



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

区块链实验报告

Ex5 DAPP

徐俊智 2213410

符秀婷 2212939

年级：2022 级

专业：计算机科学与技术

指导教师：苏明

2024 年 12 月 5 日

目录

一、 实验目的	1
二、 前期准备	1
三、 智能合约	1
(一) 数据结构	1
(二) 函数	2
(三) 优化 gas 成本	3
四、 客户端	4
五、 运行结果	8

一、 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序(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。

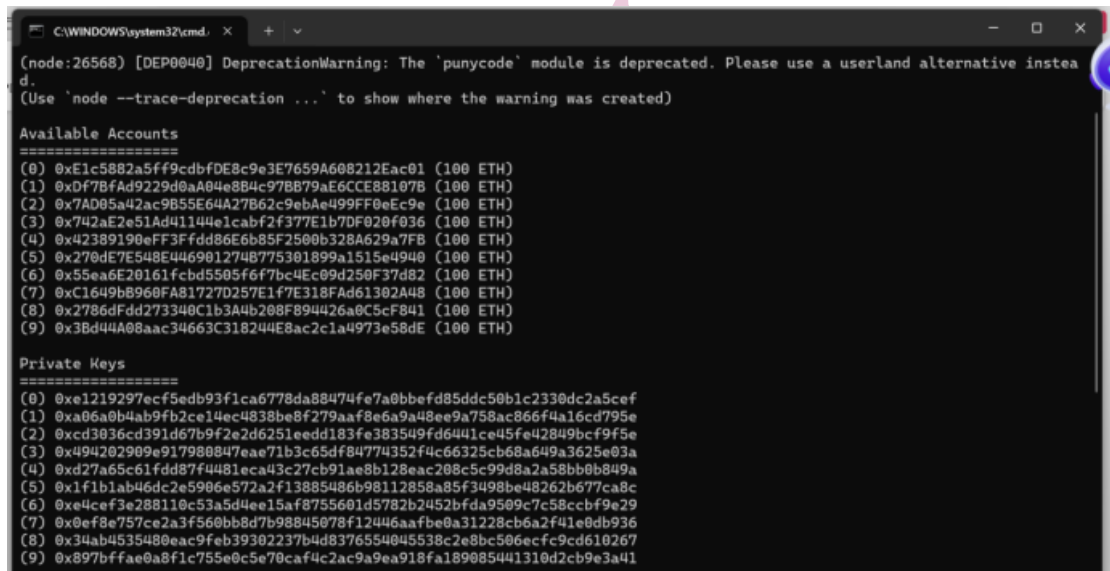
二、 前期准备

1. 安装 Node.js 的 LTS 版本 1<https://nodejs.org/en/>
2. 安装 Ganache CLI
在命令行中运行

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

运行节点

```
1 ganache-cli
```



```
(node:26568) [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) 0xE1c5882a5ff9c9dbfDE8c9e3E7659A608212Eac01 (100 ETH)
(1) 0xDf7BfAd9229d0aA04e8B4c97BB79aE6CCE88107B (100 ETH)
(2) 0x7AD05a42ac9B55E64A27B62c9ebAe499FF0eEc9e (100 ETH)
(3) 0x742aE251Ad41144e1cabf2f377E1b7DF020f036 (100 ETH)
(4) 0x42389190eFF3Fdd86E6b85F2500b328A629a7FB (100 ETH)
(5) 0x270dE7E548E446901274B775301899a1515e4940 (100 ETH)
(6) 0x55ea6E20161fcbd5505f6f7bc4Ec09d250F37d82 (100 ETH)
(7) 0xC1649bB960FA81727D257E1f7E318FAd61302A48 (100 ETH)
(8) 0x2786dFdd27340C1b3A4b708F894426a0C5cF841 (100 ETH)
(9) 0x3Bd44A08aac34663C318244E8ac2c1a4973e58dE (100 ETH)

Private Keys
=====
(0) 0xe1219297ecf5edb93f1ca6778da88474fe7a0bbefd85ddc50b1c2330dc2a5cef
(1) 0xa06a0b4ab9fb2ce14ec4838be8f279aaf8e6a9a48ee9a758ac866f4a16cd795e
(2) 0xcd3036cd391d67b9f2e2d6251eadd183fe383549fd6441ce45fe42849bcf9f5e
(3) 0x494202909e917980847eae71b3c65df84774352f4c66325cb68a649a3625e03a
(4) 0xd27a65c61fdd87f4481eca43c27cb91ae8b128eac208c5c99d8a2a58bb0b849a
(5) 0x1f1b1ab46dc2a5906e572a2f13885486b9811285a85f3498be48262b677ca8c
(6) 0xe4cef3e288110c53a5d4ee15af8755601d5782b2452bfda9509c7c58ccbf9e29
(7) 0x0ef8e757ce2a3f560bb8d7b98845078f12446aafbe0a31228cb6a2f41e8db936
(8) 0x34ab4535480eac9feb39302237b4d8376554045538c2e8bc506ecfc9cd610267
(9) 0x897bffae0a8f1c755e0c5e70caf4c2ac9a9ea918fa189085441310d2cb9e3a41
```

