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 容器虚拟化技术能够和宿主机共享系统资源并且实现容器间的隔离，是当前技术研究的热点，已经成为容器的代名词。基于 Docker 的私有容器云平台获得广泛的应用，开始逐步取代传统虚拟机为基础构建的云计算系统。与传统的云计算集群类似，一个高效且强大的容器编排管理系统对数据中心资源利用率和集群性能具有至关重要的作用，Kubernetes 是当前容器云系统中应用最为广泛的容器调度系统，其轻量开源和强大的服务发现、集群监控和错误恢复能力深受用户好评。但 Kubernetes 过度专注于调度器能力和性能，其自身的资源分配和调度算法单一往往导致整个容器云集群资源利用率和均衡性很差，尤其是在多种计算框架同时部署的情况下其调度缺陷暴露无遗。

近年来，OpenShift 容器云平台用户针对 Kubernetes 调度器的不足往往需要开发自己的调度算法来适应特定的应用场景，这对追求性能和资源利用率的普通用户往往具有一定的难度，本文在深入分析 OpenShift 私有容器云平台 Kubernetes 调度器核心调度技术后，提出了一种全新的调度方案，本文主要内容如下:

- 基于开源的 OpenShift Origin 构建私有容器云平台 Paladin，在该平台上开发部署 Hadoop、Spark、MPI、Storm、Regraph 等十多种分布式计算框架，能够根据用户需求快速构建分布式计算平台。
- 深入研究 Kubernetes 调度核心技术，针对其调度算法的不足，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，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。
- 针对新提出的调度方案，在 CloudSim-4.0 容器云仿真平台进行调度仿真，并与 Kubernetes 默认调度方案、Random 调度方案进行对比，能够极大提升容器云集群资源利用率和实现负载均衡性
- 在私有容器云平台 Paladin 上设计并实现该调度方案，使用多个计算框架同时进行调度性能测试，能够极大缩短多计算框架的应用执行时间。

关键词：Docker；容器云；调度策略；OpenShift 平台；计算框架；FAHP

Abstract

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summarization of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Key words are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 key words, with semi-colons used in between to separate one another.

Key Words: Docker; container cloud; scheduling strategy; openshift platform; computing framework; FAHP

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呈现爆炸式的增长，如何快捷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。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。这虚拟化方式存在巨大的虚拟化开销，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 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 调度流程的基础上，设计并实现了一种新的调度方案，部署在基于开源 OpenShift Origin 开发的私有容器云 Paladin 上，极大提升了多种计算框架应用的性能。

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，本文主要关注服务器端大规模的云计算。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，再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，达到最终的目标。

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，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 []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 [] 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 [] 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 [] 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，从而提升集群资源的利用率；文献 [] 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，对推动云计算的普及起到的巨大的推动作用。

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、访问控制等，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。以容器为虚拟化技术可以实现虚拟化较小的开销，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，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%。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

容器云和传统的以虚拟机与基础构建的云计算一样需要一个性能强大的容器编排管理器负责容器的调度、创建、销毁、监控、重启、错误恢复以及服务的组合灯工作，决定容器云集群的性能和资源利用率，容器调度器同时也是推动容器迅速实现应用的重要因素。当前主要有三大主流的容器调度框架: 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。在调度架构和调度模式上三种调度框架也各不相同，Swarm 中调度算法主要包括最少容器、最多容器和随机调度三种；Memsos 则侧重于负载均衡，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；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。几种调度方式各有优缺点，用户可以根据自己的需求不同选取响应的容器调度框架构建不同的容器云计算环境，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，在多种计算框架应用部署的情况下实现资源利用最大化，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，设计和实现了一个基于数学方法 FAHP 集群资源建模和参数自动求解的调度器，主要工作如下:

(1) 基于开源的 OpenShift Origin 构建实验室私有 PaaS 层容器云平台 Paladin，在该平台上开发部署十多种分布式计算框架，普通用户可以根据实际需求快速

配置和构建自己所需的计算环境。

(2) 深入分析 OpenShift 容器编排管理器 Kubernetes 的调度流程和不足之处，提出了一种综合考虑应用特性的多维资源空闲率权重的评价函数和调度方法，该方法充分考虑容器应用 CPU、内存、磁盘、网络带宽和已部署 Pod 数量的影响，最后通过对集群物理节点资源和应用资源的数学建模，利用 FAHP 方法自动构建满足一致性要求的模糊成对比矩阵和判断矩阵，实现应用参数权重的自动求解，选取评分最高的节点作为容器应用的调度目标。

(3) 在容器仿真平台 CloudSim-4.0 上对新的调度方案进行仿真，对比分析 Kubernetes 调度方法以及 Random 调度方法的性能，实验表明能够极大提高集群资源利用率和实现负载均衡。

(4) 在私有容器云平台 Paladin 上设计开发该调度方案，部署多种计算框架应用进行应用性能测试，实验表明新的调度方案能极大缩短多种计算框架的执行时间，提升私有容器云集群性能。

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新兴 Docker 容器技术构建的容器云，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器方法。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，设计一种新的调度方法和调度流程，新的调度方案充分考虑容器应用特点和集群物理资源的特点，主要包括其调度流程、多维资源建模、反馈器的设计以及全新的评价函数构建。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。第五章首先使用 CloudSim-4.0 进行仿真环境的搭建，对比分析新的调度方法和 Kubernetes 默认调度方案以及 Random 调度方法在集群资源利用率、负载均衡性方面的性能。然后在实验室私有容器云平台 Paladin 上开发部署十几种分布式计算框架，将新的调度方案应用于 Paladin 中，使用多个计算框架应用对其性能进行测试。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 私有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，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，其底层是 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) 子系统限定特定的进程使用资源的量，然后让 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，并且实现较好的跨平台。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。

