

华 中 科 技 大 学

网络空间安全学院

本科：《区块链技术与应用》实验报告

姓 名_____

班 级_____

学 号_____

联系方式_____

分 数_____

评 分 人_____

实验报告及代码和设计评分细则

评 分 项 目		满分	得分	备注
Fabric 实验	区块链网络搭建	7.5		
	安装链码并解释代码	7.5		
	调用链码，进行投票	5		
	调用链码，查看投票结果	5		
	操作流畅度	10		
Ethereum 实验	私有链搭建情况	10		
	重入攻击复现情况	10		
	重入攻击原理解释	5		
	操作流畅度	10		
文档格式（段落、行间距、缩进、图表、编号等）		10		
感想（含思政）		10		
意见和建议		10		
实验报告总分		100		
教师签名			日 期	

目 录

FABRIC 实验	1
一、 实验概述	1
1.1 实验名称	1
1.2 实验目的	1
1.3 实验环境	1
1.4 实验内容	1
1.5 实验要求	2
二、 实验过程	3
2.1 任务1 实验步骤及结果	3
2.2 任务2 实验步骤及结果	4
2.3 代码实现	9
三、 实验分析	11
3.1 遇到的问题	11
3.2 针对问题采取的解决方法	11
3.3 设计方案存在的不足	11
四、 实验总结	12
4.1 实验感想	12
4.2 意见和建议	12
ETHEREUM 实验	13
一、 实验概述	13
1.1 实验名称	13
1.2 实验目的	13
1.3 实验环境	13
1.4 实验内容	13
1.5 实验要求	14
二、 实验过程	15
2.1 任务1 实验步骤及结果	15
2.2 任务2 实验步骤及结果	19
2.3 代码实现	21
三、 实验分析	23
3.1 遇到的问题	23
3.2 针对问题采取的解决方法	23
3.3 设计方案存在的不足	23
四、 实验总结	24
4.1 实验感想	24
4.2 意见和建议	24

Fabric 实验

一、 实验概述

1.1 实验名称

Fabric 实验。

1.2 实验目的

本实验旨在学习和掌握如何使用 Hyperledger Fabric v2.x 进行区块链网络的搭建与操作。通过实验，我们将完成 Fabric 测试网络的构建、智能合约的开发与部署，以及通过命令行和应用程序调用智能合约，以实现区块链中投票和计票的功能。这有助于熟悉区块链技术的应用，并了解其在去中心化网络中的工作流程。

1.3 实验环境

操作系统：Ubuntu 22.04.4 LTS

编程语言：Go（用于智能合约和应用程序开发）

其他：

- Git 版本控制工具
- cURL 用于数据传输
- Docker 和 Docker Compose 用于容器化环境
- JQ 用于处理 JSON 格式数据
- Hyperledger Fabric v2.5.4 和其相关组件（Fabric CA v1.5.12）
- JDK 用于 Java 开发需要
- Maven 项目管理工具，对 Java 项目进行构建、依赖管理

1.4 实验内容

(1) **环境搭建：**安装和配置 Fabric 所需的组件，包括 Git、cURL、Docker、Go 等。

- (2) **Fabric 测试网络搭建：**使用 `network.sh` 脚本启动包含排序节点、peer 节点和 CouchDB 状态数据库的 Fabric 网络，并创建一个名为 "mychannel" 的通道。
- (3) **智能合约开发：**设计并实现一个投票智能合约，包含 `VoteUser`、`GetUserVote` 和 `GetAllVotes` 等功能。
- (4) **智能合约部署：**将开发的智能合约部署到测试网络中。
- (5) **命令行调用：**使用命令行接口调用智能合约，进行投票和查询操作。
- (6) **应用程序调用：**使用 Fabric Gateway 通过应用程序调用智能合约，实现对区块链账本的交互。

1.5 实验要求

- (1) 按照实验任务指导手册中的要求独立完成实验；
- (2) 提交实验设计报告和源代码；实验设计报告必须包括程序流程图，源代码必须加详细注释。
- (3) 实验设计报告需提交纸质档和电子档，源代码需提交电子档。
- (4) 基于自己的实验设计报告，通过实验课的上机试验，将源代码编译成功，运行演示给实验指导教师检查。

二、 实验过程

2.1 任务 1 实验步骤及结果

2.1.1 环境搭建

首先，完成相关环境的搭建，具体的参数如下所示，实验搭建完成可参照图 1-1。

- Ubuntu 22.04 LTS 64 位
- git 2.34.1
- curl 7.81.0
- Docker 24.0.7
- Docker Compose 1.29.2
- Golang 1.18.1
- jq 1.6

```
hyt@hyt-ubuntu:~$ git --version
git version 2.34.1
```

```
hyt@hyt-ubuntu:~$ curl --version
curl 7.81.0 (x86_64-pc-linux-gnu) libcurl/7.81.0 OpenSSL/3.0.2 zlib/1.2.11
brotli/1.0.9 zstd/1.4.8 libidn2/2.3.2 libpsl/0.21.0 (+libidn2/2.3.2) libs
```

```
hyt@hyt-ubuntu:~$ docker version
Client:
Version:      24.0.7
API version:  1.43
Go version:   go1.21.1
Git commit:   24.0.7-0ubuntu2~22.04.1
Built:        Wed Mar 13 20:23:54 2024
OS/Arch:      linux/amd64
Context:      default
```

```
hyt@hyt-ubuntu:~$ docker-compose version
docker-compose version 1.29.2, build unknown
docker-py version: 5.0.3
CPython version: 3.10.12
OpenSSL version: OpenSSL 3.0.2 15 Mar 2022
```

```
hyt@hyt-ubuntu:~$ go version
go version go1.18.1 linux/amd64
```

```
hyt@hyt-ubuntu:~$ jq --version
jq-1.6
```

图 1 - 1 环境搭建完成截图

安装 Fabric 组件包括 Docker 镜像、二进制文件和示例代码。将 Fabric 的可执行文件路径加入环境变量，确保在任何位置都可以调用 `peer`、`orderer` 等命令。

2.1.2 搭建 Fabric 测试网络

\$./network.sh up createChannel -ca -c mychannel -s couchdb 命令启动了包含一个排序节点、两个 Peer 节点、三个 CA 以及 CouchDB 作为状态数据库的测试网络。CouchDB 提供了支持复杂查询的文档存储，这对于某些智能合约的查询操作更有利。并且可以通过\$ docker ps -a 命令用于验证所有 Fabric 容器是否成功启动，包括排序节点、Peer 节点和 CA 容器，如下图 1-2 所示。确保网络组件运行正常是下一步部署智能合约的基础。

```
Anchor peer set for org 'Org2MSP' on channel 'mychannel'
Channel 'mychannel' joined
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
7ffb6ea9824b	hyperledger/fabric-peer:latest	"peer node start"	About a minute ago	Up About a minute	0.0.0.0:9051->9051/tcp, :::9051->9051/tcp, 7051/tcp, 0.0.0.0:9445->9445/tcp, :::9445->9445/tcp
8651f56e1827	hyperledger/fabric-peer:latest	"peer node start"	About a minute ago	Up About a minute	0.0.0.0:7051->7051/tcp, :::7051->7051/tcp, 0.0.0.0:9444->9444/tcp, :::9444->9444/tcp
ac1c7a91dfb0	couchdb:3.3.3	"tini -- /docker-ent..."	2 minutes ago	Up About a minute	4369/tcp, 9100/tcp, 0.0.0.0:5984->5984/tcp, :::5984->5984/tcp
5315c09f4b8d	couchdb:3.3.3	"tini -- /docker-ent..."	2 minutes ago	Up About a minute	4369/tcp, 9100/tcp, 0.0.0.0:7984->5984/tcp, :::7984->5984/tcp
218cf69073ba	hyperledger/fabric-orderer:latest	"orderer"	2 minutes ago	Up About a minute	0.0.0.0:7050->7050/tcp, :::7050->7050/tcp, 0.0.0.0:7053->7053/tcp, :::7053->7053/tcp, 0.0.0.0:9443->9443/tcp, :::9443->9443/tcp
c9fea80c37fb	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	4 minutes ago	Up 4 minutes	0.0.0.0:8054->8054/tcp, :::8054->8054/tcp, 7054/tcp, 0.0.0.0:18054->18054/tcp, :::18054->18054/tcp
bb8b513a42ff	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	4 minutes ago	Up 4 minutes	0.0.0.0:7054->7054/tcp, :::7054->7054/tcp, 0.0.0.0:17054->17054/tcp, :::17054->17054/tcp
3c00715943c4	hyperledger/fabric-ca:latest	"sh -c 'fabric-ca-se..."	4 minutes ago	Up 4 minutes	0.0.0.0:9054->9054/tcp, :::9054->9054/tcp, 7054/tcp, 0.0.0.0:19054->19054/tcp, :::19054->19054/tcp

图 1 - 2 Fabric 测试网络组成截图

2.2 任务 2 实验步骤及结果

2.2.1 开发和部署智能合约

首先对投票智能合约进行设计，其主要包含以下功能：VoteUser(username):给指定用户投一票；GetUserVote(username): 查询指定用户的票数；GetAllVotes(): 查询所有用户的票数。这些功能涵盖了记录和查询投票数据的基本操作。实现这些功能有助于理解智能合约如何在链上记录交易和数据。然后进行智能合约部署，执行命令\$./network.sh deployCC -ccl go -ccp ./HUST-blockchain-exp/vote-smartcontract/ -ccn vote 此步骤将投票合约部署到指定的通道上，设置了智能合约的名称和路径，如下图所示。确保链码成功部署后，网络中的 Peer 节点可以执行该合约。如图 1-3 所示。


