MOACdocs-chn Documentation

MOAC

2019 年 07 月 11 日

目录:

CHAPTER 1

名词解释

1.1 Vnode

墨客主网（母链）节点，用于主网挖矿，主网账本同步，主网交易以及子链数据传输的节点。

1.2 Scs

墨客子链节点，用于子链挖矿，子链账本同步以及子链业务逻辑执行的节点，也称为子链矿工

1.3 子链矿工池

存储子链矿工的池子，本质上池子是一个智能合约，需要子链节点注册

1.4 vnode 代理池

存储 vnode 代理节点，本质上池子是一个智能合约，需要 vnode 节点

1.5 子链控制合约

用于控制整个子链的流程

1.6 监听节点（Monitor）

监听节点是一个特殊的 Scs 节点，可以用来监听某条子链的运行情况，当一个节点成为监听节点后，其只负责同步该子链的区块信息，不参与子链出块。Dapp 用户可以通过该节点监控子链运行情况

1.7 flush

子链的一个特殊操作，每条正常运行的子链每隔一段时间需要向母链进行状态刷新，并且同时完成：Scs 矿工的收益发放；有币子链和母链之间的货币充提等操作

CHAPTER 2

子链要点

2.1 概述

子链是在母链之上的独立的区块链系统，子链中的每个节点称为 scs，scs 的通讯通过母链，并定期向母链进行数据背书。子链上可以单独跑业务逻辑，母链无需知道。

母链上可以跑多条子链，墨客采用分片技术，随机将 scs 分配给不同的子链。

子链需要依赖于母链来运行，因此，运行一条子链需要 M 个母链节点（vnode）和 N 个子链节点（scs）。

普通子链节点 scs 可以称作子链矿工，其主要负责子链的出块，是维持一条子链稳定安全运行的根本。

子链节点下的母链节点，需要选取一部分注册成 vnode 代理的节点，vnode 代理节点的作用是用于维护子链稳定运行，vnode 代理节点同样有一个代理矿工池

部署子链控制合约时需要指定以上两个池子的地址。

子链合约部署完后，即可调用 registeropen 来召唤池子里的 scs 注册；当数量达到预期后，就可以调用 registerclose 来初始化子链区块，让子链开始运行。

子链需要每隔一段时间发起 flush 操作，将关键状态写入母链进行背书。除此之外，flush 还将完成节点收益分配和有币区块链的母子链充提操作。

可以调用子链控制合约里的 registerasmonitor 来将一个子链节点注册成一个侦听节点。侦听节点只负责信息查询，不参与普通子链节点的服务，适合 dapp 用户部署来监控子链运行状态。

如果普通子链节点状态不稳定或弄虚作假，在 flush 的时候，有几率被没收押金，强制退出这条子链的服务。这时，可以调用子链控制合约的 registeradd 方法，将 scs 池子里的其他 scs 作为备用节点来为这条子链服务。

部署子链方可以调用子链控制合约的 close 方法关闭这条子链。此时会进入清算状态，待所有子链收益结清后，即关闭子链。请注意，子链业务逻辑清算不包含。

子链节点本身可以调用子链控制合约的 withdraw 方法来结束为某一子链服务，并得到返还的押金。

2.2 子链中的 MOAC 押金和消耗

为了使子链安全稳定的运行，MOAC 引入的押金机制，主要体现在以下几个方面：

1、每个子链节点在第一次启动后，将会有一个唯一的墨客钱包地址，在部署子链前，需要在向这个地址打入 1 个 MOAC 作为运行费用；

2、每个子链节点注册进入子链矿工池时，需要向矿工池缴纳一定的押金，最小值由子链控制合约设置，最大值不限。子链节点每被选中一次，将会扣除一定数额的押金，当押金被扣完后，该节点将不会再参与新的子链，退出子链时，可以调用方法取回押金；

3、当一个子链节点注册成一个监听节点时，需要缴纳一定的押金；当退出子链时可以取回押金；

4、押金一般不会扣除，但在 flush 时，如果有企图作弊的节点，将会按照规则踢出子链，并扣除押金，不再返还；

子链中的 MOAC 消耗：

首先，调用子链方法不会消耗任何 gas，但是，dapp 运营方需要向子链控制合约地址打入一定量的 MOAC 维持子链运行，这部分 MOAC 将会消耗在给子链矿工费用和主链充提 gas 返还上。

2.3 子链部署的注意点

1、子链 scs 的 vnode 需要互相 addPeer 以保证通讯畅通。同时，这些 vnode 建议尽量 add 外面的节点以保证主链高度一致。

2、子链 scs 的时钟请同步互联网标准时间。

CHAPTER 3

部署子链前的准备工作

3.1 Vnode 节点

墨客主网节点版本来源: https://github.com/MOACChain/moac-core/releases/

此文档采用的版本：1.0.5 环境：windows testnet 浏览器：http://47.75.144.55:3000/home

在测试环境 testnet 启动节点： moac-windows-4.0-amd64.exe –testnet –rpc –rpcapi "chain3,mc,net,db,personal,admin,miner"

验证:

```
windows command 执行 moac-windows-4.0-amd64.exe attach \\.\pipe\moac.ipc
运行 concole 命令 mc.blockNumber 检查是否同步到最新区块
```

3.2 各类账号

可以运行 concole 命令 personal.newAccount() 创建账号；mc.accounts 查看账号；

按序号查询余额：mc.getBalance(mc.accounts[0])

测试环境的公共提币地址：http://119.28.13.213:3000/

注意：后续消耗 gas 的操作都需要执行 personal.unlockAccount(mc.accounts[0]) 对应账号进行解锁

准备账号列表：（示例地址参考后续的命令操作）

```
子链操作账号：进行创建合约，发起交易等基本操作： 0x87e369172af1e817ebd8d63bcd9f685a513a6736
主链 vnode 收益账号：     0xf103bc1c054babcecd13e7ac1cf34f029647b08c
子链 scs 收益账号：0xa934198916cd993c73c1aa6e0c0e7b21ce7c735b ␣
↪0x2e7c076dbf6e61207a0ddb1b942ef7da8fd139f0
```

3.3 chain3 的 nodejs 环境

安装：npm install chain3

验证:

```
> chain3 = require('chain3');
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> chain3.mc.blockNumber  检查是否获得当前区块
```

