毕 业 论 文

题　目 微服务可视化运行监测系统的设计与实现

姓　名　　田昕峣

学　号　　15051204

指导教师　王焘　张文博　李娟

日　期　　2019 年 6 月

北京工业大学

毕业设计（论文）任务书

题目　微服务可视化运行监测系统的设计与实现

专业学号姓名　软件工程　15051204　田昕峣

主要内容、基本要求、主要参考资料等：

主要内容：本课题面向微服务软件系统中服务数量众多，服务交互复杂，运行时动态变化的特点，研究微服务的动态监测、异常报警、可视化展示等技术，开发微服务软件系统的可视化监测管理工具，实现系统管理人员能够通过可视化界面高效监测系统运行状态，达到对运行过程中出现的问题进行及时响应处理的目标。

基本要求：

1、调研当前主流的微服务管理系统；
2、研究微服务的动态监测、异常报警、可视化展示等技术；
3、开发微服务软件系统的可视化监测管理工具；
4、应用工具验证原型系统的有效性。

时间安排：

1、2018.9---2018.12 调研并查阅资料，设计微服务管理系统技术架构。
2、2019.1---2019.3　研究自动化的微服务注册与发现，持续配置及多版本演化，高效监测及异常状态检测方法。
3、2019.3---2019.6　实现微服务可视化管理系统，并完成毕业论文。

参考文献：

[1] Tao Wang, Jiwei Xu, Wenbo Zhang, Zeyu Gu, Hua Zhong. Self-adaptive cloud monitoring with online anomaly detection. Future Generation Computer Systems, 80(3): 89-101, March 2018.
[2] Tao Wang, Wenbo Zhang, Chunyang Ye, Jun Wei, Hua Zhong, and Tao Huang, FD4C: Automatic Fault Diagnosis Framework for Web Applications in Cloud Computing, IEEE Transactions on Systems, Man and Cybernetics: Systems, (TSMC-S),46(1), pp. 61-75, Jan., 2016.
[3] Tao Wang, Jun Wei, Wenbo Zhang, Hua Zhong, Tao Huang, Workload-Aware Anomaly Detection for Web Applications, Journal of Systems and Software, 89(3), 19-32, 2014.
[4] Tao Wang, Jun Wei, Feng Qin, Wenbo Zhang, Hua Zhong, Tao Huang, Detecting Performance Anomaly with Correlation Analysis for Internetware, SCIENCE CHINA Information Sciences, 56(8), 082104(15), 2013.

完成期限：2019 年 6 月 10 日

指导教师签章：

专业负责人签章：

年　月　日

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：　　日期：

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，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、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：　　导师签名：　　日期：

摘要

随着容器技术和微服务架构等理念和技术的不断发展和深入，微服务应用的服务质量监测和控制变得日渐重要。微服务架构以服务为核心，并以云计算平台作为支撑，是实现传统软件云计算化的重要手段，是软件职责细化和功能高度解耦的体现，是未来云计算环境下软件开发、软件部署、软件运维和软件使用的重要手段。针对当前微服务管理所存在的服务治理复杂、服务间流量难以控制、服务版本难以管理等问题，本文使用轻量级容器技术和微服务架构，设计并实现了一种新型微服务监控和管理可视化系统，并通过典型微服务应用的实例研究验证了系统的有效性。该系统将微服务中的业务逻辑和网络通信分离，利用微服务基本单元，对微服务集群的网络状况进行监测和控制，并提供可视化用户界面，具有轻量化服务治理、服务间流量控制、服务版本管理等技术特色，提供良好的用户体验。

关键词： 云计算； 微服务； 容器技术； 可视化系统； 运行监测

Abstract

With the continuous development and deepening of concepts and technologies such as container technology and microservice architecture, the quality of service monitoring and control of microservice applications has become increasingly important. Microservice architecture is based on service and supported by cloud computing platform. It is an important means to realize traditional software cloud computing. It is a manifestation of software responsibility refinement and high functional decoupling. It is software development and software in the future cloud computing environment. An important means of deployment, software operation and maintenance and software using. Aiming at the problems of current microservice management, such as complex service management, difficult control of service traffic, and difficulty in controlling service versions, this paper designs and implements a new microservice monitoring and management visualization based on lightweight container technology and microservice architecture. The effectiveness of the system are verified by cases of typical microservice applications. The system separates the business logic and network communication in the microservice, and uses the microservice basic unit to monitor and control the network status of the microservice cluster, and provides a visual user interface, with technical features like lightweight service management, service flow control, and service version control, to provide a good user experience.

Keywords: cloud computing; microservice; container technology; visualization system; operation monitoring

北京工业大学毕业设计（论文）
目录

1.绪论

1.1 研究背景

随着云计算技术的蓬勃发展，基于云计算的开发模式也逐渐变成软件开发的主流。云计算技术不仅促进了计算机软硬件及体系结构的发展，也引发了软件使用方式上的变革。同时， IT 资源服务化的思想日益普及，呈现出一切皆服务(X as a service，XaaS)的趋势，服务成为了云计算的本质和核心概念，以 IaaS、PaaS 和 SaaS 为代表的服务模型已经得到了广泛的使用和实践[1]。随着微服务和容器技术发展的不断深入，以 Docker 技术为代表的容器化微服务技术逐渐渗透到云计算的各个层面，系统从开发、部署到运维整个过程都可以微服务化。微服务架构 (microservices architecture)成为一种架构风格和设计模式。该模式提倡将应用分割成一系列细小的服务，每个服务专注于单一业务功能，运行于独立的进程中，进而使得服务之间边界清晰，并可以采用轻量级通信机制(如 HTTP/REST API)相互沟通、配合来实现完整的应用，满足业务和用户的需求。微服务作为架构模式的变革，其诞生绝非偶然，它是当传统服务架构在互联网时代遭遇挑战时，人们对于架构模式、开发和运维方法论的一种反思[2]。

微服务在过去 5 年中的发展趋势尤为迅猛。不论是在互联网领域还是传统制造业领域，微服务都成为了技术热点，以微服务架构和以 Docker 技术为代表的容器技术逐渐成为传统企业进行互联网技术转型的核心。

1.2 课题研究的意义

微服务架构是当今软件开发的趋势。由于云计算的发展和流行，基于云计算的微服务架构及其相应的开发模式也得到了蓬勃发展。其聚合性高、耦合度低的特性和跨语言等诸多好处为软件开发者和运维人员带来了便利。然而，目前并没有一款能够综合微服务治理、监测和控制的微服务监控一体化平台，供软件开发人员和运维人员通过可视化和非可视化的方式进行使用。[3]

从微服务架构的特性考虑，对于不同的被监测服务的获取方式、结构、种类和针对不同数据的特性，并且结合现有的知识与资源，对监测设计的复杂程度进行简化，从中抽象出更加便于理解，具备可扩展性的微服务可视化监测模型。考虑到微服务架构是基于云平台，而云平台具有诸如准确性、低消耗、耦合性、实时性、可扩展性等诸多特性，故微服务监控系统可以同时利用云计算的复杂性和动态性这两大特点，从而更好地负载平衡以及资源调度，并为软件开发者和运维人员提供直接的便利。

对于云计算这一庞大而复杂的系统，如果要使其各项服务的性能达到最优或趋近于最优，我们就必须掌控其中的各项资源的使用和运行情况。在这种情况下，就需要对微服务架构中的各个微服务模块进行监测，并使用易于运维人

员理解的方式进行可视化。微服务各项指标和资源的监测不仅可以及时发现节点上的运行故障，也可以精准定位到故障位置，从而可以很快速的恢复系统性能，或根据服务需要进行有针对性的扩容或缩容；同时，及时掌握整个微服务应用中各项资源的使用状况，为负载均衡以及任务调度提供了可靠的数据保障。

基于以上两点主要需求，本文面向微服务架构设计并实现一个高效微服务资源监测及可视化的分布式系统软件。

1.3 本文组织结构

本文共分为七个章节，具体章节内容如下：

第一章为引言。本章主要就本文工作的研究背景进行了说明。容器化微服务为监控带来了很多问题与挑战，本文面向容器化微服务监测与可视化工具的设计与实现问题展开。

第二章为相关技术。本章对微服务监测追踪相关的技术以及概念进行详细解释以及对比。在本章，分别从适用场景、基本思想、实现原理等方向对当前工作进行总结，并比较当前方法的优缺点，指出当前工作的不足。

第三章为系统需求分析。在设计、实现软件系统之前，清楚地分析并掌握系统的各项需求尤为重要。本章将着重分析目前微服务架构所遇到的诸多问题，并针对这些问题提出相应的解决方案，并将其作为该系统所应满足的需求进行后续的设计与实现。

第四章为微服务监测与可视化架构的设计。本章提出了整体架构的原理、思想和设计，包括微服务间的通讯、基于规则的度量值监测、跨语言分布式追踪、阈值报警和数据可视化模块。本章也对本文的具体思路进行了阐述。

第五章为微服务监测架构的实现过程，本章首先进行了为服务监测与可视化系统的需求分析，接着基于第三章内容进行了基于架构的具体实现细节和集成过程。本微服务可视化工具支持度量值的收集和相应微服务间依赖关系的展现。

第六章为实验验证结果与分析，包含验证实验的设计过程和实验结果的展示与分析，主要是对容器化微服务资源的相关指标进行展示和异常情况的监测报警，还有微服务治理的部分实验验证。

第七章为结束语。本章主要对前述内容进行总结。描述了本文的主要工具以及在本文创作过程中所遇到的问题，同时针对本文工作的不足，提出了一些未来工作的展望。

1.4 本章小结

本章主要介绍了微服务的技术背景以及目前微服务架构的发展现状，从而引出本课题的研究意义。在本章的最后，笔者简要说明了本文的行文结构和每一章的主要内容，以求给予读者一个清晰、明确的文章脉络。

2.相关技术

本章首先将对微服务和容器的相关概念进行简要阐述，接着介绍本文所设计的微服务可视化监测系统所需要的部分核心技术，旨在提出一个容器化微服务的监控可视化架构并对其进行设计和优化，最终构建出一个能够基本用于可视化监控微服务应用中各个微服务单元的资源及使用情况的分布式系统。

2.1 微服务与容器技术

2.1.1 微服务

微服务是一种架构风格，一个大型复杂的软件应用由一个或多个微服务组成。应用中的各个微服务组件可被独立部署、运行、升级，且各个微服务之间是松耦合的。每个微服务仅关注于完成单一的任务并保证自身的服务质量。在所有情况下，每个任务代表着一个小的业务能力，而在功能上则表现为一个统一的整体。这种所谓的“统一的整体”表现出来的是统一风格的用户界面，统一的权限管理，统一的安全策略，统一的部署过程，统一的日志管理和审计方法，统一的调度方式，统一的访问入口等等。微服务的目的是有效的拆分应用，实现敏捷开发和部署。微服务的优点有易于开发和维护、启动较快、局部修改易于部署、跨语言开发、按需伸缩等，缺点有运维要求较高、分布式的复杂性、接口调整成本高等[4]。

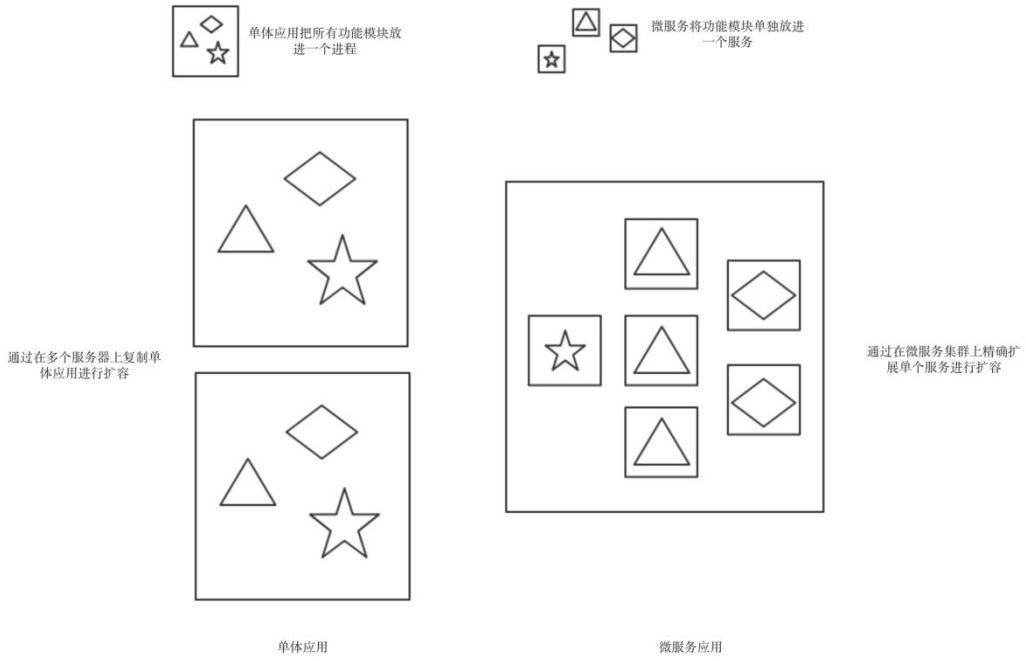


图 2-1 单体应用与微服务

2.1.2 容器技术

容器技术已经成为一种被开发者广泛认可的虚拟化技术和服务器资源的共享方式，容器技术可以在按需构建容器技术操作系统实例的过程当中为系统管理员提供极大的灵活性。基于 LXC（Linux Container）或者 Docker[5] 等技术的容器技术，以其快速、轻量、面向服务的特点被越来越多的企业所使用。其基本原理是利用操作系统内核的隔离技术，以进程的方式对服务器资源进行虚拟化使用。现有的主流隔离方案主要采用 Linux Namespace 以及 Cgroups 等技术实现。容器的主要特点在于其运行于宿主机操作系统之上，能够共享主机内核，容器内性能接近原生操作系统性能，且能够快速启停、资源利用率高。此外，容器技术的另一个特点在于能够实现开箱即用，在面对需要灵活快速缩容扩容的场景下非常方便。同时容器生态下丰富强大的各类工具和各类镜像资源也提高了容器的可用性。但是其缺点在于容器之间的隔离性仍然存在不足之处，某些情况下位于同一宿主机上的容器间会存在一定的影响。容器技术是实现微服务很好的选择，将微服务应用以及各种工具库和依赖包整合到一个可移植的容器中，发布到任何流行的 Linux Distribution 机器上，也可以实现虚拟化。容器完全使用沙箱机制，不依赖于任何语言、框架包括系统，因此，基于容器技术的微服务封装可与经过优化的强大基础架构相结合。

