重庆大学

课 程 设 计

设计题目　基于以太坊的微博系统设计与实现

学生姓名　王　浩, 李涵威

学　　号　20184347, 2018xxxx

专业班级　信息安全 2 班, 信息安全 1 班

指导教师　叶春晓

2021 年 12 月 26 日

目录

一、项目简介 **1**

1.1 概述 1

1.2 传统微博与去中心化微博的对比 1

二、系统框架 **2**

2.1 运作流程 2

2.2 方案选型 2

2.2.1 以太坊客户端 2

2.2.2 开发框架 2

2.2.3 前端应用框架 3

2.3 总体设计 3

三、功能说明 **3**

四、设计思路 **4**

4.1 创建项目 4

4.2 合约 5

4.2.1 数据结构 5

4.2.2 合约方法 6

4.3 前端应用 8

五、结果展示 **8**

六、小组分工 **8**

一、 项目简介

1.1 概述

这是一个运行在以太坊上的去中心微博系统，去中心化意味着没有一个中心化机构能够控制你发送的微博，你发送的微博是由你完全控制的，任何人无法删除、关闭你的微博。一旦你的微博发出去后，只有你自己能删除它。

1.2 传统微博与去中心化微博的对比

传统微博（如新浪微博）就是一个中心化的应用平台，新浪公司就是整个微博平台的中心。新浪公司制定新浪微博的运行规则，开发出整个微博平台，为其提供中央服务器，维持着整个新浪微博的运转，并不断地向外推广，吸引用户使用。一切商业行为都是为了追逐利益的，新浪公司运营新浪微博，也是为了吸引广告主投放广告，从而获得巨额的广告收入。

在中心化的微博平台中，大致流程如图 1 所示，博主（发微博者）会编辑微博发送到新浪微博平台中，新浪微博将微博推送给观众（看微博者），观众查看微博，微博中会夹杂着一些广告，观众看微博时也会看到一些广告。广告主会为广告的浏览量和点击量，支付广告费给新浪公司。

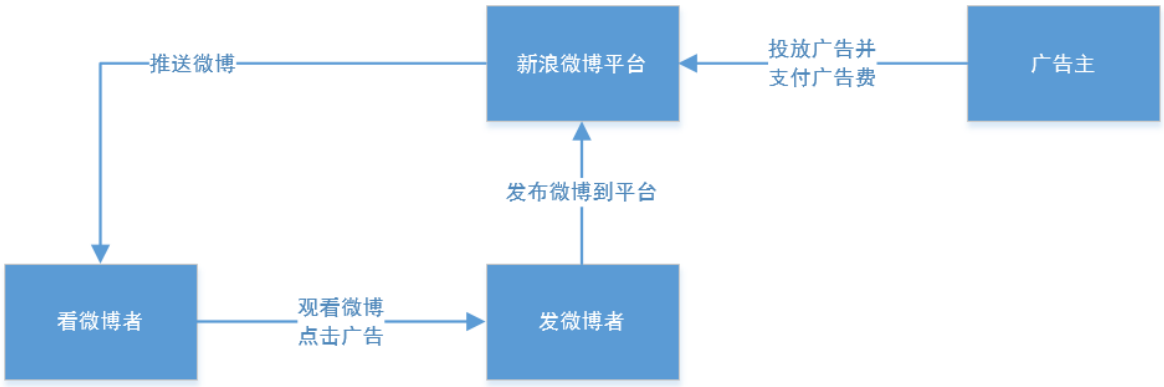

图 **1**　传统中心化微博运作流程

与传统微博平台不同，在去中心化微博平台中，将没有中心机构，没有中央服务器，主要是通过区块链技术，运用分布式自治组织（DAO）的组织架构，实现微博平台的自治。让每一个微博参与者都成为微博平台的所有者，他们将共享微博平台获得的全部收益。