```
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network$ ./network.sh deployCC -ccl go -ccp ./HUST-blockchain-exp/vote-smartcontract/ -ccn vote
Using docker and docker-compose
deploying chaincode on channel 'mychannel'
executing with the following
- CHANNEL_NAME: mychannel
- CC_NAME: vote
- CC_SRC_PATH: ./HUST-blockchain-exp/vote-smartcontract/
- CC_SRC_LANGUAGE: go
- CC_VERSION: 1.0
- CC_SEQUENCE: auto
- CC_END_POLICY: NA
- CC_COLL_CONFIG: NA
- CC_INIT_FCN: NA
- DELAY: 3
- MAX_RETRY: 5
- VERBOSE: false
executing with the following
- CC_NAME: vote
- CC_SRC_PATH: ./HUST-blockchain-exp/vote-smartcontract/
- CC_SRC_LANGUAGE: go
- CC_VERSION: 1.0
Vendoring Go dependencies at ./HUST-blockchain-exp/vote-smartcontract/

Attempting to Query committed status on peer0.org2, Retry after 3 seconds.
+ peer lifecycle chaincode querycommitted --channelID mychannel --name vote
+ res=0
Committed chaincode definition for chaincode 'vote' on channel 'mychannel'
:
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vsc
cc, Approvals: [Org1MSP: true, Org2MSP: true]
Query chaincode definition successful on peer0.org2 on channel 'mychannel'
```

图 1 - 3 部署智能合约成功截图

2.2.2 调用智能合约

(1) 命令行调用

首先调用智能合约进行投票，调用智能合约的 `VoteUser` 方法，为用户 `username1` 投票，如图 1-4 所示。通过指定 `peerAddresses` 和 `tlsRootCertFiles`，确保投票请求能够成功传递到所有 `Peer` 节点。然后查询用户的投票情况，验证用户 `username1` 的票数，确保调用 `VoteUser` 方法后数据在账本上得到正确更新。同时也可以查询所有用户的投票情况，以检查所有用户的票数，验证智能合约在区块链账本中记录和存储数据的完整性。

```
2024-11-03 17:55:02.101 CST 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result: status:200 payload:"{\"id\":0,\"username\":\"username1\",\"votes\":1}"
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network$ peer chaincode query -C mychannel -n vote -c '{"Args":["GetUserVote", "username1"]}'
{"id":0,"username":"username1","votes":1}
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network$ peer chaincode query -C mychannel -n vote -c '{"Args":["GetAllVotes"]}'
[{"id":0,"username":"username1","votes":1}]
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network$
```

图 1 - 4 命令行调用智能合约截图

(2) 应用程序调用

由于整个围绕 vote.app 中的 main.go 程序展开，我对代码进行的更加具体的了解和分析。首先，创建 gRPC 连接，此步骤通过 newGrpcConnection() 函数创建一个 gRPC 连接，以与 Hyperledger Fabric 网关服务器通信，该连接用于在应用程序和区块链网络之间进行远程过程调用（RPC），确保数据传输安全且可靠。使用 grpc.WithTransportCredentials() 来实现安全连接，通过传递 TLS 证书和 peerEndpoint 来验证和加密连接。newIdentity() 函数生成一个客户端身份（identity.X509Identity），基于 X.509 证书，用于标识应用程序在网络中的身份，此身份在 Hyperledger Fabric 网络中验证和授权交易请求。newSign() 函数使用私钥创建签名函数，以对交易进行数字签名，确保交易在网络中传输时具有完整性和不可抵赖性，此功能通过加载存储在 keyPath 目录中的私钥文件实现。然后，连接到 Fabric 网关，此步骤使用客户端身份、签名函数和 gRPC 连接建立与 Fabric 网关的连接。通过 client.Connect()，应用程序与区块链网络交互来提交和查询交易，超时设置确保在网络延迟时应用程序不会无限等待。接着，获取网络和智能合约实例，GetNetwork() 函数用于指定要访问的通道，而 GetContract() 函数获取指定链码的实例，此步骤是调用链码函数的前提。

同时，实现用户交互逻辑，该循环提供了用户界面，允许用户选择不同的功能来调用链码，包括查询所有用户票数、给用户投票、查询单个用户票数等，具体的使用方式和运行结果如下图所示。每个选项对应不同的链码函数调用：使用 EvaluateTransaction() 执行查询交易，获取所有用户的投票结果；使用 SubmitTransaction() 提交投票交易，将数据写入账本；也可执行查询交易，获取指定用户的票数。最后，格式化 JSON 输出，格式化 JSON 数据以提高可读性，便于用户理解查询结果。

```
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network/HUST-blockchain-examples/vote-app$ go run main.go
2024/11/03 18:07:07 ===== application-golang starts =====
2024/11/03 18:07:07 Please input your choice:
2024/11/03 18:07:07 1: Get all users' votes
2024/11/03 18:07:07 2: Vote for user
2024/11/03 18:07:07 3: Query a user's vote by username
2024/11/03 18:07:07 9: Quit
1
[
  {
    "id": 0,
    "username": "username1",
    "votes": 1
  }
]

2
2024/11/03 18:08:13 Please enter your username you want to vote:
username1

3
2024/11/03 18:10:10 Please enter your username you want to query:
username1
{
  "id": 0,
  "username": "username1",
  "votes": 2
}

