青岛理工大学

毕 业 设 计(论 文)

题目 基于区块链技术的票据操作系统的设计与实现

学生姓名：赵 帅

指导教师：周 炜

信息与控制工程 学院 软件工程 专业 软件 173 班

2021 年 5 月 22 日

摘　要

在现实生活中，票据的存在大大丰富了金融市场。票据可以支持实体经济的发展，票据也可推动货币的流动，票据是支持社会金融发展的有力工具。当今由于互联网技术的迅速发展，票据存在的主要形式为电子票据。央行建立的电子商业汇票系统使票据的操作电子化。但是，此系统是标准的以银行为中心的票据系统，此类票据系统在使用上有诸多问题，比如无法验证票据真伪，容易产生违规交易，票据处理效率低下和票据信息安全性差等。因此，基于去中心化的浪潮，将区块链技术应用到票据系统中，是一种较好的应用方案。

通过对票据知识的了解和对各种票据行为的具体分析，本项目决定以票据背书、票据承兑和票据贴现操作为示例，对票据操作系统进行开发。本系统采用联盟链技术，依靠于 Linux 基金会的开源项目企业级区块链开发平台 Hyperledger Fabric。此平台为各个交易实体间提供去中心化应用的实现。在本票据操作系统中，区块链网络中设置四个组织，分别为一个银行组织和三个公司组织。在公司与公司之间和公司与银行的交易的过程中，体现票据承兑、背书、贴现三个操作功能的实现和区块链技术在此票据系统中的应用。本系统将票据信息、交易的记录等信息都存储在区块链网络中。银行可以查询所有票据的信息以及历史记录，以实现监管功能。公司通过银行实现开票的功能，公司可以查询与自己相关的票据信息，并且对票据实现承兑、背书、贴现等基本操作。银行也可以对自己的待贴现票据进行操作。

在本票据操作系统中，采用前后端分离的系统设计。前端使用 Vue 框架进行开发。后端使用 Gin 框架进行系统搭建，使用 Go 语言实现后端 SDK 的创建以及业务功能接口的编写。同时，使用 Go 语言编写区块链中的智能合约，通过后端中的 SDK 去调用智能合约中的方法，以实现对区块链中账本数据的操作。在底层区块链中，使用 CouchDB 数据库来存储数据。编写 yaml 类型的配置文件和 shell 文件实现底层区块链网络的搭建，智能合约的打包和安装。

关键词：区块链，Hyperledger Fabric，承兑，背书，贴现，Gin，Vue，Go

ABSTRACT

In reality, the existence of bills greatly enriches the financial market. Bills can support the development of the real economy, they can also facilitate the flow of money, and they are a powerful tool to support the financial development of society. Today, due to the rapid development of internet technology, the main form of bill existence is electronic bills. The electronic commercial bill of exchange system established by the central bank has enabled the electronic operation of bills. However, this system is a standard bank-centric bill system. There are many problems with the use of such bill systems, such as the inability to verify the authenticity of bills, the tendency to generate irregular transactions, inefficient bill processing and poor security of bill information. Therefore, based on the wave of decentralisation, the application of blockchain technology to the note system is a better application solution.

Through knowledge of bills and specific analysis of various bill behaviours, this project decided to develop a bill operating system using bill endorsement, bill acceptance and bill discounting operations as examples. The system uses federated chain technology and relies on Hyperledger Fabric, an open source project of the Linux Foundation for enterprise-grade blockchain development, which provides decentralised application implementations between various transaction entities. In this note operating system, four organisations are set up in the blockchain network, one banking organisation and three corporate organisations. The implementation of the three operational functions of acceptance, endorsement and discounting of bills and the application of blockchain technology in this bill system are reflected in the process of company-to-company and company-to-bank transactions. This system stores the information of the notes and the records of the transactions in the blockchain network. The bank can check the information of all the bills as well as the history records to achieve the regulatory function. The company can check the information of the bills related to itself and perform basic operations such as accepting, endorsing and discounting the bills through the bank. Banks can also perform operations on their own bills to be discounted.

In this note operating system, a separate front and back-end system design is used. The front end is developed using the Vue framework. The back-end uses the Gin framework to build the system, and the Go language is used to create the back-end SDK and write the business function interfaces. At the same time, the smart contract in the blockchain is written in Go language, and the methods in the smart contract are called through the SDK in the backend to achieve the operation of the ledger data in the blockchain. In the underlying blockchain, a CouchDB database is used to store the data. Write yaml type configuration files and shell files to build the underlying blockchain network and package and install the smart contracts.

Key Word: Blockchain, Hyperledger Fabric, Endorse, Acceptance, Discount, Gin, Vue, Go

目录

第 1 章 绪论

1.1 研究背景

随着票据系统的成熟以及互联网经济的发展，传统票据形式的弊端逐渐显现。在纸质票据阶段，票据存在真实性的问题，纸质票据的信息并不同步，从而极易被克隆或者伪造，并且不易对假票进行鉴别。纸质票据也很可能在用户的携带保存过程中丢失。并且，由于票据承兑人和收款人信息不对称，可能出现票据到期时，收款人未收到款项的状况。电子票据阶段，即便互联网技术使票据交易的处理更加便利，但仍然存在诸多问题。比如处理票据的效率不高，监管和审查成本高。并且，可能会出现由于系统缺陷，造成票据的安全问题，如一票多卖，出租账户以及票据信息泄漏等问题。这些问题都暴露了当前票据系统的不完善，阻碍了票据交易的进一步发展。

区块链是一种计算机新兴技术，自中本聪提出“一种点对点的电子现金系统”后，区块链发展至今。区块链本质是一种结构，即由一种存储结构“区块”组成的链状结构。“区块”通过保存前一个区块的地址，与前一个区块连接在一起。“区块”中包含记录交易信息的“账本”。每次对“账本”中的信息进行进行修改，就需要征得其他区块的同意，由此机制形成了“共同监督”的模式，这样也形成了“去中心化”的特点。这样，区块链的信任机制便建立起来。在征得同意的过程中，包括不同的规则，按照对应规则完成其认可要求，即可对“账本”信息进行操作。因此，在这些规则下，区块链的安全性得以保证。即便有一些不法分子想通过强大的算力去修改账本，但此类修改操作需要征得超过 51%以上“区块”的同意，通过这种方式带来的收益反而远远超过了操作成本，从而形成攻击悖论。因此，区块链技术可以保证数据的安全，可以保证数据无法被篡改，并且实现了去中心化，建立了良好的信任机制。

在现实生活中，人们听闻最多的可能是“比特币”，是一种电子加密货币，这是在区块链 1.0 时代的主要产物。在区块链 2.0 时代，基于以太坊区块链，以太坊加入了智能合约和认证标准，成为了一个去中心化的应用开发平台，主要应用于金融领域。现在到了区块链 3.0 时代，基于 Linux 基金会的开源项目企业级区块链开发平台 Hyperledger Fabric，区块链技术已经可以在众多领域展开应用。而

不仅仅局限于加密货币或金融领域。使用 Hyperledger Fabric 平台，企业可以开发信息管理平台，从而确保信息安全。也可以开发认证系统。溯源系统等，在现实生活中达到“去中心化”的目标，降低系统成本，构建良好的信任机制。

在本项目中，对于票据信息的真实性问题，通过用户之间的共享账本，票据系统用户可以在不依赖第三方平台的基础上，验证票据的真实性。其次，对于违规交易的问题，区块链将信息存储于分布式账本中，可以确保票据信息的同步，避免违规交易。利用区块链技术还可以解决系统操作效率低下的问题。在票据操作完成后，分布式账本中进行信息同步，自动完成对账。并且，由于是对本地的账本信息进行检索，可以提高检索速度，提升票据处理的效率。对于系统的安全性问题，区块链技术可以保证票据系统的数据安全和网络安全。 区块链技术具有不可篡改的特点，票据数据无法篡改。区块链中数字签名和安全传输的过程，解决了交易请求被攻击或伪造的问题。确保了票据的数据和网络安全。

1.2 国内外研究现状

目前，由于区块链技术的迅速发展，特别是由于 Linux 基金会的开源项目企业级区块链开发平台 Hyperledger Fabric 出现，很多企业都开始开发区块链技术在票据系统中的应用。

国内研究上，2016 年 12 月浙商银行成功构建基于区块链技术的移动数字汇票平套，并在 2017 年 1 月 3 日，该产品正式上线应用并完成首笔交易。这标志着区块链技术在银行中的主要业务的正式应用。在该票据中，用户可以在移动端实现对票据的基本操作，比如票据的签发、签收、转让、买卖等功能。2018 年 1 月 25 日，上海票据交易所成功上线并运行了数字票据交易平台。并且，工商银行工商银行、中国银行、浦发银行和杭州银行在数字票据交易平台顺利完成了基于区块链技术的数字票据签发、承兑、贴现和转贴现业务。2018 年 6 月 7 日，中国人民银行也搭建完成了基于区块链技术的票据系统，可实现将国内企业的票据数字化。

国外研究上，有众多基于区块链技术的金融交易方面的应用。比如基于区块链系统的交易记账系统，基于区块链技术的供应链财务系统。这些研究与应用都是将区块链技术应用于金融交易方向与隐私保护方向的应用，与本票据操作系统

有异曲同工之处，都可以体现区块链技术的去中心化，安全的特点，以及其应用在票据领域的广阔前景。

虽然，近年来很多银行甚至公司都开始开发自己的票据系统，但是这些系统都是针对个人而开发，都是根据特定需求而开发，缺少通用性。并且，由于其开发的保密性，相关的开发资料也很稀有，因此，针对区块链在票据领域的开发还没有公共的标准，相关的研究以及在项目开发的工作和交流方面存在着很大的困难。

1.3 本文研究的主要内容

本文研究的主要内容是根据区块链的去中心化，安全以及不可篡改等特征去解决当前票据系统数据缺乏安全性，交易中心化程度高和信息维护成本高等问题。根据这些问题，本文的研究内容主要包括下面几个方面：

（1）分析基于区块链技术的票据操作系统的可行性

当前，基于区块链技术的票据操作系统的应用十分稀少，只有比较大型的银行进行了此类开发，在网络上的开源项目和资料十分缺乏。基于这种情况，很有必要对此类系统的需求与可行性进行细致分析，以确定系统的确切需求，以及开发是否真实可行。

（2）搭建区块链底层网络

由于初次接触区块链技术，首先需要去学习区块链的一些基础知识，比如 peer 节点、orderer 节点、通道、组织、智能合约等。在了解一些基础知识后，需要去了解区块链底层搭建的流程，了解搭建的原理，搭建的流程以及搭建的方法。然后就需要进行实际的操作。在阿里云租用 Linux 服务器，下载 Hyperledger Fabric 项目到云服务器，编译 Fabric 源码，设计网络架构，编写配置文件并编写 shell 脚本以运行配置文件从而搭建区块链底层网络。

（3）票据基础知识和票据操作行为

票据操作涉及一些票据的专业知识，因此需要在开发前学习一些票据知识。首先，需要搞清票据出票人、票据承兑人、票据收款人和票据持有人之间的关系。其次，需要细致了解票据背书行为的意义。在此票据系统中主要涉及三个票据操作，票据承兑、票据背书和票据贴现。

票据承兑。即承诺并兑现票据，一般由票据的付款方执行此操作。在银行发布票据后，票据的承兑人需要进行同意承兑或拒绝承兑的操作。如果同意承兑，则需要承担票据付款方的债务关系，如果拒绝承兑，则票据作废。

票据背书，即票据的当前收款方将票据收款权利转移到其他公司用户操作。此操作只可由票据的收款方进行。票据的收款人可以向除本人和票据承兑人外的用户发出背书请求，如果对方同意，则转移票据的债务关系到此公司用户，如果拒绝，则票据不发生改变。

票据贴现，即票据当前收款方将票据收款权利转移到银行用户的操作。根据业务需要，票据收款方需要快速结算票据，此时，可以对银行给予一定的利息补偿，从而将票据的收款方关系转移到银行用户。银行可以选择同意或拒绝贴现。如果对方同意，则转移票据的债务关系到此银行用户，如果拒绝，则票据不发生改变。

（4）设计票据操作系统

在学习票据相关知识后，可以着手设计系统业务。系统中设置一个银行账户用于模拟现实票据业务，设置三个公司账户，用于实现模拟票据的承兑、背书和贴现操作以及其他基础票据操作。

（5）实现基于区块链技术的票据操作系统

结合分析的票据操作系统的业务逻辑，去搭建区块链底层网络。根据业务需求，计划项目功能，编写智能合约和后端功能模块。编写前端，调用后端方法，传递数据，现实数据。设计交互良好的用户界面，方便用户使用。

1.4 论文结构

本节介绍本项目的具体实现，主要包括下面七个章节的内容：

（1）“绪论”：是介绍本项目开发的产生背景、展示目前国内外研究的现状和主要的研究内容。

（2）“项目关键技术的分析”：对该项目涉及到的关键技术进行介绍，包括区块链技术、Go 语言、Gin 框架、Vue 框架。

（3）“需求分析”：对基于区块链的票据操作系统的需求分析以及可行性分析进行介绍。

(4) “系统总体设计”：介绍系统的总体架构。包括系统总体设计说明、系统总体架构设计、系统主要功能模块、系统流程图以及系统的数据存储结构。

(5) “系统详细设计”：按照从前端到后端再到区块链网络的顺序介绍登陆模块、银行模块和企业模块，详细的展示了数据结构设计，展示了系统时序图。

(6) “系统功能实现”：介绍本系统的环境搭建、以及主要功能模块的具体实现流程。

(7) “结论与展望”：对本次系统的开发过程与结果做出总结，指出此系统的不足之处和对未来的展望。

第 2 章 项目关键技术的分析

2.1 区块链技术

区块链技术是一种计算机领域的新兴技术。区块链采用一种链式的区块结构储存信息，这些区块使用密码学互相关联，每个区块包含前一个块的加密散列、时间戳和交换信息。使用区块链，我们可以通过“共识机制”，安全地使用区块链存储信息，每个人都可以看到区块链内的信息，但不能随意进行修改。区块链将跟踪所有交易，并利用分布式系统进行验证每个交易。从而实现一种去中心化的，十分安全的信息存储系统。

2.1.1 对等网络

对等网络也称为 P2P 网络，是区块链网络实现的重要基础。P2P 网络不同于传统的客户端/服务器结构，在 P2P 网络中，每个节点既可以是客户端也可以是服务器，每个节点的身份是平等的。每个节点都为区块链网络共享自己的一部分资源，用于共同组成区块链底层网络。所有节点都会参与校验和广播交易及区块信息，并且维持与其他节点的连接。在如此结构下，区块链网络中的每个节点都十分重要，其共同组成区块链的完整结构，使区块链网络具有极高的安全性和抗攻击性。

对等网络中有多种拓扑结构模型，每个拓扑结构都有其优缺点。基于不同拓扑结构模型，区块链可以改变其去中心化程度，从而针对不同应用，达到不同的功能要求。

区块链网络中的对等网络与 P2P 网络也并不完全相同。相比普通的 P2P 网络，区块链网络的结构更为复杂，区块链网络包括私有链、公有链和联盟链，因而需要更加复杂的网络结构。

2.1.2 分布式账本

在区块链网络中，记录信息的账本是分布式存储的。区块链中每一个有记账功能的节点，都会在自己的节点上存储一份完整的账本。每当发生交易，其中一个节点中的账本进行交易后，其他节点就会在一定时间内快速同步账本，以保持所有分布式账本的一致性。分布式账本在计算机内是以区块链的形式存在的。在

每一个区块上，区块将信息数据转化为 hash 值，上一个区块存储下一个区块的 hash 值，由此找到下一个区块的位置，由此连成区块链。这样，每当一个区块上的账本信息发生变化，其 hash 值改变，便会影响其他的区块，影响整个区块链。正是由这种分布式账本的形式，确保了区块链技术的去中心化，也确保了安全性。

2.1.3 共识机制

在一个区块链网络中，其数据并非存储在一个固定位置，而是存储在每个区块链的节点中，每个节点中都保存有一个账本，由此形成分布式账本结构。每当执行一笔交易，或者修改一次信息，账本内的数据发生变化，区块链中所有的节点就会在一定时间内相互同步彼此的账本信息，以保证每个节点内账本的一致性。在同步账本信息的过程中，需要采用一定的算法去执行同步过程，这便是“共识算法”。由“共识算法”来达成“共识机制”。

根据不同的应用场景，可以采用不同的共识算法。目前的“共识算法”主要有四类，分别为工作量证明机制（POW）、权益证明机制（POS）、.委托权益证明机制（DPOS）、验证池共识机制（POOL）。比特币在区块的生成过程中使用的就是 POW 共识机制，以太坊中也使用的是 POW 共识机制。

2.1.4 智能合约

智能合约的概念是美国著名计算机科学家尼克·萨博在 1994 年提出的。其定义的智能合约是一种能够自动执行的协议，即在无人监管的情况下，运行匿名用户按照合约内容进行交易。如今，随着区块链技术的发展，特别是区块链 2.0 时代之后，智能合约被应用到区块链技术中。

在区块链中，智能合约的作用就是约束区块链网络中交易的进行。开发者将交易规则写入智能合约中，再将智能合约安装到区块链网络的容器当中。这样，在安全的范畴下，每当用户发起交易，区块链网络就会按照用户的请求，遵循智能合约中的交易规则，去完成用户发起的交易，此过程是区块链自动进行的，无需他人的监督。由此，这样的交易规则确保了区块链网络中交易的规范性与安全性，防止了他人恶意输入，从而造成系统错误。并且，在区块链网络中遵循智能合约执行交易时，所有的交易都会被自动记录到区块链的账本中，包括记录一些修改内容，从而确保了数据的可溯源性，确保了数据的存储安全。

2.2 Hyperledger Fabric

2.2.1 通道

通道是 Fabric 中极其重要的概念，通过通道，开发者可以将共享一个账本的几个节点放到一个通道中，同时，也可以将业务上不允许共享一个账本的节点进行分隔，由此确保节点对数据访问的安全性。通道相当于一个群聊，节点相当于参加群聊的用户，当用户参加此群聊后，即可访问当前群聊内的信息，节点加入通道后，也可共享账本的内容。如果不想某个用户接收到群聊的信息，也就要求此用户不加入此群聊，不将此节点加入通道即可。当然，一个节点，也可加入多个通道，从而可以访问多个账本的数据。

2.2.2 Fabric 中的智能合约

在区块链中，智能合约的作用就是约束区块链网络中交易的进行。开发者将交易规则写入智能合约中，再将智能合约安装到区块链网络的容器当中。这样，在安全的范畴下，每当用户发起交易，区块链网络就会按照用户的请求，遵循智能合约中的交易规则，去完成用户发起的交易。在 Fabric 中，对智能合约进行了改造。目前，可以使用 Go、Java、Node.js 和 Javascript 等语言进行开发，从而方便了开发者的开发过程。

2.2.3 Fabric 的权限系统

Fabric 中，其主要应用方向为企业级的区块链应用平台，所以大多数情况下，区块链应用组成的是一个联盟链。当联盟中的组织想要对区块链的账本进行交易，就需要获得授权。Fabric 中有 MSP 系统，它在组织加入区块链网络时，会为其颁发 Fabric-ca 证书文件，这样，组织才有了发起交易的权限。

2.2.4 Fabric 的共识机制

“共识机制”是帮助区块链网络中的节点同步账本信息的一种机制。Fabric 中将 “共识机制”设置为可插拔的模块。Fabric 中的”共识机制“并非固定的，在开发过程中，开发者可以根据不同的系统吞吐量需求，确定应用不同的共识机制。目前，Fabric 中提供了三种共识机制，分别是 Solo 共识机制、Kafka 共识机制和 raft 共识机制。Solo 共识机制效率较低，Kafka 共识机制可以有效提高共识达成效率，从而提高系统吞吐量。

