|  |  |
| --- | --- |
| **学校代码** | **10699** |
| **分 类 号** | **TP315** |
| **密 级** |  |
| **学 号** | **2016201734** |

|  |  |
| --- | --- |
| **题目** | **面向舰船平台管理的** |
|  | **SOA技术研究** |

|  |  |
| --- | --- |
| **作者** | **索鹏** |

|  |  |
| --- | --- |
| **学科、 专业** | **计算机软件与理论** |
| **指 导 教 师** | **吴 健** |
| **申请学位日期** | **2019年1月** |

西 北 工 业 大 学

硕 士 学 位 论 文

（学位研究生）

|  |  |
| --- | --- |
| 题目： | 面向舰船平台管理的 |
|  | SOA技术研究 |

作 者： 索 鹏

学科专业： 计算机软件与理论

指导教师： 吴 键

2019年1月

**Title: Research on SOA Technologies in Shipborne Platform Management**

**By**

**Suo Peng**

**Under the Supervision of Professor**

Wu Jian

A Dissertation Submitted to

Northwestern Polytechnical University

In partial fulfillment of the requirement

For the degree of

Master of Computer Software and Theory

Xi’an P. R. China

January 2019

摘 要

舰船平台管理系统采用计算机控制技术、网络技术等对舰船平台的主要系统如推进系统、电力系统、损管系统和辅机系统等实施集中监视、控制和管理。该系统要求将现代舰艇的各类运算操作、基础数据集成到一个统一的、公共计算环境中，实施分布或集中式管理。舰船平台上集成的应用应能根据使用过程中的反馈进行快速的更新。

面向服务架构(SOA)是一种构建分布式应用的软件设计风格。服务是可进行远程调用、具有完整的业务逻辑并独立运行的软件组件。客户端使用标准数据格式和协议与服务进行交互，从而使服务开发实现技术独立。微服务架构是将持续交付的软件开发理念和容器技术等新思想注入到SOA中形成的一种架构风格，采用微服务架构，可以灵活的由现有的服务组合成新的应用，并且满足现代舰船应用系统快速迭代开发和高可用等需求。

本文研究面向舰船平台管理软件服务集成技术，并提出基于微服务架构的舰船平台服务管理系统。采用微服务架构可以保证平台的可靠性，即系统中一个模块（即一个微服务）的故障不会影响到整个系统的使用，而故障模块由于粒度小，从而可以对其进行快速的调试，维护和部署，使故障模块对系统的影响降到最低。此外通过将关键模块进行多实例部署，可以实现系统整体的高可用性。而微服务的快速部署特点使系统中的模块或应用服务根据负载情况进行动态扩展，使得系统具有更大的弹性。

论文首先介绍了微服务架构的相关概念和技术，然后对面向舰船平台管理的软件集成系统进行需求分析，使用UML对各个模块及模块间的通信过程进行建模，最终设计并实现了基于微服务架构的舰船平台服务管理系统中核心模块的技术方案。舰船平台服务管理系统为各个子系统微服务提供服务发现功能，为客户端屏蔽了服务实例的物理地址；并提供熔断机制，在服务调用出现故障或性能问题时由系统接管并直接返回错误信息，防止客户端由于异常服务调用而阻塞；系统还提供安全认证和权限管理，对客户端进行认证并审核服务调用权限。最后，论文通过模拟一个应用场景对设计的舰船平台服务管理系统原型进行实验验证，结果表明本文所设计与实现的系统合理、可行。

关键词：面向服务架构，微服务架构，舰船平台服务管理系统

**Abstract**

Shipborne platform management system uses computer control technologies and network technologies to monitor, control and manage major systems on the shipborne platform such as propulsion system, power supply system, damage control system and auxiliary equipment system. It demands the consolidation of various computing operations and fundamental data sets into a single and public computing environment to apply centralized or distributed management. Applications integrated through the shipborne platform should be updated timely in respond to feedback from daily use.

Service-Oriented Architecture (SOA) is a style of software design for building distributed applications. A service is a software component that can be remotely invoked, has self-contained business logic and runs independently. Clients use standard data format and protocol to interact with services, as a result, service development can be made technically neutral. Microservice architecture is an architectural style that combines SOA with continuous delivery software development process and new technologies such as container technology, by adopting microservice architecture, existing services can be combined flexibly to form new applications and modern shipborne applications’ requirements of fast iterative development and high availability can be met.

This paper studies service integration technologies in shipborne platform management software and suggests a design of a shipborne service management system based on microservice architecture. By adopting the microservice architecture, the resulting system is resilient, which means the faulting of one module (microservice) will not affect the whole system, and due to its small granularity, the faulting module can be debugged, maintained and deployed more quickly such that the negative effect of the faulting module is brought to the minimum. Besides, the system can be made highly available by deploying multiple instances of the core modules. And the fast deployment of microservices means that modules and services in the platform can be scaled dynamically according to their loads, so the system is more elastic.

This paper first introduces relevant concepts and technologies in SOA and microservice architecture, and then it analyzes the requirements of the shipborne service management system, using UML to create models for the modules and the communication processes among them. Finally, it designs and implements the technical proposal of the core modules of the shipborne service management system based on microservices architecture. The proposed shipborne service management system provides service discovery function to each microservice in the subsystems, abstracting away the physical locations of service instances for invoking clients; it also provides circuit breaker mechanism, which means the platform will take over and return error messages when the service is broken or when its performance suffers, so that client will not block when invoking errant services; the platform also provides authentication and authorization services, identifying the clients and checking their credentials. In the end, this paper verifies the proposed shipborne service management system by simulating a real usage scenario, and the result shows that the system is reasonable and its design is feasible.

**Key words**：SOA, microservice architecture, shipborne service management system

目 录

[摘 要 I](#_Toc534580226)

[**Abstract** II](#_Toc534580227)

[1 绪论 1](#_Toc534580229)

[1.1 研究背景与意义 1](#_Toc534580230)

[1.2 国内外研究现状及发展趋势 2](#_Toc534580231)

[1.3 论文主要工作及贡献 3](#_Toc534580232)

[1.4 论文组织结构 4](#_Toc534580233)

[1.5 本章小结 5](#_Toc534580234)

[2 相关概念与关键技术概述 7](#_Toc534580235)

[2.1 微服务架构 7](#_Toc534580236)

[2.2.1 微服务和单体架构对比 7](#_Toc534580237)

[2.2.2 微服务和传统SOA对比 7](#_Toc534580238)

[2.2.3 微服务架构特点 8](#_Toc534580239)

[2.2 Spring Boot 8](#_Toc534580240)

[2.3 REST 9](#_Toc534580241)

[2.4 Docker容器技术 10](#_Toc534580242)

[2.4.1 Docker简介 10](#_Toc534580243)

[2.4.2 Docker和虚拟机的区别 10](#_Toc534580244)

[2.4.3 Docker和微服务架构 11](#_Toc534580245)

[2.5 Hystrix熔断技术 11](#_Toc534580246)

[2.5.1 Hystrix简介 11](#_Toc534580247)

[2.5.2 Hystrix线程隔离 12](#_Toc534580248)

[2.5.3 Hystrix熔断 12](#_Toc534580249)

[2.5.3 Hystrix降级处理 13](#_Toc534580250)

[2.6 微服务安全技术 13](#_Toc534580251)

[2.6.1 OAuth 2.0认证协议 13](#_Toc534580252)

[2.6.2 JSON Web Token 15](#_Toc534580253)

[2.7 本章小结 16](#_Toc534580254)

[3 舰船平台服务管理系统的需求分析 17](#_Toc534580255)

[3.1 微服务系统面临的挑战 17](#_Toc534580256)

[3.2 舰船平台服务管理的系统目标 18](#_Toc534580257)

[3.3 舰船平台服务管理系统的功能性需求分析 18](#_Toc534580258)

[3.3.1 外部工具链 18](#_Toc534580259)

[3.3.2 服务集成 20](#_Toc534580260)

[3.3.3 基础服务 24](#_Toc534580261)

[3.4 舰船平台服务管理系统的非功能性需求分析 25](#_Toc534580262)

[3.5 本章小结 26](#_Toc534580263)

[4 舰船平台服务管理系统的概要设计 27](#_Toc534580264)

[4.1 舰船平台服务治理方案 27](#_Toc534580265)

[4.2 模块设计 29](#_Toc534580266)

[4.2.1 API网关核心模块 29](#_Toc534580267)

[4.2.2 负载均衡模块 30](#_Toc534580268)

[4.2.3 熔断器 30](#_Toc534580269)

[4.2.4 服务注册中心 30](#_Toc534580270)

[4.2.5 授权认证模块 30](#_Toc534580271)

[4.2.6 监控模块 31](#_Toc534580272)

[4.2.7 管理模块 31](#_Toc534580273)

[4.2.8 微服务 31](#_Toc534580274)

[4.2.9 数据库 31](#_Toc534580275)

[4.3 数据库设计 31](#_Toc534580276)

[4.3.1 实体-联系模型设计 31](#_Toc534580277)

[4.3.2 数据库模式设计 32](#_Toc534580278)

[4.4 接口设计 35](#_Toc534580279)

[4.4.1 API网关核心模块 35](#_Toc534580280)

[4.4.2 负载均衡模块 36](#_Toc534580281)

[4.4.3 熔断器 36](#_Toc534580282)

[4.4.4 服务注册中心 37](#_Toc534580283)

[4.4.5 授权认证模块 37](#_Toc534580284)

[4.4 系统部署方案设计 38](#_Toc534580285)

[4.5 本章小结 39](#_Toc534580286)

[5 面向舰船平台管理系统的详细设计与实现 41](#_Toc534580287)

[5.1 API网关的设计与实现 41](#_Toc534580288)

[5.1.1 API网关的架构 41](#_Toc534580289)

[5.1.2 API网关的核心类设计 42](#_Toc534580290)

[5.1.3 API网关过滤器实现 44](#_Toc534580291)

[5.2 负载均衡模块的设计与实现 45](#_Toc534580292)

[5.2.1 负载均衡模块的架构 46](#_Toc534580293)

[5.2.2 负载均衡模块的核心类设计 47](#_Toc534580294)

[5.2.3 负载均衡算法实现 48](#_Toc534580295)

[5.3授权认证模块的设计与实现 50](#_Toc534580296)

[5.3.1 授权认证模块的架构 50](#_Toc534580297)

[5.3.2 授权模块 50](#_Toc534580298)

[5.3.3 认证模块 51](#_Toc534580299)

[5.4 本章小结 54](#_Toc534580300)

[6 面向舰船平台管理框架的测试与验证 55](#_Toc534580301)

[6.1应用场景仿真 55](#_Toc534580302)

[6.2验证环境 56](#_Toc534580303)

[6.3验证原理和验证方法 57](#_Toc534580304)

[6.4验证流程及结论 57](#_Toc534580305)

[6.6本章小结 63](#_Toc534580306)

[7 总结与展望 65](#_Toc534580307)

[7.1 论文总结 65](#_Toc534580308)

[7.2 下一步工作展望 65](#_Toc534580309)

[参考文献 67](#_Toc534580310)

[致 谢 70](#_Toc534580311)

[攻读硕士学位期间发表的学术论文和参加科研情况 71](#_Toc534580312)

1 绪论

本章首先介绍了面向舰船平台管理系统的研究背景和意义，通过分布式技术的演变过程、面临的问题以及解决的手段，确立了采用SOA技术实现舰船平台管理是当前的主流趋势。通过讨论近年来微服务架构流行的原因和该架构本身所具有的优势，提出使用微服务架构作为舰船平台管理服务化的实现手段。通过分析国内外企业微服务架构的成功使用案例，明确微服务架构中所需确立的服务治理方案，提出了微服务框架的主要特点。最后对论文的工作和结构进行了简单的介绍，使读者对论文的研究内容有一个初步的了解。

1.1 研究背景与意义

舰船系统的信息化、自动化和智能化管理是未来重要发展趋势之一。随着信息技术的发展和舰船软件系统和设备的日益复杂化，对舰船上的各个分系统进行统一管理和信息集成的需求也越来越明显。舰船平台管理系统就是利用计算机及网络技术将主推进系统、损管系统、电力系统、辅机系统、监控系统、三方系统、保障系统以及各类信息系统等相对独立的子系统连接起来，实现信息集成，从而对全舰实现以信息为中心的监测与管控[1]。

通过对舰船内部各个子系统的集中式的管理，实现各个系统应用数据的共享，从而可以实现通过分析全舰的状态迅速做出关键决策，实现最优的分布式实时性智能控制；通过消除舰船中存在的互相独立的系统形成的“信息孤岛”，促进各个系统的协同合作，强化了对舰船的综合管理能力，同时还能实现节能减员，从而产生巨大的经济效益；通过提供平台管理标准化接口和完整的治理规范，加快了新系统及应用的开发和上线速度；通过将集成的各种状态信息通过卫星传输至远程监控指挥系统，有助于实现船岸一体化发展[2]。

相比民用船舶，舰船的各个系统在操作性、实时性、精确性和维护性等方面都具有更加严格的要求。目前，国外新型研制的舰船都已经装备上了平台管理系统来提高舰船的信息集成和自动化程度，例如美国下一代大型驱逐舰DDX就采用了平台管理系统，使得具备巡洋舰规模的该舰上的舰员只有100人左右，极大的提高了人员利用率，此外，英国的45型驱逐舰、德国的F124护卫舰、法国和意大利联合研制的“地平线”级驱逐舰、韩国的KDX-2等舰船上都已采用类似的平台管理系统[3]。虽然我国第一艘航空母舰“辽宁号”已经下水服役，但与世界海军强国相比，特别是在平台管理系统软件方面，仍存在较大差距。而国内舰船的各个子系统相对独立，并采用不同的技术标准、开发语言和体系架构进行设计和开发，这给系统的集成和管理带来了更大的挑战[4]。因此，有必要对面向舰船平台管理的相关技术进行深入研究，对平台管理系统进行概要设计，掌握舰船平台管理系统的设计方法和关键技术，加快我国舰船信息化建设进程，减少对国外技术的依赖的同时满足国防建设的需要。

1.2 国内外研究现状及发展趋势

对舰船平台管理系统的研究可以发现，舰船平台管理系统所采用的技术手段与同时代计算机技术的发展有着紧密的联系，都由最初的单体架构朝着分布式架构演变，这是因为单体架构庞大、不易理解从而不易维护、系统下线成本高、线上又难以管理，并且各个单体架构的应用系统相互孤立。于是分布式技术一出现就得到舰船软件开发和研究人员的极大关注，而分布式技术又由早期的远程过程调用(RPC, Remote Procedure Call)逐渐演化到现在的面向服务架构（SOA，Service Oriented Architecture）。RPC将发生在一个进程内部的过程调用方法扩展到分布式系统中，使得客户端进程可以调用远程服务器进程的函数，该技术也由此得名。随着面向对象技术的兴起，远程方法调用（RMI，Remote Method Invocation）将面向对象的开发模型应用到RPC中，从而实现了不同机器不同进程中对象的相互调用。面向对象的思想和分布式技术的结合产生了分布式对象技术，具有典型代表的是Java RMI和CORBA，但分布式对象技术在具体的使用过程中出现了很多的令人头疼的问题，例如对象的接口描述没有缺乏对依赖进行描述的方法，分布式对象中间件的编程模型过于底层导致学习以及开发的难度大，以及编程模型没有分离出安全、失效处理以及并发等分布式系统中常见的公共功能，这些都给分布式对象编程带来了极大的困难。为了解决这些问题，分布式组件技术应运而生，其中最具代表性的是微软公司提出的DCOM模型、Sun公司开发的EJB模型以及CORBA的组件模型CCM（CORBA Component Model），组件通过合约描述了其提供和依赖的接口，不同的组件可以根据接口的相互兼容性进行灵活的组合，通过使用现成的组件可以快速组合出需要的应用。这种特性使得国外的舰船平台逐渐开始采用组件技术进行开发，国内也出现了基于分布式组件技术的舰船平台管理系统的研究[5]。但上述各个组件模型在舰船平台管理系统中的应用依然存在问题，其中DCOM组件存在二进制版本兼容问题[6]；EJB与Java面向对象编程语言紧耦合，这导致与非Java语言开发的组件进行组装时会存在问题；而CCM虽然不依赖于具体的编程语言，但其特性过于复杂，增加了开发难度和开发人员的学习成本。

SOA的出现解决了分布式构件技术的种种问题，通过将系统进行垂直切分为小粒度且松耦合的服务，可以实现类似分布式构件的灵活性；通过标准化通信协议，SOA很好地解决了服务通信的兼容性；通过完整的服务治理规范，降低了开发难度。美国海军提出的“全舰计算环境(TSCE, Total Ship Computing Environment)”的概念中，提出利用企业服务总线(ESB, Enterprise Service Bus)实现舰载应用系统的集成[7]。然而，使用ESB来实现SOA，仍存在不少问题。首先，ESB会成为单点故障，而ESB自身作为一个庞大的单体组件和单体架构软件一样具有不少的问题，同时，ESB作为重量级中间件，对其进行功能扩展需要复杂的设置，而各个ESB没有公共标准，各个商业化平台互相不兼容，这无疑增加用户的学习成本[8]。

近年来，作为SOA的一种实现和部署方案，微服务架构（MSA，Microservice Architecture）去除了传统SOA所使用的ESB，其采用分布式服务治理框架，基础设施自动化以及轻量的通信协议，并融合领域驱动设计、持续交付、小型自治团队等软件工程的思想[9]。特别是当下虚拟化技术的逐渐成熟，如何更好的利用虚拟化平台从而增加对硬件资源的利用率成为热点研究问题，而微服务架构以其动态、灵活、自动化的特性作为云平台上的最佳架构模式。而国内外也涌现出许多优秀的微服务框架，目前国内以阿里巴巴的开源框架Dubbo影响力最大，其主要目标是实现高性能和透明的远程服务调用，并实现分布式服务治理方案[10]。国外也有微服务框架的成功应用案例，比如Netflix公司通过采用微服务架构，实现了系统的高可用性，应用的快速迭代开发和部署能力，提高了服务的复用性，降低了服务的响应时间，其他开发团队也有类似的经验[11][12]。国内外优秀的微服务框架具有共同的特点，其中包括提供自动化的服务注册与发现机制，轻量的通信协议支持以及灵活的路由配置，并同时提供在服务器间实现负载均衡的能力，框架还提供自动化部署途径以及对服务进行监控、组合、降级等服务治理能力[13]。

TSCE中提出了对舰船上的计算、存储、网络等硬件资源进行虚拟化，从而实现了基础设施即服务（IaaS, Infrastructure as a Service）[7]。而微服务通过容器技术，可以快速的部署到IaaS中，从而可以充分的利用IaaS提供的基础设施[14]。采用微服务架构实现舰船平台管理是未来海军平台化建设的发展方向。

国内对于面向舰船平台管理的技术的研究仍旧处于理论性探索和初步实践的阶段，虽然目前有一些高校和研究所在开展相关研究与开发工作，但研究的主要内容主要还是围绕如何利用构件技术来实现舰船平台管理[5]，这与国外仍存在较大差距。本文借鉴TSCE中有关采用SOA技术来实现面向舰船平台管理的思想，并进一步提出采用微服务来实现舰船平台管理，给出了一套舰船平台服务管理系统的设计方案，希望能对我国舰船平台管理的发展做出贡献。

1.3 论文主要工作及贡献

本课题来自于实际横向合作项目，对面向舰船平台管理的SOA技术进行了研究，特别是如何通过系统化方法来对舰船平台服务应用进行统一管理。通过跟踪舰船基础设施和平台管理的发展方向，本文提出采用微服务架构来实现舰船平台管理。通过对微服务架构中采用的相关SOA技术的研究，进一步研究满足面向舰船平台管理中服务注册发现、服务安全、服务生命周期管理和服务监控需求的舰船平台服务管理系统。

在设计该系统的过程中，本文参考了现有优秀微服务框架的特点和舰船平台管理的业务需求，采用软件工程的研究手段对舰船平台服务管理系统进行了需求分析，给出了功能设计方案和概要设计方案，并对核心模块给出了详细的设计和实现方案。该方案使用REST API接口规范，采用RPC调用方式来实现高性能的服务集成。舰船平台服务管理系统与Spring Boot实现无缝集成，同时保证开发编码的低侵入性，并实现服务的灵活可配置和服务运行状态的可监控。系统通过日志记录实现对服务调用时间、次数、性能等数据的统计与追踪。系统还实现负载均衡，路由映射，身份认证与权限控制等微服务管理中的核心机制，同时本文还给出了舰船平台服务管理的非功能性需求。

本文按照软件工程的基本方法，使用UML用例图进行需求分析、使用流程图描述系统业务流程、使用架构图呈现系统的各个模块、使用类图和时序图描述系统的详细设计。在使用各种图形对系统的流程和模块进行建模的同时，还对系统功能的设计与实现进行了详细的描述。

1.4 论文组织结构

本文通过对面向舰船管理SOA技术的研究，特别是近期微服务架构中的SOA技术研究，设计与实现基于微服务架构的舰船平台管理系统。本文从介绍项目背景和国内外研究现状开始，论述了开发服务化舰船平台管理系统的迫切需求；通过对各种相关技术的分析，未实现系统的需求分析和概要设计打下基础。接下来论文对舰船平台管理系统进行了从需求分析、概要设计、详细设计到系统验证的流程，最终实现系统的可行性研究。

本文一共有七章，具体结构如下：

第一章，绪论。本章主要介绍论文的背景、意义、国内外的发展趋势和技术变革对舰船平台管理系统带来的影响，最后介绍了论文的组织结构。

第二章，介绍舰船平台服务管理系统中出现的关键概念和使用的部分技术。首先对微服务架构进行介绍，然后介绍了微服务开发框架Spring Boot，微服务架构思想REST，微服务部署方案Docker容器，Hystrix微服务熔断技术和微服务安全标准Auth 2.0等微服务主要技术和框架。

第三章，对舰船平台服务管理系统进行需求分析。通过对采用微服务进行系统管理所面临的挑战进行分析，结合舰船平台管理的特殊性，挖掘系统的功能性和非功能性需求，并最终完成系统功能模块的划分。

第四章，对舰船平台服务管理系统进行概要设计。通过对总体架构设计、模块设计、接口设计和数据库设计的描述完成对系统的微服务架构实施方案的介绍。

第五章，对舰船平台服务管理系统的核心模块进行详细设计与实现。主要描述了API网关、负载均衡、和认证模块的详细设计与实现。

第六章，对舰船平台服务管理系统进行实验验证。通过模拟应用场景对系统进行测试与验证，证明系统的可行性和有效性。

第七章，总结与展望。对本文的研究进行了总结，并对下一步的工作进行安排，同时对未来的研究方向进行了展望。

1.5 本章小结

本章首先介绍了研究背景和意义，然后通过描述国内外舰船平台管理系统的发展和技术演变，结合当下面向服务架构的实现技术，提出了采用微服务架构实现舰船平台管理这一创新性理念。最后本章对论文的主要工作和组织结构进行了介绍。

2 相关概念与关键技术概述

2.1 微服务架构

采用微服务架构实现软件系统在2011年由一个软件架构小组首次提出，用于描述它们使用的一种实现SOA的架构机制[15]。微服务架构从被当时的被质疑逐渐被人们所接受，当前被业界普遍接受的关于微服务架构的观念是由Martin Fowler提出的：微服务是将软件划分为一组单一功能的、可以独立开发部署运行的服务，服务运行于相互独立的进程之中，服务之间采用轻量的方式进行通信[16]。对于大型的平台化应用，把具有相同功能的冗余代码提取成独立的模块，以微服务的形式提供其他应用进行调用，通过将单体应用进行划分，形成一个由微服务组成的公共基础服务库，每个服务拥有自己的文档和维护团队，增加了系统组件的可靠性和系统开发的灵活性[17]。

2.2.1 微服务和单体架构对比

传统的单体架构根据系统中模块间职责的不同分为表示层、业务逻辑层和数据访问层[18]。该架构的优势为系统容易进行开发和设计，并且由于采用该架构开发应用的历史已久，有大量成熟的框架可以使用，系统的人工部署也相对容易，开发人员只需将应用打包并复制到应用服务器上运行即可。

随着业务的不断增加，单体架构的缺陷逐渐体现出来。首先是其扩展能力差，因为采用单体架构设计的系统中的功能模块相互间高度耦合，当需要对相关功能进行扩展的时候，会引入大量的相关依赖。另外，在单体架构下对一个模块进行修改，意味着要对整个应用进行重新的编译、测试和部署，大大增加了开发周期。单体架构中的一个模块成为瓶颈，如果要采用多实例部署的方案来提高性能，则需要对整个应用进行水平复制，这极大的浪费了硬件资源，增加了运营成本。

微服务就是为了解决单体架构的问题，通过将单体应用拆分成服务，每个服务可以独立扩展和修改。当系统性能出现问题时，只需将个别微服务进行多实例部署即可。这种水平扩展能力不仅通过合理的解决系统瓶颈降低了系统维护成本，而且由于微服务具有灵活部署能力，可以实现运行时的动态扩展，减少了人工干预和可能出现的错误，从而增加了系统的稳定性。

2.2.2 微服务和传统SOA对比

SOA时将系统按照不同功能拆分成相互独立的服务单元，并通过具有良好定义的接口和标准通用的通信协议对服务进行组合来构建软件系统的一种软件组件模型[19]。从定义上看，传统SOA和微服务在思想上是一致的，作为SOA的一种实现方法，微服务架构思想将敏捷开发、自动化部署和云计算思想融入传统SOA，使其呈现自身的特性和侧重点[20]。

微服务摒弃了在传统SOA中居核心地位的企业服务总线，使得服务更加独立灵活；微服务推荐使用REST形式的接口来替代Web服务的SOAP协议，使用轻量级的JSON作为推荐的数据传输格式；微服务的粒度更小，使得开发周期短，利于服务的部署和横向扩展[21]。

2.2.3 微服务架构特点

每个微服务具有单一的职责，作为基本模块通过组合形成应用。由于使用标准化接口进行交互，且不存在中心化的大型中间件的依赖，每个服务可以采用不同的技术手段进行开发。微服务具有自治性，被独立的开发、部署、运行和维护。服务间的通信通常采用HTTP + JSON或AMQP等轻量级协议。微服务通常实现一个小而完整的业务功能，从而一个服务开发团队只由几个人组成，并且由开发团队负责运维。微服务之间相互解耦，增加了应用开发的灵活性并限制了模块出错的传播范围。故而微服务架构已经成了当代构建分布式系统的发展方向[22]。

2.2 Spring Boot

Spring框架通过依赖注入、切面编程和模板技术将J2EE开发者从EJB复杂的开发模型中解救出来。通过依赖注入，开发者不用管理对象的创建以及对象间的依赖关系；并通过面向接口编程，降低了组件之间的耦合度，增加了程序的灵活性和可扩展性；通过切面，Spring支持将开发过程中的公共服务提取出来，以注解的方式应用到各个需要的业务逻辑中，增加了代码的复用程度；通过模板技术，Spring封装了开发过程中与业务逻辑无关的工作，如获取数据库连接和进行异常处理等，从而增加了代码的清晰度，降低了程序员的工作量。

Spring Boot是Spring社区对Spring框架进行的进一步精简和优化，具体而言，使用Spring Boot构建微服务，具有以下好处[23]：

1. 提供starter依赖

在开发一个Spring项目的过程中，项目需要依赖很多第三方库，例如日志库、模板库、序列化库等。通常使用Maven或Gradle将一个项目的依赖进行统一管理，尽管如此，当项目的依赖变得愈来愈多时，依赖的维护、项目依赖关系的分析以及依赖之间的兼容性都会给项目的开发和维护带来很大的困难。Spring Boot按功能领域将开发某一功能所需要的依赖统一起来，并以一个总的starter依赖项进行发布，开发者在开发过程中只需引入一个依赖项就可以传递获取互相兼容的一组依赖。Starter依赖项增加了项目依赖结构的清晰度，降低了依赖维护的难度，解决了依赖间的兼容性问题。

