电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS

论文题目 DESIGN A BLOCKCHAIN-BASED DIGITAL ASSET MANAGEMENT SYSTEM THAN MONITORS THE VALUE OF AN ASSET THROUGHOUT ITS LIFE CYCLE

专　　业　电子信息工程

学　　号　2014200105026

作者姓名　申耀峻

指导教师　XX

摘　要

伴随着比特币的诞生，区块链技术开始步入人们的眼球。不同于传统的中心化服务系统，如政府，银行，大公司数据库中心。区块链由分散世界的众多不同节点共同维护，具备去中心化，数据不可篡改，数据公开等特点。传统数字资产管理系统依赖中心化媒介，资产所有权转移往往需要支付高额中介费用，且难以保证交易的有效及数据的安全。本文将基于以太坊平台，搭建资产管理交易系统，通过区块链及智能合约，可确保交易有效，资产交易历史可溯源，以及支付安全。在知识产权保护和确权等方面可发挥积极作用。

关键词：区块链，数字资产管理，智能合约，知识产权保护，以太坊

ABSTRACT

With the creation of Bitcoin, Blockchain has appeared in our view. Different from traditional centralized service system, like Bank, Government and data center of big company. Blockchain is maintained by various nodes around the world together, which is decentralized, immune to tamper and opened to public. Traditional Digital Asset Management System (DAMs) depends on centralized intermediary. Transfer of digital assets will pay amounts of intermediary fees but hard to ensure the validity of transactions and the data security. This paper intends to build a DAMs based on the Ethereum platform. With the use of blockchain and smart contract, it will ensure the validity of transaction, traceability of asset and transaction, and the payment security, which will have a positive influence on the protection and authorization of intellectual properties.

Keywords: Blockchain, Digital Asset Management, Smart Contract, Intellectual property, Ethereum

目录

第一章 绪论

1.1 项目研究的背景与意义

1.1.1 数字资产

随着互联网的迅速发展，人们产生的绝大多数信息都由过去以书信，实体媒介为载体，转变到现在通过以互联网为主要媒介来进行快速传播。与此同时，互联网上的数字资产也占据着越来越大的比重。个人在网络上发表的一篇论文，在音乐平台发布的原创音乐作品，以及个人精心拍摄的一张图片，都属于创作者受知识产权保护的数字资产。

信息互联网时代，“信息共享”精神极大丰富了互联网内容，但同时也使得侵权行为处于难以监管，无法追溯的地位。P2P 软件使得人们可以随意下载，传播未经授权的他人著作，电影，音乐，无疑将极大损害创作者利益，挫伤其创作积极性。

1.1.2 比特币与区块链

2008 年，中本聪（Satoshi Nakamoto）发表论文《比特币：一种点对点的电子现金系统》，表示由于当前的互联网贸易需要引入第三方中介支付机构，将不可避免的引起人们对中介机构的可信度怀疑。同时因交易数据的可撤销及销毁性，人们无法对在线交易凭证保持信任。因而提出建立一种不依赖第三方可信机构，完全点对点的电子支付系统，即比特币系统。这种系统具备以下几个特点：

1. 去中心化，系统由众多节点共同维护，基于工作量证明争夺记账权，个人与个人之间仅需要私钥和公钥地址即可完成交易转账，不需要任何第三方中介机构的介入。

2. 有限货币供应，比特币重量固定 2100 万枚，后续矿工主要激励将来源于交易手续费，限制了货币的通货膨胀。

3. 交易数据不可篡改，以区块为单位，每 15 分钟将当前所有未处理交易写入一个区块中，生成该区块哈希值，并将该区块的哈希值写入下一区块中，形成区块链，因此如需修改任何区块的内容，都需要重新计算该区块之后的所有区块哈希值，从个人算力的角度来看一般难以实现。

1.1.3 智能合约

比特币作为区块链 1.0 的主要应用，实现了点对点的可信电子支付。而智能合

约作为区块链 2.0 的代表，则通过图灵完备的编程语言，可用来创建，确认，转移各种不同类型的数字资产以及合约。不同于传统合约的执行依赖于合约双方的信任，以及背后法律机构和政府的保证。智能合约可根据合约双方初始预设事件，当一定条件满足时，自动触发相关函数，执行对应的条件。比如一份数字资产，在买家完成付款交易后，即自动将相关知识产权转移至买家名下，而无需第三方公正机构的介入。智能合约具备以下特点：

1. 智能合约内容可由双方共同决定，且合约代码数据保存于区块链之上，双方可随时查看，保证公开透明性。
2. 历史数据作为区块链的一部分，不可篡改，但可通过预设函数进行更新，且所有更新操作和数据都记录在区块链上。
3. 自动执行，但达到触发条件时，合约相关条款自动执行，无需第三方介入。

1.1.4 知识产权保护

目前，互联网上大量的知识产权的侵权问题都难以得到妥善处理。原因在于数字资产的维权存在取证困难，时间周期长，成本过高，赔偿过低等问题。另一方面，版权确权问题也困扰着创作者。有些作品需要创作者经过法律登记才能具备完整著作权，而未经登记的作品在遭受侵权行为时，很容易存在确权难的问题。同时，个人知识产权缺少可信任的版权授权及出售平台，创作者难以从作品中直接获取收益，也不利于知识产权的合法流通。因此，如果能结合区块链进行版权登记与版权转移，无疑将极大的降低知识产权的保护成本以及使用效率，更好的促进信息社会的发展。

1.2 本文主要创新

本文将基于以太坊系统，通过智能合约，开发数字资产管理及交易平台。本平台将具备数字资产登记，授权，共享，转移，注销等功能，并基于每一份资产，记录该资产生命周期内所有交易数据和使用情况，使人们可轻易对资产进行溯源与维权

1.3 本论文的结构安排

第一章简要介绍了本课题的意义和背景。

第二章主要阐述了该系统背后的技术原理，包括以太坊平台技术架构，智能合约原理，以及前端开发架构选择，使得对区块链系统以及前端交互有一定了解。

第三章主要描述了系统所需要的功能以及初步架构需求。

第四章将主要以流程图的形式描述该系统的整体架构，使得对每一个开发模块的功能和应用具备清晰的认识，并使用代码描述具体实现。

第五章讲述了系统目前所做的单元测试，保证系统在修改过程中功能的完好。

第六章对项目进行总结，分析不足和有待改进之处。

第二章 技术背景及相关架构原理

2.1 以太坊

2.1.1 以太坊介绍

以太坊是一个开放的区块链开发平台，任何人都可以在以太坊上开发或使用基于区块链技术的去中心化应用。就像比特币一样，只是一个分布式系统，没有特定的人或组织可以控制它，同时不同于比特币仅作为点对点的支付系统，以太坊不仅具备支付能力，同时设计了以太坊虚拟机（EVM），可支持智能合约在系统上的运行，从而大大扩展了区块链技术的应用范围。

以太坊开发团队预先编写好了开发去中心化应用所需的基本工具和编程语言，以及相关接口方法。如 go-Ethereum 可运行完整的以太坊节点以及搭建私有链，Ganache 可运行本地测试链，Solidity 作为图灵完备的智能合约编写语言，以及运行环境 EVM。同时本项目主要运用 Web3.js，来通过前端页面调用智能合约函数。这些工具的完备保证了对区块链底层细节的封装，从而使得开发者可专注于应用核心功能的开发。

2.1.2 Merkle 树

以太坊是一个基于状态机的系统，当前的系统便是由所有状态，包括账户，交易，收据所组成。其系统状态保存于三棵改良 Merkle 树上，即世界状态树，交易树，收据树。在世界状态 Merkle 树上，一个叶子节点保存了一个数据块的哈希值，而一个数据块保存了一个账号的相关数据。父节点则保存了所有子节点所存哈希值总和的哈希，即 ParentHash=hash(sum(ChildHash))。所以 Merkle 树也被称为哈希树

