智能合约实验

版权归杜文亮所有
本作品采用 Creative Commons 署名-非商业性使用-相同方式共享 4.0 国际许可协议授权。如果您重新混合、改变这个材料，或基于该材料进行创作，本版权声明必须原封不动地保留，或以合理的方式进行复制或修改。

1 概述

智能合约是一种运行在区块链上的程序，其代码存储在区块链的特定地址中，是不可更改的。当预定条件满足时智能合约便会自动执行，其数据（状态）同样存储在区块链上。本实验的目标是帮助学生理解这类程序的工作原理以及如何编写简单的智能合约代码，学生将使用所提供的智能合约代码，在 SEED 区块链仿真器上进行实验。本实验涵盖的主题包括：

- 智能合约开发
- Remix IDE
- 部署智能合约，与智能合约交互
- 向智能合约发送资金，以及从智能合约提取资金

实验环境。 本实验已在我们预先构建的 Ubuntu 20.04 虚拟机上进行了测试，学生可以从 SEED 官网上下载。

2 实验设置：启动区块链仿真器

本实验将在 SEED 互联网仿真器（以下简称仿真器）中进行。如果这是你第一次使用仿真器，请认真阅读本节内容，同时建议教师们组织专门的实验课程帮助学生了解和熟悉仿真器的操作。

下载仿真器文件。 请从网页上下载 `Labsetup.zip` 文件，解压缩后会得到仿真器文件。这些仿真器文件存储在 `Labsetup/emulator_*` 文件夹中。对于 AMD64 机器，文件夹名为 `emulator_NN`；对于 Apple 硅芯片机器，则为 `emulator_arm_NN`。数字 `NN` 表示区块链网络上的节点数量。如果虚拟机的内存小于 4GB，建议学生选择较小的版本。

要运行仿真器，我们只需要这些容器文件。这些文件是由 `Labsetup/emulator_code` 文件夹中的 Python 代码生成的。除非你想修改仿真器文件，否则无需运行该代码（需要从 GitHub 安装 SEED Emulator 库才能运行此代码）。希望修改仿真器的老师可以修改 Python 代码并生成自己的仿真器文件。

为了简化操作，在仿真器内部运行的区块链使用的是 Proof-of-Authority （PoA）共识协议，而不是在 MAINET 中使用的 Proof-of-Stake 协议。实验中的活动不依赖于任何特定共识协议。

启动仿真器。 进入 `emulator` 文件夹，并运行以下 docker 命令以构建和启动容器。下面列出的命令是在 SEED VM 上创建的别名。如果您不是使用 SEED VM，可以使用原始命令。

```
$ dcbuild       # 别名为: docker-compose build
$ dcup          # 别名为: docker-compose up
```

我们建议您在提供的 SEED Ubuntu 20.04 虚拟机中运行仿真器，但在安装了 docker 软件的普通 Ubuntu 20.04 操作系统上进行操作也不会有问题。对于较新的操作系统版本，`docker-compose` 命令已被集成到 `docker` 命令中，您可以使用 `"docker compose"` 替代 `docker-compose` 运行它。读者可以从此链接找到 docker 手册。如果您是首次使用容器设置 SEED 实验环境，阅读用户手册非常重要。

所有容器都在后台运行。要对容器执行命令，我们通常需要在该容器上获取一个 shell。首先，我们需要使用 `docker ps` 命令来找到容器的 ID，然后使用 `docker exec` 在该容器上启动一个 shell。我们在 `.bashrc` 文件中为此创建了别名。

```
$ dockps        // 别名为: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // 别名为: docker exec -it <id> /bin/bash

// 以下示例说明如何在 hostC 上获取 shell
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注意: 如果 docker 命令需要容器 ID, 不需要输入完整的 ID,
//       输入前面的一部分就可以了, 只要能唯一匹配一个 ID 就行。
```

如果您在设置实验环境时遇到问题，请阅读手册中的“常见问题”部分以获取解决方案。

停止仿真器。 要停止仿真器，我们只需停止所有容器。我们可以回到运行 `"docker-compose up"` 命令的终端中，输入 `Ctrl-C`。这将停止所有容器但不会删除它们，即容器中的数据仍然保留，并可以通过再次运行 `"docker-compose up"` 来恢复容器的运行。如果我们想要删除它们，则可以运行 `"docker-compose down"` 命令。另一种方法是打开一个不同的终端（但仍处在 `emulator` 文件夹中），直接运行该命令。这将停止并删除所有容器。

```
$ dcdown        # 别名为: docker-compose down
```

