中图分类号：

论文编号：10357E20301216

安徽大学

硕士学位论文

云原生环境下微服务配置优化方法研究及框架实现

作者姓名　　贾成成

专业学位类别　　电子信息硕士

专业学位领域　　计算机技术

指导教师　　张以文

中图分类号：

论文编号：10357E20301216

专业硕士学位论文

云原生环境下微服务配置优化方法研究及框架实现

作者姓名　　贾成成

专业学位类别　　电子信息硕士

专业学位领域　　计算机技术

指导教师　　张以文

Research on Configuration Optimization Methods for Microservices in Cloud Native Environment and Framework Implementation

A Dissertation Submitted for the Degree of Master

Candidate：JIA Chengcheng

Supervisor：ZHANG Yiwen

中图分类号：

论文编号：10357E20301216

硕 士 学 位 论 文

云原生环境下微服务配置优化方法研究及框架实现

作者姓名	贾成成	申请学位级别	硕士
指导教师姓名	张以文	职　　称	教授
学科专业	电子信息 计算机技术	研究方向	微服务架构
学习时间自	2020 年 9 月 12 日	起至	2023 年 7 月 1 日止
论文提交日期	2023 年 5 月 23 日	论文答辩日期	2023 年 5 月 17 日

摘 要

随着云原生和微服务技术的发展，越来越多的企业选择在云环境中部署和运行应用程序。微服务是一种在云原生环境中被广泛采用的软件架构风格，通过将单个应用拆分成许多小的服务，使得每个服务运行在单独的进程中，并且可以独立部署，微服务之间借助网络进行交互。微服务架构应用程序的性能与微服务的资源配置以及微服务自身包含的软件参数设置紧密相关。然而，微服务应用中服务数量大，可配置的参数共同组成了高维的搜索空间，并且服务之间依赖复杂，参数之间也存在相互影响，因此如何对微服务进行配置优化以提高应用性能是一个重要且充满挑战的研究课题。

围绕微服务应用配置优化问题，目前已有的解决方案主要分为两类：第一类通过优化 CPU、内存等资源配置以提高微服务应用的性能。第二类通过优化微服务应用自身所包含的 Nginx、Redis、MongoDB 等软件参数以提高应用性能。然而这两种方案忽略了资源配置与软件参数之间存在的依赖关系，未对两者进行协同优化，也忽略了由此产生的高维配置搜索空间的挑战。针对已有工作的不足，本文对云原生环境下微服务应用的配置优化问题进行了研究，提出了资源配置与软件参数协同优化方案，并进行了框架的设计与实现。实验证明，该方案能够为微服务应用带来更大的性能提升，在工作负载吞吐量为500请求每秒时，本文提出的方案相比单独调整资源配置，性能提高了 22.94%，相比单独调整软件参数，性能提高了 17.95%。实验还显示在其它负载吞吐下，本文提出的方案相比单独调整的方案，也能带来更高的性能提升，且负载吞吐量越大提升效果越明显。此外为了简化微服务应用的配置优化流程，本文还实现了一种自动化的云原生微服务应用在线配置优化框架。本文的主要工作总结如下：

（1）针对单独调整微服务应用中资源配置或软件参数所带来的优化空间损失问题，本文提出了基于资源配置与软件参数协同优化的方案。该方案将微服务应用的配置优化视为黑盒优化问题，采用了配置优化领域广泛使用的贝叶斯优化算法进行解决。由协同优化带来的高维参数搜索空间问题，本文通过定位对微服务应用性能影响巨大的关键服务，以降低参数搜索空间。在本地集群上的实验证明，相较于单独调整资源配置或软件参数的方案，本文提出的协同优化方案带来了更大的性能提升。同时，为了优化资源能效，避免保守的资源分配策略为微服务提供了不必要的系统资源，本文在实验过程中优化了微服务应用的性能评价指标，提高了系统资源利用率。

（2）针对云原生环境微服务应用中含有大量的微服务与软件，难以人工手动进行

配置优化的问题，本文设计并实现了云原生微服务应用在线配置优化框架。该框架借助云原生技术栈和 DevOps 工具，避免了系统管理人员手动调整容器资源配置与软件参数、进行微服务应用部署和性能测试等重复繁琐的步骤，使微服务配置优化流程更加简洁高效。此外，该框架还提供了用户自定义微服务接口与算法接口，进一步增强了框架的可扩展性。

关键词：云原生，微服务，配置优化，贝叶斯优化

Abstract

With the development of cloud native and microservices technologies, more and more enterprises are choosing to deploy and run their applications in cloud environments. Microservices is a widely adopted software architecture style in cloud native environments, which splits a single application into many small services, each running in a separate process and able to be deployed independently, while interacting with each other through a network. The performance of microservices-based applications is closely related to the resource configuration of the microservices and the parameter settings of the software in the microservices application. However the large number of services in a microservices application results in a high-dimensional search space formed by the jointly configurable parameters., and there are complex dependencies between services and mutual influences between parameters. Therefore, how to optimize the configuration of microservices to improve application performance is an important and challenging research topic.

Existing solutions to microservices configuration optimization problems can be divided into two categories: the first optimizes the performance of microservices applications by optimizing resource configuration such as CPU and memory. The second optimizes the software parameters included in microservices applications, such as Nginx, Redis, and MongoDB, to improve application performance. However, these two types of solutions ignore the dependencies between resource configuration and software parameters, do not perform coordinated optimization, and ignore the challenges posed by the high-dimensional configuration search space. To address the shortcomings of existing work, this thesis investigates the configuration optimization problem of microservices applications in cloud native environments and proposes a coordinated optimization solution for resource configuration and software parameters, along with the design and implementation of a framework. Experiments show that the proposed solution can bring greater performance improvements to microservices applications. When the workload throughput is 500 requests per second, the proposed solution can improve performance by 22.94% compared to optimizing resource configuration alone and by 17.95% compared to tuning software parameters alone. Experiments also show that the proposed solution can bring higher performance improvements than the separately optimized solution under other workload throughputs, with the improvement effect becoming more obvious as the workload throughput increases. In addition, to simplify the configuration optimization process of microservices applications, an automated cloud native microservices application online configuration tuning framework is also implemented in this

thesis. The main contributions of this thesis are summarized as follows:

(1) To address the problem of loss of optimization space resulting from tuning resource configurations or software parameters individually, this thesis proposes a coordinated optimization scheme for resource configuration and software parameters in microservices applications. This scheme treats microservices application configuration optimization as a black-box optimization problem and employs Bayesian optimization algorithms widely used in the configuration optimization field to solve it. Due to the high-dimensional parameter search space resulting from coordinated optimization, this thesis reduces the parameter search space by identifying key services that have a significant impact on microservices application performance. The local cluster experiments have shown that compared to schemes that optimize resource configurations or software parameters alone, the proposed coordinated optimization scheme brings greater performance improvements. In addition, to optimize resource efficiency and avoid unnecessary system resource allocation that may result from conservative resource allocation strategies, this thesis optimizes the performance evaluation metrics of microservices applications during experiments to increase system resource utilization.

(2) To address the problem of manual configuration optimization for cloud native microservices applications containing a large number of microservices and software parameters, this thesis designs and implements an online configuration optimization framework for cloud native microservices applications. This framework leverages cloud native technology stacks and DevOps tools to avoid manual tuning of container resource configurations and software parameters by system administrators, and repetitive and tedious steps such as microservices application deployment and performance testing, making the microservices configuration optimization process more concise and efficient. In addition, the framework provides user-defined microservice interfaces and algorithm interfaces to further enhance its scalability.

Keywords: Cloud Native, Microservices, Configuration Optimization, Bayesian Optimization

目　录

图表清单

图目录：

表目录：

Figures and Tables Directory

Figures Directory:

Tables Directory:

第一章 绪论

1.1 研究背景与意义

云原生[1]是近几年云计算行业内十分火热的领域。云原生这个术语已经存在多年，随着谷歌、CoreOS、红帽、华为等公司在 2015 年联合创立云原生计算基金会(Cloud Native Computing Foundation，CNCF)，云原生开始从理念转变为开源实现，其代表技术包括容器化、微服务、自动化部署、弹性伸缩等，这些技术旨在优化云环境中的应用程序开发、部署和管理流程。云原生技术在动态基础设施环境中的应用，如私有云、公有云和混合云等环境，能够帮助组织或企业构建和运行具备弹性扩展能力的应用程序。由于云原生能够适应新兴领域，如人工智能、大数据、边缘计算[2]和 5G[3]的需求，因此云原生正逐渐成为这些领域中的重要技术。2021 年 CNCF 年度调研报告中显示已有 96%的受访组织或企业正在使用或评估云原生相关技术栈[4]。

微服务架构是云原生环境下流行的软件架构风格，目前已经在工业界实际生产环境中被广泛采用，但微服务架构也给应用稳定高效运行带来了新的挑战。微服务应用中所面临的请求是动态变化的，为了保证微服务应用 7*24 小时稳定高效地运行，开发者通常会为处于微服务应用关键路径上的服务预留多余的硬件资源，以满足峰值负载下的性能约束，导致整体系统资源利用率不高[5]，因此微服务应用的高效运行必须权衡应用性能与系统资源使用情况。事实上微服务应用为开发者提供了大量的可配置的参数，主要包括容器的资源配置和作为服务部署的软件参数。具体来说：

（1）容器的资源配置主要包括容器的规格（CPU、内存、网络带宽等系统资源）和容器的副本数[6]。系统资源并不是越多越好，在微服务应用中要根据每个服务自身的业务特点对容器的资源配置做出合理的选择，选择不当则会造成应用端到端性能无法满足用户要求，或者产生不必要的成本开销。

（2）作为服务部署的软件参数，例如 Redis、MongoDB、Memcached、Nginx 等软件自身存在大量可调节参数，当这些软件作为微服务应用中一个服务提供时，可以通过调节这些软件的参数优化微服务应用性能。

微服务架构提高了应用的开发效率，通过调节微服务应用的资源配置和软件参数可以优化微服务应用的性能。不同于传统软件，微服务架构具有组件化、依赖复杂等特点，这使得微服务应用的配置优化工作，难以直接使用针对传统软件提出的优化方

法[7,8]，因此需要对微服务架构下的配置优化工作进行专门的研究。同时微服务应用中服务之间请求动态变化，依赖复杂，部署测试流程繁琐[9]，设计并实现云原生微服务应用在线配置优化框架，以简化配置优化流程也具有重要意义。

1.2 国内外研究现状

配置优化方面已有的相关工作主要集中在解决传统软件的性能问题，如 Hadoop[10]、Spark[11]、Flink[12]等大数据框架提供了大量参数用以调节框架运行时性能。由于云原生环境下微服务应用具有组件化、依赖复杂等特点，这使得传统软件上相关工作很难被直接迁移到微服务应用中。虽然最近有部分工作开始关注于微服务应用的配置优化问题，但他们主要对微服务应用的资源配置或软件参数进行了独立的考虑，未能对两类配置参数进行协同优化，造成了性能优化空间的损失。

1.2.1 传统软件配置优化现状

经过多年的发展，大数据已从一个新兴技术领域逐渐转变为社会经济发展中各个领域的重要因素、资源、动力与理念。针对大数据生态中的各个组件，如 Hadoop、Spark、Flink 等，研究人员提出了一系列的方法来优化其性能。同时随着数据量的增加对数据库软件的性能优化也成为了一个重要的研究方向。本节分别对这两方面的相关工作进行介绍。

Mathiya[13]等通过实验证明，相较于 Hadoop 默认的参数配置，良好的自定义参数能够提高集群资源利用率，进而提升系统的性能，证明了对大数据软件进行配置优化具有重要意义。Herdotu 等[14]提出了 Starfish，通过建立精确的成本模型来模拟 MapReduce 的执行过程，并使用一个小型模拟器组件模拟任务调度决策，可以让用户在不熟悉 Hadoop 的内在原理的情况下，也能够对该系统进行优化。该作者还提出了 Elastisizer[15]将作业层面的软件参数与资源配置组合在一起，扩展了 Starfish。通过在小型集群和小型数据集上多次执行 MapReduce 作业，建立了回归树模型对运行成本开销进行预测。并且 Elastisizer 使用递归随机搜索来同时确定最佳集群大小和作业参数配置，以便在请求时间或成本预算内执行完 MapReduce 作业。Liao 等[16]提出了一个基于搜索的 Hadoop 调优系统 Gunther，它将配置优化视为一个黑箱优化问题，使用遗传算法搜索参数配置，可以在不到 30 次的迭代下找到接近最优的参数配置。此外 Gunther 忽略了对性能影响不大的参数以减少搜索时间。

上述工作集中于 Hadoop 框架，除此之外，研究人员还对大数据生态下的其它软件配置优化问题进行了深入研究。Petridis 等[17]提出了一种基于少量实验来调整 Spark 框架参数的试错方法。具体来说，基于文档和过去的执行经验，作者首先选择了 12 个重要的参数，然后在不同的基准应用下测试这些参数对性能的影响。基于测试结果，作者以方块图的形式开发了一种方法，其中包含 7 个对性能最具影响力的参数，然后通过从方块图中获取不同的参数配置，执行 Spark 负载进行配置优化。Yu 等[18]为内存数据处理框架（如 Spark）引入了一种参数自动调整方法，作者考虑到了数据量大小和高维参数方面的问题，他们首先提出了一个分层预测模型，该模型由许多分层排列的子模型组成，中心思想是建立几个较简单的模型，而不是单一的复杂模型。接下来，作者采用遗传算法（Genetic Algorithm，GA）来寻找最佳参数配置。Li 等[19]在其研究中提出使用生成对抗网络来优化 Spark 的配置参数，它可以通过使用较少的训练数据来构建性能预测模型，同时不会降低模型的准确性。除此之外，还使用了优化的遗传算法来搜索参数空间，以获得最佳的配置方案，在五个典型的工作负载上，搜索到的参数配置相比默认配置，Spark 性能均得到了提高。此外，Fekry 等[20]提出了 Tuneful 配置优化框架，可对数据处理框架 Spark 进行配置优化。Tuneful 对敏感度分析（Sensitivity Analysis，SA）方法进行了改进，采用多轮 SA 逐步选择关键参数，以提高优化效率。针对 Flink 框架，Guo 等[21]提出了一种引导式机器学习的方法来自动调整 Flink 的配置参数，作者先使用生成对抗网络生成一些样本数据为 Flink 的配置与性能之间建立模型，然后用引导式机器学习算法为 Flink 集群寻找最优配置，与其它基于机器学习的大数据软件自动配置优化方法相比，引导式机器学习的方法可以大大减少训练数据收集和最佳配置搜索所需的时间。Q-Flink[22]是一种适用于混布工作负载下的 Flink 共享资源控制策略，它监测共享资源在混布工作负载中相互竞争的情况，考虑不同工作负载对服务质量(Quality of Service，QoS)的不同要求，然后对共享资源进行分配，以减少作业服务质量的违规率。

随着大数据、云计算、互联网和移动互联网的发展，数据量呈指数级增长。在研究人员对大数据生态软件配置优化问题研究的同时，传统的关系型数据库也难以满足当今海量数据的存储和处理需求。MongoDB、Cassandra 等新型数据库在这个背景下逐渐变得流行起来。它们有着高性能和高可扩展性的特点，能够更好地处理大规模和高复杂度的数据。同时云计算的发展也使得数据库可以更加灵活地部署和扩展，更好地支持数据的分布式存储和处理。正确选择数据库软件的配置参数对提高性能和降低成

本至关重要，因此数据库软件的配置优化问题也得到了研究人员的广泛关注。Van 等[23]等设计了一个自动调优工具 OtterTune，用于优化数据库管理系统的配置参数。OtterTune 利用历史经验，采用监督和非监督机器学习方法的组合，实现了选择关键参数、建立工作负载映射和生成推荐参数设置的目标。它采用 Lasso 算法选择关键参数，但每次调用需要大量的时间和内存开销，以处理庞大的历史数据。该工具在 MySQL、Postgres 和 Actian Vector 三个数据库管理系统上进行了实验，虽然每个试验的测量时间很短，只有 5 分钟，但是仅仅为了收集初始数据 OtterTune 就花费了三个多月的时间。CGPTuner[24]使用贝叶斯优化对数据库管理系统进行调优，该系统并不需要收集大量初始数据进行引导，并且考虑了 Java 虚拟机、操作系统、物理机等层次的配置参数。在 Cassandra 和 MongoDB 上已经得到了很好的应用。Kanellis 等[25]针对数据库系统，使用分类回归树算法（Classification And Regression Tree，CART）选择关键参数，并且在构建随机森林的过程中可以捕获参数之间非线性的关系。作者通过实验证明只有少数几个关键参数（约 5 个）会对系统性能产生较大的影响，并且在不同负载下，关键参数是基本保持一致的，然而关键参数保持一致这个结论在微服务场景下是不成立的。

上述相关研究局限于大数据软件和数据库等传统软件的配置优化问题，然而现今软件架构已经改变，微服务架构为越来越多公司接受和采用，尽管基于传统软件的配置优化方面的研究具有重要的价值，但是它们只考虑单个软件栈的参数调整，并未考虑多个软件、不同软件的参数相互影响等存在于微服务应用中的特点，因此传统软件上的配置优化工作难以直接应用到微服务中，需要进一步扩展配置优化相关工作研究的范围，以全面评估和提高微服务应用的性能。

1.2.2 微服务配置优化现状

由于微服务应用包含多个服务、服务之间依赖复杂且参数之间存在相互影响，因此传统软件上的相关工作很难直接应用到微服务中。目前研究人员开始针对微服务场景下的配置优化问题进行研究，该场景下已有的工作主要分为两类方案：第一类通过优化 CPU、内存等资源配置以提高微服务应用性能；第二类通过优化微服务应用所部属的 Nginx、Redis 以及 MongoDB 等软件自身配置参数以提高微服务应用性能。下面将分别介绍这两类方案。

在资源配置优化方面，Reiss 等[26]通过研究谷歌公布的集群使用信息数据集[27]，指出集群中资源类型和使用方式的异质性，这种异质性会降低传统资源调度技术的有效

性。Reiss 还指出集群中工作负载是高度动态的，由许多短期工作和少量具有稳定资源利用率的长期运行工作组成。总的来说，作者通过对谷歌公布的数据集进行研究，证明了在云计算环境下对资源配置进行优化的必要性。在此基础上 Jyothi 等[28]为了解决云计算服务端追求高资源利用率和客户端追求应用高性能两者之间的矛盾，先对用户服务等级目标（Service-level Objective，SLO）进行了合理的定义，然后分析客户端作业之间的依赖关系，并利用历史信息进行建模，使用周期性预定（Recurring Reservation）的思想对服务端资源配置进行管理，在提高系统资源利用率的情况下降低了 SLO 违规的次数。Gias 等[29]设计了一个微服务应用系统资源自动扩缩控制器，对服务副本数和 CPU 资源用量进行调整，并且同时考虑了水平扩缩[30]和垂直扩缩[31]对微服务应用性能的影响。该控制器使用分层排队网络（Layered Queueing Network，LQN）模型对微服务应用性能、服务副本数、CPU 资源用量三者进行建模。最后使用遗传算法对最优参数配置进行搜索。FIRM[32]也对微服务应用的资源配置进行了研究，它是一个针对微服务应用资源管理框架，可以用于缓解微服务之间由于资源竞争导致的 SLO 违规，框架使用支持向量机（Support Vector Machine，SVM）检测服务性能异常，异常发生时，通过添加更多的 Pod[33]或 CPU、内存等系统资源，对相应的服务进行扩容。FIRM 的缺陷主要有两点，第一只有在性能异常发生后才会进行系统资源自动扩缩，并且大规模的自动缩放可能需要几分钟的冷启动，或者至少几秒钟的热启动，因此需要较长的响应时间，有可能使服务异常持续一段较长的时间。第二由于 FIRM 通过系统异常进行触发，当分配给微服务的资源超过实际所需用量时，它无法对系统资源进行回收，造成资源浪费。对于资源无法自动缩放的问题，Autopilot[34]基于滑动窗口算法设计的资源配置推荐器可以在服务转向平稳运行或者负载峰值消退后，回收之前过多分配置的系统资源，一定程度上解决了 FIRM 无法对资源进行回收的问题，提高了资源利用率。Zhang 等[35]考虑到微服务应用中服务分层的特点，提出微服务应用资源配置管理模型 Sinan，它可以在线运行，根据服务的运行状态和端到端的 QoS 目标，动态地调整各层服务的系统资源用量。Sinan 首先使用空间探索算法来检查可能的资源配置搜索空间，通过收集的数据训练两个模型：卷积神经网络（Convolutional Neural Network，CNN）模型，用于微服务应用短期性能预测；提升树（Boosting Tree）模型，用于评估微服务应用长期性能演变。两个模型的结合使 Sinan 既能检查资源配置的近期效果，又能考虑到系统中工作负载的相似性，优化精度比单一模型更加准确。

