2023 届研究生硕士学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：51205903032

East China Normal University

# 硕士学位论文

## MASTER'S DISSERTATION

# 论文题目：基于 Kubernetes 的弹性伸缩与动态调度策略设计与实现

| | |
|---|---|
| 培养单位: | 数据科学与工程学院 |
| 学　　科: | 软件工程 (数据科学与工程) |
| 研究方向: | 计算教育学 |
| 指导教师: | 钱卫宁 教授，王伟 教授 |
| 学位申请人: | 史经犇 |

2023 年 05 月

Dissertation Master's Degree in 2023

University Code: 10269

Student ID: 51205903032

EAST CHINA NORMAL UNIVERSITY

# Design and Implementation of elastic scaling and dynamic scheduling strategy based on Kubernetes



| | |
|---|---|
| Department: | School of Data Science and Engineering |
| Discipline: | Software Engineering (Data Science and Engineering) |
| Research direction: | Computational Education |
| Supervisor: | Prof.QIAN Weining, Prof.WANG Wei |
| Candidate: | SHI Jingben |



May, 2023

# 摘 要

随着云计算技术的不断发展，以 Docker 与 Kubernetes 为核心的云原生技术逐渐普及，Kubernetes 凭借其强大、丰富的容器编排能力已经成为目前容器编排的事实标准。但是，Kubernetes 中默认的伸缩机制与调度机制均比较简单、单一，在实际使用时存在着一定的不足，在复杂的应用场景下难以满足不同用户的需求。Kubernetes 默认伸缩机制与调度机制所存在的缺陷主要表现在，Kubernetes 默认基于水平伸缩控制器 (HPA) 实现了被动的响应式伸缩机制，在扩缩容时可能会响应不及时，影响服务质量、浪费集群资源。而 Kubernetes 默认的静态调度机制仅考虑 CPU 与内存资源来进行节点调度，仅考虑 CPU 和内存资源的调度方式可能会导致集群资源性能瓶颈问题以及集群负载不均衡问题；静态调度机制无法根据 Kubernetes 最小管理单元 Pod 的运行状况动态调整资源，存在一定的资源浪费。为解决上述问题，本文以 Kubernetes 为研究对象，针对 Kubernetes 中默认的伸缩机制与调度机制进行改进。主要工作内容如下:

(1) **构建一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略:** 本文构建了一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与 HPA 配置的弹性伸缩策略, 在调度时能够通过负载预测结果与集群中 HPA 配置对 Kubernetes 中的基本管理单元 Pod 提前进行扩缩容操作，以此来应对突发性的任务负载。该自定义弹性扩缩容策略在高负载来临之前，可以提前扩容，从而降低用户请求的平均响应时间，提升服务质量；在低负载来临之前，可以提前缩容，从而减少集群中不必要的资源浪费。

(2) **构建自定义 Kubernetes 扩展调度器，并提出一种基于集群负载均衡度的调度策略:** 本文基于 Scheduling Framework 构建了自定义 Kubernetes 调度器，该调

度器在进行节点调度时, 会考量多种资源来选择最合适调度的节点, 比如 CPU、内存、网络 I/O、磁盘 I/O、磁盘容量等资源指标。此外，本文还提出一种基于集群负载均衡度的调度策略，在进行节点打分时，首先划分 Kubernetes 最小管理单元 Pod 的类型，然后根据 Pod 类型和所需资源对节点进行资源倾向性打分，节点上各种资源的均衡度分数越高，则节点分数越高，该节点越适合调度。其中，通过使用层次分析法 (AHP) 计算节点上各类资源的均衡度分数在节点分数中的权重，优化节点打分机制。该策略提升了调度后集群负载的均衡性，有效缓解了节点上某种类型资源的性能瓶颈问题。

(3) **提出一种动态调整 Pod 资源限额的方案:** 本文基于 Kubernetes 基本部署单元 Pod 的实际运行状态提出一种动态调整 Pod 资源限额的方案，通过 Pod 的历史资源使用量计算 Pod 在运行时的资源需求，使用 Linux 控制群组机制 (Cgroup) 直接对 Pod 资源参数进行调整，从而在不重新构建 Pod 的情况下实现 Pod 中容器资源配额的动态修改，修改后配置立即生效。该方案提升了集群资源的利用率，减少了 Pod 预留资源的浪费。

总的来说，本文所提出的三个工作点优化了 Kubernetes 中默认的伸缩机制和调度机制，主要解决了资源利用和负载均衡问题，提高了集群资源利用率，降低了资源使用成本，可以为相关云计算研究人员在处理资源利用和任务调度问题时提供一定的借鉴意义。

**关键词：** Docker, Kubernetes, 负载预测, 负载均衡, 动态调度

# ABSTRACT

With the continuous development of cloud computing technology, cloud-native technology with Docker and Kubernetes as the core has gradually become popular. And Kubernetes has become the standard for container orchestration with its powerful container orchestration capability. However, the default scaling mechanism and scheduling mechanism in Kubernetes are both relatively simple and single which have certain shortcomings in actual scenario, and it's difficult to meet the needs of different users in complex application scenarios.

