hyperchain Documentation

发布

Hyperchain Corp.

12月 14, 2017

快速入门:

CHAPTER 1

准备工作

1.1 操作系统版本要求

以下图表说明了Hyperchain对于不同操作系统的版本要求。

不同平台版本要求

操作系统	系统版本	系统架构
RHEL	6 或更新	amd64, 386
CentOS	6 或更新	amd64, 386
SLES	11SP3 或更新	amd64, 386
Ubuntu	14.04 或更新	amd64, 386
macOS	10.8 或更新	amd64, 386

1.2 安装Go语言开发环境

因为Hyperchain使用Go语言来实现它的各个组件，所以需要安装Go语言开发环境。

1.2.1 下载 Go

Go为Mac OS X、Linux和Windows提供二进制发行版。如果您使用的是不同的操作系统，您可以下载Go源代码并从源代码安装。

在这里下载适用于您的平台的最新版本Go：下载 - 请下载 `1.7.x` 或更新

1.2.2 安装 Go

请按照对应于您的平台的步骤来安装Go环境：安装Go，推荐使用默认配置安装。

- 对于Mac OS X 和 Linux操作系统，默认情况下Go会被安装到`/usr/local/go/`，并且将环境变量`GOROOT`设置为该路径`/usr/local/go`.

```
export GOROOT=/usr/local/go
```

- 同时，请添加路径 `GOROOT/bin` 到环境变量`PATH`中，可以使Go工具正常执行。

```
export PATH=$PATH:$GOROOT/bin
```

1.2.3 设置 GOPATH

您的Go工作目录 (GOPATH) 是用来存储您的Go代码的地方，您必须要将他跟您的Go安装目录区分开 (GOROOT)。

以下命令是用了设置您的GOPATH环境变量的，您也可以参考Go官方文档，来获得更详细的内容: https://golang.org/doc/code.html.

- 对于 Mac OS X 和 Linux 操作系统 将 GOPATH 环境变量设置为您的工作路径:

```
export GOPATH=$HOME/go
```

- 同时添加路径 GOPATH/bin 到环境变量PATH中，可以使编译后的Go程序正常执行。

```
export PATH=$PATH:$GOPATH/bin
```

- 由于我们将在Go中进行一系列编码，您可以将以下内容添加到您的~/.bashrc文件中:

```
export GOROOT=/usr/local/go
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin:$GOROOT/bin
```

1.2.4 检查Go安装结果

创建和运行这里描述的hello.go应用: https://golang.org/doc/install#testing.

如果您正确设置了Go运行环境，您应该能够从任何目录运行hello程序，并看到程序成功执行。

1.3 安装 Go vendor

Go vendor是管理包及其依赖项的工具。此工具将依赖的包复制到项目的vendor目录中，并将其版本记录在名为vendor.json的文件中。

1.3.1 安装命令

```
go get -u github.com/kardianos/govendor
```

1.3.2 检查Go vendor安装结果

为了要验证您的govendor安装正确，可以通过查看govendor版本信息来检验。

在命令提示符下，键入以下命令并确保您看到了govendor版本信息:

```
$ govendor --version
v1.0.9
```

1.3.3 更多信息

您可以转到项目的主页了解更多细节。 - Go vendor

1.4 安装合约编译器(可选)

Hyperchain 支持用Solidity编写的智能合约，然后将它编译为字节码并部署到区块链中。

鉴于我们是用Solidity语言编写的合约，所以需要确保我们已经安装名为`solc`的合约编译器。

我们已经在源码中提供了一些平台的通用安装包，您可以直接使用他们来快速安装 `solc` ，您也可以参考官方文档来完成安装 - 安装Solidity.

CHAPTER 2

快速入门

如果您还没有完成上一篇中提到的所有准备工作, 请先完成它们，再继续下一步的操作。本快速入门告诉您如何从源代码构建Hyperchain，如何启动一个Hyperchain集群。

2.1 编译Hyperchain

2.1.1 拉取代码

克隆代码到您的GOPATH工作目录下：

```
mkdir -p $GOPATH/src/github.com/hyperchain
cd $GOPATH/src/github.com/hyperchain
git clone https://github.com/hyperchain/hyperchain
```

2.1.2 编译代码

请确保您已经安装了正确的Go工具，如有问题，请参见准备工作.

编译Hyperchain：

```
cd $GOPATH/src/github.com/hyperchain/hyperchain
govendor build
```

您也可以执行 `go build`来编译。

2.2 启动 Hyperchain

由于Hyperchain集群需要至少4个节点建立一个BFT系统，我们建议用以下几种模式启动Hyperchain节点：

- 单服务器模式 - 本地运行4个节点
- 多服务器模式 - 多服务器运行4个节点

2.2.1 单服务器模式 - 本地运行4个节点

我们提供了一个工具脚本名为local.sh, 可以用来快速部署运行本地4个Hyperchain节点。

```
cd $GOPATH/src/github.com/hyperchain/hyperchain/scripts
./local.sh
```

如果脚本输出以下信息，说明Hyperchian节点已经正常运行了。

```
$./local.sh
...
...
start up node 1 ... done
start up node 2 ... done
start up node 3 ... done
start up node 4 ... done
```

2.2.2 多服务器模式 - 多服务器运行4个节点

SSH免密通路

因为我们使用的server.sh工具脚本在执行ssh操作时，会提示输入远程服务器 的密码，所以我们建议您打通与远程服务器之间的SSH免密通路。

1. 本地节点生成SSH秘钥对，密码请设置为空：

```
ssh-keygen

Generating public/private key pair.
Enter file in which to save the key (/home/hyperchain/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/hyperchain/.ssh/id_rsa.
Your public key has been saved in /home/hyperchain/.ssh/id_rsa.pub.
```

2. 将SSH公钥拷贝到 Hyperchain 节点, 请用您远程服务器上的用户名代替以下命令中的 {username}

```
ssh-copy-id {username}@node1
ssh-copy-id {username}@node2
ssh-copy-id {username}@node3
ssh-copy-id {username}@node4
```

分发部署 Hyperchain

我们提供了一个工具脚本名为server.sh, 可以用来快速分发到4个节点部署运Hyperchain。

1. 首先请您将4台服务器的IP地址填入到 hyperchain/scripts目录下的serverlist.txt文件中。

格式如下所示：

```
$ cat $GOPATH/src/github.com/hyperchain/hyperchain/scripts/serverlist.txt
172.16.1.101
172.16.1.102
172.16.1.103
172.16.1.104
```

2. 使用server.sh启动远程多个Hyperchain节点。

```
cd $GOPATH/src/github.com/hyperchain/hyperchain/scripts
./server.sh
```

如果脚本输出以下信息，说明Hyperchian节点已经正常运行了。

```
$./server.sh
...
...
start up node 1 ... done
start up node 2 ... done
start up node 3 ... done
start up node 4 ... done
```

CHAPTER 3

Hyperchain 使用示例

在这一节中，我们将会用一些例子，来介绍一下Hyperchain平台的简单使用方法。

3.1 HyperCli

我们推荐使用HyperCli作为您的管理工具。

HyperCli 是一个基于Hyperchain平台开发的命令行工具，方便管理员操作Hyperchain平台。HyperCli提供了丰富的命令行接口，下面我们会介绍它的智能合约、交易处理等相关功能。

3.2 合约样例1 - Set/Get Hash

这里是一个实现setHash和getHash功能的智能合约。

```
contract Anchor{
    mapping(bytes32 => bytes32) hashMap;

    function setHash(bytes32 key,bytes32 value) returns(bool,bytes32){
        if(hashMap[key] != 0x0){
            return (false,"the key exist");
        }
        hashMap[key] = value;
        return (true,"Success");
    }

    function getHash(bytes32 key) returns(bool,bytes32,bytes32){
        if(hashMap[key] == 0x0){
            return (false,"the key is not exist",0x0);
        }
        return (true,"Success",hashMap[key]);
    }
}
```

3.2.1 编译合约

如果您已经安装了solc，您只需通过一个简单的命令就能得到合约的字节码。如果您没有安装编译器，也可以直接使用以下的字节码，该字节码是本合约的编译结果。

字节码

```
0x606060405261015c806100126000396000f3606060405260e060020a60003504633cf5040a81146100295780063d7fa1
```

假设您的智能合约文件名为sample1.sol, 您可以通过以下命令获得合约字节码:

```
solc --bin sample1.sol
```

3.2.2 部署合约

HyperCli提供了一个合约部署的功能，以下是该功能的参数:

```
$ ./hypercli contract deploy --help

NAME:
   hypercli contract deploy - Deploy a contract

USAGE:
   hypercli contract deploy [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --directory value, -d value  specify the contract file directory
```

您可以将合约的字节码作为--payload 选项的值，使用以下的命令来部署合约:

```
./hypercli contract deploy --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x606060405261015c806100126000396000f3606060405260e060020a60003504633cf5040a81146100295780063d7fa
```

该命令的意思是HyperCli使用地址为000f1a7a08ccc48e5d30f80850cf1cf283aa3abd的账户来部署合约。

如果命令执行正确，您将看到以下输出结果:

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xb1b7d4f083ac65679ddd31a9b864fc8ca1ec75eee2f7a46cca1b223eae94527c","vmType":
↪"EVM","contractAddress":"0xbbe2b6412ccf633222374de8958f2acc76cda9c9","gasUsed
↪":69660,"ret":
↪"0x606060405260e060020a60003504633cf5040a81146100295780063d7fa10071461007b575b610002565b34610002
↪","log":[]}}
```

从结果中可以得到部署后的合约地址，在这个例子中，合约地址为:

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

之后它将会被用于合约调用的操作中。

3.2.3 调用合约

HyperCli提供了一个合约调用的功能，以下是该功能的参数：

```
$ ./hypercli contract invoke --help

NAME:
   hypercli contract invoke - Invoke a contract

USAGE:
   hypercli contract invoke [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --to value, -t value         specify the contract address
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --args value, -a value       specify the args of invoke contract
```

在本例中，您至少需要指定两个参数的值，才能调用合约中的函数：

- payload选项: 函数调用的字节码
- to选项: 合约地址

我们提供了一些函数调用的字节码，您可以直接使用它们。

setHash

调用setHash函数，设置key1 = value1, 以下是该调用的字节码：

```
0xd7fa10076b65793100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000076616c756531000000000000
```

以下是该合约的地址：

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

您可以通过以下命令调用合约：

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0xd7fa10076b65793100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000076616c756531000000000000
↪--to 0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xa28350777a964f5ab6f4ef355131c0c241388ac6e8548c191aa5b3b94af95571","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":20477,"ret":
↪"0x0000000000000000000000000000000000000000000000000000000000000000153756363657373000000000000000000000000000000000000000000000000
↪","log":[]}}
```

getHash

调用getHash函数，获得key1对应的value值， 以下是该调用的字节码：

```
0x3cf5040a6b65793100000000000000000000000000000000000000000000000000000000
```

以下是该合约的地址:

```
0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

您可以通过以下命令调用合约:

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x3cf5040a6b65793100000000000000000000000000000000000000000000000000000000 --to␣
↪0xbbe2b6412ccf633222374de8958f2acc76cda9c9
```

如果命令执行正确，您将看到以下输出结果:


```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0x185dba5451ace5ffcf4c11d10968e1e4ed299eb78ca6ddda65539dfca2fc56df","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":523,"ret":
↪"0x000000000000000000000000000000000000000000000000000000000000000153756363657373000000000000000
↪","log":[]}}
```


3.3 合约样例2 - Simulate Bank

这里是另一个实现了资产管理的合约示例。

```
contract SimulateBank{
    address owner;
    bytes32 bankName;
    uint bankNum;
    bool isInvalid;
    mapping(address => uint) public accounts;
    function SimulateBank( bytes32 _bankName,uint _bankNum,bool _isInvalid){
        bankName = _bankName;
        bankNum = _bankNum;
        isInvalid = _isInvalid;
        owner = msg.sender;
    }
    function issue(address addr,uint number) returns (bool){
        if(msg.sender==owner){
            accounts[addr] = accounts[addr] + number;
            return true;
        }
        return false;
    }
    function transfer(address addr1,address addr2,uint amount) returns (bool){
        if(accounts[addr1] >= amount){
            accounts[addr1] = accounts[addr1] - amount;
            accounts[addr2] = accounts[addr2] + amount;
            return true;
        }
        return false;
    }
    function getAccountBalance(address addr) returns(uint){
        return accounts[addr];
    }
}
```

3.3.1 编译合约

如果您已经安装了solc，您只需通过一个简单的命令就能得到合约的字节码。如果您没有安装编译器，也可以直接使用以下的字节码，该字节码是本合约的编译结果。

字节码

```
0x6060604052604051606080610200833950606c0604052516080516a0516001839055600282905560038054f0100000
```

假设您的智能合约文件名为sample2.sol,您可以通过以下命令获得合约字节码:

```
solc --bin sample2.sol
```

3.3.2 部署合约

HyperCli提供了一个合约部署的功能，以下是该功能的参数:

```
$ ./hypercli contract deploy --help

NAME:
   hypercli contract deploy - Deploy a contract

USAGE:
   hypercli contract deploy [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --directory value, -d value  specify the contract file directory
```

您可以将合约的字节码作为--payload 选项的值，使用以下的命令来部署合约:

```
./hypercli contract deploy --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x6060604052604051606080610200833950606c0604052516080516a0516001839055600282905560038054f010000
```

该命令的意思是HyperCli使用地址为000f1a7a08ccc48e5d30f80850cf1cf283aa3abd的账户来部署合约。

如果命令执行正确，您将看到以下输出结果:

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0x6790126ca4c072f53d1684dff9e080098db931358d8eca04c833373ae580ed9e","vmType":
↪"EVM","contractAddress":"0xbbe2b6412ccf633222374de8958f2acc76cda9c9","gasUsed
↪":109363,"ret":
↪"0x606060405260e060020a60003504635e5c06e2811461003f578063867904b41461005c57806393423e9c146100a8
↪","log":[]}}
```

从结果中可以得到部署后的合约地址，在这个例子中，合约地址为:

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

之后它将会被用于合约调用的操作中。

3.3.3 调用合约

HyperCli提供了一个合约调用的功能，以下是该功能的参数：

```
$ ./hypercli contract invoke --help

NAME:
   hypercli contract invoke - Invoke a contract

USAGE:
   hypercli contract invoke [command options] [arguments...]

OPTIONS:
   --namespace value, -n value  specify the namespace, default to global (default:
↪"global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --payload value, -p value    specify the contract payload
   --to value, -t value         specify the contract address
   --extra value, -e value      specify the extra information
   --simulate, -s               simulate execute or not, default to false
   --args value, -a value       specify the args of invoke contract
```

在本例中，您至少需要指定两个参数的值，才能调用合约中的函数：

- payload选项: 函数调用的字节码
- to选项: 合约地址

我们提供了一些函数调用的字节码，您可以直接使用它们。

发布资产

调用issue函数，给账户0x1234567发布资产1,000,000,000，以下是该调用的字节码：

```
0x867904b4000000000000000000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000
```

以下是该合约的地址：

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

您可以通过以下命令调用合约：

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x867904b4000000000000000000000000000000000000000000000000000000000000000000000000123456700000000000000000000000000
↪--to 0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0x9f602fc0c4ac12d383cf765e2931978a0e5daaec666888e242485bd6314e17f9","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":20498,"ret":"0x0000000000000000000000000000000000000000000000000000000000000001
↪","log":[]}}
```

交易资产

调用transfer函数，从账户0x1234567向账户0x2345678转移1个资产，以下是该调用的字节码：

```
0xbeabacc8000000000000000000000000000000000000000000000000000000000000000012345670000000000000000000000000
```

以下是该合约的地址：

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

您可以通过以下命令调用合约：

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0xbeabacc8000000000000000000000000000000000000000000000000000000000000000012345670000000000000000000000000
↪--to 0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0x2d779df9db29541102e98ed9996263db42da6c13a2a6ac5c7ba9c606acecba28","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":25733,"ret":"0x0000000000000000000000000000000000000000000000000000000000000001
↪","log":[]}}
```

查询资产

调用getAccountBalance函数，查询账户0x1234567的资产，以下是该调用的字节码：

```
0x93423e9c0000000000000000000000000000000000000000000000000000000001234567
```

以下是该合约的地址：

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

您可以通过以下命令调用合约：

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x93423e9c0000000000000000000000000000000000000000000000000000000001234567 --to␣
↪0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xcca7b19dde84bf174243398de4107ee0b783b6e243176e162a2239bae1f475f7","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":353,"ret":"0x00000000000000000000000000000000000000000000000000000003b9ac9ff",
↪"log":[]}}
```

调用getAccountBalance函数，查询账户0x2345678的资产，以下是该调用的字节码：

```
0x93423e9c0000000000000000000000000000000000000000000000000000000002345678
```

以下是该合约的地址：

```
0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

您可以通过以下命令调用合约：

```
./hypercli contract invoke --from 000f1a7a08ccc48e5d30f80850cf1cf283aa3abd --
↪payload␣
↪0x93423e9c0000000000000000000000000000000000000000000000000000000002345678 --to␣
↪0x1e548137be17e1a11f0642c9e22dfda64e61fe6d
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0x04b82a4fcdadcf102d559b4eb6a29030f7ef29195f40a5f9b986021b48b48552","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed
↪":353,"ret":"0x0000000000000000000000000000000000000000000000000000000000000001",
↪"log":[]}}
```

3.4 交易处理

HyperCli提供了交易处理相关的功能，以下是该功能的参数：

```
$ ./hypercli tx --help
NAME:
   hypercli tx - transaction related commands

USAGE:
   hypercli tx command [command options] [arguments...]

COMMANDS:
     send     send normal transactions
     info     query the transaction info by hash
     receipt  query the transaction receipt by hash

OPTIONS:
   --help, -h  show help
```

可以看到tx命令有三个子命令，下面我们将介绍这三个命令。

3.4.1 发送交易

send命令可以用来发送普通的交易，以下是它的参数：

```
$ ./hypercli tx send --help
NAME:
   hypercli tx send - send normal transactions

USAGE:
   hypercli tx send [command options] [arguments...]

OPTIONS:
   --count value, -c value      send how many transactions (default: 1)
   --namespace value, -n value  specify the namespace to send transactions to␣
↪(default: "global")
   --from value, -f value       specify the account (default:
↪"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd")
   --to value, -t value         specify the contract address
   --password value, -p value   specify the password used to generate signature␣
↪(default: "123")
   --amount value, -a value     specify the amount to transfer (default: 0)
   --extra value, -e value      specify the extra information
   --snapshot value, -s value   specify the snapshot ID
   --simulate                   simulate execute or not
```

您可以用以下命令发送交易，它的意思是连续发送10笔交易：

```
./hypercli tx send -c 10
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xefbc9f9b5048337fbaf64047ad5eff03c40c1c76991b0364686ba3620e2c5ea3","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,
↪"ret":"0x0","log":[]}}

...
...

{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,
↪"ret":"0x0","log":[]}}
```

选取结果中的一条txHash，它之后将被用于查看交易的信息和回执。在这个例子中，我们选取这一个txHash：

```
0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

3.4.2 交易信息

info命令可以用来获取交易的信息，以下是它的参数：

```
./hypercli tx info --help
NAME:
   hypercli tx info - query the transaction info by hash

USAGE:
   hypercli tx info [command options] [arguments...]

OPTIONS:
   --hash value                 specify the tx hash used to query the detailed␣
↪information
   --namespace value, -n value  specify the namespace to query transaction␣
↪information (default: "global")
```

您可以用以下命令获取交易信息，它的意思是获取某一笔指定txHash的交易信息：

```
./hypercli tx info --hash␣
↪0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","hash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","blockNumber
↪":"0xf","blockHash":
↪"0x2592e0f3e1f156effe93325a60b1190533be2053aa102eb182bd64f95b28080a","txIndex":
↪"0x0","from":"0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd","to":
↪"0x6201cb0448964ac597faf6fdf1f472edf2a22b89","amount":"0x8","timestamp
↪":1512022032747286761,"nonce":5475154203949975728,"extra":"","executeTime":"0x5",
↪"payload":"0x0"}}
```

3.4.3 交易回执

receipt命令可以用来获取交易的回执，以下是它的参数：

```
$ ./hypercli tx receipt --help
NAME:
   hypercli tx receipt - query the transaction receipt by hash

USAGE:
   hypercli tx receipt [command options] [arguments...]

OPTIONS:
   --hash value                 specify the tx hash used to query the transaction␣
↪receipt
   --namespace value, -n value  specify the namespace to query transaction receipt␣
↪(default: "global")
```

您可以用以下命令获取交易回执，它的意思是获取某一笔指定txHash的交易回执：

```
./hypercli tx receipt --hash␣
↪0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d
```

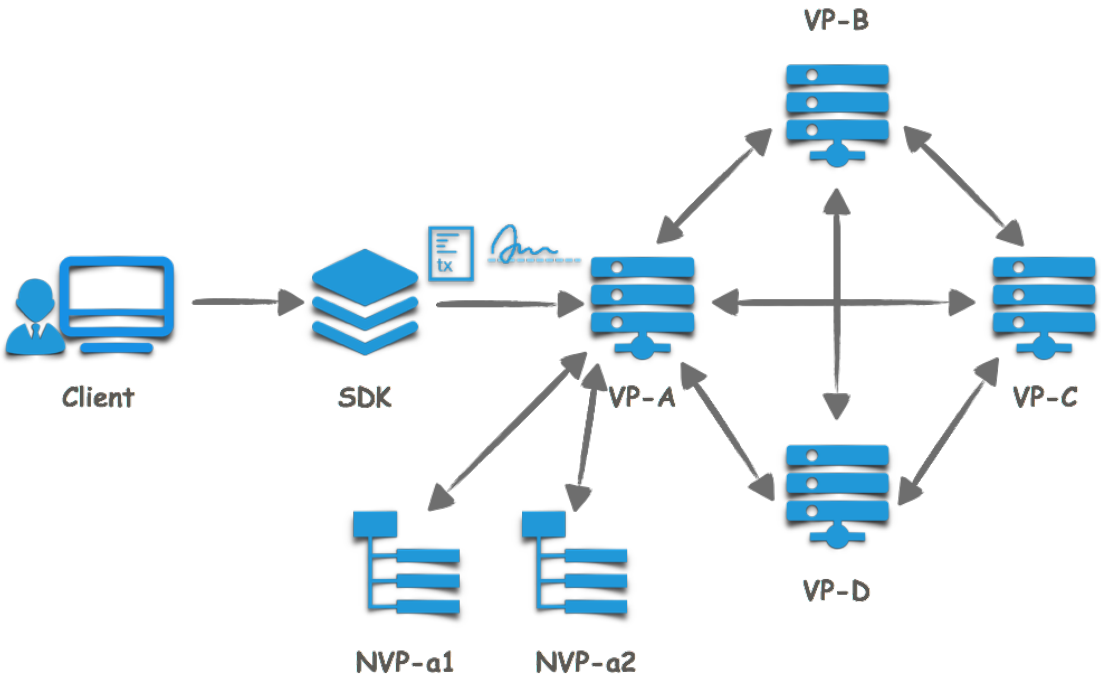
如果命令执行正确，您将看到以下输出结果：

```
{"jsonrpc":"2.0","namespace":"global","id":1,"code":0,"message":"SUCCESS","result":
↪{"version":"1.3","txHash":
↪"0xc47b64ddad2be542bfdc5164d447317f5152142ac7961c88332b25f04b31783d","vmType":
↪"EVM","contractAddress":"0x0000000000000000000000000000000000000000","gasUsed":0,
↪"ret":"0x0","log":[]}}
```

CHAPTER 4

交易执行流程

本文意在描述一条交易从最初产生到最终上链的交易机制。场景包括发起交易的客户端和与之直连的共识节点A，客户端通过SDK与共识节点A以交易的形式与区块链账本进行交互；共识节点A与其他共识节点B、C、D...全连接，同时共识节点A有两个用于备份的记账节点a1和a2。



假设

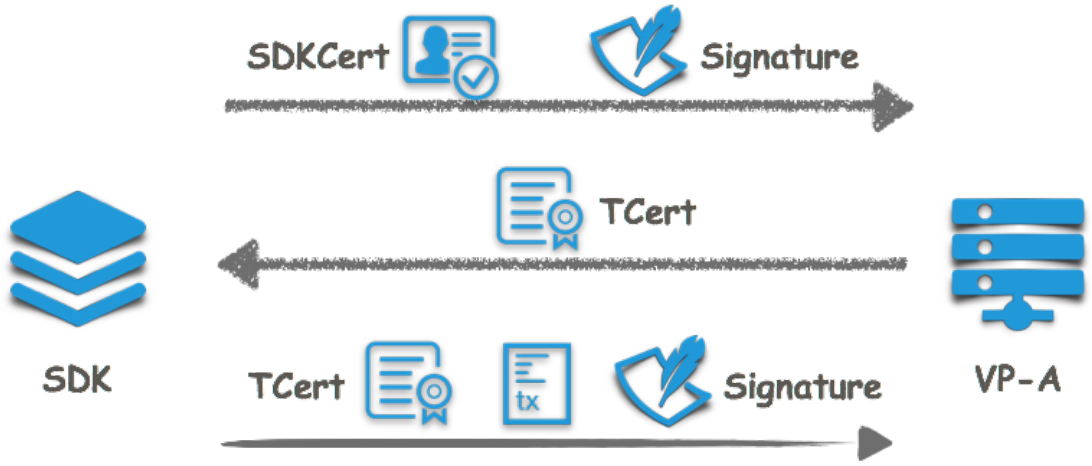
我们假设客户端已经拿到证书管理中心（CA）颁发的准入证书（SDKCert），智能合约已经部署在同一命名空间的区块链节点上。

客户端发起交易

现在要做的是发起一笔交易请求（调用已部署合约中的一个方法）。

客户端首先通过调用SDK的接口初始化一个 *HyperchainAPI* 对象，初始化过程中，SDK会以SDKCert和公钥向共识节点A请求获取发起交易所需的TCert。然后通过调用SDK的 *Transaction* 接口生成一条交易，SDK会用客户端指定的私钥对交易进行签名，再对交易进行JSONRPC协议封装后

用TCert对应的私钥进行消息签名。SDK与节点间支持 *HTTP/HTTPS* 短连接和 *WebSocket* 长连接。



节点受理交易，发送到全网共识节点

节点A收到交易后先进行TCert验证，节点只会对通过TCert验证的请求进行处理。节点的API模块还会做如下交易验证：

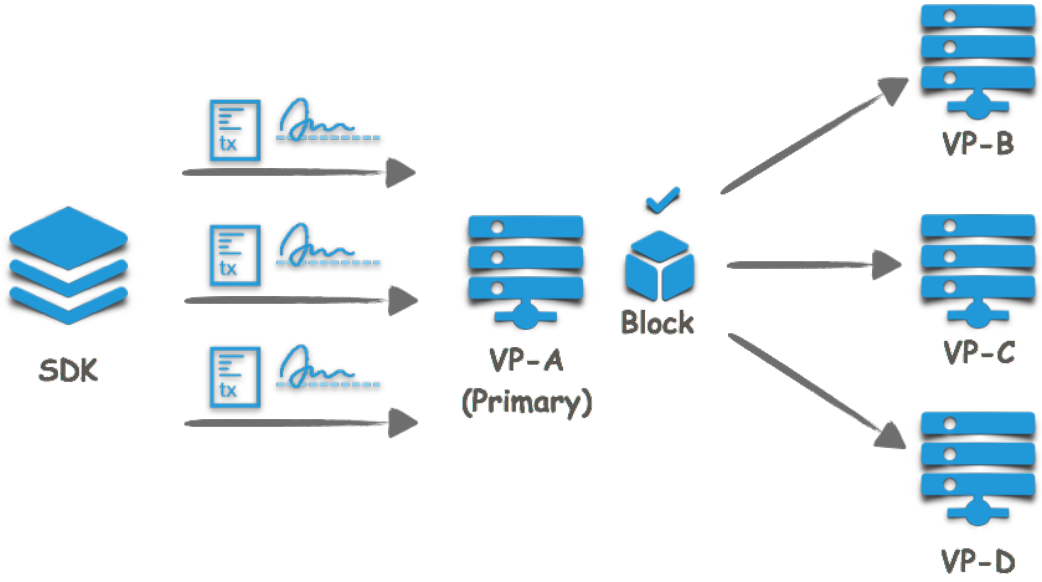
（1）根据流控配置确认是否接受交易请求； （2）验证交易字段的合法性，包括交易的格式，是否为空值，以及时间戳的合法性； （3）是否已经提交过相同交易（重放攻击）； （4）验证交易签名。

