# 面向多计算框架的容器云资源调度研究与实现

（申请清华大学工学硕士学位论文）

培 养 单 位：计 算 机 科 学 与 技 术 系
学　　科：计 算 机 科 学 与 技 术
研　究　生：龚　坤
指 导 教 师：武　永　卫　教　授

二〇一九年六月

# Research and Implementation of Container Cloud Resource Scheduling for Multi-dimensional Computing Framework

Thesis Submitted to

**Tsinghua University**

in partial fulfillment of the requirement

for the professional degree of

**Doctor of Engineering**

by

**Gong Kun**

**( Computer Science and Technology )**

Thesis Supervisor : Professor Wu Yongwei

**June, 2019**

# 关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

**（保密的论文在解密后应遵守此规定）**

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

# 摘 要

Docker 容器虚拟化技术是一种共享操作系统内核的虚拟化解决方案，基于 Docker 容器化技术的云平台日益发展，逐步成为下一代云计算的核心。容器云中资源调度器对集群的性能和资源利用率起到决定性作用，Kubernetes 容器编排引擎以其强大的服务发现、集群监控、错误恢复能力成为当前应用最为广泛的容器云平台调度系统。但 Kubernetes 过度专注于容器性能和编排能力，其调度算法的单一导致整个容器云集群资源利用率和负载均衡性较差，尤其是在多计算框架容器应用对大数据处理的场景下，其调度算法的缺陷暴露无遗。

OpenShift Origin 是一个基于主流容器技术 Docker 和 Kubernetes 构建的 PaaS 层云平台，其调度性能完全依赖容器编排的调度算法。实验室项目大数据高效能存储与处理容器云平台 Paladin 的服务层是基于开源 OpenShift Origin 研发，针对其调度性能不足，在多计算框架下大数据处理场景下对其调度流程进行优化，提出了一种新的调度策略，极大提升了集群性能。本文主要贡献如下:

- 基于开源的 OpenShift Origin 和实验室项目 Paladin Storage 构建大数据存储与处理容器云平台 Paladin，在该平台上开发 Hadoop、Spark、MPI、Storm 等十多种分布式处理框架，用户可以快速构建大数据处理框架和容器伸缩。
- 深入研究 Kubernetes 调度核心技术，针对其调度算法不足，提出了一种基于多维资源空闲率权重的评价函数和调度方法 MRWS(Multidimensional Resource Weights Scheduling)。该方法综合考虑物理节点 CPU、内存、磁盘、网络带宽空闲率和已部署的容器应用个数等因素影响，使用模糊层次分析法 FAHP(Fuzzy Analytic Hierarchy Process) 对集群资源自动建模并求解容器应用多维资源权重参数，选取最大评分节点进行容器调度。
- 针对 MRWS 调度策略，在容器云仿真平台 ContainerCloudSim 上进行大规模调度仿真，并与 Random、FirstFit、Kubernetes 默认调度策略在资源利用率和负载均衡性方面进行对比。
- 在 Paladin 上设计并实现 MRWS 调度策略，针对多计算框架下大数据处理场景，比较 MRWS 与其他调度策略下的集群性能。新的调度策略无论在单计算框架容器应用还是多计算框架容器应用混合执行效率都有极大的提升。

**关键词**：Kubernetes；OpenShift 平台；多计算框架；调度策略；ContainerCloudSim；FAHP 权重参数

# Abstract

Docker container virtualization technology is a virtualization solution that shares the operating system kernel. The cloud platform based on Docker containerization technology is gradually becoming the core of the next generation cloud computing. In the container cloud, the resource scheduler plays a key role in the performance and resource utilization. Kubernetes container orchestration engine has become the most widely used container cloud platform scheduling system with its powerful service discovery, cluster monitoring and error recovery capabilities. However, Kubernetes is overly focused on container performance and orchestration capabilities. The single scheduling algorithm leads to poor resource utilization and load balancing of the entire container cloud cluster, especially in the scenario of multi-computing framework container application for big data processing.

OpenShift Origin is a PaaS layer container cloud platform which builds on Docker and Kubernetes. Its scheduling performance relies entirely on Kubernetes' scheduling algorithm. Lab project big data high-performance storage and processing container cloud platform Paladin's service layer develops from open source OpenShift Origin, for its lack of scheduling performance, in the multi-computing framework under the big data processing scenario to optimize its scheduling process, proposed a A new scheduling strategy, which greatly improves cluster performance. The main contributions of this paper are as follows:

- The big data storage and processing container platform Paladin is based on open source OpenShift Origin and lab project paladin storage. On the platform, more than ten distributed processing frameworks such as Hadoop, Spark, MPI and Storm are developed, and users can quickly build big data processing and telescopic container.
- In-depth study of Kubernetes scheduling core technology and for its lack of scheduling algorithm, an evaluation function and scheduling method based on multi-dimensional resource idle rate weight (MRWS) (Multidimensional Resource Weights Scheduling) is proposed. The method considers the physical node CPU, memory, disk, network bandwidth idle rate and the number of deployed container applications, and uses the Fuzzy Analytic Hierarchy Process (FAHP) to automatically model the cluster resources and solve the weight parameters. The MRWS stategy selects the largest scoring node for container scheduling.

- Large-scale scheduling simulation was performed on the container cloud simulation platform ContainerCloudSim for the MRWS scheduling strategy,and compared with the default scheduling strategy of Kubernetes, FirstFit and Random in terms of resource utilization and load balancing.
- The MRWS scheduling strategy is designed and implemented on Paladin platform, and the cluster performance under MRWS and other scheduling strategies is compared for big data processing scenarios under multi-computing framework. The new scheduling strategy has greatly improved the efficiency of hybrid execution in both single-computing framework container applications and multi-computing framework container applications.


**Key Words:** Kubernetes; OpenShift platform; multiple computing framework; scheduling strategy; ContainerCloudSim; FAHP weight parameters

# 目 录

# 第 1 章　引言

随着计算机互联网技术的飞速发展，网络规模不断扩大，各行业中应用业务量和数据都呈现爆炸式的增长，如何存储和高效处理各种应用产生的海量数据已成为各大互联网公司面临的一个巨大挑战。继分布式计算、网格计算和并行计算后，一种全新的将整个互联网资源聚合起来处理数据的计算模式应运而生: 云计算[1]。这种按需付费、共享资源、统一管理、可伸缩、可度量的计算模式发展迅猛，已经给信息产业带来了巨大的影响。

然而，云计算往往是以虚拟机作为云主机进行构建，将用户对资源的需求和软件服务虚拟化成虚拟机资源，然后进行虚拟机创建、操作系统安装以及应用部署。这种虚拟化方式需要巨大的虚拟化开销，并且不同的虚拟机操作系统不同，其应用跨平台性较差。近年来，容器逐步取代虚拟机技术成为云计算的主流技术，给云计算带来新的革命，尤其是以 Docker[2] 为代表的容器虚拟化技术获得了飞速的发展，基于 Docker 技术的容器云如雨后春笋般出现。云平台以其快速的应用部署、启动、交付以及优异的跨平台性能迅速占领市场。

与传统的云计算模式相似，一个强大且高效的资源调度系统对集群性能和资源利用率起到决定性作用，Docker 虚拟化作为一种新型的容器云技术解决方案，其容器编排能力还存在很多不足。Google 开发的 Kubernetes 是容器云中容器调度系统的优秀代表，其轻量开源和强大的编排能力深受深受人们好评，但是其调度算法单一和资源利用不均衡性成为制约其性能的重要因素，本文在深入研究 Kubernetes 调度核心后，针对其调度性能不足，设计并实现了一种新的调度方案，在基于开源 OpenShift Origin 和实验室项目 paldin storage 构建的大数据存储与处理的容器云平台 Paladin 上设计并实现新的调度方案，极大提升了大数据处理场景下多种计算框架容器应用的性能。

## 1.1　云计算调度概述

云计算是当前较为普遍的一种分布式计算方式，通过将计算资源聚合成资源共享池对外提供按需付费、弹性计算的能力，对当前的计算机技术带来巨大的影响。其服务资源池有基础设施即服务 (IaaS，Infrastructure as a Service)、平台即服务 (PaaS，Platform as a Service) 以及软件即服务 (SaaS，Software as a Service) 三种服务模式。其计算类型根据用户对象的不同可以划分为公有云、私有云和混合云，典型的公有云有 Google Gmail、Amazon 的 EC2、微软 Azure、阿里云 ECS、百度

云、腾讯云等。云计算需要底层的虚拟化技术作为支撑，当前维基百科收录的就有超过 60 种虚拟化技术，其中基于 X86 体系的虚拟化超过 50 种，当然也有 RISC 体系的虚拟化，主要包括硬件虚拟化、操作系统层虚拟化、桌面虚拟化、应用程序虚拟化以及网络虚拟化等。从虚拟化的实现方式上可以划分为宿主架构和裸金属架构两种方式，其中宿主架构中虚拟机作为宿主操作系统的一个进程进行调度和管理，主要在个人的 PC 端应用较为广泛，如 VirtualBox、VMware Workstation、WindowVirtual PC 等。裸金属架构则不需要主机操作系统，直接以 Hypervisor 运行于物理硬件上，主要应用于服务器的虚拟化。应用最为广泛的如微软的 Hyper-V、VMWARE 的 ESX、开源的 XEN 和 KVM 等，云计算虚拟化架构中通常以虚拟机的方式提供给用户，用户根据自己的资源需求和软件服务申请合适的虚拟机进行服务，在进行大规模云计算环境构建时应选择合适的虚拟化架构方式，实现统一管理和跨平台的资源调度，综合利用各种虚拟化的性能优势。

在云计算中，资源的调度器对集群的性能和资源利用率起到决定性作用，是云计算的核心。传统虚拟机式的云计算集群对调度策略有相当多的研究[3]，主要集中在降低系统能耗、提高数据中心资源利用率、集群服务器的负载均衡以及基于成本模式的资源管理研究。文献 [4] 提出一种根据虚拟机负载动态调节处理器电压和频率来降低集群能耗；文献 [5] 通过动态分配云计算中心的虚拟机，减少服务器的数量来节约能耗；文献 [6] 通过提出一种中心平衡器的平衡算法来实现集群服务器的负载均衡；文献 [7] 将应用需求和物理机计算资源建模，基于蚁群算法、粒子群算法等迭代方式求解最佳分配策略，减少服务器的数量，提升集群资源利用率；文献 [8] 提出面向市场的体系结构和资源分配调度方法，该体系结构通过 SLA 资源分配器实现用户和服务商的协商，从而实现资源的优化配置。由此看出，在虚拟机式的云计算中，人们对云计算资源调度方法进行深入的研究，广泛应用于当前的云计算系统中，对推动云计算普及起到巨大的作用。

## 1.2　容器云调度概述

虚拟机是当前云计算的主要实现形式，也是云计算的核心技术之一，除了虚拟机，容器在云计算中应用越加广泛，容器云发展迅猛。当前容器虚拟技术以 Docker 为典型代表，Docker 的底层实现是 LXC(Linux Container)，LXC 的资源管理完全依赖于内核的控制组 (cgroups)。和传统的虚拟技术不同、LXC 提供的虚拟环境是在操作系统层面实现的，主要面向进程，LXC 提供的虚拟环境也就是容器，操作系统可以为容器分配各种 CPU 时间、I/O 时间、内存等，并提供单独的命名空间。其隔离性主要依赖于 Linux 内核的 namespace 特性，命名空间让进程之间彼此隔离，

这种既能与宿主机共享资源又能同时提供用户隔离的虚拟化方案迅速受到人们关注。容器虚拟化技术可以实现较小的虚拟化开销，应用可以实现快速的部署、交付以及较好的跨平台性。以 Docker 为基础构建的 CaaS(Container as a Service) 应运而生[9]，各大互联网公司投入巨资进行研发，根据 451 Research 预测，容器作为一种高速成长型的工具，年增幅高达 40%。容器将作为应用最为广泛的云工具，超过 OpenStack、PaaS 以及其他相关的产品，根据其预测，应用容器将从 2016 年的 7.62 亿美元增长到 2020 的 27 亿美元，其预测是根据 125 家应用容器厂商为基础做出的。容器的管理和调度市场也在进行快速的组合并购，Apprenda 收购 Kubernetes 支持者 Kismatic，思科收购 Docker Swarm 支持者 ContainerX 等，这些活动都加速了容器云的飞速发展，当前较为出色的有 Google Container Engine、SAE、Cloud Foundry、AWS ECS、Red Hat OpenShift 等。

容器云和以虚拟机与基础构建的云计算系统一样需要一个性能强大的容器编排管理器进行容器调度、创建、销毁、监控、重启、错误恢复以及服务组合等工作。容器调度器既是实现容器推广的重要推力也是决定集群的性能和资源利用率的关键因素。当前主要有三大主流的容器编排引擎[10]: Docker Swarm、Apache Mesos 以及 Google Kubernetes，其中应用最为广泛的要属开源轻量，性能强大的 Kubernetes。在调度算法上 Swarm 主要包括最少容器、最多容器和随机调度三种；Memsos 更多使用传统的虚拟机调度方法，如 DRF(Dominant Resource Fairness) 实现资源分配，侧重于负载均衡；Kubernetes 使用两阶段过滤评分选取最大评分的节点实现资源调度。几种调度方式和算法各有优缺点，用户可以根据自己的需求不同选取相应的容器调度框架构建其容器云计算环境，实现资源更好的分配和调度，本文主要对 Kubernetes 的调度方式进行研究，在大数据处理多种计算框架应用部署的提升集群资源利用率和负载均衡，使应用的执行时间更短。

## 1.3　论文主要工作和结构安排

### 1.3.1　论文的主要工作

基于 Docker 容器虚拟化技术的 PasS 层 OpenShift 容器云平台使用 Kubernetes 进行容器管理和调度，该调度方式通过预选和优选两阶段选取评分最优的节点作为容器调度的目标，在优选阶段仅考虑内存和 CPU 的影响因素。本文针对其调度方式造成的资源利用率较低和负载不均衡的缺点，在大数据存储与处理容器云平台 Paladin 上，设计和实现了一个新的调度方案，本文主要工作如下:

(1) 基于开源的 OpenShift Origin 和实验室项目 Paladin Storage 构建大数据存储与处理容器云平台 Paladin，在该平台上开发 Hadoop、Spark、MPI、Storm 等

十多种分布式处理框架，用户可以快速构建大数据处理框架和容器伸缩。

(2) 深入研究 Kubernetes 调度核心技术，针对其调度算法不足，提出了一种基于多维资源空闲率权重的评价函数和调度方法 MRWS(Multidimensional Resource Weights Scheduling)。该方法综合考虑物理节点 CPU、内存、磁盘、网络带宽空闲率和已部署的容器应用个数等因素影响，使用模糊层次分析法 FAHP(Fuzzy Analytic Hierarchy Process) 对集群资源自动建模并求解容器应用多维资源权重参数，选取最大评分节点进行容器调度。

(3) 针对 MRWS 调度策略，在容器云仿真平台 ContainerCloudSim 上进行大规模调度仿真，并与 Random、FirstFit、Kubernetes 默认调度策略在资源利用率和负载均衡性方面进行对比。

(4) 在 Paladin 上设计并实现 MRWS 调度策略，针对多计算框架下大数据处理场景，比较 MRWS 与其他调度策略下的集群性能。新的调度策略无论在单计算框架容器应用还是多计算框架容器应用混合执行效率都有极大的提升。

### 1.3.2　本文结构安排

本文总共分为六个章节，第一章主要阐述传统虚拟机技术构建的云计算和 Docker 容器技术构建的容器云的区别，云计算底层虚拟化技术的基本架构、典型的云计算服务和云计算中资源调度方法。接着介绍容器云底层的容器虚拟化技术支撑，代表性的容器云服务以及三大主流的容器编排器，进而引出 Kubernetes 容器编排器的不足和本文需要研究解决的问题。第二章首先对比 Docker 容器虚拟化技术和传统虚拟机技术以及 Docker 构建的容器云平台，接着比较分析容器云中三种主要的调度系统，最后针对 Kubernetes 容器编排管理器的组织架构、调度流程和原理以及其调度的不足进行深入分析。第三章针对 Kubernetes 调度器不足，提出一种综合考虑容器应用和集群物理资源的特点全新的调度方案。新方案主要包括其调度流程改进、多维资源建模、反馈器的设计以构造评价函数。第四章使用 FAHP 方法解决法解决新的调度方案中多维资源权重的问题，对容器应用和集群资源进行数学建模、自动构满足一致性要求的建模糊成对比矩阵和判断矩阵，自动求解应用资源权重参数。第五章首先在 ContainerCloudSim 容器云仿真平台上进行大规模调度仿真，对比分析 MRWS 和 Kubernetes 默认调度方案、FirstFit 以及 Random 调度方法在集群资源利用率、负载均衡性方面的性能。然后在容器云平台 Paladin 上开发部署十几种分布式计算框架，设计和实现 MRWS 调度方法，在大数据处理的多计算框架场景下进行性能测试。最后一章对本文工作进行总结，展望未来的研究方向。

# 第 2 章　OpenShift 容器云平台调度系统

本章从 Docker 虚拟化技术出发，介绍在 Docker 基础上构建的三种典型集群管理系统: 一体式调度系统、两层调度系统和共享状态调度系统。分析了在 Docker 和容器编排管理器 Kubernetes 上构建的开源容器云平台 OpenShift 架构，其底层 Kubernetes 容器调度器的核心组件和调度原理。

## 2.1　Docker 虚拟化与 OpenShift 平台

### 2.1.1　Docker 虚拟化技术

虚拟机是云计算的核心技术之一，而以 Docker 为代表的容器虚拟化技术近几年大有取代虚拟机之势，逐步成为一种主流的技术。Docker 是一种操作系统层面的虚拟化技术，其底层以 LXC(Linux Container) 作为支撑。和传统的虚拟技术面向操作系统或虚拟硬件不同，Docker 是面向进程提供虚拟运行环境，其提供的虚拟环境就是容器。操作系统 Linux 可以为容器分配资源，如 CPU 时间、I/O 时间、内存、外设访问控制等，并通过内核控制组 (cgroups) 子系统限定特定进程使用资源的量，然后使用 Linux 内核的 namespace 隔离容器间的进程。这样就可以实现一个高级的容器引擎，开发者可以快速构建、部署和发布应用，并且应用具有较好的跨平台性。从资源管理角度而言，Docker 依赖于 LXC、LXC 基于 cgroups 子系统,Docker 主要是对容器进行封装，管理容器的生命周期、查询和控制相关信息、而所有与操作系统的交互都是通过 libcontainer 容器引擎完成。

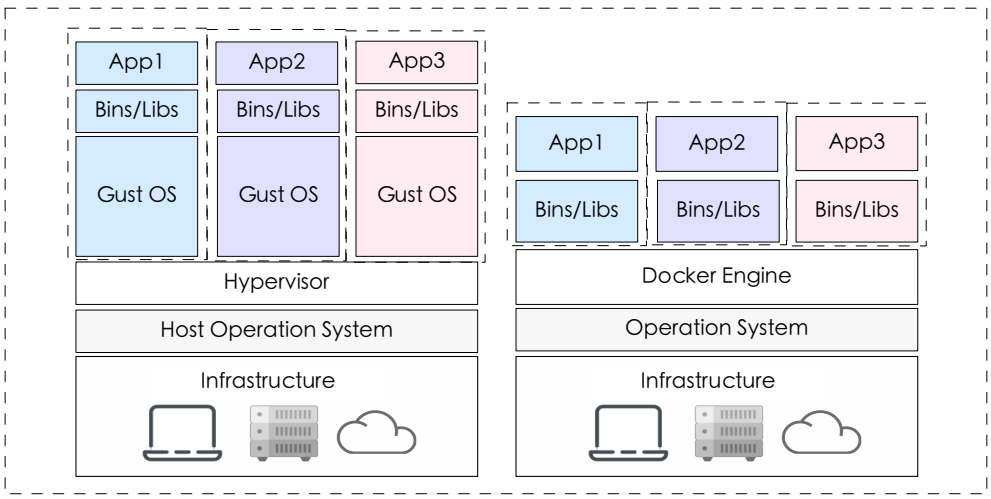



图 2.1　容器与虚拟机对比

图 2-1 中基础设施 Infrastructure 可以是个人电脑、服务器、云主机等，主机操作系统是运行在基础设施上的系统，主要是 Linux 的各种版本。虚拟机管理系统 (Hypervisor) 可以实现在主机操作系统上独立运行多个子操作系统，在子操作系统上安装完应用所需的各种依赖后就可以实现应用资源的隔离。相对于虚拟机，Docker 要简便很多，当前所有的 Linux 版本以及 MacOS、Windows 都能运行 Docker，Docker Engine 取代了 Hypervisor，负责管理 Docker 容器并与操作系统通信，各种应用直接打包到镜像文件中，实现容器间的隔离。

对比 Docker 和虚拟机的架构[11-12] 发现，Docker 直接通过守护进程与操作系统进行通信、容器管理以及资源分配，实现容器与主操作系统的隔离。Docker 没有资源开销较大的子操作系统，各容器直接与主操作系统共享资源，其虚拟化开销大幅缩小，应用启动时间甚至达到毫秒级。用户可以快速构建、部署和交付应用，并且具有较强的跨平台性。虽然 Docker 拥有众多的优势，但其隔离仅仅是在进程层面进行，并不能完全隔离整个运行环境。因此，用户需要根据自己的实际应用场景，在彻底隔离运行环境的需求下选择虚拟机技术，如果仅是应用层面的隔离选择容器技术，如数据库、前端、后端等。

### 2.1.2　OpenShift 容器云平台

Docker 是当前主流容器化技术的代表，Kubernetes 作为现阶段应用最为广泛的容器编排引擎，OpenShift[13] 将这两种主流技术相结合服务于企业，是红帽公司提供的一款开源容器云平台。该平台底层以 Docker 作为容器引擎驱动，Kubernetes 作为容器编排组件，对外提供多种开发语言、中间件、数据库以及极易操作的用户界面、DevOps(Development and Operations) 工具等。允许开发者和开发团队在该平台上进行应用的构建、测试、部署以及发布，是一个完整的容器应用云平台。该平台可以运行和支持有状态和无状态的应用，为容器应用提供较强的安全防护，包括基于用户的访问控制、检查机制以及强制隔离措施。OpenShift 平台还实现了多种综合云原生服务，便于快速智能、灵活开发应用、构建各种分布式系统，支持多种云环境包括 Amazon Web Service、Azure、Google 云平台以及 VMware 等，为开发运维团队提供一个通用的平台和工具，保持持续的开发和测试。OpenShift 分为开源的社区版 OpenShift Origin 和收费的企业版 OpenShift Enterprise，本文实验项目大数据存储与处理容器云平台 Paladdin 服务层是基于 OpenShift Origin 构建的。

从技术堆栈的角度分析，OpenShift 容器云平台自下而上可以划分为基层架构层、容器引擎层、容器编排层、PaaS 服务层、界面及工具层[14]。下面分别对这几个层次进行介绍：

1. 基础架构层。提供 OpenShift 运行所需的基础设施和环境，包括物理机、云主机、虚拟机、各种公有云、私有云以及混合云等。OpenShift 支持多种操作系统，如 CentOS7 以上、Fedora21、Red Hat Enterprise Linux 等，包括在 Linux 基础上专门定制和优化容器化运行环境的操作系统 Atomic Host，该系统可以为应用提供高度一致的运行环境，保证集群的稳定和安全。容器应用虽然具有较强的跨平台性，但其前提是要求底层操作系统的内核和配置必须一致，因为容器的隔离依赖于 Linux 的内核。
2. 容器引擎层。以当前主流的 Docker 作为 OpenShift 容器引擎，Docker 已广泛应用于各种社区和企业生产环境中，经过了安全、稳定和高可用的检验。OpenShift 并未修改任何原生的 Docker 代码，只是将 Docker 的开放性和庞大的镜像资源无缝衔接到平台上，Docker 的普通用户可以快速整合到平台中，所有的应用最终到底层都生成一个 Docker 实例。
3. 容器编排层。容器的编排对容器云的性能和资源利用效率具有决定性作用，OpenShift 最终选择开源轻量的 Kubernetes 作为其容器编排引擎。Kubernetes 已在 Google 内部使用多年，其诞生初衷就是为解决大规模集群中容器的调度和管理问题。OpenShift 平台中很多基本的概念和重要组件如 Namespace、Pod、Replication Controller 等都继承自 Kubernetes。OpenShift 同样只是将 Kubernetes 进行叠加式的整合，并未修改其原生代码和对象，用户依然可以通过原生的命令操作 Kubernetes 的组件和对象。
4. PaaS 服务层。容器云的目的在于对外提供服务，OpenShift 在 PaaS 服务层提供了多种开发语言、框架、数据库以及中间件，极大提升了上层应用的开发、部署和交付速度。OpenShift 有一个专门的社区以及 Docker Hub 提供各种应用的镜像，用户可以快速获取一个应用的基本镜像，构建自己所需的镜像，Red Hat 的 JBoss 中间件几乎全部实现了容器化。
5. 界面及工具层。OpenShift 平台强大的界面及工具极大帮助普通用户高效完成相关应用业务，用户可以通过 Web 进行自助服务，平台将自动从 Docker Hub 中拉取所需的镜像进行应用构建，全自动化的服务极大降低了运维成本和提升服务效率。此外，OpenShift 平台还提供 S2I(Source to Image) 服务，用户开发完成后可以自动整合到镜像中，快速实现交付，提升开发、测试、部署效率。针对用户端接入问题，平台提供 Web 控制台、IDE 集成、命令行工具、以及 RESTful API 编程接口，是一个完善的企业级平台。

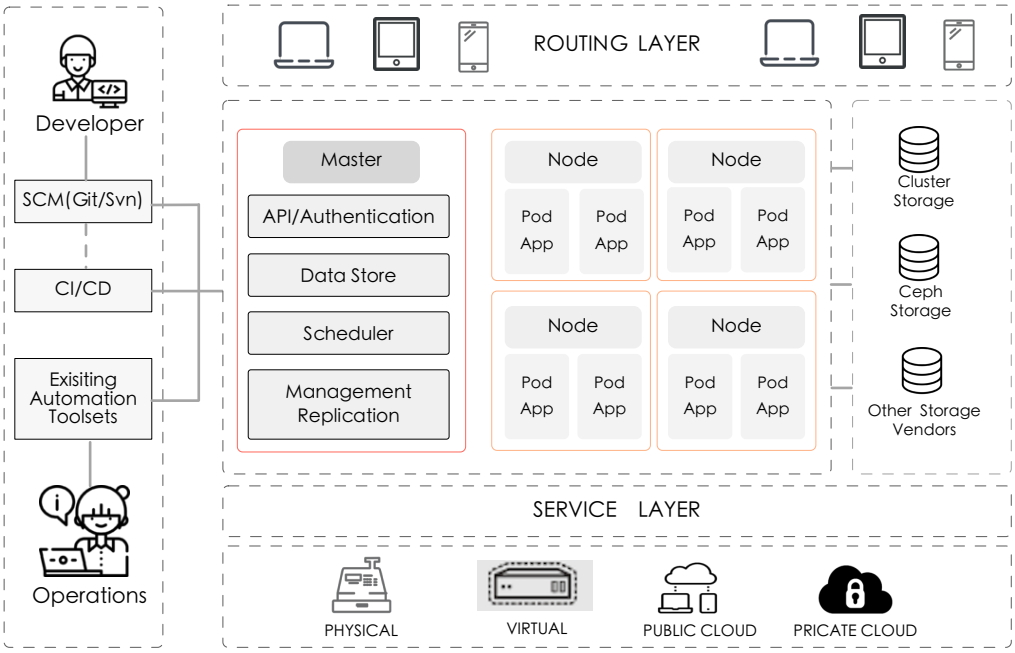



图 2.2　OpenShift Origin 架构图

如图 2.2 所示，OpenShift 平台的核心组件包括 Master、Node、Pod、Scheduler、Service、Storage 等。Mater 是主控节点，可以配置高可用的多个主控节点，负责管理和维护 OpenShift 集群的状态。Master 上运行的 API Server 是其核心组件，所有用户的 Web Console 以及 RESTful API 服务都通过该组件进行访问认证控制，各 Node 节点也会定期轮询 API Server 更新其状态和容器的状态。Data Store 将所有的状态信息存储在分布式的数据库 Etcd 中，并通过 Raft 一致性协议保证其数据的一致性，Etcd 可以安装在主控节点，也可以单独安装到集群之外。Scheduler 调度控制器进行 Pod 资源的分配和调度，收集各节点资源情况，选择最优的节点作为容器应用的调度目标。Replication Controller 异常自检测和恢复组件，负责监控集群中容器应用的状态和数量是否和用户要求一致，通过启动和关闭容器应用满足用户的需求。Node 节点通过接收 Master 节点指令维护容器应用。Pod 是 OpenShift 平台调度器调度的最小单元，一些容器与容器之间往往存在较大的关联性，将几个联系紧密的容器部署在一个 Pod 中进行调度是容器云平台的主要创新点，如分布式数据库多个容器放在一个 Pod。容器是一个非持久化的对象，一旦容器重启或销毁，其状态信息将会随之销毁，集群每次给 Pod 分配的 IP 地址不同，要对外提供统一持久的服务，需要 Service 组件，该组件能将所有的信息转发到其对应的容器 IP 和端口上。此外，还有 Router、Persistent Storage、Registry、Haproxy、Kubelet 等都是集群的重要组成组件。数据的持久化存储可以是集群的数据库、分布式数据库或者其他的数据库中，当前支持的有 NFS、Ceph RDB、GlusterFS 等。