二、 系统框架

2.1 运作流程

该去中心化微博系统部署在以太坊区块链上，采用智能合约作为数据存储后端，采用 Web 前端提供用户操作界面。整个系统的流程如图 2 所示。

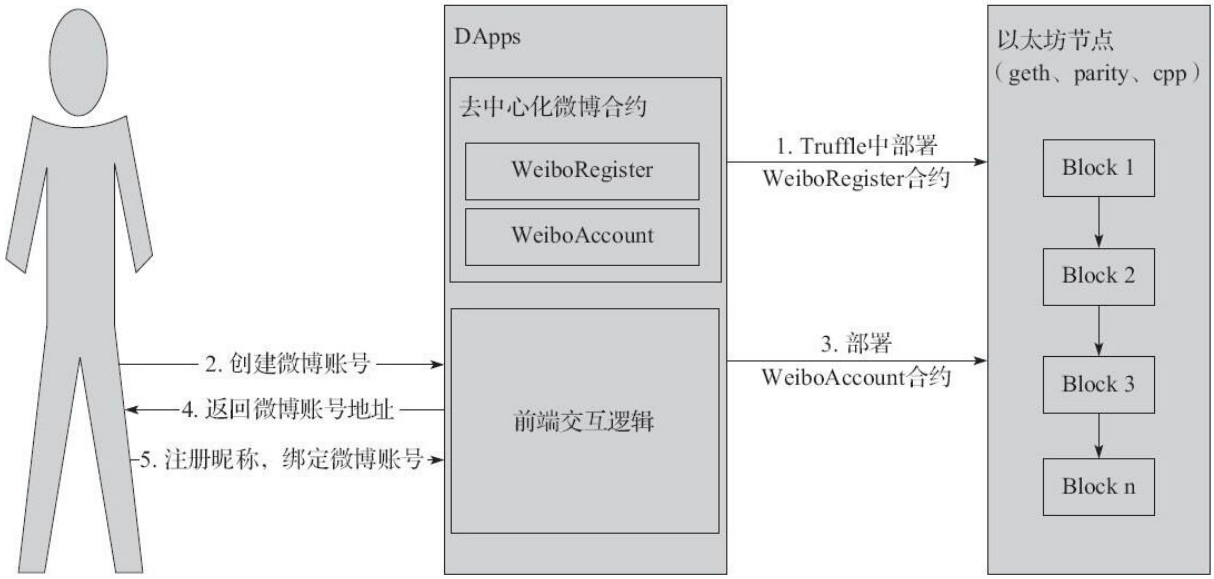

图 **2**　去中心化微博的运作流程

2.2 方案选型

2.2.1 以太坊客户端

在目前开发 DApp 去中心化应用中，Ganache 和 Geth 这两种以太坊客户端使用较为普遍，本项目可以同时运行部署在 Ganache 和 Geth 中。但是在测试开发中，比较推荐 Ganache。Ganache 是基于 Node.js 开发的以太坊客户端，整个区块链的数据驻留在内存，发送给 Ganache 的交易会被马上处理而不需要等待挖矿时间。Ganache 可以在启动时创建一堆存有资金的测试账户，它的运行速度也更快，因此更适合开发和测试。

2.2.2 开发框架

本项目使用 Truffle 开发工具。Truffle 是基于以太坊的智能合约开发工具，支持对合约代码的单元测试, 非常适合测试驱动开发。同时内置了智能合约编译器，只要使用脚本命令就可以完成合约的编译、部署、测试等工作，大大简化了合约的开发生命周期。

