• 前言

LiteSDK是一个轻量JavaSDK工具，提供与Hyperchain区块链平台交互的接口以及一些处理工具。该文档面向Hyperchain区块链平台的应用开发者，提供Hyperchain Javasdk SDK的使用说明。

第**二**章．初始化

2.1 创建HttpProvider对象

HttpProvider是一个接口，负责管理与节点的连接，实现HttpProvider接口的类需要提供底层的通信实现，目前LiteSDK已有默认的实现类DefaultHttpProvider，创建DefaultHttpProvider需要通过Builder模式创建，示例如下：

**• public** **static** **final** String node1 = "localhost:8081";

• HttpProvider httpProvider = **new** DefaultHttpProvider.Builder()

•                 .setUrl(node1)

•                 .https(tlsca, tls\_peer\_cert, tls\_peer\_priv)

•                 .build();

• setUrl() 可以设置连接的节点URL（格式为ip + jsonRPC端口）

• https()设置启动https协议连接并设置使用的证书(需要传的参数类型为输入流)。

2.2 创建ProviderManager对象

每个节点的连接都需要一个HttpProvider，而ProvideManager负责集成、管理这些HttpProvider，创建ProvideManager有两种方式，一种是通过createManager()创建，另一种是和HttpProvider一样通过Builder模式创建。使用前者创建会使用ProvideManager的默认配置参数，而如果想定制更多的属性则需要通过后者的方式创建，示例如下：

• // 方式1

• ProviderManager providerManager = ProviderManager.createManager(HttpProvider);

•

• // 方式2

• providerManager = **new** ProviderManager.Builder()

•                 .namespace("global")

•                 .providers(httpProvider1, httpProvider2, httpProvider3, httpProvider4)

•                 .enableTCert(sdkcert\_cert, sdkcert\_priv, unique\_pub, unique\_priv)

•                 .build();

**方式一**：

只需要传HttpProvider对象，其他都使用ProvideManager的默认配置，如不启用证书、使用的namespace配置项为global。

**方式二：**

**• namespace()可以设置对应的namespace名;**

**• providers()设置需要管理的HttpProvider对象们;**

**• enableTCert()设置使用的证书(需要传的参数类型为输入流)。注：例子中未出现的方法还有一个cfca(InputStream sdkCert, InputStream sdkCertPriv)，功能与enableTCert()相同，两者的区别是证书校验是否通过cfca机构，且在创建ProvideManager对象过程中两个方法只能使用其中一个。**

2.3 创建服务

​相关的一类服务集合由一个专门的Service接口管理，并通过对应的实现类实现具体的创建过程（如封装发送请求需要附带的参数）。LiteSDK通过ServiceManager类负责管理创建所有的Service对象，以下是一个创建获取节点信息的服务的例子：

• // 将ProviderManager对象作为参数，通过getNodeService()创建NodeService类型的对象

• // NodeService为声明的接口， 实际类型为NodeServiceImpl

• NodeService nodeService = ServiceManager.getNodeService(providerManager);

•

• // 通过调用NodeService提供的方法创建相应的服务，类型为Request<NodeResponse>

• NodeRequest nodeRequest = nodeService.getNodes();

实际上每个服务创建对应创建一个请求，这个请求都继承了共同的父类——Request，LiteSDK将根据不同的Service接口，返回不同Request子类，同时将用户调用接口的参数params封装到Request请求中，而在创建Request的过程中会附带一个具体的响应类型的声明，该响应类型也将根据不同的Service接口与Request绑定。

Request拥有send()和sendAsync()同步发送和异步发送两个方法：

• send(): 同步发送返回Request根据不同接口绑定的Response

• sendAsync(): 异步发送返回Request根据不同接口绑定了Response的Future接口

2.4 获取结果

同样地，响应也都继承了共同的父类——Response，通过调用Request的send()方法得到，LitesSDK会将不同的返回结果result根据接口封装成不同的Response子类，如 2.3 所说Response类型在生成Request时绑定。Response可以获取状态码、状态消息等，而不同的Response可以获取到不同的结果，有时也需要进一步获取到更具体的信息。示例如下：

• NodeResponse nodeResponse = nodeRequest.send();

• System.out.println(nodeResponse.getResult());

当ProvideManager管理多个节点连接时，返回的节点信息应该是一个数组，这时就需要调用示例中的getResult()方法将返回结果转换成更准确的类型。

第三章. 交易

LiteSDK的交易接口需要用到交易体，交易体的应用场景分为两类：一类是普通的转账交易，不涉及虚拟机，一类是合约交易，和虚拟机相关。两者虽然都名为交易，但实际执行的功能和应用场景都不同，且转账交易的实现由TxService提供，合约交易的实现由ContractService提供。