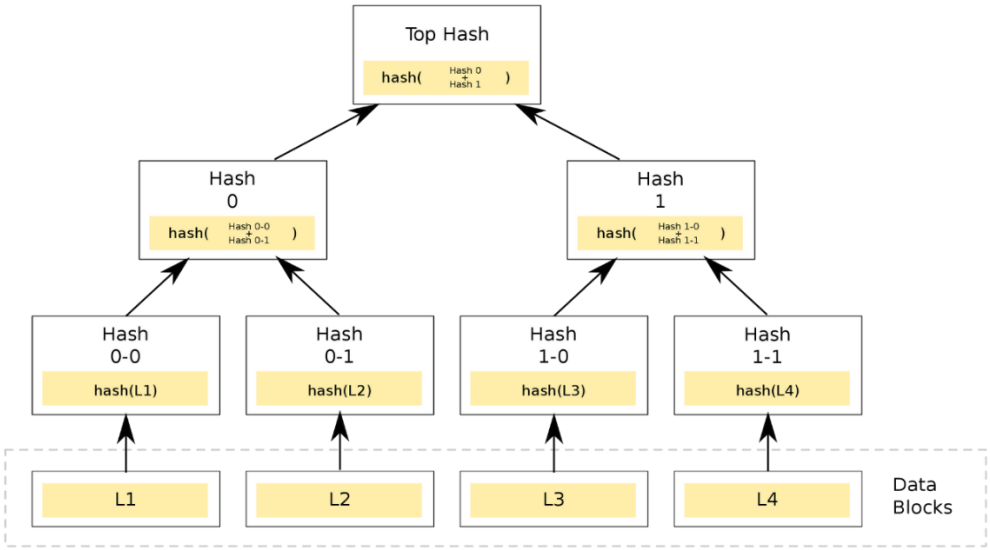

图 2-1 Merkle 树原理图

在一棵哈希树上，任何一个叶子节点的数据发生改变，都将导致其父节点的哈希值发生变化，而因为每一个父节点的哈希值都受其子节点所影响，最终会导致叶子节点的所有相关父节点，乃至根节点的哈希值都发生改变。因此，任何一个账号数据的改变，都将使得根节点哈希值发生改变。基于这一特点，Merkle 树具有了两个重要特性：

1. 我们不需要通过对比每一个叶子节点来确保数据未被篡改，而仅仅通过对比根节点即可做到以一点。
2. 我们可以通过一致性检测保证特定的数据被正确的保存在树上而不需要对整棵树进行检索。

第一个特点使得我们仅需将根节点哈希值保存在区块头上，就可以保证过去某一区块所对应的状态未被篡改，而不需要将所有数据状态保存于区块链中。而第二点则使得在分布式多节点系统中可以快速对账号信息进行校验。而无需一一查找对比。

2.1.3 账号状态

不同于比特币使用的 UTXO（未用交易输出）模型来对地址（账号）余额以及交易进行管理结算。以太坊使用传统的基于账号的方式来对账号数据进行管理。世界状态由所有账号的当前状态信息，包括余额，随机数，存储数据，代码而组成。并通过账号地址与账号信息进行对应。世界状态并不直接存储于区块链上，而是通

过世界状态树储存，并将根节点写入当前区块中。从创世区块开始，世界状态中账号的每一次更新都通过“交易（Transaction）”来实现，包括转账以及合约数据存储。

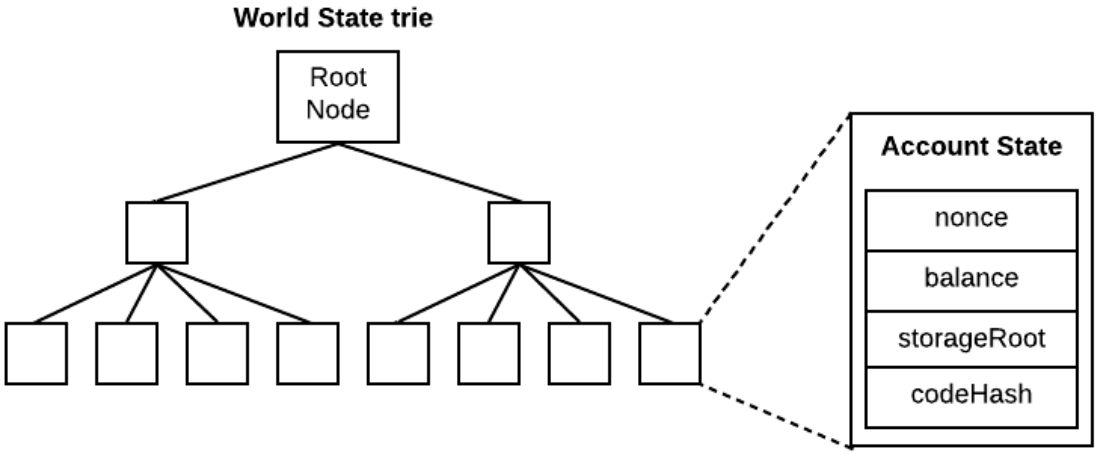

图 2-2 世界状态树结构图

在以太坊中存在两种账号类型，外部账号以及合约账号。外部账号可用于个人，组织，以及公司账户，使用公钥作为账号地址以及私钥作为密匙，可用于交易以太币以及部署智能合约。合约账号则是合约部署的地址，每一份智能合约都具有独一的账号地址。

合约账号的状态包含了合约运行需要的所有信息，包括该账号所具有的以太币以及与该合约相关的交易。外部账号和合约账号的状态具有以下信息：

1. 随机数（Nonce）

对于外部账号表示从该地址发出的交易总数，对于合约账号表示通过该合约创建的其他合约数量。

2. 余额（Balance）

该账号具有的以太币余额。

3. 存储根哈希（Storage Root）

保存该账号存储树的根哈希值，仅合约账号具有存储树，用以保存合约内的存储变量, 树中的变量可通过调用智能合约函数实现修改，外部账号该值为空。

4. 代码哈希（Code Hash）

保存合约编译后的二进制代码的哈希值，外部账号为空

对于账号而言，除了代码哈希，其他状态都是可变的，比如一个账号发送以太币到其他账号，随机数会增加，账户余额也会根据交易发生变化。而代码哈希的不可变则保证了合约一旦部署，没有任何人可以修改该合约的代码内容，从而防止在部署后他人对合约进行篡改。如果合约需要进行更新或是发现漏洞，将需要部署一

份新的合约，原合约需要预先定义清除函数并调用才能清楚合约数据，否则将永远保存在以太坊上。因此在部署前需要在 Ganache 进行部署测试确保合约的正常运行。

2.1.4 交易

在以太坊中，任何状态的更新都通过交易实现，交易的类型有三种：

1．两个外部账户之间的转账，该交易改变了发送和接受账号的余额状态。

2．外部账号部署合约，创建新的合约账号。

3．外部账号调用合约函数，并修改合约内存储变量。

所有类型的交易都需要支付一定的 Gas，对于转账而言，支付 Gas 高的交易将会被矿工优先处理，因为矿工可以获得当前区块所有交易的 Gas 作为挖矿收益。而对于部署智能合约以及调用合约而言，Gas 则与合约需要存储的数据，以及合约代码所需要的操作数相关。举个简单例子，一个 256 位的 int 数保存在账户存储树中需要消耗 20000 个 Gas，而一个 ADD 操作，比如 int a += 1，需要 3 个 Gas。代码执行过程中，临时变量存储于内存（memory）中不消耗 gas，但在函数执行完毕后自动清除，只有存储在存储树中的变量可以永杰保存。总得来说，交易可以定义为任何需要存储，或是修改以太坊中数据状态的行为都称之为“交易”，而“交易”总是需要支付一定的 Gas 来完成的。每一笔交易包含以下信息：