上述研究主要集中在微服务应用 CPU、内存等资源配置的优化方面，另外微服务

应用中所部署的软件也存在大量可供调节的参数，通过对软件参数调整也可以优化微服务应用的性能，目前通过调整微服务部署的软件参数对应用进行配置优化的相关研究较少，下面对该方案已有的研究进行介绍。Mahgoub 等[36]提出了 OPTIMUSCLOUD 系统针对云环境下的数据库软件进行配置优化，通过建模的方式对最优配置进行搜索。Wang 等[37]扩展了开源调优工具 KeenTune[38]，对企业级云原生应用软件和服务进行自动化配置优化，在 MySQL、OceanBase、Nginx、Ingress-Nginx 等软件和服务上取得了不错的效果。然而这些工作局限于调整单个微服务应用中存在的软件，未对微服务全链路进行考虑。Somashekar 等[39,40]对微服务应用的配置优化工作进行了相对更加全面的考虑，通过协同调优作为服务部署的 Nginx、Memecached、MongoDB 等软件的配置参数来优化微服务应用性能。以上通过调整微服务应用软件参数，进行应用性能优化的工作局限性在于：未将软件参数与微服务资源配置进行协同考虑，而在峰值负载时调整资源配置能够为微服务应用带来更大的性能提升。

综上所述，目前国内外配置优化问题的相关工作，主要关注解决传统软件性能问题，难以直接适用于微服务应用。目前已有的少量关于微服务应用配置优化的工作，分开考虑了资源配置和软件参数，没有将两者协同考虑，导致了微服务应用性能优化空间的损失。

1.3 本文主要研究内容及章节安排

本文的研究重点是云原生环境下微服务应用的配置优化方法及框架实现。由于传统软件的配置优化方法不适用于微服务应用，以及现有微服务应用的配置优化方法忽略了资源配置与软件参数之间的相互依赖关系，损失了部分性能优化空间。另外微服务应用组件众多，依赖关系复杂，手动调优非常繁琐。因此，本文的主要工作包括：

（1）针对微服务应用配置优化问题中单独调整资源配置或软件参数的方案，会损失部分性能优化空间的缺陷，本文提出了微服务应用资源配置与软件参数协同优化的方案。在本地集群上的实验证明，相较于单独调整的方案，本文提出的方案在不同的工作负载下均能为微服务应用带来更大的性能提升，并且负载吞吐量越大提升效果越明显。另外，为了避免保守的资源分配策略为微服务分配了过多的系统资源，本文在实验过程中优化了微服务应用的性能评价指标，并通过实验证明，该指标能够提高系统资源利用率。

（2）针对云原生环境微服务应用组件众多、依赖复杂、难以人工手动进行配置优

化的问题，本文设计并实现了云原生微服务应用在线配置优化框架。该框架借助容器编排以及 DevOps 等云原生技术栈，避免了系统管理人员手动对微服务应用进行配置变更、应用部署、性能测试等重复繁琐的步骤，使配置优化流程更加简洁高效。为了增加该框架的适用性和可扩展性，框架给用户提供了微服务应用接口与算法接口，可以让用户根据自己的场景进行定制化实现。

聚焦于云原生环境微服务应用的配置优化问题，本文整体的内容安排如下：

第一章为绪论，对本文所研究的配置优化工作相关背景进行了介绍，同时解释了微服务场景下配置优化工作的意义，然后总结了相关国内外研究现状，最后阐明了本文的主要研究内容、工作以及具体每项工作的意义。

第二章为相关技术基础，详细阐述了本文研究过程中使用到的云原生相关技术理论，包括容器、容器编排、微服务架构以及 DevOps 技术。此外，本章还对这些技术在本文研究中的作用进行了详细的解释。

第三章为云原生微服务应用的资源配置与软件参数协同优化，提出并详细描述了针对云原生环境微服务应用的配置优化方法，以及该方法所面临的挑战，然后针对每一个挑战提出了相应的解决方案。最后介绍了实验相关的设置并进行了全面的实验，结合实验结果对本章提出的方法进行了充分的分析。

第四章为云原生微服务应用在线配置优化框架设计与实现，该章首先对配置优化框架进行了整体性的阐述，然后对框架进行了详细的需求分析，并从多个角度分析框架设计。最后在需求分析与框架设计的基础上实现了配置优化框架中的各个功能模块，并进行了全面的测试，确保框架能够稳定高效运行。

第五章为总结与展望，该章对本文进行了总结，同时以本文为基础对未来可继续深入研究的问题进行了展望。

第二章 相关技术基础

云原生是一种基于容器、微服务、云计算等技术的软件开发技术栈，旨在简化应用的开发、部署和运维流程，使应用能够高效运作。云原生环境中应用通常是由多个互相依赖的小型服务组成，各个服务可以独立部署、运行与维护。因此，软件开发迭代流程更加敏捷，并且相对于传统巨石架构应用（Monolithic Architecture），每个服务需要更少的计算资源，拥有更高的可扩展性。在云原生的软件开发方式下，开发人员使用云原生技术栈自动化应用的构建、测试和部署流程，并且可以快速地扩缩微服务应用，使应用在不同平台上具有较高的可扩展性，这是传统的巨石架构应用无法实现的。现今企业需要创建可扩展、灵活以及弹性的应用，以便快速地响应用户需求，因此他们选择在云基础设施上利用云原生等现代化工具和技术进行应用开发。云原生技术栈主要为采用者带来了以下方面的竞争优势：

（1）效率提高：通过利用云原生技术栈中的 DevOps、持续集成、持续交付等敏捷开发实践，开发人员可以基于自动化工具和云基础设施快速构建应用。

（2）成本降低：采用云原生技术栈可以节省购买和维护物理基础设施的成本，减少长期运营支出。此外，节省的成本也能够令应用的最终用户受益。

（3）可用性保证：云原生技术栈使公司能够构建弹性且高可用的应用，应用不会因功能更新而停机，并且可以在负载峰值时为服务扩展系统资源，增加了应用可用性。

云原生的发展过程中，CNCF 于 2015 年 7 月 21 日成立，是推广和普及云原生应用的重要力量。CNCF 作为 Linux 基金会最大的子基金会之一，主要的工作重点在于推广开源社区中具有重大潜力与价值的云原生相关技术，如 Kubernetes[41]、OpenTelemetry[42]、gRPC[43]等，使用这些技术可以使云原生项目的开发更加完善与可靠。随着云原生技术的普及和成熟，基于云平台的软件也越来越多，如图 2.1 所示这些软件包括了构建云原生应用的各个方面，CNCF 对它们进行了归纳和整理，并按层次划分，主要包含以下五个层次：

基础设施层：作为云原生技术栈的基础，由操作系统、存储、网络和其他第三方云提供商管理的计算资源组成。

运行时层：为容器的运行提供资源与技术支撑，包括数据存储、网络管理和容器运行时（如 Docker、Containerd 等）。

编排和管理层：负责整合各种云组件，令它们作为一个整体单元运行。通常开发人

员使用 Kubernetes 等编排工具来部署、管理和扩展云原生应用。

应用定义和开发层：由用于构建云原生应用的软件技术组成，例如数据库、RPC 框架、容器以及持续集成持续交付等技术。

可观察性和分析工具层：用于监控、评估和改进云原生应用的运行状况，例如运维人员通过监控 CPU 使用率、内存和延迟等指标，确保应用的服务质量在规定的范围内。

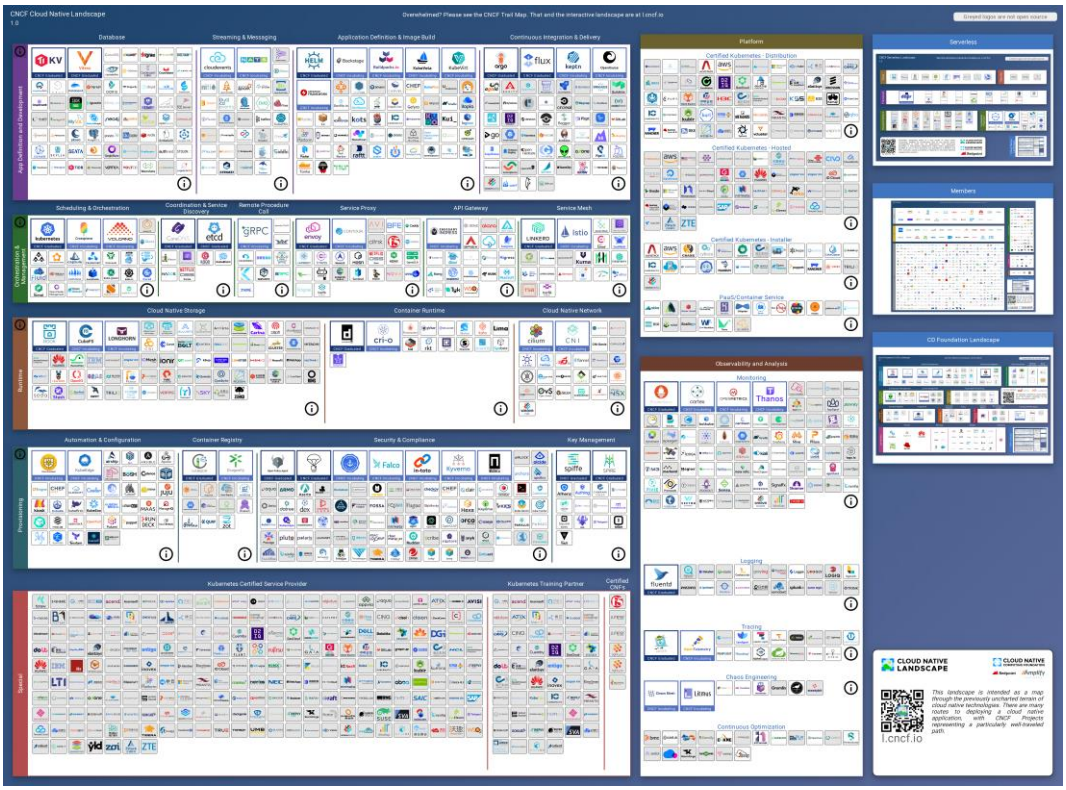

图 2.1 云原生全景图[①]

Figure 2.1 Cloud Native Landscape

2.1 容器与容器编排

2.1.1 Docker 相关技术

在云原生环境中，随着应用开发周期缩短，容器技术成了开发过程中难以缺失的一环。相对于虚拟机来说，容器具有更大的灵活性，提供接近系统原生的性能。在所有容器解决方案中，Docker 在市场上占据着领先地位，它提供了完整的软件打包和交付功能，使应用可以在不同的平台和系统上一致地运行。

Docker 容器技术的核心部分主要集中在三个方面：Cgroups[44]提供了限制容器系统

[①] 图片来源：https://landscape.cncf.io/images/landscape.pdf

资源用量的技术，以便管理系统资源的分配情况；Namespace[45]提供了一种内核级别的隔离环境，确保容器内的进程不会对外界造成影响；Rootfs[46]是构建容器镜像所使用的核心技术，允许用户在容器内使用模拟的文件系统，以满足其各种不同的需求。

本文在对微服务应用进行配置优化时，主要通过 Cgroups 技术限制微服务的系统资源用量。Cgroups，即 Linux Control Group，是 Linux 内核中的一项用于限制进程资源使用的技术，并且可以在多用户环境下进行细粒度的资源控制，是容器中的关键技术之一。Cgroups 将每个可控资源定义为一个子系统[47]，并以文件系统的形式对外提供操作接口，用户可以通过修改文件内容来调整进程在一段时间内所能够使用的系统资源总量。本文主要通过 cpu 和 memory 两个子系统来限制微服务应用中容器的系统资源用量。Cgroups 中的各个子系统以及它们的相应功能如表 2.1 所示。

表 2.1 Cgroups 可控子系统

Table 2.1 Cgroups Controllable Subsystem

子系统	功能
cpu	限制进程的 CPU 资源用量
cpuacct	统计进程的 CPU 使用情况
cpuset	为进程分配单独的 CPU 节点或者内存节点
memory	限制进程的内存使用量
blkio	限制进程的块设备 I/O 的使用量
devices	控制进程能够访问的设备列表
net_cls	标记进程的网络数据包，使用流量控制模块对数据包进行控制
freezer	挂起或者恢复进程
ns	控制不同进程使用不同的 Namespace

2.1.2 容器编排相关技术

容器编排是一种自动化的管理工具，它负责自动执行管理容器化工作负载和服务所需的大部分操作，从而使开发人员可以专注于开发和构建应用。容器编排工具能够解决容器生命周期中的各种问题，如配置、部署、弹性扩展、网络通信和负载平衡等。通过容器编排，可以控制容器服务之间的交互，以确保服务顺利运行。容器具有轻量化和生命周期短暂的特点，在生产环境中使用容器时需要大量的支持工作，特别是当容器技术与微服务架构结合时，大型微服务应用可能包含数百甚至上千个容器，庞大的容器数量为应用的管理工作带来了巨大的复杂性，人工手动管理在生产环境中是不

现实的。因此，需要使用容器编排工具来协助完成这种复杂的操作，容器编排工具可以通过声明式的方式自动完成应用管理的大部分工作，从而降低应用管理难度。容器编排工具为容器化的环境带来了以下好处：

操作简化：这是容器编排带来的最重要好处，也是采用容器编排工具的主要原因。容器技术为微服务应用管理引入了额外的复杂性，如果不使用容器编排进行自动化管理，应用将很快失去控制。

弹性：容器编排工具可以自动重启、扩展容器和集群，提高应用在基础设施中断或故障期间生存并保持在线的能力，令应用更加具有弹性。

安全性：容器编排的自动化操作可以减少或避免人为产生的错误，有助于保持容器化应用的安全性。

随着容器技术的快速发展，市场上出现了许多容器编排工具，如 Apache Mesos、Docker Swarm 和 Kubernetes 等。在这些工具中，Docker Swarm 和 Kubernetes 是目前最为普及并且广泛使用的[48]，其中，Kubernetes 已经成为了容器编排领域的行业标准。

Docker Swarm

Docker Swarm 是由 Docker 公司推出的容器编排工具，它在 2014 年 12 月的 DockerCon 大会上首次发布。Docker Swarm 的特色在于它完全利用 Docker 容器原本的 API 来实现集群管理，是 Docker 容器原生的编排工具，具有卓越的易用性。

在 Swarm 集群中，节点表示一个加入 Swarm 集群的 Docker Engine 实例，它是一个轻量级的容器运行时环境，可以在单个主机上运行多个隔离的容器。如图 2.2 的 Docker Swarm 架构所示，节点可以分为管理节点（Manager Node）和工作节点（Worker Node）两种类型。管理节点负责 Swarm 集群的管理和调度。一个 Swarm 集群至少需要一个管理节点，也可以将多个管理节点组成集群，以实现高可用性和负载均衡。在 Swarm 集群中，管理节点通常用于：创建和维护集群的状态；分配任务（Task）到工作节点上；安排容器的部署和更新；监控集群状态并处理运行时故障。工作节点是集群中运行容器的主机，一个 Swarm 集群可以拥有多个工作节点。工作节点不会处理集群状态的管理和调度相关任务，它们只负责运行容器及管理容器的生命周期，如运行容器实例、处理容器的日志和事件以及存储和检索容器镜像等。需要注意的是，Docker Swarm 集群中的管理节点可进行配置，令其作为额外的工作节点运行。

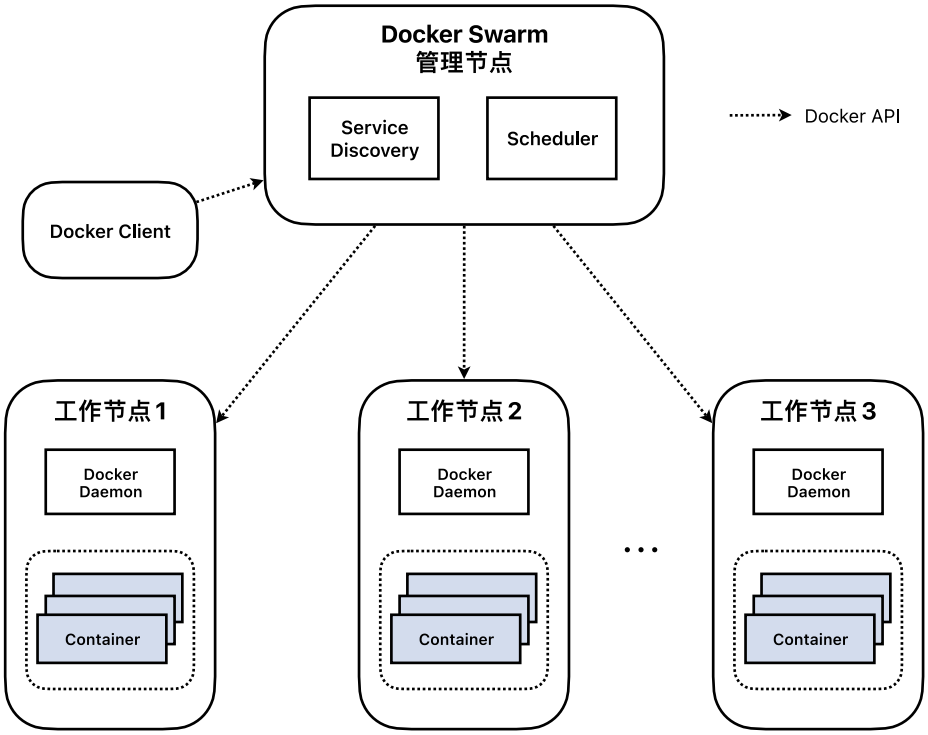

图 2.2 Docker Swarm 架构图

Figure 2.2 Docker Swarm Architecture

Kubernetes

Kubernetes 是由谷歌公司开发的强大且功能全面的容器编排工具，该工具于 2014 年被开源发布，现在已经成为容器编排领域的事实标准。Kubernetes 的诞生，源于谷歌公司 15 年生产环境的运维管理经验以及社区中最佳实践的总结，是谷歌内部集群管理系统 Borg[49]的继承者。Kubernetes 通过简化应用的部署和管理，并提供自动化的容器编排功能，从而提高应用的可靠性，并能够节约用户大量时间和资源，是目前最先进的容器编排解决方案。

与 Docker Swarm 类似，Kubernetes 集群可以被划分为两个部分：控制面和计算设备（或者称为数据面），在集群中，计算机被称为节点，可以是物理机也可以是虚拟机。其中少量的节点用作控制面即 Master Node，它们负责维护集群的预期状态，并接受来自集群管理员的命令，然后决定如何分配系统资源和容器来执行任务。其他的节点则被规划为计算设备即 Worker Node，它们负责根据控制面的管理运行应用和工作负载。控制面的节点在集群中扮演着重要的角色，而计算设备的节点则承担着运行应用和负载的主要任务。此外，可以通过设置令控制面也运行应用和工作负载。

Kubernetes 集群的控制面与计算设备各自包含多个核心组件，组件之间协同工作，共同完成了整个集群的管理和计算，并确保集群的高效运行。如图 2.3 所示，控制面有

