Blockchain Final Project Part 1: Warming Report
1. 以太坊的安装，私有链创世区块搭建，私有链节点的加入
    1.1 以太坊的安装
    1.2 私有链创世区块搭建
        1.2.1 配置创世区块
        1.2.2 启动区块链
    1.3 加入私有链节点
2. 区块字段解释
3. 日志输出解释
4.编写智能合约，并部署在链上进行调用
    4.1 进行交易
    4.2 创建和编译智能合约
    4.3 部署智能合约
    4.4 调用智能合约
5. 对交易的字段进行解释

Blockchain Final Project Part 1: Warming Report

1. 以太坊的安装，私有链创世区块搭建，私有链节点的加入

1.1 以太坊的安装

安装geth客户端：Windows10环境，官网下载.exe文件，安装；

命令行输入`geth version`检查安装；

安装Go环境，自行谷歌；

1.2 私有链创世区块搭建

创世区块是一个区块链的第一个区块。同时，创世区块也会初始化节点的行为。

1.2.1 配置创世区块

在目标文件夹中创建`genesis.json`文件：

```
{
    "config": {
        "chainId": 22,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip155Block": 0
    },
    "alloc": {},
    "coinbase":
```

```
        "0x0000000000000000000000000000000000000000",
    "difficulty": "0x400",
    "extraData": "",
    "gasLimit": "0x2fefd8",
    "nonce": "0x0000000000000038",
    "mixhash":
        "0x0000000000000000000000000000000000000000000000000000000000000000",
    "parentHash" :
        "0x0000000000000000000000000000000000000000000000000000000000000000",
    "timestamp" : "0x00"
}
```

`chainId` 指定了独立的区块链网络 ID，不同 ID 网络的节点无法互相连;

`homesteadBlock` 以太坊的早期版本需要的配置值，可留为 0;

`eip155Block/eip158Block` EIP 是指 “Ethereum Improvement Proposals”，他们被设计来代替 `homesteadBlock` 处理硬分叉等事宜，在私有链上我们无需设置，同样设置为 0 即可。

`alloc` 用来预置账号以及账号的以太币数量，因为私有链挖矿比较容易，所以我们不需要预置有币的账号，需要的时候自己创建即可以;

`coinbase` 矿工的账号，随便填;

`extraData` 附加信息，随便填，可以填你的个性信息;

`difficulty` 设置当前区块的难度，如果难度过大，cpu挖矿就很难，这里设置较小难度，其的含义是“平均每 `difficulty` 次 hash 值的尝试就会生成一个有效的区块”;

`gasLimit` 该值设置对GAS的消耗总量限制，用来限制区块能包含的交易信息总和，因为我们是私有链，所以填最大;

`mixhash` 与nonce配合用于挖矿，由上一个区块的一部分生成的hash。注意他和nonce的设置需要满足以太坊的Yellow paper, 4.3.4. Block Header Validity, (44)章节所描述的条件;

`nonce` nonce就是一个64位随机数，用于挖矿，注意他和mixhash的设置需要满足以太坊的Yellow paper, 4.3.4. Block Header Validity, (44)章节所描述的条件;

`timestamp` 设置创世块的时间戳;

`parentHash` 上一个区块的hash值，因为是创世块，所以这个值是0;

1.2.2 启动区块链

用以下命令初始化区块链，生成创世区块和初始状态:

```
geth --datadir /path/to/datadir init /path/to/genesis.json
```

`--datadir` 指定区块链数据的存储位置，可自行选择一个目录地址。

```
Chen Yanan@ChenYanan MINGW64 /d/ethereum
$ geth --datadir private init genesis.json
INFO [11-04|03:10:48.768] Maximum peer count                       ETH=25 LES=0 total=25
INFO [11-04|03:10:48.809] Allocated cache and file handles         database=D:\\ethereum\\private\\
geth\\chaindata cache=16 handles=16
INFO [11-04|03:10:48.958] Writing custom genesis block
INFO [11-04|03:10:48.958] Persisted trie from memory database      nodes=0 size=0.00B time=0s gcnod
es=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [11-04|03:10:48.973] Successfully wrote genesis state         database=chaindata
                hash=84e71d…97246e
INFO [11-04|03:10:48.973] Allocated cache and file handles         database=D:\\ethereum\\private\\
geth\\lightchaindata cache=16 handles=16
INFO [11-04|03:10:49.079] Writing custom genesis block
INFO [11-04|03:10:49.079] Persisted trie from memory database      nodes=0 size=0.00B time=0s gcnod
es=0 gcsize=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [11-04|03:10:49.080] Successfully wrote genesis state         database=lightchaindata
                  hash=84e71d…97246e
```