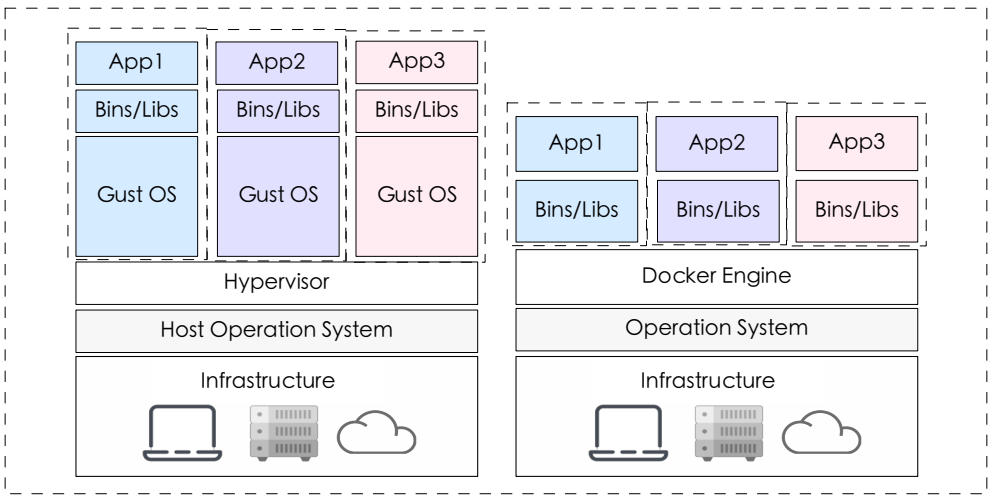

图 2.1　容器与虚拟机对比

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，最主要的是 Linux 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资源应用的隔离。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，负载管理 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 和虚拟机的架构发现，Docker 直接通过守护进程与操作系统进行通信，管理容器并进行资源分配，实现容器与主操作系统的隔离。没有臃肿的子操作系统，各容器直接与主操作系统共享资源，节约了大量的磁盘空间，其虚拟化开销极大缩小，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 具有如此多的优势，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，需要彻底隔离用户的需求下选择虚拟机技术，如果仅是应用层面的隔离可以选择容器，如数据库、前端、后端等。

2.1.2　OpenShift 容器云平台

Docker 是当前主流的容器技术代表，Kubernetes 作为现阶段应用最为广泛的容器编排引擎，OpenShift 将这两种主流技术应用于企业，作为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，是一套完整的容器应用云平台。在该平台上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；实现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，本文实验主要在开源的 OpenShift Origin 上进行分析和测试。从技术堆栈的角度分析，OpenShift 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，如图 2.2 所示。下面分别对这几个层次进行介绍：

1. 基础架构层。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，最后专门针对 Atomic Host 进行支持，是对企业版的 Linux 进行定制和优化的操作系统，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夸平台型，其前提是要求底层操作系统的内核和配置必须一致，因为其隔离依赖于 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，所有的应用最终到底层都生成一个 Docker 实例，只是将 Docker 的开放性和大量的镜像文件无缝衔接到平台上，对 Docker 的普通用户可以快速整合到平台中。
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如 Namespace、Pod、Replication Controller 等都继承自 Kubernetes，OpenShift 同样只是将 Kubernetes 进行叠加使用，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 的对象。
4. PaaS 服务层。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，构建自己所需的环，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 进行鼠标操作，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，用户可以最大限度的自由发挥。

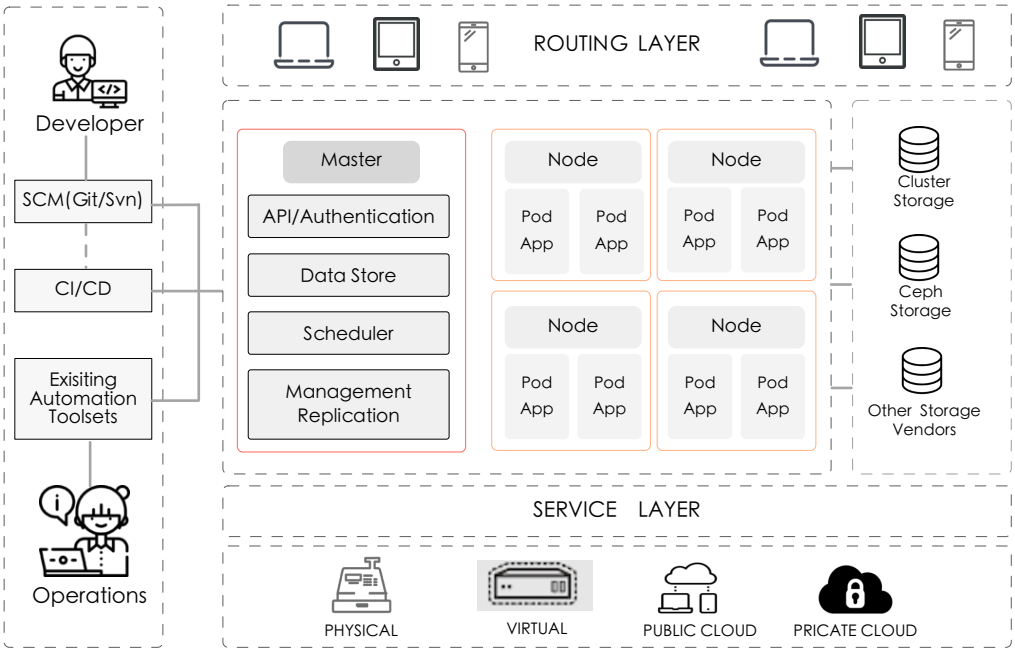

图 2.2　OpenShift Origin 架构图

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 ceph 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，收集过各节点资源情况，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，自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，一些容器应用和应用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，提升应用的效率，如分布式数据库。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 容器技术是容器集群的核心技术，但一个高效且强大的容器编排引擎也是集群的重要组成部分。一个好的调度器既要提现出作业调度的“公平”性，同时兼顾其性能和鲁棒性，要能应用的实际的生产环境中。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。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。其代表分别是 Docker 公司的 Swarm、Apache 的 Mesos、Google 的 Kubernetes。

