中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

面向容器编排平台的自动化调优系统研究

作者姓名：王新元

指导教师：赵琛 研究员

中国科学院软件研究所

学位类别：电子信息硕士

学科专业：软件工程

培养单位：中国科学院软件研究所

2024 年 6 月

Research on Automated Optimization System for Container Orchestration Platform

A thesis submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Electronic and Information Engineering

In Software Engineering

By Xinyuan Wang

Supervisor: Professor Chen Zhao

Institute of Software Chinese Academy of Sciences

June 2024

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。承诺除文中已经注明引用的内容外，本论文不包含任何其他个人或集体享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　期：

中国科学院大学
学位论文使用授权声明

本人完全了解并同意遵守中国科学院大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　期：　　日　期：

摘 要

基于云原生的软件开发和部署范式已受到产业界和学术界的广泛重视，云原生通过容器编排平台，如 Kubernetes 等，进行应用程序的部署和维护，旨在提高应用程序的敏捷性、可靠性和可维护性，适应动态变化的业务需求和技术环境。

但在目前容器编排平台的管理和维护中，存在着不合理的参数配置以及资源调度情况。参数配置方面，容器编排平台忽略集群中应用程序参数的显著影响，没有考虑在集群管理过程中调整应用程序相应的参数。同时，传统优化方案依赖于人工经验，容易存在偏差，且难以覆盖多种类型的负载情况和日渐繁多的参数配置。资源调度方面，容器编排平台主要基于单一的指标和简单的阈值判断而决定收缩或扩张资源，忽略被广泛应用于确保基于云原生架构的应用程序和服务的可靠性、性能和可用性的 SLO（Service Level Objective，服务水平目标）指标，且难以处理微服务架构为代表的整体的复杂集群场景。

因此，面向容器编排平台，如何通过资源和参数的优化，在提高整个集群的性能表现的同时降低资源消耗，是当前集群优化中的一个关键问题。为了解决上述问题，本研究工作内容包括：

1. 设计并实现一个面向容器编排平台的容器集群自动化调优系统。该系统架构设计根据容器编排平台 Kubernetes 机制深度定制，由控制模块、调优模块、参数资源管理模块、观测模块共四个功能模块构成。四个模块基于用户声明的需求文件，协同工作，实现自动化调优。

2. 针对参数配置优化，提出了离线-在线结合的识别优化方法，包括基于集成学习的离线辅助训练方法和基于机器学习的在线的参数调优方法。通过构建负载知识库和参数知识库，能够在负载到达集群时，有效利用离线先验知识，准确识别负载标签并进行初始化的参数推荐，同时能够在较优的配置基础上进行快速的迭代优化。

3. 针对资源分配优化，提出资源-参数联合调优方法及资源下降算法，将微服务系统中资源分配策略的优化与集群环境及子应用程序的参数配置优化进行结合，在满足用户自定义的 SLO 前提下，为微服务系统提供基于实时工作负载的资源分配和参数设置的最佳组合，以更低的资源消耗维持原有性能，实现节省资源的目的。或消耗同样的资源，提高系统处理能力，实现提供更高的并发支持能力的目的。

本研究使用四种不同的应用程序（Nginx、Memcached、Redis 和 PostgreSQL）验证了优化系统的有效性。结果表明，本研究实现的自动化调优

系统能够使得应用的平均性能提升达到 46.55%至 208.20%。此外，在具有实际工作负载的微服务架构 HotelReservation 场景中，该优化系统在满足服务水平目标，即保证平均延迟和 P99 延迟以满足用户使用需求的同时，可节省 37.5%以上的资源，或在相同资源情况下支持用户并发数达到默认配置的 3 倍以上。

关键词：云原生，容器编排平台，参数配置，资源分配，自动化调优系统

Abstract

The cloud native software development and deployment paradigm has received widespread attention from the industry and academia. Cloud native deploys and maintains applications through container orchestration platforms such as Kubernetes, aiming to improve the agility, reliability, and maintainability of applications, adapt to dynamic business needs and technological environments. However, there are unreasonable parameter configurations and resource scheduling allocations in the management and maintenance of container orchestration platforms. In terms of parameter configuration, the container orchestration platform ignores the significant impact of application parameters in the cluster, and does not consider adjusting the corresponding parameters of the application in the cluster management process. Meanwhile, traditional solutions rely on manual experience, are prone to bias, and are difficult to cover multiple types of load situations and increasingly diverse parameter configurations. In terms of resource scheduling, container orchestration platforms mainly decide to shrink or expand resources based on a single indicator and simple threshold judgments, ignoring the Service Level Objective (SLO) indicator widely used to ensure the reliability, performance, and availability of applications and services based on cloud native architecture, and making it difficult to handle complex cluster scenarios represented by microservice architecture as a whole.

Therefore, based on the container orchestration platform, how to improve the performance of the entire cluster environment while reducing cost consumption through resource and parameter optimization is a key issue in current cluster performance optimization. To address the aforementioned issues, the scope of this study includes:

1. Design and implement an automated optimization system for container clusters based on a container orchestration platform. The system is deeply customized based on the container orchestration platform Kubernetes mechanism, consisting of four functional modules: control module, tuning module, parameter resource management module, and observation module. Four modules work together to achieve automated tuning.

2. A combination of offline and online recognition optimization methods has been proposed for parameter configuration optimization, including offline auxiliary training methods based on ensemble learning and online parameter tuning methods based on machine learning. By constructing a load knowledge base and a parameter knowledge base, it is possible to effectively utilize offline prior knowledge when the load reaches the cluster, accurately identify and initialize parameters, and perform rapid iterative optimization on the basis of optimal configuration.

3. For resource allocation optimization, a resource parameter joint optimization

method and a resource descent algorithm are proposed. The optimization of resource allocation strategies in microservice systems is combined with the optimization of sub application parameter configuration. Under the premise of meeting user defined SLO, the optimal combination of real-time workload based resource allocation and parameter settings is provided to maintain the original performance with lower resource consumption, or to provide higher concurrency support with the same resource consumption.

This study validated the effectiveness of the optimized system using four different applications: Nginx, Memcached, Redis, and PostgreSQL. The results show that the automated tuning system implemented in this study can improve the average performance of the application by 46.55% to 208.20%. In addition, in the microservice architecture HotelReservation scenario with actual workloads, this optimization system can save more than 37.5% of resources while meeting the service level goal of ensuring average latency and P99 latency to meet user usage needs, or use the same resources to support user concurrency reaching more than three times the default configuration.

Key Words: Cloud native, Container orchestration platform, Parameter configuration, Resource allocation, Automatic optimization system

目 录

图目录

表目录

第 1 章 绪论

本章首先介绍面向容器编排平台的自动化调优系统研究的背景及意义，其次介绍资源分配和参数调优方法的相关工作和研究现状，然后简要阐述本文的主要研究内容，最后介绍本文的总体组织结构。

1.1 研究背景及意义

云原生的体系中，容器编排平台如 Kubernetes（K8s）、YARN、Omega、Mesos[1][2][3]等，作为自动化部署、管理和扩展容器化应用程序的工具，可以帮助用户和开发人员能够独立于底层基础设施，轻松地管理大规模的容器集群，确保应用程序能够高效、可靠地运行。但在这一体系中，由用户指定的应用程序的资源需求和参数配置很难避免地会出现不妥当的问题，这种情况出现在各种容器编排平台的使用当中。其中由云原生计算基金会（CNCF）[4]管理和支持的 Kubernetes 受到最多用户的欢迎，被众多企业应用于实际生产当中。根据 Red Hat 报告[5]，截至 2021 年 Kubernetes 承载了 88% 的容器化工作负载，成为行业的事实基准。因此，本研究后续的展开将面向开源的 Kubernetes 容器编排平台，但研究的设计思路适用于上述各种容器编排平台。

如前所述，容器编排平台帮助用户管理集群，但并不负责为其部署的各种应用的参数配置进行管理，需要进行额外的优化工作。如果参数配置不妥当、或参数优化效果较差，会使得效能偏低，无法满足实际生产需求。如图 1-1 所示，以 Web 应用服务器 Tomcat[12]为例，X 坐标轴是其参数 maxConnections 的取值，Y 坐标轴是其 socketBuffer_Byte 的取值，Z 坐标轴则是不同参数取值情况下，其在网页导航负载上的吞吐率性能表现。可以看出，不同参数取值的最佳表现和最差表现差距可达数十倍。

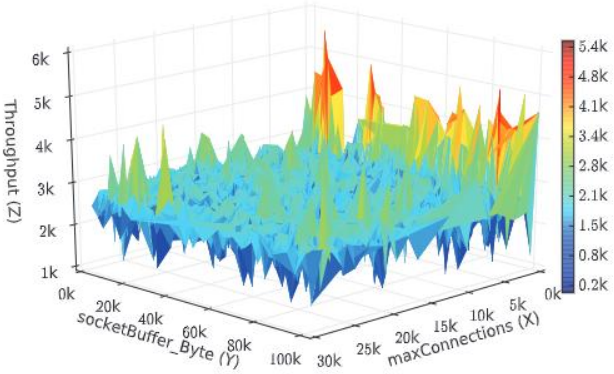

图 1-1 Tomcat 不同配置下的性能表现

Figure 1-1 Performance of Tomcat with different configurations

最近的研究表明性能问题与开源云系统中 50%的配置相关补丁和 30%的与配置相关的论坛问题[6]密切相关，这说明不合理参数配置导致的性能问题普遍存在于云系统中各种应用程序当中。因此，优化应用参数配置以提高云平台资源利用率和系统效率的重要性不言而喻。同时，较为稳定的物理机上的应用适用的参数配置，往往并不适用与云平台上部署的应用。根据研究[7]中针对容器线程数和分配资源大小的研究，云环境中最优的线程配置和资源分配可能与实际负载压力存在较强的负相关性，这与物理机上参数配置的传统经验不符。类似地，图 1-2 展示的是 Nginx 基于主机 CPU 计数的工作流程的自动配置示例[8]。但是在容器环境中，CPU 计数可能与实际资源分配不匹配，从而导致自动配置出现错误，使得系统整体性能下降。

```
1 ...
2 if (ngx_strcmp(value[1].data, "auto") == 0) {
3     ccf->worker_processes = ngx_ncpu;
4     return NGX_CONF_OK;
5 }
6 ...
7
8 ...
9 #if (NGX_HAVE_SC_NPROCESSORS_ONLN)
10    ngx_ncpu = sysconf(_SC_NPROCESSORS_ONLN);
11 #endif
12 ...
```

图 1-2 Nginx 自动配置示例

Figure 1-2 Example of Nginx automatic configuration

此外，随着应用程序的版本更迭，参数数目日渐增多，优化难度极大。Apache Hadoop[9]作为一个开源的大数据平台有大约 400 个参数，由于其巨大的处理能力，在各种组织中广泛使用。Hive 是基于 Hadoop 的数据仓库工具，有大约 1200 个参数，可以存储、查询、分析数据，方便决策人员和数据分析人员统计分析历史数据。Apache Spark[10]也是一个开源的大数据平台，它以分布式方式存储、管理和处理大数据，有大约 180 个参数需要调整。以正确的方式调整这些参数将改善系统的准确性、吞吐量和执行时间，这些参数的调优一定程度上能够依赖人工经验，通过在某个区间内取一个估计值的方式完成调优。但是，考虑到应用程序会面临不同类型的负载情况，人工估计的范围值的准确性难以保证。此外，研究表明并非所有的参数都对这些应用的性能有重大影响，只有少数一部分会明显影响系统的性能，例如，Hadoop 和 Hive 整体共有 109 个参数会对性能产生影响，Spark 则有 30 个参数会对性能产生影响[11]。因此，识别出云集群上部署的不同应用中值得优化的参数同时进行优化，是十分具有必要性的。

另一方面，在资源分配层面，Kubernetes 的默认策略是基于资源请求和限

制来管理容器的资源。Pod 是 Kubernetes 创建和管理的最小部署单元，在其配置文件当中可以指定容器的资源请求和限制，包括 CPU 和内存。资源请求是容器运行所需的最小资源量，而资源限制则是容器能够使用的最大资源量。Kubernetes 会根据这些参数来调度和管理容器的资源分配，确保每个容器都有足够的资源来正常运行，同时避免资源浪费和争用。如果没有显式指定资源请求和限制，Kubernetes 会使用默认值来进行资源分配管理。正是通过设置资源请求和限制，既可以确保应用程序具有足够的资源，又可以确保容器不会使用超过指定数量的资源，从而防止容器消耗过度，避免过分占用资源导致的浪费。但这样的缺陷是：①容易出现不合理分配的情况，造成资源浪费和资源争用可能会影响集群的整体资源利用率；②难以确定合适的限制值，在实践中某些应用程序，需要的资源可能会随着时间和负载的变化而发生变化；③无法做到更细粒度的资源管理，仅能管理 CPU 和内存资源。

同时，Kubernetes 实现两种资源管理策略用于控制 Pod 的资源使用量，分别是 VPA（Vertical Pod Autoscaler）和 HPA（Horizontal Pod Autoscaler）[13]。VPA 是垂直扩展策略，通过收集应用程序的历史资源使用率数据，自动调整 Pod 的容器资源请求和限制。相比之下，HPA 则是一种水平扩展策略，它可以根据 CPU 利用率或其他指标，自动扩展或缩减 Pod 的数量。但两种资源管理策略存在的弊端是，均通过简单的阈值判断进行决策，增加资源副本数来横向扩展应用程序，或通过在容器中增加 CPU、内存等资源，以提高应用程序的负载能力，获得更优的性能表现。但如果没有正确调整，可能会导致过度缩放或不足缩放。此外，还包括准入控制、负载均衡、健康检查等机制，在表 1-1 进行详细介绍。

表 1-1 Kubernetes 默认资源调度策略

Table 1-1 Kubernetes default resource scheduling strategy

特点	技术	功能
调度	kube-scheduler	根据不同的策略将容器分配给工作节点。
准入控制	更改和验证控制器	在发布对象之前，但在对请求进行身份验证和授权之后，拦截对 Kubernetes API 服务器的请求，这提高了集群的安全性。
负载均衡	轮训机制	在多个容器实例之间分配负载，复杂策略可以由外部负载平衡提供。
健康检查	启动、活跃、准备和关闭探测器	控制容器是否可以答复请求。
容错	副本控制器	指定并维护所需数量的容器。

自动伸缩	水平自动缩放器	通过自动增加或减少 Pod 数量来重塑 Kubernetes 的工作负载，可使用自定义指标。

此外，这些策略均不是 SLO（Service Level Objective）[14][15]导向的，存在优化方向与用户需求相背离的风险。SLO 是指服务水平目标，用于定义和衡量服务的可靠性和性能，是服务质量指标的一种量化表达，它帮助团队和组织确定服务的期望水平，并为评估服务提供商的表现提供依据。因此，感知 SLO、以 SLO 为优化导向，是极具实际意义的[16]。

综上所述，本研究设计实现一个面向容器编排平台的容器集群自动化调优系统。该系统由控制模块、调优模块、参数资源管理模块、资源模块，共计四个功能模块构成。能够基于实时工作负载，进行离线-在线结合的识别和优化，能够有效利用历史经验，识别当前负载，并动态优化应用程序参数。此外，该系统同时支持资源和参数的联合优化，通过资源-参数联合调优方法及资源下降算法，在满足用户自定义的 SLO 前提下，提供基于实时工作负载的资源分配和参数设置的最佳组合。其优势在于：

(1) 离线训练和在线实时反馈相结合；

(2) 兼顾参数配置和资源分配；

(3) 面向用户实际需求，自动化调优。

本系统能够对云场景中部署的单一应用程序或微服务架构进行调优，以达到提高性能表现、提高资源利用率、提高用户并发程度、满足 SLO 的目的。

1.2 相关工作与研究现状

基于容器编排平台的面向集群的优化，主要包括资源分配优化和参数配置优化两部分。资源分配优化指的是改善容器编排平台为每个容器分配的资源数量（CPU、内存等）。参数配置优化指的是针对系统内核、容器运行时或应用程序等的参数配置的调优。

1.2.1 资源分配方法

目前集群资源的分配调度的优化，主要包括传统集群资源分配方法、基于数学建模的资源分配方法、基于启发式算法的资源分配方法以及基于机器学习的资源分配方法这四个方向。

1.2.1.1 传统集群资源分配方法

传统集群资源分配调度算法主要有以下三种[69]：

1. Min-Min 算法的策略是将任务调度到各项性能最高的节点上，这样可能会造成任务的过度集中，可能会调度时延较高或者节点崩溃的问题。
2. First-Fit 算法在调度任务时会尽可能将任务放到利用率较高的节点上，

这样会提高集群内的资源利用率也减小了碎片化的问题，但是可能会造成任务时延上升，服务质量下降。

3. Most-Full 算法会将任务调度到节点资源最多的节点上，这样可能会造成节点资源碎片过多的问题。

传统的集群资源分配调度算法在工作时，会带来服务质量下降和碎片化等问题，往往不能满足集群实际生产的需求。

1.2.1.2 基于数学建模的资源分配方法

线性规划、动态规划、整数线性编程（ILP，Integer Linear Programming）等数学建模技术可以将调度问题分析为一组受限方程，然后使用标准的优化技术来确定问题的最佳解决方案。但是 ILP 方法有很高的计算成本和复杂度，因此，这类方法只能用于小规模的问题[17]。AdaptiveConfig 将调度器的资源分配机制及其可变影响因素（配置、调度约束、可用资源和工作负载状态）转换为规则引擎中的业务规则和事实，从而在工作性能比较中推理这些相关因素，同时将巨大配置空间的集群级搜索转换为等效的动态编程问题[18]。

1.2.1.3 基于启发式算法的资源分配方法

启发式搜索算法往往基于一定的规则来进行搜索，找出可优化参数，进而找到更优配置，在传统的调度中被广泛使用。一些最知名的启发式策略是先入为主，先进先出，以及主导资源公平（DRF，Dominant Resource Fairness）。其中，Beltre A 等人[19]综合考虑到 DRF 的使用，以及任务的资源需求和平均等待时间，构建一个公平策略驱动的元调度器。但由于主导资源公平算法需要计算每个租户的主导份额[20]，随着租户数量的增加，调度开销变得非常显著。因此，该算法的应用一般都有较低的复杂度，明显存在优化质量欠佳的问题。

此外，元启发法是一种基于多个启发式算法集成的方法，其灵感来自于自然界中出现的智能过程和行为。这类算法的两个重要特征是对最适配的选择和对环境的适应，包括遗传算法（GA，Genetic Algorithm）[70]、蛙跳算法（FLA，Frog Leaping Algorithm）[71]和粒子群算法（PSO，Particle Swarm Optimization）[50]等等。Lin M 等人[21]建立基于容器的微服务调度的多目标优化模型，并提出蚁群算法来解决调度问题，利用可行解的质量评估函数来确保信息素更新的有效性，并结合多目标启发式信息来提高最优路径的选择概率。此外，Chandrasekaran K 等人[73]采用时间分片抢占的思想来解决 CPU 资源在容器间的分配不均问题。Yang Y 等人[74]使用优先级策略来解决集群网络资源在多任务请求下的竞争问题。Herbein S 等人[75]也采用优先级思想来解决了多容器下的磁盘 I/O 负载均衡的需求。但是，上述方法受限于启发式算法的局限性，不可避免容易出现过早收敛于非全局最优解的问题，同时，存在优化时间过长的缺点。

1.2.1.4 基于机器学习的资源分配方法

机器学习算法具有从大数据中学习和训练的能力，可以根据它们所学的内容做出智能调度决策。

Frim[15]使用微服务的各类负载，以及使用最终 SLO 当作奖励，来进行强化学习训练，从而对特定的微服务场景进行自动资源扩缩容。Frim 中值得注意的是：为了完成整个强化学习过程，Frim 引入错误注入机制，从而使得强化学习模型可以应对各式各样的特殊场景。由于 Frim 基于大量的前期训练以及其奖励机制，所以 Frim 并不能多目标优化，其更新策略时需要对模型进行完全的重新训练。这使得 Frim 在面对使用场景（用户工作负载，集群环境等）变化时，往往需要较大的代价进行重新训练。

Cliantro[22]提出的问题是希望在针对真实场景中在不需要预训练的情况下对真实场景的 SLO 基于资源分配进行优化。Cliantro 放弃了使用代理度量来对整个集群的负载进行识别，而是直接使用真实世界的端对端指标（平均延迟、P99 延迟等）来构建模型，从而更直接的做出资源-性能映射。除此之外，Cilantro 基于不同的搜索策略，提出学习机制和策略选择的解耦，通过整体微服务的反馈对整个微服务集群进行多目标调优。

Deeprest[23]则是使用深度学习算法基于微服务中 API 的流量的变化来预测不同的微服务应用的资源利用率（CPU、内存、IO）等。这一工作目标是对微服务的资源利用率和错误判断进行分析预测。其不足之处在于，这个工作需要庞大的数据量进行训练，同时仅仅是作为整个系统的观测和错误分析工具，并没有基于预测来进行资源或者参数推荐来优化性能。

Harmony[24]使用黑盒方法，不依赖详细的分析性能建模，采用深度强化学习。在一个神经网络中，将集群和任务状态映射到任务放置决策，基于不同位置决策对应的奖励样本，建立辅助奖励预测模型，使用有限的历史样本进行训练，对未知的位置决策计算奖励。类似地，DL2[25]使用深度学习驱动的深度学习集群的调度器，包括在线监督学习和通过强化学习的现场反馈。

1.2.2 参数调优方法

与资源的分配和调度不同，Kubernetes 等容器编排平台并不直接为集群中部署的不同类型的任务负载进行参数的优化，而是由用户自行指定或采用默认初始配置。这种复杂集群场景（例如微服务架构）的参数调优，与单一软件场景中有着巨大的区别。目前的研究当中，缺乏对整个系统乃至微服务的参数调优工作，这也是本研究相较于先前的资源分配优化工作的一个显著优势，即考虑资源分配的同时，兼顾针对不同应用程序的参数优化。

目前的参数优化工作往往针对于单一软件场景，例如面向应用程序的参数优化和面向系统内核的参数优化。这虽然与复杂集群场景的参数优化存在差异，

但这些工作在一定程度上可以给本研究提供一定的参考。

1.2.2.1 面向应用程序的参数优化

典型的应用程序例如数据库、分布式计算框架，都有许多参数优化工作。

针对数据库调优，BestConfig 使用基于搜索的启发式算法的，通过分割与发散采样（Divide and Diverge Sampling）和递归边界搜索（Recursive Bound and Search）递归寻找可能更好的配置。但是抽样和测试耗时较长，而且可能陷入局部最优[11]。CDBTune[26][27]和 QTune[28]使用了强化学习的方法来进行参数训练。通过将数据库状态到配置项之间的关系抽象成一个策略神经网络和一个值网络来进行反馈，将数据库调参问题转化为一个马尔科夫链决策过程，最终通过不断地自我训练得到最优参数。Ottertune[29]基于贝叶斯优化方法，提出通过基于数据库内部状态来做负载映射，进行负载分类，重用同类型负载的历史训练数据，在数据量较大时有助于帮助快速找到最优解。

针对分布式计算框架，如 Apache Spark，可以采用 SVM 建立模型来预测应用程序的执行时间[30]，或使用 ANN 建立性能预测模型[31]。然而，SVM、ANN 等都是浅层的机器学习方法，当数据集有较多的噪声或复杂的高维度时，它们的表现并不理想。Tunful[76]将增量灵敏度分析和贝叶斯优化相结合，使用少量执行从高维搜索空间中识别接近最优的配置，有效提升优化速度。GPBMOO[77]则针对 Spark 执行过程中调整的内核和驱动程序内存的数量，基于帕雷托最优，实现多目标优化的目标。DAC[88]实现一种数据大小感知的 Spark 配置调整系统，使用分层建模方法将工作负载执行时间近似为输入数据大小和配置的函数，并利用遗传算法根据模型估计的执行时间搜索良好的配置。Wang 等人[89]利用回归树来调整 Spark 配置，共调整 16 个配置参数，并将性能提高了 36%，但这种方法需要大量的执行样本来构建准确的回归树模型。

1.2.2.2 面向系统内核的参数优化

华为的 A-Tune[32]是一款基于 AI 开发的系统性能优化引擎，通过实时数据采集、传输到训练服务器，利用人工智能技术，对业务场景建立精准的系统画像，感知并推理出业务特征，进而做出智能决策，匹配并推荐最佳的系统参数配置组合，使业务处于最佳运行状态；TuxML[33][34]使用随机森林算法基于配置项对编译出的系统内核进行错误检测和镜像大小预测；TEAMS[35]提出一种新的转移进化感知模型转移方法，利用可配置系统的结构，以最小的额外处理量将初始预测模型转移到其未来版本，对不同版本的 Linux 内核配置进行迁移学习，并得到了很好的效果。KernTune[36]采用支持向量机（SVM）的方法对系统的当前负载做分类，持续监控某些性能指标，每当这些指标发生变化时，系统就会确定新的系统类，并动态调整更新内核的运行时参数。

1.3 本文研究内容

本文以目前的云原生体系结构中，容器编排平台所部署和维护的应用程序为研究对象，针对其不合理的资源调度分配以及参数配置导致性能下降、资源浪费、无法满足用户需求的问题，深入分析优化这一现状所面临的挑战，进而设计、构建面向容器编排平台的集群自动化调优系统。本系统面向参数配置和资源分配两个问题，分别提出离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法。

