FISCO BCOS EN Documentation

Release v2.0.0

fisco-dev

Jul 02, 2019

Contents

FISCO BCOS is a reliable, secure, efficient and portable blockchain platform with proven success from many partners and successful financial-grade applications.

- Github homepage
- Insightful articles
- Code contribution
- Feedback
- Application cases
- WeChat group
- WeChat official account

Overview

- To fast build a blockchain system based on FISCO BCOS 2.0, please read Installation
- To deploy multi-group blockchain and the first blockchain application based on FISCO BCOS 2.0, please read Quick Guide
- To know more about functions of FISCO BCOS 2.0, please read Config files and items, Node access, Parallel transactions, Distributed storage, OSCCA computing in Operation Tutorial
- Console： **Interactive command tool** to visit blockchain nodes and check status, deploy or call contract, etc.
- Deployment tool(Generator)： to support operations like building blockchain, expansion, etc., **recommended for business level applications**. You can learn the operation methods in Quick Guide
- Web3SDK： offer APIs for node status, blockchain system configuration modification and nodes to send transactions.
- The detailed introduction of browser is in Browser
- JSON-RPC interface is introduced in JSON-RPC API
- System design documentation: System design

Key features

- Multi-group: Quick Guide Operation Tutorial Design Documentation
- Parallel computing: Operation Tutorial Design documentation
- Distributed storage: Operation Tutorial Design documentation

---

Important:

- This technical documentation is only adaptable for FISCO BCOS 2.0 and above versions. For FISCO BCOS 1.3.x users, please check Ver.1.3 Documentation
- The new features of FISCO BCOS 2.0 are introduced here
- FISCO BCOS 2.0 and its adaptability are illustrated here

CHAPTER 1

Introduction

FISCO BCOS platform is a open source platform to be applicable in financial industry collaboratively built by the FISCO open source working group. Members in the working group include: WeBank, Shenzhen Securities Communication, Tencent, Huawei, Digital China, Forms Syntron, Beyondsoft, Yuexiu Financial Holdings (FinTech), YIBI Technology.

1.1 Sublimation of alliance chain: Collaborative Business and Open Consortium Chain

Business is a competitive and free economic activity. It is naturally easy to result in survival of the fittest, monopoly, and even rent seeking. Especially after the global financial crisis in 2008, the appeared malady of "Too Big to Fail" led to a series of technological and business revolution, and started an era of "centralized" to "distributed".

In this context, blockchain technology was sprouted in 2008 and gradually became matured. Through the consensus mechanism, distributed ledger, encryption algorithm, smart contract, peer-to-peer communication, distributed computing architecture, distributed storage, privacy protection algorithm, cross-chain protocol and other technical modules in blockchain solution, blockchain technology can help the participating parties in business model to achieve the cooperation of reciprocity and mutual trust so that to promote the progress from "information Internet" to "trusted Internet", and also to make business model moving toward "Collaborative" possible.

"Collaborative Business" model defined by WeBank, is a new type of production relationship established by a number of peer-to-peer commercial interests communities and a new economic activity to implement organization and administration, functional division, value exchange, joint provision of products and services, and benefit share through pre-set transparent rules. The main features of Collaborative Business are multi-participation, resources share, intelligent collaboration, value integration, mode transparentizing, cross boundaries, and so on. A mature scenario of Collaborative Business has requirements for production materials to be held by multiple parties, product and service capabilities to be built by multiple parties, mutual relationship in business, and rules of product and profit-sharing to be transparent.

The biggest difference between Collaborative Business and the business models of chain and franchise store and shared business, which are prevailing previously, is not the person, product, or information platform but the objective technology to be the bridge to connect each other. It is true that if not to encourage open source technology, it may evolve into another kind of monopoly. Therefore, for developing Collaborative Business, we must always maintain the attitude of open source technology. All the participants cooperating with each other through open source community is able to eliminate monopoly. This will help small and micro enterprises to

achieve business value, thereby stimulating economic growth, employment and innovation, and realizing anti-monopoly.

To develop open source blockchain technology is significant, but the choice of technology route is also crucial. The primitive blockchain technology originates from virtual currency and public chain projects; however, the public chain projects always make for financing and their users targeted on the profit made by trade, so everyone pays more attention to the coin price but not the blockchain application ability.

Since the tokens of public chain are essentially "class currency" and "class security", they have been severely suspended by regulatory. After they are washed out, alliance chain technology has shouldered the responsibility of pushing the blockchain technology to move forward.

In 2018, "Open Consortium Chain" was announced in the industry. The alliance chain was appealed to actively encourage open source to move alliance from close to public, so that people can really feel the benefits of good experience, increased efficiency, cost reduction, trust enhancement, data interchange, and traceability of responsibility brought by blockchain. And the vision of Collaborative Business is realized.

A new generation of Open Consortium Chain proposes some new requirements on blockchain underlying technology. Except the standard blockchain characteristics, there are still several aspects that need to be strengthened:

First, because of Open Consortium Chain is not a single chain, it needs the technology to support multi-chain parallel and cross-chain communication, and at the same time, it needs the ability to support the massive transaction request from the Internet.

Second, it needs the ability to form alliances and build chains quickly and at low cost, so that each demand party can efficiently establish an alliance chain network to make the cooperation of chain building among enterprises becomes simple as building a "chat group".

Finally, it needs to be opened to achieve full trust among alliance members.

Open Consortium Chain is not only conducive to reducing the cost on trial and error for enterprise, but also promoting the development of the commercial society in the direction of credibility and transparency, and comprehensively reduces the risks of operation, moral hazard, credit, information protection, etc during cooperation.

Adhere to our above goals and vision, we officially release the FISCO BCOS 2.0 version based on the "Open Consortium Chain" technology route.

1.2 FISCO BCOS 2.0

FISCO BCOS 2.0 version has been upgraded and optimized on the basis of the original version, and has made major breakthroughs in scalability, performance, and ease for use, including:

- Implement **group architecture**. In a global network of multiple nodes, there may be much sub-networks which are made up by multiple subsets of nodes to maintain a separate ledger. The consensus and storage among these ledgers are independent of each other and have good scalability and security. In group architecture, parallel expansion can be better achieved to meet the needs of high frequency transaction scenario for financial industry. Meanwhile, group architecture can quickly support the requirements of chain building. It greatly reduces the difficulty of operation and maintenance, and truly enables the chain building among enterprises is as easy as building a "chat group".
- Support **Distributed Storage** to enable storage to break through the limitation of single machine and support lateral spreading. Separating between calculation and storage can improve system's robustness to prevent data from being affected even if the node executive server is failure. Distributed storage defines a standard data accessing CRUD interface which can adapt to multiple storage systems and support both SQL and NoSQL data management methods to support multiple business scenarios.
- Implement **pre-compiled contract framework** to break the EVM performance bottleneck. It supports transaction concurrent processing to greatly increase the transaction processing throughput. Precompiled contract is implemented in C++ and built into the underlying system. Blockchain can automatically recognize the exclusive transaction information of calling contract, construct the DAG dependency, and plan an efficient parallel transaction execution path. In the best case, the performance is increased by N times (N = CPU cores).

- In addition, FISCO BCOS 2.0 version is continue to optimize network transmission models, computing storage processes, and so on for greatly improving performance. For the architecture, it is continuous to upgrade the high availability and ease for use in terms of storage, network, and computing. The core modules are continuous to remodel and upgrade for ensuring the system's robustness.

More 2.0 version features will be introduced in depth in the following sections. Please see [2.0 New Release] (./what_is_new.md).

1.3 FISCO BCOS 1.0

Looking back at the evolution of FISCO BCOS, we have been working on the balance of performance, security, availability and compliance.

- In terms of performance, FISCO BCOS has made a lot of optimizations on overall architecture and transaction processing such as the use of efficient consensus algorithms, paralelizing computing, reducing repetitive computation, upgrading critical computing units, and so on. Furthermore, the core breakthrough of its performance is not only in single-chain, but also in optimizing architecture design based on single-chain performance, and achieving the flexible, efficient, reliable, and secure parallel computing and parallel scalability capability. It can help developers to achieve the performance they need for their business scenarios by simply adding machines. In general, the FISCO BCOS platform optimizes the network communication model, adopts the Byzantine Fault Tolerance(BFT) consensus mechanism, and combines with the multi-chain architecture and cross-chain interaction scheme to solve the performance problems of concurrent access and hotspot accounts so that to meet the need of high frequency transaction scenario for financial industry.
- In terms of security, the FISCO BCOS platform achieves comprehensive security in the application, storage, network, and host layers through node admission control, reliable key management, and flexible access control. In the design of privacy protection, FISCO BCOS platform supports permission management, physical isolation, and national cryptography algorithm (standard algorithm certified by national cryptographic bureau). Meanwhile, it implements multiple privacy protection algorithms including homomorphic encryption, zero knowledge proof, group signature, ring signature, etc. as open projects to public.
- In terms of usability, FISCO BCOS is designed to run at 7 * 24 hours to achieve a high availability needed for financial grade. In terms of regulatory, it supports regulatory and auditing agencies can join as observation nodes to obtain real-time data for regulating and auditing. In addition, it provides multiple development interfaces to make it easier for developers to compile and call smart contract.

1.4 Summary

In practice, FISCO BCOS has grown into a stable, efficient and secure blockchain underlying platform through testing in many external agencies, multiple applications, and longstanding running in production environments.

The subsequent content of this document will detail the tutorial of construction, installation, smart contract deployment and calling of FISCO BCOS 2.0 version, as well as an in-depth introduction to the overall architecture of FISCO BCOS 2.0 and the design of each module.

CHAPTER 2

New features in 2.0 version

2.1 Group architecture

Group architecture is the main one of many new features in FISCO BCOS 2.0, which is inspired by the group chat mode that everyone is familiar with. The group building is very flexible, so a few persons can quickly build a topic group to communicate. The same person can participate in multiple groups, and send and receive messages in parallel. The existing groups can continue to add members.

In a network with group architecture, depending on different service scenario, there may be multiple different ledgers exist. The blockchain node may select a group to join according to the business relationship, and participate in the process of data sharing and consensus of the corresponding ledgers. The characteristics of the architecture are:

- Each group independently implements the consensus process, and the participants in the group decide how to conduct consensus. The consensus within one group is not affected by other groups. Each group has an independent ledger to maintain its own transaction and data, so that groups can be decoupling each other to operate independently, and achieve a better privacy isolation;
- The agency's nodes need to be deployed only once, and can participate in different multi-party collaboration services through group settings, or divide one service into groups according to user, time and other dimensions. The group architecture can be rapidly expanded in parallel. While expanding the scale of business, it greatly simplifies the complexity of operation and reduces the management cost.

For more group introductions, please refer to Group Architecture Design Document and Group Usage Tutorial

2.2 Distributed storage

FISCO BCOS 2.0 has added a support for distributed data storage, which allows nodes to store data in remote distributed systems for overcoming the limitations of localized data storage. This program has the following advantages:

- Support multiple storage engines, select high-availability distributed storage systems, and support data expansion easily and quickly;
- Isolating calculations and data, so node failures will not cause data anomalies;
- Data can be stored in a more secure quarantine area like remote end, which makes sense in many scenarios;

- Distributed storage not only supports the Key-Value form, but also supports the SQL method for making business development easier;
- The storage of world state has changed from the original MPT storage structure to distributed storage for avoiding the problem of the performance degradation caused by the rapid expansion of world state;
- Optimize the structure of the data storage for saving storage space.

Meanwhile, FISCO BCOS 2.0 is still compatible with the local storage mode in 1.0 version. For more information on storage, please refer to Distributed Storage Operations Manual.

2.3 Parallel computing model

In 2.0 version, a parallel processing mechanism for contract transactions has been added to further increase the concurrent throughput of contracts.

In 1.0 version and most of traditional blockchain platforms, transaction is packaged into a block, and executed serially in the transaction sequence in a block. In 2.0 version, based on pre-compiled contracts, a set of parallel transaction processing models has been implemented. Based on this model, transaction mutex variables can be customized. During the block execution process, based on the transaction mutex variable, the system will automatically build a transaction dependency graph–DAG, what to execute the transaction in parallel based on, can the performance increase several times (depending on the number of CPU cores).

For the introduction to more parallel computing models, please refer to the parallel transaction's Design Document and User Manual.

2.4 Precompiled contract

FISCO BCOS 2.0 provides a pre-compiled contract framework that supports compiling contracts in C++. The advantages are faster contract calling, faster running, less resources consuming, and easier computing in parallel. These greatly improve the efficiency of entire system. FISCO BCOS has built-in multiple system-level contracts that provide access control, permission management, system configuration, and CRUD-style data access etc.. These features are naturally integrated into the underlying platform without manual deployment.

FISCO BCOS provides standardized interfaces and examples to help users with secondary development for making it easy for users to compile high-performance business contracts, and to easily deploy them to FISCO BCOS. The pre-compiled contract framework is compatible with the EVM engine to form a "dual-engine" architecture. The users who are familiar with the EVM engine can select to combine Solidity contracts with pre-compiled contracts to achieve significant efficiency while meeting business logic.

In addition, CRUD and the similar operations are also implemented by precompiled contracts. For more introductions to precompiled contracts, please refer to Precompiled Design Documentation and Precompiled Contract Development Documentation.

2.5 CRUD contract

FISCO BCOS 2.0 has added a contractual interface specification that conforms to the CRUD interface for simplifying the cost of migrating mainstream SQL-oriented business applications to blockchain. The benefits are:

- Similar to the traditional business development model, so it reduces the cost of contract development learning;
- Contracts only need to care about the core logic. Storage and calculation are separated to facilitate contract upgrading;
- The underlying logic of CRUD is implemented based on pre-compiled contracts. Data storage adopts distributed storage. These make more efficient;

Meanwhile, 2.0 version is still compatible with the contract in 1.0 version. For more introduction to CRUD interface, please refer to CRUD interface Manual.

2.6 Console

FISCO BCOS 2.0 has added console as an interactive client tool in FISCO BCOS 2.0.

The console installation is simple and convenient. After simple configuration, console can communicate with the chain node. Console has rich commands and good interactive experience. Users can query the blockchain status, read and modify configuration, manage blockchain nodes, deploy and call contract through console. Console helps users to manage, develop, operate and maintain blockchain, and to reduce the complicated operation and the threshold of use.

Compare to traditional scripting tools such as nodejs, console is simple to install and has better experience. For details, please refer to Console Manual.

2.7 Virtual machine

In 2.0 version, the latest Ethereum virtual machine version has been introduced for supporting Solidity 0.5 version. At the same time, the EVMC extension framework is introduced for supporting the expansion of different virtual machine engines.

The underlying internal integration supports interpreter virtual machines. It can be extended to support the virtual machines such as WASM/JIT in the future.

For more information on virtual machines, please refer to [Virtual Machine Design Document] (./design/virtual_machine/index.html)

2.8 Key management service

In 2.0 version, disk encryption has been remodeled and upgraded. When the disk encryption function is enabled, relying on KeyManager service for key management causes more secure.

KeyManager is released on Github open source. The interaction protocol between the node and KeyManager is open. It supports agency to design the KeyManager service that conforms to its own key management specification, such as hardware encryption technology.

For more details, please refer to Manual Document and Design Document.

2.9 Admission control

In 2.0 version, we have remodeled and upgraded the admission mechanism, including the admission of network access and group access to securely control chain and data access in different dimensions.

We adopt new permission control system, design access permission based on list, and support CA blacklist mechanism, which can shield the evil/failed nodes.

For details, please see Admission Mechanism Design Document

2.10 Asynchronous event

In 2.0 version, the mechanisms such as transaction on chain asynchronous notifications, blocks on chain asynchronous notifications, and custom AMOP message notifications are supported together.

2.11 Module remodeling

In 2.0 version, we have remodeled and upgraded the core modules to perform modular unit testing and end-to-end integration testing. The automated continuous integration and continuous deployment is supported.

CHAPTER 3

Compatibility

FISCO BCOS 2.0.0-rc3

New features

- Distributed storage (Operation Manual)
- CRUD interface (Operation Manual)

Change description, compatibility and upgrade instructions

- FISCO BCOS v2.0.0-rc3

FISCO BCOS 2.0.0-rc2

New features

- Parallel computing model (Operation Manual) (Operation Tutorial)
- Distributed storage (Operation Manual)

Change description, compatibility and upgrade instructions

- FISCO BCOS v2.0.0-rc2

FISCO BCOS 2.0.0-rc1

New features

- Group architecture (Operation Tutorial) (Design Document)
- Console (Installation) (Operation Manual)
- Virtual machine
- Compile contract (Operation Manual)
- CRUD contract (Operation Tutorial)
- Key management service (Operation Manual)
- Admission control (Operation Manual)

Change description, compatibility and upgrade instructions

- FISCO BCOS v2.0.0-rc1

FISCO BCOS 1.x Releases

FISCO BCOS 1.3 version:

- 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 version:

- FISCO BCOS 1.2.0 Release

FISCO BCOS 1.1 version:

- FISCO BCOS 1.1.0 Release

FISCO BCOS 1.0 version:

- FISCO BCOS 1.0.0 Release

FISCO BCOS preview version:

- FISCO-BCOS 1.5.0 pre-release

View node and data versions

- View node binary version: `./fisco-bcos --version`
- Data format and version of communication protocol: to get it via 'supported_version' configuration item in the configuration file config.ini

---

3.1 v2.0.0-rc3

---

v2.0.0-rc2 upgrades to v2.0.0-rc3

- **Compatible upgrade** : Directly replace the binary of the v2.0.0-rc2 node with rc3 binary. The upgraded version fixes bugs in v2.0.0-rc2 but does not enable the new features in v2.0.0-rc3. **after upgrading to v2.0.0-rc3, cannot roll back to v2.0.0-rc2**
- **Full upgrade** : Refer to Install to build new chain and resubmit all historical transactions to the new node. The upgraded node contains the new features in v2.0.0-rc3.
- v2.0.0-rc3 Release Note

3.1.1 Change description

New features

- Distributed storage: [The new support for underlying connecting to MySQL directly through database connection pool] (../manual/distributed_storage.html#id2)
- Distributed storage: [The new support for RocksDB engine, which to be used for storage by default when building new chain] (../manual/configuration.html#id14)
- Distributed storage: The new support for CRUD interface. The console in 1.0.3 version or above provides class SQL statements to read and write blockchain data

Updates

- complete the ABI decoding module
- modify the error codes in precompiled contract and RPC interface and unify them to negative number
- optimize the storage module; increase the cache layer and support to configure cache size
- optimize the storage module; allow to submit block in pipelining. You can configure `[storage] . max_capacity` to control the memory size that is allowed
- move the distributed storage configuration item `[storage]` from the group genesis file to the group ini configuration file
- the default storage is upgraded to RocksDB and still supports the old version of LevelDB
- adjust the splicing logic of transaction mutex variables to improve the degree of parallelism of the transactions between different contracts

Fix

- fix the abnormal termination that may occur when CRUD interface contract opens parallel

3.1.2 Compatibility note

RC3 **Forward Compatibility**. The older versions can directly upgrade by replacing program, but they cannot launch the new features for this release. If you need to use the new features, you need to build chain again.

3.2 v2.0.0-rc2

---

v2.0.0-rc1 upgrade to v2.0.0-rc2

- **Compatible upgrade** : Directly replace the binary of the v2.0.0-rc1 node with rc2 binary <https://github.com/FISCO-BCOS/FISCO-BCOS/releases/download/v2.0.0-rc2/fisco-bcos.tar.gz>‘_ . The upgraded version fixes bugs in v2.0.0-rc1 but does not enable the new features such as parallel computing, distributed storage, etc. in v2.0.0-rc2. **after upgrading to v2.0.0-rc2, cannot roll back to v2.0.0-rc1**
- **Full upgrade** : Refer to Install <../installation.html>‘_ to build new chain and resubmit all historical transactions to the new node. The upgraded node contains the new features in v2.0.0-rc2.
- v2.0.0-rc2 Release Note

3.2.1 Change description

New features

- Parallel computing model: Parallel contract development framework, Parallel Transaction Executor (PTE)
- Distributed storage: AMDB, SQLStorage

Optimization

- optimize the logic of block packing transaction number, and dynamically adjust the number of block packing transactions according to the execution time.
- optimize the process of block synchronization to make block synchronization faster
- optimize the codec of the upcoming transaction, the verification of the transaction and the coding of disk in parallel
- optimize the logic of transaction executing return code to make return code more accurate
- upgrade storage modules to support concurrent reading and writing

Other features

- add network data packet compression
- add compatibility configuration
- add chainID and group ID to the transaction code
- add binary cache in transaction
- add timestamp information in gensis block
- add some precompile demos
- support using Docker to build chain
- delete unnecessary logs
- delete unnecessary and repeat operations

Bug fix

- the bug of program exiting caused by asInt abnormity when processing parameters in RPC
- the bug in which the transaction has not been processed in pool when the transaction executing 'Out of gas'
- the bug that can be replayed with the same transaction binary between different groups
- the problems of performance degradation caused by 'insert' operation
- some stability problems have been fixed

3.2.2 Compatibility note

3.3 v2.0.0-rc1

---

v1.x upgrades to v2.0.0-rc1

- **v2.0.0-rc2 is not compatible with v1.x so v2.0.0-rc1 cannot directly parse the historical block data generated by v1.x**, but the old data can be recovered by performing historical transaction on the new chain at v2.0.0-rc1
- **build 2.0's new chain** : Refer to install
- v2.0.0-rc1 Release Note

3.3.1 Change description

Architecture

1. Add group architecture: each group has independent consensus and storage. System throughput can be lateral spreading based on lower operation cost.

2. Add distributed data storage: supports nodes storing data in remote distributed systems to achieve computing and data isolation, high-speed capacity expansion, and data security level enhancement.
3. Add support for precompiled contracts: the underlying implements pre-compiled contract framework based on C++, is compatible with the solidity calling method, and improves the performance of smart contract execution.
4. Introducing evmc extension framework: support for extending different virtual machine engines.
5. Upgrade remodeling P2P, consensus, sync, Transaction execution, transaction pool, block management module.

Protocol

1. Implement a set of CRUD basic data access interface specification contract. To compile business contracts based on CRUD interface to implement traditional SQL oriented business development process.
2. Support mechanisms such as transaction asynchronous notification, block putting on chain asynchronous notification, and custom AMOP message notification.
3. Upgrade Ethereum virtual machine version and support Solidity 0.5.2 version.
4. Upgrade RPC module.

Security

1. UpgradeDisk encryption and provide private key management service. When the disk encryption function is enabled, to manage paivate key depands on KeyManager service.
2. UpgradeAdmission mechanism. Through introducing the network access mechanism and the group access mechanism, to control the access of chain and data in different dimensions.
3. UpgradeAuthority control system. Design access permission based on table

Others

1. Provide an entry-level building chain tool.
2. Provide modular unit testing and end-to-end integration testing and support automated continuous integration and continuous deployment.

3.3.2 Compatibility note

Compatible version

CHAPTER 4

Installation

This chapter will introduce the required installations and configurations of FISCO BCOS. For better understanding, we will illustrate an example of deploying a 4-node consortium chain in a local machine using FISCO BCOS.

4.1 To build a single-group consortium chain

This section takes the construction of single group FISCO BCOS chain as an example to operate. We use the `build_chain.sh` script to build a 4-node FISCO BCOS chain locally in `Ubuntu 16.04` system.

Note:

- To update an existing chain, please refer to compatibility chapter.
- It is similar to build a multi-group chain, interested can be referred to here .
- This section uses pre-compiled static *fisco-bcos* binaries which tested on CentOS 7 and Ubuntu 16.04.

4.1.1 Prepare environment

- Install dependence

`build_chain.sh` script depends on `openssl, curl` and is installed by using the following instructions. For CentOS system, to replaces `apt` with `yum` in the following command. For macOS system, to executes `brew install openssl curl`.

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

- Create operation directory

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

- Download `build_chain.sh` script

```
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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
```

4.1.2 Build a single-group 4-node consortium chain

Execute the following command in the fisco directory to generate a single group 4-node FISCO chain. It is necessary to ensure that the `30300~30303, 20200~20203, 8545~8548` ports of the machine are not occupied.

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

Note:

- The -p option specifies the starting ports, which are p2p_port, channel_port, jsonrpc_port. For security reasons, jsonrpc/channel listens to 127.0.0.1 by defaults. **If you require to access external network, please add -i parameter**.

If the command is executed successfully, `All completed` will be output. If the execution fails, please check the error message in the `nodes/build.log` file.

```
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 following command to get FISCO-BCOS console
 bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪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] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
[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
```

4.1.3 Start FISCO BCOS chain

- Execute the following command to start all nodes

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

Success will output a response similar to the following, otherwise, please use `netstat -an | grep tcp` to check whether the machine's `30300~30303, 20200~20203, 8545~8548` ports are occupied.

```
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
```

4.1.4 Check process

- Execute the following command to check whether the process is started

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

In normal situation, the output will be similar to the following. If the number of processes is not 4, then the reason why the process does not start is that the port is occupied.

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

4.1.5 Check log output

- Execute the following command to view the number of nodes that node0 links to

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

In normal situation, the connecting messages will be output continuously. From the output messages, we can see that node0 has links with the other three nodes.

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

- Execute the following command to check whether it is in consensus

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

In normal situation, the message will be output `++++Generating seal` continuously to indicate that the consensus is normal.

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

4.2 Using console

Console links nodes of FISCO BCOS through Web3SDK so as to realize functions like blockchain status query, call and deploy contracts. The instructions of console are introduced here.

4.2.1 Prepare environment

```
# back to fisco directory
$ cd ~/fisco
# Install openjdk. In macOS, execute 'brew cask install java' to install java
$ sudo apt install -y default-jdk
# get console
$ bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪download_console.sh)
# Copy the console configuration file. If the node does not use the default port,␣
↪please replace 20200 in the file with another port.
$ cp -n console/conf/applicationContext-sample.xml console/conf/applicationContext.
↪xml
# Configure the console certificate
$ cp nodes/127.0.0.1/sdk/* console/conf/
```

Important:

- if the console has been configured correctly, but it reports the following exception when starting console in CentOS system:

  Failed to connect to the node. Please check the node status and the console configuration.

  this is caused by the in-built JDK version of CentOS system(who will lead to verification failure of console and nodes). Please download and install Java 8 or above version from OpenJDK official website or Oracle official website (for detailed installation steps please check Additional), and start console after finishing installation.

4.2.2 Start console

```
# back to console folder
$ cd ~/fisco/console
# start console
$ ./start.sh
# if it outputs following information, then the console has been started␣
↪successfully, otherwise please check if the node ports in conf/
↪applicationContext.xml are configured correctly.
=====================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________  ______   ______    ______    ______         _______    ______    ______␣
↪   ______
|        \|      \ /      \  /      \  /      \       |       \  /      \  /      ␣
↪\  /      \
| $$$$$$$$ \$$$$$$|  $$$$$$\|  $$$$$$\|  $$$$$$\      | $$$$$$$\|  $$$$$$\|  $$$$$
↪$$\|  $$$$$$\
| $$__       | $$  | $$___\$$| $$   \$$| $$  | $$      | $$__/ $$| $$   \$$| $$  | $
↪$| $$___\$$
| $$  \      | $$   \$$     \ | $$      | $$  | $$      | $$    $$| $$      | $$  | $
↪$ \$$     \
| $$$$$      | $$   _\$$$$$$$\| $$   __ | $$  | $$      | $$$$$$$\| $$   __ | $$  | $
↪$ _\$$$$$$$\
| $$        _| $$_ |  \__| $$| $$__/  \| $$__/ $$      | $$__/ $$| $$__/  \| $$__/ $
↪$|  \__| $$
| $$       |   $$ \ \$$    $$ \$$    $$ \$$    $$      | $$    $$ \$$    $$ \$$    $
↪$ \$$    $$
 \$$        \$$$$$$  \$$$$$$   \$$$$$$   \$$$$$$        \$$$$$$$   \$$$$$$   \$$$$
↪$   \$$$$$$

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

4.2.3 Acquire information through console

```
# acquire client ends version information
[group:1]> getNodeVersion
{
    "Build Time":"20190121 06:21:05",
    "Build Type":"Linux/clang/Debug",
    "FISCO-BCOS Version":"2.0.0",
    "Git Branch":"master",
    "Git Commit Hash":"c213e033328631b1b8c2ee936059d7126fd98d1a"
}
# acquire node connection information
[group:1]> getPeers
[
    {
        "IPAndPort":"127.0.0.1:49948",
        "NodeID":
↪"b5872eff0569903d71330ab7bc85c5a8be03e80b70746ec33cafe27cc4f6f8a71f8c84fd8af9d7912cb5ba068901fe
↪",
        "Topic":[]
    },
    {
        "IPAndPort":"127.0.0.1:49940",
        "NodeID":
↪"912126291183b673c537153cf19bf5512d5355d8edea7864496c257630d01103d89ae26d17740daebdd20cbc645c9a
↪",
        "Topic":[]
    },
    {
        "IPAndPort":"127.0.0.1:49932",
        "NodeID":
↪"db75ab16ed7afa966447c403ca2587853237b0d9f942ba6fa551dc67ed6822d88da01a1e4da9b51aedafb8c64e9d20
↪",
        "Topic":[]
    }
]
```

4.3 To deploy or call HelloWorld contract

4.3.1 HelloWorld contract

HelloWorld contract offers 2 interfaces which are `get()` and `set()` and are used to acquire/set contract variety `name`. The contract content is as below:

```
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;
```

```
    }
}
```

4.3.2 Deploy HelloWorld contract

For quick experience, the console comes with HelloWorld contract and is placed under console folder `solidity/contracts/HelloWorld.sol`. So, users only have to deploy it using the following command.

```
# input the following instruction in console, if it is deployed successfully, the␣
↪contract address will be returned
[group:1]> deploy HelloWorld
contract address:0xb3c223fc0bf6646959f254ac4e4a7e355b50a344
```

4.3.3 Call HelloWorld contract

```
# check the current block number
[group:1]> getBlockNumber
1
# call get interface to acquire name variety, the contract address here is the␣
↪returned address of deploy instruction
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 get
Hello, World!
# check the current block number, it remains the same, because get interface will␣
↪not change the ledger status
[group:1]> getBlockNumber
1
# call set to set name
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 set "Hello,␣
↪FISCO BCOS"
0x21dca087cb3e44f44f9b882071ec6ecfcb500361cad36a52d39900ea359d0895
# check the current block number again, if it increased, then it has generated␣
↪block and the ledger status is changed
[group:1]> getBlockNumber
2
# call get interface to acquire name variety, check if the setting is valid
[group:1]> call HelloWorld 0xb3c223fc0bf6646959f254ac4e4a7e355b50a344 get
Hello, FISCO BCOS
# log out console
[group:1]> quit
```

Note: To deploy contract can also specify the contract version number of the deployment through the `deployByCNS` command, using method reference here. To call contract via the `callByCNS` command, using the method reference here.

CHAPTER 5

Tutorials

This chapter will introduce the basic process and related core concept for quick development of DApp on FISCO BCOS. We will also provide company users a toolkit tutorial for easier development and deployment.

5.1 Core concept

Blockchain is intricated with multiple technical approaches. This chapter will illustrate the basic concept of blockchain and knowledges about some relative theories. You can skip this chapter if you are familiar with these techniques.

5.1.1 What is blockchain

Blockchain is a concept proposed after bitcoin. In Satoshi Nakamoto's Paper about bitcoin, he didn't mention about blockchain but described the data structure as "chain of block".

Chain of block is an organization type of data, a chain-like structure connected by hashes of blocks. Blockchain refers to a comprehensive technology intricated with many techniques to maintain and manage the chain of block and form an immutable distributed ledger.

Blockchain technology is adapted to build an unforgeable, immutable and traceable block-chain data structure in an equal networking environment through transparent and trustable rules, a way to realize and manage trusted data generation, in & out and usage. As of the technical structure, blockchain is an all-rounded solution formed by multiple information technologies including distributed structure and storage, block-chain data structure, p2p networking, consensus algorithm, cryptography algorithm, game theory and smart contract.

Blockchain technology and eco-system is originated from bitcoin. Today, serious look has been taken on this technology by broad industries like finance, justice, supply chain, entertainment, social administration, IoT, etc. They want to put its great technical value into extensive distributed cooperation. Meanwhile, progress has also been seen in the blockchain technology and product model. FISCO BCOS blockchain platform concentrates on capabilities of improving security, performance, usability, friendly operation, privacy protection, compliance and regulation based on blockchain technology. It grows together with the community eco-system to better present functions like multi-participation, smart cooperation, professional division of labor, and value sharing.

Ledger

Ledger, as its name suggests, is used for managing data of accounts or transaction records and supports functions like distributed ledger, reconciliation and settlement, etc.. In multi-lateral cooperation, partners want to co-maintain or share a real-time, correct and safe distributed ledger to eliminate inequality of message and improve efficiency and ensure funds and business security. And blockchain are often regarded as a core technology for building up distributed sharing ledger, contributed by joint efforts of technologies like block-chain data structure, multi-party consensus mechanism, smart contract, global state storage, which can realize a consistent, trustable, safe and immutable traceable sharing ledger. Ledger contains contents like block number, transactions, accounts and global state.

Block

Block is a data structure built in chronological order. The first block of blockchain is called Genesis Block, and the latter blocks are identified with block number. Each block number increases one by one. New block will take the hash of the former block and generate a unique data print by hash algorithm and local block data, so that a coherent block-chain structure, namely blockchain, is formulated. This delicate data structure design makes it possible that data on chain saved in order and is traceable and verifiable. If any of the data is changed, it will cost extremely high because of being stuck in verification of all chain.

The basic data structure of a block is block header and block body. Block header contains block height, hash, generator's signature, state root. Block body contains the returning message of a transaction data list. The size of block will be different according to the size of the transaction list, and it won't be too large considering network transmission, up to between 1M to 10M byte.

Transaction

Transaction can be seen as a request data targeting blockchain system to deploy contract, call contract interface, maintain life cycle of contract and manage assets, conduct value exchange, etc.. The basic data structure of transaction includes sender, receiver and transaction data. User can create a transaction and sign it with its private key, then send on chain (through interface like sendRawTransaction). And it will be consensus by several nodes, executes smart contract code, generates the status data assigned with the transaction and packs it into block and saves with status data. Now the transaction is confirmed and gets its duty and consistency.

As the transaction is confirmed, a receipt will be created and saved in the block correspondently for storage of execution information like result code, log and gas consumption. User can use hash of the transaction to check its receipt to know whether it is finished. Equivalent with "write" transaction, there is a "read-only" method of invocation for reading data on chain.

It shares the similar request method with transaction but invokes functions by call(). When the node receives "read-only" invocation request, it will return with the requested accessed parameter status without inviting the request into consensus process to avoid modification of data on chain.

Account

In the blockchain system designed in account model, account represents the uniqueness of user and smart contract.

In the blockchain system adapting private-public key, user creates a public and private key pair and calculates a unique address string by hash or other algorithms to be the account of this user. User uses private key to manage assets in this account. Sometimes there might not be storage space for user account, so smart contract will manage user's data instead, this account is called "exterior account".

As of smart contract, when one smart contract is deployed, it has an only address on chain, which is also called contract account that pointing at the index for status bit, binary code, related status data, etc.. During operation of smart contract, it will load binary code through this address and visit data in global state storage by the index of status data. Then, according to the operation result, it will write the data into global state storage and update status data index in the contract account. To deactivate smart contract, user only has to change its status bit into invalid, and the real data of the contract account won't be cleared out usually.

Global state

FISCO BCOS uses account model design, which contains another storage space for smart contract operation result besides space for blocks and transactions. The status data generated during smart contract execution is confirmed by consensus mechanism and saved distributedly on each node to ensure global consistency, verifiability and immutability. And therefore, it's called "global" status.

Because of status storage space, blockchain is able to save various kinds of data, including user account information like balance, etc., smart contract binary code and operation result or other related data. During execution, smart contract will acquire some data from status storage for calculation, laying the foundation for complex contract logic realization.

On the other hand, status data maintenance costs high in storage, as the chain keeps operating, status data will keep inflating, like adapting the complex data structure Patricia Tree results in expansion of capacity. Therefore, status data can be cut or optimized in some cases, or use storage solution like distributed data warehouse for more extensive data capacity.

Consensus mechanism

Consensus mechanism is the core concept of blockchain, as there will not be blockchain if without consensus. Blockchain is a distributed system where nodes cooperate to calculate and witness the execution of transactions and confirm the final result. It brings these loosely-coupled untrusted participants together to be trusted partners, and keeps the cooperation being consistent and lasting, which can be abstracted as the process of "consensus", and the related algorithms and strategies are called consensus mechanism.

Node

A computer installed with software and hardware concerning blockchain system and join the blockchain network can be called a "node". Nodes take part in network communication, logic calculation, data verification and storage of block, transaction and status, etc., and provide client ends interfaces for transaction process and data inquiry. The identification of node adapts public-private key mechanism, which generates a string of unique Node ID to ensure its uniqueness on blockchain.

According to the involvement of calculation and storage of data, nodes can be categorized into consensus node, observation node and lightweight node. Consensus node fully participates in consensus process, packing block as accountant and verifying block as verifier. Observation node doesn't join consensus process but synchronize data for verification and storage as a data service provider. Lightweight node only synchronizes block headers and few transactions and status data, providing verification for certain online transaction or data inquiry.

Consensus algorithm

Consensus algorithm needs to handle several core problems:

1. pick out a role as with ledger right within the system, and start ledgering as a leader.
2. Participator adapts undeniable and immutable algorithm, verifies in multi-levels and takes the ledger from leader.
3. Data synchronization and distributed cooperation can make sure that all participants receive the same and correct result.

Common algorithm of blockchain includes Proof of Work, Proof of Stake and Delegated Proof of Stake that are often used in public chain, and Practical Byzantine Fault Tolerance (PBFT), RAFT that are often used in consortium chain. Besides, some advanced consensus algorithms often organically combine the above mentioned algorithms with random number generator to improve security, energy consumption and performance, size or other issues.

FISCO BCOS consensus module is designed pluggable and supports many consensus algorithms, including PBFT and RAFT currently, to realize broader and faster consensus algorithms.

Smart contract

Smart contract, first proposed by Nick Szabo in 1995, is a contract defined in numeric term and can execute clauses automatically. Numeric term means that contract has to be realized by computer codes, for as long as the agreement is reached among participants, the right and responsibility established by smart contract will be executed automatically and its result is undeniable.

FISCO BCOS applies smart contract in not only asset management, rules definition and value exchange, but also overall configuration, maintenance, governance and authority setting, etc.

Life cycle of smart contract

The life cycle of smart contract contains steps of design, development, test, deployment, operation, upgrade and deactivation.

Developers edit, compile and unit-test the codes of smart contract, the development language can be solidity, C++, java, go, javascript, rust, etc.. The choice of language depends on the type of virtual machine. After passing test, the contract will be published on chain using deployment command and confirmed by consensus algorithm.

The contract will be called by transaction afterwards after validated. When contract needs upgrade, user has to repeat the above mentioned steps from development to deployment so as to release new contract version, which will own a new address and independent storage space instead of covering the old one. The new contract can access the data in the old contract through the interface, or migrate the old contract data into its own storage. The best practice is of the "behavior contract" for designing execution process and "data contract" for storing data. It decouples data and contract, so when there is change in process but not data, the new behavior contract can access the existed data contract.

Revoking an old contract doesn't mean clearing all of its data, but only setting its status to "invalid" so that this contract can't be called any more.

Smart contract virtual machine

To run digital smart contract, blockchain system needs compiler and actuator that are capable of compilation, analysis and code execution, which is called virtual machine system. After the contract is edited and complied by the compiler, user sends deployment transaction to deploy it on blockchain system. Once the transaction has passed consensus process, the system will allocate a binary code with a unique address to reserve contract. When a contract is called by another transaction, virtual machine actuator loads code from contract storage and executes to output execution result.

In a blockchain system which emphasizes security, transactional routines and consistency, the virtual machine should possess sandbox features to block uncertain factors, like random number, system time, exterior file system and network, as well as invasion of malicious code to make sure consistent execution result and safe process of one transaction and one contract on different nodes.

Currently popular virtual machines includes EVM, controlled Docker, WebAssembly, etc.. The virtual machine model of FISCO BCOS adapts modularization design and supports broadly-used EVM. More kinds of adaptable virtual machines can be expected in the future.

Turing complete

Turing machine or Turing complete is a classical concept in computing field. It is an abstract computing model proposed by mathematician Alan Mathison Turing (1912-1954) and is stretched into blockchain field, referring to a model where contract supports logical computing like judgement, jump, cycle and recursion, process ability of multiple data types like integer, byte string and structure, and even has object-oriented features like inheritance, derivation and interface. This will make it possible for complex transactional logics and complete contract execution, which is distinct from simple script that only supports operand stack.

Most blockchain systems appearing after 2014 support Turing complete smart contract to make them highly compilable. On top of some basic features of blockchain (like multi-party consensus, immutability, traceability and

security), they can realize contract with transactional logics, such as The Ricardian Contract, and smart contract is also adaptable.

The execution of contract needs to process the Halting problem, namely to judge whether the program can solve the input problem within limited time and terminate execution to release resource. Now imagine that a contract is deployed in the whole network, when being called it will also be executed on every node. And if the contract is an infinite cycle, then there is possibility to use up all the system resources. So, the addressing of Halting problem is an important issue of Turing complete computing system in blockchain.

5.1.2 Consortium blockchain

Usually blockchain is divided into 3 types: public blockchain, consortium blockchain and private blockchain. Public blockchain can be joined by anybody at any time, even anonymously. Private blockchain is owned by an entity (an agency or a nature person) and is managed and used in private way. Consortium blockchain is usually formed by multiple entities who made an agreement on certain protocols, or built a business alliance. Whoever wants to join consortium blockchain needs verification, often with knowable identity. For there is access control, consortium blockchain is also called "permissioned blockchain".

Consortium blockchain has access control and identity management in most sections from creation, member joining to operation and transaction. Operations on chain can be monitored by permission setting. Consensus mechanism like PBFT based on multi-party multi-round verification voting is adopted here, instead of energy-consuming POW mining mechanism. Therefore, its network scale is relatively controllable and can be massively optimized in aspects like transaction delay, transactional consistency and certainty, concurrency and capacity.

Consortium blockchain has inherited the advantages of blockchain and is more adaptable for some difficult business cases which requires high performance capacity, regulation and compliance, like finance, justice or others that related to entity economy. The way consortium blockchain possesses is suitable both for business compliance and stability and for innovation, which is also been encouraged by the country and the industry.

Performance

Performance indicator

The most popular performance indicator for software system is TPS (Transaction Per Second), the transaction volume that the system could process and confirm per second. The higher the TPS, the better the performance. Besides TPS, the performance indicators for blockchain also include Delay ACK and the network scale, etc..

Delay ACK is the total time used from transaction arriving on blockchain to final confirmation after a series of process including verification, calculation and consensus. For example, each block on bitcoin network costs 10 minutes. And transaction will be processed mostly by 6 blocks, that is 1 hour. In PBFT algorithm, transaction can be confirmed in seconds with final certainty, which caters to the need of finance transactions.

The network scale refers to the number of co-working consensus nodes the system supports on the premise of assured TPS and Delay ACK. It's believed usually by insiders that the node scale is around hundred level if using PBFT consensus algorithm in a system, and increment of the node scale will result in decrease of TPS and increase of Delay ACK. The consensus mechanism where the accounting group is chosen by random number algorithm can fix the problem.

Performance optimization

The is two options of performance optimization: scale up and scale out. scale up is to optimize the configurations of software and hardware on the basis of limited resources to lift up processing ability, like using more efficient algorithm or hardware acceleration, etc.. Scale out means good extendibility of system structure. It uses Sharding and Partition to carry out various users and transaction flows. As long as adding software and hardware resources appropriately, it can load more requests.

Performance indicators, software structure, hardware configuration like CPU, internal memories, storage scale and internet bandwidth, are all closely related. And following the increment of TPS, there will be more pressure for storage capacity, which needs to be considered comprehensively.

Security

Security is a big topic, especially for blockchain system that built on distributed network with multi-party engagement. In system level, problems like internet attack, system penetration, data corruption or leakage should be concerned. In transaction level, we should consider about unauthorized operation, logic errors, asset impairment caused by system stability and privacy invasion.

To ensure security we need to focus on "the shortest board of buckets" and prepare with comprehensive security strategy providing all-rounded protection that meets high security standard. With best practices in security and equal security for all participants, this will make sure the security within overall network.

Access mechanism

Access mechanism refers to the processes for either agencies or persons who want to build or join the blockchain, like multi-party verification of the subject to make sure it has knowable identity, trustable quality and reliable technology before starting the creation work of consortium blockchain, so the verified node will be added into blockchain and allocated with public and private keys that can send transactions. After the access process is done, information of the agency, node or staff will be registered on blockchain or reliable information services, where every operation can be traced down to each agency and person.

Permission control

Permission control on consortium blockchain means controls on data read-and-write in various sensitivity levels by different staff. It contains permissions in contract deployment, data access in contract, block data syncing, system parameters access and change, node start-stop, etc.. There can be more permission controls according to different transactional needs.

Permissions are allocated to roles if using Role-Based Access Control model design. One example for reference is to divide roles into operation manager, transaction operator, application developer, O&M manager, administrator. And each role can be further subdivided to meet other needs. The complete model could be very huge and complicated, so it should be designed properly to adapt to transactional needs as well as security concerns.

Privacy protection

Business cooperation based on blockchain structure requires all parties to output and share data for calculation and verification. In complicated business context, agencies want better control on their data. And there is also a growing need for personal data privacy protection. Therefore, how to protect the private part of shared data and prevent from privacy leakage during operation becomes an important problem.

Administration is the first area to address privacy protection. When the system starts running, it should keep the principle of "minimum authorized and express consent", complete life-cycle management of data collection, storage, application, disclosure, deletion and recovery, and establish daily management and emergency management system. For those business transactions with high sensitivity, there should be a regulation role for checking and auditing from a third party so that all sections can be supervised.

Technically, data masking, transaction separation or system physical isolation or other ways can control the scope of data distribution. Meanwhile, cryptographic methods like Zero-knowledge Proof, Secure Multi-Party Computation, Ring Signature, Group Signature and Blind Signature can protect data through strong encryption.

Physical isolation

Physical isolation is a radical method to avoid privacy data leakage. In this way, only the participants who share data can communicate in the network layer, others will not be able to communicate or exchange even one byte of data.

There is another method called logic isolation in which participants can receive other data but with access limits or encryption, so unauthorized participants or those with no keys have no right of access and change. However, with the technological development, data that are limited in access or encrypted may be decoded in the future.

For data with extremely high sensitivity, it's good to use physical isolation strategy to eradicate possibility of being cracked. But meanwhile, it will cost more in detailed screening of data's sensitivity level and need thorough planning and enough hardware resources to load different data.

Governance and regulation

Governance of consortium blockchain

Governance of consortium involves coordination, incentive mechanism, safe operation and regulation audit of multiple participants. The core is to sort out the responsibility and right, work flow of each party to build up a smooth development and O&M system, guarantee compliance and create precaution and emergency management for security issues. Rules need to be stipulated to make sure every participant reaches agreement and conducts thorough execution to accomplish governance.

A typical reference model for consortium blockchain is that all participants co-found a consortium blockchain committee for joint discussion and decision making, design roles and tasks according to transaction needs, for instance, some agencies work on development and some join operation management, and every agency takes part in transactions and O&M with smart contract managing rules and maintaining system data. The committee and regulator can be given with right of permission control, verifying and setting permissions for transactions, agencies and staff. When it comes to emergency, they can carry out emergency operations like resetting accounts and adjusting transactions through agreed rules in smart contract. When the system needs to be upgraded, the committee will take the responsibility of coordinating each party.

On a consortium blockchain with complete governance mechanism, there will be peer-to-peer cooperation of all participants, including asset transactions and data exchanging, which improves operation efficiency greatly and business innovation as well as guarantees the compliance and security.

Fast deployment

The general steps to build a blockchain system include: acquire hardware resources including server, internet, memories, hard disk, etc.; configure the environment by choosing an operation system, opening a network port and making strategies, bandwidth planning and storage space allocation, etc.; acquire binary executable software or compile it from the source code; configure the blockchain system, including genesis block configuration, parameter configuration and log configuration; configure multi-party interconnection, including node access configuration, port discovery, consensus participants list, etc.; configure client ends and developer tools, including the console and SDK, etc.. There are so many complicated and repetitive steps, like the management of certificates and public and private keys, which form a high entry barrier.

Therefore, to simplify and quicken the process of building blockchain with low error rate and cost, these need to be considered: First, standardize the target deployment platform and prepare in advance the operation system, reliable software list, network bandwidth, network strategy and other key software and hardware, match the version with parameters to make the platform available and ready for use. Currently there are cloud services or docker that can help building standardized platform.

Then, take the user experience into consideration by optimizing the formation, configuration and networking of blockchain software and offering toolkit for fast and automatic networking, so users will not be tangled with miscellaneous details but can start operating blockchain with few steps.

FISCO BCOS emphasizes the deployment experience for users and offers command-line for one-click deployment to help developers expedite development and debugging environment building. It provides networking tool of business level for flexible parameters configuration of the host and network, manages relative certificates for easier cooperation among companies when they co-network on blockchain. By optimized deployment method, it shortens the time of building blockchain into a few minutes or within half an hour.

Data governance

Blockchain requires data to be verified in each layer and leaves traceable records. Usually, the solution is to save all the data on all nodes (except lightweight nodes), resulting in data inflation and capacity intensity. It is especially obvious with cases that bear massive services. After some time, regular storage solution has limited data capacity, and it costs high to adopt mass storage. Besides, security should also be concerned. The permanent storage of all data may face risk of data leakage. Therefore, it is important to better the design of data governance.

Some strategies of data governance are concluded here: cutting and transfer, parallel expansion and distributed storage. It depends on specific cases to determine which one is suitable.

For data with strong temporal features, like in a case that account clearing happens one time a week, then the data before the week will not be calculated or verified again. The old data can be transferred from node to big data storage to meet the need of data traceability and verifiability and long storage life for transactions. When there is lower data pressure for nodes and history data is kept off-line, more attention can be put on security strategies.

When it comes to snowballing transaction cases, like when users and contract copies increase tremendously, it can allocate each of them to different logic partitions, each of which owns independent storage space and bears certain quantity of data. When data is near capacity limit, it will arrange more resources to store new data. Partition design makes it easier to do resources allocation and cost management.

Combining the strategies of data cutting and transfer with parallel expansion, the cost of data capacity and security of data can get better controlled, and it also benefits the execution of massive transactions.

O&M monitoring

Blockchain system presents high consistency in its foundation and operation logic. Different nodes often share the same software and hardware system. Its standardized features bring convenience for operation and maintenance staff. They can use the commonly-used tools, O&M strategy and workflow or others to build, deploy, configure blockchain system and handle faults to realize low O&M cost and high efficiency.

O&M staff has limited authority to operate in consortium blockchain. They have permission to modify system configuration, process start-stop, check operation log and detect troubles, but they are not involved in transactions and cannot check user data or transaction data that rates high in privacy security.

During the operation of system, they can monitor all the operational indicators and evaluate the health of system status through monitoring system. It will send warning messages when there appear faults, so the staff can response and handle them immediately.

The monitoring system covers status of fundamental environment, like CPU occupation rate, system memories rate and incremental, IO status of disk, internet connection quantity and traffic, etc..

The monitoring of blockchain system includes block number, transaction volume and virtual machine computation, and voting and block generation of consensus nodes, etc..

The monitoring of interface includes counting, time consumed, and success rate of API callings.

The monitoring data can be output from log or network interface for agencies to connect with the existing monitoring systems so the monitoring ability and O&M workflows can be multiplexed. When the O&M staff receive the warning message, they can use the O&M tool offered by consortium blockchain to view system information, modify configuration, start-stop process and handle faults, etc..

Regulation audit

With the development of blockchain technology and business exploration, the blockchain platform needs a function to support regulation to prevent it from being against regulation rules and laws, or becoming the carrier for money washing, illegal financing and criminal transactions.

The audit function is mainly designed to meet the needs of audit and internal control, responsibility confirmation and event tracing of blockchain system. It should be combined with effective techniques to do accurate audit management according to the specific industrial standards.

Regulators can join the blockchain system as nodes, or interact with blockchain system through interfaces. They can synchronize all the data for audit analysis and trace overall transaction flows. And if they detect exceptions, they can send instruction with regulation authority. Also, they can monitor transactions, participants and accounts to realize "penetrative regulation".

FISCO BCOS supports regulation audit in aspects like roles and access control design, function interface and audit tool.

5.2 Build the first blockchain application

In this chapter we will introduce a whole process of business application scenario development based on FISCO BCOS blockchain. The introduce includes business scenario analysis, contract design implementation, contract compilation, and blockchain development. Finally, we introduce an application module implementation which is to implement calling access to the contract on blockchain through the Web3SDK we provide.

This tutorial requires user to be familiar with the Linux operating environment, has the basic skills of Java development, is able to use the Gradle tool, and is familiar with Solidity syntax. Through the tutorial, you will learn the following:

1. How to express the logic of a business scenario in the form of a contract
2. How to convert Solidity contract into Java class
3. How to configure Web3SDK
4. How to build an application and integrate Web3SDK into application engineering
5. How to call the contract interface through Web3SDK, and to understand its principle

The full project source code for the sample is provided in the tutorial. Users can quickly develop their own applications based on it.

Important: Please refer to Installation documentation to complete the construction of the FISCO BCOS blockchain and the console download. The operation in this tutorial is assumed to be carried out in the environment of the documentation building.

5.2.1 Sample application requirements

Blockchain is naturally tamper-proof and traceable. Its characteristics make it more attractive to the financial industry. In this article, we will provide an easy example of asset management development and ultimately achieve the following functions:

- Ability to register assets on blockchain
- Ability to transfer funds between different accounts
- Ability to check the amount of assets in account

5.2.2 Contract design and implementation

When developing an application on blockchain, for combining with business requirements, it is first necessary to design the corresponding smart contract to determine the storage data that contract needs, and on this basis, to determine the interface provided by the smart contract. Finally, to specifically implement each interface.

Storage design

FISCO BCOS provides a contract CRUD interface development model, which can create table through contracts, and add, delete, and modify the created table. For this application, we need to design a table `t_asset` for storage asset management. The table's fields are as follows:

- account: primary key, asset account (string type)
- asset_value: asset amount (uint256 type)

account is the primary key, which is the field that needs to be passed when the `t_asset` table is operated. Blockchain queries the matching records in the table according to the primary key field. The example of `t_asset` table is as follow:

Interface design

According to the design goals of the business, it is necessary to implement asset registration, transfer, and query functions. The interfaces of the corresponding functions are as follows:

```
// asset amount query
function select(string account) public constant returns(int256, uint256)
// asset registration
function register(string account, uint256 amount) public returns(int256)
// asset transfer
function transfer(string from_asset_account, string to_asset_account, uint256␣
↪amount) public returns(int256)
```

Full source code

```
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 {
        // create a t_asset table in the constructor
        createTable();
    }

    function createTable() private {
        TableFactory tf = TableFactory(0x1001);
        // asset management table, key : account, field : asset_value
        // |  asset account(primary key)      |     asset amount       |
        // |-------------------- |-------------------|
        // |        account      |    asset_value    |
        // |---------------------|-------------------|
        //
        // create table
```

```
        tf.createTable("t_asset", "account", "asset_value");
    }

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

    /*
    description: query asset amount according to asset account

    parameter:
            account: asset account

    return value:
            parameter1:  true to return to 0 or no account to return to -1
            parameter2:  it is valid when the first parameter is 0, the amount of
↪assets
    */
    function select(string account) public constant returns(int256, uint256) {
        // open table
        Table table = openTable();
        // query
        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")));
        }
    }

    /*
    description : asset registration
    parameter :
            account : asset account
            amount  : asset amount
    return value:
            0  regist successfully
            -1 asset account already exists
            -2 other error
    */
    function register(string account, uint256 asset_value) public returns(int256){
        int256 ret_code = 0;
        int256 ret= 0;
        uint256 temp_asset_value = 0;
        // to query whather the account exists
        (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));
            // insert
            int count = table.insert(account, entry);
            if (count == 1) {
                // true
                ret_code = 0;
            } else {
                // false. no permission or other error
```

```
                ret_code = -2;
            }
        } else {
            // account already exists
            ret_code = -1;
        }

        emit RegisterEvent(ret_code, account, asset_value);

        return ret_code;
    }

    /*
    description : asset transfer
    parameter :
            from_account : transferred asset account
            to_account : received asset account
            amount :  transferred amount
    return value:
            0  transfer asset successfully
            -1 transferred asset account does not exist
            -2 received asset account does not exist
            -3 amount is insufficient
            -4 amount is excessive
            -5 other error
    */
    function transfer(string from_account, string to_account, uint256 amount)␣
↪public returns(int256) {
        // query transferred asset account information
        int ret_code = 0;
        int256 ret = 0;
        uint256 from_asset_value = 0;
        uint256 to_asset_value = 0;

        // whather transferred asset account exists?
        (ret, from_asset_value) = select(from_account);
        if(ret != 0) {
            ret_code = -1;
            // not exist
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;

        }

        // whather received asset account exists?
        (ret, to_asset_value) = select(to_account);
        if(ret != 0) {
            ret_code = -2;
            // not exist
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        if(from_asset_value < amount) {
            ret_code = -3;
            // amount of transferred asset account is insufficient
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        if (to_asset_value + amount < to_asset_value) {
            ret_code = -4;
            // amount of received asset account is excessive
```

```
            emit TransferEvent(ret_code, from_account, to_account, amount);
            return ret_code;
        }

        Table table = openTable();

        Entry entry0 = table.newEntry();
        entry0.set("account", from_account);
        entry0.set("asset_value", int256(from_asset_value - amount));
        // update transferred account
        int count = table.update(from_account, entry0, table.newCondition());
        if(count != 1) {
            ret_code = -5;
            // false? no permission or other error?
            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));
        // update received account
        table.update(to_account, entry1, table.newCondition());

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

        return ret_code;
    }
}
```

Note: The implementation of the `Asset.sol` contract requires to introduce a system contract interface file `Table.sol` provided by FISCO BCOS. The system contract file's interface is implemented by the underlying FISCO BCOS. When a business contract needs to operate CRUD interface, it is necessary to introduce the interface contract file. `Table.sol` contract detailed interface reference here.

5.2.3 Contract compiling

In the previous section, we designed the storage and interface of the contract `Asset.sol` according to business requirements, and implemented them completely. However, Java program cannot directly call Solidity contract. Solidity contract file needs to be compiled into a Java file first.

Console provides a compilation tool that can store the `Asset.sol` contract file in the `console/contract/solidity` directory, and uses the `sol2java.sh` script provided in the console directory to compile. The operation is as follows:

```
# switch to fisco/console/ directory
$ cd ~/fisco/console/
# compile contract, to specify a Java's package name parameter behind, you can␣
↪specify the package name according to the actual project path.
$ ./sol2java.sh org.fisco.bcos.asset.contract
```

After running is successful, the java, abi, and bin directories will be generated in the `console/contracts/sdk` directory as shown below.

```
|-- abi # The generated abi directory, which stores the abi file generated by␣
↪Solidity contract compilation.
|   |-- Asset.abi
|   |-- Table.abi
|-- bin # The generated bin directory, which stores the bin file generated by␣
↪Solidity contract compilation.
|   |-- Asset.bin
```

```
|   |-- Table.bin
|-- contracts # The source code file that stores Solidity contract. Copy the␣
↪contract that needs to be compiled to this directory.
|   |-- Asset.sol # A copied Asset.sol contract, depends on Table.sol
|   |-- Table.sol # System CRUD contract interface file provided by default
|-- java  # Storing compiled package path and Java contract file
|   |-- org
|        |--fisco
|             |--bcos
|                  |--asset
|                       |--contract
|                            |--Asset.java  # Java file generated by the Asset.
↪sol contract
|                            |--Table.java  # Java file generated by the Table.
↪sol contract
|-- sol2java.sh
```

The `org/fisco/bcos/asset/contract/` package path directory is generated in the java directory. The directory contains two files `Asset.java` and `Table.java`, where `Asset.java` is the necessary file that Java application to call `Asset.sol` contract.

`Asset.java`'s main interface:

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

public class Asset extends Contract {
    // Asset.sol contract  transfer interface generation
    public RemoteCall<TransactionReceipt> transfer(String from_account, String to_
↪account, BigInteger amount);
    // Asset.sol contract  register interface generation
    public RemoteCall<TransactionReceipt> register(String account, BigInteger␣
↪asset_value);
    // Asset.sol contract  select interface generation
    public RemoteCall<Tuple2<BigInteger, BigInteger>> select(String account);

    // Load the Asset contract address, to generate Asset object
    public static Asset load(String contractAddress, Web3j web3j, Credentials␣
↪credentials, ContractGasProvider contractGasProvider);

    // Deploy Assert.sol contract, to generate Asset object
    public static RemoteCall<Asset> deploy(Web3j web3j, Credentials credentials,␣
↪ContractGasProvider contractGasProvider);
}
```

The load and deploy functions are used to construct the Asset object, and the other interfaces are used to call the interface of the corresponding solidity contract. The detailed use will be introduced below.

5.2.4 SDK configuration

We provide a Java engineering project for development. First, to get the Java engineering project:

```
    # get the compressed package of Java engineering
    $ cd ~
    $ curl -LO https://github.com/FISCO-BCOS/LargeFiles/raw/master/tools/asset-app.
↪tar.gz
    # extract to get the Java engineering project asset-app directory
    $ tar -zxf asset-app.tar.gz
```

The directory structure of the asset-app project is as follows:

```
|-- build.gradle // gradle configuration file
|-- gradle
|   |-- wrapper
|       |-- gradle-wrapper.jar //  the related code implementation for downloading␣
↪Gradle
|       |-- gradle-wrapper.properties // configuration information used by the␣
↪wrapper, such as the gradle's version etc..
|-- gradlew // shell script for executing wrapper commands under Linux or Unix
|-- gradlew.bat // batch script for executing wrapper commands under Windows
|-- src
|   |-- main
|   |   |-- java
|   |         |-- org
|   |             |-- fisco
|   |                   |-- bcos
|   |                         |-- asset
|   |                               |-- client // place the client calling class
|   |                                     |-- AssetClient.java
|   |                               |-- contract // place the Java contract class
|   |                                     |-- Asset.java
|   |-- test
|       |-- resources // store code resource files
|           |-- applicationContext.xml // project configuration file
|           |-- ca.crt // blockchain ca certificate
|           |-- node.crt // node ca certificate
|           |-- node.key // node private key
|           |-- contract.properties // file that stores the deployment contract␣
↪address
|           |-- log4j.properties // log configuration file
|           |-- contract //store Solidity contract files
|                   |-- Asset.sol
|                   |-- Table.sol
|
|-- tool
    |-- asset_run.sh // project running script
```

Project introduced Web3SDK

The project's `build.gradle` file has been introduced to Web3SDK and no need to be modified. The introduction method is as follows:

- Web3SDK introduces the related jar package of Ethereum's solidity compiler, so you need to add Ethereum's remote repository to the `build.gradle` file:

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

- introduce Web3SDK jar package

```
compile ('org.fisco-bcos: web3sdk: 2.0.3')
```

Certificate and configuration file

- Blockchain node certificate configuration

Copy the SDK certificate corresponding to the blockchain node

```
# come into~directory
# copy the node certificate to the project's resource directory
$ cd ~
$ cp fisco/nodes/127.0.0.1/sdk/* asset-app/src/test/resources/
```

- applicationContext.xml

Note:

If the rpc_listen_ip set in the chain is 127.0.0.1 or 0.0.0.0 and the channel_port is 20200, the `applicationContext.xml` configuration does not need to be modified. If the configuration of blockchain node is changed, you need to modify `applicationContext.xml`. For details, please refer to SDK Usage Document.

5.2.5 Business development

We have introduced how to introduce and configure Web3SDK in your own project. In this section, we will introduce how to call a contract through Java program, as well as use an example asset management to explain. The asset-app project already contains the full source code of the sample, which users can use directly. Now we introduces the design and implementation of the core class `AssetClient`.

`AssetClient.java`: to implement contract deployment and calling by calling `Asset.java`. The path is `/src/main/java/org/fisco/bcos/asset/client`. The initialization and the calling process are carrying out in this class.

- initialization

The main function of the initialization code is to construct the Web3j and Credentials' objects, which are needed to be used when creating the corresponding contract class object (calling the contract class's deploy or load function).

```
// initialize in the function 'initialize'
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
Service service = context.getBean(Service.class);
service.run();

ChannelEthereumService channelEthereumService = new ChannelEthereumService();
channelEthereumService.setChannelService(service);
// initialize the Web3j object
Web3j web3j = Web3j.build(channelEthereumService, 1);
// initialize the Credentials object
Credentials credentials = Credentials.create(Keys.createEcKeyPair());
```

- construct contract class object

You can use deploy or load functions to initialize the contract object. They are used in different scenarios. Deploy applies to the initial deployment contract, and load is used when the contract has been deployed and the contract address is known.

```
// deploy contract
Asset asset = Asset.deploy(web3j, credentials, new StaticGasProvider(gasPrice,␣
↪gasLimit)).send();
// load contract address
Asset asset = Asset.load(contractAddress, web3j, credentials, new␣
↪StaticGasProvider(gasPrice, gasLimit));
```

- interface calling

Use the contract object to call the corresponding interface and handle the returned result.

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

5.2.6 Running

So far we have introduced all the processes of asset management application by using blockchain and how to implement the functions. Then we can run project and test whether the function is normal.

- compilation

```
# switch to project directory
$ cd ~/asset-app
# compile project
$ ./gradlew build
```

After the compilation is successful, `dist` directory will be generated under the project root directory. There is an `asset_run.sh` script in the dist directory to simplify project operation. Now let's start verifying the requirements set out in this article.

- deploy `Asset.sol` contract

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

- register asset

```
$ 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
```

- query asset

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

- transfer asset

```
$ 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
```

Summary: By now, we have built an application based on FISCO BCOS alliance chain through contract development, contract compilation, SDK configuration and business development.

5.3 Deploy Multi-Group Blockchain System

This chapter takes the star networking and parallel multi-group networking as an example to guide you to the following.

- Learn to deploy multi-group blockchain with `build_chain.sh` shell script;
- Understand the organization of the multi-group blockchain created by `build_chain.sh`
- Learn how to start the blockchain node and get the consensus status of each group through the log;
- Learn how to send transactions to the given group and get the block generation status through the log;
- Understand node management of the blockchain system, including how to add/remove the given consensus node;
- Learn how to create a new group.

Important:

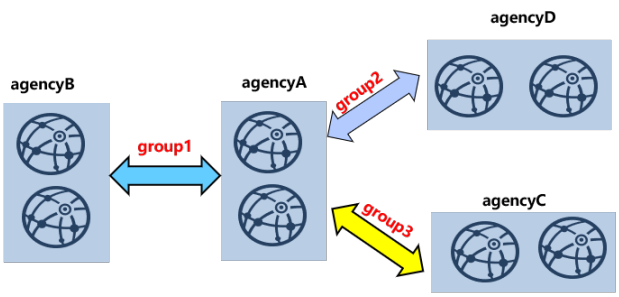
- `build_chain.sh` is suitable for developers and users to use quickly, does not support expansion operations
- Build an enterprise-level business chain, recommend to use generator

---

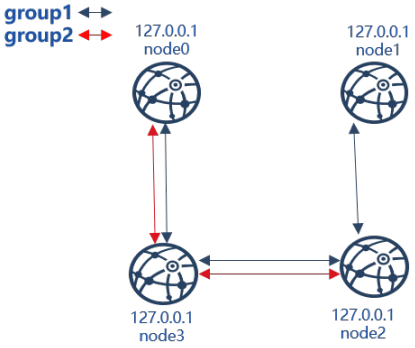
5.3.1 Introduction of star networking topology and parallel multi-group networking topology

As shown in the following figure, the star networking topology and the parallel multi-group networking topology are two widely used networking methods in blockchain applications.

- **Star networking topology**: The nodes of the central agency belong to multiple groups, and runs multiple institutional applications. Nodes of the other agencies belongs to different groups and runs their respective applications;
- **Parallel multi-group networking topology**: Each node in the blockchain belongs to multiple groups and can be used for multi-service expansion or multi-node expansion of the same service.



Star networking topology



Parallel multi-group networking topology

The following is a detailed description of how to deploy a seven-node star networking blockchain system and a four-node parallel multi-group networking blockchain.

5.3.2 Installation dependency

Before deploying the FISCO BCOS blockchain, you need to install software such as `openssl`, `curl`, etc. The specific commands are as follows:

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

# Ubuntu
$ sudo apt install -y openssl curl

# Mac OS
$ brew install openssl curl
```

5.3.3 Star networking topology

In this chapter, we deploy a star networking blockchain system with four agencies, three groups and seven nodes in the local machine.

Here is the detailed configuration of star networking blockchain:

- `agencyA`: belongs to `group1`、`group2`、`group3`, including 2 nodes with the same IP address `127.0.0.1`;
- `agencyB`: belongs to `group1`, including 2 nodes with the same IP address `127.0.0.1`;
- `agencyC`: belongs to `group2`, including 2 nodes with the same IP address `127.0.0.1`;
- `agencyD`: belongs to `group3`, including 2 nodes with the same IP address `127.0.0.1`.

In a star network topology, the core node (in this case, the agencyA node) belongs to all groups and has a high load. It is recommended to deploy it separately on a machine with better performance.

Important:

- In the actual application scenario, it is **not recommended to deploy multiple nodes on the same machine**. It is recommended to select the number of deployed nodes in one machine according to the **machine loading**. Please refer to the hardware configuration
- **In a star network topology**, the core node (in this case, the agencyA node) belongs to all groups and has a high load. It is recommended to deploy it separately on a machine with better performance.
- **When operating in different machines, please copy the generated IP folder to the corresponding machine to start. The chain building operation only needs to be performed once!**

Generate configuration for star networking blockchain

`build_chain.sh` supports to generate configurations for blockchain with any topology, you can use this script to build configuration for the star networking blockchain.

Prepare for dependency

- Create an operation directory

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

- Download the build_chain.sh script

```
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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
```

Generate configuration for star networking blockchain system

```
# Generate ip_list(configuration)
$ 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

# Check the content of ip_list
$ cat ipconf
# Meaning of the space-separated parameters:
# ip:num: IP of the physical machine and the number of nodes
# agency_name: agency name
# group_list: the list of groups the nodes belong to, groups are separated by comma
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
```

Create node configuration folder for star networking blockchain using build_chain script

Please refer to the build_chain for more parameter descriptions of `build_chain.sh`.

```
# Generate a blockchain of star networking and make sure ports 30300~30301, 20200~
↪20201, 8545~8546 of the local machine are not occupied
$ 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
================================================================
......Here to omit other outputs......
================================================================
[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
[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

# The generated node file is as follows:
nodes
|-- 127.0.0.1
|   |-- fisco-bcos
|   |-- node0
|   |   |-- conf  # node configuration folder
|   |   |   |-- 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 # stores the information of Node ID
```

```
|   |   |-- config.ini # node configuration file
|   |   |-- start.sh  # shell script to start the node
|   |   `-- stop.sh   # shell script to stop the node
|   |-- node1
|   |   |-- conf
......omit other outputs here......
```

Note: If the generated nodes belong to different physical machines, the blockchain nodes need to be copied to the corresponding physical machine.

Start node

FISCO-BCOS provides the `start_all.sh` and `stop_all.sh` scripts to start and stop the node.

```
# Switch to the node directory
$ cd ~/fisco/nodes/127.0.0.1

# Start the node
$ bash start_all.sh

# Check node process
$ 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
```

Check consensus status of groups

When no transaction is sent, the node with normal consensus status will output +++ log. In this example, `node0` and `node1` belong to `group1`, `group2` and `group3`; `node2` and `node3` belong to `group1`; `node4` and `node5` belong to `group2`; `node6` and `node7` belong to `group3`. Check the status of node by `tail -f node*/log/* | grep "++"`.

Important:

Node with normal consensus status prints +++ log, fields of +++ log are defined as:

- `g:`: group ID;
- `blkNum`: the newest block number generated by the Leader node;
- `tx`: the number of transactions included in the new block;
- `nodeIdx`: the index of this node;
- `hash`: hash of the newest block generated by consensus nodes.

```
# Check if node0 group1 is normal(Ctrl+c returns to the command line)
$ 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....

# Check if node0 group2 is normal
$ 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...

# ... To check node1, node2 node for each group is normal or not, refer to the␣
↪above operation method...

# Check if node3 group1 is normal
$ 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...

# Check if node5 group2 is normal
$ 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...
```

Configuration console

The console connects to the FISCO BCOS node through the Web3SDK, it is used to query the blockchain status and deploy the calling contract, etc. Detailed instructions of console is here.

Important: The console relies on Java 8 and above, and Ubuntu 16.04 system needs be installed with openjdk 8. Please install Oracle Java 8 or above for CentOS.

```
# Switch back to ~/fisco folder
$ cd ~/fisco

# Download console
$ bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪download_console.sh)

# Switch to console directory
$ cd console

# Copy node certificate of group 2 to the configuration directory of console
$ cp ~/fisco/nodes/127.0.0.1/sdk/* conf/

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

Important: When connecting node with the console, we should make sure that the connected nodes are in the group configured by the console

The configuration of the console configuration file `conf/applicationContext.xml` is as follows.. The console is connected to three groups from node0 (`127.0.0.1:20200`), to get more information of console configuration, please refer to here.

