区块链第五次作业

2111460 张洋 2111617 尚然

实验要求

项目有两个主要组件:一个智能合约，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，它使用web3.js观察区块链，并且可以调用智能合约中的函数。

实验代码

智能合约

智能合约 BlockchainSplitwise允许用户之间管理债务，包含以下功能：

1. **定义用户结构**：每个用户对应一个 User 结构，其中包含一个映射，记录他们欠其他用户的债务。同时，记录了用户的最后操作时间戳。

```
struct User {
    mapping (address => uint32) debts; // 映射其他用户的地址到欠他们的债务
    uint timestamp; // 用户的最后操作时间戳
}
```

2. **查看债务**：通过 `lookup()` 函数，可以查看特定债务人针对特定债权人的债务。

```
// 一个映射，包含所有用户的地址和他们的User结构实例
mapping (address => User) public users;
// 返回债务人欠债权人的债务
function lookup(address debtor, address creditor) public view returns (uint32 ret)
{
    return users[debtor].debts[credi
```

3. **添加债务**：通过 `add_IOU()` 函数，可以添加债务人对债权人的债务。同时，该函数考虑了债务循环的存在，并会相应地做出调整。

```
function add_IOU(address creditor, uint32 amount, address[] memory cycle, uint32
minDebt) public {
    // 检查输入的金额和最低债务不能为负数
    require (amount >= 0, 'Negative amount');
    require (minDebt >= 0, 'Negative minDebt');

    if(cycle.length == 0){
        // 如果没有循环，直接增加债务
        users[msg.sender].debts[creditor] += amount;
    } else {
        // 检查发件人是否在循环的末尾
        for(uint i = 0; i < (cycle.length - 1); i++){
            // 确保循环中所有的债务都大于或等于最小债务
            require(lookup(cycle[i], cycle[i+1]) >= minDebt);
            // 减去从每个用户的债务中的最小债务
            users[cycle[i]].debts[cycle[i+1]] -= minDebt;
        }

        // 为最后一个用户增加债务，并减去最小债务
        users[cycle[cycle.length-1]].debts[cycle[0]] += amount;
        users[cycle[cycle.length-1]].debts[cycle[0]] -= minDebt;
    }

    // 更新债务人和债权人的最后操作时间
    update_timestamp(msg.sender);
    update_timestamp(creditor);
}
```

4. **更新和查看时间戳**：每次债务操作后，都会更新用户的最后操作时间戳。通过 `get_timestamp()` 函数可以查看用户的最后操作时间戳。

```
// 更新用户的最后操作时间
function update_timestamp(address user) public {
    users[user].timestamp = block.timestamp;
}

// 返回用户的最后操作时间
function get_timestamp(address user) public view returns (uint timestamp){
    return users[user].timestamp;
}
```

客户端

abi 和 contractAddress

```
var abi = [
    {
        "constant": false,
        "inputs": [
            {
                "internalType": "address",
                "name": "creditor",
                "type": "address"
            },
            {
                "internalType": "uint32",
                "name": "amount",
                "type": "uint32"
            },
            {
                "internalType": "address[]",
                "name": "cycle",
                "type": "address[]"
            },
            {
                "internalType": "uint32",
                "name": "minDebt",
                "type": "uint32"
            }
        ],
        "name": "add_IOU",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
                "internalType": "address",
                "name": "user",
                "type": "address"
            }
        ],
        "name": "update_timestamp",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [
            {
```

```
                "internalType": "address",
                "name": "user",
                "type": "address"
            }
        ],
        "name": "get_timestamp",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "timestamp",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [
            {
                "internalType": "address",
                "name": "debtor",
                "type": "address"
            },
            {
                "internalType": "address",
                "name": "creditor",
                "type": "address"
            }
        ],
        "name": "lookup",
        "outputs": [
            {
                "internalType": "uint32",
                "name": "ret",
                "type": "uint32"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [
            {
                "internalType": "address",
                "name": "",
                "type": "address"
            }
```

```
        ],
        "name": "users",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "timestamp",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }
]

var contractAddress = '0x7EBc6632750369FfdadeDd5e33442A670581b443'
```

功能代码

主要包含一系列 JavaScript 函数，这些函数与 Ethereum 上部署的 BlockchainSplitwise 智能合约交互，包含以下功能：

1. **查找邻居**：`findNeighbors()` 函数用于查找特定用户欠款的用户。

```
function findNeighbors(node) {
    var neighbors = [];
    var users = getUsers();
    for(var i = 0; i < users.length; i++) {
        if(BlockchainSplitwise.lookup(node,users[i]).toNumber() > 0){
            neighbors.push(users[i]); // If the neighbor is a creditor
        }
    }
    return neighbors;
}
```

