Kubernetes 异构集群多指标调度策略研究

孙贵发[1],孙建鹏[2],解西国[2]
(1.郑州大学计算机与人工智能学院,河南 郑州 450001;2.曙光信息产业(北京)有限公司,北京 100193)

摘　要: 异构集群中充分利用各节点资源是提高集群资源利用率的关键。Kubernetes 作为容器编排领域的首选系统,其主要功能之一是将作业调度到合适节点。合理调度策略能够减少资源碎片的产生,提高节点资源利用率,减少作业等待时间。在异构集群资源调度场景下,Kubernetes 默认的中央处理器(CPU)与内存调度算法已难以有效应对多维资源作业调度需求。面对这一挑战,提出一种融合层次分析法(AHP)和装箱调度思想的多指标调度算法,不仅考虑了传统的 CPU 和内存资源指标,还融入了存储、类 GPU 加速卡、镜像等多种资源指标,实现了更全面的资源评估,同时将用户主观决策与节点客观资源限制量化相结合,从多个节点中决策出最符合作业需求的节点进行调度,并基于 Kubernetes 调度框架设计了自定义调度器。使用多指标调度算法的自定义调度器能够更高效利用节点资源,减少因资源碎片化造成的作业等待。在异构集群中对不同类型作业的混合部署进行实验,实验结果表明,多指标调度算法能够有效提高集群资源利用率,减少了 49.79%的 Pod 等待调度时间,这为未来大型云平台和数据中心在调度策略上的优化提供了重要参考。

关键词: 异构集群;容器编排;Kubernetes 调度策略;装箱调度;层次分析法;调度框架
中图分类号: TP391　　**文献标志码:** A　　**DOI:** 10.19678/j.issn.1000-3428.0070438

Research on Multi-Criteria Scheduling Strategies for Kubernetes Heterogeneous Clusters

SUN Guifa[1], SUN Jianpeng[2], XIE Xiguo[2]
(1. School of Computer Science and Artificial Intelligence, Zhengzhou University, Zhengzhou 450001, Henan, China;
2. Dawning Information Industry (Beijing) Co., Ltd., Beijing 100193, China)

【Abstract】 The efficient utilization of resources across nodes in a heterogeneous cluster is pivotal for enhancing the overall resource utilization of the cluster. One of the primary functions of Kubernetes, the preferred system for container orchestration, is to schedule pods onto appropriate nodes. A well-designed scheduling strategy can reduce resource fragmentation, enhance the utilization of node resources, and minimize job waiting times. In the context of resource scheduling within heterogeneous clusters, the default Central Processing Unit (CPU)- and memory scheduling algorithms of Kubernetes have become inadequate for addressing the demands of multi-dimensional resource job scheduling. To address this challenge, a multi-criteria scheduling algorithm integrating an Analytic Hierarchy Process (AHP) and binpacking scheduling concepts is proposed. This algorithm considers not only traditional CPU and memory resource indicators but also incorporates various other resource indicators such as storage, GPU-like accelerator cards, and images, enabling a more comprehensive resource assessment. It also combines subjective user decisions with quantified objective node resource constraints to select the most suitable node from multiple candidates for job scheduling. A custom scheduler is designed based on the Kubernetes scheduling framework. The use of a custom scheduler with a multi-criteria scheduling algorithm can more efficiently utilize node resources and reduce the job waiting time caused by resource fragmentation. Experiments on mixed deployments of different types of jobs in heterogeneous clusters demonstrate that the multi-criteria scheduling algorithm can effectively increase cluster resource utilization and reduce Pod scheduling wait times by 49.79%. This provides an important reference for optimizing scheduling strategies for future large-scale cloud platforms and data centers.

【Key words】 heterogeneous cluster; container orchestration; Kubernetes scheduling strategy; binpacking scheduling; Analytic Hierarchy Process (AHP); scheduling framework

基金项目: 国家重点研发计划(2021YFB0300200)。
作者简介: 孙贵发,男,硕士研究生,主研方向为容器编排与调度;孙建鹏(通信作者),高级工程师、博士;解西国,博士。
收稿日期: 2024-10-08　　**修回日期:** 2025-01-06
E-mail: sunjp@sugon.com

0 引言

随着大规模计算技术的持续演进，云计算作为分布式计算的前沿代表，显著提高了计算资源的利用效率[1-2]。以 Docker 为代表的容器虚拟化技术[3-4]的成熟，解决了复杂应用开发过程中环境一致性的难题。鉴于成本效益的考虑与安全性的驱动，科学、工程、大数据和人工智能领域的研究者纷纷将目光转向大规模计算平台，推动了计算作业与服务作业由传统架构向云原生架构转型[5]。Kubernetes 作为目前最流行与功能最强大的容器管理平台[6]，凭借其卓越的容器编排能力，已成为大规模计算系统中部署容器化应用的事实标准。

在构建高效运行大规模系统的过程中，将作业调度到合适节点是核心环节[7]。Kubernetes 默认调度器具有负载平衡、弹性伸缩等良好特性，能够应对通用环境[8]。但在复杂异构集群中，Kubernetes 基于 CPU 与内存的默认调度策略已无法满足集群复杂的多维异构资源的挑战[9]，特别是默认负载均衡策略在处理高资源需求作业时，可能会产生集群资源充足但节点资源不足，导致无法调度。为应对此难题，在 Kubernetes 调度策略方面，研究人员展开了一系列研究工作。文献[10]引入遗传算法，提出一种细粒度资源调度策略，以应对 CPU-GPU 异构算力平台的负载均衡与 GPU 显存资源分配问题。文献[11]提出一种基于资源利用率的自动缩放系统(RUBAS)，可以动态调整 Kubernetes 集群中运行容器的分配。文献[12]提出一种面向分布式可分负载理论(DLT)作业的服务质量(QoS)感知联合资源优化框架 Qore-DL，对提交、排队和运行 3 个阶段的作业进行分配与调整以提高作业完成率与资源分配率。文献[13]设计实现了一种扩展的调度模型，它基于启发式算法，决定如何将传入的计算请求分配给容器，容器放置在哪些边缘节点上，以及将已部署的容器迁移到哪里，解决边缘服务器中容器放置和迁移问题。研究人员设计了一种异构资源细粒度调度策略，利用 Kubernetes[14]设备插件机制收集每个节点上 GPU 的详细信息并提交给调度算法，在原有 CPU 和内存过滤算法的基础上改进调度器的打分算法，有效调度 GPU 并且避免资源竞争。文献[15]设计一种自定义调度器，在指定节点部署应用的资源不足时，将等待调度的应用调度到集群其他节点上，降低应用等待调度时间。文献[16]基于深度学习云平台应用场景提出一种负载饱和调度策略，支持对用户指定的外部拓展资源进行评分，减少了资源碎片，提高资源利用率。文献[17]基于 KCSS 调度算法优化设计了 F-KCSS 调度算法，针对云平台上 GUI 应用 Pod 的调度偏好，根据任务类型对待调度队列中的 Pod 进行重新排序，并在打分阶段直接调度相同任务，拥有更好的资源利用率和更快的调度速度。文献[18]提出了一种名为 K-TAHP 的负载均衡策略，该策略结合了基于逼近理想解排序法(TOPSIS)与层次分析法(AHP)，改善了 Kubernetes 集群的负载均衡性能，解决集群在长时间运行后可能出现的因负载不均衡导致性能下降的问题。文献[19]提出一种结合了装箱与传播调度原则，并基于 TOPSIS 算法的多指标调度策略，用于处理集群多指标调度问题。该策略在 Docker Swarm 平台中展现出了良好的应用效果。尽管上述研究在资源调度方面取得了显著进展，但它们在处理多维异构资源时的资源碎片化和用户作业等待时间方面仍有改进空间。

