Test Documentation

Release FISCO BCOS

Test

Nov 01, 2018

Contents

FISCO BCOS平台是金融区块链合作联盟（深圳）（以下简称：金链盟）开源工作组以金融业务实践为参考样本，在BCOS开源平台基础上进行模块升级与功能重塑，深度定制的安全可控、适用于金融行业且完全开源的区块链底层平台。 金链盟开源工作组获得金链盟成员机构的广泛认可，并由专注于区块链底层技术研发的成员机构及开发者牵头开展工作。其中首批成员包括以下单位（排名不分先后）：博彦科技、华为、深证通、神州数码、四方精创、腾讯、微众银行、越秀金科。 FISCO BCOS平台基于现有的BCOS开源项目进行开发，聚焦于金融行业的分布式商业需求，从业务适当性、性能、安全、政策、技术可行性、运维与治理、成本等多个维度进行综合考虑，打造金融版本的区块链解决方案。 为了让大家更好的了解FISCO BCOS区块链开源平台的使用方法。本文档按照Step By Step的步骤详细介绍了FISCO BCOS区块链的构建、安装、启动，智能合约部署、调用等初阶用法，还包括多节点组网、系统合约等高阶内容的介绍。 本文档不介绍FISCO BCOS区块链设计理念及思路，详情请参看白皮书。

CHAPTER 1

快速指引

Important:

快速搭链工具

- FISCO BCOS 物料包
- FISCO BCOS docker

手工搭链

- 手工搭链

国密版FISCO-BCOS

- 国密版FISCO BCOS
- 国密版web3sdk

web3sdk

- SDK使用指南
- SDK应用开发指南

CHAPTER 2

手工搭链

2.1 环境要求

配置	最低配置	推荐配置
CPU	1.5GHz	2.4GHz
内存	2GB	4GB
核心	2核	4核
带宽	1Mb	5Mb
操作系统		CentOS （7.2 64位）或Ubuntu（16.04 64位）
JAVA		Java(TM) 1.8 && JDK 1.8

2.2 程序部署

2.2.1 获取代码

新建一个目录，例如/mydata

```
sudo mkdir -p /mydata
sudo chmod 777 /mydata
cd /mydata
```

clone 源码

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

2.2.2 编译安装FISCO-BCOS

切换到FISCO-BCOS目录下，执行FISCO-BCOS编译安装脚本

```
cd FISCO-BCOS
sh build.sh
```

检查是否安装成功

```
fisco-bcos --version
#成功: FISCO-BCOS version x.x.x
```

2.3 基础配置

2.3.1 配置根证书

生成链的根证书

```
cd /mydata/FISCO-BCOS/tools/scripts/

#bash generate_chain_cert.sh -o 根证书生成的目录
bash generate_chain_cert.sh -o /mydata
```

2.3.2 配置机构证书

生成机构（agency）证书，假设生成机构test_agency

```
cd /mydata/FISCO-BCOS/tools/scripts/

#bash generate_agency_cert.sh -c 生成机构证书所需的根证书所在目录 -o 机构证书生成目录 -n 机
构名
bash generate_agency_cert.sh -c /mydata -o /mydata -n test_agency
```

2.4 创世节点

2.4.1 生成创世节点

生成节点的目录、配置文件、启动脚本、身份文件、证书文件。并自动部署系统合约。

```
cd /mydata/FISCO-BCOS/tools/scripts/

#sh generate_genesis_node -o 节点文件夹生成位置 -n 节点名 -l 节点监听的IP -r 节点的RPC端口␣
↪-p 节点的P2P端口 -c 节点的Channel Port端口 -d 机构证书存放目录 -a 机构证书名
#创世节点
bash generate_genesis_node.sh  -o /mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -
↪c 8891 -d /mydata/test_agency/ -a test_agency
```

若成功，得到创世节点信息

```
Genesis node generate success!
----------------------------------------------------------------
Name:                          node0
Node dir:                /mydata/node0
Agency:                        test_agency
CA hash:                A809F269BEE93DA4
Node␣
↪ID:                d23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9
RPC address:               127.0.0.1:8545
P2P address:               127.0.0.1:30303
Channel address:        127.0.0.1:8891
SystemProxy address:       0x919868496524eedc26dbb81915fa1547a20f8998
God address:               0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
```

(continues on next page)

(continued from previous page)

```
State:                          Stop
------------------------------------------------------------------
```

记录下创世节点的RPC address，之后会用到

```
RPC address:  127.0.0.1:8545
```

2.4.2 启动创世节点

直接到创世节点文件目录下启动

```
cd /mydata/node0
bash start.sh
#关闭用 sh stop.sh
```

2.4.3 创世节点加入联盟

让创世节点成为参与共识的第一个成员

```
cd /mydata/FISCO-BCOS/tools/scripts/
```

设置需要操作的链的RPC端口（此时链上只有一个创世节点），输入y回车确认。

```
#bash set_proxy_address.sh -o 节点的RPC address
bash set_proxy_address.sh -o 127.0.0.1:8545
```

将创世节点注册入联盟中，参与共识

```
#bash register_node.sh -d 要注册节点的文件目录
bash register_node.sh -d /mydata/node0/
```

2.4.4 验证创世节点启动

验证进程

查看创世节点进程

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

若看到创世节点进程，表示创世节点启动成功

```
app   67232      1   2 14:51 ?        00:00:03 fisco-bcos --genesis /mydata/node0/
↪genesis.json --config /mydata/node0/config.json
```

验证可共识

查看日志，查看打包信息

```
tail -f /mydata/node0/log/info* |grep +++
```

等待一段时间，可看到周期性的出现如下日志，表示节点间在周期性的进行共识，节点运行正确

```
INFO|2018-08-10 14:53:33:083|++++++++++++++++++++++++++ Generating seal␣
↪on9272a2f7d8fba7e68b1927912f97797447cf94d92fa78222bb8a2892ee814ba8#31tx:0,
↪maxtx:0,tq.num=0time:1533884013083
INFO|2018-08-10 14:53:34:094|++++++++++++++++++++++++++ Generating seal␣
↪onf007c276c3f2b136fe84e40a84a54eda1887a47192ec7e2a4feff6407293665d#31tx:0,
↪maxtx:0,tq.num=0time:1533884014094
```

2.5 增加节点

2.5.1 准备

查看创世节点信息

```
cd /mydata/FISCO-BCOS/tools/scripts/

#bash node_info.sh -d 要查看信息的节点目录 -o 生成节点信息文件
bash node_info.sh -d /mydata/node0/ -o node0.info
```

得到创世节点的关键信息。创世节点的关键信息已经在节点信息文件node0.info中记录下来

```
Node␣
↪ID:                 d23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9
P2P address:               127.0.0.1:30303
SystemProxy address:       0x919868496524eedc26dbb81915fa1547a20f8998
God address:               0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
```

2.5.2 生成节点

生成节点的目录、配置文件、启动脚本、身份文件、证书文件。

注意：端口不要和其它节点重复

```
#bash generate_node -o 节点文件生成位置 -n 节点名 -l 节点监听的IP -r 节点的RPC端口 -p 节点
的P2P端口 -c 节点的Channel Port端口 -e 链上现有节点的P2P端口列表，用","隔开（如指向创世节点和
自己 127.0.0.1:30303,127.0.0.1:30304） -d 机构证书存放目录 -a 机构证书名 -f 创世节点的信息
文件
bash generate_node.sh -o /mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c 8892 -e␣
↪127.0.0.1:30303,127.0.0.1:30304 -d /mydata/test_agency -a test_agency -f node0.
↪info
```

若成功，得到节点信息

```
Node generate success!
----------------------------------------------------------------------
Name:                        node1
Node dir:                 /mydata/node1
Agency:                        test_agency
CA hash:                A809F269BEE93DA5
Node␣
↪ID:                 9042d0cbe004c8441ccc92370ab4c36ac197c7c015aa186b3ae7ff3cd4e649a20e82273b9cb5
RPC address:               127.0.0.1:8546
P2P address:               127.0.0.1:30304
Channel address:        127.0.0.1:8892
SystemProxy address:       0x919868496524eedc26dbb81915fa1547a20f8998
God address:               0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
State:                       Stop
----------------------------------------------------------------------
```

2.5.3 启动节点

直接到节点文件目录下启动

```
cd /mydata/node1
bash start.sh
#关闭用 sh stop.sh
```

2.5.4 节点加入联盟

让节点成为参与共识的成员

```
cd /mydata/FISCO-BCOS/tools/scripts/
```

设置需要操作的链的RPC端口（若之前已设置，则无需重复设置）

```
#bash set_proxy_address.sh -o 节点的RPC address
bash set_proxy_address.sh -o 127.0.0.1:8545
```

将节点注册入联盟中，参与共识

```
#bash register_node.sh -d 要注册节点的文件目录
bash register_node.sh -d /mydata/node1/
```

2.5.5 验证节点启动

验证进程

查看节点进程

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

若看到节点进程，表示创世节点启动成功

```
app   70450      1   0 14:58 ?        00:00:26 fisco-bcos --genesis /mydata/node1/
↪genesis.json --config /mydata/node1/config.json
```

验证已连接

查看日志

```
cat /mydata/node1/log/* | grep "topics Send to"
```

看到发送topic的日志，表示节点已经连接了相应的另一个节点

```
DEBUG|2018-08-10 15:42:05:621|topics Send to:1 nodes
DEBUG|2018-08-10 15:42:06:621|topics Send␣
↪tod23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9d15842110b3db6239
↪0.0.1:30303
```

验证可共识

查看日志，查看打包信息

```
tail -f /mydata/node1/log/* |grep +++
```

可看到周期性的出现如下日志，表示节点间在周期性的进行共识，节点运行正确

```
INFO|2018-08-10 15:48:52:108|++++++++++++++++++++++++++ Generating seal␣
↪on57b9e818999467bff75f58b08b3ca79e7475ebfefbb4caea6d628de9f4456a1d#32tx:0,
↪maxtx:1000,tq.num=0time:1533887332108
INFO|2018-08-10 15:48:54:119|++++++++++++++++++++++++++ Generating seal␣
↪on273870caa50741a4841c3b54b7130ab66f08227601b01272f62d31e48d38e956#32tx:0,
↪maxtx:1000,tq.num=0time:1533887334119
```

CHAPTER 3

使用指南

3.1 区块链节点

节点是区块链上的执行单元。多个节点彼此连接，构成一个P2P网络，承载了区块链上的通信，计算和存储。节点入网后（加入联盟），成为区块链上的一个共识单位。多个节点参与共识，确保了区块链上交易的一致。

3.1.1 节点文件目录、分类

FISCO-BCOS的节点包含了下列必要的文件。此处不列举节点在运行时生成的文件。

```
node0
|-- genesis.json    #创世块文件（创世块信息，god账号，创世节点）
|-- config.json     #节点总配置文件（IP，端口，共识算法）
|-- log.conf        #节点日志配置文件（日志格式，优先级）
|-- start.sh        #节点启动脚本
|-- stop.sh         #节点停止脚本
|-- data
|   |-- bootstrapnodes.json     #节点启动时需访问的peers列表
|   |-- ca.crt                  #链根证书私钥
|   |-- agency.crt              #机构证书私钥
|   |-- node.crt                #节点证书私钥
|   |-- node.ca
|   |-- node.csr
|   |-- node.json
|   |-- node.key
|   |-- node.nodeid
|   |-- node.param
|   |-- node.private
|   |-- node.pubkey
|   `-- node.serial             #节点证书序列号
|-- keystore
|-- fisco-bcos.log    #节点启动日志
`-- log   #节点运行日志目录
```

其中，按类型归类：

- 配置文件：genesis.json、config.json、log.conf、bootstrapnodes.json

- 证书文件：ca.crt、agency.crt、node.crt、node.csr、node.key、node.private、node.pubkey
- 功能文件：node.json
- 信息文件：node.nodeid、node.serial、node.ca
- 日志文件：fisco-bcos.log、log文件夹
- 操作脚本：start.sh、stop.sh

3.1.2 配置文件

genesis.json

genesis.json中配置创世块的信息，是节点启动必备的信息。

```
{
     "nonce": "0x0",
     "difficulty": "0x0",
     "mixhash": "0x0",
     "coinbase": "0x0",
     "timestamp": "0x0",
     "parentHash": "0x0",
     "extraData": "0x0",
     "gasLimit": "0x13880000000000",
     "god":"0xf78451eb46e20bc5336e279c52bda3a3e92c09b6",
     "alloc": {},
     "initMinerNodes":[
↪"d23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9d15842110b3db6239d
↪"]
}
```

字段说明

配置项	说明
timestamp	创世块时间戳(毫秒)
god	内置链管理员账号地址（填入<u>2.2 生成god账号</u> 小节中生成的地址）
alloc	内置合约数据
initMinerNodes	创世块节点NodeId（填入<u>2.3 生成节点身份NodeId</u>小节中生成NodeId）

config.json

节点的总配置文件，配置节点的IP，端口，共识算法，data目录等等

```
{
        "sealEngine": "PBFT",
        "systemproxyaddress":"0x919868496524eedc26dbb81915fa1547a20f8998",
        "listenip":"127.0.0.1",
        "cryptomod":"0",
        "rpcport": "8545",
        "p2pport": "30303",
        "channelPort": "8891",
        "wallet":"./data/keys.info",
        "keystoredir":"./data/keystore/",
        "datadir":"./data/",
        "vm":"interpreter",
        "networkid":"12345",
        "logverbosity":"4",
        "coverlog":"OFF",
        "eventlog":"ON",
```

(continues on next page)

(continued from previous page)

```
        "statlog":"OFF",
        "logconf":"./log.conf"
}
```

字段说明

注意：rpcport 和 channelPort 仅限于被机构内的监控、运维、sdk等模块访问，切勿对外网开放

配置项	说明
sealEngine	共识算法（可选PBFT、RAFT、SinglePoint）
systemproxyaddress	系统路由合约地址（生成方法可参看部署系统合约）
listenip	节点监听IP
cryptomod	加密模式默认为0
rpcport	RPC监听端口）（若在同台机器上部署多个节点时，端口不能重复）
p2pport	P2P网络监听端口（若在同台机器上部署多个节点时，端口不能重复）
channelPort	链上链下监听端口（若在同台机器上部署多个节点时，端口不能重复）
wallet	钱包文件路径
keystoredir	账号文件目录路径
datadir	节点数据目录路径
vm	vm引擎 （默认 interpreter ）
networkid	网络ID
logverbosity	日志级别（级别越高日志越详细，>8 TRACE，4<=x<8 DEBUG日志，<4 INFO日志）
coverlog	覆盖率插件开关（ON或OFF）
eventlog	合约日志开关（ON或OFF）
statlog	统计日志开关（ON或OFF）
logconf	日志配置文件路径（日志配置文件可参看日志配置文件说明）
dfsNode	分布式文件服务节点ID，与节点身份NodeID一致 （可选功能配置参数）
dfsGroup	分布式文件服务组ID （10 - 32个字符） （可选功能配置参数）
dfsStorage	指定分布式文件系统所使用文件存储目录（可选功能配置参数）

log.conf

log.conf中配置节点日志生成的格式、路径和优先级。

```
* GLOBAL:
    ENABLED                 =   true
    TO_FILE                 =   true
    TO_STANDARD_OUTPUT      =   false
    FORMAT                  =   "%level|%datetime{%Y-%M-%d %H:%m:%s:%g}|%msg"
    FILENAME                =   "./log/log_%datetime{%Y%M%d%H}.log"
    MILLISECONDS_WIDTH      =   3
    PERFORMANCE_TRACKING    =   false
    MAX_LOG_FILE_SIZE       =   209715200 ## 200MB - Comment starts with two␣
↪hashes (##)
    LOG_FLUSH_THRESHOLD     =   100  ## Flush after every 100 logs

* TRACE:
    ENABLED                 =   true
    FILENAME                =   "./log/trace_log_%datetime{%Y%M%d%H}.log"

* DEBUG:
    ENABLED                 =   true
    FILENAME                =   "./log/debug_log_%datetime{%Y%M%d%H}.log"
```

(continues on next page)

(continued from previous page)

```
* FATAL:
    ENABLED                     =   true
    FILENAME                    =   "./log/fatal_log_%datetime{%Y%M%d%H}.log"

* ERROR:
    ENABLED                     =   true
    FILENAME                    =   "./log/error_log_%datetime{%Y%M%d%H}.log"

* WARNING:
     ENABLED                     =   true
     FILENAME                    =   "./log/warn_log_%datetime{%Y%M%d%H}.log"

* INFO:
    ENABLED                     =   true
    FILENAME                    =   "./log/info_log_%datetime{%Y%M%d%H}.log"

* VERBOSE:
    ENABLED                     =   true
    FILENAME                    =   "./log/verbose_log_%datetime{%Y%M%d%H}.log"
```

字段说明

配置项	说明
FORMAT	日志格式，典型如%level
FILENAME	例如/mydata/nodedata-1/log/log_%datetime{%Y%M%d%H}.log
MAX_LOG_FILE_SIZE	最大日志文件大小
LOG_FLUSH_THRESHOLD	超过多少条日志即可落盘

bootstrapnodes.json

配置节点启动时主动去连接的节点。在连接成功后，节点会自动同步彼此的peers，进而连接更多的节点。

```
{"nodes":[
        {"host":"127.0.0.1","p2pport":"30303"},
        {"host":"127.0.0.1","p2pport":"30304"}
]}
```

3.1.3 证书文件

参考证书说明

3.1.4 功能文件

node.json

节点注册入网需要提供的文件，在生成节点证书时自动生成。

```
 {
     "id":
↪"d23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9d15842110b3db6239d
↪",
     "name":"node0",
     "agency":"test_agency",
```

(continues on next page)

(continued from previous page)

```
    "caHash":"A809F269BEE93DA4"
}
```

3.1.5 信息文件

node.nodeid

保存节点nodeid信息，在生成节点证书时自动生成。

```
d23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9d15842110b3db6239da2a
```

node.serial

保存节点证书的序列号，在生成节点证书时自动生成。

```
A809F269BEE93DA4
```

3.1.6 日志文件

fisco-bcos.log

节点的启动日志。若节点无法启动时，查看此日志。

log文件夹

节点运行时打印出的日志，按照日志优先级，存放于此。

```
log/
|-- debug_log_2018081319.log
|-- error_log_2018081319.log
|-- fatal_log_2018081319.log
|-- info_log_2018081319.log
|-- log_2018081319.log          #全部的日志
|-- stat_log_2018081319.log     #统计日志
|-- trace_log_2018081319.log
|-- verbose_log_2018081319.log
`-- warn_log_2018081319.log
```

3.1.7 操作脚本

start.sh

必须cd到脚本所在目录下才能正确运行此脚本。执行后，节点在后台被启动。

```
cd /mydata/node0
sh start.sh
```

stop.sh

与start.sh配合，必须cd到脚本所在目录下才能正确使用此脚本停掉节点。

```
cd /mydata/node0
sh stop.sh
```

3.2 证书说明

FISCO-BCOS网络采用面向CA的准入机制，保障信息保密性、认证性、完整性、不可抵赖性。

一条链拥有一个链证书及对应的链私钥，链私钥由链管理员拥有。并对每个参与该链的机构签发机构证书，机构证书私钥由机构管理员持有，并对机构下属节点签发节点证书。节点证书是节点身份的凭证，并使用该证书与其他节点间建立SSL连接进行加密通讯。sdk证书是sdk与节点通信的凭证，机构生成sdk证书，允许sdk与节点进行通信。

因此，需要生成链证书、机构证书、节点证书，sdk证书。文件后缀介绍如下：

后缀	说明
.key	私钥
.srl	文件存储序列号
.csr	证书请求文件
.crt	Certificate 证书
.pubkey	公钥
.private	私钥.key编码得到
.p12	PKCS#12格式来储存密钥
.keystore	用做web3sdk的SSL证书
.crl	证书吊销列表

3.2.1 角色定义：

- FISCO-BCOS准入机制中，不同角色拥有不同的密钥与证书文件，共同保障信息保密性、认证性、完整性、不可抵赖性。FISCO-BCOS中，共有四种角色，分别是链管理员，机构，节点和SDK。

链管理员：

- 链管理员管理链的私钥，可以向机构颁发机构证书。

```
ca.crt 链证书
ca.key 链私钥
ca.srl 链序列号
```

机构：

- 机构拥有机构私钥，可以颁发节点证书和sdk证书。

```
ca.crt 链证书
agency.crt 机构证书
agency.csr 机构证书请求文件
agency.key 机构私钥
agency.srl 机构序列号
ca-agency.crt 机构和链的证书
```

节点：

- 节点管理节点私钥。

```
ca.crt 链证书
node.ca节点证书相关信息，应用于系统合约
node.crt 节点证书
node.csr节点证书请求文件
node.key节点私钥
node.private 节点私钥编码得到
node.pubkey 节点公钥
node.serial 节点序列号
```

SDK：

- sdk管理sdk私钥。

```
ca.crt 链证书
clent.keysotre 用做web3sdk的SSL证书
keystore.p12 用pkcs#12存储的密钥
sdk.crt sdk证书
sdk.csr sdk证书请求文件
sdk.key sdk私钥
sdk.private sdk私钥编码得到
sdk.pubkey sdk公钥
```

3.2.2 证书生成流程：

FISCO-BCOS的证书生成流程如下，详见证书操作部分

生成链证书：

- 运行生成链证书脚本generate_chain_cert.sh
- 请求链私钥ca.key，用ca.key生成链证书ca.crt

生成机构证书：

- 运行机构证书生成脚本generate_agency_cert.sh
- 首先请求机构私钥agency.key
- 用机构私钥agency.key 得到公钥证书原始文件agency.csr
- 用链证书ca.crt，链私钥ca.key，公钥证书原始文件agency.csr生成机构证书agency.crt。机构证书生成不需要机构私钥

生成节点证书：

- 运行节点证书生成脚本generate_node_cert.sh
- 首先请求secp256k1模块得到node参数，用这些参数得到node.key私钥
- 然后用node.key得到公钥node.pubkey
- 用私钥node.key得到node公钥证书原始文件node.csr
- 用公钥证书原始文件node.csr，机构私钥agency.key，机构证书agency.crt，node公钥node.pubkey得到node证书node.crt。节点证书生成不需要节点私钥

- 将node.key DER编码写入node.private
- 提取node.key里的数据得到nodeID
- 提取node.crt里的数据得到node.serial

生成sdk证书：

- 运行sdk证书生成脚本generate_sdk_cert.sh
- 首先请求secp256k1模块得到sdk参数，用这些参数得到sdk.key私钥
- 然后用sdk.key得到公钥sdk.pubkey
- 用私钥sdk.key得到sdk公钥证书原始文件sdk.csr
- 用公钥证书原始文件sdk.csr，机构私钥agency.key，机构证书agency.crt，sdk公钥sdk.pubkey得到sdk证书sdk.crt
- 将sdk.key DER编码写入sdk.private
- 最后把ca.crt和agency.crt写入ca-agency.crt

数字证书就是区块链网络中标志通讯各方身份信息的一串数字，提供了一种在网络上验证通信实体身份的方式，数字证书不是数字身份证，而是身份认证机构盖在数字身份证上的一个章或印（或者说加在数字身份证上的一个签名）。FISCO-BCOS网络采用证书机制，对节点的准入进行管理，从而保证整个网络有效，可靠，安全地进行通信。

3.3 合约入门

本文作为基础，供读者初步理解智能合约的编写、部署和调用。若需要开发应用，请使用高级的智能合约开发框架web3sdk。

3.3.1 准备

查看需要部署智能合约的链，对应的节点RPC端口

```
cd /mydata/FISCO-BCOS/tools/scripts/

#sh node_info.sh -d 要查看信息的节点目录
sh node_info.sh -d /mydata/node0/
```

得到端口

```
RPC address:                127.0.0.1:8545
```

设置需要操作的链的RPC端口，输入y回车确认。

```
cd /mydata/FISCO-BCOS/tools/script/

#sh set_proxy_address.sh -o 节点的RPC address
sh set_proxy_address.sh -o 127.0.0.1:8545
```

此后，所有的操作都会发送到127.0.0.1:8545端口上，即node0上。

3.3.2 编写合约

```
cd /mydata/FISCO-BCOS/tools/contract/
vim HelloWorld.sol
```

HelloWorld.sol的实现如下

```
pragma solidity ^0.4.2;
contract HelloWorld{
    string name;
    function HelloWorld(){
       name="Hi,Welcome!";
    }
    function get()constant returns(string){
        return name;
    }
    function set(string n){
        name=n;
    }
}
```

3.3.3 编译、部署合约

直接使用deploy.js，自动编译和部署合约。

```
babel-node deploy.js HelloWorld #注意后面HelloWorld后面没有".sol"
```

输出，可看到合约地址，部署成功。

```
deploy.js  ......................Start.....................
Soc File :HelloWorld
HelloWorldcomplie success!
send transaction success:␣
↪0xa8c1aeed8e85cc0308341081925d3dab80da394f6b22c76dc0e855c8735da481
HelloWorldcontract address 0xa807685dd3cf6374ee56963d3d95065f6f056372
HelloWorld deploy success!
```

3.3.4 调用合约

编写合约调用程序

用nodejs实现，具体实现方法请直接看demoHelloWorld.js源码。

```
vim demoHelloWorld.js
```

调用合约

执行合约调用程序

```
babel-node demoHelloWorld.js
```

可看到合约调用成功

```
{ HttpProvider: 'http://127.0.0.1:8545',
  Ouputpath: './output/',
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
HelloWorldcontract address:0xa807685dd3cf6374ee56963d3d95065f6f056372
HelloWorld contract get function call first :Hi,Welcome!
send transaction success:␣
↪0x6463e0ea9db6c4aff1e3fc14d9bdb86b29306def73e6d951913a522347526435
```

(continues on next page)

(continued from previous page)

```
HelloWorld contract set function call , (transaction hash␣
↪: 0x6463e0ea9db6c4aff1e3fc14d9bdb86b29306def73e6d951913a522347526435)
HelloWorld contract get function call again :HelloWorld!
```

3.4 系统合约

系统合约是FISCO BCOS区块链内置的智能合约。一条链对应唯一的系统合约。系统合约实现了对链的控制和管理。如节点注册，机构准入等等。

系统合约是一组合约的集合，包括：

- 系统代理合约
- 节点管理合约
- 注销证书合约
- 权限管理合约
- 全网配置合约

节点相关

系统合约在创世节点生成时，脚本已自动将其部署到链上，并设置了节点config.json文件中的systemproxyaddress来指向系统合约的地址。重新更新systemproxyaddress的节点需重新启动才能生效。

操作相关

配置：在操作前，需用脚本`set_proxy_address.sh`配置需要操作的链。

操作目录：`FISCO-BCOS/tools/systemcontract`

3.4.1 系统代理合约

系统代理合约是系统合约的统一入口。

它提供了路由名称到合约地址的映射关系。

源码路径：systemcontract/SystemProxy.sol

接口说明

接口名	输入	输出	备注
获取路由信息getRoute	路由名称	路由地址、缓存标志位、生效块号	无
注册路由信息setRoute	路由名称、路由地址、缓存标志位	无	若该路由名称已存在，则覆盖

web3调用示例如下（可参看systemcontract/deploy.js）：

```
console.log("register NodeAction.....");
func = "setRoute(string,address,bool)";
params = ["NodeAction", NodeAction.address, false];
receipt = await web3sync.sendRawTransaction(config.account, config.privKey,␣
↪SystemProxy.address, func, params);
```

工具使用方法

查看所有系统合约信息：

```
babel-node tool.js SystemProxy
```

示例输出如下：

```
{ HttpProvider: 'http://127.0.0.1:8701',
  Ouputpath: './output/',
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :SystemProxy
Func :undefined
SystemProxy address 0x210a7d467c3c43307f11eda35f387be456334fed
-----------------SystemProxy route----------------------
0 )TransactionFilterChain=>0xea8425697a093606309eb85e4447d6f333cff2fe,false,395
1 )ConfigAction=>0x09e4f1b4fa1713339f5aa17b40fa6f9920c7b278,false,396
2 )NodeAction=>0xcc46c245e6cca918d43bf939bbb10a8c0988548f,false,397
3 )CAAction=>0x8ab1175c6e7edb40dd0ed2a52ceaa94afb135a64,false,398
4 )ContractAbiMgr=>0x707024221d2433067b768c4be3a005c5ece8df40,false,399
5 )ConsensusControlMgr=>0x007f2c2751bbcd6c9a630945a87a3bc2af38788c,false,400
6 )FileInfoManager=>0xe0caa8103ea05b5ce585c05d8112051a0b213acf,false,401
7 )FileServerManager=>0xe585cc5b8ca7fb174a0560bf79eea7398efaf014,false,402
```

输出中即是当前系统路由表的所有路由信息。

3.4.2 节点管理合约

节点管理合约主要功能是维护网络中节点列表。 网络中节点加入或退出都需要与节点管理合约进行交互。

源码路径：systemcontract/NodeAction.sol

接口说明

接口名	输入	输出	备注
节点入网 registerNode	节点ID, 节点名称 节点机构、证书序列号	布尔结果	若该节点ID已存在，则忽略
节点出网 cancelNode	节点ID	布尔结果	若路由名称不存在，则忽略

web3调用示例如下（可参看systemcontract/tool.js）：

```
var instance=getAction("NodeAction");
var func = "registerNode(string,string,string,string)";
var params = [node.id,node.name,node.agency,node.caHash];
var receipt = web3sync.sendRawTransaction(config.account, config.privKey, instance.
↪address, func, params);
```

工具使用方法

请参看 注册记账节点、退出记账节点。

3.4.3 注销证书合约

注销证书合约主要功能是维护注销证书信息列表。

源码路径：systemcontract/CAAction.sol

（1）接口说明

接口名	输入	输出	备注
登记 add	证书序列号、证书公钥、节点名称	布尔结果	若该证书信息不存在，则新建
移除 remove	证书序列号	布尔结果	若该证书不存在，则忽略
查询证书信息 get	证书序列号	证书序列号、公钥 公钥，节点名称、块号	无

web3调用示例如下（可参看systemcontract/tool.js）：

```
var instance=getAction("CAAction");
var func = "add(string,string,string)";
var params = [ca.serial,ca.pubkey,ca.name];
var receipt = web3sync.sendRawTransaction(config.account, config.privKey, instance.
↪address, func, params);ig.account, config.privKey, instance.address, func,␣
↪params);
```

工具使用方法

查看注销证书列表

```
babel-node tool.js CAAction all
```

编写证书配置

编写文件：ca.json。将序列号填入hash字段。配置status，0表示不可用，1表示可用。其它字段默认即可。如下，让node2的证书可用。即status置1。详细介绍，请参考机构证书准入说明。

```
{
    "hash":"8A4B2CDE94348D22",
    "status":1,
    "pubkey":"xxxx",
    "orgname":"xx银行",
    "notbefore":20170223,
    "notafter":20180223,
    "whitelist":"192.168.1.1;192.168.1.2;192.168.1.3",
    "blacklist":"192.168.1.11;192.168.1.12;192.168.1.13"
}
```

登记注销证书

ca.json 中status置为1

```
babel-node tool.js CAAction update ca.json
```

移除注销证书

ca.json 中status置为0

```
babel-node tool.js CAAction update ca.json
```

3.4.4 权限管理合约

权限管理合约是对区块链权限模型的实现。

一个外部账户只属于一个角色，一个角色拥有一个权限项列表。

一个权限项由合约地址加上合约接口来唯一标识。

源码路径：systemcontract/AuthorityFilter.sol 交易权限Filter

systemcontract/Group.sol

接口说明

合约	接口名	输入	输出	备注
角色	设置用户权限组权限项 setPermission	合约地址、合约接口、权限标记	无	无
	获取权限标记 getPermission	合约地址、合约接口	权限标记	无
交易权限Filter	设置用户所属角色 setUserGroup	用户外部账户、用户所属角色合约	无	无
	交易权限检查 process	**用户外部账户、交易发起账户、**合约地址、合约接口、交易数据	无	无

web3调用示例如下（可参看systemcontract/deploy.js）：

```
var GroupReicpt= await web3sync.rawDeploy(config.account, config.privKey, "Group");
var Group=web3.eth.contract(getAbi("Group")).at(GroupReicpt.contractAddress);
#......省略若干行......
abi = getAbi0("Group");
params  = ["Group","Group","",abi,GroupReicpt.contractAddress];
receipt = await web3sync.sendRawTransaction(config.account, config.privKey,␣
↪ContractAbiMgrReicpt.contractAddre
ss, func, params);
```

工具使用方法

检查用户外部账户权限

```
babel-node tool.js AuthorityFilter 用户外部账户、合约地址、合约接口
```

自主定制

继承TransactionFilterBase实现新的交易Filter合约。并通过addFilter接口将新Filter注册入TransactionFilterChain即可。

3.4.5 全网配置合约

全网配置合约维护了区块链中部分全网运行配置信息。

目标是为了通过交易的全网共识来达成全网配置的一致更新。

源码路径：systemcontract/ConfigAction.sol

全网配置项说明

配置项	说明	默认值	推荐值
maxBlockHeadGas	块最大GAS	2,000,000,000	2,000,000,000
intervalBlockTime	块间隔(ms)	1000	1000
maxBlockTranscations	块最大交易数	1000	1000
maxNonceCheckBlock	交易nonce检查最大块范围	1000	1000
maxBlockLimit	blockLimit超过当前块号的偏移最大值	1000	1000
maxTranscationGas	交易的最大gas	30,000,000	30,000,000
CAVerify	CA验证开关	false	false

接口说明

接口名	输入	输出	备注
设置配置项 set	配置项、配置值	无	若配置表中已存在，则覆盖
查询配置值 get	配置项	配置值、块号	无

web3调用示例如下（可参看systemcontract/tool.js）：

```
var func = "set(string,string)";
var params = [key,value];
var receipt = web3sync.sendRawTransaction(config.account, config.privKey, instance.
↪address, func, params);
console.log("config :"+key+","+value);
```

使用方法

查询配置项

```
babel-node tool.js ConfigAction get 配置项
```

设置配置项

```
babel-node tool.js ConfigAction set 配置项 配置值
```

3.5 基本操作

本文提供了FISCO-BCOS的基本操作。包括证书操作，节点操作，链操作及信息查看。

3.5.1 证书操作

生成链证书 （CA）

脚本：generate_chain_cert.sh

说明：在指定位置生成链的根证书CA

参数：

必要参数

flag	参数	说明
-o	指定链证书生成的位置	

可选参数

flag	参数	说明
-m	无	证书生成时，手动输入参数，不采用默认参数
-g	无	生成国密链证书，这里必须设置
-d	指定国密证书生成脚本存放路径	默认是FISCO-BCOS/tools/cert/GM，采用默认值，不需要输入参数内容
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_chain_cert.sh -h
```

生成证书

```
bash generate_chain_cert.sh -o /mydata
```

在目录下生成

```
#tree /mydata
/mydata
|-- ca.crt
`-- ca.key
```

生成机构证书

脚本：generate_agency_cert.sh

说明：用链的根证书，在指定的位置生成机构的证书

参数：

必要参数

flag	参数	说明
-c	链证书（CA证书）所在目录	脚本用CA证书生成机构证书
-o	指定的机构证书文件夹生成位置	
-n	机构名	机构证书文件夹用此名字命名

可选参数

flag	参数	说明
-m	无	证书生成时，手动输入参数，不采用默认参数
-g	无	生成国密机构证书，这里必须设置
-d	指定国密证书生成脚本存放路径	默认是FISCO-BCOS/tools/cert/GM，采用默认值，不需输入参数内容
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_agency_cert.sh -h
```

生成证书

```
bash generate_agency_cert.sh -c /mydata -o /mydata -n test_agency
```

在目录下生成

```
#tree test_agency
test_agency/
|-- agency.crt
|-- agency.csr
|-- agency.key
|-- ca.crt
`-- cert.cnf
```

生成节点证书

脚本：generate_node_cert.sh **说明**：用机构证书，在节点的data目录下生成节点证书。在使用本脚本前，请先用generate_node_basic.sh生成节点目录。

参数：

必要参数

flag	参数
-a	机构名
-d	机构证书文件夹
-n	节点名
-o	指定的节点证书输出到的data文件夹

可选参数

flag	参数	说明
-r	国密证书生成脚本存放路径	默认是FISCO-BCOS/tools/cert/GM，采用默认值
-s	国密sdk名称	产生国密版节点证书时，必须产生sdk证书
-g	无	生成国密节点证书，这里必须设置
-m	无	证书生成时，手动输入参数，不采用默认参数
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_node_cert.sh -h
```

生成证书

```
bash generate_node_cert.sh -a test_agency -d /mydata/test_agency -n node0 -o /
↪mydata/node0/data
```

节点目录下生成证书，身份，功能等文件，用*标出

```
node0/
|-- config.json
|-- data
|   |-- agency.crt *
|   |-- bootstrapnodes.json
|   |-- ca.crt *
|   |-- node.ca *
|   |-- node.crt *
|   |-- node.csr *
|   |-- node.json *
|   |-- node.key *
|   |-- node.nodeid *
|   |-- node.param *
|   |-- node.private *
|   |-- node.pubkey *
|   `-- node.serial *
|-- keystore
|-- log
|-- log.conf
|-- start.sh
`-- stop.sh
```

生成SDK证书

脚本：generate_sdk_cert.sh

说明：在指定机构的证书目录下生成机构对应的SDK证书。此脚本相对独立，只有需要使用web3sdk，才需使用此脚本。

参数：

必要参数

flag	参数	说明
-d	机构证书文件夹	用机构名生成机构的SDK证书

可选参数

flag	参数	说明
-m	无	证书生成时，手动输入参数，不采用默认参数
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_sdk_cert.sh -h
```

生成证书

```
bash generate_sdk_cert.sh -d /mydata/test_agency
```

输入一系列密码后，在机构证书目录下生成sdk文件夹

```
#tree test_agency/
test_agency/
|-- agency.crt
|-- agency.csr
|-- agency.key
|-- agency.srl
|-- ca-agency.crt
|-- ca.crt
|-- cert.cnf
`-- sdk
    |-- ca.crt
    |-- client.keystore
    |-- keystore.p12
    |-- sdk.crt
    |-- sdk.csr
    |-- sdk.key
    |-- sdk.param
    |-- sdk.private
    `-- sdk.pubkey
```

3.5.2 节点操作

生成创世节点

脚本：generate_genesis_node.sh

说明：生成创世节点，并自动内置系统合约。其中会调用generate_node_basic.sh，generate_node_cert.sh、generate_genesis.sh和deploy_system_contract.sh生成创世节点的目录、文件、证书和系统合约。生成的创世节点是关闭状态。创世节点生成后，需启动，并注册，将其加入联盟中参与共识。

参数：

必要参数

flag	参数	说明
-o	指定创世节点的文件夹生成位置	
-n	创世节点名	
-l	节点监听端口对应的IP	
-r	节点RPC端口号	根据情况自定，端口不冲突即可
-p	节点P2P端口号	根据情况自定，端口不冲突即可
-c	节点channel端口号	根据情况自定，端口不冲突即可
-a	节点所属机构名	
-d	节点所属机构证书文件夹	

可选参数

flag	参数	说明
-s	god账号地址	手动指定god账号地址
-g	无	表明生成国密版FISCO-BCOS创世节点
-m	无	证书生成，手动输入参数，不采用默认参数
-h	无	查看帮助

操作：查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_genesis_node.sh -h
```

生成创世节点

```
bash generate_genesis_node.sh -o /mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -c␣
↪8891 -d /mydata/test_agency -a test_agency
```

生成创世节点所有的文件，并自动部署上了系统合约

```
#tree node0/
node0/
|-- config.json
|-- data
|   |-- 4bcbbeb4
|   |   |-- 12041
|   |   |   |-- extras
|   |   |   |   |-- 000003.log
|   |   |   |   |-- CURRENT
|   |   |   |   |-- LOCK
|   |   |   |   |-- LOG
|   |   |   |   `-- MANIFEST-000002
|   |   |   `-- state
|   |   |       |-- 000003.log
|   |   |       |-- CURRENT
|   |   |       |-- LOCK
|   |   |       |-- LOG
|   |   |       `-- MANIFEST-000002
|   |   `-- blocks
|   |       |-- 000003.log
```

(continues on next page)

(continued from previous page)

```
|   |       |-- CURRENT
|   |       |-- LOCK
|   |       |-- LOG
|   |       `-- MANIFEST-000002
|   |-- abiname
|   |   |-- 000003.log
|   |   |-- CURRENT
|   |   |-- LOCK
|   |   |-- LOG
|   |   `-- MANIFEST-000002
|   |-- agency.crt
|   |-- bootstrapnodes.json
|   |-- ca.crt
|   |-- event.log
|   |-- geth.ipc
|   |-- IPC_MappedFile
|   |-- keys.info
|   |-- keys.info.salt
|   |-- node.ca
|   |-- node.crt
|   |-- node.csr
|   |-- node.json
|   |-- node.key
|   |-- node.nodeid
|   |-- node.param
|   |-- node.private
|   |-- node.pubkey
|   |-- node.serial
|   |-- pbftMsgBackup
|   |   |-- 000003.log
|   |   |-- CURRENT
|   |   |-- LOCK
|   |   |-- LOG
|   |   `-- MANIFEST-000002
|   |-- RPC_MappedFile
|   `-- UTXO
|       |-- db
|       |   |-- 000003.log
|       |   |-- CURRENT
|       |   |-- LOCK
|       |   |-- LOG
|       |   `-- MANIFEST-000002
|       |-- extra
|       |   |-- 000003.log
|       |   |-- CURRENT
|       |   |-- LOCK
|       |   |-- LOG
|       |   `-- MANIFEST-000002
|       `-- vault
|           |-- 000003.log
|           |-- CURRENT
|           |-- LOCK
|           |-- LOG
|           `-- MANIFEST-000002
|-- fisco-bcos.log
|-- genesis.json
|-- keystore
|-- log
|   |-- debug_log_2018081521.log
|   |-- error_log_2018081521.log
|   |-- fatal_log_2018081521.log
```

(continues on next page)

(continued from previous page)

```
|   |-- info_log_2018081521.log
|   |-- log_2018081521.log
|   |-- stat_log_2018081521.log
|   |-- trace_log_2018081521.log
|   |-- verbose_log_2018081521.log
|   `-- warn_log_2018081521.log
|-- log.conf
|-- myeasylog.log
|-- start.sh
`-- stop.sh
```

生成普通节点

脚本：generate_node.sh

说明：用创世节点的nodeid、系统代理合约地址、创世节点的p2p地址，生成普通节点。其中会调用generate_node_basic.sh、generate_node_cert.sh和generate_genesis.sh，生成节点的目录、文件和证书。生成的节点是关闭状态。节点启动后，会自动连接创世节点，同步系统合约。

参数：

必要参数

flag	参数	说明
-o	指定节点的文件夹生成位置	
-n	节点名	
-l	节点监听端口对应的IP	
-r	节点RPC端口号	根据情况自定，端口不冲突即可
-p	节点P2P端口号	根据情况自定，端口不冲突
-c	节点channel端口号	根据情况自定，端口不冲突即
-e	链上节点P2P接口URL的列表	节点启动时，主动连接链上节点的P2P接口同步数据。 此列表用”,”分割，如：127.0.0.1:30303,127.0.0.1:30304
-d	节点所属机构证书文件夹	
-a	节点所属机构名	

可选参数

flag	参数	说明
-i	创世节点nodeid	用于生成与创世节点相同的创世块文件。创世节点的nodeid可用node_info.sh去查
-s	god账号地址	用于生成与创世节点相同的创世块文件。创世节点god账号地址可用node_info.sh去查
-x	系统代理合约地址	用于配置与链相同的系统代理合约地址。可用node_info.sh去查任意的节点获取。
-f	创世节点信息文件	从创世节点的信息文件（用node_info.sh脚本生成）读取创世节点nodeid， god账号地址，系统代理合约地址。替代-i -s -x命令
-g	无	表明生成国密版FISCO-BCOS普通节点
-m	无	证书生成时，手动输入参数，不采用默认参数
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_node.sh -h
```

生成节点

```
bash generate_node.sh -o /mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c 8892 -e␣
↪127.0.0.1:30303,127.0.0.1:30304 -d /mydata/test_agency -a test_agency -x␣
↪0x919868496524eedc26dbb81915fa1547a20f8998 -s␣
↪0xb862b65912e0857a49458346fcf578d199dba024 -i xxxxxx
```

或

```
bash generate_node.sh -o /mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c 8892 -e␣
↪127.0.0.1:30303,127.0.0.1:30304 -d /mydata/test_agency -a test_agency -f node0.
↪info
```

生成节点的全部文件

```
tree node1
node1
|-- config.json
|-- data
|   |-- agency.crt
|   |-- bootstrapnodes.json
|   |-- ca.crt
|   |-- node.ca
|   |-- node.crt
|   |-- node.csr
|   |-- node.json
|   |-- node.key
|   |-- node.nodeid
|   |-- node.param
|   |-- node.private
|   |-- node.pubkey
|   `-- node.serial
|-- genesis.json
|-- keystore
|-- log
|-- log.conf
|-- start.sh
`-- stop.sh
```

生成节点基本文件

脚本：generate_node_basic.sh

说明：在指定目录下，生成节点目录，并在目录下生成节点的配置文件和操作脚本。此时，节点还缺少证书文件，功能文件和信息文件，需要继续使用generate_node_cert.sh来生成。或直接用generate_node.sh直接生成节点的所有文件。

参数：

必要参数

flag	参数	说明
-o	指定节点的文件夹生成位置	
-n	节点名	
-l	节点监听端口对应的IP	
-r	节点RPC端口号	根据情况自定，端口不冲突即可
-p	节点P2P端口号	根据情况自定，端口不冲突即可
-c	节点channel端口号	根据情况自定，端口不冲突
-e	链上节点P2P接口URL的列表	节点启动时，主动连接链上节点的P2P接口同步数据。此列表用","分割，如：127.0.0.1:30303,127.0.0.1:30304

可选参数

flag	参数	说明
-x	系统代理合约地址	用于配置与链相同的系统代理合约地址。 可用node_info.sh去查任意的节点获取。
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_node_basic.sh -h
```

生成节点基本文件

```
bash generate_node_basic.sh -o /mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -c␣
↪8891 -e 127.0.0.1:30303,127.0.0.1:30304
```

在目录下生成

```
#tree node0/
node0/
|-- config.json
|-- data
|   `-- bootstrapnodes.json
|-- keystore
|-- log
|-- log.conf
|-- start.sh
`-- stop.sh
```

3.5.3 链操作

生成god账号

说明：在生产条件下，需要手动生成god账号，提供给相关脚本生成节点文件。

操作：

到指定目录下生成god账号

```
cd /mydata/FISCO-BCOS/tools/contract
node accountManager.js > godInfo.txt
cat godInfo.txt
```

得到生成的god账号信息，此信息请妥善保存，在对链操作的某些场景

```
privKey : 0xc8a92524ac634721a9eac94c9d8c09ea719f3a01e0ed1f576f673af6eb90aeea
pubKey :␣
↪0xb2795b4000981fb56f386a00e5064bd66b7754db6532bb17f9df1975ca884fc7b3b3291f9f3b20ee0278e610b8814
address : 0xb862b65912e0857a49458346fcf578d199dba024
```

其中god账号地址是需要作为参数提供给其它脚本的

```
address : 0xb862b65912e0857a49458346fcf578d199dba024
```

生成创世块文件

脚本：generate_genesis.sh

说明：用god账号地址，创世节点的nodeid，生成创世块文件，若不指定god账号地址，用默认的god账号地址。

参数：

必要参数

flag	参数	说明
-o	指定创世块文件生成位置	创世块文件genesis.json
-i 或 -d	创世节点的nodeid 或文件夹	二选一

可选参数

flag	参数	说明
-i	创世节点nodeid	
-d	创世节点文件夹	
-s	god账号地址	手动指定god账号地址
-g	无	表明生成国密god账号
-h	无	查看帮助

操作：查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts/
bash generate_genesis.sh -h
```

生成创世块文件

```
bash generate_genesis.sh -i␣
↪4af70363e2266e62aaca5870d660cc4ced35deae83b67f3dffebd0dcfa3b16d96d8fe726f9fea0def06a3bbde47261b
↪-o /mydata/node1 -r 0xb862b65912e0857a49458346fcf578d199dba024
```

生成的创世块文件genesis.json

```
{
    "nonce": "0x0",
    "difficulty": "0x0",
    "mixhash": "0x0",
    "coinbase": "0x0",
    "timestamp": "0x0",
```

(continues on next page)

(continued from previous page)

```
    "parentHash": "0x0",
    "extraData": "0x0",
    "gasLimit": "0x13880000000000",
    "god":"0xb862b65912e0857a49458346fcf578d199dba024",
    "alloc": {},
    "initMinerNodes":[
↪"4af70363e2266e62aaca5870d660cc4ced35deae83b67f3dffebd0dcfa3b16d96d8fe726f9fea0def06a3bbde47261
↪"]
}
```

配置待操作链的端口

脚本：config_rpc_address.sh

说明：对正在运行中的链进行操作（如：注册节点，部署合约，调用合约）时，需先将全局proxy变量指向待操作的链。具体的，是将全局proxy变量设置为链上某个节点的RPC端口，即设置/mydata/FISCO-BCOS/tools/web3sdk/config.js的proxy为相应的节点RPC的URL。设置一次即可永久生效，无需重复设置。

参数：

必要参数

flag	参数	说明
-o	某节点RPC的URL	全局proxy变量指向待操作的链RPC的URL

可选参数

flag	参数	说明
-w	web3lib文件夹地址	手动指定web3lib地址，默认在../web3lib
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts
bash set_proxy_address.sh -h
```

设置RPC的URL（节点的RPC的URL用node_info.sh获取）

```
bash set_proxy_address.sh -o 127.0.0.1:8545
```

yes回车确认后，写入全局的proxy变量中

```
proxy="http://127.0.0.1:8545"
```

部署系统合约

脚本：deploy_system_contract.sh

说明：用在生成创世节点时（generate_generate_node.sh），已经自动调用此脚本部署了系统合约。若需要重新手动部署系统合约，则调用此脚本。并同时将链上其它节点的config.json修改为与创世节点相同的systemproxyaddress，并重启节点使其生效。一般来说，不推荐重新部署系统合约。重新部署系统合约意味着对链管理的重置，需要链上所有机构都同意，且需要重启链上所有的节点。在使用此脚本前，需调set_proxy_address.sh设置全局proxy地址（若设置则无需重复设置）。

参数：

必要参数

flag	参数
-d	创世节点文件夹

可选参数

flag	参数	说明
-w	web3lib文件夹地址	手动指定web3lib地址，默认在../web3lib
-s	systemcontract文件夹地址	手动指定systemcontract地址，默认在../systemcontract
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts
bash set_proxy_address.sh -h
```

部署系统合约

```
bash deploy_systemcontract.sh -d /mydata/node0
```

得到系统代理合约地址

```
SystemProxy address: 0xbac830dee59a0f2a33beddcf53b329a4e1787ce2
```

将链上其它节点的systemproxyaddress也修改为相同的地址

```
sed -i '/systemproxyaddress/c  \\t\"systemproxyaddress\":\
↪"0xbac830dee59a0f2a33beddcf53b329a4e1787ce2\",'  /mydata/node1/config.json
sed -i '/systemproxyaddress/c  \\t\"systemproxyaddress\":\
↪"0xbac830dee59a0f2a33beddcf53b329a4e1787ce2\",'  /mydata/node2/config.json
```

注册节点

脚本：register_node.sh

说明：此脚本将某个指定的节点注册入网，让此节点参与到区块链的共识中。在链初始化初期，链上无任何一个节点被注册，此时由创世节点进行共识，其它节点都是观察者节点。当有一个节点被注册后，由注册的节点进行共识。此时，创世节点退化成一个普通的节点，也需被注册后才能参与共识。在使用此脚本前，需调set_proxy_address.sh设置全局proxy地址（若设置则无需重复设置）。

参数：

必要参数

flag	参数
-d	需要注册节点的文件夹

可选参数

flag	参数	说明
-w	web3lib文件夹地址	手动指定web3lib地址，默认在../web3lib
-s	systemcontract文件夹地址	手动指定systemcontract地址，默认在../systemcontract
-g	无	表明注册国密版FISCO-BCOS节点到系统合约
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts
bash register_node.sh -h
```

将指定节点目录，将节点注册入网

```
bash register_node.sh -d /mydata/node0
```

用node_all.sh脚本可看到节点入网情况

```
----------node 0---------
id=4af70363e2266e62aaca5870d660cc4ced35deae83b67f3dffebd0dcfa3b16d96d8fe726f9fea0def06a3bbde47261
name=node0
agency=test_agency
caHash=BE4790D7B2BA3D1A
Idx=0
blocknumber=59
```

3.5.4 状态查询

查看节点存活

说明：直接查看节点进程

操作：

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

可看到相应的节点进程，用启动目录区分不同的节点

```
app   57342      1 23 15:24 ?        00:00:02 fisco-bcos --genesis /mydata/node0/
↪genesis.json --config /mydata/node0/config.json
app   57385      1 37 15:24 ?        00:00:01 fisco-bcos --genesis /mydata/node1/
↪genesis.json --config /mydata/node1/config.json
```

查看节点基本信息

脚本：node_info.sh

说明：指定节点目录，输出节点的基本信息。

参数：

必要参数

flag	参数
-d	需要查看信息的节点文件夹

可选参数

flag	参数	说明
-o	节点信息文件名	指定生成的节点信息文件
-g	无	输出国密版的节点信息
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts
bash node_info.sh -h
```

指定节点目录，查看节点信息

```
bash node_info.sh -d /mydata/node1/ -o node1.info
```

得到节点关键信息

```
-------------------------------------------------------------------
Name:                          node1
Node dir:                  /mydata/node1/
Agency:                        test_agency
CA hash:                  BE4790D7B2BA3D1B
Node␣
↪ID:                  f56014d3ef617a8d461c6531a68b183594203b269e82ead83cc37f47cd67c3f45798b8a1c20ec
RPC address:                 127.0.0.1:8546
P2P address:                 127.0.0.1:30304
Channel address:          127.0.0.1:8892
SystemProxy address:         0xbac830dee59a0f2a33beddcf53b329a4e1787ce2
God address:         0xb862b65912e0857a49458346fcf578d199dba024
State:                        Running (pid: 69599)
-------------------------------------------------------------------
```

同时生成了节点信息文件node1.info

```
{
    "listenip" : "127.0.0.1",
    "rpcport" : 8546,
    "p2pport" : 30304,
    "channelPort" : 8892,
    "systemproxyaddress" : "0xbac830dee59a0f2a33beddcf53b329a4e1787ce2",
    "god" : "0xb862b65912e0857a49458346fcf578d199dba024",
    "id" :
↪"f56014d3ef617a8d461c6531a68b183594203b269e82ead83cc37f47cd67c3f45798b8a1c20edc5a0d93d68fcd253d
↪",
    "name" : "/mydata/node1/config.json",
    "agency" : "test_agency",
    "caHash" : "BE4790D7B2BA3D1B"
}
```

查看链上被注册的节点

脚本：node_all.sh

说明：输出链上所有被注册的节点，即链上参与共识的节点。在使用此脚本前，需调set_proxy_address.sh设置全局proxy地址（若设置则无需重复设置）。

参数：

可选参数

flag	参数	说明
-w	web3lib文件夹地址	手动指定web3lib地址，默认在../web3lib
-s	systemcontract文件夹地址	手动指定systemcontract地址，默认在../systemcontract
-h	无	查看帮助

操作：

查看用法

```
cd /mydata/FISCO-BCOS/tools/scripts
bash node_all.sh -h
```

直接调用脚本查看

```
bash node_all.sh
```

得到被注册的节点

```
NodeIdsLength= 2
----------node 0---------
id=4af70363e2266e62aaca5870d660cc4ced35deae83b67f3dffebd0dcfa3b16d96d8fe726f9fea0def06a3bbde47261
name=node0
agency=test_agency
caHash=BE4790D7B2BA3D1A
Idx=0
blocknumber=59
----------node 1---------
id=f56014d3ef617a8d461c6531a68b183594203b269e82ead83cc37f47cd67c3f45798b8a1c20edc5a0d93d68fcd253d
name=node1
agency=test_agency
caHash=BE4790D7B2BA3D1B
Idx=1
blocknumber=60
```

查看节点连接状态

说明：通过查看日志，过滤某个关键字，查看指定节点的链接情况

操作：

```
cat /mydata/node1/log/* | grep "topics Send to"
```

看到发送topic的日志，表示节点已经连接了相应的另一个节点

```
DEBUG|2018-08-10 15:42:05:621|topics Send to:1 nodes
DEBUG|2018-08-10 15:42:06:621|topics Send␣
↪tod23058c33577f850832e47994df495c674ba66273df2fcb1e6ee7d7e1dbd7be78be2f7b302c9d15842110b3db6239
↪0.0.1:30303
```

查看节点共识状态

说明：通过查看日志，过滤某个关键字，查看指定节点的共识状态

操作：

```
tail -f /mydata/node1/log/* |grep +++
```

可看到周期性的出现如下日志，表示节点间在周期性的进行共识，节点运行正确

```
INFO|2018-08-10 15:48:52:108|++++++++++++++++++++++++++ Generating seal␣
↪on57b9e818999467bff75f58b08b3ca79e7475ebfefbb4caea6d628de9f4456a1d#32tx:0,
↪maxtx:1000,tq.num=0time:1533887332108
INFO|2018-08-10 15:48:54:119|++++++++++++++++++++++++++ Generating seal␣
↪on273870caa50741a4841c3b54b7130ab66f08227601b01272f62d31e48d38e956#32tx:0,
↪maxtx:1000,tq.num=0time:1533887334119
```

查看节点启动日志

说明：节点启动日志在节点每次启动时刷新，若节点无法启动，可查看此日志。

操作：

```
cat /mydata/node0/fisco-bcos.log
```

查看节点运行日志

说明：节点在运行时，在所在目录下的log文件夹里实时的打印一系列的日志。根据日志等级的划分，可查看相应的日志输出。

操作：

查看目录下的日志

```
log
|-- debug_log_2018081521.log
|-- error_log_2018081521.log
|-- fatal_log_2018081521.log
|-- info_log_2018081521.log
|-- log_2018081521.log        #全局日志
|-- stat_log_2018081521.log
|-- trace_log_2018081521.log
|-- verbose_log_2018081521.log
`-- warn_log_2018081521.log
```

按日期和时刻查看日志

```
cat log_2018081521.log
```

查看实时刷出的日志

```
tail -f log_2018081521.log
```

3.6 控制台

控制台能以IPC的方式直接连接区块链节点进程。使用控制台，能直接查看到区块链上的信息。若需要更直观更全面的区块链数据展现，请使用FISCO-BCOS的区块链浏览器。

3.6.1 登陆控制台

连接节点的data目录下的geth.ipc文件。

```
ethconsole /mydata/node0/data/geth.ipc
```

登录成功，可看到successful字样。

```
Connecting to node at /mydata/nodedata-1/data/geth.ipc
Connection successful.
Current block number: 37
Entering interactive mode.
>
```

3.6.2 查看区块

如查看块高为2的区块

```
web3.eth.getBlock(2,console.log)
```

可得到相应的区块信息

```
> null { author: '0x0000000000000000000000000000000000000000',
  difficulty: { [String: '1'] s: 1, e: 0, c: [ 1 ] },
  extraData: '0xd78095312e302b2b302d524c696e75782f672b2b2f496e74',
  gasLimit: 2000000000,
  gasUsed: 33245,
  genIndex: '0x0',
  hash: '0xa3c2b1eda74f26c688e78bffcc71c8561e49dc70fbfbd71b85c3b79a2c16bc81',
  logsBloom:
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
  miner: '0x0000000000000000000000000000000000000000',
  number: 2,
  parentHash: '0x96d095047e68d4db8ae0c74638cf6365d5ee7fbc329d50a5a78a85189e1b105e',
  receiptsRoot: '0xb668c383b2ee76f9e2d0a8b0d9b301fb825896a9a597268ca3cbdd979c4a53da
↪',
  sha3Uncles: '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
  size: 71,
  stateRoot: '0x20993fb96c5a26eb9f158c194bcd4da845afde2c2095e359259001279de23ec2',
  timestamp: 1513087469641,
  totalDifficulty: { [String: '2'] s: 1, e: 0, c: [ 2 ] },
  transactions: [
↪'0x63749a62851b52f9263e3c9a791369c7380acc5a9b6ee55dabd9c1013634e355' ],
  transactionsRoot:
↪'0xc9c0b09f236fba3bad81f6fb92b9ca03b6ccb1aaf53ef600621c0a0fd0fd4bb1',
  uncles: [] }
```

3.6.3 查看交易

根据交易哈希查看交易。如，使用之前调用HelloWord的合约的交易哈希。

```
web3.eth.getTransaction(
↪'0x63749a62851b52f9263e3c9a791369c7380acc5a9b6ee55dabd9c1013634e355',console.log)
```

可得到相应的交易信息

```
> null { blockHash:
↪'0xa3c2b1eda74f26c688e78bffcc71c8561e49dc70fbfbd71b85c3b79a2c16bc81',
  blockNumber: 2,
  from: '0x04804c06677d2009e52ca96c825d38056292cab6',
  gas: 30000000,
  gasPrice: { [String: '0'] s: 1, e: 0, c: [ 0 ] },
  hash: '0x63749a62851b52f9263e3c9a791369c7380acc5a9b6ee55dabd9c1013634e355',
  input:
↪'0x4ed3885e0000000000000000000000000000000000000000000000000000000000000020000000000000000000000
↪',
  nonce: 67506452,
  to: '0x1d2047204130de907799adaea85c511c7ce85b6d',
  transactionIndex: 0,
  value: { [String: '0'] s: 1, e: 0, c: [ 0 ] } }
```

3.6.4 查看交易回执

根据交易哈希查看交易回执。如，使用之前调用HelloWord的合约的交易哈希。

```
web3.eth.getTransactionReceipt(
↪'0x63749a62851b52f9263e3c9a791369c7380acc5a9b6ee55dabd9c1013634e355',console.log)
```

可得到相应的交易回执

```
> null { blockHash:
↪'0xa3c2b1eda74f26c688e78bffcc71c8561e49dc70fbfbd71b85c3b79a2c16bc81',
  blockNumber: 2,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 33245,
  gasUsed: 33245,
  logs: [],
  transactionHash:
↪'0x63749a62851b52f9263e3c9a791369c7380acc5a9b6ee55dabd9c1013634e355',
  transactionIndex: 0 }
```

3.6.5 查看合约代码

根据交易合约地址查看合约。如，用之前部署的HelloWorld合约地址。

```
web3.eth.getCode('0x1d2047204130de907799adaea85c511c7ce85b6d',console.log)
```

可以得到HelloWorld的合约二进制代码

```
> null
↪'0x60606040526000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff168
↪'
```

3.6.6 查看节点连接

执行

```
//ethconsole版本不同命令稍有不同
web3.admin.getPeers(console.log)
//或
web3.admin.peers(console.log)
```

可看到连接了其它的节点

```
> null [ { caps: [ 'eth/62', 'eth/63', 'pbft/63' ],
    height: '0x25',
    id:
↪'b5adf6440bb0fe7c337eccfda9259985ee42c1c94e0d357e813f905b6c0fa2049d45170b78367649dd0b8b5954ee91
↪',
    lastPing: 0,
    name: 'eth/v1.0/Linux/g++/Interpreter/RelWithDebInfo/0/',
    network: { remoteAddress: '127.0.0.1:30403' },
    notes: { ask: 'Nothing', manners: 'nice', sync: 'holding & needed' } } ]
```

CHAPTER 4

高级合约调用（web3sdk）

Welcome to web3sdk project :-)

web3sdk提供访问fisco-bcos节点的java API

- web3sdk git: FISCO-BCOS/web3sdk
- web3sdk issues

基本特性	• 链上链下(AMOP), 为联盟链提供安全高效的消息信道 • web3j(Web3 Java Ethereum Đapp API)
扩展特性	• 交易结构修改：增加对web3sdk使用者透明的randomid和blocklimit; • 提供系统合约部署工具和系统合约使用工具 • 支持使用国密算法发交易 • 支持将智能合约代码转换成java代码

4.1 环境要求

Important: 使用web3sdk前，请确保：

FISCO BCOS节点环境搭建完成 参考 FISCO-BCOS入门

java版本符合要求 要求 jdk1.8+，推荐使用jdk8u141

网络连通 检查web3sdk连接的FISCO BCOS节点channelPort是否能telnet通，若telnet不通，需要检查网络连通性和安全策略

4.2 SDK编译

安装依赖软件

部署web3sdk之前需要安装git, dos2unix依赖软件:

- **git**：用于拉取最新代码
- **dos2unix**: 用于处理windows文件上传到linux服务器时，文件格式无法被linux正确解析的问题；

centos:

```
$ sudo yum -y install git dos2unix
```

ubuntu:

```
$ sudo apt install git tofrodos
$ ln -s /usr/bin/todos /usr/bin/unxi2dos && ln -s /usr/bin/fromdos /usr/
```

编译源码

执行如下命令拉取并编译源码:

```
#=== 创建并进入web3sdk源码放置目录（假设为~/mydata/） =====
$ mkdir -p ~/mydata
$ cd ~/mydata

#==== 拉取git代码 ====
$ git clone https://github.com/FISCO-BCOS/web3sdk

#===编译we3bsdk源码，生成dist目录 ===
$ cd web3sdk
$ dos2unix *.sh
$ . ./compile.sh

#===编译成功后，web3sdk目录下生成dist文件夹，目录结构如下==========
$ tree -L 2
.
├── build
│   ├── classes
│   ├── ... 省略若干行...
├── build.gradle
├── dist
│   ├── apps   #存放web3sdk.jar
│   ├── bin    #存放可执行脚本compile.sh和web3sdk
│   ├── contracts   #合约存储目录
│   └── lib    #所有jar包存放目录
├── README.md
├── src
│   ├── ...省略若干行...
└── tools
|   ├── bin
|   └── contracts
```

web3sdk编译成功后，会生成dist目录，dist目录主要内容如下:

目录	说明
dist/apps	存放web3sdk编译生成的jar包web3sdk.jar
dist/bin	• web3sdk：调用web3sdk.jar执行web3sdk内方法(如部署系统合约、调用合约工具方法等) • compile.sh：将dist/contracts目录下的合约代码转换成java代码，供开发者使用
dist/conf	配置目录, 用于配置节点信息、证书信息、日志目录等
dist/contracts	合约存放目录，compile.sh脚本可将存放于该目录下的合约代码转换成java代码
dist/lib	存放web3sdk依赖库的jar包

4.3 配置文件

Important:

- 配置web3sdk前，请确保参考 web3sdk编译文档 成功编译web3sdk
- 配置web3sdk前，请先生成客户端证书，并将证书拷贝到web3sdk/dist/conf目录：

1. 手动搭链：客户端证书生成参考 FISCO-BCOS快速入门 **基础配置中的SDK证书配置**；
2. 由 FISCO-BCOS物料包搭建的链 搭建的FISCO-BCOS链：客户端证书生成参考 SDK证书生成
3. 国密版FISCO-BCOS链SDK证书生成参考 SDK证书生成

4.3.1 配置java客户端相关信息

web3sdk客户端配置

打开web3sdk/dist/conf目录的applicationContext.xml文件,部分信息可以先用默认的, **先关注这些配置项**：

```
<!-- 系统合约地址配置，在使用./web3sdk SystemProxy|AuthorityFilter等系统合约工具时需要配置 -->
<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
    <property name="systemProxyAddress" value="【系统合约代理地址,对应节点config.json里的systemproxyaddress】" />
    <!--GOD账户的私钥-->
    <property name="privKey" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的privKey】" />
    <!--GOD账户-->
    <property name="account" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的address】" />
    <property name="outPutpath" value="./output/" />
</bean>

<!-- 区块链节点信息配置 -->
<bean id="channelService" class="org.bcos.channel.client.Service">
    <property name="orgID" value="WB" /> <!-- 配置本机构名称 -->
        <property name="allChannelConnections">
            <map>
                <entry key="WB"> <!-- 配置本机构的区块链节点列表（如有DMZ，则为区块链前置）-->
                    <bean class="org.bcos.channel.handler.ChannelConnections">
                        <property name="caCertPath" value="classpath:ca.crt" />
                        <property name="clientKeystorePath" value="classpath:client.keystore" />
                        <property name="keystorePassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="clientCertPassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="connectionsStr">
                            <list>
                                <value>[nodeid]@[ip]:[channelPort]</value>
                            </list>
                        </property>
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

找到 web3sdk/dist/conf/applicationContext.xml文件的【区块链节点信息配置】 一节，配置keystore密码

```
<property name="keystorePassWord" value="【生成client.keystore时对应的keystore密
码】" />
<property name="clientCertPassWord" value="【生成client.keystore时对应的证书密码】
↪" />
```

配置节点信息，请务必注意：ip、端口，和连接的FISCO-BCOS节点必须一致, 节点id可以是任意非空字符串

```
<property name="connectionsStr">
    <list>
        <!--节点配置：【节点id，可以是任意字符串】@【IP】:【channel port端口】-->
        <value>node1@127.0.0.1:8891</value>
    </list>
</property>
```

其他配置 调用SystemProxy|AuthorityFilter等系统合约工具时需配置系统合约地址SystemProxyAddress和GOD账户信息；GOD账号默认为0x776bd5cf9a88e9437dc783d6414bccc603015cf0 ,GOD账号私钥默认为bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd

```
<!-- 系统合约地址配置，使用系统合约工具时需配置-->
<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
    <!--系统合约地址：【系统合约代理地址,对应节点config.json里
的systemproxyaddress】-->
    <property name="systemProxyAddress" value="0x0" />
    <!--GOD账户的私钥：-->
    <!--非国密版FISCO-BCOS获取GOD账户和账户私钥：【参考https://fisco-bcos-
↪documentation.readthedocs.io/zh_CN/latest/docs/web3sdk/config_
↪web3sdk.html】-->
    <!--国密版FISCO-BCOS获取GOD账户和账户私钥：【参考https://fisco-bcos-
↪documentation.readthedocs.io/zh_CN/latest/docs/guomi/config_guomi.
↪html#sdk】-->
    <property name="privKey" value=
↪"bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd" />
```

(continues on next page)

(continued from previous page)

```
    <!--GOD账户-->
    <property name="account" value=
↪"0x776bd5cf9a88e9437dc783d6414bccc603015cf0" />
    <property name="outPutpath" value="./output/" />
</bean>
```

日志级别配置：修改log4j.properties文件，将 log4j.rootLogger = INFO , C , D , E 中INFO改为 DEBUG 或者ERROR

Important:

- **节点id查询方法**：

1. 查询节点id：若节点服务器上，节点数据目录所在路径为~/mydata/node0/，则节点id可以在~/mydata/node0/data/node.nodeid文件里查到;
2. channelPort、系统合约地址systemcontractaddress等信息查询: 若节点服务器上，节点数据目录所在路径为~/mydata/node0/，在~/mydata/node0/config.json里可查到;

- **god账号信息查询**：

1. 手动搭链：

   ① **非国密版FISCO-BCOS**：设源码位于~/mydata/FISCO-BCOS目录，则god账号信息位于~/mydata/FISCO-BCOS/tools/scripts/godInfo.txt文件中；若搭链过程中使用系统默认god账号，则god账号位于~/mydata/FISCO-BCOS/tools/scripts/god_info/godInfo.txt文件;

   ② **国密版FISCO-BCOS**：设源码位于~/mydata/FISCO-BCOS目录，则god账号位于~/mydata/FISCO-BCOS/tools/scripts/guomi_godInfo.txt文件中；若搭链过程中使用系统默认god账号，则god账号位于~/mydata/FISCO-BCOS/tools/scripts/god_info/guomiDefaultGod.txt

2. 使用 FISCO-BCOS物料包 搭链: 参考 god账号说明 获取god账号信息;

- **这里的端口是对应config.json里的channelPort，而不是rpcport或p2pport**
- **list段里可以配置多个value，对应多个节点的信息，实现客户端多活通信**

4.3.2 测试是否配置成功

测试web3sdk与节点连接是否正常

在web3sdk/dist目录下调用TestOk，非国密版web3sdk输出 `=====INIT ECDSA KEYPAIR From private key===` 等提示，说明web3sdk与节点连接正常，否则请参考 faq【dist/bin/web3sdk运行出错】.

具体测试过程如下：

```
# 进入web3sdk目录(设源码位于~/mydata/web3sdk/dist中)
$ cd ~/mydata/web3sdk/dist

# 调用测试合约TestOk
$ java -cp 'conf/:apps/*:lib/*' org.bcos.channel.test.TestOk
=====================================================================
=====INIT ECDSA KEYPAIR From private key===
============to balance:4
============to balance:8
```

(Ok合约详细代码可参考 Ok.sol)

4.3.3 applicationContext.xml详细介绍

applicationContext.xml配置项详细说明

applicationContext.xml主要包括如下配置选项:

encryptType	**配置国密算法开启/关闭开关(默认为0)** • 0: 不使用国密算法发交易 • 1: 使用国密算法发交易
systemProxyAddress	系统代理合约地址，对应节点config.json中的systemproxyaddress值
privKey	GOD账号私钥,对应生成GOD账号产生godInfo.txt的privKey
account	GOD账号,对应‘生成GOD账号<https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/usage/tools.html>‘_产生的godInfo.txt的address
ChannelConnections	• caCertPath: ca.crt证书路径，默认为classpath:ca.crt • clientKeystorePath: client.keystore证书路径，默认为classpath:client.keystore • keystorePassWord: 生成client.keystore时对应的密码 • clientCertPassWord: 生成client.keystore时对应的密码 • nodeid:SDK连接的FISCO BCOS节点ID，从节点data/node.nodeid文件获取 • ip: SDK连接的FISCO BCOS节点外网ip • channelPort: SDK连接的FISCO BCOS节点channelPort, 对应config.json的channelPort

4.4 应用开发指南

4.4.1 应用开发步骤

主要开发步骤

使用web3sdk开发区块链java应用主要包括如下过程:

1. 数据结构和接口设计，编写合约，并将合约代码转换成java代码
2. 编写应用程序：调用合约接口完成合约部署和调用逻辑
3. 配置java应用
4. 运行并测试java应用

4.4.2 编写合约

合约功能设计：实现简单计数器

实现一个简单的计数器，主要功能包括：

- 设置和读取计数器名字、增加计数、读取当前计数功能。
- 通过receipt log的方式，把修改记录log到区块中，供客户端查询。

(注: receipt log用处很大，是区块链和业务端同步交易处理过程和结果信息的有效渠道)

智能合约代码Counter.sol

根据合约功能要求可实现智能合约 Counter.sol ， 合约代码如下：

```
pragma solidity ^0.4.2;
contract Counter{
    string name;
    uint256 counter;
    //event handle
    event counted(uint256 c,uint256 oldvalue,uint256 currvalue,string memo);
    event changename(string oldname);

    function Counter(){
       name="I'm counter";
       counter = 0;
    }

    function setname(string n){
            name=n;
            changename(n);
    }

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

    function addcount(uint256 c,string memo)
    {
           uint256 oldvalue = counter;
        counter = counter+c;
        counted(c,oldvalue,counter,memo); //event
    }

    function getcount()constant returns(uint256){
        return counter;
    }
}
```

将合约代码Counter.sol转换为java代码Counter.java

web3sdk提供了 `counter_compile.sh` 脚本将Counter.sol转换成Counter.java:

```
# 进入合约编译脚本所在目录 (设web3sdk位于~/mydata目录)
$ cd ~/mydata/web3sdk/dist/bin
# 执行合约编译脚本
# (com是java代码所属的包，转换后可手动修改)
$ bash counter_compile.sh org.bcosliteclient
```

查看生成的java代码

```
$ cd ~/mydata/web3sdk/dist/output
$ tree
# ...此处省略若干输出...
├── Counter.abi   # Counter.sol编译生成的abi文件
├── Counter.bin   # Counter.sol编译生成的bin文件
└── org
    └── bcosliteclient
        ├── Counter.java # Counter.sol转换成的java代码
        ├── Evidence.java
        ├── EvidenceSignersData.java
        └── Ok.java
```

output目录生成了合约的.abi, .bin等文件，以及org/bcosliteclient/Counter.java文件。

这个java文件可以复制到客户端开发环境里，后续建立的java工程的对应的包路径下。

若转换成java代码时报错，请参考 faq【合约转换成java代码出错】 .

Counter.sol对应的Counter.java代码如下：

Counter.java

4.4.3 搭建并配置java应用

下载java应用bcosliteclient

- FISCO-BCOS提供了示例应用bcosliteclient，该应用在 CounterClient.java 中提供Counter.sol合约部署和调用功能。应用下载链接如下:

  bcosliteclient.zip

编译bcosliteclient应用

```
# 解压应用程序(设下载的压缩包bcosliteclient.zip位于~/mydata目录下)
$ cd ~/mydata && unzip bcosliteclient.zip

# 编译bcosliteclient应用
$ cd bcosliteclient && gradle build
```

此时bcosliteclient应用目录如下：

```
$ tree -L 2
├── bcosliteclient # 编译生成目录
│   ├── bin        # 包含部署和调用Counter.sol合约的可执行脚本
│   ├── conf       # 配置文件，包含客户端配置文件applicationContext.xml，客户端证书
│   └── lib        # jar包目录
├── build          # 编译生成的临时目录
│   ├── ...省略若干行...
├── build.gradle
├── lib
│   ├── fastjson-1.2.29.jar
│   └── web3sdk.jar     # 应用引用的web3sdk jar包
└── src
    ├── bin             # 包含可执行程序bcosclient
    ├── contract
    |── org             # 源码目录
    └── resources       # 配置文件目录
```

配置java应用

参考 web3sdk配置 配置java应用，主要配置选项包括：

```
<!-- 系统合约地址配置，在使用./web3sdk SystemProxy|AuthorityFilter等系统合约工具时需要配置 -->
<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
    <property name="systemProxyAddress" value="【系统合约代理地址,对应节点config.json里的systemproxyaddress】" />
    <!--GOD账户的私钥-->
    <property name="privKey" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的privKey】" />
    <!--GOD账户-->
    <property name="account" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的address】" />
    <property name="outPutpath" value="./output/" />
</bean>

<!-- 区块链节点信息配置 -->
<bean id="channelService" class="org.bcos.channel.client.Service">
    <property name="orgID" value="WB" /> <!-- 配置本机构名称 -->
        <property name="allChannelConnections">
            <map>
                <entry key="WB"> <!-- 配置本机构的区块链节点列表（如有DMZ，则为区块链前置） -->
                    <bean class="org.bcos.channel.handler.ChannelConnections">
                        <property name="caCertPath" value="classpath:ca.crt" />
                        <property name="clientKeystorePath" value="classpath:client.keystore" />
                        <property name="keystorePassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="clientCertPassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="connectionsStr">
                            <list>
                                <value>[nodeid]@[ip]:[channelPort]</value>
                            </list>
                        </property>
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

4.4.4 部署和调用合约

部署Counter.sol合约

按照上节操作配置好java应用工程后，可调用相关接口部署和调用Counter.sol合约。

```
# 设bcosliteclient应用位于~/mydata目录
$ cd ~/mydata/bcosliteclient/bcosliteclient/bin
# 部署合约
$ chmod a+x bcosclient && ./bcosclient deploy
-----> start test !
init AOMP ChannelEthereumService
-->Got ethBlockNumber: 42
Deploy contract :null,address :0x8bc176465048ec377a824b7cf36f3cd7452cd093
<--start blockNumber = 42,finish blocknmber=43
```

由输出结果看出，合约部署成功，合约地址为 0x8bc176465048ec377a824b7cf36f3cd7452cd093，且部署成功后，区块链系统区块高度由42增加为43.