三、 智能合约

(一) 数据结构

```
1 struct Debt {
2     uint32 amount;
3 }
```

定义了一个名为 Debt 的结构体, 用于记录债务金额, 金额类型为 uint32, 其最大可表示的债务约为 2^{32} (约 40 亿), 并且合约中会进行溢出检查。

```
1 mapping(address => mapping(address => Debt)) internal all_debts;
```

创建了一个嵌套的映射, 用于跟踪债务关系。外层键是债务人的地址, 内层键是债权人的地址, 值是对应的 Debt 结构体, 通过这种方式可以记录每个债务人对每个债权人的债务情况。

为了确保合约的安全性, 使用 `internal` 关键字确保映射中的数据只能在合约内部进行修改和查询。

(二) 函数

```
1 function lookup(address debtor, address creditor) public view returns (uint32
    ret) {
2     ret = all_debts[debtor][creditor].amount;
3 }
```

查询债务人欠债权人的总金额。

`public` 关键字表示可以从合约外部调用, 而 `view` 关键字声明函数是视图函数, 该函数仅作查询使用, 不会修改合约的状态。

```
1 function add_IOU(address creditor, uint32 amount, address[] memory path,
    uint32 \texttt{\detokenize{min_on_cycle}}) public {
2     address debtor = msg.sender;
3     require(debtor != creditor);
4     require(amount > 0);
5     Debt storage iou = all_debts[debtor][creditor];
6     if (\texttt{\detokenize{min_on_cycle}} == 0) {
7         iou.amount = add(iou.amount, amount);
8         return;
9     }
10    require(\texttt{\detokenize{min_on_cycle}} <= (iou.amount + amount));
11    require(verify_and_fix_path(creditor, debtor, path, \texttt{\detokenize{
        min_on_cycle}}));
12    iou.amount = add(iou.amount, (amount - \texttt{\detokenize{min_on_cycle
        }}}));
13 }
```

为调用者添加一个债务。

- 要求债务人不能是债权人自己, 金额必须为正数。
- 获取当前债务人对债权人的债务记录 (Debt 结构体实例)。
- 根据 `min_on_cycle` (路径上的最小债务金额) 的值分情况处理:
 - 如果 `min_on_cycle` 为 0, 直接增加债务金额, 调用 `add` 函数进行带溢出检查的加法操作。
 - 如果 `min_on_cycle` 不为 0, 需要验证它小于等于当前债务金额与新增金额之和, 并且通过 `verify_and_fix_path` 函数验证并处理债务循环相关的路径情况, 最后更新债务金额, 更新方式是先将当前债务金额加上 (新增金额减去 `min_on_cycle` 的值, 也就是按照处理债务循环的逻辑来调整最终的债务记录。

```
1 function verify_and_fix_path(address start, address end, address[] memory
    path, uint32 \texttt{\detokenize{min_on_cycle}}) private returns (bool
    ret) {
```

```

2   if (start!= path[0] || end!= path[path.length - 1]) {
3       return false;
4   }
5   if (path.length > 12) {
6       return false;
7   }
8   for (uint i = 1; i < path.length; i++) {
9       Debt storage iou = all_debts[path[i - 1]][path[i]];
10      if (iou.amount == 0 || iou.amount < \texttt{\detokenize{min_on_cycle}}) {
11          return false;
12      } else {
13          iou.amount -= \texttt{\detokenize{min_on_cycle}};
14      }
15  }
16  return true;
17 }