2. **获取所有用户**：`getUsers()` 函数用于获取系统中所有的用户 (债权人或债务人)。

```
function getUsers() {
    var IOU_calls = getAllFunctionCalls(contractAddress, 'add_IOU');
    var users = [];

    for (var i = 0; i < IOU_calls.length; i++) {
        if (!users.includes(IOU_calls[i].from)) {
            users.push(IOU_calls[i].from);
        }

        if(!users.includes(IOU_calls[i].args[0])){
            users.push(IOU_calls[i].args[0]);
        }
    }
    return users;
}
```

3. **获取总债务**：`getTotalOwed()` 函数用于获取特定用户欠的总债务。

```
function getTotalOwed(user) {
    var owed_amount = 0;
    var neighbors = findNeighbors(user);
    for(var i = 0; i < neighbors.length; i++) {
        var debt = BlockchainSplitwise.lookup(user, neighbors[i]).toNumber();
        owed_amount += debt;
    }
    return owed_amount;
}
```

4. **获取最后活动时间**：`getLastActive()` 函数用于获取用户最后一次发送或接收 IOU 的时间。

```
function getLastActive(user) {
// Get all the add_IOU calls, and check the first including our user (either as
creditor or sender)
    var IOU_calls = getAllFunctionCalls(contractAddress, 'add_IOU');
    for (var i = 0; i < IOU_calls.length; i++) {
        if (IOU_calls[i].from === user || IOU_calls[i].args[0] == user){
            var ts = BlockchainSplitwise.get_timestamp(user);
            if(ts === 0){
                return null;
            }else{
                return ts;
            }
        }
    }
    return null;
}
```

5. **添加债务**：`add_IOU()` 函数用于向系统添加 IOU (债务)。这个函数会考虑到债务循环的存在，并相应地调整债务。

```
function add_IOU(creditor, amount) {
    var loop = doBFS(creditor, web3.eth.defaultAccount, findNeighbors);

    if(loop === null){
        var cycle = [];
        BlockchainSplitwise.add_IOU(creditor, amount, cycle, 0); // Problem:
running out of gas
    } else {
        // Select minimum
        var minDebt = amount; // Initialize minimum debt with paying amount
        for(var i = 0; i < (loop.length-1); i++){ // If length = 1?
            var debt = BlockchainSplitwise.lookup(loop[i], loop[i+1]).toNumber();
            if(debt < minDebt){
                minDebt = debt;
            }
        }
        BlockchainSplitwise.add_IOU(creditor, amount, loop, minDebt)
    }
}
```

