用”出炉，FISCO BCOS社区四个应用入选

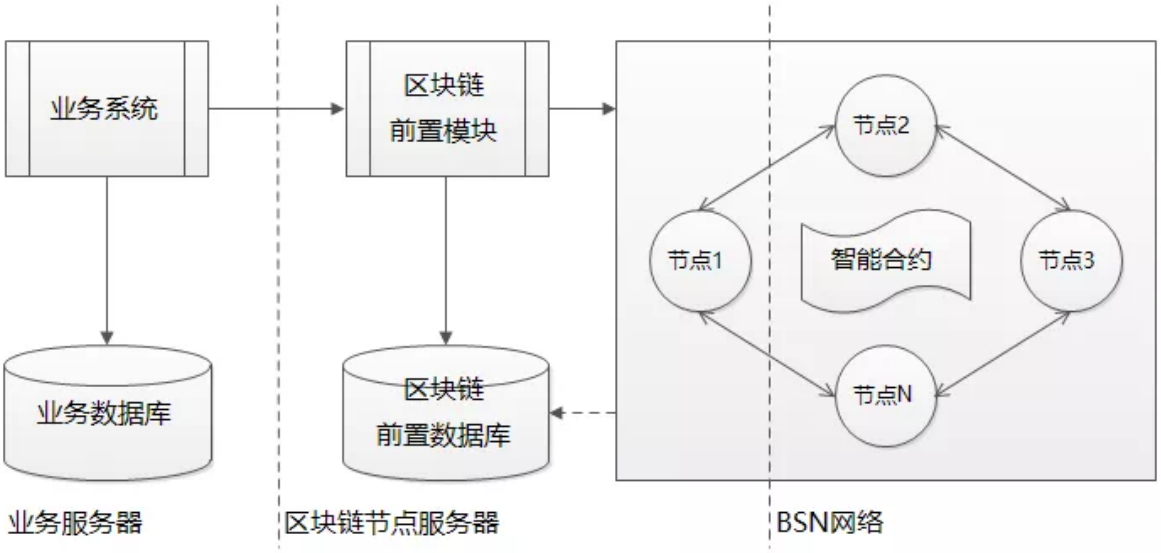
历经一个多月的征集，BSN于近日公布了首批“官方指定区块链应用”名单，其中4个应用基于FISCO BCOS区块链底层研发，覆盖存证、防伪溯源、供应链管理等领域。本着“把最合适的区块链应用，展示给最需要的用户”的目的，BSN发展联盟开发者委员会根据应用准入机制对提交作品进行了审核及综合考量，本次优先选出9种分类的12个区块链应用，作为首批入选BSN的指定应用。

基于FISCO BCOS区块链底层研发的4个应用分别是：链动时代区块链存证服务系统、农业产业全过程溯源云平台区块链应用、惠运链、伊OS透明建造平台。

应用名称	应用提供方	分类	底层框架
链动时代区块链存证服务系统	武汉链动时代科技有限公司	存证	FISCO BCOS
农业产业全过程溯源云平台区块链应用	浙江天演维真网络科技股份有限公司	防伪溯源	FISCO BCOS
惠运链	南京安链数据科技有限公司	供应链管理	FISCO BCOS
伊OS透明建造平台	深圳市建信筑和科技有限公司	其他	FISCO BCOS

链动时代：区块链存证服务系统

链动时代区块链存证服务系统（以下简称“inBC存证系统”）基于BSN上的FISCO BCOS联盟链构建。inBC存证系统帮助用户基于API接口扩展已有业务系统，实现电子证据的存证保全、调用核验。可广泛应用到电子合同、版权保护、证件证书、防伪溯源、公益捐赠等场景和领域。



天演维真：农业产业全过程溯源云平台区块链应用

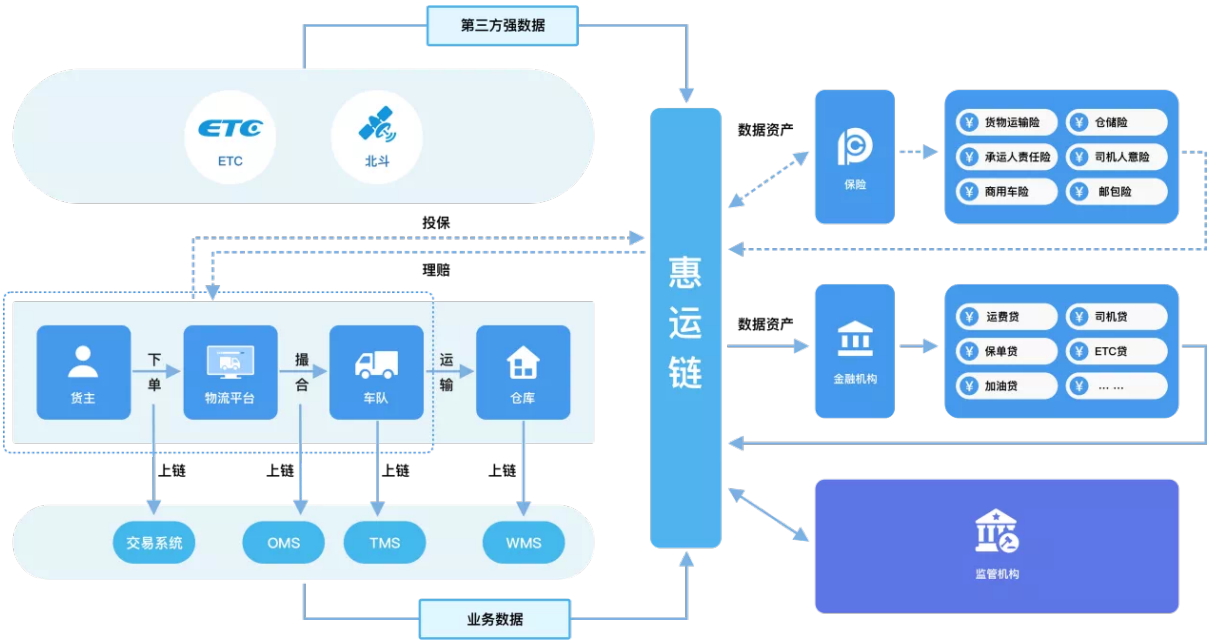
该平台充分结合物联网、区块链、云计算、大数据和地理信息等技术，在图形界面的软件环境下，实现信息采集、审核处理、控制执行、科学决策的“集成化、可视化、网络化和桌面化”。平台通过连接生产、加工、仓储、物流和消费各个环节，梳理统一的产品标准和管控流程，规范企业生产经营行为，提

升企业质量管控能力，切实保障产品品质。同时，将这些信息同步开放给消费者，增强消费者认知，打造消费信任。

目前，苏州市阳澄湖大闸蟹行业协会集体商标防伪追溯体系、赣南脐橙质量安全追溯示范项目等应用均使用该平台。

安链数据：惠运链

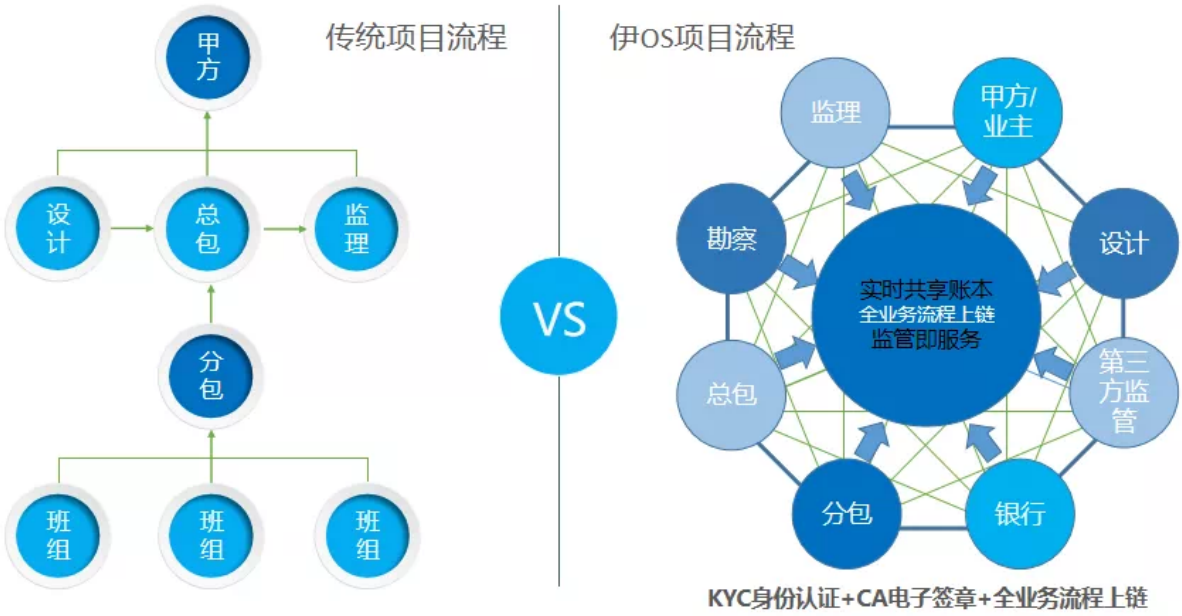
惠运链是安链科技在为物流无车承运平台、保险、银行等企业提供的物流保险供应链金融解决方案。在货运交易和运输物流管理业务场景下，提炼多方协同的单据和信息，通过区块链技术的应用，将物流公司、保险机构、金融机构等生态链节点接入联盟链中，优化资源利用率，提高物流行业整体协作效率，利用可信数据推动保险和金融机构与物流行业融合。



目前惠运链合作用户包括南京融贸通智慧物流科技有限公司、江苏新宁现代物流股份有限公司、太平洋保险、招商银行等。

建信筑和：伊OS透明建造平台

伊OS透明建造平台是深圳市建信筑和科技有限公司研发的一款基于区块链技术应用的建造行业全生命周期管理系统。该平台着力为工程项目构建完整的信用生态系统，运用区块链、大数据等前沿科技技术，协助甲方执行项目全生命周期的管理，让项目责任可追溯，项目管理透明化，让过程变得公平公正。目前平台已为中国雄安集团、深圳建科院未来大厦项目提供服务。



什么是“BSN官方指定应用”?

BSN发展联盟开发者委员会根据客户需求和业界产品的分布，划分了14个相对通用的区块链应用分类以及“其它”分类，共计15个应用分类。分类覆盖供应链管理、供应链金融、司法存证、电子合同、防伪溯源等方面。每个区块链应用分类下只引入3个有代表性的产品解决方案。方案经开发者委员会审核通过后，将作为BSN官方指定推荐的区块链应用，在BSN的各个渠道进行广泛推荐。第二批指定应用也在紧锣密鼓的上线准备中，如您也想加入BSN官方指定应用，欢迎与社区小助手联系。

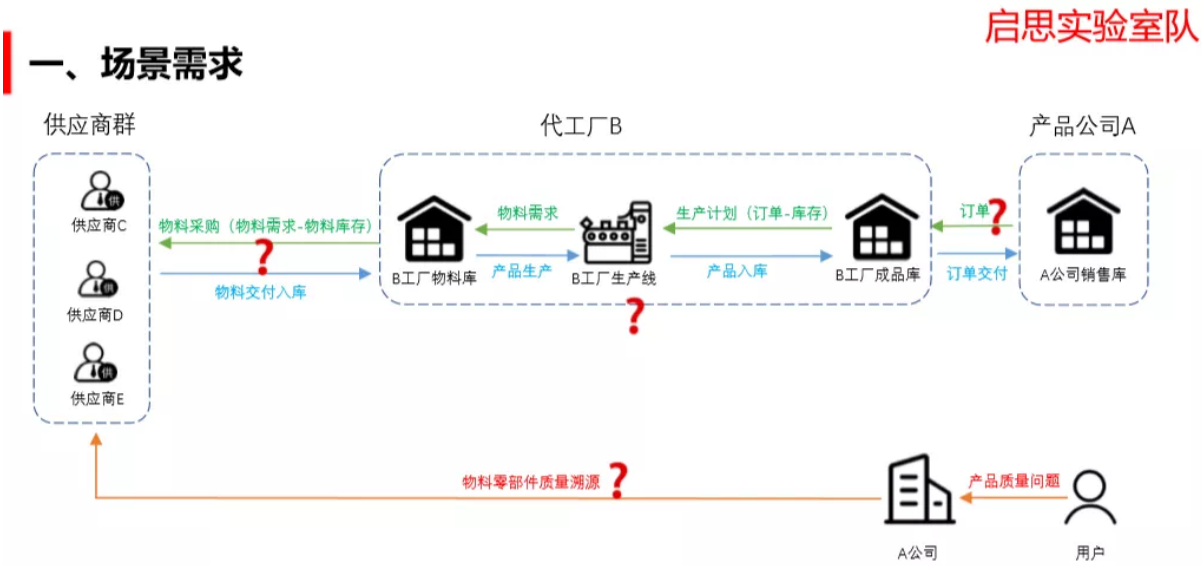
22.7.5 长虹启思实验室：制造业生产协同及质量溯源方案

作者：启思实验室

为什么选择生产协同及质量溯源场景?

