Exercise 5 DApp

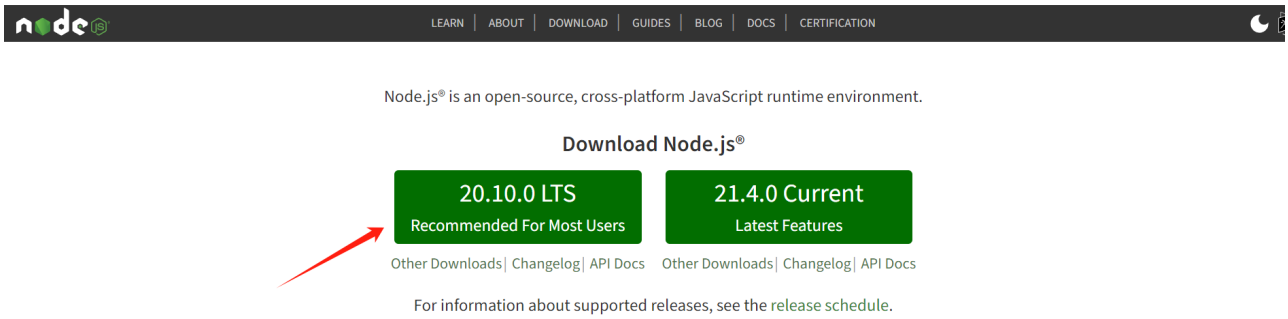
姓名：李帅东　学号：2111231
姓名：齐明杰　学号：2113997

1　实验目的

- 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(DApp)。
- 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。
- 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。
- 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　安装Node.js