CHAPTER 4

子链的部署方法

4.1 部署 Vnode 矿池合约

首先部署 vnode 矿池合约，VnodeProtocolBase，如果加入现成的 vnode 矿池，则可以忽略此步骤。

加入矿池的代理 Vnode 节点被用于提供子链调用服务和子链历史数据中转服务的节点。

以下为 nodejs 部署示例：最低保证金为 2 moac

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'VnodeProtocolBase.sol';
> contract = fs.readFileSync(solfile, 'utf8');
> output = solc.compile(contract, 1);
> abi = output.contracts[':VnodeProtocolBase'].interface;
> bin = output.contracts[':VnodeProtocolBase'].bytecode;
> VnodeProtocolBaseContract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> VnodeProtocolBase = VnodeProtocolBaseContract.new( 2, { from: chain3.mc.accounts[0], ␣
↪data: '0x' + bin,  gas: '5000000'});
> chain3.mc.getTransactionReceipt(VnodeProtocolBase.transactionHash).contractAddress
```

部署完毕后，获得 vnode 矿池合约地址 0x22f141dcc59850707708bc90e256318a5fe0b928

注意：gas 不要设置太大，不然会触发错误 exceeds block gas limit undefined

4.2 vnode 设置代理并加入矿池

修改 vnode 目录配置文件 vnodeconfig.json: VnodeBeneficialAddress 里设置收益账号: 0xf103bc1c054babcecd13e7ac1cf34f029647b08c

这个账号也作为 vnode 的 address，矿池中对应这个 vnode 的唯一编号

调用 vnode 矿池合约 register 方法加入矿池

参数:

```
from: 子链测试账号
value: 押金，必须大于矿池合约的设置值
to: vnode 矿池合约地址
data: register(address,string)
```

关于 data 传递调用 register 参数说明:

```
根据 ABI chain3.sha3("register(address,string)") =␣
↪0x32434a2e90725ed590daff07a244305001c58c49f7bef73ce5e7249acf69f561
        取前 4 个字节 0x32434a2e
第一个参数 address 传 vnodeconfig.json 的 VnodeBeneficialAddress (前面补 24 个 0, 凑足
32 个字节)
        000000000000000000000000f103bc1c054babcecd13e7ac1cf34f029647b08c
第二个参数 string 传 vnode 提供给子链的调用地址 192.168.10.209:50062
端口号对应 vnodeconfig.json 的 VnodeServiceCfg
        string 数据类型 + string 数据长度 + string 内容
        string 数据类型: 0000000000000000000000000000000000000000000000000000000000000040
        string 内容: data = new Buffer('192.168.10.209:50062', 'utf8').toString('hex');
                后面补 0, 凑足 32 个字节 ␣
↪3139322e3136382e31302e3230393a353030363200000000000000000000000000
        string 数据长度: 3139322e3136382e31302e3230393a3530303632 20 字节
                0000000000000000000000000000000000000000000000000000000000000014
        data =
↪'0x32434a2e000000000000000000000000f103bc1c054babcecd13e7ac1cf34f029647b08c000000000000000000000000000
↪'
```

调用示例:

```
> amount = chain3.toSha(5,'mc')
> data =
↪'0x32434a2e000000000000000000000000f103bc1c054babcecd13e7ac1cf34f029647b08c000000000000000000000000000
↪';
```

(下页继续)

(续上页)

```
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction({ from: chain3.mc.accounts[0], value:amount, to:
↪'0x22f141dcc59850707708bc90e256318a5fe0b928', gas: "5000000", gasPrice: chain3.mc.
↪gasPrice, data: data });
```

验证：访问 Vnode 矿池合约的 vnodeCount

```
> chain3.mc.getStorageAt("0x22f141dcc59850707708bc90e256318a5fe0b928",0x02)   // 注意
vnodeCount 变量在合约中变量定义的位置（16 进制）
```

4.3 部署子链矿池

目前针对不同的共识协议，可以创建对应的子链矿池，接受对应 SCS 的注册，并缴纳保证金，进入矿池后，成为子链的候选节点

如果加入现成的子链矿池，则可以忽略此步骤

部署 SubChainProtocolBase.sol 示例: 共识:POR 最低保证金: 2moac

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'SubChainProtocolBase.sol';
> contract = fs.readFileSync(solfile, 'utf8');
> output = solc.compile(contract, 1);
> abi = output.contracts[':SubChainProtocolBase'].interface;
> bin = output.contracts[':SubChainProtocolBase'].bytecode;
> subchainprotocolbaseContract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> subchainprotocolbase = subchainprotocolbaseContract.new( "POR",  2, { from: chain3.mc.
↪accounts[0],  data: '0x' + bin,  gas: '5000000'});
> chain3.mc.getTransactionReceipt(subchainprotocolbase.transactionHash).contractAddress
```

部署完毕后，获得子链矿池合约地址 0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac

4.4 设置启动 scs

这里我们设置两个 scs 节点

确认 userconfig.json 配置

```
VnodeServiceCfg 为代理 vnode 地址: 192.168.10.209:50062
Beneficiary 为收益账号:
        0xa934198916cd993c73c1aa6e0c0e7b21ce7c735b
        0x2e7c076dbf6e61207a0ddb1b942ef7da8fd139f0
```

分别通过命令启动 scsserver-windows-4.0-amd64 –password “123456”（生成 scs keystore 的密码）

然后在生成的 keystore 文件中分别获得 scs 地址

```
d4057328a35f34507dbcd295d43ed0cccf9c368a
0x3e21ba36b396936c6cc9adc3674655b912e5fa54
```

最后给 scs 转入 moac 以支付必要的交易费用

```
> amount = 20;
> scsaddr = '0xd4057328a35f34507dbcd295d43ed0cccf9c368a';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:chain3.toSha(amount,'mc
↪'), to: scsaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data: ''});
> scsaddr = '0x3e21ba36b396936c6cc9adc3674655b912e5fa54';
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:chain3.toSha(amount,'mc
↪'), to: scsaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data: ''});
```

可以通过查询余额进行验证

```
> chain3.mc.getBalance('0xd4057328a35f34507dbcd295d43ed0cccf9c368a')
> chain3.mc.getBalance('0x3e21ba36b396936c6cc9adc3674655b912e5fa54')
```

4.5 将 scs 加入子链矿池

调用子链矿池合约 register 方法加入矿池

参数:

```
from:  子链测试账号
value: 押金，必须大于矿池合约的设置值
to: 子链矿池合约地址
data: register(address)
```

