学校代码：10246
学　　号：17212040263

硕 士 学 位 论 文

（专业学位）

一种基于状态反馈负载均衡策略的区块链服务平台设计与实现

A Design and implementation of blockchain service platform based on state feedback load balancing strategy

院　　系：　计算机科学技术学院
专业学位类别(领域)：　计算机技术
姓　　名：　赵晓峰
指 导 教 师：　卢暾 教授
完 成 日 期：　　年 月 日

指导小组成员名单

顾　宁　　教　授
张　亮　　教　授
卢　暾　　教　授
丁向华　　副教授

目 录

摘 要

区块链技术近年来发展迅速，相关应用如雨后春笋般蓬勃发展。区块链应用生态圈逐渐形成，绝大多数的应用通过接入区块链公共服务机构如以太坊(Ethereum)、超级账本(HyperLedger)等加入区块链网络。应用的接入依赖于单一的节点导致了整个应用的流量压力全部集中到一个 Geth 客户端上，使得基于以太坊的区块链应用的并发性能较低。同时，由于区块链技术多应用于数字信息交易领域，对数据安全要求较高，因此单 Geth 客户端的架构在容灾和可靠性方面亦存在不足。

针对上述问题，结合区块链与相关业务场景的特点，设计和实现高性能以太坊服务系统极为必要。因此，本文结合微服务思想设计了以太坊后台服务架构，基于 Docker 构建了多 Geth 客户端的以太坊服务平台，基于 Geth 客户端在 Docker 容器中的运行状态设计了反馈负载均衡策略，并最终实现了面向私募股权交易的区块链应用平台。本文主要工作如下：

1. 提出一种基于微服务思想的面向私募股权的区块链服务平台设计方案。根据业务特点和技术特点设计灵活的应用架构，可以在资源耗费最小的情况下满足业务要求，同时在需要的时刻快速进行系统延展。

2. 基于以太坊自身特点提出了一种状态反馈负载均衡策略，该策略考虑区块链节点运行状态建立状态空间，对状态空间中的节点进行负载分配实现系统的可靠性和高性能。并将该负载均衡策略应用于基于容器化技术思想构建了一种灵活的底层区块链服务平台。该平台将以太坊客户端运行于容器中作为单一节点，结合负载均衡策略实现可靠性和高性能。

3. 实现了面向私募股权的区块链服务系统，包括账户管理、转账管理和合约发布等业务功能，以及状态监控、网关管理等后台模块功能。结合区块链的智能合约等技术特点实现了私募股权交易自动化结算与科学管理功能。

关键字：以太坊、高并发、微服务、负载均衡、私募股权
中图分类号：TP3

Abstract

As Blockchain has developed rapidly in recent years, related applications have mushroomed as well. The blockchain application ecosystem is gradually forming, and most applications join the Blockchain network by accessing Blockchain public service agencies such as Ethereum and HyperLedger. The access of these applications relies on a single client node, which causes the traffic pressure of the entire application to be concentrated on a Geth client, making the Ethereum-based blockchain application's concurrent performance poor. At the same time, because blockchain technology is mostly used in the field of digital information transactions, and requires high data security, the architecture of a single Geth client also has shortcomings in disaster tolerance and reliability.

In view of the above problems, it is extremely necessary to design and implement a high-performance Ethereum service system based on the characteristics of the blockchain and related business scenarios. Therefore, this research designs the Ethereum back-end service architecture based on the idea of the microservices, builds an Ethereum service platform with multiple Geth clients based on Docker, designs a feedback load balancing strategy based on the running state of the Geth client in the Docker container, and finally implements A blockchain application platform for private equity transactions. The main work of this research is as follows:

1. Propose a design scheme of a private equity-oriented blockchain service platform based on the idea of microservices. Design a flexible application architecture based on business features and technical traits, which can meet business requirements with minimal resource consumption, and quickly expand the system at the moment of need.

2. Based on the characteristics of Ethereum, a state feedback load balancing strategy is proposed. This strategy considers the operating status of the blockchain nodes to establish a state space, and performs load distribution on nodes in the state space to achieve system reliability and high performance. This load balancing strategy is applied to build a flexible underlying blockchain service platform based on the idea of containerization technology. The platform will run the Ethereum client in the container as a single node, and combine the load balancing strategy to achieve reliability and high performance.

3. Implemented a private equity-oriented blockchain service system, including business functions such as account management, transfer management, and contract

issuance, as well as background module functions such as status monitoring and gateway management. Combined with the technical characteristics of the smart contracts of blockchain, it has realized the functions of automatic settlement and scientific management of private equity transactions.

Keywords: Ethereum, high concurrency, micro-services, load balancing, private equity
Chinese Library Classification: TP3

第一章 绪论

1.1 研究背景

私募股权(Private Equity) 由美国投资家本杰明格雷厄姆创立，是一种非上市企业或单位进行的非公开项目招募活动[1]，此类招募活动不限于资金募集、项目招标等。私募股权源于金融领域，首先被用于面向特定对象释放企业私有股权以达到募集资金的作用[3]。私募股权市场广大，据二八定律知 20%的人持有 80%的社会财富，而 20%的人持有的财富中，有 80%属于私募产品[4]。随着经济与时代的发展，私募股权的范围越来越广，私募股权所募集的目标不再局限于资金，劳动力、资源、技术等等均成为募集对象[4]。在私募股权活动中，一般涉及需求发布方与需求投标方。需求发布方公开部分信息用于招募需求投标方，投标方团队通过发布方公布的有限项目信息评估判断是否进行投标以及后续合作。在发布方与投标方的后续合作中，双方就合作项目详细信息进行相互之间的沟通[5]。私募股权活动针对的是特定的对象，且需求发布方需要在保证项目隐私性的情况下公布部分关键信息以招募需求投标方，因此构建私募股权交易平台以保证合作方的信息交流以及招募方发布的信息隐私性变得尤为重要[2]。同时，为保证交易平台权益，需要考虑投资方与项目实施方在交易过程中不能越过平台实施合作。

私募股权平台是面向私募股权活动的中间交易平台，用户可以发布需求信息或者对特定需求进行投标。该平台极大得降低了私募股权活动中合作方的交易难度，同时由于平台的监管提高了合作方之间的信任度[6]。私募股权平台的构建需要考虑用户隐私、用户发布的项目的信息安全性、用户之间股权活动交易等方面问题[7]。同时，由于私募股权项目需求等信息的价值很高，私募股权平台需要考虑自身的权益问题，即不能使用户可以在利用平台提供的信息后越过平台直接进行合作。基于以上问题考虑，采用区块链技术作为底层技术。

区块链(BlockChain)技术是一种集合加密、分布式存储、共识机制和智能合约等技术的去中心化分布式账本技术。它是由中本聪于 2008 年提出的革命性技术，基于其去中心化特点构建了一套名为“比特币”的数字交易体系[8]。近年来，包括比特币、以太币等在内电子货币得到了巨大的发展[9]。区块链技术的应用不仅仅局限于币圈，自 20 年以太坊诞生，大量区块链应用在金融、版权保护、电子货币等领域应用生态逐渐建立，如今，多家相关应用和企业蓬勃发展[9]。其特有的去中心化特性极大的改变了原有的第三方信任体系，使得人们的信任机制得到了颠覆性的改变。区块链的优势了去中心化和可信任，交易双方在没有第三方的情况下进行交易，且交易信息可以永久保存[8]。当前的区块链技术应用

主要基于第三方平台如以太坊、超级账本等构建，开发者构建应用后通过官方提供的客户端接入网络，并参与数据校验、数据打包等链上活动。

综合上述需求，本文提出了一个面向私募股权的区块链服务系统。该系统融合容器化技术、微服务思想以及云计算领域中的负载均衡模型，不仅极大的提高了区块链技术作为底层技术的稳定性、安全性和并发性能保障，同时使得区块链技术的应用更加便利和灵活。该服务系统中的业务设计是面向私募股权领域，基于区块链的智能合约设计代币作为平台通用的股权激励，将项目需求信息打包永久性存储到区块链中。同时，通过事件记录区块链网络中的交易过程。该系统的应用架构采用微服务应用架构，根据区块链读写数据特点构建相应的服务提供者。底层区块链节点与单个的服务均运行于容器中，提高系统的整体可靠性和系统的延展性。在底层区块链服务平台与后台服务之间，设计和应用基于状态反馈的负载均衡策略，提高系统的并发性能。

1.2 国内外研究现状

由于私募股权交易可理解为电子信息交易的一种，因此其涉及第三方信任机制[1]。即交易双方互不信任，但由于共同信任第三方，因此他们之间依然可以完成交易。因此，在私募股权交易中，构建第三方交易平台非常重要。然而，第三方信任机制过于依赖第三方，当对平台信任度不够的情况下，交易便无法顺利完成。区块链技术的去中心化机制解决了第三方信任机制问题，信任机制由简单的第三方证明转移到工作量证明 PoW(Proof of work)、股东权益证明 PoS(Proof of States)等[8]。等人在构建私募股权交易平台的过程中引入了区块链技术[3][4]。

目前，区块链技术在构建交易平台中的应用方法，主要采用接入区块链服务平台如以太坊(Ethereum)、超级账本(Hyperledger)等[10]。众多区块链应用(DApp, Distribution Application)依托于上述平台落地实施，形成了相应的区块链生态。以太坊是由 Vitalik Buteri 于 2013 年 12 月提出的可编程区块链，该平台除了可基于内置的以太币(ether)实现数字货币交易，还提供了图灵完备的编程语 Solidity 以便赐额智能合约，首次在区块链领域提出了智能合约的概念[11]。 超级账本是由 Linux 基金会于 2015 年 12 月发起的开源区块链项目，其目的在于发展跨行业的区块链平台[12]。区块链平台有公有链、私有链和联盟链三种，对于公有链，节点可自由上链和下链；对于私有链，节点必须在经过授权后才能上链;联盟链的节点一般对应实体机构，各机构组成利益联盟。Ethereum 设计之初面向公有链，但在 Ethereum2.0 中同时加入了私有链模式，可以通过简单的设置选择私有链和公有链。但以太坊依然是公有链中应用最广泛平台，Quorum、Monax、DFINITY 等应用或平台均基于以太坊构建。超级账本则在联盟链中应用广泛，

其成员已包括 IBM、Intel、J.P.Morgan、SWIFT 等 130 多名成员[10]。而以太坊的扩展性非常差，因此本文考虑引入微服务和容器等技术对以太坊的应用方式加以改进。

微服务技术自 2010 年开始兴起的分布式应用架构技术，该技术结合容器化技术在敏捷开发、自动化运维等领域应用广泛，其各个服务独立开发部署的架构优点不仅可以提高项目开发效率，同时可以提高系统的扩展性和运行稳定性[12][13]。微服务技术在应用开发领域取得了巨大的成功，J Lawson 等人提出了一种基于微服务的通信系统[14]；A Balalaie 的研究说明微服务技术可实现自动化运维且是一种原生的云服务架构[15]。而微服务技术往往与容器化技术结合使用，J Stubbs 等人提出了一种使用 Docker 容器的分布式系统架构[16],M Amaral 评估了结合容器技术的微服务架构的表现，认为微服务结合容器技术能发挥其最大的性能优势[17]。容器化技术是一种虚拟服务器技术，其特点是轻量-仅拥有系统运行最基本的组件、安全-容器具有沙盒机制等特点[18],使用最广泛的容器是 Docker[19]。Docker 是微服务中各个组件运行的基础环境，而微服务中各个组件之间的调用需要考虑云计算领域的负载均衡技术[20]。本文根据 Ethereum 的 Geth 客户端在 Docker 中运行的特点设计合理的负载均衡机制，使得系统的整体负载效果达到最大。

负载均衡发展历史悠久，有非常多经典的负载均衡算法如加权、轮询、最少连接等[21]。该类负载均衡最大的缺陷在于忽略了服务节点的特点以及当前状态。采用了将任务流进行分组，并且考虑对其进行优先级划分的方式得到了较高的负载效率[22]。负载均衡的目标在于使得系统稳定且资源消耗最少得情况下得到最少得任务完成时间[23]。等人提出的相空分析方法首先建立以服务器参数为坐标轴的相位空间，然后将服务节点的参数归一化投射到该相位空间中[24]。等人提出了基于相位空间的负载均衡方法，充分考虑了节点的状态进行负载和调度[25]。

综上所知，区块链技术中以太坊可定制智能合约，同时设置公链与私链，适合基于其建立企业化应用。但同时，以太坊通过官方客户端 Geth 向外提供接口的方式导致其在构建应用过程中部署麻烦且不易扩展。微服务是一种应用成功的高可用分布式架构，结合私募股权相关业务特点定制特定的以太坊微服务系统。该服务服务系统中对服务节点的调用采用负载均衡算法，使服务系统的负载效率达到最高。目前，对于云计算领域的负载均衡算法，文献[24]提出了节点相位空间法调度，该方法考虑服务节点的 CPU 占用率、内存占用率等因素进行负载，达到了非常好的调度效果，但是该方法未考虑服务节点的业务特性。本文将根据私募股权相关业务进行对服务节点进行设计，并根据以太坊在 Docker 中的运行特定设计负反馈负载均衡算法并应用到上述以太坊微服务系统中，最后将该服务系统应用到私募股权业务场景中检验其可用性以及稳定性、扩展性和并发性等性

能。

1.3 主要工作和创新点

本文主要工作是分析了构建私募股权交易平台面临的困难，提出了以区块链技术为底层核心技术的解决方案，结合私募股权领域的业务特点以及区块链技术的优点，实现了面向私募股权的区块链服务系统。在设计和实现系统的过程中，本文充分考虑区块链技术的特点和局限性，通过优化系统架构，以及设计适合本系统的基于状态的反馈负载均衡策略，提高系统的可靠性、安全性和延展性，并通过性能对比实验对系统进行了评估和验证。