为了更高效地在大规模计算平台上利用 Kubernetes 管理异构资源，旨在达到尽量减少资源碎片、减少用户作业等待时间的目的，本文深入分析 Kubernetes 默认调度策略并进行调整改进，提出一种融合层次分析法的多指标装箱调度策略，该调度策略摒弃了传统的负载均衡理念，倾向于采用更精简的节点集来运行作业。通过引入资源权重机制，此策略有效降低高权重资源的碎片化程度，提高异构集群中的高权重资源利用率。此外，本文基于调度框架设计实现了自定义调度器，该调度器优化了资源指标的评分算法，在集群的应用中取得了良好的调度效果，进一步减少了用户作业的等待时间。

1 Kubernetes 简介

Kubernetes 是 Google 开源的一个容器编排引擎[6]，用于自动化部署、容器调度，以及管理并拓展容器化应用，是当今互联网云计算基础设施的重要组成部分。

1.1 Kubernetes 架构

Kubernetes 集群由若干个管理节点与多个工作节点构成，整体架构如图 1 所示。其中，管理节点由 Etcd、Kube-API Server、Kube-Scheduler、Kube-Controller-Manager 等组件组成，是集群的核心。Etcd 组件负责存储集群的配置与状态信息，Kube-API Server 组件是集群组件交互的 API 接口，Kube-Scheduler 与 Kube-Controller-Manager 组件分别负责整个集群的调度与管理。工作节点接受管理节点的调度管理，由 Kubelet、Kube-proxy 与

Container Runtime 组成，其中 Kubelet 是守护进程，与管理节点通信，Kube-proxy 负责集群内部与外部的网络通信[6]。Docker 是最常用的 Container Runtime，负责 Pod 中容器的运行[1]。

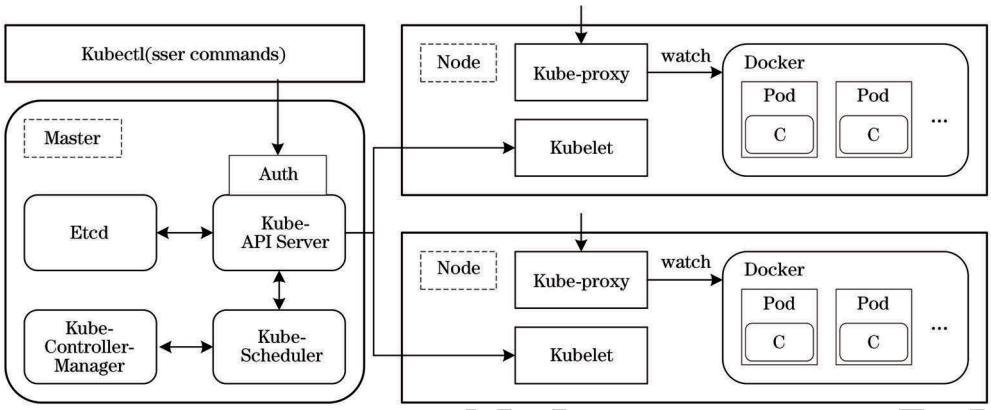

图 1 Kubernetes 架构

Fig. 1 Kubernetes architecture

1.2 Kubernetes 默认调度流程

Kube-Scheduler 作为集群的默认调度器，负责将 Pod 调度到集群中特定节点。Kube-Scheduler 的调度过程主要分为 3 个阶段，分别是预选、优选与绑定阶段。其中，预选阶段会根据 Pod 需求对所有工作节点遍历，判断节点资源是否充足，筛选出符合要求的节点。在优选阶段，调度器会根据调度策略对预选阶段筛选出的可调度节点进行评分，其中的核心评分策略包括：

1）LeastRequestedPriority（默认权重 1）。尽量将 Pod 调度到计算资源占用小的节点，打分算法如式（1）所示：

$$s_1=\frac{\dfrac{C_{\text{cpu}}-\sum R_{\text{cpu}}}{C_{\text{cpu}}}\times 10+\dfrac{C_{\text{mem}}-\sum R_{\text{mem}}}{C_{\text{mem}}}\times 10}{2} \tag{1}$$

式中：C_{cpu}、C_{mem} 分别为节点可申请的 CPU 与内存资源量；R_{cpu}、R_{mem} 分别为 Pods 请求的 CPU 资源与内存资源量。

2）BalancedResourceAllocation（默认权重 1）。该策略均衡了节点 CPU 和内存的配比，尽量选择在部署 Pod 后各项资源更均衡的机器，即 CPU 和内存使用率越接近的节点评分越高。打分算法如式（2）所示：

$$s_2=10-\left(\frac{P_{\text{PodsReq}_{\text{cpu}}}}{P_{\text{NodeCap}_{\text{cpu}}}}-\frac{P_{\text{PodsReq}_{\text{mem}}}}{P_{\text{NodeCap}_{\text{mem}}}}\right)\times 10 \tag{2}$$

式中：$P_{\text{PodsReq}_{\text{cpu}}}$ 与 $P_{\text{PodsReq}_{\text{mem}}}$ 分别为 Pods 申请的 CPU 资源与内存资源量；$P_{\text{NodeCap}_{\text{cpu}}}$ 与 $P_{\text{NodeCap}_{\text{mem}}}$ 为节点可申请 CPU 资源与内存量。

节点最终得分是所有评分的加权和，计算方法如式（3）所示：

$$s_{\text{FinalScore}}=\sum_{i=1}^{n} w_{\text{weight}_i}\times s_i \tag{3}$$

式中：w_{weight_i} 为评分函数权重；s_i 为不同评分函数的评分。

调度器根据评分排序并选出综合得分最高的节点，在绑定阶段将待调度 Pod 绑定到该节点，并更新节点与 Pod 状态，完成调度过程。

通过分析上述打分机制，可以看出 Kubernetes 默认调度算法的核心在于实现负载均衡策略以调度 Pod，该策略在考虑节点 CPU 与内存资源利用率均衡的同时，也趋向集群节点资源的负载均衡。Kubernetes 默认调度策略能适应大多数常规调度场景，但在大规模异构计算平台中，不仅需要考虑除 CPU 与内存之外的资源指标，同时也需要追求资源的高效利用，例如面对如图 2 所示的场景时，集群中有 8 个 CPU 且已占用 6 个 CPU，待调度 Pod 需求 2 个 CPU，使用 Kubernetes 默认调度策略则会面临虽然集群资源充足，但无法调度到任何节点的情形，造成资源浪费，延长作业等待时间。

1.3 Kubernetes 调度框架

Kubernetes 调度框架[20]是面向 Kubernetes 调度器的一种插件结构，它由一组直接编译到调度程序中的“插件”API 组成，这些 API 允许大多数调度功能以插件的形式实现，同时使调度“核心”保持简单且可维护。调度框架定义了一系列扩展点，如图 3 所示[20]。调度器插件注册后在一个或多个扩展点处被调用。这些插件中的一部分用于提供信息，而另一些可以改变调度决策。