1. 提供自动配置功能

通过采用“惯例优于配置”的思想，Spring Boot为Spring框架的许多配置项提供了默认选项。这样一来，通过少量的配置就能完成基于Spring框架的服务开发。Spring Boot会根据类路径上配置的依赖包进行扫描并自动配置各种bean，而不需要开发者进行显式的bean声明。Spring Boot在提供默认配置的同时也允许开发者进行显式的配置，在最大化简化开发工作量的同时保留了Spring框架的灵活度。

1. 提供命令行界面

Spring Boot提供了一个可以执行Groovy脚本的命令行，Groovy采用类似Java的语法，但对Java语法进行了精简，使开发一个原型程序所需的代码量大大的缩小。使用Groovy和Spring Boot命令行，允许开发者用极少量的代码实现一个微服务，由于Groovy是解释型语言，使用者可以在命令行上对自己的想法进行快速的验证。Spring Boot通过命令行界面提供了快速迭代和原型开发的能力。

1. Spring Boot Actuator

Spring Boot Actuator为由Spring Boot开发的微服务增加了许多REST API接口。通过这些接口客户端可以访问到一切有关微服务的信息，包括服务中运行的bean、服务的配置属性和环境变量、服务中每一个线程的函数调用栈、服务的健康情况、服务调用的次数统计和服务调用历史纪录等。可以说使用Spring Boot开发微服务，直接解决了服务监控这一大难题。

1. 集成运行容器

不同于传统的Spring Web应用，Spring Boot可以将开发的微服务打包成jar包，其中包含服务运行所需要的容器，不像war包需要部署到Java Servlet容器中，只需要运行环境支持JVM即可。这样做简化并统一了部署环境，减少了对Servlet容器的依赖，符合微服务动态扩展和自动化部署的要求。

正是上述Spring Boot的这些特点，使得其成为一款优秀的微服务开发框架。

2.3 REST

2000年Roy Fielding在其博士论文中首次提出REST（Representational State Transfer,表述性状态转移）的概念[24]。REST包含一组架构原则和约束规范，它强调以资源为中心，将网络上可访问的所有对象都抽象成资源，整个Web被视为资源的一个集合，其中的每个资源由唯一的标识符进行标识。满足REST规范的服务就被称作RESTful服务。RESTful服务提供规范的接口对资源进行操作，对资源的访问通常是无状态、可缓存的。微服务由于要进行多实例部署，服务通常也应该使无状态的，因为这样可以最大化地实现微服务灵活的运行模式，使用REST规范设计和开发的微服务，更加符合微服务架构提出来的初衷[22]。

在当前主流的对Web服务通信进行实现的方案中，基于REST原则架构的Web服务于传统的采用SOAP消息和使用XML实现RPC对比起来在结构上更加简洁，故而越来越多的公司开始采用REST风格来实现Web服务，微服务的核心思想与REST原则所传达的思想不谋而合，都强调轻量级、规范、易于操作的通信协议和接口规范。根据亚马逊建设服务构架的过程中总结的经验，RESTful Web服务的性能优于基于SOAP的Web服务[25]。实际上，REST的出现对Web服务从接口到通信消息的设计的影响都非常大，其模型的易理解、接口的简洁性对于设计和开发者而言来说使用非常方便，REST基本已经取代了传统Web服务中基于SOAP和WSDL的接口设计。

论文提出的舰船平台服务管理系统采用REST风格实现，部署在该平台上的微服务的接口设计满足RESTful API的设计规范，即使用URL路径来标识资源，用HTTP方法来表明对资源进行的操作，RESTful API的范例如表2-1。

表2-1 REST API范例

|  |  |  |
| --- | --- | --- |
| URL | HTTP方法 | 接口描述 |
| /attributes | GET | 查看所有属性 |
| /attribute | POST | 创建新属性 |
| /attribute/{id} | GET | 查看ID为id的属性 |
| /attribute/{id} | DELETE | 删除ID为id的属性 |
| /attribute/{id} | PATCH | 更新ID为id的属性 |

2.4 Docker容器技术

2.4.1 Docker简介

Docker是使用容器来开发、部署和运行应用的平台。容器包含运行应用所需要的环境，包括代码、运行时环境、环境变量和配置文件等。容器间相互独立并且包含独立的网络、内存和文件子系统。容器引擎负责对容器进行管理，容器引擎通过扩展操作系统内核，使在同一引擎上运行的容器共享同一个操作系统，从而允许在一个操作系统上运行几百个容器的同时具有接近运行在操作系统上的性能，故Docker被称为轻量级容器。

Docker主要有两种用途：持续集成和持续部署[26]。为了部署一个应用，首先在Dockerfile中使用各种命令来生成镜像，该镜像作为生成Docker容器的模板被推送到本

地的注册中心。Docker镜像通过tar文件的形式存在，故而部署一个使用Docker打包的应用只需复制一个文件到目标机器上即可，故Docker成为最佳的自动化部署方案。Docker镜像使用分层的设计，每次容器数据发生改变需要持久化时，Docker只保存变化的部分，这样极大地降低了镜像的大小，从而使得Docker镜像可以快速的复制和部署。Docker平台提供了一组API用来管理容器镜像和容器的执行，而Docker注册中心和DockerHub提供镜像的复用，使我们能在任何地方快速部署服务。

2.4.2 Docker和虚拟机的区别

图2-1显示了Docker和虚拟机在架构上的区别[27]。在虚拟机中，虚拟机监控程序运行在宿主操作系统之上，而虚拟机操作系统则运行在虚拟机监控程序上，由监控程序负责管理。而Docker使用容器技术，使得不同的容器直接运行在宿主操作系统上，由Docker守护进程进行管理。

（a）虚拟机 （b）Docker

图2-1 Docker与虚拟机的架构区别

作为虚拟化技术，容器比虚拟机更轻量，消耗的系统资源更少。虚拟机通过在硬件上建立抽象来实现虚拟化，而容器中的应用通过相应的容器进程拥有自己的进程ID空间，文件系统结构和网络接口，从而实现了对内核的虚拟化。由于虚拟机包含整个操作系统环境，而该环境中的许多组件对于运行应用来说是多余的，故而虚拟机浪费了很多硬件资源，而容器只包含支持运行应用的环境，因而运行容器的开销更小。通常运行在监控程序上的虚拟机占用主机10%-15%的资源，而Docker只使用极少量的资源[28]。有研究表明Docker容器的运行速度比虚拟机平均快26倍[28]。监控程序的开销随着在其之上运行的虚拟机的增加成指数倍增长，这使得在同一台机器上可以运行的Docker容器数远超过虚拟机[29]。Docker容器本身不存在启动过程，所以容器内的应用进程可以实现瞬间启动。这一特点对于需要随时启停进行动态扩展的微服务来说也是至关重要的。

2.4.3 Docker和微服务架构

由此可见，Docker容器和微服务是天生的一对，Docker容器的低开销和快速部署的能力使其特别适合实现微服务架构，Docker容器具有的自动化构建和部署工具从而可以很好的适应微服务持续部署思想，Docker将应用与运行环境一起打包的思想进一步简化了服务的部署工作，使得部署服务不再需要先对运行环境进行安装和配置，做到了开发阶段和部署阶段的无缝融合。Docker容器和微服务共同提升系统的模块化，促进代码复用，实现运行过程可重现，以及实现了细粒度的分布式系统的扩展[30]。

2.5 Hystrix熔断技术

2.5.1 Hystrix简介

Hystrix是由Netflix进行设计和研发的用来实现Netflix云平台上系统的快速恢复的开源框架。在分布式系统中，特别是在微服务架构中，应用要调用很多外部服务接口来完成业务逻辑功能，这导致如果外部接口出现响应缓慢等问题会直接级联影响到多个应用和服务, 从而最终导致整个系统的崩溃。对外部的模块依赖的越多，这种情况出现的风险就越高[31]。Hystrix就是通过线程池隔离、熔断和降级这三种手段来应对服务接口响应缓慢对系统带来的风险。

2.5.2 Hystrix线程隔离

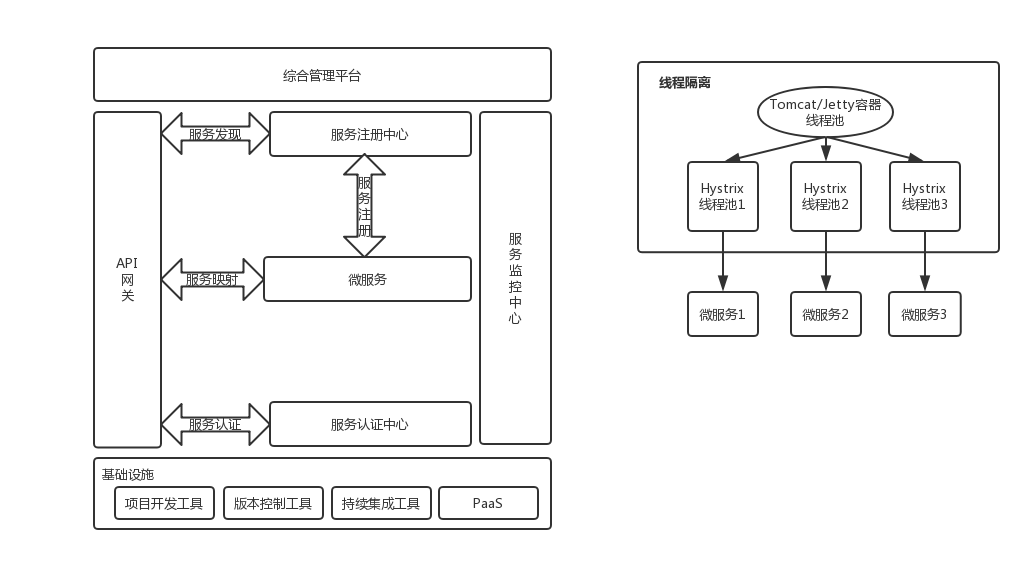
 Hystrix线程隔离的机制如图2-2所示。

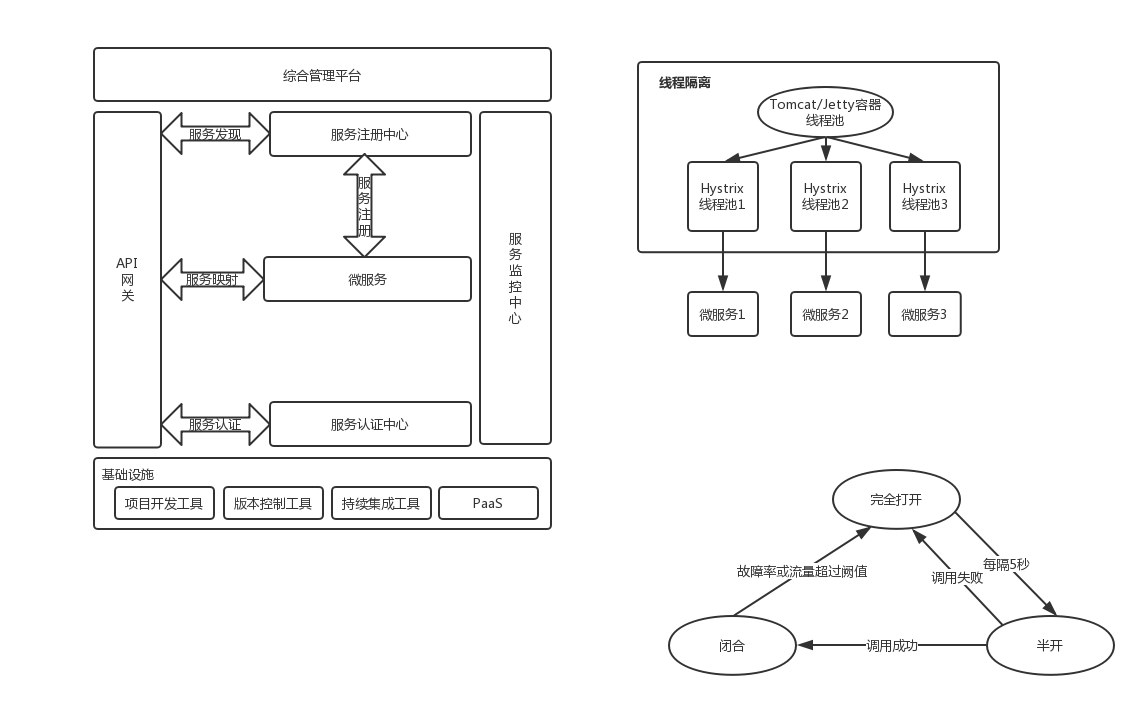
图2-2 Hystrix线程隔离机制

Hystrix将Tomcat/Netty的线程池资源做了进一步划分，对同一个微服务的请求在一个线程池内进行处理。若一个微服务发生故障导致对其的调用延迟，堆积的调用请求会挂入处理该微服务请求的相应线程池的等待队列中。而如果没有该机制的存在，堆积的请求会耗尽容器的线程资源，导致容器无法处理其他可以正常完成的服务调用请求。通过线程隔离机制，Hystrix将服务调用对系统资源的影响范围做了限制；通过和熔断机制一起工作，可以极大的减少故障服务对系统资源的占用，增加了整个系统的可靠性。

2.5.3 Hystrix熔断

熔断机制是模拟电路中的熔断器，当电流过大时，熔断装置将电路断开，从而保护家里的各种电器。Hystrix中的熔断器起到了同样的功能，具体是指当A服务调用B服务而B服务因为故障等原因一直不做反应时，A服务不是被无限阻塞，而是由实现熔断机制的后台线程将调用进行接管，直接返回调用错误信息给A服务，让A服务能够快速的对这种情况进行处理。熔断器分为三种状态：完全打开，半开，闭合。三种状态间的转换如图2-3所示。

1. 闭合状态：该状态下，所有的服务调用请求都允许通过，当发生因服务响应时间过长而被熔断器截断请求时，熔断器开始每隔一段时间该时间段内发生这种故障的次数和请求调用的次数，当故障比率和请求次数都超过阙值时，熔断器进入完全打开状态。其中熔断器进入打开状态的阙值和统计频率都可以进行配置。

图2-3 熔断器状态转换图

1. 完全打开状态：该状态下，熔断器接管所有的服务请求。若服务请求有备选的方案或默认值，则熔断器将采取该方案；若没有，熔断器直接拒绝调用请求。这样做给故障服务带来的恢复的机会而不会导致因用户或代码的反复调用尝试使得服务器进一步恶化。熔断器在此状态下每隔5秒会进入半开状态。
2. 半开状态: 在此状态下，Hystrix会允许一个调用通过，如果该调用成功，则Hystrix认为服务已经恢复，熔断器进入闭合状态，若没有调用成功，熔断器重新进入完全打开状态。进入半开状态的时间间隔也是可以配置的。

2.5.3 Hystrix降级处理

降级处理时在服务调用出现故障时采用退而求其次的方法，不是返回错误信息让调用者进行错误处理，而是尽可能的返回有用的信息，比如备选的轻量级方案或默认值。降级处理最常见的应用场景是在数据获取的过程中，如果数据服务调用失败，则尝试另一个数据源，如果又失败，则在本地数据库找到一个可用的默认数据返回。

2.6 微服务安全技术

2.6.1 OAuth 2.0认证协议

舰船上不同的子系统可能由不同的研发单位研发，而不同的子系统也可能由不同的部门负责监管，这造成了舰船平台中的服务存在于不同的信任域中，每个域可能使用不同的用户名和认证机制。如何跨域进行身份管理是OAuth主要解决的问题。

传统的跨域进行服务调用的方法是将信任域中用户的用户名和密码给需要请求调用该域中服务但处于另一个信任域的第三方应用来获取相应服务的调用权限。这样做有一些明显的问题：

1. 第三方应用需要存储用户的密码，而且密码通常使用明文存储，增加了密码泄露的风险；
2. 会导致第三方应用和用户一样具有完全的服务请求权限，这一般不是用户想要的，用户需要限制第三方应用具有的调用权限，例如只能调用查询信息功能而不能对信息进行删除或修改；
3. 当用户通过该方法给多个第三方应用授权时，修改其中一个应用的授权方案变得很困难，因为这需要用户修改密码，而修改密码后会影响到其他第三方应用的授权。

OAuth(Open Authentication)通过将用户角色与客户端角色进行分离，并引入授权层来解决传统密码共享的授权模式所存在的问题。OAuth是一个开放认证协议，允许第三方应用不使用用户的用户名和密码就可以完成对用户资源请求的认证和授权，并且通过授权机制限制第三方应用可以进行的操作[32]。

OAuth 2.0协议中的有四个主要的角色：

1. 资源所有者：拥有资源并能对资源的访问权限进行授权的实体，通常对应用户这一角色；
2. 资源服务器：用来托管受保护的资源的服务器，需要时用访问令牌来对服务器上的资源进行请求；
3. 客户端：经由资源所有者授权并代表资源所有者进行对资源服务器上的资源进行访问请求的应用程序；
4. 授权服务器：对资源所有者进行身份验证并生成并分发令牌给客户端的服务器。

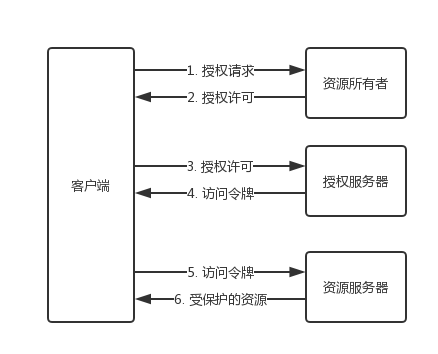
OAuth 2.0协议中四种角色间的交互如图2-4所示[33]。

图2-4 OAuth2.0 协议流程

OAuth2.0协议流程具体的描述如下：

1. 客户端向资源所有者发出授权请求；
2. 客户端接受到资源所有者返回的授权许可；
3. 客户端与授权服务器进行身份验证，通过授权许可请求访问令牌；
4. 授权服务器对客户端身份和授权许可进行验证，验证通过后生成和颁发令牌；
5. 客户端使用访问令牌请求资源服务器上受保护的资源；
6. 资源服务器验证访问令牌，若令牌有效则返回请求的资源。

授权许可作为代表资源所有者的凭据，具有以下几种类型[34]：

1. 授权码：这种许可类型不是从资源所有者获得的，而是客户端引导资源所有者到授权服务器进行身份验证，授权服务器引导资源所有者带授权码返回客户端；
2. 隐式许可：作为针对浏览器中用JavaScript等脚本语言实现的客户端进行优化的授权类型。作为资源所有者的授权，客户端直接获得访问令牌。因为减少了获取令牌所需HTTP调用请求数，这种类型的许可提高客户端的响应速度，但由于颁发令牌时授权服务器没有对客户端进行身份验证，所以其安全性有所降低。
3. 资源所有者密码授权：这种类型的授权直接使用资源所有者的密码凭据，凭据只用来请求访问令牌，并不需要在客户端进行存储，因为之后客户端使用访问令牌和刷新令牌作为凭据，故比直接共享密码的安全性高；
4. 客户端凭据：当客户端自己是资源所有者，或与授权服务器事先进行商议授权请求时，可以使用客户端凭据作为授权许可。

访问令牌由资源服务器和授权服务器实施其有效范围和时限，当访问令牌失效时，OAuth 2.0提供刷新令牌的机制对访问令牌进行更新。刷新令牌由授权服务器发放，用于在当前访问令牌过期时获取新的访问令牌。当客户端使用过期的令牌访问资源服务器时，资源服务器会返回访问无效，此时客户端使用刷新令牌向授权服务器获取一个新令牌。一般新令牌的权限和时限要逐步降低，这样可以增加系统的安全性。刷新令牌只供授权服务器使用，不会被发送到资源服务器。刷新令牌是为了保证访问令牌的安全性，虽然访问令牌使用起来比较方便，但随着使用次数的增加，其安全性能逐步降低，这时使用一直在客户端保存的刷新令牌对访问令牌进行更新，增加了访问令牌的安全性。

2.6.2 JSON Web Token

令牌是整个OAuth协议的核心机制，授权服务器生成令牌并分发给客户端，资源服务器接收并验证客户端发送的令牌，并通过令牌附带的权限决定是否同意客户端的请求。OAuth协议中只规定了获取令牌和使用令牌的方式，而没有规定令牌的格式。而授权服务器和资源服务器需要知道令牌的格式来生成和验证令牌。

JSON Web Token（JWT）就是规定了令牌格式和序列化方式的开放标准[35]。JWT采用基于URL安全的Base64编码方式对令牌进行编码。JWT令牌格式三部分：头部、载荷、和签名，这三部分通过“.”字符隔开。

1. JWT头部

JWT头部用于描述令牌的基本信息，如令牌的类型和令牌使用的签名算法等。例如：

{

“typ” : “JWT”,

“alg” : “none”

}

该JWT头表明此令牌为JWT令牌格式，没有采用任何签名算法。

1. JWT载荷

JWT载荷包含两类信息：

1. 标准属性：JWT预先设置好的属性；
2. 自定义属性：令牌使用者可以自定义各种符合JSON格式的属性，比如在认证过程中用户的身份信息等，但一般不建议使用自定义属性存储敏感信息，因为JWT采用的编码是未经加密的，任何人可以使用Base64解码器对令牌进行解码，这意味者属性中包含的信息都是明文信息。
3. 签名

授权服务器在签发令牌前需要对令牌进行数字签名用于确保令牌信息没有被篡改，签名方式是先将JWT头部和载荷分别使用Base64进行编码，编码后用字符“.”连接，连接后的字符串和服务器的私钥一起使用JWT头部声明的签名算法进行签名得到JWT令牌的第三部分。

采用JWT标准实现的令牌具有以下优点[36]：

1. 由于采用JSON格式，JWT具有广泛通用性，绝大多数编程语言都可以对令牌信息进行提取；
2. 授权服务器可以将用户权限信息放在JWT载荷中，这样资源服务器在进行访问权限控制时不需要通过数据库访问就可以查询到权限信息；
3. JWT构成简单，占用空间小，可以在HTTP报文头、HTTP访问参数和HTTP表单中进行传输。
4. JWT编码后不会跟URL的编码格式起冲突，并且不会被中间件系统中的JSON反序列化组件误识别，从而增强了JWT令牌的通用性。

2.7 本章小结

本章主要介绍了在对舰船平台服务化治理方案的研究过程中遇到的相关重要概念和使用的有关技术。首先对SOA的演变过程进行了介绍，给出微服务架构实现SOA的优势，然后介绍了流行的微服务开发框架Spring Boot。接下来对微服务通信设计所使用的REST架构风格、部署方案使用的Docker技术、熔断所使用的Hystrix技术进行了描述。最后，本章对微服务架构中实现安全认证所广泛使用的OAuth 2.0协议和JWT令牌标准规范进行了描述。

3 舰船平台服务管理系统的需求分析

本章从微服务系统所面临的一般性挑战和面向舰船平台服务管理的系统目标出发，分析实现面向舰船平台管理的舰船平台服务管理系统的功能性和非功能性需求。本章将对各个功能模块进行建模，从而确立系统的功能性需求。此外，本章还从易用性、性能、安全性、可靠性、可管理和可扩展性等方面对系统的非功能性需求进行了分析和描述。

3.1 微服务系统面临的挑战

现如今舰船系统的设备越来越复杂，舰船平台服务管理系统要能够对舰船内部各个子系统进行统一管理，传统的单体式架构所构建的垂直应用已经不能满足舰船平台统一管理和子系统间的交互和集成的需求。面向服务架构比单体式架构更加灵活，更能满足当代舰船信息一体化需求。通过将各个子系统应用的公共业务进行抽取并独立出来作为公共基础服务，形成平台服务管理模式下的底层服务基础设施。采取微服务架构实现面向服务架构相比传统的ESB方式对舰船平台服务管理系统进行设计与实现，一方面可以保证这些基础组件的高可用和负载均衡，另一方面可以降低新应用的开发难度，从而适应新型舰船需求灵活多变的特征。然而，采取微服务架构仍面临各种各样的挑战[37][38]。

总的来说，采用微服务架构构建舰船平台服务管理系统并实现微服务带来的好处，具有以下挑战：

1. 实现灵活性的同时保证高性能

传统的单体架构业务流程中的模块以函数的形式呈现，模块间的调用发生在同一个进程之中，而微服务架构将各个模块划分出来形成独立自治的服务，虽然这降低了系统维护的难度，增加了系统的灵活部署和扩展的能力，但模块间的调用也由之前的进程内的函数调用转变为进程间的远程调用，增加了调用延迟。而微服务的粒度越小，实现业务逻辑的服务调用次数就越多，如何多次远程服务调用的情况下保持系统的性能给舰船平台服务管理系统的设计带来了巨大的挑战。

1. 实现微服务的快速开发和部署

微服务架构要求服务快速迭代开发和持续部署的同时还要实现权限控制、负载均衡、健康监控等功能。而微服务的细粒度意味着微服务开发数量的增加，如果开发人员要在每一个服务中考虑上述功能的实现，这无疑增加重复的工作，降低了开发效率。所以要求微服务系统提供开发框架，封装公共功能的实现，规范接口设计标准，提供集成开发和部署工具，实现项目的从初始化、编译到集成和部署的自动化。

1. 微服务架构中的安全性和稳定性考虑

对于任何系统来说，安全性和稳定性都是需要首先满足的特性，不安全或者不稳定的系统是无法使用的，而对于舰船系统来说，系统的安全性和稳定性有更高的要求。微服务架构中的安全性要求系统提供完善的身份认证和访问控制机制。而稳定性要求当服务出现故障或错误的时候，系统能够及时的检测并进行服务恢复。系统还应通过多实例部署、服务降级或服务熔断等机制来提高系统和服务调用的稳定性。

1. 确立微服务的治理方案

在舰船系统服务化的过程中，随着服务数量的增长，服务的规范治理显得尤为重要。当代舰船的计算基础设施逐渐走向虚拟化，以IaaS或PaaS的形式呈现。故可以参考云服务的治理模式，实现服务的独立部署、升级和扩容。云计算的弹性伸缩，多实例部署，健康监控和容量规划等服务运维措施都可以融入到舰船平台系统的设计当中。

把微服务治理应用到舰船系统中，要采用线上线下相结合的方式。通过服务的上下线审核、服务文档管理和服务协作规范等制度的规定来实现服务的线下治理。通过服务目录、服务监控、服务日志、用户管理和服务权限管理等手段来实现服务的线上治理。如何协调和实现统一的线上线下的服务治理方案，也是设计舰船服务化平台管理系统需要重点考虑的问题。

3.2 舰船平台服务管理的系统目标

舰船平台服务管理的主要目标是为了推进舰船平台化发展，引入服务治理规范，设计和实现面向服务管理的微服务开发和集成框架。

经过研究现有的微服务开发和集成框架，首先，为了降低服务的开发成本，舰船平台服务管理系统应提供服务开发框架，封装服务通信、服务生命周期和健康管理等模板代码，降低开发人员的工作量，提高开发效率。系统还应该提供服务注册发现的功能，可以随时查看服务实例的运行状况；系统应该提供与微服务治理策略相一致的辅助工具，包括服务的上下线管理、服务发布管理、服务目录管理等；系统还应该提供身份认证与访问权限控制，保证各项操作都存在日志记录从而可以随时对违规操作进行审计；此外系统还需提供服务配置和部署的自动化工具，形成规范的服务综合治理平台。