2.2　Ganache CLI安装

输入以下命令：

```
npm install -g ganache-cli
```

进行安装。

然后输入以下命令：

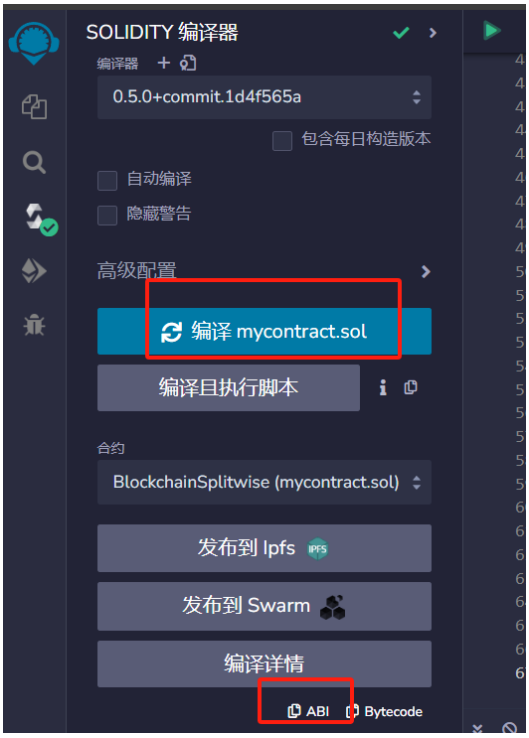
```
ganache-cli
```

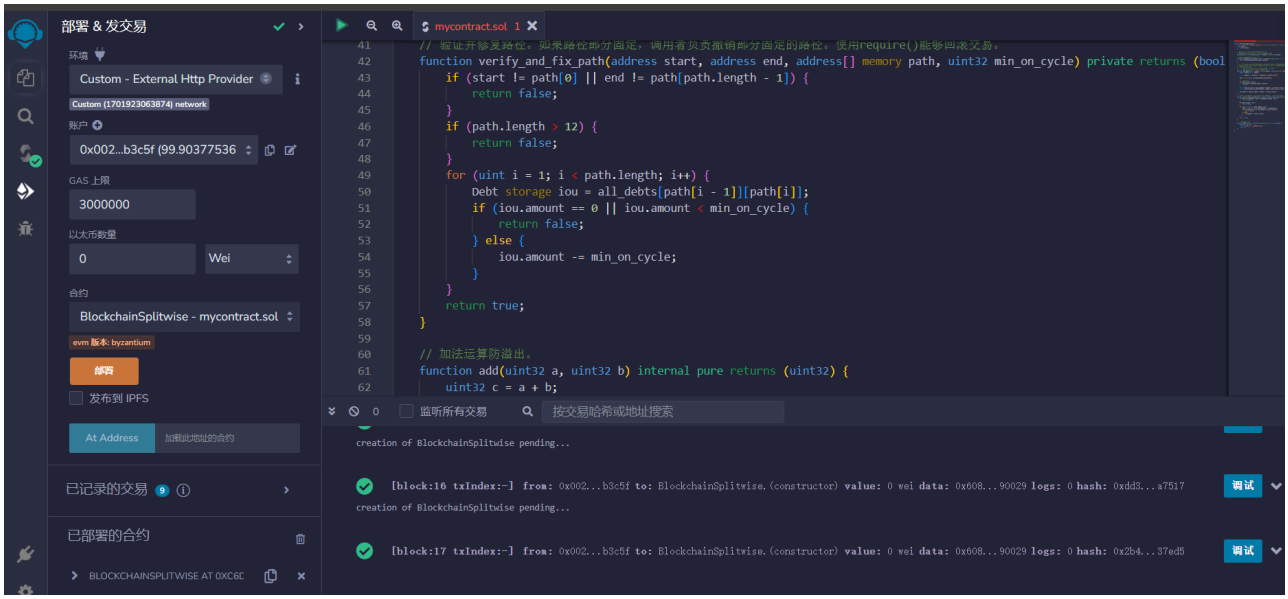
来运行节点，运行之后，可以看到有如下的代码：

```
管理员: C:\windows\system32\cmd.exe - "node"  "C:\Users\Administrator\AppData\Roaming\npm\\node_modules\ganac...
Microsoft Windows [版本 10.0.19045.3693]
(c) Microsoft Corporation。保留所有权利。

D:\study\大三\区块链\实验\Ex5>ganache-cli
Ganache CLI v6.12.2 (ganache-core: 2.13.2)

Available Accounts
==================
(0) 0x3e67026d5AFA4593c76EebC9cC60CD7A4157B754 (100 ETH)
(1) 0x980a7Ec9ce6Bf53eEE796e2Db053f64435d5238c (100 ETH)
(2) 0xE65c7350DC135DBf007b1B5C5569408E6Ca384Ca (100 ETH)
(3) 0xc785B6A8210492460970994C90dcb23A70d818c3 (100 ETH)
(4) 0xB140d5071f8Be73041671Be4BD0953d90d4C1157 (100 ETH)
(5) 0x980811f427d160797143AEC73AA0CF1DB3e8e750 (100 ETH)
(6) 0x08De8636DEE881e94001f845046fC0A895fF3b9E (100 ETH)
(7) 0x2F39ecE42BAb261EC83523B1458Bc56eb04DbfCE (100 ETH)
(8) 0xBb8370c61E22dFdB418E2c956466648Bac5595F9 (100 ETH)
(9) 0xfAfF53a9215207e0365158373905b93A15493d62 (100 ETH)

Private Keys
==================
(0) 0xd43e3c27e26cd0d4c5fbc53d77e9fd91273e40459c6896fc63d148002ebff753
(1) 0xdd176ab02921aa8ef93fdc5bc80f44e3abd2a5b6fc6719aee84e897220994fc2
(2) 0xb3e5520eb6b82b25737d8a003daf52de3fbff90bd4c109e2ac7c3f072a0d0504
(3) 0x2dde5afb5087026b7c062703a7d6cf1a396a08f4be03d781c80a7bb24e0d508b
(4) 0x6941ec68587382cf78f853756d0ca0d6892e982bb38b0df81a503128e262770f
(5) 0x8c6eaa4d020fdc5583dbf2dfaf627aec855c21770198fbdd6e1d94c1718e121a
(6) 0x46e35742c9903390d2d331682fd657f4f7e4b196ea276974396095bc654db1a2
(7) 0x97eeb5472bdacef006490555820be66120246d3502e90ea55b3b9ac2c9a43161
(8) 0xafc60a5884d2358e2af11f4a2a57367f27ef1620b799df882e41cb13e232a444
```

2.3 remix配置与前端配置

- `https://remix.ethereum.org` 打开该网站，按照Word文件中的教程部署
- 新建 `mycontract.sol` 文件,并在该文件中编写 `contract Splitwise` 代码内容详见 `mycontract.sol` 文件。(代码描述详见下文)
- 点击“编译”按钮，开始编译。如下图所示，说明编译成功！





