华东师范大学数据科学与工程学院实验报告

课程名称：区块链与分享型数据库	**年级**：大三	**上机实践成绩**：
指导教师：张召	**姓名**：唐益	**学号**：10215501445
上机实践名称：基于区块链的投票系统		**上机实践日期**：2024.6.18
上机实践编号：	**组号**：	**上机实践时间**：

一、实验目的

本项目参考了 petshop 实验以及群里发的 demo，使用 truffle 和 Vue3 的框架完成了一个基于以太坊的区块链投票系统，主要完成了智能合约的编写，智能合约与前端的交互以及简单的前端页面的展示，在完成了实验功能要求的基础上尝试实现了匿名投票系统的零知识证明，最终实现的 web 页面的使用演示视频已经随报告和代码一同附上。

二、实验任务

（1） 完成投票系统智能合约的编写
（2） 完成合约的部署以及前端的展示
（3） 在前端对于投票构建默克尔树进行零知识证明

三、实验环境

truffle
Vue3

四、实验过程

我会从智能合约的编写以及智能合约与前端的编写两个方面来对于实验过程进行记录，同时在本次实验中其实遇到了一些问题，也会在实验过程中详细地记录。

4.1 智能合约的编写

首先是智能合约的编写，我定义了一个 Vote 的结构体来保存并维护与投票活动相关的信息，我选择了记录 id(为了确保每个投票活动有唯一标识)、creator（创建者 address，为了实现发起投票要转账功能）、name（活动名称）、rewardAmount（奖励金额，为了实现给获胜者转账）、ended（为了记录投票活动结束与否，如果结束就不能再参与投票）、winner（获胜者 address，同样为了实现给获胜者转账）、candidate（候选人 id，用户可以给参与投票成为获胜者的人投票）、hasJoined（记录用户是否加入这个投票活动成为候选人，因为候选人不能给这个投票活动投票并且不能二次加入活动）、votesReceive（每个候选人获取投票

数，因为要根据这个来判断 winner）、voterCount（整个投票活动参与的投票数，因为要根据这个来判断是否 ended）。

```
contract VotingSystem {
    struct Vote {
        uint id;
        address creator;
        string name;
        uint rewardAmount;
        bool ended;
        address winner;
        address[] candidates;
        mapping(address => bool) hasJoined;
        mapping(address => uint) votesReceived;
        uint voterCount;
    }
```

接下来，我会从功能的角度来分别描述所实现的功能：

4.1.1 发起投票 -- createVote

Step1：定义一个 Vote 的结构体变量 newVote，并将它 push 到 votes 投票数组中。

Step2：接着按照上面所罗列的 Vote 结构体的信息进行初始化，name 和 rewardAmount 为调用函数时传入的变量，id 设置为已有投票活动的数量，creator 就是调用函数的地址，ended 一开始未结束，参与投票的人为 0。

Step3：然后维护 voteCount++，即已发起的投票活动的数量，这也是为了确保每个投票活动的唯一 id。

Step4: 实现发起活动的人向合约部署者的支付一定费用，最后返回的是投票 id。

```
function createVote(string memory _name, uint _rewardAmount) public payable returns (uint) {

    votes.push();
    Vote storage newVote = votes[votes.length - 1];

    newVote.id = voteCount;
    newVote.creator = msg.sender;
    newVote.name = _name;
    newVote.rewardAmount = _rewardAmount;
    newVote.ended = false;
    newVote.voterCount = 0;

    emit VoteCreated(newVote.id, _name, msg.sender);
    voteCount++;
    payable(owner).transfer(msg.value);
    return voteCount - 1;
}
```

4.1.2 参与投票成为候选人 – joinVote

Tips：在这一部分我一开始是使用 require 函数来保证比如说参与投票成为候选人要支付一定金额给投票发起人、候选人不能给这个投票活动投票这样的条件，但是这样会引发一个问题就是在前端交互的时候 MetaMask 会直接强制拒绝不满足条件的交易，但是我所希望

