



南開大學
Nankai University

南开大学

计算机学院和网络空间安全学院

《区块链基础及应用》实验报告

Ex5: 复杂的去中心化应用程序 DAPP

姓名：陆皓喆 学号：2211044 专业：信息安全

姓名：张泽睿 学号：2213873 专业：信息安全

指导教师：苏明

2024 年 12 月 8 日

目录

1 实验目的	2
2 实验流程	2
2.1 运行 Ganache CLI	2
2.2 remix 配置与前端配置	2
2.3 代码设计	5
2.3.1 合约函数设计	5
2.3.2 客户端函数设计	8
3 实验结果	10

1 实验目的

1. 使用 Solidity 和 web3.js 在以太坊 (Ethereum) 上实现一个复杂的去中心化应用程序 (DApp)。
2. 编写一个智能合约和访问它的用户客户端，学习 DApp 的“全栈”开发。
3. 目标是编写一份合同，使这两个合同函数使用的存储和计算量最小化。这将使 gas 成本最小化。
你可以假设交易量足够小，在客户端搜索整个区块链是可行的，但你不应该假设唯一的用户是你钱包里的那些人——换句话说，就是 web3.eth。因为账户并不包含系统中所有可能的用户。
4. 提交的内容将根据它是否正确回答查询，以及是否产生合理的汽油费来评分。在提交之前，请确保将您的合约的 Solidity 代码从 Remix 复制并粘贴到 mycontract.sol 中。

2 实验流程

2.1 运行 Ganache CLI

在控制台输入以下命令：

```
1 ganache-cli
```

节点运行后可以看到如下界面：

```
Starting RPC server
Available Accounts
=====
(0) 0x7B84c69BAEeDf19865A8EF7Afc090ee86557fB0d (1000 ETH)
(1) 0x1362A88AB5D3BE79153a17877C09789651d2EB42 (1000 ETH)
(2) 0x238b6F2e72b74E0C0Cab166ef0BE73b381aa1FBf (1000 ETH)
(3) 0x7aCdcD34E26Bbc1C25335FFF298331D1cF9E8CD (1000 ETH)
(4) 0xA0a784934a2554fc1aB0FFD900c2C7FE6883137c (1000 ETH)
(5) 0x533797e57df1fb9A19679150bd8A769Fa007419 (1000 ETH)
(6) 0x87cBfe0e15e66640054CEO44BD6cF7DCOA10F56 (1000 ETH)
(7) 0x1fd72510E6E49333DBF1a6ad5dA011c4056e492E (1000 ETH)
(8) 0x7601B7e1344CBFFD2aE90c82e4E784D3650fC3A (1000 ETH)
(9) 0x2724eB0b184875247b64B4dE767AaE0886c25DDe (1000 ETH)

Private Keys
=====
(0) 0xb7dfcd7df2ae8a3a0462f90ddd559de41d896fffc17f4619327b15528b6ebe994
(1) 0xe0af8634aff1382d8543042c67857b3dfa729fdbd77c4422b6ed2184820a7742
(2) 0x9751ea1fe5feb422cc0efed4f05e409cb5f957f659b9ceee4a09e5997a9a4e26
(3) 0xafeaa0697c33ba0d181a505caeal65b406696040248e23bfcd6872acc33f358a
(4) 0xc3ffe0d0fb80d90d49602854eb0821746bd5f3d5c58aacd85aba7b7c473820fe
(5) 0x19357e23dd9fc182b2bf3cfb485e378602add65340c5157894f681f54825d0b7
(6) 0xf9179b35cbd0ae1161d64b79d92a522e85c315763a1b7aac1b0f9fc58f7281df
(7) 0x7ac32f9a5eb1c76983be83131895c295d27768082220f509a2e229d698b3628d
(8) 0xda0fb0e4f24e5fb3761d3faf2d69c2b96402c4add39c656060985b8c6336a22a
(9) 0xa854eab419fe477748967d3bc58910e7cc66dff8e6f6ca668d061d1491c7261b

HD Wallet
=====
```

