



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

区块链实验报告

---

## 实验 5: 去中心化应用程序 DAPP

---

姓名：徐海滢、文雅竹

年级：2022 级

专业：计算机科学与技术

指导教师：苏明

2024 年 12 月 8 日

# 目录

<b>一、 实验目的</b>	<b>1</b>
<b>二、 实验内容</b>	<b>1</b>
<b>三、 实验过程</b>	<b>1</b>
(一) 实验环境配置 . . . . .	1
1. Nodejs 安装 . . . . .	1
2. Ganache CLI 安装 . . . . .	1
3. remix 使用 . . . . .	2
4. 客户端部署 . . . . .	3
(二) 智能合约 . . . . .	4
1. 债务记录 . . . . .	4
2. 添加债务 (add_IOU) . . . . .	4
3. 查找债务 (lookup) . . . . .	5
(三) 循环债务解决 . . . . .	5
1. 解决循环债务 (resolve_cycle) . . . . .	5
2. 查找循环债务 (find_cycle) . . . . .	5
3. 解决债务路径 (resolve_path) . . . . .	6
(四) 客户端脚本 . . . . .	7
1. 智能合约交互函数 . . . . .	7
2. 用户界面更新函数 . . . . .	11
3. 用户界面交互 . . . . .	12
<b>四、 实验结果</b>	<b>14</b>
<b>五、 实验心得</b>	<b>16</b>
(一) 遇到的问题 . . . . .	16
(二) 合约减小开销的措施 . . . . .	16

## 一、 实验目的

本次实验的目的是通过在以太坊（Ethereum）上实现一个去中心化的应用程序（DApp），来学习和掌握区块链基础及其应用。实验将涉及使用 Solidity 编写智能合约以及使用 web3.js 创建用户客户端，从而深入理解 DApp 的“全栈”开发流程。

## 二、 实验内容

1. 创建一个去中心化的应用来跟踪借贷关系，即 Splitwise 的区块链版本。
2. 编写智能合约，实现债务的记录和查询功能。
3. 开发用户客户端，通过 web3.js 与智能合约交互。
4. 实现债务循环的检测和解决机制。

## 三、 实验过程

### （一） 实验环境配置

#### 1. Nodejs 安装

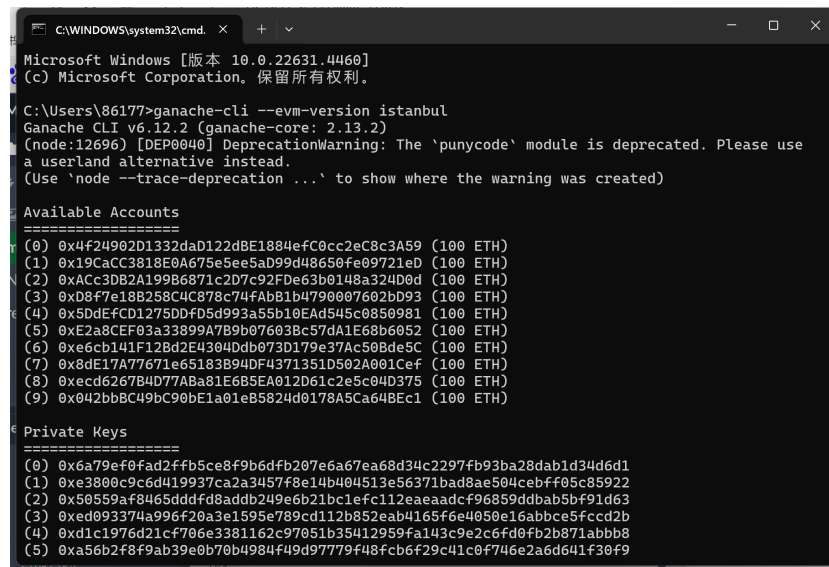


图 1: node.js

#### 2. Ganache CLI 安装

1. 在命令行中运行 `npm install -g ganache-cli` 来安装 Ganache CLI
2. `ganache-cli -evm-version istanbul`