转账交易

转账交易的实现主要是TxService提供，主要有两个接口。

• Request<TxHashResponse> sendTx(Transaction transaction, **int**... nodeIds);

•

• Request<TxHashesResponse> sendBatchTxs(ArrayList<Transaction> transactions, ArrayList<String> methods, **int**... nodeIds);

两个Request分别绑定了TxHashResponse和TxHashesResponse，当拿到这两个响应时调用polling()方法就可以获取真正的交易回执。前者返回ReceiptResponse，后者返回元素是ReceiptResponse的ArrayList。转账交易和合约接口类似，主要的不同在于交易体的创建，转账交易通过内部类Builder调用transfer()方法创建。

**• class** Builder {

•     **public** Builder transfer(String to, **long** value);

• }

•

• // example:

• Transaction transaction = **new** Transaction.Builder(account.getAddress()).transfer("794BF01AB3D37DF2D1EA1AA4E6F4A0E988F4DEA5", 0).build();

合约接口

以交易体结构为核心的交易主要应用在合约交易上，即将想要执行的操作和数据封装成一笔交易体，再调用合约服务(ContractService)的接口去执行。

绑定合约接口的Response子类只有TxHashResponse，里面封装了ReceiptResponse类型的参数，实际是tx hash，拿到TxHashResponse后调用polling方法可通过tx hash去查找获取真正的交易回执。

TxHashResponse的主要方法如下：

• /\*\*

•  \* 通过交易hash获取交易回执.

•  \*

•  \* @return 返回 ReceiptResponse

•  \* @throws RequestException -

•  \*/

**• public** ReceiptResponse polling() **throws** RequestException;

•

• /\*\*

•  \* 获取交易hash.

•  \*

•  \* @return 交易hash

•  \*/

**• public** String getTxHash();

LiteSDK的合约接口较特殊，目前提供了部署合约、调用合约、管理合约三种接口。

**• public** **interface** ContractService {

•     Request<TxHashResponse> deploy(Transaction transaction, **int**... nodeIds);

•

•     Request<TxHashResponse> invoke(Transaction transaction, **int**... nodeIds);

•

•     Request<TxHashResponse> maintain(Transaction transaction, **int**... nodeIds);

• }

根据要创建的合约服务不同，封装的Transaction交易体也会不同。并且LiteSDK支持HVM、EVM两种形式的合约，这两种也会影响到交易体的创建。

**创建交易体并调用服务的具体流程如下：**

账户创建

这个过程分为两步，先创建AccountService对象，再利用该对象创建账户，示例如下：

如第二章所说，创建Service对象需要指定ProviderManager对象，且使用genAccount()创建账户时需要指定加密算法，如示例中使用SMRAW算法（只有ECRAW、SMRAW不需要密码参数，其余的加密算法需要手动设置password）。

• AccountService accountService = ServiceManager.getAccountService(providerManager);

• Account account = accountService.genAccount(Algo.SMRAW);

AccountService提供的接口如下：

**• public** **interface** AccountService {

•     Account genAccount(Algo algo);

•

•     Account genAccount(Algo algo, String password);

•

•     Account fromAccountJson(String accountJson);

•

•     Account fromAccountJson(String accountJson, String password);

• }

目前Account服务支持的所有加密算法如下：

**• public** **enum** Algo {

•     ECDES("0x02"),

•     ECRAW("0x03"),

•     ECAES("0x04"),

•     EC3DES("0x05"),

•

•     SMSM4("0x11"),

•     SMDES("0x12"),

•     SMRAW("0x13"),

•     SMAES("0x14"),

•     SM3DES("0x15");

• }

交易体创建

LiteSDK使用Builder模式来负责对Transaction的创建，通过调用build()函数来获取到Transaction实例。HVM和EVM分别有各自的Builder：HVMBuilder、EVMBuilder，继承同一个父类Builer。目前Builder模式提供了五种交易体的封装，分别对应部署合约、调用合约、升级合约、冻结合约、解冻合约，其中前两个服务的交易体分别定义在HVM、EVM各自的Builder子类中，后三者都是管理合约这一服务的子服务，定义在父类Builder中。

**• class** Builder {

•     Builder upgrade(String contractAddress, String payload);

•     Builder freeze(String contractAddress);

•     Builder unfreeze(String contractAddress);

•     Transaction build();

• }

•

**• class** HVMBuilder **extends** Builder {

•     Builder deploy(InputStream fis);

•     Builder invoke(String contractAddress, BaseInvoke baseInvoke);

• }

•

**• class** EVMBuilder **extends** Builder {

•     // 当合约无构造参数时使用，不需abi参数

•     Builder deploy(String bin);

•     // 当合约需要提供abi解析构造方法参数时使用

•     Builder deploy(String bin, Abi abi, FuncParams params);

•     Builder invoke(String contractAddress, String methodName, Abi abi, FuncParams params);