9
2024/11/03 18:10:19 ===== application-golang ends =====
```

图 1 - 5 应用程序调用智能合约截图

综上所述，通过以上步骤，成功搭建了一个 Hyperledger Fabric 区块链测试网络并部署了投票智能合约。命令行调用投票和查询功能均成功执行，表明网络组件和智能合约的部署及操作正确无误。

2.2.3 调用智能合约——Java 实现

首先 Java 开发需要安装 JDK 和 Maven，随即创建 Maven 项目，在 pom.xml 中配置 Fabric Gateway Java SDK，通过 mvn clean install 命令，更新并下载所有依赖。然后，创建项目目录结构，创建连接配置文件 connection.json，此文件用于定义 Fabric 网络结构，包括通道、Peer 节点和 CA 的信息，可以使用 Fabric 网络的 YAML 文件转换为 JSON 格式，或者使用网络提供的配置。并准备钱包目录用于存储应用程序用户的身份(appUser)。可以使用 Fabric SDK 提供的工具或命令生成该身份，并将相关证书和私钥存放到 wallet 目录中。接着，使用

Hyperledger Fabric 提供的 Fabric CA CLI 注册并获取用户身份，将证书和私钥文件存储在 wallet 目录中，将用户的 X.509 证书和私钥文件移动到 wallet 目录中，确保身份命名一致，然后在 Main.java 中编写代码以从 wallet 中加载身份并用于网关连接。这样，Java 开发环境、SDK、项目结构和所需的配置文件均已准备就绪，可用于开发和运行 Hyperledger Fabric 的应用程序，如图 1-6 所示。

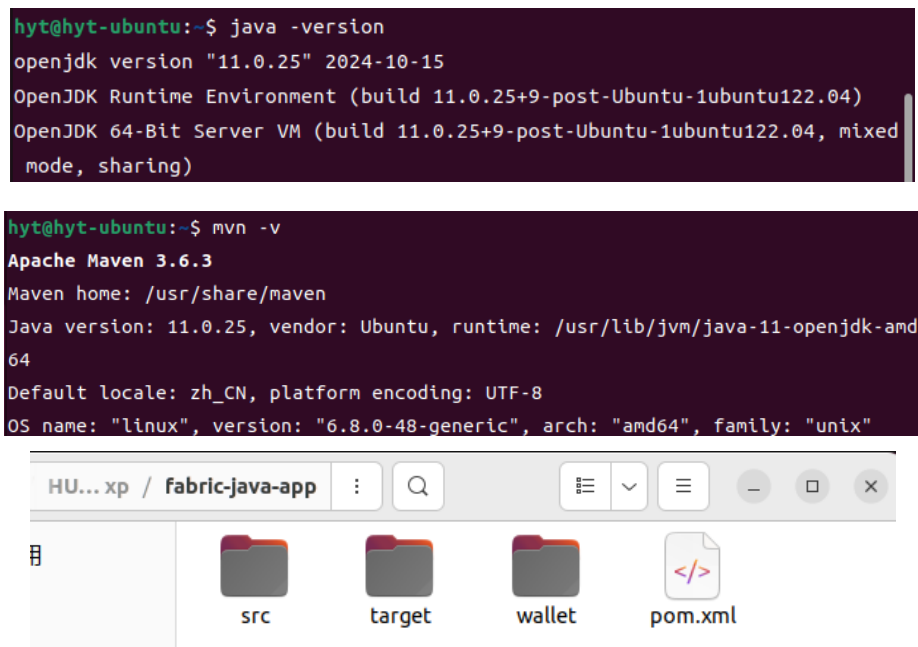


图 1 - 6 环境搭建和相关程序准备截图

接下来我们展示一下编译运行的步骤，在项目根目录下通过 `$ mvn clean compile` 命令来编译代码，如图 1-7，编译成功后，通过和 `$ mvn exec:java -Dexec.mainClass="org.example.Main"` 命令运行应用程序，如图 1-8，可以在下一部分“代码实现”查看详细实现具体代码，Main.java 的代码实现了同 main.go 的相同功能，调用实用过程一致便不再赘述。

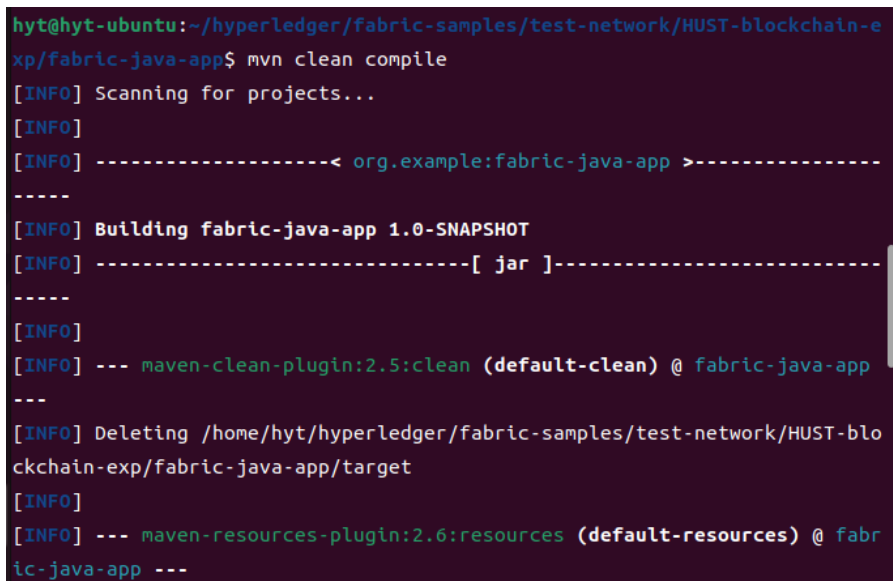


图 1 - 7 应用程序编译过程截图(部分)

```
hyt@hyt-ubuntu:~/hyperledger/fabric-samples/test-network/HUST-blockchain-exp/fabric-java-app$ mvn exec:java -Dexec.mainClass="org.example.Main"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.example:fabric-java-app >-----
-----
[INFO] Building fabric-java-app 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- exec-maven-plugin:3.5.0:java (default-cli) @ fabric-java-app ---
-
===== Java application starts =====
Adding User1 identity to wallet...

... 7 more
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----
```

图 1 - 8 应用程序调用智能合约截图(部分)

2.3 代码实现

实验实现过程中的主要命令和代码，均在上部分“任务 1 实验步骤及结果”中给出，此部分便不再阐述，这部分我将对使用 Java 编程语言实现应用程序调用智能合约的核心代码进行展示和具体逻辑解释，同时我在写代码时相关部分均进行了注释说明。

下面是 Main.java 的代码实现了一个基于 Hyperledger Fabric 的 Java 应用程序，通过 Hyperledger Fabric Gateway SDK 连接到 Fabric 网络，并与链码交互完成基本的投票功能。核心包括：

- **连接网络**：通过 Gateway 类连接到 Fabric 网络。
- **链码交互**：通过 Contract 类调用链码方法，完成交易或查询。
- **身份管理**：通过钱包系统加载和管理用户身份。

(1) 程序入口 **main()**方法，调用 **connect()** 方法，连接到 Hyperledger Fabric 网络，与链码交互，根据用户的选择调用相应的链码函数，支持多轮输入，直到用户选择退出。

调用 **connect()** 方法，该方法返回一个 Gateway 实例，代表与 Fabric 网络的连接。如果连接失败，则打印错误消息并退出程序。使用 **gateway.getNetwork("mychannel")** 获取通道 mychannel，使用 **network.getContract("vote")** 获取链码 vote。然后，循环等待用户输入，根据用户的输入选择不同的操作。最后在程序结束时调用 **gateway.close()**，关闭网关连接释放资源。

```

11 public class Main {
12     public static void main(String[] args) {
13         System.out.println("===== Java application starts =====");
14
15         try {
16             // 配置网关并连接到 Fabric 网络
17             Gateway gateway = connect();
18             if (gateway == null) {
19                 System.err.println("Failed to connect to the gateway.");
20                 return;
21             }
22
23             Network network = gateway.getNetwork("mychannel");
24             Contract contract = network.getContract("vote");
25
26             Scanner scanner = new Scanner(System.in);
27             int choice;
28             do {
29                 System.out.println("Please input your choice:");
30                 System.out.println("1: Get all users' votes");
31                 System.out.println("2: Vote for user");
32                 System.out.println("3: Query a user's vote by username");
33                 System.out.println("9: Quit");
34
35                 choice = scanner.nextInt();
36                 scanner.nextLine(); // Consume newline character

```

(2) 连接网络 `connect()` 方法，该方法用于配置和建立与 Hyperledger Fabric 网络的连接。首先配置钱包路径，如果钱包中不存在 User1 的身份，手动将其加载，并指定网络连接配置文件路径（`connection-org1.yaml`），其中包含了网络拓扑和节点信息，然后创建网关连接，配置网关连接，使用钱包中的身份 User1 和网络配置文件，并开启服务发现（`discovery(true)`），以便动态发现网络中的节点，调用 `connect()` 方法返回连接。

