Qtum区块链开发指南 -- Hello World!

作者：cryptominder

简介

在作者之前的一篇文章（https://steemit.com/qtum/@cryptominder/qtum-blockchain-development-environment-setup）中，描述了如何利用Docker部署一个3节点的Qtum区块链环境。这篇指南将会以此为基础，希望读者在阅读本文之前先配置好相关环境。

作为Qtum智能合约开发系列教程的第一篇文章，我们将从大家熟悉的以太坊智能合约实例“Ethereum Greeter tutorial”（参考https://ethereum.org/greeter）开始。本文会该实例中相同的Solidity智能合约，并用Qtum命令行工具（即 `qtum-cli` ）进行创建和运行。

希望在阅读完本教程后，读者能对Qtum区块链有一个更加深入的认识，进而开发出更多有趣的去中心化应用（DApp）。

为何选择Qtum而不是以太坊?

Qtum结合了几个比较成熟的区块链生态系统的优势，并进行了多种创新，这些区块链包括：

Bitcoin：Qtum采用了和比特币相同的UTXO模型（当然也包含由此带来的安全性），同样支持简单支付协议（SPV）；

Ethereum：Qtum目前采用了以太坊运行智能合约的虚拟机（也即EVM）；

Blackcoin：Qtum借鉴了黑币的权益证明共识机制（PoS），并对其进行多项改进；

Qtum实现了账户抽象层（AAL），使得UTXO模型能够和以太坊的账户模型进行无缝交互。并且除了EVM外，Qtum近期还宣布会支持更多不同虚拟机。

由于支持SPV，Qtum客户端可以在带宽/内存受限的设备上运行，在这些设备上下载和保存所有区块信息是不太现实的。Qtum的SPV客户端可以在iOS（如iPhone，iPad）及Android设备上运行，甚至在卫星上也可以运行Qtum。需要说明的是，虽然Qtum支持移动设备，但移动端市场只是他们蓝图中的一部分，Qtum关注的领域非常广泛。

由于Qtum采用PoS共识机制，这使得在低功耗设备上挖矿成为可能 -- 比如树莓派。这比目前比特币和以太坊采用的PoW机制要环保得多。

同时，为了防止分叉，Qtum推出分布式自治协议（DGP）以实现区块链参数的动态调整。

基于以上的一些原因（当然还有其他原因，比如他们出色的团队组成等），作者认为Qtum区块链提供了许多

超越以太坊的特性。

准备开始

除了作者之前提到的本教程的准备外，这里补充几点：

- 确保你已经安装了Docker，并已经设置并运行好Qtum regtest环境
- 你运行以下所有命令时所在的开始路径包含对应config文件（如node1*qtumd.conf），以及对应datadir数据路径（如node1*data）
- 在需要的时候，你可以为以下命令创建batch/shell脚本

另外，在文中提到的 `$PWD` 环境变量代表Linux和MacOs/OSX的当前途径。在Windows中，应采用 `%cd%` 。

在docker使用-v选项时，命令行注意要使用全路径。

第一步 - 你准备好了么?

在开始之前，请务必检查以下环境是否都已设置妥当。首先我们要确认3个qtumd节点正确运行：

```
$ docker ps -f name=qtumd
```

以上命令会运行3个Docker容器（即qtumd*node1, qtumd*node2, qtumd_node3）。

接下来，请确保目前区块高度大于或等于600，并且有至少一个钱包的余额大于0：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/q
tum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli geti
nfo
```

以上命令正常情况下应该返回类似如下信息：

```
{
  "version": 140301,
  "protocolversion": 70016,
  "walletversion": 130000,
  "balance": 2000000.00000000,
  "stake": 0.00000000,
  "blocks": 600,
  "timeoffset": 0,
  "connections": 2,
  "proxy": "",
  "difficulty": {
    "proof-of-work": 4.656542373906925e-10,
    "proof-of-stake": 4.656542373906925e-10
  },
  "testnet": false,
  "moneysupply": 12000000,
  "keypoololdest": 1507588445,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00400000,
  "errors": ""
}
```

从以上信息中我们可以看到，当前区块高度为600，可用余额为2000000.00000000。这些余额足够我们进行本教程所有内容。

如果你通过其他节点运行getinfo命令（比如qtumd*node2和qtumd*node3），区块高度应该和上面一致，或者更高（取决于你等待的时间），但是余额会是0。例如：

