

智臻链

京东数科出品

JD Chain

快速入门指南 **v1.0**

2019.03

目 录

1 简介	1
1.1 准备	1
2 数据快速上链	2
2.1 服务连接	2
2.2 用户注册	2
2.3 数据账户注册	3
2.4 写入数据	4
2.5 查询数据	5
2.6 合约发布	8
2.7 合约执行	9
3 快速开发合约代码	10
3.1 开发	10
3.2 编译	10
3.3 发布	12
3.4 执行	13

1 简介

本指南适用于已经成功构建基本开发环境，且希望快速进行智能合约开发的用户。

用户在 `contract-compile` 模块下，首先根据 `com.jd.blockchain.contract` 包下的合约样例，编写新合约。然后执行 `mvn clean package` 进行编译，在输出路径中找到对应的 `xxx.jar` 合约文件。再将此合约压缩包在测试链上部署并执行。

快速使用样例下载地址：<https://github.com/blockchain-jd-com/jdchain-starter.git>，在此工程的 `readme.txt` 文件中也有相关说明。

1.1 准备

用户可直接通过 maven 中央库来获取 jar。

1.1.1 基于 SDK “数据快速上链” 方式的 maven 坐标

```
<dependency>
  <groupId>com.jd.blockchain</groupId>
  <artifactId>sdk-client</artifactId>
  <version>0.8.2.RELEASE</version>
</dependency>
```

1.1.2 基于“快速开发合约代码”方式的 maven 坐标

```
<dependency>
  <groupId>com.jd.blockchain</groupId>
  <artifactId>contract-model</artifactId>
  <version>0.8.2.RELEASE</version>
</dependency>
```

2 数据快速上链

2.1 服务连接

```
//创建服务代理

public static BlockchainKeyPair CLIENT_CERT = BlockchainKeyGenerator.getInstance().generate();
final String GATEWAY_IP = "127.0.0.1";
final int GATEWAY_PORT = 80;
final boolean SECURE = false;

GatewayServiceFactory serviceFactory = GatewayServiceFactory.connect(GATEWAY_IP,
GATEWAY_PORT, SECURE,
    CLIENT_CERT);

// 创建服务代理;

BlockchainService service = serviceFactory.getBlockchainService();
```

2.2 用户注册

```
// 创建服务代理;

BlockchainService service = serviceFactory.getBlockchainService();

// 在本地定义注册账号的 TX;

TransactionTemplate txTemp = service.newTransaction(ledgerHash);
SignatureFunction signatureFunction =
asymmetricCryptography.getSignatureFunction(CryptoAlgorithm.ED25519);
CryptoKeyPair cryptoKeyPair = signatureFunction.generateKeyPair();
BlockchainKeyPair user = new BlockchainKeyPair(cryptoKeyPair.getPubKey(),
cryptoKeyPair.getPrivKey());

txTemp.users().register(user.getIdentity());

// TX 准备就绪;
PreparedTransaction prepTx = txTemp.prepare();
// 使用私钥进行签名;
CryptoKeyPair keyPair = getSponsorKey();
prepTx.sign(keyPair);

// 提交交易;

prepTx.commit();
```

2.3 数据账户注册

```
// 创建服务代理；
BlockchainService service = serviceFactory.getBlockchainService();
// 在本地定义注册账号的 TX；
TransactionTemplate txTemp = service.newTransaction(ledgerHash);
SignatureFunction signatureFunction =
asymmetricCryptography.getSignatureFunction(CryptoAlgorithm.ED25519);
CryptoKeyPair cryptoKeyPair = signatureFunction.generateKeyPair();
BlockchainKeyPair dataAccount = new BlockchainKeyPair(cryptoKeyPair.getPubKey(),
cryptoKeyPair.getPrivKey());

txTemp.dataAccounts().register(dataAccount.getIdentity());

// TX 准备就绪；
PreparedTransaction prepTx = txTemp.prepare();
// 使用私钥进行签名；
CryptoKeyPair keyPair = getSponsorKey();
prepTx.sign(keyPair);

// 提交交易；
prepTx.commit();
```

2.4 写入数据

```
// 创建服务代理；
BlockchainService service = serviceFactory.getBlockchainService();

HashDigest ledgerHash = getLedgerHash();
// 在本地定义注册账号的 TX；
TransactionTemplate txTemp = service.newTransaction(ledgerHash);

// -----
// 将商品信息写入到指定的账户中；
// 对象将被序列化为 JSON 形式存储，并基于 JSON 结构建立查询索引；
String commodityDataAccount = "GGhhreGeasdfasfUUfeh9932lkae99ds66jf==";
Commodity commodity1 = new Commodity();
txTemp.dataAccount(commodityDataAccount).set("ASSET_CODE",
commodity1.getCode().getBytes(), -1);
// TX 准备就绪；
PreparedTransaction prepTx = txTemp.prepare();

String txHash = ByteArray.toBase64(prepareTx.getHash().getBytes());
// 使用私钥进行签名；
CryptoKeyPair keyPair = getSponsorKey();
prepTx.sign(keyPair);

// 提交交易；
prepTx.commit();
```