当交易通过以上验证后，会被提交到共识模块，共识模块将收到的交易向全网的共识节点进行广播。

交易的共识：排序，验证，写块

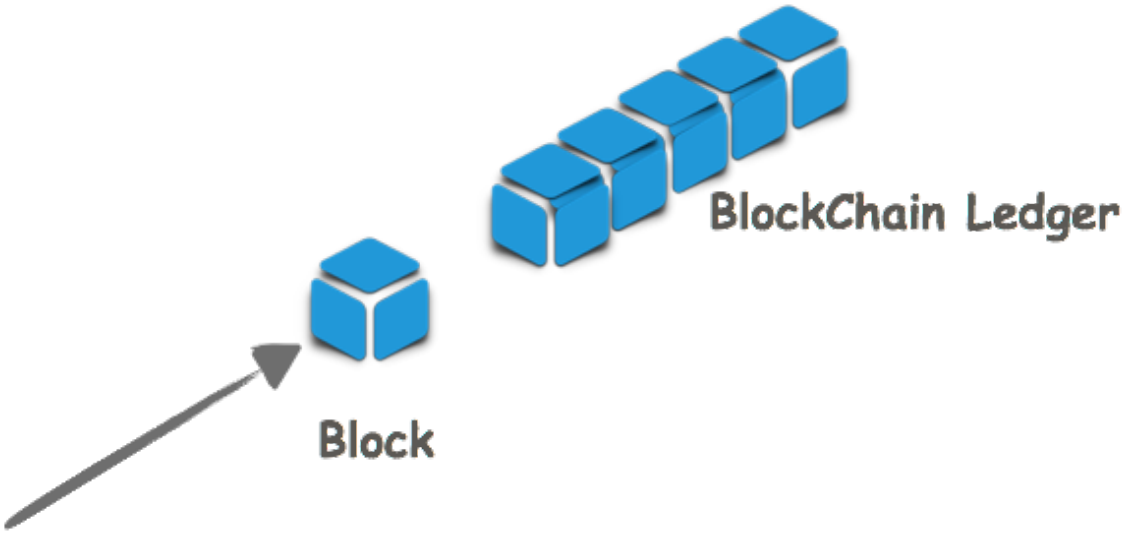
交易将会经历共识算法（RBFT）的三步走流程。

1. 预准备 · *Pre-Prepare* 共识主节点会将一定时间内（或者一定数量）的交易定序后打包成一个区块，然后发送到全网进行共识。



2. 准备 · *Prepare* 所有共识节点对区块进行预处理，并广播结果哈希。

3. 提交 · *Commit* 所有共识节点写入区块，并更新区块链账本。



在执行过程中发现的非法交易会被存储到数据库的非法交易记录中，并不会记录到区块链账本上。区块链账本上存储了所有合法交易。

所有共识节点在区块成功生成后也会将区块推送到所连接的所有记账节点进行记账。

交易回执

SDK在 *Transaction* 接口中实现了定时向节点获取交易结果，即查询交易回执。区块链网络设置的交易打包数量和打包时间会影响交易的延迟。

CHAPTER 5

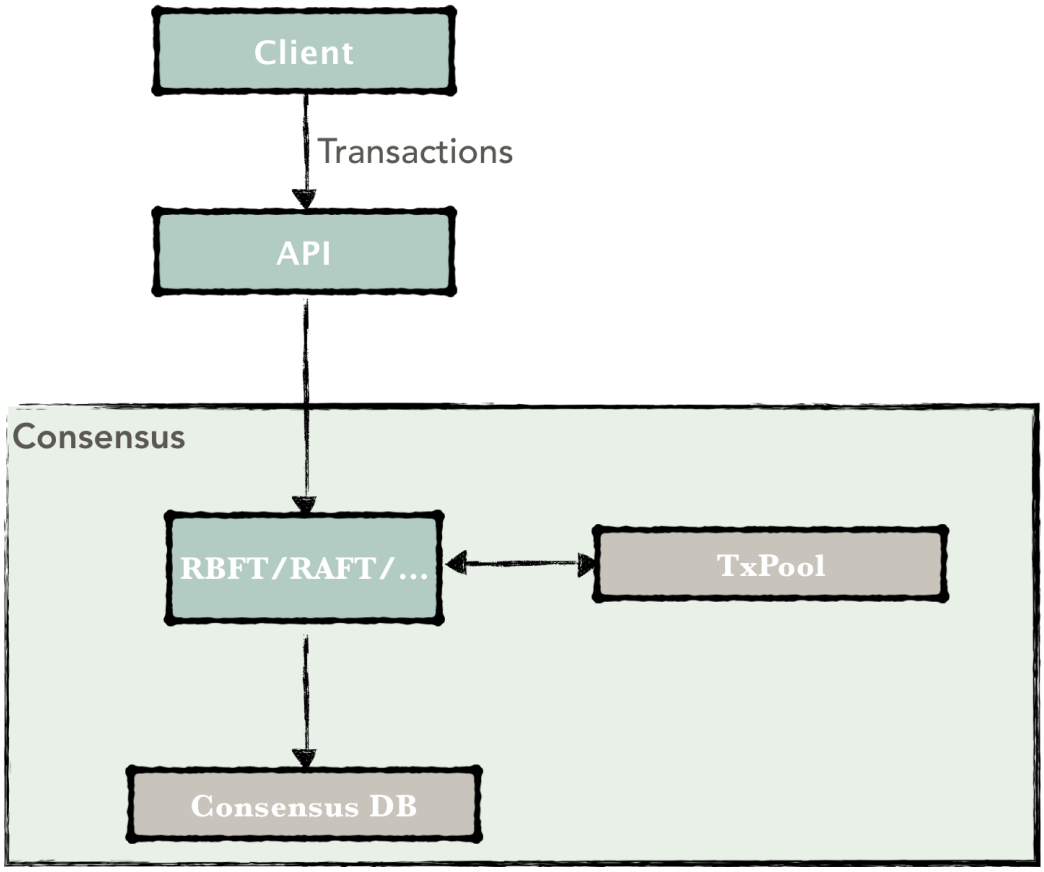
共识

5.1 1. 概述

共识机制是保证区块链中所有共识节点（即验证节点：validating peer， VP）按照相同顺序执行交易、写入账本的基础，而记账节点（即非验证节点：non-validating peer， NVP）只需要从其所连接的共识节点中同步账本信息，因此无需参与共识。

Hyperchain平台支持可插拔的共识机制，可以针对区块链的不同应用场景提供不同的共识算法，当前版本已经实现了PBFT算法的改良算法：高鲁棒拜占庭容错算法RBFT（Robust Byzantine Fault Tolerance），其算法构思来源于多篇论文(尤其是Aardvark)，后续将陆续支持RAFT等共识算法。

客户端发送交易到Hyperchain平台，API层解析出交易后转发给共识模块，共识模块接收并缓存交易到本地的交易池（TxPool）中，交易池承担着缓存交易与打包区块的作用，因此是作为共识模块的子模块实现的。另外，共识模块还需要维护一个共识的数据库，用于存储算法所需的变量以便宕机后的自主恢复，例如RBFT算法需要维护视图，PrePrepare，Prepare，Commit等共识信息。



5.2 2. RBFT相关变量

在一个由N个节点（N>=4）组成的共识网络中，RBFT最多能容忍f个节点的拜占庭错误，其中：

$$f = \lfloor \frac{N-1}{3} \rfloor$$

而能够保证达成共识的节点个数为：

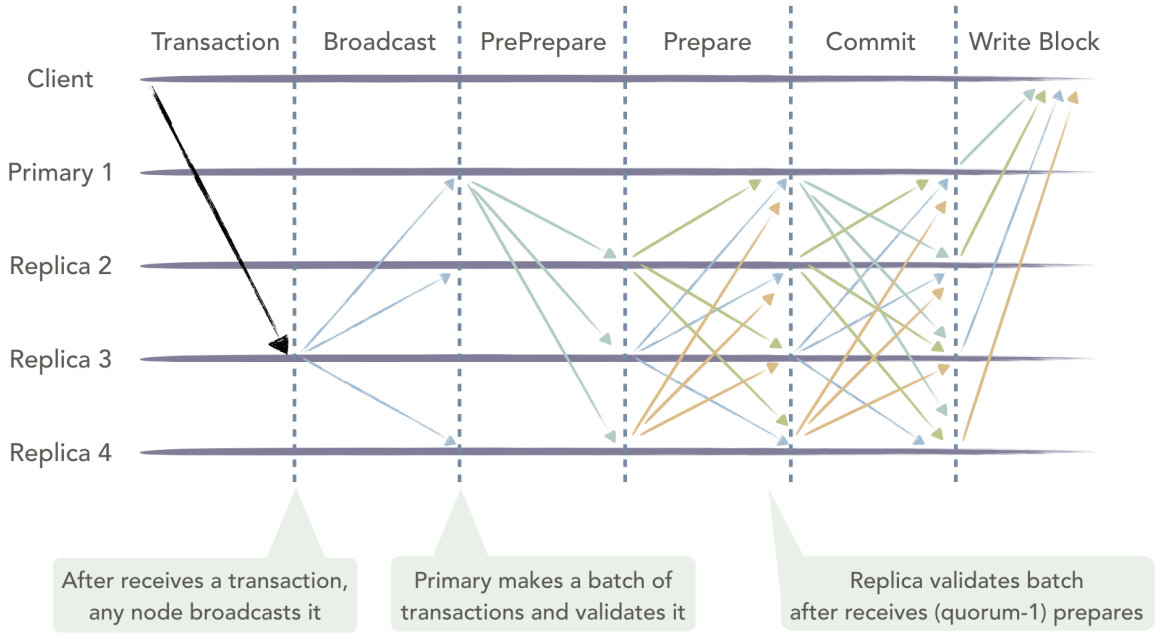
$$quorum = \lceil \frac{N+f+1}{2} \rceil$$

5.3 3. RBFT常规流程

RBFT的常规流程保证了区块链各共识节点以相同的顺序处理来自客户端的交易。RBFT同PBFT的容错能力相同，需要至少3f+1个节点才能容忍f个拜占庭错误。下图为最少集群节点数下的共识流程，其N=4，f=1。图中的Primary1为共识节点动态选举出来的主节点，负责对客户端发来的交易进行排序打包，Replica2，3，4为从节点。所有节点执行交易的逻辑相同并能够在主节点失效时参与新主节点的选举。

5.3.1 常规流程

RBFT共识保留了PBFT原有的三阶段处理流程（PrePrepare、Prepare、Commit）的同时增加了重要的交易验证（validate）环节，在保证对交易执行顺序达成共识的同时也保证了对区块验证结果的共识。



RBFT常规流程在原生的PBFT算法中穿插了交易验证环节，主节点将交易打包成块后先行验证，并将验证结果包含到PrePrepare消息中进行全网广播，这样PrePrepare消息中既包含了排好序的交易信息也包含了区块验证结果。从节点在收到主节点的PrePrepare消息后先检查消息的合法性，检查通过后广播Prepare消息表明本节点同意主节点的排序结果；在收到（quorum-1）个Prepare消息后从节点才会开始验证区块，并将验证结果与主节点的验证结果进行比对，比对结果一致则广播Commit表明本节点同意主节点的验证结果，否则直接发起ViewChange表明本节点认为主节点有异常行为。RBFT常规流程具体分为如下几个步骤：

1. **交易转发阶段**：客户端将交易发送到区块链中的任意节点（包括共识节点与记账节点），其中记账节点在收到交易后会主动转发给与其相连的共识节点；而共识节点在收到客户端的交易后将其广播给其他共识节点，这样所有共识节点的交易池中都会维护一份完整的交易列表；
2. **PrePrepare阶段**：主节点按照如下策略进行打包：用户可以根据需求自定义打包的超时时间（batch timeout）与打包的最大区块大小（batch size），主节点在超时时间内收集到了足够多（超过最大区块大小个数）的交易或者超时时间到达后仍未收集到足够多的交易都会触发主节点的打包事件。主节点将交易按照接收的时间顺序打包成块，并进行验证，计算执行结果，最后将定序好的交易信息连同验证结果等写入PrePrepare消息中广播给所有共识节点，开始三阶段处理流程；
3. **Prepare阶段**：从节点在收到主节点的PrePrepare消息后，首先进行消息合法性检查，检查当前的视图与区块号等信息，检查通过后向共识节点广播Prepare消息；
4. **Commit阶段**：从节点在收到（quorum-1）个Prepare消息以及相应的PrePrepare消息后进行验证，并将验证结果与主节点写入PrePrepare消息中的验证结果进行比对，比对结果一致则广播Commit表明本节点同意主节点的验证结果，否则直接发起ViewChange表明本节点认为主节点存在异常行为，需要切换主节点；
5. **写入账本**：所有共识节点在收到quorum个Commit消息后将执行结果写入本地账本。

Hyperchain通过在共识模块中加入验证机制，可以保证从节点对主节点的每一次排序打包的结果进行校验，尽早地发现主节点的拜占庭行为，提升了系统的稳定性。

5.3.2 检查点

为了防止运行过程中产生过多的消息缓存，共识节点需要定时清理一些无用的消息缓存。RBFT通过引入PBFT算法中的检查点（checkpoint）机制进行垃圾回收并将检查点的大小K固定设置为10。节点在写入到K的整数倍个区块后达到一个检查点，并广播该检查点的信息，待收集到其他（quorum-1）个共识

节点相同的检查信息后就达到了一个稳定检查点（stable checkpoint），随后即可清理该检查点之前的一些消息缓存，保证了运行过程中消息缓存不会无限制地增长。

5.3.3 交易池

交易池是共识节点用于缓存交易的场所，交易池的存在一方面限制了客户端发送交易的频率，另一方面也减少了主节点的带宽压力。首先，通过限制交易池的缓存大小，Hyperchain平台可以在交易池达到限制大小后拒绝接收来自客户端的交易，这样，在合理评估机器性能的情况下，通过合理设置交易缓存大小，可以最大限度地利用机器性能而又不至于出现异常。其次，共识节点在接收到来自客户端的交易后先将其存到自己的交易池中，随后向全网其他共识节点广播该条交易，保证了所有共识节点都维护了一份完整的交易列表；主节点在打包后只需要将交易哈希列表放到PrePrepare消息中进行广播即可，而不用将完整的交易列表打包进行广播，大大减轻了主节点的出口带宽压力。如果从节点在验证之前发现缺少了某些交易，也只需要向主节点索取缺少的那些交易而不用索取整个区块里面所有的交易。

5.4 4. RBFT视图变更

RBFT视图变更能够解决主节点成为拜占庭节点的问题。在RBFT算法中，参与共识的节点可根据角色分为主节点和从节点。主节点最重要的功能是将收到的交易按照一定策略打包成块，为交易定序，并让所有节点按照此顺序执行。然而，如果主节点发生宕机、系统错误或者被攻占（即成为拜占庭节点），从节点需要及时发现主节点异常并选举产生新的主节点。这将是所有BFT类算法为满足稳定性必须要解决的问题。

5.4.1 视图

在RBFT与PBFT中，都引入了视图（View）概念，即每次更换一个主节点的同时都会切换视图。目前RBFT采用轮换的方式切换主节点，并且view从0开始只增不减。当前的view和总节点数量N决定了主节点id：

$$PrimaryId = (view + 1) \bmod N$$

5.4.2 可检测到的拜占庭行为

目前RBFT能够检测到的主节点的拜占庭行为主要有2种场景：

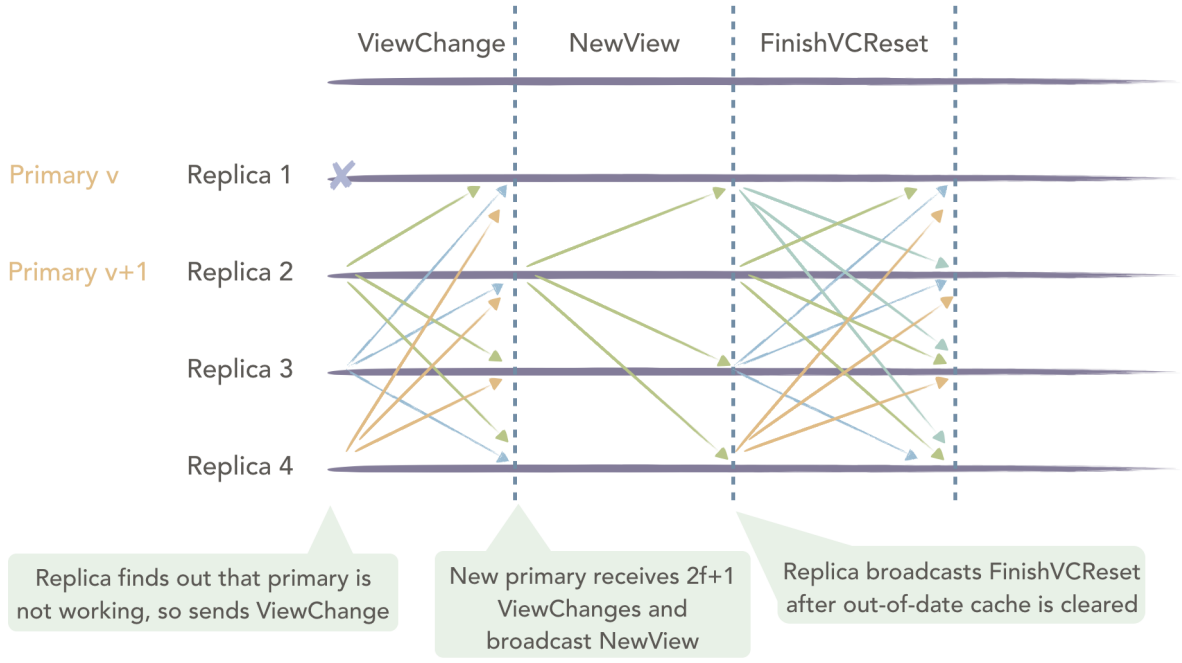
1. 主节点停止工作，不再发送任何消息；
2. 主节点发送错误的消息。

对于场景一，RBFT由nullRequest机制保证，行为正确的主节点会在没有交易发生时，向所有从节点定时发送nullRequest来维持正常连接。如果从节点在规定时间内没有收到nullRequest，则会触发ViewChange流程选举新的主节点。

对于场景二，从节点会对主节点发出的消息进行验证，如上一节中提到的包含在PrePrepare消息中的验证结果，如果从节点验证不通过的话，会直接发起ViewChange流程选举新的主节点。

此外，RBFT还提供了可配置的ViewChangePeriod选项。用户可以根据需要设置此选项，每写入一定数量区块后进行主动的ViewChange轮换主节点，一来能够缓解主节点作为打包节点的额外压力，二来也使所有参与共识的节点都能承担一定的打包工作，保证了公平性。

5.4.3 视图变更流程



上图中，Primary 1为拜占庭节点，需要进行ViewChange。在RBFT中的ViewChange流程如下：

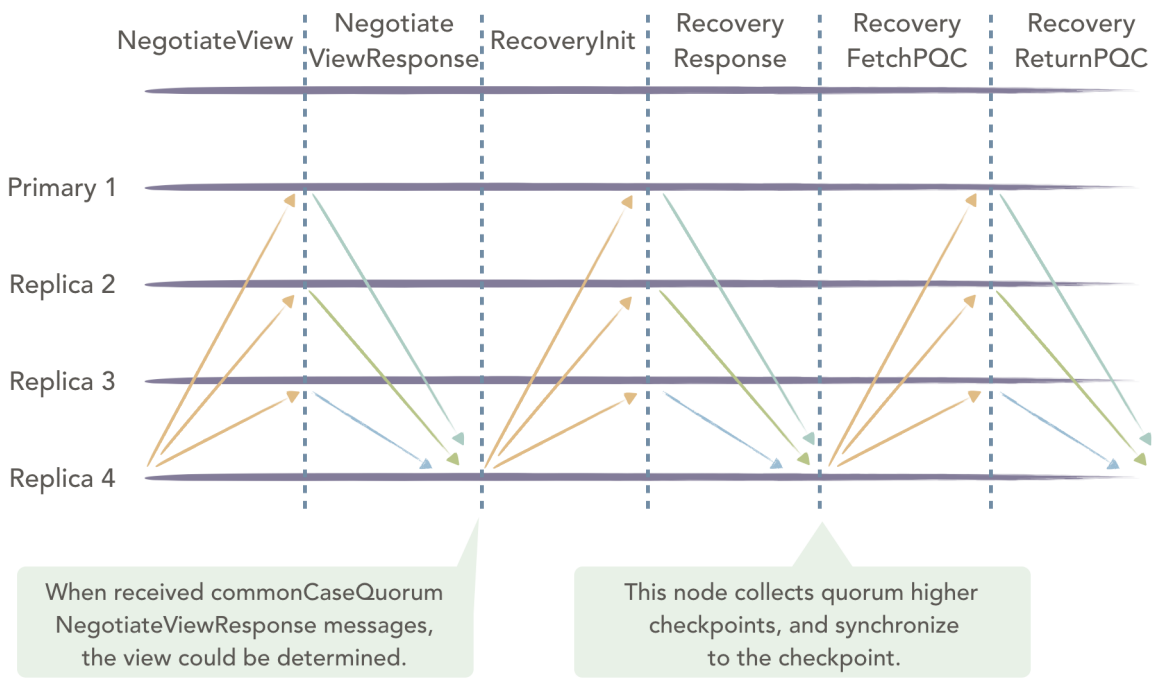
1. 从节点在检测到主节点有异常情况（没有按时收到nullRequest消息）或者接收到来自其他f+1个节点的ViewChange消息之后会向全网广播ViewChange消息，自身view从v更改为v+1；
2. 新视图中主节点收到N-f 个ViewChange消息后，根据收到的ViewChange消息计算出新视图中主节点开始执行的checkpoint和接下来要处理的交易包，封装进NewView消息并广播，发起VcReset；
3. 从节点接收到NewView消息之后进行消息的验证和对比，如果通过验证，进行VcReset，如果不通过，发送ViewChange消息，进行又一轮ViewChange；
4. 所有节点完成VcReset之后向全网广播FinishVcReset；
5. 每个节点在收到N-f个FinishVcReset消息之后，开始处理确定的checkpoint后的交易，完成整个ViewChange流程。

由于共识模块与执行模块之间是异步通信的，而ViewChange之后执行模块可能存在一些无用的validate缓存，因此共识模块需要在ViewChange完成之前通知执行模块清除无用的缓存，RBFT通过VcReset事件主动通知执行模块清除缓存，并在清理完成之后才能完成ViewChange。

5.5 5. RBFT自主恢复

区块链网络在运行过程中由于网络抖动、突然断电、磁盘故障等原因，可能会导致部分节点的执行速度落后于大多数节点。在这种场景下，节点需要能够做到自动恢复才能继续参与后续的共识流程。为了解决这类数据恢复的问题，RBFT算法提供了一种动态数据自动恢复的机制(recovery)，recovery通过主动索取现有共识网络中所有节点的视图、最新区块等信息来更新自身的存储状态，最终同步至整个系统的最新状态。在节点启动、节点重启或者节点落后的时候，节点将会自动进入recovery，同步至整个系统的最新状态。

5.5.1 自主恢复流程



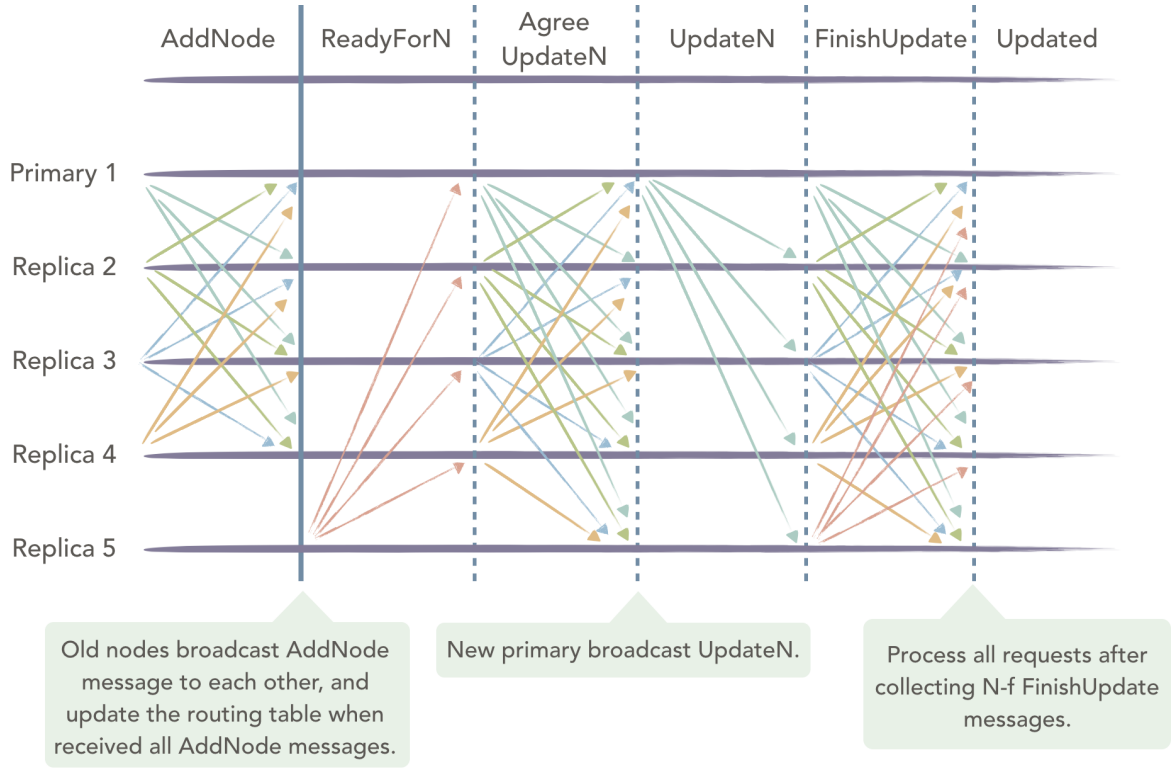
上图中，Replica 4为落后节点，需要进行recovery。此节点在RBFT中的自动恢复流程如下：

1. Replica 4 首先广播NegotiateView消息，获取当前其余节点的视图信息；
2. 其余三个节点向Replica 4发送NegotiateViewResponse，返回当前视图信息。
3. Replica 4 收到quorum个NegotiateViewResponse消息后，更新本节点的视图；
4. Replica 4 广播RecoveryInit消息到其余节点，通知其他节点本节点需要进行自动恢复，请求其余节点的检查点信息和最新区块信息；
5. 正常运行节点在收到RecoveryInit消息之后，发送RecoveryResponse，将自身的检查点信息以及最新区块信息返回给Replica 4节点；
6. Replica 4节点在收到quorum个RecoveryResponse消息后，开始尝试从这些response中寻找一个全网共识的最高的检查点，随后将自身的状态更新到该检查点；
7. Replica 4节点向正常运行节点索要检查点之后的PQC数据，最终同步至全网最新的状态。

5.6 6. RBFT节点增删

在联盟链场景下，由于联盟的扩展或者某些成员的退出，需要联盟链支持成员的动态进出服务，而传统的PBFT算法不支持节点的动态增删。RBFT为了能够更加方便地控制联盟成员的准入和准出，添加了保持集群非停机的情况下动态增删节点的功能。

5.6.1 新增节点流程



上图中，Replica 5为待新增的节点。RBFT节点的动态新增节点流程如下：

1. 新增节点Replica 5通过读取配置文件信息，主动向现有节点发起连接，确认所有节点连接成功后更新自身的路由表，并发起recovery；
2. 现有节点接收到Replica 5的连接请求后确认同意该节点加入，然后向全网广播AddNode消息，表明自己同意该新节点加入整个共识网络；
3. 当现有节点收到N条（N为现有区块链共识网络中节点总数）AddNode消息后，更新自身的路由表，随后开始回应新增节点的共识消息请求（在此之前，新增节点的所有共识消息是不予处理的）；
4. Replica 5完成recovery之后，向全网现有节点广播ReadyForN请求；
5. 现有节点在收到ReadyForN请求后，重新计算新增节点加入之后的N,view等信息，随后将其与PQC消息封装到AgreeUpdateN消息中，进行全网广播；
6. Replica 5加入后的共识网络会产生一个新的主节点，该主节点在收到N-f个AgreeUpdateN消息后，以新的主节点的身份发送UpdateN消息；
7. 全网所有节点在收到UpdateN消息之后确认消息的正确性，进行VCReset；
8. 每个节点完成VCReset后，全网广播FinishUpdate消息；
9. 节点在收到N-f个FinishUpdate消息后，处理后续请求，完成新增节点流程。

CHAPTER 6

账本

6.1 1. 概述

账本是hyperchain平台中的重要模块，负责区块链账本数据的维护与组织。账本数据可以分成两部分：

- 区块链数据
- 账户数据

其中，区块链数据包括：区块、交易、回执等数据。这部分也就是我们传统意义上所说的区块链。而后者指代的是区块链上所有账户状态的集合，该状态集统称为**世界状态**。由于需要支持智能合约，因此hyperchain与以太坊一样，摒弃了比特币的UTXO模型而采用账户模型来组织数据，因而这部分数据称为账户数据。