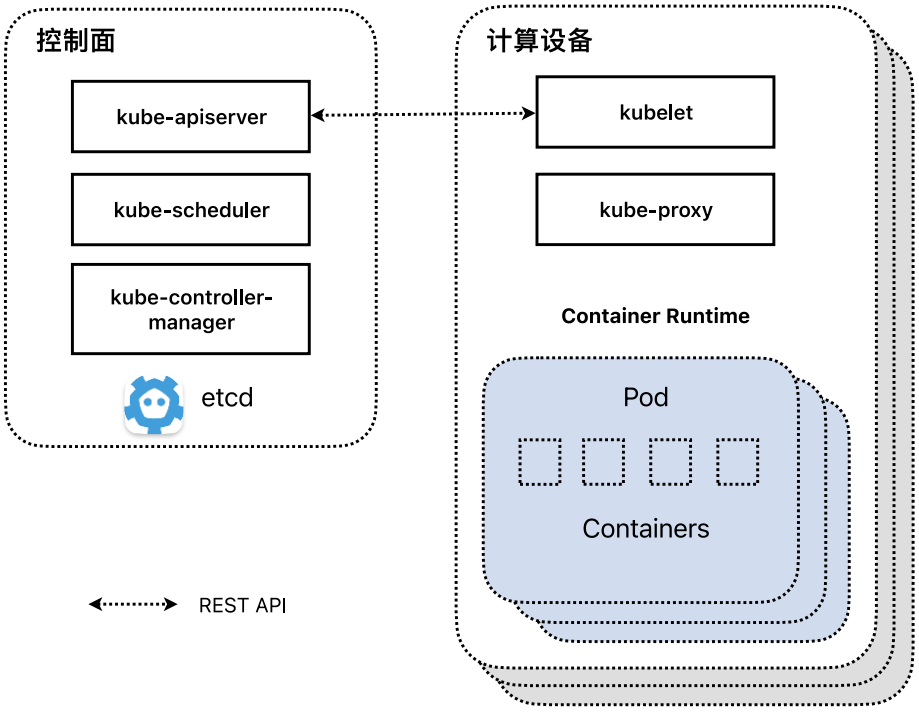

图 2.3 Kubernetes 架构图

Figure 2.3 Kubernetes Architecture

四个核心组件，他们的功能分别是：

etcd：负责保存集群的全局状态信息。

kube-apiserver：公开 Kubernetes API，通过公开的 API 可以对 Kubernetes 资源对象进行操作。

kube-scheduler：负责使用调度算法，结合定义的策略为新创建的 Pod 提供高效可靠的调度服务，选择最合适的节点进行部署。

kube-controller-manager：负责维护集群的状态，包括节点状态、服务状态、任务状态等。

计算设备中存在三个核心组件，这些组件也各自负责不同的功能：

kubelet：负责管理和监控节点上的容器。它会从 kube-apiserver 中获取 Pod 的清单，并确保节点上的所有 Pod 和容器都处于清单中所定义的状态。

container runtime：管理镜像并实际运行容器。它是容器生命周期管理的核心。

kube-proxy：为 Kubernetes 集群中的服务提供服务发现和负载均衡等功能，并实现多种网络代理模式以满足不同需求，同时提供网络隔离和安全性，确保服务之间的通信安全可靠。

Kubernetes 和 Docker Swarm 是用于容器编排的两种不同解决方案，它们适用于不同的场景。对于初学者来说，Docker Swarm 是一种更加简单易用的解决方案，提供了简洁的界面和简单的管理功能，特别是当您需要管理的容器规模比较小时，那么 Docker Swarm 可能是一个更加合适的选择。相比之下，Kubernetes 则更加适用于处理复杂的工作负载，具有更为完整的生态体系，包括监控、安全、高可用性、自我修复等功能。Kubernetes 是编排大规模容器化工作负载时最为流行的选择[50]，并且已经成为了容器编排领域的行业标准。如果工作负载复杂，需要更为完整的生态进行管理，那么 Kubernetes 是更好的选择。

本文后续章节的实验以及云原生微服务应用在线配置优化框架的实现中，在容器编排工具方面对 Docker Swarm 和 Kubernetes 两种容器编排方案进行了支持。

2.2 微服务架构

软件架构是软件工程的关键组成部分，随着软件工程技术的不断提高，软件架构也在不断发展演进。最近几年，随着云计算、物联网、大数据和人工智能等技术的普及，更加需要软件架构具有弹性、可扩展性、可演化性以及可维护性。同时，在设计软件架构时还需考虑到安全性、可靠性和性能等方面的因素。因此，软件架构朝着更加灵活、易用、高效和可靠的方向发展。在软件架构的发展历程中，出现了一系列经典的架构风格，比如巨石架构、面向服务架构（Service-oriented Architecture，SOA）和微服务架构等。

巨石架构是一种整体化的系统设计方式，其特点在于所有的功能和数据都被集中在一个单独的应用中，而没有明显的分层结构和功能分工。整个应用的进程紧密地耦合在一起，作为一个整体提供服务。但是这意味着，任何需求变更都会影响到整个应用。随着代码库的增长，代码的复杂度增加，导致代码库的维护成本急剧上升，增加或改进一个巨石架构应用的功能会变得十分困难，巨石架构为应用增加了可用性的风险。

SOA 是一种基于服务的软件架构风格，它将应用程序拆分为独立的服务单元，这些服务单元可以独立部署、独立开发以及独立升级，服务之间通过标准化的接口进行通信，以完成特定的业务功能[51]。SOA 的设计目标是提高应用的可维护性、可扩展性

和可重用性，降低应用开发和维护的成本。

与 SOA 相比，微服务架构是一种更加细粒度的服务化设计风格。微服务架构强调的是将应用程序拆分成更小的、自治的服务单元，每个服务单元都有自己的数据存储、业务逻辑和界面。微服务架构的设计目标是实现更加灵活、可伸缩和可维护的应用。云原生环境下应用通常使用微服务架构。如图 2.4 所示，与传统的巨石架构和 SOA 相比，微服务架构具有许多优势[52]。首先，每个微服务都是独立的，并且功能专一，这意味着它们可以单独开发和部署，使得微服务架构具有更高的弹性，可以随时进行更新和扩展以满足特定的功能需求。其次，微服务架构支持业务代码的持续交付、集成和快速部署，这使得应用从开发到上线的时间缩短，从而提高了应用开发效率。因此，微服务架构更符合现代软件开发的需求，在现代软件开发中具有重要的地位和价值。但微服务架构并不是一颗银弹，在为软件开发提供优势的同时，也令应用的监控、维护以及性能优化等方面的工作难度更高。

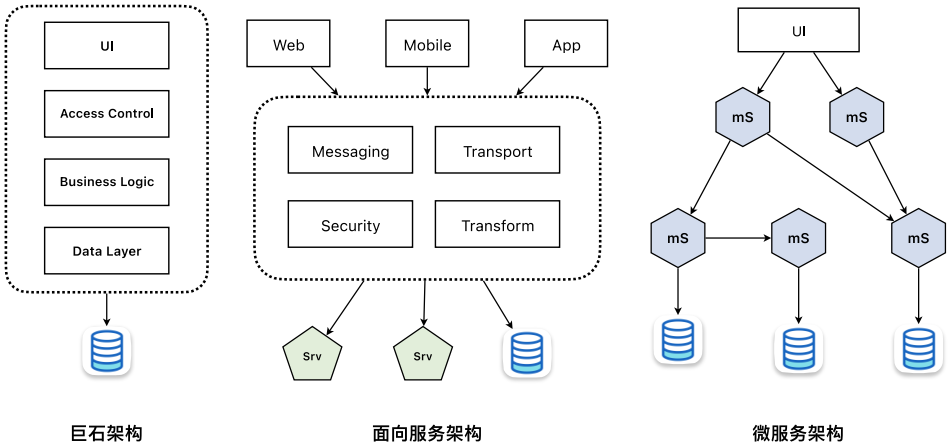

图 2.4 软件架构对比

Figure 2.4 Software Architecture Comparison

2.3 DevOps 相关技术

DevOps[53]技术可以极大地促进微服务架构应用的交付和部署速度，而微服务架构其独立部署和可测试的特性为 DevOps 提供了更快速、更高效的测试和部署的途径，因此 DevOps 和微服务架构形成了一个协同配合的整体。

传统的软件开发方法和架构，会耗费开发团队数月的时间进行应用基础功能的研发。并且开发新功能会对大部分原有代码产生影响，从而使开发团队花费大量时间将其集

成到原代码库中。在集成完成后，质量保证和安全团队对当前代码库进行测试，这也需要耗费大量时间。因此，使用传统开发方式的团队难以对市场和用户的需求进行快速响应并上线。为了应对快速变化的需求，敏捷软件开发方法已开始在开发团队中流行，逐渐取代了传统的开发方法。传统的软件管理是一种线性的模式，而敏捷开发的核心思想是对软件生命周期进行迭代式管理，它把软件开发周期分成若干个小周期，从而使得代码库可以进行小而频繁的更新迭代。敏捷开发主要包括持续集成和持续交付两个方面，能够使软件团队快速响应变化的需求，加速软件开发和交付的流程。

DevOps 是一种基于敏捷开发思想的流程，它在软件生命周期中引入新的工具和流程以提高软件开发效率和质量。DevOps 主要由软件开发（Development）和 IT 技术运维（Operations）两部分组成，强调开发人员和运维人员之间的合作和沟通。DevOps 也提倡持续集成和持续交付，确保软件的构建、测试和发布过程更加快速、频繁且可靠。与传统的软件开发方法及基础设施的管理流程相比，DevOps 能够更快速地推进软件生命周期中的流程，从而使企业在市场竞争中获得更大的优势。Dynatrace[54]把 DevOps 的生命周期划分为八个关键阶段，这些阶段主要分为开发阶段和运维阶段两个大类。如图 2.5 所示，DevOps 生命周期中的各阶段是相互补充和互相依存的，各阶段的工作通过协作完成，且需要不断循环迭代以提高 DevOps 的效率与质量。

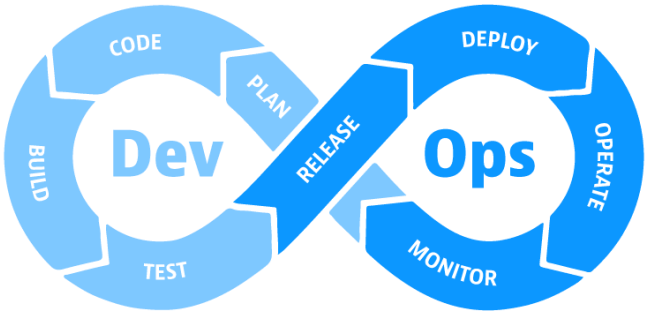

图 2.5 Dynatrace 对 DevOps 生命周期的划分

Figure 2.5 Dynatrace's Segmentation of the DevOps Lifecycle

在云原生时代，GitOps[55]也为越来越多的 DevOps 团队所采用，这是由 Weaworks 提出的一种管理云原生应用和基础架构的工作流程。GitOps 是一种基于 Git 仓库的持续交付模式，它将应用程序的配置和基础设施管理操作都纳入到 Git 版本控制系统中进行管理。开发人员在 Git 仓库中编写和管理代码的同时，使用 Git 仓库管理系统及应用相关的配置，通过持续集成和部署流水线自动地将代码部署到生产环境。当需要更新系统

及应用配置时，只需要提交相关文件到 Git 仓库，GitOps 工具可以自动的将更改应用到生产环境中。GitOps 的优势在于它简化了应用部署和管理的流程，并且提高了应用的可靠性和可维护性。因为所有配置都存储在 Git 仓库中，团队可以更容易地管理和审查变更，并且在应用出现问题时快速地回滚到之前的状态。此外，GitOps 还可以帮助保持生产环境的一致性。综上所述，GitOps 是一种利用 Git 管理系统配置的方法，通过自动化的 CI/CD 流水线实现应用持续交付和部署，提高应用系统的可审核性和可复制性。本文将在实现云原生微服务应用在线配置优化框架的过程中，充分利用 DevOps 和 GitOps 的理念和工具，在保证参数配置的可靠性和一致性的同时，实现配置优化框架的流程自动化以及微服务应用配置参数的快速变更。

第三章 云原生微服务应用的资源配置与软件参数协同优化

微服务架构非常适合用于设计面向用户的在线应用，在这种类型的应用中，对用户和应用提供方来说，性能都是非常重要的因素。通常一个完整的面向用户的在线微服务应用包括前端服务（如 Nginx）作为服务网关，提供流量管理及认证授权等功能；业务逻辑服务，根据用户请求数据提供逻辑处理和数据转换等功能；存储服务，通常各业务逻辑服务拥有独立的存储后端（如 Redis、MongoDB），提供用户数据存储以及加速缓存功能。典型的在线微服务应用架构如图 3.1 所示。

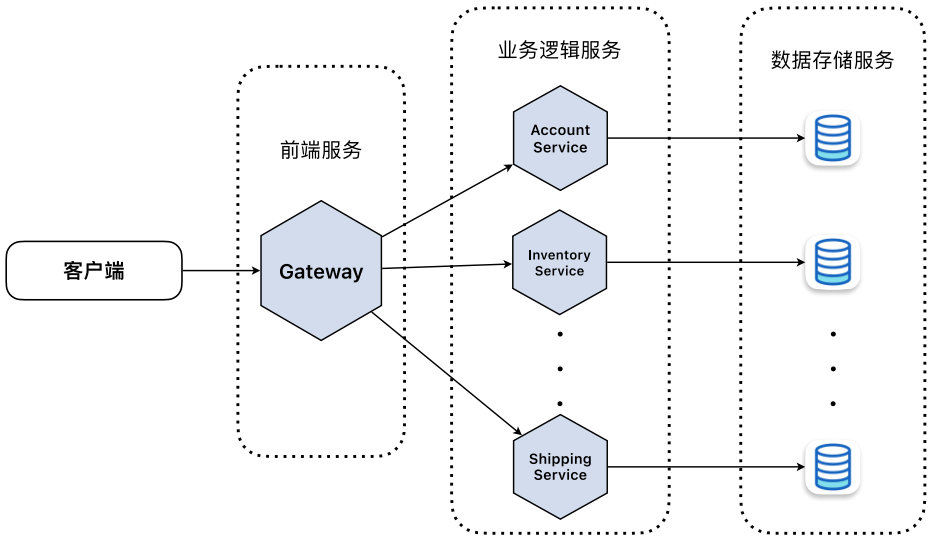

图 3.1 微服务应用组成

Figure 3.1 Microservice Application Composition

尽管微服务架构具有众多优点，但是也给应用持续稳定高效的运行带来了巨大挑战。配置参数会对应用性能产生重大影响，促使研究人员对应用的配置优化进行了大量研究。然而，现有的相关工作主要集中于传统软件的配置优化问题，很难直接迁移到微服务架构的应用。少量微服务场景下的相关工作主要关注系统资源的分配和管理，最近几篇工作[38-40]开始关注到微服务应用中软件参数的优化，目前尚未有相关研究将微服务应用的资源配置与软件参数协同考虑进行配置优化，这导致了应用性能优化不充分。本章提出了云原生微服务应用资源配置和软件参数的协同优化的方案，将贝叶斯优化应用到微服务应用配置优化问题上，并在本地集群上进行实验，验证了协同优化方案的有效性，相比单独调整资源配置或软件参数的方案能够进一步提升应用性能。

3.1 问题描述

本章重点关注如何通过协同调整云原生环境中微服务应用资源配置和软件参数，进行微服务应用性能优化。微服务应用的配置参数调整可以看作一个优化问题。设$f(c)$表示在参数配置c下运行微服务应用并对其进行负载测试后所得到的性能。其中，参数配置c是由微服务应用中所有可调节参数组成的向量，该向量的维数非常高，假设c的维数为N，则$c_i(i = 1,2,\dots,N)$表示参数配置中第i个参数的取值。注意这里所说的参数配置是指所有参数取值组成的向量。设C为参数配置c的所有可能取值的集合，$c^* \in C$表示微服务应用性能指标达到最优时的参数配置，即：

$$c^* = argmax_{c \in C}\big(f(c)\big) \tag{3.1}$$

综上所述，云原生环境下微服务应用配置优化问题的目标就是找到最优参数配置c^*，以优化微服务应用的性能。

在云原生环境下微服务应用的场景中，为了最大化应用性能，可以取性能指标的负数形式作为目标函数。例如，对延迟指标取反，则在达到最大性能指标时，微服务应用的延迟是最低的。在微服务场景下无法对配置优化目标函数$f(\cdot)$的具体结构做出假设，此外，结合实际情况，找到最优参数配置需要在合理的时间内完成。需要特别说明的是，微服务应用中有许多指标（如每秒请求数量，延迟）可用于评估应用性能[56]，目标函数$f(\cdot)$可以代表任何能够反应微服务应用性能的指标，本章选取工业和学术界常用的P_{99}端到端延迟[57-59]来评估微服务应用的性能。

3.2 研究挑战

传统软件的配置优化工作已经是一项十分具有挑战的任务，在微服务架构下，通过资源配置与软件参数协同优化的方式，对微服务应用进行性能优化更加难以实施，因为它面临着如下挑战：

（1）参数之间依赖复杂。首先单个服务内部的参数存在相互依赖，一个参数往往不只对服务的一个方面产生影响。例如，在 MongoDB 中调整缓存参数，不仅会影响到缓存的大小，还会影响到数据库处理并行读事务的能力。其次微服务应用中不同的服务依赖彼此的输出完成自身功能逻辑，因此单个服务上参数的影响范围会扩展到与其存在依赖关系的其它服务。最后微服务应用中服务通常以容器形式部署，不同的容器可能部署在同一主机上,共享同一物理资源，例如内存、磁盘空间、CPU 等，因此存在

资源竞争的现象，系统资源的设置存在干扰关系。总的来说微服务应用的配置优化问题需要考虑参数之间存在复杂的依赖关系。

（2）配置参数与应用性能之间关系非线性。云原生环境下微服务应用性能与配置参数的取值之间并不是简单的线性关系。系统线程池是一个很好的例子，并不是线程池越大微服务应用性能越高：如果设置的值过低，会导致 CPU 资源利用率较低，线程池资源不足；而如果设置的值过高，则可能导致 CPU 资源的竞争。因此，线程池大小过低或者过高都会降低应用性能。配置参数与性能之间非线性的关系，使得预测微服务应用配置优化的结果非常困难，很难通过训练为云原生环境下微服务应用的配置优化问题建立一个准确的性能预测模型。

（3）高维度的参数搜索空间。云原生环境下的微服务应用规模庞大，大型微服务应用通常由数百至数千个相互作用的服务组成，每个服务都可以对其 CPU、内存等系统资源进行配置。此外，微服务应用部署的软件（如 Nginx、Redis、MongoDB 等）还额外增加了大量可调节的软件参数，并且这些参数的取值可能是连续的，也可能是离散的，因此可配置的参数共同组成了高维度的参数搜索空间。

为了解决上述挑战，需要正确、高效的方法来描述微服务应用的配置优化问题。

3.3 参数搜索空间降维策略

本节将对上述云原生环境下微服务应用具有高维的参数搜索空间这一挑战进行深入讨论，并介绍了基于关键服务识别的有效解决方法。

在微服务应用中，假设存在n个服务，每个服务有$p_i(i = 1,2, \dots, n)$个可供调节的参数，因此，参数配置向量的维度$N = \sum_{i=1}^{n} p_i$。如果平均每个参数有v个可能的取值，那么所有可能参数配置的数量$|C| = v^N$，这表明随着服务数量的增加，参数的搜索空间呈指数级增长。为了缩小参数搜索范围需要采取适当的策略，以使配置优化工作集中在微服务应用中某一服务子集上。关键服务识别方法就是一种有效的解决方案，它可以确定哪些服务是影响微服务应用性能的关键，并将配置优化工作的重点放在这些服务上，以有效地控制参数搜索空间的大小。

3.3.1 关键路径识别

微服务中的关键路径是调用链路最长的一条路径[60]，关键路径上的服务对微服务应用性能有很大的影响，在整个请求处理过程中，路径上任何一个服务的延迟或故障

都可能影响该调用链的端到端延迟。因此，可以将关键路径上的服务配置优化作为配置优化工作的重点。

本节使用 Uber 开源的分布式链路追踪工具 Jaeger[61]来收集微服务应用运行时的调用链路信息。为了识别微服务应用中的关键路径，本节采用了加权最长路径算法[32]。该算法通过遍历微服务应用调用链路生成的依赖图，找到所有在关键路径上的服务并返回。具体关键路径的识别算法如算法 3.1 所示。

算法 3.1：关键路径识别算法

输入：微服务应用的调用链路图T，调用链路中的起始服务$curNode$

输出：微服务应用的关键路径CP

```
1：procedure CRITICALPATH(T, curNode)
2：     初始化关键路径CP ← ∅
3：     将当前节点所代表的服务加入到CP
4：     if curNode.childNodes == None then
5：         返回关键路径CP
6：     end if
7：     获取当前节点上一个返回节点lrc ← curNode.lastReturnChild
8：     执行CriticalPath(T, lrc)获取以lrc为起始节点的关键路径curCP
9：     将curCP中的节点添加到CP中
10：    for cn in curNode.childNodes do
11：        if cn.happenBefore(lrc) then
12：            执行CriticalPath(T, cn)获取以cn为起始节点关键路径curCP′
13：            将curCP′中的节点添加到关键路径CP中
14：        end if
15：    end for
16：    返回关键路径CP
17：end procedure
```