用户可以根据实际需求以插件形式设计调度算法，并且将它们与默认插件一起配置以实现自定义调度器。

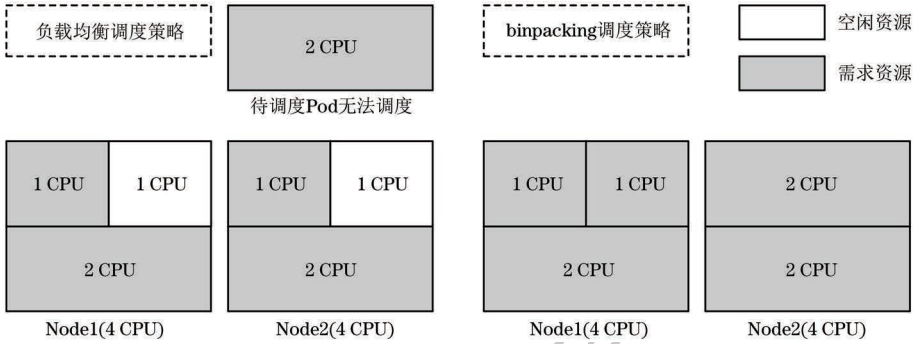

图 2 装箱调度与负载均衡调度

Fig. 2 Binpacking scheduling and load balancing scheduling

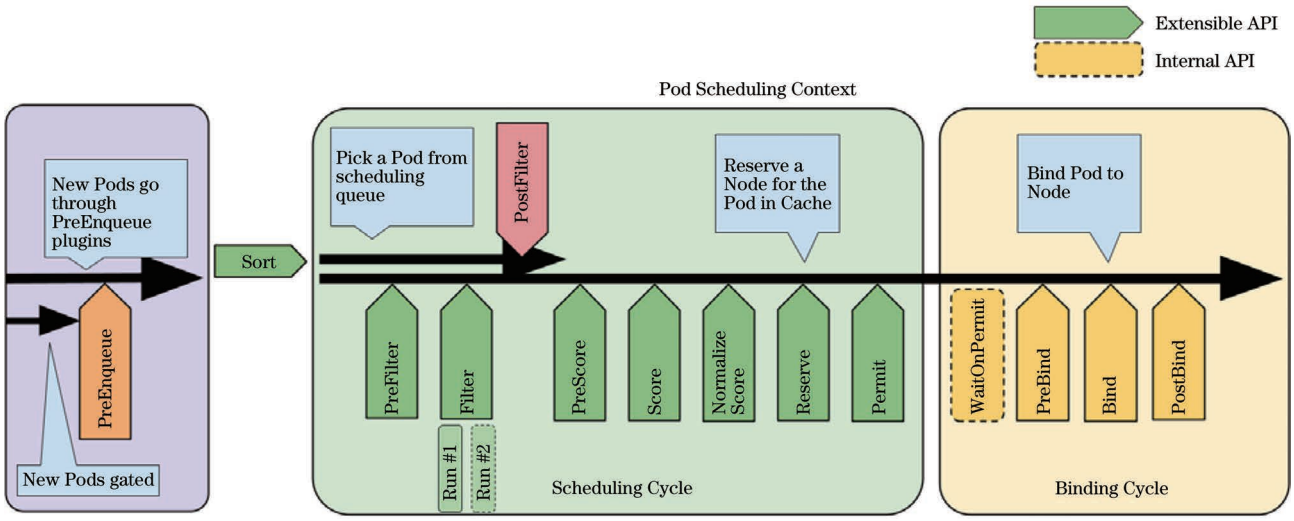

图 3 Kubernetes 调度框架

Fig. 3 Kubernetes scheduling framework

2 算法设计

本文提出一种融合层次分析法思想的多指标装箱调度策略，在考虑不同资源权重的同时能够最大限度地减少资源碎片化，提高集群中节点资源利用率，降低作业整体运行时间。

2.1 装箱调度

装箱调度[21]是一种注重资源利用率的调度策略，它倾向于将待调度作业分配到已经被占用的节点上。在新作业到达时，调度器会从集群中选择已经被占用的节点中资源利用率最高的节点来运行新的作业，而不是选择新的空闲节点。随着时间的推移，通过对作业的紧凑化配置，调度程序可以将更多作业打包到集群中。与在各节点之间平均分配作业的负载均衡调度策略相比，装箱调度能够最大限度地减少资源碎片化，提高资源利用率，如图 2 所示。

2.2 层次分析法

面对大规模异构计算平台中的多维资源调度的挑战，AHP[22]可以有效处理多个资源层级或多指标因素的复杂决策场景。该算法可以很好地将用户对不同资源的主观定性权重与集群资源的客观定量指标相结合，通过将调度过程分解为多个指标的若干层次，并利用定性指标模糊量化方式计算得出不同资源的权重占比，以此作为多指标调度策略的一种系统方法，其计算过程分为三步：建立对应的层次结构模型、判断矩阵的构造、层次单排序与一致性检验。

2.2.1 层次模型建立

层次分析法将决策问题分为目标层 Z，准则层 C，方案层 P 3 个层次。针对大规模异构计算平台下的 Kubernetes 集群的作业调度研究，本文将异构集群中节点的 CPU、内存、存储、容器镜像状态与类 GPU 加速卡数量作为资源指标，构建如图 4 所示的递阶层次结构。

2.2.2 判断矩阵的构造

在层次分析法中，为了能将矩阵中各要素的重要性(权重)定量显示，层次分析法引进矩阵的判断标度(1～9 标度法)，按照指标的重要性程度分为 9 个标度，如表1所示。通过将各指标相互比较确

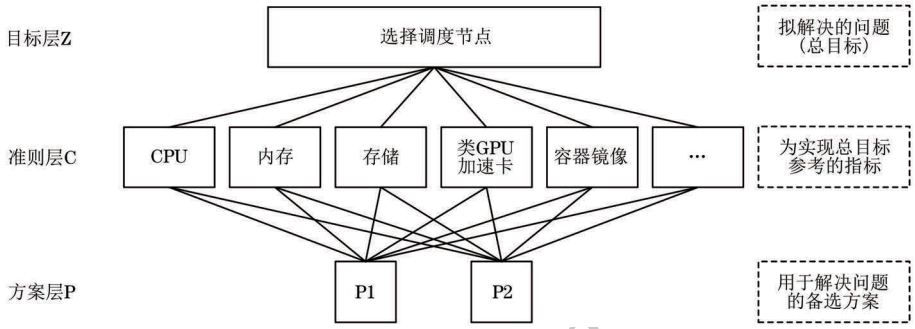

图 4 调度层次结构

Fig. 4 Structure of scheduling hierarchy

定其对于目标决策的权重，即构造判断矩阵 $\boldsymbol{A}=(a_{ij})$，其中 a_{ij} 表示元素 i 和元素 j 的重要性之比。

表 1 矩阵判断标度

Table 1 Matrix evaluation scale

标度	含义
1	表示两个元素相比，具有同样的重要性
3	表示两个元素相比，前者比后者稍重要
5	表示两个元素相比，前者比后者明显重要
7	表示两个元素相比，前者比后者极其重要
9	表示两个元素相比，前者比后者强烈重要
2,4,6,8	表示上述相邻判断的中间值
1～9 的倒数	表示相应两元素交换顺序比较的重要性

2.2.3 层次单排序与一致性检验

层次单排序是指针对上一层某因素与本层次中各因素进行两两对比，从而进行层次排序的过程。具体计算方式分为以下几步：

1)对于判断矩阵 $\boldsymbol{A}$，利用判断矩阵计算各指标对目标层(节点)的权重(权系数)，计算其权重向量 $\boldsymbol{W}$ 与最大特征值 $\lambda_{\max}$。其计算方式有几何平均法、算术平均法、迭代法等多种方式。层次分析法采用几何平均法对判断矩阵规模、计算条件与预算精度的需求进行计算，其计算过程分为 3 个步骤。