## 2.2　容器集群资源调度系统

Docker 容器虚拟化是容器集群的核心技术，但一个高效强大的容器编排引擎也是集群不可或缺的部分。一个好的调度器既要提现出作业调度的“公平”性，又要兼顾其性能和鲁棒性，还要应用于实际的生产环境中。在集群数据中心，可以通过应用对资源的需求感知用户部署的应用类型，通常可以划分为 CPU 密集、内存密集、I/O 密集和网络带宽密集型应用[15-16]。在集群上通常是多种密集型应用同时部署和运行，调度器如何进行资源分配至关重要，这就使得调度变得异常复杂和困难，往往不存在最优解决方案。传统虚拟机的云计算中心资源调度已有相当多的研究，针对容器集群，各大容器产生也相继推出了几款优秀的容器编排引擎，根据其调度架构的不同可以划分为一体式调度系统、两层调度系统、共享状态调度系统。其典型代表分别是 Docker Swarm、Apache Mesos、Google Kubernetes。

### 2.2.1　一体式调度系统

一体式的调度使用单一的调度节点处理所有的请求，通过固定的调度算法调度所有的作业。这种调度方式导致调度扩展性很差，用户很难灵活定制自己的调度策略，而且所有的调度信息在单节点上进行运算，不能并行执行，单节点成为调度性能瓶颈。Docker 公司 2014 年发布的容器编排管理工具 Swarm[17-18] CLI 中，无需进行安装，拥有活跃的社区，易于搭建并且已应用于实际的生产环境中。

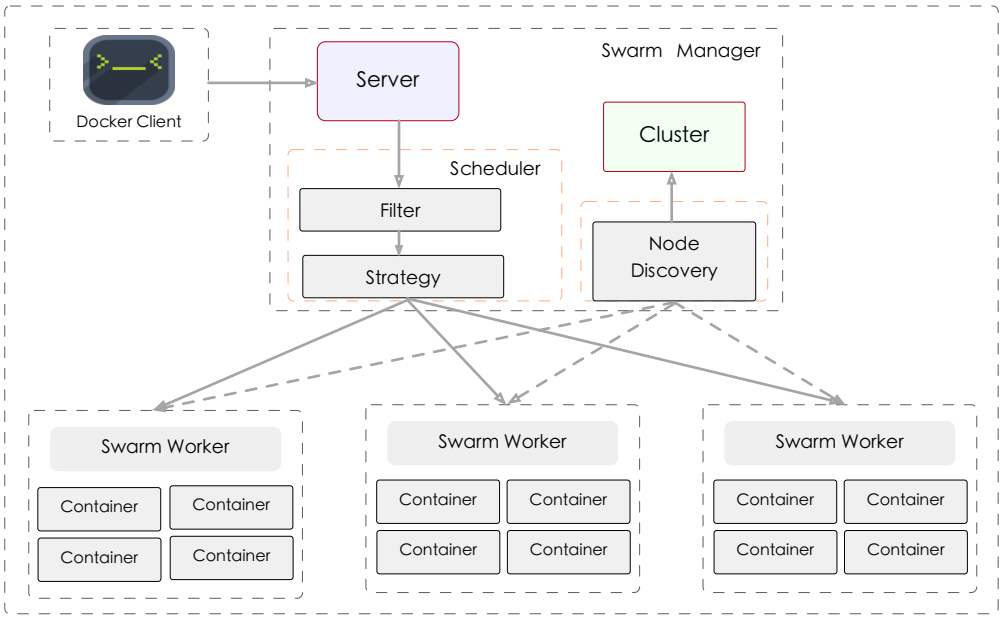



图 2.3　Swarm 架构图

Swarm 集群由管理节点和工作节点构成，其中管理节点可以配置多个，实现

高可用的多管理模式，内部通过 RAFT 算法实现管理节点高可用性。管理节点上除 Docker Daemon 守护进程、Load Balancing、Scaling 组件外，最为重要的是 Scheduler 和 Discovery 组件，其中 Discovery 负责集群中节点的发现和状态更新，Scheduler 首先根据用户的限制对节点进行筛选，然后使用内置的调度策略进行应用调度。工作节点主要运行 Docker Daemon 和 Load Balancing，根据控制节点的指令运行调度过来的容器应用。在调度器的过滤模块主要提供了约束过滤器和健康过滤器，此外还可以配置吸引力过滤器、依赖过滤器和端口过滤器。

约束过滤器是通过一定的约束条件筛选节点，集群中每个节点都带有一个 key-value 标签，对于一些特殊的应用可以指定其 label 进行调度到指定的节点上运行；健康过滤器过滤掉不健康的节点，避免容器调度后运行失败；吸引力过滤器是将新的容器链接到已经创建的容器上，实现共同运行和销毁，主要通过镜像和标签吸引两种方式。镜像吸引是将容器直接调度到拥有该镜像的节点上，避免重复开销镜像下载时间，节约网络资源，标签吸引是通过标签指定链接到已创建的旧容器实现共同工作；依赖过滤器是新容器依赖于其他的容器，可能会共享磁盘卷、或在同一个网络栈上等；端口过滤器将需要特定开发端口的容器运行到开放该端口的节点上，避免容器不可用的情况发生。

Swarm 的调度策略主要包括 Random、Spread 和 Binpack 算法[19]，下面分别对其算法进行简单的介绍:

1. Random 算法。该算法随机从过滤完的节点中选取一个节点进行调度判断，如果该节点满足条件则将容器调度到该节点上，否则随机选取下一个节点直至找到合适的节点调度或返回调度错误信息。
2. Spread 算法是最少容器算法，该算法的初衷是保证容器集群的负载均衡。每次遍历一遍集群中每个节点上运行的容器数量，选择容器数量最少的节点进行容器调度，若该节点不满足条件，则依次从后往前进行调度尝试，直至找到满足条件的节点或返回调度错误。
3. Binpack 算法是最多容器算法，该算法的目的在于最大化利用集群中节点资源，和 Spread 算法相反，每次从集群中选择运行容器数量最多的节点进行容器调度，若满足条件则将容器调度至该节点，否则依次从前往后尝试调度容器到节点，直至找到合适节点或返回错误。

### 2.2.2　两层调度系统

两层调度系统是将资源调度和作业调度分开，资源调度层只负责给计算框架分配所需的资源，具体的作业调度由每个计算框架的调度算法完成。在一些成熟

的处理框架中如 Hadoop、Spark、MPI 等有相对成熟和高效的调度算法，两层调度系统将这些调度算法集成进来，通过一个轻量的资源共方式来控制资源的分配和访问。一旦资源分配给某个计算框架，其他计算框架不能使用该资源，直至该资源被释放，造成集群资源利用效率不高。Apache Mesos[20] 是最为典型的两层调度系统，最初由加州伯克利分校的 AMPLab 开发，后在 Twitter 获得广泛应用。

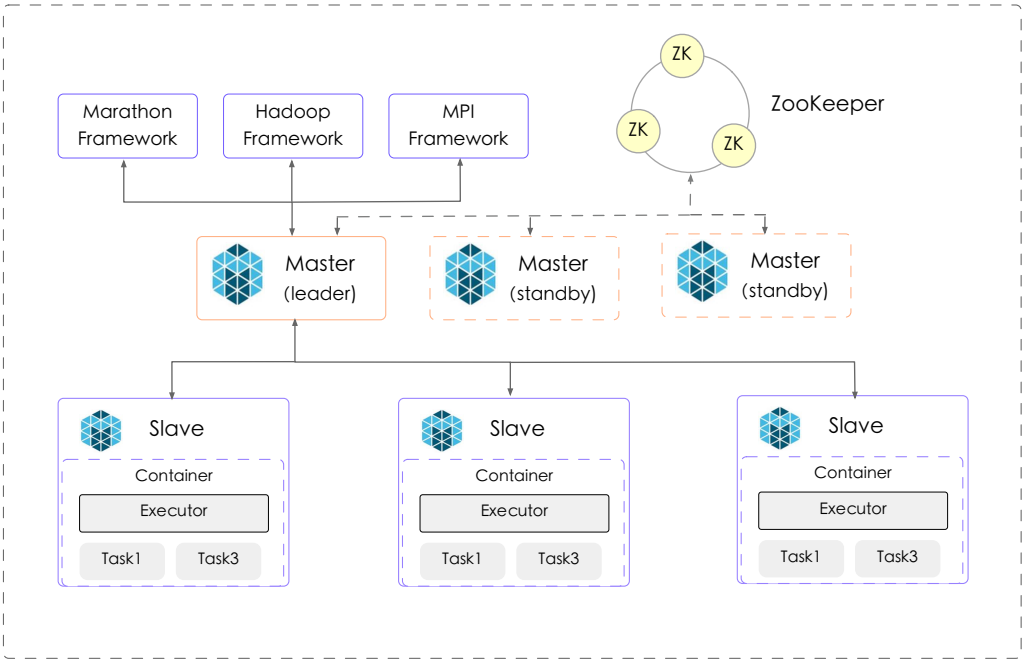



图 2.4　Apache Mesos 架构图

如图 2.4 所示，Mesos 的总体架构也采用主从设计，Master 节点作为集群的管理控制节点，可以有一个或者多个 (最好为单数)，为了防止单节点故障，通过 ZooKeeper 提供一致性服务。每次在多个 Master 中选举一个作为 leader 对外提供服务，其他的 Master 副本随时保证和 Master 的状态一致，一旦服务出现故障，立即进行新的备份选举，从而实现集群的高可用性。集群的架构可以分为两层: 计算框架层和 Master 调度层。下层的 Master 管理众多的计算节点，负责收集各节点的资源情况并作出分配决定，上层的计算框架层负责实际任务的调度，从而将整个调度分为了任务调度层和资源调度层，两个层级互不干扰，分别使用自身的调度算法，提升了集群调度策略的可扩展性。从架构的组成部分来看，主要有五大组成部分，下面分别对组成部分的功能进行介绍:

1. ZooKeeper 组件。ZooKeeper 作为一款 Hadoop 项目中分布式系统的协调系统，主要用于解决分布式应用数据管理问题，如应用配置管理、统一命名服务、状态同步服务以及组服务和集群管理等。ZooKeeper 非常简单易用、功

能丰富可靠、并且提供了通用协议下开源共享的存储库，其核心就是一个精简的文件系统，可以提供一些简单和抽象的操作。在 Mesos 中、ZooKeeper 主要用于解决 Master 节点的状态一致性和高可用问题，避免单节点故障，实现集群持续稳定的对外服务。

2. Mesos Master 组件。Master 是整个集群的控制器，是整个集群调度的核心节点，既需要对底层各 Slave 节点的资源进行收集和管理，也需要通过一定的资源分配策略提供资源给上层的各处理框架 Framework。当前对各 Framework 的资源分配策略为 DRF[21](Dominant Resource Fairness) 算法，这是一种针对多维资源 (CPU、内存、I/O、网络带宽等) 不同需求设计的公平调度算法。Master 还负责资源的访问控制，一旦某个资源分配给了特定的 Framework，该资源必须等框架释放后才能再次进行分配。
3. Mesos Slave 组件。该组件作为调度底层具体的执行者，接收来自 Master 的指令，将自身的资源分配给每个执行器，执行器上运行一个或多个任务，并将各任务作为容器运行起来。此外，Slave 节点还定期向 Master 节点汇报资源使用情况作为 Master 调度器的调度依据。Slave 上还运行一个 containerizer 用来管理容器的生命周期，包括容器的创建、更新、监控和销毁。
4. Framework 组件。Framework 负责将各计算框架如 Hadoop、Spark、MPI 等注册接入到集群中，Master 的调度器负责对其需要的资源进行分配，具体的任务调度则由各计算框架完成。各计算框架通过调用 Master 的 API 进行任务的创建和调度请求，Master 再将任务下发到 Slave 上执行。
5. Executor 组件。Executor 负责启动框架内部的 Task 任务，各种计算框架接入 Mesos 的方式，接口不同，因此要接入一个新的计算框架就需要编写一个新的 Executor，用来通知 Mesos 如何启动框架中的 Task 任务。

Mesos 作为一款优秀的分布式资源管理框架，采用双层调度机制，资源分配层负责将资源分配给计算框架，计算框架使用自身的任务调度器执行任务调度。Mesos 可以对集群的资源进行细粒度的划分，按照计算框架实际任务的需求进行资源分配，极大提升了集群资源利用率。Mesos 不需要清楚各 Framework 的具体调度逻辑，只需要通过 API 向上提供资源分配即可，具有较强可扩展性。Meso 是模块化的实现，新增一个 Framework 不需要对 Mesos 进行重新编码，可以快速实现接入。Master 节点使用 ZooKeeper 保证其状态一致性，能够实现高可用性。但是，Mesos 对底层的资源采用“悲观锁”的方式进行控制，一旦被某个 Framework 占用，必须等到其释放才能进行新的资源分配，其并发性受到极大的限制。独立的调度框架只能访问集群部分状态信息，往往不能进行调度优化。

### 2.2.3　共享状态调度系统

在一体式调度系统中，资源分配和任务调度都由中心调度器进行管理，并且集成了具体的调度算法。在两层调度系统中，资源分配由资源调度层完成，任务调度由具体的计算框架自己完成。一体式的调度很好的保证了全局状态的一致性，但是扩展性较差，两层调度系统虽然可扩展性较好，但是集群状态的一致性较难保证，并且容易造成资源的竞争和死锁。为了解决这些不足，共享状态调度系统被提了出来，其核心在于所有的调度逻辑共享集群状态，选择最优的节点进行资源分配和任务调度。其中最为典型就是 Google 推出的 Borg[22]、Kubernetes 以及使用事务方式解决一致性管理问题的 Omega[23]，其中 Kubernetes 以其轻量开源的特点深受大家喜爱，各大主流的互联公司都加大了对其支持力度。

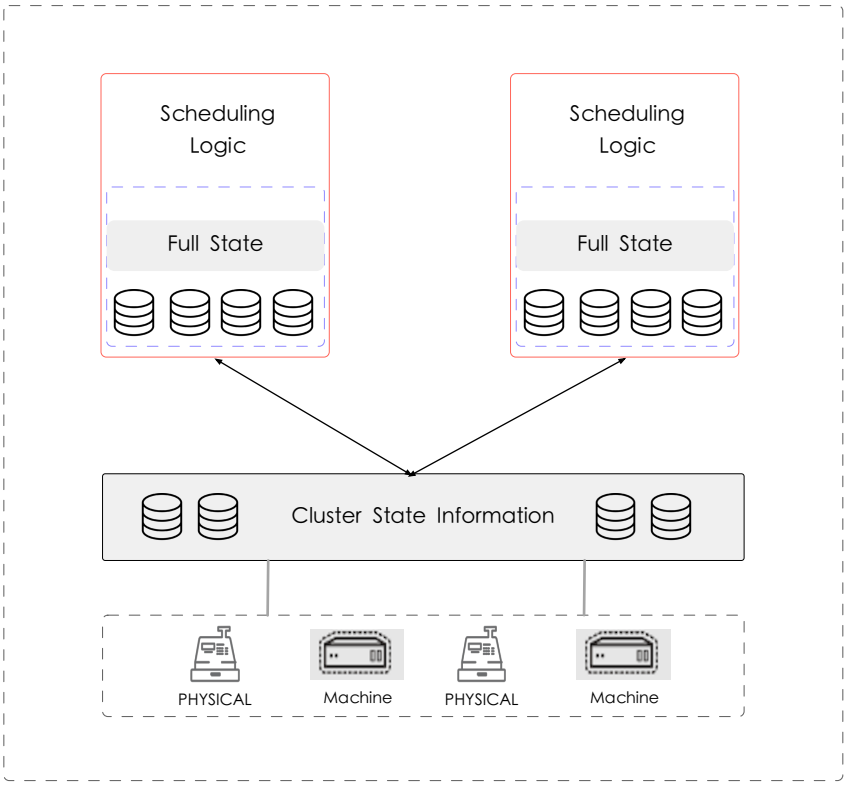



图 2.5　共享状态资源调度系统

在共享状态调度系统中，每一个上层的调度器都可以对整个集群的状态进行访问，资源对所有的调度器都是透明的，可以实现自由竞争，不存在单一的资源分配器，因此也不存在单一节点访问瓶颈。集群不仅支持节点资源的快速增加，也支持调度器的扩展，用户根据自己的实际应用场景开发特殊的调度器，可以很轻易的集成到集群中。为了保证集群的高并发性和可扩展性，共享状态调度系统采用“乐观锁”[24] 对底层资源进行并发控制。具体的实现是通过对集群的所有状态都

增加一个版本属性，每次提交时比较提交的版本和当前版本号大小，若版本号小于当前的版本号，则不进行任何处理，只有提交版本号大于当前数据版本号的操作才被接受，更新完状态后版本号递增。“乐观锁”并发控制虽然会增加资源访问的冲突数，影响系统的吞吐率，但在实际的系统中依然在一个可以接受范围，下一个小节将对 Kubernetes 调度系统架构和流程做详细的分析。

## 2.3　Kubernetes 资源调度系统

Kubernetes 是典型的基于共享状态的调度系统，是一个轻量的开源平台，用于容器化应用管理和服务，使用 Label 和 Pod 的概念将容器划分为逻辑单元，实现相关容器的共同调度和部署。本小节针对其核心组件和整体架构进行深入的分析，尤其是对其调度算法和不足进行研究，为下一步提出新的调度方案提供依据。

### 2.3.1　Kubernetes 简介

Kubernetes 源自 Google 的 Borg 项目，Borg 是 Google 集群管理工具，稳定地管理全球上百万台服务器多年。为了在容器云竞争中占据领导地位，Google 基于 Borg 的管理经验，研发了基于 Docker 的容器编排工具 Kubernetes，并在 2014 年将其开源，逐步发展成为一个大的生态技术圈。作为一个跨主机的应用容器编排引擎，Kubernetes 提供了一系列强大的功能，包括应用容器部署、资源调度、服务发现、动态扩容以及错误恢复等。Kubernetes 同时以强大的集群管理能力用于支撑分布式系统，实现了多租户应用、服务发现、服务注册、负载均衡、故障自处理和恢复、在线扩容、细粒度调度、资源配额管理等功能，完美定义了构建业务系统的标准化架构。除集群管理方面的强大功能外，Kubernetes 还提供完整的开发、测试、部署、运维监控等各种开发管理工具。

Kubernetes 逐步发展成一个巨大生态圈，为容器的编排提供一个简单、轻量的方式，最重要的是用户可以实现功能定制。当前采用 Kubernetes 的云计算服务商和用户越来越多，如微软、Yahoo、IBM、华为、VMware、网易、阿里、华为等，还有一些初创公司灵雀云、青云等都采用 Kubernetes 作为容器云的管理系统。

Kubernetes 拥有强大而活跃的社区，众多开发者不断对其进行迭代更新，其代码更加完善。当前 Kubernetes 社区的支持者有 Google、CoreOS、RedHat、华为、网易、阿里云、浙大 SEL 实验等。Google 在 2015 年联合其他 20 多家公司成立了开源组织 CNCF(Cloud Native Computing Foundation)，加入 OpenStack 社区，致力于 Kubernetes 的应用推广，支持其在当前公有云、私有云等更多基础设施平台上应用，提供更加简便丰富的工具集，更好的服务用户。

### 2.3.2　Kubernetes 架构和组件

Kubernetes 是一个主从架构体系，该集群管理器很好的解决了扩容和升级两大难题，具有较强的横向扩展能力，下面是 Kubernetes 的整体架构图[25]。

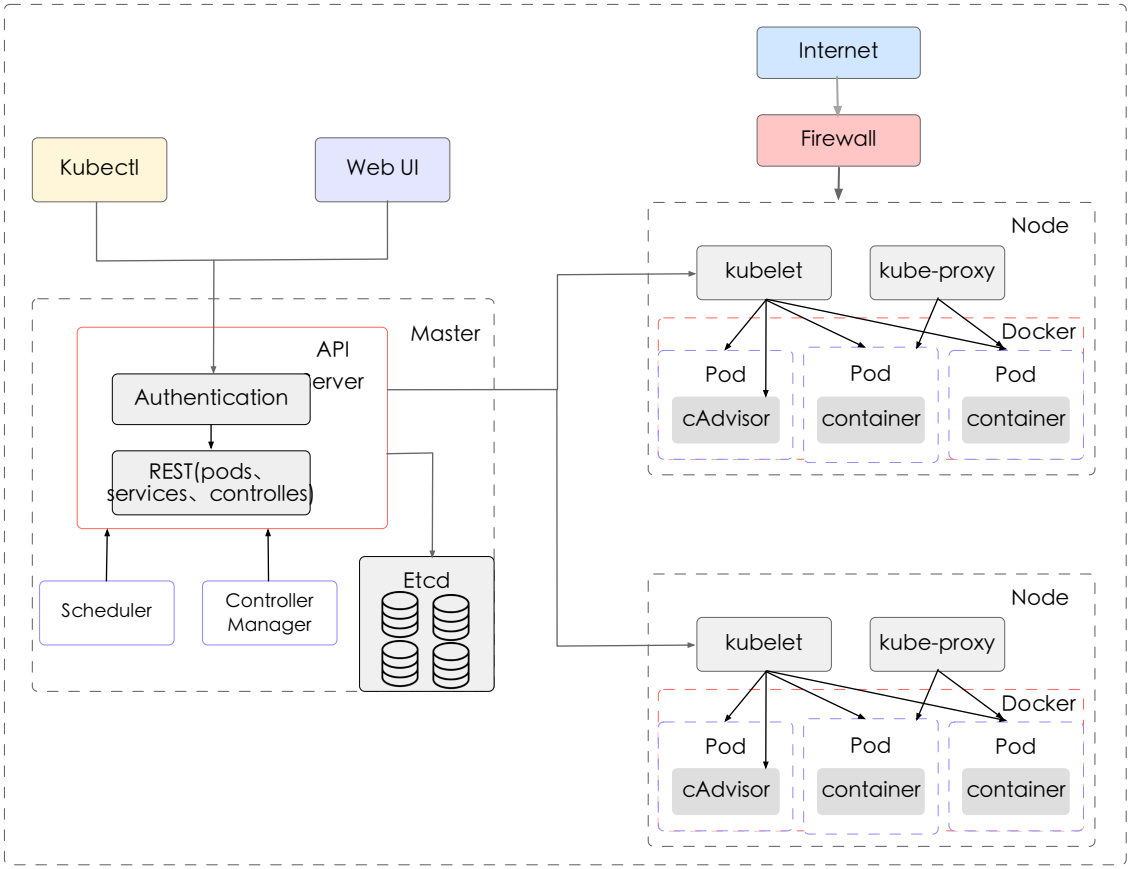



图 2.6　Kubernetes 体系架构图

如图 2.6 所示，Kubernetes 主从架构主要由控制节点 Master、工作节点 Node 以及外部工具集 kubectl、web UI 等附加依赖组成。Master 作为集群的控制节点，主要负责集群的管理调度，由 API Server、Scheduler、Etcd、Controller Manager 等组成，实现其主要功能。Node 节点主要根据控制节点的指令执行具体的任务，实现应用容器的运行，主要由 kubelet、kube-proxy、cAdvisor、Container Runtime 等组成。外部可以通过 kubectl 命令工具对集群进行增删查改的操作，也可以使用 Web UI 与集群进行交互。下面对集群中的重要组件分别进行介绍:

1. API Server 组件。API Server 是系统管理指令的统一入口，负责对外提供 RESTful 的 API 服务功能，所有对集群的操作都需要通过 API Server 组件进行交互，是集群外部和内部的通信枢纽中心，同时也是资源配额限制的入口，通过 Authentication 提供集群完备的安全认证机制。API Server 接收外部 Web

browser 或 kubectl 的命令请求，将 REST 对象和状态持久化到 Etcd 中存储，同时和 Node 节点上的 kubelet 进行交互。

2. Scheduler 组件。集群调度组件，负责集群资源调度和 Pod 分配工作，Scheduler 监控集群中未分配的 Pod，根据其对资源的约束条件和集群资源可用性，将 Pod 调度到实际的 Node 上运行。这是一个可插拔的模块，用户可以开发自己的调度器集成到集群中，其调度流程和调度策略后面会详细的介绍。
3. Controller Manager 组件。控制管理器提供服务发现、集群管理、Pod 扩容、服务绑定、应用生命周期管理等功能。其中 Node Controller 用于管理节点、Replication Controller 用于应用容器管理，保证容器副本和需求一致、Namespace Controller 用于命名空间管理、Service Controller 提供负载和服务代理、Persistent Controller 管理维护 Persistent Volume 和 Persistent Volume Claim 等。
4. Etcd 组件。一个高可用、强一致性的服务发现键值存储仓库，用于保存集群中所有的网络配置和对象的状态信息，是一个中心数据库的地位，进行分布式的部署，通过 watch 机制进行服务更新支持。
5. Kubelet 组件。Kubelet 是运行在 Node 节点上的控制器，用于裁决和驱动容器的执行层，是 API Server 和 Pod 的主要实现者。单个 Pod 中可以运行多个容器和存储数据卷，能够将 Pod 和相关的依赖项很方便地打包迁移，API Server 进行访问控制，Scheduler 进行资源的调度，但是最终 Pod 能否在 Node 上运行成功是由 kubelet 决定的。此外，kubelet 通过 cAdvisor 组件对 Node 节点的状态、资源进行监控，定期汇报给控制节点，存储在 Etcd 中。
6. Kube-Proxy 组件。负责负载均衡和反向代理组件，通过创建 Pod 的代理服务，用户可以通过 IP 地址直接访问 Pod 应用，实现服务到 Pod 的路由和转发。此外，kube-proxy 还实现了一个高可用的负载均衡解决方案，