3.3 舰船平台服务管理系统的功能性需求分析

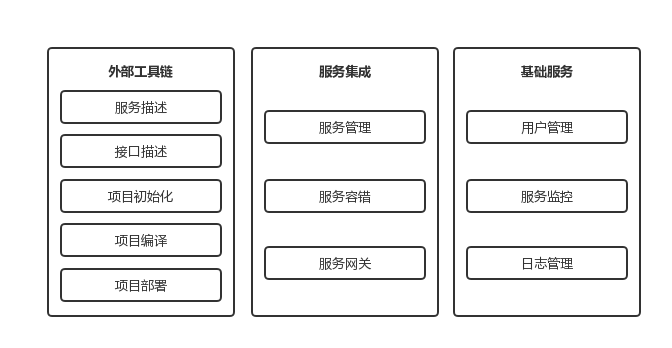
根据舰船平台管理系统的系统目标，可以将系统按功能划分为三个模块，包括外部工具链、服务集成和基础服务三大功能模块，具体如图3-1所示。

3.3.1 外部工具链

外部工具链是用来支持服务开发与部署的自动化工具的统称。外部工具链主要供服务开发人员使用，开发人员通过使用外部工具链实现对服务和接口的完整描述，以及对项目开发和部署进行管理。其用例图如图3-2所示。其主要功能包括服务描述、接口描述、项目初始化、项目编译和项目部署。

具体的需求描述如下：

图3-1 舰船平台服务管理系统功能架构

1. 服务描述

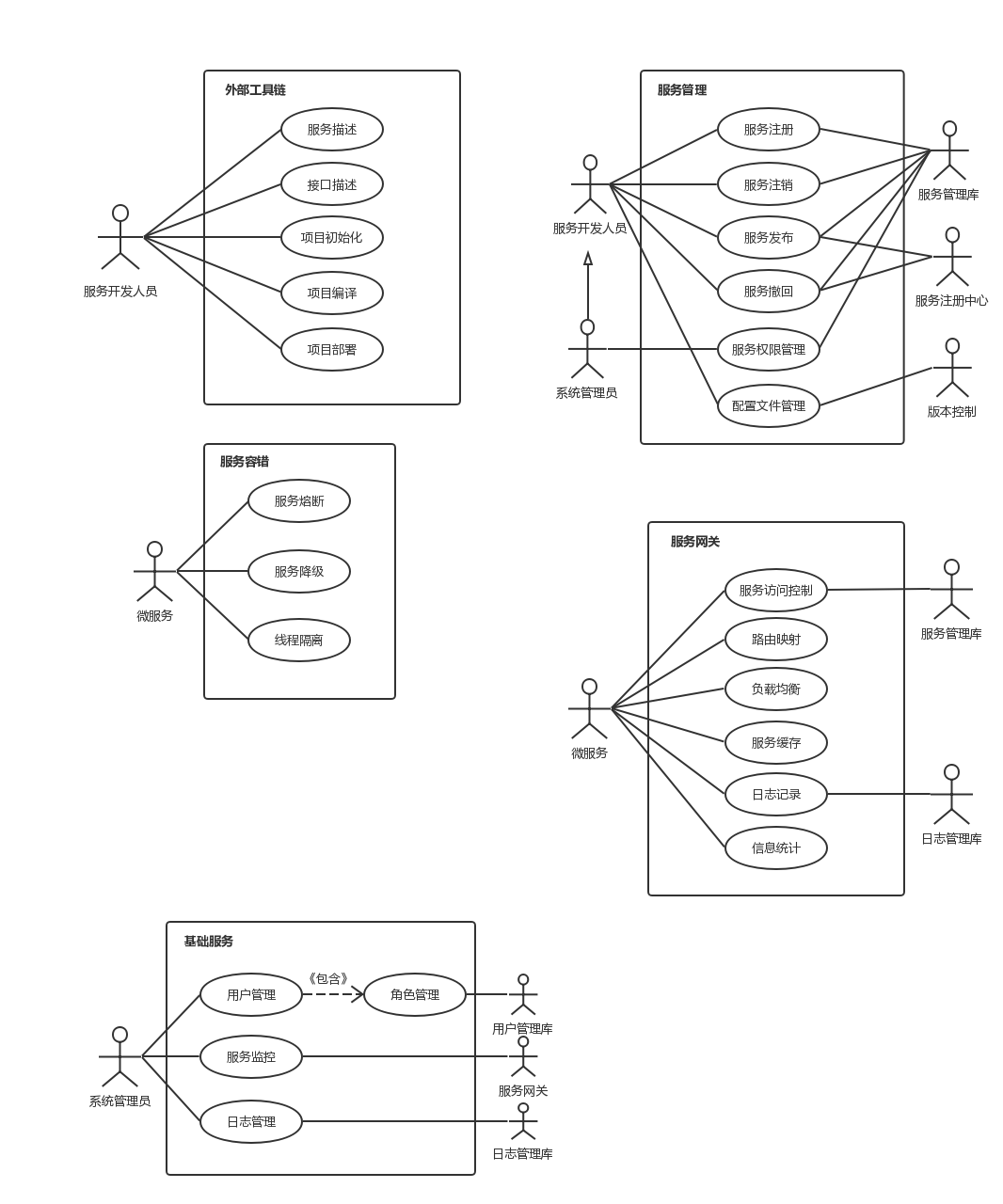
系统应支持通过JSON文件描述服务的主要信息，包括服务的标识、服务的描述、服务版本信息、服务的接口、服务开发的依赖、服务地址等服务描述信息。服务描述文件由服务开发人员进行编写并上传至服务注册中心，服务描述文件不仅是服务调用的规范性文件，还作为服务调用者与服务开发者之间的合同。服务调用者通过查询服务注册文件来规范对服务的调用，包括查询服务名称，服务接口，服务实现功能描述，服务提供者的联系方式，帮助服务调用方更好使用服务来完成舰船管理的目标。

图3-2 外部工具链的用例图

1. 接口描述

使用接口定义语言对接口进行描述，包括接口所属的服务的名称、接口的名称、接口的调用路径和接口接受的参数类型、接口返回的结果类型。Spring Boot定义了一套良好的具有REST风格的接口描述规范，允许开发者使用便捷的注解方式将REST接口与函数接口对应起来，简化了对接口实现的开发工作。

1. 项目初始化

系统应提供开发框架，或利用已有的成熟开发产品，自动生成符合开发规范的目录结构，生成自动化构建工具的模板配置文件。系统应该生成符合Maven等主流构建工具的开发、测试目录和pom.xml等文件，方便系统的自动化构建、测试和部署。

1. 项目编译

面向舰船平台服务系统应能够支持采用Maven等自动化构建工具对项目进行自动化的构建，编译脚本根据接口描述文件在目录框架中生成相应的服务代码框架，该代码框架集成了与面向舰船平台服务系统关键组件进行交互的相关代码，自动引入相关依赖，并提供系统的规范接口供服务开发人员调用。

1. 项目部署

系统应提供项目的自动化部署脚本，将使用脚本文件通过Maven命令将服务源代码构建生成的jar包制作成相应的Docker镜像，并根据服务描述文件的内容将镜像部署成Docker容器或部署到舰载集群服务器上。

3.3.2 服务集成

服务集成包括了系统在实现服务调用过程中所需的一系列功能，进一步可以划分为服务管理、服务容错和服务网关三个功能子模块。

1. 服务管理

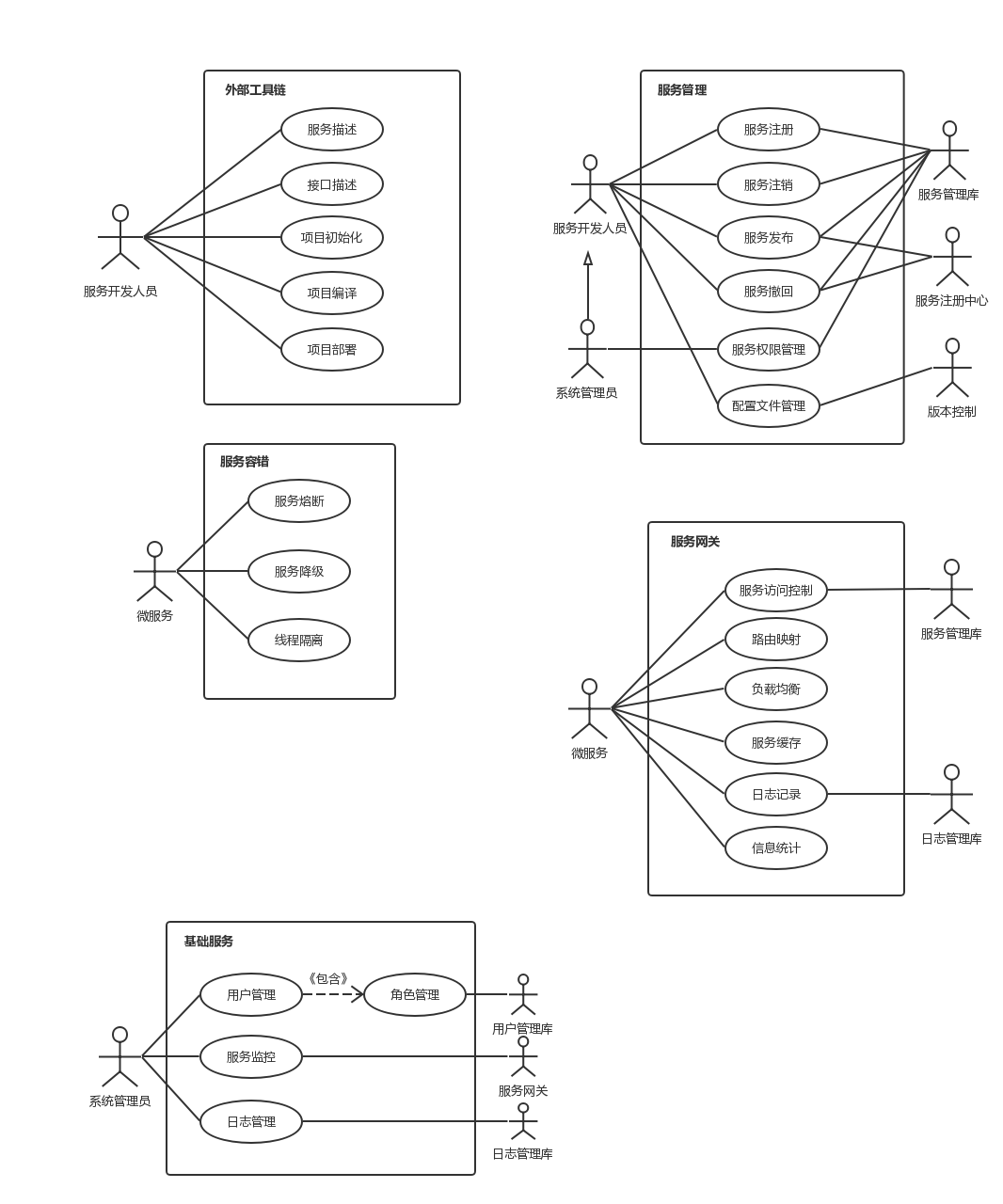
服务管理主要是系统管理员和服务开发人员对服务的元数据和配置文件进行管理，其用例图如图3-3所示。

图3-3 服务管理的用例图

服务管理包括的功能有：

1. 服务注册/注销：开发并测试好的服务完成部署后，需登录系统并借助本功能向系统注册，注册的信息连同服务描述文件一同保存于服务管理库中，注册的信息包括服务的分类信息和描述服务的各种元数据。服务注销是服务注册的反操作，它从服务管理库中删除服务的分类与描述信息与服务描述文件。服务的注册与注销都要有完整的审核机制，系统应该提供权限机制对服务的注册和注销操作进行管理。
2. 服务发布/撤回：刚注册的服务对外界是不可见的，需要系统管理员或具有相应权限的服务开发者利用此功能将已注册的服务通过注册中心向外界发布，使其对外部可见。服务撤回是服务发布的反操作，使服务调用者无法从注册中心发现服务的地址，从而无法对服务进行访问。加入服务发布/撤回功能有助于舰载环境线下的审核管理，可以设定具体的权限的发布/撤回流程来规范服务的上下线工作。
3. 服务权限管理：对权限的增删改查。服务开发人员可以对平台中所有已发布的服务进行选择并订购，服务管理员对订购行为进行审核，若审核通过，则给予订购者以服务授权。服务授权的结果是给订购者颁发一个OAuth令牌，在服务运行阶段，所有对授权服务的访问都依据该令牌进行身份认证访问控制。对来自于同一子系统或同一用户的服务，缺省认为它们之间当然具有相互访问（接口调用）的授权，除非服务提供者另外指定访问权限。系统还应支持对服务调用权限的修改、删除和查询，包括对开发人员可以使用的OAuth令牌查询，所需调用的接口需要的权限查询和已有的权限查询，对权限的修改包括对权限本身能控制的调用范围进行修改和对开发人员具有的权限进行修改，对权限的删除包括令牌，权限的删除等。
4. 配置文件管理：系统应负责将服务的配置进行集中管理和版本控制，并提供机制将更新的配置文件推送到每一个受影响的服务实例上，从而保证服务实例运行环境和运行初始状态的一致性，当修改服务配置后，服务的所有实例将接到通知并更新配置信息。
5. 服务容错

服务容错主要是为调用其他服务的微服务提供failfast、failover和隔离机制，保证服务故障不会消耗系统的资源和影响上游服务，其用例图如图3-4所示。

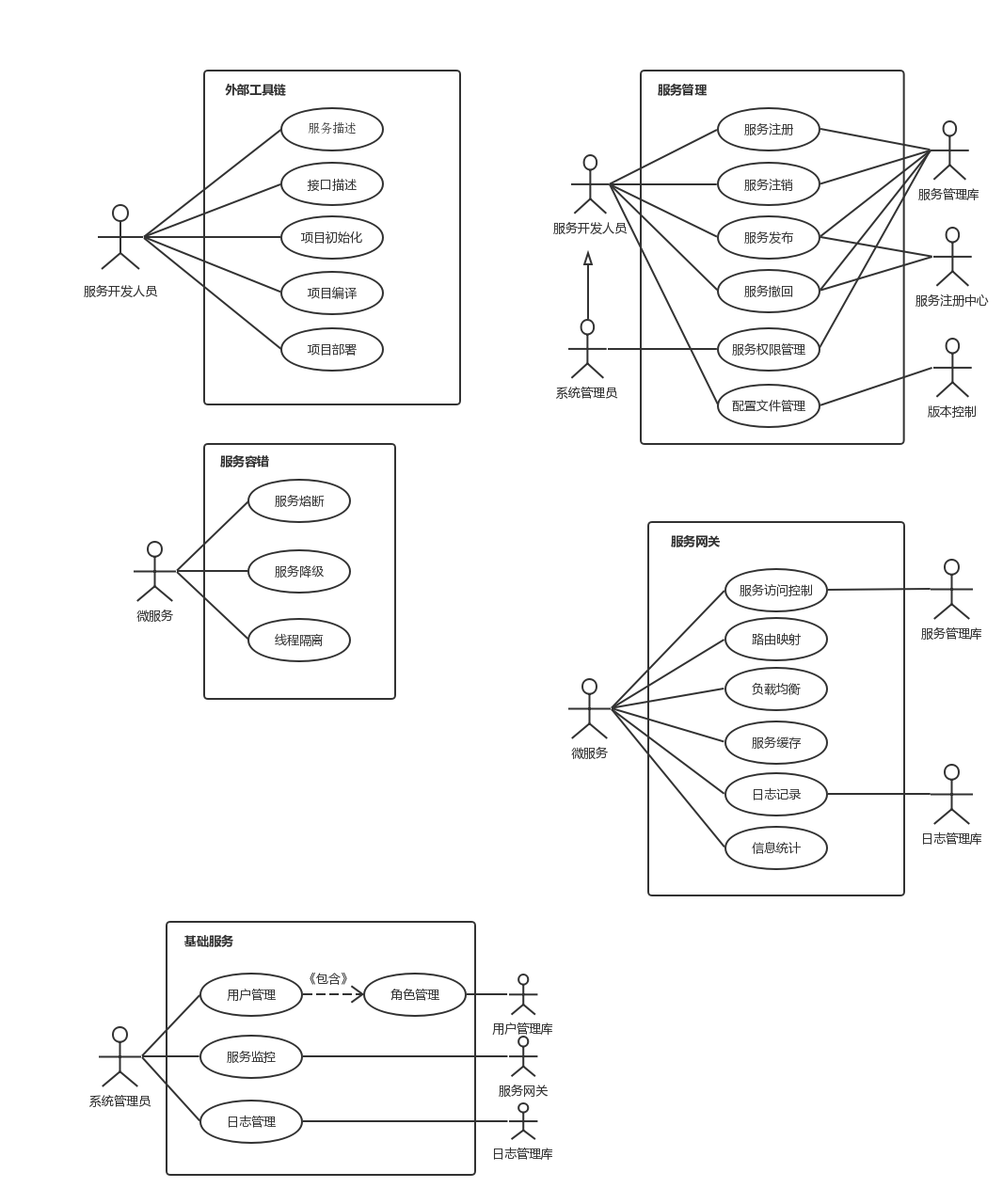
服务容错包括的功能有：

图3-4 服务容错的用例图

1. 服务熔断：这是处理服务故障并借以提高系统性能的机制。当服务过长时间未做出响应时，系统应该接管该服务的调用，选择直接返回异常供服务调用者处理，或者提供降级机制使调用方不会无休止的挂起。在处理服务故障的过程中，系统应对服务的访问结果进行统计分析，如果服务调用和失败次数超过一定阙值，系统应阻断对该服务的任何调用请求，以防止服务因过载而出现的性能的进一步恶化。在服务熔断后，系统应提供恢复机制，及时的监控服务的恢复信息，一旦服务恢复，则重新允许服务的调用通过，保证系统的正常运行。
2. 服务降级：这是在服务出现故障时，系统通过用户注册的备选方案对其他可替代的服务进行调用或返回默认值，从而使系统不会因为没有调用成功而返回空值导致调用发阻塞，或导致同一事务的其他服务调用阻塞。
3. 线程隔离：这是防止一个微服务调用线程的阻塞导致虚拟机线程资源耗尽而无法调用其他微服务的情况，通过将每个微服务的调用限制在一个线程池中，可以将调用阻塞的影响限制在线程池范围内，而机器整体的线程资源不会受影响。
4. 服务网关

服务网关是对系统上所管理的微服务调用的唯一入口，网关依据服务管理所产生的信息，对所部署并发布的服务的调用提供支持，所有对所发布的服务的访问经过网关进行统一的处理，作为微服务的反向代理，网关接收服务调用者的请求，并根据路由规则将请求转发到具体的服务实例上，其用例图如图3-5所示。

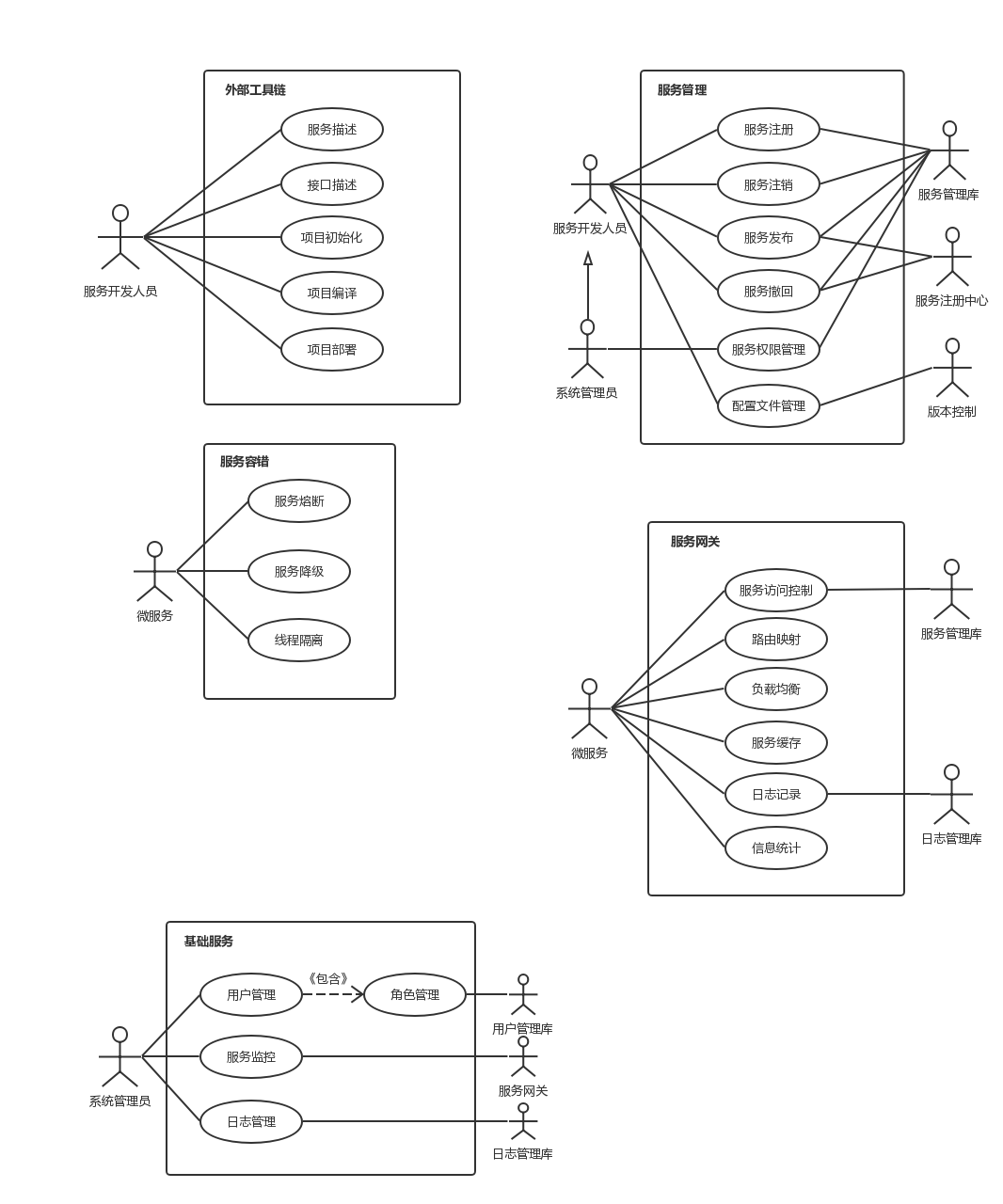
服务网关的主要功能包括：

图3-5 服务网关的用例图

1. 服务访问控制：依据服务订购与授权信息，对每一个发布的服务的接口调用进行权限核查与控制，对于通过权限检查的请求进行进一步处理，未通过的请求进行日志记录和返回错误响应码。
2. 路由映射：系统对外提供统一的服务调用URL，并提供从该路径到具体服务实例的地址的转换。路由映射作为服务网关的主要功能，将服务实例地址进行统一的管理起来，为微服务调用者屏蔽了具体的物理地址细节，简化服务调用过程。
3. 负载均衡：如果后台被调用服务启动了多个实例，可从服务注册中心获取请求服务的所有实例，通过轮询或随机等策略将服务调用请求交给不同的实例进行处理，防止每次服务调用都访问同一个实例而导致该实例崩溃而其他服务实例处于空闲状态。系统在进行负载均衡的同时保持服务会话的完整性，且后台服务应该是无状态的，从而可以对服务按负载进行动态的扩容和缩容。
4. 服务缓存：这是提升服务性能的机制之一。系统依据一定的原则，对后台服务调用的结果进行缓存，对后续到达的相同访问请求，直接利用缓存结果进行响应。
5. 日志记录：系统对服务的调用接口、调用参数、调用结果、请求调用服务的信息和调用失败信息等记录到日志，用于系统的监控和审计。
6. 信息统计：系统对接口的调用次数和服务的响应时间等进行统计，用于对系统的性能进行分析。

3.3.3 基础服务

基础服务包括用户管理、服务监控和日志管理三个子功能，基础服务用于对舰船平台服务管理基础设施进行管理、监控和维护，使用基础服务的主要是系统管理员，管理员通过系统提供的管理界面实施基础服务的功能，其用例图见图3-6。

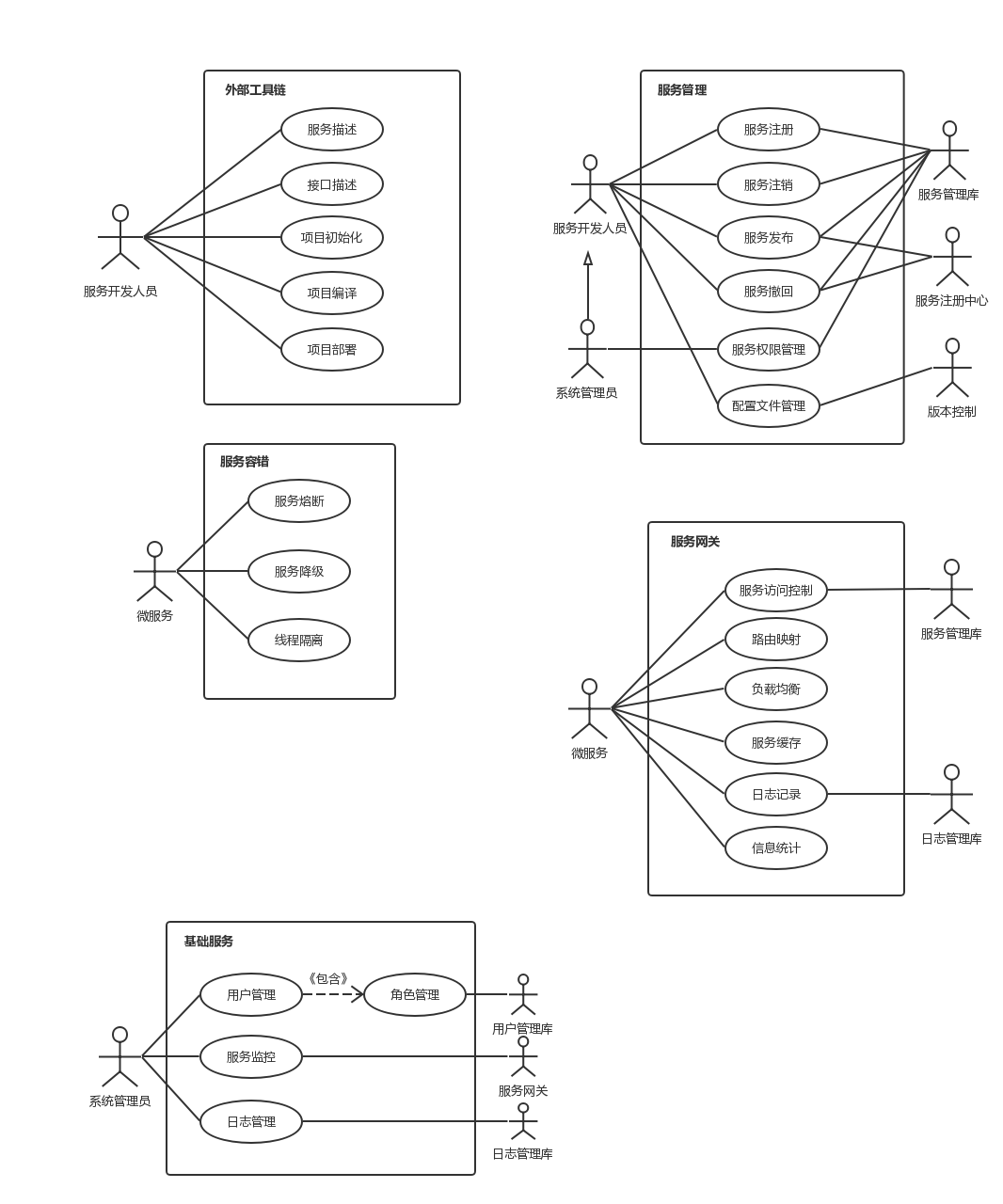
该功能模块的具体功能描述如下：

图3-6 基础服务模块的用例图

1. 用户管理

用户管理部分主要使用基于角色的访问控制模型(RBAC)对用户和权限进行管理。主要包括用户管理和角色管理。

1. 用户管理：对用户信息的增删改查。用户管理包括系统管理员添加用户，删除用户，修改用户信息和查看用户信息，也包括其他用户对自己信息的维护，包括修改密码，查询权限，进行个人信息修改等。
2. 角色管理：对角色的增删改查。角色代表用户对系统所管理的服务进行调用的权限。管理员通过修改删除角色来对系统中接口的调用权限和用户的权限进行管理。用户也可以通过此功能来查看自己所拥有的权限。
3. 服务监控