2.2.5 Fabric 的节点

Fabric 中拥有多种节点，主要分为 Orderer 节点和 Peer 节点两种。其中，Peer 节点按照功能可分为 Leader Peer(主节点），Anchor Peer（锚节点），Endorse Peer（背书节点）和 Committer Peer（确认节点）。一个 Peer 可以担任不同的角色。Orderer 节点负责对交易进行排序执行的操作。

2.2.6 Fabric 的交易流程

区块链的重要特征之一就是能够保证交易的安全进行。但 Fabric 的交易流程与公有链有很大的不同。

下面介绍 Fabric 的整个交易流程：

（1）客户端首先需要获取权限，需要去 CA 中注册并认证，从而得到一些加密文件，比如私钥，从而获得发起交易的权限。

（2）客户端获取权限后，需要使用相应的 SDK 提供的开发包构建交易提案。提案中包含通道信息、要调用的链码信息、时间戳、客户端的签名、调用链码中的函数和参数。通过提交这些信息，完成客户端对交易的发起。

（3）客户端发起交易提案后，背书节点需要对收到的交易提案进行验证。主要验证交易提案格式是否正确、交易在之前并未提交过、提交交易提案的客户端签名是否有效和提交交易提案的请求者是否在该通道中有相应的执行权限。验证通过后，背书节点会调用链码，进行模拟交易，产生交易结果，将结果背书并发给客户端。这时的交易是模拟进行的，并不对账本中的数据产生影响。

（4）客户端收到模拟数据集。如果是执行查询操作的数据集，则会对查询结果进行检查，如果收到执行更新账本操作的数据集，则会构建交易请求，将其发送给 Orderer 节点，以进行进一步的交易。

（5）Orderer 收到交易请求后，并不对交易请求内的数据集进行检查。而是将交易进行排序，创建交易区块，将其广播给一个通道内所有 Leader Peer 节点。

（6）当 Leader Peer 节点收到交易后，回对交易进行验证。Leader Peer 节点会检查交易消息的结构是否正确、是否重复、是否背书和读写集版本。在验证通过后，将交易写入本地分类账本中。

（7）在 Leader Peer 节点执行完成交易后，会同步广播，将账本信息与同一通道内的其他 Peer 节点内的账本信息进行同步，完整最终交易。

2.3 Go 语言

Go 语言是一种静态类型，可编译，可并行化，并具有垃圾回收的编程语言，通常被称为 golang。它被认为是面向对象的程序设计语言，使用对象组合代替类继承。Go 语言由 Google 公司研发，并与 2009 年发布为开源项目。它的主要目标是兼具 Python 等动态语言的灵活性和 C 语言等编译语言的性能与安全性。Go 语言的语法有 Python 和 C++的影子，Go 语言具有严格的语法要求，比如不允许有未使用的变量存在。就是因为这样严格的语法要求，Go 语言在几秒钟内便可编译大型程序。Go 语言在 Web 开发领域有卓越的表现，也是现在 Web 开发的热门语言。

2.4 Gin 框架

Gin 框架是 Go 语言的为框架，其封装优雅，官方提供的 API 功能全面、友好。并且其快速灵活，十分适合进行 Web 项目的快速开发。并且 Gin 框架中使用自身的 net/http 开发包，可以帮助开发者使用 Go 语言进行简单并高效的 Web 开发，开发性能也十分优秀。基于 Gin 框架，不仅可以减少开发困难，也可以形成一定的开发规范，形成标准的代码格式。

2.5 CouchDB

CouchDB 是由 Apache 软件基金会开发的一个开源数据库。CouchDB 使用 JSON 类型数据进行存储。并且其中有一个基于 HTTP 的 RESTful 类型的 API，开发者可以使用 http 协议的 API 访问文档，并且使用 Web 浏览器查询索引。CouchDB 中的存储结构是基于文档的，因此，使用 CouchDB 数据库时，无需担心数据结构，其为用户提供强大的数据映射，可以进行多种数据操作。并且，CouchDB 数据库中的数据易于复制，方便共享并同步数据。

2.6 Docker 技术

Docker 是一个客户端-服务器(C/S)架构程序。Docker 客户端只需要向

Docker 服务器 或者守护进程发出请求，服务器或者守护进程将完成所有工作并返回结果。Docker 提供了 一个命令行工具 Docker 以及一整套 RESTful API。可以在同一台宿主机上运行 Docker 守护进程和客户端，也可以从本地的 Docker 客户端连接到运行在另一台宿主机上的远程 Docker 守护进程。

2.7 RPC 协议

在早期的计算机系统中，每个程序都是彼此分隔的，因此想要实现相同的功能，必须分别在两个程序中编写相同的程序，这样的开发效率低下，并且浪费很多的计算机资源。这时，开发人员试图开发一种协议，以帮助开发人员在程序开发中可以调用另一个程序中的功能，从而达到功能复用的目的。这样的技术，一直至今，就演变成 RPC 协议。程序猿可以使用 RPC 协议，去调用远程进程上的一套工具，大大减少了开发难度与工作量。本系统中采用的 gRPC 工具是就是基于 Go 语言的 PRC 协议包，

2.8 Vue 框架

Vue 框架是由国人开发的用于搭建用户界面的渐进式框架。Vue 被设计为 MVVM 的模式，可以自底向上逐层应用。Vue 的核心库只关心视图，开发简单，并且开发者可以使用第三方视图库，从而用户可以根据自身需要，选择不同的工具库，开发自己风格的界面。

2.9 本章小结

本章节主要是对本论文中所要在系统中应用的技术进行简单介绍。主要介绍了区块链技术的概念，流程和核心技术。同时介绍了 Hyperledger Fabric 平台相对于区块链技术中应用所做出的改进。介绍了编写智能合约和后端程序所要用到的 Go 语言，Gin 框架。还介绍了数据库 CouchDB，Docker 以及 RPC 协议。

第 3 章 基于区块链技术的票据操作系统的需求分析

3.1 票据系统流程分析

当今的电子票据系统在功能上已经十分成熟。但是，几乎所有的票据系统都是完全中心化的，大部分以银行为中心，信息也存储在银行的数据库中。这样，不仅存在一定的数据安全问题，并且还存在信息维护成本高，信息处理效率低和因信息不同步而出现的一系列问题。因此，在本系统中，保留当今银行与用户的交易模式，但应用区块链技术，在信息的处理上做出改变。以承兑、背书、贴现这三大票据基本功能为例，模拟现实中的电子票据系统，从而实践将区块链技术应用于票据系统的可行性。

本票据系统中，包括登陆模块、公司操作模块和银行操作模块三大功能模块，主要通过用户与用户之间和用户与银行之间的票据交易流程，体现区块链在票据系统当中的应用。下面将分别介绍三个功能模块的具体流程。

3.1.1 登陆模块

对于登陆模块，主要是模拟业务层上，传统信息管理系统的登陆功能。用户或者银行根据开发者事先内置的账号和密码到系统进行存储，输入正确，进入正是操作界面。登陆模块的流程如图 3.1 所示。

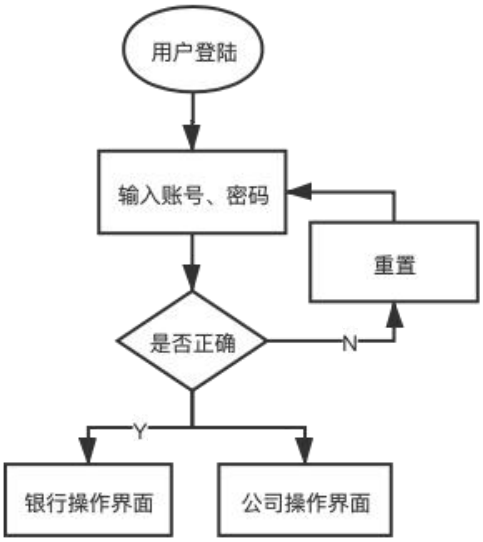

图 3.1 登陆模块流程分析

3.1.2 银行模块

对于银行，在票据系统当中承担者对票据的监督作用，并且也包含一些对票决的操作功能。银行可以查询所有票据的基本信息，查询票据的变更历史记录。由此满足对票据操作的监督要求，并且可以为用户提供交易记录凭证。除此之外，像传统票据系统一样，银行仍然负责开票的功能，开票之后，通知相关公司进行承兑操作。对于一些公司想要快速结算票据，可以通过贴现，向银行给予一定的补偿，从而讲收款人的关系转移到银行，从而达到快速结算的目的，因此，银行也负责处理公司发出的贴现请求，同意贴现请求，则将票据关系转移到银行本身，拒绝贴现请求，则贴现失败。银行处理待贴现票据功能流程如图 3.2 所示，银行开票功能流程如图 3.3 所示。

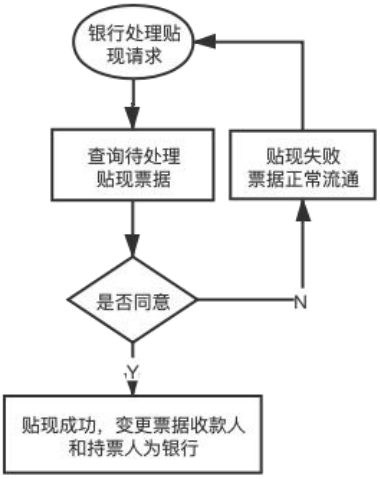

图 3.2 银行处理待贴现票据流程分析

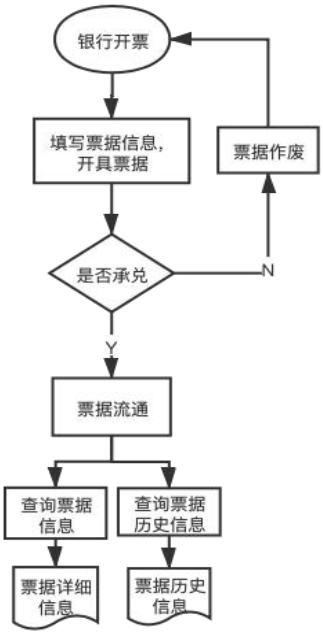

图 3.3 银行开票流程分析

3.1.3 公司模块

对于公司，在票据系统中拥有对与自己相关票据操作的功能。公司可以查看自己作为票据承兑人、票据收款人和票据持票人身份的票据信息。同时，对于自己作为票据收款人和票据持票人身份的票据，可以申请进行背书和贴现操作。进行背书操作，需要填写被背书人的 ID 和名称，填写完成后即发出背书请求，等待被背书人处理请求。进行贴现操作，则以系统中的银行为默认贴现对象，向银行发出贴现请求，等待处理。公司可以处理开票时，银行发出的承兑请求。如果同意承兑，则票据正式进入票据系统中正常流通。如果拒绝承兑，则票据被废除。公司也可以处理自己作为被背书人的从其他公司发来的票据请求，选择接收背书或拒绝背书。公司用户处理待承兑票据功能流程如图 3.4 所示，公司用户处理待背书票据功能流程如图 3.5 所示，公司用户请求票据背书功能流程如图 3.6 所示，公司用户请求票据贴现功能流程如图 3.7 所示.

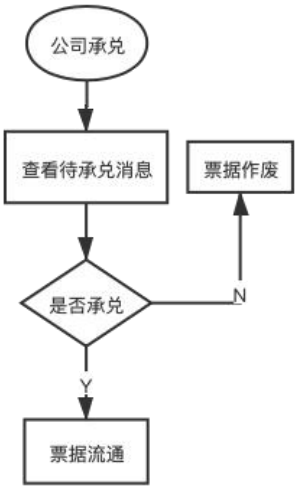

图 3.4 公司处理待承兑票据流程分析

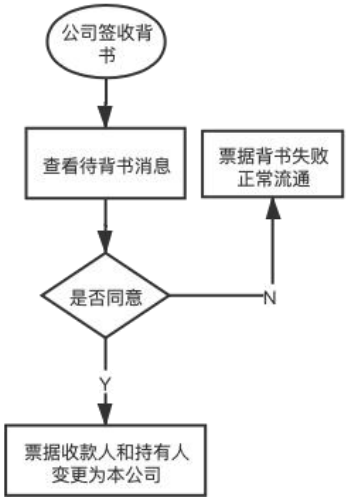

图 3.5 公司处理待背书票据流程分析

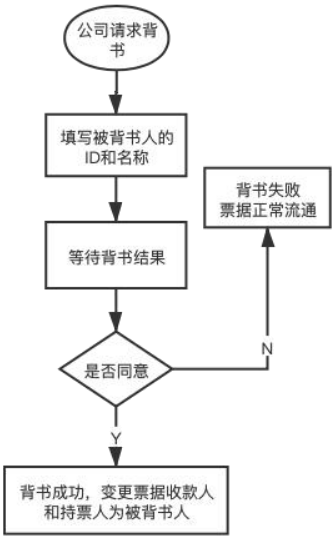

图 3.6 公司请求票据背书流程分析

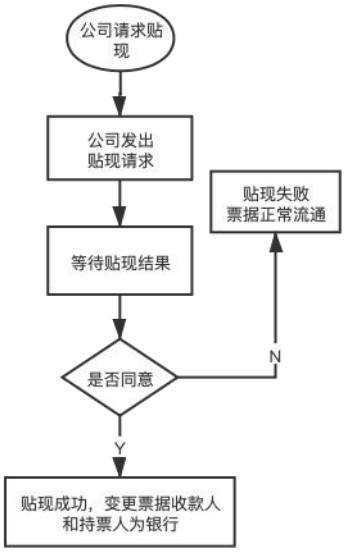

图 3.7 公司请求票据贴现流程分析

3.2 系统功能需求分析

根据上述流程分析，我们可将系统功能需求分为三个方面，分别是登陆功能需求分析、银行功能需求分析和公司功能需求分析，票据系统的各个功能由上述三个功能的交互而展开，下面将以用例图的形式来对这三个方面进行需求分析。

3.2.1 登陆功能需求分析

在票据操作系统中，需要依靠系统内置的账号密码保证系统的最基本安全。用户通过在系统中输入账号密码，系统需要进行对应的账号密码验证，并且根据账号来判断用户身份，帮助用户进入相应的页面。

3.2.2 银行功能需求分析

银行用户在票据系统中具有查看所有票据、查询票据历史信息、发行票据和处理待贴现票据等功能。银行用户用例图如图 3.8 所示：

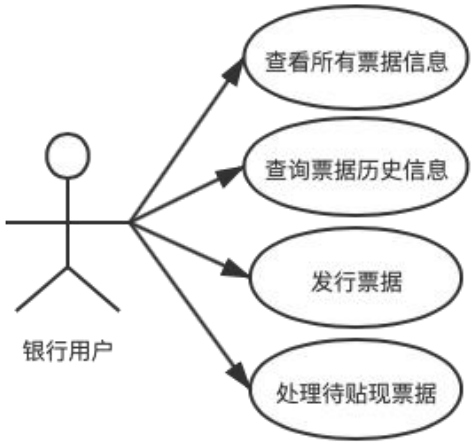

图 3.8 银行用户用例图

（1）查询所有票据信息

用例描述： 在整个票据系统中，尽管不可以对票据信息进行任意的删除或修改，但仍然需要一个角色进行监管，银行可以查询所有的票据信息，从而起到一定的监管作用。
事件流描述： 1、银行用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“查看所有票据”的按钮，即可显示所有票据信息。
业务规则： 填写正确登陆信息，以正常进入银行操作界面。

（2）查询票据历史信息

用例描述： 银行除了查看所有票据的当前信息外，还可以查询所有的票据的历史变更信息，也可起到一定的监管作用。
事件流描述： 1、银行用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“查询票据历史”的按钮，显示查询界面。 3、在票据编号的输入框中输入想要查询的票据编号，再点击“确定”按钮，即可显示出

该票据的历史操作记录。 4、如果该票据不存在，则查询不出任何信息。
业务规则： 需要填写正确的票据单号，否则无法查询到历史信息。

（3）发行票据

用例描述： 为模拟现实中的票据系统的业务流程，银行仍然具有发行票据的职责。银行通过填写票据信息、发行票据，票据会自动向承兑人发起承兑请求，此时票据状态为 made 状态，待票据被承兑后，即可进入正常发行状态。否则票据自动作废。
事件流描述： 1、银行用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“发行票据”的按钮，显示发行票据界面。 3、在此界面需要输入票据编号、票据金额、票据类型、票据发布时间、票据到期时间、票据承兑人 ID 和名称、票据收款人 ID 和名称、票据持票人 ID 和名称，输入完成点击下方的“发行票据”按钮，发行票据。 4、点击“发行票据”按钮后，后台会自动对输入的票据编号进行验证，如果已存在相同票据编号的票据，则发行失败，需要重新输入票据信息。
业务规则： 需要填写新的票据编号，并且填写正确的用户 ID 和名称，以免造成票据无效。

（4）处理待贴现票据

用例描述： “贴现”是票据的收款人或持票人为了快速结算票据，通过补偿，将票据的收款人或持票人关系转移给银行的操作。此处银行用户处理的待贴现票据，是其他公司用户发起的贴现请求。银行用户可以选择拒绝或接受，拒绝后，票据会显示”贴现失败信息“，票据状态回归正常发行状态。接受后，票据的收款人 ID 及名称和持票人 ID 及名称都会自动变为银行用户的 ID 和名称，票据回归正常发行状态。由此完成贴现操作。
事件流描述： 1、银行用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“处理待贴现票据”的按钮，显示处理待贴现票据界面。该界面中，显示收

到的所有贴现请求，可显示贴现请求提出人的名称，并且可以显示票据的详细信息。最后是对贴现请求的同意或拒绝按钮。 3、如果点击同意按钮，票据的收款人 ID 及名称和持票人 ID 及名称都会自动变为银行用户的 ID 和名称，票据回归正常发行状态。
业务规则： 在银行用户操作时，需要根据对应票据进行操作，需要操作准确。

3.2.3 公司功能需求分析

公司用户在票据系统中具有查看作为承兑人关系票据、作为收款人关系票据、作为持票人关系票据、发出背书请求、发出贴现请求、处理待承兑票据和处理待背书票据等功能。公司用户用例图如图 3.9 所示：

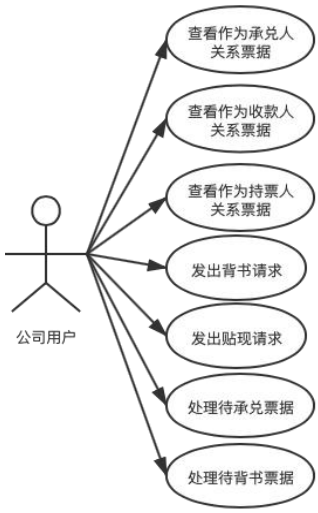

图 3.9 公司用户用例图

（1）查看作为承兑人关系票据

用例描述： 公司用户需要查询与自己公司相关的票据信息。由于票据关系的复杂性，本票据系统将公司的票据查看功能进行分类。此功能中，公司用户可以查看自己作为票据的承兑人而相关的所有票据信息。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“承兑人关系票据”的按钮，将显示所有自己作为票据的承兑人而相关的所有票据，点击“详细信息”按钮，即可查看票据的详细信息。

业务规则： 填写正确的账号和密码，以正常进入公司用户操作界面。

(2) 查看作为收款人关系票据

用例描述： 公司用户需要查询与自己公司相关的票据信息。由于票据关系的复杂性，本票据系统将公司的票据查看功能进行分类。此功能中，公司用户可以查看自己作为票据的收款人而相关的所有票据信息。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入银行操作界面。 2、点击左侧“收款人关系票据”的按钮，将显示所有自己作为票据的收款人而相关的所有票据，点击“详细信息”按钮，即可查看票据的详细信息。
业务规则： 填写正确的账号和密码，以正常进入公司用户操作界面。