2.5 查询数据

注：详细的查询可参考模块 `sdk-samples` 中 `SDK_GateWay_Query_Test` 相关测试用例

```
// 创建服务代理；
BlockchainService service = serviceFactory.getBlockchainService();

// 查询区块信息；
// 区块高度；
long ledgerNumber = service.getLedger(LEDGER_HASH).getLatestBlockHeight();
// 最新区块；
LedgerBlock latestBlock = service.getBlock(LEDGER_HASH, ledgerNumber);
// 区块中的交易的数量；
long txCount = service.getTransactionCount(LEDGER_HASH, latestBlock.getHash());
// 获取交易列表；
LedgerTransaction[] txList = service.getTransactions(LEDGER_HASH, ledgerNumber, 0, 100);
// 遍历交易列表
for (LedgerTransaction ledgerTransaction : txList) {
    TransactionContent txContent = ledgerTransaction.getTransactionContent();
    Operation[] operations = txContent.getOperations();
    if (operations != null && operations.length > 0) {
        for (Operation operation : operations) {
            operation = ClientOperationUtil.read(operation);
            // 操作类型：数据账户注册操作
            if (operation instanceof DataAccountRegisterOperation) {
                DataAccountRegisterOperation daro = (DataAccountRegisterOperation) operation;
                BlockchainIdentity blockchainIdentity = daro.getAccountID();
            }
            // 操作类型：用户注册操作
            else if (operation instanceof UserRegisterOperation) {
                UserRegisterOperation uro = (UserRegisterOperation) operation;
                BlockchainIdentity blockchainIdentity = uro.getUserID();
            }
            // 操作类型：账本注册操作
            else if (operation instanceof LedgerInitOperation) {
                LedgerInitOperation ledgerInitOperation = (LedgerInitOperation) operation;
                LedgerInitSetting ledgerInitSetting = ledgerInitOperation.getInitSetting();

                ParticipantNode[] participantNodes = ledgerInitSetting.getConsensusParticipants();
            }
        }
    }
}
```

```

// 操作类型：合约发布操作
else if (operation instanceof ContractCodeDeployOperation) {
    ContractCodeDeployOperation ccdo = (ContractCodeDeployOperation) operation;
    BlockchainIdentity blockchainIdentity = ccdo.getContractID();
}
// 操作类型：合约执行操作
else if (operation instanceof ContractEventSendOperation) {
    ContractEventSendOperation ceso = (ContractEventSendOperation) operation;
}
// 操作类型：KV 存储操作
else if (operation instanceof DataAccountKVSetOperation) {
    DataAccountKVSetOperation.KVWriteEntry[] kvWriteEntries =
        ((DataAccountKVSetOperation) operation).getWriteSet();
    if (kvWriteEntries != null && kvWriteEntries.length > 0) {
        for (DataAccountKVSetOperation.KVWriteEntry kvWriteEntry : kvWriteEntries) {
            BytesValue bytesValue = kvWriteEntry.getValue();
            DataType dataType = bytesValue.getType();
            Object showVal = ClientOperationUtil.readValueByBytesValue(bytesValue);
            System.out.println("writeSet.key=" + kvWriteEntry.getKey());
            System.out.println("writeSet.value=" + showVal);
            System.out.println("writeSet.type=" + dataType);
            System.out.println("writeSet.version=" + kvWriteEntry.getExpectedVersion());
        }
    }
}
}
}
}

// 根据交易的 hash 获得交易；注：客户端生成 PrepareTransaction 时得到交易 hash；
HashDigest txHash = txList[0].getTransactionContent().getHash();
Transaction tx = service.getTransactionByContentHash(LEDGER_HASH, txHash);

```



```
// 获取数据;

String commerceAccount = "GGhhreGeasdfasfUUfeh9932lkae99ds66jf==";
String[] objKeys = new String[] { "x001", "x002" };
KVDataEntry[] kvData = service.getDataEntries(LEDGER_HASH, commerceAccount, objKeys);
long payloadVersion = kvData[0].getVersion();

// 获取数据账户下所有的 KV 列表
KVDataEntry[] kvData = service.getDataEntries(ledgerHash, commerceAccount, 0, 100);
if (kvData != null && kvData.length > 0) {
    for (KVDataEntry kvDatum : kvData) {
        System.out.println("kvData.key=" + kvDatum.getKey());
        System.out.println("kvData.version=" + kvDatum.getVersion());
        System.out.println("kvData.type=" + kvDatum.getType());
        System.out.println("kvData.value=" + kvDatum.getValue());
    }
}
```