运行后可以看见如下输出：



```
C:\WINDOWS\system32\cmd. x + v
Microsoft Windows [版本 10.0.22631.4460]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\86177>ganache-cli --evm-version istanbul
Ganache CLI v6.12.2 (ganache-core: 2.13.2)
(node:12696) [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) 0x4f24902D1332daD122dBE1884efC0cc2eC8c3A59 (100 ETH)
(1) 0x19CaCC3818E0A675e5ee5aD99d48650fe09721eD (100 ETH)
(2) 0xAcc3DB2A199B6871c2D7c92FDe63b0148a324D0d (100 ETH)
(3) 0xD8f7e188258C4C978c74fAb81bW790007602bd93 (100 ETH)
(4) 0x5DdEfCD1275DDfD5d993a55b10EAd545c0850981 (100 ETH)
(5) 0xE2a8CEf03a33890A7B0b07603Bc57dA1E68b6052 (100 ETH)
(6) 0xe6cb141F12Bd2E4304Ddb073D179e37Ac50Bde5C (100 ETH)
(7) 0x8dE17A77671e65183B94DF4371351D502A001Cef (100 ETH)
(8) 0xecd6267B4D77ABa81E6B5EA012D61c2e5c04D375 (100 ETH)
(9) 0x042bbBC49bC90bE1a01eB5824d0178A5Ca64BEc1 (100 ETH)

Private Keys
=====
(0) 0x6a79ef0fad2ffb5ce8f9b6dfb207e6a67ea68d34c2297fb93ba28dab1d34d6d1
(1) 0xe3800c9c6d419937ca2a3457f8e14b404513e56371bad8ae504cebff05c85922
(2) 0x50559af8465dddf8addb249e6b21bc1efc112eaaadc9f6859ddb5b5f91d63
(3) 0xed093374a996f20a3e1595e789cd112b852eab4165f6e4050e16abbce5fcd2b
(4) 0xd1c1976d21cf706e3381162c97051b35412959fa143c9e2c6fd0fb2b871abbb8
(5) 0xa56b2f8f9ab39e0b70b4984f49d97779f48fcb6f29c41c0f746e2a6d641f30f9
```

图 2: ganache 运行

### 3. remix 使用

- 打开网站 <https://remix.ethereum.org>, 按照 Ex5.pdf 文件中的教程部署
- 编译并部署 Solidity 代码

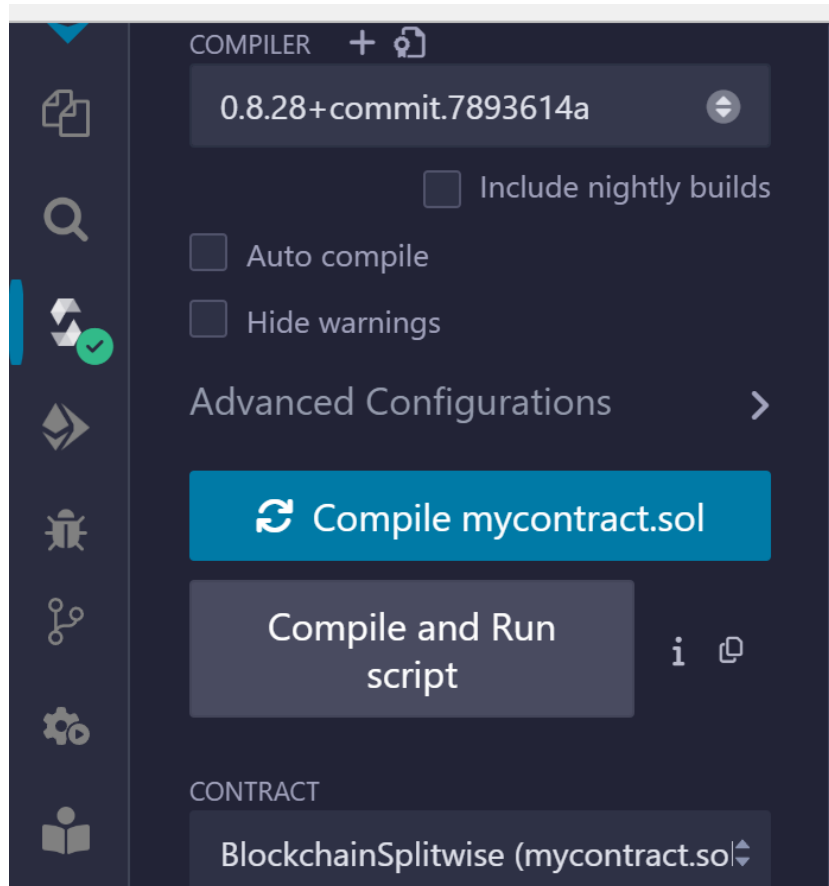


图 3: 编译成功



图 4: 部署成功

- 获取 ABI 和地址并复制到 script.js 文件的相应位置。

#### 4. 客户端部署

客户端部署部分主要涉及与智能合约的交互，通过使用 web3.js 库在前端与以太坊网络进行通信。以下是客户端部署的步骤和相关代码实现。

1. 安装依赖库在开始部署客户端之前，首先需要安装 web3.js 库，它是一个与以太坊区块链进行交互的 JavaScript 库。在项目目录下运行以下命令安装 web3.js:

```
1 npm install web3
```

2. 配置 Web3 实例在客户端代码中,使用 web3.js 库创建一个 Web3 实例,以便与 Ganache 区块链或真实的以太坊网络进行交互。以下是配置 Web3 实例的代码:

```

1  const Web3 = require('web3');
2  // 使用 Ganache 本地节点的 HTTP RPC URL
3  const web3 = new Web3('http://127.0.0.1:8545');