```
docker run -i --network container:qtumd_node2 -v ${PWD}/node2_qtumd.conf:/home/qtu
m/qtum.conf:ro -v ${PWD}/node2_data:/data cryptominder/qtum:latest qtum-cli getinfo
```

将会返回：

```
{
  "version": 140301,
  "protocolversion": 70016,
  "walletversion": 130000,
  "balance": 0.00000000,
  "stake": 0.00000000,
  "blocks": 605,
  "timeoffset": 0,
  "connections": 2,
  "proxy": "",
  "difficulty": {
    "proof-of-work": 4.656542373906925e-10,
    "proof-of-stake": 4.656542373906925e-10
  },
  "testnet": false,
  "moneysupply": 12100000,
  "keypoololdest": 1507588475,
  "keypoolsize": 100,
  "paytxfee": 0.00000000,
  "relayfee": 0.00400000,
  "errors": ""
}
```


这是正常的，我们可以继续。

随后，我们需要验证是否有已存在的智能合约，请运行一下命令：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli list contracts
```

你应该可以看到结果列出5个智能合约：


```
{
  "0000000000000000000000000000000000000083": 0.00000000,
  "0000000000000000000000000000000000000080": 0.00000000,
  "0000000000000000000000000000000000000081": 0.00000000,
  "0000000000000000000000000000000000000082": 0.00000000,
  "0000000000000000000000000000000000000084": 0.00000000
}
```


这些合约都不是我们自己部署的，在本教程中，请忽略它们。

接下来我们都会用qtumd_node1节点运行所有qtum-cli命令，但我强烈建议你在学习过程中可以尝试在其它节点运行，看看结果如何。

第二步 -- 获取相关工具：solc和ethabi

我们将采用Solidity文档中推荐的Docker镜像作为Solidity编译器，可以通过以下方法获取镜像：

```
$ docker pull ethereum/solc
```

请确保获取后其能够正确运行：

```
$ docker run --rm -v ${PWD}:/solidity ethereum/solc:stable --version
```

以上命令的正确返回结果为：

```
solc, the solidity compiler commandline interface
Version: 0.4.17+commit.bdeb9e52.Linux.g++
```

请注意我们用到的 `-v ${PWD}:/solidity` 选项。镜像中的/sodility路径是工作路径，因此我们需要把当前的绝对路径映射到那里。

你可以通过以下命令获取solc的使用帮助：

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest --help
```

同时我们还需要ethabi命令行工具。作者创建了一个对应的镜像：https://hub.docker.com/r/cryptominder/ethabi/。

可以通过以下命令获取镜像：

```
$ docker pull cryptominder/ethabi
```

检查是否可以正确运行：

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest --help
```

以上命令返回结果如下：

```
Ethereum ABI coder.
  Copyright 2016-2017 Parity Technologies (UK) Limited

Usage:
    ethabi encode function <abi-path> <function-name> [-p <param>]... [-l | --leni
ent]
    ethabi encode params [-v <type> <param>]... [-l | --lenient]
    ethabi decode function <abi-path> <function-name> <data>
    ethabi decode params [-t <type>]... <data>
    ethabi decode log <abi-path> <event-name> [-l <topic>]... <data>
    ethabi -h | --help

Options:
    -h, --help         Display this message and exit.
    -l, --lenient      Allow short representation of input params.

Commands:
    encode             Encode ABI call.
    decode             Decode ABI call result.
    function           Load function from json ABI file.
    params             Specify types of input params inline.
    log                Decode event log.
```

或许你又注意到，我们运行docker时添加了 `-v ${PWD}:/ethabi` 选项。因为镜像中工作路径为/ethabi，因此我们需要对当前路径进行映射。

好了，现在一切准备就绪，我们可以正式开始了。

第三步 -- 编译智能合约

这里采用与https://ethereum.org/greeter一致的实例进行演示：

创建如下文件，命名为helloworld.sol:

```
contract mortal {
    /* Define variable owner of the type address */
    address owner;

    /* This function is executed at initialization and sets the owner of the contr
act */
    function mortal() { owner = msg.sender; }

    /* Function to recover the funds on the contract */
    function kill() { if (msg.sender == owner) selfdestruct(owner); }
}

contract greeter is mortal {
    /* Define variable greeting of the type string */
    string greeting;

    /* This runs when the contract is executed */
    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* Main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

通过如下命令编译Solidity代码：