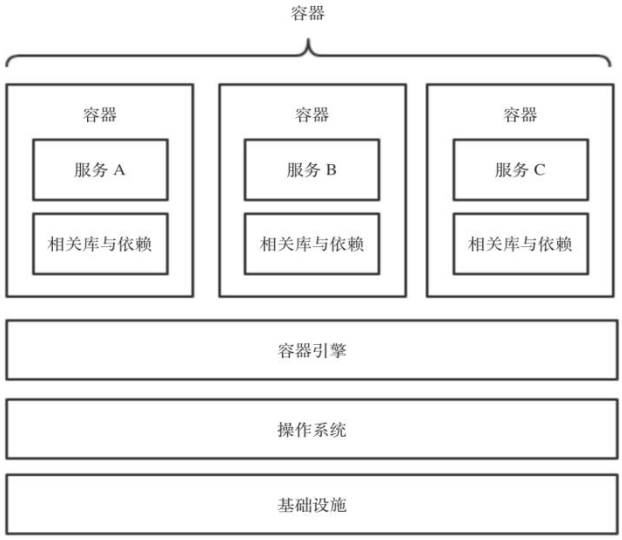


图 2-2 容器技术

2.2 微服务应用的支撑技术

2.2.1 容器集群管理框架

目前，微服务架构的开发与实现通常都是基于容器技术，这就带来了一个严峻的挑战，即有效地管理这些繁杂琐碎的装载了微服务应用各组件的容器。

我们需要一个专门用于全方位管控容器生命周期、对容器启动或停止进行管理和调度等功能的管理平台，来辅助各种各样的容器。

使用容器集群管理平台，可是在物理主机或虚拟主机上以集群的形式运行容器化的应用。容器集群管理平台的关键，是提供一个 “以容器为中心” 的基础架构，满足一些在生产环境中运行微服务应用的常见需求，如多进程以容器的形式进行协同工作、微服务实例的扩容与缩容、负载均衡和滚动更新等。

在所有容器集群管理平台中，目前最为流行的是由 Google 和 IBM 公司开发的容器集群管理框架—— Kubernetes[6]。准确来讲，Kubernetes 是一个容器集群管理系统，一个开源的平台。它可以实现容器集群的自动化部署、自动扩缩容、维护等功能。

Kubernetes 还具有许多可以用于容器管理的特性，诸如快速部署应用、快速扩展应用、无缝对接新的应用功能、节省资源和优化硬件使用资源等功能。

2.2.2 网络代理

在服务网格模式中，每个微服务的基本单元都配备了一个用于微服务间通讯的网络代理，我们称其为“Sidecar”，用于服务之间的通信。这些通讯代理通常与微服务的业务逻辑代码共同部署，并且它不会被业务逻辑代码所感知。这些代理组织起来形成了一个轻量级网络代理矩阵以便统一管理和配置。这个基础设施层，也就是上文提到的服务网格。这些代理不再是孤立的组件，它们本身是一个具备完善功能的服务通讯网络。它可以以 Sidecar 的形式部署在每个主机上，所有的 Sidecar 形成一个透明的通信网格，每个应用程序发送和接收来自本地主机的消息，并且不知道网络的拓扑结构。在实际的开发过程中，我们通常将 Sidecar 注入微服务的每个基本单元中，所有传入传出的网络流量（ TCP、UDP 及其之上的协议 ）将通过 Sidecar 被转发至相应目的地。因此，每个 Sidecar 都可以监控所有的服务间 API 调用，并记录每次服务调用所需的时间以及是否成功完成。这种做法很好地屏蔽了不同编程语言的差异性，并且几乎不用对业务代码进行改动和浸入。

Envoy[7] 是一个高性能的开源 L4 及 L7 代理和通信总线，是一个独立的流程，旨在与每个微服务单元的服务并行运行，在实际应用中我们经常使用其作为 Sidecar 代理。Envoy 使用单进程多线程模式：一个主线程，多个工作线程。主线程协调和管理这多个线程来工作。每个线程都独立监听服务，并对请求进行过滤和数据的转发等。在实际工作中，Envoy 会启动一个或者多个 Listener，监听来自上游或下游服务的请求。当 Listener 接收到新的请求时，会根据关联的 Fliter 模板初始化配置这些 Fliter，并根据这些 Fliter 链对这些请求做出处理，有三种类型的 Network (L3/L4) Fliter：读、写、读/写 Fliter。 Envoy 包含了一个 HTTP Router Filter，该 Filter 可以用来实

现更高级的路由功能。它可以用来处理边缘流量/请求(类似传统的反向代理)，同时也可以构建一个服务与服务之间的 Envoy 网格。

2.2.3 服务网格

本课题旨在针对不同的微服务监控工具的客户端进行集成，对通过微服务中每个模块的网络流量进行统一监控，以隔离微服务中不同语言服务的差异性，从而实现业务逻辑与服务通讯的解耦。目前微服务架构中较为新颖的服务网格[8]的概念很适合本文的应用场景。

服务网格是一个专注于处理服务间通信的基础设施层。通常情况下，微服务应用有着复杂的服务拓扑结构，而服务网格保证请求可以在这些拓扑中可靠地穿梭。在实际应用当中，服务网格通常是由一系列轻量级的网络代理组成的，它们与应用程序部署在一起，但应用程序不需要知道它们的存在。

对于不同类型或功能的微服务应用而言，可视化监测的需求侧重点也会有所不同。本课题期望通过使用一个统一的中间件或基础设施层将不同监测追踪模块进行统一管理，并可以通过该中间件与后端监测追踪服务进行交互，从而提高其可扩展性。

Istio [9]是目前使用最为广泛的服务网格，由 Google、IBM 和 Lyft 共同开发和维护。作为一个服务间通信的基础设施层，其解耦了应用逻辑和服务访问中版本管理、安全防护、故障转移、监控遥测等切面的问题。在 Istio 1.0.6 版本中，其控制面板包括 Pilot、Mixer 和 Istio-auth 三大组件。本课题主要使用其用于配置路由规则的 Pilot 组件和用于收集遥测数据的 Mixer 组件，。Pilot 管理和配置部署在特定 Istio 服务网格中的所有 Sidecar 代理实例，指定其使用什么规则在 Sidecar 代理之间路由流量，还维护了服务网格中所有服务的规范模型，并使用它来让 Envoy 通过内置的服务发现机制了解服务网格中的其他微服务应用实例。基于此统一流量的入口和出口，而对于和监测后端解耦的需求，则可以基于 Mixer 组件来进行。 Mixer 提供应用程序代码和基础设施后端之间的通用中间层，我们设计将策略决策从应用程序层转移到配置中。应用程序代码不再与特定后端集成，而是与 Mixer 进行相当简单的集成，由 Mixer 负责统一的后端系统进行监测数据的收集与管理。由于平台独立的特性， Istio 可以在各种环境中运行，包括跨云、预置环境、Kubernetes、Mesos 等，目前与 Kubernetes 共同使用的情况最为广泛。这些功能极大的减少了应用程序代码，底层平台和策略之间的耦合。耦合的减少不仅使服务更容易实现，而且还使运维人员更容易地在环境之间移动应用程序部署，或换用新的策略方案。因此，使用 Istio 的结果就是微服务应用的业务逻辑从本质上变得更加容易管理。

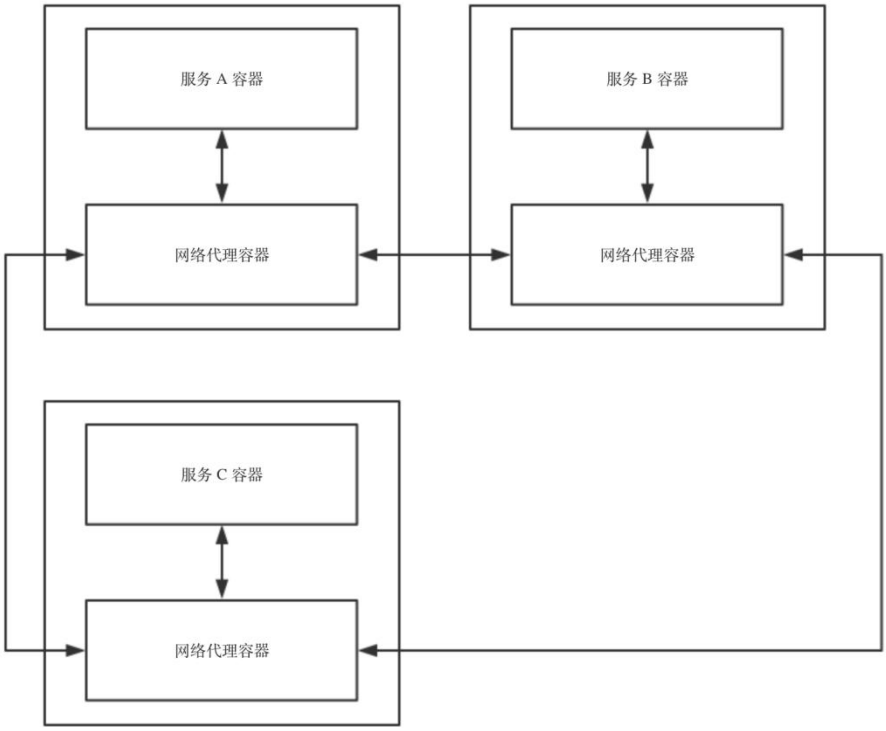


图 2-3 服务网格示意图

2.3 微服务执行轨迹追踪

对于服务中产生的追踪数据，需要进行收集，需要关心采样方式的问题以及针对不同的微服务追踪的后端系统，可能会支持不同的数据格式，需要将其转化为适配的数据格式。在采样方式方面，Zipkin[10] 和 Jaeger[11] 都基于了 Dapper 论文中采样模式，在客户端和服务端都可设置采样，并且可根据场景调节采样方式，Jaeger 提供了更加丰富的采样方式，在数据格式方面，目前 Zipkin 和 Jaeger 这两种分布式追踪工具的后端收集格式已经做到了互相兼容，可考虑支持更多的分布式追踪后端工具。

本追踪系统期望支持多种数据库存储。Zipkin 支持 Cassandra、mysql 等数据库，Jaeger 支持两种流行的开源 NoSQL 数据库作为跟踪存储后端：Cassandra 3.4+和 Elasticsearch[12] 5.x / 6.x。目前正在进行使用其他数据库的社区开发，如 ScyllaDB，InfluxDB 和 Amazon DynamoDB。

在追踪过程和服务依赖关系展示方面需要基于需求做到可定制，Zipkin 和 Jaeger 都提供简单的 JSON API 获取数据，给 Web UI 使用，Web 界面显示每个应用程序有多少跟踪请求，还提供了一个依赖关系图。可发现延迟或错误问题，可以根据应用程序，跟踪长度，注释或时间戳过滤或排序所有跟踪,且用户界面可自定义。

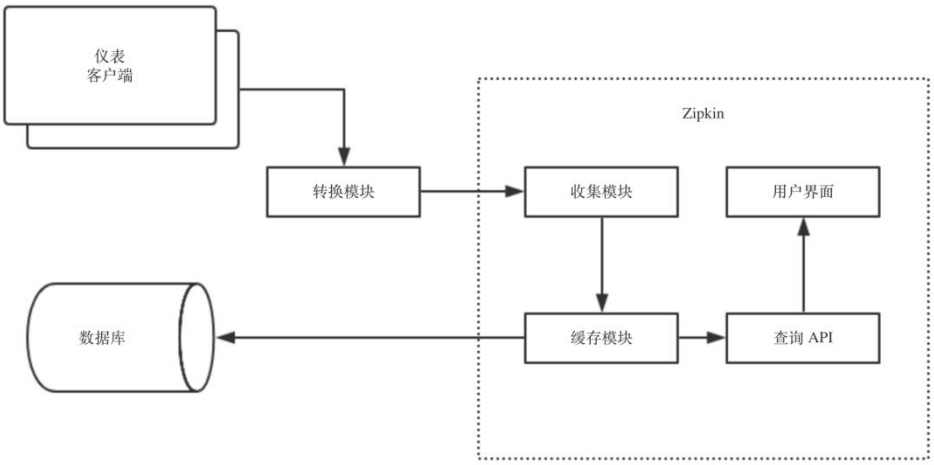


图 2-4 Zipkin 架构图

2.4 监控与可视化

2.4.1 监测数据的持久化——Prometheus

Prometheus[13] 是一个开源监控系统，它前身是 SoundCloud 的警告工具包。从 2012 年开始，许多公司和组织开始使用 Prometheus 用于缓存或持久化其系统的各项数据，并进行监控。该项目的用户社区非常活跃，且正在有越来越多的开发人员和用户参与到该项目中。目前它是一个独立的开源项目，且不依赖与任何公司。为了强调这点和明确该项目治理结构，Prometheus 在 2016 年继 Kurberntes 之后，加入了 Cloud Native Computing Foundation 。