(1)通过对判断矩阵 $\boldsymbol{A}$ 的每行进行元素求积并进行 $1/n$ 幂运算，从而得到不同指标的相对权重，计算式如下：

$$\overline{\boldsymbol{W}}_i=\sqrt[n]{\prod_{j=1}^{n}a_{ij}}\ ,\ i,j=1,2,\cdots,n$$

式中：n 为判断矩阵的阶数；a_{ij} 则表示判断矩阵 $\boldsymbol{A}$ 的第 i 行、第 j 列元素。

(2)将上一步得到的相对权重值进行归一化处理，进而得到每个指标的综合权重。具体来说，将 $\boldsymbol{W}_i$ 归一化为排序权向量(使向量中各元素之和等于 1)，记为特征向量 $\boldsymbol{W}$，其中 $\boldsymbol{W}$ 的元素为同一层元素相对上层某一因素的相对权值排序，是判断层次单排序的结果。其中 $\boldsymbol{W}_i$ 的计算方式如下：

$$\boldsymbol{W}_i=\frac{\overline{\boldsymbol{W}}_i}{\sum_{i=1}^{n}\overline{\boldsymbol{W}}_i}$$

(3)求解判断矩阵 $\boldsymbol{A}$ 的最大特征根 $\lambda_{\max}$，$\lambda_{\max}$ 可以用于反映矩阵的一致程度。

$$\lambda_{\max}=\frac{1}{n}\sum_{i=1}^{n}\frac{(\boldsymbol{AW})_i}{\boldsymbol{W}_i}$$

2)在得到权系数后，根据得到最大特征根 $\lambda_{\max}$ 与对应的最大正规化特征向量 $\boldsymbol{W}$，本文可以得到 $\boldsymbol{A}$ 的特征根与特征向量，计算方式如式(4)所示：

$$\boldsymbol{AW}=\lambda_{\max}\boldsymbol{W} \tag{4}$$

3)当判断指标为 1 个或 2 个时，由于不存在足够的比较关系来产生逻辑错误，因此一致性是自动满足的，此时无需计算一致性比率(CR，R_{CR})值。若判断指标大于或等于 3 个，其对应的判断矩阵通常是不一致的。为了将其对应特征根的特征向量作为被比较因素的权向量，应将判断矩阵不一致程度控制在容许的范围内。一致性检验的目的是确定所构建的判断矩阵是否存在逻辑问题(例如权重 $a>b$，$b>c$，但 $c>a$ 的逻辑不一致情况)。为了检验判断矩阵的一致性，本文需要计算一致性指标(CI，I_{CI})和一致性比率以避免出现逻辑错误。一致性指标计算如式(5)所示：

$$I_{CI}=\frac{\lambda_{\max}-n}{n-1} \tag{5}$$

式中：$\lambda_{\max}$ 为判断矩阵 $\boldsymbol{A}$ 的最大特征根；n 为判断矩阵的阶数；当 CI 为 0 时 $\boldsymbol{A}$ 一致，CI 越大，$\boldsymbol{A}$ 的不一致性程度越严重。随机一致性指标(RI，I_{RI})取值如表 2 所示，表中 n 为判断矩阵的阶数。

4)对应的 R_{CR} 值计算如式(6)所示：

$$R_{CR}=\frac{I_{CI}}{I_{RI}} \tag{6}$$

当 $R_{CR}<0.1$ 时，表明判断矩阵 $\boldsymbol{A}$ 的一致性程度被认为在容许范围内，此时可用 $\boldsymbol{A}$ 的特征向量开展权向量计算；若 $R_{CR}\geqslant 0.1$，则判断矩阵可能出现了元素权重逻辑错误，应考虑对判断矩阵 $\boldsymbol{A}$ 进行修正。

表 2 随机一致性指标取值

Table 2 The values of the random consistency index

n	RI 取值
1	0
2	0
3	0.58
4	0.90
5	1.12
6	1.24
7	1.32
8	1.41
9	1.45

2.3 调度器设计

2.3.1 调度器扩展方式

考虑到实际环境中的各种复杂性，Kubernetes 调度器采用插件化的实现方式，这种方式允许用户对调度策略进行修改以实现用户自定义调度策略。Kubernetes 提供了 4 种方式扩展其默认调度器。

1)在 Kubernetes 调度器源码合适的位置进行修改，然后重新编译程序。此方法会对 Kubernetes 核心源码进行侵入式修改，易引发安全漏洞，且需要根据上游的调度程序不断调整更新保持一致，可移植性不佳。

2)修改调度器的配置文件。但此方式仅能对 Kubernetes 内置的调度策略进行开关与权重调整操作，无法实现用户自定义的调度策略。

3)实现调度器扩展程序。其核心实现是一个可配置的 Webhook，内部包含过滤器和优先级两个端点，分别对应调度周期中的过滤和打分两个主要阶段。此方式采用 http 请求的方式调用接口，无法支持高吞吐量业务。Kubernetes 在 v1.16 版本后已弃用此方式。

4)按照调度框架嵌入方式，Kubernetes 在传统调度逻辑的基础上进行了整合与再拆分，将基本流程分为调度队列循环、调度循环与绑定循环三部分(如图 3 所示)，并在各个部分实现对应传统的预选函数和优选函数，这极大增强了调度器的灵活性。

在综合考虑调度器的性能与兼容性的基础上，本文选择第 4 种方式扩展调度器，即基于调度框架嵌入方式实现自定义调度器，并对调度框架调度循环的多个不同拓展点进行更改和扩展。Kubernetes 扩展点如图 5 所示。具体来说，在过滤阶段，本文向调度器中整合了自定义的预选函数，旨在筛选出符合硬件需求的节点；而在打分阶段，不仅调整了现有打分函数的权重，还引入了全新的评分函数。该函数融合了装箱调度思想与 AHP，实现了对节点更为精确的评估。这一系列的改进涵盖了预过滤、过滤、预打分、打分，以及分数标准化等多个关键扩展点，从而确保调度器在保持高性能的同时，也具备高度的灵活性与适应性。

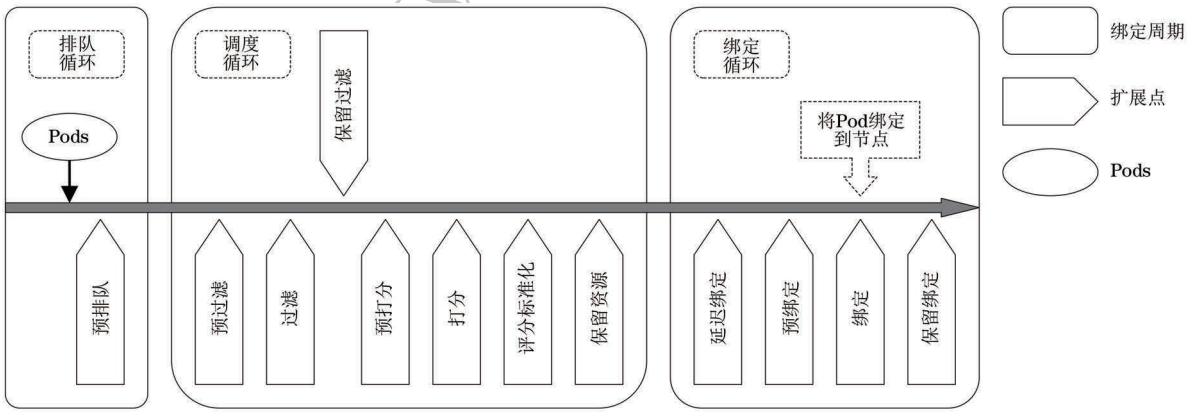

图 5 Kubernetes 扩展点

Fig.5 Kubernetes extension points

2.3.2 预过滤与过滤

预过滤(PreFilter)扩展点的程序用于预处理检查过滤之前 Pod 或集群的有关信息，是信息性质扩展点。调度器在此计算 Pod 的资源需求，如图 6 所示，包括普通容器与初始化容器，用于构建后续容器的环境，一般仅运行 1 次。

