实验一报告

智能合约基础与以太坊机制初探

Lab 1: Smart Contract Basics & Ethereum Mechanism Exploration

姓名：范舒舰

学号：23121677

专业：网络空间安全

2026 年 3 月 15 日

目录

1 实验目的

1. **掌握开发环境**
   熟悉 Remix IDE（https://remix.ethereum.org/），这是一个用于编写、编译和部署智能合约的基于浏览器的集成开发环境。

2. **掌握 Solidity 基础**
   学习 Solidity 合约的基本结构，包括代码版权声明（License）、编译器版本（Pragma）和合约定义（Contract）。

3. **理解以太坊账户模型**
   探索以太坊中的两类账户：

   - 外部账户（Externally Owned Accounts, EOA）
   - 合约账户（Contract Accounts）

4. **探索区块链操作的经济模型**
   理解以下两者之间的区别：

   - 改变状态的交易（消耗 Gas）
   - 只读调用（在本地执行，不消耗 Gas）

2 实验背景与实验要求

2.1 实验背景

传统的捐款箱往往透明度较低。收集到的资金可能被管理员挪用，且捐赠者无法验证资金的具体去向。区块链技术提供了一个透明且可验证的账本。在本实验中，你将实现一个去中心化的捐赠合约（Tip Jar），确保：

- **透明性**：任何人都可以检查区块链上的合约余额和捐赠记录。
- **通过代码逻辑实现安全性**：资金只能根据智能合约中预定义的规则进行处理。

本实验展示了如何通过程序化规则在去中心化系统中建立信任。

2.2 实验要求

2.2.1 基础功能（必做）

编写并部署一个名为 `TipJar` 的智能合约，包含以下功能：

3.1.1. **管理员机制**

使用变量 `address public owner;` 存储合约创建者的地址。owner 应在构造函数中初始化：`owner = msg.sender;`

3.1.2. **捐赠接口**

实现一个 `payable` 函数 `donate()`，允许用户向合约发送 ETH。你可以使用以下内置变量：

- `msg.sender` – 调用者的地址
- `msg.value` – 随交易发送的 ETH 数量

3.1.3. **余额查询**

实现一个 `view` 函数，返回合约中存储的当前 ETH 余额。例如：

Solidity

```
function getBalance() public view returns (uint) {
    return address(this).balance;
}
```

3.1.4. **资金提取**

实现函数 `withdraw()`，确保只有合约所有者可以提取资金。使用 `require()` 强制执行访问控制。例如：`require(msg.sender == owner, "Only owner can withdraw");`

2.2.2 进阶功能（必做）

通过实现以下机制扩展你的合约：

3.2.1. **黑名单机制**

维护一个映射： `mapping(address => bool) public blacklist;` 管理员应能将特定地址标记为黑名单，黑名单地址不允许进行捐赠。示例逻辑：

```
require(!blacklist[msg.sender], "Address is blacklisted");
```

3.2.2. **众筹目标**

定义一个常量目标金额：

`uint public constant GOAL = 5 ether;` 一旦合约余额达到此目标，`donate()` 函数应停止接收捐赠。示例逻辑：

```
require(address(this).balance < GOAL, "Funding goal reached");
```

3.2.3. **数据分析**

在合约中跟踪以下统计数据：

- 单笔最高捐赠金额

- 独立捐赠者总数

建议使用的变量： `uint public highestDonation; uint public donorCount; mapping(address => bool) public hasDonated;` 即使一个捐赠者多次捐赠，也只能计算一次。

3.2.4. 时间锁（冷却期）

实施时间限制，使得所有者只能在合约部署 5 分钟后进行第一次提取。使用变量：`uint public deployTime;` 在部署期间初始化：`deployTime = block.timestamp;` 仅在以下条件满足时允许提取：`block.timestamp >= deployTime + 5 minutes;`

3 开发环境说明

本实验使用浏览器中的 Remix IDE 作为智能合约开发环境。Remix IDE 集成了 Solidity 编辑器、编译器、部署面板和合约交互界面，能够在不配置本地链的情况下完成从编写到测试的完整流程。

本实验开发环境如下：

- 开发工具：Remix IDE
- 编程语言：Solidity
- 编译器版本：latest local version - soljson-v0.8.31+commit.fd3a2265.js
- 部署环境：Remix VM

4 理论基础

4.1 Solidity 合约基本结构说明

一个 Solidity 智能合约通常包含以下几个基本部分：

1. **许可证声明（License Declaration）**
   用于说明代码许可证，例如：

```
// SPDX-License-Identifier: MIT
```

2. **Pragma 版本声明（Pragma Version）**
   用于指定合约编译器版本范围，例如：

```
pragma solidity ^0.8.31;
```

3. **合约定义（Contract Definition）**
   用于定义合约的状态变量、构造函数、修饰器以及函数逻辑，例如：

```
contract TipJar {
    // state variables
    // constructor
    // functions
}
```

在本实验中，TipJar 合约通过状态变量保存管理员地址、部署时间、黑名单信息以及统计数据，并通过函数实现捐款、查询余额和提款等功能。

4.2 以太坊账户模型说明

以太坊中主要存在两种账户：

4.2.1 外部账户（EOA, Externally Owned Account）

外部账户由用户通过私钥控制。在 Remix 中，ACCOUNT 下拉菜单中的地址就是外部账户。其特点包括：

- 由私钥控制
- 可主动发起交易
- 可调用智能合约函数
- 可携带 ETH 与合约交互

在本实验中，不同测试账户分别扮演部署者、普通捐款者和黑名单地址等角色。

4.2.2 合约账户（Contract Account）

合约账户在智能合约部署后生成，对应 Deployed Contracts 面板中的合约地址。其特点包括：

- 不由私钥控制
- 由智能合约代码控制
- 不能主动发起交易
- 在被调用时自动执行预定义逻辑

本实验中的 TipJar 即为一个合约账户，它用于接收捐款、记录数据和执行提款规则。

4.3 区块链操作的经济模型说明

在以太坊中，区块链操作可分为两类：

4.3.1 改变状态的交易（State-changing Transactions）

改变区块链状态的操作需要广播到网络，并由节点执行和记录，因此会消耗 Gas。例如：

- 调用 `donate()` 发送 ETH
- 调用 `withdraw()` 提取资金
- 调用 `setBlacklist()` 修改黑名单状态

这类操作都会改变合约状态，因此需要支付 Gas。

4.3.2 只读调用（Read-only Calls）

只读调用不会改变区块链状态，通常可以在本地执行，因此不消耗 Gas。例如：

- 调用 `getBalance()`
- 查询 `owner`
- 查询 `highestDonation`
- 查询 `blacklist(address)`

通过本实验可以清晰观察到：写操作会产生交易记录，而读操作仅返回当前合约状态。

4.4 关键 Solidity 变量与函数说明

本实验中使用了以下关键概念：

- `msg.sender`：当前调用者地址
- `msg.value`：本次交易随附发送的 ETH 数量
- `address(this).balance`：当前合约地址持有的 ETH 余额
- `require()`：条件检查，不满足时交易回滚
- `block.timestamp`：当前区块时间戳，用于实现时间锁

5 合约设计说明

5.1 合约主要状态变量

本实验设计的 `TipJar` 合约主要使用以下状态变量：

表 1: 主要状态变量说明

变量名	作用说明
`owner`	保存合约创建者地址，用于管理员权限控制
`GOAL`	众筹目标金额，设定为 5 ether
`blacklist`	黑名单映射，记录禁止捐款的地址
`highestDonation`	记录当前最高单笔捐款金额
`donorCount`	记录唯一捐款者数量
`hasDonated`	标记某地址是否已经捐款过
`deployTime`	记录合约部署时间
`firstWithdrawalDone`	记录是否已经完成第一次提款

5.2 合约功能说明

1. **管理员初始化**
   通过构造函数将部署者设置为 owner，并记录部署时间。

2. **捐款功能**
   使用 `payable` 函数接收 ETH，并在函数内部进行黑名单检查、募资目标检查以及统计数据更新。

3. **余额查询**
   返回合约账户中当前持有的 ETH 数量。

4. **提款功能**
   限制只有 owner 可调用，且第一次提款必须满足部署 5 分钟后的时间限制。

5. **黑名单管理**
   管理员可设置某地址是否处于黑名单状态。

6. **数据分析功能**
   自动统计最高单笔捐款和唯一捐款者数量。

6 实验步骤（基础部分）

6.1 步骤一：编写基础功能代码

首先在 Remix IDE 中新建 `TipJar.sol` 文件，完成管理员机制、捐赠接口、余额查询和提款功能的实现。基础部分需要满足以下要求：使用 `address public owner;` 保存部署者地址，在构造函数中执行 `owner = msg.sender;`；实现 `payable` 函数 `donate()` 接收 ETH；实现 `getBalance()` 返回合约余额；实现 `withdraw()` 并通过 `require()` 限制仅 owner 可调用。

