文章编号:2096-1472(2026)10-0033-08
DOI:10.19644/j.cnki.issn2096-1472.2026.010.006

一种基于任务驱动的 Kubernetes 异构任务调度模型

赵 羿,江凌云

(南京邮电大学通信与信息工程学院,江苏 南京 210003)
✉ 719323738@qq.com; jiangly@njupt.edu.cn

摘 要:针对企业云原生环境中 Kubernetes 默认调度器在异构资源和多样化工作负载调度中暴露出的响应延迟高、资源碎片化严重和集群稳定性不足等行业共性问题,展开优化研究。尽管已有研究采用多目标决策和机器学习等方法对调度策略进行改进,但仍普遍存在优化目标单一、动态适应能力弱、对实时敏感任务支持有限以及对集群健康状态关注不足等缺陷。为此,提出一种任务驱动的 Kubernetes 异构任务调度模型。该模型结合改进的 TOPSIS(逼近理想解)多准则决策与事件触发机制,实现对多维资源的动态评估与节点综合排序,并通过资源预留及并行绑定策略显著降低调度延迟。实验结果表明,该模型在调度延迟、I/O 及内存压力等多个性能指标上均显著优于传统调度器及典型优化算法,能够有效提升高并发异构环境下的资源利用率与系统稳定性,为行业提供了一种兼顾效率、实时与稳定性的调度解决方案。

关键词:任务调度;Kubernetes;逼近理想解法;任务驱动;延迟优化;集群健康;内存优化
中图分类号:TP319 **文献标识码**:A

A Task-Driven Heterogeneous Task Scheduling Model for Kubernetes

ZHAO Yi, JIANG Lingyun

(*School of Communications and Information Engineering, Nanjing University of Posts and Telecommunications, Nanjing* 210003, *China*)
✉ 719323738@qq.com; jiangly@njupt.edu.cn

Abstract: This paper conducts an optimization study targeting common industrial issues of the Kubernetes default scheduler in enterprise cloud-native environments, including excessive response latency, severe resource fragmentation, and insufficient cluster stability when scheduling heterogeneous resources and diversified workloads. Although existing research has improved scheduling strategies via multi-objective decision-making, machine learning and other approaches, such studies generally suffer from drawbacks such as single optimization objectives, poor dynamic adaptability, limited support for latency-sensitive tasks, and inadequate focus on cluster health status. To address these gaps, a task-driven scheduling model is proposed for heterogeneous tasks on Kubernetes. Combining improved TOPSIS (Technique for Order Preference by Similarity to an Ideal Solution) multi-criteria decision-making with an event-triggered mechanism, the model enables dynamic evaluation of multi-dimensional resources and comprehensive node ranking. Meanwhile, it greatly reduces scheduling latency through resource reservation and parallel binding policies. Experimental results verify that the proposed model outperforms the default scheduler and mainstream optimized algorithms across multiple metrics such as scheduling latency, I/O pressure and memory pressure. It effectively boosts resource utilization and system stability under high-concurrency heterogeneous scenarios, delivering an industrial scheduling solution that balances efficiency, real-time performance and stability.

Key words: task scheduling; Kubernetes; TOPSIS; task-driven; latency optimization; cluster health; memory optimization

0 引言(Introduction)

随着云计算和容器化技术的迅猛发展,Kubernetes[1]已成为云原生应用编排与管理的事实标准,被广泛应用于企业级生产环境中。然而,它的默认调度器在面对异构计算资源(包括

收稿日期:2025-12-09
基金项目:国家自然科学基金项目资助(92067201);江苏省重点研发计划项目资助(BE2020084-4)

www.rjgczz.com

现场可编程门阵列((Field Programmable Gate Array,FPGA)、CPU、GPU 等)[2]和多样化工作负载类型(如计算密集型 AI 任务、内存密集型大数据处理、I/O 密集型流媒体服务等)时,逐渐暴露出多方面的性能瓶颈[3]。传统调度机制基于静态优先级和硬性资源过滤策略[4],难以在调度延迟、资源利用率、系统压力等多个维度间实现有效平衡,尤其在高并发和实时性敏感场景中,容易出现响应延迟高、资源碎片化严重、集群稳定性下降等问题,严重影响用户体验和系统效率。

尽管近年来国内外学者围绕 Kubernetes 调度优化开展了大量研究,提出了包括多目标决策方法(如 TOPSIS(Technique for Order Preference by Similarity to IdealSolution))[5]、机器学习预测模型和任务类型感知策略在内的多种改进方案,这些研究在一定程度上提升了调度性能,但仍存在明显局限:多数方法仅聚焦单一优化目标,缺乏对多维资源的动态协同与权衡能力;对集群负载突发变化和复杂异构环境的适应性较差;在调度实时性方面优化有限,无法有效支持对延迟敏感型应用,同时欠缺对集群健康以及稳定性的研究。因此,构建一个能够同时兼顾效率、稳定性与实时性的调度模型,已成为 Kubernetes 在高性能计算和云原生场景[6]中亟须解决的关键问题。

针对上述挑战,本文提出了一种基于任务驱动的 Kubernetes 异构任务调度模型。该模型融合改进的 TOPSIS 多准则决策算法与事件触发的即时调度机制,通过对 CPU 使用率、内存占用、容器数量和磁盘 I/O 等多维资源指标进行动态标准化与加权评估,实现对集群节点的综合排序与最优选择。同时,引入任务驱动机制,取消传统动态优先级队列,通过资源预留和并行绑定策略大幅降低任务排队与调度延迟。本文还构建了完整的调度延迟数学模型,从理论层面分析和验证了本文方法在降低延迟、缓解系统压力方面的优势。

1 相关工作(Related work)

近年来,国内外学者在 Kubernetes 调度优化领域进行了大量研究,提出了多种改进方案和方法。这些研究主要集中在以下几个方向:

多目标优化调度算法:Gao 等[7]使用层次分析法(Analytic Hierarchy Process,AHP)权重矩阵与逼近理想解相结合的算法来解决异构任务资源分配不均的问题。然而,他们的研究没有考虑集群长期运行后负载不平衡的问题和集群健康性问题;Nguyen 等[8]提到了几个影响 Pod 水平自动扩缩(Horizontal Pod Autoscaler,HPA)性能的应用程序级指标,同时指出了优化 HPA 的方向,但是实验中没有使用 Kubernetes 系统默认指标测试;张磊[9]提出了一种基于改进 TOPSIS 算法的 Kubernetes 调度策略,通过动态权重调整机制适应不同类型的工作负载,但是在针对延迟敏感型任务时调度时延还是很大。