There are the following drawbacks in default scaling and scheduling mechanisms of Kubernetes. By default, Kubernetes implements reactive and responsive scaling mechanism based on the horizontal scaling controller (HPA), and it maybe occur that Kubernetes responses untimely, and it affects service quality and wastes cluster resources. Kubernetes only considers CPU and memory resources by default static scheduling mechanism during scheduling, which maybe lead to performance bottlenecks for some resources and load imbalance in Kubernetes cluster. This static scheduling mechanism cannot dynamically adjust the configuration parameters of running Pods, and there is a certain amount of resource waste. Therefore, this thesis takes Kubernetes as the research object, and proposes solutions to the problems of existing scaling and scheduling mechanisms in Kubernetes. The main work in this thesis is as follows.

(1) **Constructing an ARIMA-GRU combined prediction model with HPA-based elastic scaling strategy：** This thesis constructs a elastic scaling policy in Kubernetes based on Autoregressive Integrated Moving Average-Gate Recurrent Unit combined Model and Horizontal Pod Autoscaling(HPA). Kubernetes use the custom scaling policy to scale up and down replicas of Pods in advance to cope with sudden loads based on load forecast results and HPA configuration. By using custom scaling policy, Kubernetes scale up replicas of Pods in advance before the high load, thus reducing the average response time of user requests and improving the quality service. Besides, by using this custom scaling policy, Kubernetes scale down replicas of Pods in advance before the low load to reduce unnecessary cluster resources waste.

(2) **Building a custom Kubernetes scheduler and proposing a scheduling policy based on cluster load balance degree：** This thesis builds a custom Kubernetes scheduler base on the Scheduling Framework. The improved scheduler selects the most appropriate nodes for Pods during scheduling by taking various resources metrics into account, such as CPU, memory, network I/O, disk I/O, disk capacity, and other resource metrics. In addition, the thesis proposes a scheduling strategy based on cluster load

balancing degree. When Kubernetes scores nodes, it first classifies Pod, then scores nodes based on the types and required resources of Pods. The higher the balancing degree scores of various resources on a node, the higher the score of that node and the more suitable that node is for scheduling. Among scoring nodes, Kubernetes uses the analytic hierarchy process(AHP) to calculate the weight of various resources balancing degree scores in node score to optimize node scoring mechanism. This scheduling strategy improves cluster load balance after scheduling and effectively alleviates the performance bottleneck of resources on nodes.

(3) **Proposing a scheme to dynamically adjust Pod resource limits：** This thesis proposes a scheme to dynamically adjust Pod resource limits, and Kubernetes calculate the resource demand of Pods during operation through historical resource usage of Pods and adjusts Pods resources using Linux control group (Cgroup) mechanism. This scheme dynamically adjusts Pod resource limits without rebuilding the Pod and the change takes effect immediately. This scheme improves the utilization of cluster resources and reduces the waste of reserved resources for Pods.

In summary, the three points proposed in this thesis optimize the default scaling mechanism and scheduling mechanism in Kubernetes. The main points mainly solve the problems of resource utilization and load balancing, and reduce the cost of resource usage. Also, they provide some reference for relevant researchers on cloud computing in dealing with resource utilization and task scheduling.