3.3.2 关键服务提取

微服务应用关键路径是由多个相互依赖的服务组成的，这些服务依赖彼此的输出完成自身业务逻辑。当关键路径上的服务任何一个遇到性能瓶颈时，都会令该调用路径的端到端延迟升高。值得注意的是，并非关键路径上的所有服务对端到端延迟有同等程度的影响。事实上，微服务应用某一调用链路的性能，关键在于该链路上性能表现最差，最不稳定的服务。因此，有必要对关键路径上的服务更进一步的分析，以确定

和优化对微服务应用性能起决定性作用的关键服务。此外，识别这些关键服务还能更进一步减少了参数搜索空间，从而使优化算法的执行效率更高。

本节介绍关键服务提取算法，它对关键路径上服务的延迟数据进行处理与分析，进一步识别对微服务应用性能起决定性作用的关键服务，然后将精力集中于关键服务的配置优化，搜索这些服务具有的资源配置与软件参数以优化微服务应用的性能。关键服务提取算法如算法 3.2 所示。

算法 3.2：关键服务提取算法

输入：微服务应用的关键路径CP，所有服务的延迟指标数据L

输出：微服务应用中的关键服务CS

1：初始化关键路径$CS \leftarrow \emptyset$

2：获取关键路径上的端到端延迟指标$L_{cp} \leftarrow L.getCPLantency()$

3：**for** cn **in** CP **do**

4：　　获取当前节点的延迟数据$l_{cn} \leftarrow L.getLantency(cn)$

5：　　使用皮尔逊相关系数计算当前节点对微服务性能的影响$c \leftarrow PCC(l_{cn}, L_{cp})$

6：　　**if** $c < \text{threshold}_{pcc}$ **then**

7：　　　　**continue**

8：　　**end if**

9：　　获取当前节点P_{99}延迟指标$P_{99} \leftarrow l_{cn}.P_{99}$

10：　　获取当前节点P_{50}延迟指标$P_{50} \leftarrow l_{cn}.P_{50}$

11：　　**if** $\frac{P_{99}}{P_{50}} < \text{threshold}_{max}$ **then**

12：　　　　**continue**

13：　　**end if**

14：　　将当前节点加入关键服务列表CS

15：返回微服务应用中的关键服务CS

算法中使用皮尔逊相关系数（Pearson Correlation Coefficient，PCC）[62]量化当前服务延迟与整个关键路径端到端延迟的相关性，PCC 被定义为两个变量的协方差除以它们标准差的乘积：

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{3.2}$$

服务的 PCC 越高，代表该服务对微服务应用整体性能的影响越大，关键路径上所有服务的 PCC 之和为1，即

$$\sum_{cn\in CP} PCC\left(l_{cn}, L_{cp}\right) = 1 \tag{3.3}$$

其中$L_{cp} = \sum_{cn\in CP} l_{cn}$代表关键路径上所有服务的端到端延迟之和。对 PCC 超过阈值的服务进一步通过P_{99}端到端延迟和P_{50}端到端延迟之间的比值量化性能指标的波动性，P_{99}/P_{50}较大时，意味着该服务长尾延迟较大，将该服务加入关键服务列表，对其进行配置优化以提升微服务应用的性能。

3.4 基于贝叶斯优化算法的配置优化方法

针对云原生环境下微服务应用资源配置与软件参数协同优化方案中，存在参数之间依赖复杂以及配置参数与应用性能关系非线性的问题，本章将其视为一个黑盒优化问题，并使用配置优化领域被广泛采用的贝叶斯优化算法进行解决。贝叶斯优化[63]是最先进的黑盒优化方法，与其他优化方法相比可以更快地收敛到最优解。贝叶斯优化算法为目标函数建立概率代理模型（Surrogate Model），迭代地更新该模型以指导搜索最有可能的解，并且能够平衡在已知具有较优目标值的区域进行搜索和全局未探索的区域进行搜索两者之间的权重。目前贝叶斯优化已被广泛应用于很多领域，如机器学习的超参数调整[64]、机器人技术[65]和资源优化[66]等方面。贝叶斯优化算法的问题形式通常为：

$$max_{x\in A}\, f(x) \tag{3.4}$$

其中x是一个d维向量，在微服务应用配置优化问题中可以表示为各参数取值，A是配置参数取值空间。$f(\cdot)$表示目标函数，可以认为是一个具有某些未知结构的黑盒，在微服务应用配置优化问题中可以表示当前参数配置对应的性能。

在贝叶斯优化的每次迭代过程中，通过最大化采集函数（Acquisition Function）的值以选取下一个探测点。采集函数将概率代理模型对未探测点的预测作为输入，并评估该点相对于当前最佳结果的改进，采集函数旨在平衡探索（Exploration）和利用(Exploitation)这两个阶段：探索是指在搜索空间内探寻未知的区域，以发现可能更好的解决方案；利用是指利用已有的信息，在已知具备良好效果的解附近进行采样，以找到更好的解决方案。在确定下一个探测点后，贝叶斯优化算法在目标系统上对该点进行评估，并收集相应指标，然后更新概率代理模型。综上可以看出概率代理模型和采集函数是贝叶斯优化中的两个关键部分。

概率代理模型用于代理评估代价高昂且复杂的目标函数，它将目标函数看作一个概

率分布，并通过采样的方式来优化该分布。概率代理模型根据模型的参数个数是否固定分为参数模型和非参数模型，非参数模型中的高斯过程由于其易于扩展、易于优化、灵活性高以及能够应对不同种类的目标函数等优势，成为了贝叶斯优化算法中常用的概率代理模型。

高斯过程的定义[67]为：令$\{X_t, t \in T\}$是一个高斯过程，当且仅当对下标集合T的任意子集$t_1, \ldots, t_k$：

$$X_{t_1,\ldots,t_k} = \left(X_{t_1}, \ldots, X_{t_k}\right) \tag{3.5}$$

是一个多元正态分布，这等同于说$\left(X_{t_1}, \ldots, X_{t_k}\right)$的任一线性组合是一单变量正态分布。更准确的说，取样函数X_t的任一线性泛函均会得出正态分布。可以写成：

$$X \sim GP(m, K) \tag{3.6}$$

即随机函数X以高斯过程（GP）方式分布，且其均值函数为m，协方差函数为K。

采集函数用于在搜索过程中选择下一个探测点，它计算每个候选点探索和利用的价值。不同的采集函数对这两者的重要性有不同的权衡，因此选择哪个采集函数会影响到最终的优化结果，下面介绍几个常见的采集函数：

（1）提升概率（Probability of Improvement，PI），测量一个新样本点比到目前为止观察到的最佳样本点具有更好的目标值的概率。在数学上，PI 函数可以表示为：

$$PI(x) = P\left(f(x) \geq \left(f(x^*) + \epsilon\right)\right) \tag{3.7}$$

其中$f(\cdot)$是近似真实目标函数的代理函数（通常是高斯过程），$f(x^*)$是到目前为止观察到的最佳目标值，P是从代理函数中获得的平均值和方差正态分布的累积分布函数。PI 值越高，新样本点x改进当前最佳样本点的可能性就越大。

（2）期望提升（Expected Improvement，EI），测量了新样本点相对于当前最佳样本点能为目标函数值带来提高的期望。在数学上，EI 函数可以表示为：

$$EI(x) = E[\max(f(x) - f(x^*), 0)] \tag{3.8}$$

其中E为数学期望，EI 值越高，新样本点x就越有可以改进当前最佳样本点的前途。

（3）置信上界（Upper Confidence Bound，UCB），选择具有最高置信上界的采样点，是代理模型期望提升和不确定性的加权和。在数学上，UCB 函数可以表示为：

$$UCB(x) = \mu(x) + \kappa\sigma(x) \tag{3.9}$$

其中$\mu(x)$是代理模型对于采样点x处期望值，$\sigma(x)$是采样点x处代理模型的标准差，κ是探索和利用的权衡参数。κ值越高，算法将越有可能对搜索空间的新区域进行探索。

3.5 实验与分析

3.5.1 实验环境设置

为了验证本章提出的资源配置与软件参数协同优化，在云原生环境下微服务应用配置优化问题上的有效性和效率，本章使用三台物理主机搭建了集群环境，每台主机的CPU 型号为 Intel i7-10700，拥有 16 个核心，并配备了 16GB DDR4 内存和 1TB 机械硬盘，主机之间通过局域网进行连接。

在该集群环境上，本节针对 DeathStarBench[68]中的社交网络微服务应用在本地集群上进行实验，该微服务应用模拟了真实世界的社交网络应用（如 Twitter），实现了注册、关注用户、发表文章、查询用户时间线等功能。实验过程中参考了工业界常见做法将社交网络微服务应用按照其每个服务的功能进行分类，如图 3.2 所示，然后将它们分别固定部署到本地三台物理主机上，避免了集群自动调度策略造成的性能波动。

实验过程中采用P_{99}端到端延迟作为微服务应用性能评估指标，需要说明的是除P_{99}端到端延迟之外其它合理的指标也是可取的。此外，为了保证实验的稳定性，每一次对微服务应用进行负载测试前都进行了系统预热，等到系统稳定后再生成负载以获取当前参数配置下微服务应用所能达到的性能。

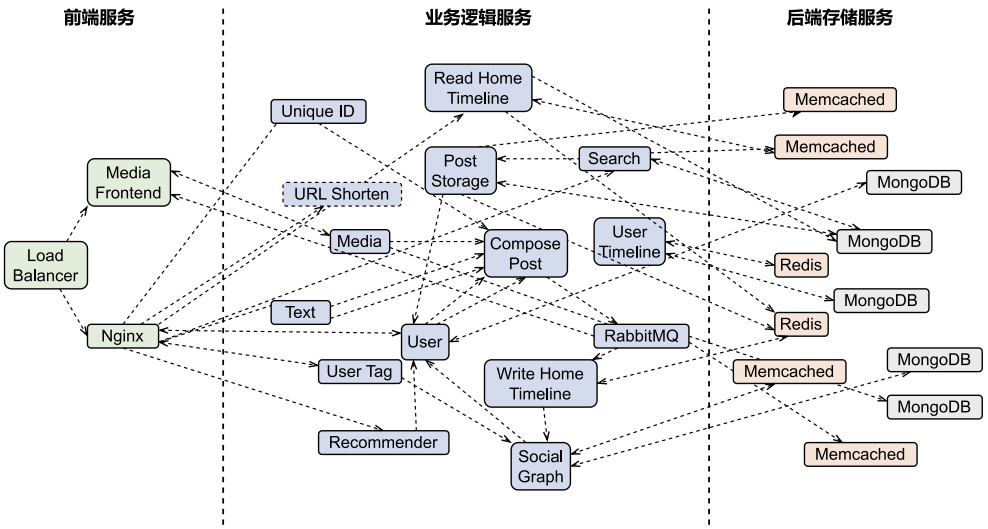

图 3.2 社交网络微服务架构

Figure 3.2 Social Network Microservice Architecture

在资源配置与软件参数协同优化的实验中，资源配置方面对社交网络微服务应用中每个服务部署容器的 CPU 以及内存系统资源参数进行调整，软件参数方面对微服务应用中存在的 Nginx、Redis、MongoDB 和 Memcached 这些软件的参数进行调整。本节参

照 DeathStarBench 开发者建议以及相关软件文档，设置了系统资源参数与软件参数的取值范围及默认值，其中每个软件至多选取 5 个最具影响力的参数，相关信息如表 3.1 所示。

表 3.1 软件参数空间

Table 3.1 Software Parameter Space

参数名称	取值范围	默认值
redis.maxmemory	[512,4096]	1024
redis.maxmemory-samples	[2,10]	5
redis.hz	[1,100]	10
redis.maxmemory-policy	noevictio, allkeys-lru allkeys-lfu, volatile-lru volatile-lfu, allkeys-random volatile-random, volatile-ttl	noeviction
mongodb.cacheSizeGB	[0.5,4]	1
mongodb.syncPeriodSecs	[5,300]	60
mongodb.blockCompressor	none, snappy zlib, zstd	snappy
memcached.m	[16,512]	32
memcached.c	[256,4096]	1024
memcached.t	[1,16]	4
nginx.worker_processes	[2,16]	8
nginx.worker_connections	[512,4096]	1024
nginx.threads	[2,48]	4
nginx.max_queue	[512,65536]	4096
nginx.tcp_nodelay	on, off	on

3.5.2 实验结果分析

（1）优化方案对比实验

本节实验比较了三种配置优化方案：只优化软件参数、只优化资源配置以及协同优

化资源配置与软件参数。图 3.3 表示在每秒 400 请求和每秒 500 请求的工作负载下，分别使用三种配置优化方案对社交网络微服务应用关键服务进行优化时，应用在优化算法每次迭代搜索到的参数配置下相应的P_{99}端到端延迟，该延迟越低越好。从结果中可以看出，由于此时负载吞吐量较低，社交网络应用对系统资源的需求不高，因此资源配置对应用性能的影响相对较低，而优化软件参数可以在低负载的情况下，带来比资源配置优化更高的性能提升，两者协同优化可以进一步降低应用端到端延迟。更具体的，只优化软件参数、只优化资源配置以及协同优化资源配置与软件参数三种配置优化方案，在每秒 400 请求的吞吐下，相比默认配置（取多次执行均值）可以分别为微服务应用带来 10.04%、2.25%以及 17.64%的性能提升；在每秒 500 请求的吞吐下可以分别带来 18.69%、13.41%以及 33.28%的性能提升。

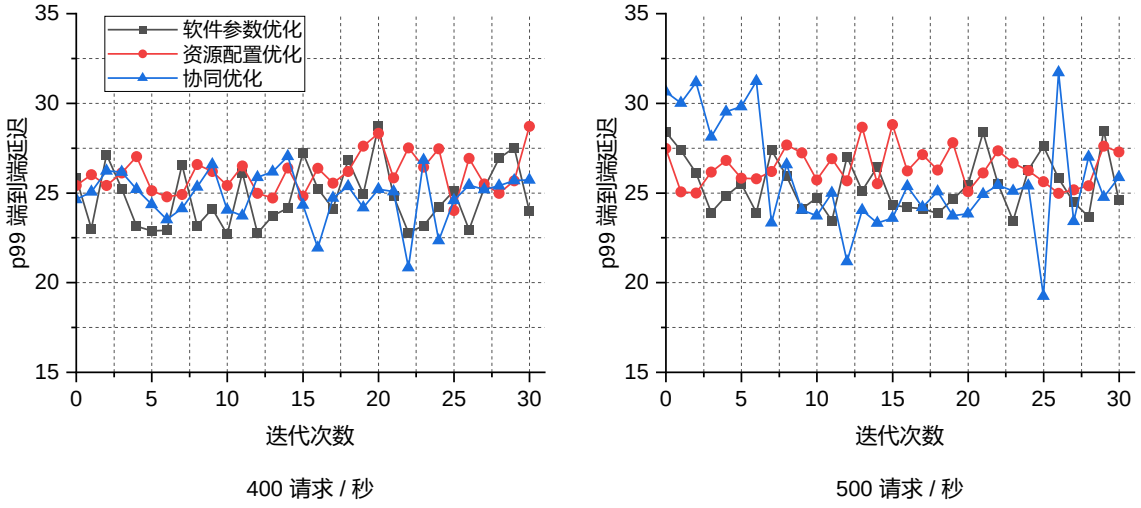

图 3.3 配置优化过程对比

Figure 3.3 Comparison of Configuration Tuning Processes

本节实验除上述负载吞吐量外，还在其它规模负载上进行了测试，结果如表 3.2 所示，可以观察到在负载吞吐量过高时，优化软件参数并不能有效地提高微服务应用性能，因为此时性能瓶颈是由系统资源不足造成的，调整资源配置能为微服务应用性能带来更大程度的提升。以负载吞吐量为每秒 800 请求为例，默认配置下其平均P_{99}端到端延迟为 69964.02 毫秒，此时已经难以满足用户需求。通过只调整微服务应用中 Nginx、Redis、Memcached、MongoDB 这些服务的软件参数，最优参数配置下P_{99}端到端延迟为 18819.58 毫秒。通过只调整这些服务的资源配置，最优情况下性能指标为 60.65 毫秒，与前者相比提升了近 3 个数量级。然而，这两种单独优化的方案未对微服务应用的参数配置空间进行充分的探索，同时对资源配置与软件参数进行协同优化，搜寻到的

最优配置能令微服务应用端到端延迟进一步降低，达到 43.21 毫秒。此外，通过表 3.2 的最后一列可以看出，资源配置和软件参数协同优化方案相较于两种单独优化的方案，在不同负载吞吐量下均能带来更大的微服务应用性能提升，且负载吞吐量越大，提升效果越明显。

表 3.2 配置优化方案效果对比

Table 3.2 Comparison of the Effectiveness of Configuration Tuning Schemes

负载吞吐量（requests/sec）	默认参数配置下性能（ms）	只调软件参数的最优性能(ms)	只调资源参数的最优性能(ms)	两者协同调整的最优性能（ms）
400	25.29	22.75	24.72	20.83
500	28.84	23.45	24.97	19.24
600	35.69	34.76	30.78	21.19
700	37873.85	7229.81	38.24	35.73
800	69964.02	18810.58	60.55	43.21
1000	101970.0	68145.21	8383.69	3183.42
1200	243623.5	188653.11	9708.17	5912.89

（2）参数降维策略有效性实验

本节实验在云原生环境下微服务应用配置优化的过程中，采用不同程度的参数搜索空间降维策略，进行资源配置与软件参数协同优化，然后对比其优化结果。具体来说本节实验采用贝叶斯优化算法，在不同工作负载下分别对微服务应用中所有服务、关键路径上的服务以及经过关键服务提取算法识别到的服务，进行资源配置与软件参数协同优化，获取最优参数配置下微服务应用P_{99}端到端延迟，计算相对于默认配置下的性能提升百分比。如图 3.4 所示，这些结果清楚地表明，基于关键服务的参数搜索空间降维策略可以显著提高微服务应用配置优化的结果。以每秒 600 请求量的负载为例，协同优化社交网络微服务应用的 28 个服务的资源配置与软件参数，可以获得 30%~34%的性能提升。调整关键路径上的服务可以获得 33%~37%的性能提升，然而仅对关键服务的配置参数进行优化可以获得 39%~42%的性能提升。这表明，仅调整关键服务提取算法识别到的服务比调整微服务应用中的全部服务能够带来更大程度的性能提升。

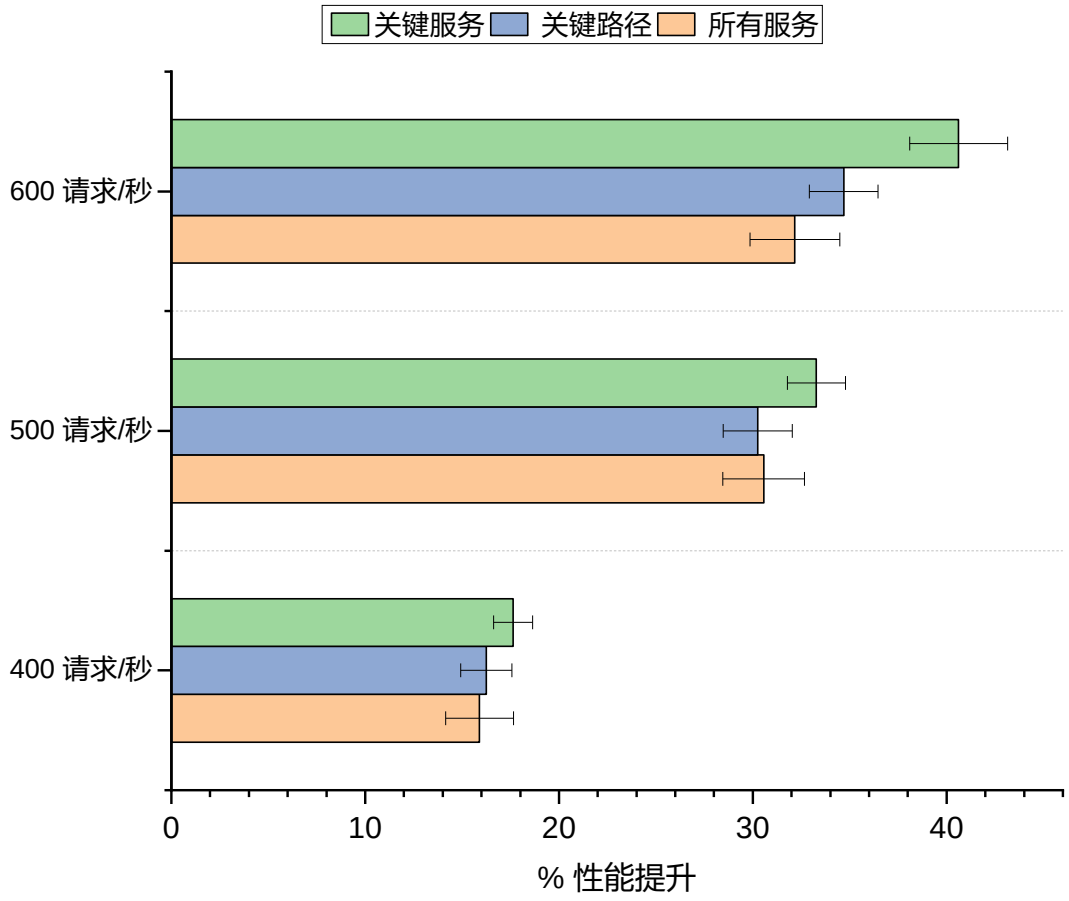

图 3.4 不同搜索空间优化效果对比

Figure 3.4 Comparison of Optimization Performance on Different Search Spaces

产生该现象的原因在于，关键服务提取算法正确识别到了对微服务应用端到端延迟影响最大的服务。此外，贝叶斯优化算法在高维参数搜索空间时表现不佳[69]，通过提取关键服务，需要进行配置优化的服务减少了约 79%，大幅降低参数搜索空间的维数，提高了优化算法的效率。

