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

郭志川 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

Research on Containerized Multi-terminal Collaborative Service Technology

A dissertation submitted to the
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Doctor of Philosophy
in Signal and Information Processing

By

Zhao Ran

Supervisor: Professor Ni Hong
Professor Guo Zhichuan

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

随着边缘计算技术的快速发展，位于网络边缘的用户终端设备在人们的数字化生活中正扮演着越来越重要的角色。用户终端需要承担越来越多的计算任务，这也对终端设备的计算能力提出了越来越高的要求。常用的云计算解决方案在用户终端服务的场景中会带来实时性差、部署成本高、带宽拥挤等问题，利用靠近用户的终端设备上空闲的计算资源协同为用户提供就近计算服务成为一种可能。

由于多终端的资源分布分散、资源异构性强，存在资源利用难度大及利用率低、服务响应时间长、用户体验较差等问题。为了提高终端资源利用率，提升用户体验，本文开展了基于容器化多终端协同服务技术研究。本文首先引入容器技术来解决终端资源的异构和管理问题，设计了基于容器化的多终端协同服务系统。进而，本文提出基于容器的多终端透明计算迁移技术，减少服务响应时间，提高用户体验。然后，本文进行多终端协同服务任务调度算法优化工作，合理分配终端资源，降低终端服务费用，减少服务响应时间。最后，本文研究基于预测的终端弹性服务技术，提高了终端资源利用率。

本文主要研究内容与成果如下：

（1）基于容器化多终端服务系统架构设计

设计了基于容器化多终端协同服务系统，及资源管理模块，引入 Docker 容器虚拟化技术对多终端资源进行虚拟化，解决了终端资源异构性问题，形成按需使用的资源池，以容器的形式提供服务和进行管理。服务管理模块，包括服务注册、服务节点选择、服务生命周期管理等功能。任务调度模块，可根据不同任务请求对资源的消耗情况，调度合适的节点进行执行。此外设计了弹性服务模块，预测用户服务请求变化趋势，调节终端服务规模。并提出了去中心化的自组织网络结构，实现多终端节点管理。

（2）基于容器的多终端透明计算迁移技术

针对终端服务中的服务响应时间长的问题，提出一种基于容器的 Web Worker 透明计算迁移技术。通过对终端底层 Web 应用执行环境中的部分接口进行修改，通过 WebSocket 通信机制，将 Web 应用中的 Web Worker 迁移到部署在边缘容器集群中的服务端进行执行。实验结果表明，使用基于容器的多终端透明计算迁移技术能够在 Web Worker 数量较多的情况下，减少 Web 应用的总执行时间，最高

能够减少 80.6%，对于提高终端用户体验有着明显的效果；相比非透明计算迁移技术，基于容器的多终端透明计算迁移技术最高能够减少应用执行时间 33.1%。

（3）多终端协同服务任务调度算法

为了解决多终端协同服务任务调度问题，合理分配终端资源，降低终端服务费用，减少任务执行时间，提高用户体验，提出一种带有随机跳出机制的动态权重蝗虫优化算法（DJGOA）。DJGOA 算法在元启发式算法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和动态的权重参数。测试实验结果表明所提出的带有随机跳出机制的动态权重蝗虫优化算法在寻找最优结果方面具有较好的效果。为了更进一步优化算法的性能，基于 DJGOA 算法，提出了一种改进蝗虫优化算法（IGOA）。IGOA 算法引入了变型的 sigmoid 函数作为非线性舒适区调节参数，并结合了基于 *Lévy* 飞行的局部搜索机制和基于线性递减参数的随机跳出策略。仿真实验结果表明，改进蝗虫优化算法在解决多终端协同服务任务调度问题上也能够取得较好的效果，相比其他对比算法效果最高提升 31.9%。

（4）基于预测的容器弹性服务策略

为了解决多终端协同服务技术中的预部署问题，促进提高终端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，提出了基于预测的容器弹性服务策略。提出了一种改进卡尔曼滤波算法用于预测用户服务请求趋势。根据预测结果弹性部署容器服务，动态调整多终端协同服务的规模，平衡提高终端资源利用率与降低用户服务请求响应等待时间之间的关系。

关键词：容器虚拟化，智能终端，协同服务，计算迁移，任务调度，弹性服务

Abstract

With the rapid development of edge computing technology, user terminal devices at the edge of the network are playing an increasingly important role in people's digital life. User terminal devices need to undertake more and more computing tasks, which also puts higher and higher requirements on the computing power of terminal devices. Commonly used cloud computing solutions can bring problems such as poor real-time performance, high deployment cost, and bandwidth congestion in the scenario of user terminal device services. It is possible to use the idle computing resources on the terminal devices close to the users to provide users with nearby computing services.

Due to the scattered resource distribution and heterogeneous resource heterogeneity of multiple terminals, there are problems such as difficulty in resource utilization, low utilization rate, long service response time, and poor user experience. In order to improve the utilization of terminal resources and enhance the user experience, this paper has carried out research on containerized multi-terminal collaborative service technology. This paper first introduces container technology to solve the heterogeneous and management problems of terminal resources, and designs a multi-terminal collaborative service system based on containerization. Furthermore, this paper proposes a container-based multi-terminal transparent computing migration technology to reduce service response time and improve user experience. Then, this paper carries out the optimization of multi-terminal collaborative service task scheduling algorithm, which rationally allocates terminal resources, reduces terminal service costs, and reduces service response time. Finally, this paper studies the terminal elastic service technology based on prediction and improves the utilization of terminal resources.

The main research contents and results of this paper are as follows:

(1) Architecture design based on containerized multi-terminal service system

The containerized multi-terminal collaborative service system and resource management module are designed. The Docker container virtualization technology is introduced to virtualize multi-terminal resources, which solves the heterogeneity of terminal resources and forms an on-demand resource pool to provide service and management in the form of containers. Service management module is proposed, including service

registration, service node selection, service lifecycle management and other functions. The task scheduling module can schedule the tasks to the appropriate nodes according to the resource consumption of the different task requests. In addition, an elastic service module is designed to predict the trend of user service requests and adjust the scale of terminal services. A decentralized self-organizing network structure is proposed to realize multi-terminal node management in this chapter.

(2) Container-based multi-terminal transparent computing offloading technology

Aiming at the problem of long service response time in terminal service, a container-based Web Worker transparent computing offloading technology is proposed. By modifying some interfaces in the underlying web application execution environment of the terminal, the Web Workers in the web application are migrated to the server deployed in the edge container cluster for execution through the WebSocket communication mechanism. The experimental results show that the container-based multi-terminal transparent computing offloading technology can reduce the total execution time of the web application by 80.6% in the case of a large number of Web Workers, which has obvious effects on improving the end user experience. Compared to the non-transparent computing offloading technology, container-based multi-terminal transparent computing offloading technology can reduce application execution time by 33.1%.

(3) Multi-terminal collaborative service task scheduling algorithm

In order to solve the multi-terminal collaborative service task scheduling problem, allocate terminal resources reasonably, reduce terminal service cost, reduce task execution time and improve user experience, a dynamic weight grasshopper optimization algorithm with random jumping (DJGOA) is proposed. Based on the meta-heuristic algorithm grasshopper optimization algorithm, the DJGOA algorithm adds a jumping mechanism based on a completely random jumping factor and a dynamic weight parameter. The experimental results show that the proposed dynamic weight grasshopper optimization algorithm with random jumping has a good effect in finding the optimal result. In order to further optimize the performance of the algorithm, an improved grasshopper optimization algorithm (IGOA) is proposed based on the DJGOA algorithm. The IGOA algorithm introduces a variant sigmoid function as a nonlinear comfort zone adjustment parameter, and combines a local search mechanism based on *Lévy* flight and a random jumping strategy based on linear descending parameters. The simulation results show

that the improved grasshopper optimization algorithm can achieve better results in solving the multi-terminal collaborative service task scheduling problem, and the effect is up to 31.9% compared with other comparison algorithms.

(4) Prediction-based elastic container service strategy

In order to solve the pre-deployment problem in multi-terminal collaborative service technology, promote the balance between the terminal resource utilization and the reduction of user service request response waiting time, a prediction-based elastic container service strategy is proposed. An improved Kalman filter algorithm is proposed to predict the trend of user service requests. According to the prediction result, the container service is flexibly deployed, the scale of the multi-terminal collaborative service is dynamically adjusted, and the relationship between the utilization of the terminal resource and the reduction of waiting time of the user service request is balanced.

Keywords: Container Virtualization, Intelligent Terminal, Collaborative Service, Computing Offloading, Task Scheduling, Elastic Service

目 录

图形列表

表格列表

符号列表

缩写

VM	Virtual Machine
LXC	Linux Container
DC	Data Center
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
CC	Cloud Computing
CV	Connected Vehicle
AP	Access Points
EC	Edge Computing
VMM	Virtual Machine Monitor
OS	Operation System
C-S	Client-Server
UDS	Unix Domain Socket
MCC	Mobile Cloud Computing
MEC	Mobile Cloud Computing
FIFO	First In First Out
GA	Genetic Algorithm
PSO	Particle Swarm Optimization
ACO	Ant Colony Optimization
ALO	Ant Lion Optimizer
WOA	Whale Optimization Algorithm
DA	Dragonfly Algorithm
GOA	Grasshopper Optimization Algorithm
SOA	Service-Oriented Architecture
AS	Autonomous System

CDC	Connectivity-based Decentralized Node Clustering
SA	Simulated Annealing Algorithm
CS	Cuckoo Search Algorithm
OBLGOA	Improved Grasshopper Optimization Algorithm Using Opposition-Based Learning
CGOA	Chaotic Grasshopper Optimization Algorithm
DJGOA	Dynamic Weight Grasshopper Optimization Algorithm with Random Jumping
IGOA	Improved Grasshopper Optimization Algorithm
ARMA	Auto-Regressive and Moving Average
GM	Grey Models
KF	Kalman Filter

第 1 章　绪论

随着信息技术的快速发展，人工智能、边缘计算、物联网等技术迅速崛起，经过近几年的飞速发展，已开始形成规模化的产业链闭环。2019 年 3 月 5 日，李克强总理在政府工作报告中首次提出“智能 +”，强调加快人工智能等新兴技术产业的研发落地。作为聚合和承载各类应用的载体，智能终端是技术落地的重要环节，也是产业链参与者开展跨界竞争、多产业链环节运营的良好切入点，对新技术产业的发展和布局都有着重要影响。也正因如此，智能终端除了为用户提供基础服务以外，正逐步向功能化、场景化、个性化延伸。

当前，资源管理问题已成为智能终端发展的重要瓶颈之一，计算任务的增长与计算资源分配不均的矛盾日益凸显。一方面，智能终端要承担的计算任务越来越重，而终端上所拥有的各类计算、存储等资源往往有限，发展已难以跟上其日渐增长的需求；另一方面，家庭、办公楼等多个场景的智能终端环境中，还存在着很多相对空闲、资源仍未得到充分利用的计算设备。

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，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、能够进行网络连接和通信、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 [3]。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 [4]。而根据欧盟委员会预测，到 2020 年将有 500 到 1000 亿个智能设备连接到互联网 [5]。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[4]，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，并更进一步推动人民群众的生活方式向更加智能化、数字化的方向发展。

各种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%[6]。智能终端设备中央处理单元（CPU）的计算能力还不能跟上终端服务对其的要求 [7–9]。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 [10]。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 [11]。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 [12]。

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在着很多不足之处。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 [13]。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，存在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 [14]，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有其专有目的的，会存在间隔为数小时的使用周期，在几个小时的工作时间以外，智能终端设备会处于空闲状态 [15]。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，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。多终端协同服务技术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，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, 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 [16]。

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、资源和服务的管理问题，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

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，合理分配和利用终端空闲资源，优化智能终端的服务质量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。

本研究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，降低终端服务响应时间，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，本文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管理利用、如何解决终端资源异构性问题、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。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，首先介绍了终端协同服务技术、容器虚拟化技术、计算迁移技术、任务调度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, 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，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，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，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，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，引入变形的 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，提出基于 *Lévy* 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，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，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，那么就需要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，这种方案会产生额外的用户等待时间。尽管相比于传统的 VM 虚拟机，Docker 容器技术的启动时间已经很短，但这个启动时间相比于用户请求等待响应时间仍然较长，不能满足用户的需求。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趋势进行预测，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更合理利用的目的。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：

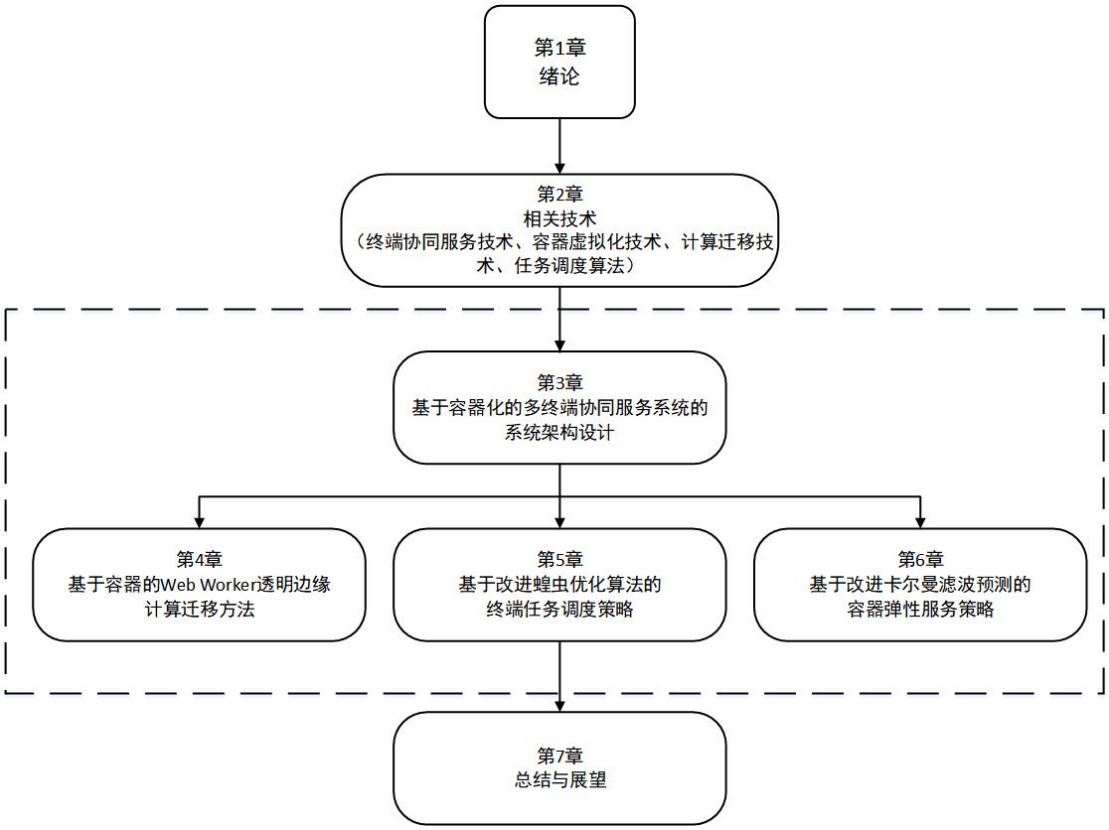

图 1.1 论文组织结构

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、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　引言

本文主要研究解决终端资源利用和终端服务优化问题的基于容器化多终端协同服务技术。在多终端协同服务技术中，利用虚拟化技术进行多终端的资源管理，利用计算迁移技术实现对多终端资源的应用，利用任务调度技术保持对终端资源利用的高效。本章对于本文的研究所涉及到的相关背景和相关技术进行铺垫和介绍，为后续章节对本文的研究内容进行详细介绍打下基础。

本章的内容结构组织如下：第2.2节介绍终端协同服务技术及相关的云计算技术、雾计算技术和边缘计算技术的相关概念和特点；第2.3节介绍了虚拟化技术特别是容器虚拟化技术的发展现状；第2.4节介绍了计算迁移技术的研究现状；第2.5节介绍了任务调度问题及元启发式算法的研究进展；第2.6节总结了本章内容。

2.2　终端协同服务技术

随着信息技术的快速发展，应用和服务对于计算、存储等资源的需求也越来越高，单一计算设备所能提供的计算、存储资源难以满足这些服务对于硬件计算、存储能力的需求。为了解决这一问题，越来越多的计算模式被研究人员所提出。云计算技术主要利用云数据中心近乎无限的资源为远端用户提供服务。随着物联网、大数据等技术的快速发展，传统的云计算模式存在实时性、安全性等问题，因此进行计算的位置逐渐由云端数据中心逐渐向网络边缘转移，出现了雾计算、边缘计算、海服务等计算模式。本研究所关注的终端协同服务技术是一种利用多个智能终端上的计算、存储等资源协同合作为用户提供服务的技术。终端协同服务技术主要关注网络边缘小范围内多个智能终端之间的资源利用与服务质量保证的问题，是一种自下而上的计算模式。虽然应用层次不同，但是传统的云计算、边缘计算等计算模式能够为终端协同服务技术提供很多经验。

2.2.1　云计算技术

随着处理和存储技术的快速发展以及互联网的普及，计算资源变得比以往任何时候都更便宜，更强大，更普遍。这种技术趋势使得云计算的计算模型成为现实。云计算是一种通过互联网管理和提供服务的范例 [17, 18]。云计算的思

想很早之前就存在，20 世纪 60 年代，约翰·麦卡锡已经设想计算设施将像一种工具（Unity）一样提供给公众 [19]。但是直到 2006 年谷歌公司（Google）使用“云计算”一词来描述通过网络提供服务的商业模型，云计算的概念才开始流行起来。云计算的概念虽然已经发展了很多年，但是并没有一个标准的定义 [20]，这里采用美国国家标准技术研究院（The National Institute of Standards and Technology，NIST）给出的定义 [17]：云计算是一种可以方便地通过网络来按需对可配置的共享计算资源池（如网络、服务器、存储、应用和服务等）进行访问的模型，可以以最少的管理工作或与服务提供商交互的方式来进行快速配置和发布。

云计算可以根据服务模式不同，划分为基础设施即是服务（Infrastructure as a Service，IaaS），平台即是服务（Platform as a Service，PaaS），软件即是服务（Software as a Service，SaaS）[21, 22]。云计算中的三种服务模式结构如图2.1所示 [23]。

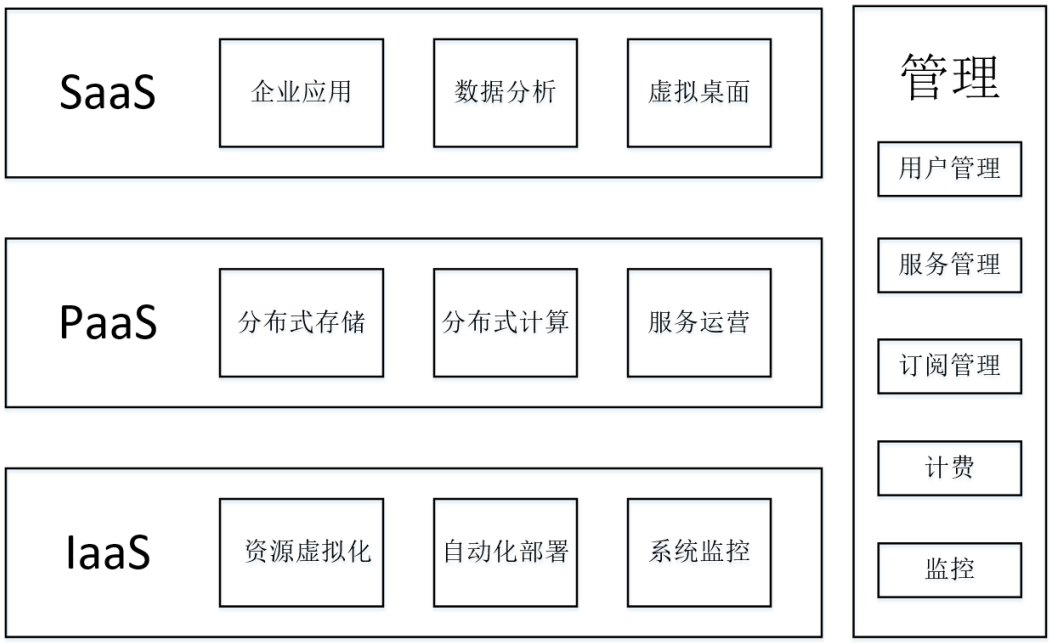

图 2.1 云计算中三种服务模式结构图

IaaS[24, 25] 是运行云所需要的硬件配置和基本服务的组合, 使用虚拟化技术将基础设施层的硬件资源如计算、存储、网络等封装起来并对外提供。IaaS 提供商在提供资源的同时，还会对资源进行管理和维护，用户可以通过合同议定或者按需付费（Pay-as-you-go）的方式购买资源。亚马逊的弹性计算服务（Elastic Compute Cloud, EC2）和简易存储服务（Simple Storage Service, S3）都属于 IaaS

服务。

PaaS 服务 [26, 27] 中，云服务提供商为企业或个人用户提供计算平台和开发、部署环境，包括相关软件应用程序集合、硬件和软件工具、虚拟机调度等等，允许开发人员通过网络来构建应用程序和服务。另外 PaaS 平台还能够满足开发人员对应用和服务进行灵活测试、快速部署、更新升级等需求。谷歌应用引擎（Google App Engine）就是一种典型的 PaaS 服务平台。

SaaS[28, 29] 是一种软件分发模型，这种模型中，应用程序由供应商或服务提供商托管并通过网络向客户提供，用户可以通过 WEB 浏览器、轻型客户端等方式对软件服务进行访问，而不必考虑底层运行平台的支持。SaaS 可以降低每个用户的初始化和维护的成本，因为这些成本由应用的所有用户共享。由于软件和服务部署在远程云数据中心的服务器中，SaaS 还可以拥有更好的可扩展性以及更方便的升级方式，用户在使用的过程中不需要考虑软件扩展和升级的问题。在信息时代，用户通过互联网访问的服务很多都是 SaaS，例如 Google、Facebook、YouTube 等等。

云计算还拥有以下特点：云计算可根据需要提供无限计算资源，从而消除了云计算用户对供应计划的需求；云计算可以消除云用户的提前计划配置要求，从而允许公司从小规模开始做起，只有在需求增加时才增加硬件资源，这对企业用户来说也很有吸引力；云计算拥有根据需要支付短期使用的计算资源的能力（例如，按小时计算的处理器资源和按天计算的存储资源），并根据需要释放它们 [30]。

但是在云计算中，用户设备的信息数据需要发送到集中式的云数据中心进行处理，长距离的信息传输会产生较高的响应时延，这会造成一些要求快速响应的服务质量较差。另外云计算模式中用户设备与远端的云数据中心进行数据交互，也会产生隐私泄露、用户信息被窃取等安全性问题。

2.2.2 雾计算技术

为了解决云计算中存在的高延迟、服务响应慢、用户隐私泄露风险高等问题，研究人员将能够提供计算和存储能力的设备部署在终端与云端之间，拉近用户终端与计算资源之间的距离，这样就产生了雾计算（Fog Computing）的概念。雾计算最早是由思科（Cisco）提出的一种计算模式。雾比云更接近地面，雾计算在云端数据中心与网络边缘终端之间引入雾层的概念，在雾层部署独立的计算、通信、存储、控制设备，在比云端数据中心更靠近用户的地方为用户提供服

务 [31]。

雾计算的架构图如图2.2所示 [31]。终端用户层是距离用户最近的一层，主要包括用户身边的一些智能终端设备，如手机、笔记本电脑等可移动终端，也包括传感器、智能设备等固定终端。终端用户层是雾计算模式中计算任务的发起者，计算任务通过接入层网络发送到雾层进行处理。接入层网络通常为有线局域网或无线接入网。雾层中的雾节点是部署在用户终端与云端之间的拥有计算、存储资源的服务设备，可以对用户终端发送来的任务进行实时处理，并将处理结果快速返回给用户终端。当收到一些较为复杂的、超出雾节点计算能力的计算任务，雾层会将这些任务通过核心网络层发送到云层中的云数据中心进行集中处理。

图 2.2 雾计算模式层次图

从思科对雾计算的定义与解释来看，雾计算被认为是云计算模式从网络核心到网络边缘的一种扩展范式。在思科提出的雾计算思想基础上，HP 实验室对雾计算给出了更进一步的定义：“一种大量异构的、普遍分布的、去中心化分布设备（通过无线网络进行连接，有的时候是自治的）互相通信互相协作并且在没有第三方干扰的情况下通过网络提供存储服务和执行计算任务的场景”[32]。IOx 是思科提供的雾设备产品，通过在连接网格路由器（Connected Grid Router,

CGR）上直接运行在虚拟机管理程序（Hypervisior）上的客户操作系统（Guest Operating System, GOS）中托管应用程序来进行工作 [33]，用户可以利用 IOx 框架来将自己开发的应用部署到雾计算支持的设备节点上。2015 年 11 月，ARM、思科、戴尔（Dell）、英特尔（Intel）、微软（Microsoft）和美国普林斯顿大学联合成立了开放雾联盟（Open Fog Consortium）该联盟旨在通过开发开放式架构、分布式计算、联网和存储等核心技术以及实现物联网全部潜力所需的领导力，加快雾计算的部署 [34]。

雾计算可以被认为是云计算的一种补充和扩展。根据文献 [35] 的描述，雾计算拥有如下特点：a) 低延时和位置感知能力；b) 广泛的地理分布；c) 移动性；d) 节点数量庞大；e) 主要采用无线网络接入；f) 强流媒体和实时应用；g) 异构性。在访问方式、控制管理、服务质量、采用设备、目标用户等方面与云计算还存在着很多不同之处，文献 [31] 对此进行了比较和总结，总结结果如表2.1所示。

雾计算拥有广泛的应用场景。文献 [35] 介绍了雾计算在车联网（Connected Vehicle, CV）方面的应用。车联网部署中会遇到大量的交互的场景，包括车辆与车辆之间的交互，车辆与接入点（Access Points, AP）的交互，AP 与 AP 之间的交互等等。雾计算具有移动性、低延时、异构性、位置感知、实时交互、遍布城市和街道等特点，可以为车联网场景提供合适的支撑平台。城市中的智能交通灯系统也可以利用雾计算技术，智能交通灯与本地的传感器进行交互，检测行人和骑车人，测量车辆距离和速度，在雾层进行计算，协调绿灯的变化。

文献 [36] 介绍了雾计算在智能家居（Smart Home）方面的应用。随着物联网的快速发展，越来越多的智能设备和传感器在家庭中连接起来提供智能家居服务。但是由于设备的异构性，不同供应商的产品很难一起工作。由于终端设备资源有限，一些需要大量计算和存储的任务，如实时视频分析等服务还是不能够正常运行。雾计算可以将所有碎片化的终端设备资源整合到一个平台中，并为这些智能家居应用提供弹性资源。智能家居中的安全应用可以通过独立的 VM 虚拟机部署在雾计算平台中，利用雾层的本地资源实时处理传感器记录的数据，提供低延时的安全服务。

文献 [33] 介绍了雾计算在移动大数据分析方面的应用。雾计算可以为大规模数据处理系统提供弹性资源，而不会出现云计算中高延迟的问题。局部数据的聚合工作可以在雾节点中运行，特别是在面对一些紧急情况的时候，雾节点可以提供及时的反馈。而对于数据更详细的分析，例如计算密集型任务，可以被安排在云端进行处理。雾和云协同工作，可以进行大数据的采集、聚合和预处理，减

