电子科技大学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

专业学位硕士学位论文

MASTER THESIS FOR PROFESSIONAL DEGREE

论文题目 基于 Kubernetes 的容器云平台资源调度策略研究

专业学位类别	**工 程 硕 士**
学　　号	201422240317
作 者 姓 名	**唐 瑞**
指 导 教 师	**于鸿洋　副教授**

分类号 _____ 密级 _____

UDC [注1] _____

学 位 论 文

基于 Kubernetes 的容器云平台资源调度策略研究

（题名和副题名）

唐瑞

（作者姓名）

指导教师 于鸿洋 副教授

电子科技大学 成 都

（姓名、职称、单位名称）

申请学位级别 硕士 专业学位类别 工 程 硕 士

工程领域名称 电子与通信工程

提交论文日期 2017.4.14 论文答辩日期 2017.6.6

学位授予单位和日期 电子科技大学 2017 年 6 月

答辩委员会主席 _____

评阅人 _____

注 1：注明《国际十进分类法 UDC》的类号。

Research on Resources Scheduling Strategy of Container Cloud Platform Based on Kubernetes

A Master Thesis Submitted to

University of Electronic Science and Technology of China

Discipline: **Master of Engineering**

Author: **Tang Rui**

Supervisor: **Associate Prof.Yu Hongyang**

School: **Research Institute Electronic Science and Technology of UESTC**

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

作者签名：_____ 日期：　　年　　月　　日

论文使用授权

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

作者签名：_____ 导师签名：_____

日期：　　年　　月　　日

摘要

以 Docker 为代表的容器技术在应用的开发、发布和部署上具有便捷性和实用性，从出现之初便受到了业界的广泛关注，由于 Docker 本身只注重于提供容器和镜像，因此需要一个集成的容器云管理平台高效地完成容器的编排部署、资源调度、服务发现、健康监控等任务。Kubernetes 凭借其强大的容器编排能力和轻量开源的特点成为了众多容器集群调度系统的领跑者，然而 Kubernetes 的资源调度策略和系统自带的调度算法都较为单一，在复杂的应用场景下往往力不从心。

本文在深入研究 Kubernetes 的核心技术后，对资源调度模块进行改进和设计，主要内容如下：

1.改进了 Kubernetes 的资源模型。Kubernetes 在进行资源调度时只考量了 CPU 和内存的影响，但是 Pod 若要正常运行还需要进行镜像下载，与持久化存储系统进行数据交互，本文在原有模型上增加了镜像下载速度和数据传输速度作为资源调度的考量因素。

2.改进了 Kubernetes 用户绑定策略。Kubernetes 的用户绑定策略较为简单，本文在此基础上设计了一种弱绑定策略，增加了用户绑定的匹配规则，支持对表达式的匹配性周期检测。

3.设计抢占式调度策略。根据重启策略将 Pod 划分为三个优先级，当宿主机资源不足时，高优先级 Pod 可以抢占低优先级 Pod 的资源，有效的提高了高优先级 Pod 的运行比例。

4.设计动态负载均衡调度策略。Kubernetes 默认调度策略中 Pod 一经调度便无法迁移，本文鉴于此设计了一种基于 Kubernetes 的动态负载均衡改进算法，该算法的静态调度负责将待调度 Pod 队列的每一个 Pod 调度到最符合其资源描述文件的节点上；动态调度则通过监控器定期将宿主机和 Pod 的运行状况反馈到调度器，调度器根据系统整体负载情况作出动态调整，将负载较高的节点的一些 Pod 迁移到负载较低的节点上，以维持系统的整体负载均衡。

关键词：Kubernetes，资源调度，用户绑定，抢占式调度，动态调度，负载均衡

ABSTRACT

Container technology, like Docker is easy and practical to develop, publish and deply the application, so it has attracted broad attention in the industry. Due to its focus only on providing container and image, an integrated container cloud management platform is required to efficiently complete a series of tasks such as container scheduling deployment, resource scheduling, service discovery and health monitoring. Kubernetes is the preferred choice in the container cluster scheduling system, but its resource scheduling strategy and scheduling algorithm are too simple to deal with complex applications.

The resource scheduling modle is improved and redesigned after the deep study of Kubernetes technology in this thesis. The main contents are as follows:

1. Kubernetes resource model is improved. Kubernetes only consider the influence of CPU and memory in resource scheduling, but if Pod works well, the data between image download and persistent storage volume are required to exchange. Compared the original one, the new model in this thesis views the image download speed and data transmission speed as the consideration factor in resource scheduling as well.

2. Kubernetes user binding strategy is improved. Its user binding strategy is comparatively simple and strong, therefore a weak binding strategy is formulated based on it in this thesis and meanwhile the matching rules for user binding is enhanced, which supports the matching period detection for expressions.

3. Preemptive scheduling strategy is designed. Pod is divided into three priorities according to the restart policy. When the host resource is insufficient, Pod with high priority can preempt the low priority resources in order to greatly raise the running ratio of Pod with high priority.

4. Dynamic load balancing scheduling stragey is designed. Kubernetes scheduling strategy is static, which means that Pods are not moved from scheduled to node to its end of lifecycle. A dynamic load balancing algorithm based on Kubernetes is formulated by combining the improved resource model and preemptive user binding strategy. Static scheduling of this algorithm is responsible for dispatching each Pod of the Pod queue to the node of most suitable resource description file; and dynamic

scheduling regularly provides the running situation of host computer and Pod through the monitor to the scheduler, and then the scheduler makes dynamic scheduling in accordance with the overall load of system, so that Pod on the higher load node is moved to the low load node to maintain the overall load balancing.

Keywords: Kubernetes, resource scheduling, user binding, preemptive scheduling, dynamic scheduling, load balancing

目 录

第一章 绪论

1.1 研究工作的背景与意义

计算机技术日新月异，其相关技术广泛应用在各行各业中，这些应用产生的海量数据促使大量开发者投入大数据的研究与学习，而对于数据中心的高需求也激发了云计算的大规模发展，现今无论是互联网巨头还是初创型公司都在争夺大数据和云计算的战略高地。

在云计算的三层服务模式中，IaaS 的资源调度单位是虚拟机，因此容易出现资源利用率低、调度运行缓慢、软件运行时环境不一等等问题，而在 IaaS 基础上的传统 PaaS 平台在应用架构和软件环境也存在较大的局限性。不论是 IaaS 平台还是 PaaS 平台都有各自擅长的应用场景，但是不能否认的是它们依然存在很多缺点，云计算领域急需一个真正有效可行的解决方案。

在这样的背景下，以 Docker 为代表的容器技术给云计算甚至是整个 IT 行业带来了新的生命，由于 Docker 在应用程序的开发、发布和部署上具有简单、便捷、实用的特点，大多数知名的国内外厂商早已开始对基于 Docker 的容器云平台做了巨大的投入。

但是 Docker 本身只能提供基本的容器管理功能，如：镜像的制作、下载和上传、容器生命周期的管理等，因此 Docker 需要一个容器集群管理系统来统筹管理分布在集群中的所有容器，从而实现容器的编排、自动化部署、任务调度、资源分配、服务发现等任务。大量优秀的管理系统也随之应运而生，如 Google 公司的 Kubernetes、Docker 自家的 Swarm、Mesos 发布的 Marathon 等，以 Docker 容器技术为核心的第三方容器云平台迎来了前所未有的机遇。

Kubernetes 构建在 Docker 容器技术之上，为用户提供了一个容器化应用的整体解决方案，它具有强大的容器编排能力，遵循微服务架构理论，并且开放开源，现今 Kubernetes 已成为 Docker 生态圈最流行的开源容器集群调度系统。Kubernetes 的资源调度模块是通过可插拔组件的方式实现的，目的是为了让开发人员能够根据实际需求定制资源调度策略，而云平台的资源调度技术是云服务的关键，是云计算得以大规模的应用和提高云平台系统性能并兼顾成本最小化的核心技术。有效合理的资源调度算法，一方面可以提高云服务商的资源利用率并最小化成本，另一方面可以为云用户带来更优质快捷的服务，因此本文着重于对 Kubernetes 资源调度策略的研究。

1.2 国内外研究现状

Kubernetes 是 Google 公司 2015 年正式推出的开源项目，在 Kubernetes 的早期版本中，使用 round robin 算法随机的为 Pod 选择宿主机，而不考虑宿主机的资源使用情况、负载均衡等因素，这种调度算法还容易引起资源冲突（如端口号、存储卷等），无法适应分布式系统的复杂应用场景。现版本的调度模块大幅调整了算法框架，现在系统采用的调度算法是 default 算法，该算法是静态调度算法，在多变的生产环境中依然表现不佳，但是 Kubernetes 的设计理念迎合了众多开发者的需求，它提供了一个可插拔的算法框架，开发者可以方便地向调度器添加自定义的资源调度算法[1]。

正因为 Kubernetes 开放开源的姿态，它从出现之初便受到业界的广泛关注，无论是 Redhat、CoreOS、IBM、华为、阿里巴巴等知名 IT 巨头，还是时速云、有容云、灵雀云等初创型企业都投入了 Kubernetes 的研发生产中，目前国内外的主要研究关注于以下部分：

1.由于现版本的用户绑定调度策略的绑定规则较为简单，Kubernetes 官方在其工作计划中提出将在未来版本中加入亲和性调度，用来代替现版本的用户绑定调度策略：PodFitsHost 算法和 PodSelectorsMatches 算法。亲和性调度包括 Node 亲和性和 Pod 亲和性，将增加 In、NotIn、Exists、DoesNotExist、Gt、Lt 等操作符来选择 Node，这种用户绑定策略更加灵活，同时将增加一些额外信息来设置亲和性调度策略[2]；

2.Kubernetes 中从 Pod 被调度到宿主机上运行开始到其生命周期截止都不会发生迁移，但是随着集群环境的变化，在 Pod 创建之初做出的调度策略在后期运行过程中可能已不再合适，为了维持系统的整体负载均衡，Kubernetes 官方社区有大批 Kubernetes 爱好者提出了重调度的方式，但是重调度策略的整个流程会涉及多个组件之间的协同工作，而并不是简单的设计一个重调度算法，具体设计方案仍在讨论中[3]；

3.Kubernetes 中不存在抢占式调度，所有 Pod 优先级相同，CoreOS 工程师提出了一种为 Pod 指定优先级的方式来进行抢占式调度，这种调度策略的优先级划分并不准确，因为所有用户都希望自己的应用优先级最高，并且这种方式需要在所有 Pod 的资源描述文件中定义优先级，这会导致工作量太大，不适合实际应用场景。

4.Kubernetes 的资源模型较为单一，只将 CPU 和内存作为资源调度的考量因素，阿里云工程师有提出将网络带宽加入到现有模型中，但是在实际应用中，很难直接通过网络带宽来衡量各节点对 Pod 的“适宜度”。

本文据于此现状，展开了对 Kubernetes 云平台资源调度策略的研究与设计。

1.3 本论文的研究内容与创新

本文主要对 Kubernetes 云平台的资源调度模块进行研究，经过研究发现 Kubernetes 现有的资源调度策略存在以下四方面缺陷：

1.Kubernetes 的用户绑定资源调度策略较为简单，其用户绑定的匹配规则较为单一，不适合复杂应用场景；Pod 一旦被调度运行，无论宿主机在后期运行过程是否依然满足匹配规则，Pod 都不会进行迁移；

2.不支持抢占式调度，所有 Pod 优先级相同，如果用户为 Pod 绑定的宿主机资源不足以进行调度，则该 Pod 将无法按匹配规则进行调度；

3.资源模型较为单一，Kubernetes 调度器的 default 算法在计算宿主机的综合得分时只考虑 CPU 和内存两方面影响，但是在实际应用中，Pod 在被调度之后还需要经过两步才能正常运行：第一步，宿主机从镜像库下载 Pod 资源描述文件指定的镜像，镜像下载速度直接影响到 Pod 的启动；第二步，Pod 启动后，需要挂载持久化存储以保证数据的持久化，宿主机与持久化存储间的网络传输速度直接影响容器化应用的读写速度；

4.Kubernetes 中 Pod 从被调度到合适的宿主机上开始，直到其生命周期截止，Pod 都不会在各宿主机间发生迁移。但是随着时间的推进，宿主机上资源的变化，Pod 的创建与删除等种种因素的影响，在 Pod 创建之初调度器做出的调度选择在此刻可能已不在适用。

本论文的主要工作就是围绕这四方面缺陷展开，通过研究 Kubernetes 云平台的资源调度原理和调度策略，改进了 Kubernetes 云平台的资源模型，并在此基础了设计改进了三种调度策略：1.在 PodSelectorsMatchs 基础上丰富了用户绑定资源调度策略的功能；2.根据 Pod 的重启策略划分优先级，设计可抢占式的资源调度策略；3.基于 Kubernetes 云平台改进了一种动态负载均衡算法。最后根据设计的 Kubernetes 云平台搭建集群环境，设计测试用例，验证改进的可行性。

其中创新点主要是以下四点：

1.改进了 Kubernetes 现有的用户绑定调度策略 PodSelectorsMatchs，丰富了匹配规则，支持对匹配表达式的周期检测；

2.提出了一种根据 Pod 的重启策略划分优先级的思路，并在此基础上设计可抢占式的用户绑定调度策略；

3.改进 Kubernetes 的资源模型，提出了在 CPU、内存的基础上加入镜像网络下载速度和持久化存储数据传输速度作为资源调度的衡量因素；

4.针对 Kubernetes 不支持动态调度的缺陷，本文结合 Kubernetes 云平台的特点改进了一种动态负载均衡算法，将调度算法分为静态调度部分和动态调度部分，该算法在集群环境变化时依然能够较好的维持系统的负载均衡，算法执行效率较 default 算法相差不多。

1.4 本论文的结构安排

本论文共有七章，下面分别介绍各章主要工作。

第一章介绍了容器及容器生态圈的背景，并引出 Kubernetes 资源调度的研究意义，并分析了 Docker 和 Kubernetes 云平台的国内外研究现状，提出本文主要工作和创新点。

第二章介绍了 Docker 和容器云平台的关键技术，由此引出了容器集群资源调度系统，对典型的调度系统进行分析对比，并重点介绍了 Kubernetes 容器集群资源调度器。

第三章在研究 Kubernetes 调度器和调度算法的基础上，结合实际应用场景，对 Kubernetes 云平台资源模型加以改进并提出了三种资源调度策略：用户绑定、抢占式调度和动态负载均衡，最后定制了一个适合本文研究的 Kubernetes 云平台。

第四章着重于设计第三章提出的用户绑定策略和抢占式调度策略，通过对 Kubernetes 现有的用户绑定算法的研究，设计了一种匹配规则多样，并支持对匹配表达式的周期检测；根据 Pod 的重启策略划分优先级并设计抢占式的用户绑定调度策略；

第五章根据第三章改进的资源模型和动态负载均衡算法设计了基于动态负载均衡的优选策略；之后设计了一种基于 Kubernetes 的动态负载均衡改进算法，并从静态调度和动态调度两方面阐述具体设计思路。

第六章搭建 Kubernetes 容器云平台，设计测试用例，测试上文设计的调度策略的正确性，并分析改进算法的性能。

第七章对本文的改进与设计进行总结，并分析本设计急需解决的问题，展望未来工作方向。

第二章 Kubernetes 云平台相关技术综述

Kubernetes 是在 Docker 的基础上建立的容器集群管理系统，本章从 Docker 容器和容器云平台出发，介绍三类容器资源调度系统，并对其典型代表 Swarm、Mesos、Kubernetes 进行对比分析，最后重点介绍了 Kubernetes 的核心概念和架构。

2.1 Docker 与容器云平台

由于以 Docker 为代表的容器技术在应用的开发、发布和部署上具有便捷性和实用性，从出现之初便受到了业界的广泛关注，以 Docker 为基础的容器云平台也由此发端。

2.1.1 Docker 与虚拟化技术

容器不是一个新概念，容器就是给应用程序提供运行环境的独立空间，独立空间中的进程拥有独立的文件系统、网络栈、进程组等，还可以为这个独立空间分配 CPU 和内存等资源。不同容器中的进程相互隔离，因此在运行时不会发生干扰和冲突。管理员可以对容器中的应用程序进行全局环境配置和监控。

容器与虚拟机都是虚拟化技术，它们之间是互补的，虚拟机利用硬件虚拟化技术（如 VT-x、AMD-V 等）进行硬件资源划分，同时使用 hypervisor 层来实现资源隔离。容器则是操作系统级虚拟化技术，通过软件的方式（Cgroup 和 Namespace）来进行资源隔离。

Docker 是将容器进行封装，主要负责进行容器生命周期管理、相关信息的管控与查询。如图 2-1，Docker 没有传统虚拟化技术中的 hypervisor，Docker 虚拟化的实现是基于 Linux 内核的 Cgroup 和 Namespace 技术，它省去了 hypervisor 层的开销，因此它是轻量级的虚拟化技术[5]。

Docker 不会直接与操作系统内核进行通信，而是通过 libcontainer 进行交互，libcontainer 是容器引擎，它通过 clone 命令来创建 Docker，通过操作 groupfs 文件实现资源管控，这样 Docker 就可以专注于处理上层业务。

2.1.2 容器云平台

Docker 容器技术的出现催生了容器云平台的诞生，容器云平台进行资源调度和分割的基本单位是容器，它封装应用软件的运行时环境，并为开发者和管理员提供了一个应用的构建、发布和运行平台，当容器云平台专注的领域不同时所表

现出的特点也不同，当它侧重于容器编排与部署、资源共享与隔离时表现的特点像 IaaS，而当它关注于应用支撑和软件运行时环境时则表现的更像 PaaS。

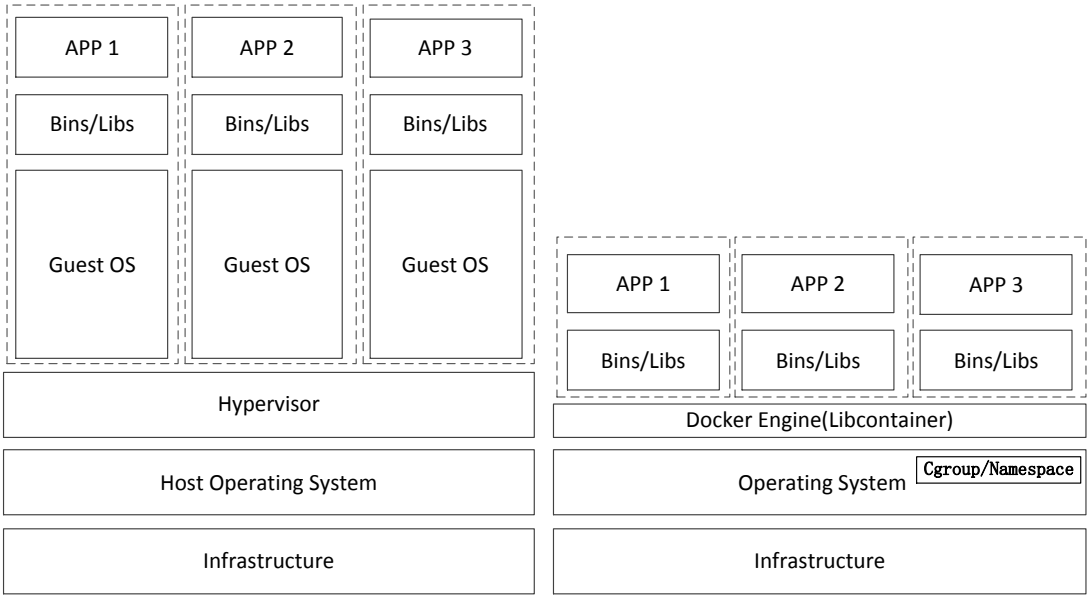

图 2-1 Docker 架构图

传统的云计算平台层次结构由下到上可以分为三层：基础设施即服务（IaaS，Infrastructure as a Service）、平台即服务（PaaS，Platform as a Service）和软件即服务（SaaS，Software as a Service）。IaaS 层负责云平台的所有资源虚拟化工作，通过软件定义的方式为用户提供虚拟资源（CPU、网络、存储等）；PaaS 层则主要负责所有应用的运行时环境和应用支撑，这样云平台的用户就可以申请这些计算单元用以部署和运行应用程序；SaaS 则是将运行的软件或服务通过 API 响应的方式传送给成千上万的用户。

如今很多 PaaS 平台已经使用容器技术并以容器为计算单元，如图 2-2，PaaS 平台用容器为应用程序提供运行时环境和系统依赖，容器在 IaaS 平台的基础上进行第二层次基础设施资源的划分；与此同时，用户也使用容器进行应用的调度和构建多实例集群。不同于 IaaS 平台，一般在 PaaS 平台使用更贴近应用程序的资源调度策略。然而，使用这个体系结构构建的传统 PaaS 平台存在一个尴尬的问题：用户在使用服务时始终无法知道容器的具体运行情况。这就会导致当一些异常情况发生时，用户不知道具体原因，更不知道怎么解决。例如在一些传统的 PaaS 平台中一旦应用程序在运行过程中发生异常情况，云平台会先删除故障的应用，然后在其它位置恢复这个应用和容器，在整个过程中，如果用户不进行干预，PaaS 平台甚至没有保存现场。

随着 Docker 容器技术逐步进入开发人员的视野，云平台的建设已经有了新的方向。传统的 PaaS 云存在限制严格、自由度较低的缺点，为了改善这些不足，在沿用了部分传统 PaaS 云平台优点的同时，增加了 PaaS 云平台的自由度。传统 PaaS 平台为用户提供基于具体语言或框架的环境，而改进的 PaaS 云为用户提供语言或框架的容器，容器是可选择的，灵活性更大。改进的 PaaS 云平台介于 IaaS 和 PaaS 之间，带有很强的容器特征，所以被称为容器云平台[7]（CaaS，Container as a Service）。

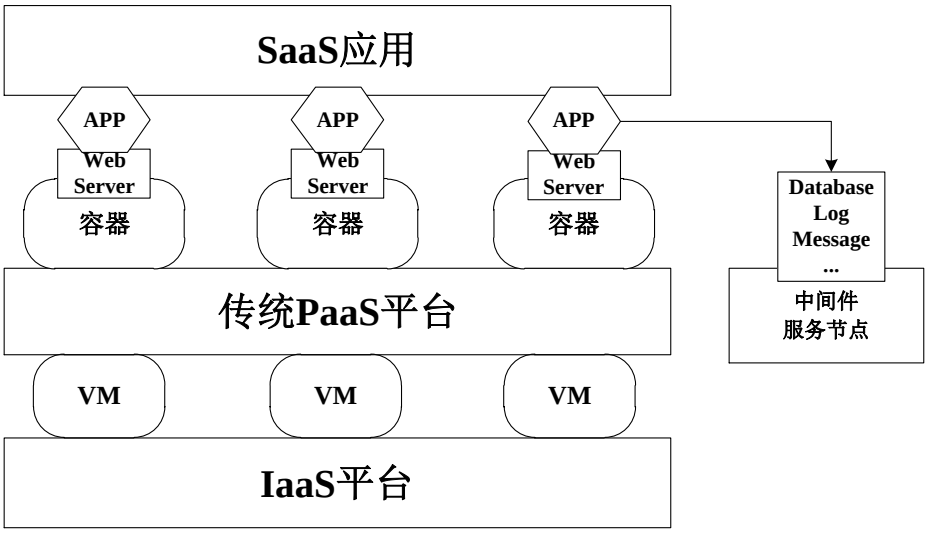

图 2-2 传统云平台层次结构

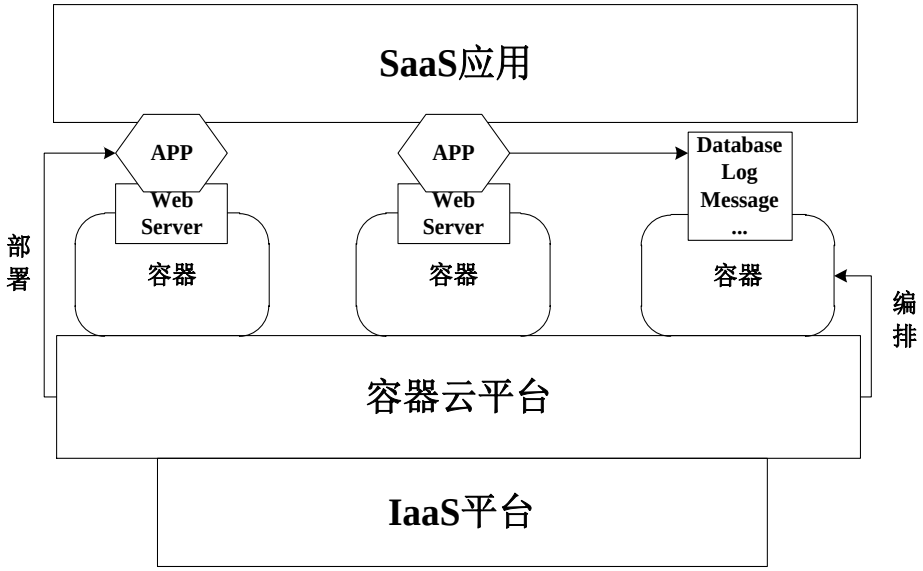

图 2-3 容器云平台层次结构

容器云平台层次结构如图 2-3，IaaS 平台可以是虚拟资源或物理资源，如：虚拟机、物理机、虚拟化存储等。容器云平台主要负责分布式配置和协同、服务发现、资源调度调度、节点管理、负载均衡、反向代理、容器编排部署、资源管控、

自动化发布和 API 控制等任务，SaaS 层是运行在容器云平台上的服务和应用。

2.1.3 容器云平台关键技术

容器云平台基本都是围绕容器型应用的运行时管理展开的，核心技术包括下述几方面[8]：

应用封装：Docker 是现今使用最多的虚拟化技术，与其它已经存在的容器技术相比，它更加轻量和完善，Docker 可以更加简单地创建和管理容器，因此可以使用 Docker 容器技术对应用进行封装。

资源调度：仅有应用的封装是远远不够的，使用 Docker 将应用封装完后还需要为容器提供 CPU、内存、网络、存储等资源需求。大规模数据中心有成千上万台服务器，在资源分配时到底为应用分配哪台或哪几台服务器，这是资源调度策略决定的，一个好的资源调度策略对用户和云服务商都是至关重要的。

应用分发：主要涉及到应用的更新上线等。例如，在进行应用的不停机更新时，老版本应用正在使用，这时可以使用 Docker 镜像仓库存储新版本应用。

网络管理：容器化应用的设计原则是面向服务的，它倡导将各个功能点设计为组件，这样可以更方便的对应用进行扩展和管理。这种设计理念也对应用组件间的网络功能和可靠性提出了更高的要求[9]。

服务发现：服务发现的目的是使容器化应用部署更具有伸缩性和灵活性，对于容器的调度不是一次性的，是需要动态调整的，容器之间需要通信以了解相互的需求，并根据需求系统做出动态调整。

面对容器云平台如此繁杂的技术，容器云平台的生态圈也在不断的加入新成员，其中不乏一些优秀的容器集群资源调度系统，下面对这些解决方案展开介绍。

2.2 容器集群资源调度系统

容器云平台是一个具有一定规模的容器集群，为了合理有效地使用集群的资源，满足差异化的数据服务和多样的任务的资源请求，需要一个集群资源调度系统对容器集群进行管理调度。现今比较流行调度系统有：Swarm、Borg、Mesos、Kubernetes 等，这些系统从系统架构的角度可以分为：集中式调度系统、两层式调度系统和共享状态调度系统[12]。

2.2.1 集中式调度系统

集中式调度系统的代表有 Swarm 和 Borg 等，集中式调度系统使用统一的调度算法进行任务调度，这些调度任务不可以并发执行。并且集中式调度系统的所有

集群相关信息存储在调度器上，这个调度器是唯一的，当集群信息出现爆炸性增长时，会使调度器负载较重，甚至会大幅降低系统的吞吐量。

Swarm 是 Docker 官方的集群管理工具，如图 2-4，在 Swarm Master 节点上运行两个模块，Discovery 负责发现集群中的节点，Scheduler 模块则负责为新创建的容器分配最优的节点。