关于 data 传递调用 register 参数说明:

```
根据 ABI chain3.sha3("register(address)") =␣
↪0x4420e4869750c98a56ac621854d2d00e598698ac87193cdfcbb6ed1164e9cbcd
        取前 4 个字节 0x4420e486
参数 address 传 scs 地址    d4057328a35f34507dbcd295d43ed0cccf9c368a   (前面补 24 个 0, ␣
↪凑足 32 个字节)
        000000000000000000000000d4057328a35f34507dbcd295d43ed0cccf9c368a
data = '0x4420e4860000000000000000000000000d4057328a35f34507dbcd295d43ed0cccf9c368a'
```

调用示例:

```
> amount = chain3.toSha(5,'mc')
> data = '0x4420e4860000000000000000000000000d4057328a35f34507dbcd295d43ed0cccf9c368a';
> chain3.mc.sendTransaction({ from: chain3.mc.accounts[0], value:amount, to:
↪'0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac', gas: "5000000", gasPrice: chain3.mc.
↪gasPrice, data: data });
```

验证: 访问子链矿池合约的 scsCount

```
> chain3.mc.getStorageAt("0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac",0x02)     // 注意
scsCount 变量在合约中变量定义的位置 (16 进制)
```

同上将另一个 scs (0x3e21ba36b396936c6cc9adc3674655b912e5fa54) 也加入子链矿池

4.6 部署子链合约

现在我们可以部署一个子链合约, 并准备将两个 scs

部署 SubChainBase.sol 示例:

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> input = {'': fs.readFileSync('SubChainBase.sol', 'utf8'), 'SubChainProtocolBase.sol':␣
↪fs.readFileSync('SubChainProtocolBase.sol', 'utf8')};
> output = solc.compile({sources: input}, 1);
> abi = output.contracts[':SubChainBase'].interface;
> bin = output.contracts[':SubChainBase'].bytecode;
> proto = '0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac' ;    // 子链矿池合约
> vnodeProtocolBaseAddr = '0x22f141dcc59850707708bc90e256318a5fe0b928' ;       // Vnode
矿池合约
```

(下页继续)

(续上页)

```
> min = 1 ;                       // 子链需要 SCS 的最小数量
> max = 10 ;              // 子链需要 SCS 的最大数量
> thousandth = 1 ;                          // 千分之几
> flushRound = 40 ;               // 子链刷新周期  单位是主链 block 生成对应数量的时间
> SubChainBaseContract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> SubChainBase = SubChainBaseContract.new( proto, vnodeProtocolBaseAddr, min, max,␣
↪thousandth, flushRound,{ from: chain3.mc.accounts[0],  data: '0x' + bin,  gas:'9000000
↪'} , function (e, contract){console.log('Contract address: ' + contract.address + '␣
↪transactionHash: ' + contract.transactionHash); });
```

部署完毕后, 获得子链合约地址 0x1195cd9769692a69220312e95192e0dcb6a4ec09

4.7 子链开放注册

首先子链合约需要最终提供 gas 费给 scs，需要给子链控制合约发送一定量的 moac，调用合约里的函数 addFund

```
根据 ABI chain3.sha3("addFund()") =␣
↪0xa2f09dfa891d1ba530cdf00c7c12ddd9f6e625e5368fff9cdf23c9dc0ad433b1
        取前 4 个字节 0xa2f09dfa
> amount = 20;
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:chain3.toSha(amount,'mc
↪'), to: subchainaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0xa2f09dfa'}
↪);
```

可以通过查询余额进行验证

```
> chain3.mc.getBalance('0x1195cd9769692a69220312e95192e0dcb6a4ec09')
```

然后调用调用合约里的函数 registerOpen 开放注册 (按子链矿池合约中 SCS 注册先后排序进行选取)

```
根据 ABI chain3.sha3("registerOpen()") =␣
↪0x5defc56ce78f178d760a165a5528a8e8974797e616a493970df1c0918c13a175
        取前 4 个字节 0x5defc56c
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x5defc56c'});
```

(下页继续)

(续上页)

验证：访问子链合约的 registerFlag 为 1，等待 scs 注册（vnode 一个 flush 周期后），访问子链合约的 nodeCount > chain3.mc.getStorageAt(subchainaddr,0x14) // 注意 registerFlag 变量在合约中变量定义的位置（16 进制）> chain3.mc.getStorageAt(subchainaddr,0x0e) // 注意 nodeCount 变量在合约中变量定义的位置（16 进制）

4.8 子链关闭注册

等到两个 scs 都注册完毕后，即注册 SCS 数目大于等于子链要求的最小数目时，调用子链合约里的函数 registerClose 关闭注册

```
根据 ABI chain3.sha3("registerClose()") =␣
↪0x69f3576fc10c82561bd84b0045ee48d80d59a866174f2513fdef43d65702bf70
        取前 4 个字节 0x69f3576f
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x69f3576f'});
```

验证：访问子链合约的 registerFlag 为 0 > chain3.mc.getStorageAt(subchainaddr,0x14) // 注意 registerFlag 变量在合约中变量定义的位置（16 进制）

SCS 自身完成初始化并开始子链运行，可观察 scs 的 concole 界面，scs 开始出块即成功完成部署子链。

CHAPTER 5

子链业务逻辑的部署

同主链相同，业务逻辑的实现也通过智能合约的方式。

5.1 DAPP 智能合约的部署

DAPP 智能合约也通过主链的 sendTransaction 发送交易到 proxy vnode 的方式进行部署。

参数:

```
to: 子链控制合约 subchainbase 的地址
gas: 不需要消耗 gas 费用，传值: 0
shardingflag: 表示操作子链， 传值: 0x1
via: 对应 proxy vnode 的收益地址
```

准备一个简单的合约 deorder.sol，部署示例

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'deorder.sol';
> contract = fs.readFileSync(solfile, 'utf8');
> output = solc.compile(contract, 1);
> abi = output.contracts[':DeOrder'].interface;
```

(下页继续)

(续上页)

```
> bin = output.contracts[':DeOrder'].bytecode;
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction({from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas:0, shardingFlag: "0x1", data: '0x' + bin, nonce: 0, via: via, });
```

验证: 合约部署成功后，Nonce 值应该是 1 可调用 monitor 的 rpc 接口 ScsRPCMethod.GetNonce 进行检查，具体详见子链接口调用部分。