(3) 查看作为持票人关系票据

用例描述： 公司用户需要查询与自己公司相关的票据信息。由于票据关系的复杂性，本票据系统将公司的票据查看功能进行分类。此功能中，公司用户可以查看自己作为票据的持票人而相关的所有票据信息。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入公司操作界面。 2、点击左侧“持票人关系票据”的按钮，将显示所有自己作为票据的持票人而相关的所有票据，点击“详细信息”按钮，即可查看票据的详细信息。
业务规则： 填写正确的账号和密码，以正常进入公司用户操作界面。

(4) 发出背书请求

用例描述： 在票据系统中，“背书”操作，即票据的收款人或持票人向其他用户转移票据关系的操作。

从而完成一些交易上需要的变化。背书操作使票据的使用更加灵活。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入公司操作界面。 2、点击左侧“收款人关系票据”的按钮，将显示所有自己作为票据的收款人而相关的所有票据，点击“详细信息”按钮，即可查看票据的详细信息。 3、在“详细信息”按钮右侧，有发起背书功能按钮。 4、点击“背书”按钮，会弹出填写框，公司用户需要填写被背书人的 ID 和名称。输入完后进行提交。 5、提交后，后台首先回对填写的被背书人信息进行验证。如果票据的被背书人为自己或者票据的承兑人，则无法提交。
业务规则： 需要填写正确的被背书人 ID 和名称，否则无法正常提交背书请求。

（5）发出贴现请求

用例描述： “贴现”是票据的收款人或持票人为了快速结算票据，通过补偿，将票据的收款人或持票人关系转移给银行的操作。在此，公司用户可以根据自身公司需求，发起贴现操作，以转移票据关系。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入公司操作界面。 2、点击左侧“收款人关系票据”的按钮，将显示所有自己作为票据的收款人而相关的所有票据，点击“详细信息”按钮，即可查看票据的详细信息。 3、在“详细信息”按钮右侧，有发起贴现功能按钮。 4、点击“贴现”按钮，系统会自动提交贴现请求。
业务规则： 公司用户需要进行正确的贴现操作，选择正确的操作票据。

（6）处理待承兑票据

用例描述： “承兑”是银行发布票据后，等待票据承兑人接受承兑的操作。在未承兑时，票据属于 made 状态，在公司用户接受承兑后，票据状态变为正常发行状态，否则票据作废。

事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入公司操作界面。 2、点击左侧“待承兑票据”的按钮，将显示所有待承兑的票据。 3、公司用户可以点击“详细信息”按钮，在操作前先查看票据的详细信息。 4、点击“同意”按钮，票据承兑成功，票据状态变为正常发行状态。 5、点击“拒绝”按钮，票据承兑失败，票据废除。
业务规则： 公司用户在进行承兑操作时，需要看清楚票据信息，进行正确操作。

（7）处理待背书票据

用例描述： 如上述对背书功能的陈述，此功能在票据发起背书请求后，票据的被背书人对背书请求进行同意或拒绝操作。若同意背书，则票据的收款人 ID 以姓名和票据的持票人 ID 及姓名都变为当前公司用户，票据状态由等待背书状态变为正常发行状态。若拒绝背书，则票据会显示“背书失败”信息，票据变为正常发行状态。
事件流描述： 1、公司用户输入账号密码进行登陆，点击“登陆”按钮进入公司操作界面。 2、点击左侧“待背书票据”的按钮，将显示所有待背书的票据。 3、公司用户可以点击“详细信息”按钮，在操作前先查看票据的详细信息。 4、点击“同意”按钮，票据的收款人 ID 以姓名和票据的持票人 ID 及姓名都变为当前公司用户，票据状态由等待背书状态变为正常发行状态。 5、点击“拒绝”按钮，则票据会显示“背书失败”信息，票据变为正常发行状态。
业务规则： 公司用户在进行背书操作时，需要看清楚票据信息，进行正确操作。

3.3 系统的非功能性约束

3.3.1 性能需求

在本票据系统中，由于采用 Go 语言和 Gin 框架等对数据处理友好的技术，因此，可以在一定程度上提高系统的性能。但由于底层区块链网络中每进行一次

操作都需要进行复杂的过程，因此，应合理设置网络中的节点数量、共识机制等，提高系统的吞吐量，尽量满足用户的需求。

3.3.2 安全性需求

为保证此系统的安全性，需要对登陆信息进行报名。在底层区块链中，要保密组织的证书、私钥等文件，从而保证交易操作的安全性。

3.4 系统的设计约束

在开发前期，通过对区块链技术以及 Hyperledger Fabric 技术的了解，Fabric 尽量要允许在 Linux 系统上。需要将系统的日志设置为 debug 类型，以便于查询出出错原因。并且为了方便查询，可将信息存储的 Key 值设置为票据的编号，从而简化查询操作。

3.5 可行性研究分析

3.5.1 技术可行性

票据系统中最重要的是票据信息的安全与真实。通过区块链技术的“去中心”特点，可以将传统大部分电子票据系统的中心化问题解决，从而没有单一的个体去存储票据信息，由此确保了信息存储的安全性。

此外，通过区块链“可溯源”和“不可修改”的特点，可以减少传统票据系统中的，票据信息被任意篡改的弊端，针对票据的所有操作都将被记录在区块链的数据中，并且，可以查询更改票据操作人，从而实现票据操作的安全性。

以上，通过区块链技术可以建立十分安全的票据操作系统。所以此项目的开发在技术上是可行的。

3.5.2 经济可行性

本次采用的区块链技术是基于开源开发平台 Hyperledger Fabric，使用时，只需到官方网站去下载并编译即可使用，无需支付费用。

另外，由于本系统基于区块链技术，区块链技术最好在 Linux 服务器上允许。所以，需要在阿里云租用一台服务器进行开发。阿里云大学生提供一次学生优惠，

可以以很低的价格租用服务器三个月，不管从价格还是从时间上，都满足此项目的开发需要。

3.5.3 操作可行性

本系统开发都是在远程服务器展开的，并且由于此系统基于 B/S 架构，最终用户只需要访问相应的网址，即可使用该系统，无需下载安装相应的应用程序，因此，操作十分方便。并且本系统界面十分友好，学习零门槛，用户可以十分容易地了解如何使用该系统。因此，此系统具有操作可行性。

3.6 本章小结

本章先分析并展示了票据系统的整个流程，然后分别对系统的三大功能模块进行了功能需求分析、非功能需求分析、设计约束，最后对系统的技术可行性、经济可行性、操作可行性进行了分析。较为全面地介绍了此票据操作系统的需求与开发流程。

第 4 章 系统总体设计

4.1 系统总体设计说明

本票据操作系统，基于目前开发方向，考虑到开发难度、开发成本以及用户可用性，决定采用 B/S 架构开发此系统。通过 B/S 架构开发的系统，用户无需下载任何应用程序，只需使用浏览器，输入网址，即可使用此系统。

4.2 系统总体架构设计

基于区块链技术的票据操作系统的总体架构如图 4.1 所示：

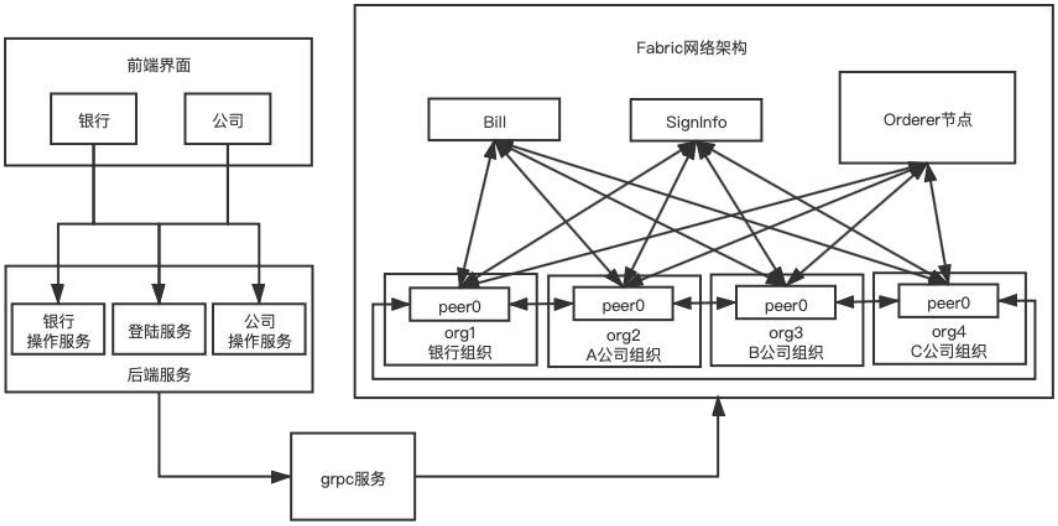

图 4.1 系统总体架构图

该票据操作系统总体采用前后端分离的设计。开发前端界面采用 Vue 框架和 ElementUI 前端开发库进行前端界面开发。总共分为登陆界面、银行用户操作界面和公司操作界面三个界面。前端用户通过操作对应的功能，向后端发送请求。后端服务器通过路由设置，接收对应的请求。后端服务器建立 SDK，对前端发起的不同请求，去执行后端中不同的封装函数，在封装好的函数中，SDK 执行语句调用智能合约中的函数，并传递需要的参数，从而实现与区块链的交互操作。区块链中的 Peer 背书节点收到交易请求后，会对交易进行背书，再将背书的数据集发送给 Orderer 节点。Orderer 节点对交易进行排序，然后发送给 Committer 节

点，此节点对背书数据集进行验证，通过验证后，执行相应的交易。最终，Peer 节点通过共识机制，为每个 Peer 节点同步相同的账本信息。

在前端界面中，用户在填写好信息，触发请求发送监听后，前端程序会根据用户的 ID 和用户填入的信息，将这些信息绑定信息结构模型，然后发送一个 RESTful 风格的 HTTP 请求。在此请求中，包含所要访问后端函数的地址，并且包含函数参数的 JSON 字符串，在后端执行功能后，将数据再返回前端，前端将数据显示在界面上。前端流程如图 4.2 所示。

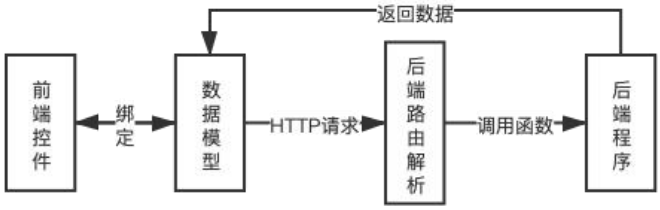

图 4.2 前端设计架构图

在后端程序中，使用 Gin 框架和 Go 语言进行开发。Go 语言本身提供 HTTP 工具包，因此可以之间满足发送服务请求的要求。后端程序中分为设置 SDK 和通过 SDK 与智能合约交互两大部分。通过配置文件，可以通过 Fabric 中的 Gateway 工具包创建 SDK。也可以调用 Gateway 工具包向智能合约发送交易请求，并收到交易结果。在编写后端程序时，我们将 SDK 和 SDK 与区块链交互的操作依照业务层功能封装到函数中，并建立路由。当后端程序收到前端发送的 HTTP 请求后，即可根据路由地址访问不同的函数，使用 SDK 与区块链交互，最终再通过 HTTP 请求，将交易执行结果返回前端。后端流程如图 4.3 所示。

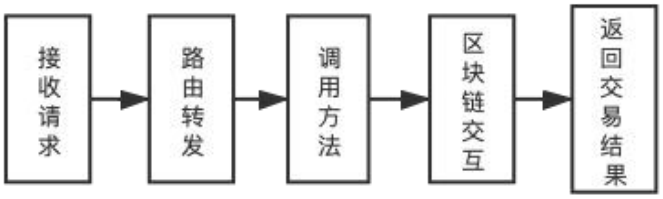

图 4.3 后端设计架构图

Fabric 底层网络中设置了四个组织，分别为银行、A 公司、B 公司、C 公司，四个组织共用一个 Peer 节点。并且设置一个 Orderer 排序节点，从而搭建功能良好而稳定的区块链网络。

4.3 系统主要功能模块

此票据操作系统主要包含三大模块，分别为登陆模块，银行用户功能模块，公司用户功能模块。登陆模块负责银行和公司登陆的第一层验证，并且帮助其进入不同的操作界面。银行用户操作可以查看所有票据信息，查询所有票据的历史变更信息，发布票据以及处理待贴现票据。公司用户可以查看与自己有关的票据信息，并且对作为收款人或持票人关系的票据进行贴现或背书操作。同时，也有处理待承兑，待背书票据的功能。系统主要功能模块如图 4.4 所示：

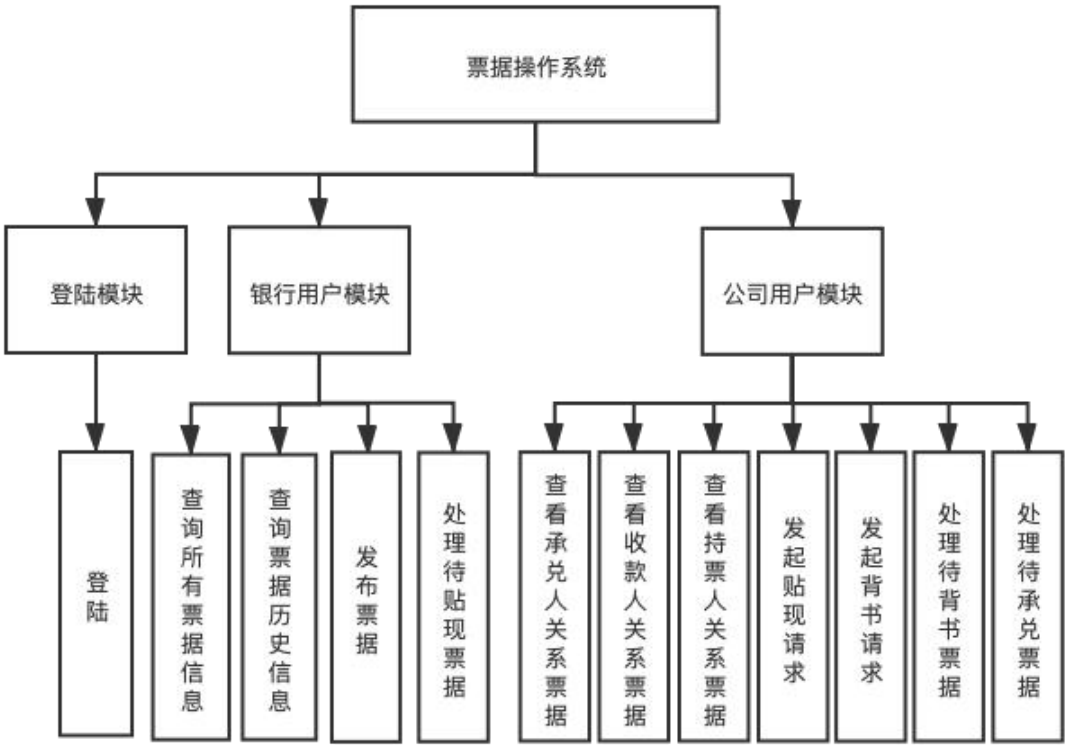

图 4.4 系统主要功能模块图

登陆模块：负责用户的登陆操作。

银行用户模块：具有对票据的信息和历史变更信息查询以及发布票据和处理待贴现票据的功能。

公司用户模块：具有查询与自己相关的票据信息的查询，发起背书请求、发起贴现请求、处理待背书票据和处理待承兑票据的功能。

4.4 系统流程

我们分别从承兑、背书、贴现三个功能的角度，通过分析银行和公司用户进行这三个功能的流程来分析整个系统的流程。

承兑是在银行发行新票据时需要进行的操作。银行填写票据信息，并提交信息。此时，系统会自动向新票据的待承兑人发送承兑请求，此时新票据的状态为“做成”状态。待承兑人可以在自己的账号中查询到待背书票据，可以选择同意承兑，此时票据的债务关系成立，票据状态改为“交付”状态，新票据在票据系统中正常流通。待承兑人也可以徐选择拒绝，此时票据的状态变为“废除”状态，并显示信息“票据承兑失败”。整个流程如图 4.5 所示。

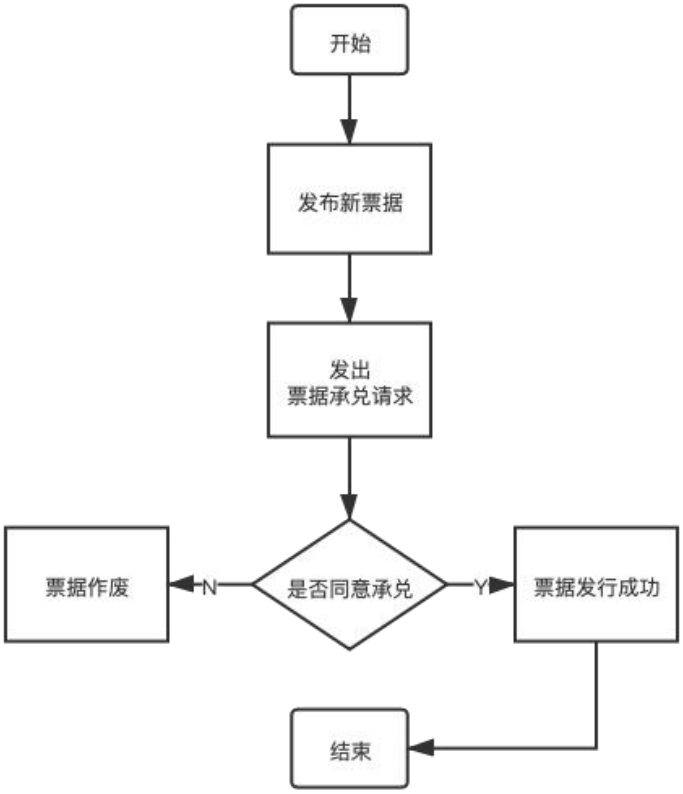

图 4.5 承兑功能流程图

背书是票据的收款人或持有人对票据的收款权和持有权转让的操作。根据公司的业务需求，票据的收款人或持有人可以对票据进行背书操作，从而转移债务关系。首先，公司用户在“收款人关系票据”列表中可以查询到自己作为票据收款人关系的所有票据。再点击“背书”按钮，则弹出填写页，公司用户需要在此填写

被背书人的 ID 和名称，点击提交即可发出背书申请，票据会变为“待背书”状态。后台会自动检查被背书人是否有效，比如，被背书人不可为自己和承兑人，否则无法提交背书申请。被背书人在自己的系统中可点击“待背书票据”按钮，即可查看自己作为票据被背书人的所有票据。如果选择同意背书，票据的收款人或持有人的 ID 和名称都将变为被背书人的 ID 和名称。显示“背书成功”信息，票据转为“发行”状态，票据回归正常票据状态。如果选择拒绝背书，则显示“背书失败”信息，票据也转为“发行”状态。整个流程如图 4.6 所示。