```

在这里, `http://127.0.0.1:8545` 是 Ganache CLI 默认提供的本地节点地址。如果使用的是其他网络 (如测试网或主网), 则需要更改为对应的 RPC 地址。

3. 启动客户端完成以上配置后, 通过运行以下命令启动客户端:

```

1  python -m http.server 8000

```

启动后, 客户端将通过 `web3.js` 库与部署的智能合约进行交互, 实现前端功能, 如添加欠条、查看债务记录等。访问本地 8000 端口就可以运行网站。

## (二) 智能合约

该智能合约实现了一个基于区块链的债务记录和循环债务解决系统。合约中包含债务追踪、用户管理和循环债务解决的功能。它通过 `lookup_table`

合约中的主要功能包括债务记录、用户管理、债务查询和循环债务解决。每个功能的代码实现和详细分析如下:

### 1. 债务记录

```

1 mapping(address => mapping(address => uint32)) public lookup_table; // 债务
   记录表

```

`lookup_table` 是一个双重映射, 用于记录债务人和债权人之间的债务金额。键值对中的第一个地址表示债务人, 第二个地址表示债权人, `uint32` 类型的值表示债务金额。

### 2. 添加债务 (add\_IOU)

```

1 function add_IOU(address creditor, uint32 amount) public {
2     require(amount > 0, "Amount_must_be_positive");
3     require(creditor != msg.sender, "Debtor_and_creditor_cannot_be_the_same");
4     ;
5     require(creditor != address(0), "Invalid_creditor_address");
6
7     uint32 current_debt = lookup_table[msg.sender][creditor];
8     require(current_debt + amount >= current_debt, "Overflow_check_failed");
9     lookup_table[msg.sender][creditor] = current_debt + amount;
10
11     if (!hasUser(msg.sender)) {
12         users.push(msg.sender);
13     }
14     if (!hasUser(creditor)) {
15         users.push(creditor);
16     }
17
18     emit IOUAdded(msg.sender, creditor, amount);

```

```

18
19     resolve_cycle(msg.sender, creditor);
20 }

```

add\_IOU 函数用于添加新的债务记录。函数首先进行基本检查，确保债务金额大于零，债务人和债权人不同，并且债权人地址有效。然后，更新债务记录，增加新的债务金额，并检查溢出。接着，判断债务人和债权人是否已经在用户列表中，如果没有，则将其加入用户列表。最后，触发 IOUAdded 事件并尝试解决循环债务。

### 3. 查找债务 (lookup)

```

1 function lookup(address debtor, address creditor) public view returns (uint32
   ) {
2     return lookup_table[debtor][creditor];
3 }

```

‘lookup’ 函数用于查询某个债务人对某个债权人的债务金额。它接受两个参数：债务人地址和债权人地址，返回债务金额。

## (三) 循环债务解决

合约还实现了一个重要的功能：解决循环债务。循环债务指的是多个用户之间相互欠款，形成债务循环。为了优化结算，合约提供了 resolve\_cycle 和 find\_cycle 函数来查找和解决这些债务循环。

### 1. 解决循环债务 (resolve\_cycle)

```

1 function resolve_cycle(address debtor, address creditor) internal {
2     bool[] memory visited = new bool[](users.length);
3     address[] memory path = new address[](users.length);
4
5     for (uint i = 0; i < users.length; i++) {
6         visited[i] = false;
7     }
8
9     find_cycle(debtor, creditor, visited, path, 0);
10 }

```

resolve\_cycle 函数是循环债务解决的入口。它初始化访问记录和路径记录，然后调用 find\_cycle 函数查找循环债务。

### 2. 查找循环债务 (find\_cycle)

```

1 function find_cycle(
2     address start,
3     address current,
4     bool[] memory visited,
5     address[] memory path,
6     uint path_length

```

```

7 ) internal {
8     uint current_index = get_user_index(current);
9
10    if (visited[current_index]) {
11        return;
12    }
13
14    visited[current_index] = true;
15    path[path_length] = current;
16    path_length++;
17
18    for (uint i = 0; i < users.length; i++) {
19        address next = users[i];
20        if (lookup_table[current][next] > 0) {
21            if (next == start && path_length > 2) {
22                resolve_path(path, path_length);
23            } else {
24                find_cycle(start, next, visited, path, path_length);
25            }
26        }
27    }
28
29    visited[current_index] = false;
30 }

```

find\_cycle 函数通过深度优先搜索算法查找债务循环。它从一个起始用户出发，递归地遍历所有用户，查找是否存在循环。如果找到循环，调用 resolve\_path 函数解决循环债务。