Geth 将在 `private` 目录下初始化区块链，并且以 `genesis.json` 作为创世区块的配置文件。之后该区块链产生的所有账号信息以及数据库信息都会被存储在该文件夹中，其中geth/chaindata中存放的是区块数据，keystore中存放的是账户数据。

对于一个区块链，需要多个节点。如果要使区块链成为peers，它们需要拥有相同的创始文件。所以要从同一个目录运行和上面相同的命令，但是要使用不同的`datadir`。

接下来用以下命令启动节点，并进入 Geth 命令行界面：

```
$ geth --identity "ChenynTestNode" --networkid 93318 --rpc --rpcport "8545" --rpcapi
"personal,eth,net,web3,miner" --rpcaddr 127.0.0.1 --datadir private --port "30303" --
nodiscover console 2>>NewTest.log
```

各选项的含义如下：

- `--identity`：指定节点 ID，区块链的标示，随便填写，用于标示目前网络的名字；
- `networkid`：设置当前区块链的网络ID，用于区分不同的网络，是一个数字，你的以太坊网络的唯一标识，你可以设置为任意值，但是不能和以太坊的保留值冲突；
- `--rpc`： 表示开启 HTTP-RPC 服务；
- `--rpcport`： 指定 HTTP-RPC 服务监听端口号（默认为 8545）；
- `--rpccorsdomain`；
- `rpcapi` 设置允许连接的rpc的客户端，一般为,eth,net,web3,personal,miner;
- `--datadir`： 指定区块链数据的存储位置；
- `--port`： 指定和其他节点连接所用的端口号，以太坊服务所运行在的网络端口号（默认为 30303）；
- `--nodiscover`： 关闭节点发现机制，防止加入有同样初始配置的陌生节点；
- `maxpeers` ： 最大网络用户数，设为 0 时代表禁用网络。

```
Chen Yanan@ChenYanan MINGW64 /d/ethereum
$ geth --identity "ChenynTestNode" --rpc --rpcport "8545" --datadir private --port "30303" --nodisc
over console
INFO [11-04|03:29:24.572] Maximum peer count                       ETH=25 LES=0 total=25
INFO [11-04|03:29:24.625] Starting peer-to-peer node               instance=Geth/ChenynTestNode/v1.
8.17-stable-8bbe7207/windows-amd64/go1.11.1
INFO [11-04|03:29:24.625] Allocated cache and file handles         database=D:\\ethereum\\private\\
geth\\chaindata cache=768 handles=1024
INFO [11-04|03:29:24.891] Initialised chain configuration          config="{ChainID: 22 Homestead:
0 DAO: <nil> DAOSupport: false EIP150: <nil> EIP155: 0 EIP158: <nil> Byzantium: <nil> Constantinopl
e: <nil> Engine: unknown}"
INFO [11-04|03:29:24.892] Disk storage enabled for ethash caches   dir=D:\\ethereum\\private\\geth\
\ethash count=3
INFO [11-04|03:29:24.892] Disk storage enabled for ethash DAGs     dir="C:\\Users\\Chen Yanan\\AppD
ata\\Ethash" count=2
INFO [11-04|03:29:24.892] Initialising Ethereum protocol           versions="[63 62]" network=1
INFO [11-04|03:29:24.894] Loaded most recent local header          number=0 hash=84e71d…97246e td=1
024 age=49y6mo2w
INFO [11-04|03:29:24.894] Loaded most recent local full block      number=0 hash=84e71d…97246e td=1
024 age=49y6mo2w
INFO [11-04|03:29:24.894] Loaded most recent local fast block      number=0 hash=84e71d…97246e td=1
024 age=49y6mo2w
INFO [11-04|03:29:24.898] Regenerated local transaction journal    transactions=0 accounts=0
INFO [11-04|03:29:24.899] Starting P2P networking
INFO [11-04|03:29:24.974] IPC endpoint opened                      url=\\\\.\\pipe\\geth.ipc
INFO [11-04|03:29:24.976] HTTP endpoint opened                     url=http://127.0.0.1:8545 cors=
vhosts=localhost
Welcome to the Geth JavaScript console!

instance: Geth/ChenynTestNode/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0
web3:1.0

> INFO [11-04|03:29:25.135] RLPx listener up                         self="enode://d39c656230372f49
a5196dcb1dc920dba61c938654f558225be644e6fb8bf7dce87096f75e73662dde9bd7d3f6c4c2d0a040f82f9747bdd2968
12189b02dca3e@172.18.152.26:30303?discport=0"
INFO [11-04|03:29:25.287] Mapped network port                      proto=tcp extport=30303 intport=
30303 interface=NAT-PMP(192.168.199.1)
```