```

97 // 配置网络连接配置文件
98 Path networkConfigPath = Paths.get("/home/hyt/hyperledger/fabric-samples/test-network/organizations/peerOrganizations/org1.example.com/connection-org1.yaml");
99
100 // 创建网关连接
101 Gateway.Builder builder = Gateway.createBuilder()
102     .identity(wallet, "User1")
103     .networkConfig(networkConfigPath)
104     .discovery(true);
105

```

(3) 链码交互逻辑，链码交互通过 Contract 对象完成，主要有以下操作：查询所有投票，调用链码的 `GetAllVote` 方法，返回所有用户的投票信息，使用 `evaluateTransaction()` 执行只读操作，返回结果；为用户投票，提示用户输入要投票的用户名，调用链码的 `VoteUser` 方法，提交投票交易，使用 `submitTransaction()` 提交交易；查询特定用户的投票，提示用户输入要查询的用户名，调用链码的 `GetUserVote` 方法，返回该用户的投票信息。

```

38         switch (choice) {
39             case 1:
40                 byte[] result = contract.evaluateTransaction("GetAllVotes");
41                 System.out.println("Result: " + new String(result));
42                 break;
43             case 2:
44                 System.out.println("Please enter the username you want to vote for:");
45                 String username = scanner.nextLine();
46                 contract.submitTransaction("VoteUser", username);
47                 System.out.println("Transaction successful");
48                 break;
49             case 3:
50                 System.out.println("Please enter the username you want to query:");
51                 username = scanner.nextLine();
52                 result = contract.evaluateTransaction("GetUserVote", username);
53                 System.out.println("Result: " + new String(result));
54                 break;
55             case 9:
56                 System.out.println("===== Java application ends =====");
57                 break;
58             default:
59                 System.out.println("Invalid choice");
60                 break;
61         }

```


三、 实验分析

3.1 遇到的问题

(1) 问题一: 在执行命令 `$./network.sh up createchannel -ca -c mychannel -s couchdb` 时, 出现 “ERROR: Get "https://registry-1.docker.io/v2/": context deadline exceeded” 的报错, 是由于 Docker 在拉取所需的镜像时出现超时错误导致的。同时一定记得重新启动 docker 服务, 否则更改情况无法保存, 并且需要注意运行安装脚本的位置正确, 在 `fabric-samples/test-network` 目录下执行相关命令, 这样才可以确保脚本能够正确引用所需的文件和配置。

(2) 问题二: 在尝试部署名为 "vote" 的链码时: 在打包链码时出现 "error in simulation: failed to execute transaction" 的错误, 在安装链码时出现 "error sending: timeout expired while executing transaction" 的错误等。

(3) 问题三: 在尝试使用 Java 编程语言实现应用程序时, 同样遇到了非常多的版本不兼容的问题, 最终通过查阅大量资料和询问老师进行了解决。

3.2 针对问题采取的解决方法

针对问题一, 首先, 编辑 `/etc/docker/daemon.json` 文件, 在其中添加多个备用的镜像源地址, 如阿里云、DockerHub 等。这样可以增加拉取镜像的成功率, 避免因单一镜像源导致的超时或连接失败。添加完成后, 保存文件并重新加载 Docker 守护进程配置, 然后重启 Docker 服务。

这个解决方案的原理是, Docker 在拉取镜像时会优先使用 `daemon.json` 中配置的镜像源, 如果某个源连接失败或超时, 会自动尝试使用其他可用的备用源。通过这种方式, 可以有效规避由于单一镜像源导致的网络问题, 提高 Docker 镜像拉取的成功率, 从而解决 `./network.sh` 脚本在执行过程中出现的 Docker 镜像拉取失败问题。

针对问题二: 我尝试检查链码代码, 确保其没有问题, 然后检查 Peer 节点的资源使用情况, 确保其有足够的资源, 并且检查网络连接状况, 确保网络稳定之后重新执行命令, 可以正确执行相关部署操作。有时候, 由于一些短暂的资源瓶颈或网络抖动, 链码部署会失败, 重新运行可能会绕过这些临时性问题, 从而成功部署链码。同时 Fabric 网络中的某些缓存或状态可能导致了部署失败, 重新运行可能会清除这些缓存或状态, 从而解决问题。

3.3 设计方案存在的不足

无。

四、 实验总结

4.1 实验感想

通过此次 Fabric 实验,我有机会将区块链的理论知识运用到实践学习中。在搭建 Fabric 框架的过程中,我学会了区块链部署和智能合约的大致流程,这为我未来在网络安全领域的工作奠定了基础。同时,我也意识到,区块链作为一项颠覆性技术,其应用场景广泛,不仅局限于金融领域,在政务、医疗、供应链等多个领域都有广泛应用前景。这种技术的广泛应用必然会对社会产生深远影响,我们作为技术从业者,要积极发挥专业优势,为社会创造更多价值,履行应尽的社会责任。

在实验过程中,我遇到了许多问题,这样的实践经历大大提升了我的问题搜索和解决能力,也培养了耐心和毅力。在配置环境和编写智能合约时,我会遇到各种报错和障碍。尤其是在尝试自己编写 Java 程序来实现应用程序调用投票智能合约时,遇到了大量问题,经常一个问题需要调试很久,推翻重来了好几次。针对这些问题,我需要主动查阅大量的文档和资料,寻找相应的解决方案。这种主动学习的过程不仅增强了我的问题分析和解决能力,也让我更加熟悉了区块链技术的方方面面。同时,我也会及时保存虚拟机的快照,以便在出现问题时可以快速恢复。

通过这次实践,提高了我对相关知识和技术的了解,更加坚定了在网络安全领域为国家和社会做出贡献的决心。未来,我将继续努力学习,提升专业技能,并时刻关注技术发展对社会的影响,为构建更加安全、公平、包容的网络空间贡献自己的力量。

4.2 意见和建议

技术发展日新月异,相关程序在不断更新,辛苦老师对《实验指导手册》也可以进行及时的更新,我们根据之前版本的操作会遇到一些共性的通过搜索和询问均无法解决的问题。最重要的还是非常非常感谢老师们非常耐心和细致地解决我们的问题,并且也很辛苦马上整理了非常完备的新版《实验指导手册》,故我可以顺利完成本次实验。

Ethereum 实验

一、实验概述

1.1 实验名称

Ethereum 实验。

1.2 实验目的

本实验的目的是让学生将从书本中学到的有关区块链的知识应用到实践中。在 Geth 环境下，自行搭建一个私有网络，并掌握以太坊的基本命令，学习编译和调用以太坊智能合约，并最终复现冲入漏洞。

本实验共有两个任务，第一个任务是使其自己尝试如何搭建一个以太坊的多节点私有网络，第二个任务是让学生了解在以太坊的架构下如何去编写、调用智能合约。学生需要了解冲入漏洞的基本原理，并设计代码来复现重入漏洞攻击。

1.3 实验环境

操作系统：Ubuntu 22.04.4 LTS

编程语言：Go 以支持 Geth 的运行和编译，Solidity 编写智能合约

其他：

- Geth（以太坊客户端）搭建私有网络
- Remix 在线合约编译工具，用于编译和部署智能合约
- Docker 和 Docker Compose 用于容器化环境
- JQ 用于处理 JSON 格式数据
- Hyperledger Fabric v2.5.4 和其相关组件（Fabric CA v1.5.12）

1.4 实验内容

- (1) 以太坊环境搭建：配置并运行本地或测试网络。
- (2) 智能合约编写与部署：使用 Solidity 编写简单的智能合约，借助 Truffle 框架进行编

译和部署。

- (3) **区块链交互**：通过 MetaMask 与智能合约进行交互操作，例如调用函数、发送交易等。
- (4) **链上数据查询与验证**：检索链上数据并验证交易结果，以理解数据的不可篡改特性。

1.5 实验要求

- (1) 按照实验任务指导手册中的要求独立完成实验；
- (2) 提交实验设计报告和源代码；实验设计报告必须包括程序流程图，源代码必须加详细注释。
- (3) 实验设计报告需提交纸质档和电子档，源代码需提交电子档。
- (4) 基于自己的实验设计报告，通过实验课的上机试验，将源代码编译成功，运行演示给实验指导教师检查。

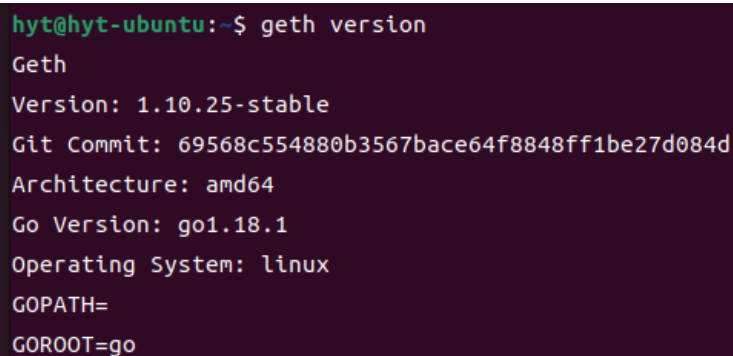
二、实验过程

2.1 任务 1 实验步骤及结果

2.1.1 环境搭建

首先，完成相关环境的搭建，按照实验指导手册中的内容，下载 go 语言环境和 Geth，并将 GOPATH 的路径/usr/local/go/bin 插入到\$HOME/..bashrc 文件中。具体的参数如下所示，实验搭建完成可参照图 2-1。

- Ubuntu 22.04 LTS 64 位
- Geth 1.10.25
- Golang 1.18.1

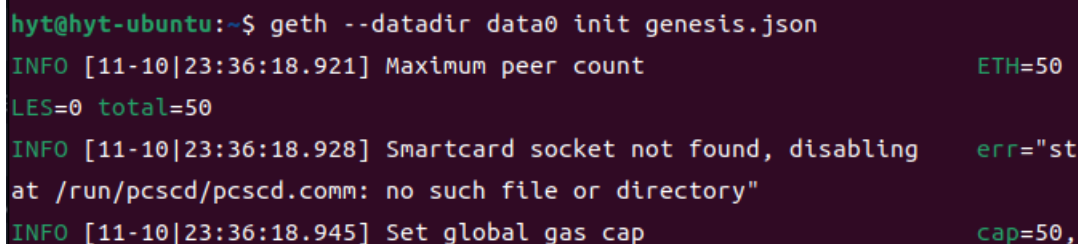