Prometheus 有诸多的特征，包括：多维度数据模型；提供 PromQL ，是一种灵活的查询语言；不依赖分布式存储，单个服务器节点的自主性。在数据收集方面，它以 HTTP 方式，通过 pull 模型拉取时间序列数据，也能够通过中间网关支持 push 模型；它有一系列的配置方案，通过服务发现或者静态配置，来发现目标服务对象；它在本地存储通过 pull 模型抓取到的所有数据，并通过一定规则进行清理和整合数据，并把得到的结果存储到新的时间序列中，PromQL 和其他 API 可以简单地可视化展示收集的数据，支持图表和界面展示，同时它能接入其他专门用于可视化的模块进行更加丰富的展示。

Prometheus 在记录纯数字时间序列方面表现非常好。它既适用于服务器等硬件指标的监控，也适用于高动态的面向服务架构的监控。对于现在流行的微服务架构，Prometheus 的多维度数据收集和数据筛选查询语言同样非常强大。它是为服务的可靠性而设计的，当服务出现故障时，它可以使开发者或运维人员快速定位和诊断问题。它的搭建过程对硬件和服务没有很强的依赖关系。

Prometheus 的另一价值在于其可靠性，甚至在很恶劣的环境下，你都可以随时访问它和查看系统服务各种指标的统计信息。但是若对统计数据需要 100%的精确度，那么它并不适用。

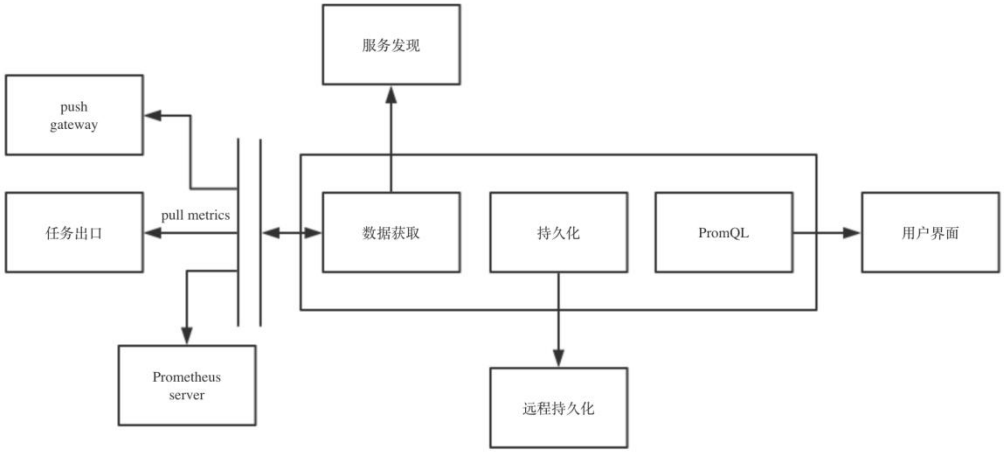


图 2-5 Prometheus 架构图

2.4.2 监测数据的可视化——Grafana

Grafana[14] 是一款采用 go 语言编写的开源应用，主要用于大规模指标数据的可视化展现，基于商业友好的 Apache License 2.0 开源协议。由于本微服务监控可视化系统采用服务网格作为基础设施以及 Prometheus 作为辅助数据监控模块，会产生大量监测数据，因此使用用于大规模指标数据的可视化工具 Grafana 是极为妥当的。

Grafana 拥有其内置的类 SQL 查询语言。通过使用该语言，用户可以快速创建出用于监测各项系统资源指标的折线图、柱状图、饼图等多种类型的可视化图表。对于不同类型的需要监测的资源，我们也可以绘制不同的图表对其进行可视化。

不仅如此，Grafana 还具有设定监测阈值、资源溢出报警的功能。通过绘制出的各类资源监测图表，Grafana 提供了设定报警值的功能。在被监测资源（如 CPU 使用率、内存等）超过由运维人员设定的报警值时，Grafana 便会触发报警器，在可视化界面上进行资源溢出报警。更进一步，使用其报警的触发器，结合我们自主研发的监测模块代码，我们甚至可以实现报警触发发送邮件、或发送短信通知运维人员等功能。

Grafana 还是高度解耦的可视化工具，其平台和数据源无关性得到了充分的体现。通过设置其数据源(Data Source)，Grafana 可以兼容目前市面上几乎全部的主流数据库或数据持久化工具。由于 Prometheus 仅具备基本的可视化功能，因此，使用 Grafana 搭配 Prometheus 数据监控系统作为本微服务可视化监控系统的可视化组件将会起到锦上添花的效果。

2.5 本章小结

本章着重介绍了支撑微服务的相关技术，包括微服务、容器技术、容器集群管理框架、网络代理和服务网格。除此之外，为了实现本文设计的可视化监控系统的相关功能，本章还简要介绍了需要集成进系统的相关开源项目，为后续章节进行技术铺垫。

3.系统需求分析

3.1 微服务管理面临的问题

本节将着重讨论使用微服务架构开发应用时所面临的主要困难与挑战。

伴随着众多企业微服务转型的浪潮，典型微服务开发模式开始在各大互联网公司普及，其中最为流行的开发模式，便是基于 Spring Cloud 的微服务开发和管理框架。但这种新兴的开发和管理框架仍然存在着许多问题：利用 Spring Cloud 框架，我们需要重复性地处理一系列基础工作，比如：服务注册、服务发现、得到服务实例后的负载均衡、熔断机制等。这些工作在 Service Mesh 出现之前统统都要开发人员在项目中用代码解决并实现，导致应用程序中加入了大量的非功能性代码。即使使用类似 Netflix OSS 的库和 Spring Cloud 的框架，开发人员依然面临着需掌握内容多、技术门槛高等诸多困难。总的来说，利用这种基于 Spring Cloud 的微服务开发和管理框架，会使运维和开发人员面临五点主要困难，下文将逐一说明。

3.1.1 微服务开发的学习成本过高

用于进行微服务开发的 Spring Cloud 框架是目前较为流行的框架，我们可以来看一下其基础库中的内容。根据 Spring Cloud 官方网站的资料，仅 Spring Cloud 的基础库就诸如 spring-cloud-Netflix、spring-cloud-consul 等 11 个子库 。而仅 spring-cloud-Netflix 子库中，又包含了诸如 eureka、hystrix、zuul 等多达 8 个子库。当然，如果继续细数，各种库的数量将会变的异常庞大[15]。

因此，开发人员从学习到熟练掌握 Spring Cloud ，并且达到在产品当中熟练运用、解决出现的问题的学习周期将会变得尤为漫长，学习成本极高。要真正理解并熟练运用 Spring Cloud 框架，需要把上述这些库的内容全部吃透，否则在遇到技术难题时依然会无法解决。

面对如此繁杂的内容，由于在真实的项目开发和落地的过程中会遇到各种问题，故大部分运维和开发人员通常需要三到六个月的时间才能达到熟练运用的程度。要达到能够独立解决问题的程度话，需要付出的时间将会更加漫长。综上所述，目前的微服务框架带给开发团队，尤其是业务开发团队所需要学习的内容极多，且门槛较高，这是目前主流微服务框架中存在着的较为主要的问题之一。

3.1.2 微服务应用组件过多

微服务架构面临着庞杂的系统实现。

在使用微服务架构进行服务开发时，我们应始终明确，微服务是为了实现功能和性能需求的手段，而不是目的。而业务开发团队的强项往往不在技术细节和其相应实现，而是对于业务逻辑的理解。就使用微服务开发手段而言，有

许多比学习微服务更加艰巨的挑战，例如：微服务的拆分、稳定且易于扩展的良好的 API 设计、跨服务的数据一致性等诸多问题。而更加严重的问题是业务开发团队往往要对旧有的系统进行微服务改造，而这对于业务开发又造成了更进一步的压力施加[16]。

因此，在业务开发团队使用基于 Spring Cloud 的微服务框架进行业务开发时，繁多且复杂的实现往往会成为第二个阻碍开发进度的难点。

3.1.3 服务治理难度较大

使用微服务架构就必须面临一系列有关于服务治理的功能。

服务治理功能总体可以包含三大类：基本功能、高级功能和运维测试类功能。其中，基本功能包括服务注册与发现、负载均衡、故障处理和恢复机制、RPC 支持、 HTTP/2 支持、协议转换与提升等；高级功能包括各类服务和应用的加密、认证授权与权利检验机制、分布式追踪功能、日志监控、度量监控等；运维测试类的功能包括动态请求路由、故障注入、高级路由支持等功能[15]。

以上仅列举出了用于实现微服务各层级中所需要的常见功能。而 Spring Cloud 的服务治理功能是远远不够满足上述微服务的服务治理功能的。如果把这些功能与 Spring Cloud 中的库一一应对，就会发现靠 Spring Cloud 直接提供的功能和类库是远远不够的。很多功能都需要在 Spring Cloud 的基础上自己解决。

3.1.4 跨语言开发难以实现

微服务架构所带来的巨大优势，就是允许不同的服务根据实际需要采用不同的编程语言进行编程。但是，当我们将代码封装到类库和框架中时，微服务的狂语言特性就变的难以实现了。因为当业务开发团队进行代码实现时，通常来说是基于一个类库或者框架来实现的[17]。一旦开始用具体编程语言开始编码的时候你就会发现，一个非常尖锐的问题：每使用一种编程语言，业务开发团队就需要为这种编程语言提供类库和框架。为了解决这个问题，通常只有如下两种解决方案：

第一种，统一编程语言，整个业务开发团队仅使用一种编程语言。

第二种，选择多种编程语言，且每有一种编程语言就写一套相应的类库。可以发现，这两种解决方案都有各自存在的问题：如果选择方案一，则微服务中的“跨语言“特性就成为了无稽之谈；如果选择方案二，就要面临数倍于统一编程语言进行开发工作的工作量。这两种方案无论哪一种都会给业务开发团队带来艰难的挑战。

3.1.5 服务的迭代升级功能难以进行

使用微服务架构，往往意味着有数以百计的服务器端的各项服务，以及数以千计的客户端。在这种情况下，会面临一个严峻的问题：即使业务开发团队有能力利用 Spring Cloud 框架将各种语言的类库都写好，也同样会面临着微服务的版本升级问题[18]。

我们知道，在软件的迭代开发过程中，服务或应用不可能在首次迭代开发的版本就功能齐全、完美无缺、不存在任何漏洞、且分发出去之后无需改动。这种理想状态是不存在的。软件开发必然是从原型版本、到 alpha 和 beta 版本、再到之后的 1.0、2.0、3.0 。在这个缓慢升级的过程中，服务功能逐渐增加，程序漏洞被逐渐修复。但是当软件分发给使用者之后，难免会出现使用者不会立刻进行软件升级的情况[19]。

在这种情况下，客户端和服务器端版本不一致就难以避免。此时，业务开发团队和运维团队不仅要非常小心地维护各个版本之间的兼容性，而且要时刻敦促服务使用者尽快进行软件升级。然而，维护服务版本的兼容性是一件极为复杂的事情：数以百计的服务器端和数以千计的客户端，每个版本都有可能存在不同；且如果使用微服务“跨语言”的特性进行开发，那么这个问题规模就会发展成为一个 N 种语言的笛卡尔积。这种规模的维护工作量，对于后期运维团队来说是难以接受的。

3.2 系统技术路线

3.2.1 微服务通讯层

如今利用微服务架构进行软件开发所面临的挑战，与计算机网络刚刚兴起时，网络通信所带给传统软件开发的挑战非常相似。在面对传统软件开发所面临的“每次开发都需要重新编写用于网络通讯的组件”这项艰巨的挑战时，当时的软件开发者们利用分层抽象与职责专一化的设计理念，以 TCP/IP 网络架构为例，将网络抽象分离出了 TCP 层和 IP 层。由特定的网络层级负责特定的功能，最终使得软件开发者不再需要每次编写软件都重新编写用于通信的软件模块。TCP/IP 网络架构出现前后，传统软件开发架构的对比如下图所示。

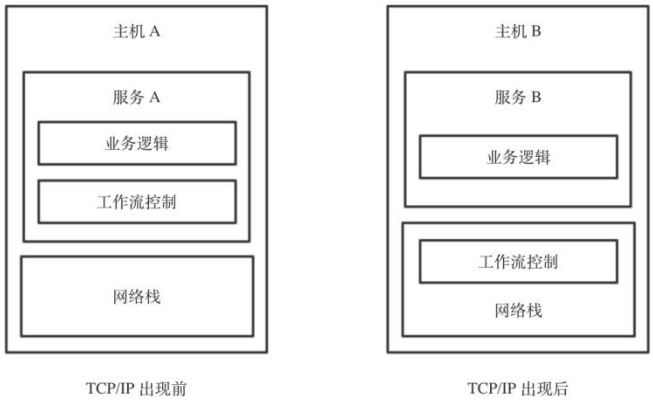


图 3-1 TCP/IP 网络架构出现前后对比图

我们利用微服务框架开发相关应用时，服务通讯的实现不应纳入我们考虑的范畴。每次开发微服务应用都要将有关于服务通讯的全部业务逻辑从头到尾实现一遍是不符合软件工程乃至计算机领域的一贯做法的。

因此，在这种情况下，我们可以借鉴将近 50 年之前计算机领域的先驱们设计网络通讯协议时的做法：将软件开发时每次重复编写的用于网络通讯的代码从业务逻辑中分离并抽象，根据功能将网络访问的技术栈下移为 TCP 协议中的不同层级；同样，在我们的微服务架构中，也可以将负责微服务应用间通讯的功能分离抽象，成为一个全新的“微服务通讯层”。下图展示了微服务技术栈下移前(即如今主流的微服务架构开发模式)与微服务技术栈下移后其层次结构的对比。