6.2 步骤二：编译基础功能代码

进入 `Solidity Compiler` 面板，选择编译器版本 `soljson-v0.8.31+commit.fd3a2265.js`，编译 `TipJar.sol`。当 Remix 显示绿色对勾时，说明代码已通过编译。

6.3 步骤三：部署并检查基础功能初始状态

进入 `Deploy & Run Transactions` 面板，将环境设置为 `Remix VM`，使用第一个测试账户部署合约。部署成功后，合约实例出现在 `Deployed Contracts` 面板中。随后依次调用 `owner`、`getBalance()`、`highestDonation` 和 `donorCount` 进行初始检查。由图1可见，owner 地址已正确初始化，且在部署初始阶段合约余额为 0，`highestDonation` 与 `donorCount` 均为 0。

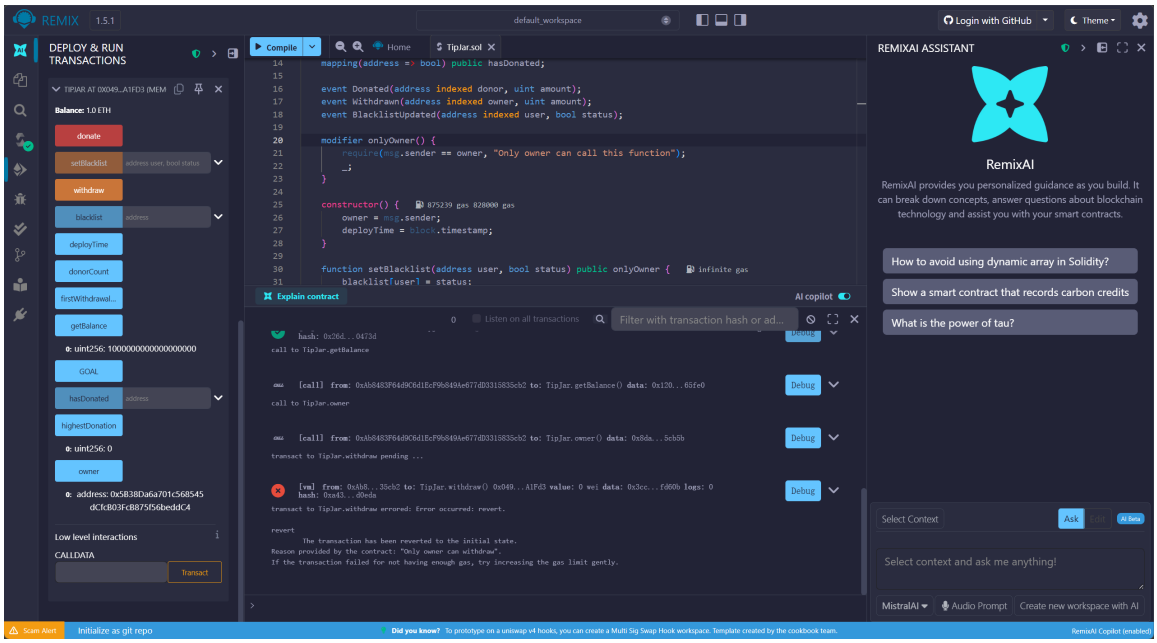

图 1: 合约部署成功后的初始状态

6.4 步骤四：测试基础功能

1. **验证管理员地址初始化**
   调用 `owner`，返回地址为 `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`，与部署时所选账户一致，说明构造函数已正确将部署者设置为 owner。

2. **测试普通账户捐赠与余额查询**
   切换到第二个测试账户 `0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2`，设置 `VALUE = 1 ether` 后调用 `donate()`。交易成功后再次调用 `getBalance()`，返回值变为 `1 ether`，说明普通账户可以正常向合约捐赠 ETH，且余额查询逻辑正确，如图2所示。

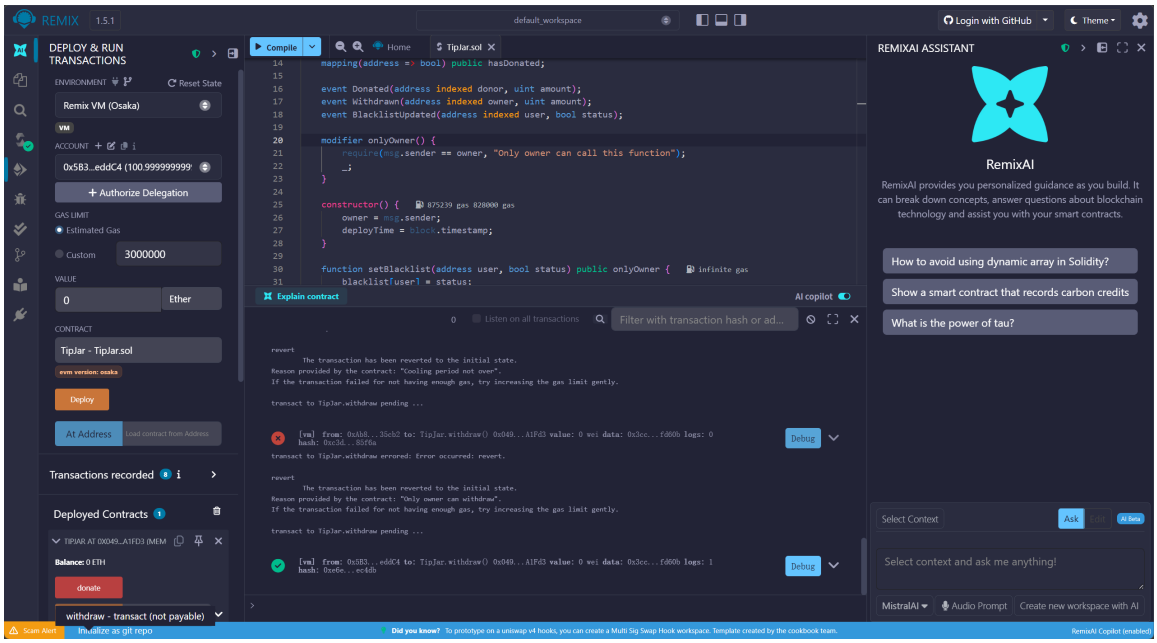

图 2: 普通账户首次捐赠成功，合约余额增加至 1 ETH

3. **验证非 owner 账户提款失败**
   保持当前账户为第二个测试账户，直接调用 `withdraw()`。交易被回滚，错误提示为 `Only owner can withdraw`，说明 `require(msg.sender == owner, ...)` 的访问控制逻辑正确生效，如图3所示。