结合私募股权相关业务特点与区块链技术自身特性，本文基于微服务思想对相关业务进行分类封装，并运行于容器中，同时设计高效的反馈负载均衡策略。最后本文实现了区块链服务系统，并通过实验验证了系统的可用性、可靠性等高性能。本文研究的主要工作有以下几点：

1. 提出一种基于微服务思想的面向私募股权的区块链服务平台设计方案。根据业务特点和技术特点设计灵活的应用架构，可以在资源耗费最小的情况下满足业务要求，同时在需要的时刻快速进行系统延展。

2. 基于以太坊(Ethereum)自身特点提出了一种状态反馈负载均衡策略，该策略考虑区块链节点运行状态建立状态空间，对状态空间中的节点进行负载分配实现系统的可靠性和高性能。并将该负载均衡策略应用于基于容器化技术思想构建了一种灵活的底层区块链服务平台。该平台将以太坊客户端运行于容器中作为单一节点，结合负载均衡策略实现可靠性和高性能。

3. 实现了面向私募股权的区块链服务系统，包括账户管理、转账管理和合约发布等业务功能，以及状态监控、网关管理等后台模块功能。结合区块链的智能合约等技术特点实现了私募股权交易自动化结算与科学管理功能。

1.4 篇章结构

本文将围绕面向私募股权的区块链服务系统设计与性能优化等方面的研究与实现安排六个章节展开介绍。

第一章，绪论。本章介绍本文的研究背景、研究目的与意义，简要介绍了私募股权相关背景和区块链技术应用现状，并简述了本文的主要工作与创新点。

第二章，相关工作与技术背景。本章主要介绍本文提出的区块链服务系统涉及的相关技术，从区块链技术、容器技术、负载均衡技术和以微服务架构为主的分布式应用技术四方面介绍。

第三章，基于微服务的区块链服务系统架构设计与负反馈负载均衡策略的设

计。本章详细描述了区块链服务系统的架构设计过程，包括底层的以太坊服务平台与微服务架构中各个模块的设计。本章还描述了负载均衡策略的的设计，包括任务流的设计，需要负载调度算法等。

第四章，基于状态反馈的负载均衡的以太坊服务系统的实现。本章描述了以太坊服务系统各个模块的具体实现，对于给出了关键代码及其说明。

第五章，面向私募股权的区块链服务系统的实现与性能评估，本章介绍了基于前两章内容的面向私募股权的区块链服务系统，并设计实验对系统进行了性能检测与分析。

第六章，总结与展望。本章总结本文总体的研究内容和方法与技术的局限性展望未来工作，并对下一步工作做出规划。

第二章 相关工作及技术背景

2.1 区块链技术

区块链技术是中本聪在 2008 年发布的比特币系统白皮书《比特币:一种点对点的数字货币系统》中提出的底层核心技术，是一种结合了分布式存储，一致性共识和密码学等多方面知识的应用技术[8]。区块链最大的特点在于实现了摆脱第三方信任机制的交易系统，在该系统中交易的确认由所有节点共同参与完成，数据采用分布式存储方式，存储到区块中[8]。区块与区块相互链接形成区块链。

2.1.1 区块链中的交易

区块链中的交易的生成是由用代币的拥有者基于私钥对该代币的上一次交易以及本次交易的接受方签署数字签名。该数字签名被附加在代币的末尾形成交易单，然后被广播到区块链网络中的其他节点[30]。在网络中的节点收到交易清单后会开始对该交易进行验证，以 PoW(Proof of Work)为例，节点进行开始被称为挖矿的复杂 hash 运算操作，首先完成交易验证工作的节点将所有的交易进行打包并广播给其他节点。当网络中的其他节点收到该区块后验证其时间戳以及交易是否有效，在确认签章有效以及没有重复花费后再将区块上链，此后数据无法更改[31]。综上所述，在区块链网络中，一个交易的完成过程包括交易的发出、数字签名的广播、共识算法验证交易的正确性、打包区块、验证区块链和上链六个步骤。区块的结构如图 2.1 所示，其中最重要的信息包括前一个区块链的哈希指针和本区块的哈希值、时间戳、完成本区块打包工作的节点(Miner)的账户等信息。图 2.2 为某区块的实际示例。

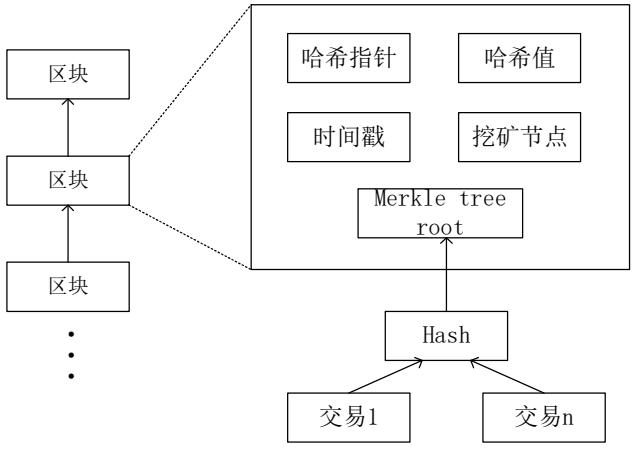

图 2.1 区块链中的区块结构

2.1.2 以太坊系统

以太坊系统是一种无需中央管理和协调机制的分布式系统，该系统可以运行智能合约和小型应用，其创造者的初衷是开发一套可以自治的“世界计算机”[15]。它在比特币的基础上更进一步，不仅可以在全网节点上验证和存储交易数据，还可以在全网所有节点中运行智能合约代码，节点使用 EVM(Ethereum Virtual Machine)运行智能合约[32]。以太坊同时具有分布式数据存储和计算的能力[33]。

以太坊的使用是通过以太坊客户端(Ethereum client, Geth)实现，用户通过以太坊客户端连接以太坊网络中的其他节点，并参与区块同步、挖矿、交易验证的工作[33]。以太坊上的所有节点地位相同，同时没有中心协同和管理节点。在成为以太坊节点后可以进行连接以太坊网络、查看以太坊区块链、发布交易和智能合约、运行智能合约以及挖矿等工作[33]。

智能合约是以太坊提供的可以运行在 EVM(Ethereum Virtual Machine)中的图灵完备语言。相对于比特币的原始脚本语言，它更高级，同时其图灵完备性意味着它可以被用来实现任何功能或者执行任和计算[34]。通常，智能合约在 Dapp(Distributed Application)中被用于定制化代币或交易等操作，如自动投票计数 Dapp[35]。

2.2 容器化技术

2.2.1 容器与虚拟机

虚拟机(Virtual Machine)和容器(Docker)均是目前云计算平台常用的虚拟化技术，虚拟机技术发展成熟，而容器技术做为新一代的虚拟化技术代表了未来虚拟技术的发展方向[36]。如图 2.3 所示位虚拟即与容器的架构图，相对于容器而言，每个虚拟机都有一个独立的 Kernel 中间件用于连接操作系统(OS)和硬件(Hardware)，该客户机操作系统会占用大量的硬件资源并安装大量依赖[37]。由于虚拟机包含一整套操作系统，因此占用大量空间，同时启动缓慢。而容器则是对应用层的抽象，多个容器共享操作系统内核 Kernel，并各自作为独立的进程运行于用户空间，因此占用空间更小，启动更快[38]。

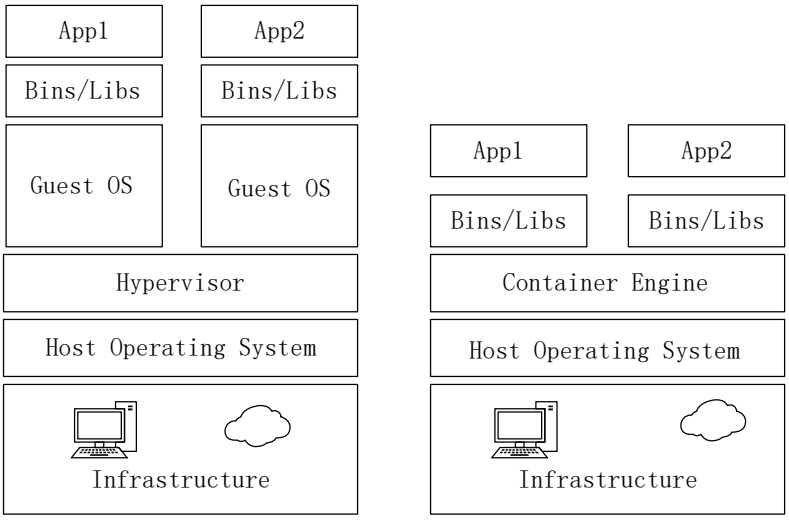

图 2.2 容器与虚拟机对比图

相对于虚拟机，使用容器可以降低硬件成本，一台虚拟机上可以运行成百上千个相互隔离的容器。其轻便性使得应用在开发和部署过程中更加快速和简便。容器技术在微服务技术中应用广泛，其分层存储以及镜像技术使得复用更加容易，更轻松的维护和扩展使其在微服务架构的发挥出最大的优势。

2.2.2 Docker

Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。如图 2.3 所示，Docker 引擎是一个包含以下主要组件的客户端服务器应用程序[38]。其中，Server Docker Daemon 是守护进程，该进程长期运行；REST API 是指定程序与守护进程通信的接口；CLI 是命令行工具，提供镜像、容器、网络和容器数据卷的管理功能。

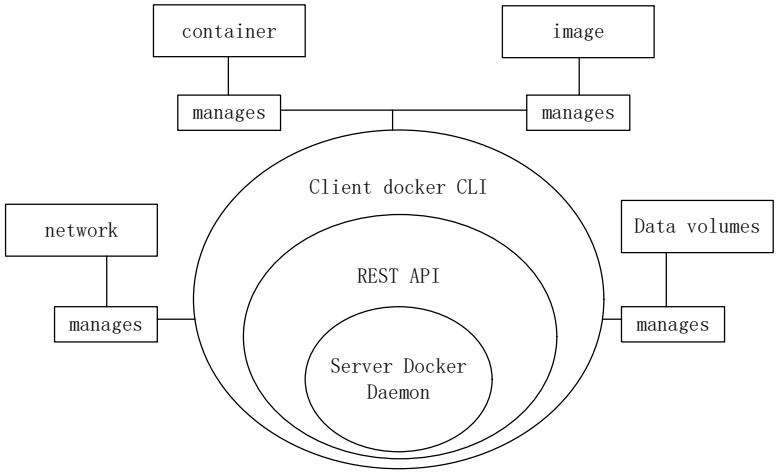

图 2.3 Docker 架构图

Docker 的运行示意图如图 2.4 所示[36]。Docker 使用客户端-服务器 (C/S) 架构模式，使用远程 API 来管理和创建 Docker 容器。Docker 容器通过 Docker 镜像来创建。容器与镜像的关系类似于面向对象编程中的对象与类。Registry 是镜像统一管理中心，客户端通过 docker pull 命令拉取镜像到 Docker 宿主机，并通过 docker run 命令基于镜像创建容器。同时，用户可根据需求通过 docker build 命令定制镜像并上传到镜像仓库 Registry 中[23]。

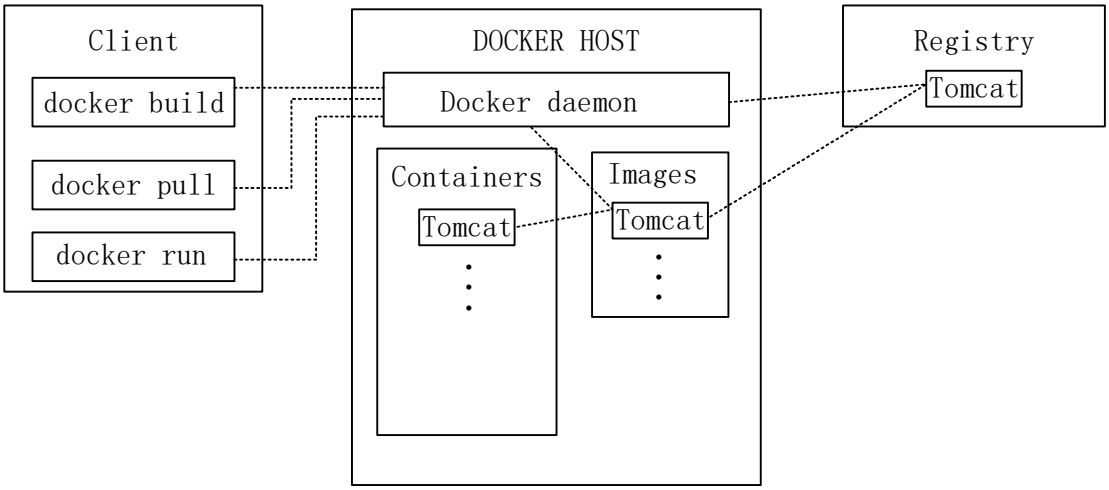

图 2.4 Docker 运行示意图

2.3 微服务架构

2.3.1 微服务应用架构

微服务架构是一种架构概念，旨在通过将功能分解到各个离散的服务中以实现对解决方案的解耦[17]。它的主要作用是将功能分解到离散的各个服务当中，从而降低系统的耦合性，并提供更加灵活的服务支持。围绕业务领域组件来创建微型应用，这些应用可独立地进行开发、管理和迭代[18]。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单[19]。

微服务架构与传统的单体应用架构相比开发效率更高，减少了开发中的相互等待和冲突；维护更简单，代码耦合度极大的降低使得微型应用更加简单；更加灵活，构建时间更短；稳定性增强，不会因为一个小问题导致整个系统崩溃；扩展性增强，可以根据需求，对特定的功能组件进行扩展以提高系统的并发性。但是因为微服务架构是分布式架构，因此存在分布式的管理和调用消耗。