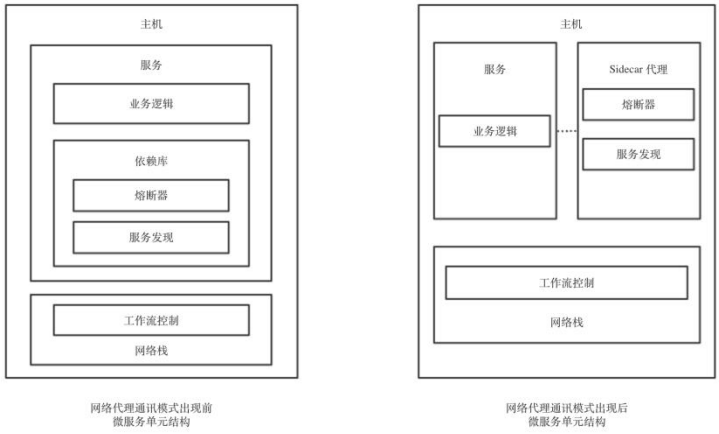


图 3-2 微服务通讯层对架构的改变

我们可以借助使用网络通讯代理来解决微服务之间交互复杂和通讯实现繁琐的问题。在微服务架构中，我们可以尝试使用 Nginx、HAproxy、Apache、Envoy 等反向代理，进而避免微服务之间的通讯产生直接连接。经由代理，我们可以将微服务发出与接收的所有流量通过代理完成，而代理将负责全部的微服务通讯功能，例如：L4 与 L7 层代理、负载均衡等功能。

3.2.2 规范统一服务通讯接口

Sidecar 是微服务架构设计中的特有名词，表示用于负责网络通信功能的模块的总称。任何与微服务应用的具体业务逻辑相解耦而独立存在于微服务基本单元内部的用于网络通信的模块或组件，都可以被称为 Sidecar 。

仅做到微服务通讯功能的分离抽象依旧存在局限。由于代理模块的通讯接口没有进行规范统一，故在这种情况下，每个微服务的通讯代理都只能为特定的基

础设施而设计和服务，通常是与开发代理模块的开发者或开发团队所设计的基础设置和框架直接绑定的，且是基于原有体系进行搭建的。

这种“一个基础设施配合一套定制设计的微服务通讯代理”的做法存在着诸多限制，而其中最大的难题就是无法做到普适通用：我们为了这个微服务框架设计的微服务通讯代理没办法直接给另外的微服务架构使用，也就是说，我们依然没有做到彻底地功能部分隔离解耦。例如， Airbub 公司所设计的微服务通讯代理一定要配合 zookeeper 框架才能进行使用，而 Netflix 公司的服务通讯代理一定要使用 Eureka 框架；这两家公司的通讯代理不能做到互相通用。

因为存在着这样的特殊背景和需求，故服务代理不能做到通用。这种情况的出现原因，是因为通讯代理本来就是基于原有体系而实现的。虽然通讯代理无法做到通用，但在原有中通讯代理却能高效地工作，而且这样还有助于各大公司形成“技术壁垒”，使自己的微服务架构处于持续不断的技术优势中，故各大公司没有意愿也没有动力去将微服务通讯的部分进行分离。

综上所述，我们需要将微服务的通讯代理的接口进行统一的规范，或单独为其编写能够普适大多数流行的微服务框架的抽象接口层。无论我们选择何种方式，都要将微服务间的通讯逻辑完全从业务逻辑中剥离，进而实现“通讯模块通用”的目标。利用 Sidecar 进行微服务基本单元间通信的示意图如下。

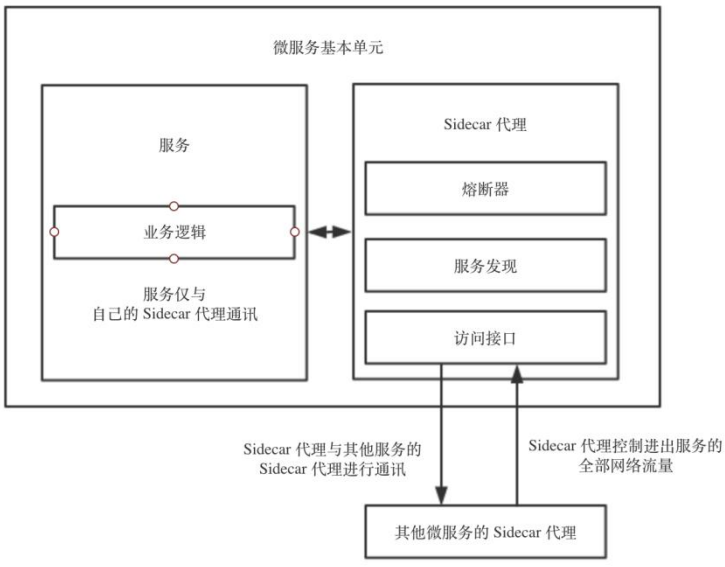


图 3-3 基于 Sidecar 模式的微服务间通讯

3.2.3 微服务监测系统

同理，集中式的监测系统也不可或缺。对于运维人员来说，实时掌握微服务集群的各种指标是尤为重要的。微服务应用往往都是基于一个分布式服务集群。对于集群而言，使用跟踪、监控和日志记录的方式将会把整个微服务集群

统一成为一个整体。同时，通过集中式的监测系统，也可以让运维人员直观地了解服务性能如何影响上游和下游服务的功能。

集中式的监测系统也应具备自定义仪表板的功能。自定义仪表板可以提供对所有服务性能的可视性，并让运维人员实时了解到微服务的性能如何影响到其他微服务；通过负责策略控制和遥测收集的功能组件，我们可以将监测系统与各个基础架构后端的实现细节完全隔离，并未运维人员提供对网格和基础架构后端之间所有交互的细粒度控制。

集中式的监测系统的所有这些功能，可以让运维人员更有效地设置、监控和实施服务上的各种操作；更为重要的是，它功能使运维人员快速有效地检测和修复微服务应用中出现的问题。

3.2.4 可视化控制面板

对于微服务模块与功能繁多、服务间调用关系复杂等问题，我们应提供集中式的控制面板加以解决。控制面板应同时具有主机命令行控制和浏览器可视化界面控制两种模式。通过命令行或可视化界面所提供的简单规则对整个微服务应用进行控制，例如：规则配置和流量路由。用户应使用该统一的控制面板轻松设置 A/B 测试、金丝雀部署和基于百分比的流量分割等重要任务。

集中式的控制面板将会给运维人员带来巨大的便利：由于微服务往往部署在成千上万的主机之上，如何进行统一管理就是一个难以解决的问题。而利用集中式的控制模版，因为人员就不必关注每个微服务的诸如位置信息、流量信息等诸多细节问题，而对整个微服务的服务应用进行集群级别的控制和调度。这种做法将会极大程度上减少运维人员的工作量。

3.3 系统核心功能需求

综合上文所述，作为一款基于服务网格开发模式的可视化监控系统，应具备五点核心功能，下文将逐一说明。

3.3.1 物理资源监测与可视化

目前微服务的开发中，Docker 容器技术已然成为主流，故下文以 Docker 为例来阐述虚拟物理资源的检测与可视化。在服务网格开发模式中，提供具体功能的服务或组件会被封装在容器，即 Docker Container 中，进而实现从从系统开发、部署到运维的整个过程的微服务化。而容器作为一个虚拟化的独立个体，可以被看作一个运行于物理主机上的独立的操作系统。同时，作为微服务载体的容器也必须保证其各项虚拟的物理资源达到容器内微服务的最低需求，进而才能保证微服务的顺利运行。因此，容器中虚拟物理资源的监测就变的尤为重要。

容器内可监测的物理资源包括但不限于：容器中的 CPU 使用率、容器中的内存使用率、容器中的存储空间占用率等。通过监测上述指标，运维人员可以

直观明确地在短时间内发现微服务可能出现的问题与隐患，进而做出适当调整。将容器的虚拟物理资源可视化也可以将系统可能出现的问题细化、精确到具体的服务或组件，进而实现“精细灾备”、“细粒度扩容”等使用传统开发模式难以或无法实现的细粒度策略。

网络资源在微服务架构中的重要性同样不言而喻。微服务架构中的每个微服务互相通讯与互相调用，都需要通过网络进行信息的交互与传输。在信息系统运维中，一个服务或组件出现问题，往往意味着其上下游服务和组件都将因此受到影响，进而发生阻塞或无法提供正常服务。因此，微服务中的网络资源监测是整个系统中服务质量保证必不可少的一个环节。

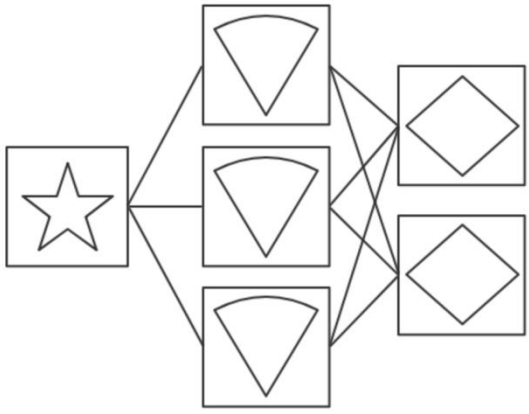
网络资源可以包含许多方面。其中，最为关键的网络资源应包括但不限于网络负载、丢包率、网络传输速度和网络带宽等。由于在微服务架构中，服务间的相互调用与通讯通常基于 REST API 模式，故服务间的“成功请求比”也必须纳入网络资源监测的范畴。通过进行上述多方位的网络资源监测，进而实现对整个服务网格的各个微服务之间以及服务网格与外部网络之间的监测。

在可视化方面，微服务中的网络资源通过使用折线图或心率图等方式，可以将晦涩繁琐的数据直接以图表形式呈现在运维人员的面前，以大幅减轻运维人员的工作量，使网络资源情况清晰、直观地展现在运维人员的面前。

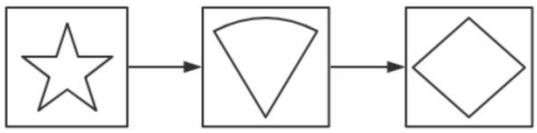
3.3.2 拓扑关系与调用链可视化

微服务架构的基石，就是确保各个微服务间的调用关系保持清晰准确。我们知道，在一个成熟的给予微服务开发模式进行开发的应用中，往往同时会有成千上万个微服务同时运作。在如此庞大又繁杂的情况下， 能够实时监测各个微服务间的调用关系及微服务上下游服务或组件的负载状况，对于发现错误隐患并提前实施预防措施都是极为有帮助的。因此，在本微服务监测与可视化系统中，加入微服务间拓扑关系的可视化功能是非常必要的。该功能不仅可以使运维人员直观地看到整个微服务架构中各个微服务之间的调用关系，还能帮助运维人员及时发现错误隐患，并提前做出防范措施；如果系统中的部分微服务已经发生故障，使用微服务间的拓扑关系及负载状况的可视化功能将会帮助运维人员快速、准确地定位错误，以便其迅速采取措施。

随着业务越来越复杂，系统也随之进行各种拆分，特别是随着微服务架构和容器技术的兴起，看似简单的一个应用，后台可能有几十个甚至几百个服务在支撑；一个前端的请求可能需要多次的服务调用最后才能完成；当请求变慢或者不可用时，我们无法得知是哪个后台服务引起的，这时就需要解决如何快速定位服务故障点。



服务间的拓扑关系示意图



调用链示意图

图 3-4 微服务的拓扑关系与调用链

3.3.3 微服务访问流量控制

相比于传统的软件一体化架构，微服务架构的优势之一是提供了简便的持续部署、持续集成策略。使用微服务的服务网络架构进行蓝绿部署、滚动部署以及金丝雀部署在采用微服务架构的企业中是普遍且易于进行的。为了支持微服务的服务网络架构这一优势，本系统应具备微服务的访问流量管理于控制功能，使开发人员或运维人员通过命令行工具或可视化界面，进行方便快捷的微服务访问流量的管理与控制，从而实现微服务架构中对于各个微服务的细粒度控制，且必须保证操作的简单快捷、易于理解，尽量避免繁琐庞杂的配置工作。

3.3.4 故障服务的检测与处理机制

故障服务的检测是微服务中服务质量保障的关键问题之一。系统需要对微服务进行定期检测，以便当微服务发生错误或故障时及时同时运维人员或报警；同样，对于故障服务的应对机制也非常重要，系统需要在服务发生错误或故障时及时让运维人员知晓，从而使运维人员及时进行错误处理。我们可以按照一定时间间隔（如每 5 秒进行一次扫描）对微服务各项关键的虚拟物理资源及网络资源进行查询访问，通过默认阈值或提前由运维人员设置阈值的方式进行及时的错误检测。在故障服务的应对机制方面，我们可以使用邮件服务或短信

服务的方式，当系统检测到错误或故障时，立刻向运维人员的邮箱发送警告邮件，或向其手机发送警告短信，使运维人员在第一时间得知微服务出现故障，并为其及时做出处理和应对提供可能。

3.4 本章小结

本章主要对该可视化监控系统进行系统需求分析。从目前微服务开发所面临的问题与挑战入手，列举了五点微服务开发的难点；针对这些难点，本文明确地提出了该系统所应具备的设计目标，并提出相应的解决方案，以此作为系统的核心功能需求。

4.微服务架构软件的可视化监控系统设计

4.1 系统设计目标

该微服务可视化系统的设计有几个关键目标，这些目标意在使系统能够应对大规模流量和高性能地服务处理至关重要。这些目标主要包含四点，下文将逐一说明详细阐述。

4.1.1 系统实现对使用者透明

若该系统想要被广泛的运维人员采纳，则应该让使用该系统的运维和开发人员只需付出很少的成本就可以从中受益。为此，该系统基于 Istio 服务网格的设计特点，将自身的通讯模块自动注入到服务间所有的网络路径中。由于 Istio 使用 sidecar 代理来捕获流量，并且在尽可能的地方自动编程网络层，以路由流量通过这些代理，因而无需对已部署的应用程序代码进行任何改动。以 Kubernetes 为例，在 Kubernetes 中，代理被注入到 Kubernetes 的最小单元 Pod 中，通过编写 控制路由规则的 iptables 来捕获流量。注入 sidecar 代理到 pod 中并且修改路由规则后，Istio 就能够调解所有流量。这个原则也同样符合性能需求。当将 Istio 应用于部署时，运维人员可以发现，为提供这些功能而增加的资源开销是很小的。所有组件和 API 在设计时都必须考虑性能和规模。