```

处理债务循环相关的验证，调整路径上债务金额。

- 检查传入的路径的起始地址是否与给定的 `start` 参数一致，以及路径的结束地址是否与给定的 `end` 参数一致
- 设定了路径长度不能超过 12 的限制，这是为了避免路径过长可能导致的高 Gas 消耗（以太坊中执行函数调用消耗的资源费用，路径越长意味着需要进行的操作和数据读取越多，Gas 消耗越大）。
- 循环遍历路径中的每一段债务关系，如果满足条件，减去 `min_on_cycle`，否则返回 `false`，表示验证失败。

```

1 function add(uint32 a, uint32 b) internal pure returns (uint32) {
2     uint32 c = a + b;
3     require(c >= a);
4     return c;
5 }

```

带溢出检查的加法函数。

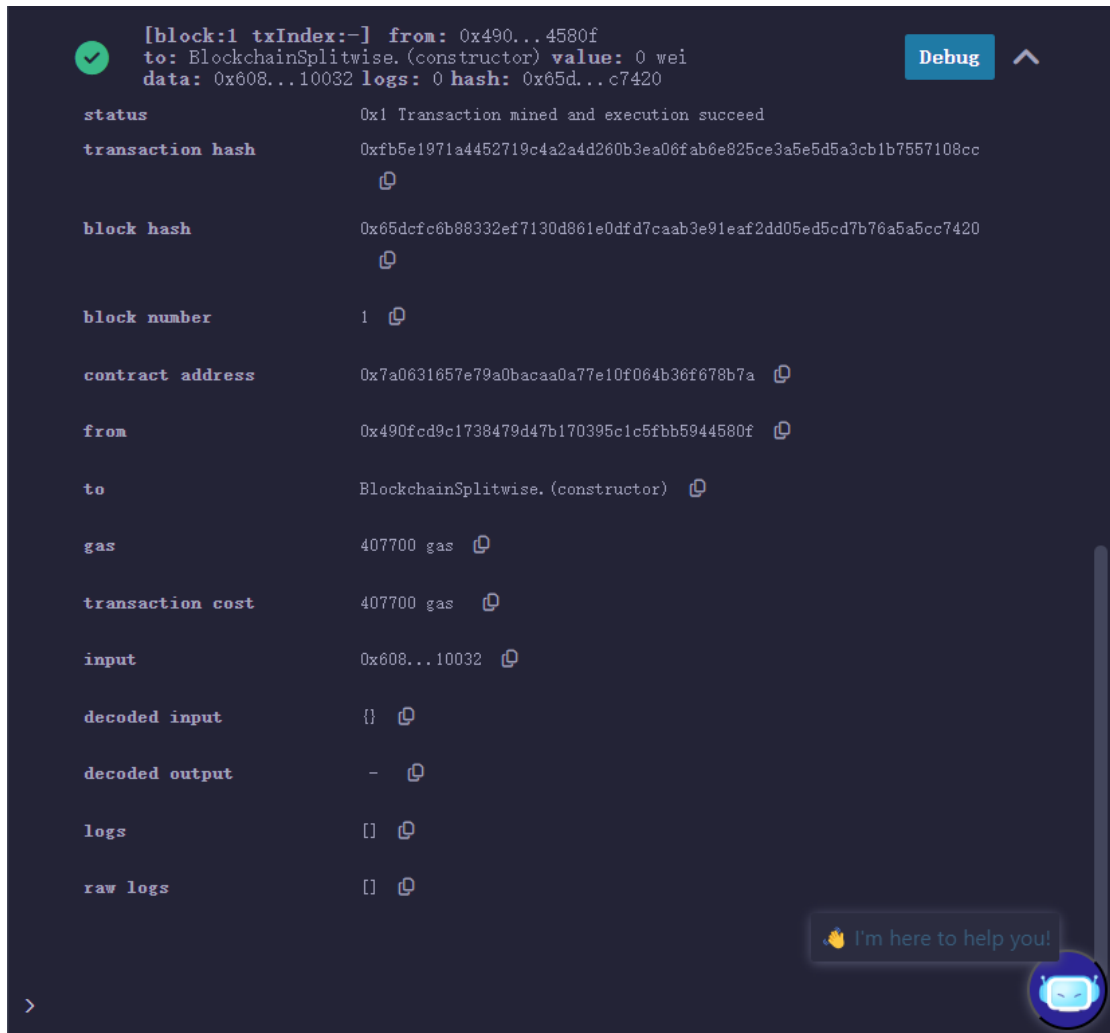
(三) 优化 gas 成本

为了优化 gas 成本，我们进行了如下工作：

1. 使用较小的数据类型（如 `uint32` 类型）来定义债务金额
2. 使用 `view` 关键字声明 `lookup` 函数是视图函数，即函数不修改合约的状态变量，但可以读取合约的状态。
3. 使用 `require` 函数来验证运行条件，在条件不满足时会回滚交易，避免了无效操作的继续执行，避免 gas 浪费。

4. 对传入的路径长度进行检查, 限制路径长度不超过 12, 防止因过长路径导致大量循环操作消耗过多 gas。

可以看到, 汽油费 gas 是一个较为合理的值



四、 客户端

```

1 function getCallData(extractor_fn, early_stop_fn) {
2     const results = new Set();
3     const all_calls = getAllFunctionCalls(contractAddress, 'add_IOU',
4         early_stop_fn);
5     for (var i = 0; i < all_calls.length; i++) {
6         const extracted_values = extractor_fn(all_calls[i]);
7         for (var j = 0; j < extracted_values.length; j++) {
8             results.add(extracted_values[j]);
9         }
10    }
11    return Array.from(results);
12 }

```

从合约的函数调用历史中提取数据。

- 首先创建了一个 Set 类型的变量 results
- 通过调用 getAllFunctionCalls 函数（后面会详细讲解这个函数）来获取所有对 add_IOU 函数的调用记录
- 遍历获取到的所有 add_IOU 函数调用记录，对于每一条记录，调用传入的 extractor_fn 函数来提取数据。再进一步遍历这些提取的值，将它们逐个添加到 results 集合中，由于 Set 的特性，重复的值会自动被去除。
- 最后通过 Array.from(results) 将 Set 转换回数组形式并返回。