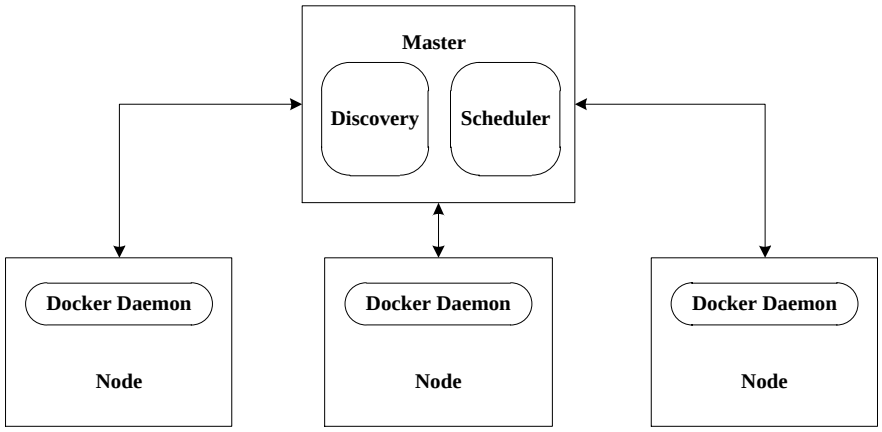

图 2-4 Swarm 架构图

Scheduler 支持 Random、Spread 和 Binpack 三种调度策略，其中 Random 策略随机选择目标节点进行 Docker 调度；Spread 和 Binpack 则根据节点可用的 CPU、存资源量和节点上当前运行的 Docker 数量做出调度决策，Spread 将 Docker 分散调度到各个节点上，以提高系统的整体负载均衡；Binpack 与 Spread 相反，将 Docker 调度到负载最高的节点上，提高集群中节点承载容器的密度，这三种调度策略各自的缺点也是显而易见的，在实际应用部署中很难使用[10,11]。

2.2.2 两层式调度系统

在两层式资源调度系统中，调度框架之间可以相互独立互不干扰地并行执行任务，每一个调度框架独立实现具体的调度算法，资源调度器只负责资源分发，与集中式调度系统最大的差别就是，两层式系统使用资源管理器管理集群所有的资源信息，并为调度框架提供计算资源。

Mesos 和 YARN 是两层式调度系统的代表，尤其是 Mesos 是现今最火的容器集群管理器之一，如图 2-5，Mesos Master 负责承上启下的作用，对下负责管理 Mesos Slave 节点，对上则负责为调度框架提供资源。调度框架之间使用不同的调度算法并行执行调度任务，Mesos Slave 负责运行具体的任务，Mesos Master 还使用 ZooKeeper 实现高可用性。

Mesos 调度器是根据 DRF（Dominant Resource Fairness）算法实现的，DRF 算

法对不同类型作业的不同资源需求设计了一种公平分配的算法。该算法的核心思想是在资源类型多样的环境下，一个作业的资源分配量由作业的主导份额资源决定，主导份额资源是作业请求的各类型资源中占据资源量最大的一种资源。

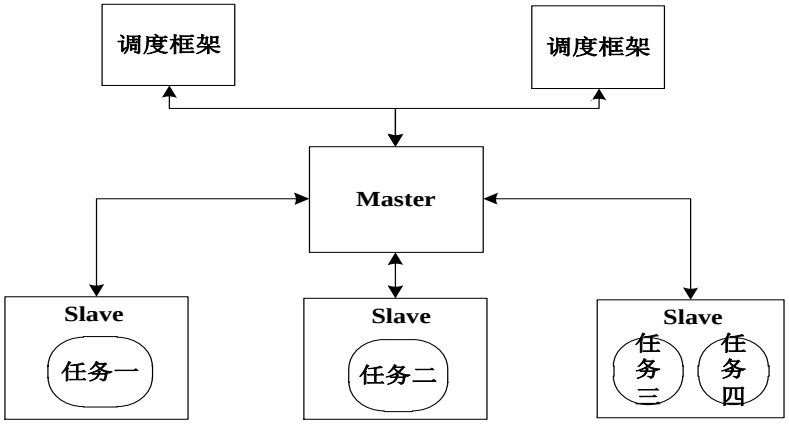

图 2-5 Mesos 架构图

下面结合图 2-6 的伪代码举例说明 DRF 算法，集群中共有 9 核 CPU 和 18GB 内存，A、B 两种作业请求的资源量分别为（1CPU，4GB）和（3CPU，1GB），对于 A 作业需要消耗 1/12 的系统 CPU 和 1/6 的系统内存，因此 A 作业的主导份额资源是内存；而 B 作业的消耗系统的资源分别为 1/4 和 1/24，因此 B 作业的主导份额资源是 CPU。DRF 将均衡左右作业的主导份额资源，A 获得 2/3 的内存而 B 获得 2/3 的 CPU，A、B 分别获得的资源量为（3CPU，12GB）和（6CPU，2GB）。此时集群可运行三个 A 作业和两个 B 作业。

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
    $C = C + D_i$ ▷ update consumed vector
    $U_i = U_i + D_i$ ▷ update i's allocation vector
    $s_i = \max_{j=1}^{m} \{u_{i,j}/r_j\}$
else
    return ▷ the cluster is full
end if

图 2-6 DRF 算法伪代码[12]

2.2.3 共享式调度系统

共享式调度系统也支持使用不同的调度框架执行不同类型的任务，它的核心是共享状态。为了提高调度系统的并发性和可扩展性，共享式调度系统使用乐观锁进行并发控制，集群的相关信息都增加了版本号，在提交的时候与当前数据的版本号进行对照，若提交的状态信息版本号比当前信息的版本号高，则允许此次提交，否则决绝提交请求，这样虽然会增加资源请求的冲突几率，但是实际应用证明，系统的整体性能并不会因为这些冲突而整体下降。

共享式调度系统的集大成者无疑是 Kubernetes，Kubernetes 利用谷歌公司在容器领域的多年技术积累，同时吸取了 Docker 社区的最佳实践，已经成为云计算服务的舵手。

在 Kubernetes 的早期版本中，使用 round robin 算法随机的为 Pod 选择宿主机，而不考虑宿主机的资源使用情况、负载均衡等因素，这种调度算法还容易引起资源冲突（如端口号、存储卷等），无法适应分布式系统的复杂应用场景。现版本的调度模块大幅调整了算法框架，现在系统采用的调度算法是 default 算法，该算法是静态调度算法，在多变的生产环境中依然表现不佳，但是 Kubernetes 的设计理念迎合了众多开发者的需求，它提供了一个可插拔的算法框架，开发者能够便宜地向调度器添加自定义的资源调度算法[13,14]。

2.2.4 综合分析

集中式调度系统使用统一的调度算法进行任务调度，这样不利于向调度增加新的调度算法；不支持作业的并发调度；所有的集群相关信息都存储在单一的调度器上，随着集群规模的扩大，爆发式的集群信息会使唯一的调度器负载过重，从而影响系统的整体吞吐量。

两层式调度系统与集中式调度系统相比进步很大，但是在实际应用场景中依然存在一些不足。各个调度框架只能获取集群的部分相关信息，这就使开发者无法为调度框架设计根据整个集群的状态信息做出决策的调度算法。两层式调度系统使用悲观锁机制来维持集群信息的一致，这样会导致各调度框架的并发性受到限制。

共享式调度系统也支持使用不同的调度框架执行不同类型的作业，它使用乐观锁来提高调度器的并发性和扩展性，虽然这样会增加资源请求的冲突数量，但是并不至于影响系统的整体性能。

如表 2-1 对三种调度系统的代表进行的对比分析，与其它调度器相比，Kubernetes 有着如下优秀特性：

1.轻量级，Kubernetes 遵循微服务架构理论，将整个系统划分为各个功能独立的组件，组件之间分工明确，可以轻易地部署在各种发行版的 Linux 系统中。

2.开放开源，为了吸引大批开发者和公司参与其生态圈的构建，Kubernetes 从一开始便开放了源代码。同时 Kubernetes 中很多功能模块（如资源调度器 Scheduler）都实现了插件化，便于研究者进行二次开发。

3.强大的容器编排能力，Kubernetes 深度集成了 Docker 和 Rkt，设计了 Pod、Label Selectors 等组件，是一个强大的容器调度系统。

表 2-1 Swarm、Mesos、Kubernetes 对比

	Swarm	Mesos	Kubernetes
支持的集群规模	10K+	50K	1K+
开源	是	是	是
可编程	是	是	是
多调度器支持	否	是	是
设计框架	单策略集中式调度框架	多策略双层调度框架	多策略共享状态框架

结合上文的总体分析，本文采用共享式调度系统中的 Kubernetes 资源调度器作为研究对象。

2.3 Kubernetes 简介

Kubernetes 是 Google 公司于 2015 年 7 月开源的容器集群管理系统，它从出现之初便受到了业界的广泛认可。Kubernetes 基于 Docker 容器技术构建了一个容器云平台，为应用程序提供了资源调度、健康监控、均衡容灾、部署运行、服务发现、扩容缩容等一整套功能。

2.3.1 Kubernetes 核心概念

Kubernetes 中大部分基本概念都是资源对象，如：Node、Pod、RC、Service 等，这些资源对象都可以使用 kubectl 执行增删改查等操作，操作结果存入持久化存储系统 etcd。

1.Pod

Pod 是 Kubernetes 的进行基本操作和资源调度的最小粒度，它由一个或多个相关的容器组合而成。

在实际的应用场景中，应用并不会部署在一个孤立的容器中，而是部署在一

组相互关联的容器组里，例如在部署一个基于 LNMP 的 WEB 服务时，通常会将 Nginx 服务器、MySQL 数据库、PHP 解释器分别放在不同的容器里，容器之间相互通信完成数据传输。在这种实际应用背景下，Kubernetes 采用了 Pod 容器组将一组功能相关联的容器作为一个整体进行资源管理和调度。

一个 Pod 中的多个容器通常是紧耦合的，这些容器运行在同一台宿主机上，它们使用相同的 IP 地址和端口，这样 Pod 里的容器就可以与其它节点上的 Pod 容器直接通信，除了这些，Pod 中的容器共享的资源资源还包括：

PID 命名空间：Pod 中的不同应用进程之间可以相互查看彼此的进程 PID；

网络命名空间：Pod 中的各个容器可以访问同一个 IP 地址和端口；

IPC 命名空间：Pod 中容器之间使用 System VIPC 或 POSIX 消息队列进行通信；

UTS 命名空间：同一 Pod 中的所有容器共享同一个主机名；

Volumes：Pod 中的各个容器可以访问在 Pod 级别定义的存储卷。

Pod 有两种类型：静态 Pod 和普通 Pod，普通 Pod 在被创建后相关信息会存入 etcd，由控制节点上的调度器调度绑定到到某个节点上，该节点的 kublet 进程在宿主机上将该 Pod 启动并运行，静态 Pod 不在 etcd 上存储，只存放在宿主机的具体文件里。

2.Label

Label 是形如 key=value 的键值对，其中键值可以由用户自定义。Kubernetes 中所有的资源对象都可以添加标签（Kubernetes 中大部分概念都是资源对象，如 Node、Pod、Service 等）。Label 的创建方式有两种：1.对象的资源描述文件中定义；2.对象创建后动态修改。

除了添加和删除，Label 最主要的操作是查询，可以使用 Label Selector 进行查询，Label Selectors 类似于 SQL 语句中的 where 查询，有基于等式和基于集合两种表达式。

3.Replication Controller

Replication Controller（简称 RC）主要用来管理 Pod 的副本，使副本数保持在设定值，当定义了一个 RC 资源对象后，控制节点上的 Controller Manager 进程就会定期查询集群中所有的目标 Pod 的数量，确保目标 Pod 数量与 RC 资源描述文件中 spec.replicas 数量相等，如果副本数过多，Controller Manager 会杀死一些目标 Pod，反之 Controller Manager 会常见一些目标 Pod。

4.Service

Service 等同于微服务，提供了一种服务代理和发现机制，Servcie 为 Pod 指定

了一个逻辑集合，并指定了访问该 Pod 集合的方式。对外，Servic 提供了单一访问接口；对内，Service 代理 Pod 集合使外层用户只需要关心服务而不需要关心 Pod 的具体运行。

2.3.2 Kubernetes 的架构和组件

Kubernetes 遵循微服务架构理论，整个系统划分出各个功能独立的组件，组件之间边界清晰，部署简单，能够方便的在各种操作系统和环境中运行。Kubernetes 是主从分布式结构的，其节点在角色上分为 Master 节点和 Node 节点。

如图 2-7，Master 节点是 Kubernetes 的控制节点，负责整个系统的调度和管理，主要包含三个组件：

1.API Server：API Server 是 Kubernetes 集群的控制节点，主要负责响应用户请求和各组件的协调工作，其封装了 Pod、Node、RC、Service 等核心对象的增删改查操作，提供了一个 RESTful 接口给外部用户和各组件内部调用，并将其维护的 REST 对象将放入 etcd 中进行持久化存储。

2.Scheduler：调度器负责集群的资源调度，为待调度 Pod 队列的每一个 Pod 绑定宿主机。Scheduler 是一个可插拔的组件，可以用其他调度器替换。

3.Controller Manager：负责操控 Kubernetes 的多种控制器，是包括 Repilcation controller（副本控制器）、Node Controller（节点控制器）等在内的控制器集合。

Node 是 Kubernetes 的工作节点，主要用于运行 Pod，主要包括以下组件：

1.kublet：kublet 负载管理控制 Pod，主要负责两方面任务：一方面从 API Server 接受创建 Pod 的请求并在宿主机上启动该 Pod，另一方面，负责监控 Pod 的运行情况并将相关信息返回给 API Server。

2.Proxy，Proxy 主要为 Pod 集合创建代理服务，它从 API Server 获取 Service 的资源描述文件，并根据相关描述创建代理服务负责处理 Service 到 Pod 的路由和转发。在实际使用中，Proxy 经常会出现问题，用户大多会选择其他的流量转发组件。

3.Docker：Pod 中运行的容器，上文已有介绍，这里不在赘述。

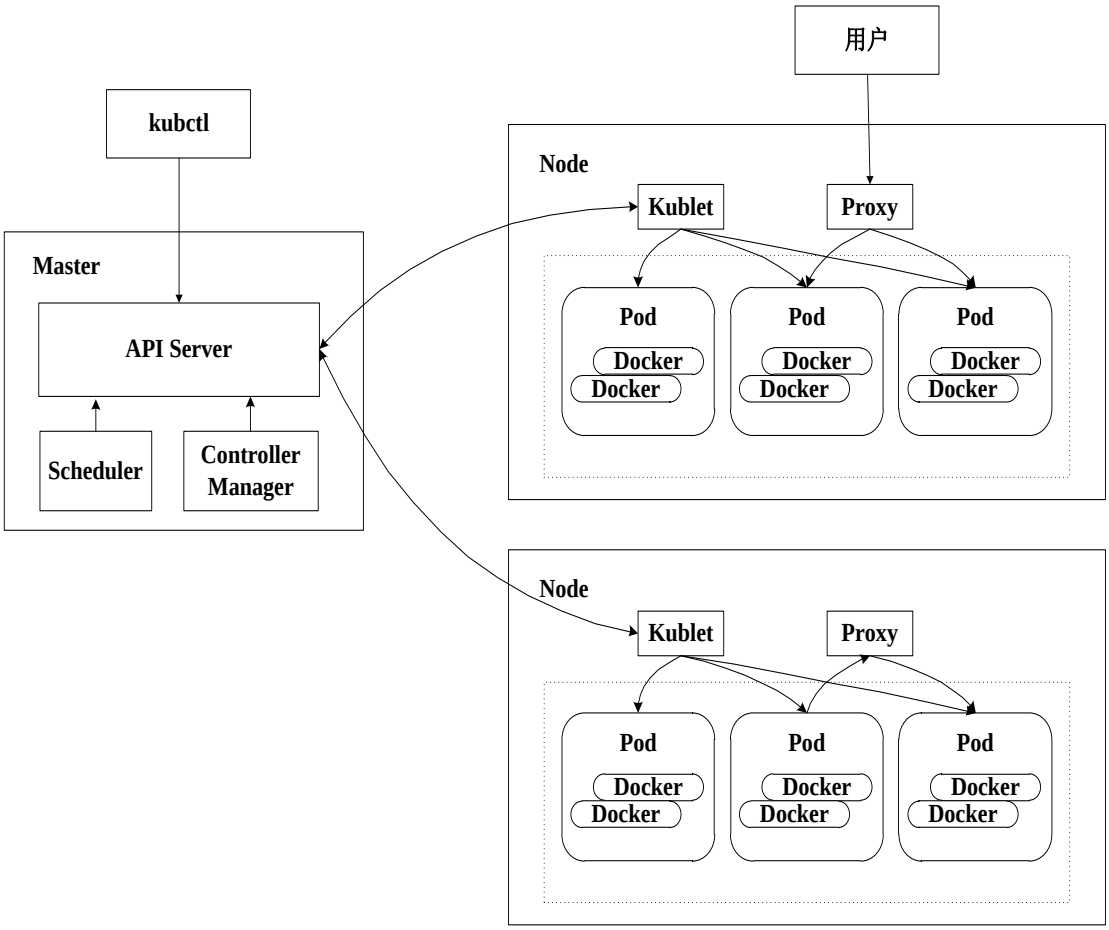

图 2-7 Kubernetes 系统架构图

2.4 本章小结

本章从 Docker 技术出发，分析了 Docker 及以 Docker 为基础的云平台的核心理念和关键技术，从而引起了对一些主流的容器集群资源调度系统的学习，并最终选择 Kubernetes 作为本文的研究对象，最后对 Kubernetes 的核心概念和架构进行介绍，为下文的 Kubernetes 资源调度策略研究奠定了基础。

第三章 Kubernetes 云平台资源调度策略研究与设计

Kubernetes 现版本采用的调度算法是 default 算法，该算法是静态调度算法，在多变的生产环境中表现不佳，但是 Kubernetes 的设计理念迎合了众多开发者的需求，它提供了一个可插拔的算法框架，开发者可以根据自己的实际需求定制调度策略。本章将对 Kubernetes 云平台资源调度策略深入研究，分析 Kubernetes 资源调度策略的不足，根据实际应用场景提出改进设计。

3.1 Kubernetes 云平台资源调度策略研究

本文主要研究的容器云平台是以 Kubernetes 为基础的，如图 3-1，Kubernetes 是管理 Docker 容器的集群管理系统，而 Docker 主要关注于提供容器和镜像，而无论是互联网巨头亦或是初创型公司，甚至是普通的用户，他们都需要一个集成的容器云平台高效地完成应用封装、资源分配、任务调度、自动化部署、服务发现、网络管理、健康监控等任务[13]，让用户能透明地享用容器技术带来的便利。

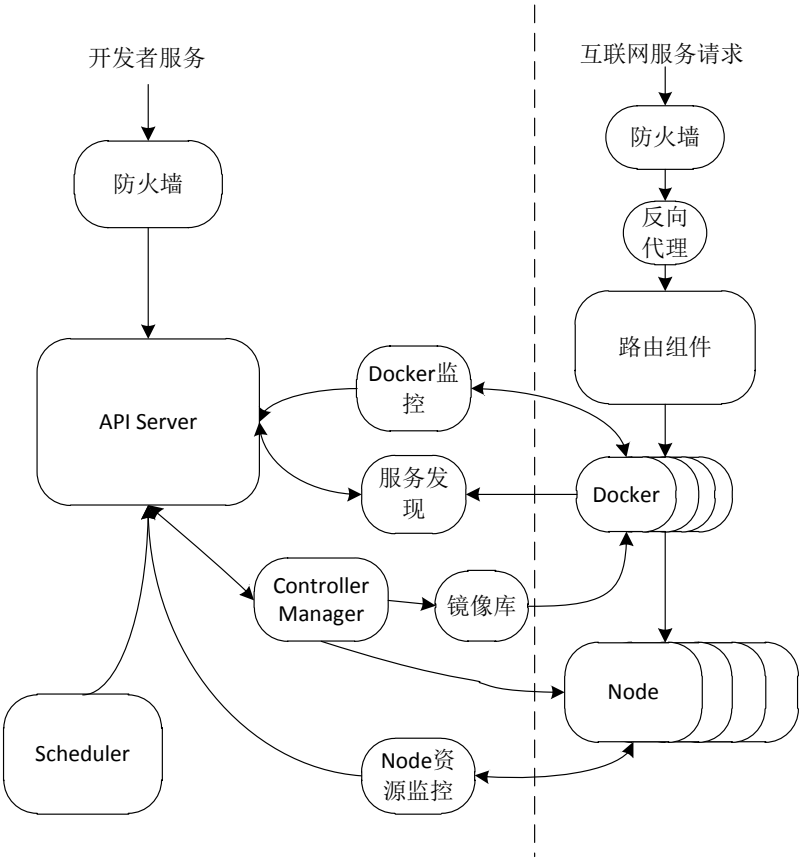

图 3-1 Kubernetes 云平台数据流图

云服务运营商希望在为用户提供可靠服务的同时使成本最小化，而用户则关注于获得优质的服务，这就需要一个有效的资源调度策略来解决上述问题。本节主要关注的就是 Kubernetes 云平台的最关键的技术——资源调度。

Kubernetes 在进行资源调度时需要统计所工作节点的资源使用情况，根据工作节点的具体情况给予评价，在评价最高的节点上将宿主机资源分配给 Pod 使用，并且要保证 Pod 在其生命周期内都有足够的资源保持运行，其中最核心的组件是调度器 Scheduler。

Kubernetes 调度器在整个资源调度的过程中承担了承上启下的重要作用，调度器负责接收新创建的 Pod，并为其指定目标节点；调度成功后，目标节点上的 Kublet 服务进程接管后续工作，负责 Pod 的管理，并将 Pod 的运行状况定期汇报给调度器。

具体来说，Kubernetes 调度器的职责是将 API Server 或 Controller Manager 新建的待调度 Pod 根据指定的调度算法或调度策略与集群中某个合适的工作节点进行绑定，并将 Pod 和 Node 的绑定信息写入 etcd 中。在整个调度过程中涉及三个对象，分别是：待调度 Pod 队列、可用 Node 队列和调度策略（算法）。简单地说，就是通过调度策略（算法）依次为待调度 Pod 队列的每一个 Pod 从可用 Node 队列中选择一个最合适的作为宿主机[17,18]。

随后，目标节点上的 Kublet 进程通过 API Server 监听到 Kubernetes 调度器发出的 Pod 绑定事件，然后从 etcd 中获取对应的 Pod 资源描述文件，下载 Image 镜像，启动容器，挂载持久化存储系统，完整的调度流程如图 3-2。

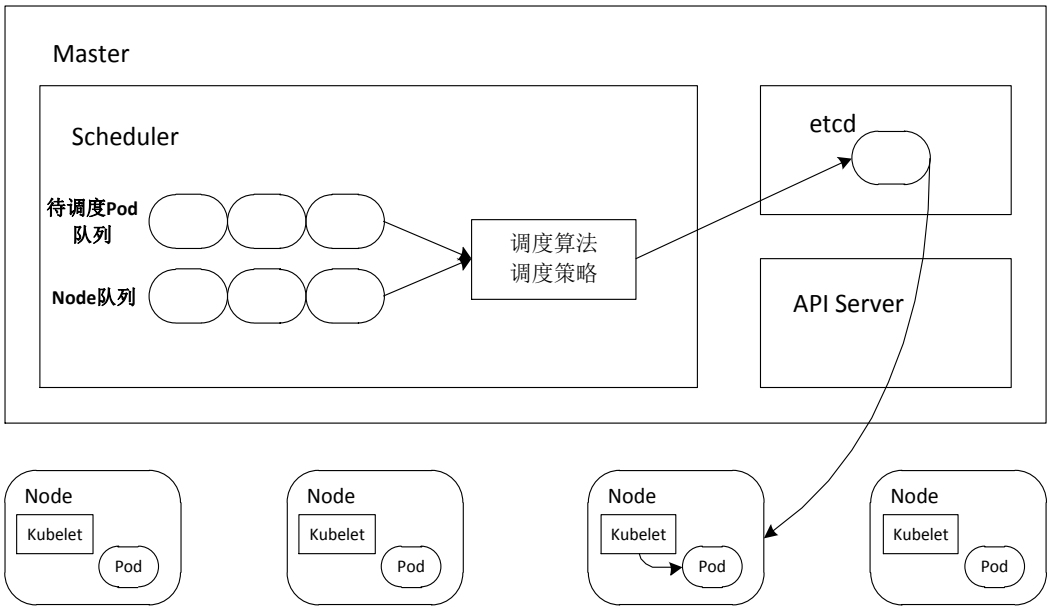

图 3-2 Scheduler 流程

3.2 default 算法研究

Kubernetes 默认采用的调度算法是系统自带的 default 算法，Scheduler 调度器提供了一个可插拔的算法框架，开发者可以根据实际需求设计调度算法或设计 default 算法的调度策略。

Kubernetes 调度器的 default 算法分为以下两步：

1.Predicates 策略，预选过程，即遍历集群所有可用的工作节点，筛选出符合预选策略的节点；

2.Priority 策略，优选过程，在第 1 步的基础上，采用优选策略计算出每个候选节点的分数，得分最高的节点成为调度目标。

3.2.1 Predicates 策略

default 算法中预选策略有五个，每个 Node 只有通过 PodFitsPorts、PodFitsResources、NoDiskConflict、PodSelectorMatches、PodFitsHost 层层过滤之后，才能进入候选队列，等待进入 Priority 过程，各预选策略简单说明如下：

1.PodFitsPorts：检查待调度 Pod 的端口是否已被该节点上其它 Pod 占用，若存在端口冲突，则过滤掉该节点。

2.PodFitsResources：检查该节点是否有足够的资源运行待调度 Pod。

3.NoDiskConflict：检查在该节点上是否存在卷冲突，若待调度 Pod 与节点上已运行的 Pod 挂载相同的存储卷，则过滤掉此节点。

4.PodSelectorMatches：若 Pod 设置了 NodeSelector 属性，则筛选出 labels 值与 NodeSelectors 匹配的节点。

5.PodFitsHost：若 Pod 设置了 HostName 属性，则筛选出指定的 Node。

3.2.2 Priority 策略

经过预选过程筛选出了符合条件的 Node 来运行 Pod，如果存在多个符合条件的 Node，那么需要选择出最优的 Node。对于每一个过滤之后的 Node，优先级函数给出一个分数：0~10（10 表示最优，0 表示最差），而每个优先级函数有权重值，Node 的最终分数就是每个优先级函数给出的分数进行加权的和，比如有两个优先级函数 *priority1* 和 *priority2*，它们的权重值分别是 *weight1* 和 *weight2*，那么该节点的得分是：

$$score = (priority1 * weight1) + (priority2 * weight2) \tag{3-1}$$

计算出各节点的最终得分后，对其按得分高低进行排序，得分最高的节点就是目标节点，如果存在多个节点分数并列第一，就随机选择一个节点进行调度。

Kubernetes Scheduler 提供的默认优选策略有：

1.LeastRequestedPriority，选择资源消耗最小节点：

(1)根据已运行的 Pod 和待调度 Pod 的资源请求量分别计算出所有候选节点的 CPU 总量 *totalMilliCPU* 和内存总量 *totalMilliMomory*；

(2)计算出各候选节点的 *cpuScore* 和 *memoryScore*，计算方法如下：

$$cpuScore = \frac{nodeCpuCapacity - totalMilliCPU}{nodeCpuCapacity} \times 10 \tag{3-2}$$

$$memoryScore = \frac{nodeMemoryCapacity - totalMilliMemory}{nodeMemoryCapacity} \times 10 \tag{3-3}$$

其中 *nodeCpuCapacity* 和 *nodeMemoryCapacity* 分别表示候选节点的 CPU 和内存资源总量，若 *cpuScore* 和 *memoryScore* 值小于 0，则结果直接返回 0；

(3)计算各候选节点的最终得分，*L_score* 是 *cpuScore* 和 *memoryScore* 的算术平均值，公式如下：

$$L\_score = \frac{cpuScore + memoryScore}{2} \tag{3-4}$$

该策略考虑了 Node 的 CPU 和内存的负载，但是忽略了工作节点的 CPU 和内存的资源均衡使用率，为此 Kubernetes 设计了 BalancedResourceAllocation。

2.BalancedResourceAllocation，选择资源使用最均衡节点

(1)根据已运行的 Pod 和待调度 Pod 的资源请求量分别计算出所有候选节点的 CPU 总量 *totalMilliCPU* 和内存总量量 *totalMilliMomory*；