根据微服务的官方定义，微服务是一系列独立的服务共同组成的系统[17]。每个独立的服务单独部署，并且相互隔离运行在自己的进程中。系统中每个服务为独立的业务开发，在运行中采用分布式管理。微服务系统强调强服务个体和弱通信，其高度容错性使得其在自动化运维和敏捷开发中得到了广泛应用[19]。

2.3.2 微服务解决方案

微服务架构的具体实践中需要解决四个问题，首先是客户端如何访问服务，对此一般采用在服务和 UI 之间设置名为 API Gateway 的代理机构，由该机构提供服务的统一入口，聚合后台服务，节省访问流量并提供过滤、流量控制等 API 管理功能；其次是服务间的通信方式，服务之间的调用分为同步调用和异步调用，同步调用一致性强，但性能较差，如 REST API 的方式基于 HTTP 实现，在调用的过程中会有大量的多余通信消耗。异步调用往往通过消息队列实现，此方式在分布式系统中应用广泛，它既能降低系统耦合，又能实现调用的缓冲，有效的防止了系统因流量过大而崩溃的情况，不过异步调用也会导致一致性的减弱。第三个问题是服务的实现，即服务如何管理，服务之间如何管理。一般有两类做法，基于客户端的服务注册与发现和基于服务端的服务注册与发现如图 2.5 所示。当服务上线时，服务提供者将自己的服务信息注册到 ZK（或类似框架），并通过心跳维持长链接，实时更新链接信息。服务调用者通过 ZK 寻址，根据可定制算法，找到一个服务，还可以将服务信息缓存在本地以提高性能。当服务下线时，ZK 会发通知给服务客户端。最后一个问题是当服务崩溃时如何处理，对于此问题一般有重试、限流、熔断、负载均衡等方式处理。其中设计合适的负载均衡算法不仅可以应对服务崩溃，可以提高系统的并发性能。

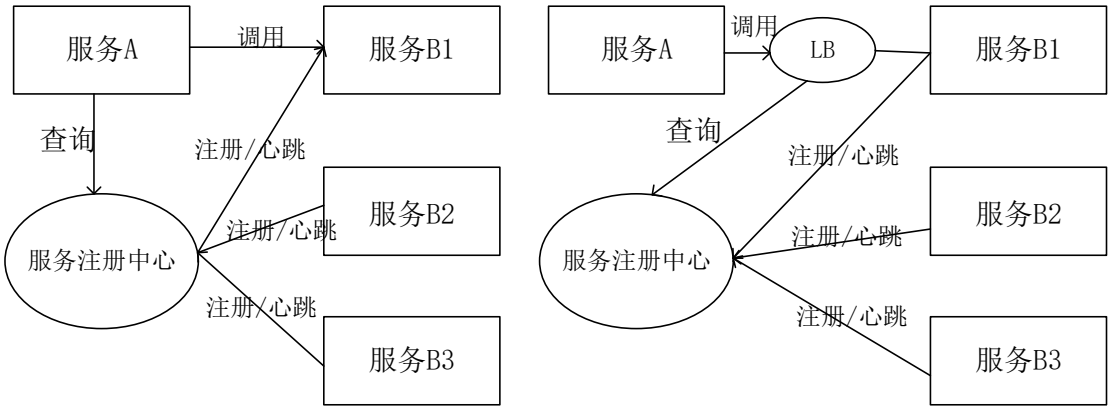

图 2.5 服务管理方式

微服务解决方案众多，其中由阿里巴巴开源的组件和云产品组成的 Spring Cloud Alibaba 尤其突出[39]。2018 年 10 月 31 日，Spring Cloud Alibaba 正式入驻了 Spring Cloud 官方孵化器，并在 Maven 中央库发布了第一个版本。Spring Cloud Alibaba 致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。该解决方案提供了一系列组件来解决以上提到的微服务应用遇到的四个问题[40]。

Sentinel：把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

Nacos：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

RocketMQ：一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。

Alibaba Cloud ACM：一款在分布式架构环境中对应用配置进行集中管理和推送的应用配置中心产品。

2.4 负载均衡技术

在微服务等分布式应用架构中，为提高系统的可靠性、容错性和并发性，往往存在多种相同的服务节点共同对外提供服务。任务在这些服务节点中的分配方式，即为负载均衡技术[24]。负载均衡策略包含任务负载任务的划分方式、任务的调度以及服务节点的架构等方面，其目的在于减少总的任务完成时间，提高服务资源的利用率以及保证良好的系统负载均衡度[25]。经典的负载均衡策略包括轮询、加权、最少连接等，此类负载均衡算法最大的问题在于未考虑到服务节点的状态[29]。在云计算领域，状态反馈负载均衡算法充分考虑服务节点的状态，根据节点状态以及任务类型进行负载分配，可以实现更好的负载效果[28]。

状态反馈负载均衡策略的设计包含四个方面，首先是负载均衡依据，其次是任务调度模型，第三是构建任务优先级模型，最后设计任务分配模型。

在负载均衡依据中，选取对服务节点状态影响较大的参数如内存占用率、CPU 占用率的归一化建立服务节点状态空间，然后将服务节点的状态向量投射到该状态空间中。若服务节点在状态空间中的位置距离原点较近，说明其资源消耗度较小；如果位置距离原点较远，则说明该服务节点的资源消耗度较高，处于资源占用状态。

2.5 小结

本章节首先介绍了区块链的交易过程和区块链结构，然后介绍了区块链应用平台 Ethereum。本章第二小结对比虚拟机技术介绍了容器化技术，并介绍了 Docker 的架构以及流程架构。本章第三小结对比单体应用架构介绍了微服务架构的实践以及解决方案 Spring Cloud Alibaba。在本章第四小结，介绍了基于反馈的负载均衡调度模型，使用公式和推理介绍了该类负载策略的工作原理和过程。

第三章 基于微服务的以太坊服务系统架构设计

3.1 区块链技术在私募股权场景应用分析

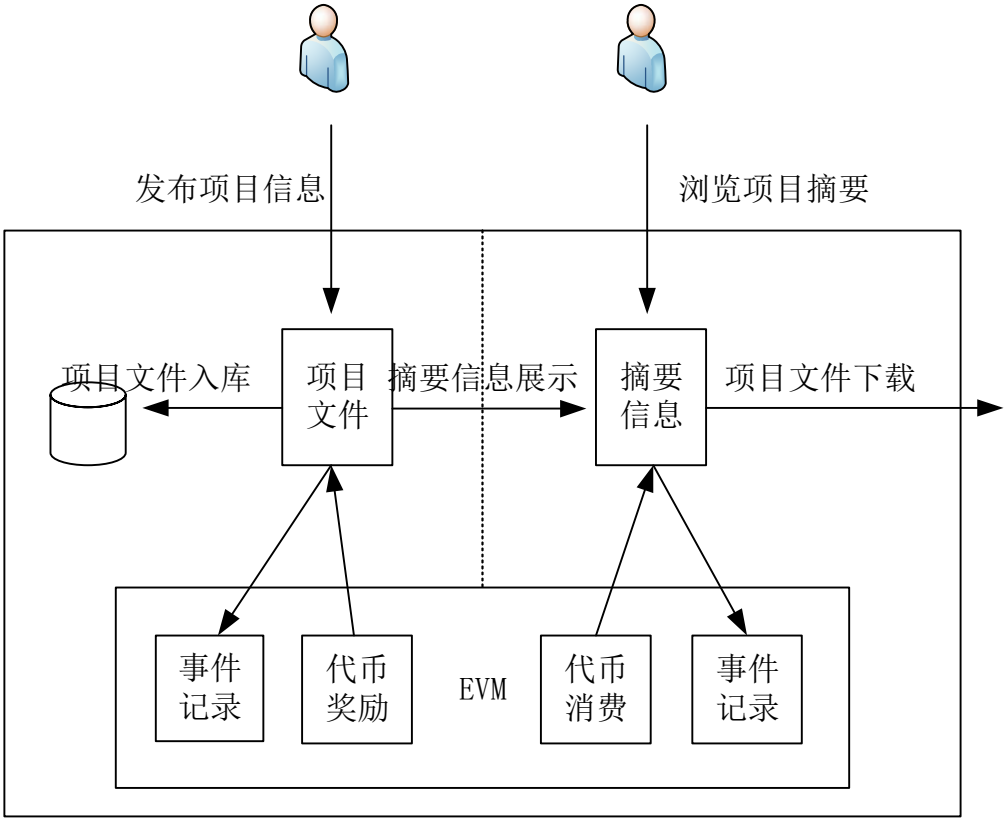

图 3.1 私募股权业务架构图

如图 3.1 所示，私募股权的业务场景下，用户主要分为两类，一类用户发布项目信息招募项目合作人；另一类用户通过浏览平台提供的项目摘要选取目标项目，在对项目感兴趣的情况下使用本平台代币购买项目。其中，发布项目信息可以得到相应的代币奖励，本平台所有交易均通过智能合约代币，项目发布、浏览项目和下载文件等操作均通过智能合约中的事件机制进行记录。

该业务场景流程图如图 3.2 所示，主要分为三种类型：管理流程、发布流程、购买流程。

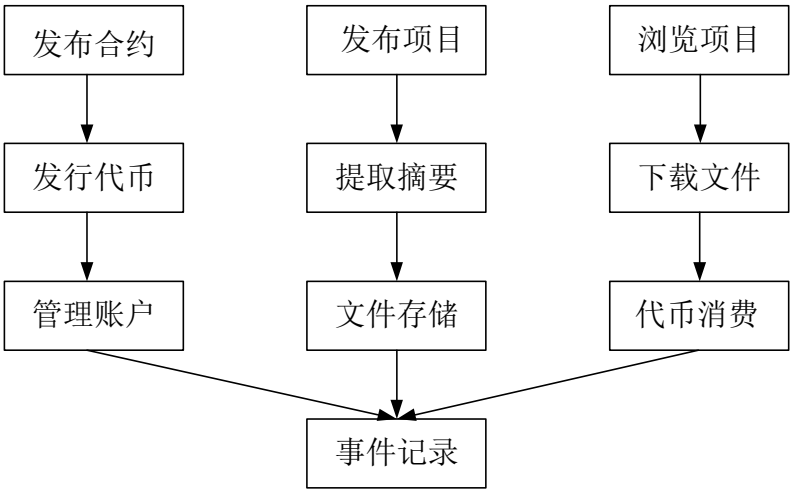

图 3.2 私募股权核心业务流程图

1）管理流程。

管理流程主要包括三个核心步骤，首先是发布智能合约，让实现编写好的智能合约运行于 EVM 中；其次是发行本系统的通用型代币；最后是管理账户。其中管理账户具体包括创建账户、给账户发币、账户转账等操作。

2）发布流程。

发布流程是在本平台发布项目的用户逻辑流程，主要包括三个核心步骤，用户发布项目到平台；后台提取项目摘要展示给需要购买项目的用户；项目文件存储到文件系统中。

3）购买流程。

购买流程针对的是购买项目的用户。此类用户首先在平台浏览需要购买的项目摘要，通过摘要判断需要购买后下载项目文件，同时消耗自己的代币进行购买。代币的来源通过充值后由智能合约发布。

3.2 基于状态空间的反馈负载均衡策略

3.2.1 负载均衡依据

以太坊的 Geth 客户端在 Docker 中的运行时，其内存占用率和 CPU 占用率会随着操作改变，同时影响了该服务节点的服务能力。本文设计合适的负载均衡调度策略来使得服务能力最大化。将运行中的容器的工作状态表示为一个参数向量，经实验验证，运行 Geth 客户端的 Docker 容器的内存占用率和 CPU 占用率对其服务能力影响较大，因此考虑将这两个参数作为建立容器状态空间依据，基于此，可以在一个二维平面映射服务节点。在本文服务系统中，服务节点的状态

大致可以分为两类，一类上述所选资源消耗度较小，其在二维平面的投影点距离原点较近，而第二类上述所选资源消耗较多，因此其在二维平面的投影距离原点较远。假定服务节点总数为 n，在任务分配的过程中根据服务节点的状态合理考虑分配模型。首先，计算 n 个服务节点在未执行任务的初始状态和执行任务后服务能力较差的状态。首先，假定二维状态平面内的服务节点投影点集合为

$$U=\{(x1, x2)|\ 0 \le x1 \le 1, 0 \le x2 \le 1\} \tag{3.1}$$

其中，x1 和 x2 是服务节点的内存占用率和 CPU 占用率的归一化指标。

读取 n 个服务节点分别在两种状态下的位置信息，并根据该信息求取到原点的距离，分别根据其距离求取平均值如下公式所示：

$$U1=\{(x1, x2)|\ 0 \le x1 \le 1, 0 \le x2 \le 1\} \tag{3.2}$$

$$Dis_{ave\min} = \frac{\sum_{i=1}^{m}\sqrt{x_1^{\ 2} + x_2^{\ 2}}}{m} \tag{3.3}$$

$$U2=\{(x1, x2)|\ 0 \le x1 \le 1, 0 \le x2 \le 1\} \tag{3.4}$$

$$Dis_{ave\max} = \frac{\sum_{i=1}^{m}\sqrt{x_1^{\ 2} + x_2^{\ 2}}}{m} \tag{3.5}$$