执行成功后，会进入javascript 控制台，以太坊私有网络搭建成功。

因为当前的终端窗口正在执行服务端的进程，所以我们新建一个新的终端窗口，并在其中运行命令：

```
geth attach http://127.0.0.1:8545
```

通过这个命令，我们建立了一个 Geth JavaScript 客户端，并且连接到了位于 8545 端口的以太坊 RPC-HTTP 服务器，并且启动一个 JavaScript 终端来接收命令：

```
$ geth attach http://127.0.0.1:8545
Welcome to the Geth JavaScript console!

instance: Geth/ChenynTestNode/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1
 modules: eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 web3:1.0
```

当我们之前成功挖矿之后，启动该链后 JavaScript 终端如下：

```
$ geth --identity "ChenynTestNode" --networkid 93318 --rpc --rpcport "8545" --rpcapi "personal,eth,
net,web3,miner" --rpcaddr 127.0.0.1 --datadir private --port "30303" --nodiscover --dev.period 1 co
nsole 2>>Test.log
Welcome to the Geth JavaScript console!

instance: Geth/ChenynTestNode/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1
coinbase: 0x7aa2c6bf78992d6d3641ac9de4ab5c8abcf2b9ad
at block: 0 (Thu, 01 Jan 1970 08:00:00 CST)
 datadir: D:\ethereum\private
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0
web3:1.0
```

`coinbase` 就是之前我们挖矿所得的以太币进入的账户，默认情况下它是本地账户中第一个账户。

新的命令行界面采用 JavaScript 语法，我们使用下面的命令查看账号信息：

```
eth.accounts
```

如果之前没有创建过账号，输出为空：

```
> eth.accounts

[]
>
```

若之前创建过账号，则账号是以数组的形式显示出来：

```
> eth.accounts

["0x7aa2c6bf78992d6d3641ac9de4ab5c8abcf2b9ad", "0xbb2a035a8b15af64403d24b799d4c51c7ba413e7"]
```

使用以下命令新建一个账号：

```
personal.newAccount('password')
```

显示生成的账号：

```
> personal.newAccount('password')

"0x0491bf7369c18fea647d57a81e51bae2c4958e24"
> eth.accounts

["0x0491bf7369c18fea647d57a81e51bae2c4958e24"]
```

可以用以下命令查看该账号余额：

```
> myAddress = "0x0491bf7369c18fea647d57a81e51bae2c4958e24"

"0x0491bf7369c18fea647d57a81e51bae2c4958e24"
> eth.getBalance(myAddress)

0
```

看到该账号当前余额为 0；

可用 `miner.start()` 命令进行挖矿，由于初始难度设置的较小，所以很容易就可挖出一些余额，`miner.stop()` 命令可以停止挖矿。

这里我在 `miner.start()` 后返回 `null`，但事实上该命令已经执行成功，只不过节点误报，返回null。查看日志或执行 `eth.blockNumber` 发现区块都在不停的增高。解决方案

这时我们查看我们的账号余额：

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

120000000000000000000
> eth.getBalance(eth.accounts[1])

0
```

1.3 加入私有链节点

在新的文件夹中添加区块初始化文件genesis.json，新建两个文件夹：`node1` 和 `node2`，以相同的网络 id 和不同的端口号创建区块链节点：

```
# Node1
geth --datadir ./node1/ init genesis.json
geth --datadir node1 --nodiscover --networkid 70141 --port 16341 console
# Node2
geth --datadir ./node2/ init genesis.json
geth --datadir node2 --nodiscover --networkid 70141 --port 16342 console
```

在第一个节点的控制台中运行命令获得节点信息：