通过上文关键路径识别算法以及关键服务提取算法定位到的服务，如表 3.3 所示。其中基于关键路径识别算法定位到的服务含有 13 个，相比社交网络微服务应用全部 28 个服务，数量减少了约 54%。基于关键服务提取算法定位到的服务含有 6 个，在关键路径的基础上需要进行配置优化的服务进一步减少了约 53%，相比社交网络全部服务减少了 79%。

表 3.3 服务提取结果

Table 3.3 Service Extraction Results

	关键路径上服务	关键服务
nginx-thrift	✓	✓
media-frontend		
media-memcached		
media-mongodb		
user-memcached		
user-timeline-mongodb	✓	
user-mongodb		
post-storage-mongodb	✓	✓
post-storage-memcached	✓	✓
url-shorten-mongodb		
url-shorten-memcached		
social-graph-mongodb		
social-graph-redis	✓	
home-timeline-redis	✓	
compose-post-redis	✓	
user-timeline-redis		
social-graph-service		
write-home-timeline-service	✓	
compose-post-service	✓	✓
post-storage-service		
home-timeline-service	✓	✓
unique-id-service		
media-service		
url-shorten-service	✓	
user-timeline-service	✓	✓
user-service		
text-service	✓	
user-mention-service		

（3）优化目标对比实验

通常情况下企业倾向于保守的资源预留策略，为微服务分配超过其实际所需的系统资源量，以保证峰值负载期间对用户请求的稳定响应。然而这种策略会造成系统资源的浪费，特别是在微服务应用负载不高时，会使整体系统资源利用率低下，造成不必要的资源浪费。随着云计算服务普及，用户需要根据系统资源用量向云服务供应商支

付费用，表 3.4 为阿里云部分云服务器实例报价。

表 3.4 阿里云服务器实例报价

Table 3.4 Alibaba Cloud Instance Pricing

实例规格	vCPU	内存（GB）	价格（￥/小时）
通用型(g6) ecs.g6.large	2	8	0.35
通用型(g6) ecs.g6.xlarge	4	16	0.7
内存型(r6) ecs.r6.large	2	16	0.46
内存型(r6) ecs.r6.xlarge	4	32	0.92
计算型(c6) ecs.c6.large	2	4	0.27
计算型(c6) ecs.c6.xlarge	4	8	0.55
通用型(g5) ecs.g5.large	2	8	0.66
通用型(g5) ecs.g5.xlarge	4	16	1.33
密集计算型(ic5) ecs.ic5.large	2	2	0.44
密集计算型(ic5) ecs.ic5.xlarge	4	4	0.89

为了节省用户对基础设施的持有成本，提高系统的运营效率。因此，在微服务应用配置优化时，管理人员需要充分考虑资源能效。本节实验中将资源效能定义为单位资源用量下微服务应用的性能，然后将资源效能作为微服务应用配置优化的目标函数$f(\cdot)$，其形式如下：

$$\begin{aligned} resource(s) &= \sum_{i}^{n}(s_i^{replica}(\alpha s_i^{cpu} + \beta s_i^{memory})) \\ f(c) &= \frac{perf(c)}{resource(s)} = \frac{C - P_{99}(c)}{resource(s)} \end{aligned} \tag{3.10}$$

其中$resource(s)$为参数配置方案c下，微服务应用的所有服务总系统资源用量。具体的对于微服务应用内所有的n个服务：$s_i^{replica}$表示第i个服务的副本数，s_i^{cpu}表示第i个服务的 CPU 用量，s_i^{memory}表示第i个服务的内存用量，α和β可以根据负载类型调整 CPU 和内存在系统资源开销指标中所占的权重，$P_{99}(c)$表示在参数配置方案c下微服务应用调用链路P_{99}端到端延迟，C为一个常数。

图 3.5 展示了分别以性能和资源效能为优化目标，对社交网络微服务应用进行资源配置与软件参数协同优化的效果对比。具体的，本节实验分别在每秒请求量为 500、600 和 700 的工作负载下进行测试。在每种工作负载下，取延迟指标最低 5 个配置方案，

计算其性能及资源用量平均值。可以看出在不同工作负载下，以性能和以资源能效为优化目标进行配置优化，可以在社交网络微服务应用上取得相近的最优性能。但是以资源能效为优化目标时，只需要 82%~86%的系统资源量就能达到该最优的性能，表明了以本节提出的资源能效作为优化目标，可以在一定程度上节省系统资源开销，提高了微服务应用系统资源利用率。

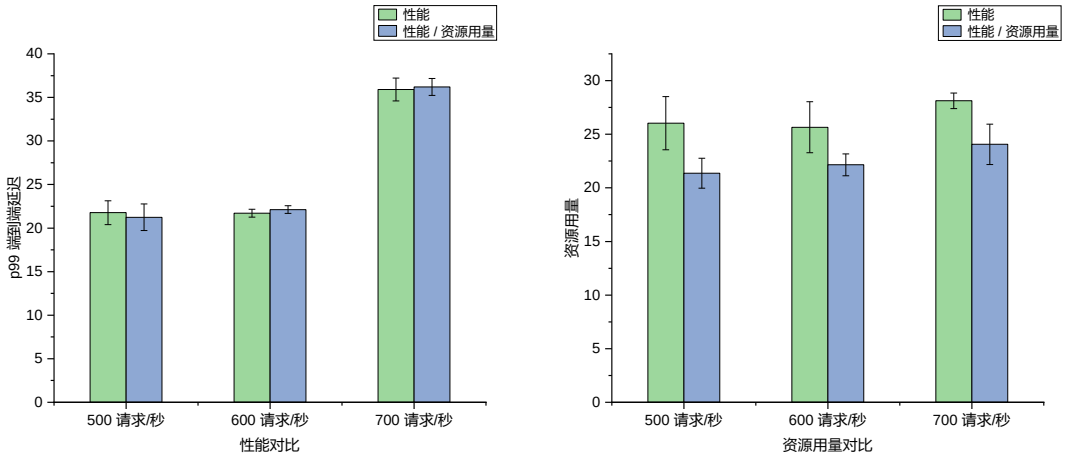

图 3.5 优化目标对比

Figure 3.5 Optimization Target Comparison

3.6 本章小结

针对云原生环境下微服务应用配置优化问题，本章提出了一种资源配置与软件参数协同优化的方案，解决已有方案不能充分探索性能优化空间的局限性。该方案使用贝叶斯优化解决参数之间依赖复杂，参数与应用性能之间关系非线性的问题；使用基于关键服务的维策略解决高纬度的参数搜索空间问题。最后通过本地集群上的实验证明了本章提出的协同优化方案的有效性，并且对其原因进行了深入分析。

第四章 云原生微服务应用在线配置优化框架设计与实现

配置优化对于应用的可靠性和性能至关重要，但云原生环境下微服务应用是分布式的并且具有高度复杂性，由许多可以进行独立开发与部署的服务组成，每个服务包含资源配置以及软件参数可供调节，组成了大量可供调节的配置参数。由于微服务应用的配置优化问题中涉及到的配置参数数量众多，且不同服务之间的依赖关系十分复杂，因此手动管理配置优化流程是耗时且容易出错的，如果配置参数设置不当，还会导致应用崩溃。为了解决难以手动操作微服务应用配置优化流程的问题，本章设计并实现了云原生微服务应用在线配置优化框架。该框架利用机器学习或传统算法自动探索广阔的配置空间，借助云原生技术栈对微服务应用进行配置变更和负载测试，最终找到最优参数配置，从而优化应用的性能。通过该框架自动化配置优化流程不仅可以节省时间和资源成本，而且可以减少配置优化过程中应用发生错误崩溃的风险，确保云原生环境下微服务应用始终稳定可用，并且最终以最优参数配置运行。

4.1 框架需求分析

本章旨在设计并实现云原生微服务应用在线配置优化框架，本节先从框架需求的角度进行分析，为后续的框架设计和实现提供有力的支持和指导。框架需求由功能性需求和非功能性需求组成。其中功能性需求是指框架需要实现的特定功能或行为，通常包括用户可以执行的操作以及框架能够提供的功能。而非功能性需求则是指在框架设计中，需要满足的非功能性要求，本章主要从可用性、性能、可靠性和可维护性这些方面分析非功能性需求。

4.1.1 功能性需求分析

为了简化云原生环境下微服务应用配置优化的流程，并避免手动调整配置参数可能产生的错误，该框架将整个微服务应用配置优化的流程自动化。总的来说该框架需要能够使用机器学习或传统算法自动探索广阔的参数搜索空间，然后借助云原生技术栈对微服务应用进行配置变更和负载测试，最终找到最佳的参数配置使微服务应用在该配置下稳定高效的运行。配置优化框架除了自动化整个微服务应用配置优化流程之外，还需要给用户提供自定义配置优化流程的能力，以满足不同用户的需求。用户可以根据自己的微服务应用特性及需求，选择不同种类的配置参数以及优化算法进行微服务应用配置优化。最后该框架还需提供系统监控的能力，以帮助用户更好地了解基础设

施和微服务应用的状态，令用户能够及时处理应用运行过程中可能出现的问题，提高应用可靠性与用户体验。综上所述本节将围绕用户需求和框架应提供能力两方面进行功能性分析。

为了满足用户对配置优化流程自定义的需求，以下功能应该被考虑：

（1）目标微服务应用选择：用户需要能够根据自身的场景选择特定类型的微服务应用进行配置优化，此外用户还可以实现框架中的微服务应用接口，然后选择自己的微服务进行配置优化。

（2）优化算法选择及算法相关参数设置：优化算法及算法相关参数设置是该配置优化框架中非常重要的功能之一。用户可以根据自己的需求和场景选择不同的优化算法，例如贪心算法、遗传算法、模拟退火等，以及对该算法的参数进行调整，例如迭代轮次、种群大小等。

（3）待调整参数类型与服务选择：在实际应用中，微服务应用可能包含多种类型的参数，例如资源配置、软件参数等。为了满足用户的需求，用户需要能够根据自己的具体情况选择需要进行调整的参数类型，以便更好地优化微服务应用的性能。另外，用户还可以选择在哪些服务上进行配置优化，如对微服务应用性能起决定性作用的关键服务，或者调用链关键路径上的服务，亦或是微服务应用的全部服务。

（4）压测负载调整：允许用户根据需求调整压测负载，包括负载吞吐量大小以及负载类型。通过改变负载大小和类型，用户可以更加全面地了解微服务应用在不同负载情况下的性能状况，从而为优化微服务应用的性能提供更为准确的数据支持。

从配置优化框架需要提供的能力方面进行考虑，主要有以下功能性需求：

（1）解析待优化的配置参数：框架需要能够正确解析用户对待优化的配置参数相关设置，然后生成待优化的配置参数集合。相关设置包括，用户可以选择优化微服务应用中不同类型的参数，即系统资源参数或软件参数，其次用户还可以选择待优化参数所在服务的范围，如微服务应用中全部服务、关键路径上的服务或关键服务。最后用户还可以为每个参数设置取值范围，以确保优化算法的搜索空间在合理的范围。

（2）自动应用参数配置：配置优化框架需要能够将优化算法生成的配置方案自动更新到微服务应用中，使微服务应用在新的配置方案下运行。此外，该功能还应确保在配置更改期间不会发生应用中断或其他错误，以保证微服务应用的可靠性和稳定性。

（3）对微服务应用进行压测：配置调优框架需要能够根据用户的负载设置对微服务应用进行压测，然后获取当前性能指标，包括延迟、吞吐量、错误率等。压测结果

可以用来评估参数配置方案的优劣，并为下一轮优化算法进行参数搜索提供参考信息。

（4）自动控制优化算法执行流程：框架需要能够自动控制优化算法的执行流程，包括自动生成参数配置方案、获得当前配置下的性能指标输入算法、推进优化算法再生成下一套参数配置以及根据用户设置终止条件，例如最大迭代次数或性能指标达到一定阈值终止优化算法的迭代。

（5）微服务应用监控：框架需要能够获取微服务应用的运行时数据，包括调用链路、性能指标和系统资源使用情况等。可以让用户实时监控微服务应用的状态和性能，并及时发现和处理问题，以提高微服务应用的可靠性和性能。此外，用户还可以使用框架提供的监控数据来了解微服务应用的性能趋势和瓶颈，以帮助他们做出更明智的配置决策。

4.1.2 非功能性需求分析

为了确保微服务应用配置优化框架的高质量和稳定性，需要考虑其非功能性需求，主要包括以下方面：

（1）可用性：框架需要保证在任何时候都能够运行和响应用户请求，即使出现一些意外的问题也应该有相应的恢复机制。

（2）性能：框架需要具备较高的响应速度和处理能力，能够快速解析参数、执行算法、生成配置方案以及自动应用参数配置。

（3）可靠性：框架需要具备高度的可靠性，能够在长时间运行过程中，不出现系统崩溃等异常情况，保持长期稳定的运行，同时需要保证生成的配置方案能够正确应用到微服务应用中。

（4）可维护性：高可维护性的设计能够让开发人员更加容易地维护和扩展代码，减少代码修改的代价和时间。这需要保证框架代码和其它资源的可读性和可理解性等。

4.2 框架设计

4.2.1 整体架构设计

根据框架需求分析的结果，为了实现微服务应用配置优化框架，需要设计一个高效、可靠、易维护的架构，本文采用了主从式架构进行了设计与实现，主从式架构也称为客户端/服务器架构或 C/S 架构。图 4.1 展示了微服务应用配置优化框架的整体架构。

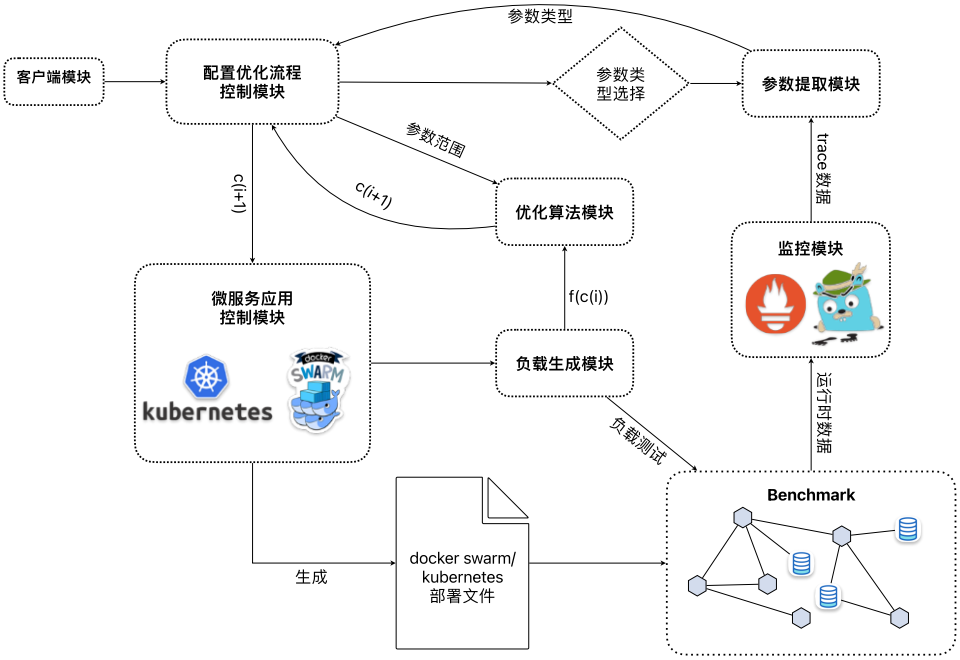

图 4.1 微服务应用配置优化框架整体架构

Figure 4.1 Microservice Application Configuration Tuning Framework Architecture

其中客户端模块是该框架的入口，提供了用户自定义配置优化过程的相关设置。配置优化流程控制模块持有优化算法实例和微服务应用控制器实例，控制整个配置优化流程的执行。优化算法实例从优化算法模块中获取，微服务应用控制器实例从微服务应用控制模块中获取。负载生成模块用于对微服务应用进行负载测试并收集性能指标。此外，该框架还包括监控模块，用于监控系统和微服务应用的运行时数据。

4.2.2 框架数据流向

图 4.2 展示了微服务应用配置优化框架中各模块之间的数据流向。其中，配置优化流程控制模块首先解析客户端对配置优化流程的相关设置，并在配置优化流程结束后向客户端返回最佳参数配置。该模块还根据客户端相关设置先从参数提取模块中获取待调整参数的名称和取值范围，再从优化算法模块获取算法实例并进行算法相关的设置，最后从微服务控制器模块获取应用控制器实例。优化算法模块在获得下一组待测试参数配置后，通过配置优化流程控制模块将其传递至微服务控制模块。微服务控制模块根据该配置生成部署文件并更新微服务应用，同时根据客户端负载相关设置使用负载生成模块对微服务应用进行负载测试，当负载测试完成后，负载生成模块将当前参数配置下的性能指标传递给优化算法模块，以便生成下一组待测试的参数配置。

监控模块负责从系统和微服务应用中收集运行时数据，然后将其中的调用链路信息传递给参数提取模块以分析微服务应用中每个服务，根据这些服务的特点可以提取不同的配置参数集合，用户可以根据自身需求选择不同的参数集合进行配置优化。

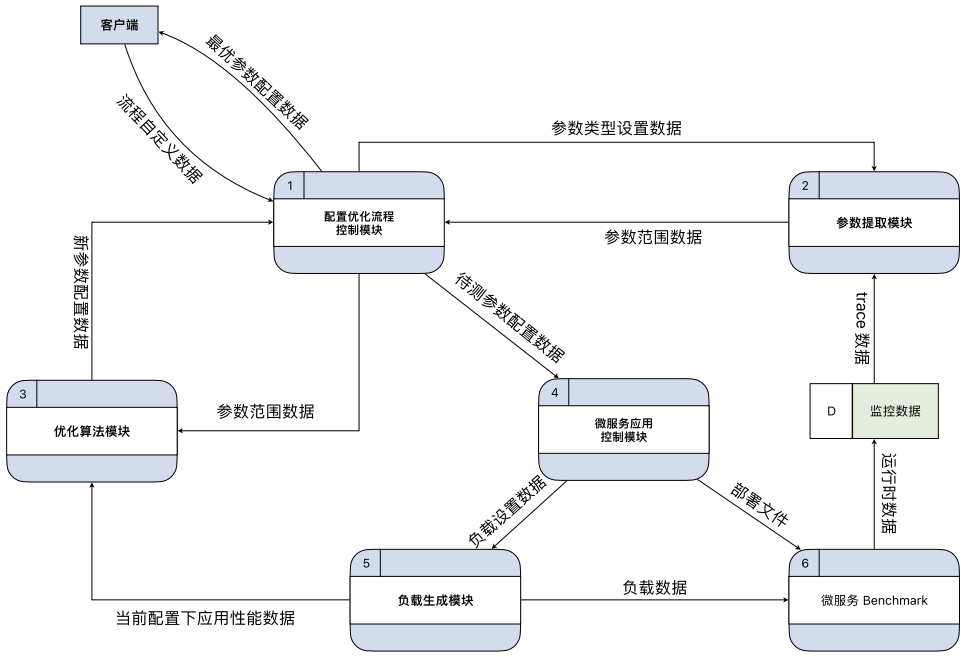

图 4.2 框架数据流向图

Figure 4.2 Framework Data Flow Diagram

4.2.3 框架层次结构

配置优化框架的层次结构如图 4.3 所示，该框架自底向上由基础设施层、操作系统层、容器编排层、应用层和客户端构成。其中每个层次的作用如下：

基础设施层：提供框架所需的基础设施资源，例如物理服务器、网络设备、存储设备等，为框架的上层提供了设备支撑。

操作系统层：安装在计算机上与底层硬件交互的软件即操作系统，用于管理和控制基础设施资源，例如内存、CPU、磁盘等，以及提供系统服务，例如进程管理、网络管理、存储管理等。