区块链数据主要通过区块的形式进行串联。所有区块被从后向前有序地链接在一个链条里，每一个区块都指向其父区块。区块中包含了一批交易，由共识模块负责统一打包并定序。区块链节点在接收到一个区块之后，在原有的**账户状态**基础上，依次执行交易，在此期间读／写相关账户的状态数据。一笔交易执行结束，也就意味着区块链**状态**进行了一次变迁。

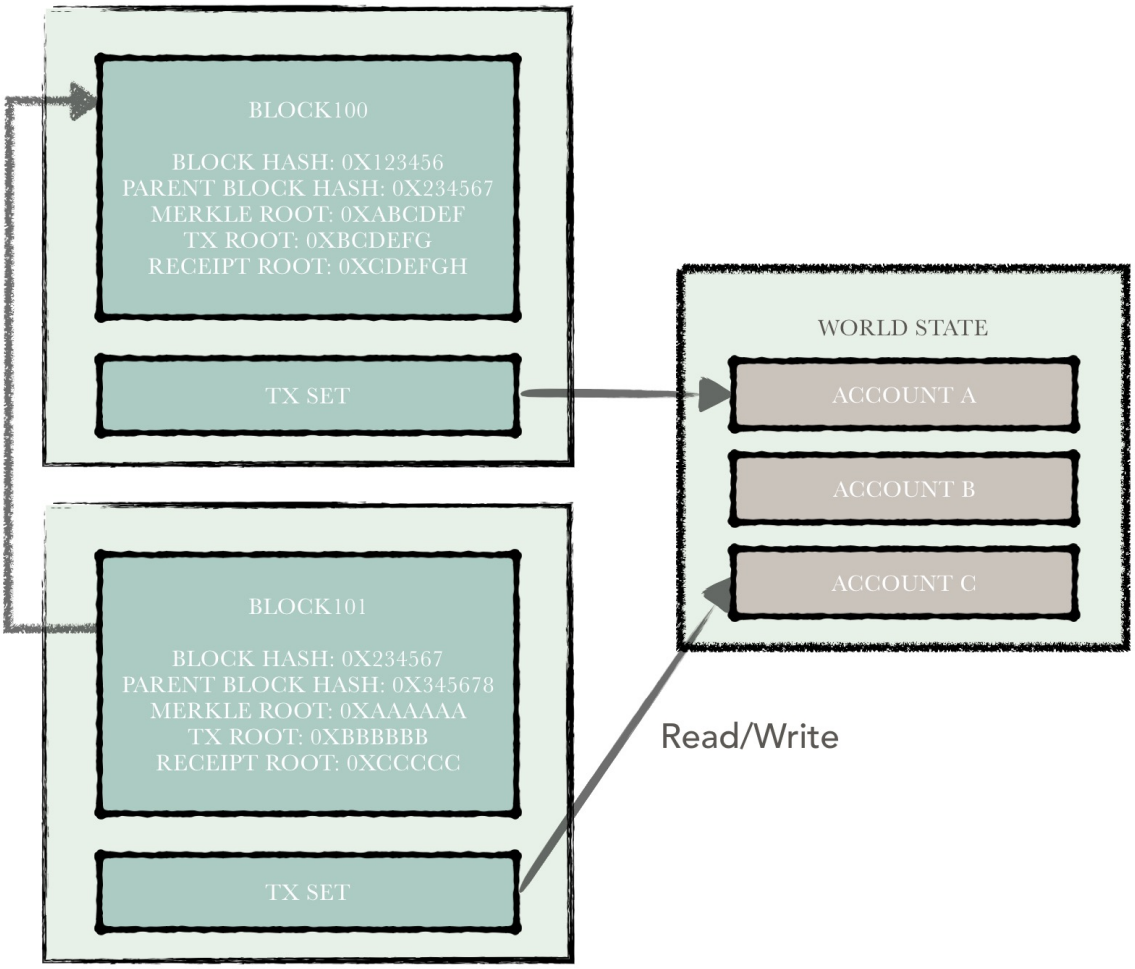
每一笔交易，在hyperchain中都会有一个对应的**交易回执**或者**非法交易记录**来表示最终的执行结果。倘若这笔交易是一笔合法的交易，则执行结束后，会将该交易执行的结果记录在交易回执中。反之，会将错误原因记录在一条非法交易记录中。

账本中各部分大体的逻辑关系可以如下图所示。

6.2 2. 区块链数据

在本章中，将介绍以下几种数据结构之间的关系：

- 区块
- 交易
- 回执
- 区块链
- 非法交易记录


BLOCK100
BLOCK HASH: 0X123456
PARENT BLOCK HASH: 0X234567
MERKLE ROOT: 0XABCDEF
TX ROOT: 0XBCDEFG
RECEIPT ROOT: 0XCDEFGH
TX SET
BLOCK101
BLOCK HASH: 0X234567
PARENT BLOCK HASH: 0X345678
MERKLE ROOT: 0XAAAAAA
TX ROOT: 0XBBBBBB
RECEIPT ROOT: 0XCCCCC
TX SET
WORLD STATE
ACCOUNT A
ACCOUNT B
ACCOUNT C
Read/Write

其中前两类数据结构在区块链网路中组成了“区块链数据“，是区块链网络中进行流转的“共识”数据；后三类数据结构由各节点维护在本地。以上五种数据结构组成了一个节点中所有的区块链数据。

6.2.1 区块

区块结构可以分成两部分：

- 区块头信息；
- 区块体信息；

区块头中主要包含一些区块链的元数据，包括：（1）区块高度（2）区块哈希（3）父区块哈希（4）账户状态哈希（5）交易集哈希（6）回执集哈希（7）时间戳（8）日志过滤数据。

区块体内包含所有的交易数据。

区块的主要作用是**封装交易数据，记录区块链状态数据**。

6.2.2 交易

交易是由外部用户发起的，在交易体中记录了用户指定的调用信息。

交易根据是否执行智能合约可以分为两类：

- 普通交易；
- 合约交易；
- 合约部署
- 合约调用

前者表示交易执行过程中不触发智能合约的运行，仅进行hyperchain提供的token的转账；后者表示交易执行过程中会触发智能合约的运行。

后者又可以分为：（1）合约部署交易（2）合约调用交易。

交易体包含如下字段：

- 版本号：指明该交易数据结构定义的版本信息，便于向后兼容；
- 交易发起者：长度为20字节的交易发起者的标识信息；
- 交易接收者：长度为20字节的交易接收者的标识信息，若本交易是合约调用交易，该字段为被调合约的地址，若该字段为空，则表明本交易为部署合约交易；
- 调用信息：
- 若本交易为普通交易，在调用信息中指定需要转账的token数量；
- 若本交易为合约调用交易，在调用信息中指定需要调用的函数以及调用参数；
- 若本交易为合约部署交易，则需要在调用信息中指定合约的二进制代码；
- 随机值：uint64的随机值（避免产生哈希相同的交易，防止重放攻击）；
- 交易签名：用户利用自己的私钥对（1）交易发起者（2）交易接收者（3）调用信息（4）时间戳（5）随机值五个字段的内容进行签名，产生的签名内容填在该字段，防止交易的内容被篡改；
- 交易哈希：将上述（1）-（5）字段加上交易签名一起进行哈希计算，获得一个哈希标识用于表示本交易。

6.2.3 回执

每一笔合法的交易，其执行结果都会被封装成一个交易回执存储在区块链上。交易回执包括：

- 版本号：指明该回执数据结构定义的版本信息，便于向后兼容；
- 交易哈希：与该回执相关的交易哈希；
- 合约地址：若该交易为部署合约交易，则新部署的合约地址被放置在该字段，否则该字段为空；
- 执行结果：若该交易为调用合约交易，则执行结果被放置在该字段，否则该字段为空；
- 虚拟机日志：智能合约运行过程中，可能会产生一系列日志，这些日志数据被放置在该字段中；
- 智能合约类型：用于表示智能合约的类型，EVM，JVM或是其他类型；

6.2.4 非法交易记录

每一笔非法的交易，其错误信息会被封装成一个非法交易记录，存储在节点本地。

除了与之相关的交易数据，非法记录中还会记载具体的错误原因，例如：（1）余额不足（2）合约调用参数错误（3）调用权限不够等。

6.2.5 区块链

一个本地的节点会维护一些区块链元数据以便用户进行查询，因此在hyperchain中，有个名为chain的数据结构，记录了这些数据，包括：

- 最新父区块哈希
- 最新区块哈希
- 最新区块高度
- 创世区块高度（默认为0，数据归档／数据恢复等操作会影响该高度的值）
- 交易总数
- 额外数据

6.2.6 共识比较

区块链节点在执行完一个区块中所有的交易后，需要将本次区块处理得到的“结果”在节点间进行比较，只有大多数节点(超过quorum个)拥有与之相同的结果时，才会将本次执行结果提交到数据库中。

而表示本区块执行结果的标识是由以下几个内容组成：

- 世界状态哈希：交易执行过程中会更改账户状态数据，当一个区块中所有的交易执行结束后，会利用bucket tree对账户集合状态进行哈希重计算，计算结果便是世界状态哈希；
- 交易集哈希：利用区块中每一笔交易的*重要字段*作为sha256算法的输入，计算得到一个用于表示整个交易集的哈希标识。重要字段为：(1）交易发起者（2）交易接收者（3）调用信息（4）时间戳（5）随机值；
- 回执集哈希：利用区块中每一个回执的*重要字段*作为sha256算法的输入，计算得到一个用于表示整个回执集的哈希标识。重要字段为：（1）虚拟机执行计数器（2）执行结果（3）虚拟机执行日志；

6.3 3. 账户数据

之前提及的区块链数据，其实可以总结为是**合约调用信息的流水集合**。而智能合约在运行过程中，需要读／写合约状态数据。接下来就介绍这部分数据的组织结构。

由于hyperchain需要兼容EVM（Ethereum Virtual Machine），而EVM与以太坊的账户体系有着较强的耦合性，因此，hyperchain的state是在以太坊的基础上，做了一系列的改造及优化得到。

6.3.1 账户类别

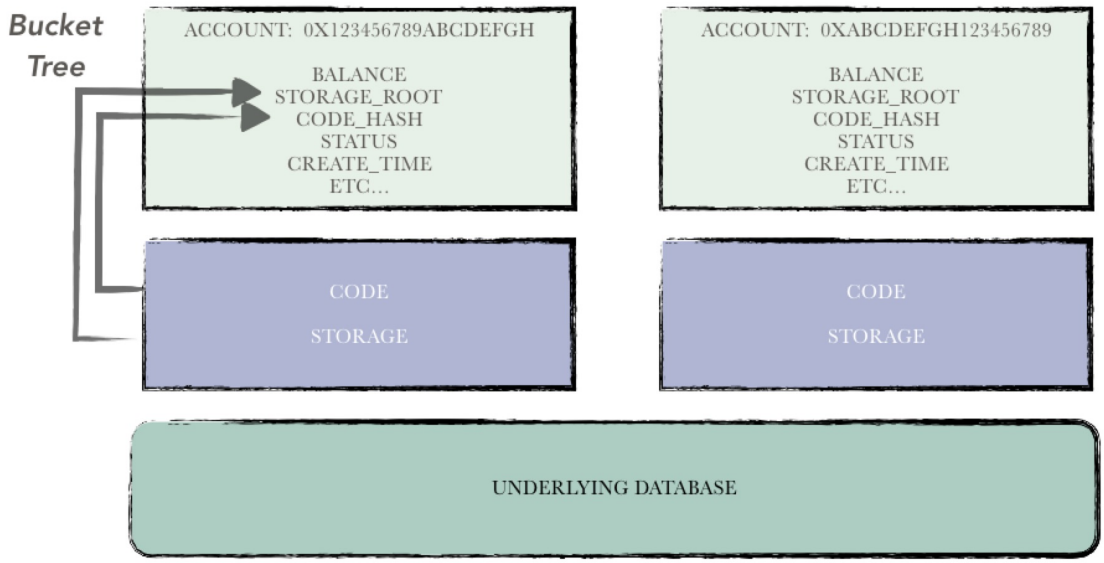
与以太坊一样，hyperchain中的账户也可以分为两类：

- 外部账户：外部账户的私钥由用户自己控制，可以主动发起交易，且这类账户不包含智能合约代码
- 合约账户：合约账户包含一段可执行的智能合约代码，且有自己的存储空间用来存储自身的状态变量。该智能合约的运行可以由外部账户发起交易进行触发，也可以由其他合约“主动调用”进行触发。

虽然两类账户在逻辑上有所区别，但是共享一套定义：

一个账户的**元数据**包括以下字段：

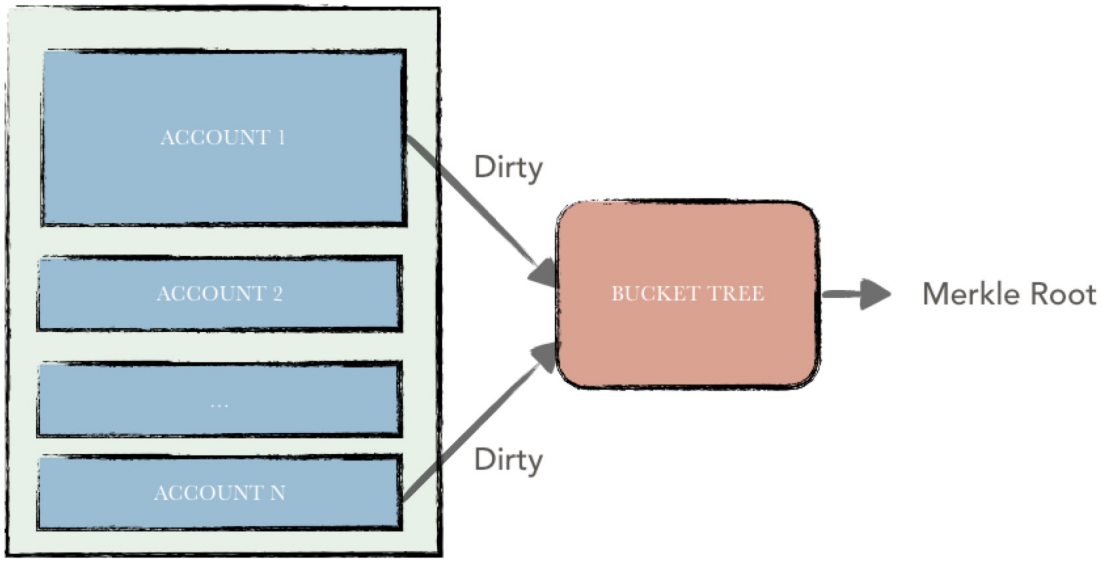
- 账户地址（20字节，由哈希函数根据一定的输入产生，不考虑哈希冲突的情况下，不会有两个相同地址的账户）；
- 余额：该余额表示该账户所拥有的hyperchain平台的token个数，这类token可以通过智能合约进行操控，也可以通过发起普通交易转账进行交易；
- 状态变量（存储空间）哈希标识：一个合约账户，需要存储其所有的状态变量，一个用于表示这些状态变量的哈希值被存储在该字段；
- 合约代码哈希：智能合约代码哈希标识；
- 状态：合约状态，普通、冻结等；
- 部署时间：若该账户为合约账户，则会记录该账户第一次被部署的时间点；
- 创建者地址：若该账户为合约账户，则会记录该账户的创建者信息；
- 已部署的合约地址列表：若该账户为外部账户，则会记录该账户部署的所有合约账户的地址；



除了以上这些"简短"的数据被放置在账户元数据里，还有（1）合约源码（2）状态变量这些需要大量存储空间的数据被直接存储，在元数据中只存储这类数据的哈希值。

合约的状态变量，其实是一系列的kv键值对。在hyperchain中，每个合约账户，**都会有一棵bucket tree**，专门用来计算该合约的状态变量哈希。每次执行一笔交易，修改一系列状态变量，从底层来看，其实就是更新了一批kv对，而这些修改集刚好可以作为bucket tree的输入，以便快速地计算"新"的状态变量哈希。

6.3.2 账户集



hyperchain将每一个账户的**元数据**进行序列化，将序列化得到的二进制作为一个账户的内容。

所有的账户数据，最终可以转换成一系列的kv对，key为该账户的地址，value为元数据序列化的内容。

对于账户集，会有一棵全局级别的bucket tree进行账户数据的哈希计算，示意图如上所示。每个账户数据仅作为bucket tree中的一条数据项，不断进行哈希计算，最终生成一个根节点，该节点的哈希值（merkle root）便是整个账户集的哈希标识。

该哈希值作为整个账户集的状态表示，不仅是共识阶段比较的依据之一，之后更是会被记录在区块头中。

6.3.3 原子性

hyperchain利用底层数据库leveldb提供的batch来保障账本的原子性。hyperchain采用rbft作为共识算法，因此整个处理流程会分成3阶段来完成。在执行阶段，所有对账本的改动会预先保存在一个leveldb的batch中；当本次执行的结果得到了足够多节点的认可，会从缓存中取出该batch，将所有的修改落盘。

CHAPTER 7

Bucket tree

7.1 概述

在hyperchain中，账本数据可以分成两部分：

- 区块链数据
- 账户数据

其中，区块链数据包括：区块、交易、回执等数据。这部分也就是我们传统意义上所说的区块链。所有区块被从后向前有序地链接在这个链条里，每一个区块都指向其父区块。

区块中包含了一批交易，由共识模块负责将接收到的交易统一定序并打包成一个区块进行分发。区块链节点在接收到一个区块之后，在原有的**状态**基础上，依次执行交易，在此期间读／写相关账户的状态数据；执行结束，将期间所有的账本改动统一写入。每一笔交易的执行，都意味着区块链进行了一次状态变迁。

在这里，区块链**状态**指代的是区块链上所有账户状态的集合，该状态集统称为**世界状态**。由于支持智能合约，因此与以太坊一样，hyperchain摒弃了比特币的UTXO模型而采用账户模型来组织数据，因而这部分数据称为账户数据。

所以，hyperchain的账本体系可以大致分为上述两部分，架构示意图如下所示。

在本文中我们不展开对账本结构的讨论。而来讨论一种用来**快速计算账户集状态**的树结构 - bucket tree。在hyperchain中，每执行完一个区块，各个节点需要在共识的第三阶段比较执行的结果是否一致。也就是说，每个节点在执行完一个区块后，账户数据需要保持一致。因此，hyperchain中使用了一种bucket tree的结构对账户数据进行哈希计算，节点间只需要通过比较该哈希值就能判断账户数据的一致性。

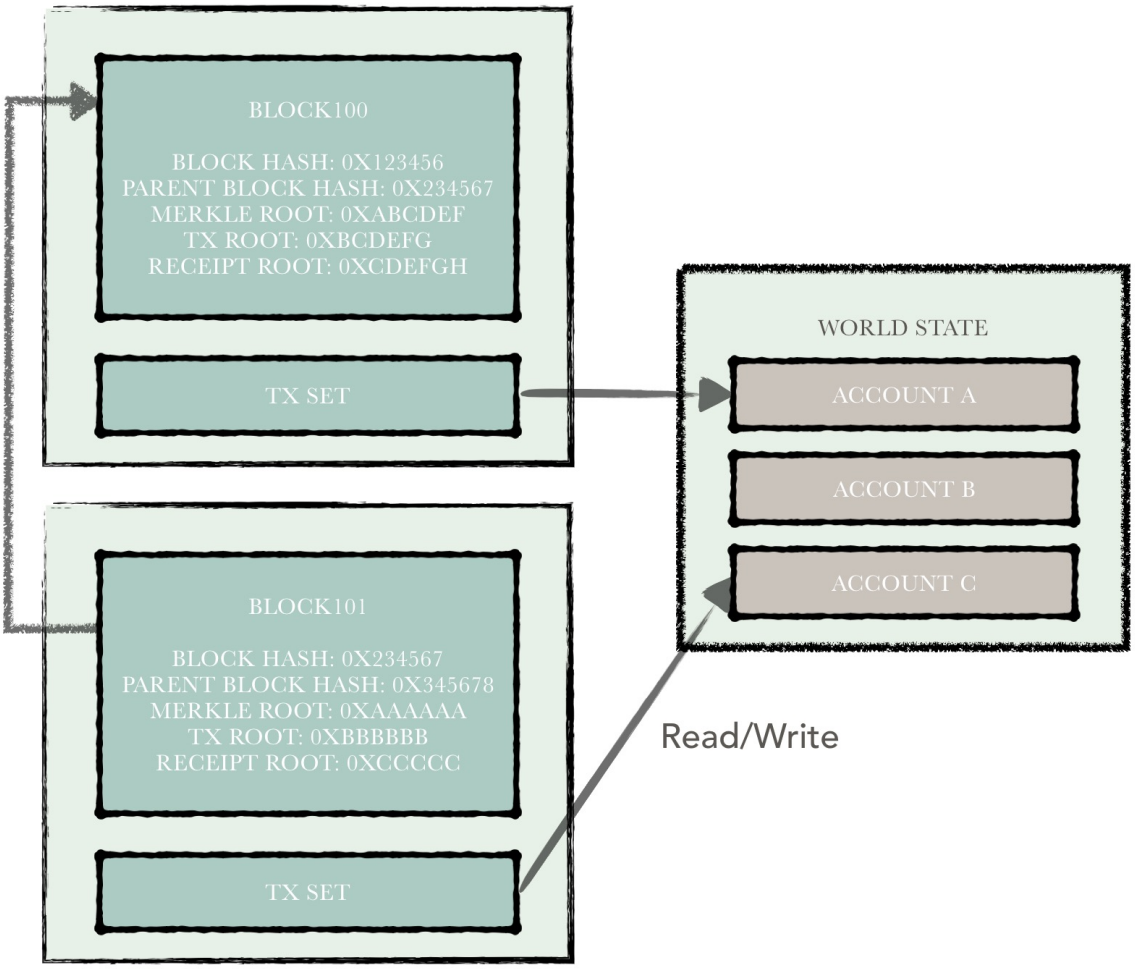
注解：值得注意的是，bucket tree并不组织和维护账户数据，而仅仅只是进行状态的哈希计算。

---

bucket tree有以下特点：

- 提供了一种快速计算账户数据哈希标示的机制；
- 提供了账本回滚的机制；

hyperchain中的bucket tree思想最早从fabric项目中借鉴得到，进行了一系列的重构和优化，使得最终的性能表现符合生产需求。下文中，将详细介绍这种树的结构、主要操作以及最终的性能表现。


BLOCK100
BLOCK HASH: 0X123456
PARENT BLOCK HASH: 0X234567
MERKLE ROOT: 0XABCDEF
TX ROOT: 0XBCDEFG
RECEIPT ROOT: 0XCDEFGH
TX SET
BLOCK101
BLOCK HASH: 0X234567
PARENT BLOCK HASH: 0X345678
MERKLE ROOT: 0XAAAAAA
TX ROOT: 0XBBBBBB
RECEIPT ROOT: 0XCCCCC
TX SET
WORLD STATE
ACCOUNT A
ACCOUNT B
ACCOUNT C
Read/Write

7.2 结构解析

bucket tree其实是揉合了两种不同的数据结构组合而成，这两种数据结构为：

- merkle树
- 哈希表

因此在介绍bucket tree的结构之前，我们首先简要地介绍一下上述两种数据结构。

7.2.1 merkle树

Merkle树是由计算机科学家 Ralph Merkle 在很多年前提出的，并以他本人的名字来命名，在比特币网络中用到了这种数据结构来进行数据正确性的验证。

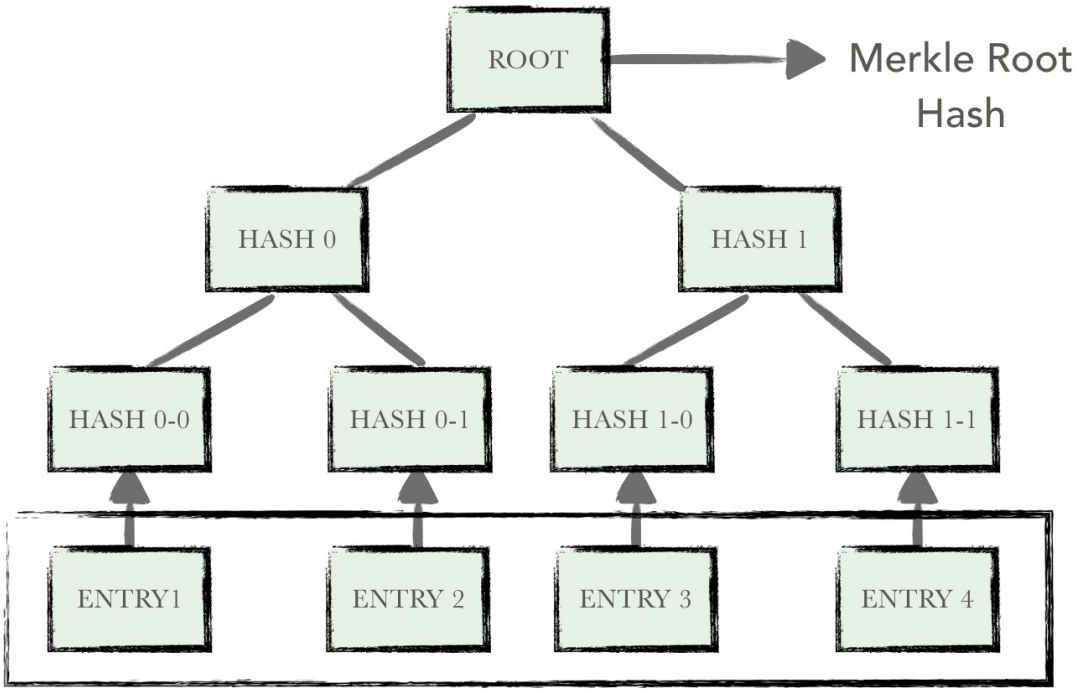
在比特币网络中，merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。此外，由于merkle树的存在，使得在比特币这种公链的场景下，扩展一种“轻节点”实现简单支付验证变成可能。

特点

- 默克尔树是一种树，大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；
- 默克尔树叶子节点的value是数据项的内容，或者是数据项的哈希值；
- 非叶子节点的value根据其孩子节点的信息，进行Hash计算得到的；

原理

在比特币网络中，merkle树是自底向上构建的。在下图的例子中，首先将ENTRY1-ENTRY4四个单元数据哈希化，然后将哈希值存储至相应的叶子节点。这些节点是Hash0-0, Hash0-1, Hash1-0, Hash1-1



将相邻两个节点的哈希值合并成一个字符串，然后计算这个字符串的哈希，得到的就是这两个节点的父节点的哈希值。

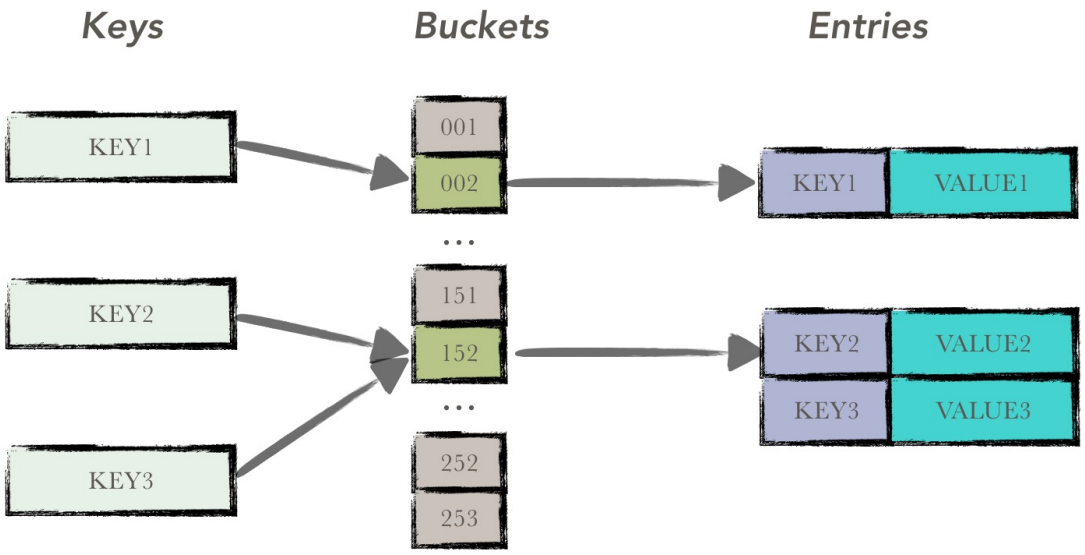
如果该层的树节点个数是单数，那么对于最后剩下的树节点，这种情况就直接对它进行哈希运算，其父节点的哈希就是其哈希值的哈希值（对于单数个叶子节点，有着不同的处理方法，也可以采用复制最后一个叶子节点凑齐偶数个叶子节点的方式）。循环重复上述计算过程，最后计算得到最后一个节点的哈希值，将该节点的哈希值作为整棵树的哈希。