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

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

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

      <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.
↪handler.GroupChannelConnectionsConfig">
        <property name="allChannelConnections">
        <list>
            <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="1" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
            <bean id="group2"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="2" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
            <bean id="group3"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
                <property name="groupId" value="3" />
                    <property name="connectionsStr">
                    <list>
                    <value>127.0.0.1:20200</value>
                    </list>
                    </property>
            </bean>
        </list>
        </property>
        </bean>

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

```
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>
</beans>
EOF
```

Start the console

```
$ bash start.sh
# The following outputted information means the console has been successfully␣
↪started. Otherwise, please check if the port configuration of the nodes in conf/
↪applicationContext.xml is correct.
=====================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ ______  ______   ______   ______       _______   ______   ______   ______
|        |      \/      \ /      \ /      \     |       \ /      \ /      \ /      ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__     | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \    | $$  \$$    \| $$     | $$  | $$    | $$    $| $$     | $$  | $$\$$    ␣
↪\
| $$$$$    | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$      _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$     |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$      \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

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

Send transactions to groups

In the above section, we learned how to configure the console, this section will introduce how to send transactions to groups through the console.

Important: In the group architecture, the ledgers are independent in each group. And sending transactions to one group only increases the block number of this group but not others

Send transactions through console

```
# ... Send HelloWorld transaction to group1...
$ [group:1]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# Check the current block number of group1, if the block number is increased to 1,␣
↪ it indicates that the blockchain is normal. Otherwise, please check if group1␣
↪is abnormal.
$ [group:1]> getBlockNumber
1

# ... Send HelloWorld transaction to group2...
# Switch to group2
$ [group:1]> switch 2
Switched to group 2.
```

```
# Send transaction to group2, return a transaction hash indicates that the␣
↪transaction is deployed successfully, otherwise, please check if the group2␣
↪works normally.
$ [group:2]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# Check the current block number of group2, if the block number is increased to 1,␣
↪ it indicates that the blockchain is normal. Otherwise, please check if group1␣
↪is abnormal
$ [group:2]> getBlockNumber
1

# ... Send transaction to group3...
# Switch to group3
$ [group:2]> switch 3
Switched to group 3.
# Send transaction to group3, return a transaction hash indicates that the␣
↪transaction is deployed successfully, otherwise, please check if the group2␣
↪works normally.
$ [group:3]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# Check the current block number of group3, if the block number is increased to 1,␣
↪ it indicates that the blockchain is normal. Otherwise, please check if group1␣
↪is abnormal
$ [group:3]> getBlockNumber
1

# ... Switch to group 4 that does not exist: the console prompts that group4 does␣
↪not exist, and outputs the current group list ...
$ [group:3]> switch 4
Group 4 does not exist. The group list is [1, 2, 3].

# Exit the console
$ [group:3]> exit
```

Check the log

After the block is generated, the node will output `Report` log, which contains fields with following definitions:

```
# Switch the node directory
$ cd ~/fisco/nodes/127.0.0.1

# Check the block generation status of group1: new block generated
$ 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

# Check the block generation status of group2: new block generated
$ 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

# Check the block generation status of group3: new block generated
$ 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
```

Node joins the group

Through the console, FISCO BCOS can add the specified node to the specified group, or delete the node from the specified group. For details, please refer to the node admission management manual, for console configuration, please reference console operation manual.

Taking how to join node2 to group2 as an example, this chapter introduces how to add a new node to an existing group.

Copy group2 configuration to node2

```
# Switch to node directory
$ cd ~/fisco/nodes/127.0.0.1

# ... Copy group2 configuration of node0 to node2 ...
$ cp node0/conf/group.2.* node2/conf

# ...Restart node2(make sure the node is in normal consensus after restart)...
$ cd node2 && bash stop.sh && bash start.sh
```

Get the ID of node2

```
# Please remember the node ID of node2. Add node2 to group2 needs the node ID.
$ cat conf/node.nodeid
6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1c1
```

Send commands to group2 through the console to add node2 into group2

```
# ...Go back to the console directory and launch the console (direct boot to␣
↪group2)...
$ cd ~/fisco/console && bash start.sh 2

# ...Join node2 as a consensus node through the console...
# 1. Check current consensus node list
$ [group:2]> getSealerList
[
   ␣
↪9217e87c6b76184cf70a5a77930ad5886ea68aefbcce1909bdb799e45b520baa53d5bb9a5edddeab94751df179d54d41
↪
   ␣
↪227c600c2e52d8ec37aa9f8de8db016ddc1c8a30bb77ec7608b99ee2233480d4c06337d2461e24c26617b6fd53acfa61
↪
   ␣
↪7a50b646fcd9ac7dd0b87299f79ccaa2a4b3af875bd0947221ba6dec1c1ba4add7f7f690c95cf3e796296cf4adc989f4
↪
   ␣
↪8b2c4204982d2a2937261e648c20fe80d256dfb47bda27b420e76697897b0b0ebb42c140b4e8bf0f27dfee64c946039
]
# 2. Add node2 to the consensus node
# The parameter after addSealer is the node ID obtained in the previous step
$ [group:2]> addSealer␣
↪6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1c
{
    "code":0,
    "msg":"success"
}
# 3. Check current consensus node list
$ [group:2]> getSealerList
[
   ␣
↪9217e87c6b76184cf70a5a77930ad5886ea68aefbcce1909bdb799e45b520baa53d5bb9a5edddeab94751df179d54d41
↪
   ␣
↪227c600c2e52d8ec37aa9f8de8db016ddc1c8a30bb77ec7608b99ee2233480d4c06337d2461e24c26617b6fd53acfa61
↪
   ␣
↪7a50b646fcd9ac7dd0b87299f79ccaa2a4b3af875bd0947221ba6dec1c1ba4add7f7f690c95cf3e796296cf4adc989f4
↪
   ␣
↪8b2c4204982d2a2937261e648c20fe80d256dfb47bda27b420e76697897b0b0ebb42c140b4e8bf0f27dfee64c946039
↪
```

```
↪6dc585319e4cf7d73ede73819c6966ea4bed74aadbbcba1bbb777132f63d355965c3502bed7a04425d99cdcfb7694a1
↪# new node
]
# Get the current block number of group2
$ [group:2]> getBlockNumber
2

#... Send transaction to group2
# Deploy the HelloWorld contract and output contract address. If the contract␣
↪fails to deploy, please check the consensus status of group2
$ [group:2] deploy HelloWorld
contract address:0xdfdd3ada340d7346c40254600ae4bb7a6cd8e660

# Get the current block number of group2, it increases to 3. If not, please check␣
↪the consensus status of group2
$ [group:2]> getBlockNumber
3

# Exit the console
$ [group:2]> exit
```

Check the block generation status of the new node through log

```
# Switch to the node directory
cd ~/fisco/nodes/127.0.0.1
# Check the consensus status of the node(Ctrl+c returns the command line)
$ 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...
......Other outputs are omitted here......

# Check the block generation status of node2 and group2: new block generated
$ 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 also reports a block with block number 3, indicating that node2 has joined␣
↪group2.
```

Stop the node

```
# Back to the node directory && stop the node
$ cd ~/fisco/nodes/127.0.0.1 && bash stop_all.sh
```

5.3.4 Parallel multi-group networking topology

Deploying parallel multi-group networking blockchain is similar with deploying star networking blockchain. Taking a four-node two-group parallel multi-group blockchain as an example:

- group 1: includs 4 nodes with the same IP `127.0.0.1`;
- group 2: includs 4 nodes with the same IP `127.0.0.1`.

In a real application scenario, it is not recommended to deploy multiple nodes on the same machine. It is recommended to select the number of deployed nodes according to the machine load. To demonstrate the parallel multi-group expansion process, only group1 is created here first. In a parallel multi-group scenario, node join and exit group operations are similar to star networking topology.

Important:

- In a real application scenario, **it is not recommended to deploy multiple nodes the same machine**, It is recommended to determine the number of deployed nodes according to the machine load
- To demonstrate the parallel multi-group expansion process, only group1 is created here first
- In a parallel multi-group scenario, the operations of node joining into a group or leaving from a group are similar to star networking blockchain.

Build blockchain with a single group and 4 nodes

Generate a single-group four-node blockchain node configuration folder with the build_chain.sh script

```
$ mkdir -p ~/fisco && cd ~/fisco
# Download build_chain.sh script
$ 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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
#Build a local single-group four-node blockchain (in a production environment, it␣
↪is recommended that each node be deployed on a different physical machine)
$ 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
```

Start all nodes

```
# Switch to the node directory
$ cd ~/fisco/multi_nodes/127.0.0.1
$ bash start_all.sh

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

Check consensus status of nodes

```
# Check consensus status of node0(Ctrl+c returns to the command line)
$ 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...
```

```
# Check consensus status of node1
$ 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...

# Check consensus status of node2
$ 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...

# Check consensus status of node3
$ 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...
```

Add group2 to the blockchain

The `genesis` configuration files in each group of parallel multi-group networking blockchain are almost the same, except group ID [group].id, which is the group number.

```
# Switch to the node directory
$ cd ~/fisco/multi_nodes/127.0.0.1

# Copy the configuration of group 1
$ cp node0/conf/group.1.genesis group.2.genesis

# Modify group ID
$ sed -i "s/id=1/id=2/g" group.2.genesis
$ cat group.2.genesis | grep "id"
# Have modified to    id=2

# Copy the configuration to each node
$ 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

# Restart node
$ bash stop_all.sh
$ bash start_all.sh
```

Check consensus status of the group

```
# Check the consensus status of node0 group2
$ 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...

# Check the consensus status of node1 group2
$ 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...

# Check the consensus status of node2 group2
$ 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...

# Check the consensus status of node3 group2
```

```
$ 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...
```

Send transactions to groups

Download console

```
# If you have never downloaded the console, please do the following to download␣
↪the console, otherwise copy the console to the ~/fisco directory:
$ cd ~/fisco
# Download console
$ bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪download_console.sh)
```

Configuration console

```
# Get 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

# Switch to console subdirectory
$ cd console
# Copy node certificate
$ cp ~/fisco/multi_nodes/127.0.0.1/sdk/* conf
```

The configuration of the console configuration file `conf/applicationContext.xml` is created as follows. Two groups (group1 and group2) are configured on node0 (127.0.0.1:20100):

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

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

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

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

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

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

Send transactions to groups via console

```
# ... Start console ...
$ bash start.sh
# The following information output indicates that the console is successfully␣
↪started. If the startup fails, check whether the certificate configuration and␣
↪the channel listen port configuration are correct.
=====================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______   _______       ________  _______  _______  _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       ␣
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\    | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$    | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$    | $$    $| $$      | $$  | $$\$$    ␣
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$    | $$$$$$$| $$   __| $$  | $$_\$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$    | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$    | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$      \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$
↪$

=====================================================================================
# ... Send transaction to group 1...
# Get the current block number
$ [group:1]> getBlockNumber
0
# Deploy the HelloWorld contract to group1. If the deployment fails, check whether␣
↪the consensus status of group1 is normal
$ [group:1]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# Get the current block number. If the block number is not increased, please check␣
↪if the group1 is normal
```

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

# ... send transaction to group 2...
# Switch to group2
$ [group:1]> switch 2
Switched to group 2.
# Get the current block number
$ [group:2]> getBlockNumber
0
# Deploy HelloWorld contract to group2
$ [group:2]> deploy HelloWorld
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
# Get the current block number. If the block number is not increased, please check␣
↪if the group1 is normal
$ [group:2]> getBlockNumber
1
# Exit console
$[group:2]> exit
```

Check block generation status of nodes through log

```
# Switch to the node directory
$ cd ~/fisco/multi_nodes/127.0.0.1/

# Check block generation status of group1, and to see that the block with block␣
↪number of 1 belonging to group1 is reported.
$ 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

# Check block generation status of group2, and to see that the block with block␣
↪number of 1 belonging to group2 is reported.
$ 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
```

Stop nodes

```
# Back to nodes folder && stop the node
$ cd ~/fisco/multi_nodes/127.0.0.1 && bash stop_all.sh
```

5.4 enterprise deployment tools

FISCO BCOS enterprise deployment tools are designed for multi-agency production environments. To ensure the security of the agency's private keys, enterprise deployment tools provides agencies' collaboration to deploy a alliance chain.

This chapter will demonstrate how to use enterprise deployment tools by deploying a **6 nodes 3 agencies 2 groups** alliance chain. For more parameter options, please refer to here

This chapter is a process that multi-agency peer-to-peer deployment and a situation that private key of the agency does not come out of intranet. The tutorial for generating configuration files of all agency nodes through single agency's clickstart can refer to FISCO BCOS Enterprise Deployment Tool ClickStart deployment

5.4.1 Download and install

download

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

install

This operation requires sudo permission.

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

Check whether the installation is successful. If it is, output usage: generator xxx

```
./generator -h
```

download fisco-bcos binary

download the latest fisco-bcos binary to `./meta`

```
./generator --download_fisco ./meta
```

check fisco-bcos version

Output will be: FISCO-BCOS Version : x.x.x-x

```
./meta/fisco-bcos -v
```

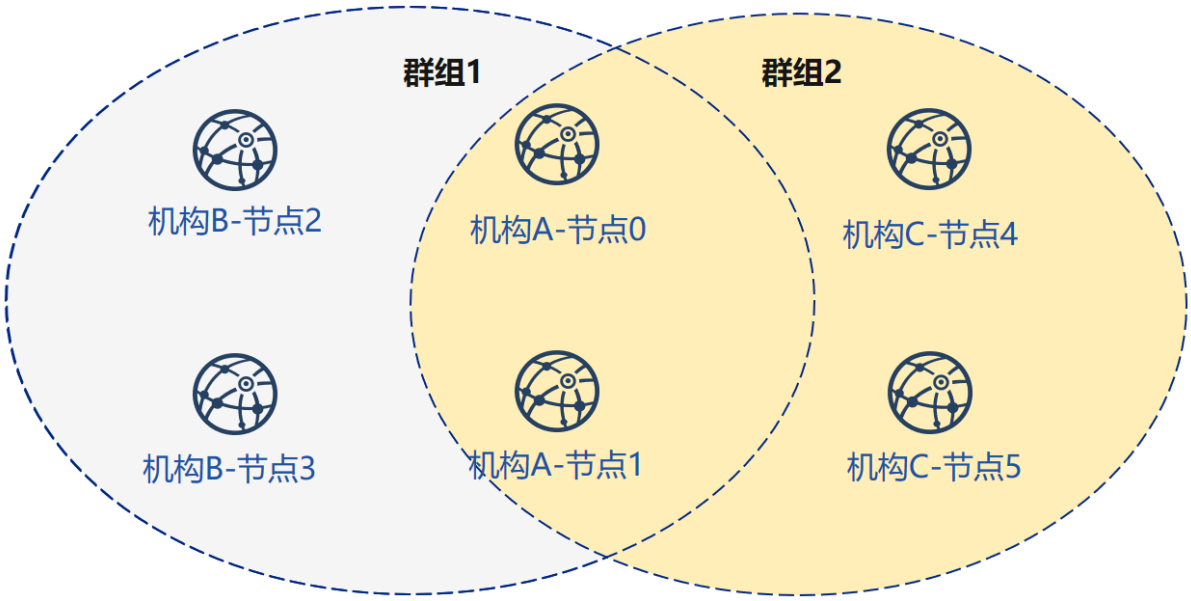
PS: If someone want to use Source Code Compile fisco-bcos binary, they need to replace the binary in the `meta` folder with the compiled binary.

5.4.2 Typical example

To ensure the security of the agency's private keys, enterprise deployment tools provides a security way to build chain between agencies. This chapter will demonstrate how to build chain between agencies in a deployment mode of **6 nodes 3 agencies 2 groups**.

Node networking overview

A networking mode of a 6 nodes 3 agencies 2 groups is shown as follows. Agency B and agency C are located in Group 1 and Group 2, agency A belongs to both Group 1 and Group 2.



Machine address

IP address of each node and port are as follows:

Important: For the VPS server in the cloud server, the RPC listening address needs to be the real address in the network interface card (such as the internal network address or 127.0.0.1), which may be inconsistent with the SSH server address where the user logs in.

---

cooperate agencies

Building chain involves the cooperation between multiple agencies, including:

- Certificate authority agency
- alliance chain member agency(next named “agency”)

Key process

In this section, we briefly provide How **Certificate authority agency** and **alliance chain member agency** cooperate to build a blockchain.

1. Initialize chain certificate

1. Certificate authority agency operation:
    - Generate chain certificate

2. Generate group 1

1. Certificate authority agency operations
    - generate agency certificate
    - send certificate to agencies
2. Operation between agencies

- modify the configuration file `node_deployment.ini`
- generate node certificate and node P2P port address file `peers.txt`

3. Select one of agencies to generate `group.genesis`
   - collect all node certificates in the group
   - modify configuration file `group_genesis.ini`
   - generate genesis block files for the group
   - distribute genesis block files to other agencies
4. Operation between agencies: generating nodes
   - collect P2P port address files of other nodes in the group
   - generate node
   - start node

3. Initialize a new institution

1. Certificate authority agency operations
   - generate agency certificate
   - send certificate to new agency

4. Generate group2

1. New agency operation
   - modify the configuration file `node_deployment.ini`
   - generate node certificate and node P2P port address file
2. Select one of agencies as group to generate genesis block
   - collect all node certificates in the group
   - modify configuration file `group_genesis.ini`
   - generate genesis block files for the group
   - distribute genesis block files to other agency
3. New agency independent operation: generate nodes
   - collect P2P port address files of other nodes in the group
   - generate nodes
   - start nodes
4. Existing agency's operations: configure new groups
   - collect P2P port address files of other nodes in the group
   - configure P2P port address of the new group and the new nodes
   - restart nodes

5. Existing nodes join group 1

1. Group 1 original agency operation:
    - send group 1 genesis block to existing node
    - configure console
    - get the entering node nodeid
    - add nodes to group1 by using console

5.4.3 Alliance chain initialization

Simply, all the operations in this example are performed on the local machine. We use different catalogs to simulate different agencies' environment and use the file copy operation to simulate the sending in the network. After performing `Download and Install` in the tutorial, please copy the generator to the corresponding agency's generator directory.

Institutional initialization

We use **generator downloaded from the tutorial as certificate agency**.

Initialize agencyA

```
cp -r ~/generator ~/generator-A
```

Initialize agencyB

```
cp -r ~/generator ~/generator-B
```

Initialize chain certificate

Operating on a Certificate authority agency. A single chain has a unique chain certificate ca.crt.

use `--generate_chain_certificate` command to generate chain certificate

operate in the certificate agency directory:

```
cd ~/generator
```

```
./generator --generate_chain_certificate ./dir_chain_ca
```

view chain certificate and private key:

```
ls ./dir_chain_ca
```

```
# the above order has explained
# From left to right,  they are chain certificate, chain private key, and␣
↪certificate configuration file
ca.crt  ca.key   cert.cnf
```

5.4.4 AgencyA, B to build group 1

Initialize agencyA

In the tutorial, for operating simply, the certificate of agency and the private key are directly generated. In actual application, the private key `agency.key` should be generated locally by agency first, and then the certificate request file is generated, and the certificate `agency.crt` is obtained from the certificate agency.

operate in the certificate generator directory:

```
cd ~/generator
```

generate agencyA certificate:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyA
```

view agency certificate and private key:

```
ls dir_agency_ca/agencyA/
```

```
# the above order has explained
# From left to right, they are agency certificate, agency private key,␣
↪intermediate file of chain certificate agency, chain certificate, certificate␣
↪configuration file
agency.crt    agency.key    ca-agency.crt ca.crt    cert.cnf
```

For sending the chain certificate, agency certificate, and agency private key to agencyA, we use an example is to send the certificate from the certificate agency to the corresponding agency through the file copy, and put the certificate in the subdirectory of meta which is agency's working directory.

```
cp ./dir_chain_ca/ca.crt ./dir_agency_ca/agencyA/agency.crt ./dir_agency_ca/
↪agencyA/agency.key ~/generator-A/meta/
```

Initialize agencyB

operate in the certificate generator directory:

```
cd ~/generator
```

generate agencyB certificate:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyB
```

For sending the chain certificate, agency certificate, and agency private key to agencyB, we use an example is to send the certificate from the certificate agency to the corresponding agency through the file copy, and put the certificate in the subdirectory of meta which is agency's working directory.

```
cp ./dir_chain_ca/ca.crt ./dir_agency_ca/agencyB/agency.crt ./dir_agency_ca/
↪agencyB/agency.key ~/generator-B/meta/
```

Important: Only one root certificate, ca.crt, can be used in an alliance chain. Do not generate multiple root certificates and private keys when deploying multiple servers. A group can only have one genesis block group.x.genesis.

AgencyA modifies configuration file

node_deployment.ini is the node configuration file. Enterprise deployment tool generates the corresponding node certificate according to the configuration of `node_deployment.ini` and the node configuration folder etc..

AgencyA modifies the `node_deployment.ini` in the conf folder, as shown below:

execute the following command in the ~/generator-A directory

```
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
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
EOF
```

AgencyA modifies configuration file

AgencyB modifies the `node_deployment.ini` in the conf folder, as shown below:

execute the following command in the ~/generator-B directory

```
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
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
```

```
p2p_listen_port=30303
channel_listen_port=20203
jsonrpc_listen_port=8548
EOF
```

AgencyA generates and sends node information

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

AgencyA generates the node certificate and the P2P connection information file. In this step, we need to use the above configuration `node_deployment.ini` and the agency certificate and private key in the agency meta folder.

```
./generator --generate_all_certificates ./agencyA_node_info
```

view generated files:

```
ls ./agencyA_node_info
```

```
# the above order has explained
# From left to right, they are the node certificate that needs to be interacted␣
↪with the agencyA and the file that node P2P connects to the address (the node␣
↪information of agency generated by the node_deployment.ini)
cert_127.0.0.1_30300.crt cert_127.0.0.1_30301.crt peers.txt
```

When the agency generates a node, it needs to specify the node P2P connection address of other nodes. Therefore, AgencyA needs to send the node P2P connection address file to AgencyB.

```
cp ./agencyA_node_info/peers.txt ~/generator-B/meta/peersA.txt
```

AgencyB generates and sends node information

execute the following command in the ~/generator-B directory

```
cd ~/generator-B
```

AgencyB generates the node certificate and the P2P connection information file:

```
./generator --generate_all_certificates ./agencyB_node_info
```

The agency that generates the genesis block needs the node certificate. In the example, the agencyA generates the genesis block. Therefore, in addition to sending the node P2P connection address file, the agencyB needs to send the node certificate to agencyA.

send certificate

```
cp ./agencyB_node_info/cert*.crt ~/generator-A/meta/
```

send the node P2P connection address file

```
cp ./agencyB_node_info/peers.txt ~/generator-A/meta/peersB.txt
```

AgencyA generates the genesis block of group1

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

AgencyA modifies `group_genesis.ini` in the conf folder. For configuration items, refer to Manuals:

```
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
```

After the command is executed, the ./conf/group_genesis.ini file will be modified:

```
;command interpretation
[group]
;group id
group_id=1

[nodes]
;AgencyA node p2p address
node0=127.0.0.1:30300
;AgencyA node p2p address
node1=127.0.0.1:30301
;AgencyB node p2p address
node2=127.0.0.1:30302
;AgencyB node p2p address
node3=127.0.0.1:30303
```

In the tutorial, we choose agencyA to generate genesis block of group. But in the actual production, you can negotiate with alliance chain committee to make choice.

This step will generate the genesis block of group_genesis.ini according to the node certificate configured in the meta folder of agencyA. In the tutorial, the agencyA's meta is required to have the node certificates name as `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`. This step requires the node certificate of agencyB.

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

distribute group1 genesis block to AgencyB:

```
cp ./group/group.1.genesis ~/generator-B/meta
```

AgencyA generates the node to which it belongs

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

AgencyA generates the node to which it belongs. This command generates the corresponding node configuration folder according to the user-configured file `node_deployment.ini`:

Note. The node P2P connection information `peers.txt` specified in this step is the connect information of other nodes in the group. In the case of multiple agencies networking, they need to be merged.

```
./generator --build_install_package ./meta/peersB.txt ./nodeA
```

view the generated node configuration folder:

```
ls ./nodeA
```

```
# command interpretation, displayed in tree style here
# The generated folder nodeA information is as follows
├── monitor # monitor script
├── node_127.0.0.1_30300 # node configuration folder with server address 127.0.0.1␣
↪and port number 30300
├── node_127.0.0.1_30301
├── scripts # node related tool script
├── start_all.sh # node startups script in batch
└── stop_all.sh # node stops script in batch
```

agencyA startups node:

```
bash ./nodeA/start_all.sh
```

view node process:

```
ps -ef | grep fisco
```

```
# command interpretation
# you can see the following process
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
```

AgencyB generates the node to which it belongs

execute the following command in the ~/generator-B directory

```
cd ~/generator-B
```

AgencyB generates the node to which it belongs. This command generates the corresponding node configuration folder according to the user-configured file `node_deployment.ini`:

```
./generator --build_install_package ./meta/peersA.txt ./nodeB
```

agencyB startups node:

```
bash ./nodeB/start_all.sh
```

Note: Startup node only needs to send the node folder corresponding to ip address. For example, the server of 127.0.0.1 only needs the node configuration folder corresponding to node_127.0.0.1_port. When deploying multiple machines, you only need to send the generated node folder to the corresponding server.

View group1 node running status

view process:

```
ps -ef | grep fisco
```

```
# command interpretation
# you can see the following process
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
```

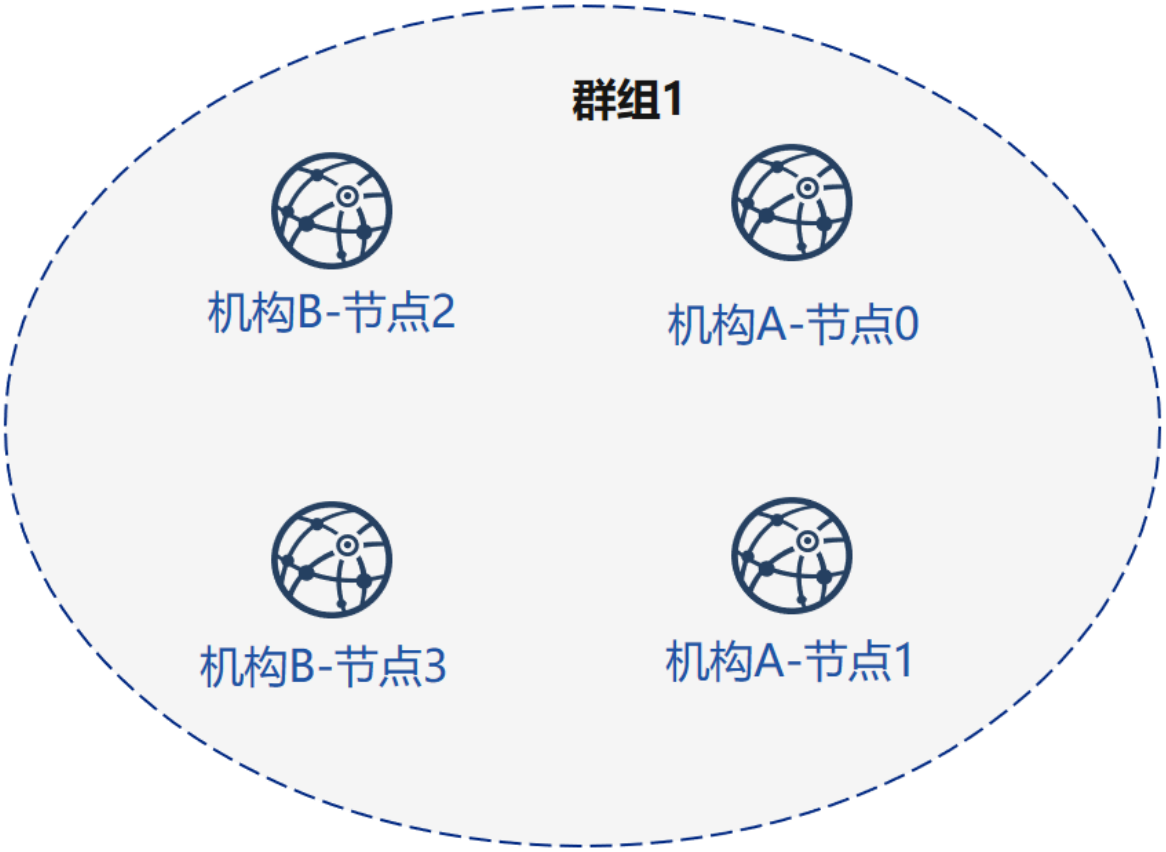
```
fisco  15402     1  0 17:22 pts/2    00:00:00 ~/generator-A/nodeA/node_127.0.0.1_
↪30301/fisco-bcos -c config.ini
fisco  15457     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30302/fisco-bcos -c config.ini
fisco  15498     1  0 17:22 pts/2    00:00:00 ~/generator-B/nodeB/node_127.0.0.1_
↪30303/fisco-bcos -c config.ini
```

view node log:

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# command interpretation
# +++ printed in log is the normal consensus of the node
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...
```

By now, we have completed the operation of agencyA,B to build group1 as shown:



5.4.5 Certificate authority initialize agencyC

operate in the certificate generator directory:

```
cd ~/generator
```

Initialize agencyC Note. Now there is a chain certificate and a private key in the generator directory. In the actual environment, agencyC cannot obtain the chain certificate and the private key.

```
cp -r ~/generator ~/generator-C
```

generate agencyC certificate:

```
./generator --generate_agency_certificate ./dir_agency_ca ./dir_chain_ca agencyC
```

view agency certificate and private key:

```
ls dir_agency_ca/agencyC/
```

```
# command interpretation
# From left to right, they are agency certificate, agency private key,
↪intermediate file of chain certificate agency, chain certificate, certificate
↪configuration file
agency.crt    agency.key    ca-agency.crt ca.crt    cert.cnf
```

For sending the chain certificate, agency certificate, and agency private key to agencyA, we use an example is to send the certificate from the certificate agency to the corresponding agency through the file copy, and put the certificate in the subdirectory of meta which is agency's working directory.

```
cp ./dir_chain_ca/ca.crt ./dir_agency_ca/agencyC/agency.crt ./dir_agency_ca/
↪agencyC/agency.key ~/generator-C/meta/
```

5.4.6 AgencyA,C build group2

Next, agencyC needs to perform a new group establishment operation with A. We take an example of agencyC generating genesis block.

AgencyA sends node information

Since agencyA has generated the node certificate and the peers file, we only need to send the previous generated node P2P connection information and the node certificate to agencyC as follows:

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

In the example, the genesis block of group is generated by agencyC, therefore the node certificate of agencyA and the node P2P connection address file are required, and the above file is sent to agencyC.

send certificate

```
cp ./agencyA_node_info/cert*.crt ~/generator-C/meta/
```

send node P2P connection address file

```
cp ./agencyA_node_info/peers.txt ~/generator-C/meta/peersA.txt
```

AgencyC modifies configuration file

AgencyC modifies `node_deployment.ini` in the conf folder as shown below:

execute the following command in the ~/generator-C directory

```
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
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
p2p_listen_port=30305
channel_listen_port=20205
jsonrpc_listen_port=8550
EOF
```

AgencyC generates and sends node information

execute the following command in the ~/generator-C directory

```
cd ~/generator-C
```

AgencyC generates node certificate and P2P connection information file:

```
./generator --generate_all_certificates ./agencyC_node_info
```

view generated file:

```
ls ./agencyC_node_info
```

```
# command interpretation
# From left to right, they are the node certificate that needs to be interacted␣
↪with the agencyA and the file that node P2P connects to the address (the node␣
↪information of agency generated by the node_deployment.ini)
cert_127.0.0.1_30304.crt cert_127.0.0.1_30305.crt peers.txt
```

When agency generates a node, it needs to specify the node P2P connection address of other nodes. Therefore, agencyC needs to send the node P2P connection address file to agencyA.

```
cp ./agencyC_node_info/peers.txt ~/generator-A/meta/peersC.txt
```

AgencyC generates genesis block of group2

execute the following command in the ~/generator-C directory

```
cd ~/generator-C
```

AgencyC modifies `group_genesis.ini` in the conf folder as shown below:

```
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
```

./conf/group_genesis.ini file will be modified after the command is executed:

```
;command interpretation
[group]
group_id=2

[nodes]
node0=127.0.0.1:30300
;agencyA node p2p address
node1=127.0.0.1:30301
;agencyA node p2p address
node2=127.0.0.1:30304
;agencyC node p2p address
node3=127.0.0.1:30305
;agencyC node p2p address
```

In the tutorial, agency C is chosen to generate genesis block of group. In the actual production, you can negotiate with the alliance chain committee to choose.

This step generates genesis block of group_genesis.ini configuration according to the node certificate configured in the meta folder of agencyC.

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

distribute genesis block of group2 to agencyA:

```
cp ./group/group.2.genesis ~/generator-A/meta/
```

AgencyC generates the node to which it belongs

execute the following command in the ~/generator-C directory

```
cd ~/generator-C
```

```
./generator --build_install_package ./meta/peersA.txt ./nodeC
```

agencyC startups node:

```
bash ./nodeC/start_all.sh
```

```
ps -ef | grep fisco
```

```
# command interpretation
# you can see the following process
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
```

AgencyA initializes group2 for existing nodes

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

Add the group2 configuration file to the existing node. This step adds the genesis block of group2 `group.2.genesis` to all nodes under ./nodeA:

```
./generator --add_group ./meta/group.2.genesis ./nodeA
```

Add the agencyC node connect file peers to the existing node. This step adds the node P2P connection address of `peersC.txt` to all nodes under ./nodeA:

```
./generator --add_peers ./meta/peersC.txt ./nodeA
```

restart agencyA node:

```
bash ./nodeA/stop_all.sh
```

```
bash ./nodeA/start_all.sh
```

View group2 node running status

view node's process:

```
ps -ef | grep fisco
```

```
# command interpretation
# you can see the following process
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
```

view node log:

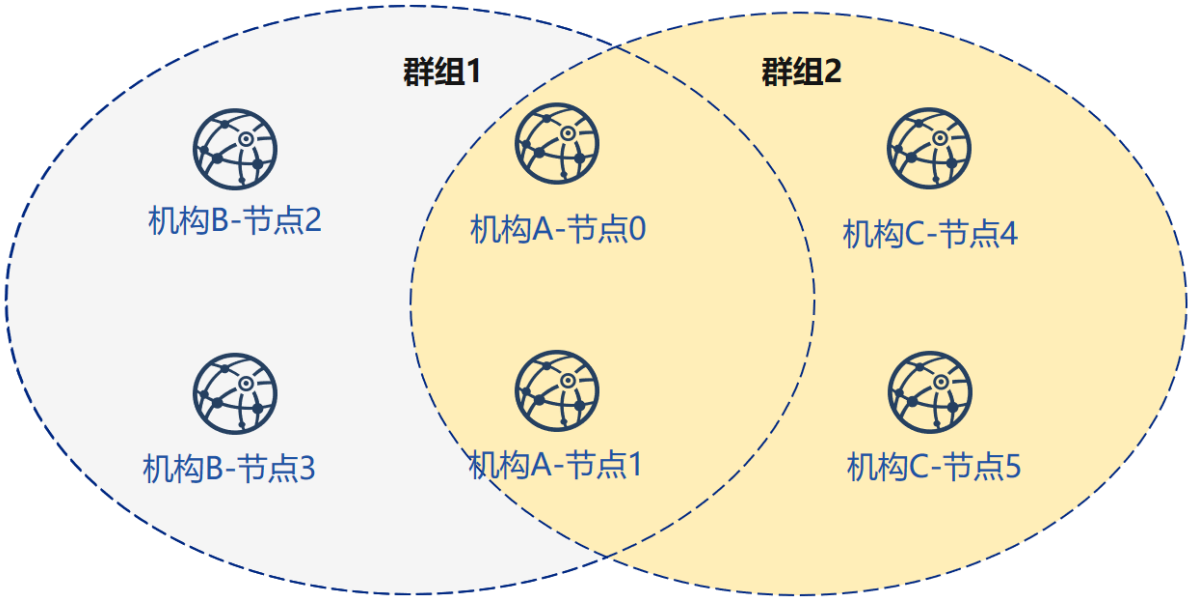
execute the following command in the ~/generator-C directory

```
cd ~/generator-C
```

```
tail -f ./node*/node*/log/log*  | grep +++
```

```
# command interpretation
#+++ rinted in log is the normal consensus of the node
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...
```

By now, we have completed the construction of agencyA,C to build group2 as shown:



5.4.7 Extended Tutorial–agencyC node joins group1

Adding a node to an existing group requires user to send a command by console. The example of adding node to group is as follows:

Now there are nodes of agencyA,B and B in group1. Adding the node of agencyC to group1 needs to get permission of the nodes in the group. To take the node of agencyA as example:

execute the following command in the ~/generator-A directory

```
cd ~/generator-A
```

Send gensis block of group1 to agencyC

Send the configuration file of group1 to agencyC

```
./generator --add_group ./group/group.1.genesis  ~/generator-C/nodeC
```

Currently, `FISCO BCOS` does not support file hot update. It is necessary to restart the node after adding genesis block of group1 to agencyC node.

restart agencyC node:

```
bash ~/generator-C/nodeC/stop_all.sh
```

```
bash ~/generator-C/nodeC/start_all.sh
```

Configure console

agencyA configure console or sdk. In the tutorial, console is used as an example:

Note: This command will complete the console configuration according to the node and group in the user-configured `node_deployment.ini`. User can directly start the console. Please ensure that java is installed before starting.

```
./generator --download_console ./
```

View agencyC node4 information

AgencyA uses the console to join agencyC node4 as observation node. The second parameter needs to be replaced with the joining node 'nodeid', which locates in the `node.nodeid` file of the node folder conf.

View the agencyC node nodeid:

```
cat ~/generator-C/nodeC/node_127.0.0.1_30304/conf/node.nodeid
```

```
# command interpretation
# you can see a nodeid similar to the following. When using the console, you need␣
↪to pass this parameter.
ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c9715
```

Register observation node by using console

start console:

```
cd ~/generator-A/console && bash ./start.sh 1
```

Use the console `addObserver` command to register the node as an observation node. In this step, you need to use the `cat` command to view `node.nodeid` of agencyC node.

```
addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
```

```
# command interpretation
# Successful execution will prompt success
$ [group:1]> addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

exit console:

```
exit
```

View agencyC node 5 information

AgencyA uses console to join node 5 of agencyC as the consensus node. The second parameter needs to be replaced with the joining node 'nodeid', which locates in the `node.nodeid` file of the node folder conf.

View the agencyC node nodeid:

```
cat ~/generator-C/nodeC/node_127.0.0.1_30305/conf/node.nodeid
```

```
# command interpretation
# you can see a nodeid similar to the following. When using the console, you need␣
↪to pass this parameter.
5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa9a
```

Register consensus node by using console

start console:

```
cd ~/generator-A/console && bash ./start.sh 1
```

Use the console `addSealer` command to register the node as a consensus node. In this step, you need to use the `cat` command to view the `node.nodeid` of agencyC node.

```
addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
```

```
# command interpretation
# Successful execution will prompt success
$ [group:1]> addSealer␣
↪5d70e046047e15a68aff8e32f2d68d1f8d4471953496fd97b26f1fbdc18a76720613a34e3743194bd78aa7acb59b9fa
{
        "code":0,
        "msg":"success"
}
```

exit console:

```
exit
```

View node

execute the following command in the ~/generator-C directory

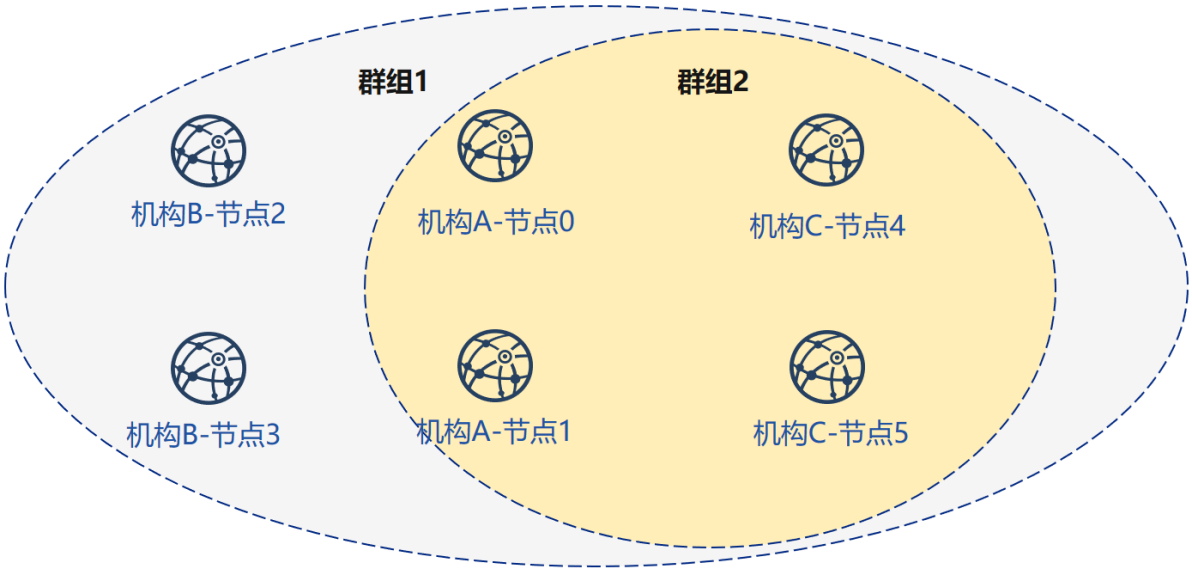
```
cd ~/generator-C
```

view the group1 information in node log:

```
cat node*/node_127.0.0.1_3030*/log/log* | grep g:1  | grep Report
```

```
# command interpretation
# Observation node will only synchronize transaction data, and will not␣
↪synchronize the consensus information in non-transaction status
# ^^^ is the transaction information of the node, and g:1 is the information␣
↪printed by group1
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
```

By now, we have completed all the operations in the tutorial shown.



In this tutorial, we have generated a multi-group architecture alliance chain with a network topology of 3 agencies, 2 groups, and 6 nodes.

If you have problems with this tutorial, please view FAQ.

5.5 Create and manage accounts

FISCO BCOS uses accounts to identify and distinguish each individual user. In a blockchain system by using public and private key, each account corresponds to a pair of public and private keys. The account named by the address string calculated by the secure one-way algorithm such as public keys to hash, that is **account address**. For distinguishing from the address of smart contract and other historical reasons, the account address is often referred to as the **external account address**. The private key only known by user corresponds to the password in the traditional authentication model. Users need to prove that they owns the private key of the corresponding account through secure cryptographic protocol for claiming their ownership of the account, and performing some sensitive account operations.

Important: In the previous tutorials, for simplifying the operation, we operate with the account provided by the tool by default. However, in actual application deployment, users need to create their own accounts and properly save the account private key to avoid the serious security problems such as account private key leakage.

In this article we will specifically introduce the creation, storage and use of accounts. Readers are required to have a certain Linux operating basis.

FISCO BCOS provides script and Web3SDK for creating account, and provides Web3SDK and console for storing account private key. Users can choose to store the account private key as a file in PEM or PKCS12 format according to their requirement. PEM format uses plaintext to store private key, and PKCS12 uses password encryption provided by users to store private key.

5.5.1 Account creation

Use script to create account

1. get script

```
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
```

execute the above command and if you see the following output, you are downloading the correct script, otherwise please try again.

```
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. Use script to generate PEM format private key

- generate private key and address

```
bash get_account.sh
```

Execute the above command to get output similar to the following. It includes the account address and the private key PEM file with the account address as the file name.

```
[INFO] Account Address   : 0xee5fffba2da55a763198e361c7dd627795906ead
[INFO] Private Key (pem) : accounts/0xee5fffba2da55a763198e361c7dd627795906ead.pem
```

- Specify the calculation account address of PEM format

```
bash get_account.sh -k accounts/0xee5fffba2da55a763198e361c7dd627795906ead.pem
```

Execute the above command. The result is as follows

```
[INFO] Account Address   : 0xee5fffba2da55a763198e361c7dd627795906ead
```

3. Use script to generate PKCS12 format private key

- generate private key and address

```
bash get_account.sh -p
```

Execute the above command to get output similar to the following. You can follow the prompts to enter the password, and generate the corresponding p12 file.

```
Enter Export Password:
Verifying - Enter Export Password:
[INFO] Account Address   : 0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1
[INFO] Private Key (p12) : accounts/0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1.p12
```

- Specify the calculation account address of p12 private key. **Enter the p12 file password as prompted**

```
bash get_account.sh -P accounts/0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1.p12
```

Execute the above command. The result is as follows

```
Enter Import Password:
MAC verified OK
[INFO] Account Address   : 0x02f1b23310ac8e28cb6084763d16b25a2cc7f5e1
```

Calling Web3SDK to create an account

```
//create normal account
EncryptType.encryptType = 0;
//create national cryptography account, which uses for sending transaction to␣
↪national blockchain node
// EncryptType.encryptType = 1;
Credentials credentials = GenCredential.create();
//account address
String address = credentials.getAddress();
//account private key
String privateKey = credentials.getEcKeyPair().getPrivateKey().toString(16);
//account public key
String publicKey = credentials.getEcKeyPair().getPublicKey().toString(16);
```

For more details on the operation, to see Creating and Using a Specified External Account.

5.5.2 Account storage

- web3SDK supports loading via private key string or file, so the private key of account can be stored in the database or in a local file.
- Local files support two storage formats, which are PKCS12 encrypted storage and PEM plaintext storage.
- When developing a service, you can select the storage management of private key according to the actual business scenario.

5.5.3 Account using

Console loads private key file

Console provides the account generation script get_account.sh. The generated account file is in the accounts directory, and the account file loaded by console must be placed in this directory.

The console startup methods are as follows:

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

Default startup

Console randomly generates an account, startup with the group number specified in console configuration file.

```
./start.sh
```

Specify group number to startup

Console randomly generates an account, startup with the group number specified on the command line.

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

- Note: The specified group needs to configure bean in console configuration file.

Use PEM private key file to startup

- Startup with the account of the specified pem file. Enter the parameters: group number, -pem, and pem file path

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

Use PKCS12 private key file to startup

- Startup with the account of the specified p12 file. Enter the parameters: group number, -p12, and p12 file path

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

Web3SDK loads private file

If the account private key file in PEM or PKCS12 format is generated by the account generation script get_accounts.sh, the account can be used by loading the PEM or PKCS12 account private key file. There are two classes of private keys to be loaded: P12Manager and PEMManager. P12Manager is used to load the private key file in PKCS12 format. PEMManager is used to load the private key file in PEM format.

- P12Manager usage example: configure the private key file path and password for the PKCS12 account in applicationContext.xml

```
<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>
```

develop code

```
//load Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
P12Manager p12 = context.getBean(P12Manager.class);

//provide password to get ECKeyPair. The password is specified when producing p12␣
↪account file
ECKeyPair p12KeyPair = p12.getECKeyPair(p12.getPassword());

//output the private key and public key in hex string
System.out.println("p12 privateKey: " + p12KeyPair.getPrivateKey().toString(16));
System.out.println("p12 publicKey: " + p12KeyPair.getPublicKey().toString(16));

//generate Credentials for web3sdk using
Credentials credentials = Credentials.create(p12KeyPair);
System.out.println("p12 Address: " + credentials.getAddress());
```

- PEMManager usage example:

configure the private key file path and password for the PEM account in applicationContext.xml

```
<bean id="pem" class="org.fisco.bcos.channel.client.PEMManager" init-method="load"␣
↪>
        <property name="pemFile" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.pem" />
</bean>
```

load with code

```
//load Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext-keystore-sample.xml");
PEMManager pem = context.getBean(PEMManager.class);
ECKeyPair pemKeyPair = pem.getECKeyPair();

//output the private key and public key in hex string
System.out.println("PEM privateKey: " + pemKeyPair.getPrivateKey().toString(16));
System.out.println("PEM publicKey: " + pemKeyPair.getPublicKey().toString(16));

//generate Credentials for web3sdk using
Credentials credentialsPEM = Credentials.create(pemKeyPair);
System.out.println("PEM Address: " + credentialsPEM.getAddress());
```

5.5.4 Account address calculation

The account address of FISCO BCOS is calculated by the ECDSA public key. The hexadecimal of ECDSA public key represents the calculation of keccak-256sum hash, and the hexadecimal of the last 20 bytes of the calculation result is taken as the account address. Each byte requires two hexadecimal to represent, so the length of account address is 40. FISCO BCOS's account address is compatible with Ethereum.

Note: keccak-256sum is different from `SHA3`. For details to refer to here.

Ethernet Address Generation

1. generate ECDSA private key

First, we use OpenSSL to generate an elliptic curve private key. The parameters of the elliptic curve is secp256k1. To run the following command to generate a private key in PEM format and save it in the ecprivkey.pem file.

```
openssl ecparam -name secp256k1 -genkey -noout -out ecprivkey.pem
```

Execute the following instructions to view the contents of the file.

```
cat ecprivkey.pem
```

You can see output similar to the following

```
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEINHaCmLhw9S9+vD0IOSUd9IhHO9bBVJXTbbBeTyFNvesoAcGBSuBBAAK
oUQDQgAEjSUbQAZn4tzHnsbeahQ2J0AeMu0iNOxpdpyPo3j9Diq3qdljrv07wvjx
zOzLpUNRcJCC5hnU500MD+4+Zxc8zQ==
-----END EC PRIVATE KEY-----
```

Next, to calculate the public key based on the private key. To execute the following command.

```
openssl ec -in ecprivkey.pem -text -noout 2>/dev/null| sed -n '7,11p' | tr -d ": \n
↪" | awk '{print substr($0,3);}'
```

You can get output similar to the following

```
8d251b400667e2dcc79ec6de6a143627401e32ed2234ec69769c8fa378fd0e2ab7a9d963aefd3bc2f8f1cceccba543517
```

2. Calculate the address based on the public key

In this section we calculate the corresponding account address based on public key. The keccak-256sum tool we need to get is available for download from here.

```
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
```

Get the output similar to the following, which is the calculated account address.

```
dcc703c0e500b653ca82273b7bfad8045d85a470
```

CHAPTER 6

Manual

This chapter provides an operation tutorial of FISCO BCOS platform to introduce its functions and operation methods.

6.1 Getting Executables

Users can choose any of the following methods to get FISCO BCOS executable. It is recommended to download the precompiled binaries from GitHub.

- The official statically linked precompiled files can be used on Ubuntu 16.04 and CentOS 7.2 version or later.
- docker image is provided officially, welcome to use. docker-hub address
- You can compile from the source code, visit here source code compilation.

6.1.1 Downloading precompiled fisco-bcos

The statically linked precompiled executable provided has been tested on Ubuntu 16.04 and CentOS 7. Please download the latest released **pre-compiled executable** from the Release.

6.1.2 docker image

From v2.0.0 version, we provide the docker image for the tag version. Corresponding to the master branch, we provide image of `latest` tag. For more docker tags please refer tohere.

build_chain.sh script adds the `-d` option to provide docker mode building for developers to deploy. For details, please refer to here.

Note: For using build_chain.sh script easily, we start docker by using `--network=host` network mode. Users may need to customize and modify according to their own network scenarios when they actually use.

6.1.3 Source code compilation

Note: The source code compilation is suitable for the experienced developers. You are required to download all library dependencies during compilation. Network connection would required and would take 5-20 minutes in total.

FISCO-BCOS is using generic CMake to generate platform-specific build files, which means the steps are similar for most operating systems:

1. Install build tools and dependent package (depends on platform).
2. Clone code from FISCO BCOS.
3. Run `cmake` to generate the build file and compile.

Installation dependencies

- Ubuntu

Ubuntu 16.04 or later is recommended. The versions below 16.04 have not been tested. You will require to have the build tools build tools and `libssl` for compiling the source code.

```
$ sudo apt install -y g++ libssl-dev openssl cmake git build-essential autoconf␣
↪texinfo
```

- CentOS

CentOS7 version or later is recommended.

```
$ sudo yum install -y epel-release
$ sudo yum install -y openssl-devel openssl cmake3 gcc-c++ git
```

- macOS

xcode10 version and above are recommended. macOS dependent package installation depends on Homebrew.

```
$ brew install openssl git
```

Code clone

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

Compile

After compilation, binary files are located at `FISCO-BCOS/build/bin/fisco-bcos`.

```
$ cd FISCO-BCOS
$ git checkout master
$ mkdir -p build && cd build
# please use cmake3 for CentOS
$ cmake ..
#To add -j4 to accelerate compilation by 4 compilation processes
$ make
```

Compile options

- BUILD_GM, off by default, national cryptography compilation flag. To enable it, use `cmake -DBUILD_GM=on ..`
- TESTS, off by default, unit test compilation flag. To enable it, use `cmake -DTESTS=on ..`
- DEMO, off by default, test program compilation switch. To open it through `cmake -DDEMO=on ...`
- BUILD_STATIC, off by default, static compilation switch, only supports Ubuntu. To open it through `cmake -DBUILD_STATIC=on ...`
- Generate source documentation.

```
# Install Doxygen
$ sudo apt install -y doxygen graphviz
# Generate source documentation locate at build/doc
$ make doc
```

6.2 Chain building script

Important: The goal of the script is to let users apply FISCO BCOS as quickly as possible. For the enterprise applications deploying FISCO BCOS, please refer to Enterprise Deployment Tools .

FISCO BCOS has provided `build_chain` script to help users quickly build FISCO BCOS alliance chain. By default, the script downloads `master` branch of the latest version pre-compiles executable program from GitHubfor building related environment.

6.2.1 Script introduction

- `build_chain` script is used to quickly generate configuration files of a chain node. For the script that depends on `openssl`, please according your own operating system to install `openssl 1.0.2` version and above. The source code of script is located at FISCO-BCOS/tools/build_chain.sh.
- For quick experience can use the `-l` option to specify the node IP and number. `-f` option supports the creation of FISCO BCOS chains for complex business scenarios by using a configuration file in a specified format. **`-l` and `-f` options must be specified uniquely and cannot coexist**.
- It is recommended to use `-T` and `-i` options for testing. `-T` enables log level to DEBUG. `-i` sets RPC and channel to listen for `0.0.0.0` while p2p module listens for `0.0.0.0` by default.

6.2.2 Help

```
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 specified 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
e.g
    ./tools/build_chain.sh -l "127.0.0.1:4"
```

6.2.3 Option introduction

- **`l`option:** Use to specify the chain to be generated and the number of nodes under each IP, separated by commas. The script generates configuration file of corresponding node according to the input parameters. The port number of each node is incremented from 30300 by default. All nodes belong to the same organization and Group.
- **`f`option**
  - Use to generate node according to configuration file. It supports more customization than `l` option.
  - Split by row. Each row represents a server, in the format of `IP:NUM AgencyName GroupList`. Items in each line are separated by spaces, and there must be **no blank lines**.
  - `IP:NUM` represents the IP address of the machine and the number of nodes on the machine.`AgencyName`represents the name of the institution to specifies the institution certificate to use. `GroupList` represents the group that the generated node belong to, split by`,`. For example, `192.168.0.1:2 agency1 1,2` represents that a machine with `ip` which is `192.168.0.1` exists two nodes. For example, 192.168.0.1:2 agency1 1,2 represents that there are two nodes on the machine with ip 192.168.0.1. These two nodes belong to agency `agency1` and belong to group1 and group2.

The following is an example of a configuration file. Each configuration item separated by a space, where `GroupList` represents the group that the server belongs to.

```
192.168.0.1:2 agency1 1,2
192.168.0.1:2 agency1 1,3
192.168.0.2:3 agency2 1
192.168.0.3:5 agency3 2,3
192.168.0.4:2 agency2 3
```

Suppose the above file is named `ipconf`, using the following command to build a chain, which indicates to use configuration file, to set the log level to `DEBUG`, and to listen for `0.0.0.0`.

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

- **`e`option[Optional]** is used to specify **full path** where `fisco-bcos` binary is located.Script will cope `fisco-bcos` to the directory named by IP number. If no path to be specified, the latest binary program of `master` branch is downloaded from GitHub by default.

```
# download the latest release binary from GitHub to generate native 4 nodes
$ bash build_chain.sh -l "127.0.0.1:4"
# use bin/fisco-bcos binary to generate native 4 nodes
$ bash build_chain.sh -l "127.0.0.1:4" -e bin/fisco-bcos
```

- **ooption[Optional]** specifies the directory where the generated configuration is located.
- **poption[Optional]** specifies the starting port of the node. Each node occupies three ports which are p2p, channel, and jsonrpc, respectively. The ports are split by, and three ports must be specified. The ports used by different nodes under the same IP address are incremented from the starting port.

```
# Two nodes occupies `30300,20200,8545` and `30301,20201,8546` respectively.
$ bash build_chain -l 127.0.0.1:2 -p 30300,20200,8545
```

- **ioption[Optional]** No parameter option. When setting this option, set the node's RPC and channel to listen to `0.0.0.0`.
- **voption[Optional]**

Used to specify the binary version used when building FISCO BCOS. build_chain downloads the latest version of [Release Page] (https://github.com/FISCO-BCOS/FISCO-BCOS/releases) by default. When setting this option, the download parameter specifies the `version` version and sets `[compatibility].supported_version=${version}` in the configuration file `config.ini`. If you specify the binary with the `-e` option, to use the binary and configure `[compatibility].supported_version=${version}` as the latest version number of [Release page](https://github.com/FISCO-BCOS/FISCO-BCOS /releases).

- **doption[Optional]**

Use the docker mode to build FISCO BCOS. When using this option, the binary is no longer extracted, but users are required to start the node machine to install docker, and their accounts have docker permission. The command to start the node in this mode is as follows

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

- **moption[Optional]** No parameter option. When setting this option, node uses mptstate to store contract local variables. By default, storagestate is used to store the contract local variable.
- **soption[Optional]** There are parameter options. The parameter is the name of db. Currently it supports three modes: rocksdb, mysql, and external. RocksDB is used by default. mysql needs to configure the information relates to mysql in the group ini file. external needs to configure topic information and start AMDB proxy.
- **coption[Optional]** No parameter option. When setting this option, the consensus algorithm for setting the node is Raft, and the default setting is PBFT.
- **Coption[Optional]**

Used to specify the chain identifier when building FISCO BCOS. When this option is set, using parameter to set `[chain].id` in the configuration file `config.ini`. The parameter range is a positive integer and the default setting is 1.

```
# The chain is identified as 2
$ bash build_chain.sh -l 127.0.0.1:2 -C 2
```

- **goption[Optional]** No parameter option. When setting this option, to build the national cryptography version of FISCO BCOS. **The binary fisoc-bcos is required to be national cryptography version when using the g option.**
- **zoption[Optional]** No parameter option. When setting this option, the tar package of node is generated.
- **toption[Optional]** This option is used to specify the certificate configuration file when certificate is generated.
- **Toption[Optional]**

No parameter option. When setting this option, set the log level of node to DEBUG. The related configuration of log reference here.

6.2.4 Node file organization

- cert folder stores root certificate and organization certificate of the chain.
- The folder named by IP address stores the certificate configuration file required by related configuration of all nodes , `fisco-bcos` executable program, and SDK in the server.
- The `node*` folder under each IP folder stores configuration file required by the node. `config.ini` is the main configuration of node. In `conf` directory, to store certificate files and group related configurations. For the configuration detail, please refer to here. Each node provides two scripts which are used to start and stop the node.
- Under each IP folder, two scripts providing `start_all.sh` and `stop_all.sh` are used to start and stop all nodes.

```
nodes/
├── 127.0.0.1
│   ├── fisco-bcos # binary program
│   ├── node0 # node0 folder
│   │   ├── conf # configuration folder
│   │   │   ├── ca.crt # chain root certificate
│   │   │   ├── group.1.genesis # the initialized configuration of group1, the␣
↪file cannot be changed
│   │   │   ├── group.1.ini # the configuration file of group1
│   │   │   ├── node.crt # node certificate
│   │   │   ├── node.key # node private key
│   │   │   └── node.nodeid # node id, represented by hexadecimal of public key
│   │   ├── config.ini # node main configuration file, to configure listening IP,␣
↪port, etc.
│   │   ├── start.sh # start script, uses for starting node
│   │   └── stop.sh # stop script, uses for stopping node
│   ├── node1 # node1 folder
│   .....
│   ├── node2 # node2 folder
│   .....
│   ├── node3 # node3 folder
│   .....
│   ├── sdk # SDK needs to be used
│   │   ├── ca.crt # chain root certificate
│   │   ├── node.crt # The certificate file required by SKD, to use when␣
↪establishing a connection
│   │   └── node.key # The private key file required by SKD, to use when␣
↪establishing a connection
├── cert # certificate folder
│   ├── agency # agency certificate folder
│   │   ├── agency.crt # agency certificate
│   │   ├── agency.key # agency private key
│   │   ├── agency.srl
│   │   ├── ca-agency.crt
│   │   ├── ca.crt
│   │   └── cert.cnf
│   ├── ca.crt # chain certificate
│   ├── ca.key # chain private key
│   ├── ca.srl
│   └── cert.cnf
```

6.2.5 Example

Single-server and single-group

To build a 4-node FISCO BCOS alliance chain on native machine for using the default start port `30300,20200,8545` (4 nodes will occupy `30300-30303,20200-20203,8545-8548`) and listening to the external network

`Channel` and `jsonrpc` ports while allowing the external network interacts with node through SDK or API.

```
# to build FISCO BCOS alliance chain
$ bash build_chain.sh -l "127.0.0.1:4" -i
# after generating successes, to output `All completed` to mention
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        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:4
[INFO] State Type        : storage
[INFO] RPC listen IP     : 0.0.0.0
[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
```

Multi-server and multi-group

Using the build_chain script to build a multi-server and multi-group FISCO BCOS alliance chain requires the script configuration file. For details, please refer to here.

6.3 Certificate description

FISCO-BCOS network adopts a CA-oriented access mechanism to support any multi-level certificate structure for ensuring information confidentiality, authentication, integrity, and non-repudiation.

FISCO BCOS uses the x509 protocol certificate format. According to the existing business scenario, a three level certificate structure is adopted by default, and from top to bottom, the three levels are chain certificate, agency certificate, and node certificate respective.

In multi-group architecture, a chain has a chain certificate and a corresponding chain private key, and the chain private key is jointly managed by alliance chain committee. Alliance chain committee can use the agency's certificate request file `agency.csr` to issue the agency certificate `agency.crt`.

Agency private key held by the agency administrator can issue node certificate to the agency's subordinate nodes.

Node certificate is the credential of node identity and uses this certificate to establish an SSL connection with other nodes for encrypted communication.

sdk certificate is a voucher for sdk communicating with node. Agency generates sdk certificate that allows sdk to do that.

The files' suffixes of FISCO BCOS node running are described as follows:

6.3.1 Role definition

There are four roles in the FISCO-BCOS certificate structure, namely the alliance chain committee administrator, agency, node, and SDK.

Alliance chain committee

- The alliance chain committee manages private key of chain, and issues agency certificate according to agency's certificate request document `agency.csr`.

```
ca.crt chain certificate
ca.key chain private key
```

When FISCO BCOS performs SSL encrypted communication, only the node with the same chain certificate `ca.crt` can establish a connection.

Agency

- Agency has an agency private key that can issue node certificate and SDK certificate.

```
ca.crt chain certificate
agency.crt agency certificate
agency.csr agency certificate request file
agency.key agency private key
```

Node/SDK

- FISCO BCOS nodes include node certificates and private keys for establishing SSL encrypted connection among nodes;
- SDK includes SDK certificate and private key for establishing SSL encrypted connection with blockchain nodes.

```
ca.crt chain certificate
node.crt node/SDK certificate
node.key node/SDK private key
```

Node certificate `node.crt` includes the node certificate and the agency certificate information. When the node communicates with other nodes/SDKs, it will sign the message with its own private key `node.key`, and send its own `node.crt` to nodes/SDKs to verify.

6.3.2 Certificate generation process

FISCO BCOS certificate generation process is as follows. Users can also use the Enterprise Deployment Tool to generate corresponding certificate

Chain certificate generation

- Alliance chain committee uses openssl command to request chain private key `ca.key`, and generates chain certificate `ca.crt` according to ca.key.

Agency certificate generation

- Agency uses openssl command to generate agency private key `agency.key`
- Agency uses private key `agency.key` to get agency certificate request file `agency.csr`, and sends `agency.csr` to alliance chain committee.
- Alliance chain committee uses chain private key `ca.key` to generate the agency certificate `agency.crt` according to the agency certificate request file `agency.csr`. And send agency certificate `agency.crt` to corresponding agency.

Node/SDK certificate generation

- The node generates the private key `node.key` and the certificate request file `node.csr`. The agency administrator uses the private key `agency.key` and the certificate request file `node.csr` to issue the certificate `node.crt` to the node/SDK.

6.4 Configuration files and configuration items

FISCO BCOS supports multiple ledger. Each chain includes multiple unique ledgers, whose data among them are isolated from each other. And the transaction processing among groups are also isolated. Each node includes a main configuration `config.ini` and multiple ledger configurations `group.group_id.genesis`, `group.group_id.ini`.

- `config.ini`: The main configuration file, mainly configures with RPC, P2P, SSL certificate, ledger configuration file path, compatibility and other information.
- `group.group_id.genesis`: group configurations file. All nodes in the group are consistent. After node launches, you cannot manually change the configuration including items like group consensus algorithm, storage type, and maximum gas limit, etc.
- `group.group_id.ini`: group variable configuration file, including the transaction pool size, etc.. All configuration changes are effective after node restarts.

6.4.1 Hardware requirements

Note: Since multiple nodes share network bandwidth, CPU, and memory resources, it is not recommended to configure too much nodes on one machine in order to ensure the stability of service.

The following table is a recommended configuration for single-group and single-node. Node consumes resources in a linear relationship with the number of groups. You can configure the number of nodes reasonably according to actual business requirement and machine resource.

6.4.2 Main configuration file config.ini

`config.ini` uses `ini` format. It mainly includes the configuration items like ** rpc, p2p, group, secure and log **.

Important:

- The public IP addresses of the cloud host are virtual IP addresses. If listen_ip is filled in external network IP address, the binding fails. You must fill in 0.0.0.0.
- RPC/P2P/Channel listening port must be in the range of 1024-65535 and cannot conflict with other application listening ports on the machine.

Configure RPC

- `listen_ip`: For security reasons, the chain building script will listen to 127.0.0.1 by default. If you need to access the RPC or use SDK through external network, please listen to **external network IP address of node** or `0.0.0.0`;
- `channel_listen_port`: Channel port, is corresponding to `channel_listen_port` in Web3SDK configuration;

- jsonrpc_listen_port: JSON-RPC port.

RPC configuration example is as follows:

```
[rpc]
    listen_ip=127.0.0.1
    channel_listen_port=30301
    jsonrpc_listen_port=30302
```

Configure P2P

The current version of FISCO BCOS must be configured with IP and Port of the connection node in the config.ini configuration. The P2P related configurations include:

-listen_ip: P2P listens for IP, to set 0.0.0.0 by default. -listen_port: Node P2P listening port. -node.*: All nodes IP:port which need to be connected to node.

- enable_compress: Enable network compression configuration option. Configuring to true, indicates that network compression is enabled. Configuring to false, indicates that network compression is disabled. For details on network compression, please refer to [here](../design/features/network_compress .md).

P2P configuration example is as follows:

```
[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
```

Configure ledger file path

[group]To configure all group configuration paths which this node belongs:

- group_data_path: Group data storage path.
- group_config_path: Group configuration file path.

  Node launches group according to all .genesis suffix files in the group_config_path path.

```
[group]
    ; All group data is placed in the node's data subdirectory
    group_data_path=data/
    ; Program automatically loads all .genesis files in the path
    group_config_path=conf/
```

Configure certificate information

For security reasons, communication among FISCO BCOS nodes uses SSL encrypted communication.[network_security] configure to SSL connection certificate information:

- data_path: Directory where the certificate and private key file are located.
- key: The data_path path that node private key relative to.
- cert: The data_path path that certificate node.crt relative to.
- ca_cert: ca certificate file path.
- ca_path: ca certificate folder, required for multiple ca.

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

Configure blacklist

For preventing vice, FISCO BCOS allows nodes to configure untrusted node blacklist to reject establishing connections with these blacklist nodes. To configure blacklist through `[crl]`:

> `crl.idx`: Blacklist node's Node ID, can get from `node.nodeid` file; `idx` is index of the blacklist node.

For details of the blacklist, refer to [CA Blacklist].(./certificate_blacklist.md)

Blacklist configuration example is as follows:

```
; certificate blacklist
[crl]
    crl.
↪0=4d9752efbb1de1253d1d463a934d34230398e787b3112805728525ed5b9d2ba29e4ad92c6fcde5156ede8baa5aca3
3787c338a4
```

Configure log information

FISCO BCOS supports boostlog. Configurations:

- `enable`: Enable/disable log. Set to `true` to enable log; set to `false` to disable log. **set to true by default. For performance test, to set this option to `false` to reduce the impact of print log on test results**
- `log_path`:log file patch.
- `level`: log level, currently includes 5 levels which are `trace`、`debug`、`info`、`warning`、`error`.After setting a certain log level, the log file will be entered with a log equal to or larger than this level.The log level is sorted from large to small by `error > warning > info > debug > trace`.
- `max_log_file_size`: Maximum size per log file, ** unit of measure is bytes, default is 200MB**
- `flush`: boostlog enables log auto-refresh by default. To improves system performance, it is recommended to set this value to false.

boostlog configuration example is as follows:

```
[log]
    ; whether to enable log, set to true by default
    enable=true
    log_path=./log
    level=info
    ; Maximum size per log file, default is 200MB
    max_log_file_size=200
    flush=true
```

Configure node compatibility

All versions of FISCO-BCOS 2.0 are forward compatible. You can configure the compatibility of node through `[compatibility]` in `config.ini`. The tool will be automatically generated when changing the configuration item to build chain, so users do not need to change it.

- `supported_version`: The version of the current node running

Important:

- view the latest version of FISCO BCOS currently supports through the command `./fisco-bcos --version | grep "FISCO-BCOS Version" | cut -d':' -f2 | sed s/[[:space:]]//g`
- In the blockchain node configuration generated by build_chain.sh, supported_version is configured to the current latest version of FISCO BCOS
- When upgrading an old node to a new node, directly replace the old FISCO BCOS binary with the latest FISCO BCOS binary.

`release-2.0.0` node's `[compatibility]` configuration is as follows:

```
[compatibility]
    supported_version=release-2.0.0
```

Optional configuration: Disk encryption

In order to protect node data, FISCO BCOS introduces Disk Encryption to ensure confidentiality. **Disk Encryption** Operation Manual Reference.

`storage_security` in `config.ini` is used to configure disk encryption. It mainly includes (for the operation of the disk encryption, please refer to Operation Manual):

- `enable`: whether to launch disk encryption, not to launch by default;
- `key_manager_ip`: Key Managerservice's deployment IP;
- `key_manager_port`: Key Managerservice's listening port;
- `cipher_data_key`: ciphertext of node data encryption key. For `cipher_data_key` generation, refer to disk encryption operation manual.

disk encryption configuration example is as follows:

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

6.4.3 Group system configuration instruction

Each group has unique separate configuration file, which can be divided into **group system configuration** and **group variable configuration** according to whether it can be changed after launch. group system configuration is generally located in the `.genesis` suffix configuration file in node's `conf` directory.

For example:`group1` system configuration generally names as `group.1.genesis`. Group system configuration mainly includes the related configuration of **group ID**、**consensus, storage and gas**.

Important: When configuring the system configuration, you need to pay attention to:

- **configuration group must be consistent**: group system configuration is used to generate the genesis block (block 0), so the configurations of all nodes in the group must be consistent.
- **node cannot be modified after launching**： system configuration has been written to the system table as genesis block, so it cannot be modified after chain initializes.

- After chain is initialized, even if genesis configuration is modified, new configuration will not take effect, and system still uses the genesis configuration when initializing the chain.
- Since genesis configuration requires all nodes in the group to be consistent, it is recommended to use build_chain to generate the configuration.

Group configuration

`[group]` configures **group ID**. Node initializes the group according to the group ID.

group2's configuration example is as follows:

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

Consensus configuration

`[consensus]` involves consensus-related configuration, including:

- `consensus_type`: consensus algorithm type, currently supports PBFT and Raft. To use PBFT by default;
- `max_trans_num`: a maximum number of transactions that can be packed in a block. The default is 1000. After the chain is initialized, the parameter can be dynamically adjusted through Console;
- `node.idx`: consensus node list, has configured with the [Node ID] of the participating consensus nodes. The Node ID can be obtained by the `${data_path}/node.nodeid` file (where `${data_path}` can be obtained by the configuration item `[secure].data_path` of the main configuration `config.ini`)

```
; Consensus protocol configuration
[consensus]
    ;consensus algorithm, currently supports PBFT(consensus_type=pbft) and␣
↪Raft(consensus_type=raft)
    consensus_type=pbft
    ; maximum number of transactions in a block
    max_trans_num=1000
    ; leader node's ID lists
    node.
↪0=123d24a998b54b31f7602972b83d899b5176add03369395e53a5f60c303acb719ec0718ef1ed51feb7e9cf4836f26
e01789233a
    node.
↪1=70ee8e4bf85eccda9529a8daf5689410ff771ec72fc4322c431d67689efbd6fbd474cb7dc7435f63fa592b98f22b1
3494db8776
    node.
↪2=7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b
922aa0ef50
    node.
↪3=fd6e0bfe509078e273c0b3e23639374f0552b512c2bea1b2d3743012b7fed8a9dec7b47c57090fa6dcc5341922c32
4a5aed2b4a
```

State mode configuration

state is used to store blockchain status information. It locates in the genesis file `[state]`:

- `type`: state type, currently supports storage state and MPT state, **defaults to Storage state**. storage state storing the transaction execution result in the system table, which is more efficient. MPT state storing the transaction execution result in the MPT tree, which is inefficient but contains complete historical information.