(2)计算出所有候选节点的 CPU 使用率 *cpuUse* 和内存使用率 *memoryUse*，计算方法如下：

$$cpuUse = \frac{totalMilliCPU}{nodeCpuCapacity} \tag{3-5}$$

$$memoryUse = \frac{totalMilliMemory}{nodeMemoryCapacity} \tag{3-6}$$

若计算得出某候选节点的 CPU 或内存使用率大于等于 1，则该节点最终得分为 0。

(3)计算各候选节点的最终得分，因为 CPU 和内存的使用率差额在 0-1 之间，故需要乘以 10，使节点的最终得分在 0 到 10 之间，计算规则如下：

$$B\_score = 10 - |cpuUse - memoryUse| \times 10 \tag{3-7}$$

候选节点的 CPU 和内存的使用率差额越越小则节点该项得分越高，当节点的

CPU 和内存使用率相同时，此时该节点各项资源使用最均衡，均衡度得分为 10。

经过上述两个步骤，节点的总得分计算公式如下：

$$finalScore = L\_score \times 1 + B\_score \times 1 \tag{3-8}$$

上式表示调度器在进行得分计算时，综合考量剩余资源量和资源均衡程度，这两个因素的影响权值因子都是 1。

图 3-3 展示了 default 算法的调度示例，集群中各节点经过 Predicate 预选策略过滤掉不符合要求的节点，然后经过 Priority 优选过程计算各节点的最终得分，其中得分最高的即为目标节点，所有的调度策略都在 Policy 资源描述文件中定义。

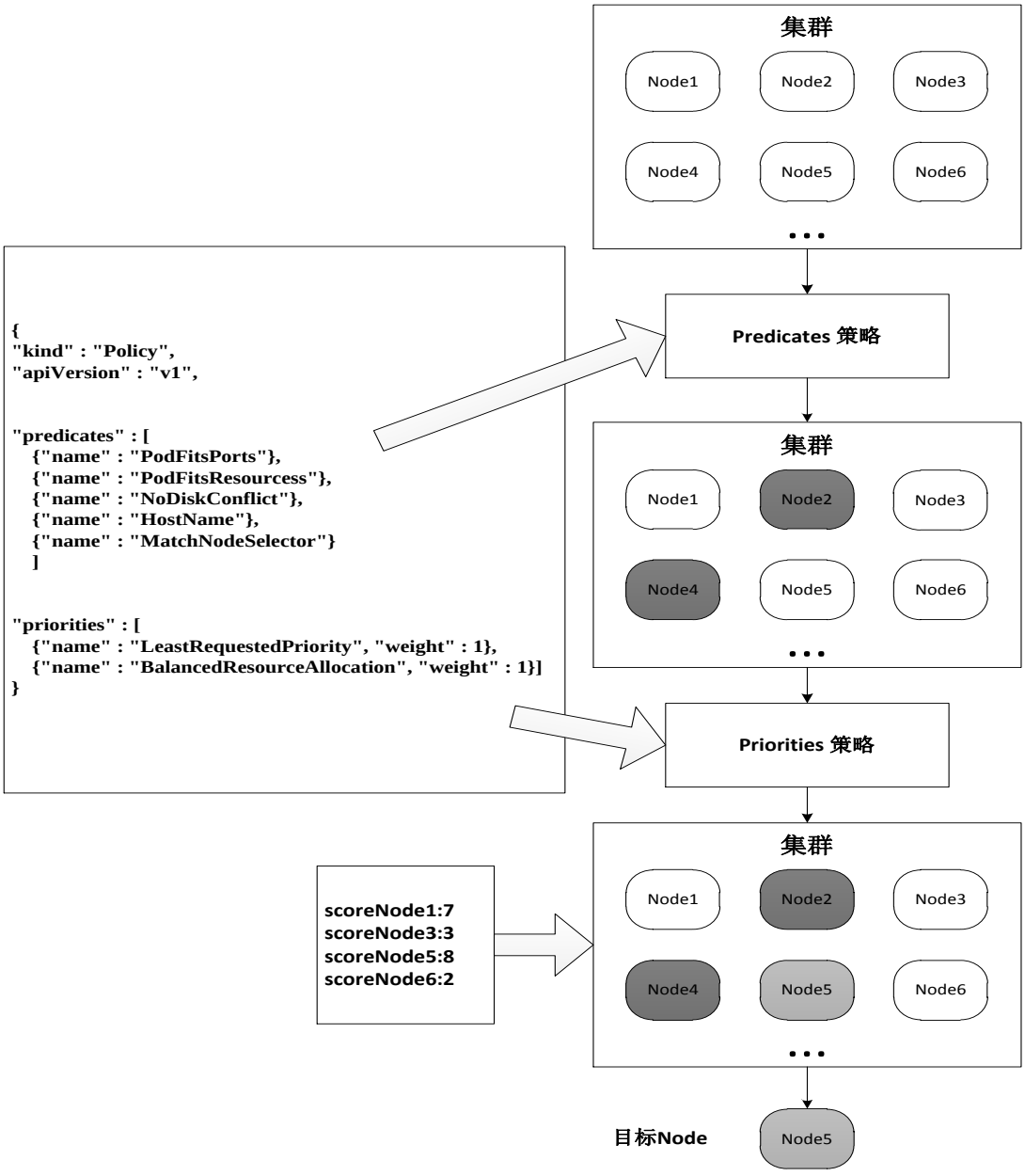

图 3-3 default 算法调度示例

3.3 Kubernetes 云平台资源调度策略设计

上文对 Kubernetes 资源调度策略的研究可以发现，Kubernetes 的资源模型只有 CPU 和内存，资源调度策略也较为简单，并不能解决很多实际问题，下文将根据实际应用场景对 Kubernetes 调度策略进行改进和设计。

3.3.1 Kubernetes 云平台资源模型改进

为了研究 Kubernetes 云平台的资源调度策略，首先需要对 Pod 宿主机的资源建立模型，便于量化 CPU、内存、网络、存储等因素对调度策略的影响。

容器和虚拟机都是虚拟化技术，虽然在实现方式上完全不同，但二者的资源需求和模型是相似的，容器和虚拟机一样也需要 CPU、内存、存储和网络。宿主机系统像虚拟机一样将容器看作成一个整体，为容器分配其所需要的资源。

在 Kubernetes 中主要的资源是 CPU 和内存，其中 CPU 的基本单位是核 core，内存的基本单位是字节 Byte。计算资源可以在 Pod 的配置阶段为每个容器指定，相关配置项分为两种：Requests 是容器请求量，表示容器运行的最低保证量，为避免某个容器无限制占用资源，还需要设定 Limits，表示容器资源使用的上限值。容器的实现资源使用量介于 Requests 和 Limits 之间，每个容器都可以指定 CPU 和内存的请求量和限制量，Pod 的资源量是其所包含的所有容器的资源量之和[19-21]。

现版本的 Kubernetes 调度器在进行资源调度时主要考量因素就是 CPU 和内存，但是这种资源调度方式忽略了网络因素对 Pod 在宿主机上启动运行的影响。控制节点的 Scheduler 组件在将待调度 Pod 和宿主机绑定后，宿主机还需要考虑下述两方面因素才能将 Pod 启动并正常运行：

1.宿主机需要到 Pod 资源描述文件指定的网络地址下载 Pod 中所有容器的镜像文件，宿主机与镜像存储系统之间的网络传输速度直接关系到 Pod 的启动速度的快慢；

2.Pod 中数据是临时的，当 Pod 销毁时，其中的数据会丢失，所以 Pod 需要通过数据卷的方式将数据持久化。因此 Pod 在启动运行后，Pod 还需要挂载持久化存储系统进行数据的存取，宿主机与持久化存储之间的数据传输速度会直接影响 Pod 中运行的应用的 IO 速度。

本设计将综合考量 CPU、内存、镜像下载速度，数据传输速度四方面因素，为 Pod 选择最合适的宿主机。

容器云平台由多个数据中心组成，一个数据中心又由若干集群构成，每个集群拥有一定数目的宿主机，宿主机运行的是 Pod。在本设计中为了方便研究，不考虑集群架构对容器云平台的影响，为此简化了上述模型，只将 Pod、宿主机（Node）、

集群作为研究对象，如图 3-4 为本文研究的对象。

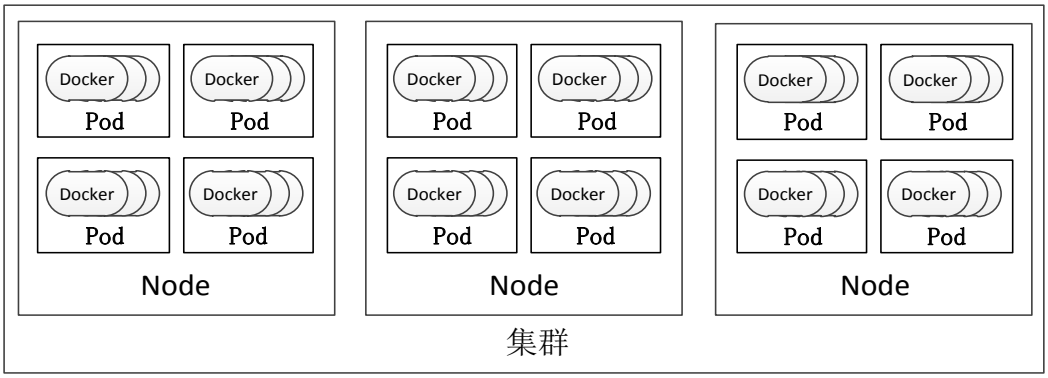

图 3-4 集群、Node、Pod 与 Docker 的关系示意图

3.3.2 资源调度策略设计

Kubernetes 资源调度策略通过 Pod 的“期望状态”和当前环境的“实际状态”的差异实现资源调度，Pod 的期望状态是通过资源描述文件体现的，然而现版本的资源描述文件存在以下问题：用户绑定策略简单；不支持抢占式调度；随着集群环境的变化，符合 Pod 创建之初请求资源量的宿主机在后期运行中可能已不再适合[22-24]。

3.3.2.1 用户绑定调度策略

在实际应用场景中，经常需要将指定的 Pod 分配到指定的工作节点上，例如，有的互联网公司为了方便管理，会将工作节点按功能进行划分，有的节点专门负责测试，产品在正式上线前就需要在专门负责测试的节点上运行，等测试通过再发布到生产环境的工作节点上运行。这就需要设计一种用户绑定调度策略。

Kubernetes 中 default 算法的 PodFitsHost 和 PodSelectorMatches 就是用于用户绑定的，PodFitsHost 直接指定 Node，这是一种强绑定，如果不存在该 Node 则会导致调度失败。而 PodSelectorsMatches 相对于直接指定的方式多了一点灵活性，该算法可以方便的将一些 Pod 调度到某些节点上。

但是在实际应用场景中，这种匹配规则还是过于简单，在以下四个场景中，PodSelectorsMatches 的不足之处显露无疑。

1.在 PodSelectorsMatches 中如果指定多条 nodeSelectors 属性，这几条属性是“且”的关系，但是在实际应用中不仅需要“且”而且需要“或”、“非”等其它指令。

2.在 Pod 调度完成后，Node 的标签值可能会改变，而这时候有的用户选择不理会该变化，继续在 Node 上运行该 Pod，而有的用户则想将该 Pod 重新调度到合

适的节点上，甚至有的用户想直接回收销毁该 Pod。

3.在 PodSelectorsMatches 中，如果 Pod 未找到合适的 Node 作为调度目的地，调度器会将其放入待调度队列的队尾，等待下次重新被调度，但是在实际应用中，用户可能急需该 Pod 调度运行，这时候用户就不再关心指定宿主机这件事上，只想要该 Pod 被尽快调度。

因此本文将根据上述场景，改进 PodSelectorsMatches，设计一种匹配规则多样的弱绑定资源调度策略。

3.3.2.2 抢占式调度策略

如果集群中存在 lables 值与 Pod 指定的 multiNodeSelectors 属性值相匹配的节点，但此节点的资源状况并不足以让该 Pod 创建并调度运行，此时可以选择忽略 Pod 设置的标签属性，按照 Kubernetes 自带算法选择最优节点作为调度目的地，这可以通过上节设计的用户绑定资源调度策略来实现。但是在某些场景中，必须选择指定的 Pod 及时的运行在匹配的节点上，这就需要释放该节点的一些资源，将优先级低的 Pod 调度到其它空闲节点上，为优先级高的 Pod 腾出资源。

然而，Kubernetes 并不支持可抢占式调度，在 Kubernetes 中所有 Pod 的优先级相同，因此需要根据 Pod 中运行的任务类型来判定各 Pod 的优先级，并根据划分的优先级设计抢占式调度策略。

如图 3-5，在集群资源不足的情况下，选择将一部分低优先级的 Pod 资源回收，直至宿主机的资源状况满足高优先级 Pod 的资源请求，并将回收的 Pod 现场信息保存至持久化存储系统，等到被重新调度运行时恢复至被回收前的状态。本文在此场景下设计一种抢占式调度策略。

3.3.2.3 动态负载均衡调度策略

Kubernetes 在进行资源调度是根据各工作节点上报的资源使用情况和 Pod 创建时请求的资源量进行的。从一个 Pod 被 Scheduler 调度到合适的工作节点上开始，到其生命周期截止，Pod 都不会在各工作节点间发生迁移。

但是随着时间的推进，工作节点上资源的变化，Pod 的创建与删除等种种因素的影响，在 Pod 创建之初调度器做出的调度选择在此刻可能已不在适用。在实际的应用场景中，往往需要将一个正在某一个工作节点运行的 Pod 迁移到一个与调度策略匹配程度更高的新工作节点上，例如下述几个场景：

1.Pod 创建之初所申请的资源量与实际使用的资源量远不相符，用户都希望自己的应用有足够的资源可以使用，所以在为 Pod 创建资源描述文件时，通常会为

Pod 申请远多于其实际运行所需的资源量，这就导致了集群中的大量的资源被浪费；

2.随着 Pod 的启动与回收，各节点中会出现越来越多的资源碎片，有些资源碎片由于太小无法分配给其它 Pod 使用而被浪费；

3.集群中各工作节点负载不均衡，有些工作节点负载很高，有些则负载很低。

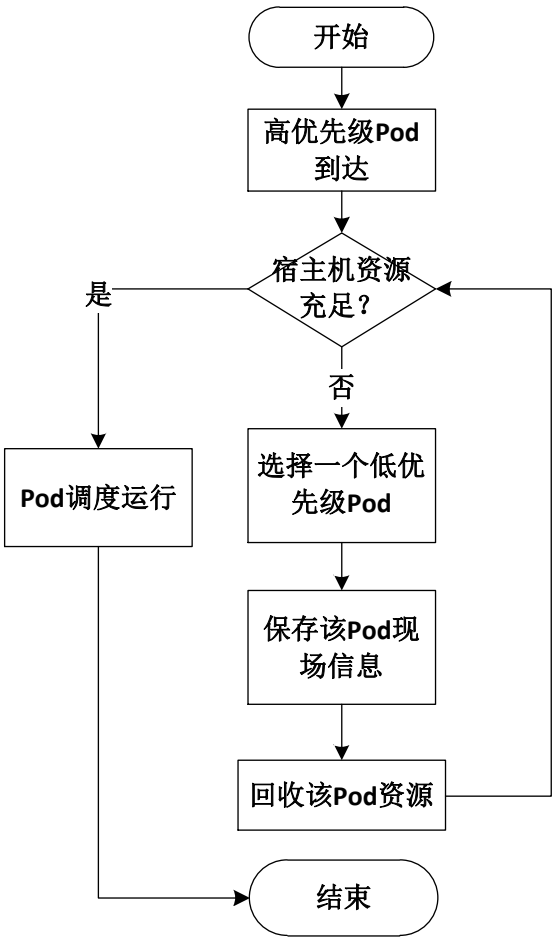

图 3-5 抢占式调度策略

为了解决上述问题，在进行资源调度的同时，系统需要周期检测各工作节点的负载情况，根据负载信息动态调整各工作节点的 Pod，具体步骤如下：

1.工作节点向控制节点汇报资源使用状况；

2.在控制节点上创建 Pod，在 Pod 的资源描述文件中指明 Pod 需求，如：申请的资源量、挂载的存储卷等，并放入待调度 Pod 队列；

3.调度器从待调度 Pod 队列取出 Pod，根据 Pod 的资源描述文件，调度算法为 Pod 选择最合适的节点；

4.待调度 Pod 在目标节点上调度运行；

5.监控程序定期采集 Pod 和宿主机的性能信息，并将这些信息存入持久化数据库 etcd；

6.分析程序从数据库读取 Pod 和其宿主机的性能数据，进行处理运算，处理后将相关信息反馈到调度器；

7.调度器根据集群整体负载信息进行资源的动态调整。

本文结合 Kubernetes 云平台的特点并使用动态负载均衡策略改进了一种基于 Kubernetes 的动态负载均衡算法，如图 3-6 该改进算法主要包括两部分：

1.静态调度：主要负责当待调度 Pod 队列不为空时，从中取出 Pod 为其选择最合适的宿主机；

2.动态调度：监控器定期将宿主机和运行任务的相关信息反馈到 etcd，调度器从 etcd 读取数据，并根据集群总体负载情况作出动态调度，将负载较高的节点的一些 Pod 迁移到负载较低的节点上。

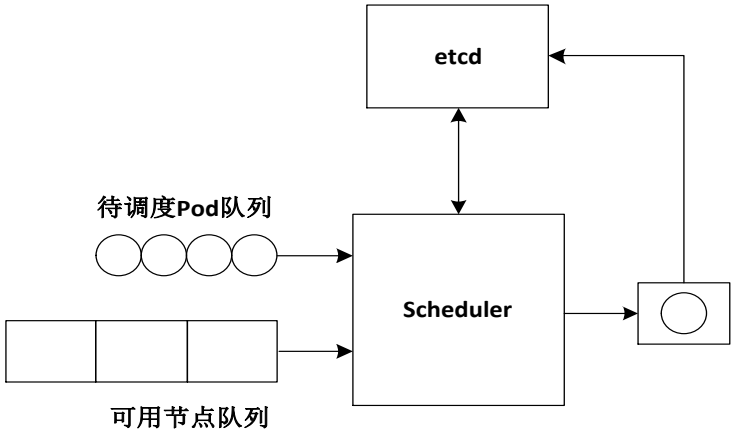

图 3-6 动态负载均衡调度

3.3.3 Kubernetes 云平台设计

为了对改进的资源模型和调度策略进行研究分析，需要根据定制一个这样的 Kubernetes 云平台：1.各 Node、Pod 和 Docker 可以相互通信；2.需要一个有效的监控收集工具，监视系统中各 Node、Pod 和 Docker 的资源状况；3.需要一个平台日志工具来收集分析系统各关键组件的日志文件；4.各节点与 Docker 镜像仓库网络连接状况良好；5.实现 Pod 数据和系统关键数据的持久化[25-29]。为此本节构建如图 3-7 的云平台。

如图 3-7，本文设计的 Kubernetes 主要包括以下部分：

1.API Serrver：

系统中所有资源对象（Node、Pod 等）的控制操作都是通过 API Server 调用完成的。

2.调度器 Scheduler：

Scheduler 是资源调度的核心，它通过 API Server 获取集群状态信息，根据调

度策略和算法为待调度 Pod 选择目标节点。

3.数据存储 etcd：

ectd 是一种键值存储系统，在本设计中，etcd 一方面负责存储 API Server 传入的 Pod 与 Node 的绑定信息，一方面还需要存储 Kubelet 反馈的工作节点负载信息。

4.覆盖网络 Flannel：

使用 Flannel 实现 Kubernetes 覆盖网络，Flannel 为各节点设定一个子网，通过隧道协议封装容器之间的通信报文，实现容器的跨主机通信。

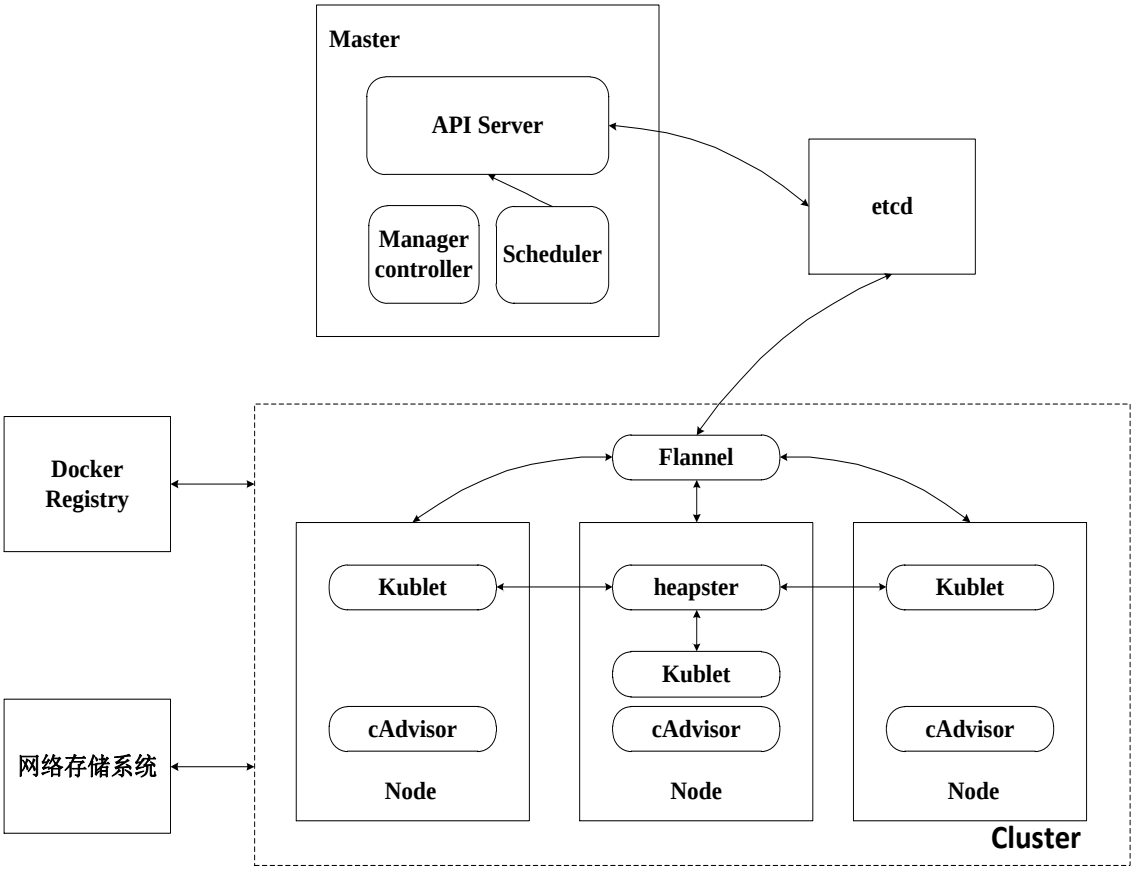

图 3-7 Kubernetes 云平台设计图

5.平台监控 cAdvisor 和 Heapster：

cAdvisor 是一个容器监控工具，它是一个守护进程，对 Node 和 Pod 的运行数据进收集统计，包括实时和历史的 CPU、内存、硬盘、网络使用情况。cAdvisor 集成在 Kubelet 中，在工作节点上可以为 cAdvisor 设置端口，通过端口号可以在宿主机直接访问 cAdvisor。

Heapster 容器集群的监控收集工具，主要通过调用 API Server 获取所有可用节点列表，然后调用 Kubelet 的 cAdvisor 收集汇总监控数据，将收集到的数据建立一

个 Kubernetes 监控模型，监控模型主要包括 Cluster、Node、Pod、container 四层。

6.平台日志 Fluentd：

主要负责管理所有节点的关键组件的日志文件，本设计使用 Fluentd 开源日志收集系统，Fluented 运行在所有的节点上，收集 Kubernetes 各组件的日志文件，然后设置上 tag，最后以 JSON 格式输出。

7.镜像仓库 Docker Registry：

Pod 被创建后还需要下载指定镜像才能正常启动运行，为此设计一个本地镜像仓库，将本文所用的 Docker 镜像存入本地镜像库，便于后期测试使用，如本文的测试用例 Guestbook 应用。

8.网络存储系统：

在 Kubernetes 系统中，Pod 重建时，数据是会丢失的，所以需要为 Pod 挂载持久化存储系统，便于进行动态调度时，将待迁移 Pod 的相关数据存入网络存储系统，待重调度完成后，重新挂载原目录，恢复 Pod 迁移前的状态。

3.3.1 改进的资源模型中，CPU 和内存使用信息可以通过 Heapster 直接获取，而镜像下载速度可以通过日志系统定期采集工作节点与镜像仓库间的平均传输速度来计算，持久化存储数据传输速度同理可用工作节点与网络存储卷的平均传输速度来衡量[26]。

3.4 本章小结

本章研究了 Kubernetes 的资源调度策略和调度算法，改进了 Kubernetes 云平台的资源模型，在进行资源调度的时候综合考量 CPU、内存、镜像网络下载速度和持久化存储数据传输速度四方面因素的影响。然后分析了 Kubernetes 默认策略的缺陷，结合实际应用场景，设计了用户绑定调度策略、抢占式调度策略和动态负载均衡度调度策略，并为本文的后续研究定制了 Kubernetes 云平台。

第四章 用户绑定和抢占式调度策略的设计

Kubernetes 现版本的用户绑定调度策略是一种强绑定方式，匹配规则较为简单，不支持对匹配规则的周期性检测，并且不支持抢占式调度，当宿主机资源不足时，Pod 则无法在宿主机上调度运行，本章对 Kubernetes 的用户绑定调度策略展开深入研究，改进了一种用户绑定调度策略，并根据 Pod 的重启策略划分优先级，综合设计了一种抢占式的用户绑定调度策略。

4.1 Kubernetes 中用户绑定调度策略的研究

在我们实际应用中经常需要把指定的 Pod 调度到指定的服务器上运行，例如一些公司将服务器集群按功能或部门分类，这就需要将特定功能的 Pod 调度到特定的机器上运行。在早期版本中，Kubernetes Master 节点上的 Scheduler 组件是按照 default 繁杂的算法为 Pod 选择一个最佳目标节点，这个过程是自动完成的，用户无法指定 Pod 的目标节点。在 Kubernetes 目前的版本中有两种方式可以实现用户绑定从而为 Pod 指定宿主机，分别是：PodFitsHost 算法和 PodSelectorsMatches 算法，下面从资源定义文件开始，分析 Kubernetes 用户绑定调度策略。

4.1.1 资源描述文件

在 Kubernetes 中，Pod 和 Node 都可以看作是资源对象，对于这些资源对象使用.yaml 格式（类似于 JSON）的文件来描述。下面的例子就是名为 backend 的 Pod 的资源描述文件：

```
apiVersion: v1
kind:Pod
metadate:
   name: backend
   labels:
    hostName:node1
spec:
    containers:
    - name: node.js-backend
       image: gcr.io/google_containers/pause:2.0
       ports:
```

- containerPort:8000

这是一个资源类型为 Pod 的资源配置文件，metadata 里的 name 属性是 Pod 的名称，labels 属性表明该 Pod 拥有一个 Label 标签：hostName=node.js1，spec 属性申明了 Pod 包含的容器组，该 Pod 定义了一个名为 node.js-backend，镜像存储在 gcr.io/google_containers/pause:2.0 的容器，然后在 8000 端口上启动容器服务。

4.1.2 PodFitsHost 策略

如果待调度的 Pod 已经在某个 Node 上运行过一次，后面该 Pod 因为一些原因被回收，现在需要重新调度，这时应该将该 Pod 尽量调度到之前运行过的 Node 上。PodFitsHost 算法主要通过 Pod 的 spec.nodeName 属性指定的 Node 名与候选的 Node 名称进行匹配，如果匹配一致则该 Node 适合调度，否则不适合。

具体算法流程如下：

1.若待调度 Pod 的 spec.nodeName 属性为空，则返回 true，表明该 Pod 未指定宿主机，按默认策略执行调度，不为空则执行步骤 2；

2.若 Pod 的 spec.nodeName 属性与 Node 的名称匹配，则返回 true 执行步骤 3，若不匹配，则返回 false，调度失败。

3.用调度器的其它预选策略：NoDiskConflict、PotFitsResources、PodFitsPorts 判断该节点是否适合调度，若是则进行调度，否则调度失败。

4.1.3 PodSelectorsMatches 策略

在 Kubernetes 中，不仅 Pod 具有 labels 标签，Node 也具有 labels 标签，该算法主要是通过判断候选 Node 的 labels 标签是否与待调度 Pod 的 nodeSelectors 属性相匹配，因为 labels 属性值并不唯一，所以匹配的结果可能不止一个。

具体算法流程如下：

1.若待调度 Pod 的 spec.nodeSelectors 属性为空，表明该 Pod 对宿主机并无具体要求，调度器按 default 算法执行调度，若该属性值不为空则步骤 2；

2.若宿主机的 labels 属性值与 Pod 的 spec.nodeSelectors 属性相匹配，则执行步骤 3，不匹配表明该节点不符合调度要求，过滤此节点；

3.用调度器的其它预选策略：NoDiskConflict、PotFitsResources、PodFitsPorts 判断该节点是否适合调度，若是则将该 Node 放入候选节点集合并执行步骤 4，否则过滤此节点；

