链滴

[转]bitsharesjs 库详解一：ChainStore

作者：hiquanta

原文链接：https://ld246.com/article/1513998054851

来源网站：链滴

bitshares开发入门：开源代码总览 介绍了比特股开源代码的总体情况，其中，bitsharesjs 位于UI层下， bitsharesjs-ws 之上。本文尝试开一个系列之头：这个系列全部解析 bitsharesjs 代码。

bitsharesjs 库有三个主要模块，ECC， Chain和Serializer。ECC是关于椭圆曲线密码学的一些贴近包操作的库，Chain是关于链上数据获取和交易发起的，Serializer是Chain的工具支持，一般无需直使用。 本文阐述Chain中的一个类: ChainStore。ChainStore的功能是链上数据的获取和缓存。本文到的代码，如无特别说明，均以bitsharesjs的根目录为相对目录的起点。

环境准备

1. 安装Nodejs到本地，建议安装当前的LTS版本，本文写作时，为 6.10.3 (如果已经安装请跳过这一步)
2. 克隆代码到本地 (命令行下执行：git clone https://github.com/bitshares/bitsharesjs.git)
3. 进入 bitsharesjs目录， npm install

注意：本系列文章依赖bitsharesjs的git版本 bdda47c2250b9b9ecf92d682849c7b5b1efae90f ，确保一致，否则可能会造成理解偏差，尤其涉及代码行号。

从测试代码说起

测试代码文件： test/chain/ChainStore.js

测试方法，命令行键入

npm run test:chain

注意这个测试会测试 test/chain目录下的所有测试文件， ChainStore只是一个。如果没有本地重钱，你会发现ChainStore会测试失败。下文教你如何修改代码来做测试。

背景说明：测试使用的是 mocha BDD测试框架 ，并且(整个项目)使用了 babel转码。

第3行

|

3

|

import { FetchChain, ChainStore } from "../../lib";

|

导入了ChainStore。

第9-15行

|

9

10

11

```
12
13
14
15
```

```
before(function() {
/* use wss://bitshares.openledger.info/ws if no local node is available */
return Apis.instance("ws://127.0.0.1:8090", true).init_promise.then(function (result) {
coreAsset = result[0].network.core_asset;
ChainStore.init();
});
});
```

所有测试用例运行之前需要做初始化：先连接上全节点，测试代码使用的是本地节点，第10行注释说明白，如果没有本地节点，那么就使用公网节点，例如openleger的。国内测试，建议改成帝国的: ws://bit.btsabc.org/ws 。 另外第13行有个 bug ， 需要在前面加上 return，否则默认 return undefine，整个函数就会resolve掉，可能导致ChainStore没有初始化完成就执行测试用例，会出错的。修改的代码应该是这个样子:

```
9
10
11
12
13
14
15
```

```
before(function() {
/* use wss://bitshares.openledger.info/ws if no local node is available */
return Apis.instance("wss://bit.btsabc.org/ws", true).init_promise.then(function (result) {
coreAsset = result[0].network.core_asset;
return ChainStore.init();
});
});
```

这样就可以测试了。但是，读者会发现，测试用例不见得全部pass。这里面有另一个BUG，下文详解。

init函数

当底层Api(bitsharesjs-ws提供的Apis)初始化OK时，必须调用ChainStore的init函数初始化，正如第3行所做的那样。

首先， ChainStore这个变量，容易混淆，这个是从 lib/chain/src/ChainStore.js这个文件导入的，这个文件定义了一个ChainStore类，但本身导出的确实ChainStore类的一个全局Singleton

|

1352

|

let chain_store = new ChainStore();

|

1352行生成了ChainStore类的一个实例。

|

1407

|

export default chain_store;

|

1407行导出这个实例。

因此测试代码导入的ChainStore，是ChainStore.js文件中定义的ChainStore类的一个全局实例。这话很绕口，多读几遍。

回到init函数，该函数返回一个promise，resolve的时候初始化成功。其他函数必须在init函数返回resolve之后调用。正因为这个特点，才有了上文所述第13行的少return的BUG。

4个测试用例的所调用的两个函数

4个测试用例实际上主要调用了ChainStore(Singleton)的两个函数:

- getAsset
- subscribe

其中 getAsset是 getObject的封装，表示获取资产。而getObject是一般的“获取对象”函数，而“象”是bitshares区块链的核心数据。对象的id是3个整数， a.b.c。其中：

- a表示空间，两个取值：1表示协议对象，这些对象会在websocket和p2p网络上传输；2表示实现象，用于节点本地存储，可认为是共识数据的衍生数据。
- b表示类别，协议对象和实现对象都有十多类不同数据。

- c表示实例，不同类型数据的实例编号。

例如

- 2.1.0 表示动态全局相关数据，例如一个抓取的实例:

{ participation: 100,
recently_missed_count: 0,
accounts_registered_this_interval: 22,
next_maintenance_time: '2017-05-24T04:00:00',
dynamic_flags: 0,
witness_budget: 76200000,
head_block_id: '0100685ba0b1d1902e8ccea5e0eac2172f679873',
time: '2017-05-24T03:47:27',
recent_slots_filled: '340282366920938463463374607431768211455',
current_witness: '1.6.72',
current_aslot: 16909777,
head_block_number: 16803931,
id: '2.1.0',
last_irreversible_block_num: 16803912,
last_budget_time: '2017-05-24T03:00:00' }