机器学习驱动的智能调度:随着人工智能技术的发展,越来越多的研究将机器学习方法应用于调度优化。Vu 等[10]提出了一种用于容器化应用的混合自动缩放方法,增加了基于机器学习的任务预测模块,并将它与突发识别模块相结合,以使自动缩放决策更加有效。文中还提出了一种 Pod 滚动更新机制,可以避免垂直自动缩放过程中的 Pods 服务中断问题。但是,他们的研究没有考虑集群中的负载均衡问题。肖俊炜[11]设计了一个基于 Pod 副本预测的 Kubernetes 自动伸缩器,以提高资源伸缩效率。但是没有考虑整个集群和 Pod 的长期运行后负载压力的问题,导致集群性能下降。He[12]提出了一种基于深度强化学习的调度框架,能够根据实时集群状态自适应调整调度策略。Yan 等[13]提出了一种基于 Kubernetes 平台的 HANSEL 系统,该系统使用双向长短期记忆(Bidirectional Long Short-Term Memory,Bi-LSTM)负载预测算法来精确预测微服务的负载,并通过强化学习结合被动和主动方法实现主动弹性伸缩,以优化 Kubernetes 的水平自动伸缩策略。该方法结合了人工智能(Artificial Intelligence,AI)模型和机器学习,进一步提升了性能,不过依然没有考虑集群长期运行的负载稳定性。Harichane 等[14]提出了一种用于运行时预测的机器学习方法,为任务选择合适的 CPU 和 GPU 资源,来缩短处理时间并提高异构环境中的用户满意度。但是该方法考虑的资源维度有限。黄敏志[15]提出一种结合随机森林和长短期记忆(Long Short-Term Memory,LSTM)网络的混合模型进行预测的智能调度算法 RNBA 实现预测式弹性扩缩容,但是没有考虑异构任务的多样性。赵逸凡等[16]提出一种基于加权变分减法双向长短期记忆模型(INFO-VMD-SABO-BiLSTM, IVS-BiLSTM)的容器水平伸缩策略,但是只提升了预测精度,没有考虑集群的稳定性与健康性。

在可持续发展背景下,集群能耗优化成为重要研究方向:Shen[17]提出了一种基于碳积分控制大规模集群的绿色优化调度算法,但是在性能指标上没有考虑(具体);Ai[18]提出了一种新型的智能控制网络来提高微电网通信性能,用于解决单一软件定义网络(Software Define Network,SDN)控制器的典型缺点。将 Kubernetes 与 SDN 微服务相结合,可以消除分层控制中的单点故障,缩短应用恢复时间,增强容器的安全性和可移植性。Ye 等[19]提出了一种多目标优化算法,在性能需求和能耗约束之间寻找最优平衡点。Garcia 等[20]提出了一种通用系统来动态调整 Kubernetes 集群规模,通过自动确定 CPU 利用率阈值来满足特定应用的需求,并提供集群自动缩放算法来获得 Kubernetes 集群中的理想节点数量,从而提高集群资源利用率并保证 QoS。但该方法只针对指定节点部署应用,无法保证整个集群的资源利用率。

大多数 Kubernetes 方法侧重于单一优化目标,难以在多维度指标间实现有效平衡。现有方案对动态环境的适应性有限,无法有效应对负载的突发变化。而且现有方法在调度延迟优化方面仍有较大提升空间,特别是对于实时性要求较高的应用场景。

2 基于任务驱动的 Kubernetes 异构任务调度模型(Task-driven heterogeneous task scheduling model for Kubernetes)

Kubernetes 作为当今主流的容器编排平台,它的核心功能是高效、合理地将用户提交的工作负载(Pod)安置到集群中合适的计算节点(Node)上。此过程称为调度(Scheduling),它是一个由多个组件精密协作、基于可插拔策略的复杂决策过程。本文提出的基于任务驱动的 Kubernetes 调度框架如图 1 所示。

从图 1 可以看出,API-Server 负责接收用户通过 kubectl 或其他客户端提交的 Pod 创建请求,并将其持久化到集群状态存储(etcd)中。此时,Pod 的数据被保存,但它的 NodeName 字段为空,标志着它处于“待调度”(Pending)状态。API-Server 的这一角色是整个调度流程的触发器和状态广播器。Scheduler 负载持续监听 API-Server,寻找 NodeName 为空的 Pod。一旦发现此类 Pod,便会启动一个多阶段的筛选与评分决策流程。它自身不直接运行 Pod,也不管理节点,而是纯粹作为一个决策引擎工作。

www.rjgcz.com

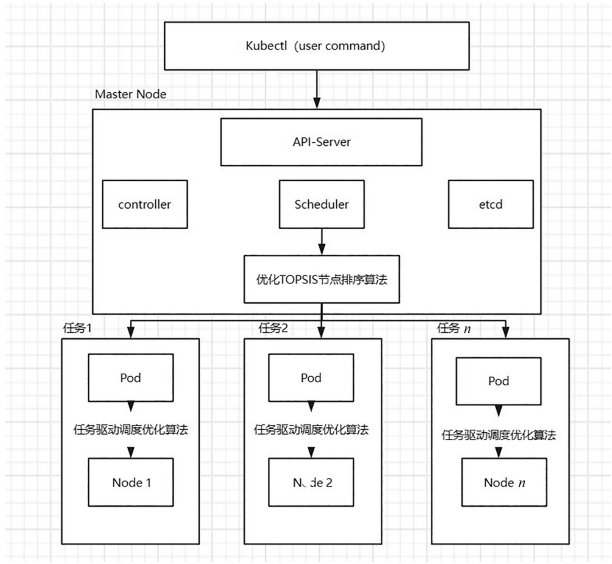

图 1　基于任务驱动的 Kubernetes 调度框架

Fig. 1 Task-driven Kubernetes scheduling framework

本文提出的异构任务调度模型包含优化 TOPSIS 节点排序算法和任务驱动调度优化算法。优化 TOPSIS 节点排序算法基于节点实时的资源指标,标准化处理资源数据,构建决策矩阵,再通过欧几里得距离法结合权重分析计算,得出每一个节点的接近度指数,并进行排序,从而确定最优的节点分数排名;使用任务驱动调度优化算法将 Pod 安置到相应的 Node 中,从而降低任务调度中动态队列阶段产生的缓存延迟。

2.1　参数模型

本文调度算法的通用参数参见表 1。集合 P 是待调度 Pod 的集合,集合 N 是节点的集合,也是在排序阶段的排序对象,RTT 是 Pod 间跨节点通信所需要的往返时间。BW 是集群中 API-Server 可以使用的网络带宽。RTT 和 BW 的取值取决于系统性能。

表1　通用参数定义

Tab. 1 General parameter definitions

符号	物理含义	典型取值	影响维度
P	待调度 Pod 集合	动态变化	系统负载
N	集群节点集合	$\|N\|$ = 节点数	调度复杂度
RTT	跨节点通信往返时延	跨区域≥1 ms,同区域=1 ms	绑定延迟
BW	API-Server 网络带宽	通常≥100 MB/s	数据传输效率

2.2　优化 TOPSIS 节点排序算法

2.2.1　传统 TOPSIS 节点排序算法及问题

在 Kubernetes 集群调度优化中,传统 kube-scheduler 采用的静态优先级和硬性过滤策略虽然能够满足基本的资源分配需求,但在面对异构工作负载和复杂调度场景时逐渐显现出局限性。相比之下,基于多准则决策的 TOPSIS 算法通过引入多维度的动态评估机制展现出显著优势。