1. Nonce

即发送账户的 nonce 值，表示发送账户的历史交易总数

2. GasPrice

该交易的 Gas 单价，根据当前以太坊网络情况动态调整

3. Gaslimit

该交易设定的 Gas 上限，即该笔交易最多可消耗的 Gas，如耗尽但未完成交易处理将导致交易失败，且消耗的 Gas 值不会返还。若为部署智能合约或调用智能合约函数，未消耗的 Gas 将返回发送者账户。

4. To

交易的对象。如果是以太币转账，则为接受地址。如果是部署合约，则值为空。如果是调用合约函数，则为合约地址。

5. Value

该交易发送的以太币数量。可发送到外部账户或合约账户。

6. Data

调用合约函数或部署合约时所需的数据，将作为参数传入构造函数或其他函数，交易对象为外部账户时为空。

7. Init

初始化合约所需的二进制合约代码，仅用于创建合约。

2.1.5 交易收据

在每个区块时间内，矿工会收集并打包当前发生的交易，率先获得记账权的矿工将获得交易的 Gas 费用以及挖矿奖励，并将当前打包的交易记入区块体。在交易所在区块被最终确认后，该区块所包含交易会插入交易树中，并将该交易所生成的收据插入收据树中，一份交易的收据包含以下信息：

1. 区块哈希（blockHash）

当前交易所在的区块哈希值

2. 区块数（blockNumber）

当前交易所在的区块编号

3. 交易哈希（transactionHash）

当前交易内容的哈希值

4. 交易编号（transactionIndex）

该交易在当前区块中的编号

5. 来自（from）

交易发起者的地址

6. 交易对象（to）

交易的接收者，创建智能合约时该值为空。

7. 累计使用的 Gas（cumulativeGasUsed）

该交易以及同一区块中该交易之前的所有交易所消耗 Gas 的总和。

8. 使用的 Gas（gasUsed）

该交易所消耗的 Gas.

9. 状态（status）

该交易的处理结果，0 表示交易失败，1 表示交易成功

10. 合约地址（contractAddress）

创建合约时返回的合约地址，其他交易类型时为空。

11. 日志（logs）

该交易附带的日志信息。

2.1.6 区块链

区块链由区块组成，而每一个区块由区块头和区块体组成。区块头中记录了当前区块所有相关信息，以及其父区块的哈希值，从而保证了区块链的不可篡改。而区块体中则记录了当前区块所包含的交易以及 uncle 区块头。下图包含了上述所有内容互相之间的关系。

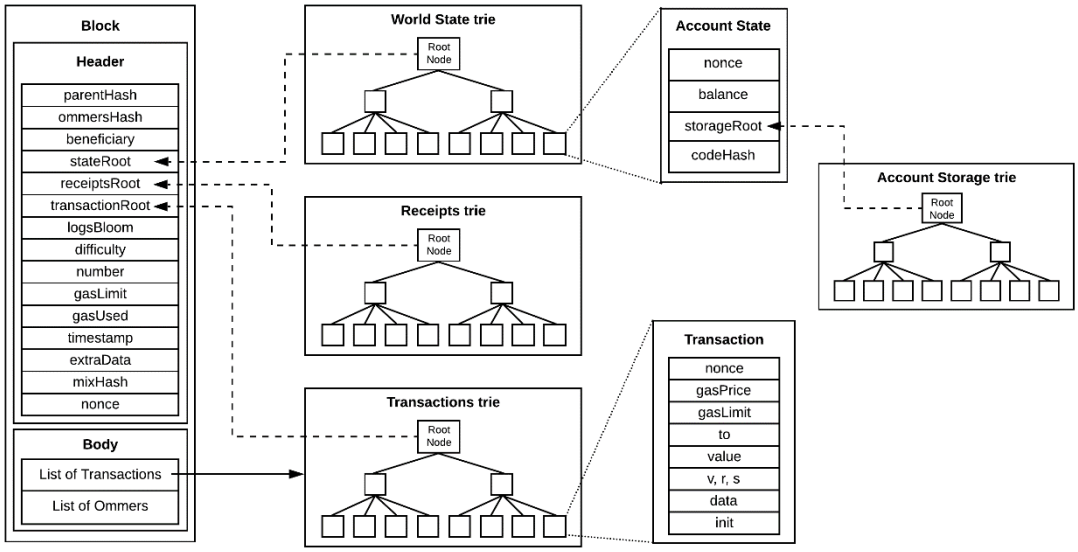

图 2-3 以太坊区块链整体结构图

区块头中保存了世界状态树，交易树，以及收据树的根节点哈希值，而这三棵状态树并不直接存储于区块链之上，而是根据已生成区块的交易内容进行实时更新。因此每一笔交易需要等待所在区块得到确认后才能生效，所需平均时间大概在 15 秒左右。同时对账户以及交易状态的查询是通过对应状态树进行，所有也只有等待区块生成后才能进行查询和下一步操作。

2.1.7 智能合约

比特币作为第一代区块链应用，仅支持将区块链作为分布式账本，来对地址之间的比特币交易进行记录。而以太坊系统的主要特点，便是对智能合约的广泛应用。以太坊智能合约使用 Solidity 语言进行编写，该语言具备面向对象，以及图灵完备的特性，可根据开发者需要编写完整的去中心化应用程序。通过 Solidity 编译器对智能合约进行编译后，可在以太坊虚拟机（EVM）运行，并生成应用二进制接口（ABI），可供 Web3 在前端调用合约函数，进行合约交互操作。

Solidity 作为静态语言，支持多种变量类型，并可通过 is 关键字实现合约继承。Solidity 主要包括两种存储方式，storaged 持久化存储，以及 memory 临时存储。通过一个简单例子可以了解合约的编写。

```
pragma solidity ^0.4.25; //定义合约所用编译器版本
// 定义合约 Mortal
contract Mortal {
    address owner; //定义地址变量，该变量默认为 Storage 存储。

    // 合约构造函数，在创建合约时自动调用
    constructor() public {
      owner = msg.sender; // msg.sender 表示当前交易发起账户，在创建合约时为合约创建者，该变量作为 memory 类型临时保存，合约创建完毕后清除。
    }
    // 自毁函数，在合约终止后调用将清楚该合约账户内所有数据，如未定义该函数，合约将永久保存。
    function kill() public {
      if (msg.sender == owner) selfdestruct(msg.sender); // 调用 selfdestruct 后将合约内以太币余额转入 msg.sender,即调用该函数的账户。
    }
}
// 合约 Greeter 继承自 Mortal，该子合约将拥有父合约所有存储变量和函数。
contract Greeter is Mortal {
    string greeting; // 定义 Storage 存储的字符串变量
    // 子合约构造函数，传递参数为 memory 存储，函数调用完成后将清楚。
    constructor(string memory _greeting) public {
        greeting = _greeting;
    }
    // 只读函数，不需要进行任何交易，仅返回 greeting 变量值
    function greet() public view returns (string memory) {
        return greeting;
```

```
    }
}
```

在合约中，仅有变量 owner 以及 greeting 将保存在合约账户 Storage 树中，其余传递的参数将在交易完成后清除。同时合约部署后将把二进制代码写入账户 code 段内，并无法修改。合约内的只读函数不修改任何合约数据，只读取并返回合约内变量，因此该行为不作为“交易”，不需要消耗 Gas。

在本项目中，将围绕智能合约进行数据存储，数据变更等操作，来完成数字资产的管理和交易。

2.2 前端编程

该项目通过 Web 应用的方式提供用户服务，因此需要搭建前端工程页面，以及后台数据存储来支撑项目的运行。

2.3.1 Node.js

基于 Chrome 浏览器 V8 引擎的 Javascript 运行环境，联合 NPM 可方便进行 JS 模块管理和应用部署。在前端的开发中需要用到大量框架以及模块，如 React,Express,Web3 等，都可以通过 Node 进行下载和引用。并可通过安装 mocha，对项目模块功能编修单元测试，保证程序功能与应用编程接口的正常运行。