容器编排层：用于部署和管理微服务应用，提供自动化容器编排和生命周期管理的功能。本文在该层次的实现过程中对最常用的容器编排工具 Docker Swarm 和 Kubernetes 进行了封装整合。

应用层：在此层次中，微服务应用的各个组件被分解为多个独立的服务，并以容器

方式运行，借助容器编排层实现微服务应用的自动部署和配置更新等功能，同时在配置优化流程控制模块的管理下搜寻最优参数配置，优化微服务服务性能。

客户端层：作为最上层，提供用户自定义配置优化流程的相关设置，并与应用层进行交互，以获得最优的微服务应用参数配置。

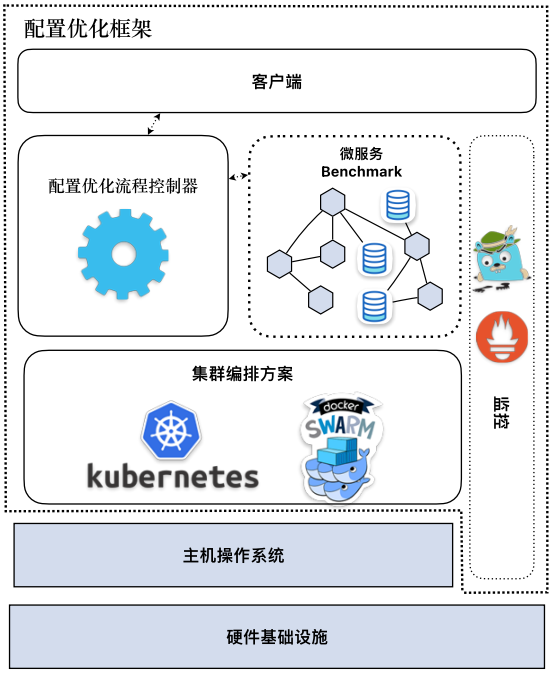

图 4.3 框架层次结构

Figure 4.3 Framework Hierarchy

此外，框架中还存在一个监控服务，贯穿操作系统层、容器编排层以及应用层，收集各层的指标并提供给客户端访问。

4.3 框架功能实现

4.3.1 客户端模块实现

客户端模块是微服务应用在线配置优化框架的入口，可以让用户对配置优化的流程进行自定义的控制。本文为客户端提供了两种实现方式：命令行工具和 RESTful API。

其中命令行工具使用 Python 中的 argparse 模块进行实现，argparse 是 Python 标准库中的一个命令行解析器，它可以处理命令行参数、选项和子命令。客户端命令行工具为 tunectl，使用"$tunectl - h$"可以查看该命令行工具的使用方式，如图 4.4 所示。

```
(tuning_microservice-3.9) master@master-ThinkStation-K:~/tuning_microservice$ tunectl -h
usage: tuning.py [-h] [-tn TASK_NAME] [-b {sn,tt,mm}] [-para PARAMETER] [-dr DIMENSIONALITY_REDUCTION] [-alg
ALGORITHM] [-acq {ei,pi,ucb}] [-it ITERATION] [-re REPITITION] [-ex EXIST]

automatic optimization framework for microservices

optional arguments:
  -h, --help            show this help message and exit
  -tn TASK_NAME, --task_name TASK_NAME
                        the name of the optimization task. can use for generating result directory.
  -b {sn,tt,mm}, --benchmark {sn,tt,mm}
                        the benchmark being used. "sn" represents for "SocialNetwork".
  -para PARAMETER, --parameter PARAMETER
                        set the kind of parameters to be optimized:
  -dr DIMENSIONALITY_REDUCTION, --dimensionality_reduction DIMENSIONALITY_REDUCTION
                        the dimensionality reduction method.
  -alg ALGORITHM, --algorithm ALGORITHM
                        the algorithm being used.
  -acq {ei,pi,ucb}, --acquisition {ei,pi,ucb}
                        the acquisition function being used.
  -it ITERATION, --iteration ITERATION
                        the number of optimization algorithm iterations.
  -re REPITITION, --repitition REPITITION
                        the number of repitition for optimaztion task.
  -ex EXIST, --exist EXIST
```

图 4.4 命令行工具 tunectl

Figure 4.4 Command Line Tool Tunectl

微服务应用在线配置优化框架还提供了 RESTful API，令用户可以通过网络对配置优化的流程进行定义设置，用户通过向服务端发送 HTTP POST 请求，服务端会解析 POST 请求中的请求体，以相关的自定义参数。POST 请求体中包含的参数与命令行工具中可设置的选项类似，其中主要的选项名称及其功能如表 4.1 所示。

表 4.1 客户端主要选项

Table 4.1 Client Main Options

选项名称	选项功能
task_name	设置本次配置优化工作的任务名称
benchmark	选择需要进行配置优化的目标微服务应用
algorithm	选择进行配置优化时使用的算法
iteration	设置优化算法的迭代次数
repitition	设置同一配置方案重复测试的次数

RESTful API 服务端的实现方式是，使用 Python 语言和 Flask Web 框架开发了一个长期稳定运行的服务端程序，并让其监听本机的 5000 端口，响应客户端发送过来的 HTTP 请求，其中 HTTP POST 请求体的数据格式如图 4.5 中$v1/microservices$所示。此外，$v1/application$是额外提供的 API，可以用于控制单个软件的配置优化流程，这里的单个软件主要是指微服务应用中所包含的某个软件，如 Redis、MongoDB、Nginx 等。

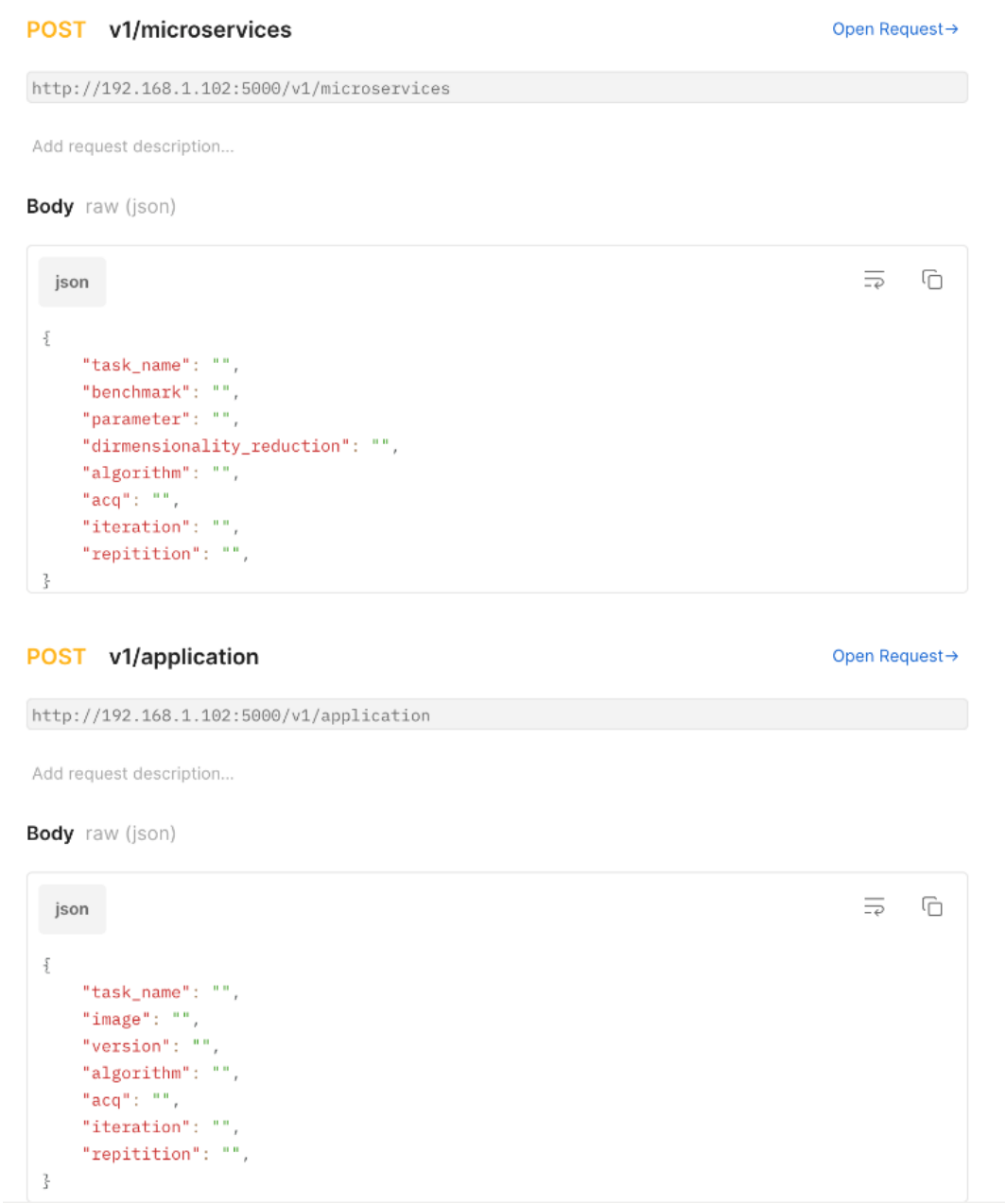

图 4.5 HTTP POST 请求数据格式

Figure 4.5 HTTP POST Request Data Format

4.3.2 优化算法模块实现

云原生微服务应用在线配置优化框架的优化算法模块实现了常用的优化算法，比如贝叶斯优化、模拟退火以及随机算法等。每种优化算法都继承自优化算法抽象类，这个抽象类定义了一些抽象方法和属性，帮助用户轻松地实现自定义的优化算法。用户只需继承该抽象类并实现必需的方法，即可完成自己的优化算法。如图 4.6 所示，自定义算法需要维护一个$pbounds$属性，用于记录待调整的参数名和每个参数可调整的范

围。此外，用户还需要实现$add_observation(conf, perf)$和$next_conf()$方法，前者用于将当前参数配置方案$conf$以及微服务应用对应的性能$perf$注册到优化算法实例中，后者用于从优化算法实例中获取下一套参数配置。

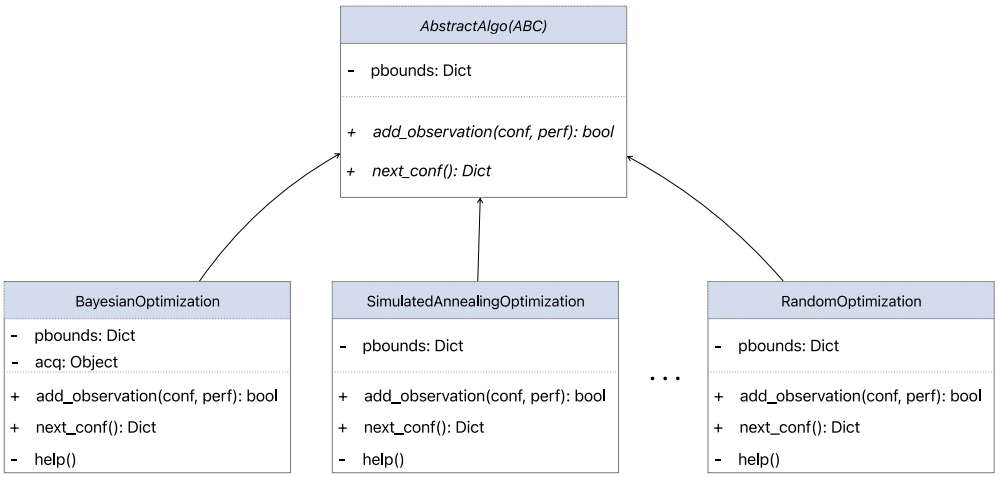

图 4.6 优化算法类图

Figure 4.6 Optimization Algorithm Class Diagram

4.3.3 配置优化流程控制模块实现

配置优化流程控制模块首先根据用户选择的优化算法和目标微服务应用获取算法实例及微服务应用实例。算法实例会根据用户对配置参数类型的选择，设置待优化参数的名称和取值范围，然后配置优化流程控制模块循环执行以下四个步骤，直到达到用户设置的算法迭代次数：

（1） 从算法实例中获取新的参数配置，并将其转化为微服务应用可读取的格式。

（2） 用新获取的参数配置方案运行微服务应用实例，并执行负载测试。

（3） 解析当前参数配置下微服务应用实例的性能指标。

（4） 更新算法实例，即将当前参数配置方案及其性能指标输入算法实例。

如算法 4.1 所示，为了提高当前参数配置下微服务应用性能测试结果的准确性，配置优化流程控制模块会以同一套参数配置方案多次运行并测试微服务应用实例，每一次测试都会解析到该参数配置对应的性能指标，多次测试完成后会得到同一参数配置下的性能指标列表，随后去除性能指标列表中的异常点，计算剩余结果的平均值，以此作为当前参数配置方案下微服务应用的性能指标。

算法 4.1：配置优化流程控制模块执行步骤

输入：客户端传递的选项设置$options$，选用的算法名称$algoName$，待调整的目标微服务应用名称$benchName$

输出：最优参数配置方案$conf$

1：根据$algoName$和$benchName$获取算法实例$algo$和微服务应用实例$bench$
2：创建结果文件夹$result_dir$用于存储优化过程的元数据
3：**for** $iterID = 1$ **to** $options.iteration$ **do**
4：　　**if** $iterID = 1$ **then**
5：　　获取默认参数配置$conf \leftarrow get_default_conf()$
6：　　**else**
7：　　从优化算法实例中获取下一个参数配置$conf \leftarrow algo.next_conf()$
8：　　**end if**
9：　　将当前参数配置记录到$result_dir$中
10：　　初始化性能指标列表$perf_list \leftarrow \emptyset$
11：　　**for** $repID = 1$ **to** $options.repetition$ **do**
12：　　以当前参数配置运行微服务应用并执行压力测试$bench.run(conf)$
13：　　解析微服务应用的性能指标$perf \leftarrow bench.parse_perf()$
14：　　将当前性能指标$perf$加入$perf_list$
15：　　**end for**
16：　　处理$perf_list$去除异常值后求其均值$perf' \leftarrow process(perf_list)$
17：　　更新算法实例$algo.add_observation(perf', conf)$
18：**end for**

4.3.4 负载生成模块实现

负载生成模块针对不同的微服务应用，选择适合该微服务应用的负载测试工具，来模拟多个用户并发的对服务器发送请求，以测试微服务应用的性能表现。可用的负载测试工具有 wrk2、locust 和 ab。

在进行负载测试之前，需要确认目标服务的地址和端口号。以 Kubernetes 为例，首先需要创建 Deployment 资源对象，它会创建一个或多个 Pod，并运行目标服务的容器。然后创建一个 Service 资源对象，指定它需要连接到的 Pod 或者 Deployment 的标签，同时指定 Service 的类型和端口号。图 4.7 是一个微服务应用中前端 Service 资源的配置文件，在该配置文件中，Service 的类型设置为了 NodePort，并且设置集群中的 30080 端口代理到 Pod 的 8080 端口，Pod 的 8080 端口代理到容器的 8080 端口，因此可以通过

Kubernetes 集群中任意一台主机 IP 地址的 8080 端口访问到前端服务。在 Docker Swarm 中，也可以采用类似的方式实现对服务端口的暴露。

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-thrift
  labels:
    death-star-project: social-network
    app: nginx-thrift
  namespace: social-network
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 30080
  selector:
    death-star-project: social-network
    app: nginx-thrift
```

图 4.7 Kubernetes 服务端口设置

Figure 4.7 Kubernetes Service Port Settings

目标服务的地址和端口号暴露之后，可以通过负载测试工具对该地址的对应端口发送负载。以 wrk2 为例，其用法如图 4.8 所示，可以在 wrk 命令的最后指定 URL，即目标服务的地址和端口号。wrk 的主要参数有：-c 设置 HTTP 连接数；-d 设置负载测试持续时间；-t 设置测试持续时长；-R 设置每个线程每秒完成的请求数。

```
master@master-ThinkStation-K:~$ wrk
Usage: wrk <options> <url>
  Options:
    -c, --connections <N>  Connections to keep open
    -D, --dist        <S>  fixed, exp, norm, zipf
    -P                     Print each request's latency
    -p                     Print 99th latency every 0.2s to file
    -d, --duration    <T>  Duration of test
    -t, --threads     <N>  Number of threads to use

    -s, --script      <S>  Load Lua script file
    -H, --header      <H>  Add header to request
    -L  --latency          Print latency statistics
    -T  --timeout     <T>  Socket/request timeout
    -B, --batch_latency    Measure latency of whole
                           batches of pipelined ops
                           (as opposed to each op)
    -v, --version          Print version details
    -R, --rate        <T>  work rate (throughput)
                           in requests/sec (total)
                           [Required Parameter]


  Numeric arguments may include a SI unit (1k, 1M, 1G)
  Time arguments may include a time unit (2s, 2m, 2h)
```

图 4.8 wrk2 帮助手册

Figure 4.8 wrk2 Manual

4.3.5 微服务应用控制模块实现

微服务应用控制模块是配置优化框架中至关重要的一个模块。该模块负责管理微服务应用的整个生命周期，包括部署和删除，同时利用负载生成模块进行性能测试。该模块使用 Ansible 结合其它云原生相关技术进行实现，屏蔽了底层 Docker Swarm 和 Kubernetes 不同的容器编排方案的差异。

Ansible 是一种自动化工具，其架构如图 4.9 所示. Ansible 可以用来自动化管理 IT 基础设施，包括计算机、服务器、网络设备和存储设备等。它使用简单易懂的语法，允许用户通过 SSH 协议在多个主机之间进行任务协调和配置管理，而不需要编写复杂的脚本。本文用 Ansible Playbook 控制微服务应用生命周期的各个阶段，Ansible Playbook 是具有良好组织单元的脚本，用于定义自动化工具 Ansible 所要执行的任务流程。

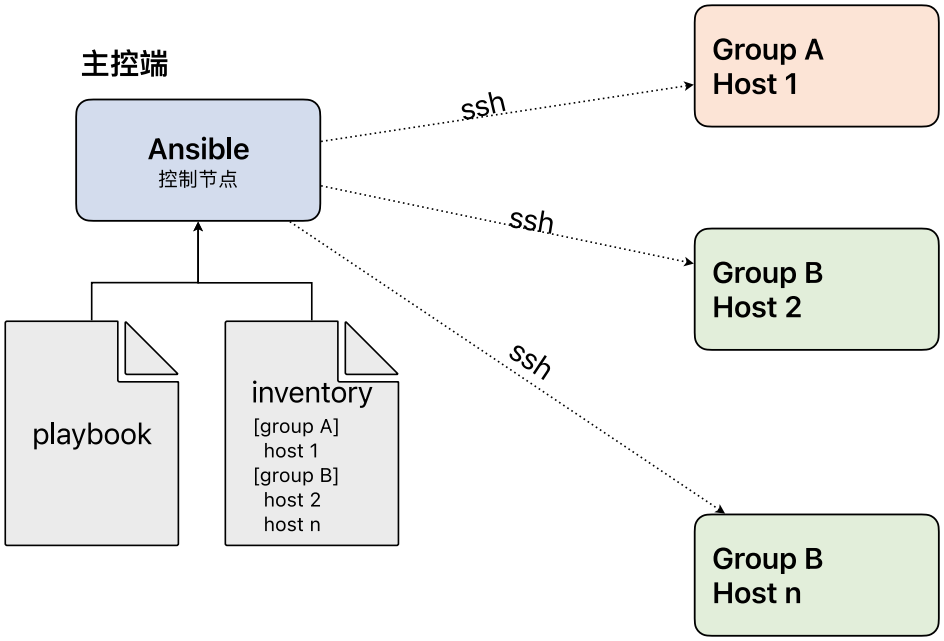

图 4.9 Ansible 架构

Figure 4.9 Ansible Architecture

在使用 Ansible 进行自动化管理时，首先需要说明受控的机器清单。在云原生微服务应用在线配置调优框架的实现中，需要对三台物理主机进行控制，其中包括 master 和 node 两类机器，如图 4.10 所示。