2.2.1　一体式调度系统

一体式的调度使用单一的调度代理处理所有的请求，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，单节点也会成为其瓶颈。Docker 公司 2014 年发布的容器编排管理工具 Swarm 是典型的一体式调度系统，同期发布的还有 Docker 管理工具 Machine 以及 Compose，合称为 Docker 三剑客。Swarm 支持 Docker 标准的 API，内置于 Docker 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。

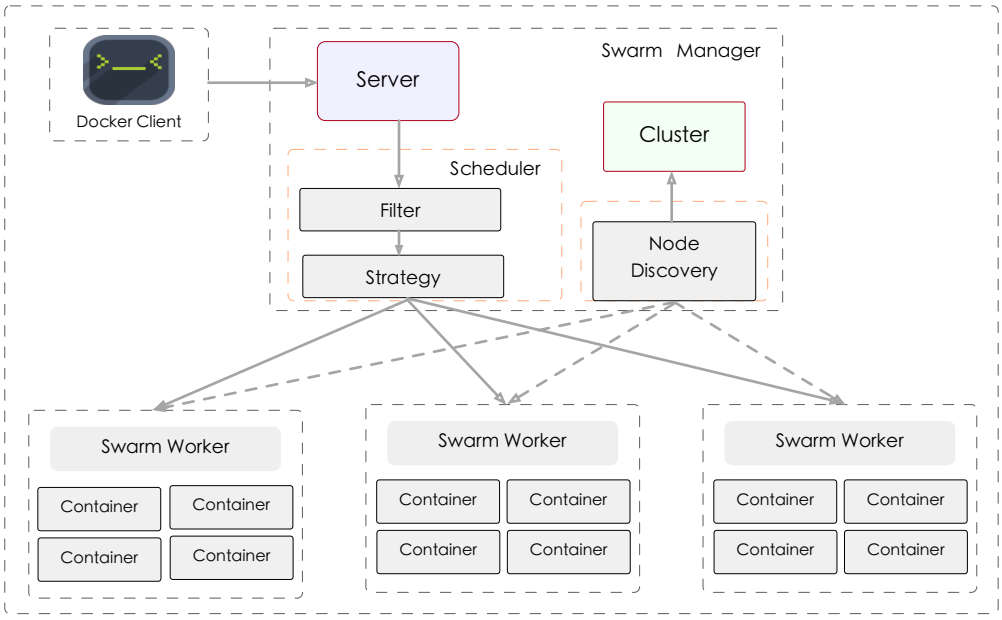

图 2.3　Swarm 架构图

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 算法实现主从的一致性。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。

约束过滤器是通过有用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，此外还可以通过镜像和标签吸引，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，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，直至找到合适的节点或返回调度错误信息。
2. Spread 算法也就是最少容器算法，该算法的初衷是保证容器进群的负载均衡。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 算法就是最多容器算法，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，否则依次从多到少尝试运行容器节点，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。在一些成熟的计算处理框架中如 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，因此也会造成资源利用效率不高。Apache Mesos 是最为典型的两层调度系统，Mesos 最初由加州伯克利分校的 AMPLab 开发，后在 Twitter 得到广泛使用和检验。

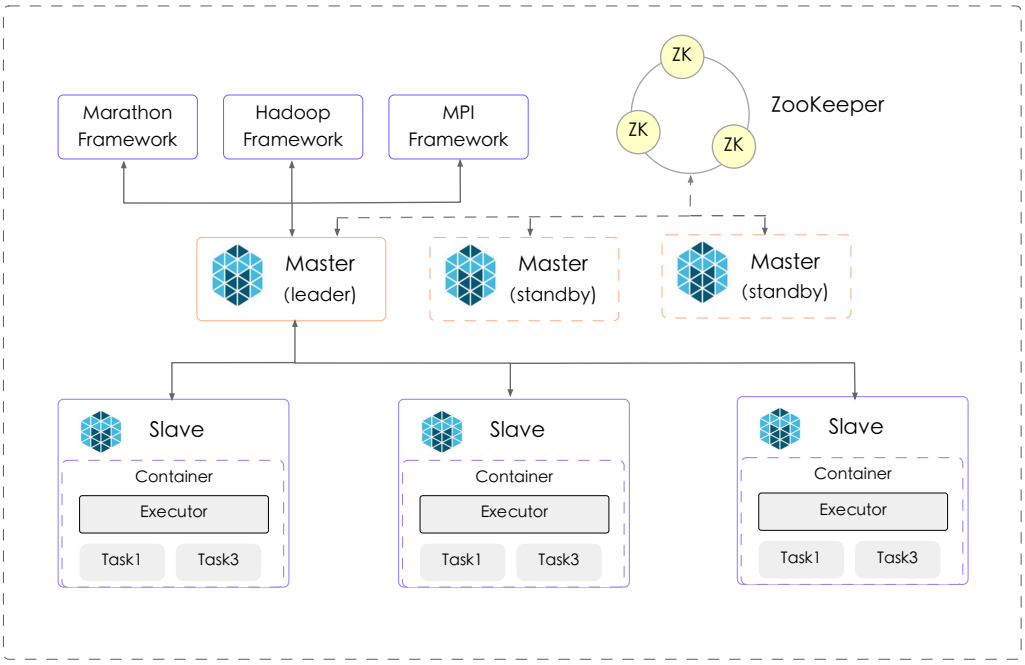

图 2.4　Apache Mesos 架构图

如图 2.4 所示，Mesos 的总体架构也采用主从设计，Master 节点运行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 的状态一直，一旦服务出现故障，立马进行新的备份选举，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，可以分别使用调度算法，提升了集群调度策略的扩展性。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中数据管理问题，如应用配置管理、统一命名服务、状态同步服务以及其他的组服务、集群管理等。ZooKeeper 操作非常简单易用、功能丰富可靠、并且提供了通用协议下的开源共享的存储库，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 主要解决 Master 节点的状态一致性和高可用问题，实现集群的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组件，既需要对底层的各 Slave 节点的资源进行管理和收集，同时通过一定的资源分配策略提供资源个体上层的各处理框架 Framework。当前对各 Framework 的资源分配策略为 DRF(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，必须等该框架释放该资源才能再次进行分配。
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的包括容器的创建、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 上具体执行。
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的调度。通过对其整体建构和核心组件的分析，Mesos 可以对分布式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，极大提升了资源的利用效率。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，具有较强的扩展性，容量不会成为制约其性能的因素。Meso 是模块化的实现，新增一个 Framework 不需要对 Mesos 进行重新编码，可以快速接