调用Counter.sol合约

使用 **bcosliteclient/bcosliteclient/bin** 目录下的 **bcosclient** 脚本调用Counter.sol合约：

```
# 设bcosliteclient应用位于~/mydata目录
$ cd ~/mydata/bcosliteclient/bcosliteclient/bin
# 调用合约(合约地址是0x8bc176465048ec377a824b7cf36f3cd7452cd093)
$ chmod a+x bcosclient && ./bcosclient call_contract␣
↪0x8bc176465048ec377a824b7cf36f3cd7452cd093
```

(continues on next page)

(continued from previous page)

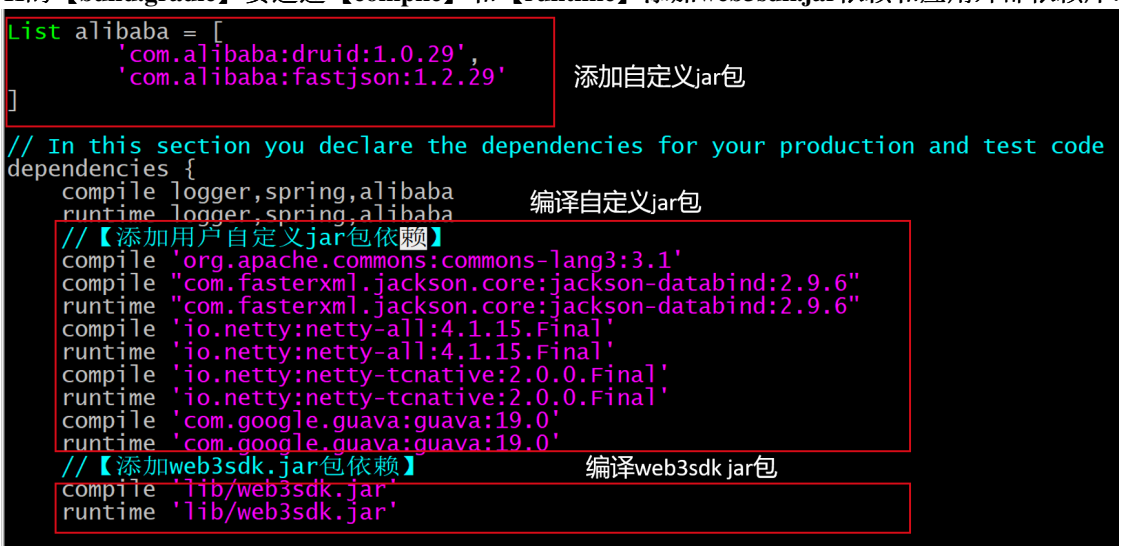
```
-----> start test !
init AOMP ChannelEthereumService
-->Got ethBlockNumber: 43
counter value before transaction:0
setname-->oldname:[MyCounter from:0,inc:100],newname=[MyCounter from:0,inc:100]
Current Counter:100
addcount-->inc:100,before:0,after:100,memo=when tx done,counter inc 100
<--start blockNumber = 43,finish blocknmber=44
```

由输出结果可看出，计数器合约Counter.sol调用成功后，计数器值增加100，区块链系统块高由43增加为44.

4.4.5 gradle文件配置说明

gradle配置文件说明

应用SDK的【build.gradle】要通过【compile】和【runtime】添加web3sdk.jar依赖和应用外部依赖库：



一个完整的build.gradle示例如下：

```
apply plugin: 'maven'
apply plugin: 'java'
apply plugin: 'eclipse'

//指定JDK版本,改成系统中版本
sourceCompatibility = 1.8
targetCompatibility = 1.8

[compileJava, compileTestJava, javadoc]*.options*.encoding = 'UTF-8'

// In this section you declare where to find the dependencies of your project
repositories {
    maven {
        url "http://maven.aliyun.com/nexus/content/groups/public/"
    }

    mavenLocal()
    mavenCentral()
}
```

(continues on next page)

(continued from previous page)

```

List logger = [
        "org.slf4j:jul-to-slf4j:1.7.10",
        "org.apache.logging.log4j:log4j-api:2.1",
        "org.apache.logging.log4j:log4j-core:2.1",
        "org.apache.logging.log4j:log4j-slf4j-impl:2.1",
        "org.apache.logging.log4j:log4j-web:2.1"
]

def spring_version="4.3.16.RELEASE"
List spring =[
        "org.springframework:spring-core:$spring_version",
        "org.springframework:spring-beans:$spring_version",
        "org.springframework:spring-context:$spring_version",
        "org.springframework:spring-tx:$spring_version",
        "org.springframework:spring-jdbc:$spring_version",
        "org.springframework:spring-test:$spring_version"
]

List alibaba = [
        'com.alibaba:druid:1.0.29',
        'com.alibaba:fastjson:1.2.29'
]

// In this section you declare the dependencies for your production and test code
dependencies {
    compile logger,spring,alibaba
    runtime logger,spring,alibaba
    //【添加用户自定义jar包依赖】
    compile 'org.apache.commons:commons-lang3:3.1'
    compile "com.fasterxml.jackson.core:jackson-databind:2.9.6"
    runtime "com.fasterxml.jackson.core:jackson-databind:2.9.6"
    compile 'io.netty:netty-all:4.1.15.Final'
    runtime 'io.netty:netty-all:4.1.15.Final'
    compile 'io.netty:netty-tcnative:2.0.0.Final'
    runtime 'io.netty:netty-tcnative:2.0.0.Final'
    compile 'com.google.guava:guava:19.0'
    runtime 'com.google.guava:guava:19.0'
    //【添加web3sdk.jar包依赖】
    compile 'lib/web3sdk.jar'
    runtime 'lib/web3sdk.jar'

            // web3j依赖
            compile 'org.apache.httpcomponents:httpclient:4.5.5',
             'org.bouncycastle:bcprov-jdk15on:1.54',
             'com.lambdaworks:scrypt:1.4.0',
             'com.squareup:javapoet:1.7.0',
             'io.reactivex:rxjava:1.2.4',
                         'com.github.jnr:jnr-unixsocket:0.15'

        //testCompile 'junit:junit:4.12'
}

sourceSets {
        main {
                java {
                srcDir 'src/main/java'
                srcDir 'src/test/java'
                }
        resources  {
```

(continues on next page)

(continued from previous page)

```
            srcDir 'src/main/resources'
        }
        }
}

jar {
        destinationDir file('dist/apps')
        archiveName project.name + '.jar'
        exclude '**/*.xml'
        exclude '**/*.properties'

    doLast {
        copy {
            from file('tools/')
            into 'dist/'
        }
                copy {
                        from configurations.runtime
                        into 'dist/lib'
                }
                copy {
            from file('src/test/resources/')
            into 'dist/conf'
        }
                copy {
                        from file('.').listFiles().findAll{File f -> (f.name.
↪endsWith('.crt') || f.name.endsWith('.keystore'))}
                        into 'dist/conf'
                }
        }
}
```

4.4.6 总结

SDK应用开发步骤总结

根据以上描述，使用web3sdk开发区块链应用主要包括如下过程：

1. 根据应用功能设计合约数据结构和接口；
2. 编写智能合约，可先用Nodejs简单验证合约代码逻辑是否正确，验证通过后，将合约代码转换成java代码
3. 编写java应用，调用合约java接口完成合约部署和调用功能
4. 配置并编译java应用
5. 应用功能测试

SDK应用部署/调用合约主要流程

参考 CounterClient.java：

1. 初始化AMOP的ChannelEthereumService
2. 使用AMOP初始化Web3j
3. 初始化交易签名密钥对
4. 初始化交易参数
5. 调用合约接口部署或调用合约

其他说明

- 从零开发SDK应用时，可使用eclipse新建java工程，编译配置文件build.gradle可参考bcosliteclient.zip中的编译配置;
- java应用跟目录的lib目录下要存放FISCO BCOS的web3sdk.jar，web3sdk升级时，直接替换java应用的web3sdk.jar到最新即可。

参考资料

- 智能合约参考文档：http://solidity.readthedocs.io/en/v0.4.24/
- AMOP: https://fisco-bcos-documentation.readthedocs.io/zh_CN/latest/docs/features/AMOP/README.html
- web3j JSON-RPC: https://github.com/ethereum/wiki/wiki/JSON-RPC
- **FISCO dev团队提供的示例应用:**
  1. 存证Demo： https://github.com/FISCO-BCOS/evidenceSample
  2. 群/环签名客户端Demo: https://github.com/FISCO-BCOS/sig-service-client
  3. depotSample服务Demo: https://github.com/FISCO-BCOS/depotSample

4.5 SDK功能列表

4.5.1 系统合约部署

文档目标

web3sdk提供了系统合约部署工具，本文主要介绍如何使用web3sdk部署系统合约

Important:

- 部署系统合约前，请参考 web3sdk入门 搭建并运行web3sdk
- 部署系统合约前，请保证web3sdk连接的FISCO-BCOS节点运行正常(可以正常出块)
- 若FISCO-BCOS节点已部署系统合约，没有必要再用web3sdk再次部署系统合约

web3sdk部署系统合约详细步骤

使用 `./web3sdk InitSystemContract` 命令部署系统合约:

```
#-----进入dist目录(设web3sdk存放于/mydata/目录)
$ cd /mydata/web3sdk/dist/bin

#-----执行部署工具InitSystemContract部署系统合约:
$ ./web3sdk InitSystemContract
=====================================================================
Start deployment...
=====================================================================
systemProxy getContractAddress 0xc9ed60a2ebdf22936fdc920133af2a77dd553e13
```

(continues on next page)

(continued from previous page)

```
caAction getContractAddress 0x014bf33e022f78f7c4bb8dbfe1d22df5168fc9bc
nodeAction getContractAddress 0x51b25952b01a42e6f84666ed091571c7836eda34
consensusControlMgr getContractAddress 0xffda2977b8bd529dd187d226ea6600ff3c8fb716
configAction getContractAddress 0xf6677fa9594c823abf39ac67f1f34866e2843399
fileInfoManager getContractAddress 0x0f49a17d17f82da2a7d92ecf19268274150eaf5e
fileServerManager getContractAddress 0xfbe0184fe09a3554103c5a541ba052f7fa45283b
contractAbiMgr getContractAddress 0x66ec295357750ce442227a6419ada7fdf9207be2
authorityFilter getContractAddress 0x2fa1ec76f3e31d2c42d21b62960625f326a044e6
group getContractAddress 0xa172b92c85a98167d96b9fde10792eb2fd4d584c
transactionFilterChain getContractAddress␣
↪0x0b78d9be55f047fb32d6fbc2c79013c0eca5d09d
Contract Deployment Completed System Agency
Contract:0xc9ed60a2ebdf22936fdc920133af2a77dd553e13
-----------------System routing table----------------------
0)TransactionFilterChain=>0x0b78d9be55f047fb32d6fbc2c79013c0eca5d09d,false,35
      AuthorityFilter=>1.0,0x2fa1ec76f3e31d2c42d21b62960625f326a044e6
1)ConfigAction=>0xf6677fa9594c823abf39ac67f1f34866e2843399,false,36
2)NodeAction=>0x51b25952b01a42e6f84666ed091571c7836eda34,false,37
3)ConsensusControlMgr=>0xffda2977b8bd529dd187d226ea6600ff3c8fb716,false,38
4)CAAction=>0x014bf33e022f78f7c4bb8dbfe1d22df5168fc9bc,false,39
5)ContractAbiMgr=>0x66ec295357750ce442227a6419ada7fdf9207be2,false,40
6)FileInfoManager=>0x0f49a17d17f82da2a7d92ecf19268274150eaf5e,false,41
7)FileServerManager=>0xfbe0184fe09a3554103c5a541ba052f7fa45283b,false,42
```

部署完毕的系统合约地址是 `0xc9ed60a2ebdf22936fdc920133af2a77dd553e13`

系统合约配置

部署完系统合约后，若要使用该系统合约，需要如下操作:

- 将 `dist/conf/applicationContext.xml` 的 `systemProxyAddress` 字段更新为输出的系统合约地址
- 将输出的系统合约地址更新到所有FISCO-BCOS节点config.json的 `systemProxyAddress` 字段，并重启节点

4.5.2 系统合约工具

文档目标

web3sdk提供了系统合约管理工具，本文档主要介绍这些管理工具，系统合约介绍参考 FISCO BCOS系统合约介绍

系统合约简单介绍

系统合约	详细说明
SystemProxy	系统合约代理合约
TransactionFilterChain	设置transaction过滤器
ConfigAction	设置/获取区块链系统参数
ConsensusControlMg	联盟控制合约
CAAction	**证书列表黑名单管理:** 包括将证书加入黑名单列表，将证书从黑名单列表删除，获取证书黑名单列表功能
ContractAbiMgr	ABI相关合约

系统合约代理合约SystemProxy

SystemProxy

功能 遍历系统代理合约路由表，输出所有系统合约地址

使用方法:

```
#进入程序web3sdk所在目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk SystemProxy #调用SystemProxy
```

节点管理合约NodeAction

NodeAction

节点加入记账者列表 `./web3sdk NodeAction registerNode ${node_json_path}` 命令将${node_json_path} (${node_json_path}是节点配置文件相对于dist/conf的路径) 指定的节点加入到FISCO BCOS区块链网络中:

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk NodeAction registerNode ${node_json_path}
```

节点配置文件主要包括如下配置项:

配置项	详细说明
id	节点node id
ip	节点IP
port	节点P2P连接端口
desc	节点描述
agencyinfo	节点所属机构信息
idx	节点序号，按照加入顺序排序

一个简单的节点配置文件node.json示例如下:

```
{
    "id":
↪"2cd7a7cadf8533e5859e1de0e2ae830017a25c3295fb09bad3fae4cdf2edacc9324a4fd89cfee174b21546f933
↪",
    "ip":"127.0.0.1",
    "port":30501,
    "desc":"node1",
    "CAhash":"",
    "agencyinfo":"node1",
    "idx":0
}
```

节点退出记账者列表 `./web3sdk NodeAction cancelNode ${node_json_path}`
将${node_json_path}指定的节点从FISCO BCOS区块链网络中退出(${node_json_path}是节点配置文件相对于~/mydata/web3sdk/dist/conf的路径，节点配置文件说明同上):

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk NodeAction cancelNode ${node_json_path}
```

显示记账列表信息 `./web3sdk NodeAction all` 查询区块链网络中所有记账节点信息:

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk NodeAction all
# 输出的记账节点信息如下:
================================================================
=====INIT ECDSA KEYPAIR From private key===
NodeIdsLength= 1
----------node 0---------
id=28f815c7222118adaca6dfdefdda76906a491ae4ef9de4d311f3f23bd2371ee9d15e2f26646d1641bf6391
nodeA
agencyA
E2746FDF0B29F8A8
```

节点证书管理合约CAAction

CAAction

将指定节点证书加入黑名单证书列表 `./web3sdk CAAction add ${node_ca_path}`
将${node_ca_path}(${node_ca_path}是节点配置文件相对于dist/conf的路径)指定的节点证书信息添加到黑名单证书列表，加入成功后，其他节点将拒绝与此节点连接:

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk CAAction add ${node_ca_path}
```

从黑名单证书列表中删除指定节点证书信息 `./web3sdk CAAction remove ${node_ca_path}`
将${node_ca_path}(${node_ca_path}是节点配置文件相对于dist/conf的路径)指定的节点证书信息从黑名单证书列表中删除，其他节点恢复与该节点的连接:

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk CAAction remove ${node_ca_path}
```

显示区块链黑名单证书列表信息 `./web3sdk CAAction all` 获取记录在系统合约CAAction中的黑名单证书列表信息:

```
#进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
$ ./web3sdk CAAction all
```

系统参数配置合约ConfigAction

ConfigAction

获取系统参数 `./web3sdk ConfigAction get ${key}` 系统合约ConfigAction读取${key}对应的值(ConfigAction中记录的系统参数说明参考 系统参数说明文档)

设置系统参数 `./web3sdk ConfigAction set ${key} ${setted_value}` 将记录在系统合约ConfigAction中${key}对应的值设置为${setted_value}(ConfigAction中记录的系统参数说明参考 系统参数说明文档)

FISCO BCOS系统合约ConfigAction主要配置选项

配置项	详细说明
maxBlockTransactions	控制一个块内允许打包的最大交易数量上限 设置范围: [0, 2000), 默认值:1000
intervalBlockTime	设置出块间隔时间 设置范围：大于等于1000, 默认值: 1000
maxBlockHeadGas	控制一个块允许最大Gas消耗上限 取值范围: 大于等于2000,000,000, 默认值: 2000,000,000
maxTransactionGas	设置一笔交易允许消耗的最大gas 取值范围: 大于等于30,000,000, 默认值: 30,000,000
maxNonceCheckBlock	控制Nonce排重覆盖的块范围 取值范围：大于等于1000, 缺省值: 1000
maxBlockLimit	控制允许交易上链延迟的最大块范围 取值范围：大于等于1000, 缺省值：1000
CAVerify	控制是否打开CA验证,取值：true或者false, 缺省值: false
omitEmptyBlock	控制是否忽略空块 取值：true或者false, 缺省值：false

通过ConfigAction配置系统参数的例子如下:

```
# 进入bin目录(设web3sdk代码位于~/mydata/目录)
$ cd ~/mydata/web3sdk/dist/bin
$ chmod a+x web3sdk
# =====更改出块时间为2s====
$ ./web3sdk ConfigAction set intervalBlockTime 2000

# =====允许空块落盘=====
$ ./web3sdk ConfigAction set omitEmptyBlock false

# ====调整一笔交易允许消耗的最大交易gas为40,000,000
$ ./web3sdk ConfigAction set maxTransactionGas 40000000

# ====调整一个块允许消耗的最大交易gas为3000,000,000
$ ./web3sdk ConfigAction set maxBlockHeadGas 3000000000

# ==== 打开CA认证开关====
$ ./web3sdk ConfigAction set CAVerify true
# ......
```

联盟控制合约ConsensusControl

ConsensusControl

部署联盟共识模板 ./web3sdk ConsensusControl deploy

获取联盟共识模板合约地址 ./web3sdk ConsensusControl list

关闭联盟共识特性 ./web3sdk ConsensusControl turnoff

4.5.3 客户端证书生成

文档目标

FISCO-BCOS提供了客户端证书生成脚本 sdk ,该脚本生成客户端证书 `ca.crt` 和 `client.keystore` , 本文档详细介绍客户端证书 `client.keystore` 生成方法。

ca.crt根证书生成方法

FISCO-BCOS区块链系统中，web3sdk与所连接的FISCO-BCOS节点必须**属于同一机构**，为了使客户端能连上FISCO-BCOS节点，web3sdk的根证书 `ca.crt` 必须同时包含FISCO-BCOS区块链的根证书和机构证书, 因此web3sdk的根证书 `ca.crt` 由链根证书和机构证书合成(设链证书为 `ca.crt` ,机构证书为 `agency.crt`, 最终输出的web3sdk根证书为 `ca.crt`),web3sdk根证书生成方法如下:

```
#将链证书拷贝到web3sdk根证书
#设链证书位于~/mydata/node0/data目录
#设web3sdk位于~/mydata/web3sdk目录
$ cd ~/mydata/web3sdk/dist/conf #进入web3sdk配置路径
$ cp ~/mydata/node0/data/ca.crt ca-agency.crt
#追加机构证书到web3sdk根证书
$ more ~/mydata/node0/data/agency.crt | cat >>ca-agency.crt
#重命名web3sdk根证书为ca.crt
$ mv ca-agency.crt ca.crt
```

client.keystore证书生成方法

`client.keystore` 是web3sdk与前置sdk连接的认证证书，在 存证案例 中, `client.keystore` 中包含的私钥也用于为交易签名，其生成方法如下:

```
#(1) web3sdk所属机构颁发sdk证书sdk.crt
# 使用ECDSA算法,生成公私钥对(sdk.pubkey, sdk.key)
# 设web3sdk位于~/mydata/web3sdk路径
 $ cd ~/mydata/web3sdk/dist/conf  #进入web3sdk配置路径
 $ openssl ecparam -out sdk.param -name secp256k1
 $ openssl genpkey -paramfile sdk.param -out sdk.key
 $ openssl pkey -in sdk.key -pubout -out sdk.pubkey

 # 生成证书sdk.crt
 $ openssl req -new -key sdk.key -config cert.cnf  -out sdk.csr
 $ openssl x509 -req -days 3650 -in sdk.csr -CAkey agency.key -CA agency.crt -
↪force_pubkey sdk.pubkey -out sdk.crt -CAcreateserial -extensions v3_req -extfile␣
↪cert.cnf
```

(continues on next page)

(continued from previous page)

```
#(2) 将生成的sdk证书导入client.keystore
# 生成临时文件keystore.p12
$ openssl pkcs12 -export -name client -in sdk.crt -inkey sdk.key -out keystore.p12
# 将keystore.p12导入client.keystore
$ keytool -importkeystore -destkeystore client.keystore -srckeystore keystore.p12␣
↪-srcstoretype pkcs12 -alias client
```

4.5.4 合约代码转换为java代码

文档目标

区块链系统因其开放性、不可篡改性、去中心化特性成为构建可信的去中心化应用的核心解决方案，Java是当前最主流的编程语言之一, 区块链系统支持java调用智能合约很重要。本文档主要介绍如何将智能合约转换成java代码

参考资料

- web3sdk git: https://github.com/FISCO-BCOS/web3sdk
- 智能合约参考文档：http://solidity.readthedocs.io/en/v0.4.24/

Important: 必须先安装好FISCO-BCOS的solidity编译器 fisco-solc ，详细步骤参考 FISCO-BCOS入门

将合约代码转换成java代码

web3sdk提供了转换脚本compile.sh，可将合约代码转换成java代码: - 转换脚本 `compile.sh` 存放路径: `web3sdk/dist/bin/compile.sh` - 转换脚本 `compile.sh` 脚本用法：(`${package}` 时生成的java代码import的包名)

1. 合约代码转换成不包含国密特性的java代码(所有版本均支持): `bash compile.sh ${package}`
2. 合约代码转换成支持国密特性的java代码(1.2.0版本 后支持):`bash compile.sh ${package} 1`

- 建议使用 1.2.0版本 后的web3sdk时，生成支持国密特性的java代码，应用有更大可扩展空间

合约代码转换为java代码操作步骤

将要转换的合约代码复制到 `web3sdk/dist/contracts` 目录

```
$ cd ~/mydata/web3sdk/tool/contracts
$ ls
EvidenceSignersData.sol  Evidence.sol  Ok.sol
#----进入compile.sh脚本所在路径(设web3sdk代码路径是~/mydata/web3sdk)
$ cd ~/mydata/web3sdk/dist/bin
```

`web3sdk/dist/contracts` 目录下所有智能合约转换成不支持国密特性的java代码

```
#执行compile.sh脚本，将~/mydata/web3sdk/dist/contract目录下所有合约代码转换
成java代码
#(com是java代码所属的包，转换后可手动修改)
$ bash compile.sh com
```

查看生成的java代码(位于~/mydata/web3sdk/dist/output)

```
$ tree
# ...此处省略若干输出...
├── Ok.abi
├── Ok.bin
└── com
    ├── Evidence.java
    ├── EvidenceSignersData.java
    └── Ok.java
```

高版本可选项：将合约转换成支持国密特性java代码(web3sdk版本号>= 1.2.0时,推荐使用)

```
$ bash compile.sh com 1
```

4.5.5 web3j API说明

文档目标

本文档简单介绍 **web3sdk常用API** 以及 **权限控制API**，适用于区块链客户端开发者。

参考资料

- web3j RPC接口
- web3j官方文档
- 权限控制API接口
- 权限控制设计文档
- 联盟链权限控制体系说明

web3sdk常用API

web3sdk常用API

获取wbe3j版本	./web3sdk web3_clientVersion
获取当前块高	./web3sdk eth_blockNumber
获取当前PBFT view	./web3sdk eth_pbftView
获取指定区块指定合约的二进制代码	./web3sdk eth_getCode address blockNumber
根据交易哈希获取交易详情	./web3sdk eth_getBlockTransactionCountByHash blockHash
获取指定账户在指定块高执行交易数	./web3sdk eth_getTransactionCount address blockNumber
获取指定块高的交易总数	./web3sdk eth_getBlockTransactionCountByNumber blockNumber
签名的交易数据上链	./web3sdk eth_sendRawTransaction signTransactionData
根据区块哈希获取区块	./web3sdk eth_getBlockByHash blockHash
根据块高获取区块	./web3sdk eth_getBlockByNumber blockNumber
获取指定区块指定位置的交易	./web3sdk eth_getTransactionByBlockHashAndIndex blockHash transactionPosition ./web3sdk eth_getTransactionByBlockNumberAndIndex blockNumber transactionPosition
根据交易哈希获取交易回执	./web3sdk eth_getTransactionReceipt transactionHash

权限控制API

权限控制API

权限控制配置 在applicationContext.xml中配置账户和账户私钥(部署权限控制合约时，建议用GOD账户和GOD账户私钥):

```
<!-- 系统合约地址配置置-->
<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
<!--系统合约-->
<property name="systemProxyAddress" value=
↪"0x919868496524eedc26dbb81915fa1547a20f8998" />
<!--GOD账户的私钥-->（注意去掉"0x"）
<property name="privKey" value=
↪"bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd" />
<!--God账户-->
<property name="account" value=
↪"0x776bd5cf9a88e9437dc783d6414bccc603015cf0" />
<property name="outPutpath" value="./output/" />
</bean>
```

权限控制接口命令

```
./web3sdk ARPI_Model
./web3sdk PermissionInfo
./web3sdk FilterChain addFilter name1 version1 desc1
./web3sdk FilterChain delFilter num
./web3sdk FilterChain showFilter
./web3sdk FilterChain resetFilter
./web3sdk Filter getFilterStatus num
./web3sdk Filter enableFilter num
./web3sdk Filter disableFilter num
./web3sdk Filter setUsertoNewGroup num account
```

(continues on next page)

(continued from previous page)

```
./web3sdk Filter setUsertoExistingGroup num account group
./web3sdk Filter listUserGroup num account
./web3sdk Group getBlackStatus num account
./web3sdk Group enableBlack num account
./web3sdk Group disableBlack num account
./web3sdk Group getDeployStatus num account
./web3sdk Group enableDeploy num account
./web3sdk Group disableDeploy num account
./web3sdk Group addPermission num account A.address fun(string)
./web3sdk Group delPermission num account A.address fun(string)
./web3sdk Group checkPermission num account A.address fun(string)
./web3sdk Group listPermission num account
```

4.6 FAQ

4.6.1 dist/bin/web3sdk运行出错

dist/bin/web3sdk运行出错

permission denied错误 web3sdk无可执行权限，尝试运行 `chmod +x dist/bin/web3sdk`

com.fasterxml.jackson.databind.JsonMappingException: No content to map due to end-of-input

可能是节点连接异常，使用如下方法排错:

- 检查dist/conf/applicationContext.xml的节点配置: 必须设置成连接的FISCO-BCOS节点的channelPort
- 检查FISCO-BCOS节点listenip: 必须设置成服务器IP或者0.0.0.0
- 检查网络连通性：telnet连接的FISCO-BCOS节点的ip和channelPort，必须能telnet通，若不通，请检查网络策略
- 检查ca证书ca.crt: 必须与连接的FISCO-BCOS节点的ca.crt一致
- 检查客户端证书：解决方法参考 FISCO-BCOS中client.keystore 的生成方法

4.6.2 合约转换成java代码出错

合约转换成java代码出错

参考 web3sdk issue1: 【使用工具包生成合约Java Wrap代码时报错】，具体解决方法:

```
#------进入web3sdk代码目录(设web3sdk是~/mydata/web3sdk-master下)
$ cd ~/mydata/web3sdk-master

#------删除已经以前的编译文件
$ rm -rf dist

#------重命名web3sdk-master
$ cd ..
$ mv web3sdk-master web3sdk
$ cd web3sdk

#-------重新编译web3sdk
$ gradle build
```

原因分析 从git载代码（Download ZIP）解压后目录为 **web3sdk-master** ，编译后生成 `dist/apps/web3sdk-master.jar` ，与 `dist/bin/web3sdk` 中配置的 `CLASSPATH` 中的配置项 `$APP_HOME/apps/web3sdk.jar` 名称不一致，导致调用工具包将合约代码转换为java代码出错

CHAPTER 5

企业搭链工具（物料包）

通过简单配置 ,可以在指定服务器上构建FISCO BCOS的区块链, 构成FISCO BCOS的节点既可以直接运行于服务器, 也可以是docker节点。 可以非常快速的搭建临时使用的测试环境, 又能满足生产环境的需求。 例如:

配置三台服务器, 每台启动两个FISCO BCOS节点, 则将生成三个安装包, 对应三台服务器, 将安装包上传到对应的服务器上, 继续按照指引安装, 在每台服务器上启动节点, 就可以组成一个区块链网络。

- **一键部署工具**: 降低FISCO BCOS部署难度
- **区块链扩容支持**: 提供FISCO BCOS区块链快速扩容支持
- **JAVA安装**：FISCO BCOS中需要使用Oracle JDK 1.8(java 1.8)环境，提供FISCO BCOS支持的JAVA组件
- **物料包搭建FISCO BCOS环境CheckList**：检查系统是否满足搭建FISCO BCOS的条件

术语简介

- 两种节点类型：**创世节点, 非创世节点**。
- **创世节点**：搭建一条新链时, 配置文件的第一台服务器上的第一个节点为创世节点, 创世节点是需要第一个启动的节点, 其他节点需要主动连接创世节点, 通过与创世节点的连接, 所有节点同步连接信息, 最终构建正常的网络。
- **非创世节点**：除去创世节点的其它节点。

版本支持

物料包与FISCO BCOS之间存在版本对应关系:

- 物料包1.2.X版本对应支持FISCO BCOS的版本为：1.3.X，即物料包1.2的版本兼容支持FISCO BCOS1.3的版本

物料包的相关流程如下:

5.1 环境依赖

5.1.1 依赖

- 机器配置参考FISCO BCOS机器配置

```
使用的测试服务器:
CentOS 7.2 64位
CentOS 7.4 64位
Ubuntu 16.04 64位
```

- 软件依赖

```
Oracle JDK[1.8]
```

依赖说明

- FISCO BCOS搭建过程中需要的其他依赖会自动安装, 用户不需要再手动安装。
- CentOS/Ubuntu默认安装或者使用yum/apt安装的是openJDK, 并不符合使用要求, Oracle JDK 1.8 的安装链接Oracle JDK 1.8 安装
- 其他依赖sudo权限, 当前执行的用户需要具有sudo权限。

5.1.2 CheckList环境检查

使用物料包之前建议使用checkList文档对当前环境进行检查, 部署生产环境时建议将checkList作为必备的流程。

5.2 部署区块链

- 本章节会通过一个示例说明如何使用物料包工具, 也会介绍使用物料包构建好的环境中比较重要的一些目录
- 如果你希望快速搭建fisco bcos测试环境，请转至部署区块链sample

5.2.1 下载物料包

```
$ git clone https://github.com/FISCO-BCOS/fisco-package-build-tool.git
```

目录结构以及主要配置文件作用:

```
fisco-package-build-tool
├── Changelog.md                          更新记录
├── config.ini                            配置文件
├── doc                                   附加文档
├── ext                                   拓展目录
├── generate_installation_packages.sh     启动shell文件
├── installation_dependencies             依赖目录
├── LICENSE                               license文件
├── README.md                             使用手册
└── release_note.txt                      版本号
```

5.2.2 配置

```
$ cd fisco-package-build-tool
$ vim config.ini
```

配置文件config.ini

```
[common]
* 物料包拉取FISCO-BCOS源码的github地址，用户一般不需要修改.
github_url=https://github.com/FISCO-BCOS/FISCO-BCOS.git
* 物料包拉取FISCO-BCOS源码之后，会将源码保存在本地的目录，保存的目录名称为FISCO-BCOS，默认拉取的代码会放入物料包同级的目录.
fisco_bcos_src_local=../
* 需要使用FISCO-BCOS的版本号，使用物料包时需要将该值改为需要使用FISCO-BCOS的版本号.
* 版本号可以是FISCO-BCOS已经发布的版本之一，链接： https://github.com/FISCO-BCOS/FISCO-
↪BCOS/releases
fisco_bcos_version=v1.3.2

[docker]
* docker开关，打开时构建的FISCO BCOS链节点为docker节点 . 0:关闭  1:打开
docker_toggle=0
* docker仓库地址.
docker_repository=fiscoorg/fisco-octo
* docker镜像版本号，使用时修改为需要的版本号.
docker_version=v1.3.1

* 生成web3sdk证书时使用的keystore与clientcert的密码.
* 也是生成的web3sdk配置文件applicationContext.
↪xml中keystorePassWord与clientCertPassWord的值，强烈建议用户修改这两个值.
[web3sdk]
keystore_pwd=123456
clientcert_pwd=123456

[other]
* CA拓展模式，目前不使用
ca_ext=0

* 扩容使用的一些参数
[expand]
* 扩容依赖的文件的目录,具体使用参考扩容流程
genesis_follow_dir=/follow/

* 端口配置，一般不用做修改，使用默认值即可，但是要注意不要端口冲突.
* 每个节点需要占用三个端口:p2p port、rpc port、channel port，对于单台服务器上的节点端口使用规则，默认配置的端口开始，依次增长。
[ports]
* p2p端口
p2p_port=30303
* rpc端口
rpc_port=8545
* channel端口
channel_port=8821

* 节点信息
[nodes]
* 格式为 : nodeIDX=p2p_ip listen_ip num agent
* IDX为索引，从0开始增加.
* p2p_ip      => 服务器上用于p2p通信的网段的ip.
* listen_ip   => 服务器上的监听端口，用来接收rpc、channel的链接请求，建议默认值为"0.0.0.0".
* num         => 在服务器上需要启动的节点的数目.
* agent       => 机构名称，若是不关心机构信息，值可以随意，但是不可以为空.
node0=127.0.0.1  0.0.0.0  4  agent
```

配置详解

- [common] section

```
[common]
* 物料包拉取FISCO-BCOS源码的github地址.
github_url=https://github.com/FISCO-BCOS/FISCO-BCOS.git
* 物料包拉取FISCO-BCOS源码之后, 会将源码保存在本地的目录, 保存的目录名称为FISCO-BCOS.
fisco_bcos_src_local=../
* 需要使用FISCO-BCOS的版本号, 使用物料包时需要将该值改为需要使用的版本号.
* 版本号可以是FISCO-BCOS已经发布的版本之一, 链接: https://github.com/FISCO-BCOS/FISCO-
↪BCOS/releases
fisco_bcos_version=v1.3.2
```

- 物料包在构建安装包过程中(非扩容流程), 会启动一个默认的临时temp节点用来部署进行系统合约的部署, 将所有的节点注册到节点管理合约, 然后导出导出系统合约信息生成genesis.json文件。
- 在扩容流程中, 需要手动将扩容节点注册到节点管理合约, 参考后面的扩容流程。
- 每个节点启动时需要占用三个端口: p2p、rpc、channel. 对于启动的临时节点temp节点, 使用就是配置的nodes section中的p2p_port、rpc_port、channel_port配置端口, 要确认端口没有被占用。
- 对于FISCO BCOS节点, 每台服务器上第一个节点使用nodes section配置的p2p_port、rpc_port、channel_port端口, 后续的节点端口依次进行端口递增。
- 工具构建安装包过程中会涉及到从github上面拉取FISCO BCOS、编译FISCO BCOS流程, 规则:

a、检查/usr/local/bin是否已经存在已经编译的fisco-bcos。

b、若存在则查看fisco-bcos的版本号与配置fisco_bcos_version字段的版本号是否一致, 一致则说明fisco-bcos是需要的版本, 该流程退出。

c、判断配置文件中fisco_bcos_src_local配置的路径是否存在名为FISCO-BCOS的文件夹, 存在说明FISCO-BCOS源码已经存在, 否则开始从github上面拉取FISCO BCOS源码, 接下来进入FISCO-BCOS目录切换tag到自己需要的分支, 然后进行编译、安装流程, 成功则fisco-bcos会安装在/usr/local/bin目录下。

- [docker] section

与docker节点搭链相关的配置, 参考FISCO BCOS docker。

- [web3sdk] section

与私钥跟证书文件的加解密相关。生成web3sdk证书时使用的keystore与clientcert的密码, 也是生成的web3sdk配置文件applicationContext.xml中keystorePassWord与clientCertPassWord的值, **建议用户修改这两个值**。

- [expand] section

扩容操作, 使用参考 (扩容流程)

- [nodes] section

```
需要部署FISCO BCOS服务器上的节点配置信息。
[nodes]
* 格式为 : nodeIDX=p2p_ip listen_ip num agent
* IDX为索引, 从0开始增加
* p2p_ip      => 服务器上用于p2p通信的网段的ip
* listen_ip   => 服务器上的监听端口, 用来接收rpc、channel的链接请求, 建议默认值为"0.0.0.0"
* num         => 在服务器上需要启动的节点的数目
* agent       => 机构名称, 若是不关心机构信息, 值可以随意, 但是不可以为空
```

- [ports] section

```
[ports]
* p2p端口
p2p_port=30303
* rpc端口
rpc_port=8545
* channel端口
channel_port=8821
```

fisco-bcos的每个节点需要使用3个端口,p2pport、rpcport、channelport, [ports]配置的端口是服务器上面的第一个节点使用的端口,其他节点依次递增

```
node0=127.0.0.1 0.0.0.0 4 agent
```

上面的配置说明要启动四个节点, 按照默认的配置:

- 第1个节点的端口：p2p 30303、rpc 8545、channel 8821
- 第2个节点的端口：p2p 30304、rpc 8546、channel 8822
- 第3个节点的端口：p2p 30305、rpc 8547、channel 8823
- 第4个节点的端口：p2p 30306、rpc 8548、channel 8824

下面以在三台服务器上部署区块链为例构建一条新链:

```
服务器ip  :  172.20.245.42 172.20.245.43 172.20.245.44
机构分别为:  agent_0   agent_1    agent_2
节点数目  :  每台服务器搭建两个节点
```

修改[nodes] section字段为:

```
[nodes]
node0=172.20.245.42  0.0.0.0  2  agent_0
node1=172.20.245.43  0.0.0.0  2  agent_1
node2=172.20.245.44  0.0.0.0  2  agent_2
```

5.2.3 创建安装包

```
$ ./generate_installation_packages.sh build
```

执行成功之后会生成build目录, 目录下有生成的对应服务器的安装包:

```
build/
├── 172.20.245.42_agent_0_genesis  //172.20.245.42服务器的安装包
├── 172.20.245.43_agent_1          //172.20.245.43服务器的安装包
├── 172.20.245.44_agent_2          //172.20.245.44服务器的安装包
├── stderr.log       //标准错误的重定向文件
└── temp             //temp节点安装目录, 用户不用关心不用关心
```

- 其中带有**genesis**字样的为创世节点所在服务器的安装包。

5.2.4 上传

- 将安装包上传到对应的服务器, 注意上传的安装包必须与服务器相对应, 否则部署过程会出错。
- 一定要确认各个服务器之间的网络可连通, p2p网段的端口网络策略已经放开。

5.2.5 安装

进入安装目录, 执行

```
$ ./install_node.sh
```

正确执行在当前目录会多一个build目录, 目录结构如下:

```
build
├── check.sh            #检查脚本, 可以检查节点是否启动
├── fisco-bcos          #fisco-bcos二进制程序
├── node0               #节点0的目录
├── node1               #节点1的目录
├── nodejs              #nodejs相关安装目录
├── node_manager.sh     #节点管理脚本
├── node.sh             #nodejs相关环境变量
├── register.sh         #注册节点入网脚本, 扩容使用
├── start.sh            #启动脚本
├── stop.sh             #停止脚本
├── systemcontract      #nodejs系统合约工具
├── tool                #nodejs工具
├── unregister.sh       #从节点管理合约删除某个节点
├── web3lib             #nodejs基础库
└── web3sdk             #web3sdk环境
```

说明:

- nodeIDX

节点IDX的目录, 示例中每台服务器启动两个节点, 所以有node0, node1两个目录

nodeIDX的目录结构如下:

```
build/node0/
├── check.sh      #检查当前节点是否启动
├── config.json   #配置文件
├── data          #数据目录
├── genesis.json #创世块文件
├── keystore
├── log           #log日志目录
├── log.conf      #log配置文件
├── start.sh      #启动当前节点
└── stop.sh       #停止当前节点
```

- node.sh

记录nodejs相关依赖的环境变量

- start.sh

节点启动脚本, 使用方式:

```
./start.sh        启动所有的节点
或者
./start.sh IDX    启动指定的节点, IDX为节点的索引, 从0开始, 比如: start.sh 0表示启动第0个节点
```

- stop.sh 节点停止脚本, 使用方式:

```
./stop.sh        停止所有的节点
或者
./stop.sh IDX    停止指定的节点, IDX为节点的索引, 从0开始, 比如: stop.sh 0表示停止第0个节点
```

- register.sh 注册指定节点信息到节点管理合约, 扩容时使用

```
./register.sh IDX
```

- unregister.sh 将指定节点从节点管理合约中删除

```
./unregister.sh IDX
```

- node_manager.sh 查看当前节点管理合约中的节点信息

```
./node_manager.sh all
```

[web3sdk使用说明链接]

[web3lib、systemcontract、tool目录作用参考用户手册]

5.2.6 启动节点

在build目录执行start.sh脚本.**注意:要先启动创世块节点所在的服务器上的节点!!!**

```
$ ./start.sh
start all node ...
start node0 ...
start node1 ...
check all node status ...
node is running.
node is running.
```

5.2.7 验证

- **一定要所有服务器上的节点正常启动之后.**

日志

```
tail -f node0/log/log_*.log  | egrep "Generating seal"
INFO|2018-08-03 14:16:42:588|++++++++++++++++++++++++++ Generating seal␣
↪on8e5add00c337398ac5e9058432037aa646c20fb0d1d0fb7ddb4c6092c9d654fe#1tx:0,
↪maxtx:1000,tq.num=0time:1522736202588
INFO|2018-08-03 14:16:43:595|++++++++++++++++++++++++++ Generating seal␣
↪ona98781aaa737b483c0eb24e845d7f352a943b9a5de77491c0cb6fd212c2fa7a4#1tx:0,
↪maxtx:1000,tq.num=0time:1522736203595
```

可看到周期性的出现上面的日志，表示节点间在周期性的进行共识，整个链正常。

部署合约

每个服务器执行install_node时, 都会在安装目录下安装nodejs、babel-node、ethconsole, 其环境变量会写入当前安装用户的.bashrc文件，使用这些工具之前需要使环境变量生效，有两种使环境变量生效的方式，选择其中一种即可：

- 方式1：退出当前登录, 重新登录一次
- 方式2：执行node.sh脚本中的内容, 首先cat node.sh, 将显示的内容执行一遍

```
$ cat node.sh
export NODE_HOME=/root/octo/fisco-bcos/build/nodejs* export PATH=$PATH:$NODE_HOME/
↪bin* export NODE_PATH=$NODE_HOME/lib/node_modules:$NODE_HOME/lib/node_modules/
↪ethereum-console/node_modules;
$ export NODE_HOME=/root/octo/fisco-bcos/build/nodejs* export PATH=$PATH:$NODE_
↪HOME/bin* export NODE_PATH=$NODE_HOME/lib/node_modules:$NODE_HOME/lib/node_
↪modules/ethereum-console/node_modules;
```

部署合约验证, 进入tool目录：

```
$ cd tool
$ babel-node deploy.js HelloWorld
RPC=http://127.0.0.1:8546
Ouputpath=./output/
deploy.js  .......................Start.....................
Soc File :HelloWorld
HelloWorldcomplie success!
send transaction success:␣
↪0xfb6237b0dab940e697e0d3a4d25dcbfd68a8e164e0897651fe4da6a83d180ccd
HelloWorldcontract address 0x61dba250334e0fd5804c71e7cbe79eabecef8abe
HelloWorld deploy success!
cns add operation => cns_name = HelloWorld
         cns_name =>HelloWorld
         contract =>HelloWorld
         version  =>
         address  =>0x61dba250334e0fd5804c71e7cbe79eabecef8abe
         abi      =>[{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"type":"function"},{"constant":true,
↪"inputs":[],"name":"get","outputs":[{"name":"","type":"string"}],"payable":false,
↪"type":"function"},{"inputs":[],"payable":false,"type":"constructor"}]
send transaction success:␣
↪0x769e4ea7742b451e33cbb0d2a7d3126af8f277a52137624b3d4ae41681d58687
```

合约部署成功。

5.2.8 问题排查

参考附录FAQ。

5.3 扩容节点

- **扩容流程与搭链流程最本质的差别是, 初次搭链时会生成一个temp节点, 进行系统合约的部署, 然后会将所有构建的节点信息都注册入节点管理合约, 最后temp节点导出生成genesis.json文件. 所以搭链结束后, 每个节点信息都已经在节点管理合约, 但是在扩容时, 需要自己注册扩容的节点到节点管理合约。(参考FISCO-BCOS系统合约介绍[节点管理合约])。**

5.3.1 使用场景

对已经在运行的区块链, 可以提供其创世节点的相关文件, 创建出非创世节点, 使其可以连接到这条区块链。

5.3.2 获取扩容文件

- 从创世节点所在的服务器获取dependencies/follow/文件夹。

该目录包含构建区块链时分配的链根证书、机构证书、创世块文件、系统合约。

将follow放入/fisco-package-build-tool/目录。

5.3.3 配置

扩容的机器为: 172.20.245.45, 172.20.245.46 分别需要启动两个节点, 机构名称分别为agent_3、agent_4, 修改节点信息配置。

```
vim config.ini
```

修改扩容参数

```
* 扩容使用的一些参数
[expand]
genesis_follow_dir=/fisco-package-build-tool/follow/
```

修改节点列表为扩容的节点信息。

```
[nodes]
node0=172.20.245.45  0.0.0.0  2  agent_3
node1=172.20.245.46  0.0.0.0  2  agent_4
```

5.3.4 扩容

```
如果是在物料包生成服务器下运行扩容命令，需要删除已经存在的build目录
./generate_installation_packages.sh expand
```

成功之后会输出`Expanding end!`并在build目录生成对应安装包

```
build
├── 172.20.245.45_with_0.0.0.0_installation_package
├── 172.20.245.46_with_0.0.0.0_installation_package
└── stderr.log
```

5.3.5 安装启动

将安装包分别上传至对应服务器, 分别在每台服务器上面执行下列命令：

- 执行安装

```
./install_node.sh
```

- 启动节点

```
cd build
./start.sh
```

5.3.6 节点入网

在扩容时, 当前运行的链已经有数据, 当前新添加扩容的节点首先要进行数据同步, 建议新添加的节点在数据同步完成之后再将节点入网. 数据是否同步完成可以查看新添加节点的块高是否跟其他节点已经一致。

以172.20.245.45这台服务器为例进行操作, 172.20.245.46操作类似：

确保节点先启动。

用户在build目录下，使用register.sh脚本进行注册操作

注册

```
$ ./register.sh 0  # 注册第一个节点
$ ./register.sh 1  # 注册第二个节点
```

验证,注册的节点是否正常:

```
$ tail -f node0/log/log_*.log   | egrep "Generating seal"
INFO|2018-07-10 10:49:29:818|++++++++++++++++++++++++++ Generating seal␣
↪oncf8e56468bab78ae807b392a6f75e881075e5c5fc034cec207c1d1fe96ce79a1#4tx:0,
↪maxtx:1000,tq.num=0time:1531190969818
INFO|2018-07-10 10:49:35:863|++++++++++++++++++++++++++ Generating seal␣
↪one23f1af0174daa4c4353d00266aa31a8fcb58d3e7fbc17915d95748a3a77c540#4tx:0,
↪maxtx:1000,tq.num=0time:1531190975863
INFO|2018-07-10 10:49:41:914|++++++++++++++++++++++++++ Generating seal␣
↪on2448f00f295210c07b25090b70f0b610e3b8303fe0a6ec0f8939656c25309b2f#4tx:0,
↪maxtx:1000,tq.num=0time:1531190981914
INFO|2018-07-10 1
```

5.3.7 问题排查

参考附录FAQ。

5.4 部署区块链sample

这里提供一个非常简单的例子, 用来示例如何使用物料包以最快的速度搭建一条在单台服务器上运行4个节点的FISCO BCOS的测试环境。

5.4.1 下载物料包

```
$ git clone https://github.com/FISCO-BCOS/fisco-package-build-tool.git
```

5.4.2 生成安装包

```
$ cd fisco-package-build-tool
$ ./generate_installation_packages.sh build
......
//中间会有FISCO-BCOS下载、编译、安装, 时间会比较久, 执行成功最终在当前目录下会生成build目录。
......
```

查看生成的build目录结构

```
$ tree -L 1 build
build/
├── 127.0.0.1_with_0.0.0.0_genesis_installation_package
├── stderr.log
└── temp
```

其中 127.0.0.1_with_0.0.0.0_genesis_installation_package 即是生成的安装包.

5.4.3 安装

假定需要将FISCO BCOS安装在当前用户home目录下, 安装的目录名fisco-bcos。

```
$ mv build/127.0.0.1_with_0.0.0.0_genesis_installation_package ~/fisco-bcos
$ cd ~/fisco-bcos
$ ./install_node.sh
..........
执行成功会生成build目录
```

5.4.4 启动

```
$ cd build
$ ./start.sh
start node0 ...
start node1 ...
start node2 ...
start node3 ...
check all node status =>
node0 is running.
node1 is running.
node2 is running.
node3 is running.
```

5.4.5 验证

```
$ tail -f node0/log/log_2018080116.log | egrep "seal"
INFO|2018-08-01 16:52:18:362|++++++++++++++++++++++++++ Generating seal␣
↪on5b14215cff11d4b8624246de63bda850bcdead20e193b24889a5dff0d0e8a3c3#1tx:0,
↪maxtx:1000,tq.num=0time:1533113538362
INFO|2018-08-01 16:52:22:432|++++++++++++++++++++++++++ Generating seal␣
↪on5e7589906bcbd846c03f5c6e806cced56f0a17526fb1e0c545b855b0f7722e14#1tx:0,
↪maxtx:1000,tq.num=0time:1533113542432
```

5.4.6 部署成功

验证之后，一条简单的测试链搭建成功。

5.4.7 问题排查

参考附录FAQ。

5.5 FISCO BCOS docker

5.5.1 构建docker运行环境

说明

使用物料包同样可以构建基于docker节点的区块链的搭建, 使用方式与构建普通节点的区块链比较类似, 区别在于最终启动的docker节点。

配置

```
[docker]
* 当前是否构建docker节点的安装包. 0: 否    1: 是
docker_toggle=1
* docker仓库地址
docker_repository=fiscoorg/fisco-octo
* docker版本号.
docker_version=v1.3.x-latest
```

- docker_toggle

构建docker节点环境时, docker_toggle设置为1。

- docker_repository

docker镜像仓库, 用户一般不需要修改。

- docker_version

docker版本号, 使用默认值即可, 版本更新时, 本地可以自动更新至最新版本。

配置节点信息

同样以在三台服务器上部署区块链为例：

服务器ip： 172.20.245.42 172.20.245.43 172.20.245.44机构分别为： agent_0 agent_1 agent_2节点数目：每台服务器搭建两个节点 修改[nodes] section字段为：

```
[nodes]
node0=172.20.245.42  0.0.0.0  2  agent_0
node1=172.20.245.43  0.0.0.0  2  agent_1
node2=172.20.245.44  0.0.0.0  2  agent_2
```

构建安装包

执行成功会输出`Building end!`并生成build目录

```
build/
├── 172.20.245.42_agent_0_genesis
├── 172.20.245.43_agent_1
├── 172.20.245.44_agent_2
├── stderr.log
└── temp
```

将安装包上传到对应的服务器。

安装

进入安装目录, 执行`./install_node`, 成功之后会生成 **docker** 目录

```
docker/
├── node0
├── node1
├── register0.sh
├── register1.sh
├── start0.sh
├── start1.sh
├── start.sh
├── stop0.sh
├── stop1.sh
├── stop.sh
├── unregister0.sh
└── unregister1.sh
```

- nodeIDX : 第IDX个节点的目录, 该目录会被映射到docker的/fisco-bcos/node/目录, 这两个目录中的内容是一致的
- nodeIDX/log : 日志目录
- start.sh 启动所有的docker节点
- stop.sh 停止所有的docker节点

- startIDX.sh 启动第IDX个docker节点
- stopIDX.sh 停止第IDX个节点
- registerIDX.sh 扩容时使用, 将第IDX个节点注册入节点管理合约, 调用的是docker中的node_manager.sh脚本, 扩容时使用
- unregisterIDX.sh 将IDX个节点从节点管理合约删除, 调用的是node_manager.sh脚本

启动

在docker目录执行start.sh脚本

注意:要先启动创世块节点所在的服务器上的节点!!!

```
 ./start.sh
start node0 ...
705b6c0e380029019a26e954e72da3748e29cec95a508bc1a8365abcfc36b86c
start node1 ...
be8bd964322a08a70f22be9ba15082dbe50d1729f955291586a0503e32d2225f
```

验证

- docker ps

通过docker ps命令查看docker节点是否启动正常

```
docker ps  -f name="fisco*"
CONTAINER ID        IMAGE                                  COMMAND                  ␣
↪CREATED             STATUS              PORTS               NAMES
be8bd964322a        fiscoorg/fisco-octo:v1.3.x-latest   "/fisco-bcos/start_n..."  ␣
↪21 minutes ago      Up 21 minutes                           fisco-node1_8546
705b6c0e3800        fiscoorg/fisco-octo:v1.3.x-latest   "/fisco-bcos/start_n..."  ␣
↪22 minutes ago      Up 22 minutes                           fisco-node0_8545
```

启动的两个docker节点的容器id分别为: be8bd964322a、705b6c0e3800, docker节点的STATUS状态为Up, 说明节点正常启动。

- 日志验证

```
tail -f node0/log/log_*.log | egrep "seal"
INFO|2018-08-13 11:52:38:534|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪onec61bbf7cb6152d523f391dfe65dd1f858ec3daa7b6df697308a0ad5219cf232#1tx:0,
↪maxtx:1000,tq.num=0time:1534161158534
INFO|2018-08-13 11:52:40:550|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪on127962f94ccb075a448ae741e69718ffc0bee4f97ccddb7bd5e8a0310f4b8980#1tx:0,
↪maxtx:1000,tq.num=0time:1534161160550
INFO|2018-08-13 11:52:42:564|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪on29ccca512d7e2bac34760e8c17807896dac914b426884a0bc28499a556811467#1tx:0,
↪maxtx:1000,tq.num=0time:1534161162564
```

说明节点周期性共识, 出块。

- 进入docker容器部署合约。

docker运行之后, docker镜像内部是一个完整的fisco-bcos的运行环境, 包括js的工具都是可用的, 可以进入docker镜像内部进行合约的部署。

以be8bd964322a为例

```
$ sudo docker exec -it be8bd964322a /bin/bash
```

- 加载环境变量

执行 `source /etc/profile`加载docker内部的一些环境变量。

- 目录

```
$ cd /fisco-bcos
```

目录结构如下：

```
/fisco-bcos/
├── node
│   ├── config.json
│   ├── fisco-data
│   ├── genesis.json
│   ├── log
│   └── start.sh
├── tool
├── systemcontract
├── web3sdk
├── nodejs
├── web3lib
└── nodejs
```

- /fisco-bcos/node : 节点目录
- /fisco-bcos/node/fisco-data : 数据目录
- /fisco-bcos/node/log : 日志目录
- /fisco-bcos/systemcontract : nodejs系统合约工具
- /fisc-bcos/tool : nodejs工具
- /fisco-bcos/web3lib : nodejs基础库
- /fisco-bcos/web3sdk : web3sdk环境
- 部署合约

进入tool目录, 部署合约

```
$ cd /fisco-bcos/tool
$ babel-node deploy.js HelloWorld
RPC=http://0.0.0.0:8546
Ouputpath=./output/
deploy.js  .......................Start.......................
Soc File :HelloWorld
HelloWorldcomplie success!
send transaction success:␣
↪0x726e328e5b53ddb3ce040424304ffd61e9ae277d6441068c45ad590003c7426a
HelloWorldcontract address 0x4437f8c9cd1e6a3e8ec9c3460c4bc209acdca052
HelloWorld deploy success!
cns add operation => cns_name = HelloWorld
         cns_name =>HelloWorld
         contract =>HelloWorld
         version  =>
         address  =>0x4437f8c9cd1e6a3e8ec9c3460c4bc209acdca052
         abi      =>[{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"type":"function"},{"constant":true,
↪"inputs":[],"name":"get","outputs":[{"name":"","type":"string"}],"payable":false,
↪"type":"function"},{"inputs":[],"payable":false,"type":"constructor"}]
send transaction success:␣
↪0xa280d823346e1b7ea332a2b4d7a7277ae380b0cc7372bef396c5205fa74b25ae
```

扩容

与普通节点的扩容流程类似, 大体来说, 构建扩容安装包, 启动节点, 将扩容的节点入网(加入节点管理合约)。

- 构建扩容安装包

参考 扩容流程 构建扩容机器上的安装包。

- 安装启动

将扩容服务器上传对应的服务器, 进入安装目录, 执行`./install_node`

执行成功之后生成**docker**目录

启动docker节点

```
$ cd docker
$ ./start.sh
```

- 节点入网

docker目录下`registerIDX.sh`脚本为注册脚本。

```
$ ./register0.sh
======================================================================
=====INIT ECDSA KEYPAIR From private key===
node.json=file:/fisco-bcos/node/fisco-data/node.json
$ ./register1.sh
======================================================================
=====INIT ECDSA KEYPAIR From private key===
node.json=file:/fisco-bcos/node/fisco-data/node.json
```

- 验证

查看 log

```
 tail -f node0/log/log*.log | egrep "seal"
INFO|2018-08-13 12:00:18:710|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪on26c826ad8d275cd2c3c53a034818acce222ad7dc8ef455de64efbf193748c9ef#1tx:0,
↪maxtx:1000,tq.num=0time:1534161618710
INFO|2018-08-13 13:00:02:040|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪onb7a72e68cbc43293dac635b3c868e83ecbe24c3f1b72e0d57a809ee72bad9ca5#4tx:0,maxtx:0,
↪tq.num=0time:1534165202040
INFO|2018-08-13 13:54:25:054|PBFTClient.cpp:343|++++++++++++++++++++++++++␣
↪Generating seal␣
↪one2d93621481bf613b065f254519bbc32689d3b2eb8c5a1680c0f4d57531f7ef5#5tx:0,maxtx:0,
↪tq.num=0time:1534168465054
```

5.5.2 私钥证书管理

- 物料包使用FISCO BCOS的工具分配证书, 工具位于下载的FSICO-BCOS目录的cert子目录, 使用方式参考FISCO-BCOS 证书生成工具。
- 构建完成各个服务器的安装包之后, 整条链的根证书、机构证书会保存在创世节点所在服务器的dependencies/cert目录 保存的目录结构为:

```
cert目录
    ca.crt
    ca.key
    机构名称子目录
```

(continues on next page)

(continued from previous page)

```
        agency.crt
        agency.key
```

- 以上面示例的配置为例, 创世节点服务器cert目录内容(详见使用指南中证书说明):

```
ca.crt
ca.key
agent_0\
    agency.crt
    agency.key
agent_1\
    agency.crt
    agency.key
agent_2\
    agency.crt
    agency.key
```

- ca.crt 链证书
- ca.key 链证书私钥
- agent_0\agency.crt agent_0机构证书
- agent_0\agency.key agent_0机构证书私钥
- agent_1\agency.crt agent_1机构证书
- agent_1\agency.key agent_1机构证书私钥
- agent_2\agency.crt agent_2机构证书
- agent_2\agency.key agent_2机构证书私钥

5.6 JAVA安装

FISCO BCOS中需要使用Oracle JDK 1.8 (java 1.8) 环境, 在CentOS/Ubuntu中默认安装或者通过yum/apt安装的JDK均为openJDK, 并不符合使用的要求, 本文是一份简单的Oracle Java 1.8的安装教程.

假设安装目录为/usr/local/

5.6.1 下载.

[JAVA 1.8下载连接]选择**jdk-8u181-linux-x64.tar.gz**下载.

5.6.2 上传解压.

将下载的jdk-8u181-linux-x64.tar.gz上传至服务器, 放入/usr/local/目录, 然后解压.

```
$ tar -zxvf jdk-8u181-linux-x64.tar.gz
$ cd jdk1.8.0_181 && pwd
$ /usr/local/jdk1.8.0_181
```

5.6.3 配置环境变量

- 全局安装, 所有用户均生效将下面的内容添加入 /etc/profile 文件的末尾.

```
vim /etc/profile
```

```
export JAVA_HOME=/usr/local/jdk1.8.0_181
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

添加完毕之后执行:

```
$ source /etc/profile
```

- 用户环境变量配置, 当前用户有效将下面的内容添加到 ~/.bash_profile 文件的末尾。 若无法修改 ~/.bash_profile 文件，则将下面的内容添加到 ~/.bashrc 文件的末尾。

```
vim ~/.bash_profile
或  vim ~/.bashrc
```

```
export JAVA_HOME=/usr/local/jdk1.8.0_181
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

添加完毕之后执行:

```
$ source ~/.bash_profile
若添加的内容到~/.bashrc文件，则执行 $ source ~/.bashrc
```

5.6.4 检查java版本

```
$ java -version
$ java version "1.8.0_181"
$ Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
$ Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
```

安装成功！

5.7 物料包web3sdk配置

请注意，物料包内置了配置好的web3sdk以及相关的环境，用户可以直接使用web3sdk。请注意，由于物料包已经生成好了链证书和机构证书，因此物料包中的web3sdk的证书配置与源码编译略有不同。

假设用户拉取web3sdk的操作git clone https://github.com/FISCO-BCOS/web3sdk 是在/mydata下进行的，则用户执行：

则用户在物料包build/web3sdk目录下，拷贝相应证书至需要自行配置的web3sdk文件夹下。

```
$ cd conf
$ cp sdk.* ca.crt client.keystore /mydata/web3sdk/dist/conf/
```

5.7.1 物料包应用开发指南

物料包内置了配置好的web3sdk，用户可以直接进入配置好的服务器下的build目录，进入web3sdk目录进行应用开发，应用开发指南

5.7.2 物料包web3sdk配置

如果用户想要在物料包下配置web3sdk，请参考web3sdk配置部分

请注意，由于物料包已经生成好了链证书和机构证书，物料包中配置文件-配置java客户端过程和文档中略有不同。

用户可以直接从编译源码部分开始配置。

修改web3sdk/dist/conf目录的applicationContext.xml文件所需要节点的config.json在物料包目录下的build/node* 下 *为希望选择的节点数 config.json中包含systemproxyaddress 系统合约地址 rpcport, p2pport,channelPort。

用户根据build/node*/config.json 配置java客户端相关数据：需要更改的有数据有：系统合约地址systemproxyaddress，node port相关信息等

```
$ vim applicationContext.xml
```

具体操作参考web3sdk-配置文件

```
如 </value>node0@0.0.0.0:8841</value>
```

修改完成后，运行

```
$ java -cp 'conf/:apps/*:lib/*' org.bcos.channel.test.TestOk
```

测试是否配置成功

5.8 CheckList

通常我们推荐使用物料包[FISCO BCOS物料包]搭建FISCO BCOS的环境, 可以屏蔽搭建过程中的一些繁琐细节。

物料包使用时, 本身即有一些依赖, FISCO BCOS对网络、yum源等外部环境也存在依赖, 为减少搭建过程中遇到的问题,建议在使用之前对整个搭建的环境进行检查, 特别是生产环境的搭建, 尤其推荐CheckList作为一个必备的流程。

5.8.1 检查项

- 操作系统
- 网络
- java环境
- openssl版本
- yum/apt源检查

操作系统

```
支持操作系统:
CentOS 7.2 64位 、 Ubuntu 16.04 64位
```

- 检查系统是否为64位系统:

使用**uname -m**命令, 64位系统的输出为x86_64, 32位系统的输出为i386或者i686.

```
$ uname -m
$ x86_64
```

- 操作系统版本检查:

```
CentOS
$ cat /etc/redhat-release
$ CentOS Linux release 7.2.1511 (Core)

Ubuntu
$ cat /etc/os-release
$ NAME="Ubuntu"
$ VERSION="16.04.1 LTS (Xenial Xerus)"
$ ID=ubuntu
$ ID_LIKE=debian
$ PRETTY_NAME="Ubuntu 16.04.1 LTS"
$ VERSION_ID="16.04"
$ HOME_URL="http://www.ubuntu.com/"
$ SUPPORT_URL="http://help.ubuntu.com/"
$ BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

网络

FISCO BCOS单节点需要使用三个端口： rpc_port、channel_port、p2p_port

- rpc_port不会有远程访问
- channel_port需要被使用web3sdk的服务访问
- p2p_port 节点之间通过互联组成p2p网络

实际中, 需要考虑channel_port、p2p_port的网络访问策略, 节点的channel_port需要被使用区块链服务的应用所在服务器连接, 每个节点的p2p_port需要能被其他节点所在服务器的连接。

检查服务器A某一个端口p能够被另一台服务器B访问的简单方法:

- 在服务器A上执行

```
sudo nc -l p    //实际检查时, 将p替换为实际端口。
```

- 在服务器B上面执行telnet命令

```
$ telnet A p  //实际检查时, 将A替换服务器ip, 将p替换为实际端口。
Trying A...
Connected to A.
Escape character is '^]'.
```

上面的结果说明成功, 服务器B确实可以访问服务器A的端口p。

- 网络不通, 通常需要运维工程师协助解决。

java环境

版本检查

FISCO BCOS需求版本Oracle JDK 1.8(java 1.8)

- [&] CentOS/Ubuntu默认安装或者通过yum/apt安装的JDK为openJDK, 并不符合使用的要求。
- [&] 可以通过java -version查看版本, Oracle JDK输出包含"Java(TM) SE"字样, OpenJDK输出包含"OpenJDK"的字样, 很容易区分。

```
Oracle JDK 输出:
$ java -version
$ java version "1.8.0_144"
```

(continues on next page)

(continued from previous page)

```
$ Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
$ Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)

OpenJDK 输出:
$ java -version
$ openjdk version "1.8.0_171"
$ OpenJDK Runtime Environment (build 1.8.0_171-b10)
$ OpenJDK 64-Bit Server VM (build 25.171-b10, mixed mode)
```

Oracle JDK安装

当前系统如果没有安装JDK, 或者JDK的版本不符合预期, 可以参考[Oracle JAVA 1.8 安装教程]。

openssl版本

openssl需求版本为1.0.2, 可以使用 openssl version 查看。

```
$ openssl version
$ OpenSSL 1.0.2k-fips  26 Jan 2017
```

服务器如果没有安装openssl, 可以使用yum/apt进行安装。

```
sudo yum/apt -y install openssl
```

yum/apt不存在openssl, 可以参考下面的替换apt/yum源。

yum/apt源检查

物料包工作过程中会使用yum/apt安装一些依赖项, 当前yum/apt源无法下载到相关依赖时, 工作工程中可能会出现一些问题(fisco-bcos-package-tool内部已经做了相关处理, 在异常执行时给用户提示, 并停止工作, 但实际环境更加复杂, 不排除有遗漏)。

对此建议可以提前检查yum/apt源。

检查列表

在服务器上面依次执行下面命令:

```
CentOS 依赖
        sudo yum -y install bc
        sudo yum -y install gettext
        sudo yum -y install cmake3
        sudo yum -y install git
        sudo yum -y install gcc-c++
        sudo yum -y install openssl
        sudo yum -y install openssl-devel
        sudo yum -y install leveldb-devel
        sudo yum -y install curl-devel
        sudo yum -y install libmicrohttpd-devel
        sudo yum -y install gmp-devel
        sudo yum -y install lsof
        sudo yum -y install crudini
        sudo yum -y install libuuid-devel

Ubuntu 依赖
```

(continues on next page)

(continued from previous page)

```
        sudo apt-get -y install gettext
        sudo apt-get -y install bc
        sudo apt-get -y install cmake
        sudo apt-get -y install git
        sudo apt-get -y install gcc-c++
        sudo apt-get -y install openssl
        sudo apt-get -y install build-essential libboost-all-dev
        sudo apt-get -y install libcurl4-openssl-dev libgmp-dev
        sudo apt-get -y install libleveldb-dev  libmicrohttpd-dev
        sudo apt-get -y install libminiupnpc-dev
        sudo apt-get -y install libssl-dev libkrb5-dev
        sudo apt-get -y install lsof
        sudo apt-get -y install crudini
        sudo apt-get -y install uuid-dev
```

如果apt/yum安装某些项失败, 说明apt/yum源不存在该依赖项。

替换yum/apt源

yum/apt源如果不满足要求, 可以考虑将源替换为阿里云的源。

- CentOS更换阿里云yum源

```
1. 备份
mv /etc/yum.repos.d/CentOS-Base.repo /etc/yum.repos.d/CentOS-Base-bak.repo
2. 下载
wget -O /etc/yum.repos.d/CentOS-Base.repo http://mirrors.aliyun.com/repo/Centos-7.
↪repo
3. 更新yum缓存
yum makecache
```

- Ubuntu 16.04更换阿里云源

```
1. 备份
sudo cp /etc/apt/sources.list /etc/apt/sources.list.old
2. 修改source.list
sudo vim /etc/apt/source.list
添加以下信息
# deb cdrom:[Ubuntu 16.04 LTS _Xenial Xerus_ - Release amd64 (20160420.1)]/ xenial␣
↪main restricted
deb-src http://archive.ubuntu.com/ubuntu xenial main restricted #Added by software-
↪properties
deb http://mirrors.aliyun.com/ubuntu/ xenial main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial main restricted multiverse␣
↪universe #Added by software-properties
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-updates main restricted␣
↪multiverse universe #Added by software-properties
deb http://mirrors.aliyun.com/ubuntu/ xenial universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates universe
deb http://mirrors.aliyun.com/ubuntu/ xenial multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-updates multiverse
deb http://mirrors.aliyun.com/ubuntu/ xenial-backports main restricted universe␣
↪multiverse
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-backports main restricted␣
↪universe multiverse #Added by software-properties
deb http://archive.canonical.com/ubuntu xenial partner
deb-src http://archive.canonical.com/ubuntu xenial partner
deb http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted
deb-src http://mirrors.aliyun.com/ubuntu/ xenial-security main restricted␣
↪multiverse universe #Added by software-properties
```

(continues on next page)

(continued from previous page)

```
deb http://mirrors.aliyun.com/ubuntu/ xenial-security universe
deb http://mirrors.aliyun.com/ubuntu/ xenial-security multiverse

3. 更新apt缓存
sudo apt-get update
```

5.9 日志清理脚本rmlogs.sh

5.9.1 介绍

使用物料包(**fisco-package-build-tool**)搭建的fisco-bcos环境, 最终在生成的build目录会有个rmlogs.sh脚本。 可以用来删除fisco-bcos生成的日志文件。

5.9.2 使用

```
bash rmlogs.sh
或者
bash rmlog.sh 需要保留日志的天数
```

rmlog.sh默认会删除三天之前生成的所有的日志文件, 可以通过传入需要保留的日志的天数来修改这个规则。

5.9.3 crontab配置

建议将rmlogs.sh配置到crontab中定期执行, 防止fisco-bcos打印过多的日志导致磁盘爆满引起其他的各种问题。 crontab设置可以参考如下, 每天执行一次。

```
0 0 * * * /data/app/fisco-bcos/build/rmlogs.sh >> /data/app/fisco-bcos/build/
↪rmlogs.log 2>&1
```

[toc]

5.10 节点监控脚本monitor.sh

5.10.1 介绍

使用物料包(**fisco-package-build-tool**)搭建的fisco-bcos环境, 最终在生成的build目录会有个monitor.sh脚本.该脚本可以用来监控节点是否正常启动或者节点所在的区块链是否正常工作, 在节点挂掉或者整个区块链无法正常工作情况下重启节点.

5.10.2 使用

monitor.sh脚本可以直接执行.

```
./monitor.sh
[2018-10-10 15:28:11]{"id":67,"jsonrpc":"2.0","result":"0x3"}
[2018-10-10 15:28:11]{"id":68,"jsonrpc":"2.0","result":"0x2ca"}
[2018-10-10 15:28:11]OK! 0.0.0.0:8545 is working properly: height 0x3 view 0x2ca
```

提示

`OK! $config_ip:$config_port is working properly: height $height view $view` 节点正常工作.

`ERROR! $config_ip:$config_port does not exist` 节点进程不存在, 节点可能已经宕机, 会自动拉起节点.

`ERROR! Cannot connect to $config_ip:$config_port` RPC请求失败, 节点可能已经宕机, 会自动拉起节点.

`ERROR! $config_ip:$config_port is not working properly: height $height and view $view no change` 节点块高、视图都没有变化, 整条链可能无法正常工作, 会自动重启节点.

5.10.3 配置crontab

建议将monitor.sh添加到crontab中，设置为每分钟执行一次，并将输出重定向到日志文件。可以日常扫描日志中的`ERROR!`字段就能找出节点服务异常的时段, 也可以在节点挂掉情况下及时将节点重启。在crontab的配置可以参考如下：

```
*/1  * * * * /data/app/fisco-bcos/build/monitor.sh >> /data/app/fisco-bcos/build/
↪monitor.log 2>&1
```

用户在实际中使用时将monitor.sh、monitor.log的路径修改即可。

5.11 FAQ

5.11.1 generate_installation_packages.sh build/expand 报错提示

- ERROR - build directory already exist, please remove it first.

fisco-package-build-tool目录下已经存在build目录, 可以将build目录删除再执行。

- ERROR - no sudo permission, please add youself in the sudoers.

