Hyperchain Java SDK Document v4.1.21

Hyperchain Java SDK Document v4.1.21

第一章 前言

此sdk文档面向Hyperchain区块链平台的应用开发者，提供hyperchain Java SDK的使用指南。

为版本更替简洁起见，1.1.8版本后，hyperchain Java sdk的版本迭代为1.9版本。3.0版本和4.0版本统一合并为4.0版本。

第二章 接口使用流程示例

说明：为了更好的理解接口的使用，本示例将从创建账户、编译合约、部署合约、调用合约和返回值解析这个大致流程作介绍，具体详细接口可参考第三章SDK文档。

2.1 创建账户

首先初始化HyperchainAPI，sdk提供不同的初始化方法，具体可参考3.1部分文档。

例如：

```java
HyperchainAPI hyperchain = new HyperchainAPI();
```

Java

空参构造方式直接使用默认配置文件内容，具体可以查看3.1.1文档。

创建账户可创建国密、非国密类型账户，具体创建类型可参考3.9文档，返回账户公私钥json字符串，然后使用Account类型封装参数。

例如：

```java
String accountJson = HyperchainAPI.newAccountRawSM2();
Account account = new Account(accountJson);
```

Java

2.2 编译合约

创建合约文件，合约可分为Solidity和Java合约两种，编译合约可参考3.3.1文档。

首先编写合约文件，使用Utils的readFile(String path)读入合约文件内容(Utils介绍可查看3.10文档)，可上传编译合约，但是推荐在本地编译好合约后在部署。

远程编译获取编译结果：

```java
String contract = Utils.readFile("/Users/Desktop/Accumulator.sol");
CompileReturn compile = hyperchain.compileContract(contract);
```

Java

然后获取合约的bin和abi，本地编译Solidity合约也需获得bin和abi(本地编译需要solc编译器，可以使用npm install -g solc安装)。

```java
List<String> listBin = compile.getBin();
List<String> listAbi = compile.getAbi();
//获取bin
String bin = listBin.get(0);
//获取abi
String abi =listAbi.get(0);
```

Java

Java合约则在本地编译，编译方式查看3.3.1文档，然后使用Utils的 `getTargetBinFromPath(String path)` 将编译后的Java合约转为bin。

2.3 部署合约

根据合约的构造函数是否有参数，实例化合约部署使用不同的方法，具体部署接口查看3.2.2.2和3.2.2.3文档。对于有参数的构造函数，使用FuncParamReal将构造参数封装（ `Solidity`

合约，不同虚拟机合约参数构造方式不同，具体请看之后详细文档），在根据合约构造参数有无调用不同的实例化合约部署方法。

例如：

```
Transcation transaction = new Transaction(account.getAddress(), bin, false,
funcParamReal, VMType.EVM);
//签名
transaction.signWithSM2(accountJson, "");
```

Java

其中VMType默认使用EVM，若为Java合约，则需显式声明使用 `JVM` 。实例化部署后，使用

```
ReceiptReturn receiptReturn = hyperchain.deployContract(transaction);
```

Java

部署合约。

部署完成后可查看部署结果:

```
//部署合约交易哈希
String txHash = receiptReturn.getTxHash();
int code = receiptReturn.getCode();
//合约地址
String contractAddress = receiptReturn.getContractAddress();
```

Java

2.4 调用合约

调用合约需要封装方法名和参数：

- Solodity合约使用 `FunctionEncode` 的 `static encodeFunction(String functionName, FuncParamReal... funcParams)` 封装；
- Java合约使用 `FunctionEncode的static encodeFunctionJava(String functionName, String... funcParams)` 封装；

参数封装查看3.2.1文档，返回编码后的String，在实例化调用合约交易(3.2.2.4文档)，之后调用合约方法(3.3.3文档)，获得回执。

例如：

```
FuncParamReal param1 = new FuncParamReal("uint32", new BigInteger("3"));
FuncParamReal param2 = new FuncParamReal("uint32", new BigInteger("2"));
//参数封装
String payloadWithParmas = FunctionEncode.encodeFunction("add", param1, param2);
Transaction transactionWithParmas = new Transaction(account.getAddress(),
contractAddress, payloadWithParmas, false, VMType.EVM);
transactionWithParmas.signWithSM2(accountJson, "");
//调用
ReceiptReturn receiptReturn = hyperchain.invokeContract(transactionWithParmas);
```

Java

2.5 返回值解析

得到返回值结果后，获得状态码可以判断是否调用成功，若调用成功，解析返回值可看到调用之后的结果。

例如：

```
if (!receiptReturn.isSuccess()) {
    //error
}
//状态码
code = receiptReturnWithParams.getCode();
//未解码结果
String rawReturnWithParams = receiptReturn.getRet();
//解码
String decodedResultWithParams = FunctionDecode.resultDecode("add", abi,
rawReturnWithParams);
```

Java

2.6 完整示例(Solidity合约)

```
/**
 * 一个简单的完整流程测试Demo - 合约构造函数无入参（创建账户、编译合约、部署合约、调用合约、返回值解析）
 */
//模拟日志打印
private void log(String s) {
    System.out.println("HyperchainAPITest ====== " + s);
}
HyperchainAPI hyperchain = new HyperchainAPI();
//创建账户
String accountJson = HyperchainAPI.newAccountRawSM2();
Account account = new Account(accountJson);
log("创建账户 accountJson : " + accountJson);
//读取合约
String contract = Utils.readFile("solidity.source/Accumulator.sol");
//编译合约
CompileReturn compile = hyperchain.compileContract(contract);
List<String> listBin = compile.getBin();
log("listBin    :" + listBin);
List<String> listAbi = compile.getAbi();
log("listAbi    :" + listAbi);
//获取bin
String bin = listBin.get(0);
log("bin    :" + bin);
//获取abi
String abi = listAbi.get(0);
log("abi    :" + abi);
//封装交易
Transaction transaction = new Transaction(account.getAddress(), bin, false);
//签名
transaction.signWithSM2(accountJson, "");
//部署合约
ReceiptReturn receiptReturn = hyperchain.deployContract(transaction);
//查询部署结果
int code = receiptReturn.getRawcode();
log("部署结果: " + code);
String contractAddress = receiptReturn.getContractAddress();
log("合约地址: " + contractAddress);
//调用合约(有参数):计算 3 + 2
FuncParamReal param1 = new FuncParamReal("uint32", new BigInteger("3"));
FuncParamReal param2 = new FuncParamReal("uint32", new BigInteger("2"));
//方法编码
String payloadWithParams = FunctionEncode.encodeFunction("add", param1, param2);
//实例化调用合约
Transaction transactionWithParmas = new Transaction(account.getAddress(),
contractAddress, payloadWithParams, false);
transactionWithParmas.signWithSM2(accountJson, "");
//同步调用合约
ReceiptReturn receiptReturnWithParams =
hyperchain.invokeContract(transactionWithParmas);
//获取与解析返回值
String rawReturnWithParams;
if (!receiptReturnWithParams.isSuccess()) {
    log("合约方法调用失败,code:" + receiptReturn.getCode());
}
code = receiptReturnWithParams.getRawcode();
log("调用合约结果状态: " + code);
rawReturnWithParams = receiptReturnWithParams.getRet();
log("调用合约结果（未解码）: " + rawReturnWithParams);
String decodedResultWithParams = FunctionDecode.resultDecode("add", abi,
rawReturnWithParams);
log("调用合约结果（解码后）" + decodedResultWithParams);
//调用合约(无参数): 查看返回值
String payloadWithoutParams = FunctionEncode.encodeFunction("getSum");
Transaction transaction1 = new Transaction(account.getAddress(), contractAddress,
payloadWithoutParams, false);
transaction1.signWithSM2(accountJson, "");
ReceiptReturn receiptReturnWithoutParams =
hyperchain.invokeContract(transaction1);
String rawReturnWithoutParams = receiptReturnWithoutParams.getRet();
log("调用合约结果（未解码）: " + rawReturnWithoutParams);
String decodedResult = FunctionDecode.resultDecode("getSum", abi,
rawReturnWithoutParams);
log("调用合约结果（解码后）" + decodedResult);
```

Java

2.7 完整示例(Java合约)

```
/**
 * 一个简单的完整流程测试Demo - 合约构造函数无入参（创建账户、编译合约、部署合约、调用合约、返回值解析）
```

```
 */
//模拟日志打印
private void log(String s) {
    System.out.println("HyperchainAPITest ====== " + s);
}
HyperchainAPI hyperchain = new HyperchainAPI();
//创建账户
String accountJson = HyperchainAPI.newAccountRawSM2();
Account account = new Account(accountJson);
log("创建账户 accountJson : " + accountJson);
//编译合约(本地编译)
String homePath = System.getenv("HOME");
String sdkPath = homePath + "/.m2/repository";
String sourceDir = "src/test/resources/javaContractExample/source1";
String destDir = "src/test/resources/javaContractExample/contract01";
String contractPath = HyperchainAPI.compileContractJava(sdkPath, sourceDir,
destDir);
log("合约编译输出路径 contractPath : " + contractPath);
//编译成功后，在编译路径，即destDir下，创建contract.properties文件，在文件//中输入两个配置，
分别为contract.name和main.class，其中contract.name配置//为你创建的合约名称，main.class配
置为合约主类的全类名。
//例如:
//contract.name=AccountSum
//main.class=cn.hyperchain.jcee.contract.examples.sb.src.AccountSum
//部署Java合约
String target = Utils.getTargetBinFromPath("javaContractExample/contract01");
Transaction transaction = new Transaction(account.getAddress(), target, false,
VMType.JVM);
transaction.signWithSM2(accountJson, "");
ReceiptReturn receiptReturn = hyperchain.deployContract(transaction);
//查询部署结果
int code = receiptReturn.getRawcode();
log("部署结果: " + code);
String contractAddress = receiptReturn.getContractAddress();
log("合约地址: " + contractAddress);
//调用合约
String input = FunctionEncode.encodeFunctionJava("invoke", "issue", "a", "100");
Transaction transaction2 = new Transaction(account.getAddress(), contractAddress,
input, false, VMType.JVM);
transaction2.signWithSM2(accountJson, "");
ReceiptReturn invokeReceipt = hyperchain.invokeContract(transaction2);
if (! invokeReceipt.isSuccess()) {
    log("调用错误");
}
log("调用状态 code : " + invokeReceipt.getRawcode());
log("调用结果 " + invokeReceipt.getRet());
```

Java

第三章 SDK文档

3.1 初始化

3.1.1 配置文件说明

配置文件hpc.properties

```
#Hyperchain Nodes IP Ports
node = {"nodes":["localhost","localhost","localhost","localhost"]}
# JsonRpc connect port
jsonRpcPort = [8081, 8082, 8083, 8084]
# webSocket connect port
webSocketPort = [10001, 10002, 10003, 10004]
#Namespace
namespace = global
#重发次数
resendTime = 10
#第一次轮询时间间隔 unit /ms
firstPollingInterval = 100
#发送一次,第一次轮询的次数
firstPollingTimes = 10
#第二次轮询时间间隔 unit /ms
secondPollingInterval = 1000
#发送一次,第二次轮询的次数
secondPollingTimes = 10
#Send Tcert during the request or not
SendTcert = false
#if sendTcert is true , you should add follow path.
ecertPath = certs/sdkcert.cert
ecertPriPath = certs/sdkcert.priv
uniquePrivPath = certs/unique.priv
```

```
uniquePubPath = certs/unique.pub
#发送重新连接请求间隔(/ms)
reConnectTime = 10000
#Use Https
https = true
#If https is true, you shoule add follow properties
tlsca = certs/tls/tlsca.ca
tlspeerCert = certs/tls/tls_peer.cert
tlspeerPriv = certs/tls/tls_peer.priv
#CFCA开关
CFCA = false
#httpManager中的httpClient超时时间(/s)
readTimeoutInHttpRm = 20
writeTimeoutInHttpRm = 20
connectTimeoutInHttpRm = 20
#websocket中的httpClient超时时间(/s)
readTimeoutInWebsocket = 20
writeTimeoutInWebsocket = 20
connectTimeoutInWebsocket = 20
#websocket的心跳包时间(/s) 注意该值需要小于上一小节中readTimeoutInWebsocket的值
heartbeatInterval = 10
```

JavaScript

HyperchainAPI对象用于提供各种接口方法，其中默认配置文件位于classpath目录下。

说明:

- `node` 表示平台各节点的IP： `jsonRpcPort` 也就是平台发送消息的端口， `webSocketPort` 为事件订阅连接的端口，他们分别对应着 `node` 中配置的IP，需要说明的是，SDK接口中需要指定节点id的时候，**id都从1开始计数，且对应关系为配置文件中node的顺序**，而非平台节点顺序；
- `namespace` 为初始化hyperchain对象时传入的namespace；
- `resendTime` 参数表示重发次数； `pollingInterval` 表示轮训去获取交易的时间间隔（分为第一次和第二次），单位为毫秒； `pollingTimes` 表示发送一次交易的轮训次数（分为第一次和第二次）；**需要注意的是轮询结束没有查到回执，需要查看具体的平台信息，排除是否是限流、响应慢等因素影响无法查询到回执**
- `SendTcert` 表示是否打开TCert认证开关， `ecertPath` 表示sdkCert的路径， `ecertPriPath` 表示sdkCert私钥的路径，该证书的作用是SDK的准入， `uniquePrivPath` 表示用户私钥路径， `uniquePubPath` 表示用户公钥路径，用于向平台申请TCert证书；
- `reConnectTime` 表示当SDK发现当节点挂掉后，发送试探请求的间隔时间，单位为毫秒；
- `https` 开启后则请求都会使用https来发送， `tlsca` 表示客户端需要验证的服务器ca， `tlspeerCert` 表示服务器需要验证的SDK的ca的公钥， `tlspeerPriv` 则为对应的ca公钥的私钥；
- `CFCA` 开关表示是否开启CFCA认证，需配合平台使用；
- `readTimeoutInHttpRm` 、 `writeTimeoutInHttpRm` 、 `connectTimeoutInHttpRm` 表示HttpRequestManager中HttpClient的相应的超时时间；
- `readTimeoutInWebsocket` 、 `writeTimeoutInWebsocket` 、 `connectTimeoutInWebsocket` 表示Websocket中HttpClient的相应的超时时间； `heartbeatInterval` 表示Websocket中心跳包的间隔时间。

3.1.2 初始化Hyperchain对象(带路径)

```
HyperchainAPI hyperchain = new HyperchainAPI(path);
```

> 注意：传入的路径即为hpc.properties文件的路径。

hpc.properties的文件格式如3.1.1所示相同。

```
HyperchainAPI hyperchain = new HyperchainAPI("./hpc.properties");
```

Java

3.1.3 初始化Hyperchain对象(带ApiProperties)

```
HyperchainAPI hyperchain = new HyperchainAPI(apiProperities);
```

```
//使用给定资源初始化
ApiProperities初始化方法
```

```
ApiProperities()      //带部分默认值，默认值与3.1.1所示的hpc.properties一致
ApiProperities(String properitiesPath) //路径为hpc.properties路径
```

JavaScript

另外对于apiProperties，可以利用set方法进行赋值，可赋值变量有：

```
String node;
String jsonRpcPort;
String webSocketPort;
int resendTimes;
int firstPollingInterval;
int firstPollingTimes;
int secondPollingInterval;
int secondPollingTimes;
boolean sendTcert;
String ecertPath;
String ecertPriPath;
String uniquePrivPath;
String uniqurPubPath;
String reConnectTime;
boolean isHttps;
String tlsca;
String tlspeerCert;
String tlspeerPriv;
boolean useCFCA;
int readTimeoutInHttpRm;
int writeTimeoutInHttpRm;
int connectTimeoutInHttpRm;
int readTimeoutInWebsocket;
int writeTimeoutInWebsocket;
int connectTimeoutInWebsocket;
int heartbeatInterval;
```

