中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

基于容器化多终端协同服务技术研究

作者姓名：赵然

指导教师：倪宏 研究员 中国科学院声学研究所

学位类别：工学博士

学科专业：信号与信息处理

培养单位：中国科学院声学研究所

2019 年 6 月

LaTeX Thesis Template
of
The University of Chinese Academy of Sciences $\pi\pi^{\pi}$

A thesis submitted to the

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Engineering

in Signal and Information Processing

By

Zhao Ran

Supervisor: Professor Ni Hong

Institute of Acoustics, Chinese Academy of Sciences

June, 2019

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关保存和使用学位论文的规定，即中国科学院大学有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

本文是中国科学院大学学位论文模板 ucasthesis 的使用说明文档。主要内容为介绍 LaTeX 文档类 ucasthesis 的用法，以及如何使用 LaTeX 快速高效地撰写学位论文。

关键词：中国科学院大学，学位论文，LaTeX 模板

Abstract

This paper is a help documentation for the LaTeX class ucasthesis, which is a thesis template for the University of Chinese Academy of Sciences. The main content is about how to use the ucasthesis, as well as how to write thesis efficiently by using LaTeX.

Keywords: University of Chinese Academy of Sciences (UCAS), Thesis, LaTeX Template

目 录

图形列表

表格列表

符号列表

字符

Symbol	Description	Unit
R	the gas constant	$m^2 \cdot s^{-2} \cdot K^{-1}$
C_v	specific heat capacity at constant volume	$m^2 \cdot s^{-2} \cdot K^{-1}$
C_p	specific heat capacity at constant pressure	$m^2 \cdot s^{-2} \cdot K^{-1}$
E	specific total energy	$m^2 \cdot s^{-2}$
e	specific internal energy	$m^2 \cdot s^{-2}$
h_T	specific total enthalpy	$m^2 \cdot s^{-2}$
h	specific enthalpy	$m^2 \cdot s^{-2}$
k	thermal conductivity	$kg \cdot m \cdot s^{-3} \cdot K^{-1}$
S_{ij}	deviatoric stress tensor	$kg \cdot m^{-1} \cdot s^{-2}$
τ_{ij}	viscous stress tensor	$kg \cdot m^{-1} \cdot s^{-2}$
δ_{ij}	Kronecker tensor	1
I_{ij}	identity tensor	1

算子

Symbol	Description
Δ	difference
∇	gradient operator
$\delta^{\pm}$	upwind-biased interpolation scheme

缩写

CFD	Computational Fluid Dynamics
CFL	Courant-Friedrichs-Lewy
EOS	Equation of State
JWL	Jones-Wilkins-Lee
WENO	Weighted Essentially Non-oscillatory
ZND	Zel'dovich-von Neumann-Doering

第 1 章　绪论

随着信息技术的快速发展，人工智能、物联网等技术与智能终端的结合得越来越紧密，在智能终端上除了为用户提供基础服务以外还要提供更智能化的服务。因此，智能终端上要承担的计算任务越来越重。但目前智能终端上所拥有的计算、存储等资源并没有跟上对于智能终端计算能力的需求。与此同时，家庭、办公楼等智能终端环境中还存在着很多相对空闲、计算资源没有得到充分利用的计算设备。越来越重的计算任务与计算资源分配不均的矛盾日益凸显，为了解决这样的矛盾，在整体资源有限的情况下使资源利用达到最优化，我们想到可以使用多个智能终端协同提供服务，提供一种能够整合多终端资源、合理利用多终端资源的服务技术。

1.1　研究背景

终端设备（Terminal Device）通常是指网络中能够对进行信息输入输出以及信息处理的节点 [1]。在本研究中，智能终端设备主要指分布在用户身边的、具有一定计算能力的、能够进行网络连接和通信的、能够为用户提供计算服务的用户设备。随着信息技术的发展，智能终端设备的计算能力越来越强，设备规模越来越小，呈现智能化、轻量化、网络化的趋势。目前常见的智能终端设备包括智能手机、平板电脑、机顶盒、路由器、可穿戴设备、智能车、智能办公环境、智能家居环境等等。

在过去的几年中，智能终端设备通过提供按需的、弹性的、实时的、方便用户随时获取的移动服务，改变了传统计算服务的格局 [2]。智能手机、平板电脑、机顶盒、路由器等智能终端设备，正在逐渐成为人们现代数字化生活的重要组成部分。随着 5G 技术的日渐成熟以及物联网技术、边缘计算技术等技术的快速发展，万物互联的时代即将到来。根据中国信息通信研究院（工业和信息化部电信研究院）2016 年的《物联网白皮书》预计，全球范围内的可穿戴设备、智能家电、自动驾驶汽车、智能机器人等新的智能设备接入物联网的数量将会达到数以百亿级，预计到 2020 年，全球联网设备数量将达到 260 亿个，物联网市场规模达到 1.9 万亿美元，全世界智慧城市总投资将达到 1200 亿美元 [3]。在未来的一段时间里，不仅智能终端设备的数量和规模大大增加，智能终端设备承担的计算任务也会越来越多，产生的数据量也会呈现爆炸式的增长。预计 2020 年，全球

联网设备产生的总数据量将会达到 44ZB[3]，智能终端设备的数据价值在不远的未来将会推动科技革命和工业革命的发展，并更进一步推动人们群众的生活方式向更智能化、数字化发展。

各种各样的新的信息技术在智能终端设备这样一个平台上的快速发展，在带来美好的技术变革的同时，也对智能终端设备的计算和存储能力提出了更高的要求。根据《思科全球云指数预测白皮书》的估计，2019 年在网络边缘上的服务器、终端设备上所进行处理和计算的数据将会占全球物联网所产生的数据总量的 45%[4]。为了增强智能终端设备的计算和存储能力，提升智能终端服务质量，人们使用了云计算技术。云计算技术通过虚拟化技术，将云端的物理实体资源，包括计算、存储、内存、网络等资源进行虚拟化，形成资源池，可以让用户终端根据具体需求和费用考虑，灵活使用。云计算中的云端服务器拥有相对“无限”资源，通过网络提供给用户，帮助具有有限资源的智能终端设备执行需要更加强大的计算资源的复杂计算 [5]。云计算是信息存储和处理的工业化过程，是信息服务业的基础设施和服务平台 [6]。在不远的未来，云计算资源可以像水、电一样，通过水管、电线和网线进行传输，只需要打开开关就可以按需获取、按量计费，成为人们日常生活中和工业生产生活中可以随时随地获取的一种基础资源 [7]。

尽管云计算能够大大增加智能终端设备的可用资源总量，但是面对呈现爆炸式增长趋势的智能终端数量以及智能终端待处理数据量，云计算的集中式处理模式仍然存着很多不足之处。云计算模式中的云数据处理中心（Data Center）距离用户的智能终端设备较远，传输时延较长，对于一些实时性较强的用户智能终端服务并不适合。海量的智能终端数据在网络上的传输也会造成带宽负载的浪费，大量的用户智能终端设备向云数据处理中心请求服务会造成网络的延迟和拥塞 [8]。另一方面，用户智能终端设备上产生的数据大多包含有用户个人隐私信息，在同云数据处理中心的数据通信过程中，存着极高的信息安全风险。

智能终端产生的海量数据，使得网络的传输能力和用户隐私安全问题成为限制万物互联时代快速发展的瓶颈，因此直接开发终端自身的潜能 [9]，利用智能终端自身的计算处理能力和智能终端之间短距离的局域网传输能力来对智能终端数据进行处理并提高智能终端服务质量成为一种可行的解决思路。在物联网中，智能设备除了作为数据源的作用外，还能够提供计算和存储功能，但是由于智能终端设备通常是专有目的的，可能只会使用几个小时，其他时间就空闲下来了 [10]。由于用户智能终端设备总体数量庞大，因此还存在着大量的空闲资源

没有得到充分利用，智能终端设备自身还有着很大的潜能可以开发利用。相比使用云计算技术中的云端资源来提供计算服务，使用智能终端设备来提供计算服务拥有很多优势。智能终端设备本身距离用户服务请求发起端更近，网络状态更好，传输时延短，服务响应更快，实时性更好。智能终端设备之间进行的局域网、短距离、高速度的传输，也能够大大减少对于公共网络带宽资源的占用，减少资源浪费。多个智能终端设备协同提供计算服务，用户的隐私数据只在内部传输，而不需要发送到很远的云端数据中心，这样面临的信息安全风险也大大降低。而且相比云计算的集中式模式，使用多智能终端协同提供计算服务能够，将智能终端大量的空闲资源利用起来，大大提高智能终端的资源利用率，减少智能终端资源浪费，而且其部署成本也更加低廉，可以直接利用现有智能终端设备，而不需要购买太多价格较为昂贵的独立服务器等设备，更不需要建立云计算模式中的云数据处理中心，部署成本较为低廉。

为了将智能终端设备上的资源加以管理和利用，我们参考云计算模式，引入了虚拟化技术。虚拟化技术能够将计算机或终端的资源如 CPU、内存、存储等抽象、整合起来，打破实体资源的整体性，形成可以按需分配的资源池，让用户能够以更高效合理的方式来对智能终端资源进行管理和利用。传统的虚拟机（Virtual Machine，简称为 VM）通常使用的是完全虚拟化技术，虚拟机启动时间非常长，也会消耗大量的额外资源来维持虚拟化功能。以 Docker 为代表的轻量级虚拟化技术近几年发展迅速，成为行业内最流行的解决方案。轻量级虚拟化技术利用基于 Linux 内核的 LXC 技术，可以以容器的形式将智能终端底层的物理实体资源按需封装起来，为用户提供服务，大大降低了对于智能终端上资源的管理和利用难度。可以说容器技术的快速发展，也是促成多智能终端协同服务技术成型的一个重要因素。

为了利用智能终端设备的潜力，还需要解决设备和应用的管理整合问题 [11]。本文研究了基于容器化的多智能终端协同服务技术，完成了一个多智能终端协同服务系统的架构设计，并着重研究了其中的几个模块，用来解决多智能终端协同技术中的设备、资源和应用的管理问题，提高智能终端设备的资源利用率，提高终端用户服务体验。

1.2 研究意义

随着信息技术的快速发展，未来智能终端设备可以成为一个获取互联网资源的入口，轻量级的自下而上的系统。

本研究针对智能终端计算任务越来越重与终端计算资源分布不均衡的问题，引入容器技术，对于多终端上的资源进行聚合管理，并通过多智能终端协同技术，合理分配利用终端空闲资源，提高智能终端的服务效率以及智能终端的资源利用率，为未来智能终端为用户提供日常生活服务以及与边缘计算、人工智能、物联网等技术的结合打下良好基础，具有重要的研究意义和应用价值。

基于以上现状，依托于中国科学院战略性先导科技专项课题，开展了基于容器化多终端协同服务技术研究，通过对基于容器化多终端协同服务系统架构的设计与研究，提高系统对于智能终端设备和终端服务应用的管理能力，并通过对基于容器的透明计算迁移技术的研究，提高系统对于智能终端计算资源的利用能力，通过研究多终端任务调度问题，提高了系统对于智能终端资源的利用率，最后通过研究基于预测的容器弹性服务问题，促进提高资源利用率和降低用户服务请求响应等待时间之间的平衡。本文的几个研究点相辅相成，最终目的是利用基于容器化的多终端协同服务技术，提高智能终端设备的资源利用率并提高终端服务的用户体验。

1.3 研究内容

为了解决终端计算能力跟不上以及终端空闲资源浪费的问题，我们研究多终端协同服务技术。引入了多终端协同技术和容器技术以后，整个系统会存在很多单一终端服务不会遇到的问题，例如多终端上的空闲资源如何使用、如何对多终端设备及其所提供的服务进行管理、如何提高智能终端设备的资源利用率、如何提高用户体验等待。本研究中主要通过研究四个方面来解决这些问题，包括基于容器化多终端服务系统架构设计、基于容器化的透明计算迁移技术、资源受限终端任务调度策略、基于预测的容器弹性服务策略。

本文针对上面提出的几个问题，首先介绍了边缘计算技术、容器虚拟化技术、计算迁移技术、任务调度算法、预测算法等相关技术和算法的研究现状，分析其目前还存在的问题。本文结合容器虚拟化技术和多终端协同服务技术，设计了基于容器化的多终端协同服务系统，并主要研究系统中的基于容器的多终端透明计算迁移技术、多终端协同服务任务调度算法优化以及基于预测的终端弹性服务技术。

本文具体研究内容如下：

1. 随着计算机技术的快速发展和边缘计算技术的逐渐成熟，位于网络边缘的用户终端设备在用户的数字化生活中正扮演着越来越重要的角色，其定位逐

渐从单一的用户服务发起者向用户服务的提供者转变。这会使用户终端承担越来越多的计算任务，这也对终端设备的服务质量和计算能力提出了越来越高的要求。但另一方面，边缘终端设备上还存在着大量没有得到充分利用的空闲资源。终端对计算能力要求越来越高与终端设备资源利用率不高之间存在着可以提升的空间，使用多终端协同服务技术来提升终端整体可用计算能力、提升终端整体资源利用率称为一种可行的方法。但是多终端协同服务技术还是会存着不少问题，如：终端资源管理、终端服务管理、多终端节点管理、节点组网等问题。为了解决这些问题，本文设计了基于容器化的多终端协同服务系统，引入容器虚拟化技术对多终端资源进行虚拟化，形成资源池，可以为上层终端服务按需使用；还参考微服务架构，提出多终端协同服务系统架构，可以进行终端服务管理；另外还提出了去中心化的自组织网络结构，进行多终端节点管理和节点组网。

2. 相比云端协同计算，边缘终端设备距离用户更近，而且拥有很多空余资源没有得到充分利用，如何将这部分距离用户更近、成本更低的资源组织利用起来为用户设备提供计算迁移服务也成为了多终端协同服务技术中的一个值得研究的问题。为了将多终端上空闲的资源整合利用起来，为终端用户提供服务，提升用户体验, 我们以 HTML5 中的 Web Worker 方法为例，研究 Web 应用透明计算迁移到以容器形式部署在边缘设备上的服务端，设计并实现了一种基于容器的 Web Worker 透明边缘计算迁移方法。一系列实验结果证明，所提出的方法能够将用户终端周围设备的空闲资源利用起来，为用户提供计算迁移服务，减少计算总执行时间，提高终端整体资源利用率，有效提高用户体验。

3. 为了在终端资源有限且终端资源异构性极强的情况下，合理调度任务请求到更合适的执行节点上，使得任务总体开销更小，减少响应时间，提高用户体验，本研究基于一种群体智能的演进式优化算法————蝗虫优化算法，提出一种带有随机跳出机制的动态权重蝗虫优化算法来解决优化问题。在该算法中，我们在原有蝗虫优化算法的基础上，增加了基于完全随机跳出因素的跳出机制，来提高算法的跳出局部最优的能力。另外，我们还根据搜索阶段的不同，使用动态的权重参数来代替原算法中的线性递减搜索单元权重参数，帮助算法在不同的搜索阶段获得更大的迭代收益。经过一系列测试函数的实验验证，我们提出的带有随机跳出机制的动态权重蝗虫优化算法能够有效提高优化算法的搜索精度及收敛速度。在此基础上，我们进一步提出一种改进蝗虫优化算法，并将其应用于解决多终端任务调度问题。在该算法中，我们引入变型的 sigmoid 函数作为非线性舒适区调节参数，增强算法的搜索能力。同时，我们提出基于 Levy 飞行的局

部搜索机制，让搜索单元在局部拥有一定的“搜索视觉”，提高算法的局部搜索能力。另外，我们还使用基于线性递减参数的随机跳出策略，增强算法跳出局部最优能力，并将成功跳出的结果影响力维持若干次迭代。经过一系列测试函数和标准测试集的实验，实验结果表明我们提出的改进蝗虫优化算法能够有效提高优化算法的搜索精度、稳定性、搜索到更优解的可能性及收敛速度。最后我们提出了终端上任务调度问题的数学模型，并将提出的改进蝗虫优化算法应用到该问题的求解过程中。实验结果证明，我们提出的改进蝗虫算法在求解终端上任务调度问题时能够得到很好的效果。

4. 对于终端服务系统来说，如果不进行终端服务预部署，则当用户服务请求到达终端服务系统的时候现场启动基于 Docker 容器的终端服务端程序还是会消耗一定时间，尽管相比于传统的 VM 虚拟机，Docker 容器技术的启动时间已经非常短了，但这个启动时间相比于用户请求等待响应时间仍然较长，不能接受。但是如果提前进行终端服务预部署，虽然可以达到缩短用户请求等待响应时间的目的，但是对于资源有限的终端来说，等待用户服务请求到达的过程会消耗大量额外的资源，同样是不能接受的。这就形成了一个矛盾的问题。为了解决是否进行预部署的问题，本研究提出了一种基于预测的容器弹性服务策略，利用改进的卡尔曼滤波算法，对未来一小段时间的用户请求流量进行预测，并根据预测结果对终端服务的容器规模进行适当调整，以达到在不增加用户请求响应等待时间、不降低用户体验的情况下对终端资源利用更合理的目的。经过一系列的仿真实验，证明所提出的基于预测的容器弹性服务策略能够有效根据用户请求流量的变化趋势动态调整终端服务规模，合理利用终端资源。

1.4 本文内容安排

本文一共分为七个章节，针对基于容器化的多终端协同服务技术进行研究，组织结构如图1.1所示，各章节内容概述如下：

第 1 章介绍了基于容器化的多终端协同服务技术的发展背景以及研究意义、本文的研究内容和本文内容安排。

第 2 章介绍了基于容器化的多终端协同服务技术的相关技术的研究现状，包括边缘计算技术、容器虚拟化技术、计算迁移技术、任务调度算法、预测算法等相关技术和算法。

第 3 章结合容器虚拟化技术和微服务架构，提出了基于容器化的多终端协同服务系统的系统架构设计及构建方法，解决了多终端节点管理、终端资源管理、

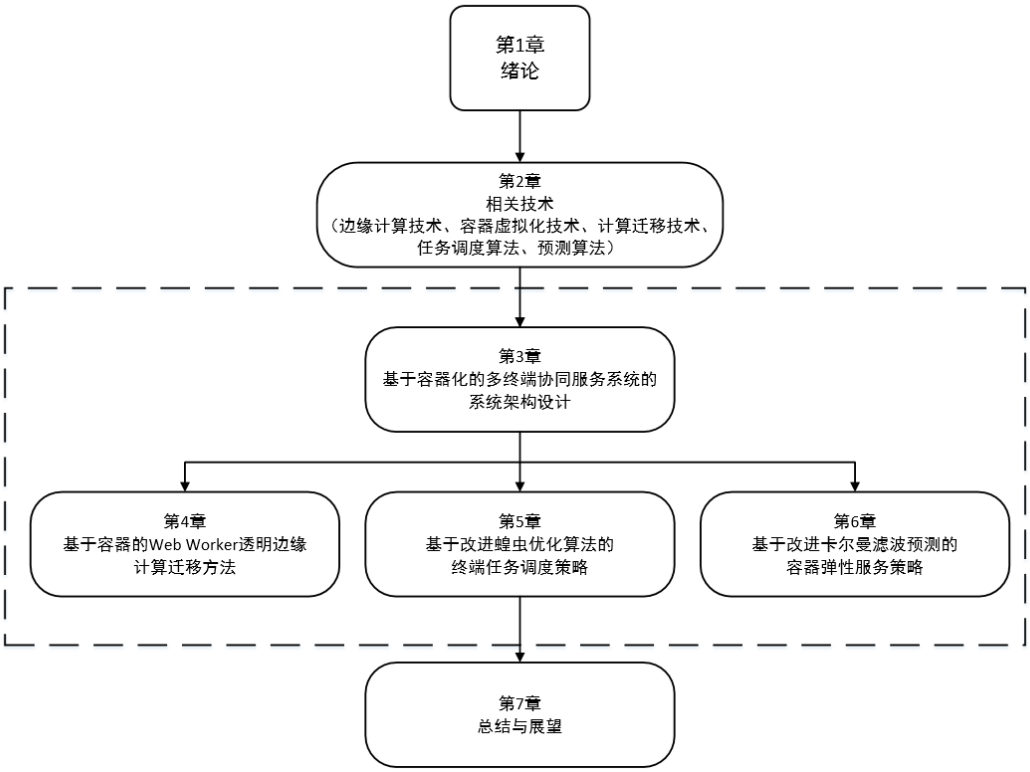

图 1.1 论文组织结构

终端服务管理等问题。后面三个研究点均为本章提出的系统中的部分具体实现。

第 4 章提出了一种基于容器的 Web Worker 透明边缘计算迁移方法，以透明计算迁移的方式将终端空闲资源利用起来为用户终端提供服务，减少任务计算执行时间，提高用户体验。

第 5 章提出了一种带有随机跳出机制的动态权重蝗虫优化算法来解决优化问题。并在此基础上提出了一种改进蝗虫优化算法，更进一步地提高了算法的性能，解决了终端任务调度问题。

第 6 章结合卡尔曼滤波算法，提出了一种基于预测的容器弹性服务策略，解决终端服务是否进行预部署的问题。

第 7 章总结了上述研究工作，并对未来工作进行了展望。

第 2 章　相关工作

2.1　云计算技术

2.2　边缘计算技术

2.3　虚拟化技术

资源管理问题是协同服务技术中的一个重要的问题 [12]。虚拟化技术是一种资源管理技术，利用底层虚拟、上层隔离等方法，将计算机中的物理资源，如计算、存储和网络等，进行抽象、切割，以更好的形式提供给用户使用 [?]。虚拟化技术将计算机上原有的固定配额的物理资源按照用户需要很方便地进行分配、提取和利用，可以提高计算机和终端资源的利用率。1959 年，克里斯托弗（Christopher Strachey）在他的学术报告《大型高速计算机中的时间共享》中提出了虚拟化的基本概念，成为了虚拟化技术的开端 [13]。在 IBM 公司、HP 公司、VMWare 公司等企业的不断努力下，经过了 60 年的发展和成熟，虚拟化技术逐渐成为云计算与边缘计算中非常重要的一种资源管理技术 [14]。用户根据自己的需求来租赁云计算或边缘计算的服务提供商所提供的对应资源配额的虚拟机，直接在虚拟化的操作系统中运行用户自己的应用而不需要考虑操作系统底层是如何实现的，就如同运行在一个真实而独立的实体物理机一样，非常透明。而虚拟化技术则被用来实现虚拟机的生命周期管理，包括创建、配置、关闭、监控、释放资源等等，维持这种对用户的透明性。应用了虚拟化技术的云计算模型如图2.1所示 [15]。