### 3. 解决债务路径 (resolve\_path)

```

1 function resolve_path(address[] memory path, uint path_length) internal {
2     uint32 min_debt = type(uint32).max;
3     for (uint i = 0; i < path_length - 1; i++) {
4         uint32 debt = lookup_table[path[i]][path[i + 1]];
5         if (debt < min_debt) {
6             min_debt = debt;
7         }
8     }
9
10    for (uint i = 0; i < path_length - 1; i++) {
11        lookup_table[path[i]][path[i + 1]] -= min_debt;
12    }
13 }

```

resolve\_path 函数用于解决找到的债务循环路径。它首先找到路径上的最小债务金额，然后减去所有路径上的债务，以达到优化债务结算的目的。

该合约是一个实现债务跟踪和解决循环债务的区块链应用程序，它主要包含以下几个关键元素和逻辑：



### 1. 状态变量

- `lookup_table`: 一个公开的映射表, 用于存储债务人 (`debtor`) 和债权人 (`creditor`) 之间的债务金额。
- `users`: 一个包含所有用户地址的数组, 用于追踪所有参与债务关系的用户。

### 2. 事件

- `IOUAdded`: 当新的债务关系被添加时触发的事件, 记录债务人、债权人和债务金额。

### 3. 函数

- `hasUser`: 检查特定用户是否已经存在于用户列表中。
- `lookup`: 查找特定债务人欠特定债权人的债务金额。
- `add_IOU`: 添加新的债务关系, 更新债务表, 并尝试解决可能产生的循环债务。
- `getUsers`: 返回所有用户的地址列表。
- `resolve_cycle`: 尝试解决债务人和债权人之间可能存在的循环债务。
- `find_cycle`: 递归查找债务循环。
- `get_user_index`: 获取用户在用户列表中的索引。
- `resolve_path`: 解决找到的循环债务路径。

该智能合约通过使用区块链技术实现了去中心化的债务记录和循环债务解决系统。通过 `'lookup_table'` 存储债务信息, 通过深度优先搜索和路径优化算法解决循环债务问题, 极大地提高了债务管理的效率和透明度。通过事件触发机制, 用户可以实时跟踪债务的变化。该合约适用于多人参与的债务管理系统, 能够有效处理复杂的债务关系和循环债务问题。

## (四) 客户端脚本

### 1. 智能合约交互函数

```
1 // 返回系统中所有用户（债权人或债务人）的列表
2 async function getUsers() {
3   try {
4     const userList = await BlockchainSplitwise.methods.getUsers().call()
5     ;
6     return userList;
7   } catch (error) {
8     console.error("获取用户列表失败:", error);
9     return [];
10  }
```

`getUsers()` 函数通过调用合约中的 `getUsers` 函数从区块链上的智能合约中检索所有注册用户(即所有参与债务关系的地址)的列表,使用 `await` 关键字等待 `BlockchainSplitwise.methods.getUsers().call()` 的执行结果。

```
1 // 查询债务金额
2 async function lookup(debtor, creditor) {
3     try {
4         const amount = await BlockchainSplitwise.methods.lookup(debtor,
5             creditor).call();
6         return parseInt(amount);
7     } catch (error) {
8         console.error(查询 debtor creditor 的金额失败:, error);
9         return 0;
10    }
```

lookup 函数通过调用智能合约的 lookup 函数查询区块链上智能合约记录的特定债务人 (debtor) 对特定债权人 (creditor) 的债务金额, 同样也是使用 await 关键字等待。

```
1 // 获取指定用户 'user' 所欠的总金额
2 async function getTotalOwed(user) {
3     let owedAmount = 0;
4     const userList = await getUsers();
5     for (let i = 0; i < userList.length; i++) {
6         const amount = await lookup(user, userList[i]);
7         owedAmount += amount;
8     }
9     return owedAmount;
10 }
```

这个函数的作用是获取指定用户 'user' 所欠的总金额。通过调用 getUsers() 函数获取系统中所有用户的列表。这个列表包括了所有可能与指定用户有债务关系的地址。接下来遍历用户列表, 对每个用户使用 lookup() 函数查询指定用户 (user) 欠该用户 (usersList[i]) 的债务金额, 将查询到的每笔债务金额累加到 owedAmount 变量中。遍历完成后, 返回累计的债务总额 owedAmount。

```
1 // 获取用户最后一次发送或接收 IOU 的时间 (基于事件监听)
2 async function getLastActive(user) {
3     try {
4         // 获取用户作为债务人的事件
5         const debtorEvents = await BlockchainSplitwise.getPastEvents('
6             IOUAdded', {
7                 filter: { debtor: user },
8                 fromBlock: 0,
9                 toBlock: 'latest'
10            });
11
12         // 获取用户作为债权人的事件
13         const creditorEvents = await BlockchainSplitwise.getPastEvents('
14             IOUAdded', {
15                 filter: { creditor: user },
16                 fromBlock: 0,
17                 toBlock: 'latest'
18            });
```

```

16     });
17
18     // 合并两个事件数组
19     const allEvents = debtorEvents.concat(creditorEvents);
20
21     if (allEvents.length === 0) {
22         return null;
23     }
24
25     // 按照区块号排序, 找到最新的事件
26     allEvents.sort((a, b) => b.blockNumber - a.blockNumber);
27     const latestEvent = allEvents[0];
28     const block = await web3.eth.getBlock(latestEvent.blockNumber);
29     return block.timestamp;
30 } catch (error) {
31     console.error("获取最后活跃时间失败:", error);
32     return null;
33 }
34 }