Java

其中参数说明同之前的配置文件，配置值和配置文件相同。

```
ApiProperities apiProperities = new ApiProperities();
Hyperchain hyperchain = new HyperchainAPI(apiProperities)
```

Java

> 注意：请确认Hyperchian平台的TCert认证开关是否关闭。请配合Hyperchain平台配置使用。

3.1.4 初始化Hyperchain对象(无参数)

无参构造函数，使用classpath文件下的默认配置，配置内容同3.1.1

注意：考虑到上层应用覆盖SDK原配置文件路径的问题，推荐使用带路径的方式，避免出现配置文件找不到的问题。

```
HyperchainAPI hyperchain = new HyperchainAPI();
```

Java

3.1.5 初始化Hyperchain对象(带ecert和unique路径)

`HyperchainAPI(String ecertPath, String ecertPriPath, String uniquePrivPath, String uniquePubPath) throws Exception`

使用自定义的ecert和unqiue路径，其余参数使用默认配置，配置内容同3.1.1

```
HyperchainAPI hyperchain = new HyperchainAPI("./ecert", "./ecertPri",
"./uniquePriv", "./uniquePub");
```

Java

3.1.6 负载均衡

在实例化HyperchainAPI对象的时候，将会先读取配置文件所有节点的信息，每次发送请求的时候都会随机选择一个节点向其发送请求，如果发现向某个节点请求失败，在下一次随机选取节点的时候就不会再选取这个节点发送消息。SDK在发现请求失败的同时会定时监测此节点的状态，如果发现节点恢复正常，下次便可朝此节点发送请求。

3.1.7 设置namespace

`public void setNamespace(String namespace)`

该方法可以修改HyperchainAPI实例的namespace

例：

```
hyperchain.setNamespace("global");
```

Java

3.1.8 获取单例HyperchainAPI实例

```
public static HyperchainAPI getInstance() throws Exception
```

该方法可以获取HyperchainAPI的单例实例（需要配置文件hpc.properties），通过懒加载的方式，同时也是**推荐**的获取HyperchainAPI的方式。

```
HyperchainAPI.getInstance();
```

Java

3.2 Transacton相关接口

3.2.1 交易参数的封装

SDK对于发送交易的参数都是需要编码的，因此在构造交易的时候需要对相应参数调用相应的SDK编码方法。

3.2.1.1 Solidity合约调用参数封装

```
public static String encodeFunction(String functionName, FuncParamReal... funcParams) throws FunctionParamException
```

```
/**
 * 函数输入编码(参数为FuncParamReal类型)
 * @param functionName 函数名称
 * @param funcParams Solidity参数
 * @return 编码之后的string
 */
例子：
FuncParamReal param1 = new FuncParamReal("uint32", new BigInteger("1"));
FuncParamReal param2 = new FuncParamReal("uint32", new BigInteger("2"));
String input = FunctionEncode.encodeFunction("add", param1, param2);
//调用合约时此input即可作为Transaction的payload参数
```

Java

说明：采用FuncParamReal作为合约参数的封装，需要知道Solidity参数对应的具体Java类型，例如：uint*对应BigInteger，bytes对应byte[]。

3.2.1.2 Java合约生成部署bin

```
public static String getTargetBinFromPath(String path) throws IOException
```

```
/**
 * 将Java合约转为target bin，然后直接放入transaction中的payload中发送
 * @param path 编译好的Java合约的路径
 * @return Target bin
 * @throws IOException error
 */
例子：
String contractPath = "src/test/resources/javaContractExample/contract01";
String target = Utils.getTargetBinFromPath(contractPath);
Transaction transaction = new Transaction(TEST_FROM, target,false, VMType.JVM);
```

Java

3.2.1.3 Java合约调用参数封装

```
public static String encodeFunctionJava(String functionName, String... funcParams)
```

```
/**
 * 函数输入编码，转换为可发送的形式
 * @param functionName 函数名称
 * @param funcParams 合约方法参数
 * @return 编码之后的String
 */
例子：
```

```
String input = FunctionEncode.encodeFunctionJava("invoke","issue","a","100");
//调用合约时此input即可作为Transaction的payload参数
```

Java

3.2.2 实例化交易

3.2.2.1 实例化普通交易

普通交易主要是用于普通转账：

`public Transaction(String from, String to, long value, boolean simulate)`

```
/**java
 *
 * @param from from账户地址
 * @param to to账户地址
 * @param value 转账金额
 * @param simulate 是否共识
 */
 // 以address1: 向address2 转账53单位值为例：
Transaction mytransaction = new Transaction(address1,address2,53L,true);
```

Java

3.2.2.2 实例化合约部署交易(无构造参数)

`public Transaction(String from, String payload, boolean simulate, VMType type)`

Java

3.2.2.3 实例化合约部署交易(有构造参数)

`public Transaction(String from, String payload, boolean simulate, VMType type, FuncParamReal... params) throws FunctionParamException`

注意：Java合约不存在有参构造函数。

```
/**
 * 部署智能合约，构造函数无参
 * @param from from账户地址
 * @param payload 智能合约bin
 * @param simulate 是否共识
 * @param type 只适用于evm合约
 * @param params 智能合约构造函数参数
 * @throws FunctionParamException 参数非法异常
 */
//例子：
//以构造函数参数为int32类型，值为8为例
//构造参数对象
FuncParamReal funcparam = new FuncParamReal(“int32”, new BigInteger("8"));
Transaction mytransaction = new Transaction(fromAddress, bin, true, VMType.EVM,
funcparam);
```

Java

3.2.2.4 实例化调用合约交易

`public Transaction(String from, String to, String payload,boolean simulate,VMType type) throws TxException`

```
/**
 * 调用合约
 * @param from 源账户
 * @param to 目的账户
 * @param payload 调用的方法信息
 * @param simulate 是否共识
 * @param type 合约类型
 * @throws TxException error
 */
示例：
String payload = FunctionEncode.encodeFunctionJava(functionName,params);
Transaction transaction2 = new Transaction(TEST_FROM, contractAddress, payload,
false, VMType.JVM)
```

Java

3.2.2.5 实例化合约管理交易

`public Transaction(String from, String to, String payload, int opcode, VMType type) throws TxException`

```
/**
 * 升级合约构造Transaction opcode为1
 *
 * @param from 账户地址
 * @param to 目的合约地址
 * @param payload 内容（升级合约为合约bin）
 * @param opcode 类型（1为升级合约，2为冻结合约，3为解冻合约）
 * @param type 合约类型
 * @throws TxException error
 */
示例：
Transaction transaction = new Transaction(account.getAddress(), contractAddress,
targetBinUpgrade, 1, VMType.EVM);
```

Java

`public Transaction(String from, String to, int opcode, VMType type) throws TxException`

```
/**
 * 冻结、解冻合约构造Transaction opcode为2或3
 *
 * @param from 账户地址
 * @param to 目的合约地址
 * @param opcode 类型（1为升级合约，2为冻结合约，3为解冻合约）
 * @param type 合约类型
 * @throws TxException error
 */
示例：
Transaction transaction = new Transaction(account.getAddress(), contractAddress,
2, VMType.EVM);
Transaction transaction = new Transaction(account.getAddress(), contractAddress,
3, VMType.EVM);
```

Java

> 注意：以上五个实例化构造函数均有一个重载的构造函数，参数列表为在原本基础上新加一个extra字段，例如：

`public Transaction(String from, String to, long value, boolean simulate, VMType type, String extra)`

```
例子：
//extra 为保留字段，用户可使用也可不使用
String extra = "123";
Transaction mytransaction = new Transaction(address1,address2,53L,true,VMType.EVM,
extra);
```

Java

3.2.2.6 Simulate说明

在新建Transaction的时候，有一个字段为"simulate"，这个字段代表这条交易是否共识，其使用方式如下：

注意：模拟交易直接返回结果，不需要轮询查询，同时该模拟交易结果只会存在于执行交易的节点。

```
例子：
//模拟调用
Account account = new Account(TEST_ACCOUNT_JSON);
Transaction transaction = new Transaction(account.getAddress(), TEST_TO, 53L,
true);
transaction.signWithSM2(TEST_ACCOUNT_JSON, "123");
ReceiptReturn result = hyperchain.sendTx(transaction);
```

Java

> 注意：此改动适用于4.0.13及以上版本！

3.2.3 交易签名

构造完成之后的交易需要进行签名，可以用如下方法进行签名

3.2.3.1 利用加密之后私钥字符串和加密密钥进行签名(非国密)

```java
/**
*@param accountJSON 加密后的密钥对JSON字符串
*@param passwd 加密密钥对密码
*/
// 初始化一笔普通交易
Transaction transaction = new Transaction(TEST_FROM, TEST_TO, 53L, false,
VMType.EVM);
transaction.sign(TEST_PRIJSON, TEST_PASSWD);
// 根据账户加密与否，可以将accountJson转换为ECPriv后签名
ECKey ecKey = ECKey.fromPrivateJSON(accountJSON, passwd);
transaction.sign(ecKey);
```

Java

3.2.3.2 利用加密之后私钥字符串和加密密钥进行签名(国密)

```java
/**
*@param accountJSON 加密后的密钥对JSON字符串
*@param passwd 加密密钥对密码
*/
// 初始化一笔普通交易
Transaction transaction = new Transaction(TEST_FROM, TEST_TO, 53L, false,
VMType.EVM);
transaction.signWithSM2(TEST_PRIJSON, TEST_PASSWD);
// 若账户没有加密，则签名时不需要密码
transaction.signWithSM2(TEST_PRIJSON);
```

Java

3.2.3.3 利用外部签名字符串进行签名

```java
/**
* @param signature 外部签名字符串
* 适用于构造交易和发送交易不在同一客户端，签名需要第三方工具
*/
Transaction transaction = new Transaction(TEST_FROM, TEST_TO, 53L, false,
VMType.EVM);
transaction.sign(signature);
```

Java

3.2.4 取得区块中的所有交易信息

`SingleValueReturn getTx(String start, String end)`

```java
/**
* 取得区块中的所有交易信息
*
* @param start 查询开始区块号
* @param end 查询结束区块号
* @return 标准格式返回值
* 如果getTx 的第二个参数为空,则会设置为"latest"
*/
SingleValueReturn result = hyperchain.getTx("1", "2");
result.getResult();
```

Java

3.2.5 查询所有非法交易

`SingleValueReturn getDiscardTx()`

```java
/**
* 查询所有非法交易
*
* @return 返回交易json列表
*/
SingleValueReturn result = hyperchain.getDiscardTx();
result.getResult();
```

Java

3.2.6 查询交易(通过交易hash)

`SingleValueReturn getTxByHash(String txHash)`

```java
/**
* 查询交易 by TxHash
*
* @param txHash 交易Hash
* @return 单值返回值 返回交易信息json
```

```
*/
SingleValueReturn result = hyperchain.getTxByHash(txHash);
result.getResult();
```

Java

3.2.7 批量查询交易(通过交易hash列表)

`SingleValueReturn getBatchTxByHash(ArrayList<String> txHashList)`

```
/**
* 查询交易 by txHash 批量查询
* @param txHashList hashList
* @return json string返回值list
*/
例:
ArrayList<String> txHashList = new ArrayList<String>();
txHashList.add(hash1);
txHashList.add(hash2);
SingleValueReturn singleValueReturn = hyperchain.getBatchTxByHash(txHashList);
String batchResult = singleValueReturn.getResult();
JSONArray batchArray = JSONArray.fromObject(batchResult);
Assert.assertEquals(2, batchArray.size());
```

Java

3.2.8 查询交易(通过区块hash和交易index)

`SingleValueReturn getTxByBlkHashAndIdx(String blkHash, int index)`

```
/**
* 查询交易 by blockHash and TxIndex
* @param blkHash 块hash
* @param index 交易序号
* @return 返回交易信息json
*/
SingleValueReturn result = hyperchain.getTxByBlkHashAndIdx(blkHash,index);
result.getResult();
```

Java

3.2.9 查询交易(通过区块号和交易index)

区块号为int类型

`SingleValueReturn getTxByBlkNumAndIdx(int blkNumber, int index)`

区块号为String类型

`SingleValueReturn getTxByBlkNumAndIdx(String blkNumber, int index)`

```
/**
* 2.5 查询交易 by block number and TxIndex
* @param blkNumber 块号
* @param index 交易索引 或者 数字字符串
* @return返回交易信息json
*/
SingleValueReturn result = hyperchain. getTxByBlkNumAndIdx(blkNumber, index);
result.getResult();
```

Java

3.2.10 发送转账交易(同步)

`SingleValueReturn sendTx (Transaction transaction) throws Exception`

```
/**
* 同步发送交易（单一参数）
*
* @param transaction 需要发送的交易（需要签名）
* @return 返回结果
*/
Transaction transaction = new Transaction(TEST_FROM, TEST_TO, 53L, false);
transaction.sign(TEST_PRIJSON, TEST_PASSWD);
ReceiptReturn result = hyperchain. sendTx(transaction);
result.getRet();
// 或者可以指定节点发送，节点id从1开始，顺序同配置文件中一致
int id = 1;
ReceiptReturn result = hyperchain. sendTx(transaction, id);
```

Java

3.2.11 批量发送交易(同步，指定每笔交易的类型)

`ArrayList<ReceiptReturn> sendBatch(ArrayList<Transaction> transactions, ArrayList<MethodType> methodTypes) throws InterruptedException, ParamErrorException`

注意：该接口将多个Transaction同步发送给节点，但是对于节点来说为同步执行批量交易。

```
/**
 * 同步批量发送交易
 *
 * @param transactions 待发送交易列表
 * @param methodTypes  交易对应的方法
 * @return 交易结果
 */
Transaction transaction = new Transaction(TEST_FROM, Wrong_TEST_TO, 53L, false,
VMType.EVM);
transaction.sign(TEST_PRIJSON, TEST_PASSWD);
List<Transaction> transactions = new ArrayList<Transaction>();
List<MethodType> methodTypes = new ArrayList<MethodType>();
Transactions.add(transaction);
methodType.add(MethodType. INVOKECONTRACT);
ArrayList<ReceiptReturn> results = hyperchain. sendBatch(transactions,
methodTypes);
results.get(0).getResult();
```

Java

3.2.12 发送转账交易(异步)

异步方式有回调函数的形式和future的模式，可根据不同的需求来使用不同的方式：

1. 回调函数：

`void sendAysncTx(final Transaction transaction, final AsyncHandler callBack)`

```
/**
* 发送交易(带签名，异步)
* @param transaction 交易结构体
* @param AsyncHandler 异步回调，需要实现此接口
*/
Transaction transaction = new Transaction(TEST_FROM, Wrong_TEST_TO, 53L, false,
VMType.EVM);
transaction.sign(TEST_PRIJSON, TEST_PASSWD);
AsyncHandlerTest handler = new AsyncHandlerTest();
hyperchain.sendAysncTx(transaction, handler);
System.out.println(handler.result);
//注意：AsyncHandlerTest类的定义
public class AsyncHandlerTest implements AsyncHandler{
    public String result = "";
    @Override
    public void onSuccess(StdReturn ret) {
        result = ret.getResult();
    }
    @Override
    public void onFailed(StdReturn ret) {
        result = "failed";
    }
}
```