表 2.1 雾计算和云计算的比较

方面	具体指标	云计算	雾计算
通信访问方式	节点的访问方式	通过互联网	通过本地网络设施
	服务的访问方式	通过核心网	通过边缘设备
	终端用户到节点距离	多跳	一跳
	对于移动前传和集中式的基带单元的负担	重	轻
	缓存与无线信号处理	集中式	兼具集中式与分布式
	无限资源管理	集中式	兼具集中式与分布式
服务质量	时延	较高（分钟级 月级）	较低（毫秒级 秒级）
	时延抖动	高	很低
	可用性	99.99%	不定
	数据传输过程受攻击可能性	可能性高	可能性很低
	安全策略	难以定义	易被定义
	移动性支持	有限支持	支持程度高
	服务来源	全球	本地
	使用成本	高	低
	网络要求	高	低
	对实时应用的支持	差	好
	任务的传输功耗	大	小
服务质量	终端用户数	数十万或数百万	数十万
	节点数目	少	多
	硬件设备平均成本/美元	1 500 3 000	50 200
	设备部署位置	远端数据中心	靠近网络边缘
	硬件	存储容量大、计算能力强	存储容量与计算能力有限
	设备部署环境	带制冷设备的大型仓库	小型仓库或室外
	服务器拥有者与管理者	大公司（如 Google）	小公司或个人
	部署速度（代价）	慢	快
通信访问方式	节点的访问方式	通过互联网	通过本地网络设施
	服务的访问方式	通过核心网	通过边缘设备
	终端用户到节点距离	多跳	一跳
	对于移动前传和集中式的基带单元的负担	重	轻
	缓存与无线信号处理	集中式	兼具集中式与分布式
	无限资源管理	集中式	兼具集中式与分布式
通信访问方式	内容产生者	主要是人	主要是传感器设备
	内容丰富程度	丰富	单一
	目标用户	互联网用户	移动用户

少数据的传输和存储，平衡数据处理的计算能力。

2.2.3 边缘计算技术

相比雾计算，边缘计算（Edge Computing）将计算资源推向离用户更近的网络边缘。边缘计算的思想跟雾计算有相似之处，这里采用文献 [2] 中提出的边缘计算的定义：“边缘计算是指在网络边缘执行计算的一种新型计算模式，边缘计算中边缘的下行数据表示云服务，上行数据表示万物互联服务，而边缘计算的边缘是指从数据源到云计算中心路径之间的任意计算和网络资源”。根据这个定义，边缘计算的模式如图2.3所示 [37]。

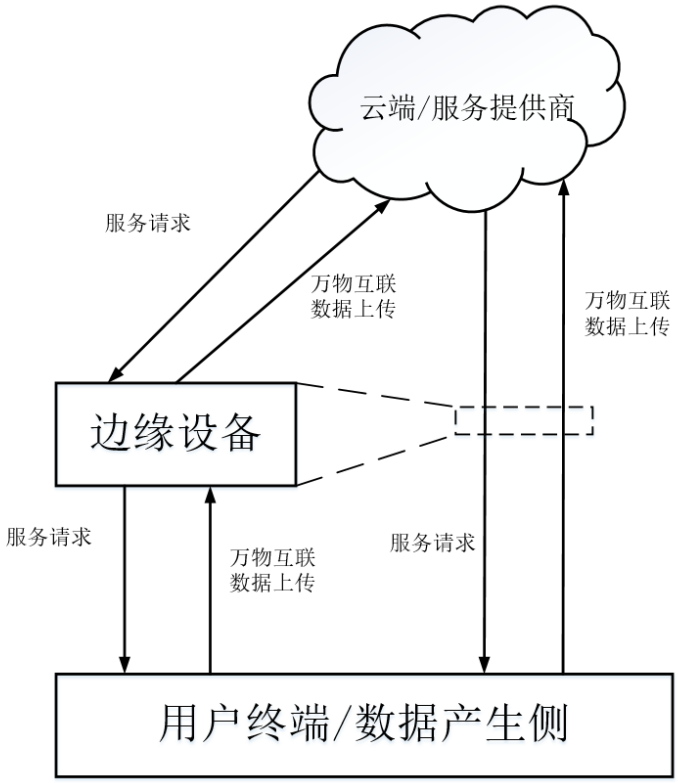

图 2.3 边缘计算模式结构图

在边缘计算范例中，边缘终端设备不仅是数据消费者，而且还充当数据生产者。在边缘终端上不仅可以从云端请求服务和数据，还可以作为云端来执行计算任务。边缘侧可以执行计算迁移，数据存储、缓存和处理等任务，并且可以从云侧向用户分发请求和交付服务 [37]。

边缘计算产业联盟（ECC）与 2018 年 12 月发布了《边缘计算参考架构 3.0》[38]，指出边缘计算通过与不同行业的特点相结合，能够在很多应用场景中给出创新的解决方案。边缘计算的应用场景包括梯联网、工业机器人、能效管理平

台、轨道交通装备预测性维护、能源网、指挥交通等等。

文献 [3] 提出了一种利用 MEC 在智慧城市（Smart City）场景中支持各种应用的框架。为了保证高体验质量（Quality of Experience, QoE），该框架引入了“跟随边缘”（follow me edge）的概念，边缘服务器提供的服务会根据各自用户的移动而移动。这种解决方案能够减少核心网络流量，确保低延时，保证用户体验质量。文献中的一些不同条件下的边缘移动性实验结果表明，存储类型和内存容量对于迁移延时的影响非常显著。

文献 [39] 研究了边缘计算的具体实现方法，提出了一种基于透明计算的物联网架构，提供解耦软件与硬件设备的面向服务计算模式，让用户可以透明地使用服务，而不需要考虑底层操作系统和异构的硬件平台。文献利用所提出的的基于透明计算的物联网架构构建可扩展的轻量级可穿戴设备服务（明明为 TCwatch），并通过比较 TCwatch 和传统智能手表在不同应用大小的情况下的延迟和能耗情况来评估 TCwatch 的动态服务性能。

文献 [13] 指出，利用 Linux 容器在边缘运行应用是一种利用边缘计算的方式。文献通过以下几个方面来评估利用 Docker 容器作为边缘计算平台支撑的性能：a) 部署和终止；b) 资源和服务管理；c) 容错；d) 缓存。实验结果表明 Docker 能够提供快速快速的部署、较少的资源占用以及良好的性能，可以作为较好的边缘计算平台支撑技术。

相比云计算，边缘计算拥有很多优点 [40]。利用边缘设备上的计算能力可以执行很多用户终端的计算任务，可以为云端数据中心分担计算压力，而且基于边缘设备对一些用户上传数据的预处理，可以抛弃大量冗余数据，精简压缩上行数据包，缓解网络带宽压力。边缘设备距离用户侧也更近，一些原本需要到云端去执行的任务可以放到边缘设备上来进行计算，减少传输延迟，缩短服务响应时间，提高用户体验。另外，由于在边缘计算的模式中，数据可以在边缘设备上进行处理，而不用将被上传到远程云数据中心，这就减少了隐私数据传输过程总被窃取以及在云端被第三方盗用的风险，提高了用户隐私数据的安全性。。严格意义上来讲，本文所研究的多终端协同服务技术，也属于边缘计算中的一种场景。

2.2.4 终端协同服务技术

随着信息技术的快速发展，人们对于终端设备所提供的娱乐、安全、智能生活等服务的需求越来越高，传统的单一终端为用户提供计算、存储资源的服务模式越来越难以满足人们对终端应用的服务质量的需求。如何在人们日常生活与

生产的环境中利用网络组织共享各类终端的计算、存储资源，协同起来为终端用户提供高质量的服务成为研究人员关注的问题，终端协同服务技术也因此而诞生。

协同服务目前并没有一个官方统一的定义，文献 [41] 中对协同服务的定义如下：协同服务指在开放的网络环境下，在具有一定主体性的软件实体之间建立通信联系，约束其交互，以使之和谐工作，而达成既定应用目标的过程。本研究所关注的终端协同服务技术是一种利用多个智能终端上的计算、存储等资源协同合作为用户提供服务的技术。终端协同服务技术主要关注网络边缘小范围内多个智能终端之间的资源利用与服务质量保证的问题，是一种自下而上的计算模式。终端协同服务技术的结构图如图2.4所示。

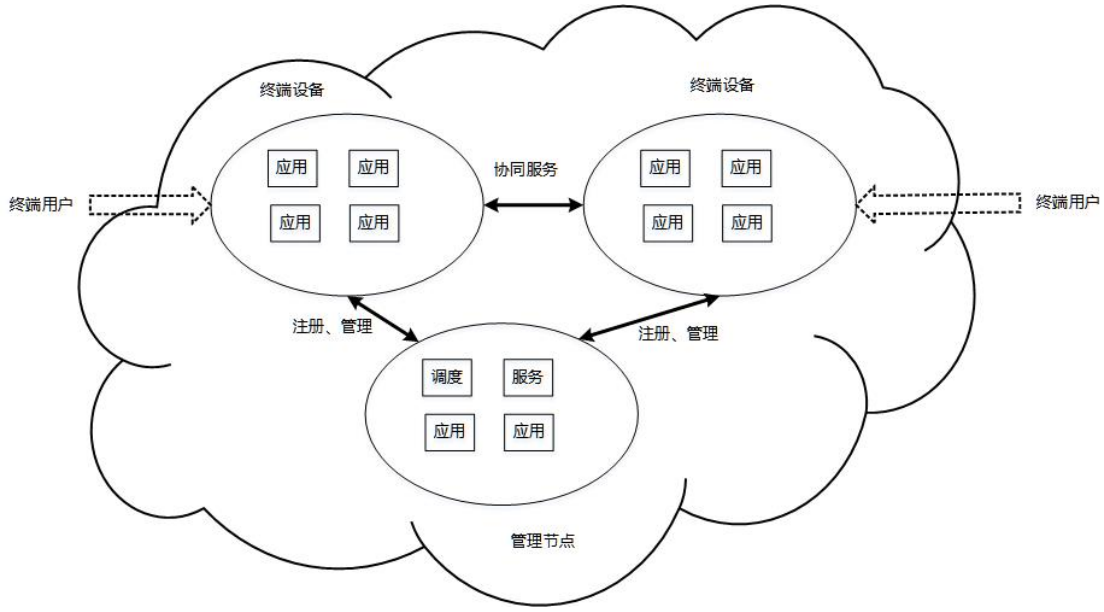

图 2.4 终端协同服务技术结构图

文献 [42] 介绍了一种被称为嵌入式云（Embedded Cloud）的设计。嵌入式云利用资源受限的异构终端设备协同服务，共享资源，提升性能，支撑更加智能的更加普遍的应用。嵌入式云设计的组成包括分布式过程描述语言（Process Description Language, PDL），分布式中间件（Distributed Middleware, DiMiWa）和基础设施。在分布式中间件和基础设施的帮助下，分布式过程描述语言可以在异构的终端设备上执行分布式过程并作为资源共享服务。

文献 [15] 介绍了一种针对嵌入式物联网终端设备协同服务的基于容器的资源管理系统的方法。该方法基于一种名为设备云（Device Cloud）的物联网架构，将云计算范式应用到终端协同服务场景中，允许用户通过使用抽象化的资源池来共享和分配终端设备资源。文献中提出的方法是基于 Docker 容器虚拟化技术

的，可以利用底层操作系统级虚拟化的资源隔离、开销小等优势。该方法引入了一种架构，可以利用 Docker 自动部署模型，允许在不同用户和设备之间的容器内共享资源。所提出的方法比较关注关注操作系统级虚拟化技术用于隔离用户代码和动态分配智能设备提供资源的可行性。

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 [43]。而虚拟化技术是解决资源管理问题的一个重要的解决办法。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 [44]。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 [45]。虽然随着计算机技术的发展，硬件价格的下降与用户需求的变化使得虚拟化技术的研究与应用一度陷入低谷，但是随着云计算等概念的兴起以及互联网服务模式的革命性变化，虚拟化技术又重新成为学术界和工业界研究的热点 [46]。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 [47, 48]。

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 [49]。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。服务提供商利用虚拟化技术，以虚拟机的形式为用户提供服务，可以让一台服务器同时支持多个用户独立访问，且每个用户之间是完全独立的，提高了服务提供商的资源利用率，降低了云服务提供的成本，同时虚拟机互相隔离的特性也提升了云服务的安全性。

虚拟化技术为云计算技术打下了良好的基础。基于虚拟化技术的云计算模型如图2.5所示 [50]。第2.2节中提到，云计算中所能够提供的服务包括基础设施即是服务（IaaS）、平台即是服务（PaaS）和软件即是服务（SaaS），这三种服务

模式都离不开虚拟化技术的支持。IaaS 能够为用户提供计算、存储等资源，这些都需要利用虚拟化技术来进行资源的抽象、资源数量的划分，并且封装成虚拟机来提供给用户远程使用。在用户使用这些资源的过程中，也要受到虚拟化技术对资源使用数量和时间的监控和管理，以及对虚拟机生命周期的管理。PaaS 能够为用户提供计算平台、相关软件应用程序集以及应用部署的平台。PaaS 除了利用虚拟化技术对资源进行管理，还利用了虚拟化技术的封装的特性，将提前封装好的包含相关软件应用程序集的应用运行环境提供给用户，并且以虚拟机为单位进行部署和运维。SaaS 在以上两种服务的基础上，利用了虚拟化技术的隔离和透明的特性，将应用程序封装在虚拟机中，通过网络向用户提供服务，用户不必关心应用本身是运行在虚拟机中还是独立的物理服务器中。这些虚拟机彼此之间是隔离的，一台服务器可以同时运行多个虚拟机独立为用户提供服务，同一台服务器上的不同虚拟机之间不会互通，保证了服务的安全性。另外，虚拟化技术在面对云计算中的分布式计算、权限管理、服务器灾难恢复等问题以及在为科学计算中的网络、安全等领域提供实验环境方面也能够提供比较合适的解决方案 [51]。

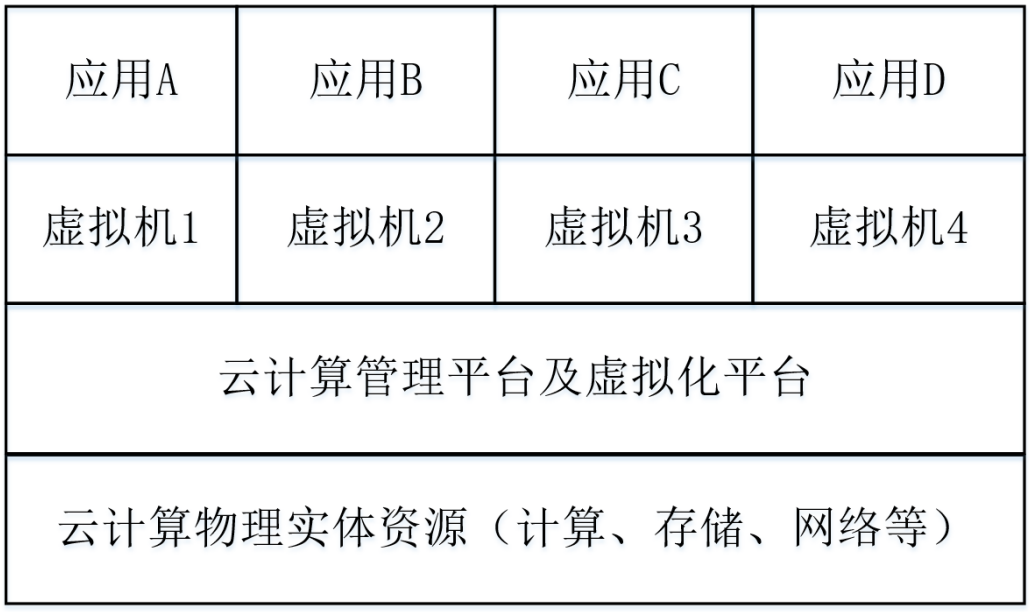

图 2.5 基于虚拟化技术的云计算模型

虚拟化技术自 1959 年被提出，经历了硬件分区技术、完全虚拟化技术、半虚拟化技术和操作系统虚拟化技术等几个发展阶段 [52]，从底层硬件层逐渐向操作系统层过渡，逐渐成为一种成熟的资源管理技术。

2.3.1 硬件分区技术

硬件分区技术是早期的虚拟化技术。硬件资源被划分为互相独立的分区，每个分区的存储和计算资源是独立的，也会安装独立的操作系统 [53]。硬件分区技术可以做到对资源进行比较细粒度的划分，但这种划分比较简单，不够灵活，也不方便系统对资源进行管理。硬件分区技术的架构图如图2.6所示 [52]。

图 2.6 硬件分区技术架构

2.3.2 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, Host OS）上面架设一层虚拟机监控器（Virtual Machine Monitor, VMM）[54] 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[55][56]。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, 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2.7所示 [57]。

云计算行业内曾经非常流行的工具如 VMware[58, 59]、KVM[60]、VirtualBox[59] 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。

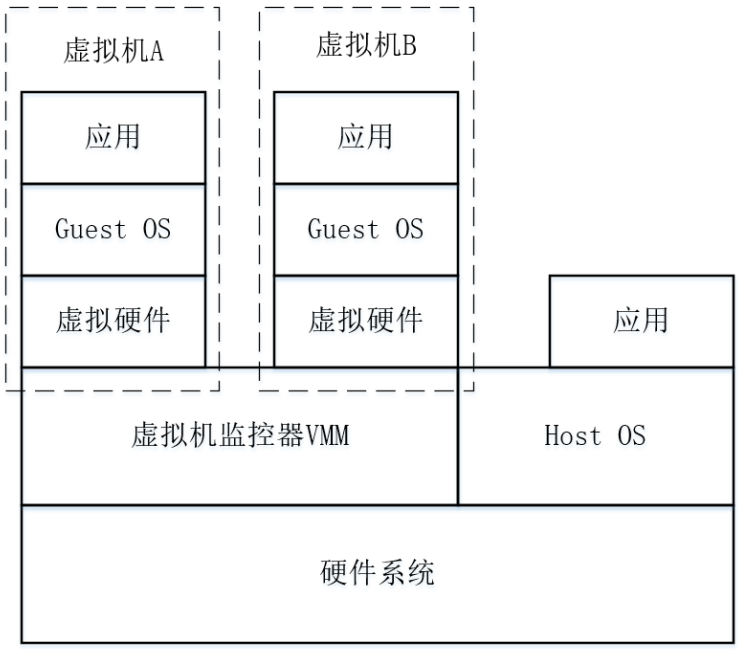

图 2.7 完全虚拟化技术架构

2.3.3 半虚拟化技术

半虚拟化技术（Para-Virtualization）是由完全虚拟化技术发展而来的。为了改善完全虚拟化技术的性能，半虚拟技术对客户操作系统内核做了一些优化，将一些底层 CPU 执行的任务转移到操作系统内核中完成。半虚拟化技术的架构图如图2.8所示 [51]。半虚拟化技术的代表是 Xen[61]。半虚拟化技术比完全虚拟化技术性能好，但是需要对操作系统内核进行修改。

2.3.4 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 [62]。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

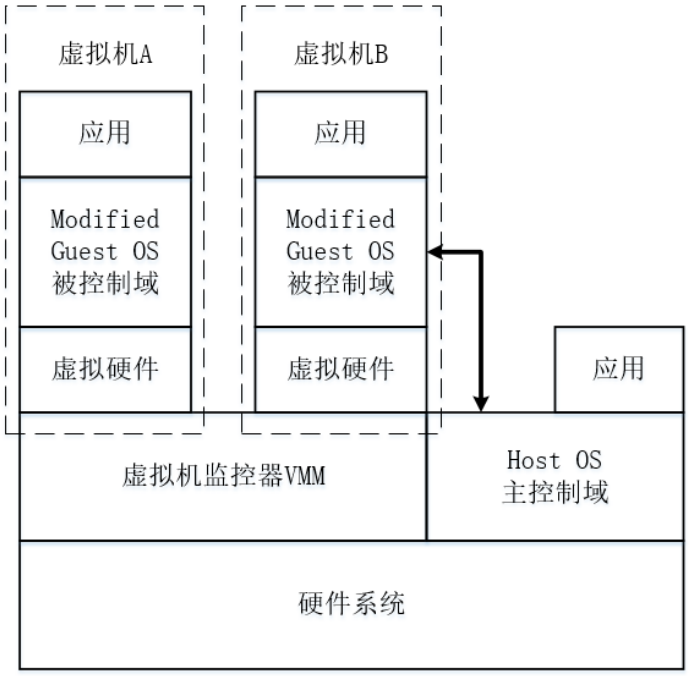

图 2.8 半虚拟化技术架构

拟化技术的架构如图2.9所示 [63]。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 [64]。

2.3.5 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 [65]。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 [66]。LXC 技术的核心技术包括 Namespace 技术和 Cgroups 技术 [52]。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 [67]。六种 namespace 隔离的系统调用如表2.2所示。

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，

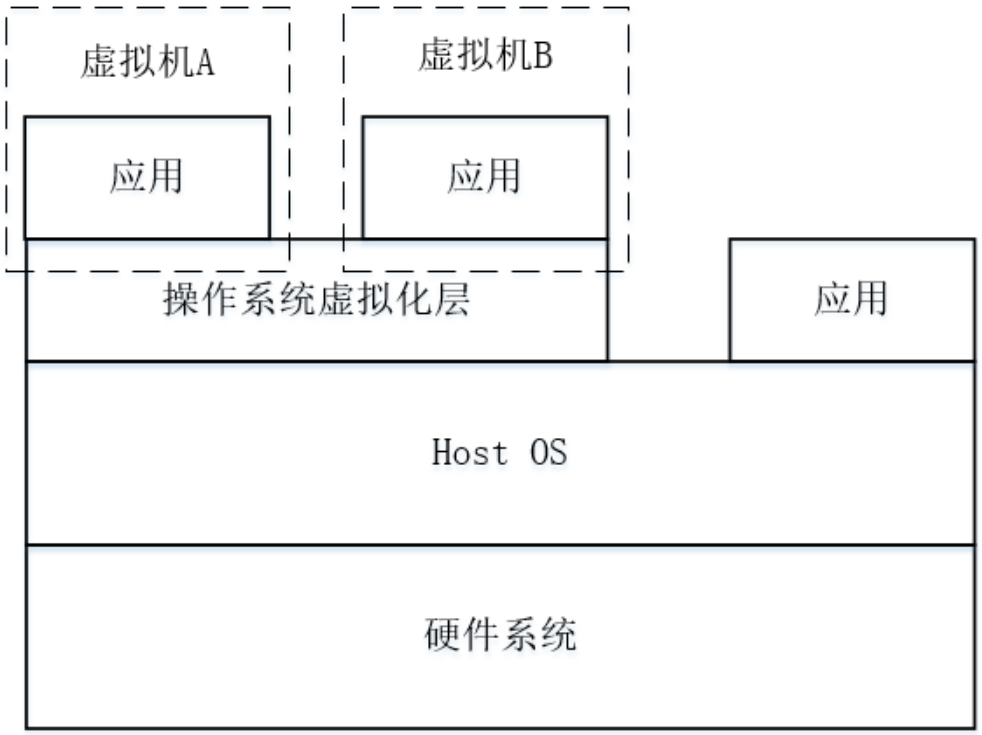

图 2.9 操作系统级虚拟化技术架构

表 2.2 Namespace 隔离的六个方面

Namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

利用 Cgroups 控制组，也能够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。

由于不需要通过 VMM 对底层硬件进行虚拟化工作，因此相比完全虚拟化技术和半虚拟化技术等传统虚拟化技术，容器虚拟化技术拥有很多优点，包括启动时间短、资源利用率高、虚拟机性能好、部署密度大等等，但同时其利用 Namespace 命名空间进行隔离的方式也使得容器虚拟化技术的隔离性不如传统虚拟化技术。表2.3对容器虚拟化技术与传统虚拟化技术的特点进行了对比和总结 [68]。

表 2.3 容器虚拟化技术与传统虚拟化技术特点对比

特点	容器虚拟化技术	传统虚拟化技术
启动时间	秒级	分钟级
资源利用率	高	低
占用空间	小，甚至 KB 级别	非常大，甚至达到 GB 级别
性能	接近宿主机本地进程	比宿主机差
部署密度	几千	几十
运行形态	运行在宿主机内核上	运行在 VMM 上

目前行业内最流行的基于容器虚拟化技术的产品是 Docker[69, 70]。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。Docker 的运行模块结构图如图2.10所示。

Docker 使用了传统的 Client-Server 架构模式（C-S）。Docker Client 为客户端，Docker Daemon 为常驻后台的服务端。用户通过 Unix Domain Socket（UDS）在 Docker Client 与 Docker Daemon 之间建立 HTTP 通信，用户对于 docker 的操作请求会被 Docker Client 翻译成 HTTP 通信指令，并发送到 Docker Daemon，通过提供的 API 接口进行执行。Docker Daemon 是 Docker 架构中的主体部分，是松耦合结构，包括 Dockerfile 编译、Registry 镜像仓库通信、Graphdriver 镜像管

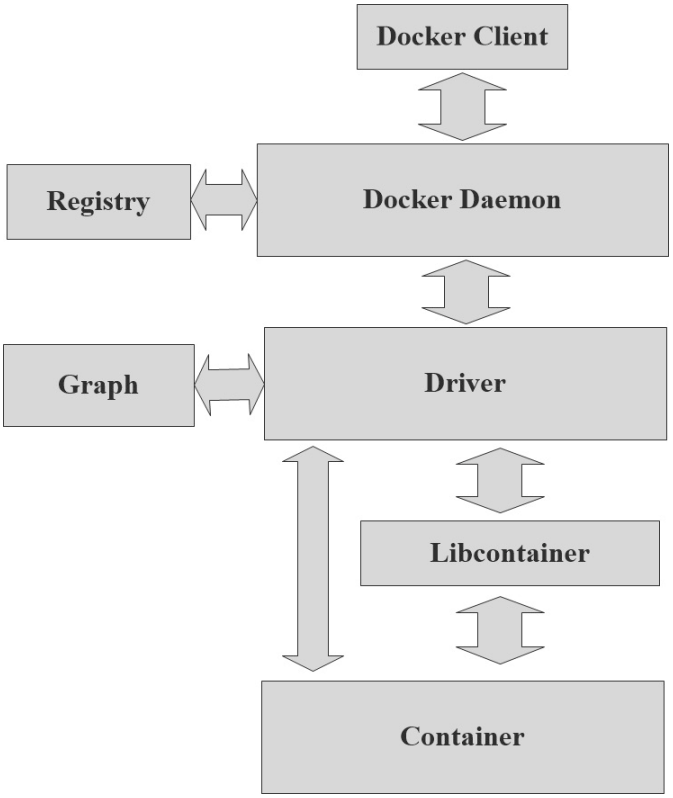

图 2.10 Docker 运行模块结构图

理驱动、Driver 文件系统驱动、Libcontainer 容器管理等模块，不同模块各司其职并有机组合，完成用户的请求。当用户需要下载镜像 Image 创建 Docker 容器时，首先，通过 Docker Client 向 Docker Daemon 发出命令请求，Docker Daemon 解析后，会从远程 Registry 中下载镜像。随后，Docker 通过 Driver 中的镜像管理驱动 Graphdriver 将下载的镜像以 Graph 的形式存储在本地。接下来，镜像管理驱动 Graphdriver 从 Graph 本地镜像存储目录中获取指定的镜像，并按照指定的规则为容器准备 rootfs，rootfs 以 readonly 方式加载并检查，然后在其上挂载 writeable 层形成一个容器的运行目录。常见的 Docker 镜像的文件系统结构图如图2.11所示。