若两棵树的根哈希一致，则这两棵树的结构、节点的内容必然相同。

采用merkle树的优势是：当某一个节点的内容发生变化时，仅需要重新计算**从该节点到根节点路径上所有树节点的哈希**，即可重新得到一个可以代表整棵树状态的哈希值。也正是因为merkle树的这个特点，使得bucket tree能够避免许多不必要的计算开销，拥有快速计算账户状态哈希的能力。

7.2.2 哈希表

哈希表，也称散列表，是大家非常熟悉的数据结构，是根据键（key）而直接访问在内存存储位置的。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做**散列表**。



有关于哈希表的描述，在此就不再赘述了。在bucket tree中，使用哈希表来维护原始的数据。

7.2.3 bucket tree

bucket tree由两部分组成：底层的哈希表以及上层的默克尔树。也就是bucket tree其实是一棵建立在哈希表上的默克尔树。

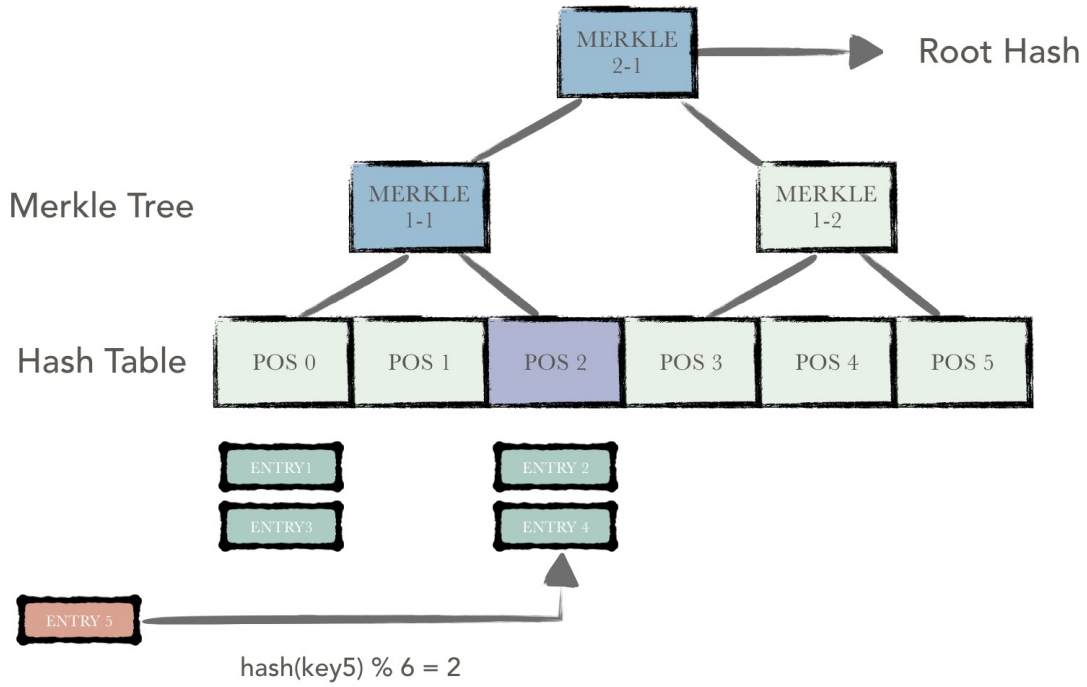
哈希表由一系列哈希桶（bucket）组成，每个桶中存储着若干被散列到该桶中的数据项（entry），所有数据项按序排列。每一个桶有一个哈希值用来表示整个哈希桶的状态，该哈希值是根据桶内所有数据项的内容进行哈希计算得到。

除了底层的哈希表以外，上层是一系列的merkle树节点。一个merkle树节点对应着下一层的n个哈希桶或者merkle树节点。这个n也称作merkle树的聚合度。该merkle树节点中维护着这n个孩子节点的哈希值，且merkle树节点本身的哈希值是根据这n个孩子节点的哈希值计算得到。

如此不断迭代，最终最上层的树节点是整棵树的根节点，该节点的哈希值就代表着整棵树的哈希。

如此设计的目的是：

- 利用merkle树的特点，使得每次树状态改变，重新哈希计算的计算代价最小；
- 利用哈希表进行底层数据的维护，使得数据项均匀分布；



例如上图中，一条新的数据项entry5插入，该数据项被散列到POS为2的桶中。该桶，即从该桶至根节点上所有的节点被标为粉红色，即为脏节点。仅对这些脏节点进行哈希重计算，便可得到一个新的哈希值用来代表新的树状态。

由于bucket tree是一棵固定大小的树（即底层的哈希表容量在树初始化之后，就无法更改了），随着数据量的增大，采用散列函数将所有的数据项进行均匀散列可以避免数据聚集的情况发生。

此外，bucket tree有两个重要的可调参数：

- capacity
- aggreation

前者表示哈希表的容量，该值越大，整棵树相对来说能够容纳的数据项的个数就越多，在聚合度不变的前提下，树高越高，从叶子节点到根节点路径上的树节点个数也越多，哈希计算次数增加；

后者表示一个父节点对应的孩子节点的个数，该值越大，表示树的收敛速度越快，在哈希表容量不变的前提下，树高更低，从叶子节点到根节点路径上的树节点个数也越少，哈希计算次数减少；但是每个默克尔树节点的size就越大，增加数据库IO开销；

7.2.4 哈希桶

哈希桶的定义如下，由一系列的数据项组成，注意这些数据项是按key的字典序排序的，每一个数据项即代表了一条用户数据（可以优化为仅存储用户数据的哈希值）。

```
type Bucket []*DataEntry
```

7.2.5 merkle节点

merkle节点的定义如下，主要的字段为与其相关的孩子节点列表，该列表中的每一个元素都是一个孩子节点的哈希值。

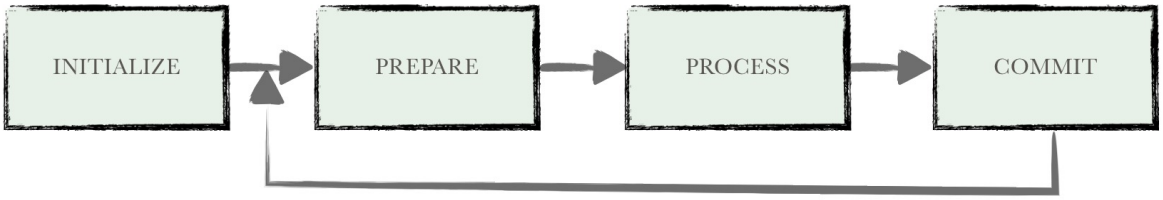
```
// MerkleNode merkleNode represents a tree node except the lowest level's hash␣
↪bucket.
// Each node contains a list of children's hash. It's hash is derived from children
↪'s content.
```

```
// If the aggreation is larger(children number is increased), the size of a merkle␣
↪node will increase too.
type MerkleNode struct {
    pos      *Position
    children [][]byte
    dirty    []bool
    deleted  bool
    lock     sync.RWMutex
    log      *logging.Logger
}
```

7.3 核心操作

bucket tree的计算过程可以分为以下四阶段：

1. Initialize
2. Prepare
3. Process
4. Commit



7.3.1 Initialize

在初始化阶段，主要进行构建树形态的构建，cache的初始化以及历史数据的恢复（从db中读取最新的根节点的哈希）

所谓树形态的构建就是利用用户配置的capacity及aggreation两个参数构建树的结构。构建函数如下：

```
var (
        curlevel  int
        curSize   int = cap
        levelInfo     = make(map[int]int)
    )
    levelInfo[curlevel] = curSize
    for curSize > 1 {
        parSize := curSize / aggr // 根据收敛系数计算下一层的节点个数
        if curSize%aggr != 0 {
            parSize++
        }
        curSize = parSize
        curlevel++
        levelInfo[curlevel] = curSize
    }
    conf.lowest = curlevel
    for k, v := range levelInfo {
        conf.levelInfo[conf.lowest-k] = v // 将每一层的节点个数信息倒置，使得根节点处
于0层，哈希桶处于最高层
    }
```

此外，bucket tree为了（1）增加读取的效率（2）防止写丢失，增加了两个cache用于缓存哈希桶和merkle节点的数据。这两个cache都是利用LRUCache实现的，每次更新时都会同步地更新cache中的内容。如此，下一次bucket tree进行哈希计算时，便可以从cache中命中热数据，尽量避免磁盘的读取。

至于防止写丢失是因为在hyperchain中，validation和commit是两个独立异步的过程，因此在进行完区块100的validation过程时，可能会立刻基于100的状态直接进行区块101的validation。而此时区块100执行过程中对账本的修改还没有被提交的数据库中，因此为了"防止写丢失"，这些内容需要从缓存中命中。

倘若此刻cache发生了容量过小未提交的内容被驱除的情况，就会导致区块101的validation的结果与其他节点不一致（通常为主节点），此时会依赖于RBFT算法进行故障处理。

其中bucketcache是用来存储哈希桶的数据，每一个哈希桶为一个cache数据项。存在的问题是，一个哈希桶本身由若干条数据项组成，随着运行时间增长，一个哈希桶的size会越来越大，导致内存占用量不断上升。

merkleNodeCache是用来存储除最高层以外的所有merkle节点数据的，merkle节点个数是固定的，每个节点的size也是有上限的，因此merkleNodeCache不存在内容占用量变大的问题。

7.3.2 Prepare

在准备阶段，bucket tree会接收由用户传入的修改集，并利用修改集的内容构建脏的哈希桶集。注意，返回的哈希桶中，内部的数据项是按字典序升序排列的。

```
func newBuckets(prefix string, entries Entries) *Buckets {
    buckets := &Buckets{make(map[Position]Bucket)}
    for key, value := range entries {
        buckets.add(prefix, key, value)
    }
    for _, bucket := range buckets.data {
        sort.Sort(bucket)
    }
    return buckets
}
```

7.3.3 Process

process也就是哈希重计算阶段，可以分为两部分（1）脏哈希桶的哈希重计算（2）脏merkle节点的哈希重计算。

脏哈希桶计算

如上图所示，在bucket tree中新插入两条数据项entry5, entry6。entry5得到的散列地址为Pos2，entry6得到的散列地址为Pos5。

哈希桶计算存在一个合并的操作，即在Pos2，需要将新插入的数据，与历史的数据进行一个**合并**，且按照固定的排序算法进行重排序，最终得到一个新的哈希桶，包含了所有的新旧数据，且按序排列。

每个哈希桶的哈希值为当前桶中数据的进行哈希计算得到的结果。

如图所示，Pos2与Pos5为两个脏的哈希桶，计算完成之后，将父节点中对应的孩子哈希值置为新的桶哈希值，即Merkle1-1，Merkle1-2为两个脏的merkle节点。

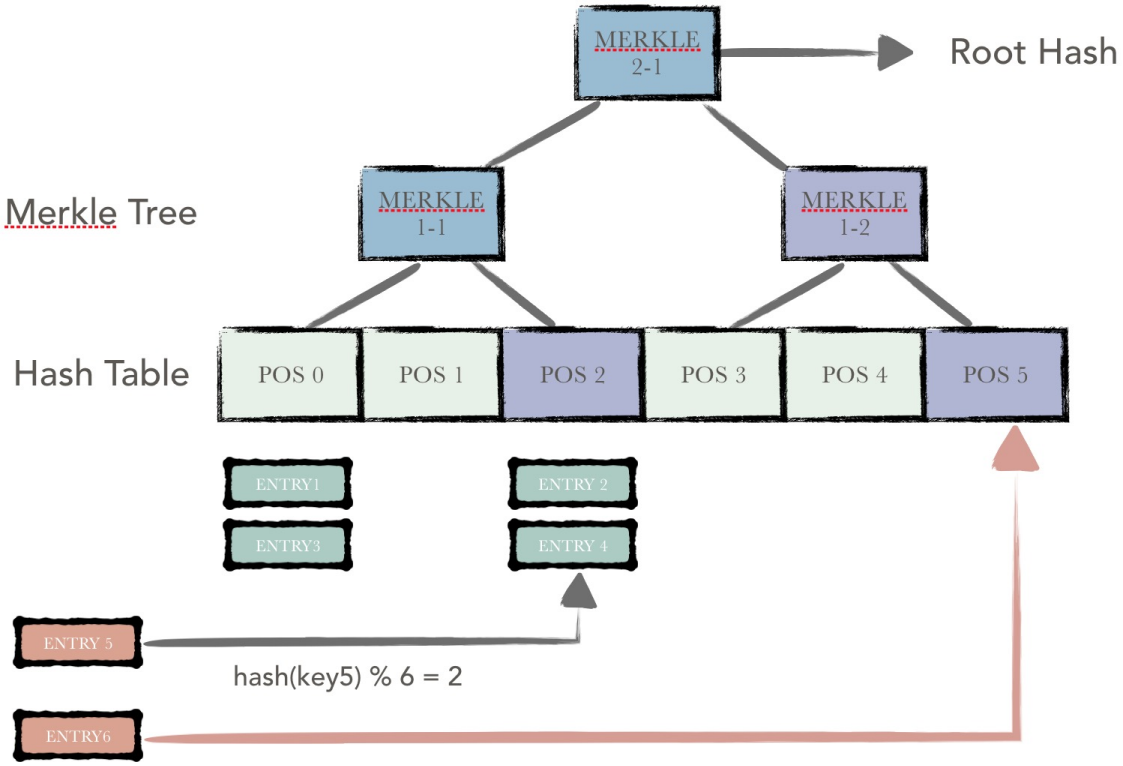
脏merkle节点计算

当哈希桶计算完成之后，便可以进行merkle节点的哈希计算。这步中仅对脏的merkle节点进行哈希重计算。

注意，merkle节点的哈希计算是分层进行的。

每个merkle节点维护其孩子节点的哈希值，若下一层的孩子节点哈希值发生变化，会在之前的计算中，就将最新的哈希值置到父节点中；对于没有发生变化的孩子节点，直接使用历史的哈希值即可。

每个merkle节点的哈希值，为其所有孩子节点哈希值的哈希计算得到。



7.3.4 Commit

计算完成之后，需要将最新的哈希桶数据、merkle节点数据进行持久化。

除此以外，所有的哈希桶数据、merkle节点数据都会被存储到缓存中，对于热数据，既可以提高数据的查找效率，也可以避免数据的写丢失情况。

CHAPTER 8

智能合约

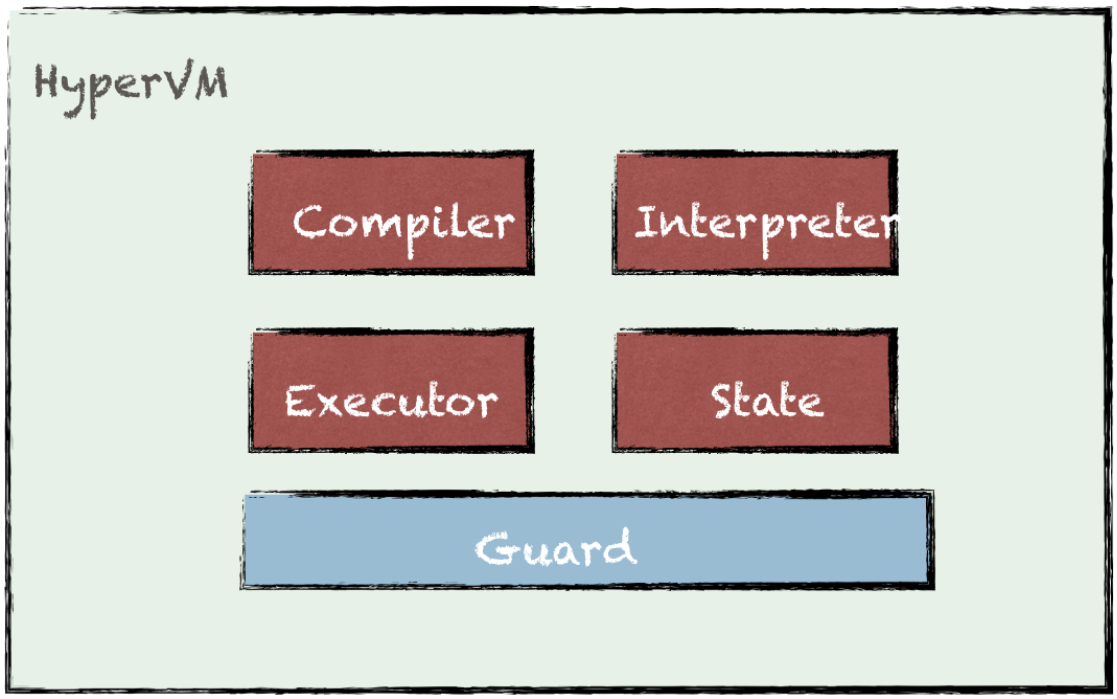
8.1 1. 智能合约简介

注解： 智能合约是指部署在区块链上的一段可以自动执行条款的计算机程序。智能合约能够根据外界输入信息自动执行预先定义好的协议并完成区块链内部相关状态的转移。

广泛意义上的智能合约还包括智能合约编程语言、编译器、虚拟机、事件、状态机、容错机制等。其中对智能合约影响较大的是智能合约编程语言以及其执行引擎。智能合约虚拟机一般为了安全起见是作为沙箱被分装起来，整个执行环境完全被隔离。虚拟机内部执行的智能合约不允许接触网络、文件系统、进程线程等系统资源。不同智能合约的安全性等级、表达的丰富性有所不同，Hyperchain系统自主研发智能合约的执行引擎HyperVM是一种通用智能合约引擎设计，允许多种不同智能合约引擎接入。目前的实现了兼容Ethereum的Solidity语言的HyperEVM和支持Java语言的智能合约引擎HyperJVM.

8.2 2. 智能合约引擎HyperVM

HyperVM 是Hyperchain自主研发的可插拔智能合约引擎通用框架，允许不同智能合约执行引擎接入。如下图所示是HyperVM的架构示意图，HyperVM的架构图提供了智能合约编译、执行等相关主要组件。其中，Compiler提供了智能合约编译相关功能，Interpreter和Executor则提供了智能合约解释以及执行相关功能，State组件赋予智能合约操作区块链账本的相关功能。Guard模块提供智能合约安全保障相关机制。



8.2.1 2.1 HyperEVM

为了最大程度利用开源社区在智能合约技术方面的研究和积累，提高智能合约的可重用性以及兼容性。HyperEVM的实现采用了完全兼容Ethereum的智能合约规范，使用Solidity作为智能合约开发语言，底层使用了优化了的Ethereum虚拟机EVM。 如下图所示为HyperEVM智能合约执行流程图:

HyperEVM执行一次交易之后会返回一个执行结果，系统将其保存在被称为交易回执的变量中，之后平台客户端可以根据本次的交易哈希进行交易结果的查询。HyperEVM的执行流程如下:

1. HyperEVM接收到上层传递的transaction，并进行初步的验证;
2. 判断transaction类型，如果是部署合约则执行3，否则执行4;
3. HyperEVM新建一个合约账户来存储合约地址以及合约编译之后的代码;
4. HyperEVM解析transaction中的交易参数等信息，并调用其执行引擎执行相应的智能合约字节码;
5. 指令执行完成之后，HyperVM会判断其是否停机，否的话跳转步骤2，否则执行步骤6;
6. 判断HyperVM的停机状态是否正常，正常则结束执行，否则执行步骤7;
7. 进行Undo操作，状态回滚到本次交易执行之前。

执行指令集模块是HyperEVM执行模块的核心，指令的执行模块有两种实现，分别是基于字节码的执行以及更加复杂高效的即时编译（Just-in-time compilation）。 字节码执行的方式比较简单，HyperEVM实现的虚拟机会有指令执行单元。该指令执行单元会一直尝试执行指令集，当指定时间未执行完成，虚拟机会中断计算逻辑，返回超时错误信息，以此防止智能合约中的恶意代码执行。

JIT方式的执行相对复杂，即时编译也称为及时编译、实时编译，是动态编译的一种形式，是一种提高程序运行效率的方法。通常，程序有两种运行方式：静态编译与动态直译。静态编译的程序在执行前全部被翻译为机器码，而直译执行的则是边翻译边执行。即时编译器则混合了这二者，一句一句编译源代码，但是会将翻译过的代码缓存起来以降低性能损耗。相对于静态编译代码，即时编译的代码可以处理延迟绑定并增强安全性。JIT模式执行智能合约主要包含以下步骤:

1. 将所有同智能合约相关的信息封装在合约对象中，然后会通过该代码的哈希值去查找该合约对象是否已经存储编译。合约对象有四个常见状态，即：合约未知，合约已编译，合约准备好通过JIT执行，合约错误。

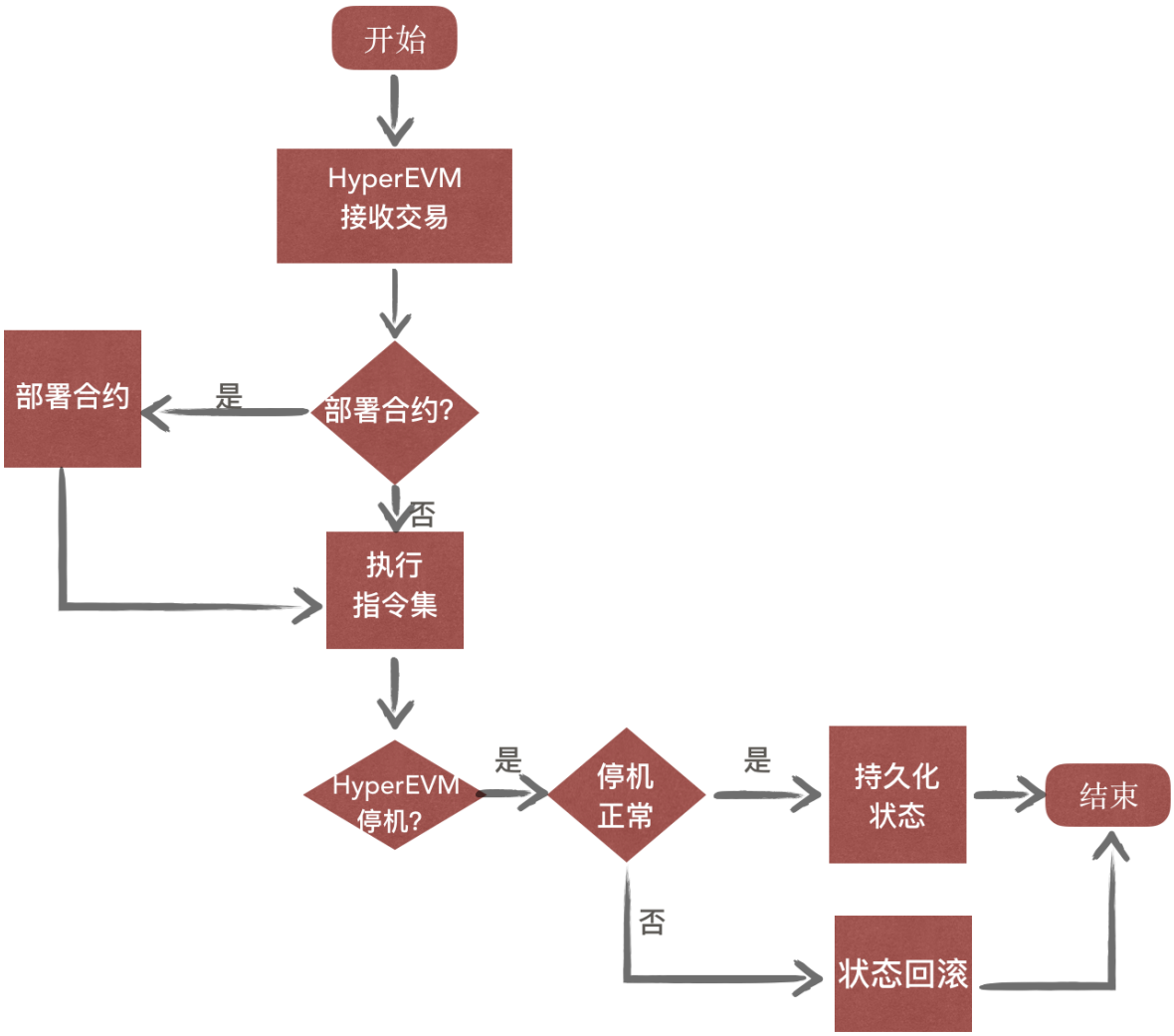


图 8.1: hyperevm-flow

2. 如果合约状态是合约准备好通过JIT执行，则HyperEVM会选择JIT执行器来执行该合约。执行过程中虚拟机将会对编译好的智能合约进一步编译成机器码并对push、jump等指令进行深度优化。
3. 如果合约状态是合约未知的情况下，HyperEVM首先需要检查虚拟机是否强制JIT执行，如果是则顺序编译并通过JTI的指令进行执行。否则，开启单独线程进行编译，当前程序仍然通过普通的字节码编译。下次虚拟机执行过程中再次遇到相同编码的合约时，虚拟机会直接选择经过优化的合约。这样合约的指令集由于经过了优化，该合约的执行和部署的效率能够获得较大的提高。

8.3 3. 智能合约使用

8.3.1 3.1 基于Solidity智能合约案例

8.3.2 编写合约

基于Solidity的智能合约同基于JS的程序类似，由一系列变量和相关函数组成。如下所示为模拟简单累加器功能的智能合约。我们以此为例简单介绍基于Solidity智能合约的基本组成部分。

```
contract Accumulator{
    uint32 sum = 0;
    function increment(){
        sum = sum + 1;
    }

    function getSum() returns(uint32){
        return sum;
    }

    function add(uint32 num1,uint32 num2) {
        sum = sum+num1+num2;
    }
}
```

Accumulator合约说明：

- 基于solidity的智能合约以关键字`contract`开头，类似Java等语言中的class关键字；
- 合约内部可以有变量和函数，上述sum 为uint32类型的简单变量，Solidity智能合约还支持map等集合类型；
- 合约允许定义执行函数以`function`关键字定义；

基于Solidity语言编写智能合约的详细规范参考Solidiy官方网站

8.3.3 编译合约

Hyperchain的智能合约的编译既可以采用Solidity官方编译器编译，也可以使用Hyperchain提供的智能合约部署JSON-RPC的接口进行编译（这种场景需要在安装Hyperchain的宿主机安装Solidity编译器sloc）。

调用Hyperchain编译Solidity智能合约命令如下：

```
curl -X POST --data
'{
    "jsonrpc":"2.0",
    "namespace":"global",
    "method":"contract_compileContract",
    "params":["contract_code"],
    "id":1
}'
```

合约编译接口调用的返回如下：

```
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "abi": [
      "[{\"constant\":false,\"inputs\":[{\"name\":\"num1\",\"type\":\"uint32\"},{\
↪"name\":\"num2\",\"type\":\"uint32\"}],\"name\":\"add\",\"outputs\":[],\"payable\
↪":false,\"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\":\
↪"getSum\",\"outputs\":[{\"name\":\"\",\"type\":\"uint32\"}],\"payable\":false,\
↪"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"increment\",\
↪"outputs\":[],\"payable\":false,\"type\":\"function\"}]"
    ],
    "bin": [

↪"0x6060604052600080546463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633a
↪"
    ],
    "types": [
      "Accumulator"
    ]
  }
}
```


其中字段bin对应的内容为该合约的字节码表示，该bin内容供后续部署使用。

8.3.4 部署合约

Hyperchain部署Solidity命令如下：


```
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global",  "method":
↪"contract_deployContract", "params":[{
"from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc ",
"nonce":5373500328656597,
"payload":
↪"0x6060604052600080546463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633a
↪",
"signature":
↪"0x388ad7cb71b1281eb5a0746fa8fe6fda006bd28571cbe69947ff0115ff8f3cd00bdf2f45748e0068e49803428999
↪",
"timestamp":1487771157166000000
}],"id":"1"}'
```


部署合约返回如下，其中result字段内容为该合约在区块链中的地址，后期对该合约的调用需要指定该合约地址。


```
{
    "jsonrpc": "2.0",
    "namespace":"global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "0x406f89cb205e136411fd7f5befbf8383bbfdec5f6e8bcfe50b16dcff037d1d8a"
}
```

8.3.5 调用合约

Hyperchain调用命令如下，其中payload为调用合约中函数以及其参数值的编码结果，to为所调用合约的地址。

```
curl localhost:8081 --data

'{
    "jsonrpc":"2.0",
    "namespace":"global",
    "method": "contract_invokeContract",
    "params":[{
            "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
            "nonce":5019420501875693,
            "payload":
↪"0x3ad14af3000000000000000000000000000000000000000000000000000000000000000100000000000000000000
↪",
            "signature":
↪"0xde467ec4c0bd9033bdc3b6faa43a8d3c5dcf393ed9f34ec1c1310b0859a0ecba15c5be4480a9ad2aaaea8416324c
↪",
            "simulate":false,
            "timestamp":1487773188814000000,
            "to":"0x313bbf563991dc4c1be9d98a058a26108adfcf81"
            }],
    "id":"1"
}'
```

