学校代码：10126　　学号：31809145

分 类 号：TP311　　编号：

论文题目

基于容器的云资源弹性策略的研究与实现

学　　院：计算机学院

专　　业：软件工程

研究方向：云网融合与大数据评测

姓　　名：陈锐锐

指导教师：李华 教授

2021 年 6 月 5 日

原 创 性 声 明

本人声明：所呈交的学位论文是本人在导师的指导下进行的研究工作及取得的研究成果。除本文已经注明引用的内容外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得内蒙古大学及其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：陈锐锐　　指导教师签名：

日　　期：2021.6.5　　日　　期：2021.6.5

在学期间研究成果使用承诺书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：内蒙古大学有权将学位论文的全部内容或部分保留并向国家有关机构、部门送交学位论文的复印件和磁盘，允许编入有关数据库进行检索，也可以采用影印、缩印或其他复制手段保存、汇编学位论文。为保护学院和导师的知识产权，作者在学期间取得的研究成果属于内蒙古大学。作者今后使用涉及在学期间主要研究内容或研究成果，须征得内蒙古大学就读期间导师的同意；若用于发表论文，版权单位必须署名为内蒙古大学方可投稿或公开发表。

学位论文作者签名：陈锐锐　　指导教师签名：

日　　期：2021.6.5　　日　　期：2021.6.5

基于容器的云资源弹性策略的研究与实现

摘 要

近年来，以 Docker 为代表的容器技术已日益成熟，其编排系统 Kubernetes 的功能日趋完善，但是 Kubernetes 现有的弹性扩缩容策略存在一些不足之处。本文分析了 Kubernetes 现有的弹性策略，并针对其存在的问题提出了优化策略。本文主要工作如下：

（1）针对 Kubernetes 现有的扩容策略对新增 Pod（Kubernetes 中最小的资源调度单元）进行调度时，未考虑多个 Pod 部署后集群负载均衡和资源消耗成本的问题，本文提出一种基于改进蚁群算法的资源调度策略，对现有的扩容策略进行优化。首先基于集群负载均衡度和资源消耗成本建立目标函数；其次改进了经典蚁群算法的信息素浓度初始值和算法终止条件等，并解决了其在资源调度中存在收敛速度慢和易陷入局部最优的问题；然后对现有扩容策略的节点选定过程进行优化；最后在 CloudSim 平台实现了优化后的弹性扩容策略，并通过实验与基本调度算法(Bind)的策略、基于经典蚁群算法(ACO)的策略和基于遗传算法(GA)的策略进行对比。

（2）针对 Kubernetes 现有的缩容策略在选择待删 Pod 时，只考虑 Pod 的状态优先级，而未考虑删除 Pod 后节点资源平衡和节点资源消耗成本的问题，本文提出了基于节点多维资源平衡和节点资源消耗成本的缩容策略，并将这两种策略同现有的缩容策略进行整合优化，使缩容时能综合考虑 Pod 的状态优先级

以及节点的资源平衡和资源消耗成本的因素，然后在 CloudSim 平台实现了优化后的弹性缩容策略，并通过实验与 Kubernetes 现有的缩容策略进行对比。

研究结果表明，本文优化的弹性扩容策略相比基本调度算法(Bind)的策略、基于经典蚁群算法(ACO)的策略和基于遗传算法(GA)的策略可以降低资源消耗成本，使集群负载更加均衡；本文优化的弹性缩容策略相比现有的缩容策略，能够降低节点资源消耗成本，减少节点资源碎片的产生，使节点资源使用更加平衡。

关键词：Kubernetes；弹性策略；资源消耗成本；集群负载均衡；资源平衡

RESEARCH AND IMPLEMENTATION OF CONAINER-BASED CLOUD RESOURCE ELASTICITY STRATEGY

ABSTRACT

In recent years, the container technology represented by Docker has become increasingly mature, and the function of its orchestration system Kubernetes has become more and more perfect. However, the existing elastic strategy of Kubernetes has some shortcomings. The existing elastic strategy of Kubernetes is analyzed, and the optimization strategies are proposed for the existing problems. The main work of this thesis is as follows:

(1) When scheduling new Pods (the smallest resource scheduling unit in Kubernetes) based on the existing expansion strategy of Kubernetes, the problem of cluster load balancing and resource consumption costs is not considered after multiple Pods deployment. An improvement based on the resource scheduling strategy of ant colony algorithm is proposed to optimize the existing expansion strategy. Firstly, the objective function based on cluster load balancing degree and resource consumption cost is established. Secondly, the initial value of pheromone concentration and the termination condition of the classical ant colony algorithm are improved, and the problems of slow convergence speed and easy to fall into local optimum in resource scheduling are solved. Thirdly, the node selection process of the existing expansion strategy is optimized. Finally, the optimized elastic expansion strategy is implemented on CloudSim platform, and it is evaluated by comparing its performance against the basic scheduling algorithm (Bind) strategy, the classical ant colony algorithm (ACO) based strategy and the genetic algorithm (GA) based strategy.

(2) In view of the existing shrinking strategy of Kubernetes, only the status priority of the Pod is considered when selecting the Pod to be deleted and the problem of node resource balance and node resource consumption costis not considered after the Pod is deleted. A shrinking strategy based on multi-dimensional resource balance and node resource consumption cost is proposed. The proposed two strategies are integrated and optimized with the existing one, so that the state priority of pod and the resource balance and resource consumption cost of the node can be considered comprehensively. Then, the optimized elastic shrinking strategy is implemented on the CloudSim platform, and compared it with the existing shrinkage strategy of Kubernetes through experiments.

The results show that compared with the strategy of Bind, ACO and GA, the optimized strategy can reduce the cost of resource consumption and make the cluster load more balanced. Compared with the existing shrinking strategy, the proposed optimized elastic shrinking strategy can reduce the cost of node resource consumption, reduce the generation of node resource fragments, and make the use of node resources more balanced.

KEYWORDS: Kubernetes; elastic strategy; resource consumption cost; cluster load balancing; resource balancing

目 录

图目录

表目录

第一章 引言

1.1 研究背景与意义

随着互联网的深入发展，云计算技术应用于各行各业。在云计算技术中，为了保证业务系统能够满足用户的多样性需求，云资源分配回收的弹性机制显得尤其重要。因此，对云资源弹性策略的研究非常必要。

近年来，以 Docker[1]为代表的容器技术迅猛发展，推动了云平台的革新，它通过镜像技术和镜像仓库有效解决了虚拟机存在的消耗资源过多、调度速度缓慢、软件堆栈环境不统一和资源利用率低等问题。但是，Docker 只关注容器和镜像[1]，难以对大量的容器化应用进行编排管理，于是就需要一个容器编排系统来协调管理集群中的容器。因此，一些优秀的容器集群编排系统开始涌现，比如以 Docker 为核心的第三方容器云平台 Kubernetes[2]、Docker Swarm[3]等。Kubernetes 是谷歌的大规模容器管理系统——Borg 衍生的开源项目，随着业界的广泛关注，已然成为容器编排领域的事实标准[2]，又因为弹性伸缩是 Kubernetes 的重要特性，故本文选择对 Kubernetes 现有的弹性扩缩容策略进行研究与优化。

目前，Kubernetes 通过 HPA（Horizontal Pod Autoscaling）功能实现 Pod 数量的自动扩容或者缩容。在扩容阶段，Kubernetes 通过现有扩容策略的优选评分过程将新增 Pod 调度到评分最高的资源节点上运行；在缩容阶段，Kubernetes 根据 Pod 的状态优先级选择待删 Pod 进行缩容。但现有的弹性扩缩容策略未考虑 Pod 部署或删除后的集群负载均衡和节点资源平衡的问题，无法保证容器集群的高效稳定运行和服务质量。同时，现有的弹性扩缩容策略也未考虑 Pod 长时间运行占用资源导致资源消耗成本过高的问题。因此，针对 Kubernetes 现有的弹性扩缩容策略进行优化，是保证容器集群稳定运行和资源消耗总成本较小的有效途径。

1.2 国内外研究现状

本文主要从 Kubernetes 调度器的优化和自定义算法两个方面介绍目前国内外基于容器的云资源弹性策略的研究。

在调度器的优化方面，文献[4]提出了基于 Kubernetes 的伸缩性分布式调度器，能应对复杂生产环境中的大型集群的需求，解决了现有调度器不稳定的问题。文献[5]设计了一种混合

状态调度框架模型，该模型将大多数任务委托给分布式调度代理，并具有调度校正功能，解决了 Kubernetes 集群中负载分布不均衡以及处理资源利用率方面波动的问题。文献[6]通过将 Kubernetes 与企业网格计算（Enterprise Grid Orchestrator，简称 EGO）调度系统结合，解决了 Kubernetes 在资源调度方面功能单一、效率低下以及调度算法设计和调度器架构过于简单的问题。文献[7]开发了一个具有全面监控机制、部署灵活和自动操作特点的资源调度平台，该平台对 Kubernetes 集群资源进行监控，考虑了系统资源利用率和应用程序服务质量指标，模块化了动态资源调度的步骤，从而提高了资源调度的效率。这种方式均是在对 Kubernetes 现有的调度器辅以系统支持，优化调度器的效率和性能，从而保证容器集群的负载均衡。

在自定义算法方面，文献[8-9]分别从扩容和缩容两方面进行研究，左等人提出一种动态资源调度算法，根据 Pod 对节点的 CPU 和内存资源的使用情况，将其部署于 CPU 和内存资源利用率接近的节点上，使集群负载更均衡[8]。张等人提出一种动态缩容算法，在选择待删 Pod 时，首先计算删除该 Pod 后的节点 CPU 和内存资源平衡度，然后优先删除平衡度较小的节点上的 Pod，从而提高节点的资源平衡度[9]。文献[10]提出了一种基于 Kubernetes 中应用程序特征的资源调度算法，根据不同类型应用程序对资源的不同使用程度进行差异化的调度，从而使集群资源得到充分利用。常等人在 Kubernetes 现有的资源调度算法研究基础上，提出关注集群最终的资源利用率，而不关注预选过程的自定义调度算法，以提升集群的整体负载均衡[11]。文献[12]提出了一种基于历史记录的 Kubernetes 资源调度算法，根据资源的实际使用量来对资源进行分配，从而提升节点资源的均衡度。文献[13]提出了基于请求频次和响应时间的容器自动扩缩容策略，使用请求频次与响应时间定量描述应用特征，在满足服务 SLA（响应时间）的同时，在扩容过程中最大化地利用计算资源。

除此之外，遗传、蚁群、蜂群、粒子群等启发式算法非常适合解决资源调度的问题。文献[14]中田等人结合不同云环境特征对调度算法研究进展进行讨论，梳理已有的启发式调度算法的调度机制及其优缺点，部分算法如表 1.1 所示。

表 1.1 启发式调度算法总结[14]

Table 1.1 Summary of Heuristic Scheduling Algorithms

算法名称	调度机制	优点	缺点
粒子群算法 PSO	模拟鸟群飞行行为，粒子在解空间追随最优的粒子进行搜索	参数少，易实现，收敛速度快	标准 PSO 易于陷入局部最优

遗传算法 GA	模拟生物进化过程，以选择、交叉和变异的方式不断迭代获得最优解	全局优化，快速搜索，过程简单，具有随机性	实现复杂，依赖参数，易早熟
蚁群算法 ACO	模拟蚂蚁觅食行为，采用正反馈机制，使得搜索过程不断收敛，最终逼近最优解	鲁棒性强，收敛结果较好，易于并行实现	收敛速度慢，易陷入局部最优

文献[15-17]使用多目标优化的遗传算法，建立资源平衡、集群负载等目标函数，解决了容器分配和资源调度过程中存在的集群负载不均衡和集群资源浪费的问题，从而使资源分配更加均衡。文献[18-20]均提出基于改进粒子群算法的资源调度方法，通过引入迭代选择抑制算子或动态惯性权重对算法进行改进，提高算法收敛速度，避免陷入局部最优，从而缩短了调度时间，提高了资源分配效率。李等人针对 Kubernetes 现有资源调度存在的资源利用率低的问题，提出基于改进蜂群算法的资源优化策略，构建资源功耗的目标函数，从而在资源调度中减少资源浪费，提高集群资源的利用率[21]。文献[22]提出一种基于猫群优化算法的资源调度优化方法，综合考虑最短时间与最优负载来构建猫群优化算法的适应度函数，提高了云计算资源利用率，保证了资源分配的均衡性。Kaewkasi 等人使用改进蚁群算法对 Docker Swarm 现有调度策略进行优化，加强集群资源平衡性，提升应用的性能[23]。文献[24-25]将蚁群算法和粒子群算法结合，针对 Kubernetes 为 Pod 选择节点时忽略集群负载的问题进行优化，从而降低集群的最大负载，使资源分配更加均衡。文献[26-28]均提出基于改进蚁群算法的优化模型，解决现有调度策略效率低下和集群负载不均衡的问题，加快最优解的收敛速度，从而提高集群服务的可靠性。在自定义算法方面，启发式算法大量应用于资源调度，分析各类算法的优缺点，由于蚁群算法具有收敛效果较好，鲁棒性强等特点，故本文选用蚁群算法作为资源调度的算法。

综上所述，在 Kubernetes 调度器的优化和自定义算法两个方面，基于容器的云资源弹性策略的研究对节点资源平衡和集群负载均衡方面做了很多研究工作，但是未考虑到 Pod 部署或删除后节点的资源消耗成本的问题。

1.3 主要研究内容

本文主要研究内容如下：

（1）介绍了 Docker 容器技术，Kubernetes 的集群架构以及现有的弹性扩缩容策略。

（2）对 Kubernetes 现有的扩容策略进行研究，针对现有扩容策略为新增 Pod 选择节点时只考虑当前性能最优的节点，即优选评分过程中得分最高的节点，而未考虑多个 Pod 部署后集群负载均衡和资源消耗成本的问题，提出了一种基于改进蚁群算法的资源调度策略，以集群负载均衡度和资源消耗成本为目标函数寻解，对现有扩容策略的节点选定过程进行优化，从而降低资源消耗成本，并使集群负载更加均衡。

（3）对 Kubernetes 现有的缩容策略进行研究，针对现有缩容策略在选择待删 Pod 时只考虑 Pod 的状态优先级，而未考虑删除 Pod 后节点资源平衡和节点资源消耗成本的问题，提出了基于节点多维资源平衡的缩容策略和基于节点资源消耗成本的缩容策略，并将这两种策略同现有缩容策略进行整合优化，使 Kubernetes 在缩容时能综合考虑 Pod 的状态优先级以及节点的资源平衡和资源消耗成本的因素。