2.2.3 前端应用框架

本项目采用 Web 前端，基于 Webpack Truffle Box 模板搭建，并使用了 jQuery 和 Bootstrap 框架简化前端页面的开发。

2.3 总体设计

本系统底层使用以太坊区块链，用户在浏览器中使用 MetaMask 连接以太坊就可以完成所有操作，系统架构如图 3 所示。

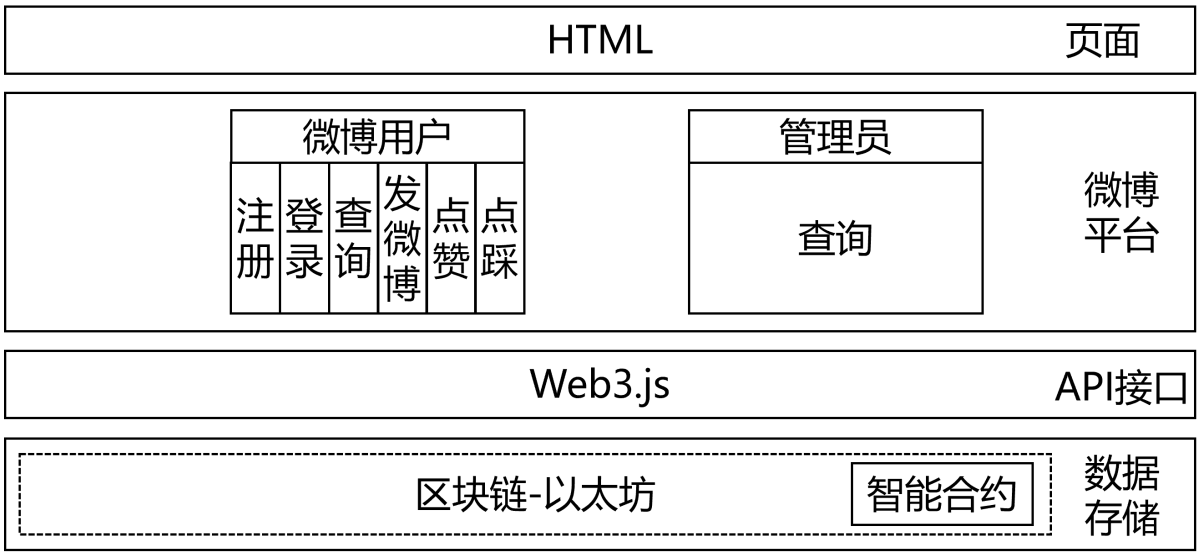

图 **3**　系统架构

三、 功能说明

本系统的核心业务围绕微博展开，其中用户可以发送微博，其他用户可以查看该微博并决定是否点赞或者点踩。系统管理员可以查看所有用户信息，例如用户的微博总数、点赞总数、点踩总数等。

本系统主要涉及两类用户：微博用户和管理员。各用户的功能如表 1 所示。从表中我们可以看到管理员除了查询系统信息外不具有其他功能，所以简单起见我们没有在系统中特别设立管理管，任何用户都可以在后台页面查看后台信息和用户列表，也就是说目前所有用户其实都拥有和管理员一样的功能。

表 1　用户功能

	需求要点	备注
微博用户	注册	使用以太坊账户注册一个微博账户，并指定一个唯一的昵称
	登录	使用以太坊账户进行登录，一般由 MetaMask 自动完成
	查询	用户查询自己的个人信息，包括 ID、昵称、历史微博等
	发微博	用户可以发送一条新的微博
	点赞	用户可以对首页看到的其他用户的微博进行点赞
	点踩	用户可以对首页看到的其他用户的微博进行点踩
管理员	查询	管理员可以查看完整的用户列表、微博总数等系统信息

四、 设计思路

4.1 创建项目

前面我们已经介绍过本项目的方案选型，我们在开发环境中使用 Ganache 作为以太坊客户端进行智能合约的部署，同时使用 Truffle 作为开发框架构建我们的项目。简化起见，我们不使用 React 项目模板，而使用 Webpack 项目模板，这样我们使用传统的前端开发技术完成前端页面的编写。

```
# 安装 Truffle
npm install -g truffle
# 创建项目目录
mkdir dapp-weibo
cd dapp-weibo
# 拉取项目模板创建新项目
truffle unbox webpack
```