```
hyt@hyt-ubuntu:~$ geth version
Geth
Version: 1.10.25-stable
Git Commit: 69568c554880b3567bace64f8848ff1be27d084d
Architecture: amd64
Go Version: go1.18.1
Operating System: linux
GOPATH=
GOROOT=go
```

图 2 - 1 环境版本截图

①配置 genesis.json 文件

根据参考文档，创建创世区块配置文件 genesis.json 文件，用于定义私有链的基本参数，如难度、gas 限制和 chainID。chainID 用于区分网络，应与主网不同，避免冲突，并且确保设置较低 difficulty 以加快挖矿速度，高 gasLimit 以支持智能合约调用。然后初始化节点，在主目录创建用于存储区块数据的文件夹，并使用 genesis.json 文件初始化创世区块，执行命令后，将在 data0 文件夹内生成与创世区块相关的数据文件。需要为每个节点执行此步骤以完成初始化。



```
hyt@hyt-ubuntu:~$ geth --datadir data0 init genesis.json
INFO [11-10|23:36:18.921] Maximum peer count                ETH=50
LES=0 total=50
INFO [11-10|23:36:18.928] Smartcard socket not found, disabling err="st
at /run/pcscd/pcscd.comm: no such file or directory"
INFO [11-10|23:36:18.945] Set global gas cap                cap=50,
```

图 2 - 2 初始化创世区块截图

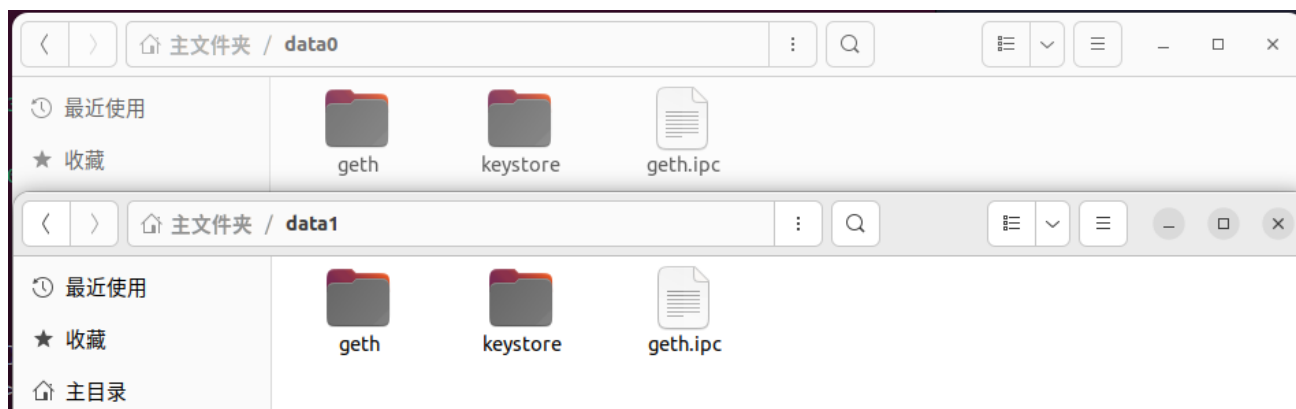


图 2 - 3 节点文件夹截图

②启动私有链节点 data0 及 data1

接着启动私有链节点，通过 `$ geth --http --datadir ~/ethereum/data0 --networkid 1108 console --allow-insecure-unlock --http.corsdomain "*"` 命令启动第一个节点，并允许 HTTP 访问，方便后续智能合约的交互，其中 `--http`，启用 HTTP RPC 服务；`--networkid`，指定网络 ID，确保网络中的节点都设置相同的 ID；`--allow-insecure-unlock`，允许解锁账户，方便测试期间进行转账等操作；`--http.corsdomain`，设置跨域支持，以允许 Web3 工具如 Remix 连接。并且在同一台机器上启动第二个节点时，需要设置不同的端口以避免冲突，如下图所示。

```
hyt@hyt-ubuntu:~$ geth --http --datadir data1 --port 30304 --authrpc.port 8552
--networkid 1108 --http.port 8546 --allow-insecure-unlock --http.corsdomain "*"
INFO [11-11|00:26:39.225] Maximum peer count                ETH=50 LES=0
total=50
INFO [11-11|00:26:39.231] Smartcard socket not found, disabling err="stat /run/pcscd/pcscd.comm: no such file or directory"
INFO [11-11|00:26:39.246] Set global gas cap                 cap=50,000,000
INFO [11-11|00:26:39.252] Allocated trie memory caches       clean=154.00MiB dirty=256.00MiB
INFO [11-11|00:26:39.253] Allocated cache and file handles   database=/home/hyt/data1/geth/chaindata cache=512.00MiB handles=524,288
INFO [11-11|00:26:39.368] Opened ancient database            database=/home/hyt/data1/geth/chaindata/ancient/chain readonly=false
INFO [11-11|00:26:39.369] Writing default main-net genesis block
INFO [11-11|00:26:45.303] Persisted trie from memory database nodes=12356 size=1.78MiB time=233.209769ms gcnodes=0 gcsizes=0.00B gctime=0s livenodes=1 livesize=0.00B
INFO [11-11|00:26:45.441]
```

图 2 - 4 启动私有链节点截图

③进入 data0 对应的文件夹，连接至对应节点

首先，在第一个节点的控制台中使用 `admin.nodeInfo` 查看节点信息，包括 `enode` 地址，然后连接节点使用 `admin.addPeer()` 方法将节点互相连接，由于无法将两个节点连接，所以我们采用在同一个节点上创建多个用户实现转账。