入和扩展新的应用，Master 节点使用 ZooKeeper 保证其状态一致性，容错性很好。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，往往不能实现优化调度。

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，两层调度系统虽然扩展性姐好，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、Kubernetes 以及使用事务方式解决一致性管理问题的 Omega，其中 Kubernetes 以其开源性深受大众好评，各大主流的互联公司都加大对其支持力度。

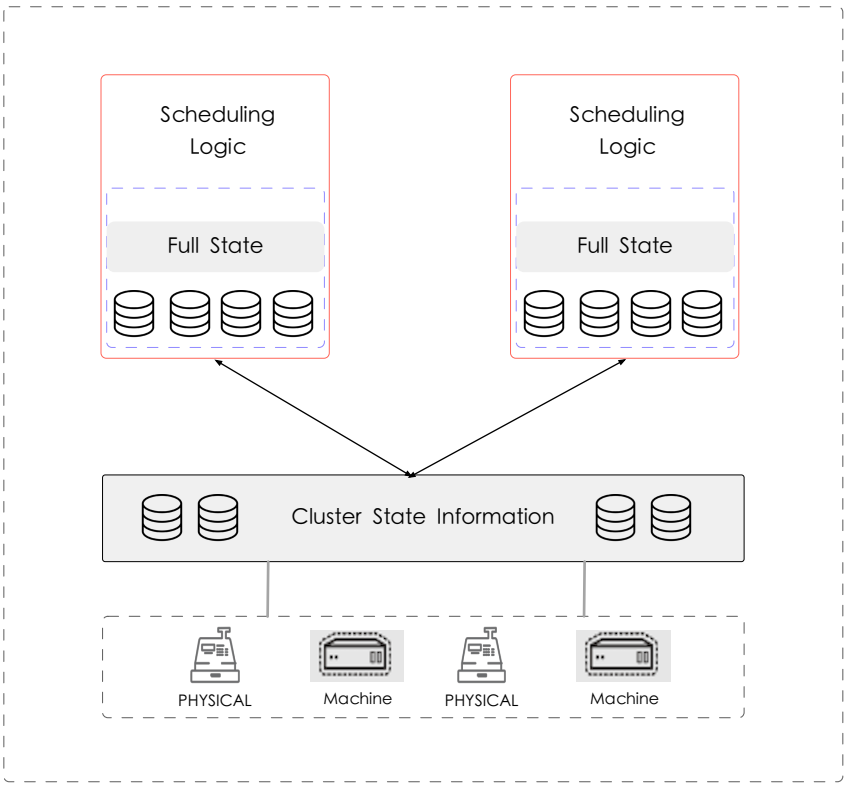

图 2.5　共享状态资源调度系统

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”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，将相关容器进行共同调度和部署。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，逐步形成一个大的生态。作为一个跨主机的应用容器编排引擎，Kubernetes 提供了一系列完整的功能，包括应用部署运行、资源调度、服务发现、动态扩容以及错误恢复等。Kubernetes 同时具备强大的集群管理能支撑分布式系统，实现了多租户应用、服务发现和服务注册、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，完整定义了构建业务系统的标准化架构层。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、运维监控在内各个环节的工具。

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开源定制。当前支持采用 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、亚新等，甚至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。当前支持 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，力推 Kubernetes 的广泛应用，支持其在当前公有云、私有云等各种基础设施平台上运行，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，下面展示其整体的架构体系。