4.1.2 可扩展性

随着运维人员和开发人员越来越依赖该系统为他们提供的功能，系统必然和他们的需求一起成长。虽然我们期望在该系统迭代开发的后续版本中继续添加新功能，但是预计到对于运用该系统的运维和开发人员来说，最大的需求是扩展策略系统、集成其他策略和控制来源，并将网格行为信号传播到其他系统进行分析，因此策略运行时支持标准扩展机制以便插入到其他服务中；此外，该系统还允许扩展词汇表，以允许基于网格生成的新信号来执行各种策略。

4.1.3 可移植性

该系统作为一个拥有许多开源模块的集成系统，在使用时其生态系统将在很多维度上存在差异。因此，该系统必须能够以最小的代价运行在任何云平台虚拟主机或预置环境中。将基于该系统的服务移植到新环境应该是轻而易举的，而使用该系统将一个服务同时部署到多个环境中也必须是可行的，例如，在多个云上进行冗余部署。

4.1.4 提供统一规范化接口

在该系统中的各个服务间的 API 调用中，策略的应用使得可以对网格间行为进行全面的控制，但对于无需在 API 级别表达的资源来说，对资源应用策略

也同样重要。例如，将配额应用到机器学习训练任务消耗的 CPU 数量上，比将配额应用到启动这个工作的调用上更为有用。因此，策略系统作为独特的服务来维护，具有自己的 API，而不是将其放到代理 sidecar 中，这容许服务根据需要直接与其集成。

4.2 系统总体设计

根据上文中系统设计目标与系统需求分析，在明确了该系统的开发目标和各项需求后，为了验证微服务的可实施性，本文使用了众多开源项目和开源工具，基于 Docker 技术和相应的容器管理平台 Kubernetes 集成部署出了一套基于上述微服务架构的云端监控平台。 它是使用微服务架构开发的应用进行测试、部署和运维的集成平台，既面向开发者提供微服务架构的管理工具和相应运行环境，也面向普通用户提供基于微服务架构的各项服务。

4.2.1 系统架构设计

该微服务监控系统的宏观架构设计如下图所示。

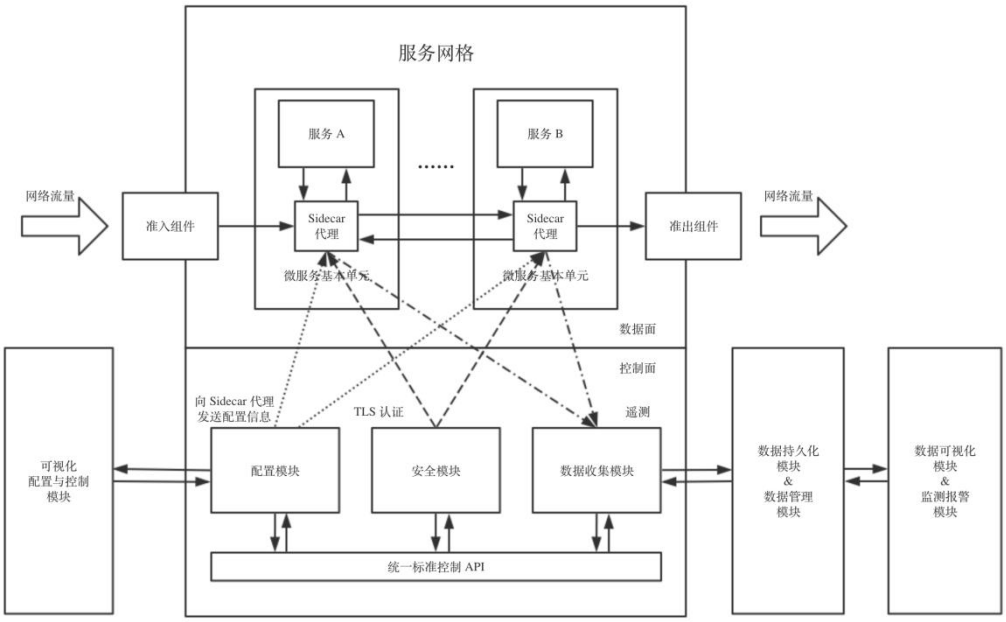


图 4-1 系统设计架构图

本系统架构主要由如下五个主要功能模块所组成。它们分别为：服务网格的数据面、服务网格的控制面、用于数据存储和数据监测管理的数据持久化和监测模块、用于数据展现的可视化模块以及用于配置各个微服务组件的配置可视化模块。下面，我们将逐一简要介绍各个模块。

服务网格的数据面是所有微服务应用的实例部署并运行的所在地。所有微服务应用都会通过虚拟化容器技术运行于其中。正如服务网格的设计初衷那样，每个微服务应用的基本单元除了自身业务逻辑之外，还包含着一个名为

Sidecar 的网络代理，所有流入和流出该微服务应用的流量将全部由 Sidecar 代理进行控制和转发。除了各个微服务应用外，服务网格的数据面还包括一个准入控制组件和一个准出控制组件。准入控制组件将控制所有欲进入服务网格集群的流量，并将所有流经其自身的流量按照配置规则进行流量控制；同理，准出控制组件控制着所有集群内部欲流出集群的流量，这些流量包括集群内部资源访问集群外部的服务、API 或数据库等多种形式。

服务网格的控制面主要用于控制数据面中所有微服务单元的生命周期、网络流量等，是整个微服务可视化监控系统的中枢，作用极为重要。其主要由四个主要组件所构成：用于配置各个微服务应用 Sidecar 代理的代理配置组件、用于保证集群内安全性的安全组件、用于遥测收集集群中各种数据的数据收集组件以及将这三个模块整个并提供统一控制操作的统一标准控制 API 。用于通过控制面提供的统一标准控制 API 对整个微服务架构中的每个微服务应用根据需求进行各种各样的控制。

数据持久化和监测模块主要配合服务网格控制面中的数据收集组件进行使用。该模块通过 pull 方式按一定时间间隔向数据收集模块索要数据，并将数据收集模块中缓存的服务网格集群中的各项数据指标持久化地存储。同时，它还可以为可视化模块提供用于可视化的源数据。在这层意义上，该模块主要可以看作是一个数据流通的中间件，起到传递集群数据至可视化模块的作用。

数据可视化模块主要是用于展现微服务集群中纷繁复杂的各项数据，通过图表的形式直观地呈现在运维人员眼前。同时，数据可视化模块还应具备设置阈值的报警功能，一旦某项指标超过了设置的报警值，就要报警通知运维人员，以便第一时间解决问题。

配置可视化模块主要起到锦上添花的功能。由于目前主流的微服务架构和服务网格是基于 Kubernetes 和 Istio 而实现的，在开发和运维过程中将会有着针对不同需求的各种各样的配置文件被编写出来。这些文件如果不经管理的话，将会随着应用运行时间的增长而越来越多、越来越繁杂。因此，该模块主要希望通过借助可视化的方式，帮助开发和运维人员管理存储于 Kubernetes 主节点上的各个配置文件，以减轻他们的负担。

4.2.2 系统层次结构设计

该微服务监控系统的系统层次结构设计如下图所示。

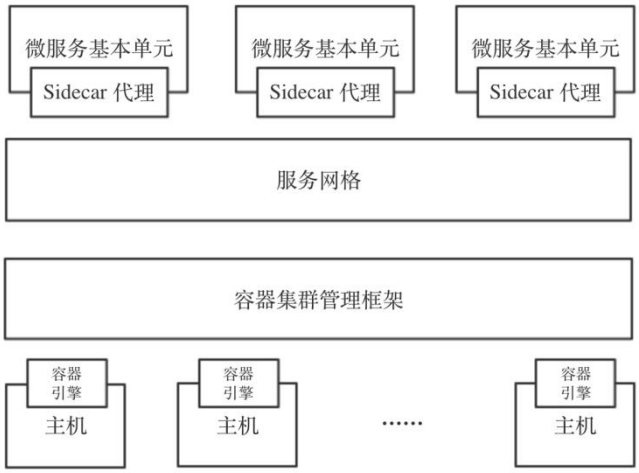


图 4-2 系统层次结构设计图

由于这张层次结构图主要分为四个层次面，故我们将按照自底向上的顺序进行阐述。

4.2.2.1 基础设施层

由于微服务架构是基于云计算技术的发展而兴起的，因此，本系统架构设计也同样是根据云计算技术中最关键的底层技术——容器虚拟化技术作为基础进行设计的。该微服务可视化监控系统的最底层——我们将其命名为“微服务基础设施层”，是由一台或多台安装有虚拟容器技术引擎的，带有操作系统的主机所组成的。每台主机可以选择安装有操作系统的物理实体主机或由云服务商提供的安装有操作系统的虚拟主机。在本系统架构设计中，我们以目前业界最流行的虚拟化容器技术—— Docker 技术为例，作为我们整个微服务可视化监控系统中的通用虚拟化容器。“微服务基础设施层”的职责，是为上层提供可使用的虚拟容器资源，包括可用的 CPU 资源、内存资源、硬盘资源和各项网络资源等。因此，在此层级中的每一台主机都要安装有操作系统以便调用各种资源；同时，由于我们的系统是针对微服务架构进行设计的，故主机必须安装有 Docker Engine 来获取系统的各种可用资源，并根据需求启动、关闭请求相应资源的容器。

该层级的作用就如同摩天大厦中的地基，是一切微服务应用成功运行的基础。

4.2.2.2 容器资源调度层

由于一台台独立的安装有 Docker Engine 的主机只能独立工作，而不具备统一调度的功能，因此“容器资源调度管理层”就该发挥它的作用了。“容器

资源调度管理层”的职责是将“微服务基础设施层”的每台独立主机进行统一管理，将底层每台主机的各项资源细节进行统一整合，将所有可利用的主机资源抽象化为主机资源池，并向上层提供简单、透明的请求调用接口。这意味着，各个主机上的每一个容器的生命周期都将由“容器资源调度管理层”负责管理。由此可见该层的意义重大，因此本文选用了目前在业界功能最为健全、稳定性最佳且最为流行的容器管理框架 —— 由 Google 公司和 IBM 公司共同研发的 Kubernetes 作为整个微服务可视化监控系统的“容器资源调度管理层”。Kubernetes 所具备基本功能初步符合我们微服务可视化监控系统的需求；而且按需添加或删除底层主机、按需分配容器资源和基于 ETCD 分布式数据库的主机间交互等功能更能起到锦上添花的作用。

4.2.2.3 微服务交互层

正如前文所叙述，目前微服务架构所面临的的最大难点，就是微服务与微服务和微服务与外部间的通讯问题在每次开发过程中都需要业务开发人员亲手实现一次；而我们提出的改进办法为：技术栈下移，从服务中分离出负责通讯的部分，并在架构层面抽象出独立的“微服务通讯层”；而这个层级就是针对上述改进所作出的实现。“微服务通讯层”的职责是将原先耦合进各项微服务中的负责服务间通信的模块，从服务代码中解耦，由此抽象而成的独立层次。在“为服务通讯层”中，我们主要使用网络通讯代理来解决微服务之间交互复杂和通讯实现繁琐的问题。每个微服务中注入的网络通讯代理由该层统一负责管理和配置。使用这种将微服务与其通讯功能抽离并独立管理的方式将会使整个微服务架构的功能划分更加明确、管理和配置服务通讯更加便捷；同时，也会极大地减轻业务开发人员和运维人员的工作量。

4.2.2.4 微服务应用层

该系统总体架构的最上层被称为“微服务应用层”。如同我们所熟知的互联网七层架构和 TCP 的四层架构一样，本架构的最上层也同样是各类微服务应用所真正发挥作用的层级。由于“微服务基础设施层“为整个系统提供相关资源、“容器资源调度管理层“负责资源的调度和管理以及“微服务通讯层“负责服务间的通讯交互；这三个层次分工明确且包含了整个微服务架构所需要的几乎全部功能，为最顶层的“微服务应用层”提供了功能齐全性能稳健的强大支持。所有微服务提供的具体服务都在这个层级上得到体现。

4.2.2.5 系统架构的设计优势

由上文可知，该为微服务可视化监控系统总共由四个层级所构成。它们自底向上分别是：“微服务基础设施层“、“容器资源调度管理层“、“微服务通讯层“和“微服务应用层”。

各层次之间职责清晰且相互透明，这种架构设计将会同时对开发人员和运维人员产生益处。由于各层次之间的透明，故开发人员在开发微服务应用时，将不再需要将注意力和精力耗费在微服务的容器管理或微服务的通讯等杂项上，而仅仅需要将全部时间和精力投入到具体业务的开发之中，也就是其只需要关注“微服务应用层“即可；运维人员也可以从中受益。由于系统架构设计中每个层级清晰的职责划分，因此在系统运维或系统发生故障时，运维人员可以通过系统层级准确地定位到出现的问题或发生的故障。

4.2.3 系统集成与部署设计

该微服务监控系统系统整体架构的集成与部署方式如下图所示。

本微服务可视化监控系统由于在底层架构设计的技术选型中选用了基于 Docker 的虚拟化容器技术以及基于 Kubernetes 的虚拟容器管理框架，故集成与部署方式亦采用相应的主从模式与之相匹配适应。