该算法的核心思想是通过计算候选节点与理想最优解和最劣解的欧氏距离进行综合排序,实现过程如下:首先,对 CPU 利用率、内存使用率、网络延迟等异构指标进行标准化处理以消除量纲差异;然后,根据实际调度需求为各指标分配动态权重;最后,计算相对贴近度得到节点的优先级评分。这种方法的优势在于能够同时兼顾资源利用率、负载均衡和网络拓扑等多维因素,避免了默认调度器单一维度优化导致的局部最优问题。

在处理 AI 训练任务时可以自动提升 GPU 指标的权重,而在调度微服务时则会优先考虑跨可用区的网络延迟。此外,TOPSIS 算法生成的量化评分还为调度决策提供了可解释性依据,有助于运维人员分析优化调度策略。在实际应用中,为了确保该算法在 Kubernetes 环境中的高效运行,需要结合 Prometheus 时序数据进行节点指标的预计算以降低调度延迟。同时,根据 Pod 的服务质量(Quality of Service,QoS)类别动态调整指标权重,并将 kube-apiserver 带宽和跨可用区延迟等关键网络参数纳入评估体系。通过与传统调度算法在资源利用率、应用响应时间等维度的对比实验,可以验证 TOPSIS 算法在复杂调度场景下的优越性能,特别是在需要平衡多种竞争性指标的大规模集群环境中,该算法展现出了更强的适应性和调度质量。

传统的 TOPSIS 多维度决策算法虽然考虑了多种指标如吞吐量、延迟等,但是并没有考虑集群长期负载工作后的稳定性与系统健康性能。同时在排序阶段后,系统仍存在周期性轮询,因此在时延性能上也有优化空间。

2.2.2　优化 TOPSIS 节点排序算法

本节主要介绍基于逼近理想解算法为节点处理异构资源的能力进行打分排序。

本文采用 CPU 使用率、内存使用率、容器数量与磁盘使用率(I/O 压力)作为节点能力评价指标,用矩阵 $\boldsymbol{X}$ 表示各个节点的指标参数,$\boldsymbol{X}$ 表达式见式(1):

$$\boldsymbol{X}=(x_{ij}),i=1,2,\cdots,n;j=1,2,\cdots,m \tag{1}$$

其中:x_{ij} 表示第 i 个节点的第 j 项指标。

这些指标中,CPU 使用率和内存使用率是正向指标(越大越好),磁盘使用率和网络延迟是负向指标(越小越好)。为了确保不同量纲指标的可比性,需要对它们进行归一化处理。

对于正向指标,采用极差归一化法,将原始值 x_{ij} 按式(2)映射到[0,1]区间,保留其"越大越优"的特性。为了防止在归一化时出现除 0 错误,式(2)中定义了一个极小正数 ε。

$$\boldsymbol{r}_{ij}=\frac{x_{ij}-\min_i x_{ij}}{\max_i x_{ij}-\min_i x_{ij}+\varepsilon} \tag{2}$$

其中:$\boldsymbol{r}_{ij}$ 表示归一化后的指标矩阵。

而对于负向指标,先将它转换为正向数值,再进行极差归一化,具体如式(3)所示:

$$\boldsymbol{r}_{ij}=\frac{\max_i x_{ij}-x_{ij}}{\max_i x_{ij}-\min_i x_{ij}+\varepsilon} \tag{3}$$

这种区分处理确保了所有指标在归一化后具有一致性,为后续欧氏距离计算奠定基础。值得注意的是,当节点资源剩余量总体≤Pod 资源需求量时,需在归一化前排除这些节点,避免无效计算。在完成指标的归一化处理后,下一步需要构建加权归一化决策矩阵以反映不同指标的相对重要性。

(1)根据集群调度策略的需求,为每个评价指标分配相应的权重值 w_j,权重值应该满足式(4):

$$\sum_{j=1}^{m} w_j = 1, w_j \geqslant 0 \tag{4}$$

这些权重可以通过层次分析法(AHP)、熵权法或由运维人员根据经验动态指定。例如:在强调资源利用率的场景中,CPU 使用率和内存使用率指标的权重可能被设置为较高值(如 0.4 和 0.3);而在低延迟优先的场景下,网络延迟的权重(如 0.5)则会显著提升。

由于本文并非着重于调参方面的创新,所以使用的权重系数为比较经典的 CPU 资源密集型任务的系数,主要参考对象为 GitHub 上一个 CRIU[21] 项目的开源权重判定标准。该项目使用一个 Spark 大数据流任务作为测试对象,与本文测试对象特性相符合。具体权重数据如下:CPU 使用率权重为 0.582,内存使用率权重为 0.239,磁盘使用率权重为 0.119,网络延迟权重为 0.060;举证一致性指标经验算约为 0.039,一致性比例经验算为 0.043<0.1 满足一致性要求。

(2)将归一化后的决策矩阵与权重向量逐列相乘,得到加权归一化矩阵 $\boldsymbol{V}_{ij}$,如式(5)所示。这一步骤实质上是将抽象的归一化值转化为具有明确物理意义的加权评分,例如:某节点

www.rjgcz.com

的 CPU 评分表示它在 CPU 维度对调度目标的贡献度。

$$\boldsymbol{v}_{ij}=\boldsymbol{w}_j\times\boldsymbol{r}_{ij} \tag{5}$$

在得到加权归一化矩阵后，下一步需要确定正负理想解并计算各备选节点与它们的欧氏距离。正理想解由每个评价指标在加权矩阵中的最优值构成，代表理论上的最佳调度目标；而负理想解则由各指标的最劣值组成，对应最不理想的调度情况。如式(6)和式(7)所示：

$$v_j^+=\max_{1\leqslant i\leqslant n} v_{ij} \tag{6}$$

$$v_j^-=\min_{1\leqslant i\leqslant n} v_{ij} \tag{7}$$

式(8)为节点 i 到正理想解的欧氏距离，式(9)为节点 i 到负理想解的欧氏距离：

$$D_i^+=\sqrt{\sum_{j=1}^{m}(v_{ij}-v_j^+)^2} \tag{8}$$

$$D_i^-=\sqrt{\sum_{j=1}^{m}(v_{ij}-v_j^-)^2} \tag{9}$$

以 Kubernetes 集群调度为例，正理想解可能由最高 CPU 剩余量、最大内存可用空间和最低网络延迟等指标值组合而成，而负理想解则相反。计算过程中，针对每个候选节点，分别测量它在多维指标空间内与正负理想解的距离，这需要综合考虑所有资源维度的差异。例如：某个节点虽然在 CPU 资源上接近正理想解，但在内存或网络延迟方面可能存在明显差距。这种距离计算能够全面反映节点整体性能与理想状态之间的偏差程度，为后续优先级排序提供量化依据。