2.3.2 React.js

JS 界面开发框架。通过模块化开发，可将页面模块组件进行复用，并可通过 State 对组件内数据状态进行管理。同时，各组件之间可通过 props 传递参数，现实组件，乃至不同页面内的通信。配合丰富的 UI 库，可快速搭建标准化前端应用，并方便进行 DOM 操作。

2.3.3 Express.js

Web 应用开发框架，提供基本 Web 开发 API，包括设置中间件来响应用户请求，设置 cookie，定义页面路由等。本项目中将通过 Express 进行后端文件上传操作。

2.3.4 Web3.js

以太坊 JavaScript 应用编程接口，包含了与以太坊节点进行交互的函数库，可

用于部署智能合约，调用合约函数，发送以太币等交易。Web3 需要连接到特点的节点，如以太坊网络或测试链进行交互。

2.3 本章小结

本章首先介绍了以太坊开发相关的技术背景，从而在开发过程中可以对以太坊系统的运作方式具有清晰的认识，之后介绍了前端开发过程中需要使用的框架。通过了解框架，在开发过程熟练可以有效提高开发效率，避免重复实现已有模块。

第三章 系统需求分析

3.1 需求概况

本文需要通过建立数字资产管理系统，满足用户进行数字资产注册，数字资产转移等需求，作为一个去中心化应用，任何人都不能修改资产合约内容，通过智能合约保证了链上资产的所有权合法性和安全性，本项目核心需求基于以下几点建立：

1. 实现资产的所有权转移
2. 实现资产的访问或使用授权
3. 监视资产的整个生命周期，从资产建立到归档
4. 在不同个体之前共享资产所有权
5. 监视资产数据的使用情况。

作为 Web 应用，用户需要能够通过 Web 前端页面完成上述功能以及操作，并将资产交易数据记录于区块链上。

3.2 智能合约

该项目需要通过创建智能合约，完成资产注册，交易，授权，共享股权等功能，并将资产信息及交易记录保存于区块链智能合约之中。智能合约需要提供预设函数接口，使得用户可通过前端页面调用合约函数，完成资产注册交易等相关操作。所需功能如下：

1. 资产注册

用户需提供以下信息进行资产注册：资产名称，编号，类型，资产总价，使用授权价格，初始所有者，初始各所有者所占股权。工厂智能合约将通过以上信息创建数字资产智能合约，并初始化合约信息。

2. 资产详情查询

创建只读类型函数，调用该函数将可返回资产当前信息：资产名称，编号，类型，资产总价，使用授权价格，当前所有者，各所有者所占股权，以及已授权使用者，供前端页面展示。

3. 资产转移

他人通过输入对应资产所有人以及所需购买的股份，并支付相应的代币到合

约地址。为防止双重交易，即在一笔交易进行时发起另一笔交易，需要在一笔交易进行时禁止其他交易的建立，并需要卖家确认交易或取消交易才能使得交易最终完成。并根据交易结果，成功或取消，将买家支付到合约地址的代币转移给卖家或退回卖家账户，同时减去卖家对应所持股份，增加到买家地址上。

4. 访问授权

需要获取该资产使用权的买家，可通过填写资产用途并支付使用费用到合约地址，智能合约在完成交易后将该买家地址填入已授权访问列表。

5. 提取利润

通过出售使用权获取的利润保存在智能合约地址之中，可通过创建利润提取函数，按照资产所有者及相应股份支付到所有者对应地址。例如，资产利润为 10 个以太币，所有者张三和李四分别享有 30%和 70%股权，则将转移给张三 3 个以太币，李四 7 个以太币。

3.3 前端页面

用户需要通过前端网页进行资产所有相关操作，因此需要用户预先安装以太坊浏览器钱包来进行注册交易等操作，如用户未安装，则需通过连接以太坊节点，为用户提供合约只读数据，展示所有资产以及对应资产详情，但无法进行资产注册交易操作。前端主要页面需求分为以下几块

1. 主页

提供该系统简单介绍，并提供链接跳转至其他相应页面

2. 资产浏览

通过工厂合约，获取所有已注册资产，并将对应资产简介展示在该页面上，并提供对应链接跳转至该资产详情页面。

3. 资产注册

上传资产文件，提供相应资产信息，将数据通过 Web3 接口发送到工厂合约创建新的资产合约，并返回资产合约地址，展示到浏览页面中。

4. 资产详情

通过合约地址调用合约获取详情函数，得到该资产当前信息，展示于页面中。

5. 资产交易

所有权交易：输。入卖家地址以及对应股份，支付对应代币发起交易。买家确认或取消后交易完成。

使用权授权：输入用途，支付使用费用，获取资产文件

6. 资产交易数据

展示资产历史所有交易数据，每笔交易包括：交易类型，用途，交易价格，转移股份，买家，卖家，交易时间，交易状态。

通过上述页面，最终可以实现用户在网页上对数字资产进行所有操作，并获取交易结果，以及实时展示资产历史交易详情。

3.4 本章小结

本章首先分析了该系统的核心需求，从而设计智能合约所需要实现的功能。再从前端界面的角度分析了如何通过网页与合约进行交互。通过本章的需求分析，明确了下一阶段的开发方向和脉络，使整体项目具有条理性。

第四章 系统实现与设计

4.1 智能合约编写

该系统基于区块链开发，为了保证资产核心数据的不可篡改以及永久保存。将把资产注册信息，交易信息，所有人，以及授权访问人信息存储于区块链上

4.1.1 工厂合约

本项目将通过预先部署工厂合约 DigitalAssetFactory，再调用工厂合约内的部署函数，来部署对应的资产合约。工厂合约作为一份完整的合约，包含了资产合约的所有代码，在部署资产合约时，将资产合约对应代码写入新的合约地址，完成资产合约的部署，在工厂合约中定义如下函数

```
contract DigitalAssetFactory{
    function createDigitalAsset(
        string initialName,
        string initialID,
        string initialType,
        uint initialPrice,
        uint initialAccessPrice,
        address[] initialOwnerAddress,
        uint[] initialShare,
        string initialDescription
        ) public{
        address newDigitalAsset = new DigitalAsset(
          initialName,
          initialID,
          initialType,
          initialPrice,
          initialAccessPrice,
          initialOwnerAddress,
          initialShare,
          initialDescription);
    }
```

}

当需要注册资产时时，调用 createDigitalAsset 函数，并传入资产注册初始信息，将自动生成一份新的智能合约 DigitalAsset，并返回该合约地址。每一份 DigitalAsset 合约代表一项数字资产，合约内存储了初始化提供的所有信息以及定义的相关操作函数。用户可以预先查看工厂合约以及资产合约的内容，来保证代码不存在漏洞，或可被人恶意操控。通过工厂合约部署，可以避免用户私下进行部署时对资产合约内容进行篡改，从而保证了所有资产合约的安全性与公平性。以这种方式部署的所有资产合约将具备相同的代码，但将不包括工厂合约的任何代码，并通过将所有资产合约地址保存在工厂合约中，从而方便对资产合约进行检索和遍历

4.1.2 资产合约

通过调用工厂合约部署资产合约后，资产合约将通过构造函数进行合约创建。

```
constructor (
 string initialName, // 资产名称
 string initialID, // 注册编号
 string initialType, // 类型
 uint initialPrice, // 资产总价值
 uint initialAccessPrice, // 使用授权价格
 address[] initialOwnerAddress, // 资产所有者
 uint[] initialShare,// 每个所有者对应所持有股份
 string initialDescription // 描述信息
 ) public {
 // 将传递的所有参数写入合约存储变量中
 AssetName = initialName;
 AssetID = initialID;
 AssetType = initialType;
 AssetPrice = initialPrice;
 AccessPrice = initialAccessPrice;
 creationTime = now;
 OwnerAddress = initialOwnerAddress;
 Description = initialDescription;
```

```
    // 使用 map 结构（类似于哈希表），将所有者地址与所持股份相对应
    for (uint i=0; i<initialOwnerAddress.length; i++) {
      OwnerShare[initialOwnerAddress[i]] = initialShare[i];
    }
  }
```