具体地，本文的研究内容展开如下：

1. 面向容器编排平台，遵循模块化分布式部署的技术路线，设计实现面向集群的自动化调优系统。其主要包括四个模块：控制模块、调优模块、参数资源管理模块、观测模块。控制模块控制管理整个调优系统的运行逻辑，同时作为控制中心与其他三个模块进行交互。调优模块部署优化算法，以有状态服务器的形式，接受性能或状态信息，并优化资源分配和参数配置。参数资源管理模块提供可自定义的参数以及资源修改接口，负责根据调优策略更改集群中容器状态。观测模块负责执行基准测试收集性能指标，或根据需求监控状态指标。
2. 在应用程序参数配置优化方面，针对如何在负载到达集群时，有效利用历史经验或专家知识库等先验知识，进行准确识别和初始化的参数推荐，以避免初始化的不合理配置导致较差的性能表现，同时能够在较优的配置基础上进行快速的迭代优化的挑战进行研究，提出了基于集成学习的离线辅助训练方法和基于机器学习的在线的参数调优方法，前者使用基于分类回归树的最佳深度的随机森林进行负载识别，并通过回归分析筛选与性能相关度较高的参数，迭代完成负载知识库和参数知识库的构建，后者包括：使用高斯过程回归作为代理模型的改进的贝叶斯优化算法或基于随机搜索和网格搜索无模型的搜索算法的调优流程，同时提供多目标优化的选项，兼顾用户自定义需求。
3. 在微服务系统优化方面，针对如何在满足用户 SLO 的前提下，兼顾资源和参数的优化，减少集群整体资源消耗，同时提高集群并发处理能力，提出了资源-参数联合调优及资源下降算法。面向 SLO 进行资源总量的调整，基于多层次优化的理念，结合变异算法完成对最佳资源分配的搜索，在搜索结果基础上优化参数配置，探索更低的资源占用以及更高的并发支持。
4. 设计了大量实验验证，在常见的不同类型应用程序，Nginx[8]、Memcached[37]、Redis[38]和 Postgresql[39]上验证优化系统的有效性。结果表明，本系统能够在各种资源分配条件下为应用程序带来性能优化，尤

其在高资源供给的情况下，本系统能够使得应用的平均性能提高超过46.55%至 208.20%。具有实际工作负载的微服务架构场景中，本系统在满足服务水平目标（Service Level Objectives, SLO），即保证平均延迟和 P99 延迟以满足用户使用需求的同时，可节省 37.5%以上的资源，或使用相同资源支持用户并发数达到默认配置的 3 倍以上。

1.4 论文组织结构

本文共分为七章，图 1-3 描述了本文各章之间的关系。

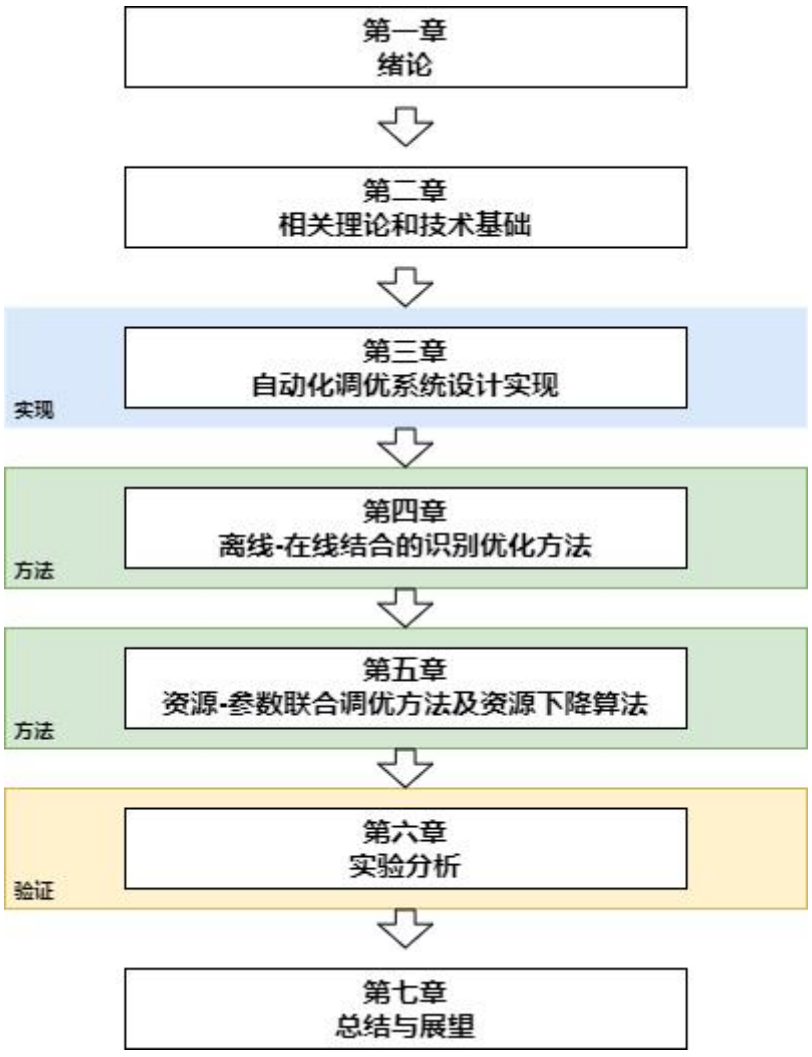

图 1-3 论文组织结构

Figure 1-3 Thesis Structure

具体每一章的详细内容安排如下：

第一章，绪论。该章首先阐述本研究的研究背景及意义，其次介绍当前集群资源配置优化和应用参数优化的分类以及研究现状，通过与其他调优方法和调优系统做对比，引出本系统的设计定位和调优目标。最后简要介绍本文所研

究的主要内容，最后给出本文的组织结构。

第二章，相关理论和技术基础。主要是针对本文系统开发所涉及的相关技术进行详细介绍。首先是虚拟化技术，对比传统虚拟化技术和容器虚拟化技术，介绍较为流行的轻量级容器虚拟化技术；其次是对典型容器编排平台 Kubernetes 技术的相关介绍，主要是对 Kubernetes 中的基本概念和系统架构进行阐述；最后对微服务架构以及典型的微服务基准测试 DeathStarBench[40]。

第三章，自动化调优系统设计实现。阐述本研究设计、实现的面向容器编排平台的优化系统的结构和功能，其主要包括四个模块：控制模块、调优模块、参数资源管理模块、观测模块。详细说明其工作流程和交互方式。

第四章，离线-在线结合的识别优化方法。本章介绍离线-在线结合的识别优化方法，主要分为两部分，基于集成学习的离线辅助训练方法和基于机器学习的在线的参数调优方法，前者包括负载知识库的构建和参数知识库的构建，后者包括基于模型和无模型的调优流程。

第五章，资源-参数联合调优方法及资源下降算法。首先介绍面对微服务调优区别于单一应用调优的两个挑战以及本研究的调优策略，其次阐述资源分配部分的变异操作函数和进化算法，最后介绍资源-参数联合调优方法及资源下降算法。

第六章，面向容器编排平台的集群自动化调优系统的实验评估。具体介绍应用程序调优和微服务调优，评估离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法的有效性。应用程序调优部分实验结果表明，本研究能够在不同资源配给的情况下，相较默认配置，使得应用程序性能表现产生明显改善。尤其是资源供给充足的情况，优化效果明显。微服务调优部分的实验，具体包括两种调优目标，分别为资源优化实验和并发优化实验。评估结果显示，本研究可使得微服务系统在更低的资源配置情况下，达到相同的性能表演或更高的并发支持能力，优化效果显著。

第七章，总结与展望。本章结合前文章节对论文进行总结概述，并结合现有工作和结果对未来工作提出展望。

第 2 章 相关理论和技术基础

本章将介绍设计与实现基于容器编排平台的集群自动化调优系统的相关技术，包括虚拟化技术、Kubernetes 技术概述、基础优化算法概述等。本章将对上述相关技术进行详细阐述，为后续的设计与实现工作奠定基础。

2.1 虚拟化技术

虚拟化技术是一种将计算资源（如计算机硬件、操作系统、存储等）抽象化的技术。通过虚拟化技术，可以将物理硬件资源转变为多个虚拟实例，从而使得多个操作系统或应用程序能够在同一台物理设备上同时运行，而彼此之间相互隔离，互不干扰[41]。在过去几年，虚拟化技术取得了重大突破，在云平台、物联网和网络虚拟化等领域发挥着关键作用。虚拟化技术具有硬件独立性，可以使软件在不同硬件平台上运行，同时还提供了卓越的隔离性和安全的用户环境。此外，虚拟化技术的可扩展性也在不断提高，能够满足不断增长的需求。这些特点使得虚拟化技术成为当今云平台、物联网和网络虚拟化等领域中不可或缺的关键技术。

2.1.1 虚拟化技术概述

虚拟化技术的起源可以追溯到 1964 年，当时 IBM 公司启动了一个名为 CP/CMS 系统的项目。该项目中的 CP（Control Program，控制程序）组件作为操作系统，负责管理在自身计算机系统上的多个副本。每个副本后来演变成了虚拟机，它们通过各自的操作系统 CMS（Cambridge Monitor System，会话式监控系统）组件进行控制[40][41]。VMware Inc 公司推出了商用化的 VMware 虚拟平台，该平台专为 X86_32 体系结构设计。这个虚拟平台可以在基于 X86_32 架构的计算机系统上运行多个虚拟机，实现硬件资源的有效利用和隔离。

VMware 的虚拟化平台是基于 Hypervisor 的虚拟化技术。Hypervisor（虚拟机监控程序）是一种软件、硬件或者组合体，它在物理计算机上创建和运行虚拟机的环境。Hypervisor 允许多个操作系统实例（虚拟机）在同一台物理计算机上同时运行，每个虚拟机都被分配一部分计算资源，并且它们之间是相互隔离的。Hypervisor 负责管理和分配物理计算机的资源，以确保各个虚拟机之间的稳定运行和隔离。通过使用 Hypervisor，可以实现服务器的虚拟化，提高硬件资源的利用率和灵活性。Hypervisor 的实现可以分为两种模式：本机或裸机管理程序模式和宿主机管理程序模式[41][42][43]。

传统虚拟化技术利用 Hypervisor 将一台计算机系统虚拟化成多台逻辑计算机，这些虚拟机共享主机端硬件资源，突破了物理限制，并有效提升了单机系统的硬件资源利用率。Hypervisor 在硬件层级上运行，独立于主机操作系统，

并与主机操作系统相互隔离。通过 Hypervisor 实现的多虚拟机环境，能够对硬件资源进行隔离，确保各个虚拟机之间的独立性和安全性。

2.1.2 容器虚拟化技术

容器（Container）虚拟化技术是一种轻量化的 Hypervisor 的替代方案。虽然它与传统的虚拟化技术在实现硬件资源虚拟化和隔离方面有所相似，但其工作方式却有着显著的不同。不同之处如图 2-1 所示，传统虚拟化技术利用 Hypervisor 为每个虚拟机创建虚拟硬件并运行完整操作系统，而容器虚拟化则直接在主机内核上运行应用进程，从而避免了硬件虚拟化和设备驱动程序的额外开销。由于容器本身是不涉及硬件虚拟化和内核系统的，多个容器之间可以共享主机的硬件和内核，因此极大降低容器的创建和维护过程的复杂程度。此外，由于容器不包含虚拟机操作系统，其镜像相较于虚拟机镜像，往往更为精简，这更加便于应用的迁移和部署。与此同时，正因为容器不需要模拟硬件或启动容器内部操作系统，其启动时间可以缩短到几毫秒以内，启动响应速度显著提高[42]。然而，这种共享宿主机操作系统的模式也有其局限性：容器虚拟化技术不能够实现虚拟机在 Windows 操作系统的主机上运行一个 Linux 容器的功能。此外，因为宿主机操作系统同时对多个应用容器开放，所以其提供的隔离性和安全性十分有限，不适用于需要更严格安全要求或者需要运行不同操作系统的场景。

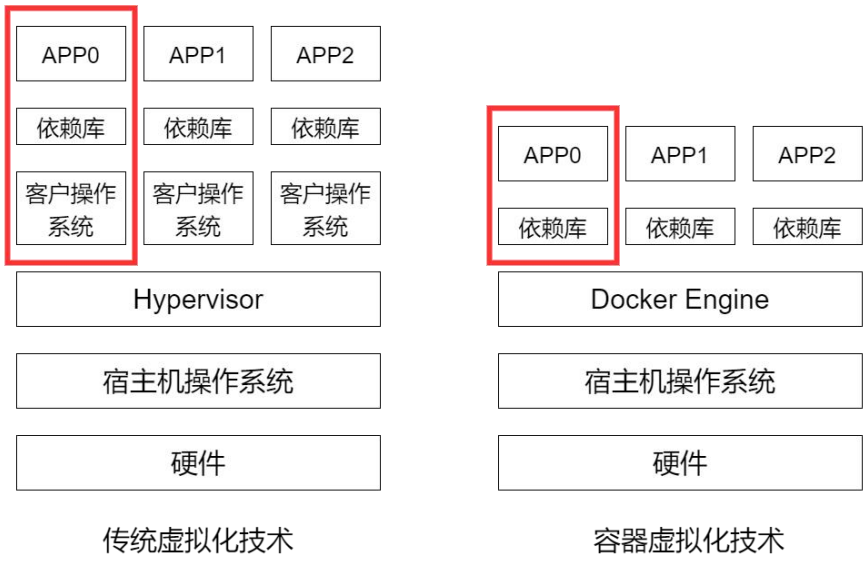

图 2-1 传统虚拟化和容器虚拟化架构对比

Figure 2-1 Comparison between traditional virtualization and container virtualization

容器虚拟化技术的隔离方案由 Namespace（命名空间）机制、Cgroups（Control Group，控制组）机制和 chroot（Change Root，切根机制）实现。

1. Namespace：Namespace 可以用来隔离进程的 PID（进程 ID）、网络、挂载点、用户 ID、IPC（进程间通信）等系统资源，从而实现了容器

间的隔离。通过使用 Namespace 技术，容器可以在同一台宿主机上运行，但彼此之间相互隔离，就好像它们在各自独立的系统上运行一样。这种隔离性使得容器在共享宿主机资源的同时能够保持相对的独立性，避免了相互之间的干扰，同时提高了安全性[42]。

2. Cgroups：控制组机制是一种非强制性机制，用于限制、记录和隔离多进程对物理资源的使用。表面上看 Namespace 机制起到了隔离进程组的功能，但只是从 Linux 内核的角度在进程间形成一种可视化的隔离，而从底层物理资源的角度看，多进程依旧是共享完整宿主机物理资源。与之对应的是，控制组机制对容器在共享主机物理资源上进行限制，通过在 Linux 的/sys/fs/cgroup 目录中进行配置，可以对一组进程施加各种限制，包括 CPU 使用、内存使用、磁盘 I/O 以及网络带宽等资源最大使用量，限制容器使用超出容器申请的硬件资源，保障其他容器的正常资源请求。使用 cgroup 可以将一组进程组织在一起，并对其施加资源限制。这样既可以确保重要的系统进程能够获得足够的资源，又可以防止某些进程占用过多的资源导致系统性能下降，从而有效地进行资源管理和调度[44]。
3. Chroot：Change Root 机制可以将进程限制在指定的目录下，使其以该目录为根目录，无法访问该目录之外的文件系统。Chroot 机制设置的根目录文件夹则会被目标进程认为是新系统根文件夹（/），为不同的容器设置不同的虚拟根目录，使多租户进程运行互不影响。使用 Chroot 可以创建一个独立的、隔离的环境，这个环境中的进程只能访问和操作 Chroot 所指定的目录及其子目录。这种隔离机制可以提供一定程度的安全性和保护，防止进程对系统其他部分进行未授权的访问和修改[42][45]。

目前，容器虚拟化技术有很多种，而其中应用最广泛的就是 Docker 轻量级 容器虚拟化，其一整套 Docker 容器管理相关的生态推动了云原生的发展。

2.1.3 Docker

Docker 的概念最早在 2013 年的 Python 开发者大会上首次提出，其设计实现包括 Docker 守护进程、Docker 客户端、Docker 镜像、Docker 容器和 Docker 仓库等部分。Docker 作为客户端-服务器架构[46]，用户通过 Docker 客户端与后端的 Docker 守护进程建立通信，通信命令包含完整的 RESTful（Representational State Transfer，具有代表性的状态传输）API[47]。

Docker Daemon（Docker 守护进程）：守护进程作为 Docker 中的主体部分，是长期驻留在宿主机后台的服务进程，负责管理 Docker 对象，如镜像、容器、网络和数据卷。它接收来自 Docker 客户端的请求，并管理 Docker 对象的创建、

启动、停止等操作。Docker 守护进程包括 Docker Server 和 Docker Engine。Docker Server 作为入口程序，接收客户端命令，根据类型和参数将其分配给相应 Handler 执行。在 Docker 守护进程中，最小任务单位是 Job，每个任务都可以抽象为一个 Job，并由 Handler 在 Docker Engine 中执行具体任务的打包。

Docker Client（Docker 客户端）：是任何遵循 Docker API 的客户端泛称[48]，用户与 Docker 交互的主要方式，用户通过 Docker 客户端向 Docker 守护进程发送请求，如构建、运行和管理 Docker 容器。具体过程为：客户端通过套接字与 Docker 守护进程中的 Docker Server 进行通信，Docker Server 会不断的监听通信端口，并将处理完成的结果信息返回给 Docker 客户端。

Docker Images（Docker 镜像）：Docker 容器的基础，包含应用运行所需的文件系统和软件。镜像是一个轻量级、独立的可执行软件包，其中包含运行某个应用程序所需的所有内容：代码、运行时、库、环境变量和配置文件。在 Docker 中，镜像可以作为容器的模板，用于创建并运行容器实例。镜像可以被分享、存储和重复使用，使得应用程序的部署变得更加可靠和高效。一般的，用户会通过 Dockerfile 文件将应用程序、中间件、编译库、网络配置、操作系统等所需要的环境打包成镜像，并由容器进行运行[49]。生成的镜像文件会通过 Docker Push 命令上传至镜像仓库进行存储。镜像仓库分为私有和公有两种。

Docker Containers（Docker 容器）：基于 Docker 镜像创建的可运行实例。每个容器都是一个独立的运行环境，包含应用程序及其依赖项，可以被启动、停止、删除等操作。

Docker Registry（Docker 仓库）：用于存储 Docker 镜像的地方，可以是公共的如 Docker Hub，也可以是私有的。用户可以从仓库中拉取镜像到本地使用，也可以将自己构建的镜像推送到仓库中分享给他人。DockerHub[50]是迄今最大的公有镜像仓库，类似于 Github，将开源的容器镜像维护起来，以供用户进行拉取使用。容器从镜像仓库中拉取镜像，容器中应包含应用程序所需的整个工具链，因此容器之间会以隔离的方式进行运行。

Docker 作为轻量级容器虚拟化技术，与传统虚拟化技术相比具有众多优势：

（1） 更快的启动时间：Docker 作为操作系统进程，不依赖宿主机操作系统，不需要像传统虚拟机那样启动整个操作系统，只需启动应用程序及其依赖，这使得在几秒内启动多个容器成为可能。

（2） 快速部署和持续的 CI/CD（持续集成/持续交付）：通过 Docker 的快速部署功能，开发人员可以将应用程序与其依赖项打包到一个独立的镜像中，并在不同环境中轻松部署，确保开发、测试和生产环境的一致性。真正做到“一次构建，到处运行”，简化了部署过程，方便应用的迭代、迁移和交付，为软件开发和部署提供了巨大的便利性和效率。

（3） 高效的资源利用率： 传统虚拟化技术使用 Hypervisor 来虚拟化硬件，每个虚拟机都需要一个完整的操作系统镜像，包括内核和系统库。这导致了较高的资源消耗，包括内存和存储空间。Docker 容器共享主机操作系统的内核，因此容器之间可以共享系统资源，减少了冗余。每个容器只包含应用程序及其依赖，这使得资源利用更加高效。

（4） 低廉的成本：Docker 生态项目开源，方便快速部署；DockerHub 作为迄今最大的公有镜像仓库，存储丰富的开源镜像，由专业的开源团队持续维护，可高效便捷投入生产环境。

2.2 Kubernetes 技术概述

随着容器化技术的广泛应用，逐渐出现了新的技术挑战和研究问题，其中包括在大量容器部署的生产环境中，如何协调和管理这些容器，提高应用程序的敏捷性、可靠性和可维护性等[5]。为了适应动态变化的业务需求和技术环境，容器编排平台和相关技术发展迅速并广为部署，目前较为流行的容器编排平台有 Kubernetes、YARN、Omega、Mesos、Docker Swarm 等。本文所涉及的系统以 Kubernetes 为基础，具体介绍 Kubernetes 技术的相关概述。

2.2.1 Kubernetes 基本概念介绍

Kubernetes 最初是由 Google 开发的一个用于管理容器化应用的开源项目，其设计目标是解决 Google 内部大规模容器集群的管理问题，后由云原生计算基金会（Cloud Native Computing Foundation，CNCF）管理[4]，是一个基于 Docker 技术的开源容器编排平台，用于管理容器化的工作负载和服务，提供了强大的容器编排功能，可以让用户更加方便地部署、扩展和管理大规模的容器化应用程序，并通过自动化的方式实现负载均衡、服务发现和自动弹性伸缩等功能。

Kubernetes 拥有集群管理的能力，采用模块化和插件化的开发模式，具有可移植和可扩展的特点。整个 Kubernetes 平台由多个组件集成，这种设计极大地方便了集群管理工程师根据实际需求进行二次开发和定制化。对于集群的部署，Kubernetes 提供了简化的流程，管理员可以通过一键式部署方式（kubeadm init）来快速部署集群，无需过多关注部署细节，从而降低了使用门槛和部署成本。在安全性方面，Kubernetes 实现了多层次的安全机制。底层多组件之间的通信需要通过数字证书进行身份验证，采用双向的 TLS 认证机制，即客户端和服务器都需要提供身份验证信息，有效防止了集群内部信息的非法访问。在应用层面，Kubernetes 具有故障检测和自我修复的功能。集群会持续监测节点、组件和容器的状态，一旦发现失败任务，系统会自动进行重启和修复，保障集群的稳定性和可靠性。这些特性使得 Kubernetes 成为一个强大且安全的集群管

理工具，为企业提供了高效、可靠的容器编排和管理解决方案。

Kubernetes 同时具备集群资源整合的功能，通过统计集群资源情况，根据容器申请资源数量和调度算法，合理地进行资源分配，实现细粒度的资源配额和集群负载均衡。随着云原生生态的不断完善，越来越多的优秀项目加入到 CNCF 的大家庭中，Kubernetes 拥有庞大的生态系统，包括各种插件、工具和服务，如 Helm（包管理工具）、Prometheus（监控工具）、Fluentd（日志收集工具）等，可以帮助用户更好地使用和扩展 Kubernetes。正是因为 Kubernetes 的安全、易用、高效等优势，越来越多的企业选择 Kubernetes 管理公司集群。

Kubernetes 常见的资源对象有 Pod、Controller（Deployment 等）、Service、Job、Label 等，此外，本研究涉及用于扩展 API 的自定义资源定义机制。这些资源对象作为 API 类型通过 YAML 文件进行创建，并通过对象 Spec 字段管理对象具体配置。具体地，如下描述了这类资源对象的管理和基础运作流程。

1. Pod 是 Kubernetes 的基本控制和管理单元[52]，也是最小的调度单位[53]。Pod 可以视为一组容器的抽象，其中包含一个或多个共享存储系统、网络接口等资源的容器。Pod 为容器提供虚拟主机环境，使其能相互协作完成任务。Pod 通过 Spec 字段设置的存储卷（Volume）路径挂载宿主机文件系统，容器可将数据读写入共享宿主机的文件系统实现数据持久化存储，即数据不因容器的崩溃而丢失[54]。Pod 在某一时刻有且只能部署在集群中的一个节点上，即 Pod 中所有容器只能运行在一台机器上，一旦被创建将会持续运行在该节点上，只有通过手动的删除才能将 Pod 驱逐[53]。Pod 是整个 Kubernetes 的核心，所有组件都是以 Pod 为基础进行运转。
2. Deployment 资源控制器用来管理无状态服务，无状态服务实例无需挂载宿主机持久化存储卷，因此可以部署在任意节点上，适合作为任务迁移的目标资源对象。Deployment 基于 RC（ReplicationController）机制，支持多副本动态扩展。创建时会根据设置值动态生成副本数量的 Pod，当 Pod 因故障退出时，Deployment 自动检测正常运行 Pod 数量与副本数进行对比，自动创建新的服务 Pod，使个数维持在副本数。当服务相应流量持续增大时，也可以通过修改副本数为服务扩容，提升响应速度。
3. Service 是为一组相同功能的 Pod 提供统一的访问接口，并将请求负载均衡的分发给各个 Pod[55]。面对多个后端程序的 Pod 为前端提供服务，当后端 Pod 所在节点发生故障，Kubernetes 自动重启恢复，由于每个 Pod 特有的 IP 地址也随之发生变化，前端无法正确访问后端 Pod IP 导致服务失败。Service 将具有相同 Selector 的 Pod（即相同功能）组织

起来，对外提供一个统一的访问 DNS 接口，这样即使 Pod 发生变化，接口地址也不会跟随变化。

4. Job 会创建一个或多个 Pod 直至成功运行的 Pod 达到指定的数量为止。当有一个 Pod 失效退出时，Job 会自动创建新的 Pod 来完成任务。
5. Label 是附着在 Kubernetes 资源对象上的键值对，旨在为资源对象提供用户设置的标识属性，以达到组织松耦合的多个 Pod 的目的。一般在调度场景中指定特殊的标签值为调度器提供筛选标准。
6. Controller 是一种核心的控制器模式，用于确保集群中的实际状态与期望状态一致。它负责监控资源对象的变化，并采取相应的操作以维持所需的状态。控制器可以包括 ReplicaSet、Deployment、StatefulSet、DaemonSet 等，它们分别用于管理 Pod 的复制、更新和状态稳定性。用户也可以编写自定义的 Controller 来满足特定的业务需求，例如自动缩放、自动恢复等。
7. CRD（Custom Resource Definition）是 Kubernetes 中用于扩展 API 的机制，允许用户定义自定义资源类型。这些自定义资源类型可以像内置资源一样被 Kubernetes API 服务器处理。用户可以使用 CRD 来定义自己的 API 资源，并编写对应的 Controller 来对这些资源进行管理和控制。通过 CRD，用户可以创建自定义资源类型，如数据库实例、消息队列实例等，然后编写 Controller 来实现这些资源的自动化管理。