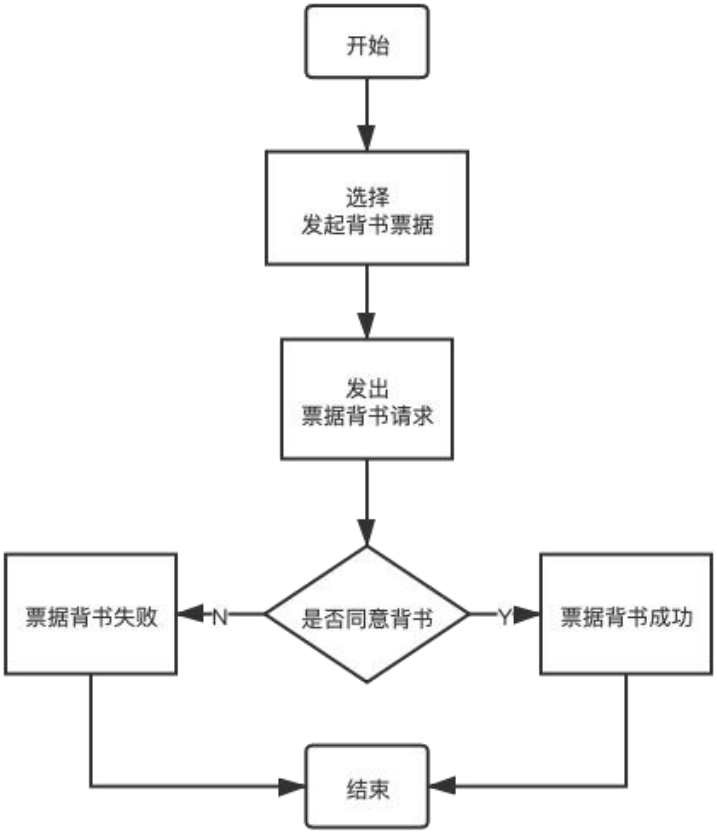

图 4.6 背书功能流程图

贴现同背书操作一样，是票据的收款人或持有人对票据的收款权和持有权转让的操作。但是此刻转移关系对象为银行。票据的收款人或持有人如果想要尽快结算票据，可以将债务关系转移到银行，并对银行进行一定的补偿，从而达到缩短票据结算时间的目的。首先，公司用户在“收款人关系票据”列表中可以查询到自己作为票据收款人关系的所有票据。再点击“贴现”按钮，系统则自动向银行发起贴现请求，此时票据状态变为“等待贴现”状态。此时银行用户可以在操作界面点击“待贴现票据”按钮，查询所有的待贴现票据。如果银行同意进行贴现，票据

的收款人或持有人的ID和名称都将变为银行的ID和名称。显示“贴现成功”信息，票据转为“发行”状态，票据回归正常票据状态。如果选择拒绝贴现，则显示“贴现失败”信息，票据也转为正常“发行”状态。整个流程如图 4.7 所示。

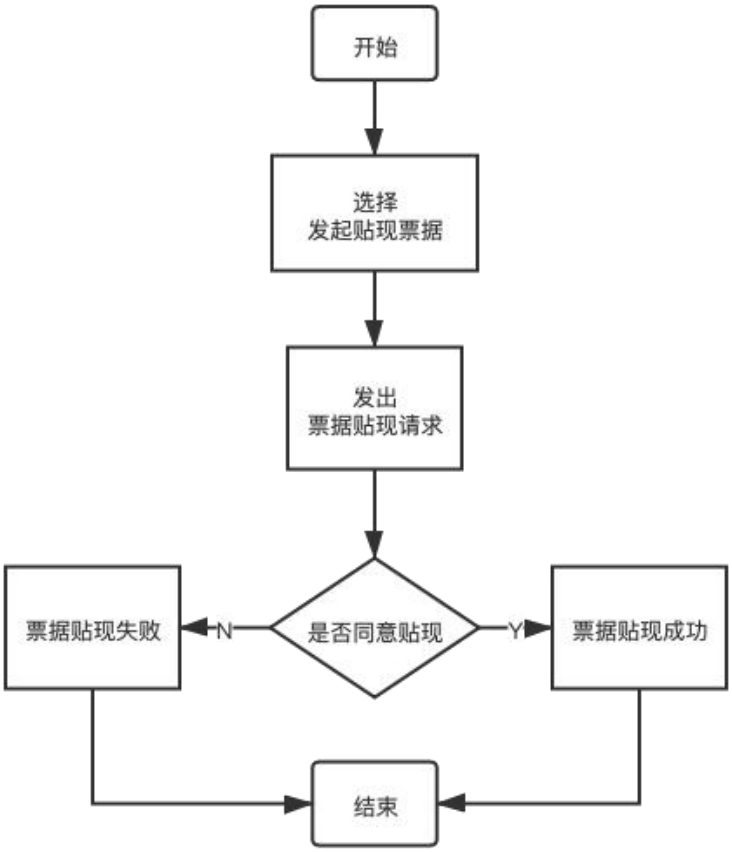

图 4.7 贴现功能流程图

4.5 数据实体存储结构

区块链底层中的数据库采用 CouchDB，与其他数据库的使用有一些不同。但是在票据系统中设计票据和用户的数据结构与以往项目类似，也可以用 E-R 图表示实体之间的关系和数据的存储结构。在 E-R 图中，主要表示了银行用户和公司用户，在开票、承兑、背书和贴现的操作过程中，对于票据操作的实体间的联系。由于本系统中，主要是对票据的处理过程，因此，在 E-R 图的表示中，主要是对票据处理操作的关系。其关系如图 4.8 所示：

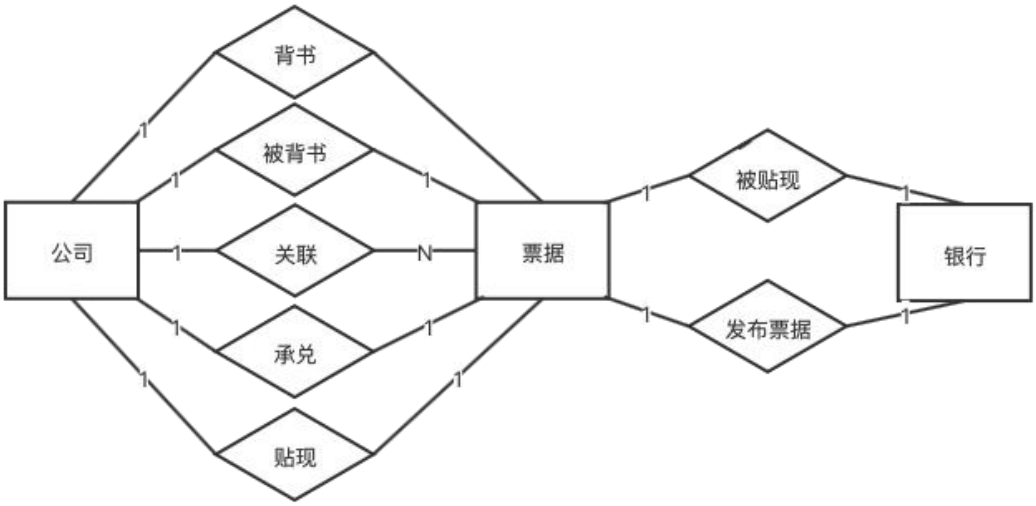

图 4.8 系统 E-R 图

票据是公司用户和银行用户操作的主要实体, 主要具有票据编号、票据金额、票据类型、票据发布时间、票据到期时间、票据开票人 ID 和名称、票据承兑人 ID 和名称、票据收款人 ID 和名称、票据持有人 ID 和名称、票据被背书人 ID 和名称、票据信息和票据状态等信息，如图 4.9 所示：

图 4.9 票据实体属性图

发布票据操作包含票据编号、票据金额、票据类型、票据发布时间、票据到期时间、票据开票人 ID 和名称、票据承兑人 ID 和名称、票据收款人 ID 和名称、票据持有人 ID 和名称、票据被背书人 ID 和名称、票据信息和票据状态等信息。如图 4.10 所示：

图 4.10 发布票据操作实体属性图

背书操作包含票据收款人 ID 和名称、票据持有人 ID 和名称、票据被背书人 ID 和名称、票据信息和票据状态等信息。如图 4.11 所示：

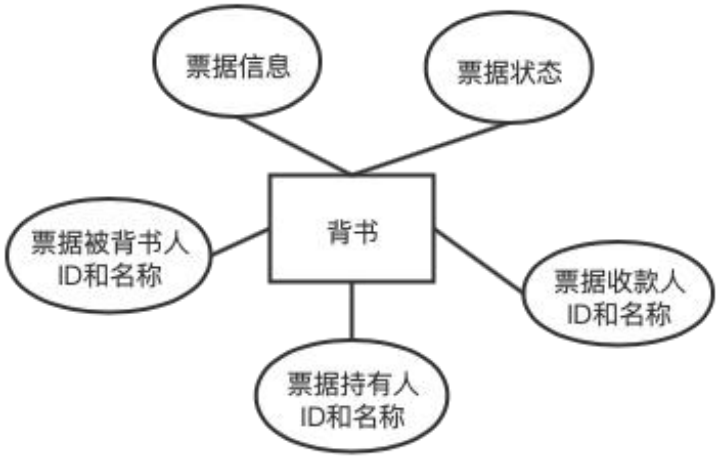

图 4.11 背书操作实体属性图

承兑操作包含票据承兑人 ID 和名称、票据信息和票据状态等信息进行操作。如图 4.12 所示：

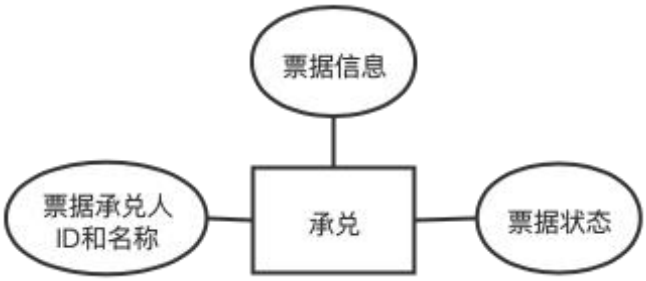

图 4.12 承兑操作实体属性图

贴现操作包含票据收款人 ID 和名称、票据持票人 ID 和名称、票据信息和票据状态等信息进行操作。如图 4.13 所示：

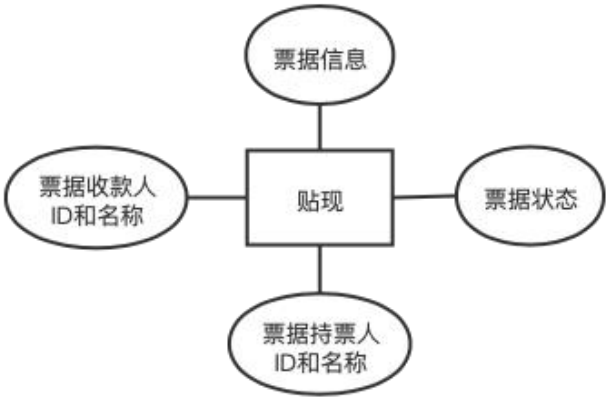

图 4.13 贴现操作实体属性图

4.5 本章小结

在本章中，我们明确了基于区块链技术的票据操作系统的总体设计，分别从系统的总体架构，系统中包含的功能模块，根据系统功能模块而划分的系统流程和数据结构四个方向介绍项目，从各个方面说明票据操作系统的设计四类，为下一章中系统的详细设计提供坚实的基础。

第 5 章 系统详细设计

5.1 登陆模块详细设计

5.1.1 前端设计

登陆模块的主要功能是作为此票据系统的第一道屏障。并且，当用户输入自己的账号和密码后，点击登陆按钮后，可以自动判断用户身份，将用户跳转到自己的操作页面。在登陆页面，有两个输入框，用户可以分别输入自己的账号和密码，输入完成后，点击登陆按钮。前端包含对输入信息的验证处理，与前端交互，从而根据返回条件，对账号进行判断。如果是银行账号，则跳转到银行用户操作界面，否则跳转到公司用户界面。

5.1.2 后端设计

后端采用 Go 语言进行开发，并且使用 Gin 框架对后端系统进行设计。后端主要分为设置路由，创建 SDK，使用 SDK 调用链码三大部分。首先，创建群组路由，将各个方面功能的方法根据业务逻辑进行划分，从而将路由分为三个路由群组，分别为登陆功能路由群组、银行用户功能路由群组和公司用户路由群组，分别根据方法的作用，设置路由路径，等待接收前端传来的请求。其次，创建 SDK，SDK 相当于链码操作的一个客户端。配置文件路径，根据配置信息去找到当前组织的证书、签名、密钥等文件，从而获得对账本操作的权限，完成 SDK 的创建。最后，是使用 SDK 调用链码。在封装好的方法中，调用 SDK 发送请求，此时，请求函数的参数包括智能合约中所要调用的方法的名称，以及智能合约中方法的参数，在执行完成此交易后，返回交易信息，最后在后端的封装函数中，将此信息返回前端。

在登陆操作模块中，后端封装函数，去区块链底层查询内置的票据系统的用户信息，返回所有用户信息后，以账号和密码为准在所有用户中进行搜索，如果包含此用户且其输入的账号和密码正确，则返回此账号的所有信息，包括账号、密码、公司名称、公司 ID。后端设置的查询登陆信息的接口信息如表 5.1 所示：

表 5.1 登陆模块后端接口设计表

接口路由	接口描述	参数	参数描述	参数例子	方法名

/A1/admin/admintest	查询系统账号信息，对登陆用户的账号和密码进行验证	(string)	用户信息结构体 json 字符串	{
				"Username":"alice",
				"Password":"123456",
				"CompanyName":"A 公司",
				"CompanyId":"acmid"
				}
					admintest

5.1.3 Fabric 网络设计

该部分的设计主要是根据用户划定组织，根据需求设置 Peer 节点和通道的数量。本系统中，一共设置了一个通道、一个 Orderer 节点、一个 Peer 节点和四个组织，四个组织分别为银行组织、A 公司组织、B 公司组织和 C 公司组织。由于此系统的规模较小，并且根据业务要求，无需设置隐私通道，所以只设置了一个 Orderer 节点一个 Peer 节点和一个通道。对银行组织、A 公司组织、B 公司组织和 C 公司组织设置的域名分别是 peer0.org1.example.com 、peer0.org2.example.com、peer0.org3.example.com、peer0.org4.example.com。分别监听 7051、8051、11051、12051 四个端口。本服务器上使用 CouchDB 数据库，监听 5984 端口。

该通道中安装有名为 endorse 的链码，链码中包含初始化方法和根据自己需求设置的智能合约方法。根据名称，后台可以调用智能合约中的方法，执行交易。

登陆功能模块主要是在链码中设置 SignInfo 的用户信息数据结构。将各个用户的信息写在智能合约的初始化函数中，通过这样的方式将用户信息内置在系统中，在链码打包、安装的时候，用户信息就被存入 CouchDB 数据库中。

后端调用智能合约中的 QuerySignInfo( )方法，查询区块链中存储的所有用户信息。通过 contractapi 提供的 TransactionContextInterface 接口，可以调用 GetStub( )方法去查询区块链世界状态中的信息，可以通过 GetStateByRange( )查询一定条件范围的信息，这样，将用户信息查询出来后，将其封装为 SignInfo 类型的对象数组，将其返回到后端。

5.2 银行用户模块详细设计

5.2.1 前端设计

银行用户前端模块分为四个功能，分别是查看所有票据信息、查询票据的历史变更记录、发行票据和处理待贴现票据。在系统操作界面左侧的按钮中，可以点击，弹出对应操作的标签页。

查看所有票据信息，向后端发送请求，查询区块链中所有的信息再返回前端显示，用户可以再查询后查看票据的简要信息，再点击“详细信息按钮”，查看票据的详细信息。由于在底层网络中，用户登陆信息和票据信息是存储在一起的，所以前端返回的数据中包括登陆信息，此时，根据信息的属性进行判断，从而过滤掉登陆信息，只显示票据信息。

查询票据的历史变更信息，银行用户可在输入框中输入票据编号，提交，即可查询到该票据的变更历史。基于底层区块链技术，每当票据有所更改，便会自动记录，从而可以搜索票据的历史更改记录。如果填写票据编号错误，则无法查询到结果。

发布票据功能，银行用户在整个票据系统中负责发布新票据。银行用户填入票据编号、票据类型、票据金额、票据发布时间、票据到期时间、票据承兑人的 ID 和姓名、票据收款人的 ID 和姓名、票据持票人的 ID 和姓名，点击提交。后台会对填写的票据编号进行判断，后台中会查询区块链中所有的票据信息，如果新的票据编号与所有票据中有票据编号相同的票据，则会提示已有该票据。如果未重复，则添加成功。前端将填入的信息封装到 Bill 结构表格中，将其发到后端，执行添加票据的操作。

处理待贴现票据，贴现是票据的持票人和收款人将其债务关系转移到银行，从而缩短结算时间的操作。银行会获得一定的补偿。点击“处理贴现票据”按钮，可以查询到所有待贴现的票据。银行可以查看票据的详细信息，之后可以决定同意或拒绝贴现。如果同意贴现，则票据关系自动转移到银行用户本身，如果拒绝，则票据关系不变。

5.2.2 后端设计

后端设计的流程与上述登陆验证大致相同。同样采用 8000 端口进行监听。

在后端中设置了银行用户操作功能的路由群组，在群组中设置了七个路由，

对应七个功能方法，分别是查询所有票据方法、发布票据方法、查询待贴现票据方法、同意贴现处理方法、拒绝贴现处理方法、查询票据交易历史方法。根据前端的不同功能，调用对应方法。

在发布票据方法中，接收传来的写入的票据信息。在票据发布时，需要将票据的状态写为“待承兑”状态。

在查询待贴现票据方法中，将查询票据状态为“待贴现”的所有票据信息，返回前端。

在同意贴现方法中，将从前端传来银行用户的 ID 和名称和被贴现票据的编号，这样将票据的收款人和持有人的 ID 和名称改为银行用户的 ID 和名称，并且将票据状态改为“发布”状态，将票据信息改为“贴现成功”信息，从而完成贴现操作。

在拒绝贴现方法中，将根据前端传来的票据编号，更改票据的状态为“发布”状态，更改票据的信息为“贴现失败”信息。

后端接口信息如表 5.2 所示：

表 5.2 银行用户后端接口设计表

接口路由	接口描述	参数	参数描述	参数例子	方法名
/B1/bank/issueBill	发布新票据	(string, string, string, string, string, string, string)	票据编号、票据类型、票据金额、票据发布时间、票据到期时间、票据承兑人的 ID 和姓名、票据收款人的 ID 和姓名、票据持票人的 ID 和姓名	{"POA10000998","2221","111","2021-01-20","2022-01-20","bank","银行","ccmid", "C 公司","acmid","A 公司", "acmid","A 公司"}	issueBill
/B1/bank/queryAllBills	查询所有票据下信息	无	无	无	queryAllBills

/B1/bank/queryWaitDiscountBills	查询待贴现票据信息	(string)	票据状态	{“dcwaitsigned”}	queryWaitDiscountBills
/B1/bank/agreeDiscountBills	同意贴现票据操作	(string, string, string, string)	票据编号，银行用户 ID 和名称、票据状态，票据信息	{“bank”, “银行”, ”public”, ”discountsuccess”}	agreeDiscountBills
/B1/bank/aDisagreeDiscountBills	拒绝贴现票据操作	(string, string, string)	票据编号、票据状态、票据信息	{“POA10000998”, ”public”, ”discountfail”}	aDisagreeDiscountBills
/B1/bank/queryHistoryById	查询票据变更历史操作	(string)	票据编号	{“POA10000998”}	queryHistoryById

5.2.3 Fabric 网络设计

与登陆验证的 Fabric 网络设计过程相似，将使用 org1 作为银行用户注册身份，使用其 SDK 调用链码中的方法，从而完成对智能合约的操作。

5.3 公司用户模块详细设计

5.3.1 前端设计

银行用户前端模块分为九个个功能，分别查看待承兑票据功能、同意承兑功能、拒绝承兑功能、查看待背书票据功能、同意背书功能、拒绝背书功能、查看自己作为票据承兑人、收款人、持票人关系的票据功能、申请背书功能、申请贴现功能。