(3)计算相似度得分S_i，也就是 TOPSIS 综合得分(贴近理想解的相似度)，如式(10)所示：

$$S_i=\frac{D_i^-}{D_i^-+D_i^+} \tag{10}$$

根据每一个节点获得的综合得分对每一个节点进行排序，最终得分靠前的节点会优先进行调度任务。

2.3 任务驱动的调度优化算法及算法延迟对比

2.3.1 任务驱动的调度优化算法

传统的 Kubernetes 采用周期性扫描和队列批处理模式，调度器定期检查待调度 Pod 队列，Pod 需要等待调度周期到来才能进入调度流程，这种做法在高负载时产生“队列堆积”效应，增加排队延迟。本文中提出的任务驱动机制采用事件触发调度，中和了排队效应产生的延迟，减少了调度延迟。

事件触发调度过程为 MasterNode 通过观察者模式注册各类事件监听器(如资源变化、任务超时、外部指令等)。当事件发生后，直接调用调度决策引擎，跳过周期性轮询从而减少延迟。例如：当 TASK_CREATED 事件触发时，立刻通过 Scheduler.resource(TASK)预生成调度策略，下达给节点完成任务的分配与绑定。

传统 KCSS 采用动态优先级队列管理待调度任务，虽然通过优先级机制实现了任务差异化管理，但本质上仍属于批处理模式，任务必须经历排队等待过程。

任务驱动型 KCSS 创新性地采用事件触发机制，在监测到新任务到达时立即启动调度流程，完全规避了队列等待环节。这种设计使得系统响应时间不再受队列长度影响，尤其在高负载场景下，避免了传统调度器常见的“队列堆积”问题，实现了稳定的低延迟特性。

本文的这种“决策即占用”的模式消除了资源竞争带来的不确定性延迟。下面将分别介绍默认 K8s 调度算法和优化 KCSS 调度算法，与本文提出的基于任务调度的优化 KCSS 算法做比较。

2.3.2 各算法的延迟对比

(1)默认 Kubernetes 调度算法延迟

默认 Kubernetes 调度器总延迟L_{default}包括队列等待延迟、决策阶段延迟和绑定延迟，具体见式(11)。

$$L_{\text{default}}=\frac{|P_{\text{pending}}|}{\mu_{\text{sched}}}+|N|\cdot\tau_{\text{filter}}+|S|\tau_{\text{score}}+\tau_{\text{etcd}} \tag{11}$$

其中：等号右侧第 1 项是队列等待延迟，该项与待调度 Pod 数量$|P_{\text{pending}}|$成正比，与调度器处理速率μ_{sched}成反比，直观体现了传统调度器在高负载场景下的线性扩展缺陷；等号右侧第 2 项和第 3 项是决策阶段延迟，包含节点过滤和策略评分两个部分，其中节点过滤延迟随集群规模$|N|$扩大线性增长，评分延迟与启用策略数$|S|$正相关，这种设计导致大规模集群中决策延迟成为主要瓶颈；τ_{etcd}是绑定延迟，它主要取决于分布式存储系统的写入性能。

式(11)不仅定量地揭示了默认调度器的性能限制，更重要的是为后续优化方向提供了理论依据，特别是队列管理、决策算法和绑定机制 3 个维度的改进空间。

在式(11)中μ_{sched}代表的是调度器处理速率(Pods/s)，主要是系统吞吐量的影响，典型的取值区间(依赖控制器性能)为 50～100。由于本文的实验无法租用云服务器改变框架，因此实际实验取值比较小，约为 51.36，使用 Linux 系统下的 K8s CronJob 工具统计得到。

τ_{filter}主要受预选阶段效率的影响，代表了单节点过滤操作耗时。本文实验中预设的阿里云节点的单节点过滤操作耗时为 0.1～0.3 ms。τ_{score}代表单策略评分计算耗时，一般取0.05～0.1 ms。τ_{etcd}代表提交到 etcd 的数据提交耗时，受 etcd 负载影响，一般取 1～5 ms。

(2) KCSS 优化调度算法的延迟

基于随机优化的 Kubernetes 定制调度器算法(KCSS)通过动态优先级机制和概率化决策策略，显著优化了传统调度器的性能。KCSS 的总延迟包括队列等待、决策和绑定 3 个阶段。其中队列等待阶段通过积分运算量化优先级对队列等待时间的非线性压缩效应(如高优先级的任务(时间敏感型 Pod)的等待时间可以被指数级缩减，而普通任务则维持基础调度序列)，有效解决了传统 FIFO 队列的公平性与实时性矛盾，从而能适应不同的异构任务与高并发任务。决策阶段采用 Metropolis-Hasting 随机游走算法，当候选节点集规模增大时，算法通过调整探索参数来平衡搜索效率与解的质量。绑定阶段进一步区分数据传输和跨可用区网络开销，凸显了分布式环境下的物理约束。证明了 KCSS 在动态负载下的适应性优势：通过不同优先级的队列管理，概率驱动的智能搜索，为云原生调度器的 QoS 保障提供了新的框架。

KCSS 算法调度延迟L_{KCSS}如式(12)所示：

$$L_{\text{KCSS}}=\int_{t_{\text{arrival}}}^{t_{\text{dispatch}}}\frac{\mathrm{d}t}{1+\eta\text{Priority}(t)}+\tau_{\text{base}}+\kappa(1-e^{-\beta|N_{\text{candidate}}|})+\frac{\text{data}_{\text{pod}}}{BW}+\text{RTT}\cdot 1_{\text{cross-zone}} \tag{12}$$

其中：t_{dispatch}和t_{arrival}分别代表 Pod 调度开始决策时的时间戳和 Pod 到达调度队列的时间戳。动态优先级中的 η 代表队列加速系数，在本文中直接取经典案例中调好的参数。τ_{base}代表基础决策时间，主要取决于调度框架的固定开销。κ 是节点搜索开销系数，与候选节点数成正相关，取值方法与队列加速系数相同。β 是随机游走收敛速率，是 Metropolis-Hasting 随机游走算法中的参数，等号右侧最后 1 项是跨区域指示函数，是一个 01 布尔值，如果 2 个节点在一个区域则不需要计算 RTT，如果不在一个区域则需要再加上 RTT。动态优先级函数 Priority(t)尤为重要，这是一个多因素复合函数，受多种维度的影响，具体数学表达式如式(13)所示：

$$\text{Priority}(t)=w_p\cdot\left(\frac{\text{deadline}_p-t}{\text{deadline}_p-t_{\text{arrival}}}\right)^{\lambda} \tag{13}$$