当需要为 Docker 容器创建网络环境时，则通过 Driver 中的网络管理驱动 Libnetwork 创建并配置 Docker 容器的网络环境；当需要限制 Docker 容器运行资源或执行用户指令等操作时，则通过 Driver 中的执行驱动 Execdriver 来完成。Libcontainer 是一套实现容器管理的 Go 语言解决方案。这套解决方案实现过程中使用了 Linux 内核特性 Namespace 与 Cgroups，同时还采用了 Capability 与文件权限控制等其他一系列技术。Libcontainer 的设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的 API。Docker 可以直接调用 Libcontainer，而最

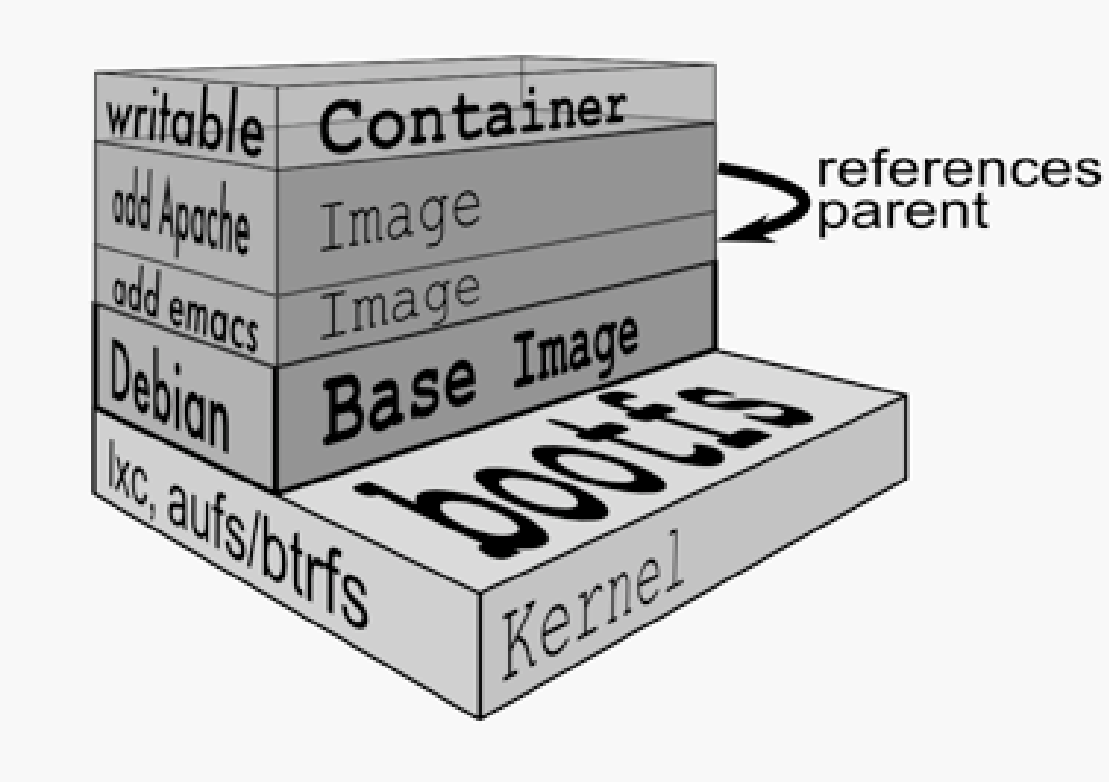

图 2.11 常见 Docker 文件系统结构图

终操纵容器的 Namespace、Cgroups、Apparmor、网络设备以及防火墙规则等，这一系列操作的完成都不需要依赖 LXC 或者其他包。基于这些特性，除了创建容器之外，Libcontainer 还可以完成管理容器生命周期的任务。目前 Libcontainer 的目标是实现容器技术的统一 API，即无论底层使用什么技术，只要实现了 Libcontainer 定义的一组接口，Docker 都可以在上面运行。这也为 Docker 实现全面的跨平台带来了可能。为了实现这一目标，Libcontainer 也逐渐从 Docker Daemon 中剥离出来，形成了 Containerd。Containerd 向 Docker 提供运行容器的 API，二者通过 grpc 进行交互。Containerd 最后会通过 runc 来实际运行容器。

2.4 计算迁移技术

2.4.1 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，终端设备资源受限问题使得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 [71]。计算迁移技术（Computation Offloading），也被称为计算

卸载技术，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。计算迁移技术的目的是扩展终端设备的计算能力，为用户提供质量更好的终端服务。

经过十几年的发展，计算迁移技术逐渐成为云计算和边缘计算中的一项重要技术 [72]。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，利用一部分通信开销来置换一部分计算、存储等资源，为用户提供质量更好的服务 [73]。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)[74]。但是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 [37]。

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)[75, 76]。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 [77]。

文献 [78] 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、能连接网络的、计算资源丰富的计算设备或计算机集群（被称为 cloudlet）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 cloudlet 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 [79]。cloudlet 模式是一种粗粒度的计算迁移技术 [77]。cloudlet 的实现过程需要迁移整个 VM Overlay，迁移时间较长，需要 60-90s，当终端附近的 cloudlet 资源不足以支撑起计算迁移服务的时候，终端会向远程的云端资源请求协助 [71]。cloudlet 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 cloudlet 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 [80, 81]。另外，对于 cloudlet 的计算迁移技术的研究也还存在着缺乏统一的部署和管理方法的不足 [71]。

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 [82]。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 [83]。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 [84–88]。文献 [89] 中提出了一种基于 Web Worker 的非透明迁移方法，通过重写 Web 应用的库，将终端上 HTML 的 Web Worker 迁移到其他服务器或其他终端设备上进行计算。文献 [90] 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

任务调度问题是一种在云计算 [91]、边缘计算 [92]、海服务 [14] 等领域常见的问题。在基于容器化的多终端协同服务技术中，不同的终端所能够提供的资源类型和大小不同，不同的任务对于不同类型资源的消耗情况也不一样，当然不同任务在不同节点上执行的费用和时间也不同。智能终端的资源是受限的，合理规划任务在节点上的执行调度，能够减少资源碎片，提高终端资源利用率 [70, 93, 94]。任务调度问题是一种 NP-hard 问题 [95]，难以通过计算得到精确的数值最优解，需要通过将任务调度问题划归为一类优化问题（Optimization Problem），建立寻优的数学模型，利用一些寻优算法来获得一个近似最优解。常用的解决方法是传统的基于贪心策略的任务调度优化算法，以及元启发式优化算法。

2.5.1 传统任务调度算法

有很多传统的基于贪心策略的任务调度优化算法 [96]。先进先出算法（FIFO Scheduler）是按照任务到达的先后顺序进行调度 [97]，Max-Min 算法是优先调度执行时间最长的任务，Min-Min 算法是优先调度执行时间最短的任务，[98, 99]。这些基于贪心策略的任务调度算法的优点是原理比较简单，方便实施，运行速度快。由于这些算法的原理过于简单，优先满足局部最优选择，导致其缺点是很容易陷入局部最优解，得不到效果更好的解决方案。尤其是当任务规模扩大的时

候，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，基于贪心策略的传统任务调度算法无法处理这种情况。

2.5.2 元启发式算法

近年来，很多元启发式算法（meta-heuristic algorithm）被应用于解决任务调度问题 [100, 101]。Goldberg 在 1988 年提出的遗传算法（Genetic Algorithm，GA）是一种非常著名的进化算法，它将自然选择理论引入到优化过程中，使用一条染色体来代表任务调度问题的一种调度方案，使用包括变异、交叉和选择在内的几个自然算子来作为演进迭代的计算法则 [102, 103]。由于存在交叉和变异的操作，GA 计算起来非常复杂，实现起来也比较困难。

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（Particle Swarm Optimization Algorithm，PSO）是 Kennedy 于 1995 年提出的经典元启发式算法，原理非常简单，容易实现 [104–106]。蚁群优化算法（Ant Colony Optimization Algorithm，ACO）的灵感来自蚂蚁在巢穴和食物来源之间搜索的自然觅食行为，蚁群优化算法利用化学信息素在蚁群搜索单位之间进行信息交流 [107–109]。人工鱼群算法通过模仿鱼类的随机游动的觅食行为来进行搜索 [110]。

近年来，一些较新的元启发式算法也被提出，而且以这些元启发式算法为基础的改进并不多。2015 年提出的蚁狮算法（Ant Lion Optimizer，ALO）的灵感来自于蚁群的狩猎行为 [111]。鲸鱼优化算法（Whale Optimization Algorithm，WOA）于 2016 年提出，该算法模拟鲸鱼的自然狩猎行为 [112]。2016 年提出的蜻蜓算法（Dragonfly Algorithm，DA）基于自然界中蜻蜓群的静态和动态行为进行搜索 [113]。2017 年，Shahrzad Saremi 和 Seyedali Mirjalili 提出了一种名为蝗虫优化算法（Grasshopper Optimization Algorithm，GOA）的新型元启发式优化算法。蝗虫优化算法利用群体内相互作用力的影响和群体外的风力及食物来源修正的影响来模拟蝗虫群的迁移及觅食行为，以求寻找到目标食物 [114]。蝗虫优化算法利用群体智慧，通过共享蝗虫群体的经验来确定搜索方向并找到最佳或近似最佳位置。蝗虫优化算法还使用演进的方法进行多次迭代，以使群体智慧更加有效。

2.6 本章小结

本章主要介绍了本文研究所涉及到的相关技术的概念以及研究现状。本章首先介绍了终端协同服务技术及相关的云计算技术、雾计算技术和边缘计算技术的概念与特点；其次介绍了虚拟化技术的起源与发展现状；然后介绍了计算迁移技术的研究现状及几种计算迁移方案的特点；最后介绍了任务调度问题和解决方法，尤其是几种近几年新提出的元启发式优化算法。本章相关技术的介绍，成为本文后续几章研究工作的基础。

第 3 章　基于容器化多终端服务系统架构设计

3.1　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 [8, 115–117]。云计算已经发展成为互联网的重要基础计算技术，能够提供全面的服务 [118]。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 [10]。但是，使用云端计算资源来为终端设备提供协同计算服务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 [13, 119]。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 [120]。

近年来，随着云计算的功能逐渐向网络边缘移动，在网络边缘设备上进行计算成为一种趋势 [121]。利用位于网络边缘的终端设备进行协同计算，能够拉近资源提供方与用户之间的距离，减少网络传输所带来的时延，保证终端服务的实时性，减少公共网络带宽负载的占用，缓解网络拥塞情况，还能够降低用户个人隐私信息被盗用的风险。另外，在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，将这部分空闲资源组织利用起来，为终端用户提供协同计算服务，能够以较低的部署成本来提高终端资源利用率和提高终端用户体验。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。由于终端设备资源分布较为分散，终端资源异构

性强，终端设备动态性、移动性强等原因，多终端协同服务技术还是会面临不少问题，包括如何对分散的终端资源进行管理，如何对终端异构性资源进行统一提供，如何在终端上部署终端协同服务，多个终端节点如何进行协同服务等等。为了解决这些问题，本章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异构资源进行虚拟化，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。

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，包括微服务架构的起源与发展；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 相关研究

随着互联网时代的到来，云计算技术和物联网技术快速发展，互联网应用提供商的硬件资源与软件资源越来越多地以服务的形式提供给用户 [122]。越来越大的用户体量、越来越复杂的服务内容对互联网公司和服务提供商的可扩展性、敏捷性、容错性等方面的能力提出了越来越高的要求，这也促使后台网络应用服务的架构形式在不断地演进，从早期的单体式架构 (Monolithic Architecture) 到后来的面向服务架构 SOA(Service-Oriented Architecture)。

在这个演进过程中，微服务架构 (Microservice Architecture) 由面向服务架构 SOA 慢慢发展而来。微服务架构于 2012 年开始出现技术雏形，并逐步取代传统的单体式架构。2014 年学者 Martin Fowler 正式提出微服务架构的概念 [123]，与此同时，容器技术的快速发展为微服务架构的大规模使用提供了基础支撑。2014 年至今，微服务架构已成为行业内最流行的服务架构。

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。

3.2.1 微服务架构国内研究现状

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 [124] 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 [125] 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 [126] 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 [127] 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 [128] 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 [129] 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 微服务架构国外研究现状

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 [130] 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 [131] 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 [132] 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 [133] 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 [134] 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 [135] 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 [136] 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 微服务架构特点

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 [137] 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 [123]：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 [123] 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 [137] 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。

从微服务架构的这些特点中可以看出，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。

3.3.2 容器虚拟化技术

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, 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，所以本研究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, 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。引入容器这种轻量级虚拟化技术，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。

3.3.3 基于容器化多终端协同服务系统层次设计

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，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

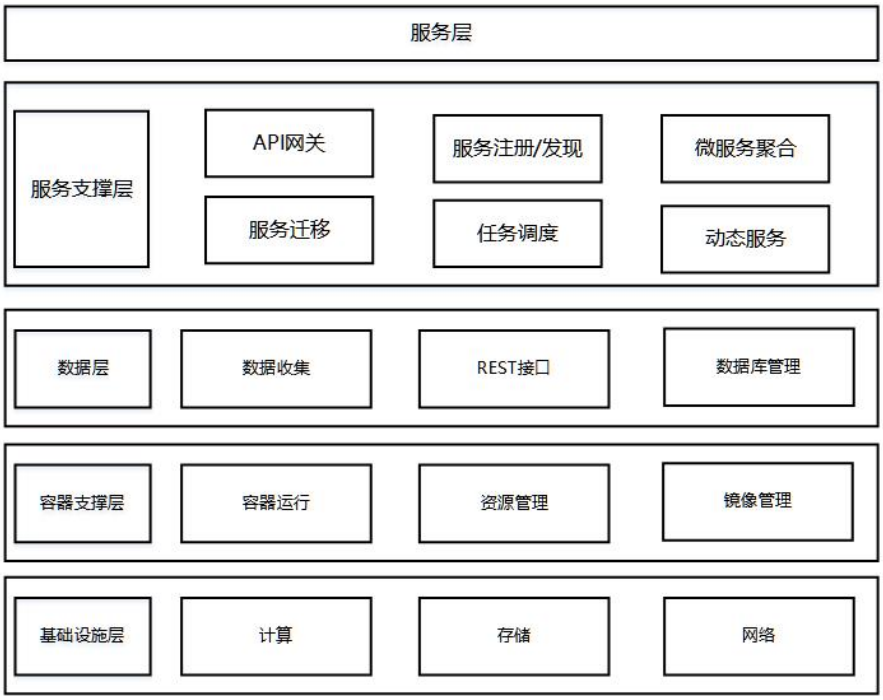

图 3.1 基于容器化多终端协同服务系统层次图

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。

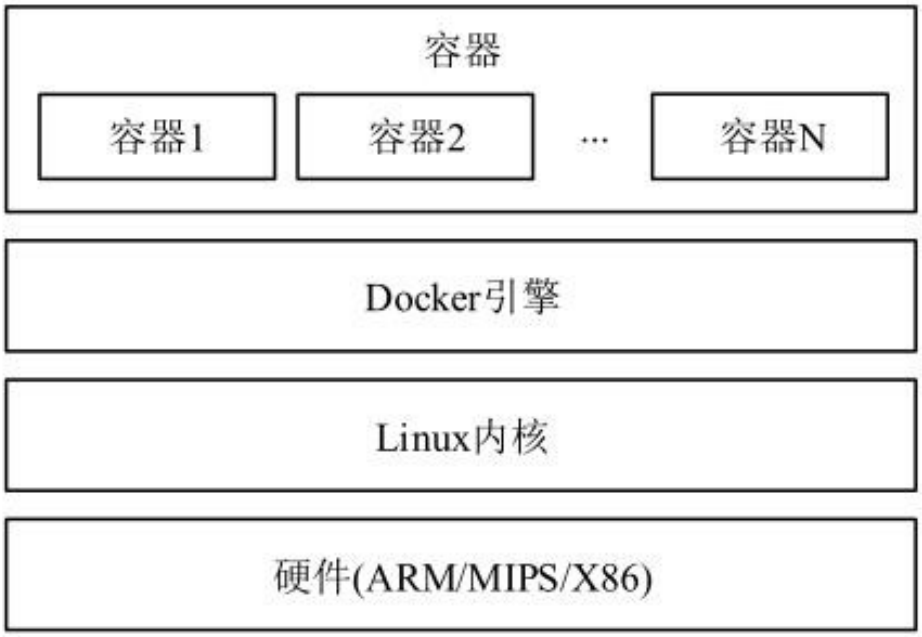

图 3.2 Docker 容器终端部署架构图

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。

3.3.4 基于容器化多终端协同服务系统架构设计

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。

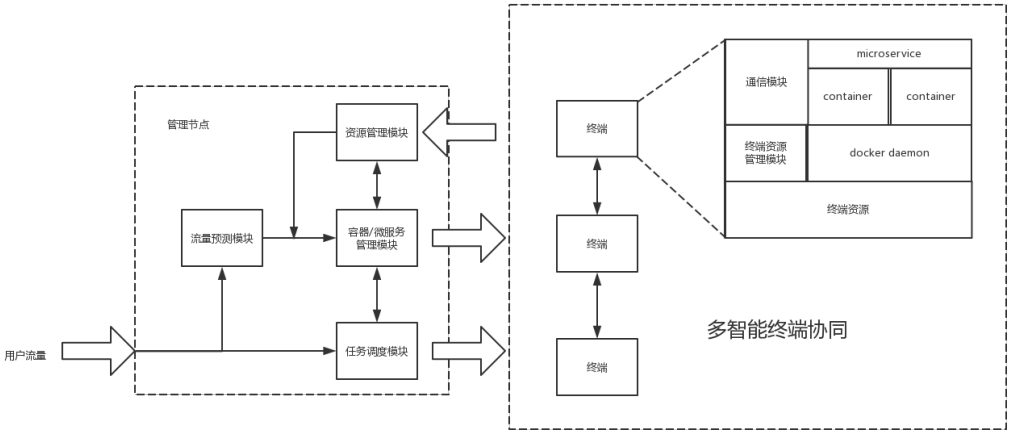

图 3.3 基于容器化多终端协同服务系统架构图

在基于容器化多终端协同服务系统架构图中，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服务模块、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 代表着层次图中的容器支撑层，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图中的服务支撑层，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。

3.3.5 多终端协同服务系统中的角色分析

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以很好地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。

在基于容器化多终端协同服务系统中，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，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。

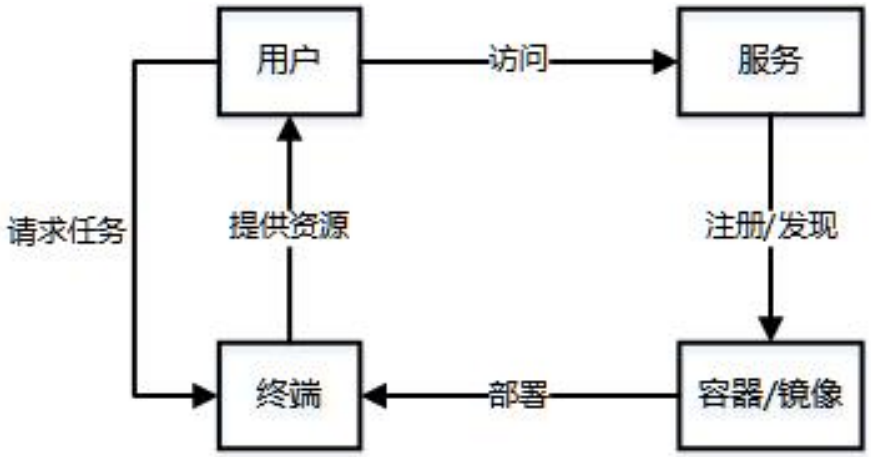

图 3.4 用户、服务、终端、容器概念之间的关系

3.4 多终端协同服务系统的构建方法

3.4.1 自治系统网络选择

互联网是由很多大大小小的自治系统 (Autonomous System, AS) 组成的 [138]，自治系统可以通过自组织网络来进行拓扑网络结构设计。自组织网络可以应用于很多领域 [139]。自组织网络的组网方式有两种模式，一种方式是优先选择若干骨干节点，并以骨干节点为中心进行扩散，逐渐将所有可用节点连接到自治域中 [140, 141]；另一种方式则是每个节点通过于邻居节点进行通信，互换消息，再根据邻居选择策略进行选择加入，逐渐建立一个联通的自治域 [139]。自组织网络与现在大部分网络不一样，现阶段大部分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自组织系统架构。

3.4.2 去中心化自组织网络构建

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。但是这并不意味着一个自治域在最初创建以后就一直不会经历大的变动。由于终端设备具有较强的动态性，经常发生节点上下线的行为，随着时间的变化，会逐渐导致自治域结构落后或者出现一些节点不受自治域管理等意外情况。所以需要对自治域的网络结构进行周期性地重新划分 [142]。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。

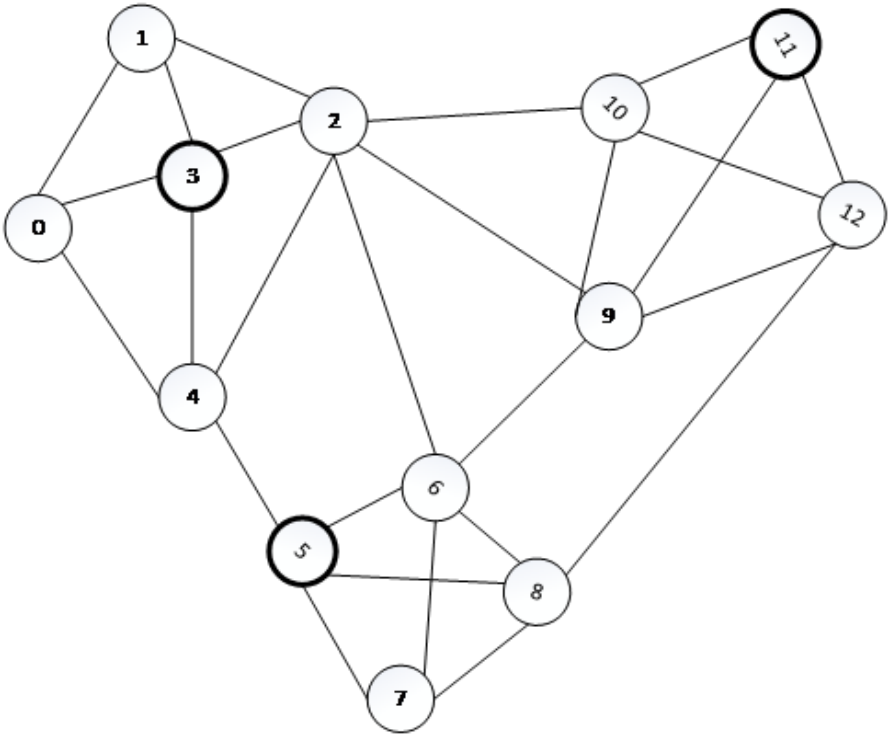

图 3.5 去中心化的自组织网络结构

3.5 本章小结

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，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。由于利用云计算计算为终端设备提供协同服务的模式存在着网络延迟高、实时性差、占用公共网络带宽资源、用户隐私信息不安全等问题，考虑将用户周围的终端设备上的空闲资源利用起来，多个终端协同为用户提供服务。

为了提高终端设备资源利用率，提高终端用户体验，本章设计了基于容器化的多终端协同服务系统。本章的主要工作包括：1）引入容器虚拟化技术对多终端异构资源进行虚拟化，形成资源池，可以为上层终端服务按需使用；2）参考微服务架构，提出多终端协同服务系统架构，包括资源管理模块、服务管理模块、任务调度模块、弹性服务模块等，为多终端协同服务技术的具体方式提供了一种方法；3）提出了去中心化的自组织网络结构，方便终端自治域进行多终端节点管理和节点组网。

本章所涉及的研究成果包括：

论文 1 篇：“微服务架构评述”（网络新媒体技术，核心期刊，已发表）。

专利 2 篇：“一种应用容器的启动方法及系统”（申请号：201610534217.X）。

“一种微服务故障检测处理方法及装置”（申请号：201711368632.3）。

软著 1 篇：“视频点播平台客户端软件”（登记号：2017SRBJ0226）

第 4 章　基于容器的多终端透明计算迁移技术

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。

本章的内容组织结构如下：第4.2节介绍了计算迁移技术的相关研究工作，包括云计算中的计算迁移技术、基于 Web Worker 的计算迁移技术以及虚拟化技术；第4.3节介绍了基于边缘容器的 Web Worker 透明计算迁移系统的模型设计，以及客户端、服务端的具体实现方法；第4.4节列出了对比实验的结果及分析；第4.5节总结了本章内容。

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 [71]。计算迁移技术（Computation Offloading），也被称为计算卸载技术，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)[74]。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 [37]。

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)[75]。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 [77]。

文献 [78] 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 cloudlet）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 cloudlet 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 [79]。cloudlet 模式是一种粗粒度的计算迁移技术 [77]。cloudlet 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 cloudlet 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 [80, 81]。

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 [82]。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 [83]。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 [84–88]。文献 [90] 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 [44]。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 [143–146] 都使用了虚拟机 VM 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开销以及资源浪费 [147]。文献 [148] 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。OS 虚拟化技术以 Docker 为代表，是一种更轻量级的虚拟化解决方案 [149]。这种轻量级的虚拟化技术不仅能够减少启动时间带来的时延，也能够极大降低虚拟化技术带来的额外资源开销。

4.3 基于边缘容器的 Web Worker 透明计算迁移技术

为了将靠近用户的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 计算迁移技术。计算迁移技术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闲状态，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利用。另外对 Web 运行环境代码进行修改，设计对 Web 应用开发者透明的计算迁移系统，将 Web Worker 迁移到服务端执行。

4.3.1 系统结构设计

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，在本研究中为用户终端上的 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，或预计执行时间较长，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 来执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，边缘容器集群是由多个用户终端设备组成的，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，接收其他设备迁移过来的计算任务。

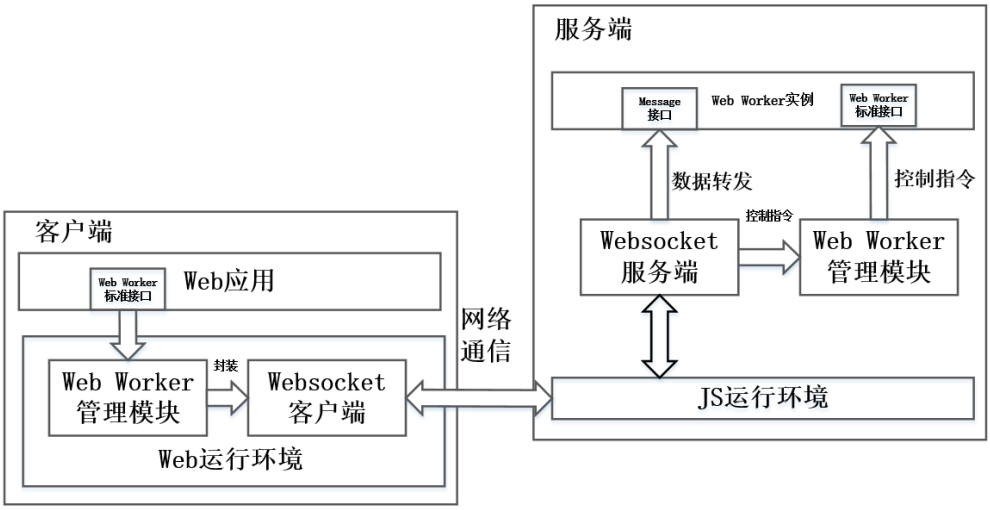

图 4.1 Web Worker 透明计算迁移

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 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 通信服务端用来接收客户端发来的消息，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，接收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。