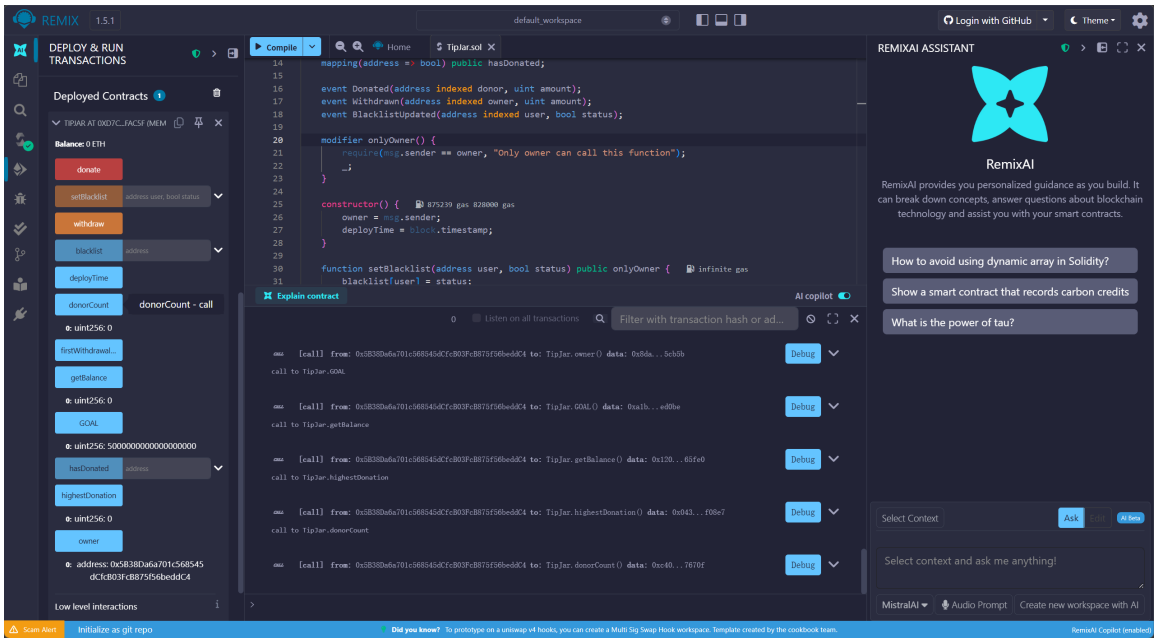

图 3: 非 owner 账户调用 `withdraw()` 失败

4. **验证 owner 可以提取资金**
   将账户切换回 owner 地址后调用 `withdraw()`。交易成功执行，随后再次调用 `getBalance()`，返回值为 0，说明 owner 具有正确的提款权限，且合约中的资金已被成功转出，如图4所示。

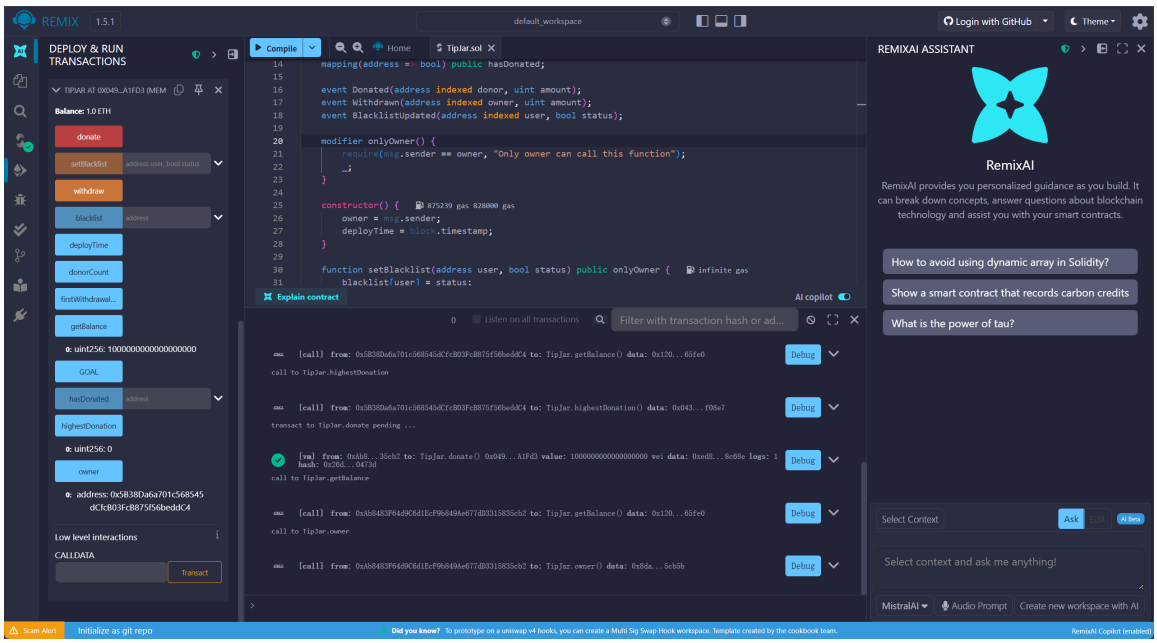

图 4: owner 成功调用 `withdraw()` 后合约余额归零

6.5 基础部分实验现象记录

根据实验截图，基础功能测试结果如下：

- 部署者地址（owner）：0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- owner 查询返回地址：0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
- 初始合约余额：0 ETH
- 第一次捐赠账户地址：0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
- 第一次捐赠金额：1 ETH
- 第一次捐赠后合约余额：1 ETH
- 非 owner 调用 withdraw() 的结果：失败
- 非 owner 调用失败提示信息：Only owner can withdraw
- owner 调用 withdraw() 的结果：成功
- 提款后合约余额：0 ETH

7 实验步骤（进阶部分）

7.1 步骤一：扩展为最终版本合约

在基础功能验证完成后，继续在 TipJar 合约中加入黑名单机制、众筹目标、数据统计与时间锁机制，形成最终版本合约。新增内容包括：

- 黑名单机制：mapping(address => bool) public blacklist;
- 众筹目标：uint public constant GOAL = 5 ether;
- 数据统计：highestDonation、donorCount、hasDonated
- 时间锁机制：deployTime 与第一次提款冷却期

7.2 步骤二：编译最终版本合约

在加入上述逻辑后，再次使用 soljson-v0.8.31+commit.fd3a2265.js 编译最终版本 TipJar.sol。编译成功后，说明最终版本合约已经满足部署与测试条件。

7.3 步骤三：部署最终版本合约并检查初始状态

重新部署最终版本合约后，依次调用 owner、getBalance()、GOAL、highestDonation 与 donorCount。图5显示，最终版本在部署时余额为 0，GOAL 已初始化为 5 ether，highestDonation 与 donorCount 均为 0，满足预期。

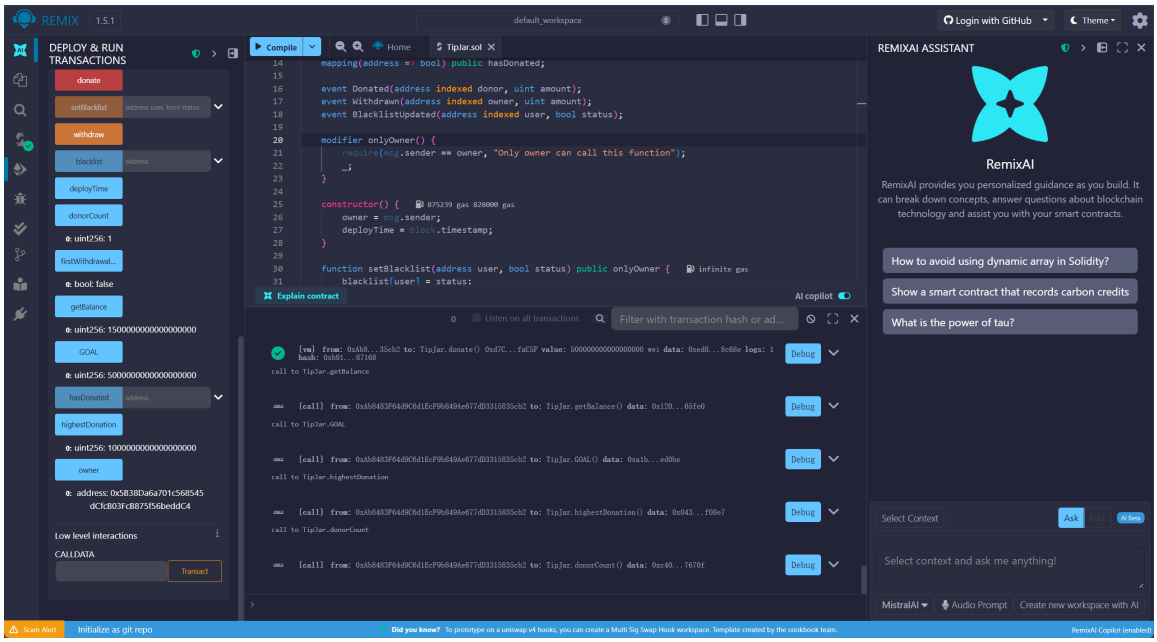

图 5: 最终版本合约部署后的初始状态

7.4 步骤四：测试进阶功能

1. **测试首次捐赠后的统计更新**
   使用第二个测试账户向合约捐赠 1 ETH。交易成功后，getBalance() 返回 1 ETH，highestDonation 更新为 1 ETH，donorCount 更新为 1，说明首次捐赠已被正确记录，如图6所示。