根据以上公式计算所致，状态平面中服务节点的分布情况大致如图所示：

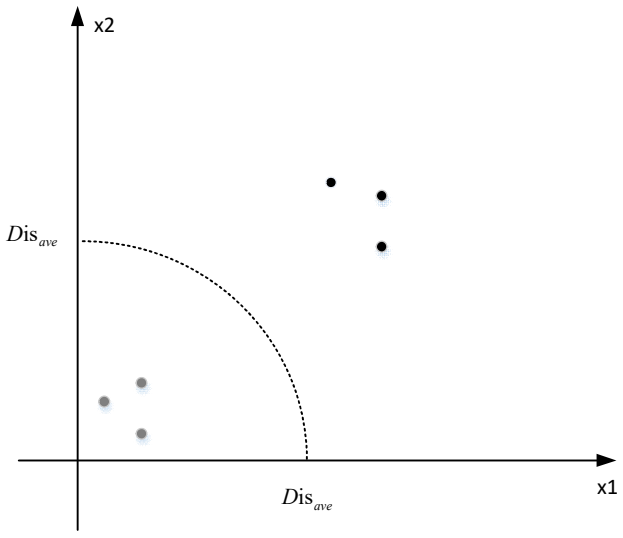

图 3.3 状态平面中的服务节点分布

计算以上平均值并以之做状态分界线：

$$Dis_{ave} = \frac{Dis_{ave\min} + Dis_{ave\max}}{2} \tag{3.6}$$

为评估以太坊服务系统的整体服务状态，定义健康参数作为负载均衡的健康参考。首先，假定 n 个服务节点，在二维状态平面中的服务节点的分布情况是有 n1 个节点在上述分界线以内，n2 个节点分布于分界线以外，然后定义健康参数为：

$$LBH = \sum_{i=0}^{m1}(Dis_{ave} - \sqrt{x_1^2 + x_2^2}) - \sum_{i=0}^{m2}(\sqrt{x_1^2 + x_2^2} - Dis_{ave}) \tag{3.7}$$

其中 n1+n2=n，因此简化该式为：

$$LBH = \sum_{i=0}^{m}(Dis_{ave} - \sqrt{x_1^2 + x_2^2}) \tag{3.8}$$

根据以上定义可知，当健康参数大于等于 0 时，服务节点的整体服务能力较强，且该指数越大说明服务能力越强。反之，如果该指数小于 0，则说明服务能力较差。

3.2.2 任务调度模型

本文设计的调度模型首先假定任务为将 m 个任务调度分配到 n 各服务节点，其中 n<m；基于区块链的应用业务场景，调度任务可根据任务类型分类为读写两种任务，对于写任务，可进一步划分，具体任务及其分类见下表：。由于读任务仅通过服务节点中的客户端调用相应的 API，故资源消耗较少，而写任务需要等待数据打包、验证等操作，因此耗时更长，资源消耗也更多。假设任务集合为 $T(n) = (t_1, t_2, ..., t_n)$，其中 t_i 为第 i 个子任务。任务 t_i 由参数表示即

$$t_i = (t_{memory} + t_{io\_memory}) \tag{3.9}$$

公式中，t_{memory} 是任务所需的内存大小，t_{io_memory} 是任务处理时需要的 IO 内存大小。

m 个容器资源可以表示成 $CON(i) = (con_1, con_2, ..., con_m)$，$con_i$ 表示第 i 个容器，其属性向量为

$$con_i = (con_{memory}, con_{io\_memory}) \tag{3.10}$$

公式 3.10 中，con_{memory} 是容器剩余可用内存，con_{io_memory} 是容器现有可用 IO 内存。

本调度模型的目标是让系统承受尽量大的并发，函数与约束条件为 max(n)；约束条件为：

$$\begin{cases} t_{i\_memory} <= con_{j\_memory} & i = 1,2,\cdots,n \\ t_{i\_iomemory} <= con_{j\_iomemory} & j = 1,2,\cdots,m \end{cases} \tag{3.11}$$

即仅在约束条件满足的时候才能将任务分配给相应的节点。

3.2.3 任务优先级模型

在区块链环境下，任务可以分为读任务 RT(Read Task)和写任务(Write Task)，读任务不消耗容器资源因此优先使用第二区域的容器，在第二区域容器均无空闲时选择第三区域的容器。对于写任务，可以分为发布合约任务 DT(Deploy Task)，发布合约任务较少，操作账户任务 MAT(Manage Accounts Task)次之，调用合约任务 MCT(Manage Contract Task)较多。本文根据任务出现的频率对其先后执行顺序，优先执行处理出现次数较少的任务。

任务优先级模型算法具体描述如下：

读任务与写任务分开执行；

读写任务可能在第二区域冲突，此时因为读任务耗时短且不消耗系统资源，优先执行；

读任务按时间先后顺序执行；

写任务分为发布合约任务，操作账户任务和调用合约任务，其在实际业务场景中出现频次依次增加，同时重要性会减小，因此优先级为降序排列；

具体任务优先级图示如下：

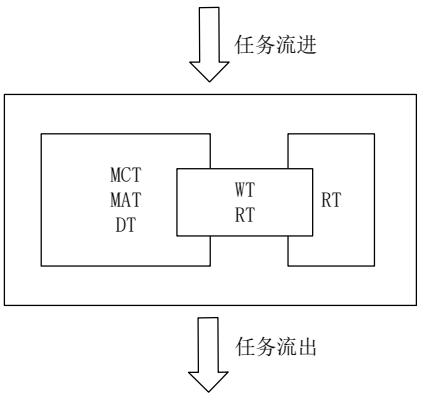

图 3.4 任务流优先级图示

3.2.4 任务分配模型与算法描述

任务的优先级决定了任务的调度顺序，在本文中，我们根据容器的健康参数将容器分为三个区域，由此，我们可以将读任务交给第二区域的容器处理，写任务交给第一区域的容器处理。第三区域的容器，进行定时回收重启处理。在各个区域内部，本文结合区块链场景下的任务的性质以及资源的使用和分配情况，构建动态优先级。

因为容器的状态影响其执行写操作，因此我们将容器按状态分为两类，具体映射到状态平面空间为 I 区、II 区。以公式(3)得到的数值 Dis_{ave} 为分界线。

我们将容器状态平面分为两个部分，I 区代表安全区，II 区代表非安全读区。

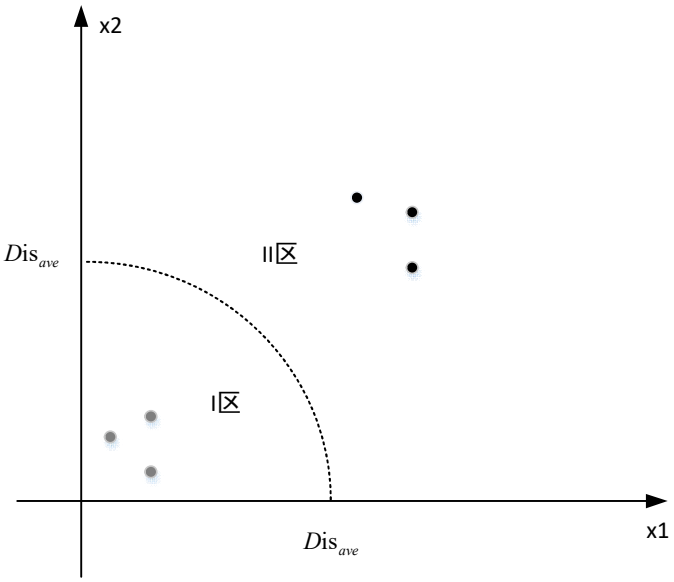

图 3.5 容器状态平面分区示意图

任务分配模型描述：

对于写任务，优先分配到 I 区，当 I 区没有节点时分配到 II 区；

对于读任务，优先分配到 II 区，当 II 区没有节点时分配到 I 区；

本文设计的面向区块链微服务化场景下的任务调度算法描述如下：

将任务根据任务的读写性质进行分类并且按时间排序；

将任务依次动态的分配到适合的区域中的容器中；

更新容器的状态信息，动态将任务分配到其中；

将第三分区中的容器依次进行重启操作，回收资源；

检查健康状态参数，若健康状态参数为负数，则暂停如任务分配，等待知道健康状态为正数则继续分配任务；

返回流程开始下一个任务的调度；

3.3 基于负载均衡的区块链服务系统设计

3.3.1 系统设计目标及原则

以太坊服务平台的系统设计应该遵循微服务架构设计原则，具体如下：

1）扩展性与伸缩性

扩展性是指系统在业务需求的需要增加的情况下能够方便快速扩展的能力；伸缩性是指指不断向集群中添加服务器来缓解不断上升的用户并发访问压力和不断增长的数据存储需求。扩展性的实现有两种方式，首先是使用消息队列进行解耦，在服务之间传递通信；其次是将业务进行拆分复用，在本架构中即将业务拆分为读写业务。

2）易维护性

本设计涉及大量的分布式组件，系统的维护工作应该尽可能地科学和简单，避免大量地维护工作带来地开销和负担。

3）高性能和资源利用率

系统的响应速度吞吐量等指标应满足高性能要求。同时，在满足系统性能要求的情况下尽可能少使用资源，使得资源地利用率达到最高。

3.3.2 系统的整体架构

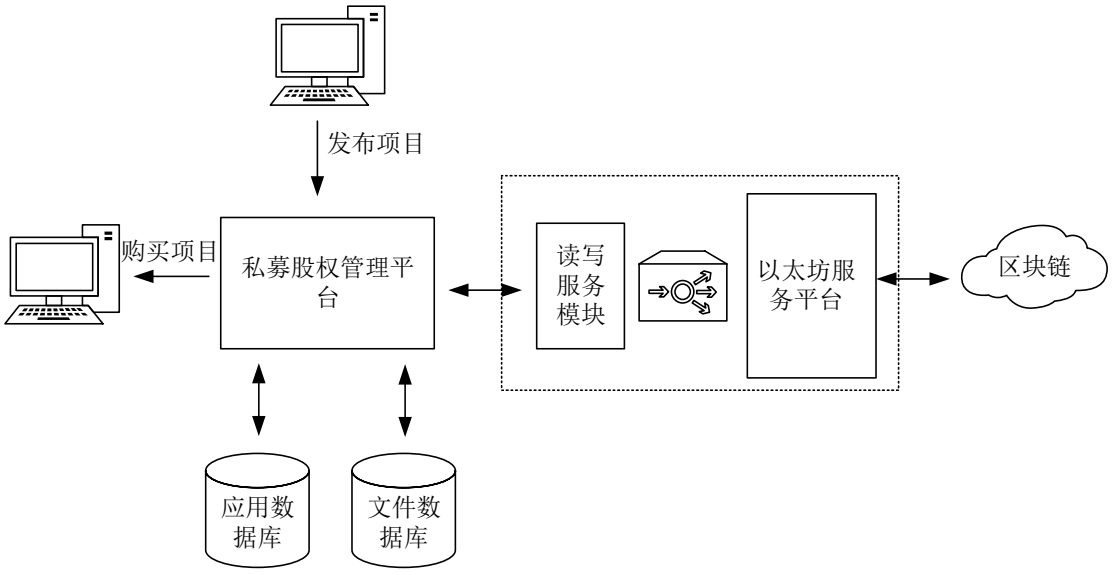

图 3.6 私募股权平台架构图

如图 3.6 所示为私募股权项目架构图，其中读写服务模块搭配负载均衡对以太坊服务平台调度构成了以太坊服务平台，底层与区块链平台进行交互，应用层对外提供业务接口；其中，私募股权平台业务相关模块简要表示。

3.3.2.1 读写服务模块

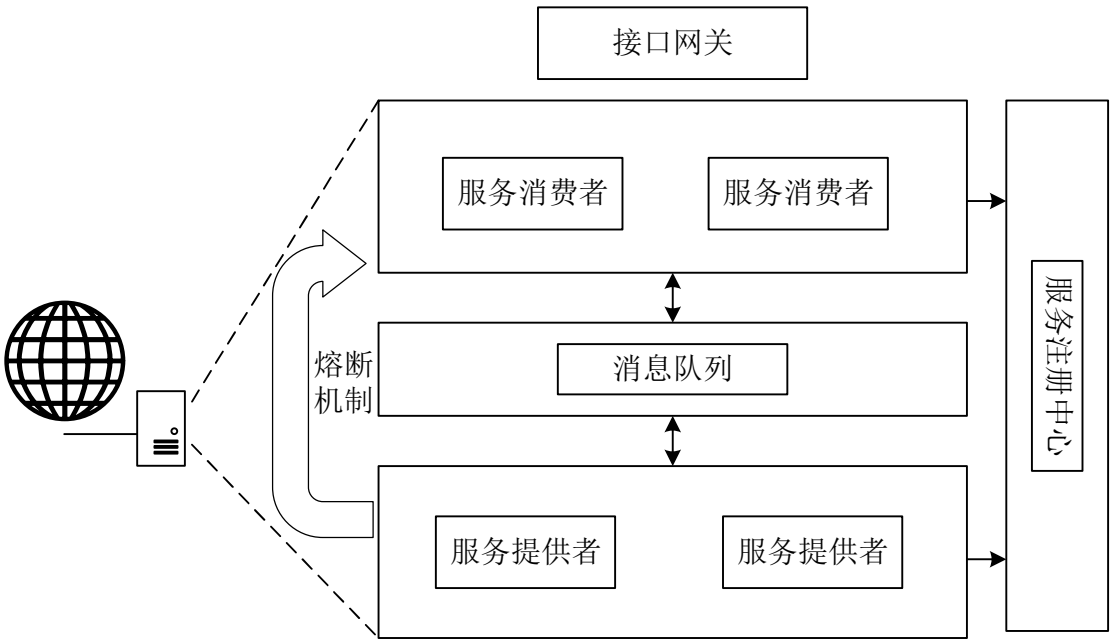

图 3.7 读写服务模块架构图

读写服务模块架构如图 3.2 所示，该模块主要由五大部分构成：服务注册中心、接口网关、消息队列、服务消费者和服务提供者。

1）服务注册中心。

服务注册中心是系统管理各个微型服务的模块。在本系统中，一个服务随时可能下线，也可能应对临时访问压力增加新的服务节点。以及服务与服务之间的感知，均有此模块实现。

2） 接口网关。

接口网关对后台服务的接口进行管理，并对外提供统一的调用接口。接口在调用不受后台单个服务的上线或下线影响。接口网关的作用可以总结为：提供统一的服务入口，让微服务对前台透明；聚合后台的服务，节省流量同时提升性能；提供安全，过滤，流量控制等 API 管理功能。