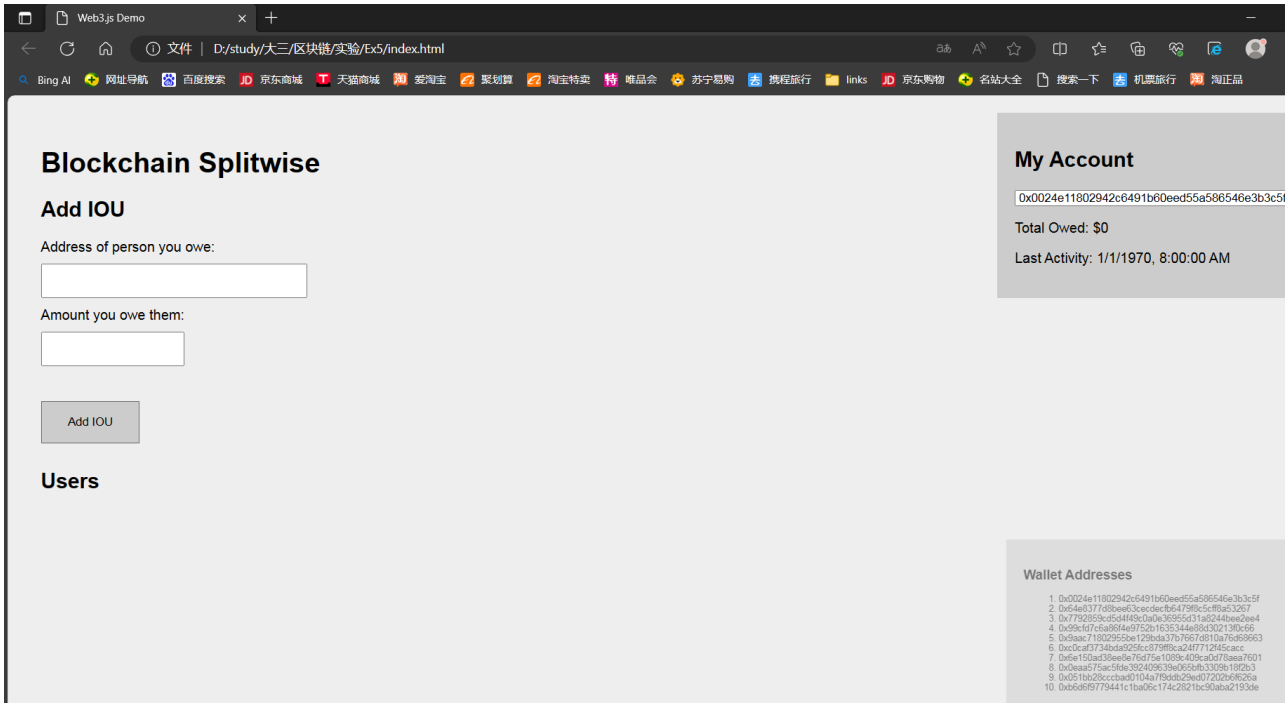
2.4 进入网页

部署完合约后，我们需要把`ABI`和`contractAddress`复制到`script.js`的指定位置中。

```
var abi = [
    {
        "constant": false,
        "inputs": [
            {
                "name": "creditor",
                "type": "address"
            },
            {
                "name": "amount",
                "type": "uint32"
            },
            {
                "name": "path",
                "type": "address[]"
            },
            {
                "name": "min_on_cycle",
                "type": "uint32"
            }
        ],
        "name": "add_IOU",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
```

```
        "constant": true,
        "inputs": [
            {
                "name": "debtor",
                "type": "address"
            },
            {
                "name": "creditor",
                "type": "address"
            }
        ],
        "name": "lookup",
        "outputs": [
            {
                "name": "ret",
                "type": "uint32"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }
];

var contractAddress = '0xEc9DC7C583Aa83718CecC6f84EE46B571f488F65'
```

然后直接点击index.html进入网页即可开始实验：



3 代码设计

在设计此去中心化应用（DApp）时，核心目标是实现一个高效且成本效益高的债务追踪系统。智能合约`BlockchainSplitwise`采用Solidity编写，致力于最小化存储和计算量，以降低gas成本。