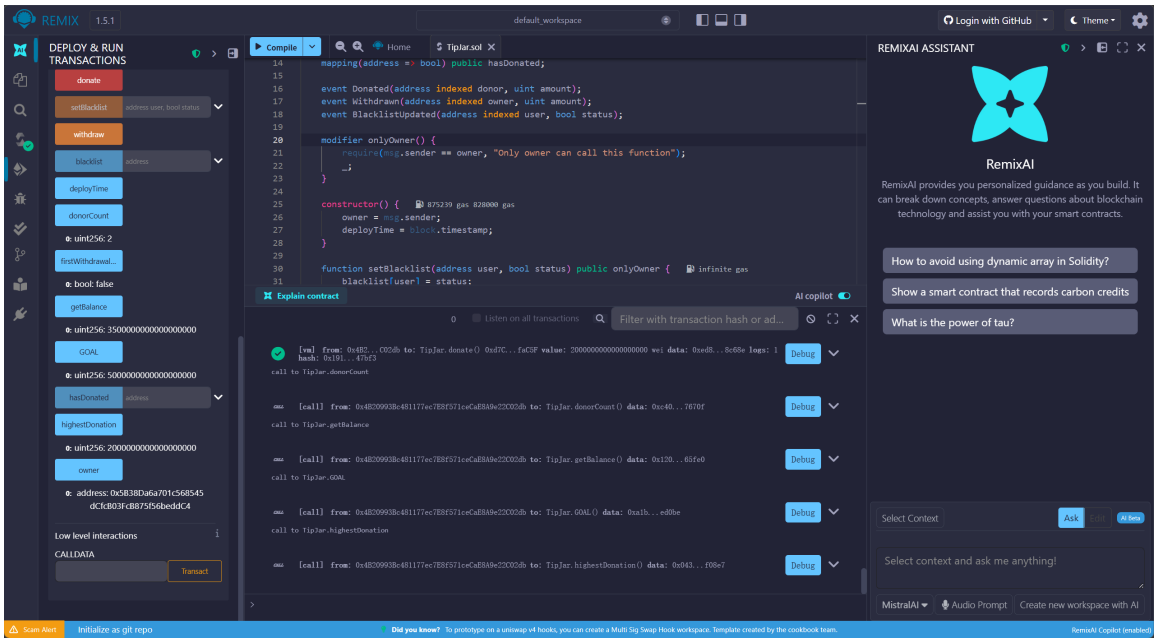

图 6: 首次捐赠后余额与统计数据更新

2. **测试重复捐赠不重复计入捐赠者数量**
   继续使用相同的第二个测试账户再次捐赠 0.5 ETH。此后 getBalance() 变为 1.5 ETH，而 donorCount 仍保持为 1，highestDonation 仍为 1 ETH，说明同一地址不会被重复统计为新的捐赠者，如图7所示。

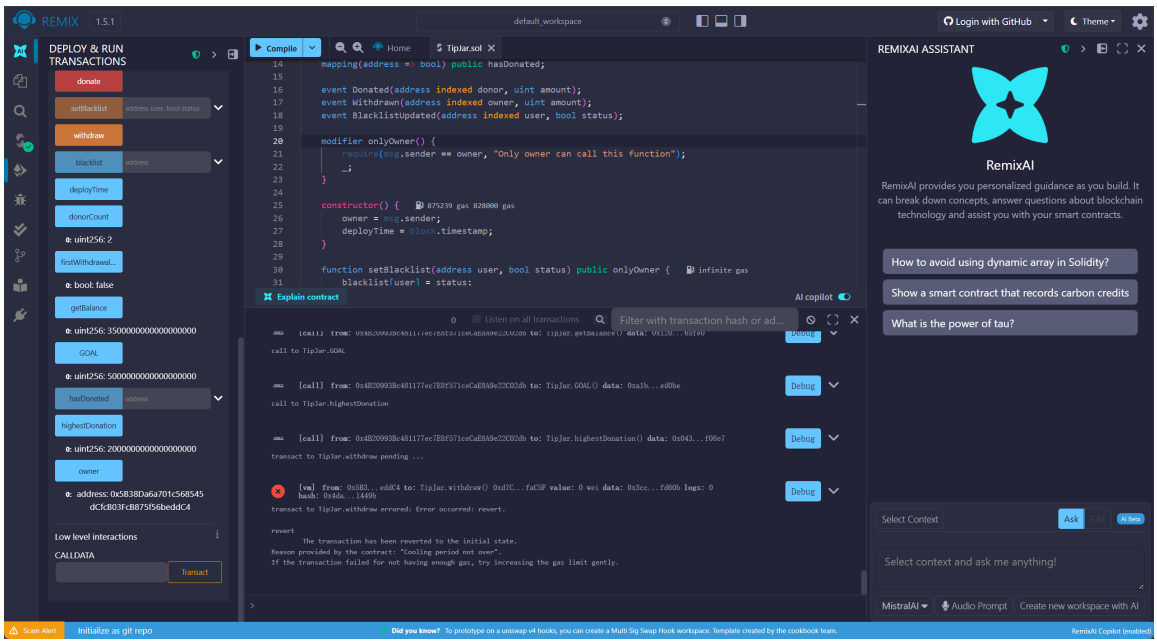

图 7: 同一地址重复捐赠后 donorCount 保持不变

3. **测试更大金额捐赠后的最高值更新**
   将账户切换为第三个测试账户 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db，

捐赠 2 ETH。交易成功后，getBalance() 变为 3.5 ETH，donorCount 更新为 2，highestDonation 更新为 2 ETH，说明最高单笔捐赠统计逻辑正确，如图8所示。

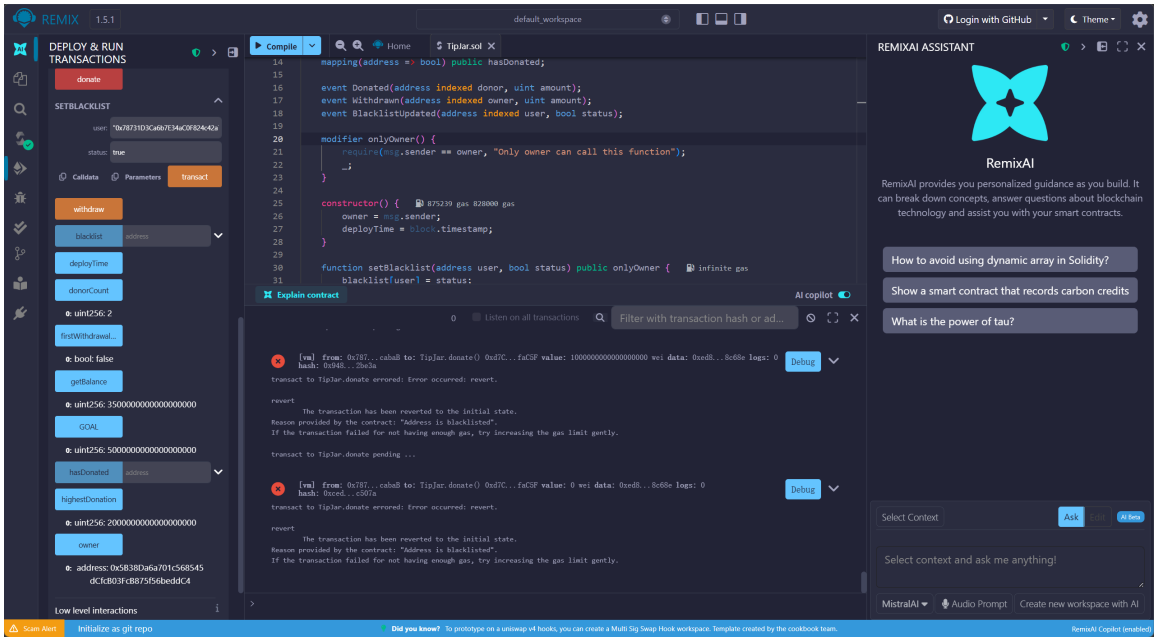

图 8: 新的更大金额捐赠更新 highestDonation

4. **测试第一次提款的时间锁机制**

在合约余额为 3.5 ETH 时，由 owner 首次调用 withdraw()。交易被回滚，错误提示为 Cooling period not over，说明第一次提款必须满足部署 5 分钟后的时间要求，如图9所示。