查看用户作为票据承兑人、收款人、持票人关系的票据功能，可点击公司用

户操作界面的左侧按钮，点击即可分别查询对应身份的所有票据，可以查看票据的详细信息。

申请背书功能和申请贴现功能，在查看收款人关系票据按钮中，用户进行操作之前，需要详细查看票据的信息。如果同意背书申请，则弹出输入框，用户需要输入票据的被背书人 ID 和名称，输入完成后，前端会对输入信息进行验证，输入的被背书人不能为自身和此票据的承兑人，否则无法提交。点击贴现按钮，则自动发出贴现请求，等待银行进行贴现操作。

5.3.2 后端设计

后端设计的流程与上述登陆验证大致相同。同样采用 8000 端口进行监听。

在后端中设置了公司用户操作功能的路由群组，在群组中设置了十一个路由，对应十一个功能方法，分别是查看待承兑票据方法、同意承兑方法、拒绝承兑方法、查看待背书票据方法、同意背书方法、拒绝背书方法、查看自己作为票据承兑人、收款人、持票人关系的票据方法、申请背书方法、申请贴现方法。

查看所有待承兑票据方法，主要根据公司用户的 ID 和票据状态为“made”两个属性进行查询。依照这两个属性，可以查询到此用户作为票据承兑人的相关的待承兑票据。

同意承兑方法，即根据票据编号，将此票据状态改为“public”状态。

拒绝承兑方法，根据票号，将此票据的状态改为“billfail”表示票据作废，将票据信息改为“waitpayfail”信息，保留票据编号、状态和信息属性，其他抹除。

查看所有待背书票据，主要根据公司用户的 ID、票据被背书人 ID 和票据状态为“made”三个属性进行查询。依照这三个属性，可以查询到此用户作为票据被背书人的相关的待背书票据。

同意背书方法，依照票据编号，将此票据的票据收款人 ID 和名称、票据持有人 ID 和名称改为票据的被背书人 ID 和名称。修改票据状态为“public”，修改票据信息为“endorsesuccess”。

拒绝背书方法，依照票据编号，将票据的被背书人 ID 和名称抹除，修改票据状态为“public”，修改票据信息为“endorsefail”。

查看自己作为票据承兑人、收款人、持票人关系的票据方法，即依照公司用户的 ID 去分别对应票据的三个身份的 ID，若对应，则查询相关所有票据。

发出背书申请方法，依照票据编号，票据被背书人 ID 和名称，修改对应票据的被背书人 ID 和名称，修改票据状态为“enwaitsign”。

发出贴现申请方法，依照票据编号，将票据的状态改为“dcwaitsign”状态。

后端接口信息如表 5.3 所示：

表 5.3 公司用户后端接口设计表

接口路由	接口描述	参数	参数描述	参数例子	方法名
/C1/company/checkWaitPayBills	查看待承兑票据	(string, string)	票据承兑人的 ID，票据状态	{"acmid","waitpay", }	checkWaitPayBills
/C1/company/agreePay	同意承兑	(string, String, string)	票据编号，票据状态，票据信息	{"POA10000998", "public", "waitpaysuccess"}	agreePay
/C1/company/disagreePay	拒绝承兑	(string, String, string)	票据编号，票据状态，票据信息	{"POA10000998", "billfail", "waitpayfail"}	disagreePay
/C1/company/checkAllPayBills	查看自己作为承兑人身份的票据	(string)	公司用户 ID	{"acmid"}	checkAllPayBills
/C1/company/checkAllAcceptBills	查看自己作为收款人身份的票据	(string)	公司用户 ID	{"acmid"}	checkAllAcceptBills
/C1/company/checkAllHoldBills	查看自己作为持票人身份的票据	(string)	公司用户 ID	{"acmid"}	checkAllHoldBills

/C1/company/endorseBill	发出背书申请	(string, string, string, string)	票据编号，票据被背书人 ID 和名称，票据状态	{"POA10000998", "acmid","A 公司", "enwaitsign"}	endorseBill
/C1/company/checkWaitEndorseBills	查询待背书票据	(string, string)	被背书人 ID，票据状态	{"acmid","dcwaitsign"}	checkWaitEndorseBills
/C1/company/agreeEndorseBill	同意背书	(string, string, string)	被背书人 ID 和名称、票据状态	{"acmid","A 公司", "dcwaitsign"}	agreeEndorseBill
/C1/company/disagreeEndorseBill	拒绝背书	(string, string)	票据状态、票据信息	{"public", "endorsefail"}	disagreeEndorseBill
/C1/company/discountBill	发出贴现申请	(string)	票据状态	{"dcwaitsigned"}	discountBill

5.3.3 Fabric 网络设计

与登陆验证的 Fabric 网络设计过程相似，将使用 org2 作为公司用户注册身份，使用其 SDK 调用链码中的方法，从而完成对智能合约的操作。

5.4 数据结构详细设计

在 Fabric 中，信息存储在区块链中的每一个区块中，每一个区块，包含区块的高度、本区块的哈希值和前一个区块的哈希值，还包括本区块中存储的交易信息。并且还包含本区块创建的时间、创建者的私钥、证书和签名。区块通过本区块中包含的前一个区块的哈希值，将区块连接在一起，形成区块链。如下图 5.1 所示：

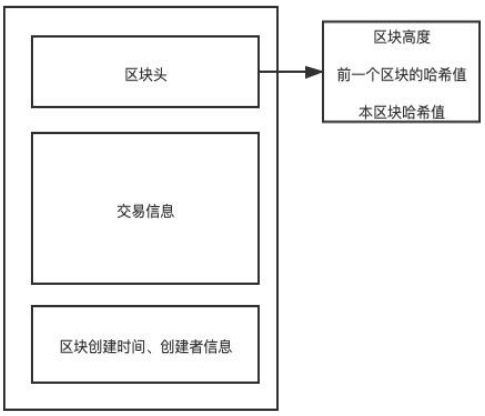

图 5.1 Fabric 的区块存储结构

在 Fabric 中，还包含世界状态数据库。世界状态数据库可以存储区块链交易的最新信息，用户可以方便、快捷地获取数据的最新值。Fabric 默认采用 LevelDB 作为存储世界状态的数据库，在本票据系统中则采用 CouchDB 数据库。在 CouchDB 数据库中，是以 Key 和 Value 进行存储的，value 则是信息的 Json 字符串的形式。为了方便查询，存储时将 Key 值设置为票据编号。因为采用 JSON 字符串格式进行存储，不必考虑主键问题，如果想要根据属性查询信息，只需构造 selector 语句，将此语句在查询世界状态时作为参数传入，就可以根据条件查询相关的信息的世界状态。Fabric 账本结构如图 5.2 所示：

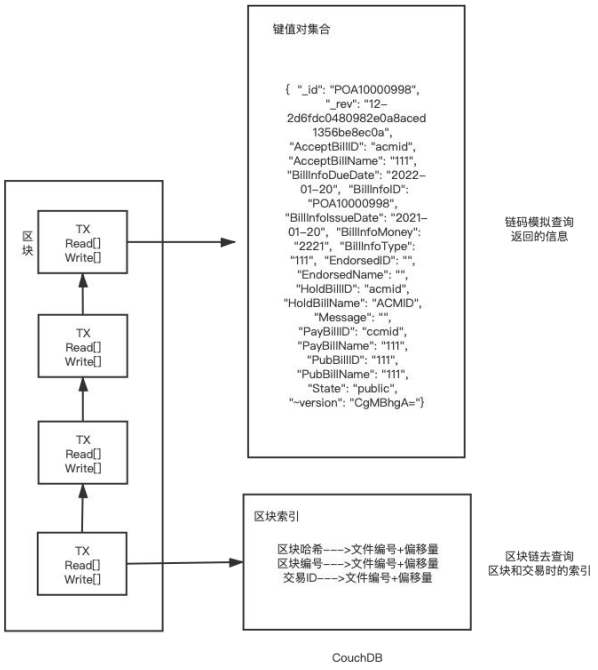

图 5.2 Fabric 账本结构图

1. 用户信息 SignInfo，用于存储票据系统的用户信息，如表 5.4 所示：

表 5.4 用户信息数据结构

属性名	类型	说明	允许为空
Username	String	用户账号	否
Password	String	用户密码	否
CompanyName	String	用户名称	否
CompanyId	String	用户 ID	否

2. 票据信息 Bill，用于存储票据系统的票据信息，如表 5.5 所示：

表 5.5 票据信息数据结构

属性名	类型	说明	允许为空
BillInfoID	String	票据编号	否
BillInfoMoney	String	票据金额	否
BillInfoType	String	票据类型	否
BillInfoIssueDate	String	票据发布时间	否
BillInfoDueDate	String	票据到期时间	否
PubBillID	String	出票人 ID	否
PubBillName	String	出票人名称	否
PayBillID	String	承兑人 ID	否
PayBillName	String	承兑人名称	否
AcceptBillID	String	收款人 ID	否
AcceptBillName	String	收款人名称	否
HoldBillID	String	持票人 ID	否
HoldBillName	String	持票人名称	否
EndorseID	String	被背书人 ID	是
EndorseName	String	被背书人名称	是
Message	String	票据消息	是
State	String	票据状态	是

5.5 智能合约设计及系统时序图

智能合约是区块链网络中，执行交易时，用户遵循的交易规范。开发者编写好智能合约后，将其部署到区块链网络中，便可以无监管地，自动执行用户提交的交易。在本票据系统中，由于大部分只是针对票据的操作，所以只设置了一个endorse 链码。在链码中，由于要对票据进行操作，首先要明确票据的数据结构，在其中声明票据结构体，并声明需要的变量。然后需要编写智能合约中的具体方法。在智能合约方法中，首先要引入 contractapi 的 TransactionContextInterface 接口，此接口中提供了针对区块链的操作方法，如读写数据。在参数写明需要的变量，从后端调用智能合约时，将参数传入智能合约方法中。在智能合约方法内部，需要进行数据检验，包括检查参数个数，检查数据格式，将数据序列化，最后调用交易方法，传入参数，执行交易，返回结果。链码编写过程如图 5.3 所示：

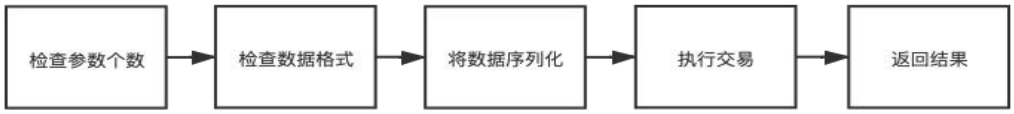

图 5.3 链码编写过程

具体系统流程在此以发出票据背书申请为例，展示详细的系统流程。

首先，在前端界面，公司用户在票据收款人关系中查询到自己作为票据收款人关系的所有票据。在查看票据信息后，对特定票据发出背书请求，在发出请求前需要填写被背书人 ID 和名称。发出背书请求后，会调用后端的背书请求函数，并且发送 form 表单以传递参数。

在后端，首先根据路由，定位到背书功能函数，获取表单数据，使用 SDK 调用智能合约中的背书功能方法，并传递表单中的数据，等待返回数据。执行智能合约中的方法时，根据传入的参数，如票据编号，票据的被背书人 ID 和名称等，将其封装为新的 Bill 对象，将该票据在区块链网络中的票据信息进行更新。这就是背书功能的整个流程。

此功能的时序图如图 5.4 所示：

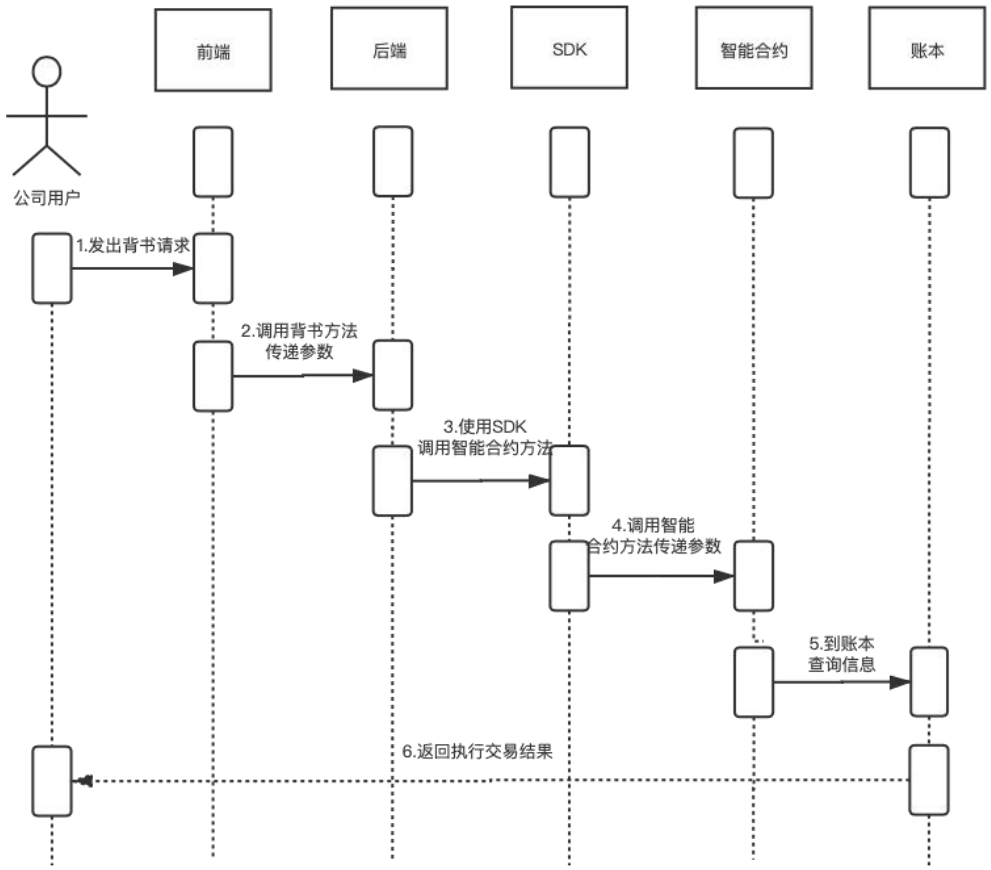

图 5.4 背书功能系统时序图

5.6 本章小结

本章是在对票据操作系统的总体设计和进行需求的详细分析后，对票据系统的开发可行性，对每个功能模块进行详细计划，对每个数据结构和系统的时序图进行了完整且详细的介绍，同时为下一章系统在代码层面的实现打下了良好的基础。

第 6 章 系统功能实现

6.1 系统开发条件

本次票据操作系统的系统规模较小，设置一个 Peer 节点和四个组织对票据的相关操作进行简单明确的演示。因此，在此只需要租用一台阿里云服务器，并且在服务器上按照 Hyperledger Fabric 源码、Go 语言环境、Vue 环境和 Docker 等开发工具。服务器的具体配置和开发软件的版本号如表 6.1 所示：

表 6.1 服务器配置表

类型	服务器 IP	名称	描述
硬件环境 (阿里云服务器)	115.28.136.131	操作系统	CentOS 7.3
		内存	2GB
		CPU	1 核
		带宽	1Mbps
软件版本	115.28.136.131	Go	最低版本：1.11 使用版本：1.15.10
		Docker	最低版本：17.06.2-ce 使用版本：20.10.6-ce
		Docker-compose	最低版本：1.11.0 使用版本：1.27.3
		Fabric	使用版本：2.2.0
		CouchDB	使用版本：2.1.1

6.2 搭建 Hyperledger Fabric 网络

在整个系统的开发过程中，由于所有的数据存于区块链网络中，并且所有功

能的实现也需要通过智能合约来实现。所以，首先要进行的是搭建 Hyperledger Fabric 网络。但是搭建 Hyperledger Fabric 网络是一个复杂的过程，并且，一开始并没有对区块链的知识有过了解。所以，整个搭建 Hyperledger Fabric 网络是开发过程中最困难的一步。最后，经过对 Hyperledger Fabric 给出的事例的学习，了解了需要通过编写配置文件，并逐步创建证书文件、创建创世区块、创建锚节点、创建通道、部署链码、批准链码部署。下面，将对搭建 Hyperledger Fabric 网络的过程进行具体说明。

6.2.1 编译 Hyperledger Fabric 源码

此过程的目的是通过到网上下载 Hyperledger Fabric 源码并编译代码，获取 Hyperledger Fabric 网络的搭建工具，在将 Hyperledger Fabric 源码编译过后，会在 Hyperledger Fabric 文件中的 bin 文件夹下生成可执行文件, 用于搭建区块链网络。

首先，到 Github 上下载 Fabric 的源代码文件到本地。然后进入下载的文件，将其版本切换为 2.2 版本。然后修改 bootstrap.sh 中的 fabric-samples 的下载地址，将其改为 Gitee 的下载地址。执行修改过后的 bootstrap.sh 二进制文件。此时会生成 Hyperledger Fabric 的示例文件，再将 Fabric 和 Fabric-ca 的压缩包传入 Fabric-samples 文件夹中，解压这两个文件，即可在 fabric-samples 下面的 bin 文件夹找到搭建 Hyperledger Fabric 网络的工具文件。下面将编写配置文件，通过这些工具文件搭建区块链网络。

6.2.2 生成证书文件

首先要生成区块链网络中的证书文件。编写 Orderer 节点和 org1、org2、org3 和 org4 组织下 Peer 节点的配置文件。其中定义了 Orderer 节点的域名和各组织内 Peer 节点的域名信息，用户的个数和 Peer 节点的个数。配置文件的内容如下：

```
OrdererOrgs:
  - Name: Orderer
    Domain: example.com
    EnableNodeOUs: true
    Specs:
      - Hostname: orderer
        SANS:
            - localhost
PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
```

```
    Template:
      Count: 1
      SANS:
        - localhost
  PeerOrgs:
    - Name: Org2
      Domain: org2.example.com
      EnableNodeOUs: true
      Template:
        Count: 1
        SANS:
          - localhost
  PeerOrgs:
    - Name: Org3
      Domain: org3.example.com
      EnableNodeOUs: true
      Template:
        Count: 1
        SANS:
          - localhost
  PeerOrgs:
    - Name: Org4
      Domain: org4.example.com
      EnableNodeOUs: true
      Template:
        Count: 1
        SANS:
          - localhost
```

编写完 Orderer 和各组织的配置文件后，即可通过 bin 文件夹中的 cryptogen 文件执行此配置文件，从而生成每个 Orderer 节点和 Peer 节点下对应各组织的证书文件，其执行命令如下：

```
cryptogen generate
        # 指定配置文件位置
        --config=./organizations/cryptogen/crypto-config-org1.yaml
        # 指定输出证书文件的目录
        --output="organizations"
cryptogen generate
        --config=./organizations/cryptogen/crypto-config-org2.yaml
        --output="organizations"
cryptogen generate
        --config=./organizations/cryptogen/crypto-config-org3.yaml
        --output="organizations"
cryptogen generate
        --config=./organizations/cryptogen/crypto-config-org4.yaml
        --output="organizations"
cryptogen generate
         --config=./organizations/cryptogen/crypto-config-orderer.yaml
         --output="organizations"
```