```
$ docker run --rm -v ${PWD}:/solidity ethereum/solc:stable --optimize --bin --abi
--hashes -o /solidity --overwrite /solidity/helloworld.sol
```

你有可能会看到以下的一些warning：

```
/solidity/helloworld.sol:6:5: Warning: No visibility specified. Defaulting to "pub
lic".
    function mortal() { owner = msg.sender; }
    ^--------------------------------------^
/solidity/helloworld.sol:9:5: Warning: No visibility specified. Defaulting to "pub
lic".
    function kill() { if (msg.sender == owner) selfdestruct(owner); }
    ^-----------------------------------------------------------------^
/solidity/helloworld.sol:22:5: Warning: No visibility specified. Defaulting to "pu
blic".
    function greet() constant returns (string) {
    ^
Spanning multiple lines.
/solidity/helloworld.sol:1:1: Warning: Source file does not specify required compi
ler version!Consider adding "pragma solidity ^0.4.17
contract mortal {
^
Spanning multiple lines.
```

这个智能合约只是一个示例，所以请忽略这些warnings。

在运行完solc命令之后，当前路径下应该会创建一些新文件：

```
greeter.abi
greeter.bin
greeter.signatures
mortal.abi
mortal.bin
mortal.signatures
```

我们后面会用到其中这3个greeter文件。

恭喜你 -- 你刚刚成功编译了一个智能合约！

第四步 -- 部署智能合约

接下来我们将会把刚刚编译好的智能合约部署到目前的3节点Qtum regtest区块链上。

在正式开始部署之前，我们需要制定拥有足够余额的节点（在第一步中提到的qtumd_node1）的地址，作为拥有智能合约的地址。这里要用到 `getaccountaddress` RPC命令来创建或获取 `greeter_owner` 账户的地址：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/q
tum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli geta
ccountaddress greeter_owner
```

该命令会返回一个Qtum地址（例如qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5，当然你的地址会和这个不一样）。如果你再次运行同样的命令，将会返回一个相同的Qtum地址。

请务必记住这个地址**qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**，我们随后会一直用到它。（再次提醒，你的地址可能和作者的不一样，请记住自己的地址）

可有注意到的是，这个地址以小写的q开头的，这是为了与Qtum主干网络的地址进行区分。主干网络的地址是以大写的Q开头的。

你可以通过以下命令查看钱包中各个账户的余额：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli listaccounts
```

从返回的数组中你可以看到 `"greeter_owner": 0.00000000` -- 这说明greeter_owner账户目前余额为0.

好啦，现在我们可以继续部署合约了。

在上面第三步中我们生成的greeter.bin文件中包含了greeter合约的二进制码（用十六进制表示），可以在Qtum的EVM上运行。

实例中greeter合约的构造函数为：

```
function greeter(string _greeting)
```

该构造函数包含一个字符串string类型的参数，所以在部署该智能合约时需要传入一个对应参数。因此，我们需要首先知道我们所需字符串的二进制表示，用于传递参数。这个时候ethabi工具就派上用场了。运行以下命令就可以得到字符串“Hello World！”的二进制编码了：

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest encode params -v string 'Hello World!' --lenient
```

输出为：

```
0000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000000000000000000000000000000000000000c48656c6c6f20576f726c6421000000000000000000000000000000000000
```

以上就是字符串“Hello World！”的二进制表示了（输出为十六进制）。当然，你可以随意选择输入不同的字符串，但请注意，由于这些字符串是要部署到区块链上的，所以字符串越长，花费的Qtum也就越多。

现在我们需要将之前生成的greeter.bin和刚刚产生的二进制编码组合起来。在macOS/OSX或者Linux上，可以通过以下方式实现：

```
$ cat greeter.bin && docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest
encode params -v string 'Hello World!' --lenient
```

这里我就不再贴出完整的结果了，因为比较长。结果的开始段应该为：

```
60606040523415610000f57600080fd5b ...
```

结尾段就是刚才Hello World！字符串对应的编码。

部署智能合约要用到Qtum的 `createcontract` RPC命令。运行需要的参数如下：

```
createcontract "bytecode" (gaslimit gasprice "senderaddress" broadcast)
Create a contract with bytcode.

Arguments:
1. "bytecode"  (string, required) contract bytcode.
2. gasLimit  (numeric or string, optional) gasLimit, default: 2500000, max: 400000
00
3. gasPrice  (numeric or string, optional) gasPrice QTUM price per gas unit, defau
lt: 0.0000004, min:0.0000004
4. "senderaddress" (string, optional) The quantum address that will be used to cre
ate the contract.
5. "broadcast" (bool, optional, default=true) Whether to broadcast the transaction
 or not.
```