除上述给出的核心组件外，Kubernetes 还有负责提供集群 DNS 服务的 kube-dns、提供外网访问入口的 Ingress Controller、提供资源监控的 Heapster、提供管理控制界面的 Dashboard、提供日志采集、存储和查询的 Fluentd-elasticsearch 组件等。

### 2.3.3　Kubernetes 调度流程

Kubernetes 调度器运行在 Master 节点上，作为一个可插拔的模块，在默认配置下，调度器可以满足大部分的需求，如特定的 Pod 分配到到指定的节点，相同集合下的 Pod 分配到不同节点，平衡各节点的资源使用率等。

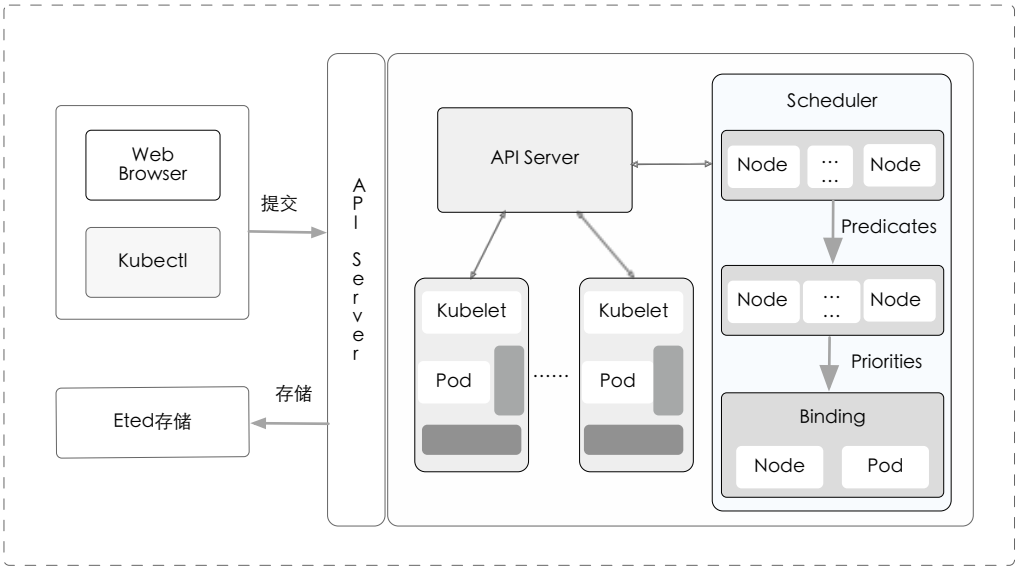



图 2.7　kubernetes 调度流程

调度器相对于普通用户而言类似一个黑盒，输入 Pod 的资源需求，输出 Pod 和节点的绑定，起到指挥中枢的作用。但是在很多业务场景下，用户希望自己的 Pod 调度可控，如制定调度算法、特殊的硬件需求的 Pod 调度到特殊的节点、通信量大的计算框架部署在相同机架上、数据需求大的 Pod 部署在存储数据的节点上等。调度器的调度流程如图 2.7 所示，命令工具 Kubectl 或 Web Browser 向 Master 上的 API Server 发送调度请求如创建 Pod，API Server 对请求做出响应处理并将处理的结果存储在 Etcd 中，同时设置 PodSpec.NodeName 为空，加入未调度 Pod 队列。调度器监控 Etcd 中未调度 Pod 队列状态，发现有未调度的 Pod 时通过调度策略尝试绑定 Pod 到节点，调度策略分为两个阶段: 预选阶段和优选阶段。预选阶段主要过滤节点，筛选满足 Pod 资源需求的节点如 CPU、内存是否满足需求，端口是否冲突以及其他特定需求等，淘汰不满足需求的节点。优选阶段将满足资源需求的节点进行综合评分，主要根据资源使用的均衡性、相同副本的容灾性等调度策略，最终选取得分最高的节点，将 Pod 调度到该节点上，并将绑定状态存储到 Etcd 中。节点上的 Kubelet 监控 Etcd 中 Pod 调度结果，接管调度的后续工作，负责 Pod 生命周期的管理，一个完整的调度结束。

## 2.4　本章小结

本章从 Docker 的基本概念出发，简要阐述 Docker 虚拟化技术底层实现的部分原理，详细对比 Docker 虚拟化和虚拟机的区别以及两种技术的优缺点。接着详细介绍了在当前流行的容器虚拟化技术 Docker 和容器编排引擎 Kubernetes 基础上构

建的 Openshfit Origin 容器云平台的技术架构，对其重要的技术层次和核心组件进行分析，从而引发对容器编排技术的讨论。针对当前流行的三种容器编排引擎架构一体式调度、两层调度和共享状态调度进行介绍和分析，分别以 Swarm、Mesos 和 Kubernetes 为例进行深入的分析。Swarm 和 Mesos 讨论了其核心组件和整体技术架构以及其调度架构的优缺点。由于 OpenShift Origin 平台采用 Kubernetes 作为容器编排引擎，因此，针对 Kubernetes 除进行核心组件介绍外，还深入剖析了其调度流程。为下一章在 OpenShift Origin 容器应用平台上提出针对多计算框架的调度方案奠定基础。

# 第 3 章　改进 OpenShift Origin 平台调度策略

Kubernetes 作为 OpenShift Origin 容器云平台的容器编排引擎，其默认配置的调度算法虽能满足大部分用户需求，但其算法较为简单，集群资源利用率较低，无法满足用户的在特定场景下的调度需求。本章在深入分析默认调度算法不足后，提出了一种新的基于多维空闲资源权重参数的评价函数和调度方案 MRWS (Multidimensional Resource Weights Scheduling)。

## 3.1　优化 Kubernetes 调度流程

Kubernetes 调度算法分为预选和优选两阶段，在预选阶段过滤掉不满足需求的节点，优选阶段对剩余节点评分，选择评分最高的节点作为调度目标。整个调度器可以分为待调度的 Pod 列表、满足条件的 Node 列表以及调度策略三部分。用户可以对其调度流程进行一定的优化，开发特定场景下的调度算法。

### 3.1.1　Default 调度算法流程

在 Kubernetes 默认调度流程中，预选阶段解决节点过滤问题，优选阶段解决节点选择问题。用户可以对两个阶段进行简单的配置，调度器将根据用户指定的预选和优选规则进行过滤和评分计算，最终输出满足条件的 Node 和 Pod 绑定策略，将 Pod 调度到 Node 节点上，针对两阶段的规则，下面做详细的介绍。

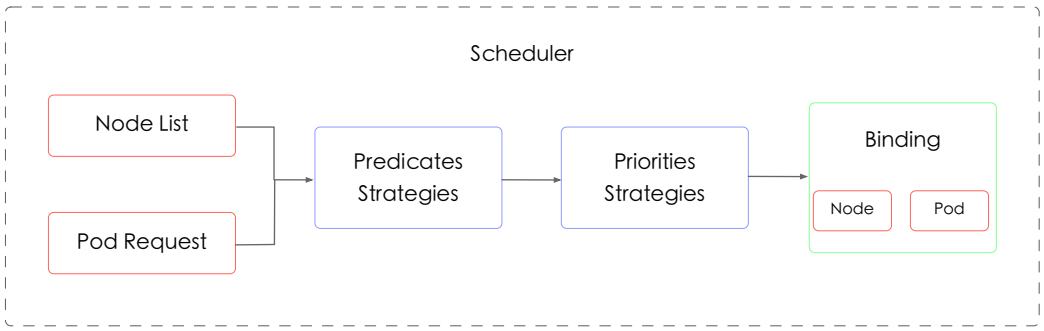



图 3.1　Default 调度算法流程

Predicates 阶段过滤掉不满足 Pod 资源需求的节点，依据一定规进行筛选，包括磁盘卷冲突、端口冲突、资源容量、节点检测、服务占用、亲和性等。1.7 版本的筛选条件主要如下，新的版本 Kubernetes 正在不断完善和更新：

(1) NoDiskConflict：检测卷冲突，在当前的规则中，两个不同 Pod 不能使用相同的卷。因此，一旦 Node 上挂载了该卷并被某个 Pod 使用，则新的 Pod 不能再调度到该 Node 上，否则造成卷冲突。不同的系统对该冲突检测范围不同，如 Google Compute Engine 在只读模式下允许多个卷，Ceph RDB 允许两个 Pod 分享部分资源，Amazon EBS 禁止两个 Pod 挂载同一个卷。

(2) NoVolumeZoneConflict：在给定 Zone 限制的条件下，检测 Node 上部署的 Pod 是否存在卷冲突，当前只限定对 PV(Persistant Volume) 范围进行支持。

(3) PodFitsHostPorts：端口冲突和服务占用检测，需要调度的 Pod 内所有的容器需要的端口在 Node 上是否被其他 Pod 占用。

(4) PodFitsResources：根据 Pod 资源的需求检测节点空闲资源量是否满足其运行需求，主要检测 CPU、内存、磁盘等资源。Kubernetes 的调度是静态调度，资源的判断是根据分配的资源量而不是资源的实际使用量。

(5) HostName：检测 Pod 是否指定了 Node 节点，所有不在指定 Node 集合内的节点都将被过滤掉。

(6) MatchNodeSelector：检测 Pod 是否指定了 MatchNodeSelector 属性，若指定了该属性，Node 的 Label 必须和该属性匹配。

(7) MaxEBSVolumeCount：确保挂载的 EBS 存储卷总合不超过设置最大值，调度器计算每个 Node 上直接或间接使用的全部卷总合，一旦超过最大值，Pod 不能调度到该节点上。

(8) CheckNodeMemoryPressure：判断节点是否存在内存压力，若存在内存压力则标记为 1，Pod 只能调度到内存标记为 0 的节点上。

(9) CheckNodeDiskPressure：判断节点是否存在磁盘压力，若存在磁盘压力则标记为 1，Pod 只能调度到磁盘标记为 0 节点。

(10) MatchInterPodAffinity：节点的亲和性过滤，检测 Pod 是否和已部署的 Pod 存在亲和性，将有亲和性的 Pod 调度到相同节点。

(11) PodToleratesNodeTaints：判断将 Pod 调度到节点后是否满足节点容忍的条件，相同的 Pod 副本为了满足容灾性，一般部署到不同的 Node 甚至是不同的机架和数据中心。

此外、还有专为 Google Compute Engine 和 Amazon EBS 配置的 MaxGCEPDVolumeCount、MaxAzureDiskVolumeCount 卷检测条件，检测节点上挂载的卷容量总和是否超过了设定的最大值。

Priorities 阶段根据一定的评分算法对预选出来的节点列表进行综合评分，评分依据主要包括资源空闲量、资源消耗平衡性、Pod 亲和性、Pod 与节点的匹配度

等。Kubernetes 使用优先函数集合内的函数对节点进行 0-10 评分，最终计算评分总和，评分越高表明 Pod 调度到该节点越适合，同时还可以为集合内的每一个函数设置一个权重值，主要的评分函数如下:

(1) LeastRequestedPriority：、根据资源空闲率计算节点得分，即节点的空闲资源与节点资源总量的比值 ((节点资源总量-节点上 Pod 的资源和-新 Pod 的需求)/总容量) 来计算。设置 CPU 和内存具有相同权重，都设置为 0.5，资源空闲率越大，剩余资源越多，节点的评分就越高。

(2) BalancedResourceAllocation：该函数必须联合 LeastRequestedPriority 一起使用，用于平衡各项资源的使用率，资源使用越均衡，节点的评分越高。Kubernetes 主要对内存和 CPU 资源消耗进行平衡，由两者的“距离”决定分值大小，即使用 10-abs(CPU 空闲率-内存空闲率)*10 进行计算。

(3) SelectorSpreadPriority：为了更好应对容灾和宕机风险，同属 Service、Replication Controller 下的 Pod 尽可能分散部署到不同的 Node 上。对于指定 Zone 的 Pod，也要尽量分散到不同区域的主机。调度器统计各 Node 上相同 Service、Replication Controller 下的 Pod，Node 上 Pod 数量越少，评分越高。

(4) NodeAffinityPriority：设置调度器的亲和性机制，主要对节点进行精确匹配。有两种选择器匹配模式，选择器“hard" 模式下设置的 NodeSelector 必须和节点的 Label 匹配，保证选择的 Node 是完全满足需求规则的，”soft" 模式下不能完全保证百分百的匹配，但会尽量满足规则要求

(5) ImageLocalityPriority：为避免镜像的重复下载对网络和磁盘资源的重复消耗，该函数根据 Pod 中运行容器需要的镜像对节点进行评分，节点上存在的镜像越多，该节点评分越高。若该节点上没有所需的镜像，评分为 0，镜像评分的权重可以根据镜像大小按比例决定。

(6) MostRequestedPriority：和 LeastRequestedPriority 相反，两者使用一个来计算评分。该函数确保使用资源越多的节点，评分越高，目的在于使用更少的服务器提供服务，节约集群资源，提升资源利用率。

(7) TaintTolerationPriority：容忍性评分函数，匹配 Pod 的 TolerationList 与节点 Taint，匹配项越多，该节点的容忍性越好，从而评分越高。

(8) InterPodAffinityPriority：用于迭代 WeightedPodAffinityTerm 的元素计算和，若该节点同时满足亲和性设置，则将该评分加入节点的整体评分中。

(9) NodePreferAvoidPodsPriority：根据节点是否设置 Anotation 属性进行评分，没有设置该属性则评分为 10，设置该属性并且调度的 Pod 正好是副本，则该节点评分为 0，目的在于避免相同副本调度到同一节点。

优先调度模块定义了一个评分函数集合，各评分函数的权重可以指定，用户根据调度依据的重要程度给各函数赋予不同的权重值，最终所有函数的评分总和就是该节点的最终得分，选择评分最高的节点作为 Pod 的调度目标。

### 3.1.2　MRWS 算法调度流程

Kubernetes 调度系统的 Default 算法核心在于优选阶段的评分函数，调度策略根据节点评分高低做出调度决策。在所有的评分函数中，除一些特殊的调度规则外，如节点亲和性、节点容忍度、节点上的镜像文件等，最为重要的是根据资源空闲率做出评分的 LeastRequestedPriority 和资源使用平衡性做出评分的 BalancedResourceAllocation 函数。这两个函数是整个评分函数集合中的核心，在其他外在规则相同的条件下，直接决定节点的评分高低。但是，这两个评分函数仅考虑了内存和 CPU 的空闲率和消耗平衡性，这种评分会造成节点其他维度资源消耗不均，如磁盘、网络带宽、节点上运行的 Pod 数量等。一旦某个维度的资源过载，节点将不能部署更多的 Pod，造成集群资源利用率低下。因此，新的调度算法 MRWS 将综合考虑集群 CPU、内存、磁盘、网络带宽以及节点运行 Pod 数量的均衡性，提升集群的负载均衡和资源利用率。

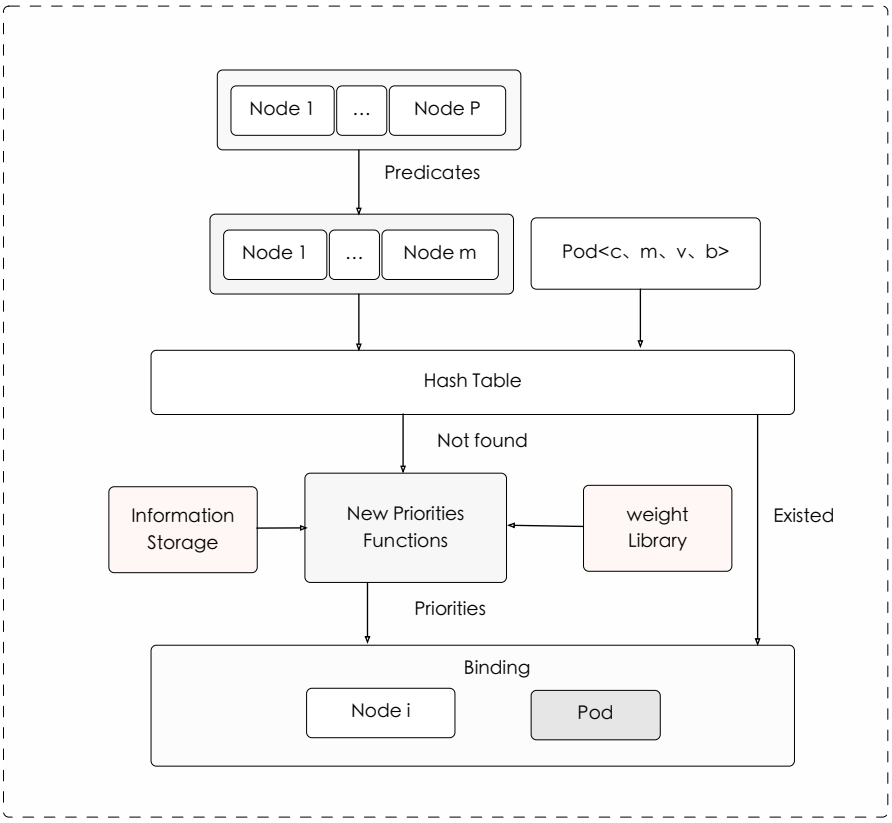



图 3.2　MRWS 算法调度流程

在 Default 算法中，优选阶段根据评分集合进行节点评分，其评分函数如下：

$$S_i = (w_1 * f_1) + (w_2 * f_2) + ... + (w_{jj} * f_j) + ...(w_p * f_p) \tag{3-1}$$

其中，$S_i$ 是第 i 个节点的总评分，$w_j$ 是第 j 个评分函数权重，$f_j$ 是第 j 个评分函数，每个函数的权重可以根据用户的需求进行指定。

如图 3.2 所示，新的调度算法 MRWS 首先对调度流程进行优化，预选阶段从集群上的 $P$ 个节点中根据预选规则选出 $M$ 个节点，这些节点满足 Pod 调度的资源需求和预先设定的筛选条件。$M$ 个节点列表和 Pod 的内存、CPU、磁盘、带宽需求作为评分阶段的输入，查找 Hash 表中该 Pod 以前是否被创建和调度过，若该 Pod 之前被调度过并且以前调度的节点同时在预选阶段剩余的节点中，将新 Pod 直接绑定到节点。通过 Hash 表的作用，可以避免重复的 Pod 被调度到不同的节点，导致容器镜像被重复下载，节约磁盘空间和网络资源，缩短 Pod 应用的调度算法计算时间，使集群具有调度记忆功能。若 Pod 是一个新建的 Pod，进入函数评分模块，信息存储模块通过节点上的 cAdvisor 资源监控代理收集集群整个资源，同时记录各节点上已分配的 Pod 资源总量和 Pod 运行数量，权重参数库存储容器应用资源权重系数。最终通过全新的评分函数体系进行节点评分，该评分函数核心在于对节点资源建模，通过数学方法 FAHP 自动求解其参数，使集群中各节点各维度资源利用更加均衡。

## 3.2　MRWS 调度算法设计

Kubernetes 调度器评分阶段的核心算法仅考虑内存和 CPU 空闲率以及消耗平衡性，容易造成其他维度资源过载，从而使集群资源利用率低下。MRWS 算法将综合考虑集群节点的 CPU、内存、磁盘、网络带宽以及节点上已运行的 Pod 数量等因素，致力于多维资源的均衡利用，尤其是在大数据处理多计算框架同时部署的场景下，让各计算框架的应用调度更加合理，缩短应用的执行时间。整个 MRWS 算法分为资源空闲率评分模块，平衡评分模块和算法均衡度评价模块三部分，下面分别对其建模和计算方法进行详细介绍。

### 3.2.1　MRWS 空闲资源评分

为了达到集群多维资源利用均衡，提升集群服务性能的目的，某个节点空闲资源越多并且各维度消耗越平均，该节点的评分就应该越高，就越适合新创建 Pod 节点的部署。针对大规模的集群，每个节点资源消耗均衡性不能完全依赖人为判断，并且评分函数的权重参数也不能全部由人工赋予，这种方式既不科学也不准

确。用户应用场景和对资源重要性判断的差异性，将导致相同应用在同一调度算法下的不同调度结果。为了避免此种情况发生，需要对预选阶段剩余的节点建模，使用一定数学方法对资源的权重参数自动求解。在构建评分算法前，先对集群节点和 Pod 资源需求进行建模 (建模中节点是预选阶段过滤后的节点):

(1) 定义集群节点和资源的符号表示。假设 $P$ 个节点列表经过预选阶段筛选后剩余 $M$ 个节点，表示为$N = (n_1, n_2, n_3 ... n_m)$。单个节点的资源维度为 $D$，在该模型中考虑节点 CPU、内存、磁盘和网络带宽四种资源因素，因此 $D$=4。集群中单个节点的资源总量表示为$S = (s_1^d, s_2^d, s_3^d ... s_m^d), \quad d \in D$，其中$s_i^d$ 表示第 i 个节点拥有的第 d 维资源的总量。同理，可以定义节点各维度资源的使用量，表示为$R = (r_1^d, r_2^d, r_3^d ... s_m^d), \quad d \in D$，其中$r_i^d$ 表示第 i 个节点上第 d 维资源的使用量。当前需要调度的 Pod 对资源的需求可以表示为$p^d, \quad d \in D$，表示 Pod 对 d 维度资源的需求量。

(2) 已部署 Pod 空闲率计算。节点上已部署 Pod 应用数量可以记录和统计，但将其作为影响调度策略的因素时没法加以限制和度量，也不能指定节点部署 Pod 总量。针对已部署 Pod 这个影响调度的因素，需要单独进行处理，使用一定的方式计算其负载和资源空闲率。考虑该因素的目的在于使每个节点部署的 Pod 数量尽量接近，若某个节点部署少量大资源需求的 Pod，另一个节点部署大量小资源需求的 Pod，虽然两个节点资源使用率相近，但是大量小资源数量 Pod 的管理会极大增加节点开销，影响该节点的服务质量。因此，应尽量使不同节点上运行的 Pod 数量均衡。采用如下的方式计算节点已部署 Pod 的负载，集群中预选阶段剩余节点上已调度的 Pod 总数 C 表示为:

$$C = (\sum_{i=1}^{m} p_i + 1) \tag{3-2}$$

其中，$p_i$ 表示第 i 个节点上已部署的 Pod 数量，可以用 $c_i$ 表示第 i 个节点已部署 Pod 的资源空闲率，如式 (3-3) 所示。

$$c_i = (1 - \frac{p_i + 1}{C}) \tag{3-3}$$

(3) 多维资源的权重系数。在评分函数模块，给评分函数集合中的每个评分函数人为赋予一个权重，最终按照总得分排名选取最高。在实际的数据中心中，根据用户调度 Pod 请求的资源数量可以感知部署的应用类型，如 CPU 密集、内存密集、I/O 密集以及网络密集型等。从经济效益角度考虑，节点上各种资源重要程度也具有较大差异，CPU 和内存较为稀缺，价格相对昂贵一点。定义权重系数$\alpha_i (i = 1, 2, 3, 4, 5)$ 分别用于表示节点 CPU、内存、磁盘 I/O、网

络带宽和已部署 Pod 数量等因素的重要程度，且 $\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 = 1$。权重参数将在下一章通过 FAHP 进行节点资源数学建模并实现自动求解，用户通过微调权重参数更加灵活使用新调度算法。

根据上述定义的集群符号，可以根据各节点资源空闲率对节点进行评分，资源空闲率模块反映节点空闲资源的数量，空闲资源越多，节点评分越高。在考虑资源空闲量时又充分考虑各维度资源的重要程度，越是宝贵的资源其重要程度越高。节点 i 空闲模块的最终评分 $v_i$ 如式 (3-4) 所示。

$$v_i = \sum_{d=1}^{D} \alpha_d * k_i^d + \alpha_5 * c_i \tag{3-4}$$

$$k_i^d = \frac{s_i^d - r_i^d - p^d}{s_i^d} \tag{3-5}$$

其中，$k_i^d$ 表示节点 i 部署 $p^d$ 后第 d 维资源的空闲率。

### 3.2.2　MRWS 平衡模块评分

在空闲资源模块评分中，各个维度的资源通过其空闲量和资源的重要程度影响评分，这会导致各维度资源利用不均衡。比如某个节点 CPU 资源空闲率很低，但其他维度空闲率很高，尽管赋予了 CPU 较高影响因子$\alpha_1$，但其整体评分依然偏高，会继续调度 CPU 密集的 Pod 到该节点，造成节点性能下降，资源消耗更加不均衡。为避免上述情况的发生，需要设计一个平衡模块，用于平衡各维度资源消耗。平衡模块用于反映节点各维度资源的均衡状况，各维度资源消耗越均衡，其评分越高。计算预选阶段剩余节点各维度资源空闲率的均值如式 (3-5)、(3-6) 所示。

$$k_v^d = \frac{(\sum_{i=1}^{m} k_i^d)}{m} \tag{3-6}$$

$$c_v = \frac{(\sum_{i=1}^{m} c_i)}{m} \tag{3-7}$$

其中，$k_v^d$ 表示经预选阶段筛选后剩余 m 个节点上第 d 维资源空闲率的平均值，$c_v$ 表示剩余节点上已部署 Pod 资源空闲率的平均值。因此，可以使用$(1 - \left|k_i^d - k_v^d\right|)$来度量节点 i 上第 d 维资源空闲率在集群中的不均衡性。同理，$(1 - |c_i - c_v|)$ 表示节点已部署 Pod 数量的不均衡性。该值越大，表明各维度资源利用越均衡，节点评分越高。平衡模块节点的最终评分$b_i$ 如式 (3-8) 所示。

$$b_i = \frac{\sum_{d=1}^{D}(\alpha_d(1 - \left|k_i^d - k_v^d\right|) + \alpha_5(1 - |c_i - c_v|))}{D + 1} \tag{3-8}$$

整个评分算法综合考虑 CPU、内存、磁盘、网络带宽以及已部署 Pod 等因素，分值由资源空闲评分和资源均衡评分组成。资源空闲率反映节点可用资源的状况，资源均衡反映节点各维度资源消耗的均衡性，单个节点最终评分如式 (3-9) 所示。

$$f_i = v_i + b_i \tag{3-9}$$

### 3.2.3　MRWS 均衡度评价模块

设计完调度算法后需要对调度算法多维资源利用均衡性进行度量，由于资源重要程度不同，不能简单用空闲资源利用率总合进行加权平均计算。在概率论和统计中，通常用方差来度量一组数据的离散程度，计算一个变量与整体均值之间的差异。因此，用预选阶段剩余节点的多维资源空闲率的方差来度量集群的负载均衡性，定义集群的负载均衡度 $u_v$ 如式 (3-11)。

$$u_i = \sqrt{\frac{1}{D+1}(\sum_{d=1}^{D}(k_i^d - k_v^d)^2 + (c_i - c_v)^2)} \tag{3-10}$$

$$u_v = (\sum_{i=1}^{m} u_i)/m \tag{3-11}$$

其中，$u_i$ 表示节点 i 上多维资源空闲率的负载均衡，该值越小，表示多维资源利用越均衡，即各维度资源利用越接近平均值。因此，用集群中满足预选阶段剩余节点负载均衡的均值表示集群的负载均衡度。这种使用方差度量集群均衡性的方法不仅适用于 MRWS 算法，对于 Kubernetes 默认的 Default 算法以及 Random、FirstFit 算法同样适用。在后面进行算法性能对比实验中，将使用上式的负载均衡度衡量不同调度算法下集群的负载均衡性。

## 3.3　本章小结

本章首先详细介绍 OpenShift Origin 容器云平台的容器编排引擎 Kubernetes 的默认调度算法 Default 的调度流程，该流程分为预选和优选两个阶段。接着对预选阶段的筛选规则和优选阶段的评分函数集合进行分析。针对其调度流程和调度算法的不足，提出了一种 MRWS 调度算法，该算法先对调度流程进行适当的优化，使其调度记忆功能。然后根据算法的需要对集群和应用资源建模，转为为数学表示，为下一章使用 FAHP 权重参数求解打下基础。最后设计了详细的评分规则，根据权重参数和资源的乘积和进行评分计算，资源的权重参数自动求解将在下一章节进行详细的介绍。