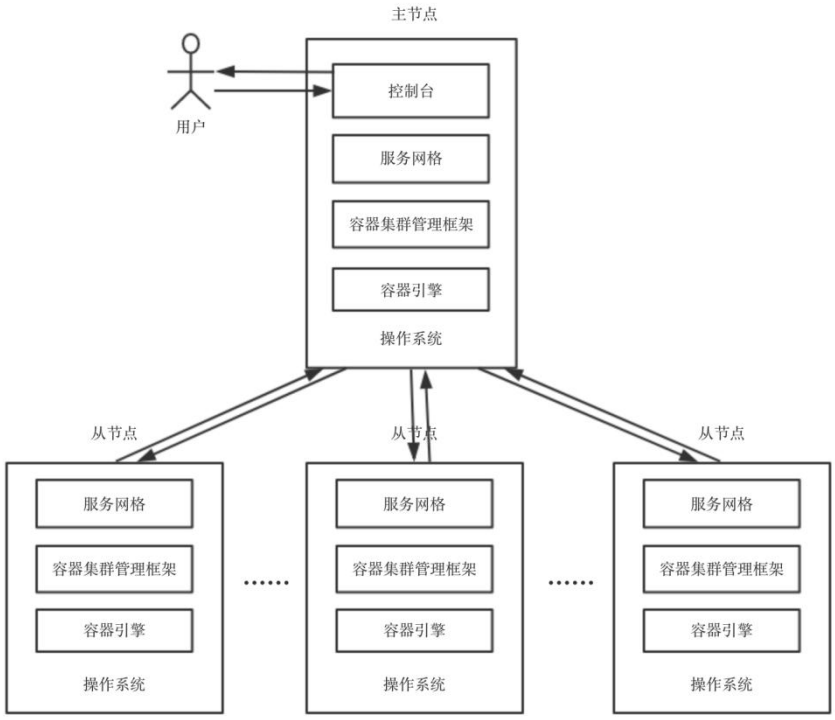


图 4-3 系统集成与部署图

上图以每一台装有操作系统的物理主机或虚拟主机为最小部署和集成单元。依据上文中的四层系统架构宏观设计，每台主机的部署方式如下。

首先，每一台主机必须安装有操作系统，这是获取和调用各项计算机物理资源的基础，也是作为每一台主机最底层的软件保障。在操作系统之上，由于我们是基于微服务架构进行整个系统架构的设计，故需要在操作系统之上安装

Docker 以使用虚拟化容器技术。Kubernetes 是管理和调度容器的关键，因此在每台已经安装 Docker 的主机上继续部署 Kubernetes。

依照上述方法依次部署数台主机，我们便会得到数个基本具备组成容器集群的主机群。在这种情况下，利用 Kubernetes 所提供的添加主从节点的功能，我们可以将所有具备进入集群条件的主机全部加入集群中。加入集群的主机总体分为两类：负责调度指挥各个主机及其容器生命周期的 Manager 主机和负责听从调度并实施具体工作的 Worker 主机。至此，一个具备虚拟容器调度及管理的容器集群就初步部署和集成完毕了。

在容器集群成型的基础上，业务开发人员和运维人员就不需要再关系诸如容器的生命周期管理等细节问题了了。他们只需要通过 Manager 主机上通过开放的容器集群管理 API 来进行开发和监控即可。之后不论是“微服务通讯层“的部署还是具体到“微服务应用层“中的应用开发，都可以通过虚拟容器和镜像的方式，通过 API 来进行安装和集成。

以上即为本微服务可视化监控系统，依据系统宏观架构所设计出的基本集成和部署方法。

4.3 系统详细设计

4.3.1 微服务的基本单元

每个微服务应用都可以由其中几个功能模块所构成。在本文中，为了方便读者的阅读，我们将微服务的最小单元称为 MicroService Pod 。一个简单的 MicroService Pod 的基本结构如下图所示。

MicroService Pod 作为本微服务可视化系统中的基本单元，其主要由其中的两个部分所构成：负责服务间通信和服务与集群外部通信的 Sidecar 代理节点，以及真正具备服务功能的服务应用 Service。我们的目的是将微服务的服务应用层与服务通信层实现解耦，而这种设计模式恰巧能够满足我们的需求：每个 MicroService Pod 中，服务实现具体业务逻辑的 Service 都只与自己的代理 Sidecar 进行通信，而 Sidecar 再负责与其他 MicroService Pod 的 Sidecar 通信以实现微服务间的交互访问，或再与集群外部网络上的节点进行交互以实现微服务与集群外部间的访问。

将 MicroService Pod 解耦为这两个部分所带来的好处是显而易见的：首先，它将每个微服务的基本单元从功能上进行解耦，使微服务基本单位的架构层次更加清晰。通信由特定的 Sidecar 代理统一负责，因此业务开发人员只需要将精力集中在具体的业务逻辑上，而不再需要过度注意网络通讯功能的开发；其次，使用统一的 Sidecar 代理使得配置规则变得清晰简单，运维人员只需要遵循一套特定的标准化配置方法即可对微服务架构中的左右微服务进行流量控制和管理；最后，网络通信由一个统一的组件负责也使得监测和控制变得更加简单可行：我们只需要使用一个统一的规范来监测所有微服务基本单位中的一个相同的组件，就可以监控整个微服务集群的流量状况。

4.3.2 服务间通讯

MicroService Pod 之间的通讯如下图所示。

微服务之间的通信是整个微服务架构中最重要的部分，其通信的质量往往直接影响着微服务架构的整体性能。基于上文 MicroService Pod 的设计，我们可以得知，微服务间的通信主要依赖于每个微服务基本单元中的 Sidecar 网络代理。每个流经 MicroService Pod 的流量由 Sidecar 进行统一的转发，并遵循由运维人员进行统一配置的 Sidecar 规则。

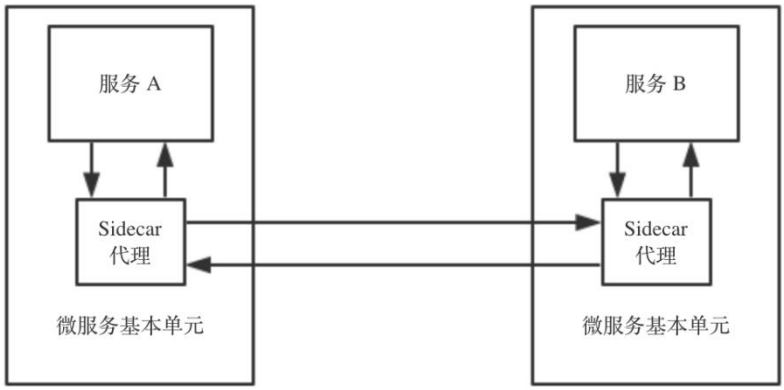


图 4-4 微服务基本单元间的通讯

以上图为例，我们可以从图中清晰地看到 MicroService Pod 间进行通信的基本流程：假设服务 A 需要请求服务 B 中的某项具体服务，则服务 A 需要先将访问请求发送给自己的 Sidecar 代理，由 Sidecar 代理读出服务 A 发送请求中的如目的地址等重要信息，根据这些信息将请求发送给相应 MicroService Pod 的 Sidecar(即为服务 B 所对应的 Sidecar)；在服务的 Sidecar 代理接收到请求后，也同样需要读出请求的各项重要信息，以确保该项请求确实是发送给服务 B 的。确认无误后，服务 B 的 Sidecar 代理将其请求发送给服务 B 。至此，一个完整的微服务间的请求过程结束。而服务 B 响应服务 A 请求也遵循同样的过程。整个微服务集群中的所有微服务基本单位全部遵循这种通信规则，即可借助 Sidecar 实现统一标准的流量监控与管理，为运维人员节省大量人力和时间成本。

4.3.3 监测系统通讯层和应用层的具体设计

基于上述微服务可视化监控系统的宏观软件结构，在该系统的各功能组件之间，根据层次结构的划分可以将处于最顶端的两个层级分为“微服务通讯层“和 “微服务应用层”。

服务网格是上文中系统宏观架构设计中，“微服务应用层” 与 “微服务通讯层“ 的总称。其中，“微服务应用层” 由一至多个 MicroService Pod 所组成，是各种微服务的集合。由于前文已经对 MicroService Pod 和其间的通信方式做了详细阐述，故此处不再赘述。

“微服务应用层”之下的 “微服务通讯层” 主要由三大功能模块和一个统一标准控制 API 所组成。这三个模块分别是：负责管理和分发配置项的配置模块、负责进行整个服务网格遥测数据收集与汇总的数据收集模块以及负责通信安全的安全模块。这三个功能模块的基本原理全部通过每个 MicroService Pod 中的 Sidecar 代理进行实现，通过对 Sidecar 代理的各种配置和各项信息的收集以满足各个模块对于功能的需求。

除此之外， “微服务通讯层“的另一个重要模块，统一标准控制 API 所负责提供程序或用户使用的统一编程接口，以对整个 Service Mesh 的各种复杂实现提供简单易懂的配置方法。

4.3.4 可视化监测与控制模块

基于上文中该系统的设计目标和设计需求中所明确提出的微服务可视化监测功能，本文给出其具备相应功能的基础设计架构和主要监测功能实现和操作流程，如下图所示。

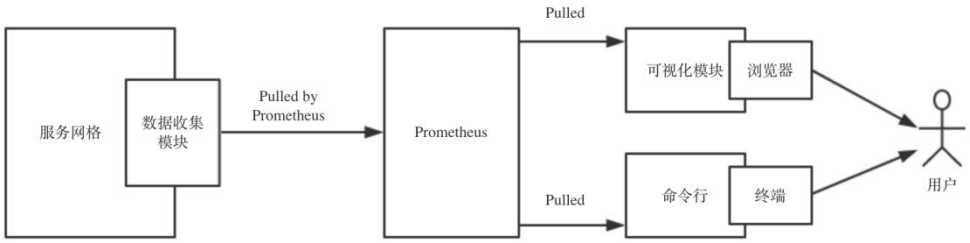


图 4-5 可视化监测设计图

负责微服务可视化监测功能的模块主要由三大部分组成：整体微服务集群中由包含 Sidecar 代理的 MicroService Pod 组成的 Service Mesh，负责全体微服务间流量传递的信息采集；处于 Service Mesh 与终端用户界面的非关系型数据库，负责缓存和传递由 Service Mesh 收集到的各项微服务网络数据；可视化界面或命令行界面，负责与运维人员进行可视化或非可视化的交互。Service Mesh 采集监测到的数据传递到运维人员眼前的基本过程如上图箭头方向所示。首先，集群中各个 Sidecar 利用其缓冲区采集并缓存通过其自身的各项数据，并由集群中负责信息收集的功能模块以定期拉取(pull)的方式进行统一汇总。其次，介于中间位置的非关系型数据库采用定期利用拉取的方式将

Service Mesh 中信息收集功能模块缓存的数据存储至数据库中，并持久化一段时间(持久化的具体时长可以由运维人员进行统一设置)；此处在架构中采用非关系型数据库而不是关系型数据库的最大考量，即各种类别微服务的信息，其格式也都往往是不相同的，因此在此采用非关系型数据库以便兼容存储各种类型各种格式的信息。最后，可视化模块或命令行工具也同样从非关系型数据库中定期获取各项信息数据，并显示在运维人员的面前，供其实时监控微服务集群的整体状况。

同理，基于上文中该系统的设计目标和设计需求中所明确提出的微服务可视化控制功能，本文给出其具备相应功能的基础设计架构和主要控制功能实现和操作流程，如下图所示。

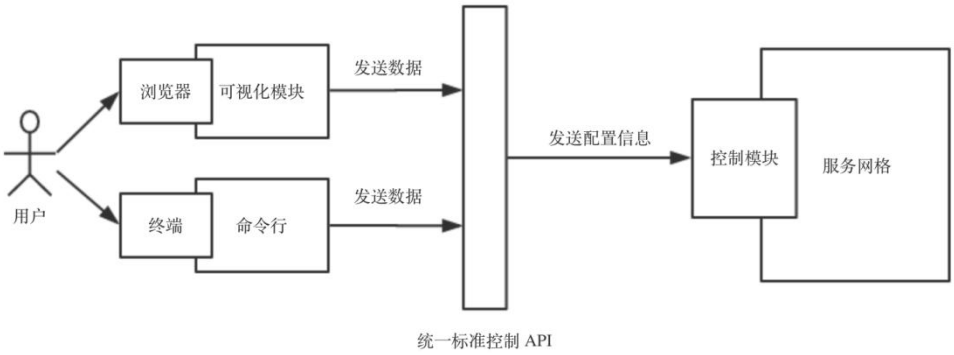


图 4-6 可视化控制设计

负责微服务可视化控制功能的模块同样由三大主要部分组成：与运维人员直接进行交互的可视化模块或命令行工具；负责进行配置信息接收的统一标准控制 API；以及负责最终接收并具体配置整体 Service Mesh 中各个 Sidecar 的配置模块。

运维人员通过点击可视化界面的按钮或勾选配置项等操作，或使用命令行工具，依照提前规定好的特定 Sidecar 的统一配置规则对集群进行控制信息的输入。命令行工具或可视化模块的后端逻辑在接收到运维人员的输入后，向统一标准控制 API 以既定格式发送输入的信息。Service Mesh 的配置模组接收到由统一标准控制 API 发送的翻译成为特定格式的配置信息后，将这些配置信息传递给微服务集群中所有需要配置的 Sidecar 代理。以这样的方式，运维人员就可以以统一的标准对集群中各种各样不同形式的微服务实现统一的控制和管理。

4.4 本章小结

本章主要进行详细的系统设计。首先，针对第三章所提出的核心功能需求，笔者从软件设计的角度提出了该系统所需具备的五点基本特性，并以图文搭配的形式，重点设计了该系统的系统架构、层次结构和集成与部署方法。在这些关键设计完成后，笔者又针对每个功能，从系统架构和功能流程的角度进行了详细设计，以求设计切实可行，能够在后续的系统实现章节成功落地，并进行相应的测试与验证。