我们（下文“我们”均指启思实验室团队）的作品是在生产制造数据信息化的基础上，通过区块链技术，在企业内部建立可信的、合约化信息流，再用多级链打通制造企业供应链和产品下游的数据信息管理，建立多个节点，实现制造企业规则合约执行与数据可信管理，从而达到降低成本、提升效益的效果。

说到这，很多朋友可能会好奇，我们为什么选择这个场景切入？首先，启思实验室（由四川长虹电器股份有限公司信息安全实验室的区块链研究小组的成员组成）本身就专注于区块链技术应用研究，且与主流区块链基础技术提供方合作，提供基于区块链的智慧家庭和工业互联网解决方案。同时，在整个工业互联网和5G发展驱动下，传统制造能力面临一次较大程度的迭代升级，团队注意力一直集中在生产制造，力争从中找到突破点，利用区块链技术助力这一波生产制造升级。其次，在传统制造业中，当产品出现质量问题时，往往会面临物料零部件质量无法溯源、难以追责的难点。具体痛点如下图所示：图中A公司为产品公司和销售公司；B公司为A公司产品的委托生产方；供应商C、供应商D和供应商E为A公司的指定物料供应商。



经过分析和研究，我们梳理了传统制造业中的几大需求：

需求1：生产计划和物料匹配自动化

B公司根据A公司的订单与现有成品库产品库存，分析出差额后制定生产计划。根据产品的生产数量计算出所需物料。生产物料除去物料库的库存后直接向供应商反馈出所需物料的数量，并且预付款项。

需求2：物料供应商及时响应物料标识上链

供应商C、供应商D、供应商E收到物料需求订单，准备发货入库。同时供应商C、供应商D、供应商E的物料匹配唯一标识，信息上链。

需求3：财务可信清算

在订单下发、物料采购、物流运送、订单交货等环节建立财务自动按约定清算的机制。

需求4：售后质量溯源

售后基于物料唯一标识的可信责任确权和质量溯源，做到及时响应。

底层选型的几点考量

了解需求后，下一步就是技术实现，在底层选型上，团队架构师康红娟主要从以下几点进行考量：

- 良好的实际可操作性。区块链应用层与实际业务贴合紧密，尤其是在合约层，重新开发部署业务型合约必然会带来反复调试。因此，底层的完备支持，对于项目成功非常重要。
- 完善的服务层功能组件。用户层与链层的交互，必须经过中间服务层的“嫁接”，而这些“嫁接模块”具备通用性，除了基本的链层功能外，服务层通用组件的完备也至关重要。
- 友好的开源氛围。对于技术极客而言，最大的快乐就是开源。

通过了解评估，团队最终选择了FISCO BCOS作为底层，主要是两方面的原因：

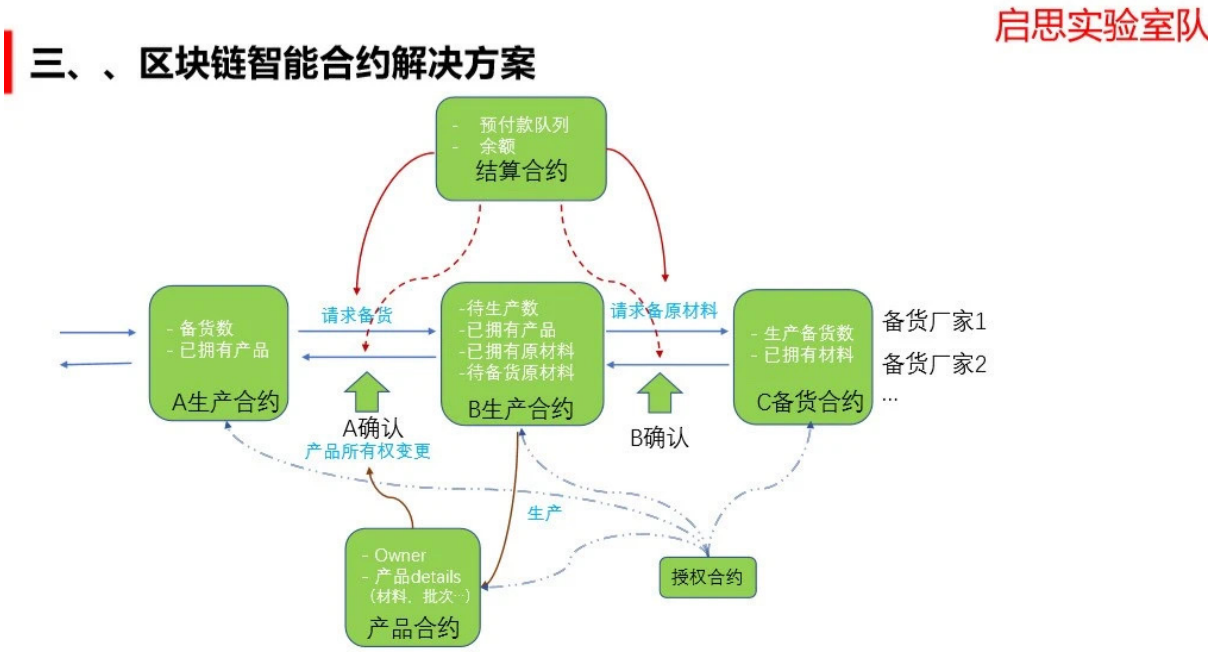
- FISCO BCOS是安全可控的国产开源联盟链，能很好满足、贴合国内企业的使用需求；
- 社区活跃度高，应用场景丰富，对于开发者的技术支持响应及时。

除此之外，FISCO BCOS还具备版本迭代及时、性能强劲、服务中间件丰富等优势。

智能合约解决方案

综上所述，我们从实际业务需求出发，结合区块链技术优势，构建“基础层、核心层、服务层、用户层”等四个关键层级，覆盖核心数据库、业务型合约、数据解析、消息解析、用户管理、业务管理等功能，构建工业互联网领域的订单式生产协同垂直解决方案。

本文我们重点分享其中的智能合约方案，方案如下图。这个方案里包括了产品合约、结算合约、生产合约、备货合约和授权合约，下面将对这几个合约逐一展开介绍。



产品合约

本合约用于完成产品生产过程中产品注册及产品所有权变更、产品溯源。

步骤如下:

- 完成对相关合约的关系设置;
- 合约的admin向某些address授权成为product生产商;
- product生产商调用updateProductPrice对产品价格进行设定;
- customer需要向生产商下产品生产订单前会先getProductPrice来比对价格的高低，然后通过payment合约向生产商下产品生产订单;
- 生产商通过payment合约获取到订单后，进行产品生产;
- 在生产产品前，生产商会通过Material合约来检查自己的产品原料储备是否充足，如果充足就消耗原料进行产品的生产注册registerProduct上链;
- 产品生产商进行交付，customer确认收到货品，确认订单，完成资金及产品所有权的变更。

```
// 厂商msg.sender更新自己productType产品的价格
function updateProductPrice(uint256 productType, uint256 newPrice) public

// 获取厂商 _to的productType的价格
function getProductPrice(address _to, uint256 productType) public view␣
↪returns(uint256 price)

// 设置供应商的原材料合约
function setMaterialContract(address _materialContract) public onlyOwner

// 设置支付合约
function setPaymentContract(address _paymentContract) public onlyOwner

// 产品生产商设置自己产品的批次号，ID，所用材料批次等信息
function registerProduct(uint256 productType, uint256 id, uint256 batchNumber,␣
↪uint256[] memory materialBatches) public
```

(continues on next page)

(续上页)

```
// 更换产品所有权，交付产品
function transferProducts(address from, address to, uint256 productType, uint256␣
↪count) public

// 获取产品详情
function details(uint256 id) public view returns(Product memory)

// 获取msg.sender所拥有的产品数组
function getMyProducts(uint256 productType) public view returns(uint256[] memory␣
↪myProductIDs)

// 获取某种产品的原材料用于哪些产品了
function trace(uint256 materialBatchNum) public view returns(uint256[] memory ids)
```

结算合约

负责用户资产，具体步骤如下：

- 余额与预付款队列；
- 产品所有权变化(手动确认)过程中的自动结算；
- 资金如果不足，不能进行诸如备货，入库等的操作；
- 充值/直接消费/预付款, 余额查询等相关功能。

生产合约

拥有库存生产个数和已拥有产品队列, 和原材料批次队列：

- 获取待生产订单，判断原料库存是否满足需求，不满足则通过结算合约向备货合约预付款下生产订单；
- 通过计算原材料队列，外部唯一标识生成，将原材料的批次信息写入产品进行生产(外部)、入库(调用产品合约的生成入口)。对库存生产个数和已拥有产品队列进行维护；
- 出库(外部调用产品合约的产品所有权变更入口)，对本合约的生产个数和已拥有产品队列库存合约的已拥有产品队列进行维护，通过结算合约进行结算。

备货合约

负责对上游厂商进行备货：

- 通过调用生产备货数对厂家B进行备货，计算已有零件数，外部调用零件入库(填入批次，数量等信息)；
- 出库(外部调用)，将零件信息的批次信息写入到厂家B, 并维护自身数据队列，完成后自动调用结算合约进行结算。

补充：因为零件不像产品一物一码，多个零件只对应到某个批次就可以，不需要单独的零件合约对零件进行维护。

授权合约

通过不同的权限等级，对各个合约中的函数进行访问限制，各种合约间角色的约定及调用权限。比如普通用户没有权限去向供货商下订单，而供货商也不可能向普通用户提供生产完成的商品。

```
// address是否有role权限
function hasRole(bytes32 role, address account) public view returns (bool)

// 获取role权限的成员数
function getRoleMemberCount(bytes32 role) public view returns (uint256)

// 根据role和index获取成员
function getRoleMember(bytes32 role, uint256 index) public view returns (address)

// 获取role角色的管理者
function getRoleAdmin(bytes32 role) public view returns (bytes32)

// 向某个account授予role权限
function grantRole(bytes32 role, address account) public virtual

// 移除account的role权限
function revokeRole(bytes32 role, address account) public virtual

// account放弃持有的role权限
function revokeRole(bytes32 role, address account) public virtual
```

智能合约解决方案

整个智能合约开发过程中，最主要其实是对整个生产流程的梳理。首先我们先进行了需求梳理，然后针对solidity语言开发出一个可用的版本，在此基础上对相关调用建立一个简单的restful服务器，完成对各个接口的测试，有一套完整的可演示流程，最后进行相关开发完成对合约的复现。

22.7.6 多企业间如何实现 “链上协同与治理”

作者：朱立派｜区块链资深开发者（深圳市建信筑和科技有限公司）

为什么选择企业间多方协同治理场景?