# 第 4 章　FAHP 多维资源建模和参数自动求解

在上一章调度算法 MRWS 的评分函数中，根据资源的重要程度使用系数 $\alpha_i(i = 1, 2, 3, 4, 5)$，　$\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 + \alpha_5 = 1$ 作为节点 CPU、内存、磁盘 I/O、网络带宽和已部署 Pod 数量的重要程度系数。本章需要对筛选后的节点和容器应用资源进行建模，并利用 FAHP 方法进行 Pod 应用多维权重参数的自动求解。

## 4.1　模糊层次分析法 FAHP

模糊层次分析法 FAHP[26-27](Fuzzy Analytical Hierarchy Process) 是在层次分析法的基础发展而来，常用于解决实际问题中影响决策问题的多种因素的权重。下面从层次分析法开始介绍，为弥补层次分析法的不足引入 FAHP 方法，然后举例说明如何使用该方法解决实际问题。

### 4.1.1　FAHP 方法介绍

在面对许多实际问题解决方案时，我们往往需要对影响方案的因素进行比较、判断、评价，然后做出决策，这种人工主观解决问题的方式不但效率低下，结果也不够准确。层析分析法 AHP[28-29](Analytical Hierarchy Process) 是由著名的美国运筹学家匹茨堡大学 T.L.Saaty 教授提出的一种层次权重决策分析方法。该方法用于解决复杂的多目标决策问题，将影响目标决策的因素分解为目标层、准则层和方案层等层次，使用数学方法对各因素权重进行精确求解。层次分析法将问题的总目标、各层子目标以及决策因素分解成多个层次结构，然后构建各层次的判断矩阵，求解判断矩阵的特征值。最大特征值对应的特征向量归一化获得上一层各元素对本层目标的优先权重，最后用加权和的方法递阶归并各备选方案对总目标的最终权重，选择权重最大的备选方案作为最终的方案。该方法是一个系统性的分析方法，先将问题分解成多个层次，然后对影响子目标的因素进行比较判断、最终进行综合决策。结构层次清晰、单层的权重参数设置都会影响到最终的决策层，不断分割量化各因素的影响。层次分析法非常实用，将定量分析和定性分析相结合，用于解决许多实际问题如电力分配、旅游决策等，该方法计算简单明了，易于掌握。最后，该方法用于模拟人们解决问题的思维，所需的定量信息较少，仅需要对影响决策因素的重要性做出判断即可。

但是，层次分析法也具有局限性，在一些场景下无法使用或者效果较差。首先，该方法只能从实际的备选方案中帮助决策者选出较优的决策方案，不能给决

策者提供新的解决方案或者反馈合理方案的意见，并且该方法是单目标决策方法，使用者只能从备选目标中选出一个最优的目标。其次，该方法模拟人类大脑决策过程，定性分析过于浓厚，只能进行粗略比较计算，不能完全做到精确模拟。最后，从层次结构转化为成对比矩阵过程中，判断者的主观因素影响过大，判断者不同，得出的最优方案也不尽相同，无法让人信服。并且随着影响决策因素的增加，构建的成对比矩阵阶数增大，计算难度和复杂度随之增加，很难一次性构建出满足一致性要求的成对比矩阵。

为减少人为主观因素对决策结果的影响，模糊层析分析法 FAHP 在层次分析法的基础上增加人为判断的模糊性，具体而言就是层析分析通过两两比较构建成对比判断矩阵。FAHP 通过两两比较构建模糊成对比矩阵，提升了层次分析法解决问题的可靠性。在进行问题判断或专家咨询时，往往不是给出一个具体值，而是一个模糊数，如三值判断 (最低可能值、最可能值和最大可能值)、二值区间等。下面介绍模糊数集的概念，对于一个明确的集合:

$$\mu_A(x) = \begin{cases} 1, x \in A \\ 0, x \notin A \end{cases} \tag{4-1}$$

在明确的集合中，$x \in A$ 时值为 1，$x \notin A$ 时值为 0。但对于一个模糊数集，并不能完全明确 x 是否属于 A，只能用 [0,1] 表示其隶属度。

$$\mu_A(x) : U \to [0, 1] \tag{4-2}$$

$\mu_A(x)$ 表示$x \in A$ 的隶属度，也称$\mu_A(x)$ 为集合 A 的隶属函数。对于任意的模糊数集，都对应一个隶属函数，隶属函数通常模仿概率论中的分布函数如正态分布、梯形分布、K 次抛物线分布、S 分布以及柯西分布等，值域在 [0,1] 上。

1983 年，荷兰学者 Van Loargoven 首次提出用三角模糊数[30-31] 作为模糊数集的判断标准，并运用三角模糊数的运算和对数最小二乘法获取权重值。设论域 R 上的模糊数 $\widetilde{M}$ 为三角模糊数，其对应的隶属函数$\mu_{\widetilde{M}}.R \to [0, 1]$ 满足下列函数:

$$\mu_{\widetilde{M}}(x) = \begin{cases} 0 & x < a \\ \frac{x-a}{b-a} & a \le x \le b \\ \frac{c-x}{c-b} & b \le x \le c \\ 0 & x > c \end{cases} \tag{4-3}$$

$\widetilde{M} = (a, b, c)$ 其中$a \le b \le c$，a 和 c 分别表示三角模糊数的上界和下界值，b 是隶属度为 1 的中间值，x=b 时表示 x 完全属于 $\widetilde{M}$，在 a 和 c 之外的不属于模糊数 $\widetilde{M}$。

定义一个置信度$\alpha$ 可以将三元组的模糊数 $\widetilde{M}$ 化为一个 $\alpha$ 割集的二元形式，从而构建割集矩阵，设定优化参数后将二元值的矩阵转化成最终的判断矩阵。

$$\widetilde{M}_\alpha = [l^\alpha, u^\alpha] = [(b-a)\alpha + a, -(c-b)\alpha + c] \quad \forall\alpha \in [0,1] \tag{4-4}$$

根据 Arnold J. Kaufmann 在文献 [32] 中的描述，$\alpha$ 割集后的基本运算规则如下：

$$\begin{aligned}
\widetilde{M}_\alpha &= [m_L^\alpha, m_R^\alpha] \\
\widetilde{N}_\alpha &= [n_L^\alpha, n_R^\alpha] \\
\widetilde{M}_\alpha \oplus \widetilde{N}_\alpha &= [m_L^\alpha + n_L^\alpha, m_R^\alpha + n_R^\alpha] \\
\widetilde{M}_\alpha \ominus \widetilde{N}_\alpha &= [m_L^\alpha - n_L^\alpha, m_R^\alpha - n_R^\alpha] \\
\widetilde{M}_\alpha \otimes \widetilde{N}_\alpha &= [m_L^\alpha n_L^\alpha, m_R^\alpha n_R^\alpha] \\
\widetilde{M}_\alpha \oslash \widetilde{N}_\alpha &= [m_L^\alpha / n_L^\alpha, m_R^\alpha / n_R^\alpha]
\end{aligned} \tag{4-5}$$

本文采用三角模糊数进行 FAHP 的权重参数求解，并将三元组的 $\widetilde{M}$=(a,b,c) 转化成 $\alpha$ 割集形式 $\widetilde{M}_\alpha = [(b-a)\alpha + a, -(c-b)\alpha + c] \quad \forall\alpha \in [0,1]$。

### 4.1.2　FAHP 求解权重参数步骤

介绍完层次分析方法和模糊层析分析方法以及模糊数的基本概念后，下面用三角模糊数作为模糊程度的衡量值，FAHP 求解权重参数流程如下：

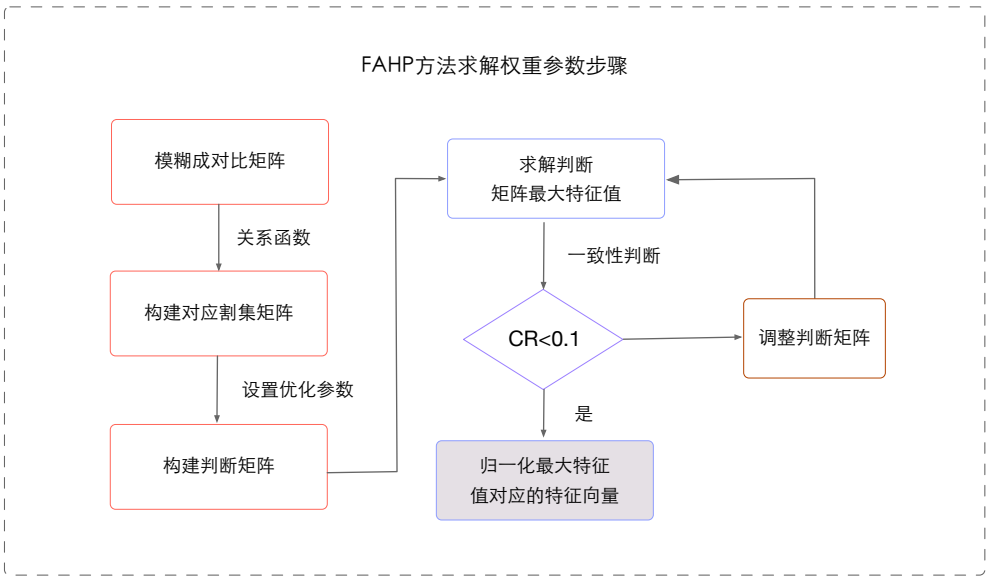



图 4.1　FAHP 计算权重参数过程

如图 4.2 所示，使用 FAHP 方法计算权重参数的步骤可以分为构建模糊成对比矩阵、转化为二元割集矩阵、设置优化参数转化为判断矩阵，求解判断矩阵的最大特征值，检验特征值是否满足一致性要求。若满足则将最大特征值对应的特征向量归一化作为权重参数值，否则需要调整判断矩阵值，重新进行特征值计算。下面详细介绍求解步骤:

1) 构建模糊成对比矩阵。在 FAHP 中，对影响决策因素进行两两对比时使用$\widetilde{1} \sim \widetilde{9}$表示其相对重要程度，值越大，表示该因素相对另一个因素对决策目标的影响越大，重要性越高。模糊数、相对重要程度、$\widetilde{M}$ 三元组及其 $\alpha$ 割集如表 4.1 所示，可以构建最终的模糊成对比矩阵 $\widetilde{A}$，其中的元素 $\widetilde{a}_{ij}$ 表示因素 i 相对 j 的重要程度，模糊成对比矩阵中元素值如式 (4-6) 所示。

$$\widetilde{a}_{ij} = \begin{cases} 1 \quad i = j \\ \widetilde{1}, \widetilde{3}, \widetilde{5}, \widetilde{7}, \widetilde{9} \; or \; \widetilde{1}^{-1}, \widetilde{3}^{-1}, \widetilde{5}^{-1}, \widetilde{7}^{-1}, \widetilde{9}^{-1} \quad i \neq j \end{cases} \tag{4-6}$$

表 4.1　模糊数、相对重要程度和割集关系表

| 模糊数 | $\widetilde{1}$ | $\widetilde{3}$ | $\widetilde{5}$ | $\widetilde{7}$ | $\widetilde{9}$ |
|---|---|---|---|---|---|
| 重要性 | 同等重要 | 稍微重要 | 重要 | 明显重要 | 非常重要 |
| $\widetilde{M}$ | (1,1,3) | (1,3,5) | (3,5,7) | (5,7,9) | (7,9,11) |
| $\widetilde{M}_\alpha$ | [1,3-2$\alpha$] | [1+2$\alpha$,5-2$\alpha$] | [3+2$\alpha$,7-2$\alpha$] | [5+2$\alpha$,9-2$\alpha$] | [7+2$\alpha$,11-2$\alpha$] |

2) 构建割集矩阵并转化为判断矩阵。构建模糊成对比矩阵后，根据表 4.1 将模糊数转化成三元组的三角模糊数，给定 $\alpha$ 后，将三元组转化成 $\alpha$ 割集 $\widetilde{M}_\alpha = [l_\alpha, u_\alpha]$，从而构建一个 $\alpha$ 割集矩阵 $\widetilde{A}_\alpha$。割集矩阵中的元素 $\widetilde{a}_{ij}^{\alpha}$ 如式 (4-7) 所示。给定一个优化参数表示判断优化程度，可以将二元组的割集矩阵转化成判断矩阵 A，从而可以对判断矩阵进行特征值和特征向量的求解。设给定的优化参数 $\mu \in [0,1]$，将割集矩阵中上下界范围转化成判断矩阵的单值，判断矩阵中的元素 $a_{ij} = (1-\mu)l_\alpha + \mu u_\alpha$，如式 (4-8) 所示。通过给定 $\alpha$ 获得二元组的割集矩阵，给定割集矩阵一个优化参数 $\mu$ 可以得到最终的判断矩阵 A。

$$\widetilde{a}_{ij}^{\alpha} = \begin{cases} 1 \quad \widetilde{a}_{ij} = 1 \\ [l_\alpha, u_\alpha] \quad \widetilde{a}_{ij} \in \widetilde{1}, \widetilde{3}, \widetilde{5}, \widetilde{7}, \widetilde{9} \\ [\frac{1}{u_\alpha}, \frac{1}{l_\alpha}] \quad \widetilde{a}_{ij} \in \widetilde{1}^{-1}, \widetilde{3}^{-1}, \widetilde{5}^{-1}, \widetilde{7}^{-1}, \widetilde{9}^{-1} \end{cases} \tag{4-7}$$

$$a_{ij} = \begin{cases} 1 & \widetilde{a}_{ij} = 1 \\ (1-\mu)l_\alpha + \mu u_\alpha & \widetilde{a}_{ij} \in \widetilde{1}, \widetilde{3}, \widetilde{5}, \widetilde{7}, \widetilde{9} \\ \frac{1-\mu}{u_\alpha} + \frac{\mu}{l_\alpha} & \widetilde{a}_{ij} \in \widetilde{1}^{-1}, \widetilde{3}^{-1}, \widetilde{5}^{-1}, \widetilde{7}^{-1}, \widetilde{9}^{-1} \end{cases} \tag{4-8}$$

3) 计算判断矩阵最大特征值和特征向量。对判断矩阵特征值的求解是一个纯数学问题，求解方阵特征值和特征向量的方法有几何平均法 (方根法)、算术平均平均法 (和法)、最小二乘法 (幂法) 和特征向量法。通常几种方式求解的特征值很相近，但也存在细微的差别，这些小的差别对实际生产会造成一定的影响。一般情况下，几何平均法和算术平均获得的结果精确度较差，最小二乘法计算复杂度较高，因此使用特征向量法对判断矩阵的特征值和特征向量进行求解。在本文中，直接使用 Python 中 numpy 库 eig() 函数求解判断矩阵的特征值和特征向量。

4) 一致性检验和归一化权重。通过特征向量法获得判断矩阵的最大特征值 $\lambda_{max}$ 及其对应的特征向量 $v_{max}$ 后，需要进行层次单排序和层次总排序的一致性检验，用于检测特征向量和真实权值的契合度，即求解的权重值是否合理。判断矩阵一致性检测方法公式如式 (4-9) 所示。

$$\begin{aligned} CI &= \frac{\lambda_{max} - n}{n-1} \\ CR &= \frac{CI}{RI} \end{aligned} \tag{4-9}$$

其中，$\lambda_{max}$ 是判断矩阵最大特征值，n 是影响决策因素的数量即判断矩阵的阶数，CI 表示一般一致性指标，RI 是随机一致性指标，因此 CR 是随机一致性比率。若 CR<0.1 则判断矩阵满足一致性要求，否则需要调整判断矩阵直至其满足一致性判断要求。当 CR<0.1 时将 $\lambda_{max}$ 对应的特征向量 $v_{max}$ 归一化，其值 w 就是所需求解的权重值。常用的随机一致性指标 RI 的值如表 4.2 所示。

表 4.2　随机一致性指标 RI 值

| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| RI | 0 | 0 | 0.58 | 0.90 | 1.12 | 1.24 | 1.32 | 1.41 | 1.45 | 1.49 |

以上是用 FAHP 方法求解权重参数的全部过程，首先对影响决策的因素进行两两比较，构建模糊成对比矩阵，用 $\widetilde{1} \sim \widetilde{9}$ 表示相对重要程度。根据成对比矩阵，给定 $\alpha \in [0,1]$ 获得割集矩阵，设定优化参数 $\mu \in [0,1]$ 获得判断矩阵，计算判断矩阵的最大特征值和对应的特征向量，检测最大特质值是否满足一致性要求，最后将最大特征向量归一化获得权重值。

### 4.1.3　FAHP 求解权重参数示例

上述介绍了 FAHP 求解权重参数的详细流程和计算方法，下面通过一个简单的例子展示该方法求解权重参数的计算过程，假设在一个物理问题中力学指数的评价由巴氏硬度、耐荷重性、耐冲击性和满水变形四个因素决定，其影响力学指数的判断因素如图 4.2 所示。

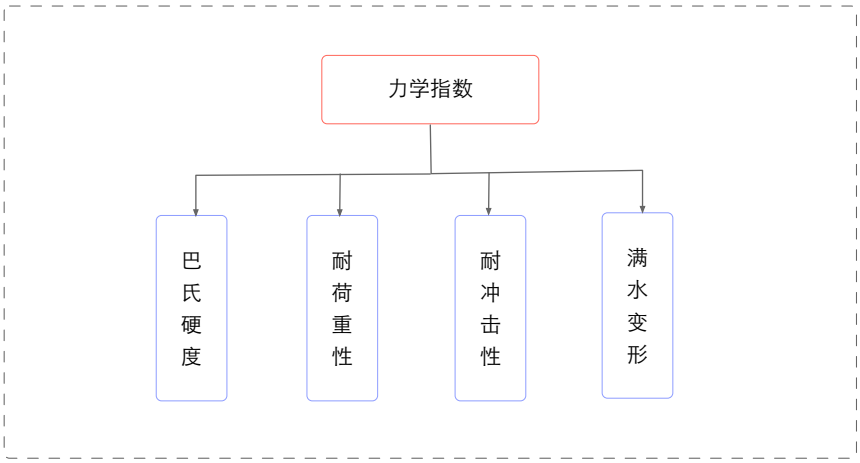



图 4.2　力学指数的评价因素

某专家通过对影响力学指数的因素进行两两对比后给出其三角模糊判断数，根据其给出的模糊数构建模糊成对比矩阵式 (4-10)，从左至右依次为巴氏硬度、耐荷重性、耐冲击性和满水变形。

$$\widetilde{A}=\begin{Bmatrix} 1 & \widetilde{3} & \widetilde{7} & \widetilde{5} \\ \widetilde{3}^{-1} & 1 & \widetilde{5} & \widetilde{3} \\ \widetilde{7}^{-1} & \widetilde{5}^{-1} & 1 & \widetilde{3}^{-1} \\ \widetilde{5}^{-1} & \widetilde{3}^{-1} & \widetilde{3} & 1 \end{Bmatrix} \tag{4-10}$$

给定 $\alpha = 0.5$ 和优化参数 $\mu = 0.5$，根据式 (4-7) 和 (4-8) 可以获得割集矩阵和判断矩阵，判断矩阵 A 如下:

$$A=\begin{Bmatrix} 1 & 3 & 7 & 5 \\ \frac{3}{8} & 1 & 5 & 3 \\ \frac{7}{48} & \frac{5}{24} & 1 & \frac{3}{8} \\ \frac{5}{24} & \frac{3}{8} & 3 & 1 \end{Bmatrix} \tag{4-11}$$

使用特征向量法对判断矩阵进行特征值和特征向量求解，最大特征值 $\lambda_{max} = 4.21154$，其对应的特征向量值 $\nu_{max} = [0.88297, 0.41986, 0.08951, 0.18992]$，使用最

大特征值对帕判断矩阵 A 的一致性进行判断，CR=0.082<0.1 满足一致性要求。对其最大特征值对应的特征向量进行归一化，其权重 w=[0.558,0.265,0.057,0.120] 分别表示巴氏硬度、耐荷重性、耐冲击性和满水变形四个因素对力学系数影响的权重系数，用于对后面的目标层做出决策。

## 4.2　容器应用多维资源权重自动求解

上一小节详细介绍了 FAHP 方法求解权重系数的流程，并用一个力学系数的例子展示其计算过程。在 MRWS 调度算法中，需要使用容器应用的权重参数进行节点评分，由于需要调度的容器应用数量庞大，不能人为的给每一个容器应用的资源需求赋一个权重值，这样既不准确也实用。因此，下面将介绍使用 FAHP 方法对容器应用的多维资源进行建模和自动化求解。

### 4.2.1　容器应用多维资源建模

在 MRWS 调度算法中，集群节点 CPU、内存、磁盘和网络带宽以及已部署的 Pod 数量作为影响容器应用调度决策的因素，预选阶段筛选后的节点作为其调度的备选节点。不同的用户和应用场景对用户的资源需求不同，容器云中允许用户配置应用的资源值既能节约成本，又能对用户实现按需服务。首先需要对影响调度的因素进行分层建模，如图 4.3 所示。整个层次可以分为目标层、准则层和方案层。目标层是选择一个预选阶段过滤的后的节点作为容器应用调度的目标；准测层是影响调度决策的因素，包括节点 CPU、内存、磁盘、网络带宽以及已部署 Pod 数量；方案层是所有满足容器资源需求的主机。从该层次模型中可以发现，每一个因素都对方案层直接施加影响，是一个全层次的模型结构。

调度算法的快慢是衡量服务的重要指标，一种快速的调度算法对容器云至关重要。MRWS 调度算法在优化的调度流程中，增加一个应用类型和对应的系数存储模块，初始可以通过数据中心感知的应用类型，分级建立部分权重系数库。一旦有新的容器应用请求，将需求的资源和库中的容器层级进行对比，如果资源需求差值在可接受范围内，直接使用以前应用的系数，实现快速调度。若以前系数库中不存在该应用类型，需要重新计算该类应用的资源权重系数，并保存到系数库中，不断扩充应用和系数对应的库，也可以人工调整参数，提升精确度。在本文的方法中，权重参数根据资源需求占用集群上节点单位资源组的比例实现自动求解，无需用户给出模糊成对比矩阵，用户也可以对权重库加以修正。

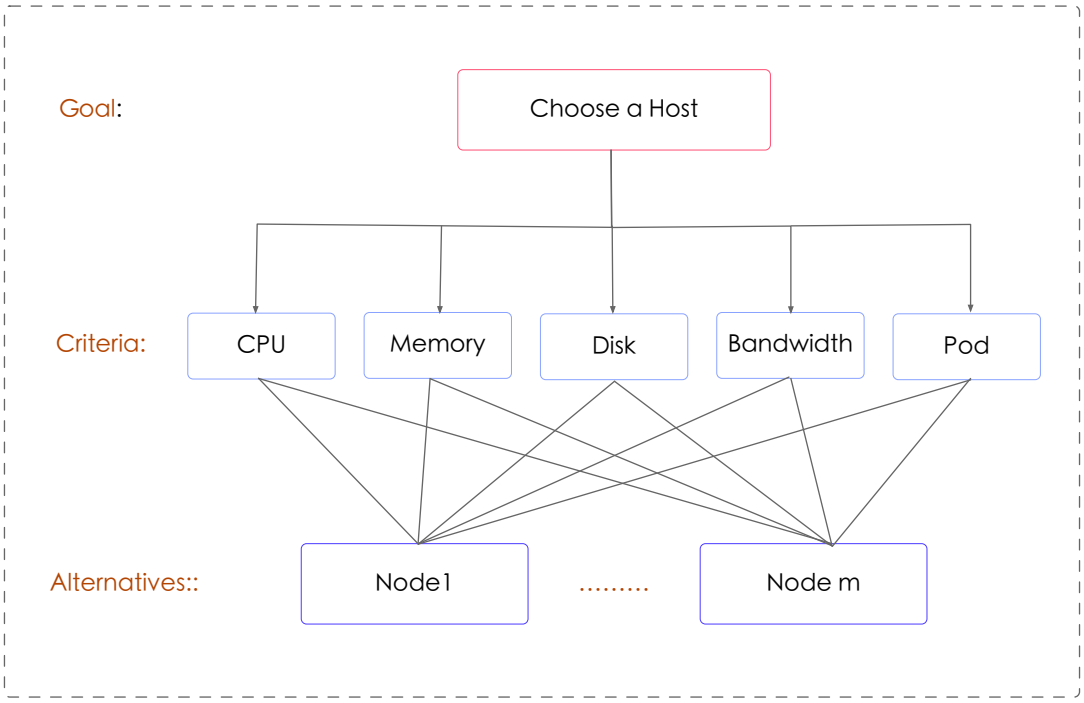



图 4.3　多维资源递阶模糊层次结构

在云计算数据中心，CPU 和内存一般是相对稀缺的资源，因此，容器应用中 CPU 和内存相对其他资源需求重要性一般要高。单个节点上部署 Pod 数量较大，已部署 Pod 这个影响决策的因素相对其他因素的重要性较弱，但是大量小资源需求 Pod 会增加节点的管理成本。在构建模糊成对比矩阵时，该因素的模糊数较小，只是作为平衡各节点运行 Pod 数量，容器应用的资源需求才是首要考虑因素。根据上述的层次结构图，下面介绍权重参数的自动求解方法。

### 4.2.2　多维资源权重参数自动求解

容器应用多维资源递阶模糊层次结构如图 4.3 所示，大量容器应用进行模糊成对比矩阵构建时完全依赖人工判断既不准确也不满足实际需求。在实际系统中，需要对每一个容器应用的多维资源进行权重参数求解，自动构建模糊成对比矩阵，从而转化为割集矩阵和判断，实现最大特征值的自动求解和判断矩阵的一致性检测。若判断矩阵不满足一致性要求，程序需要能自动调整判断矩阵直至其满足一致性检验。因此，如何对影响决策的 CPU、内存、磁盘、网络带宽和已部署 Pod 等几个影响决策的因素构建模糊成对比矩阵成为解决问题的关键，整体的解决思路是用待调度的容器所需求的资源占单位资源组的百分比进行两两对比。设某个容器应用对各维度在资源需求和节点单位资源组如表 4.3 所示，需要求解各维度资源的权重系数 $w_c, w_m, w_d, w_b, w_p$ 且 $w_c + w_m + w_d + w_b + w_p = 1$。

表 4.3　某容器应用和节点资源总量表

| Resources | CPU | Memory | Disk | Bandwidth | Pod |
|---|---|---|---|---|---|
| U-Node | C | M | D | B | .. |
| Pod | $C_0$ | $M_0$ | $D_0$ | $B_0$ | 1 |
| Weight | $w_c$ | $w_m$ | $w_d$ | $w_b$ | $w_p$ |

在构建模糊成对比矩阵时，根据容器应用需求资源和单位资源组的比值之间的差值构建模糊数。如在对比 CPU 和内存的重要性时，采用 $|C_0/C - M_0/M|$ 作为判断依据，使用 $\widetilde{1} \sim \widetilde{9}$ 表示相对重要程度的大小，在 [0,1] 范围内将其切分为 5 份，根据比值差值自动构建模糊数。模糊成对比矩阵的模糊数、容器应用需求各维度资源和单位资源组的比值差值对应如表 4.4 所示。在数据中心的集群中，各节点的资源构成通常是异构的，即单个服务器各维度资源不同，不能直接使用容器应用与各节点资源比值的差值获取模糊数，因此需要抽象出单个资源组作为评判的依据。如 1 个 core 的 CPU、1G 内存、50G 磁盘、10M 带宽作为一个单位资源组，单位资源组的大小根据集群实际情况构建，将容器应用和单位资源组的比值差值作为模糊数的依据。最后使用表 4.4 的方法构建模糊成对比矩阵。