需求计算方式如式(7)所示：

$$I_{\mathrm{Pod_{req}}}=\max(I_{\mathrm{IC_1}},I_{\mathrm{IC_2}},\cdots,I_{\mathrm{IC_n}})+\sum_{i=1}^{n}C_n \quad (7)$$

式中：$I_{\mathrm{Pod_{req}}}$ 为 Pod 总资源需求；I_{IC} 为启动容器资源需求。

调度器会按配置顺序调用不同策略的过滤扩展点程序，用于过滤无法运行 Pod 的节点。节点通过所有的过滤程序，即代表资源满足Pod所有资源需

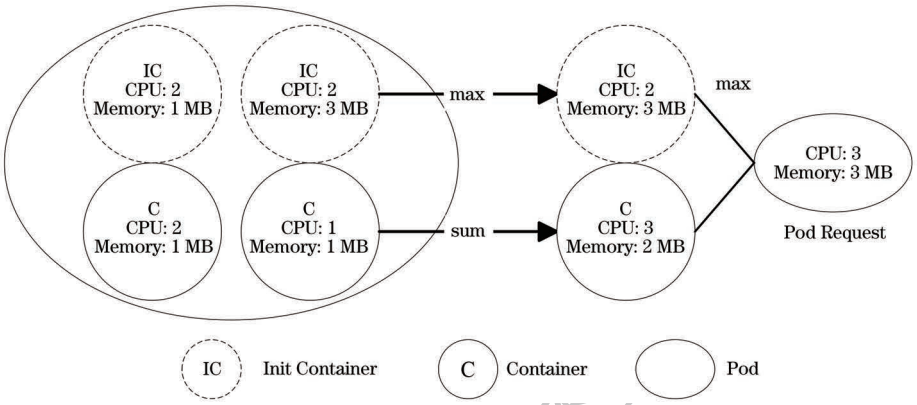

图 6 Pod 资源需求计算示意图

Fig. 6 Schematic diagram of Pod resource requirements calculation

求，则调度插件返回空表示节点通过过滤。如果没有节点通过所有的过滤拓展点程序，Pod 将被标记为不可调度并返回调度队列等待重新调度。

2.3.3 预打分，打分与分数标准化

预打分扩展点同样是信息性扩展点，用于预处理获取打分阶段会用到的一些数据信息。打分扩展点则用于调度策略的实现，该拓展点是调度器设计的核心。在这一拓展点中，依据表 1 中的判断矩阵标度，按照 CPU、内存、临时存储、类 GPU 加速卡、镜像状态的指标顺序，预设并构建了权重矩阵 $\boldsymbol{A}$，如式(8)所示。权重矩阵 $\boldsymbol{A}$ 的第 i 行、第 j 列元素代表第 i 个指标相对第 j 个指标的重要性比值。本文通过判断矩阵可以计算出各个资源指标的综合权重值 w_{weight_i}，如式(9)所示：

$$\boldsymbol{A}=\begin{bmatrix} 1 & 2 & 3 & 1/3 & 1/2 \\ 1/2 & 1 & 2 & 1/4 & 1/3 \\ 1/3 & 1/2 & 1 & 1/4 & 1/3 \\ 3 & 4 & 4 & 1 & 3 \\ 2 & 3 & 1 & 1/3 & 1 \end{bmatrix} \tag{8}$$

$$w_{\text{weight}_i}=\frac{\left(\prod_{j=1}^{n} a_{ij}\right)^{\frac{1}{n}}}{\sum_{i=1}^{n}\left(\prod_{j=1}^{n} a_{ij}\right)^{\frac{1}{n}}},i=1,2,\cdots,n \tag{9}$$

式中：a_{ij} 为判断矩阵 $\boldsymbol{A}$ 的第 i 行、第 j 列元素。标准化拓展点则是遵循 Kubernetes 的标准规范，对打分阶段中节点的资源分数进行归一化处理。

首先，拓展点的调度策略会基于 CPU、内存、临时存储、类 GPU 加速卡、镜像状态指标，通过计算已申请资源与可用资源的比值按顺序构建数据切片，其中镜像状态使用特定方式转化为分数量化描述。此方式根据容器所需镜像的扩散程度(即节点中有多少节点已存在此镜像)与镜像大小占该节点所有镜像大小的比例打分，即如果一个节点有很多非容器所需镜像，则该节点存在直接运行其他 Pod 的“潜力”，应当避免过度消耗该节点资源，更应该放到“潜力”较小的节点，也就是总镜像数更少的节点。计算镜像分数如式(10)所示，分数标准化如式(11)所示：

$$s=\sum_{i=1}^{n}\frac{N_{\text{Image}[i].\text{nodes}}}{N_{\text{TotalNodes}}}\times T_{\text{Image}[i].\text{size}} \tag{10}$$

$$s_{\text{final}}=s_{\max}\times\frac{s-T_{\text{Threshold}_{\min}}}{T_{\text{Threshold}_{\max}}-T_{\text{Threshold}_{\min}}} \tag{11}$$

式中：s 为镜像初始得分；$N_{\text{TotalNodes}}$ 是集群中节点的数量；$N_{\text{Image}[i].\text{nodes}}$ 是所需镜像的节点数量；$T_{\text{Image}[i].\text{size}}$ 是镜像的大小，单位 Byte；s_{final} 是最终分数；$s_{\max}$、$T_{\text{Threshold}_{\min}}$ 与 $T_{\text{Threshold}_{\max}}$ 为固定值。其他资源则使用其对应资源数量描述。

然后，根据用户对不同资源主观性判断构建判断矩阵，并通过层次分析法计算出不同资源在评价体系中的对应权重。将得到的各资源权重与资源切片中对应元素进行积运算，得到的节点分数如式(12)所示：

$$s_{\text{node}}=\sum_{i=1}^{n}F[i]\times w_{\text{weight}_i} \tag{12}$$

式中：s_{node} 是节点的资源得分；$F[i]$是资源切片；w_{weight_i} 为资源综合权重。

打分机制依据申请资源与可用资源的比值提高节点分数，该策略按照不同资源的权重，给予能恰好用完资源的节点更高分数，旨在实现资源最大化利用的装箱调度思想，即该调度策略会尽可能填满节点后再转向下一个节点进行调度。

2.4 调度器实现

2.4.1 调度器开销

本文调度策略的实现是基于对调度框架内不同扩展点的扩展与调整，摒弃了传统装箱调度算法中资源频繁重新分配的方式，转而通过改进

Kubernetes 调度框架中的打分函数来实现。与默认调度器相比，调度延迟的变化微乎其微。这主要得益于调度器在过滤阶段与调度打分阶段进行了优化，而这些阶段在整个调度过程中的计算开销相对较小。此外，由于未涉及复杂的资源重新分配，因此未产生额外的分配开销。同时，运行作业不会抢占已分配的调度器资源，因此在高负载与低负载下调度延迟的变化也极小。综上，在调度层面本文的调度器与默认调度器相比，几乎没有额外的调度开销。