当前登录的用户需要有sudo权限。

- ERROR - Unsupported or unidentified Linux distro.

当前linux系统不支持, 目前FISCO-BCOS支持CentOS 7.2+、Ubuntu 16.04。

- ERROR -Unsupported or unidentified operating system.

当前系统不支持, 目前FISCO-BCOS支持CentOS 7.2+、Ubuntu 16.04。

- ERROR - Unsupported Ubuntu Version. At least 16.04 is required.

当前ubuntu版本不支持, 目前ubuntu版本仅支持ubuntu 16.04 64位操作系统。

- ERROR - Unsupported CentOS Version. At least 7.2 is required.

当前CentOS系统不支持, 目前CentOS支持7.2+ 64位。

- ERROR - Unsupported Oracle Linux, At least 7.4 Oracle Linux is required.

当前CentOS系统不支持，目前CentOS支持7.2+ 64位。

- ERROR - unable to determine CentOS Version

当前CentOS系统不支持, 目前CentOS支持7.2+ 64位。

- ERROR - Unsupported Linux distribution.

不支持的linux系统.目前FISCO-BCOS支持CentOS 7.2+、Ubuntu 16.04。

- ERROR - Oracle JDK 1.8 be requied

需要安装Oracle JDK 1.8。目前物料包的web3sdk仅支持Oracle JDK1.8版本，尚未支持其他的java版本，请参考物料包java安装

- ERROR - OpenSSL 1.0.2 be requied

物料包需要openssl 1.0.2版本，请升级openssl版本。

- ERROR - failed to get openssl version

无法获取openssl的版本，请尝试从新安装openssl。

- ERROR - XXX is not installed.

XXX没有安装，请尝试安装该依赖。

- ERROR - FISCO BCOS gm version not support yet.

物料包不支持国密版本的FISCO BCOS的安装。

- ERROR - At least FISCO-BCOS 1.3.0 is required.

物料包工具支持的FISCO BCOS的版本最低为v1.3.0。

- ERROR - Required version is xxx, now fisco bcos version is xxxx"

当前fisco-bcos版本与配置的版本不一致, 建议手动编译自己需要的版本。

- ERROR - node.nodeid is not exist.
- ERROR - ca.crt is not exist
- ERROR - maybe bash *.sh

无法为当前node生成证书，请检查证书文件是否完好。

- ERROR - temp node rpc port check, *** is in use
- ERROR - temp node channel port check, *** is in use
- ERROR - temp node p2p port check, *** is in use

rpc/channel/p2p port被占用，请尝试修改端口号从新进行安装。

- ERROR - temp node rpc port check, XXX is in use.

temp节点使用的rpc端口被占用, 可以netstat -anp | egrep XXX查看占用的进程。

- ERROR - temp node channel port check, XXX is in use.

temp节点使用的channel端口被占用, 可以netstat -anp | egrep XXX查看占用的进程。

- ERROR - temp node p2p port check, XXX is in use.

temp节点使用的p2p端口被占用, 可以netstat -anp | egrep XXX查看占用的进程。

- ERROR - git clone FISCO-BCOS failed.

下载FISCO-BCOS源码失败, 建议手动下载。

- ERROR - system contract address file is not exist, web3sdk deploy system contract not success.

temp节点部署系统合约失败。

- ERROR - Oracle JDK 1.8 be requied, now JDK is `java -version`.
- ERROR - there has no node on this server.

当前服务器没有任何节点，请至少为一台服务器至少配置一个节点。

- ERROR - fisco-bcos –newaccount opr faild.

god账户配置失败，请从新尝试安装。

- ERROR - channel port is not listening.

部署系统合约失败，请更新物料包版本。

- ERROR - system contract address file is not exist, web3sdk deploy system contract not success.

系统合约部署失败，请检查java jdk配置是否正确。

- ERROR - system contract address file is empty, web3sdk deploy system contract not success.

系统合约部署失败，请从新下载物料包。

5.11.2 docker 安装报错提示

- ERROR - docker is not installed

docker未安装, 用户需要自己安装docker程序, apt/yum install docker.

- ERROR - docker dictionary already exist, remove it first.

docker目录已存在，请移除当前docker节点安装包目录之后重试。

- ERROR - docker pull failed, docker service not start or repository? version info error,repository.

docker服务器启动失败，请检查docker配置、运行是否正常。

- ERROR - there is already fisco-bcos docker named fisco-node.

当前docker内已经有名称已存在的文件夹，请更改名称。

5.11.3 generate_installation_packages.sh build/expand 直接退出。

查看build/stderr.log内容, 查看错误信息。

5.11.4 start.sh 提示 ulimit: core file size: cannot modify limit: Operation not permitted

无法通过脚本修改core文件大小限制, 不影响节点的启动。

5.11.5 start.sh 显示nodeIDX is not running.

这个提示是说nodeIDX启动失败, 可以ps -aux | egrep fisco 查看是否正常启动. 可以执行`cat node/nodedirIDX/log/log`查看节点启动失败的原因。

常见的原因:

- libleveldb.so No such file or directory.

```
./fisco-bcos: error while loading shared libraries: libleveldb.so.1: cannot open␣
↪shared object file: No such file or directory
```

leveldb动态库缺失, 安装脚本里面默认使用 yum/apt 对依赖组件进行安装, 可能是 yum/apt 源缺失该组件。

可以使用下面命令手动安装leveldb, 若leveldb安装不成功可以尝试替换yum/apt的源。

```
[CentOS]sudo yum -y install leveldb-devel
[Ubuntu]sudo apt-get -y install libleveldb-dev
```

如果leveldb已经安装, 则可以尝试执行`sudo ldconfig`, 然后执行start.sh, 重新启动节点.

- FileError

```
terminate called after throwing an instance of 'boost::exception_detail::clone_impl
↪<dev::FileError>' what():  FileError
```

操作文件失败抛出异常, 原因可能是当前登录的用户没有安装包目录的权限, 可以通过ls -lt查看当前文件夹对应的user/group/other以及对应的权限, 一般可以将安装包的user改为当前用户或者切换登录用户为安装包的user用户可以解决问题。

当程序异常退出，程序提示dir exists时，需要进行以下操作

```
1、删除fisco-bcos文件
$ sudo rm -rf /usr/local/bin/fisco-bcos
2、删除build目录
$ rm -rf build
3、重新执行创建安装包的命令
$ ./generate_installation_packages.sh build
```

CHAPTER 6

国密版FISCO BCOS

Welcom to 国密版FISCO BCOS project :-)

为了充分支持国产密码学算法，金链盟基于 国产密码学标准,实现了国密加解密、签名、验签、哈希算法，并将其集成到FISCO BCOS平台中，实现了对 **国家密码局认定的商用密码** 的完全支持。

下表对比了 **国密版FISCO-BCOS** 和 **非国密版FISCO-BCOS** :

算法类型	国密版FISCO BCOS	非国密版FISCO BCO
签名算法	SM2 (公私钥长度：512 bits, 256 bits)	ECDSA (公私钥长度: 512 bits, 256 bits)
哈希算法	SM3 (哈希串长度: 256 bits)	SHA3 (哈希串长度: 256 bits)
对称加解密算法	SM4 (对称密钥长度: 128 bits)	AES (加密秘钥长度: 256 bits)

(注：国密算法SM2, SM3, SM4均基于 国产密码学标准 开发)

国密版FISCO-BCOS主要包括如下特性	• web3sdk和nodejs使用国密SM2对交易进行签名 • 节点间使用符合国密SSL_VPN标准的openssl库通信 • 节点可验证国密SM2算法签名后的交易 • 使用国密SM4算法对落盘数据进行加密
国密版FISCO BCOS不支持的特性	• 暂不支持控制台ethconsole发交易 • 暂不支持web3sdk与fisco-bcos节点通信采用国密SSL • 暂不支持兼容老数据，因此使用者请注意，国密版本只适 合在新业务中使用

6.1 环境要求

配置	最低配置	推荐配置
CPU	1.5GHz	2.4GHz
内存	1GB	4GB
核心	2核	4核
带宽	1Mb	5Mb
操作系统		CentOS （7.2 64位）或Ubuntu（16.04 64位）
JAVA		Java(TM) 1.8 && JDK 1.8

6.2 编译安装

6.2.1 拉取源码

安装依赖软件

centos系统安装如下依赖软件:

```
$ sudo yum -y install git dos2unix lsof
```

ubuntu系统安装如下软件:

```
$ sudo apt install git lsof tofrodos
$ ln -s /usr/bin/todos /usr/bin/unxi2dos && ln -s /usr/bin/fromdos /usr/bin/
↪dos2unix
```

拉取源码

```
# 进入源码存放目录(设位于~/mydata)
$ cd ~/mydata

# 从git拉取源码
$ git clone https://github.com/FISCO-BCOS/FISCO-BCOS
```

6.2.2 编译源码

编译国密版FISCO BCOS

安装依赖包(执行scripts/install_deps.sh脚本):

```
# 进入FISCO BCOS源码目录(设FISCO-BCOS源码位于/mydata目录)
$ cd /mydata/FISCO-BCOS

# 为了防止windows脚本上传到linux环境下引起的不兼容问题，使用dos2unix格式化所有脚本
$ dos2unix `find . -name "*.sh"`
```

编译国密版FISCO-BCOS(-DENCRYPTTYPE=ON):

```
# 进入源码目录(设位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS
```

(continues on next page)

(continued from previous page)

```
# 调用build.sh脚本编译国密版fisco-bcos
# 注：（当前用户需要有sudo权限，期间可能会多次输入密码）
# -g: 编译国密版FISCO-BCOS（国密链必须设置该选项）
$ bash build.sh -g

# 确认fisco-bcos是国密版本
$ fisco-bcos --version
FISCO-BCOS version 1.3.2-gm  # 有-gm，表明是国密版FISCO BCOS
FISCO-BCOS network protocol version: 63
Client database version: 12041
Build: ETH_BUILD_PLATFORM/ETH_BUILD_TYPE

# 注：若上次编译失败，本次继续编译时可能会报错，此时需要删掉源码目录下deps/src/目录中缓存包
后重新使用build.sh编译，一般包括如下命令：
# rm -rf deps/src/*-build
# rm -rf deps/src/*-stamp
```

6.3 证书生成

FISCO BCOS网络采用面向CA的准入机制，国密版FISCO BCOS要生成三级证书，包括：链证书，机构证书和节点证书。

6.3.1 生成链证书

FISCO BCOS提供generate_chain_cert.sh脚本生成链证书：

```
# 进入脚本所在目录（设FISCO-BCOS位于~/mydata目录）
cd ~/mydata/FISCO-BCOS/tools/scripts

# 在~/mydata目录下生成国密版FISCO BCOS链证书
#-----------------------------------------------------
#-o ：生成的链证书所在路径，这里是~/mydata
#-g ：生成国密链证书，这里必须设置
# 注：（若要手动输入链证书信息，请在下面命令最后加上-m选项）
#-----------------------------------------------------
$ bash ./generate_chain_cert.sh -o ~/mydata -g
# 此时/mydata目录下生成链证书私钥gmca.key、证书gmca.crt和参数gmsm2.param
$ ls ~/mydata/
gmca.crt  gmca.key  gmsm2.param

# 查看链证书脚本generate_chain_cert.sh使用方法：
$ bash ./generate_chain_cert.sh -h
Usage:
 -o <ca dir> Where ca.crt ca.key generate  # CA证书生成目录
 -m Input ca information manually     # 手动输入CA信息
 -g Generate chain certificates with guomi algorithms #生成国密版链证书
 -d The Path of Guomi Directory  #GM证书生成脚本所在路径，默认为FISCO-BCOS/cert/GM
 -h This help
Example:
 ./generate_chain_cert.sh -o /mydata -m  # 非国密版链证书生成用法，手动输入证书信息
 ./generate_chain_cert.sh -o /mydata      # 非国密版链证书生成用法，使用默认证书信息
guomi Example:
 ./generate_chain_cert.sh -o ~/mydata -m -g # 国密版链证书生成用法，手动输入证书信息
 ./generate_chain_cert.sh -o ~/mydata -g     # 国密版链证书生成用法，使用默认证书信息
```

6.3.2 生成机构证书

generate_agency_cert.sh 脚本用于生成机构证书：

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

#-------------------------------------------------------------
# -c: 链证书所在目录，这里是~/mydata
# -o: 生成的机构证书所在目录，这里是~/mydata
# -n: 机构证书名，这里是test_agency
# -g: 生成国密机构证书，这里必须设置
# 注: (若要手动输入机构证书信息，请在下面命令最后加上-m选项)
#-------------------------------------------------------------
$ bash ./generate_agency_cert.sh -c ~/mydata -o ~/mydata -n test_agency -g

# 此时~/mydata/test_agency目录下生成机构证书gmagency.crt和证书私钥gmagency.key
$ ls ~/mydata/test_agency/
gmagency.crt  gmagency.key  gmca.crt  gmsm2.param

# 查看generate_agency_cert.sh脚本用法
$ bash ./generate_agency_cert.sh -h
Usage:
 -c <ca dir> The dir of ca.crt and ca.key  # 指定颁发机构证书的CA证书和私钥所在目录
 -o <output dir> Where agency.crt agency.key generate # 指定机构证书输出目录
 -n <agency name> Name of agency  # 指定机构名
 -m Input agency information manually  # 若设置，表明手动输入机构证书信息，否则使用默认信息
 -g Generate agency certificates with guomi algorithms # 生成国密版机构证书
 -d The Path of Guomi Directory
 -h This help
Example:
 bash ./generate_agency_cert.sh -c /mydata -o /mydata -n test_agency # 非国密版机构证
书生成示例
 bash ./generate_agency_cert.sh -c /mydata -o /mydata -n test_agency -m
guomi Example:          #国密版机构证书生成示例
 bash ./generate_agency_cert.sh -c ~/mydata -o ~/mydata -n test_agency -g
 bash ./generate_agency_cert.sh -c ~/mydata -o ~/mydata -n test_agency -m -g
```

6.3.3 生成节点证书

generate_node_cert.sh 脚本用于生成节点证书：

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# 使用~/mydata/test_agency目录下机构证书为节点node0颁发证书，生成的证书置于~/mydata/node0/
↪data目录下
#-------------------------------------------------------------
# -a: 机构证书名，这里是test_agency
# -d: 机构证书所在路径，这里是~/mydata/test_agency
# -n: 节点名称，这里是node0
# -o: 节点证书所在路径，这里是~/mydata/node0/data
# -s: sdk证书名，这里是sdk1
# -g: 生成国密节点证书，这里必须设置
# (若要手动输入节点证书信息，请在下面命令最后加上-m选项)
#-------------------------------------------------------------
$ bash ./generate_node_cert.sh -a test_agency -d ~/mydata/test_agency -n node0 -o ~
↪/mydata/node0/data -s sdk1 -g

# 此时在~/mydata/node0/data目录下生成节点证书&&客户端连接证书
```

(continues on next page)

(continued from previous page)

```
 $ ls ~/mydata/node0/data -1
ca.crt       # 节点和客户端验证所需的CA证书
ca.key
client.keystore
gmagency.crt
gmca.crt
gmennode.crt
gmennode.key
gmnode.ca
gmnode.crt
gmnode.json
gmnode.key
gmnode.nodeid
gmnode.private
gmnode.serial
sdk1             #存储客户端连接节点证书
server.crt
server.key

# 设编译好的web3sdk位于~/mydata/web3sdk目录下，则将客户端证书拷贝到相应目录：
$ cp ~/mydata/node0/data/sdk1/* ~/mydata/web3sdk/dist/conf

# 查看节点证书脚本generate_node_cert.sh用法
$ bash ./generate_node_cert.sh  -h
Usage:
 -a <agency name> The agency name that the node belongs to # 为节点颁发证书的机构名
 -d <agency dir> The agency cert dir that the node belongs to # 机构证书所属目录
 -n <node name> Node name  # 节点名
 -o <output dir> Data dir of the node # 节点证书存放目录
 -m Input agency information manually # 手动输入节点证书信息
 -r The path of GM shell scripts directory
 -g generate guomi cert # 产生国密版节点证书
 -s the sdk name to connect with the node # 产生国密版节点证书时，必须产生sdk证书，-s指
定sdk名称
 -h This help
Example: # 非国密版节点证书生成示例
 bash ./generate_node_cert.sh -a test_agency -d /mydata/test_agency -n node0 -o /
↪mydata/node0/data
 bash ./generate_node_cert.sh -a test_agency -d /mydata/test_agency -n node0 -o /
↪mydata/node0/data -m
guomi Example: #国密版FISCO-BCOS节点证书生成示例
 bash ./generate_node_cert.sh -a test_agency -d ~/mydata/test_agency -n node0 -o ~/
↪mydata/node0/data -s sdk1 -g
 bash ./generate_node_cert.sh -a test_agency -d ~/mydata/test_agency -n node0 -o ~/
↪mydata/node0/data -m -s sdk1 -g
```

6.4 SDK证书

生成节点证书时，会在节点目录下同时生成SDK证书，如上例中node0的sdk证书位于`/mydata/node0/data/sdk1`目录。 也可使用`gmsdk.sh` 脚本手动生成sdk证书，**但生成后需要**:

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/cert/GM

# 执行国密版SDK生成脚本gmsdk
#------------------------------------------------------------
# 注：生成SDK证书时，不能输入空口令，口令长度至少为6个字符
#------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
$ bash ./gmsdk.sh sdk1 #生成的sdk证书位于sdk1目录下

# 用新生成的sdk证书sdk要连接的节点相关证书(设sdk要连接node0)
$ cp sdk1/* ~/mydata/node0/data/
```

注意事项

FISCO-BCOS采用三级证书链【链证书–> 机构证书—> 节点证书】认证，同一条链的所有机构证书必须由同一个链证书颁发，节点间证书认证才能成功。

6.5 搭建创世节点

注意事项

创建创世节点配置genesis.json前，必须确保已在创世节点data目录生成了节点证书，参考 证书生成

6.5.1 创世节点配置生成

FISCO BCOS提供generate_genesis.sh 脚本创建创世节点配置genesis.json:

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# 注意：请妥善保存god账号信息
# 生成god账号：
$ fisco-bcos --newaccount > guomi_godInfo.txt
$ cat guomi_godInfo.txt
address:0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad # god账号
publicKey:9c39ff254a673ec069c5aec56d206531ec45a2165f97df0b7259ad393dbde90b3f6896925060006ff6a4319
↪#god账号公钥
privateKey:f5474c4c9b1a3d922bc583b261c4ec45cd2bacb19030c82f9ad20d64acd431a8 #god账号
私钥

# 配置rpc端口：(设创世节点P2P端口是8545)
#-----------------------------------------------------------
# -o: 创世节点监听ip: 创世节点rpc端口，这里是127.0.0.1:8545
#-----------------------------------------------------------
$ bash ./config_rpc_address.sh -o 127.0.0.1:8545
Attention! This operation will change the target <rpcport url> of all tools under␣
↪tools/. Continue?
[Y/n]: ]y
$ cat ../web3lib/config.js
var proxy="http://127.0.0.1:8545"; #proxy地址更新为127.0.0.1:8545

var encryptType = 1;// 0:ECDSA 1:sm2Withsm3 #国密算法设置为1
//console.log('RPC='+proxy);
var output="./output/";
//console.log('Ouputpath='+output);

module.exports={
        HttpProvider:proxy,
        Ouputpath:output,
        EncryptType:encryptType,
        privKey:"bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd",
```

(continues on next page)

(continued from previous page)

```
        account:"0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3"
}

#------------------------------------------------------------
# -d: 创世节点路径，这里设置为~/mydata/node0
# -o: 生成的创世节点配置genesis.json所在目录，这里设置为~/mydata/node0
# -s: 可选，god账户地址，默认为0x3b5b68db7502424007c6e6567fa690c5afd71721
#------------------------------------------------------------
# 初始化创世节点配置genesis.json
$ bash ./generate_genesis.sh -d ~/mydata/node0 -o ~/mydata/node0 -s␣
↪0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad -g
God account address: 0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad
~/mydata/node0/genesis.json is generated

# 此时在~/mydata/node0/目录下生成genesis.json，内容如下:
$ cat ~/mydata/node0/genesis.json
{
     "nonce": "0x0",
     "difficulty": "0x0",
     "mixhash": "0x0",
     "coinbase": "0x0",
     "timestamp": "0x0",
     "parentHash": "0x0",
     "extraData": "0x0",
     "gasLimit": "0x13880000000000",
     "god":"0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad",
     "alloc": {},
     "initMinerNodes":[
↪"08eff55799d66d7426e64e44384c8e3eb5d849b4b5dbf21a32e5b954d787cc6c2500b8d25e14bf90e327940b0a2d3e
↪"]
}

# 若使用默认god账号0x3b5b68db7502424007c6e6567fa690c5afd71721生成创世节点配置，可使用如下命
令:
# ./generate_genesis.sh -d ~/mydata/node0 -o ~/mydata/node0 -g # 生成~/mydata/
↪genesis.json
# god账号信息存储在~/mydata/FISCO-BCOS/tools/scripts/god_info/guomiDefaultGod.txt中

# generate_genesis.sh 脚本功能
$ bash ./generate_genesis.sh -h
Usage:
    -o    <output dir>            Where genesis.json generate #创世块配置文件genesis.
↪json所在目录
    -i/-d <genesis node id/dir>   Genesis node id or dir #创世节点node id
Optional:
    -d    <genesis node dir>      Genesis node dir of the blockchain #创世节点所在目录
    #god账号地址，非国密版默认是0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
    -s    <god address>           Address of god account(default:␣
↪0xf78451eb46e20bc5336e279c52bda3a3e92c09b6)
    #为国密版FISCO BCOS产生genesis.json
    -g                            Generate genesis node for guomi-FISCO-BCOS
    -h                            This help # #显示帮助信息
Example: #生成非国密版FISCO BCOS genesis.json示例
    bash ./generate_genesis.sh -d /mydata/node0 -o /mydata/node1
    bash ./generate_genesis.sh -i xxxxxxxxxxxx -o /mydata/node1
    bash ./generate_genesis.sh -d /mydata/node0 -o /mydata/node1 -s␣
↪0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
    bash ./generate_genesis.sh -i xxxxxxxxxxxx -o /mydata/node1 -s␣
↪0xf78451eb46e20bc5336e279c52bda3a3e92c09b6
GUOMI Example: #生成国密版FISCO BCOS genesis.json示例
    bash ./generate_genesis.sh -d ~/mydata/node0 -o ~/mydata/node0 -g
```

(continues on next page)

(continued from previous page)

```
    bash ./generate_genesis.sh -i xxxxxxxxxxxx -o ~/mydata/node0 -g
    bash ./generate_genesis.sh -d ~/mydata/node0 -o ~/mydata/node0 -s␣
↪0x3b5b68db7502424007c6e6567fa690c5afd71721 -g
    bash ./generate_genesis.sh -i xxxxxxxxxxxx -o ~/mydata/node0 -s␣
↪0x3b5b68db7502424007c6e6567fa690c5afd71721 -g
```

6.5.2 创世节点环境初始化

FISCO-BCOS提供generate_genesis_node.sh脚本初始化节点环境并部署系统合约，下面使用该脚本初始化创世节点：

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

#调用generate_genesis_node.sh生成创世节点环境并部署系统合约
#-----------------------------------------------------------
# -o: 创世节点所在目录，这里设置为~/mydata
# -n: 创世节点名称，这里设置为node0
# -l: 创世节点监听ip，这里设置为127.0.0.1(多机多节点环境设置为外网IP或0.0.0.0)
# -r: 创世节点rpc端口，这里设置为8545(必须保证端口不冲突)
# -p: 创世节点p2p端口，这里设置为30303(必须保证端口不冲突)
# -c: 创世节点channel port端口，这里设置为8891(必须保证端口不冲突)
#-----------------------------------------------------------
$ bash ./generate_genesis_node.sh -o ~/mydata -n node0 -l 127.0.0.1 -r 8545 -p␣
↪30303 -c 8891 -g
---------- Generate node basic files ---------- #初始化创世节点配置信息
RUN: sh generate_node_basic.sh -o ~/mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -
↪c 8891 -e 127.0.0.1:30303 -g
#... 此处省略若干行 ...
SUCCESS execution of command: sh generate_node_basic.sh -o ~/mydata -n node0 -l␣
↪127.0.0.1 -r 8545 -p 30303 -c 8891 -e 127.0.0.1:30303 -g
---------- Deploy system contract ----------  # 部署系统合约
RUN: sh deploy_systemcontract.sh -d ~/mydata/node0 -g
Pre-start genesis node
Deploy System Contract
Start depoly system contract
RUN: babel-node deploy_systemcontract.js ~/FISCO-BCOS/tools/web3lib/tmp_config_
↪deploy_system_contract.js
SystemProxycomplie success!
#... 此处省略若干行 ...
### 显示系统合约信息
-----------------SystemProxy route ----------------------
0 )TransactionFilterChain=>0x08f8eeb7959660ed84b18b2daf271dc19e62aaf9,false,22
1 )ConfigAction=>0xbc2b0ca104ac0a824b05c1e055f24b1857e69b35,false,23
2 )NodeAction=>0x22af893607e84456eb5aea0b277e4dffe260fdcd,false,24
3 )CAAction=>0xa92014f1593bbaa1294be562f0dbfbc7aca0d579,false,25
4 )ContractAbiMgr=>0xe441c93f05d2d200c9e51fdac87b9851483aa341,false,26
5 )ConsensusControlMgr=>0xb53b1513c2edf88c0a27f3670385481821cc0818,false,27
6 )FileInfoManager=>0xa9e38b700f6462b21e595402e870bc51ae852768,false,28
7 )FileServerManager=>0x3eeff72da75f3a2a1a97d958142b08ca75b86b5a,false,29
SUCCESS execution of command: babel-node deploy_systemcontract.js /data/chenyujie/
↪guomi/FISCO-BCOS/tools/web3lib/tmp_config_deploy_system_contract.js
/data/chenyujie/guomi/FISCO-BCOS/tools/scripts
Stop genesis node
Reconfig genesis node
SystemProxy address: 0xee80d7c98cb9a840b9c4df742f61336770951875
Deploy System Contract Success!
SUCCESS execution of command: sh deploy_systemcontract.sh -d ~/mydata/node0 -g
#... 此处省略若干行 ...
```

(continues on next page)

(continued from previous page)

```
# 显示节点信息
------------------------------------------------------------------
Name:
Node dir:                ~/mydata/node0 #创世节点名
Agency:
CA hash:                 D14983471F0AC975
Node ID:
↪3d4fe4c876cac411d4c7180b5794198fb3b4f3e0814156410ae4184e0a51097a01bf63e431293f30af0c01a57f24477a
↪#node id
RPC address:             127.0.0.1:8545
P2P address:             127.0.0.1:30303
Channel address:         127.0.0.1:8891
SystemProxy address:     0xee80d7c98cb9a840b9c4df742f61336770951875 #系统代理合约地址
God address:             0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad # god账号地地址
State:                   Running (pid: 11524)  #节点运行状态
------------------------------------------------------------------

# generate_node.sh脚本功能
$ bash ./generate_genesis_node.sh -h
Usage:
 -o <output dir> Where node files generate # 节点所处目录
 -n <node name> Name of node # 节点名
 -l <listen ip> Node\'s listen IP # 监听IP
 -r <RPC port> Node\'s RPC port  # RPC端口
 -p <P2P port> Node\'s P2P port # P2P端口
 -c <channel port> Node\'s channel port # channel port
 -a <agency name> The agency name that the node belongs to #国密版FISCO-BCOS不需关注
 -d <agency dir> The agency cert dir that the node belongs to #国密版FISCO-BCOS不需关
注
Optional:
 -m Input agency information manually #手动输入证书信息，国密版FISCO-BCOS不需关注
 -g Generate guomi genesis node # 国密版FISCO-BCOS加该选项
 -h This help
Example: #非国密版FISCO-BCOS使用示例
 ./generate_genesis_node.sh -o /mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -c
↪8891 -d /mydata/test_agency -a test_agency
guomi Example: #国密版FISCO-BCOS使用示例
 ./generate_genesis_node.sh -o ~/mydata -n node0 -l 127.0.0.1 -r 8545 -p 30303 -c
↪8891 -g
```

说明

若本步执行异常，可能是国密版nodejs环境初始化出错，请通过 `init_guomi_nodejs.sh` 脚本重新初始化nodejs环境

```
cd ~/mydata/FISCO-BCOS/tools/scripts && bash init_guomi_nodejs.sh
```

6.5.3 check创世节点环境

创世节点部署成功后，需要check创世节点进程和是否正常出块：

```
# check创世节点是否正常出块
$ tail -f ~/mydata/node0/log/log_2018080809.log | grep +++
INFO|2018-08-08 09:21:18:109|++++++++++++++++++++++++++++ Generating seal
↪onff05d8b4386fc1a058b9c9da7816fa1e340d0bffcd008424104b2ed48740ace4#1tx:0,
↪maxtx:1000,tq.num=0time:1533691278109
INFO|2018-08-08 09:21:19:138|++++++++++++++++++++++++++++ Generating seal
↪on7e78c3a28b652a243ec2d2ffe2c3c927469bddef53cfaf9cab9128b7930f3c50#1tx:0,
↪maxtx:1000,tq.num=0time:1533691279138
```

(continues on next page)

(continued from previous page)

通过以上输出可看出，创世节点进程已启动，且出块正常。

6.6 普通节点环境搭建

6.6.1 生成节点证书

证书生成可参考创世节点证书生成 ,若普通节点与创世节点属于同一机构，必须使用同一机构证书颁发节点证书。

```
# 进入脚本所在目录(设源码位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# 在~/mydata/node1/data目录下生成普通节点证书
#------------------------------------------------------------
# -a: 节点所在机构名称，这里设置为test_agency
# -d: 机构证书所在目录，这里设置为~/mydata/test_agency/
# -n: 普通节点名称，这里设置为node1
# -o: 普通节点机构证书所在目录，这里设置为~/mydata/node1/data
# -s: sdk证书名称，这里设置为sdk1
# 注: (若要手动输入节点信息，请在下面命令最后加上-m选项)
#------------------------------------------------------------
$ bash ./generate_node_cert.sh -a test_agency -d ~/mydata/test_agency/ -n node1 -o␣
↪~/mydata/node1/data -s sdk1 -g

# ~/mydata/node1/data目录下生成的证书如下:
$ ls ~/mydata/node1/data/ -1
ca.crt
ca.key
client.keystore
gmagency.crt
gmca.crt
gmennode.crt
gmennode.key
gmnode.ca
gmnode.crt
gmnode.json
gmnode.key
gmnode.nodeid
gmnode.private
gmnode.serial
sdk1
server.crt
server.key
```

6.6.2 初始化普通节点环境

类似于 创世节点环境初始化‘_ ,普通节点使用generate_node.sh脚本初始化普通节点环境。

```
# 进入脚本所在目录(设源码位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# 查看创世节点信息:
#------------------------------------------------------------
# -d: 指定创世节点目录，这里是~/mydata/node0
# -g: 指定节点类型是国密版FISCO-BCOS，这里必须设置
```

(continues on next page)

(continued from previous page)

```
#------------------------------------------------------------
$ bash ./node_info.sh -d ~/mydata/node0 -g
------------------------------------------------------------------
Name:
Node dir:               /mydata/node0
Agency:
CA hash:                D14983471F0AC975
Node ID:
↪3d4fe4c876cac411d4c7180b5794198fb3b4f3e0814156410ae4184e0a51097a01bf63e431293f30af0c01a57f24477a
RPC address:            127.0.0.1:8545
P2P address:            127.0.0.1:30303
Channel address:        127.0.0.1:8891
SystemProxy address:    0xee80d7c98cb9a840b9c4df742f61336770951875
God address:            0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad
State:                  Running (pid: 11524)
------------------------------------------------------------------


#------------------------------------------------------------
# -o : 普通节点所在目录,这里是~/mydata
# -n : 普通节点名称，这里是node1
# -l : 普通节点监听IP，单机多节点可用127.0.0.1
# -r : 普通节点RPC端口，这里设置为8546(注：不同节点RPC端口不能冲突)
# -p : 普通节点p2p连接端口，这里设置为30304(注：不同节点p2p端口不同，且不能和所有其他端口冲突)
# -c : channel port端口，这里设置为8892(注：不同节点channel port端口不同，且不能和所有其他端
口冲突)
# -e : bootstrapnodes.json配置，配置相邻节点的IP和p2p端口，这里设置为{创世节点ip:创世节
点p2p端口}和{本节点ip:本节点p2p端口}
# -x : 系统代理合约地址，这里为0xee80d7c98cb9a840b9c4df742f61336770951875(通过 bash ./
↪node_info.sh -d ~/mydata/node0 -g获取，对应【SystemProxy address】)
# -i : 创世节点node id (通过bash ./node_info.sh -d ~/mydata/node0 -g获取，对应【Node␣
↪ID】)
# -s : god账号地址，这里为0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad(通过bash ./node_
↪info.sh -d ~/mydata/node0 -g获取，对应【God address】)
# -g: 表示生成国密节点，必须加上该选项
#------------------------------------------------------------
$ bash ./generate_node.sh -o ~/mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c␣
↪8892 -e 127.0.0.1:30303,127.0.0.1:30304 -x␣
↪0xee80d7c98cb9a840b9c4df742f61336770951875 -i␣
↪3d4fe4c876cac411d4c7180b5794198fb3b4f3e0814156410ae4184e0a51097a01bf63e431293f30af0c01a57f24477a
↪-s 0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad -g
# 创建节点环境
---------- Generate node basic files ----------
RUN: sh generate_node_basic.sh -o ~/mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -
↪c 8892 -e 127.0.0.1:30303,127.0.0.1:30304 -x␣
↪0xee80d7c98cb9a840b9c4df742f61336770951875 -g
#...此处省略若干行....
SUCCESS execution of command: sh generate_node_basic.sh -o ~/mydata -n node1 -l␣
↪127.0.0.1 -r 8546 -p 30304 -c 8892 -e 127.0.0.1:30303,127.0.0.1:30304 -x␣
↪0xee80d7c98cb9a840b9c4df742f61336770951875 -g
# 创建普通节点genesis.json
---------- Generate node genesis file ----------
# ... 此处省略若干行...
# 输出普通节点信息
------------------------------------------------------------------
Name:
Node dir:               ~/mydata/node1
Agency:
CA hash:                F4AC757508FF6AB2
Node ID:
↪9940c84c1964095c7a3e0daa37c5cbe718dfb2a20def6df19ffd84438e307fa63427920fbf76550e13b318ed0464b68
```

(continues on next page)

(continued from previous page)

```
RPC address:              127.0.0.1:8546
P2P address:              127.0.0.1:30304
Channel address:          127.0.0.1:8892
SystemProxy address:      0xee80d7c98cb9a840b9c4df742f61336770951875
God address:              0xf02a10f685a90c3bfc2eccd906b75fe3feeec9ad
State:                    Stop
------------------------------------------------------------------

## generate_node.sh用法
$ bash ./generate_node.sh -h

Usage:
 -o <output dir> Where node files generate  #普通节点所在目录
 -n <node name> Name of node  #普通节点名称
 -l <listen ip> Node\'s listen IP #普通节点监听IP(推荐填外网IP)
 -r <RPC port> Node\'s RPC port #普通节点RPC端口
 -p <P2P port> Node\'s P2P port #普通节点P2P端口
 -c <channel port> Node\'s channel port #普通节点channel port
 -e <bootstrapnodes> Node\'s bootstrap nodes    #普通节点bootstrapnode.json配置，主要包括要连接节点的IP和端口
 -a <agency name> The agency name that the node belongs to #普通节点所属机构(国密版搭建过程忽略该参数)
 -d <agency dir> The agency cert dir that the node belongs to #普通节点机构证书目录(国密版搭建过程忽略该参数)
 -i <genesis node id> Genesis node id   #普通节点所属链的创世节点node id
 -s <god address> God address # 普通节点所属链的god账号
 -x <system proxy address> System proxy address of the blockchain # 普通节点所属链的系统合约地址
Optional:
 -m Input agency information manually #手动输入机构证书信息(国密版搭建过程忽略该参数)
 -g Create guomi node #生成国密版普通节点
 -h This help
Example: #非国密版generate_node.sh使用示例
 bash ./generate_node.sh -o /mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c 8892␣
↪-e 127.0.0.1:30303,127.0.0.1:30304 -d /mydata/test_agency -a test_agency -x␣
↪0x919868496524eedc26dbb81915fa1547a20f8998 -i xxxxxx -s xxxxxx
GuomiExample: #国密版generate_node.sh使用示例
 bash ./generate_node.sh -o ~/mydata -n node1 -l 127.0.0.1 -r 8546 -p 30304 -c␣
↪8892 -e 127.0.0.1:30303,127.0.0.1:30304 -x xxxxx -i xxxxxx -s xxxxxx -g
```

6.6.3 启动节点进程

FISCO BCOS在节点目录下提供 start.sh和stop.sh来启停节点：

```
# 进入节点目录
$ cd ~/mydata/node1

# 调用start.sh脚本启动进程
$ ./start.sh
```

6.6.4 check节点环境

启动普通节点后，check普通节点进程和error日志：

```
# 查看普通节点进程状态：创世节点和普通节点进程均启动
$ ps aux | grep fisco | grep -v grep
root     20995  0.2  0.1 2171448 13316 pts/3   Sl   09:15   0:05 ./fisco-bcos --
↪genesis /mydata/node0/genesis.json --config /mydata/node0/config.json
```

(continues on next page)

(continued from previous page)

```
root     21727  0.7  0.1 2023984 14092 pts/4   Sl   09:52   0:01 ./fisco-bcos --
↪genesis /mydata/node1/genesis.json --config /mydata/node1/config.json

# 通过最新日志check节点连接：刷出"Recv topic"日志，表明节点连接正常
$ tail -f ~/mydata/node1/log_2018081219.log | grep "Recv topic"
DEBUG|2018-08-12 19:34:13:659| Recv topic type:0 topics:0
DEBUG|2018-08-12 19:34:14:659| Recv topic type:0 topics:0
DEBUG|2018-08-12 19:34:15:659| Recv topic type:0 topics:0
DEBUG|2018-08-12 19:34:16:659| Recv topic type:0 topics:0
```

6.7 节点入网

注意事项

节点入网时，请确保首先注册创世节点

6.7.1 创世节点入网

FISCO BCOS提供register_node.sh工具用于节点入网，创世节点入网过程如下：

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# -d: 创世节点所在目录
# -g: 创世节点类型是国密版FISCO-BCOS
$ bash ./register_node.sh -d ~/mydata/node0 -g
RUN: babel-node tool.js NodeAction register ~/mydata/node0/data/gmnode.json
{ HttpProvider: 'http://127.0.0.1:8546',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :NodeAction
Func :register
SystemProxy address 0xee80d7c98cb9a840b9c4df742f61336770951875
node.json=~/mydata/node0/data/gmnode.json
NodeAction address 0x22af893607e84456eb5aea0b277e4dffe260fdcd
send transaction success:␣
↪0xbfc83175af76dd7e466b75ecd76cd6fd328a4b700233943a81187ea72b0c6bf7
SUCCESS execution of command: babel-node tool.js NodeAction register ~/mydata/
↪node0/data/gmnode.json
~/mydata/FISCO-BCOS/tools/scripts
Register Node Success!

# 创世节点配置~/mydata/node0/data/gmnode.json如下:
$ cat ~/mydata/node0/data/gmnode.json
{
 "id":
↪"730195b08dda7b027c9ba5bec8ec19420aa996c7ce72fa0954711d46c1c66ae8c2eeaa5f84d1f7766f21ba3dc822bc
↪",
 "name":"",
 "agency":"",
 "caHash":"AF33DEB4033C0D47"
}
```

6.7.2 普通节点入网

普通节点入网过程如下：

```
# 进入脚本所在目录(设FISCO-BCOS位于~/mydata目录)
$ cd ~/mydata/FISCO-BCOS/tools/scripts

# -d: 普通节点目录，这里是~/mydata/node1
# -g: 普通节点类型是国密版FISCO-BCOS，必须设置
$ bash ./register_node.sh -d ~/mydata/node1 -g
RUN: babel-node tool.js NodeAction register /mydata/node1/data/gmnode.json
{ HttpProvider: 'http://127.0.0.1:8545',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :NodeAction
Func :register
SystemProxy address 0xee80d7c98cb9a840b9c4df742f61336770951875
node.json=~/mydata/node1/data/gmnode.json
NodeAction address 0x22af893607e84456eb5aea0b277e4dffe260fdcd
send transaction success:␣
↪0xc67d4e08a03a7094244e3de100979e1f0e50b7f9d83be5691d3833e3ddfcb97b
SUCCESS execution of command: babel-node tool.js NodeAction register ~/mydata/
↪node1/data/gmnode.json { HttpProvider: 'http://127.0.0.1:8545',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :NodeAction
Func :register
SystemProxy address 0xee80d7c98cb9a840b9c4df742f61336770951875
node.json=~/mydata/node1/data/gmnode.json

~/mydata/FISCO-BCOS/tools/scriptsNodeAction address␣
↪0x22af893607e84456eb5aea0b277e4dffe260fdcd
# 查看记账节点信息
RUN: babel-node tool.js NodeAction all
{ HttpProvider: 'http://127.0.0.1:8545',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
send transaction success:␣
↪0x1f923378d2640acad78378ee2f21002213cb9f81dfcb7b0f2e42ea5a960a08e6
SUCCESS execution of command: babel-node tool.js NodeAction register ~/mydata/
↪node1/data/gmnode.json
~/mydata/FISCO-BCOS/tools/scripts
RUN: babel-node tool.js NodeAction all

Soc File :NodeAction
Func :all
SystemProxy address 0xee80d7c98cb9a840b9c4df742f61336770951875
NodeAction address 0x22af893607e84456eb5aea0b277e4dffe260fdcd
NodeIdsLength= 2
----------node 0---------
id=3d4fe4c876cac411d4c7180b5794198fb3b4f3e0814156410ae4184e0a51097a01bf63e431293f30af0c01a57f2447
name=
agency=
caHash=D14983471F0AC975
Idx=0
blocknumber=30
```

(continues on next page)

(continued from previous page)

```
----------node 1---------
id=9af16c4543919589982932b57bb97b162f8eba522037a95e7b013780911c2b0ffdef775b5387b2a4f4867b1271a063
name=
agency=
caHash=95F1A5C35D8CFFA7
Idx=1
blocknumber=31
SUCCESS execution of command: babel-node tool.js NodeAction all{ HttpProvider:
↪'http://127.0.0.1:8545',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :NodeAction
Func :all
```

6.7.3 查看节点入网情况

FISCO-BCOS提供了`node_all.sh`命令查看记账节点信息：

```
$ bash ./node_all.sh
RUN: babel-node tool.js NodeAction all
{ HttpProvider: 'http://127.0.0.1:8545',
  Ouputpath: './output/',
  EncryptType: 1,
  privKey: 'bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd',
  account: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3' }
Soc File :NodeAction
Func :all
SystemProxy address 0xee80d7c98cb9a840b9c4df742f61336770951875
NodeAction address 0x22af893607e84456eb5aea0b277e4dffe260fdcd
NodeIdsLength= 2
----------node 0---------
id=3d4fe4c876cac411d4c7180b5794198fb3b4f3e0814156410ae4184e0a51097a01bf63e431293f30af0c01a57f2447
name=
agency=
caHash=D14983471F0AC975
Idx=0
blocknumber=30
----------node 1---------
id=9af16c4543919589982932b57bb97b162f8eba522037a95e7b013780911c2b0ffdef775b5387b2a4f4867b1271a063
name=
agency=
caHash=95F1A5C35D8CFFA7
Idx=1
blocknumber=31
SUCCESS execution of command: babel-node tool.js NodeAction all
```

从输出信息可看出，创世节点和普通节点均成功入网。

6.7.4 check节点入网情况

使用如下命令检查创世节点入网情况，若输出+++等打包信息，表明创世节点入网成功：

```
$ tail -f ~/mydata/node0/log/log_2018081220.log | grep +++
INFO|2018-08-12 20:33:13:431|++++++++++++++++++++++++++ Generating seal␣
↪on31e1a94c1feb79a4145272a9c5175636d7c24cf4ed90b0b2f5471e4323e5e89e#34tx:0,
↪maxtx:0,tq.num=0time:1534077193431
```

(continues on next page)

(continued from previous page)

```
INFO|2018-08-12 20:33:15:457|++++++++++++++++++++++++++ Generating seal␣
↪ondbfa0c0cac0e39f0d22c0c6fa3c21e77e15a3c31d8c81dac580dfbf95b2f96cb#34tx:0,
↪maxtx:0,tq.num=0time:1534077195457
```

同样地，使用如下命令检查普通节点入网情况，若输出+++等打包信息，表明普通节点入网成功：

```
$ tail -f ~/mydata/node1/log/log_2018081220.log | grep +++
INFO|2018-08-12 20:33:36:696|++++++++++++++++++++++++++ Generating seal␣
↪on17d28b77047be017be9ec7ebd048b3b9b711cf75dcbdc1eabfe9cd57d8d6e7f7#34tx:0,
↪maxtx:0,tq.num=0time:1534077216696
INFO|2018-08-12 20:33:38:718|++++++++++++++++++++++++++ Generating seal␣
↪onf082c29bcadab361a1bd88853964f8daac643265e4b1a786d669d58a99ce3833#34tx:0,
↪maxtx:0,tq.num=0time:1534077218718
```

congratulations :)

至此，您已经成功搭建一条可用的国密版FISCO-BCOS链

- 更高级的使用方法请参考 FISCO-BCOS系统合约
- 国密版web3sdk配置和使用方法请参考 国密版web3sdk

6.8 国密版web3sdk搭建

注意事项

1. 搭建国密版web3sdk前，请参考 web3sdk快速搭建文档 编译web3sdk;
2. 搭建国密版web3sdk前，请参考 国密版FISCO-BCOS快速搭建文档 搭建一条可用的国密版FISCO-BCOS链

6.8.1 生成客户端证书

国密版FISCO-BCOS生成节点证书同时会生成SDK证书，请参考 国密版FISCO BCOS证书生成.

生成SDK证书时，直接将节点证书拷贝到`web3sdk/dist/conf`目录即可:

```
# 设web3sdk连接的节点位于~/mydata/node0目录, sdk名称为sdk1
# 设web3sdk位于~/mydata/web3sdk目录
$ cp ~/mydata/node0/data/sdk1/* ~/mydata/web3sdk/dist/conf
```

6.8.2 配置SDK

web3sdk配置

web3sdk中开启国密算法，需要将 *encryptType* 选项设置为1，其他选项参考 非国密版web3sdk配置 :


```
<!--encryptType: 国密算法开关，0表示关闭国密算法，1表示开启国密算法-->
<bean id="encryptType" class="org.bcos.web3j.crypto.EncryptType">
        <constructor-arg value="1"/>
</bean>

<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
        <!--systemProxyAddress: 系统合约地址-->
        <!--privKey: 发送交易的私钥,权限控制工具中建议使用GOD私钥-->
        <property name="systemProxyAddress" value="0x919868496524eedc26dbb81915fa1547a20f8998" />
        <property name="privKey" value="bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd" />
        <property name="account" value="0x776bd5cf9a88e9437dc783d6414bccc603015cf0" />
        <property name="outPutpath" value="./output/" />
</bean>

<!--配置节点连接信息-->
<bean id="channelService" class="org.bcos.channel.client.Service">
        <property name="orgID" value="WB" />
        <property name="connectSeconds" value="10" />
        <property name="connectSleepPerMillis" value="10" />
        <property name="allChannelConnections">
                <map>
                        <entry key="WB">
                                <bean class="org.bcos.channel.handler.ChannelConnections">
                                        <!--caCertPath: 客户端CA证书ca.crt存储路径，默认是classpath:ca.crt-->
                                        <!--clientKeystorePath: 客户端keystore证书client.keystore存储路径-->
```

6.8.3 check国密版web3sdk

测试web3sdk与节点连接是否正常

类似于 非国密版web3sdk， 国密版web3sdk也可以通过TestOk测试web3sdk与节点连接是否正常，若输出 `INIT GUOMI KEYPAIR from Private Key` 和 `to balance` 等日志，则说明国密版web3sdk与节点连接成功。

```
# 进入web3sdk目录(设源码位于~/mydata/web3sdk/dist中)
$ cd ~/mydata/web3sdk/dist

# 调用测试程序TestOk
$ java -cp 'conf/:apps/*:lib/*' org.bcos.channel.test.TestOk
======================================================================
=====INIT GUOMI KEYPAIR from Private Key
====generate kepair from priv␣
↪key:bcec428d5205abe0f0cc8a734083908d9eb8563e31f943d760786edf42ad67dd
generate keypair data succeed
####create credential succ, begin deploy contract
####contract address is: 0xee80d7c98cb9a840b9c4df742f61336770951875
============to balance:4
============to balance:8
============to balance:12
... 此处省略若干行 ...
```

CHAPTER 7

特性详解

7.1 总体介绍

FISCO-BCOS特性介绍

7.2 基础特性

7.2.1 FISCO BCOS系统合约介绍

作者：fisco-dev

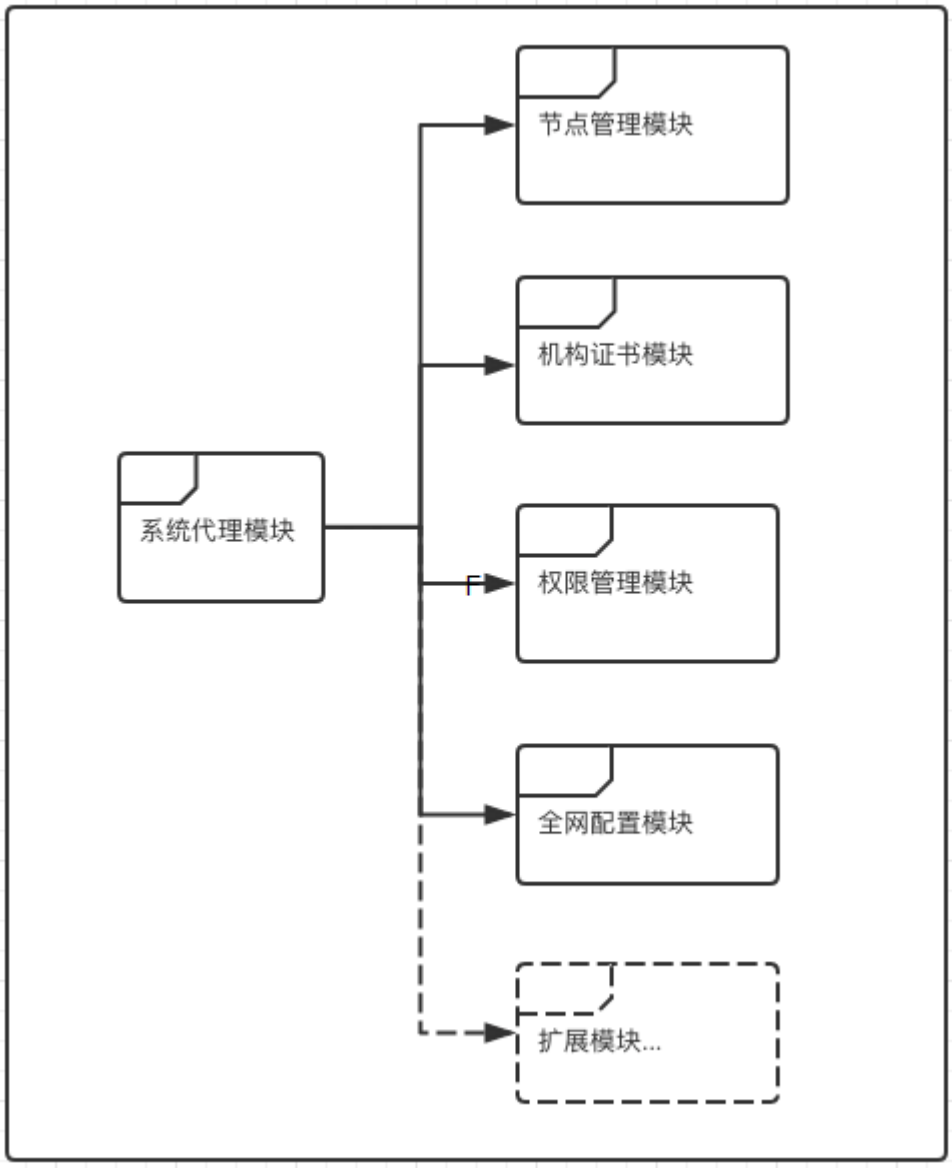
- [FISCO BCOS系统合约介绍](#FISCO BCOS系统合约介绍)
  - 设计概述
  - 实现概述
    - 系统代理合约
    - 节点管理合约
    - 机构证书合约
    - 权限管理合约
    - 全网配置合约
  - 自定义扩展
    - 示例*1*-自定义业务配置合约
    - 示例*2*-自定义业务权限*Filter*合约

设计概述

FISCO BCOS区块链为了满足准入控制、身份认证、配置管理、权限管理等需求，在网络启动之初，会部署一套功能强大、结构灵活且支持自定义扩展的智能合约，统称系统合约。

系统合约原则上由区块链管理员在网络启动之初部署全网生效。若是在网络运行期间重新部署变更升级，则需要在全网所有节点许可的情况下由区块链管理员来执行操作。

当前FISCO BCOS系统合约主要有五个模块，系统代理模块、节点管理模块、机构证书模块、权限管理模块、全网配置模块。系统合约的模块可以根据需要自定义扩展，它既可以供区块链核心调用也可以对DAPP提供服务。每个模块由一个或多个智能合约来实现。模块结构图如下：



模块图

实现概述

当前FISCO BCOS对系统代理模块、节点管理模块、机构证书模块、权限管理模块、全网配置模块都做了对应的合约实现。合约源代码目录为systemcontractv2/。下面依次介绍各个合约实现的接口与逻辑。

系统代理合约

SystemProxy.sol是系统代理模块的实现合约。它实现了路由到合约地址的命名服务，提供了系统合约的统一入口。内部实现中是通过mapping类型成员变量_routes来维护所有的路由表信息。路由表信息项的数据结构主要是：

```
struct SystemContract {
        address _addr;                  #合约地址
        bool _cache;                   #缓存标志位
        uint _blocknumber;            #生效区块高度
    }
```

主要接口如下：

| 接口 | 输入参数 | 输出参数 | 说明 | | ——— | :———————————— | ———————————— | ———————— | | getRoute | string key#路由名称 | address, bool,uint # 合约地址，缓存标志位，生效区块高度 | 获取路由信息 | | setRoute | string key, address addr, bool cache# 路由名称，合约地址，缓存标志位，生效区块高度 | 无 | 设置路由信息，若该路由名称已存在，则覆盖 |

节点管理合约

NodeAction.sol是节点管理模块的实现合约。它实现了对网络中所有节点列表信息的登记、管理、维护功能。每当网络中有节点加入或退出都必须与节点管理合约进行交互。

在FISCO BCOS中节点分为三种类型：核心节点、全节点、轻节点。

```
enum NodeType{
        None,
        Core,    //核心
        Full,    //全节点
        Light    //轻节点
    }
```

节点信息的数据结构是：

```
struct NodeInfo{
        string                  id;                         #节点身份ID
        string                  ip;                 #机器IP
        uint                  port;                     #机器端口
        NodeType    category;           #节点类型
        string                  desc;                 #节点描述
        string                  CAhash;               #节点机构证书哈希
        string                  agencyinfo;  #节点其他信息
        uint                  idx;                      #节点序号
        uint                  blocknumber;#区块高度
    }
```

主要接口如下：

| 接口 | 输入参数 | 输出参数 | 说明 | | ———— | :———————————— | ———- | ———————- | | registerNode | string _id,string _ip,uint _port,NodeType _category,string _desc,string _CAhash,string _agencyinfo,uint _idx #节点身份ID、IP、端口、节点类型、节点描述、节点CA哈希、节点agency、节点序号 | bool #注册结果 | 注册节点，若该节点信息已存在，则忽略 | | cancelNode | string _id #节点身份ID | bool #注册结果 | 注销节点，若该节点信息不存在，则忽略 |

机构证书合约

CAAction.sol是机构证书模块的实现合约。它实现了对网络中所有节点的机构证书信息的登记、管理、维护功能。当网络启用机构证书验证功能的情况下，网络中节点加入或退出都需要与机构证书合约进行交互。

机构证书的数据结构是：

```
struct CaInfo{
        string  hash;                   #节点机构证书哈希
```

(continues on next page)

(continued from previous page)

```
        string pubkey;                  #证书公钥
        string orgname;                  #机构名称
        uint notbefore;                  #证书启用日期
        uint notafter;                  #证书失效时间
        CaStatus status;          #证书状态
        string    whitelist;#IP白名单
        string    blacklist;#IP黑名单
        uint    blocknumber;#生效区块高度
    }
```

主要接口如下：

| 接口 | 输入参数 | 输出参数 | 说明 | | —— | ——————————- | ——————————- | ——————- | | update | string _hash,string _pubkey,string _orgname,uint _notbefore,uint _notafter,CaStatus _status,string _whitelist,string _blacklist # 证书哈希、证书公钥、机构名称、 证书启用日期、 证书失效时间、证书状态、IP白名单、IP黑名单 | bool #更新结果 | 更新证书信息， 若该证书信息不存在，则新建证书记录 | | get | string _hash#证书哈希 | string,string,string,uint,uint,CaStatus,uint# 证书哈希、证书公钥、机构名称、证书启用日期、证书失效时间、证书状态、生效区块块号 | 查询证书信息 |

权限管理合约

FISCO BCOS基于角色的身份权限设计有三要点：一个外部账户只属于一个角色；一个角色拥有一个权限项列表； 一个权限项由合约地址加上合约接口来唯一标识。

当前FISCO BCOS权限管理模块主要由TransactionFilterChain.sol、TransactionFilterBase.sol、AuthorityFilter.sol、Group.sol四个合约来实现。

TransactionFilterChain是对Filter模型的实现框架。它在内部维护了一个实现继承于TransactionFilterBase的Filter合约地址列表。它对区块链核心提供了统一的权限检查接口process。process执行过程中会对Filter合约地址列表中的所有Filter依次执行process函数，以完成所有需要的权限检查。

TransactionFilterBase是Filter的基类合约。所有Filter必须要实现它的process接口。AuthorityFilter是继承于TransactionFilterBase的角色权限Filter实现。它的process接口实现了对用户所属角色组的权限项进行检查逻辑。

Group是对角色的实现。它内部维护了该角色的所有权限项的mapping标志位。

主要接口如下：

| 合约 | 接口 | 输入参数 | 输出参数 | 说明 | | ——————— | ————- | ——————————- | ——— | ——- | | TransactionFilterBase | process | address origin, address from, address to, string func, string input# 用户外部账户、交易发起账户、合约地址、合约接口、交易输入数据 | bool#处理结果 | 权限检查 | | Group | setPermission | address to, string func, bool perrmission# 合约地址、合约接口、权限标记 | bool#处理结果 | 设置角色权限项 |

全网配置合约

ConfigAction.sol是全网配置模块的实现合约。它维护了FISCO BCOS区块链中全网运行的可配置信息。配置信息可以通过交易广播上链来达到全网配置的一致性更新。原则上只能由区块链管理员来发出全网配置更新交易。

ConfigAction.sol的内部实现中维护了配置项信息的mapping 成员变量。

主要接口如下：

| 接口 | 输入参数 | 输出参数 | 说明 | | —- | ———————— | ——————— | —— | | set | string key, string value# 配置项、配置值 | 无 | 设置配置项 | | get | string key #配置项 | string, uint# 配置值、生效区块高度 | 查询配置值 |

当前FISCO BCOS主要有以下全网配置项：

| 配置项 | 说明 | 默认值 | 推荐值 | | ——————— | ———————————- | ——— | ——— | | maxBlockHeadGas | 块最大GAS （16进制） | 200000000 | 20000000000 | | intervalBlockTime | 块间隔(ms) （16进制） | 1000 | 1000 | | maxBlockTranscations | 块最大交易数（16进制） | 1000 | 1000 | | maxNonceCheckBlock | 交易nonce检查最大块范围（16进制） | 1000 | 1000 | | maxBlockLimit | blockLimit超过当前块号的偏移最大值（16进制） | 1000 | 1000 | | maxTranscationGas | 交易的最大gas（16进制） | 20000000 | 20000000 | | CAVerify | CA验证开关 | FALSE | FALSE |

自定义扩展

示例1-自定义业务配置合约

假设业务需要利用系统合约框架，自定义业务配置合约以对业务相关合约提供配置服务。大体可以参考以下步骤来扩展：

1. 根据业务合约需求，实现业务配置合约的设置配置项接口set和查询配置值接口get。
2. 部署业务配置合约，获得业务配置合约链上地址。
3. 调用系统代理合约SystemProxy的setRoute接口，将业务配置合约地址注册到路由表中。
4. 至此，业务配置合约已经完成在系统代理合约的路由注册，已可在业务交易中调用。

业务配置合约的使用方法：

1. 调用SystemProxy的getRoute接口运行时获得业务配置合约地址。
2. 通过业务配置合约地址调用查询配置值接口get获得配置值。

示例2-自定义业务权限Filter合约

假设业务需要增加业务权限校验逻辑，则可以利用权限管理合约的Filter机制来无缝扩展。大体可以参考以下步骤来扩展：

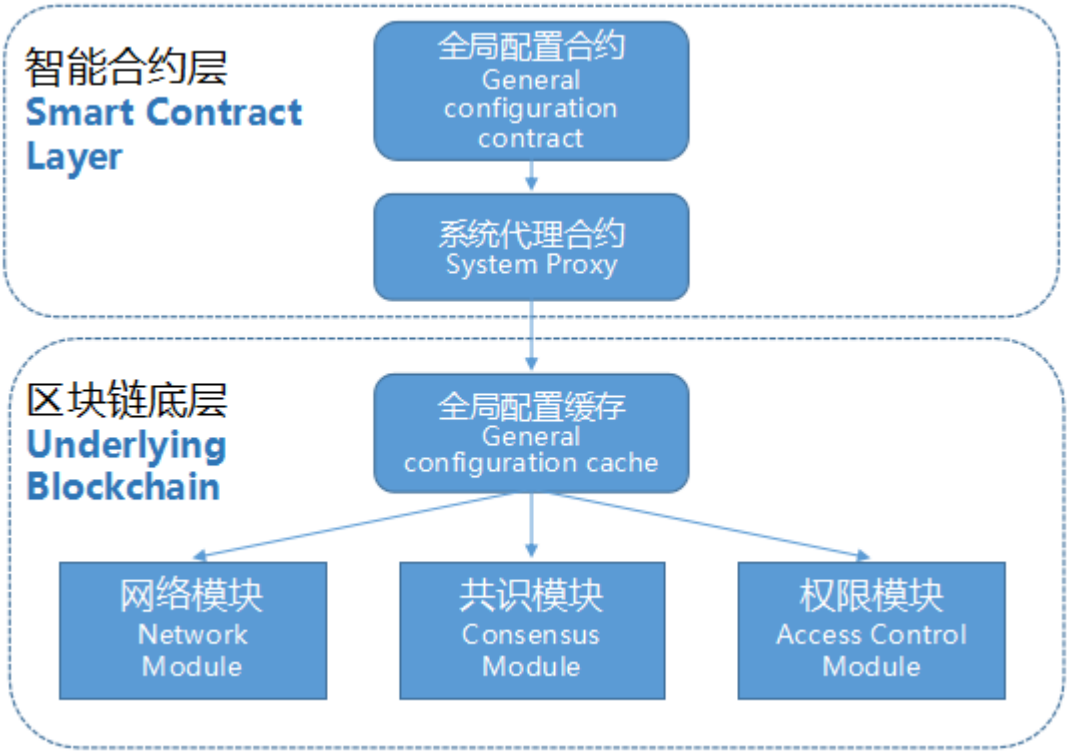
1. 继承于TransactionFilterBase实现一个业务权限Filter合约，业务权限Filter合约根据业务需要的权限校验逻辑实现process接口。
2. 部署业务权限Filter合约，获得对应的合约地址。
3. 调用系统代理合约SystemProxy的getRoute接口，获得TransactionFilterChain合约地址。
4. 调用TransactionFilterChain合约的addFilter接口，将业务权限Filter合约地址注册到Filter合约列表中。
5. 至此，业务权限Filter合约已经启用。

7.2.2 系统参数说明文档

作者：fisco-dev

设计方式

区块链网络通常是由多个网络节点组成的一个分布式系统，有一些参数配置是需要整个分布式系统的各个节点保持一致的，而且后续更新维护也需要做到全网同步，一致更新。 为此，在FISCO BCOS中引入了一种采用智能合约管理系统参数的方法，设计方式如下图所示：



示例输出

系统参数列表

maxBlockTranscations

作用：控制一个块允许打包的最大交易数量上限

范围：(0,2000]

缺省值：1000

intervalBlockTime

作用：控制出块间隔时间（单位：毫秒）

范围：大于等于1000

缺省值：1000

maxBlockHeadGas

作用：控制一个块允许最大Gas消耗上限

范围：大于等于2,000,000,000

缺省值：2,000,000,000

maxTranscationGas

作用：控制一笔交易允许最大Gas消耗上限

范围：大于等于30,000,000

缺省值：30,000,000

maxNonceCheckBlock

作用：控制Nonce排重覆盖的块范围

范围：大于等于1000

缺省值：1000

maxBlockLimit

作用：控制允许交易上链延迟的最大块范围

范围：大于等于1000

缺省值：1000

CAVerify

作用：控制是否打开CA认证

范围：true 或者 false

缺省值：false

omitEmptyBlock

作用：控制是否忽略空块（空块是指无交易的块，忽略空块是指不落盘存储空块）

范围：true 或者 false

缺省值：true

更改系统参数

在创世块节点中执行命令，调用合约接口，更改参数配置（建议在创世块执行，理论上在其他节点也可以操作）

更改系统参数方式为：

```
babel-node tool.js ConfigAction set 【参数名】 【参数值】
```

查看系统参数方式为：

```
babel-node tool.js ConfigAction get 【参数名】
```

举例： 更改出块时间间隔（注意：参数值目前get/set都是以16进制计算表示的）

```
cd systemcontractv2;
babel-node tool.js ConfigAction set  intervalBlockTime  1000
```

举例： 允许空块落盘

```
cd systemcontractv2;
babel-node tool.js ConfigAction set  omitEmptyBlock false
```

[TOC]

7.2.3 CNS(Contract Name Service)服务

一.概述

1.合约调用概述

调用合约的流程包括： 编写合约、编译合约、部署合约。以一个简单的合约HelloWorld.sol为例子：

```
//HelloWorld.sol路径为FISCO-BCOS/tool/HelloWorld.sol
pragma solidity ^0.4.2;
contract HelloWorld{
    string name;
    function HelloWorld(){
       name="Hi,Welcome!";
    }
    function get()constant returns(string){
        return name;
    }
    function set(string n){
        name=n;
    }
}
```

对合约进行编译之后可以获取到合约接口abi的描述,数据如下：

```
[
  {
    "constant": false,
    "inputs": [
      {
        "name": "n",
        "type": "string"
      }
    ],
    "name": "set",
    "outputs": [

    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  },
  {
    "constant": true,
    "inputs": [

    ],
    "name": "get",
    "outputs": [
      {
        "name": "",
        "type": "string"
      }
    ],
    "payable": false,
    "stateMutability": "view",
    "type": "function"
  },
  {
    "inputs": [
```

(continues on next page)

(continued from previous page)

```
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "constructor"
  }
]
```

然后将合约部署到区块链,可以获取到一个地址address,比如：0x269ab4bc23b07efeb3c3fd52eecfc4cbe6a50859。最后使用address结合abi,就可以实现对合约的调用,各种SDK工具的使用方式都有所不同,但是本质上都是对address与abi使用的封装。

2. CNS简介

从合约调用流程可以看出来,调用流程必须的元素是合约ABI以及部署合约地址address。这种方式存在以下的问题:

1. 合约ABI描述是个很长的JSON字符串, 是调用端需要, 对使用者来说并不友好。
2. 合约地址是个魔数, 不方便记忆, 要谨慎维护, 丢失后会导致合约不可访问。
3. 在合约每次重新部署时, 调用方都需要更新合约地址。
4. 不便于进行版本管理以及合约灰度升级。

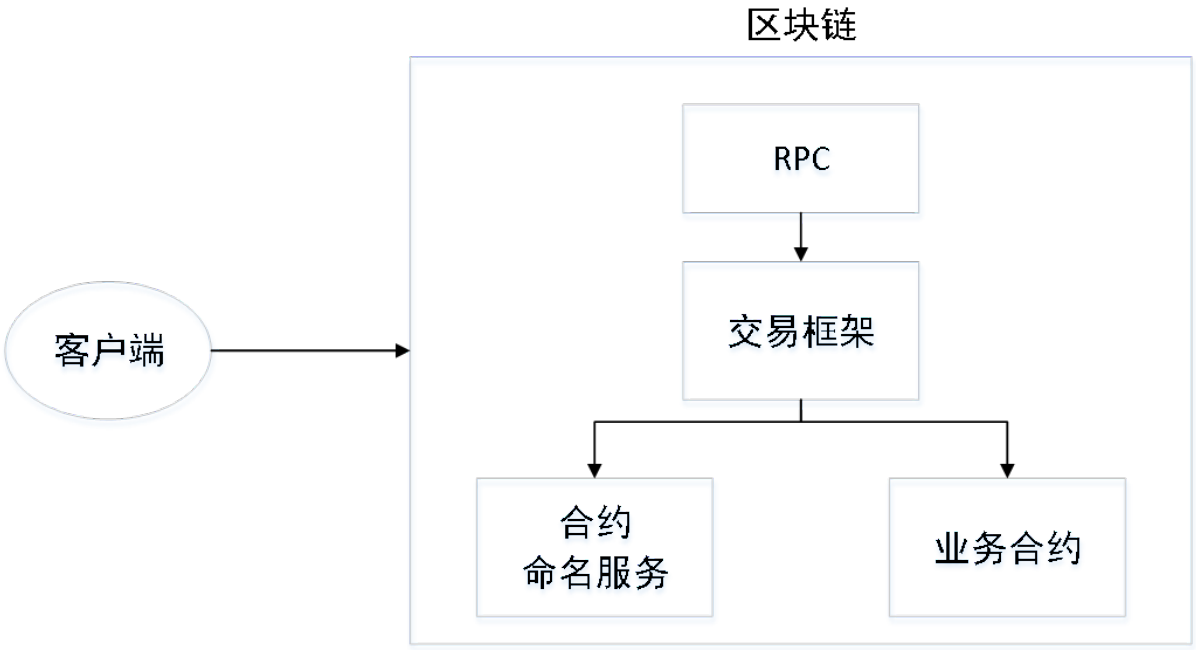
CNS合约命名服务,提供一种由命名到合约接口调用的映射关系。CNS中,调用合约接口时,传入合约映射的name,接口名称以及参数信息。在底层框架的CNS Manager模块维护name与合约信息的映射关系,将根据调用传入的name 、接口 、参数, 转换底层EVM需要的字节码供EVM调用。

下面给出一个图示来说明下CNS服务的优势:

1. 不在需要维护并不友好的合约ABI和合约地址address。
2. 调用方式更简单友好,只需要合约映射的CNS名称,调用接口名称,参数信息。
3. 内置的版本管理特性, 为合约提供了灰度升级的可能性。

二.实现

1. 总体框架



在整个框架中, 命名服务模块提供命名服务, 客户端请求RPC调用合约服务的交易, 交易框架会首先访问合约命名服务模块, 从而解析出要访问的真实合约信息, 构造合约调用需要的信息, 进而对业务合约发出调用，并访问结果给客户端。

2. 主要模块

a. 管理合约模块

在管理合约中保存命名服务中name与合约信息的映射关系,合约具体信息包含合约地址、abi、版本号等,并且提供接口供外部辅助工具cns_manager.js实现添加、更新、覆盖、重置功能。同时,底层交易框架内存中会有一份该合约内容的备份,在该合约内容发生变动时,内存同步更新。

- 当前CNS中实现的映射关系为： 合约名+合约版本号 => 合约详情(abi 合约地址等)
- 合约实现： systemcontractv2/ContractAbiMgr.sol
- 辅助合约： ContractBase.sol(位于tool/ContractBase.sol)
- 对部署的合约进行多版本版本管理,可以让合约继承ContractBase.sol,在构造函数中调用ContractBase.sol的构造函数初始化version成员。
- 注意：ContractAbiMgr合约在系统合约中维护,所以需要使用CNS服务时需要首先部署系统合约。

b. 辅助工具

调用管理合约提供的接口, 提供添加、更新、查询、重置功能。

- 工具: tool/cns_manager.js

```
babel-node cns_manager.js
 cns_manager.js Usage:
        babel-node cns_manager.js get    contractName [contractVersion]
        babel-node cns_manager.js add    contractName
```

(continues on next page)

(continued from previous page)

```
        babel-node cns_manager.js update contractName
        babel-node cns_manager.js list [simple]
        babel-node cns_manager.js historylist contractName [contractVersion]␣
↪[simple]
        babel-node cns_manager.js reset contractName [contractVersion] index
```

功能介绍:

- 命令： add参数： contractName 合约名功能： 添加contractName的信息到管理合约中注意：如果管理合约中contractName对应的信息已经存在,会操作失败。此时可以 1. 更新当前合约的版本号,使用CNS方式调用时,指定版本号 2. 执行update操作,强行覆盖当前信息。

```
//第一次add Test成功
babel-node cns_manager.js add Test
cns add operation => cns_name = Test
        cns_name =>Test
        contract =>Test
        version  =>
        address  =>0x233c777fccb9897ad5537d810068f9c6a4344e4a
        abi      =>[{"constant":false,"inputs":[{"name":"num","type":"uint256"}],
↪"name":"trans","outputs":[],"payable":false,"stateMutability":"nonpayable","type
↪":"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"",
↪"type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{
↪"constant":false,"inputs":[],"name":"Ok","outputs":[],"payable":false,
↪"stateMutability":"nonpayable","type":"function"}]

//第二次add,失败
babel-node cns_manager.js add Test
cns_manager.js  .....................Begin.....................
 [WARNING] cns add operation failed , ====> contract => Test version =>  is␣
↪already exist. you can update it or change its version.
```

- 命令： get参数： 1. contractName 合约名 2. contractVersion 版本号[可省略]功能： 获取contractName对应contractVersion版本在管理合约中的信息

```
babel-node cns_manager.js get HelloWorld
cns_manager.js  .....................Begin.....................
 ====> contract => HelloWorld ,version =>
        contract    = HelloWorld
        version     =
        address     = 0x269ab4bc23b07efeb3c3fd52eecfc4cbe6a50859
        timestamp   = 1516866720115 => 2018/1/25 15:52:0:115
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
```

- 命令： update 参数： contractName 合约名 功能： 更新contractName在管理合约中的信息注意：当contractName对应版本contractVersion在管理合约中不存在时会update失败,此时可以先执行add操作;被覆盖掉的信息可以通过historylist命令查询到,通过reset命令恢复。

```
babel-node cns_manager.js update Test
cns_manager.js  .....................Begin.....................
 ====> Are you sure update the cns of the contract ?(Y/N)
Y
cns update operation => cns_name = Test
        cns_name =>Test
        contract =>Test
        version  =>
        address  =>0x233c777fccb9897ad5537d810068f9c6a4344e4a
```

(continues on next page)

(continued from previous page)

```
        abi       =>[{"constant":false,"inputs":[{"name":"num","type":"uint256"}],
↪"name":"trans","outputs":[],"payable":false,"stateMutability":"nonpayable","type
↪":"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"",
↪"type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{
↪"constant":false,"inputs":[],"name":"Ok","outputs":[],"payable":false,
↪"stateMutability":"nonpayable","type":"function"}]
发送交易成功: 0x1d3caff1fba49f5ad8af3d195999454d01c64d236d9ac3ba91350dd543b10c13
```

- 命令： list 参数：[simple] 功能： 列出管理合约中所有的信息,没有simple参数时,打印合约的详情,否则只打印合约名称跟版本号。

```
babel-node cns_manager.js list simple
cns_manager.js  .......................Begin.......................
 cns total count => 11
        0. contract = ContractAbiMgr ,version =
        1. contract = SystemProxy ,version =
        2. contract = TransactionFilterChain ,version =
        3. contract = AuthorityFilter ,version =
        4. contract = Group ,version =
        5. contract = CAAction ,version =
        6. contract = ConfigAction ,version =
        7. contract = NodeAction ,version =
        8. contract = HelloWorld ,version =
        9. contract = Ok ,version =
        10. contract = Test ,version =
```