Java

1. future模式：

`FutureTask sendAysncTx(final Transaction transaction)`

```
/**
* 发送交易(带签名，异步，future实现)
* @param transaction 交易结构体
* @return FutureTask 异步调用
*/
Transaction transaction = new Transaction(account.getAddress(),
account1.getAddress(), 21L, false);
transaction.signWithSM2(TEST_ACCOUNT_JSON, TEST_PASSWD);
FutureTask<ReceiptReturn> futureTask = hyperchain.sendAysncTx(transaction);
String txHash = futureTask.get().getTxHash();
System.out.println(txHash);
```

Java

3.2.13 查询指定区块中交易平均处理时间

区块号查询支持int或者String类型

`SingleValueReturn getTxAvgTimeByBlkNum(int from,int to)`

`SingleValueReturn getTxAvgTimeByBlkNum(String from, String to)`

```
/**
* 查询指定区块中交易平均处理时间
* @param from 启始区块编号
* @param to 结束区块编号
* @return 返回平均处理时间,16进制（ns）
*/
SingleValueResult result = hyperchain. getTxAvgTimeByBlkNum (from, to);
result.getResult();
```

Java

3.2.14 查询指定交易中的回执信息

`ReceiptReturn getTransactionReceipt(String TxHash)`

```
/**
* 查询指定交易中的收据信息
* @param TxHash 交易hash
* @return ReceiptReturn Receipt返回
*/
ReceiptReturn result =
hyperchain.getTransactionReceipt("0x25cc67f7cb5c5393f419c2ea2be8ac8736826a64fcda5b
1afef2694f8712032e");
System.out.println(result.getCode());
System.out.println(result.getContractAddress()); //合约地址
System.out.println(result.getRet()); //调用合约返回值
// 或者可以指定节点发送，节点id从1开始，顺序同配置文件中一致
ReceiptReturn result =
hyperchain.getTransactionReceipt("0x25cc67f7cb5c5393f419c2ea2be8ac8736826a64fcda5b
1afef2694f8712032e", id);
```

Java

3.2.15 批量查询指定交易中的收据信息

`ArrayList<ReceiptReturn> getBatchTxReceipt(ArrayList<String> txHashLis t)`

若某一条交易hash查询不存在，则该回执会以error形式返回在回执中

```
/**
* 查询指定交易中的收据信息 批量查询
*
* @param txHashList 交易hash list
* @return 标准格式返回值
*/
例：
ArrayList<String> txHashList = new ArrayList<String>();
txHashList.add(hash1);
ArrayList<ReceiptReturn> receiptReturn = hyperchain.getBatchTxReceipt(txHashList);
Assert.assertEquals(1, receiptReturn.size());
for(ReceiptReturn re : receiptReturn){
      Assert.assertEquals(66, re.getTxHash().length());
}
```

Java

3.2.16 查询区块间交易量

`public SingleValueReturn getTxCountByContractAddr(String from, String to, String contractAddress, Boolean txExtra)`

```
/**
 * 查询区块间交易量
 *
 * @param from            起始区块
 * @param to              终止区块
 * @param contractAddress address
 * @param txExtra         have extra or not, false default
 * @return string
 */
BlockReturn block = hyperchain.getLatestBlock();
```

```
SingleValueReturn result = hyperchain.getTxCountByContractAddr("1",
"latest","0x8cba1a4db81853e62c72b2bc11e3d0e90c111daa",false);
System.out.println(result.getResult());
```

Java

3.2.17 查询区块交易数量

`SingleValueReturn getBlkTxCountByHash(String blkHash)`

```
/**
* 查询区块交易数量
* @param blkHash 区块hash
* @return 交易数目 16进制表示
*/
SingleValueReturn result = hyperchain. getBlkTxCountByHash(blkHash);
result.getResult();
```

Java

3.2.18 查询链上所有交易量

`SingleValueReturn getTxCount()`

```
/**
* 查询链上所有交易量
* @return 返回相应的json数据
*/
SingleValueReturn result = hyperchain.getTxCount();
result.getResult();
```

Java

3.2.19 根据时间戳查询交易

时间戳支持BigInteger或者String类型

`SingleValueReturn getTxByTime(BigInteger startTime, BigInteger endTime)`

`SingleValueReturn getTxByTime(String startTime, String endTime)`

```
/**
 * 根据时间查询Transaction
 * @param startTime 开始时间戳 ns
 * @param endTime 结束时间戳 ns
 * @return 返回txs 的json
 */
SingleValueReturn result = hyperchain.getTxByTime(new BigInteger("1"), new
BigInteger("1778959217012956575"));
JSONArray retArray = JSONArray.fromObject(result.getResult());
JSONObject obj = JSONObject.fromObject(retArray.get(0));
```

Java

3.2.20 查询下一页交易

`public SingleValueReturn getNextPageTxs(String blkNum, int txIndex, String minBlkNum, String maxBlkNum, int separated, int pageSize,boolean containCurrent, String address)`

```
/**
 * 查询下一页交易
 *
 * @param blkNum          从该区块开始计数
 * @param txIndex         起始交易在blkNumber号区块的位置偏移量
 * @param minBlkNum       截止计数的最小区块号
 * @param maxBlkNum       截止计数的最大区块号
 * @param separated       表示要跳过的交易条数（一般用于跳页查询）
 * @param pageSize        表示要返回的交易条数
 * @param 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
 * @param address         合约地址
 * @return TransactionResult 数组
 */
例：
BlockReturn blockReturn = hyperchain.getLatestBlock();
BigInteger maxBlockNum = new BigInteger(blockReturn.getNumber().substring(2), 16);
BigInteger minBlockNum = new BigInteger("1");
Integer blockNum = Integer.parseInt(blockReturn.getNumber().substring(2), 16);
blockNum = blockNum - 50 > 0 ? blockNum - 50 : 1;
BigInteger blockNumIndex = new BigInteger(blockNum + "");
```

```
StdReturn result = hyperchain.getNextPageTxs(blockNumIndex.toString(), 0,
minBlockNum.toString(), maxBlockNum.toString(), 0, 2, false, TEST_TO);
Assert.assertEquals(2, JSONArray.fromObject(result.getResult()).size());
```

Java

3.2.21 查询上一页交易

`public SingleValueReturn getPrevPageTxs(String blkNum, int txIndex, String minBlkNum, String maxBlkNum, int separated, int pageSize,boolean containCurrent, String address)`

```
/**
* 查询上一页交易
*
* @param blkNum          从该区块开始计数
* @param txIndex         起始交易在blkNumber号区块的位置偏移量
* @param minBlkNum       截止计数的最小区块号
* @param maxBlkNum       截止计数的最大区块号
* @param separated       表示要跳过的交易条数（一般用于跳页查询）
* @param pageSize        表示要返回的交易条数
* @param 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
* @param address         合约地址
* @return TransactionResult 数组
*/
例：
BlockReturn blockReturn = hyperchain.getLatestBlock();
BigInteger blockNum = new BigInteger(blockReturn.getNumber().substring(2), 16);
BigInteger minBlockNum = new BigInteger("1");
StdReturn result = hyperchain.getPrevPageTxs(blockNum.toString(), 0,
minBlockNum.toString(),blockNum.toString(), 0, 1, false, TEST_TO);
Assert.assertEquals(1, JSONArray.fromObject(result.getResult()).size())
```

Java

3.2.22 本地获取交易hash

`public String getTransactionHash(long gasLimit)`

```
/**
* 本地计算交易的哈希值
*
* @param gasLimit 该参数请与hyperchain配置文件中gas配置项值一致，默认数值为Transaction.DEFAULT_GAS_LIMIT
*/
例：
Transaction transaction = new Transaction(account.getAddress(), TEST_TO, 53L,
false);
transaction.signWithSM2(TEST_ACCOUNT_JSON, TEST_PASSWD);
transaction.getTransactionHash(Transaction.DEFAULT_GAS_LIMIT));
```

Java

3.2.23 查询区块交易数量

`public SingleValueReturn getBlkTxCountByNum(String blkNum)`

```
/**
* 查询区块交易数量
*
* @param blkNum
* @return 数目 16进制表示
*/
BlockReturn blockReturn = hyperchain.getLatestBlock();
SingleValueReturn countReturn =
hyperchain.getBlkTxCountByNum(blockReturn.getNumber());
Assert.assertEquals(0, countReturn.getRawcode());
```

Java

3.2.24 获取交易签名哈希

`public SingleValueReturn getSignHash(Transaction transaction)`

```
/**
* 获取交易签名哈希
*
* @param transaction
*/
Account account = new Account(TEST_ACCOUNT_JSON);
```

```
Transaction transaction = new Transaction(account.getAddress(), TEST_TO, 53L,
false);
transaction.signWithSM2(TEST_ACCOUNT_JSON, TEST_PASSWD);
SingleValueReturn singleValueReturn = hyperchain.getSignHash(transaction);
Assert.assertEquals(true, singleValueReturn.getResult().startsWith("0x"));
```

Java

3.2.25 查询指定时间区间内的非法交易

`public SingleValueReturn getDiscardTxByTime(BigInteger startTime, BigInteger endTime)`

```
/**
* 获取交易签名哈希
*
* @param startTime 开始时间戳 ns
* @param endTime   结束时间戳 ns
*/
hyperchain.setRequestNode(2);
SingleValueReturn result = hyperchain.getDiscardTxByTime(new BigInteger("1"), new
BigInteger("1778959217012956575"));
System.out.println(result);
hyperchain.unSetRequestNode();
```

Java

3.2.26 查询区块区间交易数量（by method ID）

`public SingleValueReturn getTxCountByMethodID(String from, String to, String contractAddress, String methodID)`

```
/**
* 查询区块区间交易数量（by method ID）
*
* @param from
* @param to
* @param contractAddress
* @param methdID
*/
BlockReturn blockReturn = hyperchain.getLatestBlock();
int end = Integer.valueOf(blockReturn.getNumber().substring(2), 16);
int start = end - 2;
SingleValueReturn singleValueReturn =
hyperchain.getTxCountByMethodID(String.valueOf(start), String.valueOf(end),
receiptReturn.getContractAddress(), methodIDMap.get("add"));
```

Java

3.2.27 查询指定时间区间内的交易数量

`public SingleValueReturn getTxsCountByTime(BigInteger startTime, BigInteger endTime)`

```
/**
* 查询交易数量 by time
*
* @param startTime 开始时间戳 ns
* @param endTime   结束时间戳 ns
*/
SingleValueReturn result = hyperchain.getTxsCountByTime(new BigInteger("1"), new
BigInteger("1778959217012956575"));
Assert.assertEquals(true, result.getResult().startsWith("0x"));
```

Java

3.3 Contract 相关接口

1. 合约调用同步接口：节点接收到SDK发送来的交易后，将会返回交易hash，由于需要等待执行以及共识的结束，同步请求将会根据平台返回的交易hash去轮询查询交易对应的回执信息，在请求到回执或者轮询结束之前，将会阻塞主线程；
2. 合约调用异步接口：异步接口将在收到交易hash后，挂起一个线程去轮询结果，不阻塞主线程，异步接口的回调函数的模式将在查询到回执后自动执行后续逻辑；而future模式则将轮询回执结果存储在内存中，等待业务逻辑的处理，相对来说更加灵活，推荐使用future模式的异步接口；
3. 异步或同步调用可根据业务需求自行选择，同时同步和异步接口均提供指定特殊节点发送的方式，通过指定节点id的形式实现（节点id从1开始计数，计数顺序为配置文件中node的顺序）。当不指定节点id的时候则会由SDK根据负载均衡的规则来发送（详见3.1.6章节

），我们推荐使用不指定节点的方式。一般需要指定节点发送交易情况有simulate交易往特殊节点发送，或者某些节点不在应用可访问权限内。

3.3.1 编译源代码

3.3.1.1 编译Solidity合约

`CompileReturn compileContract(String sourceCode)`

```
/**
* 编译源代码
* @param sourceCode 源代码
* @return CompileReturn 可以得到编译返回值
*/
CompileReturn result = hyperchain.compileContract(TEST_SOURCECODE);
System.out.println(complieResult.getAbi());
System.out.println(complieResult.getBin());
System.out.println(complieResult.getTypes());
```

Java

说明：编译代码的返回值会返回3个数组,abi,bin,types,这三个数组是一一对应的,例如:

```
abi: [abi1,abi2,abi3]
bin: [bin1,bin2,bin3]
types:[contranctName1 ,contranctName2 ,contranctName3 ]
```

JavaScript

对应的, contranctName1 对应的 `abi` 为abi1, `bin` 为bin1

3.3.1.2 编译Java合约

`public static String compileContractJava(String sdkPath, String sourceDir, String destDir) throws ParamErrorException, CompileException`

```
/**
 * 编译Java合约
 * @param sdkPath The jar file compile need
 * @param sourceDir Java file path
 * @param destDir Target path
 * @return Target path if success
 * @throws ParamErrorException Param error
 */
例子：
String homePath = System.getenv("HOME");
String sdkPath = homePath + "/.m2/repository";
String sourceDir = "src/test/resources/javaContractExample/source1";
String destDir = "src/test/resources/javaContractExample/contract01";
String path = HyperchainAPI.compileContractJava(sdkPath,sourceDir,destDir);
```

Java

> 说明：Solidity合约是发送到平台编译的，而Java合约是在本地编译的。调用此方法时用户需要有编译合约需要的jar包(已在sdk中包含)，用户也可以自己实现此方法。在编译完成后，还需在destDir目录下创建一个名为contract.properties，在其中写入contract.name=合约名称，main.class=合约主类全类名。

例如：

```
contract.name=AccountSum
main.class=cn.hyperchain.jcee.contract.examples.sb.src.AccountSum
编译后文件架构示例：
|-contract                                          //编译成功合约路径
|-----cn.hyperchain.jcee.contract.examples.sb.src
|---------AccountSum.class                          //编译后合约
|-----contract.properties                           //配置文件
```

JavaScript

3.3.2 部署合约

3.3.2.1 同步

`ReceiptReturn deployContract(Transaction transaction) throws InterruptedException`

不同合约的Transaction构建请参考3.2部分文档。

```
/**
* 部署合约
* @param from 部署账户地址
* @return 部署交易Hash SingleValueReturn
* @throws InterruptedException
*/
Account account = new Account(TEST_ACCOUNT_JSON);
Transaction transaction = new Transaction(account.getAddress(),TEST_BIN,false,
VMType.EVM);
transaction.sign(TEST_ACCOUNT_JSON,TEST_PASSWD);
ReceiptReturn result = hyperchain.deployContract(transaction);
result.getRet();
// 或者可以指定节点发送，节点id从1开始，顺序同配置文件中一致
int id = 1;
ReceiptReturn result = hyperchain.deployContract(transaction, id);
```

Java

3.3.2.2 异步

异步方式有回调函数的形式和future的模式，可根据不同的需求来使用不同的方式：

1. 回调函数：

`void deployAysncContract(final Transaction transaction,final AsyncHandler callBack)`

```
/**
* 3.2 部署合约(构造函数)
* @param Transaction 交易结构体
* @param AsyncHandler 异步回调，需要实现此接口
* @throws Exception
*/
AsyncHandlerTest handler = new AsyncHandlerTest();
Transaction transaction = new Transaction(TEST_FROM,targetBin,false, VMType.EVM);
transaction.sign(TEST_PRIJSON,TEST_PASSWD);
hyperchain.deployAysncContract(transaction,handler);
while(handler.result==""){
       Thread.sleep(100);
}
System.out.println(handler.result);
JSONObject jsonObject = JSONObject.fromObject(handler.result);
String address = jsonObject.getString("contractAddress");
// 注意：AsyncHandlerTest类的定义
public class AsyncHandlerTest implements AsyncHandler{
    public String result = "";
    @Override
    public void onSuccess(StdReturn ret) {
        result = ret.getResult();
    }
    @Override
    public void onFailed(StdReturn ret) {
        result = "failed";
    }
}
```