• }

下面是创建各个服务的交易体Transaction的实例。

部署合约

HVM

• InputStream payload = FileUtil.readFileAsStream("hvm-jar/hvmbasic-1.0.0-student.jar");

•

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).deploy(payload).build();

EVM

• InputStream inputStream1 = FileUtil.readFileAsStream("solidity/sol2/TestContract\_sol\_TypeTestContract.bin");

• InputStream inputStream2 = FileUtil.readFileAsStream("solidity/sol2/TestContract\_sol\_TypeTestContract.abi");

• String bin = FileUtil.readFile(inputStream1);

• String abiStr = FileUtil.readFile(inputStream2);

•

• FuncParams params = **new** FuncParams();

• params.addParams("contract01");

• Transaction transaction = **new** Transaction.EVMBuilder(account.getAddress()).deploy(bin, abi, params).build();

• // 如果要部署的合约无构造函数，则调用如下

• // Transaction transaction = new Transaction.HVMBuilder(account.getAddress()).deploy(bin).build();

创建交易体时需要指定要部署的合约的bin、abi文件的字符串内容以及合约名。

调用合约

HVM

hvm调用合约有两种方式：

- invoke bean调用

- 直接调用合约方法（类似evm）

1. invoke bean调用如下：

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).invoke(contractAddress, invoke).build();

创建交易体时需要指定**合约地址**和**invoke bean**（HVM中新提出的概念，可点击链接**http://hvm.internal.hyperchain.cn/#/**了解）。

2. 直接调用合约方法如下：

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).invoke(contractAddress, params).build();

params类型为**InvokeDirectlyParams**，具体的构造方式见附录。

升级合约

HVM

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).upgrade(contractAddress, payload).build();

创建交易体时需要指定合约地址和读取新合约jar包得到的字符串

EVM

• Transaction transaction = **new** Transaction.EVMBuilder(account.getAddress()).upgrade(contractAddress, payload).build();

创建交易体时需要指定合约地址和升级的新合约的bin文件字符串。

冻结合约

HVM

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).freeze(contractAddress).build();

创建交易体时需要指定合约地址。

EVM

• Transaction transaction = **new** Transaction.EVMBuilder(account.getAddress()).freeze(contractAddress).build();

创建交易体时需要指定合约地址。

解冻合约

HVM

• Transaction transaction = **new** Transaction.HVMBuilder(account.getAddress()).unfreeze(contractAddress).build();

创建交易体时需要指定合约地址。

EVM

• Transaction transaction = **new** Transaction.EVMBuilder(account.getAddress()).unfreeze(contractAddress).build();

创建交易体时需要指定合约地址。

交易体签名

通过Transaction提供的sign()方法，需要指定Account对象。

• transaction.sign(account);

创建请求

这个过程分为两步，先创建ContractService对象，再指定之前构造的交易体调用相应的服务接口，示例如下：

发送交易体

这个过程实际分为两步，调用send()部署合约拿到响应，再对响应解析拿到ReceiptResponse（执行结果），这是合约相关接口独有的，其他接口一般只需要调用send()方法拿到响应就结束了。

• ContractService contractService = ServiceManager.getContractService(providerManager);

• Request<TxHashResponse> contractRequest = contractService.deploy(transaction);

第四章. Transaction接口(TxService)

注：该章的Transaction与第三章的交易体概念不同，该章的接口主要主要用于查询之前在链上的执行信息，将返回的信息封装为Transaction结构体。

TxService接口繁多，返回的执行结果根据情况封装共对应四种响应：

• TxResponse

• TxCountWithTSResponse

• TxCountResponse

• TxAvgTimeResponse

分别对应的结构如下：

**TxResponse**

通过result接收返回结果，result实际结构是内部类Transaction，可通过getResult()方法得到。

**• public** **class** TxResponse **extends** Response {

•     **public** **class** Transaction {

•         **private** String version;

•         **private** String hash;

•         **private** String blockNumber;

•         **private** String blockHash;

•         **private** String txIndex;

•         **private** String from;

•         **private** String to;

•         **private** String amount;

•         **private** String timestamp;

•         **private** String nonce;

•         **private** String extra;

•         **private** String executeTime;

•         **private** String payload;

•         **private** String signature;

•         **private** String blockTimestamp;

•         **private** String blockWriteTime;

•     }

•     **private** JsonElement result;

• }

**TxCountWithTSResponse**

通过result接收返回结果，result实际类型是内部类TxCount，可通过getResult()方法得到。

**• public** **class** TxCountWithTSResponse **extends** Response {

•     **private** **class** TxCount {

•         **private** String count;

•         **private** **long** timestamp;

•     }

•     **private** TxCount result;