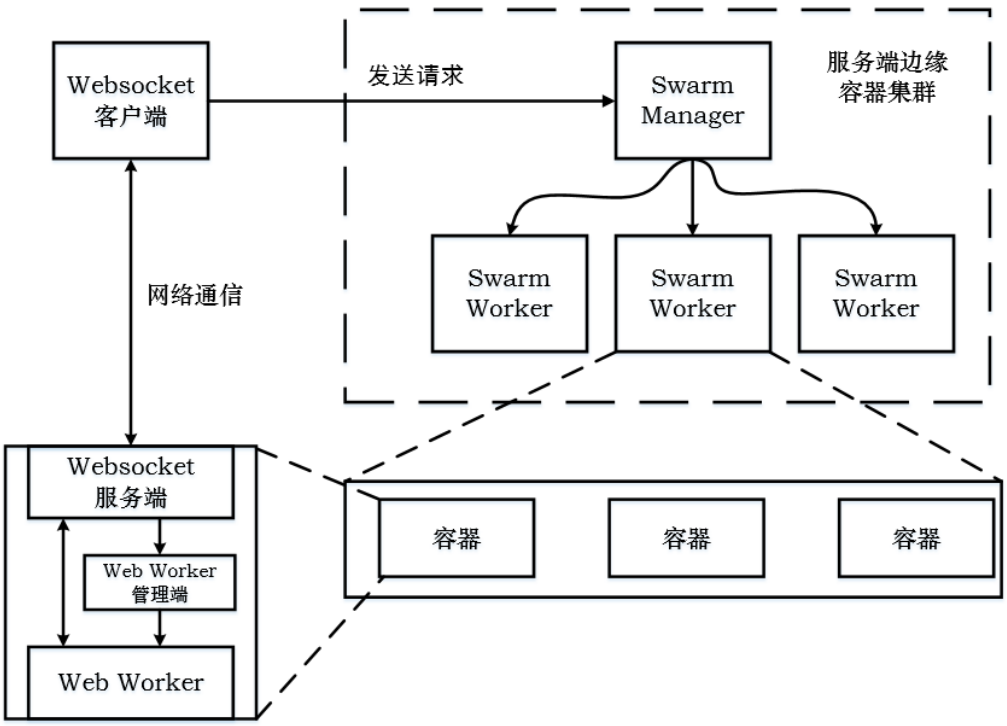

图 4.2 透明计算迁移系统服务端结构图

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，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)。在这个实验中，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，图片的大小是一定的，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 需要处理多个任务的情况。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，这体现了所提出的的计算迁移技术的透明性。经过修改后的 chromium 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，本研究完成了容器集群迁移性能测试实验。在本实验中图片大小为 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4.3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。

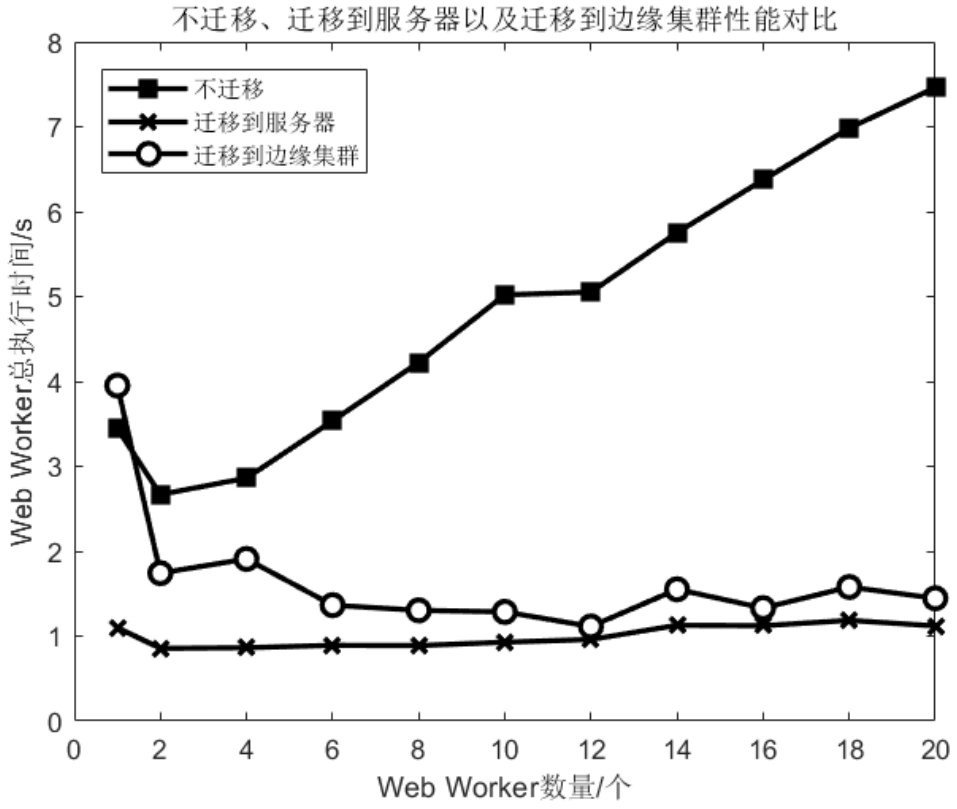

图 4.3 容器集群迁移性能测试实验

从图4.3中可以看出，当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，计算执行时间大大增加，用户服务体验显著变差。当迁移到边缘树莓派集群上的时候，相比不进行计算迁移的情况，终端服务质量提升明显，虽然总计算执行时间比迁移到服务器的情况稍差，但相差不是很显著，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，相比不进行计算迁移的情况拥有较好的服务质量控制能力。当 Web Worker 数量为 20 的时候，不进行计算迁移时候的总执行时间为 7.461 秒，透明迁移到边缘树莓派集群时候的总执行时间为 1.445 秒，优化效果为 80.6%。当迁移到服务器上的时候，计算执行时间最短，尤其是在 Web Worker 数量为 1 的时候，效果非常明显，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，而且在实际的应用场景中，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实验预期的效果。

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，本研究完成了透明计算迁移性能测试实验。在本实验中，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4.4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。

从图4.4中可以看出，相比容器集群迁移性能测试实验，图片大小变大，总任务量变多，总的计算时间也相应增加。相比其他两种计算迁移方式，不使用计算迁移系统的情况的计算执行总时间最长，服务质量最差，而且随着 Web Worker 数量的增加，计算时间大大增加，用户的服务体验显著变差。当 Web Worker 数量为 20 的时候，不进行计算迁移时候的总执行时间为 11.131 秒，透明迁移到边缘树莓派集群时候的总执行时间为 3.515 秒，性能提升 68.4%。另外值得注意的一点是，在不使用计算迁移系统的情况下，当使用 4 个 Web Worker 来执行任务的时候，总计算执行时间最短，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，会出现因各个线程之间抢占 CPU 而导致切换时间大大增加影响总执行时间的情况。当使用计算迁移进行计算的时候，总的执行时间都会大大缩短，尤其是当 Web Worker 数量多于 4 个的时候，服务性能优化效果十分明显，这也是基于边缘树莓派集群中拥有多设备多核处理器的原

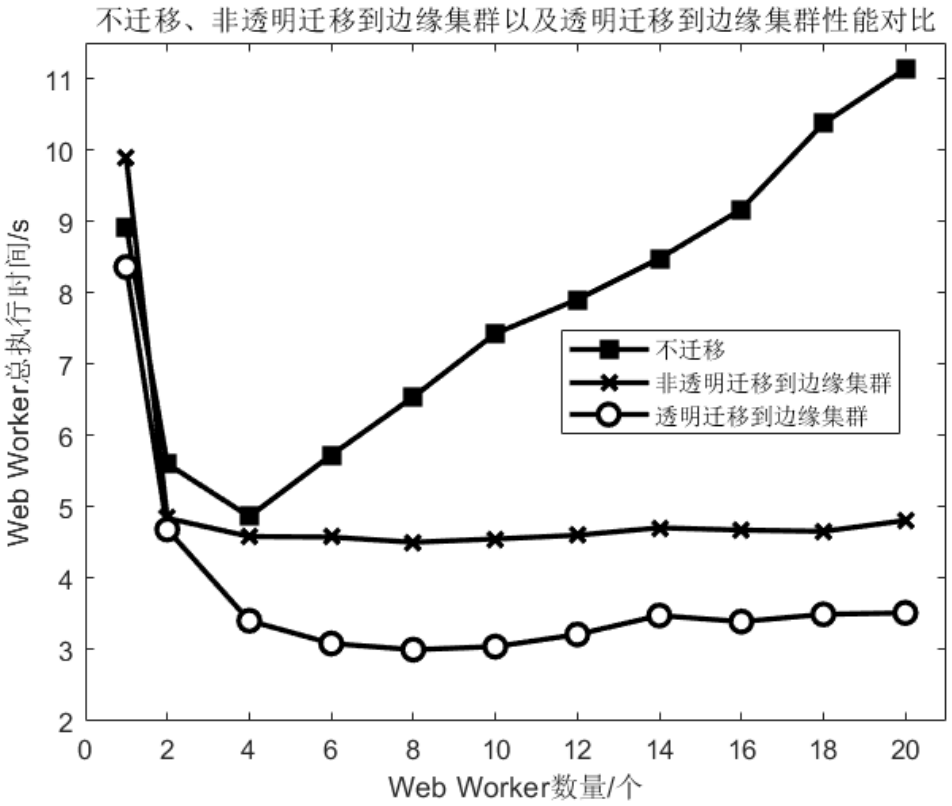

图 4.4 透明计算迁移性能测试实验

因。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。当 Web Worker 数量为 8 的时候，使用非透明计算迁移方式的计算执行总时间为 4.551 秒，使用透明计算迁移方式的计算执行总时间为 3.000 秒，性能提升 33.1%。

4.5 本章小结

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。在基于容器化多终端协同服务技术中，计算迁移技术成为将用户周围终端设备上的空余资源利用起来，为用户提供协同服务的具体实现方式。

本章基于多终端协同服务技术的应用场景，提出了一种基于容器的 Web

Worker 透明边缘计算迁移技术。本章通过重新设计开发客户端 Web 运行环境，改写其中的 Web Worker 相关接口，在对上层 Web 应用保持透明的情况下，利用 WebSocket 将生成 Web Worker 的请求发送到服务端进行计算，实现了 Web Worker 的透明计算迁移。另外，利用 Docker Swarm 容器集群，将透明计算迁移的服务端部署到用户周围的边缘终端上面，方便对边缘终端设备的空余资源进行管理和利用。本章通过两个实验证明所提出的基于容器的 Web Worker 透明边缘计算迁移技术可以很好地利用用户周围的边缘终端设备的空余资源，相比不使用计算迁移、直接在本地进行计算的方式，可以大大减少 Web Worker 的总执行时间，当 Web Worker 数量增加的时候，也能够将 Web Worker 总执行时间的增长情况控制在一个相对很小的水平。实验结果表明，所提出的基于容器的 Web Worker 透明边缘计算迁移技术，对于减少 Web 应用中 Web Worker 的总执行时间，提高用户体验有着很显著的效果。

本章所涉及的研究成果包括：

论文 1 篇：“基于容器的 Web Worker 透明边缘计算迁移系统”（微电子学与计算机，核心期刊，在投）。

第 5 章　多终端协同服务任务调度算法

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。

本章的内容组织结构如下：第5.2节介绍了任务调度问题算法的相关研究工作，包括传统的基于局部贪心策略的任务调度算法以及基于元启发式算法的任务调度算法；第5.3节介绍了蝗虫优化算法（GOA）的背景、原理、数学模型以及数学表达式；第5.4节提出了一种带随机跳出机制的动态权重蝗虫优化算法（Dynamic Weight Grasshopper Optimization Algorithm with Random Jumping，DJGOA），介绍了其随机跳出机制以及动态权重策略，列出了所提出的带随机跳出机制的动态权重蝗虫优化算法在测试函数上的测试结果，并根据测试结果对带随机跳出机制的动态权重蝗虫优化算法的性能进行了分析；第5.5节在第5.4节提出的算法的基础上，进一步提出了一种改进蝗虫优化算法（Improved Grasshopper Optimization Algorithm，IGOA），介绍了其非线性舒适区控制参数、基于 *Lévy* 飞行的局部搜索机制以及基于线性递减参数的随机跳出策略，列出了所提出的改进蝗虫优化算法在测试函数上的测试结果，并根据测试结果对改进蝗虫优化算法的性能进行了分析；第5.6节介绍了任务调度问题，提出了多终端协同服务环境中的任务调度问题的数学模型，并将第5.5节所提出的改进蝗虫优化算法应用到该模型中来求最优解，进行了仿真实验并分析实验结果；第5.7节对本章内容进行了总结。

5.2 相关工作

5.2.1 传统任务调度问题求解方法

任务调度问题是将多个任务调度到约束下的多个节点的优化问题。任务调度问题是一个 NP-hard 问题 [95]，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。有很多传统的基于贪心策略的任务调度优化算法 [96]。先进先出算法（FIFO Scheduler）是按照任务到达的先后顺序进行调度 [97]，Max-Min 算法是优先调度执行时间最长的任务，Min-Min 算法是优先调度执行时间最短的任务，[98, 99]。这些基于贪心策略的任务调度算法的优点是原理比较简单，方便实施，运行速度快。但同样是因为这些算法的原理过于简单，优先满足局部最优选择，导致其缺点是很容易陷入局部最优解，得不到效果更好的解决方案。尤其是当任务规模扩大的时候，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，基于贪心策略的传统任务调度算法无法处理这种情况。

5.2.2 启发式算法求解方法

近年来, 学者们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（Meta-Heuristic Algorithm）被用来处理复杂的优化问题。元启发式算法具有操作简单和开销较少的优点, 能够在最优化问题（Optimization Problem）中通过评价、迭代、演进等方式逐步找到全局最优解或近似全局最优解。

Goldberg 于 1988 年提出了遗传算法 (Genetic Algorithm, GA)[102, 103, 150, 151]。遗传算法是一种经典的元启发式算法，将自然选择理论引入到寻优的过程中，算法模拟了达尔文生物进化论中的包含自然选择和遗传机制的生物进化过程的计算模型，通过模拟自然进化的过程来搜索问题空间内的最优解。遗传算法用一个染色体来表示一种可行的解，并利用交叉、变异、选择等运算符实现可行解的进化，经过若干轮的不断迭代和演进，最终得到比较好的全局最优解或近似最优解。虽然遗传算法的性能非常好, 但是遗传算法的操作比较复杂, 运算量较大，收敛速度较慢，不适合应用于多终端协同任务调度问题的求解。Kirkpatrick 于 1983 年提出模拟退火算法 (Simulated Annealing Algorithm,SA)，是一种基于概率的算法。模拟退火算法模拟固体退火过程，从某一较高初温出发，随着温度参数的不断下降, 动态调整跳出的概率，在解空间中随机寻找目标函数的全局最优解，即在局部最优解中以一种动态的概率进行跳出并最终趋于全局最优 [152–154]。但尽管如此，模拟退火算法还是很容易陷入局部最优。

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。Kennedy 和 Eberhart 在 1995 年提出粒子群算法 (Particle Swarm Optimization, PSO)，是一种非常经典的元启发式算法 [104–106]。该算法最初是受到飞鸟集群活动的规律性启发，利用群体智慧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。粒子群算法的原理简单，易于实现, 而且性能非常好。蚁群优化算法 (Ant Colony Optimization Algorithm, ACO) 是受蚂蚁在蚁巢和食物来源之间自然觅食行为的启发而提出的一种算法 [107, 108, 155]。蚁群算法利用化学信息素在蚂蚁群之间进行通信和交换信息，以控制整个蚁群的搜索方向。布谷鸟算法 (Cuckoo Search Algorithm, CS) 是 Yang 在 2009 年提出的一种元启发式算法 [156]。布谷鸟算法的灵感来自布谷鸟的繁殖行为，该算法使用 *Lévy* 飞行来跳出局部最优。

近年来，研究人员提出了一些新的元启发式算法。2015 年提出的蚁狮算法 (Ant Lion Optimizer, ALO) 是受到蚁狮的捕猎行为所启发 [111]。2016 年，研究人员模仿鲸鱼捕猎的自然行为，提出了鲸鱼优化算法 (Whale Optimization Algorithm, WOA)[112, 157]。2016 年受自然界中蜻蜓集群的一些静态和动态的行为，研究人员提出了蜻蜓优化算法（Dragonfly Algorithm, DA）[113, 158–160]。在 2017 年，Shahrzad Saremi 和 Seyedali Mirjalili 提出了一种新的元启发式优化算法，蝗虫优化算法 (Grasshopper Optimization Algorithm,GOA)[114]。蝗虫优化算法结合蝗虫群体内部的相互作用力和来自蝗虫群外的风力、重力以及食物吸引力的影响来模拟蝗虫群的迁徙、觅食行为，寻找目标食物。蝗虫优化算法能够比较好地运用群体智能的力量，通过在蝗虫集群内部分享经验，进行多轮迭代演进，不断修正搜索方向，最终找到最优位置或者近似最优位置。

自蝗虫优化算法提出以来，研究人员在此基础上提出了一些改进优化算法，但不是很多，提升效果也不够好。Ewees 等人于 2018 年提出了基于反向学习的改进蝗虫优化算法 (Improved Grasshopper Optimization Algorithm Using Opposition-Based Learning, OBLGOA)[161]。基于反向学习的改进蝗虫优化算法引入了反向学习策略，以生成的反向解作为备选的可行解，使用这种策略可以提高算法的收敛速度，但是由于反向学习策略缺乏随机性，对算法性能的提升效果较为有限。Sankalap Arora 等人于 2018 年提出了混沌蝗虫优化算法 (Chaotic Grasshopper Optimization Algorithm, CGOA)[162]。混沌蝗虫优化算法引入混沌映射因子来提高算法性能。文献中使用了 10 种不同的混沌映射因子来测试混沌理论的效果，但是因为不同的混沌因子在处理不同的 benchmark 测试函数的时候效果不同，对

算法整体的效果提升不够理想。由于蝗虫优化算法本身能够利用群体智慧，不断迭代演进，直到搜索到最优或近似最优解，算法性能较好，但是研究人员对其的改进缺乏随机性，改进效果不够明显，所以本研究通过引入随机跳出因子、动态调节参数、*Lévy* 飞行等方法，提出了一种带随机跳出机制的动态权重蝗虫优化算法，优化算法性能，并在此基础上提出了一种改进蝗虫优化算法，进一步提升算法性能，并将其应用于解决基于容器化多终端协同服务任务调度问题。

5.3 蝗虫优化算法

蝗虫优化算法模拟了自然界中蝗虫群的迁徙和觅食的行为。蝗虫群为了寻找一个有食物的新栖息地，不断进行迁徙。在这个过程中, 蝗虫群内部蝗虫之间的相互作用里会对每一个蝗虫个体造成位置的影响。来自蝗虫群外的风的力量和重力也会影响蝗虫集群整体的移动轨迹。另外，对于迁徙过程来说，目标食物的位置也是一个重要的影响因素。这个迁徙过程贯穿了蝗虫的一生，包括幼虫期和成虫期。在蝗虫幼虫期，其群体的主要特点是在局部地区的小步伐和缓慢移动。而在蝗虫的成虫期，其群体的特点是长距离移动和比较突然的跳动。

优化问题的搜索过程可以被划分为两个阶段，探索阶段 (exploration process) 和开发阶段 (exploitation process)。在探索阶段中，鼓励搜索单元 (Search Agent) 进行快速、突然的移动，寻找更多的潜在目标区域。而在开发阶段中，搜索单元往往只在局部进行移动和搜索，以寻找附近更优、更精确的解。蝗虫的集群中成虫期和幼虫期两个阶段的特点与优化问题搜索过程中的探索阶段和开发阶段这两个阶段的特点非常符合。将蝗虫集群抽象为一群搜索单元，进行数学建模，借鉴其迁徙和觅食的移动过程的特点，可以提出蝗虫优化算法。

Seyedali Mirjalili 在文献 [114] 中提出了蝗虫群体迁徙的数学模型。具体的模拟公式如公式5.1所示。

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

在公式5.1中，变量 X_i 是第 i 个搜索单元的位置，变量 S_i 代表蝗虫集群内部搜索单元之间的相互作用力对第 i 个搜索单元的影响程度，变量 G_i 代表蝗虫集群外部重力因素对第 i 个搜索单元的影响程度，变量 A_i 代表风力的影响因素。变量 S_i 的定义公式如公式5.2所示。

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

在公式5.2中，变量 d_{ij} 代表第 i 个搜索单元和第 j 个搜索单元之间的欧式距离，计算方法如公式5.3所示。

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

在公式5.2中，变量 $\widehat{d_{ij}}$ 代表第 i 个搜索单元和第 j 个搜索单元之间的单位向量，计算方法如公式5.4所示。

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

在公式5.2中，s 是一个函数，用于计算蝗虫集群之间的社会关系 (social relationship) 影响因子，该函数定义如公式5.5所示。

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

在公式5.5中，e 是自然底数，变量 f 代表吸引力因子，参数 ql 代表吸引力长度。在应用于解决数学优化问题的时候，为了优化数学模型，公式5.1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。再加上迭代过程中的时间参数，这样演进计算公式就变成了如公式5.6所示。

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

在公式5.6中，x^iter_i 代表第 $iter$ 次迭代过程中的第 i 个搜索单元的位置。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探索阶段和开发阶段两个阶段的比例。参数 c 的计算方式如公式5.7所示。

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

在公式5.7中，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。

在整个搜索过程中，每个位置的优劣要靠适应度函数 (fitness function) 来判断，在优化问题中，适应度函数通常为所要求解的目标函数，通过适应度函数计算所得到的值为适应度值 (fitness value)。在任务调度问题中，适应度函数通常是根据具体任务调度模型所建立的评价函数。在优化问题的求解过程中，公式5.6作为演进公式，被不断循环迭代来寻找最优解，每轮迭代得到的最优的适应度值即被视为目前得到的最优解的值，得到最优解的位置则被视为当前的食物目标位置，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：

算法 1 蝗虫优化算法

初始化蝗虫集群，设置位置边界 u 和 l 的值
初始化参数，包括 *cmax*, *cmin*, *MaxIteration*, *iter* 等
使用随机矩阵初始化所有搜索单元的位置
计算目标适应度值，记录目标位置
while $(iter < MaxIteration$ 且 $targetfitness > destinationfitness)$ **do**
    通过公式5.3和公式5.4计算 d_ij 和 $\widehat{d_ij}$ 的值
    通过公式5.5计算 $s(d_ij)$ 的值
    通过公式5.6来更新搜索单元的位置 x_iter_i
    计算适应度值
    if 当前适应度值优于目标适应度值 **then**
        更新目标适应度值和目标最优解位置
    end if
    $iter = iter + 1$
end while
返回得到的目标最优值和目标最优解的位置

5.4 带随机跳出机制的动态权重蝗虫优化算法 (DJGOA)

蝗虫优化算法性能良好，具有理论基础简单和易于实现的优点。原始的蝗虫优化算法改变了蝗虫的舒适区域，可以使搜索单元通过迭代收敛到全局最优解。但是蝗虫算法仍然存在着一些不足之处。

蝗虫算法使用线性递减参数进行搜索过程收敛，这使得搜索过程的探索和开发两个阶段很难区分。蝗虫优化算法无法充分利用所有的搜索迭代次数，当最大迭代次数增加时，例如迭代次数从 500 次提高到 1500 次时，蝗虫优化算法得到的最优适应度值并没有出现明显的改进。另外，由于缺乏随机性，蝗虫优化算法搜索过程很容易陷入局部最优。而蝗虫优化算法原理简单、易于实现的优势对于跳出局部最优帮助不大，这反而可能成为蝗虫优化算法的一种劣势。

5.4.1 动态权重

蝗虫优化算法使用线性递减参数来控制蝗虫的舒适区并使所有搜索单元向目标位置靠拢。但是线性递减参数不能增强搜索过程中探索和开发这两个阶段的影响。在探索阶段，蝗虫优化算法不能向目标搜索区域周围快速收敛，搜索单元只是在整个搜索空间中游荡，这无法为后期开发阶段奠定坚实的基础。而在开发阶段，线性递减参数常常使得搜索单元移动速度过快，滑过局部最佳位置。

线性递减参数机制不能使蝗虫优化算法充分利用整个迭代过程。本研究引入动态权重参数机制以提高算法对迭代过程的利用率。改进的搜索过程分为三个阶段，即前期阶段，中间阶段和后期阶段。在前期阶段，迭代公式中目标位置的权重被设置的更高，以使搜索过程快速收敛。在中间阶段，控制参数是稳定的，以使算法探索搜索空间。在后期阶段，搜索单元中的目标位置权重应该相对更小，以使搜索单元深入搜索局部最优解位置。具有动态权重参数的蝗虫优化算法的迭代公式如公式5.8所示。

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

在公式5.8中，参数 m 是用来调节搜索过程的动态权重参数。为了与三个搜索阶段的特点相契合，参数 m 的取值设置如公式5.9所示。

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

5.4.2 随机跳出机制

原始的蝗虫优化算法没有使用跳出机制，所有搜索单元仅仅根据蝗虫集群内部社会关系影响和外部目标食物吸引力的影响而进行移动。原始蝗虫优化算法的这种缺点会导致算法在搜索过程中比较容易陷入局部最优位置，这也大大影响了蝗虫优化算法的搜索精度。

引入随机跳出策略可以帮助蝗虫优化算法提高跳出局部最优位置的能力。设置参数 p 为跳出阈值。在每一轮迭代结束之前，将当前最佳适应度值与上一轮迭代得到的最佳适应度值进行比较。如果当前最佳适应度值与上一轮最佳适应度值的比值高于阈值 p，则可以认为算法没有在这一轮迭代中寻找到更优的解，所以启动随机跳出机制。在最优目标位置的周围利用随机初始化的方法生成新的搜索单元。随机初始化方法如公式5.10所示。

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

在公式5.10中，*tempPos* 是新生成的搜索单元的位置，*curPos* 是当前最优目标解的位置。*iniRan* 是管理随机跳跃边界的初始化范围参数。如果 *iniRan* 设置得比较高，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 带随机跳出机制的动态权重蝗虫优化算法流程

