实验一 以太坊私有链搭建

实验背景

以太坊（英文 Ethereum）是一个开源的有智能合约功能的公共区块链平台，通过其专用加密货币以太币（Ether）提供去中心化的虚拟机（“以太虚拟机” Ethereum Virtual Machine）来处理点对点合约。

以太坊的概念首次在 2013 至 2014 年间由程序员 Vitalik Buterin 受比特币启发后提出，大意为“下一代加密货币与去中心化应用平台”，在 2014 年通过 ICO 众筹开始得以发展。

截至 2018 年 2 月，以太币是市值第二高的加密货币，仅次于比特币。

以太坊是一个平台，它上面提供各种模块让用户来搭建应用，如果将搭建应用比作造房子，那么以太坊就提供了墙面、屋顶、地板等模块，用户只需像搭积木一样把房子搭起来，因此在以太坊上建立应用的成本和速度都大大改善。具体来说，以太坊通过一套图灵完备的脚本语言（EthereumVirtual Machinecode，简称 EVM 语言）来建立应用，它类似于汇编语言，我们知道，直接用汇编语言编程是非常痛苦的，但以太坊里的编程并不需要直接使用 EVM 语言，而是类似 C 语言、Python、Lisp 等高级语言，再通过编译器转成 EVM 语言。

上面所说的平台之上的应用，其实就是合约，这是以太坊的核心。合约是一个活在以太坊系统里的自动代理人，他有一个自己的以太币地址，当用户向合约的地址里发送一笔交易后，该合约就被激活，然后根据交易中的额外信息，合约会运行自身的代码，最后返回一个结果，这个结果可能是从合约的地址发出另外一笔交易。需要指出的是，以太坊中的交易，不单只是发送以太币而已，它还可以嵌入相当多的额外信息。如果一笔交易是发送给合约的，那么这些信息就非常重要，因为合约将根据这些信息来完成自身的业务逻辑。

合约所能提供的业务，几乎是无穷无尽的，它的边界就是你的想象力，因为图灵完备的语言提供了完整的自由度，让用户搭建各种应用。白皮书举了几个例子，如储蓄账户、用户自定义的子货币等。

实验目的

虽然以太坊是一个公有链系统，但是我们可以通过设置一些参数来运行自己的私有链节点，在自己的私有链上进行开发和测试不需要同步公有链数据，也不需要花钱来买以太币，节省存储空间和成本，而且很灵活很方便。本次实验将介绍使用 geth 客户端搭建私有链的操作步骤，同时会解释在这个过程中用到的各个命令及选项的含义和作用，最后会介绍 geth

的 Javascript Console 中的一些常用功能。

实验步骤

步骤一 客户端安装（1） go-ethereum 客户端安装

```
brew tap ethereum/ethereum
brew install ethereum
```

（2） 安装测试

安装完成之后在命令行输入

```
geth --help //能成功显示输出帮助，则表示已经成功安装
```

步骤二 搭建私有链

（1） 准备创始区块配置文件

以太坊支持自定义创世区块，要运行私有链，我们就需要定义自己的创世区块，创世区块信息写在一个 json 格式的配置文件中。首先将下面的内容保存到一个 json 文件中，例如 genesis.json。

```
{
  "config": {
        "chainId": 10,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
  "alloc"      : {},
  "coinbase"   : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x20000",
  "extraData"  : "",
  "gasLimit"   : "0x2fefd8",
  "nonce"      : "0x0000000000000042",
  "mixhash"    : "0x0000000000000000000000000000000000000000000000000000000000000000
  "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000
  "timestamp"  : "0x00"
}
```

ps：直接从官网复制，chainId 为 0， 但是会在合约部署出现问题，所以在这里改为 10。

（2） 初始化：写入创始区块

准备好创世区块配置文件后，需要初始化区块链，将上面的创世区块信息写入到区块链中。首先要新建一个目录用来存放区块链数据，假设新建的数据目录为 ~/privatechain/data0，genesis.json 保存在~/privatechain 中，此时目录结构应该是这样的：