- 命令： historylist 参数：contractName 合约名称 contractVersion 合约版本号[可省略] 功能： 列出contrcactName对应版本号contractVersion被update操作覆盖的所有合约信息

```
babel-node cns_manager.js historylist HelloWorld
cns_manager.js  .......................Begin.......................
 cns history total count => 3
 ====> cns history list index = 0 <====
        contract    = HelloWorld
        version     =
        address     = 0x1d2047204130de907799adaea85c511c7ce85b6d
        timestamp   = 1516865606159 => 2018/1/25 15:33:26:159
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
 ====> cns history list index = 1 <====
        contract    = HelloWorld
        version     =
        address     = 0x9c3fb4dd0a3fc5e1ea86ed3d3271b173a7084f24
        timestamp   = 1516866516542 => 2018/1/25 15:48:36:542
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
 ====> cns history list index = 2 <====
        contract    = HelloWorld
        version     =
        address     = 0x1d2047204130de907799adaea85c511c7ce85b6d
        timestamp   = 1516866595160 => 2018/1/25 15:49:55:160
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
```

(continues on next page)

(continued from previous page)

- 命令： reset 参数 : 1. contractName 合约名称 2. contractVersion 合约版本号[可省略] 3. index 索引功能： 将被覆盖的信息恢复,index为historylist查询到的索引

c. RPC接口

底层rpc接口的修改,支持CNS方式的调用：

> 注意： 只是修改了rpc的接口,原来的请求方式仍然兼容。rpc的格式详情参考：https://github.com/ethereum/wiki/wiki/JSON-RPC

- eth_call

```
请求:
{
  "jsonrpc": "2.0",
  "method": "eth_call",
  "params": [
    {
      "data": {
        "contract": "",    //调用合约名称
        "version": "",     //调用合约的版本号
        "func": "",        //调用合约的接口
        "params": [        //参数信息

        ]
      }
    },
    "latest"
  ],
  "id": 1
}

返回:
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "result": [],          //返回结果,格式为json数组
    "ret_code": 0,
    "ret_msg": "success!"
  }
}
```

- eth_sendTransaction

```
请求:
{
  "jsonrpc": "2.0",
  "method": "eth_sendTransaction",
  "params": [
    {
      "data": {
        "contract": "",    //调用合约名称
        "version": "",     //调用合约的版本号
        "func": "",        //调用合约的接口
        "params": [        //参数
        ]
      },
```

(continues on next page)

(continued from previous page)

```
      "randomid": "2"
    }
  ],
  "id": 1
}

返回:
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": "" //交易hash
}
```

- eth_sendRawTransaction rpc请求以及返回格式跟之前完全相同,不同为之前rlp编码data字段为十六进制字符串,现在data的值改为:

```
"data": {
        "contract": "",    //调用合约名称
        "version": "",     //调用合约的版本号
        "func": "",        //调用合约的接口
        "params": [        //参数信息

        ]
      }
```

d. RPC接口JS封装

路径: web3lib/web3sync.js接口:

```
callByNameService
sendRawTransactionByNameService
```

三.使用例子

本模块提供一些CNS一些场景下使用的例子, 供大家参考

```
// 测试合约
// 路径 tool/HelloWorld.sol
pragma solidity ^0.4.4;
contract HelloWorld{
    string name;
    function HelloWorld(){
       name="Hi,Welcome!";
    }
    function get()constant public returns(string){
        return name;
    }
    function set(string n) public{
        name=n;
    }
}
```

- 合约部署:babel-node deploy.js HelloWorld

  在depoy.js中, 合约部署成功时会默认调用cns_mangager add功能, 而且会默认认为文件名与合约名相同, 如果添加失败, 需要后续部署的人自己决策:

1. 实际上文件名与合约名不相同, 重新调用cns_manager add

2. 只是测试合约的部署,不需要处理
3. 同一个合约修改bug或者因为其他原因需要升级, 此时可以执行update操作
4. 当前已经add的合约仍然需要CNS方式调用, 可以修改合约的版本号(参考多版本部署)。

```
    //成功例子
    babel-node deploy.js Test0
    deploy.js  .......................Start.....................
    Soc File :Test0
    Test0编译成功!
    Test0合约地址 0xfc7055a9dc68ff79a58ce4f504d8c780505b2267
    Test0部署成功!
    cns add operation => cns_name = Test0
             cns_name =>Test0
             contract =>Test0
             version  =>
             address  =>0xfc7055a9dc68ff79a58ce4f504d8c780505b2267
             abi      =>[{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{
↪"constant":false,"inputs":[],"name":"HelloWorld","outputs":[],"payable":false,
↪"stateMutability":"nonpayable","type":"function"}]
    发送交易成功: 0x84d1e6b16c58e3571f79e80588472ab8d12779234e75ceed4ac592ad1d653086

    //失败例子,表示该合约已经存在对应信息
    babel-node deploy.js HelloWorld
    deploy.js  .......................Start.....................
    Soc File :HelloWorld
    HelloWorld编译成功!
    HelloWorld合约地址 0xc3869f3d9a5fc728de82cc9c807e85b77259aa3a
    HelloWorld部署成功!
     [WARNING] cns add operation failed , ====> contract => HelloWorld version => ␣
↪is already exist. you can update it or change its version.
```

-多版本部署对于add操作时,因为添加的合约对应的版本号已经存在时, 则会添加失败, 此时可以更新合约的版本号。继承ContractBase.sol指定版本号。

```
pragma solidity ^0.4.4;
contract HelloWorld is ContractBase("v-1.0"){
    string name;
    function HelloWorld(){
       name="Hi,Welcome!";
    }
    function get()constant public returns(string){
        return name;
    }
    function set(string n) public{
        name=n;
    }
}
```

再次部署

```
babel-node deploy.js HelloWorld
deploy.js  .......................Start.....................
Soc File :HelloWorld
HelloWorld编译成功!
HelloWorld合约地址 0x027d156c260110023e5bd918cc243ac12be45b17
HelloWorld部署成功!
cns add operation => cns_name = HelloWorld/v-1.0
```

(continues on next page)

(continued from previous page)

```
        cns_name =>HelloWorld/v-1.0
        contract =>HelloWorld
        version  =>v-1.0
        address  =>0x027d156c260110023e5bd918cc243ac12be45b17
        abi      =>[{"constant":true,"inputs":[],"name":"getVersion","outputs":[{
↪"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":
↪"function"},{"constant":false,"inputs":[{"name":"n","type":"string"}],"name":"set
↪","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function"},
↪{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type":"string"}
↪],"payable":false,"stateMutability":"view","type":"function"},{"constant":false,
↪"inputs":[{"name":"version_para","type":"string"}],"name":"setVersion","outputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
发送交易成功: 0x9a409003f5a17220809dd8e1324a36a425acaf194efd3ef1f772bbf7b49ee67c
```

此时合约版本号为： v-1.0

- RPC调用接口

```
1. 调用HelloWorld默认版本（即没有指定版本号）的set接口
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{
↪"data":{"contract":"HelloWorld","version":"","func":"set","params":["call defaut␣
↪version"]},"randomid":"3"}],"id":1}'  "http://127.0.0.1:8746"

{"id":1,"jsonrpc":"2.0","result":
↪"0x77218708a73aa8c17fb9370a29254baa8f504e71b12d01d90eae0b2ef9818172"}

2. 调用HelloWorld默认版本（即没有指定版本号）的get接口
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"HelloWorld","version":"","func":"get","params":[]}},"latest"],"id":1}
↪'  "http://127.0.0.1:8746"

{"id":1,"jsonrpc":"2.0","result":"[\"call defaut version\"]\n"}

3. 调用HelloWorld v-1.0版本的set接口
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{
↪"data":{"contract":"HelloWorld","version":"v-1.0","func":"set","params":["call v-
↪1.0 version"]},"randomid":"4"}],"id":1}'  "http://127.0.0.1:8746"

{"id":1,"jsonrpc":"2.0","result":
↪"0xf43349d7be554fd332e8e4eb0c69e23292ffa8d127b0500c21109b60784aaa1d"}

4. 调用HelloWorld v-1.0版本的get接口
 curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"HelloWorld","version":"v-1.0","func":"get","params":[]}},"latest"],
↪"id":1}'  "http://127.0.0.1:8746"

{"id":1,"jsonrpc":"2.0","result":"[\"call v-1.0 version\"]\n"}
```

- 合约升级如果合约需要升级的情况下, 可以执行执行update操作。对HelloWorld进行升级, 首先重新部署, 因为HelloWorld之前被cns_manager添加, 所以会提示添加失败, 然后执行update操作。

```
babel-node cns_manager.js update HelloWorld
cns_manager.js  .......................Begin.....................
 ====> Are you sure update the cns of the contract ?(Y/N)
Y
cns update operation => cns_name = HelloWorld
        cns_name =>HelloWorld
        contract =>HelloWorld
        version  =>
        address  =>0x93d62e961a6801d3f614a5add207cdf45b0ff654
```

(continues on next page)

(continued from previous page)

```
        abi      =>[{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
发送交易成功: 0xc8ee384185a1aaa3817474d6db6394ff6871a7bc56a15e564e7b1f57c8bfda1a

再调用get接口:
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"HelloWorld","version":"","func":"get","params":[]}},"latest"],"id":1}
↪'  "http://127.0.0.1:8746"
{"id":1,"jsonrpc":"2.0","result":"[\"Hi,Welcome!\"]\n"}

返回 'Hi,Welcome!'。
说明当前调用的合约就是刚才部署的新合约。
```

- CNS合约重置update之后, 需要将原来的合约找回, 可以通过reset进行。首先查找当前合约对应版本有多少update被覆盖的合约。

```
babel-node cns_manager.js historylist HelloWorld
cns_manager.js  ......................Begin.....................
 cns history total count => 4
 ====> cns history list index = 0 <====
        contract    = HelloWorld
        version     =
        address     = 0x1d2047204130de907799adaea85c511c7ce85b6d
        timestamp   = 1516865606159 => 2018/1/25 15:33:26:159
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
 ====> cns history list index = 1 <====
        contract    = HelloWorld
        version     =
        address     = 0x9c3fb4dd0a3fc5e1ea86ed3d3271b173a7084f24
        timestamp   = 1516866516542 => 2018/1/25 15:48:36:542
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
 ====> cns history list index = 2 <====
        contract    = HelloWorld
        version     =
        address     = 0x1d2047204130de907799adaea85c511c7ce85b6d
        timestamp   = 1516866595160 => 2018/1/25 15:49:55:160
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
 ====> cns history list index = 3 <====
        contract    = HelloWorld
        version     =
        address     = 0x269ab4bc23b07efeb3c3fd52eecfc4cbe6a50859
        timestamp   = 1516866720115 => 2018/1/25 15:52:0:115
        abi         = [{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
```

(continues on next page)

(continued from previous page)

```
然后查看需要被找回的合约是哪个。
babel-node cns_manager.js reset HelloWorld 3
cns_manager.js  .......................Begin.....................
 ====> Are you sure update the cns of the contract ?(Y/N)
Y
cns update operation => cns_name = HelloWorld
         cns_name =>HelloWorld
         contract =>HelloWorld
         version  =>
         address  =>0x269ab4bc23b07efeb3c3fd52eecfc4cbe6a50859
         abi      =>[{"constant":false,"inputs":[{"name":"n","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":true,"inputs":[],"name":"get","outputs":[{"name":"","type
↪":"string"}],"payable":false,"stateMutability":"view","type":"function"},{"inputs
↪":[],"payable":false,"stateMutability":"nonpayable","type":"constructor"}]
发送交易成功: 0x4809a6105916a483ca70c4efe8e306bc01ca5d937515320d09e94a83f4a91b76

此时再调用HelloWorld的get接口:
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"HelloWorld","version":"","func":"get","params":[]}},"latest"],"id":1}
↪'  "http://127.0.0.1:8746"
{"id":1,"jsonrpc":"2.0","result":"[\"call defaut version\"]\n"}

返回为call defaut version, 说明当前CNS调用的合约已经是最后一次比覆盖的合约。
```

- js调用

```
//调用HelloWorld get接口
var result = web3sync.callByNameService("HelloWorld","get","",[]);

//调用HelloWorld v-1.0 get接口
var result = web3sync.callByNameService("HelloWorld","get","v-1.0",[]);

//调用HelloWorld set接口 sendRawTransaction
var result = web3sync.sendRawTransactionByNameService(config.account,config.
↪privKey,"HelloWorld","set","",["test message!"]);

//调用HelloWorld v-1.0 set接口 sendRawTransaction
var result = web3sync.sendRawTransactionByNameService(config.account,config.
↪privKey,"HelloWorld","set","v-1.0",["test message!"]);
```

附录一. 重载函数的调用

solidity支持函数重载, 当solidity中存在重载函数时, 使用CNS调用的参数跟之前有所不同:

```
//file : OverloadTest.sol
pragma solidity ^0.4.4;
contract OverloadTest {
    string public msg;
    uint256 public u;

    function OverloadTest() {
        msg = "OverloadTest Test";
        u = 0x01;
    }

    function set(string _msg) public {
        msg = _msg;
    }
```

(continues on next page)

(continued from previous page)

```
    function set(uint256 _u) public {
        u = _u;
    }

    function get() public constant returns(string){
        return msg;
    }

    function get(uint256 i) public constant returns(uint256){
        return u;
    }
}
```

在OverloadTest.sol合约中:set函数是一个重载函数, 一个函数原型为set(string),另一个为set(uint256).get函数也是一个重载函数, 一个函数原型为get(), 另一个为get(uint256).

部署合约:

```
babel-node deploy.js OverloadTest
RPC=http://0.0.0.0:8546
Ouputpath=./output/
deploy.js  .......................Start.......................
OverloadTest编译成功!
发送交易成功: 0xff8a5708b3f7b335570a50639f2073e5e0b8b2002faa909dc75727059de94f4e
OverloadTest合约地址 0x919868496524eedc26dbb81915fa1547a20f8998
OverloadTest部署成功!
cns add operation => cns_name = OverloadTest
         cns_name =>OverloadTest
         contract =>OverloadTest
         version  =>
         address  =>0x919868496524eedc26dbb81915fa1547a20f8998
         abi      =>[{"constant":false,"inputs":[{"name":"_msg","type":"string"}],
↪"name":"set","outputs":[],"payable":false,"type":"function"},{"constant":false,
↪"inputs":[{"name":"_u","type":"uint256"}],"name":"set","outputs":[],"payable
↪":false,"type":"function"},{"constant":true,"inputs":[],"name":"msg","outputs":[{
↪"name":"","type":"string"}],"payable":false,"type":"function"},{"constant":true,
↪"inputs":[],"name":"get","outputs":[{"name":"","type":"string"}],"payable":false,
↪"type":"function"},{"constant":true,"inputs":[{"name":"i","type":"uint256"}],
↪"name":"get","outputs":[{"name":"","type":"uint256"}],"payable":false,"type":
↪"function"},{"constant":true,"inputs":[],"name":"u","outputs":[{"name":"","type":
↪"uint256"}],"payable":false,"type":"function"},{"inputs":[],"payable":false,"type
↪":"constructor"}]
===>> namecall params = {"contract":"ContractAbiMgr","func":"addAbi","version":"",
↪"params":["OverloadTest","OverloadTest","","[{\"constant\":false,\"inputs\":[{\
↪"name\":\"_msg\",\"type\":\"string\"}],\"name\":\"set\",\"outputs\":[],\"payable\
↪":false,\"type\":\"function\"},{\"constant\":false,\"inputs\":[{\"name\":\"_u\",\
↪"type\":\"uint256\"}],\"name\":\"set\",\"outputs\":[],\"payable\":false,\"type\
↪":\"function\"},{\"constant\":true,\"inputs\":[],\"name\":\"msg\",\"outputs\":[{\
↪"name\":\"\",\"type\":\"string\"}],\"payable\":false,\"type\":\"function\"},{\
↪"constant\":true,\"inputs\":[],\"name\":\"get\",\"outputs\":[{\"name\":\"\",\
↪"type\":\"string\"}],\"payable\":false,\"type\":\"function\"},{\"constant\":true,
↪\"inputs\":[{\"name\":\"i\",\"type\":\"uint256\"}],\"name\":\"get\",\"outputs\":[
↪{\"name\":\"\",\"type\":\"uint256\"}],\"payable\":false,\"type\":\"function\"},{\
↪"constant\":true,\"inputs\":[],\"name\":\"u\",\"outputs\":[{\"name\":\"\",\"type\
↪":\"uint256\"}],\"payable\":false,\"type\":\"function\"},{\"inputs\":[],\
↪"payable\":false,\"type\":\"constructor\"}]",
↪"0x919868496524eedc26dbb81915fa1547a20f8998"]}
发送交易成功: 0x56e2267cd46fddc11abc4f38d605adc1f76d3061b96cf4026b09ace3502d2979
```

对于重载函数, 在使用CNS方式调用时, func参数需要指定完整的函数原型, 不能仅仅只指定函数的名称:

调用get()时, “func”:”get()”调用get(uint256 i)时, “func”:”get(uint256)”调用set(string _msg)时, “func”:”set(string)”调用set(uint256 _u)时, “func”:”set(uint256)”

下面是调用的示例:

```
调用get()接口:
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"OverloadTest","version":"","func":"get","params":[]}},"latest"],"id
↪":1}'  "http://127.0.0.1:8546"
{"id":1,"jsonrpc":"2.0","result":"[\"OverloadTest Test\"]\n"}

调用get(uint256 i)接口:
curl -X POST --data  '{"jsonrpc":"2.0","method":"eth_call","params":[{"data":{
↪"contract":"OverloadTest","version":"","func":"get(uint256)","params":[1]}},
↪"latest"],"id":1}'  "http://127.0.0.1:8546"
{"id":1,"jsonrpc":"2.0","result":"[\"1\"]\n"}

使用js调用set(string _msg):
var result = web3sync.sendRawTransactionByNameService(config.account,config.
↪privKey,"OverloadTest","set(string)","",["test message!"]);

使用js调用set(uint256 _i)):
var result = web3sync.sendRawTransactionByNameService(config.account,config.
↪privKey,"OverloadTest","set(uint256)","",["0x111"]);
```

附录二. java客户端的使用.

我们还是以上述HelloWorld.sol合约为例,提供一个完整的例子。

1. 参考上面的流程, 部署HelloWorld.sol合约, 使用cns_manager.js工具注册HelloWorld合约信息到CNS管理合约。
2. 下载web3sdk, 版本号需要>=V1.1.0, web3sdk的使用以及下载：https://github.com/FISCO-BCOS/web3sdk
3. 使用web3sdk生成HelloWorld的java wrap代码参考教程。在示例中使用HelloWorld.sol生成的java代码的包名为:org.bcos.cns, 生成的代码为:

```
package org.bcos.cns;

import java.math.BigInteger;
import java.util.Arrays;
import java.util.Collections;
import java.util.concurrent.Future;
import org.bcos.channel.client.TransactionSucCallback;
import org.bcos.web3j.abi.TypeReference;
import org.bcos.web3j.abi.datatypes.Function;
import org.bcos.web3j.abi.datatypes.Type;
import org.bcos.web3j.abi.datatypes.Utf8String;
import org.bcos.web3j.crypto.Credentials;
import org.bcos.web3j.protocol.Web3j;
import org.bcos.web3j.protocol.core.methods.response.TransactionReceipt;
import org.bcos.web3j.tx.Contract;
import org.bcos.web3j.tx.TransactionManager;

/**
 * Auto generated code.<br>
 * <strong>Do not modify!</strong><br>
 * Please use the <a href="https://docs.web3j.io/command_line.html">web3j command␣
↪line tools</a>, or {@link org.bcos.web3j.codegen.
↪SolidityFunctionWrapperGenerator} to update.
 *
```

(continues on next page)

(continued from previous page)

```
 * <p>Generated with web3j version none.
 */
public final class HelloWorld extends Contract {
    private static final String BINARY =
↪"6060604052341561000c57fe5b5b60406040519081016040528060b81526020017f48692c57656c636f6d652100000
↪";

    public static final String ABI = "[{\"constant\":false,\"inputs\":[{\"name\":\
↪"n\",\"type\":\"string\"}],\"name\":\"set\",\"outputs\":[],\"payable\":false,\
↪"type\":\"function\"},{\"constant\":true,\"inputs\":[],\"name\":\"get\",\
↪"outputs\":[{\"name\":\"\",\"type\":\"string\"}],\"payable\":false,\"type\":\
↪"function\"},{\"inputs\":[],\"payable\":false,\"type\":\"constructor\"}]";

    private HelloWorld(String contractAddress, Web3j web3j, Credentials␣
↪credentials, BigInteger gasPrice, BigInteger gasLimit, Boolean isInitByName) {
        super(BINARY, contractAddress, web3j, credentials, gasPrice, gasLimit,␣
↪isInitByName);
    }

    private HelloWorld(String contractAddress, Web3j web3j, TransactionManager␣
↪transactionManager, BigInteger gasPrice, BigInteger gasLimit, Boolean␣
↪isInitByName) {
        super(BINARY, contractAddress, web3j, transactionManager, gasPrice,␣
↪gasLimit, isInitByName);
    }

    private HelloWorld(String contractAddress, Web3j web3j, Credentials␣
↪credentials, BigInteger gasPrice, BigInteger gasLimit) {
        super(BINARY, contractAddress, web3j, credentials, gasPrice, gasLimit,␣
↪false);
    }

    private HelloWorld(String contractAddress, Web3j web3j, TransactionManager␣
↪transactionManager, BigInteger gasPrice, BigInteger gasLimit) {
        super(BINARY, contractAddress, web3j, transactionManager, gasPrice,␣
↪gasLimit, false);
    }

    public Future<TransactionReceipt> set(Utf8String n) {
        Function function = new Function("set", Arrays.<Type>asList(n),␣
↪Collections.<TypeReference<?>>emptyList());
        return executeTransactionAsync(function);
    }

    public void set(Utf8String n, TransactionSucCallback callback) {
        Function function = new Function("set", Arrays.<Type>asList(n),␣
↪Collections.<TypeReference<?>>emptyList());
        executeTransactionAsync(function, callback);
    }

    public Future<Utf8String> get() {
        Function function = new Function("get",
                Arrays.<Type>asList(),
                Arrays.<TypeReference<?>>asList(new TypeReference<Utf8String>() {}
↪));
        return executeCallSingleValueReturnAsync(function);
    }

    public static Future<HelloWorld> deploy(Web3j web3j, Credentials credentials,␣
↪BigInteger gasPrice, BigInteger gasLimit, BigInteger initialWeiValue) {
        return deployAsync(HelloWorld.class, web3j, credentials, gasPrice,␣
↪gasLimit, BINARY, "", initialWeiValue);
```

(continues on next page)

(continued from previous page)

```
    }

    public static Future<HelloWorld> deploy(Web3j web3j, TransactionManager␣
↪transactionManager, BigInteger gasPrice, BigInteger gasLimit, BigInteger␣
↪initialWeiValue) {
        return deployAsync(HelloWorld.class, web3j, transactionManager, gasPrice,␣
↪gasLimit, BINARY, "", initialWeiValue);
    }

    public static HelloWorld load(String contractAddress, Web3j web3j, Credentials␣
↪credentials, BigInteger gasPrice, BigInteger gasLimit) {
        return new HelloWorld(contractAddress, web3j, credentials, gasPrice,␣
↪gasLimit, false);
    }

    public static HelloWorld load(String contractAddress, Web3j web3j,␣
↪TransactionManager transactionManager, BigInteger gasPrice, BigInteger gasLimit)
↪{
        return new HelloWorld(contractAddress, web3j, transactionManager, gasPrice,
↪ gasLimit, false);
    }

    public static HelloWorld loadByName(String contractName, Web3j web3j,␣
↪Credentials credentials, BigInteger gasPrice, BigInteger gasLimit) {
        return new HelloWorld(contractName, web3j, credentials, gasPrice, gasLimit,
↪ true);
    }

    public static HelloWorld loadByName(String contractName, Web3j web3j,␣
↪TransactionManager transactionManager, BigInteger gasPrice, BigInteger gasLimit)
↪{
        return new HelloWorld(contractName, web3j, transactionManager, gasPrice,␣
↪gasLimit, true);
    }
}
```

在生成的代码中多了两个loadByName函数。

1. 函数调用

```
package org.bcos.main;

import java.math.BigInteger;
import java.util.concurrent.Future;

import org.bcos.channel.client.Service;
import org.bcos.cns.HelloWorld;
import org.bcos.web3j.abi.datatypes.Utf8String;
import org.bcos.web3j.crypto.Credentials;
import org.bcos.web3j.crypto.ECKeyPair;
import org.bcos.web3j.crypto.Keys;
import org.bcos.web3j.protocol.Web3j;
import org.bcos.web3j.protocol.channel.ChannelEthereumService;
import org.bcos.web3j.protocol.core.methods.response.TransactionReceipt;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
        public static void main(String[] args) throws Exception {

                ApplicationContext context = new ClassPathXmlApplicationContext(
↪"classpath:applicationContext.xml");
```

(continues on next page)

(continued from previous page)

```
                //init service
                Service service = context.getBean(Service.class);
        ChannelEthereumService channelEthereumService = new␣
↪ChannelEthereumService();
        channelEthereumService.setTimeout(10000);
        channelEthereumService.setChannelService(service);

        //init web3
        Web3j web3j = Web3j.build(channelEthereumService);
        service.run();

        //初始化交易签名私钥
        ECKeyPair keyPair = Keys.createEcKeyPair();
        Credentials credentials = Credentials.create(keyPair);

            BigInteger gasPrice = new BigInteger("99999999");
            BigInteger gasLimit = new BigInteger("99999999");

            //通过loadByName方式构建合约对象时, 后面通过合约对象调用合约接口时, 会以CNS的方式
调用
            HelloWorld instance = HelloWorld.loadByName("HelloWorld", web3j,␣
↪credentials, gasPrice , gasLimit);

            //调用HelloWorld set接口
            Future<TransactionReceipt> receiptResult = instance.set(new Utf8String(
↪"HelloWorld Test."));
            receiptResult.get();

            //调用HelloWorld get接口
                Future<Utf8String> result = instance.get();
                System.out.println("HelloWorld get result = " + result.get().
↪toString());


                return;
        }
}
```

通过loadByName方式构建合约对象时, 后面通过合约对象调用合约接口时, 会以CNS的方式调用。

> HelloWorld instance = HelloWorld.loadByName("HelloWorld", web3j, credentials, gasPrice , gasLimit);

HelloWorld的合约对象通过loadByName方式构建, 所以后续的get跟set的调用都是以CNS方式进行调用的。

- 说明：对于合约XX.sol生成的java Wrap代码中的loadByName原型如下：

```
 public static XX loadByName(String contractName, Web3j web3j, Credentials␣
↪credentials, BigInteger gasPrice, BigInteger gasLimit) {
        return new XX(contractName, web3j, credentials, gasPrice, gasLimit, true);
    }

public static XX loadByName(String contractName, Web3j web3j, TransactionManager␣
↪transactionManager, BigInteger gasPrice, BigInteger gasLimit) {
        return new XX(contractName, web3j, transactionManager, gasPrice, gasLimit,␣
↪true);
    }
```

其中, contractname参数格式为：合约的名称@合约版本号, 如果合约没有版本号, 则为合约的名称。

1. 总结 使用java客户端调用CNS的步骤为:a. 使用JS工具部署合约.b. 使用cns_nameger.js工具注册合约信息到CNS管理合约.c. 使用websdk工具生成合约的java Wrap代码.d. 将生成的代码加入自己的工

程, 通过loadByName接口构造合约对象.e. 调用合约接口.

7.3 性能

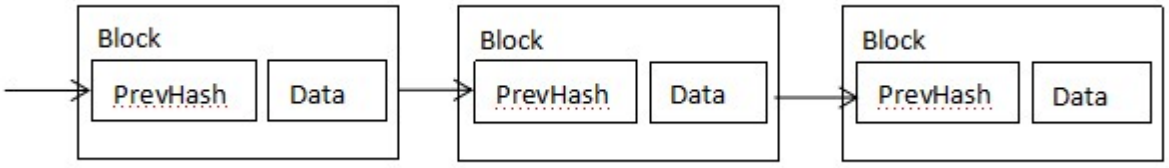
7.3.1 应用于区块链的多节点并行拜占庭容错共识算法

作者：**fisco-dev**

1. 缩略语和关键术语定义

区块链：

区块链是由一系列区块组成的一条链，每个块上除了记录本块的数据还会记录上一块的Hash值，通过这种方式组成一条链。区块链的核心理念有两个：一个是密码学技术，另一个是去中心化思想，基于这两个理念做到区块链上的历史信息无法被篡改。一个区块由块头和块体组成，其中块头定义包括该区块高度h，上一个区块的hash值prevHash等重要字段，而块体主要存储交易数据。



对等网络：

与传统中心化网络不同，对等网络指的是参与者通过P2P协议组成网络，参与者都是对等的。对等网络具有以下几个特征：

1. 非中心化：不需要一个中心服务器，资源和服务分散个各个节点上，所有的数据传输和服务的实现都在节点之间进行。
2. 健壮性强：节点可以随意加入、退出网络，不会对服务造成任何影响。
3. 可扩展性强：支持扩展节点，从而扩展系统。譬如基于P2P协议的文件下载，加入用户越多下载速度越快。
4. 高性价比：参与P2P网络的节点一般是普通机器，构建出的整个网络系统能提供工业级的服务，具备相当高的成本优势。

节点：

对等网络中的每一个参与者就是一个节点，节点参与网络组建和数据交换。在区块链对等网络中，一个节点是指一个具有唯一身份的参与者，该节点具有一份完整的账本拷贝，具有参与区块链对等网络共识和账本维护的能力。

共识算法：

区块链对等网络中的各个节点通过一种算法对一批交易进行确认，并确保所有节点对这批数据具有一致的确认结果，这种算法就是区块链的共识算法。

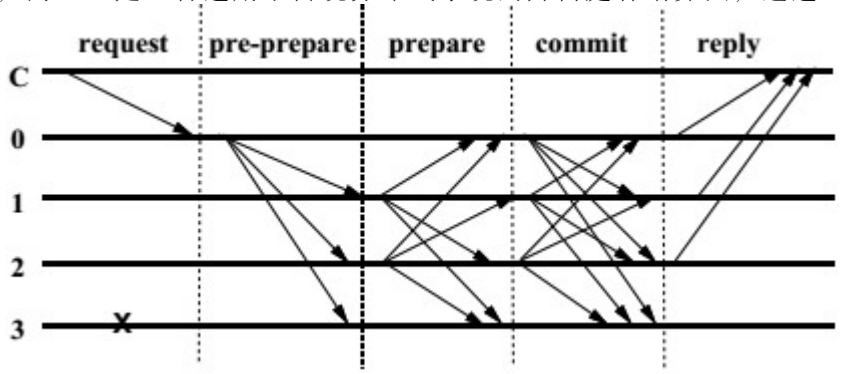
拜占庭容错:

拜占庭容错来源于拜占庭将军问题，在一个对等网络系统中节点可能会以任意形式运行，或者联合起来估计作恶。只要这种故障节点数量在一定范围内，该系统仍然能正常运行，那么就称该系统具有拜占庭容错。

2. 传统共识算法介绍

2.1传统技术方案

现有区块链的共识算法主要包括工作量证明（POW）、权益证明（POS）、委托权益证明（DPOS）以及可用拜占庭容错算法（PBFT）等，其中POW、POS、DPOS主要适用于比特币等公有链，而PBFT是一种适用于传统分布式系统的拜占庭容错算法，通过三轮广播通信完成共识算法。



2.2传统共识算法不足

- POW通过算力竞争获得共识，造成大量能源消耗，而且这种算法会导致出块时间不稳定；
- POS、DPOS需要通过代币数量来控制共识，容易造成代币集中化，使得共识被少数人控制，少数人可以联合作恶破坏网络；
- PBFT是一种可用的拜占庭容错算法，但是由于该算法的三个阶段是串行执行，存在共识效率低的问题。

3. 应用于区块链的多节点并行拜占庭容错共识算法

3.1 节点角色

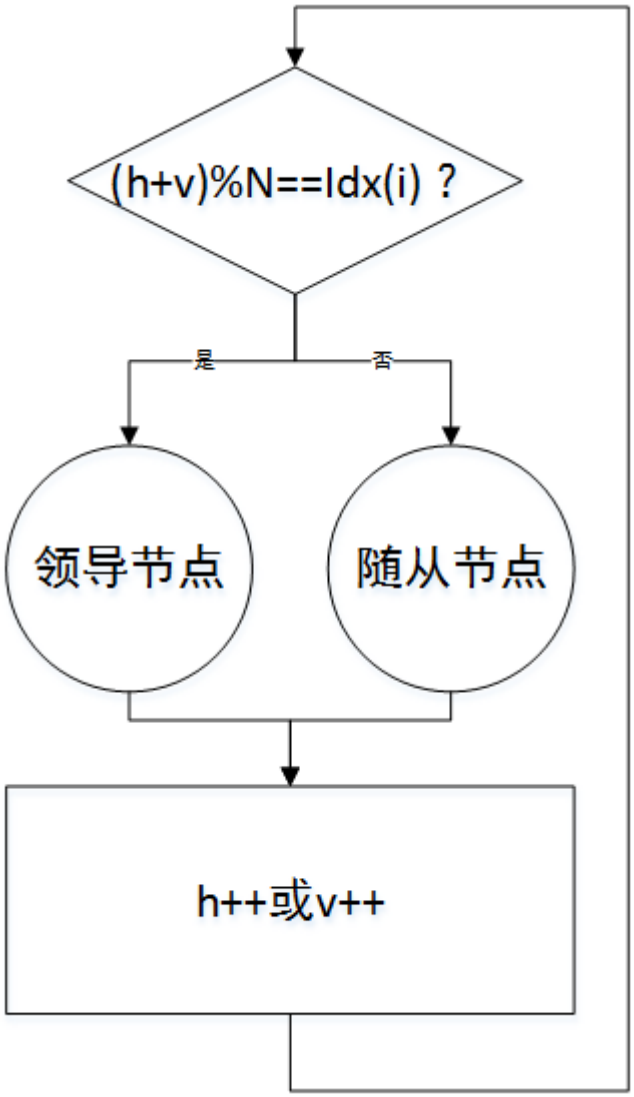
区块链节点的角色有两种，分别是“领导节点”和“随从节点”。

- 领导节点：负责对交易进行打包成块，把块广播给其他节点，通过共识过程对块中所有交易进行确认，从而使得区块链的区块高度不断增加。
- 随从节点：负责接收从领导节点发送来的区块，对区块中的交易进行确认，所有交易都确认完毕就对该块进行签名验证，从而使共识达成。

3.2 角色变迁

在本算法中，节点的角色不是固定不变的，随着时间迁移节点角色也会进行变迁。区块链网络由一个个节点组成，假设一共有N个节点，对节点从0,1,2...N-1进行编号，每个节点对应一个唯一的Idx(i)。一个节点的角色判断通过公式（h+v）%N来决定，其中h是区

块链当前块高度，v是当前视图（视图的定义在3.4节会详细阐述）。角色变迁图如下所

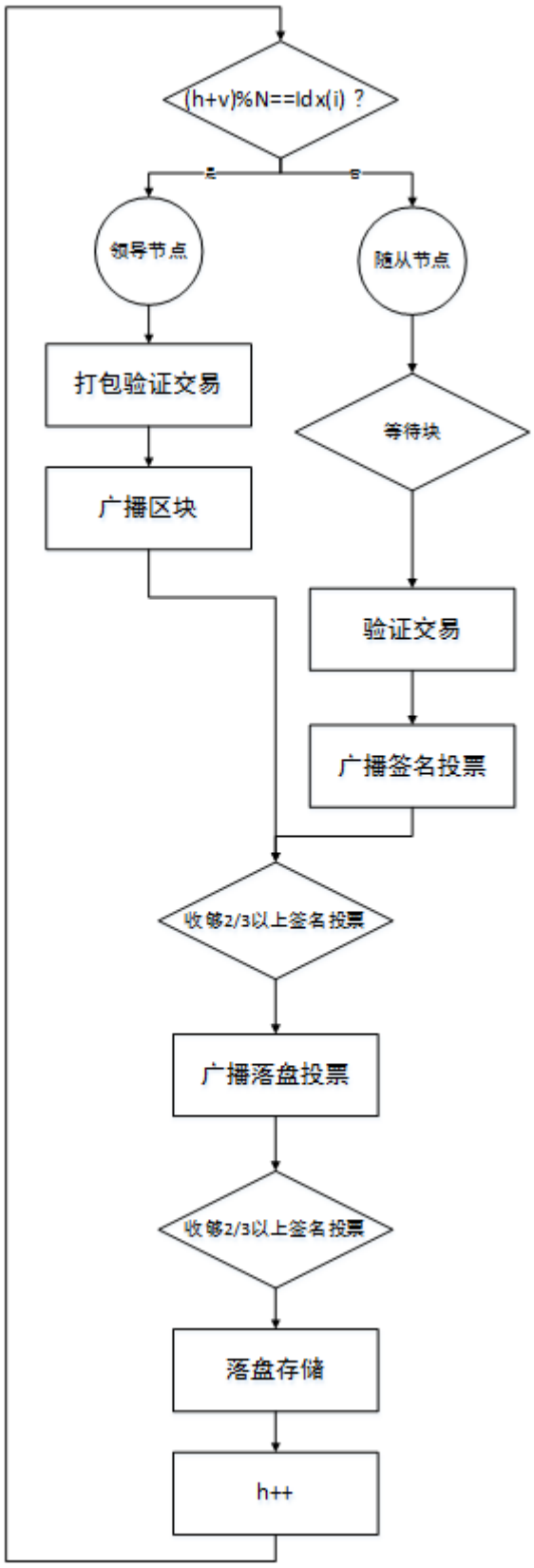


示：

3.3 共识过程

共识过程就是区块链网络对一批交易进行确认，并达到全网一致的过程。共识过程分为以下几个阶段：

1. 选举领导：通过3.2描述的算法推选出一个领导，有别于其他基于投票选举领导的算法，在本专利中是通过共识计算选出合适的领导，这种方式具有更高的效率。
2. 打包验证交易：选举出的领导节点，将会一批交易进行打包验证，组成一个区块，区块的产生也就由领导节点负责。
3. 签名投票：随从节点对领导节点发送来的区块，进行每一笔交易确认验证，全部通过之后发送对该块的一个投票签名。
4. 落盘投票：所有节点在收到2/3以上节点的签名投票之后，广播落盘投票。
5. 落盘提交：所有节点在收到2/3以上节点额落盘投票之后，把该块进行落盘存

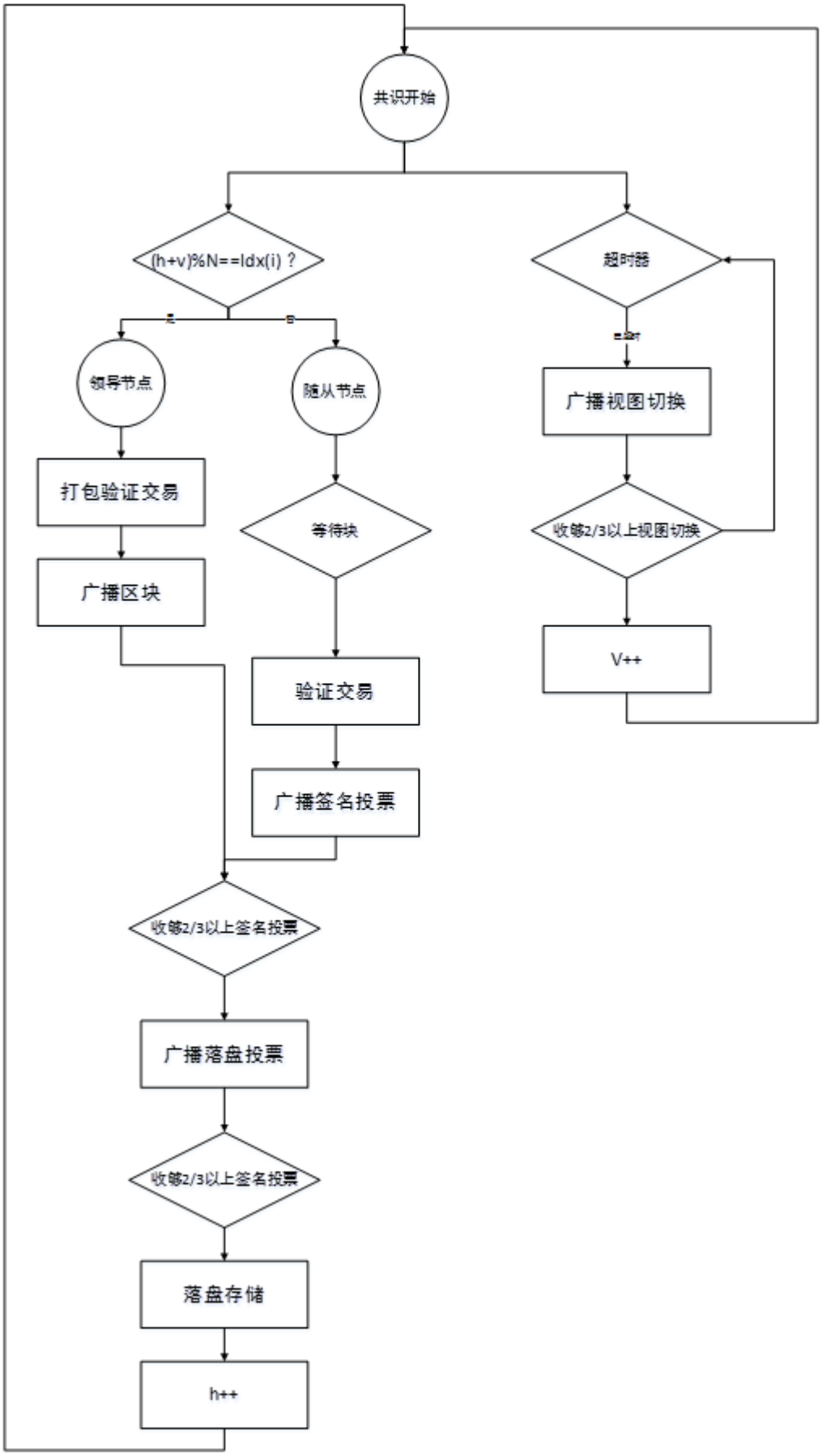


储。

3.4 异常处理机制

在3.3的描述的共识过程几个阶段，每个阶段都有可能因为出现错误、超时或者故意作恶等各种原因致使无法顺利进入下一个阶段，从而使共识无法达成。本专利引入异常处理机制解决这种问题。把

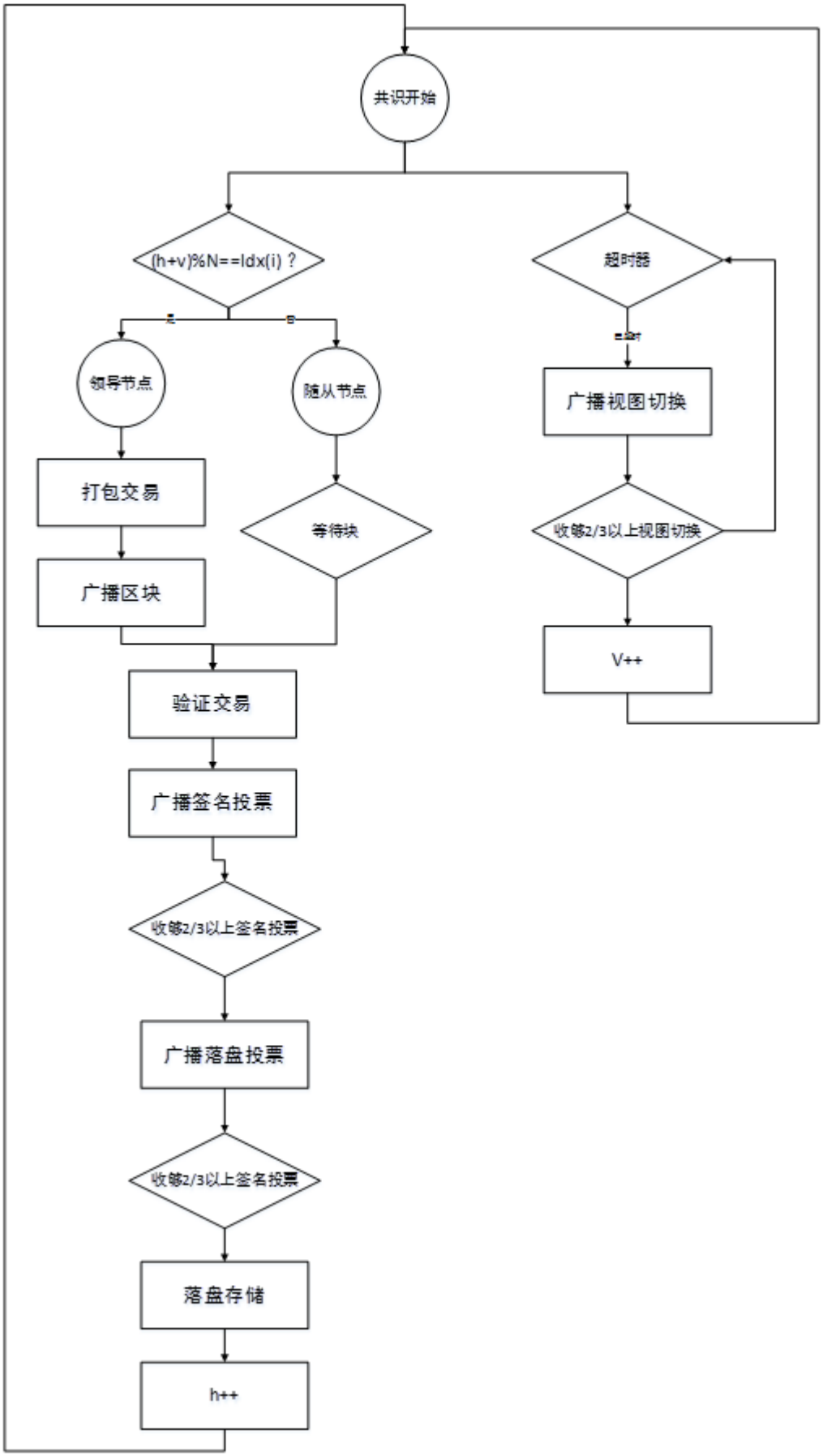
一次共识共识的全过程定义为一个视图，所有阶段需要在同一个视图下完成。当一个节点完成块h的落盘存储之后，意味着它就需要开始块h+1的共识过程，此时会对块h+1的共识设置一个超时器，当到达超时还未完成共识过程就会引起视图切换过程。视图切换的过程首先是将自己的视图v++，然后把v全网广播告知所有节点，如果收到2/3以上节点都有相同的视图v切换请求，就顺利切换到下一个视


共识开始
(h+v)%N==Idx(i) ?
超时器
领导节点
随从节点
广播视图切换
打包验证交易
等待块
收够2/3以上视图切换
广播区块
V++
验证交易
广播签名投票
收够2/3以上签名投票
广播落盘投票
收够2/3以上签名投票
落盘存储
h++

图。

3.5 并行机制

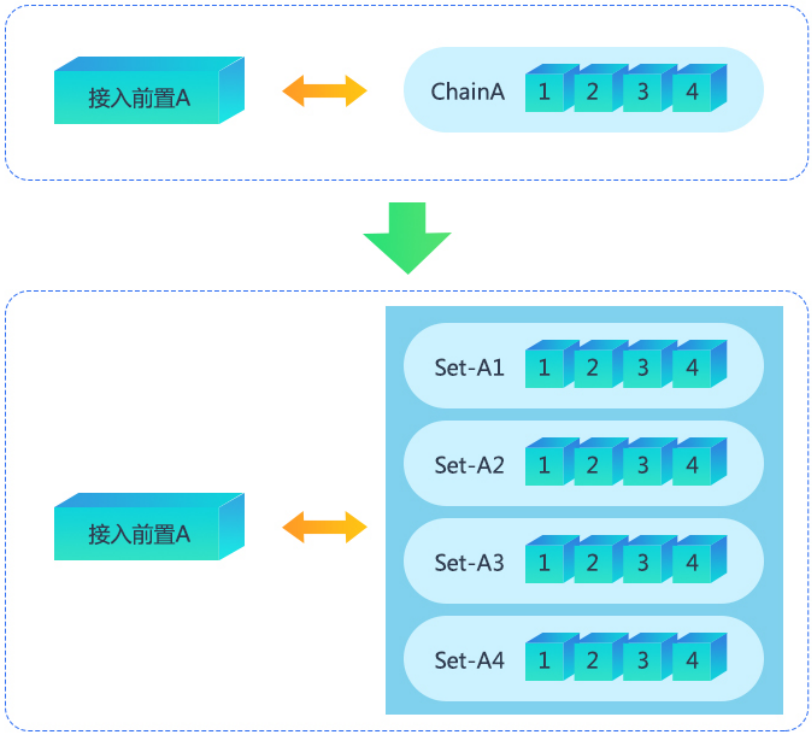
在3.3介绍的共识过程中，打包验证交易和验证交易分别是领导节点和随从节点对交易进行确认的操作，这是整个共识过程中最耗时的环节。从图中可以看出，打包验证交易和验证交易是串行执行的，首先要由领导节点完成打包验证交易，随从节点的验证交易才能开始进行，假设交易确认耗时为T，其他过程总耗时为T'，那么整个共识的耗时就为2*T+T'。本专利对交易确认机制提出并行化的改进设计，整体共识耗时降为T+T'，大大提高了共识效


共识开始
(h+v)%N==Idx(i) ?
超时器
领导节点
随从节点
广播视图切换
打包交易
等待块
收够2/3以上视图切换
广播区块
V++
验证交易
广播签名投票
收够2/3以上签名投票
广播落盘投票
收够2/3以上签名投票
落盘存储
h++

率。

7.3.2 并行计算和热点账户解决方案

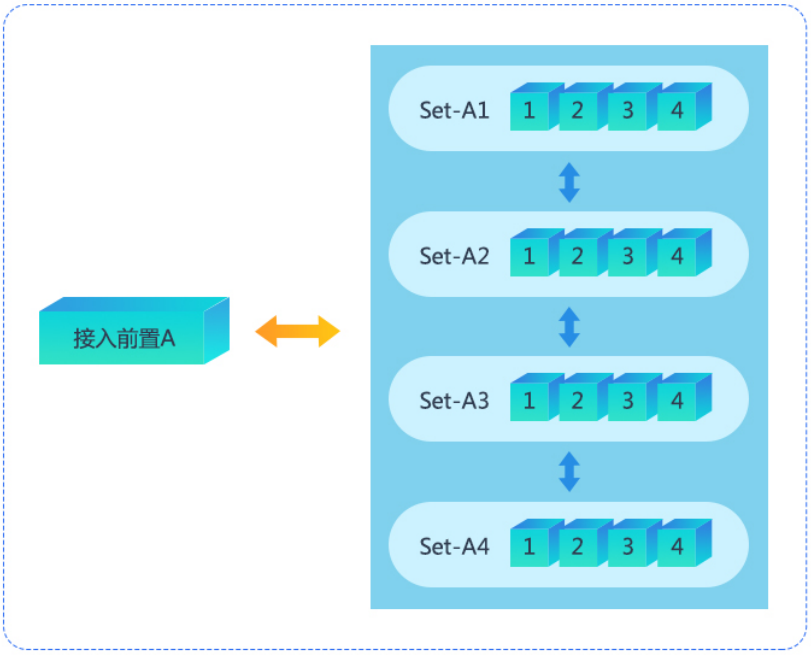
作者：fisco-dev在研究和实现区块链平台和进行业务落地的过程中，我们意识到，区块链的运行速度会受多种因素影响，包括加密解密计算、交易广播和排序、共识算法多阶段提交的协作开销、虚拟机执行速度等，以及受CPU核数主频、磁盘IO、网络带宽等硬件性能影响。由于区块链是先天的跨网络的分布式协作系统，而且强调安全性、可用性、容错性、一致性、事务性，用较复杂的算法和繁琐的多参与方协作来获得去信任化、数据不可篡改以及交易可追溯等特出的功能优势，根据分布式的CAP原理，在同等的硬件资源投入的前提下，区块链的性能往往低于中心化的系统，其表现就是并发数不高，交易时延较明显。 我们已经在多个方面对系统运行的全流程进行细致的优化，包括加密解密计算，交易处理流程，共识算法，存储优化等，使我们的区块链平台在单链架构时，运行速度达到了一个较高的性能水准，基本能满足一般的金融业务要求。同时我们也意识到，对于用户数、交易量、存量数据较大或可能有显著增长的海量服务场景，对系统提出了更高的容量和扩展性要求，单链架构总是会遇到软件架构或硬件资源方面的瓶颈。而区块链的系统特性决定，在区块链中增加节点，只会增强系统的容错性，增加参与者的授信背书等，而不会增加性能，只增加节点不能解决问题，这就需要通过架构上的调整来应对性能挑战，所以，我们提出了“并行计算，多链运行”的方案。并行多链的架构基本思路是在一个区块链网络里，存在多个分组，每个组是一个完整的区块链网络，有独立的软件模块，硬件资源，独立完成机构间共识，有独立的数据存储。根据可定制的路由规则，参与到区块链网络的所有机构和用户，或者区块链里不同类型的交易，可以接入到不同的分组里，每个分组处理特定的一部分交易，当机构或用户数增加，交易量变大或者交易类型增加，都可以快捷的增加分组，并在路由策略里进行设定，将新增的流量分配到新的分组里。并行多链架构类似数据库的分库分表，或者互联网服务的分SET模型，理论上只要投入足够的资源，则系统能处理的流量没有上限，整个系统具有足够的弹性。同时，一个区块链网络里的多个分组秉承逻辑和配置高一致性的原则，在商业规则、运营管理上都使用统一的策略，比如，每个分组上的智能合约是完全相同的，核心配置数据也是相同的，只有分组里的机构、用户以及交易类型有所不同。或者，虽然因为分组间功能设计的差异，导致不同分组上的智能合约有所不同，如一些分组是处理用户在线交易，强调低时延性，其他分组处理机构间的对账和清结算，关注批量数据处理，那么部署在这些分组上的智能合约会有所不同，但都会通过所有机构以及区块链的运营委员会共同确认，通过共识算法保证部署实施的一致性，公开性，不可篡改性。总之，平台提供了基础的分组的策略和实现、路由模块、并行多链的构建工具等，如何根据业务场景设计不同的分组，如根据机构维度，用户维度还是交易维度，甚至是时间维度等，都可以再进行灵活的设计和操作。



并行计算多链架构

在实现了区块链分组后，分组之间有可能出现互相发生交易的场景，实际上就是不同区块链系统之间的通信和交易，类似“跨链”的架构。在这个环节，需要关注的是分组间的通信可靠

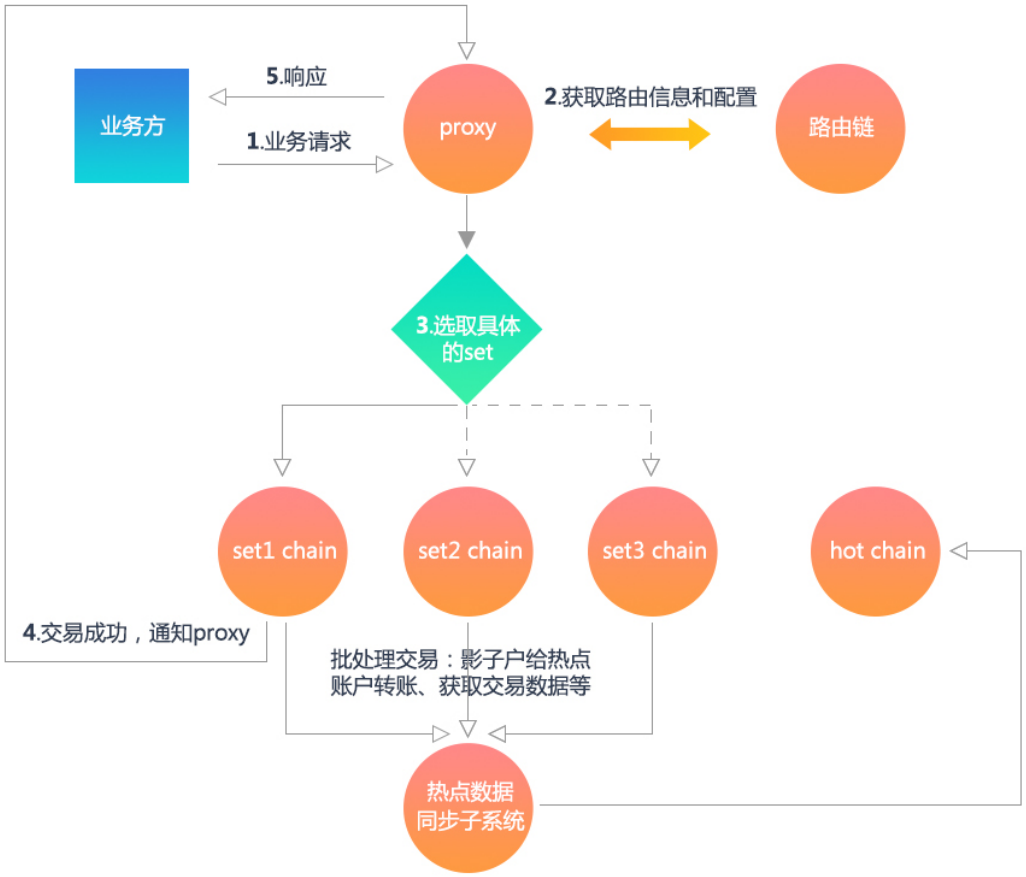
性，分布式事务完整性和一致性，以及分组之间可验证、不可篡改、可追溯的互信性和交易安全



性。

并行计算跨链架构

在此版本，我们根据金融业常见的“热点帐户”场景，提出了一种解决方案。在很多金融交易场景里，可能会出现大量的独立用户帐户和少数集中的一个或多个热点帐户产生交易的情况，如用户往某个热卖中的商户付款，或者用户频繁从某个帐户中提现或者获取优惠券、积分或者其他资产等，由于用户帐户数量较大, 相对来说，这些被集中访问的商户帐户，就被称为“热点帐户”。热点帐户在完成和用户的交易之外，还需要汇总所有的交易结果，计算总分帐，余额等，以便完成其特有的商业流程，如清结算



等。 热点账户架构

由于针对热点账户的交易量较大且所有用户都可能和它发生交易，我们考虑设计多个并行的交易链，首先将用户按照一定的性能模型分组，每个针对用户的分组而构建的独立的区块链组件，我们称为“用户交易链”。举例：预估为每个分组100万用户，5个分组能容纳500万用户（实际的每组能容纳的用户数需要根据业务场景实测评估），这样我们构建了5个“用户交易链”。然后，热点帐户本身可以集中在一个热点账户的链上（也可以分配在某一个分组里），热点账户链主要用于准实时的汇总各“用户交易链”的账务，以管理热点账户的总分帐，如总收入，总支出，帐户准实时余额等。为了支持用户和热点账户的交易，热点帐户在每个“用户交易链”上，都会设立一个影子户，用户在实时交易时，实际上是和“用户交易链”内的热点帐户影子户发生交易，每次交易都在用户交易链内部进行共识，不同的分组可以并行的进行交易计算，互不相关，用户和影子户之间的交易完成后，即意味着用户和热点帐户的交易完成。系统的容量和用户交易链的个数有关，用户交易链越多，系统容量越大，用户体验得到了保障。在用户交易链上，热点帐户影子户里只保存该分组里的总分帐，即一部分用户进行交易后，影子户里产生的收入、支出等，用户交易链会定期构建一次链内账目清算交易，并向热点帐户链发起一次跨链汇总交易，热点帐户链接收到交易之后，会到用户交易链去验证交易发送者的身份、汇总交易的存在性、账目的真实性和准确性，验证成功后，在热点帐户链上继续进行账目计算和入账操作，流程结束。整个过程会通过链间的中继，进行多次双向通信，且在不同的链上完整的执行共识确认。用户和影子户的交易可在一次共识的时间段内完成，时延较短，以满足用户体验。热点账户的总分帐计算为准实时完成，其时延取决于定时发起汇总交易的间隔，以及用户交易链和热点帐户链的共识时间。系统会保证用户交易链和热点帐户链之间的交易不错、不乱、不漏，具备事务一致性和完整性。并行多链计算是一个基础的系统扩展方案，热点帐户的解决方案是一个场景性的实现，充分理解并行多链计算和跨链交易的实现后，可以针对有海量需求的各种金融交易场景，设计出不同的方案来，以解决具体的场景问题。

7.3.3 UTXO账户模型

以太坊上基于UTXO模型的转账交易方案——使用手册

目录

1 基本介绍

1.1 方案背景

1.1.1 原有以太坊转账方案的不足之处

- 对于以太坊的账户模型，来源账户在给去向账户进行转账时，直接从本账户的余额中扣减转账数额，但无法确定所扣减的数额全部/部分来源于先前哪一笔交易，有哪些前置的消费条件，从而无法进行转账前的业务逻辑校验。
- 当同一来源账户在给多个去向账户进行转账时，存在多个交易共同操作来源账户余额的情况，基于以太坊数据一致性方案中交易列表及交易回执的有序性要求，区块链网络无法对转账交易进行并行处理。

返回目录

1.1.2 本方案解决上述问题的思路

- 本方案通过引入具有特定数额、有明确所有权标识的Token为转账的操作对象，而包含有Token的交易为UTXO交易。采用Token的拆分逻辑后，Token能够明确记录其来源及其交易限制条件，并在下一次转账交易中该Token被消费使用时也能根据交易限制条件进行包括业务逻辑校验在内的多种校验形式，从而提升了区块链网络中**转账交易的可扩展性**。
- 在本方案中，UTXO交易过程跟踪的是每个Token的所有权的转移，而不是账户的状态变化，因而各UTXO交易之间不共享任何状态、不会相互干扰，进而UTXO交易可以并发执行。在交易共识阶段，对满足并行执行条件的交易并发执行，而其他交易串行执行，进而可在一定程度上提升**以太坊网络中交易的转账效率**。

返回目录

1.2 交易原则

本文档描述一种以太坊上基于UTXO模型的转账交易方案。与UTXO模型类似，本方案中的转账交易有以下三个原则：

- 所有交易起于资产登记交易；
- 除了资产登记交易之外，所有的交易输入都必须来自于前面一个或者几个交易的输出；
- 每一笔的交易支出总额等于交易输入总额。

返回目录

1.3 数据类型

本方案的基础数据对象为Token、UTXOTx、Vault。Token为转账操作的基本单位，有特定的价值标记及所有权描述。UTXOTx描述一转账交易中来源Token与去向Token的对应关系，Vault记录一账号名下所有已花费及尚未花费的Token。三者关系如下图：

docs/features/UTXO/./imgs/UTXO.jpg

三者关系

上述三类数据基于原以太坊交易生成，独立于以太坊数据，采用LevelDB进行持久化存储。同时，本方案重点关注Token信息而非Token之间转换信息，为数据记录及使用方便，因此将Token数据记录与UTXOTx数据记录分离。

返回目录

2 使用说明

本文档分别提供UTXO交易在web3sdk(以下简称sdk)及nodejs中的使用说明。其中sdk的一般性使用说明请参考web3sdk使用说明文档，其中nodejs的一般性使用说明请参考FISCO BCOS区块链操作手册。如

无特定说明，sdk执行UTXO交易命令的位置位于web3sdk工程根目录执行gradle build生成的dist/bin目录下，nodejs执行UTXO交易命令的位置位于FISCO-BCOS/tool目录下。

2.0 前期准备：启用UTXO交易

在发送UTXO相关交易（资产登记交易及转账交易）前，需发送以下交易来启用UTXO交易。启用后，区块链支持原以太坊交易和UTXO交易的发送，因此后续发送原以太坊交易时不需另行关闭UTXO交易。

```
// 启用UTXO交易，Height为区块链当前块高+1，为十六进制格式（含0x前缀）
./web3sdk ConfigAction set updateHeight ${Height}                    // sdk命令
babel-node tool.js ConfigAction set updateHeight ${Height}       // nodejs命令，该命令在systemcontract目录下执行
```

返回目录

2.1 资产登记交易及转账交易

2.1.1 资产登记交易及转账交易相关的命令

```
// sdk命令，其中Type参数取值范围为{1,2,3}，分别代表基础交易例子、限制Token的使用账号为特定账号例子和限制某一账号的日转账限额例子
./web3sdk InitTokens $(Type)                          // 资产登记交易
./web3sdk SendSelectedTokens $(Type)                  // 转账交易

// nodejs命令
babel-node demoUTXO.js InitTokens                     // 资产登记交易
babel-node demoUTXO.js SendSelectedTokens             // 转账交易
```

在后续给出的资产登记交易及转账交易的三种使用例子中，使用nodejs的执行例子进行说明，sdk的执行例子类似，不再赘述。

返回目录

2.1.2 交易命令相关json字段说明

资产登记交易脚本中的json字段记录了本次资产登记操作所需生成的Token个数、单个Token所有权校验类型、单个Token所有者信息（即转账对象）、单个Token数额大小等内容。除上述必须字段外，Token的业务校验逻辑字段及备注字段为可选字段。资产登记交易的json字段如下：

```
txtype:1（交易类型为资产登记操作）
txout:交易输出列表，资产登记操作生成的Token数组（数组最大限制为1000）
        checktype:所有权校验类型，string，必须字段（可选P2PK和P2PKH）
        to:转账对象，string，必须字段（如果checktype为P2PK，本字段为账号地址；如果checktype为P2PKH，本字段为账号地址的哈希值）
        value:数额大小，string，必须字段（限制数额为正整数格式）
        initcontract:模板合约地址，string，可选字段（与initfuncandparams配对使用）
        initfuncandparams:调用模板合约所需传入的函数及参数，string，可选字段（ABI序列化之后结果，与initcontract配对使用）
        validationcontract:校验合约地址，string，可选字段
        oridetail:新创建Token的备注，string，可选字段
```

转账交易将脚本中json字段的交易输入Token列表进行消费。交易输入列表中除必须的Token key字段外，为实现业务逻辑校验功能所传入的字段需要与该Token生成时所设置的校验逻辑匹配。转账交易的json字段如下：

```
txtype:2（交易类型为转账操作）
txin:交易输入列表，本次转账消费的Token数组（数组最大限制为1000）
        tokenkey:转账使用的Token，string，必须字段（Token地址）
```

(continues on next page)

(continued from previous page)

```
        callfuncandparams:业务校验逻辑所需传入的函数及参数，string，可选字段（ABI序列化之后结果，当所消费的Token中存在校验合约地址时使用，通用合约及实例合约均需）
        exefuncandparams:执行业务校验逻辑（更新链上数据）所需传入的函数及参数，string，可选字段（ABI序列化之后结果，当所消费的Token中存在校验合约地址时使用，只限通用合约需要）
        desdetail:Token的转账备注，string，可选字段
txout:交易输出列表，转账操作生成的Token数组，内容同资产登记交易的txout
```

说明：

- 本方案通过在Token生成阶段挂载智能合约和Token使用阶段执行智能合约的方式来实现对一Token使用过程中的业务逻辑限制。用户可通过部署自定义的智能合约而不修改FISCO BCOS程序来实现不同的业务逻辑限制。
- 在挂载智能合约阶段，涉及三个字段，分别为initcontract、initfuncandparams和validationcontract。所涉及的智能合约有三种分类，分别为模板合约、实例合约和模板合约。
- 对于validationcontract字段（业务校验合约地址），用户可指直接传入已部署到链上的智能合约的地址（此时使用的智能合约我们称之为通用合约，该通用合约与特定Token无关）。如validationcontract字段的输入为空，且initcontract和initfuncandparams两字段有值时，将尝试通过initcontract和initfuncandparams生成validationcontract字段。其中initcontract的字段值也需为已部署到链上的智能合约的地址（此时使用的智能合约我们称之为模板合约，该模板合约被执行后生成的实例合约与特定Token相关，每个Token都有独立对应的实例合约）。
- 对于callfuncandparams和exefuncandparams两字段的使用，前者用于链上数据的只读判断，后者用于链上数据的可写更新，两字段中所调用的智能合约的函数接口是不一致的，但所传入参数是一致的。
- 本方案后续使用**锁定参数**、**解锁参数**、**验证过程**来描述一业务校验的实现逻辑。

返回目录

2.1.3 基础交易例子

资产登记交易：给账号0x3ca576d469d7aa0244071d27eb33c5629753593e登记资产，生成的Token价值为100单位，所有权校验类型为P2PK，json描述为：

```
var param = "{\"utxotype\":1,\"txout\":[{\"to\":\
↪"0x3ca576d469d7aa0244071d27eb33c5629753593e\",\"value\":\"100\",\"checktype\":\
↪"P2PK\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

相关执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js InitTokens
Param:
{ utxotype: 1,
  txout:
   [ { to: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '100',
       checktype: 'P2PK' } ] }
send transaction success:␣
↪0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1
Receipt:
{ blockHash: '0xf986fd0c11c62a08926f23bc05f19db385d9515b7540c1b963fb6298c6978d3c',
  blockNumber: 31,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1',
  transactionIndex: 0 }
```

转账交易：账号0x3ca576d469d7aa0244071d27eb33c5629753593e使用上述新铸的Token（记为Token1,在交易哈希为0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1的交易中生成）给账号0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3转账60价值单位，所有权校验类型为P2PK，将找零40价值单位，所有权校验类型为P2PKH，json描述为：

```
var Token1 = "0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1_0
↪";
var shaSendTo = "0x"+sha3("0x3ca576d469d7aa0244071d27eb33c5629753593e").toString();
var param = "{\"utxotype\":2,\"txin\":[{\"tokenkey\":\""+Token1+"\"}],\"txout\":[{\
↪"to\":\"0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3\",\"value\":\"60\",\
↪"checktype\":\"P2PK\"},{\"to\":\""+shaSendTo+"\",\"value\":\"40\",\"checktype\":\
↪"P2PKH\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

相关执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js SendSelectedTokens
Param:
{ utxotype: 2,
  txin: [ { tokenkey:
↪'0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1_0' } ],
  txout:
   [ { to: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3',
       value: '60',
       checktype: 'P2PK' },
     { to: '0x14a940d346b813e2204c74c13bbd556dc7c79a4e78e82617004d5ff77aa3b582',
       value: '40',
       checktype: 'P2PKH' } ] }
send transaction success:␣
↪0x06f0e326294e39183be0b04b4f5b60c927b0e43caaf9b709af1b62cd0f74951a
Receipt:
{ blockHash: '0x6c35f421fdd8c43cf8163db9620b67f9c08cf0cda79ce1da63e0612637c36b16',
  blockNumber: 32,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0x06f0e326294e39183be0b04b4f5b60c927b0e43caaf9b709af1b62cd0f74951a',
  transactionIndex: 0 }
```

返回目录

2.1.4 可扩展转账限制逻辑例子一（通过模板合约生成实例合约）：限制Token的使用账号为特定账号

资产登记及转账交易前需先部署tool目录下的UserCheckTemplate.sol合约。资产登记交易时，需添加限制条件，传入所部属的合约地址、调用接口及特定账号白名单，传入的接口及账号白名单使用ABI编码。转账交易时，需传入验证接口及相应交易来源账号的ABI编码结果。

对于有不同特定账号的限制条件的Token，上述合约只需部署一次，记合约部署后的地址为0x7dc38c5e144cbbb4cd6e8a65091da52a78d584f5。ABI编码详细信息可参考https://github.com/ethereum/wiki/wiki/Ethereum-Contract-ABI。

资产登记交易：给账号0x3ca576d469d7aa0244071d27eb33c5629753593e登记资产，生成的Token价值为100单位，所有权校验类型为P2PK，生成的Token只允许config.account使用（不失一般性可描述config.account为0x3ca576d469d7aa0244071d27eb33c5629753593e，即此资产登记交易的发起方），json描述为：

```
var initContractAddr = "0x7dc38c5e144cbbb4cd6e8a65091da52a78d584f5";
↪      // 模板合约地址，用于创建实例合约，创建Token传入
// tx_data为调用模板合约的函数说明及参数，创建Token传入
var initFunc = "newUserCheckContract(address[])";
↪      // 模板合约中的函数说明（含参数，没有空格）
```

(continues on next page)

(continued from previous page)

```
var initParams = [[config.account]];                                            ␣
↪                                  // 调用函数传入的参数列表
var init_tx_data = getTxData(initFunc, initParams);                                      ␣
↪       // ABI序列化
var param = "{\"utxotype\":1,\"txout\":[{\"to\":\
↪"0x3ca576d469d7aa0244071d27eb33c5629753593e\",\"value\":\"100\",\"checktype\":\
↪"P2PK\",\"initcontract\":\""+initContractAddr+"\",\"initfuncandparams\":\""+init_
↪tx_data+"\",\"oridetail\":\"Only userd by config.account\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

其中detail字段可以用来描述只归某些特定账号所使用的信息。**需确认资产登记交易生成的Token中validationContract字段非0，才表示该Token附带了验证逻辑。**相关执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js InitTokens
Param:
{ utxotype: 1,
  txout:
   [ { to: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '100',
       checktype: 'P2PK',
       initcontract: '0x7dc38c5e144cbbb4cd6e8a65091da52a78d584f5',
       initfuncandparams:
↪'0xe8ef928900000000000000000000000000000000000000000000000000000000000000200000000000000000000000
↪',
       oridetail: 'Only userd by config.account' } ] }
send transaction success:␣
↪0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b
Receipt:
{ blockHash: '0xdb10177f2cce8f2528c9035de16ac170e2e04031db08b9c9a7d328139c567d90',
  blockNumber: 35,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b',
  transactionIndex: 0 }
```

转账交易：账号0x3ca576d469d7aa0244071d27eb33c5629753593e使用上述新铸的Token（记为Token1，在交易哈希为0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b的交易中生成）给账号0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3转账60价值单位，所有权校验类型为P2PK，将找零40价值单位，所有权校验类型为P2PKH，所生成的Token不再有使用账号的限制，json描述为：

```
var Token1 = "0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b_0
↪";
var checkFunc = "check(address)";
var checkParams = [config.account];
var check_tx_data = getTxData(checkFunc, checkParams);
var shaSendTo = "0x"+sha3("0x3ca576d469d7aa0244071d27eb33c5629753593e").toString();
var param = "{\"utxotype\":2,\"txin\":[{\"tokenkey\":\""+Token1+"\",\
↪"callfuncandparams\":\""+check_tx_data+"\"}],\"txout\":[{\"to\":\
↪"0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3\",\"value\":\"60\",\"checktype\":\
↪"P2PK\"},{\"to\":\""+shaSendTo+"\",\"value\":\"40\",\"checktype\":\"P2PKH\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

相关执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js SendSelectedTokens
Param:
{ utxotype: 2,
  txin:
```

(continues on next page)

(continued from previous page)

```
   [ { tokenkey:
↪'0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b_0',
       callfuncandparams:
↪'0xc23697a8000000000000000000000003ca576d469d7aa0244071d27eb33c5629753593e' } ],
  txout:
   [ { to: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3',
       value: '60',
       checktype: 'P2PK' },
     { to: '0x14a940d346b813e2204c74c13bbd556dc7c79a4e78e82617004d5ff77aa3b582',
       value: '40',
       checktype: 'P2PKH' } ] }
send transaction success:␣
↪0xe351f004f9bf163f5f8905d1addd79e6aa46d0295fabae2433bf055f38e299d9
Receipt:
{ blockHash: '0xb3898ceff824ce3776f3bbf3bb1ccb89be156ded5c4f452811fa831f4582ce66',
  blockNumber: 36,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0xe351f004f9bf163f5f8905d1addd79e6aa46d0295fabae2433bf055f38e299d9',
  transactionIndex: 0 }
```

上述例子描述了一种**通过智能合约实现所有权简易验证**方式。在本例子中，虽然要求传入checkParams参数（解锁参数），但在验证过程中，验证合约通过tx.origin获取转账交易发送方地址，判断其是否为该Token生成时设定的特定账号之一（解锁参数）。

更一般化的所有权验证方式为**通过ecrecover()校验数字签名的合法性来进行所有权验证**。在Token生成时传入的锁定参数为一随机数的哈希值，转账该Token过程中解锁参数为该Token所有者在合约外部对锁定参数签名后生成的rsv三个数值，验证过程为使用Solidity的ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)函数，判断函数的返回结果是否与tx.origin一致，其中的hash参数为随机数的哈希值，rsv为解锁参数。

同时，**MS（多重签名）**是上述基于ecrecover()进行所有权验证功能的进一步扩展。

另一方面，对于**对Token进行一段时间的冻结**的业务限制逻辑，锁定参数为使用Solidity自身携带的函数now (uint)来获取一Token生成时当前区块的timestamp A和冻结Token的时间限制，解锁参数缺省，验证过程为判断转账该Token过程中当前区块的timestamp B与锁定参数timestamp A的差值是否满足冻结条件。