1.4 论文组织结构

本文的组织结构如下：

第一章 引言。首先介绍了本文研究的背景和意义，然后分析国内外基于容器的云资源弹性策略的研究现状，最后说明本文的研究内容以及组织结构。

第二章 容器及相关技术概述。首先介绍 Docker 容器技术和 Kubernetes 的基本概念、系统架构以及现有的弹性策略，然后介绍了蚁群算法的原理、优缺点和基本流程，为后续章节的研究设计做铺垫。

第三章 弹性扩容策略的优化设计。针对 Kubernetes 现有的扩容策略未考虑节点资源消耗成本以及集群负载均衡的问题，提出了一种基于改进蚁群算法的资源调度策略，对现有扩容策略进行优化。

第四章 弹性缩容策略的优化设计。针对 Kubernetes 现有的缩容策略只考虑 Pod 的状态优先级，而未考虑节点资源平衡和资源消耗成本的问题，提出了基于节点资源平衡和节点资源消耗成本的两种缩容策略，并将这两种策略同现有的缩容策略进行整合优化。

第五章 实验与分析。将本文优化的扩容与缩容策略通过云计算仿真平台 CloudSim 实现，并设计对比实验，验证本文策略的可行性和有效性。

第六章 总结和展望。总结全文工作并给出下一步工作。

第二章 容器及相关技术概述

本章首先介绍了容器技术 Docker 核心概念和基本架构，然后阐述了容器集群编排系统 Kubernetes 的基本概念、系统架构和现有弹性策略，最后对蚁群算法的原理、优缺点和基本流程进行介绍。

2.1 容器技术简介

2.1.1 容器技术

容器[29]是一种沙盒技术，主要是为了将服务运行在其中，与外界隔离，也是为了方便它可以被转移到其它宿主机。通俗来说，容器就是箱子，箱子里面有软件运行所需的环境。开发人员可以把这个箱子移至任何机器，且不影响里面软件的运行，因此容器技术能够大大提升工作效率。

虚拟化技术[30]可以在有限的环境，给用户提供足够的资源，这样可以为用户带来灵活性和扩展性。但是，hypervisor[31]虚拟化技术仍存在一些不足之处，例如性能较差、易造成资源浪费等。为了解决 hypervisor 虚拟化技术存在的问题，容器（Container）虚拟化技术应运而生。

容器的出现可以解决多操作系统堆栈问题：

（1）容器技术编排管理（Orchestration）可以实现在一台服务器上的所有实例使用相同的操作系统[32]。

（2）许多容器技术的系统应用程序堆栈都是相同的。

但是，对于大型容器集群，将系统的副本存储在本地可能会增加升级过程的难度。

2.1.2 Docker 容器

Docker 是基于 Go 语言开发的开源的容器技术，源码托管于 GitHub 上，也符合 Apache2.0 协议。自从 2013 开始，Docker 由于便捷和轻量级的特点被人们熟知，逐渐变得流行，推动了云计算的革新。而且 Docker 从 GitHub 到 RedHat 在 RHEL6.5 中都非常活跃，甚至 Docker 支持谷歌的计算引擎的运行[33]。

Docker 的架构模型是 C/S（客户端服务器），并且 Docker 容器由 Docker 镜像来创建，容器和镜像的关系就像面向对象语言中的对象和类的关系。

Docker daemon[34]是运行在主机后台的守护进程，作为服务端接受并处理客户的创建、运行和分发容器的请求。而 Docker Client 可以通过可执行的命令与 Docker daemon 进行通信。Docker Register 用来保存镜像仓库。当 Docker Client 发送运行容器的命令，Docker daemon 接收该命令并从用户自己的镜像仓库拉取 Docker 镜像，实现容器的运行。Docker 架构如图 2.1 所示。

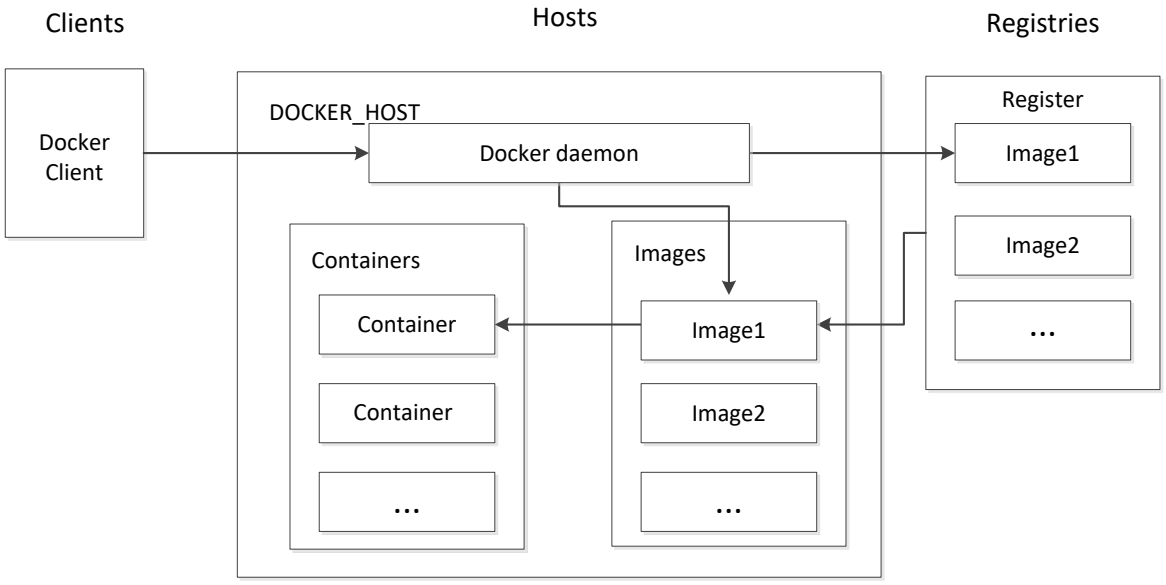


图 2.1 Docker 架构[34]

Figure 2.1 Docker Architecture

Docker 容器在开发和运维方面具有以下优势：

（1）快速的交付和部署。工作人员可以通过直接从镜像仓库拉取软件镜像进行部署，从而简化软件开发到测试发布整个过程。

（2）环境标准化与稳定性。使用 Git 之类的工具复制 Docker 镜像可确保 Docker 提供的环境的标准化和稳定性。

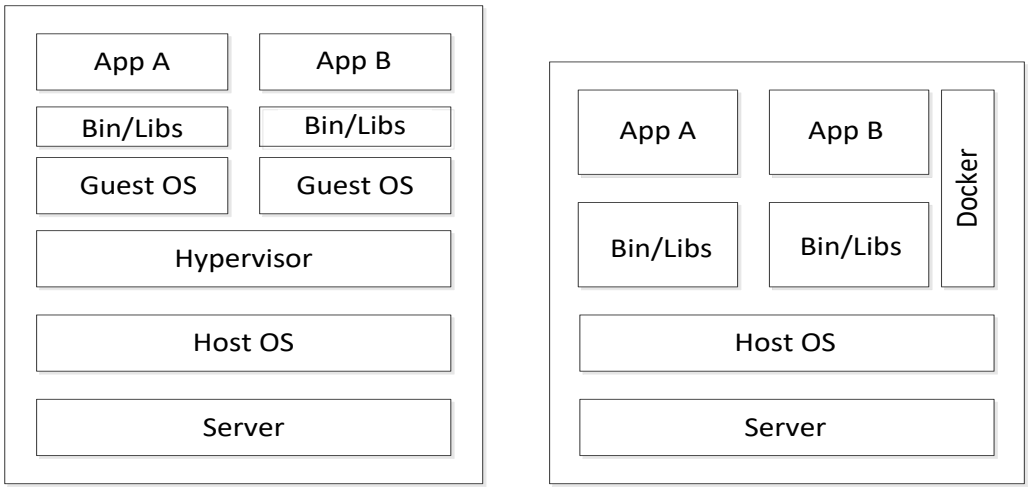
（3）合理的资源利用。不同于虚拟机，Docker 之间可以共享资源，更合理地利用资源。

（4）简单的升级管理。随着操作配置的升级更新，Dockerfile[35]可以自动执行高效的容器管理。

（5）易于迁移和扩展。Docker 可以在虚拟机或物理机等许多不同平台上工作，便于 Docker 的迁移。

2.1.3 Docker 与传统虚拟化的区别

传统虚拟化技术和 Docker 技术的体系架构[34]如图 2.2 所示。



a）传统虚拟化技术架构　　b）Docker 技术架构

a) Traditional Virtualization Technology Architecture　　b) Docker Technical Architecture

图 2.2 传统虚拟化技术与 Docker 技术体系架构[34]

Figure 2.2 Architecture of Traditional Virtualization Technology and Docker Technology

由图 2.2 可知，Docker 技术没有 GuestOS 层和 Hypervisor 层，换句话说，Docker 与宿主机共用操作系统内核，而每个虚拟机都有自己的操作系统。并且 Docker 之间可以相互分享资源，而虚拟机则相反，每个虚拟机都独享自己所占有的资源。因此，Docker 消耗更少的资源，更加轻量级，而虚拟机则会消耗更多的资源，易造成资源的浪费，更加重量级。

Docker 与传统虚拟机在各种特性方面的对比如表 2.1 所示。

表 2.1 Docker 容器技术与传统虚拟机技术的特性对比[34]

Table 2.1 Comparison of The Characteristics of Docker Container Technology and Traditional Virtual Machine Technology

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般 MB	一般 GB
性能	接近原生	较弱
系统支持	单机支持上千容器	几十个

2.2 Kubernetes

Kubernetes[36]是基于 Go 语言开发的容器编排系统，具有以下特点[37]：

（1）可移植性：Kubernetes 可以在包括物理机和虚拟机在内的多种环境中运行。

（2）可扩展性：Kubernetes 是模块化和插件化的平台，方便对集群中应用服务的扩展。

（3）自动化程度高：Kubernetes 中服务的扩容、缩容以及滚动更新等生命周期是自动化管理的。

2.2.1 Kubernetes 的基本概念

Kubernetes 中的 Pod、Label、RC 等资源对象通过 kubectl 工具运行或者存储在 etcd 中，而 Kubernetes 的组件可以操作这些资源对象来完成资源调度。

（1）Pod 是 Kubernetes 中最小的资源调度单元，包含一个或多个容器。Pod 中的每个容器都在特定端口上运行，它们之间共享命名空间（namespace）。

（2）Label 是键值对标签，可以贴附在资源对象上，以此来表示资源对象的属性。

（3）RC（Replication Controller）负责创建和管理 Pod 的副本，若是 Pod 副本的数量过少，则 RC 会增加新的副本，反之，若 Pod 副本过多，则 RC 会删除冗余副本。

2.2.2 Kubernetes 的系统架构

Kubernetes 由 Master 节点、Node 节点和客户端命令行工具 kubectl 组成，属于主从分布式架构[38]，系统架构如图 2.3 所示。

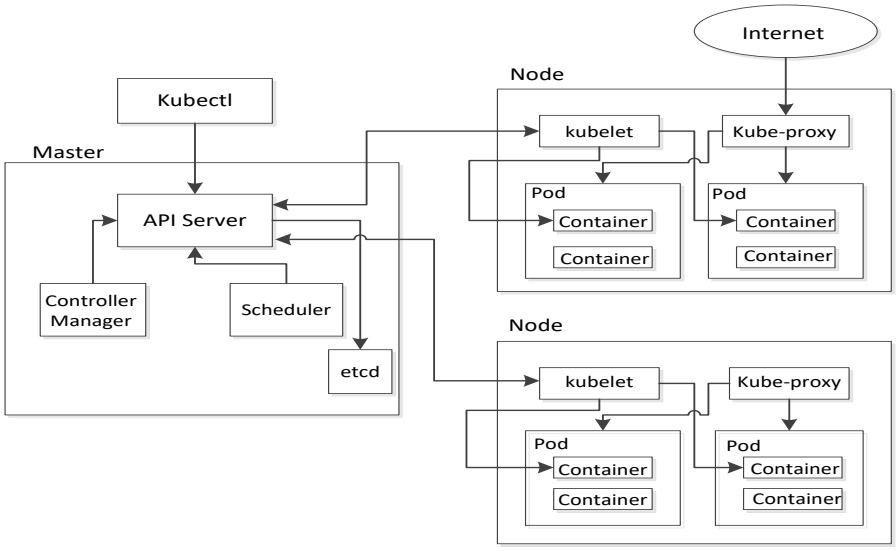


图 2.3 Kubernetes 系统架构[38]

Figure 2.3 Architecture of Kubernetes System

Master 节点是控制节点，主要功能是调度集群的资源，与集群中的各个 Node 节点进行通信并为其分发任务，并且 Master 节点上执行 Kubernetes 的操作命令。Master 节点主要包含以下四个组件：

（1）API Server 是 Kubernetes 系统的外部接口，客户端和其它组件可以调用它提供的 REST API，支持水平扩展，主要负责对资源对象的增删改查。

（2）Controller Manager 的主要功能是管理 Kubernetes 集群中的各种资源对象以及命名空间。

（3）etcd 是一个键值存储系统，具有高度可用性，可以持久储存集群的配置，以此来存储集群的状态。

（4）Scheduler 是 Kubernetes 的默认调度器，主要作用是通过现有的调度策略将待调度的 Pod 调度到合适的资源节点上。CM 会将 Pod 对资源的需求写入 etcd，当 Scheduler 接收到调度信息，就会对 Pod 进行调度操作。

Node 节点主要包含以下两个组件：

（1）kubelet 是 Node 节点启动的服务进程，负责处理 Master 节点分发到 Node 节点上的任务，对容器组的生命周期进行管理。

（2）kube-proxy 是 Node 的核心组件，主要是将服务访问请求转发到后端的多个 Pod 实例上。

2.2.3 Kubernetes 现有弹性策略

目前，Kubernetes 通过 HPA 功能实现对 Pod 数量的自动扩容或者缩容。在扩容阶段，通过 Kubernetes 现有的扩容策略将新增的 Pod 调度到集群中性能最好即节点优选过程中分数值最高的节点上[39]；在缩容阶段，Kubernetes 现有的缩容策略根据 Pod 的状态优先级选择待删 Pod。