图 2.1: 运行命令获得对应的地址

2.2 remix 配置与前端配置

打开网站 <https://remix.ethereum.org/>，新建文件 BlockchainSplitwise.sol，并编写 contract Splitwise 代码。

点击编译按钮开始编译，并赋值对应的 ABI：

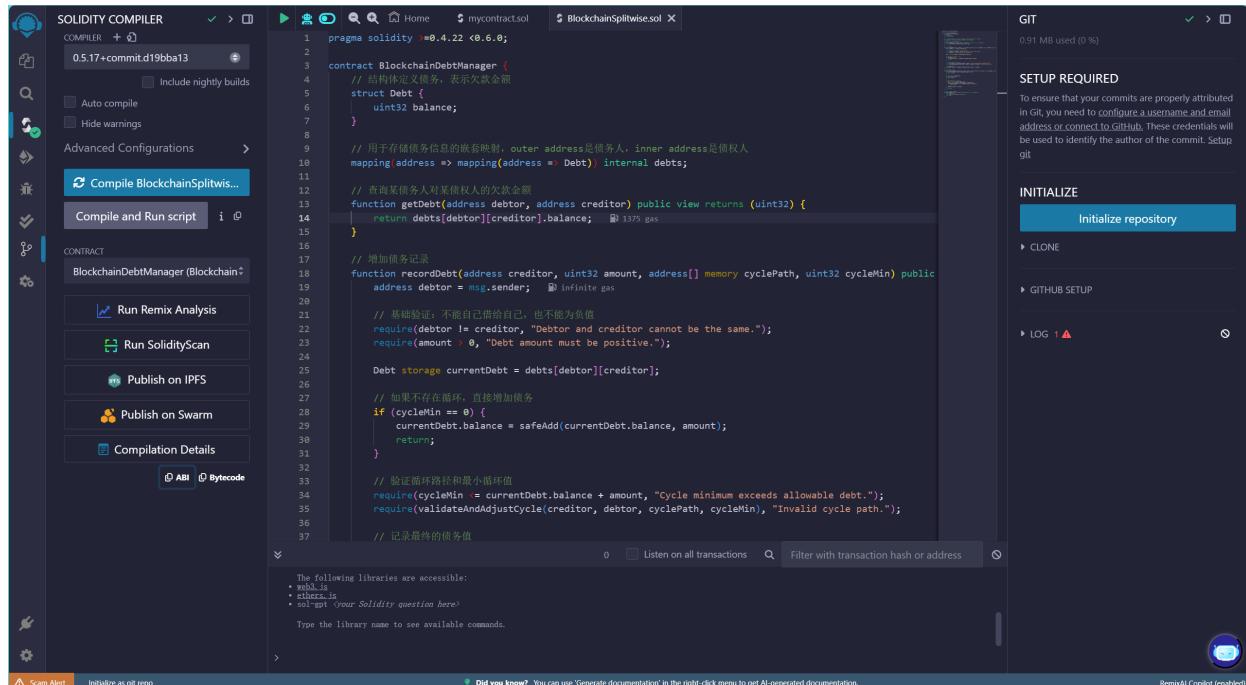


图 2.2: 编译运行

点击部署并赋值 address:

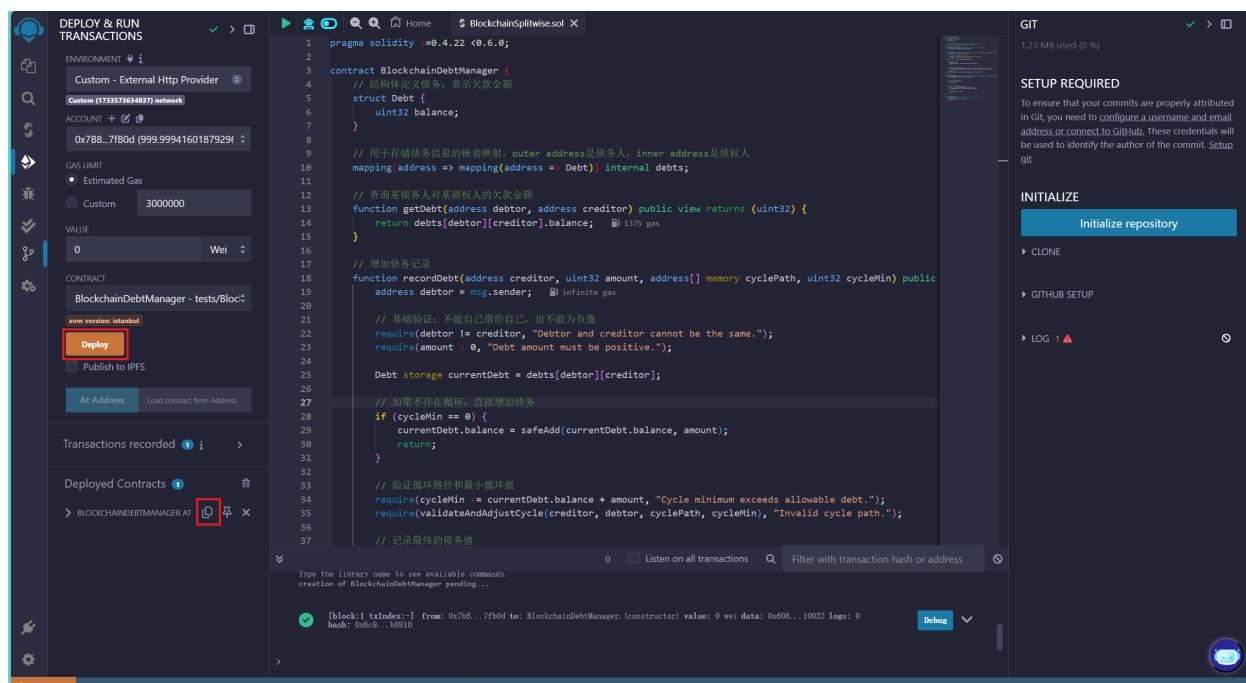


图 2.3: 部署并赋值地址

我们将对应的 ABI 和 address 复制进 js 脚本中:

```

1 var abi = [
2   {
3     "constant": false,

```

```
4     "inputs": [
5         {
6             "internalType": "address",
7             "name": "creditor",
8             "type": "address"
9         },
10        {
11            "internalType": "uint32",
12            "name": "amount",
13            "type": "uint32"
14        },
15        {
16            "internalType": "address[]",
17            "name": "path",
18            "type": "address[]"
19        },
20        {
21            "internalType": "uint32",
22            "name": "min_on_cycle",
23            "type": "uint32"
24        }
25    ],
26    "name": "add_IOU",
27    "outputs": [],
28    "payable": false,
29    "stateMutability": "nonpayable",
30    "type": "function"
31},
32{
33    "constant": true,
34    "inputs": [
35        {
36            "internalType": "address",
37            "name": "debtor",
38            "type": "address"
39        },
40        {
41            "internalType": "address",
42            "name": "creditor",
43            "type": "address"
44        }
45    ],

```

```

46     "name": "lookup",
47     "outputs": [
48       {
49         "internalType": "uint32",
50         "name": "ret",
51         "type": "uint32"
52       }
53     ],
54     "payable": false,
55     "stateMutability": "view",
56     "type": "function"
57   }
58 ];
59
60 var contractAddress = '0x68c49C93fA54a6691BE504cA2fBfca382b7de4B1';

```

2.3 代码设计

2.3.1 合约函数设计

我们设计的合约函数如下所示：

```

1 pragma solidity >=0.4.22 <0.6.0;

2
3 contract BlockchainDebtManager {
4   // 结构体定义债务，表示欠款金额
5   struct Debt {
6     uint32 balance;
7   }
8
9   // 用于存储债务信息的嵌套映射，outer address 是债务人，inner address 是债权人
10  mapping(address => mapping(address => Debt)) internal debts;
11
12  // 查询某债务人对某债权人的欠款金额
13  function getDebt(address debtor, address creditor) public view returns
14    (uint32) {
15    return debts[debtor][creditor].balance;
16  }
17
18  // 增加债务记录
19  function recordDebt(address creditor, uint32 amount, address[] memory
→ cyclePath, uint32 cycleMin) public {
    address debtor = msg.sender;