2.3.1　完全虚拟化技术

完全虚拟化技术（Full Virtualization Technology）通过在宿主机操作系统（Host Operation System，简称为 Host OS）上面架设一层虚拟机监控器（Virtual Machine Monitor，简称为 VMM）[16] 来实现对硬件的虚拟化，完全虚拟化技术中的虚拟机监控器，通常为超级监控器，即 Hypervisor 或者 Supervisor[17][18]。VMM 在虚拟硬件的基础上再进一步虚拟出一套用户的操作系统（Guest Operation System，简称为 Guest OS），供用户操作使用。为了方便阐述，在本文中，通过完全虚拟化技术实现的虚拟机简称为 VM 虚拟机。完全虚拟化技术的架构如图2.2所示 [19]。

云计算行业内曾经非常流行的工具如 VMware[20][21]、KVM[22]、VirtualBox[21] 等，都属于比较传统的完全虚拟化技术，这些完全虚拟化工具都拥有非

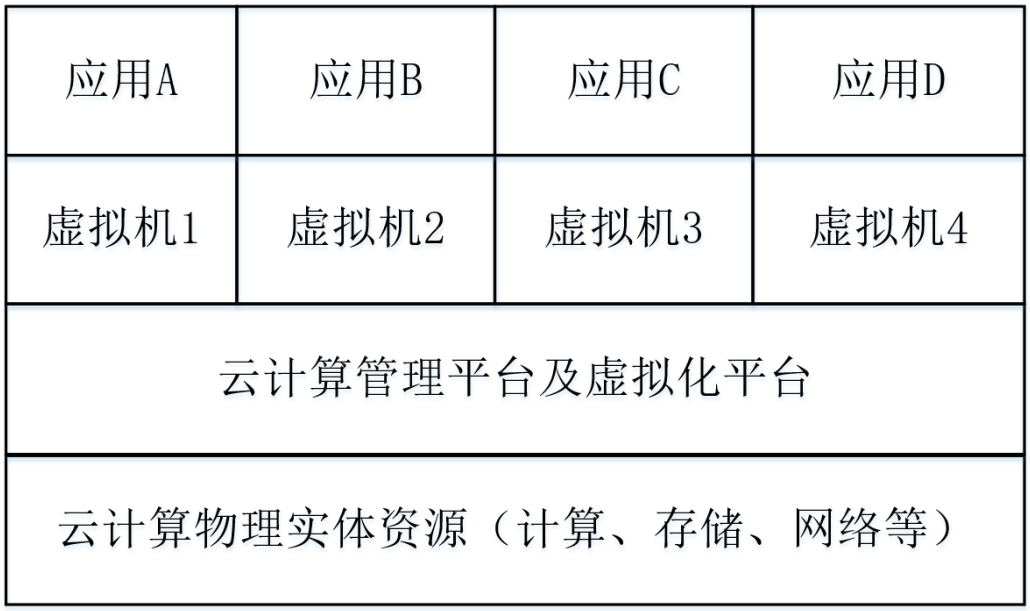

图 2.1 应用了虚拟化技术的云计算模型

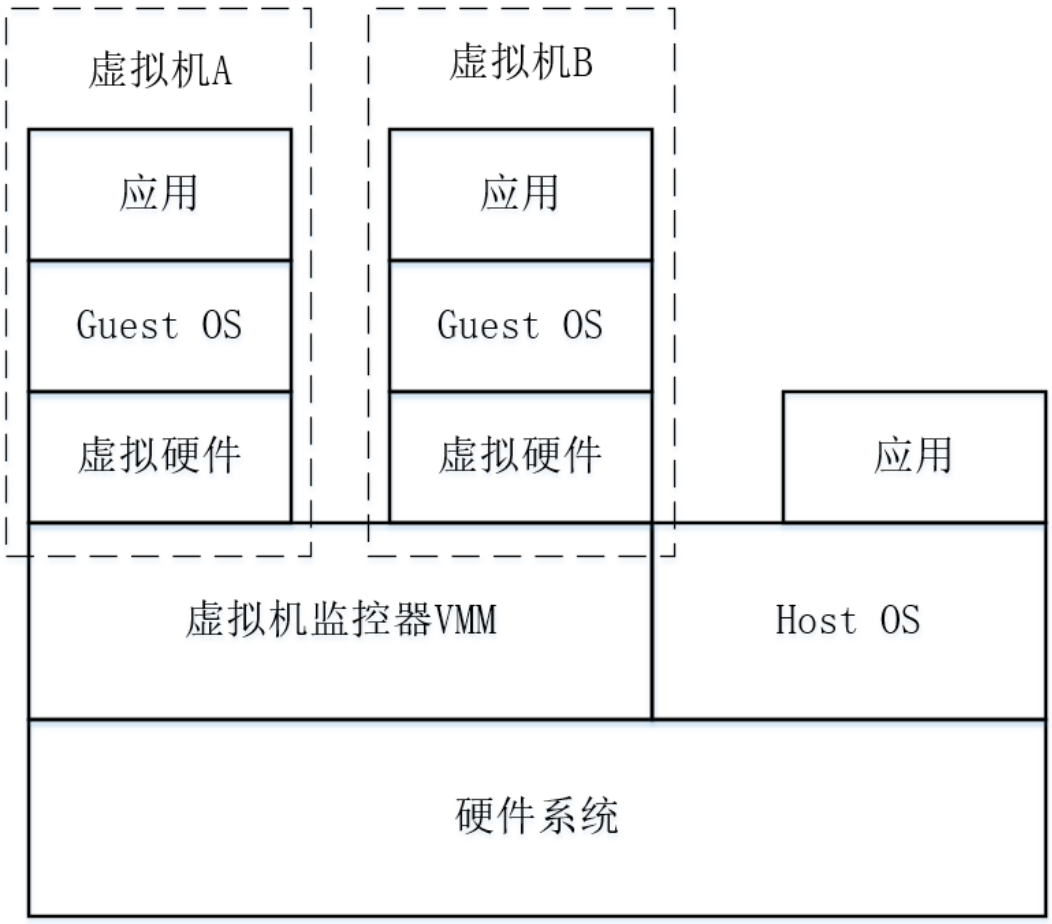

图 2.2 完全虚拟化技术架构

常良好的对硬件资源的管理、隔离、虚拟、利用的功能。但是另一方面，完全虚拟化技术使用了虚拟机监控器 VMM 来虚拟硬件层，能够提供相对较完整独立、隔离性好的虚拟化环境，但是启动速度慢，需要消耗大量额外资源来维持虚拟化环境。

2.3.2 操作系统级虚拟化技术

操作系统级虚拟化技术（Operation System Virtualization Technology），是一种轻量级虚拟化技术 [23]。与传统的完全虚拟化技术相比，操作系统级虚拟化技术不需要对底层硬件进行虚拟化，而是直接在操作系统层上采用隔离的方法虚拟出与宿主机操作系统互相隔离的虚拟机。对此有一个生动形象的比喻，如果将物理实体宿主机比作一座房子，那么其底层硬件则是地基，操作系统是房子的支撑墙，系统中运行的应用则是房子内的具体房间。而传统的完全虚拟化技术可以比喻为在原有房子的地基之上另外实现一层新的地基（虚拟机监控器 VMM），并在新地基上重新修建支撑墙（Guest OS）和房间（虚拟机）。相应地，操作系统级虚拟化技术则是直接借用了宿主机原有的地基（底层硬件）和支撑墙（操作系统），并在原来的房间里利用隔板隔离出来新的房间（虚拟机）。操作系统级虚拟化技术的架构如图2.3所示 [24]。由于没有使用虚拟机监控器 VMM 来对硬件进行虚拟化，所以操作系统级虚拟化技术启动速度非常快，而且也不会消耗非常多的额外虚拟化开销，非常轻量级 [25]。

2.3.3 容器虚拟化技术

容器虚拟化技术（Container Virtualization Technology）属于操作系统级虚拟化技术的一种 [26]。容器虚拟化技术虚拟出来的“虚拟机”通常被称为容器（Container）。容器技术底层使用的是基于 Linux 内核的 Linux Container（LXC）技术，而不需要进行额外的虚拟化操作。LXC 技术的核心技术包括 Namespace 技术和 Cgroups 技术。Namespace 技术可以为容器提供一个独立的、隔离的命名空间，其中包含 PID（进程）、UTS（host name）、MNT（文件）、NET（网络）、IPC（进程间交互）、USER（用户）等六个方面。不同命名空间的进程彼此之间看不到，Namespace 单独隔离出来的命名空间中的容器与其宿主机之间也是彼此看不到，只能通过 UTS 将对方当作网络中的另一个主机节点，这样就保持了容器的独立性和隔离性。而通过 Cgroups 技术，可以对每个容器所拥有的 CPU、内存、存储、输入输出等资源进行进程级别的管理和配置，这样就使得用户可以利用容器对终端资源进行按需分配管理。同时，利用 Cgroups 控制组，也能够对对

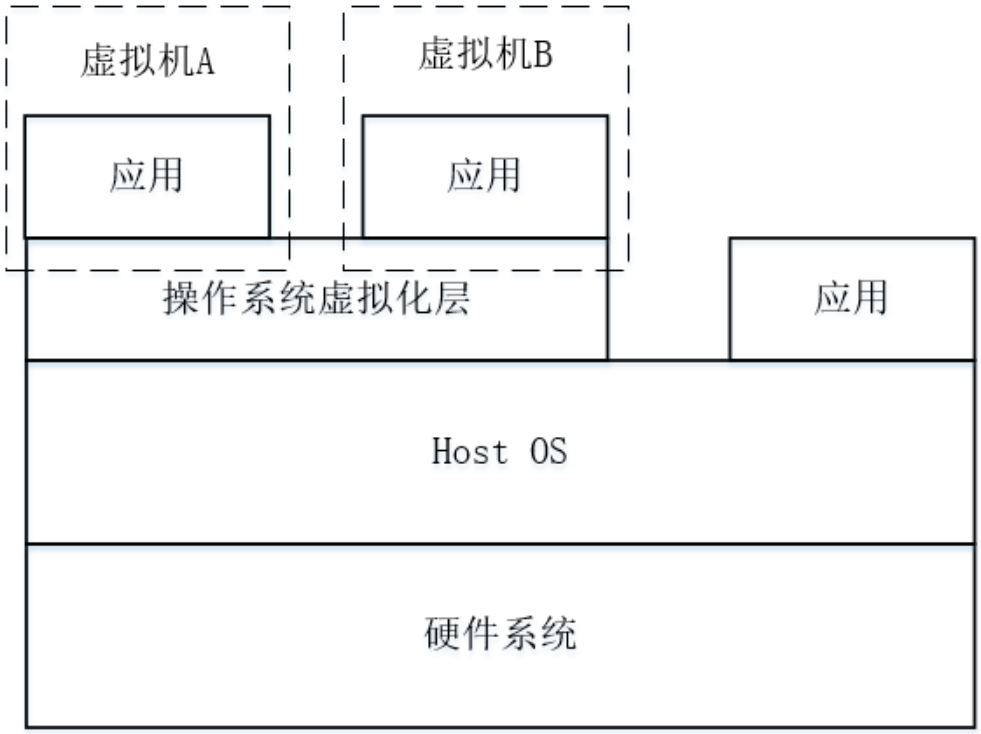

图 2.3 操作系统级虚拟化技术架构

容器内各种资源的消耗情况做监控。因为容器底层仍然是利用宿主机的操作系统，所以相比完全虚拟化技术，容器虚拟化技术基本没有太大的额外性能损耗，启动一个容器也相当于启动一个进程，启动时间基本为秒级时间。目前行业内最流行的基于容器虚拟化技术的产品是 Docker[27][28][29]。本文中所涉及到的容器虚拟化技术均是使用 Docker 应用实现的。

2.4 计算迁移技术

2.4.1 云计算中的计算迁移技术

传统的终端设备的服务模式通常是终端设备在本地进行计算任务的运行，依靠终端设备自身的计算能力来保证服务质量。而随着边缘智能终端设备所要承担的计算任务越来越重，单一终端设备本地计算难以满足终端服务和任务对终端计算能力的要求，计算迁移技术逐渐成为了一个可行的解决方案。计算迁移技术，也被称为计算卸载技术（computation offloading），是指将终端设备上的计算任务，通过网络的形式发送到其他设备上进行计算，再将计算结果通过网络返回给终端设备。

在边缘计算中，通常使用的方法是将边缘终端设备上的计算任务根据具体需求迁移一部分或迁移全部到云端计算资源上进行处理。云端计算资源通常是部署在云数据中心的机房，可以提供计算、存储、网络等资源。这种计算模式又被称为 Mobile Cloud Computing (MCC)。使用 MCC 的计算模式，可以让终端设备提供更加复杂的计算服务，并且减少终端设备的能源消耗。但另一方面，终端设备与云数据中心之间会出现大量数据交互，产生大量网络延迟，并且可能会出现信息泄露等隐患，终端服务的实时性与安全性都不能得到保证。

为了拉近云数据中心与用户终端之间的距离，云端计算资源也会部署在网络边缘，例如基站、边缘服务器等等。这种计算模式又被称为 Mobile Edge Computing (MEC)。与 MCC 类似，MEC 也是一种集中式的云计算资源，需要资源服务器、中央管理器、移动网络等基础设施的支持，部署起来会比较复杂。目前对于 MEC 中计算迁移技术的的研究，主要集中在计算迁移决策、计算资源管理及移动性管理三个方面，而对与 MEC 系统中的计算迁移方案的设计与实现研究较少。

文献【Cloudlets: Bringing the cloud to the mobile user】中介绍了一种计算迁移系统，利用部署在用户身边的可信的、计算资源丰富的计算设备（被称为 cloudlets）为移动用户提供服务。当用户在终端设备上产生服务请求时，终端设备可以在 cloudlets 上快速建立定制化的虚拟机，将计算任务迁移到对应虚拟机中并利用其资源来运行，用户可以利用轻量级客户端通过无线网络进行访问。cloudlets 模式是一种细粒度的计算迁移技术 []。这种计算迁移模式并没有在远端虚拟机中运行整个应用程序，而是将应用动态分割成多个可单独运行的组件，综合考虑每个组件的资源消耗情况、当前 cloudlets 状态、组件之间的依赖关系以及计算迁移决策等问题，来决定每个组件是否进行计算迁移。cloudlets 计算迁移模式相比 MCC 和 MEC，与用户之间的距离更近，传输时延更小，非常适合图书馆、咖啡厅、办公室等聚集场所。但 cloudlets 的缺点是需要在区域内单独进行部署，需要额外的硬件、场地、服务器等成本，使用范围有局限性。

2.4.2 基于 Web 的计算迁移技术

Web Worker 是一种基于 HTML5 的多线程方法，本文的研究对象就是面向 Web Worker 的计算迁移系统。基于 Web Worker 的服务系统可以拥有更好的跨平台性，应用开发者在开发服务应用的过程中可以不必考虑底层的系统架构。基于 Web Worker 的计算迁移方法分为透明迁移和非透明迁移两类。基于 Web Worker 的非透明迁移通常采用的方法是重写标准的 Web Worker 的接口 API，并在编写 Web 应用的时候以新的库的形式导入修改，使得 Web 应用在运行的时候通

过 WebSocket 通知运行在云端的服务端程序生成相应的 Web Worker 来进行计算[1234]。文献 [] 中提出了一种基于 Web Worker 的透明迁移方法，通过修改 Web Worker 运行环境代码来实现计算迁移的过程，但是没有做具体实现。

2.5 任务调度算法

2.5.1 传统任务调度

2.5.2 启发式算法

第 3 章　基于容器化多终端服务系统架构设计

3.1　引言

本章的内容结构组织如下：第3.2节介绍了一些相关技术的研究工作，包括自治系统的组网技术，微服务架构；第3.3节介绍了多终端协同服务系统的架构设计，包括引入微服务架构、终端在系统中的身份、系统模块设计等内容；第3.4节提出了多终端协同服务系统的构建方法，包括自治系统的网络选择、自治系统的组网方式、自治系统的管理节点选择等内容；第3.5节总结了本章内容。

3.2　相关工作

3.2.1　自治系统组网技术

3.2.2　微服务架构

前面加上康威定律和 microservice。

在互联网服务行业中，微服务架构逐渐成为非常流行的架构，在学术界关于微服务架构也同样有一些研究，下面分别从国内研究和国外研究两方面介绍国内外关于微服务架构的研究工作。

3.2.2.1　国内研究现状

国内相关研究文献中对于微服务的研究工作主要集中在利用微服务框架搭建新的服务系统或者将原有的服务系统微服务化，并分析服务系统的功能性需求和非功能性需求，解决微服务化过程中遇到的一些问题。北京大学的龙新征等人在文献 [4] 中针对微服务架构中的服务注册、服务发现、负载均衡等常见问题，提出了一种分层的微服务框架，基于这个框架设计并实现了“北京大学校园移动信息服务平台”，为校内师生提供信息服务。同济大学的郭栋等人在文献 [5] 中设计了一种基于微服务架构构建的云件 PaaS 平台，可以在不需要对传统软件做出任何改动的情况下，将软件部署到云端 WEB 服务器，并通过浏览器以 WEB 的形式为终端用户提供软件服务。北京交通大学的谭一鸣在文献 [6] 中对平台化服务框架的功能性需求和非功能性需求进行了分析，设计并实现了使用 API 网关构建的微服务系统。南京大学的唐文宇在文献 [7] 中分析了微服务的安全性访问需求，设计并实现了一个安全系统来解决微服务中的第三方应用审核认证及微服务权限访问控制问题。北京交通大学的肖仲垚在文献 [8] 中分析了业务微服务

化过程中对于通信框架的高效简洁高可用等需求，设计并实现了一种微服务通信框架。北京邮电大学的宋鹏威在文献 [9] 中设计了一个“开放式微服务框架”，并归纳出了一组行业相关开发者在开发过程中所需要注意的内容和原则。作者还基于这些内容和原则完成了一个“开放式数据采集、存储和分析服务”，证明了该框架对微服务的开发指导工作的有效性。

3.2.2.2 国外研究现状

国外相关研究文献中对于微服务技术的研究工作可以分为 4 个方面: 介绍微服务架构，并结合具体项目来说明效果；开发微服务相关组件或工具，实现微服务架构下的某些特殊功能；评估比较微服务架构的性能; 宏观讨论企业从单体式架构转变为微服务架构的原因及可能遇到的问题。

在具体微服务架构项目方面，Hasselbring 在文献 [10] 中介绍了在欧洲最大的电子商务平台之一的 otto.de 上引入微服务架构，沿业务服务进行垂直分解，为应用和服务提供高可扩展性和高可用性的能力，并解决了微服务化过程中的耦合、集成、可伸缩性、监视和开发等问题。Innerbichler 等人在文献 [11] 中设计并开发了一种基于微服务架构的物联网平台 NIMBLE，该平台通过分散，可扩展服务的组合来实现平台的核心业务功能，服务之间以及平台用户、制造商、供应商、传感器和 Web 资源之间的通信通过简单的协议和轻量级机制来支持。在开发微服务组件方面，Granchelli 等人在文献 [12] 中设计了一种用于微服务体系结构恢复的原型工具 MicroART，能够生成基于微服务的系统软件架构模型。Mayer 等人在文献 [13] 中提出了一种用于微服务的监控和管理的实验仪表，能够满足用户不同需求并支持集成不同监控设施来收集微服务运行数据。在评估微服务架构性能方面，Amaral 等人在文献 [14] 中利用主从式和嵌套式容器两种模型来比较微服务架构中的 CPU 和网络性能。Ueda 等人在文献 [15] 中使用了 Acme Air 来分析比较微服务和单体架构运行 Node.js 和 Java 的性能，实验结果表明相同的硬件配置下微服务的额外开销更大，消耗时间也更多。在宏观讨论企业微服务化可能遇到的问题方面，Kalske 等人在文献 [16] 中指出，迈向微服务架构的典型原因是复杂性、可扩展性和代码所有权，企业微服务化面对的挑战可以分解为技术挑战和组织挑战，前者包括微服务的解耦、划分服务边界代码重构等，后者则包括根据康威定律将大的团队分为可以自主工作的小型团队等等。

3.3 多终端协同服务系统架构设计

3.3.1 终端资源特点

终端资源拥有如下特点：

- 终端资源总体数量庞大
- 终端资源单体数量较少
- 终端之间异构型强
- 终端局部网络通信时延小

3.3.2 微服务架构

微服务架构是一种互联网应用服务的软件架构，主要应用于互联网应用服务的服务端软件开发。微服务架构由面向服务架构 SOA 发展而来，其核心理论基础来自于康威定律 [2] 中关于组织结构与其设计的系统结构之间关系的描述，即任何组织设计的系统，其结构都是组织本身沟通结构的复制。

2014 年学者 Martin Fowler 正式提出微服务架构的概念 [3]：微服务架构以一套微小的服务的方式来开发和部署一个单独的应用，这些微小的服务根据业务功能来划分，通过自动化部署机制独立部署运行在自己的进程中，微服务之间使用轻量级通信机制来进行通信。一个典型的微服务架构应该包括客户端、微服务网关、服务发现、微服务原子层、数据库、部署平台等模块，根据不同应用类型及服务规模，可以增加负载均衡、权限认证、服务熔断、日志监控等模块，来满足服务的非功能性需求。

虽然 Martin Fowler 给出了微服务的一种定义，但是他也同时指出，微服务并不局限于该定义。Martin 尝试归纳和描述微服务架构风格所具有的共同特点，这些特点并不是所有微服务架构风格都要拥有的, 也不是用来定义微服务架构本身的, 而是微服务架构风格被希望要拥有的特点。也就是说，微服务架构风格不是微服务化的终点，而是微服务化的方向。下面简单介绍一下文献 [3] 中总结的几个微服务架构风格的特点。

- 服务组件化：微服务中，服务可以被当作进程外组件，独立进行部署，服务之间利用网络服务请求或者远程过程调用来进行通信。一个好的微服务架构的目标是通过服务合同中的解耦服务边界和进化机制来帮助各个微服务独立部署运行。微服务架构的设计者希望对任何一个组件或者服务的改动和升级都只需要重新部署该服务而不需要重新部署整个应用程序，并且在升级过程中尽可能少地改变服务间通信的接口。

• 围绕业务功能组织服务：当把一个大的应用拆分成小的部分的时候，通常的方法都是根据技术层面分为 UI 团队、服务端逻辑团队和数据库团队。但是这种拆分团队的方式会使得即使一个简单的变动都会导致整个团队需要耗费时间和预算来适应和协调。康威在文献 [2] 中提出了康威定律，其中有一条提到：任何组织设计的系统，其结构都是组织本身沟通结构的复制。根据康威定律的这条描述，微服务架构采用围绕业务功能来拆分应用和组织服务的方法。在微服务架构中，设计组织被分为小的开发团队，与之相对应的是，应用被拆分为小的服务，应用之间的沟通过程就是团队之间的沟通过程。为保证应用之间沟通过程清晰明确，团队之间需要划分清晰的服务边界。这样的划分方法也要求每个小团队本身提供开发过程中所需要的所有技能。