Java

1. future模式：

`FutureTask deployAysncContract(final Transaction transaction)`

```
/**
 * 异步部署智能合约
 *
 * @param transaction 智能合约交易
 * @return futureTask 异步返回结果
 */
Account account = new Account(TEST_ACCOUNT_JSON);
String input = FunctionEncode.encodeFunction(methodName,funcParams);
Transaction transaction = new Transaction(account.getAddress(), bin, input,
false);
transaction.signWithSM2(TEST_ACCOUNT_JSON, TEST_PASSWD);
FutureTask<ReceiptReturn> futureTask = hyperchain.deployAysncContract(transactio);
String result = futureTask.get().getRet();
```

Java

3.3.3 调用合约

3.3.3.1 同步

`ReceiptReturn invokeContract(Transaction transaction) throws InterruptedException`

```
/**
* 调用智能合约
* @param transaction 智能合约交易
* @return 返回单个string返回值
*/
Account account = new Account(TEST_ACCOUNT_JSON);
String input = FunctionEncode.encodeFunction(methodName,funcParams);
Transaction transaction = new Transaction(account.getAddress(), contractAddress ,
input, false, VMType.EVM);
transaction.sign (TEST_ACCOUNT_JSON,TEST_PASSWD);
ReceiptReturn result = hyperchain.invokeContract(transaction);
result.getRet();
// 或者可以指定节点发送，节点id从1开始，顺序同配置文件中一致
int id = 1;
ReceiptReturn result = hyperchain.invokeContract(transaction, id);
```

Java

合约调用结果需要解码后才能正常使用，上述方式未将合约结果解码，可以采用传入abi的重载调用函数来自动将返回结果解码，如没有特殊的需求，我们推荐在外部解码，不要一步到位，具体解码规则可以参考3.7文档：

`ReceiptReturn invokeContract(Transaction transaction,String methodName,String abi) throws Exception`

```
/**
 * 便捷调用
 * @param transaction 交易
 * @param methodName 需要调用的方法名
 * @param abi abi
 * @return 返回解析之后的ReceiptReturn
 * @throws Exception 异常
 */
ReceiptReturn ret = hyperchain.invokeContract(transaction,methodName,abi);
System.out.println(ret.getDecodedRet().get(0));
// 同样，便捷调用也支持向特殊id的节点发送
int id = 1;
ReceiptReturn result = hyperchain.invokeContract(transaction,methodName,abi,id);
```

Java

3.3.3.2 异步

异步方式有回调函数的形式和future的模式，可根据不同的需求来使用不同的方式：

1. 回调函数：

`void invokeContractAsync(Transaction transaction,final AsyncHandler callBack) throws Exception`

```
/**
* 调用智能合约
* @param transaction 智能合约交易
* @param callback 回调函数，实现接口AsyncHandler
*/
Transaction transaction = new Transaction(TEST_FROM,TEST_BIN,false, VMType.EVM);
AsyncHandlerTest handler = new AsyncHandlerTest();
hyperchain.invokeContractAsync(transaction2,handler);
while(handler.result==""){
        Thread.sleep(100);
}
System.out.println(handler.result)
```

Java

```
// 注意：AsyncHandlerTest类的定义
public class AsyncHandlerTest implements AsyncHandler{
    public String result = "";
    @Override
    public void onSuccess(StdReturn ret) {
        result = ret.getRet();
    }
    @Override
    public void onFailed(StdReturn ret) {
        result = "failed";
```

```
    }
}
```

Java

1. future模式：

`FutureTask invokeContractAsync(final Transaction transaction)`

```
/**
 * 异步调用合约
 *
 * @param transaction 智能合约交易
 * @return futureTask 异步返回结果
 */
Transaction transactionWithParams = new Transaction(account.getAddress(),
contractAddress, payloadWithParmas, false);
transactionWithParams.signWithSM2(TEST_ACCOUNT_JSON, TEST_PASSWD);
FutureTask<ReceiptReturn> futureTaskWithParams =
hyperchain.invokeContractAsync(transactionWithParams);
String txHashWithParams = futureTaskWithParams.get().getTxHash();
```

Java

同样的，我们为异步接口也增加了调用后解码的便捷调用和指定节点发送的特殊调用，接口形式如同步所示，这里就不在赘述，这里只展示异步便捷调用的接口：

`void invokeContractAsync(final Transaction transaction, String methodName, String abi, final AsyncHandler callBack) throws Exception`

`FutureTask invokeContractAsync(final Transaction transaction, final String methodName, final String abi)`

3.3.4 智能合约管理

`ReceiptReturn maintainContract(Transaction transaction) throws InterruptedException`

合约管理共分为合约升级、合约冻结和合约解冻（opcode分别对应为1、2和3），构造不同操作的Transaction可以查看Transaction部分的文档。

合约的具体升级流程如下：

1. 客户端发起升级合约调用，传入新合约的源码并使调用的操作码为1；

2. hyperchain虚拟机判断调用合约的操作码为1，判断为升级合约操作；

3. 虚拟机进行一次执行，目的是得到存储的可直接执行的源码 （说明：对于Solidity合约编译后的bin并不是虚拟机真正执行的bin，一般会在部署时处理为可直接执行的源码，并存储在账本中，在调用时直接取出执行）；

4. 将调用合约地址在账本中对应的源码替换为上一步得到的将可直接执行的源码；

5. 结果共识成功后，交易执行完毕。

```
/**
 * 智能合约升级
 *
 * @param transaction 智能合约交易
 */
Account account = new Account(TEST_ACCOUNT_JSON);
// payload为升级后的合约bin
Transaction transaction = new Transaction(account.getAddress(), contractAddress,
targetBinUpgrade, 1, VMType.EVM);
transaction.sign (TEST_ACCOUNT_JSON,TEST_PASSWD);
ReceiptReturn invokeResult = hyperchain.maintainContract(transaction);
System.out.println("result >> " + invokeResult.getRet());
```

Java

```
/**
 * 冻结、解冻合约
 *
 * @param transaction 智能合约交易
 */
// 冻结、解冻合约不需要payload，将对对应的合约地址的合约进行相应操作
Transaction transaction = new Transaction(account.getAddress(), contractAddress,
2, VMType.EVM);
```

```
// Transaction transaction = new Transaction(account.getAddress(),
contractAddress, 3, VMType.EVM);
transaction.sign (TEST_ACCOUNT_JSON,TEST_PASSWD);
ReceiptReturn invokeResult = hyperchain.maintainContract(transaction);
System.out.println("result >> " + invokeResult.getRet());
```

Java

3.3.5 获取合约状态

`SingleValueReturn getContractStatus(String contractAddress)`

```
/**
 * 获取合约状态
 * @param contractAddress 合约地址
 * @return
 */
SingleValueReturn contractStatus = hyperchain.getContractStatus(contractAddress);
System.out.println("contract status >> " + contractStatus.getResult());
```

Java

3.3.6 查询已部署合约列表

`public SingleValueReturn getDeployedList(String address)`

```
/**
 * 查询已部署的合约地址列表
 * @param address 部署者地址
 * @return 已部署的所有合约地址
 */
Account account = new Account(TEST_ACCOUNT_JSON);
SingleValueReturn singleValueReturn =
hyperchain.getDeployedList(account.getAddress());
System.out.println(singleValueReturn.getResult());
JSONArray results = JSONArray.fromObject(singleValueReturn.getResult());
String one = (String) results.get(0);
System.out.println(one.length());
```

Java

3.3.7 获取账户余额

`public SingleValueReturn getBalance(String address)`

```
/**
 * 查询账户余额
 * @param address 账户地址
 * @return resule
 */
SingleValueReturn balance =
 hyperchain.getBalance("0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd");
System.out.println(balance.getResult());
```

Java

3.3.8 获取合约字节编码

`public SingleValueReturn getCode(String address)`

```
/**
 * @param address 合约地址
 * @return 单值json string返回值
 */
SingleValueReturn singleValueReturn = hyperchain.getCode(address);
String contractInHexByte = singleValueReturn.getResult();
Assert.assertEquals(true, contractInHexByte.startsWith("0x"));
```

Java

3.3.9 获取合约数量

`public SingleValueReturn getContractCountByAddr(String address)`

```
/**
 * 获取合约数量
 * @param address 合约地址
 */
SingleValueReturn singleValueReturn =
hyperchain.getContractCountByAddr(account.getAddress());
String contractInHexByte = singleValueReturn.getResult();
Assert.assertEquals(true, contractInHexByte.startsWith("0x"));
```

Java

3.3.10 查询合约部署者

`public SingleValueReturn getCreator(String contractAddress)`

```
/**
* 查询合约部署者
* @param contractAddress 合约地址
*/
SingleValueReturn singleValueReturn = hyperchain.getCreator(contractAddress);
String creatorAddress = singleValueReturn.getResult();
Assert.assertEquals("0x"+account.getAddress().toLowerCase(), creatorAddress);
```

Java

3.3.11 查询合约部署时间

`public SingleValueReturn getCreateTime(String contractAddress)`

```
/**
* 查询合约部署时间
* @param contractAddress 合约地址
*/
SingleValueReturn singleValueReturn = hyperchain.getCreateTime(contractAddress);
String createTime = singleValueReturn.getResult();
Assert.assertEquals(true, createTime.endsWith("CST"));
```

Java

3.4 Block相关接口

3.4.1 取得最新区块信息

`BlockReturn getLatestBlock()`

```
/**
* 取得最新的区块信息
* @return 区块信息
*/
BlockReturn blockReturn = hyperchain.getLatestBlock();
System.out.println(blockReturn.getResult());
```

Java

3.4.2 取得指定区块列表

支持使用String类型的区块号，isPlain表示是否包含交易，默认false

`ArrayList<BlockReturn> getBlocks(BigInteger from, BigInteger to)`

`ArrayList<BlockReturn> getBlocks(String start, String end)`

`ArrayList<BlockReturn> getBlocks(BigInteger from, BigInteger to, Boolean isPlain)`

`ArrayList<BlockReturn> getBlocks(String from, String to, boolean isPlain)`

```
/**
 * 取得指定开始与结束的区块
 *
 * @param from 开始区块号
 * @param to   结束区块号
 * @param isPlain   区块是否包含交易(注意：此字段为可选字段，默认是false)
 * @return 返回区块信息
 */
ArrayList<BlockReturn> blockReturns = hyperchain.getBlocks(new BigInteger("1"),new BigInteger("2"));
Assert.assertEquals(2,blockReturns.size());
```

Java

3.4.3 取得指定区块 by hash

`BlockReturn getBlkByHash(String blockHash, boolean isPlain)`

```
/**
 * 查询区块信息 by blk Hash
```

```
 * block_getBlocksByHash
 *
 * @param blockHash blk hash
 * @param isPlain 返回的区块是否包含交易
 * @return 区块信息
 */
BlockReturn blockReturn =
hyperchain.getBlkByHash("0xec86d97fb08dd10c4a238c7ba2993ee62d3f33df70b7b25595328a8
f36da08ab", false);
System.out.println(blockReturn.getResult());
```

Java

3.4.4 批量取得指定区块 by hash list

`ArrayList<BlockReturn> getBatchBlkByHash(ArrayList<String> blockHashLis t)`

`ArrayList<BlockReturn> getBatchBlkByHash(ArrayList<String> blockHashList, Boolean isPlain)`

```
/**
* 查询区块信息 by blk Hash list 批量查询(重载方法)
* block_getBatchBlocksByHash
*
* @param blockHashList blk hash list
* @param isPlain 返回区块是否包含交易 (注意：该字段可选，默认为false)
* @return 区块信息
*/
hashList.add(latestBlock.getHash());
hashList.add(secondBlock.getHash());
ArrayList<BlockReturn> returns = hyperchain.getBatchBlkByHash(hashList);
Assert.assertEquals(2, returns.size());
Assert.assertEquals(66, returns.get(0).getHash().length());
```

Java

3.4.5 取得指定区块 by number

`BlockReturn getBlkByNum(BigInteger blkNumber, Boolean isPlain)`

```
/**
 * 查询区块信息 by blk Number
 *
 * @param blkNumber 区块 Number
 * @param isPlain 返回的区块是否包含交易
 * @return 区块信息
 */
 BlockReturn blockReturn = hyperchain.getBlkByNum(new BigInteger("2",16), false);
 Assert.assertTrue(blockReturn.getRawcode() == 0);
```

Java

3.4.6 取得指定区块 by number(参数为String)

`BlockReturn getBlkByNum(String blkNumber, Boolean isPlain)`

```
/**
 * 查询区块信息 by blk Number(重载方法)
 *
 * @param blkNumber 区块 Number 可以为`latest`
 * @param isPlain   返回的区块是否包含交易
 * @return 区块信息
 */
例：
BlockReturn blockReturn2 = hyperchain.getBlkByNum("latest", false);
Assert.assertTrue(blockReturn2.getRawcode() == 0);
```

Java

3.4.7 批量取得指定区块 by number list取得最新区块信息

支持以String为区块号查询

`ArrayList<BlockReturn> getBatchBlkByNum(ArrayList<BigInteger> blkNumber List)`

`ArrayList<BlockReturn> getBatchBlkByStrNum(ArrayList<String> blkNumberL ist)`

`ArrayList<BlockReturn> getBatchBlkByNum(ArrayList<BigInteger> blkNumberList, Boolean isPlain)`

`ArrayList<BlockReturn> getBatchBlkByStrNum(ArrayList<String> blkNumberList, boolean isPlain)`

```
/**
* 上述为两个重载方法
* 查询区块信息 by batch blk Number 批量查询
* @param blkNumberList  区块 Number list
* @param isPlain 返回的区块是否包含交易 (注意，该字段可选，默认为false)
* @return 区块信息
*/
numberList.add(latestNumber);
numberList.add(latestNumber.subtract(new BigInteger("1")));
numberList.add(latestNumber.subtract(new BigInteger("2")));
ArrayList<BlockReturn> blockReturn = hyperchain.getBatchBlkByNum(numberList);
Assert.assertTrue(blockReturn.get(0).getRawcode()  == 0);
ArrayList<BlockReturn> blockReturn = hyperchain.getBatchBlkByNum(numberList,
true);
Assert.assertTrue(blockReturn.get(0).getRawcode()  == 0);
```

Java

3.4.8 取得区块平均生成时间

支持以String为区块号查询

`SingleValueReturn getAvgGenerateTimeByBlockNumber(BigInteger from,BigInteger to)`

`SingleValueReturn getAvgGenTimeByBlkNum(String from, String to)`

```
/**
* 查询区块平均生成
* @param from 起始区块
* @param to 结束区块
* @return 执行时间 16进制 单位 ms
*/
SingleValueReturn singleValueReturn = hyperchain.getAvgGenTimeByBlkNum(new
BigInteger("2"),new BigInteger("3"));
System.out.print(singleValueReturn.getResult());
```

Java

3.4.9 根据区块生成时间查询区块

支持以String类型为时间戳查询

`BlockNumReturn getBlksByTime(BigInteger startTime, BigInteger endTime)`

`BlockNumReturn getBlksByTime(String startTime, String endTime)`

```
/**
* 根据区块生成时间查询区块
* @param startTime 开始查询时间
* @param endTime 结束查询时间
* @return 返回BlockNumReturn
*/
BlockNumReturn blockNumReturn = hyperchain.getBlksByTime(new BigInteger("2"),new
BigInteger("3"));
blockNumReturn.getResult();
```