5.2 DAPP 智能合约的调用

DAPP 智能合约的调用也通过主链的 sendTransaction 发送交易到 proxy vnode 的方式进行。

例如 deorder.sol 有个方法 createOrder(string ordernum)

参数:

```
to: 子链控制合约 subchainbase 的地址
nonce: 调用 monitor 的 rpc 接口 ScsRPCMethod.GetNonce 获得
gas: 0 不需要消耗 gas 费用
shardingflag: 0x1 表示操作子链
via: 对应 proxy vnode 的收益地址
data: chain3.sha3("createOrder(string)") 取前 4 个字节 0x03b7c764，加上传值凑足 32 个字节
```

调用示例:

```
> nonce = 1
> data = '0x03b7c764000000000000000000000000000000000000000000000000000000000000000100000000000000000000000000000000000000000000000000000000000000010000001'
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { nonce: nonce, from: chain3.mc.accounts[0], value:0, to:␣
↪subchainaddr, gas:0, shardingFlag:'0x1', data: data, via: via,});
```

验证: 每次操作成功后，Nonce 会自动增加 1 或者直接调用 monitor 的 rpc 接口 ScsRPCMethod.GetContractInfo 获得合约变量的方式进行验证。

CHAPTER 6

子链的监听及子链接口调用

6.1 SCS Monitor

Monitor 是一种特殊的子链 SCS 节点，其主要可以用于监控子链的状态和数据。

Monitor 不参与子链的交易共识，只是同步区块数据，提供数据查询

子链启动的方式与 scs 区别在于参数不同，主要定义了 rpc 接口的访问控制

```
scsserver-windows-4.0-amd64 --password "123456" --rpcdebug --rpcaddr 0.0.0.0 --rpcport␣
↪2345 --rpccorsdomain "*"
```

子链运行后，Monitor 可以调用子链控制合约 subchainbase 中的 registerAsMonitor 方法进行注册

调用 registerAsMonitor 参数说明:

```
根据 ABI chain3.sha3("registerAsMonitor(address)") =␣
↪0x4e592e2f6fc52405522577d357d824c923989a62e4916d9b689311d8b2a6192c
        取前 4 个字节 0x4e592e2f
参数 address 传 scs keystore 文件中的地址 （前面补 24 个 0， 凑足 32 个字节）
data = '0x4e592e2f000000000000000000000000d135afa5c8d96ba11c40cf0b52952d54bce57363'

注意 registerAsMonitor 不同版本参数
> data = contractInstance.registerAsMonitor.getData(
↪'0xd135afa5c8d96ba11c40cf0b52952d54bce57363','127.0.0.1')
```

调用示例:

```
> amount = chain3.toSha(1,'mc')
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> data = '0x4e592e2f0000000000000000000000000d135afa5c8d96ba11c40cf0b52952d54bce57363';
> chain3.mc.sendTransaction({ from: chain3.mc.accounts[0], value:amount, to:␣
↪subchainaddr, gas: "5000000", gasPrice: chain3.mc.gasPrice, data: data });
```

验证：观察 SCS monitor concole 界面开始同步子链区块，或者调用子链合约的 getMonitorInfo 方法

```
> contractInstance = SubChainBaseContract.at('0xb877bf4e4cc94fd9168313e00047b77217760930
↪')

> contractInstance.getMonitorInfo.call()
```

6.2 子链 RPC 接口

根据子链启动的参数，用户可以访问对应端口，调用接口获取子链相关数据

以调用接口 GetScsId 为列，获得当前的 scs 编号，即 scs keystore 文件中的地址。

关于 rpc http 的接口访问，可以用类似 postman 之类的工具进行快捷测试

```
header 设置:
        Content-Type = application/json
        Accept = application/json

Body 设置:
        {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetScsId","params":{}}

post 返回结果:
        {
                "jsonrpc": "2.0",
                "id": 0,
                "result": "0xd135afa5c8d96ba11c40cf0b52952d54bce57363"
        }
```

同时也可以通过 nodejs 的方式调用

```
> request = require('request');
> url  = "http://127.0.0.1:2345/rpc";
```

(下页继续)

(续上页)

```
> data = {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetScsId","params":{}};
> request({ url: url, method: "POST", json: true, body: data, headers: {"Content-Type":
↪'application/json', "Accept": 'application/json'}}, function(error, response, result)
↪{if (!error && response.statusCode == 200) {console.log(result)}});
```

以下是几个常用的 RPC 接口及调用 body 示例

GetNonce：获得子链的 nonce，这是调用子链 DAPP 合约的必要参数之一，每当子链交易发送后会自动加 1

```
SubChainAddr：子链合约地址
Sender：查询账号， 每个账号在子链有不同的 nonce
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetNonce",
                        "params":{"SubChainAddr":
↪"0x1195cd9769692a69220312e95192e0dcb6a4ec09",
                                  "Sender":"0x87e369172af1e817ebd8d63bcd9f685a513a6736"
                         }
          }
```

GetBlockNumber：获得当前子链的区块高度

```
SubChainAddr：子链合约地址
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetBlockNumber",
                "params":{"SubChainAddr":"0x1195cd9769692a69220312e95192e0dcb6a4ec09"}
          }
```

GetBlock: 获得当前子链的指定的区块信息

```
SubChainAddr：子链合约地址
Sender：查询账号
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetBlock",
                "params":{"number":1000,"SubChainAddr":
↪"0x1195cd9769692a69220312e95192e0dcb6a4ec09"}
          }
```

GetSubChainInfo：获得当前子链的信息

```
SubChainAddr：子链合约地址
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetSubChainInfo",
                "params":{"SubChainAddr":"0x1195cd9769692a69220312e95192e0dcb6a4ec09"}
          }
```

GetBalance：获得对应账号在子链中的余额

```
SubChainAddr: 子链合约地址
Sender: 查询账号
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetBalance",
                "params":{"SubChainAddr":"0x1195cd9769692a69220312e95192e0dcb6a4ec09",
                        "Sender":"0x87e369172af1e817ebd8d63bcd9f685a513a6736"
                        }
         }
```