很多朋友可能会好奇，为什么会有“区块链上的联合公司”这样看起来天马行空的想法? 首先，区块链技术很适用于“多方协同治理”场景。多中心化自治组织开放式治理能力体现在任何人只要有相应的凭证，就可以公开行使治理。受此启发，我想基于联盟链实现类似功能：联盟各方都具备事先约定的治理能力，通过区块链技术保证治理过程的公平、公正、公开、可追溯和不可抵赖。其次，公司实际业务往来之间对“多方协同治理”存在真实场景需求。

比如一般公司间的业务往来常涉及项目、资金两大类，如果多家公司需要联合管理某个项目，且有资金往来，就可以考虑使用区块链技术实现“链上协同与治理”。大家可保持对项目进展的全局视野一致，同时，任何签字确认的流程都由对应私钥签名后触发，更容易实现责任到人。

链上协同与治理实现思路

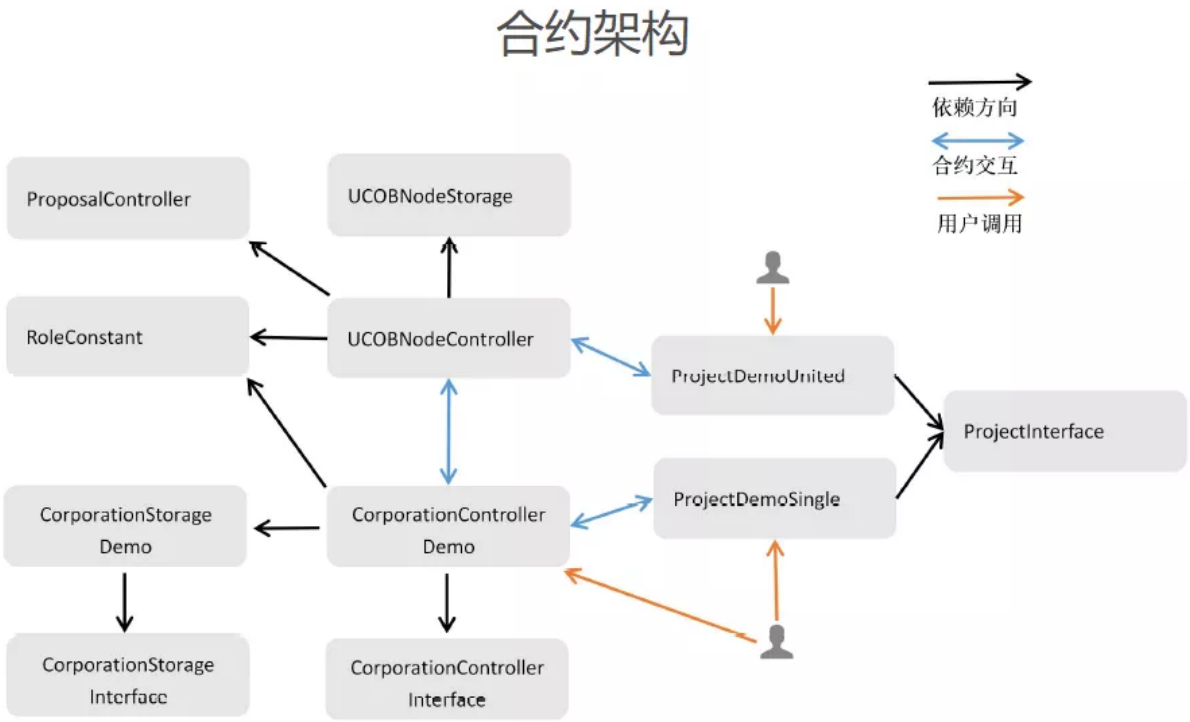
各家公司在区块链上以单独的“公司合约”形式存在，只要实现了“公司合约接口”便可自定义公司内部业务逻辑与内部组织关系。公司想要加入联合公司时，首先提出申请，部署自己的“公司合约”；然后由已在联合公司中的成员以新部署的“公司合约地址”作为参数发起提案；在得到联合公司中大多数成员投票通过后，便可以正式成为联合公司中的一员。各家公司参与的项目将单独以“联合项目合约”的形式存在，联合公司内任一家成员公司都可以发起联合项目。

首先依据“项目合约接口”开发“联合项目合约”，部署到区块链上；并以“联合项目合约的地址”作为提案中的参数，发起提案；联合公司中每一家公司可以根据提案中的合约地址查看合约，决定是否投票该提案；当得到联合公司中大多数成员公司投票通过后，即成为“联合项目合约”。

区块链智能合约设计思路与关键逻辑

合约设计思路

在合约设计上，参考了 FISCO BCOS开源社区《智能合约编写之Solidity的编程攻略》文章里的思路，采用“数据、管理、控制”分层的设计方法。本智能合约方案主要有三大模块：联合治理模块、公司模块、项目模块，合约交互主要发生在这三大模块的合约之间。



- **联合治理模块**：提案与投票系统，联合公司成员管理系统，联合公司间资金流转系统；
- **公司模块**：单个公司管理系统，单个公司内部资金流转系统；
- **项目模块**：多个公司的联合项目管理，单个公司的内部项目管理。

其中，“联盟管理模块”集中管理“公司模块”合约和“项目模块”合约，管理机制主要为“投票-注册”；公司合约、项目合约在实现对应接口合约方法的基础上自定义业务逻辑，并以单独合约的形式上链。

合约功能上，主要有以下几点：

- 投票注册功能，只有投票数超过一定比率，新公司才能成为联合公司一员，新项目才能认定为联合项目；
- 项目管理功能，如项目管理员的设置；
- 基于角色的权限控制，自定义角色和权限；
- 资金流转，包括公司之间的资金流转（涉及跨合约调用）和公司内部的资金流转；
- 资金发行功能，依据投票决定是否发行资金。

关键逻辑的合约代码实现

这里介绍项目中一些关键逻辑的合约代码实现，以“存储类智能合约”的所有权转移为例。本项目采取了“存储、逻辑、控制”分层设计的思路，部署者在部署“存储类智能合约”后必须转移合约所有权关系给控制器类智能合约，存储类合约方法如下：

```
function transferOwnership(address newOwner) public onlyOwner {
require(newOwner != address(0), "Ownable: new owner is the zero address");
emit OwnershipTransferred(_owner, newOwner);
_owner = newOwner;
}
```

上述“newOwner”参数必须为对应的“控制器合约”地址。这样，“存储类智能合约”通过修饰器“modifier onlyOwner()”保证了只有对应的“控制器智能合约”才可以修改“存储类智能合约”的数据。部署完成后在“控制器合约”中通过如下方法可验证是否已具备“存储类合约”的所有权。

```
function checkUCOBNodeStorageSafety() public view returns (bool) {
return ucobNodeStorage.owner()==address(this);
}
```

控制器类智能合约的代码逻辑可以升级，通过投票来决定是否升级。

```
function changeUCOBNodeStorageOwner(bytes32 proposalId) public {
require(proposalPassed(proposalId,...));
...
ucobNodeStorage.transferOwnership(UCOBNodeControllerAddress);
...
}
```

当投票通过后，“存储类智能合约”的所有权关系会转移到新的“控制器合约”地址上，数据不变，但是业务逻辑“升级”了。

22.7.7 打造全场景透明管理，建信筑和联手FISCO BCOS助力建造业数字化

作者：房少君 | 建信筑和 CTO

公众号对话框回复【建信筑和】获取方案PDF

随着工业4.0时代持续推进，区块链逐渐挥别纯粹的技术自赏和金融标签，渗透进各类实体产业。

在传统建造场景下，行业信息不透明、管理协同性差、信息化水平不高等“痼疾”，使得建造项目的推进过程存在管理难、追责难、监管难、信任难的问题。“区块链+建造”的结合，利用区块链技术，瞄准这些行业痛点，提出了应用层面的切实解决方案。

深圳市建信筑和科技有限公司（以下简称：建信筑和）基于FISCO BCOS区块链平台开发的“伊OS透明建造解决方案”，正是区块链技术在建造行业场景化应用的典型代表。

该方案凭借全场景管理模式的领先优势，入选区块链服务网络BSN首批官方指定应用，并在工信部中国电子技术标准化研究院主办的2020年第四届中国区块链开发大赛上，斩获桂冠。

信息不对称+管理难协同，建造业痛点待破局

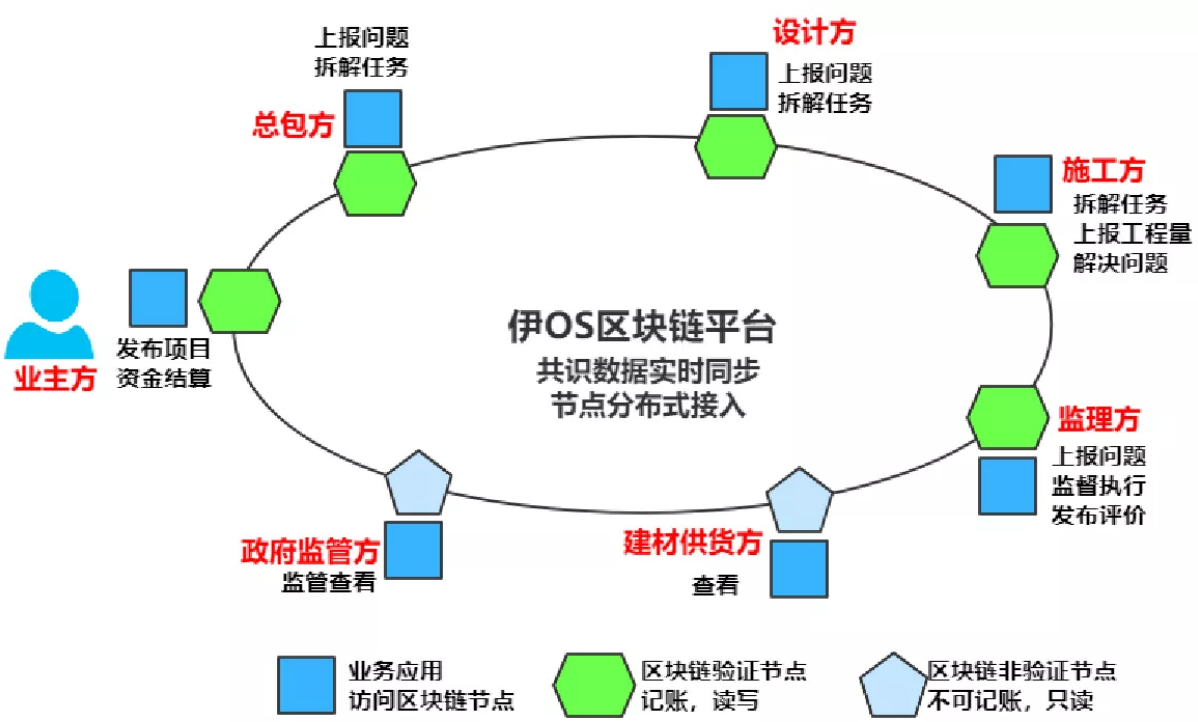
在传统建造业中，“信息不对称+管理难协同”是制约行业发展的重要因素。建造业上下游链条很长，如果产业链上每个环节的参与者之间信息不透明，无形中提高了工作协同的沟通、管理成本。诸如施工方和设计方对于设计方案的扯皮、各主体间款项账期的节点把控模糊等问题层出不穷，拖慢整体工程进度。同时，信息不透明也使甲方难以实现全流程式管理穿透，管理不及时、不到位为项目埋下了违规风险，进一步推高信任成本，反过来又加重了工作协同难度。

在传统行业中，建造业信息化程度一直处于较低水平，建设过程中涉及业主投资方、监管方、代建方、咨询方、设计方、施工方、监理方、运营方等众多单位，同时随着当前各类工程项目投资规模的不断扩大，项目管理和资金监管方面压力大，这些问题制约着建造业生态的健康成长。因此，传统建造业的破局路径之一，就是尽快打破现状，利用信息化手段推动建造业实现项目管理“全场景、全主体、全流程透明协同”，而区块链，则是加速技术化进程的独特手段。

在此背景下，建信筑和加入FISCO BCOS区块链开源生态，基于FISCO BCOS区块链技术打造了面向建造行业的全生命周期管理系统——“伊OS透明建造解决方案”，以推进建造行业实现“全场景透明化管理”。该解决方案运用区块链分布式存储/共享、智能合约等技术，与建造行业已有成熟的IT工具如设计软件、造价系统、智慧工地、BIM系统等进行整合，同时结合大数据和人工智能等先进技术，有效破除了行业痛点，构建了一个具有管理穿透、公开透明、信息共享、信用评价的全过程透明建设平台。