通过将所需资产初始化参数资产名称，注册编号，类型，资产总价值，使用授权价格，资产所有者，所有者对应所持股份，以及该资产描述信息传入构造函数，并将注册数据存储在约之中。资产合约通过对应地址和所持股权份额，实现了数字资产的股权共享，并可按照股权份额获取相应收益。

4.1.3 资产交易记录

通过在合约中定义一个结构体，保存与一笔交易相关的数据，并将该结构体放入交易记录数组中。通过遍历该数组即可获取该资产相关联的所有交易信息。

```
  struct Transaction{
    string transactionType; //交易类型
    string usage;// 资产用途
    uint transactionPrice; // 交易价格
    uint transactionShare; // 交易的股权份额
    address buyer;// 交易买家
    address seller;// 交易卖家
    uint transactionTime;// 交易时间，unix 时间戳
    string status;// 交易状态
  }
  // 交易记录数组，用于存放所有交易记录
  Transaction[] public transactions;
```

交易类型包含两种，资产使用授权交易以及股权转移交易。资产用途用于在使用授权交易中记录资产的使用方式。交易的股权份额用于在显示股权转移交易中转让的股权份额。交易时间显示交易所在区块确认的时间，也就是交易在区块链上最终确认的时间，而非交易发起的时间。交易状态分为取消，待定，以及成功三种，表示当前交易是否完成或取消。

4.1.4 资产交易

对数字资产的交易以股份交易的形式进行，需要传递购买股权的对象，购买的

股权份额，以及该交易用途三个参数。并发送购买该股权份额所需的以太币。代码如下：

```
// 资产股权份额转让交易
function ownershipTransaction(string usage, address seller, uint
share) public payable{
      require(OwnerShare[seller] >= share); // 确保卖家股份大于等于买家购
买股份
      uint sharePrice = (AssetPrice*share)/100; // 确定该股份售价
      require(msg.value == sharePrice); // 确定金额正确
      // 确定当前资产无待定交易
      require(Buyer == 0);
      require(Seller == 0);
      // 记录交易信息
      Transaction memory newTransaction = Transaction({
        transactionType: "OwnershipShare Transfer",
        usage: usage,
        transactionPrice: sharePrice,
        transactionShare: share,
        buyer: msg.sender,
        seller: seller,
        transactionTime: now, //以 unix 时间戳形式保存，在前端转换为当地时
间显示
        status: "Pending" // 交易进入待定状态，等待确认或取消
      });
      // 设定买家和卖家
      Buyer = msg.sender;
      Seller = seller;
      transactions.push(newTransaction);// 添加交易到交易记录数组中
    }
```

每一笔交易进入待定状态后，买家的以太币会进入合约账户中，为了避免产生双重支付，或股权变更导致原有待定交易失效，该资产合约将无法处理新的股权交易。待对应股权转让者进行交易确认或取消操作后，才能进行新的交易。

交易确定函数：

```
function checkTransaction() public {
  require(Buyer != 0); // 要求当前存在待定交易
  require(msg.sender == Seller);// 要求该函数调用者为股权卖家

  Seller.transfer(transactions[transactions.length-
1].transactionPrice); // 向卖家转账

  OwnerShare[Buyer] = OwnerShare[Buyer] +
transactions[transactions.length-1].transactionShare; // 买家获得所售的
股份
  OwnerShare[Seller] = OwnerShare[Seller] -
transactions[transactions.length-1].transactionShare; // 卖家减去所售股
份

  // 将买家加入股东数组中
  OwnerAddress.push(Buyer);
  Buyer = 0; // 清空当前买家
  Seller = 0; // 清空当前卖家

  transactions[transactions.length-1].status = "Success"; // 修改交
易状态为成功
}
```

函数要求调用者为交易的卖家，并验证当前是否存在交易。调用函数后，合约内的交易金额将转入卖家账户，同时减去卖家所出售的股权，并在买家地址上增加相应股权。最后将买家加入股东数组，清空当前交易双方地址信息，并将交易状态修改为成功。代表当前交易已顺利完成。

如果卖家对当前交易有异议，也可通过调用取消交易函数来取消当前交易。

交易取消函数：

```
function cancelTransaction() public {
 require(Buyer != 0);// 要求当前存在待定交易
```

```
        require(msg.sender == Seller);// 要求函数调用者为当前交易卖家

        Buyer.transfer(transactions[transactions.length-
1].transactionPrice); // 将钱退回买家

        transactions[transactions.length-1].status = "Cancelled"; // 修改
交易状态为取消

        Buyer = 0; // 清除交易双方地址信息
        Seller = 0;
    }
```

调用该函数后，将清除合约中的交易地址信息，将交易金额退回买家，并将交易状态修改为取消。取消的交易数据依然永久保存在交易数组中。

4.1.5 资产使用授权

通过一个账户发起获取使用授权的交易，并支付使用价格后，将该账户地址计入使用授权数组，并返回资产下载链接。交易函数如下：

```
    function accessTransaction(string usage) public payable{
     require(msg.value == AccessPrice);// 要求支付的金额与使用价格相等
     // 记录交易信息
     Transaction memory newTransaction = Transaction({
       transactionType: "Access Authorization",
       usage: usage,
       transactionPrice: AccessPrice,
       transactionShare: 0,
       buyer: msg.sender,
       seller: address(this),
       transactionTime: now,
       status: "Success" // 交易状态为成功
     });

     transactions.push(newTransaction); // 将该笔交易加入交易记录数组
     accessUser.push(msg.sender); // 将买家加入使用授权列表
```

```
    }
```

不同于股权转让交易，使用授权交易因不涉及他人股权的转移，在买家发起后将直接进入成功状态，而无需其他人进行确认。交易完成后，买家将被授权访问和使用该资产。交易金额将作为资产利润，转入合约账户。

提取利润函数：

```
function takeOutMoney() public {
      // require(msg.sender == )
      uint total = address(this).balance; // 该合约利润总额

      // 根据各股东对应资产股权份额，转让对应利润
      for(uint i = 0; i<OwnerAddress.length; i++){
        address currentOwner = OwnerAddress[i];
        uint share = OwnerShare[currentOwner];
        currentOwner.transfer(total*share/100);
      }
}
```