（1）Kubernetes 现有的扩容策略主要分为预选过程、优选过程和选定过程。

现有的扩容策略为 Pod 选择节点的示例过程如图 2.4 所示。

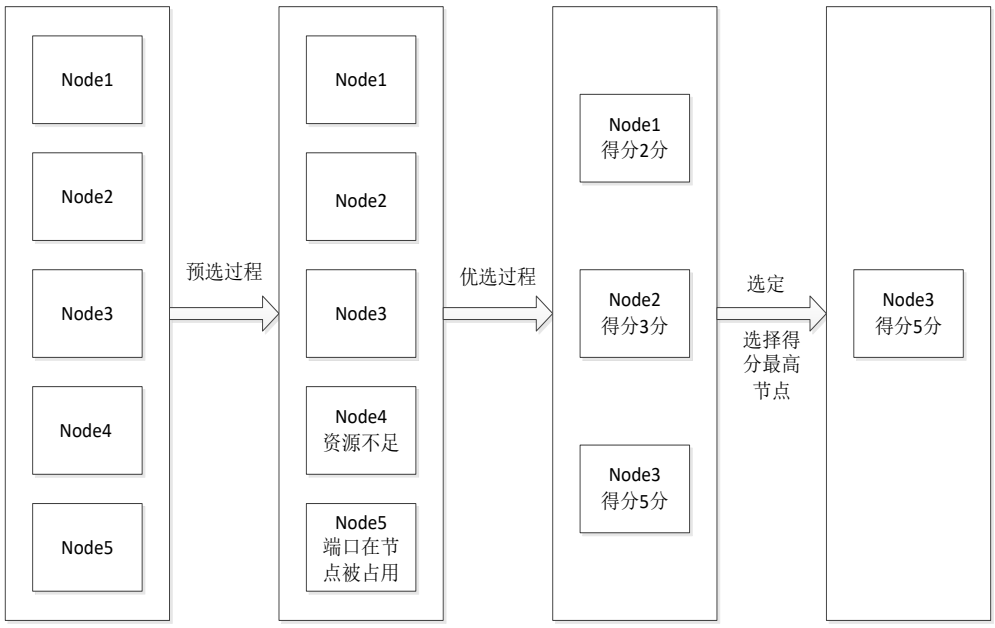


图 2.4 Kubernetes 现有的扩容策略选择节点的过程示例

Figure 2.4 An Example of The Process of Selecting Nodes for Kubernetes' Existing Expansion Strategy

a. 预选过程：根据 Kubernetes 集群中所有 Node 资源节点的情况而定。遍历这些资源节点，通过预选策略对节点进行过滤，排除资源不足的节点、容器运行时端口冲突的节点等，则剩余节点为符合条件的候选资源节点。

b. 优选过程：对候选资源节点进行评分，然后进行排序。默认的优选评分策略有 LeastRequestedPriority、BalanceResourceAllocation 等。默认的各个调度的策略权重为 1，因此，节点得分为各个评分策略得分的和，然后按照得分进行排序。

c. 选定过程：通过优选过程的评分策略得到各个候选资源节点的分值，将容器运行在得分最高的节点上。

（2）Kubernetes 缩容策略在选择待删 Pod 时，首先根据状态优先级对 Pod 进行排序，最后依次选择状态优先级小的 Pod 作为待删 Pod，具体步骤如下：

a. 首先根据 Pod 是否已就绪分为未就绪 Pod 和已就绪 Pod，未就绪 Pod 的优先级小于已就绪 Pod 的优先级。

b. 根据未就绪 Pod 是否已被调度分为未调度 Pod 和已调度 Pod，未调度 Pod 的优先级小于已调度 Pod 的优先级。

c. 根据已调度的 Pod 所处的状态分为 Pending、Unknown、Running，优先级依次增大。

d. 对于已就绪的 Pod，按照就绪时间的长短确定优先级，时间较短 Pod 的优先级较小。

e. 对于上面原则下优先级相等的 Pod 根据其内部容器的重启次数确定优先级，重启次数较多的 Pod 优先级较小。

f. 对于上面原则下优先级相等的 Pod 根据其创建时间的早晚确定优先级，时间晚的 Pod 优先级较小。

以上为 Kubernetes 现有的弹性策略。现有的扩容策略未考虑多个 Pod 部署后集群负载均衡和资源消耗成本的问题，现有的缩容策略未考虑删除 Pod 后节点资源平衡和节点资源消耗成本的问题，因此需要进行深入研究。

2.3 蚁群算法简介

在自定义算法方面，启发式算法大量应用于资源调度，由于蚁群算法具有收敛效果较好，鲁棒性强等特点，故本文选用蚁群算法作为资源调度的算法。本节对蚁群算法的原理、优缺点和基本流程进行介绍。

2.3.1 蚁群算法的原理

蚁群算法（Ant Clony Optimization，简称 ACO）是由 Dorigo[40]等人提出的一种启发式算法，该算法的启发性来源是真实蚂蚁的觅食过程，使用信息素作为通信媒介。初始蚂蚁在寻找食物时，会沿路留下信息素物质，而其后的蚂蚁就会根据之前蚂蚁留下的信息素，向着信息素浓度较高的方向爬行，最终找到食物，故大量蚂蚁觅食就表现出信息素的正反馈[41]。

蚁群算法原理如下：

（1）初始蚂蚁面对多条路径，任选其中一条爬行，并沿路撒下信息素；

（2）随后的蚂蚁遇到同一个路口时，会根据之前蚂蚁留下的信息素浓度选择走哪一条路。信息素浓度越高，则蚂蚁选择该路径的概率越大。

（3）随着之后蚂蚁均遵循（2）规则，则越短的路径上的信息素浓度越高。

（4）最终，蚂蚁觅食找到最短路径，即最优路径。

2.3.2 蚁群算法的优缺点

蚁群算法非常适合解决资源调度问题[42]，其具体的优缺点如下：

（1）优点

a. 较强的鲁棒性。蚁群算法模型稍作修改便适用于其他问题，蚁群算法得到的最优解与初始蚂蚁的路径选择关系不大。

b. 易与其他算法结合。由于蚁群算法具有较少的参数，较低的复杂性，可以与其他算法结合解决问题，优化其性能。

c. 并行性。每只蚂蚁的选择互不影响，相互独立，可以并行实现，以寻找最优解。

（2）缺点

a. 易陷入局部最优。由于之后的蚂蚁会选择之前蚂蚁留下的信息素浓度高的路径，容易导致多只蚂蚁均选择同一条路径，从而陷入局部最优。

b. 收敛速度慢。蚁群算法的计算能力有局限性，求解速度慢，并且蚂蚁随机选择下一个爬行点，虽有助于找到全局最优解，但是需要更长时间发挥正反馈作用。

2.3.3 蚁群算法的基本流程

以 TSP[43]（Travel Salesperson Problem，即旅行商问题）为例，蚁群算法的基本流程如图 2.5 所示。

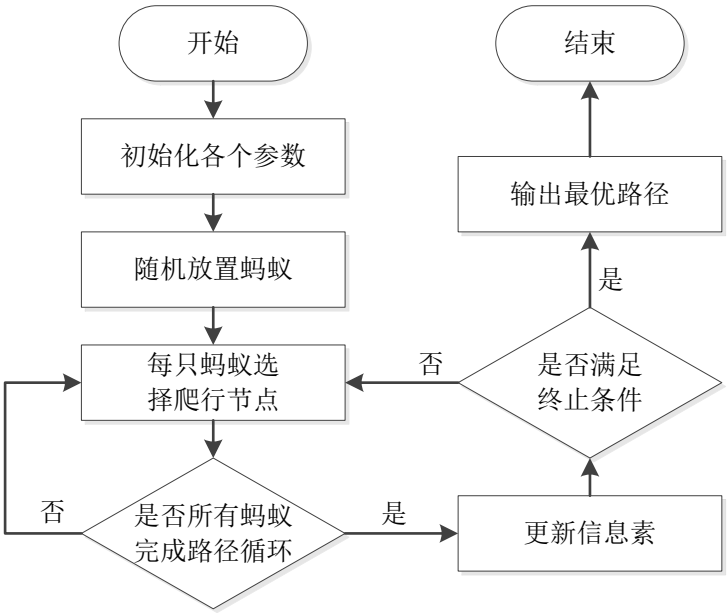


图 2.5 蚁群算法基本流程图

Figure 2.5 Basic Flow Chart of Ant Colony Algorithm

基本步骤如下：

（1）初始化各个参数以及城市之间路径上的信息素浓度；

（2）假设有 x 只蚂蚁，将这些蚂蚁随机放置到各个城市；

（3）蚂蚁按照概率转移规则选择下一个城市节点，并将走过的城市加入禁忌表；

（4）当每只蚂蚁完成路径循环，则更新路径上的信息素浓度，否则，转到步骤（3）；

（5）若是达到算法终止条件，则输出最优路径，结束流程，否则，清空禁忌表并转到步骤（3）。

在本文优化的弹性扩容策略中，通过改进的蚁群算法对 Kubernetes 现有扩容策略的节点选定过程进行优化，为新增 Pod 选择部署节点。

2.4 本章小结

本章首先介绍了 Docker 容器技术的基本概念以及 Docker 与传统虚拟化技术的对比，然后对最流行的容器集群编排系统 Kubernetes 的核心概念和系统架构做了介绍，着重分析了 Kubernetes 现有的弹性扩缩容策略，最后对经典蚁群算法的原理、优缺点以及基本流程进行了概述，为本文的主要研究内容做铺垫。

第三章 弹性扩容策略的优化设计

本章对 Kubernetes 现有的扩容策略存在的问题进行分析，针对这些问题，本章提出了一种基于改进蚁群算法（EACO）的资源调度策略，并对现有的扩容策略进行优化，优化后的弹性扩容策略分为节点预选、优选评分和基于改进蚁群算法的资源调度三个过程，而基于改进蚁群算法的资源调度策略以集群负载均衡度和资源消耗成本为目标函数进行寻解。

3.1 Kubernetes 扩容策略问题分析

通过对 Kubernetes 现有扩容策略的分析可知，现有的扩容策略将新增 Pod 调度到合适的资源节点的过程主要包括预选、优选和选定，预选过程排除不符合条件的节点，优选过程对候选资源节点进行评分，然后选定得分最高的节点进行部署。但是，Kubernetes 现有的扩容策略仍存在一些不足之处，主要表现在以下两个方面。

（1）Kubernetes 现有扩容策略为新增 Pod 选择节点时，优先考虑当前性能最优的节点，即节点优选评分过程中得分最高的节点，但是未考虑多个 Pod 部署后集群负载均衡的问题，例如多个 Pod 可能被调度到同一个节点上，导致 Kubernetes 集群中部分节点部署的 Pod 数量过多，部分节点部署的 Pod 数量较少，从而造成集群负载不均衡的问题。

（2）Kubernetes 现有扩容策略为新增 Pod 选择节点时，未考虑多个 Pod 部署后资源消耗成本的问题，由于不同资源节点单位时间内的资源单价不同，当多个长期运行的 Pod 部署后，可能会占用单位时间内资源单价较高的节点的资源，从而造成资源消耗成本过高的问题。

针对上述问题，本章提出在为新增 Pod 选择节点时，要综合考虑集群负载均衡和资源消耗成本的因素。对于集群负载均衡，用集群中单个节点所分配 Pod 数量的最大值与待调度 Pod 总数量的比值来衡量，该值越小，说明集群负载越均衡；对于资源消耗成本，由于不同资源节点单位时间内的资源单价不同，所以用所有 Pod 部署到各个资源节点上，在节点开始工作后单位时间内所消耗的资源总成本来衡量。

而如何保证所有的新增 Pod 部署后集群负载更加均衡和资源消耗成本较小，是本文研究的重点。

3.2 节点预选和优选评分

上一节对 Kubernetes 现有的扩容策略存在的问题进行了分析，本节主要介绍本文优化后的弹性扩容策略的节点预选和优选评分两个过程。

3.2.1 节点预选过程

节点预选过程采用 Kubernetes 现有扩容策略的预选策略，预选策略实际上就是节点过滤器，执行该操作时，会根据预选过程的规则逐一对资源节点进行判断，节点需要同时满足所有的预选规则，才能将其作为 Pod 调度的资源候选节点。

节点预选策略[44]包括 CheckNodeCondition（检查节点是否满足调度条件）、PodFitsHostPorts（检查节点上的 Pod 端口是否被占用）、PodFitsResource（检查节点上可用资源是否满足 Pod 需求）等规则。

以 PodFitsResource 规则为例，即检查节点上的可用资源（CPU、内存）是否满足 Pod 的运行需求。假设Q_{ci}和Q_{mi}（i=1,2,…,m）分别表示Pod_i的 CPU 和内存请求量，N_{cj}和N_{mj}（j=1,2,…,n）分别表示节点 j 的 CPU 和内存空闲量。

PodFitsResource 规则如下：

若满足$Q_{ci} \le N_{cj}$且$Q_{mi} \le N_{mj}$，则 j 节点为候选资源节点。

3.2.2 优选评分过程

先通过预选策略选择出符合条件的候选节点，然后对候选资源节点进行评分并排序，本文考虑节点的资源空闲率和节点的资源平衡性作为评分标准。

（1）节点的资源空闲率

尽量将 Pod 调度到资源占用比较小的资源节点上，依据节点空闲的资源量与节点总资源量的比值决定，即计算(节点资源总量-节点上 Pod 所占资源量之和)/节点资源总量。假设$Scpu$表示资源节点空闲 CPU 占节点 CPU 总容量的比例，计算方法如公式 3.1 所示。

$$Scpu = \frac{Tcpu - Ecpu}{Tcpu} \tag{3.1}$$

其中，$Tcpu$ 表示节点的 CPU 总量，$Ecpu$ 表示节点上已部署 Pod 占用的 CPU 量。

假设$Smem$ 表示资源节点空闲内存占节点内存总容量的比例，计算方法如公式 3.2 所示。

$$Smem = \frac{Tmem - Emem}{Tmem} \tag{3.2}$$

其中，*Tmem* 表示节点的内存总量，*Emem* 表示节点上已部署 Pod 所占内存之和。

假设 *Score*1 表示节点资源空闲率得分，计算方法如公式 3.3 所示。

$$Score1 = \frac{Scpu + Smem}{2} \tag{3.3}$$

（2）节点的资源平衡性

节点的资源平衡性，即节点的 CPU 和内存使用率接近。若是节点的 CPU 和内存使用情况较为平衡，会避免节点的 CPU 使用过多，而内存剩余较多，同理，也会避免节点的内存已经使用过多，而 CPU 却剩余较多。节点资源平衡越好，选择该节点的可能性越大。假设 *Score*2 表示资源的平衡性得分，计算方法如公式 3.4 所示。

$$Score2 = \frac{1 - abs(Scpu - Smem)}{2} \tag{3.4}$$

*Score*2 值越大，节点资源的平衡越好。

（3）候选资源节点总分

假设 *Score* 表示候选节点的总分，计算方法如公式 3.5 所示。

$$Score = Score1 + Score2 \tag{3.5}$$

依次对所有候选资源节点进行评分后，根据总分值高低进行排序，分数值越高表示节点的资源空闲率越大，平衡性越好。而节点最高的分数值作为改进蚁群算法的信息素浓度初始值。

3.3 基于改进蚁群算法的资源调度

经过节点预选和优选评分过程，将得到的候选节点的最高分数值作为本文改进蚁群算法的信息素浓度初始值，然后以集群负载均衡度和资源消耗成本为目标函数，通过本文提出的改进蚁群算法（EACO）的资源调度策略将 Pod 调度到最优的资源节点。本节主要介绍本文提出的基于改进蚁群算法的资源调度策略。