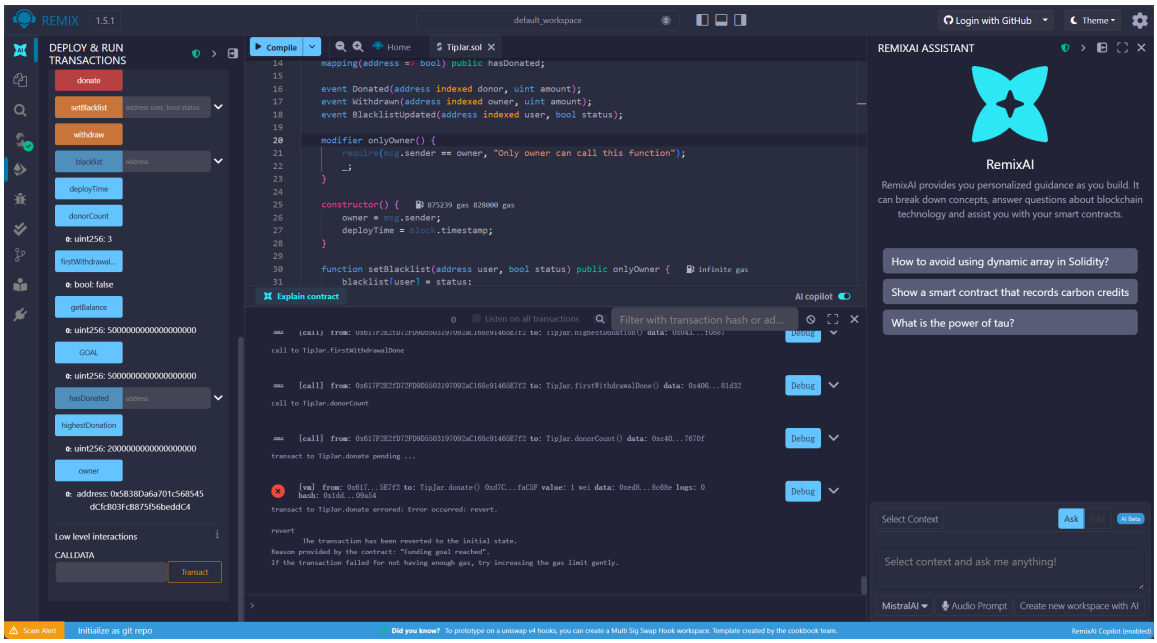

图 9: 首次提款因未达到冷却期而失败

5. **测试黑名单机制**

由 owner 调用 `setBlacklist()` 将地址 `0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB` 加入黑名单。随后使用该地址调用 `donate()`，交易失败并显示 `Address is blacklisted`，说明黑名单机制生效，如图10所示。

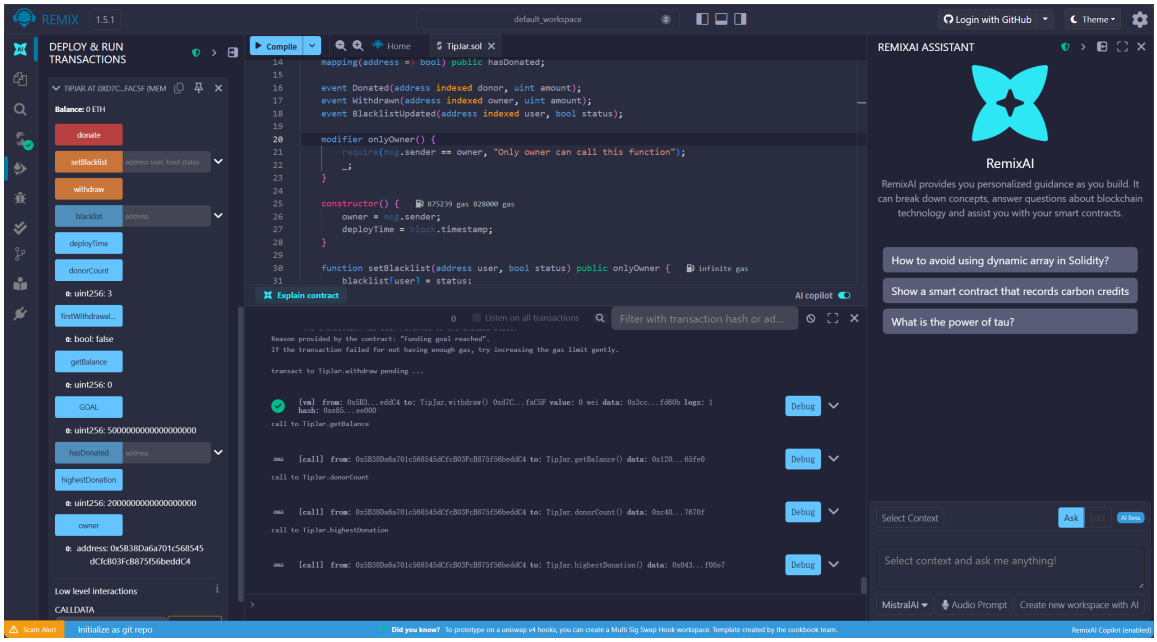

图 10: 黑名单地址调用 `donate()` 失败

6. **测试众筹目标限制**

在后续继续捐赠至合约余额达到 `5 ETH` 后，再次尝试调用 `donate()`。交易失败并显示 `Funding goal reached`，说明众筹目标限制已经生效。此时 `donorCount` 为 3，`highestDonation` 仍为 `2 ETH`，如图11所示。

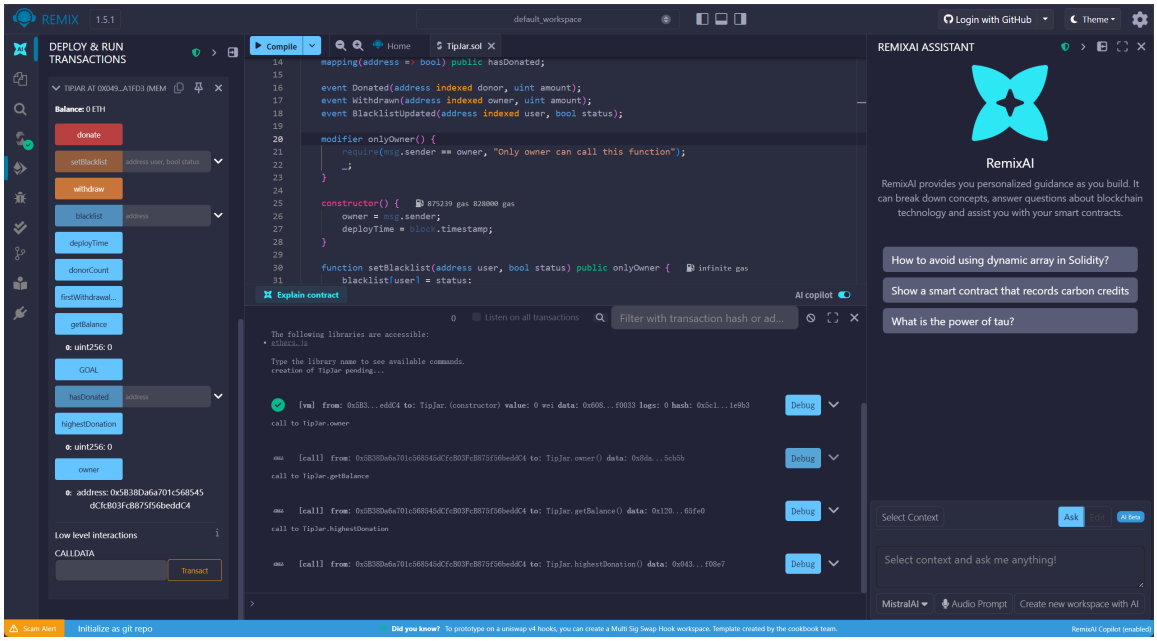

图 11: 合约余额达到 `GOAL` 后再次捐赠失败

7. **测试冷却期结束后的 owner 提款成功**
   在满足时间锁条件后，由 owner 再次调用 `withdraw()`。交易成功执行，随后 `getBalance()` 返回 0，说明合约中的资金已被全部提取，提款逻辑正确，如图12所示。