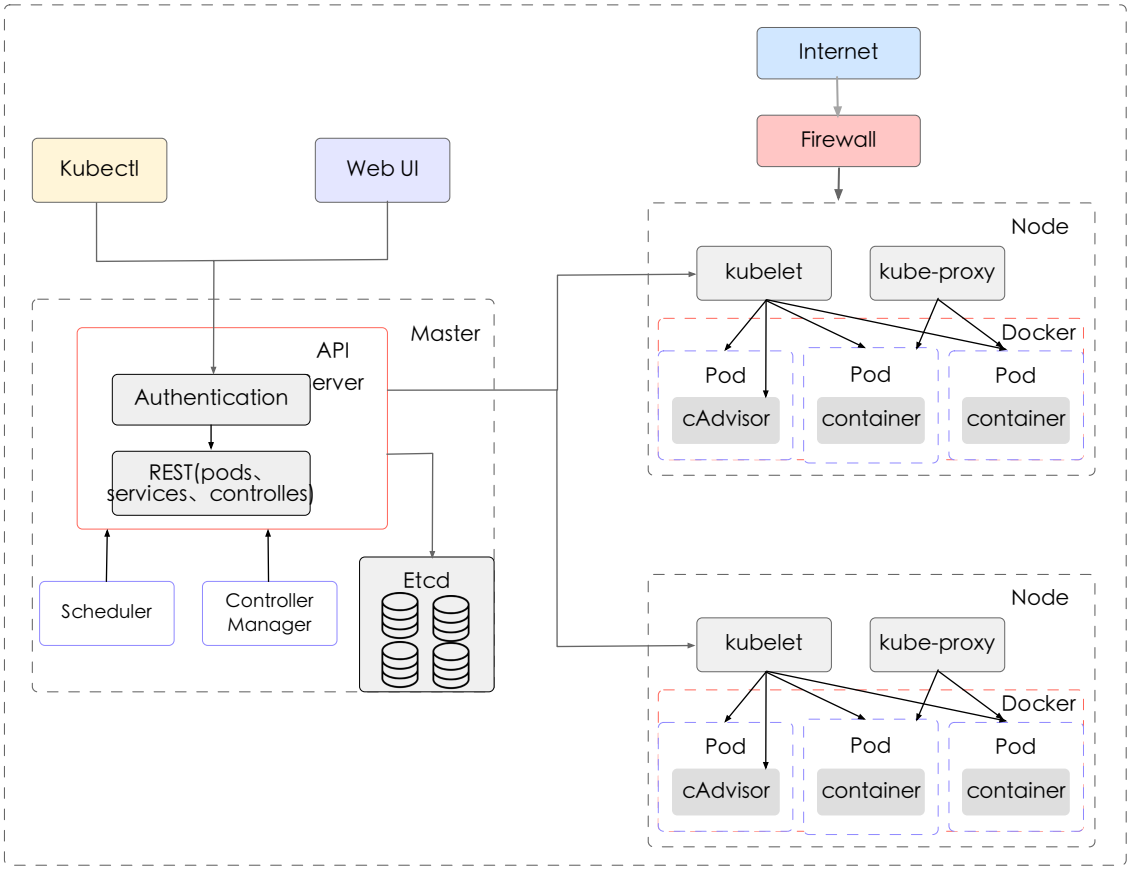

图 2.6　Kubernetes 体系架构图

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羡慕会详细的介绍。
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。主要有 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。

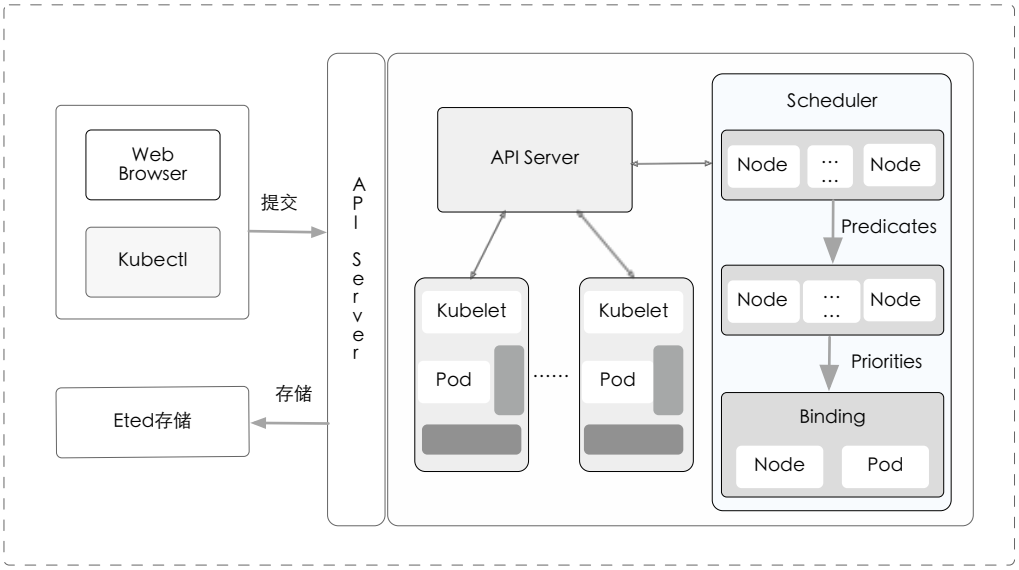

图 2.7　kubernetes 调度流程

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。接着详细介绍了集成当前流行的 Docker 和容器编排引擎 Kubernetes 技术的 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，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，还深入盐焗了其调度流程。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，也不能满足用户的在特定场景下的调度。本章在深入分析默认调度算法后，提出设计了一种新的基于多维资源空闲率权重的评价函数和调度方案 MRWS (Multidimensional Resource WeightSscheduling)。

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。用户除开发特殊的调度算法外，还可以针对其调度流程进行适当的优化。

3.1.1　Default 调度算法流程

在 Kubernetes 默认调度流程，预选阶段解决节点过滤，优选阶段解决最优节点选择问题，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。

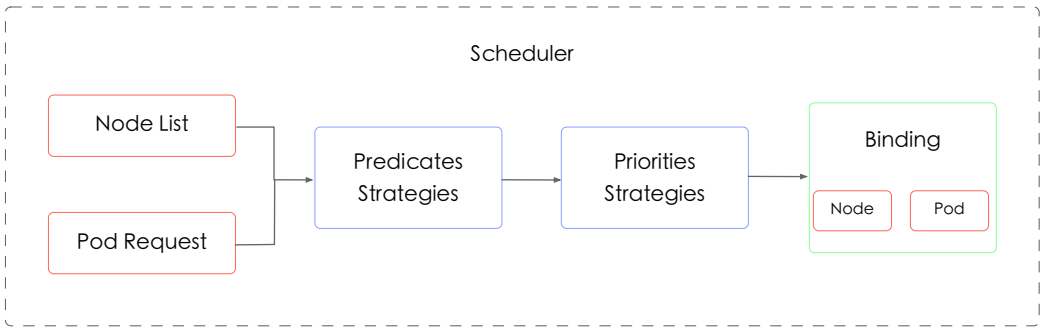

图 3.1　Default 调度算法流程

Predicates 策略过滤掉不满足 Pod 需求的节点，主要依据磁盘卷是否冲突、端口冲突、资源容量、节点检测、服务占用、亲和性筛选等条件。1.7 版本的筛选条件主要如下，新的版本 Kubernetes 还在不断完善和更新：

(1) NoDiskConflict：检测卷冲突，在当前的规则中，两个不同 Pod 不能使用相同的卷此一旦 Node 上已经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(PersistantVolume) 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，所有不在制定 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、还有转为 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合是否超过设定的最大值。

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

等因素。Kubernetes 使用优先函数集合进行 0-10 对节点进行评分，最终计算评分总合，评分越高表明 Pod 调度到该节点越适合，同时还可以为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合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 I/O、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 将更多关注于集群 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。