```

```
20
21 // 基础验证: 不能自己借给自己, 也不能为负值
22 require(debtor != creditor, "Debtor and creditor cannot be the same.");
23 require(amount > 0, "Debt amount must be positive.");
24
25 Debt storage currentDebt = debts[debtor][creditor];
26
27 // 如果不存在循环, 直接增加债务
28 if (cycleMin == 0) {
29     currentDebt.balance = safeAdd(currentDebt.balance, amount);
30     return;
31 }
32
33 // 验证循环路径和最小循环值
34 require(cycleMin <= currentDebt.balance + amount, "Cycle minimum exceeds
35 ← allowable debt.");
36 require(validateAndAdjustCycle(creditor, debtor, cyclePath, cycleMin),
37 ← "Invalid cycle path.");
38
39 // 记录最终的债务值
40 currentDebt.balance = safeAdd(currentDebt.balance, amount - cycleMin);
41
42 // 验证并调整循环路径
43 function validateAndAdjustCycle(address start, address end, address[] memory
44 ← path, uint32 cycleMin) private returns (bool) {
45     if (start != path[0] || end != path[path.length - 1]) {
46         return false;
47     }
48
49     // 限制路径长度
50     if (path.length > 12) {
51         return false;
52     }
53
54     // 遍历路径并调整循环内的债务值
55     for (uint i = 1; i < path.length; i++) {
56         Debt storage edgeDebt = debts[path[i - 1]][path[i]];
57         if (edgeDebt.balance == 0 || edgeDebt.balance < cycleMin) {
58             return false;
59         }
60         edgeDebt.balance -= cycleMin;
```

```

59     }
60     return true;
61 }
62
63 // 安全的加法操作，避免溢出
64 function safeAdd(uint32 a, uint32 b) internal pure returns (uint32) {
65     uint32 sum = a + b;
66     require(sum >= a, "Addition overflow.");
67     return sum;
68 }
69

```

这段 Solidity 代码实现了一个简单的债务管理系统，其中包括债务的记录、查询和循环债务路径的处理。以下是对代码的详细讲解。

结构体 Debt 结构体

Debt 用于表示每个债务记录，包含一个 balance 字段表示债务金额：

```

1 struct Debt {
2     uint32 balance;
3 }

```

债务映射 debts

debts 是一个嵌套的映射，用来存储债务信息。外层映射的键是债务人 (debtor) 的地址，内层映射的键是债权人 (creditor) 的地址，值是 Debt 结构体，用来存储债务金额。

查询债务 getDebt

getDebt 函数允许用户查询某个债务人对某个债权人的欠款金额，返回的是 uint32 类型的债务余额。

记录债务 recordDebt

recordDebt 函数用于记录一笔新的债务。它接收以下参数：

- **creditor**: 债权人的地址。
- **amount**: 债务金额。
- **cyclePath**: 循环路径（如果存在债务循环）。
- **cycleMin**: 循环路径中每一环节的最小债务值。

函数首先验证债务人和债权人不能是同一人，且债务金额必须大于零。如果没有债务循环 (cycleMin == 0)，则直接增加债务金额。如果存在循环，则验证循环路径和最小债务值，确保路径中的每个债务都能承受最小债务值。

验证并调整循环路径 validateAndAdjustCycle

validateAndAdjustCycle 函数验证并调整债务循环路径。它确保路径的起点和终点正确，路径长度不超过 12 个地址，并且每个路径中的债务都能承受最小债务值 (cycleMin)。如果路径验证成功，则在路径中的每个债务记录中减去 cycleMin。