• 基础设施自动化：许多开发团队都是使用持续交付和持续集成技术来构建微服务架构的应用和系统的,这使得基础设施自动化技术得到了广泛的应用。而随着容器技术、云计算技术等技术在过去几年的快速发展，基础设施自动化技术取得了长足的进步，这也间接降低了微服务构建、部署、运行的复杂性。

根据微服务架构的这些特点，我们可以看出，微服务架构非常适合应用到多终端协同服务系统中。对于终端服务来说，终端拥有海量资源，但这些资源以较为分散的、单体资源较少、终端总数量较多、终端之间异构性强的特点分布在终端上面。将微服务架构引入到多终端协同服务系统中来，可以将终端资源的特点与微服务架构的特点很好地结合起来。

对于单体资源有限的终端来说，庞大而复杂的单体式服务应用会消耗大量终端资源，单个终端难以满足其服务的质量。而海量的空闲终端资源分布较为分散，对于维持和提升单体式服务质量也并没有帮助。将单体式服务改编成组件化的微服务后，每个组件式的微服务都只会消耗较少的终端资源，这样分散的终端资源也能够有效维持和提升微服务的质量，另一方面也是提升了终端资源的利用率。微服务架构围绕业务功能组织服务，拆分后的各个小的应用之间通过轻量级通信机制进行数据通信。而终端之间通常通过局域网交换信息，网络传输速度比较快，节点之间距离近，时延较小，相比传统的终端与云端之间进行远距离通信要更加具有实时性，响应速度更快，终端服务质量也更好。微服务架构使用自动化的基础设施，通常利用容器技术来对服务进行部署和构建，与终端服务系统利用容器技术来管理利用终端资源、形成终端资源池、并以容器的形式对外提供服务的特点非常契合。

3.3.3 使用容器化

为了将终端空余资源更好地利用起来，为终端用户提供质量更优的服务，本研究结合终端资源特点，并引入微服务架构的特点，提出了基于容器化的多终端协同服务系统。要将终端上分散的资源利用起来为用户提供服务，使用虚拟化技术是一种很好的方案。虚拟化技术能够将计算机或终端的资源如 CPU、内存、存储等抽象、整合起来，打破实体资源的整体性，形成可以按需分配的资源池，让用户能够以更高效合理的方式来对计算机或终端资源进行管理和利用。

虚拟化技术的具体实现形式是虚拟机。虚拟机是一种运行在宿主机上面的，利用虚拟化技术形成的独立的、隔离的虚拟主机，每台虚拟机中可以独立运行属于它自己的不同的操作系统，并在虚拟机内部运行独立的应用程序，从内部运行的应用程序的角度来看，虚拟机就像一台完整的计算机或终端一样，虚拟机的用户、应用程序和虚拟机操作系统不能够分辨虚拟机跟一台物理主机（bare-metal host）之间的区别。由于虚拟化技术的抽象、隔离等特点，多台虚拟机可以同时运行在一台物理主机上，共享宿主机的资源，而且它们彼此之间并不会互相干扰，只会把对方当作另一台独立的计算机或终端主机，运行在虚拟机中的应用程序的安全性更好。另一方面，每台虚拟机都可以从资源池中根据需要配置相应的 CPU、内存、存储等资源，相比直接运行在物理主机上，运行在虚拟机中的应用程序的可用性更好。

传统的虚拟机（Virtual Machine，简称为 VM）通常使用的是完全虚拟化技术，在宿主机底层硬件之上独立于操作系统创建了一层虚拟机监视器（Hypervisor），将底层硬件环境进行虚拟化，为上层 VM 虚拟机的操作系统的运行提供支持。使用 VM 虚拟机可以为用户提供一个隔离得非常彻底的运行环境。但是 VM 虚拟机从底层虚拟出一整套操作系统，需要占用非常大的存储空间。而且由于完全虚拟化技术是处理器密集型技术，会带来大量额外的运行开销，大大加重宿主机的运行负载。这些显然与终端资源的单个终端资源有限的特点有冲突，资源利用率低。另外，VM 虚拟机的启动时间非常长，甚至长达几分钟，也非常不适合追求快速响应的终端服务系统。

因为 VM 虚拟机应用于终端服务系统的资源管理会带来一系列问题，所以我们使用了容器技术来进行终端服务中的资源管理。容器技术是一种轻量级虚拟化技术，是一种操作系统（Operation System，简称为 OS）层的虚拟化技术。容器技术可以在宿主机操作系统之上隔离出一个独立的“虚拟机”，这个“虚拟机”通常被称为容器。容器技术底层使用的是基于 Linux 内核的 Linux Container（LXC）

技术，而不需要进行额外的虚拟化操作。LXC 技术的核心技术包括 Namespace 技术和 Cgroups 技术。Namespace 技术可以为容器提供一个独立的、隔离的命名空间，其中包含 PID（进程）、UTS（host name）、MNT（文件）、NET（网络）、IPC（进程间交互）、USER（用户）等六个方面。不同命名空间的进程彼此之间看不到，Namespace 单独隔离出来的命名空间中的容器与其宿主机之间也是彼此看不到，只能通过 UTS 将对方当作网络中的另一个主机节点，这样就保持了容器的独立性和隔离性。而通过 Cgroups 技术，可以对每个容器所拥有的 CPU、内存、存储、输入输出等资源进行进程级别的管理和配置，这样就使得用户可以利用容器对终端资源进行按需分配管理。同时，利用 Cgroups 控制组，也能够对对容器内各种资源的消耗情况做监控。因为容器底层仍然是利用宿主机的操作系统，所以相比 VM 虚拟机，容器技术基本没有太大的额外性能损耗，启动一个容器也相当于启动一个进程，启动时间基本为秒级时间。引入了容器这种轻量级虚拟化技术，我们可以在更为有效地对多终端协同服务系统中的终端资源进行管理和利用的同时，避免了传统 VM 虚拟机对于终端资源的额外消耗和浪费。

3.3.4 层次图

本研究设计了基于容器化多终端协同服务系统，其层次图如图3.1所示。

系统最底层为基础设施层，主要为终端设备为上层服务所能提供的资源，包括计算、存储、网络等资源。这些资源为物理实体资源，处于空闲状态，未被组织起来，难以直接向上层服务提供。为了将这些终端物理实体资源组织管理起来，我们在基础设施层上面增加了容器支撑层。这一层中利用容器虚拟化技术，实现容器运行时支撑、容器生命周期管理、容器对底层实体资源的管理、容器内部资源使用情况监控、容器镜像仓库管理等等功能。Docker 技术近几年发展迅速，成为容器虚拟化技术的代表，本研究中涉及到的实验及系统设计，均使用 Docker 技术来代表容器技术。Docker 容器技术在终端上部署的架构图如图3.2所示。容器支撑层中的容器运行时的具体实现是 Docker Container。对容器的生命周期管理主要通过 Docker Containerd 来实现，具体的生命周期管理工作还包括容器的创建、运行、关闭等。对终端资源的管理和对容器资源的监控主要通过 Docker 的 Cgroups 来实现，具体的资源管理工作还包括对终端可用资源的监控和上报、对终端资源进行资源池化、按需划分终端资源、监控本地资源使用情况、当资源利用出现异常时进行报警处理等。对容器镜像仓库的管理主要是通过 Docker Image Repository 来管理系统内所涉及到的服务的镜像，具体的镜像管理工作还包括镜像的存储、更新、分发下载等。

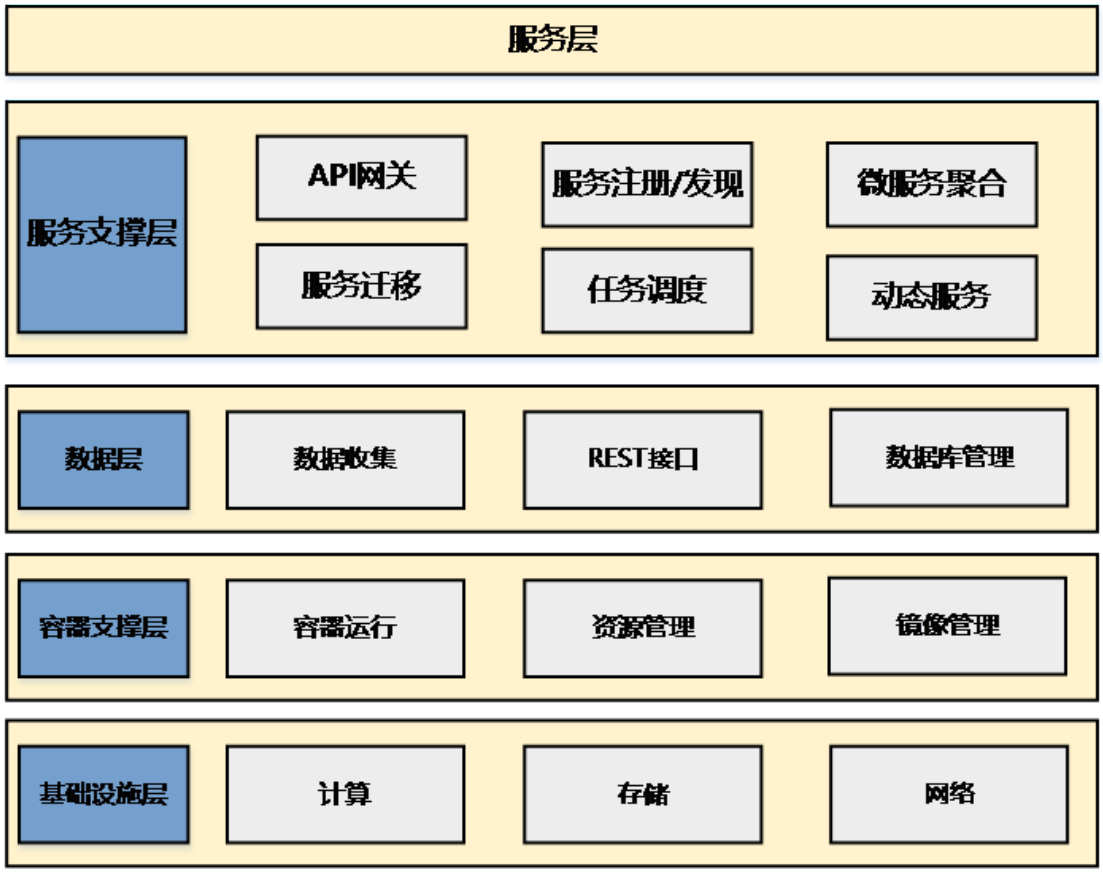

图 3.1 基于容器化多终端协同服务系统层次图

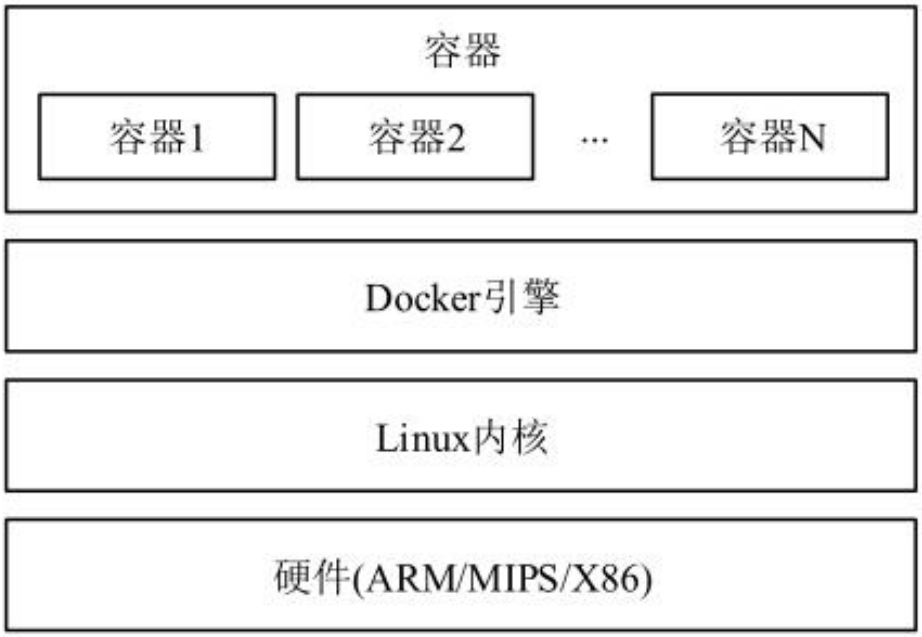

图 3.1 Docker 架构图

图 3.2 Docker 容器架构图

容器上面一层是数据层，是承上启下的一层。这一层的主要功能是对底层上报的资源数据进行收集整理、为上层微服务之间的通信提供 REST 接口、对多终端协同服务系统的数据库进行管理等等。数据层上面是服务支撑层，这一层借鉴了很多微服务架构中的模块，主要包括：

- API 网关：接收用户请求并将其分解为具体的微服务请求
- 服务注册：向系统中注册新的可提供的微服务
- 微服务聚合：将多个微服务的计算结果整合起来返回给用户
- 服务迁移：当出现节点变动的时候，比如节点出现故障下线，或者新节点上线的时候，将服务迁移到其他合适节点上继续运行
- 任务调度：根据任务类型及节点资源类型，将任务调度到最合适的终端节点上运行，达到最优调度
- 动态服务：根据用户请求流量，动态调整服务规模大小

整个系统的最顶层则是服务层，这一层主要包含多终端协同起来为用户提供的各种服务。同时这一端也是更多地交给了终端服务的开发者来实现。终端服务的开发者只需要将开发好的服务应用，通过服务注册模块注册到系统中即可。

3.3.5 结构图

本研究设计的基于容器化多终端协同服务系统，其架构图如图3.3所示。

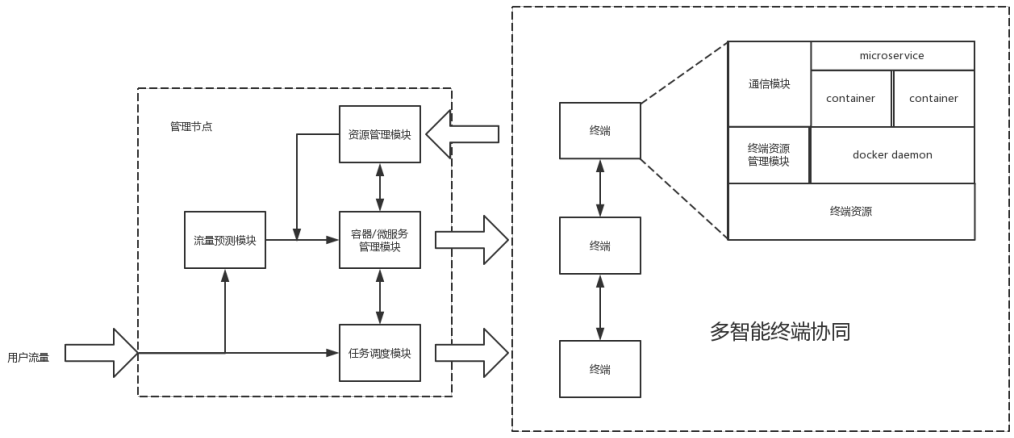

图 3.3 基于容器化多终端协同服务系统架构图

在架构图中，整个系统可以分为管理节点和普通执行节点。在管理节点中包含了弹性模块、资源管理模块、服务管理模块以及任务调度模块。当一个新的终端节点上线的时候，需要向资源管理模块汇报该节点的资源情况，包括可提供资源类型及本地资源负载情况等。当一个新的服务上线的时候，需要向服务管理节

点进行注册，向镜像仓库上传镜像，并根据资源管理模块的安排，分配合适的节点运行容器。当用户请求流量到达管理节点的时候，会由任务调度模块进行统一调度，根据服务部署情况及终端节点资源情况选择合适的节点和容器进行执行。另外，当用户请求流量到达管理节点的时候，弹性服务模块会记录该数据，对未来一段时间的用户请求流量情况进行预测，并根据资源管理模块的反馈情况，计算出服务规模的最优大小，由服务管理模块进行服务规模弹性调整。这其中的任务调度模块与弹性服务模块也是本文后面两章的研究重点。

在架构图中，每个终端物理节点代表着层次图中的基础设施层，终端上运行的 Docker Daemon 和 Docker Container 代表着层次途中的容器支撑层，管理节点与普通执行节点之间进行的数据交换代表着层次图中的数据层，管理节点中的几个模块代表着层次途中的服务支撑层，终端中运行的微服务应用程序则代表着层次图中的服务层。

3.3.6 终端在系统中的角色

基于容器化多终端协同服务系统中的终端有三个身份，每个终端需要完成终端本身为用户提供的工作，即“本地服务”，当终端本身资源不足不能够很好的完成用户请求的任务，则应该通过系统中的调度中心向其他空闲节点请求提供相应资源来进行协同服务，而当终端本身资源有剩余的时候，该终端又可以通过调度中心将本身的资源以容器虚拟化的形式向系统中的其他节点提供出去。

在这个系统中，用户、服务、终端、容器这几个概念之间的关系如图3.4所示。用户是整个服务过程的发起者，能够通过自己身边的任何一个设备访问该设备提供的服务，用户对服务的每一次访问都是一次请求任务。服务代表终端能够为用户提供某种服务的能力，是一个比较虚的概念，具体包含两种能力：提供该服务应用的虚拟化容器（或虚拟化镜像）和能够支持该容器快速运行的相应资源。新的服务上线需要向系统注册服务信息，提供服务镜像，上报服务运行所需要的相应资源，并暴露相应服务端口。用户访问服务的过程，实质是用户通过服务对外暴露的端口访问部署在终端上的容器，并由终端向用户提供相应的资源。

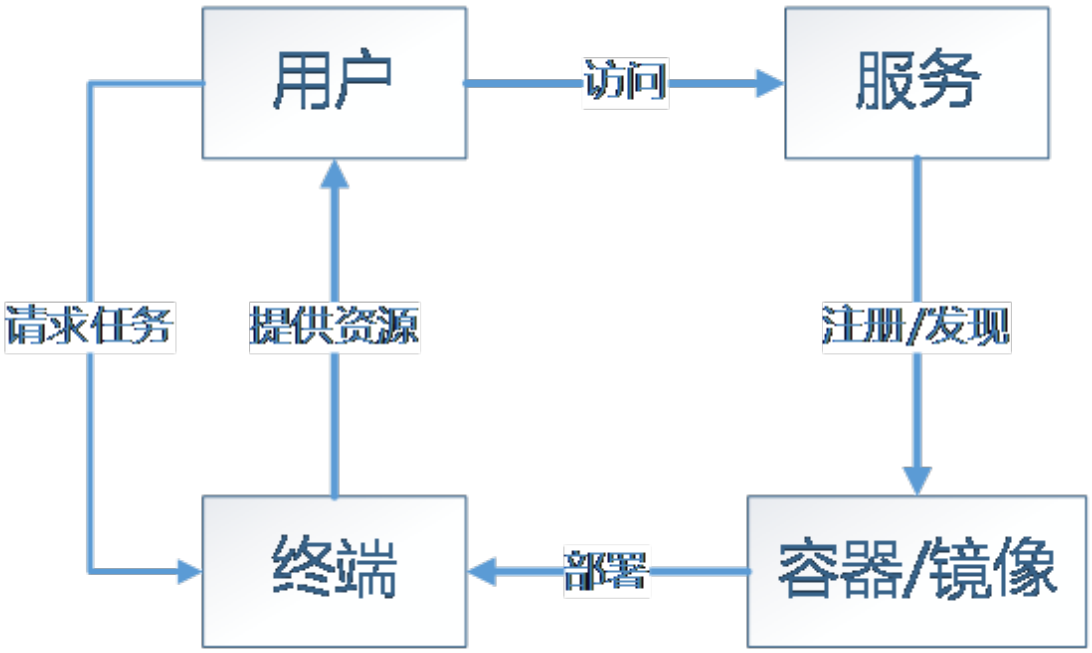

图 3.4 用户、服务、终端、容器概念之间的关系

3.4 多终端协同服务系统的构建方法

3.4.1 自治系统的网络选择

自组织网络的应用遍布于军事、数据通信和突发事件处理等领域，其体系结构是指包含协议和拓扑的网络整体设计，拓扑是自组织网络研究中的典型问题。动态条件下建立满足应用的自组织网络是网络体系结构研究的最终目标。自组织网络与现在大部分网络不一样，由于现阶段网络是一种基于客户端－服务器模式下的网络，通过客户端发送请求，利用服务端接收反馈需求，其中各种网络设备在网络中具有特定角色，而自组织网络中的各种设备是对等的。在信息交互时，自组织网络既可以作为客户端，也可做服务端。因为预先建立的基础骨干网设施还不够完善，所以对于终端系统来说，应该使用去中心化的系统架构。

3.4.2 去中心化的自组织网络构建

对于整体而言，利用各个节点间的连接情况来构建一个 Connectivity-based Decentralized Node Clustering（CDC）。首先选择若干初始种子节点，初始种子节点的选择算法可以进一步进行研究。每个种子向周围的邻居节点发送消息，消息中会携带 ID、种子节点 ID、权重、TTL、发送节点相关信息等信息，邻居节点在收到消息后会对该消息做一定处理，并加入节点自身相关的信息，再发送到它的邻居节点。节点消息不断传递下去，直到传递到某个已经确定属于其他集群的节点，或者权重信息消耗尽，或者 TTL 减小到 0，则认为消息传递结束，消息传

递过程中经过的节点形成一个集群，集群中的节点互相交换相关信息。在集群内部，可以通过考虑上线时间、稳定程度、负载情况、计算能力、邻居数量、网络状态等信息，选择一个超级节点作为集群的任务调度节点。而当有新的节点上线的时候，不需要重新进行集群的划分，而是应该让该节点向周围邻居节点广播消息，根据网络相关程度、上线时间、地理位置、特殊 Tag 等方法选择周围邻居节点中的一个加入其所在集群。这样就形成了一个整体去中心化、终端节点自治的一个多终端协同服务系统的底层结构，如图3.5所示是一个简化的模型。