本章提出了一种带随机跳出机制的动态权重蝗虫优化算法。这种带随机跳出机制的动态权重蝗虫优化算法的搜索过程分为初始化阶段，参数设置阶段，计算阶段和适应度值更新阶段四个阶段。在初始化阶段，设置包括位置边界在内的一些模型参数，并在边界内随机初始化最初的搜索单元集群位置。在参数设置阶段，搜索过程进入迭代循环，根据公式5.9设置动态权重参数 m。在计算阶段，根据公式5.5计算蝗虫集群内部的社会关系影响，并利用公式5.8计算搜索单元的新的位置。在适应度值更新阶段，如果当前适应度值优于历史的最佳目标适应度值，则更新目标最优解的适应度值和位置。如果当前适应度值与上一轮迭代的

适应度值之比高于之前设置的跳出阈值 p，则使用随机跳出策略生成新的搜索单元，通过公式5.10跳出局部最优位置。带随机跳出机制的动态权重蝗虫优化算法的算法流程如算法2所示。

算法 2 带随机跳出机制的动态权重蝗虫优化算法

初始化蝗虫集群，设置位置边界 u 和 l 的值
初始化参数，包括 *cmax*, *cmin*, *MaxIteration*, *iter*, *iniRan* 等
使用随机矩阵初始化所有搜索单元的位置
计算目标适应度值，记录目标位置
while ($iter < MaxIteration$ 且 $target fitness > destination fitness$) **do**
    通过公式5.9设置动态权重参数 m 的值
    通过公式5.3和公式5.4计算 d_ij 和 $\widehat{d_ij}$ 的值
    通过公式5.5计算 $s(d_ij)$ 的值
    通过公式5.8来更新搜索单元的位置 x_iter_i
    计算适应度值
    if 当前适应度值优于目标适应度值 **then**
        更新目标适应度值和目标最优解位置
    end if
    if 当前适应度值与上一轮迭代适应度值的比值大于跳出阈值 p **then**
        根据公式5.10生成新的搜索单元
        if 新的搜索单元得到的适应度值优于目标适应度值 **then**
            更新目标适应度值和目标最优解位置
        end if
    end if
end while
返回得到的目标最优值和目标最优解的位置

5.4.4 实验结果

5.4.4.1 实验设置

为了评估所提出的带随机跳出机制的动态权重蝗虫优化算法的性能，本研究进行了一系列实验，将所提出的带随机跳出机制的动态权重蝗虫优化算法 (DJ-GOA) 与原始的蝗虫优化算法 (GOA)、一种最近提出的元启发式算法蜻蜓优化算法 (DA) 和一种经典的启发式算法粒子群算法 (PSO) 进行了比较。实验中使用到

的 benchmark 测试函数为国际通用标准测试函数集 CEC benchmark[163]，文献[114] 中也使用过其中的部分测试函数，本小节选用其中的 13 个测试函数。13 个 benchmark 测试函数分为两种类型。函数 $F_1 - F_7$ 是单峰测试函数，整个测试函数只存在一个局部最优解，也即是全局最优解，单峰测试函数能够测试算法的收敛速度和局部搜索能力。函数 $F_8 - F_{13}$ 是多峰测试函数，存在多个局部最优解，多峰测试函数能够测试算法的全局搜索能力以及跳出局部最优解的能力。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。

表 5.1 F_1-F_7 单峰测试函数

Function	Dim	Range	f_{min}
$F_1(x) = \sum_{i=1}^{n} x_i^2$	6/30	[-100,100]	0
$F_2(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$	6/30	[-10,10]	0
$F_3(x) = \sum_{i=1}^{n} (\sum_{j=1}^{i} x_j)^2$	6/30	[-100,100]	0
$F_4(x) = max_i\{\lvert x_i \rvert, 1 \le i \le n\}$	6/30	[-100,100]	0
$F_5(x) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	6/30	[-30,30]	0
$F_6(x) = \sum_{i=1}^{n} ([x_i + 0.5])^2$	6/30	[-100,100]	0
$F_7(x) = \sum_{i=1}^{n} i x_i^4 + random[0,1)$	6/30	[-1.28,1.28]	0

在表格中，Dim 代表解空间的维度，在本小节中的所有测试中，测试函数均取 6。Range 代表每个维度上解取值的上下界范围。f_{min} 为预期的函数最优解的适应度值。为了解决测试对比几种算法的性能，本小节的实验使用了包含 30 个搜索单元的集群，最大迭代次数设置为 500。每个算法的每个实验重复进行 30 次，以观察和比较其统计结果。几个对比算法中，粒子群算法的参数设置参考文献 [104]，蜻蜓优化算法的参数设置参考文献 [113]，蝗虫优化算法的参数设置参考文献 [114]。而本小节所提出的带随机跳出机制的动态权重蝗虫优化算法，跳出阈值 p 取 0.95，管理随机跳跃边界的初始化范围参数 $iniRan$ 取 2。

表 5.2 F_8-F_{13} 多峰测试函数

Function	Dim	Range	f_{min}
$F_8(x)=\sum_{i=1}^{n}-x_i\sin(\sqrt{\lvert x_i\rvert})$	6/30	[-500,500]	$-418\times Dim$
$F_9(x)=\sum_{i=1}^{n}[x_i^2-10\cos(2\pi x_i)+10]$	6/30	[-5.12,5.12]	0
$F_{10}(x)=-20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2})-$ $exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i))+20+e$	6/30	[-32,32]	0
$F_{11}(x)=\frac{1}{4000}\sum_{i=1}^{n}x_i^2-\prod_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}})+1$	6/30	[-600,600]	0
$F_{12}(x)=\frac{\pi}{n}\{10\sin(\pi y_1)+\sum_{i=1}^{n-1}(y_i-1)^2[1+10\sin^2(\pi y_{i+1})]+$ $(y_n-1)^2\}+\sum_{i-1}^{n}u(x_i,10,100,4)+\sum_{i=1}^{n}u(x_i,10,100,4)$ $y_i=1+\frac{x_i+1}{4}$ $u(x_i,a,k,m)=\begin{cases}k(x_i-a)^m x_i>a\\0-a<x_i<a\\k(-x_i-a)^m x_i<-a\end{cases}$	6/30	[-50,50]	0
$F_{13}(x)=0.1\{\sin^2(3\pi x_1)+\sum_{i=1}^{n}(x_i-1)^2[1+\sin^2(3\pi x_i+1)]+$ $(x_n-1)^2[1+\sin^2(2\pi x_n)]\}+\sum_{i=1}^{n}u(x_x,5,100,4)$	6/30	[-50,50]	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，最优值和最差值，以描述 DJGOA，GOA，DA 和 PSO 的算法性能。

从表5.3中可以看出，DJGOA 的性能优于其他 3 种算法。在所有 13 个 benchmark 测试函数中，有 10 个函数的测试结果显示，DJGOA 得到的平均值比其他几种算法得到的平均值要好上几个数量级。而对于另外 3 个 benchmark 测试函数得到的结果，DJGOA 的表现略差于其他的几种算法，但仍然能够维持在同一数量级上。而且对于上面提到的这 3 个 benchmark 测试函数中的 2 个，DJGOA 可以在获得最优值方面做得更好。实验结果表明，DJGOA 在标准差和最差值方面表现也更好，这说明 DJGOA 在寻优的过程中能够表现得更稳定，更不容易找到较差的解。DJGOA 可以在 13 个测试函数中的 12 个中获得更好的最优值结果，这意味着 DJGOA 能够比其他 3 种算法更好地找到更优的解。

对于单峰测试函数，GOA 的性能比较好，而 DJGOA 可以在原来的基础上更进一步地提升 GOA 算法的性能，甚至可以将搜索结果的精度提高了几个或几十个数量级。对于多峰测试函数，DJGOA 同样可以帮助提升算法性能，特别是在寻找更精确的最优值的时候。总之，使用了动态权重参数和随机跳出机制的 DJGOA 算法可以在全局搜索和局部搜索中拥有比 GOA、DA 和 PSO 更优秀的搜索能力。

5.4.4.3 威尔科克森秩和检验

为了验证 DJGOA、GOA 和 PSO 种算法在 13 个 benchmark 测试函数上的实验结果的置信水平，本小节进行了威尔科克森秩和检验 (Wilcoxon rank-sum test)。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著的。

5.4.4.4 收敛曲线分析

DJGOA、GOA、DA 与 PSO 算法在 13 个测试函数上的迭代收敛曲线结果如图5.1所示。途中横坐标为迭代次数，纵坐标为算法在当前轮次中所能搜索到的最优适应度值，由于最优适应度值跨度较大，为了更明显地显示出算法效果，这里对纵坐标取对数。收敛曲线表明，相比另外 3 种算法，DJGOA 可以使搜索过程更快速地收敛。这也说明所提出的动态权重参数机制可以帮助算法更充分地利用每一次迭代。

表 5.3 $F_1 - F_{13}$ 测试函数实验结果

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：$F_1 - F_{13}$ 测试函数实验结果

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

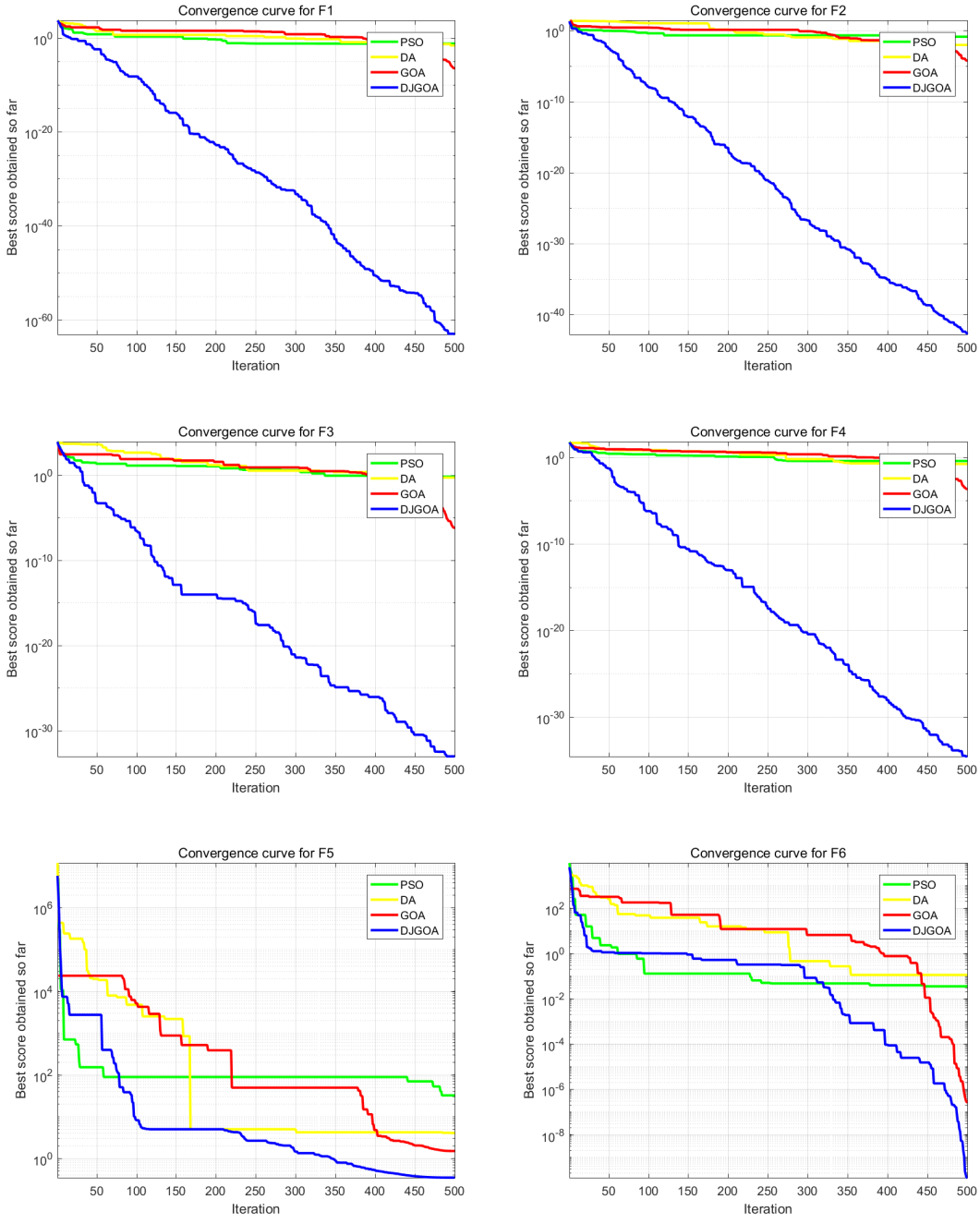

图 5.1 4 种算法在 $F_1 - F_{13}$ 测试函数上搜索的收敛曲线

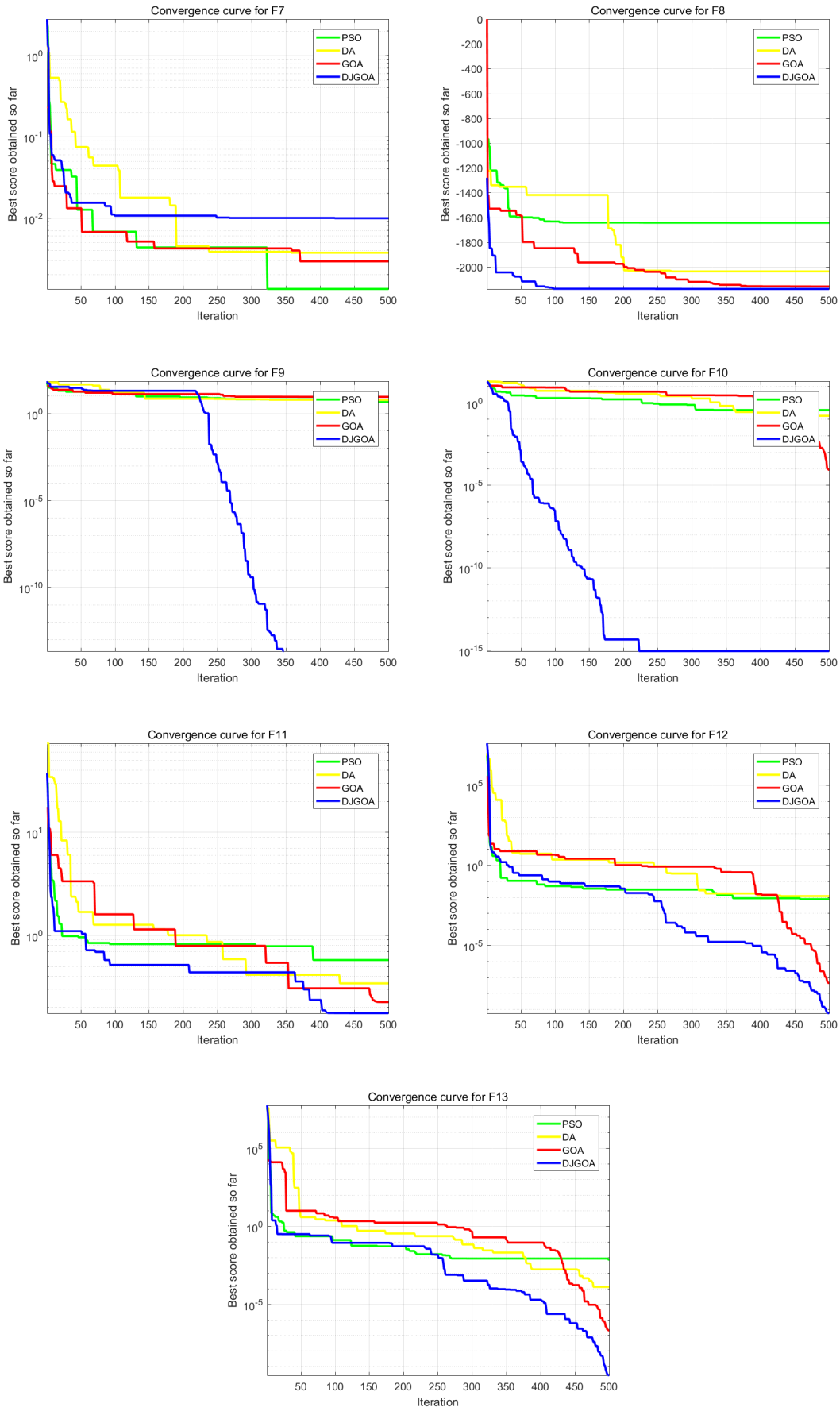

图 5.1 4 种算法在 $F_1 - F_{13}$ 测试函数上搜索的收敛曲线 (续)

表 5.4 3 种算法在测试函数 $F_1 - F_{13}$ 上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				

5.5 改进蝗虫优化算法 (IGOA)

如第5.4小节所提到的，蝗虫优化算法具有简单的理论基础，易于实施，但同时它也还有一些阻碍算法找到更优解的缺点。线性减小的舒适区控制参数不能帮助原始的蝗虫优化算法充分利用每一次迭代。而且由于缺乏随机因素，原始的蝗虫优化算法几乎没有变化，这使得算法的搜索过程很容易陷入局部最优。为了解决这些不足，本小节在第5.4小节所提出的 DJGOA 算法的基础上进一步引入了三个改进：非线性舒适区控制参数，基于 *Lévy* flight 的局部搜索机制和基于线性递减参数的随机跳出策略。本小节详细介绍了这三项改进工作的细节。

5.5.1 非线性舒适区控制参数

原始的蝗虫优化算法通过控制舒适区的半径使搜索单元一步步迭代，逐渐收敛到全局最优解附近。在搜索过程的探索阶段，舒适区控制参数应该足够大，以使搜索单元能够获得足够的搜索空间，以快速收敛到近似最优解的附近。在搜索过程中的开发阶段，舒适区控制参数应该较小，以使得搜索代理能够避免超速移动，并且能够在局部最优附近进行精确地搜索。线性递减的舒适区控制参数并不能做到使算法的搜索能力与迭代搜索期间经历的探索和开发这两个阶段协调一致。

为了使算法的搜索能力与两个搜索阶段相匹配并且增强算法在不同阶段的搜索能力，IGOA 引入了 sigmoid 函数来作为舒适区的控制参数。sigmoid 函数是

常用的阈值函数和非线性调整因子。它被广泛应用于信息科学领域。sigmoid 函数的表达式如公式5.11所示。

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

基于 sigmoid 函数变形的非线性舒适区控制参数表达式如公式5.12所示。

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

在公式5.12中，参数 w 是调节参数，取值范围为 [0,1]。参数 cx 定义如公式5.13所示。

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

在公式5.13中，参数 v 是精度调节参数，取值范围为 [1,10]。结合了基于 sigmoid 函数变形的非线性舒适区控制参数的蝗虫优化算法迭代公式如公式5.14所示。

$$X_i = m \times S_i + \widehat{T_d} \tag{5.14}$$

5.5.2 基于 *Lévy* 飞行的局部搜索机制

原始蝗虫优化算法中的所有参数都是确定性的，非常缺乏随机性。这会导致算法在搜索迭代演进的过程中缺乏创造性，每个搜索单元只能搜索相对确定的位置。将随机因子引入到确定性系统中是提高性能的常用的方法。

Lévy 飞行是由 Paul *Lévy* [113] 提出的一种随机搜索游走方法，是一种非常有效的提供随机因子的数学方法。由于 *Lévy* 飞行实现起来非常复杂，所以研究人员通常使用一种模拟算法来代替 *Lévy* 飞行，该方法的表达式如公式5.15所示。

$$Levy(d) = 0.01 \times \frac{r_1 \times \sigma}{|r_2|^{\frac{1}{\beta}}} \tag{5.15}$$

在公式5.15中，d 是要解决问题的维度，r_1 和 r_2 是两个取值为 [0,1] 区间的随机数，β 是一个常数，根据文献 [113] 的描述，通常取值为 1.5。σ 的计算方式如公式5.16所示。

$$\sigma = \left(\frac{\Gamma(1+\beta) \times \sin(\frac{\pi\beta}{2})}{\Gamma(\frac{1+\beta}{2}) \times 2^{(\frac{\beta-1}{2})}}\right)^{\frac{1}{\beta}} \tag{5.16}$$

在公式5.16中，$\Gamma(x) = (x-1)!$。

为了扩展搜索单元的搜索半径，增强算法的随机性以及局部最优值的搜索能力，本小节提出了一种基于 $Lévy$ 飞行的局部搜索机制。当一轮的位置更新过程结束的时候，可以以一定概率通过 $Lévy$ 飞行对每个搜索单元的位置进行调整。调整公式的定义如公式5.17所示。在某种程度上，$Lévy$ 飞行可以在搜索单元向着最优解位置搜索的过程中为其提供一定的“视觉”，使得搜索单元可以“看到”它们周围的一片较小的区域内的情况。

$$X_i = X_i + 10c \times s_{threshold} \times Levy(dim) \times X_i \tag{5.17}$$

在公式5.17中，$s_{threshold}$ 是控制飞行方向和变化概率的阈值参数。$s_{threshold}$ 的计算方法如公式5.18所示。

$$s_{threshold} = sign(x_{trans} - 1) + sign(x_{trans} + 1) \tag{5.18}$$

在公式5.18中，$sign(x)$ 是 sign 符号函数，x_{trans} 是取值在 [-3,3] 之间的随机数。

5.5.3 基于线性递减参数的随机跳出策略

蝗虫优化算法的基础理论是初级的。该算法只关注了收敛到全局最优的过程，而忽略了跳出局部最优的机制。因此，蝗虫优化算法的搜索过程很容易陷入局部最优，使得搜索无法更进一步进行下去。

为了提高蝗虫优化算法的跳出局部最优的能力，本小节提出了基于线性递减参数的随机跳出策略。当搜索单元搜索到当前最优解的位置的时候，新的目标位置可以替换旧的目标位置。如果没有搜索到最优解，则可以开始启动基于线性递减参数的随机跳出机制。该机制的方法如公式5.19所示。

$$X_i^{new} = ((0.5 - rand(0,1)) \times 2 + 1)X_i \tag{5.19}$$

在公式5.19中，X_i 是第 i 个搜索单元的位置，X_i^{new} 是第 i 个搜索单元进行随机跳出以后得到的新位置。如果 X_i^{new} 拥有更优的适应度值，那么它将会取代 X_i。这样一次成功的跳出行为就发生了。

原始蝗虫优化算法的演进公式仅仅将当前迭代获得的最佳位置作为搜索方向，但是忽略了一些其他可能有用信息。为了使新发生的成功跳出动作所能提

供的信息的影响力持续下去，IGOA 将搜索单元的位置迭代演进公式变成如公式5.20所示。

$$X_i^{iter+1} = m \times S_i + (1-p)\widehat{T}_d + p \times X_i^{iter} \tag{5.20}$$

在公式5.20中，p 是用于控制搜索单元位置影响的协调参数。p 在第一次迭代的时候初始化为 0。如果搜索单元没有进行跳出或者跳出局部最优失败，p 仍会被设置为 0，以确保只有 S_i 和 T_d 才能影响下一次迭代演进。当搜索单元完成一次成功跳出时，p 被设置为在接下来的 3 次迭代中线性递减为 0 的变量，以使成功跳出的行为的影响持续到接下来的 3 次迭代演进过程中。经过一些试验，p 的递减间隔被设置为 0.3478，并且本文并未对此取值进行详细的研究讨论。这样在本研究中，参数 p 的计算方法如公式5.21所示。

$$p = \begin{cases} p - 0.3478 & p > 0 \\ 0 & p \le 0 \\ 3 \times 0.3478 & when\ jumping\ out\ successfully \end{cases} \tag{5.21}$$

5.5.4 改进蝗虫优化算法流程

本小节中提出了一种改进的蝗虫优化算法。改进蝗虫优化算法的搜索过程分为初始化阶段，迭代演进阶段，适应度更新阶段和跳出阶段四个阶段。

在初始化阶段，设置参数，并随机初始化所有搜索单元的初始位置。在这一阶段中，还计算了最优目标的位置和相应的最优适应度值。在迭代演进阶段，搜索循环开始进行。每个搜索单元都通过公式5.20移动到新的位置。非线性舒适区参数 m 根据公式5.12进行设置。之后每个搜索单元根据公式5.17以一定的概率进行 *Lévy* 飞行调整，并生成新的搜索位置。在适应度值更新阶段，计算新位置的适应度值。如果新的适应度值优于当前的全局最优适应度值，则新的位置可以取代旧的全局最优目标的位置。如果新的适应度值没有比当前全局最优目标的适应度值更优，那么该搜索单元就会进入跳出阶段。在此阶段，搜索单元尝试根据公式5.19跳出局部最优，并计算新的适应度值。如果新的适应度值优于当前搜索单元本身的适应度值，那么新的位置可以取代旧的搜索单元的位置，完成一次成功的跳出。此时参数 p 也由根据公式5.21进行更新。如果新的适应度值没有优于当前搜索单元本身的适应度值，那么搜索单元不会跳到新的位置上，会仍然停留在旧的位置上。到目前为止，循环求解过程中的一次迭代已经完成。在达到最大迭代次数之后，循环过程结束，此时的全局最优适应度值和全局最优目标位置就

是最终的寻优求解的结果。改进蝗虫优化算法的伪代码算法图如算法3所示。改进蝗虫优化算法的算法流程框图如图5.2所示。

算法 3 改进蝗虫优化算法

初始化参数
通过随机矩阵初始化蝗虫集群的初始位置
计算初始目标适应度值，记录目标位置
while $(iter < MaxIteration$ 且 $targetfitness > destinationfitness)$ **do**
    根据公式5.12设置非线性舒适区控制参数 m
    根据公式5.20更新 x_i 的位置
    x_i 搜索单元根据公式5.17进行 $Lévy$ 飞行
    计算适应度值
    if 当前适应度值优于目标适应度值 **then**
        更新目标适应度值和目标最优解位置
    else
        x_i 搜索单元根据公式5.19进行跳出
        计算适应度值
        if 当前适应度值优于搜索单元自身适应度值 **then**
            更新搜索单元的位置
        end if
        根据公式5.21设置参数 p 的值
    end if
end while
返回得到的目标最优值和目标最优解的位置

5.5.5 实验结果

5.5.5.1 实验设置

为了评估所提出的改进蝗虫优化算法 (IGOA) 的性能，本小节进行了一系列实验。在本小节的工作中，IGOA 算法与 6 个元启发式算法的性能进行了比较，包括原始的蝗虫优化算法 (GOA)，基于反向学习的蝗虫优化算法 (OBLGOA)，最近几年新提出的三种元启发式算法，鲸鱼优化算法（WOA），蜻蜓优化算法（DA）和蚁狮优化算法（ALO），以及经典的启发式算法，粒子群优化算法（PSO）。在本小节的实验中，其他 6 种算法的参数设置如文献 [104, 111–114, 161] 所示。