安全加法 safeAdd

safeAdd 函数是一个安全的加法操作，防止溢出。它会检查加法结果是否溢出，如果溢出则触发异常。

2.3.2 客户端函数设计

我们设计的客户端函数如下所示：

```
1 // 获取所有函数调用数据
2 function getCallData(extractor_fn, early_stop_fn = null) {
3     const results = new Set();
4     const all_calls = getAllFunctionCalls(contractAddress, 'add_IOU',
5         ↪ early_stop_fn);
6     all_calls.forEach(call => {
7         extractor_fn(call).forEach(value => results.add(value));
8     });
9     return Array.from(results);
10 }
11
12 // 获取债权人列表
13 function getCreditors() {
14     return getCallData(call => [call.args[0]]);
15 }
16
17 // 获取特定用户的所有债权人
18 function getCreditorsForUser(user) {
19     return getCreditors().filter(creditor => BlockchainSplitwise.lookup(user,
20         ↪ creditor).toNumber() > 0);
21 }
22
23 // 查找路径中的最小欠款金额
24 function findMinOnPath(path) {
25     return path.slice(1).reduce((minOwed, debtor, i) {
26         const creditor = path[i];
27         const amountOwed = BlockchainSplitwise.lookup(debtor, creditor).toNumber();
28         return minOwed == null || minOwed > amountOwed? amountOwed : minOwed;
29     }, null);
30 }
```

```

29
30 // 获取系统中的所有用户（包括债务人和债权人）
31 function getUsers() {
32     return getCallData(call => [call.from, call.args[0]]);
33 }
34
35 // 获取用户的总欠款金额
36 function getTotalOwed(user) {
37     return getCreditors().reduce((totalOwed, creditor) => totalOwed +
38         BlockchainSplitwise.lookup(user, creditor).toNumber(), 0);
39 }
40
41 // 获取用户的最后活动时间戳
42 function getLastActive(user) {
43     const all_timestamps = getCallData(call => (call.from === user || call.args[0]
44         === user)? [call.timestamp] : []);
45     return all_timestamps.length? Math.max(...all_timestamps) : null;
46 }
47
48 // 向系统中添加一笔 IOU 记录
49 function add_IOU(creditor, amount) {
50     const debtor = web3.eth.defaultAccount;
51     const path = doBFS(creditor, debtor, getCreditorsForUser);
52
53     if (path) {
54         const min_on_cycle = Math.min(findMinOnPath(path), amount);
55         return BlockchainSplitwise.add_IOU(creditor, amount, path, min_on_cycle);
56     }
57     return BlockchainSplitwise.add_IOU(creditor, amount, [], 0);
58 }
```

这段 JavaScript 代码实现了一些关于债务管理的操作，包括获取债务人和债权人的信息、计算总债务、查找循环债务路径、记录 IOU 等功能。

getCallData——获取所有函数调用数据

该函数调用 getAllFunctionCalls 获取所有与 add_IOU 相关的函数调用数据，并通过传入的 extractor_fn 函数提取相关数据。返回去重后的数据列表。

getCreditors——获取所有债权人

通过 getCallData 函数获取所有债权人的列表，债权人是 add_IOU 函数调用中的第一个参数。

getCreditorsForUser——获取特定用户的所有债权人

该函数返回给定用户的所有债权人，过滤出对该用户有债务的债权人。

findMinOnPath——查找路径中的最小债务

此函数遍历路径中的债务，返回路径上最小的欠款金额。

getUsers——获取所有用户（债务人和债权人）

此函数返回所有债务人和债权人的列表。

getTotalOwed——获取用户的总欠款

此函数计算并返回特定用户的总欠款金额。

getLastActive——获取用户的最后活动时间

该函数返回特定用户的最后一次活动时间戳。

add_IOU——增加一笔 IOU 记录

此函数根据债务人和债权人的关系添加一笔 IOU。若存在循环债务路径，则进行路径优化，减少最小债务金额。

3 实验结果

我们主要使用以下三个地址：

- 1 A: 0x94253d0ba8d8586add99c4c7d6ef4b91d0532ab
- 2 B: 0x1fbde13025ff6839e077c6cb56acae503945c0bd
- 3 C: 0xe9dda854ce9effc818afab6e4d59b80b3c6c5e29