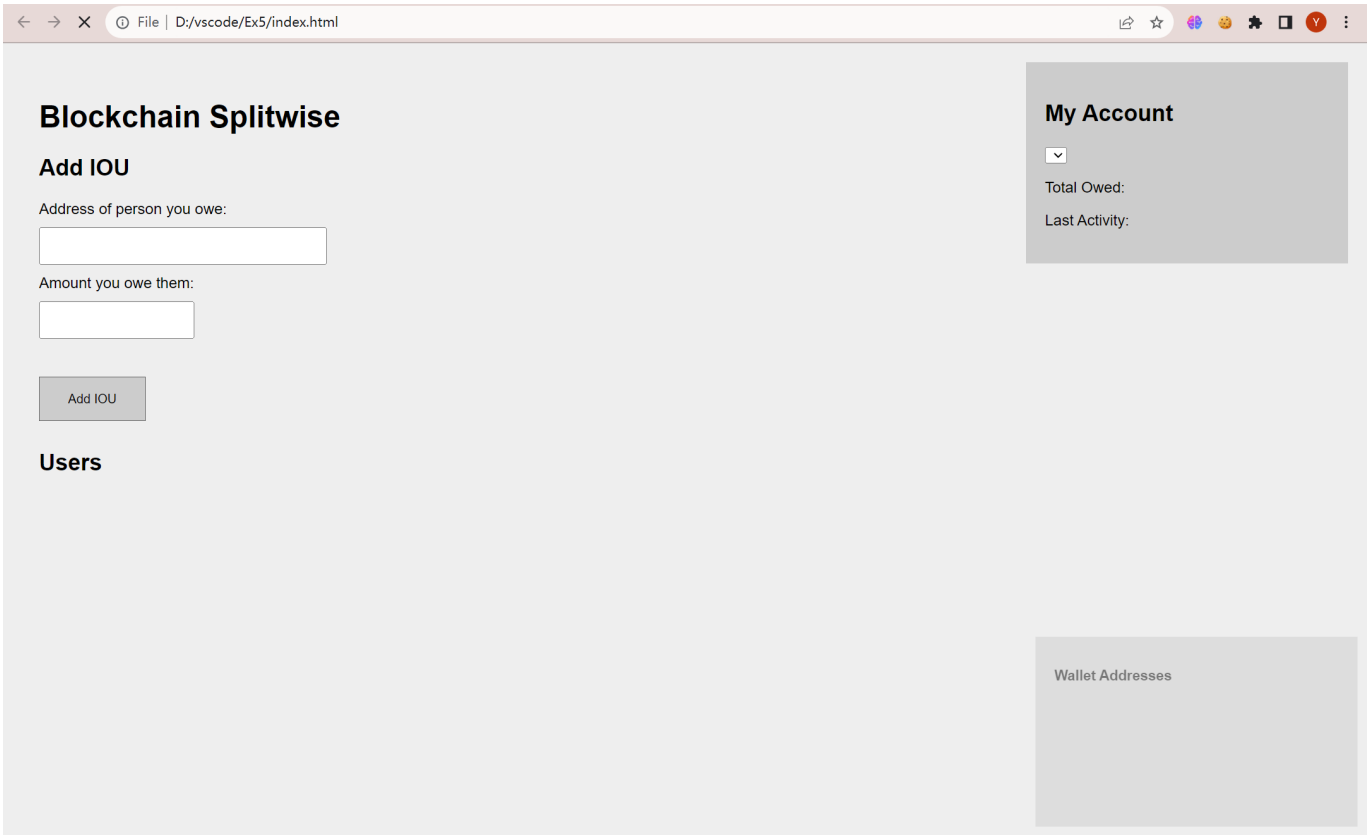
6. **获取所有函数调用**：`getAllFunctionCalls()` 函数用于搜索区块历史，寻找所有在特定合约上调用特定函数的情况。

```
function getAllFunctionCalls(addressOfContract, functionName, earlyStopFn) {
    var curBlock = web3.eth.blockNumber;
    var function_calls = [];
    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});
                  function_calls.push({
                      from: txn.from,
                      args: args,
                      timestamp: b.timestamp,
                  })
                  if (earlyStopFn &&
                          earlyStopFn(function_calls[function_calls.length-1])) {
                      return function_calls;
                  }
              }
          }
      }
      curBlock = b.parentHash;
    }
    return function_calls;
}
```