这里的 `bytecode` 就是刚刚我们组合出来的以 `60606040523415610000f57600080fd5b` 开头的二进制编码。 `gasLimit` 我们将采用默认的 `2500000` 。而 `gasPrice` 则取 `0.00000049` 。对于 `senderaddress` ,对我来说就是我刚才记下来的地址**qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**(对于读者来说，这个地址可能不同，请根据自己生成的地址进行修改)。 `broadcast` 我们暂时不设置，它会自动采用默认值。

那么整条命令如下（读者注意要用自己的地址和二进制码）：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/q
tum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli crea
tecontract 60606040523415610000f57600080fd5b604051610317380380610317833981016040528
080516000805460016a060020a0319163360016a060020a031617905591909101905060018180516
100599291602001906100605650506100fb565b828054600181600116156101000203166002900049
060005260206000209060 1f016020900481019282601f106100a157805160ff1916838001178555610
0ce565b828001600101855582156100ce579182015b828111156100ce57825182559160200191906000
101906100b3565b506100da9291506100de565b5090565b6100f891905b808211156100da576000815
56001016100e4565b90565b61020d8061010a6000396000f300606060405263ffffffff7c010000000
000000000000000000000000000000000000000000000000000600035041663 41c0e1b58114610047578
063cfae32171461005c57600080fd5b34156100525760 0080fd5b61005a6100e6565b005b341561006
757600080fd5b61006f610127565b604051602080825281908101838181518152602001915080519060
200190808383600 05b838110156100ab57808201518382015260200161009356 5b505050509050908
10190601f1680156100d857808203805160018360200361010 00a031916815260200191505b5092505
050604051809103 90f35b6000543373ffffffffffffffffffffffffffffffffffffffff90811691161
4156101255760005473ffffffffffffffffffffffffffffffffffffffff16ff5b565b61012f6101cf5
65b60018054600181600116156101000203166002900480601f016020809104026020016040519081
016040528092919081815260200182805460018160011615610100020316600290048015 6101c557806
01f1061019a576101008083540402835291602001916101c5565b820191906000526020600020905b8
154815290600101906020018083116101a857829003601f168201915b505050505090509056 5b60206
040519081016040526000815290560 0a165627a7a723058209a62630a1678b0014fdfe901ed4f21cd2
51e9b7863cfccbf79b1870bcc2e1de10029000000000000000000000000000000000000000000000000
0000000000000020000000000000000000000000000000000000000000000000000000000000c4
8656c6c6f20576f726c6421000000000000000000000000000000000000000000 2500000 0.00000049
 qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5
```

运行该命令我们应该会得到一个错误，提示说：`Sender address does not have any unspent outputs`.这是因为在我们生成的这个地址下还没有UTXO，也就是地址对应的余额不足。为了正常部署合约，我们需要从我们钱包中发3个QTUM到对应的地址：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/q
tum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli send
toaddress qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5 3
```

该命令会返回一个交易id，比如7d84a578a1d56ca71b1ea85b2028c34cce03bafe65dd782d3d8e41884eefa471。

之后我们需要等待一会儿，直到至少一个新的区块产生，然后利用`listunspent` RPC命令就可以查看对应地址（对我来说是**qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5**，再次提醒读者要修改成自己的地址）的余额了：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/q
tum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli list
unspent 1 9999999 [\"qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5\"]
```

以上命令的输出结果大致如下：

```
[
  {
    "txid": "5b9b376deef5c10e31acca1f2ff0be64179878048de753e20c2a04901fbf689b",
    "vout": 0,
    "address": "qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5",
    "account": "greeter_owner",
    "scriptPubKey": "76a914002edb387c05038b700f97ce9dc40e305805c8df88ac",
    "amount": 3.00000000,
    "confirmations": 1,
    "spendable": true,
    "solvable": true
  }
]
```

注意，上述 `listunspent` 命令的参数中的 `[\"qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5\"]` 是一个JSON数组，其中双引号 `"` 前有转义符 `\` 。

现在我们的地址中就有可用的UTXO了，再次运行上面的 `createcontract` 命令，我们将获得类似如下结果：