用浏览器打开 index.html，首先设置 A 欠 B 15 元：

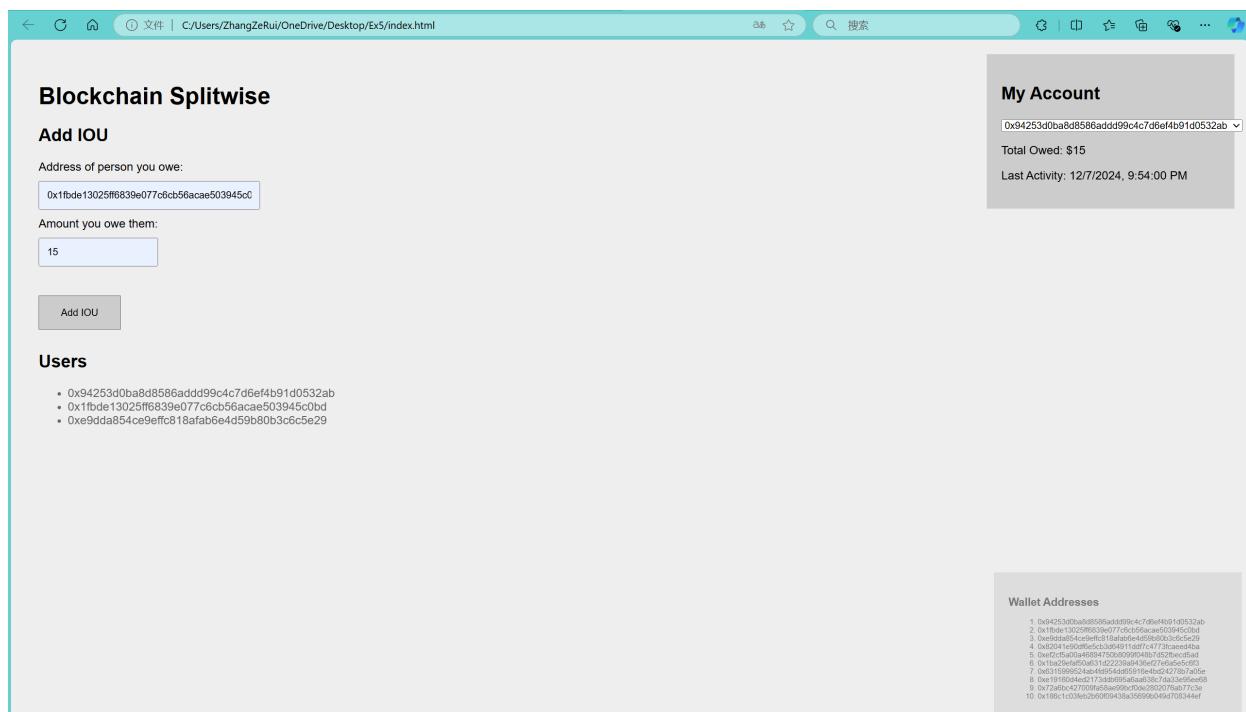


图 3.4: A 欠 B 15 元

然后，设置 B 欠 C 14 元：

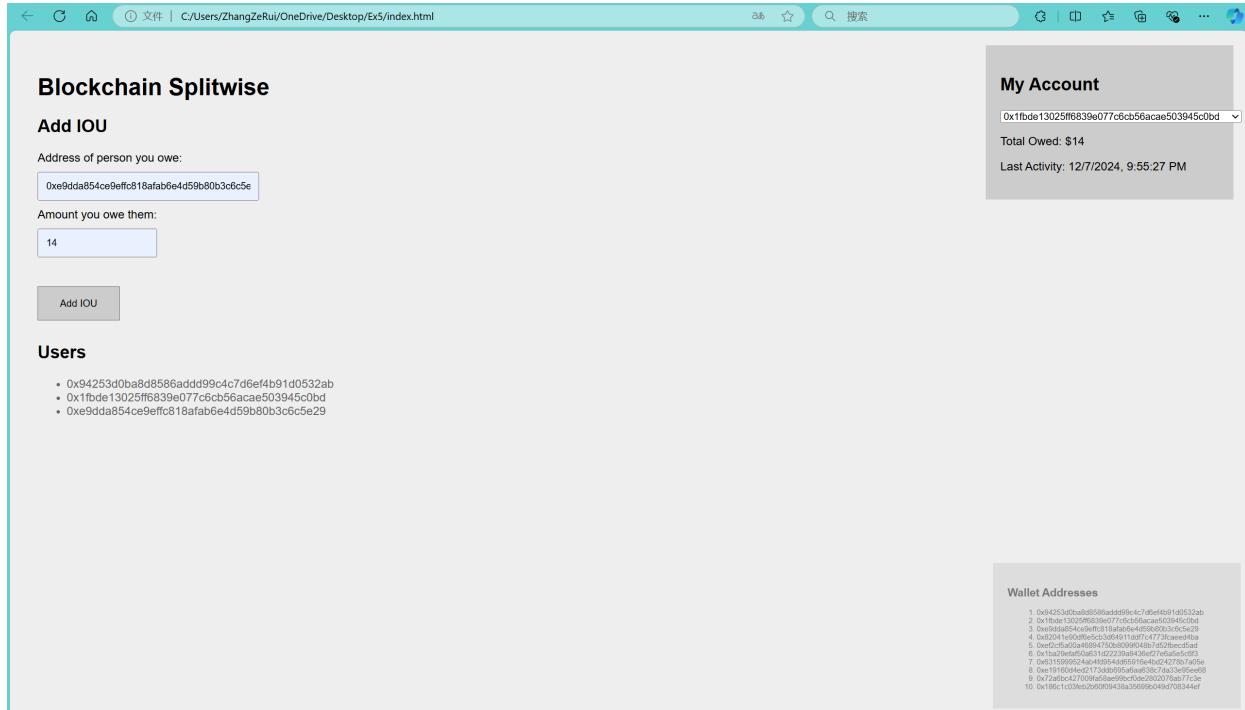


图 3.5: B 欠 C 14 元

最后，设置 C 欠 A 13 元：

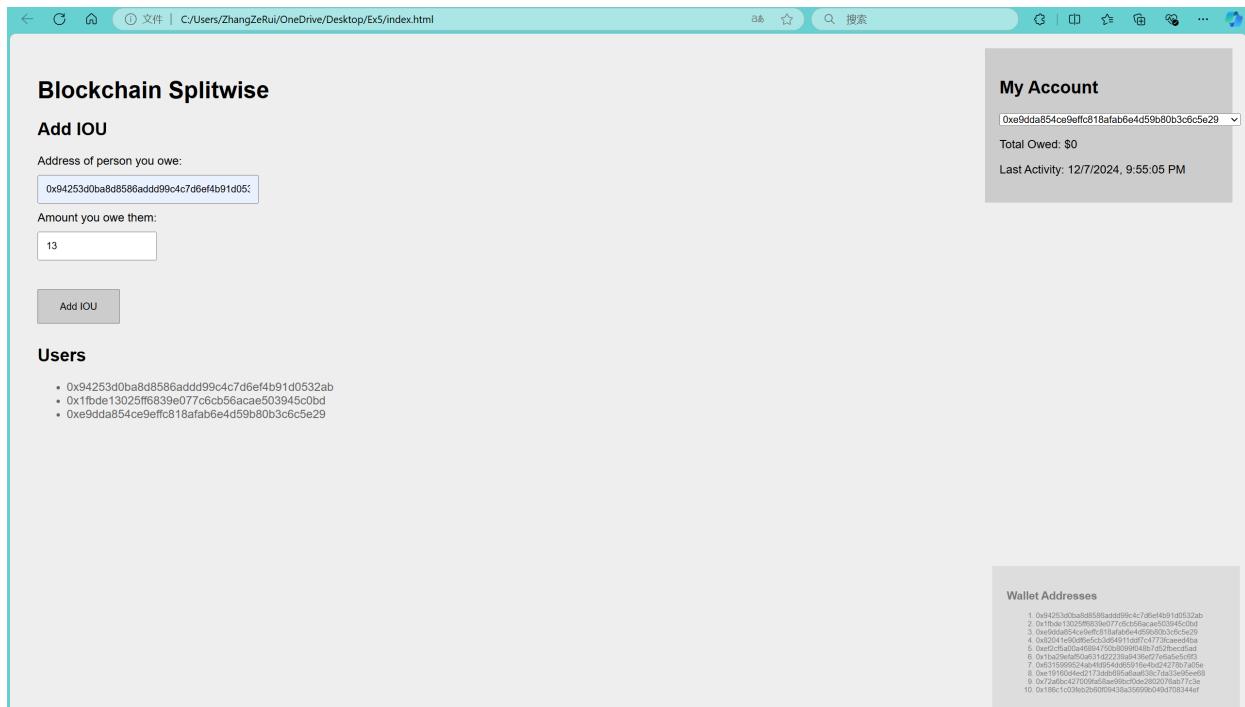