3）消息队列。

消息队列是服务与服务之间的异步调用方式，它既能减低调用服务之间的耦合，又能成为调用之间的缓冲，确保消息积压不会冲垮被调用方，同时能保证调

用方的服务体验，继续干自己该干的活，不至于被后台性能拖慢。

4）服务消费者。

服务消费者是轻量的，根据业务特点分别封装了读写相关操作，在接口网关对服务消费者的调用的时候即可实现第一级负载调度，本级负载调度可采用轮询或加权等方式，对本系统性能影响较小。

5）服务提供者。

服务提供者重量的耗时的，提供直接操作以太坊客户端的功能。在服务消费者和服务提供者之间使用消息队列进行解耦和做任务缓冲。服务者和提供者均由服务注册中心进行统一管理。

3.3.2.2 以太坊服务平台

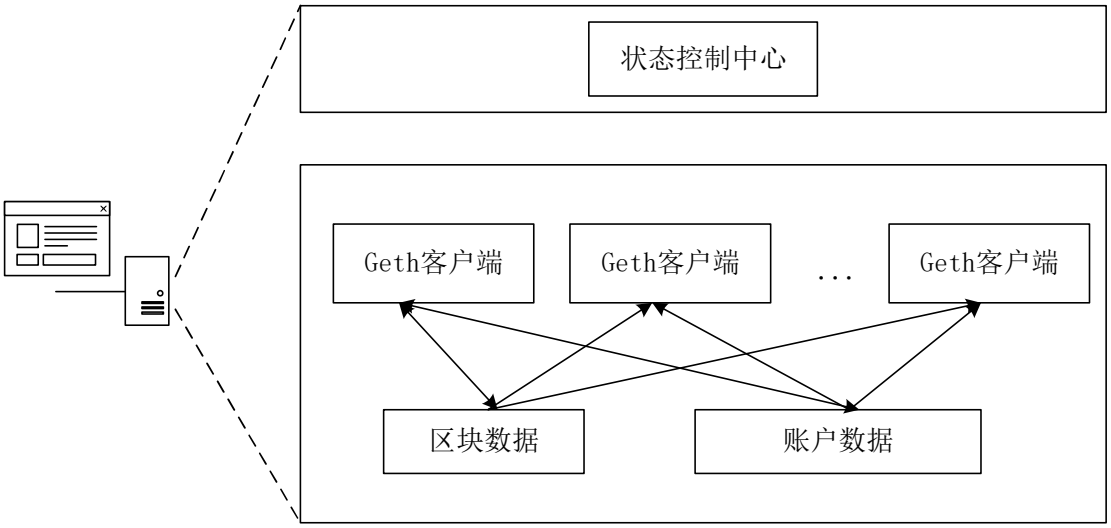

图 3.8 以太坊服务平台架构图

以太坊服务平台如图 3.8 所示，主要分为两个部分：Geth 客户端与状态控制中心。

1） Geth 客户端。

Geth 客户端是以太坊官方提供的与以太坊网络交互的工具，提供常用操作的 RPC 调用 API。将 Geth 客户端封装运行于 Docker 容器中对外提供服务，由 Docker-Compose 进行容器管理。

2） 状态控制中心

状态控制中心用于管理 Geth 客户端的上线与资源回收，同时监控容器的状态，提供给负载均衡模块做反馈负载均衡依据。

3.4 本章小结

本章首先分析了私募股权平台中的业务特点，介绍了以太坊在该项目中的应用架构形式，总结了该架构存在的安全性稳定性以及吞吐量等方面问题。然后，根据分析的结果，设计了反馈负载均衡算法，考虑服务节点的状态并根据任务性质对任务进行分类调度。最后，基于微服务的思想设计了以太坊服务平台的架构，并对介绍了其中的核心模块。

第四章 以太坊服务系统实现

本章是基于第三章的系统设计，对以太坊服务平台的系统实现，如图 5.1 所示，为基于负载均衡的以太坊服务平台实现的技术架构。

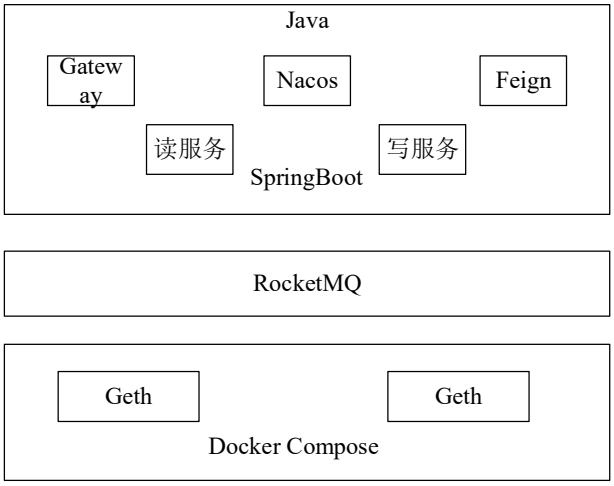

图 4.1 基于负载均衡的以太坊服务平台实现技术架构图

如图所示，使用 Java 语言基于 SpringBoot 框架进行开发，基于微服务思想开发后台服务系统，根据读写性能的差异特点分别实现读、写业务服务，并对外提供 REST API。使用 nacos 作为服务注册中心，在服务提供者与消费者之间使用 Kafka 消息中间件实现服务间的异步通信。基于 Docker 封装以太坊 Geth 客户端，并使用 Docker Compose 作为容器编排工具，实现底层以太坊环境的快速搭建。

4.1 系统的业务组件

4.1.1 写操作服务

私募股权平台中的写业务服务具体实现包含了会将信息写入区块链系统的所有操作，其中操作大致分为两个部分，智能合约操作部分和账户操作部分。为了更好的解耦，对以上业务进行纵向拆分，分为服务提供者与服务消费者，服务消费者与服务提供者之间使用 RocketMQ 进行消息的缓存以缓解整个系统的负载压力。整体架构如图 1 所示，其中服务消费者对外提供 RESTAPI 接口，向下通过消息队列调用服务提供者，因此较轻量级；服务提供者向上提供消费者的调用，向下连接以太坊服务网络，因此较重量级。

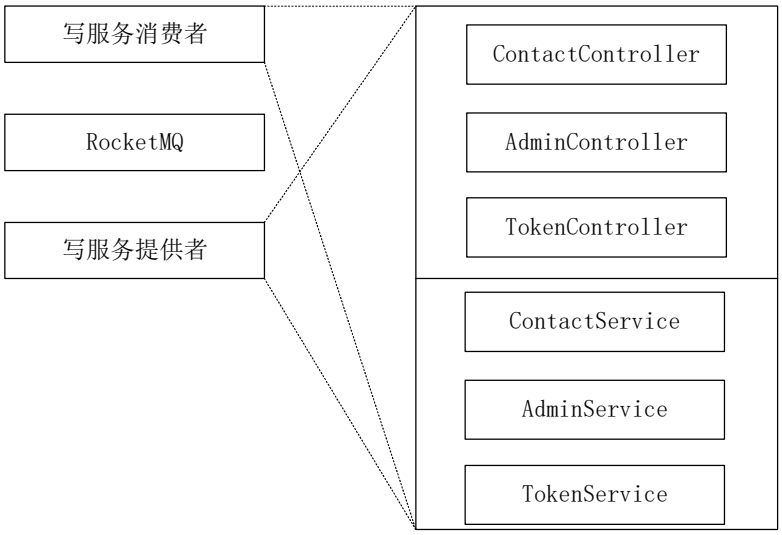

图 4.2 写服务架构图

对于写服务消费者，基于 SpringBoot 实现，因此具有以下启动类。本文后续所有的服务提供者与服务消费者均在项目根目录下包含类似的启动类。其中@SpringBootApplication 是 SpringBoot 项目的启动注解，@EnbleDiscoverClient 是方便 APIGateway 检测的注解，@ EnableFeignClients 是熔断机制需要使用到的注解。

服务消费者启动类

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class EthereumConsumerFeignApplication {
    public static void main(String[] args) {
        SpringApplication.run(EthereumConsumerFeignApplication.class, args);
    }
}
```

服务消费者的内部结构如图 2 所示，其中 BaseResult 是整个系统的 API 的返回值的基本结构，分别由数字 200 和 500 代表此次操作的结果是成功还是失败，其核心代码如下所示：

系统返回体核心信息

```
public class BaseResult implements Serializable {
    public static final int STATUS_SUCCESS = 200;
```

```
    public static final int STATUS_FAIL = 500;
    private int status;
    private String message;
    private static BaseResult createResult(int status, String message){
        BaseResult baseResult = new BaseResult();
        baseResult.setStatus(status);
        baseResult.setMessage(message);
        return baseResult;
    }
}
```

写服务主要包括三个主要接口类，即合约操作类、账户管理类，它们包含的接口及其解释如表所示：

Controller	API	Input	Output	Meaning
Contract	/contract/deploy	Contract.class gasPrice gasLimit	ContractAddress Status	发布智能合约
AdminManage	/admin/new	password	AccountAddress Status	创建新账户
	/admin/unlock	address&password	AccountAddress Status	解锁新账户
TokenManage	/token/add	address&value	AccountAddress Status	为某账户添加代币
	/token/delete	address&value	AccountAddress Status	从某账户删除代币
	/token/transfer	from&to&value	Status	在账户之间转移代币
	/token/set	address&value	AccountAddress Status	设置某个账户的代币值
	/token/publish	address time projectName projectHash	projectAddress	发布项目
	/token	address	buyProAddress	购买项目

		time projectName projectHash		

图 4.1 写服务主要接口

服务提供者与服务消费者的轻量级与重量级体现在具体实现上，消费者的 Service 实现仅通过注解调用提供者提供的相应接口，而提供者需要通过 web3j 工具与 Geth 客户端建立连接。如下表所示分别为消费者与提供者的 Service 实现举例：

服务消费者实现举例

```
@FeignClient(value = "ethereum-provider")
public interface AdminManageService {
    @RequestMapping(value = "/admin/newaccount")
    public BaseResult personalNewAccount(String password);
}
```

服务提供者实现举例

```
@Service
public class AdminManageServiceImpl implements AdminManageService {
    Admin web3j = Admin.build(new HttpService("http:192.168.198.141:8545"));
    @Override
    public String personalNewAccount(String password) throws IOException {
        CompletableFuture<NewAccountIdentifier>
newAccountIdentifierCompletableFuture =
web3j.personalNewAccount(password).sendAsync();
        if (newAccountIdentifierCompletableFuture != null)
            return newAccountIdentifierCompletableFuture.toString();
        return null;
  }
```

4.1.2 读操作服务

私募股权平台中的读业务服务的实现与写业务的实现类似，该服务的功能目标是从以太坊网络中读取账户、代币等信息，不改变以太坊上的信息。同样，该

模块的结构分为服务提供者与服务消费者，子模块基于 SpringBoot 实现。整体架构如图 2 所示，其中服务消费者对外提供 RESTAPI 接口服务，服务提供者通过 web3j 与 Geth 客户端建立 Http 连接进行通信。

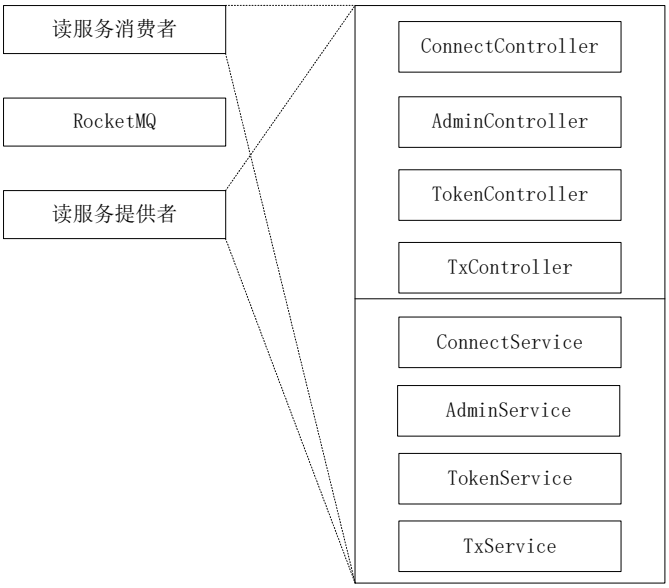

图 4.3 读业务服务模块

如前文所述，该模块在根目录下包含 SpringBoot 启动类，系统返回类 BaseResult 的定义不变。读服务主要包括四个主要接口类，即连接管理类，账户管理类， 代币管理类和交易管理类，它们包含的接口及其解释如下表所示：

Controller	API	Input	Output	Meaning
Connect	/conn/connect		Status	连接以太坊
	/conn/blockNum		blockNum	获取区块数
	/conn/gasPrice		gasPrice	获取当前 gas
AdminManage	/admin/list		Accounts Status	列举全部用户
TokenManage	/token/balance	Address	Balance Status	获取某账户的代币
	/token/total		Balance Status	获取总代币金额
TxManage	/tx/blockhash	blockHash	Block Status	获取某区块信息
	/tx/blocknumber	blockNum	Block Status	通过 BlockNum 获取区块信息
	/tx/txhash	txHash	Tx Status	获取某条交易信息
	/tx/txNum	blockNum	txNum Status	获取某个区块中的交易数量

	/tx/receipt	txHash	Address Status	获取某交易的接收方

其中，消费者与服务提供者的实现原理与写操作类似，此处不再说明。

4.2 系统功能组件

4.2.1 服务注册中心

服务注册中心是微服务系统中发现、配置和管理各个服务的机构，在 Spring Cloud Netflix 阶段采用的是 Eureka 作为服务注册和发现服务器，而在 Spring Cloud Alibaba 解决方案中采用的是 Nacos 组件解决此问题。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。 Nacos 是构建以“服务”为中心的现代应用架构 (例如微服务范式、云原生范式) 的服务基础设施。

Nacos 的 docker-compose.yaml 文件

