期末最终报告

15331201
林 昆

1. 概述

项目地址：https://github.com/Lamken/answer_me
answer_me 是一个基于区块链的轻量化问答社区。此文档介绍 answer_me 是一个基于区块链的轻量化问答社区的智能合约部署及运行方法

2. 技术选择

在 truffle 框架部署智能合约的基础上，使用 ganache 作为私链，同时使用 react 和 boost 提供 UI

- 智能合约部署：truffle
- 底层区块链实现：使用 ganache 作为测试私链
- UI：使用了 react 和 bootstrap

3. 选题背景与依据

需求背景

知识付费越来越成为当前知识交流社区的重要盈利模式，然而回答者常常良莠不齐，同时由于当前主流社区存在中心化问题，导致部分回答可能会被屏蔽与修改。对于提问者而言，在使用区块链技术之后，能够完整并客观记录每一位回答者的回答轨迹，为提问者提供依据，从而选择更可信的回答。对于回答者而言，每一个优质回答都能得在所有人见证下得到报酬，激发回答动力，使社区能够有更多的专业回答。

区块链的优势

众所周知，区块链重要的特性是其去中心化与“共同承认”，而在忠实记录回答者回答轨迹上能够让每一位用户都能得到一份完整的备份，从而实现高质量而客观高端付费知识问答社区

功能简述

提问者发布问题与报酬，并在多个回答者提供的问题答案中确认一个有用并支付报酬。回答对所有用户可见。

4. 使用说明

运行环境

```
Truffle:        v5.0.0 (core: 5.0.0)
Solidity:       v0.5.0 (solc-js)
Node:           v8.10.0
npm:            v5.6.0
ganache:        v1.2.2.24
```

在运行 dapp 前，可以先运行 project 下的配置脚本 set_enviroment.sh

```
git clone https://github.com/Lamken/answer_me
cd answer_me/project
sh set_enviroment.sh
cd client
#  此处需要运行 ganache
truffle migrate
npm start
```