区块链技术助力，建造实现全场景穿透管理

“伊OS透明建造解决方案”采取了场景式存证、数字签名、加密算法、智能合约等区块链技术，打破了原有单线任务框架，实现了去中心化的任务协同和数据流转。将原有项目链条（产业链条）上的参与者，例如项目方、设计、施工、勘察、总包、分包、监理、班组等，置于同一任务场景中，不再具有严格的上下游流程关系，原来极易形成彼此掣肘的“物流、资金流、信息流”也能做到有机协同。



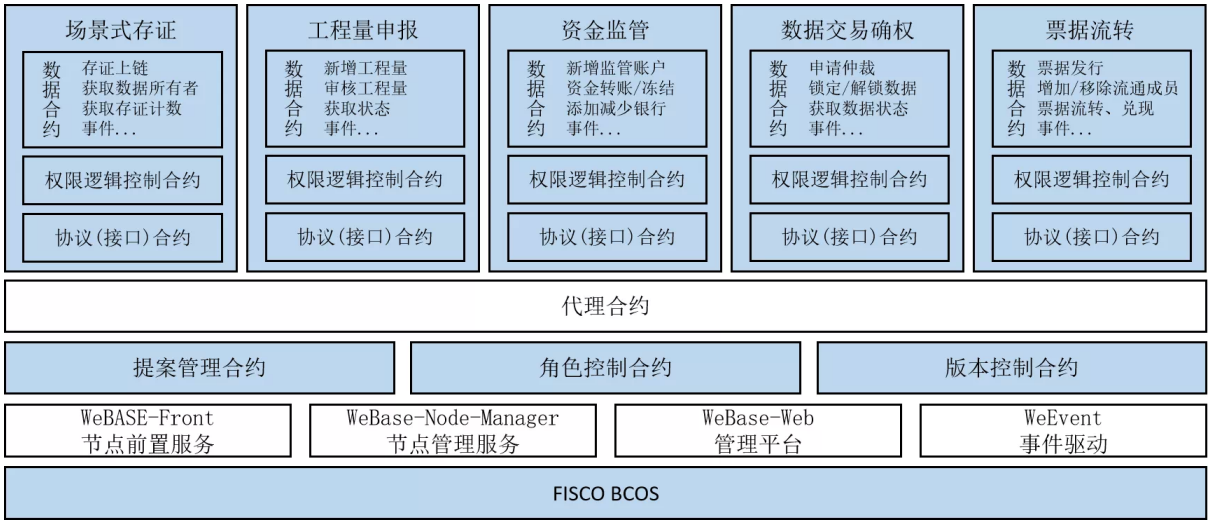
同时由于智能合约触发机制的存在，工程资金流向有了明确的监管方向，资金流转的每一步都可透明，这将进一步确保资金使用的合理、合规、合法。同时，工程资金透明也使得工程质量和安全更有保障，形成建造产业良性循环。

“伊OS透明建造解决方案”不仅能够实现单项目多标段管理，还可以实现多项目管理，目前已被广泛运用在建造行业，有效提升项目运行效率。截止2020年5月底，伊OS平台上运行项目超300个，参建单位达400多家。在某集团在建的100+个管网项目中，伊OS系统协助项目负责人全面的管理现场施工的过程、进度、质量、安全进行，通过奖惩机制成功落地执行，让项目问题解决率从60%提升到80%。

建信筑和CTO房少君表示，区块链技术遇见建造行业，不仅可以对项目全过程实施文件共享、资金监管、工程量申报、质量安全、绩效考核等全场景“俯瞰”，而且能有效点亮原先项目管理、工程结算及资金监管上的“盲区”，“事前任务分布-事中实时监督-事后责任追溯”自此形成了高效协同的有机闭环。

万字报告选定FISCO BCOS，携手重塑建造信用生态

为了有效地打造“伊OS透明建造解决方案”，建信筑和选择了FISCO BCOS。“为了甄选区块链底层平台，我们团队专门做了一份过万字的研究报告，最终选定FISCO BCOS，主要是因为它拥有不可多得的语言优势，生态组件、节点部署和技术支持等都非常丰富、易用和高效。”房少君介绍，在FISCO BCOS技术底层上，建信筑和有效地完成了五大业务场景的系统化构建：场景式存证、工程量申报、资金监管、数据交易确权和票据流转。



基于FISCO BCOS区块链技术的“伊OS透明建造解决方案”能够确保项目实施过程中物流流转和信息流转一一对应、资金流向透明监管、物流全程可追溯和资金到账的高度匹配，不仅是区块链在建造行业中的项目管理流程再造，更是新型“全场景”式管理模式下的信用生态重塑。

社区长期征集基于FISCO BCOS研发的区块链应用，如果您有正在研发或已经上线的应用，欢迎点击“阅读原文”告诉我们，您的应用值得被更多人看见。

22.8 社区开发实践

22.8.1 鲲鹏平台编译并运行FISCO-BCOS 2.6.0

一：申请鲲鹏服务器（已经有鲲鹏服务器略过此步）

1. 注册

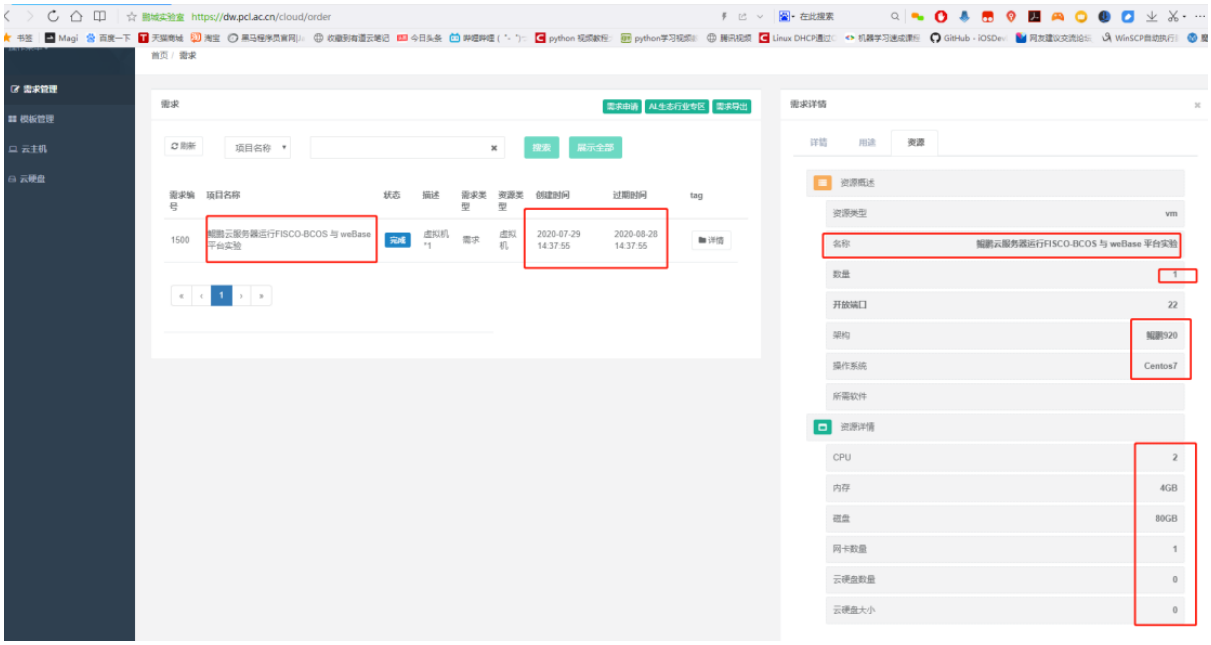
打开 鹏城实验室官网 进行账户注册，注册过程中需要填写真实名称、工作单位、邮箱，收到激活邮件后需要手动点击激活连接地址激活账户