看到的是交易后系统会提示在哪里出错以及错误原因是什么，后来经过学长的建议是通过将所有的 require 全部改成 if 加一个 VoteError 事件的做法，通过条件判断是否释放事件，然后再前端监听事件，这样前端可以获取错误的反馈并且呈现。

Step1：本函数要传入的是要参加的投票活动 id., 对于要参与的投票活动 id 不合法、已经参加过这个投票，投票已经结束、不够支付金额四个方向来进行错误事件的反馈。

Step2：维护 currentVote.candidates，将调用函数的 address 加入，以及将这个 address 的 currentVote.hasJoined 设置为 True。

Step3：为了设置投票结束规则，我设置的是参与投票的人达到三个就活动结束（当然这个数字也可以从前端传入，由于我的 Ganache 的测试链用户有限我就简单地设置了三），所以在 joinVote 的最后，我实现并调用了一个 endVote(_voteId);的函数，这个函数下一条中有具体阐释。

```
function joinVote(uint _voteId) public payable {
    //require(_voteId < votes.length, "Invalid vote ID");
    Vote storage currentVote = votes[_voteId];
    //require(!currentVote.ended, "Vote has already ended");
    //require(!currentVote.hasJoined[msg.sender], "You have already joined this vote");
    //require(msg.value > 0, "You must pay to join the vote");

    if (_voteId >= votes.length) {
        emit VoteError(_voteId, msg.sender, "Invalid vote ID");
        return;
    }
    if (currentVote.hasJoined[msg.sender]) {
        emit VoteError(_voteId, msg.sender, "You have already joined this vote");
        return;
    }
    if (currentVote.ended) {
        emit VoteError(_voteId, msg.sender, "Vote has already ended");
        return;
    }
    if (msg.value <= 0) {
        emit VoteError(_voteId, msg.sender, "Insufficient payment");
        return;
    }
    currentVote.candidates.push(msg.sender);
    currentVote.hasJoined[msg.sender] = true;

    if (currentVote.voterCount >= 3) {
        endVote(_voteId);
    }
}
```

4.1.3 到达一定人数自动结束投票并奖励 winner – endVote

Step1：遍历候选人，使用打擂台的方法比较谁获得的投票数最多，将最多得票的候选人的 address 记录进 winner。

Step2：向 winner 的地址转账 reward 金额。

```
function endVote(uint _voteId) public payable{
    //require(_voteId < votes.length, "Invalid vote ID");
    Vote storage endingVote = votes[_voteId];
    //require(!endingVote.ended, "Vote has already ended");

    uint maxVotes = 0;
    address winner;

    for (uint i = 0; i < endingVote.candidates.length; i++) {
        if (endingVote.votesReceived[endingVote.candidates[i]] > maxVotes) {
            maxVotes = endingVote.votesReceived[endingVote.candidates[i]];
            winner = endingVote.candidates[i];
        }
    }

    endingVote.ended = true;
    endingVote.winner = winner;

    if (winner != address(0)) {
        payable(winner).transfer(endingVote.rewardAmount);
    }

    emit VoteEnded(_voteId, winner);
}
```

4.1.4 投票—vote

Step1：和之前所说一样，将候选人不能投票、活动结束不能投票的错误提醒放在 if 判断里触发 VoteError 事件。

Step2：投票需要传入参数活动 id 和候选人 address，维护 votesReceived[_candidate]加一以及整个投票活动获得票数 voterCount 加一。

```
function vote(uint _voteId, address _candidate) public {
    //require(_voteId < votes.length, "Invalid vote ID");
    Vote storage targetVote = votes[_voteId];
    //require(!targetVote.ended, "Vote has already ended");
    //require(targetVote.hasJoined[_candidate], "Candidate has not joined this vote");
    //require(!targetVote.hasJoined[msg.sender], "Candidates cannot vote");

    if (targetVote.hasJoined[msg.sender]) {
        emit VoteError(_voteId, msg.sender, "Candidates cannot vote");
        return;
    }

    if (targetVote.ended) {
        emit VoteError(_voteId, msg.sender, "Vote has already ended");
        return;
    }

    targetVote.votesReceived[_candidate]++;
    targetVote.voterCount++;

    if (targetVote.voterCount >= 3) {
        endVote(_voteId);
    }

    emit Voted(_voteId, _candidate, msg.sender);
}
```