```
1 function getCreditors() {  
2     return getCallData((call) => {  
3         return [call.args[0]];  
4     }, null);  
5 }
```

获取系统中所有的债权人地址列表。通过调用 getCallData 函数，传入提取函数 (call) => return [call.args[0]];，从 getAllFunctionCalls 获取到的每一条 add_IOU 函数调用记录（用 call 表示）中，提取出第一个参数（也就是债权人地址），并将其包装成一个数组返回。同时，传入 null 作为 early_stop_fn，表示不需要提前停止获取函数调用记录的过程，会完整遍历所有相关记录来提取债权人地址。

```
1 function getCreditorsForUser(user) {  
2     var creditors = [];  
3     const all_creditors = getCreditors();  
4     for (var i = 0; i < all_creditors.length; i++) {  
5         const amountOwed = BlockchainSplitwise.lookup(user, all_creditors[i])  
6             .toNumber();  
7         if (amountOwed > 0) {  
8             creditors.push(all_creditors[i]);  
9         }  
10    }  
11    return creditors;  
}
```

获取指定用户的所有债权人地址列表。

- 首先创建一个空数组 creditors，用于存储最终筛选出来的债权人地址。
- 调用 getCreditors 函数获取到所有的债权人地址列表（不管是否与指定用户有债务关系），并将其存储在 all_creditors。
- 遍历所有的债权人地址，对于每一个债权人地址，调用合约实例的 lookup 函数来查询指定用户对该债权人的债务金额，并通过 toNumber 方法将返回的可能是以太坊特定格式的数值转换为普通的 JavaScript 数字类型。如果查询到的债务金额大于 0，就将这个债权人的地址添加到 creditors 数组中，最后将其返回。

```
1 function findMinOnPath(path) {
2   var minOwed = null;
3   for (var i = 1; i < path.length; i++) {
4     const debtor = path[i - 1];
5     const creditor = path[i];
6     const amountOwed = BlockchainSplitwise.lookup(debtor, creditor).
7       toNumber();
8     if (minOwed === null || minOwed > amountOwed) {
9       minOwed = amountOwed;
10    }
11  }
12  return minOwed;
13 }
```