3.3.1 形式化定义

本文资源调度过程中的一些形式化定义如下：

定义 1　假设 Pod 集合为 P，P=$\{Pod_i | 1 \le i \le m\}$，表示当前 Pod 集合有 m 个新增 Pod，每个新增 Pod 只能调度到一个资源节点上。

定义 2　假设候选资源节点集合为 N，N=$\{Node_j|1\leq j\leq n\}$，表示满足预选条件的 n 个候选资源节点构成的集合。

定义 3　假设分配矩阵由 Y 表示，Pod 集合与资源节点集合 N 的分配矩阵 Y=$\{y_{ij}|1\leq i\leq m$，$1\leq j\leq n\}$，y_{ij}为分配节点，表示Pod_i分配到$Node_j$上。

定义 4　禁忌表 $tabu$，当 t 时刻第 k 只蚂蚁选择分配节点y_{ij}后，则分配节点集合$\{y_{ij}|1\leq j\leq n\}$加入禁忌表。

针对以上形式化定义，本章求解的目标函数如 3.3.2 节所述。

3.3.2 目标函数的建立

现有扩容策略在对新增 Pod 进行调度时，未考虑多个 Pod 部署后，可能造成的集群负载不均衡以及 Pod 长期运行占用资源导致节点资源消耗成本过高的问题，因此，本文提出构建集群负载均衡度和资源消耗成本的目标函数。

（1）集群负载均衡度目标函数

集群负载均衡度通过集群中单个节点所分配 Pod 数量的最大值与待调度 Pod 总数量的比值来衡量。集群负载均衡度的目标函数值f_l计算方法如公式 3.6 所示。

$$f_l=\frac{\max\left(Node_{jl}\right)}{m}\quad 1\leq j\leq n \tag{3.6}$$

其中，$Node_{jl}$表示节点 j 上分配的 Pod 的数量，$max(Node_{jl})$表示当前集群中单个节点所分配 Pod 数量最大的值，m 为待调度的 Pod 数量，f_l值越小，说明集群负载越均衡。

（2）资源消耗成本目标函数

Pod 调度到资源节点后，会占用资源节点的 CPU 和内存资源，由于不同资源节点单位时间内资源单价不同，可以构建出所有新增 Pod 调度于各个资源节点上的资源消耗成本的目标函数。

假设f_{ic}表示为Pod_i分配到$Node_j$上单位时间内资源消费的总成本，计算方法如公式 3.7 所示。

$$f_{ic}=\cos t_{jcpu}*h_{ci}+\cos t_{jmem}*h_{mi} \tag{3.7}$$

其中，$cost_{jcpu}$为 j 资源节点单位时间内的 CPU 单价，$cost_{jmem}$为 j 资源节点单位时间内的内存单价，h_{ci}和h_{mi}分别为Pod_i所需的 CPU 和内存的资源总量。

所有 Pod 分别分配到节点后，单位时间内资源消费成本f_{cost}计算如公式 3.8 所示。

$$f_{cost} = \sum_{i=1}^{m} f_{ic} \tag{3.8}$$

所有 Pod 全部分配在单位时间内资源单价最大的节点上，资源单价最大以节点 CPU 和内存的最大单价和来衡量，资源消费的最大成本$f_{maxcost}$计算如公式 3.9 所示。

$$f_{\max cost} = \sum_{i=1}^{m} \left(cost_{\max cpu} * h_{ci} + cost_{\max mem} * h_{mi} \right) \tag{3.9}$$

所有 Pod 全部分配在单位时间内资源单价最小的节点上，资源单价最小以节点 CPU 和内存的最小单价和来衡量，资源消费的最小成本$f_{mincost}$计算如公式 3.10 所示。

$$f_{\min cost} = \sum_{i=1}^{m} \left(cost_{\min cpu} * h_{ci} + cost_{\min mem} * h_{mi} \right) \tag{3.10}$$

所以，归一化后的资源消耗成本目标函数f_c计算如公式 3.11 所示。

$$f_c = \frac{f_{cost} - f_{\min cost}}{f_{\max cost} - f_{\min cost}} \tag{3.11}$$

其中，f_c越小，表示资源消耗成本目标函数值越小，即该算法所得分配方案越好。

（3）总目标函数

总目标函数是集群负载均衡度目标函数和资源消耗成本目标函数的加权求和，计算如公式 3.12 所示。

$$F = a * f_c + b * f_l \tag{3.12}$$

其中，f_c为当前分配方案的资源消耗的总成本，f_l为当前分配方案的集群负载均衡度，a 和 b 为两者权重，根据用户需求设定，$a+b=1$，$a \in [0,1]$，$b \in [0,1]$。F 表示总目标函数值，该值越小，说明对应的分配方案越好。

3.3.3 改进蚁群算法在资源调度中的转换

建立好目标函数后，对改进蚁群算法在解决本文资源调度问题时进行转换，从而构建改进蚁群算法在资源调度中的模型。转换方式如下：

（1）在本文资源调度中，以所有 Pod 部署后的集群负载均衡度和资源消耗成本为目标函数求解最优的部署方案。

（2）在本文资源调度中，蚂蚁的爬行节点为由 Pod 集合和候选资源节点集合构成的分配矩阵的节点，分配矩阵的每个节点代表一个 Pod 在资源节点上的分配情况，即y_{ij}代表编号为

i 的 Pod 分配到编号为 j 的资源节点上。而经典的蚁群算法一般以候选的资源节点作为蚂蚁爬行的节点。

（3）在本文资源调度中，信息素是作用于由 Pod 集合和候选资源节点集合构建的分配矩阵的节点上。

（4）在本文资源调度中，信息素浓度的初始值为优选评分过程中候选节点最高的分值。

本文针对经典蚁群算法存在的收敛速度慢，易陷入局部最优问题的改进方式如下：

（1）针对收敛速度慢问题，本文采用精英蚂蚁[45]的方式改进，精英蚂蚁是全部蚂蚁完成自己的路径循环后，所有路径中所得总目标函数值最小的路径对应的蚂蚁。精英蚂蚁会像其它蚂蚁一样更新完自己路径上分配节点的信息素后，再重复一遍信息素的更新，以此加快算法收敛。

（2）针对易陷入局部最优的问题，本文采用轮盘赌[46][47]的方式改进，即在蚂蚁选择下一个分配节点时，将选择每个分配节点的概率看作是轮盘的一个扇面，旋转轮盘，指针停在哪一个扇面上就选择对应概率的分配节点，这样增加蚂蚁选择下一个分配节点的随机性，从而避免陷入局部最优。轮盘示例如图 3.1 所示。

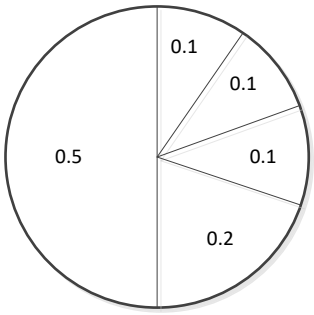


图 3.1 轮盘示例图[46]

Figure 3.1 An Example Diagram of A Roulette Wheel

3.3.4 改进蚁群算法的资源调度模型

将改进蚁群算法应用于本文资源调度问题的解决，构建改进蚁群算法的资源调度模型。

（1）蚂蚁位置及信息素的初始化

设定 x 只蚂蚁，初始每只蚂蚁会随机选择 Pod 集合 P=$\{Pod_i|1\leq i\leq m\}$与资源节点集合 Node=$\{Node_j|1\leq j\leq n\}$的分配矩阵 Y 的某个节点y_{ij}作为起始节点。

本文将优选评分过程中所有候选节点最高的分数值设置为信息素浓度初始值，公式如 3.13 所示。

$$\tau_{ij}(0) = MaxScore \tag{3.13}$$

其中，*MaxScore* 表示所有候选节点中最高的分数值；$\tau_{ij}(0)$表示所有分配节点的信息素浓度初始值。

（2）蚂蚁爬行路径

蚂蚁随机放置在 *m* 个新增 Pod 和 *n* 个候选资源节点组成的分配矩阵的节点上，而蚂蚁选择下一个节点y_{ij}的概率P_{ij}^{k}的计算公式如 3.14 所示。

$$P_{ij}^{k} = \begin{cases} \dfrac{\tau_{ij}^{\alpha}(t)\eta_{ij}^{\beta}(t)}{\sum_{n \in allowed_k} \tau_{in}^{\alpha}(t)\eta_{in}^{\beta}(t)}, & j \in allowed_k \\ 0, \text{ 其他} \end{cases} \tag{3.14}$$

公式 3.14 中的各项含义如下：

$allowed_k$表示蚂蚁k下一步可以选择的分配节点集合；相反的，$tabu_k$表示第 *k* 只蚂蚁的禁忌表，即此蚂蚁已走过的分配节点集合。由于一个 Pod 只能分配到一个资源节点上，所以当 *t* 时刻第 *k* 只蚂蚁选择分配节点y_{ij}后，就将分配节点集合$\{y_{ij}|1 \leq j \leq n\}$加入禁忌表；

$\tau_{ij}(t)$表示 *t* 时刻在分配节点y_{ij}上的信息素浓度；

α 为反映信息素重要程度的启发因子，会影响其它蚂蚁选择下一个爬行的分配节点，该值越大，则该蚂蚁越可能选择其他蚂蚁爬行过的节点；

β为反映蚂蚁选择分配节点受重视程度的启发因子；

$\eta_{ij}(t)$表示 *t* 时刻Pod_i分配到$Node_j$的期望值，该值为总目标函数的倒数，如公式 3.15 所示。

$$\eta_{ij}(t) = \frac{1}{F_{ij}^{k}(t)} \tag{3.15}$$

其中，$F_{ij}^{k}(t)$表示 *t* 时刻第 *k* 只蚂蚁选择分配节点y_{ij}时的总目标函数值。

同时为了避免算法陷入局部最优，蚂蚁选择下一个允许选择的分配节点时，采用轮盘赌算法的方式，将每个分配节点的概率看作是轮盘的一个扇面，旋转轮盘，指针停在哪一个扇面上就选择对应概率的分配节点。这样做概率较小的分配节点也有可能被选中，增加了蚂蚁选择下一个爬行节点的随机性，避免算法陷入局部最优。

（3）信息素更新

在资源调度过程中，随着蚂蚁不断地爬行搜索，之前的蚂蚁在某个分配节点上留下的信息素会有一定的挥发，而之后的蚂蚁又会在该分配节点上留下新的信息素，因此每只蚂蚁在完成一次路径循环后，会对路径上分配节点的信息素浓度进行更新，采用精英蚂蚁的规则来更新 *t+s* 时刻在分配节点y_{ij}上的信息量。

当所有蚂蚁完成一次路径循环，分配节点信息素更新公式如 3.16、3.17 所示。

$$\tau_{ij}(t+s)=(1-\rho)\tau_{ij}(t)+\sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t) \tag{3.16}$$

$$\Delta\tau_{ij}^{k}(t)=\begin{cases}\dfrac{Q}{F_k}, & \text{若第}k\text{只蚂蚁在本次循环中选择 } y_{ij}\\ 0, & \text{其他}\end{cases} \tag{3.17}$$

本文采用精英蚂蚁策略对信息素浓度更新进行改进，精英蚂蚁是全部蚂蚁完成自己的路径循环后，所有路径中所得总目标函数值最小的路径对应的蚂蚁。当全部蚂蚁完成对自己路径上分配节点信息素浓度的更新后，还会额外对精英蚂蚁所爬行路径上的分配节点的信息素浓度进行更新，但要把更新值乘上一个参数 *e*。精英蚂蚁策略更新信息素浓度的公式如 3.18、3.19 所示。

$$\tau_{ij}(t+s)=(1-\rho)\tau_{ij}(t)+\sum_{k=1}^{m}\Delta\tau_{ij}^{k}(t)+e*\Delta\tau_{ij}^{*}(t) \tag{3.18}$$

$$\Delta\tau_{ij}^{*}(t)=\begin{cases}\varphi\dfrac{Q}{F^{*}}, & \text{若蚂蚁选择}y_{ij}\\ 0, & \text{其他}\end{cases} \tag{3.19}$$

公式 3.16、3.17、3.18、3.19 中，ρ表示信息素挥发系数，则 1-ρ表示信息素残留因子，ρ的取值范围为：$\rho\in[0,1]$；$\Delta\tau_{ij}^{k}(t)$表示蚂蚁 *k* 在时间 *t* 到时间 *t+s* 内的访问过程中，在分配节点y_{ij}上留下的信息素浓度，即蚂蚁 *k* 本次循环中分配节点y_{ij}上的信息素增量；Q 表示信息素强度，为常量；F_k表示蚂蚁 *k* 在本次循环中总目标函数值；$\Delta\tau_{ij}^{*}(t)$表示精英蚂蚁额外增加的信息素量；e的取值范围为：$e\in[0,1]$；F^{*}表示精英蚂蚁在本次循环中总的目标函数的值；φ为精英蚂蚁的数量。

（4）算法终止条件

在本文提出的基于改进的蚁群算法（EACO）的资源调度策略中，算法的终止条件不再是算法迭代固定次数，如迭代 50 次算法终止，而是当算法迭代了 n 次时，若最近两次迭代循环中得到的总目标函数值相差不到 0.5%（该值根据多次实验设定），则算法终止，否则进入下一次循环。通过对算法终止条件的改进，可以有效避免算法迭代次数设置太小，不一定找到最优解，而循环次数设置太大，会消耗过多时间的情况。