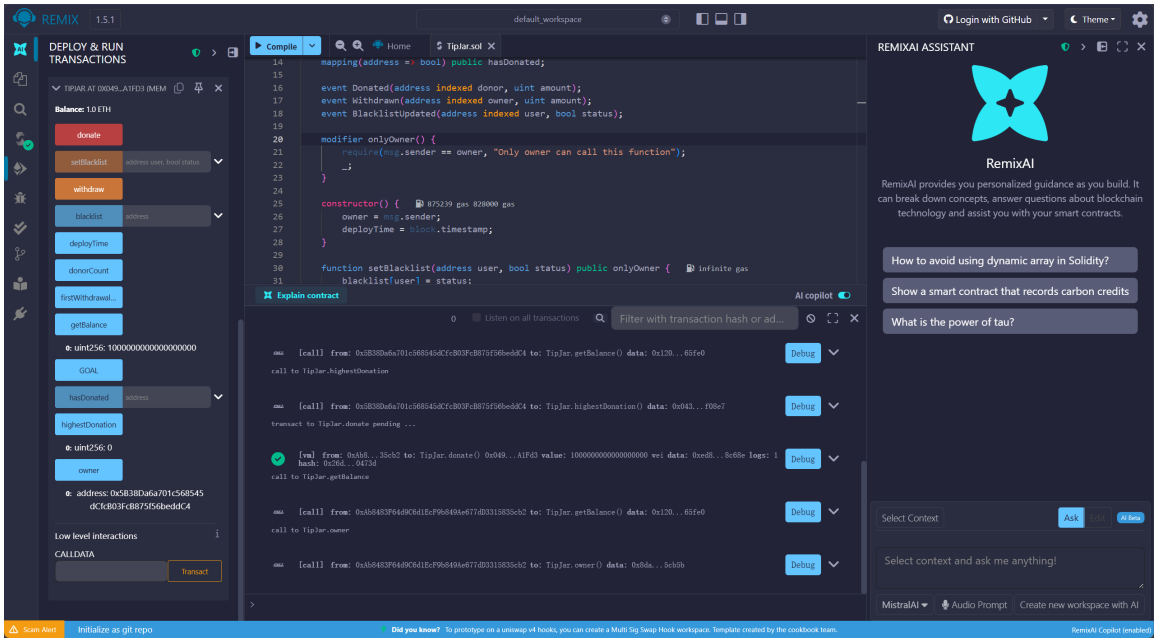

图 12: 冷却期结束后 owner 成功提款并使余额归零

7.5 进阶部分实验现象记录

根据实验截图，最终版本合约测试结果如下：

- 初始 GOAL 查询结果：5 ETH
- 被加入黑名单的地址：0x78731D3Ca6b7E34aC0F824c42a7cC18A495cabaB
- blacklist(address) 查询结果：true
- 黑名单地址尝试捐赠的金额：1 ETH
- 黑名单地址捐赠结果：失败
- 黑名单地址捐赠失败提示信息：Address is blacklisted
- 第一个独立捐赠者地址：0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
- 第二个独立捐赠者地址：0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db
- 重复捐赠测试账户地址：0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
- 最高单笔捐赠金额：2 ETH

- 独立捐赠者数量：3
- 达到目标金额前的余额：3.5 ETH
- 达到目标金额后的余额：5 ETH
- 达到目标金额后是否允许继续捐赠：不允许
- 达到目标后再次捐赠的提示信息：Funding goal reached
- owner 在冷却期内调用 withdraw() 的结果：失败
- 冷却期内调用失败提示信息：Cooling period not over
- owner 在冷却期结束后调用 withdraw() 的结果：成功
- 提款后的合约余额：0 ETH

8 功能验证

8.1 部署成功验证

图13展示了最终版本 TipJar 合约在 Remix VM 中成功部署后的界面。截图中可以看到 Deployed Contracts 面板、已部署的合约实例以及各项初始状态变量。

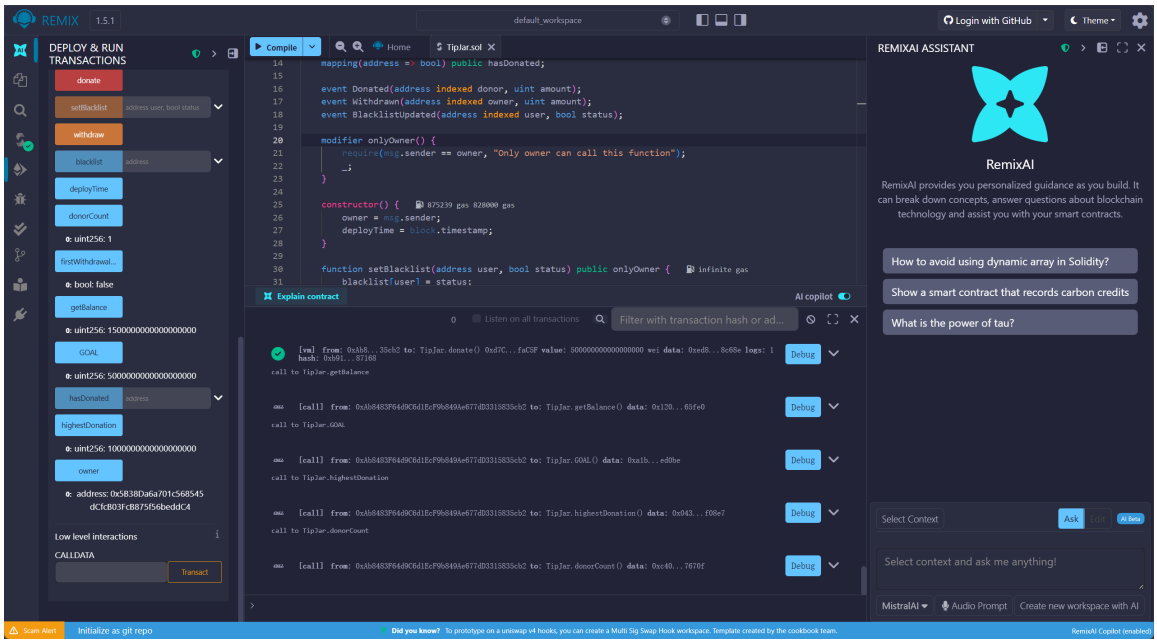

图 13: 最终版本合约部署成功截图

8.2 黑名单机制验证

由图14可见，owner 已将测试地址加入黑名单。随后该地址尝试调用 `donate()` 时，交易被回滚，并返回 `Address is blacklisted` 错误信息，说明黑名单限制已正确生效。