执行完成后，在 test-network 下的 organizations 文件中，可看到这些文件。

6.2.3 生成创世区块

生成创世区块也需要编写配置文件，再通过工具执行此文件。首先要编写一

个 configtx.yaml 配置文件，在其中填写各个组织的名称、ID、证书文件路径、锚节点和组织策略。然后，使用 configtxgen 工具执行 configtx.yaml 配置文件，具体执行语句如下：

```
# 生成系统创世区块
configtxgen
      -profile TwoOrgsOrdererGenesis        # 指定配置文件中的信息
      -channelID systemchannel              # 指定系统通道
      -outputBlock ./system-genesis-block/genesis.block      #指定输出目录
# 生成账本创世区块
configtxgen
      -profile TwoOrgsChannel               # 指定配置文件中的信息
      -channelID mychannel                  # 指定使用通道
      -outputCreateChannelTx ./channel-artifacts/mychannel.tx      #指定输出目录
# 创建锚节点
configtxgen
       -profile TwoOrgsChannel        # 指定配置文件中的信息
       -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx     #指定输出目录
       -channelID mychannel          # 指定使用通道
       -asOrg Org1MSP        # 指定组织信息
# 创建锚节点
configtxgen
       -profile TwoOrgsChannel        # 指定配置文件中的信息
       -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx     #指定输出目录
       -channelID mychannel          # 指定使用通道
       -asOrg Org2MSP        # 指定组织信息
# 创建锚节点
configtxgen
       -profile TwoOrgsChannel        # 指定配置文件中的信息
       -outputAnchorPeersUpdate ./channel-artifacts/Org3MSPanchors.tx     #指定输出目录
       -channelID mychannel          # 指定使用通道
       -asOrg Org3MSP        # 指定组织信息
# 创建锚节点
configtxgen
       -profile TwoOrgsChannel        # 指定配置文件中的信息
       -outputAnchorPeersUpdate ./channel-artifacts/Org4MSPanchors.tx     #指定输出目录
       -channelID mychannel          # 指定使用通道
       -asOrg Org4MSP        # 指定组织信息
```

6.2.4 启动 Fabric 网络

Fabric 网络的运行是在 Docker 容器中的，包括各组织，各组织的 CA 和 CouchDB 数据库。因此，要编写 docker-compose 配置文件将这些 Fabric 网络的组件配置于 Docker 容器中。执行 docker-compose 配置文件命令如下：

6.2.5 创建通道和安装链码

```
MAGE_TAG=latest docker-compose
-f ./docker/docker-compose-test-net.yaml      #指定 docker-compose 配置文件目录
-f ./docker/docker-compose-couch.yaml
-f ./docker/docker-compose-ca.yaml
up -d   # 执行启动 Docker 容器命令
```

经过上述操作后，已经启动了 Fabric 网络。此时，需要创建通道、更新锚节点，并将智能合约打包，安装到通道上并获得各组织对部署智能合约到通道上的许可。

创建通道命令如下：

```
peer channel create
-o localhost:7050   # 连接的 orderer 的地址
-c mychannel        # 指定创建的通道名称
--ordererTLSHostnameOverride orderer.example.com
-f ./channel-artifacts/mychannel.tx         # 指定交易文件目录
--outputBlock ./channel-artifacts/mychannel.block         #指定输出交易信息文件
--tls          # 和 orderer 通信时是否启用 tls
--cafile     # 使用 tls 时，所使用的 Orderer 证书
 /opt/goWork/src/github.com/hyperledger-fabric/scripts/fabric-samples/test-network/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

以 org1 的锚节点为例，更新锚节点命令如下：

```
# 配置基本变量信息
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
export CORE_PEER_ADDRESS=localhost:7051
# 更新锚节点
peer channel update
-o localhost:7050
--tls
--cafile
    ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
-c mychannel
-f ./channel-artifacts/Org1MSPanchors.tx
# 将锚节点交易信息加入账本创世区块中
peer channel join
-b ./channel-artifacts/mychannel.block
```

部署链码到通道的执行命令如下：

```
# 打包智能合约
peer lifecycle chaincode package ${PACKAGENAME} # 打包链码命令
    --path ../EndorseBillProject/fabric/chaincode/go  # 指定链码文件路径
    --lang golang     # 指定智能合约编写的语言
    --label ${PACKAGENAME}_1 #指定打包标签，一般为智能合约文件名称和版本号
# 在指定 Peer 节点上安装链码
peer lifecycle chaincode install ${PACKAGENAME}
# 获得各节点组织的部署链码的许可
# 以下各配置主要为表明链码的详细信息
peer lifecycle chaincode approveformyorg
-o localhost:7050
--ordererTLSHostnameOverride orderer.example.com
--channelID mychannel
--name ${CHAINNAME}
--version ${CHAINCODEVERSION}
--package-id $CC_PACKAGE_ID
--sequence ${CHAINCODESEQUENCE}
--tls
--cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tls
cacerts/tlsca.example.com-cert.pem
# 获取许可后，提交链码，完成部署，在此以 org1 为例
peer lifecycle chaincode commit
-o localhost:7050
--ordererTLSHostnameOverride orderer.example.com
--channelID mychannel
--name ${CHAINNAME} -
-version ${CHAINCODEVERSION}
--sequence ${CHAINCODESEQUENCE}
--tls                                                                    --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tls
cacerts/tlsca.example.com-cert.pem
--peerAddresses localhost:7051 # org1 节点地址
--tlsRootCertFiles    # org1 证书路径
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/
tls/ca.crt
```

此时，已完成区块链网络的搭建，并将链码部署到区块链网络中。以上所有的命令操作与环境变量的配置都将按照顺序写入 shell 文件中，只需只需 shell 文件，即可只需上述所有命令，完整 Hyperledger Fabric 网络的搭建。

6.3 主要功能模块的实现

6.3.1 登陆模块实现

登陆模块使整个票据系统开始的地方。用户通过输入系统内置的账号和密码，系统将自动跳转到其身份下的操作界面。登陆模块页面如下图 6.1 所示：

图 6.1 登陆模块界面

用户登陆功能前端实现如下，在登陆界面输入账号和密码后，点击登陆操作。此时，前端会获取输入信息，将其绑定到拥有 Username 和 Password 属性的表单中，将登陆提交操作封装为 admin 函数，在 admin 函数中，封装 API 请求，将请求连同表单发送到后端，执行 admintest 函数。admintest 函数中向后端查询所有登陆信息，对当前登陆用户的信息进行验证，如果正确，则向前端发回该用户在区块链网络中的所有用户信息，如账号、密码、用户 ID 和名称。此时，由于系统中只包含一个银行用户，所有根据账号判断用户应跳转到哪个操作界面。账号为 admin 的用户，跳转到银行用户操作界面，账号为其他的，跳转到公司用户操作界面。admin 函数中也很进行简单的输入验证，比如判断输入信息是否完整或者输入账号和密码是否正确。admin()函数的代码如下所示：

```
admin(){
  // 对输入进行判空验证
if(this.form.Username !== "" && this.form.Password !== ""){
  // 根据账号名称决定跳转页面
  if(this.form.Username === "admin"){
  // 封装 API，向后端发送请求，调用后端的 admintest 函数，并传递前端输入信息
this.axios.post('http://115.28.136.131:8000/A1/admin/admintest', this.form)
.then(function (res) {
  // 将后端返回的用户信息绑定到 formjs 表单中
  this.formjs = JSON.parse(res.data);
  // 跳转到银行用户界面，并向跳转页面传递用户信息
  this.$router.push({name:"bankInterface",
params:{CompanyName:this.formjs.CompanyName,CompanyId:this.formjs.CompanyId},
requiredAuth:true});
  }.bind(this))
  .catch(function (err) {
  // 如果输入信息错误，则进行判错提示
      alert("请填写正确信息!")
      this.form.Username=""
      this.form.Password=""
  if (err.response){
  //失败
      console.log(err.response)
  }
  }.bind(this))
    }else{
this.axios.post('http://115.28.136.131:8000/A1/admin/admintest', this.form)
  .then(function (res) {
```

```
            this.formjs = JSON.parse(res.data);
            this.$router.push({name:"companyInterface",
params:{CompanyName:this.formjs.CompanyName,CompanyId:this.formjs.CompanyId},
requiredAuth:true});
        }.bind(this))
        .catch(function (err) {
           alert("请填写正确信息!")
           this.form.Username=""
           this.form.Password=""
        if (err.response){
        //失败
            console.log(err.response)
        }
        }.bind(this))
    }
      }else{
          alert("请填写完整信息!")
      }
    }
}
```

后端接收到此 Post 请求后，会调用 admintest()方法。首先使用 ShouldBind 方法将传递到后端的表单进行数据绑定，将其绑定到 SignInfo 类型的变量中。然后使用 SDK 调用智能合约中的 QuerySignInfo 方法，查询所有用户信息，将返回的所有用户信息进行序列化，根据返回信息，与前端传来的信息进行对比，从而判断输入信息是否正确，如果正确，则将对应用户的全部信息发送回前端。后端中的 admintest()方法的代码如下所示：

```
func admintest(ctx *gin.Context) {
    // 绑定传来的 form
    var signinfo SignInfo
    err := ctx.ShouldBind(&signinfo)
    if err != nil {
       fmt.Printf("绑定失败: %s\n", err)
    } else {
       fmt.Printf("绑定成功: %s\n", err)
    }
    // 调用智能合约中的 QuerySignInfo 方法，查询所有用户信息
    results, err := contract.SubmitTransaction("querySignInfo")
    if err != nil {
       fmt.Printf("Failed to submit transaction: %s\n", err)
       os.Exit(1)
    }
    // 声明 SignInfo 类型结构体数组
    var jsonresults []SignInfo
    // 将返回数据序列化，存入上述结构体数组
    json.Unmarshal(results, &jsonresults)
    // 遍历查询结果,如果用户名密码符合，则返回其 signinfo，否则返回失败
    for i := range jsonresults{
    // 如果数据符合条件，返回此 signinfo
       if (jsonresults[i].Username == signinfo.Username && jsonresults[i].Password ==
signinfo.Password){
        fmt.Println(jsonresults[i])
        data, _ := json.Marshal(jsonresults[i])
        ctx.JSON(http.StatusOK,string(data))
```

智能合约中，调用 QuerySignInfo 方法，根据 contractapi.TransactionContextInterface 接口中提供的对区块链中数据的操作方法，根据范围查询，查询系统中所有用户信息。由于查询多个用户的信息，因此需要用 SignInfo 类型的结构体数组接收数据。区块中交易操作后，返回一个数据的迭代器，通过对迭代器中数据的迭代，首先将数据解序列化，然后将其分别加入结构体数组中，操作完成，将结果返回后端。智能合约中 QuerySignInfo 方法代码如下：

```
func (s *SmartContract) QuerySignInfo(ctx contractapi.TransactionContextInterface)
([]SignInfo, error) {

        // 根据范围查询，查询系统中所有的用户信息
        startKey := ""
        endKey := ""
        resultsIterator, err := ctx.GetStub().GetStateByRange(startKey, endKey)
        if err != nil {
           return nil, err
        }

        defer resultsIterator.Close()
        // 声明 SignInfo 类型结构体数组，接收返回数据
        var results []SignInfo
        // 迭代上述查询到的结果，处理其中的信息
        for resultsIterator.HasNext() {
              queryResponse, err := resultsIterator.Next()
              if err != nil {
              return nil, err
                }

        var signinfo SignInfo
        // 将其中一条数据反序列化，存入 SignInfo 类型变量中
        err = json.Unmarshal(queryResponse.Value, &signinfo)
        if err != nil {
           return nil, err
        }
        // 将存好一条信息的 SignInfo 类型变量加入 SignInfo 类型结构体数组
        results = append(results, signinfo)
     }
        // 返回查询到的结果
        return results, nil
   }
}
```

登陆模块功能到此完成。

6.3.2 银行用户模块实现

银行用户模块是针对银行在票据系统中的身份，设置了查看所有票据、发行票据、处理贴现票据和查询票据历史四个功能。其入口界面如图 6.2 所示。

图 6.2 银行用户功能入口

查看所有票据信息是银行对票据系统一种查询与监管的功能，可以查询到底层区块链的所有票据信息。下面将介绍查看所有票据功能，其操作界面如下图 6.3 所示：

图 6.3 查看所有票据功能界面图

此功能前端代码实现如下。查询所有票据信息功能如下。点击查看所有票据按钮后，右侧弹出功能标签页，在其中显示查询到的所有票据信息，点击详细信息按钮，则弹出页面显示票据的详细信息。在点击查看所有票据按钮后，自动执行 searchAllBills( )方法，在其中向后端发送 Post 请求，查询所有票据信息，返回结果后，用 Bill 类型结构体数组接收数据。由于用户信息和票据信息是存在一起的，所以返回的结果包含用户信息，在此，遍历查询到的每组数据，通过判断每组数据的 PayBillID 属性是否为空，从而判断此组信息是否为票据类型信息，如果是，则将其显示到界面上。

```
data() {
  return {
    bills:[{
        BillInfoID : "",
        BillInfoMoney : "",
        BillInfoType : "",
        BillInfoIssueDate : "",
        BillInfoDueDate : "",
        PubBillID : "",
        PubBillName : "",
        PayBillID : "",
        PayBillName : "",
        AcceptBillID : "",
        AcceptBillName : "",
        HoldBillID : "",
        HoldBillName : "",
```

```
            EndorsedID : "",
            EndorsedName : "",
            Message : "",
            State : "",
}],
      finalbills:[]
  };
},
methods: {
  // 查询所有票据信息
searchAllBills(){
// 向后端发送请求，调用 queryAllBills 方法
this.axios.post('http://115.28.136.131:8000/B1/bank/queryAllBills')
      .then(function (res) {
      // res 为返回的查询数据，用 bills 表单进行接收
      this.bills = JSON.parse(res.data);
      // 对返回的所有数据进行筛选，如果包含 PayBillID 则说明为票据信息
      for(let item of this.bills) {
          if (item.PayBillID == ""){
            continue
          }else{
      // 将判断为票据的信息加入 finalbills 表单中，进行显示
          this.finalbills.push(item);
      }}
      //设置下拉框默认值
      }.bind(this))
      .catch(function (err) {
        if (err.response){
      //失败
        console.log(err.response)
       }
      }.bind(this))
},
```

后端接收前端发送的 Post 请求，调用 queryAllBills( )方法，在其中向智能合约中发送交易请求，调用其中的 QueryAllBill( )方法，返回查询结果，将其发送给前端。

```
func queryAllBills(ctx *gin.Context) {
        // 向区块链发送交易请求，调用智能合约中的 queryAllBill 方法，查询票据信息
        results, err := contract.SubmitTransaction("queryAllBill")
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
        // 将查询结果以字符串形式发回前端
        ctx.JSON(http.StatusOK,string(results))
}
```

智能合约中，调用 QueryAllBill( )方法，根据 contractapi.TransactionContextInterface 接口中提供的对区块链中数据的操作方法，根据范围查询，查询系统中所有票据信息。由于查询多个票据的信息，因此需要用 Bill 类型的结构体数组接收数据。区块中交易操作后，返回一个数据的迭代器，通过对迭代器中数据的迭代，首先将数据解序列化，然后将其分别加入结构体数组中，操作完成，将结果返回后端。此智能合约方法与查询用户信息方法类似，在此不作代码展示。

查看所有票据功能到此实现完成。

发布票据功能，是在票据系统中，整个系统的开始，银行仍承担发布票据的操作，之后相关用户即可对票据进行操作。此功能操作图如图 6.4 所示：

图 6.4 发布票据操作界面

发布票据功能前端实现如下。在发布票据功能标签页，填入票据编号、票据金额、票据类型、票据发布时间、票据到期时间、票据承兑人 ID 和名称、票据收款人 ID 和名称、票据持有人 ID 和名称。填写好之后点击发布票据按钮。票据会对其进行判空验证。然后会对票据编号进行判错验证，查询所有票据信息，将输入的票据编号与区块链中存有的所有票据编号进行对比，如果含有此票据，则票据编号重复，发布失败。如果票据编号未重复，则将所有输入信息封装于表单中，并且设置票据发布人本银行用户，发送到后单 issueBill()方法中，存入此票据，发送请求成功，显示发布成功提示。由于已展示过票据类型表单，如下只展示发布票据方法 issueBill( )的内容。

```
issueBill(){
// 对输入信息进行判空操作
      if (this.bills.BillInfoID == "" || this.bills.BillInfoMoney == "" || this.bills.BillInfoType == "" || this.bills.BillInfoIssueDate == "" || this.bills.BillInfoDueDate == "" || this.bills.PayBillID == "" || this.bills.PayBillName == "" || this.bills.AcceptBillID == "" || this.bills.AcceptBillName == "" || this.bills.HoldBillID == "" || this.bills.HoldBillName == ""){
         alert("请填写完整信息！ ")
   }else{
      var isreap = 0
      for(let item of this.allbills) {
      // 判断是否已经存在
      if (item.BillInfoID === this.bills.BillInfoID){
      isreap = 1
      }
   }
      if (isreap === 1){
         alert("已有此票据存在！ ")
      }else{
```

```
      // 将银行费发票人自动存为银行用户
      this.bills.PubBillID = "bank"
      this.bills.PubBillName = "管理员"
this.axios.post("http://115.28.136.131:8000/B1/bank/issueBill", this.bills)
      .then(function (res) {
        alert("发布票据成功!  ")
      //成功
        }.bind(this))
      .catch(function (err) {
      if (err.response) {
        console.log(err.response)
      }
        }.bind(this))
      }
   }
},
```

后端接收前端发送的 Post 请求，调用 issueBill( )方法，在其中使用 ShouldBind 方法将传来的表单数据进行绑定，再向智能合约中发送交易请求，调用其中的 IssueBill( )方法，并传递填写的票据信息，执行交易。

```
func issueBill(ctx *gin.Context) {
        // 绑定传来的 form
        var bill Bill
        err := ctx.ShouldBind(&bill)
      if err != nil {
        fmt.Printf("绑定失败: %s\n", err)
      } else {
        fmt.Printf("绑定成功: %s\n", err)
      }
        // 调用智能合约中的 issueBill 方法，并传递票据信息
        results, err := contract.SubmitTransaction("issueBill", bill.BillInfoID,
          bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate,
bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID,
bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName)
      if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
      }
        fmt.Println(string(results))
}
```

智能合约中，调用 IssueBill( )方法，将所有参数信息包装为 Bill 结构体类型，此时，将票据的被背书人 ID 和名称和票据信息设置为空，将票据状态设置为 made 状态，调用 PutState( )方法，将此票据信息存入区块链中，将票据编号作为 Key 值。

```
func (s *SmartContract) IssueBill(ctx contractapi.TransactionContextInterface, billInfoID
string, billInfoMoney string, billInfoType string, billInfoIssueDate string, billInfoDueDate
string, pubBillID string, pubBillName string, payBillID string, payBillName string,
acceptBillID string, acceptBillName string, holdBillID string, holdBillName string) error {
        //将参数包装为 Bill 结构体类型
        bill := Bill{
        BillInfoID: billInfoID,
        BillInfoMoney: billInfoMoney,
```

```
        BillInfoType: billInfoType,
        BillInfoIssueDate: billInfoIssueDate,
        BillInfoDueDate: billInfoDueDate,
        PubBillID: pubBillID,
        PubBillName: pubBillName,
        PayBillID: payBillID,
        PayBillName: payBillName,
        AcceptBillID: acceptBillID,
        AcceptBillName: acceptBillName,
        HoldBillID: holdBillID,
        HoldBillName: holdBillName,
        EndorsedID: "",
        EndorsedName: "",
        Message: "",
        State: "made",
    }
        //结构体转 json
        billAsBytes, _ := json.Marshal(bill)
        // 以票据编号作为 key 值，将票据信息存入区块链中
        return ctx.GetStub().PutState(bill.BillInfoID, billAsBytes)
}
```

票据发布功能到此实现完成。

处理贴现请求功能，此功能是对公司用户向银行发出贴现请求的处理功能。银行用户可以查看待贴现票据的详细信息，同意或拒绝此贴现请求。处理贴现请求功能操作界面如图 6.5 所示：

图 6.5 处理贴现请求功能操作界面

处理贴现请求功能前端实现如下。在处理贴现请求功能标签页，会显示所有待贴现票据，此功能是由前端向后端发送 Post 请求，调用后端的 queryWaitDiscountBills( )方法，查询票据系统中所有票据状态为“dcwaitsigned”的票据。在银行用户仔细查看票据的信息后，如果决定同意贴现，则向后端发送请求调用 agreeDiscountBills( )方法，并传递新的票据表单，在此表单中，将票据的持票人 ID 和名称和收款人 ID 和名称都变为银行用户的 ID 和名称，由此转移票据的交易关系。若拒绝贴现请求，则向后端发送请求调用 aDisagreeDiscountBills( )方法，对票据的关系人 ID 和名称不做操作。上述查询所有待贴现票据共、 同意贴现功能和拒绝贴现功能分别调用前端的 agreeDiscountBills( )、disagreeDiscountBills( )和 searchAllBills( )方法。在此展示同意贴现功能和查询所有待贴现票据功能的代码，其代码如下所示：