EtherView。 我们实现了一个名为 `EtherView` 的简单 Web 应用程序，以显示区块链上的活动。如果想使用这个应用程序，将浏览器（在 VM 中）指向 `http://localhost:5000/`。在 `Blocks` 页面中，您可以看到新创建的区块和最近的交易。如果没有人发送交易，则区块大多是空的，即不包含任何交易。一旦我们开始发送交易，我们应该能够看到它们。用户可以点击区块和交易来查看它们的详细信息。

3 任务 1：使用 Remix 进行智能合约开发

智能合约的开发环境有很多，包括 Hardhat、Remix 和 Truffle。在本实验中，我们将使用 Remix IDE（集成开发环境）来编写、编译和调试 Solidity 代码。Remix IDE 不仅提供了代码编辑功能，还支持智能合约的测试、调试和部署。

3.1 任务 1.a：将 Remix 连接到 SEED 仿真器

Remix IDE 有在线版本和桌面版本，我们将使用在线 IDE，因此不需要进行任何安装，只需要访问这个网址 `https://remix.ethereum.org/` 即可以直接在浏览器中使用 Remix IDE。Remix IDE 还提供了一份详细的使用指南（链接 `https://remix-ide.readthedocs.io/`）。

Remix 可以通过多种方式连接到区块链，如 Remix VM、MetaMask 和外部 HTTP。Remix VM 是一个在浏览器中的运行的沙箱区块链，它模拟了区块链的行为，但并不是一个真实的区块链。在本实验中，我们将 Remix 连接到 SEED 区块链仿真器，这是一个私有的区块链环境，运行的是真正的区块链程序。这种连接可以通过注入式提供者（如 MetaMask）或外部 HTTP 提供者来实现。

在本任务中，我们将使用 MetaMask 作为与 SEED 区块链仿真器交互的工具。首先，我们需要将 MetaMask 连接到 SEED 区块链仿真器，然后配置 Remix，在需要与区块链进行交互时通过 MetaMask 来实现。MetaMask 不仅是连接到区块链的桥梁，也是一个管理账户的钱包，我们需要从一个账户发送交易，而 MetaMask 帮助我们管理这些账户。

在之前的实验中，我们已经在我们的浏览器中安装了 MetaMask 插件，如果你之前没有完成那个实验，你可以参考 `https://github.com/seed-labs/seed-labs/blob/master/manuals/emulator/metamask.md` 中的说明进行安装。恢复账户的助记词短语为：

```
gentle always fun glass foster produce north tail security list example gain
```

设置好 MetaMask 后，打开 Remix IDE，在 Remix IDE 的界面中，找到并点击 `"Deploy & Run Transactions` 菜单，将 `Environment` 设置为 `"Injected Provider - MetaMask"`，如果一切设置正确，你可以在 `Account` 下拉菜单中看到几个显示余额的账户。

3.2 1.b：编写、编译和部署智能合约

在本任务中，我们将使用 Remix IDE 部署一个名为 `Hello.sol` 的简单智能合约，该合约代码在`Labsetup/contract` 目录中。

- 编写代码：进入 Remix 的 `Workspaces` 菜单，会看到几个文件夹，在 `contract` 文件夹中，创建一个名为 `Hello.sol` 的新文件，将提供的 `Hello.sol` 文件的内容复制并粘贴到这个新文件中。
- 编译代码：进入 `"Solidity compiler"` 菜单，编译 `Hello.sol` 程序。
- 部署合约：进入`"Deploy & run transactions"` 菜单，点击 `Deploy` 按钮，MetaMask 会弹出一个窗口并要求你确认交易，如果您没有看到 MetaMask 的弹出窗口，请检查你的 `Environment` 设置以确保已经选择了 MetaMask。成功部署合约后，你将收到来自 MetaMask 的确认信息，接下来，转到 EtherView 界面查看你的部署交易是否已经被添加到区块链中。