4.若候选节点为空，则调度失败。不为空则计算候选节点集合上各工作节点的最终得分，得分最高的节点为目标节点，如果存在多个得分一样的节点，则随机

选择一个作为该 Pod 的目标节点。

4.1.4 综合分析

如图 4-1，Kubernetes 进行用户绑定调度策略时主要依据是根据 Pod 的资源描述文件的 nodeName 属性和 nodeSelectors 属性，当二者不为空时，则遍历集群所有可用节点，找出与资源描述文件匹配的节点集合，再通过其它 Predicate 策略和 Priority 策略选出最佳节点进行调度。

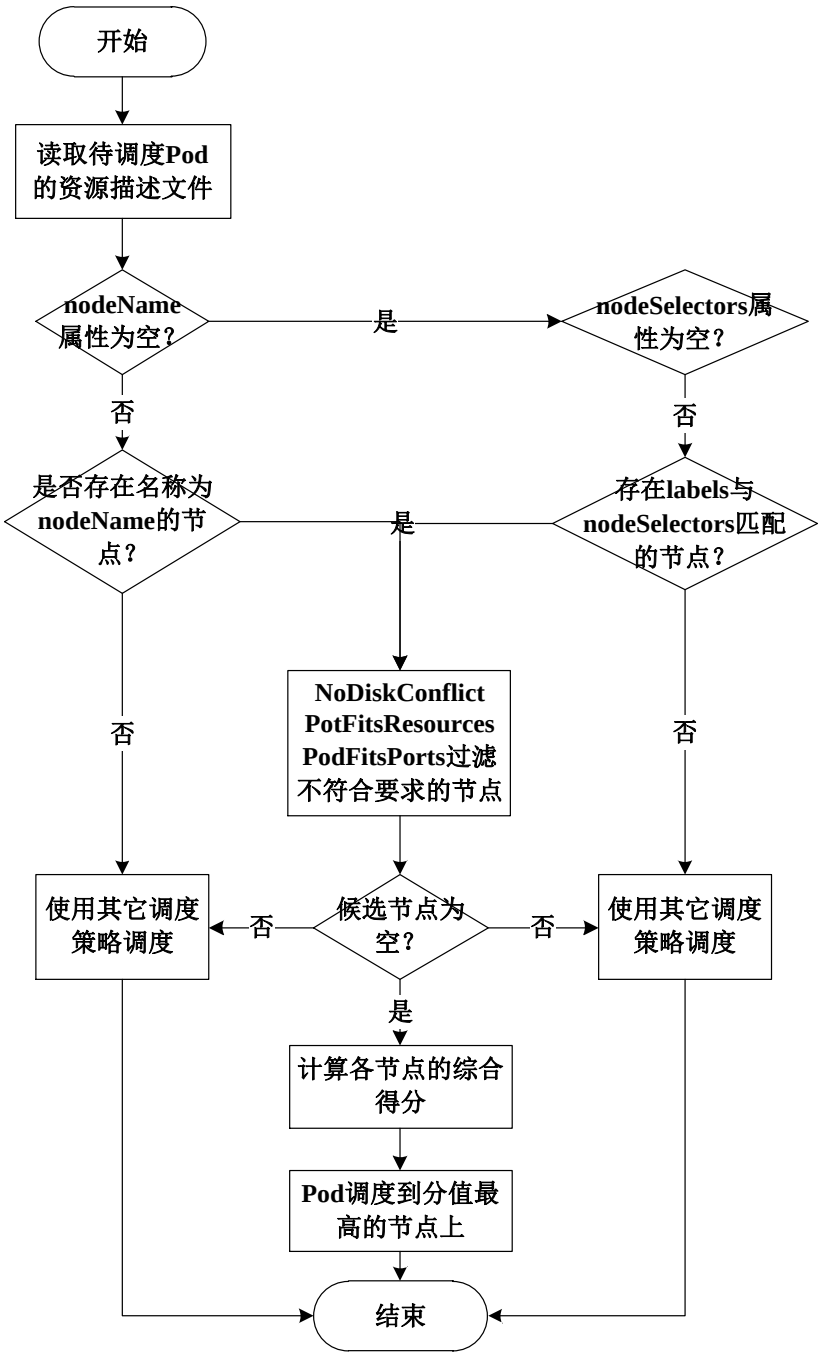

图 4-1 Kubernetes 用户绑定调度策略

上述的调度方式是一种强绑定方式，匹配规则较为简单，在选择节点的过程中，一旦工作节点的相关属性值与 Pod 资源描述文件不匹配，或者属性匹配但资源不足，调度器都会无视一切用户需求，选择其他调度策略进行调度。显然这种调度方式不适合复杂的应用场景，下文将结合实际场景改进 Kubernetes 的用户绑定策略，并设计一种抢占式调度方式。

4.2 用户绑定调度策略的设计

本节通过改进 Pod 的资源描述文件，在 PodSelectorsMatches 基础上设计了一种改进的用户绑定调度策略，以应对用户在为 Pod 指定宿主机的复杂场景。

4.2.1 Pod 和 Node 资源描述文件的设计

Kubernetes 里的所有资源对象都是采用 yaml 或者 JSON 格式的文件来定义描述的。因此如果要使用本算法将指定 Pod 调度到指定的节点上，需要为 Pod 和 Node 添加相应的匹配标签。

直接使用 kubectl 命令行为指定 Node 添加标签。例如为服务器名为 node.js-backend 的节点指定标签 hostname 值为 node.js-backend。，使用如下命令：

kubectl label nodes node.js-backend hostname= node.js-backend

然后为 Pod 的描述文件添加一个 multiNodeSelectors 字段来表示匹配的具体规则。例如我们可以指定一个名为 node.js-backend 的 Pod 定义如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: node.js-backend
  "multiNodeSelectors": {
  "nodeLabelChange": "still",
  "collectionIsEmpty": "default",
    "expressions": [{
        "key": "kubernetes.io/hostname",
        "operator": "in",
        "values": ["apache", "nginx"]
    }]
}
spec:
```

```
containers:
- name: node.js-backend
  image: gcr.io/google_containers/pause:2.0
```

相关字段的具体含义如下：

1.expressions 是一个对象数组，存储一个或多个表达式，这里的设置表明只有 Node 的 labels 中包含 key= kubernetes.io/hostname，并且其 value 为“apache”或者“nginx”时才能成为调度的目标主机。其中 in（属于）操作符表示“或”运算，另外还有操作符“notIn”表示不属于。此处只有一个表达式，多个表达式之间是且的关系。

2.nodeLabelChange 属性设置为 still 表示在后期运行中当节点的标签发生更改时，导致 Pod 的匹配表达式不再匹配时，该 Pod 将仍在该节点上运行，若设置为 reshedule，则表示当节点的标签发生更改时，Pod 会重新调度到合适的目标节点上，还可以设置为 kill，表示节点的标签发生更改将回收此 Pod。

3.collectionIsEmpty 属性表示按 expressions 表达式查找出的候选节点集合为空时调度器的应对方式。有两个可选值，default 表示，在挑选目标节点时最好按照匹配表达式的规则寻找最佳 Node，但是当不存在表达式所描述的节点时使用 default 算法的其它调度策略重新选择宿主机，这种方式适合那些希望 Pod 尽快被调度运行的场景，是一种弱绑定。reshedule 表示目标节点不存在时将该 Pod 放入待调度 Pod 队列中等待下次被重新调度，直到找到匹配宿主机为止，是一种强绑定。

4.2.2 用户绑定调度策略的具体设计

根据上文的资源描述文件，设计如下绑定方式，流程图如图 4-2，具体执行步骤如下：

1.调度器检查待调度 Pod 的 multiNodeSelectors 属性值，如果值为空，或者未指定该字段值，则使用 Scheduler 默认的 default 调度算法执行调度。如果该属性值不为空，则执行步骤 2；

2.根据 multiNodeSelectors 对象中 expressions 字段确定调度目标节点集合，如果集合为空，根据 collectionIsEmpty 的设定值选择执行方式，如果该值为 reshedule，则该 Pod 进入待调度 Pod 队列的队尾，等待下一次调度，如果集合为空但设定值为 default 则使用默认调度算法进行调度。如果集合不为空，执行步骤 3；

3.执行调度的 priority 过程，为集合中各节点打分，得分最高的为调度目的地。若存在多个分数一样的节点则随机选择一个节点作为调度目标节点；

4.对于 nodeLabelChange 设置为 reschedule 的 Pod，要周期性比对 Node 的 labels 标签是否改变，若 Node 的 lables 与 Pod 的指定的 multiNodeSelectors 属性值不再匹配，则将该 Pod 再放入待调度 Pod 队列，等待重新调度，若 nodeLabelChange 设置为 kill，则将比对结果不一样的 Pod 销毁，回收其所占资源。

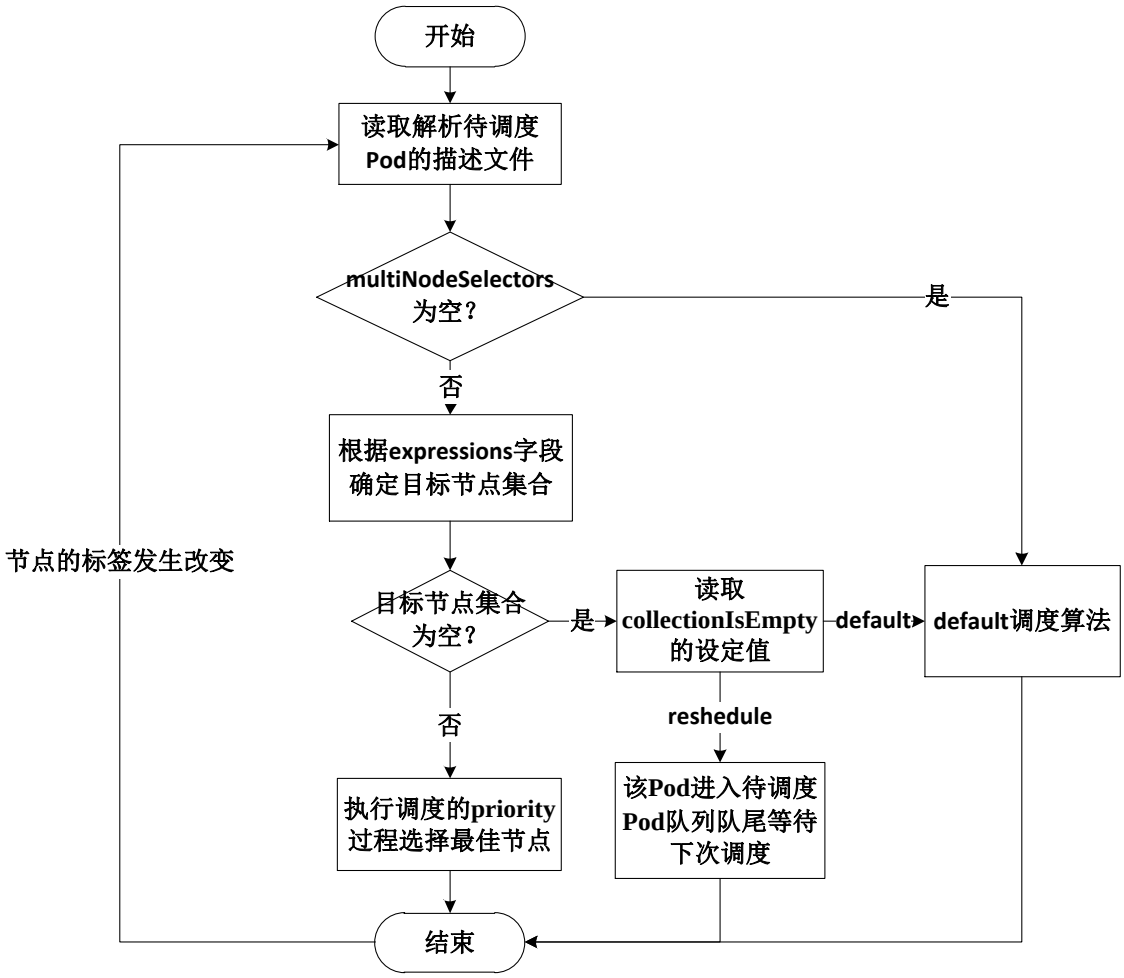

图 4-2 用户绑定调度策略流程图

4.2.3 存在的问题

在上述的用户绑定调度策略中第二步和第三步之间，如果集群中存在 lables 值与 Pod 指定的 multiNodeSelectors 属性值相匹配的节点，但此节点的资源状况并不足以让该 Pod 创建并运行，此时有三种调度方式，第一种最为简单，将该 Pod 放回待调度 Pod 队列等待下次重新调度；第二种是忽略 Pod 设置的标签属性，使用其它调度策略选择得分最高的节点作为调度目的地；第三种情况适用于那些希望指定的 Pod 要及时的运行在指定的节点上，这就需要释放该节点的一些资源，将优先级低的 Pod 调度到其它空闲节点上，为优先级高的 Pod 腾出资源。然而，Kubernetes 并不支持优先级抢占式调度，应该怎么将 Pod 调度到其它节点上，选择

哪些 Pod 进行迁移，哪些 Pod 的优先级高，哪些 Pod 的优先级低，这也是下文重点解决的问题。

4.3 抢占式调度策略的设计

在实际应用场景中，不仅是集群，甚至是一台服务器都承载着规格不一的数据业务和应用程序，例如：批作业处理、图形图像处理、实时计算、web 服务等，它们有些是高 CPU 消耗型，有些是高内存消耗型，还有些是高 IO 型，这些作业对实时性要求不一，对于向终端用户提供服务或应用层提供基础设施服务的作业就需要较高的实时性，例如 Web 服务、DNS 解析等，而对于一些批处理作业就对实时性没有太高要求。因此需要根据 Pod 中运行的作业类型判定各个 Pod 的优先级，对于高优先级的 Pod 可以抢占低优先级 Pod 的资源，为此本文设计了一种抢占式调度策略。

要想实现抢占式调度，首先需要对 Pod 的优先级进行划分，明确具体抢占规则，然后将较低优先级的 Pod 按资源使用量降序储存在优先级队列中，在资源回收时，逐一回收此队列上的资源直至宿主机的资源状况满足待调度 Pod 的资源请求量。

4.3.1 Pod 优先级的划分

本设计根据 Pod 的重启策略进行优先级的划分，Pod 的重启策略是指 Pod 因为种种原因退出后，重启 Pod 的方式。Kubernetes 共提供了三种重启方式，重启策略可以通过 Pod 资源描述文件的 spec.restartPolicy 属性来设置，该值为 Always 时表示无论 Pod 因为何种原因终止退出，都要将其重新启动，这是默认的重启策略；若该属性值为 OnFailure 则表示当 Pod 因为异常而终止运行时进行重启；Never 则不论 Pod 发生出现什么状况，都不会重新启动。

由上面重启策略可见，如果 Pod 的 spec.restartPolicy 属性被设置为 Never 的 Pod 只需要开始运行，并不在意运行时发生的种种状况，重启方式为 OnFailure 的 Pod 只有在非正常终止才会被重启，它只需要完整的度过 Pod 的生命周期，而对于重启策略被设置为 Always 的 Pod 就需要一直在后台运行。显然可以将 Pod 根据重启策略划分为三个优先级，重启策略为 Always 的 Pod 优先级最高，OnFailure 次之，Never 最低。

为此将抢占式调度的具体规则设定如下：

1.将宿主机的所有运行中 Pod 按重启策略分为三个优先级，Always 最高，OnFailure 次之，Never 最低。

2.在Pod资源配置文件相关标签允许的情况下高优先级的Pod可以抢占低优先级的 Pod 的系统资源，但是无论在什么时候低优先级 Pod 都不允许抢占高优先级 Pod 的系统资源。

3.不允许优先级相同的 Pod 相互抢占资源，这样可以避免同优先级的 Pod 循环互抢，降低系统的吞吐率。

4.3.2 优先级队列的实现

Kubernetes 中计算资源主要是指 CPU 和内存，创建 Pod 的时候可以指定每个容器的资源请求 resources.request，Pod 的资源请求就是 Pod 中所有容器的资源请求之和。设定该值可以保证 Pod 有足够的资源来运行，并且防止其它 Pod 无限制的使用资源。

资源大小的衡量是用配额来表示的，下面是一个 Pod 的资源描述文件里与资源请求相关的字段：

```
…
resources:
   memory:”256Mi”
   cpu:”500m”
…
```

其中 CPU 的单位是核（core），内存的单位是字节（Byte），该 Pod 请求的资源为 256MByte 内存和 0.5 核的 CPU。

在实际使用中大多数 Pod 实际的资源使用率是远远达不到在创建时请求的资源量，所以这里在衡量 CPU 的资源使用量时使用的是 CPU 实际的资源使用量 cpuUse，但是考虑到应用程序对内存的需求容易出现突然增长的情况，为了保证 Pod 不会因为内存不足而无法正常运行，这里对内存的衡量是内存的请求量 memoryRequest。

因为只有高优先的 Pod 可以抢占低优先级 Pod 的资源，所以在每台宿主机只需要为较低优先级的 Pod 维护两个队列即可。在 Kubernetes 中，主要的资源就是内存和 CPU，通过 cAdvisor 获取宿主机上所有的 Pod 的相关资源信息，将较低优先级的 Pod 建立两个队列：onFailue 队列和 Never 队列，onFailue 队列按照资源的使用量将重启策略为 OnFailure 的 Pod 降序存储，如图 4-3，具体规则是按内存的请求量排序，如果内存请求相同，则按照 CPU 使用量，同理建立 Never 队列。

本调度算法在回收 Pod 资源时是按 Pod 队列的顺序进行的，将 Pod 按资源使用量降序排列可以减少 Pod 的回收个数，这样系统不会因为频繁的进行 Pod 的回

收与重调度而浪费大量的工作时间，影响系统的性能。

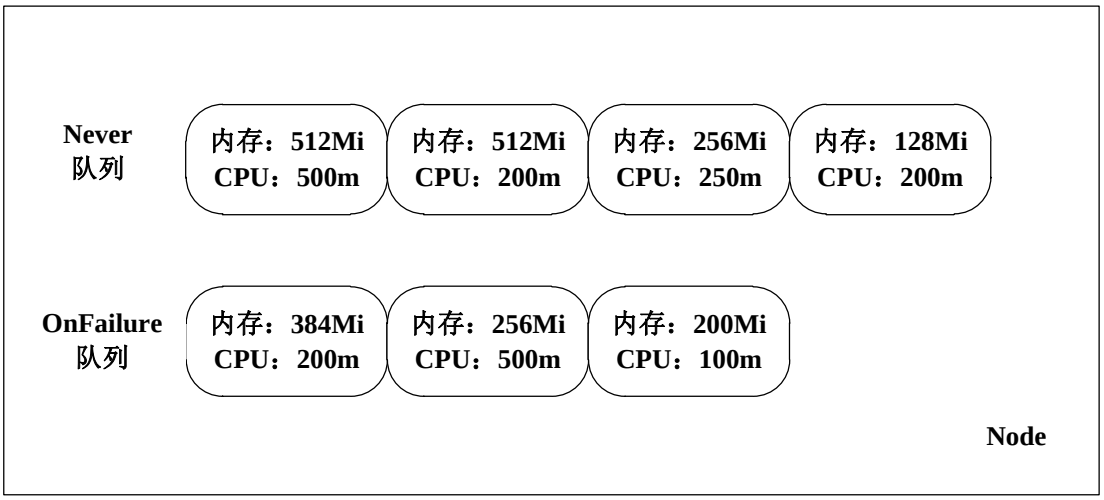

图 4-3 优先级队列示例

下面以 Never 队列为例，介绍优先级队列的基本结构和基本操作，优先级队列的节点数据结构表示如下：