- 1.3.x 表示各种类型的资产
    - 1.3.0 核心资产BTS
    - 1.3.113 锚定资产bitCNY
- 1.2.x 表示各个账号
    - 1.2.0 理事会多重签名账号
    - 1.2.121 理事会成员巨蟹的账号 bitcrab
    - 1.2.12376 理事会成员abit的账号 abit
- 1.7.x 表示用户提交的限价单
- 1.8.x 表示call order(我还真没搞清楚是什么意思，请留言)
- 1.11.x 表示用户相关的活动历史，提交限价单，取消限价单，转账给别人，收到转账等等

常用对象列表 可参看大部分的对象类型。

好，回到getObject函数，这个函数总是立即返回的，返回值有三种情况:

- 返回 Map 类型 的对象，表示缓存中有了这个对象
- 返回null，表示没有这个Object(id无效)
- 返回undefined，表示正在查询API节点，需要以后重新调用

getAsset是getObject的封装，因此返回值同样遵守这个约定。由于getObject立即返回，而调用的候如果返回undefined，怎么等呢？用 subscribe函数。 subscribe函数是通用的事件监听函数， 当ebsocket连接之后，任何从API节点的事件，都会触发所有的监听者(subscriber)。

这个设计本身是否足够好？我认为不够好，因为subscribe会导致大量的无效监听，而getObject和suscribe的联合使用，从理论上说不一定能达到预期的效果： 因为监听者无法区分事件本身，而JS的异特性会导致不确定性。从测试代码来说，4个测试用例并行执行，和webSocket的事件触发次序的不定性，会导致subscribe里面的getAsset函数不一定得到想要的结果。如果改写其中的一个测试用例设成it.only （忽略其他的测试用例)，目前我的测试结果是总可以通过的，但从理论上，我仍然不相这种单个测试用例的测试方法：万一监听到一个不相关的事件呢？从这个意义上来说，测试代码还不写得正确，现有测试代码怎么改成逻辑自洽的还很难。

另外，就ChainStore来说，测试代码的覆盖也完全不够，下面看看例子代码。

例子代码

例子代码在这里： examples/chainstore.js

运行

npm run example:chainStore

可以发现一直打印ChainStore的全局动态对象 2.1.0的当前值。

例子代码比测试代码简单，用到的函数是getObject，运行例子代码会对上文提到的无效监听设计有观的认识。

例子代码的修改

例子代码太简单了，只获得一个动态全局对象，不利于理解很多其他的概念。我通过阅读ChainSore.j的源代码，改了下，可以获得巨蟹的账号情况，注意其中的资产和操作历史：

```
import {Apis} from "bitsharesjs-ws";
import {ChainStore} from "../lib";

Apis.instance("wss://bitshares.openledger.info/ws", true).init_promise.then((res) => {
console.log("connected to:", res[0].network);
ChainStore.init().then(() => {
ChainStore.subscribe(getBitcrabAccount);
});
});

function getBitcrabAccount() {
var bitcrab = ChainStore.getAccount('bitcrab');


if( bitcrab) {
    var bitcrabObj = bitcrab.toJS();
    console.log('my account', bitcrabObj);
```

```
    var balances = bitcrabObj.balances;

    if( balances ){
        for (var assetId in balances ){
            var asset = ChainStore.getAsset(assetId);

            if( asset ){
                var assetObj = asset.toJS();
                console.log('asset:', assetId, assetObj);

                if( assetObj.dynamic_asset_data_id) {
                    var dynamicAsset = ChainStore.getObject(assetObj.dynamic_asset_data_id);

                    if( dynamicAsset ){
                        var dynamicAssetObj = dynamicAsset.toJS();
                        console.log('asset dynamic:', assetId, dynamicAssetObj);
                    }
                }
            }

            // var balance =;

            var balance = ChainStore.getObject( balances[assetId]);
            console.log('asset balance:', assetId, balance.toJS());

        }
    }

    var history = bitcrabObj.history;

    if( history ){
        history.forEach( function(h){
            console.log('history', h);
            console.log('opration', h.op);
        });

    }
}

}
```

运行修改的例子代码会不停的输出巨蟹的账号相关信息。关于操作历史，最重要的是什么操作？op的据结构是二元组，第一个数表示操作类型，第二个对象表示具体的数据。而操作类型可以在 lib/chain/rc/ChainTypes.js 里面找到，代码我就不贴了。

总结

关于ChainStore的代码解读就这些了，这个过程我总结下来:

- ChainStore的接口设计不算特别合理。怎么样才更好呢？是一个值得思考的问题。
- 业务逻辑和代码需要结合起来，比如a.b.c对象的意义，操作类型的意义。
- ChainStore测试和例子的质量不高，大体可判断，bitsharesjs总体的代码质量有待改进，如果对质要求高，可以考虑直接使用钱包和节点的 JSON RPC API。