```

hyt@hyt-ubuntu:~$ cd data0
hyt@hyt-ubuntu:~/data0$ geth attach ipc:geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.10.25-stable-69568c55/linux-amd64/go1.18.1
at block: 0 (Thu Jan 01 1970 08:00:00 GMT+0800 (CST))
datadir: /home/hyt/data0
modules: admin:1.0 debug:1.0 engine:1.0 eth:1.0 ethash:1.0 miner:1.0 net:
1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
> personal.newAccount()
Passphrase:
Repeat passphrase:
"0x4b03b00782152d46c31cf9c12a17add76aaa703c"
>

> admin.nodeInfo
{
  enode: "enode://75f58ef9e00396b32ff07fb47cbe5697ee13601cf08774ddc3c9c6ba
12843076dad6c288784c34389d62d0ff1626c1b161bae03875618d475e91fb8e00285eaf@5
8.19.99.130:30303?discport=62206",
  enr: "enr:-Ku4Q0lIy1GJ2fhELuwHNLRS4V7fod0PVfjIyJh4U78FnK-c_WFSJrL0pkATt
sG_Rttc28Evp0qiEp7wLDz4IwTVGaGAZMWuQZrg2V0aMfGhHUN0k2AgmlkgnY0gmlwhDoTY4KJ
c2VjcDI1NmsxoQN19Y754A0Wsy_wf7R8vLaX7hNgHPCHdN3Dyca6EoQwdoRzbmFwwIN0Y3CCdL
-DdWRwgvL-hHVkcDaCd18",
  id: "3530cb55c93ccac5aa38efe05f1cf447c8715d22f884a10acb65659ab478bd40",
  ip: "58.19.99.130",
  listenAddr: "[::]:30303",
  name: "Geth/v1.10.25-stable-69568c55/linux-amd64/go1.18.1",

> admin.addPeer("enode://75f58ef9e00396b32ff07fb47cbe5697ee13601cf08774ddc
3c9c6ba12843076dad6c288784c34389d62d0ff1626c1b161bae03875618d475e91fb8e002
85eaf@58.19.99.130:30303?discport=62206")
true

```

图 2 - 5 连接节点截图

④解锁节点 `data0` 下的账户 0，开始挖矿，并通过其向节点 `data0` 下的账户 1 进行转账连接成功后，可以在控制台测试节点间的挖矿和同步，如图 2-6 所示为为挖到矿的情景。

```
INFO [11-11|01:02:29.520] Successfully sealed new block          number=
17 sealhash=b15b9e..015145 hash=61a9b2..566b0f elapsed=173.168ms
INFO [11-11|01:02:29.521] 🔗 block reached canonical chain          number
=10 hash=5ca55a..b9c788
INFO [11-11|01:02:29.522] ⚡ mined potential block                  number
=17 hash=61a9b2..566b0f
INFO [11-11|01:02:29.523] Commit new sealing work                  number=
18 sealhash=5596d1..1b2f4f uncles=0 txs=0 gas=0 fees=0 elapsed=1.884ms
INFO [11-11|01:02:29.526] Commit new sealing work                  number=
18 sealhash=5596d1..1b2f4f uncles=0 txs=0 gas=0 fees=0 elapsed=5.358ms
```

图 2 - 6 挖到矿情景截图

执行 `miner.start()`，会使用当前的矿工账户进行挖矿并获取代币收益，可以指定参数设置线程数量，然后通过观察区块高度 `eth.blockNumber`，可以验证节点间的数据同步情况。挖矿过程中，每个新生成的区块都会在节点间同步。在节点下添加两个账户，解锁用户并且执行命令进行转账。需要注意的是，不论是进行交易还是发布智能合约，都需要矿工进行挖矿才能在整个区块链网络被确认，整个过程如图 2-7 所示。

```
> eth.accounts
["0x4b03b00782152d46c31cf9c12a17add76aaa703c", "0xaca1445d31ab49f76ca2570c
be21b43941486ace", "0xd606f8c78e9d48472518225f85ca3eace8fc37d8"]