```
{
  "txid": "85d3c46886790cc164291500f3ed6bed20792c307b666a6fc490bd16c800c148",
  "sender": "qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5",
  "hash160": "002edb387c05038b700f97ce9dc40e305805c8df",
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606"
}
```

如果你看到这个结果，那么再次恭喜你 -- 你刚刚成功部署了一个智能合约！

第五步 -- 检查已部署的智能合约

我们可以回过头看看上面 `createcontract` 返回的结果，你可以注意到 `sender` 的地址就是我们提供的Qtum地址。而 `hash160` 的值其实是 `sender` 的hash值，你可以通过 `fromhexaddress` 和 `gethexaddress` Qtum RPC命令对两者进行相互转换。

结果中的 `address` 包含了刚刚部署在区块链上的智能合约地址。这时候如果你再次运行第一步中的 `listcontracts` 命令，你会发现多了一个结果：

```
"fd648ac3e7f89fd049507602d3d025cc90000606": 0.00000000,
```

你的合约地址也被包含其中了。

现在让我们用 `getaccountinfo` RPC命令来获取刚部署的智能合约的信息(请注意，对我来说合约地址是

上述的 `fd648ac3e7f89fd049507602d3d025cc90000606` ，对于读者来说，请使用自己刚生成的合约地址)：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli getaccountinfo fd648ac3e7f89fd049507602d3d025cc90000606
```

结果如下（不完全相同）：


```
{
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606",
  "balance": 0,
  "storage": {
    "290decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563": {
      "0000000000000000000000000000000000000000000000000000000000000000": "0000000000000000000000002edb387c05038b700f97ce9dc40e305805c8df"
    },
    "b10e2d527612073b26eecdfd717e6a320cf44b4afac2b0732d9fcbe2b7fa0cf6": {
      "0000000000000000000000000000000000000000000000000000000000000001": "48656c6c6f20576f726c6421000000000000000000000000000000000000000018"
    }
  },
  "code": "606060405263ffffffff7c010000000000000000000000000000000000000000000000000000000060003504166341c0e1b5811461004757806cfae32171461005c57600080fd5b34156100525760008fd5b61005a6100e6565b005b34156100675760008fd5b61006f610127565b604051602080825281908101838181518152602001915080519060200190808383600005b838110156100ab5780820151838201526020016100935650505050905090810190601f168015610d85780820380516001836020036101000a031916815260200191505b509250505060405180910390f35b6000543373ffffffffffffffffffffffffffffffffffffffff908116911614156101255760005473ffffffffffffffffffffffffffffffffffffffff16ff5b565b61012f6101cf565b60018054600181600116156101000203166002900480601f016020809104026020016040519081016040528092919081815260200182805460018160011615610100020316600290048015610c55780601f1061019a576101008083540402835291602001916101c5565b820191906000526020600020905b8154815290600101906020018083116101a857829003601f168201915b505050505090505090565b602060405190810160405260008152905600a165627a7a723058209a62630a1678b0014fdfe901ed4f21cd251e9b7863cfccbf79b1870bcc2e1de10029"
}
```


从输出结果可以看出该智能合约的余额为0，且其中存储了两个item。

那么存储在合约中的到底是什么呢？通过查看合约源码 `helloworld.sol` ，你可以发现其中包含2个变量：

```
owner (inherited from mortal)
greeting (declared in greeter)
```

正如你所预想的那样，第一个item（也就是索引

为 `0000000000000000000000000000000000000000000000000000000000000000` 的变量）存储的是owner的地址，仔细对比你会发现它包含了 `sender` 的 `hash160` 地址。这就说明 `sender` 就是这个智能合约的owner。

那么第二个item呢（也就是索引
为 `0000000000000000000000000000000000000000000000000000000000000001` 的变量）？这其实是 `Hello World!` 字符串的十六进制表示。我们可以通过在线工具（比如 http://www.rapidtables.com/convert/number/hex-to-ascii.htm）把它转成ASCII码，或者可以直接运行：

```
$ echo 48656c6c6f20576f726c6421000000000000000000000000000000000000000018 | xxd -r -p
```

来验证这是否确实为 `Hello World!` 。

第六步 -- 运行智能合约

这一部分我们将尝试调用部署的greeter智能合约中的greet函数。

首先我们要获取greet函数的编码，可以同样采用ethabi工具获取：

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest encode function /et
habi/greeter.abi greet
```

以上命令将会返回： `cfae3217` 。

同时，我们可以用第三步生成的greeter.signatures文件查看：