图 3.6: C 欠 A 13 元

上述过程会形成一个环状结构，我们设计的合约会自动消去最小权重，下面是消去最小权重后的结果：

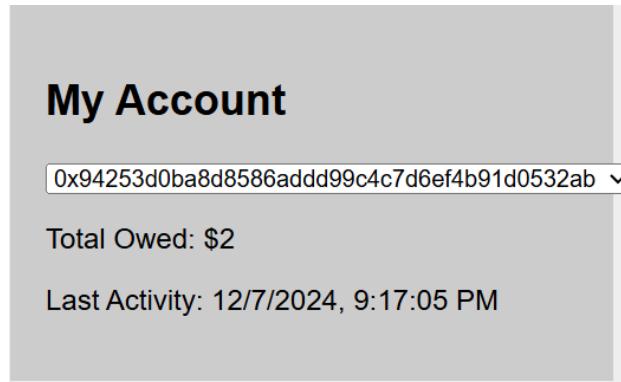


图 3.7: A 的账户

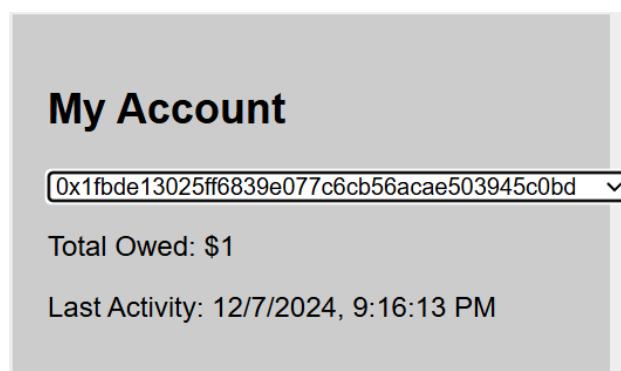


图 3.8: B 的账户