• }

**TxCountResponse**

**• public class**TxCountResponse**extends**Response**{**

**•     private String**result**;**

**• }**

通过result接收返回结果，result实际类型是String，可通过getResult()方法得到。

**TxAvgTimeResponse**

通过result接收返回结果，result实际类型是String，可通过getResult()方法得到。

**• public class**TxAvgTimeResponse**extends Response {**

**•     private String**result**;**

**• }**

4.1 查询指定区块区间的交易(getTransactions)

参数：

• from 区块区间起点

• to 区块区间终点

• nodeIds 说明请求向哪些节点发送

**• Request<TxResponse> getTx(BigInteger** from**, BigInteger** to**, int...** nodeIds**);**

重载方法如下：

**• Request<TxResponse> getTx(String** from**, String** to**, int...** nodeIds**);**

4.2 查询所有非法交易(getDiscardTransactions)

参数：

• nodeIds 说明请求向哪些节点发送

**• Request<TxResponse>** getDiscardTx**(String** from**, String** to**, int...** nodeIds**);**

4.3 查询交易 by transaction hash(getTransactionByHash)

参数：

• txHash 交易hash

• nodeIds 说明请求向哪些节点发送

**• Request<TxResponse>** getTxByHash**(String** txHash**, int...** nodeIds**);**

参数：

• txHash 交易hash

• isprivateTx 是否获取隐私交易，若设false，则该方法和上一个方法作用一样

• nodeIds 请求向哪些节点发送

• Request<TxResponse> getTxByHash(String txHash, boolean isPrivateTx, int... nodeIds);

4.4 查询交易by block hash(getTxByBlockHashAndIndex)

参数：

• blockHash 区块哈希值

• index 区块内的交易索引值

• nodeIds 请求向哪些节点发送

• Request<TxResponse> getTxByBlockHashAndIndex(String blockHash, **int** index, **int**... nodeIds);

4.5 查询交易by block number(getTxByBlockNumAndIndex)

参数：

• blockNumber 区块号

• index 区块内的交易索引值

• nodeIds 请求向哪些节点发送

• Request<TxResponse> getTxByBlockNumAndIndex(**int** blockNumber, **int** idx, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getTxByBlockNumAndIndex(String blockNumber, String idx, **int**... nodeIds);

4.6 查询指定区块区间交易平均处理时间(getTxAvgTimeByBlockNumber)

参数：

• from 区块区间起点

• to 区块区间终点

• nodeIds 说明请求向哪些节点发送

• Request<TxAvgTimeResponse> getTxAvgTimeByBlockNumber(BigInteger from, BigInteger to, **int**... nodeIds);

重载方法如下：

• Request<TxAvgTimeResponse> getTxAvgTimeByBlockNumber(String from, String to, **int**... nodeIds);

4.7 查询链上所有交易量(getTransactionsCount)

参数：

• nodeIds 说明请求向哪些节点发送。

• Request<TxCountWithTSResponse> getTransactionsCount(**int**... nodeIds);

4.8 查询交易回执信息by transaction hash(getTransactionReceipt)

参数：

• txHash 交易hash。

• nodeIds 说明请求向哪些节点发送。

• Request<ReceiptResponse> getTransactionReceipt(String txHash, **int**... nodeIds);

4.9 查询区块交易数量by block hash(getBlockTxCountByHash)

参数：

• blockHash 区块hash。

• nodeIds 说明请求向哪些节点发送。

• Request<TxCountWithTSResponse> getBlockTxCountByHash(String blockHash, **int**... nodeIds);

4.10 查询区块交易数量by block number(getBlockTxCountByNumber)

参数：

• blockNumber 区块号。

• nodeIds 说明请求向哪些节点发送。

• Request<TxCountWithTSResponse> getBlockTxCountByNumber(String blockNumber, **int**... nodeIds);

4.11 获取交易签名哈希(getSignHash)

**部署合约时**

参数：

• from 发起者地址。

• nonce 16位的随机数，该值必须为十进制整数。

• extra(可选) 额外信息。

• payload 字节编码。

• timestamp 交易时间戳。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getSignHash(String from, BigInteger nonce, String extra, String payload, BigInteger timestamp, **int**... nodeIds);

• Request<TxResponse> getSignHash(String from, BigInteger nonce, String extra, String payload, BigInteger timestamp, **int**... nodeIds);

**普通交易**

参数：

• from 发起者地址。

• nonce 16位的随机数，该值必须为十进制整数。

• extra（可选） 额外信息。

• value 交易值。

• timestamp 交易时间戳。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getSignHash(String from, String to, BigInteger nonce, String extra, String value, BigInteger timestamp, **int**... nodeIds);

• Request<TxResponse> getSignHash(String from, String to, BigInteger nonce, String value, BigInteger timestamp, **int**... nodeIds);

4.12 查询指定时间区间内的交易(getTransactionsByTime)