Important: **storage state** is recommended. MPT State is not recommended except for special requirements.

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

Gas configuration

FISCO BCOS is compatible with Ethereum virtual machine (EVM). In order to prevent DOS from attacking EVM, EVM introduces the concept of gas when executing transactions, which is used to measure the computing and storage resources consumed during the execution of smart contracts. The measure includes the maximum gas limit of transaction and block. If the gas consumed by the transaction or block execution exceeds the gas limit, the transaction or block is discarded.

FISCO BCOS is alliance chain that simplifies gas design. **It retains only maximum gas limit of transaction, and maximum gas of block is constrained together by consensus configuration max_trans_num and transaction maximum gas limit.**

FISCO BCOS configures maximum gas limit of the transaction through genesis `[tx].gas_limit`. The default value is 300000000. After chain is initialized, the gas limit can be dynamically adjusted through the console command.

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

6.4.4 Ledger variable configuration instruction

Variable configuration of the ledger is located in the file of the `.ini` suffix in the node `conf` directory.

For example: `group1` variable configuration is generally named `group.1.ini`. Variable configuration mainly includes transaction pool size, PBFT consensus message forwarding TTL, PBFT consensus packing time setting, PBFT transaction packaging dynamic adjustment setting, parallel transaction settings, etc..

Configure storage

Storage currently supports three modes: RocksDB, MySQL, and External. Users can choose the DB to use according to their needs. RocksDB has the highest performance. MySQL supports users to use MySQL database for viewing data. External accesses mysql through data proxy, and users need to start and configure the data proxy. The design documentation can be referenced AMDB Storage Design. Since the RC3 version, we have used RocksDB instead of LevelDB for better performance, but still supports LevelDB.

Public configuration item

- `type`: The stored DB type, which supports `RocksDB`, `MySQL` and `External`. When the DB type is RocksDB, all the data of blockchain system is stored in the RocksDB local database; when the type is `MySQL` or `External`, the node accesses mysql database according to the configuration. All data of blockchain system is stored in mysql database. For accessing mysql database, to configure the AMDB proxy. Please refer to here for the AMDB proxy configuration.
- `max_capacity`: configures the space size of the node that is allowed to use for memory caching.
- `max_forward_block`: configures the space size of the node that allowed to use for memory block. When the blocks exceeds this value, the node stops the consensus and waits for the blocks to be written to database.

Database related configuration item

- `topic`: When the type is `External`, you need to configure this field to indicate the AMDB proxy topic that blockchain system is interested in. For details, please refer to here.
- `max_retry`: When the type is `External`, you need to configure this field to indicate the number of retries when writing fails. For details, please refer to here.
- `db_ip`: When the type is `MySQL`, you need to configure this field to indicate the IP address of MySQL.
- `db_port`: When the type is `MySQL`, you need to configure this field to indicate the port number of MySQL.
- `db_username`: When the type is `MySQL`, you need to configure this field to indicate the MySQL username.
- `db_passwd`: When the type is `MySQL`, you need to configure this field to indicate the password corresponding to the MySQL user.
- `db_name`: When the type is `MySQL`, you need to configure this field to indicate the database name used in MySQL.
- `init_connections`: When the type is `MySQL`, this field can be optionally configured to indicate the initial number of connections established with MySQL. The default value is 15, and it is fine to use it.
- `max_connections`: When the type is `MySQL`, this field can be optionally configured to indicate the maximum number of connections established with MySQL. The default value is 20, and it is fine to use it.

The following is an example of the configuration of [storage]:

```
[storage]
    ; storage db type, rocksdb / mysql / external, 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=
```

Transaction pool configuration

FISCO BCOS opens the transaction pool capacity configuration to users. Users can dynamically adjust the transaction pool size according to their business size requirements, stability requirements, and node hardware configuration.

Transaction pool configuration example is as follows:

```
[tx_pool]
    limit=150000
```

PBFT consensus message broadcast configuration

In order to ensure the maximum network fault tolerance of the consensus process, each consensus node broadcasts the message to other nodes after receiving a valid consensus message. In smooth network environment, the consensus message forwarding mechanism will waste additional network bandwidth, so the `ttl` is introduced in

the group variable configuration item to control the maximum number of message forwarding. The maximum number of message forwarding is `ttl-1`, and **the configuration item is valid only for PBFT**.

Setting consensus message to be forwarded at most once configuration example is as follows:

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

PBFT consensus packing time configuration

The PBFT module packing too fast causes only 1 to 2 transactions to be pack in some blocks. For avoiding wasting storage space, FISCO BCOS v2.0.0-rc2 introduces `min_block_generation_time` configuration item in the group variable configuration `group.group_id.ini`'s `[consensus]` to manager the minimum time for PBFT consensus packing. That is, when the consensus node packing time exceeds `min_block_generation_time` and the number of packaged transactions is greater than 0, the consensus process will start and handle the new block generated by the package.

Important:

- `min_block_generation_time` is 500ms by default
- The longest packing time of consensus node is 1000ms. If the time is exceeded 1000ms and the number of transactions packed in the new block is still 0, the consensus module will enter the logic of empty block generation, and the empty block will not be written to disk;
- `min_block_generation_time` cannot exceed the time of empty block generation which is 1000ms. If the set value exceeds 1000ms, the system defaults min_block_generation_time to be 500ms.

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

PBFT transaction package dynamic adjustment

For the impact causing by CPU loading and network latency on system processing power, PBFT provides an algorithm that dynamically adjusts the maximum number of transactions that can be packed in a block. The algorithm dynamically can adjust the maximum number of transactions according to the state of historical transaction processing. The algorithm is turned on by default, and it can be turned off by changing the `[consensus].enable_dynamic_block_size` configuration item of the variable configuration `group.group_id.ini` to `false`. At this time, the maximum number of transactions in the block is the `[consensus].max_trans_num` of `group.group_id.genesis`.

The configuration of closing the dynamic adjustment algorithm for the block package transaction number is as follows:

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

Parallel transaction configuration

FISCO BCOS supports execution of transactions in parallel. Turning on the transaction parallel execution switch to enable for improving throughput. **Execution of the transaction in parallel is only effective in the storage state mode**.

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

6.4.5 Dynamically configure system parameters

FISCO BCOS system currently includes the following system parameters (other system parameters will be extended in the future):

Console provides **setSystemConfigByKey** command to modify these system parameters. **getSystemConfigByKey** command can view the current value of the system parameter:

Important: It is not recommended to modify tx_count_limit and tx_gas_limit arbitrarily. These parameters can be modified as follows:

- Hardware performance such as machine network or CPU is limited: to reduce tx_count_limit for reducing business pressure;
- gas is insufficient when executing blocks for comlicated business logic: increase tx_gas_limit.

```
# To set the maximum number of transactions of a packaged block to 500

> setSystemConfigByKey tx_count_limit 500
# inquiry tx_count_limit
> getSystemConfigByKey tx_count_limit
[500]

# To set block gas limit as  400000000
> getSystemConfigByKey tx_gas_limit 400000000
> getSystemConfigByKey
[400000000]
```

6.5 Console

Console is an important interactive client tool of FISCO BCOS 2.0. It establishes a connection with blockchain node through Web3SDK to request read and write access for blockchain node data. Console has a wealth of commands, including blockchain status inquiry, blockchain nodes management, contracts deployment and calling. In addition, console provides a contract compilation tool that allows users to easily and quickly compile Solidity contract files into Java contract files.

6.5.1 Console command

Console command consists of two parts, the instructions and the parameters related to the instruction:

- **Instruction**: instruction is an executed operation command, including blockchain status inquiry and contracts deployment and calling. And some of the instructions call the JSON-RPC interface, so they have same name as the JSON-RPC interface. **Use suggestions: instructions can be completed using the tab key, and support for displaying historical input commands by pressing the up and down keys.**
- **Parameters related to the instruction**: parameters required by instruction call interface. Instructions to parameters and parameters to parameters are separated by spaces. The parameters name same as JSON-RPC interface and the explanation of getting information field can be referred to JSON-RPC API.

6.5.2 Common command link:

Contract related commands

- useCNS to deploy and call contract (**recommend**)

  - deploy contract: deployByCNS
  - call contract: callByCNS
  - query CNS deployment contract information: queryCNS
- deploy and call contract normally
  - deploy contract: deploy
  - call contract: call

Other commands

- query block number:getBlockNumber
- query Sealer list:getSealerList
- query the information of transaction receipt: getTransactionReceipt
- switch group: switch

6.5.3 Shortcut key

- `Ctrl+A`: move cursor to the beginning of line
- `Ctrl+D`: exit console
- `Ctrl+E`: move cursor to the end of line
- `Ctrl+R`: search for the history commands have been entered
- ↑: browse history commands forward
- ↓: browse history commands backward

6.5.4 Console response

When a console command is launched, the console will obtain the result of the command execution and displays the result at the terminal. The execution result is divided into two categories:

- **True:** The command returns to the true execution result as a string or json.
- **False:** The command returns to the false execution result as a string or json.
  - When console command call the JSON-RPC interface, error code reference here.
  - When console command call the Precompiled Service interface, error code reference here.

6.5.5 Console configuration and operation

Important: Precondition： to build FISCO BCOS blockchain, please refer to Building Chain Script or Enterprise Tools.

Get console

```
$ cd ~ && mkdir fisco && cd fisco
# get console
$ bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪download_console.sh)
```

The directory structure is as follows:

```
|-- apps # console jar package directory
|   -- console.jar
|-- lib # related dependent jar package directory
|-- conf
|   |-- applicationContext-sample.xml # configuration file
|   |-- log4j.properties  # log configuration file
|-- contracts # directory where contract locates
|   -- solidity  # directory where solidity contract locates
|       -- HelloWorld.sol # normal contract: HelloWorld contract, is deployable␣
↪and callable
|       -- TableTest.sol # the contracts by using CRUD interface: TableTest␣
↪contract, is deployable and callable
|       -- Table.sol # Table contract interface required to be introduced by CRUD␣
↪contract
|   -- console  # The file directory of contract abi, bin, java compiled when␣
↪console deploys the contract
|   -- sdk      # The file directory of contract abi, bin, java compiled by␣
↪sol2java.sh script
|-- start.sh # console start script
|-- get_account.sh # account generate script
|-- sol2java.sh # development tool script for compiling solidity contract file as␣
↪java contract file
|-- replace_solc_jar.sh # a script for replacing the compiling jar package
```

Contract compilation tool

Console provides a special compilation contract tool that allows developers to compile Solidity contract files into Java contract files. Two steps for using the tool:

- To place the Solidity contract file in the `contracts/solidity` directory.
- Complete the task of compiling contract by running the `sol2java.sh` script (**requires specifying a java package name**). For example, there are `HelloWorld.sol`, `TableTest.sol`, and `Table.sol` contracts in the `contracts/solidity` directory, and we specify the package name as `org.com.fisco`. The command is as follows:

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

After running successfully, the directories of Java, ABI and bin will be generated in the `console/contracts/sdk` directory as shown below.

```
```bash
|-- abi # to compile the generated abi directory and to store the abi file␣
↪compiled by solidity contract
|   |-- HelloWorld.abi
|   |-- Table.abi
|   |-- TableTest.abi
|-- bin # to compile the generated bin directory and to store the bin file␣
↪compiled by solidity contract
|   |-- HelloWorld.bin
|   |-- Table.bin
|   |-- TableTest.bin
|-- java  # to store compiled package path and Java contract file
|   |-- org
|       |-- com
|           |-- fisco
|               |-- HelloWorld.java # the target Java file which is compiled␣
↪successfully
|               |-- Table.java  # the system CRUD contract interface Java file␣
↪which is compiled successfully
```
```

```
|               |-- TableTest.java  # the TableTest Java file which is compiled␣
↪successfully
```

In the java directory, `org/com/fisco/` package path directory is generated. In the package path directory, the java contract files `HelloWorld.java`, `TableTest.java` and `Table.java` will be generated. `HelloWorld.java` and `TableTest.java` are the java contract files required by the java application.

**Note: ** The downloaded console contains `solcJ-all-0.4.25.jar` in the `console/lib` directory, so it supports the 0.4 version of the contract compilation. If you are using a 0.5 version contract compiler or a national cryptography contract compiler, please download the relevant contract compiler jar package, and replace `solcJ-all-0.4.25.jar` in the `console/lib` directory. It can be replaced by the `./replace_solc_jar.sh` script. To specify the jar package path, the command is as follows:

```
# To download solcJ-all-0.5.2.jar and to place in console directory, the example␣
↪usage is as follows

$ ./replace_solc_jar.sh solcJ-all-0.5.2.jar
```

Download contract compilation jar package

0.4 version contract compilation jar package

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

0.5 version contract compilation jar package

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

National cryptography 0.4 version contract compilation jar package

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

National cryptography 0.5 version contract compilation jar package

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

Configure console

- Blockchain node and certificate configuration:
  - To copy the `ca.crt`, `node.crt`, and `node.key` files in the sdk node directory to the `conf` directory.
  - To rename the `applicationContext-sample.xml` file in the `conf` directory to the `applicationContext.xml` file. To configure the `applicationContext.xml` file, where the remark content is modified according to the blockchain node configuration. **Hint: If the listen_ip set through chain building is 127.0.0.1 or 0.0.0.0 and the channel_port is 20200, the `applicationContext.xml` configuration is not modified. **

```
<?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>  <!-- each group need to configure a bean -->
                            <bean id="group1"  class="org.fisco.bcos.channel.
↪handler.ChannelConnections">
                                   <property name="groupId" value="1" /> <!--␣
↪groupID -->
                                   <property name="connectionsStr">
                                          <list>
                                                 <value>127.0.0.1:20200</
↪value>  <!-- IP:channel_port -->
                                          </list>
                                   </property>
                            </bean>
                     </list>
              </property>
       </bean>

       <bean id="channelService" class="org.fisco.bcos.channel.client.Service"␣
↪depends-on="groupChannelConnectionsConfig">
              <property name="groupId" value="1" /> <!-- to connect to the group␣
↪with ID 1 -->
              <property name="orgID" value="fisco" />
              <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
       </bean>

</beans>
```

Configuration detail reference here.

Important: Console configuration instructions

- If the console is configured correctly, but when it is launched on Cent0S system, the following error occurs:

  Failed to connect to the node. Please check the node status and the console configuration.

  It is because the JDK version that comes with the CentOS system is used (it will cause the console and blockchain node's authentication to fail). Please download Java 8 version or above from OpenJDK official website or Oracle official website and install (specific installation steps refer to Appendix). To launch the console after installation.

- When the console configuration file configures multiple node connections in a group, some nodes in the group may leave the group during operation. Therefore, it shows a norm which is when the console is polling, the return information may be inconsistent. It is recommended to configure a node or ensure that the configured nodes are always in the group when using the console, so that the inquired information in the group will keep consistent during the synchronization time.

Launch console

Start the console while the node is running:

```
$ ./start.sh
# To output the following information to indicate successful launch
================================================================================================
Welcome to FISCO BCOS console(1.0.3)!
Type 'help' or 'h' for help. Type 'quit' or 'q' to quit console.
 ________ _______  _______   _______   _______       ________   _______   _______   _______
|        |       \/       \ /       \ /       \     |        \ /       \ /       \ /       \
↪\
| $$$$$$$$\$$$$$|  $$$$$$|  $$$$$$|  $$$$$$\     | $$$$$$$|  $$$$$$|  $$$$$$|  $$$$$
↪$\
| $$__      | $$ | $$___\$| $$   \$| $$  | $$     | $$__/ $| $$   \$| $$  | $| $$___\
↪$$
| $$  \     | $$  \$$    \| $$      | $$  | $$     | $$    $| $$      | $$  | $$\$$
↪\
| $$$$$     | $$  _\$$$$$$| $$   __| $$  | $$     | $$$$$$$| $$   __| $$  | $$_\$$$$$
↪$\
| $$       _| $$_|  \__| $| $$__/  | $$__/ $$     | $$__/ $| $$__/  | $$__/ $|  \__|
↪$$
| $$      |   $$ \\$$    $$\$$    $$\$$    $$     | $$    $$\$$    $$\$$    $$\$$
↪$$
 \$$       \$$$$$$ \$$$$$$  \$$$$$$  \$$$$$$       \$$$$$$$  \$$$$$$  \$$$$$$  \$$$$$
↪$

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

Launch script description

To view the current console version:

```
./start.sh --version
console version: 1.0.3
```

Account using method

Console loads private key

The console provides the account generation script get_account.sh (for the script tutorial, please refer to Account Management Document. The generated account file is in the accounts directory, and the account file loaded by console must be placed in the directory.

The console startup methods are as follows:

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

Default start

The console randomly generates an account that is started with the group number specified in the console configuration file.

```
./start.sh
```

Specify group number to start

The console randomly generates an account that is started with the group number specified on the command line.

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

- Note: The specified group needs to configure 'bean' in the console configuration file.

Start with PEM format private key file

- Start with the account of the specified pem file, enter the parameters: group number, -pem, and pem file path

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

Start with PKCS12 format private key file

- Start with the account of the specified p12 file, enter the parameters: group number, -p12, and p12 file path

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

6.5.6 Console command

help

Enter help or h to see all the commands on the console.

```
[group:1]> help
--------------------------------------------------------------------------------
↪--
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.
deploy                                   Deploy a contract on blockchain.
deployByCNS                              Deploy a contract on blockchain by CNS.
desc                                     Description table information.
exit                                     Quit console.
getBlockByHash                           Query information about a block by hash.
getBlockByNumber                         Query information about a block by block␣
↪number.
getBlockHashByNumber                     Query block hash by block number.
getBlockNumber                           Query the number of most recent block.
getCode                                  Query code at a given address.
getConsensusStatus                       Query consensus status.
getDeployLog                             Query the log of deployed contracts.
getGroupList                             Query group list.
getGroupPeers                            Query nodeId list for sealer and observer␣
↪nodes.
getNodeIDList                            Query nodeId list for all connected nodes.
```

```
getNodeVersion                          Query the current node version.
getObserverList                         Query nodeId list for observer nodes.
getPbftView                             Query the pbft view of node.
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.
getTransactionReceipt                   Query the receipt of a transaction 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.
grantPermissionManager                  Grant permission for permission␣
↪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.
listPermissionManager                   Query permission information for␣
↪permission 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.
revokePermissionManager                 Revoke permission for permission␣
↪configuration by address.
revokeSysConfigManager                  Revoke permission for system␣
↪configuration by address.
revokeUserTableManager                  Revoke permission for user table by table␣
↪name and address.
setSystemConfigByKey                    Set a system config.
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.
-------------------------------------------------------------------------------
↪--
```

**Note: **

- help shows the meaning of each command: command and command description
- for instructions on how to use specific commands, enter the command -h or –help to view them. E.g:

```
[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

To run switch or s to switch to the specified group. The group number is displayed in front of the command prompt.

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

[group:2]>
```

**Note: ** For the group that needs to be switched, make sure that the information of the group is configured in `applicationContext.xml` (the initial state of this configuration file only provides the group 1 configuration) in the `console/conf` directory, the configured node ID and port in the group are correct, and the node is running normally.

getBlockNumber

To run getBlockNumber to view block number.

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

getSealerList

To run getSealerList to view the list of consensus nodes.

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

getObserverList

To run getSealerList to view the list of observer nodes.

```
[group:1]> getObserverList
[
    "037c255c06161711b6234b8c0960a6979ef039374ccc8b723afea2107cba3432dbbc837a714b7da20111f74d5a24e91
]
```

getNodeIDList

To run getNodeIDList to view the nodes and the list of nodeIds connected to p2p nodes.

```
[group:1]> getNodeIDList
[
    "41285429582cbfe6eed501806391d2825894b3696f801e945176c7eb2379a1ecf03b36b027d72f480e89d15bacd43462
    "87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9ea
    "29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5d9
    "d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b75
]
```

getPbftView

To run getPbftView to view the pbft viewgraph.

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

getConsensusStatus

To run getConsensusStatus to view the consensus status.

```
[group:1]> getConsensusStatus
[
    {
        "accountType":1,
        "allowFutureBlocks":true,
        "cfgErr":false,
        "connectedNodes":3,
        "consensusedBlockNumber":6,
        "currentView":40,
        "groupId":1,
        "highestblockHash":
"0xb99703130e24702d3b580111b0cf4e39ff60ac530561dd9eb0678d03d7acce1d",
        "highestblockNumber":5,
        "leaderFailed":false,
        "max_faulty_leader":1,
        "node index":3,
        "nodeId":
"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
",
```

```
        "nodeNum":4,
        "omitEmptyBlock":true,
        "protocolId":264,
        "sealer.0":
↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",
        "sealer.1":
↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",
        "sealer.2":
↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",
        "sealer.3":
↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪",
        "toView":40
    },
    [
        {

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪":39
        },
        {

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪":36
        },
        {

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪":37
        },
        {

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪":40
        }
    ],
    {
        "prepareCache_blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        "prepareCache_height":-1,
        "prepareCache_idx":"65535",
        "prepareCache_view":"9223372036854775807"
    },
    {
        "rawPrepareCache_blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        "rawPrepareCache_height":-1,
        "rawPrepareCache_idx":"65535",
        "rawPrepareCache_view":"9223372036854775807"
    },
    {
        "committedPrepareCache_blockHash":
↪"0xbbf80db21fa393143280e01b4b711eaddd54103e95f370b389af5c0504b1eea5",
        "committedPrepareCache_height":5,
        "committedPrepareCache_idx":"1",
        "committedPrepareCache_view":"17"
    },
    {
        "signCache_cachedSize":"0"
    },
```

```
    {
        "commitCache_cachedSize":"0"
    },
    {
        "viewChangeCache_cachedSize":"0"
    }
]
```

getSyncStatus

To run getSyncStatus to view the synchronization status.

```
[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

To run getNodeVersion to view the node version.

```
[group:1]> getNodeVersion
{
        "Build Time":"20190107 10:15:23",
        "Build Type":"Linux/g++/RelWithDebInfo",
        "FISCO-BCOS Version":"2.0.0-rc1",
        "Git Branch":"master",
        "Git Commit Hash":"be95a6e3e85b621860b101c3baeee8be68f5f450"
}
```

getPeers

To run getPeers to view the peers of node.

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

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

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

                ]
        }
]
```

getGroupPeers

To run getGroupPeers to view the list of consensus and observer node of the group where the node is located.

```
[group:1]> getGroupPeers
[

↪cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bcc
↪

↪ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54b
↪
```

```
␣
↪0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca8
↪
␣
↪2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a0
]
```

getGroupList

To run getGroupList to view the list of group:

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

getBlockByHash

To run getBlockByHash to view block information according to the block hash. Parameter:

- Block hash: The hash starting with 0x.
- Transaction sign: to set it false by default, the transaction in the block only displays the hash. To set it true, it displays the transaction specific information.

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

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54b
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        "0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d"
    ],
    "transactionsRoot":
↪"0x516787f85980a86fd04b0e9ce82a1a75950db866e8cdf543c2cae3e4a51d91b7"
```

```
}
[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

To run getBlockByNumber to view block information according to the block number. Parameter:

- Block number: decimal integer.
- Transaction sign: to set it false by default, the transaction in the block only displays the hash. To set it true, it displays the transaction specific information.

```
[group:1]> getBlockByNumber 1
{
    "extraData":[

    ],
    "gasLimit":"0x0",
    "gasUsed":"0x0",
    "hash":"0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855",
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number":"0x1",
    "parentHash":
↪"0xeccad5274949b9d25996f7a96b89c0ac5c099eb9b72cc00d65bc6ef09f7bd10b",
    "sealer":"0x0",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bc
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
    "stateRoot":"0x9711819153f7397ec66a78b02624f70a343b49c60bc2f21a77b977b0ed91cef9
↪",
    "timestamp":"0x1692f119c84",
    "transactions":[
        "0xa14638d47cc679cf6eeb7f36a6d2a30ea56cb8dcf0938719ff45023a7a8edb5d"
    ],
    "transactionsRoot":
↪"0x516787f85980a86fd04b0e9ce82a1a75950db866e8cdf543c2cae3e4a51d91b7"
}
```

getBlockHashByNumber

To run getBlockHashByNumber to get hash through block number Parameter:

- Block number: decimal integer.

```
[group:1]> getBlockHashByNumber 1
0xf6afbcc3ec9eb4ac2c2829c2607e95ea0fa1be914ca1157436b2d3c5f1842855
```

getTransactionByHash

To run getTransactionByHash to check the transaction information through transaction hash. Parameter:

- Transaction hash: the transaction hash starting with 0x.

```
[group:1]> getTransactionByHash␣
↪0xed82e2cda98db8614677aba1fa8a795820bd7f68a5919a2f85018ba8c10952ac
{
        "blockHash":
↪"0x77e5b6d799edabaeae654ac5cea9baacd6f8e7ace33531d40c7ed65192de1f02",
        "blockNumber":"0x5a",
        "from":"0x7a5b31b49c6e944e9e1768785b1bc9a96cea0c17",
        "gas":"0x1c9c380",
        "gasPrice":"0x1",
        "hash":"0xed82e2cda98db8614677aba1fa8a795820bd7f68a5919a2f85018ba8c10952ac
↪",
        "input":
↪"0x10009562616c6963650000000000000000000000000000000000000000000000000000000000000006a6f726500000000000000
↪",
        "nonce":"0x18711fff2ea68dc8b8dce4a3d3845c62a0630766a448bd9453a9127efe6f9e5
↪",
        "to":"0x738eedd873bb9722173194ab990c5b9a6c0beb25",
        "transactionIndex":"0x0",
        "value":"0x0"
}
```

getTransactionByBlockHashAndIndex

To run getTransactionByBlockHashAndIndex to inquire transaction information through block hash and transaction index. Parameter:

- Block hash: the transaction hash starting with 0x.
- Transaction index: decimal integer.

```
[group:1]> getTransactionByBlockHashAndIndex␣
↪0x77e5b6d799edabaeae654ac5cea9baacd6f8e7ace33531d40c7ed65192de1f02 0
{
        "blockHash":
↪"0x77e5b6d799edabaeae654ac5cea9baacd6f8e7ace33531d40c7ed65192de1f02",
        "blockNumber":"0x5a",
        "from":"0x7a5b31b49c6e944e9e1768785b1bc9a96cea0c17",
        "gas":"0x1c9c380",
        "gasPrice":"0x1",
        "hash":"0xed82e2cda98db8614677aba1fa8a795820bd7f68a5919a2f85018ba8c10952ac
↪",
        "input":
↪"0x10009562616c6963650000000000000000000000000000000000000000000000000000000000000006a6f726500000000000000
↪",
        "nonce":"0x18711fff2ea68dc8b8dce4a3d3845c62a0630766a448bd9453a9127efe6f9e5
↪",
        "to":"0x738eedd873bb9722173194ab990c5b9a6c0beb25",
        "transactionIndex":"0x0",
        "value":"0x0"
}
```

getTransactionByBlockNumberAndIndex

To run getTransactionByBlockNumberAndIndex to inquire transaction information through block number and transaction index. Parameter:

- Block number: decimal integer.
- Transaction index: decimal integer.

```
[group:1]> getTransactionByBlockNumberAndIndex 2 0
{
        "blockHash":
↪"0x77e5b6d799edabaeae654ac5cea9baacd6f8e7ace33531d40c7ed65192de1f02",
        "blockNumber":"0x5a",
        "from":"0x7a5b31b49c6e944e9e1768785b1bc9a96cea0c17",
        "gas":"0x1c9c380",
        "gasPrice":"0x1",
        "hash":"0xed82e2cda98db8614677aba1fa8a795820bd7f68a5919a2f85018ba8c10952ac
↪",
        "input":
↪"0x10009562616c6963650000000000000000000000000000000000000000000000000000000000000006a6f726500000000000000
↪",
        "nonce":"0x18711fff2ea68dc8b8dce4a3d3845c62a0630766a448bd9453a9127efe6f9e5
↪",
        "to":"0x738eedd873bb9722173194ab990c5b9a6c0beb25",
        "transactionIndex":"0x0",
        "value":"0x0"
}
```

getTransactionReceipt

To run getTransactionReceipt to inquire transaction receipt through transaction hash. Parameter:

- Transaction hash: the transaction hash starting with 0x.
- contract name: Optional. The contract name generated by transaction receipt. To use this parameter can parse and output the event log in the transaction receipt.
- event name: optional. Event Name. To specify this parameter to output the specified event log information.
- event index number: optional. Event index. To specify this parameter to output the event log information of the specified event index location.

```
[group:1]> getTransactionReceipt␣
↪0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1
{
    "blockHash":"0x68a1f47ca465acc89edbc24115d1b435cb39fa0def53e8d0ad8090cf1827cafd
↪",
    "blockNumber":"0x5",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xc44e7a8a4ae20d6afaa43221c6120b5e1e9f9a72",
    "gasUsed":"0x8be5",
    "logs":[
        {
            "address":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0x66f7705280112a4d1145399e0414adc43a2d6974b487710f417edcf7d4a39d71
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
    "transactionHash":
↪"0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1",
```

```
    "transactionIndex":"0x0"
}

[group:1]> getTransactionReceipt␣
↪0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1 TableTest
{
    "blockHash":"0x68a1f47ca465acc89edbc24115d1b435cb39fa0def53e8d0ad8090cf1827cafd
↪",
    "blockNumber":"0x5",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xc44e7a8a4ae20d6afaa43221c6120b5e1e9f9a72",
    "gasUsed":"0x8be5",
    "logs":[
        {
            "address":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0x66f7705280112a4d1145399e0414adc43a2d6974b487710f417edcf7d4a39d71
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x000000000000000000000000000000000000000000000000000000000000000000040000000000000000000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
    "transactionHash":
↪"0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1",
    "transactionIndex":"0x0"
}
---------------------------------------------------------------------------------
↪----------
Event logs
---------------------------------------------------------------------------------
↪----------
InsertResult index: 0
count = 1
---------------------------------------------------------------------------------
↪----------

[group:1]> getTransactionReceipt␣
↪0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1 TableTest␣
↪InsertResult
{
    "blockHash":"0x68a1f47ca465acc89edbc24115d1b435cb39fa0def53e8d0ad8090cf1827cafd
↪",
    "blockNumber":"0x5",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xc44e7a8a4ae20d6afaa43221c6120b5e1e9f9a72",
    "gasUsed":"0x8be5",
    "logs":[
        {
            "address":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0x66f7705280112a4d1145399e0414adc43a2d6974b487710f417edcf7d4a39d71
↪"
            ]
        }
```

```
    ],
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
    "transactionHash":
↪"0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1",
    "transactionIndex":"0x0"
}
---------------------------------------------------------------------------------
↪----------
Event logs
---------------------------------------------------------------------------------
↪----------
InsertResult index: 0
count = 1
---------------------------------------------------------------------------------
↪----------

[group:1]> getTransactionReceipt␣
↪0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1 TableTest␣
↪InsertResult 0
{
    "blockHash":"0x68a1f47ca465acc89edbc24115d1b435cb39fa0def53e8d0ad8090cf1827cafd
↪",
    "blockNumber":"0x5",
    "contractAddress":"0x0000000000000000000000000000000000000000",
    "from":"0xc44e7a8a4ae20d6afaa43221c6120b5e1e9f9a72",
    "gasUsed":"0x8be5",
    "logs":[
        {
            "address":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
            "data":
↪"0x0000000000000000000000000000000000000000000000000000000000000001",
            "topics":[
                "0x66f7705280112a4d1145399e0414adc43a2d6974b487710f417edcf7d4a39d71
↪"
            ]
        }
    ],
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪",
    "output":"0x0000000000000000000000000000000000000000000000000000000000000001",
    "status":"0x0",
    "to":"0xd653139b9abffc3fe07573e7bacdfd35210b5576",
    "transactionHash":
↪"0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1",
    "transactionIndex":"0x0"
}
---------------------------------------------------------------------------------
↪----------
Event logs
---------------------------------------------------------------------------------
↪----------
InsertResult index: 0
count = 1
---------------------------------------------------------------------------------
↪----------
```

getPendingTransactions

To run getPendingTransactions to inquire the transactions waiting to be processed.

```
[group:1]> getPendingTransactions
[]
```

getPendingTxSize

To run getPendingTxSize to inquire the number of transactions waiting to be processed.

```
[group:1]> getPendingTxSize
0
```

getCode

To run getCode to inquire contract code according contract address. Parameter:

- Contract address: The contract address starting with 0x(To deploy contract can get contract address).

```
[group:1]> getCode 0x97b8c19598fd781aaeb53a1957ef9c8acc59f705
0x60606040526000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff16806
```

getTotalTransactionCount

To run getTotalTransactionCount to inquire the current block number and the total number of transaction.

```
[group:1]> getTotalTransactionCount
{
        "blockNumber":1,
        "txSum":1,
        "failedTxSum":0
}
```

deploy

To run deploy to deploy contract. (By default, HelloWorld contract and TableTest.sol are provided for example) Parameter:

- Contract name: deployment contract name (can be suffixed with .sol). It can name as either HelloWorld or HelloWorld.sol.

```
# To deploy HelloWorld contract
[group:1]> deploy HelloWorld.sol
contract address:0xb3c223fc0bf6646959f254ac4e4a7e355b50a344

# To deploy TableTest contract
[group:1]> deploy TableTest.sol
contract address:0x3554a56ea2905f366c345bd44fa374757fb4696a
```

Note:

- For deploying a user-written contract, we just need to place the solidity contract file in the `contracts/solidity/` directory of the console root, and then deploy it. Press the tab key to search for the contract name in the `contracts/solidity` directory.
- If the contract need to be deployed refers to other contracts or libraries, the reference format is `import "./XXX.sol";`. The related contracts and libraries are placed in the `contracts/solidity/` directory.

- If contract references the library library, the name of library file must start with `Lib` string to distinguish between the normal contract and the library file. Library files cannot be deployed and called separately.
- **Because FISCO BCOS has removed the transfer payment logic of Ethereum, the solidity contract does not support using `payable` keyword. This keyword will cause the Java contract file converted by solidity contract to fail at compilation. **

getDeployLog

Run getDeployLog to query the log information of the contract deployed by **current console** in the group. The log information includes the time of deployment contract, the group ID, the contract name, and the contract address. parameter:

- Log number: optional. To return the latest log information according to the expected value entered. When the actual number is less than the expected value, it returns by the actual number. When the expected value is not given, it returns by the latest 20 log information by default.

```
[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
```

Note: If you want to see all the deployment contract log information, please check the `deploylog.txt` file in the `console` directory. The file only stores the log records of the last 10,000 deployment contracts.

call

To run call to call contract. Parameter:

- Contract name: the contract name of the deployment (can be suffixed with .sol).
- Contract address: the address obtained by the deployment contract. The contract address can omit the prefix 0. For example, 0x000ac78 can be abbreviated as 0xac78.
- Contract interface name: the called interface name.
- Parameter: determined by contract interface parameters.

**Parameters are separated by spaces. The string and byte type parameters need to be enclosed in double quotes; array parameters need to be enclosed in brackets, such as [1,2,3]; array is a string or byte type and needs to be enclosed in double quotation marks, such as [“alice”, “bob”]. Note that there are no spaces in the array parameters; boolean types are true or false. **

```
```text
# To call the get interface of HelloWorld to get the name string
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 get
Hello, World!

# To call the set interface of HelloWorld to set the name string
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 set
↪"Hello, FISCO BCOS"
transaction hash:0xa7c7d5ef8d9205ce1b228be1fe90f8ad70eeb6a5d93d3f526f30d8f431cb1e70

# To call the get interface of HelloWorld to get the name string for checking␣
↪whether the settings take effect
```
```

```
[group:1]> call HelloWorld.sol 0xc0ce097a5757e2b6e189aa70c7d55770ace47767 get
Hello, FISCO BCOS

# Call the create interface of TableTest to create the user table t_test. The␣
↪create interface calls the createResult event, and the event log will output.

# Call the create interface of TableTest to create the user table t_test, the␣
↪create interface calls the createResult event, and the event log will output.

# event log consists of the event name, the event log index number, and the event␣
↪variable, which is convenient for users to view the status of the variable after␣
↪sending transaction. createResult event records the value count which is␣
↪returned by the create interface creation table.

[group:1]> call TableTest.sol 0xd653139b9abffc3fe07573e7bacdfd35210b5576 create
transaction hash:0x895980dd6ef37004bb32a7f417daa3b5d0bdb1f16e8a62cc9251e5948c612bb5
----------------------------------------------------------------------------------
↪----------
Event logs
----------------------------------------------------------------------------------
↪----------
CreateResult index: 0
count = 0
----------------------------------------------------------------------------------
↪----------

# To call the insert interface of TableTest to insert the record, the fields are␣
↪name, item_id, item_name
[group:1]> call TableTest.sol 0xd653139b9abffc3fe07573e7bacdfd35210b5576 insert
↪"fruit" 1 "apple"
transaction hash:0x6393c74681f14ca3972575188c2d2c60d7f3fb08623315dbf6820fc9dcc119c1
----------------------------------------------------------------------------------
↪----------
Event logs
----------------------------------------------------------------------------------
↪----------
InsertResult index: 0
count = 1
----------------------------------------------------------------------------------
↪----------

# To call TableTest's select interface to inquiry records
[group:1]> call TableTest.sol 0xd653139b9abffc3fe07573e7bacdfd35210b5576 select
↪"fruit"
[[fruit], [1], [apple]]
```

Note: TableTest.sol contract codeReference here。

deployByCNS

Run deployByCNS and deploy the contract with CNS. Contracts deployed with CNS can be called directly with the contract name.

Parameter:

- Contract name: deployable contract name.
- Contract version number: deployable contract version number(the length cannot exceed 40).

```
# To deploy HellowWorld contract 1.0 version
[group:1]> deployByCNS HelloWorld.sol 1.0
contract address:0x3554a56ea2905f366c345bd44fa374757fb4696a
```

```
# To deploy HellowWorld contract 2.0 version
[group:1]> deployByCNS HelloWorld.sol 2.0
contract address:0x07625453fb4a6459cbf14f5aa4d574cae0f17d92

# To deploy TableTest contract
[group:1]> deployByCNS TableTest.sol 1.0
contract address:0x0b33d383e8e93c7c8083963a4ac4a58b214684a8
```

Note:

- For deploying the contracts compiled by users only needs to place the solidity contract file in the `contracts/solidity/` directory of the console root and to deploy it. Press tab key to search for the contract name in the `contracts/solidity/` directory.
- If the contract to be deployed references other contracts or libraries, the reference format is `import "./XXX.sol";`. The related contract and library are placed in the `contracts/solidity/` directory.
- **Because FISCO BCOS has removed the transfer payment logic of Ethereum, the solidity contract does not support using `payable` keyword. This keyword will cause the Java contract file converted by solidity contract to fail at compilation. **

queryCNS

Run queryCNS and query the CNS table record information (the mapping of contract name and contract address) according to the contract name and contract version number (optional parameter).

Parameter:

- Contract name: deployable contract name.
- Contract version number: (optional) deployable contract version number.

```
[group:1]> queryCNS HelloWorld.sol
----------------------------------------------------------------------------------------
↪----------
|                   version                   |                   address
↪          |
|                     1.0                     |
↪0x3554a56ea2905f366c345bd44fa374757fb4696a  |
----------------------------------------------------------------------------------------
↪----------

[group:1]> queryCNS HelloWorld 1.0
----------------------------------------------------------------------------------------
↪----------
|                   version                   |                   address
↪          |
|                     1.0                     |
↪0x3554a56ea2905f366c345bd44fa374757fb4696a  |
----------------------------------------------------------------------------------------
↪----------
```

callByCNS

To run deployByCNS and deploy the contract with CNS. Parameter:

- Contract name and contract version number: The contract name and contract version number are separated by colon, such as `HelloWorld:1.0` or `HelloWorld.sol:1.0`. When the contract version number is omitted like `HelloWorld` or `HelloWorld.sol`, the latest version of the contract is called.
- Contract interface name: The called contract interface name.

- Parameter: is determined by the parameter of contract interface. **The parameters are separated by spaces, where the string and byte type parameters need to be enclosed in double quotation marks; the array parameters need to be enclosed in brackets, such as [1, 2, 3]. The array is a string or byte type with double quotation marks such as ["alice", "bob"]; the boolean type is true or false.**

```
# To call the HelloWorld contract 1.0 version to set the name string by the set␣
↪interface
[group:1]> callByCNS HelloWorld:1.0 set "Hello,CNS"
transaction hash:0x80bb37cc8de2e25f6a1cdcb6b4a01ab5b5628082f8da4c48ef1bbc1fb1d28b2d

# To call the HelloWorld contract 2.0 version to set the name string by the set␣
↪interface
[group:1]> callByCNS HelloWorld:2.0 set "Hello,CNS2"
transaction hash:0x43000d14040f0c67ac080d0179b9499b6885d4a1495d3cfd1a79ffb5f2945f64

# To call the HelloWorld contract 1.0 version to get the name string by the get␣
↪interface
[group:1]> callByCNS HelloWorld:1.0 get
Hello,CNS

# To call the HelloWorld contract 2.0 version to get the name string by the get␣
↪interface
[group:1]> callByCNS HelloWorld get
Hello,CNS2
```

addSealer

To run addSealer to add the node as a consensus node. Parameter:

- node's nodeId

```
[group:1]> addSealer␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

addObserver

To run addObserver to add the node as an observed node. Parameter:

- node's nodeId

```
[group:1]> addObserver␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

removeNode

To run removeNode to exit the node. The exit node can be added as a consensus node by the addSealer command or can be added as an observation node by the addObserver command. Parameter:

- node's nodeId

```
[group:1]> removeNode␣
↪ea2ca519148cafc3e92c8d9a8572b41ea2f62d0d19e99273ee18cccd34ab50079b4ec82fe5f4ae51bd95dd788811c971
{
        "code":0,
        "msg":"success"
}
```

setSystemConfigByKey

To run setSystemConfigByKey to set the system configuration in key-value pairs. The currently system configuration supports `tx_count_limit` and `tx_gas_limit`. The key name of these two configuration can be complemented by the tab key:

- tx_count_limit: block maximum number of packaged transactions
- tx_gas_limit: The maximum number of gas allowed to be consumed

Parameters:

- key
- value

```
[group:1]> setSystemConfigByKey tx_count_limit 100
{
        "code":0,
        "msg":"success"
}
```

getSystemConfigByKey

To run getSystemConfigByKe to inquire the value of the system configuration according to the key. Parameter:

- key

```
[group:1]> getSystemConfigByKey tx_count_limit
100
```

grantPermissionManager

Run grantPermissionManager to grant the account's chain administrator privileges. parameter:

- account address

```
[group:1]> grantPermissionManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

**Note: For an example of the using permission control related commands, refer to Permission Control Manual Document. **

listPermissionManager

To run listPermissionManager to inquire the list of permission records with administrative privileges.

```
[group:1]> listPermissionManager
-------------------------------------------------------------------------------------------
↪----------
|                    address                    |                  enable_num          ␣
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                       2              ␣
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

revokePermissionManager

To run revokePermissionManager to revoke the permission management of the external account address. parameter:

- account address

```
[group:1]> revokePermissionManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantUserTableManager

Run grantUserTableManager to grant the account to write to the user table.

parameter:

- table name
- account address

```
[group:1]> grantUserTableManager t_test 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listUserTableManager

Run listUserTableManager to query the account's table that has writing permission to the user table.

parameter:

- table name

```
[group:1]> listUserTableManager t_test
-------------------------------------------------------------------------------------------
↪----------
|                    address                    |                  enable_num          ␣
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                       2              ␣
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

revokeUserTableManager

Run revokeUserTableManager to revoke the account's writing permission from the user table.

parameter:

- table name
- account address

```
[group:1]> revokeUserTableManager t_test 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantDeployAndCreateManager

Run grantDeployAndCreateManager to grant the account's permission of deployment contract and user table creation.

parameter:

- account address

```
[group:1]> grantDeployAndCreateManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listDeployAndCreateManager

Run listDeployAndCreateManager to query the account's permission of deployment contract and user table creation.

```
[group:1]> listDeployAndCreateManager
----------------------------------------------------------------------------------------
↪----------
|                    address                    |                  enable_num          ␣
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                       2              ␣
↪          |
----------------------------------------------------------------------------------------
↪----------
```

revokeDeployAndCreateManager

Run revokeDeployAndCreateManager to revoke the account's permission of deployment contract and user table creation.

parameter:

- account address

```
[group:1]> revokeDeployAndCreateManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantNodeManager

Run grantNodeManager to grant the account's node management permission.

parameter:

- account address

```
[group:1]> grantNodeManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listNodeManager

Run the listNodeManager to query the list of accounts that have node management.

```
[group:1]> listNodeManager
-------------------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

revokeNodeManager

Run revokeNodeManager to revoke the account's node management permission.

parameter:

- account address

```
[group:1]> revokeNodeManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantCNSManager

Run grantCNSManager to grant the account's permission of using CNS. parameter:

- account address

```
[group:1]> grantCNSManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listCNSManager

Run listCNSManager to query the list of accounts that have CNS.

```
[group:1]> listCNSManager
--------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2
↪          |
--------------------------------------------------------------------------------------
↪----------
```

revokeCNSManager

Run revokeCNSManager to revoke the account's permission of using CNS. parameter:

- account address

```
[group:1]> revokeCNSManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

grantSysConfigManager

Run grantSysConfigManager to grant the account's permission of modifying system parameter. parameter:

- account address

```
[group:1]> grantSysConfigManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
        "msg":"success"
}
```

listSysConfigManager

Run listSysConfigManager to query the list of accounts that have modified system parameters.

```
[group:1]> listSysConfigManager
--------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d  |                      2
↪          |
--------------------------------------------------------------------------------------
↪----------
```

revokeSysConfigManager

Run revokeSysConfigManager to revoke the account's permission of modifying system parameter. parameter:

- account address

```
[group:1]> revokeSysConfigManager 0xc0d0e6ccc0b44c12196266548bec4a3616160e7d
{
        "code":0,
```

```
        "msg":"success"
}
```

quit

To run quit, q or exit to exit the console.

```
quit
```

[create sql]

Run create sql statement to create a user table in mysql statement form.

```
# Create user table t_demo whose primary key is name and other fields are item_id␣
↪and item_name
[group:1]> create table t_demo(name varchar, item_id varchar, item_name varchar,␣
↪primary key(name))
Create 't_demo' Ok.
```

Note:

- The field types for creating table are all string types. Even if other field types of the database are provided, the field types have to be set according to the string type.
- The primary key field must be specified. For example, to create a t_demo table with the primary key field as name.
- The primary key of the table has different concept from the primary key in the relational database. Here, the value of the primary key is not unique, and the primary key value needs to be passed when the blockchain underlying platform is handling records.
- You can specify the field as the primary key, but the setting fields such as self-incrementing, non-empty, indexing, etc do not work.

desc

Run desc statement to query the field information of the table in mysql statement form.

```
# query the field information of the t_demo table. you can view the primary key␣
↪name and other field names of the table.

[group:1]> desc t_demo
{
    "key":"name",
    "valueFields":"item_id,item_name"
}
```

[insert sql]

Run insert sql statement to insert the record in the mysql statement form.

```
[group:1]> insert into t_demo (name, item_id, item_name) values (fruit, 1, apple1)
Insert OK, 1 row affected.
```

Note:

- must insert a record sql statement with the primary key field value of the table.

- The enter values with punctuation, spaces, or strings containing letters starting with a number requires double quotation marks, and no more double quotation marks are allowed inside.

[select sql]

Run select sql statement to query the record in mysql statement form.

```
# query the records contain all fields
select * from t_demo where name = fruit
{item_id=1, item_name=apple1, name=fruit}
1 row in set.

# query the records contain the specified fields
[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.

# insert a new record
[group:1]> insert into t_demo values (fruit, 2, apple2)
Insert OK, 1 row affected.

# use the keyword 'and' to connect multiple query condition
[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.

# use limit field to query the first line of records. If the offset is not␣
↪provided, it is 0 by default.
[group:1]> select * from t_demo where name = fruit limit 1
{item_id=1, item_name=apple1, name=fruit}
1 row in set.

# use limit field to query the second line record. The offset is 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.
```

Note:

- For querying the statement recording sql, the primary key field value of the table in the where clause must be provided.
- The limit field in the relational database can be used. Providing two parameters which are offset and count.
- The where clause only supports the keyword 'and'. Other keywords like 'or', 'in', 'like', 'inner', 'join', 'union', subquery, multi-table joint query, and etc. are not supported.
- The enter values with punctuation, spaces, or strings containing letters starting with a number requires double quotation marks, and no more double quotation marks are allowed inside.

[update sql]

Run update sql statement to update the record in mysql statement form.

```
[group:1]> update t_demo set item_name = orange where name = fruit and item_id = 1
Update OK, 1 row affected.
```

Note:

- For updating the where clause of recording sql statement, the primary key field value of the table in the where clause must be provided.

- The enter values with punctuation, spaces, or strings containing letters starting with a number requires double quotation marks, and no more double quotation marks are allowed inside.

[delete sql]

Run delete sql statement to delete the record in mysql statement form.

```
[group:1]> delete from t_demo where name = fruit and item_id = 1
Remove OK, 1 row affected.
```

Note:

- For deleting the where clause of recording sql statement, the primary key field value of the table in the where clause must be provided.
- The enter values with punctuation, spaces, or strings containing letters starting with a number requires double quotation marks, and no more double quotation marks are allowed inside.

6.5.7 Appendix: Java environment configuration

Install Java in ubuntu environment

```
# Install the default Java version (Java 8 version or above)
sudo apt install -y default-jdk
# query Java version
java -version
```

Install Java in CentOS environment

```
# To inquire the original Java version of centos
$ rpm -qa|grep java
# To delete the Java version that is inquired
$ rpm -e --nodeps java version
# To inquire Java version. It is deleted finish without version number appears
$ java -version
# To create new folder to install Java 8 version or above. To put the downloaded␣
↪jdk in the software directory
# Download Java 8 version or above from openJDK official website (https://jdk.java.
↪net/java-se-ri/8) or Oracle official website (https://www.oracle.com/technetwork/
↪java/javase/downloads/index.html). For example, to download jdk-8u201-linux-x64.
↪tar.gz
$ mkdir /software
# To unzip jdk
$ tar -zxvf jdk-8u201-linux-x64.tar.gz
# To configure the Java environment and edit the /etc/profile file.
$ vim /etc/profile
# After opening the file, to enter the following three sentences into the file and␣
↪exit
export JAVA_HOME=/software/jdk-8u201-linux-x64.tar.gz
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
# profile takes effect
$ source /etc/profile
# To inquire the Java version. If the result shows the version you just downloaded,
↪ the installation is successful.
java -version
```

6.6 Smart contract development

FISCO BCOS platform currently supports three smart contract forms which are Solidity, CRUD, and pre-compiled.

- The Solidity contract is the same as Ethereum on supporting the latest version.
- The CRUD interface supporting the distributed storage pre-compilation contract in the Solidity contract, which can store the data of Solidtiy contract in the AMDB table structure, realizes the separation of contract logic and data.
- The precompiled (precompiled) contract is developed in C++ and built into the FISCO BCOS platform. It has better performance than the Solidity contract. Its contract interface that needs to be pre-determined when compiling, is suitable for the scenarios with fixed logic but consensus, such as group configuration. The development of precompiled contracts will be introduced in the next section.

6.6.1 Solidity contract development

- Solidity official file
- Remix online IDE

6.6.2 To use contract CRUD interface

Accessing AMDB requires using the AMDB-specific smart contract interface `Table.sol` which is a database contract that can create tables and add, delete, and modify the tables.

Note: To make the table created by AMDB accessible to multiple contracts, it should have a unique name that acknowledged globally. So it is unable to create tables with same name within one group on the same chain

`Table.sol` file code is as follows:

```
pragma solidity ^0.4.24;

contract TableFactory {
    function openTable(string) public constant returns (Table);  // open table
    function createTable(string,string,string) public returns(int);  // create␣
↪table
}

// inquiry conditions
contract Condition {
    //equal to
    function EQ(string, int) public;
    function EQ(string, string) public;

    //unequal to
    function NE(string, int) public;
    function NE(string, string)  public;

    //greater than
    function GT(string, int) public;
    //greater than or equal to
    function GE(string, int) public;

    //smaller than
    function LT(string, int) public;
    //smaller than or equal to
    function LE(string, int) public;
```

```
    //limit the number of return record
    function limit(int) public;
    function limit(int, int) public;
}

// single entry data record
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 set(string, int) public;
    function set(string, string) public;
}

// data record set
contract Entries {
    function get(int) public constant returns(Entry);
    function size() public constant returns(int);
}

// Table main type
contract Table {
    // select interface
    function select(string, Condition) public constant returns(Entries);
    // insert interface
    function insert(string, Entry) public returns(int);
    // update interface
    function update(string, Entry, Condition) public returns(int);
    // remove interface
    function remove(string, Condition) public returns(int);

    function newEntry() public constant returns(Entry);
    function newCondition() public constant returns(Condition);
}
```

To provide a contract case `TableTest.sol`. The code is as follows:

```
pragma solidity ^0.4.24;

import "./Table.sol";

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

    // create table
    function create() public returns(int){
        TableFactory tf = TableFactory(0x1001);  // TableFactory's address is␣
↪fixed at 0x1001
        // To create a table t_test. Table's key_field as name. Table's value_
↪field as item_id and item_name.
        // key_field indicates the row that AMDB's primary key value_field␣
↪represents in the table. The row can be multiple and spearated by commas.
        int count = tf.createTable("t_test", "name", "item_id,item_name");
        emit CreateResult(count);

        return count;
    }
```

```
    // inquiry data
    function select(string name) public constant returns(bytes32[], int[],␣
↪bytes32[]){
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

        // If the condition is empty, it means no filtering. You can use␣
↪conditional filtering as needed.
        Condition condition = table.newCondition();

        Entries entries = table.select(name, condition);
        bytes32[] memory user_name_bytes_list = new bytes32[](uint256(entries.
↪size()));
        int[] memory item_id_list = new int[](uint256(entries.size()));
        bytes32[] memory item_name_bytes_list = new bytes32[](uint256(entries.
↪size()));

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

            user_name_bytes_list[uint256(i)] = entry.getBytes32("name");
            item_id_list[uint256(i)] = entry.getInt("item_id");
            item_name_bytes_list[uint256(i)] = entry.getBytes32("item_name");
        }

        return (user_name_bytes_list, item_id_list, item_name_bytes_list);
    }
    // insert data
    function insert(string name, int item_id, string item_name) public␣
↪returns(int) {
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

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

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

        return count;
    }
    // update data
    function update(string name, int item_id, string item_name) public␣
↪returns(int) {
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

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

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

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

        return count;
    }
    // remove data
```

```
    function remove(string name, int item_id) public returns(int){
        TableFactory tf = TableFactory(0x1001);
        Table table = tf.openTable("t_test");

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

        int count = table.remove(name, condition);
        emit RemoveResult(count);

        return count;
    }
}
```

`TableTest.sol` has called the intelligent contract `Table.sol` of AMDB, which implements creating the user table `t_test` and the functions of adding, deleting and changing `t_test`. The `t_test` table is structured as follows. This table records the item and item's numbers used by a employees.

The client requiring to call the contract code which is converted to Java file, needs to put TableTest.sol and Table.sol into the directory contracts/solidity, and TableTest.java is generated by the compile script of sol2java.sh.

6.6.3 Precompiled contract development

1. Introduction

Precompiled contract is a natively supported feature of Ethereum: a contract that uses C++ code to implement specific functions at the underlying platform for EVM module calling. FISCO BCOS inherits and extends this feature, and has developed a powerful and easy-to-expand framework on this basis of it.precompiled design principle.

This article is an introductory to guide users on how to implement their own precompiled contracts and how to call them.

2. Implement precompiled contracts

2.1 Process

The process of implementing a pre-compiled contract:

- **assign contract address**

For calling a solid contract or pre-compiled contract, you need to distinguish it by the contract address and address space.

The address range of user assigned interval is `0x5001-0xffff`. Users needs to assign an unused address to the new precompiled contract. **The precompiled contract addresses must be unique and not conflicting**.

List of precompiled contracts and address assignments implemented in FISCO BCOS:

- **define contract interface**

It is similar to solidity contract. When designing a contract, you need to determine the ABI interface of the contract first. The ABI interface rules of the precompiled contract are exactly the same as the solidity. solidity ABI link.

> When defining a precompiled contract interface, you usually need to define a solidity contract with the same interface, and empty the function body of all interfaces. This contract is called **interface contract** of the precompiled contract. The interface contract need to be used when calling the precompiled contract.

```
    pragma solidity ^0.4.24;
    contract Contract_Name {
        function interface0(parameters ... ) {}
        ....
        function interfaceN(parameters ... ) {}
    }
```

- **design storage structure**

When a precompiled contract involves a storage operation, it needs to determine the stored table information (table name and table structure. The stored data will be uniformly abstracted into a table structure in FISCO BCOS)storage structure.

Note: This process can be omitted without involving a storage operation.

- **implement contract logic**

For implementing the calling logic of the new contract, you need to implement a new C++ class that needs to inherit [precompiled] (https://github.com/FISCO-BCOS/FISCO-BCOS/blob/master/libblockverifier/Precompiled.h) #L37) to overload the call function for achieving the calling behaviour of each interface.

```
    // libblockverifier/Precompiled.h
    class Precompiled
    {
        virtual bytes call(std::shared_ptr<ExecutiveContext> _context,␣
↪bytesConstRef _param,
            Address const& _origin = Address()) = 0;
    };
```

The call function has three parameters:

`std::shared_ptr<ExecutiveContext> _context` : the context for the transaction execution saving

`bytesConstRef _param` : calling the parameter information of the contract. The calling corresponding contract interface and the parameters of interface can be obtained from _param parsing.

`Address const& _origin` : transaction sender for permission control

How to implement a Precompiled class will be detailed in the sample below.

- **register contract**

Finally, the contract address and the corresponding class need to be registered to the execution context of the contract, so that the execution logic of the contract can be correctly recognized when the precompiled contract is called by the address. To view the registered pre-compiled contract list.

Registration path:

```
    file        libblockverifier/ExecutiveContextFactory.cpp
    function    initExecutiveContext
```

2.2 sample contract development

```
// 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;
    }
}
```

The above source code is the HelloWorld contract written by solidity. This chapter will implement a precompiled contract with the same function to enable user step by step to have an visual understanding to the precompiled contract. sample c++source code path:

```
    libprecompiled/extension/HelloWorldPrecompiled.h
    libprecompiled/extension/HelloWorldPrecompiled.cpp
```

2.2.1 assign contract address

Referring to the address range, the address of the HelloWorld precompiled contract is assigned as:

```
0x5001
```

2.2.2 define contract interface

We need to implement the HelloWorld contract function. The interface is the same as the HelloWorld interface. HelloWorldPrecompiled interface contract:

```
pragma solidity ^0.4.24;

contract HelloWorldPrecompiled {
    function get() public constant returns(string) {}
    function set(string _m) {}
}
```

2.2.3 design storage structure

HelloWorldPrecompiled needs to store the string value of the set, so when it comes to storage operations, you need to design the stored table structure.

table name: `_ext_hello_world_`

table structure:

The table stores only a pair of key-value pairs. The key field is hello_key and the value field is hello_value. For storing the corresponding string value, it can be modified by the set(string) interface and obtained by the get() interface.

2.2.4 implement call logic

To add the HelloWorldPrecompiled class, overload the call function, and implement the calling behavior of all interfaces.call function source code.

The user-defined Precompiled contract needs to add a new class for defining the calling behaviour of the contract in the class. In the example, for adding the HelloWorldPrecompiled class, the following work must complete:

- interface registration

```
// define all interfaces in the class
const char* const HELLO_WORLD_METHOD_GET = "get()";
const char* const HELLO_WORLD_METHOD_SET = "set(string)";

// register interface in the constructor
HelloWorldPrecompiled::HelloWorldPrecompiled()
{
    // name2Selector is a member of the Base class Precompiled, which saves the␣
↪mapping relationship of the interface call.
    name2Selector[HELLO_WORLD_METHOD_GET] = getFuncSelector(HELLO_WORLD_METHOD_
↪GET);
    name2Selector[HELLO_WORLD_METHOD_SET] = getFuncSelector(HELLO_WORLD_METHOD_
↪SET);
}
```

- table creation

define the table's name and field structure

```
// define the name
const std::string HELLO_WORLD_TABLE_NAME = "_ext_hello_world_";
// define the key field
const std::string HELLOWORLD_KEY_FIELD = "key";
// "field0,field1,field2" define other fields, multiple fields separated by commas,
↪ such as "field0,field1,field2"
const std::string HELLOWORLD_VALUE_FIELD = "value";
```

```
// In the call function, the table is opened when it exists, otherwise the table␣
↪is created first.
Table::Ptr table = openTable(_context, HELLO_WORLD_TABLE_NAME);
if (!table)
{
    //  table is created while it does not exist
    table = createTable(_context, HELLO_WORLD_TABLE_NAME, HELLOWORLD_KEY_FIELD,
        HELLOWORLD_VALUE_FIELD, _origin);
    if (!table)
    {
        //  fail to create and return false
    }
}
```

After getting the operation handle of the table, user can implement the specific logic of the table operation.

- call interface distinguishing

Parsing _param with getParamFunc can distinguish the call interface.

Note: the contract interface must be registered in the constructor

```
uint32_t func = getParamFunc(_param);
if (func == name2Selector[HELLO_WORLD_METHOD_GET])
{
    // get() call interface logic
}
else if (func == name2Selector[HELLO_WORLD_METHOD_SET])
{
    // set(string) call interface logic
}
else
{
    // unknown interface, call error, return error
}
```

- Parameter parsing and result return

The parameters during calling the contract are included in the _param parameter of the call function. They are encoded according to the Solidity ABI format. The `dev::eth::ContractABI` utility class can be used to parse the parameters. Similarly, when the interface returns, the return value also needs to be encoded according to the format. Solidity ABI。

In `dev::eth::ContractABI` class, we need to use two interfaces `abiIn abiOut`. The former serializes the former user parameter and the latter can parse the parameter from the serialized data.

```
// to serialize ABI data. c++ type data serialized to the format used by evm
// _id: The corresponding string of the function interface declaration, which␣
↪generally default to ""
template <class... T> bytes abiIn(std::string _id, T const&... _t)
// to parse serialized data into c++ type data
template <class... T> void  abiOut(bytesConstRef _data, T&... _t)
```

The sample code below shows how the interface works:

```
// for transfer interface: transfer(string,string,uint256)

// Parameter1
std::string str1 = "fromAccount";
// Parameter12
std::string str2 = "toAccount";
// Parameter13
uint256 transferAmoumt = 11111;

dev::eth::ContractABI abi;
// serialization, abiIn first string parameter default to ""
bytes out = abi.abiIn("", str1, str2, transferAmoumt);

std::string strOut1;
std::string strOut2;
uint256 amoumt;

// parse parameter
abi.abiOut(out, strOut1, strOut2, amount);
// parse after
// strOut1 = "fromAccount";
// strOut2 = "toAccount"
// amoumt = 11111
```

Finally, the HelloWorldPrecompiled call function is implemented completely.source code link.

```
bytes HelloWorldPrecompiled::call(dev::blockverifier::ExecutiveContext::Ptr _
↪context,
    bytesConstRef _param, Address const& _origin)
{
    // parse function interface
    uint32_t func = getParamFunc(_param);
    //
    bytesConstRef data = getParamData(_param);
    bytes out;
    dev::eth::ContractABI abi;

    // open table
    Table::Ptr table = openTable(_context, HELLO_WORLD_TABLE_NAME);
    if (!table)
    {
        // table is created while it does not exist
        table = createTable(_context, HELLO_WORLD_TABLE_NAME, HELLOWORLD_KEY_FIELD,
            HELLOWORLD_VALUE_FIELD, _origin);
        if (!table)
```

```
        {
            // fail to create table. no authority?
            out = abi.abiIn("", CODE_NO_AUTHORIZED);
            return out;
        }
    }

    // to distinguish the calling interface and specify the calling logic of each␣
↪interface
    if (func == name2Selector[HELLO_WORLD_METHOD_GET])
    {  // get() call interface
        // default to return value
        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) call interface

        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())
        {  // value exists, update
            count = table->update(HELLOWORLD_KEY_FIELD_NAME, entry, table->
↪newCondition(),
                std::make_shared<AccessOptions>(_origin));
        }
        else
        {  // value does not exist, insert
            count = table->insert(
                HELLOWORLD_KEY_FIELD_NAME, entry, std::make_shared<AccessOptions>(_
↪origin));
        }

        if (count == CODE_NO_AUTHORIZED)
        {  // no table operation authority
            PRECOMPILED_LOG(ERROR) << LOG_BADGE("HelloWorldPrecompiled") << LOG_
↪DESC("set")
                                   << LOG_DESC("non-authorized");
        }
        out = abi.abiIn("", u256(count));
    }
    else
    {  // parameter error, unknown calling interface
        PRECOMPILED_LOG(ERROR) << LOG_BADGE("HelloWorldPrecompiled") << LOG_DESC("␣
↪unknown func ")
                               << LOG_KV("func", func);
        out = abi.abiIn("", u256(CODE_UNKNOW_FUNCTION_CALL));
    }
```

```
    return out;
}
```

2.2.5 Register contract and compile source code

- Register Precompiled contract. Modify `FISCO-BCOS/cmake/templates/UserPrecompiled.h.in`, register the address of `HelloWorldPrecompiled` contract in its below function. Default to be existed, and revoke annotation.

```
void␣
↪dev::blockverifier::ExecutiveContextFactory::registerUserPrecompiled(dev::blockverifier::Executi
↪context)
{
    // Address should in [0x5001,0xffff]
    context->setAddress2Precompiled(Address(0x5001), std::make_shared
↪<dev::precompiled::HelloWorldPrecompiled>());
}
```

- Compile source code. Please read here to install dependencies and compile source code.

Note: The implemented HelloWorldPrecompiled.cpp and header files should be placed under FISCO-BCOS/libprecompiled/extension directory.

- Build FISCO BCOS consortium blockchain Given that it is stored under `FISCO-BCOS/build` directory, use the following instruction to build chain for node 4. For more options please read here.

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

3 Calling

From the user's viewing, the pre-compiled contract is basically the same as the solidity contract. The only difference is the solidity contract can obtain the contracted address after deployment while the per-compiled contract can be used directly without deployment because of the pre-compiled contract address is pre-allocated.

3.1 Call HelloWorld precompiled contract using console

Create HelloWorldPrecompiled.sol file under console contracts/solidity with the content of interface declaration:

```
pragma solidity ^0.4.24;
contract HelloWorldPrecompiled{
    function get() public constant returns(string);
    function set(string n);
}
```

After the nodes are built by compiled binaries, deploy console v1.0.2 and above version and execute the following statement to call contract:

```
[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 Call solidity

Now, we try to create precompiled contract object and call its interface in Solidity contract. Create HelloWorldHelper.sol file in console contracts/solidity with the following content:

```
pragma solidity ^0.4.24;
import "./HelloWorldPrecompiled.sol";

contract HelloWorldHelper {
    HelloWorldPrecompiled hello;
    function HelloWorldHelper() {
        // call HelloWorld precompiled contract
        hello = HelloWorldPrecompiled(0x5001);
    }
    function get() public constant returns(string) {
        return hello.get();
    }
    function set(string m) {
        hello.set(m);
    }
}
```

Deploy HelloWorldHelper contract and call the interface of HelloWorldHelper contract, you will get the following result:

```
[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]>
```

6.7 Parallel transaction

FISCO BCOS provides development structure for parallelable contract. Contract developed under the structure regulation can be parallelly executed by nodes of FISCO BCOS. The advantages of parallel contract include:

- high TPS: multiple independent transaction being executed at the same time can utilize the CPU resources to the most extent and reach high TPS
- scalable: improve the performance of transaction execution with better configuration of machine to support expansion of applications

The following context will introduce how to compile, deploy and execute FISCO BCOS parallel contract.

6.7.1 Basic knowledge

Parallel exclusion

Whether two transactions can be executed in parallel depends on whether they are mutually **exclusive**. By exclusive, it means the two transactions have intersection in their contract storage variables collection.

Taking payment transfer as an example, it involves transactions of payment transfer between users. Use transfer(X, Y) to represent the access of user X to user Y. The exclusion is as below.

Here are detailed definitions:

- **exclusive parameter**: parameter that is related to "read/write" of contract storage variable in contract **interface**. Such as the interface of payment transfer transfer(X, Y), in which X and Y are exclusive parameters.
- **exclusive object**: the exclusive content extracted from exclusive parameters. Such as the payment transfer interface transfer(X, Y). In a transaction that calls the interface, the parameter is transfer(A, B), then the exclusive object is [A, B]; for another transaction that calls parameter transfer(A, C), the exclusive object is [A, C].

To judge whether 2 transactions at the same moment can be executed in parallel depends on whether there is intersection between their exclusive objects. Transaction without intersection can be executed in parallel.

6.7.2 Compile parallel contract

FISCO BCOS provides **parallel contract development structure**. Developers only need to adhere to its regulation and define the exclusive parameter of each contract interface so as to realize parallelly executed contract. When contract is deployed, FISCO BCOS will auto-analyze exclusive object before the transaction is excuted to make non-dependent transaction execute in parallel as much as possible.

So far, FISCO BCOS offers two types of parallel contract development structure: solidity and Precompiled contract.

Solidity development structure

Parallel solidity contract shares the same development process with regular solidity contract: make `ParallelContract` as the base class of the parallel contract and call `registerParallelFunction()` to register the interface.

Here is a complete example of how ParallelOk contract realize parallel payment transfer

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol";  // import ParallelContract.sol

contract ParallelOk is ParallelContract // make ParallelContract as the base class
{
    // contract realization
    mapping (string => uint256) _balance;

    function transfer(string from, string to, uint256 num) public
    {
        // here is a simple example, please use SafeMath instead of "+/-" in real␣
↪production
        _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];
    }

    // register parallel contract interface
    function enableParallel() public
    {
        // function defined character string (no blank space behind ","), the␣
↪former part of parameter constitutes exclusive parameter (which should be put␣
↪ahead when designing function)
        registerParallelFunction("transfer(string,string,uint256)", 2); //␣
↪critical: string string
        registerParallelFunction("set(string,uint256)", 1); // critical: string
    }

    // revoke parallel contract interface
    function disableParallel() public
    {
        unregisterParallelFunction("transfer(string,string,uint256)");
        unregisterParallelFunction("set(string,uint256)");
    }
}
```

The detail steps are:

（1）make `ParallelContract` as the base class of contract

```
pragma solidity ^0.4.25;

import "./ParallelContract.sol"; // import ParallelContract.sol

contract ParallelOk is ParallelContract // make ParallelContract as the base class
{
   // contract realization

   // register parallel contract interface
   function enableParallel() public;

   // revoke parallel contract interface
   function disableParallel() public;
}
```

（2）Compile parallel contract interface

Public function in contract is the interface of contract. To compile a parallel contract interface is to realize the public function of a contract according to certain rules.

Confirm whether the interface is parallelable

A parallelable contract interface has to meet following conditions:

- no call of external contract
- no call of other function interface

Confirm exclusive parameter

Before compiling interface, please confirm the exclusive parameter of interface. The exclusion of interface is the exclusion of global variables. The confirmation of exclusive parameter has following rules:

- the interface accessed global mapping, the key of mapping is the exclusive parameter
- the interface accessed global arrays, the subscript of a array is the exclusive parameter
- the interface accessed simple type of global variables, all the simple type global variables share one exclusive parameter and use different variable names as the exclusive objects.

Confirm parameter type and sequence

After the exclusive parameter is confirmed, confirm parameter type and sequence according to following rules:

- interface parameter is limited to: **string**、**address**、**uint256**、**int256** (more types coming in the future)
- exclusive parameter should all be contained in interface parameter
- all exclusive should be put in the beginning of the interface parameter

```
mapping (string => uint256) _balance; // global mapping

// exclusive variable from, to are put at the beginning of transfer()
function transfer(string from, string to, uint256 num) public
{
    _balance[from] -= num;  // from is the key of global mapping, the exclusive␣
↪parameter
    _balance[to] += num; // to is the key of global mapping, the exclusive␣
↪parameter
}

// the exclusive variable name is put at the beginning of the parameter of set()
function set(string name, uint256 num) public
{
    _balance[name] = num;
}
```

（3）Register parallelable contract interface

Implement enableParallel() function in contract, call registerParallelFunction() to register parallelable contract interface, and implement disableParallel() function to endow the contract with ability to revoke parallel execution.

```
// register parallelable contract interface
function enableParallel() public
{
    // function defined character string (no blank space behind ","), the␣
↪parameter starts with exclusive parameters
    registerParallelFunction("transfer(string,string,uint256)", 2); // transfer␣
↪interface, the former 2 is exclusive parameter
    registerParallelFunction("set(string,uint256)", 1); // transfer interface, the␣
↪first 1 is exclusive parameter
}

// revoke parallel contract interface
function disableParallel() public
{
    unregisterParallelFunction("transfer(string,string,uint256)");
    unregisterParallelFunction("set(string,uint256)");
}
```

（4）Deploy/execute parallel contract

Compile and deploy contract through Console or Web3SDK. Here we use console as an example.

deploy contract

```
[group:1]> deploy ParallelOk.sol
```

call `enableParallel()` interface to make ParallelOk executed parallelly

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744␣
↪enableParallel
```

send parallel transaction `set()`

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 set
↪"jimmyshi" 100000
```

send parallel transaction `transfer()`

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 transfer
↪"jimmyshi" "jinny" 80000
```

check transaction execution result `balanceOf()`

```
[group:1]> call ParallelOk.sol 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744␣
↪balanceOf "jinny"
80000
```

The following context contains an example to send massive transaction through SDK.

Precompile parallel contract structure

Parallel precompiled contract has the same compilation and development process with regular precompiled contract. Regular precompiled contract uses Precompile as the base class to implement contract logical. Based on this, Precompile base class offers 2 virtual functions for parallel to enable implementation of parallel precompiled contract.

（1）Define the contract as parallel contract

```
bool isParallelPrecompiled() override { return true; }
```

（2）Define parallel interface and exclusive parameter

It needs attention that once contract is defined parallelable, all interfaces need to be defined. If an interface is returned with null, it has no exclusive object. Exclusive parameter is related to the implementation of precompiled contract, which needs understanding of FISCO BCOS storage. You can read the codes or consult experienced programmer for implementation details.