其中：w_p称为静态权重，在云原生环境中代表 Pod 的基准优先级；$\left(\frac{\text{deadline}_p-t}{\text{deadline}_p-t_{\text{arrival}}}\right)^{\lambda}$是时间紧迫性因子，它的分子表示时间富裕度也即当前时间 t 到截止时间 deadline_p 的差值，分母表示 Pod 允许调度的时间窗口，就是从到达时间t_{arrival}到deadline_p的时间差；指数 λ 代表紧迫性敏感度，值越大则越接近截止时间的时候优先级飙升得越快。

www.rjgcs.com

参数在代码中都配置在 params 函数中。

```
params = {
    'β' : 0.5,      #  队列加速
    'γ' : 0.1,      #  收敛速率
    'ε' : 0.2,      #  探索概率
    'κ' : 2.0,      #  节点搜索开销(ms)
}
```

而 deadline$_p$ 时间戳的配置如下所示。

```
annotations:
    scheduling.alpha.
    kubernetes.io/deadline :
    "2023-07-01T15:04:05Z"
```

(3)本文的优化调度算法的延迟

本文提出的任务驱动型优化 KCSS 调度算法的延迟 $L_{优化KCSS}$ 如式(14)所示:

$$L_{优化KCSS}=\log\left(1+\frac{|N_{feasible}|}{K_{parallel}}\right)+\max\left(\frac{data_{pod}}{BW},RTT_{min}\right)+\frac{T_{commit}}{B_{queue}} \tag{14}$$

其中:$1+\frac{|N_{feasible}|}{K_{parallel}}$ 代表了并行搜索延迟,体现了算法通过多线程并行处理带来的性能提升。$K_{parallel}$ 表示并行工作线程的数量;$N_{feasible}$ 是满足调度条件的候选节点数。整体呈指数增长的原因是反映并行搜索的边际收益递减(更多协程的加速效果逐渐减弱)像多线程爬楼梯,线程越多爬得越快,但超过一定数量后协调开销抵消收益。例如:当节点数从 100 增加到 1000 时,延迟仅增加约 2 倍而非线性增长。这种设计特别适合大规模集群环境,能够有效控制决策时间。$K_{parallel}$ 和 $N_{feasible}$ 在本文中是先静态取值,然后随实验再进行观测计算,静态取值的数值已经在 yaml 配置文件中写死,默认初始并行线程 16 满足可调度节点数=1,之后的参数变化可以在 API-Server 中随时用命令查看。

$\left(\max\frac{data_{pod}}{BW},RTT_{min}\right)$ 代表绑定延迟,是指将 Pod 配置从控制平面传到目标节点,耗时取决于数据大小和带宽。它在数据传输延迟 $\frac{data_{pod}}{BW}$ 和网络延迟 RTT_{min} 两者之间取最大值。网络延迟是在最低往返时间中取最大值,体现木桶效应(两者谁更慢谁主导)。

$\frac{T_{commit}}{B_{queue}}$ 代表提交过程的延迟。分子 T_{commit} 表示将调度决策写入 etcd 所需要的时间;分母 B_{queue} 表示 API-Server 的请求处理吞吐量。

关于分母的取值,在不同运行环境中的取值并无绝对。例如:在轻量化 MiniKube 平台上(2 核 4 GB)为 5～50(请求数/s),在一般的原生单机 Kubernetes 环境中取值为 200～400(请求数/s)。测量方法可以使用本地 Hey 命令计算,也可以使用专业压测工具 K6 进行测试。本文物理机一次理想实验情况下的测试数据如下,数值为 361.33。

```
http_reqs.....................: 21690   361.33/s
iteration duration.............: avg=827.5ms min=10
```

该延迟模型清楚地展示了优化 KCSS 算法相比传统调度器加入任务驱动算法的优势,当有任务流进来时会立即启动调度流程,并且保持高并发状态,挤掉了传统 KCSS 中的队列延迟从而进一步优化延迟。

3　实验与结果分析(Experiments and result analysis)

为了验证提出的异构任务调度模型的可用性与性能,本节将从调度延迟、CPU 使用率、I/O 压力和内存压力等 4 个方面进行实验验证。选择默认 kubectl 算法、KCSS 算法、AKCSS 算法以及 RNBA 算法作为对标对象。

3.1　实验环境

为了比较多种对比算法,采用统一的实验环境和节点配置,具体配置如下:

3 个节点的配置都为 2CPU2GPU 的节点,全部使用 k8sv1.26.8 版本。

内存使用率为 1250 MB/2048 MB,业务内存 983 MB(48%)+系统 6%自适应权重选择内存 65% 的 node(1331 MB)。

在数据采集方面,本实验采用在集群中部署 prometheus 探针的方法,在探针网站的可视化窗口中采集数据。

使用一个部署在本地的集群进行实验测试,该集群包括 1 个 JobManager 节点和 2 个 TaskManager 节点,3 个节点都以虚拟机的方式部署在 VMware WorkStation Pro 软件上,用来模拟 3 台物理机节点,3 台机器分别搭建了版本兼容的 Docker 和 Kubernetes。总体集群架构如图 2 所示。

图 2　集群结构

Fig. 2 Cluster structure

集群所使用节点的配置见表 2。表 2 中所使用的 iso 的印象文件配置在本地。

表2　节点配置信息

Tab. 2 Node configuration information

计算机资源	版本信息
操作系统	LinuX CentOS 7 64 位
处理器	16 核
硬盘(SCSI)	100 GB
CD(IDE)	C:\Users\DELL\Downloads\CentOS-7-x86_64-DVD-2207-02.iso
网络	NAT
Docker	21.10.4
Kubernetes	1.26.8

本文实验中所使用的节点管理工具为 MobaXterm 软件,通过配对 ip 地址密钥方式,可以使用 Muti-Paste 模式一键下达命令,并在多个虚拟机上执行,不需要再一个一个地输入指令。本节所有测试用例均在该模式下进行,在该模式下的多节点运行界面如图 3 所示。

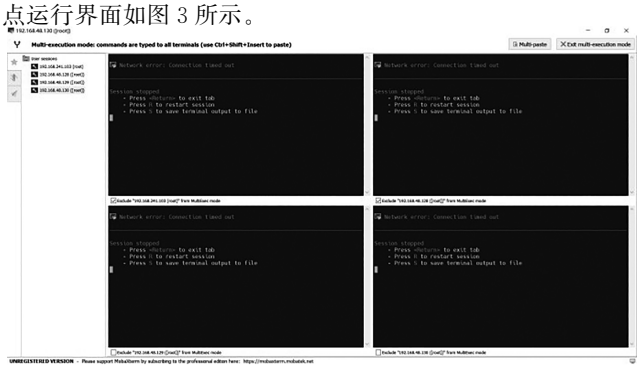

图 3　节点命令控制界面

Fig. 3 Node command control interface

www.rjgcz.com

3.2 测试指标

(1)Pod 调度延迟

Pod 调度延迟是最能反映集群处理请求响应敏感度的指标,直接把 4 种集群下 Pod 调度开始调度的时间轴取出就能知道调度延迟的差异。

(2)CPU 使用率

在 Kubernetes 集群的性能实验中,CPU 使用率(CPU Utilization)用于量化计算资源的利用效率。它表示在特定时间窗口内,节点或容器实际消耗的 CPU 时间占其可用 CPU 总时间的百分比。该指标反映了工作负载对计算资源的占用情况,是评估调度算法效率、资源分配合理性以及系统扩展性的重要依据。计算公式如式(15)所示:

$$C_n(t)=\left(\frac{\text{cpu}_{\text{used}}}{\text{cpu}_{\text{available}}}\right)\times 100\% \tag{15}$$