3.1 合约函数

```
pragma solidity >=0.4.22 <0.6.0;

contract BlockchainSplitwise {
    // 债务结构，代表所欠的金额。最大债务为2^32 ~= 40亿。合约中会检查溢出。
    struct Debt {
        uint32 amount;
    }

    // 跟踪债务。映射从债务人到债权人及其债务。
    // 例如，debts[Alice][Bob] = 10 表示Alice欠Bob 10。
    mapping(address => mapping(address => Debt)) internal all_debts;

    // 查询债务人欠债权人的总债务额。
    function lookup(address debtor, address creditor) public view returns (uint32 ret) {
        ret = all_debts[debtor][creditor].amount;
    }

    // 添加债务，记录msg.sender欠债权人更多金额。path 可能是从债权人到msg.sender的现有路径，表示通过添加此IOU将创建一个循环。
    // 'min_on_cycle' 是提议的循环中最小金额，将从所有债务中减去（包括正在添加的债务）。
    // 函数验证以下内容：
    //    1. 如果给出路径，它确实存在并连接债权人和债务人。
    //    2. min_on_cycle 必须是循环中的最小值。
    function add_IOU(address creditor, uint32 amount, address[] memory path, uint32 min_on_cycle) public {
        address debtor = msg.sender;

        require(debtor != creditor, "债权人不能是债权人自己。");

        Debt storage iou = all_debts[debtor][creditor];

        // 检查溢出。
        if (min_on_cycle == 0) {
            iou.amount = add(iou.amount, amount);
            return;
        }
```

```
35          require(min_on_cycle <= (iou.amount + amount), "创建的循环中最小值
    不能大于金额。");
36          require(verify_and_fix_path(creditor, debtor, path, min_on_cycle),
    "提供的路径不正确。");
37
38          iou.amount = add(iou.amount, (amount - min_on_cycle));
39      }
40
41      // 验证并修复路径。如果路径部分固定，调用者负责撤销部分固定的路径。使用
    require()能够回滚交易。
42      function verify_and_fix_path(address start, address end, address[]
    memory path, uint32 min_on_cycle) private returns (bool ret) {
43          if (start != path[0] || end != path[path.length - 1]) {
44              return false;
45          }
46          if (path.length > 12) {
47              return false;
48          }
49          for (uint i = 1; i < path.length; i++) {
50              Debt storage iou = all_debts[path[i - 1]][path[i]];
51              if (iou.amount == 0 || iou.amount < min_on_cycle) {
52                  return false;
53              } else {
54                  iou.amount -= min_on_cycle;
55              }
56          }
57          return true;
58      }
59
60      // 加法运算防溢出。
61      function add(uint32 a, uint32 b) internal pure returns (uint32) {
62          uint32 c = a + b;
63          require(c >= a, "加法运算溢出。");
64          return c;
65      }
66  }
67
```

智能合约`BlockchainSplitwise`的设计旨在提供一个去中心化的债务追踪系统。该合约中的主要元素和逻辑包括：

1. **债务结构（Debt）**：结构体用于存储单一债务的金额，考虑到以太坊的限制，采用了uint32类型，并在合约中实现了溢出检查。
2. **债务映射**: 使用嵌套映射结构（`mapping(address => mapping(address => Debt))`）来存储和跟踪债务人和债权人之间的债务关系。

3. **查询函数（lookup）**：允许查询特定债务人欠给特定债权人的债务总额。这是一个只读函数，不会修改合约状态。
4. **添加债务函数（add_IOU）**：实现了债务人向债权人添加新债务的功能。特别考虑了债务循环的问题，提供了解决循环的机制。
5. **路径验证和修复（verify_and_fix_path）**：用于验证和修正债务人和债权人之间的债务路径。这个私有函数确保了债务关系的逻辑一致性。
6. **防溢出的加法（add）**：安全的加法运算，确保数据操作的安全性。

3.2 客户端函数

```
// TODO: Add any helper functions here!
/**
 * 获取调用数据
 * @param {Function} extractor_fn - 用于提取数据的函数
 * @param {Function} early_stop_fn - 用于提前停止的函数
 * @returns {Array} - 提取的数据数组
 */
function getCallData(extractor_fn, early_stop_fn) {
    const results = new Set();
    const all_calls = getAllFunctionCalls(contractAddress, 'add_IOU', early_stop_fn);
    for (var i = 0; i < all_calls.length; i++) {
        const extracted_values = extractor_fn(all_calls[i]);
        for (var j = 0; j < extracted_values.length; j++) {
            results.add(extracted_values[j]);
        }
    }
    return Array.from(results);
}

/**
 * 获取债权人列表
 * @returns {Array} 债权人列表
 */
function getCreditors() {
    return getCallData((call) => {
        // call.args[0] is the creditor.
        return [call.args[0]];
    }, /*early_stop_fn=*/null);
}

/**
 * 获取用户的债权人列表
 * @param {string} user - 用户名
 * @returns {Array} - 债权人列表
 */
```