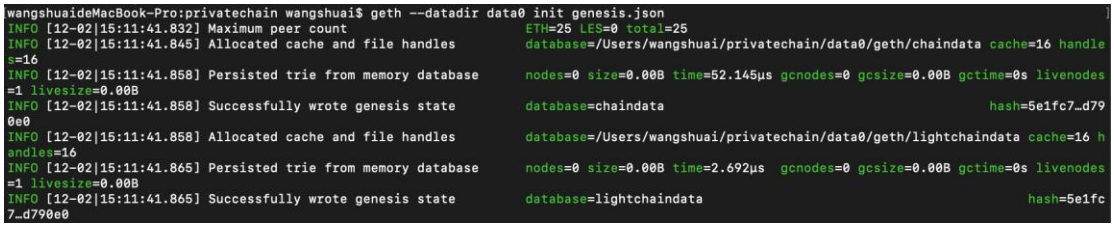
```
privatechain
├── data0
└── genesis.json
```

接下来进入 privatechain 中，执行初始化命令：

```
$ cd privatechain
$ geth --datadir data0 init genesis.json
```

上面的命令的主体是 geth init，表示初始化区块链，命令可以带有选项和参数，其中 --datadir 选项后面跟一个目录名，这里为 data0，表示指定数据存放目录为 data0，genesis.json 是 init 命令的参数。

运行上面的命令，会读取 genesis.json 文件，根据其中的内容，将创世区块写入到区块链中。如果看到以下的输出内容，说明初始化成功了。



初始化成功后，会在数据目录 data0 中生成 geth 和 keystore 两个文件夹，此时目录结构如下：

```
privatechain
├── data0
│   ├── geth
│   │   └── chaindata
│   │       ├── 000002.ldb
│   │       ├── 000003.log
│   │       ├── CURRENT
│   │       ├── LOCK
│   │       ├── LOG
│   │       └── MANIFEST-000004
│   └── keystore
└── genesis.json
```

其中 geth/chaindata 中存放的是区块数据，keystore 中存放的是账户数据。

（3） 启动私有链节点

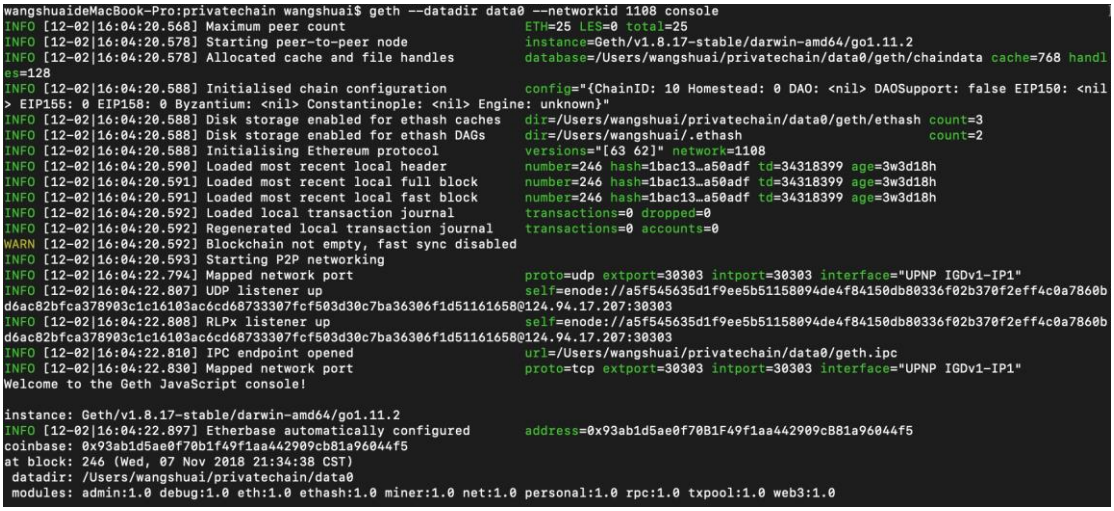
初始化完成后，就有了一条自己的私有链，之后就可以启动自己的私有链节点并做一些操作，在终端中输入以下命令即可启动节点：

```
geth --datadir data0 --networkid 1108 console
```

上面命令的主体是 geth console，表示启动节点并进入交互式控制台，--datadir 选项指定使用 data0 作为数据目录，--networkid 选项后面跟一个数字，这里是 1108，表示指定

这个私有链的网络 id 为 1108。网络 id 在连接到其他节点的时候会用到，以太坊公网的网络 id 是 1，为了不与公有链网络冲突，运行私有链节点的时候要指定自己的网络 id。