(3)I/O 压力

I/O 压力是衡量 Kubernetes 节点存储子系统性能瓶颈的关键指标。高 I/O 压力表现为显著的 I/O 操作延迟增加和吞吐量下降,会直接导致 Pod 中的应用性能劣化,并可能触发 Pod 驱逐,最终影响服务质量和集群稳定性。Kubernetes 中的 I/O 压力源于对节点本地存储和运行时存储的过度争用。它的数学定义如式(16)所示:

$$F_n(t)=\frac{D_{\text{avail}}^{\text{threshold}}-D_{\text{avail}}^{n}(t)}{D_{\text{avail}}^{\text{threshold}}} \tag{16}$$

其中:$D_{\text{avail}}^{n}(t)$表示 n 节点在 t 时刻主要文件系统(如 nodefs)的可用磁盘空间;$D_{\text{avail}}^{\text{threshold}}$表示磁盘空间的驱逐阈值(如 5 GiB)。

(4)内存压力

在 Kubernetes 集群资源管理中,内存压力是衡量节点内存资源稀缺性的关键指标。高内存压力会触发 Pod 驱逐机制,导致服务中断、调度开销增加及系统稳定性下降。因此,将内存压力维持在一个极低的水平是集群自动扩缩容(Autoscaling)和调度器(Scheduler)优化的重要目标。本文将它形式化为一个可量化的极小化指标,并建立相应的数学模型。内存压力指标定义为当前可用内存偏离预设安全阈值的相对程度。对于一个拥有 n 节点的集群,它的全局内存压力指标可表述为各节点压力值的聚合。对于节点 n 在 t 时刻的内存压力指数如式(17)所示:

$$P_n(t)=\frac{M_{\text{avail}}^{\text{threshold}}-M_{\text{avail}}^{n}(t)}{M_{\text{avail}}^{\text{threshold}}} \tag{17}$$

其中:$M_{\text{avail}}^{n}(t)$表示节点 n 在 t 时刻的 memory.available 实测值;$M_{\text{avail}}^{\text{threshold}}$表示预设的驱逐阈值(如 500 MiB),此值为集群策略参数。测量值 memory.available 和驱逐阈值$M_{\text{avail}}^{\text{threshold}}$的物理含义是指在评估时刻,节点内存子系统中,无需等待调度交换(Swap)或终止进程,即可立即分配给用户的应用程序(包括 Pod 中的进程)或使用的内存容量估计值。

根据以上指标,可以综合衡量节点的可用性、实用性以及集群健康程度,从而提高用户使用的体验感。从用户请求响应的时间角度看,Pod 调度延迟越低,集群对任务的响应越快;从对资源的消耗角度来看,CPU 使用率越高代表需求的计算机资源越多;在 Kubernetes 优化这一方面来看,Kubernetes 平台已经完善了计算机资源分配,所以一般在遇到 CPU 资源集中型任务时,多数会选择在任务拆分层面对任务进行拆分而不是在集群端进行优化;从集群健康以及可用性的角度来看,I/O 压力和内存压力指标越低,则集群的承载能力和韧性越强,在遇到短时高频的任务流来袭时任务更加不容易崩溃,同时在实际使用中也更加省心。

3.3 实验结果与数据分析

3.3.1 实验结果

(1)Pod 调度延迟

如图 4 所示,任务驱动 KCSS 算法是最时间敏感的,启动时间要比其他 2 个 KCSS 算法要快,这是因为使用任务驱动算法挤掉了原本的任务缓存池,当有任务进来时会立马启动调度流程。本文算法的启动时间比现有的 AKCSS 算法要快 11%,所以本文模型对于原本的调度算法来说提升是巨大的。

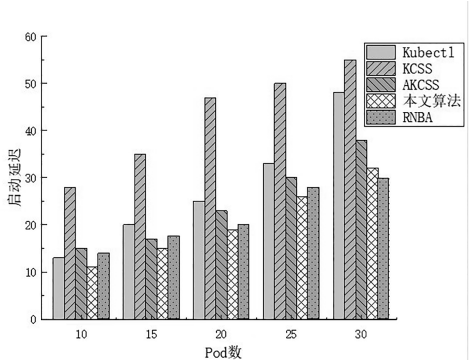

图 4 各算法的 pod 调度延迟

Fig. 4 Pod scheduling latency of different algorithms

(2)CPU 使用率

如图 5 所示,可以分析出 Kubernetes 采用默认调度器(kube-scheduler)基于资源请求(Requests)和静态优先级的调度策略。呈现出 CPU 资源调度速度开始对比较慢,而且无法应对 CPU 资源密集型任务,在启动集群后的 12 min CPU 资源调度才开始快速攀升。

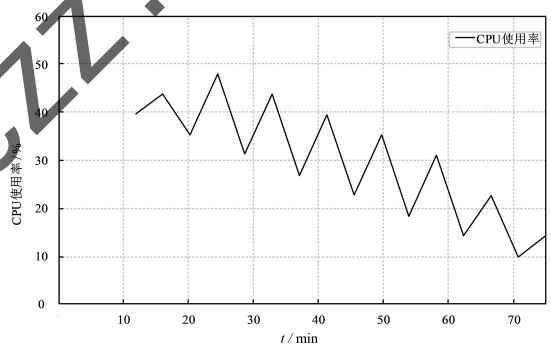

图 5 Kubectl 的 CPU 使用率

Fig. 5 CPU usage of Kubectl

Kubernetes 默认调度器在 CPU 使用率管理上表现出静态性和保守性,虽保障了基础稳定性,但牺牲了资源效率。

经典 KCSS 调度策略下的 CPU 使用率变化曲线如图 6 所示。可以看出,相较于默认的 Kubectl 调度方法,KCSS 调度模型对 CPU 资源的调度明显更快,CPU 资源的攀升速度和开始攀升的时间点都更快,所以在面对异构任务时更有优势。

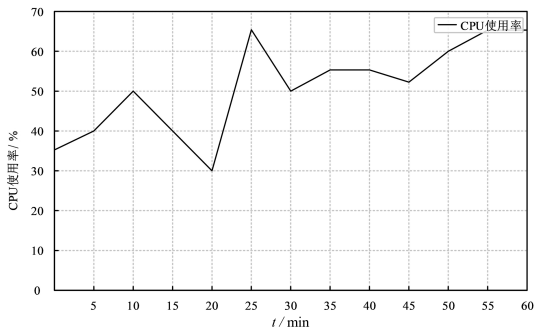

图 6 KCSS 调度策略下的 CPU 使用率

Fig. 6 CPU usage under KCSS scheduling policy

www.rjgczz.com

KCSS 作为一种基于随机优化和动态资源感知的调度算法,在 CPU 使用率管理上展现出显著区别于 Kubernetes 默认调度器的特性。它通过动态负载均衡、实时资源预测和智能超售策略,实现了高资源利用率与稳定性的协同优化。