服务监控通过服务网关收集的信息向系统管理员显示系统的运行状况，服务监控包括服务健康监控和服务异常监控。

1. 服务健康监控：对服务实例的运行状况进行监控，包括服务实例是否正在运行，服务响应时间是否超时等。
2. 服务异常监控：系统组件作为微服务也受到服务网关的管理，服务网关通过对系统组件的调用情况来监控系统组件的运行状况，发生异常提醒运维人员及时进行修复。
3. 日志管理

日志管理包括对服务网关进行收集的日志的查询和汇总，方便系统管理员对系统的审计跟踪。

3.4 舰船平台服务管理系统的非功能性需求分析

舰船平台服务管理系统除了要满足上述的功能性需求以外，还要满足易用、高性能、安全、可靠、可管理和可扩展性等非功能性需求。

1. 易用性

任何系统的开发目标都是为了降低使用者的工作量，舰船平台服务管理系统作为服务集成和管理系统，应该从开发、测试、部署到监控全方位降低开发人员和运维人员的工作量。具体来说，系统应该通过提供公共模板简化服务开发，提供自动部署工具以简化服务部署，通过良好的UI界面和自动化反应机制以简化服务监控。

1. 高性能

舰船平台的性能要求虽然不像机载平台那么高，但由于面对作战要求的特殊性，要求能及时对数据进行处理并对事件进行响应。API网关处理服务请求的时间不能影响服务的正常调用。由于一个业务的处理可以包含多个微服务调用，这要求平台上的服务的请求处理时间不得超过1秒，对可能超时的请求，应采用异步方式进行处理，从而不会给上游服务带来过高的延迟。

1. 安全性

舰船平台服务管理系统应该满足舰载环境对安全性的需求，对于服务间传递的信息，要通过HTTPS协议进行加密通讯，对安全性要求高的数据一律采用加密手段进行传输，防止数据被非法劫持或篡改。

对于舰船平台服务管理系统，还应对服务调用请求进行身份认证和权限控制，防止服务的越权请求调用，同时应建立健全的安全规范，避免系统发生重大安全隐患。

1. 可靠性

对于API网关和服务注册中心等平台的核心部件，应保持高可用和可靠性。舰船平台的核心组件应能保证99.99%的高可用，并且应考虑战时出现部分基础设施损毁的情况下仍保证平台功能的可用性。

1. 可管理性

系统中服务的部署、运行和调用可以实现监控、日志、数据采集和追踪。通过提供图形用户界面对系统以及系统中运行的服务进行管理配置、监控统计、追踪审计，并可以对系统自身的运行情况进行监控和管理。

1. 可扩展性

可扩展性包括系统的功能模块可通过插件进行功能扩展，即允许用户自行开发功能插件，系统应该留好开发接口供开发人员进行扩展；系统还应该可通过控制版本兼容性对服务接口进行扩展，实现逐步地上线新的功能而不影响已有应用的运行；此外，在系统中的服务实例部署应可以动态扩展，根据负载的情况自动的进行服务实例的增减，以满足在服务请求高峰有足够处理资源的同时在系统负载低的情况下减少实例部署，节约系统的计算资源等。

3.5 本章小结

本章对基于微服务架构的舰船平台服务管理系统进行需求分析。采用微服务架构需要保证服务调用的性能，实现服务开发部署的快速迭代，保证系统的安全稳定并制定规范的治理方案。所以舰船平台管理系统应提供易于使用开发框架和服务描述框架，提供服务调用过程中所需的服务注册发现、路由映射，提供身份认证、权限控制、负载均衡、熔断等机制来保证安全性和稳定性，提供健康监测、信息统计、日志记录来实现服务治理，同时提供自动化部署以实现微服务的伸缩性。系统在满足上述功能行需求的同时，还应满足易用、可扩展和可靠性等非功能性需求。

4 舰船平台服务管理系统的概要设计

完成了对系统的需求分析后，本章主要描述舰船平台服务管理系统的概要设计，给出舰船平台服务治理方案、然后对方案中的核心模块API网关以及与其关键的系统模块进行了模块的划分、接口设计以及数据库设计。

4.1 舰船平台服务治理方案

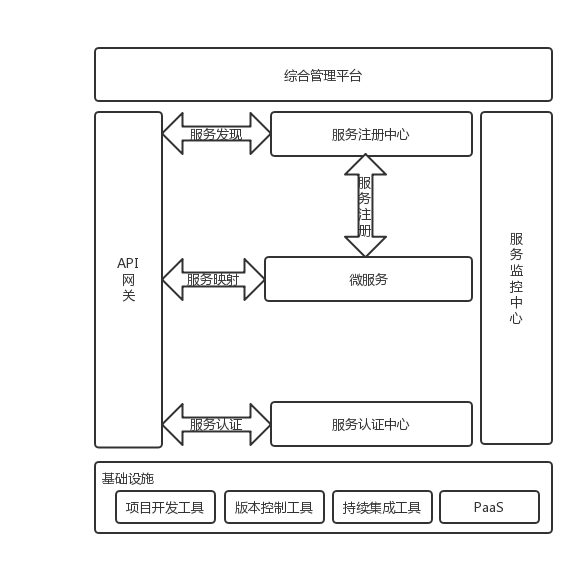
舰船服务平台化的核心在于提出一个合理完整的服务治理方案，该方案包括系统服务化所需的开发、集成、部署环境和安全、监控、管理手段。治理方案应该遵循微服务的基本原则，即服务小而自治、服务间通过轻量级REST API通信、服务独立部署、失效隔离和有效的日志和跟踪[39]。图4-1展示了面向舰船平台管理的服务治理方案的架构。

图4-1 舰船平台服务治理架构

该治理方案包含以下几个部分：

1. 服务综合管理平台

服务综合管理平台负责对整个系统进行管理，是系统的综合门户。通过该平台服务开发者可以对服务进行注册撤销和各种查询工作，可以查询微服务列表，查找每个微服务的接口信息、实例信息、相互依赖关系、统计数据、服务运行状态等系统应用运行的关键数据和信息。管理员可以在查询的基础上增加对用户、角色和权限进行管理。平台同时集成监控中心，可以对系统关键组件的运行状况进行监控，对系统的异常进行处理。

1. API网关

API网关作为调用所有微服务的入口，提供统一的调用接口，并将该接口映射到具体的服务实例地址。API网关作为反向代理，接受服务调用请求并对请求进行处理，如身份认证、路由映射、报文转换和日志记录等，然后将请求发送到服务实例进行处理。API网关还通过一定的策略在服务的所有实例之间进行选择，保持系统间负载的平衡。此外API网关还通过熔断等机制保证服务不会无限阻塞，从而及时地释放系统资源，保证系统稳定可靠地运行。

1. 服务注册中心

服务注册中心负责服务描述信息、服务接口信息以及服务实例信息的存放。服务注册中心提供API进行服务注册、查询和上下线功能。服务注册中心还定期地对系统中管理的所有服务实例进行健康检查，并及时更新服务列表信息，从而给系统其他组件提供准确的服务实例运行信息。API网关通过服务注册中心查找服务实例的地址来进行路由映射的功能。

1. 服务监控中心

服务监控中心进行服务健康检查，对服务日志进行采集，并对服务的运行状态进行监控。当监控到服务故障时，及时更新服务注册中心，并通过管理平台通知运维人员进行检测与维护。开发人员可以通过监控中心采集的日志来分析调用过程中的安全漏洞和性能缺陷。通过对服务调用进行追踪可以发现性能瓶颈，从而使开发人员可以对系统进行有效的优化。

1. 服务认证中心

服务认证中心负责验证令牌的生成、管理和分发。令牌用于调用过程中对服务调用者进行身份认证和调用权限的确认。服务认证中心通过验证服务调用者传递的令牌确认调用者身份，然后查找与身份相关的调用权限，决定是否允许调用请求。系统的关键组件如注册中心、监控中心等也以微服务的形式存在，故也受到服务认证中心的保护。

1. 基础设施

基础设施包含支持服务开发、部署、运维的系统工具和服务运行环境，这些工具虽然没有直接参与服务调用，但极大地简化了服务开发人员、服务运维人员和系统管理员的工作，从而也被纳入平台服务治理方案的范围：

1. 项目开发工具：负责服务开发过程中的目录结构管理，项目依赖管理以及代码生成和调试等辅助开发工具。
2. 版本控制工具：负责服务代码和服务配置的版本控制，记录代码和配置文件的版本信息。
3. 持续集成工具：对编译、版本控制、代码生成以及部署进行整合，利用自动化脚本完成代码提交到版本控制工具后进行自动编译、打包、Docker镜像的生成以及自动部署到PaaS平台。
4. PaaS平台：将舰载硬件计算资源整合后进行虚拟化的平台。平台提供一致的操作系统环境，支持Docker镜像的部署和管理，以及对平台本身进行监控和管理。

4.2 模块设计

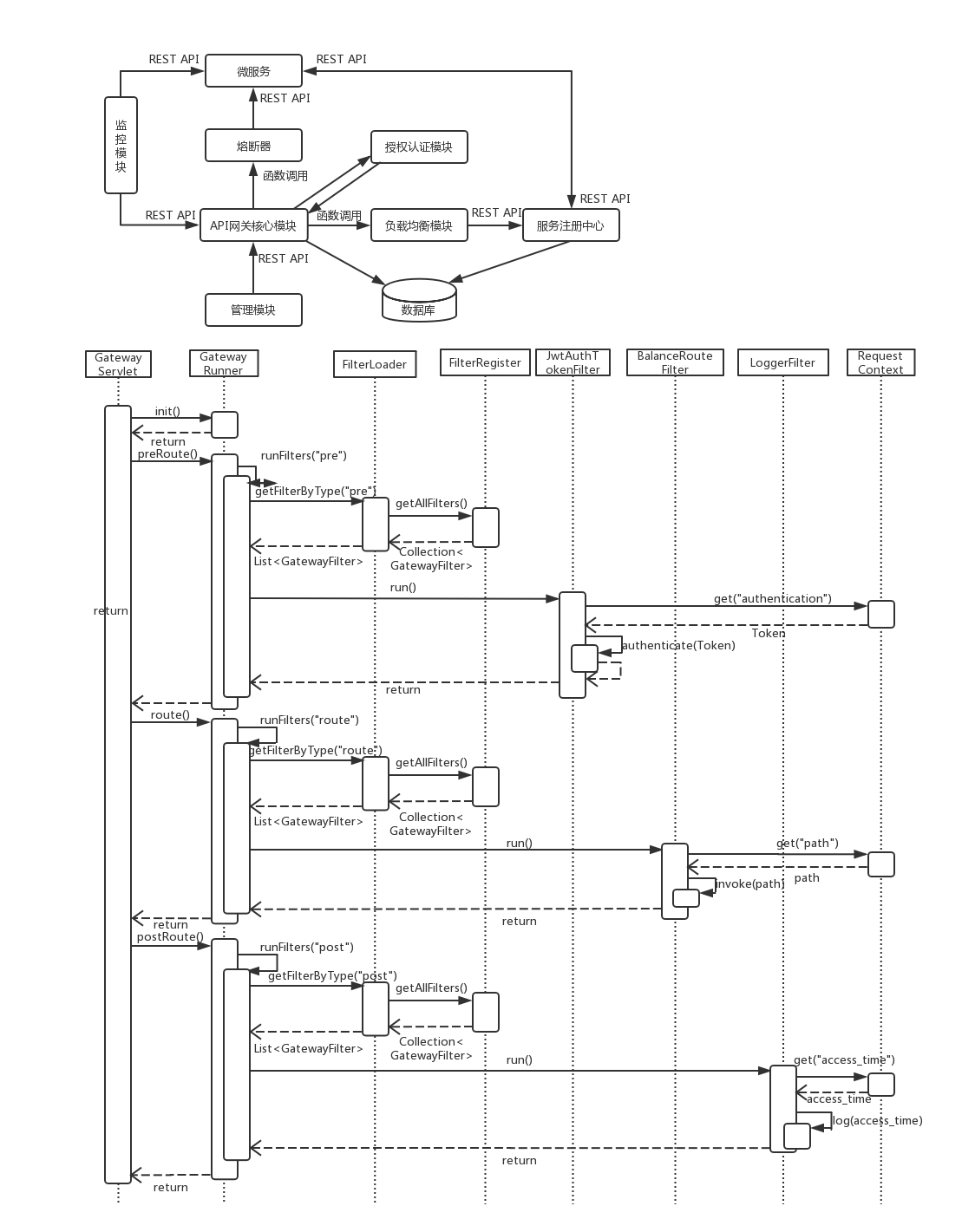
舰船平台服务管理系统的模块划分如图4-2所示。

图4-2 舰船平台服务管理系统模块

模块与模块间的调用关系描述如下：

4.2.1 API网关核心模块

API网关核心模块是调用系统管理的服务的统一入口，通过提供统一的服务调用的URL，屏蔽了具体服务实例的路由细节，其与负载均衡模块、授权认证模块一起构成API网关。API网关作为反向代理拦截服务调用的请求，并通过一系列处理后代理客户端发出请求，并将处理结果返回给客户端。API网关发出请求前进行的处理包括：

1. 身份认证：根据请求中包含的令牌向认证模块发出请求，确定服务调用者身份，检查服务调用权限，若身份验证通过且调用者具有调用权限，则进行后续处理，否则返回HTTP 403状态码；
2. 日志记录：将记录调用者的ID，调用时间，调用的服务及接口等调用信息记录到日志库中；
3. 路由映射：将对外暴露的服务地址转换为服务实例的具体地址，这一功能是通过调用负载均衡模块完成的；
4. 请求报文处理：对HTTP请求报文进行处理，包括报文格式转换，报文头内容替换，根据报文头的相关字段实现动态路由等；
5. 调用次数统计：对接口的调用次数进行更新，并更新相关的数据库；
6. 负载均衡：API网关通过负载均衡模块实现了服务的均匀分发；
7. 认证与授权：API网关通过授权认证模块实现了对服务调用者身份的识别和请求的权限检查；

API网关进行服务调用时，不是直接将请求发送到服务实例，而是调用熔断器来实现服务容错的相关功能，具体过程间熔断器模块的描述。

在API网关接收到响应结果并将结果返回给客户端前，API网关还可以进行相应的处理：

1. 响应时间统计：对调用的响应时间进行计算，并更新相关的统计数据。
2. 响应报文处理：对HTTP响应报文进行处理，包括报文格式转换，报文头内容替换等。

API网关处了进行相应的业务逻辑处理，还提供REST API接口暴露网关的运行数据、统计数据和路由数据供监控模块使用。

4.2.2 负载均衡模块

负载均衡模块负责将服务调用请求平均分发到每一个服务实例中，从而平衡系统的负载。负载均衡模块以库函数的形式供API网关调用。其通过服务ID发送REST请求到服务注册中心获取服务实例列表，然后根据数据库中有关该服务的负载均衡策略运行负载均衡算法，从服务实例列表中选出一个服务实例返回给API网关进行路由映射。

4.2.3 熔断器

熔断器负责具体的服务调用，它以代理模式代替API网关进行服务调用，通过代理机制，熔断器可以在被调用的服务发生故障或相应时间过长的情况下做出处理。通常的处理方法有：

1. 服务熔断：中断服务调用，直接抛出异常。
2. 服务降级：通过配置的策略在服务调用出现异常时选择其他方案对调用进行相应，比如调用候选服务或返回默认值等。

4.2.4 服务注册中心

服务注册中心时服务治理方案中的一个重要组件，负责服务的注册与发现。服务注册中心通过提供一系列REST API来进行服务实例查询，服务列表查询，服务实例注册等功能。服务注册中心通过在对服务实例进行注册的同时更新相关数据库，供管理模块使用。服务注册中心通过定期调用微服务的健康接口来检测服务实例的运行状况，当发现服务故障时更新实例列表，剔除发生故障的服务实例信息。

4.2.5 授权认证模块

授权认证模块对应服务治理方案中的服务认证中心，负责令牌的生成、分发和验证。认证模块生成的访问令牌与相关的访问权限一起存储在数据库中。授权认证模块提供REST API接口通过API网关暴露给外界使用，主要用于请求令牌调用。

4.2.6 监控模块

监控模块对应服务治理方案中的服务监控中心，通过对微服务暴露出的状态信息接口和API网关提供的统计信息接口的调用，对系统上运行的微服务和各个组件进行健康监控，运行状态监控，统计信息监控等。该模块采用MVC模式进行设计，可以集成到服务治理方案中的综合管理平台上。

4.2.7 管理模块

管理模块负责对系统用户、服务、权限和令牌等进行管理。主要通过对数据库的操作实现对各个实体的增删改查。对系统参数如负载均衡等进行配置，并实现对系统中各类信息的查看，包括系统日志、服务列表、服务运行信息等。该模块采用MVC模式进行设计，可以集成到服务治理方案中的综合管理平台上。

4.2.8 微服务

微服务作为被系统管理的对象，其本身通过代码框架集成了很多统计服务运行信息的功能，并通过REST API接口形式暴露给系统相关组件进行调用。微服务在启动时，通过调用服务注册中心的API进行服务注册，将服务实例的地址等相关信息传递给注册中心进行保存。微服务提供健康监测接口供服务注册中心和监控模块调用。

该架构中系统的关键组件如API网关，认证模块和服务注册中心也作为微服务运行，故这些组件也提供相应的API供监控模块和管理模块进行调用，实现系统的运行态的监控和配置管理。

4.2.9 数据库

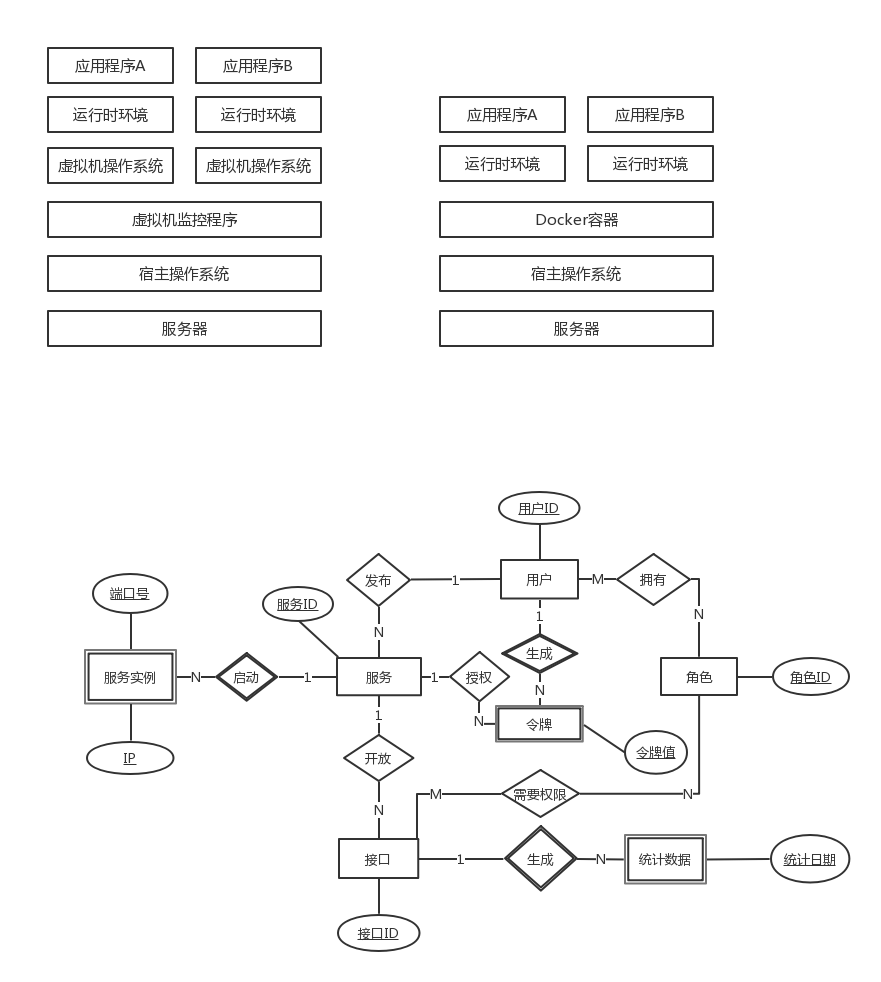
数据库用来存储系统中的各种实体的相关信息，包括用户、服务、令牌、权限、接口、服务实例、日志等。系统的各个组件都与数据库进行交互，认证模块从数据库中获取权限信息，API网关将日志记录到数据库中，负载均衡模块从数据库中获取负载均衡策略，服务注册中心向数据库中记录服务实例的信息，管理模块通过数据库对各类信息进行查看和管理。

本系统使用关系型数据库来实现该模块，数据库的具体设计见4.4节。

4.3 数据库设计

4.3.1 实体-联系模型设计

图4-3为系统数据的实体关系图，图中给出了系统中主要的实体，分别为用户、服务、角色、接口、令牌、服务实例和统计数据。

图4-3 平台管理系统E-R图

其中用户拥有角色，角色代表访问一类资源的权限，用户与角色是多对多的关系；用户发布服务，发布关系是多对一的关系，每个服务只由一个用户发布；服务可以启动多个服务实例，其中服务实例是一个弱实体，是启动关系的全部参与者，每个服务实例由端口号、IP和服务ID来标识；用户生成令牌用来进行身份验证，令牌是弱实体，是生成关系的全部参与者，每个令牌由用户ID和令牌值标识；令牌授权服务进行接口调用，令牌与服务是多对多的关系，一个令牌可以颁发给多个服务，一个服务也可以获取多个令牌；服务开放接口，接口与服务是多对一的关系；接口对应一组角色，用来表示可以调用该接口的角色，接口与角色是多对多的关系；接口生成统计数据，统计数据是弱实体，是生成关系的全部参与者，统计数据由接口ID和统计日期标记。图中接口路径和调用次数是非键属性，其他非键属性没有在图上标明。

根据如上的分析结果，下一节进行详细的数据库模式的设计。

4.3.2 数据库模式设计

根据上述E-R表，可以设计出系统涉及的数据库模式，下面将描述每张表和其约束关系。

表4-1展示了用户表user的结构，该表主要用于登录系统。

表4-1 用户表user

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| user\_id | bigint(20) | 主键 | 否 | 用户ID |
| name | varchar(20) |  | 否 | 用户姓名 |

表4-1（续） 用户表user

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| username | varchar(64) |  | 否 | 系统登录用户名 |
| password | varchar(64) |  | 否 | 系统登录密码 |

表4-2展示了角色表的结构，其中角色值为前缀为“ROLE\_”的字符串，如“ROLE\_ADMIN”。

表4-2 角色表role

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| role\_id | bigint(20) | 主键 | 否 | 角色ID |
| role\_value | varchar(20) |  | 否 | 角色值 |
| role\_describe | varchar(200) |  | 否 | 角色描述 |

表4-3展示了用户角色分配表，表示用户角色的分配关系，其中user\_id和role\_id共同构成表的主键，而user\_id是指向用户表的外键，role\_id是指向角色表的外键。

表4-3 用户角色分配表user\_role

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| user\_id | bigint(20) | 主键、外键 | 否 | 用户名 |
| role\_id | bigint(20) | 主键、外键 | 否 | 用户拥有的角色名 |
| create\_time | datatime |  | 否 | 创建时间 |

表4-4展示了令牌表，令牌由用户登录系统由授权服务器生成，然后分发给服务进行接口的授权调用，其中user\_id和token\_value共同构成表的主键，user\_id为指向用户表的外键，表明生成该令牌的用户。

表4-4 令牌表token

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| user\_id | bigint(20) | 主键、外键 | 否 | 用户名 |
| token\_value | varchar(100) | 主键 | 否 | 令牌值 |
| create\_time | datetime |  | 否 | 创建时间 |
| invalid\_time | datatime |  | 否 | 令牌到期时间 |
| refresh\_token | varchar(100) |  | 是 | 刷新令牌 |

表4-5展示了服务表，其中user\_id为指向用户表的外键，表示发布该服务的用户。

表4-5服务表service

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| service\_id | bigint(20) | 主键 | 否 | 服务ID |
| user\_id | bigint(20) | 外键 | 否 | 发布服务的用户 |
| service\_path | varchar(50) |  | 否 | 服务路径 |
| service\_describe | varchar(200) |  | 否 | 服务描述 |
| publish\_time | datetime |  | 否 | 发布时间 |
| lb\_strategy | varchar(20) |  | 是 | 负载均衡策略 |

表4-6展示了授权表，其中service\_id和token\_value共同构成了该表的主键，service\_id为指向服务表的外键，表示得到授权的服务；token\_value为指向令牌表的外键，表示代表授权的令牌；user\_id为指向用户表的外键，表示分发该令牌的用户。

表4-6 授权表authority

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| service\_id | bigint(20) | 主键、外键 | 否 | 服务ID |
| token\_value | varchar(100) | 主键、外键 | 否 | 令牌值 |
| user\_id | bigint(20) | 外键 | 否 | 用户ID |

表4-7展示了接口表：

表4-7 接口表interface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性名称 | 属性类型 | 约束条件 | 允许为空 | 描述 |
| interface\_id | bigint(20) | 主键 | 否 | 接口ID |
| service\_id | bigint(20) | 外键 | 否 | 接口所属的服务ID |
| interface\_path | varchar(100) |  | 否 | 接口路径 |
| interface\_describe | varchar(200) |  | 否 | 接口描述 |

表4-8展示了访问控制表，每个接口通过赋予的角色来决定调用者所应具有的权限，只有具有相应权限的服务才能调用接口。该表中interface\_id和role\_id共同构成表的主键，其中interface\_id为指向接口表的外键，表示需要调用权限的接口，role\_id为指向角色表的外键，表示接口所需要的权限。

表4-8访问控制表require\_auth

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 属性名称 | | 属性类型 | | 约束条件 | | 允许为空 | 描述 |
| interface\_id | bigint(20) | | 主键，外键 | | 否 | | 需要权限的接口ID |
| role\_id | bigint(20) | | 主键，外键 | | 否 | | 接口所需的权限角色ID |

表4-9展示了服务实例表，该表包含了实例运行的相关信息，由ip，port和service\_id共同构成该表的主键，service\_id是指向服务表的外键，表示实例所属的服务。

表4-9服务实例表service\_instance

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 属性名称 | | 属性类型 | | 约束条件 | | 允许为空 | | 描述 |
| ip | varchar(50) | | 主键 | | 否 | | 实例ip地址 | |
| port | bigint(6) | | 主键 | | 否 | | 实例端口号 | |
| service\_id | bigint(20) | | 主键，外键 | | 否 | | 实例所属的服务ID | |
| health\_url | varchar(100) | |  | | 否 | | 健康检查url | |
| is\_visible | varchar(10) | |  | | 否 | | 是否对外发布 | |

表4-10展示了统计数据表，数据统计表用于统计接口每日调用的次数以及统计的日期：

表4-10 统计数据表stats

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 属性名称 | | 属性类型 | | 约束条件 | | 允许为空 | | 描述 |
| interface\_id | bigint(20) | | 主键、外键 | | 否 | | 统计的接口ID | |
| stats\_date | date | | 主键 | | 否 | | 统计日期 | |
| count | bigint(20) | |  | | 否 | | 接口的调用次数 | |