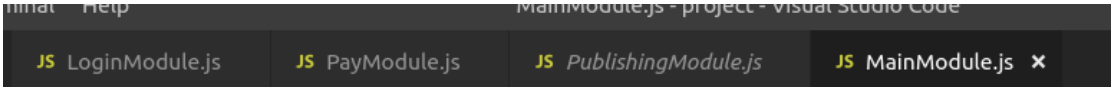
5. UI 设计

设计概述

由于时间较紧（失学失业还有期末），UI 更多参照的是知乎的 UI

基于 React 框架的 UI 设计

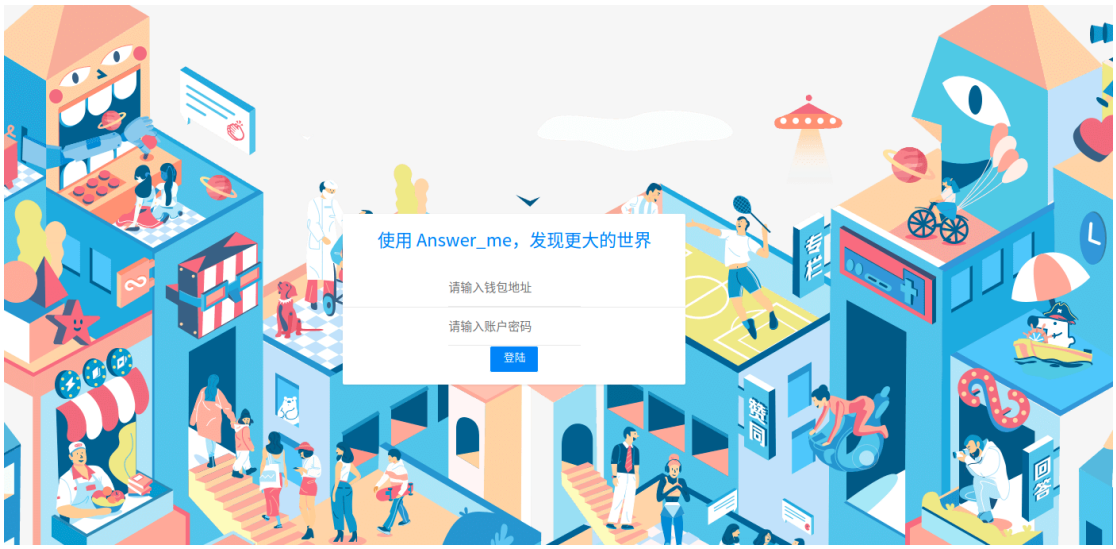
根据智能合约的功能，基于 React 设计了以下四个 UI



- 用户登陆界面
- 用于浏览已有答案和发起提问的主页面
- 提问者采纳回答的页面
- 回答的页面

用户登陆界面（伪知乎）

其中用户直接使用账户钱包地址和密钥登陆，这样不仅能够简化对于用户账户的存储和注册操作，还能让用户无需记住在此间的账户与密码。



用于浏览已有答案和发起提问的主页面

- 用户能够看到当前所有的提问与被采纳的回答
- 能根据问题关键字进行搜索（实际上这个功能还没有做好）
- 也能在找不到自己所需要的答案时发起提问

提问者采纳回答的页面

问题发出者能够看到原本问题的描述与回答者的回答，再决定是否采纳该回答

```
 return (
<Panel bsStyle="primary" id="acceptpanel">
<Panel.Heading>
  <Panel.Title componentClass="h3">选择最佳回答</Panel.Title>
</Panel.Heading>
<Panel.Body>

  <div class="alert alert-warning" role="alert">
    清选择您的最佳回答并支付回答者酬劳
  </div>

  <div class="form-group" id="questionContent">
    <label>问题描述</label>
    <div type="text" id="acceptinfo">{questioner}</div>
  </div>

  <div class="form-group" id="answerContent">
    <label>回答内容</label>
    <div type="text" id="acceptinfo">{answer}</div>
  </div>

  <div class="form-group" id="priceButton">
    <label>支付酬劳</label>
    <div type="text" id="acceptinfo">{code}</div>
  </div>

  <div id="acceptbutton">
    <Button bsStyle="success" class="form-control" onClick={(event) => {
    this.props.confirmAnswer(this.props.delegator);
    }}>采用该回答</Button>
  </div>
```

6. 智能合约逻辑说明

以下将陈述部署的智能合约

智能合约结构体

```
struct QuestionTransaction {
    address payable questioner; // 提问者
    address payable respondent; // 回答者
    uint price; // 问题价格
    string question; // 问题内容
    string answer; // 回答内容
    State state;
}
```

其中的 State 指代问题的可编辑和已采纳回答的状态，将会在《问题的可编辑与关闭状态》进行讨论

问题的可编辑与关闭状态

用户提出的问题将分为两个状态，分别是开放回答与已采纳回答。在开放回答中其他用户可以凭借自己的经验回答问题，而用户一旦采纳了回答，其他用户将不能够继续回答该问题。这样的设计或许能够简化社区回答。

```
enum State { // 问题状态
    Open,
    Close
}
```

存储智能合约的数据结构

使用了 map 可变长的特性，其本质还是使用队列的方式进行存储，并维护一个问题数目的变量，使其始终指向队尾

```
// 用队列存储提问交易，最新的交易放到队尾
mapping(uint => QuestionTransaction) transationList;
uint32 transactionCount = 0; // 队尾，即交易数量

// ------------------------end 合约变量---------------------------------
```

读取智能合约的信息

```
// ------------------------得到合约信息---------------------------------
function getTransaction(uint id) public view returns(address) {
    //  根据问题id得到问题合约的地址
    return TransationList[id];
}

function getQuesioner(uint id) public view returns(address) {
    // 根据id得到提问者的钱包地址
    return transationList[id].questioner;
}

function getRespondent(uint id) public view returns(string memory) {
    // 根据id得到提回答者的钱包地址
    return transationList[id].respondent;
}

function getPrice(uint id) public view returns(uint) {
    // 根据id得到该问题的酬劳
    return transationList[id].price;
}

function getQuestion(uint id) public view returns(uint) {
    // 根据id得到该问题的内容
    return transationList[id].question;
}

function getAnswer(uint id) public view returns(uint) {
    // 根据id得到该问题的回答
    return transationList[id].answer;
}

function getTransactionCount() public view returns(uint) {
    // 现有问题数目
    return transactionCount;
}
// ------------------------end 得到合约信息---------------------------------
```

用户提问

将最新的提问放到队尾，将默认的回答者指向提问者本身

```
// ----------------------提问逻辑处理模块---------------------------------
function publish(string memory delegate, uint memory price) public payable {
    require(price >= 1 ether, "支付的金额必须大于1个以太币"); // 支付的金额必须大于1个以太币
    valueInfo[msg.sender] = msg.value;
    delegateInfo[delegate].push(msg.sender);
    destinationInfo[msg.sender] = destination;
    transactionCount = transactionCount + 1; // 提问合约放到队尾
    transationList[transactionCount].requestor = msg.sender;
    transationList[transactionCount].pay = msg.value;
    transationList[transactionCount].respondent = msg.sender; // 默认为回答者为提问者本身
    transationList[transactionCount].state = State.Open; // 提出的问题默认为开放编辑状态
}
// ----------------------end 提问逻辑处理模块---------------------------------
```