项目创建好后目录结构如下：

```
|--app --------------------前端项目
|--|--src -----------------网页源码html\css\javascript
|--contracts --------------智能合约
|--migrations -------------合约部署脚本
|--test -------------------合约测试脚本
```

之后我们使用下面这些命令完成项目的构建、部署、测试、启动等：

```
# 编译智能合约
truffle compile
# 部署智能合约
truffle migrate
# 测试智能合约
truffle test
# 启动前端
cd app && npm run dev
# 构建前端
cd app && npm run build
```

4.2 合约

4.2.1 数据结构

我们需要在合约中存储用户信息和微博信息，我们将其分别定义为一个结构体，其中微博结构体需要存储微博创建的时间戳、微博内容、点赞数量和点踩数量，微博用户结构体需要存储用户 ID、用户账户地址、用户名、微博数量、所有微博。具体定义如下：

```
// 博文
struct Blog {
    uint256 timestamp;
    string content;
    bool isValid;
    uint256 like;
    uint256 dislike;
}

// 微博用户
struct User {
    uint256 id;
    address addr;
    string name;
    bool isValid;
    uint256 blogNumber;
    mapping(uint256 => Blog) blogs;
}
```

同时为了方便我们在合约方法中通过账户地址、用户 ID、用户名等各种不同的方式查询用户，我们定义了下面几个映射表：

```
// 存储用户的hash表
mapping(address => User) users;
mapping(address => string) address2name;
```

```
mapping(string => address) name2address;
mapping(uint256 => address) id2address;
```

4.2.2 合约方法

创建一个微博用户时需要检测该用户昵称是否被占用以及该账户地址是否以及注册，如果出现昵称占用或者账户以及被注册则提示注册失败，反之则在users映射表中新建一个微博用户，同时维护好所有映射关系。具体实现如下：

```
// 创建用户
event CreateUser(address sender, bool isSuccess, string message);

function createUser(string memory name) public {
    // name已经被占用
    if (name2address[name] != address(0)) {
        emit CreateUser(msg.sender, false, "昵称已被占用");
        return;
    }
    // msg.sender 已经注册过
    if (bytes(address2name[msg.sender]).length != 0) {
        emit CreateUser(msg.sender, false, "您的账户已被注册");
        return;
    }
    // 昵称长度超过64字节限制
    if (bytes(name).length == 0 || bytes(name).length > 64) {
        emit CreateUser(msg.sender, false, "昵称长度不合理");
        return;
    }
    users[msg.sender].id = userNumber;
    users[msg.sender].addr = msg.sender;
    users[msg.sender].name = name;
    users[msg.sender].blogNumber = 0;
    users[msg.sender].isValid = true;
    // 保存id到账户的映射关系
    id2address[userNumber] = msg.sender;
    // 保存账户到昵称的双向映射关系
    address2name[msg.sender] = name;
    name2address[name] = msg.sender;
    userNumber++;
    emit CreateUser(msg.sender, true, "新用户创建成功");
}
```

发微博时需要检测博文长度是否超过限制，如果超过限制则提示发送失败，否则在对应用户的blogs 映射表中新建一条微博，同时需要将微博的点赞数和点踩数都置为 0。

具体实现如下：

```
// 发微博
event PostBlog(address sender, bool isSuccess, string message);

function postBlog(string memory content) public {
    // 微博长度超过160字节
    if (bytes(content).length == 0 || bytes(content).length > 160) {
        emit PostBlog(msg.sender, false, "博文长度不合理");
        return;
    }
    users[msg.sender].blogs[users[msg.sender].blogNumber].timestamp = now;
    users[msg.sender].blogs[users[msg.sender].blogNumber].content = content;
    users[msg.sender].blogs[users[msg.sender].blogNumber].like = 0;
    users[msg.sender].blogs[users[msg.sender].blogNumber].dislike = 0;
    users[msg.sender].blogs[users[msg.sender].blogNumber].isValid = true;
    users[msg.sender].blogNumber++;
    emit PostBlog(msg.sender, true, "微博发布成功");
}
```