GetDappState: 获得子链 DAPP 合约的状态

```
SubChainAddr: 子链合约地址
Sender: 子链合约地址创建者地址
Body: {"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetDappState",
                "params":{"SubChainAddr":"0x1195cd9769692a69220312e95192e0dcb6a4ec09",
                        "Sender":"0x87e369172af1e817ebd8d63bcd9f685a513a6736"
                 }
         }
```

getContractInfo: 获得子链 DAPP 智能合约全局变量

```
SubChainAddr: 子链合约地址
Reqtype:  查询类型 0: 查看合约全部变量 , 1: 查看合约某一个数组变量 , 2: 查看合约某一个
mapping 变量 , 3: 查看合约某一个结构体变量, 4: 查看合约某一简单类型变量 (单倍长度存储的变量)
, 5: 查看合约某一变长变量 (如 string、bytes)
Storagekey: 十六进制字符串, 查询的变量在合约里面的 index , 查询全部变量时可以不填
Position: 十六进制字符串, 当 Reqtype==1 时, Position 为数组维度 (从 0 开始); 当 Reqtype==2
时, Position 为 mapping 下标
Structformat: 针对结构体变量, 1: single (简单类型变量单倍长度存储的变量) , 2: list (简单类型
数组变量) 3: string 变长变量 (如 string、bytes), 若结构变量为 ContractInfoReq, Structformat␣
↪= []byte{ '1' ,' 3' ,' 3' ,' 3' }

获取合约 index 1 的 address 对应 Body:
{"jsonrpc":"2.0","id":0,"method":"ScsRPCMethod.GetContractInfo",
"params":{"subChainAddr":"0x1195cd9769692a69220312e95192e0dcb6a4ec09",
        "Request":[{"Reqtype":4,
                  "Storagekey":[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
↪0,0,0,1],
                  "Position":[],
                  "Structformat":[]}
                  ]
        }
```

(下页继续)

(续上页)

```
}
```

CHAPTER 7

子链节点的更替

CHAPTER 8

关闭子链

CHAPTER 9

母子链货币交互简介

墨客支持有币子链，并且提供母链货币和子链原生币之间的兑换（充提）。

当前，墨客提供三种类型的母子链货币交互，以下一一介绍

9.1 母链 MOAC 和子链原生币交互

这个是最基础的一种货币兑换。使用者可以在主链上充值 MOAC，然后最早在下一个 flush 周期在子链上获取子链原生币。同理，使用者可以提出子链原生币，并最早在下一个 flush 周期获得主链 MOAC。

9.1.1 子链部署准备

参考子链部署章节，完成部署子链的准备工作。

```
子链操作账号: 0x87e369172af1e817ebd8d63bcd9f685a513a6736
vnode 矿池合约地址: 0x22f141dcc59850707708bc90e256318a5fe0b928
vnode 代理地址: 0xf103bc1c054babcecd13e7ac1cf34f029647b08c    192.168.10.209:50062
子链矿池合约地址: 0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac
scs0: 0x075447a6df1fde4f39243bc67a945312ff36c193  确保启动并加入子链矿池
scs1: 0x7932f827c90c5f06c0177f642a07edfa73ee3044  确保启动并加入子链矿池
scs monitor: 0xa5966600efb221097ce6a8ba1dc6eb1d5b43ef83
```

9.1.2 erc20 部署

这类似一个管理合约，负责和子链货币之间的转换

参考官方示例的 erc20 合约 DirectExchangeToken.sol 调用示例：注意参数 tokensupply 为 erc20 的 token 总量，exchangerate 为 moac 与子链 token 的兑换比率

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> input = {'': fs.readFileSync('DirectExchangeToken.sol', 'utf8'), 'StandardToken.sol':␣
↪fs.readFileSync('StandardToken.sol', 'utf8'), 'ERC20.sol': fs.readFileSync('ERC20.sol',
↪ 'utf8'), 'SafeMath.sol': fs.readFileSync('SafeMath.sol', 'utf8')};
> output = solc.compile({sources: input}, 1);
> abi = output.contracts[':DirectExchangeToken'].interface;
> bin = output.contracts[':DirectExchangeToken'].bytecode;
> tokensupply = 100000000;      // total supply
> exchangerate = 100;
> directexchangetokenContract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> dtoken = directexchangetokenContract.new( tokensupply, exchangerate,{ from: chain3.mc.
↪accounts[0],  data: '0x' + bin,  gas:'9000000'} , function (e, contract){console.log(
↪'Contract address: ' + contract.address + ' transactionHash: ' + contract.
↪transactionHash); });
```

部署完毕后，获得 erc20 合约地址 0x5fcb383cf80a9d2961f4d8ba0a4b9e4a34157da7

9.1.3 subchainbase 部署

注意这个子链合约在官方基础上进行了修改，增加了 setToken 和 MintToken 相关的若干方法。部署 SubChainBase.sol 示例:

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> input = {'': fs.readFileSync('SubChainBase.sol', 'utf8'), 'SubChainProtocolBase.sol':␣
↪fs.readFileSync('SubChainProtocolBase.sol', 'utf8')};
> output = solc.compile({sources: input}, 1);
> abi = output.contracts[':SubChainBase'].interface;
> bin = output.contracts[':SubChainBase'].bytecode;
```

(下页继续)

(续上页)

```
> proto = '0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac' ;    // 子链矿池合约
> vnodeProtocolBaseAddr = '0x22f141dcc59850707708bc90e256318a5fe0b928' ;       // Vnode
矿池合约
> min = 1 ;                    // 子链需要 SCS 的最小数量
> max = 10 ;            // 子链需要 SCS 的最大数量
> thousandth = 1 ;                       // 千分之几
> flushRound = 40 ;             // 子链刷新周期  单位是主链 block 生成对应数量的时间 (10 秒
一个 block)
> SubChainBaseContract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> SubChainBase = SubChainBaseContract.new( proto, vnodeProtocolBaseAddr, min, max,␣
↪thousandth, flushRound,{ from: chain3.mc.accounts[0],  data: '0x' + bin,  gas:'9000000
↪'} , function (e, contract){console.log('Contract address: ' + contract.address + '␣
↪transactionHash: ' + contract.transactionHash); });
```

部署完毕后, 获得子链合约地址 0xe9463e215315d6f1e5387a161868d7d0a4db89e8

9.1.4 ERC20 初始化

在子链的 register open 前先调用 subchainbase 的 setToken 方法

参数: DirectExchangeToken 的合约地址

根据参数 DirectExchangeToken 的地址, 实例化合约内部的 DirectExchangeToke 对象, 并将 DirectExchangeToken 合约的 totalsupply 复制给合约变量 BALANCE