表 4.4　模糊数和资源比值差值对应表

| 差值 | (0, 0.2] | (0.2, 0.4] | (0.4, 0.6] | (0.6, 0.8] | (0.8, 1) |
|---|---|---|---|---|---|
| 模糊数 | $\widetilde{1}$ | $\widetilde{3}$ | $\widetilde{5}$ | $\widetilde{7}$ | $\widetilde{9}$ |
| 差值 | (-0.2,0] | (-0.4, -0.2] | (-0.6, -0.4] | (-0.8, -0.6] | (-1, -0.8] |
| 模糊数 | $\widetilde{1}^{-1}$ | $\widetilde{3}^{-1}$ | $\widetilde{5}^{-1}$ | $\widetilde{7}^{-1}$ | $\widetilde{9}^{-1}$ |

针对已部署 Pod 这一影响调度的因素，其重要性相对其他资源要低，容器应用对该资源的需求以及节点上能部署的 Pod 总数都无法度量，不能像其他资源一样直接通过比值的差值作为模糊数的获取依据。因此，将 CPU、内存、磁盘和网络带宽四种资源的比值从小到大排序，相较于 Pod 这一因素的重要性依次获得模糊数为 $(\widetilde{3}, \widetilde{5}, \widetilde{7}, \widetilde{9})$，反之，Pod 相对于其他资源的模糊数为 $(\widetilde{3}^{-1}, \widetilde{5}^{-1}, \widetilde{7}^{-1}, \widetilde{9}^{-1})$。

容器应用资源需求和单位资源组比值的差值可以获取各因素之间的两两对比的模糊数，从而构建模糊成对比矩阵，最终自动获取判断矩阵。求解出判断矩阵的特征值和对应的特征向量，判断矩阵满足一致性判断后，归一化出各因素的权重值，实现权重参数的自动求解。下面用一个实际的例子详细展示模糊成对比矩阵的构建以及最终权重参数的求解过程。

若某个集群的单位资源组配置如表 4.5 所示，集群中各节点各维度资源的配置都是单位资源组的整数倍，单位资源组构成集群资源的最小组成单元。提出单

位资源组的概念主要是为解决异构集群的场景，通常数据中心的集群都是异构的，尤其是在云计算中心中，需要将互联网上大量闲置的计算机作为服务计算单元，这种整合的闲置资源，并不是同构服务集群。

表 4.5　物理集群单位资源组配置

| 类型 | CPU(M) | MEM(M) | Disk(G) | BW(M) |
|---|---|---|---|---|
| 数量 | 1200 | 8000 | 500 | 50 |

在该数据中心集群上进行应用容器的调度，设某个应用容器对各维度资源的需求和集群单位资源组的比值如表 4.6 所示。

表 4.6　某应用容器的资源需求

| 类型 | CPU(M) | MEM(M) | Disk(G) | BW(M) | Pod |
|---|---|---|---|---|---|
| 数量 | 300 | 1100 | 210 | 40 | 1 |
| 占比 | 0.25 | 0.1375 | 0.42 | 0.8 | - |

根据各维度资源所占比值的差值作为模糊数的依据，可以构建出该容器应用的模糊成对比矩阵，已部署 Pod 因素影响较弱，其他因素与该因素对比获取的模糊数较大。自动构建的模糊成对比矩阵如式 (4-12) 所示，行和列依次为 CPU、内存、磁盘、网络带宽以及已部署 Pod 等影响决策的因素，通过两两对比确定模糊数后获得一个 5*5 的模糊成对比方阵。

$$\widetilde{A}=\begin{Bmatrix} 1 & \widetilde{1} & \widetilde{1}^{-1} & \widetilde{5}^{-1} & \widetilde{5} \\ \widetilde{1}^{-1} & 1 & \widetilde{3}^{-1} & \widetilde{7}^{-1} & \widetilde{3} \\ \widetilde{1} & \widetilde{3} & 1 & \widetilde{3}^{-1} & \widetilde{7} \\ \widetilde{5} & \widetilde{7} & \widetilde{3} & 1 & \widetilde{9} \\ \widetilde{5}^{-1} & \widetilde{3}^{-1} & \widetilde{7}^{-1} & \widetilde{9}^{-1} & 1 \end{Bmatrix} \tag{4-12}$$

给定割集参数 $\alpha = 0.5$，优化参数 $\mu = 0.5$，参数 $\alpha$ 反映模糊数中最大可能值的模糊程度，即真实值与最大可能值的距离，当 $\alpha = 1$ 时，模糊数的真实值就是最大可能值，$\alpha = 0$ 时模糊程度出现最大范围，真实值和最大可能值接近程度最低。$\mu$ 作为优化参数，在模糊给定 $\alpha$ 后反映上界和下界的取值，$\mu = 1$ 时取上界值，$\mu = 0$ 时取下界值。两个参数对权重参数的影响可以参考文献 [26]，该文献对两个参数对权重产生的影响进行了详细的分析。将上述的模糊成对比矩阵转化为割集矩阵，

然后再转变为实际判断矩阵如 (4-13) 所示。

$$A = \begin{Bmatrix} 1 & \frac{3}{2} & \frac{3}{4} & \frac{5}{24} & 5 \\ \frac{3}{4} & 1 & \frac{3}{8} & \frac{7}{48} & 3 \\ \frac{3}{2} & 3 & 1 & \frac{3}{8} & 7 \\ 5 & 7 & 3 & 1 & 9 \\ \frac{5}{24} & \frac{3}{8} & \frac{7}{48} & \frac{9}{80} & 1 \end{Bmatrix} \tag{4-13}$$

使用特征向量法求解判断矩阵的特征值和对应的特征向量，获得的特征值 $\lambda_{max} = 5.2498$，计算一般一致性指标 CI=(5.2498-5)/4=0.0623，查找随机一致性 RI 表在 n=5 时其值 RI=1.12，因此随机一致性比率 CR=CI/RI=0.0558<0.1，判断矩阵满足随机一致性比率 CR<0.1 的要求。

满足一致性的判断矩阵 A 最大特征值 $\lambda_{max}$ 对应的特征向量 $\nu_{max}$ 的值为 $(0.2291, 0.1452, 0.3607, 0.8903, 0.0600)$，将该特征向量归一化获得容器应用 CPU、内存、磁盘、网络带宽以及已部署 Pod 几个因素的权重系数 $w = (w_c, w_m, w_d, w_b, w_p) = (0.136, 0.086, 0.214, 0.528, 0.036)$。从权重参数可以看出，该容器应用是一个网络密集型的应用，该应用对网络带宽资源需求更多，其权重参数更大，因此，权重参数也能反映容器应用对某个维度资源的需求量。至此，一个完整的自动构建模糊成对比矩阵并实现各维资源权重参数自动求解过程完成。

MRWS 调度算法中，容器应用在进行调度时需要对集群中节点进行评分，选择评分最高的节点作为容器调度目标，在对节点空闲资源评分时需要使用容器应用需求资源权重系数作为乘积系数。在式 (3-4) 和 (3-8) 中，$\alpha$ 系数就是待调度容器应用各维资源的权重系数，该系数大小反映待调度容器对某个维度资源需求量。如一个 CPU 密集型应用，系数 $\alpha_1$ 较大，节点中若某个节点 CPU 资源空闲较多，则该节点评分相对较高，Pod 调度到该节点的几率就越大。使用 FAHP 对待调度容器 Pod 各维资源进行层次建模后，通过容器需求资源和资源单元组的比值之间差值构建模糊成对比矩阵，最终自动求解出来的各资源权重值就是空闲资源评分系数。其中单位资源组大小、$\alpha$ 割集系数和优化参数的选取对调度评分都至关重要。使用系数调节后的 MRWS 算法可以实现容器云集群中各维资源的均衡消耗，将 CPU 密集型应用调度到 CPU 剩余资源较多并且各维资源使用较为平衡的节点上，其他密集型的应用调度相似。因此，这种调度方法将各种密集型容器应用尽可能分散混合部署到节点上，避免大量相同密集型集中调度到相同节点上，造成某一维度的资源资源不足，尤其是在大数据处理多维计算框架应用调度的场景下，能够提升集群资源利用率，极大提升了集群的服务性能。

## 4.3　本章小结

在 MRWS 调度算法中，需要求解容器应用各维度资源的权重参数作为节点评分乘积系数，本章介绍 FAHP 求解容器应用各维资源权重参数的方法和过程。首先从层次分析法 AHP 开始，介绍该方法的历史背景和解决问题的能力。为减少专家判断的主观性和模糊性，使用各种分布函数作为模糊数，选定三角模糊数作为判断依据，从而引入了模糊层次分析法 FAHP，并使用 $\widetilde{1} \sim \widetilde{9}$ 表示因素之间相对重要程度。接着详细介绍该方法求解权重系数的步骤，首先构建模糊成对比矩阵，给定置信度 $\alpha$ 将其转化为割集矩阵，设置优化参数 $\mu$ 后转化为判断矩阵，求解判断矩阵最大特征值和特征向量，检测该判断矩阵是否满足一致性要求。通过调整判断矩阵，满足一致性判断要求后，归一化最大特征值对应的特征向量，然后通过一个力学系数的例子展示各计算步骤的详细计算过程。

最后，使用 FAHP 求解容器应用各维度资源的权重参数，先对各资源需求和备选节点进行层次建模，获得一个全层次的模型结构。将容器应用各维度资源和单位资源组的比值之间的差值作为自动构建模糊成对比矩阵的依据，获得模糊成对比矩阵，并转化为割集矩阵和判断矩阵，实现权重参数的自动求解。使用一个具体的容器应用展示该应用各维度资源的权重系数自动计算过程，最终获得的权重参数就是 MRWS 调度算法中用于各节点空闲资源和资源平衡性的评分计算系数。

# 第 5 章　大数据存储与处理容器云平台 Paladin 和调度实验

提出 MRWS 调度方案优化 OpenShift 容器云平台底层容器编排引擎 Kubernetes 调度流程和调度算法后，需要对其资源利用率和负载均衡性进行评估测试。本章实验主要包括如下几个部分:

(1) 在 ContainerCloudSim 容器云仿真平台上进行大规模容器应用调度仿真、对比 Kubernetes 默认的 Default 算法、Random、FirstFit 和 MRWS 调度算法在资源利用率和负载均衡性方面的性能。

(2) 基于开源 OpenShift Origin 和实验室项目 Paladin storage 开发了大数据存储与处理的 Paladin 容器云平台，该平台是一个集海量数据存储管理、多计算框架快速部署、调度优化、用户注册、按需服务等多功能的 PaaS 平台。

(3) 在 Paladin 开发部署数十种大数据处理框架如 Hadoop、Spark、Storm 等，将其打包成镜像文件，Push 到仓库中，用户可以快速构建大数据处理环境。

(4) 在 Paladin 容器云平台设计并实现 MRWS 算法的调度器，使用多计算框架容器应用混合部署进行调度测试，对比几种调度方案的性能。

## 5.1　ContainerCloudSim 仿真 MRWS 调度方案

实验室研发的大数据存储与处理容器云平台 Paladin 是在小规模物理集群上部署，不能进行调度方案的大规模测试和分析。为了评估 MRWS 调度方案的性能和可靠性，首先在容器云仿真平台 ContainerCloudSim[33] 上进行大规模的仿真实验。首先构建仿真实验环境，然后开发 Kubernetes 默认 Default 调度算法、Random 和 MRWS 调度算法，使用大规模负载进行测试分析。

### 5.1.1　ContainerCloudSim 容器云仿真平台

#### 5.1.1.1　CloudSim 云仿真平台

容器和容器编排技术的逐渐成熟推动了容器云的飞速发展，在计算中心的容器云上为了评测资源调度策略和容器服务性能，一个容器云仿真平台变得异常重要。调度方案经过仿真平台大量测试和对比分析，不仅可以节约开发时间，也能避免资源浪费和减少试错成本。不同的应用场景下针对新的调度方案，如果部署大规模容器云平台进行性能分析测试，绝大部分小公司和开发人员并不具备这种条件。在传统的云计算模式下，应用服务的组成、供应、配置和部署条件较为复杂，

当用户需求和系统配置动态变化时，评估一个调度策略以及工作负载是相当困难的，一个优秀的云计算平台模拟器可以很好解决这个问题。云平台仿真模拟器通过控制环境变量和重复试验可以加速理论研究和开发过程，根据需求和应用场景不同，各大公司和研究机构推出了一系列云计算平台仿真工具。

MDCSim 是一个全面、灵活、可扩展的多层数据中心模拟器，整个模拟平台分为通信层、内核层和用户层建模，通信层用于模拟模拟集群内部通信、内核层模拟调度和分析系统性能、用户层用于模拟各种应用。该模拟器可以根据底层不同硬件特点进行混合建模，用于评估数据中心的能耗，让用户在保持低功耗的同时实现集群服务性能的提升。GroudSim 是一个基于 Java 的模拟器，用于模拟科学应用在网格和云设施上执行问题，模拟完成后给用户提供了基础的统计和分析功能。NetworkCloudSim 用于解决网络模拟问题，弥补其他模拟器对网络细节关注不足，支持 MPI 和工作流。TeachCloud 用于对 MapReduce 应用建模并集成一个负载生成器，提供图形化的接口和实现定制的网络拓扑结构。CloudSim 是墨尔本大学 Gridbus 项目推出的云计算仿真软件，既能对系统性能和应用服务模拟、仿真、试验，也能评估资源调度策略的优劣。

CloudSim 是一个开源的仿真软件，最大特点就是提供一个虚拟化引擎，帮助数据中心建立和管理各种虚拟化服务。支持大规模云计算资源管理和调度模拟，将数据中心的资源虚拟化为资源池，CloudSim 的分层体系架构图如 5.1 所示。

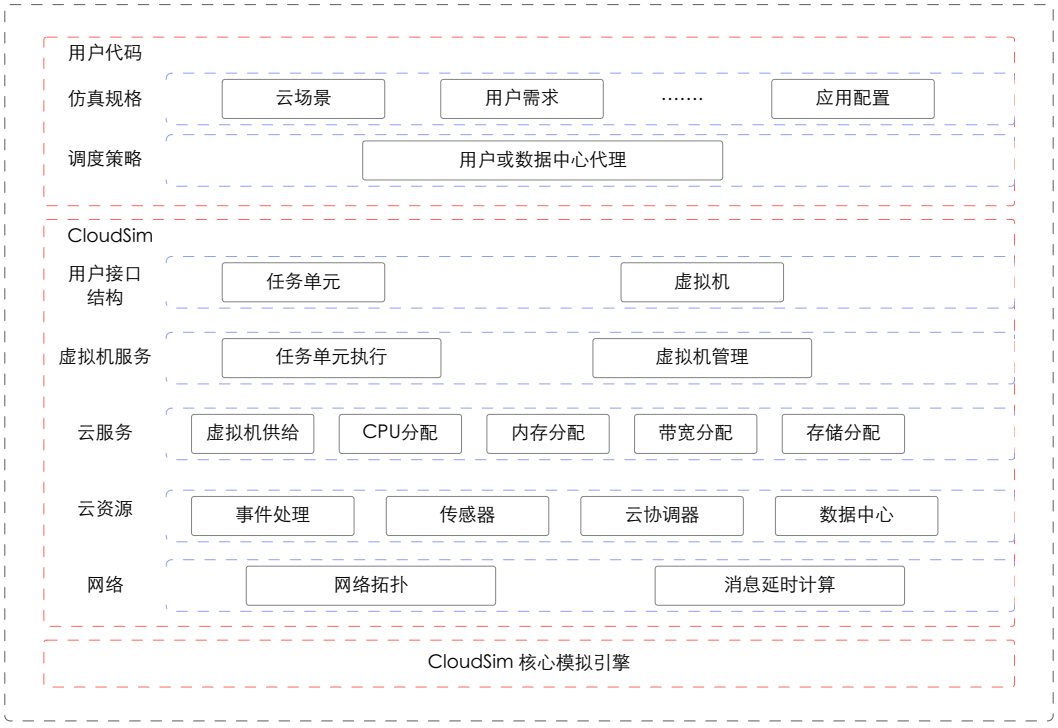



图 5.1　CloudSim 分层架构图

CloudSim 是一个分层构建的体系结构，从下到上一次为 CloudSim 核心引擎层、网络、云资源、云服务、虚拟服务、用户接口结构、资源调度以及仿真规格层。下面一次介绍各层次的大致作用:

(1). CloudSim 核心引擎层是离散数据模拟引擎 SimJava，为上层提供系统组件构建如服务、数据中心、客户端、代理、虚拟机等，查询和时间处理、通信、模拟时钟等功能。

(2). Network 层模拟网络组件如资源收集、数据集、负载测试、信息服务等，对网络设施建模，支持高层软件组件。

(3). 云资源层主要是 Host 主机和数据中心，主机的核心硬件设施通过数据中心类建模，处理服务请求，构成虚拟资源池。

(4). 云服务层给客户端分配特定应用的 VM，同时给 VM 分配处理内核、内存、磁盘以及网络带宽，可以执行用户新的 VM 提供策略，有助于一定目标优化。

(5). 虚拟机服务层提供任务执行和虚拟机管理，定义一系列虚拟机创建、销毁、合并、迁移等操作管理，执行基于云环境的应用服务。

(6). 用户接口层向用户提供 VM 任务单元和虚拟机，将下层的虚拟资源打包成虚拟机提供给用户。

(7). 用户代码层是用户根据实际应用场景和需求，定制应用的规格和调度策略，将应用加入数据中心的代理中，按照资源调度策略进行调度。

CloudSim 有一些重要的类和核心概念，针对这些类的大致作用进行简单的介绍:

1. DataCenter 类封装底层的 Host 主机，提供虚拟化的资源，保证每个数据中心至少存在一台运行的 Host 主机，同时提供虚拟化网络并内置了一个调度组件，为虚拟机和主机分配 CPU、内存、网络带宽等资源。
2. DataCenterBroker 类是数据中心代理，负责虚拟机和云任务列表的提交。
3. VM 类是虚拟机类，运行在 Host 上，多个 VM 共享 Host 资源。
4. Cloudlet 类是云任务类，根据用户的设置构建云计算和调度任务。
5. VmAllocationPolicy 类是虚拟机分配策略类，该类实现了虚拟机分配给 Host 主机的调度策略，用户可以重写该分配策略。
6. CloudletScheduler 类实现多种分配策略，虚拟机内部应用共享处理器的策略，时间共享还是空间共享。

此外，CLoudSim 还有数据中心资源配置类 DataCenterCharacteristics、扩展虚拟机分配策略的主机类 Host、带宽分配策略类 BwProvisioner、模拟网络延时行为类 NetworkTopology、模拟存储区域网类 SanStorage、虚拟分配主存类 RamProvisioner、云协调器类 CloudCoordinator 等一些列重要的类，共同完成 CloudSim 功能。

#### 5.1.1.2　ContaienrCloudSim 容器云仿真平台

随着容器技术的迅猛发展，CaaS(Congtianer as a Service) 作为一种新型的服务模式变得越来越普遍。上述介绍的各种云计算仿真平台以及早期的 CloudSim 版本并不支持容器仿真，一个支持容器应用仿真的平台变得越加紧迫。为了缩短容器云上新方法的开发时间，墨尔本大学的研究人员利用 CloudSim 虚拟化的特点，在其基础上开发了 ContainerCloudSim，专门用于数据中心容器应用的模拟。在最新的 CloudSim-4.0 上已经集成 ContainerCloudSim，提供 Docker 容器应用仿真支持，ContainerCloudSim 与 CloudSim 生态圈关系如图 5.2 所示。

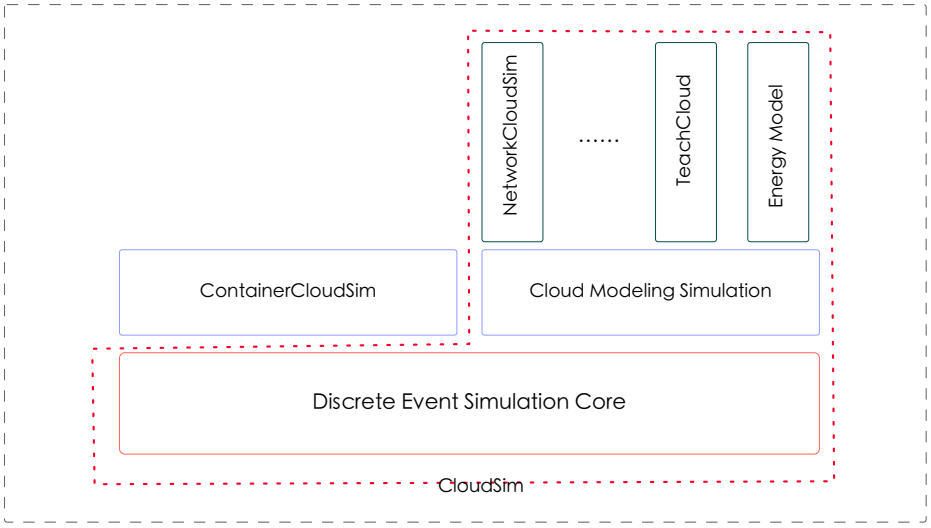



图 5.2　ContainerCloudSim 与 CloudSim 生态圈

集成 ContainerCloudSim 的 CloudSim-4.0 云平台模拟器已完整支持容器云的仿真，新的版本具有如下几个特点:

(1) 支持数据中心大规模云计算建模和仿真。

(2) 支持服务器虚拟化和主机的建模仿真，定制虚拟机调度策略。

(3) 支持容器应用程序的建模和仿真。

(4) 支持能量感知的计算资源建模和仿真。

(5) 支持网络拓扑结构和消息传递应用建模和仿真。

(6) 支持动态插入元素、停止和恢复的模拟。

(7) 支持混合云的建模和仿真。

在 ContainerCloudSim 部分，提供容器、VMs、Host、数据中心资源包括 CPU、内存和存储的管理功能，实现动态监控系统性、控制容器内应用的执行以及给容器提供虚拟机。容器的模拟器要能给研究人员提供容器调度方案间的对比，容器

调度策略决定容器如何被调度到虚拟机上，以及各种调度算法之间的对比和评测。算法的能耗问题也是容器模拟器应该关注的重点，可以提供各种算法的能耗度量，容器的合并和迁移也是模拟器的一大功能。最后，模拟器要能够支持容器的扩展性，在 CaaS 容器环境中，容器的数量是远远多于虚拟机的。ContainerCloudSim 是在 CloudSim 基础上开发而来，也是一个分层的架构，整体架构如下所示。

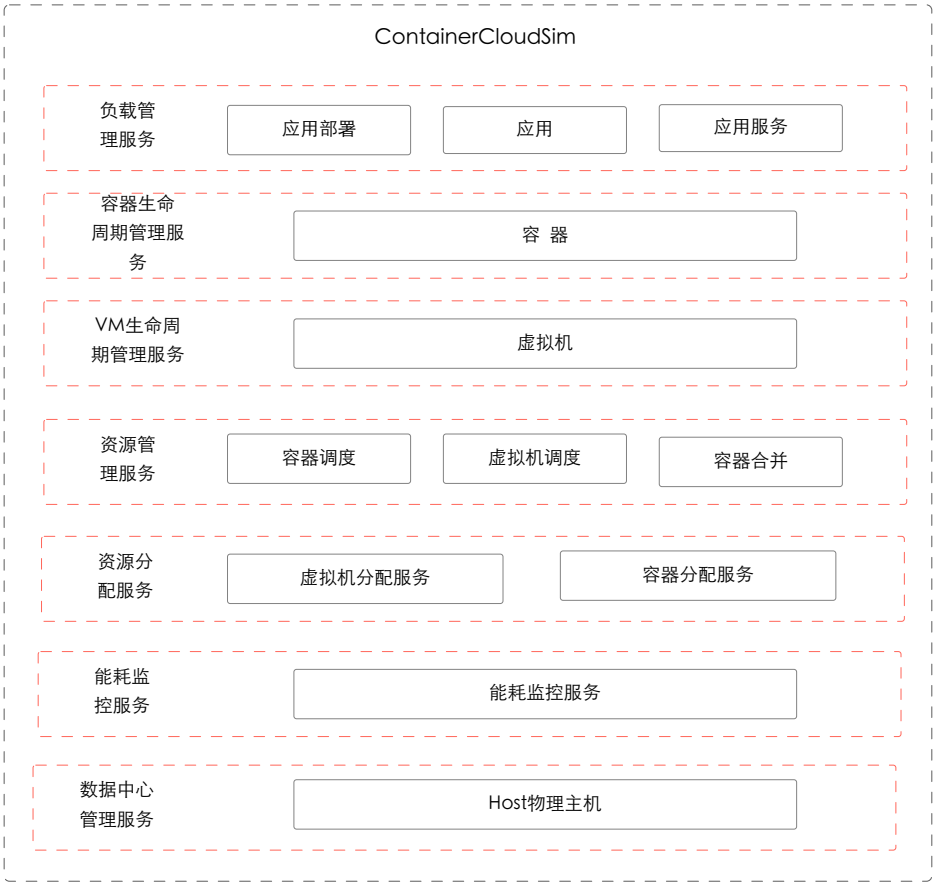



图 5.3　ContainerCloudSim 分层架构图

从底至上依次分为数据中心管理服务、能耗监控服务、资源分配服务、资源管理服务、虚拟机生命周期管理服务、容器生命周期管理服务和负载管理服务等。每个层次的大致作用如下:

1. 负载管理服务层关注于客户端应用的注册、部署、调度、应用层级的性能以及应用健康监控。
2. 容器生命周期服务管理层负责容器生命周期管理，包括创建容器、注册容器到系统中、启动、停止、重启、从一个主机迁移到另一个主机以及容器销毁。除此之外，还负责执行和管理在容器中任务，监控任务资源利用率。
3. 虚拟机生命周期管理服务层负责虚拟机的管理包括创建、启动、停止、重启、

销毁、迁移以及资源利用率监控。

4. 资源管理服务层负责容器在满足资源需求和软件环境的虚拟机上创建，虚拟机在满足资源需求的主机上创建，由容器调度、虚拟机调度和合并服务构成。容器调度根据容器调度策略调度容器到虚拟机、虚拟机调度根据虚拟机调度策略调度虚拟机到主机，合并策略通过合并容器减少主机需求，最小化资源碎片。
5. 资源分配层服务管理虚拟机和容器的资源分配，由容器分配服务和虚拟机分配服务构成。容器分配服务负责虚拟机资源分配给容器，虚拟机分配服务负责主机资源分配给虚拟机。
6. 能耗监控服务负责数据中心主机能耗监控，构建必要的能耗模型。
7. 数据中心管理服务负责管理数据中心资源，主机开关机以及监控资源利用率。

至此，ContainerCloudSim 容器云仿真平台介绍完毕，具体的仿真执行流程具体的实现可以参见其源代码。

### 5.1.2　MRWS 资源利用率实验

在 ContainerCloudSim 容器云仿真平台上的 ContainerPlacementPolicy 库下开发 MRWS、Kubernetes 默认 Default 算法，该类自带 Ramdom 算法，对比三种算法下集群资源的利用率。该容器云仿真平台影响调度因素较多，需要根据实际的需要进行一些必要的设置。