2. 登录

登陆注册的账户以后点击“开发者云”按钮

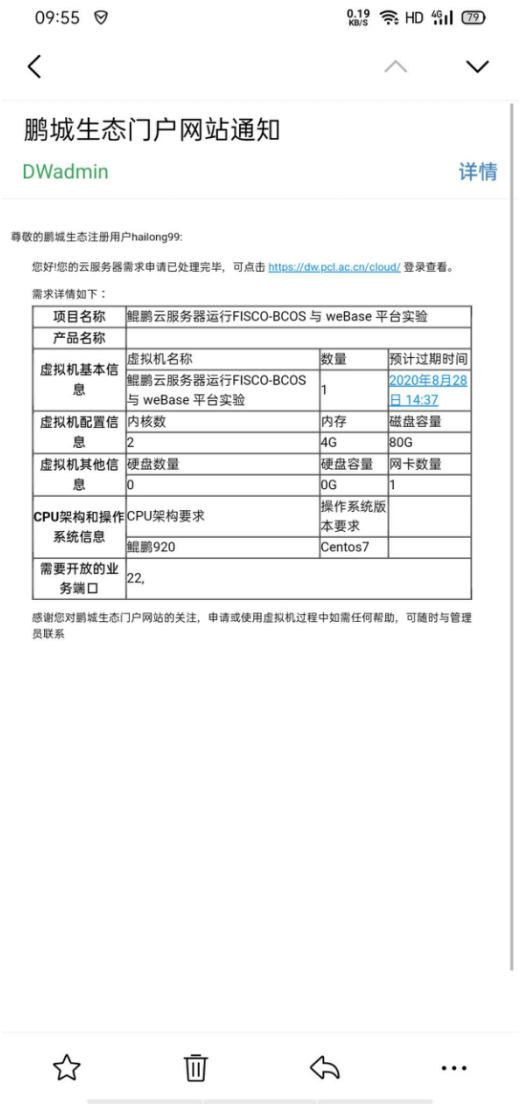
3. 填写申请订单

在打开的页面中自己填写“需求申请”订单，按照自己实际项目需求规划硬件配置、系统版本、使用多长时间



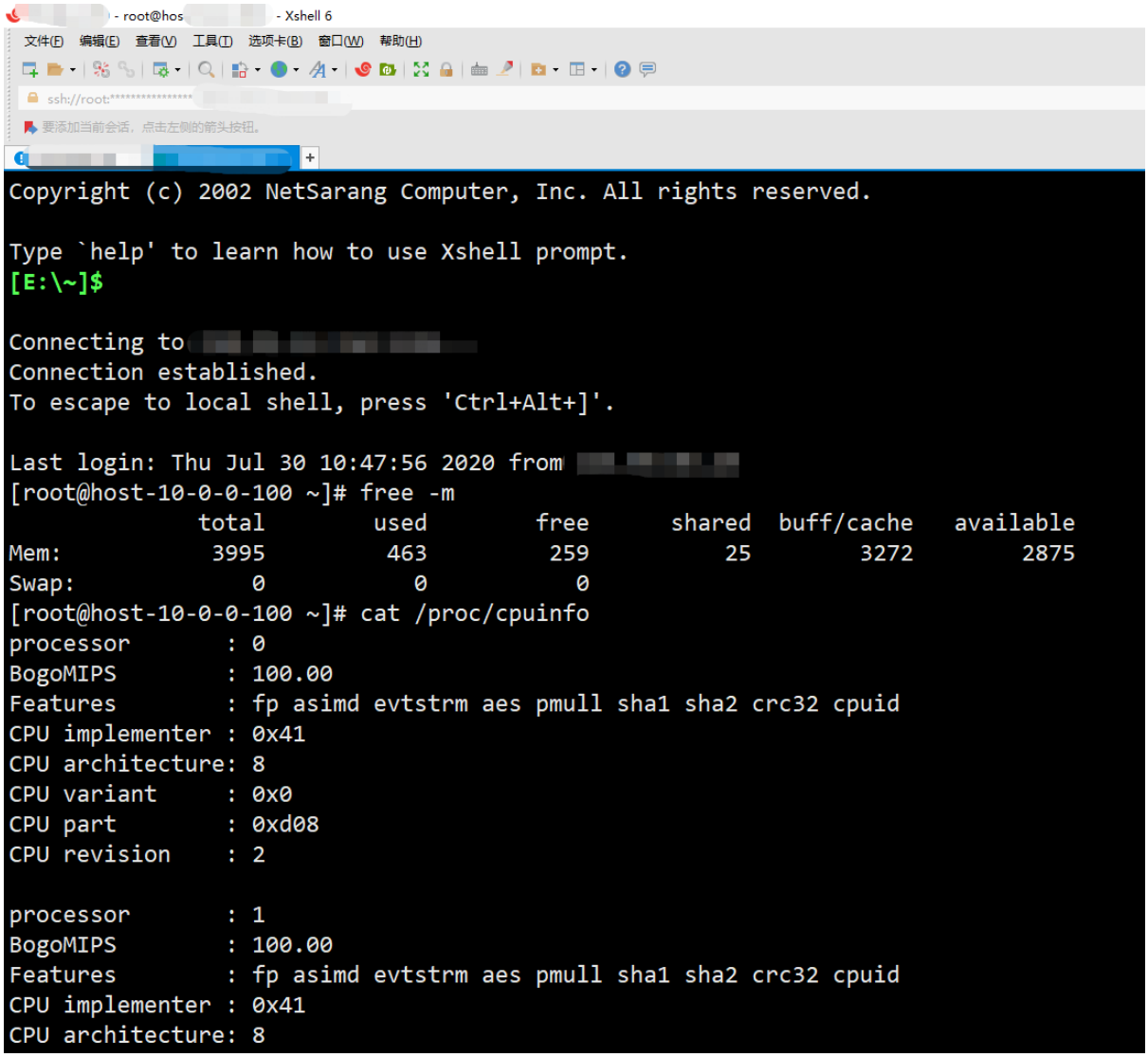
4. 等待通知

需求订单填写无误提交以后，等待官方批准，批准完成以后会收到批准结果邮件通知



5. 通过申请

登录鲲鹏服务器查看服务器信息，至此鲲鹏服务器准备完成

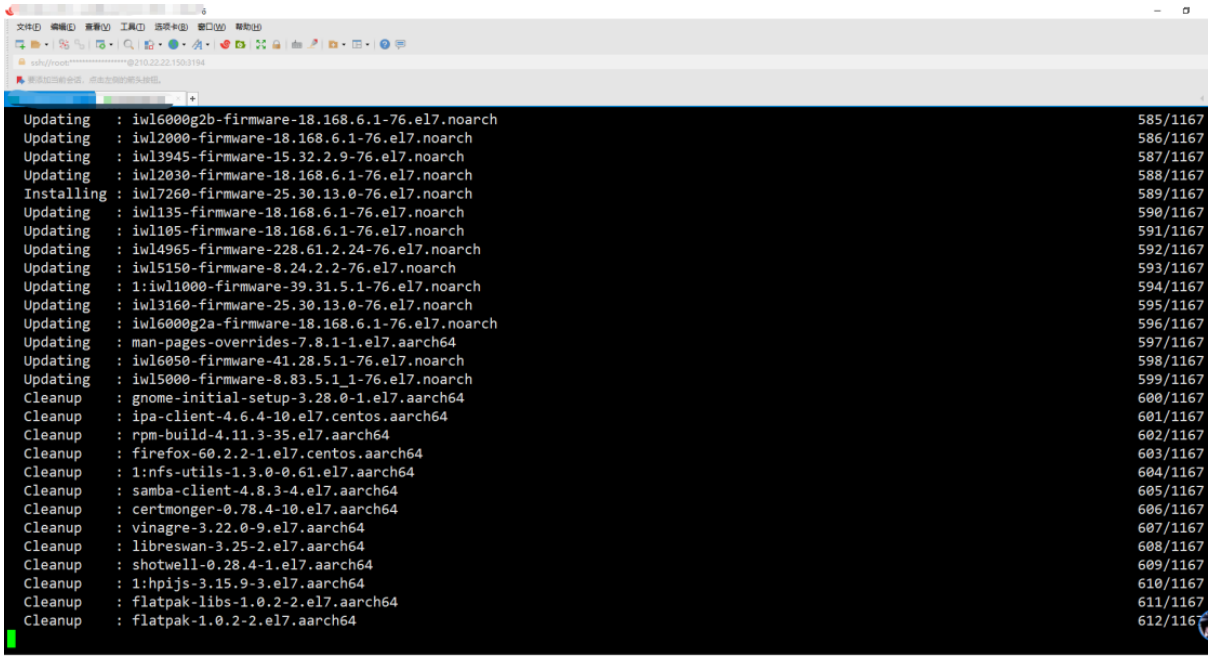


二：在鲲鹏服务器安装基础软件

1. 更新软件

以 Centos 为例，使用 yum 进行更新（此过程会下载更新包，请保持网络畅通并耐心等待）

```
yum update
```

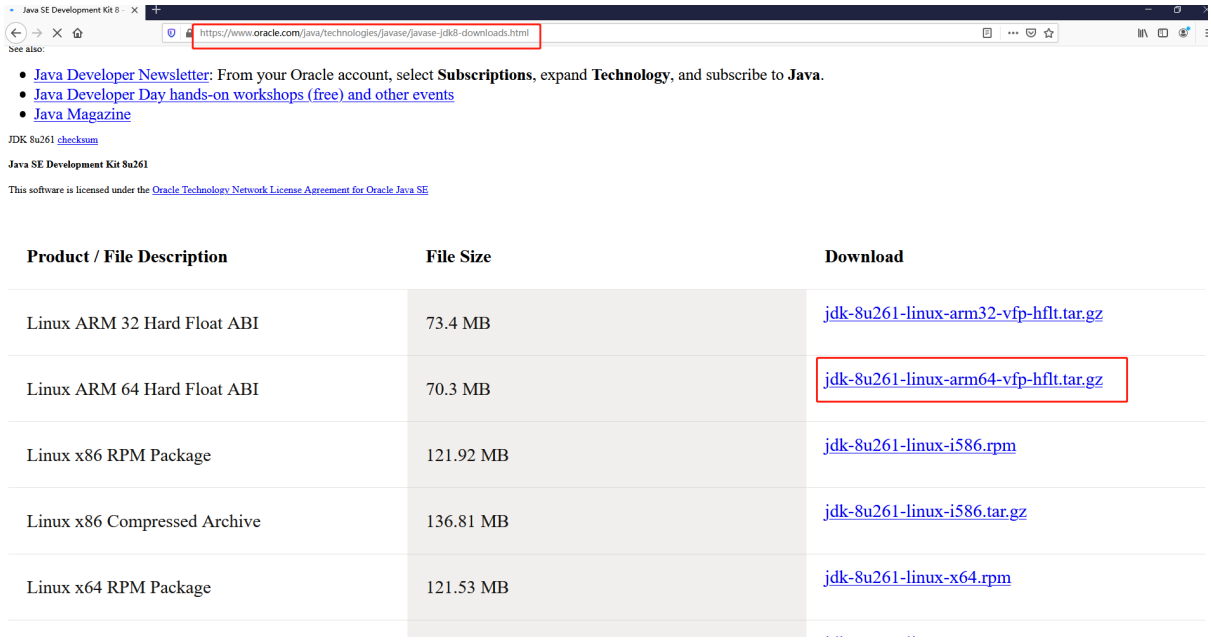


2. 安装基础依赖

```
sudo yum install -y epel-release centos-release-scl
sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git flex patch bison gmp-
↪static devtoolset-7
```

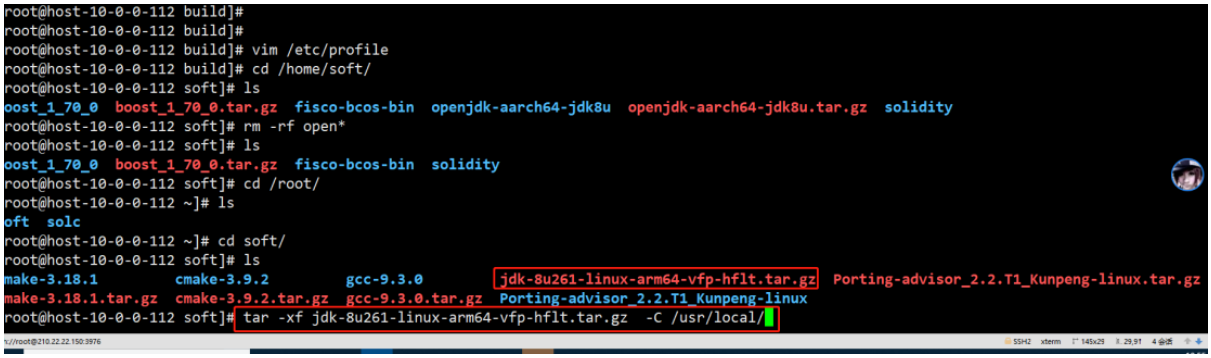
3. 安装鲲鹏版本jdk-1.8

- 安装 jdk 从 oracle官网 下载 jdk-1.8



- 解压文件到/usr/local中

```
tar -xf jdk-8u261-linux-arm64-vfp-hflt.tar.gz  -C /usr/local/
```



- 设置环境变量

```
vim /etc/profile

// 在文件最后添加:
export JAVA_HOME=/usr/local/jdk1.8.0_261
export PATH=$JAVA_HOME/bin:$PATH

//使变量生效
source /etc/profile
```

```
# /usr/share/doc/setup-*/uidgid file
if [ $UID -gt 199 ] && [ "`/usr/bin/id -gn`" = "`/usr/bin/id -un`" ]; then
    umask 002
else
    umask 022
fi

for i in /etc/profile.d/*.sh /etc/profile.d/sh.local ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null
        fi
    fi
done

unset i
unset -f pathmunge



export PATH=$PATH:/usr/local/share/cmake-3.9

#java
export JAVA_HOME=/usr/local/jdk1.8.0_261
export PATH=$JAVA_HOME/bin:$PATH

"/etc/profile" 85L, 1949C
```

三：编译 FISCO-BCOS 源码

1. 下载编译依赖

- FISCO 编译依赖许多的文件，直接从 git 上下载的话会很慢，所以这里先从 gitee 下载相应的依赖，后面编译时，直接进行 copy

```
cd
git clone https://gitee.com/FISCO-BCOS/LargeFiles.git
```

2. 下载源码

- 克隆 FISCO BCOS 代码

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

3. 执行预编译

```
cd FISCO-BCOS
git checkout master
mkdir -p build && cd build

// CentOS请执行此命令，其他系统不需要
source /opt/rh/devtoolset-7/enable

cmake3 .. -DARCH_NATIVE=on
```

4. 执行编译

- 执行 make, 执行到下载文件这里会出现卡顿，这里直接 ctrl + c 取消

```
# make
Scanning dependencies of target jsoncpp
[  0%] Creating directories for 'jsoncpp'
[  0%] Performing download step (download, verify and extract) for 'jsoncpp'
-- Downloading...
   dst='/root/FISCO-BCOS/deps/src/jsoncpp-1.7.7.tar.gz'
   timeout='none'
-- Using src='https://github.com/open-source-parsers/jsoncpp/archive/1.7.7.tar.gz'
-- verifying file...
       file='/root/FISCO-BCOS/deps/src/jsoncpp-1.7.7.tar.gz'
-- Downloading... done
-- extracting...
     src='/root/FISCO-BCOS/deps/src/jsoncpp-1.7.7.tar.gz'
     dst='/root/FISCO-BCOS/deps/src/jsoncpp'
```

5. 复制依赖包到相应目录

```
//如果提示是否覆盖，输入 y
cp ${HOME}/LargeFiles/libs/* ${HOME}/FISCO-BCOS/deps/src
```