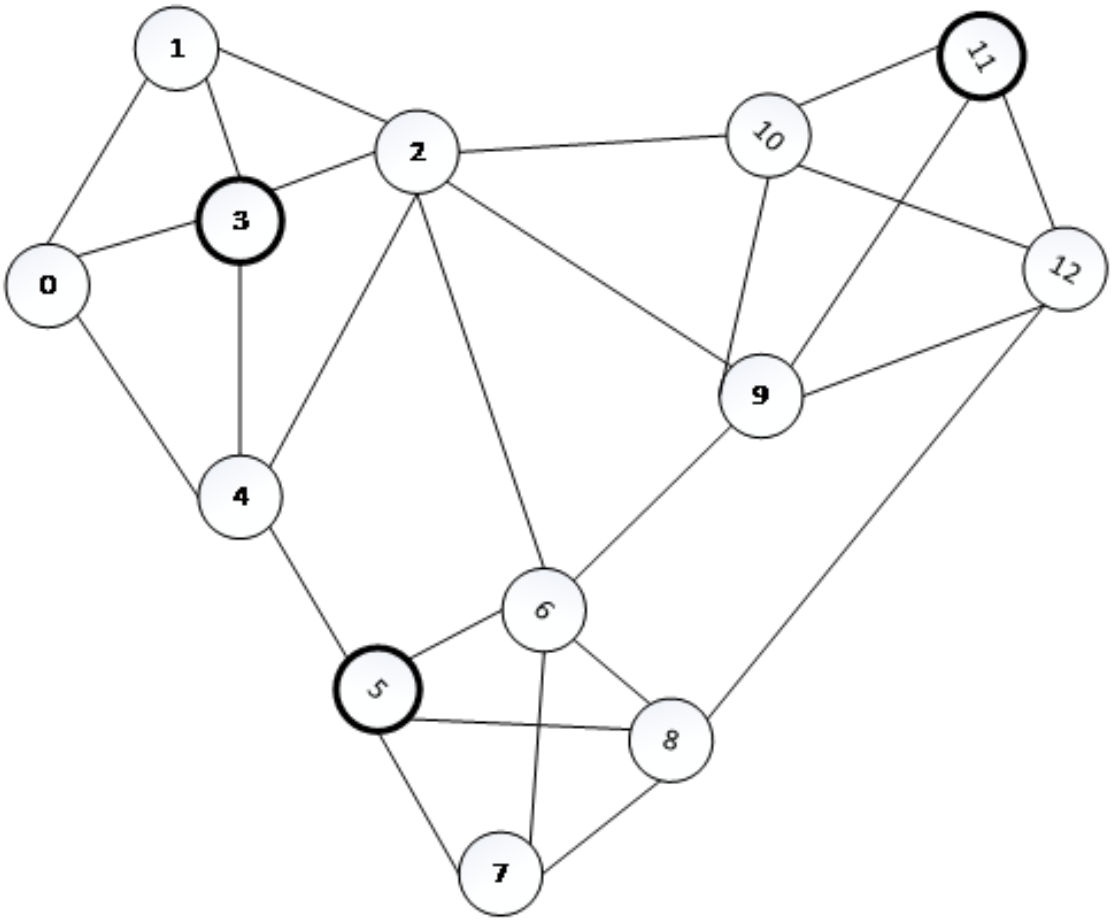

图 3.5 去中心化的自组织网络结构

3.5 本章小结

第 4 章　基于容器化服务资源提供技术

4.1　引言

随着计算机技术的快速发展和边缘计算技术的逐渐成熟，位于网络边缘的用户终端设备在用户的数字化生活中正扮演着越来越重要的角色，其定位逐渐从单一的用户服务发起者向用户服务的提供者转变。这会使用户终端承担越来越多的计算任务，这也对终端设备的服务质量和计算能力提出了越来越高的要求。为了满足用户对于服务质量的要求，人们提出了计算迁移技术，将终端设备上的计算任务通过网络迁移到云端服务器上来完成。但是引入云端协同的计算迁移技术，还是存在着时延较大、数据隐私性不能保证、成本较高等问题，难以满足用户需求。而另一方面，边缘终端设备距离用户更近，而且拥有很多空余资源没有得到充分利用，如何将这部分距离用户更近、成本更低的资源组织利用起来为用户设备提供计算迁移服务也成为了计算迁移技术中的一个值得研究的问题。本文以 HTML5 中的 Web Worker 方法为例，研究 Web 应用透明计算迁移到以容器形式部署在边缘设备上的服务端，设计并实现了一种基于容器的 Web Worker 透明边缘计算迁移系统，并利用若干实验来验证该系统能够将周围设备的资源利用起来，减少计算总执行时间，提升用户体验。

4.2　相关研究

4.2.1　计算迁移技术

传统的终端设备的服务模式通常是终端设备在本地进行计算任务的运行，依靠终端设备自身的计算能力来保证服务质量。而随着边缘智能终端设备所要承担的计算任务越来越重，单一终端设备本地计算难以满足终端服务和任务对终端计算能力的要求，计算迁移技术逐渐成为了一个可行的解决方案。计算迁移技术，也被称为计算卸载技术（computation offloading），是指将终端设备上的计算任务，通过网络的形式发送到其他设备上进行计算，再将计算结果通过网络返回给终端设备。

在边缘计算中，通常使用的方法是将边缘终端设备上的计算任务根据具体需求迁移一部分或迁移全部到云端计算资源上进行处理。云端计算资源通常是部署在云数据中心的机房，可以提供计算、存储、网络等资源。这种计算模式又被称为 Mobile Cloud Computing (MCC)。使用 MCC 的计算模式，可以让终端设

备提供更加复杂的计算服务，并且减少终端设备的能源消耗。但另一方面，终端设备与云数据中心之间会出现大量数据交互，产生大量网络延迟，并且可能会出现信息泄露等隐患，终端服务的实时性与安全性都不能得到保证。

为了拉近云数据中心与用户终端之间的距离，云端计算资源也会部署在网络边缘，例如基站、边缘服务器等等。这种计算模式又被称为 Mobile Edge Computing (MEC)。与 MCC 类似，MEC 也是一种集中式的云计算资源，需要资源服务器、中央管理器、移动网络等基础设施的支持，部署起来会比较复杂。目前对于 MEC 中计算迁移技术的的研究，主要集中在计算迁移决策、计算资源管理及移动性管理三个方面，而对与 MEC 系统中的计算迁移方案的设计与实现研究较少。

文献【Cloudlets: Bringing the cloud to the mobile user】中介绍了一种计算迁移系统，利用部署在用户身边的可信的、计算资源丰富的计算设备（被称为 cloudlets）为移动用户提供服务。当用户在终端设备上产生服务请求时，终端设备可以在 cloudlets 上快速建立定制化的虚拟机，将计算任务迁移到对应虚拟机中并利用其资源来运行，用户可以利用轻量级客户端通过无线网络进行访问。cloudlets 模式是一种细粒度的计算迁移技术 []。这种计算迁移模式并没有在远端虚拟机中运行整个应用程序，而是将应用动态分割成多个可单独运行的组件，综合考虑每个组件的资源消耗情况、当前 cloudlets 状态、组件之间的依赖关系以及计算迁移决策等问题，来决定每个组件是否进行计算迁移。cloudlets 计算迁移模式相比 MCC 和 MEC，与用户之间的距离更近，传输时延更小，非常适合图书馆、咖啡厅、办公室等聚集场所。但 cloudlets 的缺点是需要在区域内单独进行部署，需要额外的硬件、场地、服务器等成本，使用范围有局限性。

4.2.2 Web Worker

Web Worker 是一种基于 HTML5 的多线程方法，本文的研究对象就是面向 Web Worker 的计算迁移系统。基于 Web Worker 的服务系统可以拥有更好的跨平台性，应用开发者在开发服务应用的过程中可以不必考虑底层的系统架构。基于 Web Worker 的计算迁移方法分为透明迁移和非透明迁移两类。基于 Web Worker 的非透明迁移通常采用的方法是重写标准的 Web Worker 的接口 API，并在编写 Web 应用的时候以新的库的形式导入修改，使得 Web 应用在运行的时候通过 WebSocket 通知运行在云端的服务端程序生成相应的 Web Worker 来进行计算 [1234]。文献 [] 中提出了一种基于 Web Worker 的透明迁移方法，通过修改 Web Worker 运行环境代码来实现计算迁移的过程，但是没有做具体实现。

4.2.3 虚拟化技术

很多计算迁移系统中也用到了虚拟化技术。虚拟化技术是一种资源管理技术，利用底层虚拟、上层隔离等方法，将计算机中的物理资源，如计算、存储和网络等，进行抽象、切割，以更好的形式提供给用户使用 []。当管理多个计算机资源的时候，利用虚拟化技术还可以将整个计算机集群进行虚拟化，形成资源池，可以更方便地管理计算机集群中的资源，也可以按需求给用户提供相应的虚拟资源，提高资源利用率，并且可以对用户保持透明，用户可以不必关心底层的资源是如何进行虚拟化和提供的。传统的抽象方法通常是完全虚拟化，在计算机底层上建立 Hypervisor 层，来对底层硬件进行虚拟化，并在其上运行虚拟化的操作系统。基于完全虚拟化技术运行的实体通常被称为虚拟机，也叫 Virtual Machine (VM)。另外一种比较流行的抽象方法是操作系统 (OS) 虚拟化，这种虚拟化技术利用 LXC 技术在宿主机的操作系统之上进行隔离、封装。基于 OS 虚拟化技术运行的实体通常被称为容器，也叫 Container。

很多计算迁移的研究 [1-4] 都使用了虚拟机 VM 来作为云端承载迁移服务程序的基础。然而因为需要对底层硬件做虚拟化，所以使用虚拟机 VM 会引入极高的启动时延，这对于响应时间敏感的终端用户服务来说是难以接受的。如果考虑提前部署虚拟机 VM 的方案，虽然能够减少启动时间带来的时延，但相对重型的虚拟机 VM 解决方案也会带来较大的额外开销，这也是一种不太能够接受的服务费用以及资源浪费 []。文献 [] 中提出使用基于 OS 虚拟化技术的容器来代替虚拟机 VM，承载云端迁移服务程序。这种轻量级的虚拟化技术不仅能够大大减少启动时间带来的时延，也能够大大降低虚拟化技术带来的额外资源开销。

4.2.4 容器集群技术

4.3 基于边缘容器的 Web Worker 透明计算迁移系统设计

为了将用户身边终端设备的空闲资源利用起来，并且方便部署管理，对 Web 应用开发者透明，本文提出一种基于边缘容器的 Web Worker 计算迁移系统。计算迁移系统的实质是利用冗余来提高终端服务系统的服务质量，而对于用户的终端环境来说，用户身边还有着大量的终端设备都处于空余状态，拥有大量的空闲资源可以用来提供计算迁移服务。同时，相比其他边缘计算迁移系统，这种身边的终端距离用户所使用的终端距离更近，网络状况、时延等情况更优，计算迁移服务质量也会更好。虽然终端空闲资源总数庞大，但是与集中式云端资源相比，终端资源分布得比较零散，不方便部署，因此我们使用 swarm 容器集群来对

提供迁移服务的终端空余资源进行管理和应用。另一方面，我们修改 Web 运行环境代码，设计对 Web 应用开发者透明的计算迁移系统，将 Web Worker 迁移到服务端进行运行。

4.3.1 系统设计，结构，模块，流程

基于边缘容器的 Web Worker 计算迁移系统分为客户端与服务端。图4.1为 Web Worker 计算迁移系统的模块图。客户端也就是用户终端上的 Web 运行环境，可以接收用户请求，通常会根据用户请求生成以多个 Web Worker 形式承载的计算任务。当用户终端资源不足以完成该计算任务，或者完成时间较长，希望能够缩短计算时间提高用户体验的时候，客户端程序可以将计算任务以 Web Worker 的形式迁移到服务端来完成。服务端是一个能够接收计算迁移任务请求的应用程序，并在本地生成 Web Worker 运行迁移过来的计算任务。服务端通常以容器的形式部署在周围有空余计算资源的其他终端上，并受整个边缘容器集群管理。需要指出的是，边缘容器集群中是由多个用户终端设备组成，每个终端设备在有计算任务的时候都可以成为客户端，向其他设备迁移计算任务，在没有计算任务的时候也都可以成为服务端，接受其他设备迁移过来的计算任务。

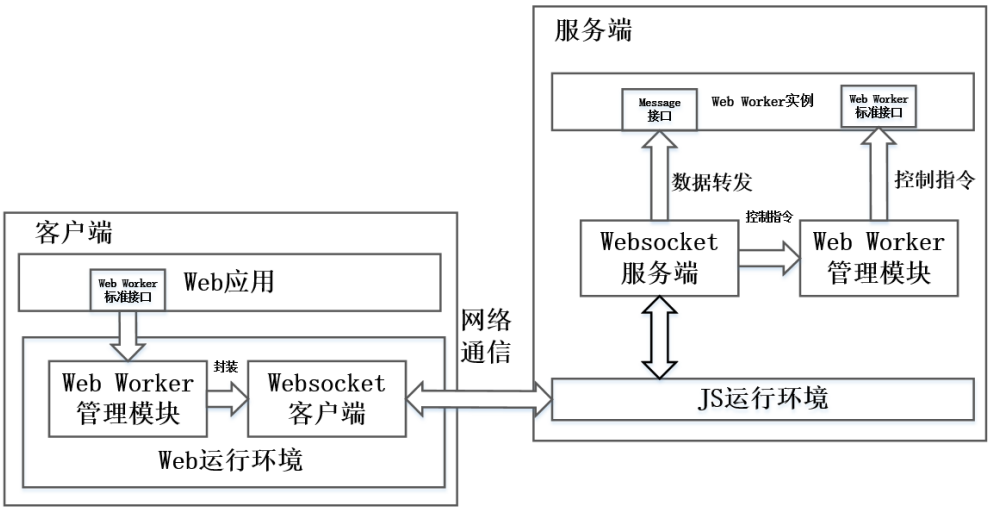

图 4.1 Web Worker 透明计算迁移

4.3.2 透明迁移的客户端

客户端包含 Web 应用和重写的 Web 运行环境两个部分。Web 运行环境在本研究中主要指运行在用户终端上的 Web 浏览器，它能够为 HTML5 及 Javascript 提供标准支持。因为设计的计算迁移系统为透明迁移，所以 Web 应用只需要调

用标准的 Web Worker 接口就可以生成 Web Worker 并与其进行通信，而不需要对 Web 应用本身做任何特殊的改动，Web 应用开发者也不需要了解底层是如何实现的。底层的 Web 运行环境被重新改写，主要包含 Web Worker 管理端和 websocket 通信客户端。当系统决定要进行计算迁移的时候，Web Worker 管理端在收到上层 Web 应用通过标准接口传来的 Web Worker 生成的请求的时候，会将该请求进行翻译并重新封装，Web 运行环境中的 websocket 通信客户端模块会通过 websocket 将封装后的请求发送到服务端进行处理。

在后续的通信中，客户端与运行在服务端的 Web Worker 也要进行很多信息交互，这些通信也是以 websocket 的形式进行的，因此需要为 websocket 设置几种通信标志，来区分不同的通信类型。本研究中主要设置三种通信标志：

- ESTABLISH: 客户端请求服务端创建新的 Web Worker，由服务端直接进行处理
- COMMUNICATE: 客户端向服务端发送的通信信息，由服务端接收后转发给服务端对应的 Web Worker
- TERMINATE: 客户端请求服务端停止对应的 Web Worker，由服务端直接进行处理

添加通信标志的操作是在客户端中的 websocket 通信客户端重新封装请求的过程中实现的。这些通信标志可以让服务端区分该信息是控制类型（ESTABLISH、TERMINATE）还是数据类型（COMMUNICATE），并根据通信类型的不同采取不同的处理方式。

4.3.3 基于容器的服务端

图4.1中右侧为服务端程序的模块。服务端程序包括底层的 Javascript 运行环境、websocket 通信服务端以及 Web Worker 管理端。Javascript 运行环境用来提供 Javascript 标准接口，支撑整个服务端程序以及 Web Worker 的运行。websocket 通信服务端用来接受客户端发来的消息，并且对其解封装，根据通信标志的不同，做相应的处理。Web Worker 管理端，接受通信服务端传来的指令，调用 Web Worker 标准接口，实现对 Web Worker 生命周期的管理，包括 Web Worker 的创建、信息交互及销毁。

图4.2为整个系统的整体架构图。服务端程序是以容器的形式部署在边缘 Docker Swarm 集群上的。这个过程需要使用 Dockerfile 生成安装有对应运行环境及服务端程序的 Docker 镜像，并将生成的镜像上传到本地镜像仓库中。接下

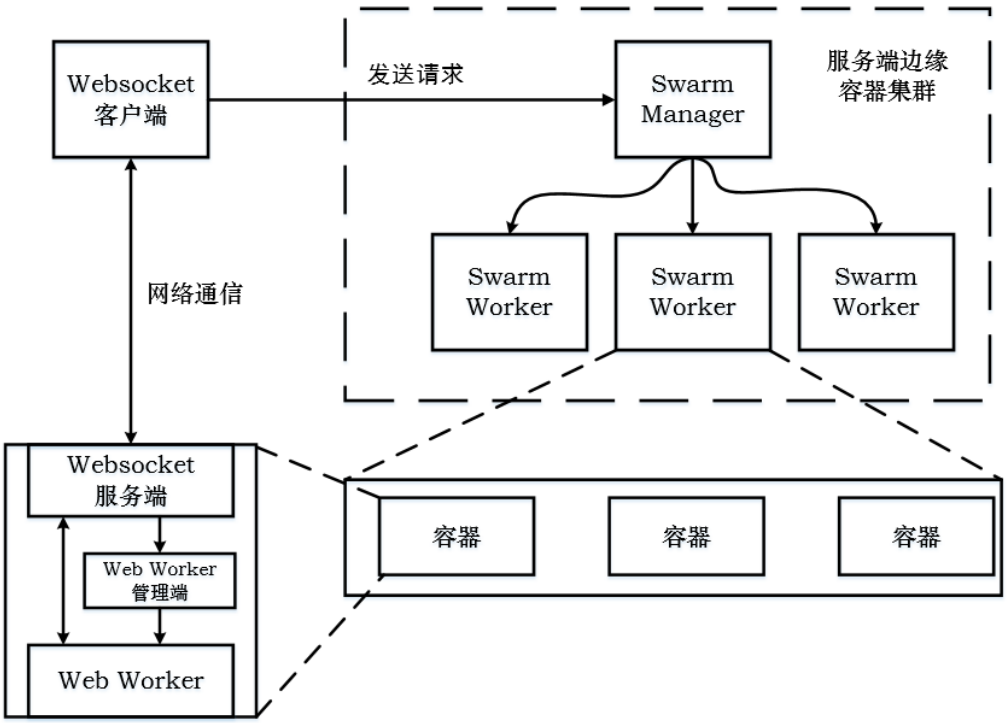

图 4.2 透明计算迁移系统服务端结构图

来需要由 Docker Swarm 集群从镜像仓库中拉取对应镜像，按照对资源以及服务规模的需求生成相应的迁移服务，并通过 Docker Swarm 的调度器在合适的终端节点上启动相应数量的副本实例，也就是运行着服务端程序的容器，打开对应端口，为用户终端提供计算迁移服务。每个容器在哪个节点上运行是由 Docker Swarm 调度器上执行的调度策略来决定的。

计算迁移的过程被分为 4 个阶段，运行时准备阶段，网络连接阶段，数据传输阶段，任务执行阶段 [?]。在运行时准备阶段中，服务端使用 Dockerfile 生成安装有对应运行环境及服务端程序的 Docker 镜像，或者直接从本地镜像仓库中下载对应镜像，创建对应服务，在节点上部署对应容器，并对外暴露服务端口。在网络连接阶段中，客户端接收到用户请求，Web 应用根据用户请求向底层 Web 运行环境发送 Web Worker 的创建请求，客户端的 Web Worker 管理端会将带有创建 Web Worker 所需要的信息的 JS 文件的 URL 加上一个 ESTABLISH 标志，封装后交给客户端的 websocket 客户端。此时 websocket 客户端会向服务端暴露出的服务 IP 和端口请求建立连接，通过服务端 Docker Swarm 的调度以后，客户端与某个容器上运行的服务端的 websocket 服务端端建立网络连接。websocket 服务端在接收到消息后会进行解封装，读取通信类型标志并根据消息内容中包含

的 URL 下载对应 JS 文件。然后服务端上的 Web Worker 管理端会在对应容器中运行该 JS 文件，创建 Web Worker。在数据传输阶段中，websocket 客户端发送的信息带有 COMMUNICATE 通信标志，代表信息中包含的内容是 Web 应用发送给 Web Worker 的数据信息，websocket 服务端解析到该标志后，会直接将信息转发给在容器中运行的 Web Worker。在任务执行阶段中，运行在边缘容器中的 Web Worker 会利用容器中配置的资源，执行对应计算任务，并将计算结果通过网络返回给用户的客户端设备。当该客户端接收到某个 Web Worker 返回的全部数据，则会向该服务端发送带有 TERMINATE 标志的信息，服务端在接收到该消息后，会销毁对应容器中运行的 Web Worker，释放相关资源。此时该容器并不会随之销毁，而是可以等待下一次计算迁移任务，直到整个 Docker Swarm 集群对该容器的生命周期进行调整。

4.4 实验结果及分析

4.4.1 实验配置

本研究中使用树莓派设备来模拟用户终端设备及边缘终端设备，搭建基于容器的 Web Worker 透明边缘计算迁移系统，并测试其对于用户服务体验的提升效果。本实验中的客户端以及服务端集群使用的树莓派型号为 raspberry pi3，配置为 1G 内存，4 核处理器。服务端 Docker Swarm 集群由局域网内的 4 个树莓派组成，其中一个作为 master，另外三个作为 worker，每个树莓派节点上都可以作为计算迁移服务的执行节点。另外为了与基于容器的边缘集群的性能做对比，我们还使用了一台局域网内的 Dell PowerEdge R 730 服务器作为服务端来进行实验。本实验中客户端的 Web 运行平台为 chromium 浏览器，服务端的 JS 执行环境为基于 V8 引擎的 Node.JS。

本实验中的测试用例为利用一个利用 Web Worker 的图形渲染 Web 应用 Ray Tracing[http: / / nerget．com / rayjs-mt / rayjs．html]。在这个应用中，Web 应用需要渲染加载一张图片，具体过程是将图片切割成若干份（本实验中为 20 份），启动一定数量的 Web Worker 来进行渲染加载。这个测试用例中，图片总大小是一样的，也就是总的任务量一定，使用的 Web Worker 数量越多，每个 Web Worker 计算的任务量就越少，同时 Web Worker 数量与切割份数不一定相同，可能出现一个 Web Worker 需要处理多份任务的情况。在本实验中，Ray Tracing 应用可以不做任何修改，直接运行在客户端树莓派的 chromium 上，然后改写后的浏览器将计算任务迁移到边缘树莓派集群或服务器上，创建 Web Worker 并进行计算，最

后将结果返回给客户端，浏览器将图片显示给用户。本实验中对基于容器的 Web Worker 透明边缘计算迁移系统服务性能的评价指标为 Web Worker 的总体执行时间，即从第一个 Web Worker 开始创建，到最后一个 Web Worker 执行完成所消耗的时间。

4.4.2 实验结果

为了测试不迁移、迁移到服务器以及迁移到边缘树莓派集群上的服务性能，我们做了实验一，图片大小为 200*200，切分为 20 份 200*10 的小图片，交给不同数量的 Web Worker 来进行计算，实验结果如图？所示，其中横坐标表示每一次实验中的 Web Worker 数量，纵坐标表示该次实验中的 Web Worker 执行总时间。