```
> admin.nodeInfo

{
  enode: "enode://ee2bf704110e215e0f25797f45851d9105e0669846a3a11fb4462e74d3aa26b555bc6e6bca11df487
57dce8a1ffc4b7ee2468c4bbf5645234535280cb4a04362@127.0.0.1:16341?discport=0",
  id: "7f5a8a14622e724bce97364e6b5ba536e72bc503cd89d2d25edcab9ff0bbf829",
  ip: "127.0.0.1",
  listenAddr: "[::]:16341",
  name: "Geth/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1",
  ports: {
    discovery: 0,
    listener: 16341
  },
  protocols: {
    eth: {
      config: {
        chainId: 41,
        eip150Hash: "0x0000000000000000000000000000000000000000000000000000000000000000",
        eip155Block: 0,
        homesteadBlock: 0
      },
      difficulty: 1024,
      genesis: "0x0d0acee0b46d4809dbd25c2bf24ac7cd56839d94da9f07286f56c69d83360cb4",
      head: "0x0d0acee0b46d4809dbd25c2bf24ac7cd56839d94da9f07286f56c69d83360cb4",
      network: 70141
    }
  }
}
```

在第二个节点的终端中运行如下命令，参数是第一个节点的 `enode`，`enode` 字段是一个可以标识该节点的协议链接：

```
> admin.addPeer("enode://ee2bf704110e215e0f25797f45851d9105e0669846a3a11fb4462e74d3aa26b555bc6e6bca
11df48757dce8a1ffc4b7ee2468c4bbf5645234535280cb4a04362@127.0.0.1:16341?discport=0")

true
```

回到第一个节点，运行如下命令，可以看到两个节点已经建立了连接：

```
> admin.peers

[{
    caps: ["eth/63"],
    enode: "enode://e08ddbb6158f1f9ca39926c2798391df9559f77549ed927aaa349dfa563d86460231a72c7058a12
06c61a5fa6cb447b40a622bb44e1acd7d09323de87e5b4814@127.0.0.1:7441",
    id: "156aef4b99a2fb76b53ece4775407f977ea83859e2b41509b91ca88db5b3dd47",
    name: "Geth/v1.8.17-stable-8bbe7207/windows-amd64/go1.11.1",
    network: {
      inbound: true,
      localAddress: "127.0.0.1:16341",
      remoteAddress: "127.0.0.1:7441",
      static: false,
      trusted: false
    },
    protocols: {
      eth: {
        difficulty: 1024,
        head: "0x0d0acee0b46d4809dbd25c2bf24ac7cd56839d94da9f07286f56c69d83360cb4",
        version: 63
      }
    }
}]
```

建立连接后，我们可以在两个节点之间进行交易；

分别在两个节点上创建账号，然后在第一个节点上开始挖矿获得交易所需的以太币，然后将其中的10转账给另一个节点：

```
> personal.unlockAccount(eth.accounts[0], 'node1')

true
> eth.sendTransaction({from: eth.accounts[0], to: "0x5312a4eeefec10d74309d3f72d9517488423e2c2", val
ue: web3.toWei(10,"ether")})
```

发送交易时 `json` 的 `to` 字段是第二个节点的账号哈希；我们在第二个节点上查看转账的结果：

```
> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether")
10
```

私有链新节点的加入与交易成功。

2. 区块字段解释

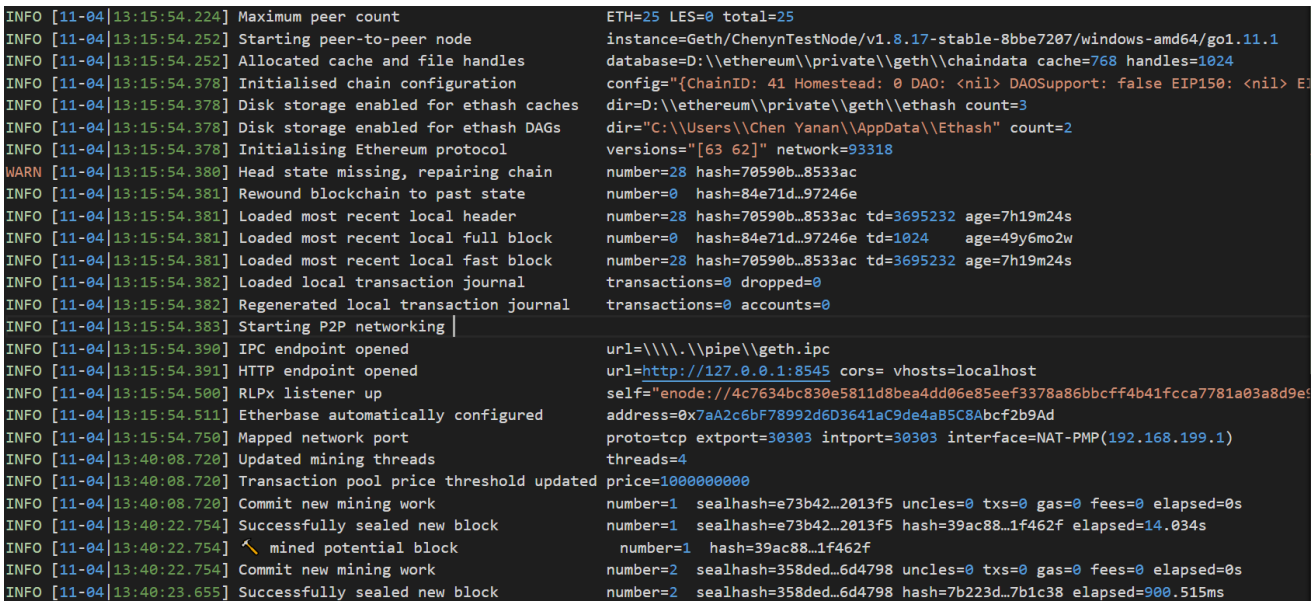
通过命令 `eth.getBlock(1)` 得到一个区块，以此为例进行字段解释，其中1为BlockNumber：