5.可视化监控系统实现

根据上文基于微服务架构的可视化监控系统设计中提出的各种设计理念和思想，本章着重讲解该系统由设计转换为实际系统时的实现方法以及技术细节。本章首先讲解多台独立主机通过容器集群管理框架组合为一个容器集群的方法，并在此基础之上，采用基于数据驱动的逻辑进行写作，按照被监测数据的“监控数据的收集、持久化、分析与处理、可视化管理、异常检测与报警”等几项主要内容组织撰写。

5.1 容器集群管理框架的搭建

容器管理平台指的是负责对容器的生命周期和其他特性进行管理的平台，通常由多台安装了容器管理工具的物理主机或虚拟云主机构成的集群所组成。Kubernetes 是目前最为通用的容器管理平台，它能够将所有安装相同版本的 Docker 主机进行集群化并统一进行管理、对启动的容器进行统一调度。其主从架构的模式使得容器管理平台的可用性和性能都得到了较高的保障。

在系统的实现中，我们在用于搭建该微服务可视化系统的主机上，安装相同版本的 Docker 并使用 kubeadm 工具安装 Kubernetes。选定至少一台主机作为集群的主节点，并在此台机器上利用 kubeadm 自带的 kube init 命令进行启动和相应配置(包括集群的各种组件、访问位置、端口号等)。当主节点初始化完毕后，利用 kubeadm 生成用于添加进入集群的密钥，并在从节点上利用 kubeadm join 命令运行这段密钥，即可将从节点添加进入集群中。

在初步形成容器管理平台的集群后，我们还要通过官方提供的各式 addon 插件对集群的网络进行搭建。在这里我们采用开源项目 calico 的组网模式，对集群进行网络配置。

在所有工作完成之后，在主节点使用 kubectl get nodes 和 kubectl get pods 命令查看所有节点和集群中的所有 pods 的运行情况。若所有 node 的状态都是 Ready 且所有 pod 全部为 Running 状态，则证明作为微服务基础设置部分的容器管理平台搭建成功。

5.2 监控数据的收集

监控数据的收集，主要是由服务网格数据面中每个微服务应用的 Sidecar 代理和服务网格控制面的数据收集组件共同作用而完成的。可以看出，作为整个微服务架构基础设施层的服务网格在此步骤中的作用不可或缺：服务网格能够提供灵活的模型来执行授权策略，并为网格中的微服务应用收集各项遥测数据。

服务网格的控制面旨在提供用于构建服务的支持功能。它们包括访问控制系统、遥测捕获系统、配额执行系统以及计费系统等等。服务传统上直接与这些后端系统集成，创建硬耦合，还有沾染特定的语义和使用选项。

同样，服务网格也提供统一抽象，使得其自身可以与一组开放式基础设施后端进行交互。这样做是为了给运维人员提供丰富而深入的控制方法，同时不给微服务的业务开发人员带来负担。服务网格旨在改变层与层之间的边界，以减少系统复杂性，消除服务代码中的策略逻辑并将控制权交给运维。

其中，服务网格控制面的数据收集组件主要负责提供策略控制和遥测收集：在每次请求执行先决条件检查之前以及在每次报告遥测请求之后，Sidecar 代理在逻辑上调用数据收集模块。该 Sidecar 代理具有本地缓存，从而可以在缓存中执行相对较大比例的前提条件检查。此外，Sidecar 代理缓冲出站遥测，故其实际上不需要经常调用数据收集组件，从而保证了运行时的性能。

在本系统的实现过程中，上述的服务网格我们使用开源项目 Istio 进行实现。而相应的数据收集组件，就是 Istio 中的 Mixer 组件来负责。每个 Sidecar 代理使用 Istio 官方建议的 Envoy 来实现。由于 Envoy 本质上是一个带有缓存功能的负载均衡调度器，且分布于整个 Istio 服务网格的每一个微服务应用之中，因此有必要对其缓存的各项微服务数据和指标进行汇总，而这也正是 Mixer 的职责所在。

Mixer 能够定位到位于服务网格中所有的 Envoy，并通过遥测的方式以一定时间间隔为周期轮询地收集每个 Envoy 中的数据，并统一汇总，缓存在自身的 cache 中，等待后续的数据持久化模块进行数据的同步以及持久化存储。

5.3 监控数据持久化

数据持久化主要由可视化监控模块中的持久化模块负责。其主要功能是将从服务网格中收集到的数据进行一定的格式化信息的处理并将之持久化，以便其余模块（ 如数据可视化模块、监测与报警模块 ）的使用。

在本系统中，该模块的最大作用除了存储数据，还有着作为连接各个模块的“数据中转站”的作用。在功能中有需要用到服务网格数据的模块，不直接从负责进行数据收集的模块中进行索取，而是通过该模块将数据提取并进行使用。

利用 Prometheus 的数据存储接口，我们可以将服务网格中收集而来的各种结构的数据进行持久化。由于 NoSQL 的设计特性使得 Prometheus 能够兼容系统中各个模块所产生的各种不同结构的数据。由于 Istio 服务网格所产生的数据都被缓存在其 Mixer 组件之中，若想对其进行持续的观测并实现后续的诸多功能，就必须将其相对持久地存储下来，这就是 Prometheus 在这个模块中最大的作用。

5.4 监控数据分析与处理

监控数据的分析与处理由数据管理模块单独负责。在本系统搭建的过程中，它的实现与数据持久化模块合二为一。在系统实现的过程中，我们使用

Prometheus 统一进行数据的持久化与监控数据的分析与处理。Prometheus 会将持久化的数据针对后续的可视化模块或微服务追踪模块进行简单的数据结构处理，以便适配其他模块，用以符合其他模块的数据接口。

5.5 可视化管理

可视化管理模块是本系统中极为重要的一环。它起到连接该系统用户和整个微服务监测系统的作用。在本系统的实现过程中，我们采用了开源项目 Grafana 作为可视化模块。Grafana 具有美观的 UI 界面，并同时具备绘制多种图表的功能。

可视化管理的功能需要和监控数据分析与处理模块共同作用才得以实现。首先，数据处理模块需要将收集到的来自服务网格的数据进行数据结构的处理，以便符合 Grafana 传入的借口要求；其次，Grafana 需要指定数据源，在本系统实现过程中我们将数据源指定为 Prometheus，同时配置各项数据源信息，如数据源的位置(基于 IP 地址和端口号进行定位)和相应用户名密码等信息；最后，我们需要在 Grafana 的 Dashboard 中绘制需要可视化的各项微服务指标，如微服务 Pod 所占用的内存、CPU 使用率等各项信息。通过 Grafana 自带的创建图表和查询语句，配合可视化操纵板，我们就可以实现微服务各项监控指标的可视化。

5.6 异常检测与报警

异常检测与报警是作为监测可视化系统中不可缺少的一环，也是自动化运维的关键。这一部分的功能我们在系统实现时与可视化管理模块 Grafana 进行了集成，即我们可以通过可视化的方式人工设定阈值来进行异常检测，并在检测值高于设定阈值时进行报警，以便立即通知运维人员进行第一时间的错误处理。

报警的方式有很多种，包括发送电子邮件和短信通知等最常用的手段。这些报警方式会在系统启动时，通过运维人员的手动设置从而生效。当系统检测到错误发生时，错误信息会根据运维人员的配置文件发送至不同的位置用以实现报警功能。

5.7 本章小结

本章主要描述了第四章中各项系统设计的实现方法。首先介绍了容器集群管理框架的搭建方法，并在之后采用数据驱动的顺序进行写作，按照被监测数据的“监控数据的收集、持久化、分析与处理、可视化管理、异常检测与报警”将各项系统核心功能的实现方法逐一详细介绍。

6.系统测试与验证

由于本微服务可视化监控系统的设计本身是基于云计算背景的，其实际运行情况和网络环境有很大的关系，在局域网环境下网络延迟不太明显，但是在广域网环境中，网络的不稳定性可能严重影响到本系统的各项性能指标。为了实际测试该系统的真实情况，本文将系统搭建并部署在了知名云服务供应商的虚拟云主机上，以求模拟最真实的使用场景，以测试系统的实际运行情况。

为了对本系统进行各项功能与性能的测试与验证，针对该系统笔者自主研发了若干套微服务应用。这些应用涵盖了微服务架构进行开发时所能遇到的几乎全部的基本问题，包括最重要的服务注册、服务发现和服务调用等各项功能，都通过这几套微服务应用得以实际地展现出来。本章的后续内容也会围绕这几套自主研发的微服务应用所展开，通过实际部署和使用微服务应用，来模拟微服务架构中的各种应用场景，从而说明本文提出的微服务架构能够合理解决目前微服务开发时所遇到的种种挑战。

6.1 微服务应用的管理

微服务应用往往是由多个微服务组件所构成的。而在目前云计算的大背景下，各个微服务组件为了满足快速启动与弹性伸缩等需求，通常都会将各个组件容器化，以镜像的形式存储在镜像仓库中，并在启动时按需拉取相应镜像。但是随着微服务应用的开发，其规模也变得逐渐庞大，组件也逐渐变多，每个组件的版本信息同样难以控制。

在该系统中，为了解决上述问题，我们采用了容器管理品平台 Kubernetes 所提供的容器管理接口，并配合服务网格 Istio，使用 yaml 文件对组成微服务应用的各式组件进行统一管理。以这种方式管理微服务应用，我们就可以把以往管理繁杂的微服务组件简化为管理控制微服务应用的 yaml 文件。这种管理目标的转移能够极大地减轻运维人员的工作量。

在这里，以笔者自主研发的微服务 “留言板 Web Demo” 为例，简要介绍微服务应用的部署。

管理并启动其中一个微服务组件的 yaml 文件关键部分如表 6-1 所示。

表 6-1 启动微服务相关容器的文件

```
---
apiVersion: networking.istio.io/v1alpha3
kind: Deployment
...
    spec:
      containers:
      - name: microweb-dbmanager
```

```
        image: xinyaotian/psqlmanager:0.0.2
        imagePullPolicy: IfNotPresent
...
---
```

利用这个 yaml 文件，我们能够将 “留言板 Web Demo” 中的数据资源组件启动。可以注意到其中的镜像是通过 Deployment 中 spec.containers.image 字段进行指定的。在本例中，我们指定了 xinyaotian/psqlmanager 为该服务组件使用的镜像，其版本号为 0.0.2 。镜像的拉取策略 imagePullPolicy 指定为 IfNotPresent，即 “若不存在则到指定的镜像仓库进行拉取” 的状态。

与此同时，读者也不难发现，通过 yaml 文件亦可以管理微服务组件的限制指标、环境变量等多种资源。

利用此种方式，我们将微服务组件的管理从具体组件简化成为各种文件。以这种管理方式减轻运维人员的具体工作量。当管理一个庞大的微服务应用时，运维人员可以不再专注于繁杂的组件，而把精力放在管理多个组件的 yaml 配置文件上即可。

```
root@ubuntu:~/microWebYaml-istio1.5# kubectl get deployment
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
coffeeshop-postgres             1/1     1            1           4d17h
coffeeshop-useropers            1/1     1            1           3d15h
microweb-dbmanager-deployment   1/1     1            1           10d
microweb-webui-v1               1/1     1            1           10d
microweb-webui-v2               1/1     1            1           10d
root@ubuntu:~/microWebYaml-istio1.5#
```

图 6-1 微服务应用的启动信息

6.2 出入集群的流量控制

微服务开发中的一个主要难点即出入微服务集群的流量控制。特别是在对于安全性需求较高的应用场景中，管理出入微服务应用的流量是至关重要的。在这个实验验证中，我们使用 Istio 和 Kubernetes 的流量准入和准出机制，配合自主研发的微服务案例和相应的配置文件，对微服务应用的准入和准出流量进行控制。

对进入集群流量进行控制的 yaml 文件关键部分如表 6-2 所示。

表 6-2 流量的准入集群控制

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
...
http:
  - match:
    - uri:
        exact: /
    - uri:
        exact: /all-queries
    - uri:
        prefix: /insert-post/
    - uri:
        prefix: /queries/
    route:
    - destination:
        host: microweb-dbmanager-svc
        port:
          number: 9090
...
---
```

通过利用本系统所提供的 Gateway 和 VirtualService 两种控制类型，我们可以将进入集群的流量严格地限制在一个端口上的指定 url 上(指定的 url 可以为正则表达式)。利用集群的准入控制，我们可以将多个微服务组件，甚至是完全不同的微服务应用的访问点路由到同一个端口。对于微服务的使用者而言，这些细节是完全隐藏的，他们会与平时使用 Web 应用完全相同，在不同的 url 之间进行跳转。但是在微服务应用的内部而言，这些 url 背后却是完全不同的微服务组件在进行支撑。

对流出集群的流量进行控制的 yaml 文件关键部分如表 6-3 所示。

表 6-3 流量的准入集群控制

```
---
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
...
spec:
  hosts:
  - 174.137.53.55
  ports:
  - number: 5432
```