```

getLastActive 函数旨在确定指定用户 (user) 在区块链上最后一次与智能合约交互的时间。使用 BlockchainSplitwise.getPastEvents 方法分别获取用户作为债务人和债权人时的 IOUAdded 事件。这包括从区块链的起始区块 (fromBlock: 0) 到最新区块 (toBlock: 'latest') 的所有相关事件, 将两个事件数组合并为一个, 以便统一处理。如果合并后的事件数组为空, 说明没有找到任何相关事件, 函数返回 null。

否则对事件数组按区块号降序排序, 选择最新的事件(allEvents[0]), 通过 latestEvent.blockNumber 获取最新事件所在的区块信息, 返回该区块的 timestamp, 即用户最后一次活跃的时间。

```

1 // 向系统添加一个 IOU ( "我欠你" )
2 async function add_IOU(creditor, amount) {
3     try {
4         const accounts = await web3.eth.getAccounts();
5         const debtor = accounts[0];
6
7         // 验证债权人地址不与债务人地址相同
8         if (creditor.toLowerCase() === debtor.toLowerCase()) {
9             alert("债权人地址不能与债务人地址相同!");
10            return;
11        }
12
13        // 验证债权人地址格式
14        if (!web3.utils.isAddress(creditor)) {
15            alert("无效的债权人地址!");
16            return;
17        }
18
19        // 验证金额
20        const amountInt = parseInt(amount);

```

```

21     if (isNaN(amountInt) || amountInt <= 0) {
22         alert("金额必须是大于0的数字!");
23         return;
24     }
25
26     // 发送交易
27     await BlockchainSplitwise.methods.add_IOU(creditor, amountInt)
28         .send({ from: debtor, gas: 300000 })
29         .on('receipt', function(receipt){
30             console.log('交易成功:', receipt);
31             updateAccountInfo();
32             populateUsers();
33         })
34         .on('error', function(error){
35             console.error('交易失败:', error);
36             alert("添加IOU失败: " + error.message);
37         });
38     } catch (error) {
39         console.error("添加IOU失败:", error);
40         alert("添加IOU失败: " + error.message);
41     }
42 }

```

add\_IOU 函数用于在区块链上创建一个新的债务记录。它允许用户输入债权人 (creditor) 的地址和欠款金额 (amount)，然后通过智能合约将这笔债务记录在区块链上。

首先获取用户的账户列表，选择第一个账户作为债务人 (debtor)。然后检查输入的债权人地址是否与债务人地址相同，如果是，则弹出警告并终止操作。接下来使用 web3.utils.isAddress 验证输入的债权人地址是否为有效的以太坊地址格式。将输入的金额转换为整数，并检查是否为正数。如果不是有效数字或小于等于 0，则弹出警告并终止操作。一切检查无误之后，调用智能合约的 add\_IOU 方法，发送交易到区块链以添加新的债务记录。这需要指定交易的发送者 (from: debtor) 和 gas 限制 (gas: 300000)。如果交易成功，打印交易收据，更新用户界面上的账户信息和用户列表。如果交易失败，打印错误信息并弹出警告。

```

1 // 获取邻居用户（欠款对象）
2 async function getNeighbors(user) {
3     const usersList = await getUsers();
4     let neighbors = [];
5     for (let i = 0; i < usersList.length; i++) {
6         const debt = await lookup(user, usersList[i]);
7         if (debt > 0) {
8             neighbors.push(usersList[i]);
9         }
10    }
11    return neighbors;
12 }

```

getNeighbors 函数旨在获取指定用户 (user) 的所有“邻居”用户，即所有该用户有债务关系（无论是作为债务人还是债权人）的其他用户。

首先, 通过调用 `getUsers()` 函数异步获取系统中所有用户的列表, 并创建一个空数组 `neighbors` 用于存储与指定用户有债务关系的其他用户。使用 `for` 循环遍历所有用户, 对于每个用户, 通过 `lookup(user, usersList[i])` 异步查询债务金额。这里 `user` 是函数参数, `usersList[i]` 是当前遍历到的用户地址。: 如果查询到的债务金额大于 0, 意味着指定用户与当前遍历的用户之间存在债务关系, 将当前遍历的用户添加到 `neighbors` 数组中。遍历完成后, 返回包含所有邻居用户地址的数组。