4.1.5 用于返回投票活动相关信息--getVotes

这个函数主要是为了方便在前端展示所有投票活动相关信息，由于之前我们记录候选人是否加入投票以及得到的票数时用到了 mapping，但是 mapping 本身不可迭代，无法通过遍

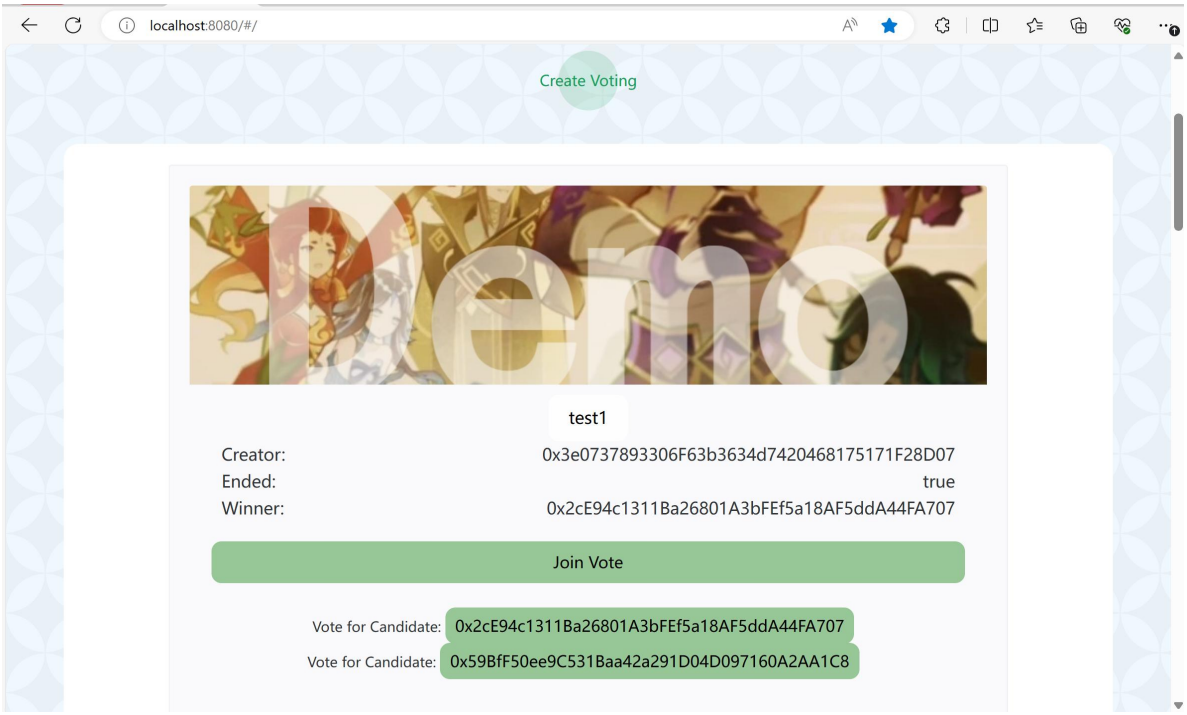
历的方式来枚举，传给前端不方便，所以在这里我又定义了一个 VoteInfo 的结构体，将 mapping 变量去除，将合约维护的数据传给前端：

```
function getVotes() public view returns (VoteInfo[] memory) {
    VoteInfo[] memory voteInfos = new VoteInfo[](voteCount);

    for (uint i = 0; i < voteCount; i++) {
        Vote storage currentVote = votes[i];

        voteInfos[i] = VoteInfo({
            id: currentVote.id,
            creator: currentVote.creator,
            name: currentVote.name,
            rewardAmount: currentVote.rewardAmount,
            ended: currentVote.ended,
            winner: currentVote.winner,
            candidates: currentVote.candidates
        });
    }
    return voteInfos;
}
```