```
    name: postsql-port
    protocol: TCP
  resolution: DNS
...
---
```

利用这种方式可以将流出集群的流量精确地指定到某个位置(通过 IP 地址与端口号进行定位)。对于一些特定的应用场景，比如微服务应用需要去某个数据库集群获取相应的数据信息，就可以使用这种方法严格地指定流出集群的流量，使其只能以指定的协议访问指定的服务，从而提高安全性。

图 6-2 流出集群的流量控制

6.3 故障注入

微服务应用中的故障注入尤为重要。错误配置的故障恢复策略(例如，跨服务调用的不兼容/限制性超时)可能导致应用程序中的关键服务持续不可用，从而破坏用户体验。

本系统的故障注入功能能在不杀死 Pod 的情况下，将特定协议的故障注入到网络中，在 TCP 层制造数据包的延迟或损坏。设计这种诸如方式的理由是，无论网络级别的故障如何，在应用层观察到的故障都是一样的，并且可以在应用层注入更有意义的故障(例如 HTTP 错误代码)，以检验和改善应用的弹性。

运维人员可以为符合特定条件的请求配置故障，还可以进一步限制遭受故障的请求的百分比。可以注入两种类型的故障：延迟和中断。延迟是计时故障，模拟网络延迟上升或上游服务超载的情况。中断是模拟上游服务的崩溃故障。中断通常以 HTTP 错误代码或 TCP 连接失败的形式表现。

在本系统中，微服务的所有 HTTP 流量都会通过 Envoy 自动重新路由。Envoy 在负载均衡池中的实例之间分发流量。除此之外，Envoy 还会定期检查池中每个实例的运行状况。

本系统同样遵循熔断器风格模式，根据健康检查 API 调用的失败率将实例分类为不健康和健康两种。换句话说，当给定实例的健康检查失败次数超过预定阈值时，将会被从负载均衡池中弹出。类似地，当通过的健康检查数超过预定阈值时，该实例将被添加回负载均衡池。

相应的错误注入 的 yaml 文件关键部分如表 6-4 所示。

表 6-4 故障注入

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percent: 10
        fixedDelay: 5s
...
---
```

6.4 灰度发布

灰度发布是微服务发布的一种常用手段，指在黑与白之间，能够平滑过渡的一种发布方式。A/B 测试就是一种灰度发布方式，让一部分用户继续用 A，一部分用户开始用 B，如果用户对 B 没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到 B 上面来。灰度发布可以保证整体系统的稳定，在初始灰度的时候就可以发现、调整问题，以保证其影响度，而我们平常所说的金丝雀部署也就是灰度发布的一种方式。

本系统同样支持这种微服务架构常用的发布模式，使用如下 yaml 即可控制进入集群的流量路由到不同服务版本的权重，从而实现灰度发布。

表 6-5 微服务的灰度发布

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
...
    route:
    - destination:
        host: microweb-webui
        subset: v1
        port:
          number: 80
      weight: 80
```

```
    - destination:
        host: microweb-webui
        subset: v2
        port:
          number: 80
      weight: 20
...
---
```

上面给出的 yaml 将流量权重指定给不同的两个版本 v1 和 v2。在这个例子中，我们给 v1 版本分配的权重为 0 ， v2 版本的权重为 100，即流量全部路由到 v2 版本上。通过修改该文件中 weight 字段的属性值，就可以实现诸如蓝绿测试、A/B 测试、灰度发布等多种微服务应用场景。

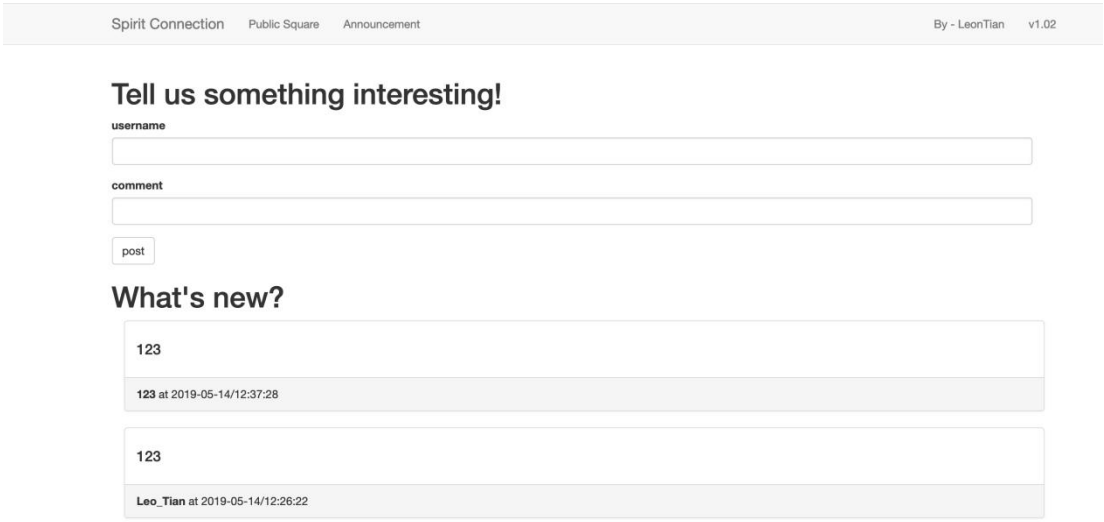


图 6-3 微服务的灰度发布

6.5 监测数据可视化

微服务应用的监测数据可视化主要由该系统中的 Grafana 组件实现。通过 Grafana 的原声语言指定监测指标并绘制图表，从而实现对微服务应用中各个组件各项资源的可视化监测。

图 6-4 使用 Grafana 监控微服务

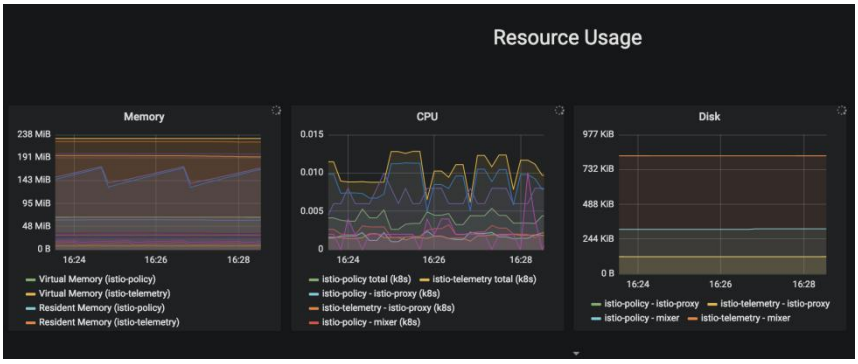


图 6-5 系统组件物理资源监测图

6.6 资源指标检测与报警

基于 Grafana 中绘制的监测资源图表，我们可以使用资源指标检测功能可视化地设定阈值，在超出阈值时实时报警。通过发送邮件的方式让运维人员第一时间掌握发生的错误信息。

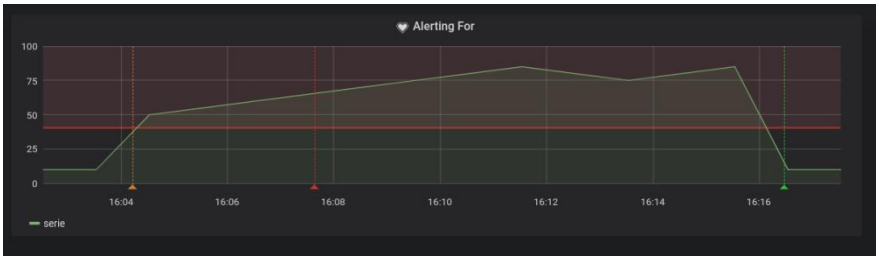


图 6-6 根据监测资源设定报警阈值

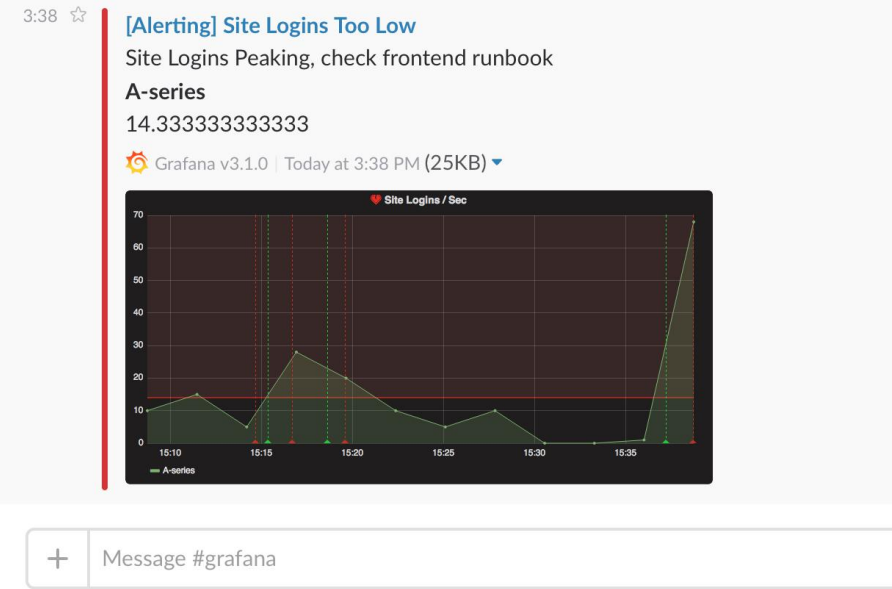


图 6-7 通过邮件接收报警信息

6.7 本章小结

本章作为本文的最后一个章节，模拟了 6 个在微服务开发中常见的应用场景，并使用真实代码与相应验证结果图片相配合的方式来测试与检验该系统的各项功能已经满足系统设计目标，达到相应要求。通过上述测试与验证，证明本系统确实改善了本文在第三章中提出的目前微服务开发所面临的诸多难题。

结束语

微服务架构的发展和成熟不仅带来传统软件开发模式的变革， 也会影响软件的开发、测试、部署和运行，进而影响整个软件行业的基础架构产生变革。在这样的环境下，利用微服务架构的方式去开发部署软件将是未来基于云计算环境下软件的主要形式。微服务架构本身体现了一切皆服务、模块化、指责专一化、高内聚、低耦合等众多软件开发的理基本念，且利用微服务架构进行软件开发，其设计理念更容易应用实际的生产当中；对用户来说基于微服务架构的各种软件也是软件使用上的新方向和趋势，能够让用户直接享受到基于云计算的软件能带来的众多好处。软件微服务化将是未来软件开发的主要应用手段，微服务架构将成为软件架构的主流开发架构之一。

北京工业大学毕业设计（论文）
参考文献

[1] 马威，韩臻，成阳 . 可信云计算中的多级管理机制研究 [J]. 信息 网络安全，2015，(7):20-25.

[2] WALRAVEN S, TRUYEN E, JOOSEN W. Comparing PaaS offerings in light of SaaS development[J]. Computing, 2014, 96(8): 669-724.

[3] 李贞昊.微服务架构的发展与影响分析[J].信息系统工程, 2017 (01):154-155.

[4] Microservices https://martinfowler.com/articles/microservices.html,2014

[5] Docker. https://www.docker.com/,2018.

[6] Kubernetes. https://kubernetes.io/,2018

[7] Envoy Documentation. https://www.envoyproxy.io/docs/envoy/latest/, 2018

[8] Service mesh for microservices.https://www.shantala.io/service-mesh-for-microservices/

[9] Istio. https://istio.io/docs/,2019

[10] Zipkin Documentation. https://zipkin.io/,2018

[11] Jaeger Documentation. http://jaeger.readthedocs.io/en/latest/,2018.

[12] ElasticSearch. http://www.elastic.co/products/elasticsearch,2018

[13]PrometheusDocumentation. https://prometheus.io/docs/introduction/overview/, 2019

[14] Grafana Documentation. https://grafana.com/docs/,2019

[15] Spring Cloud Documentation. https://spring.io/projects/spring-cloud/,2019

[16] 孙海洪.微服务架构和容器技术应用[J].金融电子化,2016(05):63-64

[17] 敖小剑. QCon 2017 年上海会议演讲实录.

https://servicemesh.gitbooks.io/awesome-servicemesh/mesh/2017/service-mesh-next-generation-of-microservice/, 2017

[18] Christian Posta.
Application Network Functions With ESBs, API Management,and now service mesh?.http://blog.christianposta.com/microservices/application-network-functions-with-esbs-api-management-and-now-service-mesh/,2017

[19] 覃璐. Medium 架构实践：避免微服务综合症.

https://www.infoq.cn/article/fv8tgq0VBeoToVT*TKy8/,2019

北京工业大学毕业设计（论文）
致谢

作为本科生的生活转眼就要接近尾声了，回首这四年的时光，我不仅学会了很多知识，还成熟了许多。我感谢这四年来老师们对我知识的传授、同学们对我友爱的表达，我感谢这段时间所有的经历对我的磨练，使我能自信而坚强地面对以后的工作和生活。

在这里，我必须要首先感谢我的毕业设计导师王焘老师，感谢他在学术与生活中给与我的指引和帮助。王老师为人和蔼可亲，学识渊博，学术风格严谨，在专业领域有很大成就，对我的毕业设计与论文非常负责：从开题、初稿形成到一遍一遍的修改与完善，王老师都给与了非常认真与细致的指导；同时，王老师还为我的毕业设计安排了以周为单位的进度规划表，在每周例会上听取我的阶段性汇报，并解答我遇到的问题、给予我细心的指导。我非常敬佩王老师严谨务实的研究精神与诲人不倦的优良美德。此外，在生活学习中难免有困惑的时候，听王老师讲话真的会有茅塞顿开之感，所谓“听君一席话，胜读十年书”，这就是智者向我传授的人生哲学，细细体会令我受益匪浅。

其次，我也同样要感谢我的校内指导老师李娟教授和中国科学院软件研究所的张文博老师，为我在校内以及校外提供的各种支持和帮助。

之后，我要感谢中国科学院软件研究所的师兄们：苏林刚师兄在我遇到问题时不分昼夜地与我讨论并给予我切实可行的解决方案；许源佳师兄为我提供的具有高屋建瓴般系统宏观架构的建议；薛晓东师兄在每次例会时为我提供的帮助。感谢师兄们的真诚付出，让我的这段时光中，在学业上更加顺利，在生活中总能感受到温暖。

最后，我还要向在百忙之中评阅论文和参加答辩的各位专家、教授致以真诚的谢意！