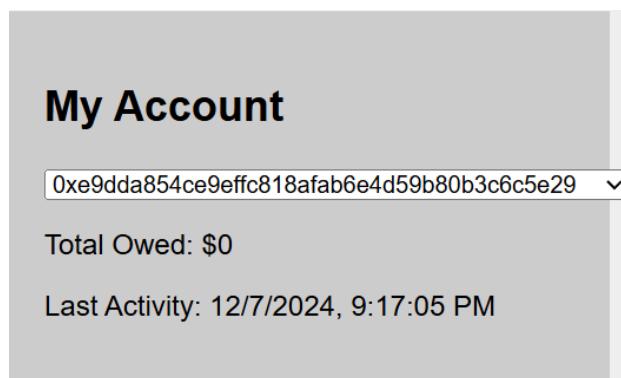


图 3.9: C 的账户

我们发现，经过消去最小的权重，我们原先是 A 欠 B 15 元，B 欠 C 14 元，C 欠 A 13 元，我们设计的合约消去了 C 欠 A 的 13 元，使 C 的欠款变为了 0，然后同时将 B 欠 C 的钱消去了 13 元，剩下 1 元；将 A 欠 B 的钱消去了 13 元，剩下 2 元。这个结果与我们的实验预期相一致。从实验结果可验证我们所设计合约的正确性！