## 2. 用户界面更新函数

```

1 // 更新账户信息
2 async function updateAccountInfo() {
3     const account = web3.eth.defaultAccount;
4     if (!account) {
5         console.error("未找到默认账户");
6         return;
7     }
8     const totalOwed = await getTotalOwed(account);
9     const lastActive = await getLastActive(account);
10    $("#total_owed").html("$" + totalOwed);
11    $("#last_active").html(timeConverter(lastActive));
12    $("#debtor_address").html(account); // 显示债务人地址
13 }
14 // 填充账户选项
15 function populateAccounts(accounts) {
16     const opts = accounts.map(function (a) { return '<option_value="' + a + '">' +
17         a + '</option>' }).join('');
18     $("#myaccount").html(opts);
19 }
20 // 填充钱包地址列表
21 function populateWalletAddresses(accounts) {
22     const listItems = accounts.map(function (a) { return '<li>' + a + '</li>' }).
23         join('');
24     $(".wallet_addresses").html(listItems);
25 }
26 // 填充所有用户列表
27 async function populateUsers() {
28     const usersList = await getUsers();
29     const listItems = usersList.map(function (u,i) { return "<li>" + u + "</li>"
30         }).join('');
31     $("#all_users").html(listItems);
32 }

```

这些函数共同构成了用户界面与区块链智能合约交互的核心部分, 允许用户查看和管理他们的债务信息, 以及系统中的其他用户信息。`updateAccountInfo()` 的功能是更新用户界面上显示的账户信息, 包括总欠款金额和最后活跃时间。`populateAccounts(accounts)` 的功能是填充下拉菜单, 显示所有可用的以太坊账户, 允许用户选择不同的账户进行操作。`populateWalletAd-`

dressses(accounts) 的功能是在用户界面上显示所有钱包地址的列表。populateUsers() 的功能是填充用户界面上的所有用户列表，显示系统中所有参与债务关系的用户的地址。，允许用户实时查看和管理系统状态。它们通过异步调用智能合约方法来获取数据，然后使用 jQuery 更新 HTML 元素，从而实现前后端的交互。

### 3. 用户界面交互

```
1 // 时间转换函数
2
3 function timeConverter(UNIX_timestamp){
4     if (!UNIX_timestamp) return "invalid_date";
5     var a = new Date(UNIX_timestamp * 1000);
6     var months = [ 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
7                   'Nov', 'Dec' ];
8     var year = a.getFullYear();
9     var month = months[a.getMonth()];
10    var date = a.getDate();
11    var hour = a.getHours();
12    var min = a.getMinutes();
13    var sec = a.getSeconds();
14    var time = date + '-' + month + '-' + year + '-' + hour + ':' + min + ':' +
15              + sec ;
16    return time;
17 }
```

此函数适用于需要将区块链事件或智能合约日志中获取的 UNIX 时间戳转换为人类可读日期格式的场景。

```
1 // 当页面加载完成后更新信息
2 $(document).ready(async function() {
3     const accounts = await web3.eth.getAccounts();
4     if (accounts.length > 0) {
5         populateAccounts(accounts);
6         populateWalletAddresses(accounts);
7         populateUsers();
8     }
9
10    // 账户切换事件
11    $("#myaccount").change(async function() {
12        const selectedAccount = $(this).val();
13        web3.eth.defaultAccount = selectedAccount;
14        await updateAccountInfo();
15    });
16
17    // 当点击添加 IOU 按钮时运行 'add_IOU' 函数
18    $("#addiou").click(function() {
19        const creditor = $("#creditor").val();
20        const amount = $("#amount").val();
21        add_IOU(creditor, amount);
22    });
23 }
```

```
22     });
23
24     // 监听 IOUAdded 事件，实时更新用户界面
25     BlockchainSplitwise.events.IOUAdded({
26         fromBlock: 'latest'
27     }, function(error, event){
28         if (error) {
29             console.error("事件监听错误:", error);
30             return;
31         }
32         console.log("IOUAdded 事件:", event);
33         updateAccountInfo();
34         populateUsers();
35     });
36 });
```

这是用户界面初始化和事件监听的核心部分，它确保了当网页加载完成后，用户界面能够正确显示账户信息，并且能够响应用户的交互，如切换账户、添加新的债务（IOU）记录，以及实时更新界面以反映区块链上发生的事件。