查找给定的债务路径上的最小债务金额。

- 首先初始化 minOwed 用于记录最小债务金额。
- 循环遍历债务路径中的每一段相邻的债务关系（从索引 1 开始，因为路径中第一个元素是起始点，比较是基于相邻的债务关系，所以从第二个元素开始），对于每一段关系，获取债务人地址（path[i - 1]）和债权人地址（path[i]），并通过调用合约实例的 lookup 函数来查询这两者之间的债务金额，同样将其转换为普通数字类型。每次查询到债务金额后，尝试更新 minOwed，最后将其返回。

```
1 function getUsers() {
2   return getCallData((call) => {
3     return [call.from, call.args[0]];
4   }, null);
5 }
```

获取系统中所有债务人和债权人的地址。通过调用 getCallData 函数，传入提取函数 (call) => return [call.from, call.args[0]];，从 getAllFunctionCalls 获取到的每一条 add_IOU 函数调用记录（用 call 表示）中，提取出函数调用的发送者地址（也就是债务人地址，通过 call.from 获取）以及债权人地址（通过 call.args[0] 获取），并将其包装成一个数组返回。同时，传入 null 作为 early_stop_fn，表示不需要提前停止获取函数调用记录的过程，会完整遍历所有相关记录来提取债权人地址。

```
1 function getTotalOwed(user) {
2   var totalOwed = 0;
3   const all_creditors = getCreditors();
4   for (var i = 0; i < all_creditors.length; i++) {
5     totalOwed += BlockchainSplitwise.lookup(user, all_creditors[i]).
6       toNumber();
7   }
8   return totalOwed;
9 }
```

计算指定用户总共欠的债务金额。

- 首先初始化 totalOwed 用于累加用户的债务金额。

- 调用 `getCreditors` 函数获取所有债权人的地址列表, 存储在 `all_creditors`。
- 遍历所有债权人地址, 对于每一个债权人, 调用合约实例的 `lookup` 函数来查询指定用户对该债权人的债务金额, 并转换为数字类型后累加到 `totalOwed`, 最后将其返回。

```

1 function getLastActive(user) {
2   const all_timestamps = getCallData((call) => {
3     if (call.from === user || call.args[0] === user) {
4       return [call.timestamp];
5     }
6     return [];
7   }, (call) => {
8     return call.from === user || call.args[0] === user;
9   });
10  return Math.max(all_timestamps);
11 }

```

获取用户最后一次发送或接收 IOU 的时间戳

- 调用 `getCallData` 函数来提取符合条件的时间戳信息, 传入的提取函数 `(call) => if (call.from === user || call.args[0] === user) return [call.timestamp]; return [];` 会判断每一条 `add_IOU` 函数调用记录, 如果调用的发送者地址 (`call.from`) 或者函数调用中的债权人地址 (`call.args[0]`) 是指定的用户, 就返回该调用记录对应的时间戳 (包装成数组形式), 否则返回空数组。同时, 传入的 `early_stop_fn` 函数 `(call) => return call.from === user || call.args[0] === user;` 用于控制当找到与指定用户相关的记录时就可以提前停止继续查找, 提高效率。
- 通过 `getCallData` 函数获取到所有符合条件的时间戳数组后, 使用 `Math.max` 函数从中找出最大的时间戳值, 这个最大值就代表了该用户最后一次参与 IOU 操作 (发送或接收) 的时间, 最后将其返回。

```

1 function add_IOU(creditor, amount) {
2   const debtor = web3.eth.defaultAccount;
3   const path = doBFS(creditor, debtor, getCreditorsForUser);
4   if (path !== null) {
5     const min_on_cycle = Math.min(findMinOnPath(path), amount);
6     return BlockchainSplitwise.add_IOU(creditor, amount, path,
7       min_on_cycle);
8   }
9   var x = BlockchainSplitwise.add_IOU(creditor, amount, [], 0);
10  return x;
11 }