首先 MRWS 算法和 Kubernetes 的 Default 算法在预选阶段使用的筛选规则相同，评分阶段除空闲资源评分函数和平衡函数外，其他评分函数相同。因此，假设其他评分函数和外部条件都相同的情况下，只需模拟比较 MRWS 算法和 Default 算发的空闲资源评分函数。在模拟器设置方面，要对比容器各种调度算法下集群资源的利用率和负载均衡性，对虚拟机的调度策略已有众多的研究，不是本文的研究方向，因此，假设 ContainerCloudSim 数据中心的单个 Host 上只运行一个虚拟机，且虚拟机的资源配置和 Host 主机相同。本文也不对 Container 的迁移算法做研究，需要在模拟器上设置容器禁止迁移，即不触发迁移阈值。由于只对容器应用调度研究，不执行具体的云任务，不设置容器、虚拟机和主机的 PE 数。测试负载是基于 PlanetLab 负载进行的更改，让 CPU、内存、磁盘和带宽的利用率从 10% 到 90% 的随机变化。虚拟机、主机的配置如表 5.1 所以，CPU 的单位是 Mips，容器应用根据负载情况随机构建。在云数据中心中，集群通常是异构的，节点拥有资源数量不同，假设存在三种主机，用于模拟异构集群。

表 5.1　主机和虚拟机配置表

| Host 类型 | VM 类型 | CPU 型号 | MIPS | 内存 (G) | 磁盘 (G) | 带宽 (M) |
|---|---|---|---|---|---|---|
| #1 | #1 | i7 7500U | 49360 | 16 | 1000 | 100 |
| #2 | #2 | i5 8200U | 65770 | 32 | 1000 | 100 |
| #3 | #3 | X6 1100T | 78440 | 16 | 1000 | 100 |

容器应用的配置根据模拟的负载情况自动生成，并对生成的负载自动求解权重参数，参数设置部分代码片段如下：

```
public static final int VM_TYPES = 3;
public static final double[] VM_MIPS = new double[]{49360, 65770,
    78440};
public static final int[] VM_PES = new int[]{};
public static final float[] VM_RAM = new float[] {(float)16000,
    (float) 32000, (float) 16000};//**MB*
public static final int VM_BW = 100;
public static final int VM_SIZE = 1000000;

public static final int HOST_TYPES = 3;
public static final int[] HOST_MIPS = new int[]{49360, 65770,
    78440};
public static final int[] HOST_PES = new int[]{};
public static final int[] HOST_RAM = new int[]{1600,3200,1600};
public static final int HOST_BW = 100;
public static final int HOST_STORAGE = 1000000;

public static final int NUMBER_HOSTS = 30;
public static final int NUMBER_VMS = 30;
public static final int NUMBER_CLOUDLETS = 200;
```

模拟实验中设置资源阈值为 0.15，一旦节点上的某种资源使用率超过该阈值后将不能部署新的容器应用，模拟实验主要测量在相同数量容器下需要服务器的数量。若节点资源利用越均衡，该节点可部署的容器应用数量越多，从而需要的节点数量越少。实验对比 MRWS 调度算法、Kubernetes 的 Default 算法简称为 KUB 算法、Random 算法以及 FirstFit 算法对资源的需求，后两种算法是 ContainerCloudSim 自带算法，只需要模拟 KUB 的内存和 CPU 均衡利用，MRWS 综合考虑 CPU、内存、磁盘、网络带宽以及已部署 Pod 应用因素的综合评分算法。模拟实验中单位资源组 $res = (cpu, memory, disk, bandwidth) = (10000Mips, 4000M, 100G, 10M)$，单个应用容器的负载是单位资源组的 10% ~ 90% 随机变化的资源需求，构建出的应用容器负载如表 5.2 所示。

表 5.2　N 个应用容器的资源配置

| 应用容器 | CPU(MIPS) | 内存 (M) | 磁盘 (G) | 带宽 (M) | Pod |
|---|---|---|---|---|---|
| 1 | 8500 | 680 | 34 | 9 | 1 |
| 2 | 4000 | 840 | 12 | 16 | 1 |
| 3 | 2000 | 3400 | 20 | 6 | 1 |
| 4 | 3400 | 1200 | 80 | 10 | 1 |
| ... | ... | ... | ... | ... | ... |
| N | 7600 | 600 | 30 | 4 | 1 |

根据 MRWS 的 FAHP 自动建模和按照资源需求比值的差值构建模糊成对比矩阵，自动求解容器应用各维度资源的权重参数，上述容器列表各维度资源对应的权重参数如表 5.3 所示。

表 5.3　应用容器资源对应的权重参数表

| 应用容器 | CPU(MIPS) | 内存 (M) | 磁盘 (G) | 带宽 (M) | Pod |
|---|---|---|---|---|---|
| 1 | 0.528 | 0.086 | 0.136 | 0.214 | 0.036 |
| 2 | 0.185 | 0.128 | 0.082 | 0.570 | 0.035 |
| 3 | 0.092 | 0.596 | 0.119 | 0.158 | 0.035 |
| 4 | 0.140 | 0.095 | 0.508 | 0.221 | 0.036 |
| ... | ... | ... | ... | ... | ... |
| N | 0.596 | 0.092 | 0.158 | 0.119 | 0.035 |

测试在相同容器应用数量的条件下所需虚拟机的数量，根据前面一个主机 Host 上只运行一个和主机配置相同虚拟机，因此虚拟机的数量也是 Host 主机的数量。实验开始时给定一定数量的 Host，三种类型 Host 数量相同，一旦出现所有主机都无法满足新的容器资源需求，同时增加三种类型 Host 一台。如 200 个应用容器时先给定 18 台主机，以后每次各类型增加一台，即每次增加三台主机，避免某一类型的主机数量过多，测试结果如表 5.4 所示。

表 5.4　相同容器数量下各算法所需主机数

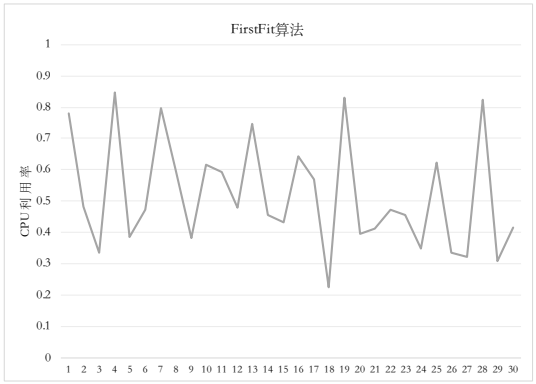
| 容器数 / 算法 | 200 | 400 | 600 | 800 | 1000 | 1200 |
|---|---|---|---|---|---|---|
| MRWS | 21 | 42 | 60 | 78 | 96 | 114 |
| KUB | 21 | 45 | 69 | 87 | 105 | 129 |
| Random | 24 | 48 | 72 | 93 | 117 | 139 |
| FirstFit | 27 | 54 | 81 | 108 | 138 | 162 |

从上表 5.4 可以得出，在应用容器数量集群规模较小的情况下，各种算法对主机数量需求相差不大，MRWS 调度算法的优势并不明显。这是由于集群中节点各维度资源过载现象较少，整体集群的资源利用率不也高。随着容器数量和主机数量的增加，Random 和 FirstFit 算法劣势越加明显，相同数量下所需的主机数量更多，这是由于随机调度和首个合适节点调度方法容易造成大量相同密集型的应用部署到同一个节点，从而使某一维度资源过载，不能部署更多的应用。Kubernetes 的 Default 算法考虑了 CPU 和内存的均衡性，其效果比其他两种算法要好，MRWS 调度算法综合考虑了各维度资源的均衡性，其效果最佳，所需主机数量最少。MRWS 还考虑节点已部署 Pod 因素，在资源利用率相差不大的情况下避免某个节点部署大量较小资源需求的容器应用，给节点容器管理造成过大开销，从而降低集群性能。容器应用应该分散和均衡地部署到集群的各节点上，提升集群服务性能。

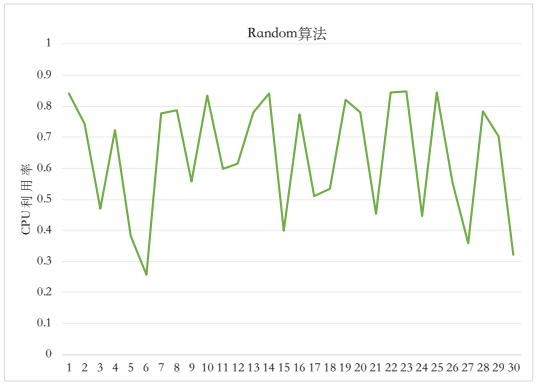
### 5.1.3　MRWS 负载均衡实验

集群服务性能一个关键问题就是负载均衡，而调度策略是影响负载均衡的核心因素。好的负载均衡策略不仅可以提升集群负载处理能力，缩短任务执行时间，也能有效避免单点故障。当前服务器的负载均衡算法一般有随机算法、轮询和加权轮询、最小连接和加权最小连接、哈希算法、IP 散列法以及 URL 散列等。一些重用的负载均衡组件有 Apache、Nginx、LVS(Linux Virtual Server)、HAproxy、KeepAlived、Memcached 等。Apache 和 Nginx 是一个 HTTP 的服务器，具有反向代理能力，通过对用户请求分流实现负载均衡；LVS 是一个虚拟的服务器集群系统，可以实现 Linux 平台下简单的负载均衡；HAproxy 提供高可用、负载均衡以及基于 TCP 和 HTTP 代理，适用于负载较大的 web 服务站点；KeepAlived 也是一个组件，用于检测服务集群中故障节点，及时处理不健康的节点；Memcached 是内存缓存系统，对于业务查询数据进行缓存，降低节点的负载，也是一个负载均衡组件。本文主要研究集群中各节点多维资源负载情况，目的在于新的调度方法是否具有更好的资源负载，各维度资源利用是否更加均衡。实验将从三个方面进行对比 MRWS 算法、Kubernetes 的 Default 算法、Random 算法和 FirstFit 算法中单个节点的相同维度资源、单个节点各维度资源均值、整个集群负载均衡性进行对比，评价指标如式 (3-10、3-11) 所示。测试过程中假定集群数量、应用容器数量不同的条件下上述三个指标的负载不均衡度。

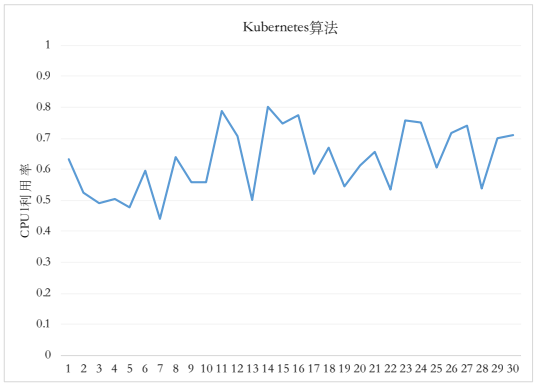
首先测试集群中各节点某个维度资源利用均衡情况，希望集群中节点上某个维度资源如 CPU 利用率更加平滑，若节点上该维度资源波动过大则表明其服务性能可能会降低，下面以 CPU 为例，其他维度资源类似。



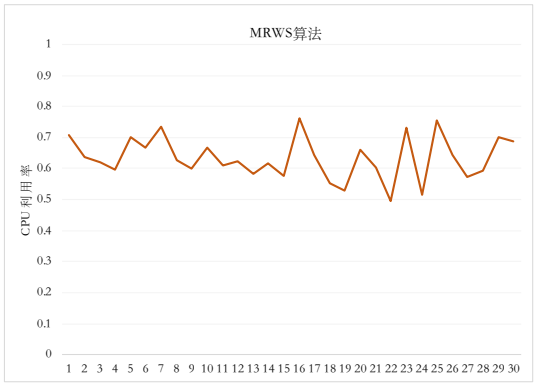

(a) FirstFit 算法 CPU 利用率




(b) Random 算法 CPU 利用率



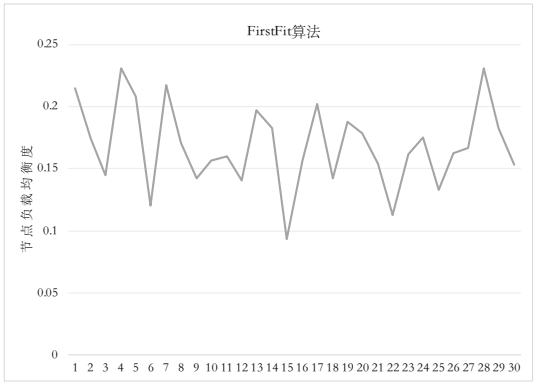

(c) kubernetes 算法 CPU 利用率



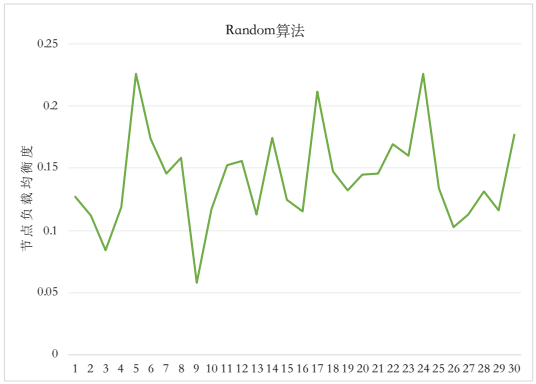

(d) MRWS 算法 CPU 利用率

图 5.4　四种调度算法下节点 CPU 利用率波动图

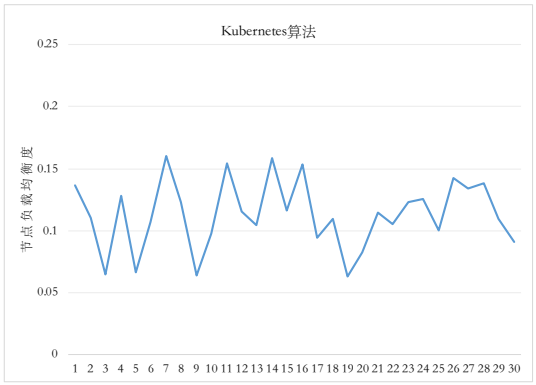
从图 5.4 看出，MRWS 调度算法下 CPU 利用率波动性较小，Kubernetes 调度算法次之，Random 和 FirstFit 算法节点 CPU 利用率波动较大。这是由于前两种算法都考虑了 CPU 利用的均衡性，其中 Kubernetes 在进行节点评分时考虑了 CPU 和内存平衡利用，MRWS 调度算法综合考虑了 CPU、内存、磁盘、网络带宽和已部署的 Pod 的因素。有效避免单个节点某个维度资源资源利用过高，出现过载现象，从而不能调度更多的应用，造成其他维度的资源浪费，MRWS 算法能够有效提升集群的服务性能和资源利用率。下面的实验使用之前定义的均衡度对集群中单个节点上资源均衡度进行测量，实验中假设应用容器数量相同，分别用四种算法对其进行调度，计算单个节点的负载均衡度，部分节点的负载均衡度如下。



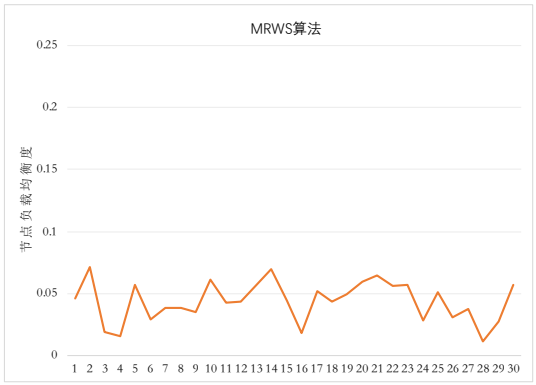

(a) FirstFit 算法节点负载均衡度




(b) Random 算法节点负载均衡度




(c) kubernetes 节点负载均衡度

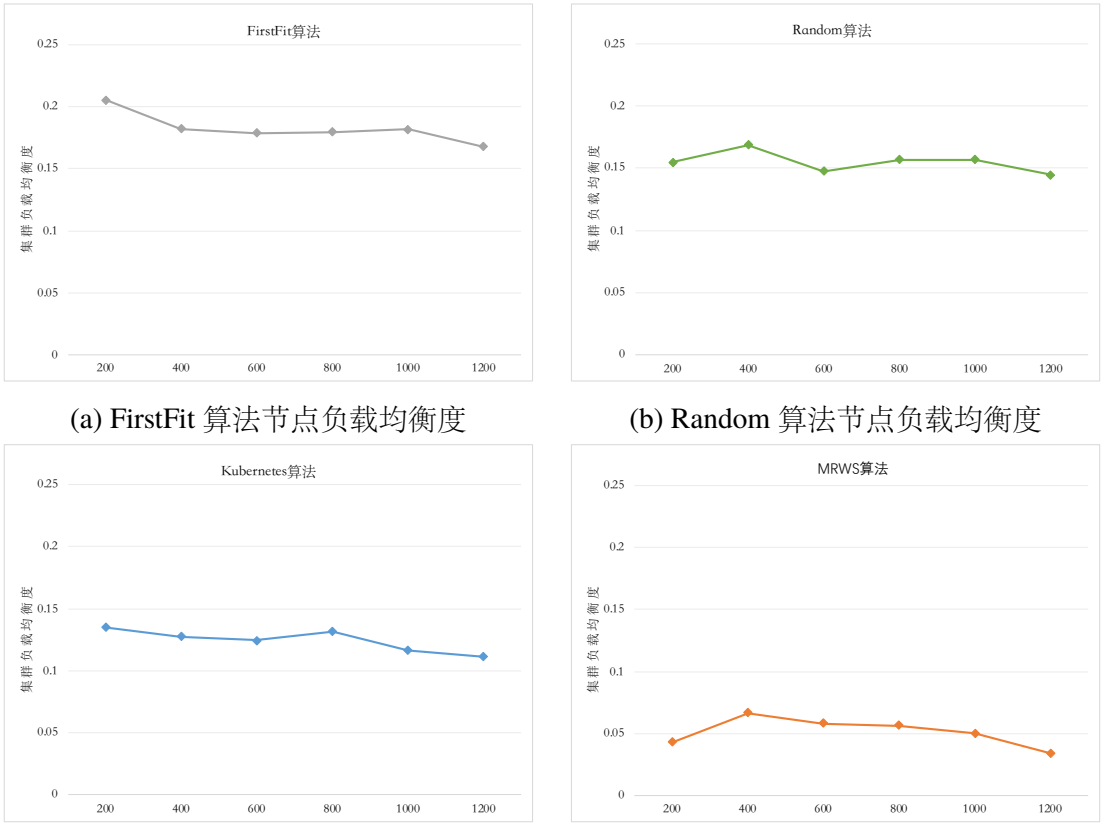



(d) MRWS 算法节点负载均衡度

图 5.5　四种调度算法下节点负载均衡度

如图 5.5 所示，四种调度算法中 MRWS 算法的负载均衡度最小，且波动也最小，Kubernetes 算法次之，Random 和 FirstFit 算法节点的负载均衡度波动最大并且节点负载均衡度较大。负载均衡度反映节点上各位资源空闲率的均衡程度，负载均衡度越大，各位资源消耗越不均衡，某个维度资源过载现象越严重。由于 Random 算法随机选择可用节点，FirstFit 算法每次选择第一个满足资源需求的节点，完全不考虑各维资源的空闲情况，导致其负载均衡度较大，节点资源利用不充分。Kubernetes 平衡 CPU 和内存利用率，其效果较另外两种要好，MRWS 综合了各维度资源空闲率进行评分，其效果最佳，不仅负载均衡度整体偏小，其波动性也小。

接下来进行四种调度算法下集群负载均衡度的对比，假定在容器应用数量不同的情况下，比较整个集群的负载均衡度情况。应用数量相同，负载均衡度越小，集群中过载的节点数量越少，相同情况下集群服务性能会越好，各维度资源消耗更加均匀，应用的调度更合理。




(a) FirstFit 算法节点负载均衡度　(b) Random 算法节点负载均衡度

(c) kubernetes 节点负载均衡度　(d) MRWS 算法节点负载均衡度

图 5.6　四种调度算法下集群负载均衡度

如图 5.6 所示，MRWS 调度算法集群的负载均衡度最小，大概为 Kubernetes 算法的二分之一，Ramdom 和 FirstFit 算法的四分之一。因此，在 MRWS 调度算法的集群中各节点资源消耗情况趋于均衡，单个节点上各维度资源利用也较为均衡，集群的服务性能更优秀。针对集群的服务性能将使用多计算框架应用混合部署在实际的实验平台上进行测试，测试集中调度算方法下混合应用的完成时间。

## 5.2　Paladin 大数据存储与处理的容器云平台部署

由于不具备大规模实际容器云环境对几种调度算法进行测试，将在实验室小规模容器云平台 Paladin 上对算法性能进行测试，实验中使用多种分布式计算框架容器应用进行混合部署和性能测试。因此，首先需要搭建一个 Paladin 实验环境，Paladin 是实验室开发的一个集海量数据存储管理，基于容器的多种分布式计算框架运行环境快速构建，存储和计算分离的多元化大数据数据处理平台。

### 5.2.1　Paladin 大数据存储与处理的容器云平台介绍

Paladin 是一个支持用户构建多种大数据计算框架、并提供海量数据存储与管理的平台。用户在平台中选择所需的计算与存储框架，如批处理 (Hadoop、Spark)，流计算系统 (Storm、Flink)、分布式的 MPI、机器学习框架 (Mahout、Tensorflow)、图计算系统（GridGraph、ReGraph) 以及数据库系统 (MongoDB、Mysql、Redis) 等等。平台能够快速构建相应的多计算框架运行环境、并且根据具体的需求提供计算任务调度、海量数据存储，通过优化的容器云多维资源利用率均衡调度算法提升资源利用率和集群服务性能。

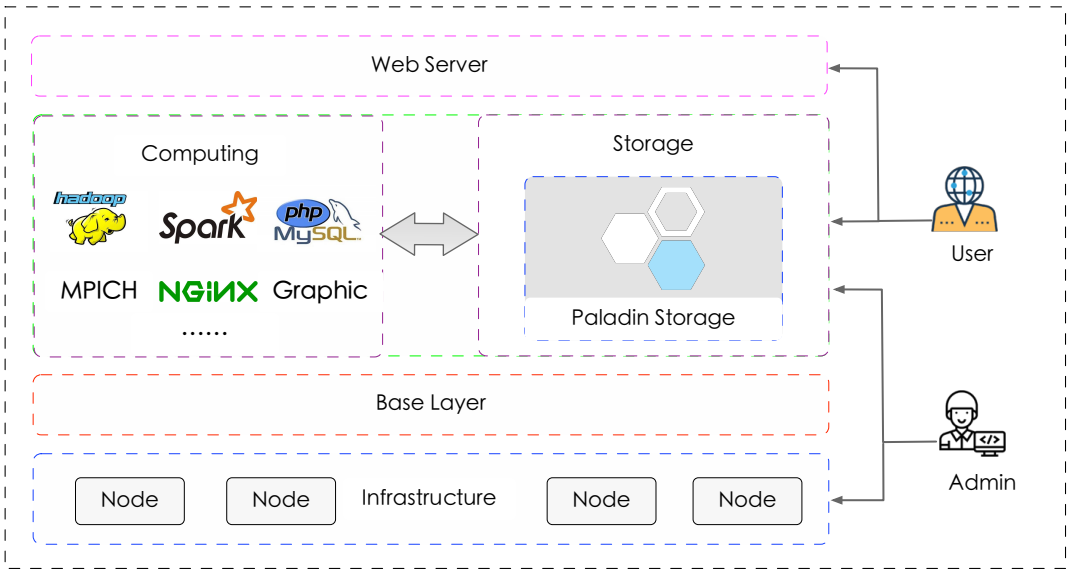



图 5.7　Paladin 平台架构图

从 Paladin 架构图可以看出，最底层是基础设施部分，即整个集群的物理机和云主机部分，中间是支持上层的中间软件服务如分布式文件系统、Docker 容器服务、OpenShift Origin 容器云等，最上层提供海量数据存储管理和多种数据处理计算框架，顶层是 Web 服务层。整个架构中最为重要的两个部分数据存储管理模块和数据处理框架模块，数据管理存储提供用户数据上传、下载、管理、浏览、分享、挂在等一个分布式大数据管理存储系统。数据处理框架提供用户快速构建各种计算框架容器集群环境，根据实际需求配置资源、容器的动态增减变动等，该部分基于开源 OpenShift Origin 进行搭建，在任何一个容器中可以实现存储数据的挂载，实现存储和计算分离。当前计算框架除支持单应用的 Nginx、Apache、PHP、MongoDB 等，也支持大量分布式数据处理框架如 Hadoop、SPark、Storm、Spark、TensorFlow 以及实验室的图计算 Regraph、GridGraph 等数十种计算框架运行环境的快速部署。用户既可以简单用该系统进行大数据存储管理，也能快速构建大数

据处理框架，实现海量数据处理，管理员对基础设施和存储节点，以及系统权限等进行管理。

### 5.2.2　Paladin 大数据存储与处理的容器云平台部署

首先需要构建小规模的 Paladin 数据存储处理平台，构建一个容器云环境，该平台集成数据存储和管理。实验中采用 6 台物理机进行部署，一个主节点 Matser，Master 节点同时也是 Node 节点，四个 Node 节点和一个存储 Master 节点，由于该存储也需要对用户提供 Web Service 的服务，因此单独使用一个节点进行部署。将用户的存储管理服务和计算框架服务分离，分别部署在两个不同的物理机上，提升集群吞吐率。S-Master 是分布式存储的 Master 节点，各物理节点的配置如下:

表 5.5　Paladin 平台物理机配置

| 资源 / 节点 | CPU 核数 | 内存 (G) | 磁盘 (G) | 带宽 (M) | IP |
|---|---|---|---|---|---|
| Master | 4 | 16 | 1000 | 100 | 192.168.1.100 |
| Node1 | 4 | 16 | 1000 | 100 | 192.168.1.101 |
| Node2 | 4 | 16 | 1000 | 100 | 192.168.1.102 |
| Node3 | 4 | 16 | 1000 | 100 | 192.168.1.103 |
| Node4 | 4 | 16 | 1000 | 100 | 192.168.1.104 |
| S-Matser | 4 | 8 | 500 | 100 | 192.168.1.110 |

各节点上需要安装的主要软件如下图所示，部署一个完整的容器云平台。在分布式计算框架部分 Master 节点上主要部署 openshift-ansible 及其依赖的各种软件包，用于安装 OpenShift 集群、openshift-origin 集群、底层的 Docker 及其私有仓库、openshift-origin 以来的一致性存储 Etcd、实验室数据存储管理需要的哭护短 paladin-client、经过优化更改后的 origin-web、操作系统 Centos 7.5。Master 节点即作为容器云的主控节点，同时也是 Node 节点，在分布式存储中作为数据存储节点。S-Mastershi 是分布式存储和管理的主控节点，使用 Centos 6.9 系统，安装 Nginx 服务、paladin-web-console 对普通用户提供数据存储管理的 web 服务，同时也是数据节点。Node 节点作为计算节点，安装 Docker、openshift-origin 集群子节点的需要的依赖软件如 kublet、Haproxy 等系列软件、同时还有 Etcd 和 paladin-client、作为存储集群的数据节点，系统是 Centos7.5。