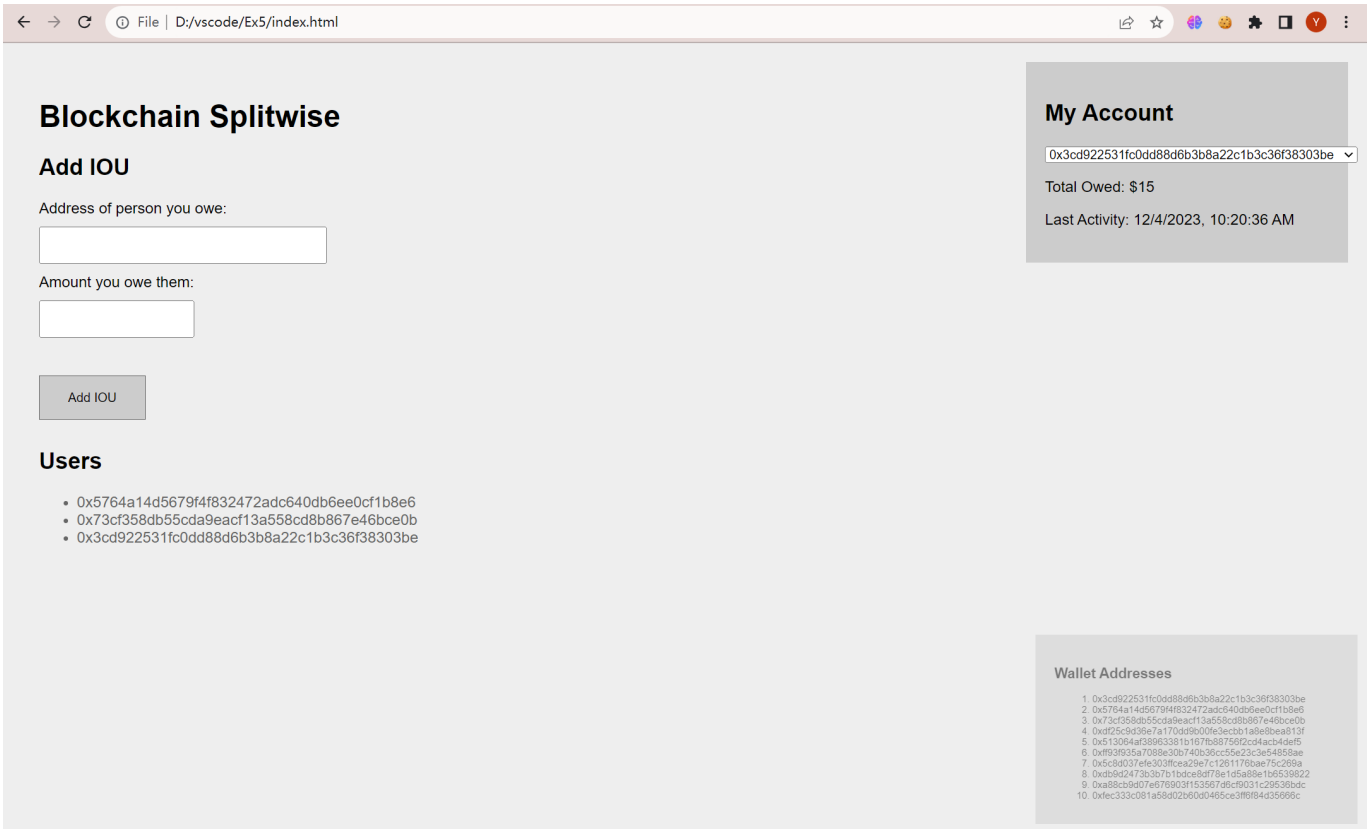
7. **广度优先搜索（BFS）**：`doBFS()` 函数提供了广度优先搜索的实现，用于找到从开始节点到结束节点的路径。

```
function doBFS(start, end, getNeighbors) {
    var queue = [[start]];
    while (queue.length > 0) {
        var cur = queue.shift();
        var lastNode = cur[cur.length-1]
        if (lastNode === end) {
            return cur;
        } else {
            var neighbors = getNeighbors(lastNode);
            for (var i = 0; i < neighbors.length; i++) {
                queue.push(cur.concat([neighbors[i]]));
            }
        }
    }
    return null;
}
```

运行截图



添加一部分信息后：



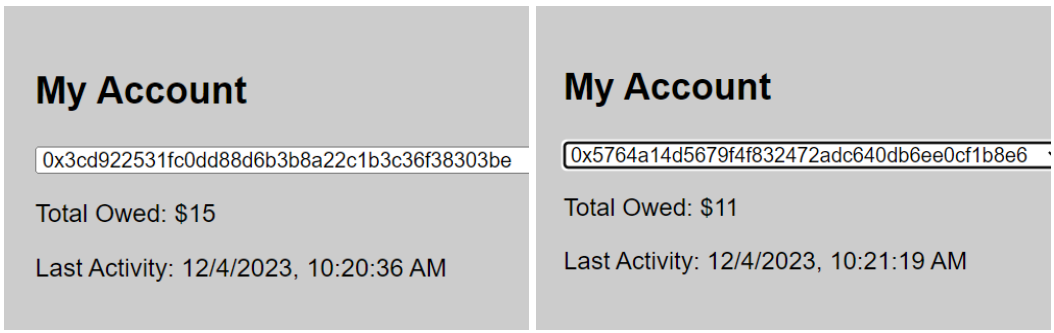
结果分析

地址如下图所示：

三个人的情况

例如:如果$A \overset{15}{\rightarrow} B$并且$B \overset{11}{\rightarrow} C$,当$C$加上$C \overset{16}{\rightarrow} A$,实际的余额将会更新为$A \overset{4}{\rightarrow} B, B \overset{0}{\rightarrow} C, C \overset{5}{\rightarrow} A$.相似的,如果$C \overset{9}{\rightarrow} A$,那么最后实际的余额将会更新为$A \overset{6}{\rightarrow} B, B \overset{2}{\rightarrow} C, C \overset{0}{\rightarrow} A$.