参数：

• startTime 起起始时间戳(单位ns)。

• endTime 结束时间戳(单位ns)。

• limit（可选） 符合条件的区块数目最大值，默认值为50。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getTransactionsByTime(BigInteger startTime, BigInteger endTime, **int**... nodeIds);

• Request<TxResponse> getTransactionsByTime(BigInteger startTime, BigInteger endTime, int limit, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getTransactionsByTime(String startTime, String endTime, **int**... nodeIds;

• Request<TxResponse> getTransactionsByTime(String startTime, String endTime, int limit, **int**... nodeIds;

4.13 查询指定时间区间内的非法交易(getDiscardTransactionsByTime)

参数：

• startTime 起起始时间戳(单位ns)。

• endTime 结束时间戳(单位ns)。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getDiscardTransactionsByTime(BigInteger startTime, BigInteger endTime, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getDiscardTransactionsByTime(String startTime, String endTime, **int**... nodeIds);

4.14 查询区块区间交易数量by contract address(getTransactionsCountByContractAddr)

参数：

• from 起始区块号。

• to 终止区块号。

• address 合约地址。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getTransactionsCountByContractAddr(String from, String to, String address, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getTransactionsCountByContractAddr(BigInteger from, BigInteger to, String address, **int**... nodeIds);

4.15 查询下一页交易(getNextPageTransactions)

参数：

• blkNumber 从该区块开始计数。

• txIndex 起始交易在blkNumber号区块的位置偏移量。

• minBlkNumber 截止计数的最小区块号。

• maxBlkNumber 截止计数的最大区块号。

• separated 表示要跳过的交易条数（一般用于跳页查询）。

• pageSize 表示要返回的交易条数。

• containCurrent true表示返回的结果中包括blkNumber区块中位置为txIndex的交易，如果该条交易不是合约地址为address合约的交易，则不算入。

• address 合约地址。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getNextPageTransactions(BigInteger blkNumber, BigInteger txIndex, BigInteger minBlkNumber, BigInteger maxBlkNumber, BigInteger separated, BigInteger pageSize, **boolean** containCurrent, String address, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getNextPageTransactions(String blkNumber, String txIndex, String minBlkNumber, String maxBlkNumber, String separated, String pageSize, **boolean** containCurrent, String address, **int**... nodeIds);

4.16 查询上一页交易(getPrevPageTransactions)

参数：

• blkNumber 从该区块开始计数。

• txIndex 起始交易在blkNumber号区块的位置偏移量。

• minBlkNumber 截止计数的最小区块号。

• maxBlkNumber 截止计数的最大区块号。

• separated 表示要跳过的交易条数（一般用于跳页查询）。

• pageSize 表示要返回的交易条数。

• containCurrent true表示返回的结果中包括blkNumber区块中位置为txIndex的交易，如果该条交易不是合约地址为address合约的交易，则不算入。

• address 合约地址。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getPrevPageTransactions(BigInteger blkNumber, BigInteger txIndex, BigInteger minBlkNumber, BigInteger maxBlkNumber, BigInteger separated, BigInteger pageSize, **boolean** containCurrent, String address, **int**... nodeIds);

重载方法如下：

• Request<TxResponse> getPrevPageTransactions(String blkNumber, String txIndex, String minBlkNumber, String maxBlkNumber, String separated, String pageSize, **boolean** containCurrent, String address, **int**... nodeIds);

4.17 查询批量交易by hash list(getBatchTxByHash)

参数：

• txHashList 交易的哈希数组, 哈希值为32字节的十六进制字符串。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getBatchTxByHash(ArrayList<String> txHashList, **int**... nodeIds);

4.18 查询批量回执by hash list(getBatchReceipt)

参数：

• txHashList 交易的哈希数组, 哈希值为32字节的十六进制字符串。

• nodeIds 说明请求向哪些节点发送。

• Request<ReceiptListResponse> getBatchReceipt(ArrayList<String> txHashList, **int**... nodeIds);

4.19 查询指定时间区间内的交易数量(getTxsCountByTime)

参数：

• startTime 起起始时间戳(单位ns)。

• endTime 结束时间戳(单位ns)。

• nodeIds 说明请求向哪些节点发送。

• Request<TxResponse> getTxsCountByTime(BigInteger startTime, BigInteger endTime, **int**... nodeIds);

第五章. BlockService相关接口

BlockService接口与TxService相似，只是获取的对象是区块信息。同样地，BlockService对象也有很多对应的响应类型：

• BlockResponse

• BlockNumberResponse

• BlockAvgTimeResponse

• BlockCountResponse

分别对应的结构如下。

**BlockResponse**

通过result接收返回结果，result实际类型是内部类Block，可通过getResult()方法得到。

**• public** **class** BlockResponse **extends** Response {

•     **private** **class** Block {

•         **private** String version;

•         **private** String number;

•         **private** String hash;

•         **private** String parentHash;