2.4.2 调度器架构设计

在完成对特定拓展点的调度策略后，本文构建了对应自定义调度器——RFA 调度器，并完成调度插件的权重分配与权限配置。

参考矩阵判断标度表，按照 CPU、内存、临时存储、类 GPU 加速卡、镜像状态指标顺序构建判断矩阵，并确定构建调度器时调度资源的综合权重。

调度器的整体架构如图 7 所示。在 Kubernetes 集群中，调度器以 Pod 形式运行。Kubernetes 支持两种主要的运行模式：独占模式与多调度器共存(Multi-Scheduler)模式。在充分利用调度策略灵活性的同时保留 Kubernetes 默认调度器的核心功能，本文选择了多调度器共存的方式，用户可通过在 Pod 规范(Spec)中指定 SchedulerName 字段选择特定的调度器。若 Pod 未指明调度器，将默认使用集群内置的默认调度器进行调度，这样在保留 Kubernetes 原有调度机制，保证稳定性与可靠性的同时也实现了对调度策略的灵活扩展与定制，满足多样化的应用场景需求。

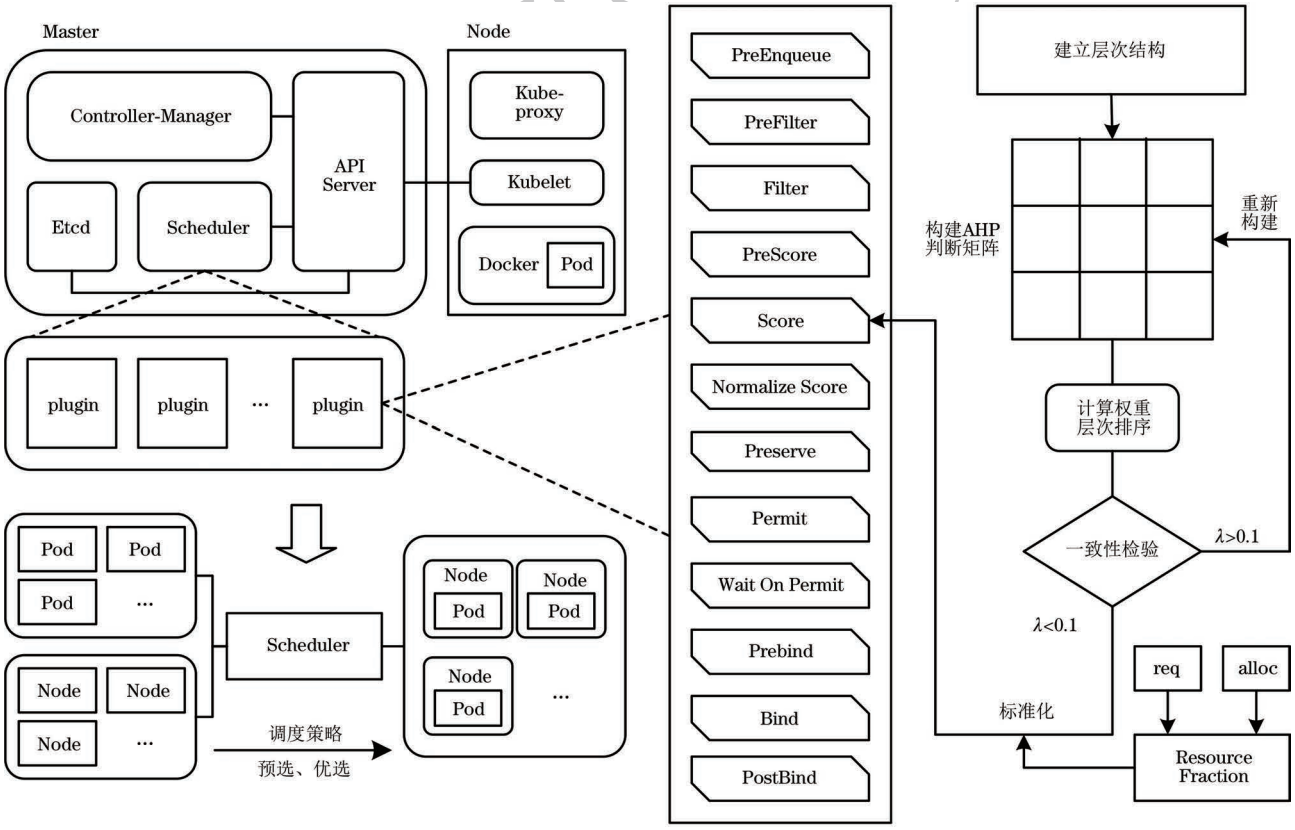

图 7 调度器架构

Fig.7 Scheduler architecture

3 实验分析

3.1 实验环境

本文在郑州大学大规模计算平台申请节点并搭建 Kubernetes 集群进行实验，评估调度策略。Kubernetes 版本采用 1.28.0 版本，Docker 26.1.4 版本，系统为 Centos 7.6，集群中有 1 个 Master 节点与 2 个 Node 节点。其中 Master 节点不参与调度，2 个 Node 节点的资源配置及实验环境如表 3 所示。

表 3 实验环境配置

Table 3 Experimental environment configuration

实验环境	具体配置
CPU	32 核
内存/GB	128
存储/GB	202
类 GPU 加速卡数/块	4

3.2 实验设计

基于集群资源上限，本文采用随机函数动态生

成不同资源的请求值，并应用于 Pod 部署文件以模拟复杂作业资源需求场景。其中，由于每个节点上均部署了 Kubernetes 的代理组件与监控组件，实际可用资源量会少于如表 4 所示的理论资源量。为进一步优化资源分配与性能测试，本文将 Pod 分为两大类：GPU 型 Pod 与 CPU 型 Pod。其中 GPU 型 Pod 中运行 dgemmtest 测试程序，该程序针对通用矩阵乘法(GEMM)进行性能测试，用于评估硬件平台类 GPU 加速器在处理大规模矩阵乘法的性能，确保 Pod 申请到的类 GPU 加速卡处于满载状态。在 CPU 型 Pod 中则运行计算集群系统浮点性能的基准程序。该程序采用高斯消元法求解大规模一元 n 次稠密线性代数方程组，使 Pod 分配到的 CPU 资源处于满载状态。

本文设置 Pod 最长运行时间为 2 min，对两种类型 Pod 进行部署测试。各 Pod 的资源需求信息如表 4 所示。

表 4 Pod 资源需求信息

Table 4 Pod resource requirements informatnion

Pod	CPU 数/个	内存/MB	存储/MB	类 GPU 加速卡数/块
Pod1	9	512	896	1
Pod2	10	512	1 024	2
Pod3	8	1 024	1 024	1
Pod4	16	1 024	896	1
Pod5	12	896	768	3
Pod6	9	1 024	896	0
Pod7	7	640	1 024	0
Pod8	14	512	896	0
Pod9	8	640	896	0
Pod10	15	1 024	640	0

3.3 实验结果与分析

在集群上对不同类型 Pod 进行部署测试与混合部署测试，首先在调度器配置文件中通过 SchedulerName 字段指定 RFA 调度器运行调度测试，完成后再指定默认调度并运行相同调度测试进行对比。在监控测试中作业运行状态与集群节点资源负载变化信息。为了全面评估调度性能，本文定义了以下实验指标。

1)作业调度时间。作业通常由一组 Pod 组成，为了更准确地衡量调度效率，该指标采用了该组 Pod 的平均等待时间，即计算在作业中所有 Pod 从提交至调度系统开始，到 Pod 被成功调度至某个节点并开始运行所经过时间的平均值。这一指标能够直观反映调度器调度效率。

2)作业运行时间。此指标计算的是从 Pod 提交至调度系统开始，直至 Pod 完成其任务并结束运行的总时长。它更直观地反映作业在集群中执行的整体效率。

将 Pod1～Pod5 组合作为类 GPU 型测试部署，将 Pod6～Pod10 组合作为 CPU 型测试部署，将 Pod1～Pod10 作为混合测试部署，收集集群运行状态信息，计算不同类型作业的 Pod 平均调度等待时间与节点运行作业时间，如图 8 所示。