2.2.2 Kubernetes 基本架构介绍

Kubernetes 分布式云计算集群架构采用主从模式，由若干个 Master 节点和 Worker 节点构成[56][57]。图 2-2 描述 Kubernetes 各个组件的布局和交互方式。Master 节点作为集群管理节点，运行多个服务端组件；Worker 节点为实际工作节点，需要与 Master 节点定期通信，汇报节点状态，因此，Worker 节点上一般运行着主机通信服务和工作服务。Master 节点也可以复用为 Worker 节点。具体有以下几个核心组件。

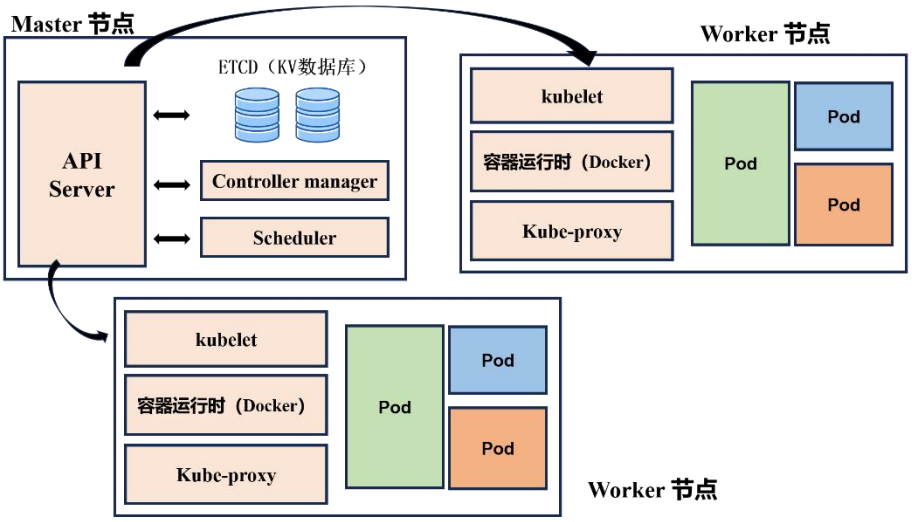

图 2-2 Kubernetes 架构图

Figure 2-2 Kubernetes architecture diagram

1. Master 节点上的核心组件：

（1）ETCD（分布式键值存储）：是一个一致性且高可用的键值（Key-Value）数据库，由 etc（UNIX 系统的 etc 配置目录）和 distributed（分布式）共同组合命名。ETCD 提供了一个简单而强大的分布式键值存储系统，可以用于存储关键数据，如配置信息、元数据等。多个 ETCD 服务器通过 Raft[58]共识算法选举出主节点作为 leader，由 leader 节点广播日志和数据，解决分布式一致性问题，整个集群的状态信息，如 Pod、Service 等的配置信息，会以键值对的形式存储在多个 ETCD 持久化存储卷中，提供对节点故障的容错机制[59]。

（2）API Server（API 服务器）：API Server 是 Kubernetes 集群的核心组件，担任集群中的“桥梁”工作，作为 Kubernetes 控制端前端，为集群控制端提供了各类资源对象的接口服务。所有的集群操作都通过 API Server 进行，包括创建、更新、删除资源对象等操作。API Server 通过通信接口与 Worker 端组件传递信息，是整个系统的数据总线和数据中心[53][57]。

（3）Controller Manager（控制器管理器）：作为集群内部的管理中心，负责集群中 Node（节点）、Pod、Namespace（命名空间）和资源的管理，其通过 API Server 提供的接口持续监控集群状态，并在发生故障时进行自动化修复。同时负责运行控制器，监控集群中的资源对象的状态，并根据预定义的规则进行调节。例如，Replication Controller 负责管理 Pod 的副本数量。

（4）Scheduler（调度器）：是 Kubernetes 集群默认的调度器，负责根据 Pod 的资源需求、调度策略和节点的可用性等因素来进行调度决策，将新创建的 Pod 分配到集群中的 Node 节点上。调度过程大体为预选和优选两步走的方式[54]，预选粗粒度筛选，找到所有符合条件的节点，再根据预设算法对所有符合条件的节点进行打分，最终实现创建 Pod 与选中 Node 之间的绑定[53]。一般的，在 Kubernetes 中可以部署多个调度器，并在创建对象的 YAML 文件中指

定调度器。

2. Worker 节点上的核心组件：

（1）Kubelet：是集群在每一个工作节点上的“代理”，主要负责创建、部署和管理 Pod，具体地，通过接收 Pod 的创建、更新、删除等指令，确保 Pod 和容器的运行状态符合预期。其次，Kubelet 会向 Master 节点上的 API Server 注册本节点，将节点加入到集群中，并定期向 API Server 汇报节点的运行情况以及从 API Server 中接收新的节点状态并进行修改。

（2）Kube-Proxy：是工作节点上的网络代理组件，负责维护节点上的网络规则，尤其是均衡网络流量负载。Kube-Proxy 管理 Service 的访问，根据特定 IP 将服务发送到正确的后端程序 Pod，是 Kubernetes 网络的核心组件。它维护节点上的网络规则，使得服务可以被集群内外的其他组件访问。

（3）Container Runtime（容器运行时）：是负责运行容器的软件，常见的 Container Runtime 包括 Docker、containerd、CRI-O 等。它负责拉取镜像、创建容器、管理容器的生命周期等操作。

通常情况下，用户会通过 Kubectl 命令工具发起创建 Pod 请求，API Server 会首先收到请求并将请求中 YAML 文件的创建信息写入 ETCD 中，读写完成后，将确认信息返回至客户端；Controller Manager 会持续监控 API Server 的监控接口，当发现了任务的信息变更，执行资源整合程序，将任务的资源信息进行整合写入 ETCD 中；同时，Scheduler 同样通过 API Server 的监控接口发现可调度的 Pod 信息，通过预设的调度算法将 Pod 与选中的节点进行绑定，并将绑定信息写入到 ETCD 中，而 Pod 信息会交给选中节点的 Kubelet 进行 Pod 创建；Kubelet 开始创建 Pod 资源对象，依次调用 CNI 接口为 Pod 创建网络、CRI 接口启动 Pod 中的容器、CSI 接口为 Pod 创建挂载卷。至此 Pod 创建成功[60]。

2.3 微服务架构

近年来，微服务作为一种新的应用开发模式引起了人们的广泛关注，并在许多场合被采用。基于微服务架构，应用程序被设计为一组独立的、细粒度的模块化服务，每个模块化服务执行单个业务任务，微服务之间使用轻量级通信机制。应用程序的需求是通过一组协作的微服务来实现的。微服务应用程序易于部署和更新，允许在不重新启动的情况下独立更新和重新部署某些服务，并具有易于连续交付的特点[61]。

具体地，微服务将分布式应用程序分解为小型独立可部署的服务，每个服务在自己的过程中运行，并通过轻量级机制进行通信。这些服务是围绕独立的业务功能构建的，可以使用不同的编程语言和不同的数据存储技术编写。它们

通常由完全自动化的部署和协调机制支持，例如在云中，使每个服务能够经常以任意的时间表部署，只需最低限度的集中管理，通常遵循行业验证的 DevOps 实践[87]。

对微服务应用程序的设计、开发和评估需要可重复的实证研究，微服务架构基准测试使得研究者能够对这种新的架构风格进行可重复的实证研究。其中，DeathStarBench[40]，是由社交网络、媒体服务和酒店预订等微服务应用程序组成的云微服务的开源基准测试套件。社交网络实现了一种具有单向关注关系的广播式社交网络，用户可以通过该网络发布、阅读社交媒体帖子并对其做出反应。媒体服务提供诸如审查、评级、租赁和流媒体电影等功能。酒店预订是一个在线酒店预订网站，用于浏览酒店信息和进行预订。这些基准使用各种编程语言，包括 Java、Python、Node.js、Go、C/C++、Scala、PHP 和 Ruby，且所有微服务都部署在单独的 Docker 容器中。

DeathStarBench 旨在帮助研究人员和工程师评估在大规模分布式系统中部署新技术或优化现有技术时的性能表现。通过使用 DeathStarBench，用户可以模拟和评估在大规模环境下各种常见的工作负载，从而更好地了解其在云基础设施中的表现。这可以帮助他们进行系统设计、优化和决策，以提高系统的性能、可靠性和效率。

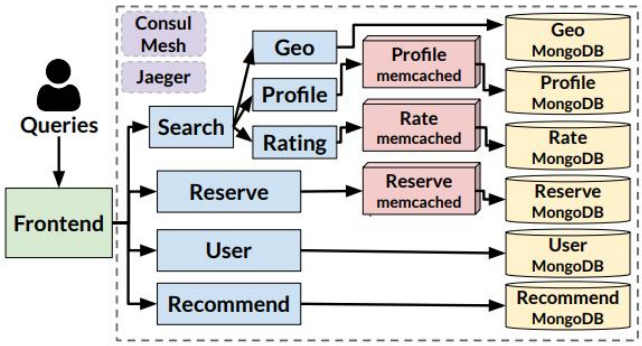

图 2-3 HotelReservation 微服务架构

Figure 2-3 HotelReservation microservice architecture

如图 2-3 所示，酒店预订微服务（HotelReservation）实现了酒店预订服务，使用 Go 和 gRPC 构建。最初的项目以多种方式扩展，包括添加后端内存和持久数据库，添加用于获取酒店推荐的推荐系统，以及添加预订酒店的功能。其中，Nginx 作为前端入口负责请求的路由和负载均衡；Memcached 用作缓存层提高系统性能；而 MongoDB 作为后端数据库主要承担数据存储和管理的任务。

2.4 本章小结

本章主要是针对本文系统开发所涉及的相关技术进行详细介绍。首先是虚拟化技术，对比传统虚拟化技术和容器虚拟化技术，介绍较为流行的轻量级容

器虚拟化技术；其次是对典型容器编排平台 Kubernetes 技术的相关介绍，主要是对 Kubernetes 中的基本概念和系统架构进行阐述；最后对微服务架构以及典型的微服务基准测试 DeathStarBench 进行概述。

第 3 章 自动化调优系统设计实现

3.1 设计总览

本研究实现的是一个面向容器编排平台的容器集群参数配置和资源分配的自动化调优系统。其目的在于实现离线训练和在线实时反馈相结合，对云场景中部署的单一应用程序或微服务架构，进行参数配置以及资源分配的自动化调优，以达到节省资源、提高资源利用率、满足 SLO 的目的。

基于上述观点，本研究构建这样一个面向容器编排平台的优化系统，如图 3-1 所示，其主要包括四个模块：控制模块、调优模块、参数资源管理模块、观测模块。其中，控制模块控制管理整个调优系统的运行逻辑，同时作为控制中心与其他三个模块进行交互。调优模块部署优化算法，以有状态服务器的形式，接受性能或状态信息，并优化资源分配和参数配置。参数资源管理模块提供可自定义的参数以及资源修改接口，负责根据调优策略更改集群中容器状态。观测模块负责执行基准测试收集性能指标，或根据需求监控状态指标。本章后续部分，将对四个模块进行详细的介绍。

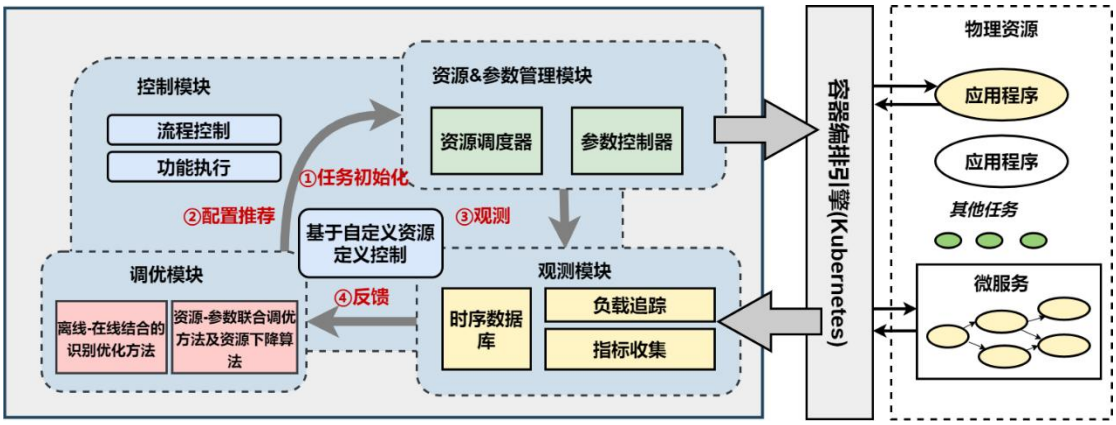

图 3-1 面向容器编排平台的自动化调优系统架构

Figure 3-1 Architecture of Automated Optimization System for Container Orchestration Platform

本系统有以下几点优势：

1. 自动化调优：调优模块可插拔式集成改进的优化算法，自动选择合适的优化算法，实现负载识别、参数调优、微服务调优等多种功能；
2. 具有高度拓展性：资源-参数模块设计为代理仓库模式，因而可以支持多软件栈、多场景的参数适配接口，包括面向各类操作系统的性能调优，以及通用应用软件的性能调优；
3. 根据 Kubernetes 机制深度定制：设计基于 Kubernetes 控制器模式开发，方便部署、拓展和发布，提供通用接口，对多样化的用户环境进行适配；

4. 确保容器指标高效观测：通过自定义资源定义（CRD），为每一次性能观测或指标收集生成一个独立对象，确保数据一致性；

3.2 控制模块

控制模块控制管理整个调优系统的运行逻辑，同时作为控制中心与其他三个模块进行交互。主要负责控制和管理集群中需要调优的资源对象、创建调优任务、管理调优流程，以及通过生成对应的调优指令对调优模块给出的推荐参数和策略进行部署。其中，管理调优流程包括：离线和在线调优的选择、调优目标资源的选择、参数和评价指标的选择等等。同时，控制模块负责控制整个系统中的调优任务的并行以及错误处理。其功能可概括为功能执行和流程控制两部分。

3.2.1 功能执行

控制模块负责执行调优的迭代过程、多个任务的并行处理以及流程中错误、超时处理等等。同时还负责各个模块之间的协调工作，例如观测负载的执行状态，服务修改资源、参数后的状态，以及调优模块和其他模块之间的通信等等。

3.2.2 流程控制

控制模块基于自定义资源定义（Custom Resource Definition，CRD）和控制器（Controller），将整个调优流程的控制抽象为不同类型 CRD 的维护和管理过程。这个过程当中，Controller 针对相应的 CRD 资源对象的变化进行监听（create、update、patch、delete 等行为），并在发现事件的改变之后，做出相应操作，其本质逻辑是维护资源当前状态和资源定义状态的一致性。

控制模块具体包括 5 个 CRD 和 3 个 Controller，如表 3-1 所示。

表 3-1 自定义资源定义和控制器概述

Table 3-1 Overview of Custom Resource Definitions and Controllers

CRD 名称	对应的 Controller	概述
task	taskcontroller	控制整个调优流程的资源，与调优模块交互，用户通过定义 task 来进行一次调优任务
controlagent	无	用于保存修改参数的名称以及修改方式，被 controlstepcontroller 读取

controlstep	controlstepcontroller	基于选择的 controlagent 的信息完成参数的修改
collectagent	无	用于保存生成基准测试或读取观测信息的方式，被 collectstepcontroller 读取
collectstep	collectstepcontroller	基于选择的 collectagent 信息完成基准测试或读取观测信息

控制模块如图 3-1 所示，负责进行在线调优流程的管理。这一过程是多个任务（task）并发执行的，针对每个需要执行的调优任务，具体的执行流程为：

1. 执行任务文件（task.yaml），创建 task 资源，由 taskcontroller 进行维护和管理。随后，taskcontroller 基于 control/collectagent 生成 control/collectstep，并在整个任务流程中，持续观测 control/collectstep 的状态。
2. controlstep 获取当前状态的参数取值，collectstep 等待应用程序执行一段时间，并获取观测数据；
3. 调优模块推荐下一轮的参数取值，controlstep 执行应用程序的更新；
4. 等待服务执行；
5. collectstep 获取当前参数在运行时的观测数据；
6. 返回步骤 3。

3.3 调优模块

调优模块作为一个独立的算法服务部署在集群当中，主要功能包括：

1. 维护着集群中各个调优任务的状态信息以及离线辅助训练得到的先验规则，包括：当前及历史的参数、资源配置信息，当前及历史的性能观测数据，离线负载知识库记录的应用-负载标签及识别模型，离线参数知识库记录的应用-负载标签及其对应的优化配置等。
2. 持续监听固定端口，等待与控制模块通过远程过程调用（Remote Procedure Calls）的方式进行交互。通讯内容主要包括迭代调优过程中的的配置推荐和性能反馈。除此之外，通讯内容还包括唯一的标识信息，以保证有状态服务器正确运行并行任务：每一轮次的迭代中，通

过 task id（任务序列号）和任务当中 step id（步骤序列号）进行校验，保证配置推荐和性能反馈的正确匹配。

3. 调优模块内部，面向两种不同的调优场景，单一应用程序调优和微服务场景调优，实现两种不同的优化策略，分别为离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法。

3.3.1 应用程序调优

针对应用程序调优，本研究将整个问题划分为两种使用场景：离线训练辅助和在线参数调优。

离线辅助训练指的是：本系统可在脱离生产环境的离线阶段，预先针对不同种类的典型应用，以及同一应用的不同压力程度和请求分布的负载，进行参数调优的尝试，构建参数调优的先验规则。具体包括：

（1）负载知识库：记录应用-负载标签及识别模型；

（2）参数知识库：记录应用-负载标签及其对应的优化配置。

当有负载到达集群的时候，基于上述先验规则，根据系统的工作负载和环境特征推荐初始参数调优范围，该范围便是待搜索的配置参数空间。

在线参数调优指的是：不阻塞应用程序的正常运行，在应用运行过程中进行参数调优或推荐的过程，本研究提供了相应的算法接口从而实现应用程序的参数优化。结合离线辅助训练，可以更快速获取优化的参数配置。

3.3.2 微服务调优

现实业务场景当中，各应用的实际使用，往往是以微服务形式进行相互间结合的。与单一应用程序的调优不同，此处面临的两个挑战是如何联合资源分配和参数调优，即：

1. 如何保证在每次资源分配的迭代时其参数是较优的，而非可能存在不合理默认项的默认配置；
2. 如何在保证达到 SLO 后，能找到使用最少资源的资源分配以及较优的参数取值。

针对挑战 1，先前的资源分配工作将资源分配作为参数空间，针对调优目标，例如用户定义的 SLO 或一些性能指标，包括吞吐率、平均延迟等，来进行迭代优化。但是在资源分配和参数配置的调优联合的场景下，如果将应用程序参数和资源混合作为模型的参数空间，沿用先前的优化方法则会带来巨大的计算代价：首先，应用参数为资源分配带来的搜索空间是指数级增长的，这使得优化方法的迭代次数增长巨大，这在在线优化的场景下是不可接受的；另一方面，参数和资源混合调优会造成难以寻找最优组合的问题，由于不同的资源分配下，最优的参数并不一致，在这种情景下资源和参数在参数空间中扮演的角

色是一致的，混合调优并不能证明迭代出的组合在其对应的资源分配中是最优的。

针对挑战 2，先前工作[22]对资源分配工作所使用的资源情况进行了如下分类:

1. 动态资源类型：针对动态资源的分配工作，并不关心集群中的资源限制，通过指定资源的扩缩容以满足 SLO；
2. 固定资源类型：针对固定资源的分配优化工作，即在某一个固定资源集群中进行资源的重新分配以满足 SLO。

可以看到，无论是动态资源类型，还是固定资源类型的工作，先前的优化算法中，并没有解决如何在满足 SLO 的同时，寻找最节省资源的资源参数策略，这也是面临的挑战之一，同时是潜在的思路。

为应对上述挑战，在微服务场景中, 本研究提供了一个参数-资源联合调优方法及资源下降算法, 用于解决微服务场景中参数资源混合调优问题，以及在微服务场景中如何选择最少的资源总量满足 SLO 以达到节省资源的目的。

3.4 参数资源管理模块

参数-资源管理模块进行参数以及资源的扩缩容以及参数的修改，提供了可以自定义的参数以及资源修改接口，可以根据每个任务的区别（例如需要修改不同的资源，以及不同应用程序参数的修改方式）来自定义需要修改的内容。在调优过程中，控制模块发送算法模块推荐的参数资源，由参数-资源模块进行对应的修改，随后进行下一轮数据的收集。

3.4.1 参数修改和资源修改的异同

与单一的资源修改不同，由于云场景中往往存在着各式各样的应用程序，存在的参数不尽相同；并且其参数修改的方式也存在着各式各样的差异。针对这一问题，参数-资源管理模块被设计成了一个插件系统，提供了不同参数类型（运行时、配置文件、环境变量等）的修改接口，可以通过编写不同应用的插件文件，调用修改接口，从而达到对各类应用程序参数修改的支持。

3.4.2 参数-资源模块的适配

在具体对每个应用程序进行适配时，本研究基于 Kubernetes 中 pod 接口开发了四种不同的参数修改接口，如表 3-2 所示，分别提供了 pod 内部的配置文件、命令行参数、环境变量以及参数热更改的接口。这些接口的优点在于可以在不修改用户本身容器镜像内容的情况下，直接进行参数修改。

表 3-2 参数修改接口概述

Table 3-2 Overview of Parameter Modification Interface

名称	功能
ConfigFilePupdater	修改应用配置文件中的配置
CLIParaPupdater	修改应用启动时命令行参数
RuntimeCLIPupdater	修改应用运行时配置
EnvPupdater	修改环境变量

3.5 观测模块

观测模块收集的指标主要包括两种，分别是：性能指标和状态指标。

1. 性能指标，即优化目标，包括延迟、吞吐率等。由实际的任务需求（task.yaml）中进行定义。在调优流程中，控制模块在需要返回给调优模块观测数据时，与观测模块进行通信，由观测模块获取对应的数据返回给控制模块。针对不同类型的性能指标的获取方式不同，生成或使用相应的 collectagent，执行 collectstep 收集性能指标。
2. 状态指标，即容器在运行负载时的状态特征。这部分的数据，通过 cAdvisor[78]进行收集，其部署为 kubelet 组件中的守护进程，负责收集、聚集、处理并导出运行中容器的信息。这些信息能够包含容器级别的资源隔离参数、资源的历史使用状况、反映资源使用和网络统计数据完整历史状况。具体包括：
   (1) CPU 使用度量：容器的总 CPU 使用时间、用户空间和内核空间的 CPU 时间、过去一段时间的 CPU 负载平均值等。
   (2) 内存使用度量：容器的总内存使用量、容器的工作集大小、内存缓存的大小、内存页交换的数量等。
   (3) 网络 I/O 度量：包括发送和接收的字节数、数据包的数量等。
   (4) 磁盘 I/O 度量：包括容器读取的总字节数、容器写入的总字节数、容器的总读取次数、容器的总写入次数等。
   (5) 磁盘使用度量：包括容器使用的文件系统的总字节数、容器可用的文件系统的总字节数等。

   这部分信息会导出至时序数据库，使用时由 collectagent 依据负载任务开始和结束的时间戳作为数组索引，从数据库获取相应数据，最终返回给控制模块。

3.6 工作流程概览

如图 3-2 所示，四个模块的工作流程始于用户提交的任务配置文件，控制模块通过执行任务配置文件（task.yaml），创建 Task 资源，启动整个调优流程。

Task 资源包括离线或在线任务，具体由 Task 控制器进行维护和管理。随后，Task 控制器基于任务配置文件，在代理仓库中选取适配的 control 和 collect 代理，生成 controlstep 和 collectstep，并在整个任务流程中，持续观测 controlstep 和 collectstep 的状态。

其中，controlstep 负责获取当前参数的取值以及执行应用程序的更新，具体情况参考参数资源管理模块介绍。collectstep 等待应用程序执行一段时间，并获取观测数据，具体情况参考观测模块介绍；

整个调优过程由控制模块管理，基于 gRPC 交互的方式，与调优模块进行交互，完成决策，具体情况参考通用模块介绍。控制模块获取决策内容，并根据具体任务需求，持续创建 controlstep 和 collectstep，每次操作对应一个 step。流程的执行终点，取决于任务配置文件（task.yaml）中的优化目标或优化终止阈值。

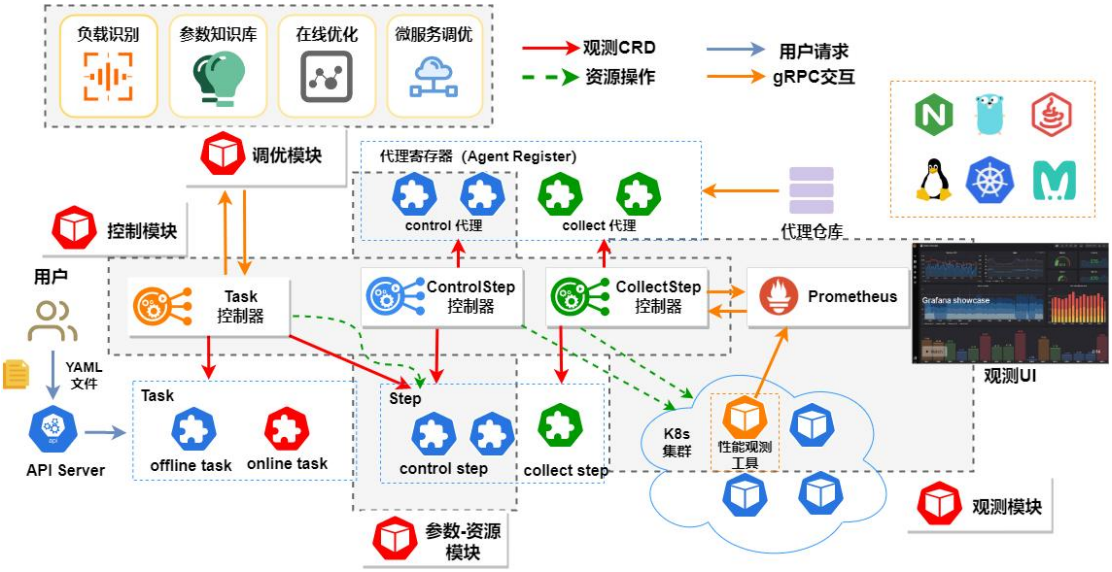

图 3-2 自动化调优系统工作流程概况

Figure 3-2 Overview of workflow for Automated Optimization System

3.7 本章小结

本章主要介绍本研究设计实现的面向容器编排平台的优化系统，详细阐释四个功能模块：控制模块、调优模块、参数资源管理模块、观测模块，包括其设计思路、功能逻辑以及工作流程。