调用示例:

```
根据 ABI chain3.sha3("setToken(address)") =␣
↪0x144fa6d7f374f07750ac69fe5e8b4f8a614ab02482bcb9f0fa3f2c98943860c6
        取前 4 个字节 0x144fa6d7
        参数传 erc20 合约地址  5fcb383cf80a9d2961f4d8ba0a4b9e4a34157da7  (前面补 24 个 0,
凑足 32 个字节)
        0x144fa6d70000000000000000000000005fcb383cf80a9d2961f4d8ba0a4b9e4a34157da7
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> data = '0x144fa6d70000000000000000000000005fcb383cf80a9d2961f4d8ba0a4b9e4a34157da7'
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: data});
```

验证:

访问子链合约的 BALANCE 为 ERC20 的 totalsupply

```
> chain3.mc.getStorageAt(subchainaddr,0x30)   // 注意 BALANCE 变量在合约中变量定义的位置
(16 进制)
```

然后调用 DirectExchangeToken 的 updateOwner 方法将合约 owner 由发布者改为 subchainbase 地址这样子链合约就可以操控 ERC20 合约，并进行方法调用。

```
根据 ABI chain3.sha3('updateOwner(address)') =␣
↪0x880cdc3156e7a1d7441d29b6ec3cab090473e3e84f7cc83d0bdcd0a696e8064b
        取前 4 个字节 0x880cdc31
        参数传 subchainbase 合约地址  e9463e215315d6f1e5387a161868d7d0a4db89e8  (前面补
24 个 0,  凑足 32 个字节)
        0x880cdc3100000000000000000000000e9463e215315d6f1e5387a161868d7d0a4db89e8
> erc20addr = '0x5fcb383cf80a9d2961f4d8ba0a4b9e4a34157da7';
> data = '0x880cdc3100000000000000000000000e9463e215315d6f1e5387a161868d7d0a4db89e8'
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: erc20addr, gas:
↪"2000000", gasPrice: chain3.mc.gasPrice, data: data});
```

验证:

访问 erc20 合约的 owner 是否由部署账号 0x87e369172af1e817ebd8d63bcd9f685a513a6736 改为子链合约地址 0xe9463e215315d6f1e5387a161868d7d0a4db89e8

```
> chain3.mc.getStorageAt(erc20addr,0x03)   // 注意 owner 变量在合约中变量定义的位置 (16 进
制)
```

9.1.5 子链注册 scs

子链开放注册

调用合约里的函数 addFund

```
根据 ABI chain3.sha3("addFund()") =␣
↪0xa2f09dfa891d1ba530cdf00c7c12ddd9f6e625e5368fff9cdf23c9dc0ad433b1
        取前 4 个字节 0xa2f09dfa
> amount = 20;
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:chain3.toSha(amount,'mc
↪'), to: subchainaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0xa2f09dfa'}
↪);
```

可以通过查询余额进行验证

```
> chain3.mc.getBalance('0xe9463e215315d6f1e5387a161868d7d0a4db89e8')
```

然后调用调用合约里的函数 registerOpen 开放注册 (按子链矿池合约中 SCS 注册先后排序进行选取)

```
根据 ABI chain3.sha3("registerOpen()") =␣
↪0x5defc56ce78f178d760a165a5528a8e8974797e616a493970df1c0918c13a175
        取前 4 个字节 0x5defc56c
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x5defc56c'});
```

验证:

访问子链合约的 nodeCount

```
> chain3.mc.getStorageAt(subchainaddr,0x0e)  // 注意 nodeCount 变量在合约中变量定义的位置
(16 进制)
```

等到两个 scs 都注册完毕后，即注册 SCS 数目大于等于子链要求的最小数目时，调用子链合约里的函数 registerClose 关闭注册

```
根据 ABI chain3.sha3("registerClose()") =␣
↪0x69f3576fc10c82561bd84b0045ee48d80d59a866174f2513fdef43d65702bf70
        取前 4 个字节 0x69f3576f
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x69f3576f'});
```

验证:

访问子链合约的 registerFlag 为 0

```
> chain3.mc.getStorageAt(subchainaddr,0x14)     // 注意 registerFlag 变量在合约中变量定义
的位置 (16 进制)
```

同时观察 scs 的 concole 界面，scs 开始出块即成功完成部署子链。

9.1.6 dapp 合约部署

部署 dapp 合约，dechat 是一个官方示例，注意合约有两个参数，分别是版主的账号和开发者的账号

部署示例:

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'dechat1.0.5.sol';
> contract = fs.readFileSync(solfile, 'utf8');
> output = solc.compile(contract, 1);
> abi = output.contracts[':DeChat'].interface;
> bin = output.contracts[':DeChat'].bytecode;
> bin += '000000000000000000000000' + '87e369172af1e817ebd8d63bcd9f685a513a6736';  // 版
主的账号
> bin += '000000000000000000000000' + '87e369172af1e817ebd8d63bcd9f685a513a6736';  // 开
发者的账号
> amount = chain3.toSha(100000000,'mc') // 要与部署 DirectExchangeToken 合约地址的 amount
参数一致
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction({from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas:0, shardingFlag: "0x1", data: '0x' + bin, nonce: 0, via: via, });
```

验证:

合约部署成功后，Nonce 值应该是 1

9.1.7 dapp 充值

调用 subchainbase 的 buyMintToken 方法充值，用户账号为发出 sendTransaction 的账号数量为 sendTransaction

buyMintToken 方法首先调用 DirectExchangeToke 对象的 buyMintToken 方法按交易记录的 amount 扣除 DirectExchangeToke 合约本身的 token 数量，增加用户账号对应 token 的数量

然后调用 transfer 给 DirectExchangeToken 合约地址转对应的 moac

再调用 DirectExchangeToke 对象的 requestEnterMicrochain 方法（传 subchainbase 地址和对应 token 数量），减去用户账号地址的对应 token 数量，增加 DirectExchangeToke 合约本身的 token 总量

最后将用户账号和对应 token 数量，加入推送结构体发至子链，等待一轮 flush 后，充值会进入到子链

调用示例:

```
根据 ABI chain3.sha3("buyMintToken()") =␣
↪0x6bbded701cd78dee9626653dc2b2e76d3163cc5a6f81ac3b8e69da6a057824cb
        取前 4 个字节 0x6bbded70
> amount = 100;
> subchainaddr = '0xe9463e215315d6f1e5387a161868d7d0a4db89e8';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value: chain3.toSha(amount,
↪'mc'), to: subchainaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data:
↪'0x6bbded70'});
```

验证:

检查账号的 moac 是否减少: > chain3.mc.getBalance(chain3.mc.accounts[0])

检查子链的 token 是否增加: 调用 monitor 的方法 ScsRPCMethod.GetBalance 获得子链 token

9.1.8 dapp 提币

调用 dapp 合约的 redeemFromMicroChain 方法，用户账号为发出 sendTransaction 的账号数量为 sendTransactio

redeemFromMicroChain 方法将用户账号和对应 token 数量加入推送结构体 redeem，等待一轮 flush 后，自动会调用子链合约的 redeemFromMicroChain 方法

然后子链合约自动调用 DirectExchangeToke 对象的 redeemFromMicroChain 方法，扣除 DirectExchangeToke 合约本身的 token 数量，增加用户账号对应 token 的数量

最后自动调用 subchainbase 的 sellMintToken 方法，自动调用 DirectExchangeToke 对象的 sellMintToken 兑换 moac

调用示例:

```
根据 ABI chain3.sha3("redeemFromMicroChain()") =␣
↪0x89739c5bf1ef36273bf0e7aeb59ffe71213a58e1f01965e75662cb21b03abb13
取前 4 个字节 0x89739c5b
> nonce = 1       // 调用 ScsRPCMethod.GetNonce 获得
> subchainaddr = '0x1195cd9769692a69220312e95192e0dcb6a4ec09';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
```

(下页继续)

(续上页)

```
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { nonce: nonce, from: chain3.mc.accounts[0], value:0, to:␣
↪subchainaddr, gas:0, shardingFlag:'0x1', data: '0x89739c5b', via: via,});
```

验证：

检查账号的 moac 是否增加: > chain3.mc.getBalance(chain3.mc.accounts[0])

检查子链的 token 是否减少: 调用 monitor 的方法 ScsRPCMethod.GetBalance 获得子链 token

9.2 母链 ERC20 和子链原生币交互

这是非常通用的一种货币兑换。使用者可以使用预先已经部署好的 ERC20，或者当场部署一个主链 ERC20，和子链的原生币进行兑换。

9.2.1 子链部署准备

参考子链部署章节，完成部署子链的准备工作。

```
子链操作账号: 0x87e369172af1e817ebd8d63bcd9f685a513a6736
vnode 矿池合约地址: 0x22f141dcc59850707708bc90e256318a5fe0b928
vnode 代理地址: 0xf103bc1c054babcecd13e7ac1cf34f029647b08c    192.168.10.209:50062
子链矿池合约地址: 0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac
scs0:   0xd81043d85c9c959d2925958c54c1a49c7bfd1fc8  确保启动并加入子链矿池
scs1:   0xe767059d768fcef12e527fab63fda68cc13e24b3  确保启动并加入子链矿池
scs monitor:    0x0964e5d73d6a40f2fc707aa3e1361028a34923f0
```

9.2.2 erc20 部署

默认一个标准的 erc20 合约，通过 allowance，transferFrom，balanceOf，transfer 等标准的方法支持货币的转移。

参考官方示例的 erc20 合约 erc20.sol，默认 decimals 为 6，totalSupply 为 100000000 乘以 10 的 6 次方。调用示例：

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'erc20.sol';
> contract = fs.readFileSync(solfile, 'utf8');
```

(下页继续)

(续上页)

```
> output = solc.compile(contract, 1);
> abi = output.contracts[':TestCoin'].interface;
> bin = output.contracts[':TestCoin'].bytecode;
> erc20Contract = chain3.mc.contract(JSON.parse(abi));
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> dtoken = erc20Contract.new( { from: chain3.mc.accounts[0],  data: '0x' + bin,  gas:
↪'9000000'} , function (e, contract){console.log('Contract address: ' + contract.
↪address + ' transactionHash: ' + contract.transactionHash); });
```

部署完毕后, 获得 erc20 合约地址 0x5042086887a86151945d2c2bb60628addf49d48c

验证：调用合约 balanceOf 方法查询部署者的余额，应该是 10 的 14 次方

> contractInstance = erc20Contract.at('0x5042086887a86151945d2c2bb60628addf49d48c') > contractInstance.balanceOf.call('0x87e369172af1e817ebd8d63bcd9f685a513a6736')

9.2.3 subchainbase 部署

注意这个子链合约在官方基础上进行了修改，增加了 erc20 合约地址和兑换比例的参数部署 SubChainBase.sol 示例:

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> input = {'': fs.readFileSync('SubChainBase.sol', 'utf8'), 'SubChainProtocolBase.sol
↪':fs.readFileSync('SubChainProtocolBase.sol', 'utf8')};
> output = solc.compile({sources: input}, 1);
> abi = output.contracts[':SubChainBase'].interface;
> bin = output.contracts[':SubChainBase'].bytecode;
> proto = '0xe42f4f566aedc3b6dd61ea4f70cc78d396130fac' ;    // 子链矿池合约
> vnodeProtocolBaseAddr = '0x22f141dcc59850707708bc90e256318a5fe0b928' ;       // Vnode
矿池合约
> ercAddr = '0x5042086887a86151945d2c2bb60628addf49d48c';     // erc20 合约地址
> ercRate = 100;    // 兑换比率
> min = 1 ;                    // 子链需要 SCS 的最小数量
> max = 10 ;           // 子链需要 SCS 的最大数量
> thousandth = 1 ;                      // 千分之几
> flushRound = 40 ;            // 子链刷新周期  单位是主链 block 生成对应数量的时间 (10 秒
一个 block)
> SubChainBaseContract = chain3.mc.contract(JSON.parse(abi));
```

(下页继续)

(续上页)

```
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> SubChainBase = SubChainBaseContract.new( proto, vnodeProtocolBaseAddr, ercAddr,␣
↪ercRate, min, max, thousandth, flushRound,{ from: chain3.mc.accounts[0],  data: '0x' +␣
↪bin,  gas:'9000000'} , function (e, contract){console.log('Contract address: ' +␣
↪contract.address + ' transactionHash: ' + contract.transactionHash); });
```

部署完毕后, 获得子链合约地址 0xb877bf4e4cc94fd9168313e00047b77217760930

验证:

访问子链合约的 BALANCE 为 ERC20 的 totalsupply