```
// take out exclusive object from parallel interface parameter, return exclusive␣
↪object
std::vector<std::string> getParallelTag(bytesConstRef param) override
{
    // get the func and data to be called
    uint32_t func = getParamFunc(param);
    bytesConstRef data = getParamData(param);

    std::vector<std::string> results;
    if (func == name2Selector[DAG_TRANSFER_METHOD_TRS_STR2_UINT]) // function is␣
↪parallel interface
    {
        // interfaces: userTransfer(string,string,uint256)
        // take out exclusive object from data
        std::string fromUser, toUser;
        dev::u256 amount;
        abi.abiOut(data, fromUser, toUser, amount);

        if (!invalidUserName(fromUser) && !invalidUserName(toUser) && (amount > 0))
        {
            // write results to exclusive object
            results.push_back(fromUser);
```

```
            results.push_back(toUser);
        }
    }
    else if ... // all interfaces needs to offer exclusive object, returning null␣
↪means no exclusive object

        return results;  //return exclusion
}
```

（3）Compile, restart node

To manually compile nodes please check here

After compilation, close node and replace with the original node binaries, and restart node.

6.7.3 Example: parallel payment transfer

Here gives 2 parallel examples of solidity contract and precompiled contract.

Config environment

The execution environment in this case:

- Web3SDK client end
- a FISCO BCOS chain

Web3SDK is to send parallel transaction, FISCO BCOS chain is to execute parallel transaction. The related configuration are:

- Web3SDK configuration
- Chain building

For pressure test on maximum performance, it at least needs:

- 3 Web3SDKs to generate enough transactions
- 4 nodes, all Web3SDKs are configured with all information of nodes on chain to send transaction evenly to each node so that the chain can receive enough transaction

Parallel Solidity contract: ParallelOk

Payment transfer based on account model is a typical operation. ParallelOk contract is an example of account model and is capable of parallel transfer. The ParallelOk contract is given in former context.

FISCO BCOS has built-in ParallelOk contract in Web3SDK. Here is the operation method to send massive parallel transactions through Web3SDK.

（1）Deploy contract, create new user, activate parallel contract through SDK

```
# parameters: <groupID> add <quantity of users created> <TPS request of the␣
↪creation operation> <user information file name>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.parallelok.
↪PerformanceDT 1 add 10000 2500 user
# 10000 users has been created in group1, creation transactions are sent at␣
↪2500TPS, the generated user information is stored in user
```

After executed, ParallelOk contract will be deployed to blockchain, the created user information is stored in user file, and the parallel ability of ParallelOk contract is activated.

（2）Send parallel transfer transactions in batch

Note: before send transactions in batch, please adjust the SDK log level to `ERROR` to ensure capacity.

```
# parameter: <groupID> transfer <transaction volume> <TPS limit of the transfer␣
↪request> <user information file> <the exclusion percentage of transaction: 0~10>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.parallelok.
↪PerformanceDT 1 transfer 100000 4000 user 2

# 100000 transactions have been sent to group1, the TPS limit is 4000, users are␣
↪the same in the user file created formerly, 20% of exclusion exists between␣
↪transactions.
```

（3）Verify parallel correctness

After parallel transaction is executed, Web3SDK will print execution result. `TPS` is the TPS executed on node in the transaction sent by SDK. `validation` is the verification of transfer transaction result.

```
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
```

We can see that the TPS of this transaction is 2905. No error (`verify_failed count is 0`) after execution result is verified.

（4）Count total TPS

Single Web3SDK cannot send enough transactions to reach the parallel execution limit of nodes. It needs multiple Web3SDKs to send transactions at the same time. TPS by simply summing together won't be correct enough when multiple Web3SDKs sending transactions, so it should be acquired directly from node.

count TPS from log file using script

```
cd tools
sh get_tps.sh log/log_2019031821.00.log 21:26:24 21:26:59 # parameters: <log file>
↪<count start time> <count end time>
```

get TPS（2 SDK, 4 nodes, 8 cores, 16G memory）

```
statistic_end = 21:26:58.631195
statistic_start = 21:26:24.051715
total transactions = 193332, execute_time = 34580ms, tps = 5590 (tx/s)
```

Parallel precompiled contract: DagTransferPrecompiled

Same with the function of ParallelOk contract, FISCO BCOS has built-in example of parallel precompiled contract (DagTransferPrecompiled) and realizes transfer function based on account model. The contract can manage deposits from multiple users and provides a parallel transfer interface for parallel transactions of payment transfer between users.

Note: DagTransferPrecompiled is the example of parallel transaction with simple functions, please don't use it for online transactions.

（1）Create user

Use Web3SDK to send transaction to create user, the user information will be stored in user file. Command parameter is the same with parallelOk, the only difference is that the object called by the command is precompile.

```
# parameters: <groupID> add <created user quantity> <TPS requests in this operation>
↪ <user information file name>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 add 10000 2500 user
# 10000 users has been created in group1, creation transactions are sent at 2500␣
↪TPS, the generated user information is stored in user file
```

（2）Send parallel transfer transactions in batch

Send parallel transfer transactions through Web3SDK

Note: before sending transactions in batch, please adjust SDK log level to `ERROR` for enough capability to send transactions.

```
# parameters: <groupID> transfer <transaction volume> <TPS limit of the transfer␣
↪request> <user information file> <transaction exclusion percentage: 0~10>
java -cp conf/:lib/*:apps/* org.fisco.bcos.channel.test.parallel.precompile.
↪PerformanceDT 1 transfer 100000 4000 user 2
# 100000 transactions has been sent to group1, the TPS limit is 4000, users are␣
↪the same ones in the user file created formerly, 20% exclusion exists between␣
↪transactions.
```

（3）Verify parallel correctness

After parallel transactions are executed, Web3SDK will print execution result. `TPS` is the TPS of the transaction sent by SDK on the node. `validation` is the verification of transfer execution result.

```
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
```

We can see that in this transaction, the TPS is 3143. No error (`verify_failed count is 0`) after execution result verification.

（4）Count total TPS

Single Web3SDK can send enough transactions to meet the parallel execution limit of node. It needs multiple Web3SDK to send tranactions. And by simply summing the TPS of each transaction won't be correct, so the TPS should be acquired from node directly.

Count TPS from log file using script

```
cd tools
sh get_tps.sh log/log_2019031311.17.log 11:25 11:30 # parameter: <log file> <count␣
↪start time> <count end time>
```

get TPS (3 SDK, 4 nodes, 8 cores, 16G memory)

```
statistic_end = 11:29:59.587145
statistic_start = 11:25:00.642866
total transactions = 3340000, execute_time = 298945ms, tps = 11172 (tx/s)
```

Result description

The performance result in the example of this chapter is tested in 3SDK, 4 nodes, 8 cores, 16G memory, 1G network. Each SDK and node are deployed in different VPS with cloud disk. The real TPS depends on the condition of your hardware configuration, operation system and bandwidth.

6.8 Distributed storage

6.8.1 Install MySQL

The currently supported distributed database is MySQL. Before using distributed storage, you need to set up the MySQL service. The configuration on Ubuntu and CentOS servers is as follows:

Ubuntu: Execute the following three commands to configure the root account password during the installation process.

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

Start the MySQL service and log in: root account password.

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

CentOS: Perform the following two commands to install.

```
yum install mysql*
# some versions of linux need to install mariadb which is a branch of mysql
yum install mariadb*
```

Start the MySQL service. Log in and set a password for the root user.

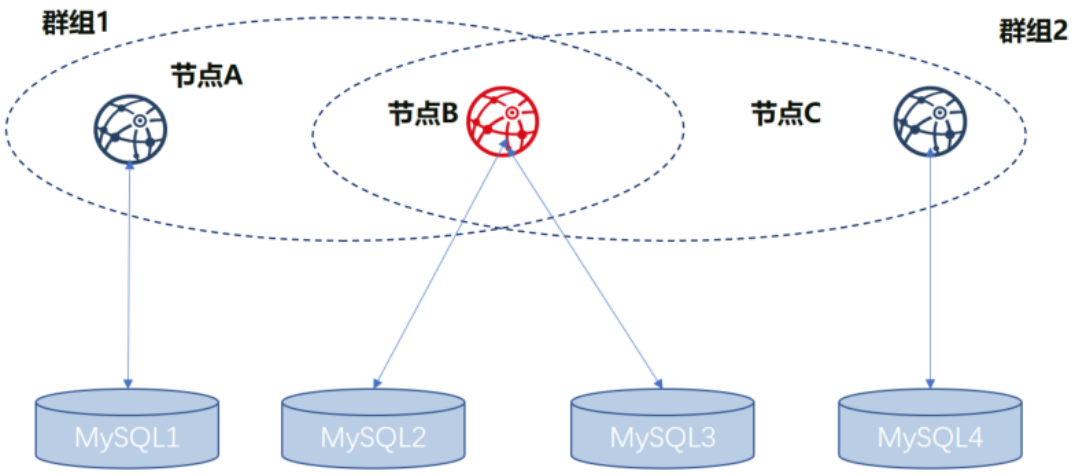
```
service mysqld start
#If mariadb is installed, to use the following command to start
service mariadb start
mysql -uroot
mysql> set password for root@localhost = password('123456');
```

6.8.2 Node directly connected to MySQL

FISCO-BCOS in version 2.0.0-rc3 supports nodes directly connected to MySQL through connection pool. Compared to the proxy access MySQL mode, this configuration is simple. No need to manually create a database. Please refer to the configuration method:

Logical architecture diagram

The multi-group architecture means that blockchain node supports launching multiple groups. The transaction processing, data storage, and block consensus among the groups are isolated from each other. Therefore, each node in the group corresponds to a database instance. For example, in blockchain network, there are three nodes A, B, and C, where A and B belong to Group1, and B and C belong to Group2. Nodes A and C correspond to one database instance respectively, and Node B corresponds to two database instances. The logical architecture diagram is as follows.



As shown in the above figure, NodeB belongs to multiple groups. The database instances which are corresponded by the same node in different groups are separate. For distinguishing the same node in different groups, the nodes of A, B, and C are respectively represented with Group1_A (NodeA in Group1, same as below), Group1_B, Group2_B, and Group2_C.

We use the above figure as an example to describe the setup configuration process in following.

Build node

Before using distributed storage, you need to complete the establishment of the alliance chain and the configuration of multiple groups. For details, refer to the following steps.

Prepare dependence

```
mkdir -p ~/fisco_direct && cd ~/fisco_direct
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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
```

Generate configuration file

```
# generate blockchain configuration file 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
```

```
# view configuration file
cat ipconf
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
```

Build blockchain with build_chain

```
### build blockchain (please confirm the ports of 30700~30702, 20700~20702, 8575~
↪8577 are not occupied)
### The difference right here is that the parameter "-s MySQL" is appended to the␣
↪command and the port is changed.
bash build_chain.sh -f ipconf -p 30700,20700,8575 -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
```

Modify node ini file

In group.[group].ini configuration file, the configuration information of MySQL is related to this feature. Suppose that the MySQL configuration information is as follows:

```
|node|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|
```

Modify the group.1.ini configuration in node0

Modify the content in the section ~/fisco_direct/nodes/127.0.0.1/node0/conf/group.1.ini[storage] and add the following content. Db_passwd is the corresponding password.

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

Modify the group.1.ini configuration in node1

Modify the content in the section ~/fisco_direct/nodes/127.0.0.1/node1/conf/group.1.ini[storage] and add the following content. Db_passwd is the corresponding password.

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

Modify the group.2.ini configuration in node1

Modify the content in the section ~/fisco_direct/nodes/127.0.0.1/node1/conf/group.2.ini[storage] and add the following content. Db_passwd is the corresponding password.

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

Modify the group.2.ini configuration in node2

Modify the content in the section ~/fisco_direct/nodes/127.0.0.1/node2/conf/group.2.ini[storage] and add the following content. Db_passwd is the corresponding password.

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

Start node

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

Check process

```
ps -ef|grep fisco-bcos|grep -v grep
fisco   111061      1  0 16:22 pts/0    00:00:04 /data/home/fisco_direct/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_direct/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_direct/nodes/127.
↪0.0.1/node1/../fisco-bcos -c config.ini
```

If it starts successfully, you can see there are 3 fisco-bcos processes. If it fails, please refer to the log to confirm whether the configuration is correct.

Check output of log

Execute the following command to view the number of nodes connected to node0 (similar to other nodes)

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

Normally, you will see an output similar to the following, and you can see that node0 is connecting to the other two nodes from it.

```
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
```

Execute the following command to check if it is in consensus

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

Normally, the output will continue to output ++++Generating seal to indicate that the consensus is normal.

```
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...
```

Send transaction by console

Prepare dependence

```
cd ~/fisco_direct;
bash <(curl -s https://raw.githubusercontent.com/FISCO-BCOS/console/master/tools/
↪download_console.sh)
cp -n console/conf/applicationContext-sample.xml console/conf/applicationContext.
↪xml
cp nodes/127.0.0.1/sdk/* console/conf/
```

Modify configuration file

Modify ~/fisco_direct/console/conf/applicationContext.xml to the following configuration (partial information)

```
<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>127.0.0.1:20700</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>
```

Start console

```
cd ~/fisco_direct/console
sh start.sh 1
#deploy TableTest contract
```

```
[group:1]> deploy TableTest
contract address:0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744
```

view the table in the database

```
MySQL -uroot -p123456 -A db_Group1_A
use db_Group1_A;
show tables;
+-----------------------------------------------------------+
| Tables_in_db_Group1_A                                     |
+-----------------------------------------------------------+
| _contract_data_8c17cf316c1063ab6c89df875e96c9f0f5b2f744_ |
| _contract_data_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)
```

call the create interface in the console

```
#create table
call TableTest 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 create
0xab1160f0c8db2742f8bdb41d1d76d7c4e2caf63b6fdcc1bbfc69540a38794429
```

view the table in the database

```
show tables;
+-----------------------------------------------------------+
| Tables_in_db_Group1_A                                     |
+-----------------------------------------------------------+
| _contract_data_8c17cf316c1063ab6c89df875e96c9f0f5b2f744_ |
| _contract_data_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_                                     |
| _user_t_test                                              |
+-----------------------------------------------------------+
```

Inserting a record to the database

```
#insert data into the table
call TableTest 0x8c17cf316c1063ab6c89df875e96c9f0f5b2f744 insert "fruit" 100 "apple
↪"
0x082ca6a5a292f1f7b20abeb3fb03f45e0c6f48b5a79cc65d1246bfe57be358d1
```

open the MySQL client and query the_user_t_test table data

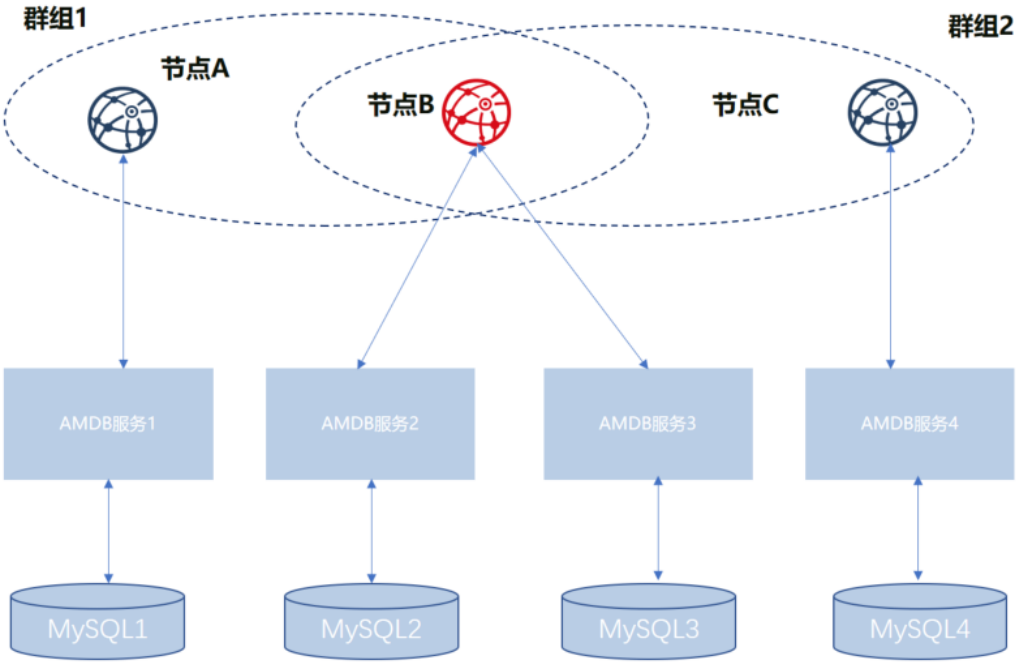
```
#view data in the user table
select * from _user_t_test\G;
*************************** 1. row ***************************
     _id_: 31
   _hash_: 0a0ed3b2b0a227a6276114863ef3e8aa34f44e31567a5909d1da0aece31e575e
    _num_: 3
 _status_: 0
     name: fruit
  item_id: 100
item_name: apple
1 row in set (0.00 sec)
```

6.8.3 Access MySQL through proxy

This operation tutorial is only valid for the 2.0.0-rc3 node version. If you need to build a distributed storage environment in 2.0.0-rc2 through "accessing MySQL through proxy", please refer to the document Distributed Storage Building Method

Logical architecture diagram

Multi-group architecture means that blockchain node supports starting multiple groups, and the transaction processing, data storage, and block consensus among the groups are isolated from each other. Therefore, each node in the group corresponds to an AMDB instance. For example, in blockchain network, there are three nodes A, B, and C, where A and B belong to group1, and B and C belong to group2. NodeA and NodeC correspond to one database instance respectively, and NodeB corresponds to two database instances. The logical architecture diagram is as follows:



As shown in the above figure, NodeB belongs to multiple groups. The AMDB server and MySQL which are corresponded by the same node in different groups are separate. For distinguishing the same node in different groups, the nodes of A, B, and C are respectively represented with Group1_A (NodeA in Group1, same as below), Group1_B, Group2_B, and Group2_C.

We use the above figure as an example to describe the setup configuration process in following.

Build nodes

Before configuring the AMDB service, you need to complete the establishment of alliance chain and the configuration of multiple groups. For details, refer to the following steps.

Prepare dependence

- create folder

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

- get `build_chain` script

```
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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
```

Generate configuration file

```
# generate blockchain configuration file 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

# view configuration file
cat ipconf
127.0.0.1:1 agencyA 1
127.0.0.1:1 agencyB 1,2
127.0.0.1:1 agencyC 2
```

Build blockchain with build_chain

```
### build blockchain (please confirm the ports of 30600~30602, 20800~20802, 8565~
↪8567 are not occupied)

bash build_chain.sh -f ipconf -p 30600,20800,8565
================================================================
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
```

Modify node ini file

Modify the group.1.ini configuration in node0

Modify the contents of the [storage] section in the ~/fisco/nodes/127.0.0.1/node0/conf/group.1.ini file, and set as following

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

Modify the group.1.ini configuration in node1

Modify the contents of the [storage] section in the ~/fisco/nodes/127.0.0.1/node1/conf/group.1.ini file, and set as following

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

Modify the group.2.ini configuration in node1

Modify the contents of the [storage] section in the ~/fisco/nodes/127.0.0.1/node1/conf/group.2.ini file, and set as following

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

Modify the group.2.ini configuration in node2

Modify the contents of the [storage] section in the ~/fisco/nodes/127.0.0.1/node2/conf/group.2.ini file, and set as following

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

Prepare amdb proxy

Get source code

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

Compile source code

```
cd AMDB;gradle build
```

After the compilation is completed, a dist directory is generated, and the file structure is as follows:

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

Configure amdb proxy

In the connection process between AMDB and node, AMDB is client and node is server. During the startup process, AMDB service is active to connect to node, and node only needs to configure topic of AMDB attention. For topic introduction, please refer to AMOP. AMDB needs to pass by the certificate access.

Certificate configuration

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

amdb instance copy

```
cd ~/fisco;
###dist_Group1_A is the amdb instance corresponding to the node Group1_A
cp AMDB/dist/ dist_Group1_A -R
###dist_Group1_B is the amdb instance corresponding to the node Group1_B
cp AMDB/dist/ dist_Group1_B -R
###dist_Group2_B is the amdb instance corresponding to the node Group2_B
cp AMDB/dist/ dist_Group2_B -R
###dist_Group2_C is the amdb instance corresponding to the node Group2_C
cp AMDB/dist/ dist_Group2_C -R
```

After the above steps, you can see the file structure of ~/fisco directory as follows:

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

DB creation

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

File configuration

amdb.properties configures the node information that AMDB service needs to connect to, and db.properties configures the connection information of database. Here we assume that the MySQL configuration information is as follows:

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

Configure amdb proxy for Group1's NodeA

configure ~/fisco/dist_Group1_A/conf/amdb.properties as following content:

```
node.ip=127.0.0.1
node.listen_port=20800
node.topic=DB_Group1_A
```

configure ~/fisco/dist_Group1_A/conf/db.properties as following content:

```
db.ip=127.0.0.1
db.port=3306
db.user=root
db.password=123456
db.database=bcos_Group1_A
```

modify ~/fisco/dist_Group1_A/conf/applicationContext.xml to the following configuration (partial information)

```
<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>127.0.0.1:20800</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

<bean id="DBChannelService" class="org.fisco.bcos.channel.client.Service">
                <property name="groupId" value="1" />
                <property name="orgID" value="fisco" />
```

```
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
                <property name="topics">
                        <list>
                                <value>${node.topic}</value>
                        </list>
                </property>
        <property name="pushCallback" ref="DBHandler"/>
</bean>
```

Configure amdb proxy for Group1's NodeB

configure ~/fisco/dist_Group1_B/conf/amdb.properties as following content:

```
node.ip=127.0.0.1
node.listen_port=20801
node.topic=DB_Group1_B
```

configure ~/fisco/dist_Group1_B/conf/db.properties as following content:

```
db.ip=127.0.0.1
db.port=3306
db.user=root
db.password=123456
db.database=bcos_Group1_B
```

modify ~/fisco/dist_Group1_B/conf/applicationContext.xml to the following configuration (partial information)

```
<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>127.0.0.1:20801</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

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

Configure amdb proxy for Group2's NodeB

configure ~/fisco/dist_Group2_B/conf/amdb.properties as following content:

```
node.ip=127.0.0.1
node.listen_port=20801
node.topic=DB_Group2_B
```

configure ~/fisco/dist_Group2_B/conf/db.properties as following content:

```
db.ip=127.0.0.1
db.port=3306
db.user=root
db.password=123456
db.database=bcos_Group2_B
```

modify ~/fisco/dist_Group2_B/conf/applicationContext.xml to the following configuration (partial information)

```
<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>127.0.0.1:20801</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

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

                <!-- communication topic configuration of the node -->
                <property name="topics">
                        <list>
                                <value>${node.topic}</value>
                        </list>
                </property>
                <property name="pushCallback" ref="DBHandler"/>
        </bean>
```

Configure amdb proxy for Group2's NodeC

configure ~/fisco/dist_Group2_C/conf/amdb.properties as following content:

```
node.ip=127.0.0.1
node.listen_port=20802
node.topic=DB_Group2_C
```

configure ~/fisco/dist_Group2_C/conf/db.properties as following content:

```
db.ip=127.0.0.1
db.port=3306
db.user=root
db.password=123456
db.database=bcos_Group2_C
```

modify ~/fisco/dist_Group2_C/conf/applicationContext.xml to the following configuration (partial information)

```
<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>127.0.0.1:20802</value>
                                        </list>
                                </property>
                        </bean>
                </list>
        </property>
</bean>

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

        <!-- communication topic configuration of the node -->
        <property name="topics">
                        <list>
                                <value>${node.topic}</value>
                        </list>
                </property>
        <property name="pushCallback" ref="DBHandler"/>
</bean>
```

Start amdb proxy

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

Start nodes

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

Check process

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

```
fisco   110803      1  1 17:25 ?        00:00:10 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main
fisco   122676      1 16 17:38 ?        00:00:08 java -cp conf/:apps/*:lib/* org.
↪bcos.amdb.server.Main

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

If it starts successfully, you can see there are 4 jave processes and 3 fisco-bcos processes. If it fails, please refer to the log to confirm whether the configuration is correct.

Check the output of log

Execute the following command to view the number of nodes connected to node0 (other nodes are similar)

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

Normally, you will see an output similar to the following, and you can see that node0 is connecting to the other two nodes from it.

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

Execute the following command to check if it is in consensus

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

Normally, the output will continue to output ++++Generating seal to indicate that the consensus is normal.

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

Send transaction by console

Please refer to the section Sending Transaction by Console in "Node Direct Connection to MySQL".

6.9 Node access

For ensuring the security of the blockchain, FISCO BCOS introduces free nodes, observer nodes and consensus nodes which can be converted to each other by the console.

6.9.1 Operation command

The console provides three commands of **AddSealer**, **AddObserver**, and **RemoveNode** to convert the specified node to Sealer, Oberserver, and RemoveNode, and can use **getSealerList**, **getObserverList**, and **getNodeIDList** to view the current list of Sealer, Observer, and other nodes.

- addSealer: to set the corresponding node as the Sealer according to the NodeID;
- addObserver: to set the corresponding node as the Observer according to the NodeID;
- removeNode: to set the corresponding node as the RemoveNode according to the NodeID;
- getSealerList: to view the Sealer in the group;
- getObserverList: to view the Observer in the group;
- getNodeIDList: to view the NodeID in the group;

For example, to convert the specified nodes to Sealer, Observer, and RemoveNode, the main operation commands are as follows:

Important: Before accessing the node, please ensure that:

- Node ID exists and can execute cat that is getting from conf/node.nodeid in the node directory
- All Sealers are normal, and they will output +++ logs.

```
# to set the node in the ~/fisco/nodes/192.168.0.1/node0 directory
$ mkdir -p ~/fisco && cd ~/fisco

# to get Node ID (to set the directory as ~/nodes/192.168.0.1/node0/)
$ cat ~/fisco/nodes/192.168.0.1/node0/conf/node.nodeid
7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b9695

# to connect console (to set the console in the ~/fisco/console directory)
$ cd ~/fisco/console

$ bash start.sh

# to convert the specified node to Sealer
[group:1]> addSealer␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969
# to view the list of Sealer
[group:1]> getSealerList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff2
]

# to convert the specified node to Observer
[group:1]> addObserver␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969

# to view the list of Observer
[group:1]> getObserverList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff2
]

# to convert the specified node to removeNode
[group:1]> removeNode␣
↪7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff220b969

# to view the list of NodeID
[group:1]> getNodeIDList
[
        7a056eb611a43bae685efd86d4841bc65aefafbf20d8c8f6028031d67af27c36c5767c9c79cff201769ed80ff2
]
[group:1]> getSealerList
[]
```

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

6.9.2 Operation cases

The following describes the operations of group expansion and node exit in detail with specific operation cases. Group expansion is divided into two phases, namely **adding nodes to the network** and **adding nodes to the group**. Also, node exit is divided into two phases, namely **exiting node from the group** and **exiting node from the network**.

Operation methods

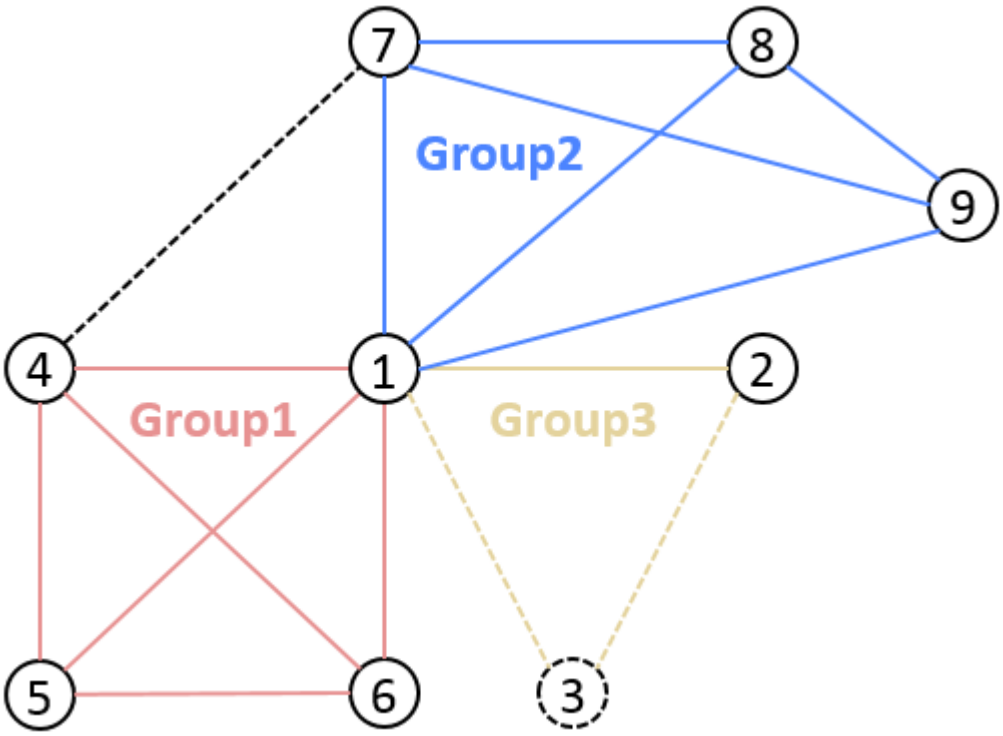
- Node configuration modification: After the node modifies its own configuration, it need to be restarted to takes effect. The involved operations include **network adding/exit and CA blacklist inclusion/removal**.
- Transaction uploading to the chain: To modify the transaction, the configuration of group consensus is needed, and the operation involves **the modification of node type**. The current sending transaction path is the pre-compiled service interface provided by the console and SDK.
- RPC inquiry： To use the command curl to inquire the information on the chain, and the operation involves **the query of the group node**.

Operation steps

In this section, the following figure is taken as an example to describe the operations of expansion and network exit show above.

The dotted line indicates that network communication can be performed among nodes, and the solid line indicates that nodes have group relationships base on the communication among nodes, and colors' difference distinguish different group relationships.

The below figure below shows a network with three groups of which Group3 has three nodes. Whether Group3 has intersection nodes with other groups does not affect the versatility of the following operation process.



Here we take the following node information of Group 3 as an example:

The folder name of node 1 is `node0`, IP port 127.0.0.1:30400, the former 4 bytes of nodeID b231b309...

The folder name of node 2 is `node1`, IP port 127.0.0.1:30401, the former 4 bytes of nodeID aab37e73...

The folder name of node 3 is `node2`, IP port 127.0.0.1:30402, the former 4 bytes of nodeID d6b01a96...

Node A to join the network

Background:

Node 3 is outside the network and wants to join it now.

Operation steps:

1 . enter nodes folder and execute `gen_node_cert.sh` to generate node folder. Here we name the folder as node2, which contains `conf/` folder;

```
# acquire script
$ curl -LO https://raw.githubusercontent.com/FISCO-BCOS/FISCO-BCOS/master/tools/
↪gen_node_cert.sh && chmod u+x gen_node_cert.sh
# execute, -c is the ca route given when the node was generated, agency is the␣
↪agency name, -o is the name of the node folder to be generated
$ ./gen_node_cert.sh -c nodes/cert/agency -o node2
```

2 . copy node 2 under `nodes/127.0.0.1/`, parallel with other node folder (`node0`, `node1`);

```
$ cp -r ./node2/ nodes/127.0.0.1/
```

3 . enter `nodes/127.0.0.1/`, copy `node0/config.ini`, `node0/start.sh` and `node0/stop.sh` to node2 directory;

```
$ cd nodes/127.0.0.1/
$ cp node0/config.ini node0/start.sh node0/stop.sh node2/
```

4 . modify `node2/config.ini`. For `[rpc]` model, modify `listen_ip`, `channel_listen_port` and `jsonrpc_listen_port`; for `[p2p]` model, modify `listen_port` and add its node information in `node`..

```
$ 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 . node 3 copies `node1/conf/group.3.genesis`(which contains **initial list of group nodes**) and `node1/conf/group.3.ini` to `node2/conf` folder, without modification;

```
$ cp node1/conf/group.3.genesis node2/
$ cp node1/conf/group.3.ini node2/
```

6 . execute `node2/start.sh` and start node 3;

```
$ ./node2/start.sh
```

7 . confirm that node 3 is connected with node 1, 2, then it has joined the network now.

```
# Open the DEBUG log to check the number and ID of nodes connected with node 2
# The following log information indicates that node 2 is connected with 2␣
↪nodes(with the former 4 bytes being b231b309 and aab37e73)
$ 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
```

Note:

- The other configurations of config.ini copied from node 1 remain the same;
- Theoretically, node 1, 2 can accomplish the extension of node 3 without changing their p2p connecting nodes list;
- The group to be chosen in step 5 are recommended to be the group to be joined by node 3;
- To keep in full connection status, we recommend users to add the information of node 3 to the p2p connecting nodes list in config.ini of node 1, 2, and restart node 1, 2.

Node A to quit the network

Background:

Node 3 is in the network and connected with node 1, 2. Now node 3 needs to quit the network.

Operation steps:

1 . For node 3, clear up the **P2P connecting nodes list**, and restart node 3;

```
# execute under node 2
$ ./stop.sh
$ ./start.sh
nohup: appending output to 'nohup.out'
```

2 . For node 1, 2, remove node 3 form their **P2P connecting nodes list**(if has), and restart node 1, 2;

3 . Confirm that node 3 has been disconnected with node 1, 2, and it has quitted the network now.

Note:

- **node 3 has to quit the group before quitting the network, which is guaranteed by users and will not be verified by the system**;
- the networking process is started by nodes. If missing step 2, node 3 can still get the p2p connecting request from node 1, 2 and start connection. It can be stopped by using CA blacklist.

Node A to join a group

Background:

Group 3 contains node 1, 2, either generates block in turn. Now node 3 wants to join the group.

Operation steps:

1. Node 3 joins the network;
2. Set node 3 as the **consensus node** using console addSealer according to the node ID;
3. check if the node ID of node 3 is included in the consensus nodes of group 3 through console getSealerList. If is, then it has joined the group successfully.

Note:

- node ID of node 3 can be acquired through *cat nodes/127.0.0.1/node2/conf/node.nodeid*;
- the first start of node 3 will write the configured group node initial list to the node system list, when the blocks stop synchronizing, **the node system lists of each node are the same**;
- **node 3 needs to have access to the network before joining the group, which will be verified by the system**;
- **the group fixed configuration file of node 3 should be the same with node 1, 2**.

Node A to quit the group

Background:

Group 3 contains node 1, 2, 3, either of which generates block in turn. Now node 3 wants to quit the group.

Operation steps:

1. set node 3 as **free node** according to its ID using console removeNode;
2. check if the node ID of node 3 is included in the consensus nodes of group 3 through console getSealerList. If not, then node 3 has quited the group.

Additional:

Note:

- node 3 can quit the group as a consensus node or observer node.

6.10 Permission control

This section will introduce the operations concerning permission control, for details please check Design of Permission Control.

For the system is defaulted with no permission setting record, any account can perform permission setting. For example, account 1 gives permission of contract deployment to itself, but account 2 also sets itself with permission of contract deployment. So the setting of account 1 becomes meaningless for every other node can add permissions freely. Therefore, before building consortium chain, confirming permission setting rules is needed. We can use *grantPermissionManager* instruction to set manager account, that is to give some account access to permission setting, which other accounts don't have.

6.10.1 Operations

The operations concerning permission control of following functions are introduced in this section:

- Permission of chain manager
- Permission of system manager
  - Permission of contract deployment and user table creation
  - Permission of Contract deployment using CNS
  - Permission of node management
  - Permission to modify system parameter
- Permission to write user table

6.10.2 Environment configuration

Configure and start the nodes and console of FISCO BCOS 2.0. For reference please check Installation.

6.10.3 Tools for permission control

FISCO BCOS offers permission control of console commands (developers can call PermissionService API of SDK API for permission control). The involved permission control commands are:

6.10.4 Permission control example

Console provides script `get_account.sh` to generate accounts. The account files will be stored in `accounts` folder. Console can set active accounts. The operation method is introduced in Console tutorial. Therefore, through console we can set account to experience permission control. For account safety, we will generate 3 PKCS12 account files under the root folder of console by `get_account.sh` script. Please remember the password during generation. The 3 PKCS12 account files are:

```
# account 1
0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252.p12
# account 2
0x7fc8335fec9da5f84e60236029bb4a64a469a021.p12
# account 3
0xd86572ad4c92d4598852e2f34720a865dd4fc3dd.p12
```

Now we can open 3 Linux terminal and log in console with the 3 accounts separately.

Log in with account 1:

```
$ ./start.sh 1 -p12 accounts/0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252.p12
```

Log in with account 2:

```
$ ./start.sh 1 -p12 accounts/0x7fc8335fec9da5f84e60236029bb4a64a469a021.p12
```

Log in with account 3:

```
$ ./start.sh 1 -p12 accounts/0xd86572ad4c92d4598852e2f34720a865dd4fc3dd.p12
```

6.10.5 Grant permission of chain manager

The 3 accounts play 3 kinds of roles. Account 1 performs chain manager, account 2 performs system manager and account 3 the regular account. Chain manager has permission for access control, namely granting permissions. System manager can manager permissions related to system functions, each of which should be granted independently, including deploying contract, creating user table, managing nodes and deploying contract with CNS and modifying system parameter. Chain manager can grant other accounts to be chain manager or system manager, or grant regular accounts to write table list.

Initial status of chain contains no permission records. Now, we can enter the console of account 1 and set itself as the chain manager, so other accounts are regular accounts.

```
[group:1]> grantPermissionManager 0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252
{
    "code":0,
    "msg":"success"
}

[group:1]> listPermissionManager
------------------------------------------------------------------------------------------
↪----------
|                    address                     |                 enable_num
↪          |
| 0x2c7f31d22974d5b1b2d6d5c359e81e91ee656252  |                      1
↪          |
------------------------------------------------------------------------------------------
↪----------
```

Account 1 is set as the chain manager.

6.10.6 Grant permission of system manager

Grant permission to deploy contract and create user table

Account 1 grants permission of system manager to account 2. At first, grant account 2 with permission to deploy contract and create user table.

```
[group:1]> grantDeployAndCreateManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listDeployAndCreateManager
-------------------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                      2
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

Log in console with account 2 and deploy TableTest contract offered by console. Code of TableTest.sol contract is here. The CRUD operations of user table t_test are also provided.

```
[group:1]> deploy TableTest.sol
contract address:0xfe649f510e0ca41f716e7935caee74db993e9de8
```

Call create API of TableTest to create user table t_test.

```
[group:1]> call TableTest.sol 0xfe649f510e0ca41f716e7935caee74db993e9de8 create
transaction hash:0x67ef80cf04d24c488d5f25cc3dc7681035defc82d07ad983fbac820d7db31b5b
-------------------------------------------------------------------------------------------
↪----------
Event logs
-------------------------------------------------------------------------------------------
↪----------
createResult index: 0
count = 0
-------------------------------------------------------------------------------------------
↪----------
```

User table t_test is created successfully.

Log in console with account 3 and deploy TableTest contract.

```
[group:1]> deploy TableTest.sol
{
    "code":-50000,
    "msg":"permission denied"
}
```

Account 3 fails to deploy contract as it has no permission.

- **Note:** deploying contract and creating user table are "2-in-1" control items. When using CRUD contracts, we suggest to create the needed tables (creating tables in building function of contract) when deploying contract, otherwise "table-missing" error may occur when reading or writing table. If it is needed to dynamically create table, the permission should be granted to minority accounts, otherwise there will be many invalid tables on blockchain.

Grant permission to deploy contract using CNS

Console provides 3 commands involving CNS:

Note: permission of deployByCNS command is controllable **and needs permission to deploy contract and use CNS at the same time**, permissions of callByCNS and queryCNS commands are not controllable.

Log in console with account 1, grant account 2 with permission to deploy contract using CNS.

```
[group:1]> grantCNSManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listCNSManager
-------------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                    13
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

Log in console with account 2, deploy contract using CNS

```
[group:1]> deployByCNS TableTest.sol 1.0
contract address:0x24f902ff362a01335db94b693edc769ba6226ff7

[group:1]> queryCNS TableTest.sol
-------------------------------------------------------------------------------------------
↪----------
|                   version                    |                   address
↪          |
|                     1.0                      |
↪0x24f902ff362a01335db94b693edc769ba6226ff7  |
-------------------------------------------------------------------------------------------
↪----------
```

Log in console with account 3, deploy contract using CNS

```
[group:1]> deployByCNS TableTest.sol 2.0
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> queryCNS TableTest.sol
-------------------------------------------------------------------------------------------
↪----------
|                   version                    |                   address
↪          |
|                     1.0                      |
↪0x24f902ff362a01335db94b693edc769ba6226ff7  |
-------------------------------------------------------------------------------------------
↪----------
```

Account 3 fails to deploy contract by CNS due to lack of permission

Grant permission to manage nodes

Console provides 5 commands related to node type management:

- **Note:** permissions of addSealer, addObserver and removeNode commands are controllable, while permissions of getSealerList and getObserverList commands are not.

Log in console with account 1, grant account 2 with permission to manage nodes.

```
[group:1]> grantNodeManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
```

```
    "code":0,
    "msg":"success"
}

[group:1]> listNodeManager
--------------------------------------------------------------------------------------
↪----------
|                    address                    |                 enable_num
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                    20
↪          |
--------------------------------------------------------------------------------------
↪----------
```

Log in console with account 2, view consensus node list.

```
[group:1]> getSealerList
[
    "01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
↪
    "279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    "320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    "c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]
```

View observer node list:

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

Set the first node ID as the observer node:

```
[group:1]> addObserver "01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
{
    "code":0,
    "msg":"success"
}

[group:1]> getObserverList
[
    "01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
]

[group:1]> getSealerList
[
    "279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    "320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    "c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]
```

Log in console with account 3, add observer node to consensus node list.

```
[group:1]> addSealer␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> getSealerList
[
    ␣
↪279c4adfd1e51e15e7fbd3fca37407db84bd60a6dd36813708479f31646b7480d776b84df5fea2f3157da6df9cad078
↪
    ␣
↪320b8f3c485c42d2bfd88bb6bb62504a9433c13d377d69e9901242f76abe2eae3c1ca053d35026160d86db1a563ab2a
↪
    ␣
↪c26dc878c4ff109f81915accaa056ba206893145a7125d17dc534c0ec41c6a10f33790ff38855df008aeca3a27ae7d9
]

[group:1]> getObserverList
[
    ␣
↪01cd46feef2bb385bf03d1743c1d1a52753129cf092392acb9e941d1a4e0f499fdf6559dfcd4dbf2b3ca418caa09d95
]
```

Account 3 fails to add consensus node for lack of permission to manage nodes. Now only account 2 has permission to add observer node to consensus node list.

Grant permission to modify system parameter

Console provides 2 commands about system parameter modification:

- **Note:** currently we support system parameter setting with key tx_count_limit or tx_gas_limit. Permission of setSystemConfigByKey command is controllable, while permission of getSystemConfigByKey command is not.

Log in console with account 1, grant account 2 with permission to modify system parameter.

```
[group:1]> grantSysConfigManager 0x7fc8335fec9da5f84e60236029bb4a64a469a021
{
    "code":0,
    "msg":"success"
}

[group:1]> listSysConfigManager
---------------------------------------------------------------------------------
↪----------
|                   address                     |              enable_num       ␣
↪          |
| 0x7fc8335fec9da5f84e60236029bb4a64a469a021  |                  23             ␣
↪          |
---------------------------------------------------------------------------------
↪----------
```

Log in console with account 2, modify the value of system parameter tx_count_limit to 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
```

Log in console with account 3, modify value of parameter tx_count_limit to 3000.

```
[group:1]> setSystemConfigByKey tx_count_limit 3000
{
    "code":-50000,
    "msg":"permission denied"
}

[group:1]> getSystemConfigByKey tx_count_limit
2000
```

Account 3 fails to set parameter due to no permission.

6.10.7 Grant permission to write user table

Account 1 can grant account 3 with permission to write user table t_test.

```
[group:1]> grantUserTableManager t_test 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd
{
    "code":0,
    "msg":"success"
}
[group:1]> listUserTableManager t_test
-------------------------------------------------------------------------------------------
↪----------
|                   address                    |                 enable_num
↪          |
| 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd  |                      6
↪          |
-------------------------------------------------------------------------------------------
↪----------
```

Log in console with account 3, insert a record in user table t_test and inquire the records.

```
[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]]
```

Log in console with account 2, update the record inserted by account 3 and inquire the records.

```
[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]]
```

Account 2 fails to update information for it has no permission to write user table t_test.

- Account 1 revoke permission of account 3 to write user table t_test.

```
[group:1]> revokeUserTableManager t_test 0xd86572ad4c92d4598852e2f34720a865dd4fc3dd
{
    "code":0,
    "msg":"success"
}

[group:1]> listUserTableManager t_test
Empty set.
```

Revoked successfully.

- **Note:** now there is no account with permission to write user table t_test, so it is back to initial status, that is, all accounts have permission to write table. Therefore, account 1 can grant another account, like account 2, with permission to write this table.

6.11 CA blacklist

This document describes the practical operation of CA blacklist. It is recommended that you read this document before you realize [CA Blacklist Introduction].(../design/security_control/certificate_blacklist.md).

The operations of the CA blacklist include **a node list to/remove from the CA blacklist**, and are implemented through modifying the configuration file and restarting.

6.11.1 Revised scope

The node configuration `config.ini` has `[certificate_blacklist]` path (optional). `[certificate_blacklist]` is the list of NodeID. node.X is the other node NodeID that the own node refuses to connect.

6.11.2 Revised example

There are three nodes which are interconnected in the network. The related information of the nodes is:

The directory of node 1 is node0, IP port is 127.0.0.1:30400, and the first four bytes of nodeID are b231b309...

The directory of node 2 is node1, IP port is 127.0.0.1:30401, and the first four bytes of nodeID are aab37e73...

The directory of node 3 is node2, IP port is 127.0.0.1:30402, and the first four bytes of nodeID are d6b01a96...

Node A puts Node B in the CA blacklist

scenario description:

Node 1 and node 2 are in the same group. Other nodes in the group and them are taking turn to generate block. Now node 1 puts Node 2 to its own blacklist.

operation steps:

1. For node0, to put the public key nodeID of node1 to its own **CA blacklist**;

```
$ cat node1/conf/node.nodeid
aab37e73489bbd277aa848a99229ab70b6d6d4e1b81a715a22608a62f0f5d4270d7dd887394e78bd02d9f31b8d366ce490
$ vim node0/config.ini
;certificate blacklist
[certificate_blacklist]
    ;crl.0 should be nodeid, nodeid's length is 128
    crl.
↪0=aab37e73489bbd277aa848a99229ab70b6d6d4e1b81a715a22608a62f0f5d4270d7dd887394e78bd02d9f31b8d366
```

1. Restart node 1;

```
# Execute in the node1 directory
$ ./stop.sh
$ ./start.sh
nohup: appending output to 'nohup.out'
```

1. Confirm that node 1 and node 2 are no longer connected by the log. The operation of putting to blacklist is completed.

```
# Under the premise of opening the DEBUG level log, to check the number of nodes␣
↪connected to the own node (node2) and the connected node information (nodeID).

# The following log indicates that the node2 is connected with two nodes (the␣
↪first 4 bytes of nodeID are b231b309 and aab37e73).
$ 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
```

Additional instructions:

- Node 0 adds node 1 to its CA blacklist, node 0 will disconnect the connection and AMOP communication with node 1;

Node A revokes Node B from CA blacklist

scenario description:

Node 1 has the nodeID of node 2 in its own CA blacklist, and node 2 has not the nodeID of node 1 in its own CA blacklist. Now node 1 revoke node 2 from its own CA blacklist.

operation steps:

1. For node 1, to revoke the public key nodeID of node 2 to its own **CA blacklist**;
2. Restart node 1;
3. Confirm that node 1 and node 2 re-establish the connection by the log. The operation of revoking from blacklist is completed.

6.12 AMOP

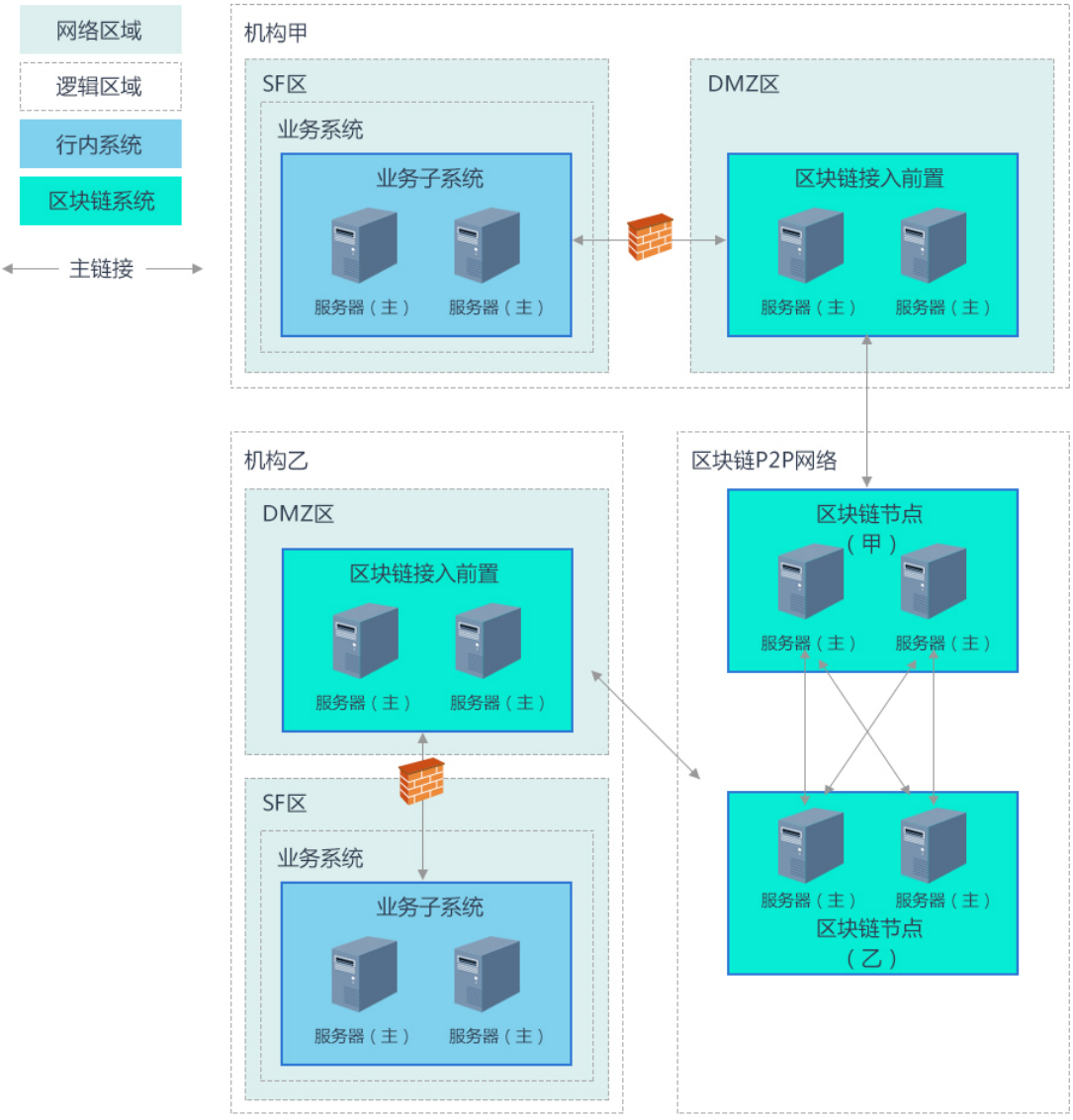
6.12.1 Introduction

Advanced Messages Onchain Protocol (AMOP) aims to provide a secure and efficient message channel for alliance chain. Each agency in the alliance chain can use AMOP to communicate as long as they deploy blockchain nodes,

whether they are Sealer or Observer. AMOP has the following advantages:

- Real-time: AMOP messages do not rely on blockchain transactions and consensus. Messages are transmitted in real time among nodes with a milliseconds delay.
- Reliability: When AMOP message is transmitting, it can automatically search all feasible link in blockchain network for communication. As long as at least one link is available, the message is guaranteed to be reachable.
- Efficiency: The AMOP message has simple structure and efficient processing logic. AMOP message's simple structure and efficient processing logic makes it to fully utilize network bandwidth and require a small amount of CPU usage only.
- Security: All communication links of AMOP use SSL encryption. The encryption algorithm is configurable.
- Easy to use: No additional configuration is required in SDK when using AMOP.

6.12.2 Logical architecture



We take the typical IDC architecture of the bank as the example to overview each region:

- SF region: The business service region inside the agency. The business subsystem in this region uses blockchain SDK. If there is no DMZ region, the configured SDK connects to the blockchain node, otherwise SDK connects to the blockchain front of DMZ region.
- DMZ region: The external network isolation region inside the agency. It is not required, if any, the region is deployed in front of blockchain.
- Blockchain P2P network: This is a logical region and deployed blockchain nodes of each agency. Blockchain nodes can also be deployed inside the agency.

6.12.3 Configuration

AMOP does not require any additional configuration. The following is a configuration case for [Web3SDK] (./configuration.md)

SDK configuration（Spring Bean）：

```
<?xml version="1.0" encoding="UTF-8" ?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.
↪springframework.org/schema/p"
  xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop="http://www.
↪springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
          http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
          http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- AMOP message processing pool configuration, which can be configured according␣
↪to actual needs -->
<bean id="pool" class="org.springframework.scheduling.concurrent.
↪ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="50" />
  <property name="maxPoolSize" value="100" />
  <property name="queueCapacity" value="500" />
  <property name="keepAliveSeconds" value="60" />
  <property name="rejectedExecutionHandler">
    <bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy" />
  </property>
</bean>

<!-- group information configuration -->
  <bean id="groupChannelConnectionsConfig" class="org.fisco.bcos.channel.handler.
↪GroupChannelConnectionsConfig">
      <property name="allChannelConnections">
        <list>
          <bean id="group1"  class="org.fisco.bcos.channel.handler.
↪ChannelConnections">
            <property name="groupId" value="1" />
            <property name="connectionsStr">
              <list>
                <value>127.0.0.1:20200</value> <!-- format: IP:port -->
                <value>127.0.0.1:20201</value>
              </list>
            </property>
          </bean>
        </list>
      </property>
    </bean>
```

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

blockchain front configuration, if there is a DMZ region:

```
<?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">

    <!-- blockchain node information configuration -->
  <bean id="proxyServer" class="org.fisco.bcos.channel.proxy.Server">
    <property name="remoteConnections">
      <bean class="org.fisco.bcos.channel.handler.ChannelConnections">
        <property name="connectionsStr">
          <list>
            <value>127.0.0.1:5051</value><!-- format:IP:port -->
          </list>
        </property>
      </bean>
    </property>

    <property name="localConnections">
      <bean class="org.fisco.bcos.channel.handler.ChannelConnections">
      </bean>
    </property>
    <!-- blockchain front listening port configuration, uses for SDK connection -->
    <property name="bindPort" value="30333"/>
  </bean>
</beans>
```

6.12.4 SDK uses

AMOP's messaging is based on the topic mechanism. A topic is set in a server first. When a client sends a message to the topic, the server can receive soon. AMOP supports multiple topic for messaging in the same blockchain network. Topic supports any number of servers and clients. When multiple servers follow on the same topic, the topic messages are randomly sent to one of available servers.

Server code:

```
package org.fisco.bcos.channel.test.amop;

import org.fisco.bcos.channel.client.Service;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.HashSet;
import java.util.Set;

public class Channel2Server {
    static Logger logger = LoggerFactory.getLogger(Channel2Server.class);

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
          System.out.println("Param: topic");
          return;
        }

        String topic = args[0];

        logger.debug("init Server");

        ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
        Service service = context.getBean(Service.class);

        // set topic to support multiple topic
        Set<String> topics = new HashSet<String>();
        topics.add(topic);
        service.setTopics(topics);

        // PushCallback class, is used to handles messages. see the Callback code.
        PushCallback cb = new PushCallback();
        service.setPushCallback(cb);

        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(
↪"====================================================================");

        // launch service
        service.run();
    }
}
```

The server's PushCallback class case:

```
package org.fisco.bcos.channel.test.amop;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.fisco.bcos.channel.client.ChannelPushCallback;
import org.fisco.bcos.channel.dto.ChannelPush;
import org.fisco.bcos.channel.dto.ChannelResponse;

class PushCallback extends ChannelPushCallback {
    static Logger logger = LoggerFactory.getLogger(PushCallback2.class);
```

```
    // onPush method is called when an AMOP message is received.
    @Override
    public void onPush(ChannelPush push) {
        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        logger.debug("push:" + push.getContent());

        System.out.println(df.format(LocalDateTime.now()) + "server:push:" + push.
↪getContent());

        // respond message
        ChannelResponse response = new ChannelResponse();
        response.setContent("receive request seq:" + String.valueOf(push.
↪getMessageID()));
        response.setErrorCode(0);

        push.sendResponse(response);
    }
}
```

client case:

```
package org.fisco.bcos.channel.test.amop;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.fisco.bcos.channel.client.Service;
import org.fisco.bcos.channel.dto.ChannelRequest;
import org.fisco.bcos.channel.dto.ChannelResponse;

public class Channel2Client {
    static Logger logger = LoggerFactory.getLogger(Channel2Client.class);

    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println("param: target topic total number of request");
            return;
        }

        String topic = args[0];
        Integer count = Integer.parseInt(args[1]);
        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

        ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");

        Service service = context.getBean(Service.class);
        service.run();

        System.out.println("3s ...");
        Thread.sleep(1000);
        System.out.println("2s ...");
        Thread.sleep(1000);
        System.out.println("1s ...");
        Thread.sleep(1000);

        System.out.println("start test");
```

```
        System.out.println(
↪"=====================================================================");
        for (Integer i = 0; i < count; ++i) {
          Thread.sleep(2000);  // It needs to take time to establish a connection.␣
↪If you send a message immediately, it will fail.

          ChannelRequest request = new ChannelRequest();
          request.setToTopic(topic); // set the message topic
          request.setMessageID(service.newSeq());  // The message sequence number.␣
↪When you need to uniquely identify a message, you can use newSeq() to generate␣
↪randomly.
          request.setTimeout(5000);  // Message timeout

          request.setContent("request seq:" + request.getMessageID());  // The␣
↪message content is sent
          System.out.println(df.format(LocalDateTime.now()) + " request seq:" +␣
↪String.valueOf(request.getMessageID())
          + ", Content:" + request.getContent());

          ChannelResponse response = service.sendChannelMessage2(request);  //␣
↪send message

          System.out.println(df.format(LocalDateTime.now()) + "response seq:" +␣
↪String.valueOf(response.getMessageID())
                  + ", ErrorCode:" + response.getErrorCode() + ", Content:" +␣
↪response.getContent());
        }
    }
}
```

6.12.5 Test

After completing the configuration as described above, user can specify a topic: topic and execute the following two commands for testing.

Launch amop server:

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2Server␣
↪[topic]
```

Launch amop client:

```
java -cp 'conf/:apps/*:lib/*' org.fisco.bcos.channel.test.amop.Channel2Client␣
↪[topic] [Number of messages]
```

6.12.6 Error code

- 99:message failed to be sent. After AMOP attempts to send message by all the links, the message is not sent to the server. It is recommended to use `seq` that is generated during the transmission to check the processing status of each node on the link.
- 102: message times out. It is recommended to check whether the server has processed the message correctly and the bandwidth is sufficient.

6.13 Storage security

The data of the alliance chain is only visible to members of the alliance. Disk encryption ensures the security of the data running on the alliance chain on the hard disk. Once the hard drive is taken out from the intranet

environment of alliance chain, the data will not be decrypted.

Disk encryption encrypts the content stored on the hard disk by the node. The encrypted content includes: the data of the contract and the private key of the node.

For specific disk encryption introduction, please refer to: Introduction of Disk Encryption

6.13.1 Key Manager deployment

Each agency has a Key Manager. For specific deployment steps, please refer to Key Manager README

6.13.2 Node building

Use the script [`build_chain.sh`] (build_chain.md) to build a node with normal operations.

```
cd FISCO-BCOS/tools
bash build_chain.sh -l "127.0.0.1:4" -p 12300 -e ../build/bin/fisco-bcos
```

Important: The node cannot be launched until the dataKey is configured. Before the node runs for the first time, it must be configured to use the disk encryption or not. Once the node starts running, it cannot be switched its state.

6.13.3 Key Manager launch

To launch `key-manager` directly. If `key-manager` is not deployed, refer to Key Manager README Introduction.

```
# parameter: port, superkey
./key-manager 31443 123xyz
```

launch successfully and print the log.

```
[1546501342949][TRACE][Load]key-manager stared,port=31443
```

6.13.4 DataKey configuration

Important: The node configured by the dataKey must be newly generated node and has not been launched.

To execute the script, define `dataKey`, and get `cipherDataKey`

```
cd key-manager/scripts
bash gen_data_secure_key.sh 127.0.0.1 31443 123456
```

The script for getting `cipherDataKey` automatically prints out the ini configuration required for the disk encryption (see below). Now, the cipherDataKey is:`cipher_data_key=ed157f4588b86d61a2e1745efe71e6ea`

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

To write the ini configuration that has been disk encryption to the node configuration file (config.ini).

```
vim nodes/127.0.0.1/node0/config.ini
```

To put it at last like this.

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

6.13.5 Encrypted node private key

To execute script and encrypt node private key

```
cd key-manager/scripts
# parameter:ip port node private key file cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 nodes/127.0.0.1/node0/conf/node.key␣
↪ed157f4588b86d61a2e1745efe71e6ea
```

The node private key is automatically encrypted after execution, and the files before encryption is backed up to the file `node.key.bak.xxxxxx`. **Please take care of the backup private key and delete the backup private key generated on the node**

```
[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!
```

If you check the `node.key`, you can see that it has been encrypted as ciphertext.

```
8b2eba71821a5eb15b0cbe710e96f23191419784f644389c58e823477cf33bd73a51b6f14af368d4d3ed647d9de681893
```

Important: All files that need to be encrypted are listed below. If they are not encrypted, the node cannot be launched.

- standard version: conf/node.key
- national cryptography version: conf/gmnode.key和conf/origin_cert/node.key

---

6.13.6 Node running

to launch node directly

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

6.13.7 Correct judgment

(1) The node runs and generates block normally, and the block information is continuously output.

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

(2) `key-manager` will print a log each time the node launches. For example, when a node launches, the log directly output by Key Manager is as follows.

```
[1546504272699][TRACE][Dec]Respond
{
   "dataKey" : "313233343536",
   "error" : 0,
   "info" : "success"
}
```

6.14 Commercial Cryptography

For fully supporting the commercial cryptography algorithm, FISCO integrates the national encryption, decryption, signature, verification, hash algorithm and SSL communication protocol in the FISCO BCOS platform based on the commercial cryptography standard. The design documents can be found in the FISCO BCOS Design Manual. commercial cryptography Version.

6.14.1 Initial deployment of FISCO BCOS commercial cryptography version

This section uses the [build_chain] (build_chain.md) script to build a 4-nodes FISCO BCOS chain locally, and uses Ubuntu 16.04 system as an example to operate. This section uses pre-compiled static fisco-bcos binaries for testing on CentOS 7 and Ubuntu 16.04.

```
# rely on the installation of Ubuntu16
$ sudo apt install -y openssl curl
# prepare environment
$ cd ~ && mkdir -p fisco && cd fisco
# download build_chain.sh script
$ 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\." |␣
↪sort -u | tail -n 1 | cut -d \" -f 4`/build_chain.sh && chmod u+x build_chain.sh
```

After performing the above steps, the structure in the fisco directory is as follows:

```
fisco
├── bin
│   └── fisco-bcos
└── build_chain.sh
```

- build a 4-nodes FISCO BCOS chain

```
# Generate a 4-nodes FISCO chain. All nodes belong to group1. The following␣
↪instructions are executed in the fisco directory.
# -p specifies the starting ports which are p2p_port, channel_port, jsonrpc_port
# According to the following instructions, it needs to ensure that the 30300~30303,
↪ 20200~20203, 8545~8548 ports of the machine are not occupied.
# -g The commercial cryptography compilation option. It will generate a node of␣
↪commercial cryptography after using successfully. Download the latest version␣
↪from GitHub by default.
$ ./build_chain.sh -l "127.0.0.1:4" -p 30300,20200,8545 -g
```

For the build_chain.sh script option, please [refer to here] (build_chain.md). The command that execute normally will output All completed. (If there is no output, refer to nodes/build.log for checking).

```
[INFO] Downloading tassl binary ...
Generating CA key...
Generating Guomi CA key...
============================================================
Generating keys ...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
```

```
Generating configurations...
Processing IP:127.0.0.1 Total:4 Agency:agency Groups:1
============================================================
[INFO] FISCO-BCOS Path   : bin/fisco-bcos
[INFO] Start Port        : 30300 20200 8545
[INFO] Server IP         : 127.0.0.1:4
[INFO] State Type        : storage
[INFO] RPC listen IP     : 127.0.0.1
[INFO] Output Dir        : /mnt/c/Users/asherli/Desktop/key-manager/build/nodes
[INFO] CA Key Path       : /mnt/c/Users/asherli/Desktop/key-manager/build/nodes/
↪gmcert/ca.key
[INFO] Guomi mode        : yes
============================================================
[INFO] All completed. Files in /mnt/c/Users/asherli/Desktop/key-manager/build/nodes
```

After the deployment of the commercial cryptography alliance chain is completed, the rest of operations are same as [installation] (../installation.md).

6.14.2 commercial cryptography configuration information

The nodes of FISCO BCOS commercial cryptography version message through using SSL secure channel. The main configuration items of the SSL certificate are concentrated in the following configuration items:

```
[network_security]

data_path: path where the certificate file is located
key: the path of the node private key relative to the data_path
cert: the path of the certificate gmnode.crt relative to data_path
ca_cert: the path of certificate gmca

;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
```

6.14.3 using commercial cryptography SDK

For details, refer to [SDK Documentation] (../sdk/sdk.html#id8).

6.14.4 using commercial cryptography console

The function of commercial cryptography console is used in the same way as the standard console. See [Console Operations Manual] (../manual/console.md).

6.14.5 commercial cryptography configuration

commercial cryptography Key Manager

Using the commercial cryptography Key Manager needs to be recompiled the standard Key Manager. The difference of them is `-DBUILD_GM=ON` option needs to be added when doing cmake.

```
# under centos
cmake3 .. -DBUILD_GM=ON
# under ubuntu
cmake .. -DBUILD_GM=ON
```

Other steps are same as the standard Key Manager. Please refer to key-manager repository.

commercial cryptography node configuration

FISCO BCOS commercial cryptography version adopts dual certificate mode, so two sets of certificates are needed for disk encryption. They are the conf/gmnode.key and conf/origin_cert/node.key. Other operations of disk encryption are the same as [Standard Edition Loading Encryption Operation] (./storage_security.md).

```
cd key-manager/scripts
#encrypt conf/gmnode.key parameter: ip port  Node private key file cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 nodes/127.0.0.1/node0/conf/gmnode.key␣
↪ed157f4588b86d61a2e1745efe71e6ea
#encrypt conf/origin_cert/node.key parameter: ip port  Node private key file␣
↪cipherDataKey
bash encrypt_node_key.sh 127.0.0.1 31443 nodes/127.0.0.1/node0/conf/origin_cert/
↪node.key ed157f4588b86d61a2e1745efe71e6ea
```

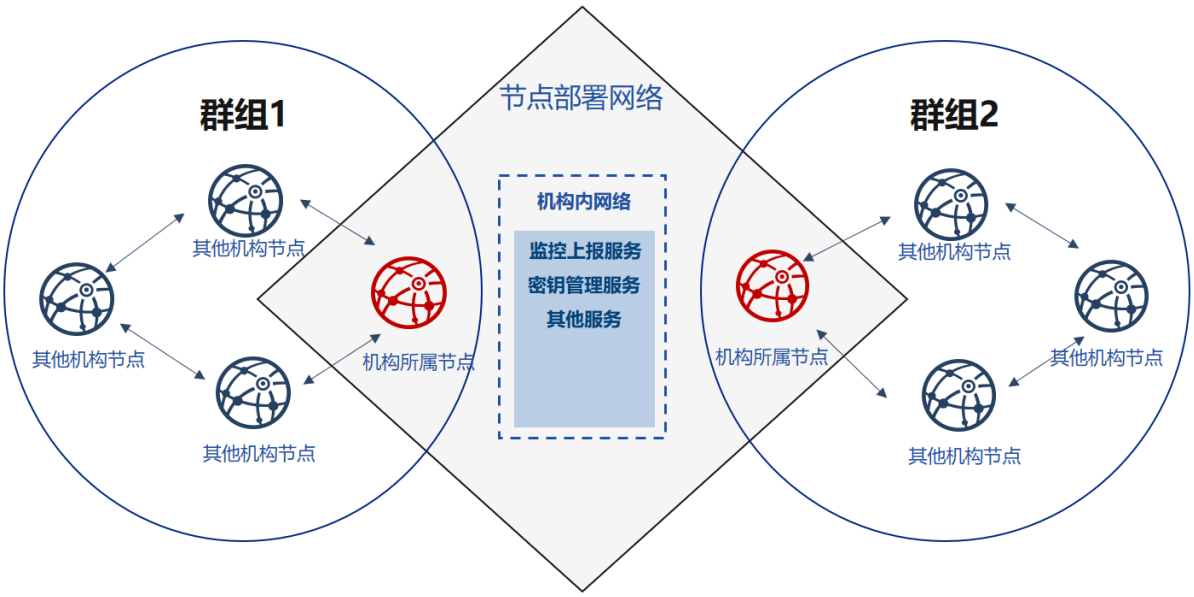
CHAPTER 7

Deployment tool

Brief introduction

FISCO BCOS generator provides companies with an easy toolkit for deployment, administration and monitoring of multi-group consortium chain.

- It eliminates the complexity of generating and maintaining blockchain and offers alternative deployment methods.
- It requires agencies to share node credentials and manage their own private key but not expose to outsider, maintaining security of all nodes.
- It helps agencies deploy nodes safely through e-certificate trading, maintaining equality of all nodes.



Design background

There cannot be exhaustive trust between equal agencies in consortium chain, where e-certificate will be needed for nodes to authenticate each other's identity. Certificate is the identity documentation for each agency. And the generation of certificate depends on its own public & private key pair. Private key represents its identity information that is private and strictly confidential. In the process of activation and operation, node signs on the

data packet with private key in order to fulfil identity authentication. Provided that an agency's private key is revealed, anyone else can pretend as the owner and get authorized without affirmation of this agency.

When initializing the group of FISCO BCOS, nodes should reach an agreement to create a Genesis Block. Genesis Block, unique and only within one group, bears the identity information of the initial nodes, which is formed through e-credential exchanging.

Current IT administration tools for consortium chain usually ignore the requirement for equality and security of companies during initialization. And initialization needs agencies to agree on identity information on Genesis Block. So, who should be the information generator is crucial. One of the solution is that an agency generates its node information first and then activate blockchain for other nodes to join in; or a third-party authority generates information for all nodes and send the node configuration files to each agency. Additionally, FISCO BCOS 2.0 adapts more private and scalable multi-group architecture. It is an architecture where data and transactions between groups are separated by running independent consensus algorithm, a way to maintain privacy and security in blockchain scenarios.

In the above models, there is always one agency who gains priority to join the consortium chain or acquires private keys of all nodes.

How to make sure the group is formed in an equal, safe and private way? How to guarantee reliable and effective operation of nodes? The privacy and security of group ledgers, as well as the confidentiality of group formation and operation, need to be achieved in an effective way.

Design concept

FISCO BCOS generator is a solution designed for problems described above. It takes into consideration the equal deployment and group formation of different agencies on the basis of flexibility, security, ease-of-use and equality.

Flexibility:

- No installation, ready to use
- Alternative deployment methods
- Allow multiple changes in architecture

Safety:

- Allow multiple changes in architecture
- Private key is kept internally
- Negotiation between agencies is based on certificates only

Ease-to-use:

- Support multiple networking models
- Alternative commands for various needs
- Monitor audit script

Equality:

- Equal authority of agencies
- All agencies co-generate Genesis Block
- Equal administrative power within groups

For consortium chain based on existed root credential, it can fast configure multiple groups on chain to adapt for different business needs.

After agreeing on the data model of node certificate, IP and port number and filling the configuration items, each agency can generate a configuration file folder locally that includes no private key. Agencies can keep their private keys internally and prevent malicious attackers in disguise of nodes or any information leakage, even if the configuration files are lost. In this way, security and usability of nodes can be achieved at the same time.

Users make agreement to generate Genesis Block and node configuration file folder, and then activate nodes so that they will conduct multi-group networking according to the configuration files.

7.1 Download & install

7.1.1 Environment dependency

Dependency of FISCO BCOS generator:

7.1.2 Download & install

Download

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

Install

```
$ cd generator
$ bash ./scripts/install.sh
```

Check if it is installed successfully

```
$ ./generator -h
# if succeed, output usage: generator xxx
```

7.1.3 Pull node binaries

Pull the latest fisco-bcos binary files to meta

```
$ ./generator --download_fisco ./meta
```

Check binaries version

```
$ ./meta/fisco-bcos -v
# if succeed, output FISCO-BCOS Version : x.x.x-x
```

PS: Source code compilation node binaries user need only to put the compiled binaries to `meta` directory.