```
version: '3.1'
services:
  nacos:
    restart: always
    image: nacos-server
    container_name: nacos
    ports:
      - 8848:8848
    volumes:
      - /usr/local/docker/nacos/webapps/test:/usr/local/nacos/webapps/test
    environment:
      TZ: Asia/Shanghai
```

首先，基于 Docker-compose 构建 nacos 的镜像，使得 nacos 的运行基于 Docker 容器。然后在服务的配置中加入 nacos 配置，具体如下：

服务中的 nacos 注册

```
spring:
  application:
    name: ethereum-provider
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

当 nacos 检测到服务运行的时候，在浏览器上输入地址 http://localhost:8848/nacos 在操作台结面会显示该服务的相关信息，如下图所示：

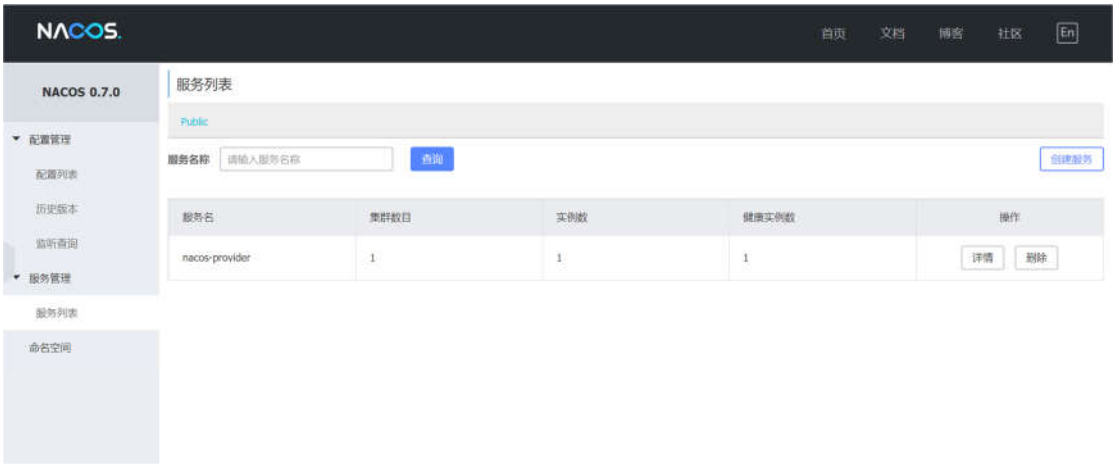

图 4.4 nacos 服务检测界面

4.2.2 API 网关

API Gateway 是为了系统提供统一的接入网关机构，Spring Cloud Gateway 是 Spring 官方基于 Spring 5.0，Spring Boot 2.0 和 Project Reactor 等技术开发的网关，Spring Cloud Gateway 旨在为微服务架构提供一种简单而有效的统一的 API 路由管理方式。Spring Cloud Gateway 作为 Spring Cloud 生态系中的网关，目标是替代 Netflix ZUUL，其不仅提供统一的路由方式，并且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控/埋点，和限流等。

Spring Cloud Gateway 的工程流程如下图所示：

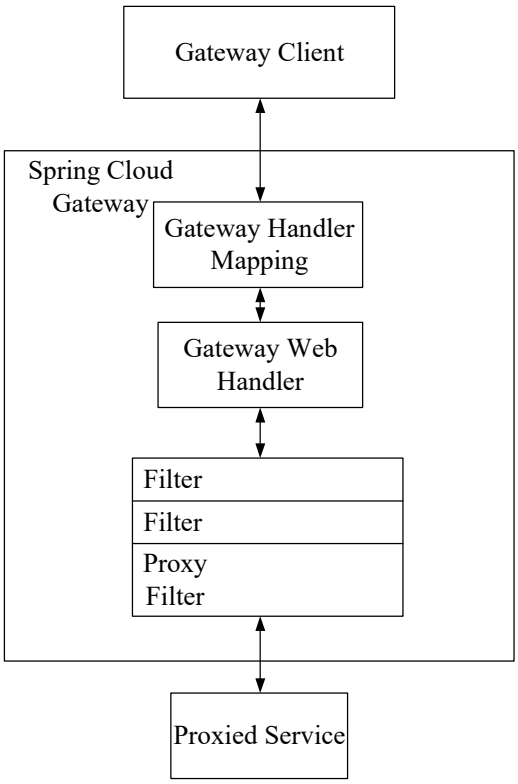

图 4.5 API Gateway 工作流程

客户端向 Spring Cloud Gateway 发出请求。然后在 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler。Handler 再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前（pre）或之后（post）执行业务逻辑。

值得注意的是，Spring Cloud Gateway 不使用 Web 作为服务器，而是 使用 WebFlux 作为服务器，Gateway 项目已经依赖了 starter-webflux，所以这里 不要依赖 starter-web。

Spring Cloud Gateway 的 application.yml 配置具体如下：

服务中的 nacos 注册

```
spring:
  application:
    name: ethereum-provider
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
```

```
spring:
  application:
    name: spring-gateway
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
    sentinel:
      transport:
        port: 8721
        dashboard: localhost:8080
    gateway:
      discovery:
        locator:
          enabled: true
      routes:
        - id: NACOS-CONSUMER
          uri: lb://nacos-consumer
          predicates:
            - Method=GET,POST
        - id: NACOS-CONSUMER-FEIGN
          uri: lb://nacos-consumer-feign
          predicates:
            - Method=GET,POST
```

API Gateway 的 Application 类上添加注解@EnableFeignClients，具体如下：

添加@ EnableFeignClients 的 Application

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

4.2.3 熔断保护

在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以通过 RPC 相互调用，在 Spring Cloud 中可以用 RestTemplate + LoadBalanceClient 和 Feign 来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证 100% 可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet 容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。为了解决这个问题，业界提出了熔断器模型。阿里巴巴开源了 Sentinel 组件，实现了熔断器模式，Spring Cloud 对这一组件进行了整合。在微服务架构中，一个请求需要调用多个服务是非常常见的，较底层的服务如果出现故障，会导致连锁故障。当对特定的服务的调用的不可用达到一个阀值熔断器将会被打开。熔断器打开后，为了避免连锁故障，通过 fallback 方法可以直接返回一个固定值。

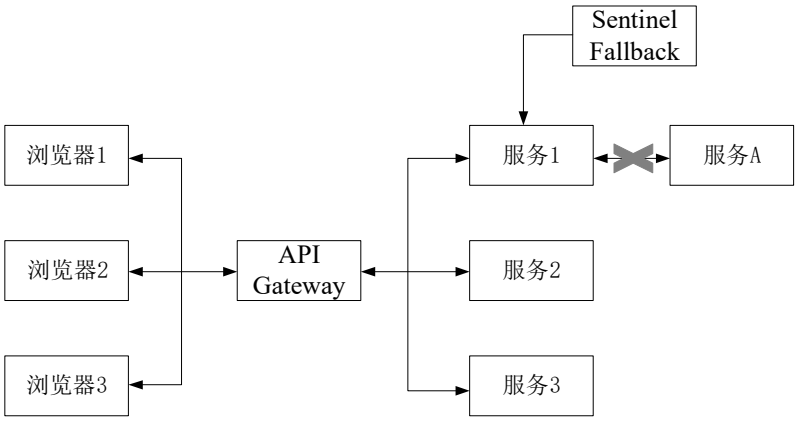

图 4.6 熔断保护原理示意图

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Sentinel 以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。如果要在您的项目中引入 Sentinel，使用 group ID 为 org.springframework.cloud 和 artifact ID 为 spring-cloud-starter-alibaba-sentinel 的 starter。

在 Service 中添加 Fallback 的指定类：

添加 Fallback 的指定类

```
@FeignClient(value = "nacos-provider", fallback = EchoServiceFallback.class)
public interface EchoService {
    @GetMapping(value = "/echo/{message}")
    String echo(@PathVariable("message") String message);
}
```

创建熔断器类并实现对应的 Feign 接口：

添加 Fallback 的指定类

```
@Component
public class EchoServiceFallback implements EchoService {
    @Override
    public String echo(String message) {
        return "echo fallback";
    }
}
```

当服务提供者不能提供服务时，熔断保护将直接返回 Fallback 中设置的信息。

4.3 区块链服务平台

以太坊的 Geth 客户端在 Docker 官方仓库 Docker Hub 中有镜像，但是由于我们需要用 Docker Compose 做容器编排工具，并且需要根据负载均衡策略重启镜像，因此本文需要重新制作镜像，并构建容器。制作镜像的 Dockerfile 内容如代码所示，在其所在目录使用命令 docker build –t 镜像名，即可制作镜像。

Dockerfile

```
FROM golang:1.13-alpine as builder
RUN apk add --no-cache make gcc musl-dev linux-headers git

ADD . /go-ethereum
RUN cd /go-ethereum && make geth

FROM alpine:latest
```

```
RUN apk add --no-cache ca-certificates
COPY --from=builder /go-ethereum/build/bin/geth /usr/local/bin/

EXPOSE 8545 8546 8547 30303 30303/udp
ENTRYPOINT ["geth"]
```

在完成镜像的构建后，基于其开启容器，采用 Docker Compose 编排，docker-compose.yaml 文件内容如代码所示。使用 docker-compose -f docker-compose.yaml up –d 命令即可启动容器。

Docker-compose.yaml

```
version: '2'
services:
    bootnode:
        container_name: bootnode
        image: ubuntu:geth
        #command: sh -c 'sleep 1000000'
        command:    bootnode -nodekey=bootnode.key
        working_dir: /data
        volumes:
          - /home/sglfe/Workspace/ethereum/study/dockernet/bootnode:/data
        ports:
          - 30301:30301

      node0:
        container_name: node0
        image: ubuntu:geth
        command:    geth --datadir ./ --networkid 88 --rpcport 8545 --rpc
--rpcaddr 0.0.0.0 --rpcapi db,eth,net,web3,personal,admin,miner --bootnodes enode:
        #command: sh -c 'sleep 1000000'
        working_dir: /data
        volumes:
          - /home/sglfe/Workspace/ethereum/study/dockernet/node0-data:/data
        ports:
          - 8545:8545
```

```
    - 30303:30303
  depends_on:
    - bootnode
```

综上，即基于 Docker 容器构建了以太坊服务平台，在负载均衡策略的配合工作下，有效的提高区块链平台对外的服务能力，在需要的时候，启动多个容器对系统服务。

4.4 负载均衡模块

负载均衡模块针对主要实现两个功能，首先是读取容器的状态数据，然后是通过对容器的状态计算得到下一个任务访问的 Geth 客户端；针对任务，该模块从 Rocket 中读取任务组后判断任务的类型，然后根据第三章的任务分发算法决定任务的执行顺序。主要函数核心代码如下所示：

代码：负载均衡

输入：

服务节点的状态数组：*gethStats*；

任务数组：*tasks*；

输出：

下一个任务：*targetTask*；

目标服务节点：*targetGeth*；

核心代码：

```
    LoadTaskToGeth(GethStat[] gethStats, Task[] tasks )
1.  int lbh = 0; //初始化健康参数 lbh
2.  List<GethStat> gethPool; //初始化可用服务节点池
3.  for gethStat in gethStats
4.      //x1 代表 geth 的内存占用率，x2 代表 cpu 占用率
5.      //Disave 的定义见 3.2 节
6.      lbh =lbh + Disave – Math.sqrt(gethStat.x1^2 + gethStat.x2^2)
7.      if Disave – Math.sqrt(gethStat.x1^2 + gethStat.x2^2) < 0
8.      then gethPool.add(gethStat)
9.  if lbh < 0 then goto 3 //如果健康指数小于 0 持续回收服务节点
10. List<Task> taskQueue //初始化任务执行序列
11. for task in tasks
```

```
12.    if task.type == w
13.        then taskQueue.add(task)
14.    else
15.        taskQueue.addFirst(task)
16. return gethPool.get(0), taskQueue.get(0)
```

4.5 本章小结

本章节详细介绍了以太坊服务平台的系统实现，使用 Java 语言基于微服务解决方案 Spring Cloud Alibaba 开发分布式后台服务程序。业务组件层基于 SpringBoot 框架分别封装读写服务模块，进行纵向拆分为服务提供者与服务消费者；系统功能组件基于 SpringBoot 框架开发，提供服务注册、API 网关和熔断保护的功能；以太坊服务平台基于 Docker 容器安装以太坊 Geth 客户端。

第五章 实验与分析

本章的主要内容是介绍和分析本文设计实现的系统工具实验结果，先从面向用户的角度介绍和分析工具使用的性能结果，再基于案例分析，验证和分析系统工具的各模块实验结果，最后总结了该方法的局限性。

5.1 系统功能评估

5.1.1 系统后台功能

本系统的后台功能即无需用户参与的区块链服务功能，如发布智能合约、发布代币、挖矿和管理账户等功能。由于本部分涉及功能较多，在对此类功能依次进行测试后，仅选取较核心功能在此作为举例说明。本部分功能测试均通过控制台和 Postman API 测试并得到结果。