调用该函数，将根据各股东拥有股权的份额，分配合约账户内利润。

4.2 智能合约编译

智能合约编写完成后，需要编译并部署到以太坊网络上才能进行交互。

合约编译流程：

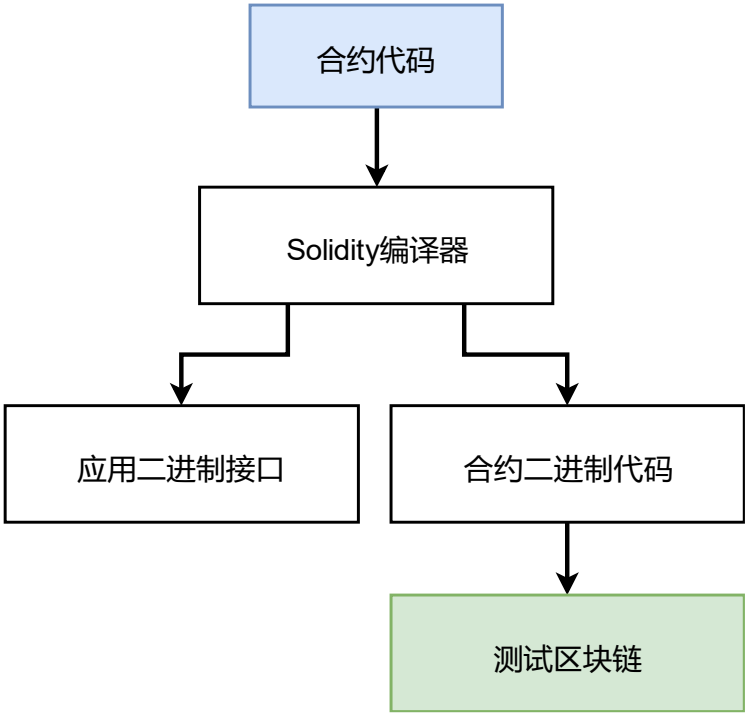

图 4-1 智能合约编译流程

合约通过在本地通过 solc 编译器进行编译，输出应用二进制接口以及二进制代码文件，再将代码部署到测试网络上。合约编译代码如下：

```
const source = fs.readFileSync(contractPath,'utf8');// 读取总合约
const output = solc.compile(source, 1).contracts;// 编译总合约

// 总合约内包含工厂合约以及资产合约，编译后将输出两份合约二进制代码
for (let contract in output){
    fs.outputJsonSync(
        path.resolve(buildPath, contract.replace(':','') + '.json'),
        output[contract]
    ); // 分别输出两份合约编译结果到不同文件
    console.log(contract + "deployed success!");
}
```

合约编译后将生成两份合约二进制代码，工厂合约文件包含工厂合约编译代码以及资产合约编译代码，资产合约文件将仅包含资产合约编译代码，这是因为在调用工厂合约部署资产合约时，需要完整的资产合约代码来进行部署，且无法从外界调用。因此资产合约代码将作为工厂合约二进制代码的一部分保存在工厂合约账户上。

4.3 智能合约部署

在合约编译后，需要预先将工厂合约部署到区块链上，之后再通过调用工厂合约函数生成新的资产合约。首先需要通过创建 Provider，连接到测试网络，再传入 Web3 构造函数中，实例化一个 web3 对象，用于部署智能合约，流程如下：

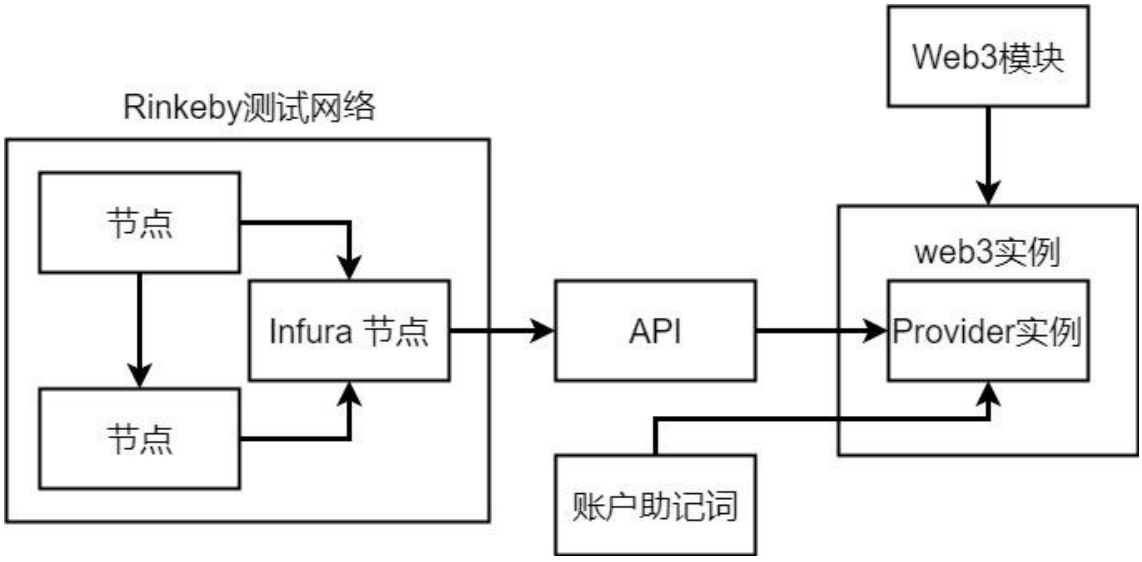

图 4-2 使用 Web3 部署合约到测试网络

创建 web3 实例代码：

```
const HDWalletProvider = require('truffle-hdwallet-provider');
// Provider 模块
const Web3 = require('web3'); // Web3 模块
const compiledFactory =
require('./build/DigitalAssetFactory.json');// 工厂合约二进制代码

// 创建 Provider
const provider = new HDWalletProvider(
    'sense wrong cross promote rate elite account security aware
okay raise replace', // 账户助记词

'https://rinkeby.infura.io/v3/c41969c6ec9c4f719a4b87bec30f28ed'
// 测试网络 API 接口
);
```

```
// 通过添加 Provider，实例化 Web 对象
const web3 = new Web3(provider);
```

完成 web3 实例创建后，通过该 web3，可将工厂合约部署到 Provider 所连接的测试网络上，部署代码如下：

```
//合约部署函数
const deploy = async () => {
    const accounts = await web3.eth.getAccounts();// 获取部署账户
    const result = await new web3.eth
        .Contract(JSON.parse(compiledFactory.interface))
        .deploy({ data: compiledFactory.bytecode })
        .send({ from: accounts[0], gas: '3000000' });
    };
deploy(); // 执行函数
```

函数部署需要分别执行三个 web3 方法，Contract 方法中传入需要部署合约编译后的 ABI，deploy 中传入该合约编译后的二进制代码。部署合约需要通过交易实现，因此 send 方法中需要传入交易发起账户地址，以及 Gas 上限。部署交易所在区块得到确认后，将返回该交易收据，其中包含了合约部署的地址。之后可使用该地址调用工厂合约函数。

4.4 智能合约调用

工厂合约部署后，如果需要在前端调用该合约函数，还需要在本地通过合约 ABI 以及 web3 实例来创工厂合约实例来调用相关函数。首先需要创建 web3 实例。

```
import Web3 from 'web3'; //引入 web3 模块


if (typeof window !== 'undefined' && typeof window.web3 !== 'undefined'){
    // 如果当前浏览器安装有 Metamask,将会在 window 对象中添加 web 对象，通过这个 web3 对象的 Provider 可以进行账户交易操作
    web3 = new Web3(window.web3.currentProvider); //创建 Web3 实例
} else {
    // 当前浏览器未安装 Metamask，需要连接其他节点提供的 API 作为 Provider，这种方式只能读取合约数据，而无法进行任何交易。
    const provider = new Web3(new Web3.providers.HttpProvider(
```

```
        'https://rinkeby.infura.io/v3/15cddac9c13945ddacad93d708c672b5'
    ));
    web3 = new Web3(provider); //创建 Web3 实例
}
export default web3; // 导出 web3 实例，可在创建合约实例时引入
```

未安装浏览器钱包 MetaMask 的用户将无法使用账户进行交易，但可以连接测试网络节点获取资产合约数据。创建完 web3 实例后，便可通过 web3 来创建智能合约实例，工厂合约实例代码如下：

```
import web3 from './web3';// 引入 web3 实例
import DigitalAssetFactory from './build/DigitalAssetFactory.json'; //引入合约 ABI

// 调用 Contract 方法，传入工厂合约 ABI 以及已部署的合约地址，创建合约实例
const instance = new web3.eth.Contract(
    JSON.parse(DigitalAssetFactory.interface),
    '0x1f0d387b6598c0a3880ebcb7833dc1d07f8d845b'
);
export default instance; // 导出工厂合约实例，可供前端页面调用合约函数
```

