南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

区块链课程实验报告

Ex5

计算机科学与技术 2211290 姚知言

信息安全 2211985 李佳璐

2024 年 12 月 11 日

目录

一、 实验过程

(一) 安装 Ganache CLI 并运行节点

```
C:\Users\DELL>npm install -g ganache-cli
npm warn deprecated ganache-cli@6.12.2: ganache-cli is now ganache; visit https://trfl.io/g7 for details

added 1 package in 46s

2 packages are looking for funding
  run `npm fund` for details

C:\Users\DELL>ganache-cli --version
Ganache CLI v6.12.2 (ganache-core: 2.13.2)

C:\Users\DELL>ganache-cli
Ganache CLI v6.12.2 (ganache-core: 2.13.2)
(node:35164) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)

Available Accounts
==================
(0) 0x1D0AE0977C8d168dEd09873512EBa69015CB119F (100 ETH)
(1) 0x1B0f88362ca3Ce23Cc134659875E843A95E17F44 (100 ETH)
(2) 0x120E25B7D9D6F564F2aBb3E8956a33b8F0530D89 (100 ETH)
(3) 0x2EC615946A3dAaAF5be43C24798B351b3eB76bca (100 ETH)
(4) 0x6338D6204B2E3bF2DBdb47b7a241Af3b517a0cdd (100 ETH)
(5) 0x33414A90A427f979989cB0c480C9c3E1429c64cF (100 ETH)
(6) 0x448BFA175688cb55d084B40e2b288E6562be28eF (100 ETH)
(7) 0xAB8fDF298d0Dd8A79F0d9127A2DDe213fAcD7E4B (100 ETH)
(8) 0x9e0FED9A468921AE64Fdf6855202099be048d1A8 (100 ETH)
(9) 0x5A20845010b39E6c5D3fEBdaBbC69b70FFEC3CE2 (100 ETH)
```