从图 8(a)的作业调度等待时间可以看出，在采用 RFA 调度器时，针对类 GPU 型作业调度测试的 Pod，其平均等待调度时间减少了 25.2 s；对 CPU 型作业测试，Pod 等待调度时间缩减了 24.2 s；而在混合型测试中，Pod 的等待调度时间也减少了 24.6 s，时间降幅了 49.79%。这归因于 RFA 调度器采用了更加合理的调度策略，该策略能更有效地利用节点资源，从而避免了因资源分布在不同节点导致的调度等待时间，即集群资源充足，但作业因资源碎片化无法在任何 1 个节点上运行，需要调度等待其他作业释放资源。

从图 8(b)的节点运行作业时间可以看出，在 RFA 调度器的管理下，节点运行类 GPU 作业的时间减少了 120 s，运行 CPU 作业的时间减少了 123 s，并在运行混合型作业测试中运行时间减少了 120 s，这是由于节点运行作业时减少了调度等待时间。相较于默认调度器，RFA 调度器表现更优。

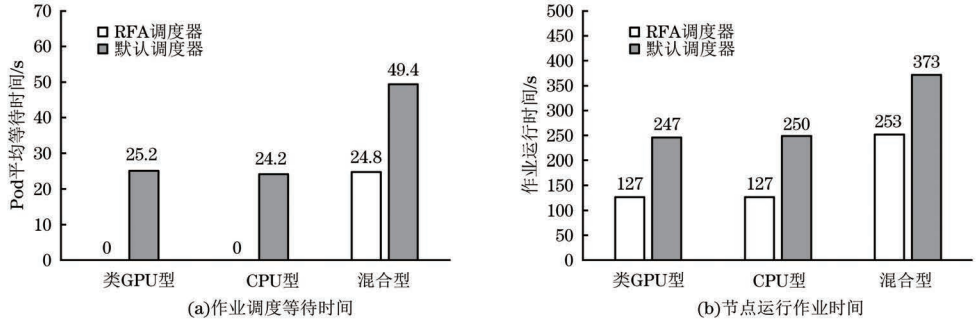

图 8 Pod 调度时间与运行时间

Fig.8 Pod scheduling time and runtime

在混合型作业的部署测试情景下，观察集群中节点的主要资源负载。各个节点不同资源的负载如图 9 所示(彩色效果见《计算机工程》官网 HTML 版，下同)，图中是每 5 s 采集 1 次数据得到的折线图。图 9(a)与图 9(b)展示了在 RFA 调度器调度下，不同节点运行作业时的部分资源负载状态。通过高权重资源——类 GPU 加速卡负载率可以看出，RFA 调度器优先调度并执行了高优先级作业(即类 GPU 加速卡依赖型 Pod)，随后再调度其他资源类型 Pod。相比之下，在图 9(c)与图 9(d)中，当节点切换至默认调度器后，由于资源调度策略不合理导致资源碎片化，Pod 无法充分有效利用节点上的类 GPU 加速卡资源。这种资源碎片化使得主要资源——类 GPU 加速卡未能保持满负载运行，相比同时期 RFA 调度器下的资源利用率较低。因此，高优先级 Pod 需要等待资源释放，只能分批运行，造成等待调度时间延长，进而延长节点运行作业的整体执行时间。

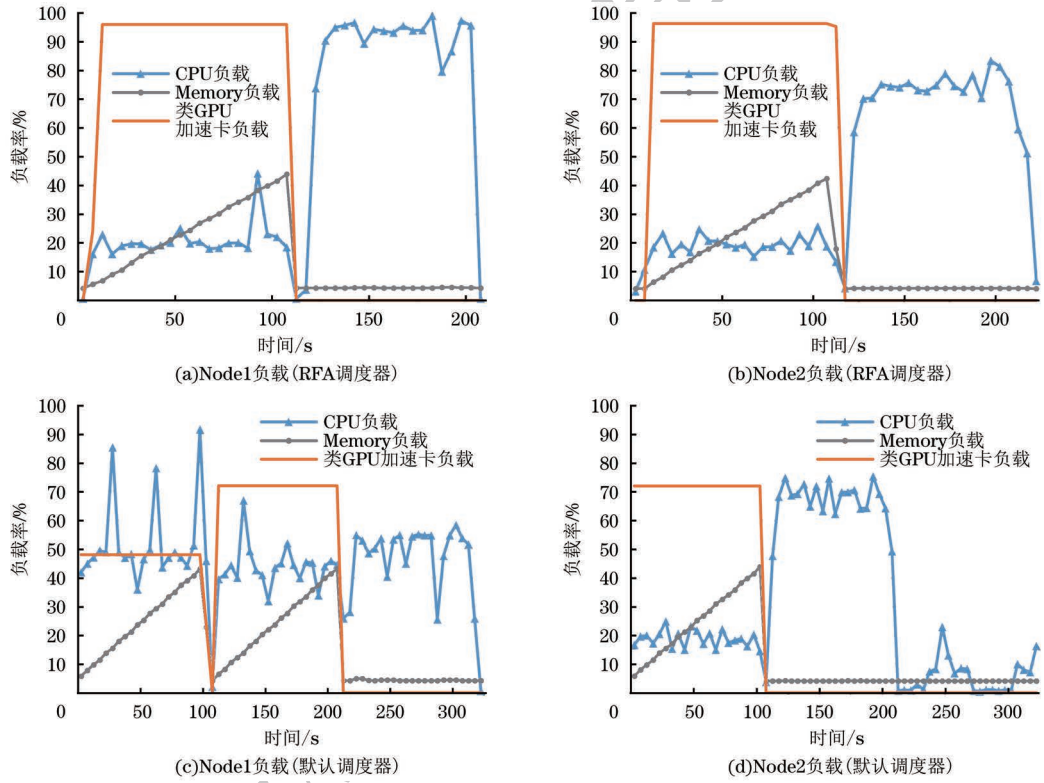

图 9 节点资源负载

Fig. 9 Nodes resource load

综上所述，从图 9 可以看出，在 RFA 调度阶段，节点对节点资源的利用率均要高于默认调度器，这充分证明了 RFA 调度器所使用的调度策略在优化资源分配和提高作业执行效率方面具有显著优势。

在混合部署测试场景中，节点运行作业时不同时刻运行 Pod 的数量如表 5 所示。从表 5 可以看出，在 RFA 调度器调度阶段，节点在大约 4 min 后将所有 Pod 任务执行完毕。而在默认调度器的调度下，因调度策略导致资源碎片化，节点在第 3 min 与第 4 min Pod 运行总数量低于同时期 RFA 调度器下 Pod 的运行总数量，导致默认调度器需要更长时间(0～6 min)运行 Pod 作业。在混合部署测试场景中 Pod 运行数量如图 10 所示，横坐标表示间隔 1 min 的时间点。实验结果表明，RFA 调度器相较默认调度器能更充分地利用集群资源。

表 5 混合部署测试下节点运行 Pod 数量

Table 5 Number of Pod running on nodes under mixed deployment testing

时间/min	RFA 调度器		默认调度器	
	Node1	Node2	Node1	Node2
1	2	3	2	3
2	2	3	2	3
3	2	3	2	2
4	2	3	2	2
5	0	0	0	1
6	0	0	0	1

使用 Kubernetes 集群环境的测试实验，从节点的负载数据图可以看出，使用自定义调度器进行调度时，节点的资源综合负载高于默认调度器调度阶段，即作业更加高效地利用了集群资源，节点在一定