运行上面的命令后，就启动了区块链节点并进入了 Javascript Console：



这是一个交互式的 Javascript 执行环境，在这里面可以执行 Javascript 代码，其中> 是命令提示符。在这个环境里也内置了一些用来操作以太坊的 Javascript 对象，可以直接使用这些对象。这些对象主要包括：

eth：包含一些跟操作区块链相关的方法 net：包含以下查看 p2p 网络状态的方法 admin：包含一些与管理节点相关的方法 miner：包含启动&停止挖矿的一些方法 personal：主要包含一些管理账户的方法 txpool：包含一些查看交易内存池的方法

web3：包含了以上对象，还包含一些单位换算的方法 步骤三 探索 Javascript Console

进入以太坊 Javascript Console 后，就可以使用里面的内置对象做一些操作，这些内置对象提供的功能很丰富，比如查看区块和交易、创建账户、挖矿、发送交易、部署智能合约等。接下来介绍几个常用功能，下面的操作中，前面带>的表示在 Javascript Console 中执行的命令。

（1） 创建账户

前面只是搭建了私有链，并没有自己的账户，可以在 js console 中输入 eth.accounts 来验证：

```
> eth.accounts
[]
```

接下来使用 personal 对象来创建一个账户：

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x0416f04c403099184689990674f5b4259dc46bd8"
```

会提示输入密码和确认密码，输入密码不会有显示，只要输入就可以了，之后就会显示新创建的账户地址。

可以创建多个账户，我们再来创建一个账户：

```
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0xb89bf2a212484ef9f1bd09efcd57cf37dbb1e52f"
```

接下来就可以查看到刚才创建的两个账户了：

```
> eth.accounts

["0x0416f04c403099184689990674f5b4259dc46bd8",

"0xb89bf2a212484ef9f1bd09efcd57cf37dbb1e52f"]
```

账户默认会保存在数据目录的 keystore 文件夹中。查看目录结构，发现 data0/keystore 中多了两个文件，这两个文件就对应刚才创建的两个账户，这是 json 格式的文本文件，可以打开查看，里面存的是私钥经过密码加密后的信息。

```
> eth.accounts
data0
├── geth
│   ├── chaindata
│   ├── LOCK
│   ├── nodekey
│   └── nodes
├── geth.ipc
├── history
└── keystore
    ├── UTC--2018-02-27T07-57-28.597232912Z--0416f04c403099184689990674f5b425
    └── UTC--2018-02-27T07-57-56.330785628Z--b89bf2a212484ef9f1bd09efcd57cf37
```

小提示：命令都可以按 Tab 键自动补全。

（2） 查看账户余额

eth 对象提供了查看账户余额的方法：

```
> eth.getBalance(eth.accounts[0])
0
> eth.getBalance(eth.accounts[1])
0
```

目前两个账户的以太币余额都是 0，要使账户有余额，可以从其他账户转账过来，或者通过挖矿来获得以太币奖励。

（3） 启动&停止挖矿

通过 miner.start()来启动挖矿：

```
> miner.start(1)
```

其中 start 的参数表示挖矿使用的线程数。第一次启动挖矿会先生成挖矿所需的 DAG 文件，这个过程有点慢，等进度达到 100%后，就会开始挖矿，此时屏幕会被挖矿信息刷屏。如果想停止挖矿，在 js console 中输入 miner.stop()：

```
> miner.stop()
```

注意：输入的字符会被挖矿刷屏信息冲掉，没有关系，只要输入完整的 miner.stop() 之后回车，即可停止挖矿。

挖到一个区块会奖励 5 个以太币，挖矿所得的奖励会进入矿工的账户，这个账户叫做 coinbase，默认情况下 coinbase 是本地账户中的第一个账户：

```
> eth.coinbase
"0x0416f04c403099184689990674f5b4259dc46bd8"
```

现在的 coinbase 是账户 0，要想使挖矿奖励进入其他账户，通过 miner.setEtherbase() 将其他账户设置成 coinbase 即可：

```
> miner.setEtherbase(eth.accounts[1])
true
> eth.coinbase
"0xb89bf2a212484ef9f1bd09efcd57cf37dbb1e52f"
```

我们还是以账户 0 作为 coinbase，挖到区块以后，账户 0 里面应该就有余额了：

```
> eth.getBalance(eth.accounts[0])
34000000000000000000
```

getBalance()返回值的单位是 wei，wei 是以太币的最小单位，1 个以太币=10 的 18 次方个 wei。要查看有多少个以太币，可以用 web3.fromWei()将返回值换算成以太币：