图 1: Ganache CLI

(二) 编译文件

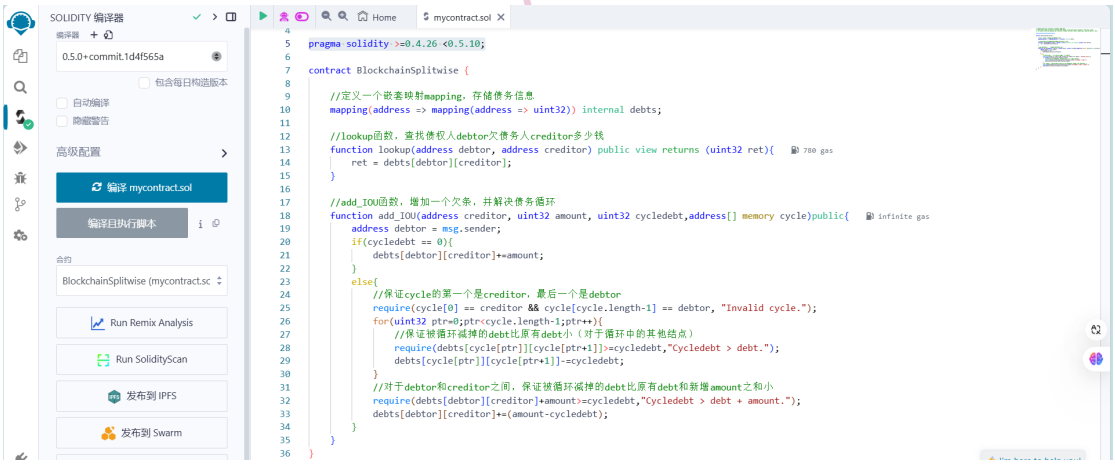

图 2: mycontract.sol

设置编译器版本，进行编译。并复制 ABI 到 script.js 。

(三) 部署

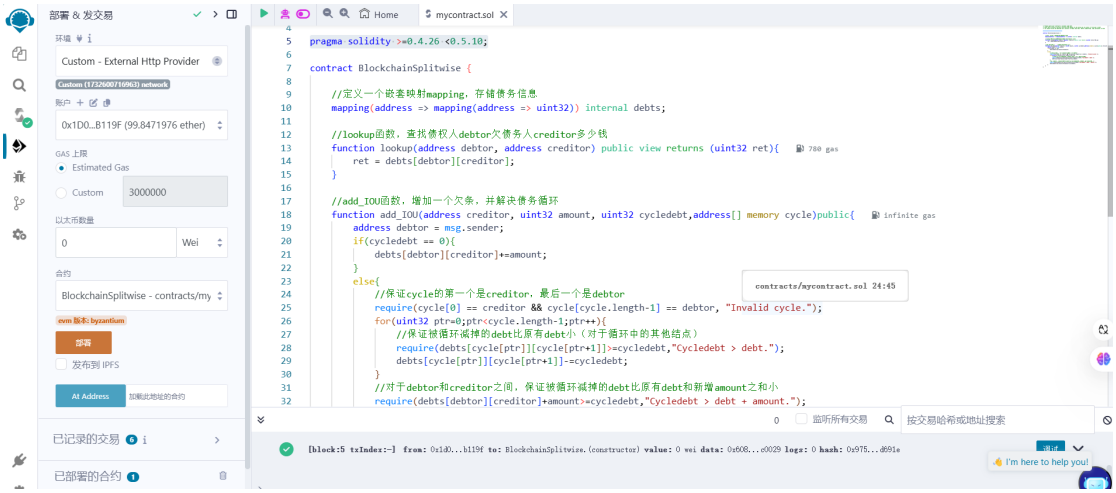

图 3: 部署

设置环境 Custom 进行部署，并复制已部署合约的地址到 script.js 的 contractAddress。

(四) 运行 DApp 程序

图 4: DApp 程序

1. 循环债务

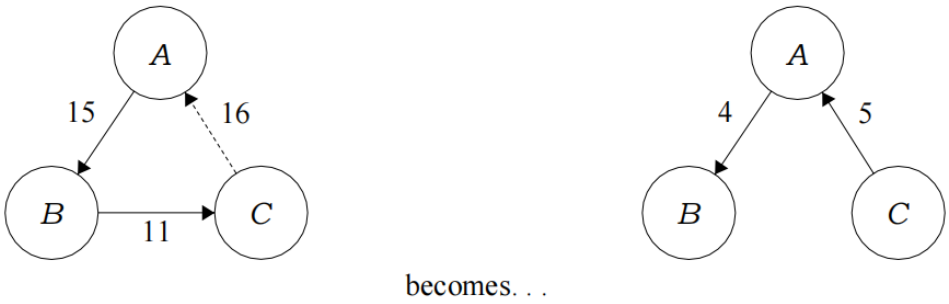

图 5: 示例交易图

使用地址

A：0x1d0ae0977c8d168ded09873512eba69015cb119f

B：0x1b0f88362ca3ce23cc134659875e843a95e17f44

C：0x120e25b7d9d6f564f2abb3e8956a33b8f0530d89

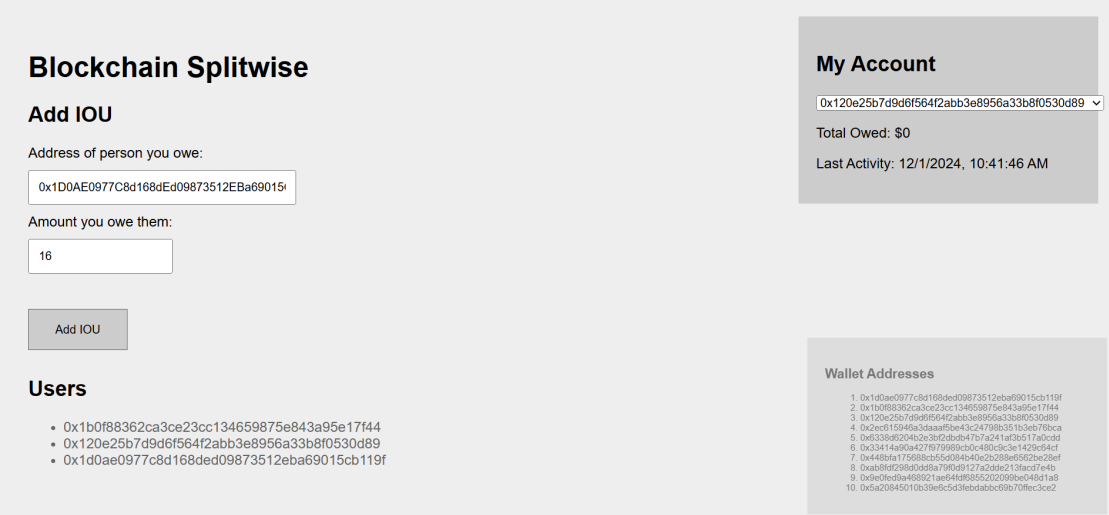

图 6: 添加 A->B 和 B->C

添加后:

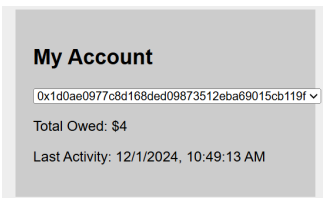

图 7: A 账户

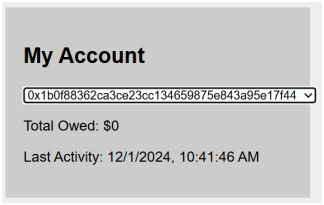

图 8: B 账户

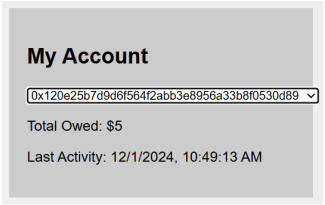

图 9: C 账户

与上述交易图相符。

2. 2 人循环债务情况

D：0x2ec615946a3daaaf5be43c24798b351b3eb76bca

E：0x6338d6204b2e3bf2dbdb47b7a241af3b517a0cdd

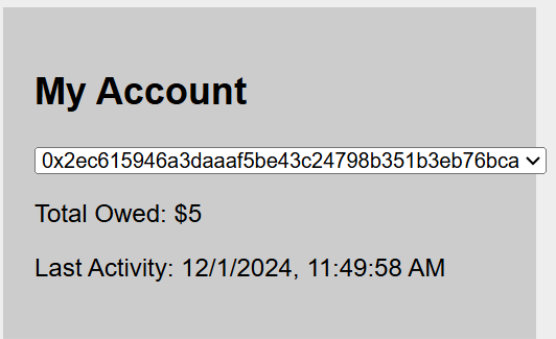

图 10: D->E

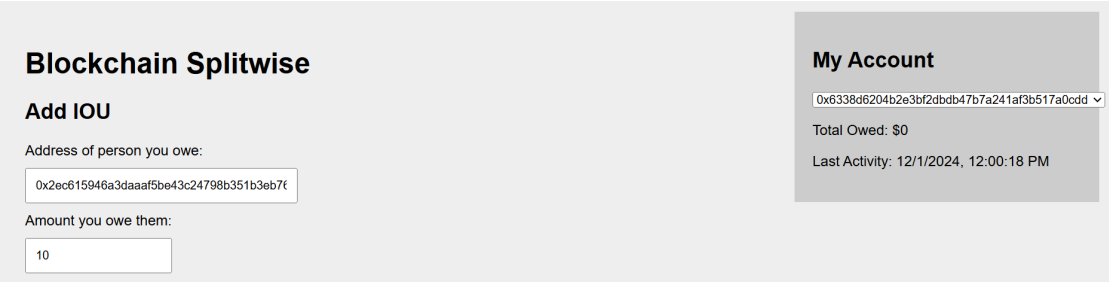

图 11: 添加 D->E

添加 D->E 后，D 对 E 的债务清零，E 对 D 债务减为 5:

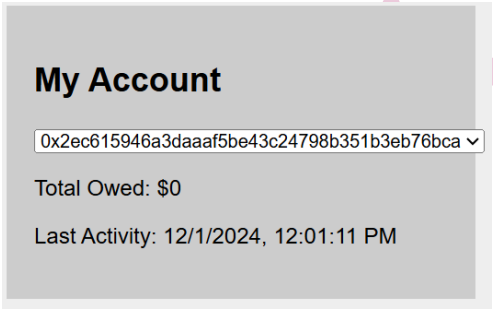

图 12: D 账户

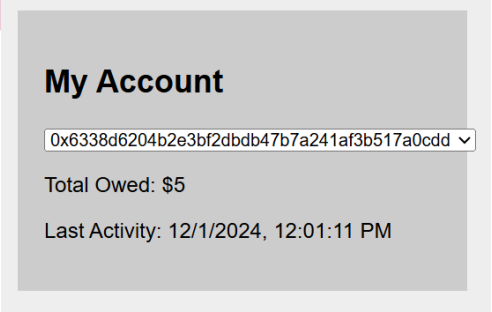

图 13: E 账户

二、 实验代码

项目有两个主要部分: 一个智能合约 mycontract.sol，用 Solidity 编写并在区块链上运行；以及一个在 web 浏览器上本地运行的客户端，使用 script.js 观察区块链，并且可以调用智能合约中的函数。

(一) 智能合约 mycontract.sol

代码实现了 BlockchainSplitwise 智能合约，用于管理区块链上的欠条信息。它通过记录债务关系，允许用户在链上添加、查询和处理欠款信息，特别是处理债务循环的情况。

```
//定义一个嵌套映射mapping，存储债务信息
    mapping(address => mapping(address => uint32)) internal debts;
```