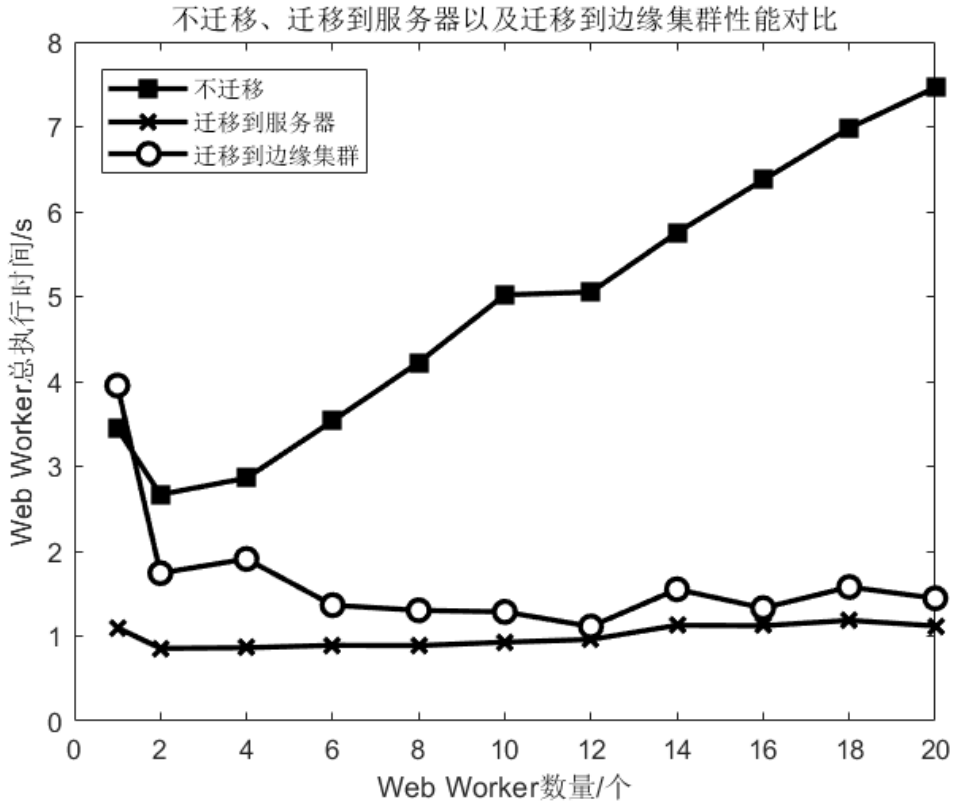

图 4.3 实验一：200*200 图片渲染

从图4.3中可以看出，不使用计算迁移系统的时候，执行总时间最长，服务性能最差，而且随着 Web Worker 数量的增加，性能变差得非常明显，让用户的服务体验变得非常差。迁移到边缘树莓派集群上的时候，相比不迁移的情况，性能提升明显，虽然执行时间比迁移到服务器的情况稍差，但相差不是很大，还是能够保持在用户可接受范围内。另外随着 Web Worker 数量的增加，迁移到边缘

树莓派集群的执行总时间会慢慢增加，但增长趋势比较缓慢，不会出现不迁移时候的明显变差的情况。迁移到服务器上的时候，执行时间最短，尤其是在 Web Worker 数量为 1 的时候，优势非常明显，但这更多的是基于服务器本身性能要远远好于单个树莓派的原因。在实际应用场景中，使用服务器来提供计算迁移服务的成本要远远高于使用边缘终端设备，而且实际场景中的服务器通常位于云端数据中心，距离用户终端设备较远，网络通信开销会增大，同时还会受限于网络状况，服务性能很可能达不到这么好的效果。

为了测试透明迁移到边缘树莓派集群与非透明迁移到集群的服务性能差异，我们做了实验二，图片大小为 400*400，切分为 20 份 400*20 的小图片，交给不同数量的 Web Worker 来进行计算，实验结果如图？所示，其中横坐标表示每一次实验中的 Web Worker 数量，纵坐标表示该次实验中的 Web Worker 执行总时间。

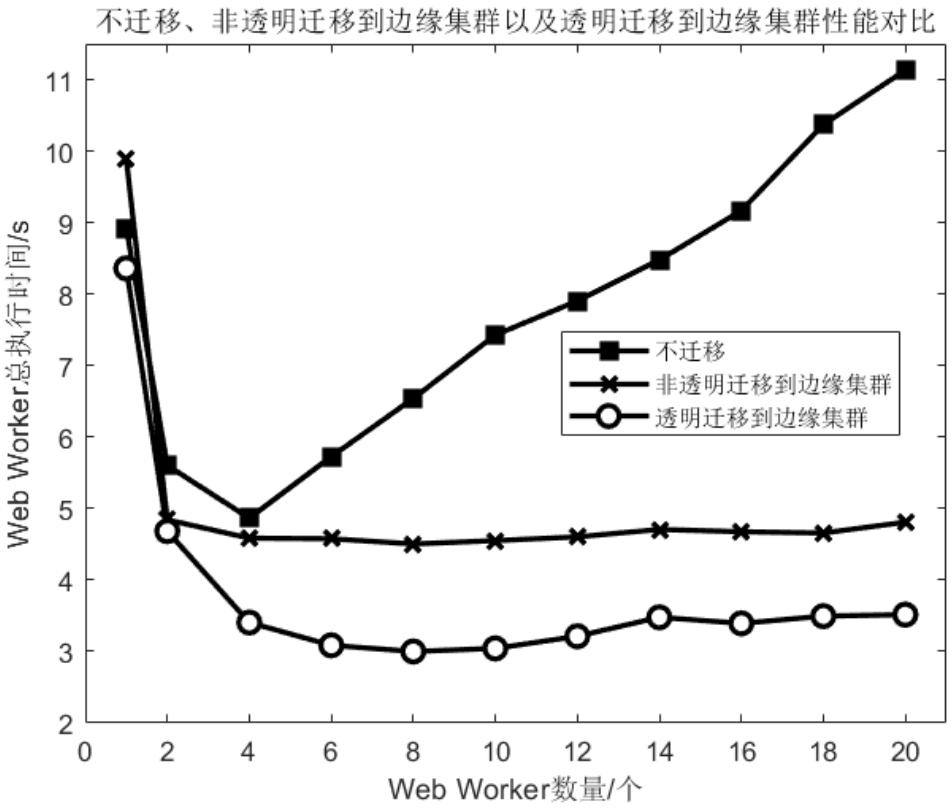

图 4.4 实验二：400*400 图片渲染

从图4.4中可以看出，相比实验一，图片大小变大，总任务量变多，总的计算时间也相应增加。不使用计算迁移系统的时候，执行总时间还是最长，服务性能最差，而且随着 Web Worker 数量的增加，性能变差得非常明显，让用户的服务体验变得非常差。另外值得注意的一点是，不适用计算迁移系统的时候，在使用

4 个 Web Worker 来执行任务的时候，总执行时间最短，这可能与树莓派设备是 4 核处理器有关，当 Web Worker 数量小于 4 个的时候，会出现树莓派的计算能力没有被充分利用的情况，当 Web Worker 数量大于 4 个的时候，会出现各个线程之间抢占 CPU 而导致切换时间大大增加影响总执行时间的情况。使用计算迁移后，总的执行时间都会大大缩短，当 Web Worker 数量多于 4 个的时候，边缘树莓派集群中多设备多核处理器的优势就体现出来了，当 Web Worker 数量增加的时候，总执行时间也不会随之增加很多，稳定在一个比较短的时间内。对比透明计算迁移方式和非透明计算迁移方式的实验结果，可以看出，透明计算迁移方式的总执行时间也要比非透明计算迁移方式的总执行时间要短一些，整体服务效果更好。

4.5 本章小结

本章。。。。。

第 5 章　资源受限终端任务调度策略

5.1　引言

任务调度问题是指系统在同时接收到多个服务请求任务的时候，将这些任务合理地分配给多个智能终端上的容器进行处理，由于不同的容器所在智能终端的处理能力不同，每个容器所占用的资源也不同，导致不同的调度方案结果会有差异，需要针对在容器化智能终端协同服务场景下的一些特点来进行取舍和进一步的优化，以追求对于终端资源利用的最大化。在该任务场景下，最常见的用户需求就是实时性需求，也即要求任务能够被快速响应、快速执行、且执行结果能够快速回传给用户，因此最小化任务完成时间是任务调度问题最主要的目标。除此之外，需要考虑的因素还涉及硬件设备功耗、分布式设备负载均衡度等。

5.2　相关工作

5.2.1　传统任务调度问题求解方法

任务调度问题是 NP-hard 问题，已有的任务调度算法主要分为两大类，一类是基于局部贪心策略的算法，另一类是启发式的智能搜索方法。贪心策略的算法，例如优先调度任务量最重的算法，或者优先调度计算时间最短的算法，其调度结果很容易陷入局部最优解，不能够合理利用所有资源。启发式算法则是确定搜索策略，通过迭代、评价等方式，逐步逼近全局最优解。

任务调度问题是将多个任务计划到约束下的多个节点的问题。任务调度问题是一个优化问题。应用多种算法来解决任务调度问题。基于最佳资源选择 (BRS) 的算法, 如最大最小、最小、苦难等, 是解决任务调度问题的传统方法。一些元启发式算法, 如 PSO 和基于 pso 的改进算法, 是处理任务调度问题的新方法。

5.2.2　启发式算法求解方法

近年来, 对任务调度问题进行了大量的研究。随着研究的进行, 许多元启发式算法被用来处理复杂的优化问题。元启发式算法具有简单的操作和较少的开销, 能够找到全局最优。

启发式算法中，如遗传算法（Genetic Algorithm），是模拟达尔文生物进化论的自然选择和遗传学机理的生物进化过程的计算模型，通过模拟自然进化过程

搜索最优解。遗传算法将一个调度方案表示为一个染色体，并利用交叉、变异等运算符实现调度方案的进化，最终得到的结果比较好，但是收敛速度较慢且运算量较大。模拟退火算法（Simulated Annealing Algorithm），仿照固体退火原理，是一种基于概率的算法，从某一较高初温出发，伴随温度参数的不断下降, 结合概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。Kennedy 和 Eberhart 在 1995 年提出粒子群算法（Particle Swarm Optimization），该算法最初是受到飞鸟集群活动的规律性启发，进而利用群体智能建立的一个简化模型，通过追随当前搜索到的最优值来寻找全局最优。

许多元启发式算法被引入到优化问题中。GA 是戈德伯格在 1988 年提出的经典元启发式算法, 它将自然选择理论引入到优化过程中, 包括突变、交叉和选择 citefonseca1995an,Whitley1994, Tanese1989DGA, Ng 广州市 2018。虽然 GA 的性能相当好, 但遗传算法的操作过于复杂, 无法实现, 不适合某些情况。一些元启发式算法的灵感来自昆虫、鱼类、鸟类和其他群体生物的自然行为。粒子群算法 (PSO) 是肯尼迪 1995 年提出的经典元启发式算法。PSO 的原理简单, 性能显著 1995 年的柠檬素颗粒, Liao2007, Gomathi2013。蚁群优化算法 (ACO) 是受蚂蚁在鸟巢和食物来源之间自然觅食行为的启发。ACO 利用化学信息素在中国的人群中进行交流。

最近提出了一些新的元启发式优化算法。关于改进这些算法的研究并不多。蚂蚁狮子优化器提出 (ALO) 在 2015 年是受狮子 citemirjilili2013 蚂蚁的狩猎行为的启发。鲸鱼优化算法 (WOA) 于 2016 年提出。WA 模拟鲸鱼的自然狩猎行为。2016 年提出的蜻蜓算法 (DA) 是受蜻蜓群在自然界中的静态和动态行为的启发。

2017 年, Shahrzad Saremi 和 Seyedali Mirjalili 提出了一种新的元启发式优化算法, 称为蝗虫优化算法 (GOA)。GOA 利用群体内部的相互作用和蜂群外的风的影响来模拟蝗虫群的迁徙行为, 寻找目标食物 citesare2013 2017 蝗虫。GOA 算法利用群智能, 通过在蝗虫群之间分享经验, 确定搜索方向, 找到最佳或近似的最佳位置。GOA 还使用了具有多个迭代的进化方法, 以提高群智能的效率。

开发了一些基于 GOA 的改进算法。OBLGOA 是由艾哈迈德 • 埃维斯在 2018 年提出的 ceewees 2013 改进。根据目前的搜索位置, 引入了基于对立面的学习策略, 以生成相反的解作为候选方案。OBL 策略可以提高算法的收敛速度, 但由于缺乏随机性, 改进有限。桑卡拉阿罗拉提出了混沌蝗虫优化算法在 2018 年 柠檬酸乱糟糟。将混沌映射应用到算法中, 提高了 GOA 的性能。采用 10 幅混沌映

射来评价混沌理论的影响。结果并不是特别理想, 因为混沌因素在处理许多基准函数时并不合适。提出了一种基于 GOA 的新算法来解决优化问题和任务调度问题。

5.3 GOA 算法

5.3.1 GOA 算法背景

5.3.2 GOA 算法数学模型

蝗虫优化算法模拟蝗虫的昆虫群行为。蝗虫蜂拥而至, 远距离迁徙, 寻找一个有食物的新栖息地。在这个过程中, 蝗虫之间的互动在蜂群内部互相影响。风的力量和蜂群外的重力影响蝗虫的轨迹。食物的目标也是一个重要的影响因素。

受上述三个因素的影响, 移民过程分为勘探和开发两个阶段。在勘探阶段, 鼓励蝗虫快速、突然地移动, 寻找更多潜在的目标区域。在开发阶段, 蝗虫往往在当地移动, 以寻找更好的目标地区。蝗虫自然实现了勘探开发寻找食物来源的两种迁徙趋势。此过程可以抽象为优化问题。蝗虫群被抽象为一群搜索代理。

Seyedali Mirjalili 在文献 [] 中提出了蝗虫群体迁移的数学模型。具体的模拟公式如下：

$$X_i = S_i + G_i + A_i \tag{5.1}$$

这里变量 X_i 是第 i 个搜索单位的位置，变量 S_i 代表蝗虫集群内部搜索单位间社会交互对第 i 个搜索单位的影响因子，变量 G_i 代表蝗虫集群外部重力因素对第 i 个搜索单位的影响因子，变量 A_i 代表风力的影响因子。变量 S_i 的定义公式如下：

$$S_i = \sum_{j=1, j\neq i}^{N} s(d_{ij})\widehat{d_{ij}} \tag{5.2}$$

这里变量 d_{ij} 代表第 i 个搜索单位和第 j 个搜索单位之间的欧式距离，计算方法如下：

$$d_{ij} = |x_j - x_i| \tag{5.3}$$

变量 $\widehat{d_{ij}}$ 代表第 i 个搜索单位和第 j 个搜索单位之间的单位向量，计算方法如下：

$$\widehat{d_{ij}} = \frac{x_j - x_i}{d_{ij}} \tag{5.4}$$

s 是一个函数，用于计算蝗虫集群之间的社会关系影响因子，该函数定义如下：

$$s(r) = fe^{\frac{-r}{l}} - e^{-r} \tag{5.5}$$

这里 e 是自然底数，变量 f 代表吸引力因子，参数 l 代表吸引力长度。当应用于解决数学优化问题的时候，为了优化数学模型，公式 1 中需要加入一些适当的改动。代表集群外部影响因子的变量 G_i 和 A_i 需要被替换为目标食物的位置。这样计算公式就变成了如下的样子：

$$x_i = c\Big(\sum_{j=1, j\neq i}^{N} c\frac{u-l}{2}s(|x_j - x_i|)\frac{x_j - x_i}{d_{ij}}\Big) + \widehat{T_d} \tag{5.6}$$

这里参数 u 和参数 l 分别代表搜索空间的上界和下界。变量 $\widehat{T_d}$ 是目标食物的位置，在优化问题的数学模型中代表所有搜索单位在整个搜索过程中所能找到的最优的解的位置。另外，参数 c 是搜索单元的搜索舒适区控制参数，改变参数 c 的大小可以平衡搜索过程中的“开拓”和“探索”两个阶段。参数 c 的计算方式如下：

$$c = cmax - iter\frac{cmax - cmin}{MaxIteration} \tag{5.7}$$

这里参数 *cmax* 和参数 *cmin* 分别是参数 c 的最大值和最小值，参数 *iter* 代表当前的迭代次数，参数 *MaxIteration* 代表最大迭代次数。

在优化问题的求解过程中，公式 4 作为演进公式，被不断循环迭代来寻找最优解，直到达到迭代终止条件为止。通常迭代终止条件为达到预设的最大迭代次数，或者所得到的最优解满足预设的最优解条件。在本研究所涉及到的优化问题中，迭代终止条件均为达到预设的最大迭代次数。在迭代演进的过程结束后，该算法可以得到一个近似的最优解的位置以及相应的最优解的值。

蝗虫优化算法的算法伪代码如1所示：

5.3.3 蝗虫算法优缺点分析

GOA 是最近受到蚱蜢自然迁移行为启发的元启发式算法。虽然 GOA 具有简单的理论基础并且易于实现，但 GOA 的性能更优越。原始的 GOA 改变了蚱

算法 1 蝗虫优化算法

initialize the swarm and set the position boundaries u and l
initialize the factors including cmax, cmin, MaxIteration
initialize all the search agents position with random origin matrix
calculate the target fitness and mark the target position
while $(iter < MaxIteration$ and $target fitness > destination fitness)$ **do**
    calculate $d_i j$ and $\widehat{d_i j}$ by equation(2)
    calculate $s(d_i j)$ by equation(3)
    update x_i by equation(4)
    calculate the fitness
    if current fitness is better than target fitness **then**
        update the target fitness and the target position
    end if
    $iter = iter + 1$
end while
Return target fitness and target position

蝗的舒适区域，这可以使目标通过迭代收敛到全局最优解。与一些经典算法相比，GOA 的收敛速度要快得多。GOA 可显着提高蝗虫的平均适应度，改善蝗虫的初始随机种群。虽然 GOA 具有这些优点，但它也存在一些阻碍算法获得更好解决方案的缺点。在对 GOA 公式进行一些理论分析和用 MATLAB 代码进行的一些实验之后，给出了 GOA 的几个缺点。

首先，GOA 使用线性递减参数使搜索过程收敛，这很难区分过程的两个阶段，即开发阶段和探索阶段。其次，在搜索过程的早期阶段的每次迭代期间，最佳解决方案的位置急剧波动。似乎最终的最佳解决方案只受搜索过程后期的影响，无论前一阶段的结果如何。GOA 理论无法充分利用所有搜索迭代。当最大迭代次数增加时，GOA 的最佳适应性不是很突出，例如，从实验结果中发现的 500 到 1500。最后，搜索过程很容易陷入局部最优。GOA 易于实现的优势无助于摆脱局部最优，这可能是 GOA 的劣势。

5.4 带随机跳出机制的动态权重蝗虫优化算法 (DJGOA)

5.4.1 动态权重

GOA 使用线性递减参数来限制搜索空间并使所有搜索代理移动到目标位置。线性递减参数不能增强搜索过程的两个阶段的影响，即开发阶段和探索阶段。在探索阶段，GOA 无法在目标搜索区域周围快速收敛，搜索代理只是在整个搜索空间中游荡，这无法为后期搜索阶段奠定坚实的基础。在开发阶段，参数通常使搜索代理滑过局部最佳位置，就像搜索代理超速运动一样。

线性递减参数机制无法使算法充分利用整个迭代。引入动态权重参数机制以提高算法的利用率。搜索进度分为三个阶段，即早期阶段，中间阶段阶段和后期阶段。在早期阶段，目标位置的权重应该更高，以使搜索过程快速收敛。在中间阶段，参数应该是稳定的，以使算法探索搜索空间。在后期阶段，搜索代理中的重力的权重应该更小，以深入利用局部最优解位置。具有动态权重参数的 GOA 公式表示为算法 [1]。新的迭代方程如下：

$$x_i = m * c\Big(\sum_{j=1,j\neq i}^{N} c\frac{u-l}{2}s(|x_j - x_i|)\frac{x_j - x_i}{d_{ij}}\Big) + \widehat{T}_d \tag{5.8}$$

where m is the dynamic weight parameter to adjust the search process. To correspond to the feature of the three phases of the search process, the parameter m is set as follows:

$$m = \begin{cases} 0.5 - \frac{(0.5-0.1)*iter}{MaxIteration*0.2} & 0 < iter \leq MaxIteration * 0.2 \\ 0.1 & MaxIteration * 0.2 < iter \leq MaxIteration * 0.8 \\ 0.05 & MaxIteration * 0.8 < iter \leq MaxIteration \end{cases} \tag{5.9}$$

5.4.2 随机跳出机制

原始的 GOA 算法没有使用跳跃机制，并且所有搜索代理仅根据社交互动和目标食物吸引力的影响而移动。原始 GOA 的机制可能导致算法陷入局部最优位置。

引入随机跳跃策略以帮助 GOA 算法提高跳出局部最佳位置的概率。参数 p 被设置为跳跃阈值。在迭代结束之前，将当前最佳适合度与最后一次迭代的最佳适应度进行比较。如果当前最佳适应度和最后一个最佳适应度的比率高于阈值 p，则可以假设该算法没有找到更好的解决方案，并且随机跳跃机制应该启动。

根据随机初始化规则在最佳位置周围生成新的搜索代理。随机初始化规则如下：

$$tempPos = curPos * ((0.5 - rand) * iniRan + 1); \tag{5.10}$$

其中 *tempPos* 是新搜索代理的位置，*curPos* 是当前最佳解决方案的位置。*iniRan* 是管理随机跳跃边界的初始化范围参数。如果 *iniRan* 更高，则算法的全局搜索能力就像模拟退火算法一样得到增强。

5.4.3 DJGOA 算法流程

本文提出了一种随机跳跃动态权重蝗虫优化算法（DJGOA）。DJGOA 的过程分为初始阶段，参数设置阶段，计算阶段和适应度更新阶段四个阶段。在初始化阶段，设置包括位置边界在内的一些因素，并在边界内随机初始化原始群体位置。在参数设置阶段，搜索过程进入迭代循环，动态权重参数根据当前迭代由 emph equation（7）设置。在计算阶段，群体内的社会交互由 emph equation（6）计算。在适应度更新阶段，如果当前适应度优于历史的最佳目标适应度，则更新目标解决方案的适合度。如果当前适应度与最后一次迭代的适应度之比高于之前设置的阈值 emph p，则使用随机跳转策略生成新的搜索代理以尝试跳出局部最优位置通过 emph equation（8）。具有随机跳跃的动态权重蝗虫优化算法（DJGOA）的伪代码显示为算法2。