```
type queue_pod struct{
    podId int           //Pod 唯一标识符
    podName string      //Pod 名称
    cpuUse float32          //CPU 使用量
    memoryRequest float32     //内存请求量
    restartPolicy string        //重启策略
    podPoint *next  //Pod 指针
    …
}
```

本文采用优先队列来实现 Never 队列，优先队列和普通队列不同，普通队列是先进先出的，而优先队列中每次的插入（入队）和删除（出队）操作都会动态调整队列中各元素的位置，每次出队操作都会去除队列中权值最高（或最低）的节点。

在 Never 队列中，常用的操作有三个：插入、查找和删除。共有三种方式实现优先队列：有序表、无序表和二叉堆。有序表插入的时间复杂度为 O(n)，删除的时间复杂度为 O(1)，这种方式适合插入操作多于删除操作的队列；无序表的插入和删除的时间复杂度与有序表相反，适合删除操作多于插入的队列。在 Never 队列中，插入和删除操作都很多，因此本设计采用第三种方式即二叉堆来实现 Never 队列，二叉堆实现的 Never 队列插入和删除的时间复杂度都是 $O(\log_2 n)$，这

样可以均衡算法的复杂度，三种操作的具体描述如下：

1.插入：根据各 Pod 的资源使用情况，向 Never 队列中插入节点，采用二叉堆的大顶推的方式存储 Pod，排序的第一关键字是内存，第二关键字是 CPU。

2.查找：使用二叉堆方式建立的 Never 队列，队首元素就是资源使用量最大的元素，因此查找操作在这里可以忽略。

3.删除：回收队首 Pod 的资源，将该 Pod 从队首删除，删除根节点后会生成两个子堆，这时候需要对子堆进行动态调整重新形成大顶堆。

4.3.3 抢占式调度策略的具体设计

根据 Pod 的重启策略建立两个优先级队列：Never 队列和 OnFailure 队列，重启策略为 Always 的 Pod 可以抢占 Never 队列和 OnFailure 队列的 Pod 资源，重启策略为 OnFailure 的 Pod 只能抢占 Never 队列的 Pod 资源，而重启策略为 Never 的 Pod 只能被抢占资源，被抢占资源后进入待调度 Pod 队列等待下一周期的调度。

抢占式调度具体执行步骤如下：

1.在宿主机上建立两个优先级队列，Never 队列按资源使用量降序存储重启策略为 Never 的 Pod，OnFailue 队列降序存储重启策略为 OnFailue 的 Pod；

2.解析待调度 Pod 的资源描述文件，获取其所需的 CPU 和内存资源以及重启策略，若重启策略为 Always 则执行步骤 3，若重启策略为 OnFailure 则执行步骤 4，否则不允许进行抢占式调度，使用其它调度方式进行调度；

3.依次回收 Never 队列的 Pod 所占用的资源，将回收的 Pod 放入待调度 Pod 队列，直到宿主机可使用资源量大于等于待调度 Pod 请求的资源值为止，将 Pod 调度运行，此时调度完成。若将 Never 队列的所有 Pod 都被回收，但是宿主机的资源状况仍未达到待调度 Pod 的资源要求，则继续按序回收 OnFailure 队列的 Pod 直至宿主机的资源情况达到此 Pod 的调度请求。若仍未达到，则不允许进行抢占式调度，使用其它调度策略；

4. 依次回收 Never 队列的 Pod 所占用的资源，将回收的 Pod 放入待调度 Pod 队列，直到宿主机可使用资源量大于等于调度 Pod 请求的资源值，此时调度完成。若将 Never 队列的所有 Pod 都被回收，但是宿主机仍未达到调度 Pod 的资源要求，使用其它方式调度；

图 4-4 显示了重启策略为 Always 的 Pod 抢占式调度的方式，重启策略为 OnFailure 的 Pod 方法类似。

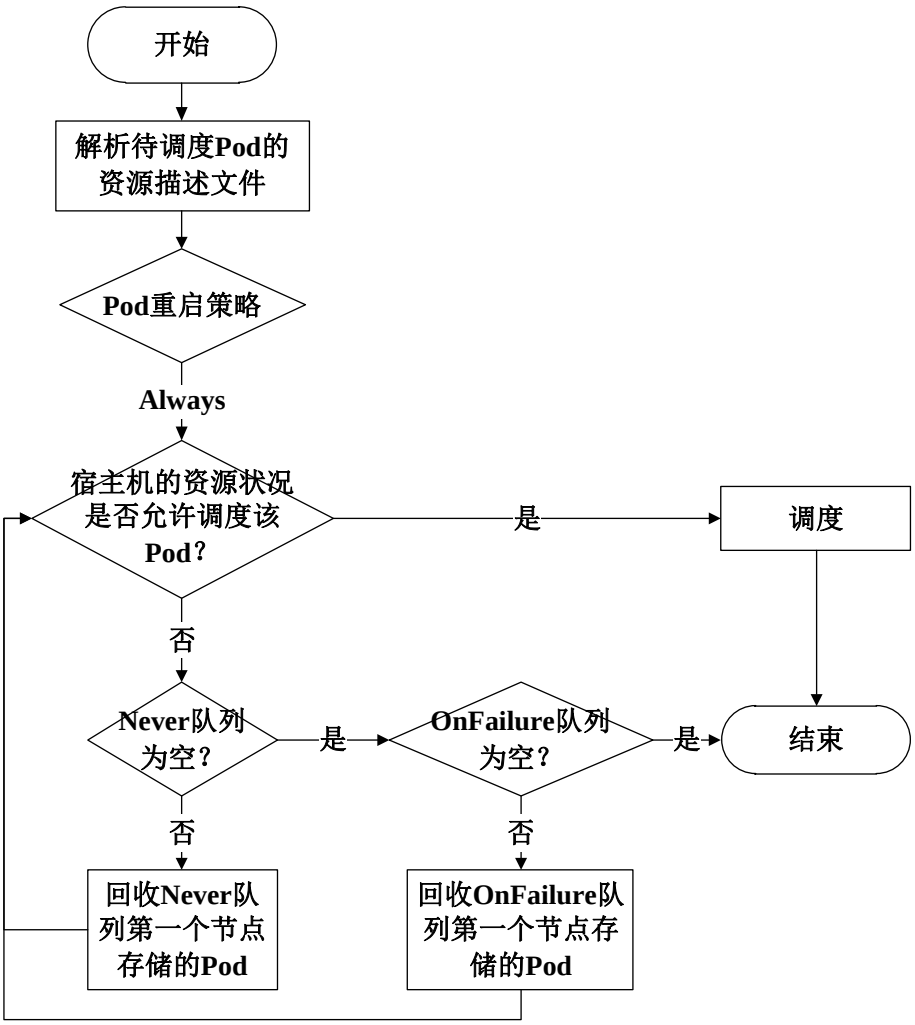

图 4-4 重启策略为 Always 的 Pod 抢占式调度流程图

4.3.4 抢占式的用户绑定调度策略综合设计

并不是所有的高优先级 Pod 都需要去抢占低优先级 Pod 的系统资源，在本设计中，在 Pod 进行用户绑定时，找到了目标节点，但目标节点的资源状况并不允许进行调度，这时可以根据 Pod 的资源描述文件的相关标签说明进行选择，可以选择进行抢占式调度，也可以按 default 算法的其它调度策略进行调度，为此本文综合用户绑定调度和抢占式调度综合设计了一种可抢占式的用户绑定调度策略 PreemptiveBinding。

这里需要对上述的 Pod 资源描述文件进行修改，在 multiNodeSelectors 中增加一个 resourceNotFit 属性，该属性有三个可选值，default 表示当资源不足时忽略 expressions 属性按默认调度策略重新选择宿主机，force 表示进行抢占式调度，reshedule 表示将待调度 Pod 放回待调度队列等待下次调度。

抢占式的用户绑定调度策略总流程图见图 4-5，

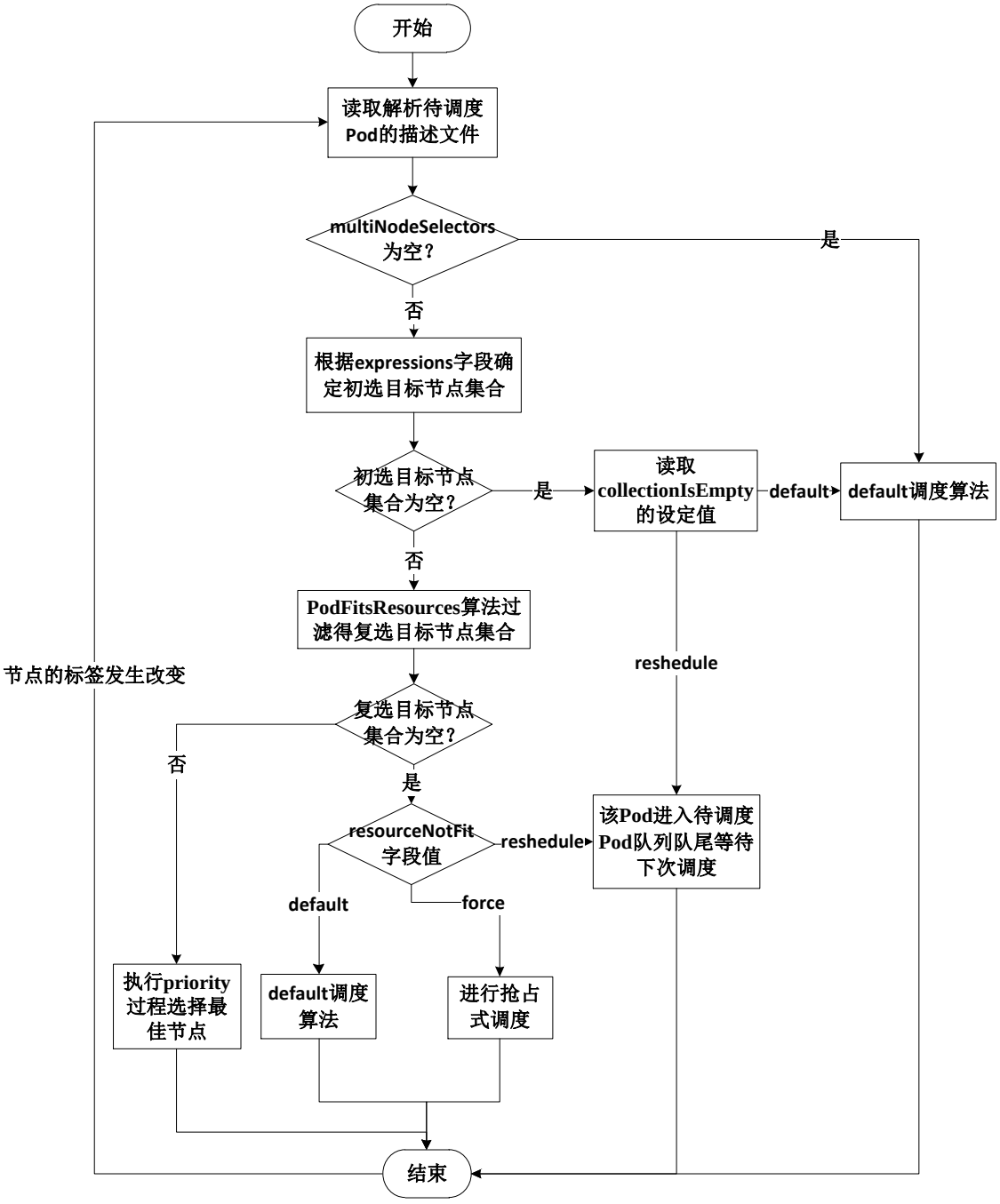

图 4-5 抢占式用户绑定调度算法流程图

具体执行步骤如下：

1.调度器检查待调度 Pod 的 multiNodeSelectors 属性值，如果该值为空，或者未指定该字段值，则使用 Scheduler 默认的 default 调度算法，如果该属性值不为空，则执行步骤 2；

2.根据 multiNodeSelectors 对象中 expressions 字段确定初选目标节点集合，如

果集合为空，则检查待调度 Pod 的 collectionIsEmpty 的设定值选择执行方式，如果该值为 reshedule，则该 Pod 进入待调度 Pod 队列的队尾，等待下一次调度，如果集合为空但设定值为 default 则使用默认调度算法进行调度，如果集合不为空，执行步骤 3；

3.根据调度器的 PodFitsResources 算法过滤初选目标节点集合，过滤结果放入复选目标节点集合，如果集合为空，则根据 Pod 的 resourceNotFit 属性值选择执行方式，如果该值为 default 则选择默认调度算法 default 重新选择宿主机，如果该值为 force，进行抢占式调度，如果该值为 reshedule 则将待调度 Pod 放回待调度队列等待下次调度；如果复选节点集合不为空，则执行步骤 4；

4.执行调度的 priority 过程，为复选目标节点集合中的各工作节点打分，得分最高的为调度目的地，若存在多个分数一样的节点则随机选择一个节点作为调度目标节点；

5.对于 nodeLabelChange 设置为 rescheduler 的 Pod，要周期性比对 Node 的 label 标签是否改变，若 Node 的 lable 与 Pod 的指定的 multiNodeSelectors 属性值不再匹配，则将该 Pod 放入待调度 Pod 队列，等待重新调度，若 nodeLabelChange 设置为 kill，则将比对结果不一样的 Pod 销毁，回收其所占资源。

4.4 本章小结

本章主要完成了第三章提出的用户绑定调度策略和抢占式调度策略。用户绑定策略在 PodSelectorsMatches 的基础上进行如下改进：1.扩充匹配表达式的语法规则，在原有的基础上增加了“或”和“非”运算；2.支持对匹配表达式的周期检测，如果后期运行中，Pod 指定匹配表达式不再匹配，可以根据设定值将 Pod 回收、重调度或仍在原宿主机上运行；3.若集群不存在 Pod 指定的宿主机，可将该 Pod 按其它方式进行调度。根据 Pod 的重启策略对 Pod 划分优先级，并实现优先级队列，设计了抢占式用户绑定策略，在宿主机资源不足时，高优先 Pod 可以抢占低优先级 Pod 的资源。最后综合用户绑定调度策略和抢占式调度策略设计了 PreemptiveBinding 调度策略。

第五章 基于 Kubernetes 云平台的动态负载均衡算法改进

在第三章已经提出，Kubernetes 中 Pod 在被调度到宿主机上启动开始，到其生命周期结束，都不会发生迁移，但是随着集群和宿主机环境的变化，有些节点负载变高，有的节点负载变低，这时候应该将负载较高的节点上的一些 Pod 迁移到负载较低的节点上，使集群的负载整体均衡，本章将结合 Kubernetes 云平台的特点对负载均衡进行研究与设计。

5.1 动态负载均衡研究

本节将对的动态负载均衡关键技术和相关算法展开深入研究，并结合 Kubernetes 云平台的特点，思考一种适合 Kubernetes 的动态负载均衡改进算法。

5.1.1 动态负载均衡关键技术

动态负载均衡是指周期性检测工作节点和集群的负载状况，根据收集的负载信息来决定调度策略，使集群中各工作节点的负载更加均衡。

在动态负载均衡算法中，主要关注四方面内容[31-34]：

1.负载分析，根据各节点的已分配资源和剩余资源状况分析集群的平均负载状况，再根据集群的负载均值来衡量各宿主机的负载程度，因此必须要有一个准确的负载衡量标准才能正确的对集群内各工作节点做出评价。

2.信息收集，算法中不仅需要指定一个合适的周期来收集负载信息，还需要考虑收集哪些资源信息，如何收集这些信息，以及怎么管理收集的负载信息，这都是信息收集关注的。

3.触发方式，即何时触发动态负载均衡调度，触发的方式可以采用外部事件驱动的方式，也可以采用阈值的方式。

4.调度策略，这是动态负载均衡最重要的部分，主要是考虑哪些节点负载过重需要进行重调度，将该节点的哪些任务迁移到负载较轻的节点上，以及如何进行迁移。

5.1.2 动态负载均衡算法

除了传统的动态负载均衡算法：最小链接法、加权最小链接法、启发式算法等，国内外有大量改进的动态负载均衡算法，如：负载最轻综合负载算法、综合利用率乘积法、预先设定加权系数、综合负载基准对比法、动态反馈综合负载均

衡法等[35-38]。

5.1.2.1 加权最小链接算法

最小链接算法是将用户作业分配到节点链接数最小的节点上，这种算法虽然实现简单，但是在面对各工作节点规格不一、性能各异的场景下算法效果并不理想。为了弥补这个缺陷，出现了加权最小链接算法，该算法根据各节点的规格情况设置 WEIGHT，工作节点的负载程度 p 用节点的连接率与 WEIGHT 的比值作为衡量标准，p 值越大负载程度越高，但是遗憾的是加权最小链接算法在面对资源需求不一的任务时仍然问题明显。

5.1.2.2 启发式调度算法

云计算的资源调度是一个 NP-hard 问题，需要从最短运行时间、最少迁移次数、最小成本等多方面考虑，而启发式算法的优点就是进行多目标优化和求解多解问题。常见的启发式算法有遗传算法、粒子群算法、蚁群算法、神经网络算法等，现在有很多研究人员使用遗传算法和粒子群算法来优化资源调度问题，但是启发式算法在面对云平台资源调度这种复杂问题时，计算量较大，响应时间较长，而且算法的实现过于复杂。

5.1.2.3 改进的算法

上述的资源调度算法都存在各自的局限性，现在有大量的动态负载均衡的改进算法。例如：预先设定加权系数，该算法通过实时采集节点负载信息对工作节点进行加权动态平衡调度，但是这种预先设置权值的方法无法准确反映多个因素造成的不均衡情况。改进的动态负载均衡算法在某一侧重面上确实可以提升系统的负载均衡程度，但是大多都存在实现复杂度过高，实现过于繁琐的问题。

5.1.3 综合分析

上述的算法大多在应对任务资源需求不一和节点性能差异的场景下很难实现负载均衡，有些算法可以实现较好的负载均衡效果，但是算法太过复杂，调度时间较长，实现不易。而且这些算法大多应用在基于虚拟机的云平台上，而容器云平台是建立在 IaaS 的基础上，结合容器云平台的特点设计的调度算法少之又少。

本设计是基于 Kubernetes 的云平台，资源调度的单位是 Pod，而不同类型的 Pod 对资源的需求是完全不一样的，因此基于容器云平台的资源调度算法应该考虑不同 Pod 的资源需求规格不同的问题。Pod 的宿主机既可以是物理机也可以是虚拟

机，因此还需要应对宿主机性能各异的情况。

Kubernetes 云平台是基于 Docker 容器的，而 Docker 是轻量级的虚拟化技术，启动速度很快，因此应该尽量减少调度算法的复杂度，避免因为调度算法的执行时间过长影响 Docker 的快速启动。

Kubernetes 默认的调度策略 default 算法是静态调度算法，除了异常情况的发生（如节点损坏），调度器是不会根据集群的负载状况进行动态调整。Kubernetes 的 default 算法在进行静态调度时只把 CPU 和内存作为衡量因素，但是在容器云平台中，一个 Pod 若要正常运行还需要挂载持久化存储系统和下载镜像文件，应该把网络传输速度作为衡量因素。

通过研究 Kubernetes 调度策略和动态反馈调度算法，本文融合 Kubernetes 云平台的资源调度特点改进了一种动态负载均衡算法，综合考虑 CPU、内存、与镜像仓库间的下载速度、与持久化存储系统间的数据传输速度四个因素来衡量宿主机负载状态，动态调整宿主机资源状况，提高集群的负载均衡程度，并尽量较少调度算法的时间复杂度。

5.2 基于动态负载均衡的优选策略设计

Kubernetes 现有的优选策略中无论是 LeastRequestedPriority 策略还是 BalancedResourcAllocation 策略都是静态调度的优选策略，并不适合衡量系统的负载均衡程度，而且没有考虑在实际应用场景，Pod 在调度运行后需要与持久化存储系统进行数据传输和从镜像存储系统下载镜像。本节综合考虑 CPU、内存、镜像网络下载速度、持久化存储网络传输速度四方面因素设计了一种更能准确衡量集群系统的资源状况的优选策略 ImprovedLoadBalancePriority。

5.2.1 节点资源的数学表示

为了量化评价各节点资源状况的优劣，根据评价标准进行资源调度，需要将节点的资源状况用数学表示式来表达。

本设计为了简化模型，研究对象为一个具有 n 台物理机或虚拟机的集群，由于容器云平台的底层既可以是物理机也可以是虚拟机，所以这里不加区分，统称为宿主机或工作节点，本设计主要考虑 CPU、内存、网络对调度策略的影响。

5.2.1.1 CPU 和内存的数学表示

集群内的 n 个工作节点表示为 $N=\{N_1,N_2,N_3,\cdots,N_n\}\quad i\in\{1,2,3,\cdots,i,\cdots,n\}$，各工作节点的资源规格表示为：

$$P=\{p(1),p(2),p(3),\cdots,p(n)\} \tag{5-1}$$

集群内标号为 i的工作节点资源规格表示为：

$$p(i)=\begin{bmatrix} c_i \\ m_i \end{bmatrix} \tag{5-2}$$

其中c_i、m_i分别表示第 i 个节点的 CPU 配额、内存配额。

假设各节点的规格系数向量为：

$$\Lambda=\{\lambda(1),\lambda(2),\lambda(3),\ldots,\lambda(n)\} \tag{5-3}$$

其中：

$$\lambda(i)=\begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix} \tag{5-4}$$

α_i，β_i分别表示编号为 i 的工作节点的 CPU 和内存的规格系数。

第 i 个节点的资源规格可表示为

$$p(i)=\begin{bmatrix} c_i \\ m_i \end{bmatrix}=\begin{bmatrix} CPU & 0 \\ 0 & MEM \end{bmatrix}\begin{bmatrix} \alpha_i \\ \beta_i \end{bmatrix}=Q\lambda(i) \tag{5-5}$$

其中$Q=\begin{bmatrix} CPU & 0 \\ 0 & MEM \end{bmatrix}$是一个对角矩阵，$CPU$、$MEM$是常量系数。衡量节点的资源优劣是通过规格系数$\lambda(i)$，性能越好的节点规格系数越高。例如对于一台双核 2G 内存的节点和一台 4 核 8G 内存的节点，可将二者的规格系数分别设置为：

$$p(1)=\begin{bmatrix} 1 \\ 1 \end{bmatrix},p(2)=\begin{bmatrix} 2 \\ 4 \end{bmatrix} \tag{5-6}$$

因此集群内各节点的资源规格可以表示为

$$P=\{Q\lambda(1),Q\lambda(2),Q\lambda(3),\ldots,Q\lambda(n)\}=Q\Lambda \tag{5-7}$$

各节点的资源规格系数可以由系统管理员根据集群内各服务器的具体性能和物理配置情况进行设置。因为不同节点存在性能差异，通过这种设置规格系数的方式可以将不同规格的工作节点区别对待。

5.2.1.2 网络资源的数学表示

Kubernetes 调度器现有的资源调度策略是：将“待调度 Pod 需要的 CPU 和内存加上候选节点上所有正在运行的 Pod 请求的 CPU 和内存”与“候选节点的 CPU 和内存总量”的比值作为衡量其节点资源状况的标准，比值最小的是得分最高的

节点，也是调度的目的地。该调度策略只将 CPU 和内存视为资源调度的影响因素，但是在实际应用中，网络传输速度的影响同样很大。调度器在创建 Pod 以后，要想 Pod 能够正常运行还需要两个基本步骤：

1.工作节点需要根据 Pod 的资源描述文件指定的地址去下载 Pod 中所有容器所需的镜像文件，而镜像下载速度直接影响了 Pod 的正常运行和业务的启动快慢。

2.由于 Pod 的数据是临时的，Pod 在被销毁时，其存储的数据也会消失，因此在 Pod 成功运行之后，需要为 Pod 中所有容器挂载持久化存储系统来进行数据的持久化存储，持久化存储与宿主机之间的网络传输速度，直接影响到 Pod 中运行的容器化应用的读写速度。

因此在本设计中将综合考虑 CPU、内存、镜像网络下载速度、持久化存储网络传输速度四方面因素来决定 Pod 的调度目的地。

因为在实际应用中，网络传输速度一般都低于网络带宽，不同于一般动态负载反馈算法，本设计考虑网络平均传输速度，具体如下：

1.各工作节点与镜像存储系统之间的镜像平均传输速度 $imageNetRate_i$；

2.各工作节点与持久化存储系统之间的数据平均传输速度 $dataNetRate_i$。

通过日志系统分别采集集群中所有可用节点与存储系统的平均传输速度，并将结果以时间戳和数据组合的方式记录在系统日志中。

5.2.2 ImprovedLoadBalancePriority 策略的设计

ImprovedLoadBalancePriority 策略通过系统中各资源的负载均值与节点的负载比值作为该节点的各资源得分，并据此计算综合得分，根据综合得分评判节点的资源状况。

5.2.2.1 负载均值

一个工作节点的 CPU 的使用率 $cpuUse_i$ 是指该节点的 CPU 在一定时间内（通常为一个采集周期）的平均利用率；一个集群 CPU 的负载均值 $averageCpuUse$ 是指在一段时间内整个集群 CPU 平均使用率的加权平均值。

集群 CPU 负载均值 $averageCpuUse$ 由节点的 CPU 规格系数加权平均表示为：

$$averageCpuUse = \frac{\sum_{1}^{n} cpuUse_i \times \alpha_i}{\sum_{1}^{n} \alpha_i} \tag{5-8}$$

其中 α_i 是第 i 个节点的 CPU 规格系数，$i \in \{1,2,3,\cdots,i,\cdots,n\}$。

一个工作节点的内存的使用率 $memoryUse_i$ 是指该节点的内存在一定时间内的平均利用率；一个集群内存的负载均值 *averageMemoryUse* 是指在一段时间内整个集群内存平均使用率的加权平均值。

集群内存负载均值 *averageMemoryUs*i由指该节点的内存规格系数加权平均表示为：

$$averageMemoryUse = \frac{\sum_{1}^{n} memoryUse_i \times \beta_i}{\sum_{1}^{n} \beta_i} \tag{5-9}$$

其中β_i是第 i 个节点的内存规格系数，$i \in \{1,2,\cdots,n\}$。

一个集群镜像网络的传输均值 *averageImageNetRate*是指在一段时间内（通常为一个采集周期）各节点与镜像存储系统的平均传输速度 $imageNetRate_i$ 的算术平均值， *averageImageNetRate* 可表示为：

$$averageImageNetRate = \frac{\sum_{1}^{n} imageNetRate_i}{n} \tag{5-10}$$

其中$imageNetRate_i$是第 i 个节点与镜像存储系统的平均传输速度，$i \in \{1,2,\cdots,n\}$。

一个集群数据网络的传输均值 *averageDataNetRate* 是指在一段时间内（通常为一个采集周期）各节点与数据存储系统的平均传输速度 $dataNetRate_i$ 的算术平均值，*averageDataNetRate* 可表示为：

$$averageDataNetRate = \frac{\sum_{1}^{n} dataNetRate_i}{n} \tag{5-11}$$

其中$dataNetRate_i$是第 i 个节点与数据存储系统的平均传输速度，$i \in \{1,2,\cdots,n\}$。

相比于使用算术平均的方式，本设计使用加权平均的方式计算 CPU 和内存的负载均值，考虑了不同节点的性能差别。而网络负载均值考虑到了两个方面，并且在计算网络负载均值使用的是平均传输速度的算术平均值，而不是网络带宽，更看重的是实际场景中的传输速度。

5.2.2.2 节点各资源得分

Kubernetes 调度器 default 算法在计算节点资源得分时采用的方式是待调度 Pod

申请的资源与宿主机实际资源状态的对比作为评价标准，但是在实际环境中，大多数 Pod 实际的资源使用情况是远低于其在创建之时所申请的资源，这就会导致集群资源的浪费。之所以会发生这种情况，是因为用户在创建 Pod 之时无法准确的估计该应用需要多少资源，用户为了自己的应用有足够的资源一般都会为 Pod 尽可能多申请一点资源。

为了更准确的评价节点的资源状况，本设计将节点实际的资源情况和集群的负载均值作为各资源得分的依据。

第 i 个节点的 CPU 得分 $cpuScore_i$ 等于这台宿主机的 CPU 利用率与集群的 CPU 负载均值之比，表示为

$$cpuScore_i = \frac{averageCpuUse}{cpuUse_i} \tag{5-12}$$

第 i 个节点的内存得分 $memoryScore_i$ 等于这台宿主机的内存利用率与集群的内存负载均值之比，表示为

$$memoryScore_i = \frac{averageMemoryUse}{memoryUse_i} \tag{5-13}$$

第 i 个节点的镜像传输速度得分 $imageNetScore_i$ 等于这台宿主机的镜像平均传输速度与集群的镜像网络负载均值之比，表示为

$$imageNetScore_i = \frac{imageNetRate_i}{averageImageNetRate} \tag{5-14}$$

第 i 个节点的数据传输速度得分 $dataNetScore_i$ 等于这台宿主机的数据平均传输速度与集群的数据网络负载均值之比，表示为

$$dataNetScore_i = \frac{dataNetRate_i}{averageDataNetRate} \tag{5-15}$$

各资源的得分取值区间是 $[0,+\infty]$，用 $cpuScore_i$ 来举例说明资源得分的具体含义，其它资源的得分含义类似，$cpuScore_i$ 的分值大于 1 表示该节点的 CPU 负载情况优于集群的 CPU 平均负载程度，小于 1 则情况反之。资源得分越高，资源情况越好，通过这种方式可以判断节点相对于集群整体的相对负载程度。

5.2.2.3 节点综合得分

在 Kubernetes 中，default 算法的优选策略将各资源得分的算术平均值作为节点的综合得分。但是在集群环境中，不同类型的资源使用情况对宿主机的性能影响不同，不同种类的应用对资源的需求也不同，有些是高 CPU 消耗型，有些是高内

存消耗型，甚至还有些是不需要与镜像存储系统和持久化数据存储系统进行数据交互的[31]。因此这里引入了权值因子来表示节点的总得分受各资源的影响程度，某一资源对 Pod 运行影响越大则该资源的权值因子越大。例如：对于 CPU 敏感型应用，在计算节点的综合得分的时候，应适度调高 CPU 的权值因子。权值因子向量表示为：

$$\Delta = (\lambda_{cpu}, \lambda_{memory}, \lambda_{imageNet}, \lambda_{dataNet}) \tag{5-16}$$

其中

$$\lambda_{cpu} + \lambda_{memory} + \lambda_{imageNet} + \lambda_{dataNet} = 1 \tag{5-17}$$

该向量的各元素表示相对应的资源对节点综合得分的贡献度。

节点的综合得分 $score_i$ 计算公式如下：

$$\begin{aligned} score_i = 10 \times [&\lambda_{cpu}(\ln cpuScore_i) + \lambda_{memory}(\ln memoryScore_i) \\ &+ \lambda_{imageNet}(\ln imageNetScore_i) + \lambda_{dataNet}(\ln dataNetScore_i)] \end{aligned} \tag{5-18}$$

对于 CPU 和内存来说，公式中对应的项的值大于 0 表示该类资源的负载小于集群的负载均值，小于 0 表示负载程度劣于集群平均状况；对于镜像网络传输速度和数据网络传输速度来说，大于 0 表示网络传输的平均速度快于集群的平均网络传输速度，反之则相反。

当 $score_i$ 大于 0 时表示该节点的综合负载情况优于集群的综合负载平均状况，可以将该节点作为任务迁入对象；当 $score_i$ 小于 0 时表示该节点的综合负载相对于集群的综合负载均值状况较差，可以将该节点作为任务迁出对象，选择一部分 Pod 进行重新调度。

5.3 负载队列的实现

本文实现动态负载均衡的思路是：在调度器中建立两个负载队列：高负载队列和低负载队列，将集群的所有节点按 ImprovedLoadBalancePriority 计算的综合得分分为两个队列，得分大于 0 的表示综合负载相对于集群较空闲，存储在低负载队列，小于 0 表示综合负载相对于集群较重，存储在高负载队列。通过周期检测集群的负载情况，将负载较高的节点的一些 Pod 迁移到负载较低的节点上，以保证集群的负载均衡。

负载队列的节点数据结构表示如下：