3.3.5 改进蚁群算法的流程

通过前面对改进蚁群算法在本文资源调度问题的应用，本节主要对本文改进的蚁群算法（EACO）的流程进行介绍，算法流程如图 3.2 所示。

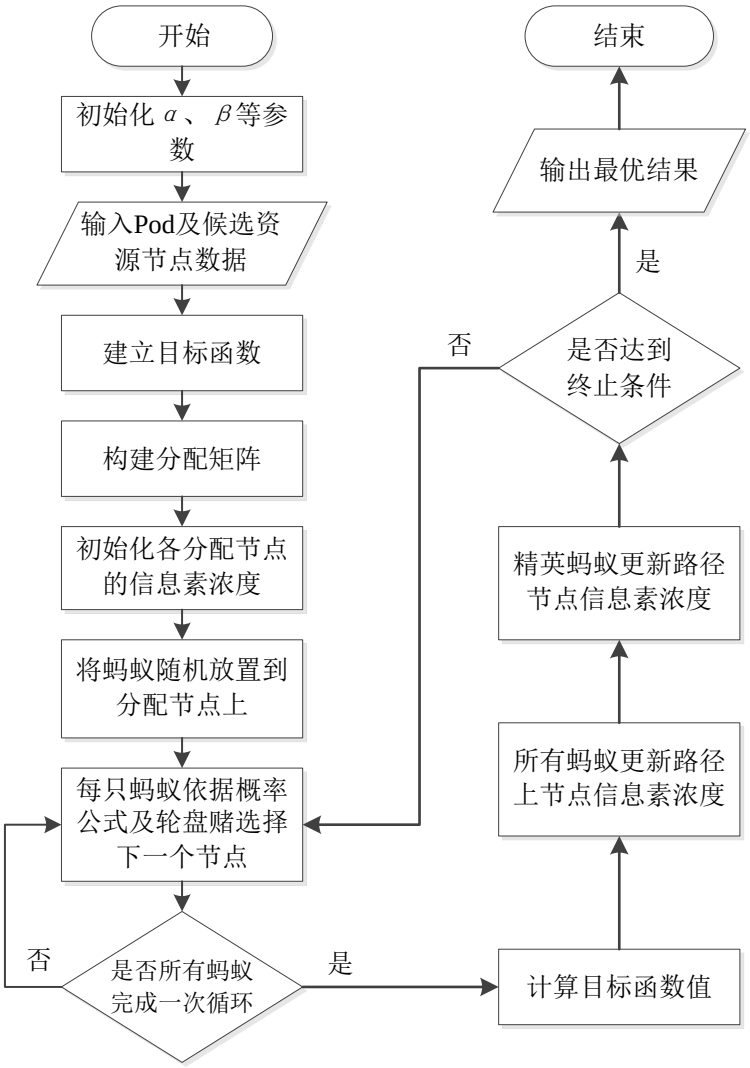


图 3.2 改进蚁群算法的流程图

Figure 3.2 Flow Chart of The Improved Ant Colony Algorithm

本文改进蚁群算法（EACO）的流程如下：

Step1 初始化参数。初始化改进算法中α、β、ρ、e、Q 常数以及目标函数权重 a 和 b 等各种参数。

Step2 建立目标函数。根据资源调度中单位时间内所有 Pod 的资源消耗成本和集群负载度建立目标函数。

Step3 构建分配矩阵。通过 Pod 集合 P 与资源节点集合 Node 构建分配矩阵 Y，将 x 只蚂蚁随机放置在分配矩阵的节点上。

Step4 初始化分配节点的信息素浓度。将候选节点的最高分值作为初始化各节点的信息素浓度。

Step5 将蚂蚁随机放置到分配节点上。

Step6 蚂蚁爬行规则。蚂蚁依据概率公式计算爬向允许选择的各个分配节点的概率，并通过轮盘赌算法决定移动的下一个节点，当 t 时刻第 k 只蚂蚁选择分配节点y_{ij}后，将分配节点$\{y_{ij}|0\leq j\leq n-1\}$加入禁忌表$tabu_k$。

Step7 若所有蚂蚁完成一次循环则跳转 Step8；否则跳转 Step6。

Step8 计算总目标函数值并更新分配节点信息素。当所有蚂蚁完成一次循环，计算总目标函数值并按照更新信息素公式对节点信息素进行更新，而精英蚂蚁再重复一遍信息素的更新过程。

Step9 若满足终止条件，循环结束输出结果即分配方案；否则清空禁忌表并跳转到 Step6。

将以上基于改进蚁群算法的资源调度策略对 Kubernetes 现有扩容策略的节点选定过程进行优化，3.4 节将介绍优化后的弹性扩容策略。

3.4 整体优化的弹性扩容策略

本节介绍优化后的弹性扩容策略，具体流程如图 3.3 所示。

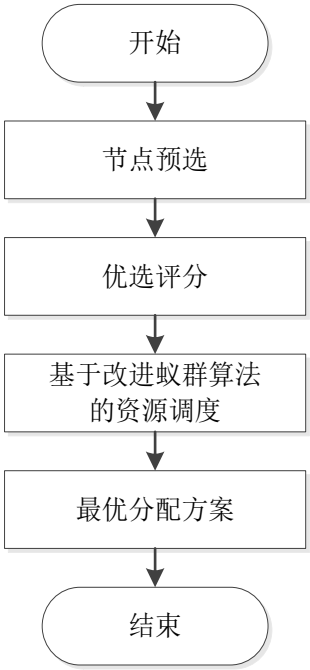


图 3.3 整体的优化扩容策略流程图

Figure 3.3 Flow Chart of Overall Optimization Expansion Strategy

优化后的扩容策略步骤如下：

Step1 节点预选，通过本文采用的预选策略排除不满足条件的节点，而满足条件的节点作为候选资源节点。

Step2 优选评分，根据节点资源空闲率及节点资源平衡性对候选资源节点评分并排序，节点最高的分数作为改进蚁群算法的初始信息素。

Step3 基于改进蚁群算法的资源调度策略选择出综合考虑集群负载度和节点消耗成本时最优的分配方案。

3.5 本章小结

本章分析了 Kubernetes 现有的资源扩容策略存在的问题，提出了一种基于改进蚁群算法（EACO）的资源调度策略，对现有的扩容策略进行优化。优化的弹性扩容策略分为节点预选、优选评分和基于改进蚁群算法的资源调度三个过程。预选过程采用 Kubernetes 现有扩容策略的预选策略，对候选资源节点评分排序后，通过改进蚁群算法的资源调度策略为新增 Pod 选择最优的分配节点，该策略考虑了资源调度中单位时间内所有 Pod 的资源消耗成本和集群负载均衡度，基于这两个因素提出了改进蚁群算法的目标函数，用来反映 Pod 分配的好坏。在综合考虑集群负载均衡度和节点资源消耗成本时，得到新增 Pod 的分配方案。

第四章 弹性缩容策略的优化设计

本章对 Kubernetes 现有缩容策略存在的问题进行分析，针对这些问题，本章提出了基于节点多维资源平衡的缩容策略和基于节点资源消耗成本的缩容策略，并将这两种缩容策略同现有的缩容策略进行整合优化。

4.1 Kubernetes 缩容策略问题分析

在 Kubernetes 中虽然已有弹性缩容策略，但是现有缩容策略仅根据 Pod 的状态优先级来选择待删 Pod，存在的问题如下：

（1）Kubernetes 现有的缩容策略选择待删 Pod 时未考虑删除 Pod 后节点资源平衡的问题。例如删除 Pod 后可能会造成节点上各维度资源使用不平衡，易产生资源碎片。某节点上部署着不同类型的 Pod，如 CPU 密集型、内存密集型、CPU 和内存平衡型等，而不同类型的 Pod 对节点上不同维度资源的需求情况不同，该节点删除某一类型的 Pod 后，大量增加对应维度的资源，其他维度资源几乎没有增加，当部署其他类型 Pod 时，节点的剩余资源可能无法满足这种类型 Pod 的资源需求，并且若是与被删除 Pod 同样类型的 Pod 以后不会再出现，那么该节点的剩余资源很难被再次利用，长时间属于空闲状态，就会产生资源碎片，造成资源浪费。

（2）Kubernetes 现有缩容策略选择待删 Pod 时未考虑删除 Pod 后节点的资源消耗成本的问题。当某资源节点删除其上任意一个 Pod 后的各维度资源平衡时，考虑节点的资源消耗成本因素，优先删除单位时间内资源消耗成本较大的 Pod，可以有效降低该节点的资源消耗成本。

通过对 Kubernetes 现有的弹性缩容策略的问题分析可知，在考虑 Pod 本身状态优先级的情况下，增加对节点各维度资源平衡和节点资源消耗成本的考虑可以有效解决这些问题。

4.2 基于节点多维资源平衡的缩容策略

上一节中对 Kubernetes 现有的弹性缩容策略存在的问题进行分析，而本节则主要考虑如何解决删除 Pod 后节点各维度资源不平衡及产生资源碎片的问题。在触发缩容操作后，会删

除一定数量的 Pod，但是由于在删除 Pod 后可能会改变各资源节点的各维度资源的利用率，因此当进行缩容时，需要考虑删除 Pod 后对节点各维度资源平衡的影响。

由于 Kubernetes 集群中某个资源节点上总是同时运行着多种不同类型的 Pod，而不同类型的 Pod（如 CPU 密集型、内存密集型、CPU 和内存平衡型等）对不同维度资源的需求情况具有较大的差异性。在删除某一类型 Pod 后，会增加对应维度的资源，可能较难满足其他类型 Pod 的需求。而若被删除的 Pod 所在的节点又在很长一段时间内得不到扩容，或者与被删除 Pod 同样类型的 Pod 以后都不会再出现，那么该节点剩余的资源将不能被再次利用，就会产生资源碎片，造成该节点资源的浪费。

针对上述问题，本文提出了一种基于节点多维资源平衡的缩容策略，在缩容操作之前本文会预先计算删除该 Pod 后所在节点的各维度资源利用率。同时通过方差来衡量资源节点在删除 Pod 后的各维度资源的平衡情况，方差值越小代表节点各维度资源越平衡，反之则越不平衡。

本文以 CPU 和内存作为两个维度的资源指标进行研究，本文提出的节点多维资源平衡缩容策略的流程如图 4.1 所示。

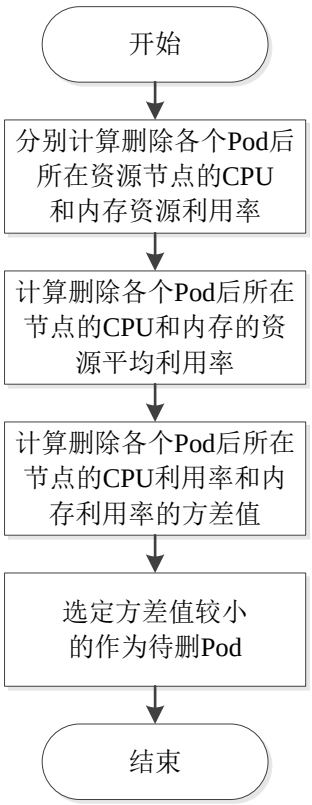


图 4.1 节点多维资源平衡缩容策略的流程图

Figure 4.1 Flow Chart of Node Multi-dimensional Resource Balance Shrinkage Strategy

假设某个资源节点上的 CPU 总量为T_{cpu}，内存总量为T_{mem}；节点已分配的 CPU 量为A_{cpu}，内存量为A_{mem}；该节点上的某个 Pod 的资源配额为 CPU 量为B_{cpu}，内存量为B_{mem}。

首先计算删除该 Pod 后的资源节点的资源利用率U_{cpu}和U_{mem}，计算公式如 4.1 和 4.2 所示。

$$U_{cpu} = \frac{A_{cpu} - B_{cpu}}{T_{cpu}} \tag{4.1}$$

$$U_{mem} = \frac{A_{mem} - B_{mem}}{T_{mem}} \tag{4.2}$$

然后再计算在删除该 Pod 后，该节点的资源平均利用率$\overline{U}$，计算公式如 4.3 所示。

$$\bar{U} = \frac{U_{cpu} + U_{mem}}{2} \tag{4.3}$$

最后计算方差值S^2，计算公式如 4.4 所示。

$$S^2 = \frac{\left(U_{cpu} - \bar{U}\right)^2 + \left(U_{mem} - \bar{U}\right)^2}{2} \tag{4.4}$$

其中，S^2为方差值，该值越小，说明删除该 Pod 后，其所在资源节点的 CPU 和内存资源使用就越平衡。

4.3 基于节点资源消耗成本的缩容策略

Kubernetes 现有的缩容策略在删除 Pod 时也未考虑到待删 Pod 所在资源节点的资源消耗成本的问题，因此在删除 Pod 时，增加对节点资源消耗成本因素的考虑，当某资源节点删除其上任意一个 Pod 后的各维度资源平衡时，优先删除单位时间内资源消耗成本较大的 Pod，从而降低该节点的资源消耗的成本。

针对于上述问题，本文提出了一种基于节点资源消耗成本的缩容策略，该策略在资源节点的各维度资源平衡的前提下，会预先计算该 Pod 所在资源节点单位时间内的资源消耗成本。

本文以 CPU 和内存作为资源指标，因此，本文提出的节点资源消耗成本缩容策略的流程如图 4.2 所示。

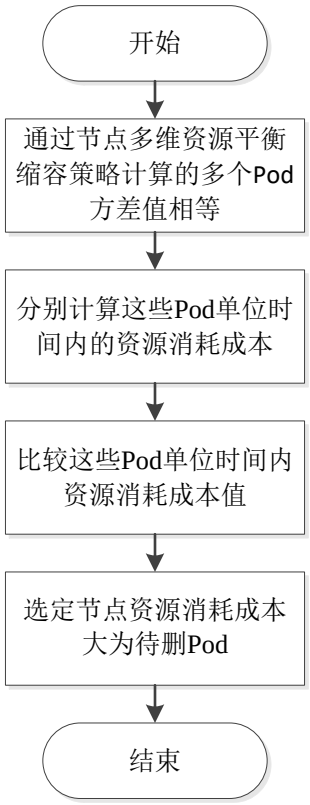


图 4.2 节点资源消耗成本缩容策略的流程图

Figure 4.2 Flow Chart of Node Resource Consumption Cost Reduction Strategy

假设某个资源节点上的某个 Pod 的资源配额为 CPU 量为B_{cpu}和内存量为B_{mem}；节点单位时间内的 CPU 单价为C_{cpu}、内存单价为C_{mem}。

计算该 Pod 所在资源节点单位时间内的资源消耗成本V，计算公式如 4.5 所示。

$$V = B_{cpu} * C_{cpu} + B_{mem} * C_{mem} \tag{4.5}$$

V表示该 Pod 所在资源节点单位时间内的资源消耗成本，值越大，说明资源节点上该 Pod 单位时间内消耗的资源成本大，优先删除资源节点上资源消耗成本大的 Pod。

4.4 整体优化的弹性缩容策略

本章通过对 Kubernetes 现有的缩容策略的分析得知，现有缩容策略在选择待删 Pod 时只考虑了 Pod 所处的状态，优先删除状态优先级较小的 Pod，而未考虑删除该 Pod 后造成的节点各维度资源不平衡，易产生资源碎片的问题和节点资源消耗成本的问题。因此，本文提出的优化的弹性缩容策略主要是解决以上未考虑的问题。

本文将基于节点多维资源平衡的缩容策略和基于节点资源消耗成本的缩容策略这两种策略同 Kubernetes 现有的缩容策略进行整合优化。由于节点各维度资源平衡和节点资源消耗成本的重要性，对于多个 Pod 处在同一种状态时，即多个 Pod 同处于未就绪状态、就绪状态、已调度状态和未调度状态等，优先考虑本文提出的这两种策略，然后再根据 Pod 时间长短、Pod 内部容器的重启次数等因素进行排序来确定待删 Pod。