7.2 One-click deployment

This chapter introduces the operation method of one-click deployment of single agency with networking model of **2 agencies, 1 group, 4 nodes**.

This turial is adaptable to the deployment of sinle agency building all nodes. For multi-agency deployment please read Deployment Tool。

7.2.1 Download and install

Download

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

7.2.2 Case analysis

In this section, we will create a networking model with topology of 2 agencies, 1 group and 4 nodes on the machine with IP `127.0.0.1`. Each node's IP and port number is:

Important: targeting the vps server in cloud servers, RPC monitoring address ahould be the real address (like internal network address or 127.0.0.1) of the network cards, which may be different with the ssh server log in by users.

7.2.3 Description

Users need to prepare a `one_click` file folder as the picture shows, which contains directories of different agencies and each needs a config file `node_deployment.ini`. Before use, please make sure that the meta folder of generator has not been operated.

View one-click deployment template folder:

```
ls ./one_click
```

```
# parameter description
# for muti-agencies, the folder needs to be created manually
one_click # user specifies a folder for one-click deployment
├── agencyA # agency A directory, node of agency A and relative files will be␣
↪generated there after the commands are executed
│   └── node_deployment.ini # agency A node config file, according to which one-
↪click deployment command generates node
└── agencyB # agency B directory, where node of agency B and relative files will␣
↪be generated after the commands are executed
    └── node_deployment.ini # agency B node config file, according to which one-
↪click deployment command generates node
```

7.2.4 Agency fill in node information

According to the tutorial, agency A and agency B place the config file under one_click folder like below:

```
cat > ./one_click/agencyAnode_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
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
EOF
```

```
cat > ./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
; 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=30302
channel_listen_port=20202
jsonrpc_listen_port=8547

[node1]
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
```

7.2.5 Build node

```
bash ./one_click_generator.sh ./one_click
```

After execution, the structure of ./one_click folder will be like:

View the one-click template folder after execution:

```
ls ./one_click
```

```
├── agencyA # Agency A folder
│   ├── agency_cert # Agency A certificate and private key
│   ├── generator-agency # generator folder automately replace agency A for␣
↪operations
│   ├── node # node generated by agency A, push to the respective server for multi-
↪machine deployment
│   ├── node_deployment.ini # node config information of agency A
│   └── sdk # sdk and console config file of agency A
└── agencyB
    ├── agency_cert
    ├── generator-agency
    ├── node
    ├── node_deployment.ini
    └── sdk
```

7.2.6 Start node

Call script to start node:

```
bash ./one_click/agencyA/node/start_all.sh
```

```
bash ./one_click/agencyB/node/start_all.sh
```

View node progress:

```
ps -ef | grep fisco
```

```
# command description
# you can get the following progress
fisco  15347     1  0 17:22 pts/2    00:00:00 ~/generator/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/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/one_click/agencyA/node/
↪node_127.0.0.1_30302/fisco-bcos -c config.ini
fisco  15456     1  0 17:22 pts/2    00:00:00 ~/generator/one_click/agencyA/node/
↪node_127.0.0.1_30303/fisco-bcos -c config.ini
```

7.2.7 View node status

View node log:

```
tail -f ~/generator/one_click/agency*/node/node*/log/log*  | grep +++
```

```
# command description
# +++ means that the node is in normal consensus status
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...
```

7.2.8 Operate console

Due to the bulk of console, there is no direct integration during one-click deployment. Users can get console through the following command.

It may take long to get console:

```
./generator --download_console ./
```

Configure console of agency A:

Copy console to agency A:

```
cp -rf ./console ./one_click/agencyA/console
```

Configure files of agency A console. Start of console needs certificate, private key and console config file:

```
cp ./one_click/agencyA/sdk/* ./one_click/agencyA/console/conf
```

Start of console needs to intall java:

```
cd ./one_click/agencyA/console && bash ./start.sh 1
```

Likewise, configure console of agency B:

```
cd ~/generator && cp -rf ./console ./one_click/agencyB/console
```

Configure files of agency B console, start of console needs certificate, private key and console config file:

```
cp ./one_click/agencyB/sdk/* ./one_click/agencyB/console/conf
```

7.2.9 Create new group and expansion

The later operations of the deployment tool is the same with Deployment tool tutorial.

The later operations of node expansion or new group division is introduced in Operation tutorial and Deplotment tool tutorial.

To create new group, users can modify `./conf/group_genesis.ini` file under the folder that executes `click2start.sh` script, and execute `--create_group_genesis`.

To expand new node, users can modify `./conf/node_deployment.ini`, use `--generate_all_certificates` to generate certificate and use `--build_install_package` to generate node.

If you have any problems when reading this tutoril, you can check FAQ

7.3 Config file

The config files of FISCO BCOS generator are placed under ./conf folder, including group genesis block config file `group_genesis.ini`, node config file `node_deployment.ini`.

User configures node config file folder by operations on files under conf folder.

7.3.1 Metadata folder meta

The meta folder of FISCO BCOS generator is metadata folder to store `fisco bcos` binaries, chain certificate `ca.crt`, agency certificate `agency.crt`, agency private key and node certificate, group genesis block file and so on.

The format of stored certificates should be cert_p2pip_port.crt. For example: cert_127.0.0.1_30300.crt.

FISCO BCOS generator will generate node config file according to the certificates under meta folder and the config files under conf folder.

7.3.2 group_genesis.ini

Through modifying the configuration of `group_genesis.ini`, user generates configuration of new group genesis block under specific directory and meta folder. Such as `group.1.genesis`.

```
[group]
group_id=1

[nodes]
;node p2p address of group genesis block
node0=127.0.0.1:30300
node1=127.0.0.1:30301
node2=127.0.0.1:30302
node3=127.0.0.1:30303
```

Important: Node certificate is needed during generating genesis block. In the above case, the config file needs 4 nodes' certificates, which are: cert_127.0.0.1_30301.crt, cert_127.0.0.1_30302.crt, cert_127.0.0.1_30303.crt and cert_127.0.0.1_30304.crt.

7.3.3 node_deployment.ini

Through modifying `node_deployment.ini` configuration, user can use –build_install_package command to generate node config file containing no private key under specific folder. Each `section[node]` configured by user is the needed node config file folder. `section[peers]` is the p2p information for connection with other nodes.

Config file example:

```
[group]
group_id=1

# Owned nodes
[node0]
p2p_ip=127.0.0.1
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
```

Read node config command. To generate node certificate and node config file folder will need to read the config file.

7.3.4 Template folder tpl

The template folder of generator is as below:

```
├── applicationContext.xml # sdk config file
├── config.ini # node config file template
├── config.ini.gm # OSCCA node config file template
├── group.i.genesis # group genesis block template
├── group.i.ini # group block configuration template
├── start.sh  # start node script template
├── start_all.sh # start nodes in batch script template
├── stop.sh # stop node script template
└── stop_all.sh # stop nodes in batch template
```

To modify the consensus algorithm of node and the configured default db, user only needs to modify the configuration of `config.ini`, re-execute the commands to set relative node information.

For details of FISCO BCOS configuration please check FISCO BCOS config file

7.3.5 P2p node connection file peers.txt

P2p node connection file `peers.txt` is the node connection information of the **other agencies** specified when generating node config file folder. When executing `build_install_package` command, it's needed to spec-

ify the p2p node connection file `peers.txt`, according to which node config file folder will start communication with other nodes.

User that executes `generate_all_certificates` command generates `peers.txt` according to the `node_deployment.ini` filled under `conf` directory. User that adopts other ways to generate certificate needs to generate p2p node connection file manually and send to peers. The format of p2p node connection file is:

```
127.0.0.1:30300
127.0.0.1:30301
```

The format: node ip:p2p_listen_port

- for multi-agency node communication, the files need to be combined

7.4 Operation Tutorial

FISCO BCOS generator contains multiple operations about node generation, expansion, group division and certificate which are introduced as below:

7.4.1 create_group_genesis (-c)

Operation Tutorial

```
$ cp node0/node.crt ./meta/cert_127.0.0.1_3030n.crt
...
$ vim ./conf/group_genesis.ini
$ ./generator --create_group_genesis ~/mydata
```

After the program is executed, `group.i.genesis` of mgroup.ini will be generated under ~/mydata folder

The generated `group.i.genesis` is the Genesis Block of the new group.

Note: In FISCO BCOS 2.0, each group has one Genesis Block.

7.4.2 build_install_package (-b)

Operation Tutorial

```
$ vim ./conf/node_deployment.ini
$ ./generator --build_install_package ./peers.txt ~/mydata
```

After the program is executed, a few file folders named node_hostip_port will be generated under ~/mydata folder and pushed to the relative server to activate node.

7.4.3 generate_chain_certificate

```
$ ./generator --generate_chain_certificate ./dir_chain_ca
```

Now, user can find root certificate `ca.crt` and private key `ca.key` under ./dir_chain_ca folder.

7.4.4 generate_agency_certificate

```
$ ./generator --generate_agency_certificate ./dir_agency_ca ./chain_ca_dir The_
↪Agency_Name
```

Now, user can locate The_Agency_Name folder containing agency certificate `agency.crt` and private key `agency.key` through route ./dir_agency_ca.

7.4.5 generate_node_certificate

```
$ ./generator --generate_node_certificate node_dir(SET) ./agency_dir  node_p2pip_
↪port
```

Then user can locate node certificate `node.crt` and private key `node.key` through route node_dir.

7.4.6 generate_sdk_certificate

```
$ ./generator --generate_sdk_certificate ./dir_sdk_ca ./dir_agency_ca
```

Now user can locate SDK file folder containing SDK certificate `node.crt` and private key `node.key` through route ./dir_sdk_ca.

7.4.7 generate_all_certificates

```
$ ./generator --generate_all_certificates ./cert
```

Note: the above command will create node certificate according to ca.crt, agency certificate agency.crt and agency private key agency.key of meta folder.

- absence of any of the above 3 files will fail the creation of node certificate, and the program will throw an exception

---

Once the execution is done, user can find node certificate and private key under ./cert folder, and node certificate under ./meta folder.

7.4.8 merge_config (-m)

–merge_config command can merge p2p sections of 2 config.ini

For instance, the p2p section in config.ini of A is:

```
[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
```

the p2p section in config.ini of B is:

```
[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 = 192.167.1.1:30300
node.3 = 192.167.1.1:30301
```

the command will result in:

```
[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 = 192.167.1.1:30302
node.3 = 192.167.1.1:30303
node.4 = 192.167.1.1:30300
node.5 = 192.167.1.1:30301
```

For example:

```
$ ./generator --merge_config ~/mydata/node_A/config.ini  ~/mydata/node_B/config.ini
```

When it works, the p2p sections in config.ini of node_A and node_B will be merged to ~/mydata/node_B/config.ini

7.4.9 deploy_private_key (-d)

–deploy_private_key command will import the private key of nodes with same name to the generated configuration file folder

For example:

```
$./generator --deploy_private_key ./cert ./data
```

If ./cert contains file folders named node_127.0.0.1_30300 and node_127.0.0.1_30301, which are placed with node private key node.key,

./data contains file folders named node_127.0.0.1_30300 and node_127.0.0.1_30301,

then this command would import private key under./cert to ./data folder

7.4.10 add_peers (-p)

–add_peers command can import the files of assigned peers to the generated node configuration file folder.

For example:

```
$./generator --add_peers ./meta/peers.txt ./data
```

If ./data contains configuration file folder named node_127.0.0.1_30300 and node_127.0.0.1_30301,

then this command will import peers file with connection information to the node configuration files `config.ini` under ./data.

add_group (-a)

–add_group command will import assigned peers file to the generated node configuration file folder.

For example:

```
$./generator --add_group ./meta/group.2.genesis ./data
```

If ./data contains configuration file folder named node_127.0.0.1_30300 and node_127.0.0.1_30301,

then this command will import connection information of Group 2 to `conf` folder of all nodes under ./data.

7.4.11 download_fisco

–download_fisco can download `fisco-bcos` binary file under assigned section.

For example:

```
$./generator --download_fisco ./meta
```

This command can download `fisco-bcos` executable binary file under ./meta folder.

7.4.12 download_console

–download_consolecan download and control console under assigned section.

For example:

```
$./generator --download_console ./meta
```

This command will configure the console under ./meta folder according to `node_deployment.ini`.

7.4.13 get_sdk_file

–get_sdk_file command can acquire `node.crt`, `node.key`, `ca.crt` and `applicationContext.xml` that are needed in configuration of console and sdk under assigned section.

For example:

```
$./generator --get_sdk_file ./sdk
```

This command will generate the above configuration file according to `node_deployment.ini` under ./sdk

7.4.14 version (-v)

–version command can help view the version code of current deployment tool.

```
$ ./generator --version
```

7.4.15 help (-h)

User can use -h or –help command to check help list

For example:

```
$ ./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]
```

7.4.16 Operation in OSCCA standard (China)

All the commands in FISCO BCOS generator are adaptable for commercial version of `fisco-bcos` (oscca-approved encryption). When using this version, every certificates and private key should be prefixed with `gm`. The description reads below:

On-off switch (-g command)

Once -g command is executed, all the operations concerning certificates, nodes and group Genesis Block will generate the above files of OSCCA standard.

generate certificate

When executing generate_*_certificate together with -g command, the generated certificate will be in OSCCA standard.

For example:

```
$ ./generator --generate_all_certificates ./cert -g
```

Note: the above command will generate node certificate according to gmca.crt, agency certificate gmagency.crt and agency private key gmagency.key placed under meta folder.

- Absence of any one of the three files will fail the generation of node certificate, and the program will throw an exception.

generate group Genesis Block in OSCCA standard

For example:

```
$ cp node0/gmnode.crt ./meta/gmcert_127.0.0.1_3030n.crt
...
$ vim ./conf/group_genesis.ini
$ ./generator --create_group_genesis ~/mydata -g
```

After executing this program, user can locate `group.i.genesis` in mgroup.ini under ~/mydata folder.

`group.i.genesis` is the Genesis Block of the new group.

generate node configuration file folder in OSCCA standard

For example:

```
$ vim ./conf/node_deployment.ini
$ ./generator --build_install_package ./peers.txt ~/mydata -g
```

After executing program, multiple folders named node_hostip_port will be generated under ~/mydata folder and pushed to the relative server to activate node.

7.4.17 Monitoring design

FISCO BCOS generator has preset monitoring script in all generated node configuration folders. Warning information of node will be sent to IP address assigned by user after configuration. FISCO BCOS generator places the monitoring script under the assigned section for node configuration files. If user assigns the folder named “data”, then user can locate it under monitor folder of data section.

For example:

```
$ cd ./data/monitor
```

Purpose of use:

1. to monitor status of node, to reactive node
2. to acquire block number of node and view information, to ensure the consensus of nodes
3. to analyze printing of node logs in last 1 minutes, to collect critical errors information of printed logs, to monitor status of node in real time
4. to assign log file or time period, to analyze information of consensus message management, block generation and transaction volume and node's health.

Warning service configuration

Before using this service, user needs to configure the warning service. Here we take the WeChat notification of server酱 as an example. You can read the configuration as reference: server酱

User associates it with personal github account and WeChat account, then uses -s command of this script to send warning message to the associated WeChat.

If user wants to try different services, user can personalize the service by modifying alarm() { # change http server} function of monitor.sh

help command

The way to check the usage of script by help command

```
$ ./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 analyzed
   -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
```

Command description:

- -s assign the configuration address of warning service. It can be the ip of warning notification.
- -m set monitor mode to statistics or monitor. Monitor is the default mode.
- -f analyze node log
- -o assign location of node
- -p set notification name. "fisco-bcos" is the default one.
- -g assign group to be monitored. All group is defaulted to be monitored.
- -d time scope of log analysis. Default to be within 10 minutes and 60 minutes as the maximum
- -r assign the receiver of warning notification

- -h help command

For example

- use script to monitor status of assigned nodes, send message to receiver Alice:

```
$ bash monitor.sh -s https://sc.ftqq.com/[SCKEY(登入后可见)].send -o alice/nodes -r␣
↪Alice
```

- use script to collect node information and send message to Alice:

```
$ bash monitor.sh -s https://sc.ftqq.com/[SCKEY(登入后可见)].send -m statistics -o␣
↪alice/nodes -r Alice
```

- use script to collect information of group 1 and group 2 of specific node log, send message to 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
```

7.4.18 handshake failed test

The `check_certificates.sh` script in scripts folder of FISCO BCOS generator contains anomaly detection with error message `handshake failed` in node log.

Acquire test script

If user needs to test node generated by `buildchain.sh`, the following command can help acquire test script:

```
$ curl -LO https://raw.githubusercontent.com/FISCO-BCOS/generator/develop/scripts/
↪check_certificates.sh && chmod u+x check_certificates.sh
```

User that deployed node with generator can acquire script from scripts/check_certificates.sh under the root directory of generator.

Test valid date of certificate

-t command of `check_certificates.sh` can test certificate by comparing the valid date with the current system time.

Example:

```
$ ./check_certificates.sh -t ~/certificates.crt
```

The second parameter can be any x509 certificate that prompt `check certificates time successful` after passing verification or exception if fails.

Certificate verification

-v command of `check_certificates.sh` will verify node certificate according to the root certificate set by user.

```
$ ./check_certificates.sh -v ~/ca.crt ~/node.crt
```

After successful verification it will prompt `use ~/ca.crt verify ~/node.crt successful`, or prompt exception if fails.

CHAPTER 8

Web3SDK

Web3SDK supports accessing nodes to check node status, modify the settings and send transactions. FISCO BCOS 2.0 documentation only adapts to Web3SDK 2.0 or above version (which only adapt to FISCO BCOS 2.0 or above version in turn). For Web3SDK 1.2.x please check Web3SDK 1.2.x Documentation.

Main features of version 2.0 include:

- provide Java API to call FISCO BCOS JSON-RPC
- support managing blockchain by precompiled contract
- provide secure and efficient message channel with Amop
- support transaction in OSCCA standard

8.1 Environment requirements

Important:

- java version

  JDK8 or above version is required. Because the lack of JCE(Java Cryptography Extension) of OpenJDK in YUM repository of CentOS will fail the connection between Web3SDK and nodes, we recommend users to download it from OpenJDK website when it is CentOS system. Download here Installation Guide

- FISCO BCOS environment building

  please check FISCO BCOS installation guide

- network connectivity

  using telnet command to test if channel_listen_port of nodes connected with Web3SDK is open. If not, please check the network connectivity and security strategy.

8.2 Import SDK to java application

Import SDK to java application through gradle or maven

gradle:

```
compile ('org.fisco-bcos:web3sdk:2.0.3')
```

maven:

```
<dependency>
    <groupId>org.fisco-bcos</groupId>
    <artifactId>web3sdk</artifactId>
    <version>2.0.3</version>
</dependency>
```

Because the relative jar archive of the solidity compiler of Ethereum is imported, we need to add a remote repository of Ethereum in the gradle configuration file build.gradle of the java application.

```
repositories {
        mavenCentral()
        maven { url "https://dl.bintray.com/ethereum/maven/" }
    }
```

Note: if the downloading of the dependent `solcJ-all-0.4.25.jar` is too slow, you can check here for help.

8.3 Configuration of SDK

8.3.1 FISCO BCOS node certificate configuration

FISCO BCOS requires SDK to pass two-way authentication on certificate(ca.crt、node.crt) and private key(node.key) when connecting with nodes. Therefore, we need to copy `ca.crt`, `node.crt` and `node.key` under `nodes/${ip}/sdk` folder of node to the resource folder of the project for SDK to connect with nodes.

8.3.2 Configuration of config file

The config file of java application should be configured. It is noteworthy that FISCO BCOS 2.0 supports Multi-group function, and SDK needs to configure the nodes of the group. The configuration process will be exemplified in this chapter by Spring and Spring Boot project.

8.3.3 Configuration of Spring project

The following picture shows how `applicationContext.xml` is configured in Spring project.

```
<?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>  <!-- each group needs to configure a beam, each␣
↪group can configure multiple nodes-->
                                <bean id="group1"  class="org.fisco.bcos.channel.
↪handler.ChannelConnections">
                                        <property name="groupId" value="1" /> <!--␣
↪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" /> <!--␣
↪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" /> <!-- configure and connect␣
↪group 1 -->
                <property name="agencyName" value="fisco" /> <!-- configure agency␣
↪name -->
                <property name="allChannelConnections" ref=
↪"groupChannelConnectionsConfig"></property>
        </bean>

</beans>
```

Configuration items of `applicationContext.xml`:

- encryptType: the switch of oscca algorithm (default as 0)
  - 0: not use oscca algorithm for transactions
  - 1: use oscca algorithm for transactions (switch on oscca algorithm, nodes will be in oscca standard, for reference of building oscca-standard FISCO BCOS blockchain please check here)
- groupChannelConnectionsConfig:
  - configure the group to be connected, which can be one or more, each should be given a group ID
  - each group configures one or more nodes, set `listen_ip` and `channel_listen_port` of `[rpc]` in node config file **config.ini**.

- channelService: configure the the group connected with SDK through the given group ID, which is in the configuration of groupChannelConnectionsConfig. SDK will be connected with each node of the group and randomly choose one node to send request.

8.3.4 Configuration of Spring Boot project

The configuration of `application.yml` in Spring Boot is exemplified below.

```
encrypt-type: 0  # 0:standard, 1:guomi
group-channel-connections-config:
  all-channel-connections:
  - group-id: 1  #group ID
    connections-str:
                    - 127.0.0.1:20200  # node listen_ip:channel_listen_port
                    - 127.0.0.1:20201
  - group-id: 2
    connections-str:
                    - 127.0.0.1:20202  # node listen_ip:channel_listen_port
                    - 127.0.0.1:20203

channel-service:
  group-id: 1 # The specified group to which the SDK connects
  agency-name: fisco # agency name
```

The configuration items of `application.yml` corresponds with `applicationContext.xml`. For detailed introduction please check the configuration description of `applicationContext.xml`.

8.4 Operation of SDK

8.4.1 Guide for development of Spring

to call API of SDK (check the settings of Web3SDK API list or related blockchain data).

call API of SDK Web3j

load the config file, connect SDK with nodes, acquire web3j object and call API. The codes are exemplified here:

```
    //read config file, start connection between SDK and nodes
    ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
    Service service = context.getBean(Service.class);
    service.run();
    ChannelEthereumService channelEthereumService = new ChannelEthereumService();
    channelEthereumService.setChannelService(service);

    //acquire Web3j object
    Web3j web3j = Web3j.build(channelEthereumService, service.getGroupId());
    //call API getBlockNumber through Web3j object
    BigInteger blockNumber = web3j.getBlockNumber().send().getBlockNumber();
    System.out.println(blockNumber);
```

Note: The transaction process of SDK is default limited to 60 seconds. Transactions with no response in 60 seconds will be considered time out. The timeline can be modified through `ChannelEthereumService`. Here is the example:

```
// set the transaction timeline to 100000 milliseconds, namely 100 second
channelEthereumService.setTimeout(100000);
```

call API of SDK Precompiled

load config file, connect SDK with nodes, acquire Precompiled Service object of SDK and call API. The codes are as below:

```
    //read config file, connect SDK with nodes and acquire Web3j object
    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";
    //fill in user private key for signature in transactions
    Credentials credentials = GenCredential.create(privateKey);
    //acquire SystemConfigService object
    SystemConfigService systemConfigService = new SystemConfigService(web3j,␣
↪credentials);
    //call API setValueByKey through SystemConfigService object
    String result = systemConfigService.setValueByKey("tx_count_limit", "2000");
    //call API getSystemConfigByKey through Web3j object
    String value = web3j.getSystemConfigByKey("tx_count_limit").send().
↪getSystemConfigByKey();
    System.out.println(value);
```

Create and use specific external account

sdk needs an external account to send transactions. Here is a way to create a external account.

```
//create regular external account
EncryptType.encryptType = 0;
//create OSCCA external account, which is needed when sending transaction to OSCCA-
↪standard blockchain nodes
// EncryptType.encryptType = 1;
Credentials credentials = GenCredential.create();
//account address
String address = credentials.getAddress();
//account private key
String privateKey = credentials.getEcKeyPair().getPrivateKey().toString(16);
//account public key
String publicKey = credentials.getEcKeyPair().getPublicKey().toString(16);
```

Use specific external account

```
//Set specific external account by specifying the private key
Credentials credentials = GenCredential.create(privateKey);
```

Load private key file

After the script `get_accounts.sh` generates PEM or PKCS12 private key file (the use of get-account script is introduced in Account management document), the account can be operated through loading PEM or PKCS12 private key file. There are 2 ways to load private key: P12Manager and PEMManager. P12Manager is to load PKCS12 private key file; PEMManager is to load PEM private key file.

- P12Manager example: Configure private key file route and password of PKCS12 account in applicationContext.xml

```
<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>
```

development code

```
//load Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
P12Manager p12 = context.getBean(P12Manager.class);
//offer password to get ECKeyPair, password is set when creating p12 account file
ECKeyPair p12KeyPair = p12.getECKeyPair(p12.getPassword());

//output private key and public key in hexadecimal string
System.out.println("p12 privateKey: " + p12KeyPair.getPrivateKey().toString(16));
System.out.println("p12 publicKey: " + p12KeyPair.getPublicKey().toString(16));

//generate Credentials for web3sdk
Credentials credentials = Credentials.create(p12KeyPair);
System.out.println("p12 Address: " + credentials.getAddress());
```

- PEMManager example

Configure private key route of PEM account in applicationContext.xml

```
<bean id="pem" class="org.fisco.bcos.channel.client.PEMManager" init-method="load"␣
↪>
        <property name="pemFile" value=
↪"classpath:0x0fc3c4bb89bd90299db4c62be0174c4966286c00.pem" />
</bean>
```

load code

```
//load Bean
ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext-keystore-sample.xml");
PEMManager pem = context.getBean(PEMManager.class);
ECKeyPair pemKeyPair = pem.getECKeyPair();

//output private key and public key in hexadecimal string
System.out.println("PEM privateKey: " + pemKeyPair.getPrivateKey().toString(16));
System.out.println("PEM publicKey: " + pemKeyPair.getPublicKey().toString(16));

//generate Credentials for web3sdk
Credentials credentialsPEM = Credentials.create(pemKeyPair);
System.out.println("PEM Address: " + credentialsPEM.getAddress());
```

To deploy and call contract through SDK

Prepare java contract file

Console offers a contract compiler for developers to compile the solidity contract to java contract. For operation steps please check here.

Deploy and call contract

The core function of SDK is to deploy/load contract, and to call API of contract to realize transactions. The deployment of contract calls the deploy method of java class and obtains contract object, which can call getContractAddress method to get the address for contract or other methods for other functions. If the contract has been deployment already, contract object can be loaded by calling load method according to the address of the deployed contract, other methods of which can also be called.

```
    //read config file, connect SDK with nodes, get web3j object
    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());
    //prepare parameters for deploying and calling contract
    BigInteger gasPrice = new BigInteger("300000000");
    BigInteger gasLimit = new BigInteger("300000000");
    String privateKey =
↪"b83261efa42895c38c6c2364ca878f43e77f3cddbc922bf57d0d48070f79feb6";
    //specify external account private key for transaction signature
    Credentials credentials = GenCredential.create(privateKey);
    //deploy contract
    YourSmartContract contract = YourSmartContract.deploy(web3j, credentials, new␣
↪StaticGasProvider(gasPrice, gasLimit)).send();
    //load contract according to the contract address
    //YourSmartContract contract = YourSmartContract.load(address, web3j,␣
↪credentials, new StaticGasProvider(gasPrice, gasLimit));
    //send transaction by calling contract method
    TransactionReceipt transactionReceipt = contract.someMethod(<param1>, ...).
↪send();
    //check data status of the contract by inquiry contract method
    Type result = contract.someMethod(<param1>, ...).send();
```

8.4.2 Guide for Spring Boot development

We take spring-boot-starter as an example. Spring Boot and Spring are similar in development process, except the distinction in config file. Here we will provide some test examples. For detail description on the projects please check the README documents.

8.4.3 Operations on OSCCA function of SDK

- Preconditions: FISCO BCOS blockchain in OSCCA standard, to build it using OSCCA algorithm, please check the operation tutorial.
- switch on OSCCA function: set encryptType as 1 in application.xml/application.yml configuration.

The OSCCA version of SDK shares the same method on calling API with the regular version. The difference is that OSCCA SDK requires a OSCCA version of java contract. To download the jar archive of OSCCA compiler, which is needed for transforming solidity contract into OSCCA version of java contract, please check here. You can also create a lib folder in src folder to place the jar archive when finishing downloading, and modify build.gradle, remove the regular jar archive to replace it with the OSCCA version.

```
 compile ("org.fisco-bcos:web3sdk:x.x.x"){ //For example: web3sdk:2.0.0
      exclude module: 'solcJ-all'
 }
 // jar archive of OSCCA contract compiler 0.4
```

```
compile files('lib/solcJ-all-0.4.25-gm.jar')
// jar archive of OSCCA contract compiler 0.5
// compile files('lib/solcJ-all-0.5.2-gm.jar')
```

It shares the same steps with regular SDK in transformation of solidity contract to OSCCA java contract and deploy/call methods.

8.5 Web3SDK API

Web3SDK API is separated into Web3j API and Precompiled Service API. Through Web3j API we can check the blockchain status, send/inquiry transactions; Precompiled Service API is to manage configurations and to realize some functions.

8.5.1 Web3j API

Web3j API is the RPC API of FISCO BCOS called by web3j object, and it has the same API name with RPC API. Please check RPC API Documentation.

8.5.2 Precompiled Service API

8.5.3 Precompiled Service API

Precomplied contract is an efficient smart contract realized in C++ on FISCO BCOS platform. SDK provides java interface of the precompiled contract, through calling which console can execute commands. For operational reference you can check the guide for console operations. SDK also provides Precompiled Servide class, including PermissionService for distributed permission control, CnsService for CNS, SystemConfigService for system property configuration and ConsensusService for node type configuration. The related error codes are collected here: Precompiled Service API Error Codes

PermissionService

SDK supports distributed permission control. PermissionService can configure permission information. The APIs are here:

- **public String grantUserTableManager(String tableName, String address):** set permissions according to user list name and exterior account address.
- **public String revokeUserTableManager(String tableName, String address):** revoke permissions according to user list name and exterior account address.
- **public List<PermissionInfo> listUserTableManager(String tableName):** inquire permission record list according to the user list name (each record contains exterior account address and effective block number).
- **public String grantDeployAndCreateManager(String address):** grant permission for exterior account to deploy contract and create user list.
- **public String revokeDeployAndCreateManager(String address):** revoke permission for exterior account to deploy contract and create user list.
- **public List<PermissionInfo> listDeployAndCreateManager():** inquire the permission record list for exterior account to deploy contract and create user list.
- **public String grantPermissionManager(String address):** grant permission to exterior accounts for permission control.
- **public String revokePermissionManager(String address):** revoke permission to exterior accounts for permission control

- **public List<PermissionInfo> listPermissionManager():** inquire permission record list of exterior accounts on permission control.
- **public String grantNodeManager(String address):** grant permission to exterior accounts for node management.
- **public String revokeNodeManager(String address):** revoke permission to exterior accounts for node management.
- **public List<PermissionInfo> listNodeManager():** inquire permission records on node management.
- **public String grantCNSManager(String address):** grant permission to exterior account for CNS.
- **public String revokeCNSManager(String address):** revoke permission to exterior account for CNS.
- **public List<PermissionInfo> listCNSManager():** inquire permission records of CNS
- **public String grantSysConfigManager(String address):** grant permission to exterior account for parameters management.
- **public String revokeSysConfigManager(String address):** revoke permission to exterior account for parameters management.
- **public List<PermissionInfo> listSysConfigManager():** inquire permission records of parameters management.

CnsService

SDK supports CNS. CnsService can configure CNS. The APIs are here:

- **String registerCns(String name, String version, String address, String abi):** register CNS according to contract name, version, address and contract abi.
- **String getAddressByContractNameAndVersion(String contractNameAndVersion):** inquire contract address according to contract name and version (connected with colon). If lack of contract version, it is defaulted to be the latest version.
- **List<CnsInfo> queryCnsByName(String name):** inquire CNS information according to contract name.
- **List<CnsInfo> queryCnsByNameAndVersion(String name, String version):** inquire CNS information according to contract name and version.

SystemConfigSerivce

SDK offers services for system configuration. SystemConfigSerivce can configure system property value (currently support tx_count_limit and tx_gas_limit)。 The API is here:

- **String setValueByKey(String key, String value):** set value according to the key（to check the value, please refer to getSystemConfigByKey in Web3j API).

ConsensusService

SDK supports configuration of node type. ConsensusService is used to set node type. The APIs are here:

- **String addSealer(String nodeId):** set the node as consensus node according to node ID.
- **String addObserver(String nodeId):** set the node as observer node according to node ID.
- **String removeNode(String nodeId):** set the node as free node according to node ID.

CRUDService

SDK supports CRUD (Create/Retrieve/Updata/Delete) operations. CRUDService of table include create, insert, retrieve, update and delete. Here are its APIs:

- **int createTable(Table table)**： create table and table object, set the name of table, main key field and other fields; names of other fields are character strings separated by comma; return table status value, return 0 when it is created successfully.
- **int insert(Table table, Entry entry)**： insert records, offer table object and Entry object, set table name and main key name; Entry is map object, offer inserted field name and its value, main key field is necessary; return the number of inserted records.
- **int update(Table table, Entry entry, Condition condition)**： update records, offer table object, Entry object, Condition object. Table object needs to be set with table name and main key field name; Entry is map object, offer new field name and value; Condition object can set new conditions； return the number of new records.
- **List<Map<String, String>> select(Table table, Condition condition)**： retrieve records, offer table object and Condition object. Table object needs to be set with table name and main key field name; Condition object can set condition for retrieving; return the retrieved record.
- **int remove(Table table, Condition condition)**： remove records, offer table object and Condition object. Table object needs to be set with table name and main key field name; Condition object can set conditions for removing; remove the number of removed records.
- **Table desc(String tableName)**： inquire table information according to table name, mainly contain main key and other property fields; return table type, mainly containing field name of main key and other property.

CHAPTER 9

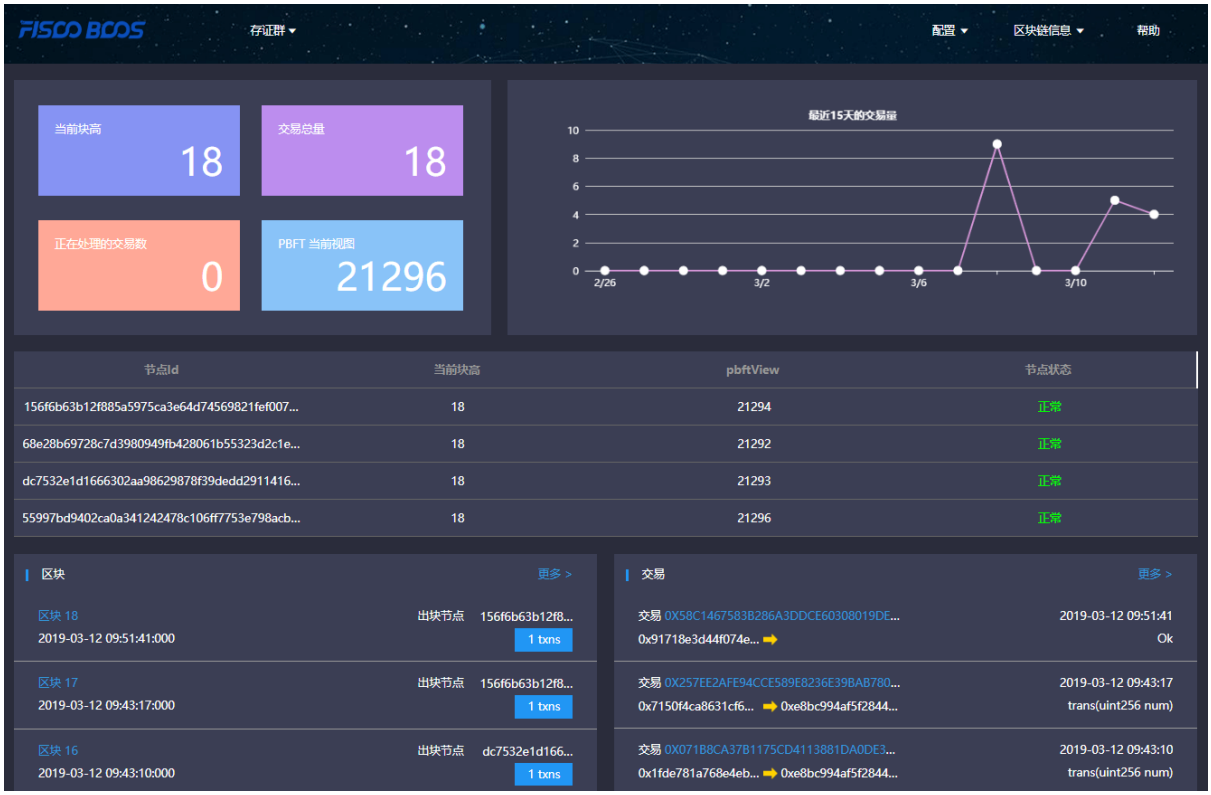
Blockchain explorer

9.1 1. Description

9.1.1 1.1 Basic introduction

This blockchain explorer is adaptable to FISCO-BCOS 2.0.0. FISCO-BCOS 1.2 or 1.3 users please check v1.2.1.

Blockchain explorer is capable of blockchain visualization and real-time presentation. Users can get the information of the blockchain through web pages. This explorer is only adaptable to FISCO-BCOS 2.0. You can learn the newest features in here. Before using this explorer, you may need to learn the groups feature of FISCO BCOS 2.0.



1.2 Main functional modules

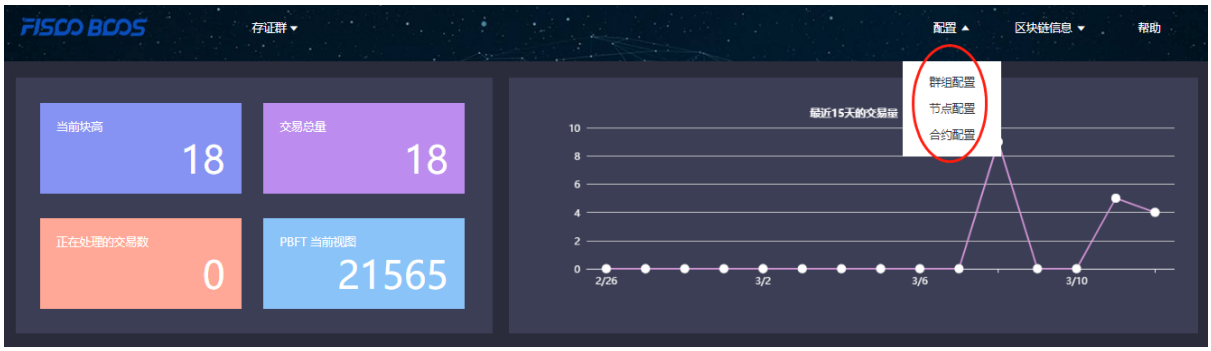
This chapter will give a brief introduction on each module of the browser for all-round understanding. Main functional modules of the blockchain browser includes: group switch module, configuration module and data visualization module.

1.2.1 Group switch module

Group switch module is adopted to access blockchain data when switching to different groups in multi-groups case.

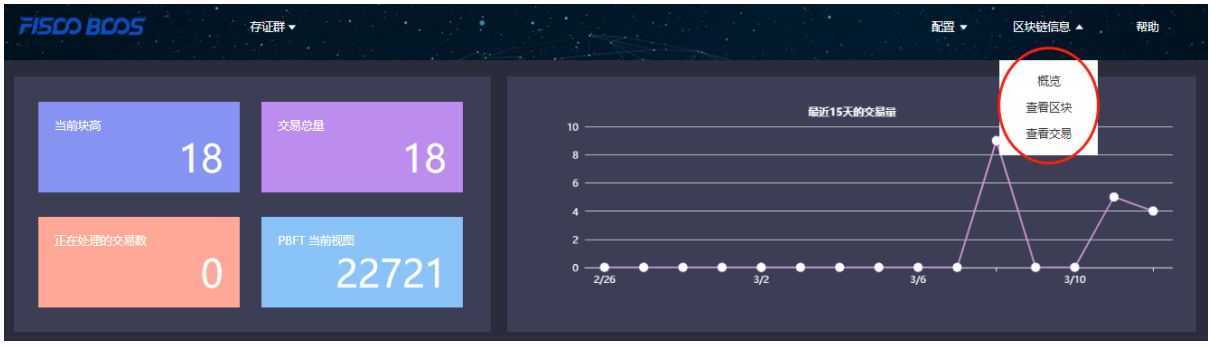
1.2.2 Configuration module

Configuration module includes group configuration, node configuration and contract configuration.



1.2.3 Data visualization module

Blockchain browser demonstrates the detail information of specific group on the chain including: overview, block information and transaction information.



9.2 2. Premises of use

9.2.1 2.1 Group building

Data shown in the blockchain browser is synchronized with blockchain. To synchronize data, initialization configuration (adding group information and node information) is needed. So users have to run a FISCO-BCOS instance and build groups before data synchronizing. FISCO-BCOS 2.0 has provided multiple convenient group building methods.

1. For developers to experience and debug quickly, we recommend the script build_chain.
2. For enterprise applications, FISCO-BCOS generator is a more considerable deployment tool.

The distinguish of the above methods lie in that the script `build_chain` is for better and quicker building experience and it helps developers generate private key of each node in groups; deployment tool doesn't automatically generate private key for safety consideration, and business users need to generate and set by themselves.

9.3 3. Building of blockchain browser

Blockchain browser can be divided into two parts: the back-end service "fisco-bcos-browser" and the front-end web page "fisco-bcos-browser-front".

We also provide two ways for browser building in the current version: **one-key setup** and manual setup.

9.3.1 3.1.1 One-click setup

One-click setup is suitable for single-machine deployment of front-end and back-end to experience quickly. The detail process is introduced in **Installation document**.

9.3.2 3.1.2 Manual setup

Back-end service building

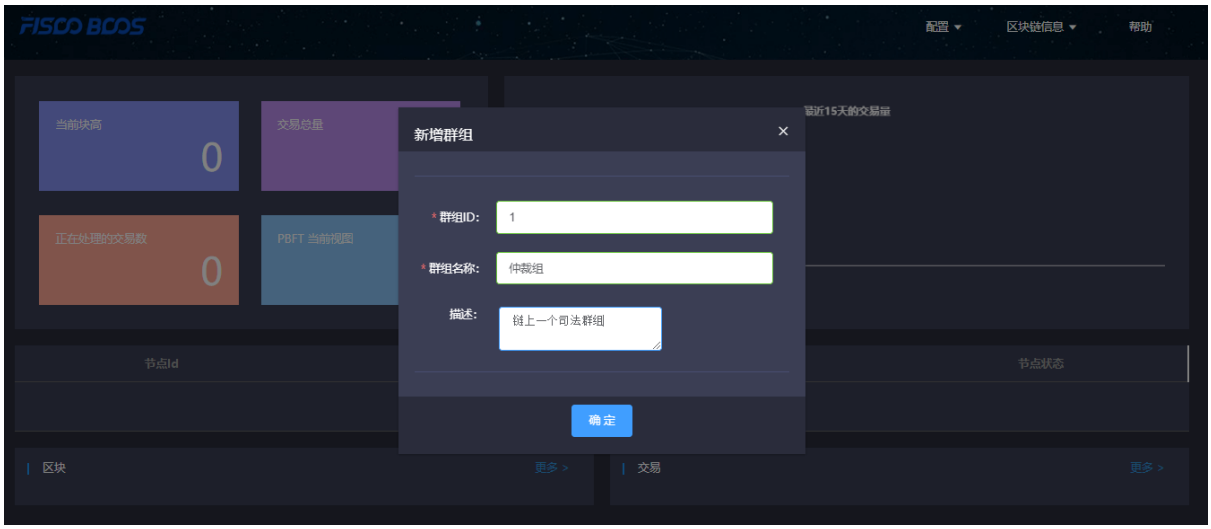
The back-end service of blockchain browser adopts JAVA back-end service, i.e., Spring Boot. The exactly building process can be referred in **Installation document**.

Front-end web page service building

Front-end conducts `vue-cli`. Also, the tutorial can be found in **Installation document**.

9.4 4. Initialization environment

9.4.1 4.1 Adding group



Once it is set up, users can access the front-end by typing IP and its port configured by nginx through web browser. Browser without group initialization will lead to new group configuration page, where the group ID, group name and group description are needed.

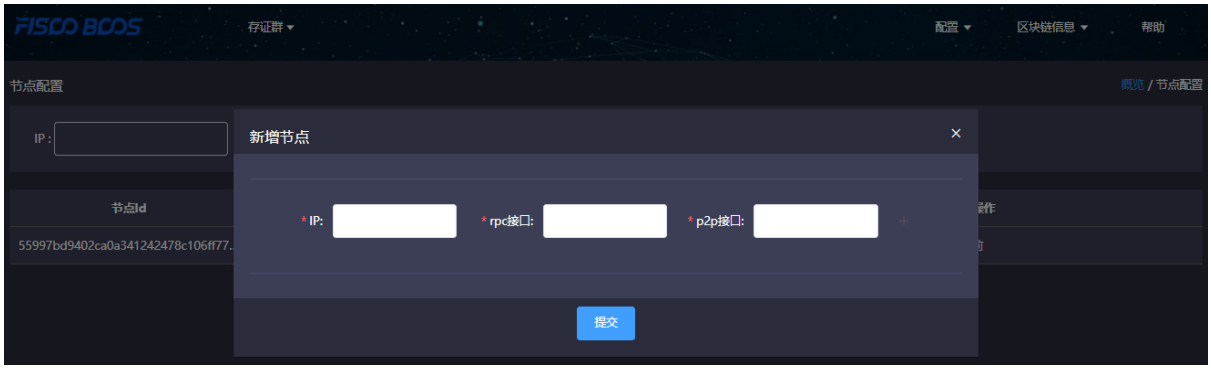
Group ID should be consistent with the specific blockchain. There are many methods to check group ID:

1. acquire rpc interface.
2. console command.

Group name should be meaningful and better understandable as a explanation of group ID.

Group description is the further illustration of the name.

9.4.2 4.2 Adding node



The next step, you need to add the node information belong to the group to obtain relative information shown in blockchain browser. RPC port and P2P port of nodes can be acquired from the file **config.ini** in the directory of a specific node.

For easy use, the newly added group will synchronize the information of shared node which configured by other groups before.

9.4.3 4.3 Adding contract

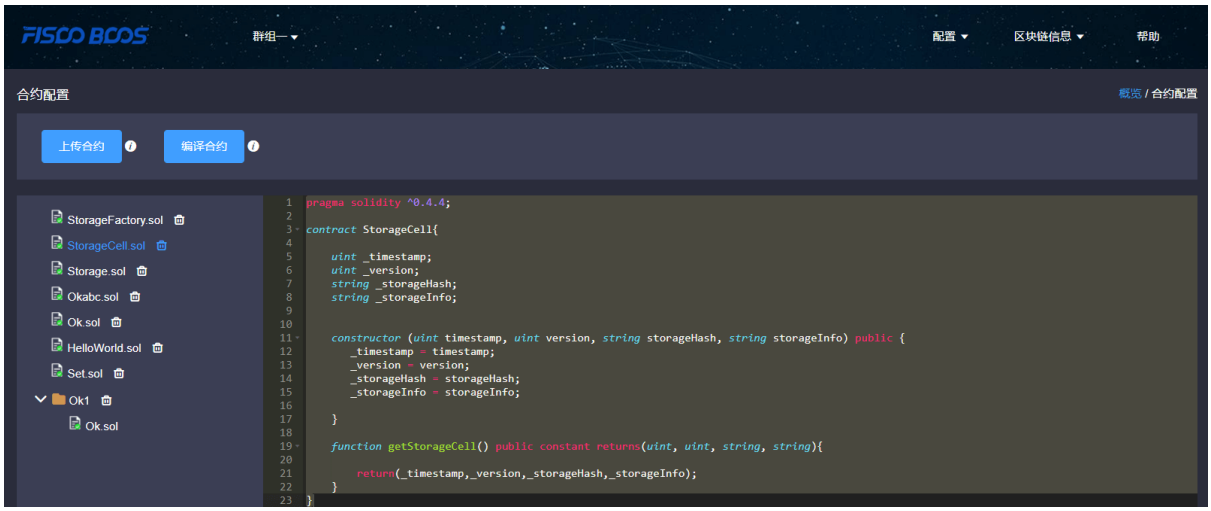
On this version, the browser provides the function of contract analysis, which requires users to import all contracts the group had deployed before. User can upload zip package (only support one-level directory) to solve namesake contract issues.

Steps of import

4.3.1 Import contract

1. Contract is required to be uploaded as `*.sol` file or zip package.
2. Zip package is adaptable to one-level directory at most and defaulted to be uploaded to root directory. Zip package can only contain `*.sol` files.

4.3.2 Compile contract



9.5 5. Functions introduction

9.5.1 5.1 Blockchain overview

5.1.1 Overall overview

Overall overview includes block number of the group, transaction volume, processing transaction amount and the PBFT view.

5.1.2 Transaction volume in 15 days

The transactions of the group in 15 days are shown in the line chart.

5.1.3 Node overview

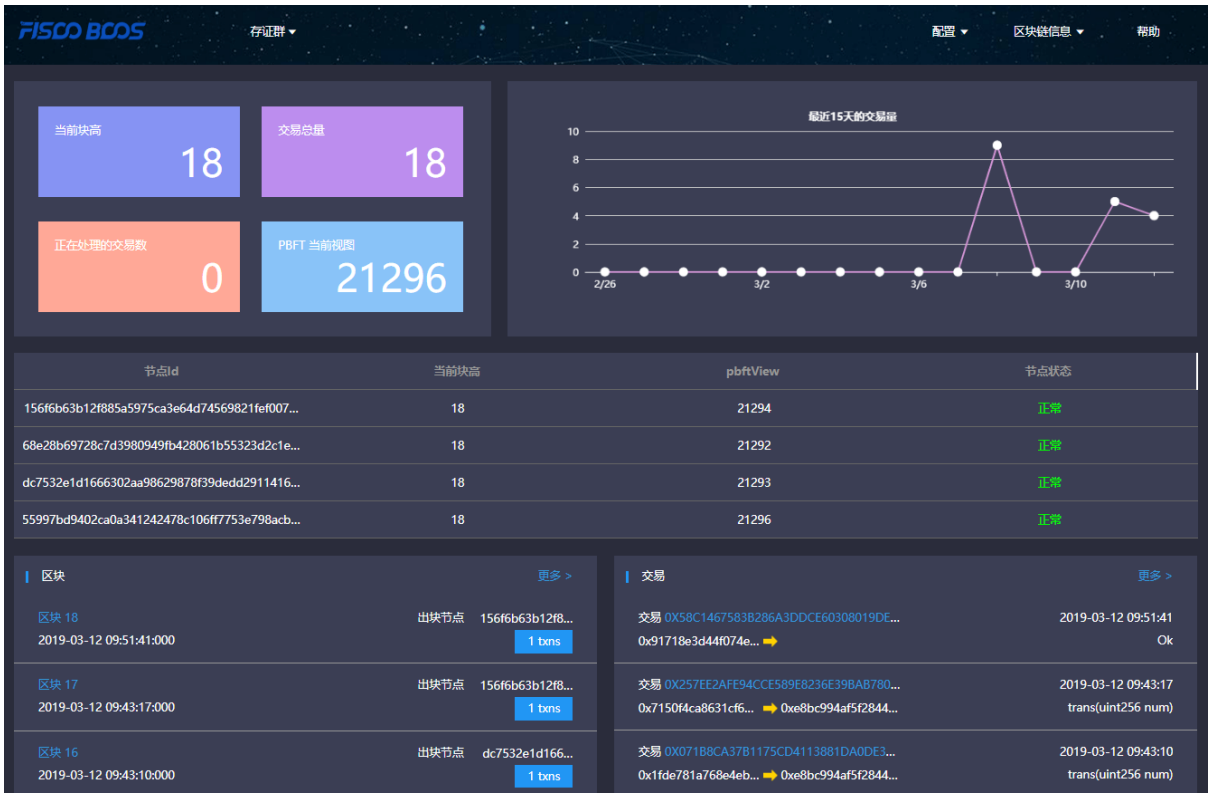
Node overview includes node ID, current block height, the PBFT view and node status.

5.1.4 Block overview

Block overview includes the information of the latest four blocks, including block height, block generator, generation time and transaction volume on the block.

5.1.5 Transaction overview

Transaction overview includes the latest four transactions, including transaction hash, transaction time, transaction sender & receiver. The information invoked by transactions can also be shown if the related contract is imported correctly.



9.5.2 5.2 Block information

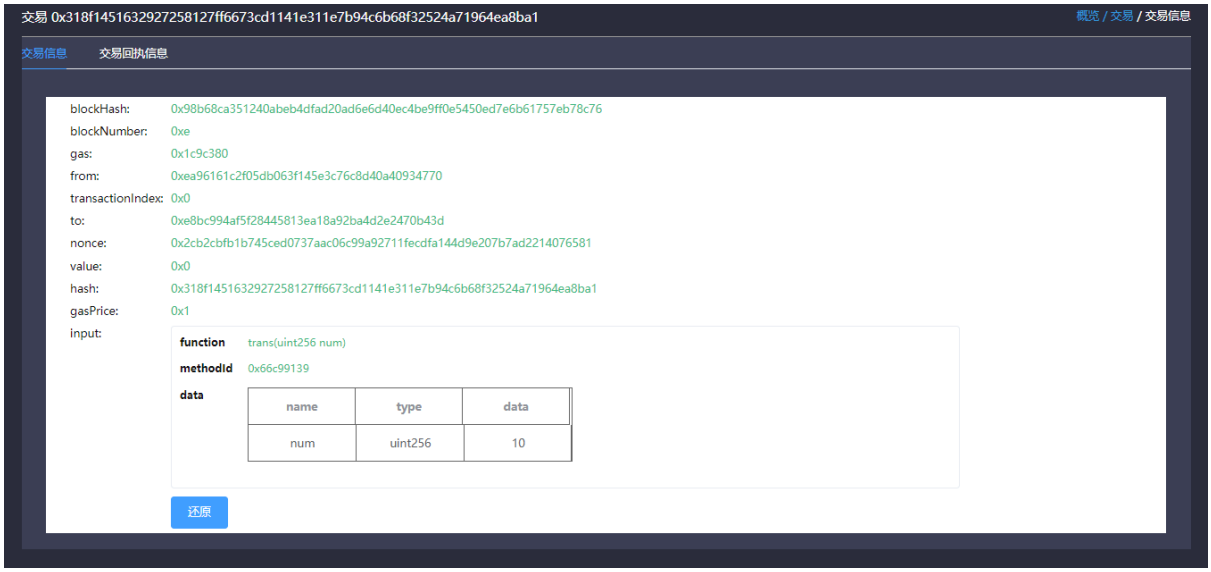
Block information includes pages of block list and block details.

9.5.3 5.3 Transaction information

Transaction information includes pages of transaction list and transaction details.

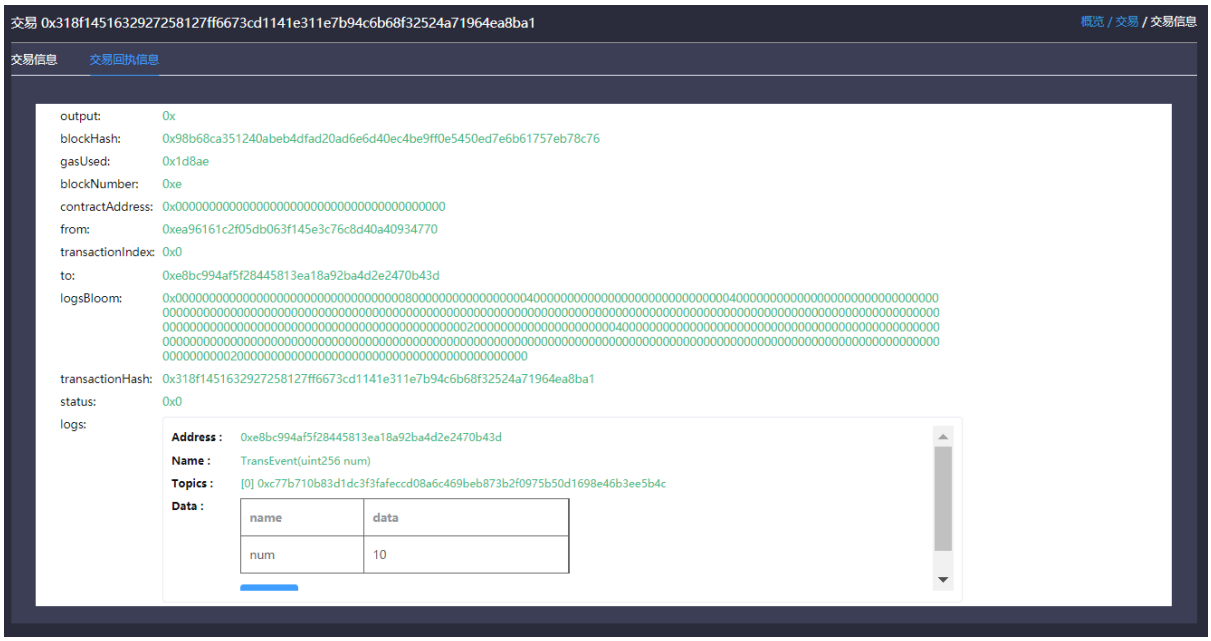
5.3.1 Transaction analysis

After contract is uploaded and compiled, blockchain browser can analyze the transaction method names and parameters. The analysis of the browser is based on correct import of contract. Therefore, when using JAVA or JS to call contract, please **save the correct version of contract.**



5.3.2 Event analysis

After contract is uploaded and compiled, blockchain browser can analyze event method names and parameters in the transaction receipts.



CHAPTER 10

System design

This chapter introduces to the developers on FISCO BCOS the design concept of the platform, including structure and implementation of every model.

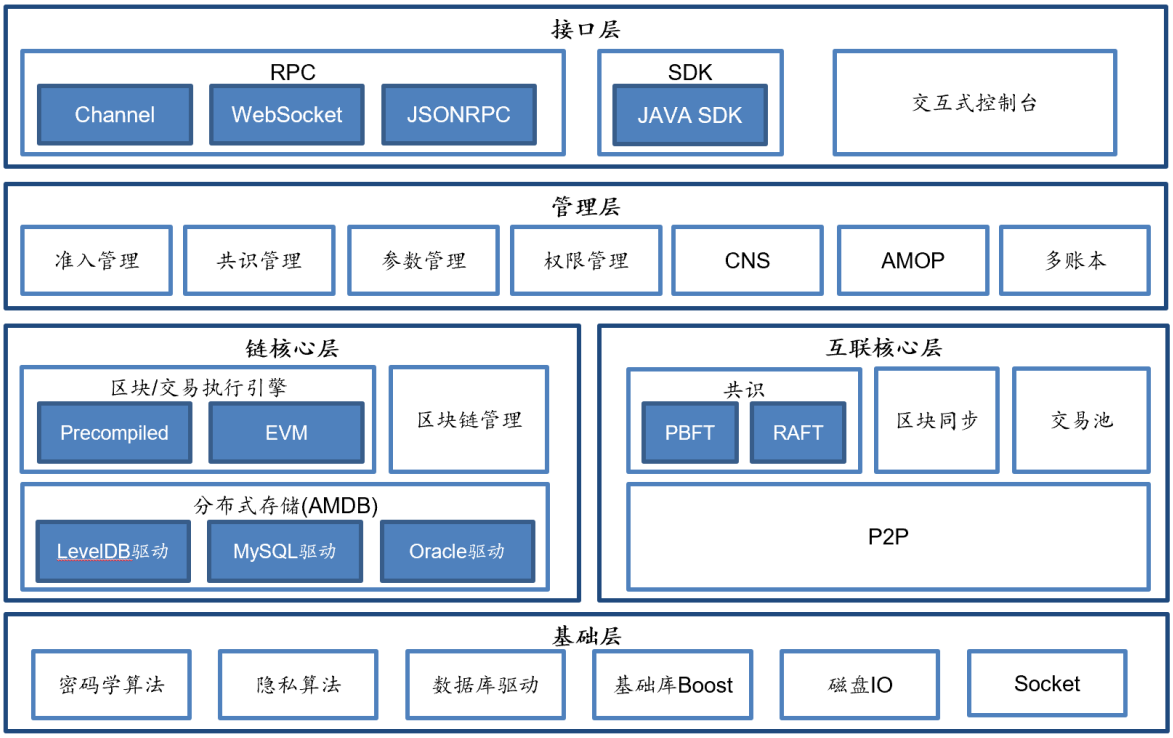
10.1 Overall network

The overall network of FISCO BCOS can be categorized into four layers: fundamental, core, administration and access.

- **fundamental**: provide basic data structure and algorithms library
- **core**: implement the core logic of blockchain. It can be further divided into 2 parts:
    1. Chain core: realize the chain data structure, transaction execution engine and storage driving of Blockchain
    2. Internetworking core: realize the basic P2P networking, consensus mechanism and syncing mechanism of blockchain
- **administration**: realize the administration of blockchain, including parameter setting, ledger management and AMOP
- **access**: access for blockchain users including RPC interface of multiple protocols, SDK and interactive console.

FISCO BCOS boasts strong scalability due to multi-group structure and a strong and stable blockchain system based on reasonable model design.

This chapter emphasizes the group structure and transaction flow (submission, package, execution and write-on-chain of transaction) of FISCO BCOS. empha



10.1.1 Multi-group structure

To fit most business scenarios, FISCO BCOS supports various functions including multi-group activation, transactions among groups, data storage and block consensus in separation engined by multi-group structure. It can safely guard the system privacy but also eliminate the difficulty in operation and maintenance of blockchain system.

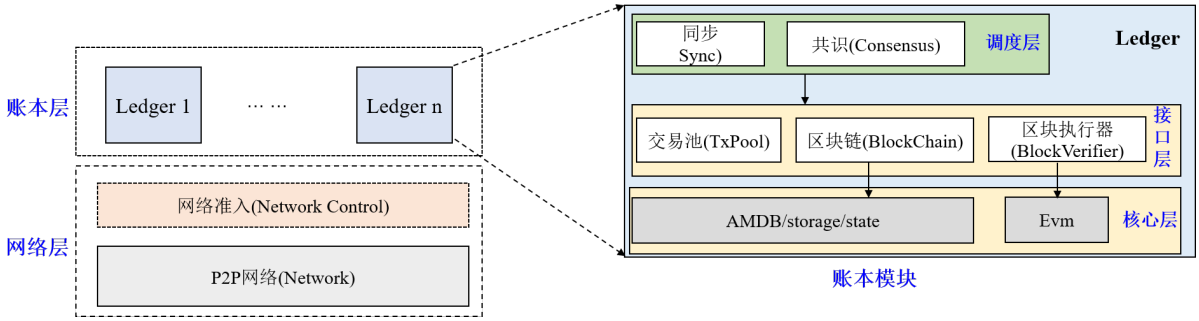
Note: For example:

Agency A, B, C, D constituted a blockchain network to operate Project 1. But now, A and B want to start Project 2 in the condition that C has no access to its data and transactions. How to realize it?

- **1.3 series FISCO BCOS** : Agency A and B build another chain to operate Project 2. Administrator needs to operate and maintain both chains and their ports.
- **FISCO BCOS 2.0** : Agency A and B build another group for Project 2. Administrator maintains only one chain.

Obviously both solutions can achieve privacy protection, but FISCO BCOS 2.0 gets advantages in scalability, operation and maintenance and flexibility.

In multi-group structure, Networking is shared among groups. Groups can isolate messages of some ledger through [networking access and whitelist] (../security_control/node_management.md).

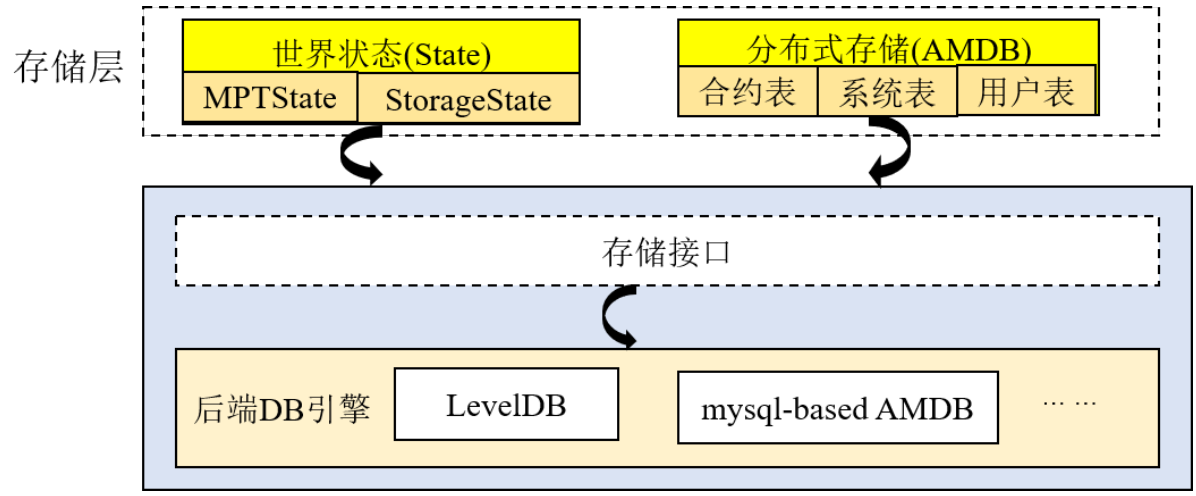


Then, data will be isolated with other groups. Every group runs consensus algorithm independently or differently. Each ledger model contains three layers: Core, Access, Administration (from the bottom to the top). This three layers will cooperate to ensure stable operation of each group in FISCO BCOS platform.

Core

Core layer is responsible for inputing group data block, block information, system table and execution result into data base.

Storage is formed by two parts: State and AMDB. State contains MPTState and StorageState who store the status information of transactions. StorageState has higher performance than MPTState, but it doesn't store history records of block. AMDB opens accesses of select, commit and update and operates contract table, system table, user table. It is pluggable and adaptable to multiple kinds of database. Currently it adapts only to LevelDB database. In the future, mysql-based AMDB will be integrated to the system.



Access

Access layer includes three models: TxPool, BlockChain and BlockVerifier.

- **TxPool**: interact with networking and administration layers. Store the transactions propagated by client ends or other nodes. Administration layer (mainly syncing and consensus models) outputs transactions in TxPool for propagation and block packing.
- **BlockChain**: interact with core layer and administration layer. The only access to bottom storage. Administration layer (syncing and consensus models) can check block number, acquire block and submit block through Blockchain.
- **BlockVerifier**: interact with administration layer, execute the block inputed by administration layer and sends result back to administration layer.

Administration

Administration layer includes two models: Consensus and Sync.

- **Consensus**: include two threads that are Sealer and Engine, one for packing transactions and another for executing consensus workflow. Sealer outputs transactions from TxPool and packs into new blocks. Engine executes consensus workflow, during which block is also executed, and submits execution result to Blockchain. Blockchain will input the information to bottom storage and trigger TxPool to delete all transactions in the on-chain block. and notifies transaction result to client ends using callbacks. Currently FISCO BCOS mainly supports consensus algorithm PBFT and Raft.

- **Sync**: propagate transactions and acquire new block. During consensus workflow, leader is responsible for block packing and leader may be switched at any time. Therefore, it is necessary to ensure that the transactions in client ends will reach every node on chain, whose sync model will propagate the new transaction to the other nodes. Considering the inconsistency of machines' performance on Blockchain, or increment of node may causing lagging of block number, Sync model offers block syncing function. Sync model sends the latest block number to other nodes so that they can download the newest block when finding the block number is lagging behind others.

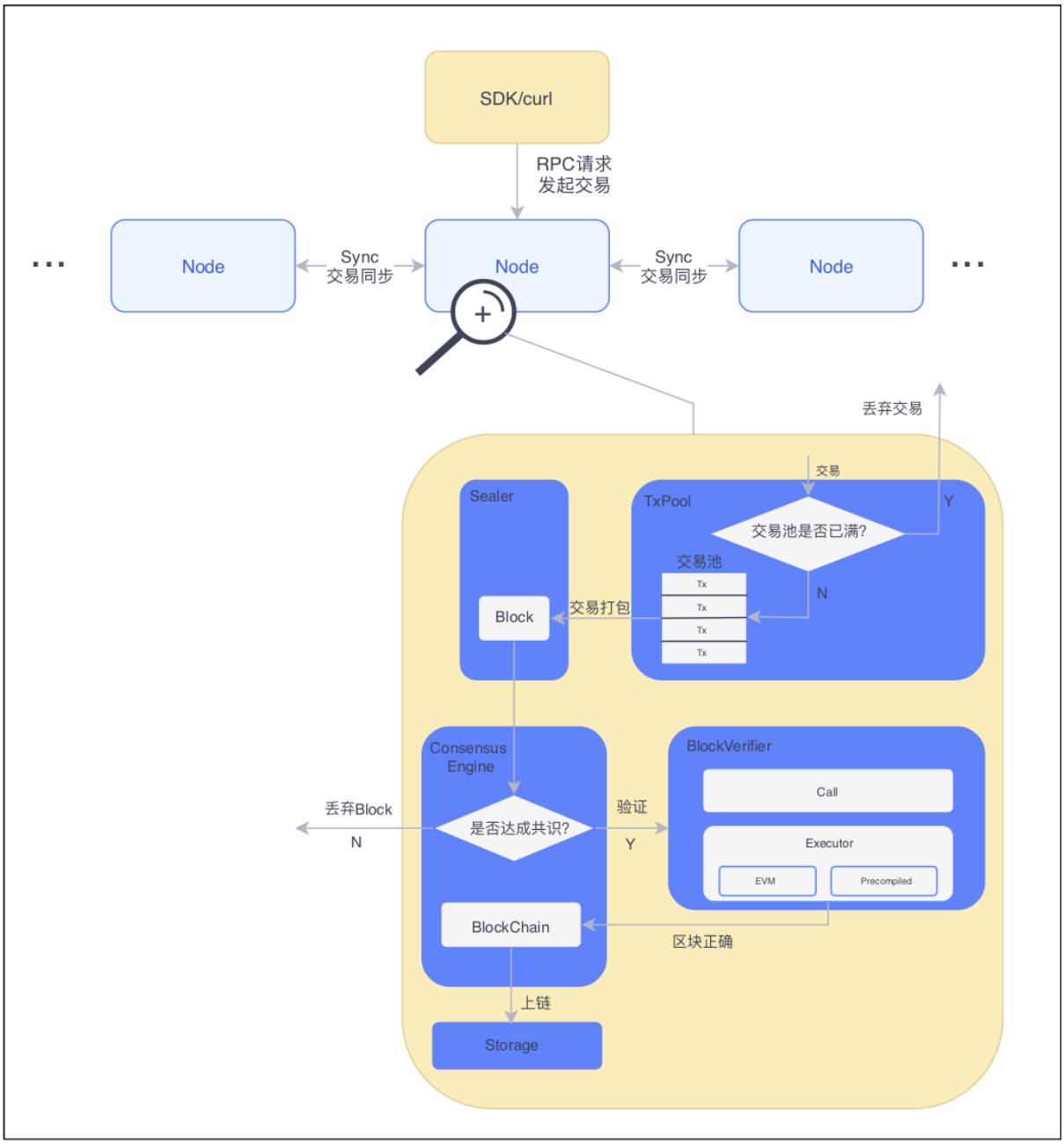
10.1.2 Transaction flow

1 Solution

User sends RPC request to node through SDK or curl command to start transaction. Node adds transaction to TxPool after receiving it. Sealer will constantly take out transactions from TxPool and pack them into block through certain conditions. After the blocks are generated, they will be verified as consensus by consensus engine. If there are no mistakes and consensus is made among nodes, the blocks will be taken on chain. When node using sync model to download missing blocks from other nodes, the execution and verification on blocks will be conducted also.

2 Structure

The overall structure is as below:



Node: Node of block

TxPool: transaction pool, the memory area maintained by node itself for temporarily saving recerived transactions

Sealer: block packager

Consensus Engine: consensus engine

BlockVerifier: block verifier, to verify the correctness of a block

Executor: execution engine, to execute single transaction

BlockChain: administration model of blockchain, the only authorized model. User should input block data as well as execute contextual data before submiting block interface. This administration model will combine the two types of data into one and send to the bottom storage

Storage: the bottom storage

main relations are as followed:

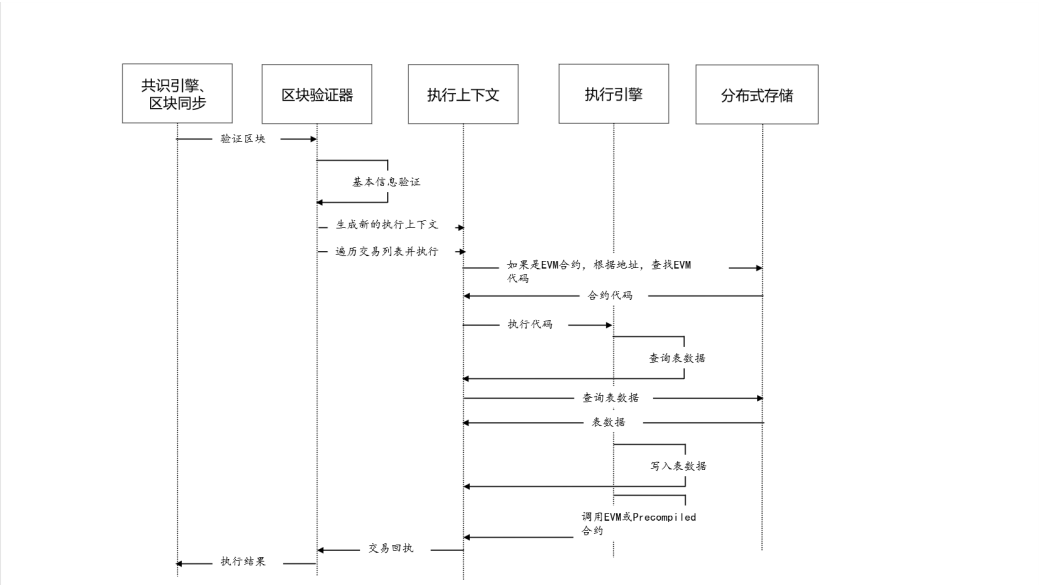
1. User starts transaction through SDK or curl command to the connected node.

2. When node receives transaction, if the current TxPool is not full, node adds transaction to TxPool and propagates to connected nodes; otherwise node discards transaction and output warning notification.
3. Sealer constantly draws transactions from TxPool and packs them into blocks and sends to consensus engine.
4. Consensus engine calls BlockVerifier to verify block and do consensus process. BlockVerifier calls Executor to execute every transaction within blocks. When the block is verified and is commonly agreed by nodes within network, consensus engine will send it to Blockchain.
5. BlockChain receives block and checks the information (block number, etc.), then inputs the block data and table date to bottom storage and moves the block on chain.

3 Process

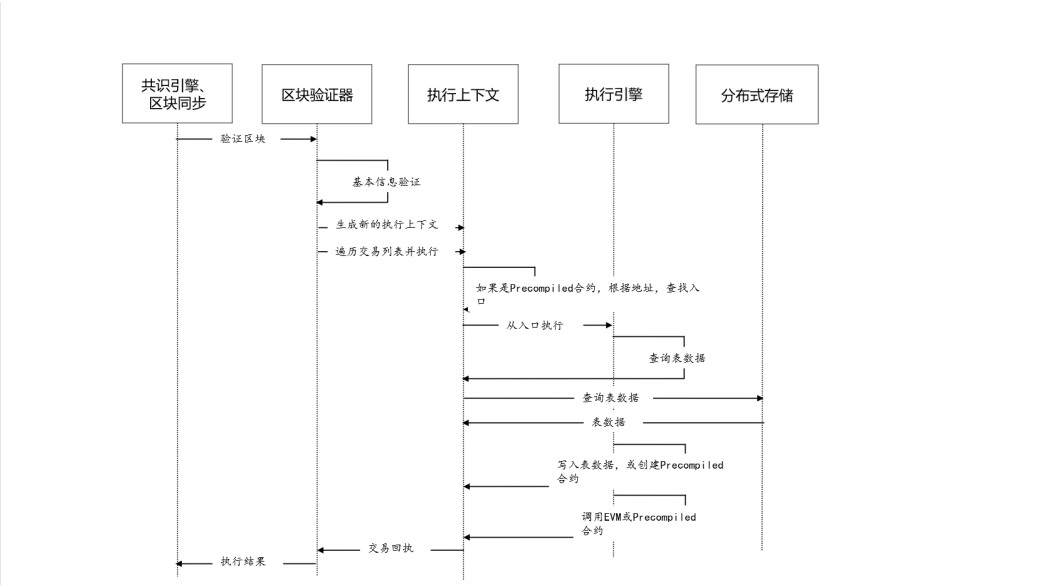
3.1 Contract Executive Process

Executive engine executes single transaction based on Executive Context, which is created by block verifier to cache data generated by executive engine during execution of temporarily-stored blocks. Executive engine supports both EVM contract and precompiled contract. EVM contract can be created through transaction or contract. The executive process is as below:



After the EVM contract is created and saved in table *sys_contracts* of executive context, the address of EVM contract will auto-increment in global state starting from 0x1000001 (customizable). During the execution of EVM contract, Storage variable is saved in table *contract_data*(contract address)_ of executive context.

Precompiled contract can be divided into permanent type and temporary type: (1) permanent precompiled contract, integrated in bottom or components with fixed address; (2) temporary precompiled contract, dynamically created during execution of EVM contract and precompiled contract, the address will auto-increment in executive context starting from 0x1000 and ending at 0x1000000. Temporary precompiled contract is only valid in executive context. Precompiled contract has no Storage variable but has to operate table with process like below:

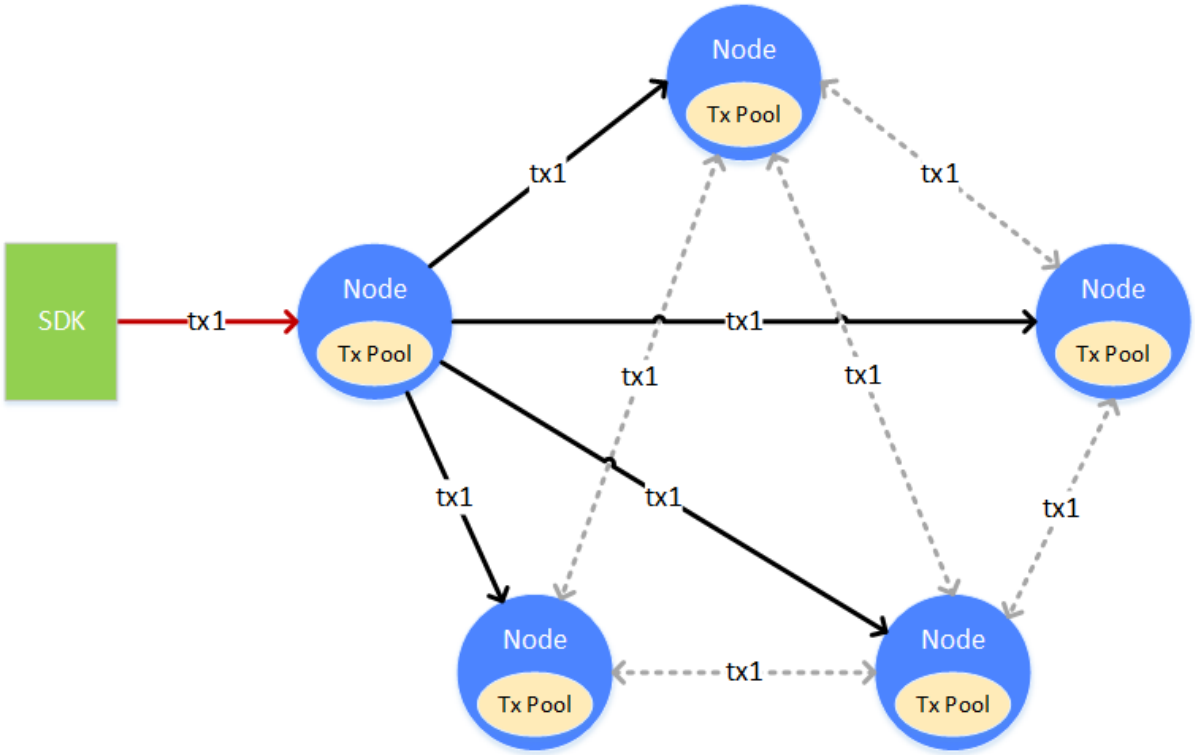


10.2 Syncing

Syncing is a fundamental function of blockchain nodes. It provides necessary conditions for consensus. Syncing can be divided into transaction syncing and status syncing. Transaction syncing ensures each transaction reaches nodes correctly. Status syncing ensures that nodes of lagging blocks can catch up with the newest status. Only nodes with the newest status can join in the consensus mechanism.

10.2.1 Transaction syncing

Transaction syncing makes is possible for transactions to reach every node, laying the foundation for sealing transaction into blocks during consensus process.



Transaction 1 (tx1) start from client end to a node, the node receives and places it into its Tx Pool for consensus process. At the same time, node will broadcast it to other nodes who will place it into their own TxPool either. For there might be miss of transaction during sending, to ensure the arrival, nodes who received transactions can further broadcast to other nodes with certain strategy.

Transaction broadcast strategy

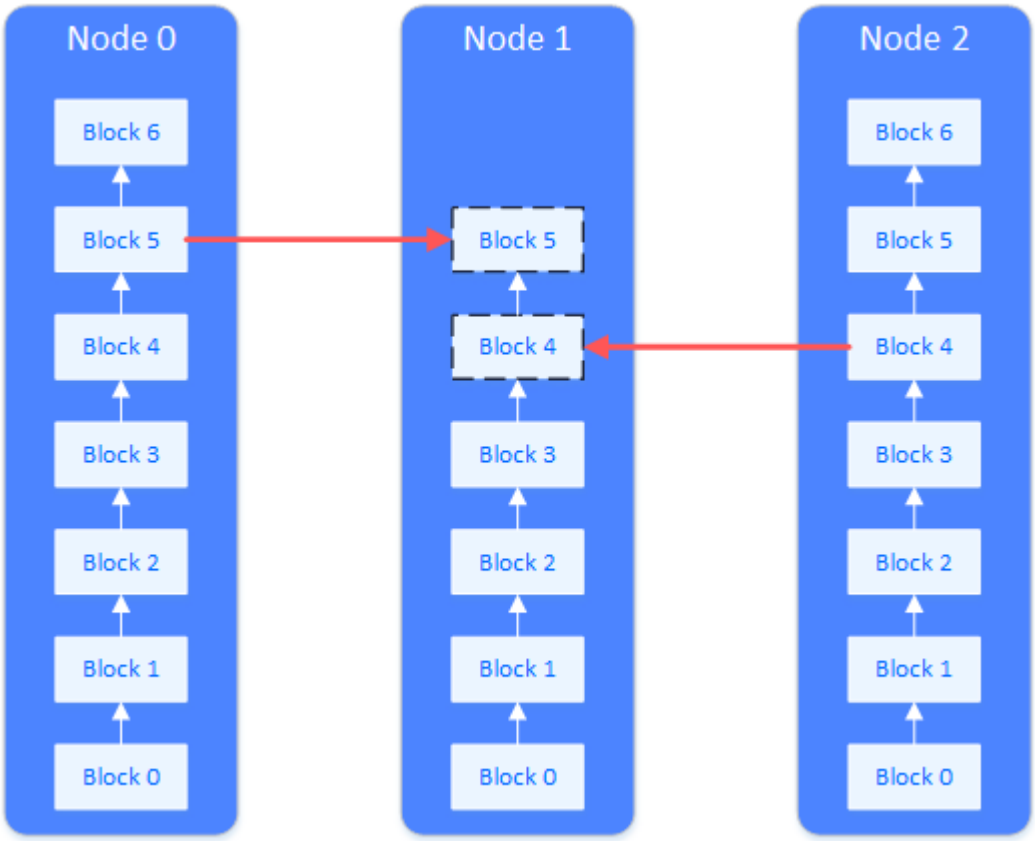
If there is no limit for nodes to transfer and broadcast transactions, the bandwidth will be overloaded, causing avalanche of transaction broadcasting. To prevent from this, FISCO BCOS adapts a delicate transaction broadcast strategy, which can reduce reiteration on the premise of transaction reachability.

- Transactions from SDK will be broadcasted to all nodes.
- Transactions broadcasted by other nodes will be re-broadcasted to nodes selected randomly in a proportion of 25%.
- One transaction on one node will be broadcasted only once. When received an existing transaction, there will not be a second broadcast.

Though the above strategy can ensure reachability of most transactions, the odds of missing still exist, which is admissible. The effort to increase reachability is to guarantee that the transaction can be sealed, in consensus, confirmed and returned with result as soon as possible. Even if a node missed the transaction, it will only effect its execution efficiency but not correctness.

10.2.2 Status syncing

Status syncing is to keep the node status updated. When we talk about the new or old status, it means the status of data on the node, namely the high or low of its block number. If a node has the highest block number on blockchain, then it is in the newest status. Only when a node is in the newest status can it be in consensus and start the consensus process of a new block.

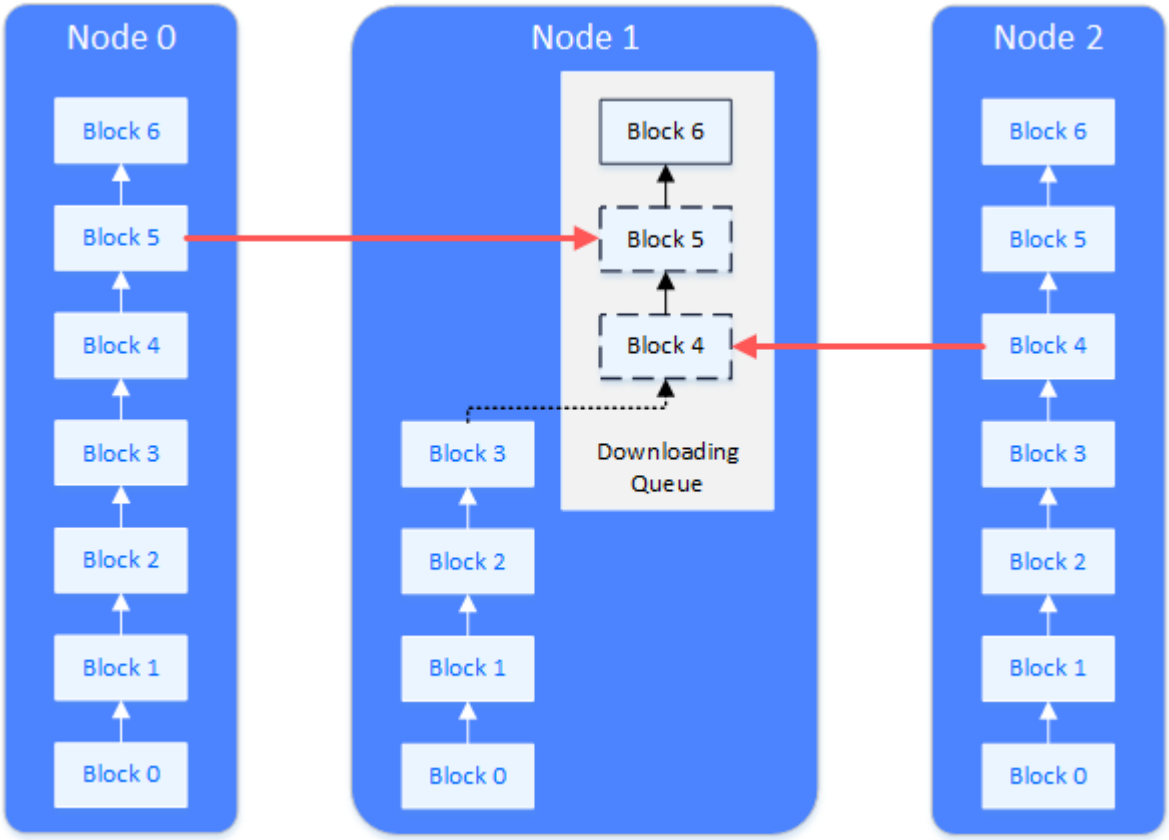


When a new node joined blockchain, or a disconnected node reconnected with the network, the node is lagging behind other nodes and not in the newest status. This is when status syncing is needed. As the picture shows,

Node 1 needs status syncing, so it will request block downloading from other nodes. The load of downloading will be spread to multiple nodes.

Status syncing and downloading queue

During operation, a node will regularly broadcast their highest block number to other nodes. Nodes who received the block number will compare it with their own block number. If a node finds its block number behind others, then it will start request for block downloading. Nodes in downloading process will pick nodes who meet the standard and send blocks for downloading. Nodes who received request will return with relative blocks.



Nodes receive blocks and maintain a download queue locally to buffer and queue up the downloaded blocks. The order of downloading queue is in accordance with block numbers. The downloaded blocks will keep joining the queue. When they are able to connect the local blockchain, they will be taken off the queue and start connection.

10.2.3 Syncing example

Transaction syncing

The process of a transaction being broadcasted to all nodes:

1. A transaction is sent to a node through channel or RPC.
2. The node receives transaction and broadcast it to other nodes.
3. Other nodes receive transaction and select 25% of other nodes to broadcast again to ensure reachability.
4. When a node received the broadcasted transaction, it will not broadcast again.

Status syncing

The logic of broadcasting when a node generating blocks

1. A node generates block.

2. This node broadcasts its newest status (newest block number, hash of the highest block, hash of the Genesis Block) to all nodes.
3. Other nodes receive the status of the peer node and update the peer data maintained locally.

Syncing of group members

A member of a group is accidentally closed at some point when other members were generating blocks. This member reconnects with the network but finds its block number behind other members:

1. The member reboots itself.
2. The member receives the status package from other members.
3. The member compares and finds its block number behind other members, then starts downloading block.
4. The member divides the missing blocks into multiple downloading requests and sends to members to even out load.
5. The member waits for return of blocks from other nodes.
6. Other nodes receive request and search the block from its blockchain and return to the rebooted node.
7. The node receives block and places it into downloading queue.
8. The node takes the block out from the downloading queue and writes it on blockchain.
9. If the downloading is unfinished, it will keep sending requests. If finished, it will switch its status and start transaction syncing and consensus process.

Syncing of a new member

When a new node has joined group, and it's its first start, the new member would need to synchronize with the block from other members:

1. non-member node is started before being registered by the group.
2. being outside of the group, the node won't do state or transaction broadcasting but wait for another member's status message.
3. with the new member not registered, members won't broadcast their status to it.
4. the manager adds new member to the group.
5. members broadcast their status to the new member.
6. the new member receives the block message and compares with its block number (which is 0), and start downloading.
7. the following process is the same as block syncing among group members.

10.3 Consensus algorithm

Blockchain system adopts consensus algorithm to ensure consistency. Theoretically, consensus is the process of commonly agreeing on a certain proposal. In distributed system, proposal is defined in a broad sense, which includes the sequence of event, who will be the leader and so on. In blockchain system, consensus is the process for consensus node to agree on transaction results.

Types of consensus algorithm

According to its tolerance on Byzantine Fault , consensus algorithms can be divided into Crash Fault Tolerance, CFT and Byzantine Fault Tolerance, BFT:

- **CFT algorithms** ： regular fault-tolerant algorithms, when it occurs to system malfunctions in network, disk or server crash down, they can still reach agreement on a proposal. Classic CFT algorithms include Paxos and Raft which has better performance and efficiency and tolerate less than a half of malfunction nodes;

- **BFT algorithms**： Byzantine fault-tolerant algorithms, besides regular malfunctions happen during consensus, it can tolerate Byzantine fault like node cheating (faking execution result of transaction, etc.). Classic BFT algorithm includes PBFT, which has lower performance and tolerates less than one third of malfunction nodes.

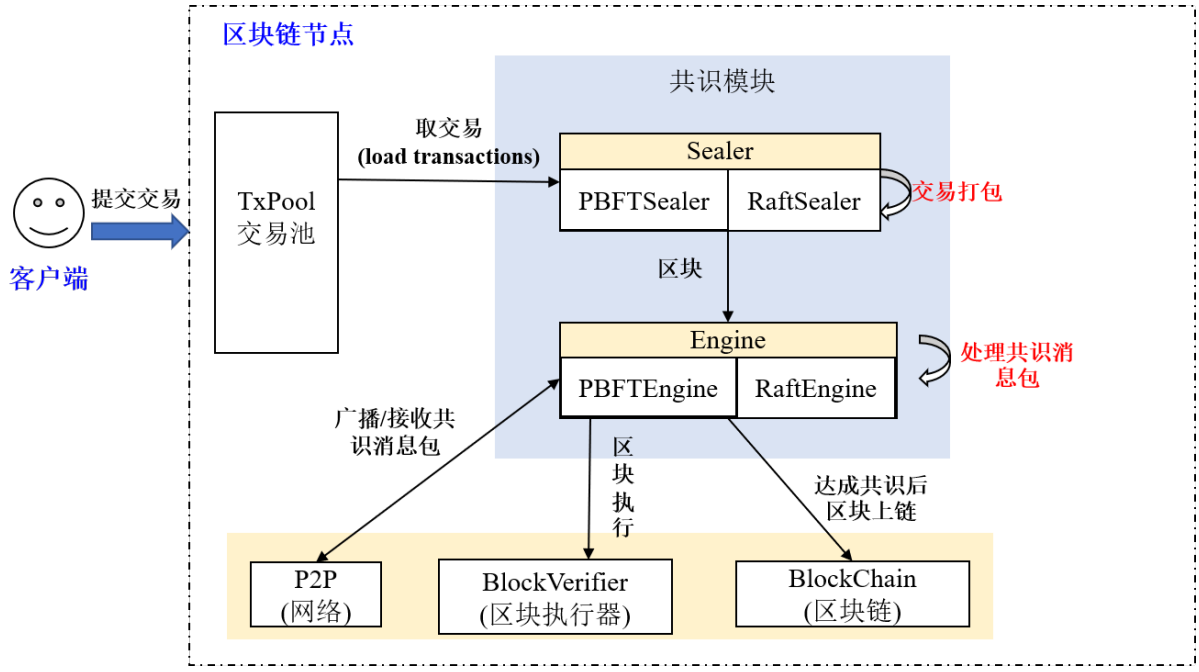
FISCO BCOS consensus algorithm

FISCO BCOS realized pluggable consensus algorithm based on multi-group structure. With different group running different consensus algorithms, consensus processes are independent in each group. Currently, FISCO BCOS supports PBFT (Practical Byzantine Fault Tolerance) and Raft (Replication and Fault Tolerant) consensus algorithms:

- **PBFT algorithm**: BFT algorithm, tolerate less than one third of malfunction nodes and malicious nodes, able to reach final consistency;
- **Raft algorithm**: CFT algorithm, tolerate half of multifunction nodes except malicious nodes, able to reach consistency.

10.3.1 Framework

FISCO BCOS realized a extensible consensus framwork with pluggable consensus algorithm. Currently it supports **PBFT(Practical Byzantine Fault Tolerance)** and **Raft(Replication and Fault Tolerant)** algorithm. The consensus model framework is as below:



Sealer thread

Sealer thread takes transaction out from txPool and seal transactions based on the highest block of nodes to generate new block and send it to engine thread. The sealer threads of PBFT and Raft are respectively PBFTSealer and RaftSealer.

Engine thread

Engine thread receives the new block locally or through internet, and finishes consensus process according to the received consensus information, and finally writes the consensused new block to blockchain, after which the transaction will be deleted from txPool. The engine threads of PBFT and Raft are respectively PBFTEngine and RaftEngine.

10.3.2 PBFT

PBFT(Practical Byzantine Fault Tolerance) consensus algorithm is still workable when it comes to minority malicious nodes (like faking message), as its message delivery system is tamper-resistant, non-forgeable and non-deniable due to digital signature, signature verification, hash and other cryptographic algorithms. Besides, it has optimized the past achievement by reducing the level of complexity of BFT from exponential to polynomial. In a system formed by (3*f+1) nodes, as long as there is more than (2*f+1) non-malicious nodes, it can reach consistency. For example: a system of 7 nodes can allow 2 nodes of Byzantine Fault.

FISCO BCOS system has realized PBFT consensus algorithm.

1. Core concepts

Node type, node ID, node index and view are core concepts of PBFT consensus algorithm. The basic concepts of blockchain system is introduced in Core concept.

1.1 Node type

- **Leader/Primary**: consensused node responsible for sealing blocks of transaction and block consensus. Each round of consensus will contain only one leader, who will be switched after a round to prevent it from faking block;
- **Replica**: replica node responsible for block consensus, each round of consensus contains multiple replica nodes with similar process each;
- **Observer**: observer node responsible for acquiring new block from consensused nodes or replica nodes, executing and verifying result, and adding the new block to blockchain.

Leaders and Replicas are named consensus nodes.

1.2 Node ID && node index

To prevent node from being malicious, each consensused node during PBFT process signs the message they send, and does signature verification on received message. Therefore, each node maintains a public and private key pair. Private key is to sign on the message it sends; public key as the node ID for identification and signature verification.

> **Node ID** : public key for signature and the unique identification of consensused node, usually a 64-byte binary string, other nodes verifies the message package by the sender node ID.

Considering the length of node ID would take up bandwidths if containing this field in consensus message, so FISCO BCOS adapts node index through which each node ID can be located in the consensused node list maintained by each node. When sending message package, by the input node index other nodes can search out node ID for signature verification:

> **Node index** : the location of each consensused node in the public node ID list

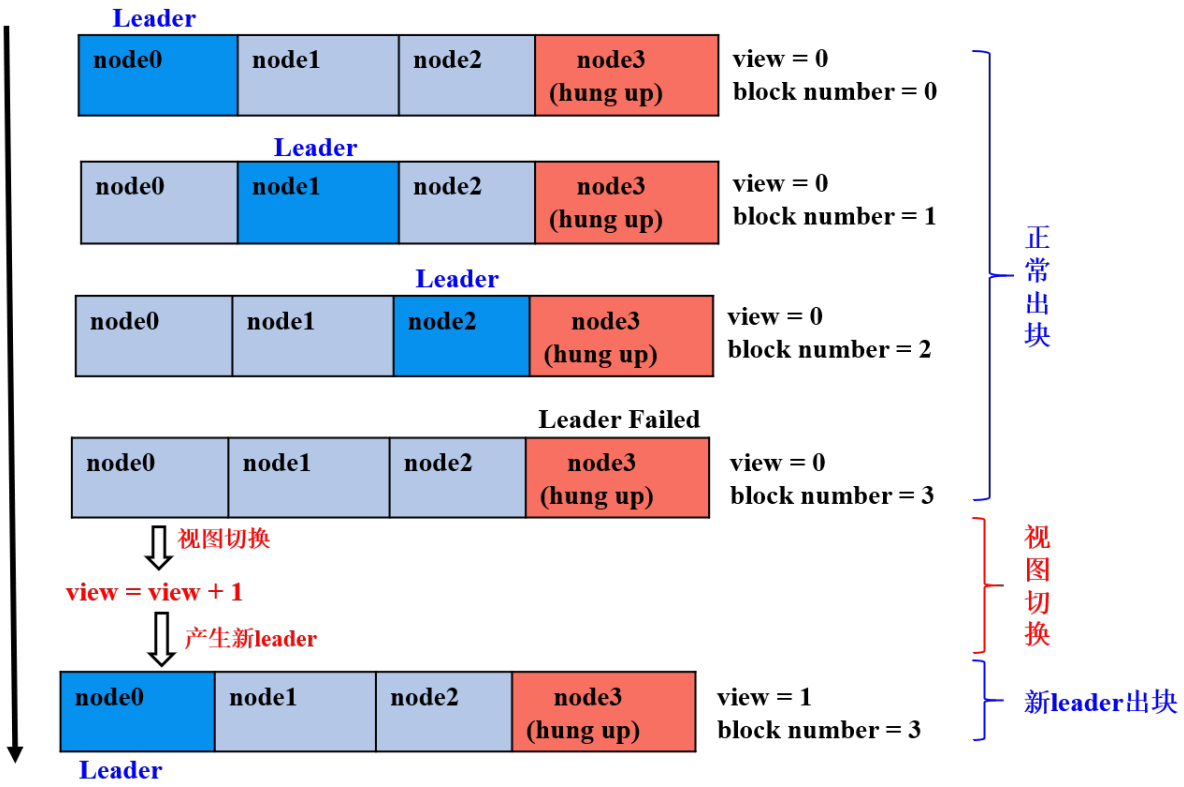
1.3 View

View is adapted in PBFT consensus algorithm to record consensus status of each node. Nodes with the same view maintain the same node list of Leader and Replicas. When Leader fails, the view will be switched. If it is switched successfully (at least 2*f+1 nodes have the same view), a new Leader will be picked according to the new view and start generating block. If not, it will keep switching until most nodes (equal or more than 2*f+1) reach the same view.

In FISCO BCOS system, the computing formula of leader index is:

```
leader_idx = (view + block_number) % node_num
```

The following drawing shows the switch of view in a FISCO BCOS system of `4(3*f+1, f=1)` nodes where node 3 is Byzantine node:



- the former 3 rounds of consensus: node 0, 1, 2 are leader nodes, `2*f+1` non-malicious nodes, nodes generates block in normal consensus status;
- the 4th round of consensus: node 3 is the leader and a Byzantine node, node 0, 2 doesn't receive the block seal from node 3 on time, the view is being switched to new view `view_new=view+1` and broadcast viewchange package, when the `(2*f+1)` viewchange packages in `view_new` are fully collected, nodes will switch it as the new view `view_new` and calculate the new leader;
- the 5th round of consensus: node 0 is the leader and keeps sealing blocks.

1.4 Consensus messages

PBFT model mainly includes **PrepareReq, SignReq, CommitReq and ViewChangeReq** 4 consensus messages:

- **PrepareReqPacket**: includes request package of block, leader generates it and broadcasts to all replica nodes who receives Prepare package and verifies PrepareReq signature, executes block and caches the execution result, in order to prevent Byzantine nodes from doing evil and ensure the certainty of the block execution result;
- **SignReqPacket**: signature request with block execution result, generated by consensused node after receiving Prepare package and executing block, SignReq request contains the hash and its signature of executed block, which are SignReq.block_hash and SignReq.sig, node broadcast SignReq to other consensus nodes for the consensus of SignReq (or block execution result);
- **CommitReqPacket**: commit request to confirm block execution result, generated by nodes who fully collected `(2*f+1)` SignReq request with the same block_hash and from different nodes. When CommitReq is broadcasted to other consensused nodes, which will add the latest block cached locally on chain after fully connecting `(2*f+1)` CommitReq requests with the same block_hash and from different nodes;
- **ViewChangeReqPacket**: request to switch view, when leader fails (networking abnormality, server crash down, etc.), other nodes will start switching view, ViewChangeReq includes the new view (marked as

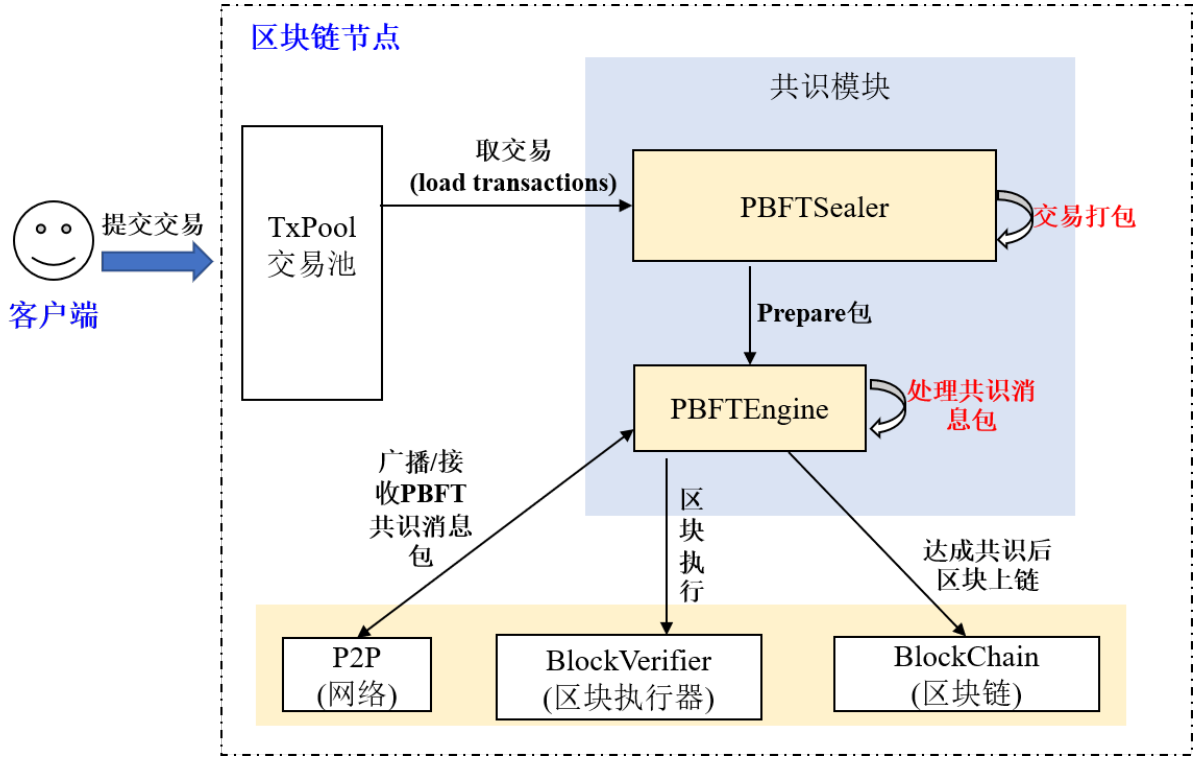
toView, the current view plus one), a node will switch its current view to toView after collecting `(2*f+1)` ViewChangeReq requests with toView and from different nodes.

Fields contained in the 4 types of messages are almost the same:

PrepareReqPacket includes information of in-process block:

2. System framework

The system framework is described as below:



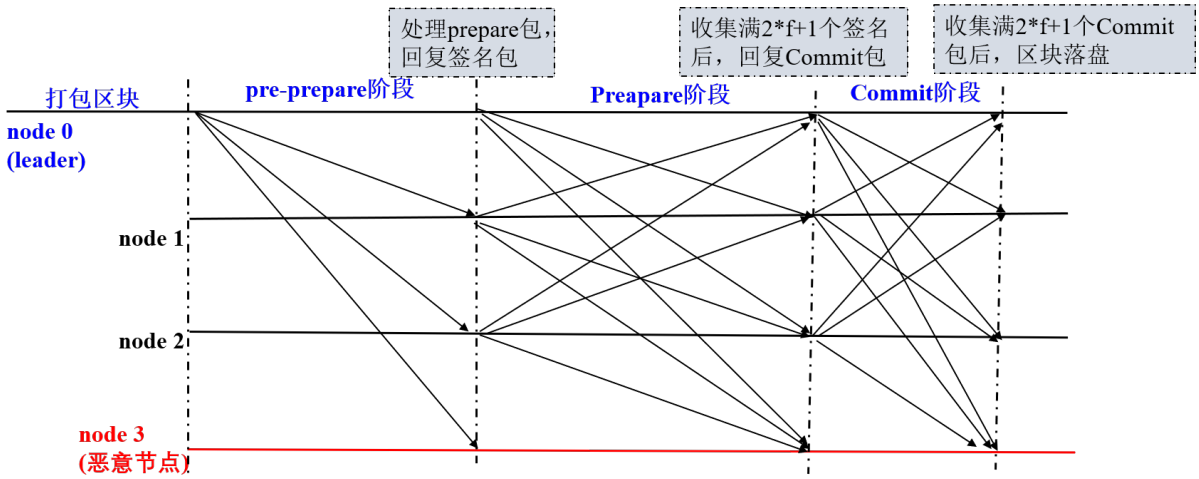
PBFT consensus process mainly contains 2 threads:

- PBFTSealer: PBFT sealer thread taking transaction out of txPool, encapsulating the sealed block into PBFT Prepare package and sending the package to PBFTEngine;
- PBFTEngine: PBFT consensus thread receiving PBFT consensus message packet from PBFT sealer or P2P network to finish consensus process, writing the consensused block to blockchain and deleting the transaction from txPool, Blockverifier executes block.

3. Core process

PBFT consensus process includes 3 phases, Pre-prepare, Prepare and Commit:

- **Pre-prepare**: executes block, generates signature package and broadcast it to all consensused nodes;
- **Prepare**: collects signature package, when a node collects `2*f+1` signature packages, it will state that it is ready from committing blocks and start broadcasting Commit package;
- **Commit**: collects Commit package, when a node collects `2*f+1` Commit packages, it will commit the locally-cached latest block to data base.

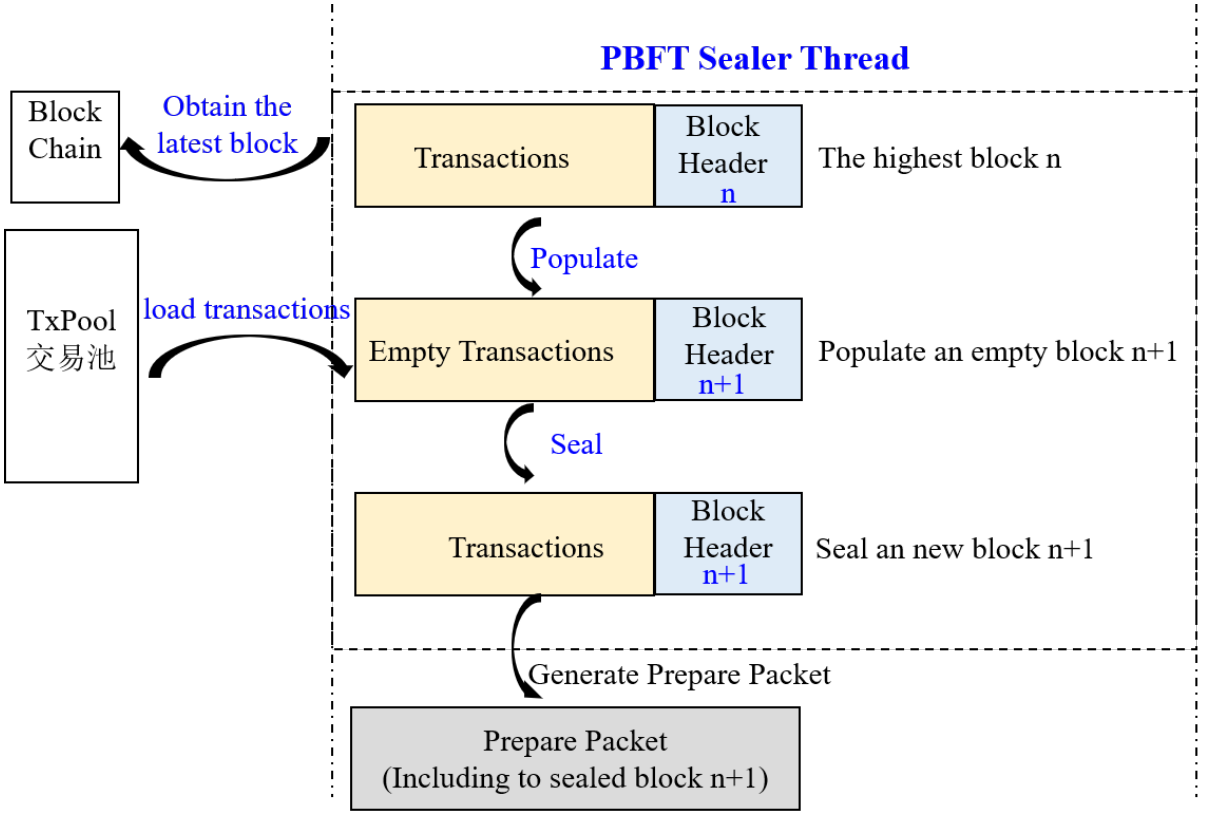


The following picture introduces the detail processes of each phase of PBFT:

3.1 Leader to seal block

In PBFT consensus algorithm, consensus nodes generate blocks in turn, each round of consensus has one leader to seal block. Leader index can be calculated through formula `(block_number + current_view) % consensus_node_num`.

Node starts sealing block when finding that the leader index is the same with the index of itself. Block sealing is mainly conducted by PBFTSealer thread, with detail functions as below:



- **generate new empty block**: acquire the latest block on blockchain, based on which new empty block will be generated (set the parent hash of the new block as the hash of the highest block, time stamp as the current time, delete transaction);
- **seal transaction from txPool**: acquire transaction from txPool after the new empty block is generated, and

insert the transaction to the new block;

- **encapsulate new block**: Sealer thread seal the transaction and sets the sealer of the new block as self index, and calculate transactionRoot of all transactions according to the sealed transaction;
- **generate Prepare package**: encode the encapsulated new block to Prepare package, broadcast to all consensused nodes in group through PBFTEngine thread, other nodes receive Prepare package and start 3 phases of consensus.

3.2 Pre-prepare phase

Consensused nodes enters pre-prepare phase after receiving Prepare package. The workflow of this phase includes:

- **Prepare package validity judgments**：judge whether the Prepare package is replicated, whether the parent hash of block in Prepare request is the hash of the highest block currently (to avoid forking), whether the block number in Prepare request equals the latest block number plus one;
- **cache valid Prepare package**: if the Prepare request is valid, cache it locally to filter replicated Prepare requests;
- **Empty block judgement**：if the transaction quantity in the block contained in Prepare request is 0, start view switching of empty block by adding one, and broadcast view switching request to other nodes;
- **execute block and cache execution result**: if the transaction quantity in the block contained in Prepare request is more than 0, call BlockVerifier to execute block and cache the executed block;
- **generate and broadcast signature package**：generate and broadcast signature package based on the hash of executed block, state that this node has finished block execution and verification.

3.3 Prepare phase

Consensus nodes enter Prepare phase after receiving the signature package. The workflow of this phase is as below:

- **signature package validity judgment**：judge whether the hash of the signature package is the same with the hash of executed block cached in Pre-prepare phase, if not, judge whether the request belongs to future block (generation of future block is caused by lower performance of the node, who is still in the last round of consensus, to judge whether it's future block: height field of the signature package bigger than the latest block number plus one); if not future block, it is invalid signature request which will be denied by node;
- **cache valid signature package**：node will cache valid signature package;
- **judge whether the cached signature packages of block cached in Pre-prepare phase reach `2*f+1`, commit packages if they are fully collected**：if the quantity of signature packages of the block hash cached in Pre-prepare phase exceeds `2*f+1`, then most nodes has executed the block and get the same result, and the node is ready to commit block and broadcast Commit package;
- **write to disk the Prepare package cached in Pre-prepare phase for backup if signature packages are fully collected**：to avoid more than `2*f+1` nodes crashing down before committing block to data base in Commit phase, the crashed nodes will again generate blocks after re-started, which will cause forking (the highest block of these nodes are different with the latest block of other nodes), therefore, it's needed to backup the Prepare package cached in Pre-prepare phase to data base, so nodes can process the backup Prepare package first after re-started.

3.4 Commit phase

Consensus nodes enter Commit phase after receiving Commit package. The workflow of this phase includes:

- **Commit package validity judgment**：mainly judge whether the hash of Commit package is the same with the block hash cached in Pre-prepare phase, if not, judge whether the request belongs to future block (the generation of future block is caused by lower performance of the node, who is still is the last round of

consensus, to judge whether it is future block: height field of Commit is bigger than the highest local block number plus one); if it's not future block, it is invalid Commit request which will be denied by node;

- **cache valid Commit package**： nodes cache valid Commit package;
- **judge whether the cached Commit packages of block cached in Pre-prepare phase reach `2*f+1`, write the new block to disk if the Commit packages are fully collected**： if the number of Commit requests of the block hash cached in Pre-prepare phase reaches `2*f+1`, most nodes are ready to commit block and get the same execution result, call BlockChain to write the block cached in Pre-prepare phase to data base.

3.5 View switching process

When the 3 consensus phases of PBFT is time-out or nodes receive empty block, PBFTEngine will try to switch to higher view (the new view toView plus one) and start ViewChange process; nodes will also start ViewChange process when receiving ViewChange package:

- **ViewChange package validity judgment**: block number in a valid ViewChange request shouldn't be less than the highest block number currently, the view should be higher than current node view;
- **cache ViewChange package**： avoid repeated process of the same ViewChange request, also the evidence to check if the node can switch view;
- **collect ViewChange package**： if the view in received ViewChange package equals the new view toView and the node has collected `2*f+1` ViewChange packages from different nodes whose views equal toView, then more than 2 thirds of nodes should be switched to toView; otherwise at least i third of nodes have other views, then the view of this node should be switched to the same with these nodes.

10.3.3 Raft

1 Definitions

1.1 Raft

Raft（Replication and Fault Tolerant）is a partition tolerant consistency protocol, which ensures the system consistency in a N-node system with (N+1)/2 (ceil to int) working nodes. For example, a 5-node system allows 2 nodes to appear Byzantine error, like node crashing down, network partition, message delay. Raft is easier to understand than Paxos, and it is proved to have the same fault tolerance and performance with Paxos. For detail introduction please check the website and dynamic demonstration.

1.2 Node type

In Raft algorithm, each node owns one of the three identities: **Leader**, **Follower** and **Candidate**:

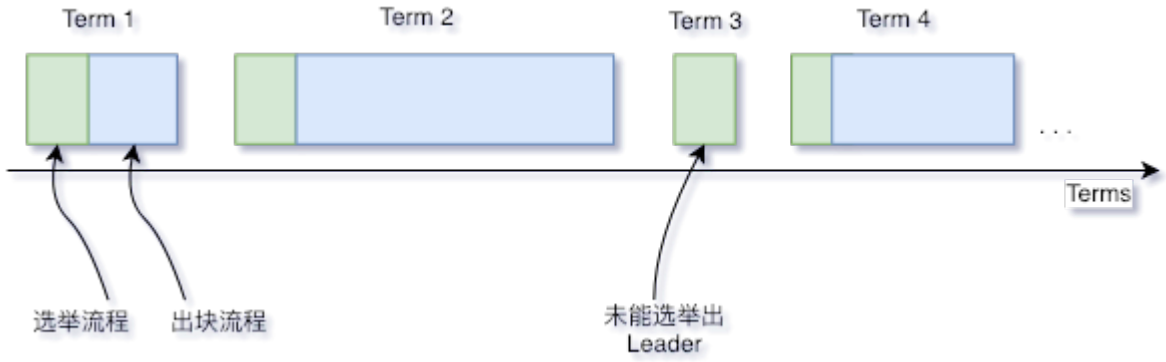
- **Leader**： interact with outsider, elected by Follower nodes, each consensus round contains one and only Leader node, who is responsible for taking transaction out of txPool, sealing block and writing it on chain;
- **Follower**： synchronize in light of leader node, select new leader node when the old leader becomes invalid;
- **Candidate**： contemporary identity of Follower nodes during election of leader.

1.3 Node ID & node index

In Raft algorithm, each nodes has a fixed and only ID (usually a 64-byte string), which is node ID; each consensus node maintains a public consensus node list, which records the ID of each consensus node. Node index refers to the location of the node in the list.

1.4 Terms

Raft algorithm divides time into Terms with uncertain length. Terms is sequential numbers. Each Term starts from election, if election succeeds, the current leader generates blocks; if fails and no leader is elected, a new Term and round of election will be started.



1.5 Message

In Raft algorithm, nodes communicate with each other by sending messages. Current Raft model contains 4 kinds of message: **VoteReq**, **VoteResp**、**Heartbeat**, **HeartbeatResp**:

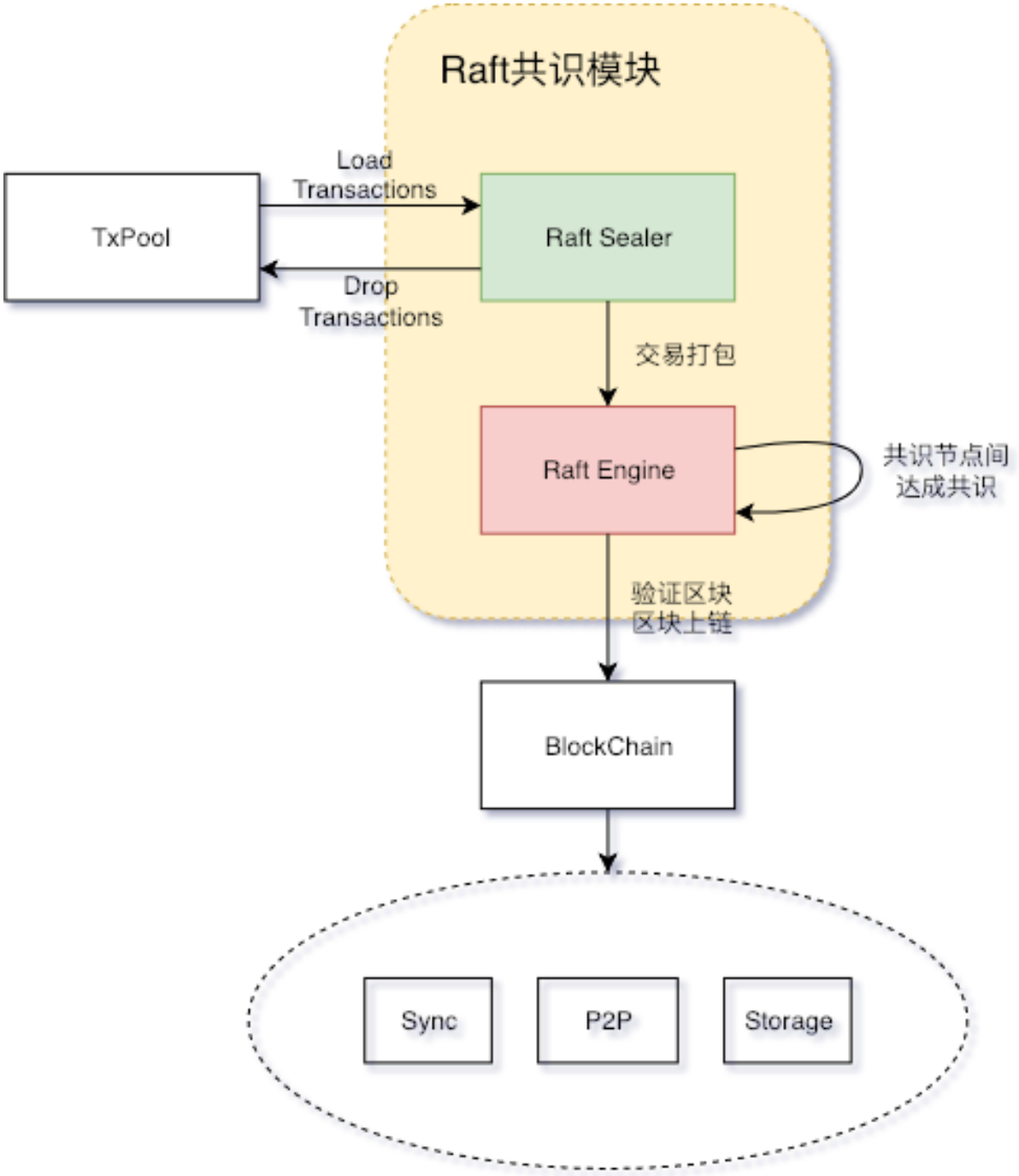
- **VoteReq**: vote request, sent by Candidate nodes to other nodes for leader election;
- **VoteResp**: vote response, used to respond the vote request by approve/disapprove the request;
- **Heartbeat**: heartbeat, sent by leader node periodically with 2 functions: (1) to maintain leader's identity, which will not change as long as leader can send heartbeat and receive response from other nodes; (2) to replicate block data, leader node will encode block data to its heartbeat and broadcast the block after sealing it, other nodes receives the heartbeat and decode the block data and place the block to their buffers;
- **HeartbeatResp**: heartbeat response after node receives heartbeat, when the heartbeat contains block data, the heartbeat response will contain hash of the block;

Fields commonly contained in messages are:

Fields exclusive in each message type include:

2 System framework

The system framework is as below:

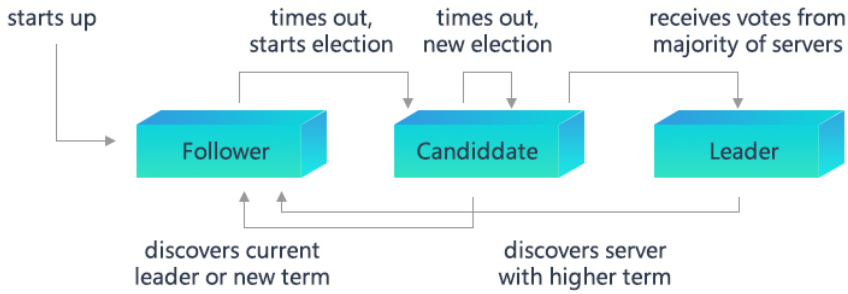


- Raft Sealer：take transaction out of txPool and seal block and send to Raft Engine for consensus; delete the on-chain transactions from txPool;
- Raft Engine：start consensus process within consensus nodes and write the consensused block to blockchain.

3 Core process

3.1 Node status transfer

The relations of node types are shown as below. And the transfer of each status will be introduced in the following context:



3.1.1 Election

Raft consensus model adapts heartbeat mechanism to start leader election. Node is set as follower of Term 0 since started. As long as follower can receive valid heartbeat or RequestVote message from leader or candidate, it will stay in follower status. If follower doesn't receive these messages in some time (which is called ***Election Timeout***), it will assume that the leader is invalid and increase its Term to become candidate, then a new round of leader election will be started:

1. Follower increases its Term, and becomes Candidate;
2. Candidate votes for itself and broadcasts RequestVote to other node for votes;
3. Candidate node keeps in Candidate status unless: (1) the node wins election; (2) Candidate receives heartbeat from other node during election; (3) no leader is elected after *Election Timeout*. Raft algorithm avoid even votes of nodes by random timer to ensure only one node will be time-out and enter candidate status and get most votes to become leader.

3.1.2 Vote

Node will respond with different strategies according to the content of received VoteReq:

1. ***Term of VoteReq less than or equal self Term***
   - If the node is Leader, disapprove the request. Candidate becomes Follower after receiving the response and increases timeline of voting;
   - If the node is not leader:
     - If Term of VoteReq is less than self Term, disapprove the request. If Candidate receives more than half of the same response, it is time-out and becomes follower and increase timeline of voting;
     - If Term of VoteReq equals self Term, disapprove the request and process no more. Each node can only vote to one Candidate on a first come basis, in order to ensure there will be only one Candidate will be elected to be Leader in each round.
2. ***lastLeaderTerm of VoteReq less than self lastLeaderTerm***

   Each node has a lastLeaderTerm field to indicate the Term of the last Leader it has witnessed. LastLeaderTerm can only be updated by Heartbeat. If the lastLeaderTerm in VoteReq is less than self lastLeaderTerm, then it has problem for leader to access the Candidate. If currently Candidate is in internet silos, it will keep sending vote request to outside, therefore nodes need to disapprove the request to stop it.
3. ***lastBlockNumber of VoteReq is less than self lastBlockNumber***

Each node contains a lastBlockNumber field to indicate the block number of the latest block witnessed by nodes. During block generation, nodes will replicate block (please check 3.2 section for details), during which there can be part of nodes who receive the new block data and part of nodes who don't. This will cause inconsistency in lastBlockNumber of each node, to solve which nodes need to vote for node who has the newest data. Therefore, node will disapprove the vote request under this circumstance.

4. ***node vote for the first time***

   To avoid Follower restarting election due to network dithering, it is stipulated that for the first time of node to vote, it should disapprove the request, and set its firstVote field to the index of the Candidate.

5. ***Not disapprove vote request in step 1~4***

   Approve the vote request.

3.1.3 Heartbeat time-out

When Leader becomes internet silos, though it can send heartbeat which can also be received by Follower, it cannot receive heartbeat response. In this situation Leader suffers internet problems, which can stuck the system for it can still send heartbeat to Follower who therefore cannot transfer status. To solve this problem, the model adopts heartbeat time-out mechanism. Leader will record each heartbeat response, if it stops update overtime, Leader will abandon its identity and transfer to Follower.

3.2 Block replication

Raft protocol strongly depends on the usability of Leader to ensure consistency of group data, because data can only be transferred from Leader to Follower. When Raft Sealer commits block data to group Leader, Leader will set the data to uncommitted status first, attach it with heartbeat for Follower to replicate and respond. When half of nodes are confirmed having received the data, it will write the data into data base, then the block data transfer to committed status. After that, Leader broadcasts the data to Follower through Sync model. The process of block replication is as below:

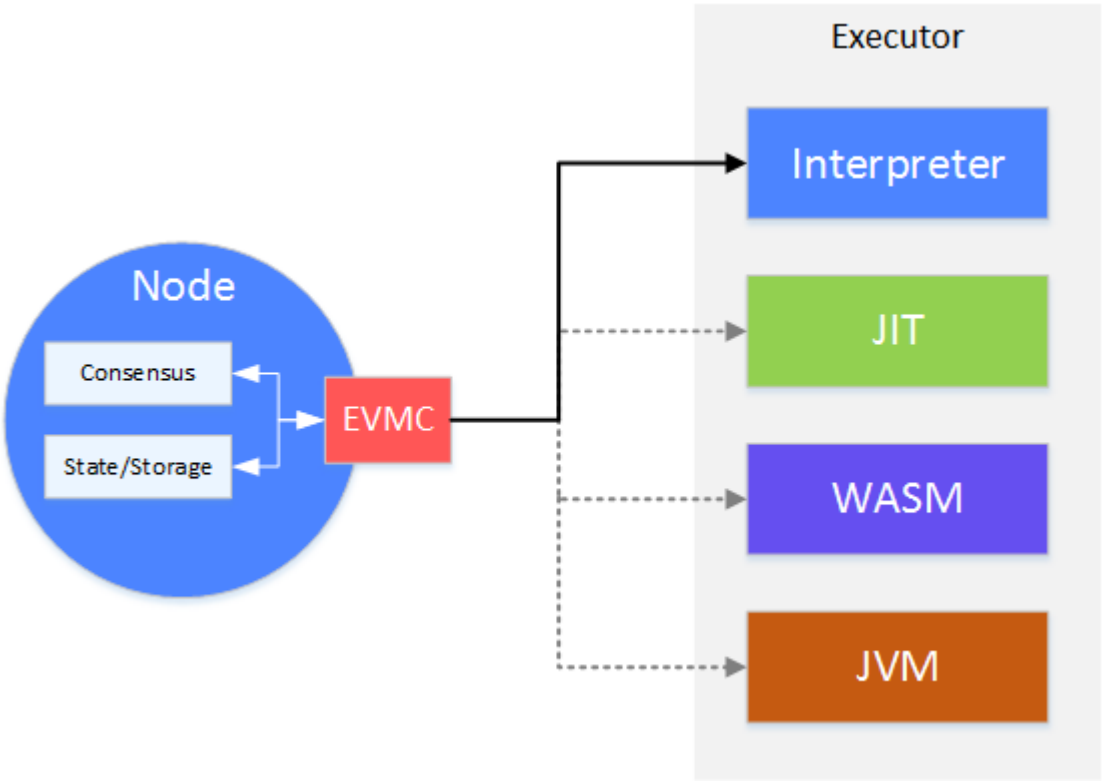
The conditions of RaftSealer to verify whether the transaction can be sealed include: (1) Leader node; (2) there is non-syncing peer; (3) uncommitBlock field is empty. Only when 3 of them are met can the transaction be sealed.

10.4 Virtual machine and contract

Execution of transaction is an important function of blockchain nodes. It is the process to extract binary code of smart contract from the transaction and execute by **Executo_**. Consensus model takes transaction out of TxPool, seals into block and calls executor to execute the transaction in block. When the transaction is being executed, it will change the State of blockchain and become new block status for storage. Executor works like a black box, inputting smart contract code and outputting state change.

With the technological development, people start to concern about the performance and usability of executor. On one hand, they want faster execution speed of smart contract on blockchain to meet large scale of transactions; on the other hand, they want to use familiar and easy development language. So, some solutions are proposed to replace traditional EVM: JIT, WASM and JVM. However, traditional EVM is coupled in node code. The first step is to abstract the API of executor to adapt for realization of all kinds of virtual machines. That is why EVMC is designed.

EVMC (Ethereum Client-VM Connector API) is the API of executor abstracted by Ethereum to adapt to all kinds of executors. FISCO BCOS currently adopts smart contract language Solidity of Ethereum, and also, the abstraction of executor API.



Consensus model will call EVMC on nodes and send sealed transaction to executor. The read/write of executor on status will be operated on the status data of nodes in return through call-back of EVMC.

Through the abstraction in EVMC level, FISCO BCOS can adapt to future executors with higher efficiency and usability. Currently, FISCO BCOS adopts executor abstracted through EVMC of traditional EVM—Interpreter. So, it supports smart contract based on Solidity. Other types of executors will be followed up in the future.

10.4.1 EVM – Ethereum Virtual Machine

The process of consensus is operated through contract deployment on blockchain. Ethereum Virtual Machine is a code executor of smart contract.

When smart contract is compiled to binary file and deployed on blockchain, users will start the execution of smart contract by calling its API. EVM executes smart contract and modifies blockchain data (status). The modified data will be in consensus to ensure consistency.

EVMC – Ethereum Client-VM Connector API

The latest version of Ethereum extracts EVM from node codes and makes it an independent model. The interaction between EVM and nodes has been abstracted to be the standard of EVMC API. Through EVMC, nodes can connect with different virtual machines, not limited to traditional virtual machine based on solidity.

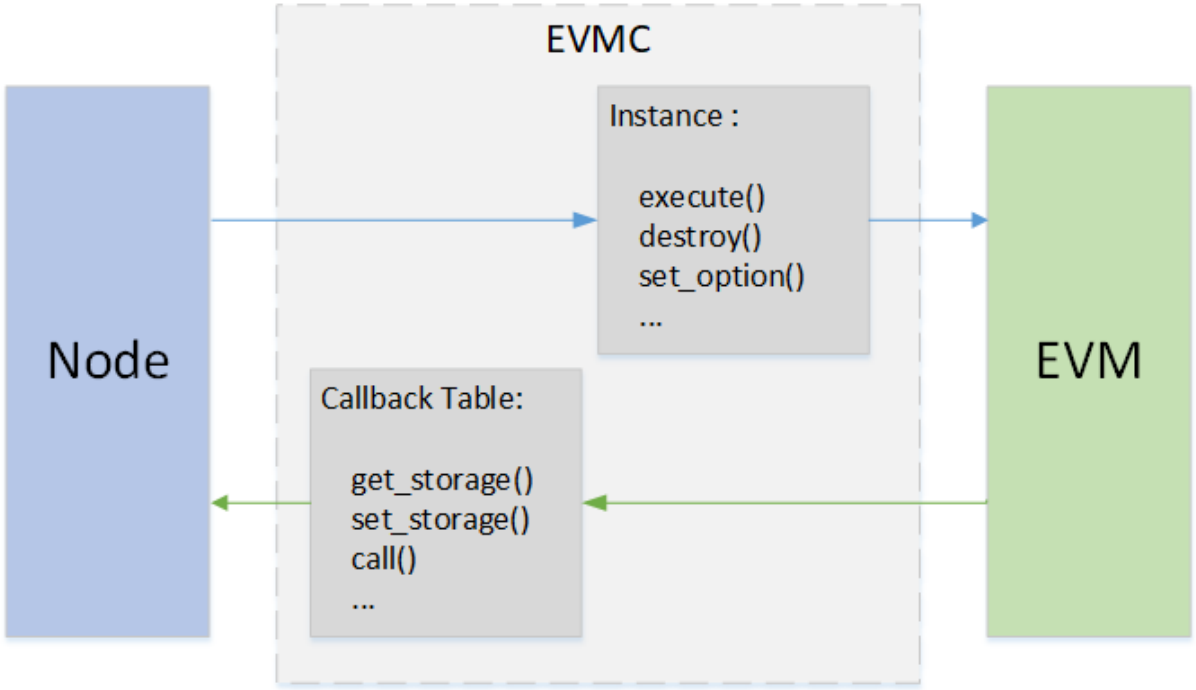
Traditional solidity virtual machine is called interpreter in Ethereum. We will introduce the implementation of interpreter in the following context.

EVMC API

EVMC has defined 2 types of API:

- Instance API: API for nodes to call EVM
- Callback API: API for EVM to call back node

EVM itself doesn't store status data. Node operates EVM through instance API, and EVM, in return, call Callback API to operate nodes' status.



Instance API

The operations of nodes ton virtual machine include create, destroy, set, etc..

API is defined in evmc_instance (evmc.h)

- abi_version
- name
- version
- destroy
- execute
- set_tracer
- set_option

Callback API

The operations of EVM on nodes mainly include read/write status and block information.

API defined in evmc_context_fn_table（evmc.h）.

- 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 execution

EVM instruction

Solidity is the execution language of contract. After compiled by solc, solidity will become EVM instruction like an assembly programming language. Interpreter has defined a complete collection of instructions. After compiled, solidity will generate binary file, which is the collection of EVM instructions. Transactions are sent in binary form to nodes, who will call EVM through EVMC to execute the instructions once receiving it. In EVM, the logic of the instructions are simulated in codes.

Solidity is based on stack. EVM calls by stack when executing binary file.

Arithmetic instruction

An ADD instruction is realized in following EVM codes. SP is the pointer of stack, which takes data from top 1 and 2 `SP[0]`, `SP[1]`, sums and writes to the top of SPP `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];
}
```

Jump instruction

JUMP instruction realize the jumping between binary codes. First, take out the address to jump to from stack top `SP[0]`, verify if it has crossed the limit, put into program counter (PC). The next instruction will start execution from where PC appoints.

```
CASE(JUMP)
{
    ON_OP();
    updateIOGas();
    m_PC = verifyJumpDest(m_SP[0]);
}
```

Read status instruction

SLOAD can inquire status data with these steps: take out the key to visit from stack top `SP[0]`, call callback function `get_storage()` with key as the parameter to check the value of key. Then, write the value to SPP top `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);
}
```

Write status instruction

SSTORE instruction can write data to node status with these steps: take out key and value from stack top 1 and 2 `SP[0]`、`SP[1]`, call callback function `set_storage()` with key and value as the parameters, write it to node status.

```
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();
    }
}
```

Call contract instruction

CALL instruction is used to call another contract according to the address. First, EVM judges if it's CALL instruction, calls `caseCall()` to take out data from stack using `caseCallSetup()`, encapsulates into msg, calls callback function of EVMC with msg as the parameter. When the `call()` is called back, Eth will start a new EVM to handle calls and return the execution result of the new EVM to current EVM through parameter of `call()`. The current EVM writes the result to SPP and ends call. It shares the same logic with the creation of contract.