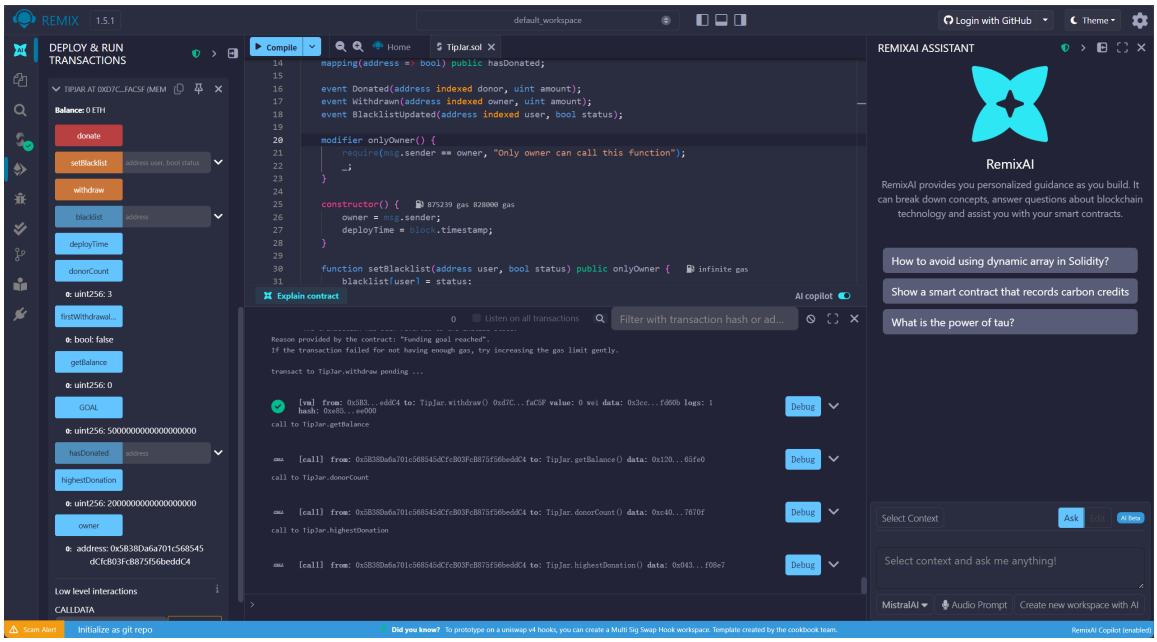

图 14: 黑名单地址尝试捐赠失败

8.3 合约余额增长验证

图15与图16展示了普通账户完成捐赠后，合约余额由 `1 ETH` 增加至 `3.5 ETH` 的过程。该结果证明合约能够正确接收捐赠并更新链上余额。

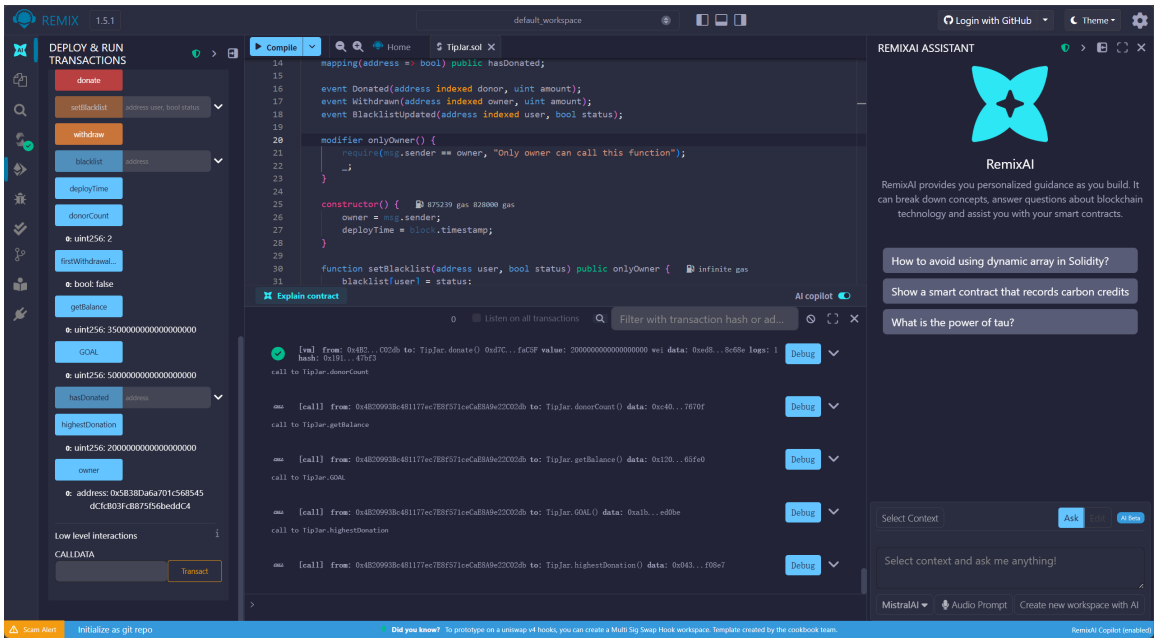

图 15: 首次捐赠后合约余额增长至 1 ETH

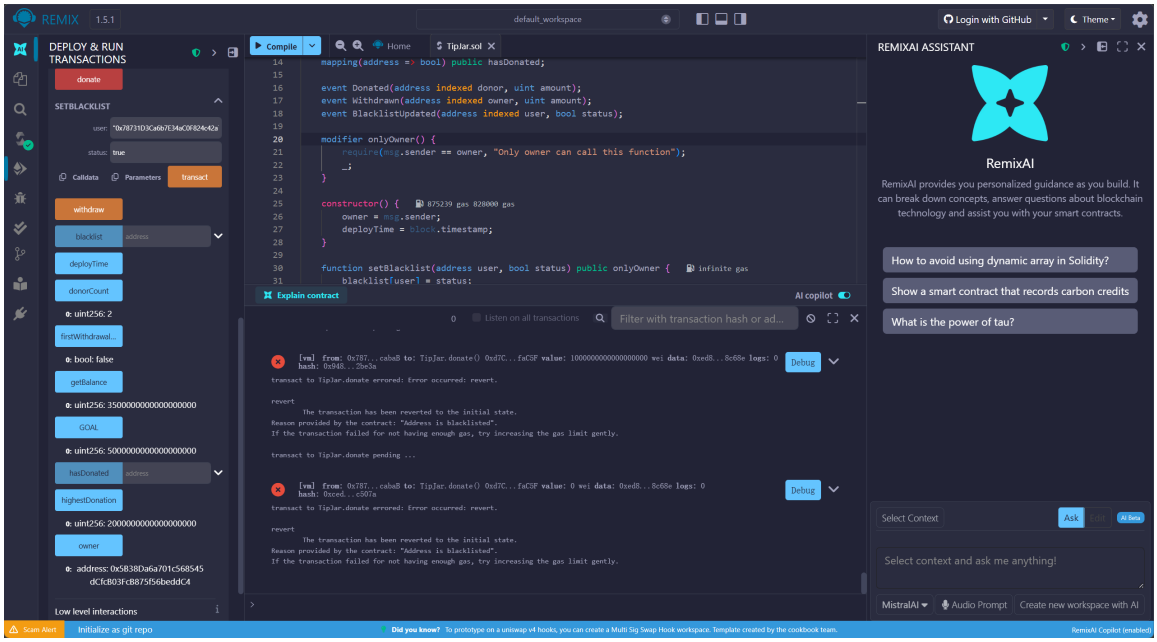

图 16: 后续捐赠后合约余额继续增长至 3.5 ETH

8.4 众筹目标与时间锁机制

图17显示合约余额达到 `GOAL = 5 ETH` 后，再次调用 `donate()` 被拒绝；图18显示首次提款因未超过冷却期而失败；图19显示冷却期结束后 owner 成功提款且余额归零。这些现象共同证明了众筹目标与时间锁机制的正确性。