•         **private** String writeTime;

•         **private** String avgTime;

•         **private** String txcounts;

•         **private** String merkleRoot;

•     }

•     **private** JsonElement result;

• }

**BlockNumberResponse**

通过result接收返回结果，result实际类型是String，可通过getResult()方法得到。

**• public** **class** BlockNumberResponse **extends** Response {

•     **private** String result;

• }

**BlockAvgTimeResponse**

通过result接收返回结果，result实际类型String，可通过getResult()方法得到。

**• public** **class** BlockAvgTimeResponse **extends** Response {

•     @Expose

•     **private** String result;

• }

**BlockCountResponse**

通过result接收返回结果，result实际类型是内部类BlockCount，可通过getResult()方法得到。

**• public** **class** BlockCountResponse **extends** Response {

•     **private** **class** BlockCount {

•         **private** String sumOfBlocks;

•         **private** String startBlock;

•         **private** String endBlock;

•     }

•     **private** BlockCount result;

• }

5.1 获取最新区块(getLastestBlock)

参数：

• nodeIds 说明请求向哪些节点发送。

• Request<BlockResponse> getLastestBlock(**int**... nodeIds);

5.2 查询指定区间的区块by block number(getBlocks)

参数：

• from 起始区块号。

• to 终止区块号。

• isPlain (可选)，默认为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。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockResponse> getBlocks(BigInteger from, BigInteger to, **int**... nodeIds);

• Request<BlockResponse> getBlocks(BigInteger from, BigInteger to, **boolean** isPlain, **int**... nodeIds);

重载方法如下：

• Request<BlockResponse> getBlocks(String from, String to, **int**... nodeIds);

• Request<BlockResponse> getBlocks(String from, String to, **boolean** isPlain, **int**... nodeIds);

5.3 查询区块by block hash(getBlockByHash)

参数：

• blockHash 区块的哈希值,32字节的十六进制字符串。

• isPlain (可选) 默认为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。

• nodeIds 说明请求向哪些节点发送

• Request<BlockResponse> getBlockByHash(String blockHash, **int**... nodeIds);

• Request<BlockResponse> getBlockByHash(String blockHash, **boolean** isPlain, **int**... nodeIds);

5.4 查询区块by block number(getBlockByNum)

参数：

• blockNumber 区块号。

• isPlain (可选) 默认为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。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockResponse> getBlockByNum(BigInteger blockNumber, **int**... nodeIds);

• Request<BlockResponse> getBlockByNum(BigInteger blockNumber, **boolean** isPlain, **int**... nodeIds);

重载方法如下：

• Request<BlockResponse> getBlockByNum(String blockNumber, **int**... nodeIds);

• Request<BlockResponse> getBlockByNum(String blockNumber, **boolean** isPlain, **int**... nodeIds);

5.5 查询区块平均生成时间(getAvgGenerateTimeByBlockNumber)

参数：

• from 起始区块号。

• to 终止区块号。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockAvgTimeResponse> getAvgGenerateTimeByBlockNumber(BigInteger from, BigInteger to, **int**... nodeIds);

重载方法如下：

• Request<BlockAvgTimeResponse> getAvgGenerateTimeByBlockNumber(String from, String to, **int**... nodeIds);

5.6 查询指定时间区间内的区块数量(getBlocksByTime)

参数：

• startTime 起始时间戳(单位ns)。

• endTime 结束时间戳(单位ns)。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockCountResponse> getBlocksByTime(BigInteger startTime, BigInteger endTime, **int**... nodeIds);

重载方法如下：

• Request<BlockCountResponse> getBlocksByTime(String startTime, String endTime, **int**... nodeIds);

5.7 查询最新区块号，即链高(getChainHeight)

参数：

• nodeIds 说明请求向哪些节点发送。

• Request<BlockNumberResponse> getChainHeight(**int**... nodeIds);

5.8 查询创世区块号(getChainHeight)

参数：

• nodeIds 说明请求向哪些节点发送。

• Request<BlockNumberResponse> getGenesisBlock(**int**... nodeIds);

5.9 查询批量区块by block hash list(getBatchBlocksByHash)

参数：

• blockHashList 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 (可选) 默认为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。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockResponse> getBatchBlocksByHash(ArrayList<String> blockHashList, **int**... nodeIds);

• Request<BlockResponse> getBatchBlocksByHash(ArrayList<String> blockHashList, **boolean** isPlain, **int**... nodeIds);

5.10 查询批量区块by block number list(getBatchBlocksByNum)

参数：

• blockNumberList 要查询的区块号数组。

• isPlain (可选) 默认为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。

• nodeIds 说明请求向哪些节点发送。

• Request<BlockResponse> getBatchBlocksByNum(ArrayList<Integer> blockNumberList, **int**... nodeIds);