5.4.4 实验结果

5.4.4.1 实验设置

为了评估所提算法的性能，进行了一系列实验。我们将所提出的 DJGOA 与 GOA 的原始算法（最近的 Dragonfly 算法（DA）的元启发式算法）和粒子群算法（PSO）的经典启发式算法进行了比较，该算法通过在 cite saremi2017grasshopper 中使用的 13 个基准函数。13 个基准功能分为两种类型。函数 emph f1-f7 是单峰函数，它测试了算法的收敛速度和局部搜索能力。函数 emph f8-f13 是多模函数，当存在多个局部最优时，它测试算法的全局搜索能力。表5.1中列出了 7 个单峰测试函数的详细信息和表达式, 表5.2中列出了 6 个多峰测试函数的详细信息和表达式。

为了解决测试功能，采用了 30 个搜索代理，最大迭代次数设置为 500. 每个算法的每个实验进行 30 次以产生统计结果。GOA，DA 和 PSO 的参数设置为论文引用引用 mirjalili2016dragonfly 引用文件。 citemi2017grasshopper。

算法 2 带随机跳出机制的动态权重蝗虫优化算法

initialize the swarm and set the position boundaries u and l
initialize the factors including cmax, cmin, MaxIteration and iniRan
initialize all the search agents position with random origin matrix
calculate the target fitness and mark the target position
while $(iter < MaxIteration$ and $targetfitness > destinationfitness)$ **do**
    set the dynamic weight parameter m by equation(7)
    calculate d_ij and $\widehat{d_ij}$ by equation(2)
    calculate $s(d_ij)$ by equation(3)
    update x_i by equation(6)
    calculate the fitness
    if current fitness is better than target fitness **then**
        update the target fitness and the target position
    end if
    if $(currentfitness/lastfitness) > p$ **then**
        generate a new search agent by equation (8)
        if the new fitness is better than target fitness **then**
            update the target fitness and the target position
        end if
    end if
end while
Return target fitness and target position

表 5.1 F1-F7 单峰测试函数

Function	Dim	Range	f_{min}
$F_1(x)=\sum_{i=1}^{n}x_i^2$	30	[-100,100]	0
$F_2(x)=\sum_{i=1}^{n}\lvert x_i\rvert+\prod_{i=1}^{n}\lvert x_i\rvert$	30	[-10,10]	0
$F_3(x)=\sum_{i=1}^{n}(\sum_{j=1}^{i}x_j)^2$	30	[-100,100]	0
$F_4(x)=max_i\{\lvert x_i\rvert,1\le i\le n\}$	30	[-100,100]	0
$F_5(x)=\sum_{i=1}^{n-1}[100(x_{i+1}-x_i^2)^2+(x_i-1)^2]$	30	[-30,30]	0
$F_6(x)=\sum_{i=1}^{n}([x_i+0.5])^2$	30	[-100,100]	0
$F_7(x)=\sum_{i=1}^{n}ix_i^4+random[0,1)$	30	[-1.28,1.28]	0

5.4.4.2 实验结果

表5.3中给出了 30 次实验的平均值，标准偏差，最佳值和最差值，以描述 DJGOA，GOA，DA 和 PSO 的性能。 从表5.3可以看出，DJGOA 的性能优于其他 3 种算法。所有 13 个基准函数的 10 个结果表明，DJGOA 的平均值比其他几个数量级的表现要好。而对于其他三个测试功能的结果，DJGOA 的表现略差于其他测试功能。实际上，DJGOA 可以通过 GOA，DA 和 PSO 获得相同数量级的结果。对于上面提到的三个测试功能中的两个，DJGOA 可以在获得最佳价值方面做得更好。实验结果表明，DJGOA 在标准偏差和最差值方面表现较好，表明 DJGOA 在搜索中更稳定。对于 12 个实验结果，DJGOA 可以获得更好的最佳值结果，这意味着 DJGOA 能够比其他 3 种算法更好地找到最佳适应度。

对于单峰测试功能，GOA 的性能非常好，DJGOA 可以大大提高 GOA 的性能。DJGOA 将搜索结果提高了几个数量级。对于多模式测试功能，DJGOA 可以帮助算法提高很多，特别是在寻找更准确的目标适应度时。总之，跳跃策略可以帮助算法进行大量的搜索，并且 DJGOA 在全局搜索和本地搜索中具有比 DA，GOA 和 PSO 更好的搜索最佳能力。

表 5.2 F8-F13 多峰测试函数

Function	Dim	Range	f_{min}
$F_8(x)=\sum_{i=1}^{n}-x_i\sin(\sqrt{\lvert x_i\rvert})$	30	[-500,500]	$-418\times Dim$
$F_9(x)=\sum_{i=1}^{n}[x_i^2-10\cos(2\pi x_i)+10]$	30	[-5.12,5.12]	0
$F_{10}(x)=-20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2})-$ $exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i))+20+e$	30	[-32,32]	0
$F_{11}(x)=\frac{1}{4000}\sum_{i=1}^{n}x_i^2-\prod_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}})+1$	30	[-600,600]	0
$F_{12}(x)=\frac{\pi}{n}\{10\sin(\pi y_1)+\sum_{i=1}^{n-1}(y_i-1)^2[1+10\sin^2(\pi y_{i+1})]+$ $(y_n-1)^2\}+\sum_{i-1}^{n}u(x_i,10,100,4)+\sum_{i=1}^{n}u(x_i,10,100,4)$ $y_i=1+\frac{x_i+1}{4}$ $u(x_i,a,k,m)=\begin{cases}k(x_i-a)^m x_i>a\\0-a<x_i<a\\k(-x_i-a)^m x_i<-a\end{cases}$	30	[-50,50]	0
$F_{13}(x)=0.1\{\sin^2(3\pi x_1)+\sum_{i=1}^{n}(x_i-1)^2[1+\sin^2(3\pi x_i+1)]+$ $(x_n-1)^2[1+\sin^2(2\pi x_n)]\}+\sum_{i=1}^{n}u(x_x,5,100,4)$	30	[-50,50]	0

表 5.3 DJGOA 算法在 13 个测试函数上的实验结果

测试函数	类型	DJGOA	GOA	DA	PSO
F1	avg	**1.66E-46**	**1.36E-07**	0.0808	0.2256
	std	**9.08E-46**	**1.79E-07**	0.1056	0.2380
	best	**5.89E-68**	**3.13E-09**	0	0.0139
	worst	**4.97E-45**	**9.43E-07**	0.4929	1.1528
F2	avg	**1.11E-32**	**6.58E-05**	0.0686	0.0737
	std	**3.28E-32**	**3.38E-05**	0.0586	0.0322
	best	**6.61E-44**	**1.18E-05**	0	0.0187
	worst	**1.54E-32**	0.0002	0.2913	0.1847
F3	avg	**6.32E-26**	**1.67E-06**	0.3440	0.9667
	std	**3.46E-25**	**2.33E-06**	0.7516	0.6671
	best	**1.62E-56**	**4.45E-08**	0.0003	0.0395
	worst	**1.90E-24**	**1.16E-05**	3.9916	2.9592
F4	avg	**4.19E-19**	0.0003	0.2142	0.4936
	std	**1.72E-18**	0.0002	0.1760	0.2202
	best	**1.49E-29**	**7.97E-05**	0	0.1757
	worst	**9.24E-18**	0.0009	0.6817	1.0899
F5	avg	9.7473	174.70	131.64	9.9186
	std	40.35	387.13	308.88	7.7555
	best	0.2213	0.1492	0.2252	2.3098
	worst	221.79	1791.14	1537.62	35.84
F6	avg	**2.99E-10**	**1.23E-07**	0.1160	0.2246
	std	**3.72E-10**	**1.05E-07**	0.1381	0.2220
	best	**1.45E-11**	**4.05E-09**	**1.13E-05**	0.0341
	worst	**1.66E-09**	**3.99E-07**	0.5918	0.8912
F7	avg	0.0072	0.0012	0.0018	0.0012
	std	0.0054	0.0011	0.0011	0.0008
	best	0.0005	0.0001	0.0002	0.0002
	worst	0.0203	0.0065	0.0044	0.0031

表 5.3 续表：DJGOA 算法在 13 个测试函数上的实验结果

F8	avg	-1641	-1835	-1951	-1889
	std	239.3	226.0	206.5	203.5
	best	-2297	-2297	-2394	-2178
	worst	-1191	-1348	-1566	-1431
F9	avg	6.1063	9.1868	5.7839	2.9096
	std	9.9879	7.4073	4.5795	1.4782
	best	0	1.9899	1.0143	0.5192
	worst	34.9422	32.8334	17.9153	6.0775
F10	avg	**1.48E-15**	0.9446	0.5148	0.3506
	std	**1.35E-15**	3.5846	0.5990	0.2632
	best	**8.88E-16**	**9.25E-05**	**7.99E-15**	0.0759
	worst	**4.44E-15**	19.5869	1.9511	1.2182
F11	avg	0.0790	0.2252	0.3789	0.3968
	std	0.1171	0.1370	0.1906	0.1401
	best	0	0.0566	0	0.1692
	worst	0.4504	0.5293	0.7729	0.6960
F12	avg	**8.56E-11**	**3.99E-08**	0.0430	0.0043
	std	**9.30E-11**	**3.08E-08**	0.1052	0.0055
	best	**3.76E-12**	**9.10E-09**	0.0001	0.0001
	worst	**4.53E-10**	**1.30E-07**	0.5282	0.0216
F13	avg	**3.22E-10**	0.0011	0.0238	0.0248
	std	**4.13E-10**	0.0034	0.0369	0.0225
	best	**1.91E-11**	**6.37E-09**	**9.97E-05**	0.0033
	worst	**1.66E-09**	0.0110	0.1527	0.1170

5.4.4.3 威尔科克森秩和检验

进行了 wilcoxon 秩和检验的实验，以验证 PSO，GOA 和 DJGOA 的置信水平。根据表5.4中显示的结果，13 个测试函数中 DJGOA 的所有 p 值均小于 0.05，这意味着得到的结论是显着的。

表 5.4 DJGOA、GOA 与 PSO 的威尔科克森秩和检验结果

测试函数	DJGOA	GOA	PSO	测试函数	DJGOA	GOA	PSO
F1	N/A	**1.73E-06**	**1.73E-06**	F8	**2.13E-06**	**7.51E-05**	N/A
F2	N/A	**1.73E-06**	**1.73E-06**	F9	0.8774	**1.73E-06**	N/A
F3	N/A	**1.73E-06**	**1.73E-06**	F10	N/A	**1.73E-06**	**1.73E-06**
F4	N/A	**1.73E-06**	**1.73E-06**	F11	N/A	**1.73E-06**	**1.73E-06**
F5	N/A	**1.73E-06**	**1.73E-06**	F12	N/A	**1.73E-06**	**1.73E-06**
F6	N/A	**1.73E-06**	**1.73E-06**	F13	N/A	**1.73E-06**	**1.73E-06**
F7	**2.56E-06**	N/A	0.0082				

5.4.4.4 Convergence Analysis

所有 13 个基准函数的 DA，GOA 和 DJGOA 迭代的最佳适应值的结果如图 [？] 所示。收敛曲线表明，DJGOA 可以使搜索过程比其他过程更快地收敛，结果也表明动态加权机制可以帮助原始 GOA 算法充分利用每次迭代。

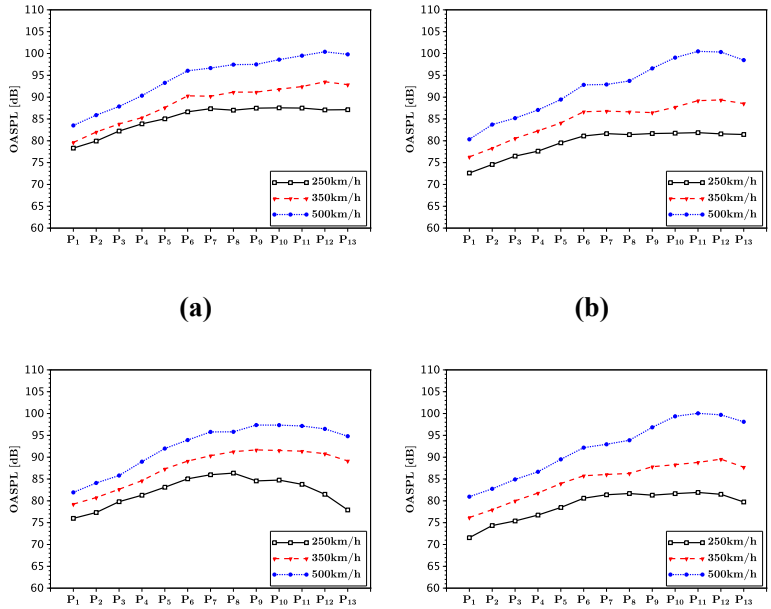

图 5.1 总声压级。(a) 这是子图说明信息，(b) 这是子图说明信息，(c) 这是子图说明信息，(d) 这是子图说明信息。

Figure 5.1 OASPL.(a) This is the explanation of subfig, (b) This is the explanation of subfig, (c) This is the explanation of subfig, (d) This is the explanation of subfig.

5.5 改进蝗虫优化算法 (IGOA)

5.5.1 蝗虫算法不足之处

GOA 具有简单的理论基础，易于实施。另一方面，它有一些阻碍算法获得更好解决方案的缺点。线性减小的舒适区无法帮助原始的 GOA 充分利用每次迭代。由于缺乏随机因素，原始算法几乎没有变化。该算法很容易陷入局部最优。为了解决这些不足，引入了三个改进：非线性舒适区参数，基于 *L acuteevy* flight 的局部搜索机制和随机跳跃策略。本部分明确描述了这三项改进的细节。

5.5.2 非线性舒适区控制参数

原始 GOA 改变了舒适区的半径，这可以使搜索代理通过迭代收敛到全局最优解。GOA 使用 comfort zone 参数来限制搜索空间。在探索阶段，舒适区参数应足够大，以使搜索代理获得足够的空间以快速找到近似最优。在开发阶段，限制因素应该很小，以准确地搜索局部最优并避免搜索代理的超速运动。然而，线性递减因子不能使搜索能力与搜索迭代期间的探索和开发阶段相协调。

为了匹配两个搜索阶段并增强算法的搜索能力，sigmoid 函数被引入到这项工作中。S 形函数是常用的阈值函数和非线性调整因子。它广泛用于信息科学领域。sigmoid 函数的公式如下所示：

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.11}$$

基于 S 形函数变体的非线性舒适区参数建议如下：

$$m = \frac{-0.5}{1 + e^{(-1.5(cx-5)+2\sin(cx))}} + u \tag{5.12}$$

这里参数 u 是调节参数，取值范围应该在 [0,1]。参数 cx 定义如下：

$$cx = \frac{v(iter + 50)}{Maxiteration} \tag{5.13}$$

这里参数 v 是精度调节参数，取值范围应该在 [1,10]。

5.5.3 基于 Levy 飞行的局部搜索机制

GOA 的所有参数都是确定性的。缺乏随机性可能导致搜索迭代期间缺乏创造力，并且每个搜索代理只能搜索确定的位置。将随机因子引入确定性系统是提高其性能的常用方法。

L acuteevy flight 是由 Paul *L acuteevy* cite mirjalili2016dragonfly 提出的随机搜索步骤，它是一种提供随机因子的有效数学方法。由于 *L acuteevy* flight 实现

起来非常复杂，因此如下使用模拟算法：

$$Levy(d) = 0.01 \times \frac{r_1 \times \sigma}{|r_2|^{\frac{1}{\beta}}} \tag{5.14}$$

where d is the dimension of the problem, r_1,r_2 are two random numbers in [0,1] and β is a constant number which is set to 1.5 according to Seyedali in citemirjalili2016dragonfly. σ is calculated as follows:

$$\sigma = (\frac{\Gamma(1+\beta) \times \sin(\frac{\pi\beta}{2})}{\Gamma(\frac{1+\beta}{2}) \times 2^{(\frac{\beta-1}{2})}})^{\frac{1}{\beta}} \tag{5.15}$$

where $\Gamma(x) = (x-1)!$.

Lévy flight 可以在所有搜索代理移动到最佳位置时为其提供视觉。搜索代理可以看到它们周围的小区域。为了扩展搜索代理的搜索半径并增强查找最佳值的能力，提出了一种基于 *Lévy* flight 的本地搜索机制。当位置更新过程结束时，应该通过 *Lévy* 航班以一定的概率调整每个搜索代理的位置。调整公式定义如下：

$$X_i = X_i + 10c \times s_{threshold} \times Levy(dim) \times X_i \tag{5.16}$$

where $s_{threshold}$ is the threshold parameter which control the direction and the probability of the variation. $s_{threshold}$ is calculated as follows:

$$s_{threshold} = sign(x_{trans} - 1) + sign(x_{trans} + 1) \tag{5.17}$$

where $sign(x)$ is the sign function and x_{trans} is a random number in [-3,3].

5.5.4 基于线性递减参数的随机跳出策略

GOA 的基本理论是基本的。该算法只关注收敛到全局最优的过程，忽略了跳出局部最优的机制。因此，GOA 的搜索过程很容易陷入局部最优，搜索无法进一步发展。GOA 易于实现的优势无助于摆脱局部最优，这可能是 GOA 的劣势。

为了提高跳出局部最优的能力，引入了随机跳跃策略。当搜索代理找到最佳位置时，新位置可以替换旧目标位置。如果没有，则随机跳跃方程开始起作用。它描述如下：

$$X_i^{new} = ((0.5 - rand) \times 2 + 1)X_i \tag{5.18}$$

where X_i is the position of the i-th search agent, and X_i^{new} is the new position after random jumping. If X_i^{new} has better fitness, it will replace X_i. Thus action of jumping out occurs successfully.

原始 GOA 的演化公式仅将当前迭代获得的最佳位置作为搜索方向，并且忽略了一些其他有用信息。为了继续新获得的跳跃动作信息的影响，将位置的演化公式转换如下：

$$X_i^{iter+1} = m \times c \times S_i + (1-p)\widehat{T}_d + p \times X_i^{iter} \tag{5.19}$$

其中 p 是用于控制搜索代理位置影响的系数参数。p 在第一次迭代时初始化为 0。如果搜索代理没有跳转或者无法跳出本地最佳位置，p 仍设置为 0，以确保只有 S_i 和 T_d 才能影响进化。当搜索代理成功跳出时，p 被设置为在三次迭代中线性递减为 0 的变量，以继续跳跃行为的影响。经过一些试验，p 的递减步骤设置为 0.3478，本文未对此进行讨论。因此 p 在这项工作中计算如下：

$$p = \begin{cases} p - 0.3478 & p > 0 \\ 0 & p \le 0 \\ 3 \times 0.3478 & when\ jumping\ out\ successfully \end{cases} \tag{5.20}$$

5.5.5 IGOA 流程

本文提出了一种改进的 Grasshopper 算法（IGOA）。IGOA 的过程分为初始阶段，演化阶段，适应性更新阶段和跳跃阶段四个阶段。

在初始化阶段，设置参数，并随机初始化所有搜索代理的原始位置。此部分还计算了最佳目标位置和相应的适应度。搜索循环开始在进化阶段工作。每个搜索代理都通过 *Equation 14* 移动到目标位置。非线性舒适区参数 m 由 emph Equation 7 设置。之后，每个搜索代理通过 *Equation 11* 以特定的概率进行 *Lévy* 航班，并生成新的位置。在更新阶段，计算新位置的适合度。如果新的适应度优于全局适应度，则新的位置可以取代旧的全球目标。如果新的适应度不比全球目标更优化，那么它就会进入跳跃阶段。在此阶段，搜索代理尝试通过 emph Equation 13 跳出局部最优值，并计算新的适应度。如果新的健身状况优于个人健身，新职位可以取代旧的个人职位。参数 p 也由 *Equation 15* 更新。到目前为止，循环中的一次迭代已经完成。在达到最大迭代次数之后，循环结束，并且适应度和目标位置被呈现为最终结果。IGOA 的伪代码显示为3。IGOA 程序的数字框架显示为5.2。

算法 3 Improved Grasshopper Optimization Algorithm

initialize the parameters
initialize the swarm position with random matrix
calculate the original target fitness and mark the target position
while $(iter < MaxIteration$ and $targetfitness > destinationfitness)$ **do**
    set the nonlinear comfort zone parameter m by equation(6)
    update x_i by equation(13)
    x_i conducts $Lévy$ flight by equation(10)
    calculate the fitness
    if current fitness is better than the target fitness **then**
        update the target fitness and the target position
    else
        x_i jumps out by equation(12)
        calculate the fitness
        if current fitness is better than the personal fitness **then**
            update the personal position
        end if
        set parameters p by equation(14)
    end if
end while
Return target fitness and target position

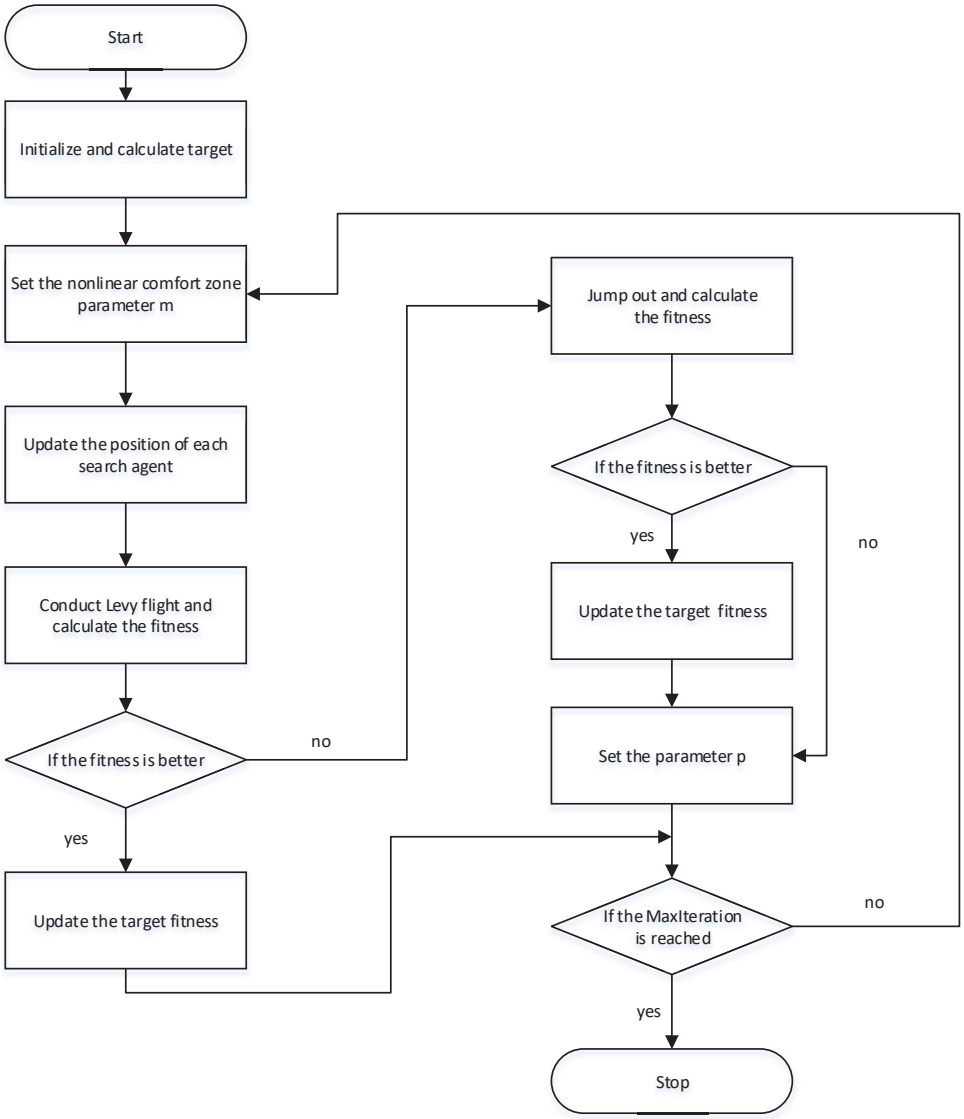

图 5.2 The figure framework of the procedure of IGOA

5.5.6 实验结果

5.5.6.1 实验设置

为了评估拟议的 IGOA 的性能，进行了一系列实验。在这项工作中，IGOA 与 6 个元启发式算法进行了比较，包括原始的 Grasshopper 优化算法（GOA），基于对立的学习 GOA（OBLGOA），最近提出的三种算法，鲸鱼优化算法（WOA），蜻蜓算法（DA）和 Ant Lion Optimizer（ALO），以及经典的启发式算法，粒子群优化算法（PSO）。将其他 6 种算法与 IGOA 相比较的参数设置为论文引用 saremi2017grasshopper，kennedy1995particle，mirjalili2016WOA，mirjalili2016dragonfly，ewees2018 改进，mirjalili2015ant。