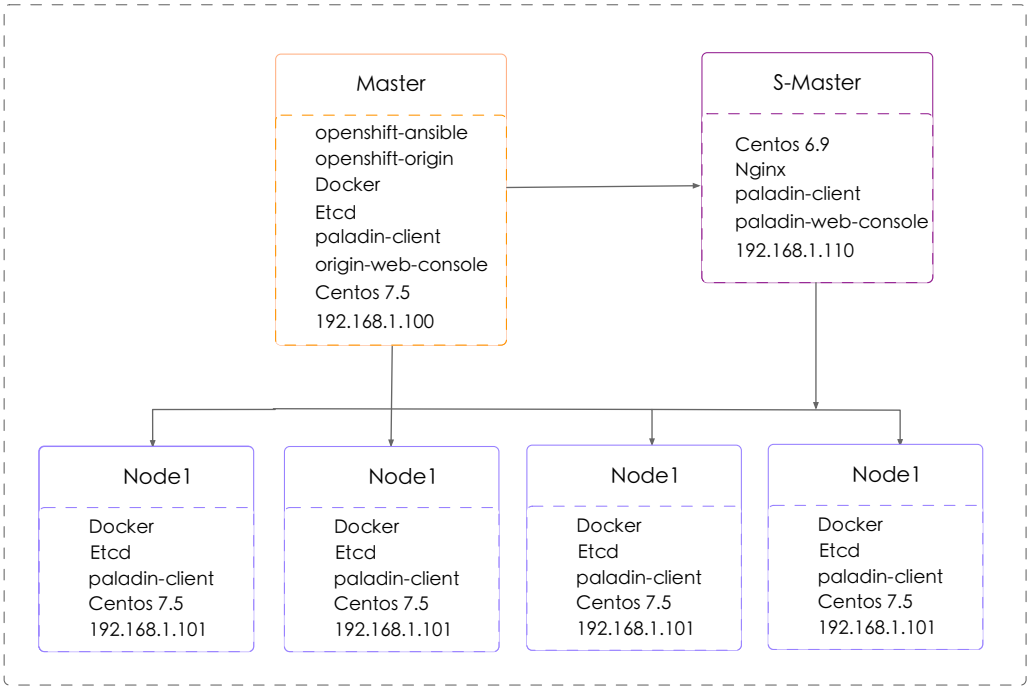



图 5.8　Paladin 平台节点分布图

安装完成后，Paladin 平台的计算框架部分，查询集群活跃节点如下:

```
[root@master images]# oc get nodes
NAME                STATUS    AGE       VERSION
master.example.com  Ready     1y        v1.7.6+a08f5eeb62
node1.example.com   Ready     1y        v1.7.6+a08f5eeb62
node2.example.com   Ready     1y        v1.7.6+a08f5eeb62
node3.example.com   Ready     1y        v1.7.6+a08f5eeb62
node4.example.com   Ready     1y        v1.7.6+a08f5eeb62
[root@master images]# oc get project
NAME                                DISPLAY NAME   STATUS
assignpods                                         Active
default                                            Active
hadoop-project                                     Active
kube-public                                        Active
kube-service-catalog                               Active
kube-system                                        Active
logging                                            Active
management-infra                                   Active
mpi-project                                        Active
openshift                                          Active
openshift-ansible-service-broker                   Active
openshift-infra                                    Active
openshift-node                                     Active
openshift-template-service-broker                  Active
spark-project                                      Active
[root@master images]#
```

图 5.9　Paladin 安装成功计算节点部分

存储 paladin-web-console 部分，用户进行数据存储管理，用户在该部分可以进行海量数据的上传、下载、删除等管理，还可以通过 paladin-client 将数据以盘的形式挂在到物理节点或者容器中，在分布式计算框架容器应用中直接使用数据，实

现存储和计算分离。

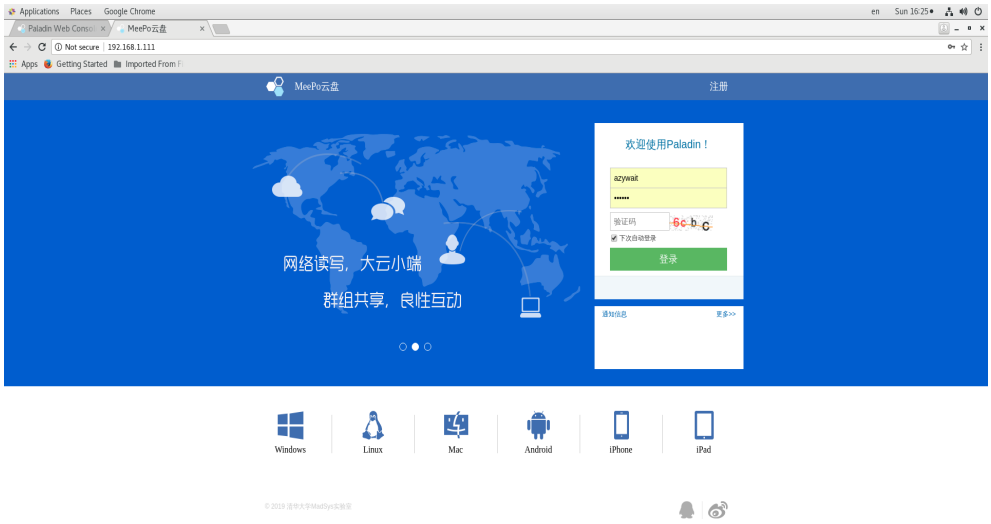



图 5.10　Paladin 数据存储管理服务

整个 Paladin 平台部署完成后，在该平台上开发部署各种大数据处理框架和容器应用，平台 origin-web-console 用户服务界面如下:

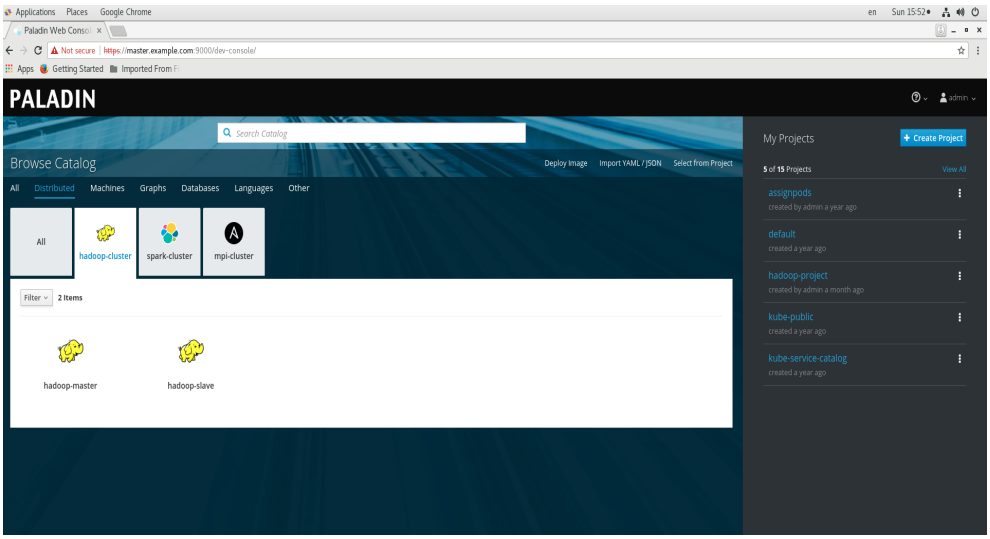



图 5.11　Paladin 数据处理框架部署服务

至此，整个 Paladin 大数据存储管理和计算框架快速部署容器云平台部署完成，实现了存储计算分离，用户既能进行大数据存储管理，实现普通的云存储系统，又能快速构建大数据处理容器应用框架，实现海量数据处理，为普通用户提供优秀的大数据容器云配套服务。

## 5.3　多计算框架容器应用开发

在构建分布式计算应用时，容器数量动态增减变化时需要能自动发现其数量的变化，并自动搭建好计算环境。如在 Hadoop 容器应用计算框架下，有 10 个计算容器，减少 4 个后，应用框架要能够快速发现并自动缩减为 6 个容器应用的 Hadoop 计算环境。因此，分布式计算框架容器应用中需要内嵌一个服务自发现组件，当前分布式有 Zookeeper、Etcd、Serf 等，本文用 Serf 做服务注册和自发现组件。

### 5.3.1　Serf 服务发现

在分布式计算框架中，有的计算框架是主从结构如 Hadoop、Spark 等，有的只是简单的计算框架如 MPI 等，因此需要一个去中心化的服务发现方案。HashCorp 开源的 Serf 是一个去中心化的服务发现和编排项目，具有成员管理、服务检测、高可用、分区容错等功能，是一个轻量级的服务组件。Serf 底层实现 Gossip 协议，一种类似病毒感染的传播协议，能够快速自动感知节点上线和下线。构成分布式计算框架中的容器应用上都维护一个成员列表，并将节点列表随机发送给周围节点，发送方式有 Push 和 Pull 两种，即主动发送和被动请求方式。Serf 有一个时效检测器，定期轮询节点列表，给列表中的节点发送心跳包，确保所有的节点在线。

Serf 在分布式处理框架的每个应用容器上运行一个 serf agent，在第一个容器应用上创建一个聚簇，后面创建容器的代理将会自动连接已存在的聚簇，形成一个集群。Serf 是一个去中心化的服务方案，没有一个集中注册的节点，是一个弱一致性的方案，但具有较高的可用性和容错性。在常见的中心化集群中，集群的状态由 master 维护，通常为了可靠性需要双备份甚至多备份主节点，主节点往往是整个集群的性能瓶颈。在集群规模较大时，信息的传播给网络带来巨大的负载，集群的管理效率和正确性受到挑战，集群规模往往有一个上限。去中心化的集群可以进行大规模扩展，其效率和可靠性几乎不受集群规模的影响，Kubernetes 底层用 Etcd 做服务发现，其规模上限通常在一万左右，而 Docker Swarm 底层用 serf，其规模可以更大，Kubernetes 的 Etcd 将会是制约其集群规模上限的重要因素。

Serf 从功能上而言，自下而上可以分为三层：Gossip 协议层、消息中间件、客户端接口。Gossip 协议层封装 Gossip 协议以及集群管理的相关操作；消息中间件对传播消息进行封装，包括管理的消息、UserEvent、Query 等各种处理逻辑，其中 UserEvent 是一个单向无需应答的消息，Query 是需要应答的双向消息；客户端接口处理集群的输出输出格式，提供 rpc 接口。Serf 是一个弱一致性的解决方案，让节点之间直接进行信息交流，提升软件平台服务自主性，是一个简单、高效、可靠的去中心集群管理和服务发现系统实现。

### 5.3.2　Hadoop 计算框架开发

在 Paldin 容器云平台计算框架部分，需要开发多种分布式处理框架，方便用户快速构建大数据处理环境。用户通过 web-console 与平台进行交互，只需通过鼠标操作就能构建所需的计算环境和配置容器资源，实现快速自动化部署。计算框架最终以模板 Template 存在集群中，所需的镜像文件在 Docker 私有仓库 registry 中，所有用户都重复使用该模板，其构建过程如下所示。

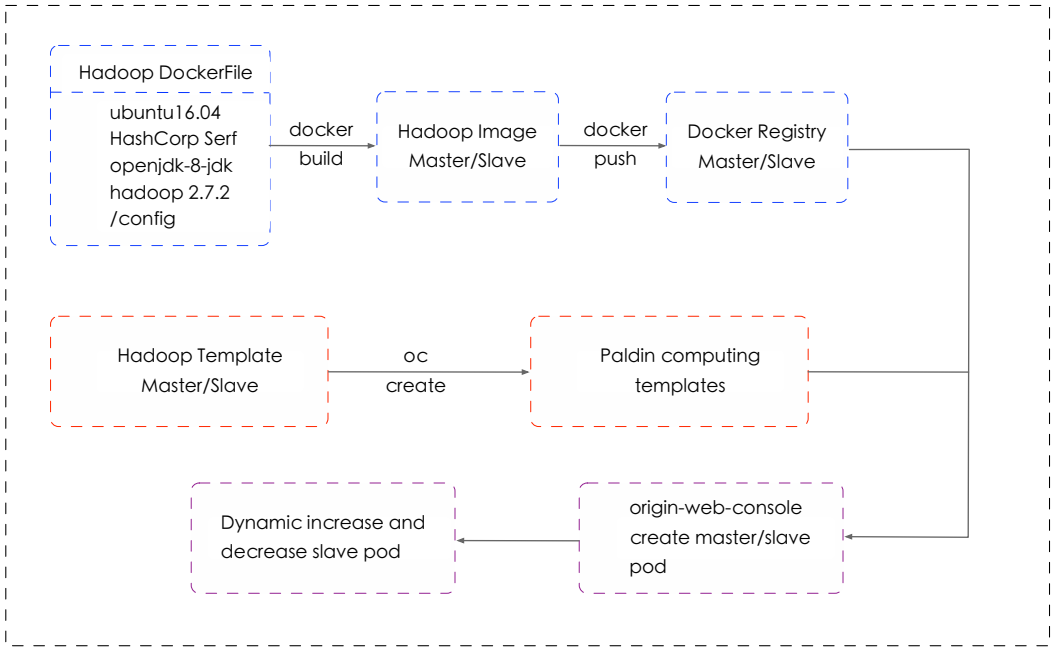



图 5.12　Hadoop 容器处理框架开发流程图

首先构建 Hadoop 环境容器应用的镜像文件，编写 Hadoop Master 和 Slave 的 DockerFile 文件，该文件在操作系统镜像 ubuntu16.04 的基础上安装 Hadoop 所需的环境，主要包括 openjdk-8-jdk、hadoop-2.7.2、wget 等，用于配置免密 dnsmasq、openssh-server 以及服务自发现 Serf，将 Hadoop 所需的配置文件一并修改，整体打包成镜像，部分 DockerFile 源码如下:

```
FROM ubuntu:16.04
MAINTAINER gongkunjxl <gongkunjxl@163.com>
RUN sed -i 's/archive.ubuntu.com/mirrors.ustc.edu.cn/g'
   /etc/apt/sources.list
RUN apt-get install -y unzip curl dnsmasq openssh-server
RUN apt-get install wget vim openjdk-7-jdk fuse libglib2.0-0
# dnsmasq configuration
ADD dnsmasq/* /etc/
# install serf
RUN curl -Lso serf.zip
   https://releases.hashicorp.com/serf/0.5.0/serf_0.5.0_linux_amd64.zip
```

```
RUN unzip serf.zip -d /bin
RUN rm serf.zip
#ADD hadoop-2.7.2.tar.gz /root/
RUN wget XXX/hadoop-2.7.2.tar.gz(根据实际需要选择版本)
RUN tar -xzvf hadoop-2.7.2.tar.gz
RUN mv hadoop-2.7.2 /usr/local/hadoop
RUN rm hadoop-2.7.2.tar.gz
```

此外，还有一些修改的配置文件和样例文件，以及 paladin-client 挂在文件系统的客户端等，使用 docker build 命令全部打包成容器 Image 文件，将文件 push 到私有仓库中，平台私有仓库的 Ip 为 172.30.7.23:5000，在以后的 template 文件中将使用该镜像文件源。为重复使用该镜像文件以及用户通过 web-console 快速构建 Hadoop 计算框架，需要做成模板放置在平台上，其模板小部分代码如下:

```
{
"kind": "Template",
"apiVersion": "v1",
"metadata": {
"name": "hadoop-master",
"annotations": {
"description": "Hadoop Master",
"iconClass": "icon-hadoop",
"tags": "hadoop"
}
 "objects": [
{
"kind": "Service",
"apiVersion": "v1",
"metadata": {
"name": "${APP_SERVICE_NAME}",
},
.......
```

该模板包含一个 Service、一个 DeploymentConfig，部署弯沉过后可以实现 Container 的自动伸缩。根据容器编排引擎 Kubernetes 底层网络原理，两个 Pod 之间可以实现直接 IP 互相访问，同一个 Namespace 下的 Pod 可以通过网桥直接通信。为了更好的实现对外服务，Pod 的 Service 服务将其 IP 地址内嵌到后来创建的同一 Namespace 下的所有 Pod 中，但在创建 Service 之前的 Pod 中不会嵌入 IP，这是实现 Pod 构成集群的最好方式。利用网络管理的这一特性，使用 Serf 对第一个容器应用创建聚簇，后面容器的 Serf 代理通过第一个容器的内嵌 IP 实现集群发现服务。如第一个容器应用是 hadoop-master，由于其创建了 Service 服务，后面创建的所有 hadoop-slave 容器中都会有 $HADOOP_MASTER_SERVICE_HOST 环境变量为 hadoop-master 的 IP 地址，实现集群的构建。通过 web-console 可以快速构建 Hadoop 处理环境，并根据资源需求实现容器的自动伸缩，按需配置服务。

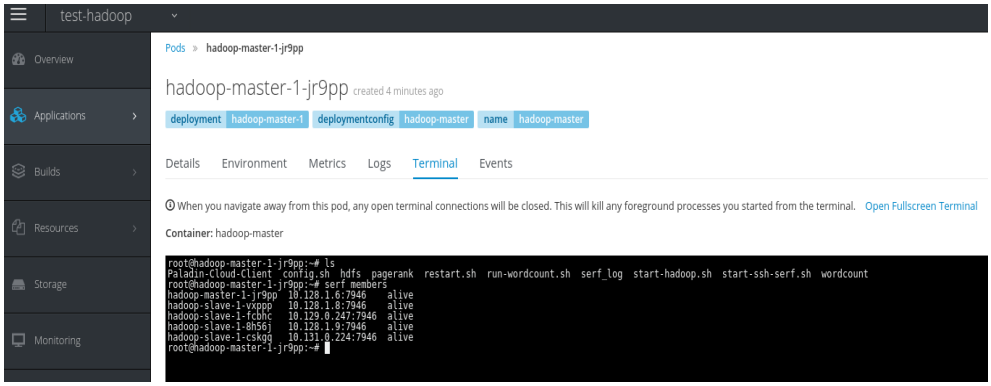



图 5.13　Hadoop 集群成员

同样可以通过 hadoop-master Pod 的 IP 地址进行访问，在浏览器中访问 Pod 的 50070 和 8088 端口，查看 Hadoop 集群状态和运行任务:

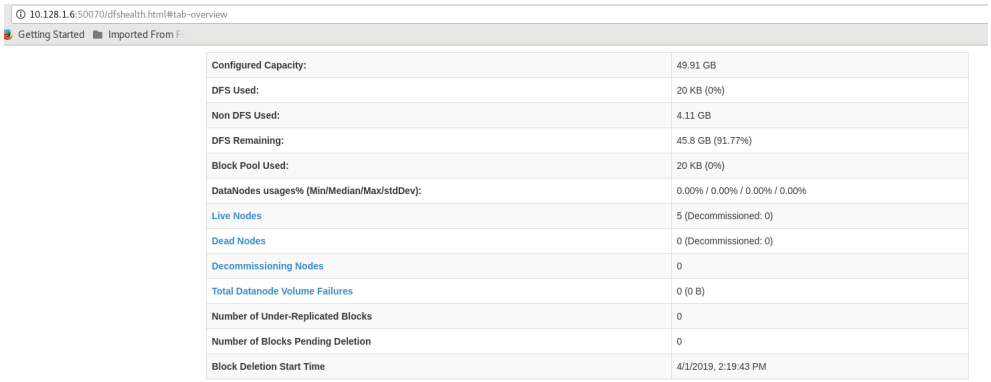



图 5.14　Hadoop 集群状态信息

在构建的 Hadoop 集群上运行 wordcount 样例，通过 8088 端口查看任务信息:

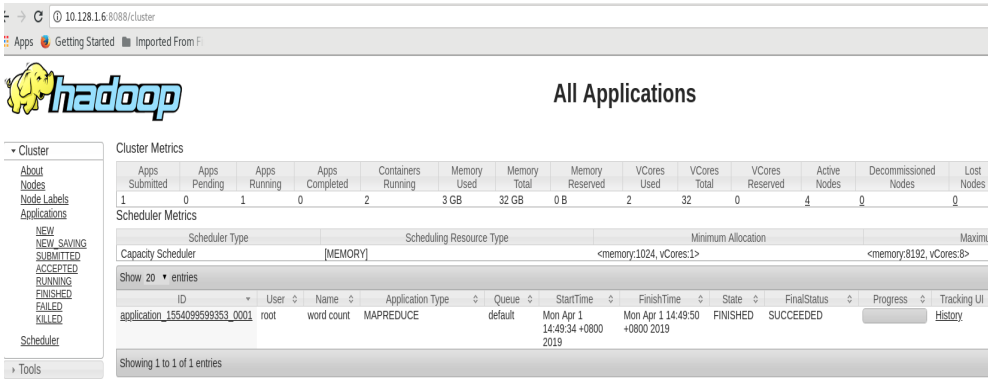



图 5.15　Hadoop 任务状态信息

至此，一个完整 Hadoop 容器应用处理框架构建完成，用户可以非常方便的在

web-console 上实现 Hadoop 环境的快速部署，并根据资源需求动态伸缩 Pod 数量，实现容器云平台的按需服务，并通过 web 查看器任务和集群状态。

### 5.3.3　其他计算框架开发

上一小节详细介绍了 Paladin 平台上 Hadoop 分布式处理框架的开发流程，使用上述的开发流程，构建其他的数据处理框架。原始的 Docker hub 中镜像文件大多只支持单个容器应用，即一个容器机器复制，比如 PHP、Go、Java、Ruby、Python 等语言类容器应用，以及 Mysql、MongoDB、MariaDB 等数据存储类容器。多分布式的数据处理框架和计算框架几乎没有，因此我们开发了离线数据批处理 Hadoop、Spark、流计算 Storm、Flink、分布式 MPI、机器学习框架 Mahout、Tensorflow、图计算系统 GridGraph、Regraph 等。众多的分布式处理可以满足绝大部分用户的需求，使得该平台将大数据存储管理和数据处理框架有机集合，并实现存储计算分离。底层基于 Docker 容器应用，集群整体的性能比虚拟机要优秀，减少了虚拟机系统隔离的开销。该平台上构建的分布式模板如下:

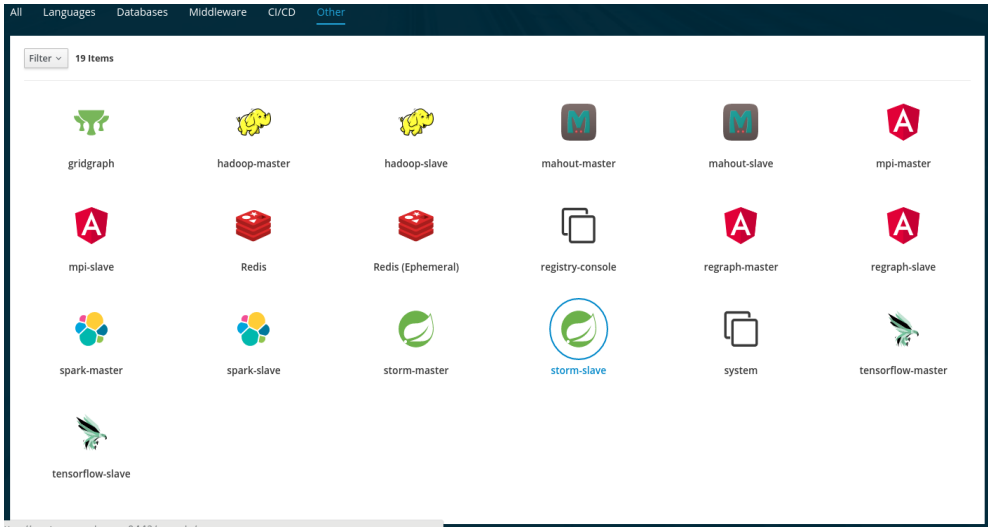



图 5.16　Paladin 平台支持的处理框架模板

用户选择需要构建的处理框架，创建其第一个容器应用及其 Service，容器应用内部的 Serf 同时创建集簇，后创建的容器应用通过 Service 提供的 IP 用其 Serf 代理自动连接集簇，构建一个集群。根据实际的资源需求，用户动态调整 Pod 数量，伸缩其资源，实现按需服务。容器应用中的 Serf 代理会自动监控容器应用的伸缩，用户调整完 Pod 数量后无需重启或重新连接集群，Serf 服务发现解决方案很好解决分布式框架容器应用上线下线问题。Paladin 平台给用户提供了一个 web-console

的可视化服务，用户无需了解部署原理和细节，甚至无需关心其构建过程，只需要根据实际需要配置处理框架的容器数量，几秒钟内就可以搭建好开发测试环境。用户把所有的精力集中在应用的开发生，完全从集群和环境配置中解放出来。该平台同时还提供数据存储管理可视化，可以挂载到所有的容器应用中，无需用户手动管理和数据节点分配，全自动化的存储计算分离。

## 5.4　多计算框架应用下 MRWS 性能测试

搭建完容器云数据管理存储和多计算框架容器应用环境快速构建平台后，需要对 MRWS 调度性能进行测试，对比 MRWS、Kubernetes、Random、FirstFit 四种调度算法的性能。由于是小规模的容器云集群，无法对资源利用率进行大规模测试，大规模的仿真在 ContainerCloudSim 已经完成，因此只进行多计算框架容器应用混合部署下几种调度方案的性能。

### 5.4.1　混合部署多计算框架

Paladin 平台只有五个实际的物理的节点，要在此小规模的集群上测试四种算法的性能，不能进行大规模的负载模拟。因此，构建集中较为典型的密集型应用容器，通过密集型应用的样例运行时间测试集中算法的优劣。Hadoop 作为离线数据批处理，在海量数据处理 map 阶段完成后不能将所有的数据都放入内存做 shuffle，要写入磁盘，其处理数据的方式大部分应用都是 I/O 密集型应用，因此用其数据读写样例作为 I/O 密集型容器应用，即 Hadoop 框架作为 I/O 密集型调度用例。Spark 处理框架的特点是将大部分数据放入内存运算，每次将 map 输出的结果保存在内存中，大部分应用是内存密集型应用，因此 Spark 的 PageRank 计算用作内存密集型应用。MPI 多用于分布式计算，网络和计算资源通常是其瓶颈，在 MPI 的 pi 计算是一种 CPU 密集型应用，因此用作调度测试用例。Paladin 集群上将混合部署 Hadoop、Spark 和 MPI 三种多计算框架，每个计算框架的容器数量相同，Hadoop 上运行 I/O 和网络密集型应用 TestDFSIO.java、Spark 上运行内存密集型应用支持向量机 PageRank、MPI 上运行 CPU 密集型应用 PI 计算。

表 5.6　用于调度算法测试的计算框架

| 框架类型 | 类型 | 运行程序 | 容器数量 |
|---|---|---|---|
| MPI | CPU 密集 | PI 计算 | 15 |
| Spark | 内存密集 | PageRank | 15 |
| Hadoop | I/O 密集 | TestDFSIO | 15 |

调度算法的开发，Paladin 是基于 OpenShift Origin 搭建的，OpenShift Origin 是一个开源的容器云平台，其底层的容器编排引擎是 Kubernetes。该平台只是搭建与 Kubernetes 上，并未修改其源码，因此开发该平台的调度方式和 Kubernetes 开发调度算法完全一样，最后只需要将调度算法的名称注册到平台的配置文件 origin-master.conf 中即可替代原有的算法。虽然 Kubernetes 提供了大量的 API，但其使用较为繁琐，整个引擎的语言都是 go 开发，因此使用 client-go 进行快速开发。client-go 是调用 Kubernetes 集群资源的客户端，对 Kubernetes 前置的 API 封装的二次开发都使用 client-go 这个第三方包实现，可以对集群资源对象进行增删查改。因此，四种调度算法也是使用 client-go 第三方包开发完成，实现与集群的交互。

进行调度算法测试前，需要构建三种计算框架的容器应用，并对容器应用进行资源配置。根据其运行应用的类型，在相应维度上进行适当对配置，CPU 密集型容器应用对 CPU 资源多配置，其他维度相对配置较少。为了测试混合部署的集群性能，不设置容器的资源使用上限，设置其资源使用下限，目的在于容器应用可以快速完成任务。如一个容器 CPU 限制使用 30%，如果物理节点上只存在一个容器，则该容器可以使用全部的 CPU 资源。主要在于测试多计算框架混合部署的场景下，几种调度算法对容器应用进行调度，整个集群完成全部计算框架处理任务的时间，该时间用于衡量集群性能，也就是调度算法的优劣。