```

用户调用此函数来添加一个新的债务记录。它会检查是否存在债务循环, 并处理路径上的债务。

- 首先通过 `web3.eth.defaultAccount` 确定债务人地址。
- 通过调用 `doBFS` 函数查找从债权人到债务人的债务路径。

- 如果找到了一条有效的债务路径,通过 `const min_on_cycle = Math.min(findMinOnPath(path), amount);` 计算 `min_on_cycle` 的值,通过 `return BlockchainSplitwise.add_IOU(creditor, amount, path, min_on_cycle);` 调用合约实例 (BlockchainSplitwise) 的 `add_IOU` 函数 (和合约里定义的同名函数对应), 将债权人地址、要添加的债务金额、找到的债务路径以及计算出的 `min_on_cycle` 值作为参数传递进去, 从而在合约中按照处理债务循环的逻辑添加这笔债务记录。
- 如果没有找到有效的债务路径,则通过 `var x = BlockchainSplitwise.add_IOU(creditor, amount, [], 0);` 直接调用合约实例的 `add_IOU` 函数添加债务。

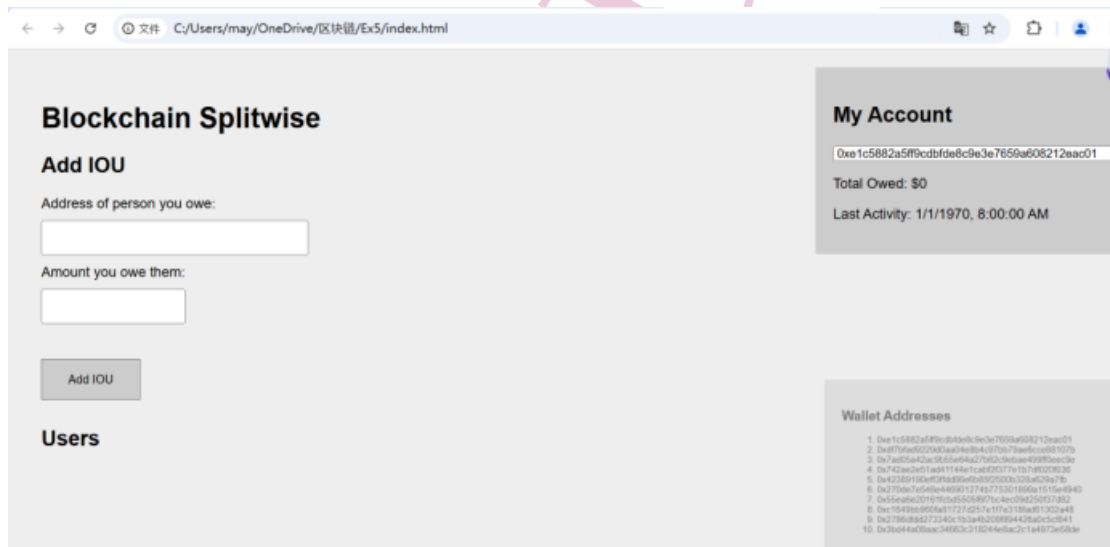
辅助函数:

`getAllFunctionCalls`: 搜索区块链历史中对指定合约 (由 `addressOfContract` 参数指定合约地址) 的特定函数 (由 `functionName` 参数指定函数名称) 的所有调用记录。