> personal.unlockAccount(eth.accounts[0], "123456", 30000)
true
> personal.unlockAccount(eth.accounts[1], "666666", 30000)
true
> eth.sendTransaction({from:eth.accounts[0],to:eth.accounts[1],value:10000
0000})
"0xec82ff8b518999fba6294b507028b14f2f4c452691e44686b821bf6dbfc0a795"
> eth.getBalance(eth.accounts[1])
0
> miner.start(10)
null
> eth.blockNumber
138
> eth.getBalance(eth.accounts[1])
8000000000
> miner.stop()
null
```

图 2 - 7 在测试节点间挖矿的截图

2.2 任务 2 实验步骤及结果

首先配置 remix 连接本地以太坊网络，remix 连接本地以太坊网络的端口，默认为 8545，可以在启动 geth 时用 -http.port 8546 指定。

①编写并部署合约

根据实验指导手册和相关资料编写 Attack.sol，实现攻击合约。保存代码后，编译器会自动对合约代码进行编译。按照图 2-8 的步骤，首先部署 EtherStore 合约。以账户 acc1 作为被攻击的受害账户，部署成功后，系统会在合约模块下方显示 EtherStore 合约的部署地址。接着，如图 2-9 所示，使用账户 acc2 作为攻击方账户部署 Attack 合约，并在部署参数中填入 EtherStore 合约的地址。Attack 合约在初始化时将利用该地址实例化 EtherStore，以便后续攻击操作能够访问和调用受害合约的相关函数。

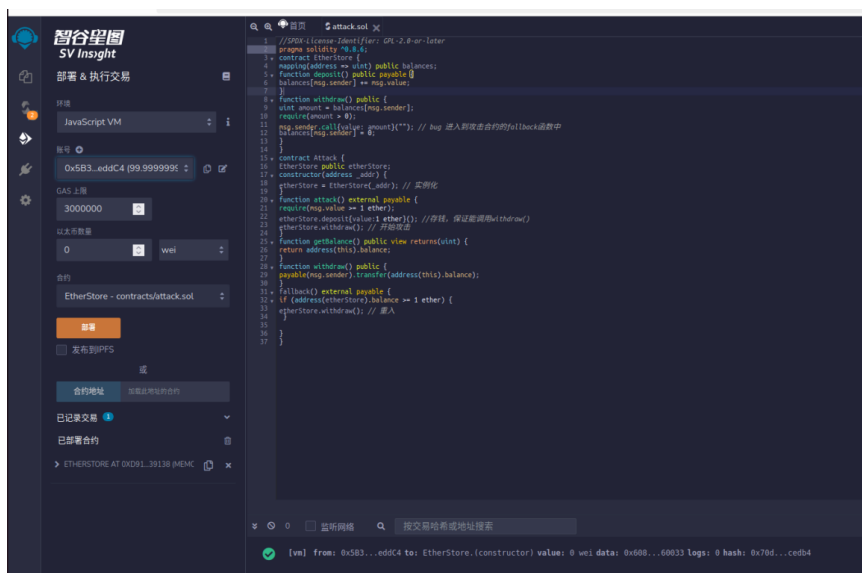


图 2-8 节点 1 智能合约部署完成截图

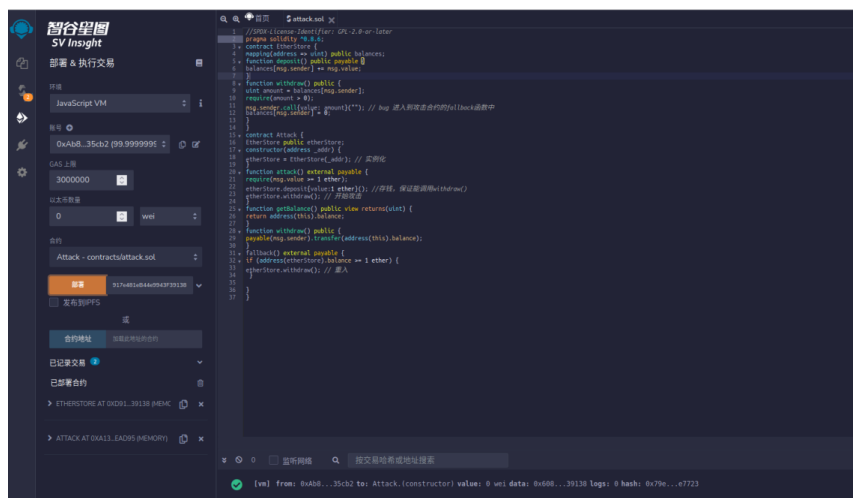


图 2-9 节点 2 攻击合约部署完成截图

②复现重入漏洞

EtherStore 合约的 `withdraw` 函数中,使用 `msg.sender.call{value: amount}("")` 将资金转出。由于 `.call()` 是一种低级调用方式,当调用者是智能合约时,目标合约的 `fallback` 函数将被执行。恶意合约 `Attack` 的 `fallback` 函数中再次调用 `withdraw`,利用 EtherStore 合约在转账前尚未更新 `balances[msg.sender]` 的漏洞,重复执行 `withdraw` 逻辑,导致多次提取相同余额。

通过对触发重入漏洞的机制的了解,我们可以知道如何才能触发重入漏洞。EtherStore 中 `balances[msg.sender] = 0;` 在 `withdraw` 函数的最后执行,因此在第一次调用 `msg.sender.call{value: amount}("")` 时,余额尚未被清零,给了恶意合约在 `fallback` 函数中再次调用 `withdraw` 的机会。`Attack` 合约的 `fallback` 函数被多次递归调用,每次调用都可以提取一次资金,直到 EtherStore 合约中的余额耗尽。

然后将理论进行实践,我们先在 EtherStore 合约中存入 10 Ether,以确保有足够余额供攻击合约提取,如图 2-10 左侧可以看到 Ether 值减少,点击 `balances` 查看被攻击者的余额,如图 2-10 右侧所示。

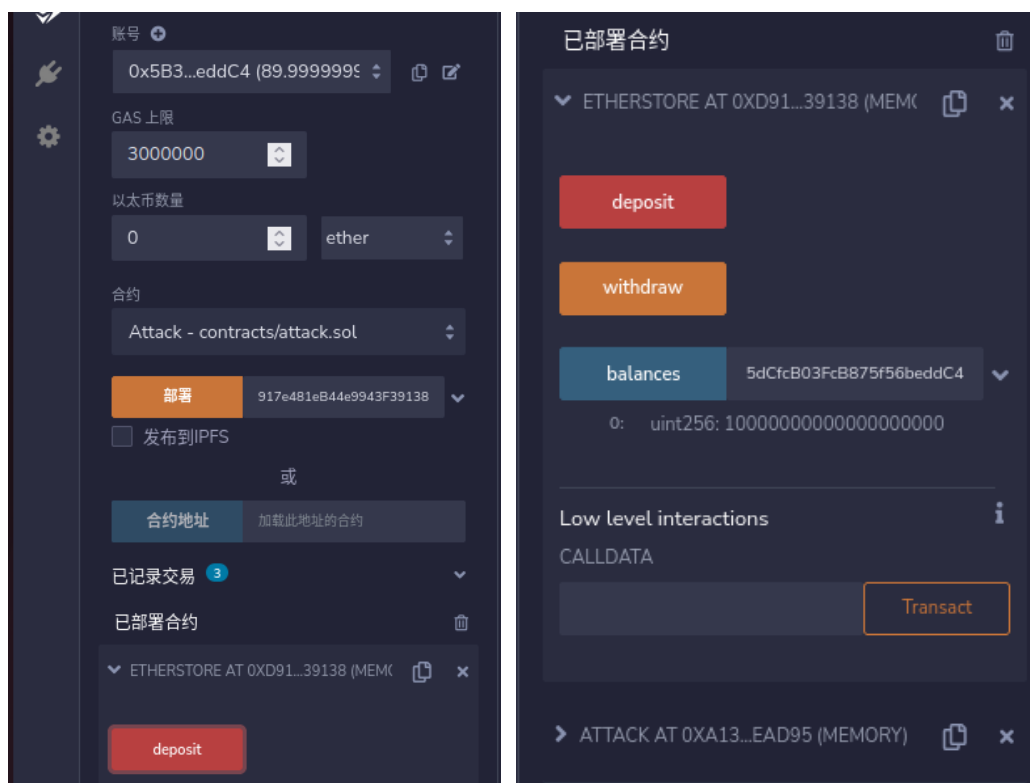


图 2 - 10 被攻击者存钱(左)和查看余额(右)截图

然后执行攻击,如图 2-11 左侧所示,调用 `Attack` 合约的 `attack` 函数,传入 1 Ether。`attack` 调用后, EtherStore 中的余额将不断被提取,直到余额耗尽。查看 `Attack` 合约的余额,应为原始余额加上攻击过程中获取的余额,操作过程中需要消费一点点 Ether,如图 2-11 右侧所示。

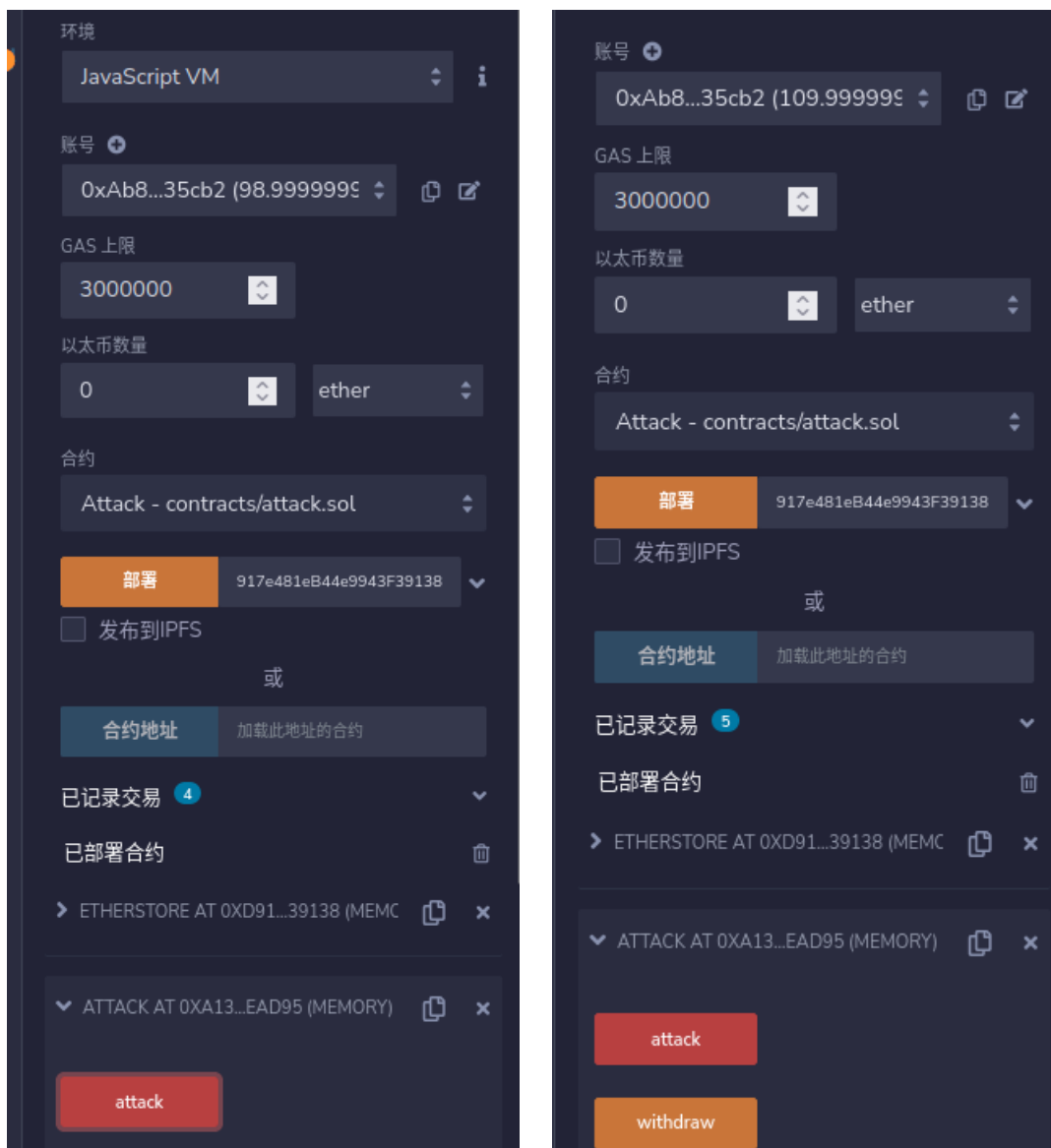


图 2 - 11 节点 2 使用攻击合约(左)和取钱(右)成功截图

2.3 代码实现

本实验的主要目标是，该部分的代码和指令指导手册中均详细地给出，并有简明地解释，同时我在上部分“任务 2 实验步骤及结果”中对部分指令在使用时进行了相关地应用型解释，下面我仅针对部分在演示过程中需要使用到地指令来进行更加具体地解释说明。

➤ **\$ personal.unlockAccount(eth.accounts[0], passwd, 30000)**

通过此命令解锁账户，其中 `eth.accounts[0]` 表示要解锁的第一个账户，`passwd` 是账户密码，`30000` 代表解锁时间。

➤ **\$ eth.sendTransaction({from:address0, to:address1, value:amount})**

进行转账的命令，指定了转出账户 `address0`、转入账户 `address1` 和转账金额 `amount`。

➤ \$ miner.start() / miner.stop()

执行此命令开始/停止挖矿，可指定线程数量。挖矿是为了在区块链网络中确认交易和获取代币收益。

➤ \$ eth.blockNumber / eth.getBalance(eth.accounts[0])

列出区块总数（忽略了创世区块）/ 查看某账户的余额，返回值的单位是 Wei，1ether = 10^{18} Wei。

同时其中编写智能合约的代码也非常重要，我们具体对其编写逻辑进行更加深入了解。

① 以太坊转账方法中的代码

.transfer(): Solidity 中 transfer() 方法的作用是进行转账操作，并只会发送 2300 gas 进行调用，当发送失败时会通过 throw 来进行回滚操作，从而防止了重入攻击。

.send(): 类似地，send() 方法也只会发送 2300 gas 进行调用，发送失败时会返回布尔值 false，不会抛出异常，利用 gas 有限的特点来防止重入攻击。

.gas().call.value(): 该方法在调用时会发送所有的 gas，发送失败时返回布尔值 false 且不抛出异常，因此不能有效防止重入攻击。

② Fallback 回退函数代码