合约调用会立即给客户端返回该交易的哈希值，后期可以根据该交易的哈希值查询具体交易的执行结果。

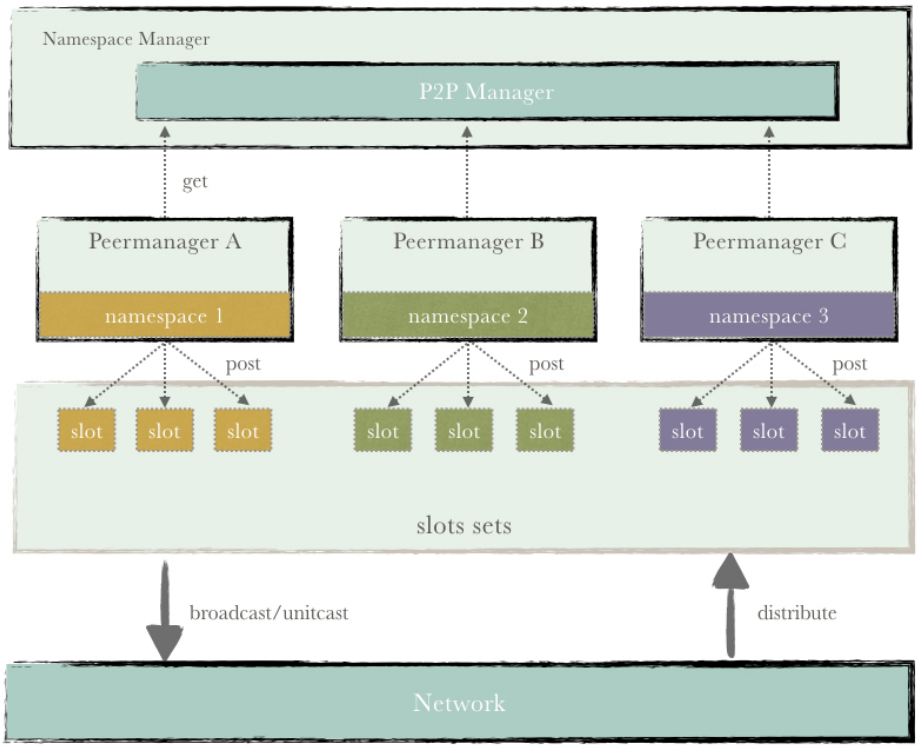
```
{
    "jsonrpc":"2.0",
    "namespace":"global",
    "id":1,
    "code":0,
    "message":"SUCCESS",
    "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

CHAPTER 9

P2P

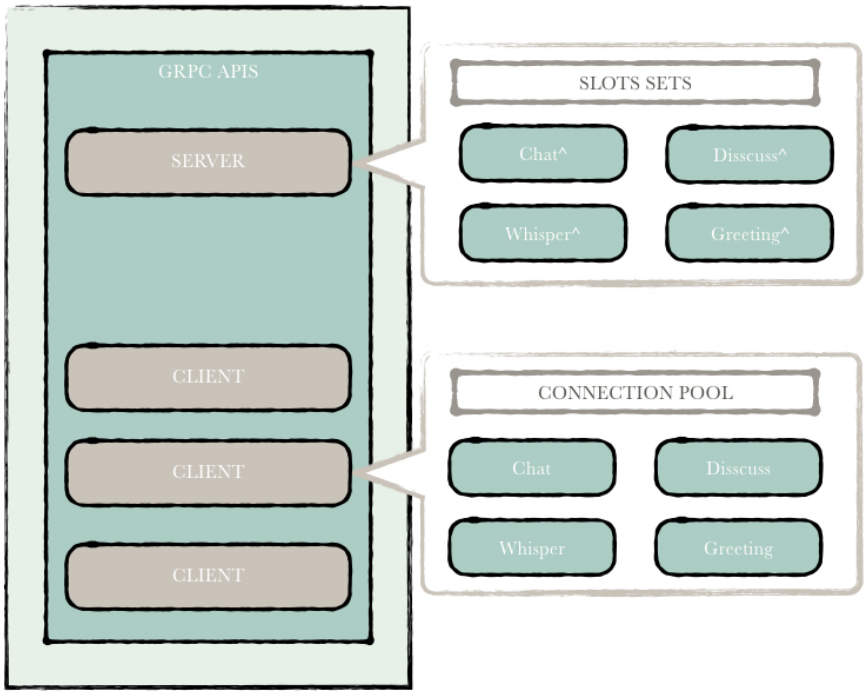
9.1 1. 概述

P2P模块是Hyperchain底层网络通信模块，它保证了通信链路的数据传输安全，用户可以通过配置文件来配置是否启用传输安全TLS、是否启动数据传输加密（或更换数据加密算法）。在该模块中，物理连接（Network层）与逻辑连接分离，模块整体架构如下图所示：



9.2 2. Hypernet

Hypernet作为Hyperchain的底层通信基础设施，通过注册slot的形式向上层提供网络通信服务，其主要的功能包括通信链路的建立，数据传输，链路安全，链路活性控制等功能。其拥有`Server`和`Client`两个重要成员。整体架构图如下：



9.2.1 Server

在Hypernet中，Server负责注册网络插口(slot)、监听服务、分发来自Client的各类消息。

网络插口slot

slot作为Hypernet同上层进行通信的主要机制，其实现其实是一个多维的线程安全的map映射，将相关方法映射到不同的消息处理器中进行处理。

slots作为Server成员，拥有一组对应到不同namespace中的slot，分别处理来自不同namespace消息。

9.2.2 Client

Client与Server相对应，主要用于处理不同的消息发送请求。通常一个Client对应于多个不同的远端Server（因为一个节点会与多个节点相连），并且与不同Server通信，交由Sever的信息将会分发到不同的namespace中的slot去处理。

TLS

传输层安全是Hyperchain默认开启的功能，采用Hyperchain内部的TLSCA签发的证书进行安全通信，从传输层面保障信息通信的安全性。进一步地，该选项是可选的。

TLS 能够在传输层保障信息传输的安全性，是目前较为通用的网络传输实施标准，在几乎所有的网络安全传输中都采用了该技术。

9.3 3. P2PManager

`P2PManager`用于分配不同namespace中的`PeerManager`，它在全局中只有一个实例。

9.3.1 PeerManager

`PeerManager`主要负责以下几个部分：

- 对外提供不同的消息发送服务接口；
- 提供相应的消息向上推送服务；
- 使用`PeerManagerEventHub`这个消息中间件对`PeerManager`的控制消息进行分发管理，进行整个逻辑网络的状态维护，同时承担较为复杂的消息处理逻辑。

在整个网络中，节点可称为`Node`或`Peer`，下面来看这两者的区别。

Peer

逻辑上的所有remote节点都称为peer, 一个远端节点对应一个peer，peer主要用于处理逻辑上的消息发送请求。其主要工作就是对消息进行加密，然后调用Hypernet Client的对应的消息发送方法，并且消息中需要附着namespace信息。

Node

Node即本地节点，它也是逻辑上的server，主要负责逻辑上的节点消息处理，对从网络中接收到的消息进行解密然后抛给Hyperchain消息中间件`eventhub`，由消息中间件识别这是哪一类消息应该交给哪一个模块去处理，比如共识模块、执行模块。

数据传输加密

数据传输加密是指对网络上传输的交易信息和通信消息的加密，根据用户的需求可以对所有在Hyperchain上传输的信息进行加密，其加密方案同TLS类似，先通过ECDH算法协商对应的会话密钥，然后利用该会话密钥对业务信息进行加密，对端进行解密。所有的节点之间的通信都会利用不同的会话密钥进行加密。这是对传输层安全的一个补充。目前hyperchain的消息可以通过配置进行对称加密，如果有更加复杂的消息加密需求可以使用该方式进行处理。

CHAPTER 10

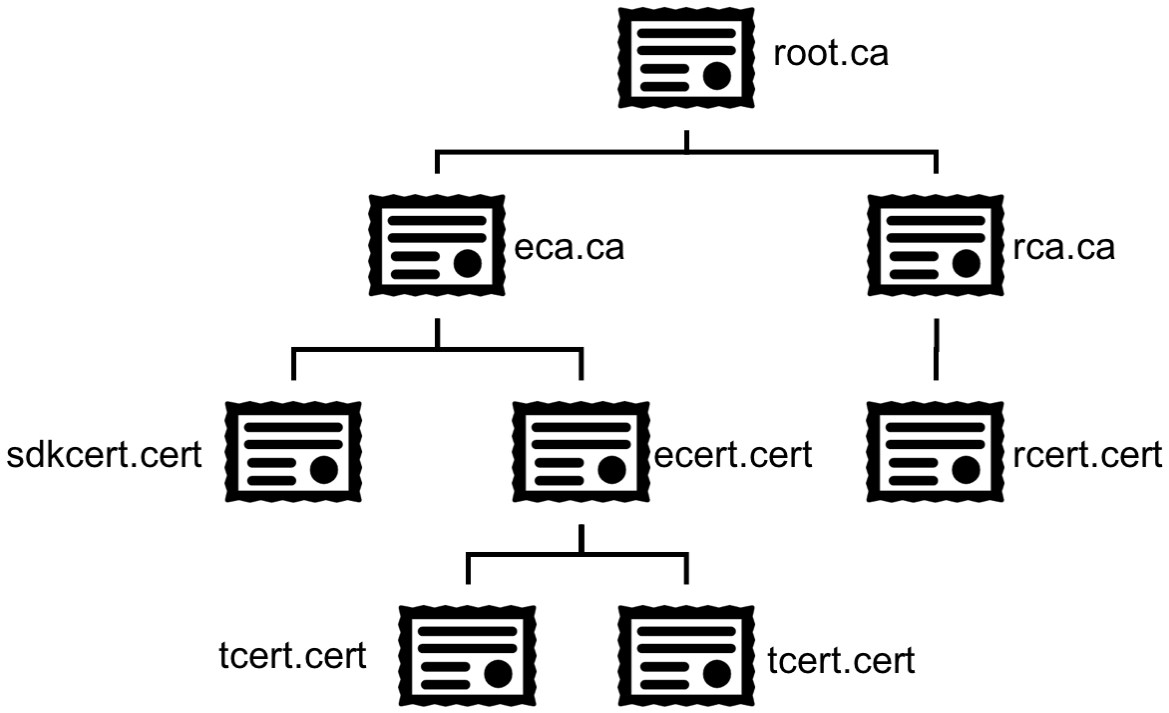
数字证书

10.1 1.概述

Hyperchain是一个联盟链级别的区块链服务平台，拥有精确的权限控制粒度，需要通过多级的CA证书来进行权限控制。权限控制主要分为两个方面：

- 节点权限控制
- 交易权限控制

首先我们需要知道的是，权限控制是Namespace级别的，即每个Namespace就会拥有一个相应CaManager来进行CA证书的管理以及Namespace级别的权限控制。以下是我们的PKI体系(证书体系)图：



- root.ca(根证书颁发机构):它代表PKI体系中的信任锚。数字证书的验证遵循信任链。根CA是PKI层次结构中最上层的CA，用于签发证书认证机构以及角色证书准入认证机构。
- eca.ca(准入证书颁发机构):用于给节点颁发节点准入证书(ecert)和sdk证书(sdkcert)。
- rca.ca(角色证书颁发机构):用于给节点颁发角色证书(rcert)。
- ecert.cert(准入证书):准入证书是长期证书，是颁发给节点的，用于节点的准入，若无准入证书，则该节点无法加入此Namespace，同时准入证书也作为交易证书的颁发机构用于颁发交易证书。
- rcert.cert(角色证书):角色证书是长期证书，是颁发给节点的，用于节点的角色的认证，若无角色证书，则此节点为NVP，无法参与共识，反之则为VP节点。
- sdkcert.cert(sdk证书):SDK证书时颁发给SDK的，用于进行SDK的身份认证，同时SDK证书时获取交易证书的（Tcert）的判别依据。
- tcert.cert(交易证书):SDK发送交易时需携带交易证书，若无交易证书或证书验证失败，则抛弃交易。

10.2 2.证书简述

Hyperchian区块链平台证书均符合ITU-T X.509国际标准，它仅包含了公钥信息而没有私钥信息，是可以公开进行发布的，所以X.509证书对象一般都不需要加密。X.509证书的格式通常如下:

```
---BEGIN CERTIFICATE---
.....PEM编码的X.509证书内容(省略).....
---END CERTIFICATE---
```

PEM编码全称是Privacy Enhanced Mail，是一种保密邮件的编码标准。通常来说，对信息的编码过程基本如下:

- 信息转换为ASCII码或其他编码方式，比如采用DER编码。
- 使用对称加密算法加密经过编码的信息。
- 使用BASE64对加密码后的信息进行编码。
- 使用一些头定义对信息进行封装，主要包含了进行正确解码需要的信息。

除此以外Hyperchain区块链平台证书内具体包含以下信息:

1、X.509版本号：指出该证书使用了哪种版本的X.509标准，版本号会影响证书中的一些特定信息。目前的版本是3。

2、证书持有人的公钥：包括证书持有人的公钥、算法(指明密钥属于哪种密码系统)的标识符和其他相关的密钥参数。

3、证书的序列号：由CA给予每一个证书分配的唯一的数字型编号，当证书被取消时，实际上是将此证书序列号放入由CA签发的CRL（Certificate Revocation List证书作废表，或证书黑名单表）中。这也是序列号唯一的原因。

4、主题信息：证书持有人唯一的标识符(或称DN-distinguished name)这个名字在 Internet上应该是唯一的。DN由许多部分组成，看起来象这样:

```
CN=Bob Allen, OU=Total Network Security Division
O=Network Associates, Inc.
C=US
```

这些信息指出该科目的通用名、组织单位、组织和国家或者证书持有人的姓名、服务处所等信息。

5、证书的有效期：证书起始日期和时间以及终止日期和时间；指明证书在这两个时间内有效。

6、认证机构：证书发布者，是签发该证书的实体唯一的CA的X.509名字。使用该证书意味着信任签发证书的实体。(注意：在某些情况下，比如根或顶级CA证书，发布者自己签发证书)

7、发布者的数字签名：这是使用发布者私钥生成的签名，以确保这个证书在发放之后没有被撰改过。

8、签名算法标识符：用来指定CA签署证书时所使用的签名算法。算法标识符用来指定CA签发证书时所使用的公开密钥算法和HASH算法。

10.3 3.CA相关配置

CA所需配置在namespace.toml配置文件内，具体参数如下：

```
[encryption]
[encryption.ecert]
 eca    = "config/certs/eca.cert"
 ecert  = "config/certs/ecert.cert"
 priv   = "config/certs/ecert.priv"

[encryption.rcert]
 #if you do not have rcert, leave this item blank
 rca    = "config/certs/rca.cert"
 rcert  = "config/certs/rcert.cert"
 priv   = "config/certs/rcert.priv"

[encryption.tcert]
 #Tcert whitelist configuration.
 whiteList = false
 listDir  = "config/certs/tcerts"

[encryption.check]
 enable     = true  #enable ERCert
 enableT    = false  #enable TCert
```

首先前六个参数，分别是配置改Namespace的相关证书路径，分别是eca,ecert和ecert对应的私钥，rcar,rcert和rcert对应的私钥。

其次是TCert相关配置，平台支持TCert白名单策略，即在`whiteList=true`时，即开启TCert白名单策略，在`listDir`参数配置下的TCert证书则是立即可用交易证书。反之，当`whiteList`为false时，则不开启白名单策略，只有在该交易证书合法性验证通过，且确定该交易证书确定为本节点且在Namespace下颁布的交易证书时才可完全验证通过。

最后的两个参数则为配置开关参数，`enable`为开启准入证书以及角色证书的校验开关，反之则不进行验证，`enableT`则为开启交易证书验证的开关配置，只有当该参数置为true时才进行交易证书的验证。动态的开关配置也使区块链配台更加灵活。

10.4 4.证书获取以及校验流程

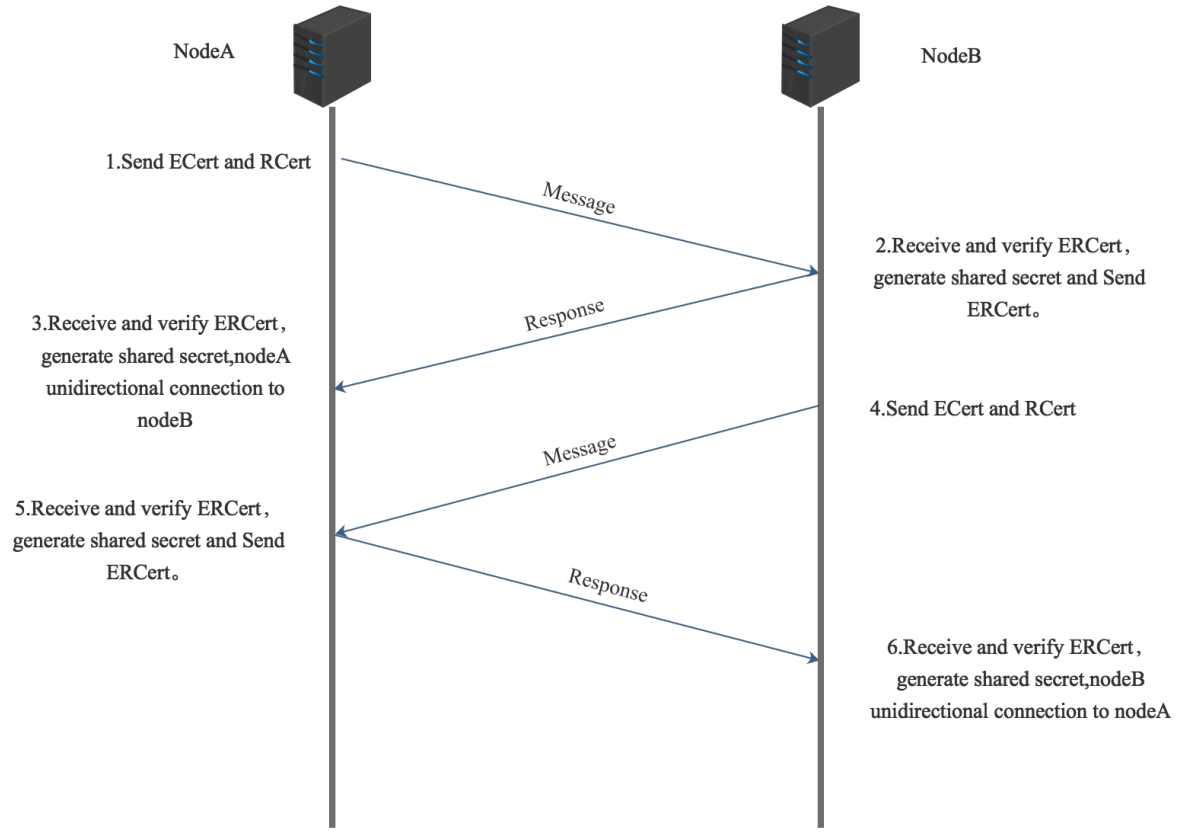
10.4.1 4.1 ECert和RCert

4.1.1 获取

准入证书以及角色证书主要是通过线下颁布进行控制，有Certgen证书签发工具进行证书的生成。

4.1.2 校验

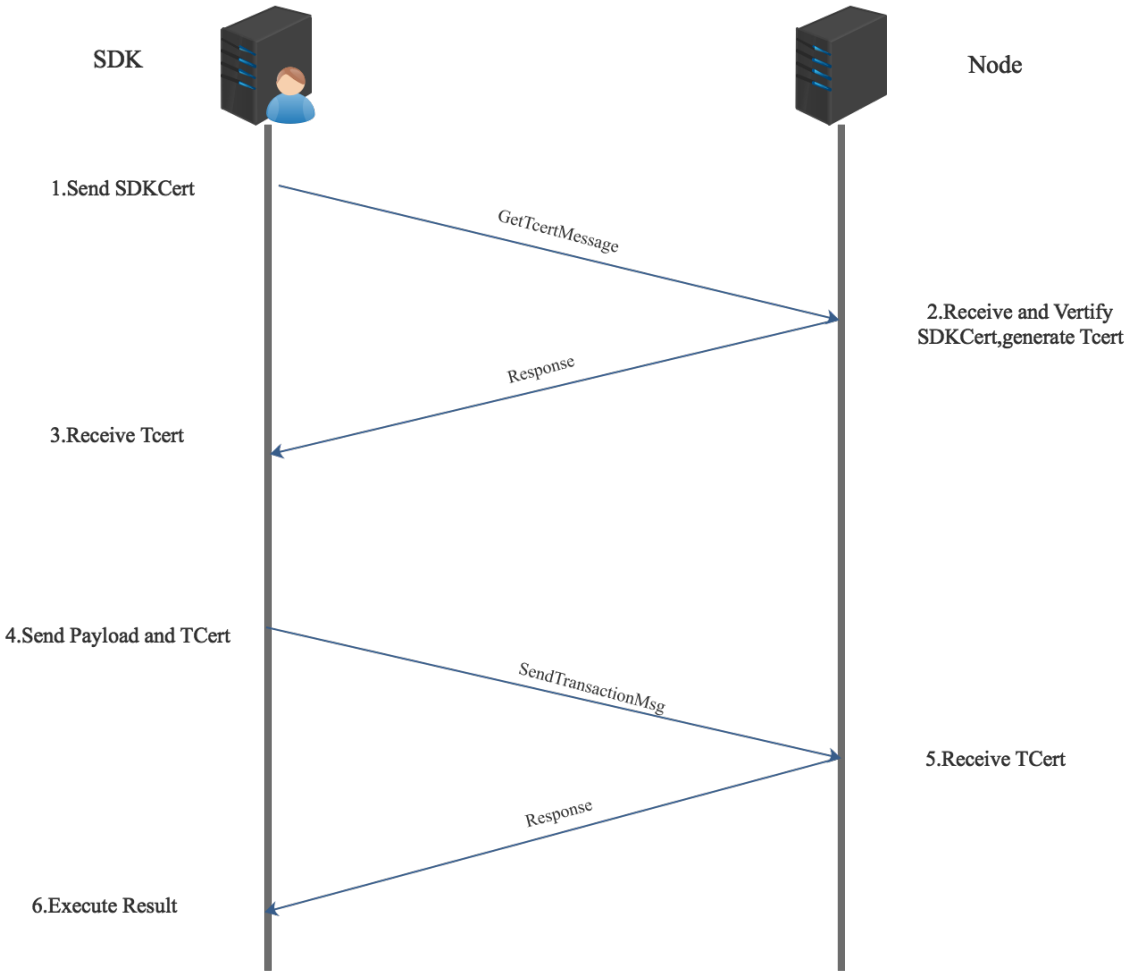
若ECert以及RCert校验开关开启，则具体验证流程如下图所示：



ECert以及RCert在节点初次握手连接时进行了证书的互换以及验证，由此来确定了节点是否允许入链以及相关连接节点的角色信息。

10.4.2 4.2 TCert

TCert的获取以及验证的流程图如下图所示：



4.2.1 获取

首先，SDK或者外部应用需要向连接节点发起GetTcert的消息，该消息需要携带SDKCert对该SDK或外部应用进行身份认证，在认证通过后进行TCert证书的生成以及颁布。

4.2.2 校验

若SDK或者外部应用获取TCert成功后，则接下来的交易需要携带相关交易证书给相关节点进行验证，只有交易证书验证通过后才会进行接下来的交易执行。

CHAPTER 11

分区共识

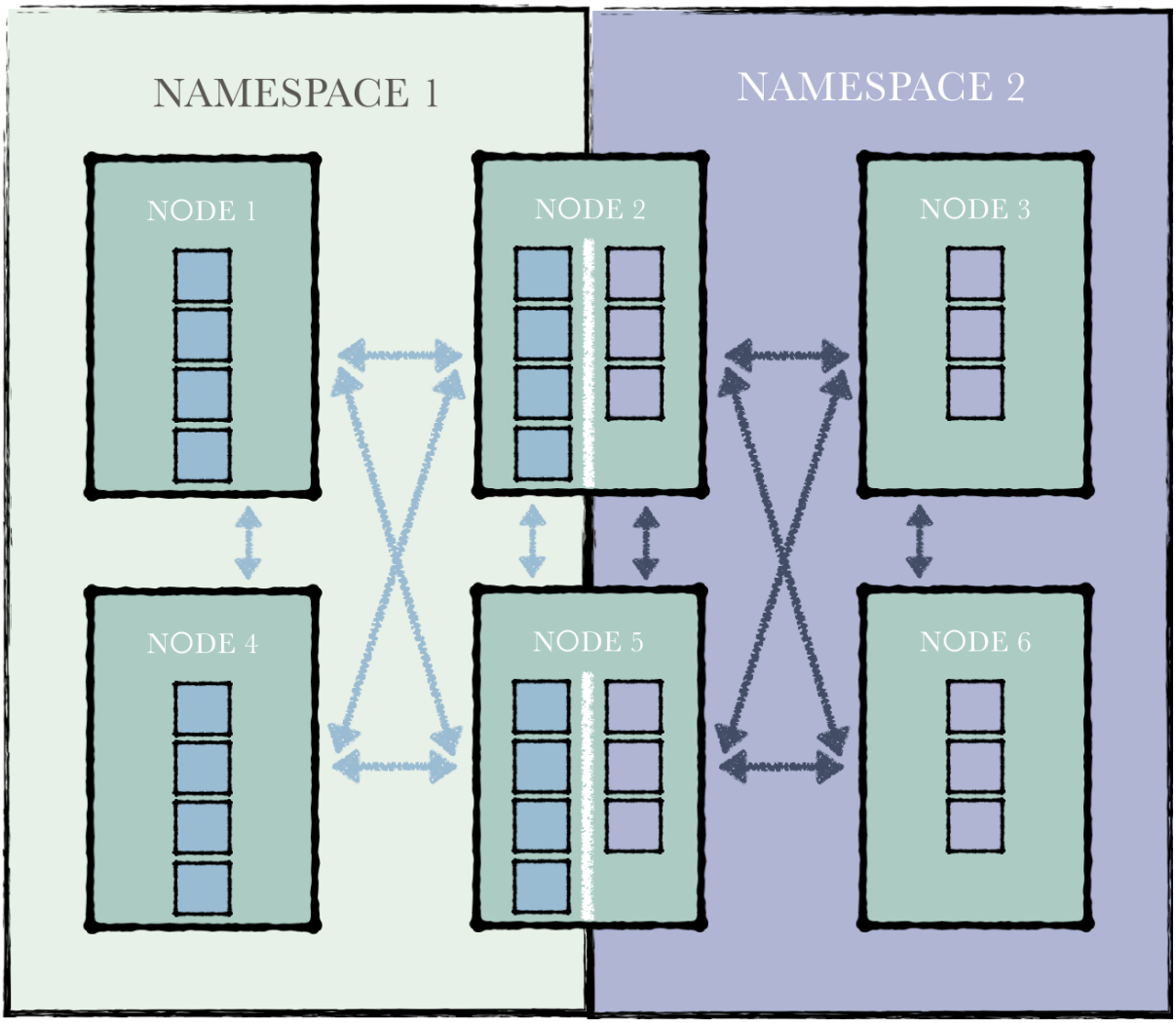
11.1 1.概述

Hyperchain通过Namespace(名字空间)机制实现区块链网络内部交易的分区共识。Hyperchain平台的使用者可以按照名字空间进行业务交易划分，同一个Hyperchain联盟链网络中的节点按照其所参与的业务组成以名字空间为粒度的子区块链网络。名字空间通过业务的交易共识、分发以及存储的物理级别隔离实现业务级别的隐私保护。

11.2 2.集群架构

名字空间可以动态创建，单个Hyperchain节点按照其业务需求可以选择参与一个或者多个名字空间。如下图所示是Namespace机制的整体集群架构图——六个节点参与两个命名空间的集群示意图，Node1、Node2、 Node4和Node5组成namespace1， 而Node2、Node3、Node5和Node6组成namespace2。 其中Node1和Node4仅参与了namespace1，Node3和Node6仅参与了namespace2，而Node2和Node4则同时参与了两个名字空间。名字空间中通过CA认证方式控制节点的动态加入和退出，每个节点可以允许参与到一至多个Namesapce中。

带特定Namespace信息的交易的验证、共识、存储以及传输仅在参与特定Namespace的节点之间进行，不同Namespace之间的交易可实现并行执行。如下图所示Node1仅能参与namespace1中交易的验证以及相应账本的维护而Node2能够同时参与namespace1和namespace2的交易执行和账本维护，但Node2中的namespace1和namespace2的账本互相隔离互不可见。



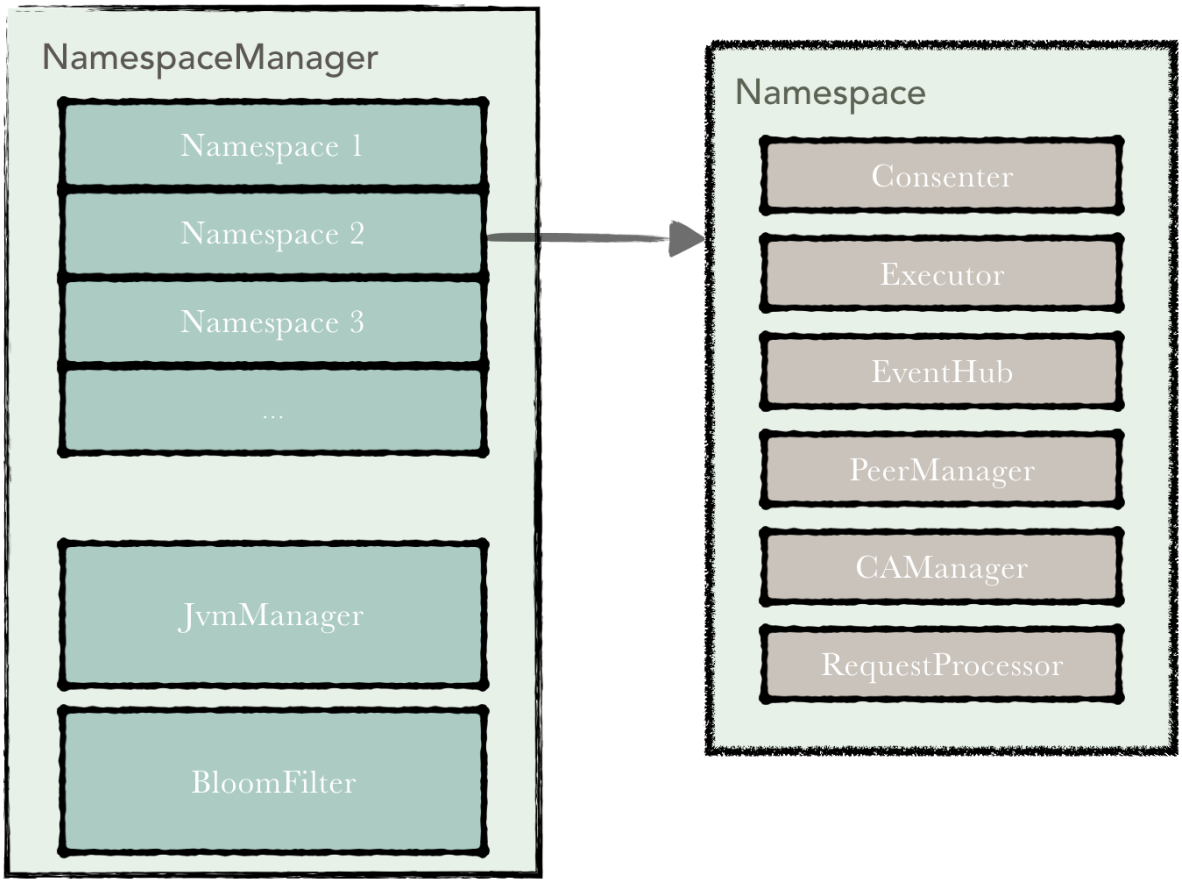
3.节点架构

加入分区共识机制之后的Hyperchain单节点将包含一个 NamespaceManager对象。NamespaceManager是分区共识机制的关键管理组件，负责namespace的注册、启动、停止、注销等一系列的生命周期状态操作。NamespaceManager中包含多个namespace，此外还包含JvmManager和BloomFilter。

具体而言：

- JvmManager 负责管理jvm执行器，JvmManager是否开启需要在配置文件中进行配置；
- BloomFilter 为交易的布隆过滤器，主要负责重复交易的检测，防止重放攻击；

分区共识中的一个分区被称为一个namespace，每个namespace之间互相隔离，包括执行空间以及数据存储空间均做到完全隔离。每个节点默认加入到名为global的namespace中。每个namespace包含consenter,executor,eventHub,peerManager,caManager,requestProcessor等关键组件，这几个关键组件实现各自namespace的共识服务，交易执行存储，模块间异步交互，节点间通讯、身份认证、交易处理等功能。



具体而言：

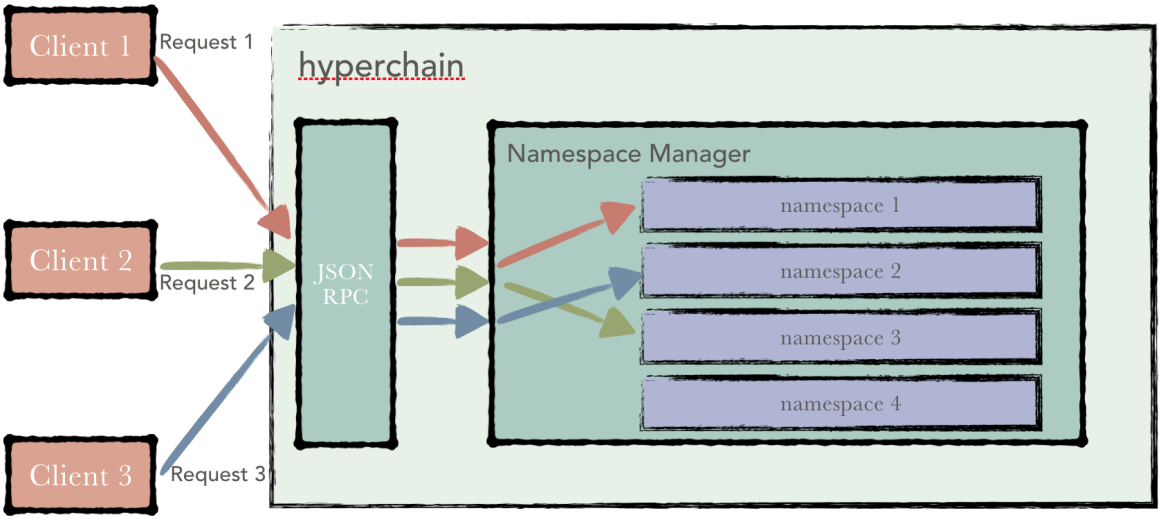
- `Consenter` 提供共识服务，目前支持RBFT算法，负责对交易进行定序，保证同一namespace内的hyperchain节点的账本一致性；
- `Executor` 为执行器，执行器主要负责namespace中的智能合约的调用以及账本状态的维护；
- `EventHub` 为事件总线，是namespace内各关键组件间事件进行异步交互的消息中转中心；
- `PeerManager` 提供节点通讯管理，负责namespace成员之间网络通信；
- `CaManager` 为证书认证中心，负责进行互联网上的身份认证；
- `RequestProcessor` 为请求处理组件。负责处理JSON-RPC消息，最终通过反射调用相应api。

Namespace的生命周期如下图所示。首先在namespaceManager中进行注册，创建并初始化此namespace。之后start此namespace，启动namespace中所有组件的功能，进行Running状态。Stop将停止此namespace中所有组件，最后在namespaceManager注销此namespace。

11.3 4.系统数据流

在引入namespace之后hyperchain节点的交易执行流程有所改变，客户端能够发送交易到其期望的namespace。交易传递到hyperchain平台之后json rpc会首先进行参数检查以及签名验证等前期工作，之后json rpc服务会将请求转发到NamespaceManager，NamespaceManager在会根据请求中的namespace字段信息将交易请求发送到具体的namespace进行执行。不同namespace之间的交易能够并发执行。对应

的namespace将调用RequestProcessor进行交易的处理，首先检查请求的参数，如果没有问题，通过反射调用相应处理函数并返回结果。



CHAPTER 12

密码学算法

12.1 1.概述

Hyperchain区块链平台支持多级密码学加密来保证数据安全，利用了以下密码学算法来保证了数据的安全性问题:

- 椭圆曲线数字签名:secp256k1,secp256r1
- 对称加密算法:3DES,AES
- 密钥交换:ECDH
- 密码杂凑算法:Keccak-256

12.2 2. 椭圆曲线数字签名

椭圆曲线数字签名算法（ECDSA）是使用[椭圆曲线密码（ECC）对数字签名算法（DSA）的模拟。ECDSA于1999年成为ANSI标准，并于2000年成为IEEE和NIST标准。它在1998年既已为ISO所接受，并且包含它的其他一些标准亦在ISO的考虑之中。与普通的离散对数问题（discrete logarithm problem DLP）和大数分解问题（integer factorization problem IFP）不同，椭圆曲线离散对数问题（elliptic curve discrete logarithm problem ECDLP）没有亚指数时间的解决方法。因此椭圆曲线密码的单位比特强度要高于其他公钥体制。椭圆曲线的图形如下图所示:

Hyperchain区块链使用了secp256k1曲线对平台交易进行签名验签，保证交易的正确性以及完整性。同时平台支持利用secp256r1曲线对节点间消息进行签名验证，保证节点间消息通信的完整性以及正确性。节点消息签名可插拔，可以通过配置文件开启，配置文件在namespace.toml下:

```
[encryption.check]
sign = true  #enable Signature
```

即当`sign=true`时，需要进行节点间消息的签名验证，反之则无需验证。

12.3 3.对称加密算法

Hyperchain区块链平台同时支持AES，3DES的对称加密算法，保证节点间的密文传输。

3DES又称Triple DES，是DES加密算法的一种模式，它使用3条56位的密钥对3DES数据进行三次加密。数据加密标准（DES）是美国的一种由来已久的加密标准，它使用对称密钥加密法.DES使用56位密钥和密码块的方法，而在密码块的方法中，文本被分成64位大小的文本块然后再进行加密。比起最初的DES，3DES更为安全。加密算法，其具体实现如下：设Ek()和Dk()代表DES算法的加密和解密过程，K代表DES算法使用的密钥，M代表明文，C代表密文，这样：

```
3DES加密过程为：C=Ek3(Dk2(Ek1(M)))
3DES解密过程为：M=Dk1(EK2(Dk3(C)))
```

AES又称为高级加密标准，是美国联邦政府采用的一种区块加密标准。AES算法基于排列和置换运算，AES 使用几种不同的方法来执行排列和置换运算。 AES是一个迭代的、对称密钥分组的密码，它可以使用128、192 和 256 位密钥，并且用 128 位（16字节）分组加密和解密数据。与公共密钥密码使用密钥对不同，对称密钥密码使用相同的密钥加密和解密数据。通过分组密码返回的加密数据的位数与输入数据相同。迭代加密使用一个循环结构，在该循环中重复置换和替换输入数据。

同时Hyperchain区块链平台支持对称加密算法的配置选择来对节点消息进行加解密，配置在namespace.toml下：

```
[encryption.security]
algo = "3des"    # Selective symmetric encryption algorithm (pure,3des or aes)
```

支持pure,3des以及aes三种参数：

- pure：不进行任何加密，明文传输
- 3des:进行3des加解密
- aes:进行aes加解密

12.4 4.密钥交换算法

ECDH即ECC算法和DH结合使用，用于密钥磋商。交换双方可以在不共享任何秘密的情况下协商出一个密钥。ECC是建立在基于椭圆曲线的离散对数问题上的密码体制，给定椭圆曲线上的一个点P，一个整数k，求解Q=kP很容易；给定一个点P、Q，知道Q=kP，求整数k却是一个难题。ECDH即建立在此数学难题之上。

同时Hyperchain在节点首次握手连接时便进行了密钥的交换，生成了彼此的共享密钥，此密钥即为之后节点间对称加密的密钥。

12.5 5.密码杂凑算法

密码杂凑算法，即Hash算法，Hyperchain平台利用Keccak256算法，来进行Hash计算，结算结果用于签名的消息摘要入参，同时Hash算法也可以用来地址(Address)的计算。

CHAPTER 13

JSON-RPC API

13.1 1. JSON-RPC概述

JSON-RPC是一个无状态且轻量级的远程过程调用(RPC)协议。它允许运行在基于socket、http等诸多不同消息传输环境的同一进程中，其使用JSON作为数据格式。发送一个请求对象至服务端代表一个RPC调用,一个请求对象包含下列成员:

- `jsonrpc`: 指定JSON-RPC协议版本的字符串,如果是2.0版本,则必须准确写为 “2.0”。
- `method`: 表示所要调用方法名称的字符串。以RPC开头的方法名，用英文句号(U+002E or ASCII 46)连接的为预留给RPC内部的方法名及扩展名,且不能在其他地方使用。
- `params`: 调用方法所需要的结构化参数值,该成员参数可以被省略。
- `id`: 已建立客户端的唯一标识id，该值必须包含一个字符串、数值或NULL值。如果不包含该成员则被认定为是一次通知调用。该值一般不为NULL,若为数值则应为整数。

当发起一次rpc调用时，服务端都必须回复一个JSON对象作为响应,响应对象包含下列成员:

- `jsonrpc`: 指定JSON-RPC协议版本的字符串，如果是2.0版本，则必须准确写为“2.0”。
- `result`: 该成员在成功时必须包含，当调用方法失败时必须不包含该成员。服务端中的被调用方法决定了该成员的值。
- `error`: 该成员在失败时必须包含，当没有错误引起时,不包含该成员。若引起错误,则该成员对象将包含code和message两个属性。
- `id`: 该成员必须包含。该成员值必须与请求对象中的id成员值一致。若在检查请求对象id时错误(例如参数错误或无效请求)，则该值必须为空值（NULL）。

13.2 2. 接口设计

Hyperchain接口主要由六块接口组成： 1. 交易服务，方法名前缀为 `"tx"`; 2. 合约服务，方法名前缀为 `"contract"`; 3. 区块服务，方法名前缀为 `"block"`; 4. 消息订阅服务，方法名前缀为 `"sub"`; 5. 节点服务，方法名前缀为 `"node"`; 6. 证书服务，方法名前缀为 `"cert"`;

接口设计基于JSON-RPC 2.0规范。所有HTTP请求均为POST请求，请求的参数包括:

- `jsonrpc`: 指定JSON-RPC协议版本的字符串,如果是2.0版本，则必须准确写为 “2.0”。

- namespace: 表示该条请求发送给哪个分区去处理。
- method: 表示所要调用方法名称的字符串，格式为：(服务前缀)_(方法名)。
- params: 调用方法所需要的结构化参数值，该成员参数可以被省略。
- id: 已建立客户端的唯一标识id，该值必须包含一个字符串、数值。

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"block_latestBlock","namespace":"global
↪","params":[],"id":1}' localhost:8081
```

返回值格式为：

- jsonrpc: 指定JSON-RPC协议版本的字符串，如果是2.0版本，则必须准确写为 “2.0”。
- namespace: 表示该条请求所属分区。
- code: 状态码。若成功，则为0，其他状态码详见表5-1。
- message: 错误信息。若成功，则为“SUCCESS”，否则为错误详细信息。
- result: 被调用方法成功执行返回的结果。
- id: 该值应与请求对象中的id值保持一致。

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.4",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
    "transactions": [
      {
        "version": "1.4",
        "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae
↪",
        "blockNumber": "0x3",
        "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
        "txIndex": "0x0",
        "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
        "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
        "amount": "0x0",
        "timestamp": 1481778652973000000,
        "nonce": 3573634504790373,
        "extra": "",
        "executeTime": "0x2",
        "payload":
↪"0x81053a70000000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪"
      }
    ]
```

```
    }
}
```

比如，接口调用成功的话，返回的字段有：jsonrpc、namespace、id、code、message、result，且code值为0，message值为“SUCCESS”，用户可通过这两个字段的值来判断接口调用是否成功，若调用失败，则code为非0值，message为错误信息。code值的定义如下：

code	含义
0	请求成功
-32700	服务端接收到无效的json。该错误发送于服务器尝试解析json文本
-32600	无效的请求（比如非法的JSON格式）
-32601	方法不存在或者无效
-32602	无效的方法参数
-32603	JSON-RPC内部错误
-32000	Hyperchain内部错误或者空指针或者节点未安装solidity环境
-32001	查询的数据不存在
-32002	余额不足
-32003	签名非法
-32004	合约部署出错
-32005	合约调用出错
-32006	系统繁忙
-32007	交易重复
-32008	合约操作权限不够
-32009	(合约)账户不存在
-32010	namespace不存在
-32011	账本上无区块产生，查询最新区块的时候可能抛出该错误
-32012	订阅不存在 (预留状态码)
-32013	数据归档、快照相关错误
-32098	请求未带cert或者错误cert导致认证失败
-32099	请求tcert失败

13.3 3. 接口概览

13.3.1 Transaction

- *tx_getTransactions*
- *tx_getDiscardTransactions*
- *tx_getTransactionByHash*
- *tx_getTransactionByBlockHashAndIndex*
- *tx_getTransactionByBlockNumberAndIndex*
- *tx_getTransactionsCount*
- *tx_getTxAvgTimeByBlockNumber*
- *tx_getTransactionReceipt*
- *tx_getBlockTransactionCountByHash*
- *tx_getBlockTransactionCountByNumber*
- *tx_getSignHash*
- *tx_getTransactionsByTime*
- *tx_getDiscardTransactionsByTime*
- *tx_getBatchTransactions*

- *tx_getBatchReceipt*

13.3.2 Contract

- *contract_compileContract*
- *contract_deployContract*
- *contract_invokeContract*
- *contract_getCode*
- *contract_getContractCountByAddr*
- *contract_maintainContract*
- *contract_getStatus*
- *contract_getCreator*
- *contract_getCreateTime*
- *contract_getDeployedList*

13.3.3 Block

- *block_latestBlock*
- *block_getBlocks*
- *block_getBlockByHash*
- *block_getBlockByNumber*
- *block_getAvgGenerateTimeByBlockNumber*
- *block_getBlocksByTime*
- *block_getGenesisBlock*
- *block_getChainHeight*
- *block_getBatchBlocksByHash*
- *block_getBatchBlocksByNumber*

13.3.4 Subscription

- *sub_newBlockSubscription*
- *sub_newEventSubscription*
- *sub_getLogs*
- *sub_newSystemStatusSubscription*
- *sub_getSubscriptionChanges*
- *sub_unSubscription*

13.3.5 Node

- *node_getNodes*
- *node_getNodeHash*
- *node_deleteVP*
- *node_deleteNVP*

13.3.6 Certificate

- *cert_getTCert*

13.4 4. 接口描述

13.4.1 tx_getTransactions

查询指定区块区间的所有交易。

Parameters

1. `<Object>`

- `from`: `<blockNumber>` - 起始区块号。
- `to`: `<blockNumber>` - 终止区块号。

`<blockNumber>`可以是十进制整数或者进制字符串，可以是"`latest`"字符串表示最新的区块。`from`必须小于等于`to`，否则会返回error。

Returns

1. `[<Transaction>]` - Transaction对象字段如下：

- `version`: `<string>` - 平台版本号。
- `hash`: `<string>` - 32字节的十六进制字符串，交易哈希值。
- `blockNumber`: `<string>` - 十六进制，交易所在区块的高度。
- `blockHash`: `<string>` - 32字节的十六进制字符串，交易所在区块的哈希。
- `txIndex`: `<string>` - 十六进制，交易在区块中的偏移量。
- `from`: `<string>` - 20字节的十六进制字符串，交易发送方的地址。
- `to`: `<string>` - 20字节的十六进制字符串，交易接收方的地址。
- `amount`: `<string>` - 转账金额。
- `timestamp`: `<number>` - 交易发生的unix时间戳（单位ns）。
- `nonce`: `<number>` - 16位随机数。
- `extra`: `<string>` - 交易的额外信息。
- `executeTime`: `<string>` - 交易的处理时间（单位ms）。
- `payload`: `<string>` - 部署合约、调用合约、升级合约的时候才有这个值，可以通过这个值追溯到合约调用的方法以及调用传入的参数。

Example1: 正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactions", "params": [{"from": 1, "to": 2}], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "result": [
    {
      "version": "1.0",
      "hash": "0x88d5b325dc9042ff92a9fa26ed8c943719bb049ac7022abd09bb85da36f531e4",
      "blockNumber": "0x2",
      "blockHash":
↪"0xc6418753c28ad6d744cb4bbe689521696ba65ad010ce24056b6f8def9fc5cdd5",
      "txIndex": "0x0",
      "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1486994814684628715,
      "nonce": 7948317390228704,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x60606040526000805463ffffffff19168155609e908190601e90396000f3606060405260e060020a60003504633a
↪"
    },
    {
      "version": "1.0",
      "hash": "0xf7149a8349f1853d8d713a15935e5059e6f55c2827f0c88f8414dd0402d6760b",
      "blockNumber": "0x1",
      "blockHash":
↪"0x4bab3f9297e737eb197d666a2f08219f94460ace08a8e1ecad87e6e52183bcd5",
      "txIndex": "0x0",
      "from": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1486994799163184948,
      "nonce": 2099818402815731,
      "extra": "",
      "executeTime": "0x7",
      "payload":
↪"0x60606040526000805463ffffffff19168155609e908190601e90396000f3606060405260e060020a60003504633a
↪"
    }
  ]
}
```

Example2: 区块不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactions", "params": [{"from": 1, "to": 2}], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
```

```
  "id": 71,
  "code": -32602,
  "message": "block number 1 is out of range, and now latest block number is 0"
}
```

13.4.2 tx_getDiscardTransactions

查询所有非法交易。

Parameters

无

Returns

1. `[<Transaction>]` - Transaction对象字段如下：

- `version`: `<string>` - 平台版本号。
- `hash`: `<string>` - 32字节的十六进制字符串，交易哈希值。串，交易所在区块的哈希。
- `from`: `<string>` - 20字节的十六进制字符串，交易发送方的地址。
- `to`: `<string>` - 20字节的十六进制字符串，交易接收方的地址。
- `amount`: `<string>` - 转账金额。
- `timestamp`: `<number>` - 交易发生的unix时间戳（单位ns）。
- `nonce`: `<number>` - 16位随机数。
- `extra`: `<string>` - 交易的额外信息。
- `payload`: `<string>` - 部署合约、调用合约、升级合约的时候才有这个值，可以通过这个值追溯到合约调用的方法以及调用传入的参数。
- `invalid`: `<boolean>` - 交易是否不合法。
- `invalidMsg`: `<string>` - 交易的不合法信息。

不合法的交易`invalid`值为true， `invalidMsg`可能为：

- **DEPLOY_CONTRACT_FAILED** - 合约部署失败；
- **INVOKE_CONTRACT_FAILED** - 合约方法调用失败；
- **SIGFAILED** - 签名非法；
- **OUTOFBALANCE** - 余额不足；
- **INVALID_PERMISSION** - 合约操作权限不够；

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getDiscardTransactions", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
```

```
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "",
      "hash": "0x100ff931204d149f88c0778f6e7b8d4b11ba3c8c720f0cc3e204b46999954ed4",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1482405417011000000,
      "nonce": 6848885244669098,
"extra": "",
      "payload": "0x60606040526002600055600256",
      "invalid": true,
      "invalidMsg": "DEPLOY_CONTRACT_FAILED"
}
  ]
}
```

Example2：若没有非法交易

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getDiscardTransactions", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": -32001,
  "message": "Not found discard transactions "
}
```

13.4.3 tx_getTransactionByHash

根据交易哈希查询交易详情。

Parameters

1. <string> - 32字节的十六进制字符串，交易的哈希值。

Returns

1. <Transaction> - Transaction对象字段如下：

- version: <string> - 平台版本号。
- hash: <string> - 32字节的十六进制字符串，交易哈希值。
- blockNumber: <string> - 十六进制，交易所在区块的高度。
- blockHash: <string> - 32字节的十六进制字符串，交易所在区块的哈希。
- txIndex: <string> - 十六进制，交易在区块中的偏移量。
- from: <string> - 20字节的十六进制字符串，交易发送方的地址。
- to: <string> - 20字节的十六进制字符串，交易接收方的地址。

- amount: <string> - 转账金额。
- timestamp: <number> - 交易发生的unix时间戳（单位ns）。
- nonce: <number> - 16位随机数。
- extra: <string> - 交易的额外信息。
- executeTime: <string> - 交易的处理时间（单位ms）。
- payload: <string> - 部署合约、调用合约、升级合约的时候才有这个值，可以通过这个值追溯到合约调用的方法以及调用传入的参数。
- invalid: <boolean> - 交易是否不合法。
- invalidMsg: <string> - 交易的不合法信息。

不合法的交易invalid值为true， invalidMsg可能为：

- **OUTOFBALANCE** - 余额不足，对应code是-32002；
- **SIGFAILED** - 签名非法，对应code是-32003；
- **DEPLOY_CONTRACT_FAILED** - 合约部署失败，对应code是-32004；
- **INVOKE_CONTRACT_FAILED** - 合约方法调用失败，对应code是-32005；
- **INVALID_PERMISSION** - 合约操作权限不够，对应code是-32008；

Example1：查询合法的交易

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getTransactionByHash", "params":[
↪"0xe652e25e617c5f193b240c0d8ff1941a8cfb1d15434eb3830892b7a8389730aa"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe652e25e617c5f193b240c0d8ff1941a8cfb1d15434eb3830892b7a8389730aa",
    "blockNumber": "0x4",
    "blockHash":
↪"0x6ea0c80c1532c273c124511e364fc0a9225e0d129e53249f8e26752ee7d7d989",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x0000000000000000000000000000000000000000",
    "amount": "0x0",
    "timestamp": 1482405601747000000,
    "nonce": 6788653222523786,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6060604052600080556002d80601460003960000f3606060405260e060020a6000350463be1c766b8114601c575b60
↪"
  }
}
```

Example2：查询非法的交易

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getTransactionByHash","params":[
↪"0x1f6dc4c744ce5e8a39e6a19f19dc27c99d7efd8e38061e80550bf5e7ab1060e1"],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.3",
    "hash": "0x1f6dc4c744ce5e8a39e6a19f19dc27c99d7efd8e38061e80550bf5e7ab1060e1",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x0000000000000000000000000000000000000000",
    "amount": "0x0",
    "timestamp": 1509448178302000000,
    "nonce": 1166705097783423,
    "extra": "",
    "payload":
↪"0x6060604052600080553415601257600080fd5b5b60026000908155580fd5b5b5b60918061002d6000396000f30060
↪",
    "invalid": true,
   "invalidMsg": "DEPLOY_CONTRACT_FAILED"
  }
}
```

Example3：查询的交易不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getTransactionByHash", "params":["␣
↪0x0e707231fd779779ce25a06f51aec60faed8bf6907e6d74fb11a3fd585831a7e"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": -32001,
  "message": "Not found transaction␣
↪0x0e707231fd779779ce25a06f51aec60faed8bf6907e6d74fb11a3fd585831a7e"
}
```

13.4.4 tx_getTransactionByBlockHashAndIndex

根据区块哈希和交易偏移量查询交易。

Parameters

1. <string> - 32字节的十六进制字符串，区块的哈希值。
2. <number> - 交易在区块中的偏移量，可以是十进制整数或进制字符串。

Returns

1. <Transaction> - Transaction对象字段见 合法交易.

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsByBlockHashAndIndex","params": [
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9", 0], "id":␣
↪71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe81d39df11779c7f83e6073cc659c7ee85708c135b6557d318e765b9f938c02f",
    "blockNumber": "0x2",
    "blockHash":
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "amount": "0x0",
    "timestamp": 1481767474717000000,
    "nonce": 8054165127693853,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000007b0000000000000000000000
↪"
  }
}
```

Example2：查询的区块不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsByBlockHashAndIndex","params": [
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9", 0], "id":␣
↪71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32001,
  "message": "Not found block␣
↪0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9"
}
```

13.4.5 tx_getTransactionByBlockNumberAndIndex

根据区块号和交易偏移量查询交易。

Parameters

1. <blockNumber> - 区块号，可以是十进制整数、进制字符串或“latest”字符串表示最新的区块。
2. <number> - 交易在区块中的偏移量，可以是十进制整数或进制字符串。

Returns

1. <Transaction> - Transaction对象字段见 合法交易.