综合考虑 Pod 本身的状态优先级和节点多维资源平衡以及节点资源消耗成本的因素，整体的优化缩容策略如下：

Step1 根据确定好的待删 Pod 数量，遍历节点上的所有 Pod，依据 Pod 本身的状态优先级进行排序，优先删除状态优先级小的 Pod。

状态优先级策略如下：

（1）首先根据 Pod 是否处于就绪的状态进行状态优先级排序，未就绪的 Pod 的优先级小于已就绪的 Pod。

（2）根据未就绪 Pod 是否处于已被调度状态进行状态优先级排序，未调度 Pod 的优先级小于已调度 Pod 的优先级。

（3）根据已调度的 Pod 所处的状态分为 Pending、Unknown、Running，优先级依次增大。

Step2 对于状态优先级相等的多个 Pod，通过节点多维资源平衡的缩容策略进行方差值计算，优先删除资源节点上计算所得方差值较小的 Pod。

Step3 对于多个 Pod 通过节点多维资源平衡缩容策略计算的方差值相同的情况，再通过节点资源消耗成本的缩容策略进行 Pod 单位时间内消耗成本的计算，然后优先删除节点上资源消耗成本较大的 Pod。

Step4 对于节点资源消耗成本一样的多个 Pod，如果它们处于就绪状态，那么就根据就绪时间的长短来进行排序，优先删除就绪时间短的 Pod。

Step5 对于资源消耗成本相等且未就绪的多个 Pod 或者就绪时间一致的多个 Pod，比较 Pod 中容器的重启次数，优先删除容器重启次数多的 Pod。

Step6 对于多个 Pod 中容器的重启次数一致的情况，比较 Pod 创建的时间，优先删除创建时间较晚的 Pod。

4.5 本章小结

本章分析了 Kubernetes 现有的资源缩容策略，针对于在删除 Pod 后存在的节点各维度资源不平衡，易产生资源碎片以及未考虑资源消耗成本的问题，提出了基于节点多维资源平衡的缩容策略和基于节点资源消耗成本的缩容策略，并将这两种策略同现有的缩容策略进行整体优化，得到了本文优化后的缩容策略。优化后的缩容策略使得删除 Pod 后节点资源使用更加平衡，并减少了资源碎片的产生，也降低节点资源消耗成本。

第五章 实验与分析

本章对优化的扩容和缩容策略进行实验验证，首先将本文提出的策略应用到 CloudSim 云计算仿真平台进行实验，然后针对于优化的扩容和缩容策略分别设计对比实验，最后对实验的结果进行分析。

5.1 实验环境

5.1.1 环境配置和数据集

本文在 ColudSim 云计算仿真平台进行实验，实验环境配置如表 5.1 所示。

表 5.1 实验环境配置表

Table 5.1 Experimental Environment Configuration Table

软/硬件	参数
操作系统	Windows10 专业版
开发平台	IDEA64
CloudSim	3.0
开发语言	Java
JDK	1.8

本文实验采用 WeaveScope（Kubernetes 的容器监控工具）监控的真实数据[48]。

5.1.2 ColudSim 简介

CloudSim 是由澳大利亚墨尔本大学的网格实验室和 Gridbus 项目宣布推出的云计算仿真软件[49]。

（1）特点：a.支持云计算项目的建模与仿真；b.支持数据中心、服务代理人、调度和分配策略。

（2）功能：提供虚拟化服务，对虚拟化服务分配处理核心时在时间共享和空间共享之间灵活切换。

（3）体系结构

CloudSim 的软件结构框架和体系结构组件包括 SimJava、GridSim、CloudSim、UserCode 四个层次[50]。SimJava 层是实现了上层仿真框架所需核心函数的仿真引擎，如仿真模块管理、组件间通信等；GridSim 层主要用于网格基础设施建模，包括相关交易规范、基础网格组件和网络的高层组件；CloudSim 层为虚拟化的云数据中心提供仿真和建模支持，包括为虚拟机、带宽和存储提供专用的管理接口；UserCode 层为云环境中的各种实体提供相关的配置函数，包括主机的数量和规范、用户提交的任务数和任务需求、虚拟机、用户数量以及用户的应用类型等。

5.2 优化扩容策略实验与结果分析

为了验证提出的优化的弹性扩容策略，本文选择云仿真平台 CloudSim3.0 对提出的策略进行实验，并针对于优化的弹性扩容策略设计实验对照组，最后对实验结果进行分析。

5.2.1 实验目的

对本文提出的优化的弹性扩容策略从两方面进行验证，一方面验证该策略是否能将新增 Pod 分配到合适的资源节点，另一方面通过基于改进的蚁群算法（EACO）的资源调度策略与基本调度算法（Bind）的策略、经典蚁群算法（ACO）的策略以及遗传算法（GA）的策略对比，验证其是否能有效降低节点资源消耗成本和集群负载均衡度。

5.2.2 优化弹性扩容策略实验

本实验从数据集中随机获取生成 10 个新增 Pod 的资源需求量和 7 个资源节点的信息，资源节点的 CPU 和内存单价随机生成，通过本文优化的弹性扩容策略进行实验，得到实验结果并进行分析，从而验证该策略的可行性。

新增 Pod 的资源配额信息如表 5.2 所示。CPU 单位为核，1m 表示千分之一的 CPU，内存单位为 M。

表 5.2 新增 Pod 的配额

Table 5.2 Quota of Newly Added Pod

Pod	CPU（m）	内存（M）
1	100	100
2	200	200
3	150	150

4	100	100
5	300	50
6	1000	50
7	500	80
8	250	70
9	300	50
10	700	60

资源节点 Node 的信息如表 5.3 所示。CPU 单价为单位时间内资源节点的 CPU 单价，单位是元/天，内存单价为单位时间内资源节点的内存单价，单位是元/天。

表 5.3 资源节点的信息

Table 5.3 Information of Resource Node

Node	CPU 总量（m）	内存总量（M）	已用 CPU（m）	已用内存（M）	CPU 单价（元/天）	内存单价（元/天）
1	8000	2048	2000	400	1	2
2	8000	2048	1550	500	3	2
3	8000	2048	1000	350	4	5
4	8000	2048	4000	600	2	3
5	8000	2048	3500	300	3	4
6	8000	2048	2350	2000	2	1
7	8000	2048	7910	700	5	3

（1）预选过程

本文默认所有 Pod 端口与资源节点端口均不冲突，新增的所有 Pod 中 CPU 的最小请求量为 100m，内存的最小请求量为 50M，由于编号为 6 的资源节点的剩余内存量不满足以及编号为 7 的资源节点的剩余 CPU 量也不满足，故编号为 1、2、3、4、5 的资源节点成为候选资源节点。候选资源节点如表 5.4 所示。

表 5.4 候选资源节点

Table 5.4 Candidate Resource Nodes

Node	CPU 总量（m）	内存总量（M）	已用 CPU（m）	已用内存（M）	CPU 单价（元/天）	内存单价（元/天）
1	8000	2048	2000	400	1	2
2	8000	2048	1550	500	3	2
3	8000	2048	1000	350	4	5
4	8000	2048	4000	600	2	3
5	8000	2048	3500	300	3	4

（2）候选节点评分情况

根据 3.2.2 节的评分规则，对预选过程选出的候选资源节点进行评分，候选节点评分情况如表 5.5 所示。

表 5.5 候选节点评分情况

Table 5.5 Score of Candidate Nodes

Node	Score1	Score2	Score
1	0.7750	0.4750	1.2500
2	0.7850	0.4750	1.2600
3	0.8500	0.4800	1.3300
4	0.6050	0.3950	1.0000
5	0.7050	0.3550	1.0600

该表中最大的分数 1.3300 作为本文改进蚁群算法中所有分配节点的信息素浓度初始值。

（3）基于改进蚁群算法（EACO）的资源调度策略的实验结果

算法参数设置：根据经验值将蚂蚁数量设置为 10，信息素初始化为表 5.5 中节点的最高分即$\tau_{ij}(0)$= 1.3300，α取 1，β取 5，ρ取 0.5，Q 取 100，e取 0.5，依据用户需求设置总目标函数的权重，本实验 a 取 0.6，b 取 0.4。所有实验结果数据保留小数点后 4 位。

分配方案实验结果图 5.1 所示：

图 5.1 分配方案实验结果图

Figure 5.1 Experimental Results of Distribution Scheme

图 5.1 表明在第 26 次迭代结束后达到算法终止条件，得到总目标函数值为 0.2627，最优的分配方案为 Pod1 分配给 Node3，Pod2 分配给 Node2，Pod3 分配给 Node2，Pod4 分配给 Node3，Pod5 分配给 Node5，Pod6 分配给 Node1，Pod7 分配给 Node4，Pod8 分配给 Node5，Pod9 分配给 Node4，Pod10 分配给 Node1。此分配方案的总目标函数值达到最优。总目标函数值随迭代次数变化的趋势图如图 5.2 所示。

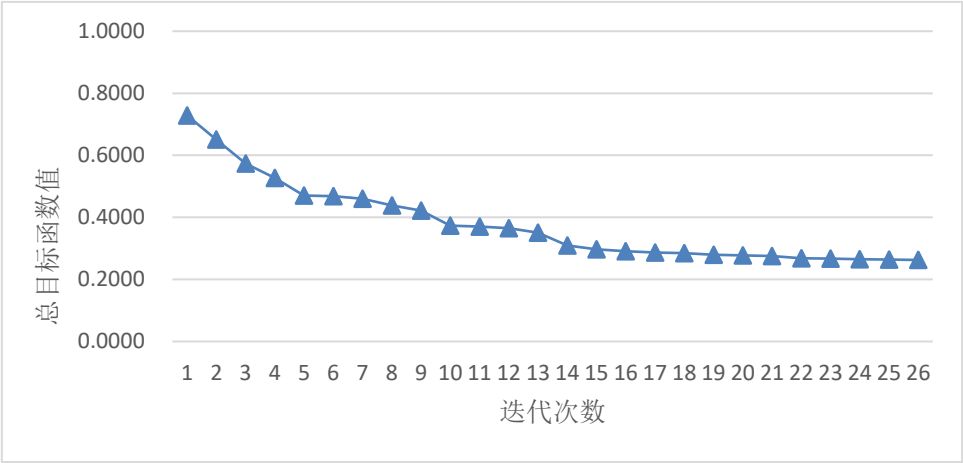


图 5.2 总目标函数值随迭代次数的趋势图

Figure 5.2 Trend of The Total Objective Function Value with The Number of Iterations

图 5.2 表明随着迭代次数的增加，总目标函数值越来越小，在第 16 次迭代时逐渐趋向平稳，在第 26 次迭代后达到 3.3.4 节中的算法终止条件，第 26 次迭代后的总目标函数值为 0.2627，与第 25 次迭代后的总目标函数值 0.2640 相差值不到 0.5%，此时，算法终止，总目标函数值趋于平稳。

5.2.3 对比实验

为了验证本文改进蚁群算法（EACO）的策略的有效性和优化性，将其与基本调度算法（Bind）的策略、经典蚁群算法（ACO）的策略、遗传算法（GA）的策略进行实验对比。通过对这些算法策略在总目标函数值、资源消耗成本目标函数值、集群负载度目标函数值以及算法终止时的迭代次数四个评价指标的对比，分析评价这些算法策略。

此次对比实验的数据均来自上述数据集[48]中，假设有 10 个候选资源节点，且这 10 个资源节点的资源量足够大，不同资源节点的剩余资源量以及单位时间内的资源单价不同，在一定范围随机生成。假设所有 Pod 的端口与资源节点均不冲突，Pod 数量从 10 个到 50 个，资源需求量在一定范围随机生成，依次运用基本调度算法（Bind）的策略、经典蚁群算法（ACO）的策略、遗传算法（GA）的策略以及本文改进的蚁群算法（EACO）的策略进行对比实验，且经典蚁群算法的参数与本文一致，所有对比算法的总目标函数的权重和算法终止条件也与本文一致。

集群中 Pod 和资源节点 Node 的信息如表 5.6 所示。

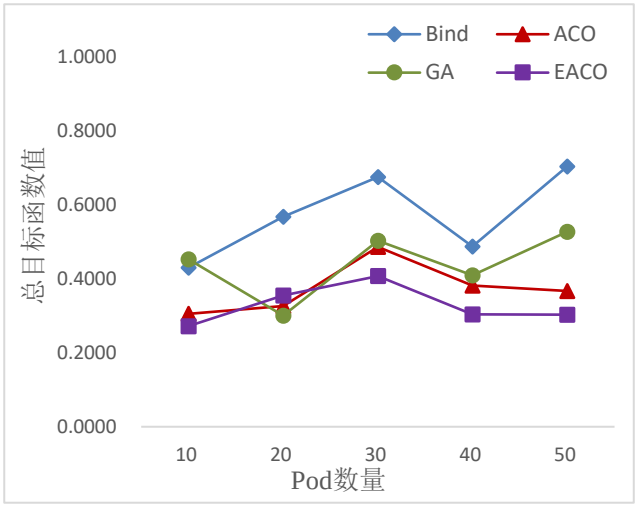
表 5.6 对比实验集群 Pod 和 Node 信息表

Table 5.6 Comparison of Experimental Cluster Pod and Node Information Table

	Pod	Node
数量	10 到 50	10
CPU 量（m）	100 到 1500	200 到 8000
内存（M）	50 到 1500	200 到 2048
CPU 单价（元/天）		1 到 10
内存单价（元/天）		1 到 10

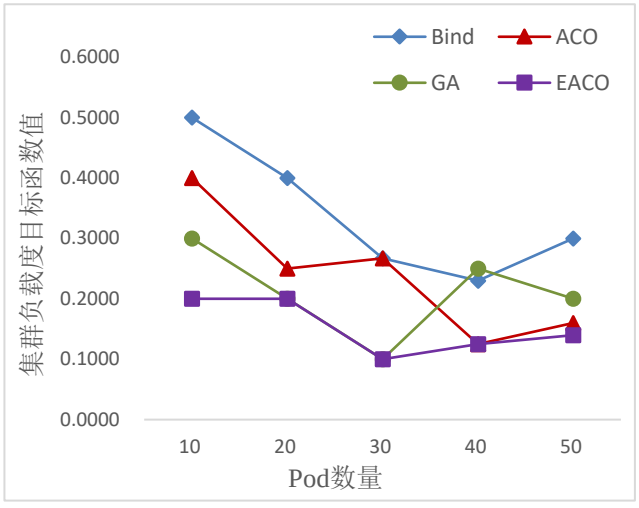
评价指标选取总目标函数值、资源消耗成本目标函数值、集群负载均衡度目标函数值以及算法终止时的迭代次数。