给指定用户的指定微博点赞/点踩时需要将用户 ID 和微博 ID 传给合约方法，方法中检测该用户的该微博是否存在如果不存在则提示失败，反之给该微博的点赞数/点踩数增加 1。具体实现如下：

```
// 给博文点赞
event LikeBlog(address sender, bool isSuccess, string message);

function likeBlog(uint256 user_id, uint256 blog_id) public {
    address addr = id2address[user_id];
    if (!users[addr].isValid) {
        emit LikeBlog(msg.sender, false, "用户不存在");
        return;
    }
    if (!users[addr].blogs[blog_id].isValid) {
        emit LikeBlog(msg.sender, false, "博文不存在");
        return;
    }
    users[addr].blogs[blog_id].like += 1;
    emit LikeBlog(msg.sender, true, "点赞成功");
}

// 给博文点踩
event DislikeBlog(address sender, bool isSuccess, string message);
function dislikeBlog(uint256 user_id, uint256 blog_id) public {
    address addr = id2address[user_id];
    if (!users[addr].isValid) {
        emit LikeBlog(msg.sender, false, "用户不存在");
```

```
        return;
    }
    if (!users[addr].blogs[blog_id].isValid) {
        emit LikeBlog(msg.sender, false, "博文不存在");
        return;
    }
    users[addr].blogs[blog_id].dislike += 1;
    emit LikeBlog(msg.sender, true, "点踩成功");
}
```

当然为了前端能够很好的给用户展示用户的个人信息，用户的微博列表，系统的用户列表等信息，我们还需要在合约中提供一系列的查询方法，例如查询用户总数、查询博文总数、根据用户 ID 查询用户信息、根据用户 ID 和微博 ID 查询微博等。由于大部分查询都是简单的将信息返回而已，所以下面不再一一列出。其中根据用户 ID 查询用户信息的方法如下：

```
// 根据id返回是否查询成功、昵称、账户、博文数、like、dislike
function getUserInfo(uint256 id)
    public
    view
    returns (
        bool isSuccess,
        string memory name,
        address addr,
        uint256 blogNumber,
        uint256 like,
        uint256 dislike
    )
{
    if (id2address[id] == address(0)) {
        return (false, "", address(0), 0, 0, 0);
    }
    address _addr = id2address[id];
    uint256 _like = 0;
    uint256 _dislike = 0;
    for (uint256 i = 0; i < users[_addr].blogNumber; i++) {
        _like += users[_addr].blogs[i].like;
        _dislike += users[_addr].blogs[i].dislike;
    }
    return (true, users[_addr].name, users[_addr].addr, users[_addr].blogNumber, _like,
        _dislike);
}
```

根据用户 ID 和微博 ID 查询微博信息的方法如下：

```
// 根据用户id和博文id返回博文信息
```

```
function getBlog(uint256 user_id, uint256 blog_id)
    public
    view
    returns (
        bool isSuccess,
        uint256 timestamp,
        string memory content,
        uint256 like,
        uint256 dislike
    )
{
    address addr = id2address[user_id];
    if (!users[addr].isValid || !users[addr].blogs[blog_id].isValid) {
        return (false, 0, "", 0, 0);
    }
    Blog memory blog = users[addr].blogs[blog_id];
    return (true, blog.timestamp, blog.content, blog.like, blog.dislike);
}
```

4.3 前端应用

五、 结果展示

六、 小组分工

我们小组共两位成员，分工情况如表 2 所示。另外我们在 GitHub 上进行项目合作，完整项目代码见仓库https://github.com/iamwhcn/dapp-weibo。

表 **2** 小组分工

姓名	学号	分工
王　浩	20184347	系统设计、报告编写
李涵威	2018xxxx	系统设计、报告编写