```
{
  difficulty: 131072,
  extraData: "0xda83010811846765746888676f312e31312e318777696e646f7773",
  gasLimit: 3144658,
  gasUsed: 0,
  hash: "0xa9e1bdc62e61a697019e1c2aa759fe7b9f64a73fe996ec8ca8d901d1fc404ba9",
  logsBloom:
"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000",
  miner: "0x0491bf7369c18fea647d57a81e51bae2c4958e24",
  mixHash: "0xaddcfbc456555ee76288432882f8b1ebcf0eaeb04239812ec24f0e417c13281f",
```

```
  nonce: "0x3f3632ca30e9ce15",
  number: 1,
  parentHash: "0x84e71d99bb574a31cceb83b131b9541491101aa7fd7aa2a28d8931d01897246e",
  receiptsRoot: "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
  sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  size: 538,
  stateRoot: "0x90efc2c6bec6d3332f9217e0987cdddc065186907285da57807841a5fe8dd221",
  timestamp: 1541278134,
  totalDifficulty: 132096,
  transactions: [],
  transactionsRoot:
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
  uncles: []
}
```


- `difficulty`：整数，当前区块的设定难度；
- `extraData`：字符串，当前块的额外数据；
- `gasLimit`：区块最多允许使用的 gas 量；
- `gasUsed`：区块中交易使用掉的 gas 量；
- `hash`：字符串，区块的哈希值，当这个区块处于pending将会返回null；
- `logsBloom`：字符串，区块日志的布隆过滤器，当这个区块处于pending将会返回null；
- `miner`：字符串，20字节，挖出该区块的矿工账号；
- `mixHash`：由 nonce 值得出，参与验证 PoW；
- `nonce`：字符串，8字节，用于验证 PoW 的随机值，POW生成的哈希，当这个区块处于pending将会返回null；
- `number`：区块编号，当这个区块处于pending将会返回null；
- `parentHash`：32字节字符串，上一个区块的哈希值；
- `receiptsRoot`： 当前区块的回执的哈希根值；
- `sha3Uncles`：字符串，32字节，叔区块的哈希值，区块的 uncle 数据的 SHA3；
- `size`：区块大小；
- `stateRoot`：字符串，32字节，区块的最终状态的哈希根值；
- `timestamp`：区块打包时的unix时间戳；
- `totalDifficulty`：区块链到当前块的总难度，整数；
- `transactions`：数组。交易对象，或者是32字节的交易哈希；
- `transactionsRoot`：字符串，32字节，所有交易的哈希根值；
- `uncles`：uncle 哈希的数组；

3. 日志输出解释



`Maximum peer count`：最大化对等数量；

`Starting peer-to-peer node`：开始P2P网络节点，实例采用的是基于go1.11.1版本的geth实例ChenynTestNode;

`Allocated cache and file handles`：分配缓存和文件句柄（打开文件的唯一标识，给一个文件、设备、socket或管道一个名字，隐藏关联细节），数据库位置在D:\ethereum\private\geth\chaindata，缓存大小为768M, 文件句柄数为1024；

`Initialised chain configuration`：初始化链配置，展示配置信息；

`Disk storage enabled for ethash caches`：ethash缓存启用磁盘存储器；

`Disk storage enabled for ethash DAGs`：ethash DAGs启用磁盘存储器；

`Initialising Ethereum protocol`：初始化以太坊协议；

`Head state missing, repairing chain`：number为28的区块头部信息丢失，进行区块修复；

`Rewound blockchain to past state`；

`Loaded most recent local header`：载入大部分最近的本地数据头；

`Loaded most recent local full block`：载入大部分最近的本地完整块数据；

`Loaded most recent local fast block`：载入大部分最近的本地最高块数据

`Loaded local transaction journal`：载入本地交易账本；

`Regenerated local transaction journal`：重新生成本地交易账本；

`Starting P2P networking`：开始p2p网络；

`IPC endpoint opened`：IPC端点开启：url=\\.\pipe\geth.ipc；

`HTTP endpoint opened`：HTTP端点开启，这里我们设置的为localhost；

`RLPx listener up`：RLPx开始监控，并打印出当前enode信息；

`Etherbase automatically configured`：Etherbase自动配置，地址为当前默认账户；

`Mapped network port`：映射网络端口参数；

`Updated mining threads`：更新挖矿进程；

`Transaction pool price threshold updated`：交易池价格阀值更新，价格为1000000000；

`Commit new mining work`：提交新的挖矿工作；

`Successfully sealed new block`：成功密封了一个新的区块；

`🔨 mined potential block`：挖到区块；

4.编写智能合约，并部署在链上进行调用

4.1 进行交易

下面我们进行一笔交易：从账户0转移10个以太币到账户1，由于账户每隔一段时间就会被锁住，所以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 0x7aa2c6bf78992d6d3641ac9de4ab5c8abcf2b9ad
!! Unsupported terminal, password will be echoed.
Passphrase: password

true
```