• Request<BlockResponse> getBatchBlocksByNum(ArrayList<Integer> blockNumberList, **boolean** isPlain, **int**... nodeIds);

重载方法如下：

• Request<BlockResponse> getBatchBlocksByStrNum(ArrayList<String> blockNumberList, **int**... nodeIds);

• Request<BlockResponse> getBatchBlocksByStrNum(ArrayList<String> blockNumberList, **boolean** isPlain, **int**... nodeIds);

第六章. Node相关接口（NodeService）

NodeService接口用于获取节点信息。NodeService对象对应的响应类型如下：

• NodeResponse

分别对应的结构如下。

**BlockResponse**

通过result接收返回结果，result实际类型是内部类Node，可通过getResult()方法得到。

**• public** **class** NodeResponse **extends** Response {

•     **public** **class** Node {

•         **private** **int** id;

•         **private** String ip;

•         **private** String port;

•         **private** String namespace;

•         **private** String hash;

•         **private** String hostname;

•         **private** **boolean** isPrimary;

•         **private** **boolean** isvp;

•         **private** **int** status;

•         **private** **int** delay;

•     }

•     **private** JsonElement result;

• }

**TODO**

目前NodeService只支持一个服务，之后会增加更多类型的服务接口。

6.1 获取节点信息

参数：

• nodeIds 说明请求向哪些节点发送。

• Request<NodeResponse> getNodes(**int**... ids);

第七章. MQ相关接口(MQService)

MQService接口用于与RabbitMQ进行交互。由于开发时间较早，MQService对应的响应类型只有MQResponse一种，这与之前提到的接口都不太相同：

MQResponse接口结构如下：

**• public** **class** MQResponse **extends** Response {

•     **private** JsonElement result;

•     **public** List<String> getQueueNames();

•     **public** String getExchanger();

• }

7.1 通知MQ服务器正常工作

参数：

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> informNormal(**int**... nodeIds)

7.2 注册队列

参数：

• from 调用该接口的账户地址

• queueName 队列名称

• routingkeys 想要订阅的消息类型

• isVerbose 推送区块时是否推送交易列表，true表示是

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> registerQueue(String from, String queueName, List<String> routingkeys, Boolean isVerbose, **int**... nodeIds);

7.3 注销队列

参数：

• from 调用该接口的账户地址

• queueName 队列名称

• exchangerName exchanger 名称

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> unRegisterQueue(String from, String queueName, String exchangerName, **int**... nodeIds);

7.4 获取所有队列名称

参数：

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> getAllQueueNames(**int**... nodeIds);

7.5 获取所有exchanger名称

参数：

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> getExchangerName(**int**... nodeIds);

7.6 删除exchanger

参数：

• exchangerName exchanger名称

• nodeIds 说明请求向哪些节点发送

• Request<MQResponse> deleteExchanger(String exchangerName, **int**... nodeIds);

第八章. Radar相关接口（RadarService）

RadarService接口用于可视化监控合约，目前只有一个接口，对应的响应也只有RadarResponse。

8.1 监听合约

参数：

• sourceCode 要监听的合约的源代码

• contractAddress 要监听的合约的部署地址

• nodeIds 说明请求向哪些节点发送。

• Request<RadarResponse> listenContract(String sourceCode, String contractAddress, **int**... nodeIds);

第九章. ArchiveService相关接口

ArchiveService接口用于快照和归档相关工作，对应的响应类型如下：

• ArchiveResponse

• ArchiveFilterIdResponse

• ArchiveBoolResponse

分别对应的结构如下：

**ArchiveResponse**

通过result接收返回结果，result实际类型是内部类Archive，可通过getResult()方法得到。

**• public** **class** ArchiveResponse **extends** Response {

•     **private** **class** Archive {

•         **private** String height;

•         **private** String hash;

•         **private** String filterId;

•         **private** String merkleRoot;

•         **private** String date;

•         **private** String namespace;

•     }

•

•     **private** JsonElement result;

• }

**ArchiveFilterIdResponse**

通过result接收返回结果，result实际类型是String，可通过getResult()方法得到。

**• public** **class** ArchiveFilterIdResponse **extends** Response {

•     **private** String result;

• }

**ArchiveBoolResponse**

通过result接收返回结果，result实际类型是Boolean，可通过getResult()方法得到。

**• public** **class** ArchiveBoolResponse **extends** Response {

•     **private** Boolean result;

• }

9.1 制作快照

参数：

• blockNumber 区块号

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveFilterIdResponse> snapshot(BigInteger blockNumber, **int**... nodeIds);

重载方法如下：

• Request<ArchiveFilterIdResponse> snapshot(String blockNumber, **int**... nodeIds);

9.2 查询快照是否存在

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> querySnapshotExist(String filterId, **int**... nodeIds);