6. 继续执行编译

- 首次编译速度比较慢，需要耐心等待

```
cd ${HOME}/FISCO-BCOS/build
make
```

7 解决编译GroupSigLib 报错

- 编译过程中，如果出现编译 GroupSigLib 失败的问题，会出现一下报错

```
[ 24%] Performing configure step for 'GroupSigLib'
-- GroupSigLib configure command succeeded.  See also /root/FISCO-BCOS/deps/src/
↪GroupSigLib-stamp/GroupSigLib-configure-*.log
[ 25%] Performing build step for 'GroupSigLib'
CMake Error at /root/FISCO-BCOS/deps/src/GroupSigLib-stamp/GroupSigLib-build-
↪RelWithDebInfo.cmake:49 (message):
  Command failed: 2

   'make'

  See also

    /root/FISCO-BCOS/deps/src/GroupSigLib-stamp/GroupSigLib-build-*.log

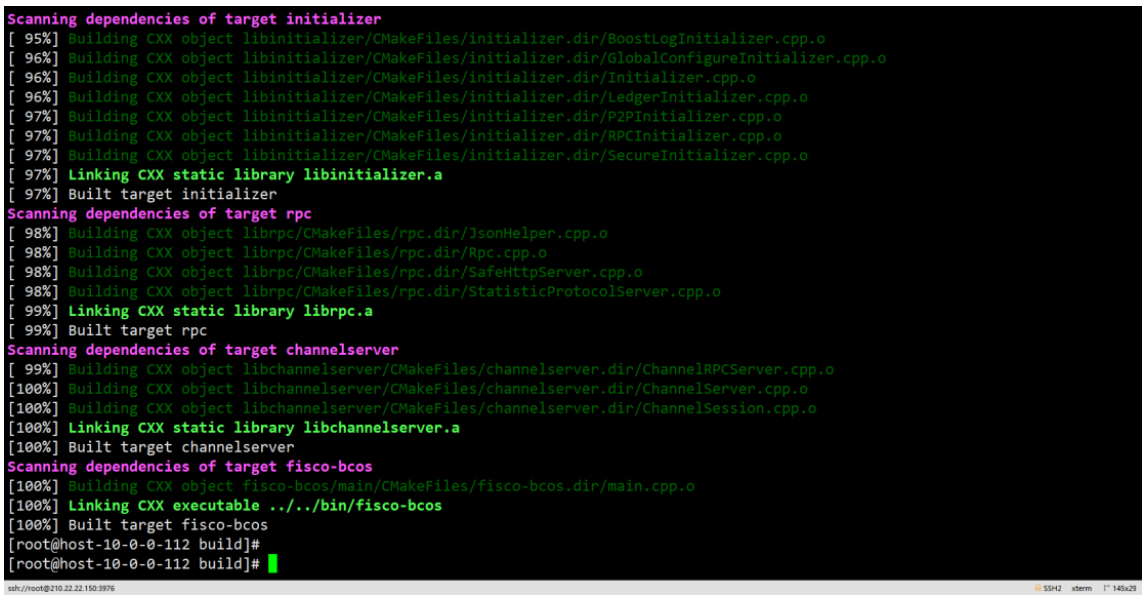

make[2]: *** [CMakeFiles/GroupSigLib.dir/build.make:115: ../deps/src/GroupSigLib-
↪stamp/GroupSigLib-build] 错误 1
```

- 解决方法:

```
cp /usr/share/automake-1.13/config.guess ${HOME}/FISCO-BCOS/deps/src/GroupSigLib/
↪deps/src/pbc_sig/config.guess
```

8. 查看编译结果

- 编译完成效果



- 查看编译结果的版本号

```
# cd bin
# ./fisco-bcos -v
FISCO-BCOS Version : 2.6.0
```

(continues on next page)

(续上页)

```
Build Time         : 20200810 09:17:18
Build Type         : Linux/g++/RelWithDebInfo
Git Branch         : master
Git Commit Hash    : a2c2cd3f504a101fbc5e97833ea0f4443b68098e
```

四：鲲鹏平台运行2群组3机构6节点底层FISCO-BCOS联盟链服务

1. 创建联盟链底层二进制可执行文件存放目录

```
cd
mkdir bin
```

2. 复制编译的fisco-bcos 文件到创建的目录中

```
cp ${HOME}/FISCO-BCOS/build/bin/fisco-bcos bin
```

3. 下载 build_chain.sh 脚本

```
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
```

4. 运行一键搭建2群组3机构6节点底层FISCO-BCOS联盟链服务脚本

```
#  ./build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545 -e bin/fisco-bcos
```

5. 脚本运行完成效果

```
# ./build_chain.sh -l 127.0.0.1:4 -p 30300,20200,8545 -e bin/fisco-bcos
Checking fisco-bcos binary...
Binary check passed.
================================================================
Generating CA key...
================================================================
Generating keys and certificates ...
Processing IP=127.0.0.1 Total=4 Agency=agency Groups=1
================================================================
Generating configuration files ...
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] Output Dir      : /root/nodes
[INFO] CA Path         : /root/nodes/cert/
================================================================
[INFO] Execute the download_console.sh script in directory named by IP to get␣
↪FISCO-BCOS console.
e.g.  bash /root/nodes/127.0.0.1/download_console.sh -f
```

(continues on next page)

(续上页)

```
============================================================
[INFO] All completed. Files in /root/nodes
```

6. 启动节点

```
cd nodes/127.0.0.1/
./start_all.sh
```

7. 查看节点状态

```
# ps -aux |grep fisco
root     17241  1.1  0.7 729792 31168 pts/1    Sl   17:32   0:00 /root/nodes/127.0.
↪0.1/node2/../fisco-bcos -c config.ini
root     17242  1.0  0.7 729792 31360 pts/1    Sl   17:32   0:00 /root/nodes/127.0.
↪0.1/node0/../fisco-bcos -c config.ini
root     17243  1.0  0.7 729792 31168 pts/1    Sl   17:32   0:00 /root/nodes/127.0.
↪0.1/node3/../fisco-bcos -c config.ini
root     17244  1.1  0.7 729408 31296 pts/1    Sl   17:32   0:00 /root/nodes/127.0.
↪0.1/node1/../fisco-bcos -c config.ini
```

8. 查看共识状态

```
# tail -f node*/log/*|grep ++
info|2020-09-04 17:34:19.452001|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=2,hash=40f8c3c5...
info|2020-09-04 17:34:18.449782|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=85ab01ec...
info|2020-09-04 17:34:17.446976|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=d7d2cfa2...
info|2020-09-04 17:34:20.454172|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=3,hash=87efbeb5...
info|2020-09-04 17:34:21.456586|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=4a9c4f2d...
info|2020-09-04 17:34:22.459794|[g:1][CONSENSUS][SEALER]++++++++++++++++␣
↪Generating seal on,blkNum=1,tx=0,nodeIdx=1,hash=d1dd4738...
```

五：鲲鹏平台安装FISCO-BCOS 控制台

说明：控制台程序依赖 java-1.8 需要提前安装好鲲鹏版本（arrch64）的java-1.8

```
# 下载控制台
$ curl -#LO https://github.com/FISCO-BCOS/console/releases/download/v2.7.1/
↪download_console.sh && bash download_console.sh
$ cd console

# 拷贝证书
cp ~/nodes/127.0.0.1/sdk/* conf

# 修改配置文件
# 如果端口没有冲突，直接复制配置文件即可，否则复制之后，修改 config.toml中的network.peers配置项为相应的 channel 端口
$ cp conf/config-example.toml conf/config.toml
```

启动控制台

```
# cd ~/console/ && bash start.sh
=====================================================================================================
Welcome to FISCO BCOS console(2.6.1)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______  _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       \
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$   \
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

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

六：鲲鹏平台通过控制台发送交易

- 查看 FISCO 版本

```
[group:1]> getNodeVersion
{
    "Build Time":"20200904 14:43:15",
    "Build Type":"Linux/g++/RelWithDebInfo",
    "Chain Id":"1",
    "FISCO-BCOS Version":"2.6.0",
    "Git Branch":"master",
    "Git Commit Hash":"a2c2cd3f504a101fbc5e97833ea0f4443b68098e",
    "Supported Version":"2.6.0"
}
```

- 部署并调用合约

```
[group:1]> deploy HelloWorld
transaction hash:␣
↪0xa71f136107389348d5a092a345aa6bc72770d98805a7dbab0dbf8fe569ff3f37
contract address: 0xd22aa109bc0708ad016391fa5188e18d35b16434

[group:1]> call HelloWorld 0xd22aa109bc0708ad016391fa5188e18d35b16434 set "asfdas"
transaction hash:␣
↪0x72f4f8c980fd0d63d57bdbcc89d6b82dda79e301f25a65f0f49726105184b596

[group:1]> call HelloWorld 0xd22aa109bc0708ad016391fa5188e18d35b16434 get
asfdas
```

22.8.2 多机部署-单群组双机构双节点组网模式实战

作者：蒲沧龙(肖越)｜上海对外经贸大学区块链与应用研究中心成员

0.需求分析

有两台服务器，那么每台机子下一个机构生成一个节点，两个连一块，即：双机构双节点单群组。

1.下载安装运维部署工具

这里假设机子上什么都没有，因为使用源码编译客户端的用户不用最后一步

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
```

这里遇到的问题是，二进制文件拉取贼慢。 然后我再本机把generator克隆下来，发现是:

```
def download_fisco(_dir):
    """[download fisco-bcos]

    Arguments:
        _dir {[type]} -- [description]
    """
    dir_must_exists(_dir)
    bin_path = _dir

    package_name = "fisco-bcos.tar.gz"
    # (status, version)\
    #     = getstatusoutput(r'curl -s https://api.github.com/repos/FISCO-BCOS/F
    #                       r'| grep "tag_name" | grep "\"v2\.[0-9]\.[0-9]\""'
    #                       r' | sort -u | tail -n 1 | cut -d \" -f 4 | sed "s/
    # if bool(status):
    #     LOGGER.error(
    #         ' get fisco-bcos verion failed, result is %s.', version)
    #     raise MCError(' get fisco-bcos verion failed, result is %s.' % versio
    version = "2.5.0"
    download_link = 'https://github.com/FISCO-BCOS/FISCO-BCOS/releases/downloa
        version.strip('\n'), package_name.strip('\n'))
    cnd_link = 'https://www.fisco.com.cn/cdn/fisco-bcos/releases/download/v{}/{
        version.strip('\n'), package_name.strip('\n'))
    if Status.use_cdn:
```

工具类的download_fisco函数这里卡主了。没有cdn的朋友可以vim修改下这个url为：

```
fisco官方cdn:
https://www.fisco.com.cn/cdn/fisco-bcos/releases/download/
or直接下载:
https://xiaoyue-blog.oss-cn-hangzhou.aliyuncs.com/fisco-bcos.tar.gz
```