通过对上述不同业务限制场景实现方法的描述，可发现**Token所有者可通过对锁定参数、解锁参数和验证过程的设定，来实现对使用某一特定Token的业务逻辑限制**。

返回目录

2.1.5 可扩展转账限制逻辑例子二（使用通用合约）：限制某一账号的日转账限额

资产登记及转账交易前需先部署tool目录下的Limitation.sol合约。此合约可用于记录不同用户的日转账限额及当日已转账数额，并提供设置日转账限额、重置当日已转账数额的接口。资产登记交易时，需传入部署的Limitation.sol合约地址作为验证逻辑入口。转账交易时，需传入相关的调用接口及数据（使用ABI编码，含验证数据及更新数据）。

记合约部署后的地址为0x3dbac83f7050e377a9205fed1301ae4239fa48e1。在UTXO交易前设置账号的日转账限额，设置日转账限额的相关脚本为demoLimitation.js。不失一般性记账号为0x3ca576d469d7aa0244071d27eb33c5629753593e。

资产登记交易：给账号0x3ca576d469d7aa0244071d27eb33c5629753593e登记资产，生成的Token价值为100单位，所有权校验类型为P2PK，限制config.account日转账限额为500（不失一般性可描述config.account为0x3ca576d469d7aa0244071d27eb33c5629753593e，即此资产登记交易的发起方），json描述为：

```
var validationContractAddr = "0x3dbac83f7050e377a9205fed1301ae4239fa48e1";
↪      // 通用合约地址，创建Token传入
var param = "{\"utxotype\":1,\"txout\":[{\"to\":\
↪"0x3ca576d469d7aa0244071d27eb33c5629753593e\",\"value\":\"100\",\"checktype\":\
↪"P2PK\",\"validationcontract\":\""+validationContractAddr+"\",\"oridetail\":\
↪"Account with Limitation per day\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

相关执行例子如下，**需确认资产登记交易生成的Token中validationContract字段非0，才表示该Token附带了验证逻辑。**

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js InitTokens
Param:
{ utxotype: 1,
  txout:
   [ { to: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '100',
       checktype: 'P2PK',
       validationcontract: '0x3dbac83f7050e377a9205fed1301ae4239fa48e1',
       oridetail: 'Account with Limitation per day' } ] }
send transaction success:
↪0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6
Receipt:
{ blockHash: '0xccf85aa676b564a5c61c06d4381b379769492a0c4c20a0c6531b63245a09a207',
  blockNumber: 40,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6',
  transactionIndex: 0 }
```

转账交易：账号0x3ca576d469d7aa0244071d27eb33c5629753593e使用上述新铸的Token（记为Token1，在交易哈希为0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6的交易中生成）给账号0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3转账60价值单位，所有权校验类型为P2PK，将找零40价值单位，所有权校验类型为P2PKH，所生成的Token不再有日转账限额的限制，json描述为：

```
var Token1 = "0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0
↪";
var checkFunc = "checkSpent(address,uint256)";
var checkParams = [config.account,60];
var check_tx_data = getTxData(checkFunc, checkParams);
var updateFunc = "addSpent(address,uint256)";
var updateParams = [config.account,60];
var update_tx_data = getTxData(updateFunc, updateParams);
var shaSendTo = "0x"+sha3("0x3ca576d469d7aa0244071d27eb33c5629753593e").toString();
var param = "{\"utxotype\":2,\"txin\":[{\"tokenkey\":\""+Token1+"\",\
↪"callfuncandparams\":\""+check_tx_data+"\",\"exefuncandparams\":\""+update_tx_
↪data+"\"}],\"txout\":[{\"to\":\"0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3\",\
↪"value\":\"60\",\"checktype\":\"P2PK\"},{\"to\":\""+shaSendTo+"\",\"value\":\"40\
↪",\"checktype\":\"P2PKH\"}]}";
await web3sync.sendUTXOTransaction(config.account, config.privKey, [param]);
```

相关执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js SendSelectedTokens
Param:
{ utxotype: 2,
  txin:
   [ { tokenkey:
↪'0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0',
```

(continues on next page)

(continued from previous page)

```
       callfuncandparams:
↪'0x3be8c7e400000000000000000000000003ca576d469d7aa0244071d27eb33c5629753593e0000000000000000000000
↪',
       exefuncandparams:
↪'0x76d9a04a000000000000000000000000003ca576d469d7aa0244071d27eb33c5629753593e0000000000000000000000
↪' } ],
  txout:
   [ { to: '0x64fa644d2a694681bd6addd6c5e36cccd8dcdde3',
       value: '60',
       checktype: 'P2PK' },
     { to: '0x14a940d346b813e2204c74c13bbd556dc7c79a4e78e82617004d5ff77aa3b582',
       value: '40',
       checktype: 'P2PKH' } ] }
send transaction success:␣
↪0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584
Receipt:
{ blockHash: '0xdd5fd6bc49759b7b750158dacf766998c4493ec32405c99c5d38b5b5e93c3da9',
  blockNumber: 43,
  contractAddress: '0x0000000000000000000000000000000000000000',
  cumulativeGasUsed: 30000,
  gasUsed: 30000,
  logs: [],
  transactionHash:
↪'0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584',
  transactionIndex: 0 }
```

返回目录

2.1.6 并行交易

本方案对满足以下条件的资产登记/转账交易将进行并行处理。并行处理的线程数与机器配置相关，但用户对交易并行无感知。

- 为UTXO类型的交易；
- 同一块中已打包的交易不涉及对同一个Token的使用；
- 交易没有设置对链上数据进行写操作的逻辑。

返回目录

2.2 查询功能

2.2.0 前期准备：账号注册

账号只有注册后才能进行GetVault、SelectTokens、GetBalance操作。账号在区块链中注册一次即可，后续链启动后会获取之前已经注册过的账号。

```
// 账号注册
./web3sdk RegisterAccount $(Account)                    // sdk命令
babel-node demoUTXO.js RegisterAccount ${Account}       // nodejs命令
```

返回目录

2.2.1 回溯

```
./web3sdk TokenTracking $(TokenKey)                     // sdk命令
babel-node demoUTXO.js TokenTracking ${TokenKey}        // nodejs命令
```

相关的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js TokenTracking␣
↪0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1
Param[0]:
{ utxotype: 8,
  queryparams:
   [ { tokenkey:
↪'0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1',
       cnt: '3' } ] }
Result[0]:
{ begin: 0,
  cnt: 3,
  code: 0,
  data:
   [ '{"in":["0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0
↪"],"out":["0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_0",
↪"0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1"]}',
     '{"out":["0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0
↪"]}' ],
  end: 1,
  msg: 'Success',
  total: 2 }
```

上述例子使用了分页查询，查询输入的Param[x]的返回结果为Result[x]，两者一一配对。Param中json字段说明如下：

```
txtype:8（交易类型为回溯Token来源交易信息）
queryparams:回溯参数列表，数组（数组大小为1）
        tokenkey:需回溯的Token Key，string，必须字段（Token地址）
        begin:分页查询的起始位置，string，可选字段（如不传入默认为0）
        cnt:分页查询的查询个数，string，可选字段（如不传入默认为10）
```

Result中json字段说明如下：

```
begin:该页查询的数据在查询结果核心数据中的起始位置（Param传入）
cnt:该页查询的查询个数（Param传入）
code:执行结果代码
data:查询结果核心数据（这里为该Token从该资产登记开始到目前转账交易的倒序列表）
end:该页查询的数据在查询结果核心数据中的结束位置
msg:执行结果说明（与执行结果代码配对，结果说明详见2.3）
total:查询结果核心数据列表的长度
```

返回目录

2.2.2 查询账号余额

```
./web3sdk GetBalance $(Account)                          // sdk命令
babel-node demoUTXO.js GetBalance ${Account}             // nodejs命令
```

相关的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js GetBalance␣
↪0x3ca576d469d7aa0244071d27eb33c5629753593e
Param:
{ utxotype: 9,
  queryparams: [ { account: '0x3ca576d469d7aa0244071d27eb33c5629753593e' } ] }
Result:
{ balance: 320, code: 0, msg: 'Success' }
```

返回目录

2.2.3 获取一账号下能满足支付数额的Token列表

```
./web3sdk SelectTokens $(Account) $(Value)                  // sdk命令
babel-node demoUTXO.js SelectTokens ${Account} ${Value} // nodejs命令
```

相关的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js SelectTokens␣
↪0x3ca576d469d7aa0244071d27eb33c5629753593e 245
Param[0]:
{ utxotype: 7,
  queryparams:
   [ { account: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '245' } ] }
Result[0]:
{ begin: 0,
  cnt: 10,
  code: 0,
  data:
   [ '0xbf7d37c245ce96fa72b669a2e2fcc006e4adacfb6dc4c3746bd1311feba0bc0e_0',
     '0x69cb330a4fe9addae3bceb3550f82f231d7a3c2ce6d7cb2e1a7bff54476562d1_0',
     '0xe351f004f9bf163f5f8905d1addd79e6aa46d0295fabae2433bf055f38e299d9_1',
     '0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1' ],
  end: 3,
  msg: 'Success',
  total: 4,
  totalTokenValue: 280 }
```

返回目录

2.2.4 查询基础数据对象（Token、UTXOTx、Vault)

查询Token信息

```
./web3sdk GetToken $(TokenKey)                              // sdk命令
babel-node demoUTXO.js GetToken ${TokenKey}                 // nodejs命令
```

Token查询的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js GetToken␣
↪0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1
Param:
{ utxotype: 4,
  queryparams: [ { tokenkey:
↪'0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1' } ] }
Result:
{ code: 0,
  data:
   { TxHash: '0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584',
     checkType: 'P2PKH',
     contractType: '',
     detail: '',
     index: 1,
     owner: '0x14a940d346b813e2204c74c13bbd556dc7c79a4e78e82617004d5ff77aa3b582',
     state: 1,
     validationContract: '0x0000000000000000000000000000000000000000',
     value: 40 },
  msg: 'Success' }
```

查询UTXOTx信息

```
./web3sdk GetTx $(TxKey)                                    // sdk命令
babel-node demoUTXO.js GetTx ${TxKey}                       // nodejs命令
```

UTXOTx查询的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js GetTx␣
↪0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584
Param:
{ utxotype: 5,
  queryparams: [ { txkey:
↪'0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584' } ] }
Result:
{ code: 0,
  data:
   { in: [ '0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0'␣
↪],
     out:
      [ '0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_0',
        '0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1' ] },
  msg: 'Success' }
```

查询Vault信息

```
// 查询一账号下的Vault，传入需查询的账号及查询Token类型（0为全查询，1为查询尚未花费的Token，2为
查询已经花费的Token）
./web3sdk GetVault $(Account) $(TokenType)              // sdk命令
babel-node demoUTXO.js GetVault ${Account} ${TokenType} // nodejs命令
```

Vault查询的nodejs执行例子如下：

```
[fisco-bcos@VM_centos tool]$ babel-node demoUTXO.js GetVault␣
↪0x3ca576d469d7aa0244071d27eb33c5629753593e 0
Param[0]:
{ utxotype: 6,
  queryparams:
   [ { account: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '0',
       cnt: '6' } ] }
Result[0]:
{ begin: 0,
  cnt: 6,
  code: 0,
  data:
   [ '0x06f0e326294e39183be0b04b4f5b60c927b0e43caaf9b709af1b62cd0f74951a_1',
     '0x278d3b3ffa7380baba00e8029aa1e8fd2455ceb82562b82cbce41c344e4b1584_1',
     '0x4c1a08acbfa16e496489de87e269aef927ea582e674d6c7e363779f8576873a0_0',
     '0x69cb330a4fe9addae3bceb3550f82f231d7a3c2ce6d7cb2e1a7bff54476562d1_0',
     '0xa1299435e1cbc019aed361f5c5975941118908747152 8c5638ad1e66f654e3b1_0',
     '0xbf7d37c245ce96fa72b669a2e2fcc006e4adacfb6dc4c3746bd1311feba0bc0e_0' ],
  end: 5,
  msg: 'Success',
  total: 9 }
Param[1]:
{ utxotype: 6,
  queryparams:
   [ { account: '0x3ca576d469d7aa0244071d27eb33c5629753593e',
       value: '0',
       begin: '6',
       cnt: '6' } ] }
Result[1]:
{ begin: 6,
  cnt: 6,
  code: 0,
  data:
   [ '0xd77d4b655c6f3a7870ef66676b1375249f1e5ff34045374a1fc244f2fdf09be6_0',
     '0xe351f004f9bf163f5f8905d1addd79e6aa46d0295fabae2433bf055f38e299d9_1',
     '0xf60b1f4e5ff6ebd2b55a8214d8a979b56912b48d1d48033dff64bd445899e24b_0' ],
```

(continues on next page)

(continued from previous page)

```
  end: 8,
  msg: 'Success',
  total: 9 }
```

2.2.5 分页查询

本方案对GetVault、SelectTokens和TokenTracking接口，提供分页查询功能。相关执行例子及字段说明见上。sdk及nodejs对上述三个接口均采用了分页查询的方案，其中sdk返回的查询结果中已汇总不同页的查询内容，因此查询结果中不再存在begin、cnt和end等json字段。

返回目录

2.3 脚本命令返回的“执行结果说明”

```
"Success",                          // "Success./执行成功",
"TokenIDInvalid",                   // "Token ID is invalid./Token ID不存在",
"TxIDInvalid",                      // "Tx ID is invalid./Tx ID不存在",
"AccountInvalid",                   // "Account is invalid./账号不存在",
"TokenUsed",                        // "Token has been used./该Token已经使用",
"TokenOwnerShipCheckFail",          // "The ownership validation of token does not␣
↪pass through./该Token所有权验证失败",
"TokenLogicCheckFail",              // "The logical validation of token does not pass␣
↪through./该Token逻辑验证失败",
"TokenAccountingBalanceFail",       // "The accounting equation verification of␣
↪transaction does not pass through./该交易会计等式验证失败",
"AccountBalanceInsufficient",       // "The balance of the account is insufficient./该账
号余额不足",
"JsonParamError",                   // "Json parameter formatting error./输入Json参数错误
↪",
"UTXOTypeInvalid",                  // "UTXO transaction type error./UTXO交易类型错误",
"AccountRegistered",                // "Account has been registered./账号已经存在",
"TokenCntOutofRange",               // "The number of Token numbers used in the␣
↪transaction is beyond the limit(max=1000)./交易用的Token参数超限（TokenMaxCnt=1000）
↪",
"LowEthVersion",                    // "Please upgrade the environment for UTXO␣
↪transaction./Eth的版本过低，无法处理UTXO交易",
"OtherFail"                         // "Other Fail./其余失败情况"
```

返回目录

3 注意事项

3.1 业务校验逻辑及验证逻辑的传入说明

用户进行转账交易时，为保证交易转账的相关信息（如发送账号、转账数额）和验证逻辑所传入的数据一致，建议系统管理者封装一层接口用于发送交易（含交易中需传入的验证信息），供外部用户调用。

返回目录

3.2 兼容性说明

- 本UTXO交易方案可实现对链原有数据的兼容；
- 本UTXO交易的启用及后续交易的发送可在关闭/启用国密功能的情况下进行。

返回目录

3.3 工具提供

- 目前提供支持UTXO交易的sdk及nodejs工具。

返回目录

7.4 易用性

7.4.1 浅谈FISCO BCOS的易用性

作者：fisco-dev

FISCO BCOS是聚焦于金融领域的区块链底层平台，期望通过开源来推动生态圈的良好发展。随着开源技术的普及和参与者数量的增加，FISCO BCOS的影响范围将愈发深广，藉此最终构建出开放共赢的区块链生态圈。

开源技术的普及和参与者数量的增加，要求区块链底层平台具备易于使用、可规模化运行的特征。因此FISCO BCOS自设计伊始就十分关注其易用性，在编译部署、开发工具、业务场景案例、以及后续的运维和治理等各个环节，用心设计，并且不断更迭优化、删繁就简，以更好地提升使用者的效率，全面降低应用和治理的成本，同时也让更多对区块链有浓厚兴趣的爱好者也能感受到技术的魅力，参与到区块链的应用体验中。

FISCO BCOS的易用性涵盖了安装部署、代码开发、业务构建、运营运维等方面，在以下5点的表现尤为突出。

一键安装脚本

FISCO BCOS支持一键快速安装部署，几条命令就可以运行FISCO BCOS。下载代码后，只需一步，即可完成安装。简单快捷，且成功率高，尤其适合初学者快速体验FISCO BCOS。

一键安装支持两种场景：

- 1）单机器2节点：即场景涉及两个区块链节点的部署，两个节点都在一台机器上；两个节点相互连接，形成一条由两个节点组成的区块链。部署只需运行shell，执行成功后，即可通过查看进程及日志，判断节点是否运行正常
- 2）两机器4节点：即场景涉及4个区块链节点的部署，其中两个在FISCO BCOS所安装机器本地，两个在另一台机器上；4个节点相互连接，形成一条由4个节点组成的区块链。部署需先生成节点，再分别启动两台机器的节点，随后即可在任一机器上执行shell验证是否正常运行。

一键安装脚本可用于开发和体验环境的构建，如需添加更多节点以组成规模更大的网络，详细安装方法请参考FISCO BCOS使用说明书相关章节。

附：

一键安装FISCO BCOS脚本使用说明

FISCO BCOS使用说明书

SDK工具

FISCO BCOS平台提供的SDK工具，可同时支持java和node.js两种开发语言。在SDK基础上,开发者可开发面向最终用户的客户端程序，调用链上节点的功能接口，在客户端上即可以访问链上部分或全部的数据，向区块链发起交易。

在SDK的设计上，可直接面向业务，提供业务级别的接口，开发者只需关注业务数据的字段以及调用返回结果，不需要了解区块链节点的具体部署情况，不需要处理异步通信的细节，即可实现业务合约的管理、执行、交易查询功能，在业务sample中会采用这种思路提供SDK。FISCO BCOS同时提供对应的说明文档和使用范例，大幅度降低开发门槛和成本，帮助开发者快速开发各种业务场景的应用。

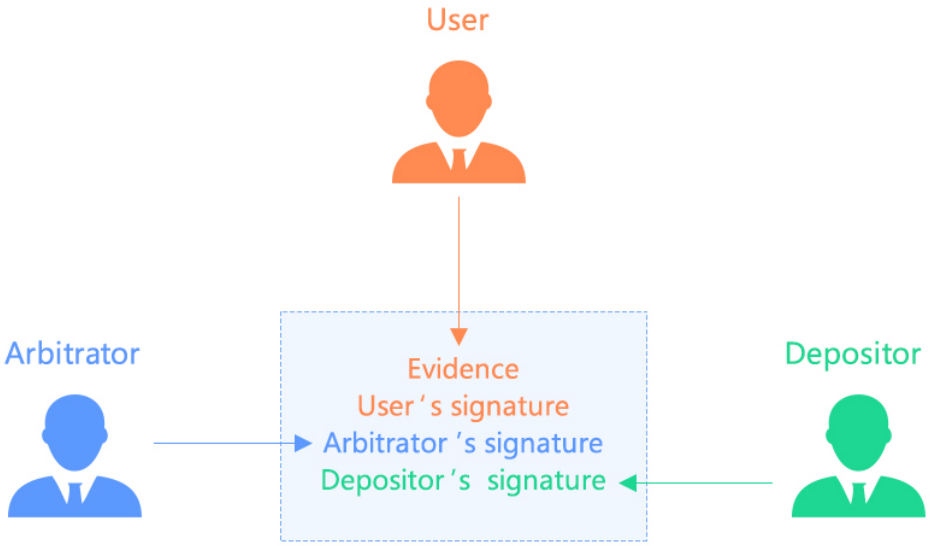
除了面向业务，SDK还可以直接调用区块链底层功能，开发者需要熟悉区块链节点所提供的底层功能接口，基本数据结构，以及节点的部署情况，SDK则为开发者屏蔽协议编解码以及异步通信，容错等技术细节，减少繁琐的重复工作，提供了极大程度的易用性。

附： web3sdk使用指南

Sample

FISCO BCOS提供了sample供开发者学习和使用，以便帮助开发者在对应业务场景下快速启动项目。以存证sample为例，提供了完整的业务sdk代码和详细的说明文档。 存证sample演示了如下4个业务流程：

- a.新建、部署工厂合约。
- b.用工厂合约新建证据合约（需要传工厂合约的地址文件）。
- c.对证据进行签名。
- d.验证证据已有的签名正确性和完整性。



存证sample流程图

存证sample为开发者提供了大量的默认配置，大大降低了用户自主配置的成本。使用一键脚本，只需配置节点ip和端口，就可以直接运行整个存证流程。sample同时配备了详细的文档说明，给用户提供step by step的使用指导，协助用户直观快速地理解系统。在此基础上，带来多种体验方式，既可以整体一键式快速体验整个存证流程，也可单步详细分析每个步骤。

目前FISCO BCOS sample所覆盖的业务场景亦在不断丰富和完善中。

附： 存证Sample说明

区块链浏览器

FISCO BCOS通过区块链浏览器构建了丰富、实时、可视化的数据监控体系，轻松跨越数据监控的技术门槛。在浏览器界面，可以直观清晰地看到FISCO BCOS的内部结构，自动查询区块的历史与实时信息，从而省去开发者自己构造报文从区块链节点的RPC端口查询的繁琐操作，能一目了然的掌握以下信息：

- 节点的ID、端口、块高等信息；
- 区块的块高、时间、交易数量、出块者以及Gas相关信息；

- 交易的Hash、所属块、块内ID、交易时间、发送者、接收者等信息；
- 网络中正在处理的尚未确认的交易信息。

FISCO BCOS浏览器监控指标亦在持续地完善中，后续还会计划通过浏览器进行链的构建和运营，一旦完成，在前端浏览器上即可实现部署节点启停、合约部署、交易发布、增删节点等操作。这将进一步大大提高易用性。

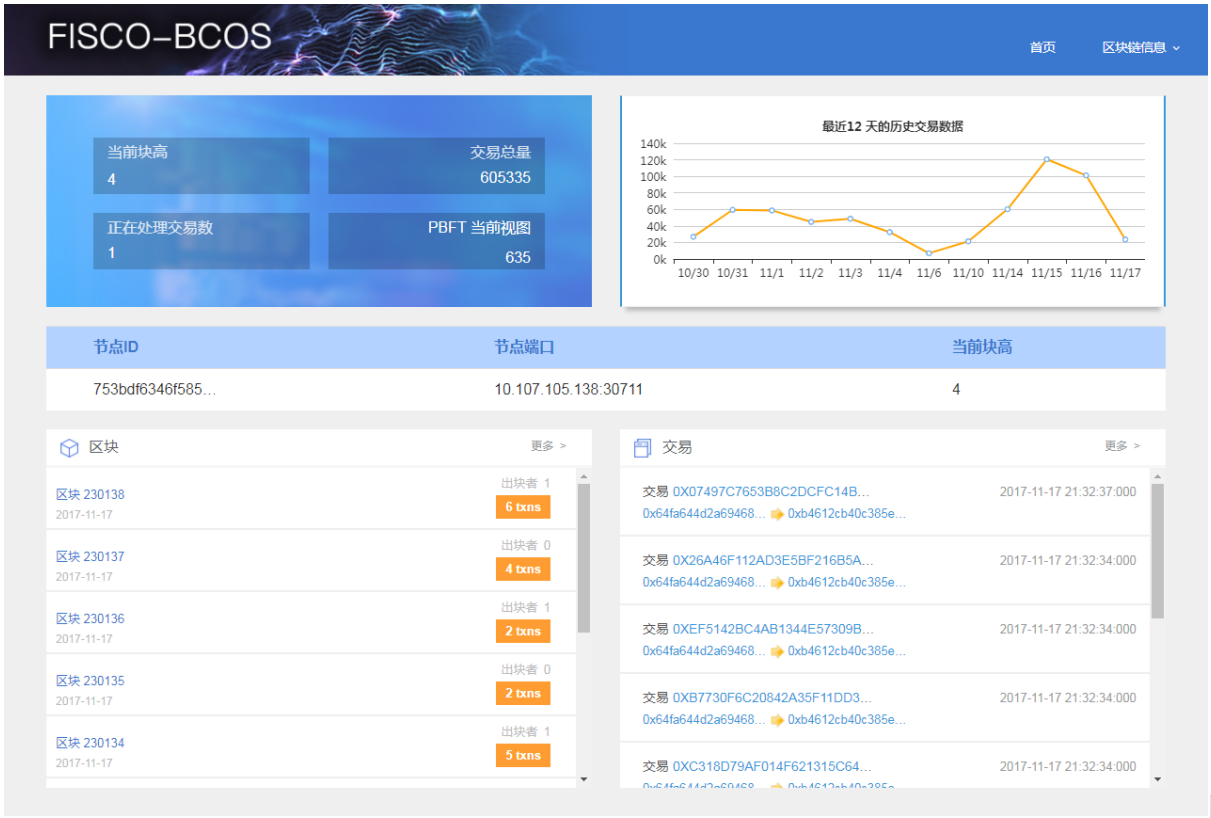


图2：区块链浏览器总览

块高	生成时间	交易数量	出块者	Gas消耗量	Gas上限
230138	2017-11-17 21:32:34:000	6	1	748362 (0.04%)	2000000000
230137	2017-11-17 21:32:31:000	4	0	498908 (0.02%)	2000000000
230136	2017-11-17 21:32:28:000	2	1	249454 (0.01%)	2000000000
230135	2017-11-17 21:32:25:000	2	0	249454 (0.01%)	2000000000
230134	2017-11-17 21:32:24:000	5	1	623635 (0.03%)	2000000000
230133	2017-11-17 21:32:21:000	5	0	623635 (0.03%)	2000000000
230132	2017-11-17 21:32:18:000	3	1	374181 (0.02%)	2000000000
230131	2017-11-17 21:32:15:000	6	0	748362 (0.04%)	2000000000
230130	2017-11-17 21:32:12:000	2	1	249454 (0.01%)	2000000000

图3：区块信息

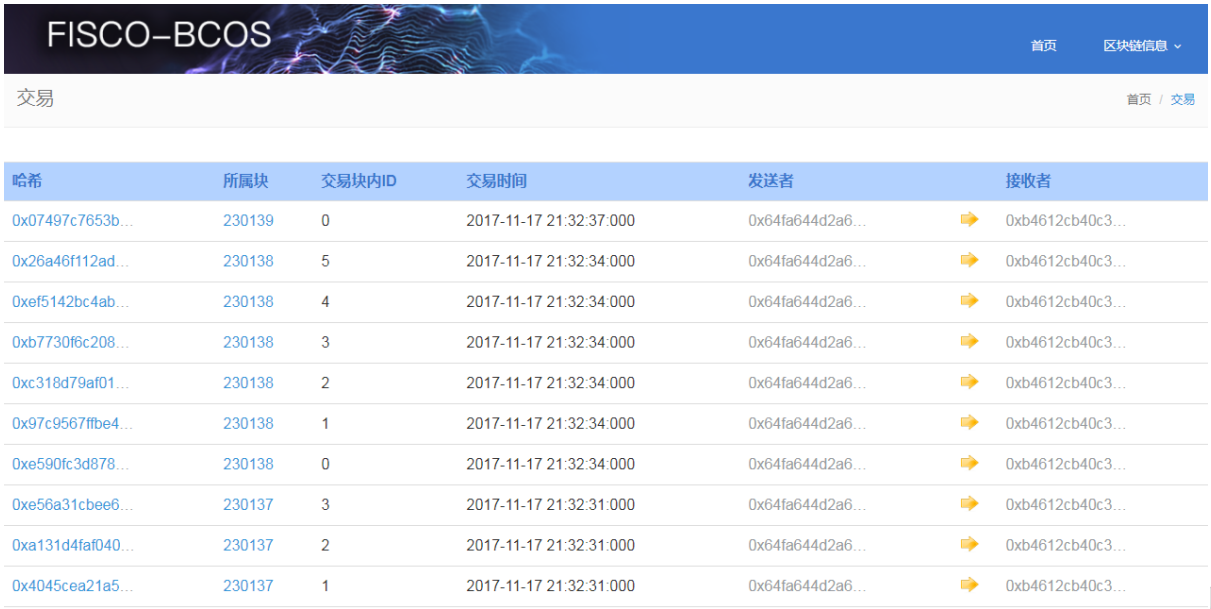


图4：交易信息

同时FISCO BCOS还预埋了监控指标，让开发者只需跟踪查看监控指标输出日志，就能快速获取区块链运行过程中的深层信息，目前日志已覆盖数据库、共识算法、交易和区块四个维度。

浏览器和预埋指标展示数据直观便捷，在满足多个层级的运营分析需求的同时，能大幅度降低应用难度，节省查询统计成本，提升监控效率。

附： 区块链浏览器说明

运维手册

FISCO BCOS为开发者配备了完善的运维手册，囊括常见错误、升级、管理、治理等模块。区块链系统的运行逻辑具有分布式一致性，不同节点的软硬件配置也基本一致，先天的具备标准化特性，开发者可参照运维手册，自主诊断，快速获取解决方案，使用相应的工具、运维策略和运维流程等对区块链系统进行构建、部署、配置以及故障处理，从而提升反应速度、降低运维成本，最终提升运营效率。

附： 运维手册： （待补充）

以上各层面的易用性，便于各领域的合作伙伴以较低成本快速搭建上层区块链应用，并且持续地高效治理和运营，在推进区块链技术的普及的同时也协助合作伙伴将精力聚焦在业务本身和商业模式的运营上，构建科技和金融深度合作的长效机制，最终达到多方受益，共同打造金融创新的区块链共赢生态。

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

7.4.2 链上信使协议AMOP使用指南

作者：fisco-dev

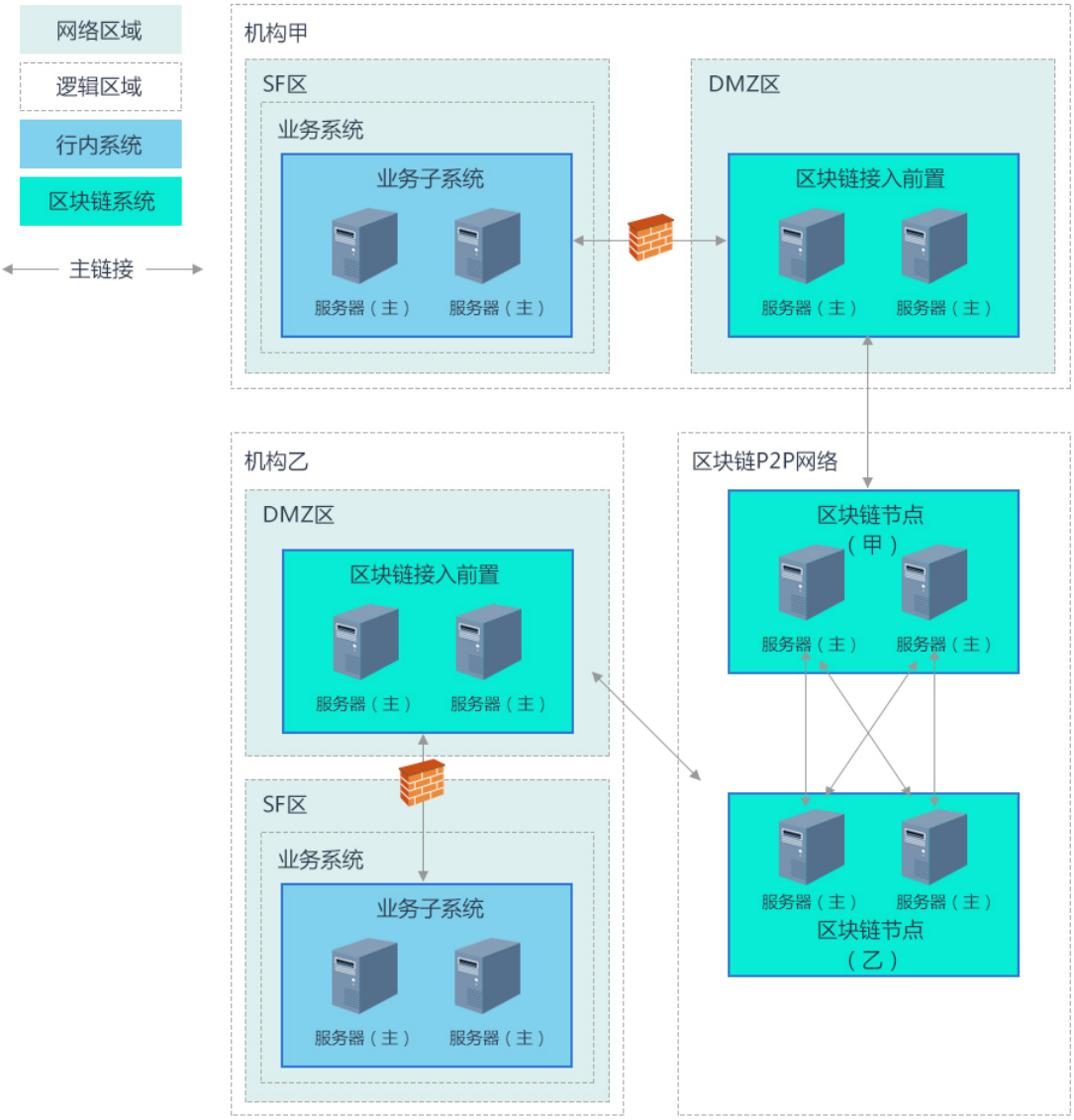
介绍

链上信使协议AMOP（Advance Messages Onchain Protocol）系统旨在为联盟链提供一个安全高效的消息信道，联盟链中的各个机构，只要部署了区块链节点，无论是共识节点还是观察节点，均可使用AMOP进行通讯，AMOP有如下优势：

- 实时：AMOP消息不依赖区块链交易和共识，消息在节点间实时传输，延时在毫秒级。
- 可靠：AMOP消息传输时，自动寻找区块链网络中所有可行的链路进行通讯，只要收发双方至少有一个链路可用，消息就保证可达。
- 高效：AMOP消息结构简洁、处理逻辑高效，仅需少量cpu占用，能充分利用网络带宽。

- 安全：AMOP的所有通讯链路使用SSL加密，加密算法可配置。
- 易用：使用AMOP时，无需在SDK做任何额外配置。

逻辑架构



以银行典型IDC架构为例，各区域概述：

- SF区：机构内部的业务服务区，此区域内的业务子系统使用区块链SDK，如无DMZ区，配置SDK连接到区块链节点，反之配置SDK连接到DMZ区的区块链前置。
- DMZ区：机构内部的外网隔离区，非必须，如有，该区域部署区块链前置。
- 区块链P2P网络：此区域部署各机构的区块链节点，此区域为逻辑区域，区块链节点也可部署在机构内部。

配置

AMOP无需任何额外配置，以下为SDK的配置案例 SDK配置（Spring Bean）：

```
        <?xml version="1.0" encoding="UTF-8" ?>
        <beans xmlns="http://www.springframework.org/schema/beans"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p=
↪"http://www.springframework.org/schema/p"
               xmlns:tx="http://www.springframework.org/schema/tx" xmlns:aop=
↪"http://www.springframework.org/schema/aop"
               xmlns:context="http://www.springframework.org/schema/context"
               xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                 http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
                 http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

        <!-- AMOP消息处理线程池配置，根据实际需要配置 -->
        <bean id="pool" class="org.springframework.scheduling.concurrent.
↪ThreadPoolTaskExecutor">
               <property name="corePoolSize" value="50" />
               <property name="maxPoolSize" value="100" />
               <property name="queueCapacity" value="500" />
               <property name="keepAliveSeconds" value="60" />
               <property name="rejectedExecutionHandler">
                       <bean class="java.util.concurrent.ThreadPoolExecutor.
↪AbortPolicy" />
               </property>
        </bean>

        <!-- 区块链节点信息配置 -->
        <bean id="channelService" class="cn.webank.channel.client.Service">
               <property name="orgID" value="WB" /> <!-- 配置本机构名称 -->
                       <property name="allChannelConnections">
                              <map>
                                     <entry key="WB"> <!-- 配置本机构的区块链节点列表
（如有DMZ，则为区块链前置）-->
                                            <bean class="cn.webank.channel.
↪handler.ChannelConnections">
                                                   <property name=
↪"connectionsStr">
                                                          <list>
                                                                 <value>
↪NodeA@127.0.0.1:30333</value><!-- 格式：节点名@IP地址:端口，节点名可以为任意名称 -->
                                                          </list>
                                                   </property>
                                            </bean>
                                     </entry>
                              </map>
                       </property>
               </bean>
        </bean>
```

区块链前置配置，如有DMZ区：

```
        <?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
```

(continues on next page)

(continued from previous page)

```
            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">

            <!-- 区块链节点信息配置 -->
                <bean id="proxyServer" class="cn.webank.channel.proxy.Server">
                        <property name="remoteConnections">
                                <bean class="cn.webank.channel.handler.
↪ChannelConnections">
                                        <property name="connectionsStr">
                                                <list>
                                                        <value>NodeA@127.0.0.1:5051
↪</value><!-- 格式: 节点名@IP地址:端口，节点名可以为任意名称 -->
                                                </list>
                                        </property>
                                </bean>
                        </property>

                        <property name="localConnections">
                                <bean class="cn.webank.channel.handler.
↪ChannelConnections">
                                </bean>
                        </property>
                        <!-- 区块链前置监听端口配置，区块链SDK连接用 -->
                        <property name="bindPort" value="30333"/>
                </bean>
        </beans>
```

SDK使用

AMOP的消息收发基于topic（主题）机制，服务端首先设置一个topic，客户端往该topic发送消息，服务端即可收到。

AMOP支持在同一个区块链网络中有多个topic收发消息，topic支持任意数量的服务端和客户端，当有多个服务端关注同一个topic时，该topic的消息将随机下发到其中一个可用的服务端。

服务端代码案例：

```
        package cn.webank.channel.test;

        import java.util.ArrayList;
        import java.util.List;

        import org.slf4j.Logger;
        import org.slf4j.LoggerFactory;
        import org.springframework.context.ApplicationContext;
        import org.springframework.context.support.ClassPathXmlApplicationContext;

        import cn.webank.channel.client.Service;

        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("参数: 接收topic");
```

(continues on next page)

(continued from previous page)

```
                                return;
                        }

                        String topic = args[0];

                        ApplicationContext context = new␣
↪ClassPathXmlApplicationContext("classpath:applicationContext.xml");
                        Service service = context.getBean(Service.class);

                        //设置topic，支持多个topic
                        List<String> topics = new ArrayList<String>();
                        topics.add(topic);
                        service.setTopics(topics);

                        //处理消息的PushCallback类，参见Callback代码
                        PushCallback cb = new PushCallback();
                        service.setPushCallback(cb);

                        //启动服务
                        service.run();
                }
        }
```

服务端的PushCallback类案例：

```
        package cn.webank.channel.test;

        import java.time.LocalDateTime;
        import java.time.format.DateTimeFormatter;

        import org.slf4j.Logger;
        import org.slf4j.LoggerFactory;

        import cn.webank.channel.client.ChannelPushCallback;
        import cn.webank.channel.dto.ChannelPush;
        import cn.webank.channel.dto.ChannelResponse;

        class PushCallback extends ChannelPushCallback {
                static Logger logger = LoggerFactory.getLogger(PushCallback2.
↪class);

                //onPush方法，在收到AMOP消息时被调用
                @Override
                public void onPush(ChannelPush push) {
                        DateTimeFormatter df = DateTimeFormatter.ofPattern("yyyy-
↪MM-dd HH:mm:ss");
                        logger.debug("收到PUSH消息:" + push.getContent());

                        System.out.println(df.format(LocalDateTime.now()) +
↪"server:收到PUSH消息:" + push.getContent());

                        //回包消息
                        ChannelResponse response = new ChannelResponse();
                        response.setContent("receive request seq:" + String.
↪valueOf(push.getMessageID()));
                        response.setErrorCode(0);

                        push.sendResponse(response);
                }
        }
```

客户端案例:

```
        package cn.webank.channel.test;

        import java.time.LocalDateTime;
        import java.time.format.DateTimeFormatter;
        import java.util.Date;
        import java.util.Random;

        import org.slf4j.Logger;
        import org.slf4j.LoggerFactory;
        import org.springframework.context.ApplicationContext;
        import org.springframework.context.support.ClassPathXmlApplicationContext;

        import cn.webank.channel.client.Service;
        import cn.webank.channel.dto.ChannelRequest;
        import cn.webank.channel.dto.ChannelResponse;

        public class Channel2Client {
                static Logger logger = LoggerFactory.getLogger(Channel2Client.
↪class);

                public static void main(String[] args) throws Exception {
                        if(args.length < 1) {
                                System.out.println("参数: 目标topic");
                                return;
                        }

                        String topic = args[0];

                        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();

                        Thread.sleep(2000); //建立连接需要一点时间，如果立即发送消息会失败

                        ChannelRequest request = new ChannelRequest();
                        request.setToTopic(topic); //设置消息topic
                        request.setMessageID(service.newSeq()); //消息序列号，唯一标识
某条消息，可用newSeq()随机生成
                        request.setTimeout(5000); //消息的超时时间

                        request.setContent("request seq:" + request.
↪getMessageID()); //发送的消息内容

                        ChannelResponse response = service.
↪sendChannelMessage2(request); //发送消息

                        System.out.println(df.format(LocalDateTime.now()) + "收到回
包 seq:" + String.valueOf(response.getMessageID()) + ", 错误码:" + response.
↪getErrorCode() + ", 内容:" + response.getContent());
                }
        }
```

测试

按上述说明配置好后，用户指定一个主题：topic，执行以下两个命令可以进行测试。

启动amop服务端：

```
java -cp 'conf/:apps/*:lib/*' cn.webank.channel.test.Channel2Server [topic]
```

启动amop客户端：

```
java -cp 'conf/:apps/*:lib/*' cn.webank.channel.test.Channel2Client [topic] [消息内
容] [消息条数]
```

错误码

- 99：发送消息失败，AMOP经由所有链路的尝试后，消息未能发到服务端，建议使用发送时生成的seq，检查链路上各个节点的处理情况。
- 102：消息超时，建议检查服务端是否正确处理了消息，带宽是否足够。

7.4.3 弹性联盟链共识框架方案

作者：fisco-dev

简介

弹性联盟链共识框架是为了弥补pbft/raft的共识仅对对等节点生效，而无法根据业务逻辑做出一些特化的缺陷所做出的措施。

因为在纯正的pbft/raft的共识模型中，首先就假设每个节点都是平等的，即没有节点具有特殊的业务性质。在联盟链中这样的性质有时并不能满足一些业务场景的要求，故而需要一套在正常的共识流程之外(保证正常共识)继续对共识流程做出限制的规则。这就是弹性联盟链共识框架方案开发的初始原因。

例如如下场景：

> 在一个10(3f+1)个节点组件的联盟链场景中，使用pbft共识算法进行共识。其中10个节点分别由A，B，C三个机构分别持有，A机构有4个节点，B机构有3个节点，C机构有3个节点。根据pbft的性质，达成共识只需要7个节点就足够了(2f+1)，因为pbft对于所有节点都是平等的，所以只需要收够10个节点中的任意7个即可。如果一次投票中的7个节点全部都是A机构和B机构的节点，那么就相当于C机构没有参加共识但是共识仍然还是通过了。所以在这样的情况下就需要额外的规则来限制共识条件，以满足业务层面上的共识要求。

注：这是一个危险的功能，请在正确认识其原理的情况下使用，否则可能轻易导致共识无法运行下去造成链的作废！请一定仔细阅读《多机构弹性共识方案设计》章节及这个章节中的《4.其他注意事项》章节中的内容。

共识的类别

在现有的方案探究及对区块链共识本质的理解下，我们划分对“共识”的界限，认为共识分为以下两类：

数据共识

参照 区块链前沿I上交所朱立：许可链的性能优化

这篇文章中的描述，认为

> 区块链中其实存在两个独立层面的共识，其一可称为“输入共识”，其含义是各节点对指令的顺序及内容达成共识，类似传统数据通信的会话层，不牵涉业务操作；其二可称为“输出共识”，其含义是业务系统受输入的驱动，状态不断发生跃迁，同时产生一系列输出，此时各参与方对业务系统进入的状态及产生的输出达成共识 。此非孤明之见，Corda 区分交易的uniqueness和validity二者[17]， Polkadot区分cononicality与validity二者，都应是有见于此。虽然这两层共识在设计上可以分离，但不同区块链对此的回应差别很大，将其说为区块链设计时面临的一大抉择也不为过。总的来说有三种方式：
>
> 第一种方式是只提供输入共识，此以Factom[18]项目为例。Factom只对数据内容和顺序进行共识，把对数据的后续检验及处理交给其他应用程序完成。由于不做输出共识，这种区块链很可能不内置智能合约功能。
>
> 第二种方式是以紧耦合的方式同时提供输入共识和输出共识，典型案例如以太坊。以太坊的区块头中不仅包含交易根，也包含状态根，通过一套机制同时达成对交易和终态的共识。
>
> 第三种方式是以分离的方式同时提供输入共识和输出共识，典型案例如Fabric 1.0[19]的设计。在Fabric 1.0中，输入共识在Orderer之间达成，Orderer只看到数据，并不理解任何业务含义，输出共识则通过Endorsor、Committer和应用层CheckPoint在一定程度上实现。

因现有的模型继承于以太坊体系，我们认为事实上这里的“输入共识”和“业务共识”都归属于共识框架层面的“数据共识”，表示对于相同的一笔交易，对于这笔交易及其执行的结果，应该在所有的副本上都是一致的，并通过共识算法保证这个要求成立。其中：

- 在每个副本上的同一条交易是相同的
- 对同一条交易的执行结果是相同的

在这两个条件下，所谓“作恶”的可能性为：

- 节点故意丢弃交易(相当于同一条交易没有同步到所有节点上)
- 节点执行交易结果不一致(故意修改交易内容，修改虚拟机实现，修改执行结果)

现有的共识算法下：

- pbft
  - 因为是轮流出块，可以防止第一个作恶(但是注意要是遵照原算法来说，同样无法防止估计丢弃交易)
  - 可以防止第二个作恶
- raft
  - raft是固定leader出块，所以无法防止第一个作恶(但是若修改raft算法使得leader可以切换，可能可以防止，但是raft的本质就是一个强leader的特性，所以这一点存疑)
  - 虽然raft本身无法保证对结果的共识，但是我们实现的raft的方式并没有使用raft的日志同步，而是依托于以太坊区块链自身的同步区块机制，这个同步机制会丢弃和自己交易执行结果不一致的区块，所以只需要修改目前的raft算法可以做到对第二点作恶的防护。

以上，我们归结为“数据共识”

业务共识

注意这里的业务共识不是上一章中引用内容中的“业务共识”的概念。这里的业务共识专门指代： 在业务层面上对数据共识 **结果的数据** 进行解读的共识

以存证为例：假设有3家机构，3家机构按照存证的业务模式进行协调

- 数据共识：每个节点副本执行pbft/raft共识后上链的数据
- 业务共识：当3家机构对于同一笔证据都进行了签名之后这个证据生效

以对用户信用评分为例：假设有3个公司联合对用户信用评级，评级方式是以每个用户为一个合约，合约中有一个数组，每家公司向数组中写入对改用户的分数，最后大家取平均分作为用户评分。

- 数据共识：每个节点副本执行pbft/raft共识后上链的数据

- 业务共识：当3家机构都对用户评级后，用户的信用才会生效

以上，我们归结为“业务共识”

弹性联盟链共识框架方案设计

在定义了业务共识的基础上，我们引入弹性联盟链共识框架方案，即在现有的共识上添加可以**由用户编写的规则**来限制共识是否成功。

目前这个功能只用于PBFT，RAFT还未支持。

注：弹性联盟链共识框架方案是在基础共识的模型上新添加的规则，也就是说需要先满足基础的共识，再满足共识框架规定的规则。

在联盟链的语境下，我们一般认为：联盟链的节点是由各个参与区块链的**“机构”**来控制，在物理上认为节点只是组成区块链的组件，在逻辑上认为参与区块链的应该是**机构**。

所以我们认为参与业务共识的基本元素应该为**机构**和这个**机构的个数**(也就是不再以节点为共识判定的元素)，这样的模型可以覆盖比较大量的场景。

适用的场景：

1. 每家机构必须具备平等的投票权，投票权和投入节点数量无关。比如每个机构都是一票。
2. 共识必须有所有机构都参与，谁都不能缺席。比如存证场景，必须提供证据的，第三方鉴证，仲裁机构都要参加。
3. 共识过程里有些机构必须参加，有些机构可选参加。比如某个场景，司法机构的签名必须存在，否则共识不能达成。
4. 和1不同，机构之间并非完全平等，每个机构具备不同的投票权重，但是不采用节点数表示，而是由签名的加权值表示，比如机构A虽然只投出一票，但是权重为3，机构B一票的权重为2，机构C的权重为1，等等。类似PoS的思想
5. 等等……

等等类似的场景。以上的场景都可以编写为“规则”来加强现有的共识体系。由于在不同的场景下需要不同的规则，所以我们需要提供一个框架来让用户可以根据自己业务的需要来部署不同的规则以达到目的。

弹性联盟链共识框架方案的设计思路

如图所示，以PBFT为例，在执行完pbft的最后阶段时，我们将收集到的签名作为参数来调用一个特殊的系统合约进行“规则”的判定。当判定成功时，表示满足规则，判定失败时，表示收集到的签名还未满足。若还会继续收到共识的包那么会继续进行判定，若是共识包都发送完全，但是仍然未满足条件，那么这个块是无法完成共识阶段，也无法落地的。

举例：

> 若4个节点分别归属于4个机构，然后规则现在设定为必须有A机构的签名才算共识完成。那么比如一个节点收到了除A机构的另两个节点的共识包的时候，按照pbft的模型，加上自己的一票总共3票的时候就已经共识完成了。但是由于弹性联盟链共识框架的限制，此时会判定当前的共识没有完成，需要继续收到A机构的共识包才可判定共识完成。

弹性联盟链共识框架的使用方式

一、启用方式

这个功能的添加是通过系统合约来实现。

机构的名字指定是在节点注册

```
babel-node tool.js NodeAction registerNode xxx.json
```

中 xxx.json 中的 Agencyinfo字段，**当这个字段一致时，标识这两个节点为同一个机构。**

在已经存在的链上启用这个功能

若是在已有的链上需要启用这个功能，那么需要重新部署系统合约(或重新部署NodeAction.sol合约替换现有的节点管理合约并执行setSystemAddr()将系统合约设置进去)，之后重新注册节点进入系统，然后再部署规则(见后文)。

在新链上启用这个功能

只需要按照文档正常部署系统合约即可，在systemcontractv2目录下运行：

```
bable-node deploy.js
```

二、使用规则

所有和该功能相关的合约与脚本工具都位于systemcontractv2目录下

1.编写规则

用户需要根据自己的需要，编写自己的规则。编写规则的文件为在systemcontractv2目录下的ConsensusControl.sol文件中的函数：

ConsensusControl.sol:

```
pragma solidity ^0.4.11;
import "ConsensusControlAction.sol";
contract ConsensusControl is ConsensusControlAction {
          // 构造函数一定需要存在
    function ConsensusControl (address systemAddr) ConsensusControlAction␣
↪(systemAddr) {
    }
          // 用户编写自己规则的函数，true代表满足规则，false代表不满足规则，参数 bytes32[]␣
↪info 代表已经收到的机构的列表， uint[] num 代表已经收到的共识包中这些机构分别的个数(与info是
对应关系)
    function control(bytes32[] info, uint[] num) external constant returns (bool) {
            return true;
    }
    // 初始化时的回调
    function init(address systemAddr) internal {}
    // NodeAction 加入节点的回调，true代表同意这个节点加入，false代表拒绝这个节点加入
          // babel-node tool.js NodeAction registerNode xxxx.json
    function beforeAdd(bytes32 agency) internal returns (bool) {
        return true;
    }
    // NodeAction 退出节点的回调，true代表同意这个节点退出，false代表拒绝这个节点退出
          // babel-node tool.js NodeAction cancelNode xxxx.json
    function beforeDel(bytes32 agency) internal returns (bool) {
        return true;
    }
}
```

用户通过编写填充这个合约中的函数来达成自己目的。

我们以 4 个节点构成链举例，其中 A 机构由2个节点，B机构1个节点，C机构1个节点。

在这个框架中：

- 文件名(ConsensusControl.sol)不可以更改
- 构造函数一定不能删除，且参数一定是address
- 父类ConsensusControlAction会提供两个成员变量 `bytes32[] public agencyList` 和 `mapping (bytes32 => uint) public agencyCountMap` 分别代表当前系统中应该具备的机构列表agencyList 和这些机构应该有的数量。所以agencyList=['A', 'B', 'C'], agencyCountMap={'A':2, 'B':1, 'C':1}
- control函数代表规则，用户通过编写这个规则来根据输入参数判定true或false来决定是否满足自己的业务共识规则，参数 `bytes32[] info` 代表已经收到的机构的列表， `uint[] num` 代表已经收到的共识包中这些机构分别的个数(与info是对应关系)。比如当前触发这次判定的时候，收到了3个共识包，其中2个是A机构的，1个是B机构的，那么 `info=['A','B'],num=[2,1]`
- init代表刚部署该合约时触发的回调
- beforeAdd 代表注册新节点时，可以通过这个函数处理是否可以让这个节点加入，比如需要限制每个机构的最大数量
- beforeDel代表节点退出时，可以通过这个函数处理是否可以让这个节点退出，比如规则为某个机构必须在时才能通过，那么这个函数可以限制这个机构至少要有一个在区块链中才行，否则这条链就无法共识了。
- 修改规则即重新编写 `control init beforeAdd beforeDel`这几个函数，并重新部署。

按照上述介绍，我们列举两种可能的场景的编写规则方式：

1)所有机构都必须参与共识

```
pragma solidity ^0.4.11;
import "ConsensusControlAction.sol";

/**
 * 要求所有机构都有签名
 */
contract ConsensusControl is ConsensusControlAction {
    function ConsensusControl (address systemAddr) ConsensusControlAction␣
↪(systemAddr) {
    }

    function control(bytes32[] info, uint[] num) external constant returns (bool) {
        uint count = 0;
        for(uint i=0; i < info.length; i++) {
            //agencyCountMap[info[i]] != 0 存在这个机构
            // num[i] != 0 这个机构传进来个数大于0
            if (agencyCountMap[info[i]] != 0 && num[i] != 0) {
                count += 1;
            }
        }
        if (count < agencyList.length)
            return false;
        else
            return true;
    }
    // init beforeAdd beforeDel 在这里不起作用，所以不进行覆写
}
```

1)必须有一个指定的机构参与共识

如指定的机构叫做 AgencyA

```
pragma solidity ^0.4.11;

import "ConsensusControlAction.sol";
/**
 * 要求 AgencyA 必须有签名
 */
contract ConsensusControl is ConsensusControlAction {
          string private agencyName = "AgencyA";
    function ConsensusControl (address systemAddr) ConsensusControlAction␣
↪(systemAddr) {
    }

    function control(bytes32[] info, uint[] num) external constant returns (bool) {
        bool isexisted = false;
        for(uint i=0; i < info.length; i++) {
            if (info[i] == stringToBytes32(agencyName) && num[i] > 0) {
                isexisted = true;
                break;
            }
        }
        return isexisted;
    }
    // 使用 beforeDel 限制 AgencyA 的节点退出
    function beforeDel(bytes32 agency) internal returns (bool) {
        // reject del the last "AgencyA" node
        if (agency == stringToBytes32(agencyName)){
            var count = agencyCountMap[agency];
            if (count - 1 <= 0) {
                return false;
            }
        }
        return true;
    }
}
```

2. 部署规则，关闭规则，列出当前机构列表

我们提供了 ConsensusControlTool.js工具来控制规则，改脚本有3个指令'deploy', 'turnoff' 和 'list'
当经过1步我们指定好或修改好规则时候，执行

```
babel-node ConsensusControlTool.js deploy
```

可以把1中的规则合约进行部署。若是想替换规则也同样执行这个指令
当我们需要关闭共识规则时，执行

```
babel-node ConsensusControlTool.js turnoff
```

当我们需要列出当前系统中的机构列表及机构数目时，执行

```
babel-node ConsensusControlTool.js list
```

3. 相关日志查看

这个模块相关的日志全部以 [ConsensusControl]作为开头。其中在合约中也可以编写日志输出(详

细使用见合约日志输出相关文章)，如：

```
pragma solidity ^0.4.11;

import "ConsensusControlAction.sol";
import "LibEthLog.sol";
contract ConsensusControl is ConsensusControlAction {
    using LibEthLog for *;

    function ConsensusControl (address systemAddr) ConsensusControlAction␣
↪(systemAddr) {
        LibEthLog.DEBUG().append("[ConsensusControl] ######### current agency list
↪######## length:").append(agencyList.length).commit();
        // for (uint i=0; i < agencyList.length; i++) {
        //     LibEthLog.DEBUG().append("[ConsensusControl]").append(i)
        //     .append(":")
        //     .append(bytes32ToString(agencyList[i]))
        //     .append(":")
        //     .append(agencyCountMap[agencyList[i]])
        //     .commit();
        // }
        // LibEthLog.DEBUG().append("[ConsensusControl] ######### end ########:").
↪commit();
    }
}
```

即可输出debug日志。

4. 其他注意事项

1. 机构的名字一定使用英文，并且不能超过bytes32的限制(solidity的限制)
2. **在工程上节点的部署一定要保证和规则要求相关**，比如当规则规定某个机构必须签名时，**这个机构**的节点一定要保证多活，否则会导致共识无法继续。比如当规则规定所有机构都要签名时，**每个机构**的节点**都要**保证多活。**因为实现上的关系，若由于规则的限制而同时不能保证满足规则的意外场景发生时，会让系统陷入死锁。**比如4个节点分别归属4个机构，然后必须要求其中一个机构签名。因为这个机构只有一个节点，那么当这个节点无法工作时，规则会判定共识失败。若不能在短时间内恢复这个节点，那么一段时间后，即使再次恢复这个节点，也无法使整个系统继续工作，因为此时整个系统已经陷入了死锁。所以在工程上目前需要保持这个机构一定要多活(即规则限定的条件要满足)。若在意外的情况下触发了这一条，那么目前只能把所有节点都重新启动可恢复。**在下个版本会改进这个缺陷**。
3. **重要！！！**部署规则的时候一定要仔细检查规则是否能够正常运行，否则若部署了一个错误的规则的时候可能会导致链的作废(举例：若当前规则为指定某个机构必须签名，但是这个链的体系中并没有这个机构存在，那么若部署了这个合约，则会导致无法出块。)

在老链(已有数据的链)上兼容或启用弹性联盟链共识框架

该功能的添加不会影响到已有的功能，与已有的数据相兼容，所以当编译出新的执行文件的时候，直接替换已有的执行文件重启后即可。但是这样不会具备弹性联盟链共识框架这个功能，只是保证不会影响其他功能。

若希望在已有的链上启用这个新的功能，则要求**重新部署系统合约**，或至少重新部署 NodeAction.sol 及 ConsensusControlMgr.sol，并将这两个合约重新注册到已有的系统合约中，之后才能使用 ConsensusControlTool.js 工具

也就是说在已有的链中替换了新版本的执行文件并希望启用该功能，需要：

- 重新部署一次系统合约，并重新执行之前对系统合约管理过的组件相关操作，并把**除了**NodeAction.sol之外的**原来的系统合约所管理的合约组件**重新注册到新的系统合约中(首

先先获取原系统合约管理组件的地址及对应关系，并参考 systemcontractv2/deploy.js 文件的写法重新注册到新系统合约中)，之后重新按照原来添加节点的方式重新添加节点恢复到原来的连接状态

- 重新部署 `NodeAction.sol` 及 `ConsensusControlMgr.sol`(参考 systemcontractv2/deploy.js 文件)，并将这两个合约重新注册到已有的系统合约中，并重新按照原来添加节点的方式重新添加节点恢复到原来的连接状态

7.4.4 可扩展的虚拟机指令ethcall

EthCall设计文档

一、功能介绍

1、描述

EthCall 是一个连接Solidity和C++的通用编程接口，内置在EVM中。开发者可将Solidity中复杂的、效率低的、不可实现的逻辑，通过C++语言实现，并以EthCall接口的形式供给Solidity。Solidity即可通过EthCall接口，以函数的方式直接调用C++的逻辑。在保证Solidity语言封闭性的前提下，提供了更高的执行效率和更大的灵活性。

2、背景

目前Solidity语言存在以下两方面问题：

（1）Solidity 执行效率低

由Solidity语言直接实现的合约，执行的效率较低。原因是Solidity的底层指令OPCODE，在执行时需要翻译成多条C指令进行操作。对于复杂的Solidity程序，翻译成的C指令非常庞大，造成运行缓慢。而直接使用C代码直接实现相应功能，并不需要如此多的指令。 若采用C++来实现合约中逻辑相对复杂、计算相对密集的部分，将有效的提高合约执行效率。

（2）Solidity 局限性

Solidity语言封闭的特性，也限制了其灵活性。Solidity语言缺乏各种优秀的库，开发调试也较为困难。若Solidity语言能够直接调用各种C++的库，将大大拓展合约的功能，降低合约开发的难度。

因此，需要提供一种接口，让Solidity能够调用外部的功能，在保证Solidity语言封闭性的前提下，提高执行效率，增加程序的灵活性。

3、相关工作

为解决上述问题，目前已有两种解决方案：

（1）SHA3接口重封装

BCOS中采用的方式。在OPCODE的SHA3接口上封装一层，通过传入的字符串标识符判断，是普通的SHA3功能，还是拓展功能。此方式在每次调用时都需要将字符串转成参数，接口效率低。且实现不够优雅。

（2）Precompiled Contracts

Eth中内置的方式。在EVM中内置预编译合约，通过Solidity语言中的关键字进行调用。此方式拓展性不好，增加功能时需多次修改编译器。且调用功能时，采用消息式（而不是函数式）的调用，在调用和返回时，都需要反复打包和解包，效率低，实现复杂。

4、EthCall

EthCall为Solidity提供了一种**函数式**调用底层C++模块的方法。

举例–同态加密

在Solidity中，传递两个密文d1，d2，返回密文同态加的结果ret。

```
//Solidity
function paillierAdd(string d1, string d2) internal constant returns (string)
{
        string memory ret = new string(bytes(d1).length);
        uint32 call_id = callId();
        uint r;
        assembly{
                //函数式调用，结果直接写到ret中，类似引用的方式
                r := ethcall(call_id, ret, d1, d2, 0, 0, 0, 0, 0, 0)    ///<-----␣
↪ethcall
        }
        return ret;
}
```

在C++中，根据调用的参数，编写函数ethcall()，实现逻辑。

```
//C++
u256 ethcall(vector_ref<char> result, std::string d1, std::string d2)
{
   /*
   *         实现同态加密逻辑，结果直接写到result中
   */
    return 0;
}
```

优势

（1）函数式调用，支持传递引用参数。调用接口开销低，编程实现简单方便。

（2）C++代码运行复杂逻辑，执行效率高。

（3）可调用C++库，功能强大，灵活性强，降低合约开发难度。

二、使用方法

EthCall的使用，可分为两部分。在Solidity端，向EthCall的传参，并使用支持EthCall的编译器进行编译。在C++端，编写与Solidity端参数对应的接口函数，并实现需要实现的逻辑。具体描述如下。

1、Solidity调用EthCall

编译器

支持EthCall的编译器项目的根目录下，为fisco-solc。使用方式与一般的Solidity编译器相同。

接口调用实现

（1）确定一个call id，唯一标识底层C++需实现的功能。

（2）确定需要传递的参数。若是string或bytes，需定义为memory类型（外层函数的参数默认为memory类型）。

（3）调用ethcall接口，参数依次排列，call id为第一个，接着为其它参数，参数数量不足10个用0填充。

举例如下：

```
uint32 callId = 0x66666;
int a;
int b;
bytes memory bstr;
string memory str;
uint r;
assembly{
        r := ethcall(callId, a, b, bstr, str, 0, 0, 0, 0, 0)
}
```

更完整的例子可参考tool/LibEthLog.sol和tool/LibPaillier.sol。

2、C++处理EthCall

在C++中，需编写与Solidity调用相对应的函数，实现EthCall调用的处理。操作目录在 libevm/ethcall 。实现过程如下（以 EthCallDemo 为例）。

（1）在EthCallEntry.h 中的 EthCallIdList 中声明一个call id，与Solidity中的call id相对应。

```
// EthCallEntry.h 中
enum EthCallIdList:callid_t
{
    LOG         = 0x10,
    PAILLIER    = 0x20,
    DEMO        = 0x66666       ///< demo，与Solidity中相对应
};
```

（2）新建 EthCallDemo.h。若需要 EthCallDemo.cpp，则在 libevm/CMakeLists.txt 中添加相应的编译选项（为了说明简单，本例没有cpp文件）。

（3）在 EthCallDemo.h 中实现EthCall逻辑。需要注意几点：

a. 引用 `#include "EthCallEntry.h"`

b. 建一个新类，继承`EthCallExecutor`模板类，模板中的参数与Solidity中的ethcall接口调用时的参数类型对应（Solidity与C++的参数类型对应关系请参考 EthCallEntry.h）。

c. 在构造函数中调用`registerEthcall()`，将call id与此要实现的类绑定在一起。

d. 实现`ethcall()`函数，功能逻辑在此函数中实现。函数的参数同样需与Solidity中调用的参数对应。

e. 实现`gasCost()`函数，编写gas消耗的逻辑。

完整的代码如下：

```
#pragma once

#include "EthCallEntry.h"
#include <libdevcore/easylog.h>
#include <libdevcore/Common.h>
#include <string>
#include <vector>

using namespace std;
using namespace dev;
using namespace dev::eth;

namespace dev
{
namespace eth
{

class EthCallDemo : EthCallExecutor<int, int, vector_ref<byte>, string> //按顺序定义除开callId之外的类型
```

(continues on next page)

(continued from previous page)

```
{
public:
    EthCallDemo()
    {
        //使用刚刚声明的CallId将EthCallDemo注册到ethCallTable中
        LOG(DEBUG) << "ethcall bind EthCallDemo--------------------";
        this->registerEthcall(EthCallIdList::DEMO); //EthCallList::DEMO =␣
↪0x66666为ethcall的callid
    }

    u256 ethcall(int a, int b, vector_ref<byte>bstr, string str) override //必
须u256，override很重要
    {
        LOG(INFO) << "ethcall " <<  a + b << " " << bstr.toString() << str;

        //用vector_ref定义的vector可直接对sol中的array赋值
        //vector_ref的用法与vector类似，可参看libdevcore/vector_ref.h
        bstr[0] = '#';
        bstr[1] = '#';
        bstr[2] = '#';
        bstr[3] = '#';
        bstr[4] = '#';
        bstr[5] = '#';

        return 0; //返回值写入sol例子中的r变量中
    }

    uint64_t gasCost(int a, int b, vector_ref<byte>bstr, string str) override //必
须uint64_t，override很重要
    {
        return sizeof(a) + sizeof(b) + bstr.size() + str.length(); //消耗的gas一般与处
理的数据长度有关
    }
};
///EthCallDemo举例结束///
}
}
```

（4）在 EthCallEntry.cpp 引用 EthCallDemo.h，并且在 EthCallContainer 中实例化 EthCallDemo。

```
// EthCallEntry.cpp 中
#include "EthCallDemo.h"

...

class EthCallContainer
{
public:
    //在此处实例化EthCall的各种功能
    EthLog ethLog;
    Paillier ethPaillier;
    EthCallDemo ethCallDemo;    ///< demo 在此处实例化
};
```

更完整的实现，可参考libevm/ethcall下的 EthLog 和 Paillier。

三、具体设计细节

###1、增加一条OPCODE

Solidity语言在运行前，被 solc 编译成 OPCODE。OPCODE 在 EVM 中被执行。EthCall 接口，通过增加一条OPCODE实现，新的OPCODE名为ETHCALL。所有EthCall的功能都通过此OPCODE调用，功能的区分通过call id完成。在编译器solc的代码 libevmcore/Instruction.h 和 libevmcore/Instruction.cpp 中，添加ETHCALL。之后重新编译solc的代码（即得到fisco-solc），即可编译别包含ethcall的的Solidity程序。再在fisco-bcos的代码的相同位置添加ETHCALL，并编写相应逻辑，即可识别ETHCALL。

定义OPCODE ETHCALL：

```
// Instruction.h
enum class Instruction: uint8_t
{
        STOP = 0x00,         ///< halts execution
        ADD,                 ///< addition operation
        MUL,                 ///< mulitplication operation
        ...
        SHA3 = 0x20,         ///< compute SHA3-256 hash
        ETHCALL = 0x2f,      ///< call eth kernel function api    <---------新增加
的OPCODE

        ADDRESS = 0x30,      ///< get address of currently executing account
        BALANCE,             ///< get balance of the given account
        ORIGIN,              ///< get execution origination address
        ...
```

添加ETHCALL 的处理逻辑：

```
// libevm/VM.cpp
void VM::interpretCases()
{
  ...
                  CASE_BEGIN(ADD)
                //OPCODE ADD 处理逻辑
                CASE_END

        CASE_BEGIN(MUL)
                //OPCODE MUL 处理逻辑
                CASE_END

        CASE_BEGIN(SHA3)
                //OPCODE SHA3 处理逻辑
                CASE_END

#ifdef EVM_ETHCALL
                CASE_BEGIN(ETHCALL)                 // <------ case ETHCALL
                {
                        ON_OP();
                        u256 ret = ethcallEntry(this, m_sp); // <------ 传入堆栈指针，
供parer用
                        m_sp -= 9;
                        *m_sp = ret;
                        ++m_pc;
                }
                CASE_END
#endif
```

###2、参数Parser

在Solidity调用EthCall接口，参数以堆栈的形式传给底层模块。底层模块为了能够正确获取到调用参数，需要依次出栈，并将Solidity类型的数据转换成C++类型的数据。在此处实现了一个Parser来完成此功能。

以第一节给出的同态加法例子为例，C++中可编写如下Parser程序获取调用参数。

```
//Parser举例--同态加参数获取
EthCallParamParser parser;
parser.setTargetVm(vm);

callid_t callId;
string d1, d2;
vector_ref<char> result;          //<-------- 返回值定义成引用类型
parser.parse(sp, callId, result, d1, d2);  //<---------- 根据堆栈指针sp，建立参数的映射，获取参数值
```

在实际实现时，开发者并不需要写上面的代码，Parser对开发者来说是隐藏的，EthCall的接口实现会自动实现上述代码，并建立参数的映射关系。开发者要做的，只是根据Solidity调用ethcall的参数，编写C++的`ethcall()`函数即可。设计细节将在"自动参数映射"一节给出。

###3、Call ID

Call ID的实现，使得EthCall仅仅只需添加一条OPCODE，即可实现各种功能。开发者在Solidity端调用ethcall时，传入的第一个参数为Call ID，在底层，通过Call ID索引不同的模块，完成不同的功能。

```
// libevm/ethcall/EthCallEntry.cpp
u256 ethcallEntry(VM *vm, u256* sp)
{
    EthCallParamParser parser;
    parser.setTargetVm(vm);

    callid_t callId;
    parser.parse(sp, callId);      //<-----  先parse出Call ID
    --sp;

    ethCallTable_t::iterator it = ethCallTable.find(callId); //<------- 根据Call ID索引不同的模块
    if(ethCallTable.end() == it)
    {
        LOG(ERROR) << "EthCall id " << callId << " not found";
        BOOST_THROW_EXCEPTION(EthCallNotFound());//Throw exception if callId is not found
    }

    return it->second->run(vm, sp);    //<---------- 调用相应模块
}
```

###4、自动参数映射

Parser对于开发者来说，是隐藏的。开发者直接根据Solidity调用ethcall时的参数，编写C++的`ethcall()`函数。EthCall就会自动为其建立参数映射关系，自动填充参数的值。

自动参数映射，将parser隐藏，开发者不需要自己写parser，而是直接根据Solidity调用ethcall时的参数，编写C++的`ethcall()`函数。

例如在Solidity中：

```
r := ethcall(call_id, ret, d1, d2, 0, 0, 0, 0, 0, 0)
```

C++对应实现`ethcall()`函数，参数会自动建立映射关系，填充到C++函数的`result`，`d1`，`d2`中：

```
u256 ethcall(vector_ref<char> result, std::string d1, std::string d2)
{
    .........
}
```

实现方式是通过可变参数模板实现的（更多细节请看源代码libevm/ethcall/EthCallEntry.h）。

```
// libevm/ethcall/EthCallEntry.h
template<typename ... T>
class EthCallExecutor : EthCallExecutorBase
{
public:
    void registerEthcall(enum EthCallIdList callId)
    {
        assert(ethCallTable.end() == ethCallTable.find(callId));

        ethCallTable[callId] = (EthCallExecutorBase*)this;
    }

    u256 run(VM *vm, u256* sp) override  //<---------- 模块运行函数 run()
    {
        EthCallParamParser parser;
        parser.setTargetVm(vm);
        std::tuple<T...> tp;//通过tuple的方式实现可变长的参数，能够对参数parse并向func传参
        parser.parseToTuple(sp, tp);  //<-------  调用parser，建立映射，填充参数值
        return runImpl(vm, tp, make_index_sequence<sizeof...(T)>());
    }

    template<typename TpType, std::size_t ... I>
    u256 runImpl(VM *vm, TpType& tp, index_sequence<I...>)
    {
        //自动把tuple中的参数展开，传入ethcall中
        this->updateCostGas(vm, gasCost(std::get<I>(tp)...));
        return ethcall(std::get<I>(tp)...);  //<------ 自动把变长参数展开，调
用ethcall()
    }

    virtual u256 ethcall(T...args) = 0;
    virtual uint64_t gasCost(T ...args) = 0;

};
```

EthCall 说明文档

一、功能介绍

1、描述

EthCall 是一个连接Solidity和C++的通用编程接口，内置在EVM中。开发者可将Solidity中复杂的、效率低的、不可实现的逻辑，通过C++语言实现，并以EthCall接口的形式供给Solidity。Solidity即可通过EthCall接口，以函数的方式直接调用C++的逻辑。在保证Solidity语言封闭性的前提下，提供了更高的执行效率和更大的灵活性。

2、背景

目前Solidity语言存在以下两方面问题：

（1）Solidity 执行效率低

由Solidity语言直接实现的合约，执行的效率较低。原因是Solidity的底层指令OPCODE，在执行时需要翻译成多条C指令进行操作。对于复杂的Solidity程序，翻译成的C指令非常庞大，造成运行缓慢。而直接使用C代码直接实现相应功能，并不需要如此多的指令。 若采用C++来实现合约中逻辑相对复杂、计算相对密集的部分，将有效的提高合约执行效率。

（2）Solidity 局限性

Solidity语言封闭的特性，也限制了其灵活性。Solidity语言缺乏各种优秀的库，开发调试也较为困难。若Solidity语言能够直接调用各种C++的库，将大大拓展合约的功能，降低合约开发的难度。

因此，需要提供一种接口，让Solidity能够调用外部的功能，在保证Solidity语言封闭性的前提下，提高执行效率，增加程序的灵活性。

3、优势

EthCall为Solidity提供了一种**函数式**调用底层C++模块的方法。存在以下优势：

（1）函数式调用，支持传递引用参数。调用接口开销低，编程实现简单方便。

（2）C++代码运行复杂逻辑，执行效率高。

（3）可调用C++库，功能强大，灵活性强，降低合约开发难度。

二、使用方法

EthCall的使用，可分为两部分。在Solidity端，向EthCall的传参，并使用支持EthCall的编译器进行编译。在C++端，编写与Solidity端参数对应的接口函数，并实现需要实现的逻辑。具体描述如下。

1、Solidity调用EthCall

编译器

支持EthCall的编译器在项目的根目录下，为fisco-solc。使用方式与一般的Solidity编译器相同。

接口调用实现

（1）确定一个call id，唯一标识底层C++需实现的功能。

（2）确定需要传递的参数。若是string或bytes，需定义为memory类型（外层函数的参数默认为memory类型）。

（3）调用ethcall接口，参数依次排列，call id为第一个，接着为其它参数，参数数量不足10个用0填充。

举例如下：

```
uint32 callId = 0x66666;
int a;
int b;
bytes memory bstr;
string memory str;
uint r;
assembly{
        r := ethcall(callId, a, b, bstr, str, 0, 0, 0, 0, 0)
}
```

更完整的例子可参考tool/LibEthLog.sol和tool/LibPaillier.sol。

2、C++处理EthCall

在C++中，需编写与Solidity调用相对应的函数，实现EthCall调用的处理。操作目录在 libevm/ethcall 。实现过程如下（以 EthCallDemo 为例）。