- 第一个 address 表示债务人的地址（debtor）。
- 第二个 address 表示债权人的地址（creditor）。
- uint32 表示债务金额（债务人欠债权人的金额）。

```
    //lookup函数，查找债权人debtor欠债务人creditor多少钱
    function lookup(address debtor, address creditor) public view returns (uint32 ret){
        ret = debts[debtor][creditor];
    }
```

- debts[debtor][creditor] 会返回债务金额

```
    //add_IOU函数，增加一个欠条，并解决债务循环
    function add_IOU(address creditor, uint32 amount, uint32 cycledebt, address[] memory cycle)public{
        address debtor = msg.sender;
        if(cycledebt == 0){
            debts[debtor][creditor]+=amount;
        }
        else{
            //保证cycle的第一个是creditor，最后一个是debtor
            require(cycle[0] == creditor && cycle[cycle.length-1] == debtor, "Invalid cycle.");
            for(uint32 ptr=0;ptr<cycle.length-1;ptr++){
                //保证被循环减掉的debt比原有debt小（对于循环中的其他结点）
                require(debts[cycle[ptr]][cycle[ptr+1]]>=cycledebt,"Cycledebt > debt.");
                debts[cycle[ptr]][cycle[ptr+1]]-=cycledebt;
            }
            //对于debtor和creditor之间，保证被循环减掉的debt比原有debt和新增amount之和小
            require(debts[debtor][creditor]+amount>=cycledebt,"Cycledebt > debt + amount.");
            debts[debtor][creditor]+=(amount-cycledebt);
        }
    }
```

- 如果 cycledebt 为 0，表示没有循环债务，那么直接增加 debtor 向 creditor 的欠款 amount。
- 如果存在循环债务（cycledebt > 0），需要进行循环债务的清理。这个过程中，函数会：
  检查 cycle 数组的合法性，要求 cycle[0] 是 creditor，cycle[cycle.length - 1] 是 debtor。

通过一个 for 循环遍历整个循环（从 cycle[0] 到 cycle[n-1]），逐个检查循环债务的合法性，并从循环中的每一对债务人和债权人之间减少 cycledebt 的债务。

最后检查债务人（debtor）和债权人（creditor）之间的债务金额是否合理，确保新增的债务金额与循环债务之和不会超出原债务金额。

(二) DApp 程序客户端 script.js

1. 获取指定用户的所有债权人

此函数在 doBFS 调用时作为参数，获取地址（结点）的所有债权人作为其邻居结点。

```
function getCreditors(user){
        var creditors = [];
        const users = getUsers();
        for (var i = 0; i < users.length; i++)
                if(BlockchainSplitwise.lookup(user, users[i]).toNumber() > 0)
                        creditors.push(users[i]);
        return creditors;
}
```

2. 获取系统中的所有用户（包括债务人和债权人）

calls（getAllFunctionCalls 的返回值）的 from 属性存储了债务人的地址，第 0 个参数存储了债权人的地址。遍历函数调用记录，将以上信息存入一个 Set 中，最后将 Set 转为数组。

```
function getUsers() {
    var users = new Set();
    var calls = getAllFunctionCalls(contractAddress, 'add_IOU');//找到所有调
        用addIOU的记录
    for (var i = 0; i < calls.length; i++) {
        users.add(calls[i].from);//debtor
        users.add(calls[i].args[0]);//creditor
    }
    return Array.from(users);//from set to array
}
```

3. 获取指定用户所欠的总金额

遍历所有用户，将 user 的所有债务金额累加。

```
function getTotalOwed(user) {
    var tot = 0;
    const users = getUsers();
    for (var i = 0; i < users.length; i++){
        tot += BlockchainSplitwise.lookup(user, users[i]).toNumber();
    }
    return tot;
}
```

4. 获取指定用户最后一次发送或接收欠条的时间戳

遍历所有当前合约调用 add_IOU 函数的记录，若记录的 debtor 或 creditor 为当前用户，则更新时间戳。

```
function getLastActive(user) {
    addressOfContract = contractAddress;
    functionName = 'add_IOU';
    var lasttimestamp = 0;
    var curBlock = web3.eth.blockNumber;
    while (curBlock !== GENESIS) {
        var b = web3.eth.getBlock(curBlock, true);
        var txns = b.transactions;
        for (var j = 0; j < txns.length; j++) {
            var txn = txns[j];
            // check that destination of txn is our contract
            if (txn.to === addressOfContract.toLowerCase()) {
                var func_call = abiDecoder.decodeMethod(txn.input);
                // check that the function getting called in this txn is '
                    functionName'
                if (func_call && func_call.name === functionName) {
                    var args = func_call.params.map(function (x) {return x.
                        value});
                    if(txn.from == user || args[0] == user){
                        if (b.timestamp > lasttimestamp)
                            lasttimestamp = b.timestamp;// 更新lasttimestamp
                    }
                }
            }
        }
        curBlock = b.parentHash;
    }
    return lasttimestamp;
}
```