4.4 接口设计

本节主要描述主要模块的接口设计，API网关，服务注册中心和授权认证模块提供REST API接口，负载均衡模块和熔断器作为库提供函数接口，数据库通过Spring Data JPA提供数据操作接口。

4.4.1 API网关核心模块

API网关核心核心模块作为调用服务的唯一入口，其调用一个服务的接口如下：

[HTTP Method] http://gatewayhost:port/apigateway/service\_id/path\_to\_service

该接口的具体描述如下：

1. gatewayhost: 网关的ip地址，port为网关的端口号；
2. /apigateway: 为所有服务的api的调用根路径，通过添加该前缀可以将所有由API网关管理的服务实例地址统一在一个命名空间进行管理，减少和API网关其他有关系统信息的接口发生冲突，增加服务命名灵活性；
3. service\_id: 服务的名称，为服务在服务注册中心注册的ID，所有服务实例共享同一个服务ID，API网关使用路由映射将gatewayhost:port/apigateway路径转换为服务实例的路径进行接口调用；
4. path\_to\_service: api的调用路径，为api在服务实例上的调用路径，API网关用查找到的实例路径和path\_to\_service组合成最终的调用路径对接口进行调用；
5. 调用还需要将持有的令牌作为HTTP请求报文头部字段Authorization的值和调用请求一并发送，令牌供API网关用来实现身份认证与权限控制的功能；
6. 调用返回的状态码表示响应状态，具体见表4-11：

表4-11 API网关接口调用返回的状态码示例

|  |  |  |
| --- | --- | --- |
| 状态码 | 适用HTTP动词 | 描述 |
| 200 OK | GET | 服务器成功返回数据 |
| 201 CREATED | POST/PUT/PATCH | 用户创建或修改数据成功 |
| 204 NO CONTENT | DELETE | 用户删除数据成功 |
| 401 Unauthorized | 所有 | 身份验证未通过 |
| 403 Forbidden | 所有 | 无权访问资源 |
| 404 NOT FOUND | 所有 | 访问的资源不存在 |

4.4.2 负载均衡模块

负载均衡模块作为库的形式由API网关进行调用，其主要目的是根据一定的负载均衡策略选择一个服务实例进行调用，其接口如下：

Instance chooseInstance(String service\_id, String load\_balancing\_strategy);

其中Instance封装了服务实例的有关信息，从而API网关可以用来组合成服务实例的地址，service\_id为API网关接口中的service\_id，表示API网关请求服务模块在相应的服务实例中选择一个实例返回。load\_balancing\_strategy为负载均衡策略，包括RoundRobin和随机策略等，负载均衡模块根据所调用服务配置的策略在可用的服务实例列表中选择一个实例进行返回。

4.4.3 熔断器

熔断器本质上是服务调用的代理，通过对实际的服务调用进行封装，熔断器可以执行服务熔断，服务降级和线程隔离的策略。因而熔断器和服务调用采用同样的接口：

ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?> entity,

Class<T> responseType, Object… args)

该接口通过调用服务，对HTTP响应报文进行封装返回，该接口的返回类型和参数描述如下：

1. ResponseEntity<T>：返回类型，作为对HTTP响应报文的封装类，参数T表明响应报文body中的JSON数据应反序列化为T类型的对象；
2. url：服务实例的地址，包括ip地址、端口号、调用路径和调用参数；
3. method：HTTP调用方法，通常有GET, POST, PUT, DELETE等；
4. entity：对GET和POST方法用于构造HTTP请求报文；
5. responseType：用于将请求报文序列化；
6. args: 用于将调用参数映射到url中。

4.4.4 服务注册中心

服务注册中心作为系统的核心组件，提供了服务查询、服务实例查询、服务信息查询、服务注册、服务注销等REST API，其具体的接口和描述如表4-12所示。

表4-12 服务注册中心REST API

|  |  |  |
| --- | --- | --- |
| REST API | 操作描述 | 输入输出描述 |
| POST /registry/services/{serviceId} | 注册新的服务实例 | 输入json描述服务元数据，注册成功返回HTTP 204状态码 |
| DELETE　/registry/services/{serviceId}/{instanceId} | 注销服务实例 | 成功返回HTTP 200状态码 |
| GET /registry/services | 查询所有服务及实例信息 | 输出json格式列表，成功返回HTTP 200状态码 |
| GET /registry/services/{serviceId} | 查询指定服务信息 | 输出json格式，成功返回HTTP 200状态码 |
| GET /registry/services/{serviceId}/{instanceId} | 查询指定实例信息 | 输出json格式，成功返回HTTP 200状态码 |
| PUT /registry/services/{serviceId}  /status?value=OUT\_OF\_SERVICE | 暂停服务所有实例 | 成功返回HTTP 200, 失败返回HTTP 500状态码 |
| PUT /registry/services/{serviceId}/{instanceId}  /status?value=OUT\_OF\_SERVICE | 暂停服务实例 | 成功返回HTTP 200，失败返回HTTP 500状态码 |
| DELETE /registry/services/{serviceId}/status | 恢复服务所有实例 | 成功返回HTTP 200，失败返回HTTP 500状态码 |
| DELETE /registry/services/{serviceId}/{instanceId}/status | 恢复服务实例 | 成功返回HTTP 200，失败返回HTTP 500状态码 |
| PUT /registry/services/{serviceId}  /metatdata?key=value | 更新服务元数据 | 成功返回 HTTP 200， 失败返回HTTP 500状态码 |
| PUT /registry/service/{serviceId}/{instanceId}  /metadata?key=value | 更新服务实例元数据 | 成功返回 HTTP 200， 失败返回HTTP 500状态码 |

4.4.5 授权认证模块

授权认证作为API网关中功能比较重要的一块，主要用于对用户的身份进行认证并且颁发令牌用作用户身份和权限的凭证，从而使得用户开发的应用可以直接凭借令牌来通过API网关调用所需的服务。API网关通过对令牌解析来完成对用户身份的确认，并通过将用户权限与服务调用权限作比较来决定是否允许用户的请求通过。授权认证供能按理说属于API网关中的一个子模块，但由于其重要性，这里和负载均衡一样，将其分离出来进行设计。

授权认证模块对外主要提供一个接口，通过API网关暴露为/apigateway/auth/token，需要调用其他服务的服务开发人员通过管理模块调用该接口获取代表服务开发人员的令牌，而服务开发人员所开发的服务则使用令牌来进行调用。该接口使用POST方法，接受使用JSON编码的方式进行用户名和密码的传输，格式如下：

{ “username” : “用户名”, “password” : “用户密码” }

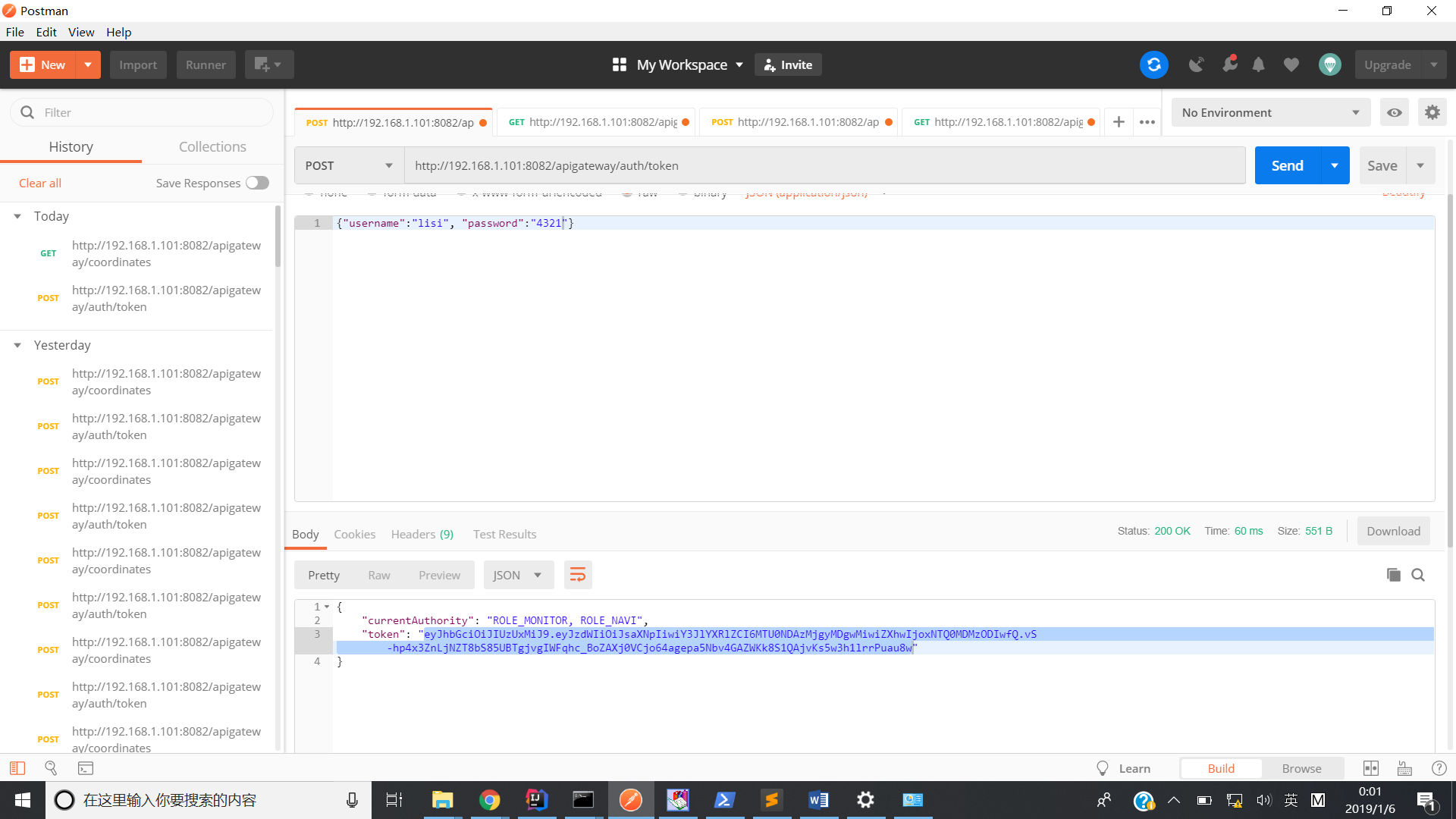
若用户输入的身份通过校验，授权认证模块返回以JSON格式编码的令牌，并告知用户所具有的权限，具体的格式见图4-4。

图4-4 授权认证模块返回的令牌数据格式

授权认证模块通过过滤器的形式与API网关核心模块对接，其接口形式为：

boolean isAuthorized(Token token);

该方法接受服务调用者传递进来的令牌，并且返回调用者是否具有权限调用请求的接口。

4.4 系统部署方案设计

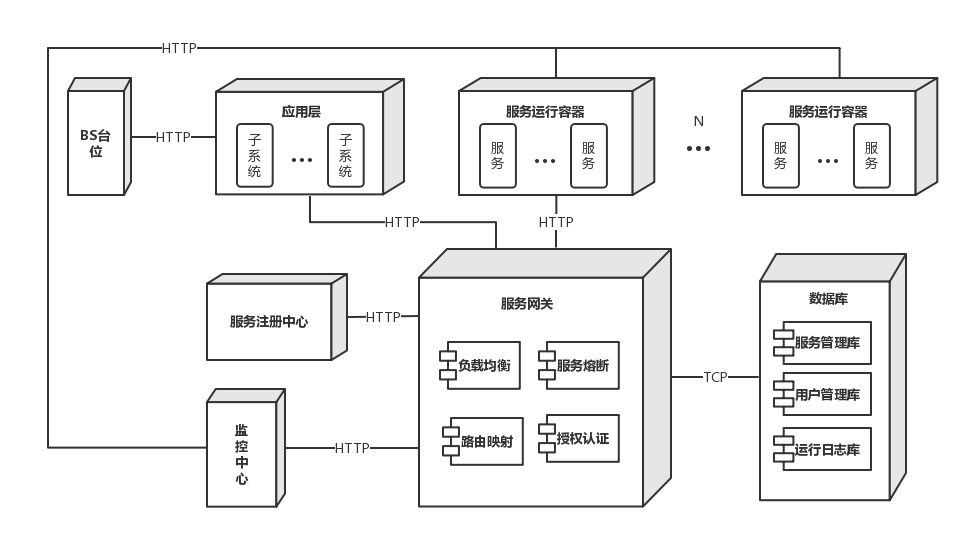
根据舰船计算环境的特点，舰船平台服务管理系统各个组件的部署方案设计如图4-5所示。

图4-5 舰船平台服务管理系统的组件部署方案

具体的部署方案如下：

1. 服务网关：与服务注册中心、监控中心通过HTTP协议进行通信，并与数据库系统通过TCP协议进行通信；
2. 监控中心：与服务运行容器进行通信来监控服务的运行状态，并与服务网关通信对外提供监控状态的查询；
3. 服务注册中心：主要提供服务的注册与发现功能，与服务网关通过HTTP协议进行通信，可独立部署，也可与服务网关部署至同一台物理设备。
4. 数据库系统：主要包括服务管理库、用户管理库和运行日志库等逻辑组件，是一个为平台提供数据的软件系统，通过TCP协议与服务网关进行通信。
5. 服务运行容器：是指服务部署时所需的中间件服务器，可独立部署。
6. B/S台位：是指管理类子系统的管护端，展示功能统一由浏览器支持并实现。与应用层的子系统采用HTTP协议进行通信。
7. 应用层：包括各子系统，监控类子系统的业务逻辑层，由各种服务协同实现。服务包括基础服务和业务服务，体现为微服务的方式，以HTTP的方式进行通信。

4.5 本章小结

本章首先给出了舰船平台服务治理方案，然后根据舰船平台服务管理系统的需求分析，对管理系统的核心模块进行了架构分析。同时，本章还设计了系统主要模块的接口，分析了系统中各实体之间的关系，设计了系统所需的数据库的表结构，并给出了系统所需的数据库表的字段、属性和描述信息。最后，本章给出来舰船平台服务管理系统的部署方案。

5 面向舰船平台管理系统的详细设计与实现

本章将使用流程图、类图和时序图来描述面向舰船平台管理系统核心模块的设计与实现，具体包括API网关、负载均衡模块、以及认证模块的设计与实现。

5.1 API网关的设计与实现

5.1.1 API网关的架构

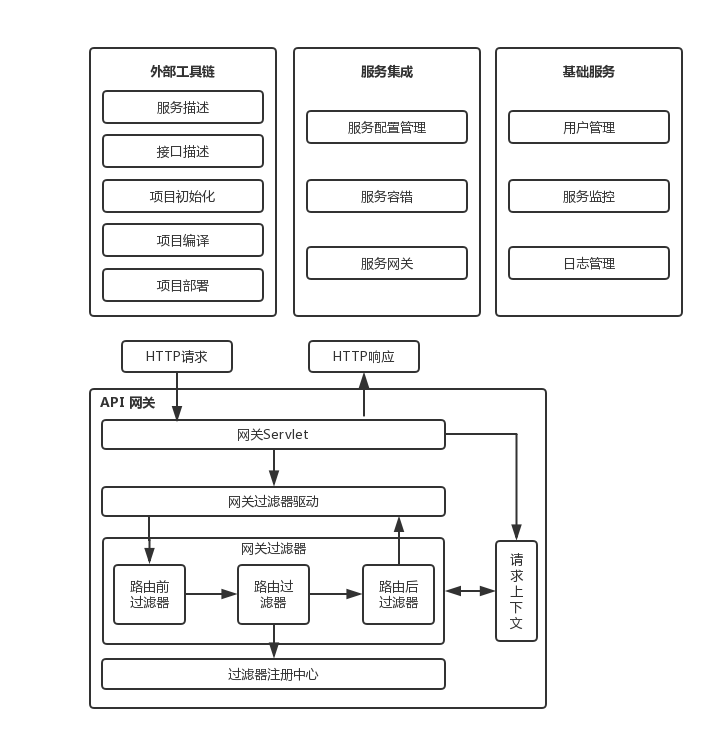
API网关的架构如图5-1所示。

图5-1 API网关架构

对其中的各个模块的描述如下：

1. 网关Servlet: 用于处理HTTP请求并对生成HTTP响应；
2. 网关过滤器驱动：驱动网关过滤器按路由前过滤器、路由过滤器和路由后过滤器的顺序执行；
3. 网关过滤器：网关过滤器是API网关的核心组件，负责实现API网管身份验证、路由映射、日志记录等功能，分为路由前过滤器、路由过滤器和路由后过滤器：
4. 路由前过滤器：负责实现路由映射前API网关所执行的功能的一组过滤器，主要实现身份验证和日志记录的功能，该模块负责与认证模块与日志库进行交互。
5. 路由过滤器：负责实现路由映射和服务调用，该模块与负载均衡模块进行交互，获取实例地址并调用目标服务。
6. 路由后过滤器：负责实现服务调用后API网关所执行的功能的一组过滤器，该模块实现调用统计与日志功能。
7. 请求上下文：作为网关过滤器实现信息共享的对象。网关Servlet将HTTP请求的相关信息封装到该对象中，例如可以将访问令牌封装到该对象中，供负责身份验证的路由前过滤器使用。该对象还管理各过滤器共享的信息，例如实现服务调用响应时间统计的调用时间就记录在该对象中。
8. 过滤器注册中心：负责存储API网关所使用的所有过滤器对象，在网关过滤器驱动进行对过滤器进行调用的过程中，首先在该注册中心中取出相关的过滤器，然后按顺序进行执行。

5.1.2 API网关的核心类设计

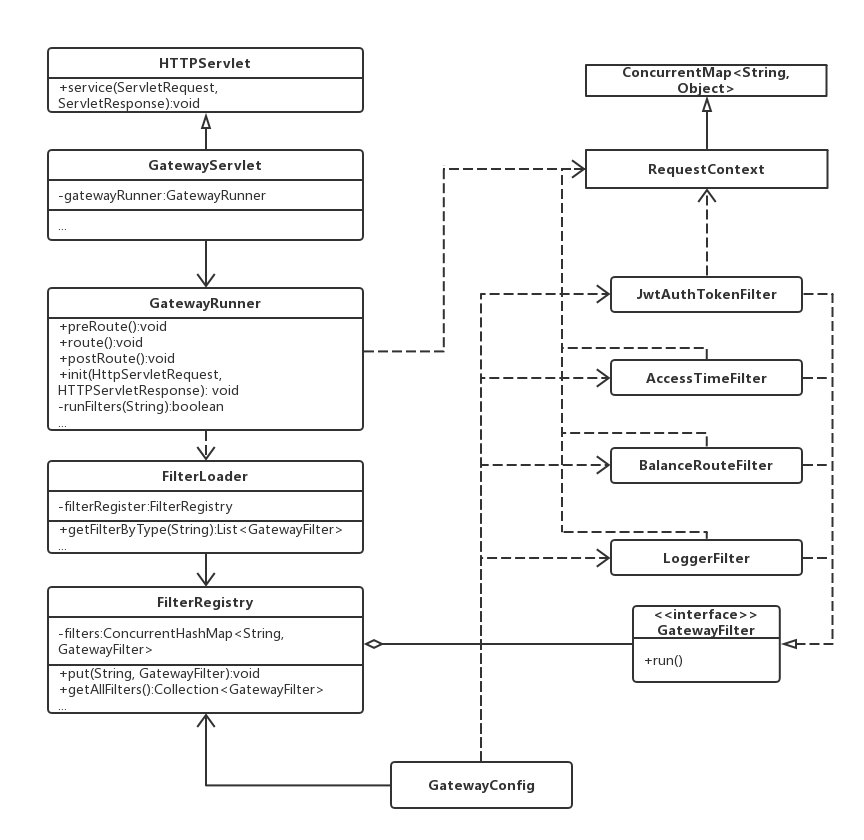
API网关的核心类及之间的关系如图5-2所示。

图5-2 API网关类图设计

其中GatewayConfig为网关的配置类，负责配置网关所需使用的bean，GatewayConfig在网关启动时实例化类JwtAuthTokenFilter, AccessTimeFilter, BalanceRouteFilter和LoggerFilter四个过滤器类。这四个类实现了GatewayFilter 接口的run方法，run方法用来实现每个过滤器的功能。其中JwtAuthTokenFilter和AccessTimeFilter为路由前过滤器，分别实现调用请求的身份认证功能和对调用时间进行统计。BalanceRouteFilter为路由过滤器，实现路由映射和服务调用功能，该类调用负载均衡模块对象方法进行服务实例的选择，然后调用断路器对象方法进行服务调用。LoggerFilter为路由后过滤器，负责实现日志功能，并和AccessTimeFilter一同实现服务响应时间的计算。GatewayConfig在初始化过滤器类的bean之后，将它们注册到FilterRegistry类中，FilterRegistry类使用一个ConcurrentHashMap<String, GatewayFilter>对象将各个过滤器bean按类别进行存放，使用字符串pre，route和post分别表示路由前过滤器、路由过滤器和路由后过滤器。

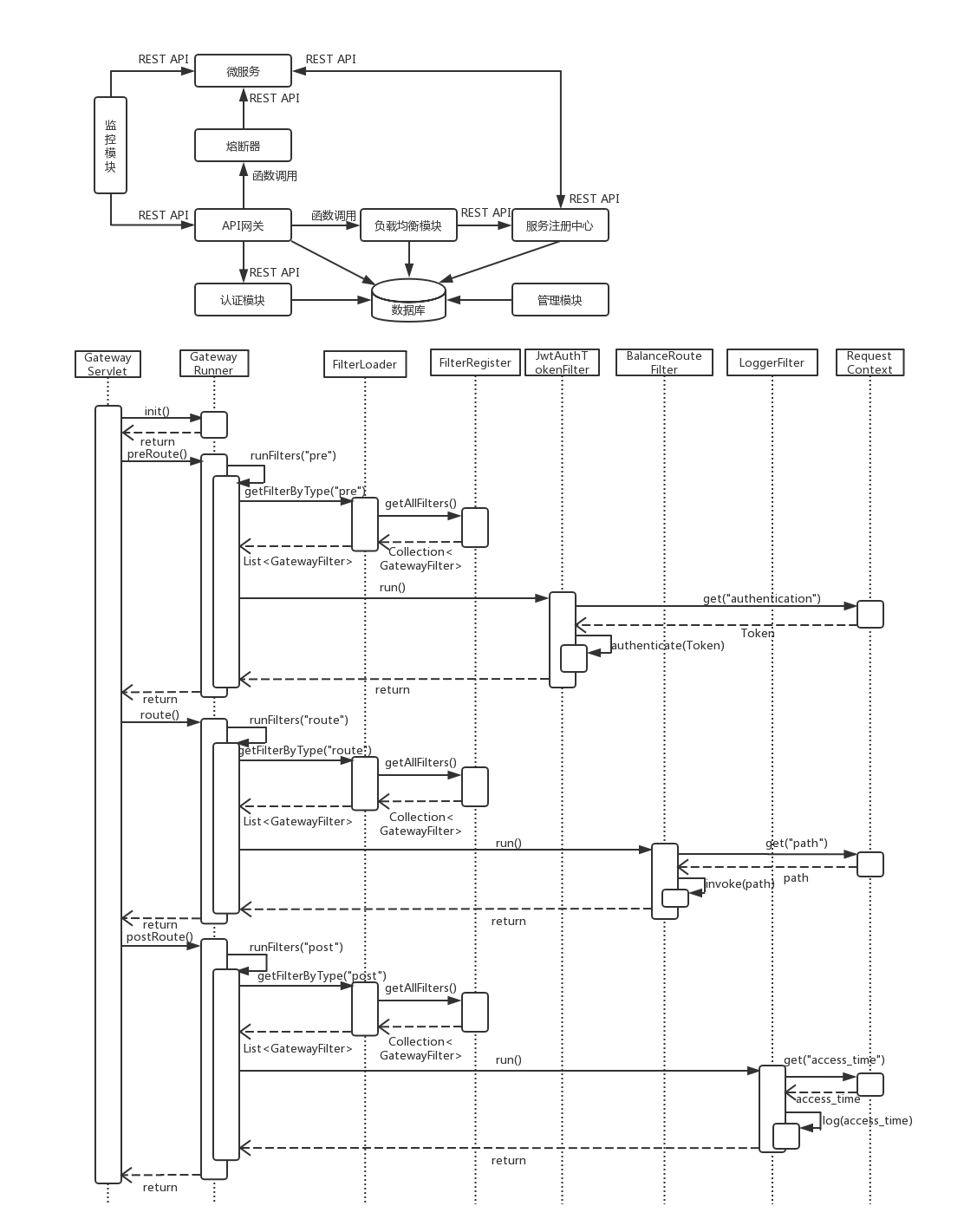
API网关处理服务调用请求的过程见图5-3：

图5-3 API网关处理服务请求时序图

GatewayServlet类即为网关Servlet，该类通过实现HttpServlet的service方法实现对网关调用请求的处理。GatewayServlet通过调用GatewayRunner类中的方法来按顺序执行过滤器和使用当前服务调用请求的信息初始化RequestContext。

RequestContext类是一个ConcurrentHashMap类，其负责将字符串映射到任意类对象。RequestContext对象除了映射HTTP请求报文头中的字段和值，还负责在各个过滤器之间传递消息。JwtAuthTokenFilter使用RequestContext对象中存储的HTTP报文头中的Authentication字段读取令牌信息，进行身份认证。BalanceRouteFilter读取报文头中的服务名进行服务映射。AccessTimeFilter和LoggerFilter通过在RequestContext中存放服务调用时间来计算服务响应时间。

GatewayRunner类方法说明如下：

1. preRoute(): 按顺序执行所有的路由前过滤器；
2. route(): 按顺序执行所有的路由过滤器；
3. postRoute(): 按顺序执行所有的路由后过滤器；
4. init(HTTPServletRequest, HTTPServletResponse): 初始化RequestContext对象；
5. runFilters(String): 调用所有由参数表示的类别的过滤器，其中参数去pre, route和post其中一个值，分别表示路由前过滤器、路由过滤器和路由后过滤器。preRoute, route和postRoute方法都使用runFilter来实现，例如，preRoute的实现：

public void preRoute() throws Exception{

if (!runFilters(“pre”)) throws Exception;

}

GatewayServlet实现的主要代码如下：

public void service (ServletRequest servletRequest, ServletResponse servletResponse) {

gatewayRunner.init((HTTPServletRequest) servletRequest,

(HTTPServletResponse) servletResponse);

gatewayRunner.preRoute();

gatewayRunner.route();

gatewayRunner.postRoute();

}

GatewayRunner的runFilters方法通过FilterLoader类的getFiltersByType实现，该方法通过调用FilterRegistry的getAllFilters方法获得注册表中的所有过滤器，然后根据参数所指定的类型对返回结果进行过滤，得到参数所指定的一类过滤器的列表，最后通过调用每个过滤器的run方法来执行其功能。

5.1.3 API网关过滤器实现

1. 日志过滤器AccessTimeFilter类

AccessTimeFilter为路由前过滤器，用来实现记录服务调用响应时间的功能，该过滤器和LoggerFilter一起来完成对响应时间的计算工作。主要在RequestContext中记录当前时间，从而让过滤器驱动在运行LoggerFilter的时候可以将RequestContext中记录的时间取出来，然后计算时间差即为服务调用的响应时间。

AccessTimeFilter的核心代码如下：

public Object run() {

RequestContext context = RequestContext.getCurrentContext();

context.set("requestTime", System.currentTimeMillis());

return null;

}

AccessTimeFilter通过实现GatewayFilter接口的run()函数来获得过滤器驱动的调用，而run()主要就是将系统时间记录到RequestContext中。