判断回答操作

只有 open 状态的问题才能回答

```
// ----------------------判断回答模块-----------------------------------

// 判断是否在问题的可回答状态
modifier isQuestionOpened(State _state) {
    require(
        _state == State.Open,
        "Error state."
    );
    _;
}

// 判断是否在问题的关闭状态
modifier isQuestionClosed(State _state) {
    require(
        _state == State.Close,
        "Error state."
    );
    _;
}
// ----------------------end 判断回答模块---------------------------------
```

回答与采纳回答操作

其中设计有不合理之处，即为每个回答都会覆盖掉之前的回答，直接写进智能合约，实际上这里应该使用一个 List，只是时间有限，先留下个 TODO 了

```
// ----------------------回答与接纳模块---------------------------------
// 回答问题
function answerQuestion(address payable addr) public payable {
    answer = msg.answer; // 回答内容
    respondent = addr;
    return msg.answer;
}

// 提问者接纳回答
function confirmAnswer(uint questionId) public payable isQuestionOpened(questionTransation[questionId]) {
    // 更新交易状态为完成
    questionTransation[questionId] = State.Close;
    delegator.transfer(questionTransation[questionId].respondent + price ether);
}
// ----------------------end 回答与接纳模块---------------------------------
```

7. 智能合约部署

Ganache

运行 ganache 作为私链，这样能够关注于上层 UI 与逻辑模块的开发。在 Ubuntu18.04 上，直接使用 ganache 的 appImage 运行，项目默认设定在 7545 端口。

编译

truffle 在根目录下的 project 进行编译

```
lamken@PC:~/blockchain/answer_me$ truffle compile
Compiling ./contracts/AnswerContract.sol...
Compiling ./contracts/Migrations.sol...
Compiling ./contracts/SimpleStorage.sol...
Writing artifacts to ./build/contracts

lamken@PC:~/blockchain/answer_me$
```

设置 Deploy

设定好部署的依赖，防止重复部署

JS 2_deploy_contracts.js

```
1  const AnswerContract = artifacts.require("AnswerContract");
2
3  module.exports = function(deployer) {
4    deployer.deploy(AnswerContract);
5  };
```

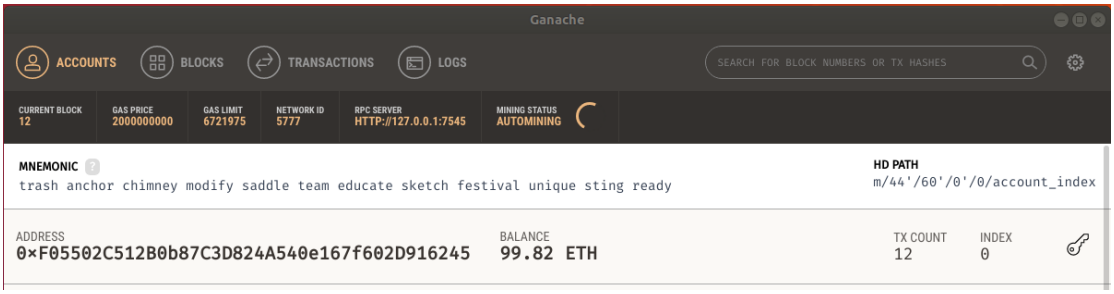
Migrate

我们将从中得到智能地址，在后面的 UI 开发中进行使用

```
2_deploy_contracts.js
=====================

   Deploying 'AnswerContract'
   --------------------------
   > transaction hash:    0x8fb4f5158a96a684e91fed73caf9a026fabf92dc9780a6bd7770601ddd2b9b29
   > Blocks: 0            Seconds: 0
   > contract address:    0xAd125dFCA83841f39Ccca61A31Ffb1a6178295a5
   > account:             0xF05502C512B0b87C3D824A540e167f602D916245
   > balance:             99.81948024
   > gas used:            2663698
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.05327396 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.05327396 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.05897212 ETH

lamken@PC:~/blockchain/answer_me$
```

智能合约地址与在私有链上的

我们可以看到在成功部署智能合约之后，账户的以太币有所消耗



8. 编写 Test

该部分由于时间紧迫，暂时仅仅测试了返回合约信息的部分，详情请查看 https://github.com/Lamken/answer_me

9. 参考链接

1. GETTING STARTED WITH DRIZZLE AND REACT (https://truffleframework.com/tutorials/getting-started-with-drizzle-and-react)

2. BUILDING DAPPS FOR QUORUM: PRIVATE ENTERPRISE BLOCKCHAINS (https://en.wikipedia.org/wiki/SHA-1)

3. ETHEREUM PET SHOP -- YOUR FIRST DAPP (https://truffleframework.com/tutorials/pet-shop)

4. 知乎 (主要 UI 参考) (https://zhihu.com)