在这一部分中，使用了 29 个众所周知的基准函数来测试所提出的 IGOA 的搜索能力。基准测试分为 3 种类型，可以评估算法的不同功能。在5.1中列出的基准 *F1-F7* 是单峰函数，只有一个最佳位置可以估计开发能力。在5.2和5.5中列出的基准 *F8-F23* 是具有多个局部最优的多模函数，可以评估探索能力 cite saremi2017grasshopper。在5.6中列出的基准 *F24-F29* 是复合函数 cite liang2005novel，它在框架下组合了一些基本的测试函数，并且更复杂。他们可以评估摆脱局部最优的表现。在 emph 表 1-4 中，*Dim* 表示基准函数的维度，*Range* 是优化问题的搜索边界，f_min 是函数的最佳适应度。

使用 Matlab 代码进行了与上述 29 个基准函数相关的一系列实验。对于 emph F1-F23，每个算法使用 30 个搜索代理进行 500 次迭代，以进行完整的搜索过程。对于 emph F24-F29，搜索过程包含 100 次迭代。每个搜索过程将针对每个算法重复 30 次以消除意外事件，并且计算一些统计数据，例如平均值（平均值），标准偏差（标准差），最佳适合度 30（最佳）和最差适合度 30（最差），比较算法的性能。此外，进行了 Wilcoxon 秩和检验，计算了 emph p-value 以证明结果的统计显着性。

5.5.6.2 实验结果

emph F1-F7 是单峰函数，只有一个全局最优，可以测试算法的开发能力。如果算法具有超强的利用能力，它可以更准确地搜索并找到更接近全局最优的解。emph F1-F7 的结果显示在表5.7中。

从表5.7可以看出，IGOA 可以在 emph F5-F7 中获得最佳平均结果，并且在 emph F1-F4 中，IGOA 的表现仅比 WOA 差。至于 std 和最差值，IGOA 也可以比 emph F3-F7 中的其他算法表现更好，这表明所提出的算法可以降低获得可怕

表 5.5 fixedmodal functions

Function	Dim	Range	f_{min}
$F_{14}(x)=(\frac{1}{500}+\sum_{j=1}^{25}\frac{1}{j+\sum_{i=1}^{2}(x_i-a_{ij})^6})^{-1}$	2	[-65,65]	0
$F_{15}(x)=\sum_{i=1}^{11}[a_i-\frac{x_i(b_i^2+b_ix_2)}{b_i^2+b_ix_3+x_4}]^2$	4	[-5,5]	0.00030
$F_{16}(x)=4x_1^2-2.1x_1^4+\frac{1}{3}x_1^6+x_1x_2-4x_2^2+4x_2^4$	2	[-5,5]	-1.0316
$F_{17}(x)=(x_2-\frac{5.1}{4\pi^2}x_1^2+\frac{5}{\pi}x_1-6)^2+10(1-\frac{1}{8\pi})\cos x_1+10$	2	[-5,5]	0.3979
$F_{18}(x)=[1+(x_1+x_2+1)^2(19-14x_1+3x_1^2-14x_2+6x_1x_2+3x_2^2)]\times$ $[30+2x_1-3x_2)^2\times(18-32x_1+12x_1^2+48x_2-36x_1x_2+27x_2^2)]$	2	[-2,2]	3
$F_{19}(x)=-\sum_{i=1}^{4}c_iexp(-\sum_{j=1}^{3}a_{ij}(x_j-p_{ij})^2)$	3	[1,3]	-3.86
$F_{20}(x)=-\sum_{i=1}^{4}c_iexp(-\sum_{j=1}^{6}a_{ij}(x_j-p_{ij})^2)$	6	[0,1]	-3.32
$F_{21}(x)=-\sum_{i=1}^{5}[(X-a_i)(X-a_i)^T+c_i]^{-1}$	4	[0,10]	-10.1532
$F_{22}(x)=-\sum_{i=1}^{7}[(X-a_i)(X-a_i)^T+c_i]^{-1}$	4	[0,10]	-10.4028
$F_{23}(x)=-\sum_{i=1}^{1}0[(X-a_i)(X-a_i)^T+c_i]^{-1}$	4	[0,10]	-10.5363

表 5.6 Composite functions

Function	Dim	Range	f_{min}
$F_{24}(CF1)$ $f_1, f_2, f_3, ...f_{10} = SphereFunction, [\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [1, 1, 1, ...1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [5/100, 5/100, 5/100, ...5/100]$	30	[-5,5]	0
$F_{25}(CF2)$ $f_1, f_2, f_3, ...f_{10} = Griewank'sFunction, [\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [1, 1, 1, ...1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [5/100, 5/100, 5/100, ...5/100]$	30	[-5,5]	0
$F_{26}(CF3)$ $f_1, f_2, f_3, ...f_{10} = Griewank'sFunction, [\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [1, 1, 1, ...1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [1, 1, 1, ...1]$	30	[-5,5]	0
$F_{27}(CF4)$ $f_1, f_2 = Ackley'sFunction, f_3, f_4 = Rastrigin'sFunction,$ $f_5, f_6 = WeierstrassFunction, f_7, f_8 = Griewank'sFunction,$ $f_9, f_{10} = SphereFunction, [\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [1, 1, 1, ...1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [5/32, 5/32, 5/32, ...5/32]$	30	[-5,5]	0
$F_{28}(CF5)$ $f_1, f_2 = Rastrigin'sFunction, f_3, f_4 = WeierstrassFunction,$ $f_5, f_6 = Griewank'sFunction, f_7, f_8 = Ackley'sFunction,$ $f_9, f_{10} = SphereFunction, [\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [1, 1, 1, ...1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [1/5, 1/5, 5/0.5, 5/0.5, 5/100, 5/100, 5/32, 5/32, 5/100, 5/100]$	30	[-5,5]	0
$F_{29}(CF6)$ $f_1, f_2 = Rastrigin'sFunction, f_3, f_4 = WeierstrassFunction,$ $f_5, f_6 = Griewank'sFunction, f_7, f_8 = Ackley'sFunction,$ $f_9, f_{10} = SphereFunction$ $[\sigma_1, \sigma_2, \sigma_3, ...\sigma_{10} = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$ $[\lambda_1, \lambda_2, \lambda_3, ...\lambda_{10}] = [0.1 * 1/5, 0.2 * 1/5, 0.3 * 5/0.5, 0.4 * 5/0.5, 0.5 * 5/100,$ $0.6 * 5/100, 0.7 * 5/32, 0.8 * 5/32, 0.9 * 5/100, 1 * 5/100]$	30	[-5,5]	0

表 5.7 Results of unimodal functions

function	type	IGOA	GOA	WOA	DA	ALO	PSO	OBLGOA
F1	avg	3.3538E-15	0.8386	**1.0082E-71**	0.0012	17.1234	6.6310E-6	2.77E-05
	std	1.9129E-15	0.8473	**5.3671E-71**	0.0008	17.8648	2.1612E-05	1.57E-05
	best	8.0120E-16	0.0683	**5.0175E-87**	0.00010	2.4120	1.5608E-07	7.75E-06
	worst	9.1885E-15	4.4591	**2.9415E-70**	0.0030	78.3488	0.0001	6.68E-05
F2	avg	2.4358E-08	10.2444	**1.5782E-51**	47.0419	4.9289	0.0953	0.0136
	std	1.0173E-08	22.2516	**4.8668E-51**	43.4815	3.7734	3.0196E-01	0.0377
	best	9.0029E-09	0.0290	**2.5368E-57**	3.8197	1.4484	0.0003	0.0011
	worst	5.7194E-08	79.1046	**2.1838E-50**	120.4026	21.3472	1.6419	0.1505
F3	avg	0.0121	1789.3452	43942.9825	4632.0793	1154.2060	227.7776	**0.0061**
	std	0.0211	1030.4488	1.6119E+04	2008.3302	1332.5907	81.0377	**0.0021**
	best	**8.6626E-12**	450.4535	17269.6957	1883.2010	249.1874	127.3043	0.0020
	worst	0.0983	4603.9086	85296.2126	10156.9819	5729.0798	425.9704	**0.0103**
F4	avg	0.0257	9.7756	56.3543	16.9378	31.4847	3.2311	**0.0165**
	std	0.0182	3.5013	25.3903	4.3129	8.2314	1.1774	**0.0081**
	best	**9.7116E-7**	3.0335	3.2199	6.5808	17.7108	1.4918	0.0010
	worst	0.0694	19.5647	89.1869	57.2014	48.8229	5.5785	**0.0297**
F5	avg	**26.4488**	965.6578	28.1591	348.5174	1615.4578	61.9257	28.3790
	std	**0.3046**	1572.3600	0.4797	553.5976	2814.6516	64.9991	0.3087
	best	25.8503	25.6988	27.2726	28.4537	143.9488	**1.7275**	27.6749
	worst	**27.0365**	7522.9967	28.7708	2223.6927	14636.9499	268.0065	28.7678
F6	avg	**1.4451E-6**	0.8997	0.3866	0.0023	20.5677	2.2973E-05	1.2542
	std	**4.8437E-7**	2.0343	0.2498	0.0055	29.2793	5.0099E-05	0.4237
	best	**5.2803E-7**	0.0203	0.0856	0.0001	3.0268	1.5390E-07	0.6616
	worst	**2.5210E-6**	11.0963	1.0626	0.0306	148.1366	0.0002	2.1687
F7	avg	**0.0010**	0.0234	0.0032	0.2504	0.1588	0.0274	0.0014
	std	**0.0014**	0.0106	0.0032	0.0768	0.0934	0.0113	0.0007
	best	**1.0746E-5**	0.0090	5.1138E-5	0.1003	0.0467	0.0115	0.0006
	worst	0.0072	0.0602	0.0118	0.4105	0.3750	0.0538	**0.0033**

解的可能性并提高算法的稳定性。与 GOA 和 OBLGOA 相比，提出的 IGOA 可以显着提高原算法的开发能力。

F8-F23 是具有多个局部最优的多模函数，可以测试算法的探索能力。如果算法在探索中不能很好地进行，那么在处理多模态函数时，搜索很可能会陷入局部最优，即使最佳的利用能力也无济于事。错误的方向可能会导致错误的结果。emph F8-F23 的结果显示在 emph 表 6 和 emph 表 7 中。

表 5.8 Results of multimodal functions-1

function	type	IGOA	GOA	WOA	DA	ALO	PSO	OBLGOA
F8	avg	-7594.1662	-7728.4324	**-9969.6875**	-6189.11	-7320.0609	-6688.3058	-7901.9216
	std	767.0277	**593.4825**	1919.0327	1833.8338	886.1388	684.8647	591.8701
	best	-9009.3177	-8903.0523	-12564.8051	**-12568.5831**	-9628.8472	-7869.7845	-9598.7278
	worst	-5993.9317	-6468.5651	-5709.2023	-5417.6748	-6101.1009	-5224.2865	**-6823.4703**
F9	avg	**0.0000**	9.4853	0.4119	8.8883	7.4670	4.5109	1.7919
	std	**0.0000**	5.4604	1.6505	4.5100	4.1855	2.8231	2.1937
	best	**0.0000**	1.9899	0.0000	0.9950	0.9959	0.9950	4.42E-09
	worst	**0.0000**	27.8586	8.1471	16.9143	16.9159	10.9445	6.9648
F10	avg	1.30e-08	3.0892	**4.56e-15**	5.0040	9.9207	1.3594	0.0010
	std	3.13e-09	0.8305	**2.18e-15**	3.1845	3.9783	0.8583	0.0002
	best	8.58e-09	1.5021	**8.88e-16**	1.1582	3.3950	0.0002	0.0005
	worst	1.97e-08	4.5855	**7.99e-15**	12.3302	16.3216	2.8857	0.0015
F11	avg	**5.50e-15**	0.6966	0.0069	0.0610	1.1803	0.0258	0.0002
	std	**5.71e-15**	0.1924	0.0379	0.0268	0.1822	0.0354	8.83E-05
	best	**6.66e-16**	0.3226	0.0000	0.0068	1.041	9.58e-07	7.36E-05
	worst	**2.52e-14**	1.0411	0.2076	0.1151	1.9360	0.1590	0.0004
F12	avg	**8.93e-08**	5.6658	0.0242	14.9630	25.6688	0.5808	0.0347
	std	**3.28e-08**	2.3767	0.0162	7.2414	14.1457	0.8898	0.0211
	best	**4.62e-08**	1.8994	0.004	6.9778	6.3337	1.89e-07	0.0051
	worst	**1.85e-07**	9.7664	0.0832	35.5319	55.4036	3.4350	0.1028
F13	avg	**0.0138**	8.7624	0.6039	24.9054	261.6986	0.2117	0.4087
	std	**0.0298**	9.0372	0.2536	15.4205	1186.8265	0.5112	0.2177
	best	**4.85e-07**	0.3005	0.1459	0.3854	1.3788	9.86e-07	0.1211
	worst	**0.0989**	35.2838	1.2995	56.9980	6534.7729	2.0239	1.1019

从表5.8和表5.9中可以发现，提议的 IGOA 可以在 16 个基准测试中的 11 个中获得最佳平均适应度，并且在 emph F10， emph F20 和 emph 23 IGOA 可以获得第二好的结果。关于标准偏差的结果表明，IGOA 的稳定性可能不如平均适应度所表现的那么好，但在大多数测试中它仍然是所有 7 种算法中最好的。关于最

表 5.9 Results of fixedmodal functions

function	type	IGOA	GOA	WOA	DA	ALO	PSO	OBLGOA
F14	avg	3.5566	**0.9980**	2.7622	2.2142	1.7242	4.5076	3.0103
	std	2.9933	**6.4341e-16**	3.3587	2.1646	1.2701	3.0100	1.5672
	best	**0.9980**	0.9980	0.9980	0.9980	0.9980	0.9980	0.9980
	worst	10.7632	**0.9980**	10.7632	10.7632	5.9288	12.6705	5.9288
F15	avg	**0.0003**	0.0071	0.0007	0.0032	0.0029	0.0038	0.0063
	std	**3.24E-05**	0.0089	0.0005	0.0062	0.0059	0.0076	0.0087
	best	**0.0003**	0.0007	0.0003	0.0006	0.0003	0.0003	0.0003
	worst	**0.0004**	0.0204	0.0022	0.0206	0.0204	0.0204	0.0210
F16	avg	**-1.0316**	-1.0316	-1.03162	-1.0316	-1.0316	-1.0316	-1.0316
	std	**4.44E-16**	8.1630e-13	1.6137e-09	1.0917e-13	5.1620e-07	6.5195e-16	4.63E-06
	best	**-1.0316**	-1.0316	-1.0316	-1.0316	-1.0316	-1.0316	-1.0316
	worst	**-1.0316**	-1.0316	-1.0316	-1.0316	-1.0316	-1.0316	-1.0316
F17	avg	**0.3979**	0.3979	0.3979	0.3979	0.3979	0.3979	0.3979
	std	**0**	7.3750e-13	8.4140e-06	2.1005e-14	1.0879e-06	0.0000	1.82E-06
	best	**0.3979**	0.3979	0.3979	0.3979	0.3979	0.3979	0.3979
	worst	**0.3979**	0.3979	0.3979	0.3979	0.3979	0.3979	0.3979
F18	avg	**3.0000**	8.4000	3.0000	3.0000	3.0000	3.0000	3.0000
	std	4.11E-08	20.5503	7.0794e-05	6.2232e-13	7.7233e-06	**6.0036e-16**	3.07E-10
	best	**3.0000**	3.0000	3.0000	3.0000	3.0000	3.0000	3.0000
	worst	**3.0000**	84.0000	3.0003	3.0000	3.0000	3.0000	3.0000
F19	avg	**-3.8628**	-3.7288	-3.8582	-3.8628	-3.8628	-3.8628	-3.8628
	std	1.67E-10	0.3062	0.0054	2.4040e-13	2.2464e-05	**2.6402e-15**	2.95E-05
	best	**-3.8628**	-3.8628	-3.8628	-3.8628	-3.8628	-3.8628	-3.8628
	worst	**-3.8628**	-2.7847	-3.8408	-3.8628	-3.8627	-3.8628	-3.8627
F20	avg	-3.2863	**-3.2943**	-3.2364	-3.2621	-3.2819	-3.2783	-3.2295
	std	0.0554	**0.0511**	0.1007	0.0610	0.0577	0.0584	0.1062
	best	**-3.3220**	-3.3220	-3.3213	-3.3220	-3.3220	-3.3220	-3.3220
	worst	**-3.2031**	-3.2031	-3.0184	-3.1981	-3.1974	-3.1996	-3.0334

表 5.9 续表：Results of fixedmodal functions

F21	avg	**-8.2870**	-6.0675	-8.5287	-5.8753	-6.6400	-5.9796	-7.2158
	std	**2.4947**	3.6827	2.7585	3.0237	3.2604	3.3646	3.3058
	best	**-10.1532**	-10.1532	-10.1498	-10.1532	-10.1532	-10.1532	-10.1532
	worst	**-5.0552**	-2.6305	-2.6292	-2.6305	-2.6305	-2.6305	-2.6303
F22	avg	**-9.3413**	-7.1190	-7.2448	-7.1785	-5.1249	-5.5091	-8.0724
	std	**2.1597**	3.6556	3.0850	3.3668	2.5737	3.1769	3.1972
	best	**-10.4029**	-10.4029	-10.4020	-10.4029	-10.4029	-10.4029	-10.4029
	worst	**-5.0877**	-1.8376	-1.8352	-1.8376	-2.7517	-1.8376	-2.7658
F23	avg	-7.9281	-4.9279	-6.5554	-6.6569	-5.4597	-4.7287	**-8.0791**
	std	**2.8536**	3.2796	3.3755	3.7558	3.0183	3.2997	3.5845
	best	**-10.5364**	-10.5364	-10.5348	-10.5364	-10.5364	-10.5364	-10.5364
	worst	**-3.8354**	-2.4217	-1.6721	-2.4217	-2.4217	-2.4273	-2.4217

佳适应度和最差适应度的结果可以表明，所提出的 IGOA 具有在几乎所有基准测试中找到最佳解决方案的最可靠能力，并且获得找到最差解决方案的最小概率。与原始的 GOA 和 OBLGOA 相比，提出的 IGOA 可以自然地提高算法的探索性能。

F24-F29 是复合测试函数，它结合了特定框架下的一些基本基准函数来构建新的可控测试函数。复合基准函数比多模式测试函数更复杂，更具挑战性，并且在评估算法的搜索能力时更有说服力。复合测试函数的结果如表5.10所示：

根据表5.10中列出的 *F24-F29* 的结果，可以看出，与其他算法相比，所提出的 IGOA 可以非常有竞争力。在 *F24* 和 *F26* 中，IGOA 可以获得最佳的适应度，在 *F25*，*F27* 和 *F29* IGOA 可以获得第二好的结果。在所有基准测试的标准差方面，IGOA 无法表现最佳或最差。IGOA 在最佳值和最差值上分别在 4 次测试和 3 次测试中表现最佳。尽管 IGOA 通常可以在标准偏差方面发挥作用，但它在寻找更好的解决方案和控制风险方面可以胜过其他算法。与原来的 GOA 和 OBLGOA 相比，IGOA 的表现要好得多。根据复合功能测试的结果，可以证明 IGOA 具有处理这种复杂和具有挑战性的问题的能力。

表 5.10 Results of composite functions

function	type	IGOA	GOA	WOA	DA	ALO	PSO	OBLGOA
F24	avg	**94.0428**	135.9242	341.2250	1098.8600	189.5840	326.5899	189.6304
	std	130.6690	127.0964	164.2449	**99.4449**	117.6155	149.6609	83.2598
	best	**3.3965**	29.6057	163.2687	796.3368	72.0563	118.9426	92.1210
	worst	504.8393	514.7878	844.9761	1261.1685	554.4573	717.0723	**411.3461**
F25	avg	324.3197	**284.6066**	521.9304	1211.2487	413.5431	393.6293	472.0768
	std	151.6351	163.1829	130.9867	**94.2232**	156.9269	126.4408	143.6411
	best	**21.2736**	28.9418	244.2571	1006.3367	72.6341	161.9665	66.0628
	worst	**497.3555**	510.3606	673.6729	1354.7747	662.1609	610.8645	606.4133
F26	avg	**525.7384**	624.5092	1052.0733	1547.6195	884.9306	608.4939	875.4033
	std	128.5247	176.1401	155.6153	155.8894	157.8576	**103.4249**	151.2467
	best	317.4166	**200.8044**	849.0492	1064.4776	644.6867	326.5611	627.7263
	worst	900.011	1196.9114	1351.5909	1710.9641	1222.5593	**828.4274**	1217.9587
F27	avg	894.6861	945.6744	902.6182	1421.5043	929.5948	**757.5836**	895.7587
	std	29.1104	101.1013	**15.0373**	46.9584	128.2259	114.6504	34.7088
	best	740.5569	638.304	896.3862	1319.5644	650.1341	**538.8135**	719.3590
	worst	**900.0053**	1030.5605	982.1588	1519.1409	1066.455	1012.7758	953.3374
F28	avg	304.9727	**142.0615**	456.9052	1354.378	194.2336	303.9595	497.1264
	std	367.9136	**102.2128**	255.4945	127.0279	182.4486	143.7137	351.6947
	best	**46.3796**	54.2782	179.8317	992.2916	87.2393	113.7433	110.5681
	worst	900.0040	**435.4893**	900.0000	1505.6248	1025.2583	675.3012	900.0049
F29	avg	900.0003	907.4664	**900.0000**	1371.4241	925.7089	931.1013	900.0004
	std	0.0001	5.1380	**0.0000**	56.4060	7.5901	19.0467	0.0002
	best	**900.0000**	901.0860	900.0000	1254.6648	913.0986	910.3009	900.0001
	worst	900.0006	920.7888	**900.0000**	1000.7252	1000.7252	1000.7252	900.0009