```
$ cat greeter.signatures
```

包含：

```
cfae3217: greet()
41c0e1b5: kill()
```

`ethabi` 工具可以用来为函数参数进行编码。然而，这个合约的函数都不带参数（除了构造函数），我们其实可以直接应用greeter.signatures文件。

总而言之，我们获得了 `greet` 函数的签名 `cfae3217` .

接下来我们试着从qtumd_node2运行智能合约中的 `greet` 函数。我们将采用 `callcontract` RPC命令来实现函数调用，该命令的参数为：

```
callcontract "address" "data" ( address )

Argument:
1. "address"          (string, required) The account address
2. "data"             (string, required) The data hex string
3. address              (string, optional) The sender address hex string
4. gasLimit             (string, optional) The gas limit for executing the contract
```

`address` 参数我们将传入之前我们部署的智能合约地址（对我来说也就是fd648ac3e7f89fd049507602d3d025cc90000606，读者请自行修改）。 `data` 参数应传入greet函数的二进制编码，也即 `cfae3217` 。剩下的两个参数我们可以暂时忽略。

所以我们的命令就是：

```
docker run -i --network container:qtumd_node2 -v ${PWD}/node2_qtumd.conf:/home/qtu
m/qtum.conf:ro -v ${PWD}/node2_data:/data cryptominder/qtum:latest qtum-cli callco
ntract fd648ac3e7f89fd049507602d3d025cc90000606 cfae3217
```

上述命令返回结果如下：

```
{
  "address": "fd648ac3e7f89fd049507602d3d025cc90000606",
  "executionResult": {
    "gasUsed": 22737,
    "excepted": "None",
    "newAddress": "fd648ac3e7f89fd049507602d3d025cc90000606",
    "output": "0000000000000000000000000000000000000000000000000000000000000020000
0000000000000000000000000000000000000000000000000000000000000c48656c6c6f20576f726c6
4210000000000000000000000000000000000000000",
    "codeDeposit": 0,
    "gasRefunded": 0,
    "depositSize": 0,
    "gasForDeposit": 0
  },
  "transactionReceipt": {
    "stateRoot": "a5891d6126e68e032b640c71a75da231e1f0e159cd7dfc21c1451b4d9db8e349
",
    "gasUsed": 22737,
    "bloom": "0000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000",
    "log": [
    ]
  }
}
```


我们比较关心的是 `output` 字段，所以我们用ethabi对其进行解码:

```
$ docker run --rm -v ${PWD}:/ethabi cryptominder/ethabi:latest decode params -t st
ring 0000000000000000000000000000000000000000000000000000000000000020000000000000000
0000000000000000000000000000000000000000000000000c48656c6c6f20576f726c64210000000000
0000000000000000000000000000000000000000
```

结果为:

```
string Hello World!
```

Bravo!你刚刚成功运行了一个智能合约!

第七步 -- callcontract v.s. sendtocontract

我们部署的智能合约中还包含另外一个函数 `kill` 。但在运行这个函数之前，我想要先对比这两个看似非常

相似的RPC命令 `callcontract` 和 `sendtocontract` 。

callcontract vs. sendtocontract

这两个命令的区别在其他地方也有详细描述，我这里再重复一遍：

- callcontract -- 该命令可以实现和已部署在Qtum区块链上的智能合约进行交互，所有过程在链下完成，并且不会再区块链上产生记录。**该操作不需要任何gas**。
- sendtocontract -- 该命令同样实现和已部署在Qtum区块链上的智能合约进行交互，但所有过程在链上完成，并且所有状态变化都会记录到区块链上。该操作允许向智能合约发送代币，并且**需要消耗gas**。

在之前的步骤中，我们采用的是 `callcontract` RPC命令，因此不需要再链上记录任何信息。接下来我们要运行智能合约中 `kill` 函数，它将会改变区块链上记录的状态，即移除智能合约地址。注意：智能合约地址在移除后仍然在区块链上存在（也就是说你可以通过区块链浏览器查询到），但移除后通过将不能再通过账户索引。

接下来可以开始运行 `kill`

我们将采用 `sendtocontract` RPC命令调用 `kill` 函数，其语法如下：

```
sendtocontract "contractaddress" "data" (amount gaslimit gasprice senderaddress br
oadcast)
Send funds and data to a contract.