第 4 章 离线-在线结合的识别优化方法

本章介绍离线-在线结合的识别优化方法，主要分为基于集成学习的离线辅助训练方法和基于机器学习的在线的参数调优方法，前者包括负载知识库的构建和参数知识库的构建，后者包括基于模型和无模型的调优流程。实现在负载到达集群时，有效利用历史经验或专家知识库等先验知识，进行准确识别和初始化的参数推荐，以避免初始化的不合理配置导致较差的性能表现，同时能够在较优的配置基础上进行快速的迭代优化，使得应用程序能够以更加符合当前任务负载，实现高性能表现。

4.1 离线辅助训练

如 3.3.1 所述，离线辅助训练指的是：本系统在脱离生产环境的离线阶段，预先针对不同种类的典型应用，以及同一应用的不同压力程度和请求分布的负载，进行参数调优的尝试，构建参数调优的先验规则。该先验规则具体包括负载知识库、参数知识库。

1. 负载知识库的核心是对应用程序的不同负载进行识别和分类，目标在于获取特征与负载类别的对应关系。具体地，面对不同类型的负载，收集应用的状态特征，根据相应的标签，进行训练和分类，在新负载到达集群时，能够快速识别负载类型。
2. 参数知识库则是在得到场景中的负载分类后，对适应负载的、性能有提升的参数进行推荐。具体地，根据待优化应用性能指标值最优时对应的参数配置生成一策略推荐库，记录每一待优化应用对应的类型及性能指标值最优时对应的参数配置。

4.1.1 负载知识库

应用负载知识库的构建目的是：在负载到达集群时，需要有效利用历史经验或专家知识库等先验知识，进行准确识别和初始化的参数推荐，以避免初始化的不合理配置导致较差的性能表现，同时能够在较优的配置基础上进行快速的迭代优化。

具体而言，需要利用负载运行时的各类状态信息，识别不同的应用类型以及具体负载类别。

4.1.1.1 负载知识概述

本研究中，应用程序的负载情况主要表示各种应用程序的不同压力程度和请求分布。压力程度一般指代客户端每秒钟请求数，较高的并发请求往往会给应用程序带来显著的处理压力，导致延迟增加、错误率提高。请求分布则表示

客户端请求类型的构成，以数据库性能基准测试工具 YCSB（Yahoo! Cloud Serving Benchmark）[62]提供的 MongoDB 的负载类型为例，共包括六种典型类型:

表 4-1 MongoDB 负载类型概述

Table 4-1 Overview of MongoDB load types

标签	类型	读 (Reads)	写 (Writes)	插入 (Insert)	扫描 (scan)
workloada	读写均衡型	50%	50%	0%	0%
workloadb	读多写少型	95%	5%	0%	0%
workloadc	只读型	100%	0%	0%	0%
workloadd	读最近写入记录型	95%	0%	5%	0%
workloade	扫描小区间型	0%	0%	95%	5%
workloadf	读写入记录均衡型	50%	0%	50%	0%

应用负载的识别本质上是一个分类问题。通过某种分类算法对训练数据集学习生成一个分类函数或分类模型，通过该函数或模型把数据记录映射到给定类别中的某一个，从而可用于数据预测。

训练数据集由一组训练样本构成，每个训练样本是一个由属性值或特征值组成的特征向量，而且每个训练样本还有一个类别标签。具体的样本形式为：$(attr_1, attr_2, \ldots, attr_n; label)$；其中$attr_i$表示属性值，$label$表示类别标签，对应不同类型或压力程度的负载。

关于属性值的选取，本研究采用 Google 开源的用于展示和分析容器运行状态的可视化工具 cAdvisor[78]，其能够收集、聚集、处理并导出运行中容器的信息。例如 CPU 使用率、CPU 时间片使用情况、内存使用量、磁盘读取字节数、磁盘写入字节数、磁盘 I/O 请求次数等。这些信息能够包含容器级别的资源隔离参数、资源的历史使用状况、反映资源使用和网络统计数据完整历史状况，这些信息在不同负载之间存在差异。

利用这些指标，在负载到达集群的时候，通过数据监测和处理，完成负载标签的识别，确认所属的负载类型，即构建一个特征与负载类别的对应关系。其工作流程如图 4-1 所示。

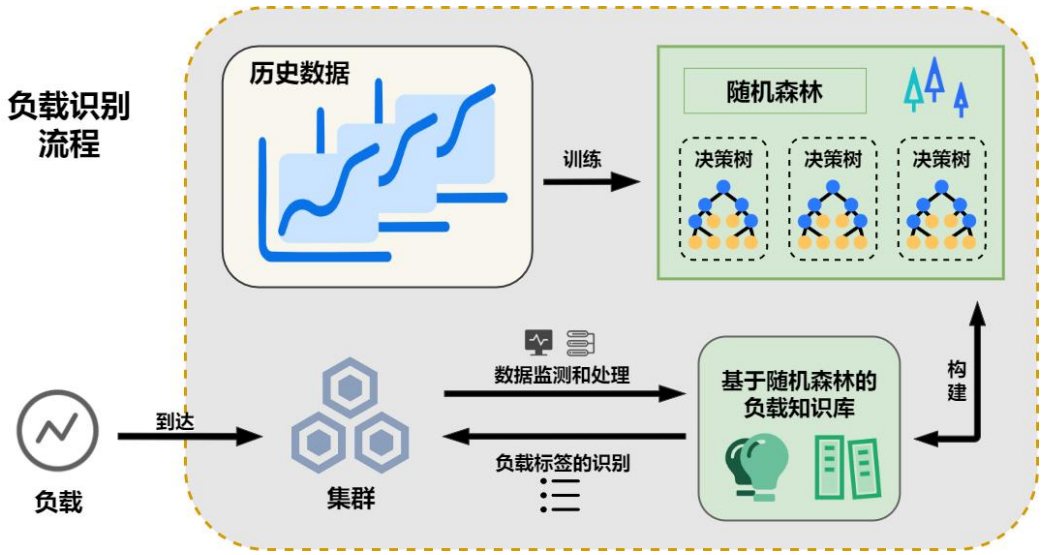

图 4-1 负载识别流程图

Figure 4-1 Flowchart for load identification

4.1.1.2 负载知识库的构建

实现负载知识库的构建，需要基于上述容器运行状态属性，运行不同的典型负载类型，获取样本数据，选择合适的分类方法对样本数据进行学习，以获得分类模型。本研究以 CART 决策树（Classification and Regression Tree）[79]为随机森林中的基础分类器，通过集成学习的方法进行分类，最终实现负载知识库的构建。

1. CART 决策树算法

决策树算法是应用最广泛的分类方法之一[80]。其中，CART 算法由 Leo Breiman 等人于 1984 年提出，是一种通用的决策树生成算法。对于分类问题，CART 使用基尼系数（Gini Index）作为划分准则，选择能够获得最小基尼指数的属性作为划分属性。

（1） 基尼系数

基尼系数$Gini(D)$表示集合D的不确定性，表示经分割后集合D的不确定性。基尼指数数值越大，样本集合的不确定性越大。

分类问题中，假设有K个类，样本点属于第k类的概率为p_k，则概率分布的基尼指数定义为：

$$Gini(P) = \sum_{k=1}^{K} P_k(1-P_k) = 1-\sum_{k=1}^{K} P_k^2$$

根据某个特征A的某个取值 A_i，将样本分为两个子集 D_1, D_2，则数据集D对于特征A的基尼系数为：

$$Gini(D,A)=\frac{|D_1|}{|D|}Gini(D_1)+\frac{|D_2|}{|D|}Gini(D_2)$$
$$=\frac{|D_1|}{|D|}(1-\sum_{k=1}^{K}(\frac{|C_{1k}|}{|D_1|})^2)+\frac{|D_2|}{|D|}(1-\sum_{k=1}^{K}(\frac{|C_{2k}|}{|D_2|})^2)$$

(2) 特征处理

CART 树是基于二叉树的模型，本研究面对的应用负载类型分类，属于多分类问题，因此需要对连续值和离散值作不同的特征处理。

① 连续值处理：连续特征的处理方式采用离散化。数据集D的特征A有m个取值，从小到大排列为：$(A_1,A_2,A_3,...,A_m)$，划分出$m-1$个取值点：$(T_1,T_2,T_3,...,T_{m-1})$，其中：$T_i=\frac{A_i+A_{i+1}}{2}$将小于$T_i$的样本划分到集合$D_T^-$，大于$T_i$的样本划分到集合$D_T^+$，计算其作为二元离散值时的基尼系数。

② 离散值处理：枚举离散特征的所有二元组合，选择基尼系数最小的进行分裂，剩余组合在接下来的分裂中继续计算。例如离散特征A有三个取值(A_1,A_2,A_3)，可以将其二元分类为三种情况：$(A_1)(A_2,A_3)$，$(A_1,A_2)(A_3)$，$(A_1,A_3)(A_2)$。计算这三种组合的二元基尼系数，如果最终选择的分裂组合为$(A_1)(A_2,A_3)$，则剩余的两个取值A_2和A_3会在接下来的分裂中继续参与计算。

2. 随机森林算法

随机森林是一种集成学习方法，它通过构建并结合多个决策树来进行预测。随机森林的基本思想是通过多数投票或平均的方式，将多个决策树的预测结果进行结合，从而获得更稳定、更准确的预测。本研究以 CART 决策树（Classification and Regression Tree）为基础分类器构建随机森林，具体构建细节包括：

(1) 自助抽样

针对随机森林中的多棵不同决策树，对于收集的容器状态数据和对应的负载标签，本研究进行有放回的随机抽样，生成多个新的训练数据子集。每个子集的大小通常与原始训练数据集的大小相同。尽管每个子集的大小与原始训练数据集的大小相同可能会导致部分重复样本的存在，但这种方法能够有效地保持数据特征的完整性和多样性，为模型训练提供稳定可靠的数据基础。

(2) 通过交叉验证设置决策树的最佳深度

随机森林中 CART 决策树的深度，也就是树中最长路径的长度，对模型的性能和复杂度有重要影响。如果决策树的深度过大，模型可能会过于复杂，这可能导致模型在训练数据上表现得很好，但在新的、未见过的数据上表现得很差。如果决策树的深度过小，模型可能会过于简单，不能捕捉到数据中的所有模式和关系，这可能导致模型在训练数据和新数据上的表现都不好。

本研究通过交叉验证，为不同负载分类决策树选取最匹配的深度。首先，定义候选的最大深度集合，即决策树深度 max_depth 的三个可能值：*[4, 7, 10]*。遍历所有可能的深度取值，创建相应的 CART 决策树分类器实例，对每一种最大深度取值，使用交叉验证（Cross-Validation）[81]来评估其性能。本研究具体综合验证效果和时间成本，选择 $5-Fold$ 交叉验证，将收集到的状态指标数据分成 5 份，模型会被训练 5 次，每次都用 4 份数据进行训练，剩下的 1 份数据进行验证。

完成所有的深度取值的评估，则返回最佳的最大深度取值和对应的准确率，基于此构建随机森林。

(3) 基于 CART 决策树和随机森林的负载识别算法

负载的自动识别，如果仅根据一次识别的结果就进行资源的参数配置的策略推荐，必然会造成系统的振荡调整，严重影响应用程序的性能。所以必须根据一段时间间隔内的多次采样数据，将数据的取值或差值，进行平均的识别结果。例如，cAdvisor 对容器的默认采样的频率为 1 次 / 秒，在识别时刻前的 5 分钟时间内，共有 180 次快照，对这 180 次数据进行处理和识别。其中，对于即时状态属性，例如 container_cpu_load_average_10s，反映过去 10 秒钟 CPU 的平均压力，保留原始数值；对于积累属性，例如 container_network_receive_packets_total，反映容器接受的总数据包的数量，则进行相邻作差。在此基础上，进行缺失值和异常值的处理。最后，根据整批次的数据，逐一通过随机森林进行分类，最终选择被识别次数最多的类别标签作为输出。

综上所述，基于 CART 决策树和随机森林的负载识别算法如 4-1 所示。

算法 4-1 基于 CART 决策树和随机森林的负载识别算法

1. **Input**：历史数据集合$hist_data$，对应负载标签$hist_label$，预处理后的待识别的状态属性集合$curr_datas$，森林中树的数量$n_estimators$，自助抽样选项$bootstrap$，测量分割质量的函数$criterion$，树的最大深度集合max_depth，剪枝的复杂性参数ccp_alpha，交叉处理次数$fold$
2. **Output**：识别出的负载类别标签$label$
3.
4. // 定义字典类型的变量，包含max_depth的候选取值，此处为[4, 7, 10]
5. $dic = \{'max\_depth': max\_depth\}$

```
6.
7.  // 基于 YAML 配置文件中调优算法相关字段，初始化随机森林模型
8.  randomforest = RandomForestClassifier(
9.    n_estimators = kwargs.get("n_estimators", 4), 树的数量
10.   bootstrap = kwargs.get("bootstrap", True), 自助抽样
11.   criterion = kwargs.get("criterion", "gini"), 基尼系数
12.   ccp_alpha = kwargs.get("ccp_alpha", 0.0005), 剪枝处理)
13.
14. // 交叉验证max_depth最佳取值，此处fold为 5，表示使用 5 折交叉验证
15. crossvalidationModel = GridSearchCV(randomforest, dic, cv = fold)
16. crossvalidationModel.fit(hist_data, hist_label)
17. best_randomforest = crossvalidationModel.best_estimator_
18.
19. // 识别分类
20. for each curr_data in curr_datas do
21.   // 逐一分类
22.   add best_randomforest.pridict(curr_data) to outputs
23. end for
24.
25. // 获取最匹配的类别标签
26. outputs = Counter(outputs).most_common(1)
27. label = outputs[0][0]
28.
29. return label
```

4.1.2 参数知识库

4.1.2.1 参数筛选

基本参数筛选，本研究参照应用的官方文档，将可以调整的参数尽可能多地纳入到调整范围内。此时，成百上千的参数优化空间极大，本研究选择使用最小绝对收缩和选择算法（Least absolute shrinkage and selection operator，LASSO）[63]来进行线性回归分析，从而计算出哪些参数对性能的影响最大。为保证应用在配置更改后的可用性，本研究基于默认配置的值，做小范围的随机微调。此时，不追求性能表现，仅观测参数修改及其造成的性能表现变化。最终根据回归系数排序，选择一批主要参数进行调优过程。具体步骤包括：

1. 将收集到的参数取值集合表示为矩阵形式的自变量，其中每一行代表一个样本，每一列代表一个参数的取值。n 个参数，m 次取样，则该矩阵规模为 $n \times m$。

2. 将性能结果作为目标变量。同上，m 次取样，则矩阵规模为 $1\times m$，与自变量的行数保持一致。
3. 对参数和性能结果进行标准化处理，以确保它们具有相同的尺度。
4. 利用 LASSO 模型拟合数据，使用带有 L1 正则化项的损失函数，该损失函数由两部分组成：数据拟合误差和正则化项。其中，数据拟合误差用来衡量模型对训练数据的拟合程度，而正则化项则用来惩罚模型的复杂度。本研究中，设置正则化系数 $alpha = 0.1$。
5. 观察拟合后得到的参数系数大小，系数绝对值较大的特征对应的参数对性能结果影响较大。其中，绝对值为正的系数表示参数与性能表现为正相关关系，绝对值为负的系数表示参数与性能表现为负相关关系。
6. 根据系数的大小排序，设置一定的阈值，保留影响较大的特征作为重要特征。

4.1.2.2 参数知识库的构建

基于目前主流的开源应用程序，例如 Nginx、Apache、PostgreSQL、Redis 等，本研究对其参数与性能关系进行了分析，具体的训练方式则是使用不同的基准测试，例如一些测试工作使用到的 ApacheBench[64]、Sysbench[65]、Pgbench[66]等基准测试工具，配置不同类型或者压力程度，使用下一节所描述的搜索算法寻找对性能最优的参数组组合。为了保证服务的可用性，本研究对不同的应用程序的参数修改范围做了相应的选择，以保证整个服务的正确性。

在资源方面，考虑到算法与业务场景是非耦合的，本研究认为，资源扩缩容和应用程序内部的参数在算法看来是一致的，即都可以作为参数的一部分进行学习和搜索，但考虑到微服务调优中的多层次资源-参数联合调优，此处将资源分配作为负载标签，仅探索不同参数的配置情况。

在符合规定 SLO 的情况下，即在当前配置满足目标性能要求的情况下，以不同的压力测试来模拟真实的使用场景并通过不同的参数、资源分配来尽可能的覆盖所有的性能范围，以便在实际的优化过程中快速达到当前资源配给情况下的最佳性能表现，具体过程如算法 4-2 所示。

算法 4-2 基于 LASSO 算法和参数调优的参数知识库构建算法

1. **Input**：默认性能表现$default_pref$，性能数据集合$perf_data$，对应参数列表$paras$，对应参数配置$para_data$，正则化系数$alpha$，求解器迭代次数max_iter，优化停止阈值tol，参数重要性筛选阈值$threshold$
2. **Output**：参数推荐知识库字典$reco$
3.
4. // 对数据进行标准化处理，以确保其具有相同的尺度

```
scaler = StandardScaler()
para_data scaled = scaler.fit_transform(para_data)
perf_data scaled = scaler.fit_transform(perf_data)
// 初始化lasso回归模型，拟合模型并获取参数的重要性系数（绝对值）
lasso_operator = Lasso(alpha = 0.1, max_iter = 2000, tol = 0.0001)
lasso.fit(para_data_scaled, perf_data_scaled)
impos = np.abs(lasso.coef_)

// 创建参数名称和重要性的映射，并根据重要性对参数进行排序和筛选
para_importance = {} // 初始化排序后的参数重要性字典
for each para, impo in paras, impos do
  if impo > threshold then // 只添加那些值大于阈值的键值对
    para_importance[para] = impo
  end if
end for

sorted_para_importance = sort(para_importance)
reco = {} // 初始化参数知识库字典
reco[default] = default_value
reco[optimal] = default_value
reco[para_importance] = sorted_para_importance
reco[para_optimal] = sorted_para_importance

while collecting do
  curr_perf = getperf()
  curr_para = getpara()
  if curr_perf > reco[optimal]
    reco[optimal] = curr_perf
    for each para in paras
      reco[para_optimal][para] = curr_paras[para]
    end for
  end if
end while

return reco
```

4.2 在线参数调优

如 3.3.1 所述，在线参数调优指的是：不阻塞应用程序的正常运行，在应用运行过程中进行参数调优或推荐的过程，本研究提供了相应的算法接口从而实现应用程序的参数优化。结合离线辅助训练，可以更快速获取优化的参数配

置。

4.2.1 基于模型的调优

基于模型的调优指的是基于代理模型，通过建立一个近似的性能评估函数来代替实际性能表现函数。其中，自变量是参数配置，因变量为应用或系统的性能表现。使用已生成的配置样本和对应的性能数据，训练机器学习模型来预测配置与性能之间的关系。具体地，本研究使用基于高斯过程回归[83]的贝叶斯优化[84]，除了贝叶斯优化算法本身高效性、自适应性、对噪声和不确定性的鲁棒性、并行化支持、可解释性等优势，本研究在此基础上进行改进，进一步提高了调优下限并加快收敛速度。

4.2.1.1 模块构成

（1）初始化策略：始化策略是指在开始优化过程之前，选择初始参数组合的策略，初始参数组合的选择对于整个优化过程的效率和性能有重要影响。本研究基于低差异序列（Low-discrepancy sequence）采样[82]来代替传统的随机序列生成方法，用于生成高质量、均匀分布的随机采样点，这样的取样方式可以避免贝叶斯优化算法陷入局部最优解，避免算法陷入不良区域，导致无法找到全局最优解。随机采样也可以帮助算法快速探索潜在较好的区域，在建模过程中帮助贝叶斯优化算法更好地估计目标函数的潜在曲面，更准确地指导参数搜索，从而加速整个优化过程，同时改善贝叶斯优化算法的收敛性，如果初始参数组合距离全局最优解较远，可能需要更多的迭代轮次才能收敛。

本研究具体使用的低差异序列采样方法包括 Halton 序列、Sobol 序列、Hammersley 序列和拉丁超立方抽样等。以下是基于低差异序列采样的配置生成的一般步骤：

a. 确定参数的取值范围和类型：首先，需要定义每个参数的取值范围和数据类型，包括浮点数、整数或离散数据等。

b. 选择合适的低差异序列生成算法：根据具体的应用需求和超参数类型，选择适合的低差异序列生成算法。

c. 对每个参数进行映射和缩放：根据参数的取值范围，对每个参数进行映射和缩放操作，将其转化到合适的范围。例如，对于浮点数参数，可以将其映射到[0, 1]的范围。

d. 使用低差异序列生成采样点：利用所选的低差异序列生成算法，生成一系列采样点。每个采样点表示一个配置。如果采样的参数配置与历史记录重复，则舍弃掉本轮选择，重新随机选择本轮参数配置取值。

e. 反映射还原：将生成的采样点反映射到原始超参数的取值范围，以得到最终的配置。

另一方面，本研究的初始化策略可结合离线辅助训练获取的负载知识库和参数知识库的先验信息。如果不依赖已有的调优经验，仅仅靠随机序列生成方法或低差异序列采样，在初始采样数量比较少的情况下，可能导致并没有采样到较优的配置。本研究在线参数调优过程中，可以结合离线辅助训练的先验知识。当新负载到达集群的时候，获取容器运行状态数据，识别负载类别，根据识别结果获取相应负载标签的较优的参数配置，并将该配置加入到初始化策略中。除此之外，应用的默认配置会一并添加到其中。这样，可以保证优化过程中，各个参数存在较合理的不劣于默认配置的选项。在迭代次数较少的情况下，提高了调优下限，使得初始参数组合的选择更加合理，更符合调优实际。

（2）代理模型：代理模型通过建立一个近似函数来代替目标函数（性能评估函数），使用已生成的配置样本和对应的性能数据，训练机器学习模型来预测配置与性能之间的关系。通过拟合已有的目标函数观测值，提供一个近似模型，避免通过大量采样和计算直接优化目标函数带来的高昂计算成本。训练好的模型能够快速预测不同参数配置组合下的目标函数值，从而加速优化过程。

具体地，本研究选择基于高斯过程的代理模型。高斯过程模型是一种概率模型，用于建模目标函数的不确定性。它通过拟合已观察到的超参数和目标函数值的联合分布来进行预测。高斯过程模型可以提供预测的均值和方差，从而提供对预测结果的不确定性估计。

（3）采样函数：根据已训练的模型，在参数空间中采样一组候选配置。这些配置被认为有较高的潜在性能，可以在接下来的优化步骤中进一步评估。在优化过程中，选择下列采样函数用于选择下一个要评估的超参数组合。具体来说，采样策略会在未探索的区域中选择潜在有价值的样本点，或者在已知有高估计值的区域进行进一步探索，以尽快找到最优解。代理模型提供对目标函数的不确定性估计，可以用于确定下一个要采样的样本点，采样函数根据代理模型的预测和不确定性信息，评估每个候选样本点的价值和信息量，选择具有最高价值的样本点进行采样。代理模型在采样点选择的过程中起到了指导作用，使本研究的在线参数调优算法更加高效地探索参数空间。具体包括：

a. 概率提升（Probability of Improvement）：概率提升采样函数计算每个超参数组合达到或超过当前最优结果的概率。它根据代理模型的预测分布，估计目标函数超过阈值的概率，并选择具有最高概率提升的超参数组合。概率提升函数更加关注达到特定目标的概率，适用于那些需要达到一定目标水平的优化问题。计算公式如下：

$$impove(x) = best(x^*) - \mu(x) - \varepsilon$$

$$PI(x) = \varphi(\frac{impove(x)}{\sigma(x)}),\ \sigma(x) > 0$$

其中$best(x^*)$表示当前最优配置x^*对应的性能值，$\mu(x)$表示配置x对应

的后验期望值，ε表示一个大于等于 0 的小数，用以调节性能的波动值。$impove(x)$表示当前配置x的提升量，$\sigma(x)$表示配置x对应的后验期标准差，$\varphi(\cdot)$表示累积分布函数。

b. 期望提升（Expected Improvement）：期望提升采样函数根据代理模型的预测结以及差异的不确定性。期望提升函数选择具有最大期望改进的超参数组合作为下一个要评估的点。这样可以在探索未知区域的同时，倾向于选择那些有望带来更大改进的超参数。计算公式如下：

$$EI(x) = impove(x) * \varphi(\frac{impove(x)}{\sigma(x)}) + \sigma(x) * \rho(\frac{impove(x)}{\sigma(x)}), \sigma(x) > 0$$