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


/**
 * 在给定路径上查找最小欠款金额。
 *
 * @param {Array} path - 路径数组，表示债务人和债权人之间的关系。
 * @returns {number} - 最小欠款金额。
 */
function findMinOnPath(path) {
    var minOwed = null;
    for (var i = 1; i < path.length; i++) {
        const debtor = path[i-1]
        const creditor = path[i];
        const amountOwed = BlockchainSplitwise.lookup(debtor,
creditor).toNumber();
        if (minOwed == null || minOwed > amountOwed) {
            minOwed = amountOwed;
        }
    }
    return minOwed;
}


// TODO: Return a list of all users (creditors or debtors) in the system
// You can return either:
//   - a list of everyone who has ever sent or received an IOU
// OR
//   - a list of everyone currently owing or being owed money
/**
 * 获取用户信息。
 * @returns {Array} 包含债务人和债权人的数组。
 */
function getUsers() {
    return getCallData((call) => {
```

```
        // call.from is debtor and call.args[0] is creditor.
        return [call.from, call.args[0]]
    }, /*early_stop_fn=*/null);
}

// TODO: Get the total amount owed by the user specified by 'user'
/**
 * 计算用户欠款总额
 * @param {string} user - 用户名
 * @returns {number} - 用户欠款总额
 */
function getTotalOwed(user) {
    // We assume lookup is up-to-date (all cycles removed).
    var totalOwed = 0;
    const all_creditors = getCreditors();
    for (var i = 0; i < all_creditors.length; i++) {
        totalOwed += BlockchainSplitwise.lookup(user,
all_creditors[i]).toNumber();
    }
    return totalOwed;
}

// TODO: Get the last time this user has sent or received an IOU, in
seconds since Jan. 1, 1970
// Return null if you can't find any activity for the user.
// HINT: Try looking at the way 'getAllFunctionCalls' is written. You can
modify it if you'd like.
/**
 * 获取用户最后活跃时间戳
 * @param {string} user - 用户名
 * @returns {number} - 最后活跃时间戳
 */
function getLastActive(user) {
    const all_timestamps = getCallData((call) => {
        if (call.from == user || call.args[0] == user) {
            return [call.timestamp];
        }
        return [];
    }, (call) => {
        // Return early as soon as you find this user.
        return call.from == user || call.args[0] == user;
    });
    return Math.max(all_timestamps);