如图 7 所示,AKCSS 调度策略下,启动 CPU 资源调度的速度和攀升时间点提升的非常多,前段有一段空白是因为在评价权重和排序阶段,阶段结束之后就会立刻开始调度资源。

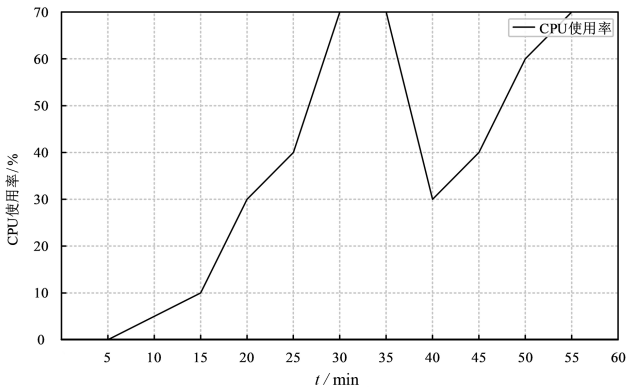

图 7　AKCSS 调度策略下的 CPU 使用率

Fig. 7 CPU usage under AKCSS scheduling policy

AKCSS 算法在设置权重分析时,根据任务的资源给定权重偏向,故在提交 30 个容器的时候,响应速度分别是默认算法以及 KCSS 算法的 1.544 倍和 1.543 倍。

如图 8 所示,基于任务驱动的优化 KCSS 算法的整体走势与 AKCSS 算法相似,都有 2 个高峰期。启动攀升和衰落的时间点也类似。这是因为本文的调度算法使用了设置权重分析、节点排序策略,所以大体走势趋同。

但是,本文的使用率变化相较于 AKCSS 算法没有那么"平滑",在整体走势相似的情况下有很多上下波动。和其他算法相比,本文方法产生的波动也是远远大于其他 3 种算法。这是因为本文算法的本质是通过剔除调度过程中的缓存池来提高性能,没有了缓存池,调速机制就会变成即时调用。当一个节点有任务进来会立刻开始进程,然后挂起,当另一个节点有任务又会立即开始进程,所以在图 8 会产生不下滑现象。

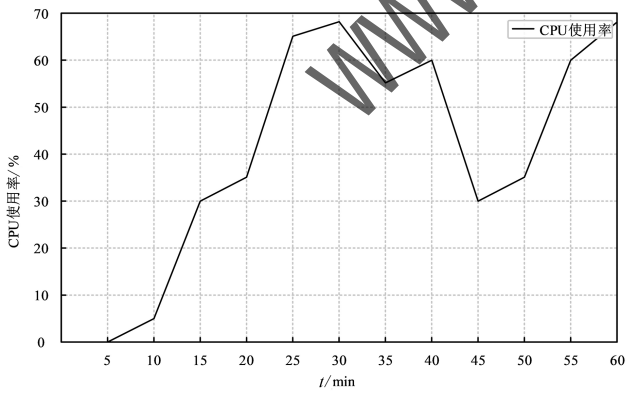

图 8　基于任务驱动的 KCSS 调度算法下的 CPU 使用率

Fig. 8 CPU usage under task-driven KCSS scheduling algorithm

(3)I/O 压力

如图 9 所示,相较于 AKCSS 算法,本文算法进一步降低了系统的 I/O 压力。其中 AKCSS 算法对比经典的 KCSS 算法,在 I/O 压力优化上的效果最显著,能优化 20%左右。

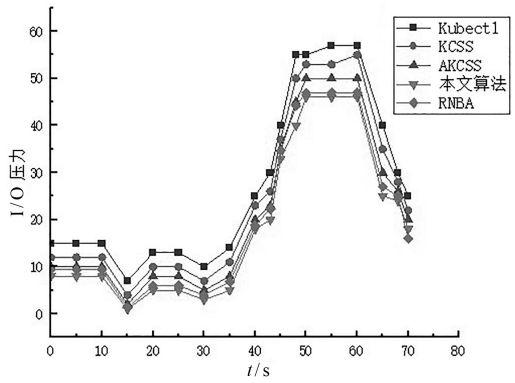

图 9　各算法的 I/O 压力指数变化

Fig. 9 I/O pressure index variation of different algorithms

(4)内存压力

Kubectl 算法、KCSS 算法、AKCSS 算法、基于任务驱动的 KCSS 调度算法和 RNBA 算法下集群的内存压力变化如图 10 所示。可以看到,相较于其他 4 种算法,基于任务驱动的 KCSS 算法在对集群系统的内存压力上是最小的。

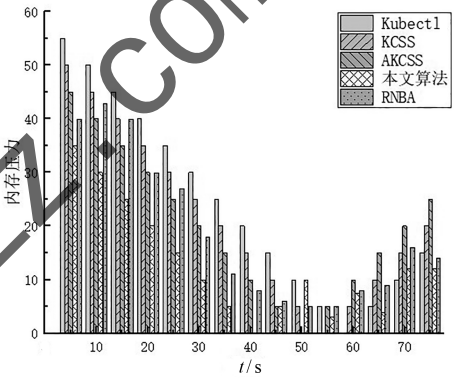

图 10　各算法的内存压力指数变化

Fig. 10 Memory pressure index variation of different algorithms

3.3.2　实验结果数据分析

单次比较理想的一次实验数据见表 3。本文算法在调度延迟、I/O 压力与内存压力方面相较于默认调度方法和经典 KCSS 算法,有很大程度上的提升。和已经有很大改进的 AKCSS 算法相比,也有一些程度的提升。

表3　单次实验数据

Tab. 3 Single experiment data

算法	Pod 调度延迟/ms	CPU 使用率/%	I/O 压力(请求数量)	内存压力/%
默认调度器	137	67	66	65
KCSS	280	68	45	61
本文算法	115	75	25	54
AKCSS	130	72	38	56
RNBA	128	78	30	60

为了实验的准确性,不能只采用 1 次实验的数据为标准,所以本文采取了 10 次实验后取平均值的方法来分析数据。10 次实验取平均之后的实验数据见表 4。

从表 4 可以看出:(1)从调度延迟方面看,本文算法对比经典的 KCSS 算法调度延迟降低了 58%,对比 AKCSS 算法调度速度提升了 11%,是在高并发环境下的又一次速度提升;(2)在 I/O 压力方面,本文算法对比默认调度的压力降低了 56%左右,对比 AKCSS 算法降低了 31%左右;(3)在内存压力方面,本文算法对比默认调度器降低了 18%左右,对比 AKCSS

www.rjgczz.com

算法降低了6%。

表4 10次实验平均数据

Tab.4 Average data of 10 experiments

算法	Pod调度延迟/ms	CPU使用率/%	I/O压力(请求数量)	内存压力/%
默认调度器	140.2	68.6	62.4	66.3
KCSS	283.5	66.4	42.7	62.1
本文算法	119.8	73.3	27.2	54.6
AKCSS	134.7	70.1	39.6	58.8
RNBA	130.4	75.2	35.5	56.7