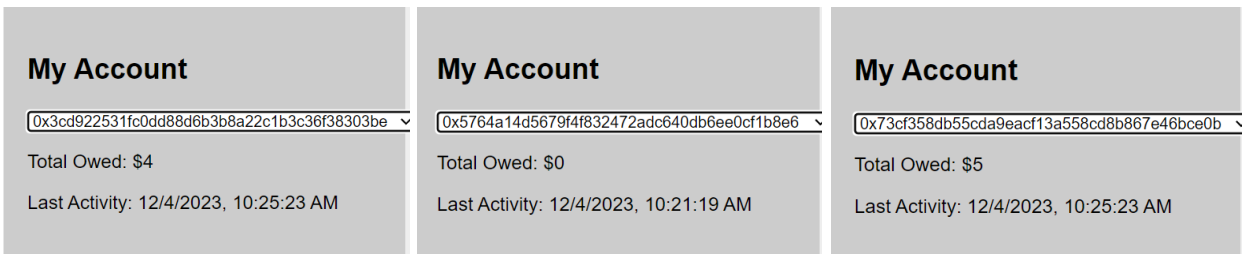
1. 选取地址1号、2号、3号三个地址进行示范，初始三人total owed皆为0。
2. 令1号欠2号`15`，2号欠3号`11`，如下图。



3. 令3号欠1号`16`。

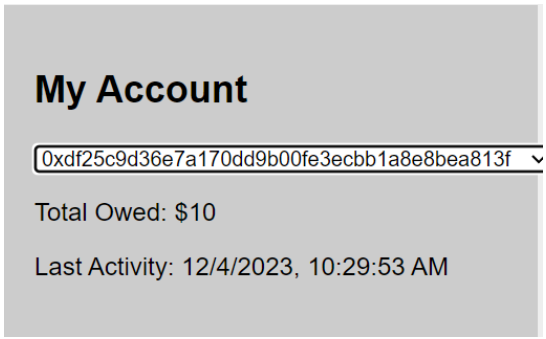
更新余额后的结果应为：1号欠2号`4`，2号欠3号`0`，3号欠1号`5`。

运行结果如下图，符合推算的结果。



两人的情况

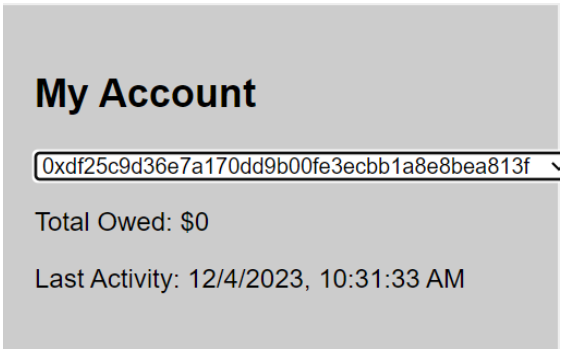
1. 选取地址4号、5号两个地址进行示范，初始两人total owed皆为0。
2. 设置4号欠5号`10`，如下图。

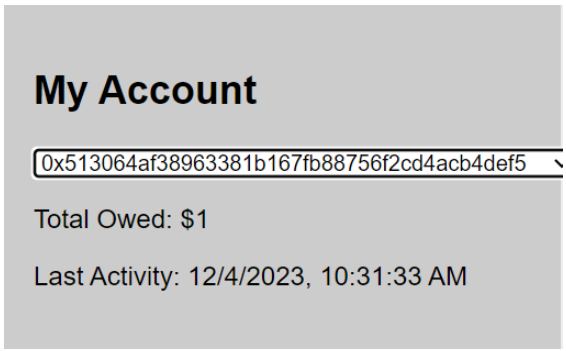


3. 设置5号欠4号 `11` 。

更新余额后的结果应为：4号欠5号 `0` ，5号欠4号 `1` 。

运行代码，查询结果如下图，符合。





总结

这个实验的主要目标是创建一个包括智能合约和Web客户端的完整的区块链项目。项目中的两个主要组件"智能合约"和"客户端"都是区块链技术的核心部分，因此对于理解这一技术非常重要。

在实验中，首先使用Solidity编写智能合约并在区块链上部署它。Solidity是一种专门为实现智能合约而设计的编程语言，它具有静态类型、支持继承和复杂的用户定义类型等特性，这使得编写复杂的智能合约成为可能。通过编写并部署智能合约，我们了解到了Solidity的基本语法和智能合约的工作原理，这对于理解区块链的核心概念非常有帮助。

然后，我们创建了一个Web客户端，它在浏览器中运行，并使用web3.js与区块链进行交互。通过这个过程，我们学习到了如何使用web3.js观察区块链的状态，以及如何调用智能合约的函数。这让我们明白了区块链的实际应用是如何运作的。

这个实验让我们更深入地理解了区块链技术，特别是智能合约和Web3.js的使用。