```
// 查询待承兑票据
searchAllBills(){
    // 调用后端 queryWaitDiscountBills 方法，查询所有待贴现票据
  this.axios.post('http://115.28.136.131:8000/B1/bank/queryWaitDiscountBills')
    .then(function (res) {
```

```
        // 用 bills 表单接收票据信息
        this.bills = JSON.parse(res.data);
        }.bind(this))
        .catch(function (err) {
        if (err.response){
        //失败
            console.log(err.response)
        }
        }.bind(this))
},
// 同意贴现操作
agreeDiscountBills(index, row) {
        // 将表格中一行中的票据信息赋值给表单，以向后端传递新的票据信息
        this.sendform.BillInfoID=row.BillInfoID
        this.sendform.BillInfoMoney=row.BillInfoMoney
        this.sendform.BillInfoType=row.BillInfoType
        this.sendform.BillInfoIssueDate=row.BillInfoIssueDate
        this.sendform.BillInfoDueDate=row.BillInfoDueDate
        this.sendform.PubBillID=row.PubBillID
        this.sendform.PubBillName=row.PubBillName
        this.sendform.PayBillID=row.PayBillID
        this.sendform.PayBillName=row.PayBillName
        // 修改票据收款人 ID 和名称为银行用户的 ID 和名称
        this.sendform.AcceptBillID=this.CompanyId
        this.sendform.AcceptBillName=this.CompanyName
        // 修改票据持有人 ID 和名称为银行用户的 ID 和名称
        this.sendform.HoldBillID=this.CompanyId
        this.sendform.HoldBillName=this.CompanyName
        this.axios.post("http://115.28.136.131:8000/B1/bank/agreeDiscountBills",this.sendfor
m)
        .then(function (res) {
            alert("操作成功！ ")
        }.bind(this))
        .catch(function (err) {
        if (err.response) {
            console.log(err.response)
        }
    }.bind(this))
},
```

后端接收三个 Post 请求，调用后端的 queryWaitDiscountBills()、agreeDiscountBills()和 aDisagreeDiscountBills()方法。其中 queryWaitDiscountBills()通过调用智能合约方法 queryWaitDiscountBills()，依据票据状态查询所有待贴现票据。agreeDiscountBills()方法调用 agreeDiscountBill()方法，将修改过票据人关系的新票据信息也写入参数，在智能合约方法中，将票据的状态改为"public"，票据消息改为"discountsuccess"，其他信息按照表单，对原始票据信息进行修改。aDisagreeDiscountBills()方法与 agreeDiscountBills()方法操作类似，去调用智能合约方法 aDisagreeDiscountBill(),但只将票据的状态改为"public"，票据消息改为"discountfail"，不对票据关系进行修改。在此展示后端 queryWaitDiscountBills()和 agreeDiscountBills()的编写。其代码如下：

```
// 查询待贴现票据
// -查询
func queryWaitDiscountBills(ctx *gin.Context) {
```

```
        // 绑定传来的 form
        var getbill Bill
        err := ctx.ShouldBind(&getbill)
    if err != nil {
        fmt.Printf("绑定失败: %s\n", err)
    } else {
        fmt.Printf("绑定成功: %s\n", err)
    }
        // 调用智能合约方法 queryWaitDiscountBills，查询所有待贴现票据
        results, err := contract.SubmitTransaction("queryWaitDiscountBills")
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
        ctx.JSON(http.StatusOK,string(results))
}
// -处理
// -同意贴现：更改状态为 Public 且更改 Message 为 DiscountSuccess,并且将 收款方面的信息更改为银行
func agreeDiscountBills(ctx *gin.Context) {
    // 绑定传来的 form
    var bill Bill
    err := ctx.ShouldBind(&bill)
    if err != nil {
        fmt.Printf("绑定失败: %s\n", err)
    } else {
        fmt.Printf("绑定成功: %s\n", err)
    }
        // 调用智能合约方法 agreeDiscountBill，并将新的票据信息传入方法中
        results, err := contract.SubmitTransaction("agreeDiscountBill", bill.BillInfoID, bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate, bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID, bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName)
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
 }
}
```

智能合约中，分别调用 queryWaitDiscountBills( )、agreeDiscountBill( )和 aDisagreeDiscountBills( )三个方法。分别根据票据系统规则，修改票据的状态和票据的消息信息。由于操作过程与上述功能的操作过程类似，在此不再展示代码。

处理贴现请求功能到此实现完成。

查询票据历史信息功能，银行用户可以查询票据发布后所有信息的历史变更数据，只需输入票据编号，即可查询。查询票据历史信息功能如下图 6.6 所示：

图 6.6 查询票据历史功能操作界面

查询票据历史功能前端实现如下，在前端票据编号输入框，输入所要查询票据的票据编号，点击查询，即可显示票据信息变更的历史记录。此功能是向后端发送 Post 请求，调用 queryHistoryById( )方法，并将票据编号绑定到表单中也发送到后端，查询历史。前端中 searchHistoryById( )方法代码展示如下：

```
searchHistoryById(){
        // 将票据编号绑定到表单中
        this.sendform.BillInfoID=this.BillInfoID
        this.sendform.BillInfoMoney=""
        this.sendform.BillInfoType=""
        this.sendform.BillInfoIssueDate=""
        this.sendform.BillInfoDueDate=""
        this.sendform.PubBillID=""
        this.sendform.PubBillName=""
        this.sendform.PayBillID=""
        this.sendform.PayBillName=""
        this.sendform.AcceptBillID=""
        this.sendform.AcceptBillName=""
        this.sendform.HoldBillID=""
        this.sendform.HoldBillName=""
    // 向后端发送请求，调用 queryHistoryById 方法查询票据历史信息
    this.axios.post('http://115.28.136.131:8000/B1/bank/queryHistoryById',this.sendform)
        .then(function (res) {
        // 用 bills 表单查询票据历史
        this.bills = JSON.parse(res.data);
    }.bind(this))
        .catch(function (err) {
    if (err.response){
        //失败
        console.log(err.response)
    }
    }.bind(this))},
```

后端接收到 Post 请求，调用 queryHistoryById( )方法，在其中接收票据编号。使用 SDK 调用智能合约方法 queryHistoryById( )，根据票据编号查询历史信息，在将历史信息返回前端进行显示。queryHistoryById( )方法代码如下

```
// -查询历史交易记录
func queryHistoryById(ctx *gin.Context){
        // 接收票据编号信息
    var getbill Bill
```

```
    err := ctx.ShouldBind(&getbill)
    if err != nil {
        fmt.Printf("绑定失败: %s\n", err)
    } else {
        fmt.Printf("绑定成功: %s\n", err)
    }
    // 调用智能合约 queryHistoryById 方法，并传递票据编号以查询
    results, err := contract.SubmitTransaction("queryHistoryById",getbill.BillInfoID)
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
        ctx.JSON(http.StatusOK,string(results))
}
```

智能合约中，调用 QueryHistoryById()方法。其中需要使用提供的 GetHistoryForKey()方法，即可跟数据的 Key 值查询信息变更记录，由于存储的票据信息的 Key 即为票据编号，所以可直接查询。由于操作过程与上述功能的操作过程类似，在此不再展示代码。

查询票据历史功能到此实现完成。在此所有银行用户模块功能实现完成。

6.3.3 公司用户模块实现

公司用户模块实现票据系统中，用户身份所需的所以功能操作，包括查看公司用户作为票据承兑人、收款人和持票人关系的所有票据信息功能、发出票据背书申请功能、票据贴现申请功能、处理待承兑票据和处理待背书票据功能。该模块的各功能入口如图 6.7 所示：

图 6.7 公司用户功能入口

查看公司用户作为票据承兑人、收款人和持票人关系的所有票据信息功能。此功能主要为公司用户在票据系统中，根据其交易身份的不同，分别查询其对应关系的所有票据，此操作也便于对公司用户的操作功能进行分类，比如，公司用户可以在收款人关系票据中发出票据背书请求和贴现请求。由于上述三个功能的

界面和实现上大致相同，在此只以查看票据承兑人关系票据为例，进行详细展开说明。票据承兑人关系票据功能界面如下图 6.8 所示：

图 6.8 查看承兑人关系票据功能界面

查看承兑人关系票据功能前端实现如下。点击此功能入口，则自动在由此显示公司用户作为票据的承兑人关系的票据。此时，在前端编写 searchAllBills( )方法，在其中向后端发送 Post 请求，调用 checkAllPayBills( )方法，并传递此公司用户的 ID，从而根据此用户 ID 查询所有票据中，票据承兑人 ID 与用户 ID 相同其票据状态为"waitpay"的票据，将其返回前端进行显示。前端 searchAllBills( )方法代码如下：

```
searchAllBills(){
      // 获取公司用户 ID
      this.sendform.PayBillID = this.CompanyId
      // 向后端发送请求，调用 checkAllPayBills 方法，并传递公司用户 ID
this.axios.post('http://115.28.136.131:8000/C1/company/checkAllPayBills', this.sendform)
      .then(function (res) {
      // 查询成功，用 bills 表单接收结果
      this.bills = JSON.parse(res.data);
   }.bind(this))
   .catch(function (err) {
   if (err.response){
   //失败
      console.log(err.response)
   }
   }.bind(this))
},
```

后端接收 Post 请求，执行 checkWaitPayBills( )方法，在其中接收公司用户 ID，调用智能合约方法 queryWaitPayBills( ),并传递参数，需要查询 PayBillID 为公司用户和 State 为 made 的票据数据，返回前端进行显示。其代码如下：

```
// 查看待承兑票据 - 需要查询 PayBillID 为个人 和 State 为 Made 的数据
func checkWaitPayBills(ctx *gin.Context){
   //接收传来的公司 ID
   var getbill Bill
   err := ctx.ShouldBind(&getbill)
   if err != nil {
      fmt.Printf("绑定失败: %s\n", err)
   } else {
      fmt.Printf("绑定成功: %s\n", err)
   }
   // 调用智能合约 queryWaitPayBills，传递公司 ID 参数。
   results, err := contract.SubmitTransaction("queryWaitPayBills",getbill.PayBillID)
   if err != nil {
```

```
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
    fmt.Println(string(results))
    ctx.JSON(http.StatusOK,string(results))
}
```

智能合约中，调用 QueryAllPayBills( )方法。在此方法中，构造了查询语句，在查询语句中可根据数据属性来查询对应要求的票据，再调用 GetQueryResult( )方法，将查询语句传入，查询结果。将结果进行数据类型转换，传回后端。此方法的代码如下所示：

```
//条件查询\-查看 pay 身份的 bill 且 state 为 Public 的票据
func (s *SmartContract) QueryAllPayBills(ctx
contractapi.TransactionContextInterface, payBillID string) ([]Bill, error) {
// 拼接查询字符串，根据传入的 PayBillID 参数和票据状态查询符合要求的票据
 queryString := fmt.Sprintf("{\"selector\":{\"PayBillID\":\"%s\", \"State\":\"public\"}}",
payBillID)
// 将查询字符串传入查询方法
   resultsIterator, err := ctx.GetStub().GetQueryResult(queryString)
   if err != nil {
      return nil, err
   }
   defer resultsIterator.Close()
   // 用以接收查询结果
   var results []Bill
   // 遍历返回数据，将其加入 Bill 数组中。
   for resultsIterator.HasNext() {
       queryResponse, err := resultsIterator.Next()
       if err != nil {
          return nil, err
       }
       var bill Bill
       err = json.Unmarshal(queryResponse.Value, &bill)
       if err != nil {
           return nil, err
       }
       results = append(results, bill)
    }
return results, nil }
```

查看承兑人关系票据实现完成。

发出票据背书申请功能。此功能是公司用户根据业务需求，更改票据收款人和票据持票人的操作。公司用户根据自身需求，在查看自己作为票据收款人关系票据时，可进行背书申请操作。其操作界面如下图 6.9 和图 6.10 所示：

图 6.9 查看可背书票据功能界面

图 6.10 发出背书申请功能界面

发出背书申请功能的前端设计如下。由于背书操作是对票据的持票人或收款人才可发起的针对票据的收受方进行更改的操作，因此将此功能设置在公司用户查看自己作为票据收款人关系的所有票据信息的界面。如上图 6.9 中，点击请求背书按钮，即可弹出上图 6.10 的填写框。在填写框中，需要填写被背书人 ID 和名称，填写完成后，点击确定，后台会对填写内容进行判空和判错，所填写被背书人不可为此票据的承兑人和公司用户本人，否认会造成票据关系混乱。检查信息正确后，点击确定，进行提交。前端编写 confirmEndorse()方法，将被背书人 ID 和名称封装到 Bill 类型表单中，向后端发送请求。调用后端 endorseBill()方法，并传递参数。发送请求成功，显示“操作成功！”提示。confirmEndorse()方法的代码如下：

```
// 提交背书申请
confirmEndorse(){
   // 对填写被背书人信息进行判空操作
   if (this.sendform.EndorsedID == "" || this.sendform.EndorsedName == ""){
        alert("请填写完整信息！")
   }else{
// 对填写被背书人信息进行判错验证
   var isreap = 0
   for(let item of this.allSignInfos) {
// 判断是否已经存在
      if (item.CompanyId === this.sendform.EndorsedID){
            isreap = 1
     }
}
      if (isreap === 0){
            alert("此用户不存在！")
      }else if(isreap === 1 && this.sendform.EndorsedID === this.CompanyId){
            alert("被背书人不能为自己！")
      }else if(isreap === 1 && this.sendform.EndorsedID === this.sendform.PayBillID){
            alert("被背书人不能为承兑人！")
```

```
      }else{
        this.axios.post("http://115.28.136.131:8000/C1/company/endorseBill", this.sendform)
      .then(function (res) {
            alert("发布票据成功！ ")
            this.endorsedialogVisible = false;
          //成功
    }.bind(this))
    .catch(function (err) {
    if (err.response) {
       console.log(err.response)
    }
    }.bind(this))
  }
  }
},
```

后端接收 Post 请求，执行 endorseBill()方法，在其中接收被背书人 ID 和名称，调用智能合约方法 endorseBill(),并传递参数, 增加 EndorsedID、EndorsedName ，修改 State 为 EnWaitSign，完成背书申请操作，其代码如下：

```
// 票据背书 --增加 EndorsedID、EndorsedName ， 修改 State 为 EnWaitSign
func endorseBill(ctx *gin.Context){
// 获取前端传递的票据被背书人的 ID 和名称
  var bill Bill
  err := ctx.ShouldBind(&bill)
  if err != nil {
    fmt.Printf("绑定失败: %s\n", err)
  } else {
    fmt.Printf("绑定成功: %s\n", err)
  }
// 调用智能合约方法 endorseBill，并传递票据的被背书人信息
  results, err := contract.SubmitTransaction("endorseBill", bill.BillInfoID,
bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate,
bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID,
bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName, bill.EndorsedID,
bill.EndorsedName)
  if err != nil {
     fmt.Printf("Failed to submit transaction: %s\n", err)
     os.Exit(1)
  }
  fmt.Println(string(results))}
```

智能合约中，调用 endorseBill()方法。在此方法中，只需构造 Bill 表单，将票据被背书人信息和票据状态存入其中，即可将调用 PutState 方法将表单信息存入世界状态中。在此不展示具体代码。

发出背书申请功能实现完成。

发出贴现请求功能，此功能是公司用户，希望根据业务需求，快速结算票据的一种方式。公司可以选择对银行进行一定的补贴，从而将票据的收款人和持票人关系转移到银行用户，从而达成快速兑现票据目的的一种方式。发出贴现请求功能界面如下图 6.11 所示：

图 6.11 发出票据贴现请求功能界面

发出贴现申请功能的前端设计如下。由于背书操作是对票据的持票人或收款人才可发起的针对票据的收受方进行更改的操作，因此将此功能设置在公司用户查看自己作为票据收款人关系的所有票据信息的界面。如上图 6.9 中，点击请求贴现按钮，后台会自动发送贴现请求，即将票据状态变为 dcwaitsigned。前端编写 discountBills( )方法，向后端发送请求。调用后端 discountBill( )方法。发送请求成功，显示“操作成功！”提示。discountBills( )方法的代码如下：

```
// 请求贴现操作
discountBills(index, row) {
  // 将票据信息封装到 sendform 表单中
  this.sendform.BillInfoID=row.BillInfoID
  this.sendform.BillInfoMoney=row.BillInfoMoney
  this.sendform.BillInfoType=row.BillInfoType
  this.sendform.BillInfoIssueDate=row.BillInfoIssueDate
  this.sendform.BillInfoDueDate=row.BillInfoDueDate
  this.sendform.PubBillID=row.PubBillID
  this.sendform.PubBillName=row.PubBillName
  this.sendform.PayBillID=row.PayBillID
  this.sendform.PayBillName=row.PayBillName
  this.sendform.AcceptBillID=row.AcceptBillID
  this.sendform.AcceptBillName=row.AcceptBillName
  this.sendform.HoldBillID=row.HoldBillID
  this.sendform.HoldBillName=row.HoldBillName
  // 向后端发送请求，调用 discountBill 方法，并传递点击此行的票据信息
  this.axios.post("http://115.28.136.131:8000/C1/company/discountBill",this.sendform)
  .then(function (res) {
      alert("操作成功！ ")
  //成功
  }.bind(this))
  .catch(function (err) {
    if (err.response) {
      console.log(err.response)
    }
  }.bind(this))
},
```

后端接收 Post 请求，执行 discountBill( )方法，在其中接收所操作票据的信息，调用智能合约方法 discountBill( ),并传递参数，将 State 改为 DcWaitSigned，完成贴现申请操作，其代码如下：

```
// 贴现操作 - 将 State 改为 DcWaitSigned
func discountBill(ctx *gin.Context){
  // 绑定前端传来的所操作票据的信息
  var bill Bill
  err := ctx.ShouldBind(&bill)
  if err != nil {
    fmt.Printf("绑定失败: %s\n", err)
  } else {
```

```
        fmt.Printf("绑定成功: %s\n", err)
    }
    // 调用智能合约中的 discountBill 方法，修改票据状态
    results, err := contract.SubmitTransaction("discountBill", bill.BillInfoID,
bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate,
bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID,
bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName)
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
}
```

智能合约中，调用 discountBill()方法。在此方法中，只需构造 Bill 表单，将新票据状态存入其中，即可将调用 PutState 方法将表单信息存入世界状态中。在此不展示具体代码。

发出票据贴现请求功能实现完成。

处理待承兑票据功能。银行发布票据后，票据状态为 made 状态，需要承兑人接受承兑请求，票据状态更改为 public 后，票据才算正式生效，需要承兑人承认票据的债务关系。处理待承兑票据功能界面如下图 6.12 所示：

图 6.12 处理待承兑票据功能界面

处理待承兑票据功能的前端设计如下。用户用户点击待承兑票据按钮后，即可显示全部待承兑票据。此时调用前端的 searchAllBills( )方法，在此方法中，将公司用户 ID 传入表单中，与请求一同发回后端界面，调用后端函数。查询公司用户 ID 和票据承兑人信息相同且票据状态为 waitpay 的票据，将其查询并返回前端界面显示。公司用户可以查看票据的详细信息，待与公司业务核对完成后，可选择同意或拒绝承兑操作。同意操作由 agreePayBills( )方法完成。即将对应票据的票据状态变为 public 状态，将票据消息改为 waitpaysuccess。也可选择拒绝承兑，由 disagreePayBills( )方法完成。当票据被拒绝承兑后，说明票据发布失败，会将票据状态改为 billfail，将票据消息改为 waitpayfail，保留票据编号，将其他票据信息全部抹除。在此展示 searchAllBills( )方法和 agreePayBills( )方法，由于拒绝承兑方法和同意承兑方法实现相似，在此不作展示。两个方法的代码操作如下：