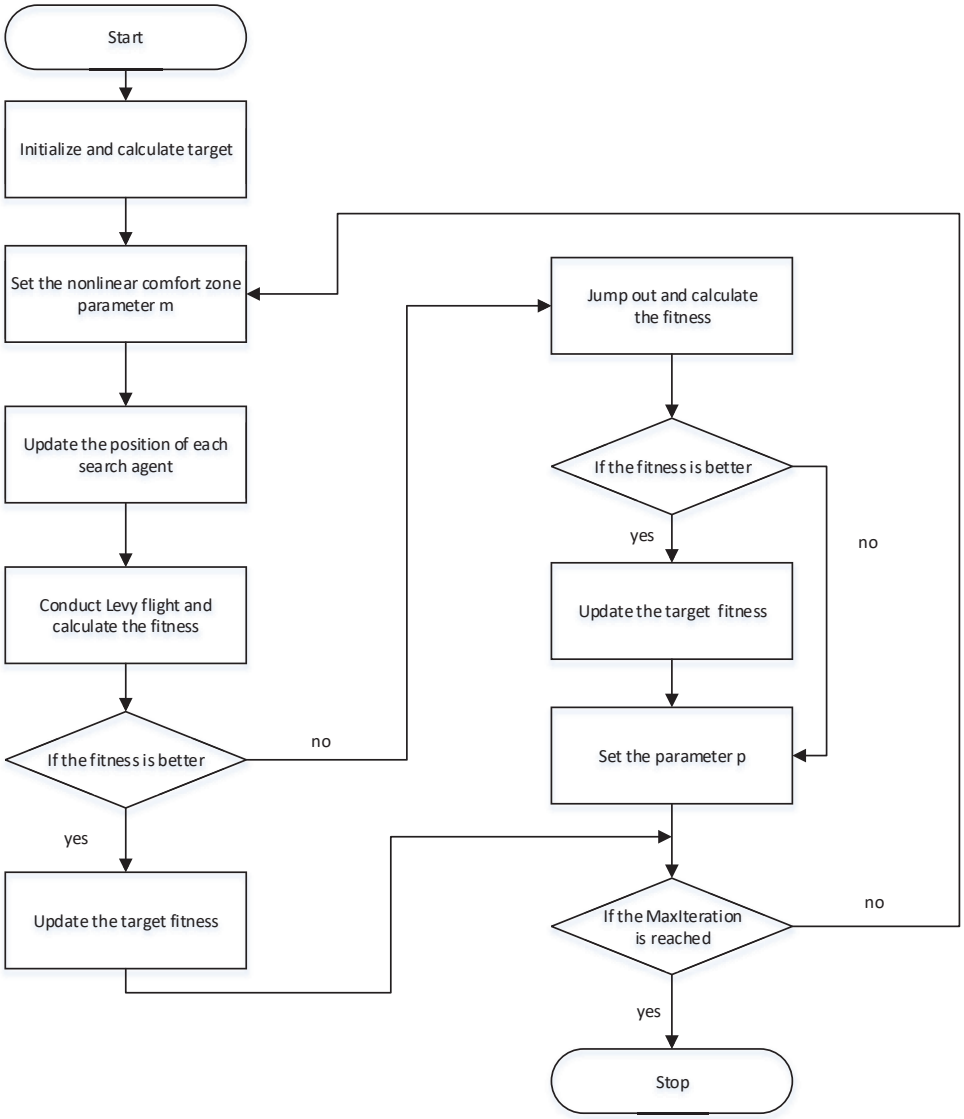

图 5.2 改进蝗虫优化算法流程框图

本小节的实验使用了 29 个著名的 benchmark 测试函数来测试所提出的改进蝗虫优化算法的搜索性能。benchmark 测试函数可以分为 3 种类型，用来评估算法的不同方面的性能。在表5.1中列出的测试函数 $F_1 - F_7$ 是单峰测试函数，只有一个局部最优位置，也即是全局最优位置，可以用来评价算法的局部搜索能力。在表5.2和表5.5中列出的 benchmark 测试函数 $F_8 - F_{23}$ 是具有多个局部最优的多峰测试函数，可以用来评价算法的全局搜索能力 [114]。在表5.6中所列出的测试函数 $F_{24} - F_{29}$ 是复合测试函数 [164]，复合测试函数在一种特定的框架内组合了一些基本的测试函数，得到更加复杂的测试函数。复合测试函数可以用来评价算法摆脱局部最优的能力。在表5.1、表5.2、表5.5和表5.6中，*Dim* 表示测试函数的维度，*Range* 是优化问题的搜索边界，f_{min} 是测试函数的预期最优适应度值。在本小节的实验中，表5.1和表5.2中所列的测试函数 $F_1 - F_{13}$ 的维度均为 30。

使用 Matlab 代码在上述 29 个 benchmark 测试函数上完成了一系列相关实验。对于测试函数 $F_1 - F_{23}$，每个算法使用 30 个搜索单元搜索，每次实验进行 500 次迭代搜索。对于测试函数 $F_{24} - F_{29}$，每次实验的搜索过程进行 100 次迭代。每次搜索实验对每个算法重复 30 次以降低意外情况的影响。通过计算一些 30 次实验的统计数据，例如平均值 (avg)，标准差 (std)，30 次重复实验中得到的最优适应度值 (best) 和 30 次重复实验中得到的最差适应度值（worst），比较算法的性能。另外，还进行了威尔科克森秩和检验，计算了 *p-value* 以证明实验结果的统计显著性。

5.5.5.2 实验结果

函数 $F_1 - F_7$ 是单峰函数，只有一个局部最优解，也即是全局最优解，可以用来测试算法的局部搜索能力。如果搜索算法具有超强的局部开发能力，它可以进行更精确的搜索，并找到更接近全局最优的解。函数 $F_1 - F_7$ 的实验结果显示在表5.7中。

从表5.7中可以看出，IGOA 可以在函数 $F_5 - F_7$ 中获得最佳的平均值结果，并且在函数 $F_1 - F_4$ 中，IGOA 的表现也仅比 WOA 差。关于标准差和最差值方面，IGOA 也可以比函数 $F_3 - F_7$ 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更差解的可能性并且提高算法的稳定性。与 GOA 和 OBLGOA 相比，所提出的 IGOA 可以显著提高原蝗虫优化算法的局部开发搜索能力。

函数 $F_8 - F_{23}$ 是具有多个局部最优解的多峰测试函数，可以测试算法的全局探索能力。如果搜索算法在搜索的过程中不能完成得很好，那么在处理多峰测试

表 5.5 F_{14}-F_{23} 多峰测试函数

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 F_{24}-F_{29} 复合测试函数

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 $F_1 - F_7$ 单峰测试函数实验结果

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**

函数时，搜索很可能会陷入局部最优，即使拥有更好的局部开发能力也不能帮助算法取得更好的结果。错误的努力方向可能会导致错误的结果。函数 $F_8 - F_{23}$ 的实验结果如表5.8和表5.9所示。

表 5.8 $F_8 - F_{13}$ 多峰测试函数实验结果

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中可以看出，所提出的 IGOA 可以在 16 个测试函数中的 11 个中获得最优的平均适应度值，并且在测试函数 F_{10}、F_{20} 和 F_{23} 中 IGOA 可以获得次优的结果。关于标准差的实验结果表明，IGOA 的稳定性可能不如在平均适应度值方面所表现的那么好，但在大多数测试中 IGOA 仍然是所有 7 种算法中最好的。关于最佳适应度值和最差适应度值的实验结果可以表明，所提出的 IGOA 具有在几乎所有测试函数的实验中寻找到最优解的可靠能力，并且 IGOA 找到最差解的概率也是最小的。与原始 GOA 以及 OBLGOA 相比，所提出的 IGOA

表 5.9 $F_{14}-F_{23}$ 多峰测试函数实验结果

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 $F_{14}-F_{23}$ 多峰测试函数实验结果（续）

function	type	IGOA	GOA	WOA	DA	ALO	PSO	OBLGOA
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

可以极大提高算法的搜索性能。

函数 $F_{24}-F_{29}$ 是复合测试函数，复合测试函数在一种特定的框架内组合了一些基本的测试函数，来构建新的、复杂的、可控的测试函数。复合测试函数比多峰测试函数更加复杂，也更具挑战性，在评估算法的搜索能力时更有说服力。复合测试函数的实验结果如表5.10所示：

根据表5.10中列出的 $F_{24}-F_{29}$ 复合测试函数实验结果，可以看出，与其他 6 种算法相比，所提出的 IGOA 非常有竞争力。在测试函数 F_{24} 和 F_{26} 中，IGOA 可以获得最优的平均适应度值，在测试函数 F_{25}、F_{27} 和 F_{29} 中 IGOA 可以获得次优的平均适应度值。在标准差方面，IGOA 表现得不是最好的，也不是最差的。IGOA 在最优值和最差值方面，分别在 4 个函数和 3 个函数中表现最佳。尽管 IGOA 在标准差方面表现得一般，但它在寻找更优解、避免更差解以及控制风险方面可以胜过其他算法。与原始 GOA 和 OBLGOA 相比，IGOA 的性能能够提升很多。根据复合函数的测试的结果，可以证明所提出的 IGOA 算法具有处理这种复杂而具有挑战性的问题的能力。

表 5.10 $F_{24}-F_{29}$ 复合测试函数实验结果

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

5.5.5.3 威尔科克森秩和检验

基于 30 次独立实验的平均值和标准差没有比较每次实验之间的差异。所得到的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显著性，本小节引入了威尔科克森秩和检验。威尔科克森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群体。本小节计算了关于 IGOA 与每个其他算法之间在测试函数 $F_1 - F_{29}$ 上的统计数据的 *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著的。IGOA 的威尔科克森秩和检验结果如表5.11所示。

从表5.11中可以看出，在大多数的测试中，IGOA 与另外的比较算法之间的 *p-values* 小于 0.05，实验所得到的结论是显著的。在某些测试函数中，某些 *p-values* 超过了 0.05，但这些函数中不同的算法会获得相同的最优解，例如测试函数 F_{14}、F_{19}、F_{20}、F_{21}、F_{22} 和 F_{23}。通过分析复合函数的实验结果，在测试函数 F_{25} 和 F_{28} 中，*p-values* 在 IGOA 和 GOA 之间超过 0.05，但是其中 GOA 能够找到最佳适应度值而 IGOA 却没有。综合考虑，仍然可以证明所提出的 IGOA 可以显著提高算法的性能。

5.5.5.4 算法测试收敛速度

本小节讨论了算法的收敛速度。IGOA 和所有其他 6 种算法在部分测试函数上搜索的收敛曲线图如图5.3所示。

水平轴是当前迭代的次数，垂直轴是迄今为止所搜索到的最佳适应度值。为了使收敛曲线的表现力更明显，在垂直轴上使用对数。图5.3中的函数收敛曲线表明，相对于整个搜索过程，在大多数搜索过程中 IGOA 都可以保持较大的斜率。这种现象意味着所提出的非线性舒适区控制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分的，这也表明基于 *Lévy* 飞行的局部搜索机制使算法更具创造性。在测试函数 F_3、F_4、F_7 和 F_{28} 中出现的突然下降，表明随机跳出策略可以帮助搜索跳出局部最优。算法在测试函数上的收敛曲线图表明，所提出的 IGOA 可以使搜索过程比其他搜索算法更快地收敛。

表 5.11 7 种算法在测试函数 $F_1 - F_{29}$ 上的威尔科克森秩和检验结果

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 7 种算法在测试函数 $F_1 - F_{29}$ 上的威尔科克森秩和检验结果（续）

function	GOA	WOA	DA	ALO	PSO	OBLGOA
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

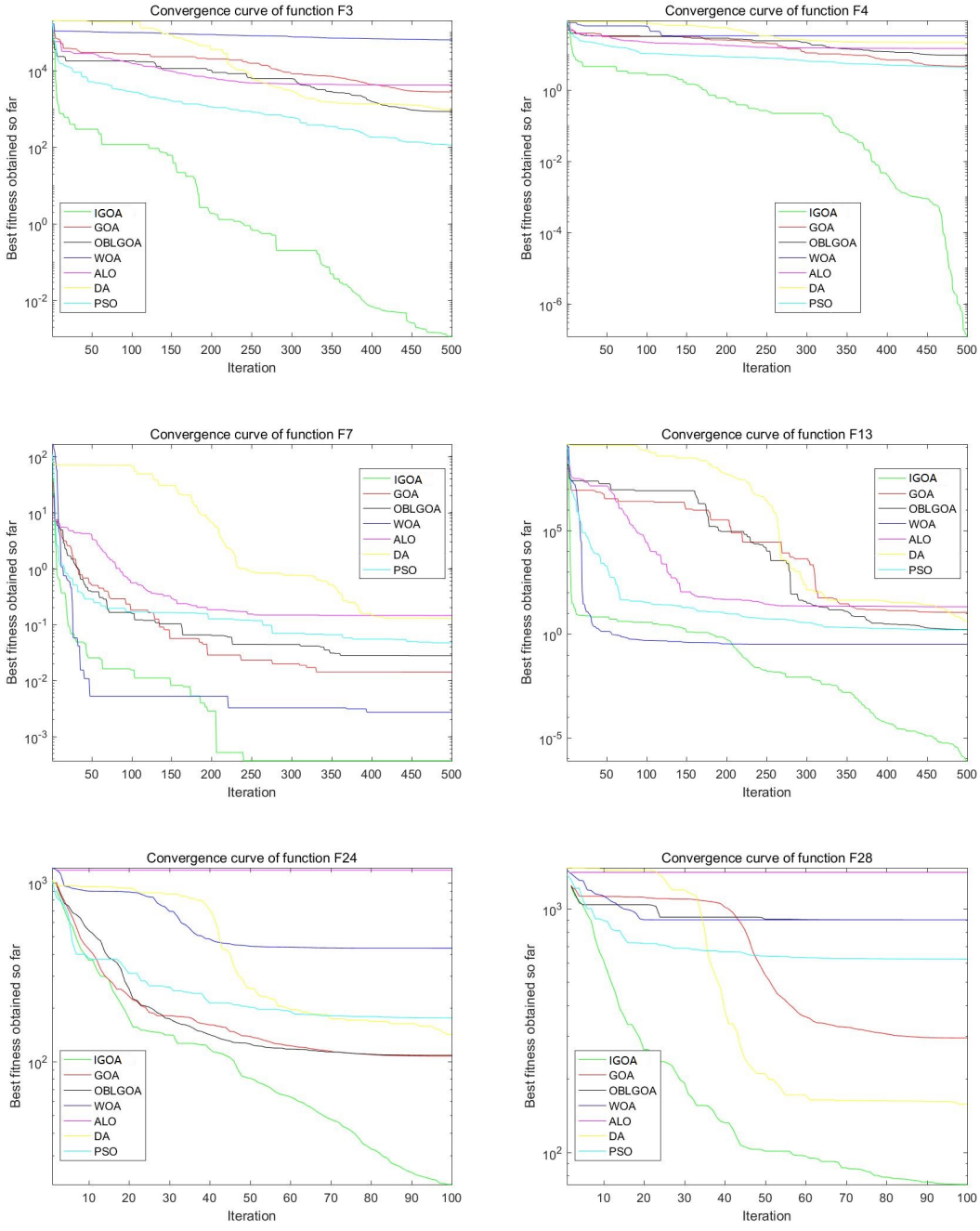

图 5.3 7 种算法在部分测试函数上搜索的收敛曲线图

5.6 多终端协同服务的任务调度问题

5.6.1 问题描述

任务调度问题是一种在云计算 [91]、边缘计算 [92]、海服务 [14] 等领域常见的问题。大量的用户会产生大量的任务。因此，当大量用户同时访问终端服务时，系统应该能够高效地处理这些任务，这是终端服务系统的一个重要问题，尤其是在终端资源有限的情况下。在任务调度问题中，一系列任务会到达任务调度中心，等待分配给特定的执行节点。

由于资源情况、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是异构的。不同的任务调度方案会产生不同的执行结果，一个优秀的任务调度解决方案可能会对整个系统服务效果产生很大影响。对于终端服务的提供商来说，将任务分配给合适的节点可以节省能源并且减少预算。而对于终端服务的使用者来所，将任务调度到高效节点可以减少等待时间并提高用户体验。

本小节将所提出的 IGOA 算法应用于多终端协同服务的任务调度问题。将 N 个任务调度到 M 个节点的一种解决方案抽象表示为搜索空间中的一个位置，每一种解决方案都是一个 N 维离散向量。每个解决方案可以对应一个系统的完工时间和预算费用，并且可以通过完工时间和预算费用来计算适应度值。搜索单元集群通过 IGOA 算法的搜索策略在整个搜索空间中搜索最佳适应度值的位置，能够得到最优或近似最优的任务调度方案。

5.6.2 任务调度模型

在本小节的研究中，假设用户启动 N 个任务，这些任务分别为 $T_1, T_2, ..., T_N$，终端服务系统有 M 个节点可以用来处理任务，这些节点分别为 $\{S_1, S_2, ..., S_M\}$。任务调度模型描述如下。

1. N 个任务的集合为 $\{T_1, T_2, ...T_N\}$，T_i 代表第 i 个任务所消耗的资源

2. M 个节点的集合为 $\{S_1, S_2, ...S_M\}$，S_j 代表第 j 个节点所能提供的最大资源

3. 只要某个节点能够提供某个任务执行所需要的足够资源，也就是 $T_i \leq S_j$，该任务就可以在该节点上执行，

4. 每个节点在同一时间只能执行一个任务，多个任务可以在同一个节点上排队执行

5. 所有任务被分为 K 种类型。表示任务类型的向量为 $tot[N]$，tot_i 代表第 i 个任务的类型，取值为 [1, K] 中的整数

6. 每个节点处理不同类型任务的能力不同。$mips[M, K]$ 是执行速度矩阵

$mips_j^k$ 代表类型 k 的任务在第 j 个节点上执行时单位时间内消耗的资源数量。et_{ij} 代表第 i 个任务在第 j 个节点上执行所消耗的时间，计算方式如公式5.22所示。

$$et_{ij} = \begin{cases} \frac{T_i}{mips_j^{tot_i}} & task\ i\ runs\ on\ node\ j \\ 0 & task\ i\ does\ not\ runs\ on\ node\ j \end{cases} \tag{5.22}$$

7. 第 j 个节点的执行时间 st_j 是通过将在第 j 个节点上执行的所有任务的执行时间相加得到的。计算方法如公式5.23所示。

$$st_j = \sum_{i=1}^{N} et_{ij} \tag{5.23}$$

总的执行时间 *Makespan* 所有节点的执行时间取最长值。计算方法如公式5.24所示。

$$Makespan = \max\{st_j | j = 1, 2, 3...M\} \tag{5.24}$$

8. 工作节点在执行任务的时候会产生额外的费用开销。费用只与工作节点的工作时间有关，不论工作节点上执行的是什么类型的任务。节点的费用向量为 $bps[M]$，代表着第 j 个节点在单位时间内产生的额外费用开销。

9. 总的费用 *Budget* 是通过将所有节点产生的费用相加得到的。*Budget* 的计算方法如公式5.24所示。

$$Budget = \sum_{j=1}^{M} (st_j \times bps_j) \tag{5.25}$$

10. 时间和费用是从两个方面来描述任务调度问题的开销情况的，因此需要一个统一的评估适应度。在这里，任务调度问题的评估适应度被定义为时间和费用通过权重参数 α 和 β 进行结合的结果。评估适应度 *fitness* 的计算如公式5.26所示。

$$fitness = \alpha Makespan + \beta Budget \tag{5.26}$$

11. 搜索过程是连续的，但问题的解决方案是离散的。所以当每个循环迭代的过程结束后，所有的搜索单元都应该向距离其最近的整数位置进行调整。

12. 同一个节点上执行的两个任务之间的切换时间被忽略。边缘节点之间的传输时间也被忽略不计。

5.6.3 实验结果

在此任务调度模块中，N 个任务被分为 K 种类型，M 个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，这里不再进行讨论。在本小节的工作中，不失一般性地，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。上述参数设定如公式5.27所示。

$$K = 4;\ N = 10;\ M = 5;\ \alpha = 0.7;\ \beta = 0.3; \tag{5.27}$$

向量 *T*, *tot*, *S*, *bps* 以及矩阵 *mips* 的设置如表5.12所示。

表 5.12 任务类型及任务消耗资源情况

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 节点拥有资源及 bps

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 任务和节点的 mips

$mips_j^k$	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

在本小节的工作中，所提出的IGOA与另外6个算法进行了比较，包括GOA、WOA、ALO、DA、PSO和OBLGOA。每种算法使用30个搜索单元进行搜索。为了减少意外因素的影响，对每种算法进行了30次独立测试，计算并比较了测试结果的平均值，标准差，最优值和最差值等统计数据。实验结果如表5.15所示。

表5.15 任务调度实验结果

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%

从表5.15中可以看出，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。IGOA拥有最小的标准差，说明该算法相比其他算法更加稳定。IGOA可以找到所有算法中的最优解，并且能够控制找到最差解的值。与其他6种算法相比，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

为了在终端资源有限且终端资源异构性强的情况下，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，本章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。该算法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，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，所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。该算法引入变形的 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，提出基于 *Lévy* 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，该算法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，实验结果表明所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。最后提出了多终端协同服务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，所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。

本章所涉及的研究成果包括：

论文 2 篇：“A Dynamic Weight Grasshopper Optimization Algorithm with Random Jumping”（3rd International Conference on Computer, Communication and Computational Sciences (IC4S2018)，EI 会议，已录用）。

“An Improved Grasshopper Optimization Algorithm for Task Scheduling Problems”（International Journal of Innovative Computing, Information and Control (IJICIC)，EI 期刊，已录用）。

第 6 章　基于预测的容器弹性服务策略

6.1　引言

在本文前面的章节中，设计了基于容器化的多终端协同服务系统，将位于用户终端边缘的多个智能终端设备上的空余资源整合起来，以容器的形式为用户终端提供服务，以提高终端资源的利用率，提高终端用户的体验。但是在智能终端上以容器的形式对外提供服务的时候，会出现一组矛盾的问题。如果采用预部署的形式，即在边缘终端上提前部署好提供服务的容器，那么收到用户的服务请求就可以马上对外提供服务。但由于智能终端的资源是有限的，终端服务不能像云端系统服务那样一直在后台运行，等待着用户请求流量的到达。这种预部署的方式使得提供服务的容器也会消耗智能终端上面的资源，在用户请求流量较低的时候会造成一种智能终端资源的浪费。如果采用非预部署的形式，即不提前部署容器，而是等到用户请求到达服务终端以后再临时创建容器提供服务，在服务完成后可以看情况销毁容器，这样就可以减小很多额外开销，节约智能终端的资源。但是这种非预部署的形式会大大增加用户的平均等待时间，因为相比一次服务的计算和网络传输开销，一次容器创建的时间还是非常长的，对于等待服务请求响应的用户来说完全不能够接受。

为了解决多终端协同服务技术中的预部署问题，促进提高终端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，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，动态调整多终端协同服务的规模，平衡提高终端资源利用率与降低用户服务请求响应等待时间之间的关系。本章提出的基于预测的容器弹性服务策略，具体实现为第3章中所提出的系统架构设计中的弹性服务模块。

本章的内容组织结构如下：第6.2节介绍了预测算法的相关研究工作，包括基于统计学模型的预测算法和基于卡尔曼滤波的预测算法；第6.3节介绍了多终端协同服务中的用户流量预测模型，以及卡尔曼滤波预测算法，并结合多终端协同服务技术场景的特点，提出一种改进的卡尔曼滤波预测算法；第6.4节介绍了基于预测的容器弹性服务策略的设计思路，对具体弹性服务策略的流程进行了设计；第6.5节列出了仿真实验的结果及分析；第6.6节总结了本章内容。

6.2 相关工作

6.2.1 早期网络流量预测算法

网络流量的可预测性在许多领域都具有重要意义，包括自适应应用，拥塞控制，准入控制和网络带宽分配等。流量预测需要准确的流量模型，可以捕获实际流量的统计特征 [165]。

早期的网络流量预测算法研究主要是基于线性预测模型的短相关模型，典型的如基于泊松模型、马尔可夫过程及增量高斯混合模型，以及在这些统计学模型的基础上加以改进的一些线性预测模型，例如 IPP 模型、MMPP 模型、MMFM 模型等等 [166]。但是由于真实网络环境中的网络流量数据拥有长相关、自相似性、突发性等特点，使用传统的泊松模型、马尔可夫模型等随机模型不能够很好地对网络流量数据进行模拟和预测 [167]。

自回归滑动平均模型 (Auto-Regressive and Moving Average Model，即 ARMA 模型) 是一种典型的传统线性预测模型，可以利用时间序列分析对过去的数据进行回归分析，通过设置多项式参数，对历史测量值进行拟合，可以对未来短时间内的流量值进行预测 [167–169]。但是基于 ARMA 模型预测算法仍然不能有效刻画网络流量性质，在对网络流量进行分析的时候可能与实际情况差异较大 [166]。

灰色模型 (Grey Models) 是邓聚龙教授于 1982 年提出的一种理论，只需要有限的数据量就可以对未知系统的行为进行估计，自提出以来便被成功应用到很多领域中的系统问题 [170]。灰色模型是一种典型的非线性预测模型，计算量较小，预测精度较高，但是预测效果不够稳定，当系统稍微发生变化的时候，灰色模型的预测结果就可能出现较大的误差 [168]。

6.2.2 基于卡尔曼滤波器的预测算法

1960 年，数学家卡尔曼提出了卡尔曼滤波器（Kalman Filter）[171]。卡尔曼滤波器是一种最优化自回归数据处理算法，这种方法通过对预测值与真实测量值进行方差加权来获得新的预测值，通过迭代不断进行预估与校正 [172]。对于一个确定性未知的动态系统来说，卡尔曼滤波器可以在考虑噪声信息干扰的情况下，对于系统的过去、现在以及未来的状态做出较为准确的估计 [173]。由于效率非常高，卡尔曼滤波器被应用于包括运动轨迹预测 [174]、电网负荷预测 [175]、气象数据融合 [176] 等多个领域。文献 [177] 中针对微服务架构，提出了一种利用模糊自适应卡尔曼滤波算法来对服务响应时间进行预测。文献 [178] 针对智慧协同网络（SINET），利用卡尔曼滤波算法对网络流量进行预测。

卡尔曼滤波器的预测模型非常简单，只需要保存当前状态的一些统计信息，即可对系统的下一个状态进行预测，具有容易实现的优点。而且卡尔曼滤波器计算简单，运行速度快，占用计算资源和存储资源少，相比前面提到的算法，与本文提出的基于容器化的多终端协同服务技术的应用场景更加适合。

6.3 基于卡尔曼滤波的预测算法

6.3.1 卡尔曼滤波器

卡尔曼滤波器的表达式是一组数学方程，能够提供最小二乘法的有效递归解 [179]。为了利用卡尔曼滤波器来解决本章中所提出的容器弹性服务问题，需要将该问题用数学表达式来描述。在本研究所提出的基于容器化的多终端协同服务技术中，通过对用户的服务请求流量的趋势进行预测，动态调整容器服务集群规模，以达到提供容器弹性服务的目的。该系统可以用表达式6.1来描述，系统中所有变量均为一维变量。

$$X_k = AX_{k-1} + BU_k + W_k \tag{6.1}$$

在公式6.1中，X_k 为系统在 k 时刻接收到的用户服务请求数量，U_k 为 k 时刻系统的控制变量，W_k 为系统过程噪声，假设为高斯白噪声（White Gaussian Noise），协方差为 Q。A 和 B 为系统参数。

而对于系统的测量模型，表达式如公式6.2所示。

$$Z_k = HX_k + V_k \tag{6.2}$$

在公式6.2中，Z_k 是系统 k 时刻的测量值，V_k 为系统的测量噪声，同样假设为高斯白噪声，协方差为 R。H 为测量系统的参数。

卡尔曼滤波器可以分为预估和校正两个阶段 [172]。在预估阶段中，系统根据时间更新方程及上一时刻系统的状态，对当前时刻的状态进行先验预估，并计算先验估计的协方差。而在校正阶段，系统通过测量的方法获得系统当前状态的观测值，通过状态更新方程计算出卡尔曼滤波增益，并对预估阶段中得到的先验估计值进行校正，得到系统当前时刻状态的后验估计值。当前时刻状态的后验估计值在下一时刻又可以根据时间更新方程计算下一时刻的系统状态先验估计。如此循环迭代，即可对系统的状态进行预测和跟踪。