表 5.7　Random 调度算法应用容器分布

| 节点 / 类型 | Master | Node1 | Node2 | Node3 | Node4 |
|---|---|---|---|---|---|
| MPI-pod | 3 | 5 | 2 | 4 | 1 |
| Spark-pod | 3 | 2 | 3 | 2 | 5 |
| Hadoop-pod | 1 | 4 | 5 | 2 | 3 |
| Total | 7 | 11 | 10 | 8 | 9 |

表 5.8　FirstFit 调度算法应用容器分布

| 节点 / 类型 | Master | Node1 | Node2 | Node3 | Node4 |
|---|---|---|---|---|---|
| MPI-pod | 5 | 2 | 5 | 2 | 1 |
| Spark-pod | 3 | 6 | 1 | 5 | 1 |
| Hadoop-pod | 2 | 5 | 3 | 5 | 1 |
| Total | 10 | 13 | 9 | 12 | 3 |

表 5.9　Kubernetes 调度算法应用容器分布

| 节点 / 类型 | Master | Node1 | Node2 | Node3 | Node4 |
|---|---|---|---|---|---|
| MPI-pod | 3 | 3 | 3 | 3 | 3 |
| Spark-pod | 3 | 3 | 3 | 4 | 2 |
| Hadoop-pod | 1 | 3 | 5 | 1 | 5 |
| Total | 7 | 9 | 9 | 8 | 10 |

表 5.10　MRWS 调度算法应用容器分布

| 节点 / 类型 | Master | Node1 | Node2 | Node3 | Node4 |
|---|---|---|---|---|---|
| MPI-pod | 3 | 4 | 3 | 3 | 2 |
| Spark-pod | 2 | 4 | 3 | 3 | 3 |
| Hadoop-pod | 3 | 2 | 3 | 3 | 4 |
| Total | 8 | 10 | 9 | 9 | 9 |

根据四种调度方案容器应用的分布可以看出，Random 随机调度三个计算框架的容器应用，不做任何的资源空闲率平衡，CPU 密集型容器应用 MPI-pod 在 Node1 上有 5 个，而 Node4 上只有 1 个。I/O 密集型容器应用 Hadoop-pod 在 Master 上有 1 个，但 Node2 上有 5 个，各种容器应用随机调度分配，但是整体的容器应用各节点上相差不大。这种调度方式必然会导致 Node4 上 CPU 资源紧缺，Node2 上 I/O 资源紧缺，影响集群性能。FirstFit 也不做资源使用率平衡，每次选择第一个满足资源的节点进行调度，导致前面的节点负载过高，后面节点负载较低，从整体容器应用分布来看，Node4 上仅分配了三个容器应用，而 Node1 节点高达 13 个，导致部分节点各种资源负载过高，部分节点资源利用率不足。Kubernetes 对 CPU 和内存的资源空闲率做了平衡，尽量使内存和 CPU 资源平衡使用，从分布看出 CPU 密集型 MPI-pod 和内存密集型 Spark-pod 在各节点上的分布较为均匀，都在 2 ~ 4 个容器应用，但是 I/O 密集型应用 Hadoop-pod 分布不均，Node1 和 Node3 上只有 1 个容器应用，Node2 和 Node4 上有 5 个容器应用，这种只注重 CPU 和内存均衡的调度方式导致集群各节点 I/O 密集型分布不均，影响集群性能。MRWS 调度方式可以看到几种密集型应用分布较为均衡，各节点的各种密集型容器应用都在 2 ~ 4 个，整体节点容器数量也在 8 ~ 10 个之间。整个集群各维度资源利用率较为均衡，集群负载较好。

### 5.4.2　单个计算框架服务性能

针对四种调度方案的容器应用分布，首先进行单个计算框架的性能测试，在容器应用中设置资源的下限，不限制应用容器使用资源的量，在同一个节点上的容器应用可以自由竞争资源。对比几种调度算法单个计算框架应用执行的时间，在 Hadoop-pod 中进行 TestDFSIO 测试，分别读写 15 个 128M 的文件，记录总体时间，Spark-pod 中进行 PageRank 程序计算，测试文件大小约 3.2M，轮数为 2000 轮迭代，MPI-pod 中进行 pi 运算，15 个节点 110000 轮计算。每个调度算法分别运行三次，取三次运算结果的平均值，各计算框架容器应用执行时间如下 (以秒为单位):

表 5.11　四种调度算法应用容器执行时间

| 容器应用 / 调度 | Hadoop-TestDFSIO | Spark-PageRank | MPI-PI |
|---|---|---|---|
| Random | 255.92 | 267.33 | 387.19 |
| FirstFit | 276.48 | 274.67 | 390.69 |
| Kubernetes | 250.44 | 252.00 | 214.42 |
| MRWS | 228.33 | 250.52 | 212.88 |

对比四种调度算法单计算框架下容器应用的执行时间，可以很明显大发现在进行 I/O 密集型容器应用 Hadoop-TestDFIO 测试时，Random、FirstFit 以及 Kubernetes 调度算法没有考虑 I/O 资源均衡利用，导致多个 I/O 密集型应用被调度到同一节点。在容器中运行应用程序时，大量 I/O 密集型应用竞争 I/O 和网络资源，导致程序效率低下，执行时间过长。MRWS 调度方式考虑 I/O 和网络带宽因素，尽量分散到集群各几点上，其性能最佳，Hadoop-TestDFSIO 执行时间最短，FirstFit 效果最差，Random 和 Kubernetes 相差不大。在单独运行内存密集型应用 Spark-PageRank 容器应用下，Random 和 FirstFit 都不考虑内存利用率均衡性，其运行时间最长，尤其是 FirstFit 将大量内存密集型调度到了同一个节点，使得内存成为该节点瓶颈，运行时间最长。Random 随机分配节点，其效果一般，Kubernetes 和 MRWS 均做了内存利用率均衡，其效果较好，各内存密集型容器大致可以分散到集群各节点上，其效果最佳。与内存类似，CPU 密集型应用表现最为明显，Pi 运算是一个典型的计算密集型应用，大量的时间花费在三角函数的运算上，Random 和 FirstFit 均不做 CPU 利用率均衡，导致大量的 CPU 密集型应用调度到同一个节点，该节点的 CPU 成为其瓶颈，运行时间较长。Kubernetes 和 MRWS 调度可以将 CPU 密集型应用尽可能分散到各节点上，运行时间大大缩短。

以 CPU 密集型应用调度为例，普通用户并不关心容器应用被调度到集群的哪

个节点，只在乎计算框架应用的执行时间最短。在分布式计算 MPI-PI 程序时，用户将任务分成 15 份发送到 15 个容器应用上进行 PI 运算，如果某个节点上部署的 MPI-pod 越多，该节点上 CPU 资源竞争压力越大，完成计算任务的时间越长。下面两图展示 FirstFit 和 Kubernetes 调度算法下集群中各节点 CPU 利用情况:

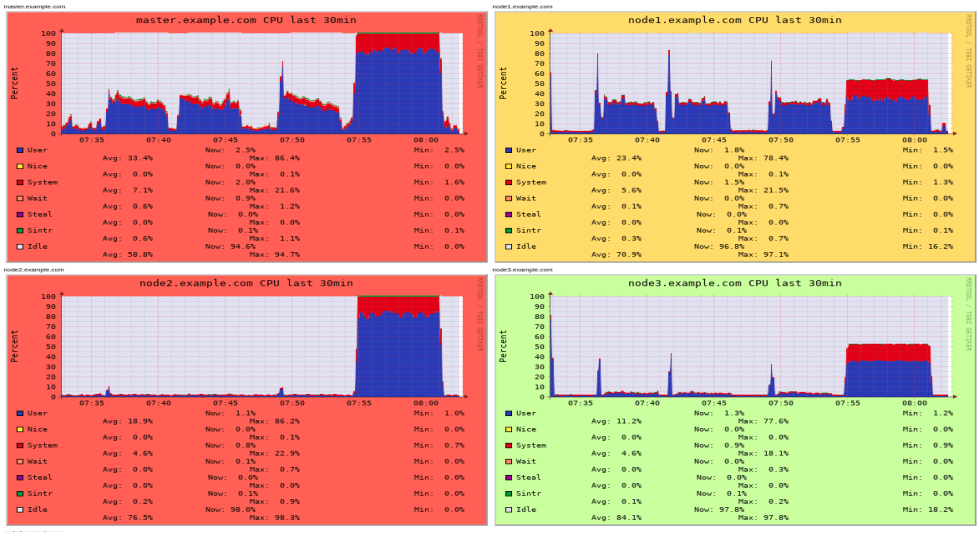



图 5.17　FirstFit 调度 MPI-pod 集群 CPU 利用情况

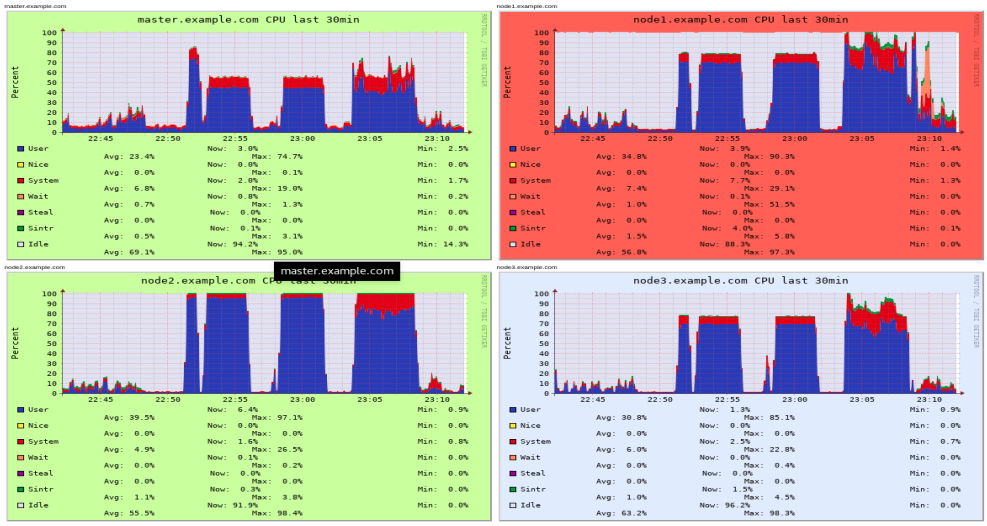



图 5.18　Kubernetes 调度 MPI-pod 集群 CPU 利用情况

上述两图是在集群上部署 Ganglia 集群监控下两种调度算法的 CPU 利用率，可以很明显发现在 FirstFit 调度下 Node3 和 Node4 上的 CPU 利用不足，而 Master 和 Node2 上 CPU 竞争激烈，影响了 MPI-PI 计算框架下容器应用的执行时间，而 Kubernetes 调度下各节点 CPU 资源利用较为充分，执行时间较短。

### 5.4.3　混合部署多计算框架下的服务性能

混合部署三种密集型容器应用框架，同时运行 Hadoop-TestDFSIO、Spark-PageRank 和 MPI-PI 容器应用程序，对比几种调度算法的执行时间。混合部署和同时运行不同计算框架类似于多用户在容器云集群上部署运行不同的应用的场景，不仅相同密集型容器应用对节点上资源进行竞争，其他计算框架也会参与竞争资源。比如同一节点上 MPI-pod 对 CPU 进行竞争，该节点上部署的 Spark-pod 也会参与竞争 CPU 资源，调度不合理将导致整个集群性能低下，所有用户应用性能都将受到影响。每种调度算法执行 3 次，取 3 次执行时间的平均值，四种调度下同时运行多计算框架容器应用执行时间如下 (秒为单位):

表 5.12　同时执行多计算框架应用容器执行时间

| 容器应用 / 调度 | Hadoop-TestDFSIO | Spark-PageRank | MPI-PI |
|---|---|---|---|
| Random | 316.46 | 584.00 | 418.43 |
| FirstFit | 355.81 | 603.33 | 454.71 |
| Kubernetes | 299.87 | 503.33 | 271.33 |
| MRWS | 269.10 | 491.66 | 256.62 |

首先将执行时间和单个计算框架容器应用单独执行时间进行对比，在混合部署同时执行多计算框架下所有应用执行时间都变慢，如 Hadoop-TestDFSIO 在 MRWS 调度下单独执行时间是 228.33，在混合部署执行下 269.10，这是由于更多的容器容器应用参与资源竞争导致整体执行时间变慢。其中 Spark-PageRank 执行时间变化最为明显，这是由于 PageRank 的运算不仅需要大量的内存资源，还需要一定的 CPU 资源，而 CPU 资源被 MPI-PI 计算框架占用，导致其执行时间变慢。

对比各调度算法下混合部署多计算框架以及同时执行多计算框架容器应用执行时间，FirstFit 性能最差，各计算框架执行时间最长，这是由于其每次选择第一个可用节点导致其资源利用不均，有的节点过载而有的节点资源利用不足。Random 和 Kubernetes 相较于 MRWS 在 Hadoop-pod 的执行时间较差，整体而言 MRWS 调度算法西能你最优，各计算框架的执行时间相较于其他调度算法执行都最短。这是由于 MRWS 调度算法综合考虑了容器应用的 CPU、内存、磁盘、网络带宽和节点已部署 Pod 的因素，利用 FAHP 自动建模和求解容器应用多维资源权重参数，采用空闲资源和资源利用均衡共同进行评分，选取最优的节点作为容器应用调度目标。这种调度方式在多计算框架下性能表现最为明显，通常在用计算框架处理大数据时 Hadoop 进行 MapReduce 时需要大量 I/O，Spark 是内存式的需要大量内存

资源，MPI 则用于计算密集型较多等。每个大数据处理框架都有自己相应的特点，在混合部署时需要充分考虑个计算框架的特点才能充分利用容器集群资源，实现多种密集型应用在节点上混合部署，提升集群服务性能。

## 5.5　本章小结

本章从实验的角度对比四种调度算法性能优劣，首先详细介绍云计算仿真平台 CloudSim 和容器云仿真平台 ContainerCloudSim，在该平台上开发四种调度算法并进行大规模调度仿真，对比几种调度方法在集群资源利用和负载均衡方面的性能。通过对比分析，MRWS 调度相较于其他几种调度方法无论在单节点各维度资源利用方面还是集群整体资源利用方面其负载均衡性较好，集群资源利用率也更高。然后介绍和部署实验室研发的大数据存储与处理容器云平台，这是一个存储与计算分离的容器云平台，平台主要用于大数据存储和处理。接着在该平台上开发数十种大数据处理框架，并以 Hadoop 批处理框架为例详细介绍其开发流程，用户通过 origin-web-console 与平台进行交互，在几秒钟之内快速构建大数据处理框架并进行容器伸缩扩展。最后，在 Paladin 平台进行多计算框架容器应用调度分析，对比几种调度算法下多计算框架容器应用的执行性能。分别以 I/O 和网络密集型应用 Hadoop-TestDFSIO、内存密集型应用 Spark-PageRank 和 CPU 密集型应用 MPI-PI 计算框架容器应用为例，在混合部署几个计算框架时单独执行和同时运行下，对比各容器应用的执行时间，从而获取几种调度算法下集群的服务性能。

# 第 6 章　总结与展望

本文从实验室“多元化大数据高效能存储与处理”实际项目出发，发现在大数据存储与处理的容器云平台 Paladin 上多计算框架混合部署下其计算性能较低，Paladin 是基于开源 OpenShift Origin 容器云平台构建，而 OpenShift Origin 的容器编排引擎是 Kubernetes。接着对主流的容器编排引擎 Kubernetes、Mesos 以及 Docker Swarm 进行学习，对 Kubernetes 的调度流程和调度算法进行深入研究，发现其调度算法在评分阶段仅考虑内存和 CPU 的空闲率以及平衡情况，导致其他维度资源利用不足，从而使集群效率低下。面对其调度方案不足，本文首先优化其调度流程，通过一个 Hash 表管理重复部署的容器应用，节约镜像下载时间。

面对 Kubernetes 容器调度算法不足，提出一个基于多维资源权重参数的调度方案，综合考虑集群中节点 CPU、内存、磁盘、网络带宽和已部署 Pod 数量因素，赋予每个影响调度因素一个权重值进行综合评分。权重参数根据容器应用对资源的需求进行建模，使用模糊层次分析法 FAHP 自动构建模糊成对比矩阵和判断矩阵，计算出满足一致性要求的权重参数作为待调度容器应用各维度资源的权重值。对于新提出的调度算法，在容器云仿真平台 ContainerCloudSim 上进行大规模调度仿真，并与 Kubernetes、Random、FirstFIt 调度算法在集群资源利用率和负载均衡方面进行对比，接着在 Paladin 平台上实际开发调度方案，进行多计算框架混合部署实际调度性能实验。实验表明新的调度算法无论在单计算框架容器应用执行效率还是同时运行多计算框架容器应用执行效率方面都有较大的提升。

总体而言，本文主要围绕大数据存储与处理的容器云平台 Paladin 多计算框架混合部署与执行效率较低的场景下做了以下几个方面工作:

(1) 调研当前容器编排引擎集中式调度、两层调度和共享状态调度的典型架构，并以实际的例子分析其优点和不足。深入分析 Kubernetes 调度流程和调度缺点，对其调度流程进行优化并提出了一种新的综合考虑容器应用 CPU、内存、磁盘、网络带宽以及已部署 Pod 等因素的调度方案 MRWS。

(2) 针对调度方案 MRWS 中多维资源权重问题，使用模糊层次分析法自动建模和构建满足一致性要求的模糊成对比矩阵和判断矩阵，实现新的调度方案。

(3) 在容器云方正平台 ContainerCloudSim 上针对新的调度方案进行大规模仿真，并与常见的 Random、FirstFit 以及 Kubernetes 默认的 Default 调度方案进行性能和负载均衡对比。

(4) 在 Paladin 平台上开发数十种大数据处理框架，用户可以实现快速构建大数

据处理环境和容器伸缩，开发新的调度方案 MRWS，并在该平台上进行多计算框架混合部署与执行下对比几种调度算法的性能，实验表明新的调度方案极大提升了集群的服务性能，使得多计算框架容器云资源调度更实用，平台实用性更强。

本文主要围绕提升大数据存储与处理容器云平台 Paladin 服务性能展开，容器资源调度是提升容器云集群的关键因素，要使一个集群更好的服务用户，在后续的研究工作中可以从以下几个方面进行:

1. 数据亲和性调度。Paladin 是一个集大数据存储与处理于一身的容器云平台，底层是一个分布式的文件系统，通过客户端网盘挂在形式实现数据访问，并提供 web-console 的数据上传、下载、删除等管理操作。因此，在数据处理框架进行数据处理时，数据的位置对其性能存在重大影响，需要研究以数据为中心的调度方案，实现数据亲和性调度，将数据存储和处理容器尽可能调度到相同或近距离的节点，减少网络传输。
2. 优先抢占式调度。在大数据批处理中有的任务无需及时处理，如 Hadoop 进行离线数据处理，有的需要进行及时处理如流计算等，调度系统需要实现优先抢占式调度，在紧急任务模式下暂停优先级较低的任务，给予优先级高任务更多的计算资源。
3. 容器迁移策略研究。多计算框架混合部署场景下资源的分配是静态的，有的用户在部署完计算框架后并不进行任务处理，根据这一特点可以实现集群资源超卖，在保证性能情况下实现经济利益最大化。一些不太活跃用户的容器可以迁移到性能较低的节点上，在用户使用时再迁回高性能的节点。
4. 任务特征调度。根据积累数据中心的数据提取更多用户计算框架处理任务的特征，根据特征制定合适的调度方案，最大化的利用集群的资源，让集群按需服务，用户有更好的服务体验。

# 参考文献


[1] Hayes B. Cloud computing[J]. Communications of the Acm, 2008, 51(7): 9-11.

[2] 浙江大学 SEL 实验室. Docker: 容器与容器云[M]. [出版地不详]: 人民邮电出版社, 2015

[3] Lin Weiwei Q D. Survey of resource scheduling in cloud computing[J]. Computer Science, 2012, 39(10): 1-6.

[4] Quang-Hung N, Nien P D, Nam N H, et al. A genetic algorithm for power-aware virtual machine allocation in private cloud[C]//Information and Communication Technology-EurAsia Conference. Heidelberg, German: [s.n.], 2013.

[5] R. B A J. Energy-aware resource alocation heuristics for eficient management of data centers for cloud computing[J]. Future Generation Computer Systems, 2012, 28(5): 755-768.

[6] Soni G, Kalra M. A novel approach for load balancing in cloud data center[C]//Advance Computing Conference. Piscataway: [s.n.], 2014.

[7] Kumar D, Raza Z. A pso based vm resource scheduling model for cloud computing[C]//IEEE International Conference on Computational Intelligence. [S.l.: s.n.], 2015.

[8] Buyya R, Yeo C S, Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities[J]. High Performance Computing, 2008, 11(4): 5-13.

[9] Kozhirbayev Z, Sinnott R O. A performance comparison of container-based technologies for the cloud[J]. Future Generation Computer Systems, 2017, 68: 175-182.

[10] Usman. Kubernetes, mesos, and swarm: Comparing the rancher orchestration engine options [EB/OL]. https://rancher.com/comparing-rancher-orchestration-engine-options/.

[11] Barik R K, Lenka R K, Rao K R, et al. Performance analysis of virtual machines and containers in cloud computing[C]//International Conference on Computing. [S.l.: s.n.], 2017.

[12] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers[C]//IEEE International Symposium on Performance Analysis of Systems and Software. [S.l.: s.n.], 2007.

[13] Lossent A, Peon A R, Wagner A. Paas for web applications with openshift origin[C]//Journal of Physics Conference Series. [S.l.: s.n.], 2017.

[14] 机械工业出版社. 开源容器云 openshift: 构建基于 kubernetes 的企业应用云平台[M]. [出版地不详]: 人民邮电出版社, 2017

[15] Peng J, Dai Y, Yi R, et al. Research on processing strategy for cpu-intensive application [J]. Journal of Systems Architecture, 2016, 70(C): 39-47.

[16] Shuangke M A, Zhou L. Schedulingstrategy of io and network intensive applications based on the priorityin cloud environment[J]. Journal of University of Shanghai for Science and Technology, 2017.

[17] Jansen C, Witt M, Krefting D. Employing docker swarm on openstack for biomedical analysis [C]//International Conference on Computational Science and Its Applications. [S.l.: s.n.], 2016.

[18] Naik N. Building a virtual system of systems using docker swarm in multiple clouds[C]//IEEE International Symposium on Systems Engineering. [S.l.: s.n.], 2016.

[19] Cerin C, Menouer T, Saad W, et al. A new docker swarm scheduling strategy[C]//IEEE International Symposium on Cloud and Service Computing. [S.l.: s.n.], 2017.

[20] Hindman B, Author, Konwinski. Mesos: A platform for fine-grained resource sharing in the data center[J]. NSDI, 2013: 429-483.

[21] Ghodsi A, Zaharia M, Hindman B, et al. Dominant resource fairness: Fair allocation of multiple resource types[J]. NSDI, 2011: 323-336.

[22] Verma A K M, Pedrosa L. Large-scale cluster management at google with borg[C]//the Tenth European Conference on Computer Systems. New York, America: [s.n.], 2015.

[23] Burns B, Grant B, Oppenheimer D, et al. Borg, omega, and kubernetes[J]. Queue, 2016, 59(5): 50-57.

[24] Halici U, Dogac A. An optimistic locking technique for concurrency control in distributed databases[J]. Software Engineering IEEE Transactions on, 1991, 17(7): 712-724.

[25] Kubernetes 中文社区. Kubernetes 设计架构[EB/OL]. https://www.kubernetes.org.cn/k8s.

[26] Kwong C K, Bai H. A fuzzy ahp approach to the determination of importance weights of customer requirements in quality function deployment[J]. Journal of Intelligent Manufacturing, 2002, 13 (5): 367-377.

[27] Hong P, Yang G X, Cai L, et al. Cloud test environment deployment based on the needs of individual users[C]//Information Science and Service Science and Data Mining. [S.l.: s.n.], 2013.

[28] Saaty T L. How to make a decision: The analytic hierarchy process[J]. European Journal of Operational Research, 1994, 24(6): 19-43.

[29] Deng Xue Z H, Li Jiaming. Analysis and application of weight calculation method of analytic hierarchy process[J]. Mathematics in Practice and Theory, 2012, 42(7): 93-100.

[30] Wang X. Triangular fuzzy function method for multiple attributes decision making[J]. Journal of Liaoning Technical University, 2010.

[31] Chou C C. The canonical representation of multiplication operation on triangular fuzzy numbers [J]. Computers and Mathematics with Applications, 2003, 45(10): 1601-1610.

[32] Eastman C M. Introduction to fuzzy arithmetic: Theory and applications : Arnold kaufmann and madan m. gupta , van nostrand reinhold, new york, 1985[J]. International Journal of Approximate Reasoning, 1987, 1(1): 141-143.

[33] Piraghaj S F, Dastjerdi A V, Calheiros R N, et al. Containercloudsim: An environment for modeling and simulation of containers in cloud data centers[J]. Software Practice and Experience, 2016.

[34] 薛瑞尼. ThuThesis: 清华大学学位论文模板[EB/OL]. 2017. https://github.com/xueruini/thuthesis.

# 致 谢

光阴似箭，岁月如流水，转眼已在园子里度过了三年时光，在三年硕士期间，老师们的谆谆教诲，同学们的热情帮助，让我不仅在学业上进步，也让我度过人生中一段快乐的时光。

首先我要感谢我的导师武永卫教授在硕士三年期间不但给予学术上的指导，在生活上也给予了悉心的照顾，更教会了我许多做人的道理。武老师是一个和蔼可亲的人，他治理下的实验室宽松、活跃以及浓厚的学术氛围让我收获颇丰。其次要由衷感谢陈康副教授和姜进磊副教授，陈康老师不仅与学生亦师亦友，更是一位计算机的百科全书，他知识面广而深，对待问题较真的态度影响实验室的每一个人，在我小论文投稿方面给予了很多指导。再次感谢三位老师在硕士期间给予的指导和帮助，祝各位老师阖家幸福！

其次我要感谢实验室同学的照顾，艾志远是第一个给予我指导和帮助的师兄，他工程能力极强，对我帮助很大。马腾、刘梦醒、杨珂、李雪、夏鑫、李志涛、何东标等实验室的同学不仅让我感受到 Madsys 实验室的温暖，也常常帮我答疑解惑，与大家在一起的时光很开心，衷心祝愿大家学业进步，事业有成！

最后我要感谢我的父母和女朋友，你们在我困惑和无助的时候给予我很大鼓励和安慰，一直在背后默默的付出和支持我。感谢你们辛勤的付出和关爱，让我能够在硕士期间安心完成学业，你们的爱是我前进的不竭动力。谨向所有在我硕士期间帮助我和指导我的老师、同学、朋友表示衷心的感谢！

感谢 LaTeX 和 ThuThesis[34]，帮我节省了不少时间。

# 声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，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，本学位论文的研究成果不包含任何他人享有著作权的内容。对本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。

签　名：_____ 日　期：_____

# 个人简历、在学期间发表的学术论文与研究成果

## 个人简历

1991 年 04 月 27 日出生于贵州省铜仁市。

2010 年 9 月考入清华大学计算机系，2014 年 7 月本科毕业并获得工学士学位。

2016 年 9 月免试进入清华大学计算机系攻读硕士学位至今。

## 发表的学术论文

[1] 龚坤, 武永卫, 陈康, 等. 容器云多维资源利用率均衡调度研究. 计算机应用研究 (已被录用, 2020 第 37 卷第 4 期)