（1）在EthCallEntry.h 中的 EthCallIdList 中声明一个call id，与Solidity中的call id相对应。

```
// EthCallEntry.h 中
enum EthCallIdList:callid_t
{
    LOG          = 0x10,
    PAILLIER     = 0x20,
    DEMO         = 0x66666        ///< demo，与Solidity中相对应
};
```

（2）新建 EthCallDemo.h。若需要 EthCallDemo.cpp，则在 libevm/CMakeLists.txt 中添加相应的编译选项（为了说明简单，本例没有cpp文件）。

（3）在 EthCallDemo.h 中实现EthCall逻辑。需要注意几点：

a. 引用 `#include "EthCallEntry.h"`

b. 建一个新类，继承`EthCallExecutor`模板类，模板中的参数与Solidity中的ethcall接口调用时的参数类型对应（Solidity与C++的参数类型对应关系请参考 EthCallEntry.h）。

c. 在构造函数中调用`registerEthcall()`，将call id与此要实现的类绑定在一起。

d. 实现`ethcall()`函数，功能逻辑在此函数中实现。函数的参数同样需与Solidity中调用的参数对应。

e. 实现`gasCost()`函数，编写gas消耗的逻辑。

完整的代码如下：

```
#pragma once

#include "EthCallEntry.h"
#include <libdevcore/easylog.h>
#include <libdevcore/Common.h>
#include <string>
#include <vector>

using namespace std;
using namespace dev;
using namespace dev::eth;

namespace dev
{
namespace eth
{

class EthCallDemo : EthCallExecutor<int, int, vector_ref<byte>, string> //按顺序定义除开callId之外的类型
{
public:
    EthCallDemo()
    {
        //使用刚刚声明的CallId将EthCallDemo注册到ethCallTable中
        LOG(DEBUG) << "ethcall bind EthCallDemo--------------------";
        this->registerEthcall(EthCallIdList::DEMO); //EthCallList::DEMO =
↪0x66666为ethcall的callid
    }

    u256 ethcall(int a, int b, vector_ref<byte>bstr, string str) override //必须u256，override很重要
    {
        LOG(INFO) << "ethcall " <<  a + b << " " << bstr.toString() << str;

        //用vector_ref定义的vector可直接对sol中的array赋值
        //vector_ref的用法与vector类似，可参看libdevcore/vector_ref.h
        bstr[0] = '#';
        bstr[1] = '#';
```

(continues on next page)

(continued from previous page)

```
        bstr[2] = '#';
        bstr[3] = '#';
        bstr[4] = '#';
        bstr[5] = '#';

        return 0; //返回值写入sol例子中的r变量中
    }

    uint64_t gasCost(int a, int b, vector_ref<byte>bstr, string str) override //必
须uint64_t, override很重要
    {
        return sizeof(a) + sizeof(b) + bstr.size() + str.length(); //消耗的gas一般与处
理的数据长度有关
    }
};
///EthCallDemo举例结束///
}
}
```

（4）在 EthCallEntry.cpp 引用 EthCallDemo.h ，并且在 EthCallContainer 中实例化 EthCallDemo。

```
// EthCallEntry.cpp 中
#include "EthCallDemo.h"

...

class EthCallContainer
{
public:
    //在此处实例化EthCall的各种功能
    EthLog ethLog;
    Paillier ethPaillier;
    EthCallDemo ethCallDemo;    ///< demo 在此处实例化
};
```

更完整的实现，可参考libevm/ethcall下的 EthLog 和 Paillier。

三、日志、异常处理方法

1. 程序运行时，log报错：EthCall id XX not found

找不到相应的call id，请检查`EthCallContainer`中是否有你的类的实例化定义。

2. Solidity编译不通过

请确认：

（1）是否用的 fisco-solc 编译器进行编译。

（2）ethcall调用时是否用0补齐10个参数。

3. C++程序编译不通过

请确认：

（1）EthCallEntry.cpp 是否include了你的.h文件。你的.h文件是否include了 EthCallEntry.h

（2）参数对应是否正确：`EthCallExecutor`类模板参数，`ethcall()`参数，`gasCost()`参数。对应参数时，无需对应call id的类型。

（3）C++端的ethcall在定义时是否加了override关键字，返回值类型必须为`u256`。 。

（4）目前不支持array(vector) 类型的调用。

4. C++的ethcall接口无法正确获取Solidity调用的值

请确认：

（1）参数对应是否正确：`EthCallExecutor`类模板参数，`ethcall()`参数，`gasCost()`参数。无需对应call id的类型。

（2）C++端的ethcall在定义时是否加了override关键字。

（3）string，bytes是否定义成memory类型。

（4）目前不支持array(vector) 类型的调用。

7.5 安全

7.5.1 FISCO BCOS权限模型

作者：fisco-dev

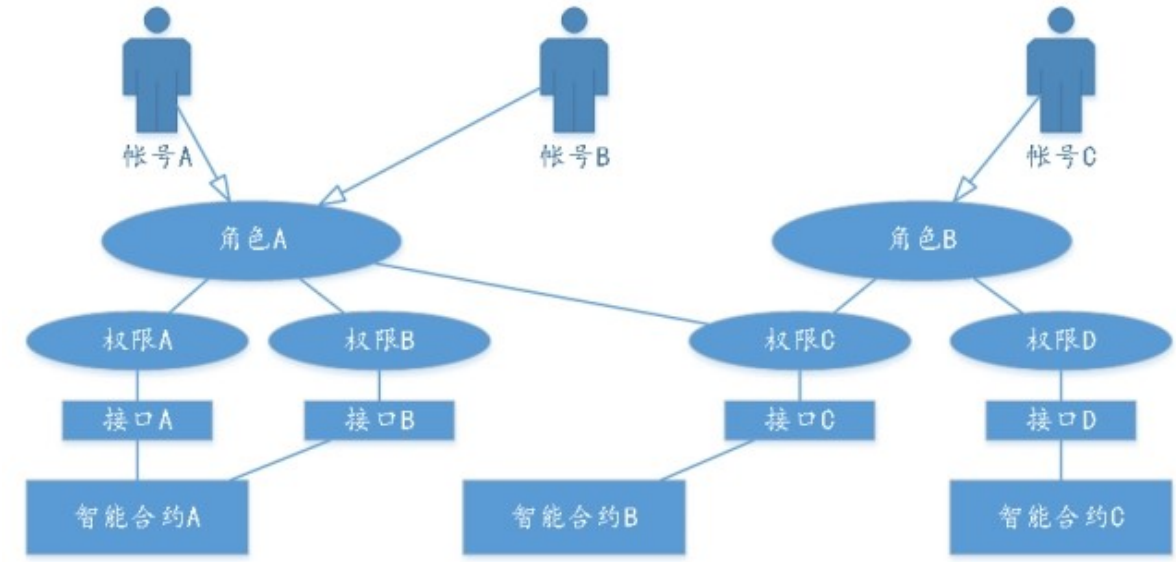
1、FISCO BCOS权限模型（ARPI）介绍

与可自由加入退出、自由交易、自由检索的公有链相比，联盟链有准入许可、交易多样化、基于商业上隐私及安全考虑、高稳定性等要求。因此，联盟链在实践过程中需强调“权限”及“控制”的理念。

为体现“权限”及“控制”理念，FISCO BCOS平台基于**系统级权限控制**和**接口级权限控制**的思想，提出ARPI(Account—Role—Permission—Interface)权限控制模型。

系统级权限控制指从系统级别控制一个账号能否部署新合约，以及能否发起对已有合约的调用。在一账号发起请求后到具体部署合约（或调用合约）的操作之前，由系统进行判断和控制，拒绝越权的操作。接口级权限控制是指权限粒度细化到合约的接口级别，当一个新合约部署生效后，管理员可赋予某个账号调用该合约全部或部分接口的权限。

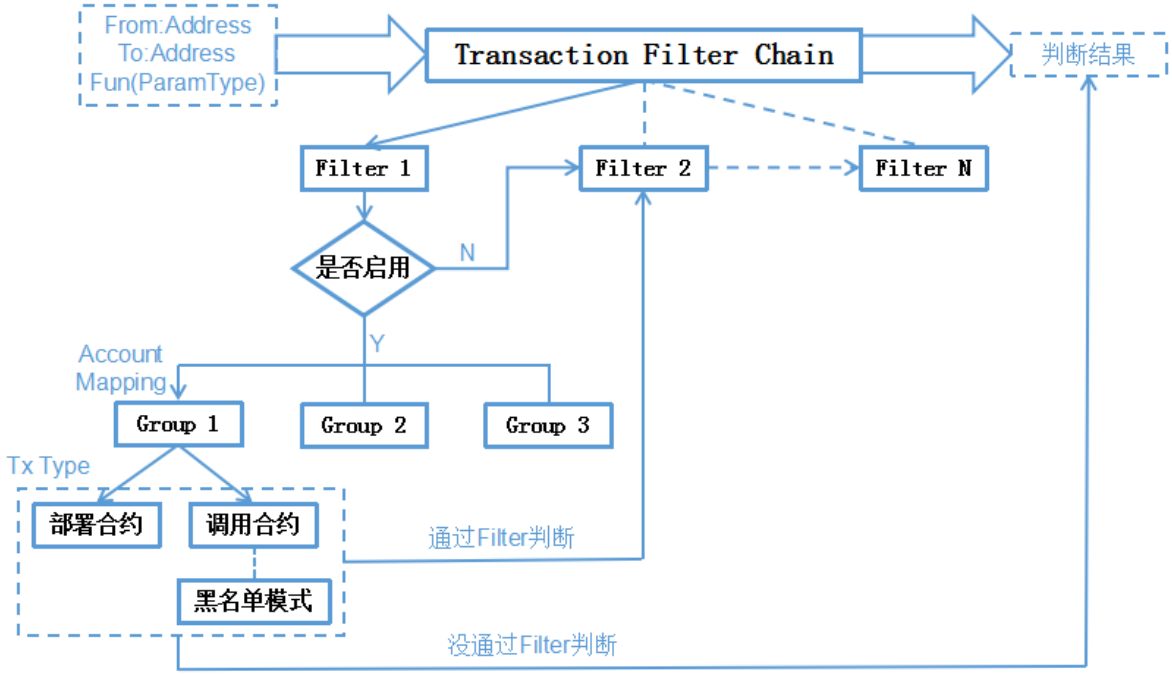
在**ARPI权限控制模型**中，分别有**账号**、**角色**、**权限**、**接口**四类对象。账号和角色的对应关系为N:1，即一个账号（个人或组织）只能对应到一个角色，但一个角色可以包含零个到多个账号。在实际操作中，同一角色可能有多个账号，但每个账号使用独立且唯一的公私钥对，发起交易时使用其私钥进行签名，接收方可通过公钥验签知道交易具体是由哪个账号发出，实现交易的可控及后续监管的追溯。角色与权限的对应关系为N:M，即一个角色拥有多个权限的集合，同时一个权限也能被多个角色所拥有。ARPI模型中权限粒度将细化到合约的接口级别，即一角色下的账号如拥有某个权限，则能调用该权限下智能合约的一个或多个接口。四类对象的整体关系可用下图描述。



账号、角色及权限

2、FISCO BCOS权限模型（ARPI）框架实现

TransactionFilterChain合约在部署系统代理合约时，将首次被部署。系统把TransactionFilterChain合约地址记录在系统路由表中，作为权限控制判断的入口，后续整个系统内权限相关的增删改查也将通过系统合约进行维护。所有权限内容记录在区块链上。交易请求发起后，系统将访问系统合约表查询该交易发起方是否有对应的权限。如果具有权限，执行交易；如果不具备，则抛出异常提示。权限控制判断的整体流程见下图描述。



权限控制判断流程

上述所提及的权限行为包括：A)能否部署合约，只有通过审核的合约才能发布到链上及供后续执行；B)能否调用合约，只有具有特定权限的账号才能通过对合约接口的调用来实现特定功能的系统控制及业务交易。

对权限控制判断流程的进一步说明：

2.1、TransactionFilterChain合约中可包含数个相互独立的Filter，每个Filter为具体的权限控制判断逻辑。用户只有通过整条TransactionFilterChain上所有Filter的权限判断，才能认为通过系统的权限控制判断，具备特定权限。每个Filter可由联盟链中不同成员负责管理，单独添加具体权限。

2.2、对于每一个Filter，均有是否**启用权限控制**的状态区分，当处于启用权限控制状态时，才进行Filter的权限判断。Filter初始化为关闭权限控制。

2.3、一个Filter内部有不同的Group，当进行权限控制判断时，先根据发起交易方（Account）所属的角色（Role），找到对应的Group及权限列表（Permission）。

2.4、Group内部将区分是**部署合约**还是**调用合约**的交易类型，部署合约的权限判断逻辑的输入是发起交易地址，调用合约的权限判断逻辑的输入是发起交易地址、被调用合约地址、调用函数及参数列表。Group初始化为没有部署合约权限。

2.5、在调用合约判断过程中，存在**黑名单模型**的判断。即使链上记录某角色具有调用一接口权限，但如果启用黑名单模式，权限控制的判断接口将返回为没有权限。Group初始化为非黑名单模型。

2.6、智能合约的函数名+参数类型列表组成一个**接口描述**。

2.7、**合约再次部署后地址将发生变化，需重新注册权限。**

3、FISCO BCOS在联盟链权限控制上的实践

角色和权限总体来说是与场景强相关的，不同业务规则和系统构建，将有不同的角色和权限规划。FISCO BCOS根据以往经验，在此列出供参考的角色体系定义及角色对应的活动权限示例，作为权限控制的一个模型。

- **链超级管理者（上帝账号）**：链超级管理者由联盟链管理委员会或公选出来的人员担任，具有所有权限，包括赋予权限的权限，可部署及执行所有合约。**其权限描述为**：给用户分配角色，给角色分配权限，含所有操作的权限。系统合约的执行需使用上帝账号执行。
- **链/Filter管理者**：链/Filter管理者的级别仅次于链超级管理者，可作为各Filter的管理人员，负责审核、修改及删除链上的节点和账号相关资料，及合约信息。**其权限描述为**：执行CAAction、NodeAction、ContractABIAction、SystemProxy合约。
- **运维管理人员**：运维管理人员是实施联盟链的非系统合约的部署和运维活动的人员，负责发布和管理应用，对节点物理资源及配置参数进行修改，不参与业务交易。**其权限描述为**：执行ConfigAction系统配置合约、部署非系统类合约（由于需从系统合约获取ConfigAction进行操作，因此也具有SystemProxy权限）。对于运维管理人员部署的非系统合约，如需使该合约全网生效（基于合约名访问而非合约地址访问合约），需链管理者通过操作ContractABIAction授权。
- **交易人员A、B...**：交易操作人员是指使用平台进行商业交易操作的人员，交易人员发起业务的交易并查询交易执行结果，不同业务场景下的交易人员可再细分角色。**其权限描述为**：执行和查询业务合约（基于业务可再细分不同交易角色）。
- **链监管人员（可以不单独作为Group）**：链监管人员负责制定权限规范，审查交易数据，一般不参与联盟链的管理，但可参加到日常交易业务中。**其权限描述为**：对操作记录的追溯（以Event方式记录部署合约和调用合约的输入要素作为审计事件）。

4、脚本使用说明

4.1、systemcontract目录下ARPI_Model.js脚本提供一键启用FISCO BCOS权限控制模型ARPI的功能。脚本操作包括启用权限控制状态，并根据**3、FISCO BCOS在联盟链权限控制上的实践**中的内容设置角色和权限。**切记**：执行脚本后，系统将启用权限控制状态，如不需要，可使用上帝账号关闭该权限控制，否则将意外地影响其他账号对合约的部署及调用。

4.2、同时systemcontract目录下提供一个AuthorityManager.js脚本，用于对TransactionFilterChain进行管理，面向FilterChain、Filter和Group三个对象提供操作查询接口。**AuthorityManager.js和ARPI_Model.js脚本均需使用上帝账号执行。**

4.2.1、FilterChain在整个权限控制框架中属于一级位置，不需索引即可获取FilterChain对象。对于FilterChain，提供了Filter上Chain、下Chain、查询及一键还原功能。具体接口说明如下：

```
babel-node AuthorityManager.js FilterChain addFilter          // 添加Filter到系统合约Chain，后加Filter名称、版本、描述三个参数
                               delFilter          // 从系统合约Chain删除Filter，后加Filter序号参数
                               showFilter         // 显示系统合约Chain的Filter
                               resetFilter        // 还原系统合约Chain到初始状态
```

FilterChain功能

4.2.2、Filter在整个权限控制框架中属于二级位置，需提供Filter在整个FilterChain中的索引才能获取该Filter对象。对于Filter，提供了启停查该Filter权限控制状态的功能，以及给账号添加新增或已有的Group（权限列表）、显示账号目前所属Group的功能。具体接口说明如下，执行时均需提供Filter序号：

```
babel-node AuthorityManager.js Filter      getFilterStatus          // 显示该Filter权限控制的状态，后加序号参数
                                enableFilter          // 启用该Filter权限控制，后加序号参数
                                disableFilter         // 关闭该Filter权限控制，后加序号参数
                                setUsertoNewGroup     // 给账号添加新角色，后加Filter序号、账号两个参数
                                setUsertoExistingGroup // 给账号添加已有角色，后加Filter序号、账号、已有Group角色三个参数
                                listUserGroup         // 显示账号的角色，后加Filter序号、账号两个参数
```

Filter功能

4.2.3、Group在整个权限控制框架中属于三级位置，需提供Filter在整个FilterChain中的索引以及Group绑定的账号才能获取该Group对象。对于Group，提供了对权限列表的增删查、部署合约和黑名单的启停接口及一账号/角色的权限列表。具体接口说明如下，执行时均需提供Filter序号和账号：

```
babel-node AuthorityManager.js Group       getBlackStatus           // 显示该角色黑名单模式的状态，后加Filter序号、账号两个参数
                                enableBlack           // 启用该角色黑名单模式，后加Filter序号、账号两个参数
                                disableBlack          // 关闭该角色黑名单模式，后加Filter序号、账号两个参数
                                getDeployStatus       // 显示该角色发布合约功能的状态，后加Filter序号、账号两个参数
                                enableDeploy          // 启用该角色发布合约功能，后加Filter序号、账号两个参数
                                disableDeploy         // 关闭该角色发布合约功能，后加Filter序号、账号两个参数
                                addPermission         // 给角色添加权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
                                delPermission         // 删除角色的权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
                                checkPermission       // 检查角色的权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
                                listPermission        // 列出账号/角色的权限，后加Filter序号、账号两个参数
```

Group功能

4.3、权限缺失的提示：

对于账号没有部署合约权限，将提示以下错误：

```
发送交易失败！ Error: NoDeployPermission .
```

无部署权限

对于账号没有调用合约权限，将提示以下错误：

```
发送交易失败！ Error: NoTxPermission .
```

无调用权限

或

```
Error: NoCallPermission.
```

无调用权限

4.4、AuthorityManager.js脚本具体使用例子，为排除干扰条件，执行前先确认config.js内的account为上帝账号。

```
// 添加Filter到系统合约Chain，后加Filter名称、版本、描述三个参数
babel-node AuthorityManager.js FilterChain addFilter NewFilter 2.0␣
↪FilterUsedForTest
// 从系统合约Chain删除Filter，后加Filter序号参数
babel-node AuthorityManager.js FilterChain delFilter 1
// 显示系统合约Chain的Filter
babel-node AuthorityManager.js FilterChain showFilter
// 还原系统合约Chain到初始状态
babel-node AuthorityManager.js FilterChain resetFilter

// 显示该Filter权限控制的状态，后加序号参数
babel-node AuthorityManager.js Filter getFilterStatus 1
// 启用该Filter权限控制，后加序号参数
babel-node AuthorityManager.js Filter enableFilter 1
// 关闭该Filter权限控制，后加序号参数
babel-node AuthorityManager.js Filter disableFilter 1
// 给账号添加新角色，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Filter setUsertoNewGroup 0␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 给账号添加已有角色，后加Filter序号、账号、已有Group角色三个参数
babel-node AuthorityManager.js Filter setUsertoExistingGroup 0␣
↪0x6ea2ae822657da5e2d970309b106207746b7b6b3 Group.address
// 说明：Group.address为Group地址。本js脚本不单独提供创建Group的功能，新Group的创建需要通
过setUsertoNewGroup和listUserGroup获取新建的Group.address
```

(continues on next page)

(continued from previous page)

```
// 显示账号的角色，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Filter listUserGroup 0␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9

// 显示该角色黑名单模式的状态，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group getBlackStatus 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 启用该角色黑名单模式，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group enableBlack 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 关闭该角色黑名单模式，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group disableBlack 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 显示该角色发布合约功能的状态，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group getDeployStatus 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 启用该角色发布合约功能，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group enableDeploy 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 关闭该角色发布合约功能，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group disableDeploy 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
// 给角色添加权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
babel-node AuthorityManager.js Group addPermission 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9 ContractA.address "set1(string)"
// 说明：ContractA.address为ContractA部署后的地址，注意后续权限操作基于合约地址，非合约名，所
以一合约重新部署后需重新添加权限
// 删除角色的权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
babel-node AuthorityManager.js Group delPermission 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9 ContractA.address "set1(string)"
// 检查角色的权限，后加Filter序号、账号、合约部署地址、函数名(参数类型列表)四个参数
babel-node AuthorityManager.js Group checkPermission 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9 ContractA.address "set1(string)"
// 列出账号/角色的权限，后加Filter序号、账号两个参数
babel-node AuthorityManager.js Group listPermission 1␣
↪0x4015bd4dd8767d568fc54cf6d0817ecc95d166d9
```

7.5.2 机构证书准入

一、功能介绍

联盟链中区块链上所有节点身份必须可追溯，能与现实实体一一对应，做到节点行为所属机构可追溯。 联盟链中区块链上所有节点颁发CA证书，为机构备案公钥证书到区块链，节点连接时验证颁发的CA机构证书有效性判断节点是否准入，对已连接的节点行为进行追溯。 联盟链管理员生成CA根证书并把CA根证书公钥提供给所有节点使用。CA根证书为每个节点颁发CA用户证书或同一机构下所有节点使用同一CA机构证书。

二、使用方式

机构证书的准入依赖系统合约，在进行机构证书准入操作前，再次请确认：

（1）系统合约已经被正确的部署。

（2）所有节点的config.json的systemproxyaddress字段已经配置了相应的系统代理合约地址。

（3）节点在配置了systemproxyaddress字段后，已经重启使得系统合约生效。

（4）/mydata/FISCO-BCOS/tools/web3lib/下的config.js已经正确的配置了节点的RPC端口。

2.1 配置节点证书

节点的证书存放目录在节点文件目录的data文件夹下。包括：

- ca.crt：区块链根证书公钥，所有节点共用。
- server.crt：单个节点的证书公钥，可公开。
- server.key：单个节点的证书私钥，应保密。

证书文件应严格按照上述命名方法命名。

FISCO BCOS通过授权某节点对应的公钥server.crt，控制此节点是否能够与其它节点正常通信。

注意：若要尝试使用AMOP（链上链下），请直接使用sample目录下的证书。AMOP暂不支持与新生成的证书进行连接。

（1）生成根证书ca.crt

执行命令，可生成根证书公私钥ca.crt、ca.key。ca.key应保密，并妥善保管。供生成节点证书使用。

```
cd /mydata/nodedata-1/data/
openssl genrsa -out ca.key 2048
openssl req -new -x509 -days 3650 -key ca.key -out ca.crt
```

（2）生成节点证书server.key、server.crt

生成节点证书时需要根证书的公私钥ca.crt、ca.key。执行命令，生成节点证书server.key、server.crt。

直接编写配置文件cert.cnf。

```
vim /mydata/nodedata-1/data/cert.cnf
```

内容如下，无需做任何修改。

```
拷贝以下内容在本地保存为cert.cnf文件
[ca]
default_ca=default_ca
[default_ca]
default_days = 365
default_md = sha256
[req]
distinguished_name = req_distinguished_name
req_extensions = v3_req
[req_distinguished_name]
countryName = CN
countryName_default = CN
stateOrProvinceName = State or Province Name (full name)
stateOrProvinceName_default =GuangDong
localityName = Locality Name (eg, city)
localityName_default = ShenZhen
organizationalUnitName = Organizational Unit Name (eg, section)
organizationalUnitName_default = webank
commonName =  Organizational  commonName (eg, webank)
commonName_default = webank
commonName_max = 64
[ v3_req ]
# Extensions to add to a certificate request
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
```

生成节点证书时需要根证书的公私钥ca.crt、ca.key。执行命令，用cert.cnf，生成节点证书server.key、server.crt。

```
cd /mydata/nodedata-1/data/
openssl genrsa -out server.key 2048
openssl req -new -key server.key -config cert.cnf -out server.csr
openssl x509 -req -CA ca.crt -CAkey ca.key -CAcreateserial -in server.csr -out␣
↪server.crt -extensions v3_req -extfile cert.cnf
```

2.2 开启所有节点的SSL验证功能

注意：1.3+版本已自动配置，若使用1.3+版本，可跳过本节。

在进行节点证书授权管理前，需开启区块链上每个节点的SSL验证功能。

此处以创世节点为例，其它节点也应采用相同的操作。

```
cd /mydata/nodedata-1/
vim config.json
```

将ssl字段置为1，效果如下。

```
"ssl":"1",
```

修改完成后重启节点，使其生效。

```
./stop.sh
./start.sh
```

其它节点也采用相同的操作，开启SSL验证功能。

2.3 配置机构证书信息

将节点的证书写入系统合约，为接下来的证书准入做准备。每张证书都应该写入系统合约中。节点的证书若不写入系统合约，相应的节点将不允许通信。

2.3.1 获取证书序列号

获取server.crt的序列号

```
cd /mydata/nodedata-2/data
openssl x509 -noout -in server.crt -serial
```

可得到证书序列号

```
serial=8A4B2CDE94348D22
```

2.3.2 编写证书准入状态文件

在tools/systemcontract目录下编写。

```
/mydata/FISCO-BCOS/tools/systemcontract
vim ca.json
```

将序列号填入hash字段。配置status，0表示不可用，1表示可用。其它字段默认即可。如下，让node2的证书可用。即status置1。

```
{
        "hash" : "8A4B2CDE94348D22",
        "status" : 1,
        "pubkey":"",
        "orgname":"",
        "notbefore":20170223,
        "notafter":20180223,
        "whitelist":"",
        "blacklist":""
}
```

证书准入状态文件其它字段说明如下。

字段	说明
hash	公钥证书序列号
pubkey	公钥证书，填空即可
orgname	机构名称，填空即可
notbefore	证书生效时间
notafter	证书过期时间
status	证书状态 0：不可用 1：可用
whitelist	ip白名单，填空即可
blacklist	ip黑名单，填空即可

2.3.3 将证书准入状态写入系统合约

执行以下命令，指定证书准入状态文件ca.json，将证书状态写入系统合约。

```
babel-node tool.js CAAction update ca.json
```

2.4 设置证书验证开关

证书验证开关能够控制是否采用证书准入机制。开启后，将根据系统合约里的证书状态（status）控制节点间是否能够通信。不在系统合约中的证书对应的节点，将不允许通信。

2.4.1 开启全局开关

执行命令，CAVerify设置为true

```
babel-node tool.js ConfigAction set CAVerify true
```

查看开关是否生效

```
babel-node tool.js ConfigAction get CAVerify
```

输出true，表示开关已打开

```
CAVerify=true,29
```

2.4.2 关闭全局开关

开关关闭后，节点间的通信不再验证证书。

执行命令，CAVerify设置为false

```
babel-node tool.js ConfigAction set CAVerify false
```

2.5 修改节点证书准入状态

已经写入系统合约的证书状态，允许修改（可用/不可用）

2.5.1 修改证书准入状态文件

修改相应证书对应的证书准入状态文件ca.json

```
/mydata/FISCO-BCOS/tools/systemcontract
vim ca.json
```

配置status，0表示不可用，1表示可用。其它字段默认即可。如下，让node2的证书不可用。即status置0。

```
{
        "hash" : "8A4B2CDE94348D22",
        "status" : 0,
        "pubkey":"",
        "orgname":"",
        "notbefore":20170223,
        "notafter":20180223,
        "whitelist":"",
        "blacklist":""
}
```

2.5.2 更新证书准入状态

执行以下命令，指定证书准入状态文件ca.json，更新证书准入状态。

```
babel-node tool.js CAAction updateStatus ca.json
```

查看证书状态

```
babel-node tool.js CAAction all
```

可看到证书状态

```
----------CA 0---------
hash=8A4B2CDE94348D22
pubkey=
orgname=
notbefore=20170223
notafter=20180223
status=0
blocknumber=36
whitelist=
blacklist=
```

三、证书注意事项:

妥善保管 server.key，切勿泄露，同时应把证书，server.crt 和 server.key 备份到安全位置

7.5.3 群签名&&环签名

目录

1 基本介绍

1.1 场景

FISCO-BCOS实现了群签名和环签名链上验证功能，主要包括如下使用场景：

（1）群签名场景

- **场景1**： 机构内成员（C端用户）或机构内下属机构通过机构将群签名信息上链，其他人在链上验证签名时，仅可获知签名所属的群组，却无法获取签名者身份，保证成员的匿名性和签名的不可篡改性；（如拍卖、匿名存证、征信等场景）
- **场景2**：B端用户将生成的群签名通过AMOP发送给上链结构（如webank），上链机构将收集到的群签名信息统一上链（如竞标、对账等场景），其他人验证签名时，无法获取签名者身份，保证成员的匿名性，监管可通过可信第三方追踪签名者信息，保证签名的可追踪性。

（2）环签名场景

- **场景1（匿名投票）**：机构内成员（C端用户）对投票信息进行环签名，并通过可信机构（如webank）将签名信息和投票结果写到链上，其他人可在链上验证签名时，仅可获取发布投票到链上的机构，却无法获取投票者身份信息
- **其他场景（如匿名存证、征信）**：场景与群签名匿名存证、征信场景类似，唯一的区别是任何人都无法追踪签名者身份
- **匿名交易**：在UTXO模型下，可将环签名算法应用于匿名交易，任何人都无法追踪转账交易双方；

返回目录

1.2 相关代码

FISCO BCOS提供了群签名&&环签名链上验证功能，下表详细介绍了相关代码模块：

| 模块 | 代码 | 说明 | | ——————— | ——————————- | ——————————- | | 依赖软件安装模块 | scripts/install_pbc.shdeploy_pbc.sh | 群签名库的实现依赖于pbc和pbc-sig库，可调用deploy_pbc.sh安装pbc和pbc-sig | | 群签名&&环签名库源码 | deps/src/group_sig_lib.tgz | 群签名&&环签名库源码压缩包 | | 编译模块 | cmake/FindPBC.cmakecmake/ProjectGroupSig.cmake | 在FISCO BCOS中编译群签名&&环签名库源码相关的cmake文件 | | 群签名&&环签名验证实现模块 | libevm/ethcall/EthcallGroupSig.hlibevm/ethcall/EthcallRingSig.h | 使用ethcall，调用群签名&&环签名库，实现群签名&&环签名链上验证功能 |

为了给用户提供最大的使用灵活性，FISCO BCOS支持群签名&&环签名ethcall开关功能（默认关闭群签名&&环签名ethcall）：

- **关闭群签名&&环签名ethcall验证功能**：群签名&&环签名库以及群签名&&环签名验证实现模块均没被编译到FISCO BCOS中，编译速度较快，但FISCO BCOS不支持群签名&&环签名链上验证功能；
- **开启群签名&&环签名ethcall验证功能**：群签名&&环签名库以及相关代码被编译到FISCO BCOS中，编译速度稍慢，但支持群签名&&环签名链上验证功能。

  下节详细描述了开启和关闭群签名&&环签名ethcall的部署方法。

返回目录

2 部署

开启和关闭群签名&&环签名ethcall之前，必须保证您的机器可正确部署FISCO BCOS，FISCO BCOS详细部署方法可参考FISCO BCOS区块链操作手册.

2.1 开启群签名&&环签名ethcall

(1) 安装依赖软件

① 安装基础依赖软件

部署FISCO BCOS之前需要安装git, dos2unix和lsof依赖软件：

- git: 用于拉取最新代码
- dos2unix && lsof: 用于处理windows文件上传到linux服务器时，可执行文件无法被linux正确解析的问题；

针对centos和ubuntu两种不同的操作系统，可分别用以下命令安装这些依赖软件：

(若依赖软件安装失败，请检查yum源或者ubuntu源是否配置正确)

```
[centos]
sudo yum -y install git
sudo yum -y install dos2unix
sudo yum -y install lsof

[ubuntu]
sudo apt install git
sudo apt install lsof
sudo apt install tofrodos
ln -s /usr/bin/todos /usr/bin/unxi2dos
ln -s /usr/bin/fromdos /usr/bin/dos2unix
```

② 安装群签名依赖软件pbc和pbc-sig

群签名依赖pbc和pbc-sig，因此打开群签名和环签名ethcall开关前，必须先安装pbc和pbc-sig，为了方便用户操作，FISCO BCOS提供了pbc和pbc-sig的部署脚本（支持在centos和ubuntu系统安装pbc和pbc-sig）：

```
#用dos2unix格式化可执行脚本，防止windows文件上传到unix，无法被正确解析
bash format.sh
#用deploy_pbc.sh脚本安装pbc和pbc-sig
sudo bash deploy_pbc.sh
```

(2) 打开群签名&&环签名ethcall编译开关

FISCO BCOS群签名和环签名相关的编译开关是在使用cmake生成Makefile文件时通过-DGROUPSIG=ON或者-DGROUPSIG=OFF设置，默认是关闭该开关的：

```
#新建build文件用于存储编译文件
cd FISCO-BCOS && mkdir -p build && cd build
#cmake生成Makefile时打开编译开关
```

(continues on next page)

(continued from previous page)

```
#**centos系统:
cmake3 -DGROUPSIG=ON -DEVMJIT=OFF -DTESTS=OFF -DMINIUPNPC=OFF ..
#**ubuntu系统:
cmake -DGROUPSIG=ON -DEVMJIT=OFF -DTESTS=OFF -DMINIUPNPC=OFF ..
```

(3) 编译并运行fisco bcos

```
#编译fisco-bcos
make -j4 #注: 这里j4表明用4线程并发编译，可根据机器CPU配置调整并发线程数
#运行fisco-bcos: 替换节点启动的可执行文件，重启节点:
bash start.sh
```

返回目录

2.2 关闭群签名&&环签名ethcall

(1) 关闭群签名&&环签名ethcall编译开关

编译fisco bcos时，将cmake的-DGROUPSIG编译选项设置为OFF可关闭群签名&&环签名ethcall功能:

```
#新建build文件用于存储编译文件
cd FISCO-BCOS && mkdir -p build && rm -rf build/* && cd build
#关闭群签名&&环签名ethcall开关
#centos系统:
cmake3 -DGROUPSIG=OFF -DEVMJIT=OFF -DTESTS=OFF -DMINIUPNPC=OFF ..
#ubuntu系统:
cmake -DGROUPSIG=OFF -DEVMJIT=OFF -DTESTS=OFF -DMINIUPNPC=OFF ..
```

(2) 编译并运行fisco bcos

```
#编译fisco-bcos
make -j4 #注: 这里j4表明用4线程并发编译，可根据机器CPU配置调整并发线程数
#运行fisco-bcos: 替换节点启动的可执行文件，重启节点:
bash start.sh
```

返回目录

3 使用

在区块链上使用群签名&&环签名功能，还需要部署以下服务:

(1) 群签名&&环签名客户端: sig-service-client

sig-service-client客户端提供了以下功能:

- 访问群签名&&环签名服务rpc接口;
- 将群签名&&环签名信息写入链上（发交易）

具体使用和部署方法请参考群签名&&环签名客户端操作手册.

(2) 群签名&&环签名服务: sig-service

sig-service签名服务部署在机构内，为群签名&&环签名客户端(sig-service-client)提供了如下功能:

- 群生成、群成员加入、生成群签名、群签名验证、追踪签名者身份等rpc接口;
- 环生成、生成环签名、环签名验证等rpc接口;

在FISCO BCOS中，sig-service-client客户端一般先请求该签名服务生成群签名或环签名，然后将获取的签名信息写入到区块链节点；区块链节点调用群签名&&环签名ethcall验证签名的有效性。

sig-service的使用和部署方法请参考群签名&&环签名RPC服务操作手册.

返回目录

4 注意事项

(1) 群签名&& 环签名ethcall兼容老版本FISCO BCOS

(2) 启用群签名&&环签名ethcall，并调用相关功能后，**不能关闭该ethcall接口，否则验证区块时，群签名和环签名相关的数据无法找到验证接口，从而导致链异常退出**

若操作者在开启并使用群签名&&环签名特性后，不小心关闭了该ethcall功能，可通过回滚fisco bcos到开启群签名&&环签名ethcall时的版本来使链恢复正常；

(3) 同一条链的fisco bcos版本必须相同

即：若某个节点开启了群签名&&环签名验证功能，其他链必须也开启该功能，否则在同步链时，无群签名&&环签名ethcall实现的节点会异常退出；

(4) 使用群签名&&环签名链上验证功能前，**必须先部署群签名&&环签名服务和群签名&&环签名客户端**

客户端sig-service-client向链上发签名信息，群签名&&环签名服务为客户端提供签名生成服务

返回目录

7.5.4 可监管的零知识证明说明

FISCO-BCOS提供了一种可被监管的零知识证明方法，为用户提供一种保护隐私的、可被验证的秘密交易框架的同时，为监管者提供监管的接口，实现对区块链上每一笔秘密交易的监管。

1. 概念说明

零知识证明，是示证者在不暴露自身秘密信息的情况下，通过某种方式，让验证着相信其拥有此秘密信息。

区块链上的零知识证明，是用户对自身需要保密但却需要被验证的数据进行转化，转化成proof提供给区块链节点。区块链节点在不知晓用户秘密数据的情况下，验证此秘密数据的正确性。任何人都不能通过proof推测出用户的秘密数据，在区块链上实现了一种可被区块链节点验证的秘密操作，但存在监管风险。

FISCO-BCOS的可监管零知识证明，是用户在FISCO-BCOS区块链上进行的任何一次零知识证明操作，都可以被监管者解密，受到监管者的监管。FISCO-BCOS区块链节点是零知识证明操作的验证着。

2. 底层库

libzkg：可监管的零知识证明库

3. 实现场景

（1）一对一匿名可监管转账

用户可在FISCO-BCOS上进行匿名转账，在不暴露身份和交易金额的情况下，实现可被区块链验证的金额划转。同时，监管者可解密所有匿名转账信息。相关工程的说明与部署方法：zkg-tx1to1

（2）开发中...

CHAPTER 8

应用实践

8.1 落地应用

8.1.1 微粒贷机构间对账平台

作者：fisco-dev

背景概述

2016年8月，微众银行联合合作行，基于BCOS早期版本推出了微粒贷机构间对账平台，这也是国内首个在生产环境中运行的银行业联盟链应用场景。

目前有多家合作行相继接入微粒贷机构间对账平台，通过区块链与分布式账本技术，优化微粒贷业务中的机构间对账流程，实现了准实时对账、提高运营效率、降低运营成本等目标。截止目前，平台稳定运行1年多，保持0故障，记录的真实交易笔数已达千万量级。

批量文件对账

微粒贷是微众银行面向微信用户和手机QQ用户推出的首款互联网小额信贷产品，为超过千万的用户提供便捷、高效的贷款服务。微粒贷与其他银行存在较为普遍的同业合作，在该合作模式下，合作银行之间的资金清算显得极为重要。

金融业务合作不同于一般合作，需要频繁地进行数据交换及对账等繁杂工作，因此“对账”是金融机构之间最普遍的需求之一，对账目的时效性和准确度要求尤为苛刻。传统的对账方式是“批量文件对账”：即机构之间会约定好某一个时间点对前一个交易日的所有数据进行汇总，按照约定格式输出成文件，并以某一种技术手段交付给其他机构进行对账。

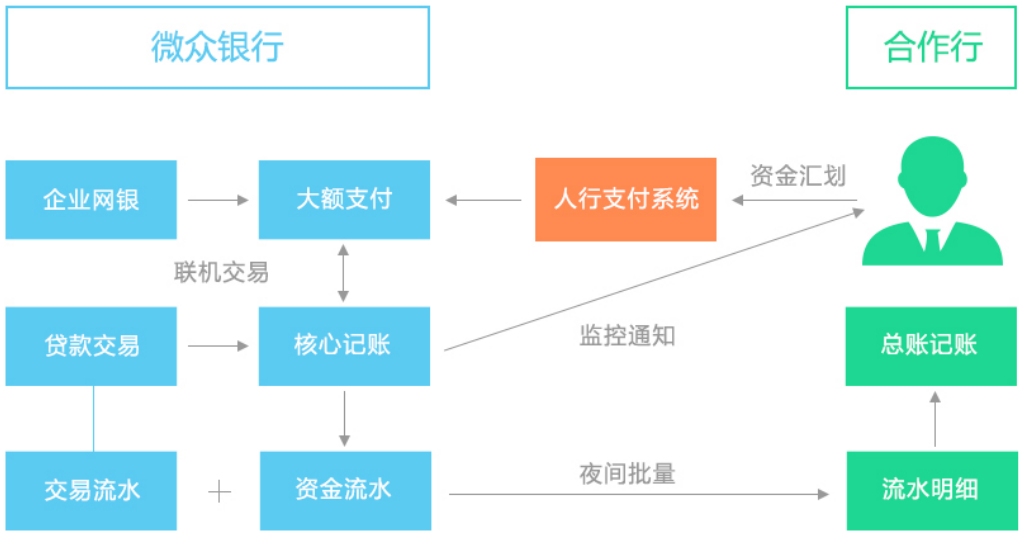


图1:

批量文件对账方案

在这种“批量文件对账”方式下，存在着一些痛点，如：

- 合作行无法实时了解到引发账户变动的贷款借还交易明细信息。
- 合作行无法及时了解到账务是否不平。
- 合作行需要自己开发对账系统。
- 缺乏统一全面的信息视图。

区块链对账方案

区块链技术是一种不可篡改的分布式账本技术，区块链技术最大的特征是“分布式账本”，即链上的各个参与机构共同拥有一个账本。区块链上所有的交易信息都会被记录，并且无法篡改，可确保数据的真实透明可追溯，非常适用于金融行业的交易数据同步和对账等场景。传统“批量文件对账”模式长久以来未能解决的问题，正是区块链技术的用武之地。微众银行基于此特征，设计了区块链对账方案，利用区块链技术将交易信息旁路上链，解决微粒贷业务系统与合作行的对账问题，降低了合作行的人力和时间成本，提升了对账的时效性与准确度。

设计原则：

- 不影响现有业务，通过旁路上链的方式，将业务数据脱敏后发送到区块链上。
- 开发一个web系统，方便合作行查询区块链上的对账结果。
- 业务数据传输、存储均采用加密方式，确保数据安全性。

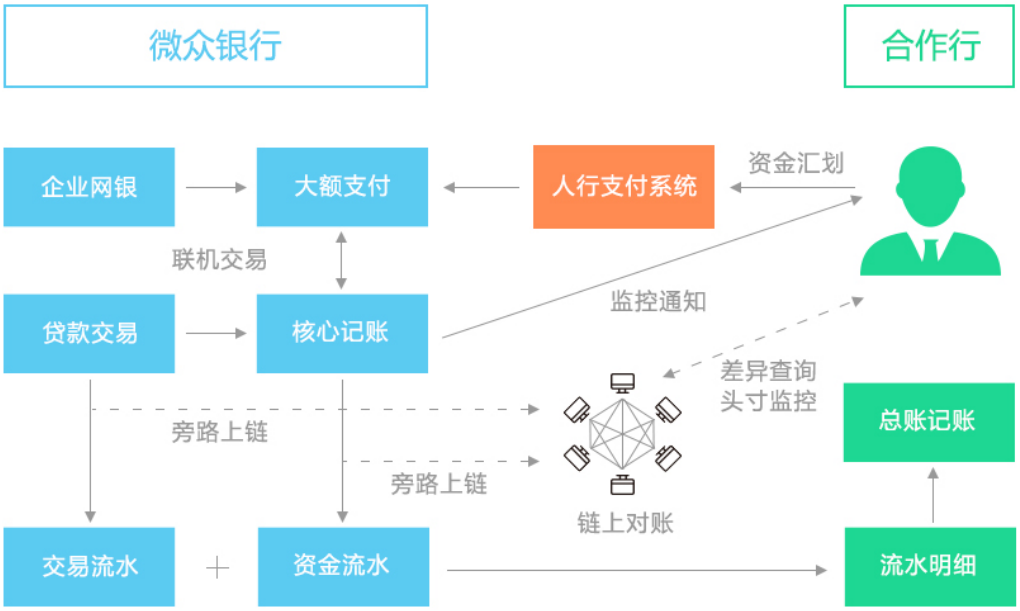


图2:

区块链对账方案

具体而言，基于区块链技术实现的微粒贷机构间对账平台具有以下优势：

• 使用简单：通过内网访问web系统，输入管理员账号密码即可登录。

• 数据实时触达：实时监测当日账户余额、当日放款总金额、当日还款总金额、当日其它划入款项总金额、当日其它划出款项总金额和当日流水数据。

• 数据安全性高：数据的通信和存储都经过加密处理。

• 数据可用性高：区块链节点之间相互同步数据，提升数据的可用性。

• 合作行可控性强：合作行可以自由选择自己的节点数为1到多个，节点可以选择部署在合作行内或公有云上，不同合作行之间的数据是物理隔离，保护隐私。

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

8.1.2 仲裁链：基于区块链的存证实践

作者：fisco-dev

背景概述

2017年10月，微众银行联合广州仲裁委（即下文“仲裁机构”）、杭州亦笔科技（即下文“存证机构”）三方基于区块链技术搭建了“仲裁链”。“仲裁链”基于区块链多中心化、防篡改、可信任特征，利用分布式数据存储、加密算法等技术对交易数据共识签名后上链，实时保全的数据通过智能合约形成证据链，满足证据真实性、合法性、关联性的要求，实现证据及审判的标准化。

2018年2月，广州仲裁委基于“仲裁链”出具了业内首个裁决书。

截止目前，“仲裁链”已经稳定运行5个多月。

平台价值

“仲裁链”接入仲裁机构，让仲裁机构参与到存证业务过程中来，一起共识、实时见证，为仲裁提供了真实透明可追溯的业务数据源。这对于仲裁机构来说，保证了链上交易信息不被篡改，有助于仲裁机构快速完成证据的核实、解决纠纷，降低仲裁过程中人力物力和时间成本，提升司法效率，降低仲裁成本；对于金融机构来说，”仲裁链”将快速有效解决纠纷，提高运营和风控效率；对于用户来说，”仲裁链”在保障合法权益的同时，能够有效保护隐私，降低解决纠纷的时间和经济成本。

平台功能

线上仲裁存证主要包括存证、取证，核证三部分：

在发起存证前，需部署存证初始条件智能合约，约定存证生效所需条件（在这里就是微众，仲裁机构，存证机构的签名）。

1）存证：业务数据经微众存证系统签名后发起上链，上链成功后通知存证机构及仲裁机构对数据签名确认。存证机构及仲裁机构收到通知后，取出链上数据进行核实后完成签名，至此整个存证流程完成。在整个存证流程中，由智能合约保证，任何方都不能更改已存证的数据，只能追加存证数据。同时，存证系统准实时异步上链，对正常的业务逻辑无影响。

2） 取证：当有取证需求时，微众银行从”仲裁链”选择数据后，提交仲裁机构。仲裁机构对微众银行提供的数据进行解析，获取到区块链相关地址信息。通过地址调取链上数据。

3） 核证：区块链核证指的是仲裁机构在取证后，调用SDK相关接口，判断存证是否满足存证生效的初始条件。

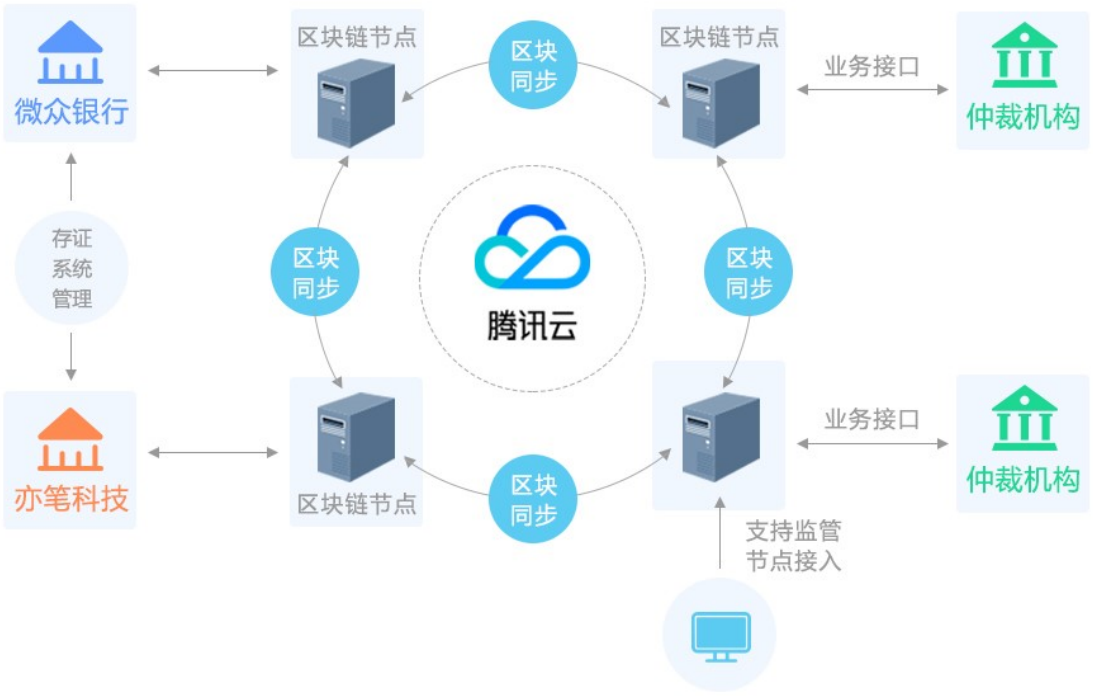


图1：”仲裁链”业务架构图

存证样例

为帮助开发者在存证场景中快速启动存证应用开发，FISCO BCOS还提供了完整的存证样例供开发者学习和使用，包括完整的业务sdk代码和详细的说明文档。适用于所有需要进行存证、核证、取证的业务

场景，尤其是需要解决多方信任问题或获取司法监管许可的情况。

FISCOBCOS存证样例演示的业务流程如下：

1. 存证参与各方需要事先约定存证生效所需条件，然后由一方调用SDK API新建、部署工厂合约，大家共识通过生效。
2. 存证业务方通过SDK API调用工厂合约新建附加了自己签名的证据合约。（需要传工厂合约的地址文件。）
3. 存证机构调用SDK API对证据进行签名确认。
4. 仲裁机构调用SDK API对证据进行签名确认。
5. 仲裁机构通过SDK API调取链上证据信息，并验证证据已有的签名正确性和完整性。

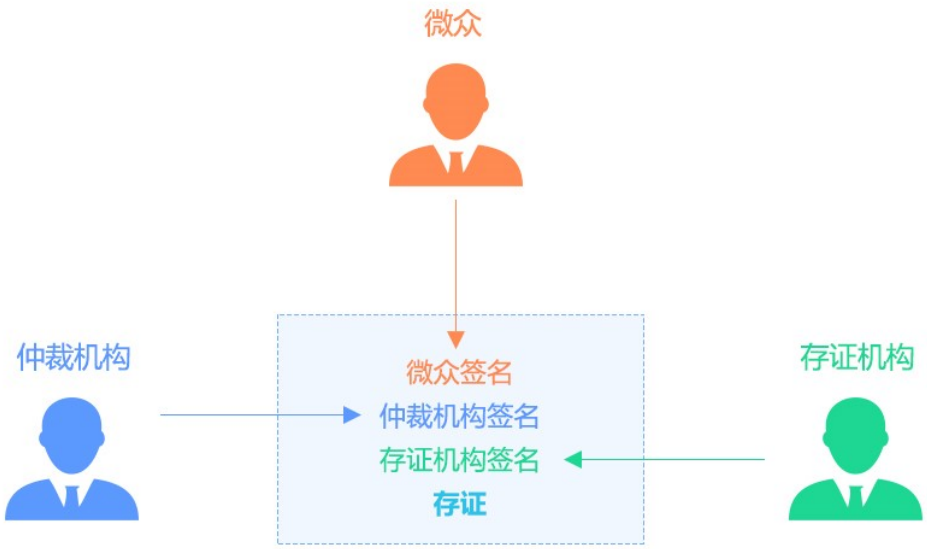


图2：存证样例流程示意图

存证样例为开发者提供了大量的默认配置，大大降低了用户自主配置的成本。使用一键脚本，只需配置节点ip和端口，就可以直接运行整个存证流程。样例同时配备了详细的文档说明，给用户提供stepby step的使用指导，协助用户直观快速地理解系统。在此基础上，带来多种体验方式，既可以整体一键式快速体验整个存证流程，也可单步详细分析每个步骤。

附：

存证样例地址：

https://github.com/FISCO-BCOS/evidenceSample

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

8.1.3 海量存证数据 + 区块链实现

作者：yibi-dev

- 海量存证数据+区块链实现
  - 区块链存证目前存在的挑战

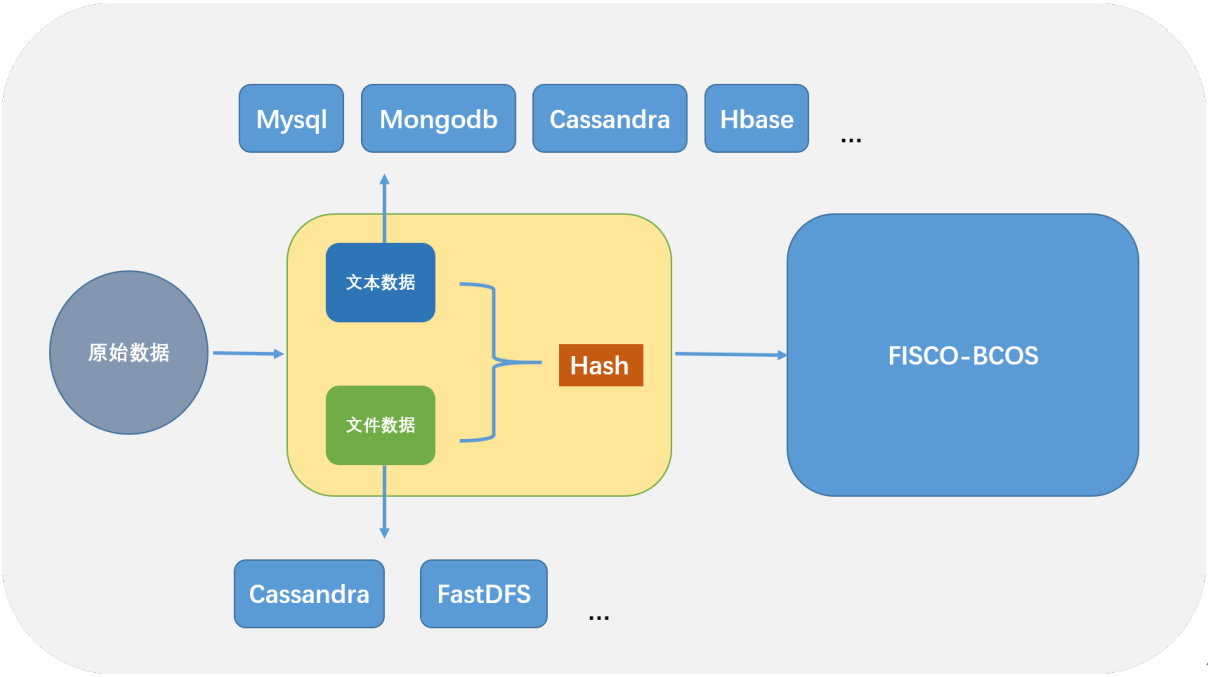
- 优化实现方案
- 亦笔科技的存证数据存储实现
  - 为什么使用*Phoenix+Hbase*
  - *Hbase*数据模型
  - *Hbase*分片基础的*Region*
  - *Hbase*在业务实现上存在的不足
  - *Phoenix*架构
  - *Phoenix*性能对比
  - *Cassandra*如何快速、可靠的存储小文件
  - *Cassandra*数据的写入
  - *Cassandra*数据的读取
- 总结

区块链存证目前存在的挑战

1. 鉴于目前区块链的发展现状，大数据量的文本、文件信息直接存储在区块链系统之中会导致系统的处理性能明显下降，并且每个节点都需要有足够大的存储空间支持，无法满足目前金融行业对系统性能的较高要求。
2. 随着存证系统日益庞大的数据量导致系统的存储与查询性能急剧下降，对系统的高可用、水平扩容也有着较高的要求。

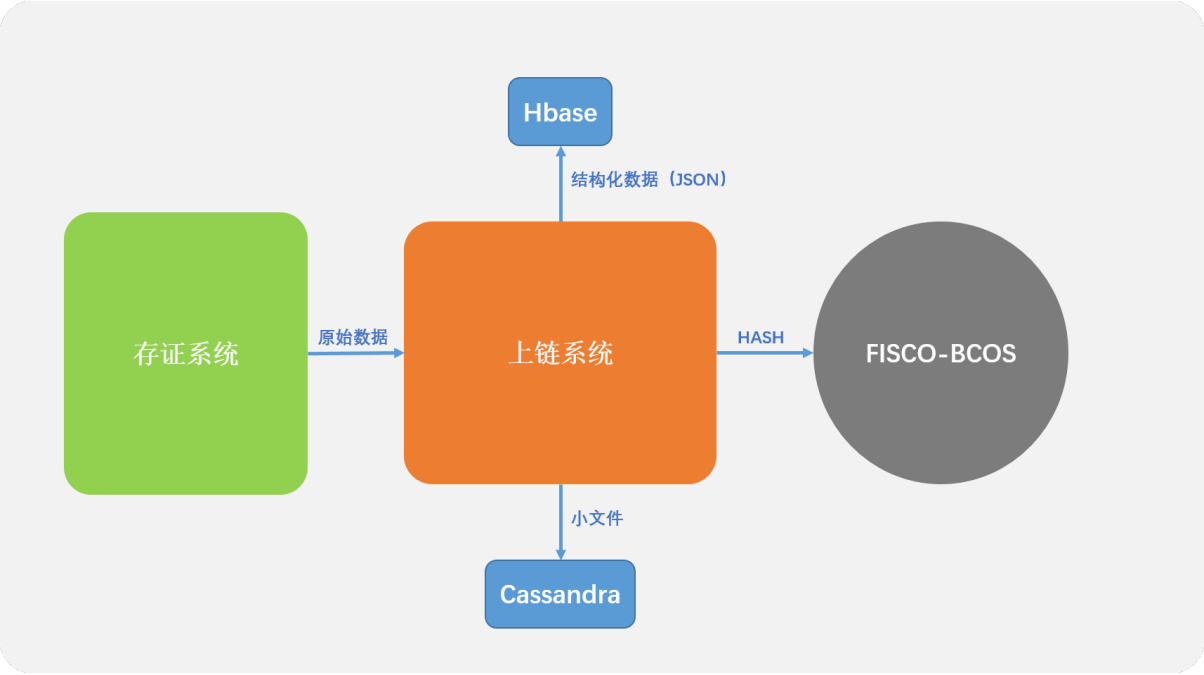
优化实现方案

在hash上链，联盟节点参与背书的基本条件不变的情况下，对原始数据进行hash计算方式及存储进行适配处理。支持document（json）、文件存储和加密的可插拔实现。



优化架构示意

亦笔科技的存证数据存储实现



亦笔存证架构示意

- Hbase对于海量文档数据存储有较好的支持，目前hadoop技术体系也相对成熟稳定。使用Phoenix+Hbase支持海量数据的存储、随机查询。
- Cassandra也是一种点对点的分布式nosql数据库，对于小文档数据存储有良好的表现。支持海量文件数据的读写，节点数据自动同步。

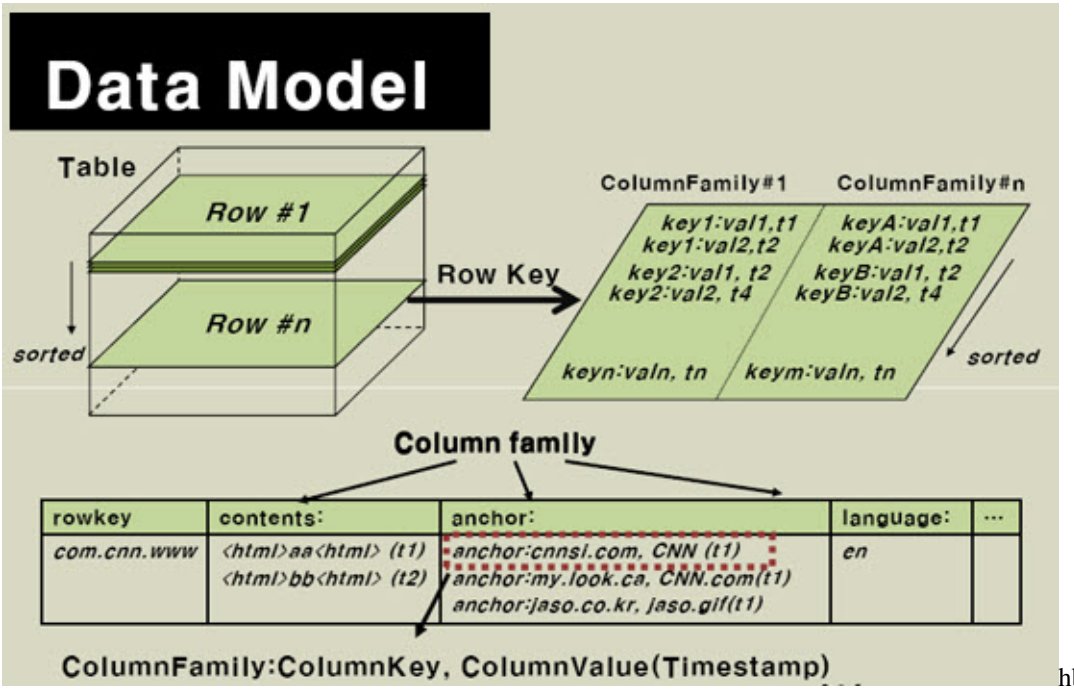
为什么使用Phoenix + Hbase

在存证场景中，许多合作对象所需要存证的数据很难保持高度一致。NoSQL数据库更适用于高扩展性的数据存储需求。海量存储、高并发同时也是很适合存证的业务诉求。而稀疏的特性也大大节约了存储空间。

hbase特性

Hbase数据模型

HBase 以表的形式存储数据。表由行和列组成。列划分为若干个列族（row family），如下图所示。



hbase数据模型

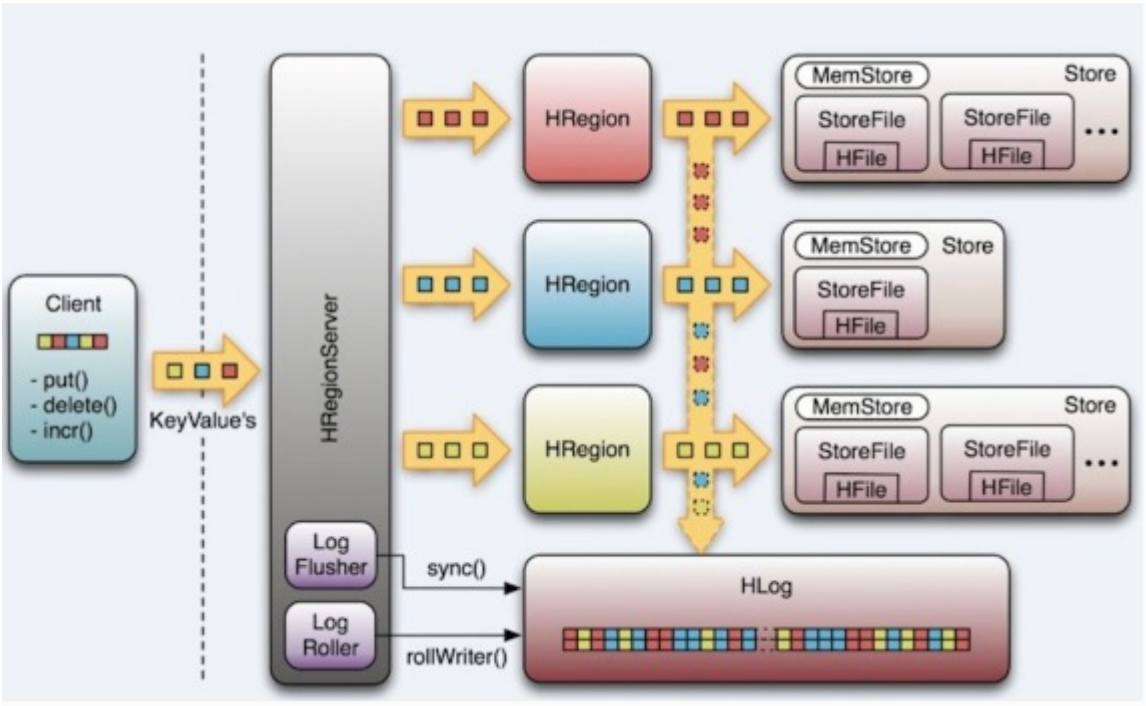
Rowkey的概念和mysql中的主键是完全一样的，Hbase使用Rowkey来唯一的区分某一行的数据。每个Region负责一小部分Rowkey范围的数据的读写和维护，Region包含了对应的起始行到结束行的所有信息。master将对应的region分配给不同的RergionServer，由RegionSever来提供Region的读写服务和相关的管理工作。

Hbase只支持3中查询方式：

1. 基于Rowkey的单行查询；
2. 基于Rowkey的范围扫描
3. 全表扫描

Hbase分片基础的Region

Region的概念和关系型数据库的分区或者分片差不多。 Hbase会将一个大表的数据基于Rowkey的不同范围分配到不通的Region中，每个Region负责一定范围的数据访问和存储。这样即使是一张巨大的表，由于被切割到不通的region，访问起来的时延也很低。



hbase数据模型

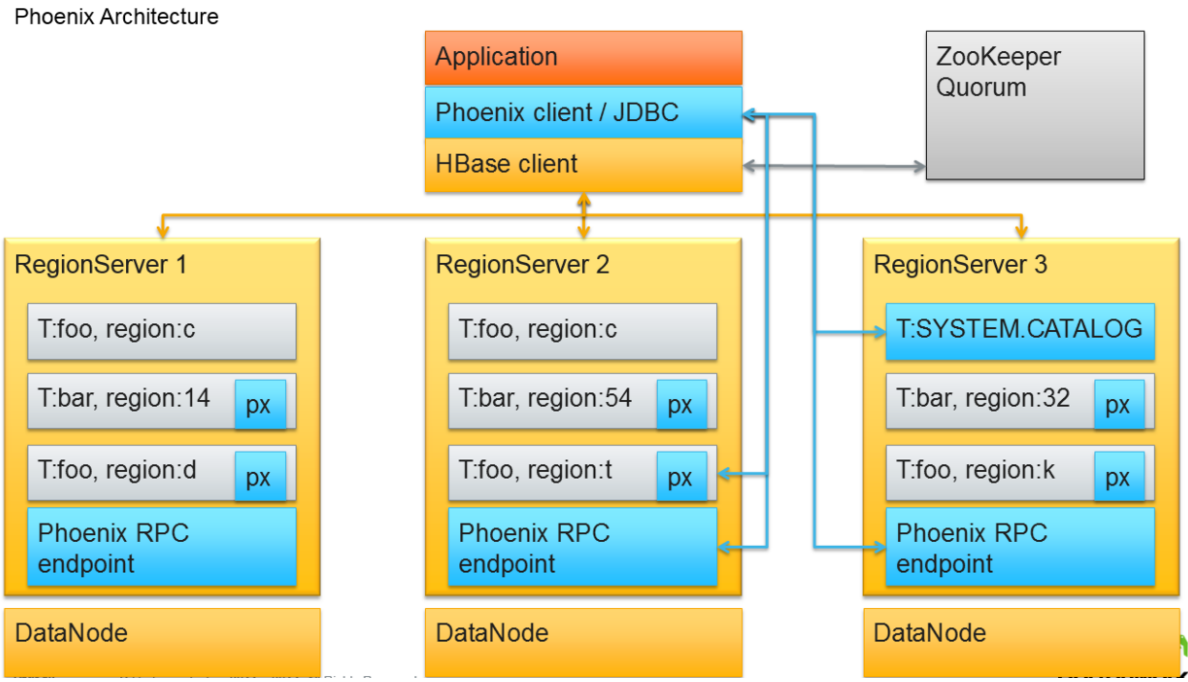
Hbase在业务实现上存在的不足

Hbase帮我们解决了结构化数据的高并发写入和读取，但是无法满足我们对于随机查询的需求。在HBase中，只有一个单一的按照字典序排序的rowKey索引，当使用rowKey来进行数据查询的时候速度较快，但是如果不使用rowKey来查询的话就会使用filter来对全表进行扫描，很大程度上降低了检索性能。而Phoenix提供了二级索引技术来应对这种使用rowKey之外的条件进行检索的场景。

Phoenix架构

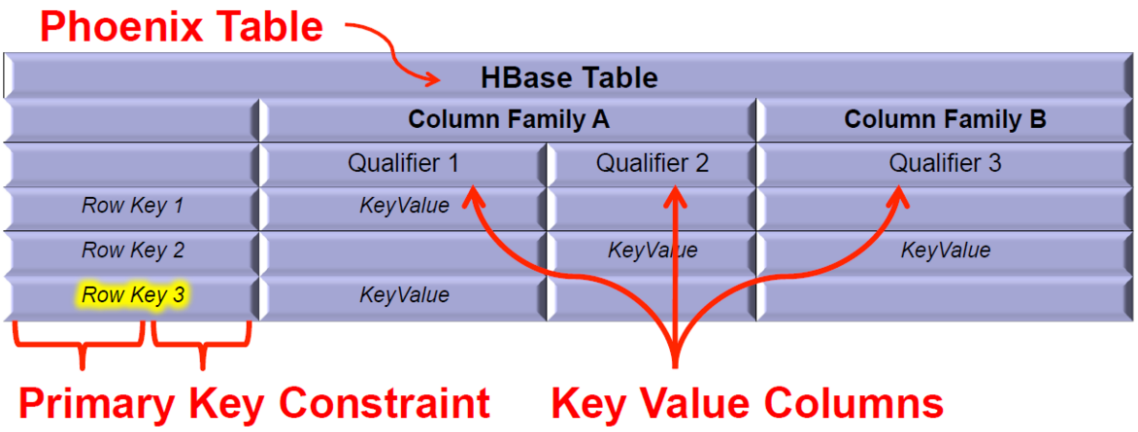
主要特性：

1. 二级索引
2. 分页查询
3. 散列表
4. 视图
5. 动态列
6. SQL支持



Phoenix架构示意

Phoenix将HBase的数据模型映成关系型数据表示意:

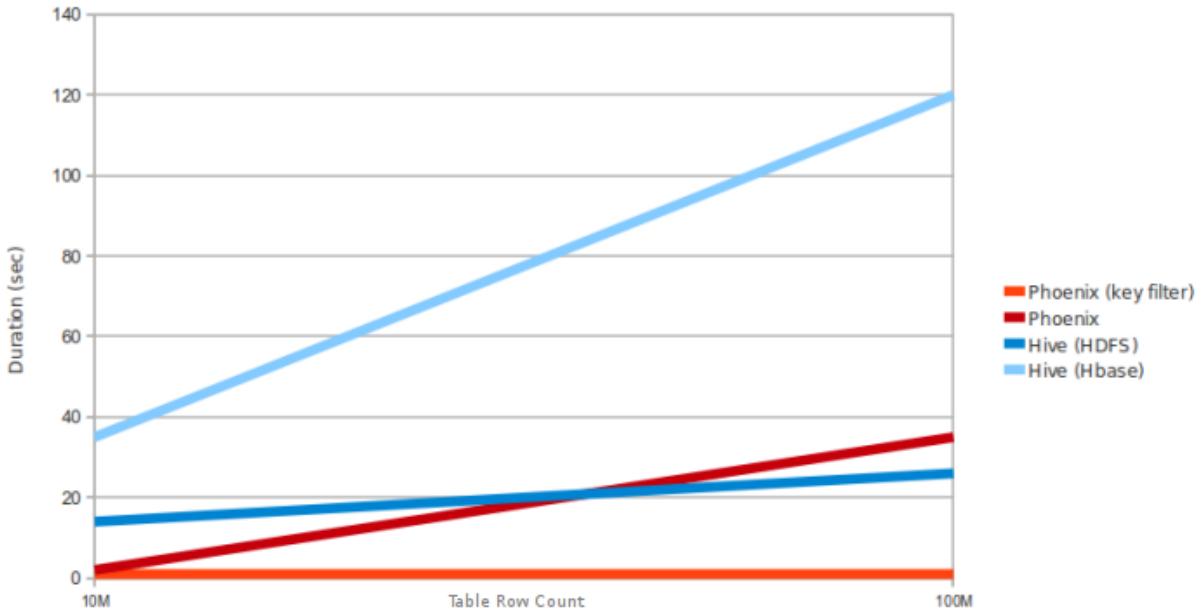


Phoenix表结构

Phoenix性能对比

Phoenix通过以下方法来奉行把计算带到离数据近的地方的哲学:

- 协处理器 在服务端执行操作来最小化服务端和客户端的数据传输
- 定制的过滤器 为了删减数据使之尽可能地靠近源数据并最小化启动代价，Phoenix使用原生的HBase APIs而不是使用Map/Reduce框架



Phoenix性能对比

Cassandra如何快速、可靠的存储小文件

Cassandra的存储机制借鉴了Bigtable的设计，采用Memtable和SSTable的方式。

Cassandra 中的数据主要分为三种：

1. CommitLog：主要记录下客户端提交过来的数据以及操作。这个数据将被持久化到磁盘中，以便数据没有被持久化到磁盘时可以用来恢复。
2. Memtable：用户写的数据在内存中的形式，它的对象结构在后面详细介绍。其实还有另外一种形式是 BinaryMemtable 这个格式目前 Cassandra 并没有使用，这里不再介绍了。
3. SSTable：数据被持久化到磁盘，这又分为 Data、Index 和 Filter 三种数据格式。

CommitLog 数据格式

CommitLog 的数据只有一种，那就是按照一定格式组成 byte 组数，写到 IO 缓冲区中定时的被刷到磁盘中持久化，在上一篇的配置文件详解中已经有说到 CommitLog 的持久化方式有两种，一个是 Periodic 一个是 Batch，它们的数据格式都是一样的，只是前者是异步的，后者是同步的，数据被刷到磁盘的频繁度不一样。它持久化的策略也很简单，就是首先将用户提交的数据所在的对象 RowMutation 序列化成 byte 数组，然后把这个对象和 byte 数组传给 LogRecordAdder 对象，由 LogRecordAdder 对象调用 CommitLogSegment 的 write 方法去完成写操作。

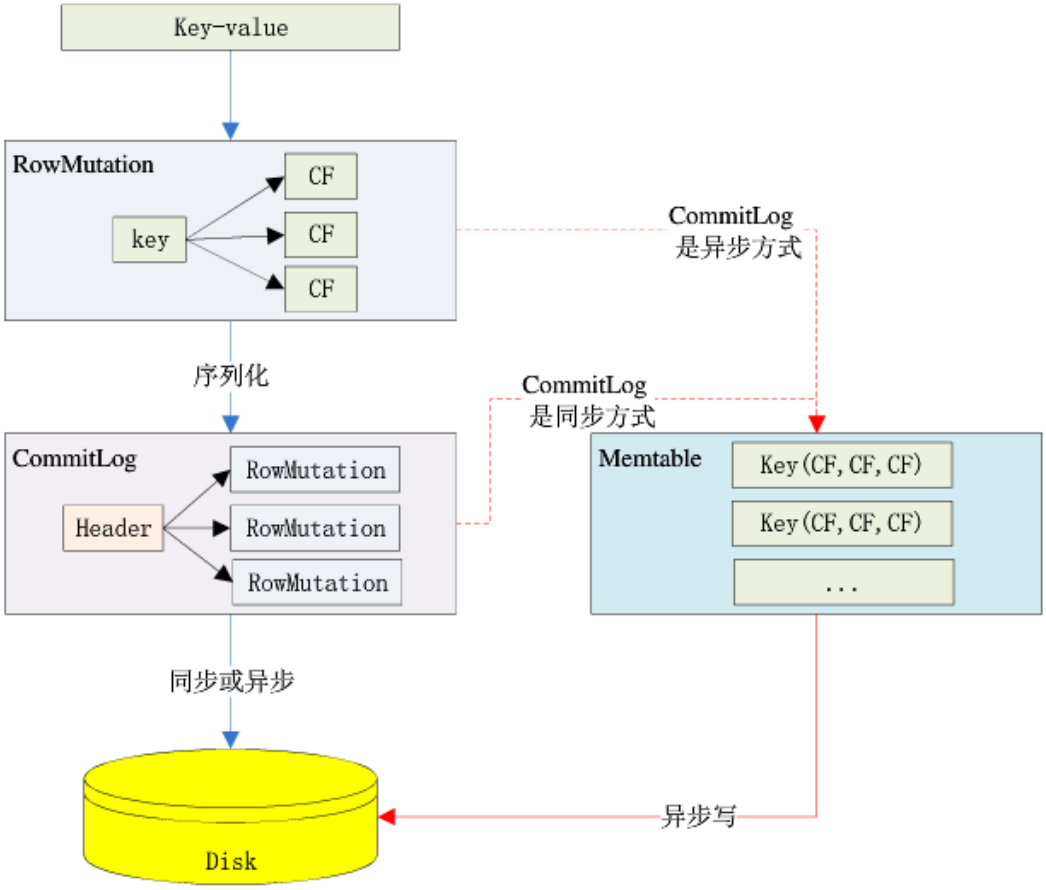
CommitLog文件数组结构

CommitLog文件数组结构

Memtable 内存中数据结构

Memtable 内存中数据结构比较简单，一个 ColumnFamily 对应一个唯一的 Memtable 对象，所以 Memtable 主要就是维护一个 ConcurrentSkipListMap<DecoratedKey, ColumnFamily> 类型的数据结构，当一个新的 RowMutation 对象加进来时，Memtable 只要看看这个结构是否 <DecoratedKey, ColumnFamily> 集合已经存在，没有的话就加进来，有的话取出这个 Key 对应的 ColumnFamily，再把它们的 Column 合

并。 Cassandra 的写的性能很好，好的原因就是因为 Cassandra 写到数据首先被写到 Memtable 中，而 Memtable 是内存中的数据结构，所以 Cassandra 的写是写内存的，下图基本上描述了一个 key/value 数据是怎么样写到 Cassandra 中的 Memtable 数据结构中的。



memtable写入

SSTable 数据格式

每添加一条数据到 Memtable 中，程序都会检查一下这个 Memtable 是否已经满足被写到磁盘的条件，如果条件满足这个 Memtable 就会写到磁盘中。 Memtable 的条件满足后，它会创建一个 SSTableWriter 对象，然后取出 Memtable 中所有的 <DecoratedKey, ColumnFamily> 集合，将 ColumnFamily 对象的序列化结构写到 DataOutputBuffer 中。接下去 SSTableWriter 根据 DecoratedKey 和 DataOutputBuffer 分别写到 Date、Index 和 Filter 三个文件中。

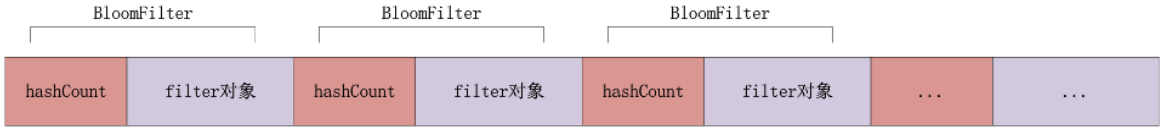
SSTable 的 Data 文件结构

SSTable的Data文件结构

Index 文件结构

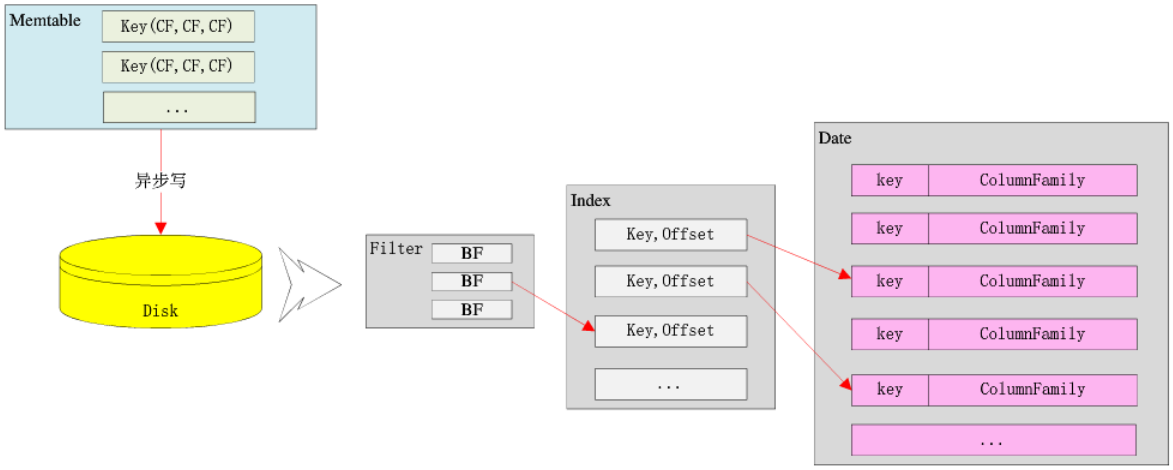
Phoenix性能对比

Filter 文件结构



Phoenix性能对比

三个文件对应的数据格式可以用下图来清楚的表示：



SSTable文件格式

这个三个文件写完后，还要做的一件事件就是更新前面提到的 CommitLog 文件，告诉 CommitLog 的 header 所存的当前 ColumnFamily 的没有写到磁盘的最小位置。

Cassandra数据的写入

CassandraServer 接收到要写入的数据时，首先创建一个 RowMutation 对象，再创建一个 QueryPath 对象，这个对象中保存了 ColumnFamily、Column Name 或者 Super Column Name。接着把用户提交的所有数据保存在 RowMutation 对象的 Map<String, ColumnFamily> 结构中。接下去就是根据提交的 Key 计算集群中那个节点应该保存这条数据。这个计算的规则是：将 Key 转化成 Token，然后在整个集群的 Token 环中根据二分查找算法找到与给定的 Token 最接近的一个节点。如果用户指定了数据要保存多个备份，那么将会顺序在 Token 环中返回与备份数相等的节点。这是一个基本的节点列表，后面 Cassandra 会判断这些节点是否正常工作，如果不正常寻找替换节点。还有还要检查是否有节点正在启动，这种节点也是要在考虑的范围内，最终会形成一个目标节点列表。最 后把数据发送到这些节点。 接下去就是将数据保存到 Memtable 中和 CommitLog 中，关于结果的返回根据用户指定的安全等级不同，可以是异步的，也可以是同步的。如果某个节点返回失败，将会再次发送数据。

Cassandra数据的读取

Cassandra 的写的性能要好于读的性能，为何写的性能要比读好很多呢？原因是，Cassandra 的设计原则就是充分让写的速度更快、更方便而牺牲了读的性能。事实也的确如此，仅仅看 Cassandra 的数据的存储形式就能发现，首先是写到 Memtable 中，然后将 Memtable 中数据刷到磁盘中，而且都是顺序保存的不检查数据的唯一性，而且是只写不删（删除规则在后面介绍），最后才将顺序结构的多个 SSTable 文件合并。这每一步难道不是让 Cassandra 写的更快。这个设计想想对读会有什么影响。首先，数据结构的复杂性，Memtable 中和 SSTable 中数据结构肯定不同，但是返回给用户的肯定是一样的，这必然会要转化。其次，数据在多个文件中，要找的数据可能在 Memtable 中，也可能在某个 SSTable 中，如果有 10 个 SSTable，那么就要在到 10 个 SSTable 中每个找一遍，虽然使用了 BloomFilter 算法可以很快判断到底哪个 SSTable 中含有指定的 key。还有可能在 Memtable 到 SSTable 的转化过程中，这也是要检查一遍的，也就是数据有可能存在什么地方，就要到哪里去找一遍。还有找出来的数据可能是已经被删除的，但也没办法还是要取。

总结

在分布式海量数据应用场景下，对于技术方案选型对整个系统的设计架构起到了非常关键的作用。在存证应用场景下，数据的写入场景远远大于读取，查询实时性要求也相对较高，Cassandra、Hbase的特性非常适用于此场景。

8.2 前沿应用sample

- 存证
- 多链并行计算
- 群签名环签名
- 一对一匿名可监管转账

CHAPTER 9

Wiki

9.1 RUN赋能计划 系列直播

9.1.1 第一期 揭秘FISCO BCOS安全架构

美女学霸工程师

揭秘FISCO BCOS安全架构

导语：

【RUN赋能计划】是FISCO BCOS社区发起的系列直播节目，我们希望用便捷趣味的方式，分享区块链行业热点、FISCO BCOS技术特性和业务实践。我们会定期邀请行业大咖，包括一些在行业内非常具有影响力的案例的负责人以及一线核心工程师来进行分享。这是【RUN赋能计划】系列技术直播的第一期。

主讲人：

陈宇杰：人称学霸小姐姐，主要负责FISCO-BCOS底层平台特性开发，前期工作偏向于隐私保护方面的算法研究和开发，并将其集成到FISCO-BCOS平台，比如说群签名、环签名；后期可能会稍微偏工程，会参与到群组多账本的特性开发中去。

01 为什么要做安全体系?