```
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> chain3.mc.getStorageAt(subchainaddr,0x30)   // 注意 BALANCE 变量在合约中变量定义的位置
(16 进制)  1ed09bead87c0378d8e6400000000
应该是 10 的 34 次方  (14 + 2 + 18)
```

9.2.4 子链注册 scs

子链开放注册

调用合约里的函数 addFund

```
根据 ABI chain3.sha3("addFund()") =␣
↪0xa2f09dfa891d1ba530cdf00c7c12ddd9f6e625e5368fff9cdf23c9dc0ad433b1
        取前 4 个字节 0xa2f09dfa
> amount = 20;
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:chain3.toSha(amount,'mc
↪'), to: subchainaddr, gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0xa2f09dfa'}
↪);
```

可以通过查询余额进行验证

```
> chain3.mc.getBalance('0xb877bf4e4cc94fd9168313e00047b77217760930')
```

然后调用调用合约里的函数 registerOpen 开放注册 (按子链矿池合约中 SCS 注册先后排序进行选取)

```
根据 ABI chain3.sha3("registerOpen()") =␣
↪0x5defc56ce78f178d760a165a5528a8e8974797e616a493970df1c0918c13a175
        取前 4 个字节 0x5defc56c
```

(下页继续)

(续上页)

```
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x5defc56c'});
```

验证:

访问子链合约的 nodeCount

```
> chain3.mc.getStorageAt(subchainaddr,0x0e)  // 注意 nodeCount 变量在合约中变量定义的位置
(16 进制)
```

等到两个 scs 都注册完毕后，即注册 SCS 数目大于等于子链要求的最小数目时，调用子链合约里的函数 registerClose 关闭注册

```
根据 ABI chain3.sha3("registerClose()") =␣
↪0x69f3576fc10c82561bd84b0045ee48d80d59a866174f2513fdef43d65702bf70
        取前 4 个字节 0x69f3576f
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value:0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: '0x69f3576f'});
```

验证:

访问子链合约的 registerFlag 为 0

```
> chain3.mc.getStorageAt(subchainaddr,0x14)     // 注意 registerFlag 变量在合约中变量定义
的位置 (16 进制)
```

同时观察 scs 的 concole 界面，scs 开始出块即成功完成部署子链。

9.2.5 dapp 合约部署

部署 dapp 合约，dechat 是一个官方示例，注意合约有两个参数，分别是版主的账号和开发者的账号

部署示例:

```
> chain3 = require('chain3')
> solc = require('solc')
> chain3 = new chain3();
> chain3.setProvider(new chain3.providers.HttpProvider('http://localhost:8545'));
> solfile = 'dechat1.0.5.sol';
```

(下页继续)

(续上页)

```
> contract = fs.readFileSync(solfile, 'utf8');
> output = solc.compile(contract, 1);
> abi = output.contracts[':DeChat'].interface;
> bin = output.contracts[':DeChat'].bytecode;
> bin += '000000000000000000000000' + '87e369172af1e817ebd8d63bcd9f685a513a6736';  // 版
主的账号
> bin += '000000000000000000000000' + '87e369172af1e817ebd8d63bcd9f685a513a6736';  // 开
发者的账号
> amount = chain3.toSha(100000000,'mc') // dapp 合约地址的 token 数量，与 erc20 的 amount
一致
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction({from: chain3.mc.accounts[0], value:amount, to: subchainaddr,
↪ gas:0, shardingFlag: "0x1", data: '0x' + bin, nonce: 0, via: via, });
```

验证:

在 scs 的 _logs 目录下搜索日志文件，查找”created contract address”，找到 dapp 合约地址:6ab296062d8a147297851719682fb5ffe081f1d3

调用 monitor 的方法 ScsRPCMethod.GetBalance 查询对应 dapp 合约地址的余额，应该等于 erc20 总量。

9.2.6 dapp 充值

调用 subchainbase 的 buyMintToken 方法充值，用户账号为发出 sendTransaction 的账号，参数分别为子链合约地址

buyMintToken 方法首先调用 erc20 合约的 allowance 检查授权，并调用 transferFrom 方法将 token 从用户账号地址转到合约地址

然后自动调用 requestEnterMicrochain 方法，将用户账号和对应 token 数量，加入推送结构体发至子链，等待一轮 flush 后，充值会进入到子链

调用示例:

```
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> data = contractInstance.buyMintToken.getData(subchainaddr, 100)
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
```

(下页继续)

(续上页)

```
> chain3.mc.sendTransaction( { from: chain3.mc.accounts[0], value: 0, to: subchainaddr,␣
↪gas: "2000000", gasPrice: chain3.mc.gasPrice, data: data});
```

验证：

检查账号的 erc20 token 是否减少: 调用 erc20 合约的 balanceOf 方法

检查子链的 token 是否增加: 调用 monitor 的方法 ScsRPCMethod.GetBalance 获得子链 token

9.2.7 dapp 提币

调用 dapp 合约的 redeemFromMicroChain 方法，用户账号为发出 sendTransaction 的账号数量为 sendTransactio

redeemFromMicroChain 方法将用户账号和对应 token 数量加入推送结构体 redeem，等待一轮 flush 后，自动会调用子链合约的 redeemFromMicroChain 方法

调用 erc20 合约的 transfer 给用户账号转对应的 token 数量

调用示例：

```
根据 ABI chain3.sha3("redeemFromMicroChain()") =␣
↪0x89739c5bf1ef36273bf0e7aeb59ffe71213a58e1f01965e75662cb21b03abb13
取前 4 个字节 0x89739c5b
> nonce = 1       // 调用 ScsRPCMethod.GetNonce 获得
> subchainaddr = '0xb877bf4e4cc94fd9168313e00047b77217760930';
> via = '0xf103bc1c054babcecd13e7ac1cf34f029647b08c';
> chain3.personal.unlockAccount(chain3.mc.accounts[0], '123456');
> chain3.mc.sendTransaction( { nonce: nonce, from: chain3.mc.accounts[0], value:0, to:␣
↪subchainaddr, gas:0, shardingFlag:'0x1', data: '0x89739c5b', via: via,});
```

验证：

检查账号的 erc20 token 是否增加: 调用 erc20 合约的 balanceOf 方法

检查子链的 token 是否减少: 调用 monitor 的方法 ScsRPCMethod.GetBalance 获得子链 token

9.3 ATO 方式

TODO

CHAPTER 10

其他

- genindex
- modindex
- search