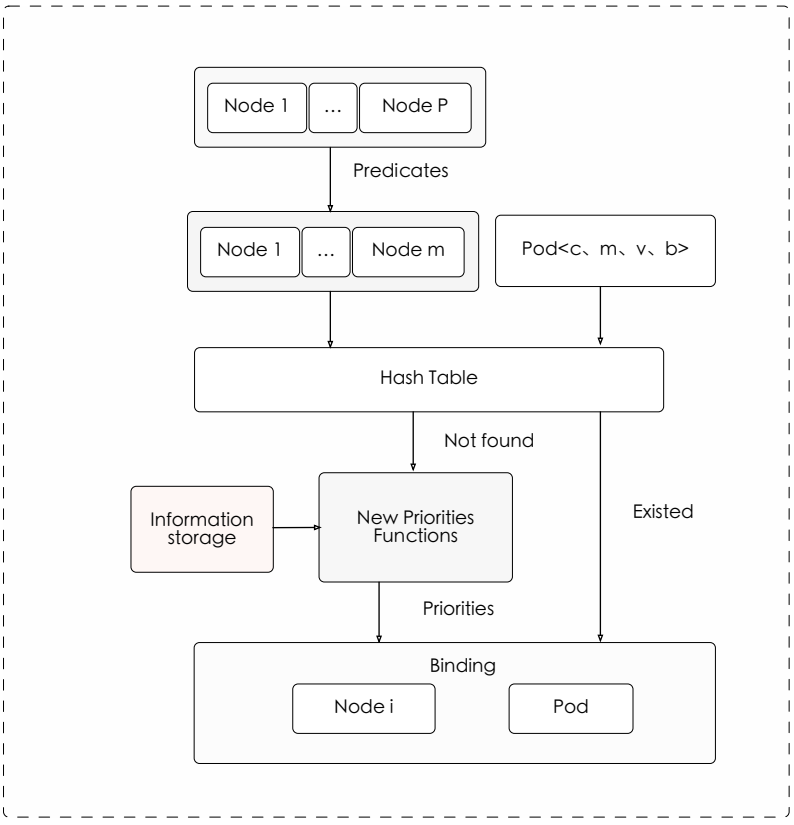

图 3.2　MRWS 算法调度流程

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：

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

其中，S_i 是第 i 个节点的总总评分，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 可以对调度流程进行优化，预选阶段从集群上的 P 个节点选择出满足预选规则的 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，Pod 中的容器镜像无需重复下载，节约磁盘空间和网络资源，同时缩短 Pod 应用的调度算法计算时间，使整个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情况，同时记录各节点上已分配的 Pod 资源总量和 Pod 运行数量。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，通过数学方法自动求解其参数，让各节点的资源使用更加均衡。

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，容易造成其他维度的资源消耗过载，从而使集群资源利用率低下。MRWS 算法将综合考虑 CPU、内存、磁盘、网络带宽以及节点上已运行的 Pod 数量等因素，更多关注于多维资源的均衡利用，尤其是在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，下面分别对其进行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种维度消耗越平均，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完全由人工赋予，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进行数学建模，资源的权重参数实现自动求解。在构建评分算法前，先对集群节点和 Pod 资源需求进行建模 (建模中节点指的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。假设有 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 CPU、内存、磁盘和网络带宽四种资源影响，因此 D=4。集群中单个节点的资源总量表示为$S = (s_1^d, s_2^d, s_3^d ... s_m^d), d \in D$，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，表示为为$R = (r_1^d, r_2^d, r_3^d ... s_m^d), d \in D$，其中$r_i^d$ 表示第 i 个节点上第 d 维资源的好销量。当前需要调度的 Pod 对资源的需求可以表示为$p^d, d \in D$，表示 Pod 对 d 维度资源的需求。

(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一个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，因此可以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的赋予一个权重，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 和内存相对较为稀缺，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节通过 FAHP(Fuzzy Analytic Hierarchy Process) 进行节点资源数学建模并使用一定的方式自动求解，用户可以通过微调重要性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，资源空闲率模块反应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α_1，但其整体评分依然偏高，会继续调度 Pod 到该节点上，这就造成该节点性能下降，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。平衡模块用于反应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约均衡，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。资源空闲率反应节点可用资源的状况，资源均衡反应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以及资源重要程度系数共同用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 算法以及其他如 Random、FirstFit 算法同样适用。在后面进行算法性能对比实验中，将使用该度量方式作为衡量集群均衡度的计算方法。

3.3　本章小结

本章首先详细 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，调度流程分为预选和优选两个阶段，对预选阶段所有的筛选规则进行分析，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，使其记忆功能。然后详细介绍了算法的三个组成部分和具体的评分规则，以及对物理资源进行建模，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。

插图索引

表格索引

公式索引

参考文献

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

致 谢

衷心感谢导师 xxx 教授和物理系 xxx 副教授对本人的精心指导。他们的言传身教将使我终生受益。

在美国麻省理工学院化学系进行九个月的合作研究期间，承蒙 xxx 教授热心指导与帮助，不胜感激。感谢 xx 实验室主任 xx 教授，以及实验室全体老师和同学们的热情帮助和支持！本课题承蒙国家自然科学基金资助，特此致谢。

感谢 LaTeX 和 ThuThesis[1]，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：_____

附录 A　外文资料原文

The title of the English paper

Abstract: As one of the most widely used techniques in operations research, *mathematical programming* is defined as a means of maximizing a quantity known as *bjective function*, subject to a set of constraints represented by equations and inequalities. Some known subtopics of mathematical programming are linear programming, nonlinear programming, multiobjective programming, goal programming, dynamic programming, and multilevel programming[1].

It is impossible to cover in a single chapter every concept of mathematical programming. This chapter introduces only the basic concepts and techniques of mathematical programming such that readers gain an understanding of them throughout the book[2,3].

A.1　Single-Objective Programming

The general form of single-objective programming (SOP) is written as follows,

$$\begin{cases} \max \ \ f(x) \\ \text{subject to:} \\ \qquad g_j(x) \le 0, \quad j = 1, 2, \cdots, p \end{cases} \tag{123}$$

which maximizes a real-valued function f of $x = (x_1, x_2, \cdots, x_n)$ subject to a set of constraints.

Definition A.1：　In SOP, we call x a decision vector, and $x_1, x_2, \cdots, x_n$ decision variables. The function f is called the objective function. The set

$$S = \left\{ x \in \Re^n \;\middle|\; g_j(x) \le 0,\ j = 1, 2, \cdots, p \right\} \tag{456}$$

is called the feasible set. An element x in S is called a feasible solution.