1. 发布智能合约

智能合约运行于以太坊 EVM(Ethereum Virtual Machine)，其中的里包括了系统中代币的设计，记录代币转移和发布项目等相关操作。在本系统中，智能合约由 solidity 语言编写，然后经过 web3j 提供的转换工具编译转为 java 可执行文件，最后通过调用接口进行发布，如图 5.1 为发布合约成功后的测试图，。。。为合约地址。

2. 创建账户和查看账户

账户是以太坊中参与挖矿、交易等相关操作的基本单位，账户的外在表现为一个 64 位的十六进制数。如图所示，是在命令行中创建和展示账户。本系统后台封装了相应的接口，通过 RPC 调用对外提供 API 接口实现此类操作，如图所示为通过接口实现该类功能的示例。

```
> eth.accounts
["0xb85ced6ba5173dffefc232782685d3ef8bd55bec"]
> personal.newAccount("123456")
"0x2e7d1274f3d9ce39d9733afebef3f27d17d1d8f5"
> eth.accounts
["0xb85ced6ba5173dffefc232782685d3ef8bd55bec", "0x2e7d1274f3d9ce39d9733afebef3f27d17d1d8f5"]
> personal.newAccount("0x2e7d1274f3d9ce39d9733afebef3f27d17d1d8f5")
"0x9a27687ccedf70549362cdb10b9676e7d76ffc3d"
> eth.coinbase
"0xb85ced6ba5173dffefc232782685d3ef8bd55bec"
> miner.setEtherbase(eth.accounts[1])
true
> eth.coinbase
"0x2e7d1274f3d9ce39d9733afebef3f27d17d1d8f5"
>
```

图 5.2 命令行操作账户

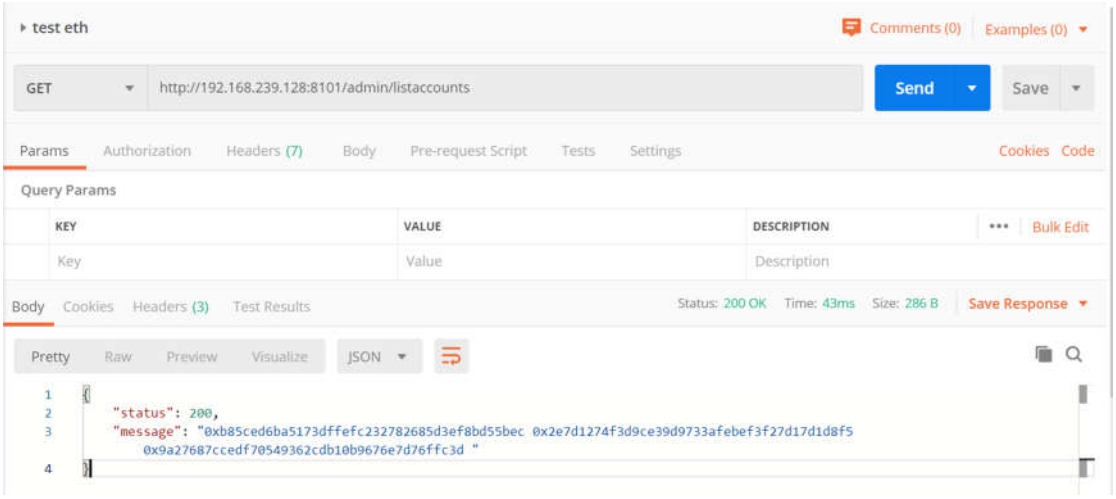

图 5.3 通过 HTTP 操作账户

3. 挖矿

挖矿是本系统中节点之间的工作量证明的共识过程，在向区块链中写入信息时，比如转账、创建账户等操作的时候，需要后台开始挖矿。如图所示是通过命令行测试此时的挖矿状态以及开启和关闭挖矿操作。挖矿会产生区块，通过区块的数量变化可以验证挖矿是否成功，如图所示，区块数量从 195 变化到 206 即在这段时间内挖矿产生了 11 个区块。如图所示，是通过 API 调用操作挖矿相关操作。

```
> eth.mining
false
> miner.start()
null
> eth.mining
true
> miner.stop()
null
> eth.blockNumber
195
> miner.start()
null
> miner.stop()
null
> eth.blockNumber
206
```

图 5.4 通过命令行进行挖矿操作

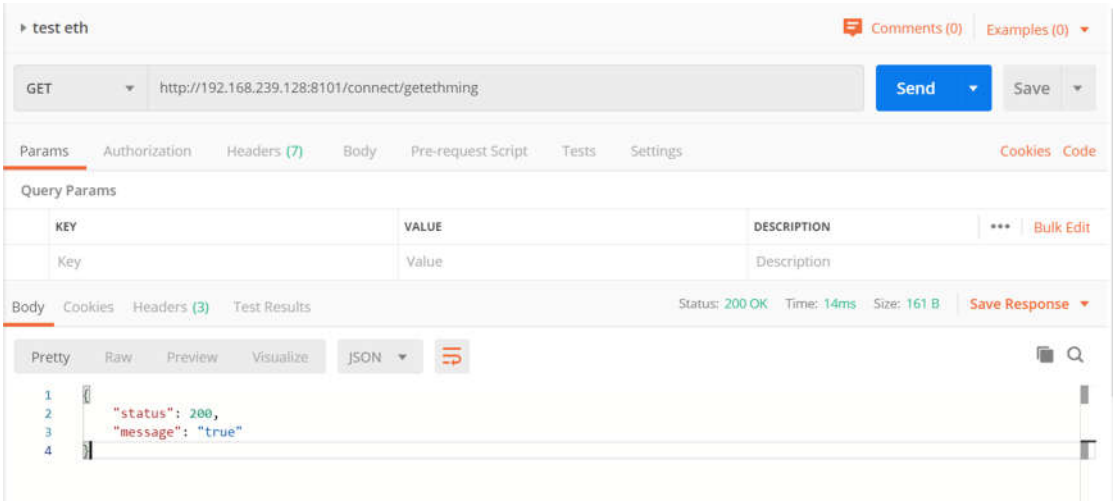

图 5.5 通过 HTTP 进行挖矿相关操作

5.1.2 用户功能

本系统的用户功能即用户参与的以太坊服务功能，如充值、发布需求和购买项目信息等。本部分的后台服务在 APP 端已有良好的界面，因此通过 APP 截图展示。

1. 账户充值

账户充值是充值人民币之后经过兑换成为本系统代币的功能，用户在购买项目信息或者需求信息的时候需要支付的是本系统的代币，因此需要进行充值。如图所示为系统充值界面，用户支付渠道包括支付宝和微信。

图 5.6 账户充值界面

2. 发布需求

需求信息类似于用户的招募信息，用户通过 APP 端发布相关需求概要、需求文件以及需求的具体要求等信息到系统中，其他用户在浏览到类似信息后判定是否要购买详细信息文件，如果需要即可通过本系统代币购买，如图所示为发布需求界面。

图 5.7 项目发布和列表展示界面

3. 购买项目

当用户浏览到需要购买的项目时，可以使用本系统代币购买详细的项目文件，如图为项目购买界面。项目发布、购买等操作均通过事件的方式记录在以太坊智能合约中。

图 5.8 项目查找和购买界面

5.2 系统性能评估

5.2.1 对照架构

本文基于微服务思想构建读写分离的以太坊后台服务系统，底层基于 Docker 容器构建多 Geth 客户端的以太坊服务平台，并基于状态空间反馈设计合理的负载均衡策略将任务合理分发到对应客户端。为了验证本系统在性能上的进步，本文对照单体应用架构的以太坊服务系统做对照实验。

单体应用在具有不同的逻辑模块化架构，但应用程序被作为一个单体进行打包和部署，系统使用 Java 实现，基于 SpringBoot 框架构建并编译成 WAR 包直接运行于应用服务器 Tomcat 中。底层客户端的调用同样使用单客户端模型，因此无需负载均衡策略。

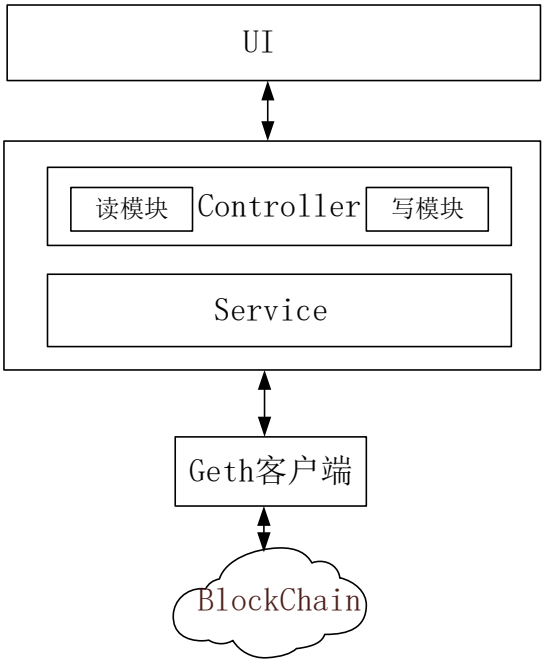

图 5.9 对照实验中的单体应用架构

如图所示为对照实验的单体应用架构，该应用内部同样区分为读和写逻辑模块，但是在实际的应用架构中是作为一个整体部署的。底层的 Geth 客户端的服务也是单个的 Geth 客户端对整个应用提供服务。

5.2.2 实验设计

目前面向私募股权平台的版权交易平台运行良好，本部分测试内容主要针对性能测试。功能测试即是否可以完成创建账户，转账，发布智能合约和挖矿等已于上一小结测试。性能测试通过与单客户端与单体区块链服务后台系统构成的区块链服务平台做并发性能对比。

本文测试环境为基于 VMware station pro 构建的虚拟机。VMware station pro 搭建于单机电脑，其配置为：

系统 Windows 10 专业版 64 位；

处理器 Intel(R) Core(TM) i7-5600U @2.6GHz；

内存 12GB；

VMware station pro 相关配置为：

版本 12.5.7 build-5813279；

系统 Ubuntu 16.04；

内存 2G；

CPU 2 核；

硬盘 20G；

容器 Docker 19.03.4；

为模拟真实环境的请求情况，本测试涉及的区块链系统操作主要分为四类，分别是连接管理、账户管理、交易管理与合约发布。四类操作选取典型操作代表做并发性测试，值得注意的是在我们无法准确的预测在真实的业务场景中各种操作的具体数量，因此本文仅将各种操作按照大致比例进行混合操作，具体比例见表 5.1。

功能分类	详细功能	所占比例
连接管理	获取某账户币值	20%
	获取当前主账户	5%
	获取本客户端所有账户	5%
	获取区块数量	10%
账户管理	创建新账户	5%
	展示当前所有账户	5%
	解锁账户	5%
交易管理	发送交易	30%
	获取交易哈希	5%
	通过交易哈希获取交易内容	5%

合约发布	发布智能合约	5%

表 5.1 并发测试数据

性能测试的维度包括两个方面，首先是并发性能，其次是资源消耗情况。并发性能通过完成固定数量的请求的时间展现，资源的消耗通过对 CPU 和内存的占用率的变化来展现。因区块链底层服务平台是多客户端架构，为了测试多客户端在提供服务方面的能力，这里将采用的测试分别采用 3 客户端、5 客户端和 8 客户端三种形式进行测试。对于并发性方面的测试，我们分别在 200、600、1000 和 1200 并发量的情况下测试响应时间(单位 s)。

5.2.3 实验结果

依照实验设计，本文通过模拟请求测试并发性，并同时检测各个容器的内存和 CPU 的占用情况。如图 5. 为并发测试结果。

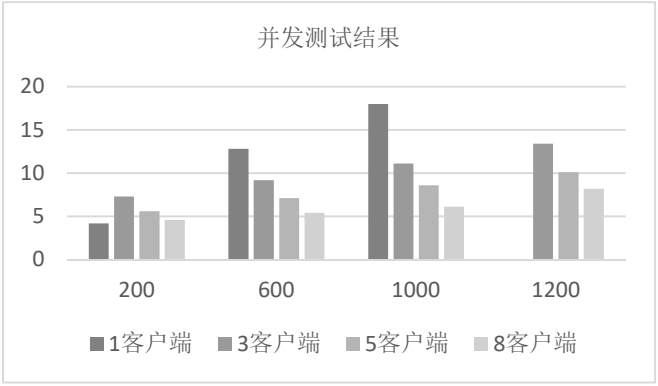

图 5.10 并发测试结果

从上图中可以看出，在 200 并发量的时候，单客户端的完成速度高于多客户端，这是因为在多客户端之间的切换需要消耗时间。因此在低并发情况下，单客户端架构性能优于多客户端架构。在并发量到 600 及往上的时候，多客户端架构的优越性逐渐体现出来，多客户端的任务完成时间小于单客户端的任务完成时间，而且随着客户端的数量的增加，任务完成的时间变得更短。在并发量达到 1200 的时候，单客户端因为阻塞问题导致无法正常完成功能。但是多客户端架构依然在没有影响的情况下完成任务。

本文在对并发性能测试的同时，为了测试在资源消耗上的优势，对容器的 CPU 占用率和内存占用率也进行了检测。本文涉及的容器较多，本测试仅选取其中最有代表性的容器进行说明。对于单体应用架构，本文考虑检测两个容器，首先是以太坊 Geth 客户端运行的容器，其次是以太坊服务系统的服务端所运行的容器。对于本文设计的架构，考虑在 3 个 Geth 客户端提供服务的情况下，其中一个 Geth 客户端运行的容器，一个读业务组件运行的容器和一个写业务组件