```
0 [node]
1 node1-ThinkStation-K    ansible_connection=ssh  ansible_user=node1
2 node2-ThinkStation-K    ansible_connection=ssh  ansible_user=node2
3
4 [master]
5 master-ThinkStation-K   ansible_connection=ssh  ansible_user=master
~
```

图 4.10 Ansible 主机列表

Figure 4.10 Ansible Iventory

在确定需要控制的机器节点后，使用 Ansible Playbook 编排每台受控机器上需要运行的任务。以使用 Docker Swarm 作为容器编排工具时为例，需要在 Ansible Playbook 中执行以下任务：

（1）获取当前参数配置文件，并将每个参数及其取值读取为环境变量。

（2）在每台受控机器上为 Docker Swarm 里的 Service 准备需要的挂载路径及文件。

（3）使用 Ansible 的 Template Module 根据参数环境变量为每个软件生成相应的配置文件，然后更新 Docker Swarm 部署文件。

（4）根据新生成的 Docker Swarm 部署文件及软件配置文件使用 Ansible Docker Stack Module 部署微服务应用。

（5）解析微服务应用的前端地址及端口，利用负载生成模块执行微服务应用的压力测试。

（6）将压力测试产生的结果信息进行归档，以供后续的结果分析。

（7）使用 Ansible Docker Stack Module 清理 Docker Swarm 集群中的微服务应用。

此外，针对以 Kubernetes 作为容器编排方案的微服务应用，本文使用 ArgoCD 实现对微服务应用的参数配置更新。其实现方式如图 4.11 所示。ArgoCD 通过 GitOps 的方式，监测 Git 仓库中的部署文件，一旦部署文件有更新，就会自动将更新推送到相关的微服务应用中。具体的在本实现中，使用算法实例生成的参数配置，替换 Helm Charts 中的相应字段并提交到代码仓库，ArgoCD 会定期检查代码仓库中的 Helm Charts 文件是否有更新，然后将更新应用到相应的微服务应用中。

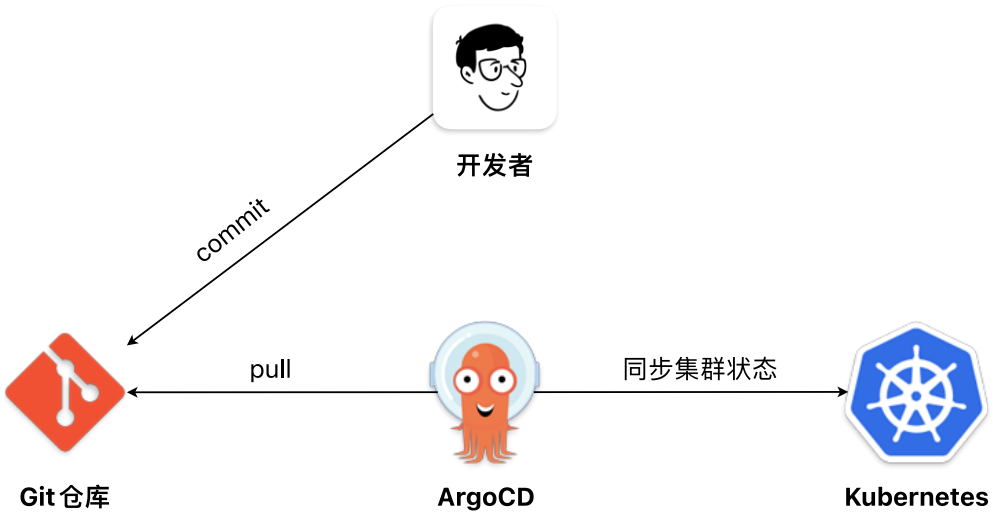

图 4.11 ArgoCD 工作流程

Figure 4.11 ArgoCD Workflow

4.3.6 监控模块实现

监控模块的实现分为微服务应用监控和系统资源监控两个方面。

对于微服务应用监控，本文采用 Uber 开源的分布式链路追踪工具 Jaeger 来收集微服务运行时的调用链路信息。为了实现这一目的，我们使用 jaeger-all-in-one 镜像部署了 Jaeger 项目的三个主要组件：Jaeger 代理、Jaeger 收集器和查询页面。具体来说，Jaeger

代理是一个守护程序，能够监听微服务应用中各服务通过 UDP 发送的数据，从而生成调用链路信息。Jaeger 收集器会检查 Jaeger 代理监听的数据，然后进行处理并持久化存储到数据库中。查询页面提供给用户一个方便的展示页面，供用户查询所需要的信息。

在系统资源监控方面，首先在三台物理主机上安装了 Node Exporter 并作为守护进程长期运行，用于采集主机系统资源使用情况，相应的指标主要从操作系统的/proc 和/sys 文件目录下采集。然后安装并配置 Prometheus Server 对三台物理主机系统信息时序数据进行抓取，其配置如图 4.12 所示。最后容器化部署了 Grafana 用来消费 Prometheus 存储的数据以大盘的方式展示监控数据。

```
19 # A scrape configuration containing exactly one endpoint to scrape:
20 # Here it's Prometheus itself.
21 scrape_configs:
22   # The job name is added as a label `job=<job_name>` to any timeseries scraped
23   - job_name: "master"
24     # metrics_path defaults to '/metrics'
25     # scheme defaults to 'http'.
26     static_configs:
27       - targets: ["192.168.1.102:9100"]
28   - job_name: "node1"
29     # metrics_path defaults to '/metrics'
30     # scheme defaults to 'http'.
31     static_configs:
32       - targets: ["192.168.1.100:9100"]
33   - job_name: "node2"
34     # metrics_path defaults to '/metrics'
35     # scheme defaults to 'http'.
36     static_configs:
37       - targets: ["192.168.1.101:9100"]
```

图 4.12 Prometheus 配置文件

Figure 4.12 Prometheus Configuration File

4.4 框架部署与测试

4.4.1 框架部署

云原生微服务应用在线配置优化框架主要使用 Python 开发，PDM 管理 Python 相关依赖，底层基于容器编排工具和自动化工具。该框架部署在本地三台物理主机上，每台主机的 CPU 型号为拥有 16 个核心的 Intel i7-10700，并且每台主机配备了 16GB DDR4 内存和 1TB 机械硬盘，主机之间通过局域网链接。每台主机的 IP 信息、操作系统版本信息以及所部署的组件信息如表 4.2 所示。

表 4.2 主机节点信息

Table 4.2 Host Node Information

主机名称	局域网 IP	操作系统	部署组件
master-ThinkStation-K	192.168.1.102	Ubuntu 18.04.6	kubernetes-control-plane docker-swarm-manager ansible-control-node node-exporter Prometheus Grafana
node1-ThinkStation-K	192.168.1.100	Ubuntu 18.04.6	kubernetes-worker-node docker-swarm-worker ansible-managed-node
node2-ThinkStation-K	192.168.1.101	Ubuntu 18.04.6	kubernetes-worker-node docker-swarm-worker ansible-managed-node

实现配置优化框架所使用的 Kubernetes、Docker Swarm、Ansible 这些基础软件的运行状态如图 4.13 所示。

```
master@master-ThinkStation-K:~$ ansible all --list
  hosts (3):
    node1-ThinkStation-K
    node2-ThinkStation-K
    master-ThinkStation-K
master@master-ThinkStation-K:~$ docker node ls
ID                            HOSTNAME                STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
midqts4n3nge21by68umgmlf6 *   master-ThinkStation-K   Ready     Active         Leader           20.10.14
mhe858hv2hvq7dcjaf1c8p17k     node1-ThinkStation-K    Ready     Active                          20.10.14
munbhuqed3xmzutyzspr5tpgq     node2-ThinkStation-K    Ready     Active                          20.10.14
master@master-ThinkStation-K:~$ kubectl get node
NAME                  STATUS   ROLES                  AGE    VERSION
master-thinkstation-k   Ready    control-plane,master   325d   v1.23.5
node1-thinkstation-k    Ready    <none>                 325d   v1.23.5
node2-thinkstation-k    Ready    <none>                 325d   v1.23.5
```

图 4.13 软件运行状态

Figure 4.13 Software Running State

4.4.2 框架测试

本节测试了云原生微服务应用在线配置优化框架的主要模块和执行流程。

客户端模块有两种实现方式，分别是命令行工具和 RESTful API。我们对这两种实现方式分别进行了测试。图 4.14 展示了命令行工具 tunectl 的测试结果，该工具能够成功依照客户端设置的选项发起微服务应用配置优化，在运行过程中 Task1_0、Task1_3 和 Task3_3 在应用初始化时存在请求失败的现象，配置优化流程控制模块能够正确识别该情况，并根据预定的策略对性能测试结果实施惩罚机制。Task4_2 中产生异常，配置

优化流程控制模块能够从异常中恢复，并继续执行接下来的配置优化流程。

```
(tuning_microservice-3.9) master@master-ThinkStation-K:~/tuning_microservice$ tunectl -tn sn_both_cs_bo_800_2 -b sn -re 4 -it 40 -para both -dr critical_services
Start tuning social_network microservice with bayesian optimization algorithm...
Task 0_0 start, please wait several minutes to generate result...
Task 0_1 start, please wait several minutes to generate result...
Task 0_2 start, please wait several minutes to generate result...
Task 0_3 start, please wait several minutes to generate result...
Task 0 finished.
Task 1_0 start, please wait several minutes to generate result...
find failed while init social graph, add 20ms to latency as a penalty mechanism
Task 1_1 start, please wait several minutes to generate result...
Task 1_2 start, please wait several minutes to generate result...
Task 1_3 start, please wait several minutes to generate result...
find failed while init social graph, add 20ms to latency as a penalty mechanism
Task 1 finished.
Task 2_0 start, please wait several minutes to generate result...
Task 2_1 start, please wait several minutes to generate result...
Task 2_2 start, please wait several minutes to generate result...
Task 2_3 start, please wait several minutes to generate result...
Task 2 finished.
Task 3_0 start, please wait several minutes to generate result...
Task 3_1 start, please wait several minutes to generate result...
Exception: Social network benchmark run failed
Task 3_2 start, please wait several minutes to generate result...
Task 3_3 start, please wait several minutes to generate result...
find failed while init social graph, add 20ms to latency as a penalty mechanism
Task 3 finished.
Task 4_0 start, please wait several minutes to generate result...
Task 4_1 start, please wait several minutes to generate result...
Task 4_2 start, please wait several minutes to generate result...
Exception: Social network benchmark run failed
Task 4_3 start, please wait several minutes to generate result...
Task 4 finished.
Task 5_0 start, please wait several minutes to generate result...
```

图 4.14 命令行工具测试

Figure 4.14 Command Line Tool Testing

配置优化框架的命令行工具 tunectl 的测试结果中，除了上文中提及的 Task，其余 Task 均正常执行，以使用 Docker Swarm 作为容器编排方案的微服务应用为例，Task 正常执行的流程如图 4.15 所示。

```
PLAY [prepare files] ***********************************************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************************************
ok: [node1-ThinkStation-K]
ok: [master-ThinkStation-K]
ok: [node2-ThinkStation-K]

TASK [check if .tmp exists] ****************************************************************************************************************************************************
ok: [master-ThinkStation-K]
ok: [node1-ThinkStation-K]
ok: [node2-ThinkStation-K]

TASK [copy deploy files] *******************************************************************************************************************************************************
skipping: [node1-ThinkStation-K] => (item=~/tuning_microservice/deploy/social_network/ansible/templates)
skipping: [node1-ThinkStation-K] => (item=~/tuning_microservice/deploy/social_network/docker-swarm/volumes/)

TASK [generate config from template] *******************************************************************************************************************************************
changed: [node1-ThinkStation-K] => (item={'src': 'swarm.yml.j2', 'dest': 'swarm.yml'})
changed: [node2-ThinkStation-K] => (item={'src': 'swarm.yml.j2', 'dest': 'swarm.yml'})
changed: [master-ThinkStation-K] => (item={'src': 'swarm.yml.j2', 'dest': 'swarm.yml'})
PLAY [socialnetwork benchmark] *************************************************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************************************************
ok: [master-ThinkStation-K]

TASK [deploy socialnetwork benchmark use docker swarm] *************************************************************************************************************************
changed: [master-ThinkStation-K]

TASK [wait for 5 seconds] ******************************************************************************************************************************************************
ok: [master-ThinkStation-K]

TASK [create output dir] *******************************************************************************************************************************************************
ok: [master-ThinkStation-K]

TASK [initial socialnetwork benchmark] *****************************************************************************************************************************************
changed: [master-ThinkStation-K]

TASK [wait for 10 seconds] *****************************************************************************************************************************************************
ok: [master-ThinkStation-K]

TASK [compose post] ************************************************************************************************************************************************************
changed: [master-ThinkStation-K]

TASK [read user timeline] ******************************************************************************************************************************************************
changed: [master-ThinkStation-K]

TASK [read home timeline] ******************************************************************************************************************************************************
changed: [master-ThinkStation-K]

TASK [wait for compose-post, read-user-timeline, read-home-timeline finish] ****************************************************************************************************
FAILED - RETRYING: [master-ThinkStation-K]: wait for compose-post, read-user-timeline, read-home-timeline finish (20 retries left).
FAILED - RETRYING: [master-ThinkStation-K]: wait for compose-post, read-user-timeline, read-home-timeline finish (19 retries left).
PLAY RECAP *********************************************************************************************************************************************************************
master-ThinkStation-K      : ok=13   changed=7    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
node1-ThinkStation-K       : ok=3    changed=1    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
node2-ThinkStation-K       : ok=3    changed=1    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
```

图 4.15 微服务应用单流程测试

Figure 4.15 Single Process Testing of Microservice Applications

图 4.16 展示了客户端模块使用 RESTful API 实现时的测试结果，用户需要首先向服

务端发送 POST 请求，以对微服务应用配置优化流程设置自定义选项，服务端返回的消息中包含是否成功设置相关信息。在服务端成功处理了用户的 POST 请求后，用户可以再发送 GET 请求，获取微服务应用配置优化过程中，当前能够获得的最优参数配置。

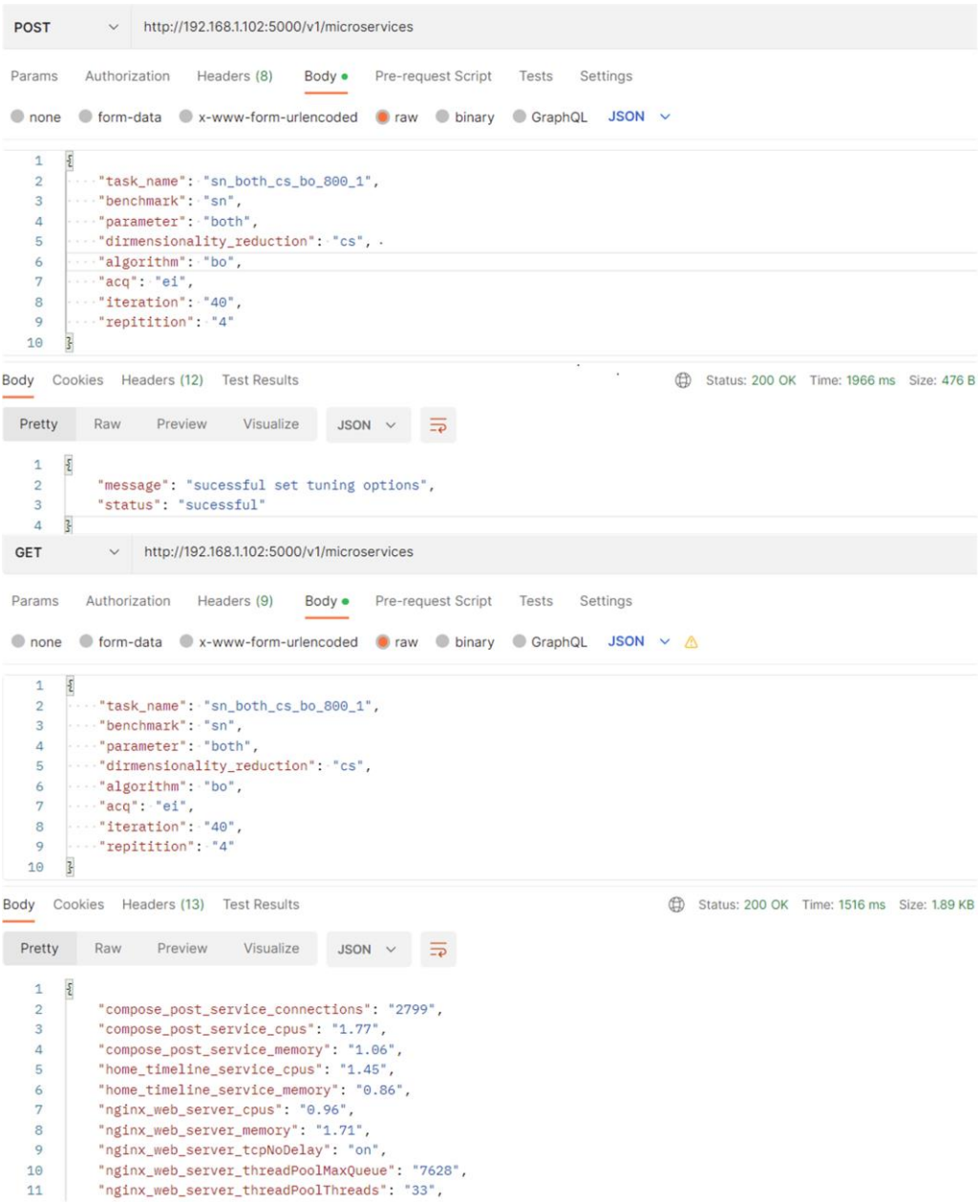

图 4.16 RESTful API 请求测试

Figure 4.16 RESTful API Request Testing

负载生成模块的测试结果如图 4.17 所示。这些结果是通过对微服务应用中的一个调用链路发送负载，然后得到的性能指标。需要注意的是，同一个微服务应用中包含多个调用链路，且配置优化框架也支持不同的微服务应用。然而，由于篇幅限制，本节并未对负载生成模块在不同微服务应用以及不同调用链路下的测试结果进行逐个展示。