然后发送交易：

```
> eth.sendTransaction({from: eth.accounts[0], to: eth.accounts[1], value: web3.toWei(10,"ether")})

"0x16aca5b595a88e9acb40302590a180afa68dbce9fb31133459dc5c8e92e5e67f"
```

发送交易后，交易需要由矿工来确认，我们输入语句`web3.fromWei(eth.getBalance(eth.accounts[1]), "ether")`发现返回0，查询两个账户的余额发现并未变化：

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

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

150000000000000000000
> eth.getBalance(eth.accounts[1])

(anonymous): Line 1:32 Unexpected token ILLEGAL (and 15 more errors)
> eth.getBalance(eth.accounts[1])

0
```

此时交易已经提交到私有链，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()返回后，txpool中pending的交易数量为0，说明交易已经被处理了，而账户1也收到以太币了：

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

null
> txpool.status

{
  pending: 0,
  queued: 0
}
> web3.fromWei(eth.getBalance(eth.accounts[1]),"ether")

10
```

4.2 创建和编译智能合约

这里我们使用Solidity语言编写智能合约，通过Remix在线编译器将合约代码编译为EVM二进制，并将其部署在区块链上的交易中以执行和调用。

我们编写一个helloWorld智能合约：

```
pragma solidity ^0.4.0;
contract helloWorld {
    string str;

    constructor(string _str) public {
        str = _str;
    }

    function getStr() constant public returns (string){
        return str;
    }
}
```

在Remix中新建一个sol文件，粘贴进去上面写好的helloWorld代码，然后点击右侧Details，弹出的界面包含了名字、字节码、元数据等内容，我们只需要其中的WEB3DEPLOY，复制出其中内容，在第一行传入参数“hello world”：

```
var _str = "hello world";
var helloworldContract = web3.eth.contract([{"constant":true,"inputs":
[],"name":"getStr","outputs":
[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function
"},{"inputs":
[{"name":"_str","type":"string"}],"payable":false,"stateMutability":"nonpayable","type"
:"constructor"}]);
var helloworld = helloworldContract.new(
   _str,
   {
     from: web3.eth.accounts[0],
```

```
      data:
'0x6080604052348015610010576000800fd5b506040516102a83803806102a8833981018060405281019080
805182019291905050508060009080519060200190610049929190610050565b50506100f5565b828054600
181600116156101000203166002900490600052602060002090601f016020900481019282601f1061009157
805160ff1916838001178555610bf565b8280016001018555821561bf579182015b828111156100be578
251825591602001919060010190610a3565b5b5090506100cc91906100d0565b5090565b6100f291905b80
8211156100ee576000816000905550600101610d6565b5090565b90565b6101a480610104600039600f30
06080604052600436106100415760003557c010000000000000000000000000000000000000000000000000
0000900463ffffffff168063b8c9e4ed14610046575b600080fd5b348015610052576000800fd5b5061005
b6100d6565b604051808060200182810382528381815181526020019150805190602001908083836000 5b83
81101561009b578082015181840152602081019050610080565b50505050905090810190601f1680156100c
857808203805160018360200361010000a031916815260200191505b509250505060405180910390f35b6060
600080546001816001161561010002031660029004806 01f0160208091040260200160405190810160405 28
092919081815260200182805460018160011615610100020316600290048015610 16e5780601f1061014357
610100808354040283529160200191610 16e565b820191906000526020600020905b815481529060010190 6
020018083116101515782900360 1f168201915b505050505090505090560 0a165627a7a7230582070e5cc7ba5
6bb00b72a60f93708ac5f052eb34c6e89a136bd15687d9e602e42d0029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
    }
 })
```