在揭秘FISCO BCOS安全架构之前，先谈一个前置问题：为什么要做安全体系?

简单地说，就是区块链本身的安全机制满足不了商业级生产的需要，尤其是金融领域的需要。

“区块链”这个概念从中本聪第一次提出，到真正完成商业级生产，其实是存在巨大鸿沟。同时，安全本身也是一种相对的概念，各行业对安全的等级要求不同，尤其是在金融行业，对性能和并发、安全和隐私、以及监管层面等等的要求是非常高的。

为什么这样说呢，我们可以从数据维度拆开看，数据要素包括价值标识，比如资产数、金额、余额这些私有的数据，还有个人信息，数据归属和控制关系，比如你这个人名下有哪些账户，这也是一种数据，还有经常容易被忽略的行为数据，包括交易频率、分时的交易数，比如有几家机构搭了一个链，A和B机构交易，B和C机构交易，A和B每天交易一万次，B和C每天交易一百次，这个其实可以说明很多问题的，比如活跃度，比如关系紧密程度，包括资金来往等等。这也是有很强隐私性的。

数据安全的解析

数据	数据要素	公有区块链	联盟链/机构
资产	价值标识，如资产数，金额，余额等	**公开**	相关方共享 可加密保护
身份	个人信息，如身份，私钥等管理	**匿名不可监管** **不可重置/找回**	不匿名可监管 可重置/找回
归属	数据归属和控制关系，如机构或自然人，采用私钥进行签名交易	保护	保护
交易	交易关系，交易发起方，接收方，牵涉的资产等	**公开（匿名）**	保护
行为	出现频度，如总交易数，交易频率，分时的交易数等	**公开**	相关方共享 可加密保护
文件	大容量数据，如合同PDF，图片，商业大数据等	私有存储或 **公网分布式存储**	独占或相关方共享 可加密保护

区分“**可见性安全**”和“**操作安全**”，分别进行保护

img

交易频率这个东西，在公链上是公开的，你可以打开比特币的公网浏览器，去看某个账号，哪年哪日哪个时候，做过几笔交易，交易的对方是哪个账号，交易金额多少，完全是可以看到的。除了行为数据，在公链上，如比特币、以太坊，资产也是公开的。因为如果他不公开，没法做到全网的验证。但相对而说，他公开的账户隐匿了个人信息，黑客、恐怖分子都可以开账号在链上进行交易，定位不到这个账号的现实身份，你只能查到他的资金数，你也不知道他是谁，没法对他进行监管。

但是这对于很多商业合作场景是难以接受的，商业合作需要保护包括资产、交易关系和出现频度的的隐私，尤其是金融领域，账户身份一定是不能匿名的，匿名没法做监管，没法做KYC。

然后大数据，就是大容量数据，公网会提供一些分布式存储，有些是公有存储。有些私密性要求比较高的场景，会要求数据存储在自己的机房，比如银行业，所有的数据，基本上一个原则，就是不出机房。

总之，商业合作，尤其是金融领域的商业合作对安全的要求远远高于互联网或者说公网区块链所说的安全标准。那安全的保护，只用区块链原生的一些办法是做不到的，因此，FISCO BCOS基于区块链原始的机构，进行了大量的设计，实现了多层次的立体安全保障，包括控制、脱敏、加密、隔离这四个层面。

02 FISCO BCOS是如何保证数据安全的?

img

FISCO BCOS基于区块链原始的结构，进行了大量的的设计，从控制、脱敏、加密、隔离这四个层面，来实现对安全和隐私的多重保护。

控制

先说控制。FISCO BCOS是属于联盟链，联盟链更多在于解决企业场景中的多主体信任问题，提高商业活动效率。联盟链与公链的区别之一就是有准入控制。节点准入控制是指进入联盟需要审核且身份可验证，目前CA技术已经相对成熟，金融机构多半采用CA。新加入的节点可以验证IP地址、证书等，验证通过后才能发起连接加入联盟链，进行共识或者流通数据。这样就能很大程度上避免由参次不齐的参与者产生的一些问题。

FISCO-BCOS区块链节点基于证书链验证，整条证书链包括三级：根证书CA===> 机构证书 ===>节点证书 ， 可以很好支持多机构联盟链准入控制场景；

举例子：

机构A，B，C构成联盟链，且机构A负责为所有其他被允许加入该链的机构颁发证书，机构D想加入该链，需要以下操作流程：

向机构A申请机构证书；

搭建节点时，每个节点均要放置机构颁发的有效节点证书，节点间通过节点证书相互认证，无有效节点证书的节点无法建立连接，从而达到了仅有被准入的机构才可加入联盟链；

用一个日常场景举例，比如，大家一起参加一个线下聚会活动，最后主持人说，我们在坐的一起建一个微信群吧，这样方便后续沟通。然后大家就打开微信面对面建群，这个时候需要输入4位数字的“口令”，这就是一个身份验证。通过这样的验证，你才能参与到这个微信群。

除了节点准入之外，FISCO BCOS还有一系列的控制机制来保障安全，包括角色设计和权限控制系统，提供了细粒度权限控制机制，链管理员通过该机制为机构指定账户分配访问合约某些功能的权限，达到权限控制目的。

比如，我们在联盟链上我们划分了开发、运营、运维、交易员和监管五种角色，如开发和运维分离是基本运作策略，开发只负责提交代码，链上仅有交易员处理账目或进行资产转换，监管可以叫停业务、冻结账号或暂停智能合约、调用接口来及时干预，具有最高等级的权限。引入权限控制的做法和传统交易系统相似，安全性有所保证

另外，还有FISCO-BCOS出块节点注册机制：在系统合约里提供了管理记账节点的工具，仅将节点注册到记账者列表，节点才可作为leader出块；若节点不可信，可调用该工具将其从记账者列表中移除；

黑名单机制：在系统合约中添加了证书黑名单管理工具，若节点不可信，链管理员可通过该工具，将其证书添加到证书黑名单，其他节点会拒绝与该不可信节点连接；

还是微信加群的例子来作比喻，那么根据角色设计不同的权限，比如超级管理员、管理员和普通群员三种，他们的权限是不一样的。超级管理员可以任命管理员、管理员可以发公告，普通群员就没有这些权限。黑名单就类似发现某个群员被盗号了，经常发三俗广告，这个时候管理员就要踢人了。

脱敏

接下来我们说脱敏。高度敏感的数据，转换成伪码上链或者考虑不上链。也可以是用hash摘要或者少量元数据的方式上链。你比如说我用户的身份证不上链，我把他转成一个伪码，但是依旧可以完成身份验证。

其实我们在日常生活中很多数据脱敏的例子，比如买火车票，身份证号码那一栏部分数字会打上星号；点个外卖，外卖单上的手机号也会有部分数字打上星号，甚至某些视频或照片上的马赛克也是属于脱敏的一种形式。

加密

加密呢，是指采用对称加密、非对称加密算法、密码信封等等，以及零知识证明，群签名，环签名、阈值签名，这些神奇的新的加密算法来进行加密。

o 对称加密

l 保证数据机密性；

l 加密和解密使用同一个秘钥；明文+密钥使用很多复杂的变换加密成密文；密文+密钥，经过很多复杂的逆变换解密成明文；

l 常见的算法包括AES， DES，DES优化算法；相比较而言，AES算法性能更高；

咱们FISCO-BCOS平台的落盘加密特性就使用了对称加密算法；

但对称加密算法有一个问题，就是如何共享对称密钥，比较落后的做法，比如说面对面交换密钥；

更好的解决方法是使用非对称加密算法。

o 非对称加密

l 密钥分为（公钥，私钥），公钥可以共享给别人，私钥则是私密信息；

l 在交换隐私信息场景中，Alice向Bob发送秘密信息时，可以用Bob公钥对消息进行加密，Bob在收到密文后，用私钥解密，获取消息，发送了私密信息时，不泄露任何消息内容；

l 主要应用场景包括：密钥交换，数字签名等；

以太坊和比特币使用的签名算法都是ECDSA

非对称加密和对称加密是可以结合在一起用，有点类似我们现在银行使用的HDS或者SSL这种方式，首先是用非对称加密的方式，先交换对称的、符合长度的一个密钥，双方以后就都用这个密钥来对数，进行加密和解密，这在区块链应用的非常非常多。

o 国密

除了对称加密和非对称加密之外，零知识证明、同态加密、环签名、群签名、阈值签名，这些是比较前沿的加密技术，目前零知识证明、同态加密、环签名、群签名已经集成到FISCO BCOS了，大家可以根据场景需要选择使用。安全多方计算、阈值签名等等一系列的密码学算法在紧锣密鼓地研究和开发中，大家可以期待一下下~

国密是由国家密码局颁发的一系列算法标准的总称，对外发布的产品都必须符合国密标准，目前区块链使用的国密标准有SM2,SM3,SM4,国密SSL技术规范。

① 非国密版FISCO-BCOS：节点与节点间、节点与客户端间使用了openssl通信协议，基于ECDHE_ECDSA_with_AES密码学套件，具有前向安全性；

② 国密版FISCO-BCOS：节点与节点：使用符合SSL_VPN标准（国家密码学标准）的SSL通信协议，基于ECDHE_SM3_with_SM4密码学套件，具有前向安全性；节点与客户端：使用openssl通信协议，基于RSA_with_AES密码学套件；

FISCO-BCOS在也通过落盘加密特性保证了数据机密性，将存储于本地磁盘的所有数据加密。

隔离

随着新型技术的出现，如量子计算机，一些密码学算法已经不安全了，所以加密算法都存在理论上被攻破的可能性，但是金融业的数据，可能要求是存五年、十年甚至永久保存的，为了防止数据被攻破，对金融业来说，最彻底的安全保护就是隔离，不把数据发给交易不相干的人。这是我们的一个安全价值观，并不是说否定加密算法或其他的一些措施，金融业对安全的要求确实是极致。

03 几种新型的加密技术

零知识证明

零知识证明指的是证明者能够在不向验证者提供任何私密的信息的情况下，使验证者相信某个论断是正确的。有一个小故事，可以帮助我们理解“零知识证明”的原理。

故事讲的是阿里巴巴，有一天被强盗抓住了，强盗向阿里巴巴拷问进入山洞的咒语。面对强盗，阿里巴巴是这么想的：如果我把咒语告诉了他们，他们就会认为我没有价值了，就会杀了我；但如果我死活不说，他们也会认为我没有价值而杀了我。怎样才能做到既让他们确信我知道咒语，但又一丁点咒语内容也不泄露给他们呢？阿里巴巴想了一个好办法，他对强盗说：“你们在离开我一箭远的地方，用弓箭指着我，当你们举起右手我就念咒语打开山洞的石门，举起左手我就念咒语关上石门，如果我做不到或逃跑，你们就用弓箭射死我。”多试了几次之后，强盗们不得不相信了阿里巴巴。

这样，阿里巴巴既没有告诉强盗进入山洞石门的咒语，同时又向强盗们证明了，他是知道这个咒语的。这就是零知识证明。

FISCO BCOS 开源了可被监管的零知识证明方法，基于零知识证明方法，为用户提供一种保护隐私的、可被所有节点验证的秘密交易框架的同时，为监管者提供解密秘密交易的监管接口。基于这个特性可以实现，真正的匿名转账，并且可以被监管。

这里真正的匿名转账，指的是通过零知识证明，能够证明这个转账确实存在，但是完全隐匿转账双方身份，这样能够实现深度的隐私保护。我开个脑洞，想起一个经典的例子，如何证明你妈是你妈，常规路径是到户口所在地开证明，这样必须暴露你妈和你的身份，零知识证明是提供另一种可能，在不查户口，不知道你妈和你是谁的情况下，也能证明你们之间的关系。看起来相当有想象空间啊。

群签名和环签名

群签名和环签名都是签名算法，它们主要用于保护发送交易者的身份信息，相较于零知识证明，性能更高，但无法做到隐匿交易内容。

(1) 群签名算法：

① 多个签名者和群主构成一个群，并持有签名私钥，某个签名者对交易签名后，其他人通过验证交易，仅可知道该签名者所属的群组，但无法得知签名者具体身份信息

② 群签名还具有可追踪性特征，比较适用于监管场景，当要追踪某笔交易发起者身份时，群主可通过该比交易的签名追踪签名者身份信息；

(2) 环签名：

类似于群签名，但其不具有可追踪特性。

多个签名者构成一个环，某个签名者对某笔交易进行签名时，会带上其他环成员的公钥信息，其他人通过整个环的所有成员公钥验证环签名的有效性，通过签名，仅可知道该签名者所属的环，但无法追踪具体的签名者身份；

这样说起来可能比较绕，举个例子你就明白了，比如在匿名投票场景中，使用环签名，可以确认该投票是有效的（是该群体成员投出），但是无法确认是谁投的；而群签名则只有“群主”，可能是监管机构，可以通过签名追查到是谁签署的。常见的应用场景除了匿名投票，还有拍卖、竞标等。

这些都属于隐私加密，隐私加密是我们研究的一个特别难、特别有挑战性的一个方向，它也有实用很强的商业意义，因为所有的商业场景，你再说怎么公开，怎么分享、共享，都是有隐私诉求的，机构和机构之间的隐私，个人和机构之间的隐私，个人信息保护等等。

值得注意的是，虽然安全和隐私保护这么重要，但是也不能说是越高越好，安全强度满足场景需要即可。有个不可能三角，说的是安全、规模、效率这三个方面不可兼得。前面提到的零知识证明、环签名群签名，算法非常复杂，效率会低一些。强隐私保护和高性能确实是难以兼得的，但在选择隐私保护方案时，我们会将其与性能做折中，在不损耗太多性能的情况下，提升系统安全性。

当然，对于新型的强安全性算法，我们也一直很关注，并考虑将其应用到区块链系统中，已经实现的包括零知识证明、群环签名(群签名算法开销不大，完全可落地于生产环境中)， 持续优化已实现的算法性能；还会持续探索安全多方计算、抗量子攻击密码学等领域。聚合签名、多签、阈值签名也是我们持续关注的领域，引入这些技术可以加固FISCO-BCOS整个平台体系的安全性。

有兴趣了解更多的同学可以参考

1、可被监管的零知识证明方法：基于零知识证明方法，为用户提供一种保护隐私的、可被所有节点验证的秘密交易框架的同时，为监管者提供解密秘密交易的监管接口。

-可实现的业务场景：匿名转账的实现和监管。

-关于该框架的实现及详细操作请见文档

零知识证明RPC服务

零知识证明库

2、群签名和环签名链上验证功能：使用群签名验证其他人在链上验证签名时，仅可获知签名所属的群组；环签名则是通过AMOP将群签名发送给上链结构；二者皆可保证用户的匿名性。

-可实现的业务场景：拍卖、竞标、匿名投票、匿名存证、匿名交易等场景。

-关于该框架的实现及详细操作请见文档

\3. FISCO-BCOS支持国密算法:

国密是由国家密码局颁发的一系列算法标准的总称，对外发布的产品都必须符合国密标准，目前区块链使用的国密标准有SM2,SM3,SM4,国密SSL技术规范。

l SM2:国密非对称算法，由公钥、私钥组成，用于签名、验签、加密、解密。

l SM3:国密杂凑算法，用于杂凑运算。

l SM4:国密对称算法，使用相同密钥进行数据加密、解密。

l SSL规范:数据传输，国密密钥交换及数据传输加密标准。

现有客户端发送交易到节点，节点与节点之间共识使用ECDSA，SHA3-256算法。数据落盘加密使用AES算法。节点与节点之间SSL连接使用ECDHE密钥交换算法进行SSL连接。

FISCO-BCOS国密版本使用SM2,SM3算法从客户端发送交易到节点，节点与节点之间共识。数据落盘加密使用SM4算法。节点与节点之间SSL连接使用国密ECC密钥交换算法进行SSL连接。

使用国密算法符合国家密码局标准，安全可靠，达到监管要求。同时，亦可选择使用编译开关打开或关闭国密算法版本。

使用说明文档:

国密版FISCO-BCOS说明文档

web3sdk支持国密的说明文档

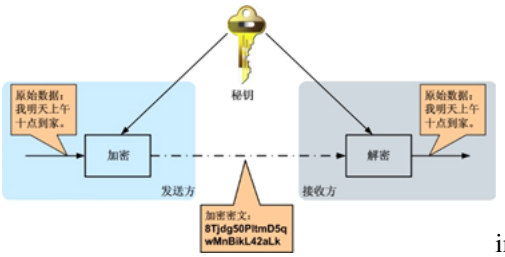
答疑

丁丁：问环签名群签名对监管者是不是透明的?

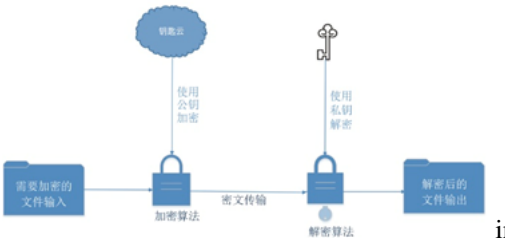
回复:环签名是一种用户完全匿名的签名方式，对监管者是匿名的，除非用户的密钥是由监管机构分发的，否则只能通过一定的逻辑限定签名人的组别和签名次数签名。群签名是可监管的，可以设置监管为群管理员，在有必要的情况下，监管可以通过一些操作获得进行群签名用户的身份。

zhaoj@轨道舱：问对称加密和非对称加密，哪个效率更好些?

回复:我们一般说加密的效率一方面包括了对明文加密和解密的速度，但是这都是针对加解密阶段来说的。另一方加密的效率也包括加密方法的密钥生成，密钥协商阶段等。对称加密的一大缺点是密钥的管理与分配，换句话说，如何把密钥发送到需要解密你的消息的人的手里是一个问题。在发送密钥的过程中，密钥有很大的风险会被黑客们拦截。现实中通常的做法是将对称加密的密钥进行非对称加密，然后传送给需要它的人。



img



img

假设场景是当Alice希望将文件加密发送给 Bob

针对于对称加密，Alice需要采用各种方式把密钥k安全的传递给Bob，然后用k对明文进行加密算法，再将加密后的密文e传递给Bob。Bob收到密文e后用k进行解密。

如果采用非对称加密。首先Alice需要去公开密钥的管理服务器查找 Bob 的公钥，然后找到 Bob 公钥对应的加密算法。Alice 在确定这个公钥的正确性之 后，用 Bob 的公钥对文件进行加密，将加密后的数据发送给 Bob。

当 Bob 收到 Alice 发送的加密文件之后，Bob 使用自己的私钥和解密算法对文件 进行解密，从而得到解密后的文件。

其实对称加密效率更高，所以主要用于大容量数据加密场景，非对称加密安全性和密钥长度有关，性能相对较差，主要基于一些难题构造的算法，常见的包括RSA，DSA，ElGamal，ECDSA等，主要运用在数字签名和密钥交换领域，前者用来验证签名者信息，有防篡改功能，后者用来交换秘密信息，如对称密钥，对称加密算法一般基于复杂的变换构造，安全性和密钥长度和变换方法，变换次数有关。

wx：问god如果不是节点，那出块节点故意不将god账户的信息出块共识，那不是就是失效了?

回复:god账号是构造创世区块的必须信息，不设置god账号，或者账号与其他节点不一致，节点就会异常，这种事情，就算是恶意节点也不会想搞死自己

邱明凯：问god毕竟只有一个节点 如果再实际使用中 这个除了故障 系统会出现问题吗

回复:god是账号，不是节点哈，有创世节点的概念，在pbft共识算法中，如果创世节点异常了，但超过三分之二节点正常，链运行正常

王路：问权限控制到合约接口，没有权限的证书身份不能访问，有不有可以直接去节点leveldb的数据库中看合约里的数据?

回复:权限控制是在底层代码里实现的，不是基于证书的，如果某个账户没有被授予调用合约接口权限，底层就会直接返回无访问权限的异常到客户端，根本无法调用合约

光路：问链管理员由谁承担呢?

回复:这个需要看具体机构具体角色设计。

王蓓：问新加入的节点D是需要给A\B\C都办法节点证书吗?

回复:新加入机构要先向链管理员申请机构证书，对应直播中的机构a管理员，机构d的节点证书由机构d管理员颁发。

希望大家多多支持FISCO BCOS【RUN赋能计划】系列技术直播，如果大家觉得不错的话，欢迎戳这里给我们打星星，也希望大家给我们提提建议，让FISCO BCOS社区变得更好。

扫码加入官方微信群，与学霸小姐姐交流~

https://github.com/FISCO-BCOS/FISCO-BCOS/raw/master/doc/FISCO-BCOS.jpeg

9.2 漫谈系列

9.2.1 WIKI：漫谈共识机制

作者：fisco-dev

共识机制是区块链领域的核心概念，无共识，不区块链。

区块链作为一个分布式系统，可以由不同的人或机构，将安装了区块链软件的计算机（简称节点）加入到网络里，然后共同计算数据、共同见证交易的执行过程，并确认最终计算结果。

如何协同这些松散耦合，互不信任的节点一起工作，达成信任关系，并保障一致性，持续性，可以抽象为“共识”过程。

共识需要解决的几个核心问题是：

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。
2. 做为互相不信任的参与者，为什么要采纳和相信某一个人给出的记账。

3. 怎么保证大家最终收到的结果都是一致的，无错的。

公链上把"激励"也作为一个核心的考虑项，来保证链的可持续发展，这是合理的，但是联盟链不一定会采用代币激励，所以本文不把激励放到这三个核心问题里。

以下介绍几个典型共识的策略：首先介绍 PoW（Proof Of Work）工作量证明。POW是比特币采用的共识算法，从诞生起运行至今，表现稳健，是史上最成功的共识算法，没有之一。POW的哲学简单可靠，大家理论上都可以发起记账，拼算力，看谁能先算出一个小概率的随机数，也就是俗称的挖矿，这个随机数在数学角度有严谨的推演，通过动态调整的难度策略限制，无论矿工投入的硬件多强，都能控制在10分钟左右挖到一个矿，算力强的矿工因为可以在这段时间内计算更多的随机数，所以更有机会比其他人先挖到矿。挖到矿的矿工就可以广播自己的记账结果，全网的其他节点可以选择接受这个结果，然后默默开始的下一轮的挖矿。挖到矿的矿工同时得到一笔算法赋予的奖励，也就是一些比特币。如果刚好两个矿工完全同时的挖到矿，那么就会出现竞争，网络上出现了两个记账结果，这个时候其他节点会随机选择一个，或者按顺序选择自己先接收到的一个，继续在它基础上进行挖矿（基于该记账结果的基础，再挖到的新的一次矿，称为新一次确认），由于网络有随机延迟，随机策略等区别，一般会有一个记账结果会被更多人接受，有机会更快的被持续确认六次，另一个就被抛弃了。这就是竞争和分叉处理。POW的表现有点像单纯的原始年代，大家都凭力量和速度去挖矿淘金，谁先挖到一块金子，就胜出一次，看起来非常的公平公正，简单粗暴，无可挑剔，其简单性也让这个体系得以稳定运行多年。毕竟拼了老命去挖矿的矿工，只会继续拼命挖矿来加固自己的成果，轻易不会作弊，矿工作弊会导致网络失去公信力，其辛苦挖来的持有的资产也会贬值。其他人想攻击这网络，需要投入的算力要比现有的矿工多，得比现有的已经很熟练很有力量的矿工更努力，如果收益不是很可观，是不值得的 – 换句话说，反正比特币网络有奖励，如果我有这超越了其他矿工算力，为何不去挖矿赚钱，破坏网络吃力不讨好干啥（除非别有所图）。于是，博弈论就这么玄妙的产生了作用。同时，这种掰腕子秀肌肉的哲学，最让人吐槽的就是不环保，能耗太多，效率较低，无脑算HASH，对科学计算也没什么贡献（曾经有将算力用于计算天文问题来挖矿的方案，但是没有形成大规模效应），参与者能通过暴力堆积硬件，采用极致优化（只能做HASH计算的）芯片，在网络里掌握话语权，也就是所谓的算力集中问题。

总结一下，之前的三个问题在POW里的解决方案

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。—虽然大家都有机会参与记账，算力强大的人更有机会成为记账者，谁劲儿大谁当擂主。
2. 做为互相不信任的参与者，为什么要接受和相信某一个人给出的记账。—因为记账者付出了算力的巨大代价做为背书，他可以为记账行为负责，可以倾向认为他不作恶。
3. 怎么保证大家最终收到的结果都是一致的，无错的。—其他人可以从网络里同步到记账者的数据，通过区块里的hash算法校验数据，因为区块数据有难以产生（需要算力），容易校验（hash容易计算）的特性，大家对数据校验通过后都采用记账者给出的数据，全网一致，如有分叉，6次确认后解决。

考虑到POW的短板，行业里提出了PoS（Proof Of Stake）权益证明的思路。PoS的基本思想是让在网络中拥有更多权益的人有机会在更短时间里做更多的决定，因为毕竟权益持有者人更倾向维护网络的利益，且害怕作恶后被惩罚，损伤自己的声望和财产。PoS的算法实现有很多变种，其中一个比较典型的是引入了"币龄"的概念，比如，一个人手里拿了一笔钱，而且持续拿了一段时间，那么基本上可以认为这个人是比较忠于这个网络的，他的权益值就比较高了。为了鼓励人们持有代币，一般是有利息可图的，所以也鼓励了人们倾向持有资产，不断增值，维持权益。对权益较高的人，如何获得记账权力，也有多个变种实现，在继续采用挖矿算法的网络里，可以加权的降低他的挖矿难度，让他更容易挖到矿，获得记账权。或者采用加权的轮询算法，轮流让权益拥有者轮流记账。可以认为，PoS引入了财富做为公信力背书的一个考虑角度，和简单粗暴谁有力量谁说了算的PoW对比，有利于降低纯计算资源的消耗，看起来也没有那么肌肉感了。

总结一下PoS：

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。—持有权益较多的一拨人竞争或轮流记账。
2. 做为互相不信任的参与者，为什么要接受和相信某一个人给出的记账。—记账者用自己的财力做为背书，大家相信他们不会轻易作恶，如果作恶，可以用经济方式惩罚他。
3. 怎么保证大家最终收到的结果都是一致的，无错的。—通过验证记账权益和数据，大家相信这一轮的记账者，一致使用他的记账结果 。

在PoS的基础上，又发展出了一个DPoS的共识算法。和PoS对比多了D，全称是"Delegated Proof of Stake，股份授权证明机制"，原理是让所有持币人都有机会选出自己的代表，比如全网有1万个参与

人，通过一定的算法，参与人以自己的代币为权益证明，选出101个代表，这些代表可以轮流或者采用PoS算法加权的获得记账权，进行记账。 DPoS理论上不要求选出的代表个体本身是权益所有人，看起来更民主，更开放。网络参与者做为选民有机会选出自己的代表，来给自己的利益代言。如果选出的代表不作为（轮到自己记账时不记账），或者作恶，可以把他们踢掉，如有必要进行惩罚（选民们也有可能被惩罚）。否则记账者有机会获得相应的奖励，也有可能将奖励发放给选出自己的民众们。

总结一下DPoS：

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。—拥有权益的散户，分别选出自己的代表参与记账。
2. 做为互相不信任的参与者，为什么要接受和相信某一个人给出的记账。—大家相信代表，因为他们能代表相当一部分人的在网络里的利益。
3. 怎么保证大家最终收到的结果都是一致的，无错的。—相信代表，一致使用他的记账结果 。

以上的都是在公有链里采用的共识算法，都能解决可信和一致性的问题，区别在于组织方式和计算代价不同，有一点相同的是：都是追求“**最终一致性**”，而不是强一致性。也就是说在网络中，由于网络故障、通信分区、权益和算力相近等情况，依旧可能有多个记账者在竞争记账，所以每次记账可以理解成一次提案，大家暂时保存这一次提案的竞争结果，最终确定下来可能要依赖多次竞争，也就是所谓的一顿烧烤解决不了的问题，那就再来一顿，然后再一顿。直到大家看到，之前的某一次记账，都被大家接受了，而且是接受了好几次，才达成**最终一致性**，这个时间是相对比较长的，比如比特币的6次确认，需要一个小时。公链里的共识以及其他交易行为，可以理解为一种概率游戏，随着时间推移和投入到共识里的工作量或权益越多，确定性的概率就越大，大到概率上几乎不可逆，或者必须投入匪夷所思的大量的资源才能推翻时，就达成了确定性。

我们的目标是实现联盟链的共识，联盟链和公有链不同的是，参与记账的人，身份是可知的、可控的，可能有监管或者经济共同利益之类的措施来约束他们。

另外，联盟链的场景倾向与追求强一致性，也就是说一笔交易发生了，就不能再被分叉或回滚推翻了。所以联盟链通常使用的是PBFT和Raft算法以及其变种。

BFT（Byzantine Fault Tolerance）即“拜占庭容错算法”，其细节说起来特别复杂，来自80年代的一个理论，据说至今没有一个完全的、普适性的权威实现，Babara Liskov在2002年提出PBFT，放松了约束来解决拜占庭问题，而因此获得了图灵奖。我们参考“实用性的拜占庭容错” Practical BFT做了一个实现，是自己从头开发的。

PBFT的记账者在初始阶段就是相对固定的，联盟链里可以根据不同的场景来选择记账者，如机构之间线下协商、线上配置，也可以引入DPoS的思想，先通过一些选举策略选出一批记账者，但这样在联盟链里可能带来额外的复杂性，如果商业场景需要，可以在既有基础上开发PBFT-dPos。

所以我们的实现是可动态配置式的记账者列表，在具备共同利益的的联盟链网络里，所有或大部分机构都参与记账，大家一起维护商业利益，如果某个机构在记账过程中犯错、作弊、或者不工作，都可以通过少数服从多数容错，然后进行事后追责，保护大家的利益。算法的容错能力是能接受1/3成员的错误和作弊，如7个成员，只要作恶或出错的成员在两个和两个以下，网络就是安全的。但如果网络里有很多个作恶者，那么这个网络可被视为失效。

BFT算法的过程是一次提案，几步提交，在这个过程中有复杂的状态机维护的细节。起始时可按一些简单的规则，如轮次机制，让某一个机构的某一个节点在某个区块高度上，成为议长，得到记账权，然后进行记账提案，其他节点对提案进行数据验证（验证签名，验证交易数据，验证合约计算结果等），觉得都ok之后，返回一个签名确认，议长收集大家的签名，达到一定的数量后，认为达成少数服从多数的效果，则将记账结果广播出去，大家在将记账结果存下来之前，还可以再进行一次投票和计票，确认大多数人都收妥了记账数据并无异议，再将记账数据存下来，开始下一轮。

可以形象的描述下：一个团体，比如有7个人，大家都有投票权，要对某次聚餐计划的选择进行投票，这个时候某一个人发起一个提议，大家看ok呀，纷纷举手，提议的人定时数数，举手的人超过4个，就达成决议，否则再换一个人提议。如果提议达成，决定了吃什么什么时候吃，发起人给大家群发个邮件制定日程，大家接受日程并给出反馈，当看到有不少人都接受了邮件日程不会反悔，那么，这顿饭就成行了。

如果团队人数较多，也是可以让一部分人成为投票代表，比如一个上百人的班里，由选出来的班干部投票就可以了。其他人成为“观察者”，无条件接受这一部分人通过拜占庭容错算法得出的结果，和DPoS的思想类似。

PBFT算法在参与人数较少的联盟链里，有相当可观的实用价值：

1. 少数服从多数，体现了民主性。
2. 大家参与投票，都能刷存在感。
3. 不需要粗暴的算力介入，相当环保。
4. 多步提交反复确认，一旦共识完成就达成一致性，可以短时间内就达成确定性的交易。
5. 每次投票都有数字签名，可以抗抵赖，不会投了票不认。
6. 可以不依赖代币经济鼓励来保证记账的正确性。

所以，PBFT相对公链的各种共识，可以理解成特定的已知身份的团队内的快速达成一致的一种高效算法，特定团队的形成也是至关重要的，所以会有PBFT-Auth(经过验证的记账人), PBFT-PoS,PBFT-DPoS等多种变化。是否需要在投票和容错的部分进行常场景性的优化（如带加权系数的投票，容错由1/3改为1/2或者改为0容错等等），要看场景需求，目前来看，保证有一个相对简单，运行稳定的PBFT算法是首要的。

总结一下PBFT:

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。–所有人平等的轮流参与记账，或者经过许可的，或者大家选出来的一拨人进行轮流记账。
2. 做为互相不信任的参与者，为什么要接受和相信某一个人给出的记账。—少数服从多数，每次记账的结果都是多数派的投票胜利。
3. 怎么保证大家最终收到的结果都是一致的，无错的。—通过多步提交反复确认，保证提案和最终结果都是正确的，是多数人承认的，是全网一致的。

PBFT的小问题在于状态机维护复杂，投票往返步骤较多，容易受网络波动影响，对网路延迟和丢包相当敏感，当参与者数量增多，不稳定因素会被放大，其稳定性和效率可能会显著下降，这也是我们在持续的踩坑和优化的部分，要做到一个企业级环境可用的，健壮的PBFT共识，是非常有挑战性的，不但需要对区块链的数据和运作哲学有很深的理解，也要对分布式系统的通信，协作，性能优化有大量的实践经验。

结合以太坊帐户模型和智能合约引擎机制，我们对PBFT也做了大量的优化，比如把串行的任务改成并行，把不必要重复计算的数据（如区块数据打包，签名等）缓存下来，快速侦测记账者存活，加快切换速度等等，都是复杂的工程，需要一次又一次迭代的长夜漫漫的脑补、实现、测试、修BUG、压测、量化，足以重新写一篇长文了。

联盟链还可以采用一种共识算法Raft，简单的说可以理解成大家选出一个记账者，如果它稳定运行没有挂掉，就由他记账，大家无条件接受他的记账结果，相信他是诚实的，如果他挂掉了，那么大家是可以通过超时或网络探测感知，然后快速启动一轮投票，来选出一个新的记账者，然后继续无条件的等它记账，这样就达成了容错性。

这在信任度较高，机构组成简单的的联盟链或者一个机构内的私有链里，是比PBFT更加高效的一种做法，因为不需要多步的反复确认，受网络影响的可能性也小很多。

但是Raft算法并不解决记账者耍花样的问题，如果选出的记账者作恶，其他人当时是没有办法识破的，最多只能通过事后检查来发现，在确定性要求较高的场景，比如收了钱就要交货的DVP场景里，容易出现钱货两空的情况。另外，Raft算法有可能在网络波动或竞争情况下出现短暂的网络分叉，需要多次确认。

Raft适合用于专注解决可用性（保持系统稳定运行），追求相对较高效率（比优化后的PBFT大致高20~50%的效率）的场景，基本忽略欺诈可能，且对交易延时要求可以放宽（等待多次确认）。

为了在速度和抗欺诈之间获得平衡，也有出现PBFT-Raft混用的共识，比如让一个记账者固定出块，其他人极少轮次的投票，或者进行快速的后置检测，如果发现Raft的记账者在作恶，立刻回滚冲正，并踢掉并惩罚作恶的记账者等等，这就有一些场景化的需求需要考虑了。

总结一下Raft:

1. 谁在这个网络里有记账权，也就是做为leader发起一次记账。–选出一个记账者，让他做为唯一的可信者进行记账，除非挂掉。
2. 做为互相不信任的参与者，为什么要接受和相信某一个人给出的记账。—无条件相信记账者，from the beginning till the last,至死不渝。

3. 怎么保证大家最终收到的结果都是一致的，无错的。—记账者说的都是对的，大家都用他的数据，是一致的。

回顾一下：POW：用尽创世纪时代的洪荒之力求生存求发展。PoS：采用权益加权，减少计算资源消耗。DPoS：一定程度的民选制度，用民意取代集中的权益，民意代表们把握话语权。PBFT：少数服从多数，快速达成共识，体现参与度和民主性，有较强的确定性和抗欺诈性。Raft：强调可用性和最终一致性，效率较高，不强调抗欺诈性。

共识算法的研究是很有意思的课题，我们也看到共识算法的选择和演进，隐约和人类社会的不同阶段是对应得上的。算法其实是社会里人和财产的关系在计算机世界的一种映射，研究共识算法，不仅是需要研究计算机理论（图灵机时代，其实计算机也能体现人性），也要去深刻的理解社会学，经济学，心理学，博弈论。毕竟，在一个不完全互信的环境里，宛如黑暗森林，需要复杂的算法来协同、斡旋各怀心事的参与者，并且能惩罚作恶的、奖励诚实的，目前社会制度其实也没有完美的做到，一切努力只是尽量的使系统运作的尽量顺畅，让世界更好一点点。

9.2.2 漫谈系列:区块链应用后台“混合架构”介绍

作者：fisco-dev

本文从应用开发者的角度出发，介绍一个简单的区块链应用的“混合型”后台架构。

1.从分层到混合架构

区块链技术体系本身是分层的，如图所示。

img

如果基于已有区块链底层平台进行业务开发，应用开发者只需要关注Dapp的设计，服务接口调用，智能合约编写即可，这几个部分除了智能合约之外，基本上和传统的应用开发差异不太大。其实图灵完备的智能合约如sodility，作为一种受控的脚本语言，其语法的复杂性和API的数量，也都远远少于java，javascript，python等大家更为熟知的计算机语言，一般来说，有计算机基础的开发者用一到两周时间专注阅读开发文档和参考样例，基本上就可以写出通顺的智能合约了。

套用常见的后台业务系统MVC设计模式+数据持久层，看起来区块链+智能合约已经基本上可以应对大部分架构，那么是否由客户端直接往区块链发送请求，由智能合约完成业务逻辑，区块链进行分布式存储，客户端获取结果数据进行展示即可？这是一种理想情况，实际业务开展中，需要考虑以下因素：

1：为了给客户端提供更好的体验（更安全的通信，更稳定的网络连接，更快的响应速度，更多的数据维度），通常需要有服务负责从各种网络接入客户端，进行协议转换，对数据进行缓存加速，整合丰富的数据包括图片视频等。

2：庞大的业务模型里，有一部分只和机构内部逻辑有关，不需要上链进行分布式交易。如用户注册管理，竞价拍卖过程，买卖单撮合，营销活动等环节，未必都需要全程上链，只需要把需要跟踪的，或者和其他机构协同的部分上链。

3： 传统业务逐步上链，有一个中间过程，需要把传统业务和链上模块进行对接，并行旁路的运行。

4： 对数据需要归集处理的流程，如统计报表，大数据挖掘，相关性分析等，不适合直接在区块链上检索数据和运行计算任务，需要把数据导出在链下深度加工处理。

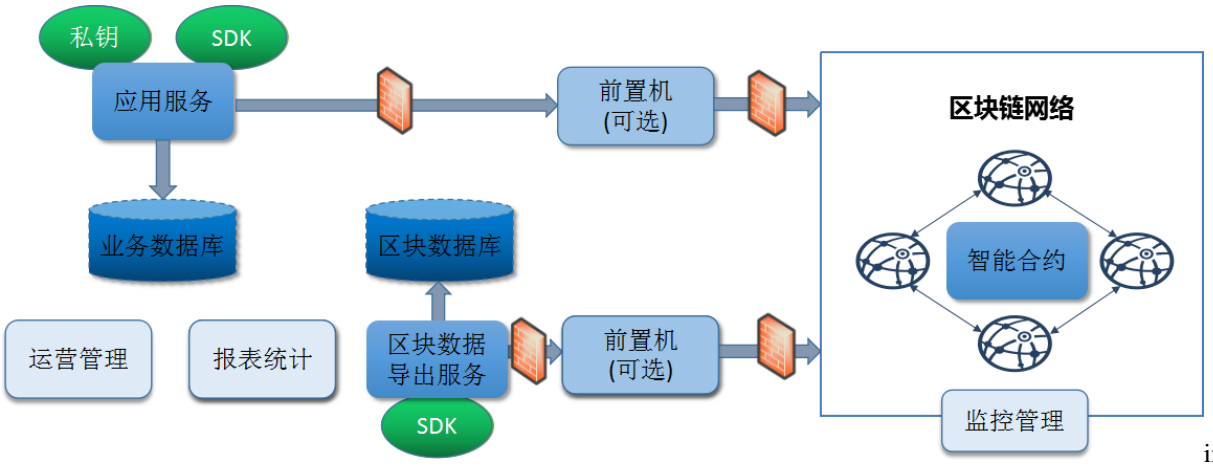
5： 和隐私相关的数据，需要在链下处理脱敏后再上链。

6： 专注于处理分布式事务的区块链底层平台不适合直接面向山呼海啸的海量用户请求，需要采用前置的上链流程进行削峰平谷，把并发访问变成异步处理。

区块链系统可以聚焦于在跨机构的、分布式的场景，实现准实时交易、数据流动、清结算等。而繁杂多变、计算密集、有特定隐私性要求的业务逻辑通常还是在链下完成，所以目前来看，“混合式架构“的应用开发，是比较务实的选择。

2. 混合架构解析

以下是一个典型的”混合架构“的概念图，基于这个架构图，可实现一个最小化的基于区块链的应用架构。以下对各模块进行一一介绍。



img

应用服务

面向具体业务逻辑的应用服务，如供应链、电商、存证、其他金融业务等。应用服务依旧可以用类似LAMP，EJB，SSH，微服务等成熟技术开发，业务数据库可用mysql，oracle等成熟的解决方案来存储不需上链的业务数据，如用户个人资料、商品介绍、仓储、营销推广数据等。 当在业务流程中产生了一笔需要上链的交易，如资产转移，由应用服务向区块链发起交易请求，并等待区块链共识确认。

应用服务可以引入私钥管理的模块，私钥由专用的算法进行加密妥善存储，由应用服务通过私钥管理模块加载并对交易签名。如机构是代理用户转发交易，交易是由用户自签名，则机构不需要管理用户私钥。

应用服务通过区块链平台提供的SDK发起交易，SDK向业务层封装网络通信，协议编解码，异常处理等细节，暴露友好的面向对象的功能接口，如智能合约定义的转账接口，应用服务面向接口编程，专注于实现业务逻辑，而不需要承担底层技术实现的开销。

应用服务和区块链通信有几种模式：

第一种，应用服务和区块链通信采用同步方式，即应用服务向区块链发起一个交易请求（sendTransaction），然后同步等待区块链系统处理反馈结果（getTransactionReceipt），再进行下一步操作。这种模式适用于强事务的场景，并发能力相对较低。

第二种，应用服务和区块链通信采用异步方式，即应用服务向区块链发起一个交易请求，然后继续处理下一个事务。区块链处理结果返回事件由sdk负责接管，SDK异步回调应用服务进行后续处理。这种模式相对灵活，并发表现较好，调度流程稍复杂，对编程要求略高，需要具备多线程，多通道，堵塞和竞争处理等基础知识。

第三种，应用服务和区块链处理流程进行隔离，应用服务将需要发往链上的交易先持久化，如放入数据库或队列，再由另一个上链服务定期从数据库或队列中pop出需要上链的数据，向区块链系统发送，并

把区块链处理的结果写入数据库或队列。这种模式适用于应用业务流程能和分布式协作流程清晰解耦的业务场景，如实时交易收妥，再准实时的进行对账清结算或数据存证等。

这几种方式根据不同的业务场景要求可选，会在系统容量，并发吞吐能力，时延方面表现各有不同。

原则上应用服务和区块链系统之间要保证数据的"不错，不丢，不乱"，所以在应用服务和区块链系统之间可以有一个"对账"流程, 定期检查应用服务请求上链的数据是否被区块链系统正确处理，对漏发的，超时或未处理成功的数据应进行重发或差错控制，以保障业务质量。

区块数据导出服务

区块链通常被认为承担了"分布式数据库"的功能，区块链上每个节点确实保存了和全链一致的数据，在当前的实现中，区块数据通常是存在文件型数据库如leveldb里，所提供的RPC数据功能接口，一般是基于单个区块、交易、合约的key-value式查询。对数据的复杂检索，范围检索，批量检索，关系型检索...目前的区块链平台大多是不能直接支持的。

"数据导出服务"是一个用于构建"区块链数据仓库"的解决方案，使区块链和已经为人熟知的关系型数据，分布式数据库，大数据平台进行有机结合。

为区块链设置一个准实时的数据ETL（Extract-Transform-Load）流程，使得链上一旦新生成区块，就能立刻把相关的区块证明，区块里包含的交易列表、交易明细、交易结果、智能合约状态数据、链上配置信息等全部导入到链外的数据仓库里，数据仓库的数据只增不减，写入后不会再发生变化，且随时可以和链上数据源进行比对验证，保证其完整性和不可篡改性，借助数据仓库本身强大的查询能力、分析能力和数据挖掘能力，可以对区块链上产生的数据进行复杂的整合和加工，以满足商业智能、监管监控、反洗钱、数据报送等场景的需求。

数据导出服务里通用的部分如ETL定时任务，区块导出，交易导出，状态数据导出等，可以由区块链平台提供参考实现，具体到业务数据，如链上的订单数据，转账数据和交易结果等，则和业务密切相关，需要业务开发者针对具体的数据结构进行解析入库。区块链平台的SDK提供了数据解析的功能。

智能合约

智能合约是区块链系统的核心能力之一，图灵完备的智能合约可以让开发者天马行空的实现业务逻辑，且保证区块链网络上的事务性、一致性。

但另一方面来看，在区块链上运行智能合约相当于在消耗全网的计算资源，网络资源，存储资源，过于复杂的业务逻辑一旦依托智能合约实现，在当前的技术阶段，会使分布式网络难以承担。

所以，基于"混合式架构"的指导思想，在设计智能合约时应进行拆解，只有和跨机构协作共享相关的关键流程和数据，才需要使用智能合约实现。非关键的，或者单个机构就能完成的业务流程，以及详细的用户个人信息，商品描述数据，如文件，视频，音频等，都不需要使用智能合约描述，如需采用区块链对这些数据进行跟踪鉴证，可以使用摘要算法等方式，将不可篡改的数据摘要上链，把明文妥善保存在机构内，需要时再通过文件传输等方式进行共享。

智能合约虽然支持继承等面向对象的特性，但考虑到合约的可维护性，应尽量保持层次扁平，接口清晰，保持简洁的互相调用的关系，避免写出庞大的运行缓慢也难以维护的智能合约。

前置机

在高安全性要求的机构，不同的模块会部署到有防火墙隔离的区域，应用服务所在的区域甚至是无外网连接的，而区块链系统先天的需要和外界连接，所以机构内的应用服务通常需要通过部署在DMZ（隔离区）前置机进行代理转发，和区块链系统通信。应用服务和前置机，前置机和外网之间，都会通过防火墙隔离。

前置机可定位成一个"通信转发器"，对应用业务逻辑无介入，只进行必要的协议转换，把通信包在内外系统之间进行转发，且服务本身无状态，便于无差别的多活部署。

如果应用服务所在区域可直接和区块链所在的网络通信，则不需要使用前置机。

其他

做为完整的应用系统，运营管理，统计报表，监控运维系统等等都是必不可少的，采用混合架构的思路，新增的模块采用数据库读写、日志输出、RPC接口方式，即可以无缝和机构内部耕耘多年已经成型的IT系统进行结合。

3. 结语

综上所述，在混合架构中，

1.应用服务接入用户，面向复杂业务场景流程，产生交易数据

2.区块链+智能合约解决机构分布式协作和一致性问题

3.数据导出服务解决数据的后处理问题

链下的归链下，链上的归链上，各司其职，完成自己擅长的工作，才能达到1+1 > 2的成效。

9.2.3 漫谈系列：聊聊区块链数据的“安全”和“控制权”

作者：fisco-dev

区块链通常会被认为是“安全”的，同时宣称”把数据主权还给用户“，听起来很厉害的样子，但这里面的”安全“和“主权”是怎么界定的呢。

先看广为人知的区块链（默认指比特币和以太坊）系统上的几个事实：

- 区块链上的数据（区块，交易）是全网广播的，所有节点都可以收到数据。
- 数据对节点来说默认是明文的，记账节点，普通节点都可以看到数据的明细。
- 在公链上，用户不需要公开身份，是匿名的，链上看到的只有一串没有什么身份意义的二进制唯一性地址。
- 用户通过私钥控制自己的资产，私钥保障了用户对账目的动账权。

类似门罗币，zcash， 加密以太坊这些项目理论上对隐私会有更好的保护，但是由于算法极其复杂，效率尚未达到理想效果，还没被广泛接受或被工业场景大规模使用，预计还需要一定的成熟期。

我们再把对数据的要素分为几个部分，每个部分的敏感程度可能都会和业务相关。

- 所代表的价值，如资产数，金额，余额等
- 数据归属和控制关系，如机构或自然人可采用所持私钥进行签名交易
- 个人信息，如身份，私钥等管理
- 出现频度，如总交易数，交易频率，分时的交易数等
- 大容量数据，如合同PDF，图片，商业大数据等

针对以上的几个技术事实以及数据要素，把区块链和传统机构的IT系统进行对比：

1: 在区块链上，用户的私钥只要不泄露，资产的交易就是安全的，用户牢牢把握对资产的控制权。这也是区块链技术的亮点之一。

同时，极客们认为在”中心化“机构里，资产的控制权是在托管资产的机构，机构背着用户在数据库动动帐就把用户的资产挪用了，这样就是不安全的（法律法规不健全和监管不严密的领域确实有这种风险）。

从另一个角度看，在区块链上，只要用户的私钥泄漏或丢失，对应的资产就失去了控制，比特币和以太坊这样的链上并没有密钥找回和重置等途径，对普通用户来说很容易成为一场灾难。而机构一般会在用户提供证明身份的资料后，实施密码的找回或重置。

另外，在区块链和“机构”结合的场景，比如虚拟资产交易所，通常用户要把自己的资产托管到交易所，并不是用自己的私钥来控制资产，交易所出现被攻击，系统故障、违规运作等问题，资产将会荡然无存，这里的安全就得由交易所背书了，和区块链关系已经不大。

2: 区块链上交易明细是全网公开的，所有人都可以看到，但由于区块链上的用户匿名性，所以通常认为没有损害个人隐私，极客们通常会采用多地址等方法来隐藏身份和混淆交易行为，中本聪本人至今还没有被发现。匿名性本身也是回避监管的一种手段。

但如果一个人“不小心”公布了他的链上地址（还不是私钥），他的一切交易行为、交易相关人都可以被追踪，被核算，交易次数和规律也可以计数。从隐私角度来说，也是很可怕的事情。

公网上的各种区块链浏览器如http://blockchain.info/zh-cn，https://etherscan.io/可以查询历史上所有的区块，交易方，金额等，并可以进行各种商业分析，相当一目了然。

与此对应的是，机构除了给监管报送和与合作伙伴之间进行必要的交互之外，所有的交易明细都不会公开，交易明细本身是一种隐私。正如我们银行帐户里的转账流水，是不可能随便给其他人看的。

另外，在典型区块链网络上，大容量数据（图片，合同，商业大数据）的共享一般用"P2P分布式存储"方案解决，数据分片加密存储在类似IPFS这样的平台上，在经济性、便利性和安全性之间取得了平衡。而金融机构的做法是数据根本就"不出机房",仅对受控的相关方进行授权，允许对指定数据的访问。在关注数据价值的商业领域，把数据copy给了别人，也就意味着失去了对数据的控制权。

对比一览：

| 数据要素 | 区块链 | 机构 | | ——————————- | ——————- | ——————- | | 所代表的价值，如资产数，金额，余额等 | 公开 | 相关方共享可加密保护 | | 数据归属和控制关系，如机构或自然人，采用私钥进行签名交易 | 保护 | 保护 | | 个人信息，如身份，私钥等管理 | 匿名不可监管不可重置/找回 | 不匿名可监管可重置/找回 | | 出现频度，如总交易数，交易频率，分时的交易数等 | 公开 | 相关方共享可加密保护 | | 大容量数据，如合同PDF，图片，商业大数据等 | 公网分布式存储 | 独占或相关方共享可加密保护 |

可见：崇尚数据共享的区块链上，"动账"控制被认为是安全的（前提是私钥不丢失），但是数据的"读"权限并没有默认被保护，同时，区块链上的私钥如何得到妥善的保管还是一个很有挑战性的课题。

常见的误区是，只要用了区块链，数据就是"安全"的，然后就可以进行和AI，大数据和区块链的结合，或者放心大胆的开展各种业务，用户对自己的数据也拥有了绝对的控制权，隐私得到了保护，仿佛不需要再做什么。

区块链的数据安全是个辩证的问题，是有前提的，机构使用区块链技术，结合原有的数据管理策略，对数据的安全需要达成读写保护的基本目标：

1： 可见性安全： 帐户信息和交易明细数据可控性共享，只发送给交易相关的机构，且能对隐私进行精细的保护。

2： 动账安全：机构或用户使用私钥控制自己的资产，有私钥重置或找回的途径，在用户私钥丢失的时候可以快速介入保护用户财产（如冻结），在有严格的界定（司法界定，全链共识等）后，可以找回用户丢失的资产。

在安全的设计上，机构应使用改进的联盟链，采用立体保护措施：

1：应用权限控制，实现接口级的读写控制。

2：准入控制，做为联盟链，只有被允许的成员机构才能加入，拥有全部或部分的数据访问和交易权限

3：节点间通信，存储层面上的加密。

4：主机和基础网络防攻击防渗透。

5：采用零知识，同态，群签名等算法保护用户隐私，隐藏用户的身份和交易的同时，保障效率且让监管可以接入

6：采用点对点通信或分片，分通道的隔离，使一部分和记账无关的商业信息不进入网络。最彻底的隔离其实是物理隔离。

7：将需要参与共识和需要共享的核心数据，和其他的数据如合同明文、批量大数据、隐私、个人信息分离，数据脱敏后再上链，或者将数据摘要上链，必要时再授权访问明文等其他数据。

安全是个很大的话题，这里仅浅谈"数据安全"的内在逻辑，要用到区块链的业务首先要对数据的敏感性和控制权进行分析和定义，根据需要采用不同的技术进行保护，怎么做没有定数，看场景下菜。

在此之外，还有共识体系的安全，通过共识防欺诈，防失效，经济模型上的安全，和业务密切相关，还有可能需要考虑到效率、体验、功能等其他维度的诉求，在安全方面进行可接受的妥协，就需要另外的文章来展开阐述了。

9.3 技术分析

9.3.1 WIKI 联盟链的权限体系

作者：fisco-dev

区块链从诞生开始，通常被认为是个“开放”、“自由”的世界，可以自由加入、自由交易、自由检索、自由退出，“权限”和“控制”相关的理念讲的是比较少的。

比特币和以太坊这两个典型的公链平台，虽然整个生态非常庞大，但在线上系统里并没有很明显的角色划分，大家都可以采用自己的区块链客户端或钱包来参与交易，也都可以成为矿工（能不能挖到矿就另说了），资产的归属和交易主要通过公私钥机制来控制，掌握了私钥就可以操作这个用户名下所有资产，也可以结合多签名等机制，对资产进行更精细的控制。

在准入许可的联盟链里，做为一个企业级的软件体系，对参与到链的活动的人也需要精细管理。原因如下：

1. 联盟链承载的业务复杂度可能超越数字资产转让，即使是同一个商业场景里的不同业务流程，能参与和应该参与的人也可能不一样；
2. 基于商业的上的隐私和安全考虑，要求对不同的人能做的操作、和能访问的范围进行区别对待；
3. 企业级生产环境对稳定性要求很高，运维升级操作特别谨慎，根据DO分离的基本思想，开发和运维的活动需要分开，再进一步：参与交易的，负责运营的，需要各司其职；
4. 可能有监管、委员会、治理人员等特殊身份存在，需要为这些特殊身份定制不一样的操作能力，这些能力不能赋予普通用户使用。

以上是必要性。简单总结下目前的做法是：

在支持智能合约的区块链平台里，可以控制某个用户以下行为:

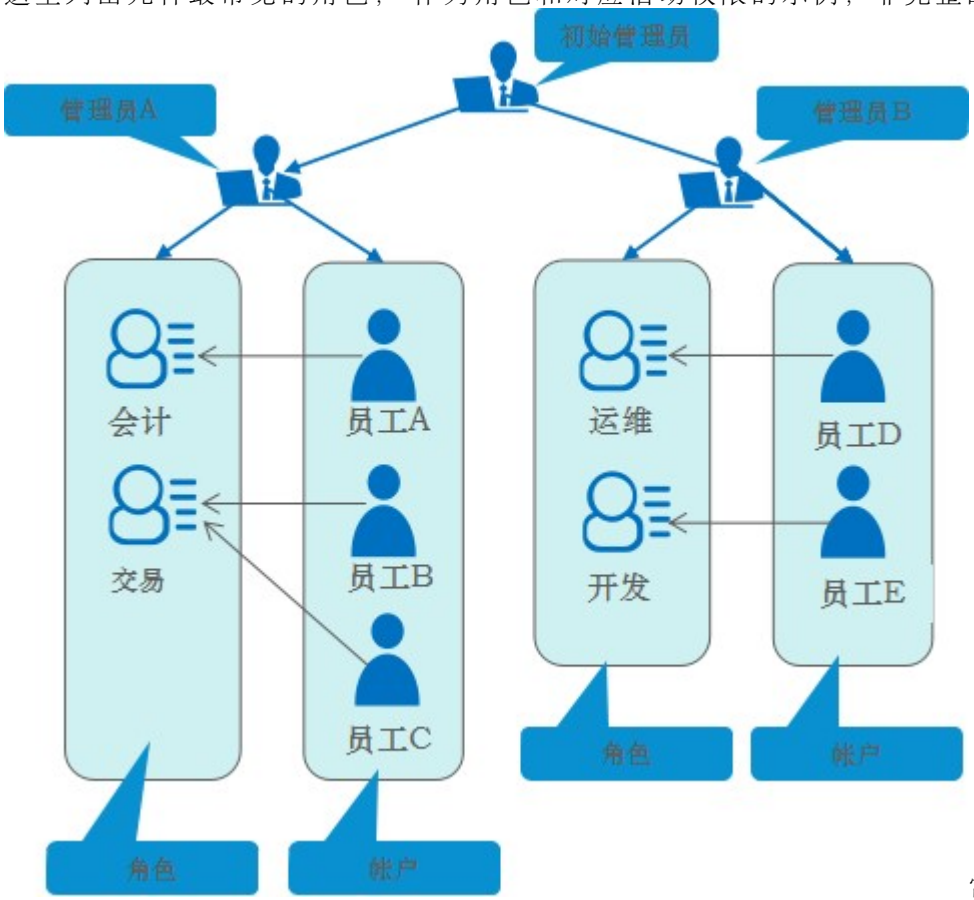
1. 能否部署合约。杜绝没有被审核过的合约发布到链上和被执行。
2. 能否调用某个合约。合约代表了一些系统能力或业务流程，如果不能调用某个合约的某个接口，那么这个用户就无法实现系统配置，系统控制，或者参与智能合约实现的业务交易流程。

————————以下是详细解析的分割线，有兴趣可以往下————————————

角色和权限总体来说是和场景强相关的，目前也没有非常标准化的实现。做为业务规则制定者和系统建设者，在做系统分析，需求分析时，就有必要对场景进行详细的分解，采用合适的设计模式，搞清楚系统的5W1H，在这个基础上进行角色和权限的规划。

- **WHO**： 都会有哪些人来使用系统，如普通用户、柜员、收单机构、发卡机构、清算机构、监管者等；
- **WHY**： 每个人来这个系统的目的是什么。如用户就是为了消费，柜员服务于用户，监管检视数据和进行必要的控制，机构开展业务的同时要维护系统稳定；
- **WHAT**： 每个人都会操作什么系统、资产、工具。如用户要用APP,看余额、要花钱、要提现，机构要有管理工具、运维工具、要开发和发布执行交易的智能合约、要清结算拿到钱款，监管机构要有大数据平台、报表或者监管工具；
- **WHEN**： 这些人什么时候会有一些动作。比如用户每天都会消费，机构的运维人员会定期升级和维护程序，不定期处理告警，清算机构在场切日切后要做清算，监管者可以选择是否实时介入交易，或者仅定期巡检，或在有问题时强势介入；
- **WHERE**： 活动场合或场所。用户通过APP随时随地在合作商户消费，APP通过机构提供的接口来发送交易，程序员在开发网开发编译，运维在IDC或ECC直接对区块链进行发布和调整，运营配置人员通过管理台或者管理工具直接或者通过远程接口进行运营数据的配置，监管机构远程调控，通过接口或者发个邮件过来指定控制措施；
- **HOW**：根据以上的5W，要梳理出都有哪些控制点、功能接口、管理流程，在每个场景，怎么一步一步的，通过什么工具什么操作指令，或者设定哪些联机检查规则去做到权限控制。形成详细设计，理清是否有技术或者其他障碍，然后分配到各模块去进行开发、部署。

这里列出几种最常见的角色，作为角色和对应活动权限的示例，非完整的角色体系定义。



常见角色

- **交易操作员**:使用平台进行商业交易操作的人员，交易操作员发起业务的交易，如发行、售卖、转账、消费、付款等（根据不同的业务场景，有不同的操作），并查询交易执行结果。不同的交易操作员，也可以再细分角色。
- **平台运营管理者**:通常由联盟链管理委员会，或公选出来的运营人员承担，负责审核、修改、删除链上的节点和帐号相关资料。如设定某一个联盟链的信息和进行初始化部署，组织各机构加入联盟链，为各机构分配对应的机构管理员和交易员权限，管理链上应用的生命周期等。平台运营管理者对联盟链的日常运营负责，一般不直接参与链上业务交易。
- **应用开发者**:使用平台提供的源代码，SDK、接口以及平台环境，开发面向用户的商业应用。应用开发将可发行的软件提交应用到平台的应用仓库，包括智能合约、APP等。在DO分离的管理模式中，开发者一般不直接对联盟链直接操作，而由运维管理者进行软件发布、参数配置等操作。为了跟踪应用的使用情况，开发者有查看相关应用的统计数据的权限。
- **运维管理者**:实施联盟链的部署和运维活动的人员。通常为各机构的运维团队承担，或者由平台运营管理者统一组织运维管理，运维人员负责发布和管理应用，管理区块链的节点物理资源，启停节点服务，修改本地节点的系统配置参数，一般不会参与业务交易。
- **监管方**：制定业务规范，审查业务数据，监督管理联盟链的运行状态，维护服务的合法、稳健运行。监管方一般不参与联盟链的日常运作管理，根据业务场景需要，监管可选择参与到业务交易中。 在不同业务的联盟链上，具体的角色定义和角色的层级关系也会有不同，可由联盟链的运营者进行定义和二次开发。

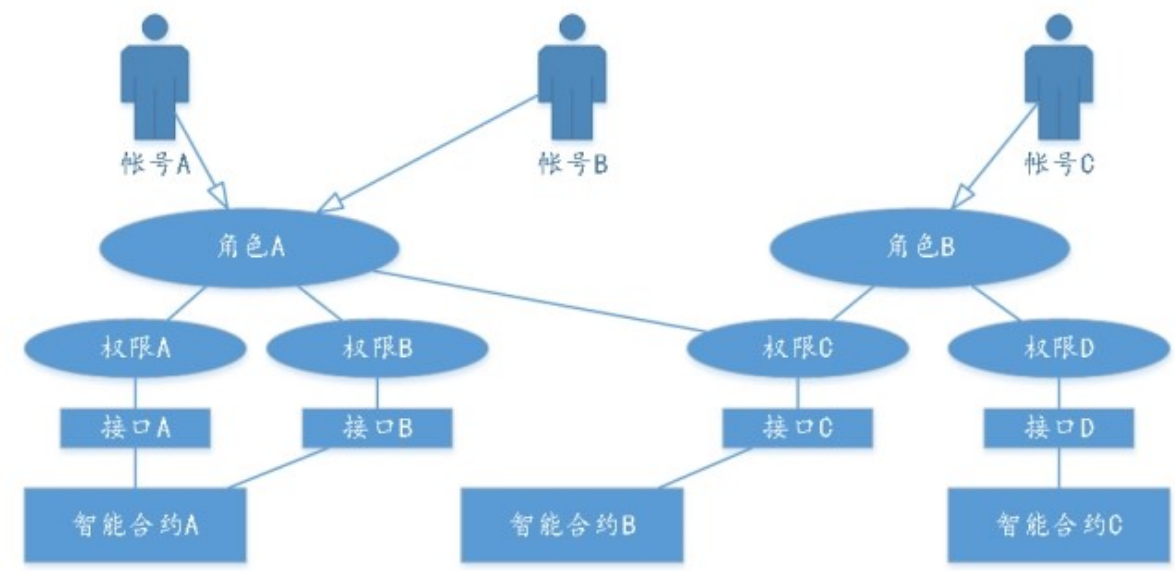
在角色和帐号模型中，帐号代表一个人或者一个IT系统，使用一个唯一的公私钥对，帐号的持有者要妥善保管私钥，不对外公开，同时把公钥公布出去，用于标识自己，当某个帐号向系统发出请求时，采用私钥进行签名，接受数据的对手方会根据发送者的公钥进行验签。

一个帐号必须对应到且只能对应到一个角色，如某个帐号的角色是“交易员”，则这个帐号不应同时做为“开发者”或“监管者”，避免“**又做运动员又做裁判员**”的可能性，又如帐号已经是“开发者”，则他不应同时是“运维人员”，避免违背“DO分离”的原则，同时他也不应成为“交易员”或“监管者”。

如果某个机构同时兼顾多个角色的职责，这个机构的人员需要进行交易，又可以做为“运维人员”参与系

统维护，那么应给该机构分配多个帐号，每个帐号的私钥掌握在不同的人员手里，每个人员只负责一种角色的工作，以规范操作流程。

同时，多个帐号可以对应到同一个角色，如一个机构可以有多个交易员帐号，都拥有进行交易的权限，每个交易员在链上的交易都可以根据签名和公钥信息对应到某个特定的交易员，便于机构的交易行为管理和交易结果追溯。同理，可以有多个运维人员、多个运营人员等。**面向角色的权限控制，以及合理的帐号分配管理，使对应到不同角色的帐号职责清晰，在链上的活动行为可控，容易追溯。**



账号、角色及权限a）角色拥有多个权限的集合。b）一个帐号只能对应到一个角色。c）一个权限对应到某智能合约的一个接口。d）帐号只能调用被授权过的智能合约的一个到多个接口。

角色和权限的控制可以实现的非常复杂，如很多MIS系统或者大型的商业化系统里，给人分配角色、分派权限的界面看起来就已经是琳琅满目，可以做出无限深度的关系树，然后有各种交叉错综复杂的权限。