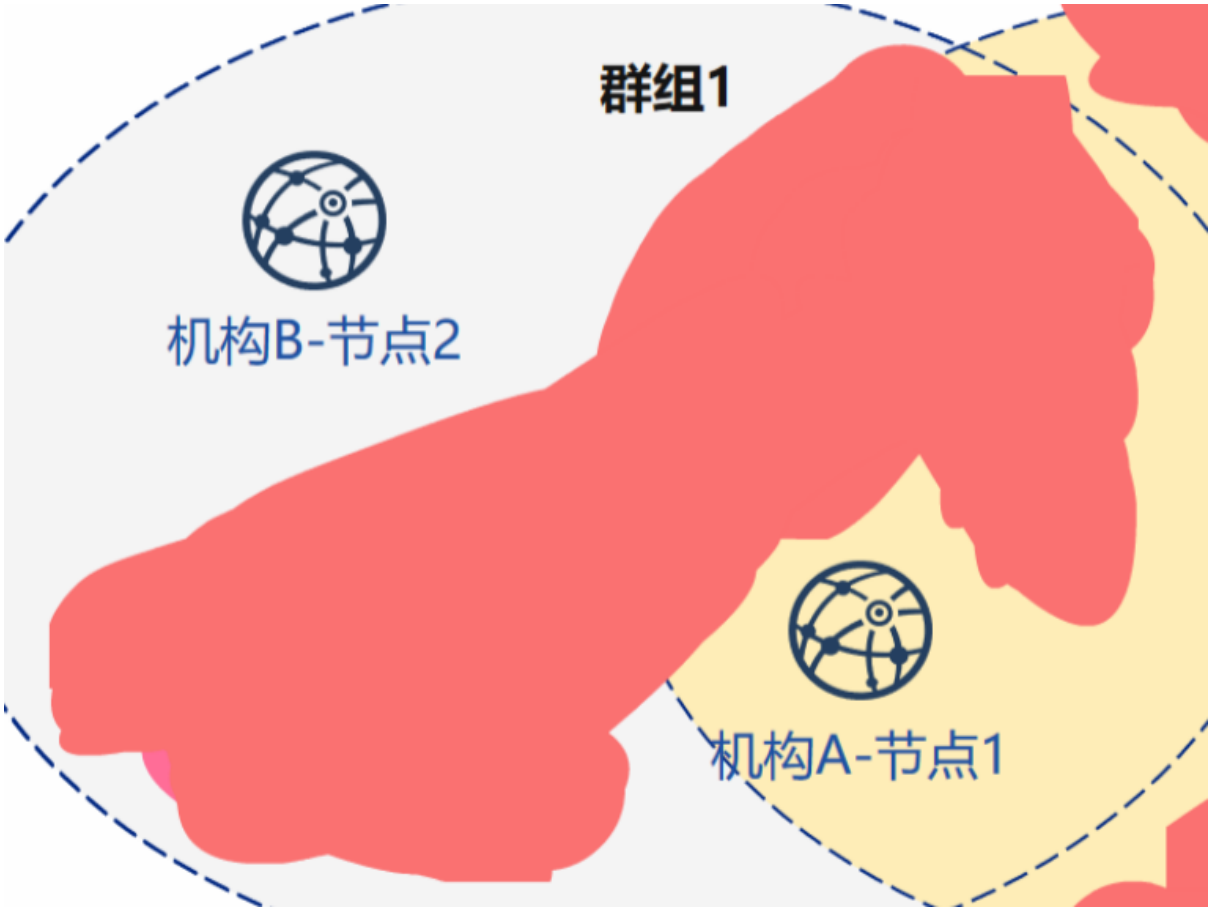
这是俺的OSS，开放使用师傅们轻点敲啊。

一秒不到就下完了。然后这个就是装好了：

```
➜ generator-C git:(master) ✗ ./meta/fisco-bcos -v
FISCO-BCOS Version : 2.5.0
Build Time         : 20200619 06:32:10
Build Type         : Linux/clang/Release
Git Branch         : HEAD
Git Commit Hash    : 72c6d770e5cf0f4197162d0e26005ec03d30fcfe
➜ generator-C git:(master) ✗
```

2.联盟链初始化

来康康拓扑图：



因为官方教程是在一个机子上配的，是节点1,2。分起来的话实际上没有1,2之分的。因为是在两台机子上，也不会存在端口冲突的情况。如果端口没有打开可能会报错，这边建议把两台机子添加白名单就行了。详情可参考：FSICO BCOS多机部署之端口开放

2.1 机构初始化

初始化机构AB

```
cd ~/generator
cp -r ~/generator ~/generator-A
cp -r ~/generator ~/generator-B
```

2.2 链初始化

生成链证书ca.crt&链私钥ca.key

```
./generator --generate_chain_certificate ./dir_chain_ca
```

链证书和链私钥在./dir_chain_ca

3.构建群组

3.1 初始化机构A

生成机构A的证书

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyA
```

证书授权机构将机构证书发送给机构，机构放到meta目录下面

```
cp ./dir_agency_ca/agencyA/* ~/generator-A/meta/
```

3.2 初始化机构B

生成机构B的证书

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyB
```

证书授权机构将机构证书发送给机构，机构放到meta目录下面

```
cp ./dir_agency_ca/agencyB/* ~/generator-B/meta/
```

3.3 机构A修改配置文件

进入机构A修改node_deployment.ini节点配置文件、生成节点配置文件夹。

```
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=you_ip
; 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 seehttps://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=0.0.0.0
channel_ip=0.0.0.0
p2p_listen_port=30300
channel_listen_port=20200
jsonrpc_listen_port=8545
EOF
```

3.4 机构B修改配置文件

进入机构B修改node_deployment.ini节点配置文件、生成节点配置文件夹。

```
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=you_ip_B
; 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.
```

(continues on next page)

(续上页)

```
; You can check accessible addresses of your network card.
; Please seehttps://tecadmin.net/check-ip-address-ubuntu-18-04-desktop/
; for more instructions.
rpc_ip=0.0.0.0
channel_ip=0.0.0.0
p2p_listen_port=30300
channel_listen_port=20200
jsonrpc_listen_port=8545
EOF
```

3.5 机构A生成并发送节点信息

生成机构A节点证书和P2P连接地址文件，根据上面修改的node_depoyment.ini生成

```
cd ~/generator-A
./generator --generate_all_certificates ./agencyA_node_info
```

机构生成节点时需要指定其他节点的P2P连接地址，这里A机构把P2P连接机构发送给了B机构

```
cp ./agencyA_node_info/peers.txt ~/generator-B/meta/peersA.txt
```

3.6 机构B生成并发送节点信息

生成机构A节点证书和P2P连接地址文件，根据上面修改的node_depoyment.ini生成

```
cd ~/generator-B
./generator --generate_all_certificates ./agencyB_node_info
```

因为需要生成创世区块，规定此机构必须要节点证书。B机构除了要发送P2P连接地址之外还要发送节点证书。

```
cp ./agencyB_node_info/cert*.crt ~/generator-A/meta/
cp ./agencyB_node_info/peers.txt ~/generator-A/meta/peersB.txt
```

3.7 机构A生成群组1创世区块

来生成创世区块。这里实际可以通过协商由那个机构生成，不一定是A。

```
cd ~/generator-A
cat > ./conf/group_genesis.ini<< EOF
[group]
group_id=1
;i am xiaoyue
;my blog-s cnmf.net.cn
;yea i love you~
[nodes]
node0=you_ip:30300
node1=you_ip_b:30300
EOF
./generator --create_group_genesis ./group
```

把group1的创世区块发送给B机构

```
cp ./group/group.1.genesis ~/generator-B/meta
```

3.8 机构A生成所属节点

生成机构A的节点

```
cd ~/generator-A
./generator --build_install_package ./meta/peersB.txt ./nodeA
```

启动节点

```
bash ./nodeA/start_all.sh
```

上面有两个需要注意的地方：

1. 在生产节点配置文件和创世区块配置文件时端口一致是没问题的，因为我不在一台机子上做测试，不会存在端口冲突的情况。但这样在复制B机构到B机子上的时候跑不起来就很尴尬。
2. rpc的ip默认是127.0.0.1，如果硬要开启的话会报警告：

```
INFO |  Generate cert by node_deployment.ini end.
WARNING |   Your rpc_ip is 0.0.0.0, this is an unsecurity way
INFO |  Generate cert to ./agencyB_node_info by node_deployment.ini.
```

如果非要开启rpc测试的话，也可以参照上面说的开启防火墙ip白名单。

4.B机构转移并生成节点

压缩：`tar cvf B.tar generator-B` 解压：`tar xvf B.tar` 然后上传下载操作

4.1 生成&启动节点：

```
./generator --build_install_package ./meta/peersA.txt ./nodeB
```

机构B启动节点：

```
bash ./nodeB/start_all.sh
```

Z.查看共识状态

```
tail -f ./node*/node*/log/log*  | grep +++
```

正确回显如下：

```
➜ generator-A git:(master) ✗ tail -f ./node*/node*/log/log*  | grep +++
info|2020-07-21 17:59:57.580270|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=e1fad7f3...
info|2020-07-21 17:59:59.592273|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=d046e6b6...
info|2020-07-21 18:59:58.646905|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=774dfece...
info|2020-07-21 19:59:57.769106|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=2a754fa0...
info|2020-07-21 19:59:59.781228|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=ddcb1871...
info|2020-07-21 20:59:57.579056|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=487cd005...
info|2020-07-21 20:59:59.591176|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=a285c504...
info|2020-07-21 21:59:57.223002|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=4701b57f...
info|2020-07-21 21:59:59.235669|[g:1][CONSENSUS][SEALER]++++++++++++++++ Generating seal on,blkNum=1,tx=0,nodeIdx=0,hash=3ca2bccb...
```

这里还有一个问题。就是上面说的对自己自信不进行测试，导致ip输错了导致共识失败，这时候是没有回显的。把后面的正则删掉就行了。能看到日志报错，通过报错去找不能共识的原因。

你可以从以下方式联系到我：

Blog: https://cnmf.net.cn/

GitHub: https://github.com/xiaoyue2019

CSDN: https://blog.csdn.net/xiaoyue2019

欢迎来俺们社区吹水鸭：

参 考 ： *https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/enterprise_tools/tutorial_detail_operation.html*

22.8.3 开发者如何善用小工具提速应用开发效率

作者：上海对外经贸大学区块链技术与应用研究中心

为什么做区块链开发工具箱?

我们（下文“”我们“均指上海对外经贸大学区块链技术与应用研究中心）注意到，区块链开发者在区块链学习和开发过程中，往往面临以下四个痛点：

- 开发工具繁杂：目前区块链相关的各类开发工具错综复杂，需要占用开发者较多时间熟悉、学习各类工具，影响开发效率。
- IDE功能简单：智能合约开发/区块链开发IDE尚处于早期阶段，功能相对简单，对开发者不够友好。
- 系统学习资料缺乏：当前可供开发者学习的各种资料良莠不齐，且比较零散，缺乏系统性。
- 学习成本高：现阶段智能合约开发/区块链开发IDE更多是面向有一定基础的专业人士，对初学者有一定学习成本。

因此我们有了这样的想法：能不能做一个集成性强、方便快捷的区块链开发工具箱，来解决区块链开发者在学习和开发过程遇到的这些常见问题?

区块链开发工具箱实现思路

相对于传统计算机领域的开发工具，区块链开发存在开发工具多、杂、许多功能需要使用特定的工具或者需要自己手动开发工具等问题。因此，我们研发的区块链开发工具箱共由4个部分组成——1个工具箱+2个子平台+1个CA系统，通过该工具箱集成多种功能，解决开发过程中经常遇到的问题，最终目标是实现使用一个主流编译器（如Remix）+一个开发工具箱即能完成整个区块链应用开发流程。

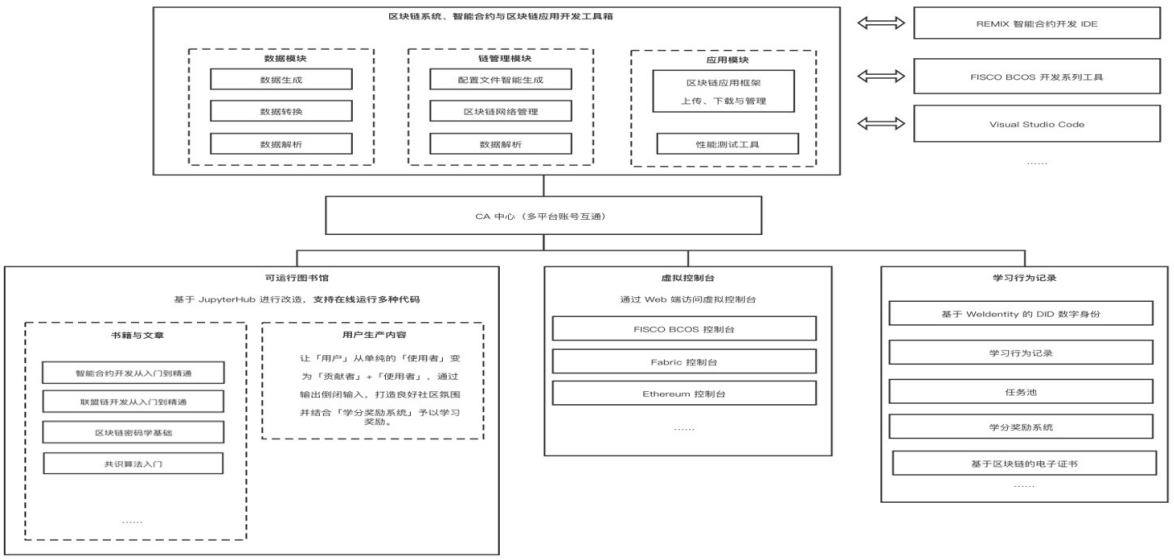
- 1个工具箱：指区块链开发工具箱，该工具箱可配合主流IDE使用，为开发者提供一些在开发过程中经常会用到的服务，如模拟地址生成、数据转换、区块链配置文件智能生成等。
- 2个子平台：分别为多用户可运行图书馆、多链虚拟控制台。多用户可运行图书馆可以为开发者提供在线学习智能合约、联盟链、共识算法等区块链文章与书籍等，同时也支持开发者自行生产内容；多链虚拟控制台则可以帮助开发者通过Web访问各主流区块链的虚拟控制台。
- 1个CA系统：基于WeIdentity 的 CA 系统，负责用户账户管理、用户行为记录与奖励等。