```
01. function() public payable {
02. }
```

这是一个没有名字的函数，它没有参数且无返回值。当满足以下几种情况时会被执行：调用合约时没有匹配到任何一个函数；没有传数据；智能合约收到以太币（为了接收以太币，fallback 函数必须被标记为 payable）。

③ 重入漏洞原理中的代码

```
01. contract MaliciousContract {
02.     address public victimContract;
03.
04.     constructor(address _victimContract) {
05.         victimContract = _victimContract;
06.     }
07.
08.     fallback function() external payable {
09.         // 恶意代码，可能会再次调用受害者合约的某些函数进行攻击
10.         if (victimContract.call.value(eth.balanceOf(this))()) {
11.             // 这里的代码可能会造成重入漏洞攻击
12.         }
13.     }
14. }
```

在这个假设的恶意合约中，定义了一个构造函数用于接收受害者合约的地址。回退函数中通过 if (victimContract.call.value(eth.balanceOf(this))()) 尝试再次调用受害者合约的某些函数，若受害者合约在处理转账时没有考虑到重入漏洞，就可能被恶意利用，导致资金或其他问题。具体的恶意行为可能会根据实际情况在 if 语句的代码块中进行编写。这样的恶意合约可能会被攻击者部署在一个外部地址，当受害者合约向该地址发送以太币时，就会触发恶意代码的执行，从而引发重入漏洞攻击。

三、实验分析

3.1 遇到的问题

(1) 问题一：实验配置环境配置过程中，出现版本不匹配的现象，导致后面以太坊用户并不能进行挖矿，需要重新降低 Geth 版本。

(2) 问题二：在实验过程中，虽然成功编译了 Geth，但在不同的终端中无法直接运行 Geth 命令，且在 HOME 目录下直接输入 export 的命令在新终端没有作用，这是由于环境变量未被新终端会话识别到。

3.2 针对问题采取的解决方法

针对问题一，可以先通过 `sudo apt remove geth` 删除原先版本，然后克隆符合要求的版本。

针对问题二，通过将 Geth 的路径加入到系统的 `~/.bashrc` 文件，注意这里必须将命令准确地输入 `.bashrc` 的文件，并使用 `source ~/.bashrc` 命令使配置生效，确保所有新开的终端会话都能识别到 `geth` 命令。

3.3 设计方案存在的不足

无。

四、实验总结

4.1 实验感想

通过本次以太坊私有链实验，我深入理解了区块链网络搭建、节点管理和以太坊智能合约开发的基本流程。特别是在重入漏洞实验中，我亲身体会到了 Solidity 编程中存在的安全隐患，也认识到智能合约安全性的重要性。在未来的开发过程中，我将更加注重代码安全性，采取合适的编码规范和安全检查措施，以降低智能合约的风险。

重入漏洞是由于在外部调用返回之前，合约内的状态变量未更新，导致多次调用造成的损失。在本实验中，通过 Attack 合约，我们成功复现了重入攻击，演示了恶意合约如何通过 fallback 函数反复调用 withdraw 以提取超额余额，需要明确地是这仅用于实验和学习目的，在实际的区块链环境中，恶意攻击是不被允许的，且会违反法律和道德规范。同时在实际开发和使用智能合约时，我们需要特别注意防范重入漏洞等安全问题，我们可以通过下面的方式有效防止重入攻击，例如可以通过将 `balances[msg.sender] = 0;` 放置在 `msg.sender.call{value: amount}("")` 之前，确保在转账前就清空用户余额；使用 `transfer` 或 `send` 函数代替 `.call{value: amount}("")`，因为 `transfer` 和 `send` 的 gas 限制较低（2300 gas）等。

此外，通过本次实验，我更加认识到网络安全专业知识对个人和社会的双重价值。面对网络安全挑战，我们不仅要提升自身的技术能力，还要增强使命感和责任意识，践行“科技向善”的理念。未来，我将继续努力学习网络空间安全的相关知识，提升技能，为构建安全、可信的网络空间贡献自己的一份力量。

4.2 意见和建议

实验指导书的内容非常详实，基本上按照步骤进行操作即可完成实验，而且对可能遇到的意外情况进行了详实地预告和指导，非常感谢老师们用心地编写和教导。

原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

已阅读并同意以下内容。

判定为不合格的一些情形：

- (1) 请人代做或冒名顶替者；
- (2) 替人做且不听劝告者；
- (3) 实验报告内容抄袭或雷同者；
- (4) 实验报告内容与实际实验内容不一致者；
- (5) 实验代码抄袭者。

作者签名：