3.3 1.c：深入了解

在本任务中，我们将探究智能合约部署过程中的实际机制，与普通的资金交易不同，部署智能合约是通过创建一种特殊的交易来完成的。请你部署一个智能合约，然后使用 EtherView 获取交易详情，将此交易与普通的资金转移交易进行对比分析，并描述它们之间的主要区别。在智能合约的部署交易中，你将看到数据字段中包含一些内容，请说明这些内容是什么，并提供相应证据支撑你的结论。

注意： 在本任务中，你可能需要获取已编译合约的字节码，这一信息可以从 Remix IDE 中获得。在 `"Solidity Compiler"` 菜单下，你将看到 `ABI` 和 `Bytecode` 按钮，你也可以前往`Labsetup/contract` 文件夹，并使用提供的 `Makefile` 来编译合约代码，从而获取所需的字节码。

4 任务 2：调用合约函数

在本任务中，我们将了解如何与智能合约交互。我们将使用之前任务中部署的 `Hello` 合约。请进入 `"Deployed Contracts"` 区域，我们能看到刚刚部署的合约的详情，包括它们的地址、当前余额和合约内定义的函数。

4.1 任务 2.a：通过本地调用函数

如果一个函数被定义为 `"public view"` 类型，它不会修改合约的状态，因此可以通过本地调用而非交易来执行，这种类型的交互是免费的。Remix 为这类函数提供了一个按钮。Remix 为 `public` 变量也提供了一个按钮，因为它给每个 `public` 变量创建了一个 `"public view"` 类型的 getter 函数。我们可以使用这些按钮与合约交互。请调用合约的所有 `"public view"` 函数，并报告你的观察结果。

4.2 任务 2.b：通过交易调用函数

修改合约状态的函数必须通过交易来调用。在 Remix 中，这类函数对应的按钮会使用不同的颜色。请调用 `increaseCounter()` 函数，并报告你的观察结果。同时，使用 EtherView 展示由此调用生成的交易详情。

接下来，在智能合约中添加一个名为 `decreaseCounter(uint)` 的新函数，编译并部署这个函数，与该函数交互并展示你的结果。

4.3 任务 2.c：深入了解

当我们发送交易来调用函数时，到底都发生了什么？请查看此类交易的 `to` 和 `data` 字段，并解释它们的含义。为了调用一个函数，我们需要提供函数的名称及其参数，这些信息被编码在 `data` 字段中。

请调用 `increaseCounter()` 函数，并使用 EtherView 获取交易详情，然后验证 `data` 字段是否确实遵循以下编码规则：

- 函数选择器：函数签名哈希值的前 4 个字节
- 参数：根据 ABI 定义的类型进行编码（每个参数 32 字节）

下面的 Python 脚本能帮助你计算函数签名的哈希值。

```
from web3 import Web3

hash = Web3.sha3(text="<function signature>")
print(hash.hex())
```

深入了解了交易调用函数的内部机制后，我们将尝试不通过 Remix 提供的按钮（这些按钮会自动构建数据字段内容）来直接通过数据字段调用函数。在 Remix IDE 的 `"Deployed contract` 区域底部，有一个名为 `"Low level interactions"` 的区域。在此区域中，我们放到 `CALLDATA` 字段的任何内容都将被直接放入交易的 `data` 字段中。请采用这种方法来调用 `increaseCounter()` 函数，并确认调用是否成功执行。

4.4 任务 2.d：发出事件

智能合约可以发出事件，以通知区块链上发生的事件。应用程序可以监听这些事件并在它们发生时采取行动。在 `Hello.sol` 中，我们声明了一个名为 `MyMessage` 的事件，然后在 `sendMessage()` 函数中，我们使用 `emit` 关键字来生成调用事件的消息。

```
event MyMessage(address indexed _from, uint256 _value);

function sendMessage() public returns (uint256) {
   emit MyMessage(msg.sender, counter);
   return counter;
}
```

生成的消息会被记录在交易收据的日志字段（log）中，应用程序可以通过监控这个字段来获取通知。Remix 提供了一个界面来显示日志字段的内容，请查看该字段，并确认其内容是否符合你的预期。

5 任务 3：向合约发送资金

在本任务中，我学习如何向智能合约发送资金。为此，我们将使用一个名为 `EtherFaucet.sol` 的合约，该合约的源代码存放在 `Labsetup/contract` 目录下。请将 Solidity 源代码复制并粘贴到 Remix IDE 中，进行编译，然后将其部署到 SEED 区块链上。