这里，Remix帮我们将代码转成了EVM可识别的样子，也就是将Solidity代码编译成web3的版本，其中也帮我们估算好了gas的金额，当我们执行这段合约时会自动扣掉我们余额中相应的数值作为gas费用。

4.3 部署智能合约

在 Geth 的 JavaScript 环境命令行界面，首先用以下命令解锁自己的账户，否则无法发送交易：

```
> personal.unlockAccount(eth.accounts[0],'password')

true
```

接下来发送部署合约的交易，将上面的web3版的代码复制过来，回车，显示如下：

```
null [object Object]
undefined
```

> 这里出现了一个问题，就是出现 `Error: exceeds block gas limit undefined` 的报错信息，表示当前合约所需的gas超过了区块的最大gas。解决方案是将创世区块的配置文件 `genesis.json` 中的参数 `gasLimit` 设置为 `0xffffffff` （十进制值为 `4294967295` ）。重新初始化，生成区块。

如果此时没有在挖矿，用 `txpool.status` 命令可看到本地交易池中有一个待确认的交易。可用以下命令查看当前待确认的交易：

```
> txpool.status

{
  pending: 1,
  queued: 0
}
> eth.getBlock("pending",true).transactions

[{
    blockHash: "0x8acb40c2ea9c27ec6cc85b6e36f172272452938bf133be114459704c1bcae5f9",
    blockNumber: 8,
    from: "0x6b54caf240653cdde660d906606a67e88f9e1f62",
    gas: 4700000,
    gasPrice: 1000000000,
    hash: "0x3ad7358a51044651badddb0b8bd9736ec35dee8e0a4d5e68bee7bad437be3553",
    input: "0x608060405234801561001057600080fd5b506040516102a83803806102a883398101806040528101908080518201929190505050806000908051906020019061004992919061005056565b50506100f5565b828054600181600116156101000203166002900490600052602060002090601f016020900481019282601f1061009157805160ff191683800117855556100bf565b828001600101855582156100bf579182015b828111156100be578251825591602001919060010190610a3565b5b5090506100cc91906100d0565b5090565b6100f291905b808211156100ee576000816000905550600101610d6565b5090565b90565b6101a4806101046000396000f300608060405260043610610041576000357c01000000000000000000000000000000000000000000000000000000900463ffffffff168063b8c9e4ed14610046575b600080fd5b348015610052576000080fd5b5061005b6100d6565b6040518080602001828103825283818151815260200191508051906020019080838360005b8381101561009b578082015181840152602081019050610080565b50505050905090810190601f1680156100c85780820380516001836020036101000a031916815260200191505b509250505060405180910390f35b60606000805460018160011615610100020316600290048060 1f0160208091040260200160405190810160405280929190818152602001828054600181600116156101000203166002900480156101 6e5780601f1061014357610100808354040283529160200191610 16e565b820191906000526020600020905b815481529060010190602001808311610151578290036 01f168201915b5050505050905090560 0a165627a7a72305820 70e5cc7ba56bb00b72a60f93708ac5f052eb34c6e89a136bd15687d9e602e42d0029000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000 0a68656c6c6f776f726c640000000000000000000000000000000000000000000000",
    nonce: 0,
    r: "0x63869ca1687bdfde91630fad38baa2fb9a5e3292617d78529634770dc2ddcb9d",
    s: "0x26c5e369807e942a49a30d978616c2b26315b4db01c37c403beb7106142f5231",
    to: null,
    transactionIndex: 0,
    v: "0x75",
    value: 0
}]
>
```

用 `miner.start()` 命令挖矿，一段时间后，交易会被确认，即随新区块进入区块链这里打印出来的是合约地址，交易hash（这在以太坊中也被认定为是一笔交易，我们付费gas给以太坊）：

```
Contract mined! address: 0x147bc07dae221df2a61ce10de825c546508ea943 transactionHash: 0x3ad7358a51044651badddb0b8bd9736ec35dee8e0a4d5e68bee7bad437be3553
```