其中 $impove(x)$表示当前配置x的提升量，$\sigma(x)$表示配置x对应的后验期标准差。

$\varphi(\cdot)$表示累积分布函数，$\rho(\cdot)$概率密度函数。

c. 下置信界（Lower Confidence Bound）：下置信界采样函数在代理模型的预测结果中综合考虑目标函数的期望和不确定性。它基于置信界的概念，在最优化和探索之间进行平衡。下置信界函数选择具有最高置信界的超参数组合作为下一个要评估的点。这样可以在已知较好结果的区域进行探索，并同时考虑目标函数的不确定性。计算公式如下：

$$LCB(x) = \mu(x) - \sqrt{\frac{\alpha * log(n * m^2)}{\beta}} * \sigma(x)$$

其中$\mu(x)$表示配置x对应的后验期望值，$\sigma(x)$表示配置x对应的后验期标准差，α，β表示调节系数，n表示优化目标数，m表示已经评估的观测总量。

d.每秒期望提升（Expected Improvement per Second，EIps）是一种用于优化问题中的指标，用于衡量在单位时间内期望改进的程度。EIps 可以用来评估候选配置的优劣。它结合了两个关键因素：期望改进和时间。时间是时序代理模型的后验期望值。计算公式如下：

$$EIps(x) = \frac{EI(x)}{\mu_t(x)}$$

其中$\mu_t(x)$表示配置x对应的后验时间期望。

e.每秒概率提升（Probability Improvement per Second，PIps）是一种用于优化问题中的指标，用于衡量在单位时间内概率改进的程度。它结合了两个关键因素：概率改进和时间。计算公式如下：

$$PIps(x) = \frac{PI(x)}{\mu_t(x)}$$

其中$\mu_t(x)$表示配置x对应的后验时间期望。

f.约束期望提升（Constrained Expected Improvement，CEI）：是一种用于在约束优化问题中衡量候选解的指标。它结合了期望改进和约束条件，用于评估候选解在满足约束的前提下的优化潜力。在约束优化问题中，除了优化目标外，还存在一些约束条件需要满足。候选解必须在约束条件下具有良好的性能。CEI通过同时考虑目标函数的期望改进和约束条件的满足程度来评估候选解的优劣。计算公式如下：

$$CEI(x) = EI(x) * \prod_{i=1}^{n} \varphi(-\frac{\mu(x)}{\sigma(x)})$$

其中$\mu(x)$表示配置x对应的后验期望值，$\sigma(x)$表示配置x对应的后验期标准差，$\varphi(\cdot)$表示累积分布函数。

4.2.1.2 调优流程

基于模型的调优的具体流程如下：

（1）确定优化目标：即针对某一个优化目标进行优化，在应用程序调优中，本研究一般会设置为吞吐量或延迟，而在微服务优化时则是相应的用户定义SLO，以及实际资源总量。

（2）参数空间的构建：算法的参数空间基于应用配置文件，获取可修改的参数列表以及其默认配置，在默认配置基础上，构造包括离散型参数和连续性参数的参数空间。每次的新采样，将作为优化过程的参数推荐。

（3）建立代理模型：本研究使用高斯过程回归作为代理模型来进行参数的在线优化，高斯过程回归的优势在于其提供了一种理论上合理的方法来权衡和探索未知参数空间，以及其相较于离线模型较为高效的迭代效率，这对在线优化十分重要。

（4）选择优化起点：随机采样一定数量的初始点，构建贝叶斯优化器是最简单的实现方式。考虑到使得初始采样尽可能分散、均匀，本研究采用低差异序列采样。

（5）循环优化过程：在每次迭代中，执行以下步骤：

a) 使用代理模型和采样策略，在已知的参数配置中寻找一些有可能取得更好结果的点进行评估，同时试图发掘搜索空间中尚未探索的区域，选择下一个参数配置选项组合进行评估；
b) 根据上一步选择的配置选项组合，更新整个应用程序，随后继续等待观测下一轮性能指标；
c) 记录性能指标，动态调整探索（Exploration）和开发（Exploitation）策略，并将其用于更新代理模型；
d) 根据代理模型的预测，选择下一个配置选项组。最终会在性能较优的参数配置组合附近进行更密集的搜索，以寻找更加精细的最优解，

直到找到可接受的最优解或者达到预设的时间上限。

具体地，如算法 4-3 所示。

算法 4-3 基于高斯过程回归的贝叶斯优化的参数迭代调优算法

```
1. Input：参数配置信息parameter_configuration，先验配置prior_paras，初始化采样配置数量initial_num，采样函数acquisition，初始化采样策略sample，代理模型surrogate，调优最大迭代次数max_iterations
2. Output：推荐参数配置组合new_configuration

3. // 基于参数配置信息生成参数配置空间
4. space = generateConfigurationSpace(parameter_configuration)
5. // 基于 YAML 配置文件中调优算法相关字段，初始化贝叶斯调优器
6. advisor = bayesianOptimizer(config_space = space,// 参数配置空间
7.         prior_paras = prior_paras，// 先验配置
8.         n_initial_points = initial_num，// 初始化采样配置数量
9.         initial_strategy = sample_type，// 初始化采样策略
10.        surrogate_type = surrogate，// 代理模型
11.        acquistion_type = acquisition) // 采样函数
12. // 迭代调优
13. for each iteration in max_iterations do
14.    // 计算新的推荐参数配置组合
15.    new_configuration = advisor.getNewConfiguration()
16.    // 基于推荐参数运行基准测试，并计算评价指标
17.    performance = calculateEvaluation(new_configuration)
18.    // 更新调优器历史数据
19.    advisor.updateObservation(observation(configuration = 
           new_configuration, performance = performance))
20. end for
21.
22. // 获取最佳性能的参数配置组合
23. return advisor.getBest().getOptimalValue()
```

4.2.2 无模型的调优

无模型的优化方法指的是直接在超参数空间中搜索最佳解，而不依赖于代

理模型。简单、直接，适用于小规模的超参数优化问题，计算开销较低。

具体的调优流程如下：

1. 确定优化目标，即针对某一个优化目标进行优化，在应用程序调优中，本研究一般会设置为吞吐量或延迟，而在微服务优化时则是相应的用户定义SLO，以及实际资源总量。

2. 参数空间的构建，算法的参数空间基于应用配置文件，获取可修改的参数列表以及其默认配置，在默认配置基础上，构造包括离散型参数和连续性参数的参数空间。每次的新采样，将作为优化过程的参数推荐。

3. 使用随机搜索或网格搜索，在参数空间当中，进行取样。为使得取样点尽可能均匀分布，采取多次取样的方式，每次取样结果计算最大最小距离。将多次取样结果当中，分散最均匀，即最大最小距离最大的一次取样，作为参数修改的样本，进行基准测试实验。

4. 在初始样本集中，找到具有最佳性能的点，并在该点周围，以坐标轴上最近的其余点的坐标为边界的有界空间中，采样的另一组点，以争取找到另一个具有更好性能的点。采样方式同样为随机搜索或网格搜索，并优先选择最分散的采样方式。

5. 如果找到新的性能更佳的点，则重复上述过程，在周围的有界空间中进行采样，递归执行以上采样步骤，直到在样本集中找不到具有更好性能的点或达到设定的迭代次数或时间。

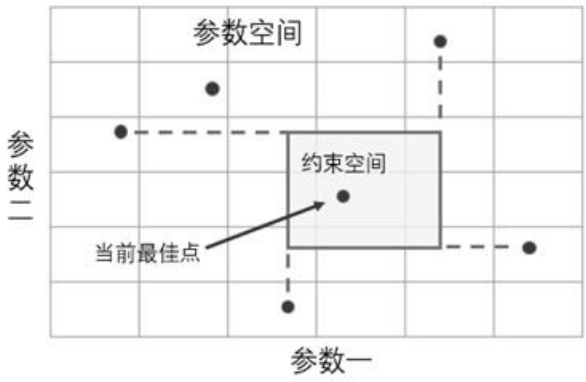

图 4-2 递归采样的示例图

Figure 4-2 Example diagram of recursive sampling

4.2.3 优化目标设置

本研究的优化过程通常使用基准测试的结果作为反馈，但这个反馈往往不是唯一的值，例如 Sysbench[65]等可以反馈数据库应用的吞吐率和延迟，最简单的方式是选择其中某一个值作为优化目标。但本研究保留不同的多目标优化策略接口，既包括接纳用户的自定义指标的优化，同时允许用户在不同的优化目标之间进行优化策略的选择。

如果用户有明确优化侧重，则可根据不同目标的重要程度，设定权重进行线性加权，将多目标优化转换为单目标的优化问题，即$\min_{x} \sum_{k=1}^{K} \lambda_k f_k(x)$，其中各个权值的关系为，$\{\lambda | \lambda_k > 0, i = 1,2...K, \sum_{k=1}^{K} \lambda_k = 1\}$。

或者，由用户提出一个优化目标值，$f^0 = (f_1^0, f_2^0, \ldots, f_K^0)$，使得每个目标函数都尽可能的逼近对应的目标值，所计算出的目标函数和实际值的差距为：$L(f(x), f^0) = \| f(x) - f^0 \|_2^{\lambda} = \sum_{k=1}^{K} \lambda_k (f_k(x) - f_k^0)^2, \lambda \in \Lambda^{++}$。这与机器学习中的损失函数类似，通过缩小该差距，便将多目标优化转换为单目标优化问题。

若没有具体的优化目标，可也在优化过程中维护一个最优值的记录，每次取值之后更新，在更新最优值的同时，更新过去每次取值的评分。这样可以在调优过程里，同时改善对于之前调优的评价。

4.3 本章小结

本章主要介绍离线-在线结合的识别优化方法，本方法能够做到基于历史经验或专家知识，构建参数调优的先验规则，有效识别负载压力并提供初始化推荐，结合在线调优流程，高效快速完成参数配置的优化。

第 5 章 资源-参数联合调优方法及资源下降算法

本研究认为当前资源分配工作并没有解决参数优化在资源分配时参数不当造成的性能浪费，故最在微服务调优过程中，本研究主要完成两个调优目标：①尽可能满足用户定义的 SLO；②在满足第一个条件下，尽可能的节省用户资源；基于此，本研究提出资源-参数联合调优方法及资源下降算法，结合进化算法，在满足用户 SLO 的前提下，兼顾资源和参数的优化，探索更低的资源占用和更高的并发支持，减少集群整体资源消耗，同时提高集群并发处理能力。具体地，首先介绍面对微服务调优区别于单一应用调优的两个挑战以及本研究的调优策略，其次阐述资源分配部分的变异操作函数和进化算法，最后介绍资源-参数联合调优方法及资源下降算法。

5.1 微服务调优策略

本研究通过以下思路来解决 3.3.2 中提到的两个挑战：

挑战 1：如何保证在每次资源分配的迭代时其参数是较优的，而非可能存在不合理默认项的默认配置。

核心问题是参数的加入使得整个调优系统的参数空间增长过大，从而难以快速迭代出较优的策略。为了解决这一问题，本研究采用多层次优化的逻辑，通过先分配资源随后进行参数调优的方式，来加速整个调优过程。

本研究假设微服务整体的性能在默认的正确配置下，是以总体资源分配的数量为主导的，故在进行资源-参数推荐时，本研究先选定对应的资源分配空间，随后进行参数调优。即首先通过进化算法对资源分配的迭代探索，选择较优的候选的资源分配方案，随后对其进行参数调优以达到最优的性能。这样可以有效避免微服务系统资源分配总量和子应用程序分配量在参数优化探索过程中，频繁变动造成的复杂优化状态。

挑战 2：如何在保证达到 SLO 后，能找到使用最少资源的资源分配以及较优的参数取值。

核心问题是如何在已经满足 SLO 的情况下，寻找一个可能存在的更加节省资源的参数-资源分配策略。为了解决这一问题，本研究提出了一种资源下降策略，在满足 SLO 后，通过对总资源量的逐渐收缩，从而找到满足 SLO 且资源总量使用较少的方案。

值得注意的是，资源可能存在的情况分为两类：

1. 初始化的资源分配能够满足 SLO；
2. 初始化的资源分配不能满足 SLO。

针对情况 1，本研究则可以通过资源下降算法，搜索满足 SLO 的情况下资

源使用较少的策略；针对情况 2，本研究则需要对其进行参数+资源优化从而使其达到 SLO，同时保留更进一步优化的可能。

5.2 基于进化算法的资源分配优化算法

进化算法（Evolutionary Algorithm）[85]是一种基于生物进化原理的优化算法，通过模拟生物进化过程中的选择、变异和遗传等机制来寻找最优解。本研究基于进化算法，用于优化资源分配策略。在这个具体的问题中，将微服务系统中的多个子应用程序，视作多个需要分配一定数量的资源的叶子节点。分配目标是找到一种最优的分配策略，以使每个叶子节点获得足够的资源，并且总资源浪费量最小。具体步骤如下：

1. 生成初始的资源分配策略：

   根据本研究场景中，微服务内的子应用程序数目（单元数）、初始分配资源数量、每个应用的最低分配数量等参数，使用公平分配策略或随机分配策略的方式生成一组初始的分配策略。这样做的目的是为了在优化过程开始时创建一个起点，还原默认情况下，微服务架构的分配情况，以便进化算法可以在此基础上逐步迭代改进。

2. 定义三种默认的突变操作函数：

   突变操作函数用于在进化算法中对当前的资源分配策略进行突变操作，以产生新的分配策略。这些突变操作可以随机交换两个叶子节点的资源，或微调每个叶子节点的资源数量。具体而言：

   a) 操作 1：首先从当前分配中选择一个叶子节点，然后随机选择另一个叶子节点。然后，将一个单位的资源分配从选定的节点转移到目标节点。

   b) 操作 2：首先计算当前分配中每个叶子节点的比例，然后根据一定的概率扰动这些比例，并生成新的分配策略。扰动的过程会严格遵循最小分配限制的资源量，避免扰动过程中出现某一应用分配资源量过低，导致整个微服务系统崩溃。

   c) 操作 3：对当前分配的每个元素应用指数变换，并引入基于比例的扰动来创建新的分配。这种方法旨在通过对原始分配值进行非线性调整来探索分配空间，同时确保新分配与原始分配在总量上保持一致。

   这些突变操作旨在增加分配策略的多样性，以帮助进化算法避免局部最优解，可以在保持总资源量不变的情况下，改变分配策略，以寻找更好的解。

3. 计算历史策略相对得分：

根据历史记录中的每个资源分配策略，获取微服务系统的 SLO 性能指标，计算其相对得分，表示该策略作为最优解的相对可能性，进而才历史记录中选择过去的最佳分配策略。

本研究没有直接将其将这些相对得分转换为概率值，而是使用指数函数进行转换，以便更突出高得分策略的概率。根据指数概率值的大小，计算每个策略被选中的概率。

具体地，对于某个固定的资源总量 C，资源分配策略列表 $L = \{L_1, L_2, ..., L_n\}$，其中 $\sum_{i=1}^{n} L_i = C$。列表中每个分配策略对应的性能指标得分为$V = \{V_1, V_2, ..., V_n\}$，将 V 的均值和标准差记为 v_{mean} 和 v_{std}。再对V进行$Z-score$标准化，计算得到 $V^{norm} = \{v_1^{norm}, v_2^{norm}, ..., v_n^{norm}\}$，其中任意$v_i^{norm} = (v_i - v_{mean})/v_{std}$。再将$V^{norm}$使用指数函数进行转换，得到 $E = \{e_1, e_2, ..., e_n\}$，将 E 的和记为 e_{sum}。最后，计算每个策略被选中的概率值 $P = \{p_1, p_2, ..., p_n\}$，其中$p_i = e_i/e_{sum}$。

4. 生成新的评估点以用于下一轮迭代：
   应用突变操作函数来生成新的分配策略，并将它们添加到待评估队列中。

具体地，如算法 5-1 所示。

算法 5-1 基于进化算法的资源分配优化算法

```
1. Input：微服务的子应用程序数（单元数）unit_nums，资源总量
resource_quantity，每个单元的最小分配资
源min_alloc_per_unit，每次变异操作选中的样本数nums
2. Output：变异后的推荐样本 new_samples
3.
4. // 基于微服务架构和资源总量生成初始的资源分配策略
5. get_initial_allocations(unit_nums,resource_quantity,min_
6.
7. // 定义三种不同类型的变异操作的随机选择函数
8. def get_mutation_op
9.   probs = [0.33, 0.67,1.0]
10.  chooser = np.random.random()
11.  if chooser < probs[0]
12.    return op1
13.  else
14.    if chooser < probs[1]
15.    return op2
```

```
    else
      return op3
  end if

// 迭代改善资源分配
while tuning do
  // 使用指数函数进行转换，以便更突出高得分策略的概率
  chosen_samples =
sample_according_to_exp_probs(samples, num)
  for each chosen_sample in chosen_samples do
    op = get_mutation_op
    new_samples = op(chosen_samples)

return new_samples
```

5.3 资源下降的联合调优算法

本研究的资源-参数联合调优方法及资源下降算法面向整体性能指标，以SLO（如吞吐率）大于M_{slo}为例，展开算法概述。当整体性能指标为 P99 延迟等优化目标为减小的指标时，SLO 即为小于M_{slo}。

整个微服务的服务数量为 n，某一次的资源-参数分配为 $a^R = [L^R, P]$，其中 $L^R = [l_1, l_2 ... l_n]$ 为 n个微服务所分配的资源, 其中 $R = \sum_{i=1}^{n} l_i$ 为当前分配资源的总量，$P = [p_1, p_2 ... p_n]$为 n 个微服务种对应的参数。本研究将初始化的分配记为 $a_0^R = [L_0^R, P_0]$，其中，L_0^R 和 P_0 为默认的完全公平资源分配和默认的参数配置。用 M 来表示某一个资源-参数分配观测到的指标，则有：

$$M_{i+j} = f(L_i^R, P_j) = f(a_{i+j}^R) \tag{1}$$

（1）式中，下标为对应的迭代次数，表示此时资源的分配已经进行了 i 次迭代，参数优化进行了 j 次迭代。

如前所述，对于任意固定的资源总量，通过资源分配优化得到的最佳结果，必然不劣于默认配置的表现性能。同理，在资源分配的基础上，通过参数优化得到的最佳结果，又必然不劣于资源分配优化得到的结果。因此，本研究假设所有的性能指标相对于资源、参数的值在一个范围内，则可以大致画出图像。

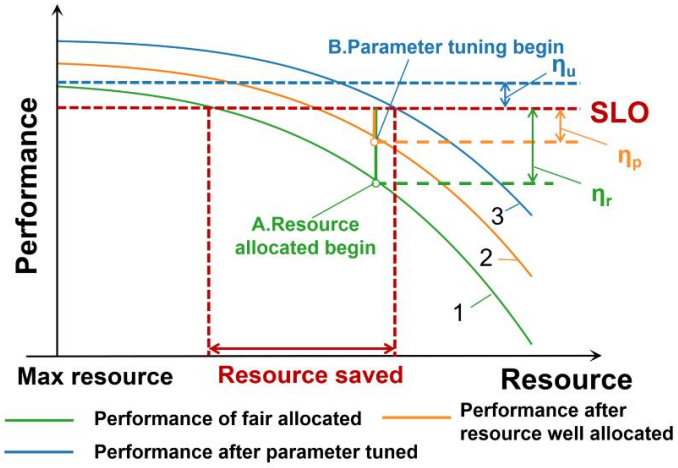

图 5-1 面向 SLO 的微服务系统优化概况

Figure 5-1 Overview of microservice system optimization for SLO

如图所示，曲线 1 为仅仅进行资源公平分配，而没有任何调优的曲线，本研究认为是整个范围的下界；曲线 2 为通过资源分配调优后到达性能最优的曲线；曲线 3 为在曲线 2 的基础上，进行参数调优后得到的最优曲线。随后，如图中所示，取一个相较于 SLO 的参数待调整范围 η_p，以及资源待调整范围 η_r，且 $\eta_r > \eta_p$。随后，本研究对两种可能出现的情况分别展开讨论：

① 初始化的资源分配满足了 SLO；

② 初始化的资源分配不能满足 SLO；

首先针对情况 1, 本研究给出如下策略：在情况 1 中，默认的资源-参数分配的情况下，取最大资源总量 R，存在：

$$M_{slo} < M_0^R = f(a_0^R) \tag{2}$$

此时整体的资源可以被视作过剩状态，需要对资源进行减少，以达到节省资源的目的。本研究中，设置每次收缩的资源数 δ_R，以及每次微调的资源数 δ_r，且 $\delta_R \gg \delta_r$。

步骤一，首先以 δ_R 的的粒度对当前的资源进行收缩，得到收缩后的资源数 R'，直到当前的性能指标满足（图中点 A）：

$$M_0^{R'} < M_{slo} - \eta_r \tag{3}$$

步骤二，随后对资源以 δ_r 的粒度进行增加，直到 R''状态，使得R''满足（图中点 B）：

$$M_{slo} > M_0^{R''} > M_{slo} - \eta_r \tag{4}$$

步骤三，随后对于当前的资源进行分配优化, 基于实时反馈的 M，进行资

源分配，即寻找一个 $a_i^{R''}$，使得：

$$M_{slo} > f(a_i^{R''}) = f(L_i^{R''}, P_0) > M_{slo} - \eta_p \quad (5)$$

超过M_{slo}，则退回步骤一。若在设定的迭代时间和轮次内无法找到，则退回步骤二。

步骤四，此时公式中的 P_0 表示当前的参数依旧为默认参数. 此时对其参数进行优化，得到性能最优的参数组合 P_j，此时性能为：

$$M_{i+j}^{R''} = f(a_{i+j}^{R''}) = f(L_i^{R''}, P_j) \quad (6)$$

步骤五，此时得到了当前在 R''上最优的性能 $M_{i+j}^{R''}$，取一个最小迭代粒度 m，此时分为以下情况：

1. 当 $M_{i+j}^{R''} < M_{slo}$ 时，当 $\delta_r > m$ 时，使 $\delta_r = \delta_r - m$，随后 $R'' = R'' + \delta_r$，若 R''大于资源总量 R 时，停止，视为达到优化极限。否则，回到步骤三；
2. 本研究设定一个上限阈值 η_u，当 $M_{i+j}^{R''} > M_{slo} + \eta_u$ 时，认为此时还有节省资源的空间，使 $\delta_r = \delta_r - m$，随后 $R'' = R'' - \delta_r$，当 $\delta_r <= 0$ 时，停止，视为达到优化极限。否则，回到步骤三；
3. 当 $M_{slo} < M_{i+j}^{R''} < M_{slo} + \eta_u$ 时，停止。此时，认为迭代到了最优的参数组合以及最节省资源的分配方式。

具体地，资源下降的联合调优算法如算法 5-2 所示。

算法 5-2 资源下降的联合调优算法

1. **Input**：服务水平目标 SLO 取值M_{slo}，资源优化潜在范围 η_r，参数优化潜在范围 η_p，性能上限阈值 η_u，当前分配资源量 R，每次收缩的资源数 δ_R，每次微调的资源数 δ_r，资源调优最大迭代次数$res_iterations$，参数调优最大迭代次数$para_iterations$
2. **Output**：优化后的性能$curr_perf$和资源占用R
3.
4. ***while*** *tuning* ***do***
5. ***while*** $curr\_pref > M_{slo} - \eta_r$ ***do***
6. $R = R - \delta_R$
7. ***while*** $curr\_pref < M_{slo} - \eta_r$ ***do***
8. $R = R + \delta_r$

```
loop1:
res_iter = 0
while curr_perf < M_slo − η_p do
  resource_tune
  res_iter += 1
  if res_iter = res_iterations
    res_iter = 0
    goto loop1
  end if
loop2:
para_iter = 0
while curr_perf < M_slo do
  parameter_tune
  para_iter += 1
  if para_iter = para_iterations
    para_iter = 0
    δ_r −= 1
    R = R + δ_r
    goto loop2
   end if
if curr_perf > M_slo + η_u
  para_iter = 0
  δ_r −= 1
  R = R − δ_r
  goto loop2
else
  break
end if

return curr_perf, R
```

针对情况 2，本研究给出如下策略：

步骤一，此时，即初始分配的资源无法满足 SLO，则整体的更新方式变为寻找当前资源总量下最优的资源-参数组合，首先进行资源分配的优化调整；

步骤二，当最优的资源分配能够满足$M_0^{R^{'}} > M_{slo} - \eta_r$时，则去判断其是否有进一步节省资源的机会；

步骤三，当最优的资源分配不能满足$M_0^{R^{'}} > M_{slo} - \eta_r$时，即可以认为此时的资源分配到了情况 1 中的步骤四情况，故本研究针对情况 2 的解

决方法时直接进入联合调优算法的步骤四部分，开始参数调优。

5.4 本章小结

本章主要介绍资源-参数联合调优方法及资源下降算法，阐述其如何应对微服务调优区别于单一应用调优的两个挑战，从而在满足用户 SLO 的前提下，兼顾资源和参数的优化，探索更低的资源占用和更高的并发支持，减少集群整体资源消耗，同时提高集群并发处理能力。

第 6 章 实验分析

本章介绍面向容器编排平台的自动化调优系统基于 Kubernetes 这一容器编排平台，在单一应用和微服务架构上进行调优任务的实验评估，包括实验环境、实验设计与评估。实验设计与评估部分，分别具体介绍应用程序调优和微服务调优，评估离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法的有效性。其中，微服务调优部分的实验包括两种调优目标，分别为资源优化实验和并发优化实验。

6.1 实验环境

本研究的实验均在总计 96 核 96GB 的集群中进行，操作系统的版本包括 Ubuntu 22.04 和 openEuler 22.03，Kubernetes 的版本为 v1.23.1，具体的实验环境如表 6-1 和 6-2 所示。

表 6-1 实验环境一

Table 6-1 Experimental Environment 1

硬件环境	CPU	Intel(R) Xeon(R) Platinum 8153 CPU @ 2.00GHz * 3
	内存容量	8 * 16GB DIMM VMW-16384MB
	硬盘容量	512 GB
软件环境	操作系统	openEuler 22.03 LTS
	Kubernetes	v1.23.1

表 6-2 实验环境二

Table 6-2 Experimental Environment 2

硬件环境	CPU	Intel(R) Xeon(R) Platinum 8358 CPU @ 2.60GHz * 6
	内存容量	8 * 16GB DIMM VMW-16384MB
	硬盘容量	512GB
软件环境	操作系统	Ubuntu 22.04 LTS
	Kubernetes	v1.23.1

6.2 实验设计与评估

本研究的具体实验评估分为两部分：

1. 应用程序调优实验评估：

   基于离线辅助训练的负载知识库和参数知识库，针对基准测试中的观测指标，使用基于模型或无模型的优化策略，对应用程序进行参数调优，获取不同的性能表现；其次使用不同的资源范围，探索不同参数在相应资源配给情况下的最佳取值，验证优化中容器编排平台所部署的应用程序的参数配置的可行性和必要性。