通过向 web3 的 Contract 方法中传入工厂合约的应用二进制接口以及已部署的合约地址，可以获取对应该工厂合约的一个本地实例。通过该实例，可在前端页面方便调用该合约下的函数，完成资产数据的获取以及注册交易。

同样，我们还需要创建资产合约实例来对资产进行交易：

```
import web3 from './web3';
import DigitalAsset from './build/DigitalAsset';

// 定义创建合约实例的方法
const createContract = (address) => {
  return new web3.eth.Contract(
    JSON.parse(DigitalAsset.interface),
    address
  )
}
export default createContract;
```

在以上代码中，实际上并没有完成实例的创建，而是定义了一个函数，该函数需要传入合约地址，才能返回一个资产合约实例。因此，当我们通过工厂合约完成合约的创建并取得资产合约地址后，才能通过该方法创建指向那个资产合约的实例。不同的合约地址也就会创建出不同的实例。

4.4 前端页面

4.4.1 资产注册

用户可以通过资产注册界面进行资产注册，注册页面如下：

图 4-3 资产注册界面

用户需要首先上传资产文件，并在相应输入框内输入资产注册信息进行资产注册。文件通过前端界面传入后台服务器进行保存处理。

前端文件上传函数：

```
  uploadHandler = () => {
    const data = new FormData();// 创建 Form 对象
    data.append('file', this.state.selectedFile);// 将上传的文件写入 data
    // 使用 post 方法将文件传入后台
    axios.post("http://localhost:5000/upload", data);
  }
```

后端文件保存函数：

```
const multer = require('multer') // 引入 multer 模块
const path = require('path');
const fs = require('fs');
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, './upload');    // 设置保存的文件夹
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname); // 创建资产文件名
  }
});
var upload = multer({ storage: storage }).single('file') //创建 multer 实例
module.exports = (app) => {
  app.post('/upload', function (req, res) {// 接收资产文件
    upload(req, res) //
    return res.status(200).send("upload success!") // 返回前端上传成功信息

  });
```

后端服务器接收到前端发送的资产文件，将其保存在 upload 文件夹中。并返回上传成功信息。

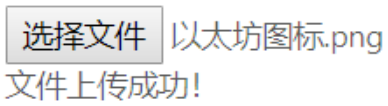

图 4-4 文件上传按钮

文件上传成功后点击确认注册，将执行资产注册函数：

```
import factory from '../ethereum/factory'; //引入工厂合约实例
```

```
await factory.methods
        .createDigitalAsset(
          this.state.AssetName,
          AssetID,
          this.state.AssetType,
          web3.utils.toWei(this.state.AssetPrice, "ether"),
          web3.utils.toWei(this.state.AccessPrice, "ether"),
          this.state.OwnerAddress,
          this.state.Share,
          this.state.AssetDescription
        )
        .send({
          from: accounts[0]
        })
```

通过引入工厂合约实例，并调用其 createDigitalAsset 函数，传入填写后经过处理的各项参数，即可发送到以太坊网络中完成资产合约的创建。创建成功后的资产将被展示在浏览界面当中。

4.4.2 资产浏览

资产注册成功后，可通过工厂合约函数，查看工厂合约内部署的资产合约地址，获取各项资产简介，界面显示如下：

图 4-5 资产浏览界面

当前通过工厂合约注册了两份资产，分别为以太坊图标，以及以太坊黄皮书。通过该界面可以查看所有以注册资产，并可通过了解详情进入该资产相关的详细界面。

4.4.3 资产详情

进入以太坊图标资产详情，将看到以下界面：

图 4-6 资产详解界面

该界面将通过实例化该资产合约，调用合约函数，获取资产相关数据信息。注册时间在以太坊上以 unix 时间戳形式保存，即 1970 年 1 月 1 日到现在为止所经过的秒数。因此需要在页面上使用以下方法

```
const creationTime = new Date(unixTimestamp * 1000);  //转换为通用日期
const creationTime = creationTime.toLocaleString();// 转换为当地日期
```

来将其转换为当地时间显示。

4.4.4 资产交易

进入资产股权购买页面，输入购买用途，所选卖家，以及欲购买的股份，即可进行资产股权交易。

图 4-7 资产股权购买界面

提交交易后，系统会自动计算所需支付的以太币，并发起交易，支付成功后进入交易记录界面可以看到该笔交易进入待定状态：

图 4-8 交易待定状态

待卖家点击确认交易后，交易将成功，

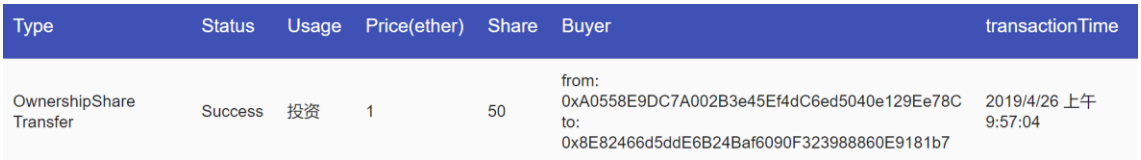

图 4-9 交易成功状态

并且资产详情将更新股东信息：

图 4-10 更新资产详情信息

出现新的股东信息，代表股权已顺利转让，该笔交易正式完成。

如需要购买该资产使用权限，可通过使用授权购买页面，并输入资产用途：

图 4-11 购买资产使用权

交易完成后，交易记录中将显示交易状态为成功，而无需他人确认操作:

Type	Status	Usage	Price(ether)	Share	Buyer	transactionTime
OwnershipShare Transfer	Success	投资	1	50	from: 0xA0558E9DC7A002B3e45Ef4dC6ed5040e129Ee78C to: 0x8E82466d5ddE6B24Baf6090F323988860E9181b7	2019/4/26 上午9:57:04
Access Authorization	Success	个人创作使用	0.1	0	from: 0x766746e2558e5B46207e2925954cda9f2fa48b3c to: 0x9193d5cc90F03C67dCa3476286AeF437088280d4	2019/4/26 上午10:06:39

图 4-12 交易成功

同时资产详情界面将更新资产使用授权情况：

图 4-13 授权使用者信息添加

使用授权获取的利润保留在合约账户余额中，因此股东可以通过在资产交易记录页面点击提取合约利润，按股权份额分股所有合约账户余额。

因为分配利润时，总是按照当前的股权份额情况分配。如果在存在利润余额的情况下进行了股权交易，则代表该股权所对应的利润也被一同出手了。在交易完成后分别查看两个股东的账户，发现账户余额分别增加了 0.05 个以太币，表示利润提取成功。

4.5 本章小节

本章详细介绍了智能合约的开发实现过程以及前端展示界面的实现结果。通过编写智能合约以及在通过前端进行部署，使得用户可以使用以太坊钱包在浏览器中与该系统进行交互操作。

第五章 系统测试

5.1 单元测试

单元测试用于对函数模块以及功能进行输入输出测试，通过编写测试案例，测试该函数在获得不同输入时获得的输出结果，并与预期结果相比较。通过对系统内主要函数功能编写单元测试，可以在系统出现故障时，快速定位出现问题的函数，从而可对函数自行修复。每一个单元测试要保证该测试的完备性，即该测试不应依赖其他外部条件，而只根据测试输入相应。如对中间函数进行测试，测试案例需提供模拟的上游输出结果，来检测该函数的输出正确性。

本项目使用 mocha 进行单元测试，通过在测试链上部署智能合约，来运行对应不同功能函数的单元测试。

5.1.1 测试合约部署

首先通过 beforeEach 函数，创建每个测试所需部署的合约：

```
beforeEach(async () => {
  accounts = await web3.eth.getAccounts();

  factory = await new
web3.eth.Contract(JSON.parse(compiledFactory.interface))
    .deploy({ data: compiledFactory.bytecode })
    .send({ from: accounts[0], gas: '5000000' });

  await factory.methods.createDigitalAsset(
    '江山美如画.jpg',
    '6324',
    '图片',
    3000000000000000000,
    100000000000000000,
    [accounts[0],accounts[1]],
    [40,60],
    '一张非常酷炫的风景照'
```

```
    ).send({
    from: accounts[0],
    gas: '3000000'
  });

  [DigitalAssetAddress] = await
factory.methods.getDeployedDigitalAssets().call();
  DigitalAsset = await new web3.eth.Contract(
    JSON.parse(compiledDigitalAsset.interface),
    DigitalAssetAddress
  );
});
```