客户端的功能主要围绕与智能合约的交互以及用户界面的更新。

#### 1. 智能合约交互函数

- `getUsers()`: 从智能合约中获取所有用户（债权人或债务人）的列表。
- `lookup(debtor, creditor)`: 查询特定债务人欠特定债权人的债务金额。
- `getTotalOwed(user)`: 获取用户最后一次发送或接收 IOU 的时间戳。
- `getLastActive(user)`: 查询特定债务人欠特定债权人的债务金额。
- `add_IOU(creditor, amount)`: 添加一个新的债务关系到智能合约。此函数首先验证输入的合法性，然后发送交易到智能合约以添加债务，并在交易成功后更新用户界面。
- `getNeighbors(user)`: 获取指定用户的邻居用户列表，即所有该用户欠款或被欠款的用户。

#### 2. 用户界面更新函数

- `updateAccountInfo()`: 此函数用于更新用户界面上显示的账户信息，包括总欠款金额和最后活跃时间。它通过调用 `getTotalOwed()` 和 `getLastActive()` 获取数据，并更新页面元素的内容。
- `populateAccounts()`: 填充账户下拉列表，允许用户选择不同的账户进行操作。
- `populateWalletAddresses()`: 显示用户的钱包地址列表。
- `populateUsers()`: 填充用户列表，显示所有参与债务关系的用户的地址。

#### 3. 事件监听

- 客户端监听智能合约的 `IOUAdded` 事件，一旦有新的债务关系被添加，就会触发事件的回调函数，该函数会调用 `updateAccountInfo()` 和 `populateUsers()` 来更新用户界面。

#### 4. UI 交互

- 当用户点击“添加 IOU”按钮时，会触发 add\_IOU() 函数，该函数会读取输入的债权人地址和金额，然后与智能合约交互以添加新的债务关系。
- 当用户在“我的账户”下拉列表中选择不同的账户时，会触发 updateAccountInfo() 函数，该函数会更新显示当前选中账户的总欠款金额和最后活跃时间。