}

```

```
// TODO: add an IOU ('I owe you') to the system
// The person you owe money is passed as 'creditor'
// The amount you owe them is passed as 'amount'
/**
 * 添加债务。
 * @param {string} creditor - 债权人的地址。
 * @param {number} amount - 债务金额。
 * @returns {void}
 */
function add_IOU(creditor, amount) {
    // 假设债务人是发起交易的人。
    const debtor = web3.eth.defaultAccount;
    // 如果债权人 -> 债务人之间存在路径（例如，债权人欠债务人债务），
    // 而不是立即添加债务，先找到路径并找到路径上的最小欠款金额。
    const path = doBFS(creditor, debtor, getCreditorsForUser);
    if (path != null) {
        const min_on_cycle = Math.min(findMinOnPath(path), amount);
        // 现在添加债务，让合约知道可能存在的循环。
        return BlockchainSplitwise.add_IOU(creditor, amount, path,
min_on_cycle);
    }
    // 没有循环，直接添加债务。
    var x = BlockchainSplitwise.add_IOU(creditor, amount, [],
/*min_on_cycle=*/0);
    return;
}
```

这些客户端函数是我智能合约的交互接口，它们允许用户通过Web界面执行各种操作：

1. **getCallData**: 此函数通过调用`getAllFunctionCalls`来获取特定合约函数的所有调用记录，并通过`extractor_fn`提取相关数据。它可以灵活地用于获取不同类型的数据，如债权人列表或用户列表。
2. **getCreditors**: 使用`getCallData`函数提取`add_IOU`调用中的债权人地址。它汇总并返回系统中所有债权人的集合。
3. **getCreditorsForUser**: 此函数为特定用户获取其债权人列表。通过调用`lookup`函数，它检查用户欠每位债权人的金额，并仅返回那些实际有债务的债权人。
4. **findMinOnPath**: 在特定路径上找到最小债务金额，这对于处理债务循环是关键。它通过遍历路径并使用`lookup`函数来确定最小债务额。
5. **getUsers**: 获取系统中所有用户（债务人和债权人）的列表。它通过分析所有`add_IOU`调用来识别参与过交易的所有用户。
6. **getTotalOwed**: 计算并返回特定用户所欠的总金额。它遍历所有债权人并累加用户欠每位债权人的债务。
7. **getLastActive**: 获取用户最后一次活动（发送或接收债务）的时间。它从所有相关的`add_IOU`调用中提取时间戳并找出最新的一个。
8. **add_IOU**: 允许用户添加新的债务。在添加之前，它检查是否存在债务循环，并在必要时调整债务额。

这些函数共同为用户提供了一个完整的界面，使他们能够与智能合约互动，查询债务信息，添加新债务，以及了解自己在系统中的活动情况。

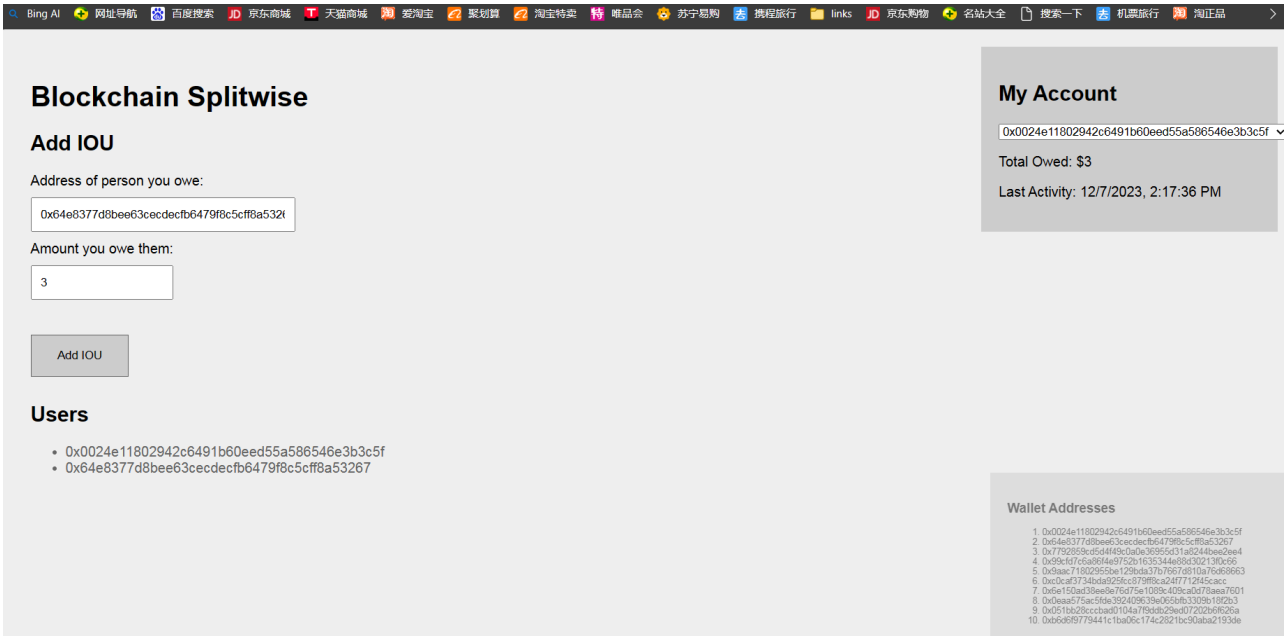
4 实验结果

进入HTML网页，对于以下三个地址：

```
1号：0x0024e11802942c6491b60eed55a586546e3b3c5f
2号：0x64e8377d8bee63cecdecfb6479f8c5cff8a53267
3号：0x7792859cd5d4f49c0a0e36955d31a8244bee2ee4
```

执行以下操作：

```
1号 欠 2号 $3
2号 欠 3号 $4
3号 欠 1号 $5
```



上述过程将会出现一个循环，消去最小权重后，理论上结果是：

```
2号 欠 3号 $1
3号 欠 1号 $2
```

结果如下图，可以看到和理论相同。