```
> web3.fromWei(eth.getBalance(eth.accounts[0]),'ether')
340
```

（4） 发送交易

目前，账户一的余额还是 0：

```
> eth.getBalance(eth.accounts[1])
0
```

可以通过发送一笔交易，从账户 0 转移 5 个以太币到账户 1：

```
> amount = web3.toWei(5,'ether')
"5000000000000000000"
> eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1],value:amount})
Error: authentication needed: password or unlock
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1
```

这里报错了，原因是账户每隔一段时间就会被锁住，要发送交易，必须先解锁账户，由于我们要从账户 0 发送交易，所以要解锁账户 0：

```
> personal.unlockAccount(eth.accounts[0])
Unlock account 0x0416f04c4030991846899906 74f5b4259dc46bd8
Passphrase:
true
```

输入创建账户时设置的密码，就可以成功解锁账户。然后再发送交易：

```
> amount = web3.toWei(5,'ether')
"5000000000000000000"
> eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1],value:amount})
INFO [02-27|16:12:33] Submitted transaction                    fullhash=0x94a
"0x94a9bacda11313ddce58d1a47555aaf59ab5614bb3c8eb4b423f46464b8507f9"
```

此时交易已经提交到区块链，返回了交易的 hash，但还未被处理，这可以通过查看 txpool 来验证：

```
> txpool.status
{
  pending: 1,
  queued: 0
}
```

其中有一条 pending 的交易，pending 表示已提交但还未被处理的交易。

要使交易被处理，必须要挖矿。这里我们启动挖矿，然后等待挖到一个区块之后就停止挖矿：

```
> miner.start(1);admin.sleepBlocks(1);miner.stop();
```

当 miner.stop() 返回 true 后，txpool 中 pending 的交易数量应该为 0 了，说明交易已经被处理了：

```
> txpool.status
{
  pending: 0,
  queued: 0
}
```

此时，交易已经生效，账户一应该已经收到了 5 个以太币了：

```
> web3.fromWei(eth.getBalance(eth.accounts[1]),'ether')
5
```

(5) 查看交易和区块

eth 对象封装了查看交易和区块信息的方法。

查看当前区块总数：

```
> eth.blockNumber
69
```

通过交易 hash 查看交易：

```
> eth.getTransaction("0x94a9bacda11313ddce58d1a47555aaf59ab5614bb3c8eb4b423f4
{
  blockHash: "0x5d410b4147a06bf4e0cfc27ca84f9854f6e879cd254185ef811a81f799ed0
  blockNumber: 69,
  from: "0x0416f04c40309918468999067 4f5b4259dc46bd8",
  gas: 90000,
  gasPrice: 18000000000,
  hash: "0x94a9bacda11313ddce58d1a47555aaf59ab5614bb3c8eb4b423f46464b8507f9",
  input: "0x",
  nonce: 0,
  r: "0xbb46294248e5c31ae6d371fd5a6dedbad4d346383b5eff94066e69e927c9cb5e",
  s: "0x4ece28bd523c97ac2a7089693a217bc0092a482c27d50a435dcd2421ec66b5e7",
  to: "0xb89bf2a212484ef9f1bd09efcd57cf37dbb1e52f",
  transactionIndex: 0,
  v: "0x37",
  value: 5000000000000000000
}
```

通过区块号查看区块：

```
> eth.getBlock(33)
{
  difficulty: 133056,
  extraData: "0xd883010703846765746887676f312e392e328664617277696e",
  gasLimit: 3244382,
  gasUsed: 0,
  hash: "0x198ec33f48858979195c6bfab631cd516a10ff5473f26598398c9d445a0e2d01",
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000
  miner: "0x0416f04c403099184689990674f5b4259dc46bd8",
  mixHash: "0xe43e60cbbb0063e712a4c3900808deff5ef582b690c17ecadbbb32dd44bc795
  nonce: "0x3dabcace6101360d",
  number: 33,
  parentHash: "0x922551d1ea1f63845b2662370f1334eb9b7554605985a93121cd32d12f59
  receiptsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e3
  sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d4
  size: 536,
  stateRoot: "0x24bd5ceedf75a25e8e065cf9553e097e405ef4d6cf38ddf64f621244aa229
  timestamp: 1519718647,
  totalDifficulty: 4488192,
  transactions: [],
  transactionsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622f
  uncles: []
}
```