5.5.6.3 wilcoxon 秩和检验

基于 30 个独立操作的平均值和标准偏差的比较没有比较每个操作之间的差异。实验结果仍有可能包含某些意外情况。为了消除这种偶然性并证明实验结果的重要性，在这项工作中引入了 Wilcoxon 秩和检验。Wilcoxon 秩和检验是零假设的非参数检验，它用于确定两个独立的数据集是否来自相同的分布式群体。在这项工作中，计算了关于 IGOA 与 *F1-F29* 上的每个其他算法之间的统计数据的 *p-values*。当 *p-value* 小于 0.05 时，可以认为两个样本之间的差异是显着的。IGOA 的威尔科克森秩和检验结果如表5.11所示。

从5.11中可以看出，在大多数基准测试中，IGOA 与另一个比较算法之间的 *p-values* 小于 0.05。在某些测试函数中，某些 *p-values* 超过 0.05，这些函数通过不同的算法获得相同的最优解，例如 *F14*，*F19*，*F20*，*F21*，*F22* 和 *F23*。通过分析复合函数的结果，在 *F25* 和 *F28* 中，*p-values* 在 IGOA 和 GOA 之间超过 0.05，其中 GOA 获得最佳适应度而 IGOA 没有。综合考虑，仍然可以证明提议的 IGOA 可以显着提高 GOA 的性能。

5.5.6.4 收敛图

本部分讨论了算法的收敛速度。用于比较部分基准函数的 IGOA 和所有其他 6 种算法的收敛曲线显示在 emph 图 2 中。

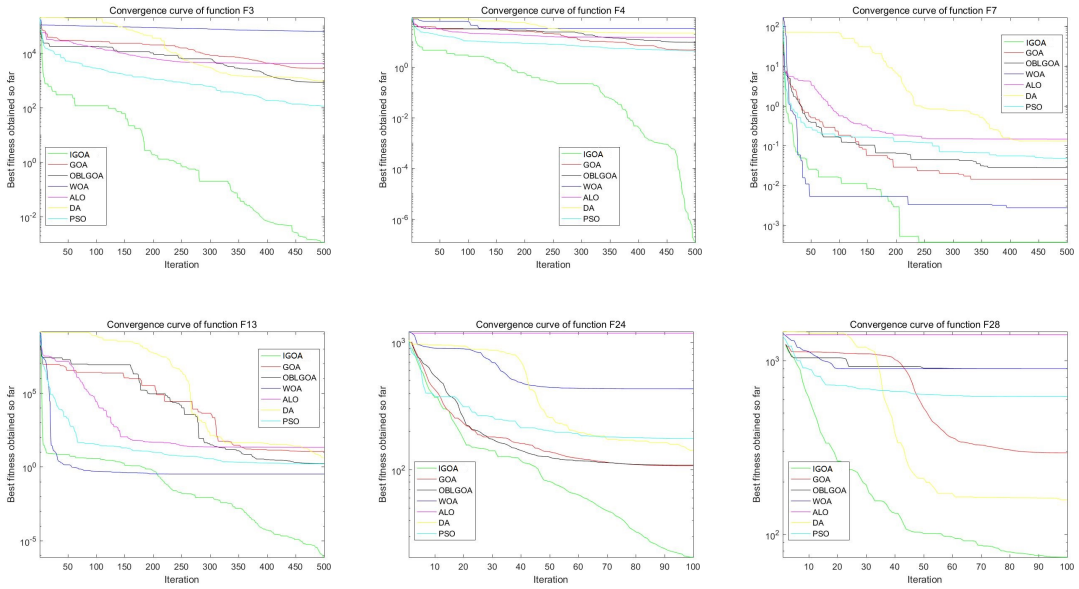

图 5.3 Convergence curves for all the 7 algorithms over some benchmark functions

水平轴是当前迭代的数量，垂直轴是迄今为止获得的最佳适应值。为了使曲

表 5.11 Results of Wilcoxon rank-sum test

function	GOA	WOA	DA	ALO	PSO	OBLGOA
F1	3.02E-11	3.02E-11	3.02E-11	3.02E-11	9.53E-07	2.03E-07
F2	3.02E-11	3.02E-11	3.02E-11	3.02E-11	7.12E-09	2.37E-10
F3	3.02E-11	3.02E-11	3.02E-11	3.02E-11	3.02E-11	5.57E-10
F4	3.02E-11	3.02E-11	3.02E-11	3.02E-11	3.02E-11	1.21E-10
F5	1.86E-09	1.60E-07	3.02E-11	3.02E-11	0.5201	5.07E-10
F6	3.02E-11	3.02E-11	3.02E-11	0.0099	3.02E-11	3.02E-11
F7	3.16E-10	0.7618	3.02E-11	3.02E-11	8.15E-11	0.0594
F8	0.0030	3.82E-09	0.7958	2.59E-06	0.0133	0.0002
F9	3.02E-11	1.24E-09	3.02E-11	3.02E-11	2.86E-11	4.98E-11
F10	3.02E-11	3.02E-11	3.02E-11	3.02E-11	1.43E-08	3.83E-05
F11	3.02E-11	4.56E-11	3.02E-11	3.02E-11	1.34E-05	3.02E-11
F12	3.02E-11	3.02E-11	3.02E-11	3.02E-11	0.0002	3.02E-11
F13	3.02E-11	3.02E-11	3.02E-11	3.02E-11	0.9117	3.02E-11
F14	1.67E-05	0.1578	0.0265	0.0082	0.7842	0.6951
F15	1.33E-10	1.07E-09	2.87E-10	2.87E-10	0.3328	2.15E-10
F16	0.0064	6.72E-10	3.02E-11	1.81E-05	4.08E-12	3.02E-11
F17	3.20E-06	3.01E-11	3.01E-11	6.78E-06	4.56E-12	3.01E-11
F18	1.73E-07	3.02E-11	3.02E-11	0.01911	1.55E-11	5.09E-08
F19	0.5997	0.002266	2.13E-05	3.33E-11	5.14E-12	0.0003
F20	0.5395	0.0003	0.00250	0.7958	0.0003	0.0001

表 5.11 续表：Results of Wilcoxon rank-sum test

F21	0.0002	1.25E-05	6.74E-06	0.0850	0.2822	5.46E-06
F22	0.0005	5.27E-05	3.83E-06	0.5493	0.0627	0.0013
F23	6.05E-07	3.83E-06	1.73E-06	0.1858	0.0009	5.27E-05
F24	0.0016	6.01E-08	0.0001	3.02E-11	3.81E-07	0.0001
F25	0.7618	1.09E-05	0.0138	3.02E-11	0.1907	1.02E-05
F26	0.0046	5.49E-11	5.57E-10	3.02E-11	0.0038	4.20E-10
F27	7.74E-06	7.34E-09	0.0064	3.02E-11	3.08E-08	2.68E-06
F28	0.3711	0.0042	0.0083	3.02E-11	0.0073	0.0001
F29	3.02E-11	2.14E-11	3.02E-11	3.02E-11	3.02E-11	0.0024

线更加可区分，在垂直轴上使用对数。 emph 图 2 中的曲线图像表明，在大多数搜索过程中，IGOA 可以相对于整个搜索过程保持较大的斜率。这种现象可能意味着所提出的非线性舒适区参数可以有助于充分利用每次迭代。曲线跨度在大多数函数中都是充足的，这可以表明基于 *L acuteevy* flight 的局部搜索机制使算法更具创造性。曲线突然出现在 emph F3， emph F4， emph F7 中， emph F28 表明随机跳跃策略可以帮助搜索跳出局部最优。收敛曲线表明，所提出的 IGOA 可以使搜索过程比其他人更快地收敛。

5.6 任务调度问题

5.6.1 问题描述

任务调度问题是云计算领域中常见的问题 cite qiao2016an，edge computing area cite guo2017energy 和 SEA Service System cite wang2015research。大量用户会产生大量任务。因此，当大量用户同时询问服务时，系统应该有效地处理任务，这可能是计算系统的基本问题，尤其是在资源受限的条件下。在任务调度问题中，到达调度中心的一系列任务等待分配给特定的执行节点。

由于资源条件，工作负载，处理速度和任务类型的不同，执行节点在此问题中可能是异构的。任务调度的不同解决方案可能导致不同的后果，并且优秀的解决方案可能会影响很多。对于服务提供商，将任务分配给适当的节点可以节省能

源并减少预算。对于服务使用者，将任务调度到高效节点可以减少等待时间并改善用户体验。

任务调度问题可能是一个优化问题，它是一个NP难问题citetawfeek2013cloud。应用了许多算法来解决任务调度问题。一些基于最佳资源选择（BRS）的算法，如Max-Min，Min-Min和Sufferage，是解决任务调度问题的传统方法cite qiao2016an。当任务数量增加时，任务的维度也会增加，这会扩大搜索空间并使优化问题更加复杂。传统算法无法处理这种情况。近年来，许多元启发式算法被应用于解决任务调度问题。Goldberg在1988年提出的遗传算法（GA）是一种着名的进化算法，它代表了一种解决任务调度问题的染色体调度方案。由于交叉和变异的操作，GA计算起来很复杂。粒子群算法（PSO）是Kennedy和Eberhard在1995年提出的经典元启发式算法.PSO将调度解决方案表示为离散粒子，并通过群体的个人最佳和全局最佳信息进行演化。PSO很容易陷入局部最优，并且在处理多模态问题时表现不佳。

最近提出了一些元启发式算法，如DA，WOA，GOA等。在这项工作中，建议的IGOA用于任务调度问题。将调度到M个节点的N个任务的解决方案抽象为表示搜索空间中的位置的N维离散矢量。每个解决方案可以对应于系统的完工时间和预算，并且可以使用完工时间和预算来计算适合度值。一些搜索代理通过IGOA的搜索策略在整个搜索空间中搜索最佳适应度。

5.6.2 任务调度模型

在本文中，我们假设用户启动emph N任务，这些任务是$T_1 \square T_2 \square ... T_N$和emph M个节点，$S_1 \square S_2 \square ... S_M$正在等待处理它们。任务调度模型描述如下。

1. The set of N tasks is $\{T_1, T_2, ... T_N\}$, and T_i represents the resource that the *i-th* task consumes.

2. The set of M nodes is $\{S_1, S_2, ... S_M\}$ and S_j represents the maximum amount of resources that the *j-th* node can provide.

3. A task can be operated at any node as long as the node can provide enough resource, which means $T_i \leq S_j$.

4. Every node can handle only one task at the same time. Multiple tasks can be run sequentially on the same node.

5. There are K types of all the tasks. The vector of types of tasks is $tot[N]$, and tot_i which is an integral value in [1, K] represents the type of *i-th* task.

6. The ability of every node to handle different type of task is different. The matrix of operating speed is $mips[M,K]$. $mips_j^k$ represents the resource that the type k task can consume on the *j-th* node in a unit time. The working time for the *i-th* task running on the *j-th* node which is et_{ij} is calculated as follows.

$$et_{ij} = \begin{cases} \frac{T_i}{mips_j^{tot_i}} & task\ i\ runs\ on\ node\ j \\ 0 & task\ i\ does\ not\ runs\ on\ node\ j \end{cases} \tag{5.21}$$

7. The makespan of the *j-th* node which is st_j is calculated by adding all the operating time of the tasks handled on the *j-th* node. The equation is shown as follows.

$$st_j = \sum_{i=1}^{N} et_{ij} \tag{5.22}$$

The total *Makespan* is the longest time for all the nodes. *Makespan* is calculated as follow.

$$Makespan = \max\{st_j | j = 1,2,3...M\} \tag{5.23}$$

8. The work node can produce the extra budget when it is working. The budget is only related to the time of working, regardless of the type of task running on it. The vector of the price of the nodes is $bps[M]$ which represents the budget that the *j-th* node generate in a unit time.

9. The total *Budget* is calculated by adding all the budget of all the nodes. *Budget* is calculated as follows.

$$Budget = \sum_{j=1}^{M} (st_j \times bps_j) \tag{5.24}$$

10. Makespan and budget can describe the cost of the problems in two aspects. A unified evaluation fitness is required. In this module, the fitness of the problem is defined as the mixture of makespan and budget with weight parameters of α and β. The *fitness* is calculated as follows.

$$fitness = \alpha Makespan + \beta Budget \tag{5.25}$$

11. The search process is continuous, but the solution to the problem is discrete. When a step of search is finished, the search agent should adjust itself to the nearest integral location.

12. The switching time between two tasks on the same node is ignored in this module. The transmission time among the edge nodes is also ignored.

5.6.3 实验结果

在此任务调度模块中，N 任务分为 K 类型，M 工作节点正在准备处理它们。任务调度模块的参数选择的细节取决于具体问题，这里不再讨论。在这项工作中，不失一般性，K，N 和 M 分别设置为 4,10 和 5。为了平衡 *Makespan* 和 *Budget* 对 *fitness* 的影响，系数 α 和 β 设置为 0.7 和 0.3。上述参数设定如表**??**所示。

$$K = 4;\ N = 10;\ M = 5;\ \alpha = 0.7;\ \beta = 0.3; \tag{5.26}$$

The vector T, tot, S and bps, and the matrix mips are set as *Table 10-12*.

表 5.12 Resources and types about tasks

task	1	2	3	4	5	6	7	8	9	10
T	40	20	30	40	25	35	45	10	5	10
tot	1	1	1	2	2	2	3	3	4	4

表 5.13 Resource and bps about nodes

node	1	2	3	4	5
S	100	50	150	100	80
bps	0.5	0.2	0.8	0.1	0.5

表 5.14 Mips about tasks and nodes

$mips_j^k$	k:1-4			
j:1-5	5	2	10	8
	2	4	2	1
	8	10	10	5
	2	1	3	4
	5	5	8	8

在这项工作中，6 个算法与提议的 IGOA 进行了比较。每种算法使用 30 个

搜索代理。为了减少意外因素的影响，对每种算法进行了 30 次测试，并计算并比较了平均值，标准差，最佳值和最差值等统计数据。实验结果如表5.15所示。

表 5.15 Results about task scheduling

algorithm	average	std	best	worst	promotion
IGOA	13.50	0.62	12.63	14.62	0
GOA	14.95	0.82	13.50	16.67	9.7%
WOA	14.08	0.75	13.02	15.93	4.1%
ALO	19.83	2.89	16.02	26.76	31.9%
DA	19.00	2.49	15.62	26.81	29.0%
PSO	17.93	1.68	13.96	22.29	24.7%
OBLGOA	14.85	0.74	13.30	16.27	9.1%

实验结果表明，所提出的 IGOA 在所有领域都优于其他算法。IGOA 在平均值方面可以做得最好，对于最小的标准偏差它可以更稳定。它可以找到所有算法的最佳解决方案，并且可以控制它可以找到的最差值。与其他算法相比，IGOA 可以将平均值分别提升 9.7%，4.1%，31.9%，29.0%，24.7%和 9.1%。

5.7 本章小结

第 6 章　基于预测的容器弹性服务策略

6.1　1 引言

这一章对应研究点 4，弹性服务。如果终端收到服务请求，再启动对应容器进行服务，则启动时间会大大增加用户等待时间。但由于终端资源有限，终端服务不能像云端服务那样一直在后台运行，等待服务请求。因此我们提出基于预测的容器弹性服务策略，根据预测结果提前弹性部署容器服务，平衡二者的关系。

6.2　相关工作

6.3　基于预测的容器弹性服务系统设计

6.4　基于卡尔曼滤波的预测算法

我们利用卡尔曼滤波的方法，采集历史用户请求数据，并对下一时间点的用户请求数量进行预测，弹性调整容器服务规模。

6.5　实验结果

6.6　本章小结

第 7 章　总结与展望

7.1　工作总结

7.2　工作展望

参考文献

[1] 史红周. 支持普适计算的智能终端服务及设备管理技术研究[D]. 中国科学院研究生院（计算技术研究所）, 2004.

[2] 施巍松, 孙辉, 曹杰, 等. 边缘计算: 万物互联时代新型计算模型[J]. 计算机研究与发展, 2017:1.

[3] 中国信息通信研究院（工业和信息化部电信研究院）. 物联网白皮书[R]. 中国信息通信研究院（工业和信息化部电信研究院）, 2016.

[4] V N C. Cisco global cloud index: Forecast and methodology, 2016–2021[R]. Cisco, 2016.

[5] NOOR T H, ZEADALLY S, ALFAZI A, et al. Mobile cloud computing: Challenges and future research directions[J/OL]. Journal of Network and Computer Applications, 2018, 115:70-85. DOI: 10.1016/j.jnca.2018.04.018.

[6] 江绵恒. 城市化与信息化——中国发展的时代机遇[J]. 信息化建设, 2010(6):8-9.

[7] BUYYA R, YEO C S, VENUGOPAL S, et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility[J]. Future Generation computer systems, 2009, 25(6):599-616.

[8] ISMAIL B I, GOORTANI E M, KARIM M B A, et al. Evaluation of docker as edge computing platform[C]//Open Systems. 2016.

[9] 王劲林, 田静, 尤佳莉, 等. 一种现场, 弹性, 自治的网络服务系统——海服务系统研究与设计[J]. Chinese Science Bulletin, 2015, 45(10):1237-1248.

[10] RENNER T, MELDAU M, KLIEM A. Towards container-based resource management for the internet of things[C]//2016 International Conference on Software Networking (ICSN). IEEE, 2016: 1-5.

[11] MORABITO R, FARRIS I, IERA A, et al. Evaluating performance of containerized iot services for clustered devices at the network edge[J]. IEEE Internet of Things Journal, 2017, 4(4):1019-1030.

[12] 文雨, 孟丹, 詹剑锋. 面向应用服务级目标的虚拟化资源管理[J]. 软件学报, 2013(2): 358-377.

[13] 本刊编辑部. 虚拟化概述[J]. 保密科学技术, 2017(10):8-10.

[14] PEARCE M, ZEADALLY S, HUNT R. Virtualization: Issues, security threats, and solutions [J]. ACM Computing Surveys (CSUR), 2013, 45(2):17.

[15] 陈思锦, 吴韶波, 高雪莹. 云计算中的虚拟化技术与虚拟化安全[J]. 物联网技术, 2015(3): 52-53.

[16] 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.

[17] KOLHE S, DHAGE S. Comparative study on virtual machine monitors for cloud[C]//2012 World congress on information and communication technologies. IEEE, 2012: 425-430.

[18] LEITE D, PEIXOTO M, SANTANA M, et al. Performance evaluation of virtual machine monitors for cloud computing[C]//2012 13th symposium on computer systems. IEEE, 2012: 65-71.

[19] OLUDELE A, OGU E C, SHADE K, et al. On the evolution of virtualization and cloud computing: A review[J]. Journal of Computer Sciences and Applications, 2014, 2(3):40-43.

[20] SAHASRABUDHE S S, SONAWANI S S. Comparing openstack and vmware[C]//2014 International Conference on Advances in Electronics Computers and Communications. IEEE, 2014: 1-4.

[21] LI P. Selecting and using virtualization solutions: our experiences with vmware and virtualbox [J]. Journal of Computing Sciences in Colleges, 2010, 25(3):11-17.

[22] LIU D, ZHANG Y Y, ZHANG N, et al. A research on kvm-based virtualization security[C]// Applied Mechanics and Materials: volume 543. Trans Tech Publ, 2014: 3126-3129.

[23] MORABITO R. Virtualization on internet of things edge devices with container technologies: a performance evaluation[J]. IEEE Access, 2017, 5:8835-8850.

[24] XAVIER M G, NEVES M V, ROSSI F D, et al. Performance evaluation of container-based virtualization for high performance computing environments[C]//2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE, 2013: 233-240.

[25] BABU A, HAREESH M, MARTIN J P, et al. System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver [C]//2014 Fourth International Conference on Advances in Computing and Communications. IEEE, 2014: 247-250.

[26] VAUGHAN-NICHOLS S J. New approach to virtualization is a lightweight[J]. Computer, 2006, 39(11):12-14.

[27] LIU D, ZHAO L. The research and implementation of cloud computing platform based on docker[C]//2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing (ICCWAMTIP). IEEE, 2014: 475-478.

[28] BERNSTEIN D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE Cloud Computing, 2014, 1(3):81-84.

[29] 马晓光, 刘钊远. 一种适用于 Docker Swarm 集群的调度策略和算法[J]. 计算机应用与软件, 2017, 34(5):283-287.

[30] 肖伟民孙鹏. 嵌入式虚拟化技术研究综述[J]. 网络新媒体技术, 2019(02):9-18.

作者简历及攻读学位期间发表的学术论文与研究成果

本科生无需此部分。

作者简历

casthesis 作者

吴凌云，福建省屏南县人，中国科学院数学与系统科学研究院博士研究生。

ucasthesis 作者

莫晃锐，湖南省湘潭县人，中国科学院力学研究所硕士研究生。

已发表 (或正式接受) 的学术论文:

[1] ucasthesis: A LaTeX Thesis Template for the University of Chinese Academy of Sciences, 2014.

申请或已获得的专利:

(无专利时此项不必列出)

参加的研究项目及获奖情况:

可以随意添加新的条目或是结构。

致　谢

感激 casthesis 作者吴凌云学长，gbt7714-bibtex-style 开发者 zepinglee，和 ctex 众多开发者们。若没有他们的辛勤付出和非凡工作，LaTeX 菜鸟的我是无法完成此国科大学位论文 LaTeX 模板 ucasthesis 的。在 LaTeX 中的一点一滴的成长源于开源社区的众多优秀资料和教程，在此对所有 LaTeX 社区的贡献者表示感谢！

ucasthesis 国科大学位论文 LaTeX 模板的最终成型离不开以霍明虹老师和丁云云老师为代表的国科大学位办公室老师们制定的官方指导文件和众多 ucasthesis 用户的热心测试和耐心反馈，在此对他们的认真付出表示感谢。特别对国科大的赵永明同学的众多有效反馈意见和建议表示感谢，对国科大本科部的陆晴老师和本科部学位办的丁云云老师的细致审核和建议表示感谢。谢谢大家的共同努力和支持，让 ucasthesis 为国科大学子使用 LaTeX 撰写学位论文提供便利和高效这一目标成为可能。