离散卡尔曼滤波的时间更新方程如公式6.3和公式6.4所示。

$$X_k = AX_{k-1} + BU_k \tag{6.3}$$

$$P_k = A^2 P_{k-1} + Q \tag{6.4}$$

在公式6.4中的 P_k 为先验估计的方差。离散卡尔曼滤波的状态更新方程如公式6.5、公式6.6和公式6.7所示。

$$K_k = HP_k(H^2 P_k + R)^{-1} \tag{6.5}$$

$$\widehat{X_k} = X_k + K_k(Z_k - HX_k) \tag{6.6}$$

$$\widehat{P_k} = (1 - K_k H)P_k \tag{6.7}$$

在公式6.6中的 $\widehat{X_k}$ 为对 k 时刻系统状态的后验估计。在公式6.7中的 $\widehat{P_k}$ 为对 k 时刻系统误差的后验估计。

6.3.2 基于终端服务的改进卡尔曼滤波算法

在本章的研究中，利用卡尔曼滤波器的方法，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。基于卡尔曼滤波的预测方法计算量相对比较小，适合在基于容器化的多智能终端协同服务技术的场景中应用。但是由于终端服务的应用场景对于网络波动比较敏感，要求预测算法能够对于增加的用户服务请求做出迅速反应，而对于减少的用户服务请求可以不做特殊处理。

为了能够对于用户服务请求流量的突然增加做出迅速反应，综合考虑基于终端服务的特点，对基于卡尔曼滤波器的预测方法进行一定优化，在卡尔曼滤波器的状态更新方程公式6.6中增加调节因子。改进后的卡尔曼滤波器的状态更新方程如公式6.8所示。

$$\widehat{X_k} = X_k + K_k(Z_k - HX_k) + m * [sign(Z_k - Z_{k-1}) + 1] \tag{6.8}$$

公式6.8中，参数 m 为调节因子增益参数，用于控制增益强度，当 m 较大的时候，改进的卡尔曼滤波器会对增加的用户流量请求给出一个较高的预测值，如

果 m 过大，会影响改进的卡尔曼滤波器的预测准确程度。函数 $sign()$ 为符号函数，帮助判断新到的用户请求数量是否相较上一个采样时间增加。如果新到的用户请求数量相比上一个采样时间并没有增加，则该调节因子取值为 0，不会对原有的卡尔曼滤波器的预测结果产生影响。

6.4 基于预测的容器弹性服务策略设计

本章的研究利用第6.3小节提出的改进卡尔曼滤波算法对终端用户服务请求数量进行短期的预测，并根据预测结果，利用本小节提出的容器弹性服务部署策略对下一时刻的容器服务的规模进行调整，以达到提高终端资源利用率，提高用户体验的目的。基于预测的容器弹性服务模块图如图6.1所示。

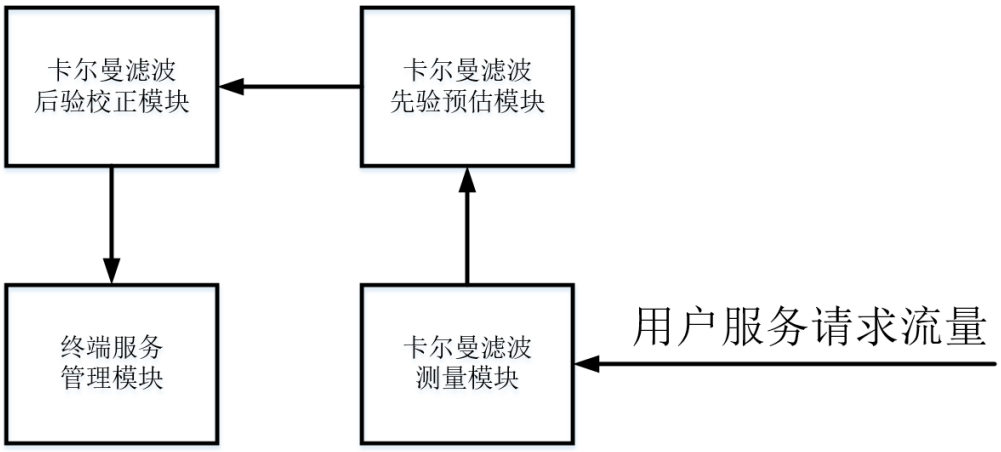

图 6.1 基于预测的容器弹性服务模块图

针对基于容器化多终端协同服务场景中的容器弹性服务问题，可以建立如下模型：

- 用户服务请求数量预测算法的采样间隔为 T
- 目标服务的单个任务执行时间为 t_x
- 每个容器的最大并发执行任务数量为 K
- t 时刻正在执行中的任务数量为 m_t
- t 时刻的终端服务容器数量为 M_t
- t 时刻新收到的用户服务请求数量为 n_t

在基于容器化多终端协同服务场景中，单一终端计算资源比较有限，虽然有一定并发计算能力，但相比云端服务的并发计算能力仍然差了很多。为了在保证

终端服务用户体验的情况下能够对终端资源更加合理地进行利用，本小节对于多终端协同服务的容器部署策略提出如下需求：

- 在保证终端服务用户体验的前提下，尽量减少容器服务规模的变动。
- 当存在容器没有执行任何计算任务的时候，关闭该容器并回收资源。
- 新的计算任务优先分配给空的容器执行，尽量减少并发。
- 当待执行计算任务总数量超过所有容器最大并发能力时，扩展容器服务规模，增加新的可用容器。

综合上述问题模型与需求，本小节提出基于改进卡尔曼滤波预测方法的容器弹性服务的部署策略如算法4所示。

算法 4 基于预测的容器弹性服务部署策略

开始
设置问题参数如 T, t_x, K 等
while 系统正常运行 **do**
    对 t 时刻用户服务请求数量 n_t 进行采样测量
    根据 n_t 分配任务执行容器，计算 t 时刻的执行中任务数量 m_t
    预测 $t+1$ 时刻的执行中任务数量为 $m_{t+1} = (m_t + n_t) * \frac{T}{t_x}$
    根据 t 时刻测量值及相关参数，利用改进的卡尔曼滤波对 $t+1$ 时刻用户服务请求数量 n_{t+1} 进行预测
    if $\frac{m_{t+1}+n_{t+1}}{M_t} > K$ **then**
        调整 $t+1$ 时刻终端服务容器数量为 $M_{t+1} = \frac{m_{t+1}+n_{t+1}}{K}$
    else
        if $\frac{m_{t+1}+n_{t+1}}{M_t} < 1$ **then**
            调整 $t+1$ 时刻终端服务容器数量为 $M_{t+1} = m_{t+1} + n_{t+1}$
        else
            维持 $t+1$ 时刻终端服务容器数量不变 $M_{t+1} = M_t$
        end if
    end if
    进入 $t+1$ 时刻
end while

如算法4所示，在系统开始运行的时候，设置好系统参数，如采样间隔 T，任务预计执行时间 t_x，每个容器允许的最大并发计算数量 K 等等。在 t 时刻对用户服务请求数量 n_t 进行采样测量，并且根据相关参数，利用改进的卡尔曼滤波预

测算法对 $t+1$ 时刻的用户服务请求数量 n_{t+1} 进行预测。另外还需要对 t 时刻所收到的实际用户服务请求分配到 M_t 个终端服务容器中进行执行，计算 t 时刻执行中的任务数量 m_t，并且根据 m_t 对 $t+1$ 时刻新的用户流量请求到达前还在执行中的任务数量 m_{t+1} 进行预测。本小节中提出的容器弹性服务部署策略根据采样周期 T 和任务预计执行时间 t_x 来进行预测，预测公式如公式6.9所示。

$$m_{t+1} = (m_t + n_t) * \frac{T}{t_x} \tag{6.9}$$

当 $\frac{m_{t+1}+n_{t+1}}{M_t} > K$ 时，算法预计 $t+1$ 时刻新到的用户服务请求数量与正在执行中的任务数量之和将会超过当前所有服务容器所能承受的最大并发数量，所以需要提前将容器服务规模扩展到 $M_{t+1} = \frac{m_{t+1}+n_{t+1}}{K}$。当 $\frac{m_{t+1}+n_{t+1}}{M_t} < 1$ 时，算法预计 $t+1$ 时刻新到的用户服务请求数量与正在执行中的任务数量之和将会少于当前所有服务容器的数量，也就是预计 $t+1$ 时刻将会出现没有任务可以执行的空闲容器，所以需要在下一时刻到来之前将容器服务规模缩减到 $M_{t+1} = m_{t+1} + n_{t+1}$。如果以上两种情况都未发生，即算法预计 $t+1$ 时刻新到的用户服务请求数量与正在执行中的任务数量之和将会正好处于当前服务容器所能够承受的范围内，为了减少并发以及减少容器服务规模的变动次数，算法选择维持服务容器的数量不变。

6.5 实验结果

使用卡尔曼滤波器对用户服务请求数量进行滤波和预测，并同 ARMA 预测模型进行对比，仿真结果如图6.2所示。

从图6.2中可以看出，相比传统的 ARMA 模型，使用卡尔曼滤波器对用户服务请求数量的预测更加接近真实值，对于进一步通过用户流量请求数量弹性调整容器服务规模更加有效。而相比原始的卡尔曼滤波器，改进后的卡尔曼滤波器对于增长的用户请求数量更加敏感，能够做出迅速的反应，这也有助于优化基于改进的卡尔曼滤波预测算法的容器弹性服务部署策略的预测效果。

使用基于 ARMA 预测算法和基于改进的卡尔曼滤波预测算法的容器弹性服务部署策略对容器服务的规模进行动态调整，仿真结果如图6.3所示。

从图6.3中可以看出，所提出的基于预测的容器弹性服务部署策略能够有效跟随真实负载的变化，在用户服务请求数量增长的时候，所提出的策略能够迅速跟随这种变化，动态扩展容器服务规模，当用户服务请求数量减少的时候，所提出的策略出于减少并发以及减少容器服务规模的变动次数的目的，能够维持容

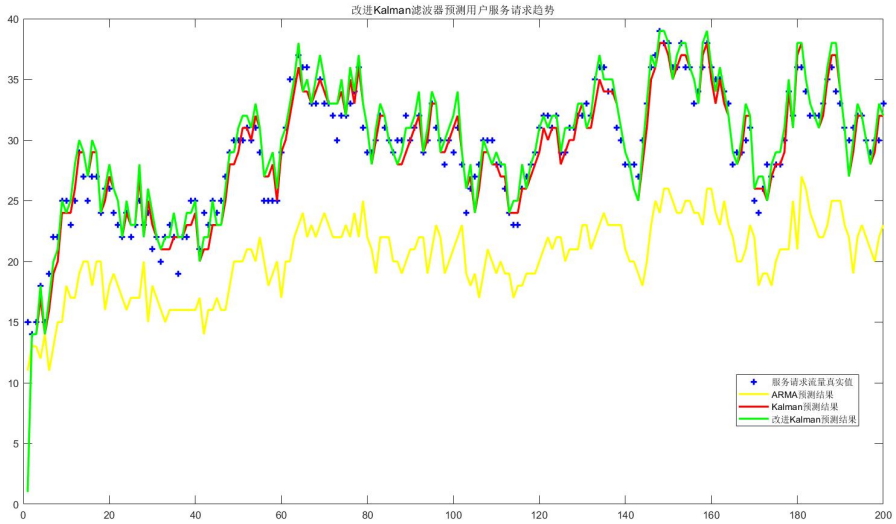

图 6.2 改进 Kalman 滤波器预测用户服务请求趋势

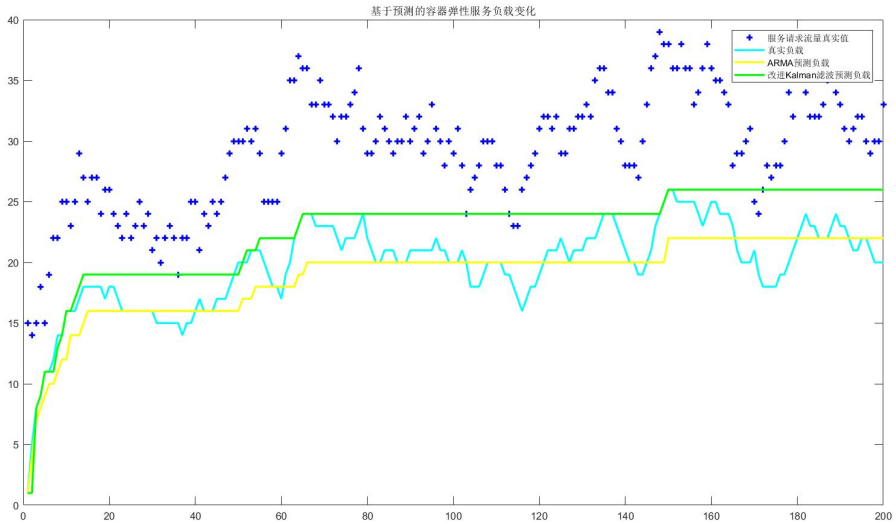

图 6.3 基于预测的容器弹性服务负载变化

器服务规模不变。相比于基于 ARMA 算法的部署策略，基于改进卡尔曼滤波预测算法的容器弹性服务部署策略的调整情况更加准确，使得因为策略调整不当而出现的执行任务数量超过容器所能承受最大并发数量的情况明显减少，提高了终端资源利用率，提高了用户体验。

6.6 本章小结

为了解决多终端协同服务技术中的预部署问题，促进提高终端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，本章提出了基于预测的容器弹性服务策略。基于卡尔曼滤波器，结合多终端协同服务的特点，提出了一种改进卡尔曼滤波算法。根据预测结果提前弹性部署容器服务，动态调整多终端协同服务的规模，平衡提高终端资源利用率与降低用户服务请求响应等待时间之间的关系。

本章所涉及的研究成果包括：

论文 1 篇：“基于卡尔曼滤波预测的容器弹性服务策略”（计算机与现代化，核心期刊，在投）

第 7 章　总结与展望

7.1　工作总结

随着边缘计算技术的快速发展，位于网络边缘的用户终端设备在人们的数字化生活中正扮演着越来越重要的角色。用户终端需要承担越来越多的计算任务，这也对终端设备的计算能力提出了越来越高的要求。同时在家庭、办公楼等很多场景中，还存在很多靠近用户的终端设备上空闲的计算资源，利用这些终端协同可以为用户提供就近的计算服务。

但是由于多终端设备资源分布分散、异构性强，存在资源利用难度大、资源利用率低、服务响应时间长、用户体验较差等问题。为此本文开展基于容器化多终端协同服务技术研究。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。

本文具体研究内容与成果如下：

1. 基于容器化多终端服务系统架构设计

针对终端设备资源分布较为分散、资源异构性强、不容易管理和利用等问题，设计了基于容器化多终端协同服务系统。在所提出的基于容器化多终端协同服务系统中，设计了资源管理模块，引入 Docker 容器虚拟化技术对多终端资源进行虚拟化，克服终端资源异构性，形成资源池，可以被上层终端服务根据对资源的类型和数量按需使用。除了对终端资源以容器的形式进行管理和利用，资源管理模块还需要对容器的生命周期进行管理。参考微服务架构，设计了服务管理模块，包括服务之间通过 REST 的轻量级通信机制进行通信、服务注册、服务节点选择、服务生命周期管理等。设计了任务调度模块，根据不同任务请求对资源的消耗情况，调度合适的节点进行执行。设计了弹性服务模块，统计用户服务请求数量，预测服务请求变化趋势，适当调节终端服务规模。另外还提出了去中心化的自组织网络结构，对多终端节点进行管理。

2. 基于容器的多终端透明计算迁移技术

为了将用户周围终端设备的空闲资源利用起来，在对应用保持透明的情况下，为 Web 应用提供计算迁移服务，提高终端资源利用率，缩短 Web 应用计算时间，提高终端用户体验，提出一种基于容器的 Web Worker 透明计算迁移技术。

通过对终端底层 Web 应用执行环境中的部分接口进行修改，将上层 Web 应用传来的 Web Worker 创建请求进行翻译并重新封装，通过 WebSocket 将封装后的请求发送到部署在边缘容器集群中的服务端进行处理。实验结果表明，相比 Web 应用本地执行，使用基于容器的多终端透明计算迁移技术能够在 Web Worker 数量较多的情况下，减少 Web 应用的总执行时间，最高能够减少 80.6%，对于提高终端用户体验有着明显的效果；相比非透明计算迁移技术，基于容器的多终端透明计算迁移技术最高能够减少应用执行时间 33.1%。

3. 多终端协同服务任务调度算法

为了解决多终端协同服务任务调度问题，选择更合适的终端执行终端服务任务，提高终端资源利用率，减少任务执行时间，提高用户体验，基于元启发式算法蝗虫优化算法，提出一种带有随机跳出机制的动态权重蝗虫优化算法。利用基于完全随机跳出因素的跳出机制，提高算法的跳出局部最优的能力；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。通过一系列 benchmark 测试函数测试，表明所提出的带有随机跳出机制的动态权重蝗虫优化算法在寻找最优结果方面具有不错的效果。为了更进一步优化算法在寻优问题和任务调度问题上的性能，基于上述改进算法基础，提出了一种改进蝗虫优化算法。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；提出基于 *Lévy* 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; 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。通过一系列 benchmark 测试函数测试，表明所提出的改进蝗虫优化算法，在寻找最优结果方面具有更好的效果。仿真实验结果表明，改进蝗虫优化算法在解决多终端协同服务任务调度问题上也能够取得较好的效果，相比其他对比算法效果最高提升 31.9%。

4. 基于预测的容器弹性服务策略

为了解决多终端协同服务技术中的预部署问题，促进提高终端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，提出了基于预测的容器弹性服务策略。基于卡尔曼滤波器，结合多终端协同服务的特点，提出了一种改进卡尔曼滤波算法。根据预测结果提前弹性部署容器服务，动态调整多终端协同服务的规模，平衡提高终端资源利用率与降低用户服务请求响应等待时间之间的关系。

7.2 工作展望

本文围绕基于容器化的多终端协同服务技术进行研究，进行了基于容器化多终端协同服务系统架构设计、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、基于预测的容器弹性服务策略等问题的研究，并取得了一些研究成果。

但是，本文的研究工作还存在着很多不足，还有着很多需要进行优化和完善的工作可以在未来做进一步的研究。针对第3章提出的基于容器化多终端协同服务系统架构，需要对提供协同服务的终端设备进行定期健康检查和终端资源状态监测，提供对终端本地服务性能优先保证的策略。针对第4章提出的基于容器的多终端透明计算迁移技术，需要考虑透明计算迁移的时机，以及研究除了面对 Web Worker 以外的其他形式的细粒度透明计算迁移技术。针对第5章提出的多终端协同服务任务调度算法，可以将任务调度模型进一步复杂化，考虑更多的复杂情况，也可以研究含有任务依赖关系的面向 DAG 图的任务调度算法。针对第6章提出的基于预测的容器弹性服务策略，可以进一步将自动化技术中的控制算法引入进来，对终端服务规模做更加精细的控制和调整。

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] TALEB T, DUTTA S, KSENTINI A, et al. Mobile edge computing potential in making cities smarter[J]. IEEE Communications Magazine, 2017, 55(3).

[4] 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.

[5] SUNDMAEKER H, GUILLEMIN P, FRIESS P, et al. Vision and challenges for realising the internet of things[J]. Cluster of European Research Projects on the Internet of Things, European Commision, 2010, 3(3):34-36.

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

[7] DINH H T, LEE C, NIYATO D, et al. A survey of mobile cloud computing: architecture, applications, and approaches[J]. Wireless communications and mobile computing, 2013, 13 (18):1587-1611.

[8] OTHMAN M, MADANI S A, KHAN S U, et al. A survey of mobile cloud computing application models[J]. IEEE Communications Surveys & Tutorials, 2014, 16(1):393-413.

[9] WANG Y, CHEN R, WANG D C. A survey of mobile cloud computing applications: perspectives and challenges[J]. Wireless Personal Communications, 2015, 80(4):1607-1623.

[10] NOOR T H, ZEADALLY S, ALFAZI A, et al. Mobile cloud computing: Challenges and future research directions[J]. Journal of Network and Computer Applications, 2018, 115: 70-85.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] ZHANG Q, CHENG L, BOUTABA R. Cloud computing: state-of-the-art and research challenges[J]. Journal of internet services and applications, 2010, 1(1):7-18.

[18] JOSEP A D, KATZ R, KONWINSKI A, et al. A view of cloud computing[J]. Communications of the ACM, 2010, 53(4).

[19] PARKHILL D F. Challenge of the computer utility[J]. 1966.

[20] SONNEK J, CHANDRA A. Virtual putty: Reshaping the physical footprint of virtual machines.[C]//HotCloud. 2009.

[21] 林伟伟, 齐德昱, 等. 云计算资源调度研究综述[J]. 计算机科学, 2012(2012 年 10):1-6.

[22] ROMAN R, LOPEZ J, MAMBO M. Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges[J]. Future Generation Computer Systems, 2018, 78:680-698.

[23] 刘英男. 基于云计算框架的终端管理系统设计与实现[D]. 西安电子科技大学, 2011.

[24] MANVI S S, SHYAM G K. Resource management for infrastructure as a service (iaas) in cloud computing: A survey[J]. Journal of network and computer applications, 2014, 41: 424-440.

[25] BHARDWAJ S, JAIN L, JAIN S. Cloud computing: A study of infrastructure as a service (iaas)[J]. International Journal of engineering and information Technology, 2010, 2(1):60-63.

[26] PAHL C. Containerization and the paas cloud[J]. IEEE Cloud Computing, 2015, 2(3):24-31.

[27] HUSSAIN A, SHARMA M P K. An architectural framework of cloud computing behind platform layer (paas)[J]. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 2018, 3(7):103-107.

[28] AMIRI A. Application placement and backup service in computer clustering in software as a service (saas) networks[J]. Computers & operations research, 2016, 69:48-55.

[29] SONG H, CHAUVEL F, SOLBERG A, et al. How to support customisation on saas: a grounded theory from customisation consultants[C]//2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). IEEE, 2017: 247-249.

[30] FOX A, GRIFFITH R, JOSEPH A, et al. Above the clouds: A berkeley view of cloud computing[J]. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 2009, 28(13):2009.

[31] 贾维嘉, 周小杰. 雾计算的概念, 相关研究与应用[J]. 通信学报, 2018, 39(5):153-165.

[32] VAQUERO L M, RODERO-MERINO L. Finding your way in the fog: Towards a comprehensive definition of fog computing[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(5):27-32.

[33] YI S, LI C, LI Q. A survey of fog computing: concepts, applications and issues[C]// Proceedings of the 2015 workshop on mobile big data. ACM, 2015: 37-42.

[34] 李子姝, 谢人超, 孙礼, 等. 移动边缘计算综述[J]. 电信科学, 2018, 34(1):87-101.

[35] BONOMI F, MILITO R, ZHU J, et al. Fog computing and its role in the internet of things[C]// Proceedings of the first edition of the MCC workshop on Mobile cloud computing. ACM, 2012: 13-16.

[36] YI S, HAO Z, QIN Z, et al. Fog computing: Platform and applications[C]//2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb). IEEE, 2015: 73-78.

[37] SHI W, CAO J, ZHANG Q, et al. Edge computing: Vision and challenges[J]. IEEE Internet of Things Journal, 2016, 3(5):637-646.

[38] 边缘计算产业联盟与工业互联网产业联盟. 边缘计算参考架构 3.0[R]. 边缘计算产业联盟与工业互联网产业联盟, 2018.

[39] REN J, GUO H, XU C, et al. Serving at the edge: A scalable iot architecture based on transparent computing[J]. IEEE Network, 2017, 31(5):96-105.

[40] 赵梓铭, 刘芳, 蔡志平, 等. 边缘计算: 平台, 应用与挑战[J]. 计算机研究与发展, 2018, 55 (2):327-337.

[41] 张静乐. 网络环境下协同服务关键技术研究[D]. 北京科技大学, 2011.

[42] LAUKKARINEN T, SUHONEN J, HÄNNIKÄINEN M. An embedded cloud design for internet-of-things[J]. International Journal of Distributed Sensor Networks, 2013, 9(11): 790130.

[43] 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.

[44] GOTH G. Virtualization: Old technology offers huge new potential[J]. IEEE Distributed Systems Online, 2007, 8(2):3-3.

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

[46] MENASCÉ D A. Virtualization: Concepts, applications, and performance modeling[C]//Int. CMG Conference. 2005: 407-414.

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

[48] D.KIRAN KUMAR T, B.RAJASEKHAR P. Review on virtualization for cloud computing[J]. International Journal of Advanced Research in Computer and Communication Engineering, 2014, 3:7748-7752.

[49] 叶蔚, 常青, 杨芳. 基于虚拟化的移动应用架构研究和设计[J]. 计算机工程与设计, 2019, 40(2):585-590.

[50] 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.

[51] 曹欣. 半虚拟化技术分析与研究[D]. 浙江大学, 2008.

[52] 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.

[53] PLESSL C, PLATZNER M. Virtualization of hardware-introduction and survey.[C]//ERSA. Citeseer, 2004: 63-69.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[C]//ACM SIGOPS operating systems review: volume 37. ACM, 2003: 164-177.

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

[63] 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.

[64] 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.

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

[66] CELESTI A, MULFARI D, FAZIO M, et al. Exploring container virtualization in iot clouds [C]//2016 IEEE International Conference on Smart Computing (SMARTCOMP). IEEE, 2016: 1-6.

[67] 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.

[68] 吴松, 王坤, 金海. 操作系统虚拟化的研究现状与展望[J]. 计算机研究与发展, 2019, 56(1):58-68.

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

[70] 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.

[71] 张文丽, 郭兵, 沈艳, 等. 智能移动终端计算迁移研究[J]. 计算机学报, 2016, 39(5):1021-1038.

[72] 崔勇, 宋健, 缪葱葱, 等. 移动云计算研究进展与趋势[J]. 计算机学报, 2017, 40(2):273-295.

[73] 徐乃凡, 王俊芳, 郭建立, 等. 面向边缘云高效能的移动终端计算迁移方法[J]. 电子测量技术, 2018, 41(20):1-6.

[74] BARBAROSSA S, SARDELLITTI S, DI LORENZO P. Communicating while computing: Distributed mobile cloud computing over 5g heterogeneous networks[J]. IEEE Signal Processing Magazine, 2014, 31(6):45-55.

[75] 董浩, 张海平, 李忠金, 等. 移动边缘计算环境下服务工作流的计算卸载[J]. 计算机工程与应用, 2019(2):6.

[76] MACH P, BECVAR Z. Mobile edge computing: A survey on architecture and computation offloading[J]. IEEE Communications Surveys & Tutorials, 2017, 19(3):1628-1656.

[77] 谢人超, 廉晓飞, 贾庆民, 等. 移动边缘计算卸载技术综述[J]. 通信学报, 2018, 39(11):138-155.

[78] SATYANARAYANAN M, BAHL P, CACERES R, et al. The case for vm-based cloudlets in mobile computing[J]. IEEE pervasive Computing, 2009(4):14-23.