4.4 调用智能合约

用以下命令可以发送交易，其中 `sendTransaction` 方法的前几个参数与合约中方法的输入参数对应。这种方式，交易会通过挖矿记录到区块链中，如果涉及状态改变也会获得全网共识：

```
> helloworld.getStr.sendTransaction({from:eth.accounts[0]})

"0x7ccc8b399202e78ea84071e1a611f2aff1ce9bf241f8bf13e45bc20db55dc9fe"
```

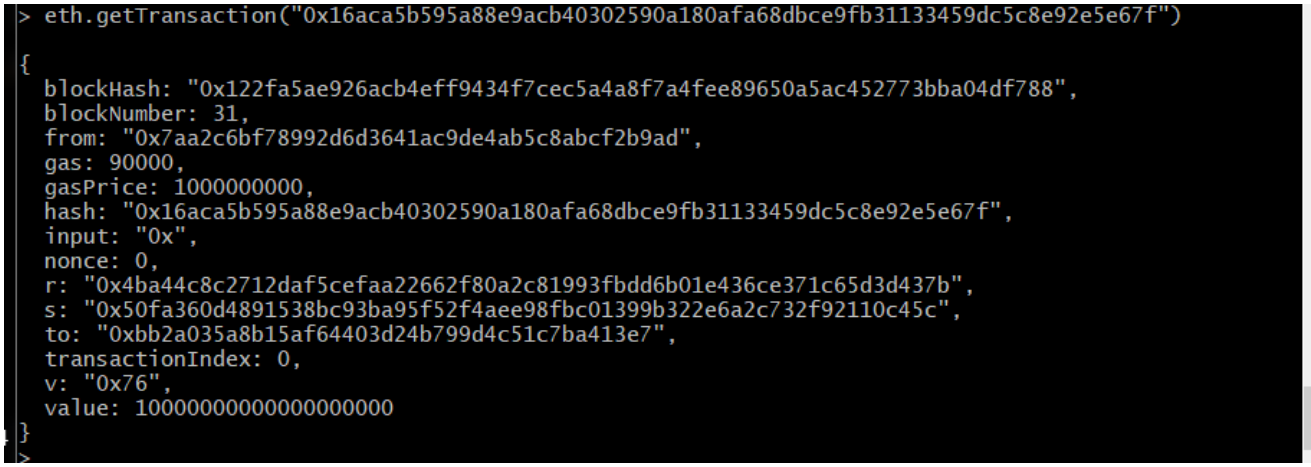
如果只是想本地运行该方法查看返回结果，可采用如下方式获取结果：

```
> helloworld.getStr()

"helloworld"
```

5. 对交易的字段进行解释

通过交易hash查看交易：




```
{
  blockHash: "0x122fa5ae926acb4eff9434f7cec5a4a8f7a4fee89650a5ac452773bba04df788",
  blockNumber: 31,
  from: "0x7aa2c6bf78992d6d3641ac9de4ab5c8abcf2b9ad",
  gas: 90000,
  gasPrice: 1000000000,
  hash: "0x16aca5b595a88e9acb40302590a180afa68dbce9fb31133459dc5c8e92e5e67f",
  input: "0x",
  nonce: 0,
  r: "0x4ba44c8c2712daf5cefaa22662f80a2c81993fbdd6b01e436ce371c65d3d437b",
  s: "0x50fa360d4891538bc93ba95f52f4aee98fbc01399b322e6a2c732f92110c45c",
  to: "0xbb2a035a8b15af64403d24b799d4c51c7ba413e7",
  transactionIndex: 0,
  v: "0x76",
  value: 10000000000000000000
}
```


以此为例对交易的字段进行解释：

`blockHash`：交易所属区块的哈希值；

`blockNumber`：交易所属区块的编号；

`from`：交易发起者；

`gas`：交易发起者为本交易支出的 gas 量，多余的会返还给发起者；

`gasPrice`：交易发起者为每个 gas 付出的 Wei 量，Wei 是计价单位，在实际换算成以太币需要去掉 18 个 0；

`hash`：交易的哈希值；

`input`：包含 data 的字节码；

`nonce`：该交易是发起者的第几笔交易，由发起人EOA发出的序列号，用于防止重播消息；

`r、s、v`：发起人EOA的ECDSA签名的三个组成部分；

`to`：以太币接收方，为 null 表示这个交易用于部署智能合约；

`transactionIndex`：交易索引；

`value`：交易的以太币个数，如果为 0 表示这个交易用于部署智能合约；