Java

3.4.10 查询指定时间段内的TPS

`public TPSReturn queryTPS(BigInteger startTime, BigInteger endTime)`

```
/**
* 查询时间段内的TPS
*
* @param startTime 开始时间戳 ns
* @param endTime   结束时间戳 ns
* @return 返回TPSReturn
*/
TPSReturn result = hyperchain.queryTPS(new BigInteger("1"), new
BigInteger("1778959217012956575"));
```

Java

3.4.11 查询创世区块号

`public SingleValueReturn getGenesisBlk()`

```
/**
* 查询创世区块号
*
*/
SingleValueReturn singleValueReturn = hyperchain.getGenesisBlk();
Assert.assertEquals(true, singleValueReturn.getResult().contains("0x"));
```

Java

3.4.12 查询最新区块号

`public SingleValueReturn getChainHeight()`

```
/**
* 查询最新区块号
*
*/
SingleValueReturn singleValueReturn = hyperchain.getChainHeight();
Assert.assertEquals(true, singleValueReturn.getResult().contains("0x"));
```

Java

3.5 节点相关接口

3.5.1 获取节点信息

`ArrayList<NodeInfoReturn> getNodes()`

```
/**
* 取得所有的节点信息
* @return 返回节点信息列表
*/
ArrayList<NodeInfoReturn> nodeInfoReturns= hyperchain.getNodes();
for (NodeInfoReturn nodeInfoReturn : nodeInfoReturns){
    System.out.println(nodeInfoReturn.getIp());
    System.out.println(nodeInfoReturn.getId());
    System.out.println(nodeInfoReturn.getPrimary());
    System.out.println(nodeInfoReturn.getDelay());
}
```

Java

3.5.2 获取节点哈希值

3.5.2.1 获取随机节点哈希值

`String getNodeHash()`

```
/**
* 获取当前节点哈希值
* @return
*/
String nodeHash = hyperchain.getNodeHash()
```

Java

3.5.2.2 获取指定节点哈希值

`public String getNodeHashByID(int id) throws ParamErrorException`

```
/**
 * 获取指定节点哈希值
 * @param id 节点的ID值,为配置文件中的顺序
 * @return 节点的hash值
 * @since 4.0.1
 */
String nodeHash = hyperchain.getNodeHashByID(1);
```

Java

3.5.3 根据节点哈希值删除节点

`boolean deleteNode(String nodeHash)`

```
/**
* 根据节点哈希值删除节点
```

```
 * @param nodeHash 节点哈希值
 * @return
 */
Hyperchain.deleteNode(nodeHash);
```

Java

3.5.4 NVP根据ID值断开与VP节点的链接

说明：如果同所有VP节点断开连接，则该NVP节点会自动关闭，该API请求应当发往NVP节点。

`boolean disconnectVP(String vpNodeHash)`

```
/**
 * 根据节点id值同VP节点断开连接
 * @param nodeid 节点id值
 * @return 删除操作结果
 */
boolean success = hyperchain.disconnectVP(nodehash);
```

Java

3.5.5 删除NVP节点

`boolean deleteNVP(String nodeHash)`

```
/**
 * 根据节点hash值删除NVP节点
 * @param nodeHash 节点hash值
 * @return 删除操作结果
 */
boolean success = hyperchain.deleteNVP(nodehash);
```

Java

3.5.6 删除VP节点

`boolean deleteVP(String nodeHash)`

```
/**
 * 根据节点hash值删除VP节点
 * @param nodeHash 节点hash值
 * @return 删除操作结果
 */
boolean success = hyperchain.deleteVP(nodehash);
```

Java

3.6 数据归档相关接口

为了解决块链式区块链数据无限增长的问题，hyperchain底层区块链平台提供了一种基于"状态快照"的数据归档方法，以解决区块链数据的存储问题。 hyperchain存储的数据从内容上可以划分为两类:

- 区块数据
- 世界状态数据

前者指的就是块链式结构"区块链"中的区块数据;而后者指的是从"创世状态"开始，历经若干次的状态变迁后 (执行交易)，当前区块链世界中所有账户的状态。

区块链网络作为一个整体，可以被看成是一个基于交易的状态机:hyperchain开始于一个初始("Genesis") 状态，然后伴随着交易的执行不断地进行状态变迁得到了了一个截止目前为止的最终状态。这个最终状态是区块链网络能接受的经过共识的区块链世界状态。这个状态保存每个账户的信息，例如账户余额，账户合约源码，账户变量数据等。

但是随着运行时间的增长，某些账户(合约账户)自身的变量数据会越来越大,因而这类数据同样需要进行归档。

针对于两种不同的存储内容，我们的数据归档也分为两个方面，其一为对块链式的区块链数据进行归档，其二为对特定的合约账户的数据进行归档。

数据归档是对**单个节点**的归档，因此需要选择**归档的节点**。SDK无法控制归档的时间，需要用户来控制。

注意：接口中所说的节点id index均从1开始，对应配置文件的顺序

3.6.1 创建快照

用户可以在任意时间点发起"快照制作请求"，而请求类型也分为两类：

（1）即时触发类请求：

顾名思义，即时触发类请求指的是当用户发起请求时，本地节点立即以当前的系统状态为基点，进行状态备份，并将备份数据作为一份"状态快照存储"。

其中参数params中的latest便是即时触发类请求的标志。

（2）条件触发类请求：有别于即时触发类请求，若用户需要在未来一个指定的时间进行快照制作，便可以使用条件触发类请求作为代替。在这里，触发条件特指区块高度。

其中参数params中的100指的是当本地区块链的高度达到100时，触发进行快照制作。

`public SingleValueReturn makeSnapshot(int blockHeight, int nodeId)`

`public SingleValueReturn makeSnapshot(String blockHeightString, int nodeId)`

```
/**
 * 创建快照
 * @param blockHeight 触发的区块高度
 * @return 包含快照标识
 */
//发起制作快照请求
SingleValueReturn makeResult = hyperchain.makeSnapshot(0, 1);（参数为0表示立即制作）
Assert.assertEquals(34,makeResult.getResult().length());
snapshotHash = makeResult.getResult();
SingleValueReturn makeResult2 = hyperchain.makeSnapshot(1000);
Assert.assertEquals(34,makeResult2.getResult().length());
```

Java

3.6.2 查询创建快照结果

用户可以通过轮询查询制作结果的方式检查前一步发送的快照制作请求是否执行成功。

`public SingleValueReturn querySnapshotResult(String snapShotID, int nodeId)`

```
/**
 * 查询创建快照结果
 * @param snapShotID 查询的快照hash
 * @return 包含快照标识
 */
Boolean flag = true;
int queryTimes = 100;
SingleValueReturn queryResult = null;
while(flag && queryTimes-- > 0){
    queryResult = hyperchain.querySnapshotResult(makeResult.getResult(), 1);
    if(queryResult.getResult().equals("true")){
        flag = false;
    }
    Thread.sleep(100);
}
Assert.assertEquals("true", queryResult.getResult());
```

Java

3.6.3 检查"状态快照"正确性

状态快照其本质上是区块链节点"世界状态"的一个备份，创建期间会为备份数据进行一次全量的哈希计算，为确认该数据的正确性。

当用户在进行数据归档操作前，我们推荐进行一次该快照数据的正确性检查。

`public SingleValueReturn checkSnapshot(String snapShotID, int nodeId)`

```
/**
 * 检查"状态快照"正确性
```

```
 * @param snapShotID 待归档的区块
 * @return 结果
 */
SingleValueReturn chechResult = hyperchain.checkSnapshot(makeResult.getResult(),
1);
Assert.assertEquals("true",chechResult.getResult());
```

Java

3.6.4 数据归档

指定的区块链数据将被转储至"历史数据库"中。历史数据库的路径可在配置文件中进行配置。

`public SingleValueReturn archiveSnapshot(String snapShotID, Boolean syncR eturn, int nodeId)`

```
/**
 * 数据归档
 * @param snapShotID 待归档的区块
 * @param syncReturn 是否同步返回,true同步返回则等待归档结束返回,false则是调用后立即返回，但不代表执行成功，需要查询归档结果判断是否成功。
 * @return 归档id
 */
SingleValueReturn archiveResult = hyperchain.archiveSnapshot(snapshotHash,  true,
1);
String archiveHash = archiveResult.getResult();
Assert.assertEquals(34, archiveHash.length());
```

Java

3.6.5 查询归档结果

与发起快照制作请求类似，数据归档同样是一个异步行为，因此需要通过额外的RPC请求查询最终的制作结果。

`public SingleValueReturn queryArchive(String snapShotID, int nodeId)`

```
/**
 * 查询归档结果
 * @param snapShotID 待查询的归档id
 * @return 结果
 */
Boolean flag = true;
int queryTimes = 100;
SingleValueReturn queryResult = null;
while(flag && queryTimes-- > 0){
    queryResult = hyperchain.queryArchive(archiveHash, 1);
    if(queryResult.getResult().equals("true")){
        flag = false;
    }
    Thread.sleep(100);
}
Assert.assertEquals("true", queryResult.getResult());
```

Java

3.6.6 列举快照

`public SingleValueReturn listSnapshot(int nodeId)`

```
/**
* 列举快照
* @param nodeId
*/
//列举快照
SingleValueReturn singleValueReturn = hyperchain.listSnapshot(1);
Assert.assertEquals(0, singleValueReturn.getRawcode());
```

Java

3.6.7 查询快照

`public SingleValueReturn readSnapshot(String filterId, int nodeId)`

```
/**
* 查询快照
* @param filterId
* @param nodeId
*/
SingleValueReturn singleValueReturn1 =
hyperchain.readSnapshot(makeResult.getResult(), 1);
```

```
JSONObject jsonObject = JSONObject.fromObject(singleValueReturn1.getResult());
Assert.assertEquals(makeResult.getResult(), jsonObject.getString("filterId"));
```

Java

3.6.8 快照删除

`public SingleValueReturn deleteSnapshot(String filterId, int nodeId)`

```
/**
* 快照删除
* @param filterId
*/
SingleValueReturn singleValueReturn2 =
hyperchain.deleteSnapshot(makeResult.getResult(), 1);
Assert.assertEquals("true", singleValueReturn2.getResult());
```

Java

3.6.9 恢复归档数据

`public SingleValueReturn restoreArchive(String snapShotID, Boolean syncReturn, int nodeId)`

```
/**
* 恢复归档数据
* @param filterId
* @param syncReturn
* @param nodeId
*/
SingleValueReturn singleValueReturn = hyperchain.restoreArchive(snapshotHash,
true, 1);
Assert.assertEquals("true", singleValueReturn.getResult());
```

Java

3.6.10 恢复所有归档数据

`public SingleValueReturn restoreAllArchive(Boolean syncReturn, int nodeId)`

```
/**
* 恢复所有归档数据
* @param syncReturn
* @param nodeId
*/
SingleValueReturn singleValueReturn2 = hyperchain.restoreAllArchive(true, 1);
System.out.println(singleValueReturn2);
```

Java

3.7 返回值解析

3.7.1 通用返回值解析(推荐)

`String resultDecode(String methodName,String abi, String encoded ) throws UnsupportedEncodingException`

本方法将会利用abi 将返回值直接解析为json字符串，拥有value(返回值)， mayvalue(推测返回值)字段，更加便捷。

```
/**
 * decode the smart contract invoke return data, and return a formatted json
string.
 * @param methodName the Smart Contract Method Name
 * @param abi the Smart Contract Abi, which should start with `[` and end with `]`
 * @param encoded the encode binary (hex) return data.
 * @return a formatted json string
 * @throws UnsupportedEncodingException if bytes con not decode with UTF-8 code
*/
// 取得交易回执
ReceiptReturn result4 = hyperchain.getTransactionReceipt(result3.getResult());
System.out.println(result4.getResult());
System.out.println(FunctionDecode.resultDecode("add",TEST_ABI,result4.getRet()));
//样例输出
//{"result":
[{"type":"java.lang.Boolean","value":"true","mayvalue":"true"}],"status":"SUCCESS"
,"txhash":"0x533a013a82c831fb7986d240bfca3b2257760c0658f12a871257337e9f77f4c4","co
de":0,"info":"invoke Success!"}
```

Java

3.7.2 复杂返回值解析(不推荐)

`ArrayList<Object> complexDecode(String methodName, String abi, String data)`

```
/**
 * 解析返回值,可以解析复杂返回值,推荐用本方法解析返回值,返回值需要自己解析并强制转换类型
 * @param methodName 需要解析的函数名
 * @param abiJson         相应整个合约的abi
 * @param data        返回数据
 * @return 返回值的一个Object Array list 需要自己向下转型
 */
String TEST_RET =
"0x0000000000000000000000000000000000000000000000000000000000000007";
String TEST_ABI = "[{\"constant\":false,\"inputs\":
[{\"name\":\"a\",\"type\":\"uint256\"}],\"name\":\"multiply\",\"outputs\":
[{\"name\":\"d\",\"type
\":\"uint256\"}],\"payable\":false,\"type\":\"function\"}]" ;
FunctionDecode.complexDecode( "multiply", TEST_ABI, TEST_RET);
```

Java

3.7.3 Java合约返回值解析

`public static String resultDecodeJava(String ret) throws UnsupportedEncodingException`

```
/**
 * Java合约解析
 *
 * @param ret 未解析的返回值
 * @return 返回封装的解析结果
 * @throws UnsupportedEncodingException
 */
String ret = receiptReturn.getRet();
String decResult = FunctionDecode.resultDecodeJava(ret);
System.out.println(decResult);
```

Java

> 说明：此解析后的结果类型皆为String，用户需要根据自己需要的类型在得到String后进行适当的转换。

3.7.4 动态类型解析说明

Solidity源代码

```
contract MyContract{
    function MutiReturns(uint256 a, bytes32 b,address c)returns(uint256 [],
bytes32[],address[]){
        uint256[] aa ;
        bytes32[] bb;
        address[] cc;
        aa. push(a);
        aa. push(a);
        bb.push(b);
        bb.push(b);
        cc. push(c);
        cc. push(c);
        return (aa, bb,cc);
    }
}
```

JavaScript

> 说明：当返回值为动态类型的时候，类似于uint256[] , bytes32[] 类型的时候，由于无法判断返回值元素的个数，所以执行返回的时候，会将动态类型全部返回，需要进行尝试解析。

例子：

例如返回值为returns(uint256 [],bytes32[],address[])

返回值原文：

```
00000000000 00000000000 00000000000 00000000000 00000000000 000000060
00000000000 00000000000 00000000000 00000000000 00000000000 000000140
00000000000 00000000000 00000000000 00000000000 00000000000 000000220
00000000000 00000000000 00000000000 00000000000 00000000000 000000006
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
```

```
48656c6c6f20576f726c642100000000 00000000000 00000000000 0000000000
48656c6c6f20576f726c642100000000 00000000000 00000000000 0000000000
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
00000000000 00000000000 00000000000 00000000000 00000000000 000000006
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
48656c6c6f20576f726c642100000000 00000000000 00000000000 0000000000
48656c6c6f20576f726c642100000000 00000000000 00000000000 0000000000
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
00000000000 00000000000 00000000000 00000000000 00000000000 000000006
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
00000000000 00000000000 00000000000 00000000000 00000000000 000000002
00000000000 00000000000 00000000000 00000000000 00000000000 000000000
00000000000 00000000000 00000000000 00000000000 00000000000 000000000
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
00000000000 00000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
```