当我们向智能合约发送资金时，如果合约中存在 `payable` 类型的函数，这些函数会被自动调用，不同的函数根据特定的条件被触发，图 1 详细描述了调用每个函数的条件。在本任务中，我们将针对每种可能的情况展开实验。

5.1 任务 3.a：直接向合约地址发送资金

用户可以直接向合约地址发送资金，这是通过常规交易完成的，在这种情况下，我们并不需要在交易中指定任何函数。然而，当合约接收到资金时，其某个函数将会被自动调用。

请使用 MetaMask 向 `EtherFaucet` 合约发送一些资金，确保使用合约的地址。一旦交易被确认，请在 Remix 上检查智能合约的余额。`amount` 变量的值应该与合约的余额相匹配（单位可能不相同）。

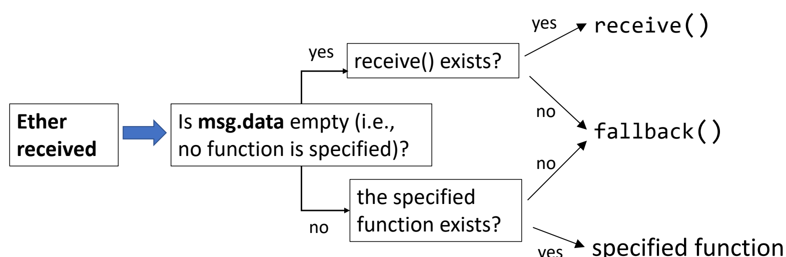

图 1: 但发送资金到智能合约时，哪个 payable 函数会被调用

请进行以下实验：

- 检查合约的 `count` 变量，以验证调用的函数是否与图 1 中描述的一致。
- 从合约中删除 `receive()` 函数，查看 Remix 将给出什么警告。

5.2 任务 3.b：向 payable 函数发送资金

我们可以在调用 payable 函数的同时向智能合约发送资金，我们将通过 Remix 执行这一操作，并调用 `donateEther()` 函数来向合约转账。在进行交易时，你需要在交易的 `value` 字段中指定要发送到合约的金额，这个金额会显示在 `Deploy` 按钮上方。被调用的函数可以通过 `msg.value` 获取资金金额。在调用函数之后，请检查合约的余额、`amount` 变量和 `donationCount` 变量的值是否符合预期。

```
function donateEther() external payable {
    amount += msg.value;
    donationCount += 1;
}
```

5.3 任务 3.c：向非 payable 函数发送资金

能够接收资金的函数必须被标记为 `payable`。请尝试从 `donateEther()` 函数中删除 `payable` 关键字，并查看 Remix 给出的警告信息。此外，我们还提供了一个名为 `donateEtherWrong()` 的函数，该函数没有被标记为 `payable`。请尝试通过调用此函数向合约发送一些资金，尽管你可能会收到一些警告，但请忽略这些警告并发送交易。这样做会发生什么？资金转移是否会成功？如果资金转移不成功，发送者会损失资金吗？

```
function donateEtherWrong() external {
    donationCount += 1;
}
```

5.4 任务 3.d：向不存在的函数发送资金

在本任务中，我们将在调用函数的同时向合约发送资金。不过，我们会故意调用一个并不存在于合约中的函数，看看实际会发生什么。

为了调用这个不存在的函数，我们需要直接设置交易的 `data` 字段，正如我们在4.3 中所做的，我们可以使用 Remix 的 `"Low level interactions"` 直接设置 `data` 字段。我们尝试调用一个名为 `foo()` 的在合约中不存在的函数，并在此过程中向合约发送一些资金。请展示这一过程的结果，资金发送会成功吗？哪个函数会被实际调用？等等。

6 任务 4：从合约发送资金

在上一个任务中，我们探讨了如何向合约发送资金。现在，我们将学习如何从合约中向收款人发送资金。在 Solidity 中，从合约发送以太币有三种不同的方法，包括 `send`、`transfer` 和 `call`，这三种方法都被 Solidity 编译器转换成 CALL 指令码，因此它们共享相同的内部执行机制，但它们之间还是存在一些差异。由于以太坊区块链的持续演变，互联网上关于哪种方法更优的讨论一直存在争议，文档说明也并不一致。