```
CASE(CALL)
CASE(CALLCODE)
{
    ON_OP();
    if (m_OP == Instruction::DELEGATECALL && m_rev < EVMC_HOMESTEAD)
        throwBadInstruction();
    if (m_OP == Instruction::STATICCALL && m_rev < EVMC_BYZANTIUM)
        throwBadInstruction();
    if (m_OP == Instruction::CALL && m_message->flags & EVMC_STATIC && m_SP[2] !=␣
↪0)
        throwDisallowedStateChange();
    m_bounce = &VM::caseCall;
}
BREAK

void VM::caseCall()
{
    m_bounce = &VM::interpretCases;

    evmc_message msg = {};
```

```
    // Clear the return data buffer. This will not free the memory.
    m_returnData.clear();

    bytesRef output;
    if (caseCallSetup(msg, output))
    {
        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;
}
```

Summary

EVM is a machine for status execution, which input as binary instruction of compiled solidity and status data of nodes and output as the modification of node status. Ethereum's adaptability for multiple virtual machines is realized through EVMC. But till now, there has not been a production available virtual machine except interpreter. It is probably too difficult to run the same codes on different virtual machines but get the same result. BCOS will keep following up its development.

10.4.2 Precompiled contract

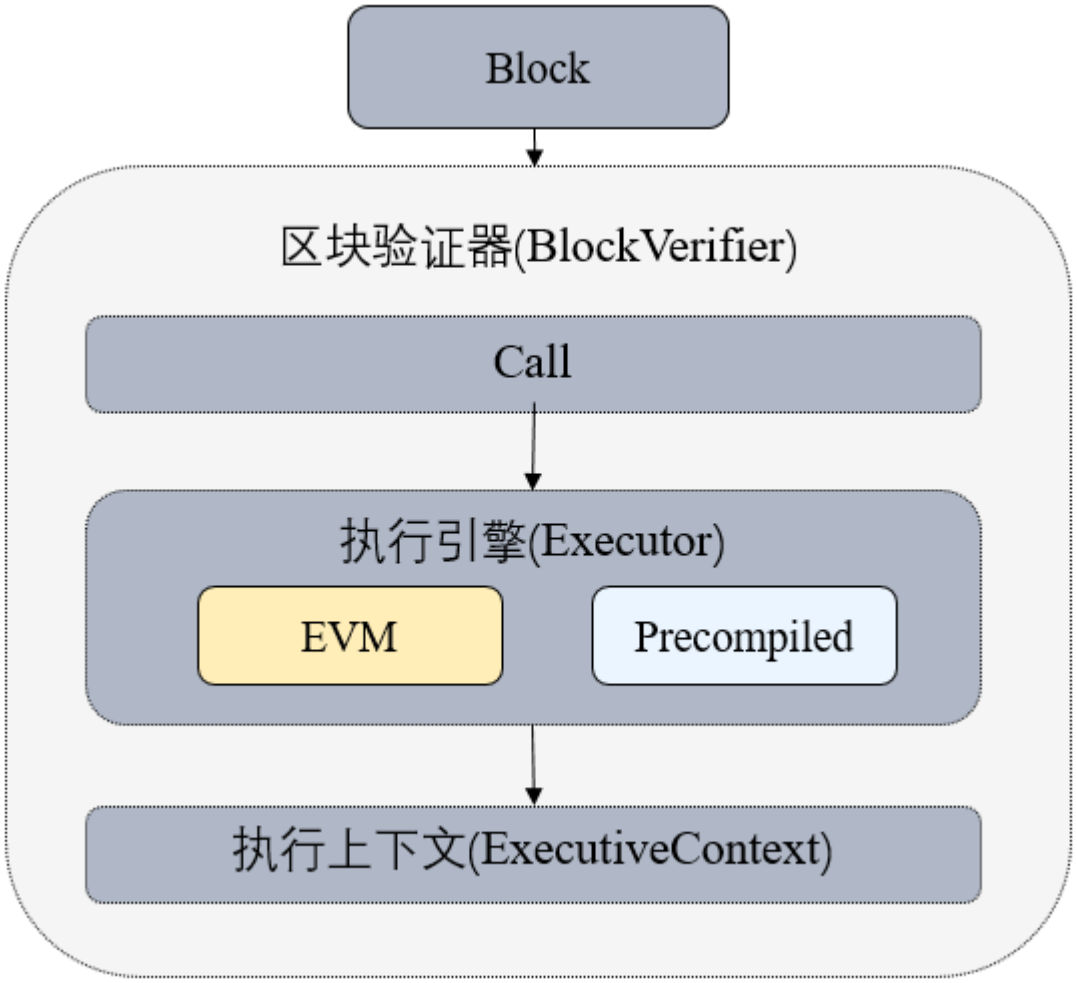
Precompiled contract offers a method to program contract using C++. It separates contract logic with data, and has better performance than solidity contract. The update can be achieved by modifying the codes in bottom layer.

Precompied contract and solidity contract

Model structure

The structure of Precompiled is as below:

- Block verifier will verify type of contract by the address of the called contract during transaction execution. Address 1-4 indicates Ethereum Precompile contract; address 0x1000-0x10000 belongs to C++ Precompiled contract, other addresses belong to EVM contract.



Crucial process

- The execution of Precompiled contract needs a object acquired through contract address.
- Each Precompiled contract object will implement `call` API, in which the detail logic of Precompile contract is implemented.
- `call` programs according to abi of transaction, acquires `Function Selector` and parameter, and executes the relative logic.

API definition

Every Precompiled contract has to realize its own `call` API, which accepts 3 parameters: `ExecutiveContext` executive context, abi code of `bytesConstRef` and exterior account address to judge whether it has write permission.`Precompiled` source code.

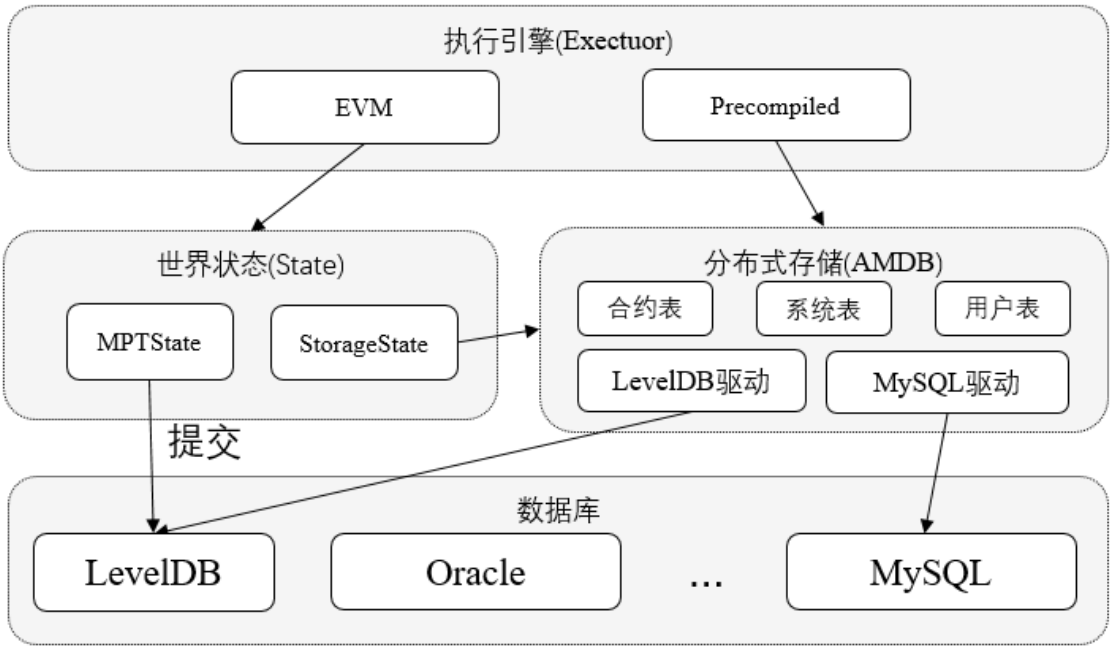
10.5 Storage model

FISCO BCOS inherits Ethereum storage while importing distributed storage with high extendability, TPS, adaptability and performance. The storge model contains 2 parts:

Global state: can further be divided into **MPTState** and **StorageState**

- **MPTState**: store account status by MPT tree, same with Ethereum
- **StorageState**: store account status by table structure of distributed storage, no historical records stored nor dependency on MPT tree, higher performance

Distributed storage (Advanced Mass Database, AMDB): realize consistency of SQL and NOSQL through abstract table structure, support all kinds of data bases by realizing the storage drive, currently support LevelDB and MySQL.



10.5.1 AMDB

Distributed storage (Advanced Mass Database, AMDB) can adapt to relational database or split to KV database through design on table structure. Theoretically AMDB supports any relational and KV databases by realizing the storage drive for different database.

- CRUD data, block data and contract code data are defaulted to be stored in AMDB without configuration. Local variables of contract can be configured as MPTState or StorageState if necessary. But contract code remains the same no matter what kind of state.
- When it's MPTState, contract local variables are stored in MPT tree. When it's StorageState, contract local variables are stored in AMDB table.
- Although data of MPTState AMDB will all be written to LevelDB finally, but they use different LevelDB instances with no transactional characteristics. Therefore, when it's configured to use MPTState, exceptions during committing data may lead to difference in data of the 2 LevelDB.

Terms definition

Table

Store data in storage table; store mapping of primary key of AMDB to Entries; support CRUD operations based on AMDB primary key; support filtering.

Entries

Store Entry, array same with primary key; different from the primary key in Mysql, AMDB primary key is used to signify which key the Entry belongs to; Entry of the same key will be stored in one Entries.

Entry

a row in a table; entry name as the key and entry value as the value to form KV structure; each entry owns a AMDB primary key; different entry can have the same AMDB primary key.

Condition

RUD APIs of Table support inputing of condition, the APIs will return filtered result according to the conditions; if the condition is empty, there won't be filtering.

Example

To illustrate the above terms in example, here is a procurement form of office supplies in a company.

Explanation

- In this table **Name** is the AMDB primary key.
- Each row is an Entry. There are 4 Entries which store data by Map:
  - 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}`
- In the Table **Name** is the primary key with 3 Entries objects. The first Entries stores 2 records of Alice; the second Entries stores one record of Bob; the third Entries stores one record of Chris.
- When calling retrieve API of Table, the API should set AMDB primary key and condition. Set AMDB primary key as Alice, condition as `price > 40`, it will retrieve Entry1.

Types of AMDB table

Each `entry` of table contains built-in fields of `_status_`, `_num_`, `_hash_`.

System table

The default system table is created in the promise of storage drive.

User table

Table created when user calls CRUD APIs, `_user_<TableName>` as table name, prefixed with `_user_` in bottom layer.

StorageState account table

`_contract_data_+Address+_` is table name. The table stores exterior accounts information. Table structure:

10.5.2 StorageState

StorageState is a method to store account status by AMDB. It has following difference with MPTState:

Every account in MPTState uses MPT tree to store data. As historical data increases, it will lead to low performance due to storage method and disk IO. Every account in StorageState is related to one table and stores its data, including `nonce`, `code`, `balance` of the account. AMDB can improve performance by supporting different databases through their storage drives. We have found in LevelDB test that StorageState is twice as much as MPTState in performance.

10.5.3 MPT State

MPT State is a classic data storage type in Ethereum. Through MPT tree, all contract data is organized to be inquired and retrieved.

MPT tree

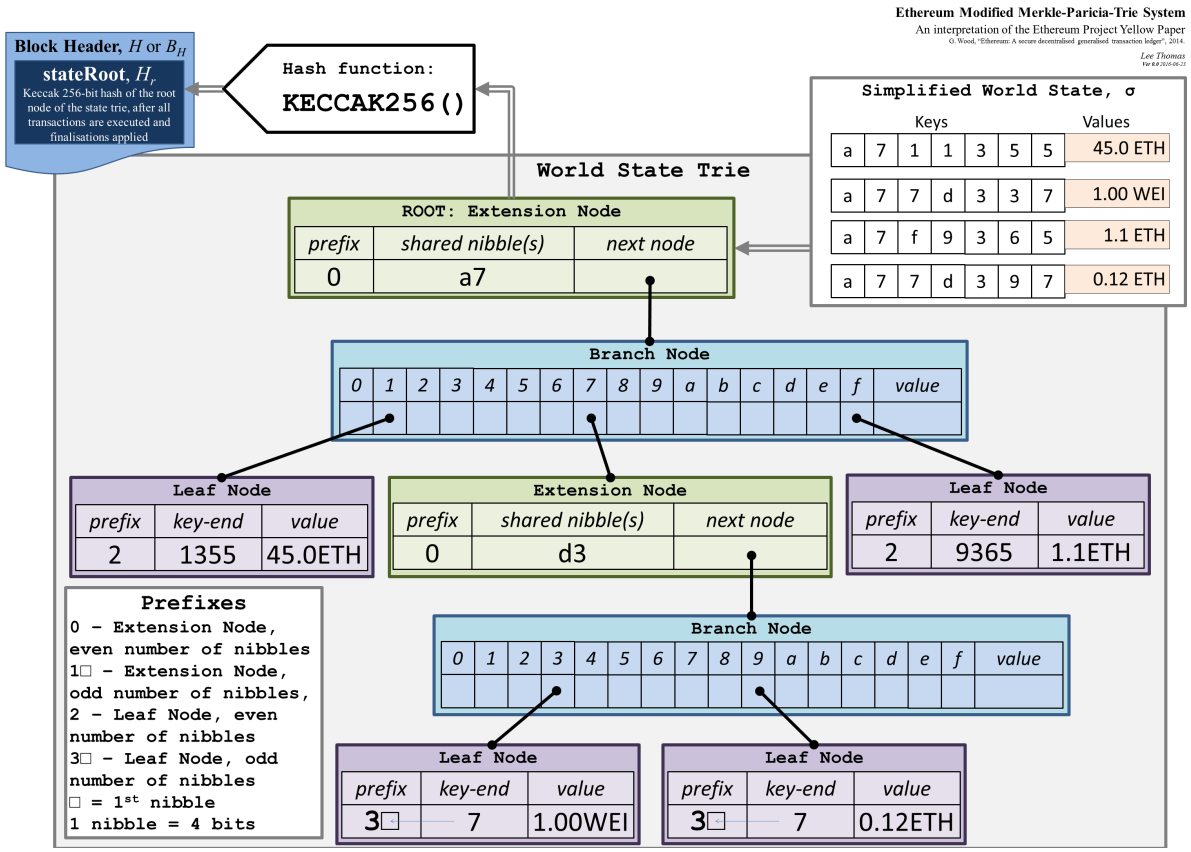
MPT(Merkle Paricia Trie) is a prefix tree to index data by hash.

In a broad sense, MPT is a prefix tree to retrieve value by key, which is using key to index in MPT tree, after passing middle nodes it will arrive at the leaf node that stores the data.

Narrowly speaking, MPT tree is a Merkle tree. Each node on tree is indexed by hash of the node. When using key to retrieve value, it will get the hash of next node first so as to retrieve the data of the next node from the bottom database, and then use key to get the hash of the next next node until it arrives at the leaf node stored with value.

When a leaf node of MPT tree updates its data, the hash of the node will be updated too. Consequently, the hash of all middle nodes backtracking to root node will be updated. Finally, hash of the root node will be updated. When searching new data, locate the root node by its new hash, and iterate back by layers to reach the new data. If searching historical data, locate the old root node by its old hash and iterate down to reach the historical data.

Implementation diagram of MPT tree (source of picture: the Ethereum Yellowpaper)



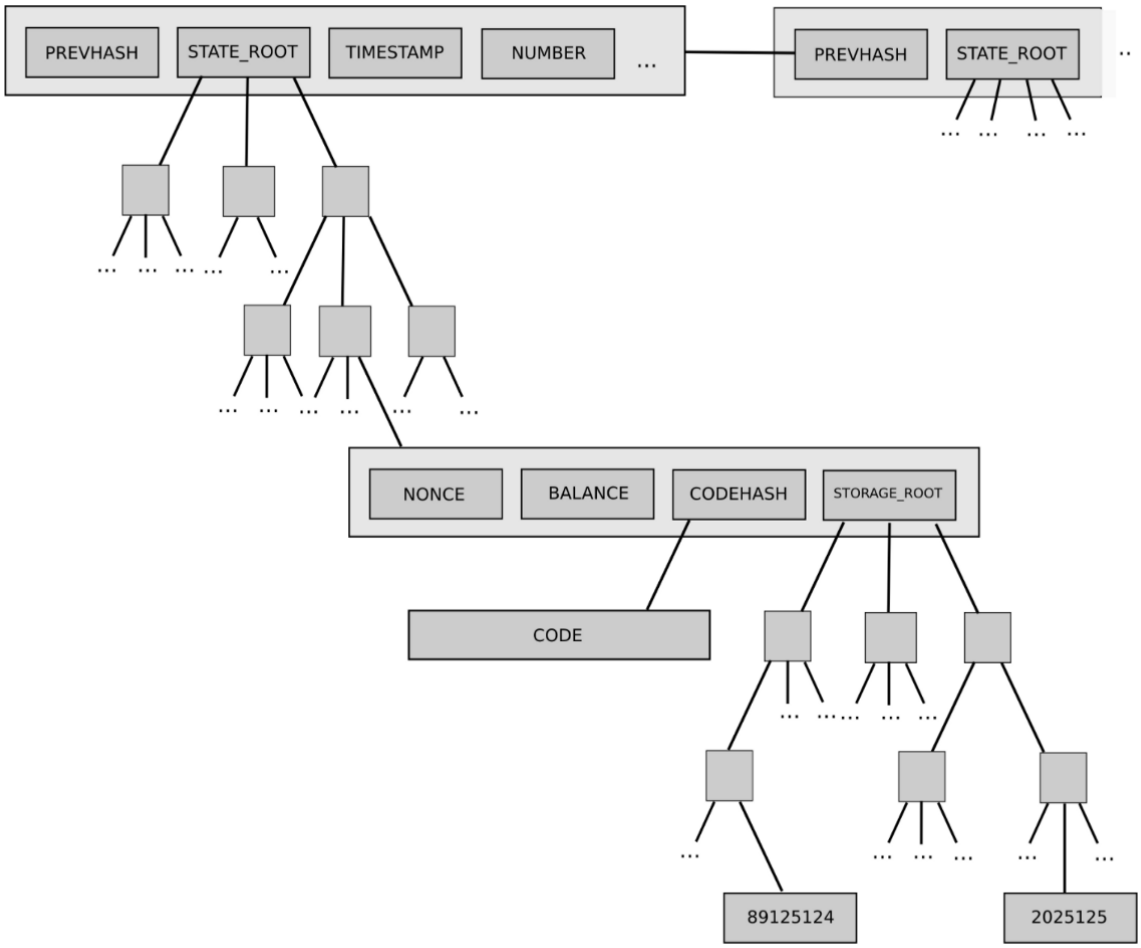
State

Data is stored in accounts in Ethereum. Each account stores contract (user) code, parameter, nonce, etc., which is retrieved by account address. Ethereum makes address as the key for query of account by MPT.

As the change of account data, the hash will change accordingly. Meanwhile, hash of the MPT root changes too. Different account data indicates different MPT root. For this aspect, Ethereum proposes a concept called State, in which the hash of MPT root is called State Root. Different state roots refers to different states, and will lead to different MPT root nodes. And then search account data of this state from different root nodes through account address. Different state may lead to different MPT root nodes and accounts.

State root is a field of block. Each block has different state. Operation of transactions in block will change data of account. Account data varies in each block, so as the block state, specifically, state root. The history data of account can be retrieved by taking out the state root of block to locate the MPT root node..

(source of picture: the Ethereum Yellowpaper)



Trade Off

MPT State is imported to retrieve data, The historical information of account can be retrieved according to the state root of block. However, it also brings out massive hash computings and breaks the consecutiveness of bottom data storage. MPT state is born with disadvantages in performance. We can say that MPT State has extremely good traceability at the cost of performance.

In transactional cases of FISCO BCOS, performance matters more than traceability. Therefore, FISCO BCOS has re-designed the bottom storage and implemented Storage State. Storage State has better performance regardless of losing part of traceability.

10.6 Security control

To ensure safe communication and access of data among nodes, FISCO BCOS adopts mechanisms of node access, CA blacklist and permission control for security control in network and storage level.

Network security control

- **SSL connection** of nodes, ensuring secrecy of communication
- **Network access mechanism**, ensuring system security by removing malicious nodes from consensus node list or group
- **Group whitelist mechanism**, ensuring independency of communication data among groups by making each group receives the messages of the counter group only
- **CA blacklist mechanism**, disconnecting malicious nodes in time
- **Distributed storage permission control mechanism** controlling the permissions of exterior accounts for contract deployment and CRUD operations on user table

Storage security control

The permission control mechanism based on distributed storage controls access in a flexible and delicate way by implementing the restriction on the storage access for exterior accounts (tx.origin), which includes contract deployment, table creation and writing.

10.6.1 Node access management

This chapter will give a brief introduction on node access management. For implementation methods please read Node access management。

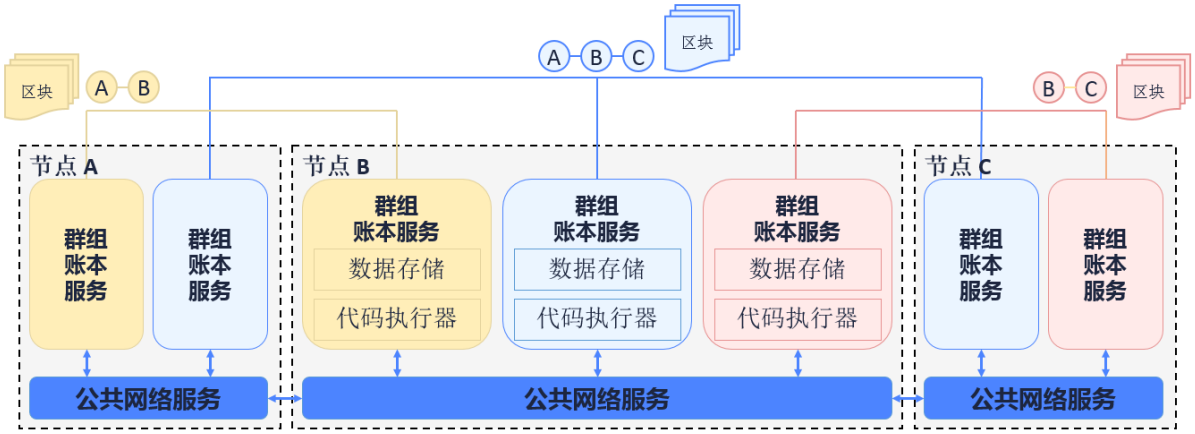
Introduction

Single-chain multi-ledger

Blockchain is a technology with decentralized and transparent distributed data storage which has lower cost in trust and highly reliability of data interaction. But meanwhile, the transaction data of blockchain faces threat of leakage:

- for public chain, any nodes can join some network and acquire all data from global ledger;
- for consortium chain, through there is network access mechanism, nodes can still acquire data from global ledger once joined blockchain.

FISCO BCOS, as a consortium chain, has proposed Single-chain multi-ledger solution to solve the privacy issue. It has also adopted **Group** concept, transferring the traditional storage/execution mechanism from one-chain one-ledger to one chain and multi-ledger and realizing independency and secrecy of data on a chain based on groups.



As the above diagram shows, node A, B, C join the blue group and co-maintain the blue ledger; node B, C join the pink group and co-maintain the pink ledger; node A, B join the yellow group and co-maintain the yellow ledger. The three groups share the public internet service but have independent ledger storage and transaction executive environment each. Client end sends transaction to a group it belongs, in which transaction and data will be consensus and stored by members, while other group can not sense or access to the transaction.

Node access mechanism

Based on groups, node access management can be divided into **Network access mechanism** and **Group access mechanism**. Rules of the mechanism are recorded in the configuration, which will be read by node once it is started to implement access control of network and group.

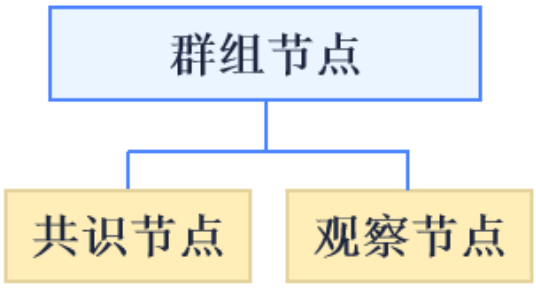
Terms definition

Node type

Nodes mentioned in this document are all permitted with network access for P2P communication. **The process of network access involves adding P2P node connection list and certificate verification.**

- **Group node**: nodes with network permission that joined a group. Group node is either consensus node or observer node. Consensus node takes part in consensus block generation and transaction/block syncing; observer node only involves block syncing. **The process of node access involves the sending of transactions to dynamically create/remove node.**
- **Free node**: nodes with network permission but not join any group. **Free node doesn't get group permission nor join consensus and syncing process.**

The relations of node:

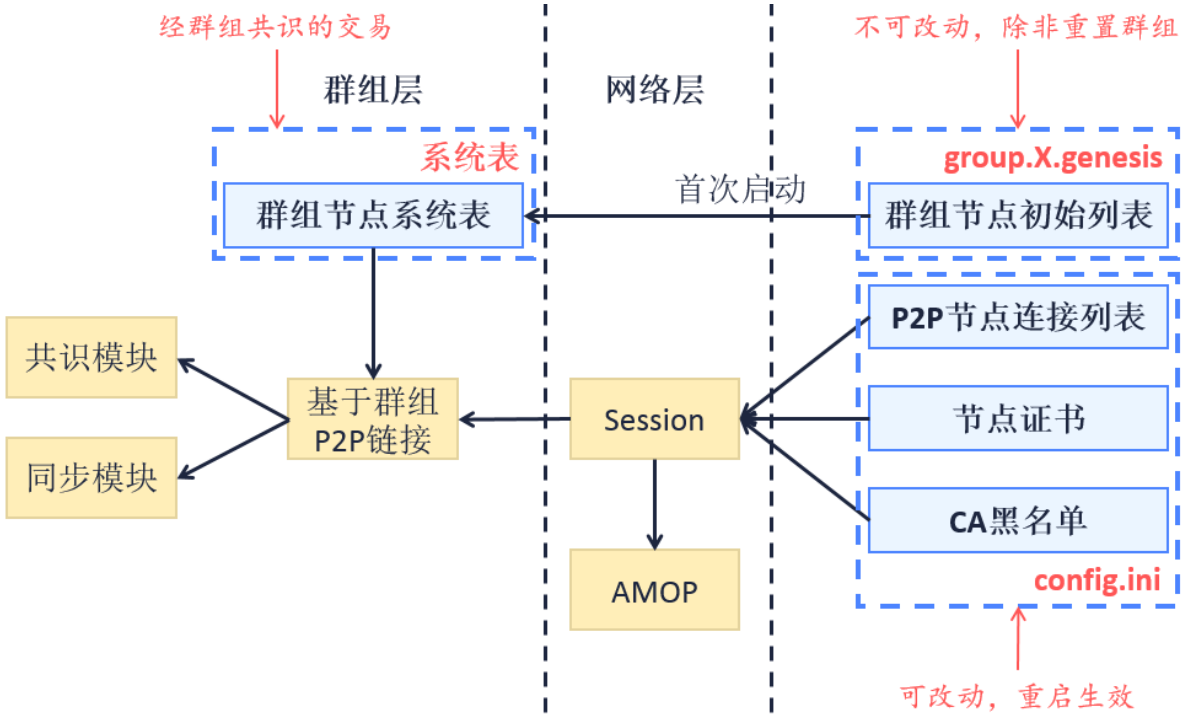


Config type

Node access config items

The config items involving in node access management are: **P2P node connection list**, **node certificate**, **CA blacklist**, **group node initial list** and **group node system list**.

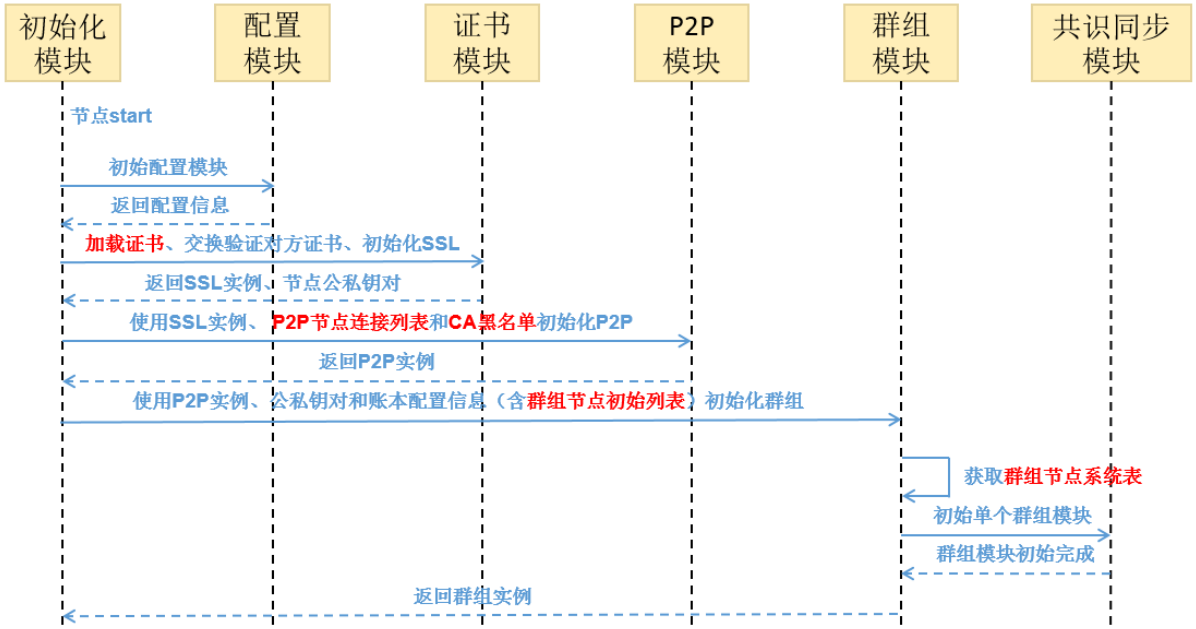
Model structure



In **the diagram of relation between config items and system model** above, A->B means that B model depends on data of A model, and B model's initialized later than model A.

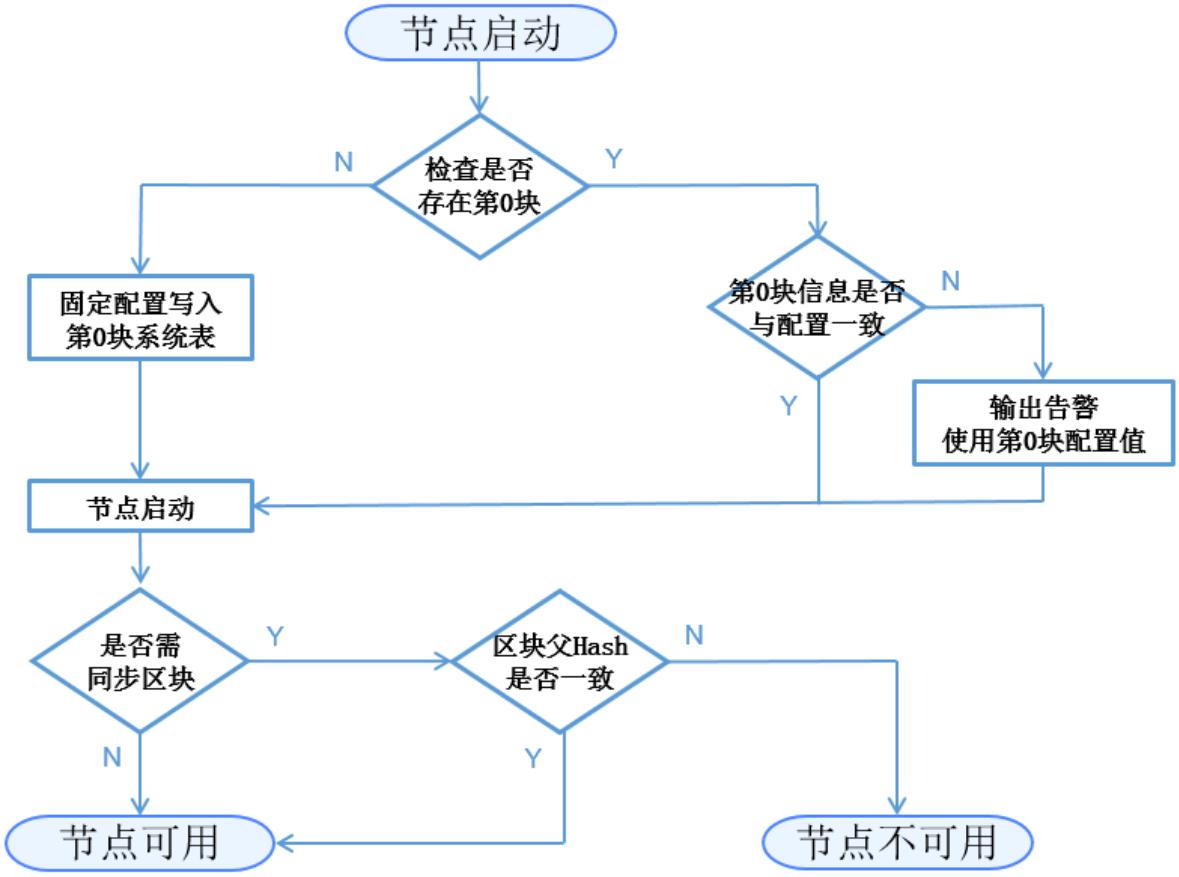
Core process

Regular initialize process



The first initialization

When a node is started at the first time, groups it belongs to write the fixed config to block 0 and commit to blockchain. The detail logic of initialization is:



Config contents concerning node access management and needing to be written in this phase include:**group node initial list->group node system list**.

Description:

- The block 0 of each node in one ledger should be the same, that is **the fixed config file** should be the same;
- The following starts of nodes should contain checking if block 0 information is the same with the fixed config file. If the config file has been modified, node will send warning message in its next start but won't interfere with other groups' operation.

Node networking based on CA blacklist

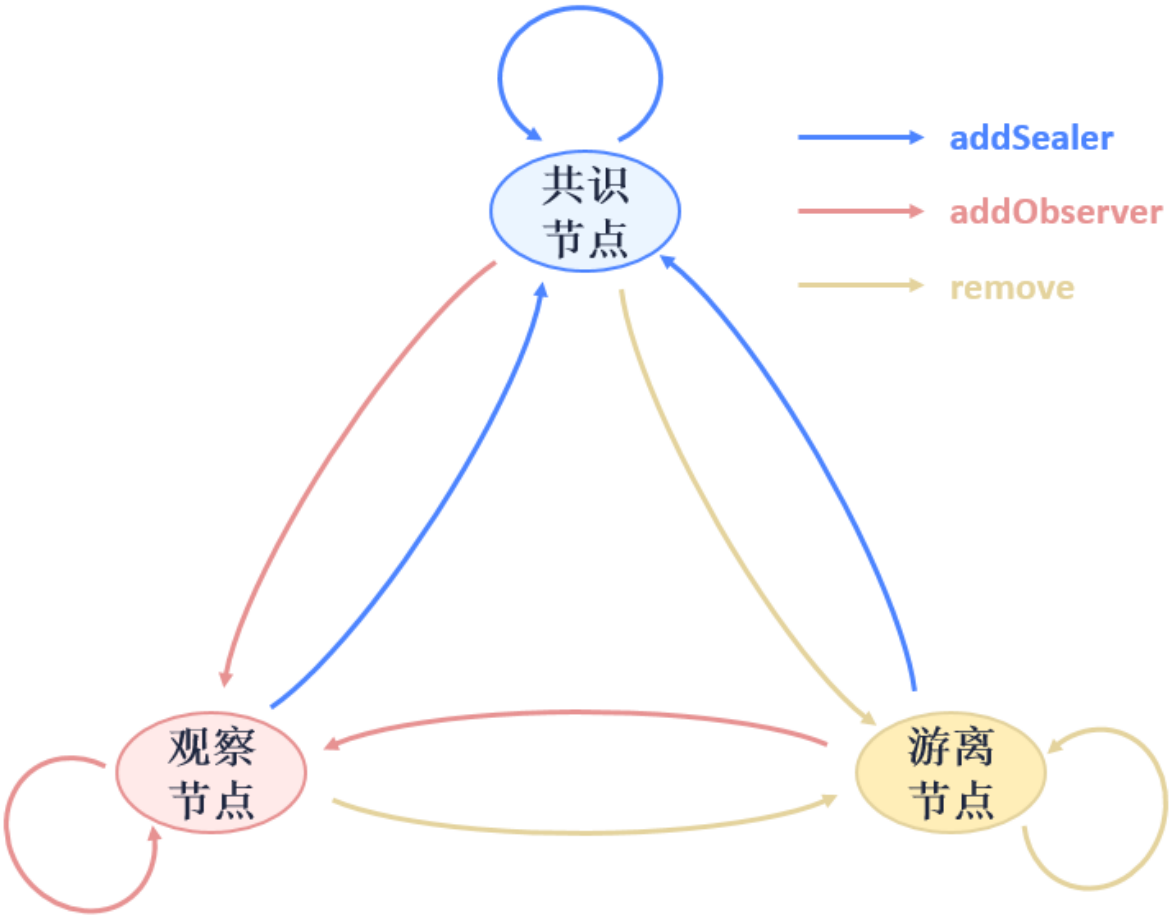
SSL certificate to confirm nodes access to blockchain. Nodes on one chain all trust a trustable third party (issuer of node certificate).

FISCO BCOS requires implementation of **SSL two-way certificate**. During handshake, nodes acquire each other's node ID from the certificates and verify if it is in CA blacklist. If is, close the connect; if not, create session.

CA blacklist mechanism also supports **SSL one-way certificate**, which is used in this case: when node has created session, it can acquire the counter node's ID from session to verify if it's in the CA blacklist. If is, shut down the session.

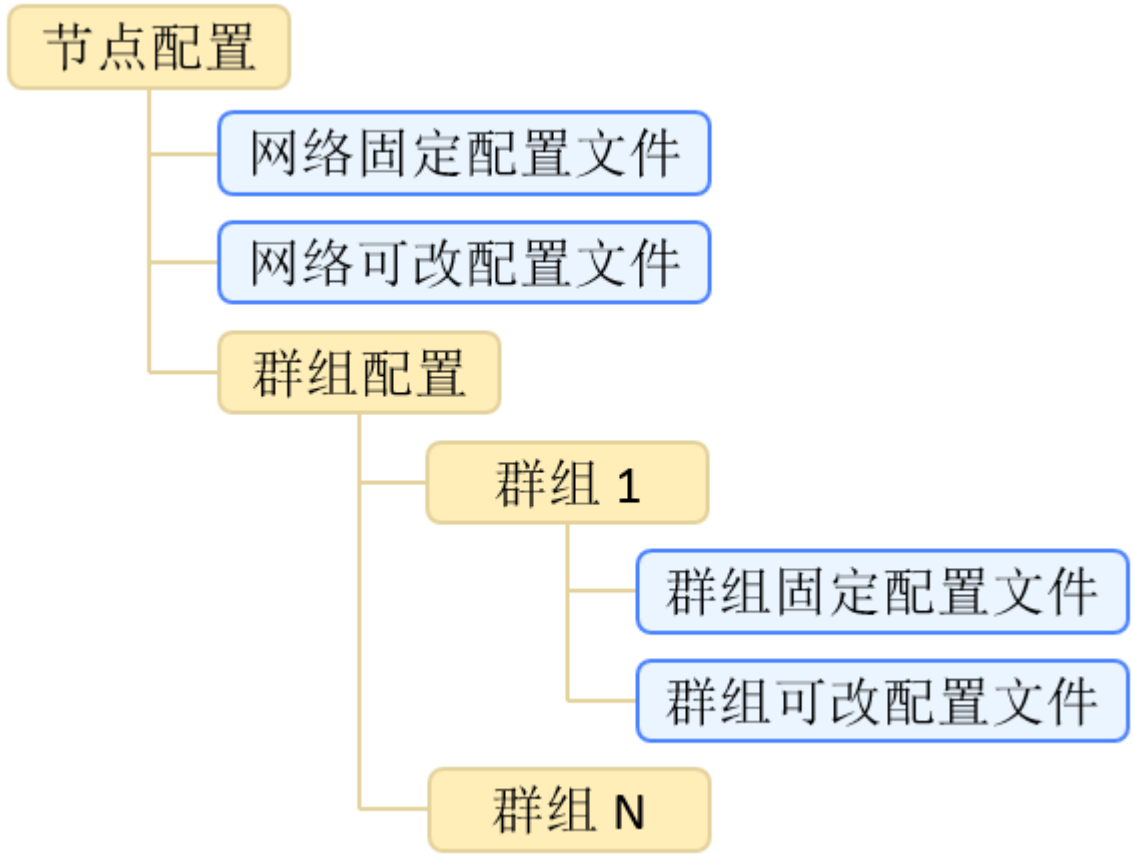
Node types and transfer

The three types of nodes (consensus node/observer node/free node) can transfer through APIs like this:



API and Config

Hierarchy of node config file



The organization rules of config file are: **independent config among groups**;**fixed config and modifiable config are independent**. The files involved include: **network modifiable config file** `config.ini`, **group fixed config file** `group.N.genesis` and **group modifiable config file** `group.N.ini`, among which `N` is the group number of node. For **network/group modifiable config file**, if the value of config item is not shown in the file, the program will use the default value.

Config file example

For **network modifiable config file **`config.ini`, node access management involves **P2P node connection list `[p2p]`**, **node certificate `[secure]`**, **CA blacklist `[certificate_blacklist]`**. `[certificate_blacklist]` is optional. Example of config items:

```
[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

;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
```

For **group fixed config file**`group.N.genesis`, node access management involves **group node initial list [consensus]**. Example of config items:

```
;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
```

Group node system list definition

Group system API definition

Group system list implements whitelist mechanism in group level (comparing to the implementation of CA blacklist). APIs offered by group system list include:

```
contract ConsensusSystemTable
{
    // modify one node to be sealer
    function addSealer(string nodeID) public returns(int256);
    // modify one node to be observer
    function addObserver(string nodeID) public returns(int256);
    // remove the node from group system list
    function remove(string nodeID) public returns(int256);
}
```

Expectation on functions

- **Modifiable config** is valid by restart after modification. In the future, dynamic loading and real-time validation can be realized;
- **CA blacklist** has realized the node-based blacklist. In the future, we will consider about blacklist based on agency.

10.6.2 CA blacklist

This section contains the brief introduction of blacklist, for the implementation please check CA blacklist operation tutorial.

Terms definition

Terms definition

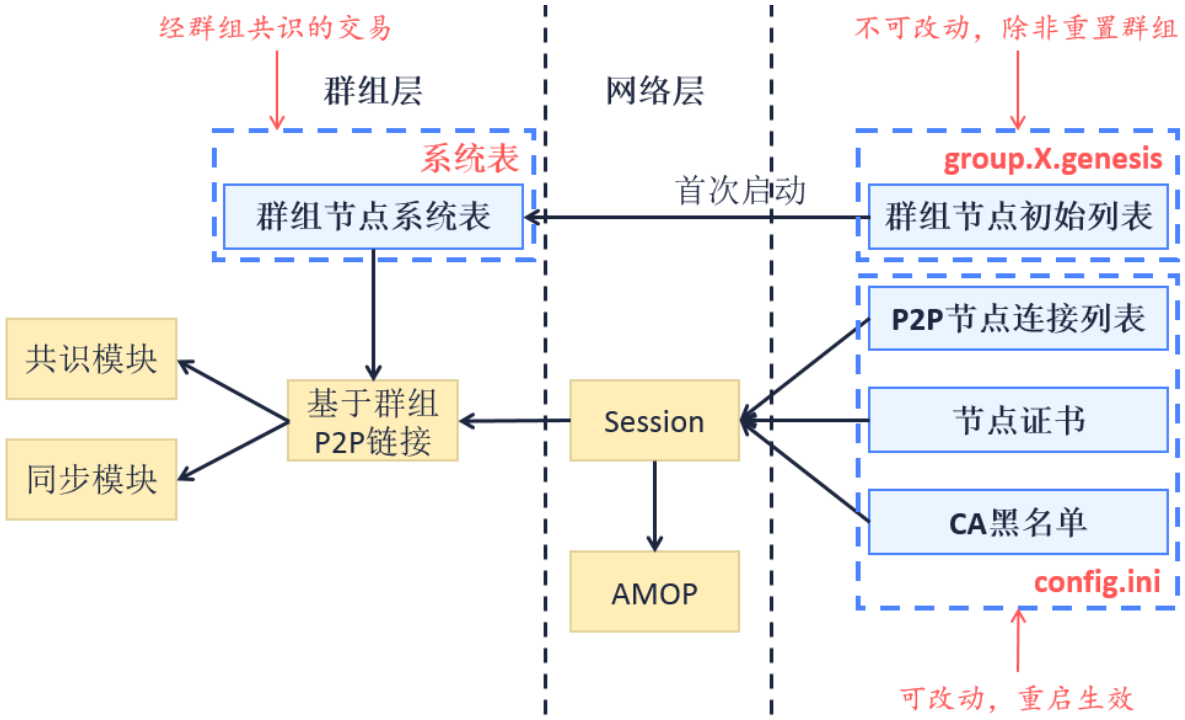
CA blacklist is also known as **Certificate Blacklist**, or CBL. CA blacklist verifies and denies connection requests from each node based on node ID in `[certificate_blacklist]` of `config.ini` configuration file.

Config types of CA blacklist:

- according to **effective area** (network config/ledger config), it belongs to **network config** and affects the node connection in the overall network;
- according to **modifiability** (modifiable config/fixed config), it belongs to **modifiable config**, modification valid after restart;
- according to **storage location** (local storage/store on chain), it belongs to **local storage**, content recorded in local instead of on chain.

Model structure

The below diagram shows the model and relations of CA blacklist. A->B informs that model B depends on the data of model A, and it is later in initialization than model A.



Core process

SSL two-way certificate is implemented in the bottom level of FISCO BCOS. During handshake, nodes acquire each other's ID from its certificate and verify if it is among the CA blacklist. If is, close the connect and resume the later process.

Effective area

- CA blacklist has evident influence in P2P node connection in network level and AMOP function by **invalidation**;
- CA blacklist has potential influence in consensus and syncing in ledger level by **interfering message/data transfer**.

Config format

`config.ini` node config adds `[certificate_blacklist]` route (`[certificate_blacklist]` is optional). CA blacklist contains node ID list. node.X indicates node ID of rejected nodes. Config format of CA blacklist is as below:

```
[certificate_blacklist]
    crl.
↪0=4d9752efbb1de1253d1d463a934d34230398e787b3112805728525ed5b9d2ba29e4ad92c6fcde5156ede8baa5aca3
    crl.
↪1=af57c506be9ae60df8a4a16823fa948a68550a9b6a5624df44afcd3f75ce3afc6bb1416bcb7018e1a22c5ecbd016a
```

Expected functions

- Modification of CA blacklist becomes valid after restarting node. In the future, dynamic loading and real-time validation are expected to be realized;
- CA blacklist has realized black list based on node. In the future, we will consider blacklist based on agency.

10.6.3 Permission control

Introduction

Comparing with public chain with open access and free transaction or query, consortium chain are required to have access control, diverse transactions, privacy protection in business level and high stability. Therefore, "permission" and "control" are strongly emphasized in the implementation of consortium chain.

In this case, FISCO BCOS has proposed a flexible and delicate permission control mechanism based on distributed storage, which performs fundamental technological supports for its governance. The permission control system manages permissions on contract deployment and table create/insert/update/delete operations (read is not included), based on the access mechanism of exterior accounts (tx.origin). In actual operations, each account owns one and only public and private key pair. Private key is for signature when sending transactions. The receiver will verify to know which account the transaction is from through the public key, helping the management and tracing back of transaction and regulation.

Rules

The rules of permission control include:

1. the minimum granularity of permission control is table, based on exterior accounts.
2. whitelist mechanism is adopted: all exterior accounts are default to have write/read permission.
3. permission table (_sys_table_access_) is used in permission control, if table name and account address are in the permission table, the account has write/read permission to the table; if not, account has only read permission.

Types

Permission control based on distributed storage contains permissions of user table and sys table. User table is created by user contract. All user tables can set permissions. System table is built in FISCO BCOS network. The design of system table is introduced in Storage Documentation. Permission control of system table is as below:

Targeting user table and each system table, SDK conducts permission management of three APIs:

- User table:
  - **public String grantUserTableManager(String tableName, String address)**: set permission information according to user table name and exterior account address.
  - **public String revokeUserTableManager(String tableName, String address)**: remove permission information according to user table name and exterior account address.
  - **public List<PermissionInfo> listUserTableManager(String tableName)**: inquire permission information according to the user table name (each record contains exterior account address and valid block number).
- _sys_tables_表:
  - **public String grantDeployAndCreateManager(String address)**: grant permission of contract deployment and user table creation to exterior account.
  - **public String revokeDeployAndCreateManager(String address)**: remove permission of contract deployment and user table creation of exterior account.
  - **public List<PermissionInfo> listDeployAndCreateManager()**: inquire permission records of contract deployment and user table creation.
- _sys_table_access_表:
  - **public String grantPermissionManager(String address)**: grant permission of management to exterior account.
  - **public String revokePermissionManager(String address)**: remove permission of management of exterior account.
  - **public List<PermissionInfo> listPermissionManager()**: inquire permission records of permission management.
- _sys_consensus_表:
  - **public String grantNodeManager(String address)**: grant node management permission to exterior account.
  - **public String revokeNodeManager(String address)**: remove node management permission of exterior account.
  - **public List<PermissionInfo> listNodeManager()**: inquire permission records of node management.
- _sys_cns_表:
  - **public String grantCNSManager(String address)**: grant CNS permission of exterior account.
  - **public String revokeCNSManager(String address)**: remove CNS permission of exterior account.
  - **public List<PermissionInfo> listCNSManager()**: inquire CNS permission records.
- _sys_config_表:
  - **public String grantSysConfigManager(String address)**: grant system parameter management permission to exterior account.
  - **public String revokeSysConfigManager(String address)**: remove system parameter management permission of exterior account.
  - **public List<PermissionInfo> listSysConfigManager()**: inquire system parameter management permission records.

Set and remove permission API and return json string, containing code and msg fields. For operations with no permission, code is set to 50000 and msg defined as "permission denied"; for those granted with permission, code is 0 and msg "success".

Data definition

Permission information is stored in the form of system table, which is named *sys_table_access* with following fields:

The insertion or update to the permission table is validated in the next block instead of current block. When the state field is 0, it means that the permission record is in normal valid status; if 1, the permission record is removed and invalid.

Design

Function design of permission control

Transaction information contains exterior account, pending table and operation methods. Pending table is user table or system table. As what shows below, system table manages system functions of blockchain; user table manages transactional function of blockchain. Exterior accounts can get permission information through permission table, and then manage system and transactional functions by operating user table and permission table.

Process design

The process of permission control: first, client end sends transaction request, node acquires transaction data to confirm exterior account, pending table and operation type; if it's write operation, check the permission (from permission list), if it has permission, execute write operation; if not, reject write operation and return no permission; if it's read operation, skip permission information check, execute read operation and return data. The process is shown below.

Tool

The operation methods of permission control in FISCO BCOS include:

- non-developers can user permission function through console command, the detailed operation is introduced in Operation tutorial for permission control.
- For developers, SDK has implemented 3 APIs (grant/remove/inquire) in light of user table and system table, so they can call PermissionService of SDK API to realize permission management.

10.7 P2P network

10.7.1 Design objective

FISCO BCOS P2P model, with basic functions for efficient, commonly-used and safe internet communication, supports unicast, multicast and broadcast of blockchain messages, synchronizes nodes status and adapts multiple protocols.

10.7.2 P2P main functions

- Blockchain node identification

The nodes on blockchain are uniquely identified by node ID, which is used for addressing nodes.

- Network connection management

It maintains TCP persistent connection between nodes, automatically disconnect and reconnect when connection exceptions occur.

- Messaging

Messages can be unicasted, multi-casted and broadcasted among nodes on blockchain.

- Status syncing

Node status can be synchronized on blockchain.

10.7.3 Blockchain node ID

Node ID is generated by the public key of ECC algorithm. Each node owns one and unique ECC key pair, as each node is identified by one and only node ID.

Usually, there files are needed when adding a node to blockchain network:

- node.key key of node in ECC format
- node.crt node certificate issued by CA
- ca.crt CA certificate provided by CA

Besides a unique node ID, nodes can follow Topic for addressing.

Node addressing:

- Addressing by node ID

To locate a node by its node ID.

- Addressing by Topic

To locate a group of nodes that follows some Topic.

10.7.4 Network Connection Management

Blockchain nodes will automatically start and maintain TCP permanent connection, or reconnect when there is exception in system and network.

CA certificate will be used to verify nodes during connection.

Connection process

10.7.5 Messaging

Messaging among nodes supports unicast, multicast and broadcast.

- Unicast, one node sends messages to another node, addressing through node ID.
- Multicast, one node sends messages to a group of nodes, addressing through Topic.
- Broadcast, one node sends messages to all nodes.

Unicast process

Multicast process

Broadcast process

10.7.6 Status syncing

Every node maintains its own status and broadcasts the Seq regularly to synchronize all nodes.

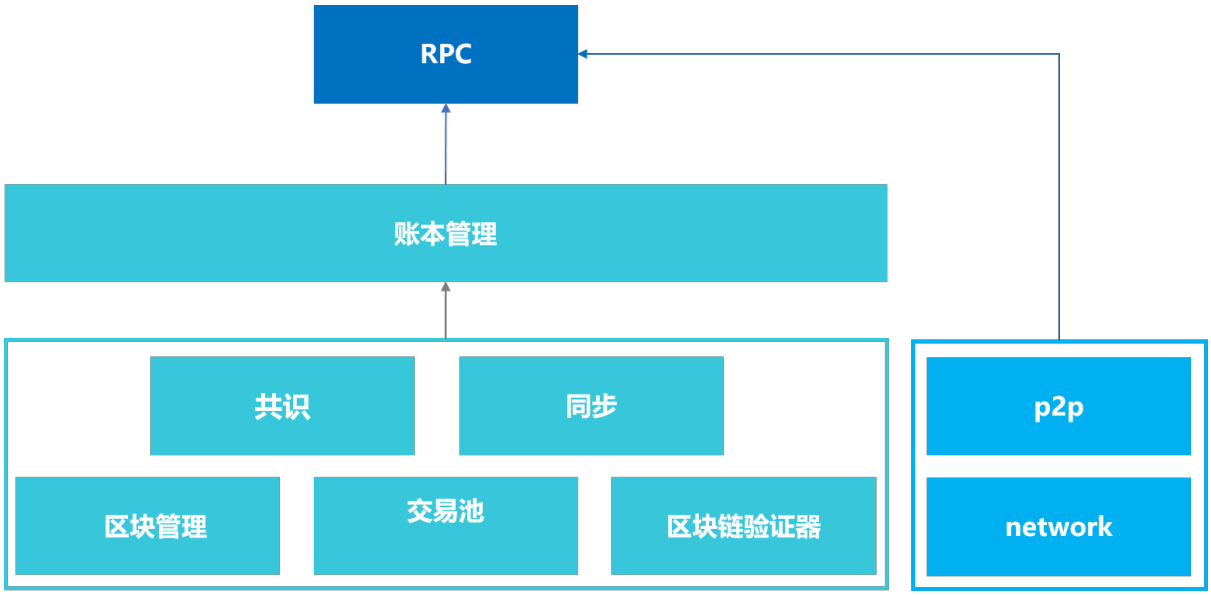
10.8 RPC

RPC(Remote Procedure Call) is a set of protocols and interfaces that the client interacts with blockchain system. The user can query the blockchain related information (such as block number, blocks, node connection, etc.) and send the transaction request through RPC interface.

10.8.1 1 Glossary

- JSON(JavaScript Object Notation): A lightweight data exchange format. It can represent numbers, strings, ordered sequences, and key-value pairs.
- JSON-RPC: A stateless, lightweight remote procedure call protocol. The specification primarily defines several data structures and their processing rules. It is allowed to run in the same process based on much different messaging environments such as socket, http, etc. It uses JSON ([RFC 4627](http://www.ietf.org/rfc/rfc4627.txt)) as the data format. FISCO BCOS adopts the JSON-RPC 2.0 protocol.

10.8.2 2 Module architecture



The RPC module is responsible for providing the external interface of FISCO BCOS. The client sends the request through RPC, and RPC obtains the relevant response by calling [book management module] (architecture/group.md) and [p2p module] (p2p/p2p.md), and returns the response to the client. The ledger management module manages the relevant modules at the underlying of blockchain through a multi-book mechanism, including [consensus module] (consensus/index.html), [synchronization module] (sync/sync.md), block management module, transaction pool module, and block verification module.

10.8.3 3 Data definition

3.1 Client request

The client request sent to blockchain node will trigger an RPC calling. The client request includes the following data members:

- jsonrpc: A string specifying the JSON-RPC protocol version, which must be accurately written as "2.0".
- method: The name of the method to call.
- params: The parameters required to call the method. The method parameters are optional. Since FISCO BCOS 2.0 enables a multiple ledger mechanism, this specification requires that the first parameter passed in must be the group ID.
- id: The established unique ID of client. The ID must be a string, a numeric value or a NULL value. If the ID does not include one of these, it is considered as a notification.

The example format of RPC request package:

```
{"jsonrpc": "2.0", "method": "getBlockNumber", "params": [1], "id": 1}
```

Note:

- NULL is not recommended as the id value in the request object because the specification will use a null value to identify the request as an unknown id.
- Decimal is not recommended as the id value in request objects because of uncertainty.

3.2 Server response

When starting an RPC calling, all blockchain nodes must respond others except the notification. The response represents as a JSON object, using the following members:

- jsonrpc: A string specifying the JSON-RPC protocol version which must be accurately written as "2.0".
- result: The correct result field. This member must be included when the response is processed successfully, and must not be included when the calling method causes an error.
- error: Error result field. The member must be included in the failure, and must not be included when no error is caused. Its parameter value must be an object defined in the [3.3] (#id6) section.
- id: Responsive id. The member must be contained. Its value must match the id value in the corresponding client request. If the id of the request object shows errors (such as a parameter error or an invalid request), the value must be null.

The example format of RPC response package:

```
{"jsonrpc": "2.0", "result": "0x1", "id": 1}
```

Note: The server response must contain a member result or error, but not both of them.

3.3 Error object

When an RPC calling encounters error, the returned response object must contain an error result field. For relative description and error codes, please check RPC error codes

10.8.4 4 RPC interface design

FISCO BCOS provides the rich RPC interfaces for client calling. They are divided into 3 categories:

- The query interface named beginning with get: For example, `[getBlockNumber]` interface, is to query the latest block number.

- `[sendRawTransaction]` interface: to execute a signed transaction and return response after blockchain achieves consensus.
- `[call]` interface: Executing a request will not create a transaction, so no blockchain consensus is required, but a response is returned immediately.

10.8.5 5 RPC interface list

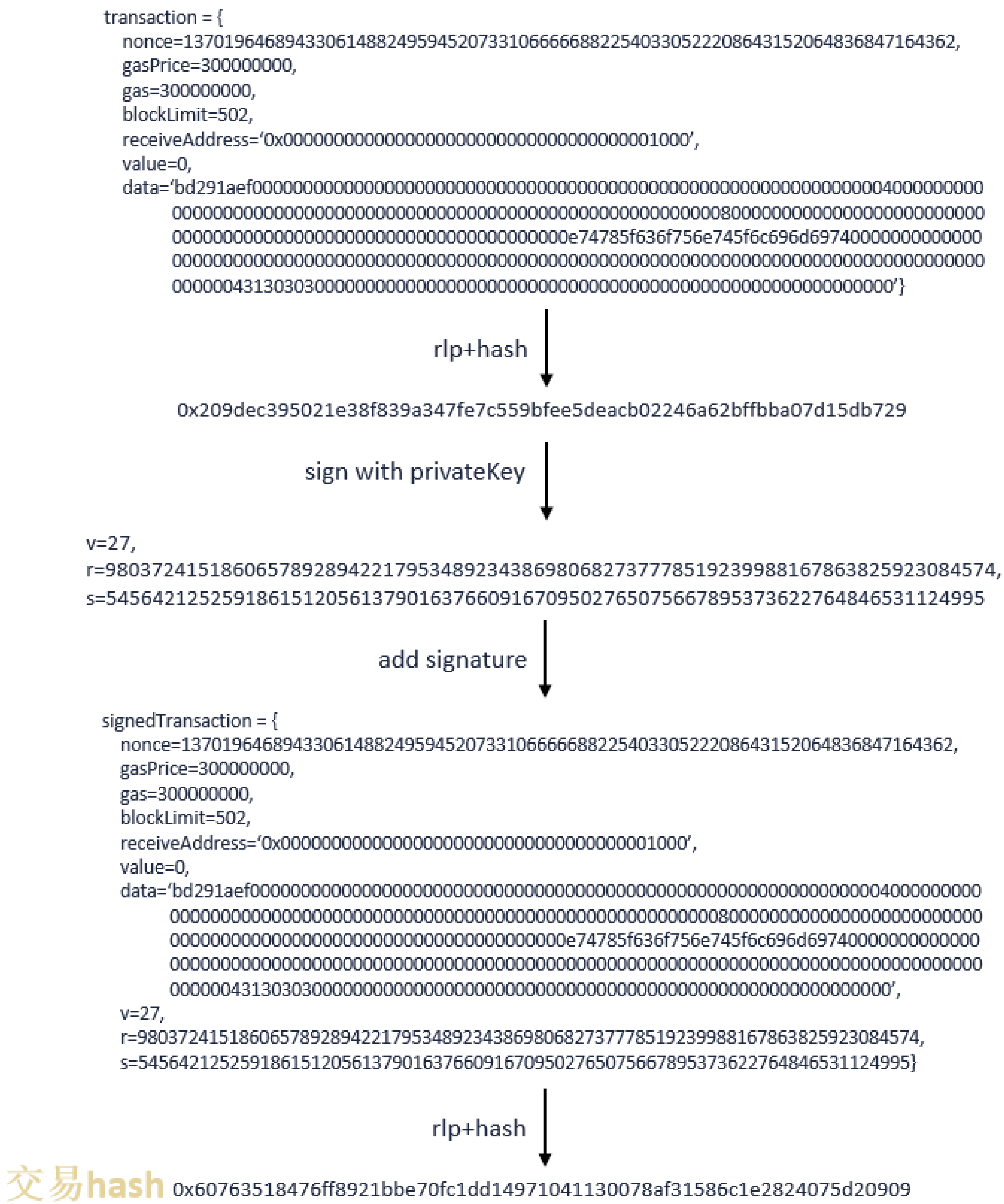
Refer to [RPC API Documentation] (../api.md)

10.9 Protocol description

10.9.1 Transaction structure and its RLP coding description

The transaction structure of FISCO BCOS has been increased or decreased some fields based on the transaction structure of the original Ethereum. The transaction structure fields of FISCO BCOS 2.0.0 are as follows:

The generation process of the hashWith field (also called transaction hash/transaction unique identifier) in RC1 is as follows:



It is similar with RC2 generation process, only that the transaction struct of `rlp+hash` in the first step is added with chainId, groupId, extraData the three fields.

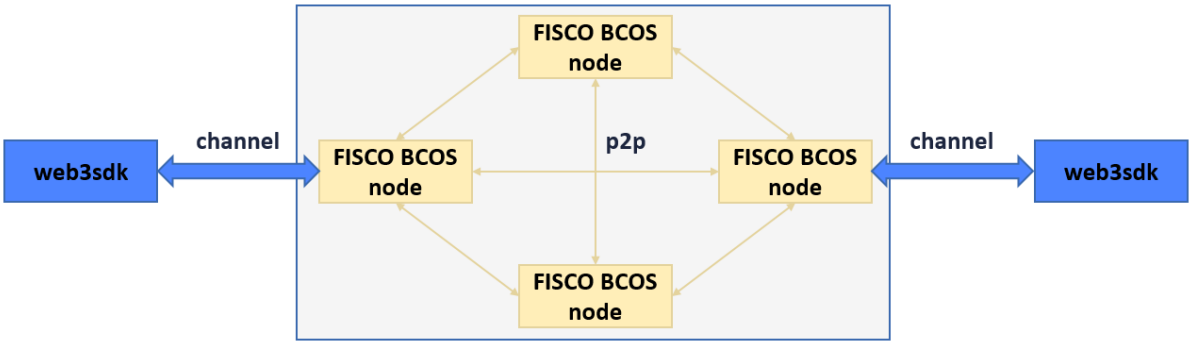
10.9.2 Block structure and its RLP coding description

The block of FISCO BCOS consists of the following five parts:

The description of each field in the block header of FISCO BCOS is as follows:

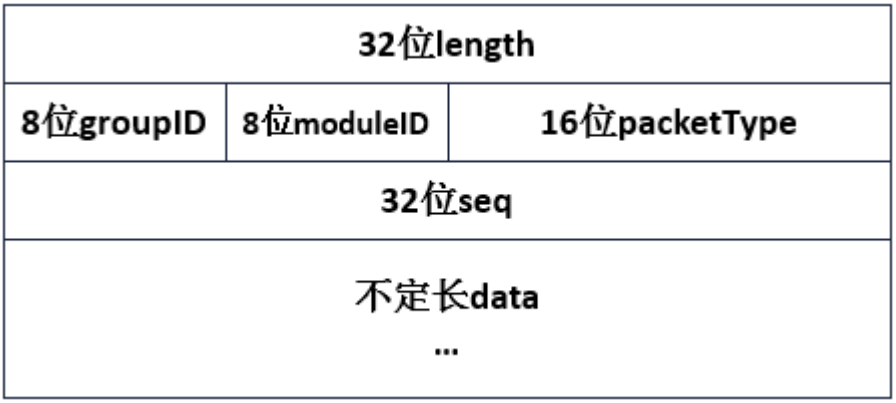
10.9.3 Network transmission protocol

FISCO BCOS currently has two types of data packet formats. The data packets communicated among nodes are in the P2PMessage format, and the data packets communicated between nodes and SDK are in the ChannelMessage format.



P2PMessage: v2.0.0-rc1

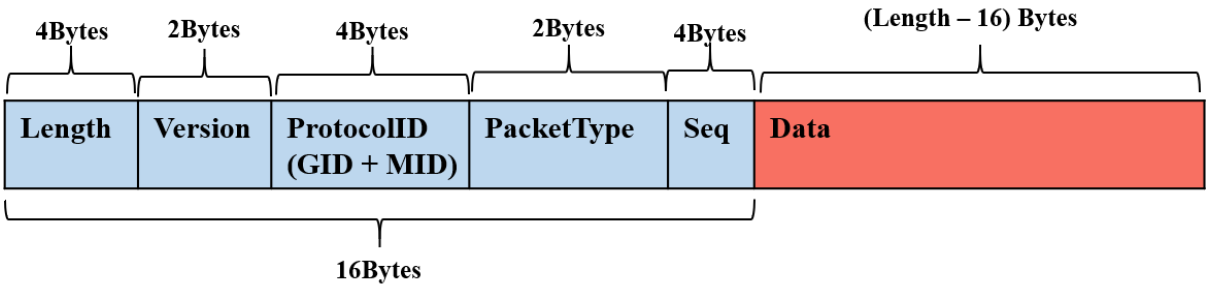
The header of v2.0.0-rc1 P2PMessage package contains 12 bytes. The basic form is:



The module ID is divided as follows:

P2PMessage: v2.0.0-rc2

V2.0.0-rc2 has expanded the range of **group ID and model ID**, **supporting 32767 groups at most**. It has also increased **Version** field for other features (like network compression) with package header being 16 bytes. The network data package structure of v2.0.0-rc2 is as below:



Additional

1. P2PMessage does not limit the packet size, and the packet size management is performed by the upper layer calling module (consensus/synchronization/AMOP, etc.);

2. The group ID and module ID can uniquely identify the protocol ID, and the relationship among them is `protocolID = (groupID << sizeof(groupID)*8) | ModuleID;`
3. The data packet distinguishes between request packet and response packet by the 16-bit binary value where the protocolID is located. The data greater than 0 is the request packet, and less than 0 is the corresponding packet.
4. The packetType currently used by AMOP include `SendTopicSeq = 1, RequestTopics = 2, SendTopics = 3`.

ChannelMessage

The packet type enumeration value and its corresponding description are as follows:

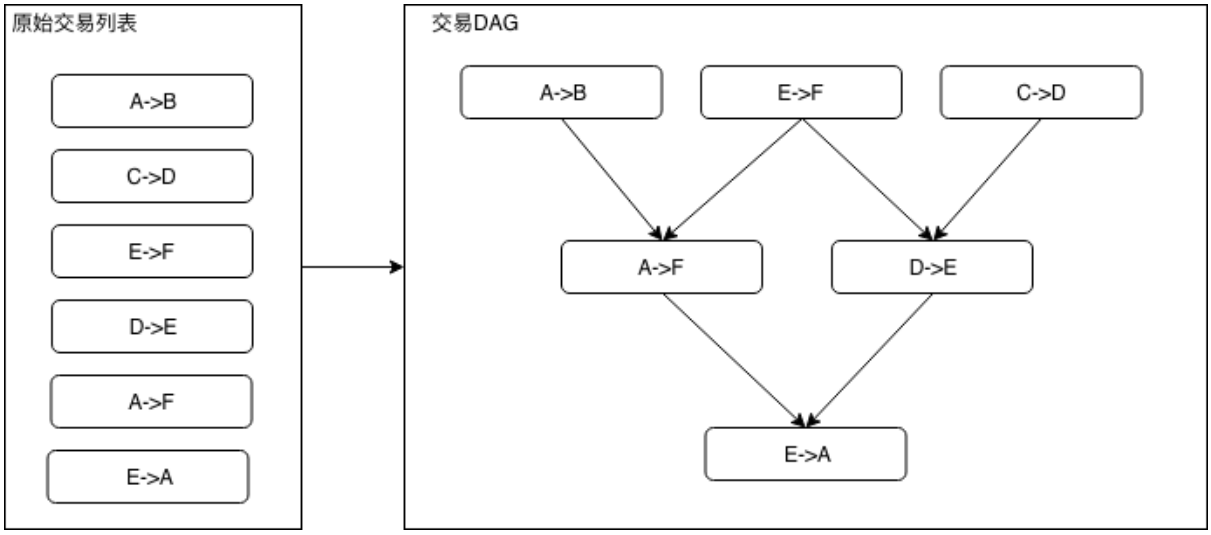
The process result enumeration value and its corresponding description are as follows:

10.10 Transaction parallel

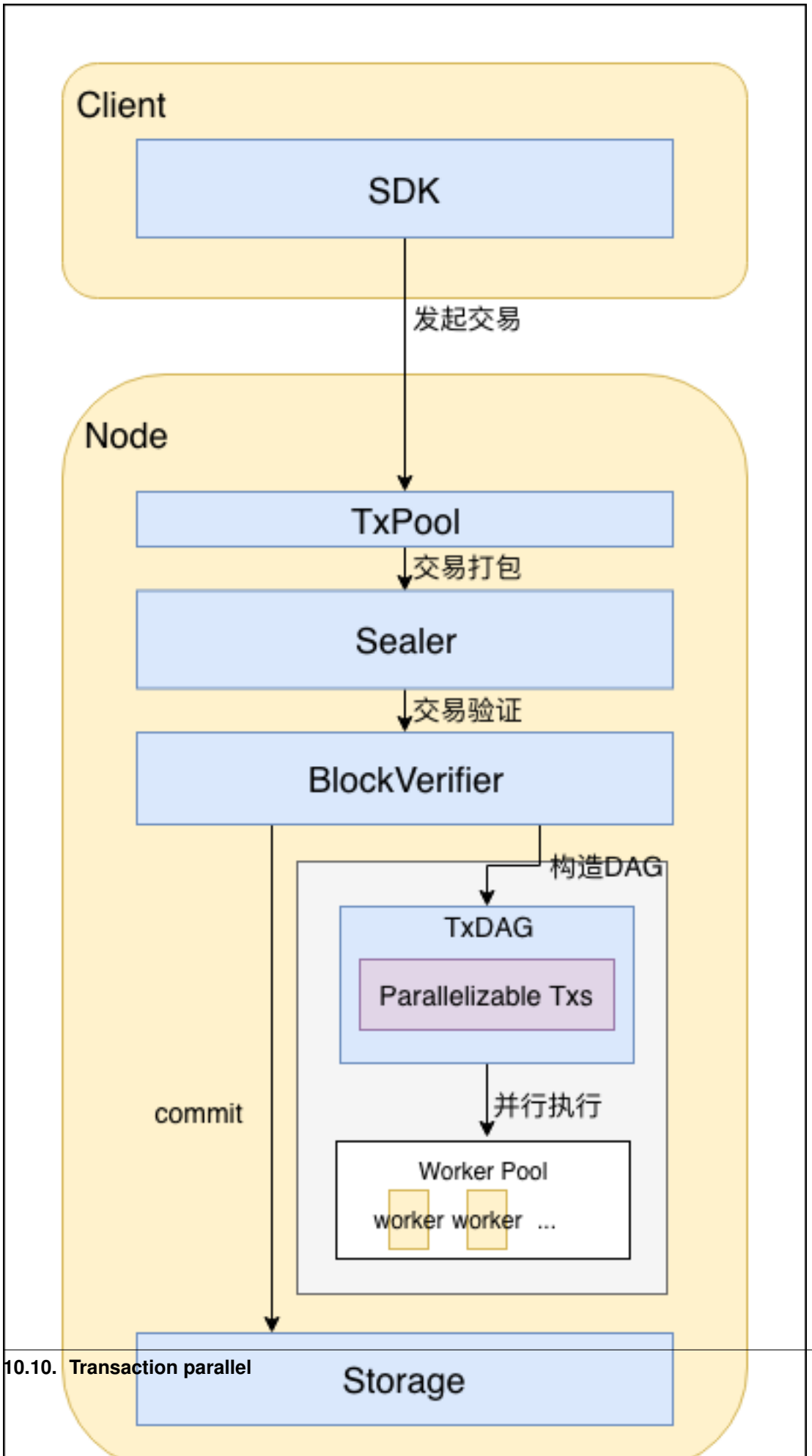
10.10.1 1 Terms definition

1.1 DAG

An acyclic directed graph is called **D**irected **A**cyclic **G**raph, or DAG. In a batch of transactions, it can recognize the mutually exclusive resource to be occupied in each transaction, and make a transaction dependency DAG according to the sequence of transaction in block and the occupation relationship of mutually exclusive resource. As the picture shows below, transaction with 0 in-degree (no dependent pre-order task) can be executed in parallel. By topological sort based on the sequence of the initial transaction list in the left graphic, you can get transaction DAG in the right graphic.



10.10.2 2 Model structure

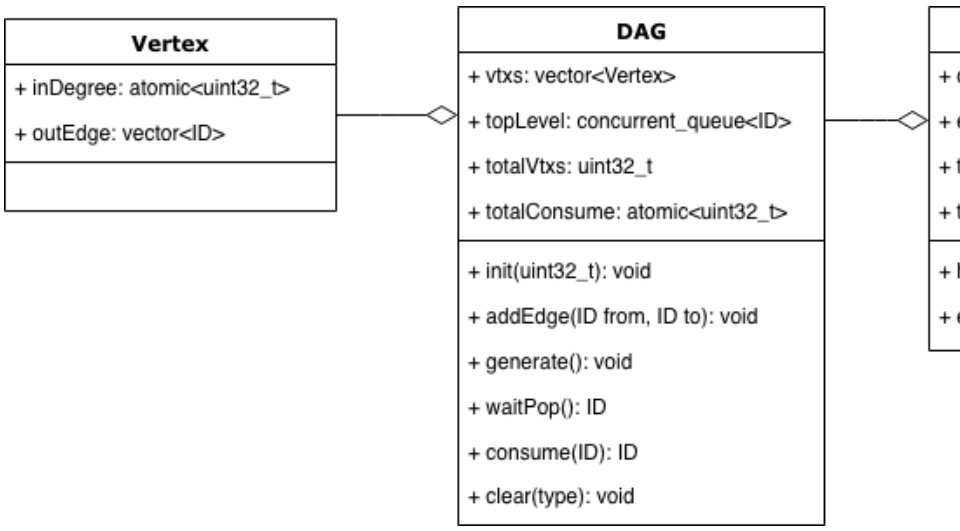


- user directly and indirectly initiates transaction through SDK, transaction can be either executed in parallel or not;
- transaction enters txPool and waits to be sealed;
- transaction gets sealed to block by Sealer and sent to BlockVerifier for verification after consensus;
- BlockVerifier generates transaction DAG according to the transaction list in block;
- BlockVerifier builds the execution context and executes transaction DAG;
- block is on chain after verified.

10.10.3 3 Crucial process

3.1 Building of transaction DAG

3.1.1 DAG data structure



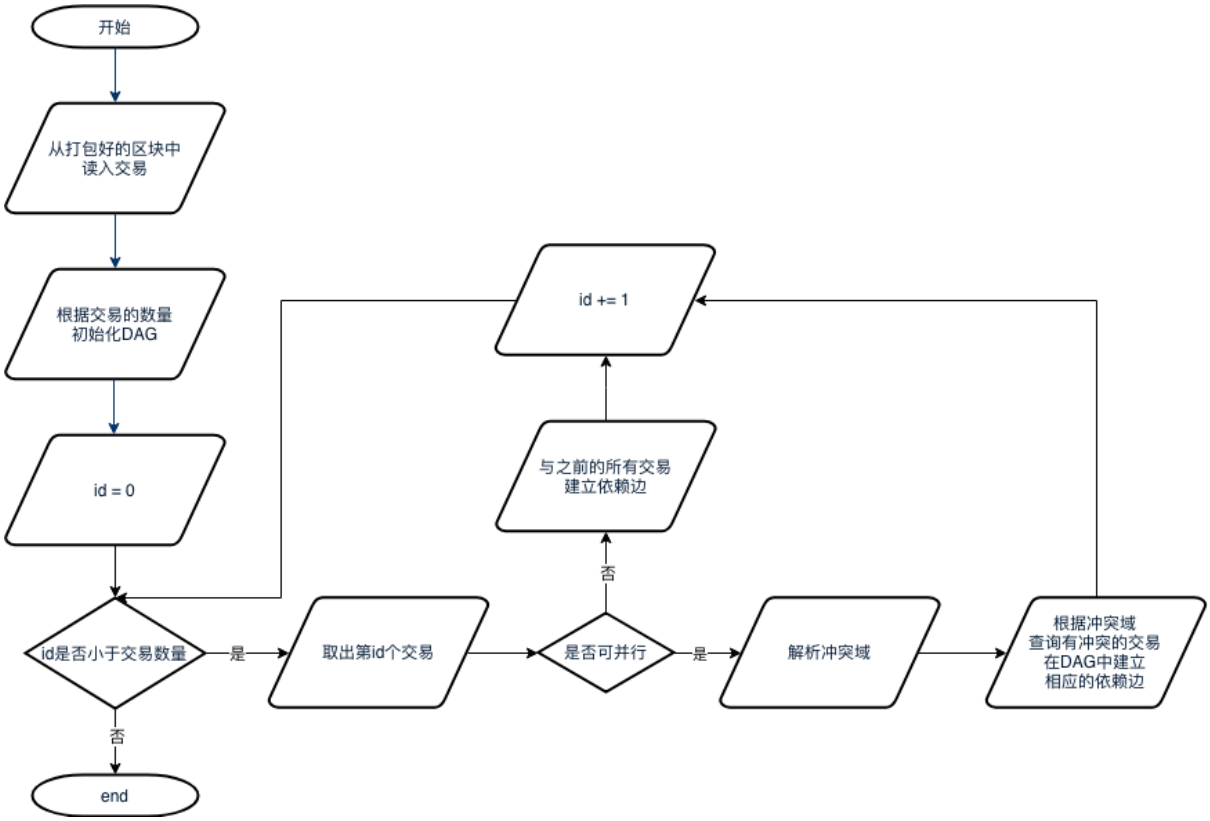
The DAG data structure in the solution is:
Note:

- Vertex
  - inDegree stores the current in-degree of Vertex;
  - outEdge stores outedge information of the Vertex, containing ID list of all out-edge connected Vertexs.
- DAG：
  - vtxs stores the list of all nodes in DAG;
  - topLevel is a concurrent queue storing node ID with 0 in-degree, supports concurrent access of multiple threads;
  - totalVtxs： total vertexs
  - totalConsume： total executed vertexs;
  - void init(uint32_t _maxSize)： initialize the maximum vertex to get maxSize DAG;
  - void addEdge(ID from, ID to)： set a directed edge between vertex from and to;
  - void generate()： build a DAG structure by the existed edges and vertexs;
  - ID waitPop(bool needWait)： wait to extract a 0 in-degree node from topLevel;
  - void clear()： clear all nodes and edges information in DAG.
- TxDAG:

- dag：DAG examples
- exeCnt：executed transactions count;
- totalParaTxs：total parallel transactions;
- txs：parallel transactions list;
- bool hasFinished()：return true if the DAG is executed, return false if not;
- void executeUnit()：take out a transaction without upper level dependency and execute;

3.1.2 Transaction DAG building process

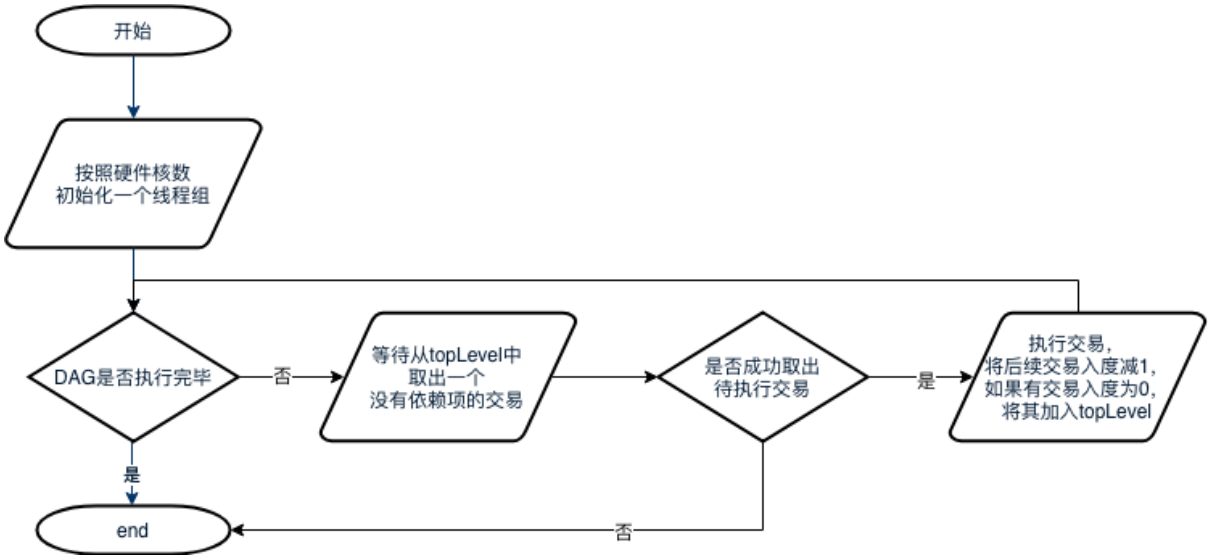
Process:



1. take out all transaction from sealed blocks;
2. initialize a DAG example with transactions amount as the maximum vertex amount;
3. read all transactions in sequence. If a transaction can be executed in parallel, analyze the collision domain and check if it collides to former transactions, if does, build dependent edge between transactions; if a transaction cannot be executed in parallel, it should be executed after all pre-order transactions are executed, and so there will be a dependent edge between it and the pre-order transactions.

3.2 DAG execution process

Process:



1. Main thread will initialize a same size thread group according to hardware coreness. If it fails to get hardware coreness, then other threads will not be created;
2. When DAG is still in execution, thread waits in loop to pop out 0 out-in degree transaction。 If it works, the transaction will be executed and the in-degree of its later dependent task minus 1. If transaction in-degree is minus to 0, it will be added to topLevel; if fails, then it means that DAG has finished execution and thread has been logged out.

10.11 Other features

For better user experience in smart contract calls and higher security, FISCO BCOS adopts Contract Name Service, or CNS, OSCCA alogorithm and disk encryption.

- **Contract Name Service**

Smart contracts are called by address in Ethereum, which may occurs to following problems:

- contract abi is a long JSON string, no need to be sensed by caller
- contract address is 20-byte magic number, difficult to remember and if lost contract will not be accessible
- One or more callers need to update contract address for re-deployment
- inconvenient for version management and grey release of contract

CNS of FISCO BCOS offers records of map relations between contract name and contract address and query function, so caller can call contract through easier contract name.

- **OSCCA alogorithm**

To support home-made cryptographic algorithm, FISCO BCOS has realized and integrated OSCCA encryption and decryption, signature, signature verification, hash, OSCCA SSL communication protocol to fully support **business encryption proved by OSCCA**.

- **Disk encryption**

Considering that data is accessible to each agency in consortium chain structure, FISCO BCOS adopts disk encryption to encrypt data stored in node database and key manager to store encryption key, ensuring data secrecy.

10.11.1 CNS

Introduction

The process to call smart contract in Ethereum includes:

1. program contract;
2. compile contract and get abi description of API;
3. deploy contract and get address;
4. encapsulate contract abi and address, and call contract by SDK or other tools.

From the process of calling contract, we know that contract abi and address are needed. This can lead to following problems:

1. contract abi is a long JSON string, no need to be sensed by caller;
2. contract address is a 20-byte magic number, difficult to remember, and contract cannot be accessed again if lost it;
3. one or more callers need to update contract address after re-deployment;
4. inconvenient for version management and grey release of contract.

To solve the problems and offer better experience for callers, FISCO BCOS has proposed **CNS**.

Terms definition

- **CNS** (Contract Name Service) offers records of map relations between contract name and address and query function, so caller can call contract by easier contract name.
- **CNS information** includes contract name, version, address and abi.
- **CNS table** stores CNS information

Advantages of CNS

- simplify the call of contract;
- support transparent update and grey release of contract.

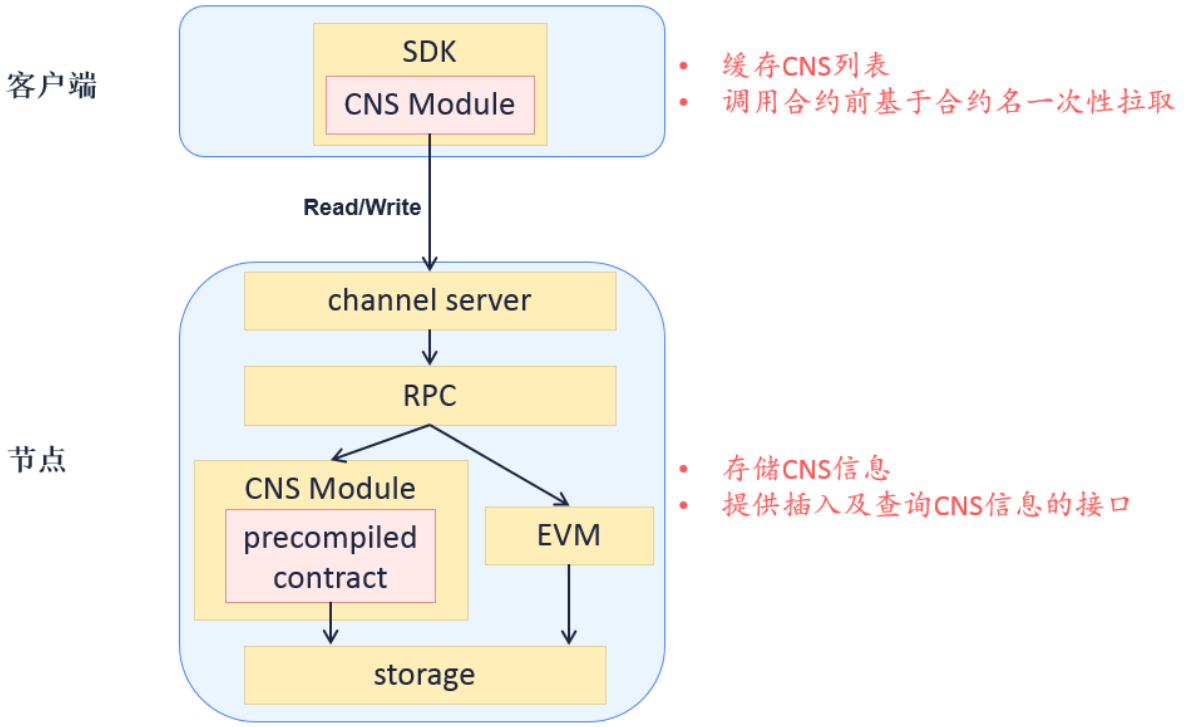
Comparison with ENS

ENS is Ethereum Name Service.

ENS has similar functions with DNS(Domain Name Service), though it doesn't offer Internet address. It expresses contract address and wallet address in the form of xxxxxx.eth web address for contract storage and payment transfer. Comparing it with CNS:

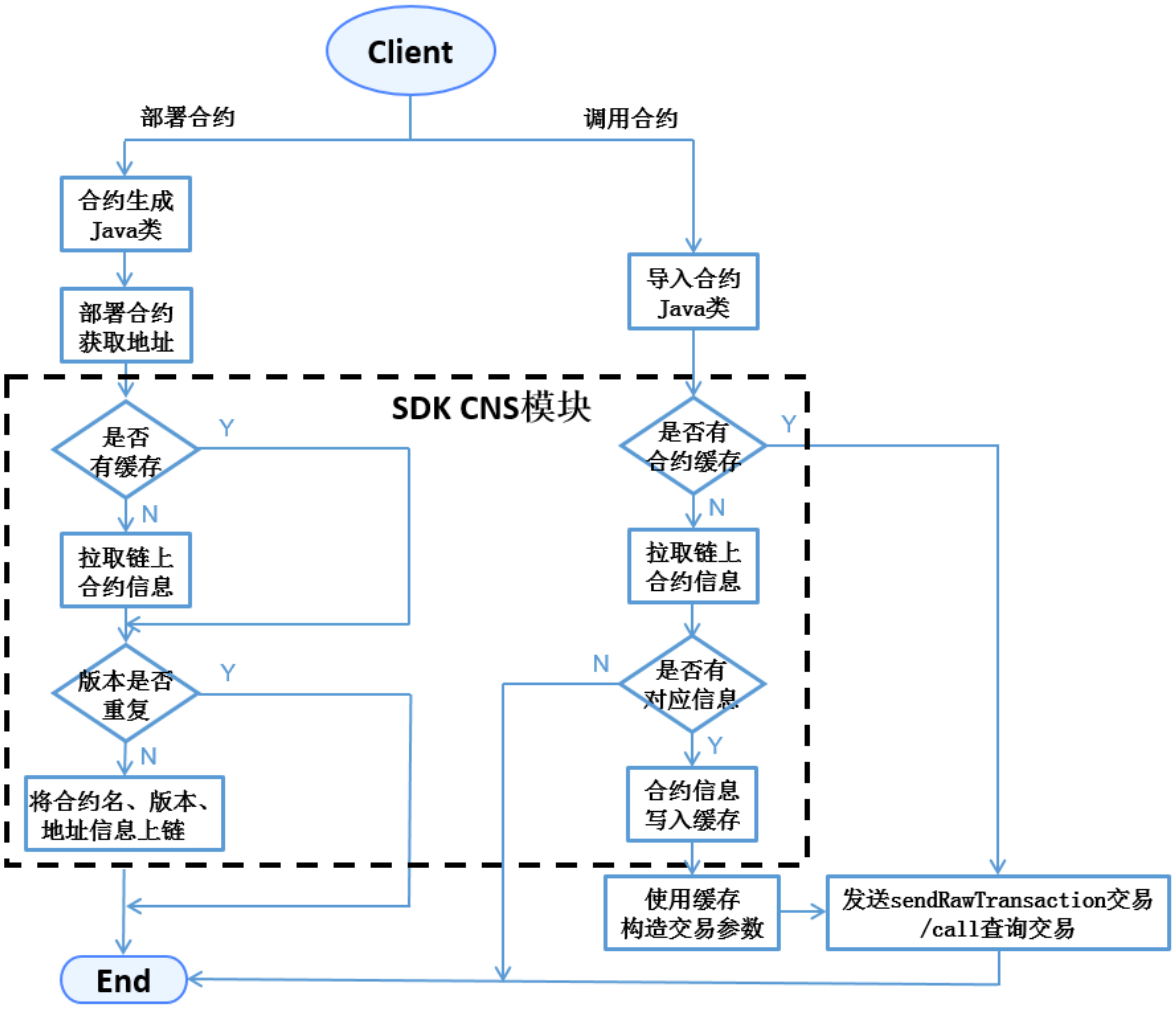
- ENS maps both contract address and wallet address, so does CNS; contract abi will be empty when it's wallet address.
- ENS has auction function, CNS does not.
- ENS supports multi-level domain name, CNS does not.

Model structure



Core process

The processes to deploy contract and call contract by SDK are:



- when deploying contract, SDK generates Java class of contract, call its deploy API to get address, and call insert API of CNS contract to write CNS information on chain.
- when calling contract, SDK imports Java class of contract and loads instantiation. Load API can input contract address (in Ethereum type) or contract name and composition of contract version (in CNS type). SDK handles CNS and gets address by calling CNS model.
- For contract lacking of version number, SDK is defaulted to call the latest version of contract.
- abi information of contract on chain belongs to optional fields.

Data structure

CNS table structure

CNS information is stored in system tables with independent ledgers. Definition of CNS table:

Contract API