4.2 Vue3 和智能合约的交互

我没有对于前端样式做很多的调整，会在报告中主要讲一下前端部署调用智能合约以及展示智能合约中维护的关于投票活动信息的内容。

页面有一个 Creat Voting 绑定 postSubmit 逻辑可以用于发起投票活动，然后每个投票活动都有展示，有 creator、winner、ended 的信息，提供了 Join Vote 的按钮绑定 joinVote 用于成为候选人，同时，投票活动列出候选人 address，直接点击候选人绑定 vote 用于投票。



4.2.1 展示投票活动相关信息以及前端返回的不正确操作报错

loadPostedData：调用 getVotes 函数，获取投票 id、创建者 address、投票名称、奖金

（转换为浮点数）、投票是否结束、获胜者 address（如果还没有 winner 就是 0）、候选人列表。

listenToEvents：从最新的区块监听 VoteError 事件，利用{ voteId, candidate, message }来记录事件的返回值，根据在智能合约里对于 VoteError 事件结合 if 的使用返回的错误编写报错消息。

handleVoteError：将 listenToEvents 收集的{ voteId, candidate, message }以 alert 的方式在前端页面弹出操作的错误提醒。

```
const votes = ref([]);
const errorMessages = ref([]);

const loadPostedData = () => {
  VotingSystemContract.deployed().then((instance) => {
    return instance.getVotes.call();
  }).then((response) => {
    votes.value = response.map(vote => ({
      id: vote.id,
      creator: vote.creator,
      name: vote.name,
      rewardAmount: parseFloat(vote.rewardAmount),
      ended: vote.ended,
      winner: vote.winner,
      candidates: vote.candidates
    }));

  }).catch((err) => {
    console.log(err.message);
  });
};
```

```
const handleVoteError = (voteId, candidate, message) => {
  errorMessages.value.push(`Error in vote ${voteId} for candidate ${candidate}: ${message}`);
  alert(`Error in vote ${voteId} for candidate ${candidate}: ${message}`);
};


const listenToEvents = async () => {
  const instance = await VotingSystemContract.deployed();

  instance.VoteError({ fromBlock: 'latest' }, (error, event) => {
    if (error) {
      console.error('Error:', error);
      return;
    }
    const { voteId, candidate, message } = event.returnValues;
    console.error(`Error in vote ${voteId} for user ${candidate}: ${message}`);
    handleVoteError(voteId, candidate, message);
  });
};

loadPostedData();
listenToEvents();
```

4.2.2 发起投票活动

postSubmit：调用 createVote 函数，由于发起投票的人要向部署合约的人支付，所以这个调用需要传入 value，value 由用户在前端交互中输入的 reward 金额决定。

```
const postModalFlag = ref(false);
const post_vote = reactive({});
const init_post_vote = () => {
  post_vote.creator = '';
  post_vote.name = '';
  post_vote.rewardAmount = 1000000;
  post_vote.info = '';
};
init_post_vote();

const showPostModal = () => {
  post_vote.id = Date.now();
  postModalFlag.value = true;
};

const postSubmit = () => {
  VotingSystemContract.deployed().then(async (instance) => {
    const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
    const account = accounts[0];
    let res = instance.createVote(post_vote.name, post_vote.rewardAmount, { from: account, value: post_vote.rewardAmount * 1e12});
    console.log(post_vote.name);
    console.log(res);
    return res;
  }).then((response) => {
    init_post_vote();
    loadPostedData();
    postModalFlag.value = false;
  }).catch((err) => {
    alert('error', err.message);
    console.log(err.message);
  });
};
```