```
// 查询待承兑票据
searchAllBills(){
    // 将此公司用户的 ID 传回后端，查询其对应属性的票据信息
    this.sendform.PayBillID = this.CompanyId
    this.axios.post('http://115.28.136.131:8000/C1/company/checkWaitPayBills',
```

```
this.sendform)
    .then(function (res) {
    // 用 Bills 表单接收查询到的票据信息
        this.bills = JSON.parse(res.data);
    }.bind(this))
    .catch(function (err) {
    if (err.response){
        //失败
        console.log(err.response)
    }
  }.bind(this))
},
// 同意承兑操作
agreePayBills(index, row){
        // 将此操作行对应票据信息存入 sendform 表单，传到后端
        this.sendform.BillInfoID=row.BillInfoID
        this.sendform.BillInfoMoney=row.BillInfoMoney
        this.sendform.BillInfoType=row.BillInfoType
        this.sendform.BillInfoIssueDate=row.BillInfoIssueDate
        this.sendform.BillInfoDueDate=row.BillInfoDueDate
        this.sendform.PubBillID=row.PubBillID
        this.sendform.PubBillName=row.PubBillName
        this.sendform.PayBillID=row.PayBillID
        this.sendform.PayBillName=row.PayBillName
        this.sendform.AcceptBillID=row.AcceptBillID
        this.sendform.AcceptBillName=row.AcceptBillName
        this.sendform.HoldBillID=row.HoldBillID
        this.sendform.HoldBillName=row.HoldBillName
        this.axios.post("http://115.28.136.131:8000/C1/company/agreePay",this.sendform)
        .then(function (res) {
            alert("操作成功！ ")
        }.bind(this))
        .catch(function (err) {
        if (err.response) {
            console.log(err.response)
        }
    }.bind(this))
},
```

后端接收三个 Post 请求，分别执行 checkWaitPayBills( )方法、agreePay( )方法和 disagreePay( )方法。在 checkWaitPayBills( )方法中接收前端传来的公司用户 ID，调用智能合约方法 QueryWaitPayBills( )方法,并传递参数，查询公司用户 ID 和票据承兑人信息相同且票据状态为 waitpay 的票据，将其传回前端进行显示。agreePay( )方法中调用智能合约方法 agreePayBill( )，将对应操作票据的 State 改为 public，将票据消息改为 waitpaybillsuccess，完成同意承兑操作，disagreePay( )方法中调用智能合约方法 disagreePayBill( )，将对应操作票据的 State 改为 billfail，将票据消息改为 waitpaybillfail，保留票据编号，将其他票据信息设置为空，作为废票。完成拒绝承兑操作，在此只展示 QueryWaitPayBills( )方法和 agreePay( )方法代码，其代码如下：

```
// 查看待承兑票据 - 需要查询 PayBillID 为个人 和 State 为 Made 的数据
func checkWaitPayBills(ctx *gin.Context){
    //接收传来的公司 ID
    var getbill Bill
```

```
    err := ctx.ShouldBind(&getbill)
    if err != nil {
        fmt.Printf("绑定失败: %s\n", err)
    } else {
        fmt.Printf("绑定成功: %s\n", err)
    }
    // 调用智能合约 queryWaitPayBills，传递公司 ID 参数。
    results, err := contract.SubmitTransaction("queryWaitPayBills",getbill.PayBillID)
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
        ctx.JSON(http.StatusOK,string(results))
}
// 同意承兑 - 将 State 变为 public
func agreePay(ctx *gin.Context){
        // 绑定传来的 form
        var bill Bill
        err := ctx.ShouldBind(&bill)
        if err != nil {
            fmt.Printf("绑定失败: %s\n", err)
        } else {
            fmt.Printf("绑定成功: %s\n", err)
        }
        // 调用智能合约方法 agreePayBill，对票据的状态进行修改
        results, err := contract.SubmitTransaction("agreePayBill", bill.BillInfoID,
bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate,
bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID,
bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName)
        if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
        }
        fmt.Println(string(results))
}
```

智能合约中，分别调用 QueryWaitPayBills()方法、agreePayBill()和 disagreePay()方法，其中在 QueryWaitPayBills()方法中，由于根据收款人 ID 和票据状态查询符合条件票据，因此需要构造查询字符串，再调用智能合约中提供的接口中的方法查询信息。agreePayBill()和 disagreePay()方法只是对票据的特定属性进行修改。由于这三个函数的类似代码操作已在上述银行处理待贴现票据功能说明中展示，在此不再进行展示。

处理待承兑票据功能实现完成。

待背书票据处理功能。在此功能中，公司用户可以查询其他公司向本公司发起背书操作的票据。在查看详细票据信息，检查票据业务后，可以决定是否接受背书申请。处理待背书票据功能界面如下图 6.13 所示：

图 6.13 处理待背书票据功能界面

处理待背书票据功能的前端设计如下。用户用户点击待背书票据按钮后，即可显示全部待背书票据。此时调用前端的 searchAllBills( )方法，在此方法中，将公司用户 ID 传入表单中，与请求一同发回后端界面，调用后端函数。查询公司用户 ID 和票据被背书人信息相同且票据状态为 enwaitsign 的票据，将其查询并返回前端界面显示。公司用户可以查看票据的详细信息，待与公司业务核对完成后，可选择同意或拒绝背书操作。同意操作由 agreeEndorseBills( )方法完成。即将对应票据的票据状态变为 public 状态，将票据消息改为 endorsesuccess。也可选择拒绝背书，由 disagreeEndorseBills( )方法完成。当票据被拒绝背书后，会将票据状态改为 public，将票据消息改为 endorsefail。在此展示 searchAllBills( )方法和 agreeEndorseBills( )方法，由于拒绝背书方法和同意背书方法实现相似，在此不作展示。两个方法的代码操作如下：

```
// 查询待背书票据
searchAllBills(){
    // 将公司用户的 ID 传回后端，查询票据被背书人 ID 与此相同的票据
    this.sendform.EndorsedID = this.CompanyId
    this.axios.post('http://115.28.136.131:8000/C1/company/checkWaitEndorseBills',
this.sendform)
        .then(function (res) {
        // 用 bills 表单接收信息
            this.bills = JSON.parse(res.data);
        }.bind(this))
        .catch(function (err) {
    if (err.response){
        //失败
        console.log(err.response)
    }
    }.bind(this))
},
// 同意背书操作
agreeEndorseBills(index, row){
    // 将此操作行数据绑定到 sendform 表单中
    this.sendform.BillInfoID=row.BillInfoID
    this.sendform.BillInfoMoney=row.BillInfoMoney
    this.sendform.BillInfoType=row.BillInfoType
    this.sendform.BillInfoIssueDate=row.BillInfoIssueDate
    this.sendform.BillInfoDueDate=row.BillInfoDueDate
    this.sendform.PubBillID=row.PubBillID
    this.sendform.PubBillName=row.PubBillName
    this.sendform.PayBillID=row.PayBillID
    this.sendform.PayBillName=row.PayBillName
    this.sendform.AcceptBillID=row.AcceptBillID
    this.sendform.AcceptBillName=row.AcceptBillName
    this.sendform.HoldBillID=row.HoldBillID
```

```
      this.sendform.HoldBillName=row.HoldBillName
      this.sendform.EndorsedID=row.EndorsedID
      this.sendform.EndorsedName=row.EndorsedName
      // 向后端发送请求，调用 agreeEndorseBill 方法，  并传递表单
      this.axios.post("http://115.28.136.131:8000/C1/company/agreeEndorseBill",this.sendfor
m)
      .then(function (res) {
         alert("操作成功！ ")
      //成功
      }.bind(this))
      .catch(function (err) {
      if (err.response) {
         console.log(err.response)
      }
   }.bind(this))
},
```

后端接收三个 Post 请求，分别执行 checkWaitEndorseBills( )方法、agreeEndorseBill( )方法和 disagreeEndorseBill( )方法。在 checkWaitEndorseBills( )方法中接收前端传来的公司用户 ID，调用智能合约方法 QueryWaitEndorseBills( )方法,并传递参数，查询公司用户 ID 和票据被背书人信息相同且票据状态为 enwaitsign 的票据，将其传回前端进行显示。agreeEndorseBill( )方法中调用智能合约方法 agreeEndorseBill( )，将对应操作票据的 State 改为 public，将票据消息改为 endorsesuccess，完成同意背书操作，disagreeEndorseBill( )方法中调用智能合约方法 disagreeEndorseBill( )，将对应操作票据的 State 改为 public，将票据消息改为 endorsefail，保留票据编号，将其他票据信息设置为空，作为废票。完成拒绝背书操作，在此只展示 QueryWaitEndorseBills( )方法和 agreeEndroseBill( )方法代码，其代码如下：

```
// 待背书票据 -- 依据 EndorsedID 和 State 为 EnWaitSign 查询
func checkWaitEndorseBills(ctx *gin.Context){
      // 绑定传来的 form
      var getbill Bill
      err := ctx.ShouldBind(&getbill)
      if err != nil {
         fmt.Printf("绑定失败: %s\n", err)
      } else {
         fmt.Printf("绑定成功: %s\n", err)
      }
// 调用方法 queryWaitEndorseBills，查找票据被背书人 ID 与公司用户 ID 相同的票据
      results, err := contract.SubmitTransaction("queryWaitEndorseBills",getbill.EndorsedID)
      if err != nil {
         fmt.Printf("Failed to submit transaction: %s\n", err)
         os.Exit(1)
      } fmt.Println(string(results))
         ctx.JSON(http.StatusOK,string(results))
}
// 同意票据背书 -- 更换承兑人
func agreeEndorseBill(ctx *gin.Context){
      // 绑定传来的 form
      var bill Bill
      err := ctx.ShouldBind(&bill)
      if err != nil {
```

```
        fmt.Printf("绑定失败: %s\n", err)
    } else {
        fmt.Printf("绑定成功: %s\n", err)
    }// 进行交易
    results, err := contract.SubmitTransaction("agreeEndorseBill", bill.BillInfoID, bill.BillInfoMoney, bill.BillInfoType, bill.BillInfoIssueDate, bill.BillInfoDueDate, bill.PubBillID, bill.PubBillName, bill.PayBillID, bill.PayBillName, bill.AcceptBillID, bill.AcceptBillName, bill.HoldBillID, bill.HoldBillName, bill.EndorsedID, bill.EndorsedName)
    if err != nil {
        fmt.Printf("Failed to submit transaction: %s\n", err)
        os.Exit(1)
    }
        fmt.Println(string(results))
}
```

在智能合约中，分别调用 QueryWaitEndorseBills( )方法 agreeEndorseBill( )和 disagreeEndorseBill( )方法，其中在 QueryWaitEndorseBills( )方法中，由于根据被背书人 ID 和票据状态查询符合条件票据，因此需要构造查询字符串，再调用智能合约中提供的接口中的方法查询信息。agreeEndorseBill( )和 disagreeEndorseBill( )方法只是对票据的特定属性进行修改。由于这三个函数的类似代码操作已在上述银行处理待贴现票据功能说明中展示，在此不再进行展示。

处理待背书票据功能实现完成。至此，公司用户的所有功能实现完成。

6.4 本章小结

本章在代码层面，对票据操作系统中的登陆、银行用户和公司用户三个模块的各自功能进行了详细介绍，并进行了代码分析，完成了此次票据操作系统开发的全部流程。

第 7 章 总结与展望

7.1 总结

本文是对区块链技术的应用开发，根据现实中的电子票据系统，使用区块链技术开发票据操作系统。首先，我们对票据知识进行了简单学习，知道了有关票据的一些基础知识和基础操作，进而对生活中票据系统的应用情况进行简单了解。然后，开始学习区块链技术的相关知识，将其与票据系统的应用相结合。挑选了票据承兑、背书和贴现等三个操作作为票据系统的基本操作，设置票据系统中分为银行和公司两种用户，两种用户通过上述三个基本操作，对票据进行交互和处理，从而展现区块链技术在票据系统中的应用。由这样的系统设计，进而设计系统开发的细节，包括搭建区块链底层网络中设置共识机制，节点数和组织数。对票据的模型设计、对银行和公司两种用户具体的操作功能设计和对界面 UI 的设计。最终，编写代码实现票据操作系统。

在本系统中，使用区块链功能，解决了电子票据系统中的很多问题，本系统中的具体解决内容如下：

（1）将票据信息存于区块链中，用户之间共享账本，票据系统用户在不依赖第三方平台的基础上，验证票据的真实性。

（2）在本系统中，基于区块链技术，将信息存储于分布式账本中，可以确保票据信息的同步，避免违规交易。

（3）在此系统中可以实现复杂的业务逻辑。用户发出票据请求后，票据系统根据智能合约中的方法执行请求，完成交易。

（4）提高票据处理效率。在票据操作完成后，分布式账本中进行信息同步，自动完成对账。每次进行查询或者修改操作前，由于是对本地的账本信息进行检索，可以提高检索速度，提升票据处理的效率。

（5）区块链技术可以保证票据系统的数据安全和网络安全。区块链技术具有不可篡改的特点，票据数据无法篡改。区块链中数字签名和安全传输的过程，解决了交易请求被攻击或伪造的问题。确保了票据的数据和网络安全。

本系统在技术上实现了如下功能：

（1）搭建区块链网络。搭建了多个组织的区块链网络，将用户操作由组织划分。并且，搭建的区块链网络能满足处理一定信息量的系统的要求。

（2）本系统采用了 B/S 架构进行开发，将本系统部署在阿里云服务器上，用户只需要打开浏览器，输入应用的网址，即可使用此系统，增加了用户对系统的易用性。

（3）本系统采用前后端分离技术进行开发。后端采用 Go 语言和 Gin 框架进行开发，前端采用 Vue 框架和 ElementUI 界面库进行开发，从而降低了系统的耦合性，便于系统的维护。

（4）本系统编写了智能合约以操作区块链中的数据。在后端，调用 Fabric-SDK-Go 方法库创建操作智能合约的 SDK，使用 SDK 去调用智能合约中的方法，实现对票据信息的操作。

（5）系统中使用 CouchDB 数据库存储区块链账本中数据的世界状态，

本系统在业务逻辑上实现了如下功能：

（1）查询票据功能。在此票据系统中，银行用户可以查询所有票据的功能和票据的历史交易记录。公司用户可以根据自己与票据的关系，查询对应关系的相关票据信息。

（2）承兑功能。由银行用户发布票据后，公司用户需要对自己作为票据承兑人关系的待承兑票据进行承兑操作。从而完成对票据付款方关系的承诺和兑现。

（3）背书功能。可由票据收款人发起，向非票据承兑人和自己发出背书申请。从而实现将票据收款人和票据持票人的票据关系的转移。被背书人可处理待背书票据，可选择接受或拒绝背书。

（4）贴现功能。可由票据收款人发起，向银行发出贴现申请。从而实现将票据收款人和票据持票人的票据关系的转移。银行可以处理待贴现票据，选择接受或拒绝贴现。

（5）系统功能上，通过在票据模型中设置票据状态一属性，从而实现了使用一个数据模型即可实现承兑、背书和贴现三种操作，简化了系统的实现。

7.2 展望

通过两个多月的学习与开发，虽然本系统已经开发成功，但远未达实际应用的标准，如下几个方面，有待改进：

（1）功能缺少：由于本系统只是以承兑、背书、贴现等功能对区块链技术的应用进行简单展示，并没有添加太多的票据操作。在实际应用中，银行会有更多票据的基础操作，从而实现对票据的各种业务需求。

（2）性能较低：本系统中只是设计了四个基础用户，但实际的票据系统会包含很多的基本用户，需要满足其并发操作的需求。因此，需要更改区块链网络的共识机制和后端数据处理方法，从而提高系统的吞吐量，提高系统的稳定性。

（3）界面简陋：本系统的界面只是对功能的简单演示，与实际的系统操作界面有很大差距，与用户交互不友好。

（4）缺乏数据验证：在实际的票据系统中，对数据的安全性要求极高，此系统中对输入信息缺乏足够的验证操作，容易因用户输入错误，造成系统混乱。

（5）数据模型简单：实际应用的票据系统中，票据具有更多的属性，此票据系统的属性只是满足演示功能的实现。但实际银行中，还要包含对票据的金融方面的复杂处理，本系统无法满足其要求。

区块链是当今的新兴技术，通过十几年的发展，区块链技术已经比较成熟。特别是通过此次开发经历，深刻认识到了区块链技术的优势和其在各领域进行应用开发的巨大潜力。随着区块链技术的不断迭代和人们对区块链技术认识的不断加深，区块链一定会成为我们生活的重要部分，为我们的生活带来便利。

致谢

在毕业设计即将完成之际，心中百感交集。四年的时光，如白驹过隙，但也留下了很多珍贵的回忆。

首先，我要感谢我的母校，为我这四年的生活提供了一个良好的生活和学习环境。在学校这个载体中，我认识了很多有趣的人，经历了不少有趣的事，也学习了很多实用的知识，感受了一个丰富的大学生活该有的样子。

其次，我要感谢我的老师。首先要感谢我毕业设计的指导老师周炜老师，正是这样一个机会,让我可以接触到区块链的相关知识,并学习一些其它的新知识。其次要感谢王成钢老师，在我参加大创时给予充分的引导与支持。还要感谢李传彬老师，在申请学校时给予的肯定与帮助。还要感谢四年之中，有幸结识的很多老师，在学习上给予的启迪。

这一年，求学路漫漫。非常感谢这一路上的许多老师的教导和鼓励，最终得以平稳上岸。

最后，要感谢我的家人。感谢他们始终在背后引导、支持与陪伴。

写于5月23日晚

参考文献

[1] 张增骏,董宁,朱轩彤.深度探索区块链.北京：机械工业出版社，2018.3

[2] 杨镇,姜信宝,朱智胜,盖方宇.深入以太坊智能合约开发.北京：机械工业出版社，2019.4

[3] 华为区块链技术开发团队.区块链技术及应用.北京：清华大学出版社，2019.3

[4] 罗金海.人人都懂区块链.北京：北京大学出版社，2018.7

[5] 鹏帅兴.区块链从入门到精通.中国青年出版社，2017.10

[6] 徐明星,田颖,李霁月.图说区块链.中信出版社，2017.7

[7] 韩锋.区块链:从数字货币到信用社会.中信出版社，2016.7

[8] 梁灏.Vue.js 实战.北京：清华大学出版社，2017.10

[9] 郑兆雄.Go Web 编程.人民邮电出版社，2017.12

[10] 柴树衫,曹春晖.Go 语言高级编程.人民邮电出版社，2019.7

[11] 刘汉伟.Vue.js 从入门到项目实践.北京：清华大学出版社，2019.3

[12] 刘博文.深入浅出 Vue.js.人民邮电出版社，2019.3

[13] 申思维.Vue.js 快速入门.北京：清华大学出版社，2018.12

[14] Daniel Drescher.Blockchain basics: a non-technical introduction in 25 steps.人民邮电出版社，2018.11

[15] Alan A.A.Donovan.The Go Programming Language.北京：机械工业出版社，2017.5