对比实验结果图如下：



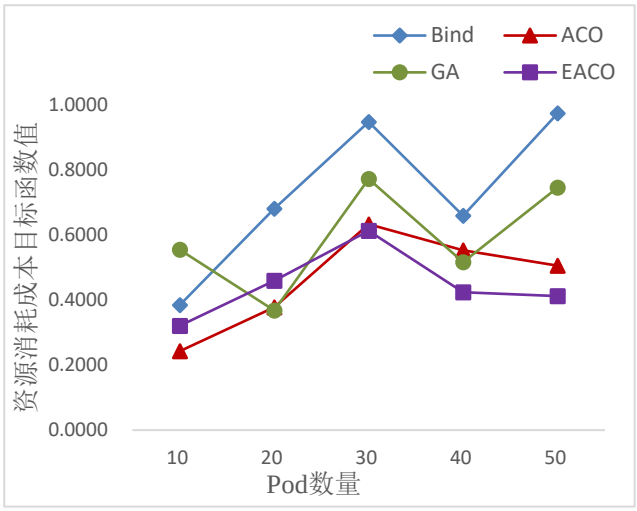
a）总目标函数值对比图

a) Comparison of Total Objective Function Value



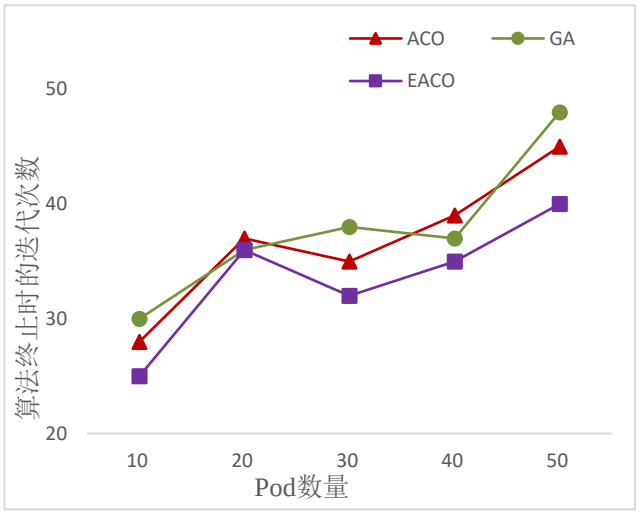
b）集群负载均衡度目标函数值对比图

b) Comparison of Cluster Load Balance Objective Function Value



c）资源消耗成本目标函数值对比图

c) Comparison of Resource Consumption Cost Objective Function Value



d）算法终止时的迭代次数对比图

d) Comparison of The Number of Iterations at The End of The Algorithm

图 5.3 不同 Pod 数量下不同策略 4 种评价指标对比图

Figure 5.3 Comparison of Four Evaluation Indexes of Different Strategies with Different Pod Numbers

在资源节点数量固定的情况下，由图 5.3 的 a）、b）、c）可知，在不同的 Pod 数量下，本文改进的蚁群算法（EACO）、经典蚁群算法（ACO）和遗传算法（GA）的总目标函数值、集群负载均衡度目标函数值和资源消耗成本目标函数值小于基本调度算法（Bind），其原因是基本调度算法在为 Pod 选择节点时未考虑多个 Pod 部署后集群负载均衡和资源消耗成本的因素。由 a）可知，在不同的 Pod 数量下，EACO 的总目标函数值大体上小于 ACO、GA 和 Bind，表明在综合考虑集群负载均衡和资源消耗成本时，本文优化策略的分配方案最优。而

当 Pod 数量为 20 时，EACO 的总目标函数值较大于 ACO 和 GA，可能由于 Pod 数量为 20 时并未找到最优的分配方案。并且从趋势来看，EACO 的总目标函数值波动性不大，说明改进算法的鲁棒性较好。由 b）可知，在不同的 Pod 数量下，EACO 的集群负载均衡度目标函数值小于或等于 ACO、GA 和 Bind。由 c）可知，当 Pod 数量为 10 时，EACO 的资源消耗成本目标函数值大于 ACO，当 Pod 数量为 20 时，EACO 的资源消耗成本目标函数值大于 ACO 和 GA，其原因可能是当 Pod 数量为 10 或 20 时，EACO 陷入局部最优，未找到最优的分配方案。其他情况下，EACO 的资源消耗成本目标函数值均小于 ACO、GA 和 Bind。综合 a）、b）、c）可知，当 Pod 数量为 10 或 20 时，EACO 可能陷入局部最优，并未找到最优的分配方案。由 d）可知，EACO 达到算法终止时的迭代次数小于或等于 ACO 和 GA，表明本文改进的蚁群算法收敛速度更快。

综上所述，综合考虑集群负载均衡和资源消耗成本的情况，在不同的 Pod 数量下，特别当 Pod 数量较大时，本文提出的基于改进蚁群算法（EACO）的资源调度策略相比经典蚁群算法（ACO）、遗传算法（GA）以及基本调度算法（Bind）的策略得到的分配方案更优。

5.3 优化缩容策略实验与结果分析

为了验证本文提出的优化的弹性缩容策略，本文选择云仿真平台 CloudSim3.0 对提出的策略进行实验，并将本文缩容策略与 Kubernetes 现有的缩容策略进行对比，最后对实验结果进行分析。

5.3.1 实验目的

对本文提出的优化的弹性缩容策略从两方面进行验证，一方面对本文提出的基于节点多维资源平衡的缩容策略进行实验验证，另一方面，对本文提出的基于节点资源消耗成本的缩容策略进行实验验证，将两种策略分别与 Kubernetes 现有的缩容策略进行对比，验证本文策略的优势。

5.3.2 节点多维资源平衡对比实验

如果随机设置节点资源量及 Pod 的数量和资源配额，不易观察此缩容策略与 Kubernetes 现有缩容策略的结果对比，为了方便观察，本文的实验设计了节点资源量及 Pod 的数量和配额。

本实验设计 4 个已就绪状态的 Pod，将这 4 个 Pod 部署在集群中同一个资源节点上，资源节点的 CPU 总量为 2000m（CPU 单位为核，1m 表示千分之一的 CPU），内存总量为 2000M（内存单位为 M）。4 个 Pod 的配额及就绪时间如表 5.7 所示。

表 5.7 Pod 配额及就绪时间表

Table 5.7 Pod Quota and Readiness Schedule

Pod	CPU（m）	内存（M）	就绪时间（min）
Pod1	100	100	5
Pod2	200	300	10
Pod3	100	1000	15
Pod4	1000	200	20

假设 Pod1 和 Pod2 是 CPU 和内存平衡型，Pod3 是内存密集型，Pod4 是 CPU 密集型。当需要删除一个 Pod 时，分别使用本文策略与现有策略进行实验，并进行对比。

（1）本文优化的弹性缩容策略

由于 Pod 均为已就绪状态，根据本文优化的缩容策略，使用基于节点资源平衡的缩容策略。分别删除各个 Pod 后，该资源节点的 CPU 利用率、内存利用率、资源平均利用率以及节点资源平衡度（方差值）如表 5.8 所示。

表 5.8 删除各个 Pod 后节点情况

Table 5.8 Node Situation after Deleting Each Pod

Pod	CPU 利用率	内存利用率	资源平均利用率	方差值
Pod1	0.6500	0.7500	0.7000	0.0025
Pod2	0.6000	0.6500	0.6250	0.000625
Pod3	0.6500	0.3000	0.4750	0.030625
Pod4	0.2000	0.7000	0.4500	0.0625

从表 5.8 可知，删除 Pod2 后方差值最小，即删除 Pod2 后该节点的资源最平衡，故根据本文策略选择的待删 Pod 为 Pod2。

（2）Kubernetes 现有的缩容策略

由于 Pod 均为已就绪状态，根据 Kubernetes 现有的缩容策略，判断 4 个 Pod 的就绪时间，根据表 5.7 可知，Pod1 的就绪时间最短，故选定的待删 Pod 为 Pod1。

（3）两种策略对比

本文优化的弹性缩容策略选择节点上的 Pod2 为待删 Pod，Kubernetes 现有的缩容策略选择节点上的 Pod1 为待删 Pod。根据两种策略分别进行缩容后，该资源节点的 CPU 以及内存利用率对比如图 5.4 所示。

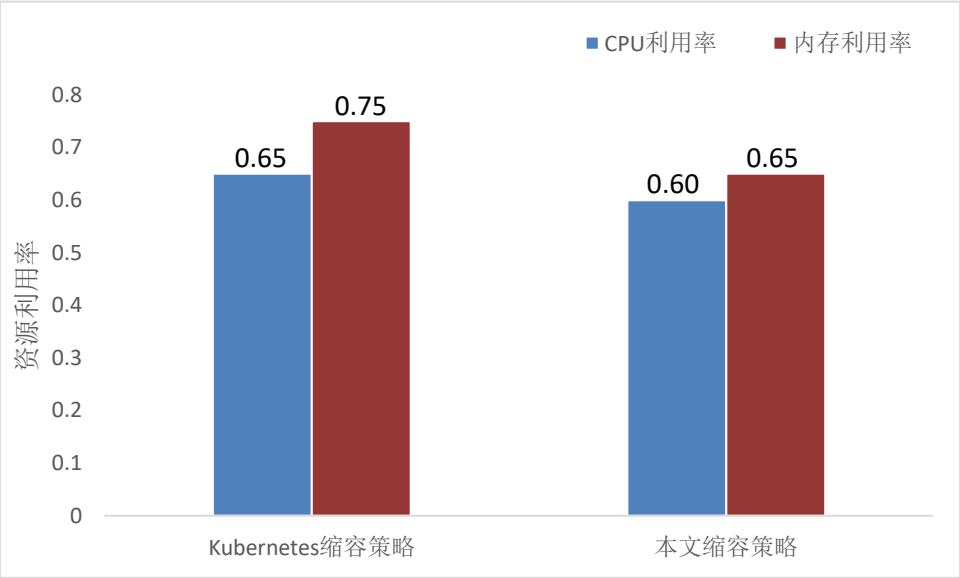


图 5.4 删除 Pod 后节点资源利用率对比图

Figure 5.4 Comparison Chart of Node Resource Utilization after Deleting Pod

从图 5.4 可以看出，根据 Kubernetes 现有的缩容策略进行缩容操作后，资源节点的 CPU 利用率和内存利用率分别为 0.65 和 0.75，而根据本文优化的弹性缩容策略进行缩容操作后，资源节点的 CPU 利用率和内存利用率分别为 0.60 和 0.65，说明通过本文优化的弹性缩容策略删除 Pod 后，节点 CPU 和内存资源的使用比 Kubernetes 现有缩容策略更均衡。

5.3.3 节点资源消耗成本对比实验

由于随机设置节点资源量及 Pod 的数量和资源配额，不易比较此缩容策略与 Kubernetes 现有缩容策略，为了方便比较，实验设计节点资源量及 Pod 的数量和配额。

本实验设计 4 个已就绪状态的 Pod，将这 4 个 Pod 部署在集群中同一个资源节点上，资源节点的 CPU 总量为 2000m（CPU 单位为核，1m 表示千分之一的 CPU），内存总量为 2000M（内存单位为 M），单位时间内的 CPU 单价为 1 元/天，单位时间内的内存单价为 2 元/天。

Pod 信息如表 5.9 所示。

表 5.9 Pod 信息表

Table 5.9 Pod Information Table

Pod	CPU（m）	内存（M）	就绪时间（min）
Pod1	500	400	5
Pod2	400	500	10
Pod3	1000	100	15
Pod4	100	1000	20

假设 Pod1 和 Pod2 是 CPU 和内存平衡型，Pod3 是内存密集型，Pod4 是 CPU 密集型。当需要删除一个 Pod 时，分别使用本文策略与现有策略进行实验，并进行对比。

通过 Kubernetes 现有的缩容策略进行缩容，由于 Pod 均为已就绪状态，由表 5.9 可知，Pod1 的就绪时间最短，故选择节点上的 Pod1 作为待删 Pod；而通过本文优化的弹性缩容策略进行缩容，因为 Pod 均是已就绪状态，先通过节点多维资源平衡的缩容策略进行方差值计算，由于将 Pod1 作为待删 Pod 和将 Pod2 作为待删 Pod 计算的方差值均为最小值 0.000625，再通过节点资源消耗成本的缩容策略进行缩容计算，Pod1 单位时间内所消耗的资源成本为 1300 元/天，Pod2 单位时间内所消耗的资源成本为 1400 元/天，优先删除资源消耗成本较大的 Pod，故选择节点上的 Pod2 作为待删 Pod。

分别通过这两种缩容策略删除 Pod 后，单位时间内节点资源消耗成本对比如图 5.5 所示。

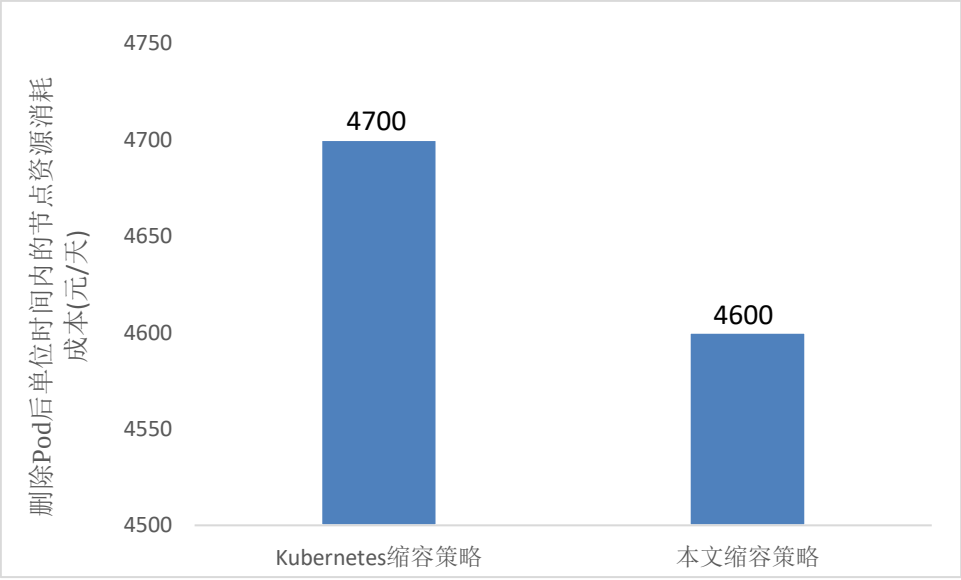


图 5.5 删除 Pod 后单位时间内的节点资源消耗成本

Figure 5.5 Node Resource Consumption Cost Per Unit Time after Deleting Pod

从图 5.5 可知，本文优化的缩容策略删除 Pod 后单位时间内的节点资源消耗成本小于 Kubernetes 现有的缩容策略删除 Pod 后单位时间内的节点资源消耗成本。故通过本文优化的缩容策略进行缩容操作后，降低了节点单位时间内的资源消耗成本。

5.4 本章小结

本章首先介绍了实验的环境以及云计算仿真平台 CloudSim，然后分别对本文中的优化的弹性扩容与缩容策略进行实验。对于本文优化的弹性扩容策略，与基本调度算法（Bind）的策略、经典蚁群算法（ACO）的策略以及遗传算法（GA）的策略进行对比实验，实验证明本文优化的弹性扩容策略能有效降低资源消耗成本，使集群负载更加均衡。对于本文优化的弹性缩容策略，与 Kubernetes 现有的缩容策略进行对比实验，实验证明本文优化的弹性缩容策略能提升节点资源的平衡性，降低节点单位时间内的资源消耗成本。