1. 日志过滤器LoggerFilter类

LoggerFilter类为路由后过滤器，该过滤器通过获取RequestContext中的信息，对服务的调用进行统计，并记录到日志库中去。

LoggerFilter的核心代码如下：

public Object run() {

RequestContext context = RequestContext.getCurrentContext();

HttpServletRequest request = context.getRequest();

String requestIP = HttpUtils.getIpAddress(request); // 获取请求的IP地址

String httpMethod = request.getMethod(); // 获取请求方法GET POST PUT Map<String, String> parameters = HttpUtils.getParams(request);

String pStr = parameters.toString(); // 获取请求参数

long requestTime = (long) context.get("requestTime"); // 获取接受请求时间

Throwable throwable = context.getThrowable(); // 获取服务调用异常

String requestURI = String request.getRequestURI()； // 获取请求的URI

String status = context.getResponseStatusCode(); // 获取请求的响应码

long responseTime=System.currentTimeMillis() - requestTime; // 计算响应时间

// 将相应的数据记录到数据库中去

return null;

}

可以看出LoggerFilter统计了GatewayServlet在RequestContext中存储的信息，包括请求方法、URI、IP、参数和响应的异常、响应码和响应时间，然后LoggerFilter可以将这些数据存储到数据库中去。

5.2 负载均衡模块的设计与实现

5.2.1 负载均衡模块的架构

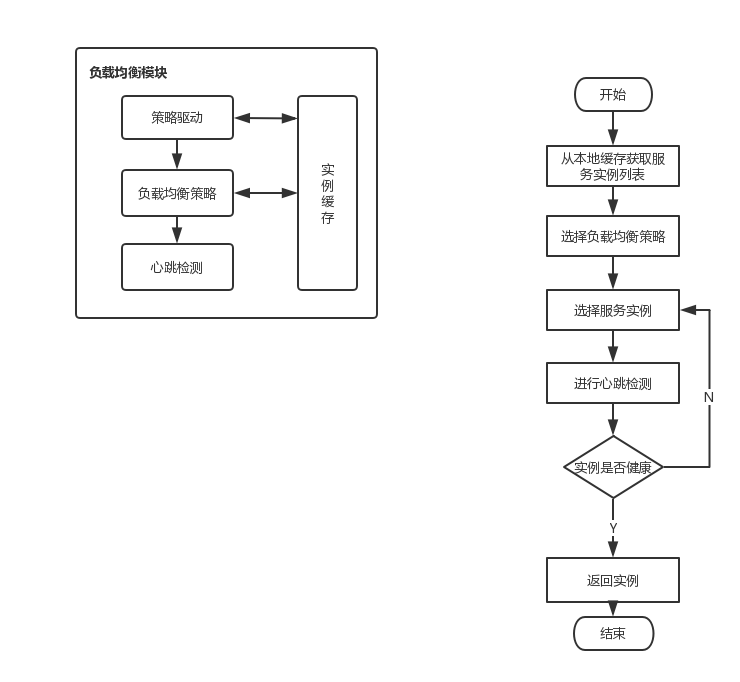
负载均衡模块的架构如图5-4所示。

图5-4 负载均衡模块架构

负载均衡模块主要由策略驱动、负载均衡策略、心跳监测和实例缓存四个子模块组成。其中策略驱动为模块入口，驱动其他子模块进行实例的选则。策略驱动用来初始化负载均衡策略列表，初始化服务实例缓存，从实例缓存中获取实例列表并实施负载均衡策略。负载均衡策略模块作为可配置的插件类，采用不同的算法实现了不同的负载均衡策略，客户端可以将服务与某个负载均衡策略绑定，使得每次对该服务的调用都会使用配置的负载均衡策略。心跳监测作为模块保证系统稳定性的一种检查机制，通过检查所选择的实例的健康状况，避免了返回的实例故障导致系统的整体性能下降。实例缓存模块负责从服务注册中心收集所有的服务实例，这是通过调服务注册中心的/registry/services接口来实现的，策略驱动首先根据所调用的服务从本地缓存中筛选所有服务的实例，然后再运行相应的负载均衡策略。

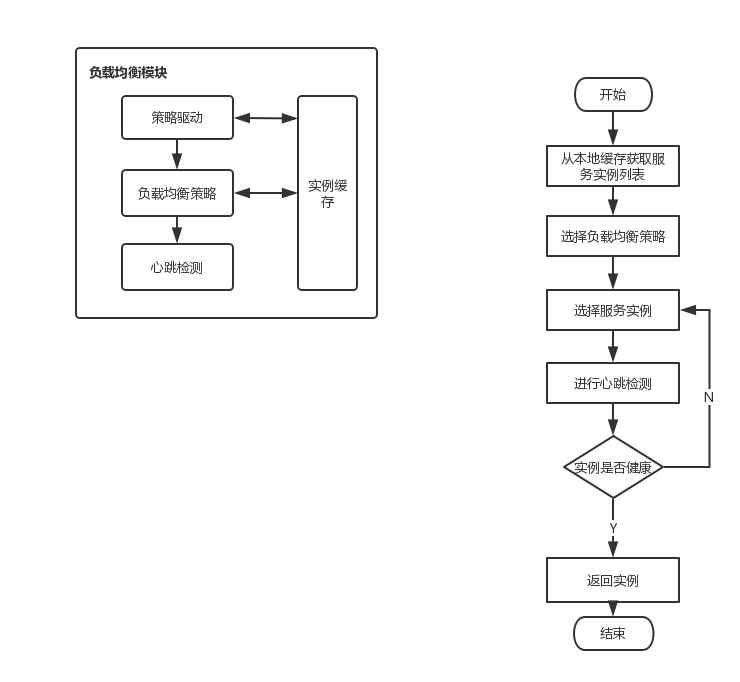
图5-5展示了负载均衡模块的工作流程。

图5-5 负载均衡模块工作流程图

5.2.2 负载均衡模块的核心类设计

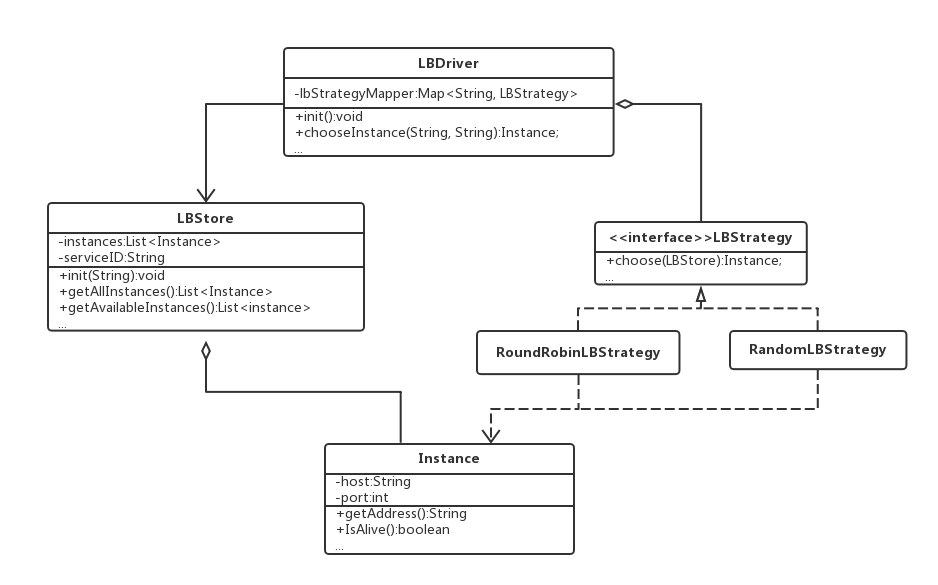
图5-6表示了负载均衡模块的核心类及相互关系。

图5-6 负载均衡模块类图

其中Instance类表示实例的地址，并负责实现心跳监测，其getAddress()方法返回ip:port形式的地址, isAlive()方法通过微服务暴露的健康接口来检查实例的健康情况。RoundRobinLBStrategy和RandomLBStrategy类继承了LBStrategy接口的choose方法，这两个类实现了具体的负载均衡策略，分别是轮转策略和随机策略，choose方法从可用的实例库中选择一个健康的实例返回给调用者。LBStore为缓存中当前服务的实例列表，其负责从本地缓存中获取所有有关服务的实例，并供负载均衡策略使用，其中init(String)方法通过服务名参数将instances成员指向缓存中服务的所有实例，getAvailableInstances()方法负责刷新本地缓存，该方法调用缓存的相关方法从注册中心中取出所有当前可用实例，getAllInstances()用于返回缓存中的所有实例。LBDriver为负载均衡算法的驱动类，是负载均衡功能的入口，负责初始化实例库对象和负载均衡策略对象，通过将负载均衡对象存放在一个Map中，允许客户端通过字符串的形式来选择使用的负载均衡策略，以达到负载均衡策略的可配置性。例如lbDriver.chooseInstance(“roundrobin”)就引导驱动对象执行轮转负载均衡策略。

负载均衡执行核心代码如下：

Instance choose(String serviceId, String lbStrategy) {

Instance instance = null;

LBStrategy strategy = lbStrategyMapper.get(lbStrategy);

LBStore lbStore = new lbStore();

lbStore.init(serviceId);

instance = strategy.choose(lbStore);

return instance;

}

5.2.3 负载均衡算法实现

1. 轮转算法：该算法从所有的实例中选择一个健康的实例返回给调用者[40]，算法的核心代码如下：

// 用来保存返回的实例

Instance instance = null;

// 用来控制查找的实例个数

int count = 0;

while(true) {

if (instance == null && count++ < 10) {

// 用于刷新本地缓存

List<Instance>availInstances = lb.getAvailableInstances();

List<Instance> allInstances = lb.getAllInstances();

int instanceCount = allInstances.size();

if (instanceCount != 0) {

// instanceIndex保存在类对象中，保证每次调用方法都取下一个实例

int nextIndex = moduloInc(instanceCount);

instance = (Instance)allInstances.get(nextIndex);

if (server == null) {

Thread.yield();

}

if (instance.isAlive())

return instance；

instance = null;

}

continue;

}

算法每次执行启用一个新线程，这样所有调用同一个服务的不同线程可以共用一个LBStrategy对象，从而实现调用该服务的负载均衡。代码中在调用getAllInstances()方法获取本地所有实例前先调用getAvailableInstances()方法刷新缓存，确保得到最新的实例，这样刚上线的实例也可以被及时的利用到。当本地实例列表为空时，算法直接返回空值，外层循环是当所选择的实例都无法进行调用（发生故障或没有处理资源）时，算法将重新选择一个实例，算法一共会尝试10次才放弃并返回空值。

由于算法运行在一个独立的线程里，计算选择实例的索引值需要考虑共享变量的互斥问题，因为所有的运行算法的线程通过同一变量来生成实例索引，只有这样才能保证所有的服务调用被均匀分发到不同的实例中，算法的moduloInc(instanceCount)实现了索引的计算，其代码如下：

int moduloInc(int count) {  
 int current;  
 int next;  
 do {  
 current = nextIndex.get();  
 next = (current + 1) % count;  
 } while(!nextIndex.compareAndSet(current, next));  
 return next;  
}

nextIndex是AtomicInteger变量，是所有线程共享的类成员变量，所以需要进行互斥的访问，这里使用AtomicInteger的compareAndSet()方法作为互斥访问原语对循环语句内的临界区操作进行互斥访问[41]。

1. 随机算法：采用随机数生成实例选择的索引，并在实例列表中选择一个实例返回[42]，核心代码如下(循环查询的代码已省略)：

List<Instance> allInstances = lb.getAllInstances();

int instanceCount = allInstances.size();

if (instanceCount == 0) {

return null;

}

//执行随机策略，random为类成员，初始化为new java.util.Random()

int index = random.nextInt(instanceCount);

instance = (Server)allInstances.get(index);

if (instance.isAlive()) {

return instance;

}

注意在随机策略下，系统不需要使用同一个变量来控制所有线程的实例选择决策，进程之间各子随机选择随机索引来选择服务实例，不需要使用共享变量，故不用使用互斥访问原语。

5.3授权认证模块的设计与实现

传统的服务安全实施方案是为每个接口进行权限校验，如果为了防止用户数据被篡改，则还要在接口调用处加上数字签名机制。在这种机制下应用程序自己承担起验证客户端发送的用户名和密码的责任，在微服务架构中，随着服务数量的增加，在每个服务中加入认证和权限校验的逻辑会增加代码的冗余，降低系统的可维护性。于是将授权认证单独抽取出来作为一个独立的微服务进行调用。

5.3.1 授权认证模块的架构

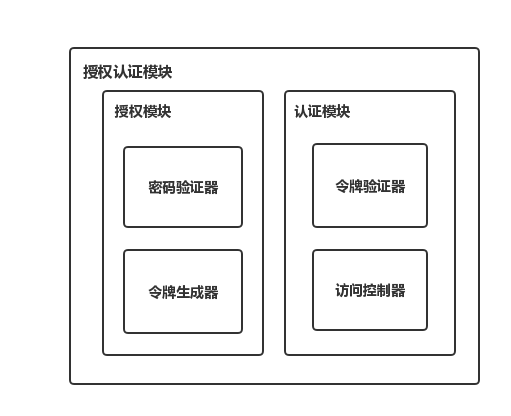
本系统的授权认证模块基于OAuth 2.0协议, 由于是用来对服务调用的应用进行身份验证与授权，故采用OAuth 2.0的客户端凭据授权许可类型。根据OAuth2.0协议对模块角色的划分，授权认证模块的架构如图5-7所示。

图5-7 授权认证模块架构

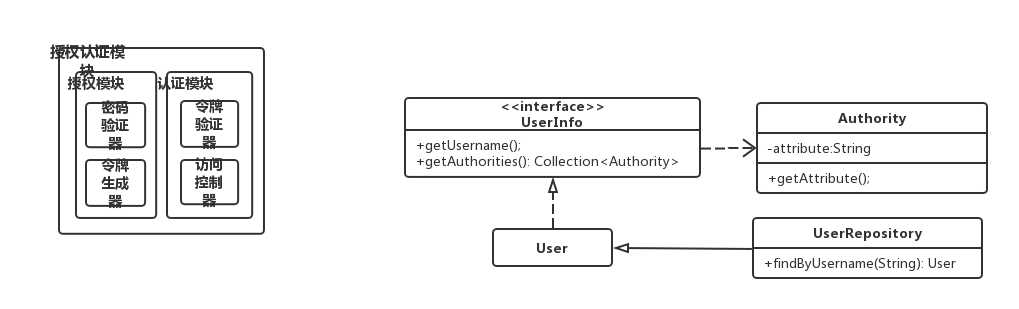
授权认证模块可以分为授权模块和认证模块，其中授权模块分为令牌生成器和密码验证器，认证模块分为令牌验证器和访问控制器，每个子模块的说明如下：

1. 密码验证器：用于请求生成令牌时对用户发送的用户名和密码进行校验；
2. 令牌生成器：密码验证器通过请求后，令牌生成器根据用户信息生成令牌，并将令牌存储在本地同时返回给用户；
3. 令牌验证器：对用户发送的令牌进行校验；
4. 访问控制器：根据用户请求的接口路径和从令牌获取的用户权限来拒绝或接受调用请求。

5.3.2 授权模块

1. 密码验证器

密码验证器比较简单，主要是将用户提供的用户名在数据库中查出相关的密码，进行加密处理后，然后将其与用户提供的密码进行比较。进行加密处理的原因是数据库中的密码在用户注册时已经经过加密处理，这样保证密码不会采用明文存储，提高密码的安全性。该模块在实现时使用到的实体类如图5-8。

图5-8 实现密码验证器用到的类图

其中User和UserRepository是Spring Data JPA的数据库ORM映射类，方便操作数据库操作，Authority类代表用户的角色，其中attribute的属性为“ROLE\_ADMIN”等表示角色的字段。UserInfo是表示用户信息的接口，将在后面的很多实现类中用到。

1. 令牌生成器

令牌生成主要依靠jjwt库来生成符合JWT标准的令牌，生成令牌的代码如下所示：

Jwts.*builder*().setClaims(claims)

.setExpiration(generateExpirationDate())

.signWith(SignatureAlgorithm.*HS512*, secret)

.compact();

其中claims是Map<String, Date>类，用来存放用户获取token的日期，jjwt库将获取token的日期，token的过期时间按JWT令牌格式通过加盐后使用HS512数字签名算法进行加密，生成最终的令牌。

5.3.3 认证模块

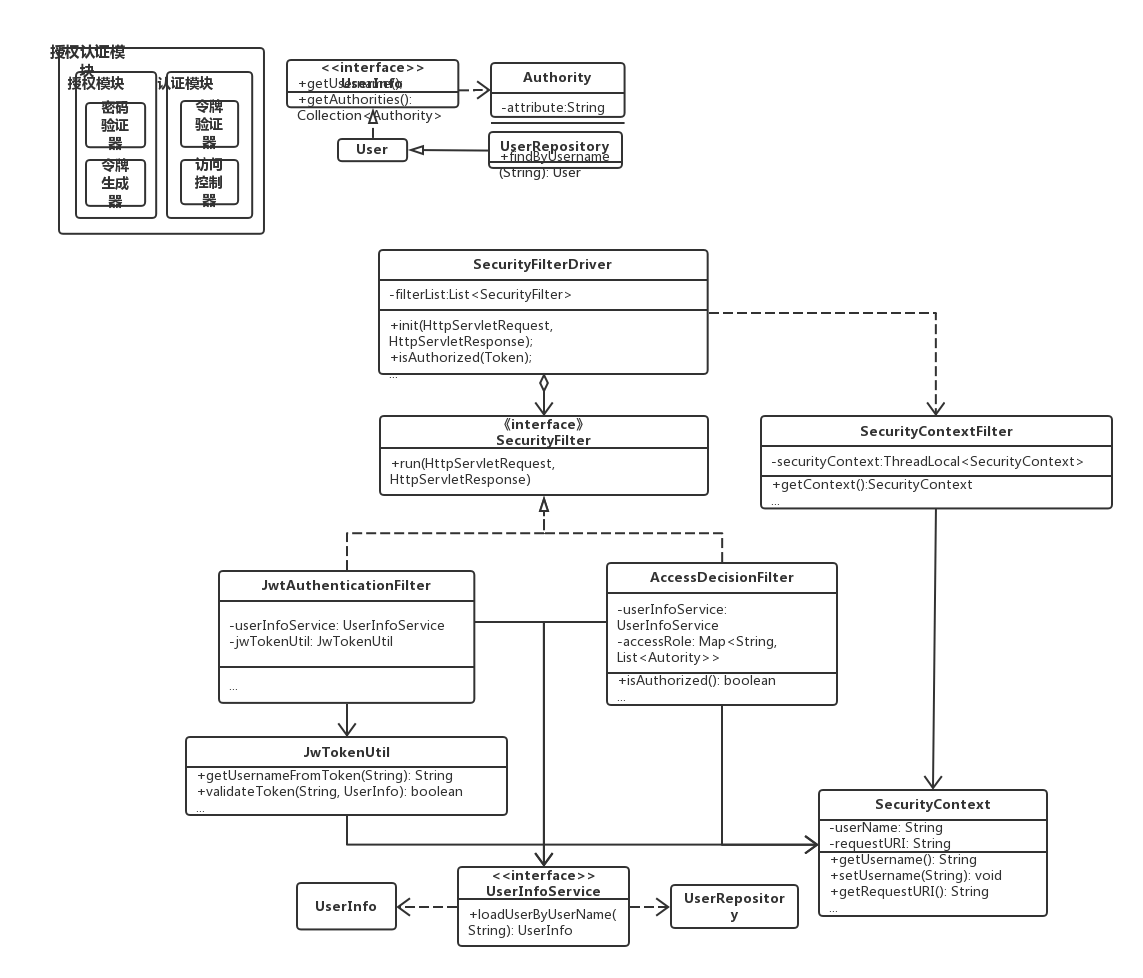
认证模块的类图如图5-9所示。

其中SecurityFilterDriver类为整个认证模块的入口，驱动分别实现令牌验证器和访问控制器的实现类JwtAuthenticationFilter和AccessDecisionFilter。SecurityFilterDriver类负责创建SecurityContext, 该类对象被封装在ThreadLocal类中，作为线程共享变量由于在两个过滤器之间传递数据，充当全局变量。

JwtAuthenticationFilter和AccessDecisionFilter两个过滤器类继承SecurityFilter接口，实现run()方法供SecurityFilterDriver类进行调用。下面分别对这两个过滤器类进行介绍：

1. JwtAuthenticationFilter类

JwtAuthenticationFilter类实现了令牌验证的功能，其有两个成员变量，jwTokenUtil成员将jjwt库的相关方法进行封装来实现JwtAuthenticationFilter的成员函数validateToken(), validateToke()函数通过将接收一个令牌和UserInfo对象来判断令牌是否过期以及令牌是否是为该用户颁发的，validateToken()的代码如下：

图5-9 认证模块的类图描述

public boolean validateToken(String token, UserInfo userInfo) {  
 User user = (User) userInfo;  
 String username = getUsernameFromToken(token);  
 Date created = getCreatedDateFromToken(token);  
 return (username.equals(user.getUsername()) && !isTokenExpired(token)；  
}

该方法从token中解析出用户名和令牌生成日期，然后判断用户身份以及令牌是否过期，对令牌的解析是由getUsernameFromToken和getCreatedDateFromToken两个函数进行完成的，getUsernameFromToken的代码如下：

public String getUsernameFromToken(String token) {  
 Claims userClaims = Jwts.*parser*().setSigningKey(secret)

.parseClaimsJws(token).getBody();

String username = userClaims.getSubject();  
 return username;  
}

其中parser，setSigningKey和parseClaimsJws方法为jjwt库方法，首先将数据进行解密，然后提取相应的字段。

JwtAuthenticationFilter工作的核心代码如下：

// 提取HTTP请求报文头中Authorization字段的值

String authHeader = request.getHeader(“Authorization”);

if (authHeader != null && authHeader.startsWith(“Bearer”)) {

// 获取Token

String authToken = authHeader.substring(tokenHead.length());

// 从token中提取username

String username = jwtTokenUtil.getUsernameFromToken(authToken);

if (username != null) {

UserInfo user = userInfoService.loadUserByUsername(username);

if (jwtTokenUtil.validateToken(authToken, user)) {

SecurityContextHolder.getContext()

.setUsername(username);

}

}

}

JwtAuthenticationFilter首先从HTTP请求中提取出Authorization头字段的值，然后从中抽取出令牌值，并从令牌值中提取出用户名。然后JwtAuthenticationFilter用用户名及userInfoService成员找出用户名的相关信息user，通过将用户信息和令牌使用validateToke方法进行验证，如果验证通过则将username保存在SecurityContext中以供访问控制的实现类AccessDecisionFilter使用。

1. AccessDecisionFilter类

该类实现了访问控制的有关逻辑，具体是使用由JwtAuthenticationFilter放入SecurityContext中已经通过验证的用户名，从数据库中取出用户的权限信息。而AccessDecisionFilter类在初始化的过程中会初始化成员accessRoleMap，这是一个Map类，将一个接口的访问路径映射到可以调用该接口的一组角色上。

AccessDecisionFilter类的核心代码如下：

boolean isAuthorized() {

SecurityContext securityContext = SecurityContextHolder.getSecurityContext();

String username = securityContext.getUsername();

UserInfo userInfo = userInfoService.loadUserByUsername(username);

// 获取用户权限

List<Authority> userAuthorities = userInfo.getAuthorities();

String uri = securityContext.getRequestURI();

// 获取访问接口uri所需的权限

List<Authority> requiredAuthories = accessRoleMap.get(uri);

// 查看用户是否由权限访问接口

for (Authority authority: userAuthorities) {

if (requiredAuthories.contains(userAuthorities))

return true;

}

return false;

}

代码中securityContext中存放的username由JwtAuthenticationFilter类放入，表示通过验证的用户名。

5.4 本章小结

本章对舰船平台服务管理系统的核心模块进行了详细的设计与实现，具体包括API网关的设计，API网关采用过滤器进行具体功能的实现，在服务调用前、中、后期加入过滤器来进行身份验证、路由映射和日志统计，这样做增加了API网关功能扩展的能力；负载均衡模块也采用了这种插件式思想，通过抽象的接口将负载均衡策略管理起来，使得用户能够根据配置来选择服务调用的负载均衡算法，这样用户可以根据不同服务的负载模式来选择合适的算法来平衡服务器的使用；本章最后对系统用来进行身份认证和访问控制的授权认证模块进行设计与实现，该模块采用令牌机制将用户的服务调用权限赋

予应用，并通过该令牌来对应用进行身份认证和控制访问请求。

6 面向舰船平台管理框架的测试与验证

本章对第五章的系统原型实现进行测试，首先对测试环境进行描述，然后按模拟测试场景对系统原型进行测试，主要进行以下验证：

1. 服务网关模块运行良好，能够完成对系统中注册的服务进行调用；
2. 负载均衡模块能够将请求分发到不同的服务实例；
3. 服务认证模块能够对用户身份进行认证；
4. 服务网关能够对拥有不同权限的调用请求进行识别，并且根据权限对服务调用请求进行过滤。

本章最后对实验结果进行了总结。

6.1应用场景仿真

本文基于通过真实的舰船应用场景，设计简单的虚拟实验场景对系统原型进行实验。该场景中，面向舰船平台管理系统的主要组件部署在独立的服务器上。该场景主要是分为控制台应用和后台服务，后台服务负责收集卫星数据，并对舰船当前坐标进行计算，后台服务的服务名为coordinates。控制台应用对坐标服务进行不间断的调用，以获取舰船的实时坐标。后台服务部署了三个实例，分别部署在独立的服务器上。具有独立的ip地址和端口号。控制台应用使用HTTP GET通过平台管理系统API网关暴露的地址http://gatewayhost:port/apigateway/coordinates进行后台数据的获取，其中gatewayhost和port是网关的ip地址和端口号。所有的服务都以jar包的形式部署在各自所在的节点上。

后台服务通过采集卫星数据进行计算来返回舰船的坐标值，这里使用简单的java类对象对后台服务生成的数据进行模拟，该包含了经度、维度和服务所部署的服务器ip地址，其代码如下：

Class Coordinates {

double longitude = 109.33

double latitude = 11.55

String ip = generateIp();

static String generateIp() {

String hp = null;

try {

hp = Inet4Address.getLocalHost().getHostAddress();

} catch (Exception) { }

return hp;

}

}

包含服务器ip地址主要为了测试负载均衡方便，与实际应用无关。

后台坐标服务直接返回一个简单的Coordinates对象，对象的序列化由Spring Boot框架的进行处理。后台服务的代码由Spring Boot开发，其核心代码如下：

@GetMapping(“/coordinates”)

Coordinates getCoordinates() {

return new Coordinates();

}

实验采用Postman软件对控制台应用进行模拟，Postman软件作为客户端调用控制台服务，并对返回结果进行解析显示，当进行负载均衡实验的时候，还会用到curl命令。