JavaScript

在解析的时候需要先解析uint256 类型：

```
2 // correct
2 // correct
327457249635 204591281676 07564932637 663996319117 01979283679 80961554385 128652
80 // wrong, should be bytes type,byte[]
327457249635 204591281676 07564932637 663996319117 01979283679 80961554385 128652
80 // wrong, should be bytes type,byte[]
33681762355 798862497152 07525798101 50033537725 // wrong, should be address type
33681762355 798862497152 07525798101 50033537725 // wrong, should be address type
```

JavaScript

然后再按 bytes32 类型解析：

```
// wrong, should be uint256 type ,  BigInteger
// wrong, should be uint256 type ,  BigInteger
Hello World! // correct
Hello World! // correct
z□Ď]0□P□□:□†// wrong, should be address type ,  byte[]
z□Ď]0□P□□:□†// wrong, should be address type ,  byte[]
```

JavaScript

然后按 address 类型解析:

```
0x0000000000000000000 00000000000 0000000002 // wrong type, should convert to
BigInteger
0x0000000000000000000 00000000000 0000000002 // wrong type, should convert to
BigInteger
0x0000000000000000000 00000000000 0000000000 // wrong type, should convert to
byte[]
0x0000000000000000000 00000000000 0000000000 // wrong type, should convert to
byte[]
0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd // correct
0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd // correct
```

JavaScript

> 强调：由于返回值并不会将返回的动态类型进行分割，sdk 也无法推定返回值的动态类型的长度，所以只能通过尝试解析的方式取得返回值，目前不支持strings 类型的解析，将会当做bytes32 类型解析。

复杂解析返回值示例

```
String abijson = "[{\"constant\":false,\"inputs\":
[{\"name\":\"a\",\"type\":\"uint256\"},{\"name\":\"b\",\"type\":\"bytes32\"},
{\"name\":\"c\",\"type\":\"address\"}],\"name\":\"MutiReturns\",\"outputs\":
[{\"name\":\"\",\"type\":\"uint256[]\"},{\"name\":\"\",\"type\":\"bytes32[]\"},
{\"name\":\"\",\"type\":\"address[]\"}],\"payable\":false,\"type\":\"function\"}]"
;
try {
ArrayList<Object> returns = FunctionDecode.complexDecode("Muti
Returns",abijson," 0x0000000000000000000 00000000000 00000000000 00000000000
0000000000600000000000 00000000000 00000000000 00000000000 00000000000 00000001400
0000000000000000000000 00000000000 00000000000 00000000000 00000002200000
0000000000000000000000 00000000000 00000000000 00000000000 00006000000 00000000000
0000000000000000000000 00000000000 00000000000 00200000000 00000000000 00000000000
0000000000000000000000 00000000000 248656c6c6f20576f726c642100000000 00000000000
0000000000000000000004 8656c6c6f20576f726c642100000000 00000000000 00000000000
0000000000000000000000 00000000000 0000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
```

```
0000000000000000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd 00000000000
0000000000000000000000 00000000000 00000000000 00000000600 00000000000 00000000000
0000000000000000000000 00000000000 000000200000 00000000000 00000000000 0000000000
000000000000 00000000000 000248656c6 c6f20576f726c642100000 00000000000
0000000000000000000000 0048656c6c6f20576f726c642100000 00000000000 00000000000
0000000000000000000000 00000000000 0000000f1a7a08ccc48e5d30f80850cf1cf283aa3abd
000000
0000000000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd 00000000000
0000000000000000000000 00000000000 00000000000 00000000600 00000000000 00000000000
0000000000000000000000 00000000000 000000200000 00000000000 00000000000
00000000000 0000000000000000000000 000200000000 00000000000 00000000000
00000000000 0000000000000000000000 00000000000 00000000000 00000000000 00000000000
00000000000 0000000000000000000000 00000000000
0000f1a7a08ccc48e5d30f80850cf1cf283aa3abd 00000000
00000000000000 00000f1a7a08ccc48e5d30f80850cf1cf283aa3abd") ;
for (Object ret : returns){
  if(ret.getClass().getName().equals("java.util.ArrayList")){
  ArrayList<Object> array = (ArrayList<Object>) ret ;
  for (Object ele :array){
  // System.out.println(ele.getClass().getName()) ;
  if (ele.getClass().getName().equals("java.math.BigInteger")){
     System.out.println(FunctionDecode.retToBigInteger(ele));
  }else if(ele.getClass().getName().equals("java.lang.String")){
     System.out.println(ele) ;
  }else{
  try {

System.out.println(FunctionDecode.retToString(ByteUtils.toHexString((byte[])ele)))
;
  }catch (UnsupportedEncodingException e){
  System.out.println(ByteUtils.toHexString((byte[]) ele)) ;
  }
}
}
 }else{
   if(ret.getClass().getName().equals("[B"]){
     System.out.println(ByteUtils.toHexString((byte[]) ret.getClass().cast(ret)))
;
   }else{
     System.out.println(ret.getClass().cast(ret)) ;
   }
  }
 }
}catch (UnsupportedEncodingException | NotSupportedTypeException e)
 {
  e.printStackTrace() ;
 }
}
```

Java

3.7.5 输入值解析

`String inputDecode(String methodName,String abi, String encodedInput ) throws UnsupportedEncodingException`

本方法需要提供包含该方法的abi，否则无法解析。若该方法不存在，将会返回相应的错误值。

```
/**
 * @param methodName 方法名
 * @param abi abi
 * @param encodedInput 需要解码的input
 * @return 解码之后的json字符串
 * @throws UnsupportedEncodingException -
 */
String sourcecode="contract MyContract{\n" +
"    function MutiReturns(uint256 a,bytes32 b,address c)returns(uint256
[],bytes32,address[]){\n" +
"        uint256[] aa;\n" +
"        address[] cc;\n" +
"        \n" +
"        aa.push(a);\n" +
"        aa.push(a);\n" +
"        \n" +
"        cc.push(c);\n" +
"        \n" +
"        return (aa,b,cc);\n" +
"        \n" +
"      }\n" +
"}";
```

```
// get the abi
CompileReturn compileReturn = hyperchain.compileContract(sourcecode,1);
List<?> abiArray = (List<?>)compileReturn.getAbi();
String abi  = abiArray.get(0).toString();
//get input
FuncParamReal param1 = new FuncParamReal("uint256",new BigInteger("2"));
FuncParamReal param2 = new FuncParamReal("bytes32","Hello World!");
FuncParamReal param3 = new
FuncParamReal("address","0x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd");
String input = FunctionEncode.encodeFunction("MutiReturns",param1,param2,param3);
if (input.startsWith("0x")) {
        input = input.substring(10);
}
String inputString = FunctionDecode.inputDecode("MutiReturns",abi,input);
System.out.println(inputString);
//样例输出
//{"methodName":"MutiReturns","decodedInput":
[{"type":"java.math.BigInteger","value":"2","mayvalue":"2"},
{"type":"bytes","value":"48656c6c6f20576f726c642100000000000000000000000000000000
0000000","mayvalue":"Hello World!"},
{"type":"address","value":"000f1a7a08ccc48e5d30f80850cf1cf283aa3abd","mayvalue":"0
x000f1a7a08ccc48e5d30f80850cf1cf283aa3abd"}]}
//错误样例
{"error":"method name is not exist or this method is constructor"}
```

Java

3.7.6 构造器解析

`String constrcutorDecode(String abi, String encodedInput ) throws UnsupportedEncodingException`

```
/**
*
* @param abi abi
* @param encodedInput 需要解码的input
* @return 解码之后的json字符串
* @throws UnsupportedEncodingException -
*/
FunctionDecode.constructorDecode(abi, encodedInput);
```

Java

3.7.7 Java输入值解析

`public static DecResult inputDecodeJava(String encodeStr) throws UnsupportedEncodingException`

本方法传入参数为编码后的Java合约输入值。

```
/**
 * inputDecodeJava decode the input message encode with encodeFunctionJava
 * @param encodeStr encode string
 * @return DecResult
 * @throws UnsupportedEncodingException error
 */
String encode = FunctionEncode.encodeFunctionJava("Test","abc","123");
DecResult decResult = FunctionDecode.inputDecodeJava(encode);
System.out.println(decResult.getMethodName());
System.out.println(decResult.getArgs());
```

Java

3.8 Log值解析

3.8.1 solidity合约log值解析

> 说明：Solidity合约中事件结果(event)会在合约执行结果的"logs"字段中返回，因此只需要根据定义的返回值进行相应的获取即可。另外，如果使用的非便捷调用方法，需要使用FunctionDecode.logDecode()方法来解析

合约event示例：

```
contract Demo3 {
    event Deposit(address indexed addr1, bytes32 msg, string msg2);
    function demo3() returns (bytes32) {
            Deposit(msg.sender, "hello world1", "123");
            return "haha";
    }
}
```

JavaScript

对于便捷调用

```
String ret = hyperchain.invokeContract(transaction, "demo3", abi);
JSONObject retObj = JSONObject.fromObject(ret);
JSONArray log = retObj.getJSONArray("log");
for (Object ress : log) {
       JSONObject res = (JSONObject) ress;
       System.out.println("topics: " + res.getString("topics"));
       System.out.println("data" + res.getString("data"));
}
```

Java

对于非便捷调用需要调用 `FunctionDecode.logDecode(abi,log)` 来解析

`public static String logDecode(String abi, String encodedLog) throws UnsupportedEncodingException`

```
/**
 * log解析
 * @param abi contract abi
 * @param encodedLog log
 * @return decode json String
 * @throws UnsupportedEncodingException error
 */
例子:
JSONArray logArray = JSONArray.fromObject(FunctionDecode.logDecode(abi,log));
```

Java

3.8.2 Java合约log值解析

`public static String logDecodeJava(String encodedLog)`

```
/**
 * 解析Java合约产生的log信息
 * @param encodedLog 原log
 * @return 解码之后的json格式的字符串
 */
例子:
ReceiptReturn result4 = hyperchain.getTransactionReceipt(result3.getResult());
String log = FunctionDecode.logDecodeJava(result4.getLog());
JSONArray logs = JSONArray.fromObject(log);
JSONObject log1 = (JSONObject) logs.get(0);
```

Java

3.9 账户相关接口

主要是提供了明文私钥相关接口，所有涵盖accountJson相关的接口都有相应的明文私钥重载实现。

目前根据加密方式不同一共有10种类型账户，由Algo枚举类指定，非国密账户和国密账户各5种。

非国密账户有:

`ECKDF2("0x01")`, `ECDES("0x02")`, `ECRAW("0x03")`, `ECAES("0x04")`, `EC3DES("0x05")`

国密账户有:

`SMSM4("0x11")`, `SMDES("0x12")`, `SMRAW("0x13")`, `SMAES("0x14")`, `SM3DES("0x15")`

3.9.1 创建账户(加密)

3.9.1.1 非国密账户

`String newAccount(String password, Algo algo) throws GeneralSecurityException`

```
/**
* 生成指定加密类型的ECDAS账户
*
* @param password 设置账户密码
```

```
* @param algo 设置账户加密类型 “0x01”为ECKDF2（目前不支持）;“0x02”为ECDES；“0x03”为ECRAW；“0x04”为ECAES;“0x05”为EC3DES。
* @return json格式私钥存储文件
* @throws GeneralSecurityException
*/
String account = HyperchainAPI.newAccount( "123", Algo.ECAES);
System.out.println(account);
```

Java

3.9.1.2 国密账户

`String newAccountSM2(String password, Algo algo)`

```
/**
* 根据给定类型来获取国密账户account json
* @param password 用户密码
* @param algo 账户加密类型 “0x11”为SMSM4；“0x12”为SMDES；“0x13”为SMRAW；“0x14”为SMAES；“0x15”为SM3DES。
* @return  json
*/
String account = HyperchainAPI.newAccountSM2( "123", Algo.SMSM4);
System.out.println(account);
```

Java

3.9.2 创建账户(未加密)

3.9.2.1 非国密账户

`String newAccountRaw() throws GeneralSecurityException`

```
/**
* 生成私钥存储(未加密)文件，json字符串返回
* @return json格式私钥存储(未加密)文件
* @throws GeneralSecurityException 加密算法异常
*/
String account = HyperchainAPI.newAccountRaw();
System.out.println(account);
```

Java

3.9.2.2 国密账户

`String newAccountRawSM2() throws GeneralSecurityException`

```
/**
* 创建账户，生成账户未加密的公私钥(json字符串形式返回)
* @return 返回raw私钥string
* @throws GeneralSecurityException 加密算法异常
*/
String account = HyperchainAPI.newAccountRawSM2();
System.out.println(account);
```

Java

3.9.3 取得账户明文私钥以及地址

由于hyperchain默认将用户私钥加密存储，需要传入密码解密用户私钥，现在提供相应解密方法。

`ECPriv newAccount() throws GeneralSecurityException`

```
//getAccount() 随机生成一个账户
ECPriv testAccount = HyperchainAPI.newAccount();
```

Java

需要说明的是ECPriv类，其中包括用户的私钥以及相应的地址。

```
public class ECPriv {
    public byte[] getAddress() {
        return address;
    }
    public byte[] getPrivateKey() {
        return privateKey;
    }
}
```

Java

3.9.4 加密明文私钥

3.9.4.1 非国密

`String encryptAccount(ECPriv privateKey, String password, Algo algo) throws GeneralSecurityException`

```
/**
* 加密明文私钥
* @param privateKey 账户明文私钥及地址
* @param password 设置账户密码
* @param algo 设置账户加密类型 “0x01”为ECKDF2（目前不支持）;“0x02”为ECDES；“0x03”为ECRAW；“0x04”为ECAES;“0x05”为EC3DES。
* @return json格式加密私钥文件
* @throws GeneralSecurityException 加密算法异常
*/
System.out.println("address is " +
ByteUtil.toHexString(account.getAddressByte()));
String keyString = hyperchain.encryptAccount(account, "123", Algo.ECAES);
System.out.println(keyString);
```

Java

3.9.4.2 国密

`String encryptAccountSM2(String plainAccountJson, String password) throws PwdException`

```
/**
* 加密明文账户
* @param plainAccountJson  明文账户
* @param password   加密密码
* @return 密文账户（json格式）
*/
String accountJson = HyperchainAPI.newAccountRawSM2();
System.out.println("明文账户: " + accountJson);
String encryptedAccountJson = HyperchainAPI.encryptAccountSM2(accountJson, "123");
```

Java

3.9.5 解密私钥文件返回ECPriv

3.9.5.1 非国密

`ECPriv decryptAccount(String encrypetedAccountJson, String pwd) throws Exception`

解密私钥文件，并返回明文私钥。

```
/**
 * 解密私钥文件
 * @param accountJson 加密私钥文件
 * @param pwd 用户密码
 * @return {@link ECPriv}对象
 * @throws Exception 解密失败异常
 */
ECPriv ecp = HyperchainAPI.decryptAccount(account,"123");
JSONObject obj = JSONObject.fromObject(account);
Assert.assertEquals(obj.getString("address"),ByteUtils.toHexString(ecp.getAddressByte()));
```

Java

> 请注意:本方法支持解密明文存储的私钥存储文件以及加密后的私钥存储文件，该方法将会自动判断私钥类型，并返回ECPriv，需要特别注意的是，明文存储的私钥解密时，可以随意传第二个pwd参数。

3.9.5.2 国密

`String decryptAccountSM2(String encryptedAccountJson, String password) throws Exception`