2. 微服务场景调优实验评估：

   微服务场景中，对资源和应用参数进行协同调优，并且以整体的端到端观测指标作为优化目标。在调优过程中，首先尽力保证 SLO 时的资源优化，根据请求并发度的不同，测试较低并发（资源充足）和较高并发（资源不充裕）两种情况。两种情况下，本研究都试图在最少资源消耗的情况下，调整微服务环境，使得整个系统尽可能满足 SLO；

   在上述实验评估的基础上，更改优化目标，探究满足 SLO 时所能支持的最大压力，更进一步的探索。证明本优化系统能够在面临不同负载压力情况时，能够以满足 SLO 为首要目标，同时兼顾减少资源消耗、支持更高并发两个目标。

6.2.1 应用程序调优

6.2.1.1 实验概述

为了印证本研究的有效性，本研究需要对不同应用程序进行调优，验证离线辅助训练和在线参数调优策略在优化应用参数方面的有效性。在优化实验中，本研究选择了四个微服务中常用的应用程序，包括 Nginx、Memcached、Redis 和 Postgresql，部署方式是在 DockerHub[50]中为相应的应用程序选择了最常用和稳定的镜像，并使用了其默认配置。这些应用程序部署在集群中，本研究根据离线辅助训练的结果进行回归分析，选择与性能相关度最高的参数，作为先验规则，进行调优实验。具体参数选择情况如表所示。

表 6-3 应用程序调优参数筛选情况

Table 6-3 Filtering of application tuning parameters

应用	版本号	回归分析筛选后的参数数量
Nginx	1.25.3	14
Redis	7.2.3	13
Memcached	3.18	21
PostgreSQL	16.1	15

与此同时，为了验证参数调整在不同的资源环境下的有效性，本研究对具有不同资源分配的应用程序进行调整，并进行相应的基准测试，探索优化后的应用程序的性能情况。具体资源配给和基准测试情况如表 6-4 所示。

表 6-4 应用程序调优测试相关配置

Table 6-4 Configurations for application tuning testing

应用	基准测试	资源配置
Nginx	ApacheBench[64]	1-8 CPU
Redis	Memtier_benchmark[86]	1-8 CPU
Memcached	Memtier_benchmark[86]	1-8 CPU
PostgreSQL	Sysbench（TPC-C）[65]	1-8 CPU

6.2.1.2 实验结果

（1）Nginx 实验结果

本实验使用超文本传输协议（HTTP）性能测试工具 ApacheBench，该工具设计意图是描绘当前服务器的执行性能，显示服务器每秒可以处理的请求数。实验中 ApacheBench 的设置如表 6-5 所示。

表 6-5 Nginx 实验设置

Table 6-5 The setup of Nginx experiment

参数	概述	设置值
-c（concurrency Number of multiple requests to make）	一次产生的请求个数（并发数）	400
-n（requests Number of requests to perform）	测试会话执行的请求个数（测试总共要访问页面的次数）	100000

优化实验覆盖 worker_processes（Nginx 工作进程的数量）、worker_connections（每个工作进程可同时处理的最大连接数）、multi_accept（启用该选项可以使工作进程尽快接受新连接）、keepalive_timeout（Keep-Alive 连接的超时时间）等参数。

其结果如图 6-1 所示，横坐标为 CPU 资源量，纵坐标为吞吐率表现。可以看出，不同资源配给的情况下本优化系统均可改善性能，尤其在 8CPU 时，吞吐率提升 46.55%（每秒处理请求数由 86315.70 提升至 126499.00）。

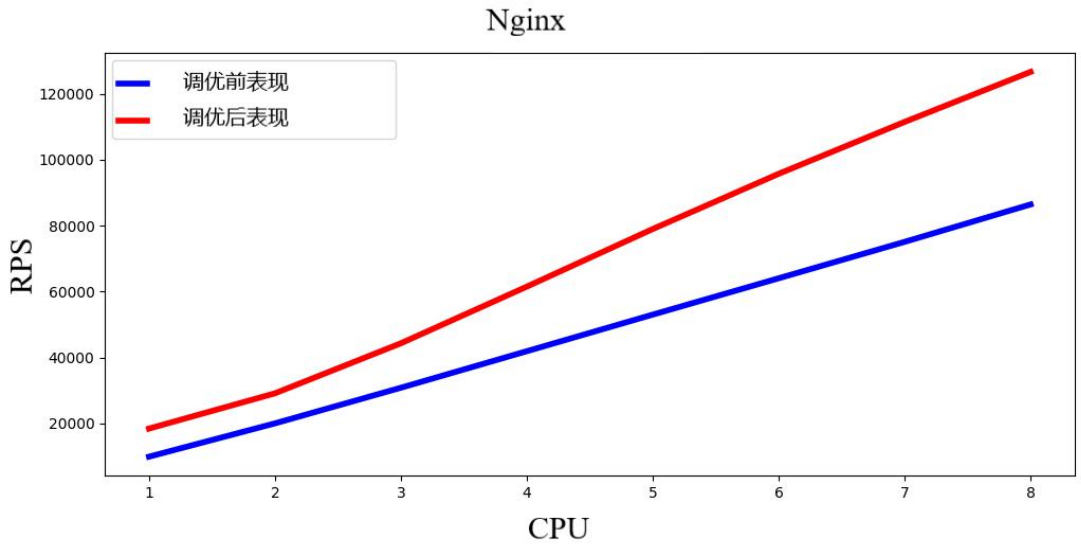

图 6-1 Nginx 实验评估结果

Figure 6-1 Evaluation results of Nginx experiment

其中，8CPU 时，迭代优化过程如图 6-2 所示，横坐标为迭代轮次，纵坐标为吞吐率表现。可以看出，随着迭代轮次的增加，本研究实现的自动化调优系统逐渐为 Nginx 推荐出更佳的参数。

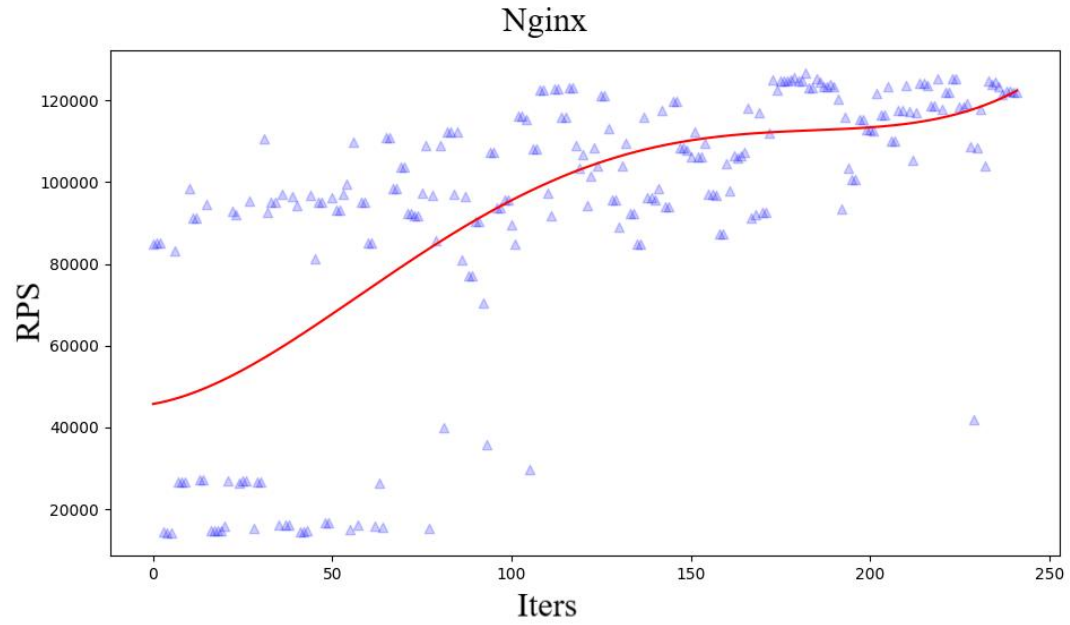

图 6-2 Nginx 迭代优化过程

Figure 6-2 The iterative optimization process of Nginx

（2）Redis 实验结果

本实验使用 Redis Labs 推出的一款命令行工具 Memtier_benchmark，该工具能够产生各种各样的流量模式，根据需求生成多种结构的数据对数据库进行压力测试，帮助了解目标数据库的性能极限。实验中 Memtier_benchmark 的设置如表 6-6 所示。

表 6-6 Redis 与 Memcached 实验设置

Table 6-6 The setup of Redis and Memcached experiment

参数	概述	设置值
-n（--requests）	指定每个客户端发送的请求总数	10000
-t（--threads）	设置线程的数量	24
-c（--clients）	设置并发客户端的数量	40
--radio	设置读写操作的比例	10:1
--key-minimum	设置键的最小大小	100000
--key-maximum	设置键的最大大小	300000
--data-size	设置每个请求的数据大小（以字节为单位）。	1024

优化实验覆盖 maxmemory（占用最大内存量）、maxmemory-policy（超过 maxmemory 值后，删除键的算法策略）、maxclients（设置同时处理客户端连接的最大数量）、appendonly（是否每次写操作后将数据追加到磁盘文件中）、tcp-backlog（设置 TCP 连接队列的长度）等参数。

其结果如图 6-3 所示，横坐标为 CPU 资源量，纵坐标为吞吐率表现。可以看出，不同资源配给的情况下本优化系统均可改善性能，尤其在 8CPU 时，吞吐率提升 208.20%（每秒处理请求数由 116832.64 提升至 360083.44）。

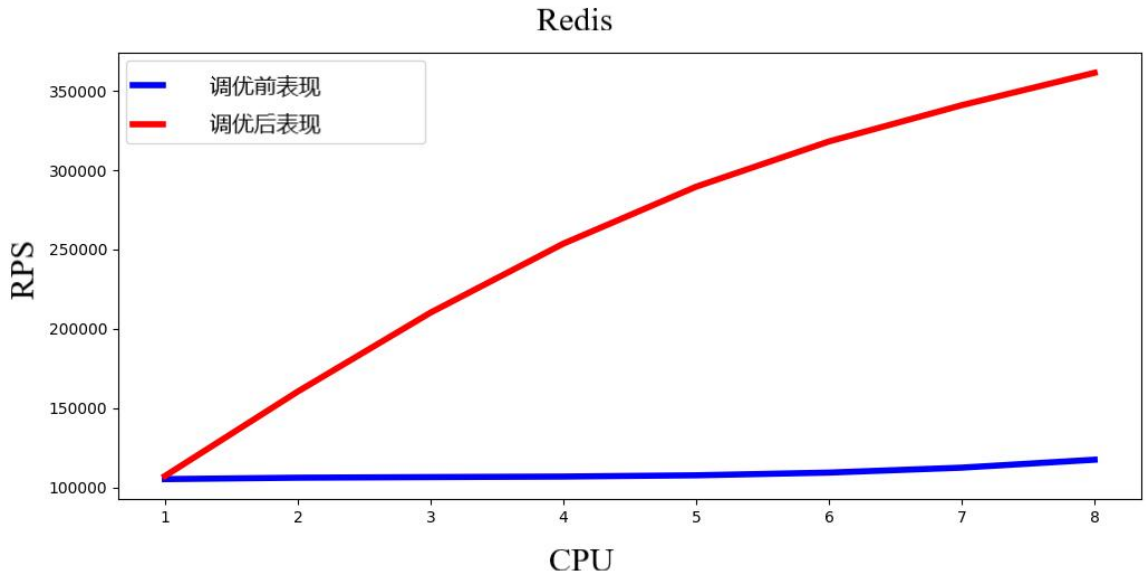

图 6-3 Redis 实验评估结果

Figure 6-3 Evaluation results of Redis experiment

其中，8CPU 时，迭代优化过程如图 6-4 所示，横坐标为迭代轮次，纵坐标为吞吐率表现。可以看出，随着迭代轮次的增加，本研究实现的自动化调优系统逐渐为 Redis 推荐出更佳的参数。

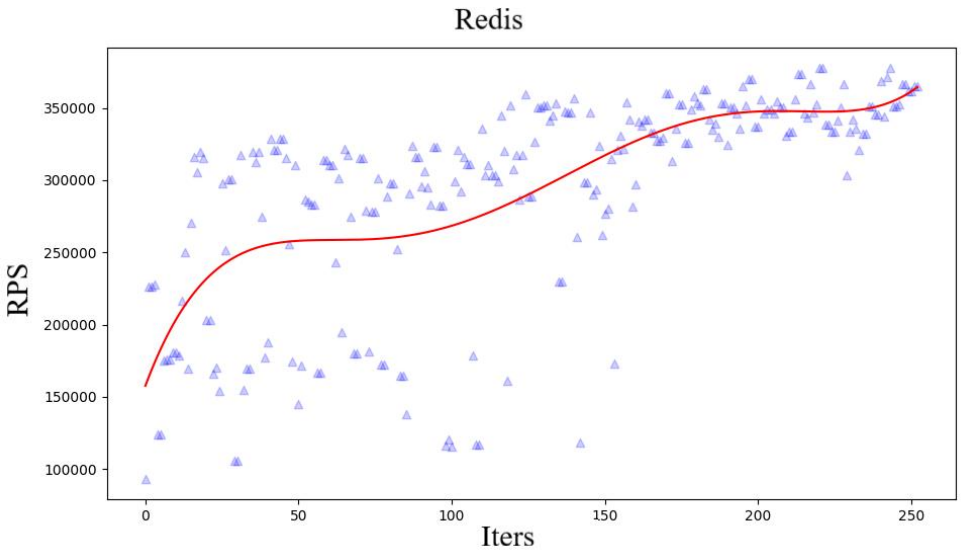

图 6-4 Redis 迭代优化过程

Figure 6-4 The iterative optimization process of Redis

（3）Memcached 实验结果

本实验使用 Redis Labs 推出的一款命令行工具 Memtier_benchmark，该工具能够产生各种各样的流量模式，根据需求生成多种结构的数据对数据库进行压力测试，帮助了解目标数据库的性能极限。实验中 Memtier_benchmark 的设置与 Redis 实验相同（如表 6-6 所示）。

优化实验覆盖 memory-limit（设置 Memcached 实例可以使用的内存限制）、lock-memory （锁定所有分配的页使用的内存）、slab-growth-factor （指定 Memcached 的 chunk 大小增长因子）、slab-min-size（指定 key+value+flags 的最小空间）、threads（使用线程数量）等参数。

其结果如图 6-5 所示，横坐标为 CPU 资源量，纵坐标为吞吐率表现。可以看出，不同资源配给的情况下本优化系统均可改善性能，尤其在 8CPU 时，吞吐率提升 51.01%（每秒处理请求数由 313686.69 提升至 473707.74）。

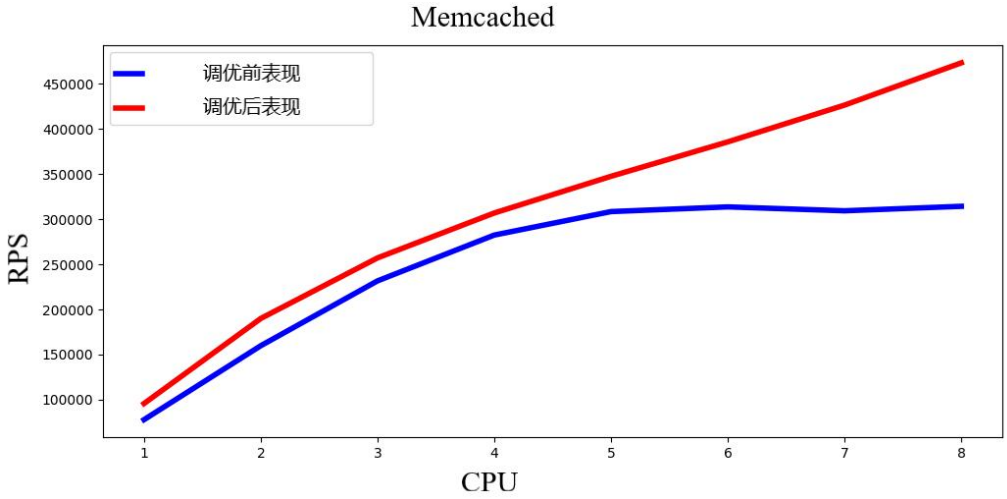

图 6-5 Memcached 实验评估结果

Figure 6-5 Evaluation results of Memcached experiment

其中，8CPU 时，迭代优化过程如图 6-6 所示，横坐标为迭代轮次，纵坐标为吞吐率表现。可以看出，随着迭代轮次的增加，本研究实现的自动化调优系统逐渐为 Memcached 推荐出更佳的参数。

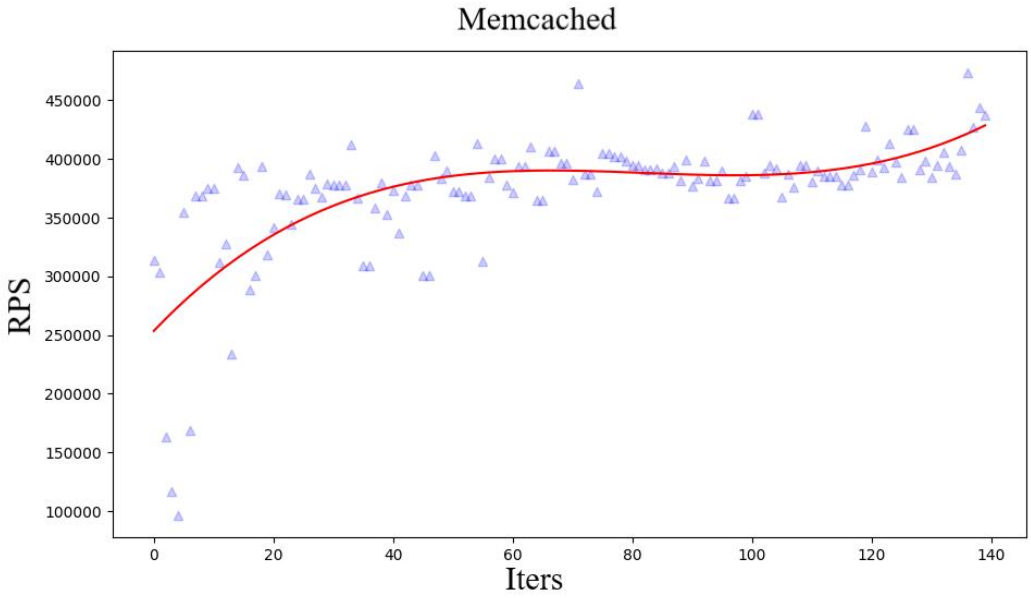

图 6-6 Memcached 迭代优化过程

Figure 6-6 The iterative optimization process of Memcached

（4）PostgreSQL 实验结果

本实验使用基于 LuaJIT 的可脚本多线程基准测试工具 Sysbench，该工具最常用于数据库基准测试，但也可用于创建不涉及数据库服务器的任意复杂工作负载。实验中 Sysbench 的设置如表 6-7 所示。

表 6-7 PostgreSQL 实验设置

Table 6-7 The setup of PostgreSQL experiment

参数	概述	设置值
-	测试脚本	oltp_read_write.lua
--threads	工作线程总数	32
--tables	指定用于测试的表的数量	300
--table-size	指定用于测试的表的大小（行数）	100000

优化实验覆盖 shared_buffers（用于存储磁盘上的数据块的内存区域的大小）、maintenance_work_mem（执行诸如 VACUUM、INDEX 创建和重建等维护操作时所能使用的内存量）、wal_buffers（写入日志缓冲区的大小）、random_page_cost（随机读取磁盘上随机页面的成本估计值）、effective_io_concurrency（在查询计划中并行 IO 操作的并发度）、work_mem

（每个查询运行时可以使用的内存量）、max_wal_size（Write-Ahead Log 日志文件的最大大小）等参数。

其结果如图 6-7 所示，横坐标为 CPU 资源量，纵坐标为吞吐率表现。可以看出，不同资源配给的情况下，本优化系统均可改善性能，尤其在 8CPU 时，吞吐率提升 49.76%（每秒处理请求数由 2048.99 提升至 3068.54）。

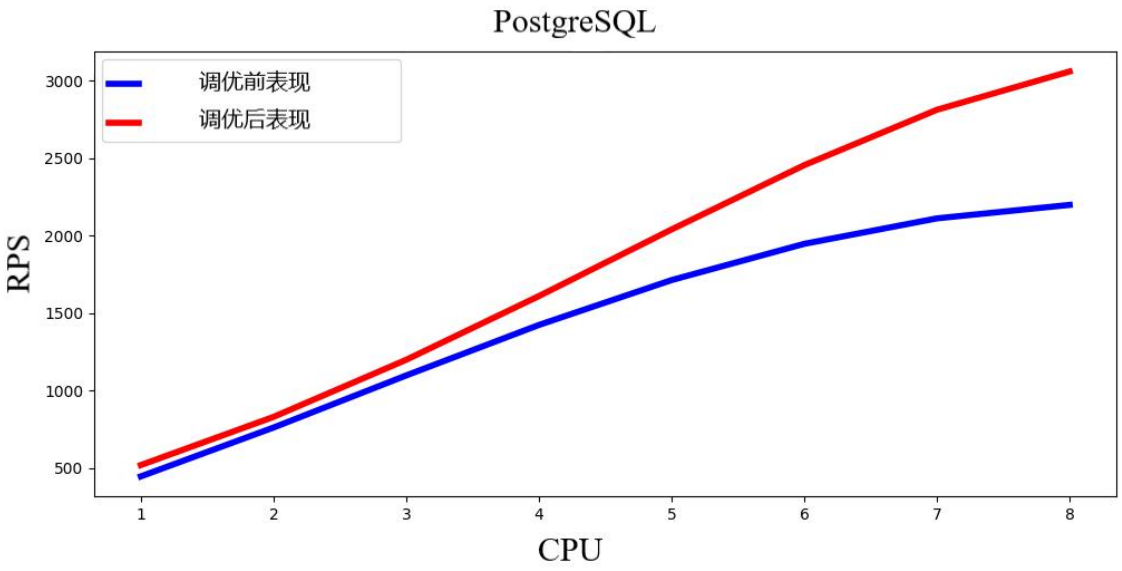

图 6-7 PostgreSQL 实验评估结果

Figure 6-7 Evaluation results of PostgreSQL experiment

其中，8CPU 时，迭代优化过程如图 6-6 所示，横坐标为迭代轮次，纵坐标为吞吐率表现。可以看出，随着迭代轮次的增加，本研究实现的自动化调优系统逐渐为 PostgreSQL 推荐出更佳的参数。

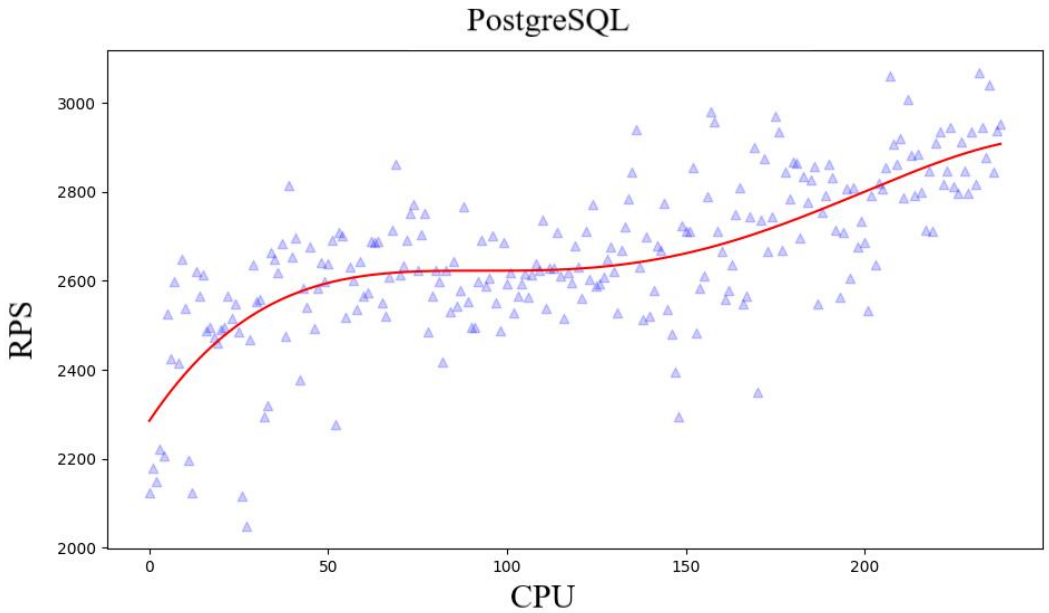

图 6-8 PostgreSQL 迭代优化过程

Figure 6-8 The iterative optimization process of PostgreSQL

综合分析：

可以看出，通过自动化调优系统的优化，应用程序往往能够达到更加的性能表现。且在资源发生变化时，默认配置和优化后配置的性能表现的差距会产生明显变化。尤其是资源供给充足的情况，默认配置中不合理的参数设置造成的性能下降极为明显。四种应用，通过参数优化，性能提升可达 46.55%至208.20%。