## 四、 实验结果

通过本次实验，我们成功实现了一个基本的债务跟踪 DApp。用户可以通过界面添加债务关系，系统能够自动检测并解决债务循环。

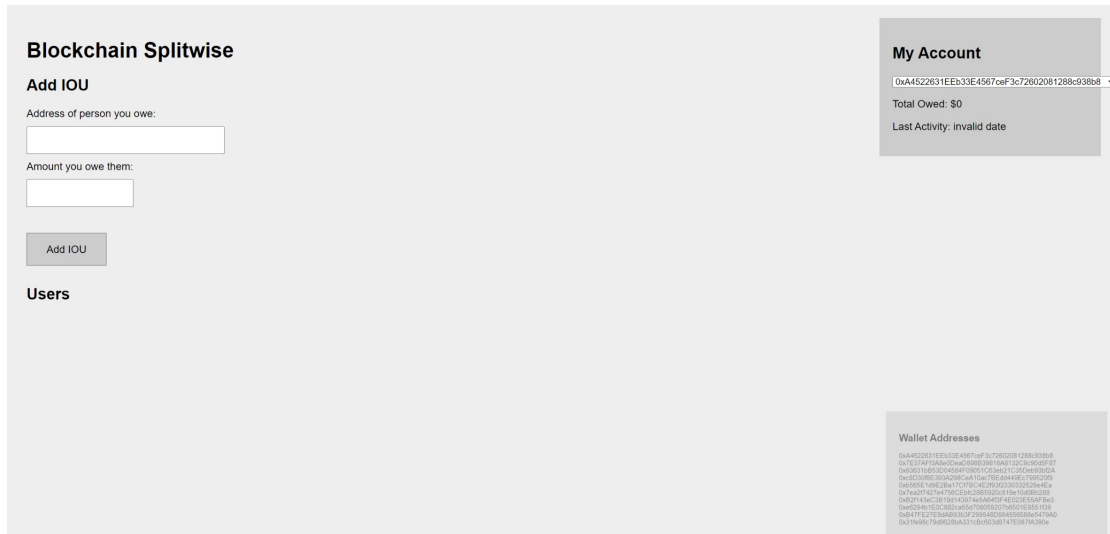


图 5: 初始界面

账户列表如下，考虑到报告的简洁和可读性，在这里按自上而下的顺序用账户 a,b,c... 进行代称。

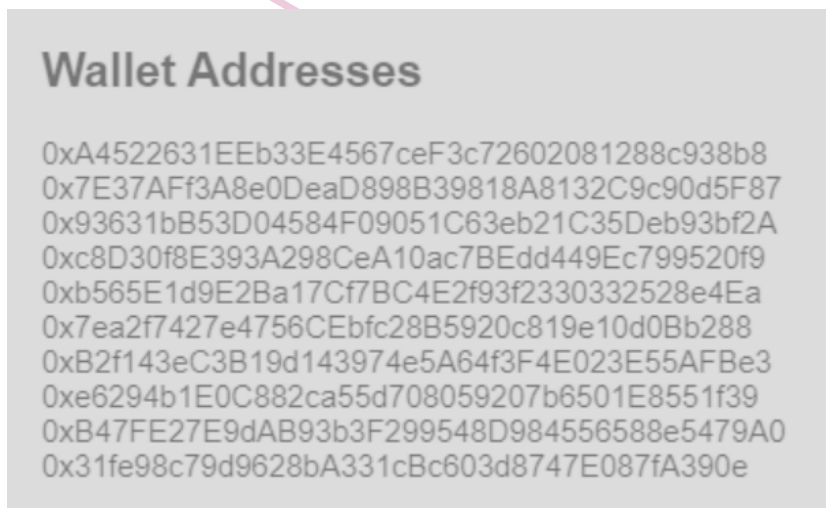


图 6: 所有账户

我们在网页中操作，使得账户 a 欠账户 b5 元。发现债务成功添加，Last Activity 成功记录了交易发生的时间，Users 中出现了参与交易的两个账户，说明项目代码编写正确，运行成功。



Blockchain Splitwise

Add IOU

Address of person you owe:

0x7E37AFf3A8e0DeaD898B39818A81\*

Amount you owe them:

5

Add IOU

Users

0xA4522631EEb33E4567ceF3c72602081288c938b8  
0x7E37AFf3A8e0DeaD898B39818A8132C9cd05F87

My Account

0xA4522631EEb33E4567ceF3c72602081288c938b8

Total Owed: \$5

Last Activity: 8 Dec 2024 19:42:4

Wallet Addresses

0xA4522631EEb33E4567ceF3c72602081288c938b8  
0x7E37AFf3A8e0DeaD898B39818A8132C9cd05F87  
0xB0310B8CD0A4FA7505FC8ba113CEdab8ADCA  
0xC8C00BE95DA20BCA19Ac7FB836446Ec7965209  
0xDAB5E165E2Ba1CC7FC42B93ED35323b4ebA  
0xF7ea2FF427ea76AC5EC25B852dc319a10dBEd38  
0x2DF6AwC3B13a8143817aaBA847E4EB3DE5AFBvJ  
0x25cb41EDC48DCa5A7005A5207AA501E85AfFD9  
0xb47FE2725AAD8D5CF2564504506A5068acF0Ad3  
0xd31fa96c7b3962BA331b0c953db74TE087A360a

图 7: a 欠 b 截图

接下来为了检验消除债务循环的功能，我们还需要进行验证。我们使用账户 c,d,e 相互借款 5 形成债务循环，进行验证：

Blockchain Splitwise

Add IOU

Address of person you owe:

0xc8D3098E9393A298C6A10ac7BE6d44

Amount you owe them:

5

Add IOU

My Account

0xc8D3098E9393A298C6A10ac7BE6d44

Total Owed: \$0

Last Activity: 8 Dec 2024 19:48:29

图 8

Blockchain Splitwise

Add IOU

Address of person you owe:

0xb665E1d9E2Ba17CF7BC4E2f93233f

Amount you owe them:

5

Add IOU

My Account

0xb6E50fEE303A298cAdA78a78E69449Ec796A20f8

Total Owed: \$0

Last Activity: 8 Dec 2024 19:54:41

图 9

Blockchain Splitwise

Add IOU

Address of person you owe:

Dx9R631tB53Dd4584F09051C63eb21c

Amount you owe them:

5

Add IOU

My Account

0xb6f6f...Total:28w1TzPfcUd...25x3Z333252eaf...

Total Owed: \$0

Last Activity: 8 Dec 2024 19:53:8

图 10

当第三个交易完成之后，发现三个账户的欠款都归为 0，说明成功消除了债务循环。

## 五、 实验心得

### (一) 遇到的问题

- 合约部署: 初次尝试在 Ganache 上部署合约时遇到问题。发现是 EVM 版本不一致导致的错误。使用 `ganache-cli --evm-version istanbul` 运行节点解决。
- 异步处理: 由于客户端函数是异步函数, 返回的是 `promise` 对象, 导致原框架无法兼容, 产生很多报错。通过学习 `async/await` 语法, 实现有效地管理异步操作。
- 用户界面更新: 在页面加载时更新用户界面数据存在问题。通过在 `$(document).ready()` 中正确调用更新函数, 确保了数据的正确加载。

### (二) 合约减小开销的措施

- 使用 `uint32`: 债务金额使用 `uint32` 而不是更大的数据类型, 这减少了存储债务信息所需的空間, 从而降低 `gas` 费用。
- 债务循环解决: 合约尝试检测并解决债务循环, 这可以减少链上不必要的债务记录, 从而降低 `gas` 费用。
- 条件检查: 在 `add_IOU` 函数中, 通过 `require` 语句进行条件检查, 确保只有有效的交易才能继续执行, 这有助于避免无效或恶意交易消耗 `gas`。