5. 为当前用户添加一条新的欠条

调用 doBFS 函数寻找环路，遍历环路计算 cycledebt（即环路中的最小值），最终调用合约中 add_IOU 函数。

```
function add_IOU(creditor, amount) {
    const debtor = web3.eth.defaultAccount;//当前账户
    var cycle = doBFS(creditor,debtor,getCreditors);//寻找环路
    var cycledebt;
    if(cycle != null){
        cycledebt = (BlockchainSplitwise.lookup(debtor, creditor).toNumber())
            +amount;//不能超过原有欠款与新增欠款之和
        for (var i=0; i < cycle.length-1 ; i++){
            var nowdebt = BlockchainSplitwise.lookup(cycle[i], cycle[i+1]).
                toNumber();
```

```
            if(cycledebt > nowdebt)
                cycledebt = nowdebt;
        }
    }
    else{
        cycledebt = 0;
        cycle = [];
    }
    BlockchainSplitwise.add_IOU(creditor, amount, cycledebt, cycle);
    return;
}
```

6. 框架代码调整

为适配我们的合约设计版本，需要对框架中 getAllFunctionCalls 函数进行以下调整:

```
//原代码
if (txn.to === addressOfContract)
//调整后的代码
if (txn.to === addressOfContract.toLowerCase())
```

调整后的代码不再有因字母大小写不一致而无法匹配的问题，代码得到正确执行。

三、 实验心得

(一) 数据结构优化

智能合约中数据结构的设计直接影响到 gas 的消耗。在我们的合约中，最重要的数据结构是嵌套映射:

```
function add_IOU(creditor, amount) {
mapping(address => mapping(address => uint32)) internal debts;
```

这种设计能够高效地存储和查询每个债务人和债权人之间的债务金额。相较于直接使用数组或其他复杂数据结构，映射（mapping）在以太坊中提供了 O(1) 的查找时间复杂度，这大大减少了查询和更新数据时的计算开销。

(二) 避免不必要的计算和存储操作

在智能合约中，每一次存储和计算都需要消耗 gas，因此我们尽量避免重复计算和不必要的存储操作。例如，在 add_IOU 函数中，我们设计了一个优化的循环债务处理方式:

```
if(cycledebt == 0) {
    debts[debtor][creditor] += amount;
} else {
    // 循环债务的处理
    require(cycle[0] == creditor && cycle[cycle.length - 1] == debtor, "
        Invalid cycle.");
    for(uint32 ptr = 0; ptr < cycle.length - 1; ptr++) {
```

```
        require(debts[cycle[ptr]][cycle[ptr + 1]] >= cycledebt, "Cycledebt >
            debt.");
        debts[cycle[ptr]][cycle[ptr + 1]] -= cycledebt;
    }
    require(debts[debtor][creditor] + amount >= cycledebt, "Cycledebt > debt
        + amount.");
    debts[debtor][creditor] += (amount - cycledebt);
}
```

- 只有在确实存在循环债务时才会进行循环债务的清理和更新，避免了不必要的循环和更新操作。
- 对债务金额进行合理的检查，避免了错误的数据更新，也降低了错误操作所需的计算和 gas 费用。
- 避免了多次调用 debts[debtor][creditor]，合并了条件判断，减少了合约执行过程中的计算。

(三) 减少外部调用的频率

合约中有很多辅助函数需要与外部环境进行交互，例如获取用户列表、查询某个用户的债务等。每次调用外部函数都会产生额外的 gas 消耗。在 getCreditors 函数中：

```
function getCreditors(user) {
    var creditors = [];
    const users = getUsers();
    for (var i = 0; i < users.length; i++)
        if(BlockchainSplitwise.lookup(user, users[i]).toNumber() > 0)
            creditors.push(users[i]);
    return creditors;
}
```

在合约内部尽量减少遍历操作，尤其是在有多个用户时。通过减少外部调用，可以显著降低 gas 消耗。对于 getUsers 这类函数，最好把它放在链下执行，只在合约中进行必要的交互。