Versione 1

Reactive-Functional Fun on the Blockchain with web3j

Creato da Yolande Poirier-Oracle il 22-feb-2017 1.10. Ultima modifica di Yolande Poirier-Oracle il 28-feb-2017 15.26.

by Conor Svensson

Reactive Java on the blockchain with web3j

In a recent issue of *Java Magazine* , I provide a primer on working with the blockchain technology Ethereum using web3j to build Java applications on it.

In this article, we're going to use web3j's reactive-functional API to shed some light on the events taking place in the public Ethereum blockchain.

The code for all examples is available at https://github.com/web3j/examples/tree/master/rx .

Background

During the past year, the technology and financial press has been full of talk about blockchain and its disruptive potential. A blockchain is a decentralized, immutable data store. Because it is immutable, data can only be appended to the blockchain. This is achieved with transactions that are grouped together into blocks, which are added to the end of the blockchain.

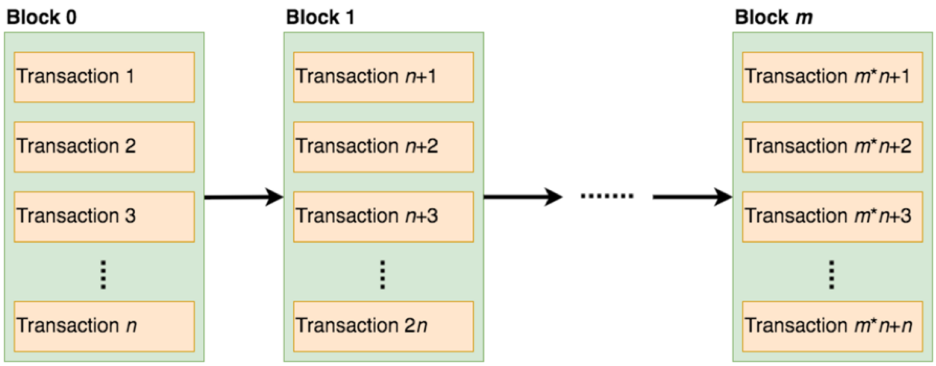

Figure 1. The structure of a blockchain

The state of the data residing in the blockchain is built by replaying the transactions or events that have gone before. You can think of it as a distributed event log.

Blockchains are completely decentralized, stored across unrelated nodes potentially in an untrusted public environment such as the internet. There are a number of different blockchain technologies in existence. Of those, Ethereum has emerged as the dominant public blockchain technology.

For more background on blockchain technology, please refer to the *Java Magazine* article).

Getting Started with Ethereum

The decentralized Ethereum network is made of clients that form the peers of the Ethereum network. To talk to the network, you need to have access to one of these clients. The easiest way to do this is to run a client yourself. The two main clients are Geth and Parity .

Once you have installed a client, you can start it as follows:

Geth:

```
$ geth --fast --cache=512 --rpcapi personal,db,eth,net,web3 --rpc --testnet
```

Parity:

```
$ parity --chain testnet
```

The client will then find other nodes to connect to and start syncing a local copy of the testnet blockchain. There are two public Ethereum blockchains—mainnet and testnet—reflecting the production and test environments, respectively.

The web3j Library

Now that you have a client running, it's easy to start talking to the Ethereum blockchain, thanks to web3j, which is a lightweight Java library for working with clients on the Ethereum blockchain. You can learn more about web3j and Ethereum at https://web3j.io .

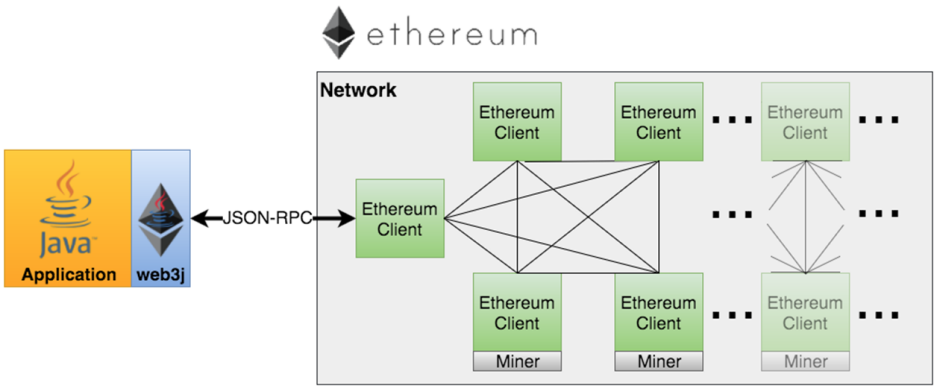

Figure 2. web3j provides an integration layer to the Ethereum blockchain for Java applications

To incorporate web3j into your project, add the following Maven dependency:

```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>2.0.0</version>
</dependency>
```

If you're working with Android, use `core-android` instead.