解密私钥文件，并返回明文私钥。

```
/**
* 解密加密账户
* @param encryptedAccountJson  加密账户json字符串
* @param password  用户密码
* @return  明文账户json字符串
* @throws Exception
*/
```

```
String plainAccountJson=HyperchainAPI.decryptAccountSM2
(TEST_ACCOUNT_JSON,"123");
Account account = new Account(plainAccountJson);
```

Java

需要说明的是: `ecKey.getAddress()` 得到的是[]byte 需要转换成HEX String才能作为接口调用的参数进行使用。

3.10 WebSocket相关接口

3.10.1 获取webSocket连接

> 说明：用户传入节点id，获取该节点的webSocket连接，用户可以在这个连接上注册多个事件，并监听，当连接上无事件监听的时候，关闭连接。

`FutureTask<WebSocketAsyncHandler> eventConnect(int id, WebSocketAsyncHandler webSocketAsyncHandler)`

```
/**
* 获取webSocket连接
* @param id 节点id
* @param webSocketAsyncHandler 表示异步回调函数
*/
HyperChainAPI hyperchain = new HyperChainAPI(String path);
//建立连接
FutureTask<WebSocketAsyncHandler> futureTask = hyperchain.eventConnect(1, new
WebSocketAsyncHandlerImpl());
```

Java

WebSocketAsyncHandler回调接口函数包括了如下三个方法:

```
public interface WebSocketAsyncHandler {
    void onSuccess();//在连接成功时触发
    void onFailed(Exception e);//在连接发生错误时触发
    void onClose(String message);//在连接关闭时触发
}
//示例中使用的WebSocketAsyncHandlerImpl实现如下
public class WebSocketAsyncHandlerImpl implements WebSocketAsyncHandler {
    private Boolean isOpen;
    private String message;
    private WebSocketState state;
    public Boolean getIsOpen() {
        return this.isOpen;
    }
    public void setIsOpen(Boolean isOpen) {
        this.isOpen = isOpen;
    }
    public String getMessage() {
        return message;
    }
    public WebSocketState getState() {
        return state;
    }
    public void setState(WebSocketState state) {
        this.state = state;
    }
    @Override
    public void onSuccess() {
        this.isOpen = true;
        this.message = "SUCCESS";
        this.state = WebSocketState.OPEN;
    }
    @Override
    public void onFailed(Exception e) {
        this.isOpen = false;
        this.message = "Client onFailure, " + "The reason is :" + e;
        this.state = WebSocketState.FAILURE;
    }
    @Override
    public void onClose(String message) {
        this.isOpen = false;
        this.message = message;
        this.state = WebSocketState.CLOSE;
    }
    enum WebSocketState {
        OPEN(1, "open"),
        FAILURE(2, "failure"),
        CLOSE(3, "close");
        private int code;
```

```
        private String state;
        private WebSocketState(int code, String state) {
            this.code = code;
            this.state = state;
        }
        public int getCode() {
            return code;
        }
        public void setCode(int code) {
            this.code = code;
        }
        public String getState() {
            return state;
        }
        public void setState(String state) {
            this.state = state;
        }
    }
}
```

Java

3.10.2 订阅事件

`String subscribeEvent(WebSocketEvent event, EventListener eventListener, Params param)`

```
/**
* 订阅事件
*
* @param event 类型
* @param eventListener 事件
* @param param 参数
* @return 返回事件的订阅号
*/
FutureTask<WebSocketAsyncHandler> futureTask = hyperchain.eventConnect(1, new
WebSocketAsyncHandlerImpl());
WebSocketAsyncHandlerImpl impl = (WebSocketAsyncHandlerImpl) futureTask.get();
Assert.assertEquals(true, impl.getIsOpen());
Assert.assertEquals(1, impl.getState().getCode());
//发起订阅
String ID = hyperchain.subscribeEvent(WebSocketEvent.BLOCK, new EventListener() {
    @Override
    public void onMessage(String result) {
        System.out.println(result);
        results.add(result + new Date().getTime());
    }
}, new BooleanParam(true));
```

Java

其中现在支持的订阅有三种类型， `block` 区块相关， `systemStatus` 系统状态相关， `logs` 合约event相关，在订阅时传入WebSocketEvent的类型，WebSocketEvent的定义如下：

```
public enum WebSocketEvent {
    BLOCK("block"),
    SYSTEMSTATUS("systemStatus"),
    LOGS("logs");
    private String event;
    private WebSocketEvent(String event) {
        this.event = event;
    }
    public String getEvent() {
        return event;
    }
}
```

Java

第二个参数EventListenr是一个在WebSocket触发onMessage()时的回调函数，即所订阅的消息有返回时会触发listenr，EventListener的定义如下

```
public interface EventListener {
    /**
     * hander the event massage
     *
     * @param result 返回(解密之后)的结果
     */
    void onMessage(String result);
}
```

Java

第三个参数Param则根据订阅的消息类型不同使用不用的Param实现：

- `block` 订阅需要BooleanParam，为 `true` 表示订阅返回区块的具体信息， `false` 表示订阅只返回区块号，具体事例见上述例子
- `systemStatus` 订阅需要SystemStatusParam参数， `systemStatus` 可以通过 `addModules(String module)` 来增加订阅的模块， `addModulesExclude(String moduleExclude)` 来增加排除的模块， `addSubType(String subType)` 来增加订阅的状态信息， `addSubTypeExclude(String subTypeExclude)` 来排除订阅的状态信息， `addErrorCodes(String errorCode)` 来增加订阅的错误码信息， `addErrorCodesExclude(String errorCodeExclude)` 来排除订阅的错误码，具体订阅示例如下：
- `logs` 订阅需要LogsParam参数，通过 `fromBlock(String from)` 来指定起始区块号，为空则默认没有区块下限； `toBlock(String to)` 来指定上限区块号，为空则默认没有上限； `addAddress(String address)` 来指定要订阅的合约地址，为空则默认订阅所有合约； `addTopics(String[] topic)` 来指定合约中具体的event事件，其中event的topic可以通过 `FunctionEecode.encode(String abi)` 来获取某个event的topic，具体使用示例如下：

`systemStatus` 订阅示例:

```
//建立连接
FutureTask<WebSocketAsyncHandler> futureTask = hyperchain.eventConnect(1, new
WebSocketAsyncHandlerImpl());
WebSocketAsyncHandlerImpl impl = (WebSocketAsyncHandlerImpl) futureTask.get();
Assert.assertEquals(true, impl.getIsOpen());
Assert.assertEquals(1, impl.getState().getCode());
//发起订阅
SystemStatusParam systemStatusParam = new SystemStatusParam.Builder()
        .addModules("p2p")
        .addSubTypes("viewchange")
        .build();
String ID = hyperchain.subscribeEvent(WebSocketEvent.SYSTEMSTATUS, new
EventListener() {
    @Override
    public void onMessage(String result) {
        System.out.println("**" + result);
        //results.add(result + new Date().getTime());
    }
}, systemStatusParam);
Thread.sleep(1000);
Assert.assertEquals(34, ID.length());
hyperchain.unSubscribeEvent(ID);
```

Java

`logs` 订阅示例:

```
//创建账户
String accountJson = HyperchainAPI.newAccountRawSM2();
Account account = new Account(accountJson);
final String abi = Utils.readFile("solidity.source/ECDemo.abi");
String bin = Utils.readFile("solidity.source/ECDemo.bin");
//封装交易
Transaction transaction = new Transaction(account.getAddress(), bin, false);
transaction.signWithSM2(accountJson, "");
ReceiptReturn receiptReturn = hyperchain.deployContract(transaction);
//查询部署结果
String contractAddress = receiptReturn.getContractAddress();
//建立连接
FutureTask<WebSocketAsyncHandler> futureTask = hyperchain.eventConnect(1, new
WebSocketAsyncHandlerImpl());
WebSocketAsyncHandlerImpl impl = (WebSocketAsyncHandlerImpl) futureTask.get();
Assert.assertEquals(true, impl.getIsOpen());
Assert.assertEquals(1, impl.getState().getCode());
//发起订阅
LogsParam logsParam = new LogsParam.Builder()
        .addAddress(contractAddress)
        .addTopics(new String[]{FunctionEncode.encodeEvent(abi).get("Event1")})
        .build();
String ID = hyperchain.subscribeEvent(WebSocketEvent.LOGS, new EventListener() {
    @Override
    public void onMessage(String result) {
        System.out.println("WebSocket result: " + result);
        Assert.assertNotEquals(null, result);
```

```
            try {
                System.out.println(FunctionDecode.logDecode(abi, result));
            } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
            }
        }
    }, logsParam);
    Assert.assertEquals(34, ID.length());
    //Thread.sleep(1000 * 60 * 2);
    FuncParamReal paramReal = new FuncParamReal("bytes32", "123");
    String payloadWithLog = FunctionEncode.encodeFunction("triggerEvent", paramReal);
    Transaction transactionWithLog = new Transaction(account.getAddress(),
    contractAddress, payloadWithLog, false);
    transactionWithLog.signWithSM2(accountJson, "");
    hyperchain.invokeContract(transactionWithLog);
    Thread.sleep(2000);
    hyperchain.unSubscribeEvent(ID);
    //合约ECDemo的定义如下:
    contract ECDemo {
        mapping(bytes32=>string) private data1;
        mapping(bytes32=>string) private data2;
        event Event1(bytes32 hash);
        function triggerEvent(bytes32 hash){
            Event1(hash);
        }
        function addData1(bytes32 hash,string data) {
            data1[hash] = data;
        }
        function addData2(bytes32 hash,string data) {
            data2[hash] = data;
        }
        function getData1(bytes32 hash) returns (string data) {
            return data1[hash];
        }
        function getData2(bytes32 hash) returns (string data) {
            return data2[hash];
        }
    }
```

Java

3.10.3 取消订阅

`void unSubscribeEvent(String id)`

```
/**
* 取消订阅
* param id 订阅事件的id
*/
//取消订阅
hyperchain.unSubscribeEvent(ID);
```

Java

3.10.4 关闭连接

`public boolean closeWebSocket()`

关闭与平台之间的WebSocket链接

```
/**
* 关闭连接
*/
//关闭连接
hyperchain.closeWebSocket();
```

Java

3.11 MQ相关接口

3.11.1 注册队列queue

`public String registerQueue(Params meta, int id)`

注册队列订阅event时间的log时，可参考WebSocket的订阅方式（例如topic的定义）。

```
/**
 * 注册queue
 * @param meta 注册信息
 * @param id 节点ID
 *
```

```
 * @return queue名称，exchanger名称
 */
ArrayList<String> array = new ArrayList<String>();
array.add("MQBlock");
array.add("MQLog");
array.add("MQException");
String qname = "gw_node4queue";
MQParam blockInfoParam = new MQParam.Builder().
        setFrom(TEST_FROM2).
        regQueueName(qname).
        regRoutingKeys(array).
        setVerbose(false).
        addModules("consensus").
        marshaltoJSON().
        build(TEST_ACCOUNT_JSON2, TEST_PASSWD);
String result = hyperchain.registerQueue(blockInfoParam, SPEC_ID);
JSONObject object = JSONObject.fromObject(result);
  /** 输出结果为:
  {
      "queueName": "node1queue",
      "exchangerName": "global_fa34664e_1529591668938108811"
  } **/
```

Java

3.11.2 删除queue

`public String unregisterQueue(String qname, String exchName, String from, String signature, int id)`

```
/**
 * 删除queue
 * @param qname queue的名称
 * @param exchName exchanger名称
 * @param from 账户地址
 * @param signature 签名
 * @param id 节点ID
 *
 * @return queue中剩余消息数目，是否删除成功，删除过程中的错误信息
 */
String qname = "gw_node4queue";
String exchName = "global_34d29974_1534129779744674886";
String signature = new UnRegParam(qname, exchName).signWithSM2(TEST_ACCOUNT_JSON2,
TEST_PASSWD);
System.out.println(signature);
String result = hyperchain.unregisterQueue(qname, exchName, TEST_FROM2, signature,
SPEC_ID);
JSONObject object = JSONObject.fromObject(result);
 /**
  输出结果为: {"count":0, "success":true, "error":"null"}
  **/
```

Java

3.11.3 查询指定节点应注册的所有的queue名称

`String getAllQueueNames(int id)`

```
/**
 * 查询指定节点应注册的所有的queue名称
 * @param id 节点ID
 *
 * @return queue名称列表的json字符串
 */
 public String testGetAllQName(int id) {
     String result = hyperchain.getAllQueueNames(id);
     return result;
 }
 /**
  输出结果为: ["node1queue"]
  **/
```

Java

3.11.4 通知平台rabbitmq-broker正常工作

`String informNormal(String brokerUrl, int id)`

```
/**
 * 通知平台rabbitmq-broker正常工作
 * @param brokerUrl broker的完整url
 * @param id 节点ID
```

```
     *
     * @return 连接建立结果，过程中的error
     */
    public void testInformNormal(String brokerUrl, int id) throws Exception {
        // 空字符串表示使用平台默认url
        String result = hyperchain.informNormal(brokerUrl, id);
        System.out.println(result);
    }
    /**
     输出结果为：{"success":true,"error":null}
     **/
```

Java

3.11.5 查询指定节点exchanger的名称

`String getExchName(int id)`

```
    /**
     * 查询指定节点exchanger的名称
     * @param id 节点ID
     *
     * @return exchanger名称
     */
    public void testGetExchangerName(int id) throws Exception {
        String result = hyperchain.getExchName(id);
        System.out.println(result);
    }
    /**
     输出结果为："global_fa34664e_1530265355500005853"
     **/
```

Java

3.11.6 删除指定节点的exchange

`public String deleteExch(String name, int id)`

```
    /**
     * 删除指定节点exchanger
     * @param name 要删除的exchanger名称
     * @param id 节点ID
     *
     * @return exchanger名称
     */
    String exchName = "global_34d29974_1533636890847100123";
    String result = hyperchain.deleteExch(exchName, SPEC_ID);
    JSONObject object = JSONObject.fromObject(result);
```

Java

3.12 工具类相关方法

3.12.1 读入文件为String

`String readFile(String path)`

```
    /**
    * 读入文件
    * param path 文件路径 相对路径绝对路径皆可
    */
    String file = Utils.readFile("hpc.properties");
```

Java

若传入为绝对路径，则直接将文件流返回为String，若传入为相对路径，则默认在classpath路径下获取。

3.12.2 十六进制转byte数组

`byte[] hexStringToByteArray(String s)`

```
    /**
    * 十六进制转byte数组
    * param s 十六进制字符串
    */
    byte[] bytes = Utils.hexStringToByteArray("0x3214");
```

Java

3.12.3 将Java合约转为target bin(构造函数无参)

`String getTargetBinFromPath(String path) throws IOException`

```
/**
 * 将Java合约转为target bin 合约无参
 *
 * @param path 编译好的Java合约的路径 支持绝对路径和相对路径(相对classpath)
 * @return Target bin
 * @throws IOException error
 */
String path = "javaContractExample/contract03";
String target = Utils.getTargetBinFromPath(path);
```

Java

3.12.4 将Java合约转为target bin(构造函数有参)

`String getTargetBinFromPath(String path, String ... params)`

```
/**
 * 将Java合约转为target bin 合约有参
 *
 * @param path 编译好的Java合约的路径 支持绝对路径和相对路径(相对classpath)
 * @param params 构造函数参数
 * @return Target bin
 * @throws IOException error
 */
String path = "javaContractExample/contract03";
String target = Utils.getTargetBinFromPath(path, "123");
```

Java

3.12.5 交易验签方法(接收Hex tring格式的公钥)