```
type queue_node struct{
    nodeId int              //工作节点节点唯一标识符
```

```
    nodeName string        //节点名称
    score float32          //节点得分
    server *next           //服务器指针
    …
}
```

在负载队列中，首先需要依次计算各节点的得分，向队列中插入节点，然后查找得分最高（或最低）的节点，最后删除该节点。因此我们采用优先队列来实现负载队列，优先队列和普通队列不同，普通队列是先进先出的，而优先队列中每次的插入和删除操作都会动态调整队列中各元素的位置，每次删除操作都会去除队列中权值最高（或最低）的节点。

对应在负载队列的三个基本操作，在优先队列中需要进行插入、删除（包括查找）。共有三种方式实现优先队列：有序表、无序表和二叉堆。有序表插入的时间复杂度为 O(n)，删除的时间复杂度为 O(1)，这种方式适合插入操作多于删除操作的队列；无序表的插入和删除的时间复杂度与有序表相反，适合删除操作主导的队列。在负载队列中，插入和删除操作都较频繁，因此本设计采用第三种方式即二叉堆来实现优先队列，二叉堆实现的优先队列插入和删除的时间复杂度都是 $O(\log_2 n)$，二叉堆实现的负载队列可以均衡算法的复杂度。

高负载队列和低负载队列都需要进行插入、查找、删除操作，高负载队列是将综合得分小于 0 的节点升序存储在队列中，而低负载队列是将综合得分大于 0 的节点降序存储，建立的负载队列示例如图 5-1。

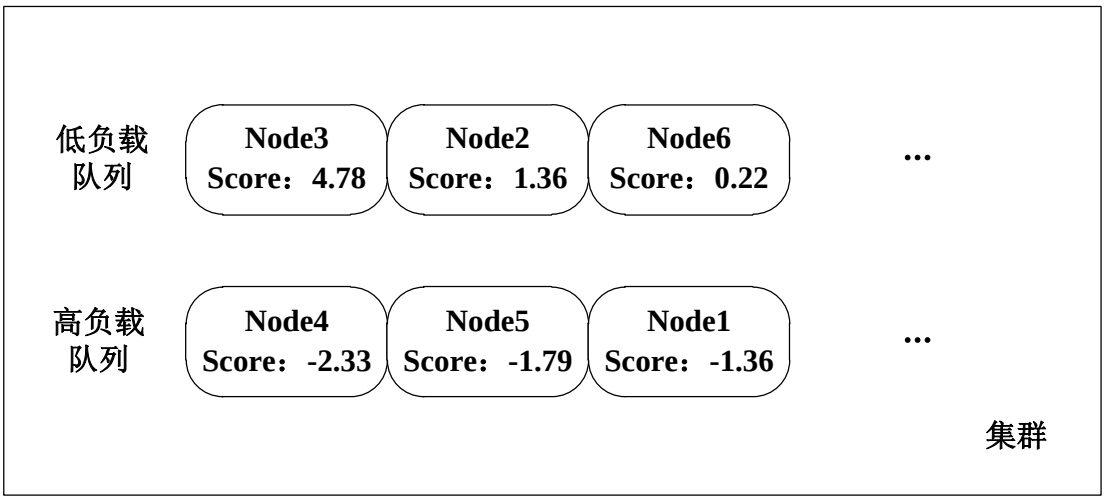

图 5-1 负载队列示例

这两个队列的基本操作类似，下面以高负载队列为例介绍该队列的基本操作，

高负载队列是根据各节点的最终得分升序建立的最小堆。具体步骤如下：

1.根据节点和集群的负载信息计算该节点的最终得分；

2.若分值小于 0，则将存储该节点相关信息的元素插入到二叉堆的末尾；

3.比较插入节点与父节点的得分大小，若父节点得分小于子节点的值，则交互二者的位置；

4.对于上述队列继续比较插入节点与父节点的分值情况，如果子节点分值更小则交换两者的位置，重复这一步操作直至没有父节点的分值小于该子节点，该节点插入完毕。

5.插入其它分值小于 0 的节点，重复上述操作，直至将所有节点遍历完毕，高负载队列构建完成。

用这种最小堆的方式建立的二叉堆，堆顶的节点就是负载最重的节点，获取该节点后，将此节点的部分 Pod 迁移到其它节点上。迁移之后该节点的得分已经发生变化，将该节点从高负载队列移除，在下一周期重新计算分值并插入相关队列。此时删除的节点是根节点，删除后会产生子堆，这时候就需要将子堆整合成新的二叉堆。

找出分值最小的节点容易，难的是删除了分值最小的节点后破坏了二叉堆的结构性，这就需要对二叉堆进行动态调整，具体步骤如下：

1.移除堆顶节点后，将末尾节点移到堆顶上；

2.比较该节点与其子节点的分值，若该节点的分值大于其子节点则交换二者的位置，对此节点重复这一步骤，直至该节点是叶子节点或者其分值小于它所有的子节点的分值，反之则调整直接结束。

5.4 动态控制

Kubernetes 目前版本中，触发重调度的主要原因有：

1.Pod 运行异常，Pod 在宿主机因为种种原因（例如与 Node 中其它 Pod 不兼容）被 Kublet 终止运行，回收其所占资源。

2.Node 运行异常，如果 Pod 的存储卷挂载在 GlusterFS 等分布式文件系统中，当 Node 因为停电或者损坏而停止工作，Master 节点会提取出存储在网络文件系统和 etcd 中的 Pod 相关信息，在其它节点上恢复这些 Pod[39-41]。

但是在动态负载均衡策略中，这显然是不够的，如图 5-2，本设计在 Kubernetes 原有方式上增加两种触发动态调度的方式：

1.外部事件触发，外部事件除了上面描述的 Pod 或 Node 运行异常，还可以是集群扩容缩容等原因，根据集群可用节点数量的增减进行动态调度。

2.节点超载保护，通过定时器周期性检测各节点的负载情况，当集群中有节点负载过重时，触发重调度，将负载过重的节点部分 Pod 迁移出。

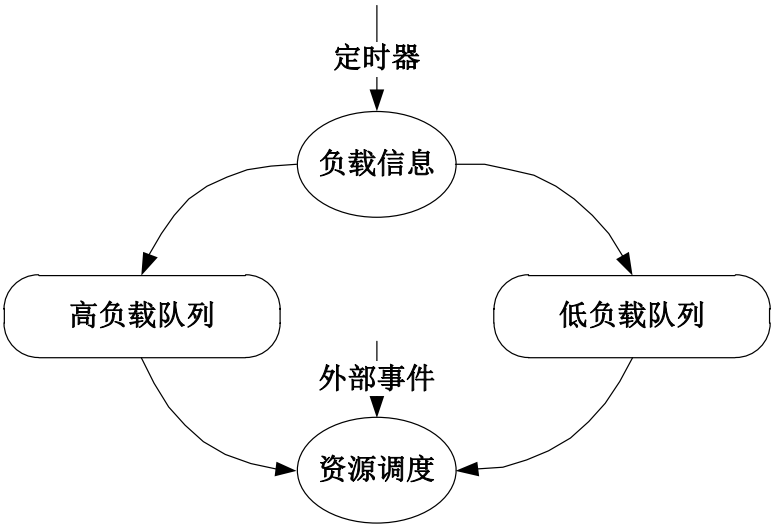

图 5-2 动态调度触发条件

为了让集群的负载更加均衡，需要将负载较重的节点一部分 Pod 迁移到负载较轻的节点上，尽量需要使集群内所有节点的资源利用率保持在一定的阈值范围 $[min, max]$，低于 min 的节点表示资源利用率较低，可以优先在该节点调度任务，高于 max 表示节点资源利用率过高，负载程度过重，需要进行动态调度，将一些任务迁移到其它节点上，实践证明较好的阈值范围是 $min \in [10\%, 20\%]$，$max \in [80\%, 90\%]$[34]。

Kublet 周期检测集群内各 Node 和 Pod 的运行情况，并将相关信息传入控制节点，控制节点根据负载情况计算各工作节点的综合得分，根据综合得分建立两个队列：高负载队列和低负载队列，并将高负载队列中综合得分低于下阈值的工作节点的一些 Pod 迁移到低负载队列中负载较低的工作节点上。

如图 5-3，动态控制的具体执行步骤如下：

1.系统初始化，设置系统资源规格系数、需求权值因子、动态调度阈值、负载信息收集周期等参数；

2.监控器按设置的周期定时获取各工作节点的负载信息，周期一般设置为 8-60 秒，将收集到的负载信息存入数据库；

3.控制节点读取数据库的负载信息，计算负载均值，各资源得分，各工作节点的综合得分；

4.控制节点的调度器根据综合得分，采用二叉堆的方式建立高负载队列和低负载队列；

5.由调度器完成资源调度，将综合得分低于下阈值的节点上部分 Pod 迁移到负载较低的节点上。

6.动态调度除了上述的定时器触发，还可以通过外部事件触发，如：Pod 或 Node 异常，集群扩容缩容、外部控制命令等。

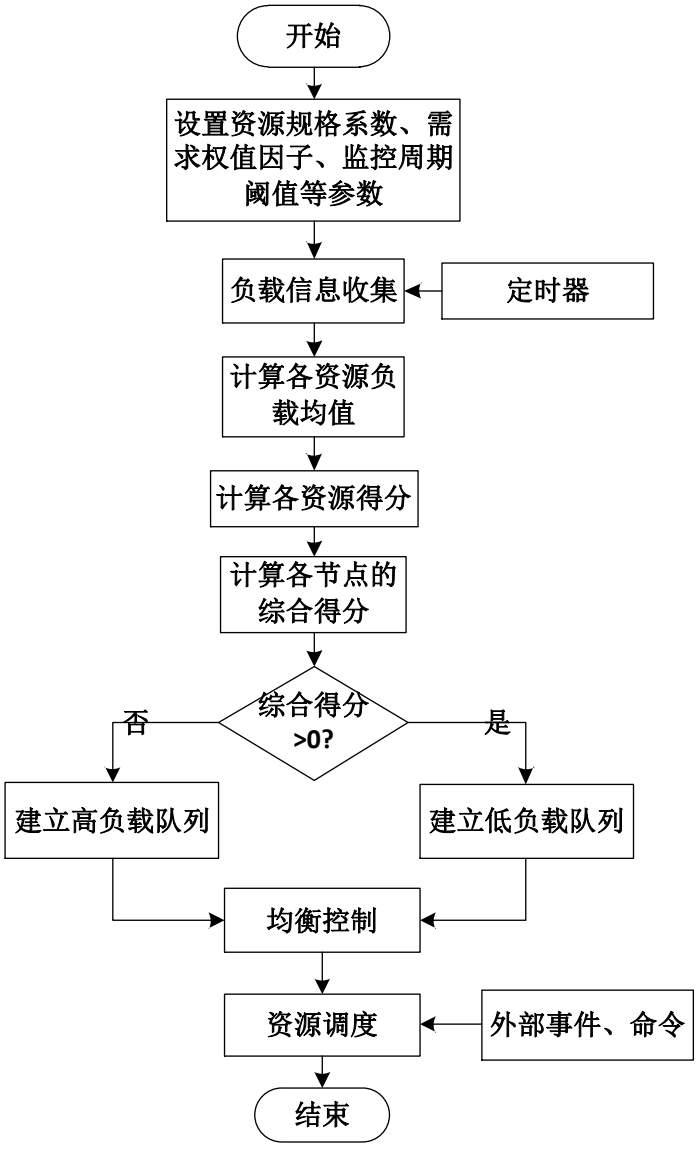

图 5-3 动态控制流程

5.5 基于 Kubernetes 的动态负载均衡改进算法的综合设计

本节在第四章和第五章的基础上对 Kubernetes 的调度策略进行总体设计，对于待调度 Pod 队列中的 Pod 使用静态调度策略进行调度，对于已经调度运行的 Pod 则使用动态调度，通过周期检测集群的负载情况，将负载较高的节点的一些 Pod 迁移到负载较低的节点上，以保证集群的负载均衡。

5.5.1 算法总体设计

如图 5-4，Kubernetes 资源调度模块最核心的组件是 Master 节点上的 Scheduler 模块，该模块负责管理待调度 Pod 和所有可用节点，并根据 etcd 上存储的集群负载信息建立高负载队列和低负载队列，根据调度算法和调度策略为待调度 Pod 选择最佳的宿主机，将 Pod 和 Node 进行绑定，并将绑定信息存入 etcd。

Node 上的 Kubelet 组件用于管控节点上的 Pod，主要负责接收 etcd 的绑定信息，在宿主机上创建并启动 Pod，Kubelet 的 cAdvisor 模块还需要定期将宿主机的负载信息反馈到 etcd 上存储。

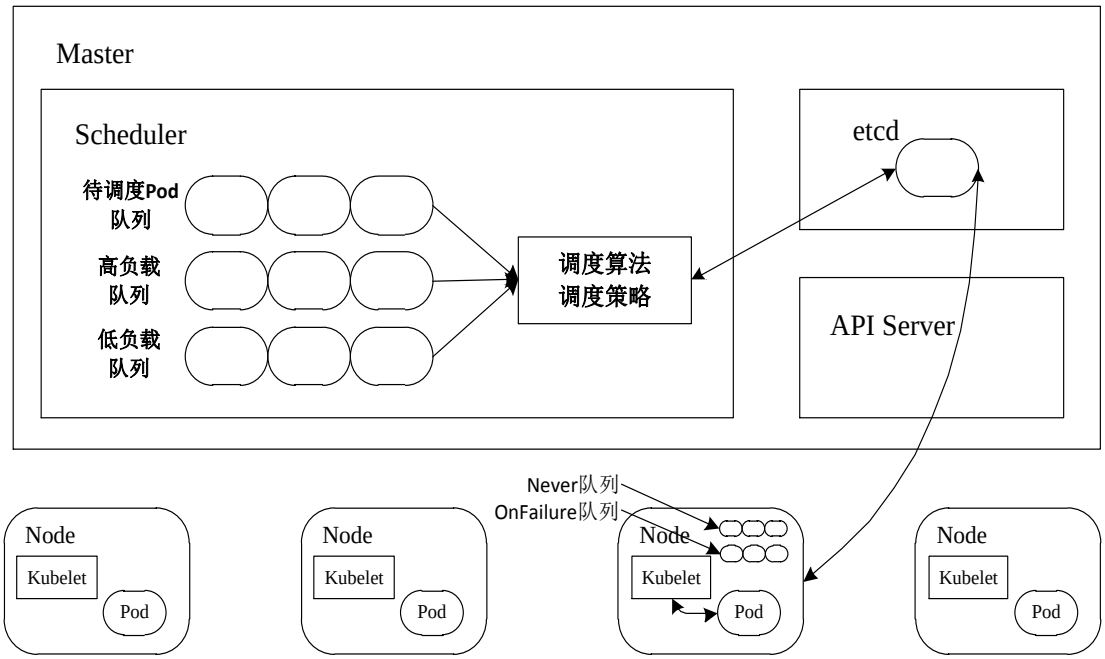

图 5-4 Kubernetes 资源调度总体设计图

根据上节的 ImprovedLoadBalancePriority 计算各节点的综合得分，得分大于 0 的按降序存入低负载队列，得分小于 0 的按升序存入高负载队列，负载队列的主要作用是：

1.静态调度时，优先为待调度 Pod 选择低负载队列中的节点，这样可以减少综合得分的计算次数，提高 Pod 的平均调度时间，具体见本节静态调度部分。

2.在进行动态调度时，将高负载队列的若干 Pod 迁移至低负载队列的节点上。具体动态调度方式见本节动态调度部分。

为每个工作节点上按 Pod 重启策略和资源量建立两个优先级队列：Never 队列和 OnFailure 队列，这两个队列主要用于：

1.抢占式调度时，重启策略为 Always 的 Pod 按序抢占 Never 队列和 OnFailure

队列的 Pod 资源，重启策略为 OnFailure 的 Pod 按序抢占 Never 队列的 Pod 资源。

2.在进行动态调度时将 Never 队列和 OnFailure 队列的 Pod 所占资源逐一回收，直至宿主机综合得分高于下阈值，分值高于下阈值的节点会触发动态调度。

本算法主要包括两部分：静态调度和动态调度，对于待调度 Pod 队列中的 Pod 使用静态调度策略进行调度，对于已经调度运行的 Pod 则周期检测集群的负载情况，将高负载的节点一些 Pod 迁移到低负载的节点上，进行动态调度以维持集群的整体负载均衡。

静态调度主要流程如下：

1.各工作节点上的 Kublet 组件收集宿主机资源负载信息，并将处理后的信息存入控制节点上的 etcd 系统中；

2.当待调度队列不为空时，调度器运用设计的静态调度算法为待调度 Pod 选择目标节点，并将绑定信息存入 etcd 中；

3.目标节点的 Kublet 从 etcd 读取绑定信息，在宿主机上创建并启动该 Pod。

动态调度的主要流程如下：

1.各工作节点上的 Kublet 组件收集宿主机资源负载信息，并将处理后的信息存入控制节点上的 etcd 系统中；

2.调度器周期性从 etcd 读取负载信息，计算节点综合得分，检查集群中是否存在负载过重的节点；

3.将负载过重的节点上的一些 Pod 资源回收，并将回收的 Pod 相关运行信息存入持久化存储系统；

4.调度器根据回收 Pod 的资源描述文件重新为其绑定目标节点；

5.目标节点从持久化存储系统中读取被回收的 Pod 运行信息，将该 Pod 调度运行。

5.5.2 算法注册

Kubernetes 调度器的调度方式是通过插件加载 AlgorithmProvider 来实现，AlgorithmProvider 是一个结构体，主要包括一组预选策略与一组优选策略。注册 AlgorithmProvider 则通过 RegisterAlgorithmProvider 实现：

func RegisterAlgorithmProvider(name string, predicateKeys, priorityKey util,StringSet)

该函数三个参数分别表示：参数名，预选策略集合，优选策略集合。

本设计中采用的预选策略见表 5-1，其中前三个是 Kubernetes default 算法自带的预选策略，PreemptiveBinding 是第四章设计的抢占式用户绑定调度策略。

表 5-1 预选策略表

预选策略名	详细说明
NoDiskConflict	判断待调度 Pod 与候选节点上已运行 Pod 是否存在磁盘冲突。
PodFitsResourses	判断候选节点是否有足够的资源调度运行该 Pod。
PodFitsPorts	判断待调度 Pod 与在候选节点上运行的 Pod 是否存在端口冲突。
PreemptiveBinding	改进的 PodSelectorMatches 算法，用于抢占式用户绑定，主要有四方面改进，1.支持多样的匹配规则；2.候选节点为空时，可根据 Pod 设定值决定调度方式；3.在宿主机标签改变时，根据设定值选择 Pod 的处理方式；4.在候选节点资源不足时支持抢占式调度。

采用的优选策略是上节设计的 ImprovedLoadBalancePriority 算法，可以更准确的衡量工作节点的“合适程度”和系统的负载均衡度，这里为采用上述预选和优选策略的组合算法命名为 KubernetesLoadBalance。

5.5.3 静态调度

静态调度主要用于按照先进先出的顺序将待调度 Pod 队列里的 Pod 按照设计的调度策略调度到合适的节点上，本文的静态调度策略主要依据是第四章设计的 PreemptiveBinding 算法和本章设计的 ImprovedLoadBalancePriority 算法。

default 算法的调度方式是：集群中所有可用节点经预选策略过滤后，计算所有候选节点的得分，得分最高的为调度目的地。与 default 算法不同，本文为了减少综合得分的计算次数，提高算法的性能，同时融合了抢占式的用户绑定策略，本节将结合负载队列设计静态调度，为了方便描述，将静态调度分为：非绑定静态调度和用户绑定静态调度。

5.5.3.1 非绑定静态调度