2.6 合约发布

```
// 创建服务代理；
BlockchainService service = serviceFactory.getBlockchainService();

// 在本地定义 TX 模板
TransactionTemplate txTemp = service.newTransaction(ledgerHash);
// 合约内容读取
byte[] contractBytes = FileUtils.readBytes(new File(CONTRACT_FILE));
// 生成用户
BlockchainIdentityData blockchainIdentity = new BlockchainIdentityData(getSponsorKey().getPubKey());

// 发布合约
txTemp.contracts().deploy(blockchainIdentity, contractBytes);

// TX 准备就绪；
PreparedTransaction prepTx = txTemp.prepare();

// 使用私钥进行签名；
CryptoKeyPair keyPair = getSponsorKey();
prepTx.sign(keyPair);
// 提交交易；
TransactionResponse transactionResponse = prepTx.commit();
assertTrue(transactionResponse.isSuccess());
// 打印合约地址
System.out.println(blockchainIdentity.getAddress().toBase58());
```

2.7 合约执行

```
// 创建服务代理;
BlockchainService service = serviceFactory.getBlockchainService();

// 在本地定义 TX 模板
TransactionTemplate txTemp = service.newTransaction(ledgerHash);

// 合约地址
String contractAddressBase58 = "";

// Event
String event = "";
// args (注意参数的格式)
byte[] args = "20##30##abc".getBytes();

// 提交合约执行代码
txTemp.contractEvents().send(contractAddressBase58, event, args);
// TX 准备就绪;
PreparedTransaction prepTx = txTemp.prepare();
// 生成私钥并使用私钥进行签名;
CryptoKeyPair keyPair = getSponsorKey();
prepTx.sign(keyPair);
// 提交交易;
TransactionResponse transactionResponse = prepTx.commit();
assertTrue(transactionResponse.isSuccess());
```

3 快速开发合约代码

3.1 开发

入门样例可参照：`com.jd.blockchain.contract.AssetContract3`，合约类实现 `EventProcessingAwire` 接口，同时在合约的入口方法上添加注解：`@ContractEvent(name = "xxx")`，形参为：`ContractEventContext eventContext`。格式如下：

```
public class AssetContract3 implements EventProcessingAwire{
    @ContractEvent(name = "xxx")
    public void test1(ContractEventContext eventContext){
    }
}
```

合约中可以通过 `ContractEventContext` 对象来调用账本中的相关方法，例如：

```
BlockchainAccount holderAccount = eventContext.getLedger().getAccount(currentLedgerHash(), assetHolderAddress);
```

3.2 编译

编译工作在 `contract-compile` 工程中进行。在控制台中执行：

```
mvn clean package
```

来直接编译生成所需的合约压缩包。

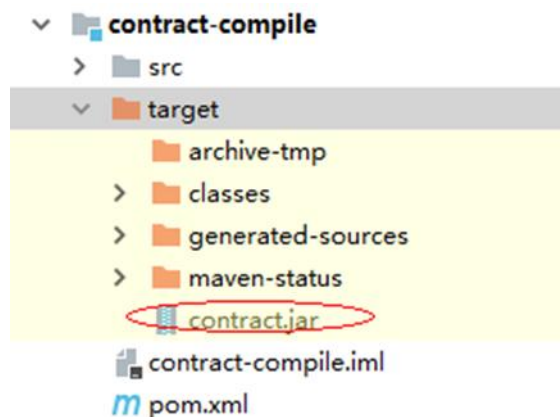
注意修改 `pom.xml` 文件中 `<filename>` 和 `<mainClass>` 这两个属性：

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <finalName>contract</finalName>
    <appendAssemblyId>>false</appendAssemblyId>
    <archive>
      <manifest>
        <mainClass>com.jd.blockchain.contract.AssetContract4</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

```
</archive>
<descriptorRefs>
  <descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
</configuration>
<executions>
  <execution>
    <id>make-assembly</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

3.3 发布

编译完成之后，根据如上的配置，会在 **target** 目录下生成对应的合约压缩文件：**contract.jar**。



3.3.1 sys-contract.properties

#常规使用;

ownerPubPath=xxx/conf/jd-com.pub

ownerPrvPath=xxx/conf/jd-com.priv

ownerPassword=xxx/conf/ownerPassword.txt

ledgerHash=6Gw3cK4uazegy4HjoaM81ck9NgYLN0KyBMb7a1TK1jt3d

host=192.168.151.45

port=8081

#合约使用如下;

event = issue-asset

chainCodePath=xxx/AssetContract3.contract

contractArgs=10##4##abc

3.3.2 发布与执行方法

运行 `jdchain-starter/src/test/.../IntegrationTest.java` 中的如下方法来发布和执行合约:

```
one_deploy_exe_contract_on_test_gateway()
```

具体代码如下:

```
/**
 * 在测试链上仅发布和执行合约;
 */
private void deploy_exe_contract_on_test_gateway(){
    //then exe the contract;
    //由于合约发布之后需要后台进行共识处理，需要一定的时间消耗，先休息 1 秒钟之后再执行;
    try {
        Thread.sleep(1000L);
        boolean deployResult = ContractDeployExeUtil.instance.deploy(host, port,
ledger,ownerPubPath, ownerPrvPath, ownerPassword, chainCodePath,contractPub);
        System.out.println("deployResult="+deployResult);
        Thread.sleep(2000L);
        boolean exeResult = false;
        exeResult = ContractDeployExeUtil.instance.exeContract(ledger,ownerPubPath, ownerPrvPath,
ownerPassword,eventName,contractArgs);
        System.out.println("execute the contract,result= "+exeResult);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

```
// 发布完成之后，会在控制台中生成如下信息：
contract's address=5SmDBPXfXoSsJmfJskkpeVbZkxfkawqWE9CE
deployResult=true
```

3.4 执行

执行即如上的方法：

```
ContractDeployExeUtil.instance.exeContract(ledger,ownerPubPath, ownerPrvPath,
ownerPassword,eventName,contractArgs);
```

在执行完成之后，可以在 API 接口中查询相关的信息。

3.4.1 根据合约地址查询

访问格式如下：

<http://192.168.151.45:7080/ledgers/6Gw3cK4uazegy4HjoaM81ck9NgYlNoKyBMb7a1TK1jt3d/contracts/5SmDBPXfXoSsJmfJskkpeVbZkxfkawqWE9CE>

结果:

```
{
  "data": {
    "address": "5SmDBPXfXoSsJmfJskpeVbZkxfkawqWE9CE",
    "chainCode":
      "UEsDBBQACAgIAACAIE0AAAAAAAAAAAAAAAAAJAAQATUVUQS1JTkYv/soAAAMAUEsHCAAAAAAC
      AAAAAAAAAAFBLAwQUAAgICAAAgJRNAAAAAAAAAAAAAAAAAFAAAAE1FVEEtSU5GL01BTklGRVNU
      Lk1G803My0xLLS7RDUstKs7Mz7NSMNQz4OVyLkpNLEIN0XWqBAIY6BnEG5qYKGj4FyUm56QqOOcXFeQ
      XJZYA1WvycvFyAQBQSwcInnx2U0QAAABFAAAAUEsDBBQACAgIAACAIE0AAAAAAAAAAAAAAAAATA
      AAAY29udHJhY3QucHJvcGVydGllc1OOKTU1MbCIKTU1MbCIKTU1MbCIKTU1MbCIKTU1MbCIKTU1Mb
      VwSU1WMDJQMDSzMjCwMjBUcA4OAfINLXi5kvPzSooSk0tsk/Nz9bJS9JJy8pOzkzMSM/P0YFJ6jsXFqSXOUJ
      4xLxcAUEsHCBckArNrAAAAegAAAFBLAQIUABQACAgIAACAIE0AAAAAAGAAAAAAAAAJAAQAAAAA
      AAAAAAAAAAAAAABNRVRBLUIORi/+ygAAUEsBAhQAFAAICAgAAICUTZ58dINEAAAAARQAAABQAA
      AAAAAAAAAAAAAAPQAAAE1FVEEtSU5GL01BTklGRVNUlk1GUEsBAhQAFAAICAgAAICUTRckArNr
      AAAAegAAABMAAAAAAAAAAAAAAAAAAAwAAAGNvbnRyYWNOLnByb3BlcnRpZXNQSwUGAAAAAA
      MAAwC+AAAAbwEAAAAA",
    "chaincodeProof": {
      "dataHash": {
        "value": "6LwUAHD8zZbANkdJL8bBHpYM2BYduZpQXzaV5T92XxyLL"
      },
      "dataNode": {
        "key": "CHAIN-CODE",
        "level": 0,
        "nodeHash": {
          "value": "6LwUAHD8zZbANkdJL8bBHpYM2BYduZpQXzaV5T92XxyLL"
        },
        "sN": 0,
        "version": 0
      },
      "levels": 1,
      "rootHash": {
        "value": "65WEJwZWpdpv5A8BWDqsJQxW33qBR16q4Bx8tzUTbSUWv"
      },
      "sN": 0
    },
    "chaincodeVersion": 0,
    "pubKey": {
      "value": "mayaPm1C7opEhSyRtmH7rkbmXAX2MvrzUguFkgcJ7mpfQY"
    },
    "rootHash": {
      "value": "65WEJwZWpdpv5A8BWDqsJQxW33qBR16q4Bx8tzUTbSUWv"
    }
  }
}
```