第六章 总结和展望

6.1 总结

本文主要分析了以 Docker 容器为基础的容器编排系统 Kubernetes 现有的弹性策略存在的问题，并针对于现有弹性策略的不足之处，分别从扩容阶段和缩容阶段两方面，提出基于改进蚁群算法的资源调度策略以及基于节点多维资源平衡和节点资源消耗成本的缩容策略，对 Kubernetes 现有弹性策略进行优化。

Kubernetes 现有弹性策略的不足之处有：

（1）扩容阶段：现有扩容策略为新增 Pod 选择节点时只考虑当前性能最优的节点，即优选过程中得分最高的节点，而未考虑多个 Pod 部署后集群负载均衡和资源消耗成本的问题。

（2）缩容阶段：现有缩容策略在选择待删 Pod 时只考虑 Pod 的状态优先级，而未考虑删除 Pod 后节点资源平衡和节点资源消耗成本的问题。

对此，本文提出了优化的弹性策略以解决上述问题，以下为本文主要的研究内容：

（1）扩容阶段：提出了一种基于改进蚁群算法的资源调度策略，对现有的扩容策略进行优化。

a. 改进蚁群算法方面，建立集群负载度和节点资源消耗成本的目标函数；针对于经典蚁群算法存在的收敛速度慢、易陷入局部最优的问题，分别通过精英蚂蚁和轮盘赌的方式解决；将 Pod 和资源节点的一次分配作为蚂蚁爬行节点；将优选评分过程中的最高分值作为分配节点的初始信息素浓度。

b. 整体优化的扩容策略针对于现有扩容策略的不足，在为新增 Pod 进行调度时，充分考虑集群的负载均衡和节点的资源消耗成本。

c. 实现了整体优化的扩容策略，同时通过与基本调度算法的策略、经典蚁群算法的策略和遗传算法的策略进行实验比较，验证优化扩容策略的可行性与优化性。

（2）缩容阶段：提出了节点多维资源平衡和节点资源消耗成本的缩容策略，并将这两种策略同现有的缩容策略进行整合优化。

a. 提出了一种节点多维资源平衡的缩容策略，解决删除 Pod 后可能会造成的节点各维度资源使用不平衡，从而产生资源碎片的问题。

b. 提出了一种节点资源消耗成本的缩容策略，考虑删除 Pod 后的节点资源消耗成本。

c. 实现了整体优化的缩容策略，同时通过与现有缩容策略进行实验比较，验证优化缩容策略的可行性与优化性。

研究结果表明，本文优化的弹性扩容策略相比于基本调度算法(Bind)的策略、基于经典蚁群算法(ACO)的策略和基于遗传算法(GA)的策略可以降低节点资源消耗成本，使集群负载更加均衡。本文优化的弹性缩容策略相比现有的缩容策略，能够降低节点资源消耗成本，减少节点资源碎片的产生，使节点资源使用更加平衡。

6.2 展望

本文对 Kubernetes 现有的弹性策略进行研究分析，并针对于现有弹性策略存在的问题，分别从扩容阶段和缩容阶段两方面进行优化。但是本文的研究仍有不足之处，需要进一步完善。

（1）本文研究的优化弹性策略均在云仿真平台上实现，未实现在 Kubernetes 的集群中，进一步研究可以将本文策略在 Kubernetes 集群中实现。

（2）本文改进的蚁群算法存在不足之处，算法的各参数依据经验值设定，进一步研究可以对参数的设定方面进行优化。

（3）本文提出的优化缩容策略考虑了删除 Pod 后对资源节点的影响，但是未考虑删除 Pod 后对集群负载等方面的影响，进一步可以在删除 Pod 后对集群的影响方面进行研究。

参考文献

[1] 易升海,彭江强,卿勇军,等.浅析 Docker 容器技术的发展前景[J].电信工程技术与标准化,2018,31(06):88-91.

[2] Ferreira A P, Sinnott R. A performance evaluation of containers running on managed kubernetes services[C]. 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE Computer Society, 2019: 199-208.

[3] Marathe N, Gandhi A, Shah J M. Docker swarm and kubernetes in cloud computing environment[C]. 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI). IEEE, 2019: 179-184.

[4] 徐凯.基于 Kubernetes 的伸缩性分布式资源调度器的设计与实现[D].西安电子科技大学,2017.

[5] Ungureanu O M, Vlădeanu C, Kooij R. Kubernetes cluster optimization using hybrid shared-state scheduling framework[C]. Proceedings of the 3rd International Conference on Future Networks and Distributed Systems. 2019: 1-12.

[6] 台慧敏.基于 Kubernetes-on-EGO 的两级资源调度器的设计与实现[D].西安电子科技大学,2017.

[7] Chang C C, Yang S R, Yeh E H, et al. A kubernetes-based monitoring platform for dynamic cloud resource provisioning[C]. GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, 2017: 1-6.

[8] 左灿,刘晓洁.一种改进的 Kubernetes 动态资源调度方法[J].数据通信,2019(02):50-54.

[9] 张启辉,未来.一种改进的 Kubernetes 动态缩容模型[J].数据通信,2019(05):38-42.

[10] Medel V, Tolón C, Arronategui U, et al. Client-side scheduling based on application characterization on kubernetes[C]. International Conference on the Economics of Grids, Clouds, Systems, and Services. Springer, Cham, 2017: 162-176.

[11] 常旭征,焦文彬.Kubernetes 资源调度算法的改进与实现[J].计算机系统应用,2020,29(07):256-259.

[12] 何龙,刘晓洁.一种基于应用历史记录的 Kubernetes 调度算法[J].数据通信,2019(03):33-36.

[13]李翔,张帆,李治江.基于微服务特性的容器自动扩缩容研究[J].武汉大学学报(工学版),2019,52(07):651-658.

[14]田倬璟,黄震春,张益农.云计算环境任务调度方法研究综述[J].计算机工程与应用,2021,57(02):1-11.

[15]Yiqiu F, Xia X, Junwei G. Cloud computing task scheduling algorithm based on improved genetic algorithm[C]. 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). IEEE, 2019: 852-856.

[16]Guerrero C, Lera I, Juiz C. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture[J]. Journal of Grid Computing, 2018, 16(1): 113-135.

[17]张松霖.容器云中基于改进遗传算法的资源分配策略[J].计算机测量与控制,2021,29(01):168-173

[18]Liu S, Yin Y. Task scheduling in Cloud Computing Based on Improved Discrete Particle Swarm Optimization[C]. 2019 2nd International Conference on Information Systems and Computer Aided Education (ICISCAE). IEEE, 2019: 594-597.

[19]Wu D. Cloud computing task scheduling policy based on improved particle swarm optimization[C]. 2018 International Conference on Virtual Reality and Intelligent Systems (ICVRIS). IEEE, 2018: 99-101.

[20]Cai Q, Shan D, Zhao W. Resource scheduling in cloud computer based on improved particle swarm optimization algorithm[J]. J. Liaoning Tech. Univ.(Natural Science), 2016, 5: 93-96.

[21]李飞,彭宇楠,苏亚松.基于改进蜂群算法的虚拟云资源调度负载优化策略研究[J].计算机产品与流通,2019(11):119-120.

[22]匡珍春,谢仕义.基于猫群优化算法的云计算虚拟机资源负载均衡调度[J].吉林大学学报(理学版),2016,54(05):1117-1122.

[23]Kaewkasi C, Chuenmuneewong K. Improvement of container scheduling for docker using ant colony optimization[C]. 2017 9th international conference on knowledge and smart technology (KST). IEEE, 2017: 254-259.

[24] Wei-guo Z, Xi-lin M, Jin-zhong Z. Research on Kubernetes' Resource Scheduling Scheme[C]. Proceedings of the 8th International Conference on Communication and Network Security. 2018: 144-148.

[25] Yang Z, Liu M, Xiu J, et al. Study on cloud resource allocation strategy based on particle swarm ant colony optimization algorithm[C]. 2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems. IEEE, 2012, 1: 488-491.

[26] Liu L, Luo T, Du Y. A new task scheduling strategy based on improved ant colony algorithm in IaaS layer[C]. 2019 International Conference on Computer, Information and Telecommunication Systems (CITS). IEEE, 2019: 1-5.

[27] He Z, Dong J, Li Z, et al. Research on Task Scheduling Strategy Optimization Based onACO in Cloud Computing Environment[C]. 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC). IEEE, 2020: 1615-1619.

[28] 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.

[29] Pahl C, Brogi A, Soldani J, et al. Cloud container technologies: a state-of-the-art review[J]. IEEE Transactions on Cloud Computing, 2017, 7(3): 677-692.

[30] Shukur H, Zeebaree S, Zebari R, et al. Cloud computing virtualization of resources allocation for distributed systems[J]. Journal of Applied Science and Technology Trends, 2020, 1(3): 98-105.

[31] Li Z, Kihl M, Lu Q, et al. Performance overhead comparison between hypervisor and container based virtualization[C]. 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA). IEEE, 2017: 955-962.

[32] Kandan R, Khalid M F, Ismail B I, et al. A Generic Log Analyzer for automated troubleshooting in container orchestration system[C]. 2020 8th International Conference on Information Technology and Multimedia (ICIMU). IEEE, 2020: 267-270.

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

[34] 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.

[35] Zhang Y, Wang H, Filkov V. A clustering-based approach for mining dockerfile evolutionary trajectories[J]. Science China Information Sciences, 2019, 62(1): 1-3.

[36] Dikaleh S, Sheikh O, Felix C. Introduction to kubernetes[C]. Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering. 2017: 310-310.

[37] Shah J, Dubaria D. Building modern clouds: using docker, kubernetes & Google cloud platform[C]. 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC). IEEE, 2019: 0184-0189.

[38] Huang W, Zhang W, Zhang D, et al. Elastic spatial query processing in openstack cloud computing environment for time-constraint data analysis[J]. ISPRS International Journal of Geo-Information, 2017, 6(3): 84.

[39] Medel V, Tolosana-Calasanz R, Bañares J Á, et al. Characterising resource management performance in Kubernetes[J]. Computers & Electrical Engineering, 2018, 68: 286-297.

[40] Dorigo M, Maniezzo V, Colorni A. Ant system: optimization by a colony of cooperating agents[J]. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 1996, 26(1): 29-41.

[41] Shang J, Wang X, Wu X, et al. A review of ant colony optimization based methods for detecting epistatic interactions[J]. IEEE access, 2019, 7: 13497-13509.

[42] Xie X, Xu K, Wang X. Cloud computing resource scheduling based on improved differential evolution ant colony algorithm[C]. Proceedings of the 2019 International Conference on Data Mining and Machine Learning. 2019: 171-177.

[43] Chen H, Tan G, Qian G, et al. Ant colony optimization with tabu table to solve TSP problem[C]. 2018 37th Chinese Control Conference (CCC). IEEE, 2018: 2523-2527.

[44] 张朝锋.K8s 预选策略和优选函数简介[EB/OL]. https://www.cnblogs.com/wn1m/p/11290733.html, 2019-08-02.

[45] Chatterjee A, Kim E, Reza H. Adaptive dynamic probabilistic elitist ant colony optimization in traveling salesman problem[J]. SN Computer Science, 2020, 1(2): 1-8.

[46] Shasha D. Roulette Angel[J]. Communications of the ACM, 2021, 64(4): 176-ff.

[47] Lloyd H, Amos M. Analysis of independent roulette selection in parallel ant colony optimization[C]. Proceedings of the Genetic and Evolutionary Computation Conference. 2017: 19-26.

[48] Weave Scope[EB/OL]. https://cloud.weave.works/demo/

[49] Piraghaj S F, Dastjerdi A V, Calheiros R N, et al. ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers[J]. Software: Practice and Experience, 2017, 47(4): 505-521.

[50] Xu H, Wang G, Luo L, et al. The Design of Reliability Simulation of Cloud System in the Cloudsim[C]. 2018 15th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP). IEEE, 2018: 215-219.

致 谢

时光飞逝，转眼间三年的研究生生活即将到达尾声，还记得三年前为了读研日夜奋战的自己，终梦想成真。在这三年里，我认识了许多的老师和朋友，学习到了许多的专业技能，收获颇丰，在此感谢所有帮助过我的人。

首先，我要真挚的感谢我的导师李华老师。从入学到即将毕业，李老师给予我很大的帮助，当遇到科研难题时，李老师会鼓励我并提出意见，帮助我走出困境；当我产生一些新的想法时，李老师会给予我肯定，鼓励我朝着自己的想法努力。从论文的开题到现在，李老师都给我许多的建议，特别是在我毫无头绪的时候，让我在论文的完成路上少走了很多弯路。李老师认真的工作作风，严谨的工作态度使得我在完成论文的过程中更加顺利，李老师的这种科研态度是我应该学习的地方。在此特别感谢李老师的帮助！

其次，我要感谢同组的王显荣老师，还有刘振宇博士师兄和郑冰博士师姐，他们在我做科研项目和完成论文的过程中提供了许多的帮助，很感谢他们对我的指导。我也要感谢我的师兄师姐们刘麒、杨珍、刘亚、连超、张筵雍、赵文欣、范美婷、李尚松对我的关心和照顾，还有感谢我的室友兼同门李晓迪、韩家茂、刘博炜对我生活上的照顾，感谢和我一起毕业的同门崔雅君和宋丹丹等对我的帮助，当然还要感谢我的师弟师妹们楼轩宇、王加峰、兰天翔、查娜、张文钊、许彤、张忱、李娜等在我论文撰写时对我的帮助。

然后我要感谢我的父母对我多年读书的支持和鼓励。正是因为他们的支持，我才能心无旁骛的搞科研，他们的理解与支持是我发奋图强的动力。从幼儿园到研究生，这么多年来他们对我的爱，对我的关心，我都默默记在心里，我也即将踏上社会，我要努力工作，报答他们。在此，感谢我的父母和妹妹，祝愿他们身体健康！

最后，特别感谢评阅我的毕业论文的各位老师们，谢谢您们的宝贵意见！

参与项目

[1] 2020-2022，内蒙古自治区科技计划项目《多元异构健康医疗数据融合平台研究及应用》，批准号 2020GG0186。

[2] 2019.01-2022.12，国家自然科学基金项目《多源数据驱动的网络服务功能链和业务服务系统的建模及其弹性评测》，批准号 61862047。

[3] 2018-2020，内蒙古自治区科技计划项目《大数据分析和 SDN 融合驱动的行业云应用支撑平台关键技术及评测》，批准号 201802028。

[4] 2018.01-2019.01，《呼和浩特市新城区社区工作办公室社区居家养老服务体系建设项目信息平台软件测试》。