在此基础上，进一步展开研究，观察不同资源配置情况下各个应用程序的不同参数的最佳取值情况。对比结果如图 6-9 所示：

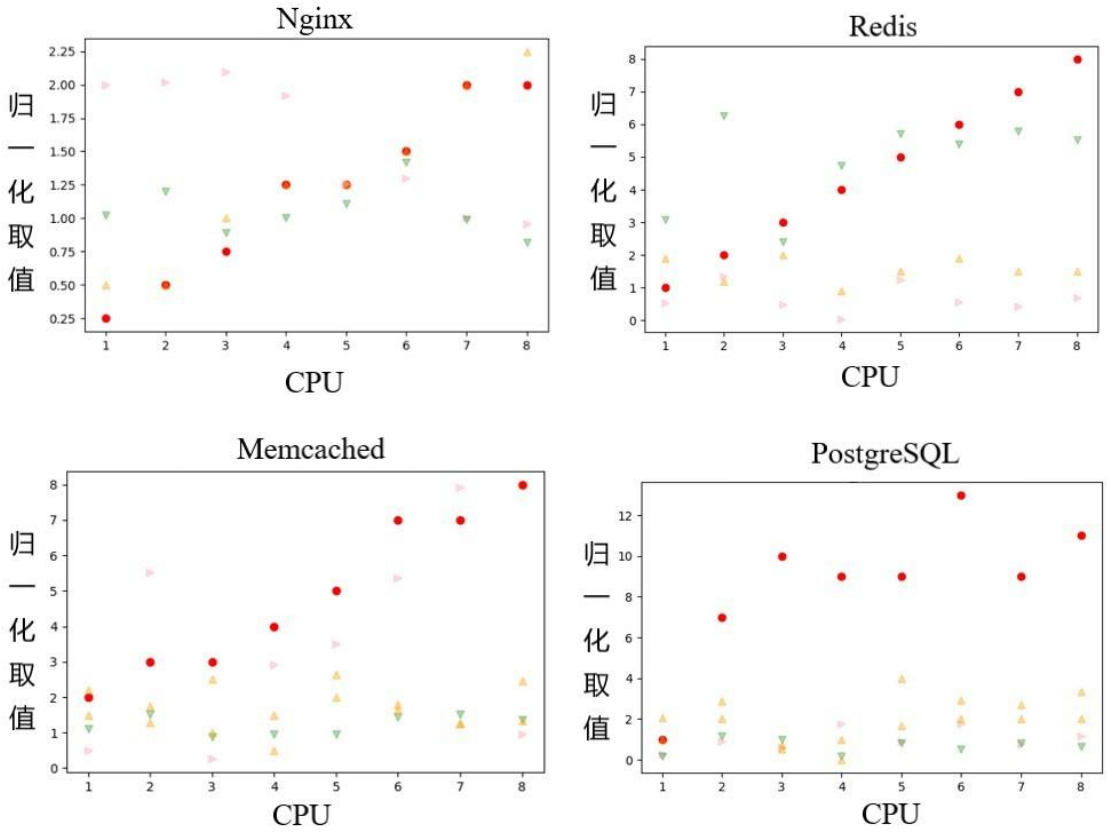

图 6-9 优化参数取值的归一化概况

Figure 6-9 Overview of normalization of optimization parameter values

图中不同颜色和形状的点代表不同的数值型参数，各个应用之间图示之间并无关联。其中，横坐标为该应用程序分配的 CPU 资源数量，纵坐标为最佳性能时该参数取值 p_{best} 与默认配置 $p_{default}$ 的取值进行归一化处理之后的值，即$p_{normalized} = p_{best}/p_{default}$。通过归一化处理，使得各个参数的优化取值与默认值的数量关系保持在同一数量级，方便对比以及规律的观察。

可以看出，部分参数的最佳选项与资源总量存在一定的相关度，这部分参数的调优往往能够依赖人工经验，在某个区间内取一个估计值，完成一定程度上的调优。但是，更多的参数并没有固定的最理想配置范围；同时，考虑到不同类型的负载情况，人工估计的范围值同样未必准确，这便证明自动化在线调

优的必要性。

6.2.2 微服务调优

本研究基于 DeathStarBench[40]中的 HotelReservation，使用 wrk2[67]来处理和提交查询工作负载，进行微服务优化实验，以验证本优化系统的优化系统的有效性。在数据观测层面，本研究使用端到端的指标来收集整个微服务系统的延迟，判断其是否能够满足根据用户需求设置的 SLO，作为优化的首要目标。在第五代英特尔至强可扩展处理器的性能报告[68]中，针对 DeathStarBench 基准测试，使用 P99 延迟小于 100ms 这一目标作为 SLO，探究其处理器可支持的吞吐率变化。本实验沿用该设定，在满足 P99 延迟小于 100ms 这一 SLO 的情况下，进一步关注资源消耗以及微服务所能支持的负载并发程度。本次实验中，资源消耗具体优化的是 CPU 资源的分配，负载并发程度指的是模拟的每秒钟客户请求的事务数。

6.2.2.1 实验概述

本次实验环境当中共有 72 个 CPU 资源，为 HotelReservation 中的 19 个应用进行分配，默认情况下 72 个 CPU 以公平策略分配给每个应用。CPU 资源在微服务系统内的分配，是通过基于进化算法的资源分配优化算法，根据历史分配情况进行突变操作得到的分配策略。

CPU 资源总量的调整，则是基于资源下降的联合调优算法，设定每次收缩和微调的固定值 δ_R 和 δ_r。考虑到本研究的优化思路为满足 SLO 的情况下，逐步减小资源消耗，资源的微调（主要是扩张）主要作为过度缩容后的弥补，因此 δ_R 的设置明显大于 δ_r，分别为 9 和 2，具体单位为 CPU 核数。η_r 和 η_p 被设置为较大值，因为当集群的处理能力降低时，P99 延迟会出现显著波动。此处，固定值或比例的设置方式均可满足实验需求。

具体到微服务架构的参数配置的修改，首先是微服务系统各个子应用程序参数的修改，覆盖 Nginx、Memcached、MongoDB 等，具体是根据其类型、范围生成相应的参数空间，以降低延迟或提高可处理的并发数为目的，迭代搜索最优解。包括除此之外，微服务架构的优化，还考虑环境变量的修改，例如 GOMAXPROCS，是 Golang 运行时环境的一个环境变量，来控制可以并发运行的线程数目，合理配置可以更好地利用服务器的多核处理能力。这部分参数的修改，与第三章中介绍的 EnvPupdater 相匹配。

本研究共进行两部分实验：

第一部分是尽力保证 SLO 时的资源优化，在 6.2.2.2 介绍，根据请求并发度的不同，测试较低并发（资源充足，能够满足 100ms 的 SLO）和较高并发

（资源不充裕，无法满足 100ms 的 SLO）两种情况。两种情况下，本研究都试图在最少资源消耗的情况下，调整微服务环境，使得整个系统尽可能满足 SLO。并在资源剩余时，尝试减少资源消耗。

第二部分是更改优化目标，探究满足 SLO 时所能支持的最大压力，在 6.2.2.3 介绍，这部分可以理解为在第一部分实验基础上，更进一步的探索。前者是节省资源，后者是最大化资源利用。

这样进行实验的目的是，证明本研究实现的面向容器编排平台的容器集群的自动化调优系统能够在面临不同负载压力情况时，以满足 SLO 为首要目标，同时兼顾减少资源消耗、支持更高并发两个目标。

6.2.2.2 资源优化

实验一：低并发度压力情况下的微服务优化

相较于本研究的实验资源，当每秒钟请求的事务数在 1500 时，并发度处于较低水平，72 个 CPU 的初始总资源足够满足 SLO。由于默认资源分配满足 SLO，执行大小为 δ_R 的收缩操作。缩容后，更新轮次计数，如果表现持续较差，则进行一次大小为 δ_r 的扩容，并更新轮次计数。

持续此过程，直到在更少的资源消耗下，达到优劣适中的延迟表现，如 5.3 中的公式（3）所述，$M_{slo} > f(a_i^{R'}) = f(L_i^{R'}, P_0) > M_{slo} - \eta_p$。本研究认为，这达到一种资源分配优化的基本理想状态，则进入参数优化阶段。在此基础上，优化参数配置，试图获取更佳的延迟表现。同理，如果这个过程中，由于参数的良好配置，使得延迟表现得到大幅改善，则可进一步优化资源消耗，进行一次大小为 δ_R 的缩容。

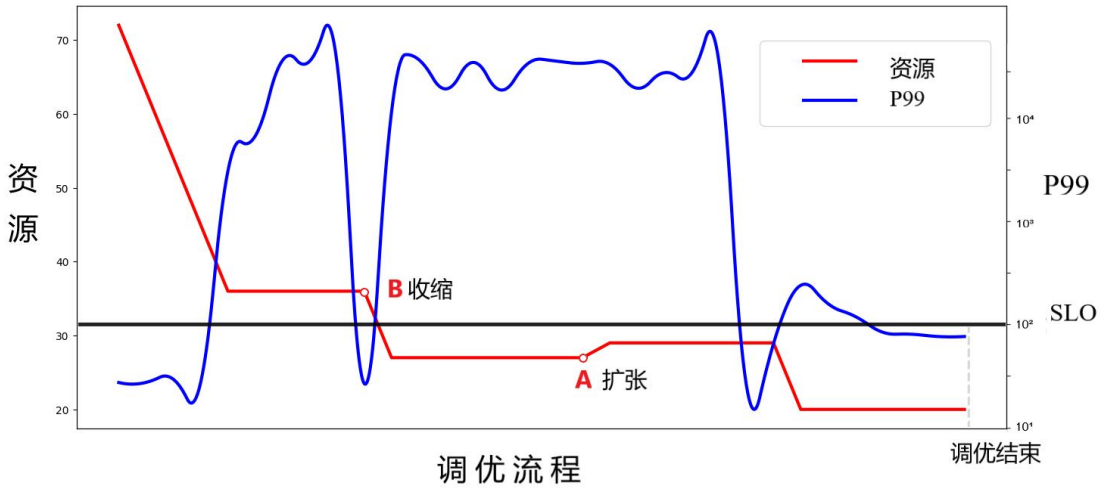

图 6-10 低并发度压力实验评估的优化过程

Figure 6-10 Optimization process for experimental evaluation under low concurrency

pressure

实验结果如图 6-10 所示，在低并发条件下，具有默认公平分配的微服务系统的延迟性能满足 SLO。在第一轮资源缩减以及分配优化后（图 6-6 中的点 B），本研究仅使用 36 个 CPU 就获得了类似的实验结果，这节省了总资源的 50%以上。此外，在优化参数后，观察到资源消耗进一步减少。最终结果表明，经过参数优化和资源的扩展和收缩，在满足 SLO 要求的同时，节省了 52 个 CPU，占到总资源的 72%以上。

实验二：高并发度压力情况下的微服务优化

相较于本研究的实验资源，当每秒钟请求的事务数在 4000 时，并发度处于较高水平，支持同样的服务时，资源显得有些不够充足，原定的 SLO 在 72 个 CPU 的支持下难以实现，P99 延迟达到 1000ms 以上。本研究同样进行实验，尝试提高性能表现，尽可能靠近 SLO。值得注意的是，本次实验当中，在原有不足的资源配额的情况下，本研究在资源分配和参数调优的过程中，找到了明显更优的配置组合，满足 SLO 的情况下，甚至完成了资源消耗的优化。

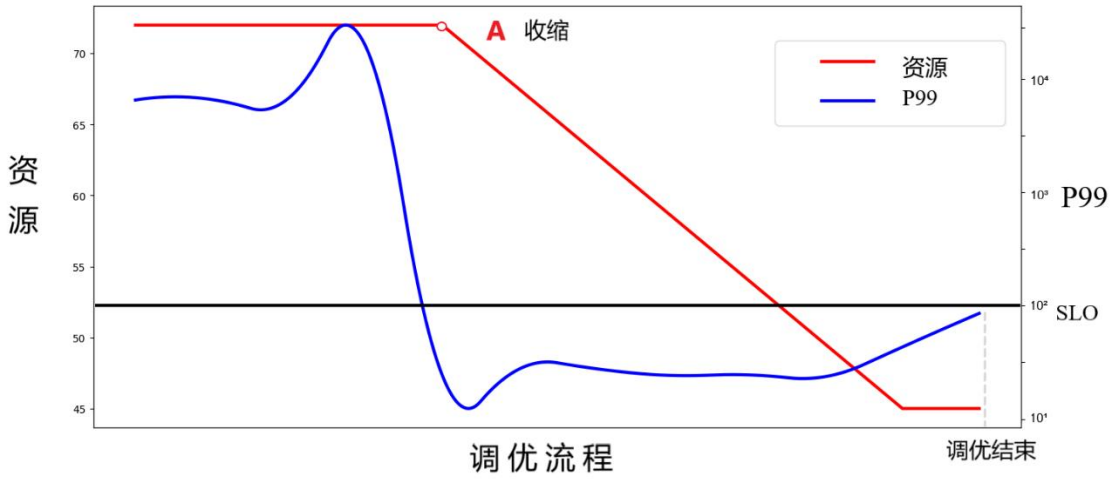

图 6-11 高并发度压力实验评估的优化过程

Figure 6-11 Optimization process for experimental evaluation under hige concurrency pressure

如图 6-11 所示，最初的公平资源分配未能满足 SLO。然后，本研究进行了资源分配优化和参数调整，从而减少了延迟。随后，在满足延迟 SLO 后减少了资源（图 6-7 中的点 A）。在资源减少之后，延迟略有增加，直到满足 SLO 后实现最佳参数配置和资源分配。最终结果表明，本系统在满足 SLO 的同时，节省了 27 个 CPU，占到总资源的 37.5%以上。

从以上两个实验可以看出，在线调优在微服务优化场景中发挥着极其重要的作用。首先，在线资源和参数调整可以在满足 SLO（实验一）的同时节省大量资源。其次，在线资源和参数调整甚至可以通过影响性能来确定整个集群是

否可以实现 SLO（实验二）。当然，这并不一定每个请求压力和资源供给关系中都能实现，但本系统能够尝试将整个系统的表现改善。

6.2.2.3 并发优化

本研究扩展第一个部分的实验，原始实验思路为，用户请求的并发度是固定的，要在该并发程度下，探索更佳的配置，保持延迟满足 SLO 的前提下，减少资源消耗。

第二部分实验，本研究转换思路，优化目标不仅仅是缩减资源，同时还额外尝试支持更多用户并发，这在实际业务场景中，尤其是资源给定，无法更改用途的时候，也是一个现实的需求。

前半部分的实验步骤与第一部分的实验相同，通过优化资源分配，得到优劣适中的延迟表现，如 5.3 中的公式（3）所述，$M_{slo} > f(a_i^{R'}) = f(L_i^{R'}, P_0) > M_{slo} - \eta_p$。在此基础上，进行参数调优，针对每次优化后的微服务，本研究逐步增加每次测试的并发度，以 1000rps 为递增幅度，同时进行参数调优，直到获得一个仍然能够保持 SLO 的最高并发值，最为当前系统的最大并发能力。实验结果如表 6-8 所示。

表 6-8 并发优化实验结果

Table 6-8 The results of concurrent optimization experiments

并发度 (rps)	调优前		调优后	
	平均延迟 (ms)	P99 延迟 (ms)	平均延迟 (ms)	P99 延迟 (ms)
1000	5.39	21.22	4.8	19.42
2000	5.83	23.1	5.66	22.82
3000	3980	7180	5.95	24.01
4000	8450	13110	6.6	26.7
5000	10770	16360	8.49	34.59
6000	12480	19610	9.51	39.33

在调优之前，微服务系统可正常支撑 2000rps 的并发度，延迟表现处于极低水平。当并发达到大约 3000 rps 时，微服务系统已无法正常响应请求。此时该系统的 P99 延迟达到 7 秒，严重超出实际业务可接受的范围。然而，经过调优，即使在 6000 rps 的并发级别下，该系统仍然能够正常工作。此时，P99 延迟减少到 39.33ms，符合 SLO 要求。因此，与公平分配的默认策略以及应用程序和环境变量的默认配置相比，本优化系统在保持 SLO 的情况下，使得微服务

系统能够支持更多用户并发。

在此基础上，本研究进行更细粒度的测试，并将请求并发度与酒店预订系统延迟表现情况绘制成曲线图，如图 6-12、6-13 所示。

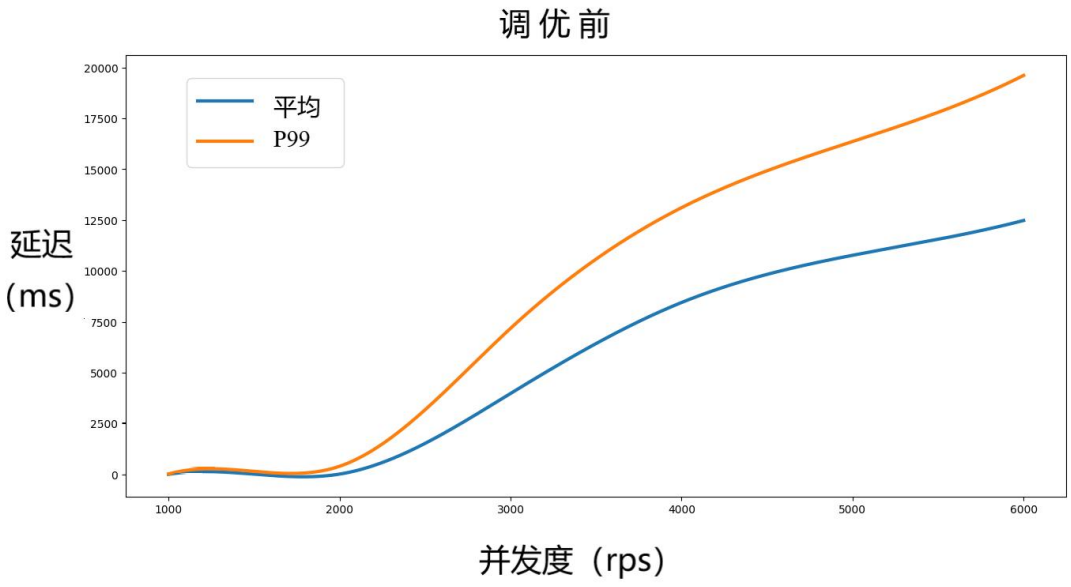

图 6-12 调优前延迟随并发程度增加的变化曲线

Figure 6-12 Curve of latency before tuning as concurrency increases

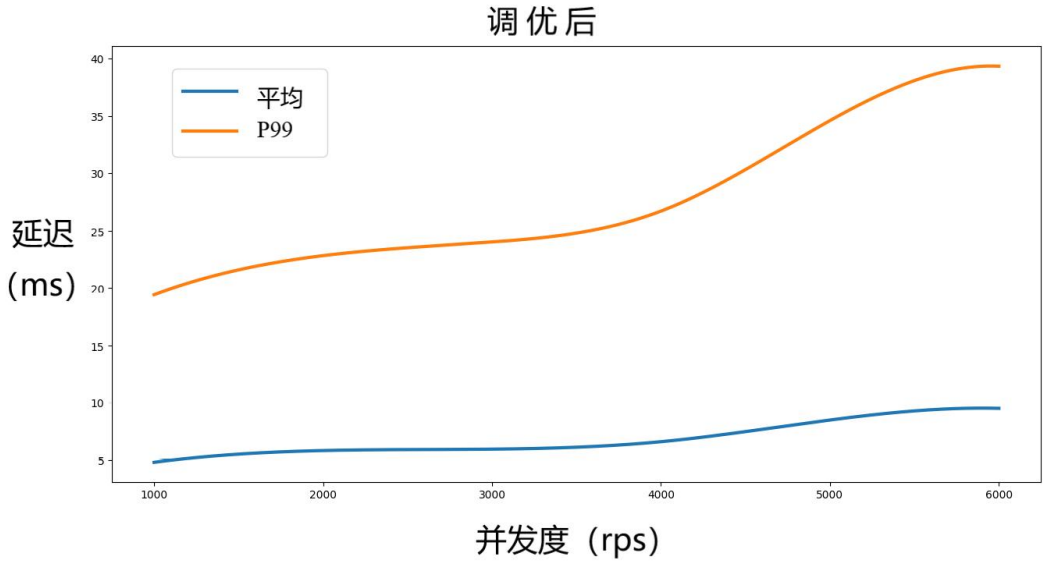

图 6-13 调优后延迟随并发程度增加的变化曲线

Figure 6-13 Curve of latency after tuning as concurrency increases

图 6-12 描绘了调优前 P99 延迟和平均延迟随并发度增加的变化曲线，而图 6-13 表示调优后 P99 延迟和平均延迟的变化曲线。

如图所示，能够观察到调优后的 P99 延迟和平均延迟随并发度升高而升高的曲线，与参数调优之前相比更加稳定。例如在调优之前，延迟情况随着用户

请求并发度的提高（每秒请求数从 1000 上升至 6000）产生剧烈变化，平均延迟从 10 的数量级（5.39ms）到超过 10000 的四个数量级（12480ms）不等，类似地，P99 延迟从 21.22ms 陡增至 19610ms，这反映出调优前的微服务系统无法支持并发度的大幅提高。当并发度超出系统能处理的范围（约 2000 rps 以上）的时候，出现系统崩溃情况，导致延迟陡增，无法满足用户设定的 SLO。

然而，在调优之后，随着并发级别的逐渐提高（每秒请求数从 1000 上升至 6000），总体延迟显示出更平稳且小幅度的增加，例如平均延迟从 4.8ms 小幅上升至 9.51ms，P99 延迟从 19.42ms 小幅上升至 39.33ms。这反映出调优后的微服务系统，有着更合理的资源分配与参数配置，可以稳定承担高并发压力。

本研究将上述实验结果与入选 OSDI 2023（CCF A）的工作 Cilantro[22]的 Github 中给出的实验数据进行对比，该工作同样是面向 Kubernetes 为代表的容器编排平台，在进行微服务调优的时候，主要关注资源分配情况的优化，通过构建资源和性能映射关系，结合不同的搜索策略，找到较为合理的资源分配情况。本研究与其相比的主要优势在于，能够基于不同的优化策略协同优化资源分配和参数配置。

表 6-9 本研究与 Cilantro 工作实验对比

Table 6-9 Comparison between this study and Cilantro

	并发度 (rps)	资源占用	平均延迟 (ms)	P99 延迟 (ms)
Cilantro	3000	152 CPU	64.206	100.934
本研究	3000	72 CPU	5.95	24.01

实验结果如表 6-9 所示，经过对比，可以发现本研究的表现明显更优。在 3000rps 这一并发级别适中的非极端的请求状态下，支持相同的用户并发请求，本研究实现的面向容器集群的自动化调优系统使用更低的的资源占用，仅有 Cliantro 工作的 47.37%，同时得到更佳的性能效果，将 P99 延迟降低到 Cilantro 工作效果的 23.79%，平均延迟更是仅有 Cilantro 工作效果的 9.27%。

基于上述实验数据，可以得出结论，在微服务系统的优化过程中，本优化系统能够以满足 SLO 为首要目标，同时兼顾减少资源消耗、支持更高并发两个目标。

6.3 本章小结

本章介绍面向容器编排平台的集群自动化调优系统基于 Kubernetes 这一容器编排平台，在单一应用和微服务架构上进行调优任务的实验评估，包括实验环境、实验设计。实验设计部分，分别具体介绍应用程序调优和微服务调优，

评估离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法的有效性。应用程序调优部分实验结果表明，本研究能够在不同资源配给的情况下，相较默认配置，使得应用程序性能表现产生明显改善。尤其是资源供给充足的情况，优化效果极为明显。四种应用，包括 Nginx、Memcached、Redis 和 Postgresql，通过参数优化，性能提升可达 46.55%至 208.20%。微服务调优部分的实验，具体包括两种调优目标，分别为资源优化实验和并发优化实验。资源优化实验中：资源充足的低并发压力情况下，经过参数优化和资源的扩展和收缩，在满足 SLO 要求的同时，节省了 52 个 CPU，占到总资源的 72%以上；资源紧张的高并发压力情况下，经过参数优化和资源的扩展和收缩，在满足 SLO 要求的同时，节省了 27 个 CPU，占到总资源的 37.5%以上。并发优化实验中，维持合理的低延迟相应（40ms），微服务系统调优后可支持的并发请求数，达到优化前的三倍以上，与 OSDI 2023 年的工作 Cilantro 相比，本研究可使得微服务系统在更低的资源配置情况下，用更低的延迟达到相同的并发程度，优化效果显著。

第 7 章 总结与展望

本章首先对本文的工作进行总结，然后对工作中存在的问题和提升空间进行归纳和展望，为未来研究指明方向。

7.1 工作总结

本文针对现有容器编排平台管理的集群中应用程序资源分配和参数配置不合理的问题进行研究，设计并实现面向容器编排平台的集群自动化调优系统，实现对集群中应用的资源分配和参数配置的优化以显著改善应用性能，并使微服务系统在满足用户 SLO 的前提下实现资源的节省以及并发度的提高。其中，具体工作内容包括：

1. 设计并实现了面向容器编排平台的集群自动化调优系统。该系统根据容器编排平台 Kubernetes 机制深度定制，主要包括四个模块：控制模块、调优模块、参数资源管理模块、观测模块。控制模块控制管理整个调优系统的运行逻辑，同时作为控制中心与其他三个模块进行交互。调优模块部署优化算法，以有状态服务器的形式，接受性能或状态信息，并优化资源分配和参数配置。参数资源管理模块提供可自定义的参数以及资源修改接口，负责根据调优策略更改集群中容器状态。观测模块负责执行基准测试收集性能指标，或根据需求监控状态指标。四个模块协同工作，实现自动化调优。
2. 研究了离线-在线结合的识别优化方法，主要分为两部分：基于集成学习的离线辅助训练方法和基于机器学习的在线的参数调优方法。前者通过基于 CART 决策树和随机森林的负载识别，进行负载知识库的构建，通过基于 LASSO 算法的参数筛选，结合调优过程，进行参数知识库的构建和参数调优；后者包括基于高斯过程回归的贝叶斯优化的参数迭代调优和无模型的调优流程。实现在负载到达集群时，有效利用历史经验或专家知识库等先验知识，进行准确识别和初始化的参数推荐，以避免初始化的不合理配置导致较差的性能表现，同时能够在较优的配置基础上进行快速的迭代优化，使得应用程序能够以更加符合当前任务负载，实现高性能表现。最后在四种典型应用上的实验评估结果表明，本文提出的离线-在线结合的识别优化方法能够在应用运行过程中，通过识别当前负载状态，进行参数调优或推荐，有效提高应用性能。
3. 研究了资源-参数联合调优方法及资源下降算法，主要分为两部分，实现基于进化算法的资源分配优化算法和资源下降的联合调优算法。前

者通过设计实现不同类型的变异操作，以及通过指数函数转换的方式评估策略得分，更高效搜索分配空间。后者是通过多层次优化的逻辑，结合资源分配和参数优化策略。最后在 DeathStarBench[40]测试工具中的 HotelReservation 的基准测试上的实验评估结果表明，本文提出的资源-参数联合调优方法及资源下降算法在满足用户 SLO 的前提下，兼顾资源和参数的优化，探索更低的资源占用和更高的并发支持，减少集群整体资源消耗，同时提高集群并发处理能力。

7.2 未来工作展望

本文设计并实现的面向容器编排平台的集群自动化调优系统分别从离线-在线结合的识别优化方法和资源-参数联合调优方法及资源下降算法两个方面进行研究，实现对集群中应用的资源分配和参数配置的优化，显著改善应用性能，并使微服务系统在满足用户 SLO 的前提下，实现资源的节省以及并发度的提高。然而，本文仍存在一定不足之处需要在后续研究中逐步完善，其可归纳为如下几个方面：

1．本研究关于参数配置调优的工作中，参数空间的构建，仍需依赖人工完成。例如不同版本应用程序参数配置情况的变化情况、数值型参数的取值区间选取和字符型参数的取值适配，都依赖人工核验才能完成，需要通过构建离线知识库的方式完成准备工作。虽然构建完成的知识库可以重复使用，但人工参与意味着自动化程度的降低，与本研究设计的系统化调优系统的初衷不符，未来的研究工作中，应当尝试探索自动化构建参数空间的方法与范式。

2．本研究关于资源分配和参数配置联合调优的工作中，采用多层次优化的逻辑，有效解决联合调优情况下存在的一定挑战。但是，多层次优化意味着优化的解耦，因此，需要更近一步探索不同的优化层次的优化终点的一致性情况，避免因为优化终点不匹配导致的局部最优或无意义的循环迭代，提高自动化调优系统的有效性。

3．本研究关于微服务场景的优化工作，使用学术界认可的开源的微服务基准测试套件 DeathStarBench，并取得显著优化效果。但受硬件环境、负载情况和资源状态等因素影响，测试套件与业界实际生产环境的优化需求可能存在差异。未来的研究工作中，需要进一步增强自动化调优系统的可靠性和稳定性，将代码开源和真实环境的落地应用作为工作重点。

参考文献

[1] Burns B, Grant B, Oppenheimer D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.