做为一个通用的联盟链平台，在这方面倾向于保持KISS，提供最底层的基础控制接口，把面向业务的角色和权限管理交给业务开发方去定制。

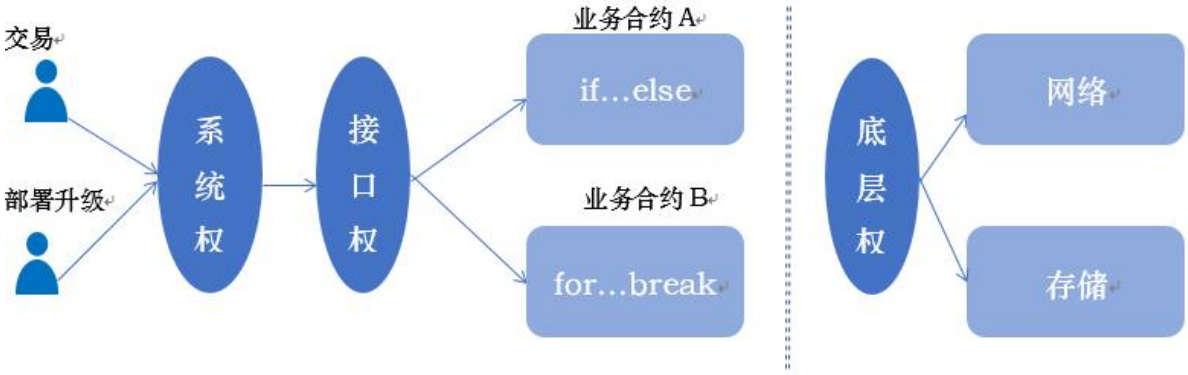
目前抽象出来的权限控制能力有两种模型：

1. **系统级权限模型**：一个角色是否能部署新合约，是否能发起对已有合约的调用，属于系统权限模型。在该角色发起请求后和部署合约或调用合约之前，系统进行判断和控制，拒绝越权的操作。如，如需限制所有的交易员只能向合约发送交易而不能部署新合约，则可以通过系统权限控制实现。
2. **接口权限模型**：作为支持智能合约的区块链平台，平台的诸多功能都是通过智能合约实现的，包括系统配置、权限配置、业务交易等、接口权限模型实现了针对智能合约接口级别的权限控制。当一个新的合约部署生效后，管理员可赋予某个角色调用该合约部分或全部接口的权限。如一个做为会计角色的交易员可以调用某个合约的查询、对账接口，另一个做为实时交易角色的交易员可以调用某个商业合约的消费、转账接口。

有了这两个基础的控制埋点之后，可以针对不同的用户和角色进行配置，精细的限制他们的行为能力。比如发卡行只能调用发卡合约，发卡行不牵涉用户和商户之间的消费流程，收单行只能调用收单的接口，负责用户的消费流程，不能代替用户去充值等等。参与者的多中心化、商业互补性和安全性得以保障。

基于智能合约实现的业务逻辑本身也可以做大量在业务层面上的的权限控制，智能合约能感知是谁（交易的From）在调用自己的某个接口，基于图灵完备的特性，像普通业务逻辑一样去写if...else就可以进行多种多样的控制了。比如控制某个用户或某个机构只能访问和自己相关的订单，不能访问其他人的等等。

更底层一点的权限控制，可以面向区块链的基础数据管理、网络层、存储层起作用，比如，控制某些节点或某些帐号是否能同步区块数据，控制是否能向其他节点广播消息，还是只能默默的从网络接受数据。总而言之，控制无极限，只要在安全和业务上有足够的必要性。



权限控制

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

9.3.2 浅谈以太坊智能合约的设计模式与升级方法

作者：fisco-dev

- 浅谈以太坊智能合约的设计模式与升级方法
  - *1.* 最佳实践
  - *2.* 实用设计案例
    - *2.1* 控制器合约与数据合约*: 1->1*
    - *2.2* 控制器合约与数据合约*: 1->N*
    - *2.3* 控制器合约与数据合约*: N->1*
    - *2.4* 控制器合约与数据合约*: N->N*
    - *2.5* 总结
  - *3.* 升级
    - *3.1* 控制器合约升级，数据合约不升级
    - *3.2* 控制器合约不升级，数据合约升级
    - *3.3* 控制器合约升级，数据合约升级
  - *4.* 数据迁移
    - *4.1* 硬编码迁移法
    - *4.2* 硬拷贝迁移法
    - *4.3* 默克尔树迁移法

以太坊EVM是当前区块链行业应用最为广泛的虚拟机。其所支持的智能合约语言是图灵完备的。该语言支持各种基础类型（Booleans，Integers，Address，String，Enum，Address等)、复杂类型（Struct，Mapping，Array等）、复杂的表达式和控制结构及接口继承等面向对象的特性。

正是由于强大的智能合约语言，原本在真实世界中的复杂商业逻辑和应用都能在区块链上轻松实现。然而需要注意的是，尽管公有链可以实现合理的GAS机制自我保护，联盟链可以用其他机制替代GAS的计算及代币化来保障EVM沙盒安全，但由于区块链运行机制的原因，智能合约的运行即使是异常运行都会在所有区块链节点上独立重复运行。因此，无论是在公有链还是联盟链运行智能合约都是非常昂贵（运算资源、存储资源）的操作。

另外，智能合约与传统应用程序有一个不同的地方在于智能合约一经发布于区块链上就无法篡改，即使智能合约中有Bug需要修复或者业务逻辑变更，它也不能直接在原有的合约上直接修改再重新发布。因此在设计之初就需要结合业务场景考虑合理的升级机制。

总而言之，智能合约实现上要达到的目标是：**完备的业务功能、精悍的代码逻辑、良好的模块抽象、清晰的合约结构、合理的安全检查、完备的升级方案**。

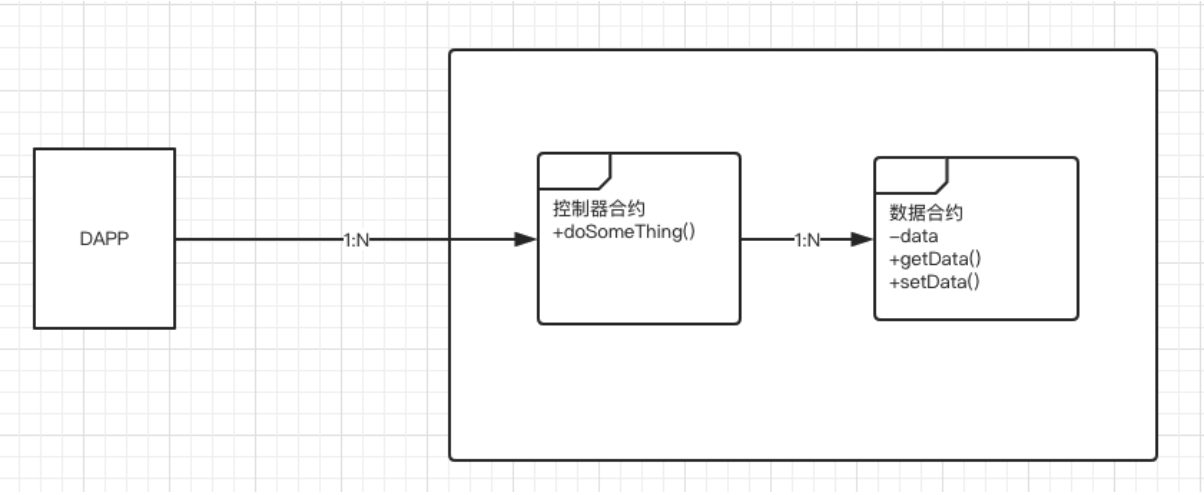
智能合约的生命周期主要有设计、开发、部署、运行、升级、销毁。在下文中主要是基于目标在设计阶段、升级阶段的一些梳理总结。

1. 最佳实践

从业务视角来看，智能合约只需要做两件事，其一是如何定义数据的结构和读写方式，其二是如何处理数据并对外提供服务接口。

为了更好的做好模块抽象和合约结构分层，将这两件事分开，既是将业务控制逻辑和数据从合约代码层面就做好分离，这样的处理在复杂业务逻辑场景中经过实践是当前被认为最佳的模式。

这个模式简称为CD（Controller-Data）模式。将合约分为两类：控制器合约（Controller Contract）与数据合约（Data Contract）。



CD模式

控制器合约通过访问数据合约获得数据，并对数据做逻辑处理，然后写回数据合约。它专注于对数据的逻辑处理和对外提供服务。根据处理逻辑的不同，常见的有命名空间控制器合约、代理控制器合约、业务控制器合约、工厂控制器合约等。一般情况下，控制器合约不需要存储任何数据，它完全依赖外部的输入来决定对数据合约的访问。特殊情况下，控制器合约可以存储某个固定的数据合约的地址或者命名空间（通过命名空间在运行时获得合约地址）。

数据合约专注于数据结构定义与所存储数据的读写裸接口。为了达到数据统一访问管理和数据访问权限控制的目的，最好是将数据读写接口只暴露给对应的控制器合约。禁止其他方式的读写访问。

基于这个模式，遵循从上至下的分析方式，从对外提供的服务接口开始设计各类控制器合约，再逐步过渡到服务接口所需要的数据模型和存储方式，进而设计各类数据合约，可以较为快速的完成合约架构的设计。

2. 实用设计案例

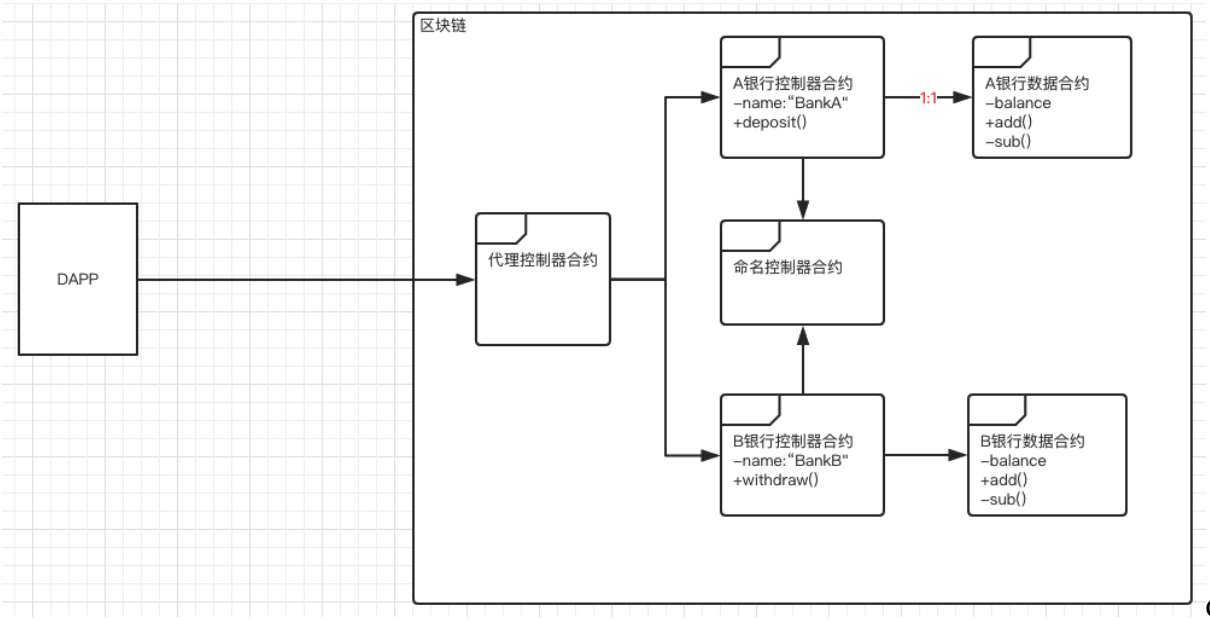
在CD模式下，根据控制器合约与数据合约之间的操作关系，从逻辑上归结为四类：

1. 控制器合约与数据合约 1->1
2. 控制器合约与数据合约 1->N
3. 控制器合约与数据合约 N->1
4. 控制器合约与数据合约 N->N

假设一个业务场景：将全国所有银行的业务和信息上链。

2.1 控制器合约与数据合约: 1->1

假设全国只有两家银行，A银行和B银行。A银行只有存款业务，B银行只有取款业务。一种可能的设计是这样的:



CD模式

代理控制器合约：面向Dapp，是所有业务合约的入口，提供命名空间服务，提供了命名空间到合约地址的映射。使得Dapp对链上合约升级导致的地址变更无感知。例如，Dapp对A银行的存款请求只需要（"BankA"，deposit，args) 即可。对B银行的取款请求只需要（"BankB"，withdraw，args）即可。代理器控制合约实现上应该是区块链底层内置的、固化的，或者是业务上极少变更的。Dapp在业务运行之前已经明确知道代理控制器合约的地址。

命名控制器合约：面向链上合约，提供命名空间服务，提供了命名空间到合约地址的映射。使得链上合约可以在运行时根据命名获得实际的合约地址。例如，A银行控制器合约向命名控制器合约请求（"BankA-Data"），可以获得A银行数据合约地址，使得A银行控制器合约可以在运行时访问A银行数据合约。它与代理控制器合约的主要不同在于服务对象的不同，代理控制器合约面向Dapp，命名控制器合约面向链上合约。另外，命名控制器合约包含有版本控制的设计（下文第3.2节介绍），可以根据需要配合灰度策略的实施。

A银行控制器合约：提供了存款服务接口deposit。部署初始化时已经明确知道自己的身份"BankA"。并且可以在运行时通过命名控制合约获得"BankA"的数据合约"BankA-Data"的地址。

A银行数据合约：保存了A银行的当前余额。提供add和sub接口给A银行控制器合约来更新余额信息。

B银行控制器合约：提供了存款服务接口withdraw。部署初始化时已经明确知道自己的身份"BankB"。并且可以在运行时通过命名控制合约获得"BankB"的数据合约"BankB-Data"的地址。

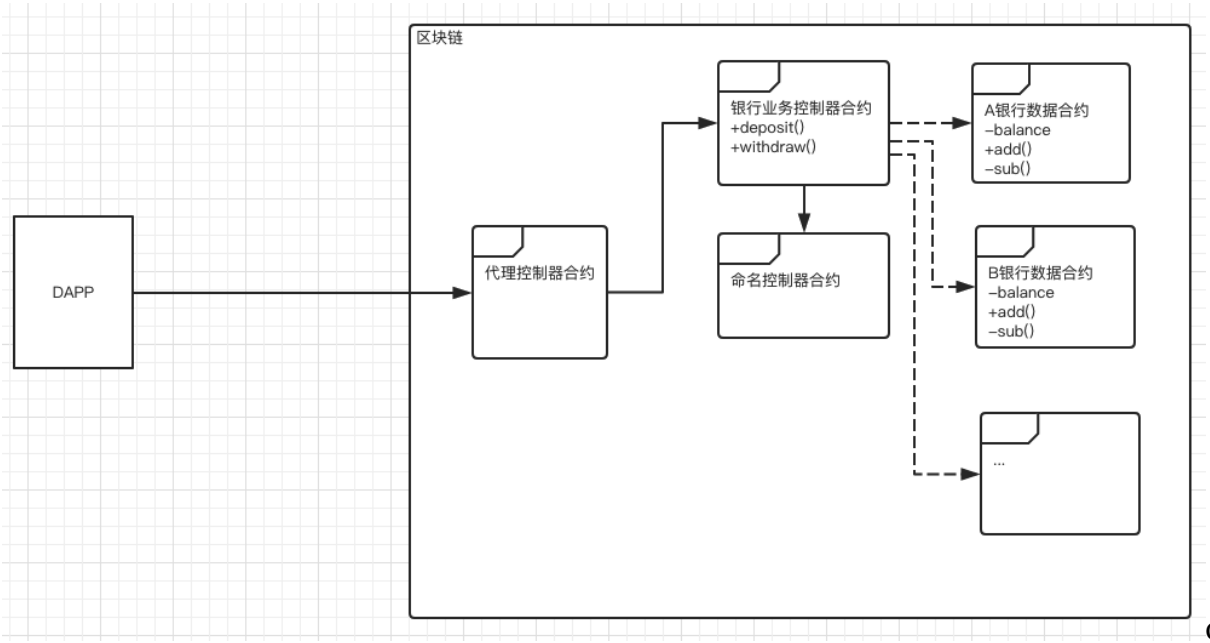
B银行数据合约：保存了B银行的当前余额。提供add和sub接口给B银行控制器合约来更新余额信息。

对A银行的存款请求的流程是这样的:

1. Dapp指定代理控制器合约地址，请求存款交易（"BankA"，deposit，money)
2. 代理控制器合约，运行时得到"BankA"对应的A银行控制器合约地址，并向A银行控制器合约请求存款交易（deposit，money)
3. A银行控制器合约的deposit接口向命名控制器合约请求A银行的数据合约"BankA-Data"的地址，并访问到A银行数据合约的数据，然后执行一些存款业务逻辑。返回结果。
4. 依次返回结果到Dapp。

2.2 控制器合约与数据合约: 1->N

假设全国有N家银行，所有银行都有存款业务和取款业务，并且业务流程都是一样的。一种可能的设计是这样的:



CD模式

这个设计与上面的2.1不一样的地方在于，将存款服务接口和取款接口都集中归结到银行业务控制器合约里面了。这意味着任何银行的存款和取款业务都由银行业务控制器合约来统一处理，处理逻辑上不再区分是A银行还是B银行，只是在数据访问的时候需要根据入参的不同来决定访问不同的银行数据合约。

还有，于2.1相比，对于Dapp而言，它发出请求的时候只需要将请求发往固定的"Bank"就可以了，不用具体关心某个银行。

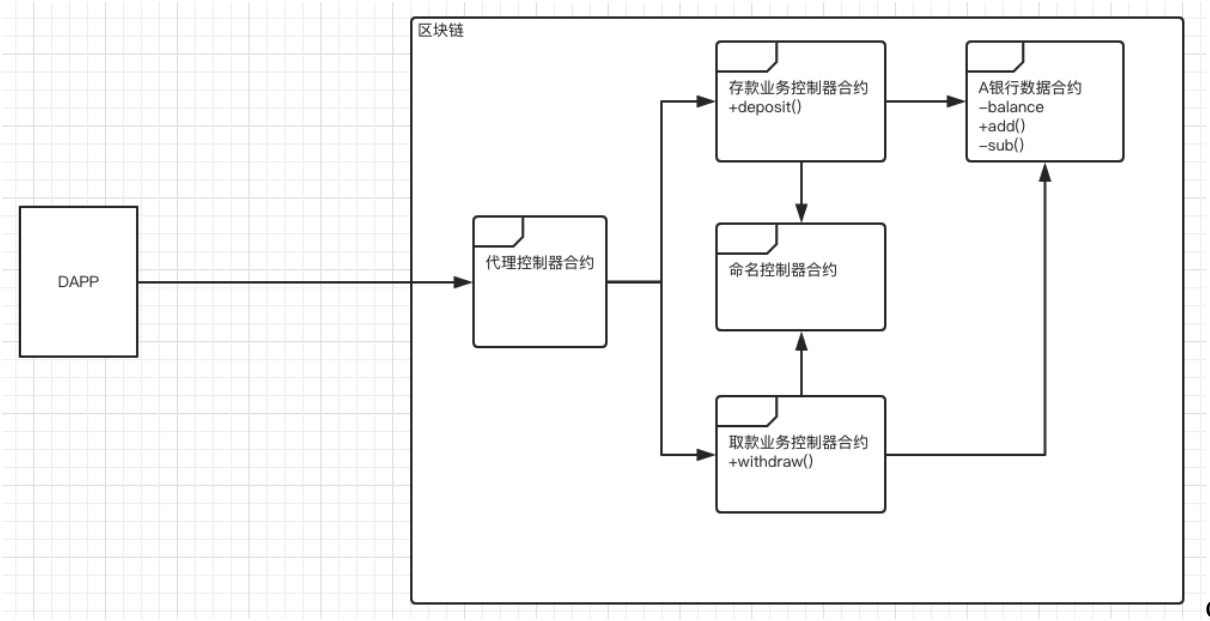
另外，由于银行有很多个，并且它们的存储结构都是一样的，因此可以设计一个银行数据合约的工厂控制器合约，来负责对新的数据合约的生成实现模板化。

对A银行的存款请求的流程是这样的:

1. Dapp指定代理控制器合约地址，请求存款交易（"Bank"，deposit，"BankA"，money)
2. 代理控制器合约，运行时得到"Bank"对应的银行业务控制器合约地址，向银行业务控制器合约请求存款交易（deposit，"BankA"，money)
3. 银行业务控制器合约的deposit接口向命名控制器合约请求A银行的数据合约"BankA-Data"的地址，并访问到A银行数据合约的数据，然后执行一些存款业务逻辑。返回结果。
4. 依次返回结果到Dapp。

2.3 控制器合约与数据合约: N->1

假设全国有N家银行，所有银行都有存款业务和取款业务，并且业务流程都是一样的，但是由于业务逻辑较为复杂，出于模块化维护的需要，需要将存款业务和取款业务做分拆。一种可能的设计是这样的:



CD模式

这个设计与上面的2.2不一样的地方在于，将存款服务接口和取款接口拆分到了不同的业务控制器合约里面了。这意味着不同的业务逻辑从模块上做了清晰的切分。对于Dapp而言，它发出请求的时候需要明确指向所对应的业务接口。

对A银行的存款请求的流程是这样的:

1. Dapp指定代理控制器合约地址，请求存款交易（“deposit”，”BankA“，money)
2. 代理控制器合约，运行时得到”deposit”对应的存款业务控制器合约地址，向存款业务控制器合约请求存款交易（”BankA“，money)
3. 存款业务控制器合约的deposit接口向命名控制器合约请求A银行的数据合约“BankA-Data”的地址，并访问到A银行数据合约的数据，然后执行一些存款业务逻辑。返回结果。
4. 依次返回结果到Dapp。

2.4 控制器合约与数据合约: N->N

此类情况可以拆解为上面三种情况的组合。不再赘述。

2.5 总结

从Dapp视角考虑，可以总结如下:

| CD模式 | 特点 | | —- | :—-: | | 1->1 | 面向业务对象 | | 1->N | 面向业务流程 | | N->1 | 面向业务接口 | | N->N | / |

3. 升级

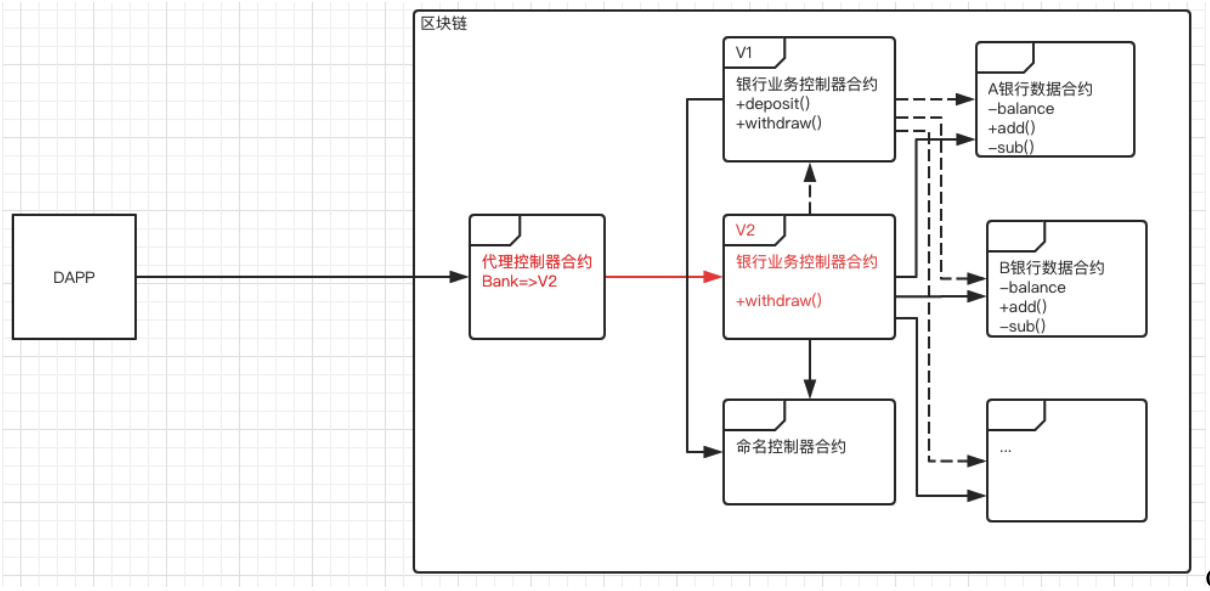
在CD模式下，在业务逻辑变更需要升级合约的情况下，根据控制器合约与数据合约的升级关系来划分，可以归纳为以下三种情况:

| 控制器合约 | 数据合约 | | —— | :–: | | 升级 | 不升级 | | 不升级 | 升级 | | 升级 | 升级 |

在升级过程中，还需要考虑是全量升级还是灰度升级？如果是灰度升级，灰度策略是怎么样的？另外，在多链场景和单链场景、跨链场景，升级过程是否有不同？多链场景的灰度策略如何考虑？新旧版本数据能否共存？如果需要数据迁移，如何做到无缝迁移?

下面以最为常见的1->N 场景来介绍不同的升级情况。

3.1 控制器合约升级，数据合约不升级



CD模式

如上图所示，银行业务控制器合约从V1升级到V2，而其他的合约和接口都是不需要更新的，假设V2版本相对V1版本只是升级withdraw这个接口。

此时，V2版本的银行业务控制器合约需要做的事情是:

1. 继承V1版本的银行业务控制器合约。
2. 增加一个指向V1版本的链上合约地址的成员变量。
3. 增加一个withdraw开关接口，允许外部账户通过普通交易来操作V2版本合约的启停灰度策略。
4. 重载withdraw接口。升级对应的接口逻辑。并且在业务逻辑真正开始执行之前，自定义实现灰度策略（譬如灰度特定用户，或者一定比例用户或者其他策略）。并且需要注意的是在打开灰度开关的情况下，如果请求没有命中灰度策略，则直接透传参数调用V1版本的合约接口，V2版本的withdraw接口不做任何额外工作。

完成V2版本的合约工作之后，即可发布一个普通交易，交易中的逻辑是，先部署V2版本的银行业务控制器合约，再将其地址更新到代理控制器合约中，使得将“Bank”映射到V2版本的合约地址上。这样控制器合约即升级完成。

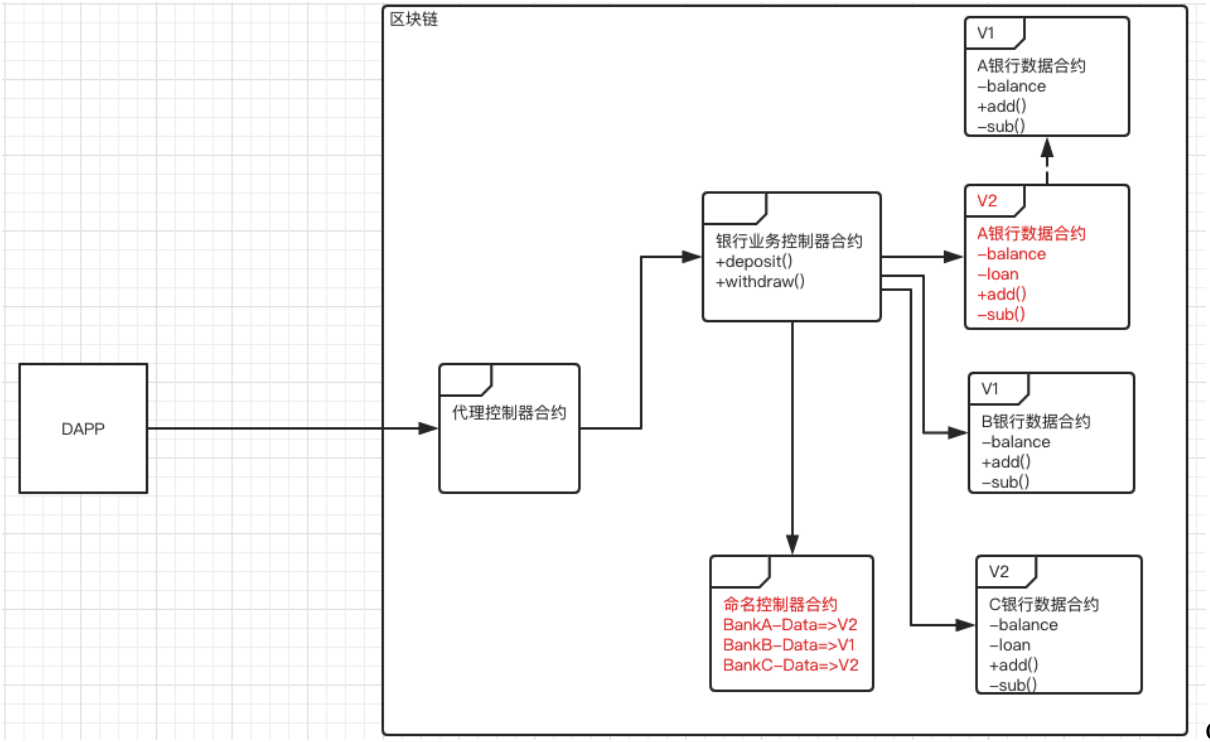
如果需要回退版本，只需要发布一个普通交易，将代理控制器合约的“Bank”映射到V1版本的合约地址上即可。

以上是单链场景的升级方法。如果是多链场景，只需根据业务的需要来判断链与链之间的灰度策略，重复单链场景的升级即可。如果是跨链场景，需要根据跨链两端的具体情况来制定升级方法。

而对于业务发起端的Dapp而言，它是无任何感知的。它对A银行的存款请求与2.2中完全一样。依旧是以（“Bank”，deposit，”BankA“，money)来发出请求。

总结而言，灰度策略定义在新版本的控制器合约中，数据无需迁移，业务无感知，无需停止服务。无缝升级。

3.2 控制器合约不升级，数据合约升级



CD模式

如上图所示，A银行数据合约从V1升级到V2。而其他的合约和接口都是不需要更新，假设V2版本相对V1版本只是增加新的数据字段loan，并假设银行业务控制器合约原本就能支持到V2版本的A银行数据合约（如果是银行业务控制器合约也需要升级则是3.3节的场景，这里不做描述）。

此时，V2版本的A银行数据合约需要做的事情是：

1. 继承V1版本的银行数据合约。
2. 增加一个新字段loan。并实现loan相关的数据接口。

需要注意的是，命名控制器合约有如下重要的设计：

1. 命名控制器合约是通过访问命名数据合约来存储和访问数据的（为了方便描述，图中并没有画出来），因此命令控制器合约是可以参考3.1节的方法来升级的。
2. 命名数据合约保存了name=>mapping(version=>address)的映射表。
3. 命名数据合约保存了name=>当前有效的version的映射表。
4. 命名控制器合约提供了对命名数据合约的name进行遍历的接口。
5. 命名控制器合约提供了对命名数据合约的映射表的变更接口。

因此，完成V2版本的数据合约之后，即可发布一个普通交易，交易中的逻辑是，先部署V2版本的A银行数据合约，并完成V1版本数据合约到V2版本数据合约的数据迁移（数据迁移方法第4节会描述），接着将V2版本数据合约地址注册到命名控制器合约，并更新BankA-Data所映射的当前有效verison=V2。此时已完成了A银行数据合约的V2版本升级。

如果需要回退版本，只需要发布一个普通交易，将命名控制器合约的BankA-Data所映射的当前有效verison=V1即可。

而对于业务发起端的Dapp而言，它是无任何感知的。它对A银行的存款请求与2.2中完全一样。依旧是以（“Bank”，deposit，”BankA“，money)来发出请求。

对于B银行而言，因为B银行数据合约并没有执行升级，所以与它相关的业务请求依然是访问的B银行数据合约的V1版本。所以，对于历史旧版本的数据合约，可以根据业务的需要来判断是否需要对历史旧版本执行升级。有些特殊场景下，需要对所有的历史旧版本数据合约进行升级，这时可以利用命名控制

器合约的遍历功能，对所有数据合约进行类似的升级。而对于新加入的C银行，它可以直接使用最新版本V2的数据合约，按照正常流程完成部署与注册，无任何额外操作。

正是由于有了命名控制器合约的版本控制逻辑，可以使得即使存在新老版本数据合约并存的情况下，业务控制器类合约依然能正常运行。而对于由于业务的发展和不断的版本升级，会带来命名数据合约的存储量膨胀，导致可能出现的性能下降的情况，依然可以套用本节所述的数据迁移与升级的方法来解决。

以上是单链场景的升级方法。如果是多链场景，只需根据业务的需要来判断链与链之间的灰度策略，重复单链场景的升级即可。如果是跨链场景，需要根据跨链两端的具体情况来制定升级方法。

总结而言，得益于命名控制器合约的版本控制设计，灰度策略可以交给业务方非常自由地选择，业务无感知，无需停止服务。无缝升级。

3.3 控制器合约升级，数据合约升级

此种情况下，实质是3.1与3.2 两种情况的混搭。

因此根据具体情况，拆解成参考3.1和3.2场景方法来执行即可。

4. 数据迁移

如3.2节所描述，在数据合约升级的场景，某些情况需要处理历史数据在新旧合约之间的迁移。迁移的方法有如下三种，各有特点。

4.1 硬编码迁移法

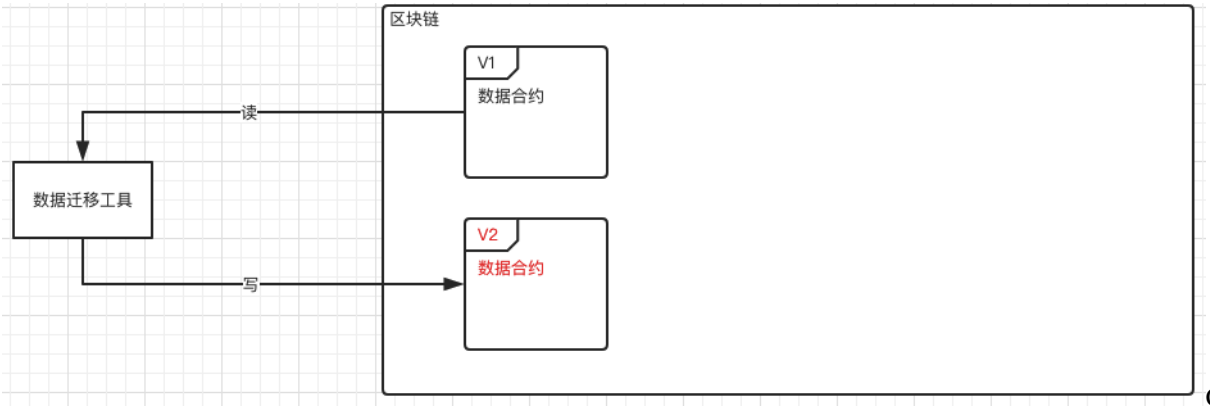
硬编码迁移法指的是，新版本的数据合约中保存一个指向旧版本数据合约的合约地址，新版本数据合约保存的是增量的数据内容。

这样相当于新版本合约保留了一份旧版本数据的指针，当新版本需要使用旧数据的时候，直接调用旧数据合约地址对应数据接口即可。这样，新旧版本数据合约可以并存，即使是在异常情况下，数据被误写到了旧版本合约上，它依然可以被新版本所访问到。

这个方法的优点是：新旧合约可以同时并存，不增加区块链存储压力，简单灵活，较强的升级容错能力。缺点：持续不断的版本升级会导致形成较长的链式逻辑关系，维护成本较高。

4.2 硬拷贝迁移法

硬拷贝迁移法指的是，新版本和旧版本之间切断逻辑关系，利用外部迁移工具，将旧版本数据逐步拷贝到链下，再从链下重新存储到新版本合约的过程。



CD模式

这个方法的优点是：无历史包袱。缺点是：大幅度增加区块链存储压力；数据迁移工具需要适配不同的数据合约，开发成本较高；迁移过程需要停止服务，否则容易出现脏数据；数据量大时，耗时长，操作复杂，容易出错，基本无法实操。

4.3 默克尔树迁移法

默克尔数迁移法要点如下：

1. 利用智能合约语言的面向对象的继承特性，使得新版本合约存储结构完全兼容旧版本合约存储结构。
2. 利用智能合约在区块链上的storage树原理，使得新版本合约的storeage树直接从旧版本合约上衍生。无需显式的迁移过程。
3. 利用区块链交易的原子性，使得新版本合约的部署、数据迁移、升级，原子完成。

这个方法拥有前面两个方法的所有优点，且简单高效，安全，实操性强。缺点：需要区块链底层功能特性的支持。

9.4 更多

9.4.1 FISCO-BCOS实践指引

本文作为一个概括性的实践指引, 包含了FISCO BCOS的环境搭建、特性简介、业务实践的介绍预览, 可以使开发者对FISCO BCOS有一个全局性的认识, 更完整信息可以参考Wiki。

环境搭建

本段落给出了FISCO BCOS多种环境搭建方式, 从快速体验到逐步搭建, 满足各类开发者的需求。

- 单节点环境指引开发者编译、搭建一个最简单的由一个节点组成的环境, 并进行合约的部署、调用; 同时，能够进行系统合约的部署, 对系统合约有一个简单的认识, 为搭建多节点链做准备。

1. 源码编译： 搭建FISCO BCOS的配置要求, 软件依赖安装, 源码获取, 源码编译介绍。
2. 创建创世节点(单节点环境搭建)：搭建只有一个节点的区块链环境, 创世节点是区块链中的第一个节点, 搭建多节点的链环境, 也需要从创世节点开始。
3. 部署调用合约：正确部署、运行sample合约。
4. 部署系统合约 : FISCO BCOS区块链为了满足准入控制、身份认证、配置管理、权限管理等需求，在网络启动之初，会部署一套功能强大、结构灵活且支持自定义扩展的智能合约，统称系统合约, 详细了解系统合约可以参考：系统合约介绍。

- 多节点组网

1. 多节点组网 : 搭建多个节点的链环境, 包括节点的加入、退出功能。
2. 机构证书准入：FISCO BCOS提供了证书准入的功能。在节点加入网络后，节点间是否能够通信，还可通过证书进行控制。在FISCO BCOS中，节点的证书代表了此节点属于某个机构。FISCO BCOS区块链中的管理者，可以通过配置机构的证书，控制相应证书的节点是否能够与其它节点通信。

- 一键快速安装部署 一键快速安装部署：为了能够让初学者快速体验, FISCO BCOS平台提供了快速安装和节点的快速部署工具, 开发者只需要非常简单的命令既可成功搭建多个节点的环境, 不再需要繁琐的配置。
- Docker节点部署Docker节点部署：FISCO BCOS同样提供了docker下的节点安装流程, 使开发者可以快速在docker环境下搭建、运行、体验。
- 物料包工具物料包工具: 使用本工具, 进行一些简单配置后, 可以创建区块链节点的安装包, 然后经过一些简单的步骤, 可以快速搭建生产可用的区块链环境。

特性简介

- 权限控制权限模型：与可自由加入退出、自由交易、自由检索的公有链相比，联盟链有准入许可、交易多样化、基于商业上隐私及安全考虑、高稳定性等要求。因此，联盟链在实践过程中需强调“权限”及“控制”的理念。 为体现“权限”及“控制”理念，FISCO BCOS平台基于系统级权限控制和接口级权限控制的思想，提出ARPI(Account—Role—Permission—Interface)权限控制模型。
- CNS(合约命名服务)CNS(Contract Name Service)：提供一种命令服务, 将合约接口调用映射为名称, 并且内置了合约的版本管理, 基于CNS开发者可以以更简单的方式调用合约接口, 可以忽略更多的繁琐流程, 而且基于CNS内置的合约版本管理机制对于合约的灰度升级会非常友好。
- AMOP(链上信使协议) AMOP(Advance Messages Onchain Protocol)：AMOP系统旨在为联盟链提供一个安全高效的消息信道，联盟链中的各个机构，只要部署了区块链节点，无论是共识节点还是观察节点，均可使用AMOP进行通讯。
- 应用于区块链的多节点并行拜占庭容错共识算法共识机制：FISCO BCOS的共识机制是在PFBT基础上进行改进, 结合区块链的特性, 增加了异常处理, 添加了并行处理的机制。
- 并行计算并行计算：区块链的系统特性决定，在区块链中增加节点，只会增强系统的容错性，增加参与者的授信背书等，而不会增加性能，只增加节点不能解决问题，这就需要通过架构上的调整来应对性能挑战，所以，我们提出了“并行计算，多链运行”的方案。并行多链的架构基本思路是在一个区块链网络里，存在多个分组，每个组是一个完整的区块链网络，有独立的软件模块，硬件资源，独立完成机构间共识，有独立的数据存储。

  完整的特性介绍可以参考： FISCO BCOS特性介绍

业务实践

本模块介绍FISCO BCOS客户端web3sk的使用, 并在此基础上给出了一个工业级的生产案例–存证sample。

web3sdk

web3sdk是FISCO BCOS的java客户端, 针对FISCO BCOS做了多项优化、改进, 添加了FISCO BCOS的多项特性。 下载地址 使用文档

存证sample

源码地址 FISCO BCOS是聚焦于金融级应用服务的区块链底层技术平台。在此基础上，FISCO BCOS团队结合区块链不可篡改、多方共识等特性，开发此sample以帮助开发者快速入门区块链存证应用开发。本sample建立了完整的存证、核证、取证业务模型，并允许司法机构参与到业务过程中实时见证。为后续的证据核实、纠纷解决、裁决送达提供了可信、可追溯、可证明的技术保障。

区块链应用系统开发TIPS

文档地址

9.4.2 FISCO BCOS“极简”Java应用开发入门

作者：fisco-dev

FISCO BCOS区块链平台部分是用于搭建多方参与的联盟链，业务开发时，可以结合智能合约和Java版的web3sdk（更多语言适配中），开发向区块链部署智能合约、发送交易、获得结果的业务模块。FISCO BCOS平台会提供相关的sdk、样例等。

本文试图介绍如何step by step的从头开发一个最简的Java业务模块，然后再基于这个模块，跑通流程，扩展更多的功能。

P.S.也可以直接参考存证业务样例, 这是一个带完整业务流程的应用样例。

前置条件

1.按照FISCO BCOS 的“2.2 安装说明”，搭建并跑起来至少一个节点，推荐搭建4个节点的链，体验整个部署流程。可使用一键搭链脚本、Docker、物料包等方式快速部署，本文不做详细阐述。如搭链有问题，可到社区里提问（参见页面里 “6.联系我们” ）。目前推荐基于FISCO BCOS 1.3以上的版本进行搭链，采用release的版本搭链更加稳定，如有任何问题也便于定位解决。

在搭链的过程中，请注意记录和整理各种关键信息，如节点ID、IP、端口、god帐号、系统代理合约地址、证书、密码等，妥善保管，防止泄露，后续可供客户端配置使用。

2.下载并编译web3sdk。web3sdk自带大量的接口，可全面读写区块链信息，基于web3sdk生成的合约接口代码基本上也可以做到不需要手写任何abi解析即可面向对象的与合约交互，编程复杂度大为降低。

因为该sdk目前的脚本是在linux上编写的shell脚本，建议在linux服务器上下载编译和使用，如果在windows、mac等开发机上开发，可以把生成的jar包复制到开发机器上。这里注意两点：第一，编译sdk时，必须使用jdk1.8不能使用openjdk；第二，公网以太坊的web3j不能直接使用，FISCO-BCOS使用的版本已经做了大幅修改。

3.在web3sdk同一台机器上，需要智能合约编译器fisco-solc,参见使用手册中 “1.2.2 安装FISCO BCOS的智能合约编译器“里的下载部署。

以上，主要环境就绪：

1：链节点环境，客户端将跟这些节点通信

2：编译环境，包括fisco-bcos平台代码的环境和web3sdk所在的环境

3：客户端开发环境，编写java项目代码的环境

以下步骤需要在多个环境中切换来切换去操作，要清晰的记住和快速切换环境，或者事先准备好各种相关配置信息（熟悉之后）。

1. 编写编译智能合约（在编译环境）

本样例提供一个最简智能合约，演示对字符串和整型数的基本操作，包括设置和读取计数器名字、增加计数、读取当前计数。并通过receipt log的方式，把修改记录log到block里，供客户端查询参考。这里提一下，receipt log用处很大，可以是区块链和业务端同步交易处理过程和结果信息的有效渠道。 合约代码如下，非常的简单。

Counter.sol

首先到下载编译了web3sdk的linux服务器上，进入web3sdk目录，如/mydata/web3sdk。

我的做法是在我的环境变量文件里设定路径。

```
vi ~/.bash_profile （或/etc/profile，随意）
```

加入两行：

```
export web3sdk=/mydata/web3sdk #注意指向正确的路径
export fiscobcos=/mydata/FISCO-BCOS #注意指向正确的路径‘’‘
```

保存退出，并执行命令使配置生效。

```
source ~/.bash_profile
```

然后

```
cd $web3sdk/dist/contracts
```

把Counter.sol放置到$web3sdk/dist/contracts下。

```
cd $web3sdk/dist/bin
```

执行编译合约的任务

```
./compile.sh org.bcosliteclient #注意参数是java项目调用合约的代码所在包名
cd $web3sdk/dist/output
```

可以看到应该生成了合约的.abi, .bin等文件，以及在当前目录的子目录下 org/bcosliteclient/Counter.java文件。这个java文件可以复制到客户端开发环境里，后续建立的java工程的对应的包路径下。

web3sdk的dist目录结构中和以上命令相关的部分如下:

|– bin #comple.sh命令行工具所在目录

|– contracts #要编译的合约都放这个目录

|– output #编译成功的输出目录

‘– org/bcosliteclient #指定包名，则合约编译成功后输出到这个目录

2. 部署合约 （在编译环境）

部署合约有两种方式，一种是采用fisco bcos平台的脚本工具。

```
cd $web3sdk/dist/contracts
cp Counter.sol $fiscobcos/tool #假定$fiscobcos环境变量已经设定，在我的服务器上是/mydata/
↪FISCO-BCOS
babel-node deploy.js Counter #注意没有.sol后缀
```

屏幕打印：

```
deploy.js
......................Start.....................
Soc File :Counter
Warning: This is a pre-release compiler version, please do not use it in␣
↪production.
Countercomplie success!
send transaction success:␣
↪0x97d6484ee835935542a87640cdd2522496214aa22700e5c6d80bdd56744d5381
Countercontract address:0x2ae8d29ef2392c7e16003b387ed52def707769e2
Counter deploy success!
```

注意把 0x2ae8d29ef2392c7e16003b387ed52def707769e2这个地址(每次运行都会有不同），记录下来。

或者在*$fiscobcos/tool/output*目录下的Counter.address文件里也会保存最近一次部署所得的地址，可cat *$fiscobcos/tool/output/Counter.address*查看

另外一种是可以通过java客户端直接部署，在java代码部分介绍。

3. 为客户端生成安全通信的证书client.keystore（在编译环境）

注意，这一步其实是和搭链密切相关的，搭链时，已经指定了特定机构，这一步需要和搭链时的机构名，证书等使用相同的信息，否则无法客户端无法和节点通信，建议就在搭链时的操作环境下执行。

假定是基于1.3.x版本搭建的链，

```
cd $fiscobcos/cert
```

执行

```
./sdk.sh [机构名] org.bcosliteclient
```

根据提示输入一些信息，包括【客户端keystore密码】等，请牢记相关密码

将在当前目录的 [机构名]/org.bcosclient目录下，可见生成的文件。

把ca.crt, client.keystore这两个文件，复制到【客户端开发环境】上java项目的classpath路径下。

4.建立一个最简的java项目（在客户端开发环境）

开发环境：依赖jdk1.8, gradle等常规的java开发工具，按这些工具的官方安装方式进行安装即可。

打开eclispe建立一个简单的java app工程，假定命名bcosliteclient（也可以直接下载附件的工程,在eclipse里直接import-> exist gradle project）。建议在workspace里也import进web3sdk的工程代码，便于调试和阅读接口代码，查看类定义和区块链接口等。

bcosliteclient.zip

可使用附件工程里的build.gradle配置文件，包含了基本的java库依赖，在classpath下放置附件工程里applicationContext.xml文件（基于web3sdk1.2版本）,默认的log4j2.xml配置文件默认把所有信息打印到控制台，便于观察，可以根据自己的喜好和需要修改

在项目目录下执行gradle build，应该会自动拉取所有相关java库到项目的lib目录下。

同时也把fisco bcos的web3sdk.jar复制到项目的lib目录下。

对java项目进行必要的配置，如文本编码方式，build path等，创建一个空的带main入口的java文件，如在org.bcosclient包路径下的bcosliteclient.java，写一两行打印输出什么的，保证这个简单的java项目能正常编译跑通，避免出现开发环境问题。

5. 为java客户端配置相关信息（在客户端开发环境）

打开java项目的applicationContext.xml文件,部分信息可以先用默认的,先关注这些配置项

```
<!-- 系统合约地址配置，在使用./web3sdk SystemProxy|AuthorityFilter等系统合约工具时需要配置 -->
<bean id="toolConf" class="org.bcos.contract.tools.ToolConf">
    <property name="systemProxyAddress" value="【系统合约代理地址,对应节点config.json里的systemproxyaddress】" />
    <!--GOD账户的私钥-->
    <property name="privKey" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的privKey】" />
    <!--GOD账户-->
    <property name="account" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.txt里的address】" />
    <property name="outPutpath" value="./output/" />
</bean>

<!-- 区块链节点信息配置 -->
<bean id="channelService" class="org.bcos.channel.client.Service">
    <property name="orgID" value="WB" /> <!-- 配置本机构名称 -->
        <property name="allChannelConnections">
            <map>
                <entry key="WB"> <!-- 配置本机构的区块链节点列表（如有DMZ，则为区块链前置）-->
                    <bean class="org.bcos.channel.handler.ChannelConnections">
                        <property name="caCertPath" value="classpath:ca.crt" />
                        <property name="clientKeystorePath" value="classpath:client.keystore" />
                        <property name="keystorePassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="clientCertPassWord" value="【生成client.keystore时对应的密码】" />
                        <property name="connectionsStr">
                            <list>
                                <value>[nodeid]@[ip]:[channelPort]</value>
                            </list>
                        </property>
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
</beans>
```

Java Config

找到”区块链节点信息配置”一节，配置密码

```
    <property name="keystorePassWord" value="【生成client.keystore时对应的密码】" />

    <property name="clientCertPassWord" value="【生成client.keystore时对应的密码】" />
```

配置节点信息：

```
<property name="connectionsStr">

    <list>

            <value>【节点id】@【IP】:【端口】</value>

    </list>

</property>
```

节点id、ip、端口，和需要连接的节点必须一致。

如节点服务器上，节点数据目录所在路径为/mydata/nodedata-1/，那么节点id可以在/mydata/nodedata-1/data/node.nodeid文件里查到，而其他信息在/mydata/nodedata-1/config.json里可以查到。

注意，这里的端口是对应config.json里的channelPort，而不是rpcport或p2pport。

在list段里可以配置多个value，对应多个节点的信息，以实现客户端多活通信。

另外，可以进行系统合约地址配置，在调用SystemProxy|AuthorityFilter等系统合约工具时需要配置。对应信息需要搭链时的服务器环境去查询：

```
        <bean id="toolConf" class="org.bcos.contract.tools.ToolConf">

            <property name="systemProxyAddress" value="【系统合约代理地址,对应节
点config.json里的systemproxyaddress】" />

     <!--GOD账户的私钥-->

     <property name="privKey" value="【对应搭链创建god帐号环境$fiscobcos/tool/godInfo.
↪txt里的privKey】" />

     <!--GOD账户-->

            <property name="account" value="【对应搭链创建god帐号环境$fiscobcos/tool/
↪godInfo.txt里的address】" />

            <property name="outPutpath" value="./output/" />

        </bean>
```

6. 编写java客户端代码调用合约（在客户端开发环境）

示例用一个单独的CounterClient.java文件来包含所有相关代码，包括初始化客户端，部署合约，修改名字，发交易调用计数器计数，查询交易回执等。注意项目目录下的Counter.java是由web3sdk的compile.sh工具根据Counter.sol合约自动生成的，不需要进行修改。

代码相当简单，可直接阅读和使用。

CounterClient.java

注意，如果采用java客户端部署合约，则把main方法里的deployCounter()注释去掉。

代码里对应sol合约的接口，做了一次setname交易，一次addcount交易，并获取交易后的结果，解析receipt（对应sol合约里的event）里的日志信息。

采用了java的Future特性来等待区块链共识，示例代码是同步堵塞等待，可以根据自己的需要，基于Future改为异步等待或响应式通知。

运行后，屏幕打印出以下信息，则大功告成

```
2018-07-09 18:25:51.963 [main] INFO  bcosliteclient(CounterClient.java:90) -␣
↪setname-->oldname:[MyCounter from:500,inc: 100],newname=[MyCounter from:500,
↪inc: 100]

2018-07-09 18:25:51.980 [main] INFO  bcosliteclient(CounterClient.java:98) -␣
↪Current Counter:600

2018-07-09 18:25:51.984 [main] INFO  bcosliteclient(CounterClient.java:106) -␣
↪addcount-->inc:100,before: 500,after: 600,memo=when tx done,counter inc 100

2018-07-09 18:25:51.998 [main] INFO  bcosliteclient(CounterClient.java:145) - <--
↪startBlockNumber = 251,finish blocknmber=252
```

到此为止，已经有了一条可运行的链，一个可自由发挥的客户端，可以继续深度开发体验如CNS，系统合约，权限等更多的FISCO BCOS功能了。

9.4.3 WIKI： 区块链应用系统开发TIPS

作者：**fisco-dev**

- 各种私钥需保存在安全的区域，包括离线硬件设备，防火墙后的机房区域等。在区块链节点上使用keystore保存私钥上是不够安全的。
- 用于交易签名的私钥由业务模块负责安全加载（从加密文件，配置文件，数据库等)，FISCO BCOS采用sdk封装掉这个过程的细节（参见开发手册）。
- 建议每个业务模块采用不同的私钥和公钥帐号，易于分辨是来自哪个业务模块的交易，方便进行权限控制
- 根据业务需要，对各交易帐号是否能部署合约，能调用哪个合约，能调用哪个合约的哪个接口进行权限控制。
- 业务采用sdk/api封装的接口和区块链节点通信。采用sendrawtransaction发送交易，，交易数据由业务层签名。构建交易时，注意设置（或由sdk设置）blocklimit和randid。
- 应用开发者设计sdk是可以有两个层次，一个是面向底层接口的通用sdk，封装了jsonrpc和协议编解码（大部分可以直接采用web3j等开源库并进行定制），提供可配置化的基础通信能力，指向特定的区块链节点，完成多活轮询等功能。另一个是，面向业务接口的sdk，呈现业务层容易理解的、基于业务字段的接口，业务sdk调用底层sdk，发送交易和获得消息回调。
- 可以采用业务数据库存储业务数据或从链上定时导出数据入库的方式，解决关联查询，批量计算（如对账和文件导出）的场景。
- 设计智能合约时，应进行逻辑和数据分离，保持数据结构的稳定性，如需要修改数据结构，原则上只增加字段，不减少字段或修改字段含义，部分非关键的字段，可以设计为字符串，由业务自行解析，以提供一定的扩展性。
- 进一步的，如果能在交易参数和receipt log里解决数据状态存储和业务流程的，则不用state存储，因为state状态读写过程较繁琐且因维护状态树会占用较大磁盘空间。
- 尽可能的把每个处理环节的处理次数，处理时间（平均时间，最大时间）打印到统计和监控日志里，采用监控检测工具进行监测和评估，预测和检测瓶颈。平台已经对关键处理环节埋点输出到日志，更细和更丰富的业务数据埋点需要开发者自行开发。
- 根据业务场景的交易频度估算带宽，建议至少保证5M以上带宽，且注意网络延时，断连等QoS指标。

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

9.4.4 FISCO BCOS和以太坊差异综述

FISCO BCOS源起于2016年中，在金融区块链合作联盟（简称金链盟）成立后，联盟内组建了对底层平台技术的研究课题组，开展区块链底层平台的方案设计，技术选型和开发实施。

本着“避免重复造轮子”的精神，我们考察了几大区块链开源平台，其中用户众多，社区活跃度高，技术发展较成熟的以太坊进入了视野，以太坊对智能合约的支持是标志性的亮点，便于实现复杂的金融业务逻辑，所以课题组选择了以太坊开源平台为技术原型，进行深度开发重构，打造符合金融场景的联盟链平台。

秉承“来自社区，回馈社区“的社区精神，以及遵循GPL v3.0开源协议，随着底层平台的成型，金链盟成立底层平台开源工作组，将命名为FISCO BCOS的联盟链底层平台向广大社区完全开源，并加入丰富的技术文档，开发者SDK，使用案例，知识库等，和社区共同发展，推进技术研究和应用落地。

FISCO BCOS最初源自以太坊C++版本，社区里对以太坊已经有所研究的技术人员可以根据之前的经验快速理解和上手，也可以复用以太坊社区已有的大量技术资料。

同时，FISCO BCOS也根据金融场景对安全、功能、性能、监管合规的要求进行大幅的修改，和以太坊已经存在较大的差异，FISCO BCOS和以太坊是面向不同领域的产品，在和社区开发者交流的过程中，经常会讨论到两者的区别和联系，本文把相关的讨论总结如下。

- **去除代币**：FISCO BCOS去掉了代币逻辑，在生成区块时不会发行和奖励代币。Account数据结构里的Balance已经失去了原有含义，不要再关注和使用。在商业逻辑中，建议开发者采用智能合约定义自己的资产类型和交易规则。去除代币的原因是，希望联盟链参与者利用区块链的分布式账本技术、密码学等底层技术特性，开展创新的金融业务，提升运营效率，降低业务成本，而不需要依赖代币发行获利。

- **GAS控制**：FISCO BCOS保留了智能合约引擎的gas控制逻辑，以保障计算安全，针对每个区块里的每个交易能消耗多少gas，在系统合约的maxBlockHeadGas和maxTranscationGas进行配置（参见使用说明的systemcontractv2/ConfigAction.sol），默认取值较大，开发者依旧可以用以太坊类似的方法对合约消耗的gas进行评估，以计算每个block里最多能包含多少个交易。最后，由于去除了代币，Gas的消耗不和代币挂钩。

- **共识算法**：FISCO-BCOS不再采用ethash等挖矿算法，没有“矿工”的角色，不需要挖矿。FISCO BCOS基于插件化的共识机制，实现实用性拜占庭容错（PBFT）共识算法，以及信任leader的Raft算法，使用者可根据需要进行插件化的配置。联盟链共识算法速度更快，交易延迟低，不需要消耗大量的计算资源，产生的交易数据一致性高，一旦生成，就具有高度的事务确定性，更适合用于金融场景，能达到秒级出块，无分叉快速确认的效果。

- **准入标准**：和以太坊允许全网自由接入不同，做为联盟链平台，FISCO-BCOS在组网时结合网络策略和节点身份等措施实现准入控制，需要经过类似“联盟委员会”或运营方的准中心化机构审核批准，或者由链上已经接入的多方进行分布式审批核准，新节点才能接入到商业网络中来，避免了未经授权的恶意节点接入，以更好的保护信息安全。准入控制的元素包括节点IP，节点标识，CA证书等，配置流程相对较繁琐和严谨，请按使用手册仔细操作，否则可能出现组网问题。

- **节点发现**: FISCO-BCOS不使用种子服务器和uPnP技术自动发现其他节点，需要精确的配置联盟链里其他的节点，才能进行互联。

- **帐号管理**:以太坊可以在节点上管理keystore，包含账户的私钥，FISCO-BCOS把私钥管理和节点彻底解耦，放到业务客户端。当以太坊节点包含keystore，可采用sendTransaction接口进行交易，需要用account.unlock指令解锁帐号，在节点上对交易进行签名，这个时候如果网络防护不当,会有一定的安全隐患。FISCO BCOS不再使用sendTransaction，而采用sendRawTransaction，由掌握私钥的客户端进行签名，安全性更高。由于去币和帐号管理的修改，FISCO BCOS也不支持原有的客户端”钱包“的概念。

- **身份标识**：以太坊上采用公钥形式，节点、帐户可匿名。联盟链出于合作和监管的需求，要求所有参与者身份可知，包括交易帐号和节点标识等，FISCO-BCOS结合PKI体系和链上合约的方式确认和公示身份，各帐号在链上的操作和交易均可揭示身份，无法抵赖。

- **存储安全**：FISCO-BCOS对存储在硬盘的区块、交易、数据都可以采用高强度的算法进行加密，密钥可能独立管理，以彻底保障数据安全。

- **权限体系**：FISCO-BCOS实现了接口级的权限控制，可以定义某一个帐号是否可以部署合约，是否能调用某一个合约的某个接口，结合智能合约内部图灵完备的逻辑，可以实现全方位的角色和权限控制，在商业环境尤为重要，用于实现各司其职，DO分离，以及对人员和业务的审计管控等。

- **依赖代码库**：FISCO-BCOS引入了更多的特性，也没有严格同步以太坊主网Cpp分支，对各种底层库的依赖已经和主网有所不同，从源码编译FISCO-BCOS和安装部署时，请参照使用手册，按FISCO-BCOS指定的依赖库列表和版本号进行环境的初始化。

- **接口字段**:举例，以太坊的交易采用nonce去重，nonce要求严格递增，意味着客户端需要等待上一个交易确认后才能发起下一个交易。为支持客户端并发的交易，FISCO-BCOS采用randomid代替nonce，客户端生成不重复的随机数写入交易，节点则可以并行处理且交易防重放。另外还有其他的一些交易字段差异（持续修改请参考代码），导致交易数据结构和以太坊已经不兼容。
- **SDK和工具**: 由于接口字段差异，以及AMOP等扩展通信协议的加入，FISCO-BCOS的客户端SDK以及运维管理工具和以太坊也有所不同，必须采用FISCO-BCOS github上发布SDK和工具，目前已经有java版(源自web3j)和nodejs版，其他语言版本陆续开发中。FISCO-BCOS没有支持Truffle，Mist等以太坊社区使用的工具，FISCO-BCOS项目自带的工具基本能满足开发和生产需求，如有其他需求，请反馈给FISCO-BCOS开发团队，以共同完善。
- **合约编译版本**: FISCO BCOS对EVM进行了指令扩展，同时以太坊主网的智能合约陆续增加了一些指令，FISCO-BCOS会关注主网的进展，如关键性更新会进行整合，在整合前的一段时间内，FISCO-BCOS的智能合约版本和以太坊主网会有所区别。编译合约采用的编译器也有所不同，请使用FISCO-BCOS项目中提供的fisco-solc(参考使用手册1.2.2 安装FISCO BCOS的智能合约编译器)。
- **合约管理**：FISCO-BCOS引入CNS（Contract Name Service)机制，对智能合约进行命名和寻址，以简化业务开发者对合约的管理，依托CNS，开发者可以精细的管理合约的版本号，执行灰度升级等操作。CNS的功能也整合在FISCO-BCOS的客户端里。
- **监控运维**：FISCO-BCOS在交易处理、共识等全流程都进行了埋点，针对性能数据、异常情况输出大量的日志，便于专业运维团队监测和把控运行情况，进行运营管理和性能调优。
- **其他**： 在代码级别，对原有代码结构进行重构，优化各消息队列，并行验证交易，HASH计算优化，修正稳定性问题等，不直接体现在接口和功能上。

  在功能上，增加了AMOP消息协议，多链并行计算架构和热点帐户解决方案，以及大量的业务案例，详细信息请参见GITHUB上的使用说明和知识库。

综上所述，FISCO-BCOS在以太坊的基础上进行深度的定制、重构、优化，以达到金融级的安全和性能表现，通过系统合约等方法极大的丰富了区块链的可治理性，并面向工业级运营运维，丰富了工具和监控信息，满足了金融业对软件质量的高标准要求。

依托金链盟和开源社区的力量，FISCO-BCOS尚在高速发展中，在隐私保护，密码学算法，存储容量和并行计算等方面大幅优化和增加新的特性，并开源发布，以满足业界对区块链和联盟链底层平台的需求，更好的推进应用落地，FISCO-BCOS已经走出了一条安全可控、自主成长的开源之路，对FISCO-BCOS有任何的反馈，请关注github项目和加入社区群，深入交流讨论。

如果您觉得本文不错，欢迎戳这里给FISCO BCOS打star:star:。

- 区块链平台调研与分析报告

CHAPTER 10

社区

10.1 贡献代码

非常感谢能有心为FISCO-BCOS社区贡献代码！

如果你希望很快速的贡献代码，OK，直接pull request到develop分支吧。

如果你希望以更专业更规范的方式开发，那么就想想自己要做什么吧?

- 我要开发新工程！
- 我要开发新特性！
- 我要修bug！
- 我发现了一个bug，但不知道怎么改。
- 我想看看是怎么release大版本的。

OK，本文都会告诉你相关的流程。

FISCO-BCOS社区的开发分支策略是git-flow，感兴趣的小伙伴可以直接跳到本文附录1了解具体情况。

有任何问题都可以直接找FISCO-BCOS管理员，联系方式在附录2。

10.1.1 新工程开发

新工程是在FISCO-BCOS生态中创建一个全新的组件。通过迭代的方式不断完善组件的功能。在开发前，可联系FISCO-BCOS社区相关成员，获取一些指导。

准备

1. 充分的调研、需求分析和知识储备
2. 联系FISCO-BCOS社区
3. FISCO-BCOS管理员新建工程，并授权给相关开发人员

开发

- 分支策略和迭代开发流程与本文档其它部分相同

10.1.2 新特性开发

新特性是对FISCO-BCOS中某个工程功能的补充，开发周期较长，在开发时需要付出较长的时间和精力。在开发前，可联系FISCO-BCOS社区相关成员，获取一些指导。

开发前

1. 从develop分支fork出一个分支，叫feature-xxxxxx
2. 将feature-xxxxxx分支授权给相关特性负责人

   tips：

   推荐在FISCO-BCOS的工程内开发，开发情况对其它开发者可见。请联系FISCO-BCOS管理员（邮件或issue），由管理员新建feature分支，并将分支授权给开发人员。

开发中

1. 不定期拉取develop分支代码，合入此feature分支
2. 多人开发feature时，通过pull request的方式，将个人分支的代码提交代码到此feature分支，特性负责人负责审核pull request

开发后

1. 拉取最新develop代码
2. 补充changelog
3. 提测
4. 测试通过后，pull request的方式提交到develop分支
5. FISCO-BCOS管理员审核pull request后合入develop分支

10.1.3 bug修复

bug是不可避免的事情。在修bug时，可与FISCO-BCOS社区内的相关开发者充分的沟通，以获取更多的建议。

在修bug前，先对bug进行评估，判断bug的严重程度，再根据程度进行相应的流程：

（1）非紧急bug：文档错误，注释错误，较小的代码错误

（2）紧急bug：代码逻辑错误，代码兼容性错误

非紧急bug修复

直接以pull request的方式提交到develop分支，给FISCO-BCOS管理员审核。

紧急bug修复

修复前

1. 从master分支fork出一个分支，叫hotfix-xxxxx分支
2. 将hotfix-xxxxx分支授权给相关负责人

   tpis：

   与feature类似，若需要在FISCO-BCOS工程内修bug，请联系FISCO-BCOS管理员（邮件或issue），由管理员新建hotfix分支，并将分支授权给开发人员。

修复中

- 由于hotfix分支较多，为避免冲突，需尽快修复

修复后

1. 拉取最新master代码，补充changelog
2. 提测
3. 测试通过后，修改版本号（FISCO-BCOS节点代码的版本号在CMakeList.txt，Changelog.md和release_note.txt中）
4. 同时提pull request到master分支和develop分支，若改动较大，可只提交到develop分支
5. FISCO-BCOS管理员审核pull request后合入master分支和develop分支

10.1.4 issue提交

不懂的问题，新发现的bug，改进的建议等，都直接提交到相应工程的issue处即可。

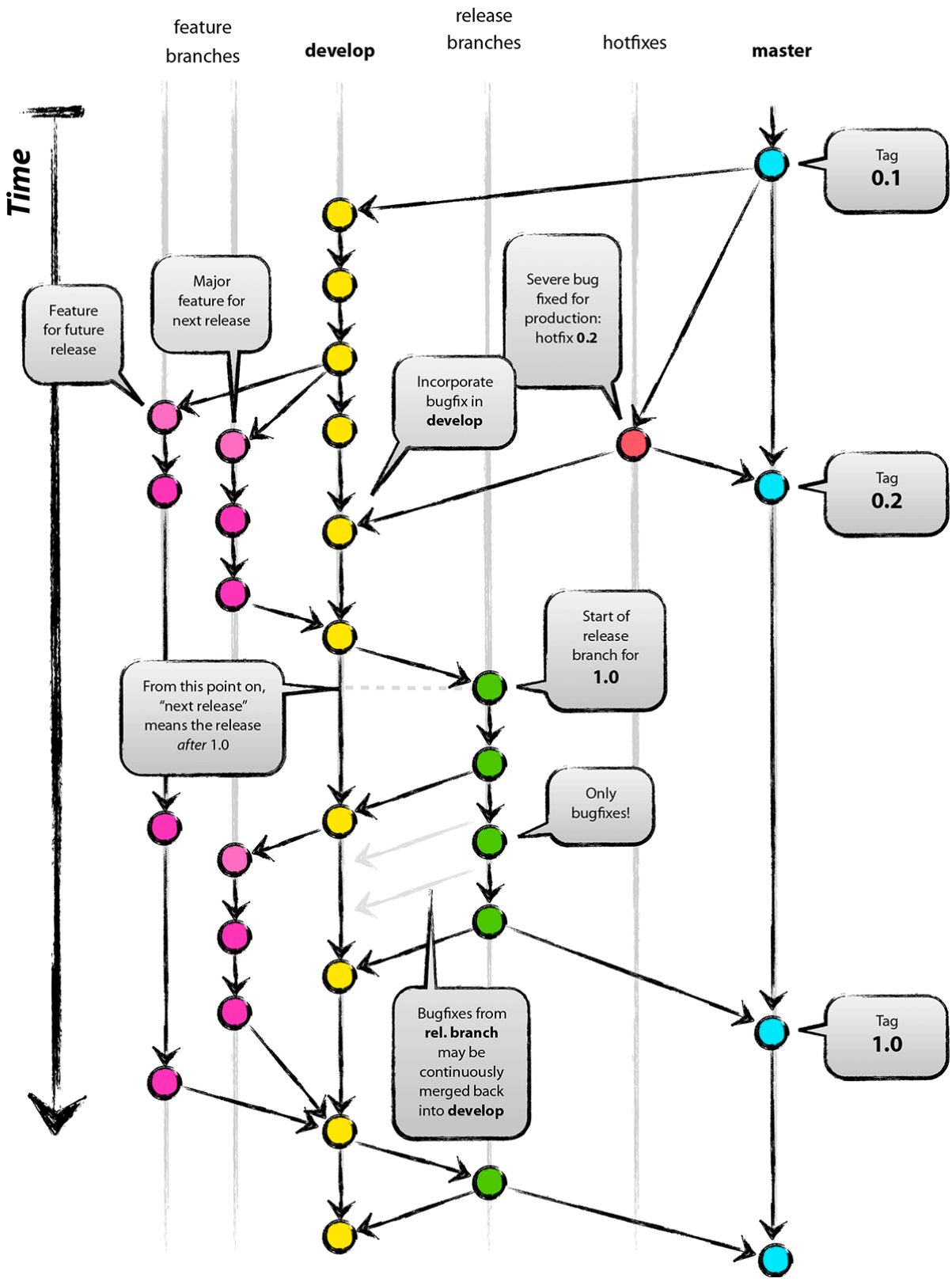
10.1.5 release版本

FISCO-BCOS社区按照计划会不定期的发布版本，发布版本的流程如下。

1. 从develop分支fork一个release分支，确定更新版本号
2. 测试，代码扫描
3. 测试和扫描通过后用pull request的方式合入master分支和develop分支
4. 基于master分支release相应的版本，并打上tag

10.1.6 附录1：Git Branching Model: git-flow

git-flow将以develop分支为核心，将feature和release过程分开。支持多个feature并行的开发和测试，同时能够并行的release和改bug。



master分支

- 绝对稳定分支，有版本号

develop分支

- develop分支从master分支fork出来，永远领先或等于master分支
- 也要求稳定，但稳定程度比master稍差，只确保feature的稳定

- 无版本号

feature-*分支

- feature-*分支从develop分支fork出来，是每个独立特性的开发分支
- feature-*分支由特性开发者管理，多人开发时，通过pull request的方式，由特性核心开发者审查后合入
- feature-*分支必须从develop分支拉取代码，同步到最新，再进行feature测试
- feature测试后才能将相关feature分支合入develop分支

release-*分支

- release-*分支根据排期，从develop分支fork出来，形成大版本号
- release-*分支在release测试时，回归的问题直接提交到release分支上
- release-*分支在release测试后，需同时合入dev分支和master分支

hotfix-*分支

- 要修bug时，从master拉出此分支
- 为避免冲突，bug要尽快修复
- bug修复代码完成后，需从master拉取最新的代码，再进行测试
- 测试通过后，同时合入develop分支和master分支，形成小版本号
- 若bug修复改动较大，可先合入develop分支，不合入master分支

10.1.7 附录2：FISCO-BCOS 管理员联系方式

邮箱：service@fisco.com.cn

微信群：添加群管理员微信号fiscobcosfan，拉您入FISCO BCOS官方技术交流群。

群管理员微信二维码:

10.2 FAQ

10.2.1 部署相关问题

1.如何快速安装部署FISCO BCOS?

解决方案：作为范例，一键快速安装部署提供了快速编译安装FISCO BCOS、并且部署2个节点的指南。

2.如何使用Docker安装FISCO BCOS?

解决方案：目前FISCO BCOS平台已经发布了Dockerfile文件和Docker Hub上预构建的镜像，并将持续同步源码的更新。想要使用Docker快速安装一个或者多个节点，请参阅使用Docker安装部署BCOS指南。

3.在build路径运行"make -j2" 卡死

解决方案：编译的过程中需要从网络上下载依赖的包，网络条件太差可能卡死。建议在网络条件良好的环境搭建FISCO BCOS，或者从其他渠道下载依赖库包后拷贝到你的编译目标路径下。亦可参

见issue：make -j2 运行卡死

4.AWS亚马逊云安装问题

4.1 AWS亚马逊云Centos-7.2安装FISCO-BCOS问题：

执行yum 显示No package cmake3 availale

```
[root@ip-172-31-1-126 fisco-bcos]# ./build.sh
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
- base: mirrors.mit.edu
- extras: mirrors.umflint.edu
- updates: mirrors.tripadvisor.com
  No package cmake3 available.
  Error: Nothing to do
```

解决方案：

安装epel

```
sudo yum -y install epel-release
```

4.2 AWS亚马逊云/阿里云 ubuntu16.04安装FISCO-BCOS问题：

执行apt显示 Unable to locate package lrzsz

```
root@ip-172-31-15-64:/fisco/fisco-bcos# apt install lrzsz
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package lrzsz
```

解决方案： 执行 `apt-get update`

4.3 部署合约报异常

```
root@ip-172-31-15-64:/fisco/fisco-bcos/tool# babel-node deploy.js HelloWorld
RPC=http://127.0.0.1:8545
Ouputpath=./output/
/fisco/fisco-bcos/tool/web3sync.js:65
let getBlockNumber = (() => {
^^^
SyntaxError: Block-scoped declarations (let, const, function, class) not yet␣
↪supported outside strict mode
  at exports.runInThisContext (vm.js:53:16)
  at Module._compile (module.js:374:25)
  at loader (/usr/local/lib/node_modules/babel-cli/node_modules/_babel-register@6.
↪26.0@babel-register/lib/node.js:144:5)
  at Object.require.extensions.(anonymous function) [as .js] (/usr/local/lib/node_
↪modules/babel-cli/node_modules/_babel-register@6.26.0@babel-register/lib/node.
↪js:154:7)
  at Module.load (module.js:344:32)
  at Function.Module._load (module.js:301:12)
  at Module.require (module.js:354:17)
  at require (internal/module.js:12:17)
  at Object.<anonymous> (/fisco/fisco-bcos/tool/deploy.js:12:16)
  at Module._compile (module.js:410:26)
```

解决方案:

查看nodejs版本

```
root@ip-172-31-15-64:/fisco/fisco-bcos/tool# node -v
v4.2.6
```

Nodejs版本需要大于6:

```
curl -sL https://deb.nodesource.com/setup_8.x | sudo -E bash -
sudo apt-get install -y nodejs
```

运行这个命令可以将nodejs升级到8以上的版本。

4.4 npm ERR! enoent ENOENT: no such file or directory

解决方案:

查看nodejs是否已经安装，执行`node -v` 可查看 如果已经安装将build.sh一件安装脚本里面的

```
sudo npm install -g cnpm --registry=https://registry.npm.taobao.org
sudo cnpm install -g babel-cli babel-preset-es2017
echo '{ "presets": ["es2017"] }' > ~/.babelrc
```

屏蔽掉,重新执行。

10.2.2 智能合约相关问题

1.如何运行智能合约?

解决方案：目前FISCO BCOS平台已经发布了contract_samples (位于项目根目录下)，示范了使用Java 和Node.js开发智能合约客户端的范例，展示了如何编译、部署、调用智能合约，供参考。更多问题细节，亦可参见issue:智能合约问题

10.3 GitHub

https://github.com/FISCO-BCOS

10.4 Issue

https://github.com/FISCO-BCOS/FISCO-BCOS/issues

10.5 金链盟

https://www.fisco.com.cn/

10.6 联系我们

FISCO BCOS平台相关的技术问题，可以到官方技术交流群讨论。

群管理员微信号为：fiscobcosfan

群管理员微信二维码:


FISCO BCOS