这些方法的主要区别在于它们转发给收款方的 gas 限制。当发起一笔交易时，最初的发送者会设置一个 gas 限制。当此交易调用合约 A，触发合约 A 向收款方发送以太币时，如果收款方也是一个合约（比如合约 B），那么合约 B 的一些代码将被执行。合约 B 的执行需要消耗 gas，它能够获得多少 gas 取决于合约 A 给合约 B 转发了多少 gas。

`send` 和 `transfer` 方法都设定了固定的 2300 gas 限制，用于转发给收款方。鉴于这一限制，收款方合约 B 能够执行的操作相对有限。设置此限制的主要目的是为了防范重入攻击。相比之下，`call` 方法默认不施加此类限制，它会转发原始发送者设定的所有 gas，我们也可以在调用 `call` 方法时自行设定 gas 限制。

由于区块链的不断演进，代码执行所需的 gas 成本可能会发生变化，2300 gas 限制在未来可能不再足够。这种变化可能会导致某些智能合约应用无法正常运行。因此，有些人建议避免使用 `send` 和 `transfer` 方法，转而使用 `call` 方法。尽管 `call` 方法容易受到重入攻击，但有多种方法可以防止这类攻击。实际上，在伊斯坦布尔硬分叉之后，`send` 和 `transfer` 方法已经逐渐被弃用。

任务：向外部拥有的账户发送资金。 我们在 `EtherFaucet` 合约中使用这三种方法实现了发送资金的功能。请使用 Remix 分别调用每种方法，并检验你是否能够从合约中接收到以太币。请注意，`_amount` 参数是以 wei 为单位的，因此如果想要获得一个以太币，我们应该使用 `1e18` 作为转账金额。

```
function getEtherViaCall(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    (bool success, ) = to.call{value: _amount}("");
    require(success, "Failure: Ether not sent");
}

function getEtherViaSend(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    bool success = to.send(_amount);
    require(success, "Failure: Ether not sent");
}
```

```
function getEtherViaTransfer(uint256 _amount) external payable {
    address payable to = payable(msg.sender);
    amount -= _amount;
    to.transfer(_amount);
}
```

请注意，在提供的示例代码中，合约允许任何人无限制地提取任何数量的以太币，这显然是不安全的。在实际的智能合约中，必须实施访问控制机制，确保只有经过授权的账户才能够从合约中接收资金。此外，为了保持代码的简洁性，我们没有在示例中加入任何防御重入攻击的措施。我们有一个专门针对重入攻击的 SEED 实验。

7 任务 5：调用另一个合约

在智能合约的开发中，一个合约可以调用另一个合约的函数。在本任务中，我们将探索这种合约间调用的机制。我们提供了一个名为 `Caller.sol` 的智能合约，可以在 `Labsetup/contract` 目录下找到。我们将使用它来调用任务 1 中部署的 `Hello.sol` 合约的函数。

代码示例如下，它包括一个从 `Hello` 合约派生的 `interface`，通过它，我们可以创建一个合约对象，并使用 `interface` 中的 API 与 `Hello` 合约进行交互。我们提供了两个函数调用示例，它们之间唯一的区别是，一个示例只执行函数调用，而另一个在调用时还向 `Hello` 合约发送资金。

```
function invokeHello(address addr, uint _val) public returns (uint) {
   Hello c = Hello(addr);
   uint256 v =  c.increaseCounter(_val);

   emit ReturnValue(msg.sender, v);
   return v;
}

function invokeHello2(address addr, uint _val) public onlyOwner
                                              returns (uint)
{
   Hello c = Hello(addr);
   uint256 v =  c.increaseCounter2{value: 1 ether}(_val);
                                            ↖ 发送资金
   emit ReturnValue(msg.sender, v);
   return v;
}

interface Hello {
    function sayHello() external pure returns (string memory);
    function getResult(uint a, uint b) external view returns (uint);
    function increaseCounter(uint k) external returns (uint);
    function increaseCounter2(uint k) external payable returns (uint);
```

```
    function getCounter() external view returns (uint);
    function sendMessage() external returns (uint256);
}
```

任务。 请部署 `Caller` 合约，并调用其 `invokeHello()` 和 `invokeHello2()` 函数。请描述您的观察并解释您所看到的现象。

8 提交

你需要提交一个详细的，带有截图的实验报告来描述你做了什么以及你观察到了什么。你还需要对有趣的或是令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。简单地附上代码而不作任何解释将不会得到学分。