Arguments:
1. "contractaddress" (string, required) The contract address that will receive the
 funds and data.
2. "datahex"  (string, required) data to send.
3. "amount"      (numeric or string, optional) The amount in QTUM to send. eg 0.1,
 default: 0
4. gasLimit  (numeric or string, optional) gasLimit, default: 250000, max: 4000000
0
5. gasPrice  (numeric or string, optional) gasPrice Qtum price per gas unit, defau
lt: 0.0000004, min:0.0000004
6. "senderaddress" (string, optional) The quantum address that will be used as sen
der.
7. "broadcast" (bool, optional, default=true) Whether to broadcast the transaction
 or not.
```

其中 `contractaddress` 参数我们继续采用之前的智能合约地址（对我来说也就是fd648ac3e7f89fd049507602d3d025cc90000606，读者请自行修改）。 `data` 参数则是之前我们从greeter.signatures看到的kill函数对应的十六进制码 `41c0e1b5` 。

我们用qtumd_node1运行命令：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli sendtocontract fd648ac3e7f89fd049507602d3d025cc90000606 41c0e1b5
```

运行完之后，理论上我们再调用 `getaccountinfo` RPC命令的话，我们应该找不到对应的合约了：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli getaccountinfo fd648ac3e7f89fd049507602d3d025cc90000606
```

然而……事实上它却依然存在！

怎么回事呢？如果你细心检查greeter合约中的kill函数的实现，你会发现程序中必须确保 `sender` 和 `owner` 一致。所以事实上，上面运行的函数相当于什么也没有做。

在设置 `senderaddress` 之前，记得检查这个sender地址中由至少一个UTXO，否则你又会得到余额不足的错误信息。请记住一点，我们之前的UTXO已经在 `createcontract` 的过程中花费掉了，这点很重要。所以请参照第四步中的步骤，再给你对应的地址中转入3QTUM。在以后的教程中，我会详细介绍gas返还的过程（也是通过UTXO返还的）。

现在我们已经确保了sender地址中有至少一个UTXO，随后就可以为 `sendtoaddress` RPC命令传入 `senderaddress` 参数了。让我们再次尝试调用 `kill` 函数，传入的参数分别为 `amount` 为0（因为这个函数在Solidity中没有被设置为payable）， `gasLimit` 设为250000，gasPrice设为0.00000049， `senderaddress` 设置为刚才对应的地址(对我来说即qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5)：

```
$ docker run -i --network container:qtumd_node1 -v ${PWD}/node1_qtumd.conf:/home/qtum/qtum.conf:ro -v ${PWD}/node1_data:/data cryptominder/qtum:latest qtum-cli sendtocontract fd648ac3e7f89fd049507602d3d025cc90000606 41c0e1b5 0 250000 0.00000049 qHaMHfbUC9sxqYNVgVEAyxD2sXf9bLc8f5
```

运行完之后，如果我们再次运行 `getaccountinfo` ，则会返回错误信息： `Address does not exist.` ,这说明刚才的函数调用成功了！

至此，在Qtum区块链上完成以太坊Greeter智能合约实例的演示就结束了，整个过程中我们只用到了 `qtum-cli` , `solc` 以及 `ethabi` 。

参考文献及扩展阅读

本教程中的许多内容参考了Qtum首席研发工程师Jordan Earls提供的文档。作者个人认为以下两个文档特别有用：

The Qtum Sparknet Faucet

Qtum Sparknet Usage and Information

教程写作计划

在接下来的系列教程中，作者会重点阐述以下一些主题：

- 以太坊虚拟机日志
- 智能合约数据类型（例如mapping， struct等）
- Payable的合约和函数
- 深入剖析selfdestruct
- Gas估计与返还
- 使用SPV客户端
- 创建新的代币Token
- 合约调用其他合约
- 全局变量(即msg, tx, block等)
- ... 以及其他更多主题

同时作者也会提及Qtum项目的更多进展。

如果你需要帮助，可以在原文发表评论，或是在Qtum subreddit（https://www.reddit.com/r/Qtum/）中找到作者本人（cryptominder）。同时，你也可以在Qtum的Slack中找到本文作者。

感谢

如果你认为本教程对你有帮助，并且有意愿从经济上支持作者的话，可以发送QTUM到作者的钱包地址（Qtum正式币）：QUa3yA8ALfQyM5eEb9sDPRWkX6sSurMs6D

（PS：作者的钱包运行在树莓派3上）

感谢支持！