9.3 检查快照是否正确

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> checkSnapshot(String filterId, **int**... nodeIds);

9.4 删除快照

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> deleteSnapshot(String filterId, **int**... nodeIds);

9.5 列出所有快照

参数：

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveResponse> listSnapshot(**int**... nodeIds);

9.6 查看快照

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveResponse> readSnapshot(String filterId, **int**... nodeIds);

9.7 数据归档（预约归档）

参数：

• filterId 快照id

• sync 是否同步

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> archive(String filterId, **boolean** sync, **int**... nodeIds);

9.8 数据归档（直接归档）

参数：

• blkNumber 区块号

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> archiveNoPredict(BigInteger blkNumber, **int**... nodeIds);

9.9 恢复某归档数据

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> restore(String filterId, **boolean** sync, **int**... nodeIds);

9.10 恢复所有归档数据

参数：

• sync 是否同步

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> restoreAll(**boolean** sync, **int**... nodeIds

9.11 查询归档数据状态

参数：

• filterId 快照id

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveBoolResponse> queryArchive(String filterId, **int**... nodeIds);

9.12 查询所有待完成的快照请求

参数：

• nodeIds 说明请求向哪些节点发送

• Request<ArchiveResponse> pending(**int**... nodeIds);

附录

附录 A Solidity与Java的编码解码

类型对应

当使用LiteSDK编译solidity合约时，由于java和solidity本身类型的不兼容，所以在调用solidity方法传参数的时候需要对java类型进行相应的编码解码，LiteSDK内部的Abi类，与solidity的abi文件对应，用来提供solidity合约的函数入参、返回值等信息，方便我们对solidity类型和java类型做转换，目前Litesdk支持的对应类型如下：

**JAVA**

**SOLIDITY**

boolean/Boolean

bool

BigInteger

int、int8、int16……int256

BigInteger

uint、uint8、uint16……uint256

String

string

byte[]/Byte[]

bytes、bytes1、bytes2……bytes32

string

address

Array/List

array

编码

编码时需要提供以下信息：

• solidity合约对应的abi对象，

• 调用方法名

• 封装后的java参数

实现java与solidity之间的类型转换。（注：如果是部署需要提供bin文件，具体参照部署合约一节）

Abi对象

通过LiteSDK提供的FileUtil工具类读取文件内容得到abi字符串，并利用Abi类的fromJson方法生成封装的Abi对象，使用方法如下：

• InputStream abiIs = Thread.currentThread().getContextClassLoader().getResourceAsStream("xxx.abi");

• String abiStr = FileUtil.readFile(abiIs);

• Abi abi = Abi.fromJson(abiStr);

调用方法名

调用方法名需要按格式$(method\_name)(type1[,type2…])填，假如solidity的函数签名为：

• function TestUint(uint8 a) returns (uint8) {

•     **return** a;

• }

则我们提供的调用方法名为TestUint(uint8)，如果函数多个参数，则调用方法名的类型之间用“**,”**分隔。

封装的java参数

LiteSDK提供了FuncParams工具类封装需要转换成solidity类型的java参数，使用方法如下：

• FuncParams params = **new** FuncParams();

• // param 是类型对应表里对应的java参数

• params.addParams(param1);

• params.addParams(param2);

•

• // 构造交易时将构造好的FuncParams对象传进去

• Transaction transaction = **new** Transaction.EVMBuilder(account.getAddress()).invoke(contractAddress, <method\_name>, abi, params).build();

解码

解码与编码类似，需要提供Abi对象、方法名和编码的solidity结果，具体可见编码一节。

调用evm合约得到交易回执ReceiptResponse后，需要对solidity合约的返回值进行解析，使用方法如下：

• String ret = receiptResponse.getRet();

**• byte**[] fromHex = ByteUtil.fromHex(ret);

•

• // 通过abi的方法名解码，由于返回值可能有多个，所以解码得到的其实是一个List<?>，当中的每个对象

• // 对应一个返回值。如该例子返回值为 int256，在java中对应的是BigInter，所以对返回的decodeResult

• // 遍历强转为BigInteger

• List<?> decodeResult = abi.getFunction("TestInt(int256)").decodeResult(fromHex);

**• for** (Object result : decodeResult) {

•     System.out.println(result.getClass());

•     System.out.println(((BigInteger) result).toString());

• }

附录B 直接调用HVM合约方法的参数封装

直接调用HVM合约方法封装参数需要用到类InvokeDirectlyParams，

示例如下：

假设调用合约方法add(int a, int b)，传入参数(10，10)；

• // 构造函数传入想要调用的方法名

• InvokeDirectlyParams.ParamBuilder params = **new** InvokeDirectlyParams.ParamBuilder("add");

• // 方法addxxx分别构造不同类型的参数

• params.addint(10);

• params.addint(100);

• InvokeDirectlyParams.params.build();