6.2验证环境

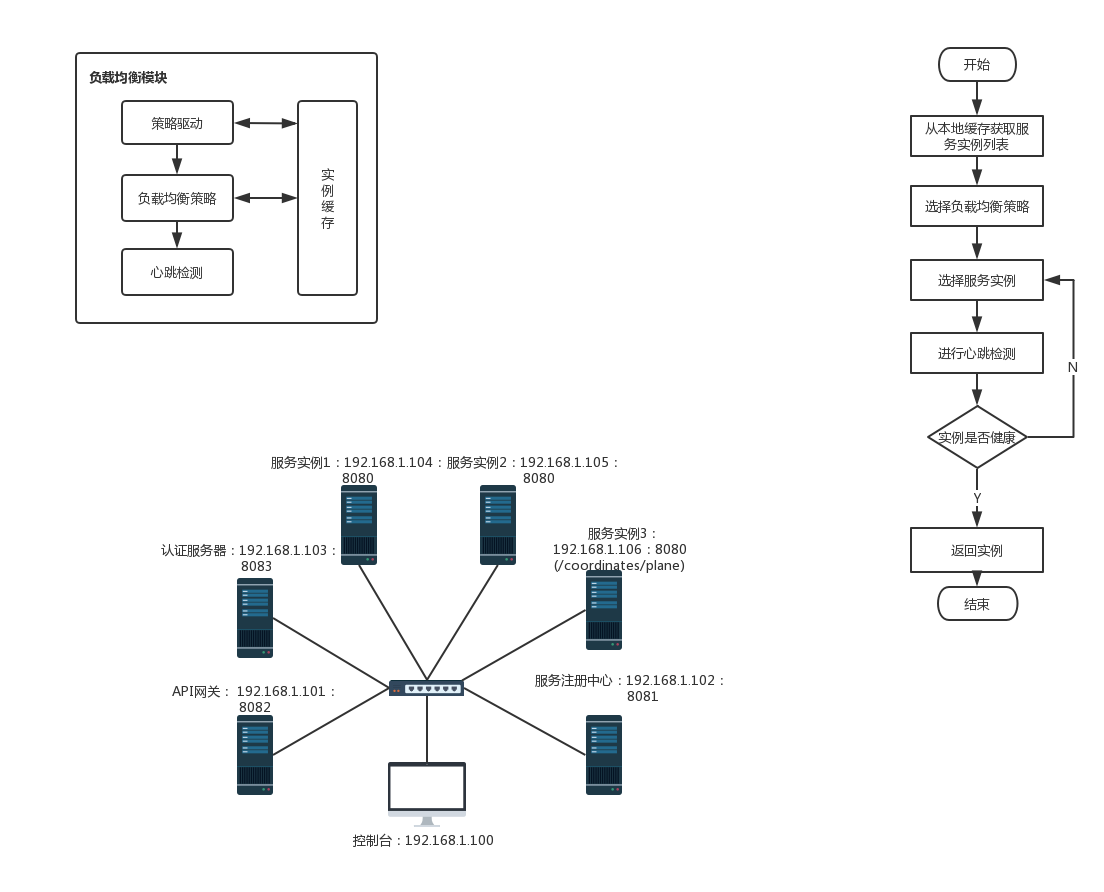
实验环境一共由七个节点构成，七个节点与真实舰船环境一样，处于同一个局域网内。除了作为模拟控制台的节点，其他节点均部署了一个服务。其中控制台的ip为192.168.1.100，实验的服务请求由该节点发出，coordinate 服务具有三个实例，分别部署在三个节点上，剩下的节点部署了系统原型的功能组件，实验环境的拓扑结构与服务的部署情况见图6-1。

图6-1 实验环境拓扑结构与服务部署图

实验使用的计算机环境配置如表5-1所示。

表6-1计算机环境配置表

|  |  |
| --- | --- |
| 系统软硬件 | 系统软硬件环境 |
| 操作系统 | Win10 64位专业版 |
| 内存 | DDR4 4G |
| CPU | 英特尔Core i7-6700 3.4GHz |
| 开发环境 | IntelliJ IDEA 2018.2.4 Ultimate Edition |
| Java版本 | JDK 8u181 |

6.3验证原理和验证方法

测试原理：通过设计多个实验用例对面向舰船平台管理系统的各个模块进行实验验证，通过观察实验结果与实验预期结果的差异，来证明系统原型是否达到了预期的设计效果。

测试方法如下：

1. 通过使用Postman模拟控制台应用对coordinates服务进行调用，然后查看返回结果，该测试测试了API网关的可用性；
2. 使用curl命令行对coordinates服务进行调用，然后查看返回结果，该测试测试了负载均衡模块的可用性；
3. 配置coordinates 服务使用的权限，通过模拟控制台应用对coordinates服务进行调用，然后查看返回结果，该测试测试了API网关能否正确拒绝未经授权的服务调用；
4. 配置用户权限，使其具有对coordinates服务调用的权限，使用Postman模拟身份认证，获取令牌，然后模拟控制台应用使用令牌访问coordinates服务，模拟获得授权的应用访问后台服务，查看返回结果，该测试测试了身份验证和访问控制的可用性；
5. 配置另一个用户的权限使其无权对coordinates服务进行调用，通过控制台使用该用户获取的令牌访问coordinates服务，模拟没有权限的应用访问后台服务，查看返回结果，该测试测试了身份验证和访问控制的可用性；
6. 将coordinates服务的权限添加给该用户，并使用Postman模拟身份认证，获取令牌，然后使用令牌模拟控制台应用访问coordinates服务，查看返回结果，该测试测试了身份验证和访问控制的可用性。

6.4验证流程及结论

在进行测试之前，要对部署的各个服务进行配置，主要是对系统的几个组件的地址进行配置，作为服务注册中心，每个部署的服务都要知道其地址才能在上面进行注册，需要进行访问控制的服务还要配置认证服务器地址，所有服务调用都要通过API网关，所以也要配置其地址。配置文件和应用程序一起打包，作为初始化变量进行读取。

coordinates的配置文件application.yml示例如下：

service.name = coordinates

registry.uri = http://192.168.1.102:8081/registry

gateway.uri = http://192.168.1.101:8082/apigateway

实验的具体流程如下：

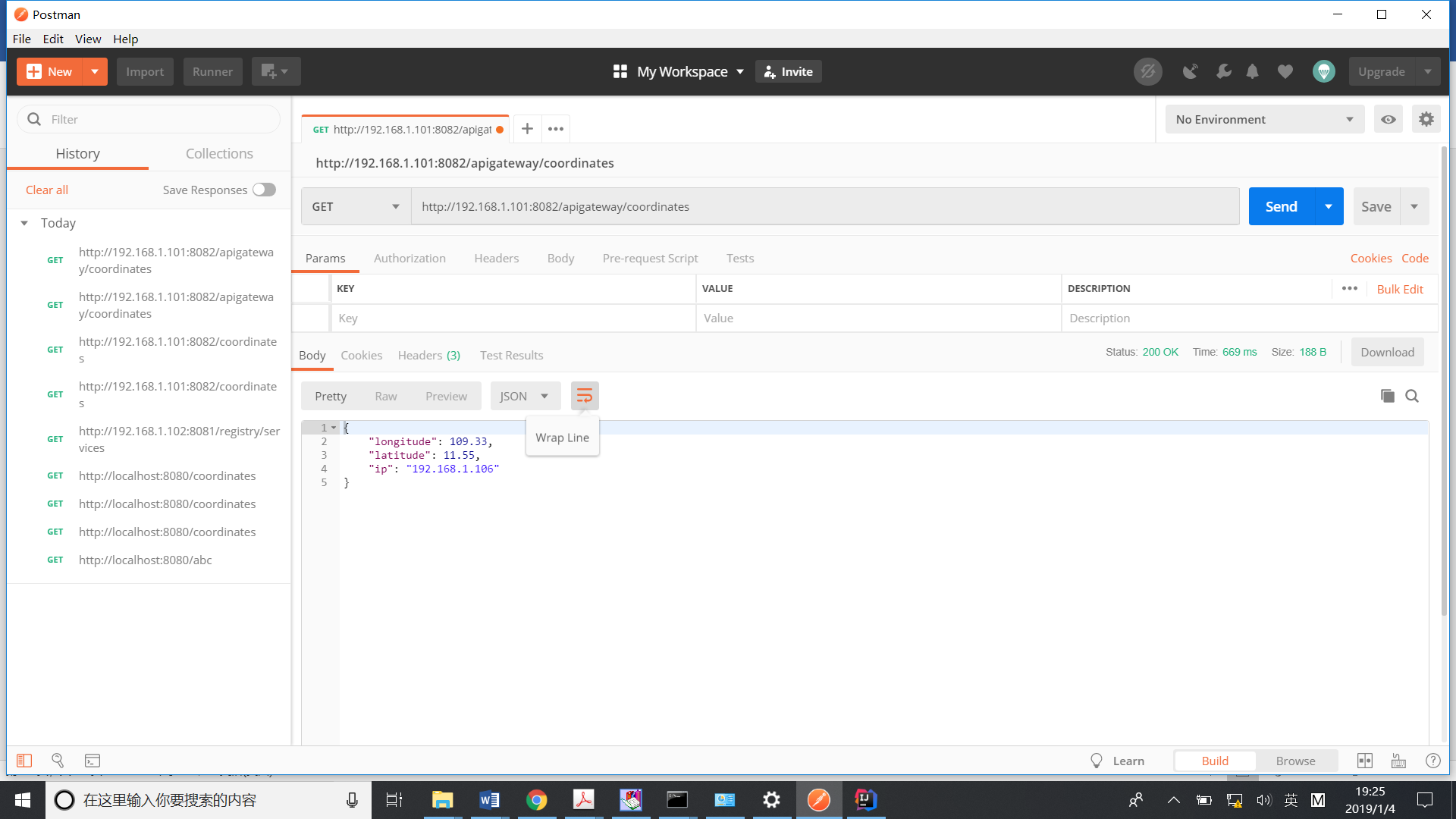
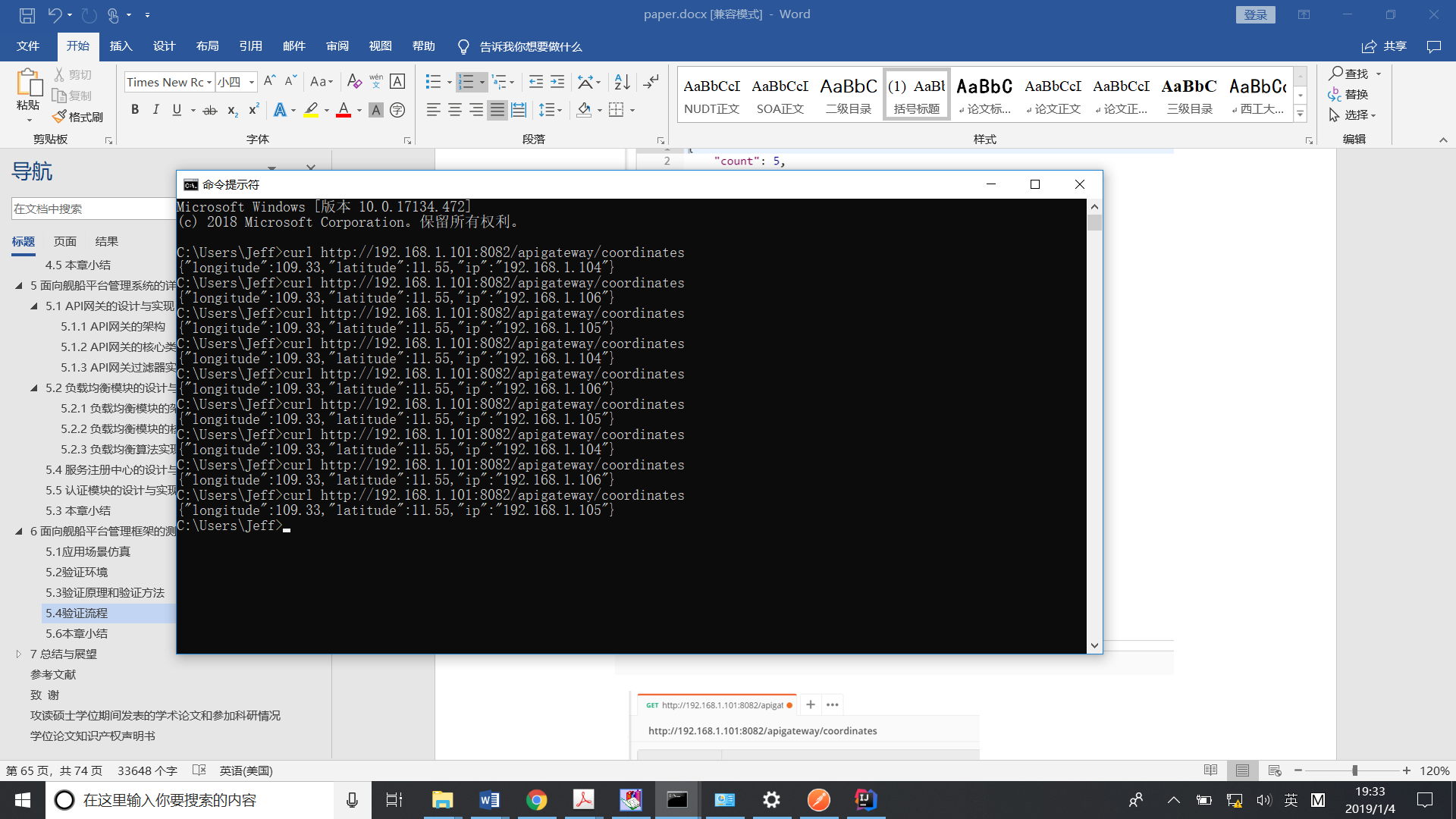
1. 使用Postman模拟通过api网关调用coordinates服务，api网关屏蔽了服务实例的接口，是调用方可以直接使用服务名来进行调用，api网关提供了统一的接口入口http://192.168.1.101:8082/apigateway，使用该入口与服务名拼接成的地址进行coordinates服务的调用，调用结果如图6-2所示。

图6-2 通过API网关调用coordinates服务返回的结果

返回结果是符合预期的，图中调用了coordinates服务位于192.168.1.106节点上的实例，该实验结果表明API网关能够正确的对服务调用请求进行路由映射，同时也说明了API网关与服务注册中心之间的通信正常，网关能从服务注册中心获取正确的服务实例地址。

1. 使用curl命令反复调用API网关暴露的coordinates服务地址，观察返回的实例地址变化。图6-3为实验结果。

由图可以看出，服务的反复调用被API网关映射到了不同的服务实例，服务的多次调用均匀的分布在三个实例上。API网关通过使用轮转策略调用负载均衡模块，实现了系统中实例负载的均匀分布，该实验结果证明API网关可以正确调用负载均衡模块，而且负载均衡模块也可以正常的进行工作。

图6-3 通过API网关反复调用coordinates服务的结果

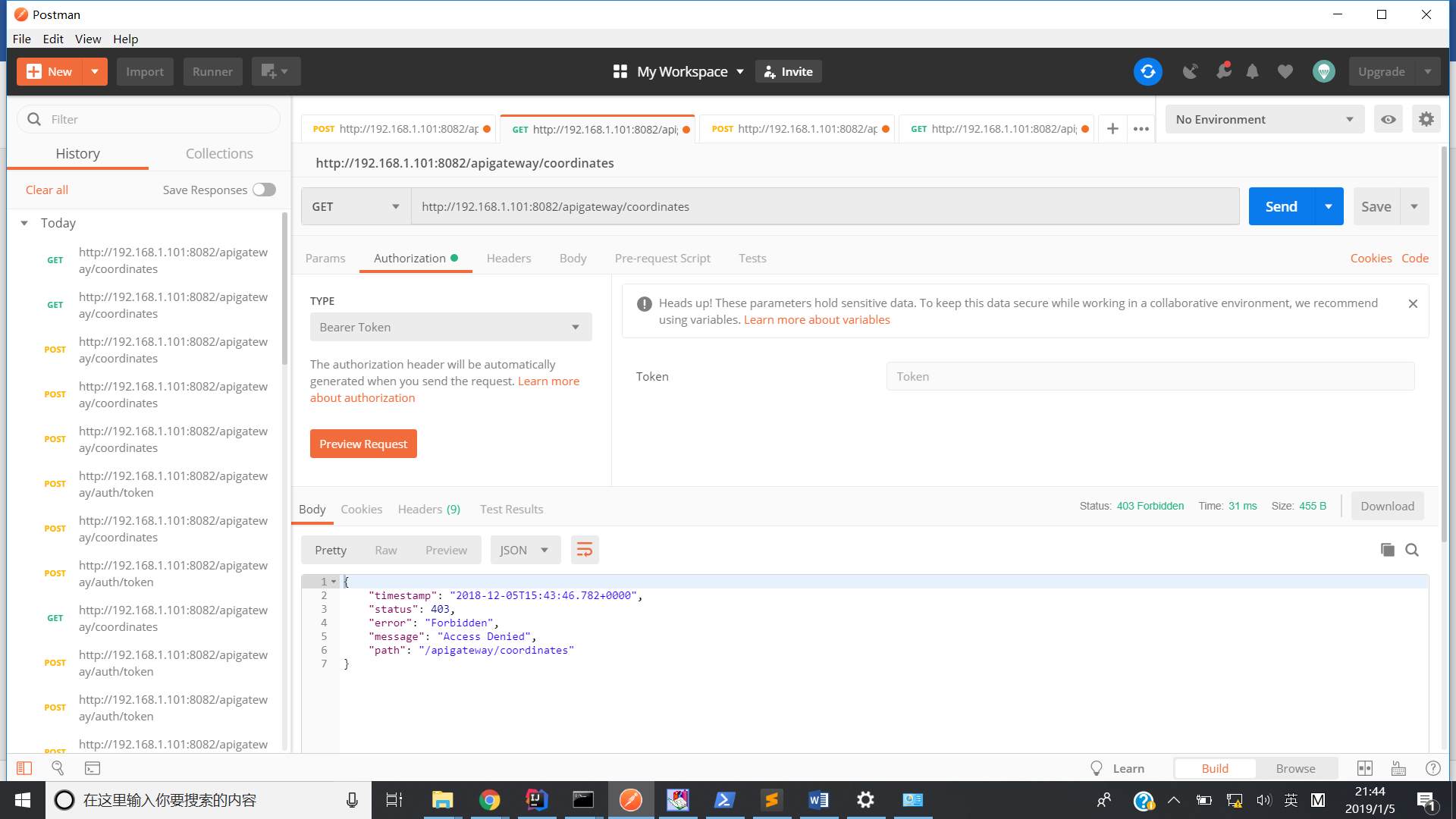
1. 配置数据库，创建角色ROLE\_MONITOR表示舰艇监控人员权限，并通过配置数据库中的权限表设置只有ROLE\_MONITOR能调用/coordinates接口，然后再通过Postman模拟控制台应用程序对coordinates服务进行调用，注意此时模拟的控制台应用程序是没有任何权限的。调用结果如图6-4所示。

图6-4 没用权限的情况下调用网关的结果

由图6-4可见调用结果符合预期，由于此时控制台应用程序(Postman)没有获取任何权限，所以API网关返回响应码403，表示请求服务的应用无权调用该服务。该实验结果表明API网关通过授权认证模块可以识别没有获得授权的服务访问请求并拒绝请求。

1. 创建用户张三，用户名为zhangsan，密码为1234，将ROLE\_MONITOR角色通过数据库赋给张三，然后模拟张三使用用户名和密码通过API网关向认证服务模块申请令牌，然后使用该令牌模拟控制台应用使用张三的权限调用coordinates服务，调用使用

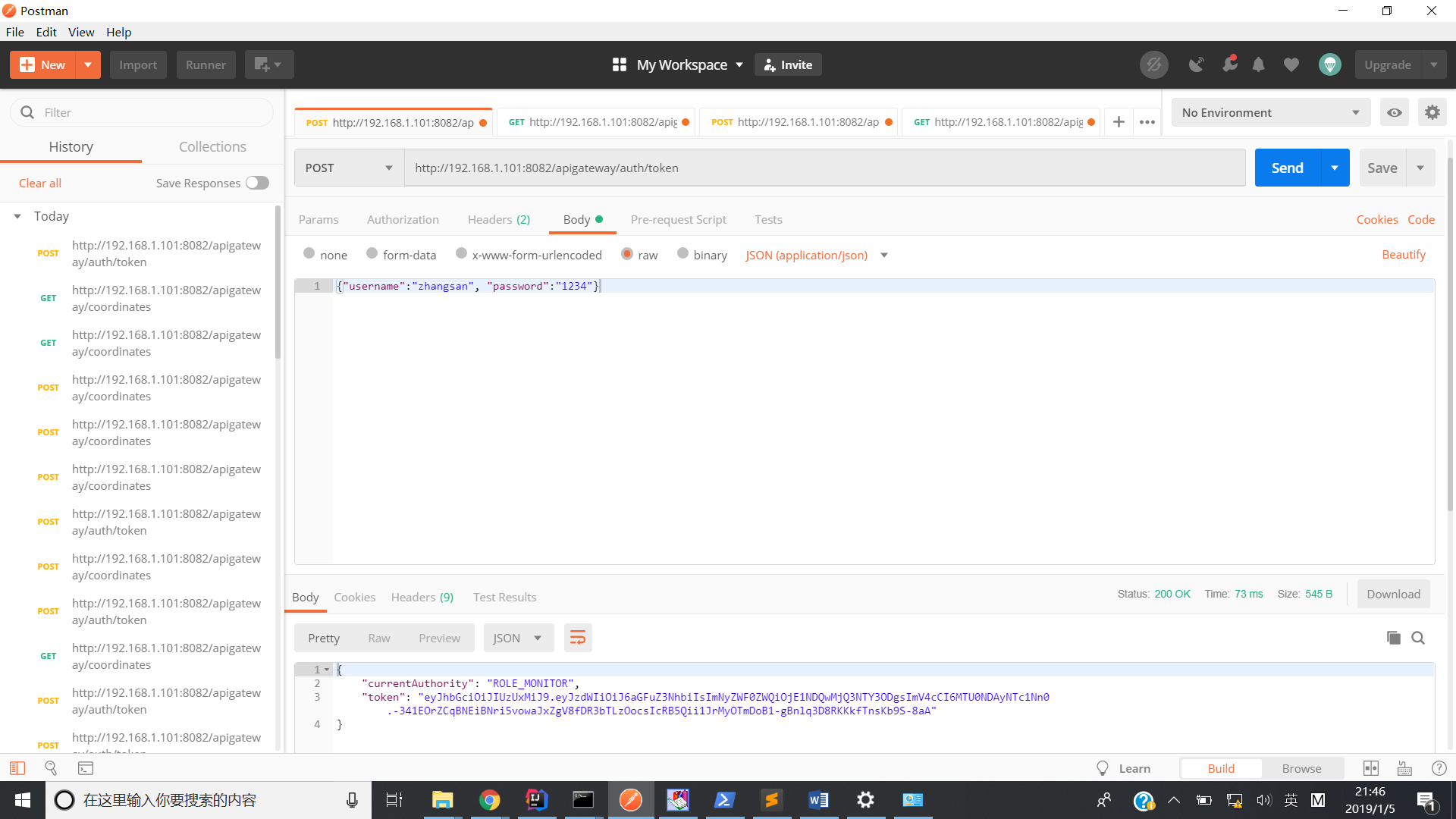
Postman将令牌填充到HTTP请求报文的Authorization字段中，然后观察调用结果。申请的令牌和调用结果如图6-5、图6-6所示。

图6-5 用户通过授权认证模块获取令牌

图6-5说明认证服务模块可以正确的生成令牌并返回给用户，图6-6正确的显示调用结果，说明API网关通过授权认证模块可以正确的识别应用权限并授权调用服务。

1. 创建用户李四，用户名为lisi，密码为4321，将ROLE\_LOGISTIC角色通过数据库赋给李四，然后模拟李四从认证服务模块申请令牌，并模拟控制台应用使用李四的令牌对coordinates服务进行调用，此时控制台应用不具有调用coordinates服务的权限，观察调用结果。申请令牌和调用结果如图6-7、图6-8所示。

图6-8说明API网关拒绝了应用的调用请求，这符合预期，因为模拟的控制台应用所具有的角色ROLE\_LOGISTIC没有对coordinate服务的调用权限。

1. 通过配置数据库给李四添加ROLE\_MONITOR权限，并重新生成令牌，然后使用Postman模拟控制台应用调用coordinates服务，此时控制台应用重新获得对coordinates服务调用的权限，生成的令牌和调用结果如图6-9，图6-10。

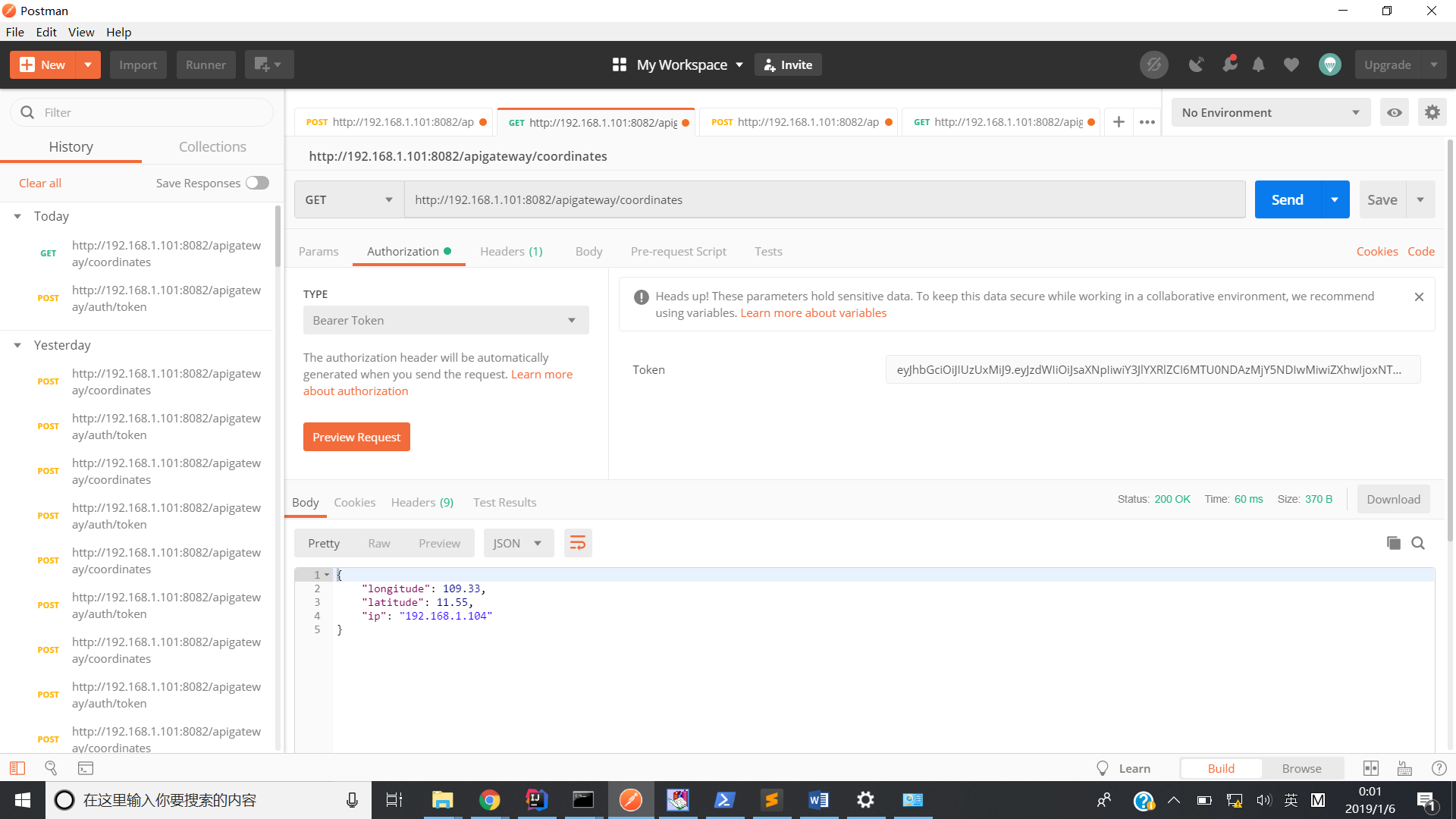
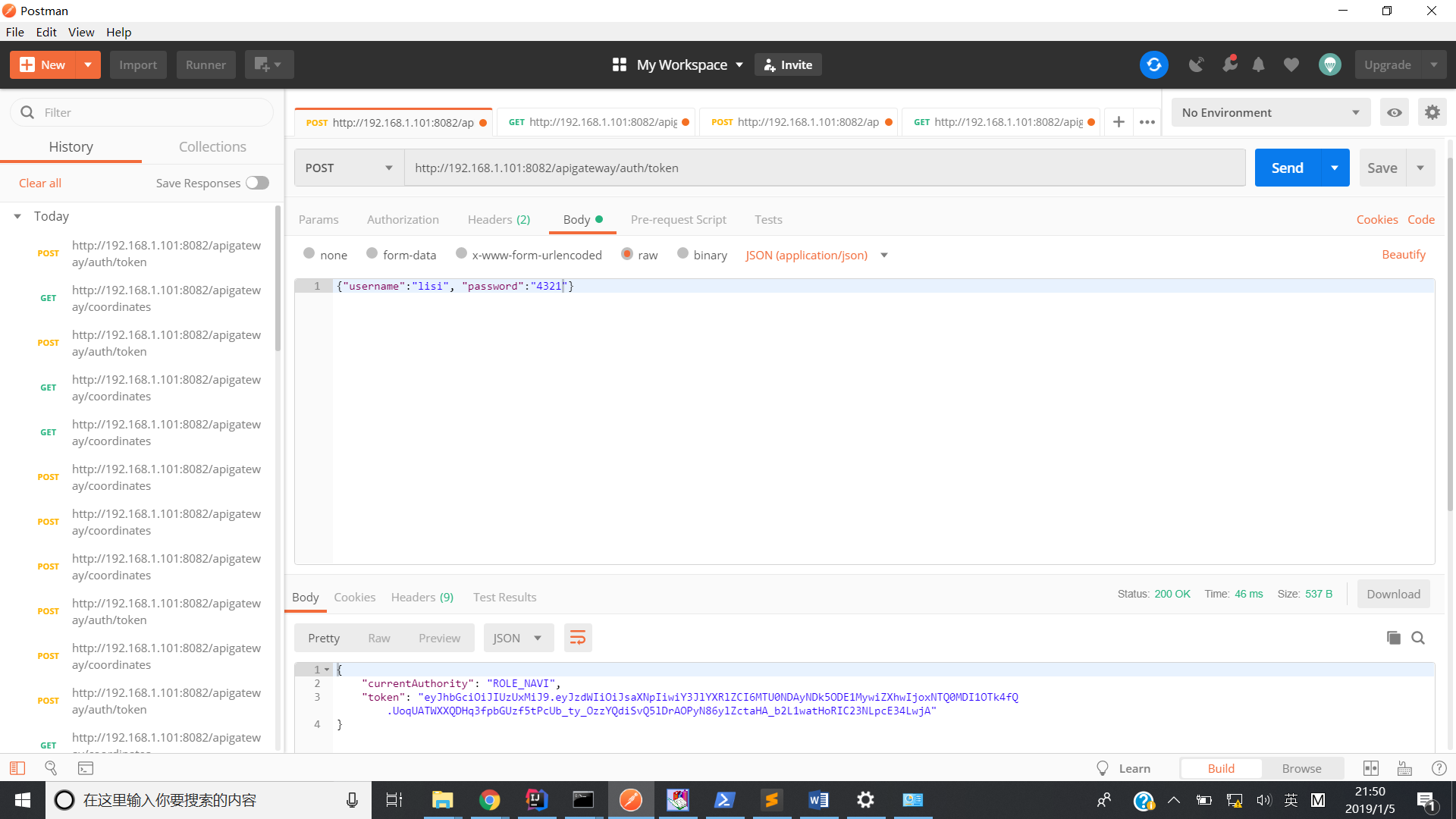
图6-6 模拟经过授权的控制台应用调用coordinates的结果