通过填入资产注册测试用例，创建工厂合约以及测试资产合约。

5.1.2 智能合约部署测试

```
  it('资产注册', () => {
    assert.ok(factory.options.address);
    assert.ok(DigitalAsset.options.address);
  });
```

通过测试合约地址的存在，来保证合约已正确部署。

5.1.3 资产简介获取

```
  it('获取资产简介', async () => {
    const DigitalAssets = await
factory.methods.getDeployedDigitalAssets().call();
    const Assets = await Promise.all(
      DigitalAssets.map((asset, index) => {
        return (
          factory.methods.Assets(index).call()
        )
      })
    );
```

```
    assert.equal(Assets[0].name, '江山美如画.jpg');
  });
```

5.1.4 获取资产详情

```
  it('获取资产详情', async () => {
    const summary = await DigitalAsset.methods.getSummary().call();
    assert.equal(summary[1], '6324');
  });
```

5.1.5 发起股权交易

```
it('发起股权交易', async () => {
    const BuyPrice = (3000000000000000000/100)*30;

    await DigitalAsset.methods.ownershipTransaction(
      '买来好玩',
      accounts[0],
      30
    ).send({
      from: accounts[2],
      value: BuyPrice,
      gas: '3000000'
    })

    const transaction = await DigitalAsset.methods.transactions(0).call();
    assert.equal(transaction.status, 'Pending');
  });
```

5.1.6 确认股权交易

```
it('确认交易', async () => {
    const BuyPrice = (3000000000000000000/100)*30;
    await DigitalAsset.methods.ownershipTransaction(
      '买来好玩',
```

```
      accounts[0],
      30
    ).send({
      from: accounts[2],
      value: BuyPrice,
      gas: '3000000'
    })

    await DigitalAsset.methods.checkTransaction().send({
      from: accounts[0],
      gas: '3000000'
    })

    const transaction = await DigitalAsset.methods.transactions(0).call();
    assert.equal(transaction.status, 'Success');
  });
```

5.1.7 取消股权交易

```
it('取消交易', async () => {
    const BuyPrice = (3000000000000000000/100)*30;
    await DigitalAsset.methods.ownershipTransaction(
      '买来好玩',
      accounts[0],
      30
    ).send({
      from: accounts[2],
      value: BuyPrice,
      gas: '3000000'
    })

    await DigitalAsset.methods.cancelTransaction().send({
      from: accounts[0],
      gas: '3000000'
```

```
    })

    const transaction = await DigitalAsset.methods.transactions(0).call();
    assert.equal(transaction.status, 'Cancelled');
  });
```

5.1.8 访问授权交易

```
it('访问授权交易', async () => {
    const BuyPrice = 100000000000000000;
    await DigitalAsset.methods.accessTransaction(
      '用来好玩'
    ).send({
      from: accounts[3],
      value: BuyPrice,
      gas: '3000000'
    })

    const user = await DigitalAsset.methods.accessUser(0).call();
    assert(user, accounts[3]);
  });
```

5.1.9 交易数据获取

```
it('获取交易数据', async ()=>{
    const BuyPrice = 100000000000000000;
    await DigitalAsset.methods.accessTransaction(
      '用来好玩'
    ).send({
      from: accounts[3],
      value: BuyPrice,
      gas: '3000000'
    })

    const transaction = await DigitalAsset.methods.transactions(0).call();
```

```
    assert(transaction.status, 'Success');
  })
});
```

5.1.10 测试结果

合约测试

√ 资产注册
√ 获取资产简介 (65ms)
√ 获取资产详情
√ 发起股权交易 (136ms)
√ 确认交易 (270ms)
√ 取消交易 (208ms)
√ 使用授权交易 (131ms)
√ 获取交易数据 (104ms)

8 passing (3s)

5.2 测试总结

通过对智能合约该项功能函数编写单元测试，保证了项目在经过修改或是添加其他功能时，原功能的正常运作。而不需要在增加新功能后手动去进行过多调试，担心原有功能受到影响。良好的单元测试使得代码可重复利用性高，也避免了以后无意对代码原有结构的破坏。

第六章 总结与展望

该系统作为数字资产管理与交易的平台，使得个人可以通过文字创作或艺术创作直接获取价值收益，可以有效激发作者创作热情。同时通过数字资产买卖双方直接点对点的交易，避免了第三方中介过多的中介交易费用，有利于使得数字资产更好的流通，从而更大限度的发挥其具备的价值。该系统未来可通过与法律机构相联合，对注册的资产自动进行知识产权认证保护。并可对侵权行为在线追踪，汇集到法律机构，通过法律进行维权与索赔。

目前该系统交易基于以太坊账户完成，比特币以及以太坊都属于匿名账户，在法律上无法很好的定义该账户与个人的直接关联。因此未来考虑通过与政府部门配合，对区块链账户进行实名信息注册，从而直接与个人信息相关联，将更有助于数字资产的监督与管理。

致　谢

首先需要感谢导师对我毕业设计的细心指导和支持，以及对项目进展的不断督促。同样非常感谢有以太坊这么好的开发平台，尽管相关文档写的不甚详尽，但还是足以支撑我完成本项目的基础开发需要。同样以太坊开发社区相关的问题回答也帮助我解决了无数开发中遇到的 Bug。这一个毕业设计做到了很多第一次，第一次系统的学习区块链知识，第一次尝试完整的编写前端界面，第一次尝试智能合约编写。尽管回头望去，大部分时间都被一两个 Bug 卡住，然后不断调试。但最后还是要感谢自己的坚持，最终完成了本科最后的毕业设计，不留下遗憾。

参考文献

[1] 吕坤, 鲍可进. 基于区块链的数字资产交易系统设计与实现[J]. 软件 导刊, 2018(7).

[2] 王醒,翁健,张悦,李明.基于信誉值创建数字资产的区块链系统[J].信息网络安全,2018(05):59-65.

[3] 韩爽,蒲宝明,李顺喜,李相泽,张笑东,王帅.区块链技术在数字资产安 全交易中的应用[J].计算机系统应用,2018,27(03):205-209.

[4] 曹冬青, 李奕滨. 浅谈区块链:由比特币到数字资产的财富新大陆[J]. 中国经贸导刊(理论版), 2018(11).

[5] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[J]. 2008.

[6] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum project yellow paper, 2014, 151: 1-32.

[7] Buterin V. A next-generation smart contract and decentralized application platform[J]. white paper, 2014.

[8] Yuan Y, Wang F Y. Towards blockchain-based intelligent transportation systems[C]. Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on. IEEE, 2016: 2663-2668.

[9] Tapscott D, Tapscott A. Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world[M]. Penguin, 2016.

[10] Zhu Y, Qin Y, Zhou Z, et al. Digital Asset Management with Distributed Permission over Blockchain and Attribute-Based Access Control[C]. 2018 IEEE International Conference on Services Computing (SCC). IEEE, 2018: 193-200.

[11] Li H, Zhu L, Shen M, et al. Blockchain-based data preservation system for medical data[J]. Journal of medical systems, 2018, 42(8): 141.

[12] Dannen C. Introducing Ethereum and Solidity[M]. Berkeley: Apress, 2017.

[13] Gencer A E, Basu S, Eyal I, et al. Decentralization in bitcoin and ethereum networks[J]. arXiv preprint arXiv:1801.03998, 2018.