```
Running 5m test @ http://192.168.1.100:8080/wrk2-api/post/compose
  1 threads and 4 connections
  Thread calibration: mean lat.: 9.110ms, rate sampling interval: 25ms
  Thread Stats   Avg      Stdev     99%   +/- Stdev
    Latency     9.34ms    2.79ms  20.53ms   86.27%
    Req/Sec    50.69     44.95   166.00     58.52%
  Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%    8.58ms
 75.000%   10.03ms
 90.000%   12.40ms
 99.000%   20.53ms
 99.900%   28.61ms
 99.990%   35.33ms
 99.999%   40.42ms
100.000%   40.42ms
```

图 4.17 负载生成模块测试

Figure 4.17 Load Generation Module Testing

监控模块分为微服务应用监控和系统资源监控两个部分。图4.18展示了微服务应用调用链路的监控情况，其中包括调用链路端到端延迟分布以及各服务自身的延迟指标。

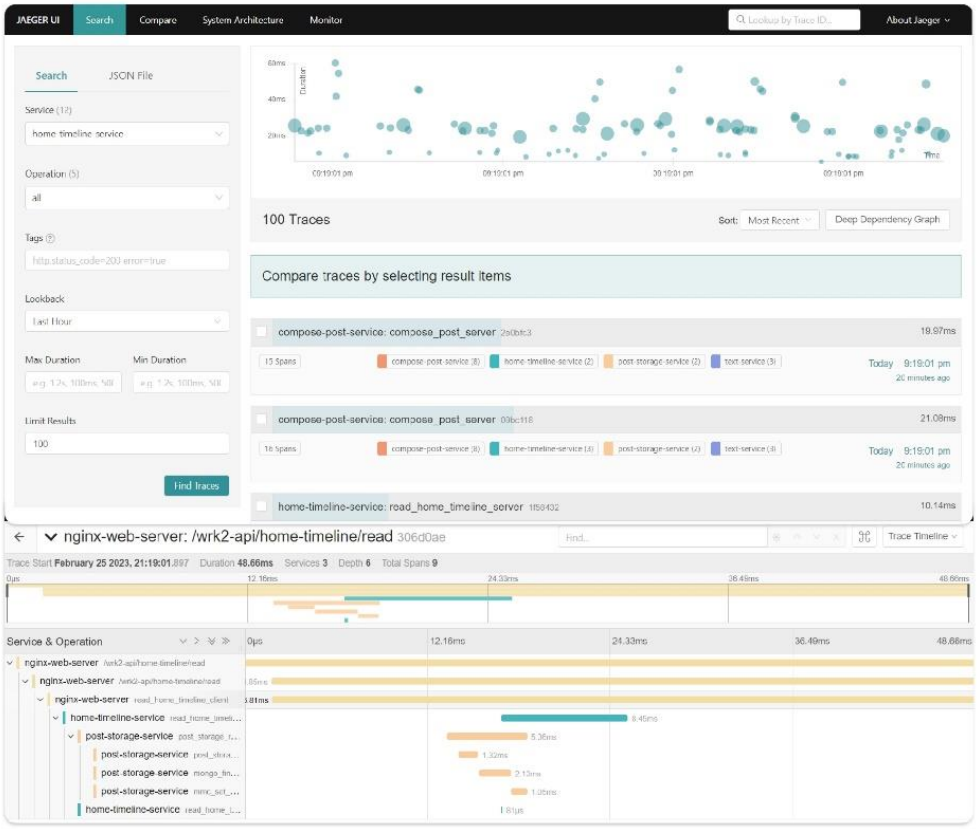

图 4.18 微服务应用监控

Figure 4.18 Microservice Application Monitoring

系统资源监控是监控模块的另一个重要组成部分，图4.19展示了系统资源监控的测

试结果，从图中可以看出在配置优化框架运行期间，系统资源的用量呈周期性变化趋势，这是因为在实验过程中会周期性的对微服务应用进行重新部署与测试。需要注意的是图 4.19 只展示了 Master 节点的资源使用情况，其它两个工作节点资源用量变化与该图类似。

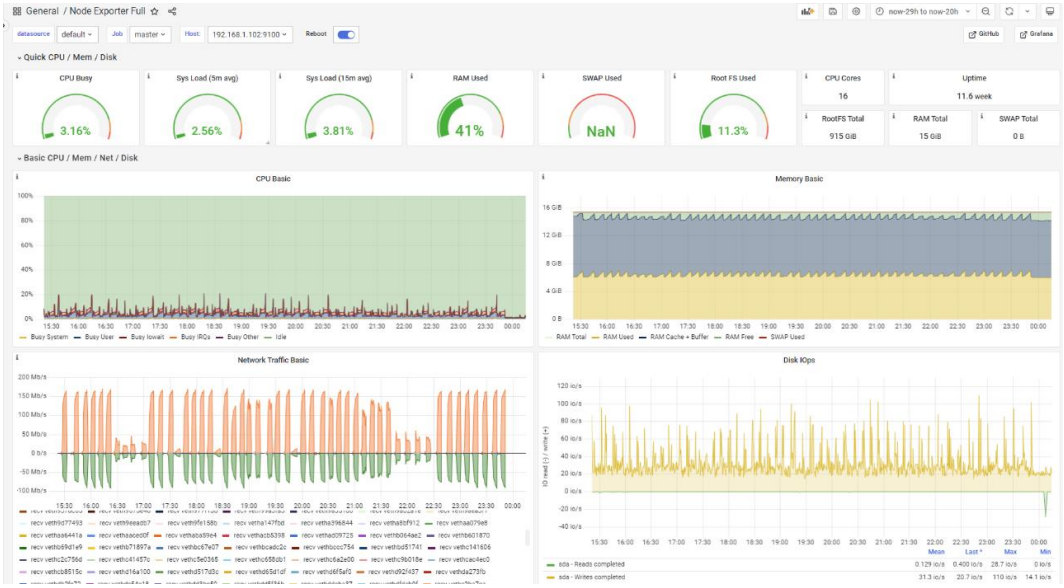

图 4.19 系统资源监控

Figure 4.19 System Resource Monitoring

4.5 本章小结

本章设计并实现了云原生微服务应用在线配置优化框架，以解决微服务场景下手工操作配置优化流程困难的问题。本章首先根据微服务应用和配置优化工作的特点进行了需求分析。接下来，从多个角度对配置优化框架进行了设计，包括整体架构、数据流向以及层级结构。然后，详细介绍了配置优化框架中每个模块的实现。最后在三台物理主机上部署并测试了该框架，结果表明该框架能够很好地完成自动化微服务应用配置优化的工作。

总结与展望

随着人工智能、大数据、云计算和5G等技术快速发展以及近几年新冠疫情的影响，推动了企业和整个社会的数字化转型，其中云计算是数字化转型的支撑力量，而云原生是一种基于云计算的软件开发和部署范式。在云原生环境中，微服务架构被广泛采用。为了提高微服务应用的性能，必须采取有效的手段对应用中存在的可调节参数进行调整。针对云原生环境下微服务应用配置优化问题，本文主要做了以下工作：

（1）为了克服微服务应用配置优化问题中仅调整资源配置或软件参数，无法充分探索配置优化空间的局限性，本文提出了一种协同优化方案，同时优化资源配置与软件参数。为了解决该方案存在的参数之间依赖复杂、配置参数与性能之间关系非线性以及高维度参数搜索空间的问题，本文在第三章提出了相应的解决方法，并且在本地集群上的实验证明，本文提出的方案相比目前已有的方案能够为微服务应用带来更大的性能提升。此外，第三章实验中还对资源能效进行了定义，并以它为优化目标，实验结果表明，该目标能够提高系统资源利用率。

（2）为了解决手动操控微服务应用配置优化流程困难，难以实施的问题，第四章设计并实现了云原生微服务应用在线配置优化框架，该框架利用云原生技术栈，避免了系统管理人员手动对微服务应用进行参数调整，从而使云原生环境下微服务应用配置优化流程更加简便和高效。

微服务应用的配置优化问题已经成为了云原生环境中的一个重要研究方向。在本文研究的基础上，未来的工作可以从以下几个方面对该问题进行深入探究和优化：

多目标优化问题：多目标优化可以更好地表示系统资源持有成本与微服务应用性能之间的关系。因此，在今后的工作中，可以考虑将微服务应用配置优化问题形式化为多目标优化问题，以更好地实现微服务应用性能与系统资源持有成本的权衡；

负载预测与配置迁移问题：在微服务应用中，负载的变化会对性能产生显著影响，因此对负载的预测和配置迁移也是一个重要的研究方向。未来的工作可以进一步探索基于机器学习的负载预测方法，并设计相应的配置迁移算法；

配置优化框架安全性问题：在配置优化过程中可能会涉及到某些敏感数据，需要采取相应的数据隐私保护措施。在本文基础上可以为配置优化框架开发安全审计和日志监控机制，以记录和分析用户的操作行为和系统事件，并及时发现和应对异常情况。

参考文献

[1] Gannon D, Barga R, Sundaresan N. Cloud-native applications[J]. IEEE Cloud Computing, 2017, 4(5): 16-21.

[2] 曾德泽, 陈律昊, 顾琳, 等. 云原生边缘计算: 探索与展望[J]. 物联网学报, 2021, 5(2): 7-17.

[3] 李铭轩, 童俊杰, 刘秋妍. 基于云原生的 5G 核心网演进解决方案研究[J]. 信息通信技术, 2020, 14(1): 63-69.

[4] CNCF. CNCF Annual Survey 2021[EB/OL], https://www.cncf.io/reports/cncf-annual-survey-2021, 2022-02-10/2023-01-02.

[5] Buzato FH, Goldman A, Batista D. Efficient resources utilization by different microservices deployment models[A]. 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)[C]. Cambridge: IEEE, 2018: 1-4.

[6] Prakash C, Prashanth P, Bellur U, et al. Deterministic container resource management in derivative clouds[A]. IEEE International Conference on Cloud Engineering (IC2E)[C]. Orlando: IEEE, 2018: 79-89.

[7] Heinrich R, Van Hoorn A, Knoche H, et al. Performance engineering for microservices: research challenges and directions[A]. 8th ACM/SPEC on International Conference on Performance Engineering Companion[C]. L'AQUILA: ACM, 2017: 223-226.

[8] Al-Debagy O, Martinek P. A Comparative Review of Microservices and Monolithic Architectures[A]. IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)[C]. Budapest: IEEE, 2018: 149-154.

[9] Hasselbring W, Steinacker G. Microservice architectures for scalability, agility and reliability in e-commerce[A]. IEEE International Conference on Software Architecture Workshops[C]. Gothenburg: IEEE, 2017: 243-246.

[10] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[A]. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)[C]. Incline Village: IEEE, 2010: 1-10.

[11] Zaharia M, Chowdhury M, Franklin MJ, et al. Spark: Cluster computing with working

sets[A]. 2nd USENIX Conference on Hot Topics in Cloud Computing[C]. Boston: USENIX Association, 2010: 10-10.

[12] Carbone P, Katsifodimos A, Ewen S, et al. Apache flink: Stream and batch processing in a single engine[J]. The Bulletin of the Technical Committee on Data Engineering, 2015, 36(4): 28-38.

[13] Mathiya BJ, Desai VL. Apache hadoop yarn parameter configuration challenges and optimization[A]. 2015 International Conference on Soft-Computing and Networks Security (ICSNS)[C]. Coimbatore: IEEE, 2015: 1-6.

[14] Herodotou H, Lim H, Luo G, et al. Starfish: A Self-tuning System for Big Data Analytics[A]. 5th Biennial Conference on Innovative Data Systems Research(CIDR)[C]. Asilomar: Online Proceeding, 2011: 261-272.

[15] Herodotou H, Dong F, Babu S. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics[A]. 2nd ACM Symposium on Cloud Computing[C]. Cascais: ACM, 2011: 1-14.

[16] Liao G, Datta K, Willke TL. Gunther: Search-based auto-tuning of mapreduce[A]. Euro-Par 2013 Parallel Processing: 19th International Conference[C]. Aachen: Springer, 2013: 406-419.

[17] Petridis P, Gounaris A, Torres J. Spark parameter tuning via trial-and-error[A]. Advances in Big Data: 2nd INNS Conference on Big Data[C]. Thessaloniki: Springer, 2017: 226-237.

[18] Yu Z, Bei Z, Qian X. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing[A]. 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)[C]. Williamsburg: ACM, 2018: 564–577.

[19] Li M, Liu Z, Shi X, et al. ATCS: Auto-tuning configurations of big data frameworks based on generative adversarial nets[J]. IEEE Access, 2020, 8: 50485-50496.

[20] Fekry A, Carata L, Pasquier T, et al. To tune or not to tune? in search of optimal configurations for data analytics[A]. Proceedings of the 26th ACM SIGKDD International

Conference on Knowledge Discovery & Data Mining[C]. Virtual Event: ACM, 2020: 2494-2504.

[21] Guo Y, Shan H, Huang S, et al. GML: Efficiently Auto-Tuning Flink's Configurations Via Guided Machine Learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(12): 2921-2935.

[22] HoseinyFarahabady MR, Jannesari A, Taheri J, et al. Q-flink: A qos-aware controller for apache flink[A]. 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)[C]. MelbourneL: IEEE, 2020: 629-638.

[23] Van Aken D, Pavlo A, Gordon GJ, et al. Automatic database management system tuning through large-scale machine learning[A]. 2017 ACM International Conference on Management of Data[C]. Chicago: CSI, 2017: 1009-1024.

[24] Cereda S, Valladares S, Cremonesi P, et al. Cgptuner: a contextual gaussian process bandit approach for the automatic tuning of it configurations under varying workload conditions[J]. VLDB Endowment, 2021, 14(8): 1401-1413.

[25] Kanellis K, Alagappan R, Venkataraman S. Too many knobs to tune? towards faster database tuning by pre-selecting important knobs[A]. Workshop on Hot Topics in Storage and File Systems[C]. Virtual Event: USENIX Association, 2020: 1-1.

[26] Reiss C, Tumanov A, Ganger GR, et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis[A]. third ACM symposium on cloud computing(SOCC)[C]. San Jose: ACM, 2012: 1-13.

[27] Wilkes J, Reiss C. Details of the ClusterData-2011-1 trace 2011[EB/OL], https://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1, 2011-01-01/2023-01-02.

[28] Jyothi SA, Curino C, Menache I, et al. Morpheus: Towards Automated SLOs for Enterprise Clusters[A]. 12th USENIX Symposium on Operating Systems Design and Implementation(OSDI)[C]. Savannah: USENIX Association, 2016: 117-134.

[29] Gias AU, Casale G, Woodside M. ATOM: Model-driven autoscaling for microservices[A]. 39th International Conference on Distributed Computing Systems (ICDCS)[C]. Dallas:

IEEE, 2019: 1994-2004.

[30] Kubernetes Documention. How does a Horizontal Pod Autoscaler work[EB/OL], https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#how-does-a-horizontalpodautoscaler-work, 2022-11-26/2023-01-02.

[31] Google Kubernetes Engine Documentation. How vertical Pod autoscaling works[EB/OL], https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler, 2022-12-27/2023-01-02.

[32] Qiu H, Banerjee SS, Jha S, et al. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices[A]. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)[C]. Virtual Event: USENIX Association, 2020: 805-825.

[33] Kubernetes Documention. How Pods manage multiple containers[EB/OL], https://kubernetes.io/docs/concepts/workloads/pods/#what-is-a-pod, 2022-12-15/2023-01-02.

[34] Rzadca K, Findeisen P, Swiderski J, et al. Autopilot: workload autoscaling at Google[A]. Fifteenth European Conference on Computer Systems(EuroSys)[C]. Heraklion: ACM, 2020: 1-16.

[35] Zhang Y, Hua W, Zhou Z, et al. Sinan: ML-based and QoS-aware resource management for cloud microservices[A]. 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems[C]. Virtual Event: ACM, 2021: 167-181.

[36] Mahgoub A, Medoff A, Kumar R, et al. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud[A]. 2020 USENIX Conference on Usenix Annual Technical Conference[C]. Virtual Event:USENIX Association, 2020: 189-204.

[37] Wang R, Wang Q, Hu Y, et al. Industry practice of configuration auto-tuning for cloud applications and services[A]. 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering[C]. Singapore: ACM, 2022: 1555-1565.

[38] OpenAnolis SIG. KeenTune[EB/OL], https://openanolis.cn/sig/keentune, 2022-11-01/2023-01-03.

[39] Somashekar G, Gandhi A. Towards optimal configuration of microservices[A]. 1st Workshop on Machine Learning and Systems[C]. Virtual Event: ACM, 2021: 7-14.

[40] Somashekar G, Suresh A, Tyagi S, et al. Reducing the Tail Latency of Microservices Applications via Optimal Configuration Tuning[A]. 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)[C]. California: IEEE, 2022: 111-120.

[41] Burns B, Grant B, Oppenheimer D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-7.

[42] Leffler G. {OpenTelemetry} and Observability: What, Why, and Why Now?[A]. Usenix SREcon22 Asia/Pacific[C]. Sydney: USENIX Association, 2022: 1-1.

[43] gRPC Documentaion. What is gRPC[EB/OL], https://grpc.io/docs/what-is-grpc, 2020-12-16/2023-01-05.

[44] Gao X, Gu Z, Li Z, et al. Houdini's escape: Breaking the resource rein of linux control groups[A]. 2019 ACM SIGSAC Conference on Computer and Communications Security[C]. London: ACM, 2019: 1073-1086.

[45] Jian Z, Chen L. A defense method against docker escape attack[A]. 2017 International Conference on Cryptography, Security and Privacy[C]. Wuhan: ACM, 2017: 142-146.

[46] Docker Documentation. Docker storage drivers[EB/OL], https://docs.docker.com/storage/storagedriver/select-storage-driver, 2022-11-24/2023-01-09.

[47] The Linux Kernel Documentation. How are cgroups implemented[EB/OL], https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, 2022-9-21/2023-01-09.

[48] Al Jawarneh IM, Bellavista P, Bosi F, et al. Container orchestration engines: A thorough functional and performance comparison[A]. 2019 IEEE International Conference on Communications (ICC)[C]. Shanghai: IEEE, 2019: 1-6.

[49] Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at Google with

Borg[A]. Tenth European Conference on Computer Systems[C]. Bordeaux: ACM, 2015: 1-17.

[50] Rejiba Z, Chamanara J. Custom scheduling in Kubernetes: A survey on common problems and solution approaches[J]. ACM Computing Surveys, 2022, 55(7): 1-37.

[51] Jeremy Cloud. Decomposing twitter: Adventures in service-oriented architecture[EB/OL], https://www.infoq.com/presentations/twitter-soa, 2013-07-03/2023-01-03.

[52] Ponce F, Márquez G, Astudillo H. Migrating from monolithic architecture to microservices: A Rapid Review[A]. 38th International Conference of the Chilean Computer Science Society (SCCC)[C]. Concepcion: IEEE, 2019: 1-7.

[53] Leite L, Rocha C, Kon F, et al. A survey of DevOps concepts and challenges[J]. ACM Computing Surveys (CSUR), 2019, 52(6): 1-35.

[54] Saif Gunja. What is DevOps? Unpacking the purpose and importance of an IT cultural revolution[EB/OL], https://www.dynatrace.com/news/blog/what-is-devops, 2021-07-13/2023-01-14.

[55] Beetz F, Harrer S. GitOps: The Evolution of DevOps[J]. IEEE Software, 2021, 39(4): 70-75.

[56] Gotin M, Lösch F, Heinrich R, et al. Investigating performance metrics for scaling microservices in cloudiot-environments[A]. 2018 ACM/SPEC International Conference on Performance Engineering[C]. Berlin: ACM, 2018: 157-167.

[57] Rahman J, Lama P. Predicting the end-to-end tail latency of containerized microservices in the cloud[A]. 2019 IEEE International Conference on Cloud Engineering (IC2E)[C]. Prague: IEEE, 2019: 200-210.

[58] Yu G, Chen P, Chen H, et al. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments[A]. Web Conference 2021[C]. Ljubljana: ACM, 2021: 3087-3098.

[59] Jia Z, Witchel E. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices[A]. 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems[C]. Virtual

Event: ACM, 2021: 152-166.

[60] Zhang Z, Ramanathan M K, Raj P, et al. {CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures[A]. 2022 USENIX Annual Technical Conference (USENIX ATC 22)[C]. Carlsbad: USENIX Association, 2022: 655-672.

[61] Uber Blog. Evolving Distributed Tracing at Uber Engineering[EB/OL], https://www.uber.com/blog/distributed-tracing, 2017-02-02/2023-01-21.

[62] Cohen I, Huang Y, Chen J, et al. Pearson correlation coefficient[J]. Noise reduction in speech processing, 2009, 2(1): 1-4.

[63] 崔佳旭, 杨博. 贝叶斯优化方法和应用综述[J]. 软件学报, 2018, 29(10): 3068-3090.

[64] Klein A, Falkner S, Bartels S, et al. Fast bayesian optimization of machine learning hyperparameters on large datasets[A]. Proceedings of the 20th International Conference on Artificial Intelligence and Statistics[C]. Fort Lauderdale: PMLR, 2017: 528-536.

[65] Torun H M, Swaminathan M, Davis A K, et al. A global Bayesian optimization algorithm and its application to integrated system design[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2018, 26(4): 792-802.

[66] Li Q, Li B, Mercati P, et al. RAMBO: Resource allocation for microservices using Bayesian optimization[J]. IEEE Computer Architecture Letters, 2021, 20(1): 46-49.

[67] Wikipedia. Gaussian process[EB/OL], https://en.wikipedia.org/wiki/Gaussian_process, 2023-01-09/2023-01-14.

[68] Gan Y, Zhang Y, Cheng D, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems[A]. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems[C]. Providence: ACM, 2019: 3-18.

[69] Moriconi R, Deisenroth M P, Sesh Kumar K S. High-dimensional Bayesian optimization using low-dimensional feature spaces[J]. Machine Learning, 2020, 109(1): 1925-1943.