*WeIdentity是一套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，由微众银行自主研发并完全开源。

github地址：https://github.com/WeBankFinTech/WeIdentity

技术文档：https://fintech.webank.com/developer/docs/weidentity/

开发工具箱+可运行图书馆+虚拟控制台功能架构图



区块链开发工具箱主要功能

一起来看一下区块链开发工具箱各组成部分的主要功能。

- **开发者工具箱**

该工具箱力求覆盖整个区块链开发环节，缩短开发周期，提升开发者效率与舒适度。

以FISCO BCOS开发全环节举例：

-启链阶段

区块链应用开发时通常需要先启动一条测试链，此时开发者可使用工具箱＞配置文件智能生成工具，快速生成区块链网络的配置文件。

-区块链网络情况监测

启动测试链后，可通过工具箱＞区块链网络情况监测，一键监测区块链网络是否正常运行。

-合约案例库

编写智能合约前，可使用工具箱＞合约案例库，查找是否有其它合约已实现相同或者相似的功能，避免重复造轮子。

-地址生成、模拟数据生成、数据转换、签名验证

合约开发过程中，通过工具箱提供的各类生成、转换、验证功能进行开发调试。

-性能测试工具

开发完成后，通过工具箱＞性能测试工具，测试代码性能并进行调优。

目前工具箱已实现或规划实现如下功能：地址生成、模拟数据生成、数据转换、签名验证、配置文件智能生成、区块链网络情况监测、合约案例库、区块链网络管理、性能测试工具等。

- **多用户可运行图书馆**

多用户可运行图书馆以JupyterHub为开发基础，集成多种语言内核，如Python、Java等，可支持Ipynb格式图书的上传、阅读或运行书中代码。

它可以帮助用户快速上手区块链开发。例如，某用户想在该图书馆中进行学习，他可以在图书馆公共知识库中找到自己感兴趣的内容，如《联盟链技术入门指南》，并通过文字+可运行代码的方式进行学习和操作；同时也可以将自己的学习笔记提交到公共知识库，反哺社区。

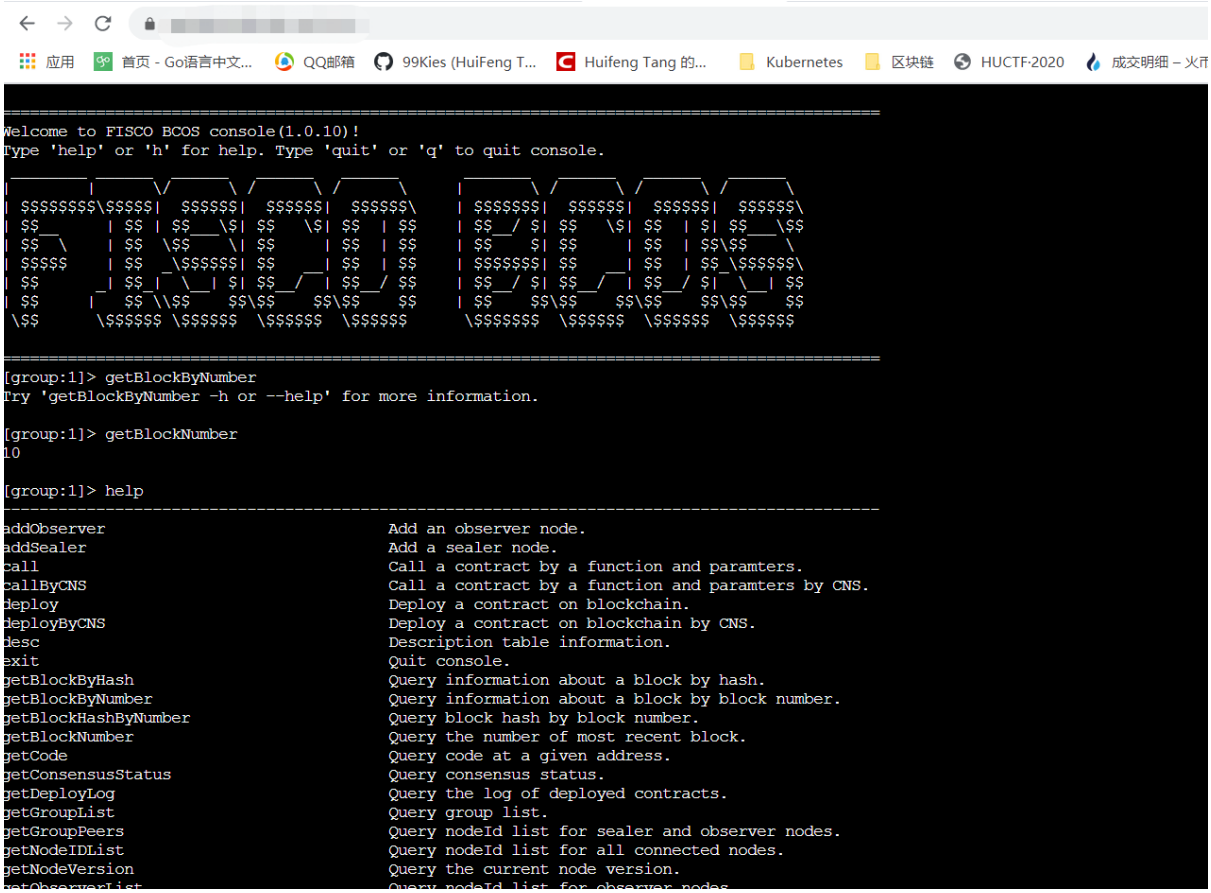
对团队而言，团队可共同运维一个内部共享的图书馆，所有成员可以在该图书馆上共享书籍、文章等，激活群体学习效能，解决孤立学习效率低下、新人培训成本高等问题。

- **多链虚拟控制台**

区块链开发者在初学阶段，并不一定需要自己去建一条链，因为目前能支持快速建链的区块链底层并不多，大部分底层平台建链还是一个比较繁琐的操作。

多链虚拟控制台可以提供这样的功能：开发者可以和他人共同使用一条测试链，通过web访问虚拟控制台并进行开发，当有多个开发者需要通过控制台在同一条测试链进行开发时，只需访问在线多链虚拟控制台即可。

目前已支持通过web访问FISCO BCOS控制台。



- **CA系统**

区块链开发工具箱CA系统基于区块链技术，对用户在多用户可运行图书馆的学习行为、学分奖励、电子证书等进行记录和管理。通过该系统，用户各项学习数据一目了然，结合电子政府和学分奖励系统，可以激励用户更积极地进行学习，反哺社区。

目前已实现或规划实现如下功能：基于WeIdentity的DiD数字身份、用户学习行为记录、学分奖励系统、基于区块链的电子证书等。

这套区块链开发工具箱已完全开源，并贡献给FISCO BCOS开源社区，目前项目正在持续完善中，我们也非常期待社区各路开发朋友能一起拥抱开源，携手共建项目，更多详情欢迎通过项目github主页了解：https://github.com/SUIBE-Blockchain/FISCO_BCOS_Toolbox/

嘉宾Q&A

Q：在做底层技术选型时，您主要考虑哪些因素？

A：一方面，我觉得需要考虑底层技术主导公司的界限感。区块链技术和其它传统技术有一个很大不同是：区块链生态建设不仅要靠主导公司努力，更要靠各方参与，如果底层技术主导公司没有设定好界限，什么都做了，其它人只能承担用户的角色，其实是有悖区块链精神的。

另一方面，底层技术需要有足够的兼容性，因为大家不会把自己绑死在某个底层平台上，所以底层平台所采用的的技术最好不被该平台所独占，例如Solidity智能合约目前很多链都在用，选择区块链底层技术框架的时候，是否支持Solidity就是很重要的参考指标。

抛开上述所说的选型要考虑的两点，FISCO BCOS最大的优势在于战略方向正确。很多细节问题可以持续优化完善，但战略方向是否正确，是这个技术底层平台是否有发展前景的重点。

Q：对于国内开源的发展，您怎么看？

A： 在这次第四届中国区块链开发大赛的参赛过程中，我们在作品准备过程中得到了FISCO BCOS开源社区很好的支持和帮助，这其实就是开源精神的一种体现。目前国内开源发展尚处于起步阶段，给开发者们带来了机遇与红利，但还有很多有意思的开源玩法有待尝试，希望开源爱好者们能一起加油，促进国内开源生态与开源精神的发展。

22.9 社区写作指南

22.9.1 文章贡献选题

基础科普类——分享你对区块链基础概念和原理进行科普和介绍；

开发实操类——分享你的开发、部署、运维等实操经验，包括但不限于踩坑经历和闭坑经验。

应用实践类——如果你对某个场景有深入研究和探索积累，欢迎分享该应用场景下的解决方案与思路。

22.9.2 文章贡献PR操作指引

如果你已经是社区贡献者，可直接按照如下操作完成文章PR提交；如果你初次尝试PR贡献，请点击了解提PR基本操作指引。

1、提PR的预置条件：Fork FISCO-BCOS-DOC到个人github仓库

1. step1：进入注册GitHub账号
2. step2: Fork FISCO-BCOS-DOC 到个人仓库

2、分支说明

PR须提到FISCO-BCOS-DOC的dev分支

3、必备工具：git 点击参考

1. mac系统：brew install git
2. windows系统：点击下载安装
3. ubuntu系统：sudo apt install git
4. centos系统：sudo yum install git

4、主要的git命令：

（以下步骤只需要执行一次即可）

- 将官方的FISCO-BCOS-DOC添加为upstream: git remote add upstream

（以下步骤每次都要操作）

每次提交PR之前，均需要执行以下命令，同步最新的官方文档：

1. 拉取官方文档dev分支的最新文档：git fetch upstream dev
2. 同步官方文档dev分支最新文档到本地：git rebase upstream/dev
   (注：本步骤可能会有冲突，如果有冲突，请解决冲突，点击参考冲突解决方法)
3. 将同步后的文档推到个人的git仓库：git push origin -f

提交个人文档的主要命令

git add, git commit, git push等，点击参考。

5、文档格式说明

1. 须采用markdown的格式编辑文章内容，点击参考markdown语法。
2. (本步骤不强制) 提交PR之前，建议先基于个人仓库的文档构建readthedocs，检查构建出来的文档显示是否符合预期，并在提交PR的时候，附带说明个人构建的readthedocs链接。

点击参考readthedocs构建方法， FISCO-BCOS-DOC的readthedocs配置选项参考下表：

6、Reviewer反馈和合入

提交PR后，Reviewer会将修改意见直接在GitHub上进行反馈，你也可以添加小助手微信FISCOBCOS010进行直接沟通；最后当Reviewer合入PR时，你的文章就被录入啦！

22.9.3 文章PR贡献写作规范

- 开头署名为撰稿人姓名，也可以展现个人所属公司或学校，例如，作者：张三 | FISCO BCOS核心开发者；
- 文章内容尽量包含引言介绍，段与段之间自然衔接和过度，文末总结；
- 文章应保证文句通顺、无语病，表述不会引起读者误解；
- 实操类文章需确保技术点准确无误，并能完成测试跑通；
- 如果文章中涉及到相关技术文档或代码仓库链接，为避免链接失效，请使用FISCO BCOS官方链接。

CHAPTER 23

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。

23.1 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

23.2 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