`public static boolean checkSign(String data, String signature, String puKey)`

```
/**
 * check the transaction signature
 *
 * @param data      the sign data
 * @param signature sig
 * @param puKey     user public key, is a hex String
 * @return bool
 * @throws IOException -
 */
ECPriv ecPriv = HyperchainAPI.newAccount();
Transaction transaction = new Transaction(ecPriv.address(), ecPriv.address(),
"0xa0a0", false, VMType.EVM);
transaction.sign(ecPriv);
Boolean success = Utils.checkSign(transaction.getHash(),
transaction.getSignature(), Hex.toHexString(ecPriv.getPublicKeyByte()));
```

Java

3.12.6 交易验签接口(接收Hex bytes格式的公钥)

`public static boolean checkSign(String data, String signature, byte[] puKey)`

```
/**
* check the transaction signature
*
* @param data      the sign data
* @param signature sig
* @param puKey     user public key, is a hex String
* @return bool
* @throws IOException -
*/
ECPriv ecPriv = HyperchainAPI.newAccount();
Transaction transaction = new Transaction(ecPriv.address(), ecPriv.address(),
"0xa0a0", false, VMType.EVM);
transaction.sign(ecPriv);
Boolean success = Utils.checkSign(transaction.getHash(),
transaction.getSignature(),ecPriv.getPublicKeyByte());
```

Java

3.12.7 SM2加密接口

`public static byte[] encryptWithSm2(String pubKey, byte[] data)`

```java
/**
* encrypt with sm2
*
* @param pubKey sm2 public key
* @param data   data need encrypt
* @return encrypted data
* @throws IOException -
*/
String ecPriv = HyperchainAPI.newAccountRawSM2();
Account account = new Account(ecPriv);
String ed = "fdsafnaiofniaofhuuasifuhnasiofhasnfhasf";
byte[] encryted = Utils.encryptWithSm2(account.getPublicKey(), ed.getBytes());
```

Java

3.12.8 SM2解密接口

`public static byte[] decryptWithSm2(String privateKey, byte[] encryptedData)`

```java
/**
* decrypted with sm2
*
* @param privateKey    pri
* @param encryptedData data
* @return data
* @throws IOException -
*/
String ecPriv = HyperchainAPI.newAccountRawSM2();
Account account = new Account(ecPriv);
String ed = "fdsafnaiofniaofhuuasifuhnasiofhasnfhasf";
byte[] encryted = Utils.encryptWithSm2(account.getPublicKey(), ed.getBytes());
byte[] decryted = Utils.decryptWithSm2(account.getPrivateKey(), encryted);
Assert.assertEquals(ed, new String(decryted));
```

Java

3.12.9 SM4加密接口

`public static String encryptSM4(byte[] plainText, String secretKey)`

```java
/**
*  SM4 encrypt method
*
* @param plainText _
* @param secretKey a 128 bit secretKey
* @return cipherText
*/
String plainText = "加密原文";
String cipherText = Utils.encryptSM4(plainText.getBytes(), "1111111111111112");
```

Java

3.12.10 SM4解密接口

`public static byte[] decryptSM4(String cipherText, String secretKey)`

```java
/**
* SM4 decrypt method
*
* @param cipherText _
* @param secretKey a 128 bit secretKey
* @return plainText
*/
String plainText = "加密原文";
String cipherText = Utils.encryptSM4(plainText.getBytes(), "1111111111111112");
byte[] plainTextRes = Utils.decryptSM4(cipherText, "1111111111111112");
```

Java

3.13 TCert相关方法

使用TCert要设置SDK配置文件中tcert的开关为true，配置好公私钥文件。公私钥文件可以通过certgen获取，openssl获取，或者直接使用info上下载证书时附带的unique.pub和unique.priv。

使用openssl可以通过如下的命令：

```
openssl ecparam -name prime256v1 -out eccparam.pem
openssl ecparam -genkey -in eccparam.pem -out unique.priv
```

```
openssl ec -in unique.priv -pubout -out unique.pub
#注意，生成unique.priv后，打开该文件，删除前三行的内容，保留-----BEGIN EC PRIVATE KEY-----
及之后的内容即可。
#生成的unique.pub和unique.priv就是相应的公私钥对
```

JavaScript

3.13.1 获取自定义TCert接口

`public TCert getTCert(String publicKeyPath, String privKeyPath, int id) t hrows Exception`

```
/**
 * 取得TCert
 *
 * @param publicKeyPath 公钥
 * @param privKeyPath 私钥
 * @param id 节点id
 * @return TCert
 */
TCert tCert = hyperchain.getTCert("certs/unique.pub", "certs/unique.priv", 1);
```

Java

3.13.2 设置自定义TCert发请求

指定某个节点的交易使用自定义请求的TCert来发送

`public void setTCert(int id, TCert tCert)`

```
/**
 * 取得TCert
 *
 * @param id 节点id
 * @param tCert TCert
 */
hyperchain.setTCert(1, tCert);
```

Java

3.13.3 取消自定义TCert发请求

取消指定某个节点的交易使用自定义请求的TCert来发送

`public void unSetTCert(int id)`

```
/**
 * 取得TCert
 *
 * @param id 节点id
 */
hyperchain.unSetTCert(1);
```

Java

3.14 Reader可视化

3.14.1 radar 合约数据导出接口

`public ReceiptReturn listenContract (String sourceCode, String addr, int index)`

```
/**
* 通知radar监听和导出已部署的智能合约数据
* 保证配置文件中已启用radar，不然无法正常调用。
*
* @param sourceCode 合约源码
* @param addr 合约地址
* @param index 需要导出数据的节点的ID
*/
String SourceCode = Utils.readFile("solidity.source/c1.sol");
int id = 1;
//deploy Contract
String contrtactAddress = deployContract("Bank", SourceCode);
// notify radar to listen to contract
hyperchain.listenContract(SourceCode, contrtactAddress, id);
```

Java

3.15 隐私交易

核心类

为了使用hyperchain的隐私交易的功能，需要使用**PrivateTransaction.java**中定义的PrivateTransaction类。

而**PrivateTransaction**除了构造函数和**Transaction**不同，在创建完对象之后，具体的使用方式和之前（非隐私）保持一致，没有额外的学习成本。

类定义

```
public class PrivateTransaction extends Transaction
```

Java

构造函数

```
public PrivateTransaction(String from, String to, String payload, boolean
simulate, String[] participants) throws TxException
```

Java

说明：构造一个用来调用EVM合约的PrivateTransaction对象。

参数说明：

- String from：用户地址
- String to：合约地址
- String payload：合约调用荷载
- boolean simulate：是否是模拟交易
- String[] participants：该隐私交易参与方的地址数组

```
public PrivateTransaction(String from, String to, String payload, boolean
simulate, VMType type, String[] participants) throws TxException
```

Java

说明：调用合约，指定合约类型

参数说明：

- String from：用户地址
- String to：合约地址
- String payload：合约调用荷载
- boolean simulate：是否是模拟交易
- VMType type：虚拟机类型
- String[] participants：该隐私交易参与方的地址数组

```
public PrivateTransaction(String from, String to, String payload, int opcode,
String[] participants) throws TxException
```

Java

说明：合约管理，更新合约，EVM

参数说明：

- String from：用户地址
- String to：合约地址
- String payload：新合约字节码
- int opcode：合约操作码
- String[] participants：该隐私交易参与方的地址数组

```
public PrivateTransaction(String from, String to, String payload, int opcode,
VMType type, String[] participants) throws TxException
```

Java

说明：合约管理，更新合约，指定合约类型

参数说明：同上

```
public PrivateTransaction(String from, String to, int opcode, String[]
participants) throws TxException
```

Java

说明：合约管理，非更新合约，EVM。

参数说明：同上。

```
public PrivateTransaction(String from, String to, int opcode, VMType type,
String[] participants) throws TxException
```

Java

说明：合约管理，非更新合约，指定合约类型。

参数说明：同上。

```
public PrivateTransaction(String from, String to, long value, boolean simulate,
String[] participants) throws TxException
```

Java

说明：普通转账交易，EVM。

参数说明：

- long value：转账金额。

```
public PrivateTransaction(String from, String to, long value, boolean simulate,
VMType type, String[] participants) throws TxException
```

Java

说明：普通转账交易，指定合约类型。

参数说明：同上。

```
public PrivateTransaction(String from, String payload, boolean simulate, String[]
participant, FuncParamReal... params) throws FunctionParamException, TxException
```

Java

说明：部署合约（构造函数有参），EVM。

参数说明：

- FuncParamReal... params：合约构造方法参数。

```
public PrivateTransaction(String from, String payload, boolean simulate, VMType
type, String[] participants, FuncParamReal... params) throws
FunctionParamException, TxException
```

Java

说明：部署合约（构造函数有参），指定合约类型。

参数说明：同上。

```
public PrivateTransaction(String from, String payload, boolean simulate, String[]
participants) throws TxException
```

Java

说明：部署合约（构造函数无参），EVM。

参数说明：

- String payload：合约字节码。

```
public PrivateTransaction(String from, String payload, boolean simulate, VMType
type, String[] participants) throws TxException
```

Java

说明：部署合约（构造函数无参），指定合约类型。

参数说明：同上。

签名

```
public void sign(String accountJSON, String passwd) throws Exception
```

Java

说明：ECDSA签名。

参数说明：

- String accountJSON：ECDSA账户accountJSON。
- String passwd：账户密码。

```
public void sign(ECPriv ecPriv) throws Exception
```

Java

说明：ECDSA签名。

参数说明：

- ECPriv ecPriv：账户私钥结构体。

支持的操作

由于重载较多，这里就不一一列出所有支持的操作了。

以下假设用户熟悉非隐私交易（即Transaction对象）的使用方法：

对于**PrivateTransaction**的使用，在HyperchainAPI中，所有RPC方法签名中有接收**Transaction**类型参数的，均可以直接传入**PrivateTransaction**作为参数使用，没有其他额外参数和学习成本，比如部署合约（deployContract）。

对于那些RPC方法中不接受**Transaction**类型参数的，但和隐私交易相关的接口的，我都提供了重载，需要额外提供一个参数**boolean isPrivateTx**，来标记该次交易为一个隐私交易，比如查询隐私交易回执（getTransactionReceipt）。

第四章 说明

1. 查询批次:所有的交易都需要提供一个查询批次信息,可以随意指定,主要是用于多批次同时查询,在返回值中也会返回对应批次的数据,如果是单次查询,可以随意指定该值(int)即可。
2. 账户存储:hyperchain 会将用户账户地址加密存储,需要用户提供密码对加密存储的私钥文件进行解密,所以在发送交易相关的接口当中,需要对交易进行签名,所以需要传入账户加密存储文件(json格式字符串)和密码。
3. Namespace说明:在创建hyperchain对象的时候默认读取的是配置文件中的namespace字段，如果配置文件中没有配置，那么默认的namespace为global，一个hyperchain对象对应的是一个namespace，只能将消息发送到自己所在的namespace，所以如果要切换namespace需要新建不同的hyperchain对象（参考Hyperchain构造方法和ApiProperities的setNamespace方法）。
4. Java合约:Java合约部署传入的是编译好的合约的路径，与solidity合约不同Java合约是需要在本地编译的，需要用户自己编译后得到。Java合约的部署调用接口与之前是一样的，不同的是new Transaction对象时的VMType不同以及个别参数的编码函数不同。
5. 返回code说明:

rawcode：对于调用结果，都拥有rawcode，含义如下：

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	请求的数据不存在
-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(出现此错误，请考虑是否限流导致，将使SDK获取不到信息)
-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
-32096	http请求处理超时
-32097	Hypercli用户令牌无效
-32098	请求未带cert或者错误cert导致认证失败
-32099	请求tcert失败
-9995	请求失败(通常是请求体过长)
-9996	请求失败(通常是请求消息错误)
-9997	异步请求失败
-9998	请求超时
-9999	获取平台响应失败

ReceiptReturn的code（不推荐使用，推荐使用rawcode）含义如下：

code	含义
200	请求成功
301	交易执行没有返回值或者传入参数错误
401	交易没有被确认
402	交易调用执行错误
403	签名失败
404	部署失败
500	其他服务器内部错误
900	未知错误

6. Java与Solidity数据类型的对应关系:

在调用有参合约方法时，需要使用FuncParamReal类的构造函数对参数进行封装，其中type为solidity中的数据类型，param为java中的数据类型，两者类型需相应的匹配

```java
public FuncParamReal(String type, Object param);
```

Java

其对应关系为：(solidity => java)

6.1、变长byte数组bytes => byte数组

```java
new FuncParamReal("bytes", "str".getBytes());
```

Java

6.2、定长byte数组bytes1-32(步长为1增加) => 字符串String

```java
new FuncParamReal("bytes1", "A");
new FuncParamReal("bytes2", "AB");
new FuncParamReal("bytes3", "ABC");
//...依次类推
```

Java

6.3、字符串string => 字符串String

```java
new FuncParamReal("string", "str2");
```

Java

6.4、int、uint、int8-256(步长为8增加)、uint8-256(步长为8增加) => 相应长度的整型

```java
new FuncParamReal("int8", -128);
new FuncParamReal("int8", 127);
new FuncParamReal("uint8", 0);
new FuncParamReal("uint8", 255);
```

Java

6.5、布尔值bool => 布尔值boolean

```java
new FuncParamReal("bool", true);
```

Java

6.6、地址address => 字符串String

```java
new FuncParamReal("address", "0xedb84fd7208f372f4e5d64abc419fef416394f9b");
```

Java

6.7、数组 => 数组

```java
new FuncParamReal("int[3]", new int[]{1,2,3});
new FuncParamReal("int8[3]", new int[]{-128,0,127});
new FuncParamReal("uint[3]", new int[]{1,2,3});
new FuncParamReal("uint8[3]", new int[]{0,100,255});
new FuncParamReal("bool[3]", new boolean[]{true,false,true});
new FuncParamReal("address[2]", new String[]
{"0xedb84fd7208f372f4e5d64abc419fef416394f9b",
        "0xfdb84fd7208f372f4e5d64abc419fef416394f9c"});
// 字符串数组、bytes数组为solidity当前版本的实验性新特性，暂不支持
```

Java

第五章 样例数据

样例账户（DES）

账户地址 from :

`0x0b110ed15f21a3ec73b051b59864ed6dec687ad9`

from 加密存储的私钥字符串:

```javascript
{"address":"0b110ed15f21a3ec73b051b59864ed6dec687ad9","algo":"0x02","encrypted":"787cf33e169a914f6d6bd2f75f0a9c2fc6e746f2b9e81dfa92b29bb767759bb8c6f431ea0e0e22ac","version":"1.0"}
```

JavaScript

账户地址 to :

```
0xf4d69ac5dc63869de4dc6add25690ad404641bf5
```

to 加密存储的私钥字符串:

```
{"address":"f4d69ac5dc63869de4dc6add25690ad404641bf5","algo":"0x02","encrypted":"e
86ba44da8b325e4c177c3ef4d2b7e68a23bc8e1249d7efcc03875c3087935f4c6f431ea0e0e22ac","
version":"1.0"}
```

JavaScript

Solidity 源代码

```
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; } }
```

JavaScript

Solidity bin

```
0x6060604052600080546 3ffffffff19168155609 e908190601e90 396000f3606060405260
e060020a60003504633ad14af381146030578 063569c5f6d146056578 063d09de08a14607c57
5b6002565b34600257600 0805463ffffffff81166004350 16024350163 ffffffff199091161790
555b005b34600257600 05463ffffffff 166040805163 ffffffff90921682525190 81900360200
190f35b34600 25760546000 805463ffffffff 19811663ffffffff909116600101 17905556
```

JavaScript

Solidity 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"}]
```

JavaScript