非绑定静态调度适用于未指定 multiNodeSelectors 的 Pod，算法流程图见图 5-5，具体步骤如下：

1.若待调度 Pod 未指定 multiNodeSelectors 属性或该属性为空，则执行步骤 2，否则使用用户绑定静态调度策略；

2.使用预选策略判断低负载队列队首元素是否适合调度，是则将该 Pod 调度到队首节点上运行，否则执行步骤 3；

3.使用三个预选策略判断低负载队列的下一个元素是否适合调度，若是进行调度，否则重复此步骤，若低负载队列没有满足条件的节点，则执行步骤 4；

4.使用 NoDiskConflict、PodFitsResourses、PodFitsPorts 策略过滤所有高负载队列和低负载队列之外的节点，若候选节点集合为空，则该 Pod 放入待调度 Pod 队列队尾，否则执行步骤 5；

5.计算候选节点结合所有节点的综合得分，得分最高的为调度目的地。

这种调度方式因为要时刻维护两个负载队列，会一定程度上增加控制节点的运算压力，但可以有效的提高 Pod 的平均调度时间。

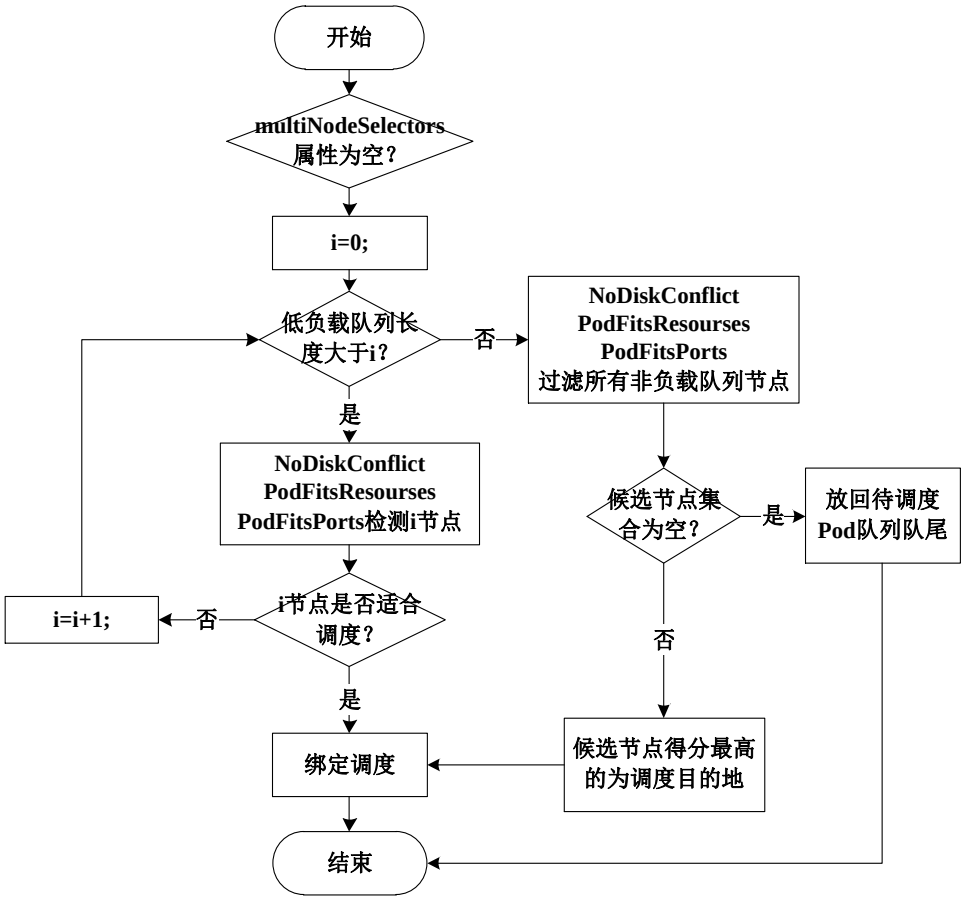

图 5-5 非绑定静态调度流程图

5.5.3.2 用户绑定静态调度

用户绑定静态调度的主要步骤如下：

1.判断待调度 Pod 队列是否为空，若不为空，执行步骤 2，否则不进行任何操作；

2.判断 multiNodeSelectors 字段是否为空，若为空使用非绑定静态调度策略进

行调度，若不为空执行步骤 3；

3.根据 multiNodeSelectors 对象中 expressions 字段确定初选目标节点集合，如果集合为空，则根据 Pod 的 collectionIsEmpty 的设定值选择执行方式，如果该值为 reshedule，则该 Pod 进入待调度 Pod 队列的队尾，等待下一次调度，如果集合为空但设定值为 default 则使用非绑定静态调度策略进行调度，如果集合不为空，执行步骤 4；

4.根据调度器的 PodFitsResources 算法过滤初选目标节点集合，过滤结果放入复选目标节点集合，如果复选集合为空，则根据 Pod 的 resourceNotFit 属性值选择执行方式，如果该值为 default 则使用非绑定静态调度策略重新选择宿主机，如果该值为 force，进行抢占式调度，如果该值为 reshedule 则将待调度 Pod 放回待调度队列等待下次调度；如果复选节点集合不为空，则执行步骤 5；

5.执行调度的 ImprovedLoadBalancePriority 过程，为集合中各节点打分，得分最高的为调度目的地。若存在多个分数一样的节点则随机选择一个节点作为调度目标节点；

6.对于 nodeLabelChange 设置为 rescheduler 的 Pod，要周期性比对 Node 的 label 标签是否改变，若 Node 的 lable 与 Pod 指定的 multiNodeSelectors 属性值不再匹配，则将该 Pod 再放入待调度 Pod 队列，等待重新调度，若 nodeLabelChange 设置为 kill，则将比对结果不一样的 Pod 销毁，回收其所占资源。

其中一次调度过程如图 5-6。

5.5.4 动态调度

动态调度通过本章第二节的 ImprovedLoadBalancePriority 策略判断各节点的负载程度，进行重新调度。Kubernetes 云平台在进行重新调度时，为了使集群的负载更加均衡，一边需要将负载过重的工作节点的一些任务进行迁移，一边又要将相对空闲的节点加入到集群的任务分配。由此需要考虑三个问题：哪些节点需要重新调度，重新调度的 Pod 应该被迁移到何处，工作节点的哪些 Pod 应该被重新调度。

Pod 选择问题可以根据上一章建立的优先级队列来解决，按序逐一回收 Never 队列的 Pod，直至释放后的节点综合得分不会再触发动态调度，若释放完 Never 队列的 Pod，宿主机的资源状况仍会触发动态迁移则按序释放 OnFailure 队列的 Pod，并将它们依次放入待调度 Pod 队列中，等待调度器将它们重新调度到更加合适的节点上。

在节点的选择这个问题上，使用 5.3 节的负载队列来实现，将高负载队列的各

节点一部分 Pod 迁移到低负载队列上。

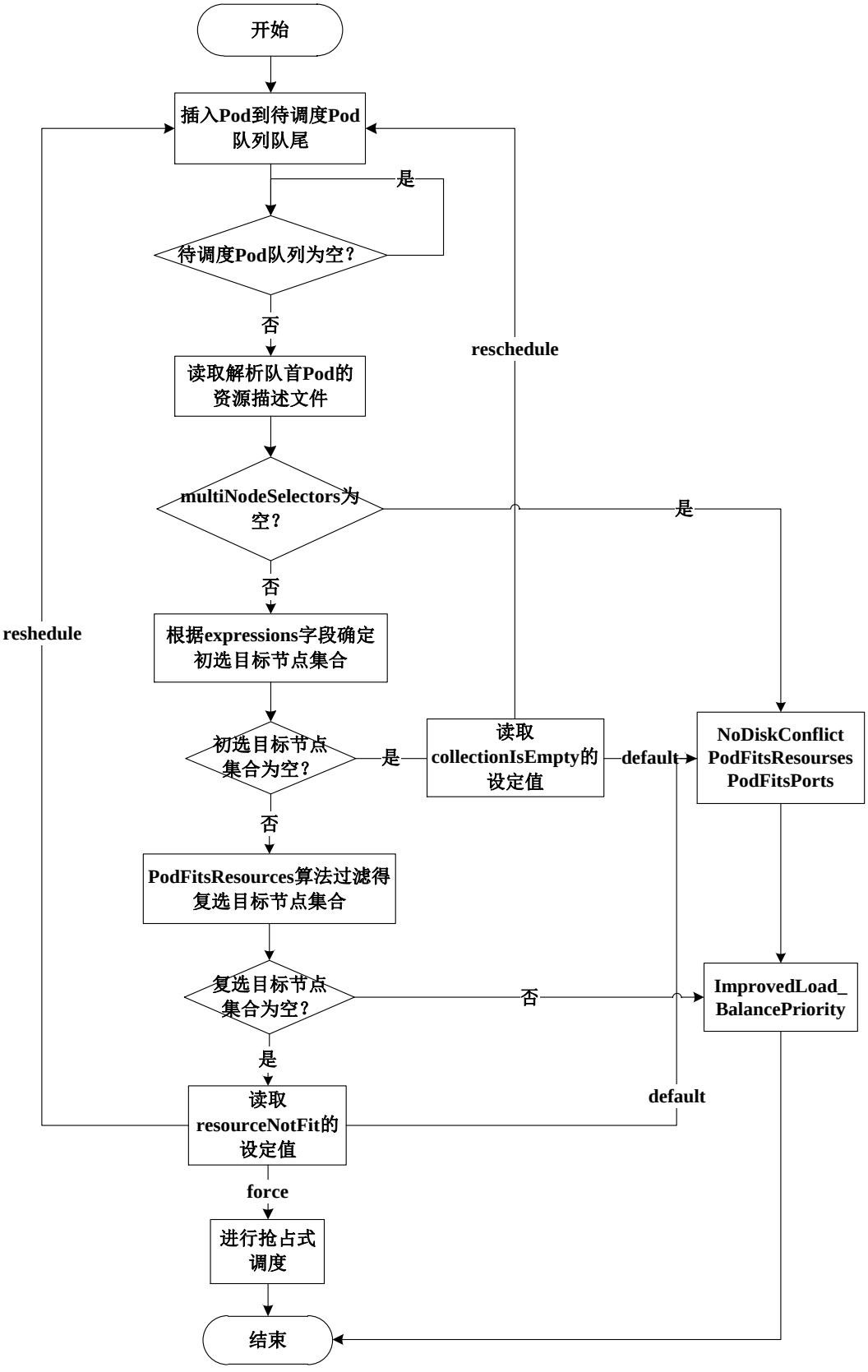

图 5-6 用户绑定静态调度流程图

动态调度的具体实现分为 4 个步骤，分选、预选、建队和调度四个过程，如图 5-7，调度器定时从 etcd 中读取所有可用节点的负载信息并计算综合得分，根据得分情况将节点分为两个队列，每个队列的节点再经过预选过程过滤掉一些不符合要求的节点，将过滤后的节点根据分值大小建立高负载队列和低负载队列，调度器从高负载队列的队首选择若干 Pod 迁移到低负载队列的节点上，这样通过动态调整的方式可以维持系统的总体负载均衡。

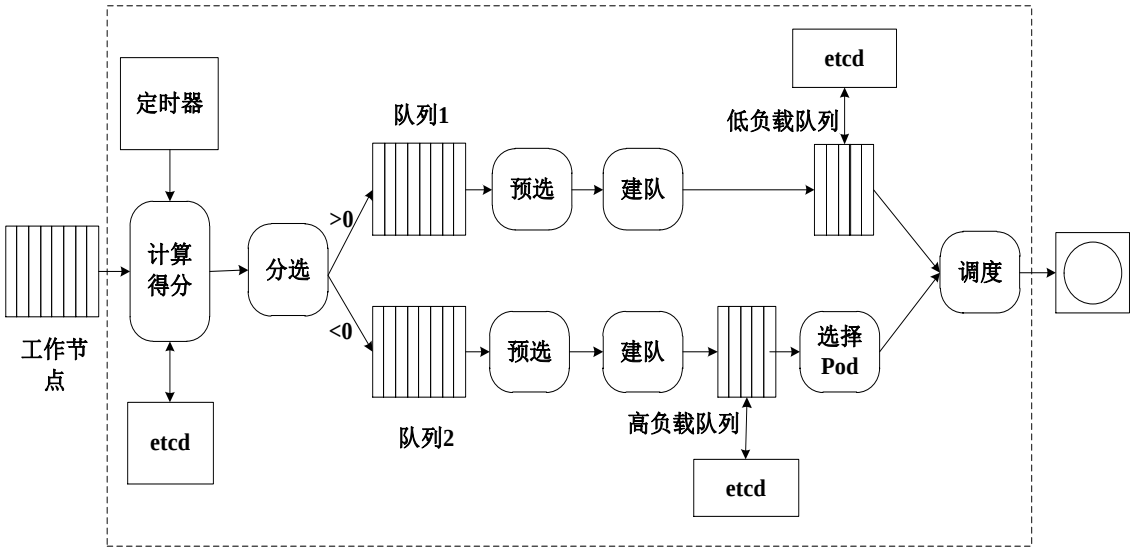

图 5-7 动态调度过程图

5.5.4.1 分选

图 5-8 是调度器进行一次分选过程的流程图，各工作节点的 Kublet 组件收集宿主机负载信息并存入 etcd，调度器从 etcd 读取各工作节点相关信息，根据 ImprovedLoadBallancePriority 算法计算各工作节点的综合得分，然后根据工作节点的得分情况分为两个队列，队列一存储综合得分大于 0 的节点，表示该节点负载状况优于集群的负载均值，队列二存储综合得分小于 0 的节点，表示该节点负载状况劣于集群的负载均值。

触发分选的条件有定时器或外部事件，其中定时器的周期此处设为 30 秒，周期太短会造成频繁调度使系统开销太大，太长则会导致系统的负载均衡衡量不准确。

5.5.4.2 预选

并不是得分大于 0 的节点资源使用状况就很好，节点的得分衡量因素有四个：CPU、内存、镜像下载速度和数据传输速度，节点可能会出现内存使用率很高，但是 CPU 和网络状况却很好，因此综合和得分依然大于 0 的情况，此时该节点不

应该被放入低负载队列。这里使用改进的 Kubernetes 调度器的优选策略 BalanceResoureAllocation 对节点的资源使用均衡程度进行评价，过滤掉资源均衡程度较差的节点。

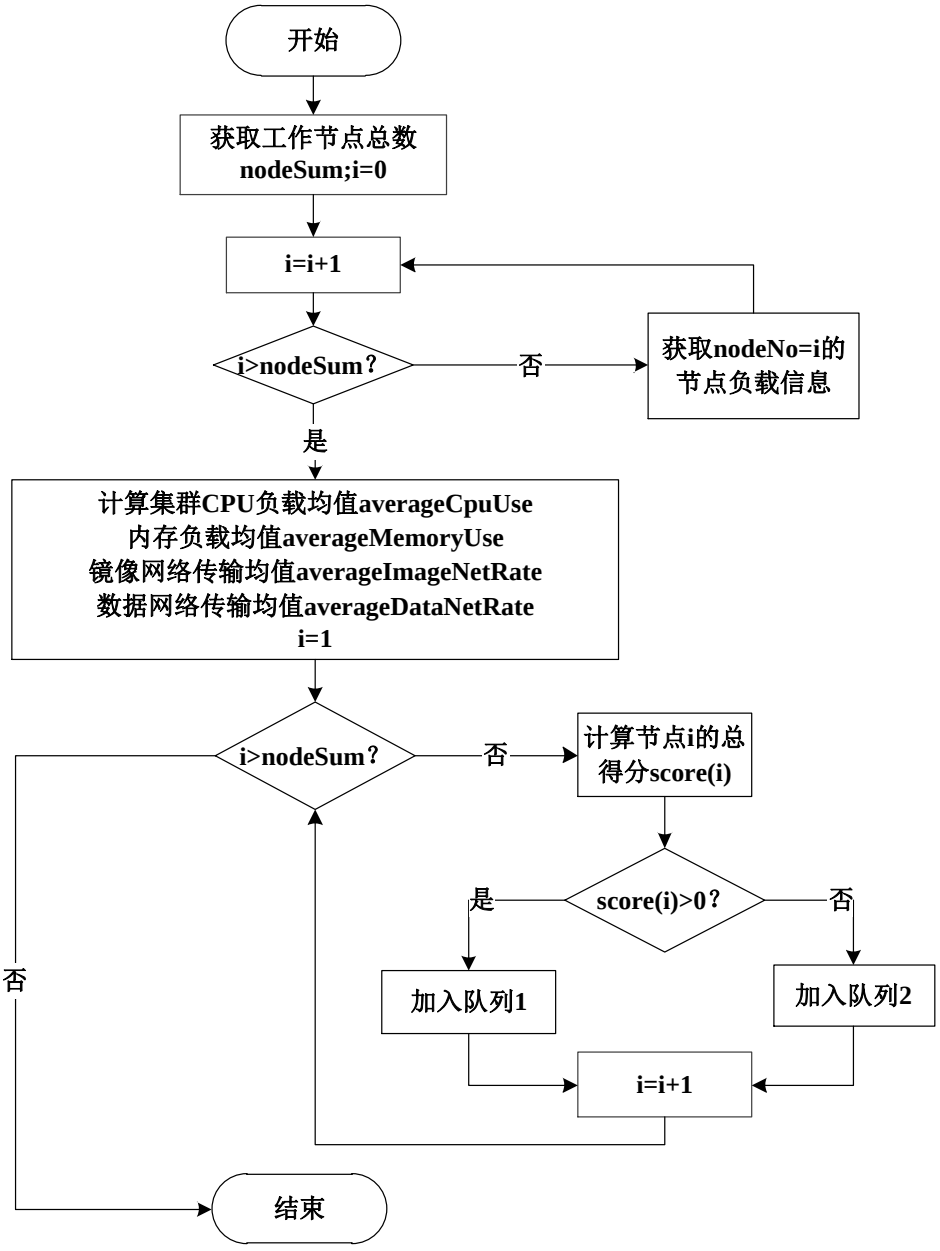

图 5-8 动态调度分选过程

改进的 BalanceResoureAllocation 算法得分计算公式如下：

$$balanceScore_i = 10 - |cpuUse_i - memoryUse_i| \times 10 \quad (5\text{-}19)$$

$balanceScore_i$ 表示编号为 i 的节点资源均衡度得分，$cpuUse_i$ 和 $memoryUse_i$ 分

别表示该节点的 CPU 使用率和内存使用率，由此公式可知，CPU 和内存的利用率越接近则得分越高，因为二者差值在 0-1 之间，所以 $balanceScore_i$ 得分区间为 0~10，分值越高则节点各资源均衡度越好。

队列 1 存储的是得分大于 0 的节点，表示该节点的资源状况优于集群的负载均值，但是一方面，对于一些得分略大于 0 的节点，并不适合作为调度目的地；另一方面，如果队列节点过多，那么后续建立低负载队列以及一些队列相关操作都会增加算法的执行时间。所以除了上述的过滤策略外，还需要设定一个上阈值，过滤掉得分低于上阈值的节点，留下得分更高的节点。

对于队列 2 的预选策略则比较简单，将负载程度不太高的节点过滤，即淘汰得分高于下阈值的节点。

5.5.4.3 建队

建立的队列主要有两种：优先级队列和负载队列，二者都是采用二叉堆的方式建立，因此队首节点就是权值最高（最低）的节点。

为每个节点上维护两个队列：Never 队列和 OnFailure 队列，用于动态调度和抢占式调度选择待回收的 Pod，优先级队列的建立方法在第四章已经详细描述了，这里不在赘述。

同时为集群中所有可用节点维持两个队列：高负载队列和低负载队列，高负载队列升序存储分值小于 0 的节点，越靠近队首，则负载状况越差；低负载队列降序存储综合得分大于 0 的节点，越靠近队首，负载状况越好。

负载队列的相关描述在上文也已提及，具体参见 5.3。

5.5.5.4 调度

这一步主要是将高负载队列的节点若干任务迁移到低负载队列上。

对于高负载队列上待迁移 Pod 的选择问题，本文采用的方式是：根据宿主机的资源使用率和集群的负载均值计算应当释放的资源量，先从 Never 队列开始逐一回收 Pod 的资源，直到释放的 Pod 资源总量大于等于应当释放的资源量。如 Never 队列内所有 Pod 均已释放，仍未达到应释放总量，则继续按上述方法释放 OnFailure 队列的 Pod 资源，具体流程详见图 5-9。

Never 队列的 Pod 重启策略为 Never，表示 Pod 只需要启动运行一次即可，不管因为什么原因被回收资源，都不会重启，表明该类 Pod 的重要程度最低，这里在回收该类 Pod 后，将 Pod 放入待调度 Pod 的队尾。

而 OnFailure 队列内的 Pod 重要程度要高于 Never 队列，Pod 只有在异常退出

才会重启，本设计在回收该类 Pod 的资源后，将其放入待调度 Pod 队列的队头，便于被及时调度到其它合适节点上运行。

对于重启策略为 Always 的 Pod，该类 Pod 优先级最高，不论因为何种原因退出都会被重启，所以本设计不允许将该类 Pod 进行动态迁移。

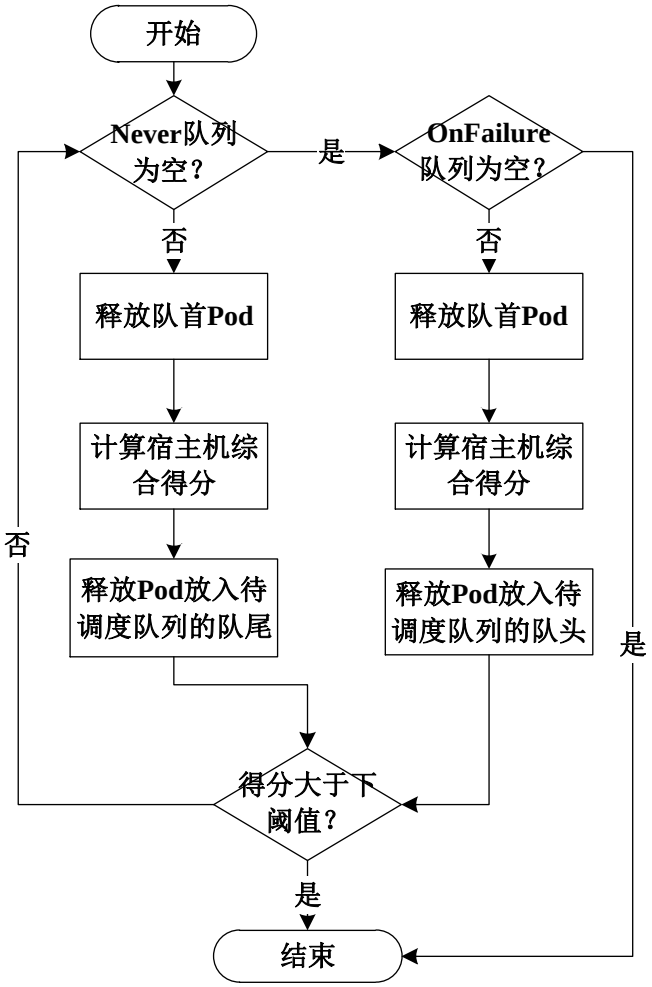

图 5-9 动态调度调度过程

5.6 本章小结

本章研究了 Kubernetes 中 default 算法的宿主机得分计算方式和动态负载均衡算法，设计了一种适合用作动态负载均衡调度的宿主机得分计算方法，该方法与 default 算法相比考虑了宿主机资源不一致、Pod 资源需求不一致、网络传输速度等因素。最后总体设计了资源调度策略，将资源调度划分为静态调度和动态调度，其中静态调度主要用于从待调度 Pod 队列读取 Pod 进行调度，而动态调度用于集群环境的动态负载均衡。

第六章 Kubernetes 云平台部署与实验结果分析

本章将对第四章、第五章设计的资源调度策略进行测试，为此需要搭建 Kubernetes 云平台并设计测试用例。下文将按照第三章设计的 Kubernetes 云平台搭建实验环境，由于用户绑定调度策略和抢占式调度策略较为简单仅进行功能点测试，而动态负载均衡调度则分别进行功能测试和性能测试。

6.1 Kubernetes 云平台部署

本设计采用 4 台安装 CentOS7.0 系统的阿里云服务器用于部署 Kubernetes realease 1.1，四台云服务器的相关信息如表 6-1。

表 6-1 云服务器资源配置

服务器名	CPU 核数	内存容量(GB)	网络带宽(Mbps)
master	2	4GB	2
node1	1	2GB	1
node2	2	4GB	2
node3	4	8GB	4

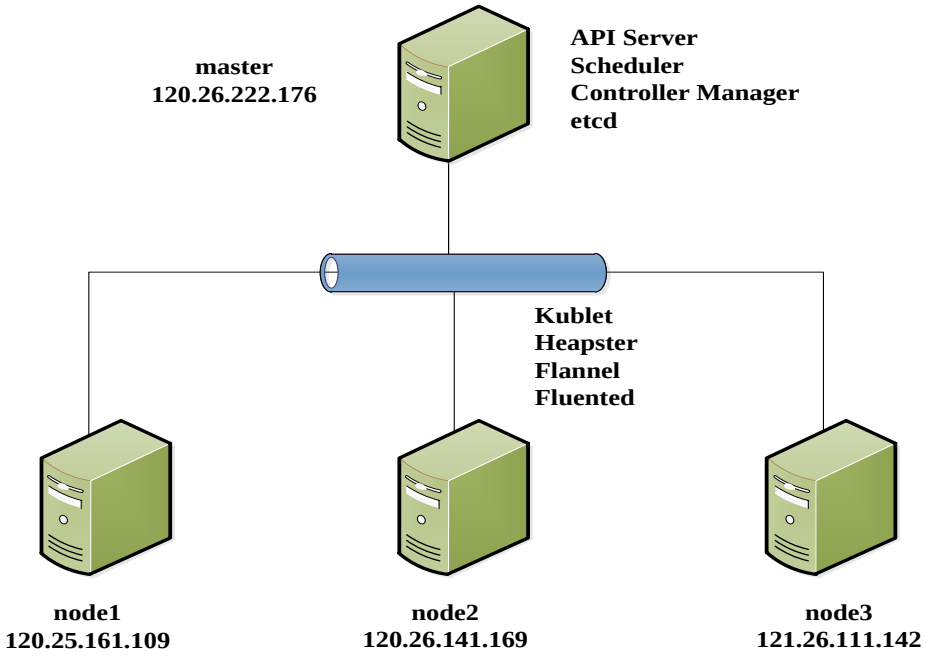

图 6-1 实验环境网络架构

四台云服务器的运行的主要组件如图 6-1，的主要软件及相关描述如表 6-2。

表 6-2 软件相关信息

软件名	主要作用
API Server1.1	为 Node、Pod、RC、Service 等对象提供增删改查的接口
Scheduler1.1	本设计的核心部分，负责 Pod 的资源调度
Controller Manager1.1	管理控制中心，负责集群内所有资源对象的管理
etcd2.2.5	负责集群关键信息的存储
Kubctl1.1	命令行工具，负责将用户的输入转换为对 API Server 的增删改查调用
Docker1.6	应用运行时环境
Kublet1.1	收集、统计 Node、Pod、Docker 的运行数据，主要包括实时和历史的 CPU、内存、磁盘、网络使用情况
ntp	使所有服务器时间保持一致
Heapster0.18.2 InfluxDb Grafana	将计 Kublet 收集的负载信息汇总，导入 InfluxDb 进行进一步处理，最终通过 Grafana 可以查看数据的图表展示
Flannel0.5.4	网络覆盖，联通各个节点
Fluented1.11	收集 Kubernetes 关键组件的日志信息

Kubernetes 集群环境搭建完毕后，在 master 节点查询关键组件的运行状况，查询结果如图 6-2。