4.2.3 参与投票

joinVote: 调用合约中的 joinVote 函数，同样的，需要传入 value，实现参与投票的人给投票活动的发起人转账。

```
const joinVote = (id) => {
  VotingSystemContract.deployed().then(async (instance) => {
    const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
    const account = accounts[0];
    return instance.joinVote(id, { from: account, value: 500000 * 1e12 });
  }).then((response) => {
    loadPostedData();
  }).catch((err) => {
    alert('error', err.message);
    console.log(err.message);
  });
};
```

4.2.4 投票

castVote： 调用 vote 函数，由于后面的关于匿名投票系统的零知识证明，对于投票进行哈希，构建默克尔树。

```
const castVote = (id, candidate) => {
  VotingSystemContract.deployed().then(async (instance) => {
    const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
    const account = accounts[0];
    await instance.vote(id, candidate, { from: account });
    loadPostedData();

    const hashedVote = hashVote(`${id}-${candidate}-${account}`);
    votesData.value.push(hashedVote);

    const tree = buildMerkleTree(votesData.value);
    merkleTree.value = tree.flat();
    merkleRoot.value = tree[tree.length - 1][0];

    const voteIndex = votesData.value.indexOf(hashedVote);
    const proofPath = generateMerkleProof(voteIndex, votesData.value);
    merkleProof.value = {
      leaf: hashedVote,
      path: proofPath,
      merkleRoot: merkleRoot.value,
    };


  }).catch((err) => {
    alert('error', err.message);
    console.log(err.message);
  });
};
```

4.2.5 零知识证明

对于一个匿名的投票系统，我希望在**不看到投票信息的情况下去验证某一个投票包含在最终的计票中**，基于这个想法同时结合 minichain 的实验，我在 castVote 中将投票使用 ${id}-${candidate}-${account}字符串计算哈希， 然后构建默克尔树，每次投票都会生成一颗包含当前投票信息的默克尔树。

buildMerkleTree：传入的是一个叶子节点的数组，和 minichain 里面一样，需要注意在叶子节点是奇数的时候需要复制一下自己，与自己计算根。为了更好地展示树的结构，维护了 children 数组，记录每一个节点的叶子节点。

```
const buildMerkleTree = (leaves) => {
  if (leaves.length === 0) return [];

  let nodes = leaves.map(leaf => ({ hash: leaf, children: [] }));
  const tree = [nodes];

  while (nodes.length > 1) {
    if (nodes.length % 2 !== 0) {
      nodes.push(nodes[nodes.length - 1]);
    }
    const newLevel = [];
    for (let i = 0; i < nodes.length; i += 2) {
      const combinedHash = CryptoJS.SHA256(nodes[i].hash + nodes[i + 1].hash).toString();
      newLevel.push({
        hash: combinedHash,
        children: [nodes[i], nodes[i + 1]],
      });
    }
    nodes = newLevel;
    tree.push(nodes);
  }
  return tree;
};
```

五、实验总结

由于基于区块链的投票系统涉及很多转账相关的操作，截图不太方便展示结果，所有的功能实现都在演示视频中展示，两个演示视频都有声音说明。通过本次的实验，完成了一个基于以太坊的投票系统，实现了智能合约的设计与编写，以及前端对于智能合约的部署和调用，尝试了一个非常简单的零知识证明的想法，对于区块链的应用以及原理有了更深的认识。

通过本次实验，对于智能合约的了解更深入了，感觉它有点像一般系统的数据库和后端的结合，它可以存储状态并且，同时它具有一些特点，比如说部署后不可更改--智能合约一旦部署了就无法更改，再更改需要重新的部署（发现有些时候 truffle migrate 没有用，可能需要 truffle migrate --reset）；比如自动执行—满足合约的条件机制就会自动执行，比如投票会自动结束且结束后会自动给 winner 转账；去中心化执行—它不依赖于某个单一的服务器；比如透明安全—智能合约执行和数据存储在区块链上，数据会很安全，因为就比如在使用数据库和后端做投票系统的话，可以在本地的数据库直接进行删除更改数据的操作，但是使用智能合约的话这就是不可能的，但是与此同时，它也牺牲了一些灵活性与可拓展性。