```
pragma solidity ^0.4.2;
contract CNS
{
    function insert(string name, string version, string addr, string abi) public␣
↪returns(uint256);
    function selectByName(string name) public constant returns(string);
```

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

- CNS contract is not exposed to users. It is the interaction API of SDK and CNS table.
- insert API can write CNS information to blockchain, containing 4 parameters: contract name, version, addr and abi information. SDK needs to verify if the composition of name and version has been already recorded in database. Only when it's not repeated can it be written on chain for transaction. When node executing transactions, precompiled logic will Double Check the data, discard the transaction if repeated. Insert API will only increase but not modify contents in CNS table.
- selectByName API's parameter is contract name, returns all version records of this contract.
- selectByNameAndVersion API's parameter are contract name and version, returns the unique address of this version of contract.

Update CNS table

Precompiled contract is an efficient smart contract implemented by C++ in FISCO BCOS structure for configuration and management of system information. The process of transaction execution of nodes in precompiled logic adopted by FISCO BCOS is:

CNS contract belongs to precompiled contract. Nodes insert and inquire CNS table through the built in C++ code logic, not through EVM. So CNS contract offers API description of function but not implementation. **CNS contract's precompiled address is presetted to 0x1004.**

Return of contract API

selectByName and selectByNameAndVersion API returns string in Json. Example:

```
[
    {
        "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 developer can realize registration and query of CNS through the following 2 APIs in `org.fisco.bcos.web3j.precompile.cns`.

registerCns

- Description：public TransactionReceipt registerCns(String name, String version, String addr, String abi)
- Function：write contract information to chain
- Parameter：name——contract name, version——contract version, addr——contract address, abi——contract abi
- Return：transaction receipt, containing result and error information (if there is).

resolve

- Description：public String resolve(String contractNameAndVersion)
- Function：inquire contract address based on contract name and version
- Parameter：contractNameAndVersion——contract name+contract version information
- Return：contract address, API throw exception if no contract information specified in parameters
- Illustration：contractNameAndVersion splits contract name and version through `:`, SDK is defaulted to call the latest version of contract for query when lacking of version information

Note：

1. Before calling API, sol contract needs to be transferred to Java class and placed it to the right folder together with abi, bin files. For detail operations please check Web3SDK;
2. The operation examples of the 2 API are introduced in the implementation of deployByCNS and callByCNS APIs inConsoleImpl.java.

Operation tool

Console offers functions to deploy, call contract and inquire on-chain contract based on name. For the detail operations please check Console.

Commands of console:

- deployByCNS：deploy contract through CNS
- callByCNS：call contract through CNS
- queryCNS：inquire CNS table information by contract name, version number (optional parameter)

10.11.2 OSCCA algorithm

Design objective

Bases on [national cryptography standard] (http://www.gmbz.org.cn/main/bzlb.html), FISCO has realized the national encryption and decryption, signature, verification, Hash algorithm, national cryptography SSL communication protocol, and integrated into the FISCO BCOS platform, and to achieve full support for **commercial password identified by the National Cryptographic Bureau**.

The national cryptography version of FISCO BCOS replaces the cryptographic algorithms of the underlying modules such as transaction signature verification, p2p network connection, node connection, and data disk encryption with the national cryptography algorithm.

The different features between national cryptography version of FISCO BCOS and the standard version are as follows:

(Note: National cryptography algorithms SM2, SM3, SM4 are developed based on [national cryptography standards] (http://www.gmbz.org.cn/main/bzlb.html))

System framework

The overall framework of the system is shown below:

National cryptography SSL 1.1 establish process

The authentication among the nodes of national cryptography FISCO BCOS selects the ECDHE_SM4_SM3 cipher suite of the national cryptography SSL 1.1 for the establishment of SSL connect. The differences are shown in the following table:

Data structure difference

The difference in data structure between the national cryptography version and the standard version of FISCO BCOS is as follows:

10.11.3 Disk encryption

Background

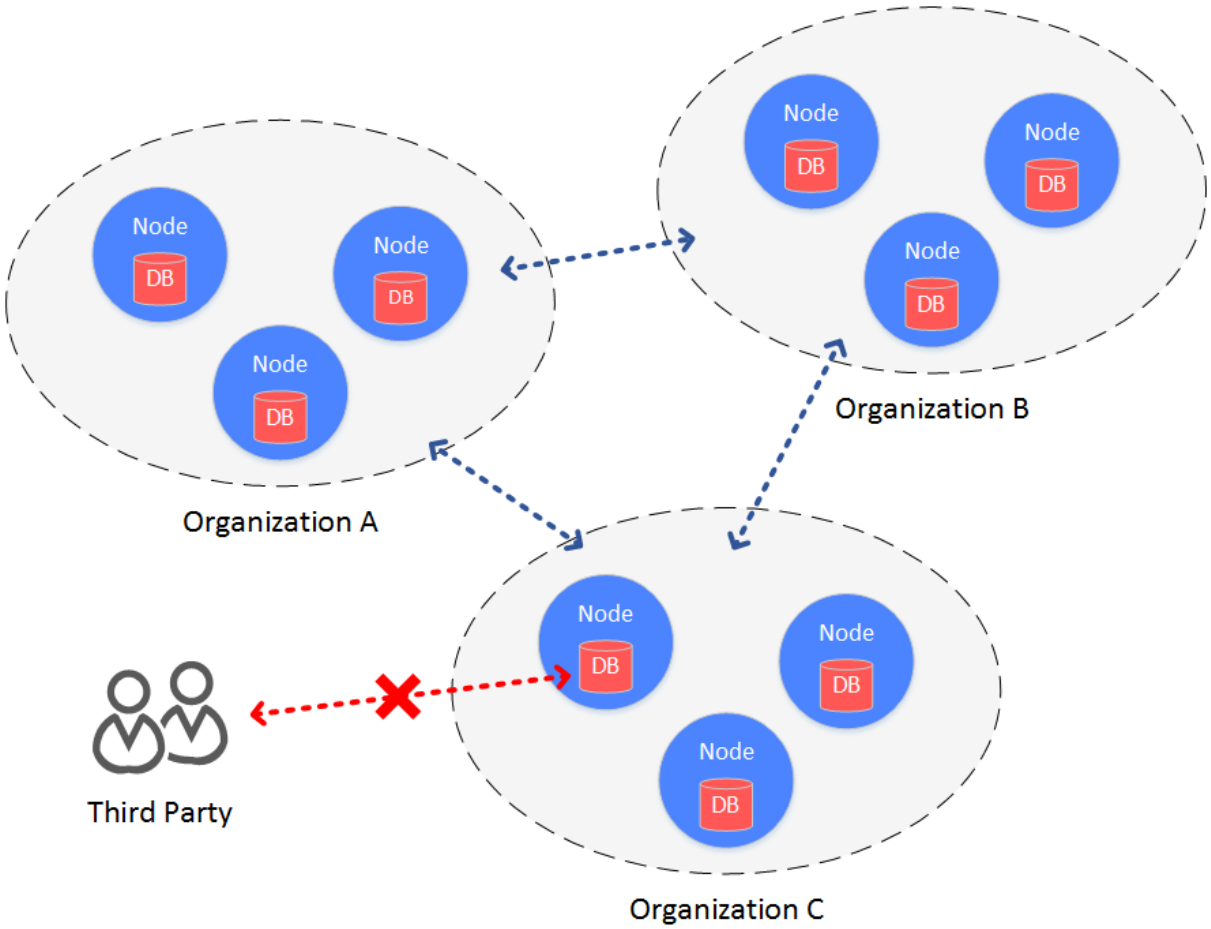
In consortium chain structure, all agencies build a chain between each other, making data accessible to each agency.

When it comes to cases the require high data security, consortium members want to prevent other agencies to access the data on blockchain. Therefore, access control needs to be adopted on the data on consortium chain.

There are 2 aspects of data access control in consortium chain:

- access control on communication data
- access control on node storage data

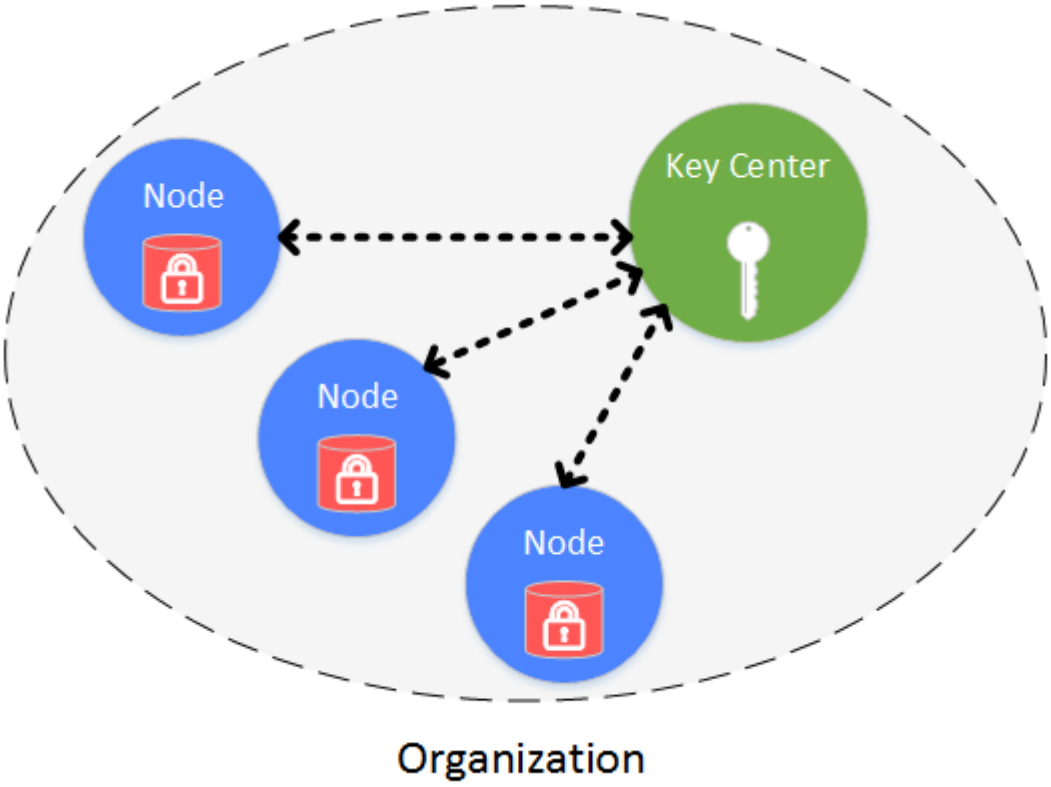
FISCO BCOS manages access of communication data through node certificate and SSL verification. The following context introduces about its access control on node storage data, which is, disk encryption.



Concept

Disk encryption is conducted inside each agency. Each agency encrypts the disk of node data internally. When the disk is taken away from the agency and the node is started in external network, the disk will be unable to be decrypted and the node fail to be started. So data in the consortium chain will be well-protected.

Solution



Disk encryption is conducted and managed securely and independently by each agency. Disk of each node is encrypted. The access of encrypted data is managed by key manager, which is deployed inside agency and manages key to node disk data that is not open to outside network. When the node is started, it will acquire the key from key manager to access its own encrypted data.

The following objects are encrypted:

- node local database: leveldb
- node private key: node.key， gmnode.key (OSCCA)

Implementation

The implementation of disk encryption is realized by dataKey (hold by node itself) and superKey (managed by key manager).

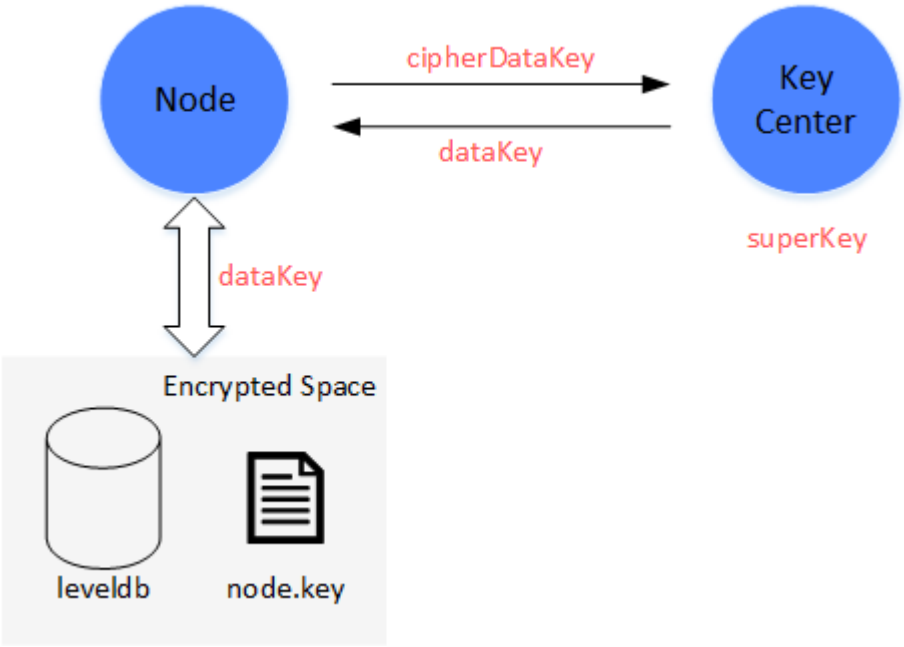
Node

- node encrypts and decrypts its Encrypted Space by dataKey.
- node doesn't store dataKey in local disk, but stores the encrypted dataKey-cipherDataKey.
- When node is started, it requests dataKey from key manager by cipherDataKey.
- DataKey is only in storage of node, when node is stoped, dataKey will be discarded.

Key Manager

Key manager holds superKey and responds to the access requests from all started nodes.

- Key Manager has to be on line in real time to respond to nodes' start request.
- When node is started, it will send cipherDataKey to Key Manager to decrypt it by superKey. If succeed, key manager will return dataK to node.
- Key Manager can only be accessed within internal network.



Process

Process of disk encryption includes node initial config and node safe operation.

Node initial config

Before started, dataKey of node needs to be configured

Important: When node is generated, before started, it has to be decided whether to adopt disk encryption. Once configured and started, node's status cannot be transferred.

（1） Manager defines dataKey of node and sends to Key Manager to acquire cipherDataKey.

（2） Configure cipherDataKey to node config file

（3） Start node

Node safe operation

When node is started, it will acquire dataKey from key manager to access local data.

（1） Start node, read cipherDataKey in config file and send to Key Manager.

（2） Key Manager receives cipherDataKey and decrypts it using superKey, sends the decrypted dataKey back to node.

（3） Node gets dataKey to interact with local data (Encrypted Space). Data in Encrypted Space will be decrypted by dataKey, which is also needed for encryption when writing data to Encrypted Space.

How can it protect data?

When a node's disk is accidentally brought to external network, the data will not be exposed.

（1） When node is started in external network, it will fail to connect with Key Manager and acquire dataKey, even though is has cipherDataKey.

（2）When node is not started, the local data can not be exposed in that there is no dataKey for decryption of Encrypted Space.

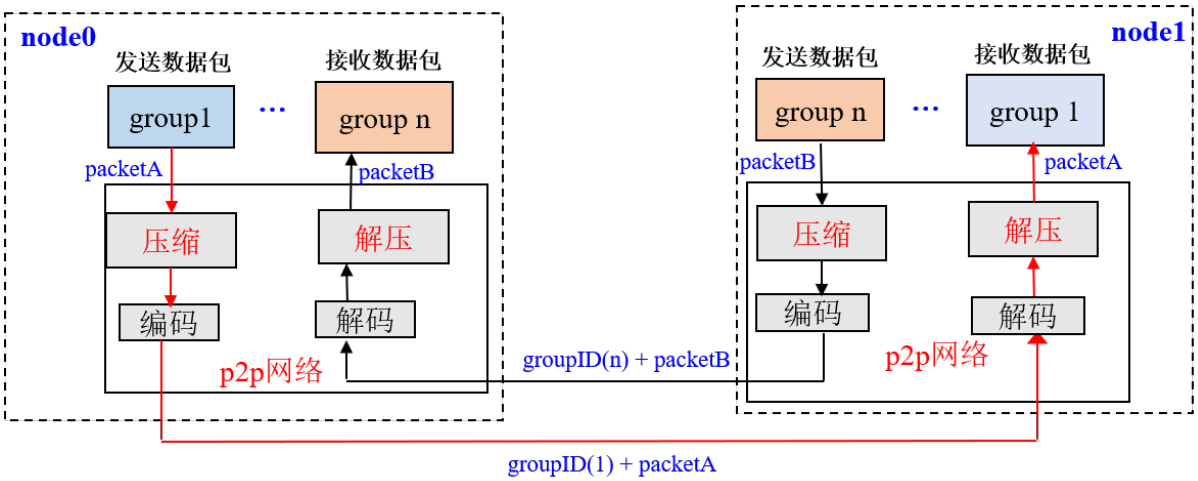
The detail operations of disk encryption are introduced here: Operations of disk encryption.

10.11.4 Network compression

In external network environment, the performance of blockchain system is limited by the network bandwidth. For minimizing the impact of network bandwidth on system performance, FISCO BCOS supports network compression from version `relase-2.0.0-rc2`. This function mainly performs network packet compression on transmit(tx) data (TXD) and packet uncompression on receive(rx) data (RXD), and transmits the unpacked data to the upper module.

System framework

Network compression is mainly implemented in P2P network underlying. The system framework is as follows:



Network compression mainly consists of two processes:

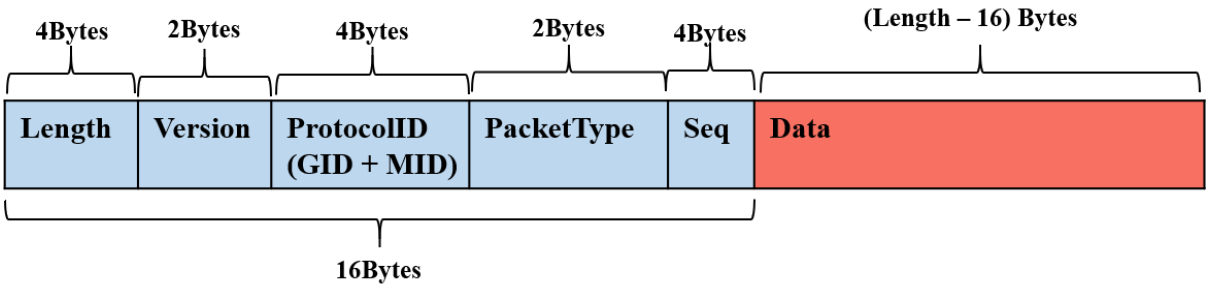
- **packet compression on transmit(tx) data**: When the group layer sending data through the P2P layer, if the data packet size exceeds 1 KB, then packet is sent to the target node after compression.
- **packet uncompression on receive(rx) data**: After receiving the data packet, the node first determines whether the received data packet is compressed. If the data packet is compressed, to decompress it and transmit to the specified group, otherwise to transmit the data transmitted to the corresponding group directly.

Core implementation

Considering performance, compression efficiency, and etc., we selected Snappy to implement data packet compression and decompression. In this section, we mainly introduce the implementation of network compression.

Data compression flag

FISCO BCOS's network packet structure is as follows:

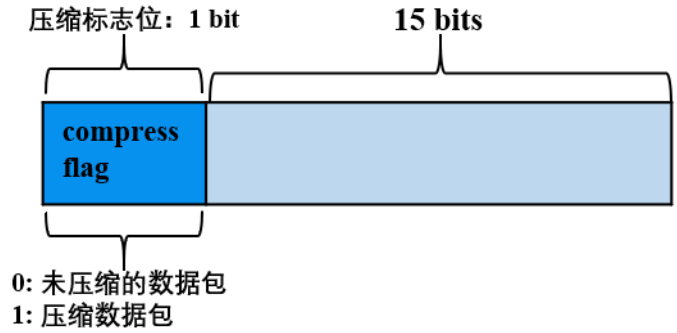


The network data packet mainly includes two parts: header and data. The header contains 16 bytes. The meanings of the fields are as follows:

- Length: the length of data packet.
- Version: extension bit, for extending network module function.
- ProtocolID: storing the group ID and module ID of Destination Network Address Translation (DNAT) for multi-group packet routing. Currently it supports up to 32767 groups.
- PaketType: tagged data packet type.
- Seq: data packet serial number

**Network compression module only compresses network data but not data packet header. **

Considering that compressing and decompressing small data packets can not save data space and waste performance, in the data compression process, the undersize packets are not compressed, and only the data packets with size larger than `c_compressThreshold` are compressed. The default value of `c_compressThreshold` is 1024 (1KB). We have extended the highest bit of Version as a packet compression flag:



- When the highest value of Version is 0, indicating that the data which is corresponding to data packet is uncompressed.
- When the highest value of Version is 1, indicating that the data which is corresponding to data packet is compressed.

Processing flow

In the following, we take a node in group1 sending network message packet groupA to other nodes(such as sending a transaction, a block, a consensus message packet, etc.) as an example to detail the key processing flow of network compression module.

Transmit(tx) data processing flow:

- Group1's group module passes packetA to P2P layer;
- When P2P determines that the packetA is greater than `c_compressThreshold`, then calls the compression interface to compress packetA, otherwise it directly passes packetA to the encoding module;

- The encoding module adds header to packetA with data compression information, ie: if packetA is compressed, the highest value of Version(header) is set to 1; otherwise, it is set to 0;
- P2P transmits the encoded data packet to the destination node.

Receive(rx) data processing flow:

- After the target machine receives the data packet, the decoding module separates the packet header, and determines whether the network data is compressed by the highest value of Version is 1 or not;
- If the network packet is compressed, the decompression interface is called to decompress part of data, and transmit the decompressed data to the specified module of group according to the GID and PID attached to the packet header; otherwise, the data packet is directly passed to the upper module.

Compatibility note

- **Data Compatibility**: not involve the changes of stored data;
- **Network Compatibility rc1**: Forward compatible, only the relase-2.0.0-rc2 node has network compression.

CHAPTER 11

JSON-RPC API

The following examples in this chapter adopt curl command, which is a data transfer tool run under command line in url language. JSON-RPC API of FISCO BCOS can be accessed by sending http post request through curl command. The url address of curl command is set as `[listen_ip]` and `[jsonrpc listen port]` of `[rpc]` in node config file. To format json, jq is used as a formatter. For the error codes, please check RPC Design Documentation. For transaction return list, please check here.

11.1 getClientVersion

return version information of node

11.1.1 Parameter

no

11.1.2 Return value

- `object` - version information, fields:
  - `Build Time`: `string` - compile time
  - `Build Type`: `string` - compile machine environment
  - `FISCO-BCOS Version`: `string` - FISCO BCOS version
  - `Git Branch`: `string` - version branch
  - `Git Commit Hash`: `string` - the newest commit hash of version
- Example

```
// 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"
  }
}
```

11.2 getBlockNumber

return the newest block number of specific group

11.2.1 Parameter

- `groupID`: `unsigned int` - group ID

11.2.2 Return value

- `string` - the highest block number (hexadecimal string start with 0x)
- Example

```
// 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"
}
```

11.3 getPbftView

return the newest PBFT View of the specific group

11.3.1 Parameter

- `groupID`: `unsigned int` - group ID

11.3.2 Return value

- `string` - the newest PBFT view
- Example

```
// 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"
}
```

Note: FISCO BCOS supports PBFT Consensus and Raft Consensus. When the blockchain adopts Raft consensus, the custom error returned by the API is:

```
{
  "error": {
    "code": 7,
    "data": null,
    "message": "Only pbft consensus supports the view property"
  },
  "id": 1,
  "jsonrpc": "2.0"
}
```

11.4 getSealerList

return the consensus nodes list of the specific group

11.4.1 Parameter

- `groupID`: `unsigned int` - group ID

11.4.2 Return value

- `array` - consensus node ID list
- Example

```
// 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
↪",

↪"622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa3aa0eb5a3b802d3a8e5e26de9d5a
↪"
    ]
}
```

11.5 getObserverList

return observer node list of the specific group

11.5.1 Parameter

- `groupID`: `unsigned int` - group ID

11.5.2 Return value

- `array` - observer node ID list
- Example

```
// 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
↪"
    ]
}
```

11.6 getConsensusStatus

return consensus status information of the specific group

11.6.1 Parameter

- `groupID`: `unsigned int` - group ID

11.6.2 Return value

- `object` - consensus status information.
- 1. When PBFT consensus mechanism is used,（PBFT design is introduced in PBFT Design Documentation), the fields are:
  - `accountType`: `unsigned int` - account type
  - `allowFutureBlocks`: `bool` - allow future blocks
  - `cfgErr`: `bool` - configure errors
  - `connectedNodes`: `unsigned int` - connected nodes
  - `consensusedBlockNumber`: `unsigned int` - the newest consensus block number

  - `currentView`: `unsigned int` - the current view
  - `groupId`: `unsigned int` - group ID
  - `highestblockHash`: `string` - the hash of the highest block
  - `highestblockNumber`: `unsigned int` - the highest block number
  - `leaderFailed`: `bool` - leader failed
  - `max_faulty_leader`: `unsigned int` - the max number of faulty nodes
  - `sealer.index`: `string` - node ID with sequence number "index"
  - `node index`: `unsigned int` - sequence number of node
  - `nodeId`: `string` - node ID
  - `nodeNum`: `unsigned int` - number of nodes
  - `omitEmptyBlock`: `bool` - omit empty block
  - `protocolId`: `unsigned int` - protocol ID
  - `toView`: `unsigned int` - current view value
  - `prepareCache_blockHash`: `string` - prepareCache hash
  - `prepareCache_height`: `int`- prepareCache height
  - `prepareCache_idx`: `unsigned int` - prepareCache sequence number
  - `prepareCache_view`: `unsigned int` - prepareCache view
  - `rawPrepareCache_blockHash`: `string` - rawPrepareCache hash
  - `rawPrepareCache_height`: `int`- rawPrepareCache height
  - `rawPrepareCache_idx`: `unsigned int` - rawPrepareCache sequence number
  - `rawPrepareCache_view`: `unsigned int` - rawPrepareCache view
  - `committedPrepareCache_blockHash`: `string` - committedPrepareCache hash
  - `committedPrepareCache_height`: `int`- committedPrepareCache height
  - `committedPrepareCache_idx`: `unsigned int` - committedPrepareCache sequence number
  - `committedPrepareCache_view`: `unsigned int` - committedPrepareCache view
  - `futureCache_blockHash`: `string` -futureCache hash
  - `futureCache_height`: `int`- futureCache height
  - `futureCache_idx`: `unsigned int` - futureCache sequence number
  - `signCache_cachedSize`: `unsigned int` - signCache_cached size
  - `commitCache_cachedSize`: `unsigned int` - commitCache_cached size
  - `viewChangeCache_cachedSize`: `unsigned int` - viewChangeCache_cached size

- 1. When Raft consensus mechanism is adopted (Raft design is introduced in Raft Design Documentation), the fields are:
  - `accountType`: `unsigned int` - account type
  - `allowFutureBlocks`: `bool` - allow future blocks
  - `cfgErr`: `bool` - configure error
  - `consensusedBlockNumber`: `unsigned int` - the newest consensus block number
  - `groupId`: `unsigned int` - group ID
  - `highestblockHash`: `string` - hash of the newest block

  - highestblockNumber: unsigned int - the highest block number
  - leaderId: string - leader node ID
  - leaderIdx: unsigned int - leader node sequence number
  - max_faulty_leader: unsigned int - the max number of faulty nodes
  - sealer.index: string - node ID with sequence number "index"
  - node index: unsigned int - index of node
  - nodeId: string - node ID
  - nodeNum: unsigned int - number of nodes
  - omitEmptyBlock: bool - omit empty block
  - protocolId: unsigned int - protocol ID

- Example

```
// Request PBFT
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,
            "connectedNodes":3,
            "consensusedBlockNumber":4,
            "currentView":153,
            "groupId":1,
            "highestblockHash":
↪"0x98e186095a88f7b1b4cd02e3c405f031950577626dab55b639e024b9f2f8788b",
            "highestblockNumber":3,
            "leaderFailed":false,
            "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":264,
            "toView":153
        },
        {
            "prepareCache_blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
```

```
            "prepareCache_height":-1,
            "prepareCache_idx":"65535",
            "prepareCache_view":"9223372036854775807"
        },
        {
            "rawPrepareCache_blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
            "rawPrepareCache_height":-1,
            "rawPrepareCache_idx":"65535",
            "rawPrepareCache_view":"9223372036854775807"
        },
        {
            "committedPrepareCache_blockHash":
↪"0x2e4c63cfac7726691d1fe436ec05a7c5751dc4150d724822ff6c36a608bb39f2",
            "committedPrepareCache_height":3,
            "committedPrepareCache_idx":"2",
            "committedPrepareCache_view":"60"
        },
        {
            "futureCache_blockHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
            "futureCache_height":-1,
            "futureCache_idx":"65535",
            "futureCache_view":"9223372036854775807"
        },
        {
            "signCache_cachedSize":"0"
        },
        {
            "commitCache_cachedSize":"0"
        },
        {
            "viewChangeCache_cachedSize":"0"
        }
    ]
}

// 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
    }
  ]
}
```

11.7 getSyncStatus

return the consensus status information of the specific group

11.7.1 Parameter

- `groupID`: `unsigned int` - group ID

11.7.2 Return value

- `object` - consensus status information, fields are:
  - `blockNumber`: `unsigned int` - the highest block number
  - `genesisHash`: `string` - hash of genesis block
  - `isSyncing`: `bool` - syncing
  - `knownHighestNumber`: `unsigned int` - the highest number of the blockchain known by the node
  - `knownLatestHash`: `string` - the latest hash of the blockchain known by the node
  - `latestHash`: `string` - hash of the newest block
  - `nodeId`: `string` - node ID
  - `protocolId`: `unsigned int` - protocol ID
  - `txPoolSize`: `string` - transaction volume in txPool
  - `peers`: `array` - connected p2p nodes in the specific group, fields of node information are:
    - `blockNumber`: `unsigned int` - the newest block number
    - `genesisHash`: `string` - hash of genesis block
    - `latestHash`: `string` - hash of the newest block
    - `nodeId`: `string` - node ID
- Example

```
// 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",
        "nodeId":
↪"29c34347a190c1ec0c4507c6eed6a5bcd4d7a8f9f54ef26da616e81185c0af11a8cea4eacb74cf6f61820292b24bc5
↪"
      },
      {
        "blockNumber": 0,
        "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "nodeId":
↪"87774114e4a496c68f2482b30d221fa2f7b5278876da72f3d0a75695b81e2591c1939fc0d3fadb15cc359c997bafc9
↪"
      },
      {
        "blockNumber": 0,
        "genesisHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "latestHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
        "nodeId":
↪"d5b3a9782c6aca271c9642aea391415d8b258e3a6d92082e59cc5b813ca123745440792ae0b29f4962df568f8ad58b
↪"
      }
    ],
    "protocolId": 265,
    "txPoolSize": "0"
  }
}
```

11.8 getPeers

return connected p2p node information

11.8.1 Parameter

- `groupID`: `unsigned int` - group ID

11.8.2 Return value

- `array` - connected p2p node information, fields are:
  - `IPAndPort`: `string` - IP and port of connected node
  - `nodeId`: `string` - node ID
  - `Topic`: `array` - Topic information followed by node
- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPeers","params":[1],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
格式化JSON:
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": [
        {
            "IPAndPort": "127.0.0.1:30308",
            "nodeId":
↪"0701cc9f05716690437b78db5b7c9c97c4f8f6dd05794ba4648b42b9267ae07cfcd589447ac36c491e760424214960
↪",
            "Topic": [ ]
        },
        {
            "IPAndPort": "127.0.0.1:58348",
            "nodeId":
↪"353ab5990997956f21b75ff5d2f11ab2c6971391c73585963e96fe2769891c4bc5d8b7c3d0d04f50ad6e04c4445c09e
↪",
            "Topic": [ ]
        },
        {
            "IPAndPort": "127.0.0.1:30300",
            "nodeId":
↪"73aebaea2baa9640df416d0e879d6e0a6859a221dad7c2d34d345d5dc1fe9c4cda0ab79a7a3f921dfc9bdea4a49bb3
↪",
            "Topic": [ ]
        }
    ]
}
```

11.9 getGroupPeers

return consensus node and observer node list in specific group

11.9.1 Parameter

- `groupID`: `unsigned int` - group ID

11.9.2 Return value

- `array` - consensus node and observer node ID list
- Example

```
// 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
↪",

↪"622af37b2bd29c60ae8f15d467b67c0a7fe5eb3e5c63fdc27a0ee8066707a25afa3aa0eb5a3b802d3a8e5e26de9d5a
↪",

↪"10b3a2d4b775ec7f3c2c9e8dc97fa52beb8caab9c34d026db9b95a72ac1d1c1ad551c67c2b7fdc34177857eada7583
↪"
    ]
}
```

11.10 getNodeIDList

return node and connected p2p node list

11.10.1 Parameter

- `groupID`: `unsigned int` - group ID

11.10.2 Return value

- `array` - node and connected p2p node ID list
- Example

```
// 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
↪"
    ]
}
```

11.11 getGroupList

return belonged group ID list

11.11.1 Parameter

no

11.11.2 Return value

- `array` - group ID list
- Example

```
// 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]
}
```

11.12 getBlockByHash

return block information inquired by block hash

11.12.1 Parameters

- `groupID`: `unsigned int` - group ID
- `blockHash`: `string` - block hash
- `includeTransactions`: `bool` - include transactions ("true" shows transaction details; "false" shows the hash of transaction only)

11.12.2 Return value

- `object` - block information, fields are:
  - `extraData: array` - extra data
  - `gasLimit: string` - the maximum gas allowed in block
  - `gasUsed: string` - gas used by all transactions
  - `hash: string` - block hash
  - `logsBloom: string` - bloom filter value of log
  - `number: string` - block height
  - `parentHash: string` - hash of parent block
  - `sealer: string` - consensus node sequence number
  - `sealerList: array` - consensus nodes list
  - `stateRoot: string` - hash of state root
  - `timestamp: string` - time stamp
  - `transactions: array` - transaction list, when `includeTransactions` is `false`, it shows the hash of transaction; when `includeTransactions` is `true`, it shows transaction details (detail fields please check getTransactionByHash)
- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",true],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",
    "logsBloom":
↪"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number": "0x1",
    "parentHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "sealer": "0x3",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bca
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966a
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
```

```
    "stateRoot":
↪"0xfb7ca5a7a271c8ffb51bc689b78d0aeded23497c9c22e67dff8b1c7b4ec88a2a",
    "timestamp": "0x1687e801d99",
    "transactions": [
      {
        "blockHash":
↪"0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",
        "blockNumber": "0x1",
        "from": "0xadf06b974703a1c25c621ce53676826198d4b046",
        "gas": "0x1c9c380",
        "gasPrice": "0x1",
        "hash": "0x022dcb1ad2d940ce7b2131750f7458eb8ace879d129ee5b650b84467cb2184d7
↪",
        "input":
↪"0x6080604052348015610010576000800160006101000a81548173ffffffffffffffffffffffff
↪",
        "nonce": "0x1a9d06264238ea69c1bca2a74cfced979d6b6a66ce8ad6f5a30e8017b5a98d8
↪",
        "to": null,
        "transactionIndex": "0x0",
        "value": "0x0"
      }
    ],
    "transactionsRoot":
↪"0x07506c27626365c4f0db788619a96df1e6f8f62c583f158192700e08c10fec6a"
  }
}

// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByHash","params":[1,
↪"0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",false],"id
↪":1}' http://127.0.0.1:8545 |jq

// Result
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "extraData": [],
    "gasLimit": "0x0",
    "gasUsed": "0x0",
    "hash": "0x910ea44e2a83618c7cc98456678c9984d94977625e224939b24b3c904794b5ec",
    "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
    "number": "0x1",
    "parentHash":
↪"0x4765a126a9de8d876b87f01119208be507ec28495bef09c1e30a8ab240cf00f2",
    "sealer": "0x3",
    "sealerList":[

↪"0471101bcf033cd9e0cbd6eef76c144e6eff90a7a0b1847b5976f8ba32b2516c0528338060a4599fc5e3bafee188bc
↪",

↪"2b08375e6f876241b2a1d495cd560bd8e43265f57dc9ed07254616ea88e371dfa6d40d9a702eadfd5e025180f9d966
↪",

↪"cf93054cf524f51c9fe4e9a76a50218aaa7a2ca6e58f6f5634f9c2884d2e972486c7fe1d244d4b49c6148c1cb524bc
↪",

↪"ed1c85b815164b31e895d3f4fc0b6e3f0a0622561ec58a10cc8f3757a73621292d88072bf853ac52f0a9a9bbb10a54
↪"
    ],
```

```
    "stateRoot":
↪"0xfb7ca5a7a271c8ffb51bc689b78d0aeded23497c9c22e67dff8b1c7b4ec88a2a",
    "timestamp": "0x1687e801d99",
    "transactions": [
      "0x022dcb1ad2d940ce7b2131750f7458eb8ace879d129ee5b650b84467cb2184d7"
    ],
    "transactionsRoot":
↪"0x07506c27626365c4f0db788619a96df1e6f8f62c583f158192700e08c10fec6a"
  }
}
```

11.13 getBlockByNumber

return block information inquired by block number

11.13.1 Parameter

- `groupID`: `unsigned int` - group ID
- `blockNumber`: `string` - block number (hexadecimal string starts with 0x)
- `includeTransactions`: `bool` - include transactions ("true" shows transaction details; "false" shows the hash of transaction only)

11.13.2 Return value

please check getBlockByHash

- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getBlockByNumber","params":[1,"0x0
↪",true],"id":1}' http://127.0.0.1:8545 |jq
```

See the result in getBlockByHash

11.14 getBlockHashByNumber

return block hash inquired by block number

11.14.1 Parameters

- `groupID`: `unsigned int` - group ID
- `blockNumber`: `string` - block number (hexadecimal string starts with 0x)

11.14.2 Return value

- `blockHash`: `string` - hash of block
- Example

```
// 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"
}
```

11.15 getTransactionByHash

return transaction information inquired by transaction hash

11.15.1 Parameters

- `groupID`: `unsigned int` - group ID
- `transactionHash`: `string` - transaction hash

11.15.2 Return value

- `object`: - transaction information, fields are:
  - `blockHash`: `string` - include block hash of this transaction
  - `blockNumber`: `string` - include block number of this transaction
  - `from`: `string` - address of sender
  - `gas`: `string` - gas provided by sender
  - `gasPrice`: `string` - price of gas from sender
  - `hash`: `string` - transaction hash
  - `input`: `string` - transaction input
  - `nonce`: `string` - nonce value of transaction
  - `to`: `string` - address of receiver, return null for transactions of creating contract
  - `transactionIndex`: `string` - transaction sequence number
  - `value`: `string` - transfer value
- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByHash","params":[1,
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f"],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "blockHash":
↪"0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa",
        "blockNumber": "0x1",
```

```
        "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
        "gas": "0x9184e729fff",
        "gasPrice": "0x174876e7ff",
        "hash": "0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f
↪",
        "input":
↪"0x48f85bce0000000000000000000000000000000000000000000000000000000000000001bf5bd8a9e7ba8b936ea70
↪",
        "nonce": "0x65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f
↪",
        "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
        "transactionIndex": "0x0",
        "value": "0x0"
    }
}
```

11.16 getTransactionByBlockHashAndIndex

return transaction information inquired by blockhash and transaction sequence number

11.16.1 Parameters

- `groupID`: `unsigned int` - group ID
- `blockHash`: `string` - block hash
- `transactionIndex`: `string` - transaction sequence number

11.16.2 Return value

please see getTransactionByHash

- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByBlockHashAndIndex",
↪"params":[1,"0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa",
↪"0x0"],"id":1}' http://127.0.0.1:8545 |jq
```

see result in getTransactionByHash

11.17 getTransactionByBlockNumberAndIndex

return transaction information inquired by block number and transaction sequence number

11.17.1 Parameters

- `groupID`: `unsigned int` - group ID
- `blockNumber`: `string` - block number (hexadecimal string starts with 0x)
- `transactionIndex`: `string` - transaction sequence number

11.17.2 Return value

please see getTransactionByHash

- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionByBlockNumberAndIndex
↪","params":[1,"0x1","0x0"],"id":1}' http://127.0.0.1:8545 |jq
}
```

see result in getTransactionByHash

11.18 getTransactionReceipt

return transaction receipt inquired by transaction hash

11.18.1 Parameters

- `groupID`: `unsigned int` - group ID
- `transactionHash`: `string` - transaction hash

11.18.2 Return value

- `object`: - transaction information, fields are:
  - `blockHash`: `string` - include block hash of this transaction
  - `blockNumber`: `string` - include block hash of this transaction
  - `contractAddress`: `string` - contract address, for creating contract, return "0x0000000000000000000000000000000000000000"
  - `from`: `string` - address of sender
  - `gasUsed`: `string` - gas used by transaction
  - `logs`: `array` - log created by transaction
  - `logsBloom`: `string` - bloom filter value of log
  - `status`: `string` - status value of transaction
  - `to`: `string` - address of receiver, for creating contract, return null
  - `transactionHash`: `string` - transaction hash
  - `transactionIndex`: `string` - transaction sequence number
- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getTransactionReceipt","params":[1,
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f"],"id":1}'␣
↪http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
        "blockHash":
↪"0x10bfdc1e97901ed22cc18a126d3ebb8125717c2438f61d84602f997959c631fa",
```

```
        "blockNumber": "0x1",
        "contractAddress": "0x0000000000000000000000000000000000000000",
        "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
        "gasUsed": "0x64d8",
        "logs": [ ],
        "logsBloom":
↪"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪",
        "status": "0x0",
        "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
        "transactionHash":
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f",
        "transactionIndex": "0x0"
    }
}
```

11.19 getPendingTransactions

return pending transaction information

11.19.1 Parameter

- `groupID`: `unsigned int` - group ID

11.19.2 Return value

- `object`: - pending transaction information, fields are:
  - `blockHash`: `string` - include block hash of this transaction
  - `blockNumber`: `string` - include block number of this transaction
  - `contractAddress`: `string` - contract address. If creating contract, return "0x0000000000000000000000000000000000000000"
  - `from`: `string` - address of sender
  - `gas`: `string` - gas provided by sender
  - `gasPrice`: `string` - price of gas from sender
  - `hash`: `string` - transaction hash
  - `input`: `string` - transaction input
  - `nonce`: `string` - nonce value of transaction
  - `to`: `string` - address of receiver, for transactions of creating contract return null
  - `value`: `string` - transfer value
- Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"getPendingTransactions","params
↪":[1],"id":1}' http://127.0.0.1:8545 |jq

// Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": {
```

```
        [
            {
                "from": "0x6bc952a2e4db9c0c86a368d83e9df0c6ab481102",
                "gas": "0x9184e729fff",
                "gasPrice": "0x174876e7ff",
                "hash":
↪"0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f",
                "input":
↪"0x48f85bce000000000000000000000000000000000000000000000000000000000000001bf5bd8a9e7ba8b936ea70
↪",
                "nonce":
↪"0x65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f",
                "to": "0xd6f1a71052366dbae2f7ab2d5d5845e77965cf0d",
                "value": "0x0"
            }
        ]
    }
}
```

11.20 getPendingTxSize

return number of pending transactions

11.20.1 Parameter

- `groupID`: `unsigned int` - group ID

11.20.2 Return value

- `string`: - number of pending transactions
- Example

```
// 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"
}
```

11.21 getCode

return contract data inquired by contract address

11.21.1 Parameter

- `groupID`: `unsigned int` - group ID
- `address`: `string` - contract address

11.21.2 Return value

- `string`: - contract data
- Example

```
// 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
↪"
}
```

11.22 getTotalTransactionCount

return current transaction total and block number

11.22.1 Parameter

- `groupID`: `unsigned int` - group ID

11.22.2 Return value

- `object`: - current transaction total and block number. fields are:
  - `txSum`: `string` - transaction total
  - `blockNumber`: `string` - block number
- Example

```
// 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"
    }
}
```

11.23 getSystemConfigByKey

return value inquired by key value

11.23.1 Parameters

- `groupID`: `unsigned int` - group ID
- `key`: `string` - support tx_count_limit and tx_gas_limit

11.23.2 Return value

- `string` - value
- Example

```
// 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"
}
```

11.24 call

execute a request that can be returned with result instantly regardless of consensus

11.24.1 Parameters

- `groupID`: `unsigned int` - group ID
- `object`: - request information, fields are:
  - `from`: `string` - address of sender
  - `to`: `string` - address of receiver
  - `value`: `string` - (optional) transfer value
  - `data`: `string` - (optional) code parameter. You can read the coding convention in Ethereum Contract ABI

11.24.2 Return value

- `string` - executed result
- Example

```
// 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": "0x1",
        "output": "0x"
    }
}
```

11.25 sendRawTransaction

execute a transaction of signing, need consensus

11.25.1 Parameters

- `groupID`: `unsigned int` - group ID
- `rlp`: `string` - transaction data of signing

11.25.2 Return value

- `string` - hash of transaction
- Example

```
// RC1 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f8ef9f65f0d06e39dc3c08e32ac10a5070858962bc6c0f5760baca823f2d5582d03f85174876e7ff8609184e729fff
↪"],"id":1}' http://127.0.0.1:8545 |jq

// RC1 Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x7536cf1286b5ce6c110cd4fea5c891467884240c9af366d678eb4191e1c31c6f"
}

// RC2 Request
curl -X POST --data '{"jsonrpc":"2.0","method":"sendRawTransaction","params":[1,
↪"f8d3a003922ee720bb7445e3a914d8ab8f507d1a647296d563100e49548d83fd98865c8411e1a3008411e1a3008201
↪"],"id":1}' http://127.0.0.1:8545 |jq

// RC2 Result
{
    "id": 1,
    "jsonrpc": "2.0",
    "result": "0x0accad4228274b0d78939f48149767883a6e99c95941baa950156e926f1c96ba"
}

// FISCO BCOS supports OSCCA algorithm, request of OSCCA is exemplified here
// 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
```

11.26 Error codes

11.26.1 RPC error code

When error occurs in a RPC call, the returned response object should contain error result field, which includes following member parameters:

- code: to show error type in integer
- message: to briefly describe the error in string
- data: contains the basic type and structured type of additional information of the error; optional parameter

There are 2 types of error code for error objects: JSON-RPC standard error code and FISCO BCOS RPC error code.

JSON-RPC standard error code

Standard error codes and their definitions:

FISCO BCOS RPC error code

FISCO BCOS RPC error codes and their definitions:

11.27 Transaction receipt status list

11.27.1 Precompiled Service API error code

CHAPTER 12

Experimental features

This chapter introduces some experimental features of FISCO BCOS for user to experience in advance, which will be released officially once they are stabilized.

12.1 Transaction analysis

The transaction in FISCO BCOS is the request data toward blockchain system to deploy contract, call contract API, maintain contract lifecycle and manage assets, value exchange, etc. There will be transaction receipt after it is confirmed: Transaction receipt and Transaction are all stored in block to record the information generated during transaction execution, like result code, log, consumed gas amount, etc. User can use transaction hash to retrieve transaction receipt and know whether the transaction is accomplished.

Transaction receipt contains three key fields: input (currently fisco bcos compiled by dev branch includes this field), output, logs.

Transaction analysis can help user to analyze the 3 fields as json data and java object.

Important: code - fisco bcos

branch：https://github.com/FISCO-BCOS/FISCO-BCOS/tree/dev

- web3sdk

  branch：https://github.com/FISCO-BCOS/web3sdk/tree/dev

  maven version：2.0.34-SNAPSHOT

12.1.1 Import jar packet

Analysis tool class is in web3sdk. First, add the following configurations in build.gradle config file and import web3sdk jar packet.

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

```
}
compile group: "org.fisco-bcos", name: "web3sdk", version: "2.0.34-SNAPSHOT"
```

12.1.2 API description

Code packet route `org.fisco.bcos.web3j.tx.txdecode`, use `TransactionDecoderFactory` factory class to build transaction decoder `TransactionDecoder`. There are 2 ways:

1. `TransactionDecoder buildTransactionDecoder(String abi, String bin);`

   abi：contract ABI

   bin：contract bin, can import empty string "" when not available

2. `TransactionDecoder buildTransactionDecoder(String contractName);`

   contractName：contract name, create `solidity` directory under root directory of the application, put the contract under `solidity` directory and get transaction decoder through contract name.

Transaction decoder `TransactionDecoder` API list:

1. `String decodeInputReturnJson(String input)`

   Analyze input, seal the result as json character string in json format

   ```
   {"data":[{"name":"","type":"","data":} ... ],"function":"","methodID":""}
   ```

   function : function signature string methodID : function selector

2. `List<ResultEntity> decodeInputReturnObject(String input)`

   Analyze input, return java List object, ResultEntity structure

   ```
   public class ResultEntity {
      private String name;  // field name, analyze output and return with empty␣
   ↪string
      private String type;  // field type
      private Object data;  // field value
    }
   ```

3. `String decodeOutputReturnJson(String input, String output)`

   Analyze output, seal the result as json string, the same format with `decodeInputReturnJson`

4. `List<ResultEntity> decodeOutputReturnObject(String input, String output)`

   Analyze output, return java List object

5. `String decodeEventReturnJson(List<Log> logList)`

   Analyze event list, seal the result as json string with json format

   ```
   {"event1 signature":[[{"name":"_u","type":"","data":}...]...],"event2 signature
   ↪":[[{"name":"_u","type":"","data":}...]...]...}
   ```

6. `Map<String, List<List<ResultEntity>>> decodeEventReturnObject(List<Log> logList)`

   Analyze event list, return java Map object, key is event signature string, List<List> is all the event parameter information in the transaction.

`TransactionDecoder` provides methods of returning json string and java object respectively to input, output and event logs. Json string helps client end to easily deal data. Java object helps to easily deal data for server.

12.1.3 Example

Here is an example of `TxDecodeSample` contract to describe the use of API:

```
pragma solidity ^0.4.24;
contract TxDecodeSample
{
    event Event1(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes _bs);
    event Event2(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes _bs);


    function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string␣
↪_s,bytes _bs) public constant returns (uint256,int256,bool,address,bytes32,
↪string,bytes)
    {
      Event1(_u, _i, _b, _addr, _bs32, _s, _bs);
      return (_u, _i, _b, _addr, _bs32, _s, _bs);
    }

    function do_event(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32,␣
↪string _s,bytes _bs) public
    {
      Event1(_u, _i, _b, _addr, _bs32, _s, _bs);
      Event2(_u, _i, _b, _addr, _bs32, _s, _bs);
    }
}
```

Use `buildTransactionDecoder` to build the transaction decoder of `TxDecodeSample` contract.

```
// 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 = TransactionDecoder.
↪buildTransactionDecoder(abi, bin);
```

Analyze input

Call function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs) API, input parameter [111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji 章鱼小丸子ljjkl;adjsfkljlkjl sadfljkjkljkl]

```
// function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _
↪s,bytes _bs)
String input =␣
↪"0x406d373b0000000000000000000000000000000000000000000000000001b207ffffffffffffffffffff
↪
String jsonResult = txDecodeSampleDecoder.decodeInputReturnJson(input);
List<ResultEntity> listResult = txDecodeSampleDecoder.
↪decodeInputReturnObject(input);
System.out.println("json => \n" + jsonResult);
System.out.println("list => \n" + listResult);
```

Output:

```
json =>
{
  "data": [
    {
      "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": "章鱼小丸子ljjkl;adjsfkljlkjl"
    },
    {
      "name": "_bs",
      "type": "bytes",
      "data": "sadfljkjkljkl"
    }
  ],
  "function": "echo(uint256,int256,bool,address,bytes32,string,bytes)",
  "methodID": "0x406d373b"
}
```

```
list =>
[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=章鱼小丸子ljjkl;adjsfkljlkjl], ResultEntity [name=_bs,␣
↪type=bytes, data=sadfljkjkljkl]]
```

Analyze output

Call `function echo(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs)` API, input parameter `[ 111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji 章鱼小丸子ljjkl;adjsfkljlkjl sadfljkjkljkl ]`, echo API returns what is input.

```
//  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 =␣
↪"0x406d373b000000000000000000000000000000000000000000000000000000000001b207ffffffffffffffffffff
↪

String output =
↪"000000000000000000000000000000000000000000000000000000000001b207ffffffffffffffffffffffffffffff
↪";

String jsonResult = txDecodeSampleDecoder.decodeOutputReturnJson(input, output);
List<ResultEntity> listResult = txDecodeSampleDecoder.
↪decodeOutputReturnObject(input, output);
System.out.println("json => \n" + jsonResult);
System.out.println("list => \n" + listResult);
```

Result:

```
json =>
[
  {
    "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": "章鱼小丸子ljjkl;adjsfkljlkjl"
  },
  {
    "name": "",
    "type": "bytes",
    "data": "sadfljkjkljkl"
  }
]

list =>
[ResultEntity [name=, type=uint256, data=111111], ResultEntity [name=, type=int256,
↪ data=-1111111], ResultEntity [name=, type=bool, data=false], ResultEntity␣
↪[name=, type=address, data=0x692a70d2e424a56d2c6c27aa97d1a86395877b3a],␣
↪ResultEntity [name=, type=bytes32, data=abcdefghiabcdefghiabcdefghiabhji],␣
↪ResultEntity [name=, type=string, data=章鱼小丸子ljjkl;adjsfkljlkjl], ResultEntity␣
↪[name=, type=bytes, data=sadfljkjkljkl]]
```

Analyze event logs

Call `function do_event(uint256 _u,int256 _i,bool _b,address _addr,bytes32 _bs32, string _s,bytes _bs)` API, input parameter `[ 111111 -1111111 false 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a abcdefghiabcdefghiabcdefghiabhji 章鱼小丸子ljjkl;adjsfkljlkjl sadfljkjkljkl ]`, analyze transaction logs

```
// transactionReceipt is the transaction receipt of calling do_event API
String jsonResult = txDecodeSampleDecoder.decodeEventReturnJson(transactionReceipt.
↪getLogs());
String mapResult = txDecodeSampleDecoder.decodeEventReturnJson(transactionReceipt.
↪getLogs());

System.out.println("json => \n" + jsonResult);
System.out.println("map => \n" + mapResult);
```

Result:

```
jsonResult =>
{
  "Event1(uint256,int256,bool,address,bytes32,string,bytes)": [
    [
      {
        "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": "章鱼小丸子ljjkl;adjsfkljlkjl"
      },
      {
        "name": "_bs",
        "type": "bytes",
        "data": "sadfljkjkljkl"
      }
    ]
  ],
  "Event2(uint256,int256,bool,address,bytes32,string,bytes)": [
    [
      {
        "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": "章鱼小丸子ljjkl;adjsfkljlkjl"
      },
      {
        "name": "_bs",
        "type": "bytes",
        "data": "sadfljkjkljkl"
      }
    ]
  ]
}

map =>
{"Event1(uint256,int256,bool,address,bytes32,string,bytes)":[[{"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":"章
鱼小丸子ljjkl;adjsfkljlkjl"},{"name":"_bs","type":"bytes","data":"sadfljkjkljkl"}]],
↪"Event2(uint256,int256,bool,address,bytes32,string,bytes)":[[{"name":"_u","type":
↪"uint256","data":111111},{"name":"_i","type":"int256","data":-1111111},{"name":"_
↪b","type":"bool","data":false},{"name":"_addr","type":"address","data":
```

CHAPTER 13

FAQ

13.1 Version

Q: What changes have been made to FISCO BCOS version 2.0 compares to previous versions? A: Please refer to here.

Q: How do developers interact with the FISCO BCOS platform? A: FISCO BCOS provides multiple ways for developers to interact with the platform. Please refer as follows:

- FISCO BCOS 2.0 version provides JSON-RPC interface. For the detail, please refer to here.
- FISCO BCOS 2.0 version provides Web3SDK to help developers quickly implement applications. For the detail, please refer to here.
- FISCO BCOS version 2.0 provides a console to help users quickly understand how to use FISCO BCOS. For the detail, please refer to here.

Q: How to build FISCO BCOS 2.0 version? A: FISCO BCOS supports multiple building methods. The common methods are:

- build_chain.sh: It is suitable for developer experience and testing FISCO BCOS alliance chain. For the detail, please refer to here。
- FISCO-Generator: For deploying and maintaining the FISCO BCOS Alliance Chain with enterprise users. For the detail, please refer to here。

Q: What is the difference on the smart contract between FISCO BCOS 2.0 version and the previous version? And how is the compatibility? A: FISCO BCOS version 2.0 supports the latest Solidity contract and precompile contract. For the detail, please [refer to here] (./manual/smart_contract.md).

Q: What is the difference between the national cryptographic version and the normal version? A: The national cryptography version FISCO BCOS replaces the cryptographic algorithms of underlying modules such as transaction signature verification, p2p network connection, node connection, and data disk encryption with the national cryptography algorithm. Meanwhile, in compiling version and certificate, disk encryption, and solidity compiling java, there are some difference on web3sdk using national cryptography version and normal version, please refer to [refer to here] (./manual/guomi_crypto.md).

Q: Does it support to upgrade to 2.0 version from 1.3 or 1.5? A: It does not.

13.2 Console

Q: Is the console instruction case sensitive? A: It is case-sensitive. The command will match exactly, and `tab` can be used to complete your command.

Q:When adding to the Sealer list or the Observer list, it will report error as nodeID is not in network, why? A: The nodes that adds to the Sealer list and the Observer list must be a member of the nodeID list that connects to the peer.

Q: To delete node will report error as nodeID is not in group peers, why? A: The node to be delete must be the peer of the group displayed in getGroupPeers

Q: Can the RemoveNodes (non-group nodes) synchronize group data? A: RemoveNodes does not participate in the consensus, synchronization, and block generation within the group. RemoveNodes can add the exit node as Sealer/Observer through the command of console `addSealer/addObserver`.

Q: If the node belongs to a different group, can it support querying information of multiple groups? A: Yes, when you enter the console, you can input the groupID you want to view: ./start [groupID]

13.3 FISCO BCOS using

Q: Where to get Ver 2.0 certificates? A: Please read Certificates Decsription

Q: What fields are contained in the transaction structure of Ver 2.0? A: Please read here Q: What are the system configuration, group configuration, and node configuration? A: System configuration refers to some configuration items that affect the ledger function and require the consensus of the ledger node in the node configuration. Group configuration refers to the configuration of the group which the node belongs to. Each group of nodes has an independent configuration. Node configuration refers to all configurable items.

Q: Can the group configuration be changed? A: Whether the configuration item could be changed can be measured by:

- The node that is first time to launch and has generated a genesis block can not be modified. This type of configuration is placed in the group.x.genesis file, where x is the group number, and it is unique in the entire chain.
- To implement consistence in ledger by sending the transaction modification configuration item.
- After the configuration file is modified, the node can be restarted to takes effect. This type of configuration is placed in the `group.x.ini` file. After the group configuration is changed, the restart can be changed locally, the changeable item becomes the local configuration. The `group.*.ini` file under nodeX/conf is changed and restarted to takes effect. The involved configuration items are [tx_pool].limit (transaction pool capacity) and [consensus].ttl (node forwarding number).

Q: Which configurations can the group configuration user change? A: The group can be modified and configured into consensus changeable configuration and manual changeable configuration.

- consensus changeable configuration: all nodes in the group are the same, and takes effect after consensus. [consensus].max_trans_num,[consensus].node.X,[tx].gas_limit.
- manual changeable configuration: it is in the `group.x.ini` file and restarted to take effect after modification. It only affects node. The configuration item has [tx_pool].limit.

Q: How to change and inquire the consensus changeable configuration? A: Consensus changeable configuration can be changed through console. It can be inquired through console and RPC interface. For detail, please [refer to here] (./design/rpc.md).

- [consensus].max_trans_num,[tx].gas_limit is changed by using the interface setSystemConfigByKey, and the corresponding configuration items are tx_count_limit, tx_gas_limit. See setSystemConfigByKey -h for details.
- [consensus].node.X's change refers to node management. The console interface refer to addSealer, addObserver, removeNode. For detail, please refer to Node Management.

Q: What is the difference between Observer node and Sealer node in group? A: Observer node can synchronize the group data, but cannot participate in consensus. Consensus nodes have the Observer permission and participate in consensus.

Q: How to incorporate contract into CNS management? A: When contract is deployed, to call the CNS contract interface, and to compile the information of contract name, version, and address information into the CNS list.

Q: How to query the contract CNS list? A: To query CNS list through the command of web3sdk console, and the query command is queried according to the contract name.

13.4 Web3SDK

Q: What does Web3SDK require to Java version? A: It requires JDK8 version or above

The OpenJDK of yum repository of CentOS lacks JCE (Java Cryptography Extension), which causes Web3SDK to fail to connect to blockchain node. When using the CentOS operation system, it is recommended to download it from the OpenJDK website. [Installation Guide] (https://openjdk.java.net/install/index.html)

Q: After the Web3SDK configuration is completed, what is the reason for the failed transaction? A: The ip, port, group number in applicationContext.xml are incorrectly filled or the node files of ca.crt, node.crt, and node.key files are missing.

13.5 Enterprise deployment tool

Q: There is pip cannot be found appears when using enterprise deployment tools. A: The enterprise deployment tool relies on python pip. To install it with the following command:

```
$ python -m pip install
```

Q: When using enterprise deployment tools, the following information appears:

```
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
```

A: The python configparser module is missing from the system. Please follow the command below to install:

```
  $ pip install configparser
```

CHAPTER 14

Community

FISCO BCOS, officially launched in December 2017, is the first China-developed open source consortium blockchain platform. It was collaboratively built by the FISCO open source working group, which was formed by Beyondsoft, Huawei, Shenzhen Securities Communications, Digital China, Forms Syntron, Tencent, WeBank, YIBI Technology, Yuexiu Financial Holdings (Fintech) and more.

14.1 FISCO BCOS resources

- Github homepage
- Technical documents
- Inlightful articles
- Code contribution
- Feedbacks
- Application cases

14.2 Join FISCO BCOS community

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