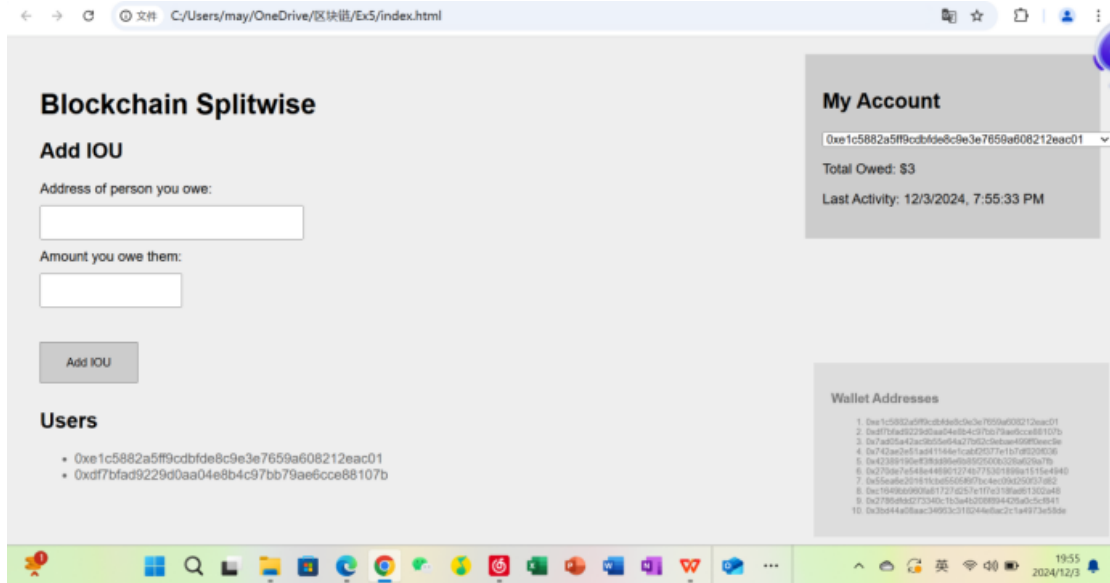
`doBFS`: 广度优先搜索。

五、 运行结果

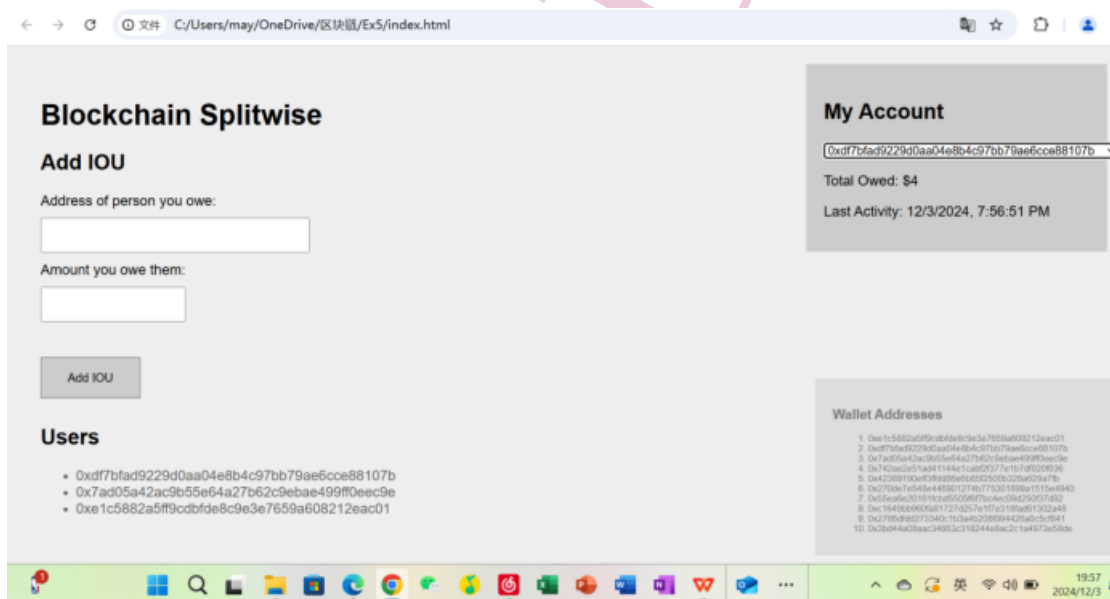
部署智能合约后, 将 `abi` 和 `contractAddress` 粘贴到 `script.js` 的相应位置上, 打开 `index.html`, 可以看到如下页面。