图6-7 具有ROLE\_NAVI角色的用户申请令牌的结果

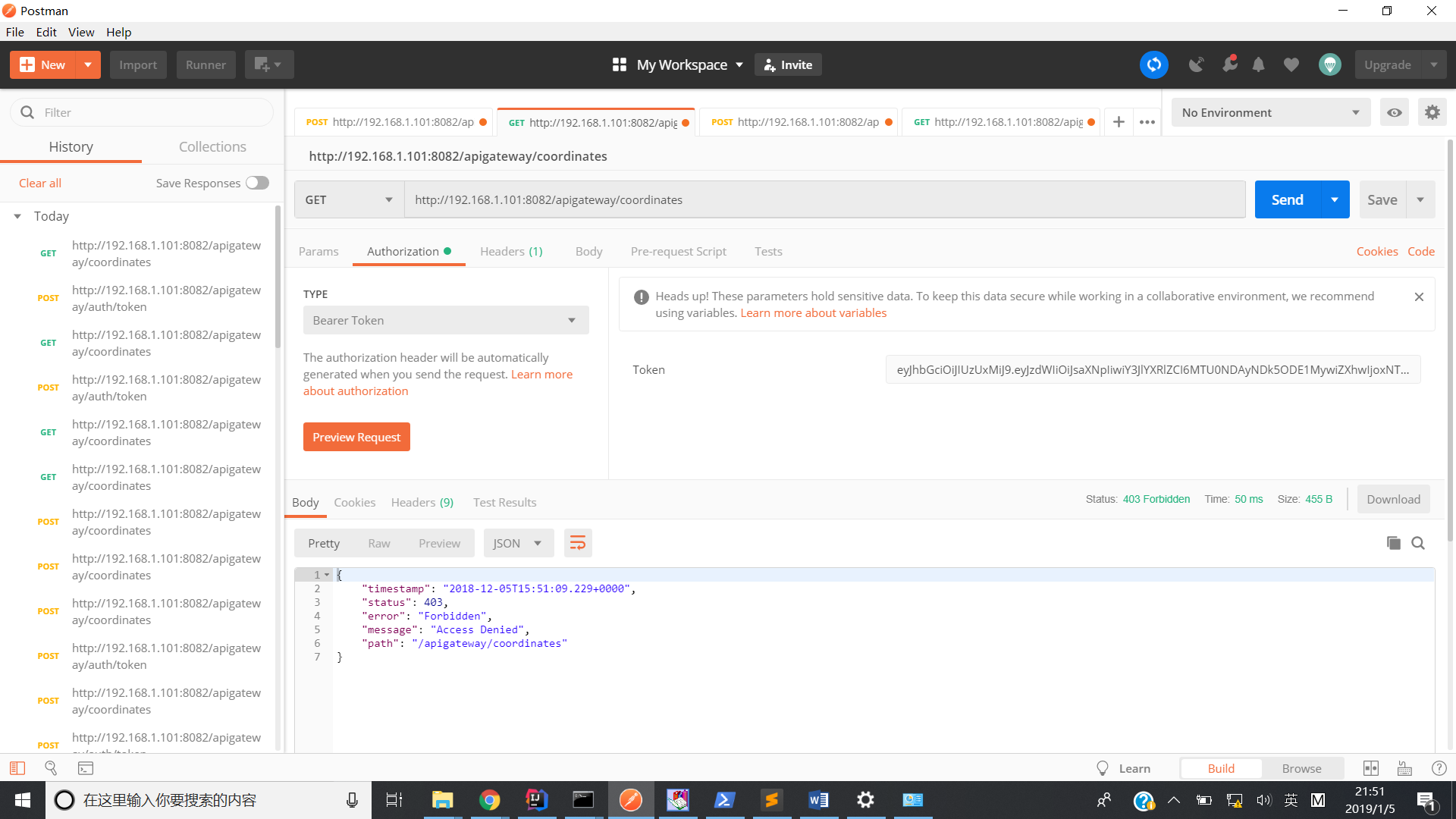
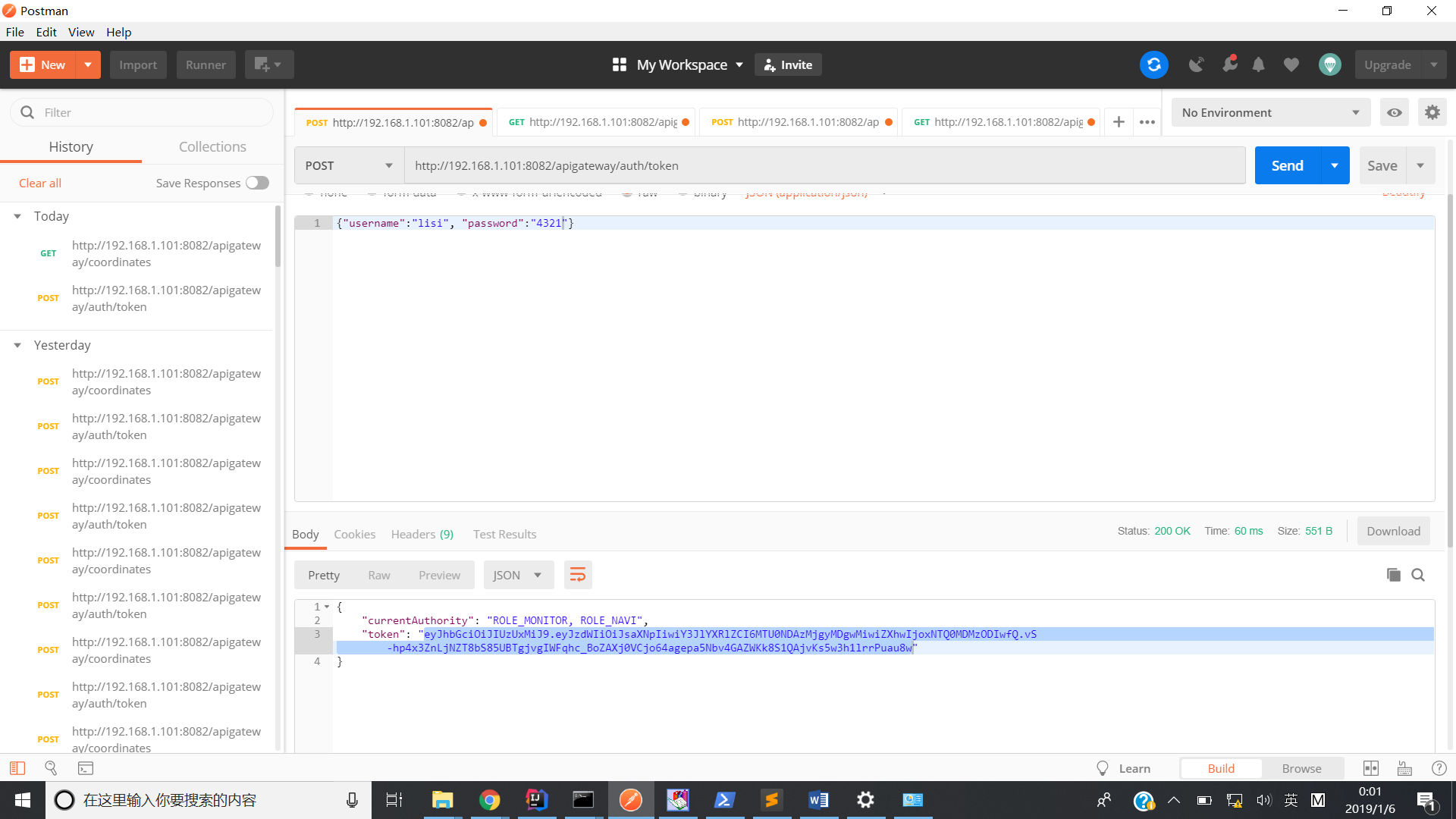
图6-8没有正确访问权限时模拟控制台应用调用coordinates服务的结果

图6-9 获得新授权的用户申请令牌的结果

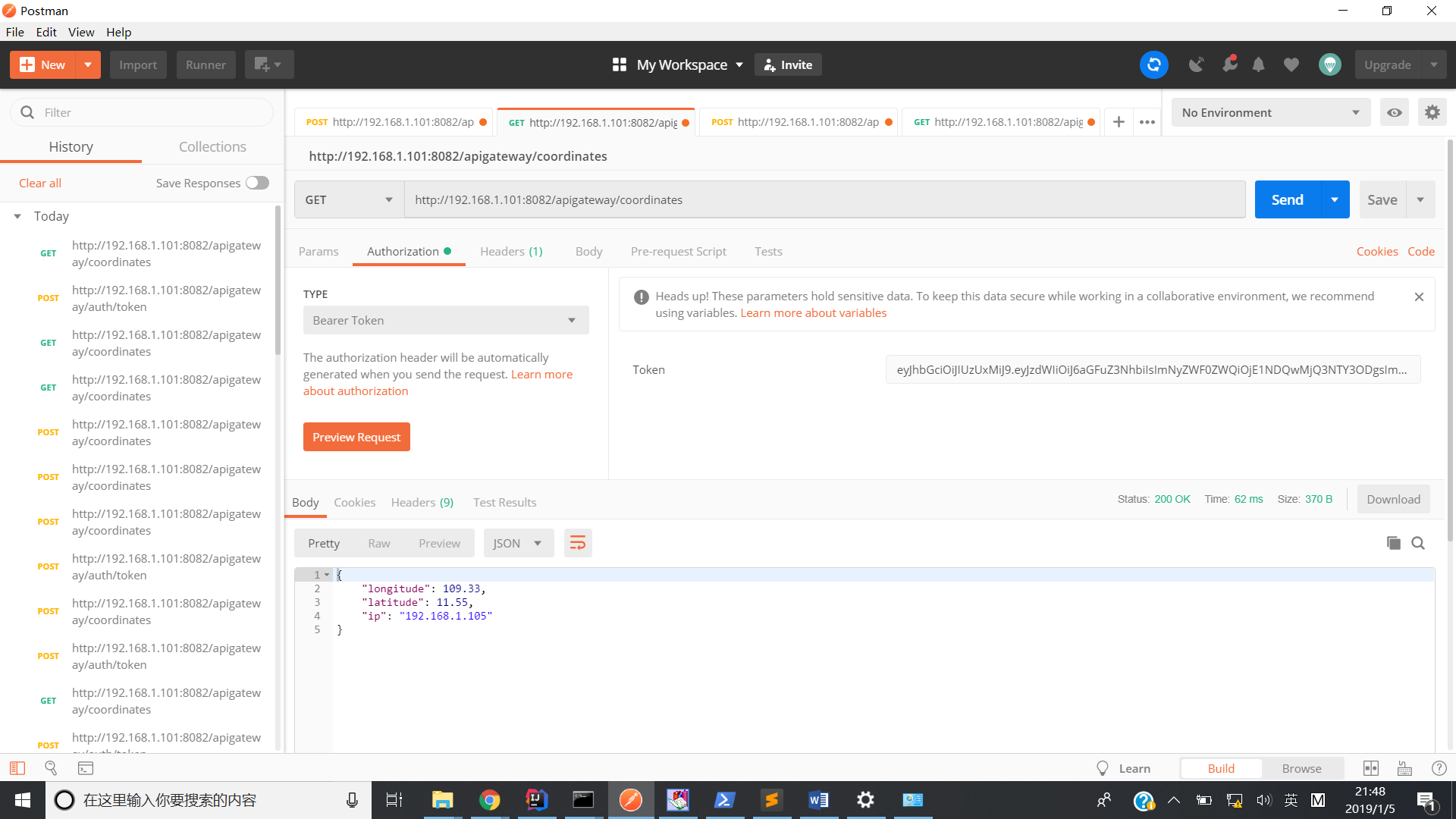
图6-10 模拟获得新权限的控制台应用调用coordinates服务的结果

图6-10显示此次服务调用正确的返回了调用结果，说明现在控制台应用获得了调用coordinates的权限，符合预期，证明API网关通过授权认证模块能正确的对应用权限进行识别，并对调用请求进行过滤。

通过上述实验，可以表明设计的面向舰船平台管理系统原型是可行的，其中API网关能正确的进行路由映射，负载均衡模块可以将调用请求平均分发到各个实例，同时验证授权模块可以对请求调用服务的应用的身份进行识别，并根据权限正确的接受或拒绝调用请求。

6.6本章小结

本章通过模拟应用场景下的服务调用对面向舰船平台管理系统原型的主要模块的功能进行了验证，包括API网关的路由映射功能，负载均衡功能，API网关的身份认证和授权管理功能等。本章首先介绍了对应用场景的仿真及实验环境，然后给出了验证原理和验证方法，最后通过每一步的实验结果表明本文设计的系统的可行的。

7 总结与展望

7.1 论文总结

为了满足我国海军平台管理信息化集成与管理需求，论文提出采用微服务架构实现舰船平台管理，并进一步给出实现该架构的面向舰船平台管理系统。通过采用面向服务架构，实现了系统模块的复用，降低了系统之间的耦合度，提高了系统的灵活性。采用微服务架构提高了舰载计算基础设施的利用率，增加了关键组件的可靠性和弹性，进一步降低服务开发、部署和维护的难度。

本人通过阅读大量有关微服务架构、微服务治理、以及SOA相关技术和框架的文献资料，总结了微服务治理中所需解决的关键性问题和微服务架构的设计原则，从而提出面向舰船平台管理系统这一具体的微服务架构实施方案。论文总结了使用微服务架构实现SOA治理的关键技术，并从面向舰船平台管理系统所需的功能性和非功能性需求出发，给出了系统的模块划分和关键模块的设计与实现。系统采用SOA基本的架构原理，采用REST架构原则将系统的关键组件封装成具有RESTful API的微服务，系统采用API网关技术对所有服务的接口进行管理，采用负载均衡和容错技术保证系统的稳定性，采用身份认证和权限控制技术对服务的调用进行管理。在文章最后，通过对模拟服务调用过程对系统的关键组件API网关与认证鉴权进行了测试验证，通过实验结果表明，采用微服务架构实现的面向舰船平台管理系统来进行SOA服务治理是可行且有效的。

7.2 下一步工作展望

本文主要在于对微服务整体架构所需的相关技术应用到舰船平台管理上的研究与验证，所以尽管在设计中给出了完整的设计方案，但在具体实现上做出了很多取舍。本文详细描述面向舰船平台管理系统主要集中在后端，故没有给出前端的具体实现，由于系统中的组件提供RESTful接口，并通过HTTP+JSON格式进行通信，故开发者可以灵活的在此基础上使用前端技术开发基于Web的用户界面。在接下来的工作中，需要继续进行研究的内容有：

1. 通信协议的进一步优化：主要考虑采用类似Protocol Buffers这种二进制数据描述语言来代替JSON作为数据传输格式，来降低空间开销和提高通信效率。
2. 系统安全性体系的规划：论文从身份认证和访问控制来讨论微服务架构的安全问题，后续要研究将数据加密技术整合到该套方案中的实现方法。
3. 自动化部署到云端的方案研究：舰船上的计算资源在逐步走向虚拟化，论文已经提出使用Docker对服务进行部署，后续将对在OpenStack平台上实现对微服务的自动化部署方案进行研究。
4. 数据存储对性能影响的研究：论文中采用关系型数据库对系统的所有数据进行存储，后期将研究在关键服务中使用Redis键值数据库存储对系统整体性能的影响。
5. 微服务架构下数据一致性的研究：通常微服务要求服务是无状态的，这样可以满足微服务动态扩展的特点。但有些服务是有状态的，比如说服务注册中心，后续还将对如何保持服务的多个实例之间的状态一致性和数据同步进行研究。

参考文献

1. 吕云飞, 王旋, 张军. 基于DDS技术的舰船平台综合管理系统[J]. 舰船科学技术, 2011, 33(5):47-52.
2. 郭蒙. 海洋工程船综合平台管理系统研究[D]. 大连海事大学, 2013.
3. 吴旭东. 舰船监控系统支撑平台研究与设计[D]. 武汉理工大学, 2013.
4. 宁小敏, 阳斌, 王奕, et al. 基于Web的船舶信息集成管理系统的设计及实现[J]. 中国舰船研究, 2011, 06(3):99-102.
5. 熊瑛, 许建. 船舶综合平台管理系统概念设计[J]. 船海工程, 2009, 38(5):36-39.
6. 许军海. 指控系统构件集成框架设计与实现[D]. 西北工业大学, 2017.
7. 董晓明, 冯浩, 石朝明, et al. 全舰计算环境体系结构和系统集成框架[J]. 中国舰船研究, 2014, 9(1):8-13.
8. Villamizar M, Garces O, Castro H, et al. 2015 10th Computing Colombian Conference (10CCC) - Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud[J]. 2015:583-590.
9. Zimmermann O. Microservices tenets[J]. Computer Science - Research and Development, 2017, 32:1-10.
10. 刘伟强. 基于Dubbo的分布式航空交通管理系统的研究与实现[D]. 2015.
11. Alshuqayran N, Ali N, Evans R. A Systematic Mapping Study in Microservice Architecture[C]// IEEE International Conference on Service-oriented Computing & Applications. IEEE, 2016.
12. Gouigoux J P, Tamzalit D. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture[C]// IEEE International Conference on Software Architecture Workshops. 2017.
13. Francesco P D, Lago P, Malavolta I. Migrating Towards Microservice Architectures: An Industrial Survey[C]// 2018 IEEE International Conference on Software Architecture (ICSA). IEEE Computer Society, 2018.
14. Kratzke N. About Microservices, Containers and their Underestimated Impact on Network Performance[C]// Cloud Computing 2015. 2015.
15. Johannes Thönes. Microservices[J]. IEEE Software, 2015, 32(1):116-116.
16. Shadija D, Rezai M, Hill R. Towards an understanding of microservices [C]// Automation and Computing (ICAC), 2017 23rd International Conference on. IEEE, 2017: 1-6.
17. Kryvinska N, Baroková A, Auer L, et al. Business value assessment of services re-use on SOA using appropriate methodologies, metrics and models[J]. International Journal of Services, Economics and Management 12, 2013, 5(4): 301-327.
18. Familiar B. From Monolithic to Microservice[M]. 2015.
19. Erl T. Service-oriented architecture (SOA): concepts, technology, and design[J]. 2005.
20. Thomas E. SOA Principles of Service Design[M]// Soa: principles of service design. Prentice Hall Press, 2007.
21. Strimbei C, Dospinescu O, Strainu R M, et al. Software architectures–Present and visions[J]. Informatica Economica Journal, 2015, 19(4): 13-27.
22. Salah T, Zemerly M J, Yeun C Y, et al. The evolution of distributed systems towards microservices architecture[C]//Internet Technology and Secured Transactions (ICITST), 2016 11th International Conference for. IEEE, 2016: 318-325.
23. Walls C. Spring Boot in Action[M]. 2016.
24. Fielding, Thomas R. Architectural styles and the design of network-based software architectures[C]// University of California, Irvine, 2000.
25. 冯新扬, 沈建京. REST和RPC:两种Web服务架构风格比较分析[J]. 小型微型计算机系统, 2010, 31(7):1393-1395.
26. Ismail B I, Goortani E M, Karim M B A, et al. Evaluation of Docker as Edge computing platform[C]// Open Systems. IEEE, 2016.
27. Preeth E N, Mulerickal F J P, Paul B, et al. Evaluation of Docker containers based on hardware utilization[C]// International Conference on Control Communication & Computing India. IEEE, 2016.
28. Anderson, Charles. Docker [Software engineering][J]. IEEE Software, 2015, 32(3):102-c3.
29. Ismail B I, Jagadisan D, Khalid M F. Determining Overhead, Variance & Isolation Metrics in Virtualization for IaaS Cloud[M]// Data Driven e-Science. Springer New York, 2011.
30. Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode[J]. 2015.
31. Izrailevsky Y, Bell C. Cloud Reliability[J]. IEEE Cloud Computing, 2018, 5(3): 39-44.
32. Chen E Y, Pei Y, Chen S, et al. Oauth demystified for mobile application developers[C]//Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM, 2014: 892-903.
33. Leiba B. Oauth web authorization protocol[J]. IEEE Internet Computing, 2012, 16(1): 74-77.
34. Hardt D. The OAuth 2.0 authorization framework[R]. 2012.
35. Jones M, Bradley J, Sakimura N. Rfc 7519: Json web token (jwt)[J]. IETF, May, 2015.
36. Richer J, Sanso A. OAuth 2 in Action[M]. Manning Publications, 2017.
37. Esposito C, Castiglione A, Choo K K R. Challenges in delivering software in the cloud as microservices[J]. IEEE Cloud Computing, 2016, 3(5): 10-14.
38. Ghofrani J, Lübke D. Challenges of Microservices Architecture: A Survey on the State of the Practice[J]. ZEUS 2018, 2018: 1.
39. Newman S. Building Microservices[M]. 2015.
40. V. Cardellini, M. Colajanni, P.S. Yu. Dynamic load balancing on web-server systems[J]. Internet Computing, IEEE, 1999, 3(3):28-39
41. Arpaci-Dusseau R H, Arpaci-Dusseau A C. Operating systems: Three easy pieces[M]. Wisconsin: Arpaci-Dusseau Books, 2014.
42. Karger D R, Ruhl M. Simple efficient load balancing algorithms for peer-to-peer systems[C]//Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2004: 36-43.

致 谢

时光荏苒，转眼间到了毕业前期，回顾两年半的研究生学习生涯，我倍感怀念和感恩。首先要感谢的是我的导师，您严谨认真的治学精神和耐心和蔼的谆谆教导让我在研究生学习阶段有了很大的提高，但最重要的是您不忘在教学之余向我传授谦逊做人、踏实做事的道理，让我在走上工作岗位这一竞争更加激烈的人生道路上能够礼貌待人、脚踏实地。我觉得我将最怀念的是您和我们一起探讨计算机理论时那种不仅要知其然，还要知其所以然的研究态度，和您对我们不正确观点的耐心纠正和指引。我将把这种认真对待基础知识，戒骄戒躁的精神应用到以后的工作当中。

同时我也要感谢教研室的邓磊老师和赵佐老师，感谢您们为我们创造的科研和学习氛围。从入学向我们推荐有用的计算机书籍到日常生活中对我们的嘘寒问暖，让我能够在科研的道路上做到心中有数、松弛有度。

我还要感谢教研室的5位同级的同学：黄亚文、马广辉、崔永硕、林盛力和石晶，和你们在一起对课堂知识的讨论、对项目的研究和对生活的憧憬将成为我在研究生阶段的美好回忆。你们和教研室其他师兄师姐和师弟师妹们一起，使我感受到一个大家庭的温暖，让我在研究生学习和生活期间不会感到孤独。

我还要感谢我的父母，正式他们的默默支持和鼓励，在我最低落的时候给予我开导和安慰，才让我在研究生阶段能够毫不气馁、勇往直前。我只有继续在今后的工作当中努力奋斗，才能回报他们的养育和教导之恩。

最后感谢在百忙之中抽出时间评阅论文的老师们，您们辛苦了！

攻读硕士学位期间发表的学术论文和参加科研情况

硕士在读期间参与的研究课题有：

1. ZZ软件构件研发与测试支持子系统.
2. 面向舰船平台管理服务化系统研究.

**西北工业大学**

**学位论文知识产权声明书**

本人完全了解学校有关保护知识产权的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属于西北工业大学。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。学校可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律注明作者单位为西北工业大学。

保密论文待解密后适用本声明。

学位论文作者签名： 指导教师签名：

年 月 日 年 月 日

———————————————————————————————————————

**西北工业大学**

**学位论文原创性声明**

秉承学校严谨的学风和优良的科学道德，本人郑重声明：所呈交的学位论文，是本人在导师的指导下进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容和致谢的地方外，本论文不包含任何其他个人或集体已经公开发表或撰写过的研究成果，不包含本人或其他已申请学位或其他用途使用过的成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式表明。

本人学位论文与资料若有不实，愿意承担一切相关的法律责任。

学位论文作者签名：

年 月 日