Definition A.2：　A feasible solution x^* is called the optimal solution of SOP if and only if

$$f(x^*) \ge f(x) \tag{A-1}$$

for any feasible solution x.

One of the outstanding contributions to mathematical programming was known as the Kuhn-Tucker conditionsA-2. In order to introduce them, let us give some definitions. An inequality constraint $g_j(x) \leq 0$ is said to be active at a point x^* if $g_j(x^*) = 0$. A point x^* satisfying $g_j(x^*) \leq 0$ is said to be regular if the gradient vectors $\nabla g_j(x)$ of all active constraints are linearly independent.

Let x^* be a regular point of the constraints of SOP and assume that all the functions $f(x)$ and $g_j(x), j = 1, 2, \cdots, p$ are differentiable. If x^* is a local optimal solution, then there exist Lagrange multipliers $\lambda_j, j = 1, 2, \cdots, p$ such that the following Kuhn-Tucker conditions hold,

$$\begin{cases} \nabla f(x^*) - \sum\limits_{j=1}^{p} \lambda_j \nabla g_j(x^*) = 0 \\ \lambda_j g_j(x^*) = 0, \quad j = 1, 2, \cdots, p \\ \lambda_j \geq 0, \quad j = 1, 2, \cdots, p. \end{cases} \tag{A-2}$$

If all the functions $f(x)$ and $g_j(x), j = 1, 2, \cdots, p$ are convex and differentiable, and the point x^* satisfies the Kuhn-Tucker conditions (A-2), then it has been proved that the point x^* is a global optimal solution of SOP.

A.1.1 Linear Programming

If the functions $f(x), g_j(x), j = 1, 2, \cdots, p$ are all linear, then SOP is called a *linear programming*.

The feasible set of linear is always convex. A point x is called an extreme point of convex set S if $x \in S$ and x cannot be expressed as a convex combination of two points in S. It has been shown that the optimal solution to linear programming corresponds to an extreme point of its feasible set provided that the feasible set S is bounded. This fact is the basis of the *simplex algorithm* which was developed by Dantzig as a very efficient method for solving linear programming.

Roughly speaking, the simplex algorithm examines only the extreme points of the feasible set, rather than all feasible points. At first, the simplex algorithm selects an extreme point as the initial point. The successive extreme point is selected so as to improve the objective function value. The procedure is repeated until no improvement in objective function value can be made. The last extreme point is the optimal solution.

Table 1　This is an example for manually numbered table, which would not appear in the list of tables

Network Topology		# of nodes	# of clients			Server
GT-ITM	Waxman Transit-Stub	600	2%	10%	50%	Max. Connectivity
Inet-2.1		6000				
Xue	Rui	Ni	ThuThesis			
	ABCDEF					

A.1.2　Nonlinear Programming

If at least one of the functions $f(x), g_j(x), j = 1, 2, \cdots, p$ is nonlinear, then SOP is called a *nonlinear programming*.

A large number of classical optimization methods have been developed to treat special-structural nonlinear programming based on the mathematical theory concerned with analyzing the structure of problems.

Figure 1　This is an example for manually numbered figure, which would not appear in the list of figures

Now we consider a nonlinear programming which is confronted solely with maximizing a real-valued function with domain $\mathfrak{R}^n$. Whether derivatives are available or not, the usual strategy is first to select a point in $\mathfrak{R}^n$ which is thought to be the most likely place where the maximum exists. If there is no information available on which to base such a selection, a point is chosen at random. From this first point an attempt is made to construct a sequence of points, each of which yields an improved objective function value over its predecessor. The next point to be added to the sequence is chosen by analyzing the behavior of the function at the previous points. This construction continues until some termination criterion is met. Methods based upon this strategy are called *ascent methods*, which can be classified as *direct methods*, *gradient methods*, and *Hessian methods* according to the information about the behavior of objective function f. Direct methods require only that the function can be evaluated at each point. Gradient methods require the evaluation of first derivatives of f. Hessian methods require the evaluation of second

derivatives. In fact, there is no superior method for all problems. The efficiency of a method is very much dependent upon the objective function.

A.1.3 Integer Programming

Integer programming is a special mathematical programming in which all of the variables are assumed to be only integer values. When there are not only integer variables but also conventional continuous variables, we call it *mixed integer programming*. If all the variables are assumed either 0 or 1, then the problem is termed a *zero-one programming*. Although integer programming can be solved by an *exhaustive enumeration* theoretically, it is impractical to solve realistically sized integer programming problems. The most successful algorithm so far found to solve integer programming is called the *branch-and-bound enumeration* developed by Balas (1965) and Dakin (1965). The other technique to integer programming is the *cutting plane method* developed by Gomory (1959).

Uncertain Programming　(*BaoDing Liu,* 2006.2)

References

NOTE: These references are only for demonstration. They are not real citations in the original text.

[1] Donald E. Knuth. The TEXbook. Addison-Wesley, 1984. ISBN: 0-201-13448-9

[2] Paul W. Abrahams, Karl Berry and Kathryn A. Hargreaves. TEX for the Impatient. Addison-Wesley, 1990. ISBN: 0-201-51375-7

[3] David Salomon. The advanced TEXbook. New York : Springer, 1995. ISBN:0-387-94556-3

附录 B　外文资料的调研阅读报告或书面翻译

英文资料的中文标题

摘要：本章为外文资料翻译内容。如果有摘要可以直接写上来，这部分好像没有明确的规定。

B.1　单目标规划

北冥有鱼，其名为鲲。鲲之大，不知其几千里也。化而为鸟，其名为鹏。鹏之背，不知其几千里也。怒而飞，其翼若垂天之云。是鸟也，海运则将徙于南冥。南冥者，天池也。

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \tag{123}$$

吾生也有涯，而知也无涯。以有涯随无涯，殆已！已而为知者，殆而已矣！为善无近名，为恶无近刑，缘督以为经，可以保身，可以全生，可以养亲，可以尽年。

B.1.1　线性规划

庖丁为文惠君解牛，手之所触，肩之所倚，足之所履，膝之所倚，砉然响然，奏刀騞然，莫不中音，合于桑林之舞，乃中经首之会。

表 1　这是手动编号但不出现在索引中的一个表格例子

Network Topology		# of nodes	# of clients			Server
GT-ITM	Waxman Transit-Stub	600	2%	10%	50%	Max. Connectivity
Inet-2.1		6000				
Xue	Rui	Ni	ThuThesis			
	ABCDEF					