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":   "tx_
↪getTransactionByBlockNumberAndIndex", "params": [2,0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "hash": "0xe81d39df11779c7f83e6073cc659c7ee85708c135b6557d318e765b9f938c02f",
    "blockNumber": "0x2",
    "blockHash":
↪"0xd198976fa8b4ca2de6b1b137552b84dc08b7cdcbebbf9388add88f4710fd2cf9",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "amount": "0x0",
    "timestamp": 1481767474717000000,
    "nonce": 8054165127693853,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000007b0000000000000000000000
↪"
    }
}
```

Example2：请求的区块不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":   "tx_
↪getTransactionsByBlockNumberAndIndex", "params": [2,0], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
```

```
  "id": 71,
  "code": -32602,
  "message": "block number 2 is out of range, and now latest block number is 0"
}
```

13.4.6 tx_getTransactionsCount

查询当前链上交易量。

Parameters

无

Returns

1. <Object>

- count: <string> - 交易数量，十六进制字符串表示。
- timestamp: <number> - 响应时间戳（单位ns）。

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionsCount", "params": [], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 71,
  "code": 0,
  "message": "SUCCESS":,
  "result": {
    "count": "0x9",
    "timestamp": 1480069870678091862
   }
}
```

13.4.7 tx_getTxAvgTimeByBlockNumber

根据指定的区块区间计算出每笔交易的平均处理时间。

Parameters

1. <Object>

- from: <blockNumber> - 起始区块号。
- to: <blockNumber> - 终止区块号。

blockNumber 可以是十进制整数或者进制字符串，可以是“latest”字符串表示最新的区块。 from 必须小于等于 to ， 否则会返回error。如果 from 和 to 的值一样，则表示计算的是当前指定区块交易的平均处理时间。

Returns

1. <string> - 十六进制字符串，表示交易的平均处理时间（单位ms）。

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTxAvgTimeByBlockNumber", "params": [{"from":10, "to":19}], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xa9"
}
```

13.4.8 tx_getTransactionReceipt

根据交易哈希返回交易回执信息。

Parameters

1. <string> - 32字节的十六进制字符串，交易的哈希值。

Returns

1. <Receipt> - Receipt对象字段如下：
   - version: <string> - 平台版本号。
   - txHash: <string> - 交易哈希。
   - vmType: <string> - 该笔交易执行引擎类型，EVM或JVM。
   - contractAddress: <string> - 合约地址。
   - gasUsed: <number> - 该笔交易所耗费的gas。
   - ret: <string> - 合约编译后的字节码或合约执行的结果。
   - log: [<Log>] - Log对象数组，表示合约中的event log信息。Log对象如下：
     - address: <string> - 产生事件日志的合约地址。
     - topics: [<string>] - 一系列的topic，第一个topic是event的唯一标识。
     - data: <string> - 日志信息。
     - blockNumber: <number> - 所属区块的区块号。
     - blockHash: <string> - 所属区块的区块哈希。
     - txHash: <string> - 所属交易的交易哈希。
     - txIndex: <number> - 所属交易在当前区块交易列表中的偏移量。
     - index: <number> - 该日志在本条交易产生的所有日志中的偏移量。

如果该笔交易还没被确认，则返回的错误码为-32001的error，如果该笔交易处理过程中发生错误，则错误可能是：

- **OUTOFBALANCE** - 余额不足，对应code是-32002；
- **SIGFAILED** - 签名非法，对应code是-32003；
- **DEPLOY_CONTRACT_FAILED** - 合约部署失败，对应code是-32004；
- **INVOKE_CONTRACT_FAILED** - 合约方法调用失败，对应code是-32005；
- **INVALID_PERMISSION** - 合约操作权限不够，对应code是-32008；

Example1：交易未被确认

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x0e0758305cde33c53f8c2b852e75bc9b670c14c547dd785d93cb48f661a2b36a "],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": -32001,
  "message": "Not found receipt by␣
↪0x0e0758305cde33c53f8c2b852e75bc9b670c14c547dd785d93cb48f661a2b36a"
}
```

Example2：合约部署出错

在这个例子中我们使用以下合约来重现这个情况：

```
contract TestContractor{
    int length = 0;

    modifier justForTest(){
        length = 2;
        throw;
        _;
    }
    function TestContractor()justForTest{
    }

    function getLength() returns(int){
        return length;
    }
}
```

我们将该合约编译后返回的bin作为*contract_deployContract*方法中参数payload的值，那么部署合约请求如下：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_deployContract", "params":[{
"from":"17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"nonce":7021040367249265,
"payload":
↪"0x6060604052600060005534600057 5b6002600556000565b5b5b603f806100266000396000f3606060405260e060
↪",
"timestamp":1487042279126000000,
```

```
"signature":
↪"0xfc1cb1986dd4ee4a5f8d8238e2f7bac1866aad235d587eb641d76270bf686418310ab7d42dc0f2575aa858a88ae7
↪"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x33aef7e6bad2ae27c23a8ab44f56aef87042f1f0b02e1b0ee5e8a304705292a6"
}
```

接着，根据返回的hash查找这条记录的receipt，会发现返回了合约部署失败的error：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x33aef7e6bad2ae27c23a8ab44f56aef87042f1f0b02e1b0ee5e8a304705292a6"],"id":1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":-32004,
  "message":"DEPLOY_CONTRACT_FAILED"
}
```

Example3：合约方法调用出错

在这个例子中我们使用以下合约来重现这个情况：

```
contract TestContractor{
    int length = 0;

    modifier justForTest(){
        length = 2;
        throw;
        _;
    }
    function TestContractor(){
    }

    function getLength()justForTest returns(int){
        return length;
    }
}
```

调用合约`getLength()`方法的请求如下：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params": [{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
"to":"0xaeccd2fd1118334402c5de1cb014a9c192c498df",
"timestamp":1487042517534000000,
"nonce":2472630987523856,
"payload":"0xbe1c766b",
"signature":
↪"0x8c56f025610dd9cb3f4ac346d35978639a536505527b7593d87f3b45c3532863728099 5ed32f6a6809069da91574
↪"}],"id": 1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x5233d18f46e9c1ed49dbdeb4273c1c1e0eb176efcedf6edb6d9fa59d33d02fee "
}
```

接着，根据返回的hash查找这条记录的receipt，会发现返回了方法调用失败的error：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x5233d18f46e9c1ed49dbdeb4273c1c1e0eb176efcedf6edb6d9fa59d33d02fee"],"id":1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":-32005,
  "message":"INVOKE_CONTRACT_FAILED"
}
```

Example4：签名非法

我们将Example3合约方法调用失败例子的参数稍微修改一下，把`from`的最后一个字母“c”改为“0”，那么调用合约请求如下：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params": [{
"from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bff0",
"to":"0xaeccd2fd1118334402c5de1cb014a9c192c498df",
"timestamp":1481872888621000000,
"nonce":9206467481004664,
"payload":"0xbe1c766b",
"signature":
↪"0x57dfa7f2c2d8c762c9c0e5ef7b1c4dda84b584f36799ab751891c8dc553862145f64d1991441c9460481af4e4231c
↪"}],"id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "id": "SUCCESS",
  "result":"0x621d09cd9d5e9027d9b82c5e1fd911ac31297775dbb0c4dab6c6fcd64310fe23"
}
```

接着，根据返回的hash查找这条记录的receipt，会发现返回了签名非法的error：

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionReceipt","params":[
↪"0x621d09cd9d5e9027d9b82c5e1fd911ac31297775dbb0c4dab6c6fcd64310fe23"],"id":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": -32003,
  "message": " SIGFAILED "
}
```

Example5

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "tx_
↪getTransactionReceipt", "params":[
↪"0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
      "version": "1.3",
      "txHash": "0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059
↪",
      "vmType": "EVM",
      "contractAddress": "0x0000000000000000000000000000000000000000",
      "ret": "0x0",
      "log": [
          {
              "address": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
              "topics": [

↪"0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"
              ],
              "data":
↪"0000000000000000000000000000000000000000000000000000000000000064",
              "blockNumber": 2,
              "blockHash":
↪"0x0c14a89b9611f7f268f26d4ce552de966bebba4aab6aaea988022f3b6817f61b",
              "txHash":
↪"0x70376053e11bc753b8cc778e2fbb662718671712e1744980ba1110dd1118c059",
              "txIndex": 0,
              "index": 0
          }
      ]
  }
}
```

13.4.9 tx_getBlockTransactionCountByHash

根据区块哈希查询区块交易数量。

Parameters

1. `<string>` - 32字节的十六进制字符串，区块的哈希值。

Returns

1. <string> - 十六进制字符串，交易数量。

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getBlockTransactionCountByHash", "params":  [
↪"0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xaf5"
}
```

Example2：查询区块不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":"tx_
↪getBlockTransactionCountByHash", "params":  [
↪"0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32001,
  "message":"Not found block␣
↪0x7a87bd1fb51a86763e9791eab1d5ecca7f004bea1cfcc426113b4625d267f699"
}
```

13.4.10 tx_getBlockTransactionCountByNumber

根据区块号查询区块交易数量。

Parameters

1. <blcokNumber> - 区块号，可以是十进制整数、进制字符串或“latest”字符串来表示最新区块。

Returns

1. <string> - 十六进制字符串，交易数量。

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":" tx_
↪getBlockTransactionCountByNumber", "params":  ["0x2"], "id": 71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0xaf5"
}
```

Example2：查询的区块不存在

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method":" tx_
↪getBlockTransactionCountByNumber", "params":  ["0x2"], "id": 71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 71,
  "code": -32602,
  "message": "block number 0x2 is out of range, and now latest block number is 0"
}
```

13.4.11 tx_getSignHash

获取用于签名算法的哈希。

Parameters

1. <Object>

- from: <string> - 20字节的十六进制字符串，交易发送方的地址。
- to: <string> - [可选] 20字节的十六进制字符串，交易接收方的地址（普通账户或合约地址）。若是部署合约，则不需要这个参数。
- nonce: <number> - 16位随机数。
- extra: <string> - [可选] 交易的额外信息。
- value或payload: <string> - value表示转账金额，payload表示合约操作对应字节编码。
- timestamp: <number> - 交易发生时间戳（单位ns）。

注解： 如果是部署合约的交易，则不要传to。若为普通转账，则传value，表示转账金额。若是部署合约、调用合约或升级合约的交易，则传payload，含义详见部署合约、调用合约或升级合约的接口。

Returns

1. <string> - 十六进制字符串，签名哈希。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getSignHash", "params":[{
    "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "nonce":5373500328656597,
    "payload":
↪"0x60606040526000805463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633a
↪",
    "timestamp":1487771157166000000 }],"id":"1"}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x2e6a644a4ca6a9daba4444995dc0dda039208e642df11db35438d18e7c3b13c3"
}
```

13.4.12 tx_getTransactionsByTime

查询指定时间区间内的所有合法交易。

Parameters

1. <Object>

- startTime: <number> - 起始时间戳（单位ns）。
- endTime: <number> - 结束时间戳（单位ns）。

Returns

1. [<Transaction>] - Transaction对象字段见 合法交易.

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"tx_
↪getTransactionsByTime","params":[{"startTime":1, "endTime":1581776001230590326}],
↪"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [{
      "version": "1.0",
      "hash": "0xbd441c7234e3b83a05c89ed5d548c3d1877306975e271a08e7354d74e45431bc",
      "blockNumber": "0x1",
      "blockHash":
↪"0xa6a4b2df16c7bdeb578aa7de7b05f9b54d96202bdc8414196741842834156ebd",
```

```
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0x0000000000000000000000000000000000000000",
      "amount": "0x0",
      "timestamp": 1481767468349000000,
      "nonce": 1775845467490815,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x6060604052346100005b6101e18061001860003960000f3606060405260e060020a60003504636fd7cc16811461
↪"
  }]
}
```

Example2：查询数据不存在

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getTransactionsByTime","params
↪":[{"startTime":1681776001230590326, "endTime":1681776001230590326}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": []
}
```

13.4.13 tx_getDiscardTransactionsByTime

查询指定时间区间内的所有非法交易。

Parameters

1. <Object>

- startTime: <number> - 起始时间戳（单位ns）。
- endTime: <number> - 结束时间戳（单位ns）。

Returns

1. [<Transaction>] - Transaction对象字段见 非法交易.

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" tx_
↪getDiscardTransactionsByTime","params":[{"startTime":1, "endTime
↪":1581776001230590326}],"id":1}'

# Result
{
  "jsonrpc": "2.0",
```

```
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
      {
          "version": "1.3",
          "hash":
↪"0x4e468969d94b92622e385246779d05981ef43869b17c8afedc7e6b5b138ae807",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd",
          "amount": "0x1",
          "timestamp": 1501586411342000000,
          "nonce": 4563214039387098,
          "extra": "",
          "payload": "0x0",
          "invalid": true,
          "invalidMsg": "OUTOFBALANCE"
      }
  ]
}
```

13.4.14 tx_getBatchTransactions

根据交易哈希批量查询交易。

Parameters

1. <Object>

- hashs: [<string>] - 交易哈希数组，哈希值为32字节的十六进制字符串。

Returns

1. [<Transaction>] - Transaction对象字段见 合法交易.

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0","method":"tx_getBatchTransactions","params":[
↪{
    "hashes":["0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602"]
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
      {
          "version": "1.3",
          "hash":
↪"0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
```

```
          "blockNumber": "0x2",
          "blockHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
          "amount": "0x0",
          "timestamp": 1509440823410000000,
          "nonce": 8291834415403909,
          "extra": "",
          "executeTime": "0x6",
          "payload": "0x0a9ae69d"
      },
      {
          "version": "1.3",
          "hash":
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602",
          "blockNumber": "0x1",
          "blockHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0x0000000000000000000000000000000000000000",
          "amount": "0x0",
          "timestamp": 1509440820498000000,
          "nonce": 5098902950712745,
          "extra": "",
          "executeTime": "0x11",
          "payload":
↪"0x60606040523415610000f57600080fd5b604051604080610160833981016040528080519190602001805191505050b
↪"
      }
  ]
}
```

13.4.15 tx_getBatchReceipt

根据交易哈希批量查询交易回执。

Parameters

1. <Object>

- hashs: [<string>] - 交易哈希数组,，哈希值为32字节的十六进制字符串。

Returns

1. [<Receipt>] - Receipt对象字段见 *Receipt*.

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"tx_getBatchReceipt","params":[{
    "hashes":["0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602"]
}],"id":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "txHash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4
↪",
      "vmType": "EVM",
      "contractAddress": "0x0000000000000000000000000000000000000000",
      "ret": "0x0000000000000000000000000000000000000000000000000000000000000005",
      "log": []
    },
    {
      "version": "1.3",
      "txHash": "0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602
↪",
      "vmType": "EVM",
      "contractAddress": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "ret":
↪"0x606060405263ffffffff7c0100000000000000000000000000000000000000000000000000000000600035041663
↪",
      "log": []
    }
  ]
}
```

13.4.16 contract_compileContract

编译智能合约。

Parameters

1. <string> - 合约源码。

Returns

1. <Object>

- abi: [<string>] - 合约源码对应的abi。
- bin: [<string>] - 合约编译而成的字节码。
- types: [<string>] - 对应合约的名称。

若源码中有多个合约，则bin为顶层合约的字节码。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"contract_
↪compileContract", "params":["contract Accumulator{    uint32 sum = 0;    function
↪increment(){          sum = sum + 1;      }       function getSum() returns(uint32)
↪{          return sum;      }    function add(uint32 num1,uint32 num2) {
↪sum = sum+num1+num2;      } }"],"id":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "abi": [
          "[{\"constant\":false,\"inputs\":[{\"name\":\"num1\",\"type\":\"uint32\"}
↪,{\"name\":\"num2\",\"type\":\"uint32\"}],\"name\":\"add\",\"outputs\":[],\
↪"payable\":false,\"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\
↪":\"getSum\",\"outputs\":[{\"name\":\"\",\"type\":\"uint32\"}],\"payable\":false,
↪\"type\":\"function\"},{\"constant\":false,\"inputs\":[],\"name\":\"increment\",\
↪"outputs\":[],\"payable\":false,\"type\":\"function\"}]"
          ],
    "bin": [

↪"0x6060604052600080546463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633a
↪"
          ],
    "types": [
          "Accumulator"
          ]
  }
}
```

13.4.17 contract_deployContract

部署合约。

Parameters

1. <Object>

- from: <string> - 20字节的十六进制字符串，交易发送方的地址。
- nonce: <number> - 16位随机数，该值必须为十进制整数。
- extra: <string> -[可选] 交易的额外信息。
- timestamp: <number> - 交易发生时间戳（单位ns）。
- payload: <string> - 合约编码。如果是**solidity**合约，则该值为contract_complieContract方法返回的bin以及构造函数参数的拼接。如果是**java**合约，该值为class文件和配置文件压缩后的字节流。
- signature: <string> - 交易签名。
- type: <string> - [可选] 指定合约执行引擎，默认为**"evm"**。如果合约代码由java语言编写，则需要设置该值为**"jvm"**。

注解： 若合约构造函数需要传参，则payload为编译合约返回的bin与构造函数参数编码的字符串拼接。

Returns

1. <string> - 32字节的十六进制字符串，交易的哈希值。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global",  "method":
↪"contract_deployContract", "params":[{
    "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "nonce":5373500328656597,
    "payload":
↪"0x6060604052600805463ffffffff1916815560ae908190601e90396000f3606060405260e060020a60003504633a
↪",
    "signature":
↪"0x388ad7cb71b1281eb5a0746fa8fe6fda006bd28571cbe69947ff0115ff8f3cd00bdf2f45748e0068e498034289992
↪",
    "timestamp":1487771157166000000
}],"id":"1"}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x406f89cb205e136411fd7f5befbf8383bbfdec5f6e8bcfe50b16dcff037d1d8a"
}
```

13.4.18 contract_invokeContract

调用合约。

Parameters

1. <Object>

- from: <string> - 20字节的十六进制字符串，交易发送方的地址。
- to: <string> - 20字节的十六进制字符串，合约地址。
- nonce: <number> - 16位随机数，该值必须为十进制整数。
- extra: <string> -[可选] 交易的额外信息。
- timestamp: <number> - 交易发生时间戳（单位ns）。
- payload: <string> - 该值为方法名和方法参数经过编码后的input字节码。
- signature: <string> - 交易签名。
- simulate: <bool> - [可选] 默认为false。true表示交易不走共识，false表示走共识。
- type: <string> - [可选] 指定合约执行引擎，默认为**"evm"**。如果合约代码由java语言编写，则需要设置该值为**"jvm"**。

注解： 说明：to合约地址需要在部署完合约以后，调用 *tx_getTransactionReceipt* 方法来获取。

Returns

1. <string> - 32字节的十六进制字符串，交易的哈希值。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_invokeContract", "params":[{
    "from":"0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "nonce":5019420501875693,
    "payload":
↪"0x3ad14af3000000000000000000000000000000000000000000000000000000000000000100000000000000000000000
↪",
    "signature":
↪"0xde467ec4c0bd9033bdc3b6faa43a8d3c5dcf393ed9f34ec1c1310b0859a0ecba15c5be4480a9ad2aaaea8416324c
↪",
    "simulate":false,
    "timestamp":1487773188814000000,
    "to":"0x313bbf563991dc4c1be9d98a058a26108adfcf81"
}],"id":"1"}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

13.4.19 contract_getCode

获取合约的字节编码。

Parameters

1. <string> - 合约地址。

Returns

1. <string> - 十六进制字节码。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"contract_
↪getCode","params": ["0xaeccd2fd1118334402c5de1cb014a9c192c498df"],"id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":
↪"0x6060604052361561005760e060020a600035046301000dd7811461005b5780638e739461146100e55780638f24
↪"
}
```

13.4.20 contract_getContractCountByAddr

获取指定账户部署的合约量。

Parameters

1. <string> - 20字节的十六进制字符串，账户地址。

Returns

1. <string> - 合约数量。

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "contract_
↪getContractCountByAddr", "params": ["0xa94f5374fce5edbc8e2a8697c15331677e6ebf0b
↪"], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x3"
}
```

13.4.21 contract_maintainContract

升级合约、冻结合约、解冻合约。 只有合约的部署者才拥有升级合约、冻结合约、解冻合约的权限。

Parameters

1. <Object>

- from: <string> - 20字节的十六进制字符串，交易发送方的地址。
- to: <string> - 20字节的十六进制字符串，合约地址。
- nonce: <number> - 16位随机数，该值必须为十进制整数。
- extra: <string> -[可选] 交易的额外信息。
- timestamp: <number> - 交易发生时间戳（单位ns）。
- payload: <string> - [可选] 编译后的新合约字节码。**升级合约才需要这个字段。**
- signature: <string> - 交易签名。
- type: <string> - 指定合约执行引擎，默认为**"evm"**。如果合约代码由java语言编写，则需要设置该值为**"jvm"**。
- opcode: 值为1表示**升级合约**，值为2表示**冻结合约**、值为3表示**解冻合约**。

注解： 说明：to合约地址需要在部署完合约以后，调用 *tx_getTransactionReceipt* 方法来获取。

Returns

1. <string> - 32字节的十六进制字符串，交易的哈希值。

Example1：升级合约

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
    "from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "timestamp":1481767474717000000,
    "nonce": 8054165127693853,
    "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000007b0000000000000000000000
↪",
    "signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fc
↪",
    "opcode": 1}],
"id": 1}'


# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

Example2：冻结合约

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
    "from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "timestamp":1481767474717000000,
    "nonce": 8054165127693853,
    "signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fc
↪",
    "opcode": 2}],
"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

Example3：解冻合约

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_maintainContract","params":[{
    "from": "17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to":"0x3a3cae27d1b9fa931458b5b2a5247c5d67c75d61",
    "timestamp":1481767474717000000,
    "nonce": 8054165127693853,
    "signature":
↪"0x19c0655d05b9c24f5567846528b81a25c48458a05f69f05cf8d6c46894b9f12a02af471031ba11f155e41adf42fc
↪",
    "opcode": 3}],
"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":"0xd7a07fbc8ea43ace5c36c14b375ea1e1bc216366b09a6a3b08ed098995c08fde"
}
```

13.4.22 contract_getStatus

查询合约状态。

Parameters

1. <string> - 20字节的十六进制字符串，合约地址。

Returns

1. <string> - 合约状态。normal表示正常状态，frozen表示冻结状态，non-contract表示非合约，即为普通转账的交易。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getStatus","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"],"id
↪": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":" normal"
}
```

13.4.23 contract_getCreator

查询合约部署者。

Parameters

1. <string> - 20字节的十六进制字符串，合约地址。

Returns

1. <string> - 20字节的十六进制字符串，合约部署者地址。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getCreator","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"],
↪"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":" 0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd "
}
```

13.4.24 contract_getCreateTime

查询合约部署时间。

Parameters

1. <string> - 20字节的十六进制字符串，合约地址。

Returns

1. <string> - 合约部署的日期时间。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getCreateTime","params": ["0xbbe2b6412ccf633222374de8958f2acc76cda9c9
↪"],"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
```

```
  "code":0,
  "message":"SUCCESS",
  "result":"2017-04-07 12:37:06.152111325 +0800 CST"
}
```

13.4.25 contract_getDeployedList

查询已部署的合约地址列表。

Parameters

1. <string> - 20字节的十六进制字符串，账户地址。

Returns

1. [<string>] - 已部署的所有合约地址。

Example

```
# Request
curl localhost:8081 --data '{"jsonrpc":"2.0", "namespace":"global", "method":
↪"contract_getDeployedList","params": ["0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
↪"],"id": 1}'

# Response
{
  "jsonrpc":"2.0",
  "namespace":"global",
  "id":1,
  "code":0,
  "message":"SUCCESS",
  "result":["0xbbe2b6412ccf633222374de8958f2acc76cda9c9"]
}
```

13.4.26 block_latestBlock

获取最新区块。

Parameters

无

Returns

1. <Block> - Block对象字段如下：

- version: <string> - 平台版本号。
- number: <string> - 区块的高度。
- hash: <string> - 区块的哈希值，32字节的十六进制字符串。
- parentHash: <string> - 父区块哈希值，32字节的十六进制字符串。

- writeTime: <number> - 区块生成的unix时间戳。
- avgTime: <string> - 当前区块中，交易的平均处理时间（单位ms）。
- txCounts: <string> - 当前区块中打包的交易数量。
- merkleRoot: <string> - Merkle树的根哈希。
- transactions: [<Transaction>] - 区块中的交易列表。

Example1：正常的请求

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪latestBlock","params":[],"id":71}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a70000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"
      }
    ]
  }
}
```

Example2：如果链上一个区块都没有

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪latestBlock","params":[],"id":71}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": -32602,
  "message": "There is no block generated!"
}
```

13.4.27 block_getBlocks

查询指定区块区间的所有区块。

Parameters

1. <Object>

- from: <blockNumber> - 起始区块号。
- to: <blockNumber> - 终止区块号。
- isPlain: <boolean> - [可选] 默认值为false，表示返回的区块包括区块内的交易信息，如果指定为true，表示返回的区块不包括区块内的交易。

blockNumber 可以是十进制整数或者进制字符串，可以是“latest”字符串表示最新的区块。 from 必须小于等于 to ，否则会返回error。

Returns

1. [<Block>] - Block对象字段见 *Block*.

Example1：返回的区块包括交易信息

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlocks", "params": [{"from":2,"to":3}], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.0",
      "number": "0x3",
      "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
      "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "writeTime": 1481778653997475900,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
      "transactions": [
```

```
        {
      "version": "1.0",
      "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
      "blockNumber": "0x3",
      "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "amount": "0x0",
      "timestamp": 1481778652973000000,
      "nonce": 3573634504790373,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x81053a70000000000000000000000000000000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪"
    }
      ]
    },
    {
      "version": "1.0",
      "number": "0x2",
      "hash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "parentHash":
↪"0xe287c62aae77462aa772bd68da9f1a1ba21a0d044e2cc47f742409c20643e50c",
      "writeTime": 1481778642328872960,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
      "transactions": [
        {
      "version": "1.0",
      "hash": "0x07d606a25d1eab009f5374950383e9c0697599e6c35999337b969ba356800168",
      "blockNumber": "0x2",
      "blockHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "txIndex": "0x0",
      "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
      "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
      "amount": "0x0",
      "timestamp": 1481778641306000000,
      "nonce": 1628827117185765,
      "extra": "",
      "executeTime": "0x2",
      "payload":
↪"0x6fd7cc160000000000000000000000000000000000000000000000000000000000000000000000000000000000000303a0000000000000000000000
↪"
    }
      ]
    }
  ]
}
```


Example2：返回的区块不包括交易信息

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlocks", "params": [{"from":2,"to":3,"isPlain":true}], "id": 1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.0",
      "number": "0x3",
      "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
      "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "writeTime": 1481778653997475900,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
    },
    {
      "version": "1.0",
      "number": "0x2",
      "hash": "0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
      "parentHash":
↪"0xe287c62aae77462aa772bd68da9f1a1ba21a0d044e2cc47f742409c20643e50c",
      "writeTime": 1481778642328872960,
      "avgTime": "0x2",
      "txcounts": "0x1",
      "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
    }
  ]
}
```

13.4.28 block_getBlockByHash

根据区块的哈希值查询区块详细信息。

Parameters

1. `<string>` - 32字节的十六进制字符串，区块的哈希值。
2. `<boolean>` - 值为`true`，表示返回的区块不包括区块内的交易。值为`false`表示返回的区块包括区块内的交易信息。

Returns

1. `<Block>` - Block对象字段见 *Block*.

Example1：返回的区块包括交易信息

```
# Request
curl -X POST -data  '{"jsonrpc":"2.0","namespace":"global","method":"block_
↪getBlockByHash","params":[
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914", false],"id
↪":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0"
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a7000000000000000000000000000000000000000000000000000000000000000400000000000000000000000
↪"
      }
    ]
  }
}
```

Example2：返回的区块不包括交易信息

```
# Request
curl -X POST -data  '{"jsonrpc":"2.0","namespace":"global","method":"block_
↪getBlockByHash","params":[
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914", true],"id
↪":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
```

```
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
  }
}
```

13.4.29 block_getBlockByNumber

根据区块高度查询区块详细信息。

Parameters

1. <blockNumber> - 区块号，可以是十进制整数、进制字符串或“latest”字符串来表示最新区块。
2. <boolean> - 值为true，表示返回的区块不包括区块内的交易。值为false表示返回的区块包括区块内的交易信息。

Returns

1. <Block> - Block对象字段见 *Block*.