New Block Subscriptions

Using a couple of lines of code, you can now hook up to the Ethereum blockchain and be notified of new blocks being added to the blockchain:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
Subscription subscription = web3j.blockObservable(false).subscribe(block -> {
    System.out.println("Sweet, block number " + block.getBlock().getNumber() + " has just been created");
}, Throwable::printStackTrace);

TimeUnit.MINUTES.sleep(2);
subscription.unsubscribe();
```

This `blockObservable` will now emit a block object to the subscriber each time a new block is appended to the Ethereum blockchain.

We need to include the `sleep` statement, because the subscription takes place asynchronously in a different thread of execution to the rest of our program.

We'll fill this example out so that it provides details about the number of transactions in each block, it provides the block hash that uniquely identifies the block, and it provides the parent hash of the previous block.

We'll restrict this example to only 10 blocks, and we'll use a latch so that we wait for the 10 blocks to be emitted.

```
CountDownLatch countDownLatch = new CountDownLatch(1);

System.out.println("Waiting for " + COUNT + " transactions...");
Subscription subscription = web3j.blockObservable(true)
        .take(COUNT)
        .subscribe(ethBlock -> {
            EthBlock.Block block = ethBlock.getBlock();
            LocalDateTime timestamp = Instant.ofEpochSecond(
                    block.getTimestamp().longValueExact()).atZone(ZoneId.of("UTC")).toLocalDateTime();
```

```
            int transactionCount = block.getTransactions().size();
            String hash = block.getHash();
            String parentHash = block.getParentHash();

            System.out.println(
                    timestamp + " " +
                            "Tx count: " + transactionCount + ", " +
                            "Hash: " + hash + ", " +
                            "Parent hash: " + parentHash
            );
            countDownLatch.countDown();
        }, Throwable::printStackTrace);

subscription.unsubscribe();
```

By running the code, we can see details for the last 10 blocks added to the Ethereum blockchain:

```
2016-12-22T00:27:11 Tx count: 3, Hash: 0xbed93a59bcd30c0f11e155109a5dbff3e56a5354e90e514108917c1f54c4182d, Parent
2016-12-22T00:27:44 Tx count: 2, Hash: 0xb147a4ca79bc7c5e767ee92be648a9fd23ee15ec64f764b3f2dc8a4cb7229a50, Parent
2016-12-22T00:27:51 Tx count: 1, Hash: 0xa2d8d3572592470e30b7fb504ce504444b1ea1208035f1545c85fc824a882e1b, Parent
2016-12-22T00:28:09 Tx count: 2, Hash: 0xf4d4a7d7a2202f8c64e72b47947adb5ae419f2bdadabec9386cb9d4a27cd7ef8, Parent
2016-12-22T00:28:17 Tx count: 2, Hash: 0x43dc1da772b3f5aea3388fb4ddf14c188753646de4971b62e90b903774c5c2bc, Parent
2016-12-22T00:28:22 Tx count: 1, Hash: 0x80d03ac26a3aba27d01beecb530b9055dd28b199a52bcc53704c8e418ba437fa, Parent
2016-12-22T00:28:27 Tx count: 0, Hash: 0x897af69eafa47c2951379780a40ee47fc3383959197419a4feadc59d1d2b2c13, Parent
2016-12-22T00:28:28 Tx count: 0, Hash: 0xf12762e30caa3d0f1c6c7deb7fff93cd00727565575b8b35a7466ca75ba112e6, Parent
2016-12-22T00:28:43 Tx count: 1, Hash: 0x3754841b596ad15b17b37dd6293db77ebb672743918877dcf92e47407a2153e5, Parent
2016-12-22T00:28:49 Tx count: 2, Hash: 0x6613c62dfd05761b3794658d341c1dab95f19db1e2bebd8cf091861c1ae88cd4, Parent
```

Functional Composition

Because we're using RxJava's `Observables` , we can easily compose them to add additional functionality.

For instance, if we wish to create an `Observable`
that emits every new transaction that is written to the blockchain, we can use our `blockObservable()`
again, and extract the list of transactions contained in the block, and then use the `flatMapIterable()`
method to emit these transactions individually:

```
web3j.blockObservable(true)
         .flatMapIterable(ethBlock -> (List) ethBlock.getBlock().getTransactions());
```

This time, we pass the `true` parameter, which requests blocks and full details of all the transactions contained in those blocks.

web3j has already implemented this composition for us via `web3j.transactionObservable()` .

Counting Ether

Ethereum comes with its own cryptocurrency, named Ether, which you can think of as similar to Bitcoin. Ether is used to pay for transactions that take place on the network. These transactions can also transfer Ether from one person to another. Details about the Ether associated with a transaction are contained in the `value` field of Ethereum transactions.

Using the transaction `Observable` we created in the previous section, we can easily start pulling information out of the blockchain in real time. For instance, to obtain the total value of transactions taking place during a given number of blocks, we can use the following:

```
CountDownLatch countDownLatch = new CountDownLatch(COUNT);

System.out.println("Waiting for " + COUNT + " transactions...");
Observable<BigInteger> transactionValue = web3j.transactionObservable()
        .take(COUNT)
        .map(Transaction::getValue)
        .reduce(BigInteger.ZERO, BigInteger::add);

Subscription subscription = transactionValue.subscribe(total -> {
    System.out.println("Transaction value: " +
            Convert.fromWei(new BigDecimal(total), Convert.Unit.ETHER) + " Ether");
    countDownLatch.countDown();
}, Throwable::printStackTrace);
```

```
countDownLatch.await(10, TimeUnit.MINUTES);
subscription.unsubscribe();
```

The code above produces the following output:

```
Transaction value: 5.011 Ether (5011000000000000000 Wei)
```

Here, we obtain the value of each emitted transaction and sum them together via the `reduce()` method. Remember that you need to be dealing with a finite stream in order for the reduce function to provide a final result. Also, not all transactions will be transfers of Ether, so don't be surprised if you receive a transaction value of 0.

Ether comes in a number of different denominations, Wei being the most granular. One Wei equals 10e-18 Ether! The `Convert` method converts the value in Wei that we get back from our transactions into Ether.

Further Observables

Other observables available. For instance, to obtain details of pending transactions—those that have not yet been grouped together into a block and appended to the blockchain—we can use this:

```
Subscription subscription = web3j.pendingTransactionObservable().subscribe(tx -> {
    ...
});
```

Also, web3j provides observables for all Ethereum API calls, such as the following trivial example, which provides the client version of your Ethereum client:

```
Web3j web3 = Web3j.build(new HttpService());  // defaults to http://localhost:8545/
web3j.web3ClientVersion().observable().subscribe(x -> {
    System.out.println(x.getWeb3ClientVersion());
});
```

The code above produces the following output:

```
Client is running version:     Geth/v1.5.4-stable-b70acf3c/darwin/go1.7.3
```

For the full list of available API calls, refer to the Ethereum interface .

There are additional event types you can subscribe to in web3j, which you can read about at https://docs.web3j.io/filters.html .

Conclusion

I have provided a brief overview of the capabilities of web3j for querying the Ethereum blockchain via its reactive-functional API. If you want to learn more about web3j and Ethereum, please head over to the web3j home page at https://web3j.io , which contains further resources.

About the Author

Conor Svensson (@conors10) is the author of web3j, the Java library for integrating applications with the Ethereum blockchain. He previously cofounded the startups Cohome and Huffle before becoming the CTO at Othera and building Othera's blockchain lending platform and exchange. He is also a regular speaker and one of the organizers of the Sydney Java user group. He blogs about tech and finance stuff, and when not in front a screen, he likes to make the most of surfing at his local beach, Maroubra in Sydney.

Join the Conversation

Join the Java community conversation on Facebook , Twitter , and the Oracle Java Blog !

2720 Visualizzazioni **Categorie:** **Tag:** jvm, otn_javase_articles_, javarx

Valutazione utenti media

(0 Valutazioni)

0 Commenti