文惠君曰：“嘻，善哉！技盖至此乎？”庖丁释刀对曰：“臣之所好者道也，进乎技矣。始臣之解牛之时，所见无非全牛者；三年之后，未尝见全牛也；方今之时，臣以神遇而不以目视，官知止而神欲行。依乎天理，批大郤，导大窾，因其固然。技经肯綮之未尝，而况大軱乎！良庖岁更刀，割也；族庖月更刀，折也；今臣之刀十九年矣，所解数千牛矣，而刀刃若新发于硎。彼节者有间而刀刃者无厚，以无厚入有间，恢恢乎其于游刃必有余地矣。是以十九年而刀刃若新发于硎。虽然，每至

于族，吾见其难为，怵然为戒，视为止，行为迟，动刀甚微，謋然已解，如土委地。提刀而立，为之而四顾，为之踌躇满志，善刀而藏之。”

文惠君曰：“善哉！吾闻庖丁之言，得养生焉。”

B.1.2　非线性规划

孔子与柳下季为友，柳下季之弟名曰盗跖。盗跖从卒九千人，横行天下，侵暴诸侯。穴室枢户，驱人牛马，取人妇女。贪得忘亲，不顾父母兄弟，不祭先祖。所过之邑，大国守城，小国入保，万民苦之。孔子谓柳下季曰：“夫为人父者，必能诏其子；为人兄者，必能教其弟。若父不能诏其子，兄不能教其弟，则无贵父子兄弟之亲矣。今先生，世之才士也，弟为盗跖，为天下害，而弗能教也，丘窃为先生羞之。丘请为先生往说之。”

图 1　这是手动编号但不出现索引中的图片的例子

柳下季曰：“先生言为人父者必能诏其子，为人兄者必能教其弟，若子不听父之诏，弟不受兄之教，虽今先生之辩，将奈之何哉？且跖之为人也，心如涌泉，意如飘风，强足以距敌，辩足以饰非。顺其心则喜，逆其心则怒，易辱人以言。先生必无往。”

孔子不听，颜回为驭，子贡为右，往见盗跖。

B.1.3　整数规划

盗跖乃方休卒徒大山之阳，脍人肝而餔之。孔子下车而前，见谒者曰：“鲁人孔丘，闻将军高义，敬再拜谒者。”谒者入通。盗跖闻之大怒，目如明星，发上指冠，曰：“此夫鲁国之巧伪人孔丘非邪？为我告之：尔作言造语，妄称文、武，冠枝木之冠，带死牛之胁，多辞缪说，不耕而食，不织而衣，摇唇鼓舌，擅生是非，以迷天下之主，使天下学士不反其本，妄作孝弟，而侥幸于封侯富贵者也。子之罪大极重，疾走归！不然，我将以子肝益昼餔之膳。”

附录 C　其它附录

前面两个附录主要是给本科生做例子。其它附录的内容可以放到这里，当然如果你愿意，可以把这部分也放到独立的文件中，然后将其 `\input` 到主文件中。

个人简历、在学期间发表的学术论文与研究成果

个人简历

xxxx 年 xx 月 xx 日出生于 xx 省 xx 县。

xxxx 年 9 月考入 xx 大学 xx 系 xx 专业，xxxx 年 7 月本科毕业并获得 xx 学士学位。

xxxx 年 9 月免试进入 xx 大学 xx 系攻读 xx 学位至今。

发表的学术论文

[1] Yang Y, Ren T L, Zhang L T, et al. Miniature microphone with silicon- based ferroelectric thin films. Integrated Ferroelectrics, 2003, 52:229-235. (SCI 收录, 检索号:758FZ.)

[2] 杨轶, 张宁欣, 任天令, 等. 硅基铁电微声学器件中薄膜残余应力的研究. 中国机械工程, 2005, 16(14):1289-1291. (EI 收录, 检索号:0534931 2907.)

[3] 杨轶, 张宁欣, 任天令, 等. 集成铁电器件中的关键工艺研究. 仪器仪表学报, 2003, 24(S4):192-193. (EI 源刊.)

[4] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press. (已被 Integrated Ferroelectrics 录用. SCI 源刊.)

[5] Wu X M, Yang Y, Cai J, et al. Measurements of ferroelectric MEMS microphones. Integrated Ferroelectrics, 2005, 69:417-429. (SCI 收录, 检索号:896KM)

[6] 贾泽, 杨轶, 陈兢, 等. 用于压电和电容微麦克风的体硅腐蚀相关研究. 压电与声光, 2006, 28(1):117-119. (EI 收录, 检索号:06129773469)

[7] 伍晓明, 杨轶, 张宁欣, 等. 基于 MEMS 技术的集成铁电硅微麦克风. 中国集成电路, 2003, 53:59-61.

研究成果

[1] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A. (中国专利公开号)

[2] Ren T L, Yang Y, Zhu Y P, et al. Piezoelectric micro acoustic sensor based on ferroelectric materials: USA, No.11/215, 102. (美国发明专利申请号)

综合论文训练记录表

学生姓名		学号		班级	
论文题目					
主要内容以及进度安排	指导教师签字： 考核组组长签字： 年　　月　　日				
中期考核意见	考核组组长签字： 年　　月　　日				

指导教师评语	指导教师签字：_____ 年　　月　　日
评阅教师评语	评阅教师签字：_____ 年　　月　　日
答辩小组评语	答辩小组组长签字：_____ 年　　月　　日

总成绩：_____

教学负责人签字：_____

年　　月　　日