Example1：返回的区块包括交易信息

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlockByNumber", "params": ["0x3", false], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c",
    "transactions": [
      {
        "version": "1.0",
    "hash": "0xf57a6443d08cda4a3dfb8083804b6334d17d7af51c94a5f98ed67179b59169ae",
    "blockNumber": "0x3",
    "blockHash":
↪"0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "txIndex": "0x0",
```

```
    "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
    "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
    "amount": "0x0",
    "timestamp": 1481778652973000000,
    "nonce": 3573634504790373,
    "extra": "",
    "executeTime": "0x2",
    "payload":
↪"0x81053a70000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪"
      }
    ]
  }
}
```

Example2：返回的区块不包括交易信息

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "block_
↪getBlockByNumber", "params": ["0x3", true], "id": 1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "version": "1.0",
    "number": "0x3",
    "hash": "0x00acc3e13d8124fe799d55d7d2af06223148dc7bbc723718bb1a88fead34c914",
    "parentHash":
↪"0x2b709670922de0dda68926f96cffbe48c980c4325d416dab62b4be27fd73cee9",
    "writeTime": 1481778653997475900,
    "avgTime": "0x2",
    "txcounts": "0x1",
    "merkleRoot":
↪"0xc6fb0054aa90f3bfc78fe79cc459f7c7f268af7eef23bd4d8fc85204cb00ab6c"
  }
}
```

13.4.30 block_getAvgGenerateTimeByBlockNumber

查询区块平均生成时间。

Parameters

1. <Object>

- from: <blockNumber> - 起始区块号。
- to: <blockNumber> - 终止区块号。

blockNumber 可以是十进制整数或者进制字符串，可以是“latest”字符串表示最新的区块。 from 必须小于等于 to ， 否则会返回error。

Returns

1. <string> - 区块的平均生成时间（单位ms）。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":" block_
↪getAvgGenerateTimeByBlockNumber","params": [{"from": 10, "to": 19}],"id":71}'

# Response
{
  "id":71,
  "jsonrpc": "2.0",
  "namespace":"global",
  "code": 0,
  "message": "SUCCESS",
  "result": "0x32"
}
```

13.4.31 block_getBlocksByTime

查询指定时间区间内的区块数量。

Parameters

1. <Object>

- startTime: <number> - 起始unix时间戳（单位ns）。
- endTime: <number> - 结束unix时间戳（单位ns）。

Returns

1. <Object>

- sumOfBlocks: <string> - 区块总数。
- startBlock: <string> - 起始区块号。
- endBlock: <string> - 结束区块号。

Example1: 正常的请求

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"block_
↪getBlocksByTime","params":[{"startTime":1481778635567920177, "endTime
↪":1481778653997475900}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "sumOfBlocks": "0x3",
```

```
    "startBlock": "0x1",
    "endBlock": "0x3"
  }
}
```

Example2：如果起始时间和终止时间均大于链上最新区块的写入时间

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"block_
↪getBlocksByTime","params":[{"startTime":1481778635567920177, "endTime
↪":1481778653997475900}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": {
    "sumOfBlocks": "0x0",
    "startBlock": null,
    "endBlock": null
  }
}
```

13.4.32 block_getGenesisBlock

查询创世区块号。

Parameters

无

Returns

1. <string> - 区块号。

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getGenesisBlock","params
↪":[],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x8"
}
```

13.4.33 block_getChainHeight

查询最新区块号。

Parameters

无

Returns

1. <string> - 区块号。

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getChainHeight","params":[],
↪"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "0x11"
}
```

13.4.34 block_getBatchBlocksByHash

根据区块哈希列表批量查询区块详细信息。

Parameters

1. <Object>

- hashes: [<string>] - 要查询的区块哈希数组，哈希值为32字节的十六进制字符串。
- isPlain: <boolean> - 值为true，表示返回的区块不包括区块内的交易。值为false表示返回的区块包括区块内的交易信息。

Returns

1. [<Block>] - Block对象数组，Block对象字段见 *Block*.

Example1：返回的区块包含交易信息

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByHash",
↪"params":[{
    "hashes":["0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b"]
}],"id":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x3",
      "hash": "0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
      "parentHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "writeTime": 1509448178829111592,
      "avgTime": "0x0",
      "txcounts": "0x0",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
      "version": "1.3",
      "hash": "0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
          "blockNumber": "0x2",
          "blockHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
          "amount": "0x0",
          "timestamp": 1509440823410000000,
          "nonce": 8291834415403909,
          "extra": "",
          "executeTime": "0x6",
          "payload": "0x0a9ae69d"
        }
      ]
    }
  ]
}
```

Example2：返回的区块不包括交易信息

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByHash",
↪"params":[{
    "hashes":["0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b"],
    "isPlain": true
```

```
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x3",
      "hash": "0x810c92919fba632471b543905d8b4f8567c4fac27e5929d2eca8558c68cb7cf0",
      "parentHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "writeTime": 1509448178829111592,
      "avgTime": "0x0",
      "txcounts": "0x0",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    }
  ]
}
```

13.4.35 block_getBatchBlocksByNumber

根据区块号列表批量查询区块详细信息。

Parameters

1. `<Object>`

- `numbers: [<blockNumber>]` - 要查询的区块号数组，区块号可以是十进制整数或者进制字符串，也可以是“latest”字符串表示最新的区块。
- `isPlain: <boolean>` - 值为`true`，表示返回的区块不包括区块内的交易。值为`false`表示返回的区块包括区块内的交易信息。

Returns

1. `[<Block>]` - Block对象数组，Block对象字段见 *Block*.

Example1：返回的区块包括交易信息

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByNumber",
↪"params":[{
    "numbers": ["1","2"]
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x1",
      "hash": "0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
      "writeTime": 1509440821032039312,
      "avgTime": "0x11",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
          "version": "1.3",
          "hash":
↪"0x7aebde51531bb29d3ba620f91f6e1556a1e8b50913e590f31d4fe4a2436c0602",
          "blockNumber": "0x1",
          "blockHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0x0000000000000000000000000000000000000000",
          "amount": "0x0",
          "timestamp": 1509440820498000000,
          "nonce": 5098902950712745,
          "extra": "",
          "executeTime": "0x11",
          "payload":
↪"0x6060604052341561000f57600080fd5b604051604080610160833981016040528080519190602001805191505050
↪"
        }
      ]
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481",
      "transactions": [
        {
          "version": "1.3",
```

```
          "hash":
↪"0x22321358931c577ceaa2088d914758148dc6c1b6096a0b3f565d130f03ca75e4",
          "blockNumber": "0x2",
          "blockHash":
↪"0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
          "txIndex": "0x0",
          "from": "0x17d806c92fa941b4b7a8ffffc58fa2f297a3bffc",
          "to": "0xaeccd2fd1118334402c5de1cb014a9c192c498df",
          "amount": "0x0",
          "timestamp": 1509440823410000000,
          "nonce": 8291834415403909,
          "extra": "",
      "executeTime": "0x6",
      "payload": "0x0a9ae69d"
        }
      ]
    }
  ]
}
```

Example2：返回的区块不包括交易信息

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","method":"block_getBatchBlocksByNumber",
↪"params":[{
    "numbers": ["1","2"],
    "isPlain": true
}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
    {
      "version": "1.3",
      "number": "0x1",
      "hash": "0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "parentHash":
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
      "writeTime": 1509440821032039312,
      "avgTime": "0x11",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    },
    {
      "version": "1.3",
      "number": "0x2",
      "hash": "0x9c41efcc50ec6af6e3d14e1669f37bd1fc0cfe5836af6ab1e43ced98653c938b",
      "parentHash":
↪"0x4cd9f393aabb2df51c09e66925c4513e23f0dbbb9e94d0351c1c3ec7539144a0",
      "writeTime": 1509440823930976319,
      "avgTime": "0x6",
      "txcounts": "0x1",
      "merkleRoot":
↪"0x97b0d9473478886f5b0aee123d5652b15d4ae3ab41cc487cda9d8885cb003481"
    }
```

```
  ]
}
```

13.4.36 sub_newBlockSubscription

订阅新区块事件并且创建一个过滤器用于通知客户端，当有一个新区块产生的时候，该区块信息会缓存在过滤器中。

Parameters

1. <boolean> - 是否返回完整数据；值为true表示返回完整 *Block*对象；值为false表示只返回区块哈希。

Returns

1. <string> - 订阅标号。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪newBlockSubscription","params":[false],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.37 sub_newEventSubscription

订阅虚拟机事件并且创建一个过滤器用于通知客户端，当虚拟机事件被触发的时候，该事件日志会缓存在过滤器中。

Parameters

1. <Object>

- fromBlock: <number> - [可选] 十进制整数，表示起始区块号；若为空则默认没有限制。起始区块号大于或等于当前最新区块号。
- toBlock: <number> - [可选] 十进制整数，表示终止区块号；若为空则默认没有限制。终止区块号是大于起始区块号的未来某一个区块号。
- addresses: [<string>] - [可选] 表示监听指定地址的合约产生的事件；若为空则表示监听所有合约产生的事件。
- topics: [<string>][<string>] - [可选] 二维字符串数组，表示事件的话题，用于事件的内容过滤。若为空表示没有过滤条件。topics可能有以下组合：
  - [A, B] = A && B

- `[A, [B, C]] = A && (B || C)`
- `[null, A, B] = ANYTHING && A && B` null 表示通配符。

Returns

1. `<string>` - 订阅标号。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪newEventSubscription","params":[{
    "fromBlock":100,
    "addresses": ["000f1a7a08ccc48e5d30f80850cf1cf283aa3abd"]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.38 sub_getLogs

获取符合条件的虚拟机事件。

Parameters

1. `<Object>`

- `fromBlock`: `<number>` - [可选] 十进制整数，表示起始区块号；若为空则默认为0。起始区块号不能小于当前创世区块号。
- `toBlock`: `<number>` - [可选] 十进制整数，表示终止区块号；若为空则默认没有限制。终止区块号不能大于当前最新区块号。
- `addresses`: `[<string>]` - [可选] 一维数组，表示监听指定地址的合约产生的事件；若为空则表示监听所有合约产生的事件。
- `topics`: `[<string>][<string>]` - [可选] 二维字符串数组，表示事件的话题，用于事件的内容过滤。`topics`可能有以下组合：
  - `[A, B] = A && B`
  - `[A, [B, C]] = A && (B || C)`
  - `[null, A, B] = ANYTHING && A && B` null 表示通配符。

Returns

1. `[<Log>]` - 事件信息，Log对象字段如下：
   - `address`: `<string>` - 20字节的十六进制字符串，产生事件的合约地址。

- topics: [<string>] - 一系列的topic。
- data: <string> - 数据段。
- blockNumber: <number> - 十进制整数，所属区块号。
- blockHash: <string> - 所属区块哈希。
- txHash: <string> - 所属交易哈希。
- txIndex: <number> - 十进制整数，所属交易在当前区块交易列表中的偏移量。
- index: <number> - 十进制整数，该日志在本条交易产生的所有日志中的偏移量。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_getLogs
↪","params":[{
    "addresses": ["0x313bbf563991dc4c1be9d98a058a26108adfcf81"]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":[
    {
        "address":"0x313bbf563991dc4c1be9d98a058a26108adfcf81",
        "topics":[
↪"0x24abdb5865df5079dcc5ac590ff6f01d5c16edbc5fab4e195d9febd1114503da"],
        "data":"0000000000000000000000000000000000000000000000000000000000000064",
        "blockNumber":4,
        "blockHash":
↪"0xee93a66e170f2b20689cc05df27e290613da411c42a7bdfa951481c08fdefb16",
        "txHash":
↪"0xa676673a23f33a95a1a5960849ad780c5048dff76df961e9f78329b201670ae2",
        "txIndex":0,
        "index":0
    }
  ]
}
```

13.4.39 sub_newSystemStatusSubscription

订阅系统状态事件。

Parameters

1. <Object>

- modules: [<string>] - [可选] 一维数组，表示要订阅哪些模块的状态信息，若为空，则表示订阅所有模块。比如：**p2p**、**consensus**、**executor**等。
- modules_exclude: [<string>] - [可选] 一维数组，表示排除哪些模块的状态信息，若为空，则表示不排除。

- subtypes: [<string>] - [可选] 一维数组，表示要订阅模块下面的哪一类状态信息，若为空，则表示订阅所有类型。比如：**viewchange**等。
- subtypes_exclude: [<string>] - [可选] 一维数组，表示要排除模块下面的哪一类状态信息，若为空，则表示不排除。
- error_codes: [<number>] - [可选] 一维数组，元素为十进制整数，表示要订阅指定的具体哪一条状态信息，若为空，则表示订阅所有状态信息。
- error_codes_exclude: [<number>] - [可选] 一维数组，元素为十进制整数，表示要排除指定的具体哪一条状态信息，若为空，则表示不排除。

Returns

1. <string> - 订阅标号。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_␣
↪newSystemStatusSubscription","params":[{
    "modules":["executor", "consensus"],
    "subtypes": ["viewchange"],
    "error_codes_exclude": [-1, -2]
}],
"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":"0x7e533eb0647ecbe473ae610ebdd1bba6"
}
```

13.4.40 sub_getSubscriptionChanges

获取所订阅的事件。通过轮询这个方法，可以实时获取订阅的事件。

Parameters

1. <string> - 订阅标号。

Returns

1. <Array> - 一维数组，所订阅的事件返回的内容。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪getSubscriptionChanges","params":["0x7e533eb0647ecbe473ae610ebdd1bba6"], "id":1}'
```

```
# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":{}
}
```

13.4.41 sub_unSubscription

取消订阅。

Parameters

1. <string> - 订阅标号。

Returns

1. <boolean> - 值为true表示取消订阅指定事件成功，否则失败。

Example

```
# Request
curl -X POST --data '{"jsonrpc":"2.0", "namespace":"global", "method":"sub_
↪unsubscription","params":["0x7e533eb0647ecbe473ae610ebdd1bba6"], "id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result":true,
}
```

13.4.42 node_getNodes

获取节点信息。

Parameters

无

Returns

1. [<PeerInfo>] - PeerInfo对象字段如下

- id: <number> - 该节点id。
- ip: <string> - 该节点IP地址。

- port: <number> - 该节点的grpc端口号。
- namespace: <string> - 该节点所在分区。
- hash: <string> - 该节点哈希值。
- hostname: <string> - 节点主机名。
- isPrimary: <bool> - 表示该节点是否为主节点。
- isvp: <bool> - 表示该节点是否为VP节点。
- status: <number> - 表示该节点的状态，值为0表示节点处于Alive状态，值为1表示节点处于Pending状态，值为2表示节点处于Stop状态。
- delay: <number> - 表示该节点与本节点的延迟时间（单位ns），若为0，则为本节点。

Example

```
# Request
curl -X POST --data '{"jsonrpc": "2.0", "namespace":"global", "method": "node_
↪getNodes", "params": [],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": [
      {
          "id": 1,
          "ip": "127.0.0.1",
          "port": "50011",
          "namespace": "global",
          "hash": "fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a
↪",
          "hostname": "node1",
          "isPrimary": true,
          "isvp": true,
          "status": 0,
          "delay": 0
      },
      {
          "id": 2,
          "ip": "127.0.0.1",
          "port": "50012",
          "namespace": "global",
          "hash": "c82a71a88c58540c62fc119e78306e7fdbe114d9b840c47ab564767cb1c706e2
↪",
          "hostname": "node2",
          "isPrimary": false,
          "isvp": true,
          "status": 0,
          "delay": 347529
      },
      {
          "id": 3,
          "ip": "127.0.0.1",
          "port": "50013",
          "namespace": "global",
          "hash": "0c89dc7d8bdf45d1fed89fdbac27463d9f144875d3d73795f64f35dc204480fd
↪",
```

```
            "hostname": "node3",
            "isPrimary": false,
            "isvp": true,
            "status": 0,
            "delay": 369554
        },
        {
            "id": 4,
            "ip": "127.0.0.1",
            "port": "50014",
            "namespace": "global",
            "hash": "34d299742260716bab353995fe98727004b5c27bde52489f61de093176e82088
↪",
            "hostname": "node4",
            "isPrimary": false,
            "isvp": true,
            "status": 0,
            "delay": 430356
        }
    ]
}
```

13.4.43 node_getNodeHash

获取当前节点哈希值（向哪个节点发送请求即获取那个节点的哈希值）。

Parameters

无

Returns

1. <string> - 节点哈希值。

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"node_
↪getNodeHash","params":[],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"
}
```

13.4.44 node_deleteVP

删除VP节点。

说明：假设当前有5个VP节点，如果要删除第3个节点，则需要向其他4个节点都发送删除节点的请求，3号节点才能成功删除。

Parameters

1. <Object>

- nodehash: <string> - 要删除的VP节点的哈希值。

Returns

1. <string> - 请求发送成功的mesage。

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"node_
↪deleteVP","params":[{"nodehash":
↪"c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete vp node"
}
```

13.4.45 node_deleteNVP

VP节点断开与NVP节点的连接。

Parameters

1. <Object>

- nodehash: <string> - 要删除的NVP节点的哈希值。

Returns

1. <string> - 请求发送成功的mesage。

Example

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0","namespace":"global", "method":"node_
↪deleteNVP","params":[{"nodehash":
↪"c605d50c3ed56902ec31492ed43b238b36526df5d2fd6153c1858051b6635f6e"}],"id":1}'

# Response
{
  "jsonrpc": "2.0",
  "namespace":"global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
```

```
  "result": "successful request to delete nvp node"
}
```

13.4.46 cert_getTCert

获取节点颁发给用户的tcert证书。

Parameters

1. <Object>

- pubkey: <string> - 十六进制表示的pem格式公钥。

Returns

1. <Object>

- tcert: <string> - tcert证书。

Example：获取tcert失败

```
# Request
curl -X POST --data ' {"jsonrpc":"2.0", "namespace":"global", "method":"cert_
↪getTCert","params":[{
"pubkey":
↪"2d2d2d2d2d424547494e20454344534120055424c4943204b45592d2d2d2d2d0a424a4b73413554414d2b5763446c
↪"}],"id":"1"}'

# Response
{
   "jsonrpc": "2.0",
   "namespace":"global",
   "id": "1",
   "code": -32099,
   "message": "signed tcert failed"
}
```

CHAPTER 14

节点操作

14.1 1. 添加节点

添加节点需要涉及以下三个文件的配置:

- `peerconfig.toml`: 对端配置文件，用于配置逻辑对端节点连接信息。
- `hosts.toml`: 主机配置文件，用于配置物理对端节点连接信息，包括节点主机与IP地址的映射。
- `addr.toml`: 地址配置文件，用于配置物理互连地址信息，包括域名与IP地址的映射。当对端节点反向连接的时候需要用到此文件的配置。

14.1.1 Example1: 在已有4个VP节点的基础上，添加第五个VP节点

新增的VP节点在启动前需配置好以下配置文件:

1. `peerconfig.toml`

```
[[nodes]]
  hostname = "node1"
  id = 1
  static = true

[[nodes]]
  hostname = "node2"
  id = 2
  static = true

[[nodes]]
  hostname = "node3"
  id = 3
  static = true

[[nodes]]
  hostname = "node4"
  id = 4
  static = true

[[nodes]]
```

```
  hostname = "node5"
  id = 5
  static = true

[self]
  caconf = "config/namespace.toml"
  hostname = "node5"
  id = 5          # 节点ID
  n = 5           # 需要连接的vp的个数
  new = true      # 是否为新加入节点
  org = false     # 是否为创世节点
  rec = false     # 是否重连
  vp = true       # 是否为VP节点
```

2. hosts.toml

本文件需要配置**所有需要连接的节点的物理地址**，hyperchain节点之间通过hostname进行相互通信，所以hostname可以配置为任意的节点通信地址。

```
hosts = [
"node1 127.0.0.1:50011",
"node2 127.0.0.1:50012",
"node3 127.0.0.1:50013",
"node4 127.0.0.1:50014",
"node5 127.0.0.1:50015"
]
```

3. addr.toml

addr采用域的形式进行声明，例如hostname为node1和node5的两个节点同属domain1而其他节点都分别属于不同的domain下，则配置文件应该按照如下配置（本配置文件为node5的addr.toml）：

```
addrs = [
"domain1 127.0.0.1:50015",
"domain2 192.168.100.20:50015",
"domain3 202.110.20.13:50015",
"domain4 127.0.0.1:50015",
]
domain = "domain1"
```

注解： 重申一下，addr.toml是为了让别的节点知道自己所在的域以及让不同的域的节点通过不同的网络地址进行互连的配置，能够允许复杂网段之间的节点互连。

14.1.2 Example2: 在已有4个VP节点的基础上，添加一个NVP节点

新增的NVP节点在启动前需配置好以下配置文件:

1. peerconfig.toml

```
[[nodes]]
  hostname = "node1"
  id = 1
  static = true

[[nodes]]
  hostname = "node2"
  id = 2
  static = true

[[nodes]]
```

```
  hostname = "node3"
  id = 3
  static = true

[[nodes]]
  hostname = "node4"
  id = 4
  static = true

[self]
  caconf = "config/namespace.toml"
  hostname = "node5"
  id = 0
  n = 4      # 需要连接的vp的个数
  new = true     # 是否为新加入节点
  org = false    # 是否为创世节点
  rec = false    # 是否重连
  vp = false     # 是否为VP节点
```

2. hosts.toml

```
hosts = [
"node1 127.0.0.1:50011",
"node2 127.0.0.1:50012",
"node3 127.0.0.1:50013",
"node4 127.0.0.1:50014",
"node5 127.0.0.1:50015"
]
```

3. addr.toml

```
addrs = [
"domain1 127.0.0.1:50015",
"domain2 127.0.0.1:50015",
"domain3 127.0.0.1:50015",
"domain4 127.0.0.1:50015",
"domain5 127.0.0.1:50015"
]
domain = "domain5"
```

14.2 2. 删除节点

我们将删除节点分为三种情况：

1. VP断开与某个VP的连接；
2. VP主动断开与NVP的连接；
3. NVP主动断开与VP的连接；

第二种和第三种的结果是一样的，均能使NVP不再同步VP数据、NVP不再转发交易给VP。

在下面的例子中，我们均假设各个节点的JSON-RPC API服务端口映射如下：

- 1号节点：8081
- 2号节点：8082
- 3号节点：8083
- 4号节点：8084
- 5号节点：8085

14.2.1 Example1: VP断开与其他VP的连接

例如，当前有5个VP节点，现在要删除5号VP节点。

首先，获取要删除的5号VP节点的哈希，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"
}
```

然后，分别向1、2、3、4、5号VP节点发送删除节点请求，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteVP","params":[{"nodehash":
↪"55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"}],"id":1,
↪"namespace":"global"}' localhost:8081/8082/8083/8084/8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete vp node, hash␣
↪55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc"
}
```

当在终端看到以下日志时，说明VP节点删除成功。

```
global::p2p 12:33:17.709 DELETE NODE␣
↪55d3c05f2c24c232a47a1f1963ace172b21d3a2ec0ac83ea075da2d2427603bc
global::p2p 12:33:17.709 delete validate peer 5
```

14.2.2 Example2: VP断开与NVP的连接

例如，当前有4个VP节点和1个NVP节点，NVP节点与1号节点相连。

首先，获取NVP节点的哈希，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8085

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
```

```
  "result": "4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"
}
```

然后，向1号节点发送删除NVP的请求，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteNVP","params":[{"nodehash":
↪"4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"}],"id":1,
↪"namespace":"global"}' localhost:8081

# Response
{
  "jsonrpc": "2.0",
  "namespace": "global",
  "id": 1,
  "code": 0,
  "message": "SUCCESS",
  "result": "successful request to delete nvp node, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad"
}
```

在1号VP节点终端看到以下日志，

```
global::p2p 13:28:02.857 delete NVP peer, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad, vp pool␣
↪size(4) nvp pool size(0)
```

同时，NVP节点也会打印以下日志说明与1号节点已经断开连接。

```
global::p2p 13:28:02.858 peers_pool.go:244 delete validate peer 1
```

说明NVP节点删除成功。

14.2.3 Example3: NVP断开与VP的连接

这种情况与Example2差不多，不同的是我们这次是向NVP节点发送删节点请求。

例如，当前有4个VP节点和1个NVP节点，NVP节点与1号节点相连。

首先，获取1号VP节点的哈希，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_getNodeHash","params":[],"id":1,
↪"namespace":"global"}' localhost:8081

# Response
{
    "jsonrpc": "2.0",
    "namespace": "global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"
}
```

然后，向NVP节点发送删除VP的请求，

```
# Request
curl -X POST -d '{"jsonrpc":"2.0","method":"node_deleteVP","params":[{"nodehash":
↪"fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"}],"id":1,
↪"namespace":"global"}' localhost:8085
```

```
# Response
{
    "jsonrpc": "2.0",
    "namespace": "global",
    "id": 1,
    "code": 0,
    "message": "SUCCESS",
    "result": "successful request to delete vp node, hash␣
↪fa34664ec14727c34943045bcaba9ef05d2c48e06d294c15effc900a5b4b663a"
}
```

在NVP节点终端看到以下日志，

```
global::p2p 13:47:17.744 delete validate peer 1
```

同时，在1号VP节点终端看到以下日志，

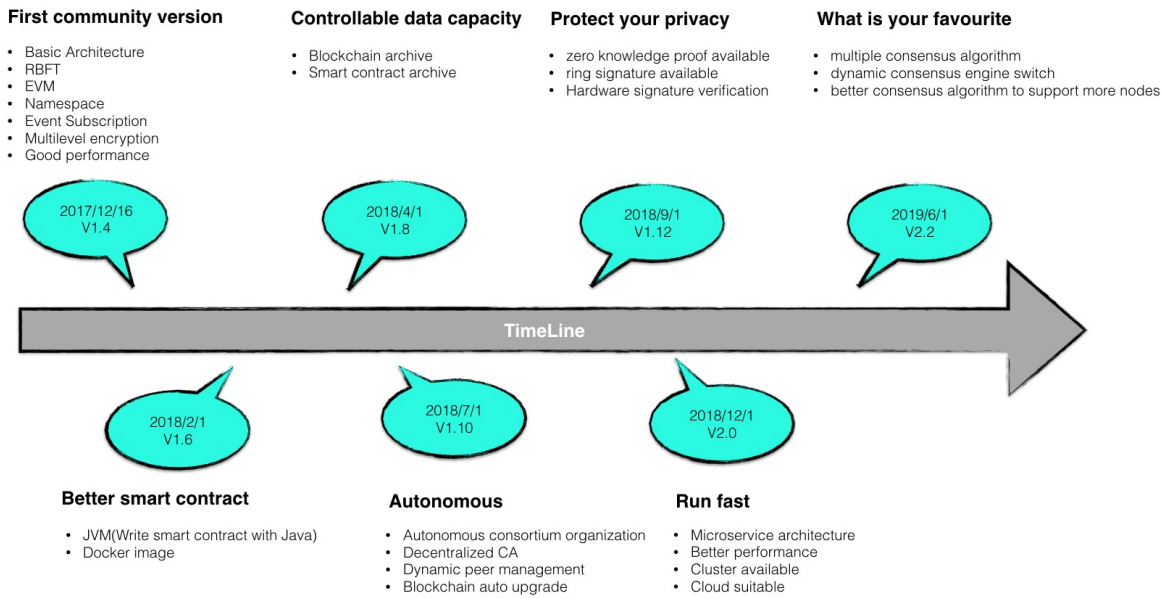
```
global::p2p 13:47:17.744 delete NVP peer, hash␣
↪4886947d8191b62a1141dbc3250a0cc61a436ca28829f40cb5a690c7449825ad, vp pool␣
↪size(4) nvp pool size(0)
```

说明VP节点删除成功。

CHAPTER 15

Hyperchain 开发路线图

Hyperchain目前在开源社区开放的是1.4稳定版本，在今后还将不断推出新的特性。欢迎Hyperchain社区的爱好者一起参与到我们的开发到中！



15.1 First community version

这是Hyperchain的第一个社区版本，包括了联盟链平台的完整组件：

1. 基于RBFT算法的共识引擎；
2. 兼容以太坊的EVM智能合约虚拟机；
3. 数据分区，从物理上实现业务隔离；
4. 丰富的事件订阅接口，实时捕捉区块链平台动态；

5. 多级加密机制，包括非对称加密，对称秘钥，基于数字证书的准入机制；
6. 可用级别的性能表现，满足大部分业务场景的需求；

15.2 Better smart contract

这个版本拟计划于2018年2月份推出，主要特性为支持用Java语言来编写智能合约，降低区块链应用开发的门槛。

除此之外，还将推出更加容易使用的hyperchain docker镜像，支持一键集群部署。

15.3 Controllable data capacity

目前现有的区块链数据都是累增的，因此区块链数据本身的存储容量将会是一个很大的问题。

我们拟将于2018年4月推出区块链数据的归档策略，可以支持区块链数据的归档以及合约数据的归档。

15.4 Autonomous

现有的联盟链准入机制存在以下几个问题：

- 存在单点故障；
- 容易被单个节点控制整个联盟链网络；
- 自动化程度较低；

因此，我们拟将在2018年7月推出一个实现去中心化的自治的准入机制的版本，能够实现自动化的成员管理，身份切换，版本升级等。

15.5 Protect your pricacy

目前在同一个分区中，区块链数据是被所有节点所共享的，从而也就没有用户隐私可言。在2018年的9月，我们将推出支持两种高级密码学特性的版本，来实现用户隐私的保护。

两个密码学技术分别是（1）零知识证明（2）环签名技术。

15.6 Run fast

目前Hyperchain的性能表现虽然能够满足大部分业务场景的需求，但是对于特别高频且复杂的场景还是束手无策。因此，我们将对现有的架构做较大的调整，以微服务，适合云计算的架构来改造hyperchain，从而达到提升性能表现的目的。

该版本计划于2018年的12月推出。

15.7 What is your favourite

在2019年的6月，我们将推出一个支持多种共识算法的版本，能够支持动态的共识引擎切换，并计划提出一个全新的共识算法，能够在联盟链中支持更大的节点规模。