```
[root@master ~]# kubectl --namespace=kube-system get pods
NAME                                          READY     STATUS              RESTARTS   AGE
dummy-2088944543-5yaek                        1/1       Running             1          14h
etcd-master.kubernetes.local                  1/1       Running             1          14h
heapster-2193675300-wsc4q                     0/1       ContainerCreating   0          1m
kube-apiserver-master.kubernetes.local        1/1       Running             2          14h
kube-controller-manager-master.kubernetes.local   1/1       Running             1          14h
kube-discovery-1150918428-7pw73               1/1       Running             1          14h
kube-dns-654381707-kp4hu                      3/3       Running             3          14h
kube-flannel-ds-60qq1                         2/2       Running             3          14h
kube-flannel-ds-qw6hn                         2/2       Running             6          14h
kube-flannel-ds-u37sx                         2/2       Running             8          14h
kube-proxy-lbdm4                              1/1       Running             1          14h
kube-proxy-seqkf                              1/1       Running             1          14h
kube-proxy-wwq32                              1/1       Running             1          14h
kube-scheduler-master.kubernetes.local        1/1       Running             1          14h
kubernetes-dashboard-3095304083-zsx4r         1/1       Running             0          57m
monitoring-grafana-2970430589-p9is5           0/1       ContainerCreating   0          1m
monitoring-influxdb-3276295126-80ehj          0/1       ContainerCreating   0          1m
```

图 6-2 master 节点关键组件运行情况

这时运行 Kubernetes Guestbook 应用测试集群各功能是否正常，在 Grafana 上查看集群资源信息，如图 6-3。

图 6-3 Grafana 展示集群资源信息

6.2 测试与实验结果分析

本节将设计测试用例对用户绑定调度策略和抢占式用户绑定策略进行功能测试，对动态负载均衡改进算法进行功能和性能测试。

6.2.1 用户绑定调度策略的测试与分析

用户绑定调度策略实现较为简单，算法复杂度较低，所以本节不对算法的性能进行测试，只测试算法的正确性。

主要对用户绑定策略的如下功能点进行测试：

1.匹配表达式的规则扩充；

2.匹配表达式的周期性检测；

3.Pod 指定目标节点不存在时，调度器的调度方式。

为了避免其他因素对测试产生影响，本节采用的 Pod 测试用例是自定义的，由系统分配端口号和存储卷位置，为每个 Pod 的设置的资源请求量为：Pod.resources.requests.cpu=10m，Pod.resources.requests.memory=64Mi，保证各工作节点有足够资源运行所有的 Pod。

各 Pod 测试用例的 multiNodeSelectors 字段设置如表 6-3：

表 6-3 匹配表达式规则扩充 Pod 测试用例

Pod 名	multiNodeSelectors

pod1	"multiNodeSelectors": {
	"nodeLabelChange": "reschedule",
	"expressions": [{
	"key": " hostname",
	"operator": "in",
	"values": ["node1"]
	}]}

pod2	"multiNodeSelectors": {
	"nodeLabelChange": " still",
	"expressions": [{
	"key": " tier",
	"operator": "notIn",
	"values": ["backend"]
	}]}

pod3	"multiNodeSelectors": {
	"nodeLabelChange": " reschedule",
	"expressions": [{
	"key": " hostname",
	"operator": "in",
	"values": ["node1", "node2"]
	}，{
	"key": "tier",
	"operator": "in",
	"values": ["frontend"]
	}]}

为三个工作节点指定如表 6-4 的标签。

表 6-4 各工作节点 labels 标签

Node 名	labels
node1	"hostname": " node1","tier": " frontend"
node2	"hostname": " node2","tier": " backend"
node3	"hostname": " node3","tier": " frontend"

使用 PreemptiveBinding 算法后，各工作节点运行的 Pod 情况如表 6-5。

表 6-5 调度后 Pod 分布情况

Node 名	运行的 Pod
node1	pod1、pod3
node2	
node3	pod2

pod1 指明匹配表示式 hostname 为 node1，最终 pod1 被调度到 node1 上运行；pod2 的匹配规则为 tier 不为 backend，匹配结果有 node1 和 node3，调度器根据这两个节点的资源状况指定调度目的地，因 node3 的配置最高且未运行任何 Pod，故 pod2 在 node3 上调度运行；pod3 指定了两条匹配规则，这两条规则是“且”的关系，最终符合匹配表达式的只有 node1，以上的实验结果表明匹配表达式的规则扩充是正确的。

下面对匹配规则的周期性检测进行测试，改变 Node 的 tier 属性值，详见表 6-6。

表 6-6 更改后的各工作节点 labels 标签

Node 名	labels
node1	"hostname": " node1","tier": " backend"
node2	"hostname": " node2","tier": " frontend "
node3	"hostname": " node3","tier": " backend"

此时各工作节点运行的 Pod 发生变化，详见表 6-7。

表 6-7 更改后的 Pod 运行情况

Node 名	运行的 Pod
node1	pod1
node2	pod3
node3	pod2

改变 Node 的 tier 标签对 pod1 的匹配结果没有影响，因此 pod1 依然运行在

node1 上;pod2 的匹配规则已不再满足,但由于它的 nodeLabelChange 属性为 still，因此 pod2 不改变宿主机；pod3 的匹配规则也不再满足，并且 nodeLabelChange=reschedule，故 pod3 重新选择目标节点，最终调度到 node2 上运行，由此可见，本文设计的匹配表达式周期检测是正确的。

为了测试候选节点为空时调度策略的正确性，设计如表 6-8 的测试用例。

表 6-8 候选节点为空的 Pod 测试用例

Pod 名	multiNodeSelectors

pod4	"multiNodeSelectors": {
	"collectionIsEmpty": "default",
	"expressions": [{
	"key": " hostname",
	"operator": "in",
	"values": ["node4"]
	}]}

pod5	"multiNodeSelectors": {
	"collectionIsEmpty": "reschedule",
	"expressions": [{
	"key": " hostname",
	"operator": "in",
	"values": ["node4"]
	}]}

运行 PreemptiveBinding 算法后发现，pod4 被调度到 node3 上运行，而 pod5 并没有在任何节点上运行。在集群中不存在 pod4 和 pod5 指定的目标节点,但 pod4 的 collectionIsEmpty 值为 default，在候选节点集合为空时忽略此匹配规则，综合考量各工作节点的资源情况，将 Pod 调度综合得分最高的节点上运行，而 pod5 由于指定 collectionIsEmpty 值为 reschedule，当集群不存在匹配表达式指定的目标节点则重新进入待调度 Pod 队列，等待再次被调度。

经过上文的测试与分析，本文在 PodSelectorsMatches 基础上改进的用户绑定调度策略，三个改进点切实可行。

6.2.2 抢占式用户绑定调度策略的测试与分析

首先对算法的正确性进行测试，为此设计如下测试场景：在某一节点上有若

干优先级不同的 Pod 正常运行，这时有不同优先级的 Pod 绑定此节点，对不同优先级的 Pod，该节点做出不同反应。

此处采用 node2 作为宿主机进行测试，在 node2 上运行两组优先级不同的 Pod，各 Pod 的相关描述见表 6-9。

表 6-9 node2 运行的 Pod 资源请求量

Pod 名	CPU 请求量	内存请求量
pod1、pod2、pod3	400m	800Mi
pod4、pod5、pod6	100m	200Mi

其中，为所有 Pod 指定 node2 为目标节点，pod1 和 pod4 的重启策略为 Never，pod2 和 pod5 的重启策略为 OnFailure，pod3 和 pod6 的重启策略为 Always。

Pod 创建后，使用 kubectl describe nodes 命令查看 node2 的资源可使用量，查询可得 CPU 可用量为 200m，内存可用量为 500Mi（这里为了方便测试结果的分析，本节将资源量进行四舍五入）。为了防止其它进程因为突发性占用大量资源对测试结果产生影响，不在 node2 上启动与本测试无关的进程。

在 node2 上依次创建并启动如表 6-10 的 Pod，每次创建完，记录 node2 上运行的 Pod 信息，记录完成后将 node2 上新创建的 Pod 删除，将被抢占的 Pod 重新调度到 node2 运行，node2 上运行的 Pod 恢复至表 6-9 的情况。重复上述操作，记录结果如表 6-11。

表 6-10 node2 接收的 Pod 信息

待创建 Pod 名	重启策略	CPU 请求量	内存请求量
neverPod1	Never	300m	1000Mi
neverPod2	Never	500m	2000Mi
onfailurePod1	OnFailure	300m	1000Mi
onfailurePod2	OnFailure	500m	2000Mi
alwaysPod1	Always	300m	1000Mi
alwaysPod2	Always	500m	2000Mi

由表 6-11 可以看出，在宿主机资源不足时，只有高优先级的 Pod 可以抢占低优先级 Pod 的资源，如：onfailurePod1、alwaysPod1、alwaysPod2，同优先级的 Pod 不可互相抢占资源，如 neverPod1、neverPod2、onfailurePod2。其中 onfailurePod2 没有成功启动运行是因为宿主机在将重启策略为 Never 的 pod1 和 pod4 回收后，宿主机的可用资源量仍小于 onfailurePod2 的请求量，而 onfailure Pod 只可以抢占

优先级比其低的 Never Pod，故 onfailurePod 调度失败。

表 6-11 node2 运行的 Pod

创建 Pod 名	node2 运行的 Pod
neverPod1	pod1、pod2、pod3、pod4、pod5、pod6
neverPod2	pod1、pod2、pod3、pod4、pod5、pod6
onfailurePod1	pod2、pod3、pod4、pod5、pod6、onfailurePod1
onfailurePod2	pod1、pod2、pod3、pod4、pod5、pod6
alwaysPod1	pod2、pod3、pod4、pod5、pod6、alwaysPod1
alwaysPod2	pod3、pod5、pod6、alwaysPod2

证明本文将 Pod 按重启策略划分优先级，在宿主机资源不足时进行的抢占式调度与设计的抢占式调度规则是吻合的。

下面比较 PreemptiveBinding 算法与 default 算法的高优先级 Pod 运行比例，为此设计三种空 Pod，三种 Pod 的重启策略分别为 Never、OnFailure、Always，其它配置信息相同，这里为了能在宿主机上尽可能多的运行 Pod，指定 Pod 的资源请求量为：Pod.resources.requests.cpu=10m，Pod.resources.requests.memory=50Mi。

分别使用 PreemptiveBinding 算法和 default 算法向 API Server 发起在 node3 上创建 Pod 的请求，一次发送 30 个，三种优先级的 Pod 各占 10 个，记录两种算法的各优先级 Pod 运行比例，并绘制图 6-4 的折线图。

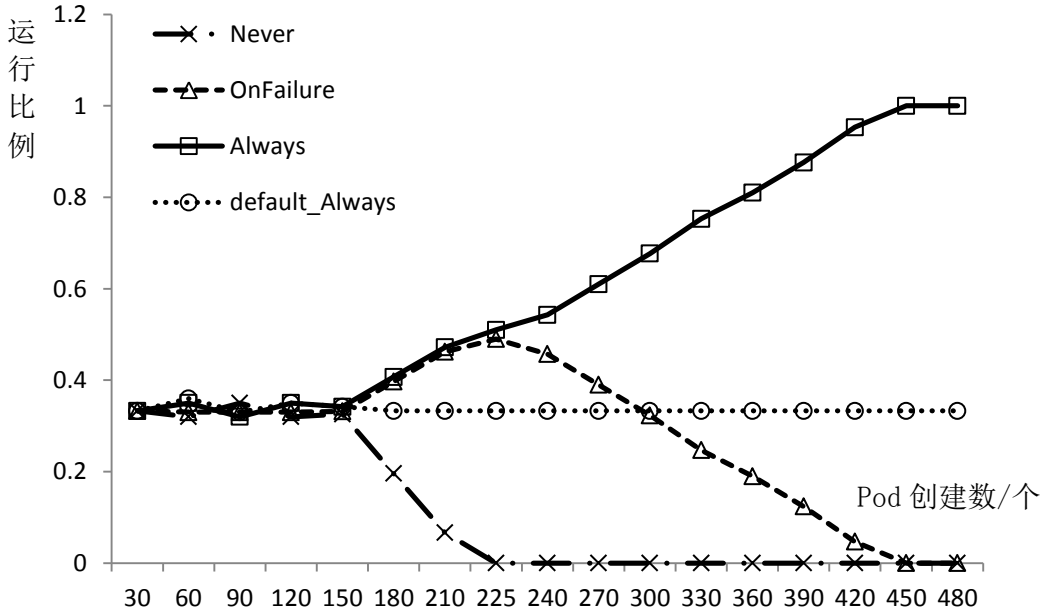

图 6-4 各优先级 Pod 运行比例对比图

由上图可以看出，使用 PreemptiveBinding 策略时，当 Pod 创建数量较少时，三种优先级的 Pod 运行比例相当，都在三分之一左右，但当 Pod 继续创建时，重启策略为 OnFailure 和 Always 的 Pod 会抢占 Never Pod 的资源，所以 OnFailure 和 Always Pod 的运行比例才会呈线性增长，当 node3 中所有 Never Pod 都被回收，这时候 Always Pod 开始抢占 OnFailure Pod 的资源，直至 node3 中 OnFailure Pod 的资源被抢占完，此时，node3 中所有的 Pod 重启策略都为 Always。

而 default 算法由于没有抢占式调度策略，当 node3 资源充足时，三种 Pod 的运行比例都在三分之一左右，当 node3 的资源使用完后，不论是何种重启策略的 Pod 皆不能调度运行，三种优先级的 Pod 运行比例相同。

上述的两种测试方法，证明本文设计的抢占式用户绑定调度策略在节点资源不足时可以有效地进行抢占式调度，与 default 算法相比能大幅提高高优先级 Pod 的运行比例。

6.2.3 基于 Kubernetes 的动态负载均衡改进算法的测试与分析

本节将从两个角度对本文设计的基于 Kubernetes 的动态负载均衡改进算法进行测试：动态负载均衡和性能。动态负载均衡主要测试当集群环境发生变化时，改进的算法是否能够较好的维持系统的负载均衡；性能测试主要比较改进算法与 default 算法的执行效率，此处用 Pod 的平均调度时间衡量。

6.2.3.1 动态负载均衡测试

为了对本算法的动态负载均衡进行测试分析，本节模拟了这样的实验环境：各资源规格不一致的工作节点在面对负载不均衡的场景下可以进行动态调整，使集群整体负载趋于稳定和均衡。

本节使用的 Pod 测试用例是改进版 Guestbook，是根据 Kubernetes 官方的 Guestbook 进行裁剪的，去除了 RC 的限制，将原来分散的 Docker 封装在一个 Pod 里，这样比原来更加轻量。改进版 Guestbook（以下简称为 Guestbook）是一个容器化 Web 应用，由 3 个 Docker 组成，它提供了一个非常简单的功能：在 Web Frontend 提交数据，Web Frontend 将数据保存到 Redis Master，然后从 Redis Slave 读取数据显示到 Web Frontend 上，运行结构如图 6-5。

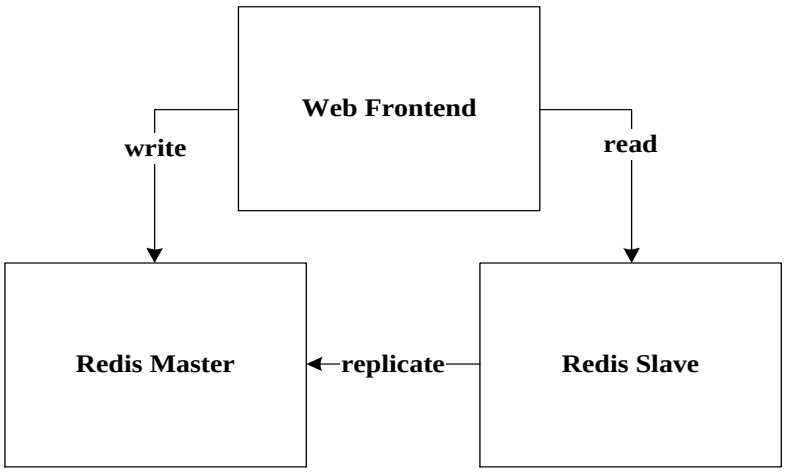

图 6-5 Guestbook 运行结构图

考虑到要让外网用户可以访问各节点上的 Guestbook 应用的前端 Docker 端口，因此需要修改 Guestbook 的资源描述文件，将 spec.type 属性设为 NodePort，这样通过任何工作节点的 IP 和 NodePort 即可访问到该节点指定 Guestbook Pod 的前端端口，其中 NodePort 指定在 5000-5050 之间，便于后续网站测试工具发起随机连接。

模拟各节点的负载状况不同使用的方法是在各节点上运行一定数目的 Guestbook 应用程序，在客户端使用 webbench 网站压力测试工具同时向多个节点的多个 Guestbook 发起连接，以模拟出工作节点正在承受高并发请求而负载变重，其中 webbench 的连接数和连接的端口随机生成。为了防止某一个 Pod 无限制占用系统资源，将 Pod 的 requests.memory.limits 设为 128M。

具体实验步骤如下：

1.在客户端向 master 节点的 API Server 组件发起 Guestbook 创建命令，创建方式为每次创建 15 个（其中三种重启策略的 Pod 各 5 个），每隔 5 分钟创建一次，共创建 60 个 Pod；

2.在客户端用 webbench 向各工作节点并行发出 web 请求，请求数和请求端口由随机函数随机生成；

3.在 master 节点上监测并记录各节点的资源使用情况，记录方式为：刚创建 Guestbook 后记录一次，创建后 4 分 30 秒一次。刚创建记录是为了计算系统调度之初的负载均衡度，而创建一段时间后记录是为了测试随着集群环境的变化，在调度之初的负载均衡度发生怎样的变化；

4.使用 Kubernetes 的 default 算法，重复上述步骤。

考虑到 Kubernetes 的 default 算法并未将网络作为资源调度的衡量因素，这里

为了更好的与 default 算法进行负载均衡度对比，在实际测试中并未将镜像下载速度和数据传输速度作为评分的衡量因素，故这里将需求权值因子设置为：$\lambda_{cpu}=\lambda_{memory}=0.5$。

根据步骤 3 统计的资源使用情况，利用公式(5-12)到公式(5-18)计算各节点的资源得分和总得分，表 6-12 展示的是使用改进算法根据第八次记录的资源使用情况计算的各资源得分。

表 6-12 使用改进算法第八次记录各节点的资源得分

节点名	Pod 数量	cpuScore	memoryScore	Score
Node1	10	0.94	1.01	-0.26
Node2	16	1.12	1.07	0.90
Node3	34	1.06	0.84	-0.58

从上表中可以看出，尽管各节点的资源规格不同，但是各节点的不同资源间得分差异不大，不存在某一节点或某一资源使用率过高造成系统瓶颈，维持了集群和宿主机的整体性能。

分别使用改进算法和 default 算法按照上述实验步骤记录的八次结果计算各节点的综合得分，根据综合得分变化趋势绘制图 6-6 和图 6-7。

图 6-6 可以看出，前两次检测时，各节点的总得分差异较大，集群的负载均衡程度较差，但是从第三次检测开始，系统中 Pod 数达到 30 个，各节点的得分逐渐趋于稳定并接近 0。当 Pod 数量较少时，各节点的分值差别较大，主要是两方面原因：1.各节点运行的 Pod 数量较少，资源充足，未达到动态调度的触发条件，主要依靠静态调度的方式进行调度；2.本测试采用的网站压力测试工具是随机的对 Guestbook 进行请求，Pod 数较少时，大多 Web 请求都会失效，容易出现连接成功的 Web 请求集中在某个节点的情况。随着创建 Pod 的增多，成功连接的 Web 请求就不会扎堆在某一个节点上，同时各节点的资源也越来越紧张，这时候触发动态调度进行 Pod 迁移，在动态调度作用下，集群中各工作节点得分接近，系统负载均衡度较好。另外在 Pod 数量较高时，虽然三个节点的资源规格不一，但是综合得分却相近，说明改进算法不受节点资源差异的影响。

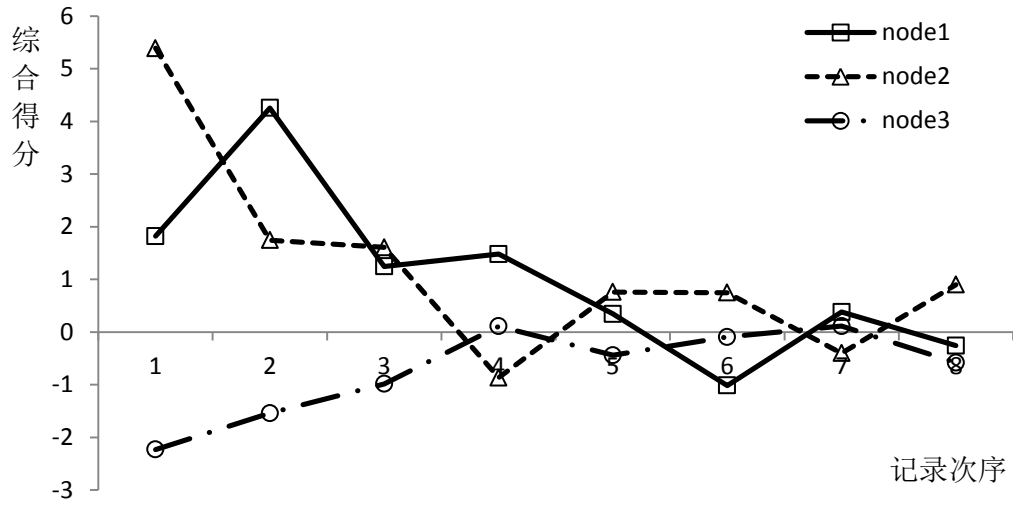

图 6-6 改进算法各节点综合得分变化曲线图

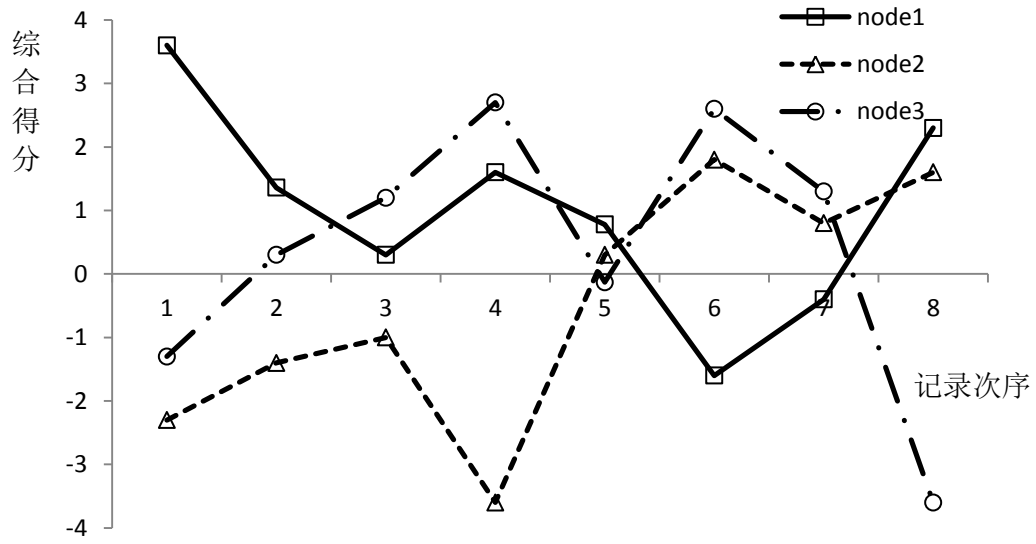

图 6-7 default 算法各节点综合得分变化曲线图

图 6-7 与图 6-6 的前两次记录情况类似，各节点的得分差异较大，为了避免因为 Pod 数较少 Web 请求扎堆的现象发生而影响分析，这里只比较第三次记录之后的情况。使用 default 算法的各节点在第 3、5、7 次出现了得分相近的情况，而第 4、6、8 次得分又趋于分散，这是由于 default 算法只有静态调度，奇数次记录的是刚经过静态调度的节点得分情况，可见 dafault 的在静态负载均衡上有较好的表现，但是随着时间的推移，webbench 的持续请求，各节点的资源状况逐渐发生改变，所以在偶数次观测的时候，得分又趋于分散。

对于各工作节点主要通过各资源的资源利用率计算最终得分，分值体现各节

点相对于集群的负载程度，大于 0 表示负载优于集群负载均值，小于 0 则劣于负载均值。而对于整个系统的负载均衡程度可以通过各节点之间的负载标准差来体现，得分越接近则系统负载越均衡。

为了更好的衡量两种算法的负载均衡度，引入下述公式计算系统的整体负载程度。

$$averageScore = \frac{1}{n}\sum_{1}^{n} score_i \tag{6-1}$$

$$stdScore = \sqrt{\frac{1}{n}\sum_{1}^{n}(score_i - averageScore)^2} \tag{6-2}$$

其中 *averageScore* 为系统平均得分，由各节点的综合得分的算术平均值计算得出，*stdScore* 为得分标准差，用来衡量系统中所有节点的负载差异：得分标准差越大，各节点负载差别越大，反之系统的整体负载更加均衡。

下面利用得分标准差比较改进的算法与 default 算法的系统负载均衡程度，按照上述公式计算并绘制图 6-8。

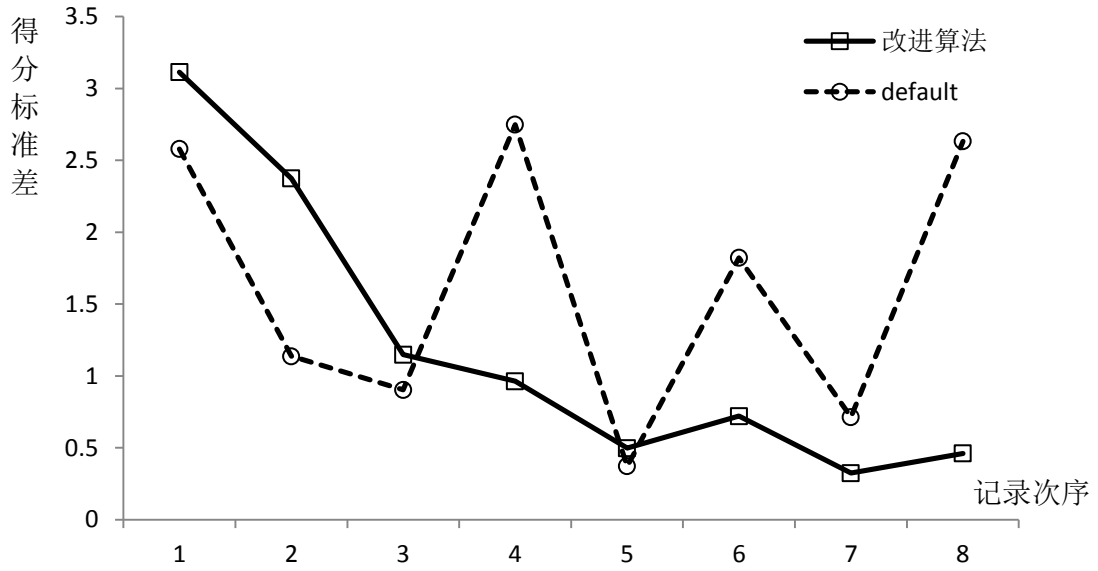

图 6-8 得分标准差变化图

上图可见，这两种算法在奇数次观测，标准差相近，偶数次观测则差别明显。default 算法本身就是为静态调度设计的，因此在奇数次观测时 default 算法都可以用静态调度的方式使系统的整体负载均衡，改进算法则通过静态调度和动态调度结合的方式来维持系统的均衡，所以尽管集群环境变了，但各节点的均衡度变化不大，而使用 default 算法的集群负载标准差波动较大。

经过上面的分析可以得出：default 算法在进行静态调度的时候可以维持系统的整体均衡度，但是随着系统环境和 Pod 运行状态的变化，系统均衡度出现较大的波动，本文针对 Kubernetes 改进的动态负载均衡算法通过静态调度和动态调度两种方式结合使用可以较好的维持系统的均衡性。

6.2.3.2 性能测试

本节的性能测试是为了比较改进算法和 default 算法的执行效率，这可以通过 Pod 的调度时间来间接比较，因此设置如下测试场景：在一个部署了 Kubernetes 的大型集群中创建大量 Pod，通过日志文件记录的 Pod 创建总时间可以评估系统的性能。

由于实验成本的原因无法使用大规模集群来进行性能测试，这里使用 Kubemark 工具来模拟较大规模集群环境。Kubemark 是 Kubernetes 社区推出的节点模拟工具，它可以在一个 CPU 核心上模拟 10 个真实节点，本文采用了单核、双核、四核处理器的服务器各一台，因此可以模拟 70 个节点的大型集群。

具体实验步骤如下：

1.在三台服务器上安装 Kubemark 工具；

2.向控制节点的 APIServer 发送 Pod 创建命令，创建 100 个 Pod；

3.从日志文件获取总调度时间；

4.删除所有的 Pod，将创建 Pod 数依次改为 200、300、…1600，重复上述步骤；

5.根据实验结果绘制图 6-9。

从图中可以看出，当创建 Pod 数较少时，二者的总调度时间相差无几，当 Pod 创建数量增多时，default 算法的 Pod 平均创建时间几乎没有变化，而改进算法的平均调度时间则有略微的上涨。default 算法从此至终都是使用静态调度算法，执行效率较高，稳定度较好。而改进算法在 Pod 数较少时，主要使用静态调度的方式，所以此时执行效率与 default 几乎持平；但 Pod 创建数增多，这时会触发动态调度，有一些节点的某些 Pod 被重新放回待调度 Pod 队列，所以总调度时间比 default 算法略高，但是这还不至于影响系统的吞吐量。另外可以看出两种算法的稳定性较好，不存在平均调度时间大幅变化的情况。

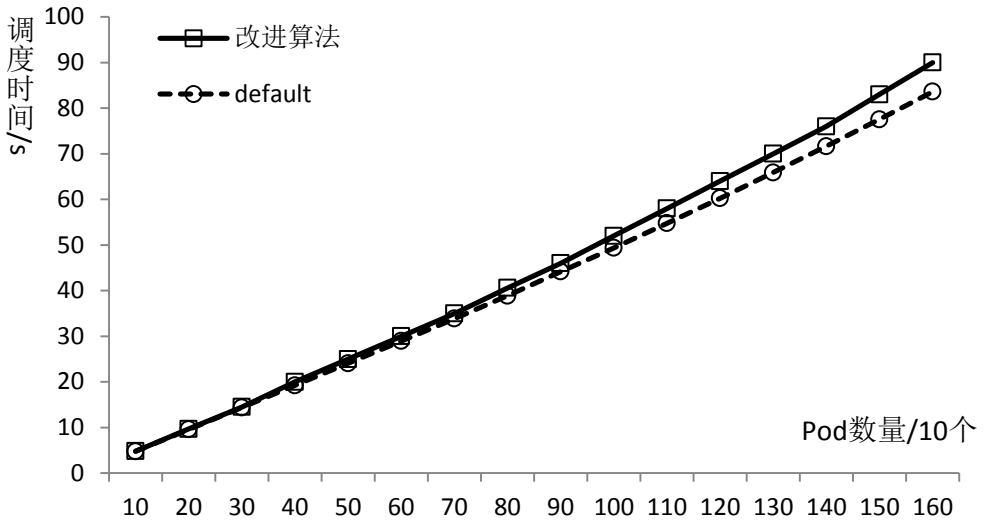

图 6-9 Pod 调度总时间对比图

综上所述，本文设计的基于 Kubernetes 的动态负载均衡改进算法使用静态调度和动态调度结合的方式可以较好的维持集群的负载均衡，并且算法的执行效率和稳定性较 default 算法相差不多。

6.3 本章小结

本章搭建了 Kubernetes 云平台并分别针对三种调度策略设计测试用例，测试表明：改进的用户绑定调度策略各功能点正确可行；抢占式用户绑定调度策略在集群资源不足时可以有效的提高高优先级 Pod 的运行比例；基于 Kubernetes 的动态负载均衡改进算法在集群负载较重时可以有效的维持系统的整体负载均衡，并且算法的性能与稳定性较 default 算法相差不多。

第七章 总结与展望

7.1 总结

本文从 Docker 技术出发，分析了 Docker 及以 Docker 为基础的云平台的核心理念和关键技术，从而引起了对一些主流的容器集群资源管理系统的学习，并最终选择 Kubernetes 进行深入研究，在对 Kubernetes 的资源调度模块进行学习过程中发现了 Kubernetes 的资源调度策略存在以下缺陷：

1.Kubernetes 的资源模型只包括 CPU 和内存，进行资源调度的考量因素较少；

2.Kubernetes 只支持静态调度，非异常情况下，Pod 在其生命周期内都不会改变其宿主机；

3.Kubernetes 现有的用户绑定调度策略是一种强绑定，策略总体较为简单，无法面对复杂的应用场景；

4.不支持抢占式调度，所有 Pod 优先级相同，在宿主机资源不足时，其它 Pod 无法调度到该机器运行。

通过对其它云平台的资源调度策略和 Kubernetes 系统的研究，并结合实际应用场景，本文主要做出如下工作：

1.提出了一种改进的 Kubernetes 资源模型，在进行资源调度时综合考虑 CPU、内存、Pod 与镜像存储系统间的镜像下载速度、Pod 与持久化存储系统间的数据传输速度四方面因素。在实际实验中，未找到较合理的测试方案对此项改进进行测试。

2.用户绑定调度策略改进，扩充现有的用户绑定匹配表达式，支持对表达式的匹配性周期检测，将现有的“强绑定”改为“弱绑定”。在实际测试中，改进的用户绑定调度策略各功能点正确有效，可以应对用户较复杂的绑定规则。

3.设计抢占式资源调度策略，根据 Pod 的重启策略划分三个优先级：Always>OnFailure>Never。在测试中，当宿主机资源不足时，高优先级 Pod 可以抢占低优先级 Pod 的资源，抢占式调度策略可以有效的提高高优先级 Pod 的运行比例。

4.改进一种动态负载均衡算法并应用到 Kubernetes 云平台上，将上述所有调度策略整合为静态调度和动态调度两部分，当待调度 Pod 队列存在候选 Pod 时选择静态调度方式进行调度，当集群系统存在某些节点负载过高时进行动态调度使系统整体负载均衡。经过测试证明：与 default 算法相比，本文设计的调度策略在多变的集群环境中可以有效的维持系统的负载均衡，但当 Pod 创建数量较多时，性

能略差于 default 算法。

7.2 展望

本文后续的研究方向如下：

1.本文的研究主要是针对 Kubernetes 资源调度策略，设计的调度策略与 Kubernetes 其它组件融合度不高，例如可以将本文设计的负载均衡策略与 Replication Controller 和 Service 融合；

2.本文改进的资源模型考虑了网络因素对调度策略的影响，但是在实际场景中，因为网络波动较大，宿主机的网络因素不容易衡量，应该对网络因素对资源调度的影响方向进行深入研究；

3.本文通过为 Pod 资源描述文件添加相关匹配规则的方式为 Pod 指定了 Node，但是实际使用中，也可以为 Pod 绑定 Pod，例如，有些 Pod 功能相互依存，此时最好将它们调度到同一节点上运行，或有些 Pod 相互冲突，应将它们分开部署；

4.本文是根据 Pod 的重启策略进行优先级划分的，这种划分方式在一些场景下并不够准确，后期研究应该对 Pod 进行任务类型区分，制定更准确的优先级划分方式；

5.本文根据集群负载状况进行动态调度，这种动态调度实际上是将某个正在运行的 Pod 强行终止，然后将其放回待调度 Pod 队列，等待下次调度为其选择更合适的宿主机，这种方式并不合理，在后期的研究中应该着重于 Pod 的热迁移研究。

6.由于成本的问题，本文的测试集群只有四台主机（一台控制节点，三台工作节点），所以设计的测试用例也比较局限，在成本允许的情况下，后期研究可以设计更多样的测试用例对本文的调度策略进行测试研究。

致谢

时光荏苒，岁月如梭，三年硕士学习生涯不经意间悄然逝去。回顾这三年，老师的谆谆教诲，同学的同窗共读，让我收获满满。

首先衷心感谢我的导师于鸿洋副教授。经师易遇，人师难遭。于老师不仅对我的实验和论文给予了巨大支持，而且以身践行教会了我许多做人做事的道理。学术上，于老师是我的良师，他用渊博的知识和开阔的视野为我指引了研究方向，循循善诱教会我如何分析问题和解决问题，而不是一味地授之以鱼。生活上，于老师是我的益友，他对生活的激情和热爱不断地激励着我，让我坚定方向，紧紧抓住每分每秒，脚踏实地做好每一件事。在这里，我再次感谢于老师三年来对我的指导和帮助，我将永生铭记！祝愿于老师阖家幸福、健康快乐！

其次要感谢教研室的各位兄弟姐妹，三人行，必有我师焉。从您们身上，我获益匪浅。在科研道路上，您们帮助我解惑，让我感受到集体的力量是伟大的，一加一远远大过于二；生活上，我们相亲相爱，让远走他乡的我并不孤单，身边充满了爱和关怀。

最后要感谢我的父母对我的养育之恩。您们不仅赋予我生命，而且含辛茹苦地供我上学直至今日。对于曾经任性的我，您们也从未放弃，不断地鼓励我、教导我，在这里想对您们说：您们辛苦了！同时还要感谢我的哥哥，因为有你作为榜样，一直鞭策着我，让我不轻易言败，再苦再累也继续坚持。

我由衷地感谢身边的每个人，谢谢家人的支持、老师的教导、同学的帮助、朋友的关心！谢谢您们！

参考文献

[1] 龚正,吴治辉,叶狄荣.Kubernetes 权威指南:从 Docker 到 Kubernetes 实践全接触[M].电子工业出版社,2016, 165-246.

[2] V. Vasudevan. Performance Evaluation of Resource Allocation Policies on the Kubernetes Container Management Framework[J]. 2016.

[3] 浙江大学 SEL 实验室.Docker 容器与容器云[M].北京:人民邮电出版社,2015, 15-114.

[4] A. M. Joy. Performance comparison between linux containers and virtual machines[C]. Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in. IEEE, 2015: 342-346.

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

[6] X. G. Yin. Research and Implementation PaaS platform based on Docker[C]. International Informatization and Engineering Associations、Atlantis Press.Proceedings of 2015 4th National Conference on Electrical,Electronics and Computer Engineering (NCEECE 2015). International Informatization and Engineering Associations、Atlantis Press, 2015: 6.

[7] Z. H. Li,Y. Zhang,Y. H. Liu. Towards a Full-Stack Dev Ops Environment (Platform-as-a-Service) for Cloud-Hosted Applications[J]. Tsinghua Science and Technology, 2017, (01): 1-9.

[8] 李雨前,杨星飞.面向容器技术资源调度关键技术对比[EB/OL]. http://www.infoq.com/cn/articles/resource-scheduling-for-container-technology,2016.

[9] 陈星宇.基于容器云平台的网络资源管理与配置系统设计与实现[D].浙江大学,2016.

[10] 卢胜林,倪明,张翰博.基于 Docker Swarm 集群的调度策略优化[J].信息技术,2016,(07): 147-151+155.

[11] 毛祺,卢胜林.基于 Docker Swarm 集群的容器迁移策略的实现[J].信息技术,2016,(09): 156-160.

[12] A. Grillet. Hot Topics in ISE-Topic 5 Comparison of Container Schedulers[J].

[13] H. Saito, H. C. C. Lee, K. J. C. Hsu. Kubernetes Cookbook[M]. Packt Publishing Ltd, 2016.

[14] H. Tariq. Understanding and Evaluating Kubernetes[J].

[15] L. V. Sudalaikkan. Preservation of Low Latency Service Request Processing in Dockerized Microservice Architectures[D]. Colorado State University. Libraries, 2017.

[16] A. Balalaie, A. Heydarnoori, P. Jamshidi. Microservices architecture enables DevOps:

migration to a cloud-native architecture[J]. IEEE Software, 2016, 33(3): 42-52.

[17] W. Wang. Kubernetes Scheduler 源码分析 [EB/OL]. http://blog.csdn.net/waltonwang/article/details/54565638, 2017.

[18] C. Sanchez. Scaling Docker with Kubernetes[J]. Website. Available online at http://www. infoq. com/articles/scaling-docker-with-kubernetes, 2015: 35.

[19] 梁俊杰.基于应用容器的云资源调度研究与实现[D].电子科技大学,2015.

[20] 余浩维.PaaS 云中 Web 容器及调度的设计与实现[D].北京邮电大学,2015.

[21] 杜军.基于 Kubernetes 的云端资源调度器改进[D].浙江大学,2016.

[22] D. Vohra. Configuring Compute Resources[M]. Kubernetes Management Design Patterns. Apress, 2017: 237-256.

[23] A. Reuther, C. Byun, W. Arcand, et al. Scheduler technologies in support of high performance data analysis[C]. High Performance Extreme Computing Conference (HPEC), 2016 IEEE. IEEE, 2016: 1-6.

[24] H. C. Deng. Improving Kubernetes Scheduler Performance[EB/OL]. https://coreos.com/blog/improving-kubernetes-scheduler-performance.html, February 22, 2016

[25] 刘敏献.基于 Docker 的服务调用拓扑分析和性能监控系统的设计与实现[D].浙江大学,2016.

[26] 仇臣.Docker 容器的性能监控和日志服务的设计与实现[D].浙江大学,2016.

[27] 孙庚泽.基于 Docker 的混合云应用编排方案研究[D].西安电子科技大学,2015.

[28] 刘思尧,李强,李斌.基于 Docker 技术的容器隔离性研究[J].软件,2015,(04): 110-113.

[29] 张建,谢天钧.基于 Docker 的平台即服务架构研究[J].信息技术与信息化,2014,(10): 131-134.

[30] K. Y. Lv. PCCTE:A Portable Component Conformance Test Environment Based on Container Cloud for Avionics Software Development[C]. Proceedings of the 2016 IEEE International Conference on Progress in Informatics and Computing(PIC). IEEE Beijing Section,China、上海财经大学、同济大学, 2016, 5.

[31] 刘琨.云计算负载均衡策略的研究[D].吉林大学,2016.

[32] 郑海荣,高敬.云计算技术与应用[J].通信技术,2013,(04): 96-98+102.

[33] 张丽梅.基于负载均衡的云资源调度策略研究[D].宁夏大学,2014.

[34] 薛玉.云计算环境下的资源调度优化模型研究[J].计算机仿真,2013,(05): 362-365.

[35] 朱泽民,张青.基于多维 QoS 和云计算的资源负载均衡调度研究[J].计算机测量与控制,2013,(01): 263-265+281.

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

[37] 王智明.云数据中心资源调度机制研究[D].北京邮电大学,2012.

[38] 田文洪,赵勇.云计算-资源调度管理[M].北京:国防工业出版社,2011, 15-115.

[39] P. Acuña. Kubernetes[M]. Deploying Rails with Docker, Kubernetes and ECS. Apress, 2016.

[40] K. Y. Lv. PCCTE:A Portable Component Conformance Test Environment Based on Container Cloud for Avionics Software Development[C]. Proceedings of the 2016 IEEE International Conference on Progress in Informatics and Computing(PIC). IEEE Beijing Section,China、上海财经大学、同济大学, 2016, 5.

[41] V. Medel, O. Rana,U. Arronategui. Modelling performance & resource management in kubernetes[C]. Proceedings of the 9th International Conference on Utility and Cloud Computing. ACM, 2016: 257-262.

攻读硕士学位期间取得的研究成果

[1] 唐瑞,于鸿洋. 一种基于 Docker 云平台的优先级队列动态反馈负载均衡资源调度方法[P]. 中国. 发明专利. 201710199712.4, 2017 年 3 月 29 日