运行的容器。如图 5.11 是 CPU 占用情况，5.12 为内存占用情况。

5.3 本章小结

本文首先介绍了本文基于反馈负载均衡策略的以太坊服务系统的主要功能，以及服务场景。随后利用 Java 后台性能测试工具，对比单体架构，评估了基于反馈负载策略的以太坊服务系统的性能。性能的评估角度从并发性压测维度出发，结果符合预期，验证了系统的性能提升以及稳定性。最后，该系统被应用于私募股权应用平台，验证了系统的实践性和应用性。

第六章 总结与展望

6.1 工作总结

随着区块链技术的发展，越来越多的区块链应用如雨后春笋般应运而生。由于区块链技术涉及相关技术点繁杂，其应用方式往往基于以太坊(Ethereum)等平台，而以太坊在应用过程中存在部署开发效率低、数据安全性保障低等问题。上海软件技术中心开发了基于以太坊的私募股权管理平台，其中以太坊服务模块存在效率低下，安全性低下的情况。且在项目开发和运维过程中存在开发效率低下，复用性较低的问题。对于以上问题，本文结合微服务思想设计和实现了以太坊服务系统，该系统提高了私募股权管理平台中区块链相关操作的效率和安全性，且提高了相关模块的运维和迁移效率。综上，本文研究内容主要包括以上部分：

首先，本文介绍了区块链应用的相关工作和国内外研究现状，总结了当前以太坊在对外提供服务时出现效率和扩展性低下的问题，无法满足在并发能力较高的情况。基于相关研究现状以及一般应用场景下的问题，本文设计和实现了一种高效稳定的以太坊服务系统。

其次，本文将以太坊客户端运行于 Docker 容器中，制作了相应的镜像，使得以太坊环境的迁移效率得到了极大的提升。本文结合微服务思想，设计和实现了以太坊服务提供者和服务调用者等子服务，并将该类子服务运行于 Docker 容器中。

然后，对运行在 Docker 中的以太坊客户端的调用，根据容器的状态设计了反馈负载均衡算法，使得系统整体的服务性能达到最高。

最后，详细充分的分析了私募股权平台应用场景下的业务特点，对与以太坊相关的业务做了横向的读写拆分并封装为相应的子服务。在该基础上结合反馈负载均衡算法设计了相应的业务调度模型。并最终实现了基于负载均衡算法的以太坊服务系统，并对其进行了功能和性能两方面的实验验证。

6.2 工作展望

本文的主要工作是研究以太坊在服务过程中出现的性能低下和数据相关操作安全性低下的问题，针对以太坊在容器中的运行状态设计了高效的负载均衡算法，并根据私募股权管理应用场景下的具体业务设计和实现了基于反馈负载均衡策略的以太坊服务系统。本文的系统设计基于微服务思想，采用基于 Java 的 Spring Cloud Alibaba 解决方案。本文设计的相关方案以及系统实现仅适用以太坊

在构建联盟链的应用场景中。为进一步提高该系统的服务效率，安全性等服务能力，可以考虑从系统架构以及负载均衡算法两个方面对系统进行优化：

微服务架构方面的改进意见：

首先，本文以太坊服务系统实现是基于 Spring Cloud Alibaba 解决方案，该方案中各子系统的实现基于 SpringBoot，而 Java 的运行需要 JVM 的支持，因此较为笨重，其开发效率与运行效率相对较低，可以考虑采用 Google 于 2009 年发布的 Golang 语言重构微服务系统中的各个子模块。Golang 是编译型语言，它更适合多处理器的分布式系统，且其协程特性使得并发性能较 Java 更好。

其次，本文微服务系统中的服务管理采用的方式是 Docker Compose，它在容器编排方面的灵活性不够。可以考虑引入 Kubernetes 提供容器的部署、编排和维护。Kubernetes 会自动去新建、监控和重启服务，管理员可以加载一个微服务，让规划器来找到合适的位置，同时，Kubernetes 也系统提升工具以及人性化方面，让用户能够方便的部署自己的应用。

最后，云计算时代已经到来，可以考虑将该服务系统进行上云处理，以实现真正的 BaaS(BlockChain As A Service)。

负载均衡算法优化：

本文主要考虑私募股权管理业务所面临的业务场景，因此，本负载均衡算法考虑容器的状态较少，换言之，本文所建立的容器状态向量空间的维度较低。因此，可以在深入研究以太坊客户端的运行特性和 Docker 容器的特性后，针对不同的应用场景设计更加高效的反馈负载均衡算法。随着云计算时代的到来，各种高效的负载均衡算法蓬勃发展，本文所设计的反馈负载均衡算法也应该与时俱进，且该领域潜力巨大，一定可以在本文所设计架构下提高以太坊服务能力。

参考文献

[1] Nadauld T D, Sensoy B A, Vorkink K, et al. The liquidity cost of private equity investments: Evidence from secondary market transactions[J]. Journal of Financial Economics, 2019, 132(3): 158-181.

[2] Antoni M, Maug E, Obernberger S. Private equity and human capital risk[J]. Journal of Financial Economics, 2019.

[3] Bernstein S, Lerner J, Sorensen M, et al. Private equity and industry performance[J]. Management Science, 2017, 63(4): 1198-1213.

[4] Faccio M, HSU H C. Politically connected private equity and employment[J]. The Journal of Finance, 2017, 72(2): 539-574.

[5] Ang A, Chen B, Goetzmann W N, et al. Estimating private equity returns from limited partner cash flows[J]. The Journal of Finance, 2018, 73(4): 1751-1783.

[6] Van Alstyne M W, Parker G G, Choudary S P. Pipelines, platforms, and the new rules of strategy[J]. Harvard business review, 2016, 94(4): 54-62.

[7] Bansraj D, Smit H T J, Volosovych V. Can Private Equity Funds Act as Strategic Buyers? Evidence from Buy-and-Build Strategies[R]. Working paper, www. ssrn. com, 2019.

[8] Nakamoto S. Bitcoin: a peer-to-peer electronic cash system, October 2008[J]. Cited on, 2019: 53.

[9] Zheng Z, Xie S, Dai H N, et al. Blockchain challenges and opportunities: A survey[J]. International Journal of Web and Grid Services, 2018, 14(4): 352-375.

[10] Cao C, Yan J, Li M X. How to Understand the Role of Trusted Third Party in the Process of Establishing Trust for E-Commerce?[J]. 2019.

[11] Sherman A T, Javani F, Zhang H, et al. On the origins and variations of blockchain technologies[J]. IEEE Security & Privacy, 2019, 17(1): 72-77.

[12] 魏生, 戴科冕. 基于区块链技术的私募股权众筹平台变革及展望[J]. 广东工业大学学报, 2019, 36(02): 37-46.

[13] 安立. 区块链在私募股权交易领域的应用[J]. 上海金融学院学报, 2017 (2017 年 02): 47-51.

[14] 邵奇峰, 金澈清, 张召, 等. 区块链技术: 架构及进展[J]. 计算机学报, 2018, 41(5): 969-988.

[15] Buterin V. A next-generation smart contract and decentralized application platform. White Paper, 2014.

[16] Namiot D, Sneps-Sneppe M. On micro-services architecture[J]. International

Journal of Open Information Technologies, 2014, 2(9): 24-27.

[17] Dragoni N, Giallorenzo S, Lafuente A L, et al. Microservices: yesterday, today, and tomorrow[M]//Present and ulterior software engineering. Springer, Cham, 2017: 195-216.

[18] Lawson J, Wolthius J. System and method for providing a micro-services communication platform: U.S. Patent 9,363,301[P]. 2016-6-7.

[19] Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables devops: Migration to a cloud-native architecture[J]. Ieee Software, 2016, 33(3): 42-52.

[20] Stubbs J, Moreira W, Dooley R. Distributed systems of microservices using docker and serfnode[C]//2015 7th International Workshop on Science Gateways. IEEE, 2015: 34-39.

[21] Amaral M, Polo J, Carrera D, et al. Performance evaluation of microservices architectures using containers[C]//2015 IEEE 14th International Symposium on Network Computing and Applications. IEEE, 2015: 27-34.

[22] Bui T. Analysis of docker security[J]. arXiv preprint arXiv:1501.02967, 2015.

[23] Anderson C. Docker [software engineering][J]. IEEE Software, 2015, 32(3): 102-c3.

[24] Peng H, Han W, Yao J, et al. The Realization of Load Balancing Algorithm in Cloud Computing[C]//Proceedings of the 2nd International Conference on Computer Science and Application Engineering. ACM, 2018: 140.

[25] Revah Y, Melman D, Mizrahi T, et al. Method and apparatus for load balancing in network switches: U.S. Patent 9,876,719[P]. 2018-1-23.

[26] Sharma M, Kini S, Tuli S A, et al. Adaptive load balancing for single active redundancy using EVPN designated forwarder election: U.S. Patent Application 10/050,809[P]. 2018-8-14.

[27] Arumugam M, Verzunov S, Kamath S, et al. Auto discovery and configuration of services in a load balancing appliance: U.S. Patent Application 10/101,981[P]. 2018-10-16.

[28] Kansal N J, Chana I. An empirical evaluation of energy-aware load balancing technique for cloud data center[J]. Cluster Computing, 2018, 21(2): 1311-1329.

[29] Soo W K, Ling T C, Maw A H, et al. Survey on load-balancing methods in 802.11 infrastructure mode wireless networks for improving quality of service[J]. ACM Computing Surveys (CSUR), 2018, 51(2): 34.

[30] Nofer M, Gomber P, Hinz O, et al. Blockchain[J]. Business & Information

Systems Engineering, 2017, 59(3): 183-187.

[31] Crosby M, Pattanayak P, Verma S, et al. Blockchain technology: Beyond bitcoin[J]. Applied Innovation, 2016, 2(6-10): 71.

[32] Mell P, Dray J, Shook J. Smart Contract Federated Identity Management without Third Party Authentication Services[J]. arXiv preprint arXiv:1906.11057, 2019.

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

[34] Leidner J L, Nugent T, Chadwick S. Systems and methods for smart contract intervention: U.S. Patent Application 16/133,932[P]. 2019-1-17.

[35] Mahajan A. A DApp for e-voting using Blockchain-enabled Smart Contracts[D]. , 2019.

[36] Cito J, Ferme V, Gall H C. Using docker containers to improve reproducibility in software and web engineering research[C]//International Conference on Web Engineering. Springer, Cham, 2016: 609-612.

[37] Rad B B, Bhatti H J, Ahmadi M. An introduction to docker and analysis of its performance[J]. International Journal of Computer Science and Network Security (IJCSNS), 2017, 17(3): 228.

[38] Chung M T, Quang-Hung N, Nguyen M T, et al. Using docker in high performance computing applications[C]//2016 IEEE Sixth International Conference on Communications and Electronics (ICCE). IEEE, 2016: 52-57.

[39] Leyi G O U, Qing C, Liang J, et al. Technical Research and Application Analysis of Microservice Architecture[J]. DEStech Transactions on Computer Science and Engineering, 2019 (iccis).

[40] Sharma S. Mastering Microservices with Java: Build Enterprise Microservices with Spring Boot 2.0, Spring Cloud, and Angular[M]. Packt Publishing Ltd, 2019.

攻读学位期间的研究成果

投稿论文

1. 面向私募股权的区块链服务系统设计，计算机应用与软件(第一作者).

申请专利

2. 基于以太坊的高性能区块链服务系统，2019112702245480(第二申请人)

致 谢

时光荏苒，岁月如梭，三年的研究生生涯转瞬即逝。回首过去的三年，感慨良多，收获也很丰厚，最幸运的是遇到一群优秀的良师益友，将这段经历书写成了最难忘的人生篇章。依然记得，2017 年在研究生复试面试上顾老师和蔼的询问是否愿意加入信息协同与计算实验室的大家庭，卢老师鼓励地表示“应该早点与我联系”。怀着对复旦研究生生活的向往和对老师们的知遇之恩的感激，我在暑假便开始了实验室的生活。实验室为我们配备了充足的设备，提供了舒适的工作和学习环境以及老师的指导和同学们的同窗陪伴，无论多久，都会是记忆里最美好的印记。

首先，我要感谢卢暾副教授，卢暾老师是我的导师。卢老师从研一开始便为我提供了阅读论文等学习技巧方面的指导，还在研二期间我的亲人因病住院期间为其捐款和鼓励我。卢暾老师对事业的严谨的态度和对学生温暖的关怀是将永远指引我前进的明灯。

其次，我要感谢顾宁教授对我的指导和帮助，所谓“高山仰止，景行行止”，顾宁老师是信息协同领域首屈一指的专家，其严谨的工作态度和求真务实的科研态度深深的影响着我。

我要感谢丁向华副教授和张鹏老师，丁老师在学术工作中严谨认真，求真务实，给我带来了深刻的影响。张鹏老师在我 2017 年入学时是实验室的在读博士生，在其博士毕业后接着在实验室开展科研工作，在论文选题和完成论文期间给予了我非常大的帮助。

特别地，我要感谢张绍华老师以及上海软件技术中心的各位领导。在研二期间，我在上海软件中心开展区块链相关科研工作，张绍华老师给予了生活上和工作上非常大的帮助。在毕业论文的选题和写作过程中，张老师给出了非常多的真知灼见，对我的整体工作起到了非常重要的指导作用。

最后，我要感谢实验室的同学们。在过去的三年中，无论是科研、技术和生活方面，你们的陪伴和鼓励都是我保持对生活的热情的最原始的动力。

无论人生的路通往何方，你们对我的帮助都是我最珍贵的宝藏，感谢你们！

复旦大学
学位论文独创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名：_____ 日期：_____

复旦大学
学位论文使用授权声明

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名：_____ 导师签名：_____ 日期：_____