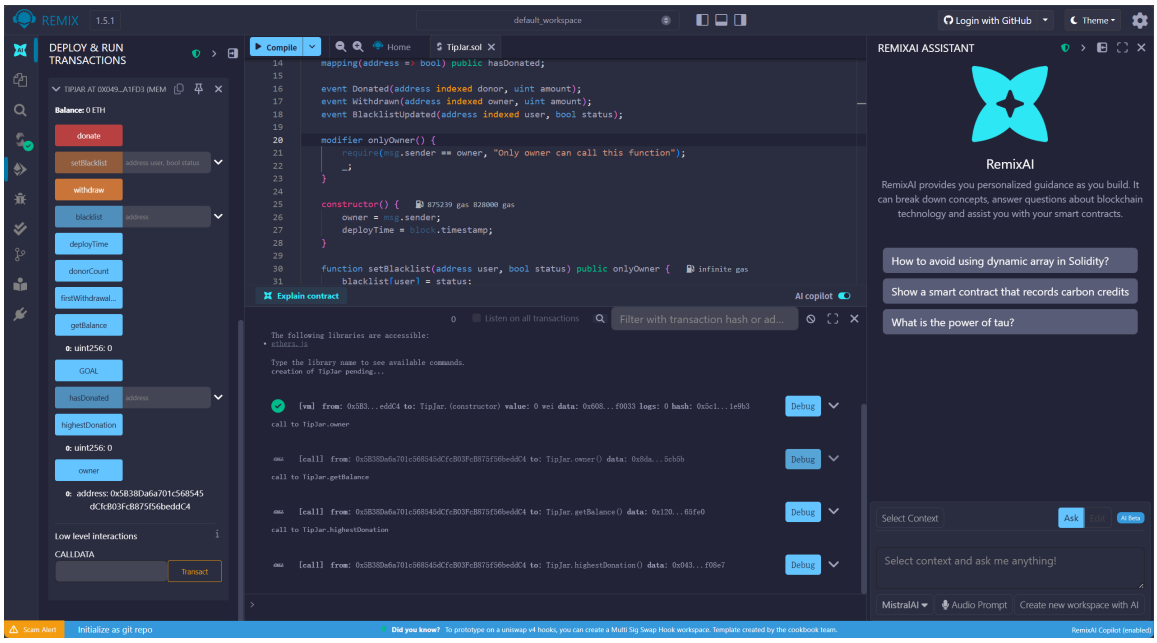

图 17: 达到众筹目标后再次捐赠失败

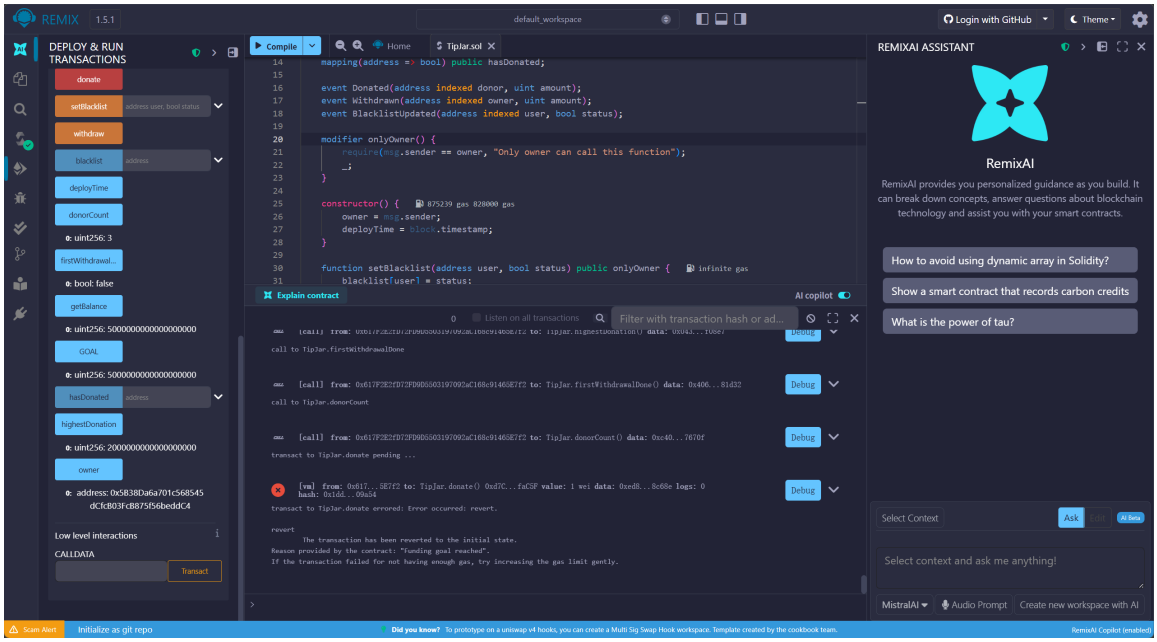

图 18: 冷却期内首次提款失败

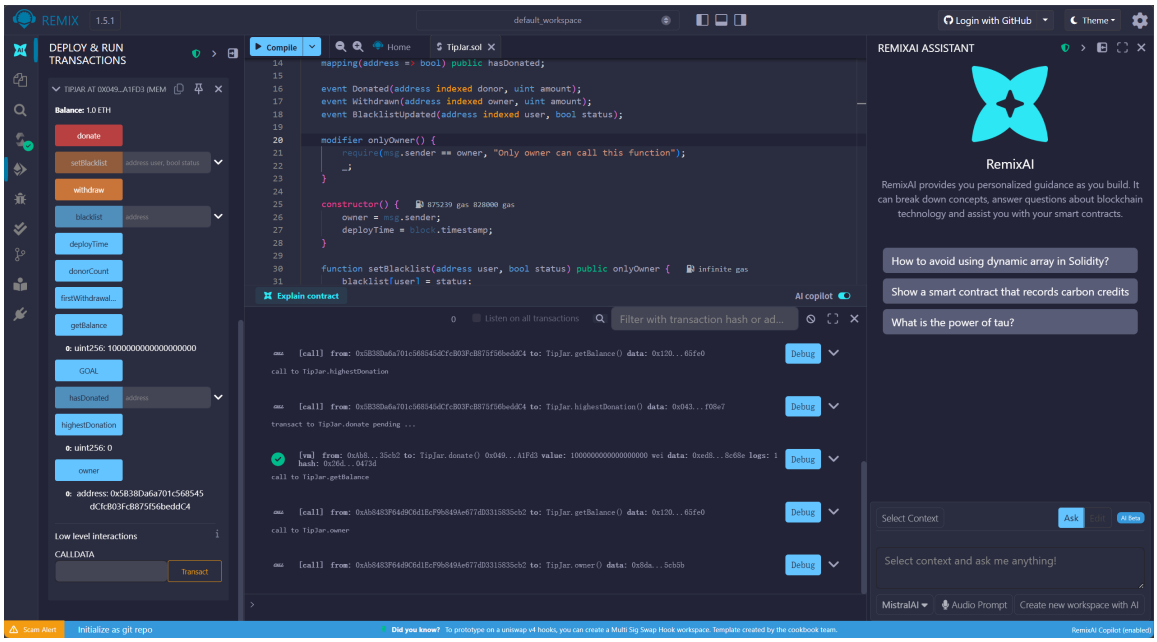

图 19: 冷却期结束后提款成功且余额归零

9 实验结果分析

通过本次实验，可以得到以下结论：

1. 智能合约能够通过代码逻辑自动实现管理员控制、捐款接收、资金限制与数据统计。
2. 以太坊中的 EOA 和合约账户具有不同角色：EOA 负责发起交易，合约账户负责执行代码逻辑。
3. 改变区块链状态的操作需要消耗 Gas，而只读调用一般不需要消耗 Gas。
4. 黑名单、众筹目标和时间锁等机制体现了智能合约在规则自动执行和资金安全约束方面的优势。
5. 通过观察合约状态变化，可以更直观理解区块链系统中的透明性与可验证性。

这里可以加入你自己的分析，例如：

- 哪个功能最容易实现，为什么？
- 哪个功能最容易出错，为什么？
- 你对 `msg.sender`、`msg.value`、`require()` 的理解
- 你对 Remix VM 与真实链部署区别的理解

10 实验总结

本实验通过实现一个去中心化打赏合约，完成了从智能合约编写、编译、部署到交互测试的完整流程。实验中不仅实现了基础的资金接收和提款功能，还通过黑名单、众筹目标、数据统计和时间锁等机制加深了对 Solidity 编程和以太坊运行机制的理解。

通过本实验，我进一步掌握了：

- Remix IDE 的基本操作流程
- Solidity 合约结构与常用语法
- 以太坊账户模型
- 状态改变交易与只读调用的区别
- 智能合约中访问控制与自动执行规则的设计方式

A 附录：最终 Solidity 源代码

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.31;

contract TipJar {
    address public owner;
    uint public constant GOAL = 5 ether;

    uint public highestDonation;
    uint public donorCount;
    uint public deployTime;
    bool public firstWithdrawalDone;

    mapping(address => bool) public blacklist;
    mapping(address => bool) public hasDonated;

    event Donated(address indexed donor, uint amount);
    event Withdrawn(address indexed owner, uint amount);
    event BlacklistUpdated(address indexed user, bool status);

    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner can call this function");
        _;
    }
```

```
    constructor() {
        owner = msg.sender;
        deployTime = block.timestamp;
    }

    function setBlacklist(address user, bool status) public onlyOwner {
        blacklist[user] = status;
        emit BlacklistUpdated(user, status);
    }

    function donate() public payable {
        require(!blacklist[msg.sender], "Address is blacklisted");
        require(msg.value > 0, "Donation must be greater than 0");

        uint previousBalance = address(this).balance - msg.value;

        require(previousBalance < GOAL, "Funding goal reached");
        require(previousBalance + msg.value <= GOAL, "Donation exceeds funding
    goal");

        if (!hasDonated[msg.sender]) {
            hasDonated[msg.sender] = true;
            donorCount++;
        }

        if (msg.value > highestDonation) {
            highestDonation = msg.value;
        }
    }

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }

    function withdraw() public {
        require(msg.sender == owner, "Only owner can withdraw");
        require(address(this).balance > 0, "No funds to withdraw");

        if (!firstWithdrawalDone) {
```

```
            require(block.timestamp >= deployTime + 5 minutes, "Cooling period
    not over");
            firstWithdrawalDone = true;
        }

        uint amount = address(this).balance;
        (bool success, ) = payable(owner).call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```