[2] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for {Fine-Grained} resource sharing in the data center[C]//8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11). 2011.

[3] Dimakopoulos V .Spark on Hadoop YARN & Kubernetes for Scalable Physics Analysis In collaboration with CERN Openlab, Intel, CMS Experiment, Fermilab[J]. 2018.

[4] Cloud Native Computing Foundation. https://www.cncf.io/

[5] Kubernetes adoption, security, and market trends report 2023. https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview

[6] Li C, Wang S, Hoffmann H, et al. Statically inferring performance properties of software configurations[C]//Proceedings of the Fifteenth European Conference on Computer Systems. 2020: 1-16.

[7] Huang H, Zhao Y, Rao J, et al. Adapt burstable containers to variable CPU resources[J]. IEEE Transactions on Computers, 2022, 72(3): 614-626.

[8] Nginx[J].Programmer, 2007.

[9] White T. Hadoop: The definitive guide[M]. " O'Reilly Media, Inc.", 2012.

[10] Spark A. A Unified Engine for Big Data Processing[J]. Commun. ACM, 2016, 59(11): 56-65.

[11] Zhu Y, Liu J, Guo M, et al. Bestconfig: tapping the performance potential of systems via automatic configuration tuning[C]//Proceedings of the 2017 symposium on cloud computing. 2017: 338-350.

[12] Christian Dupuis.Professional Apache Tomcat 6[J]. 2007.

[13] Jiang C, Wu P. A Fine-Grained Horizontal Scaling Method for Container-Based Cloud[J]. Scientific Programming, 2021, 2021: 1-10.

[14] Kannan R S, Subramanian L, Raju A, et al. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks[C]//Proceedings of the Fourteenth EuroSys Conference 2019. 2019: 1-16.

[15] Qiu H, Banerjee S S, Jha S, et al. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices[C]//14th USENIX symposium on operating systems design and implementation (OSDI 20). 2020: 805-825.

[16] Kim E, Lee K, Yoo C. Network SLO-aware container scheduling in Kubernetes[J]. The Journal of Supercomputing, 2023, 79(10): 11478-11494.

[17] Kaur K, Garg S, Kaddoum G, et al. KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem[J]. IEEE Internet of Things Journal, 2019, 7(5): 4228-4237.

[18] Han R, Liu C H, Zong Z, et al. Workload-adaptive configuration tuning for hierarchical cloud schedulers[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(12): 2879-2895.

[19] Beltre A, Saha P, Govindaraju M. Kubesphere: An approach to multi-tenant fair scheduling for kubernetes clusters[C]//2019 IEEE cloud summit. IEEE, 2019: 14-20.

[20] Kash I A, O'Shea G, Volos S. DC-DRF: Adaptive multi-resource sharing at public cloud scale[C]//Proceedings of the ACM symposium on cloud computing. 2018: 374-385.

[21] Lin M, Xi J, Bai W, et al. Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud[J]. IEEE access, 2019, 7: 83088-83100.

[22] Bhardwaj R, Kandasamy K, Biswal A, et al. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback[C]//17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023: 623-643.

[23] Chow K H, Deshpande U, Seshadri S, et al. DeepRest: deep resource estimation for interactive microservices[C]//Proceedings of the Seventeenth European Conference on Computer Systems. 2022: 181-198.

[24] Bao Y, Peng Y, Wu C. Deep learning-based job placement in distributed machine learning clusters[C]//IEEE INFOCOM 2019-IEEE conference on computer communications. IEEE, 2019: 505-513.

[25] Peng Y, Bao Y, Chen Y, et al. Dl2: A deep learning-driven scheduler for deep learning clusters[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(8): 1947-1960.

[26] Zhang J, Liu Y, Zhou K, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning[C]//Proceedings of the 2019 international conference on management of data. 2019: 415-432.

[27] Zhang J, Zhou K, Li G, et al. CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system[J]. The VLDB Journal, 2021, 30(6): 959-987.

[28] Li G, Zhou X, Li S, et al. Qtune: A query-aware database tuning system with deep reinforcement learning[J]. Proceedings of the VLDB Endowment, 2019, 12(12): 2118-2130.

[29] Van Aken D, Pavlo A, Gordon G J, et al. Automatic database management system tuning through large-scale machine learning[C]//Proceedings of the 2017 ACM international conference on management of data. 2017: 1009-1024.

[30] Gao Z, Wang T, Wang Q, et al. Execution Time Prediction for Apache Spark[C]//Proceedings of the 2018 International Conference on Computing and Big Data. 2018: 47-51.
[31] Rahman M A, Hossen J, Sultana A, et al. A smart method for spark using neural network for big data[J]. International Journal of Electrical and Computer Engineering, 2021, 11(3): 2525.
[32] go.sum · openEuler https://gitee.com/openeuler/A-Tune/ blob/master/go.sum.
[33] Acher M, Martin H, Pereira J A, et al. Learning very large configuration spaces: What matters for Linux kernel sizes[D]. Inria Rennes-Bretagne Atlantique, 2019.
[34] Acher M, Martin H, Pereira J A, et al. Learning from thousands of build failures of Linux kernel configurations[D]. Inria; IRISA, 2019.
[35] Martin H, Acher M, Pereira J A, et al. Transfer learning across variants and versions: The case of linux kernel size[J]. IEEE Transactions on Software Engineering, 2021, 48(11): 4274-4290.
[36] Yi L, Connan J. KernTune: self-tuning Linux kernel performance using support vector machines[C]//Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. 2007: 189-196.
[37] Curioso A, Bradford R, Galbraith P. Expert PHP and MySQL[M]. John Wiley & Sons, 2010.
[38] 曾超宇, 李金香. Redis 在高速缓存系统中的应用[J]. 微型机与应用, 2013, 32(12): 11-13.
[39] Geschwinde E, Schönig H J. PostgreSQL developer's handbook[M]. Sams Publishing, 2002.
[40] Gan Y, Zhang Y, Cheng D, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems[C]//Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 3-18.
[41] Rodríguez-Haro F, Freitag F, Navarro L, et al. A summary of virtualization techniques[J]. Procedia Technology, 2012, 3: 267-272.
[42] Eder M. Hypervisor-vs. container-based virtualization[J]. Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), 2016, 1.
[43] Morabito R, Kjällman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison[C]//2015 IEEE International Conference on Cloud Engineering. IEEE, 2015: 386-393.
[44] Noronha V, Lang E, Riegel M, et al. Performance evaluation of container based virtualization on embedded microprocessors[C]//2018 30th International Teletraffic Congress (ITC 30). IEEE, 2018, 1: 79-84.

[45] Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors[C]//Proceedings of the 2Nd ACM SIGOPS/EuroSys european conference on computer systems 2007. 2007: 275-287.

[46] Potdar A M, Narayan D G, Kengond S, et al. Performance evaluation of docker container and virtual machine[J]. Procedia Computer Science, 2020, 171: 1419-1428.

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

[48] 佟旭.基于 Docker Swarm 集群的资源亲和性调度算法[D].兰州大学,2018.

[49] Kwon S, Lee J H. Divds: Docker image vulnerability diagnostic system[J]. IEEE Access, 2020, 8: 42666-42673.

[50] Combe T, Martin A, Di Pietro R. To docker or not to docker: A security perspective[J]. IEEE Cloud Computing, 2016, 3(5): 54-62.

[51] 徐珉.Docker 环境下容器编排工具的选择[J].集成电路应用,2017,34(07):6266.DOI:10.19339/j.issn.1674-2583.2017.07.014.

[52] Medel V, Rana O, Bañares J Á, et al. Modelling performance & resource management in ku bernetes[C]//Proceedings of the 9th International Conference on Utility and Cloud Compu ting. 2016: 257-262.

[53] Sahu N, Thakre S C. A survey on Kubernetes architec-ture and its significance[R]. EasyChair, 2022.

[54] Kayal P. Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improve ment Scope[C]//2020 IEEE 6th World Forum on Internet of Things (WF-IoT). IEEE, 2020: 16.

[55] 董迎港.基于 Kubernetes 的异构资源调度的研究与实现[D].北京邮电大学,2021.DOI:10.26969/d.cnki.gbydu.2021.002291.

[56] 罗永安,包梓群,赵恪振,余隆勇.Kubernetes 的资源动态调度设计研究[J].软件导刊,2022,21(02):109-114.

[57] 陈博,周亦敏.基于 Kubernetes 的 CI/CD 平台[J].计算机系统应用,2020,29(12):268 271.DOI:10.15888/j.cnki.csa.007682.

[58] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm[C]//2014 USE NIX Annual Technical Conference (Usenix ATC 14). 2014: 305-319.

[59] Han S, Lee S. Persistent store-based dual replication system for distributed SDN control ler[C]//2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT). IEEE, 2016: 1-2.

[60] Telenyk S, Sopov O, Zharikov E, et al. A Comparison of Kubernetes and Kubernetes-Com patible Platforms[C]//2021 11th IEEE International Conference

on Intelligent Data Acquisi tion and Advanced Computing Systems: Technology and Applications (IDAACS). IEEE, 2021, 1: 313-317.

[61] Fazio M, Celesti A, Ranjan R, et al. Open issues in scheduling microservices in the cloud[J]. IEEE Cloud Computing, 2016, 3(5): 81-88.

[62] GitHub - brianfrankcooper/YCSB: Yahoo! Cloud Serving Benchmark — github.com. https://github.com/ brianfrankcooper/YCSB.

[63] Kato T, Uemura M. Period analysis using the least absolute shrinkage and selection operator (Lasso)[J]. Publications of the Astronomical Society of Japan, 2012, 64(6): 122.

[64] Brunelle J F, Nelson M L, Balakireva L, et al. Evaluating the SiteStory transactional web archive with the ApacheBench tool[C]//Research and Advanced Technology for Digital Libraries: International Conference on Theory and Practice of Digital Libraries, TPDL 2013, Valletta, Malta, September 22-26, 2013. Proceedings 3. Springer Berlin Heidelberg, 2013: 204-215.

[65] Ahmed M, Uddin M M, Azad M S, et al. MySQL performance analysis on a limited resource server: Fedora vs. Ubuntu Linux[C]//Proceedings of the 2010 Spring Simulation Multiconference. 2010: 1-7.

[66] Abadi D J. Data management in the cloud: Limitations and opportunities[J]. IEEE Data Eng. Bull., 2009, 32(1): 3-12.

[67] GitHub- giltene/wrk2: A constant throughput, cor rect latency recording variant of wrk — github.com. https://github.com/giltene/wrk2.

[68] 5th Generation Intel® Xeon® Scalable Processors. https://edc.intel.com/content/www/us/en/products/performance/benchmarks/5th-generation-intel-xeon-scalable-processors/

[69] 李启锐,彭志平,崔得龙,何杰光.容器云环境虚拟资源配置策略的优化[J].计算机应用, 2019, 39(03): 784-789.

[70] 陈国良,王熙法,庄镇泉等.遗传算法及其应用[M].人民邮电出版社, 1999.

[71] 徐俊,项倩红,肖刚.基于改进混合蛙跳算法的云工作流负载均衡调度优化[J].计算机科学, 2019, 46(11): 315-322.

[72] 杨维,李歧强.粒子群优化算法综述[J].中国工程科学, 6(5): 87-94.

[73] Chandrasekaran K , Joseph C T . IntMA: Dynamic Interaction-Aware Resource Allocation for Containerized Microservices in Cloud environments[J]. Journal of Systems Architecture, 2020, 111:101785.

[74] Yang Y , Dusia A . Network Quality of Service in Docker Containers[C]// IEEE International Conference on Cluster Computing. IEEE, 2015.

[75] McDaniel S, Herbein S, Taufer M. A two-tiered approach to I/O quality of service in docker containers[C]//2015 IEEE International Conference on Cluster Computing. IEEE, 2015: 490-491.

[76] Fekry A, Carata L, Pasquier T, et al. Tuneful: An online significance-aware configuration tuner for big data analytics[J]. arXiv preprint arXiv:2001.08002, 2020.

[77] Öztürk M M. Tuning parameters of Apache Spark with Gauss–Pareto-based multi-objective optimization[J]. Knowledge and Information Systems, 2024, 66(2): 1065-1090.

[78] Goettel B C. CadVisor (CAD Equipment Analysis and Selection System)[C]//Computing in Civil Engineering. ASCE, 1994: 2140-2144.

[79] Timofeev R. Classification and regression trees (CART) theory and applications[J]. Humboldt University, Berlin, 2004, 54.

[80] Safavian S R, Landgrebe D. A survey of decision tree classifier methodology[J]. IEEE transactions on systems, man, and cybernetics, 1991, 21(3): 660-674.

[81] Blockeel H, Struyf J. Efficient algorithms for decision tree cross-validation[J]. Journal of Machine Learning Research, 2002, 3(Dec): 621-650.

[82] Bratley P, Fox B L, Niederreiter H. Implementation and tests of low-discrepancy sequences[J]. ACM Transactions on Modeling and Computer Simulation (TOMACS), 1992, 2(3): 195-213.

[83] Banerjee A, Dunson D B, Tokdar S T. Efficient Gaussian process regression for large datasets[J]. Biometrika, 2013, 100(1): 75-89.

[84] Frazier P I. A tutorial on Bayesian optimization[J]. arXiv preprint arXiv:1807.02811, 2018.

[85] Zhang Y, Wang S, Ji G. A comprehensive survey on particle swarm optimization algorithm and its applications[J]. Mathematical problems in engineering, 2015, 2015.

[86] memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached.https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/

[87] Aderaldo C M, Mendonça N C, Pahl C, et al. Benchmark requirements for microservices architecture research[C]//2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). IEEE, 2017: 8-13.

[88] Yu Z, Bei Z, Qian X. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing[C]//Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. 2018: 564-577.

[89] Wang G, Xu J, He B. A novel method for tuning configuration parameters of spark based on machine learning[C]//2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS). IEEE, 2016: 586-593.

附录一 Task 示例

```
apiVersion: k8stune.k8stune/v1
kind: Task
metadata:
  labels:
    app.kubernetes.io/name: task
    app.kubernetes.io/instance: task-sample
    app.kubernetes.io/part-of: k8stune
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: k8stune
  name: nginx-online-task
  namespace: paperexp
spec:
  TaskConfig:
    object:  # 需要调优对象的列表，此处仅有使用  deployment 部署的  Nginx
      type: deployment
      namespace: default
      name: nginx
    taskType: online # 在线参数调优
    taskPolicy: default # 不使用离线知识库
    onlineCycle: "30s" # 每轮次观测的间隔时长
  parameters:
    - agentInfo:
        filePath: /etc/nginx/nginx.conf # 配置文件路径
        name: "nginx-controlagent" # Agent 名称
        namespace: "nginx" # Agent 命名空间
        updater: "K8sAppandRuntimePupdater" # 使用的  Pupdater
      paramterSet: # 需要修改的参数
        - name: "nginx.access  log"
        - name: "nginx.multi_accept"
                        ……
        - name: "nginx.worker_processes"
        - name: "nginx.worker  connections"
  metrics: # 基准测试或观测信息
    - agent:
        name: "nginx-collectagent" # collectagent 的名称和命名空间
        namespace: "nginx"
      benchPerformance: # 作为反馈的指标
        - name: "rps"
```

附录二 ControlAgent 示例

```
apiVersion: k8stune.k8stune/v1
kind: ControlAgent
metadata:
  name: nginx-controlagent #名字和命名空间
  namespace: paperexp
spec:
  agentInfo:
    filePath: /etc/nginx/nginx.conf #修改参数的文件
    name: nginx-controlagent
    updater: K8sAppandRuntimePupdater
  parameterSet: #可修改参数
  - desc: Enabling or Disabling nginx access
    dtype: string #类型
     # 获取参数的值的方式 %s 是参数名称
    get: grep -m1 'access_log' %s | awk -F ' ' '{print $2}'
    name: nginx.access_log # 参数名称
    options: #string 类型的具体选项
    - /var/log/nginx/access.log main
    - "off"
    #修改参数的方式，两处 %s 分别是参数名称和值
    set: sed -i 's#access_log .*#access_log %s;#g' %s
  - desc: Enabling or Disabling nginx error_log
    dtype: string
    get: grep -m1 'error_log' %s | awk -F ' ' '{print $2}'
    name: nginx.error_log
    options:
    - /var/log/nginx/error.log
    - /dev/null
    set: sed -i 's#error_log .*#error_log %s;#g' %s
  - dataRange:
      lower: "1"
      upper: "5"
    desc: work process
    dtype: int
    get: grep -m1 'worker_processes' %s | awk -F ' ' '{print $2}'
    name: nginx.worker  processes
    set: sed -i 's#worker_processes .*#worker_processes %s;#g' %s
  - dataRange:
      lower: "256"
      upper: "2048"
    desc: connection
    dtype: int
    get: grep -m1 'worker_connections' %s | awk -F ' ' '{print $2}'
    name: nginx.worker  connections
    set: sed -i 's#worker_connections .*#worker_connections %s;#g' %s
  - dataRange: #float 和  int 类型参数进行取值范围的限定
      lower: "1"
      upper: "9"
    desc: gzip_comp_level
    dtype: int
    get: grep -m1 'gzip_comp_level' %s | awk -F ' ' '{print $2}'
    name: nginx.gzip_comp_level
    set: sed -i 's#gzip_comp_level .*#gzip_comp_level %s;#g' %s
  - desc: multi_accept
    dtype: string
    get: grep -m1 'multi_accept' %s | awk -F ' ' '{print $2}'
    name: nginx.multi_accept
    options:
    - "on"
```

```
60.          - "off"
61.          set: sed -i 's#multi_accept .*#multi_accept %s;#g' %s
62.        - desc: proxy_buffers
63.          dtype: string
64.          get: grep -m1 'proxy_buffers' %s | awk -F ' ' '{print $2}'
65.          name: nginx.proxy_buffers
66.          options:
67.          - 4 4k
68.          - 6 4k
69.          - 6 8k
70.          - 4 8k
71.          set: sed -i 's#proxy_buffers .*#proxy_buffers %s;#g' %s
72.        - desc: proxy_busy_buffers_size
73.          dtype: string
74.          get: grep -m1 'proxy_busy_buffers_size' %s | awk -F ' ' '{print $2}'
75.          name: nginx.proxy_busy_buffers_size
76.          options:
77.          - 8k
78.          - 12k
79.          set: sed -i 's#proxy_busy_buffers_size .*#proxy_busy_buffers_size %s;#g' %s
80.        - desc: sendfile
81.          dtype: string
82.          get: grep -m1 'sendfile' %s | awk -F ' ' '{print $2}'
83.          name: nginx.sendfile
84.          options:
85.          - "on"
86.          - "off"
87.          set: sed -i 's#sendfile .*#sendfile %s;#g' %s
88.        - desc: tcp_nopush
89.          dtype: string
90.          get: grep -m1 'tcp_nopush' %s | awk -F ' ' '{print $2}'
91.          name: nginx.tcp_nopush
92.          options:
93.          - "on"
94.          - "off"
95.          set: sed -i 's#tcp_nopush .*#tcp_nopush %s;#g' %s
96.        - dataRange:
97.            lower: "0"
98.            upper: "4096"
99.          desc: gzip_min_length
100.         dtype: int
101.         get: grep -m1 'gzip_min_length' %s | awk -F ' ' '{print $2}'
102.         name: nginx.gzip_min_length
103.         set: sed -i 's#gzip_min_length .*#gzip_min_length %s;#g' %s
104.       - desc: gzip
105.         dtype: string
106.         get: grep -m1 'gzip' %s | awk -F ' ' '{print $2}'
107.         name: nginx.gzip
108.         options:
109.         - "on"
110.         - "off"
111.         set: sed -i 's#gzip .*#gzip %s;#g' %s
112.       - desc: gzip  buffers
113.         dtype: string
114.         get: grep -m1 'gzip  buffers' %s | awk -F ' ' '{print $2}'
115.         name: nginx.gzip_buffers
116.         options:
117.         - 4 32k
118.         - 8 64k
119.         - 8 32k
120.         - 16 64k
121.         - 16 32k
122.         - 2 16k
123.         set: sed -i 's#gzip_buffers .*#gzip_buffers %s;#g' %s
```

致 谢

在我的硕士学习之旅即将结束之际，我想表达对所有支持和帮助过我的人深深的感谢。这里的每个字，每一页的内容，都浸透了来自他们的鼓励、启示与支持。尽管这段旅程充满挑战，但正是这些无形的力量使我能够坚持下来，并最终为人生的这一阶段画上句号。

首先要感谢我的导师赵琛老师和实验室主任武延军老师，两位老师的专业知识和敬业精神让我受益匪浅，深厚的学术造诣和严谨的科研态度为我树立了榜样。学业上，两位老师在操作系统、智能软件等领域极高的造诣激起我对科研的兴趣，带我走进了科研的殿堂；生活中，两位老师为我们提供了良好的科研环境，帮我们扫平了很多生活上的阻碍。两位老师不仅让我学到了很多前沿的专业知识，他们强烈的社会责任感和使命感也不断地感染着我，一直是我学习的榜样。

其次，感谢张立波老师和侯朋朋老师对我研究生期间科研工作的热情指导和无私分享。在整个研究过程中，两位老师既是我的学术导向灯塔，更是智慧的朋友，在我遇到困难和迷茫时伸出援手，提供宝贵的建议和支持。感谢智能软件研究中心的吴敬征老师、李玲老师、于佳耕老师、殷佳毅老师在我的科研道路上提供的支持与帮助。感谢纪然老师、丁子辰老师、李彩丽老师、汤诗豪老师对研究生日常事务的耐心解答。

我还要感谢实验室的其他同学，包括潘庆霖师兄、王皓师兄、王伯英师兄、何家泰同学、王承杰同学、赵瑞霖师弟等，感谢他们在和我一起讨论学术问题时给了我一些启发。

此外，我还要感谢我的家人。感谢我的父母，他们对我的学业始终给予无条件的支持和爱护，并对我充满信心。感谢我的女朋友姚雨晴，她的理解、鼓励和爱让我在最困难的时刻也能坚持下去。他们的支持和爱是我求学之路的坚强后盾。

最后，我要感谢所有给予我灵感和力量的人，感谢那些在我的学业旅程中以任何形式帮助和支持我的人。虽然在这里未能一一列举，但我将铭记在心，并以我的成就来回报他们的帮助和鼓励。

我深知，没有他们的帮助和支持，完成这篇论文是不可能的。因此，我将这份致谢送给所有在这段旅程中与我同行的人。

作者简历及攻读学位期间发表的学术论文与其他相关学术成果

作者简历：

2017 年 9 月——2021 年 7 月，在南开大学计算机学院获得学士学位。

2021 年 9 月——2024 年 7 月，在中国科学院大学软件研究所攻读硕士学位。

已发表（或正式接受）的学术论文：

申请或已获得的专利：

1）王新元，侯朋朋，何家泰，张开创，于佳耕，武延军. 一种基于机器学习的 Linux 系统内核配置的策略推荐方法及装置. 申请号：CN202310909401.

2）王新元，何家泰，黄山云，马成宇，张开创，侯朋朋，于佳耕，武延军. 一种基于机器学习的云服务平台负载策略推荐方法及装置. 申请号：CN202310792705.

3）王新元，武延军. 一种基于神经辐射场和语义分割的少样本三维重构方法及装置. 申请号：CN202310960660.

参加的研究项目及获奖情况：

1）参与 123 项目 1-1

2）参加国家重点研发专项 “面向数据中心的 RISC-V 基础软件生态技术研究 (2023YFB4503900)” 子课题 5-2 “RISC-V 性能与稳定性的研究与评估”

3）软件著作权已授权：基于主流发行版默认值的内核配置项取值推荐软件 V1.0. 登记号：2023SR0481878

4）2021 年中国科学院大学硕士生二等学业奖学金

5）2022 年中国科学院大学硕士生三等学业奖学金

6）2023 年中国科学院大学硕士生二等学业奖学金