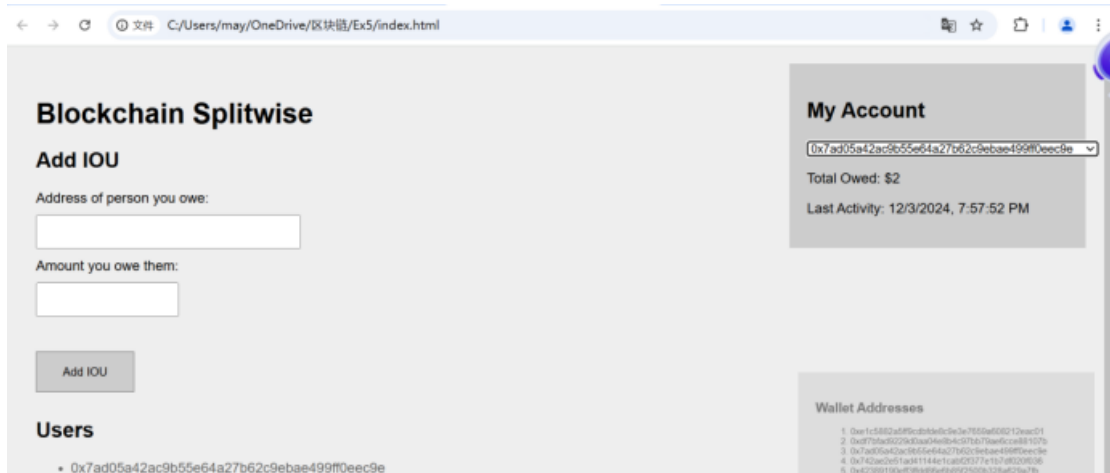
选择用户 1, 输入用户 2 的地址, 向用户 2 添加欠款 \$3, 点击 Add IOU, 可以看到, Users 列表中多了这两个用户的地址, 用户 1 的欠款数变为 \$3, 用户 1 与用户 2 的最近调用时间变为 12/3/2024, 7:55:33 PM。



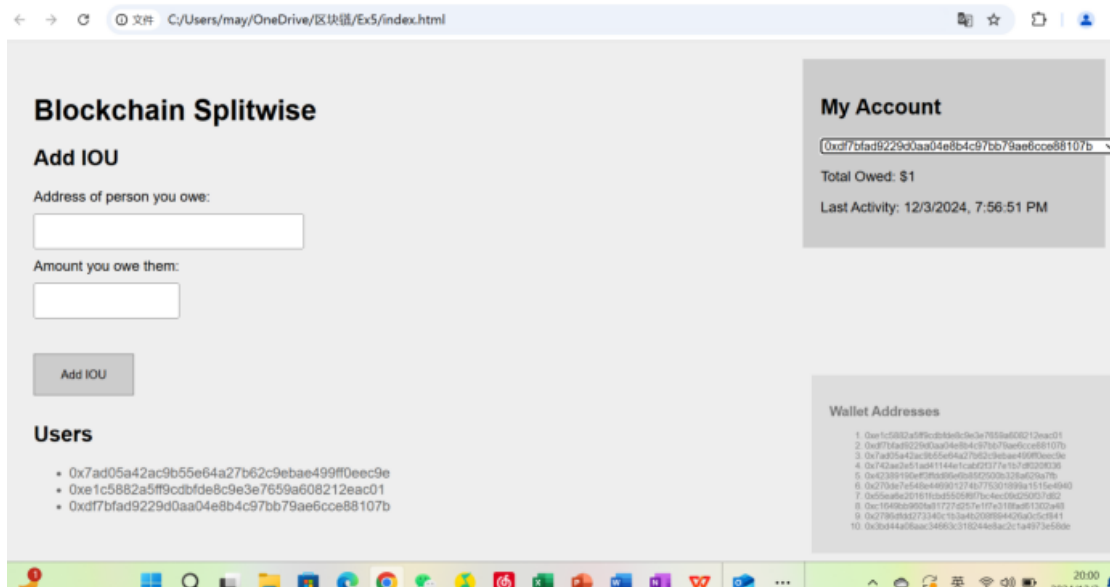
选择用户 2，输入用户 3 的地址，向用户 3 添加欠款 \$4，点击 Add IOU，可以看到，Users 列表中多了用户 3 的地址，用户 2 的欠款数变为 \$4，用户 2 与用户 3 的最近调用时间变为 12/3/2024, 7:56:51 PM。



选择用户 3，输入用户 1 的地址，向用户 1 添加欠款 \$5，点击 Add IOU，可以看到，用户 3 的欠款数变为 \$1，用户 3 与用户 1 的最近调用时间变为 12/3/2024, 7:57:52 PM。



用户 2 的欠款数变为 \$1, 用户 2 与用户 3 的最近调用时间变为 12/3/2024, 7:57:52 PM。



上述过程说明, 债务循环问题的得到解决。