因为采用了10次实验取平均的操作,需要补充标准差结果和置信区间分析。将10次实验的原始数据和平均数据按标准差公式$\sqrt{\frac{1}{9}\sum_{i=1}^{10}(x_i-\bar{x})^2}$,$n=10$处理$x_i$为单次实验的原始数据,$\bar{x}$是上表中的统计平均数据,数据标准差结果见表5。

表5 实验数据标准差结果

Tab.5 Experimental data standard deviation results

算法	Pod调度延迟/ms	CPU使用率/%	I/O压力(请求数量)	内存压力/%
默认调度器	3.1	1.3	2.8	1.4
KCSS	3.8	1.1	2.1	1.1
本文算法	2.3	1.2	1.5	1.0
AKCSS	2.5	1.8	1.0	1.4
RNBA	2.7	1.1	1.6	1.3

从结果来看,所有算法的标准差均处于较小范围(最大仅3.8),说明10次重复实验的数据重复性、可靠性较高,实验操作或环境波动的影响有限。在延迟方面,KCSS算法波动最大,稳定性最差,本文算法的标准差最小,稳定性较高。

4 结论(Conclusion)

本文的基于任务驱动的优化KCSS调度算法在降低延迟以及系统压力方面有一定的提升,通过权重排序和任务驱动挤占缓存的方法,取消了任务进入集群后固有的pending状态,从而提升速度和性能,对用户使用的舒适度和集群系统的健康性有一定的提升。代价是因为更复杂的机制,从而造成了CPU使用率的升高。成熟的商用集群编排工具本身对CPU密集型任务和高CPU使用场景就有一定的应对能力,而且在节点级的任务中,大部分研究人员在面对CPU资源密集型的异构任务时,一般会采取在任务侧拆分任务的方法来保证集群不崩塌,大部分工作都是做在任务侧的,而不会从集群的角度下手,所以我们认为这些代价可以接受。

实验结果表明,本文算法能够有效支持高并发、低延迟的异构任务调度场景,显著提升集群的整体资源利用率与系统稳定性。

本文不仅为Kubernetes在异构环境中的资源调度提供了新的算法框架和实现路径,也为构建高性能、高可靠的云原生基础设施提供了理论依据和工程实践参考,对推动容器化技术在企业复杂应用场景中的落地具有重要的实际意义与推广价值。

参考文献(References)

[1] Goethals T, De Turck F, Volckaert B. Extending Kubernetes clusters to low-resource edge devices using virtual kubelets[J]. IEEE Transactions on Cloud Computing, 2020, 10(4): 2623-2636.

[2] Shrimal G. Using heterogeneous cloud computing to manage resources in sustainable cyber-physical systems [C/OL]// SMART, 2022: 153-157. https://ieeexplore.ieee.org/abstract/document/10047642/.

[3] 董爱强,胡波,胡学勇,等.基于K8S插件的容器集群资源自动弹性伸缩技术优化研究[J].自动化与仪器仪表,2024,(11):198-202.

[4] 唐毅,张逍遥,高森,等.基于K8s的异构环境混布智能算力调度平台设计与实现[J].通讯世界,2025,32(2):178-180.

[5] Nguyen N T, Kim Y. A design of resource allocation structure for multi-tenant services in Kubernetes cluster[C/OL]// APCC, 2022: 651-654. https://ieeexplore.ieee.org/abstract/document/9943782/.

[6] Yin J, Zhao Y, Wang H. A static task allocation and scheduling algorithm for Kubernetes cluster [C/OL]// ICISCAE, 2024: 175-179. https://ieeexplore.ieee.org/abstract/document/10761792/.

[7] Gao R, Xie X, Guo Q. K-TAHP: a Kubernetes load balancing strategy base on TOPSIS+AHP[J]. IEEE Access, 2023, 11: 102132-102139.

[8] Nguyen T T, Yeom Y J, Kim T, et al. Horizontal pod autoscaling in kubernetes for elastic container orchestration[J]. Sensors, 2020, 20(16): 4621.

[9] 张磊.面向无人机直播的边缘服务器资源预测与调度研究[D].西安:西安工业大学,2024:11-14.

[10] Vu D D, Tran M N, Kim Y. Predictive hybrid autoscaling for containerized applications[J]. IEEE Access, 2022, 10: 109768-109778.

[11] 肖俊炜.基于负载预测的Kubernetes容器自动伸缩方法研究[D].长沙:中南林业科技大学,2025:32-46.

[12] He R J. Parallel machine scheduling problem with time windows: a constraint programming and tabu search hybrid approach[C/OL]// ICMLC, 2005: 2939-2944. https://ieeexplore.ieee.org/abstract/document/1527445/.

[13] Yan M, Liang X M, Lu Z H, et al. HANSEL: adaptive horizontal scaling of microservices using Bi-LSTM[J]. Applied Soft Computing, 2021, 105: 107216.

[14] Harichane I, Makhlouf S A, Belalem G. KubeSC-RTP: smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems[J]. Concurrency and Computation: Practice and Experience, 2022, 34(21): e7108.

[15] 黄敏志.面向Kubernetes的调度策略研究[D].南京:南京邮电大学,2024:9-22.

[16] 赵逸凡,江凌云.基于混合预测模型的容器水平伸缩策略研究[J].软件工程,2025,28(6):24-29.

[17] Shen Z. Design of energy consumption prediction and optimization algorithm for central air conditioning systems integrating deep learning[C/OL]// ICIRDC, 2023: 722-727. https://ieeexplore.ieee.org/abstract/document/10510978/.

[18] Ai F. Design and implementation effect of intelligent algorithms for hotel energy consumption management[C/OL]// ICIRDC, 2024: 144-149. https://ieeexplore.ieee.org/abstract/document/11146163/.

[19] Ye X, Yin Y, Lan L. Energy-efficient many-objective virtual machine placement optimization in a cloud computing environment[J]. IEEE Access, 2017, 5: 16006-16020.

[20] Garcia M A, Lorenzon A F. A systematic literature review of I/O optimization in HPC and cloud computing environments [C/OL]// SBACPADW, 2024: 6672. https://ieeexplore.ieee.org/abstract/document/10764543/.

[21] 胡中玉,赵长名,王旨.基于分层特性的容器预转储实时迁移方法[J].软件导刊,2025,24(6):120-127.

作者简介:

赵 羿(2001-),男(汉族),常州,硕士生。研究领域:云原生调度。

江凌云(1974-),女(汉族),合肥,副教授,硕士。研究领域:下一代网络技术、泛在无线通信、多网融合协同、6G通信系统。本文通信作者。

www.rjgcz.com