[79] VERBELEN T, SIMOENS P, DE TURCK F, et al. Cloudlets: Bringing the cloud to the mobile user[C]//Proceedings of the third ACM workshop on Mobile cloud computing and services. ACM, 2012: 29-36.

[80] LI Y, WANG W. Can mobile cloudlets support mobile applications?[C]//IEEE INFOCOM 2014-IEEE Conference on Computer Communications. IEEE, 2014: 1060-1068.

[81] ZHANG J, ZHOU Z, LI S, et al. Hybrid computation offloading for smart home automation in mobile cloud computing[J]. Personal and Ubiquitous Computing, 2018, 22(1):121-134.

[82] 王硕, 孙鹏, 郭志川, 等. 嵌入式 Web 应用引擎的设计与实现[J]. 网络新媒体技术, 2016, 5(1):38-44.

[83] WANG Z, DENG H, HU L, et al. Html5 web worker transparent offloading method for web applications[C]//2018 IEEE 18th International Conference on Communication Technology (ICCT). IEEE, 2018: 1319-1323.

[84] ZBIERSKI M, MAKOSIEJ P. Bring the cloud to your mobile: Transparent offloading of html5 web workers[C]//2014 IEEE 6th International Conference on Cloud Computing Technology and Science. IEEE, 2014: 198-203.

[85] HWANG I, HAM J. Wwf: web application workload balancing framework[C]//2014 28th International Conference on Advanced Information Networking and Applications Workshops. IEEE, 2014: 150-153.

[86] HWANG I, HAM J. Cloud offloading method for web applications[C]//2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. IEEE, 2014: 246-247.

[87] GONG X, LIU W, ZHANG J, et al. Wwof: an energy efficient offloading framework for mobile webpage[C]//Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. ACM, 2016: 160-169.

[88] KURUMATANI S, TOYAMA M, CHEN E Y. Executing client-side web workers in the cloud [C]//2012 9th Asia-Pacific Symposium on Information and Telecommunication Technologies (APSITT). IEEE, 2012: 1-6.

[89] 王昭, 邓浩江, 胡琳琳, 等. 一种多服务端 HTML5 Web Worker 迁移系统设计与实现[J]. 计算机应用与软件, 2018, 35(9):27-31.

[90] ZHANG X, JEON W, GIBBS S, et al. Elastic html5: Workload offloading using cloud-based web workers and storages for mobile devices[C]//International Conference on Mobile Computing, Applications, and Services. Springer, 2010: 373-381.

[91] QIAO N, YOU J, SHENG Y, et al. An efficient algorithm of discrete particle swarm optimization for multi-objective task assignment[J]. IEICE TRANSACTIONS on Information and Systems, 2016, 99(12):2968-2977.

[92] GUO J, SONG Z, CUI Y, et al. Energy-efficient resource allocation for multi-user mobile edge computing[C]//GLOBECOM 2017-2017 IEEE Global Communications Conference. IEEE, 2017: 1-7.

[93] HUANG W, LI X, QIAN Z. An energy efficient virtual machine placement algorithm with balanced resource utilization[C]//2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. IEEE, 2013: 313-319.

[94] TSENG H W, WU R Y, CHANG T S. An effective vm migration scheme for reducing resource fragments in cloud data centers[C]//Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems. ACM, 2014: 320-325.

[95] TAWFEEK M A, EL-SISI A, KESHK A E, et al. Cloud task scheduling based on ant colony optimization[C]//2013 8th international conference on computer engineering & systems (ICCES). IEEE, 2013: 64-69.

[96] 乔楠楠, 尤佳莉. 一种面向网络边缘任务调度问题的多方向粒子群优化算法[J]. 计算机应用与软件, 2017, 34(4):309-315.

[97] ZAHARIA M, BORTHAKUR D, SARMA J S, et al. Job scheduling for multi-user mapreduce clusters[R]. Technical Report UCB/EECS-2009-55, EECS Department, University of California …, 2009.

[98] TABAK E K, CAMBAZOGLU B B, AYKANAT C. Improving the performance of independenttask assignment heuristics minmin, maxmin and sufferage[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(5):1244-1256.

[99] 杜玉霞, 刘方爱, 郭磊. Min-Min 调度算法的研究与改进[J]. 计算机工程与应用, 2010, 46(24):107-109.

[100] AL-OLIMAT H S, ALAM M, GREEN R, et al. Cloudlet scheduling with particle swarm optimization[C]//2015 Fifth International Conference on Communication Systems and Network Technologies. IEEE, 2015: 991-995.

[101] 刘运, 程家兴, 林京. 基于高斯变异的人工萤火虫算法在云计算资源调度中的研究[J]. 计算机应用研究, 2015(3):834-837.

[102] FONSECA C M, FLEMING P J. An overview of evolutionary algorithms in multiobjective optimization[J]. Evolutionary computation, 1995, 3(1):1-16.

[103] D W. A genetic algorithm tutorial[J]. Statistics and computing, 1994, 4(2):65-85.

[104] KENNEDY J, EBERHART R. Particle swarm optimization[C]//Neural Networks, 1995. Proceedings., IEEE International Conference on: volume 4. IEEE, 1995: 1942-1948.

[105] LIAO C J, TSENG C T, LUARN P. A discrete version of particle swarm optimization for flowshop scheduling problems[J]. Computers & Operations Research, 2007, 34(10):3099-3111.

[106] KRISHNASAMY K, et al. Task scheduling algorithm based on hybrid particle swarm optimization in cloud computing enviroment[J]. Journal of Theoretical & Applied Information Technology, 2013, 55(1).

[107] DORIGO M, GAMBARDELLA L M. Ant colonies for the travelling salesman problem[J]. biosystems, 1997, 43(2):73-81.

[108] DORIGO M, DI CARO G. Ant colony optimization: a new meta-heuristic[C]//Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406): volume 2. IEEE, 1999: 1470-1477.

[109] HAO H, JIN Y, YANG T. Network measurement node selection algorithm based on parallel aco algorithm[J]. Journal of Network New Media, 2018.

[110] 桓自强, 倪宏, 胡琳琳, 等. AAFSA-RA: 一种采用高级人工鱼群算法的多资源分配方法[J]. 西安交通大学学报, 2014, 48(10):120-125.

[111] MIRJALILI S. The ant lion optimizer[J]. Advances in Engineering Software, 2015, 83: 80-98.

[112] MIRJALILI S, LEWIS A. The whale optimization algorithm[J]. Advances in engineering software, 2016, 95:51-67.

[113] MIRJALILI S. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems[J]. Neural Computing and Applications, 2016, 27(4):1053-1073.

[114] SAREMI S, MIRJALILI S, LEWIS A. Grasshopper optimisation algorithm: theory and application[J]. Advances in Engineering Software, 2017, 105:30-47.

[115] MA X, ZHAO Y, ZHANG L, et al. When mobile terminals meet the cloud: computation offloading as the bridge[J]. IEEE Network, 2013, 27(5):28-33.

[116] QI H, GANI A. Research on mobile cloud computing: Review, trend and perspectives[C]// 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP). ieee, 2012: 195-202.

[117] LIU F, SHU P, JIN H, et al. Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications[J]. IEEE Wireless communications, 2013, 20(3): 14-22.

[118] LUAN T H, GAO L, LI Z, et al. Fog computing: Focusing on mobile users at the edge[J]. arXiv preprint arXiv:1502.01815, 2015.

[119] MAO Y, ZHANG J, LETAIEF K B. Dynamic computation offloading for mobile-edge computing with energy harvesting devices[J]. IEEE Journal on Selected Areas in Communications, 2016, 34(12):3590-3605.

[120] SUN X, ANSARI N. Edgeiot: Mobile edge computing for the internet of things[J]. IEEE Communications Magazine, 2016, 54(12):22-29.

[121] MAO Y, YOU C, ZHANG J, et al. A survey on mobile edge computing: The communication perspective[J]. IEEE Communications Surveys & Tutorials, 2017, 19(4):2322-2358.

[122] 马斌, 冯波. 海云协同环境下服务运营环境关键技术的探讨[J]. 网络新媒体技术, 2015, 4(2):1-9.

[123] LEWIS J, FOWLER M. Microservices: a definition of this new architectural term[J]. MartinFowler.com, 2014, 25.

[124] 龙新征, 彭一明, 李若淼, 等. 基于微服务框架的信息服务平台[J]. 东南大学学报 (自然科学版), 2017, 1.

[125] 郭栋, 王伟, 曾国荪. 一种基于微服务架构的新型云件 PaaS 平台[J]. 信息网络安全, 2015, 15(11):15-20.

[126] 谭一鸣. 基于微服务架构的平台化服务框架的设计与实现[D]. 北京: 北京交通大学, 2017.

[127] 唐文宇. 面向 SOA 架构微服务的安全系统的设计与实现[D]. 南京: 南京大学, 2016.

[128] 肖仲垚. 微服务通信框架的设计与实现[D]. 北京: 北京交通大学, 2017.

[129] 宋鹏威. 开放式微服务框架的设计与应用[D]. 北京邮电大学, 2017.

[130] HASSELBRING W, STEINACKER G. Microservice architectures for scalability, agility and reliability in e-commerce[C]//2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017: 243-246.

[131] INNERBICHLER J, GONUL S, DAMJANOVIC-BEHRENDT V, et al. Nimble collaborative platform: Microservice architectural approach to federated iot[C]//2017 Global Internet of Things Summit (GIoTS). IEEE, 2017: 1-6.

[132] GRANCHELLI G, CARDARELLI M, DI FRANCESCO P, et al. Microart: A software architecture recovery tool for maintaining microservice-based systems[C]//2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017: 298-302.

[133] MAYER B, WEINREICH R. A dashboard for microservice monitoring and management[C]//2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017: 66-69.

[134] AMARAL M, POLO J, CARRERA D, et al. Performance evaluation of microservices architectures using containers[C]//2015 IEEE 14th International Symposium on Network Computing and Applications. IEEE, 2015: 27-34.

[135] UEDA T, NAKAIKE T, OHARA M. Workload characterization for microservices[C]//2016 IEEE international symposium on workload characterization (IISWC). IEEE, 2016: 1-10.

[136] KALSKE M, MÄKITALO N, MIKKONEN T. Challenges when moving from monolith to microservice architecture[C]//International Conference on Web Engineering. Springer, 2017: 32-47.

[137] CONWAY M E. How do committees invent[J]. Datamation, 1968, 14(4):28-31.

[138] 牛力人, 刘宇靖, 彭伟, 等. 互联网关键自治系统的地理分布特性分析[J]. 信息安全与技术, 2015(2015 年 04):22-26.

[139] 寇明延, 熊华钢, 李峭, 等. 面向任务能力的自组织网络体系结构[J]. 系统工程与电子技术, 2010, 35(5):980-986.

[140] RYU B, ANDERSEN T, ELBATT T, et al. Multitier mobile ad hoc networks: architecture, protocols, and perfomance[C]//IEEE Military Communications Conference, 2003. MILCOM 2003.: volume 2. IEEE, 2003: 1280-1285.

[141] KHAN Z H, CATALOT D G, THIRIET J M. Hierarchical wireless network architecture for distributed applications[C]//2009 Fifth International Conference on Wireless and Mobile Communications. IEEE, 2009: 70-75.

[142] 刘福杰, 常义林, 沈中, 等. 一种自组织网络管理实现方法的研究[J]. 西安電子科技大學學報, 2004, 31(2):182-185.

[143] CHAE D, KIM J, KIM J, et al. Cmcloud: Cloud platform for cost-effective offloading of mobile applications[C]//2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE, 2014: 434-444.

[144] CHUN B G, IHM S, MANIATIS P, et al. Clonecloud: elastic execution between mobile device and cloud[C]//Proceedings of the sixth conference on Computer systems. ACM, 2011: 301-314.

[145] KOSTA S, AUCINAS A, HUI P, et al. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading[C]//2012 Proceedings IEEE Infocom. IEEE, 2012: 945-953.

[146] SHI C, HABAK K, PANDURANGAN P, et al. Cosmos: computation offloading as a service

for mobile devices[C]//Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing. ACM, 2014: 287-296.

[147] ISMAIL B I, GOORTANI E M, AB KARIM M B, et al. Evaluation of docker as edge computing platform[C]//2015 IEEE Conference on Open Systems (ICOS). IEEE, 2015: 130-135.

[148] WU S, NIU C, RAO J, et al. Container-based cloud platform for mobile computation offloading[C]//2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2017: 123-132.

[149] 陈霄, 郭志川, 孙鹏, 等. 基于 Web 浏览器的远程容器登录系统设计[J]. 网络新媒体技术, 2017, 6(6):61-63.

[150] TANESE R. Distributed genetic algorithms for function optimization[J]. 1989.

[151] NAGHAM AZMI AL-MADI E A M, Khulood Abu Maria, AL-MADI M A. A structured-population human community based genetic algorithm (hcbga) in a comparison with both the standard genetic algorithm (sga) and the cellular genetic algorithm (cga)[J]. ICIC Express Letters, 2018, 12(12):1267-1275.

[152] KIRKPATRICK S, GELATT C D, VECCHI M P. Optimization by simulated annealing[J]. science, 1983, 220(4598):671-680.

[153] HONG G, ZUCKERMANN M, HARRIS R, et al. A fast algorithm for simulated annealing [J]. Physica Scripta, 1991, 1991(T38):40.

[154] RUTENBAR R A. Simulated annealing algorithms: An overview[J]. IEEE Circuits and Devices magazine, 1989, 5(1):19-26.

[155] 郝航, 金跃辉, 杨谈, 等. 基于并行化蚁群算法的网络测量节点选取算法[J]. 网络新媒体技术, 2018(2018 年 01):7-15.

[156] YANG X S, DEB S. Cuckoo search via lévy flights[C]//2009 World Congress on Nature & Biologically Inspired Computing (NaBIC). IEEE, 2009: 210-214.

[157] HU H, BAI Y, XU T. A whale optimization algorithm with inertia weight[J]. WSEAS Trans. Comput, 2016, 15:319-326.

[158] ABDEL-BASSET M, LUO Q, MIAO F, et al. Solving 0–1 knapsack problems by binary dragonfly algorithm[C]//International Conference on Intelligent Computing. Springer, 2017: 491-502.

[159] KS S R, MURUGAN S. Memory based hybrid dragonfly algorithm for numerical optimization problems[J]. Expert Systems with Applications, 2017, 83:63-78.

[160] SONG J, LI S. Elite opposition learning and exponential function steps-based dragonfly algorithm for global optimization[C]//2017 IEEE International Conference on Information and Automation (ICIA). IEEE, 2017: 1178-1183.

[161] EWEES A A, ELAZIZ M A, HOUSSEIN E H. Improved grasshopper optimization algorithm using opposition-based learning[J]. Expert Systems with Applications, 2018, 112:156-172.

[162] ARORA S, ANAND P. Chaotic grasshopper optimization algorithm for global optimization [J]. Neural Computing and Applications, 2018:1-21.

[163] MOLGA M, SMUTNICKI C. Test functions for optimization needs[J]. Test functions for optimization needs, 2005, 101.

[164] LIANG J J, SUGANTHAN P N, DEB K. Novel composition test functions for numerical global optimization[C]//Proceedings 2005 IEEE Swarm Intelligence Symposium, 2005. SIS 2005. IEEE, 2005: 68-75.

[165] FENG H, SHU Y. Study on network traffic prediction techniques[C]//Proceedings. 2005 International Conference on Wireless Communications, Networking and Mobile Computing, 2005.: volume 2. IEEE, 2005: 1041-1044.

[166] 高波. 基于时间相关的网络流量建模与预测研究[D]. 哈尔滨工业大学, 2013.

[167] 段智彬, 孙恩昌, 张延华, 等. 基于 ARMA 模型的网络流量预测[J]. 中国电子科学研究院学报, 2009, 4(4):352-356.

[168] 陈广居, 梁鹏, 王坤. 网络流量预测模型研究[J]. 信息通信, 2017(8):191-194.

[169] 林义勇, 鞠福昌, 卢艳. ARMA 模型在局域网短期流量预测中的应用研究[C]//第十一届全国信号和智能信息处理与应用学术会议专刊. 2017.

[170] KAYACAN E, ULUTAS B, KAYNAK O. Grey system theory-based models in time series prediction[J]. Expert systems with applications, 2010, 37(2):1784-1789.

[171] KALMAN R E. A new approach to linear filtering and prediction problems[J]. Journal of basic Engineering, 1960, 82(1):35-45.

[172] 彭丁聪. 卡尔曼滤波的基本原理及应用[J]. 软件导刊, 2009, 8(11):32-34.

[173] MURUGANANTHAM A, TAN K C, VADAKKEPAT P. Evolutionary dynamic multiobjective optimization via kalman filter prediction[J]. IEEE transactions on cybernetics, 2016, 46 (12):2862-2873.

[174] 成光, 刘卫东, 魏尚俊, 等. 基于卡尔曼滤波的目标估计和预测方法研究[J]. 计算机仿真, 2006, 23(1):8-10.

[175] 刘鑫, 滕欢, 宫毓斌, 等. 基于改进卡尔曼滤波算法的短期负荷预测[J]. 电测与仪表, 2019, 56(03):42-46.

[176] 周艳青, 薛河儒, 姜新华, 等. 基于改进的卡尔曼滤波算法的气象数据融合[J]. 计算机系统应用, 2018, 27(4):184-189.

[177] 郝庭毅, 吴恒, 吴国全, 等. 面向微服务架构的容器级弹性资源供给方法[J]. 计算机研究与发展, 2017, 54(3):597-608.

[178] 贾濡, 潘沭铭. 基于卡尔曼滤波的流量预测机制[J]. 中国电子科学研究院学报, 2018, 13 (06):82-87.

[179] WELCH G, BISHOP G, et al. An introduction to the kalman filter[J]. 1995.

致　谢

时光如白驹过隙，一转眼走到了二十载求学生涯的最终章。曾经无数次幻想，在毕业论文的收尾工作里，要如何来书写致谢这一章，然而当这一天真的到来的时候，却又迟迟不敢动笔。一则是因为一路走来，要感谢的人很多，老师、同学、朋友、家人，万千回忆凝聚于笔尖，竟不知从何处着墨。二则是因为总是害怕这一章写完，就真的要同自己的学生时代告别了。一直拖到今天，望着窗外凌晨 4 点的北京，听着陈绮贞的《送别》的歌声，终于还是决定开始写了。

首先要感谢我的导师倪宏老师这五年来对我的指导与帮助。倪老师无论平时工作多么繁忙都会抽出时间指导我的科研工作，在我进行论文选题、学术写作、博士开题答辩、博士中期答辩等工作的过程中，倪老师为我提出了很多宝贵的意见与建议，倪老师认真负责的做事风格、严谨的治学态度、高瞻远瞩的学术视野，都让我受益匪浅。

感谢我的导师郭志川老师，在我攻读博士学位期间对我的整体科研方向以及具体的研究内容都进行了认真细致的指导，并在我学术论文写作和修改的过程中为我提供了很多帮助。在研究组内的科研工作中，郭老师也带领我参与了很多科研项目的立项和研究的过程，让我在求学之路上收获颇丰。

感谢国家网络新媒体工程技术研究中心主任王劲林研究员。王老师总是能够深入浅出地分析学术问题，为我指点迷津，对我的论文选题工作给予了很多的帮助。在中心学习的这几年，王老师渊博的学识、认真务实的科研态度、勤奋的工作作风、广阔的学术视野，都给我留下了深刻的印象，值得我好好学习。王老师不仅关心每一位同学的科研进展，更关心大家的身心健康。在我读博期间遇到来自学习和生活的巨大压力的时候，是王老师通过积极的沟通、主动的交流、耐心的倾听和认真的指导帮助我走出心理困境，坦然面对压力，顺利完成学业。

感谢中心副主任邓浩江研究员。邓老师在保研面试中让我有机会进入这个实验室完成博士阶段的学习和工作。也感谢邓老师在我读博期间对我研究方向及论文的指导以及对我生活和身体健康方面的关心。

感谢终端研究室主任朱小勇老师。在中心的工程工作与科研工作中，接触最多的就是项目组长朱老师了。感谢朱老师带着我参与了很多工程项目。从安徽广电智能终端项目到欧洲媒体服务支撑平台项目，从北京到深圳，从 DSP 二楼的终端组到 DSP 五楼的小黑屋，朱老师用认真负责的工作态度、优秀的技术能

力、丰富的工程经验以及无比的耐心与包容，帮助我从编程小菜鸟成长为可以独当一面的工程骨干。那些跟着朱师兄一起做项目、一起出差、一起熬夜赶项目节点、一起开会讨论的日子，是我迅速收获、迅速成长的时光，也是我读博阶段非常难忘的经历。也感谢朱老师在科研项目中对我的指导与帮助。无论是嵌入式容器的研究，还是多终端协同服务技术的研究，朱老师都能够帮助我理清研究的思路，选择合适的技术路线，并为我的研究工作提出了很多建设性意见，帮助我完成相关的论文和专利。

感谢终端组韩锐老师、胡琳琳老师、周学志老师、马凤华老师、刘春梅老师对我的科研工作与工程工作的指导与帮助。

感谢的程钢老师、叶晓舟老师、曾学文老师、陈晓老师、王玲芳老师、刘学老师、盛益强老师、脱立恒老师、田野老师、冯丽茹老师、姜艳老师、陈君老师、冯新老师、任晓青老师、杨静漪老师、徐莉老师等在我读博期间对我的指导与帮助。

感谢中心主管学生工作的卢美英老师、尤佳莉老师、杨中臻师兄这五年来对我们的生活方面与身心健康方面的关心与照顾。

感谢中心的吴京洪师兄、董海韬师兄、樊浩师兄、卓煜师姐、黄河师兄、曹作伟师兄、李超鹏师兄在我初入实验室、论文写作、论文投稿、找工作、准备毕业等过程中提供的宝贵经验与帮助，让我少走了很多弯路。

感谢终端组的王旭师兄、黄兴旺师兄、肖伟民师兄、耿筱林师姐、耿立宏师兄、刘丽琴师姐、陈霄师弟、李超师弟、马博韬师弟、宋锐星师妹、包沙如拉师妹、王慧鑫师妹、宋雅琴师妹、刘轶峰师弟、董翰泽师弟、冯航伟师弟、张晶师妹、刘彤师妹、方立师弟、段英杰师妹、许勇师弟、杨丹师妹、晁一超师弟。忘不了组里那些朝夕相处的日子：新人破冰时紧张的自我介绍，饭后闲聊的欢声笑语，每周组会上的工作交流，元旦的时候一起聊天跨年，找工作时候的互相帮助，科研过程中的学术讨论、迷茫无助时候的陪伴与关心。那些充满汗水与欢笑的时光，一点一滴，组成了我读博生活的日常。

特别感谢新媒体 14 级一起求学的同窗好友麻朴方、许丹凤、田娟娟、伍洪桥、姜凯华、薛寒星、廖怡、常乐、李强、黎江源、邓丽君、郑抗、贾正义、唐志斌、王昭、唐政治。这五年来经历的那些当时只道是寻常的时光，那些本以为长得看不到尽头的日子，都成为我最闪光的回忆。还没说再见，已经开始怀念。怀念在怀柔一起滑雪一起爬山的日子，怀念跨年夜的彻夜长谈，怀念一起复习准备期末考试的时光，怀念答辩前夜的并肩作战，怀念最艰难的时候大家的互相鼓

励和加油打气。感谢你们让我知道，我不是一个人在战斗，成长之路上有你们的陪伴，真好！

特别感谢王旭和廖怡，感谢你们在我最低谷的时候花时间陪伴我，耐心地听我倾诉那些傻瓜的念头，认真地开导我帮助我，让我逃离那个梦魇的吞噬，顺利完成学业。

特别感谢伍洪桥。从清华到果壳，从紫荆公寓到青年公寓，从五道口到雁栖湖，从佳木斯到垦丁，都留下了我们友谊的见证。九年同学，七年室友，一起吃饭，一起打球，一起游戏玩耍，一起学习成长，感谢你多年以来的陪伴与帮助。

特别感谢一生的挚友张恩泽和韩智敏。感谢你们多年以来做我坚实的后盾和力量的源泉。虽然远隔千山万水，但是我知道你们一直在我身边。

特别感谢我的父母，感谢你们给了我健康的身体和快乐的成长环境，让我能够成长成为今天的自己。感谢你们二十多年来在背后默默地付出，给我无条件的支持与信任，在我得意的时候给我警示，在我失意的时候给我安慰与鼓励，你们永远是我前进的动力，谢谢你们！

最后，感谢每一位陪我走过这段路的人们，千言万语，锦书遥寄，长亭送别，相顾依依。

赵然

2019 年 4 月于北京

作者简历及攻读学位期间发表的学术论文与研究成果

作者简历

赵然，男，内蒙古通辽人，1992 年出生，中国科学院声学研究所博士研究生。

2010 年 08 月——2014 年 07 月，在清华大学自动化系获得学士学位。

2014 年 09 月——2019 年 07 月，在中国科学院声学研究所攻读博士学位。

已发表 (或正式接受) 的学术论文:

[1] Ran Zhao, Hong Ni, Hangwei Feng, Yaqin Song, Xiaoyong Zhu. An Improved Grasshopper Optimization Algorithm for Task Scheduling Problems[J]. International Journal of Innovative Computing, Information and Control, 2019（EI 期刊，已录用）

[2] Ran Zhao, Hong Ni, Hangwei Feng, Xiaoyong Zhu. A Dynamic Weight Grasshopper Optimization Algorithm with Random Jumping[C]. 2018 3rd International Conference on Computer, Communication and Computational Sciences (IC4S2018),2018（EI 会议，已录用）

[3] 赵然, 朱小勇. 微服务架构评述 [J]. 网络新媒体技术,2019,8(01):58-61+65.（核心期刊，已发表）

[4] 赵然, 郭志川, 朱小勇. 基于容器的 Web Worker 透明边缘计算迁移系统 [J]. 微电子学与计算机.（核心期刊，在投）

[5] 赵然, 郭志川, 朱小勇. 基于卡尔曼滤波预测的容器弹性服务策略 [J]. 计算机与现代化.（核心期刊，在投）

申请或已获得的专利:

1. 朱小勇,赵然,一种微服务故障检测处理方法及装置. 申请号:201711368632.3

2. 朱小勇，常乐，郭志川，赵然，一种 Docker 容器多进程管理方法及系统. 申请号：201611090130.4

3. 朱小勇，邓丽君，郭志川，赵然，常乐，一种基于嵌入式系统的 RancherOS ros 核心模块移植方法. 申请号：201611069177.2

软件著作权:

1. 朱小勇，赵然，视频点播平台客户端软件. 登记号：2017SRBJ0226

2. 韩锐、李超、赵然、朱小勇、郭志川,机顶盒IP升级软件. 登记号:2017SRBJ0116

参加的研究项目:

1. 中国科学院声学研究所率先行动：端到端虚拟化关键技术研究与系统开发. 编号：Y654101601

2. 中国科学院声学研究所青年英才：嵌入式容器文件系统关键技术研究（编号：QNYC201714）

3. 中国科学院先导专项课题：SEANET 技术标准化研究与系统研制（编号：XDC02010801）

4. 欧洲媒体服务支撑平台项目

5. 国家科技支撑计划课题:电视商务综合体新业态应用示范（编号:2012BAH73F02）

获奖情况:

2016-2017 学年，获中国科学院大学三好学生