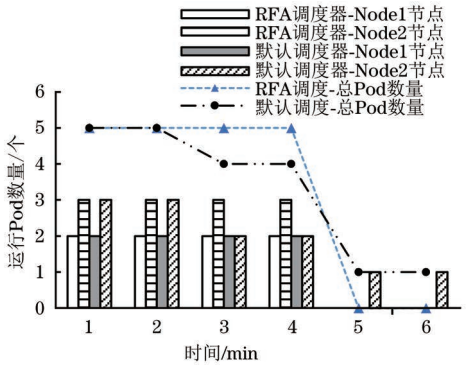

图 10 混合部署下 Pod 运行数量

Fig. 10 Number of running Pod under mixed deployment

时间内运行了更多 Pod，从而在减少 Pod 等待调度时间的同时提高了集群资源利用率。测试结果中不同类型作业调度测试中作业执行时间均少于默认调度器，这验证了本文调度策略的有效性。

4 结束语

在异构大规模计算平台混合部署调度的复杂环境中，面对多维资源需求下 Pod 调度过程中面临的资源碎片化、节点资源利用率低下等带来的挑战，本文提出一种优化的装箱调度算法，并基于调度框架完成了调度器的设计与实现。该策略通过在调度预选、优选核心阶段进行优化，综合考虑多维资源相对权重与各节点资源余量，实现作业到达时 Pod 到节点的最佳调度，从而尽量减少未来因资源碎片化产生的 Pod 调度等待情景。实验结果表明，改进的多指标调度策略相比较默认调度策略能够有效减少因资源碎片化造成的作业等待调度时间，提升集群资源利用率。后续将考虑集群规模，在现有研究的基础上，对调度器在其他调度环境下性能优化进行研究。

参考文献

[1] PUHAN S, PANDA D, MISHRA B K. Energy efficiency for cloud computing applications: a survey on the recent trends and future scopes[C]//Proceeding of International Conference on Computer Science, Engineering and Applications. Washington D. C., USA: IEEE Press, 2020: 1-6.

[2] HARDIKAR S, AHIRWAR P, RAJAN S. Containerization: cloud computing based inspiration technology for adoption through Docker and Kubernetes[C]// Proceedings of the 2nd International Conference on Electronics and Sustainable Communication Systems. Washington D. C., USA: IEEE Press, 2021: 1996-2003.

[3] MERKEL D. Docker: lightweight Linux containers for consistent development and deployment[J]. Linux Journal, 2014, 239(2): 1-2.

[4] 吴逸文，张洋，王涛，等. 从 Docker 容器看容器技术的发展：一种系统文献综述的视角[J]. 软件学报，2023，34(12)：5527-5551.
WU Y W, ZHANG Y, WANG T, et al. Development exploration of container technology through Docker containers: a systematic literature review perspective[J]. Journal of Software, 2023, 34(12): 5527-5551. (in Chinese)

[5] JARAMILLO D, NGUYEN D V, SMART R. Leveraging microservices architecture by using Docker technology[C]// Proceeding of Southeast Conference 2016. Washington D. C., USA: IEEE Press, 2016: 1-5.

[6] Google. Kubernetes [EB/OL]. [2024-09-01]. https://kubernetes.io/docs/concepts/overview/.

[7] SENJAB K, ABBAS S, AHMED N, et al. A survey of Kubernetes scheduling algorithms[J]. Journal of Cloud Computing, 2023, 12(1): 1-26.

[8] CARRIÓN C. Kubernetes scheduling: Taxonomy, ongoing issues and challenges[J]. ACM Computing Surveys, 2023, 55(7): 1-37.

[9] SUN Y, XIANG H G, YE Q K, et al. A review of Kubernetes scheduling and load balancing methods[C]// Proceedings of the 4th International Conference on Information Science, Parallel and Distributed Systems. Washington D. C., USA: IEEE Press, 2023: 284-290.

[10] 孙毅，王会梅，鲜明，等. Kubeflow 异构算力调度策略研究[J]. 计算机工程，2024，50(2)：25-32.
SUN Y, WANG H M, XIAN M, et al. Research on heterogeneous computing scheduling strategy for Kubeflow[J]. Computer Engineering, 2024, 50(2): 25-32. (in Chinese)

[11] RATTIHALLI G, GOVINDARAJU M, LU H, et al. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes[C]//Proceedings of the 12th International Conference on Cloud Computing. Washington D. C., USA: IEEE Press, 2019: 33-40.

[12] HUA Q, QIAN S Y, YANG D Y, et al. Qore-DL: a QoS-aware joint optimization framework for distributed deep learning training[J]. Journal of Systems Architecture, 2022, 130: 102640.

[13] OLEGHE O. Container placement and migration in edge computing: concept and scheduling models[J]. IEEE Access, 2021, 9: 68028-68043.

[14] 李俊俊，董建刚，李坤. 基于 Kubernetes 的集群节能策略研究[J]. 计算机工程，2024，50(9)：82-91.
LI J J, DONG J G, LI K. Research on Kubernetes-based cluster energy-saving strategy[J]. Computer Engineering, 2024, 50(9): 82-91. (in Chinese)

[15] 高荣，谢晓兰，刘亚荣，等. Kubernetes 资源调度策略[J]. 计算机工程与设计，2024，45(6)：1896-1902.
GAO R, XIE X L, LIU Y R, et al. Kubernetes resource scheduling strategy[J]. Computer Engineering and Design, 2024, 45(6): 1896-1902. (in Chinese)

[16] 王彬丞，王平辉，武文博，等. 面向深度学习 Kubernetes 负载饱和调度算法设计与实现[J]. 郑州大学学报(理学版)，2024，56(4)：21-27.
WANG B C, WANG P H, WU W B, et al. Design and implementation of Kubernetes load saturation scheduling algorithm for deep learning [J]. Journal of Zhengzhou University (Natural Science Edition), 2024, 56(4): 21-27. (in Chinese)

[17] 何丰. 基于 Kubernetes 的异构容器云平台的系统架构与方法优化[D]. 杭州电子科技大学，2022.
HE F. System architecture and optimization method of heterogeneous container platform based on Kubernetes[D].

Hangzhou: Hangzhou dianzi university, 2022. (in Chinese)
[18] GAO R, XIE X, GUO Q. K-TAHP: a kubernetes load balancing strategy base on TOPSIS + AHP [J]. IEEE Access, 2023, 11: 102132-102139.
[19] MENOUER T, DARMON P. New scheduling strategy based on multi-criteria decision algorithm [C]//Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. Washington D. C., USA: IEEE Press, 2019: 101-107.
[20] Google. Kubernetes [EB/OL]. [2024-09-01]. https://kubernetes.io/zh-cn/docs/concepts/schedulingevicti-on/scheduling-framework/.
[21] 郭袁贾. 基于二维装箱问题的 TTE 调度表生成算法[J]. 计算机工程与设计, 2021, 42(8): 2159-2166.
GUO Y J. TTE scheduling table generation algorithm based on two dimensional packing problem [J]. Computer Engineering and Design, 2021, 42(8): 2159-2166. (in Chinese)
[22] 刘道清, 扈红超, 霍树民. 容器云中面向持久化存储的拟态防御技术研究[J]. 计算机工程, 2024, 50(2): 165-179.
LIU D Q, HU H C, HUO S M. Research on persistent storage-oriented mimic defense technology in container clouds [J]. Computer Engineering, 2024, 50(2): 165-179. (in Chinese)

文字编辑 薛晋栋
栏目编辑 宋 圆