**Keywords:** *Docker, Kubernetes, Load Prediction, Load Balancing, Dynamic Scheduling*

# 目　录

# 插图

# 表格

# 第一章　绪论

## 1.1　研究背景

云计算是将网络资源、数据库资源、存储资源、计算资源等资源通过互联网提供给用户使用的一种计算模式，用户可以按照自己的需求使用云计算服务，并且只为最终使用到的资源进行付费即可 [1, 2]。云计算技术支持用户、企业或者公司按需获取计算能力，云计算技术具有灵活的弹性扩展能力 [3]；并且，云计算技术简化了开发者在全球不同地域部署应用的流程；此外，云计算技术中海量数据的管理技术也保证了数据的安全性以及数据访问的高效性。

云计算按照不同的分类标准，可以划分为多种类型。如果从服务模式上进行划分，云计算自下到上可以分为 IaaS 服务、PaaS 服务、SaaS 服务 [4, 5]。IaaS 服务是对基础架构中各类基础设施的抽象，是通过软件对物理硬件资源进行的一种封装，这也是最基本的云计算服务类别。具体来说，IaaS 服务把计算资源、I/O 资源、内存容量、存储介质等资源进行资源池化，是针对底层硬件资源的一种托管方案，为企业、组织屏蔽了底层基础设施的管理细节 [6]。PaaS 服务作为中间层，把开发平台作为一种服务提供给用户，用户无需关注和维护底层基础架构，专注于业务应用即可 [7, 8]。PaaS 服务所面向的不是终端用户，而是应用软件开发人员，软件开发人员能够在 PaaS 平台上构建、部署和维护应用。SaaS 服务是一组面向最终用户的远程计算应用服务，第三方云资源提供商不仅在自家云资源上部署应用，并且还要管理和维护基础架构、应用服务，以及负责应用服务的升级。用户可以根据业务需求直接通过网络订阅和使用第三方云资源提供商的软件服务，完全不需要关注软件的安装和升级 [9, 10]。

随着虚拟化技术和云计算技术的蓬勃发展，互联网行业也迎来了巨大的变革。因为在传统 IaaS 服务中最基本的资源调度单位是虚拟机，而基于虚拟机的传统虚拟化技术存在着资源利用率低、体量较重、调度效率低等问题 [11]。在不断变化的

云计算环境中，Docker 凭借开源、轻量级、启动速度快、资源利用率高、方便移植等优势迅速成为当前企业部署云平台的首选底层虚拟化技术 [12, 13]。但是，Docker 自身只能提供基础的容器操作功能，所以在面对大规模数量的容器时，对这些容器进行优化管理、合理分配资源、统一编排就显得尤为重要。因此，一大批优秀的容器管理系统应运而生。目前容器管理系统主要有 Docker 公司所开发的 Swarm[14]、Google 公司所贡献的 Kubernetes[15, 16]、Apache 公司所开发的 Mesos[17]。其中，Kubernetes 是谷歌公司的开源项目，是谷歌公司内部大规模集群管理工具 Borg[18] 的开源版本。Kubernetes 作为目前最主流的容器编排工具，各大企业、公司已经将 Kubernetes 作为大规模容器生产场景下容器集群编排系统的事实标准。

根据 CNCF 基金会每年所发布的云原生报告，在 2016 年 CNCF 基金会所进行的调查中仅有 23% 的受访者表示正在使用 Kuebrnetes，但是在 2020 年所进行的云原生调查中有 83% 的企业、公司正在使用 Kubernetes。并且，Kubernetes 的市场占有率还在持续上升，由此可见 Kubernetes 的受欢迎程度以及多个企业、公司对 Kubernetes 的认可程度。Kubernetes 的发展历程如下图1.1所示，

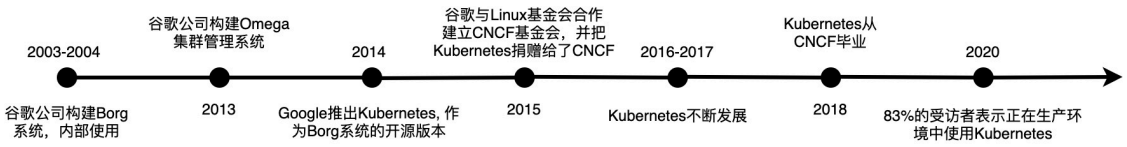



图 1.1: Kubernetes 发展时间线

## 1.2 国内外研究现状

云计算平台中的资源调度一直是很重要的研究方向，随着云原生技术的快速发展，容器作为新型云计算资源中的代表，容器管理和资源调度逐渐成为国内外学者的研究热点 [19, 20]。

### 1.2.1 云计算场景下的负载预测

在实际生产场景中，云资源请求序列与时间节点之间具有一定相关性 [21]，因此可以从时间序列预测分析的角度对云资源请求提前预测，在云平台中提前对资源进行调度分配以及编排部署，从而可减少用户请求的响应时间，提升资源利用率

[22, 23]。针对不同场景下的云资源请求，国内外很多学者提出了多种预测模型。

H Yuan 等人在分布式云数据服务场景下，针对云服务的工作负载提出了一种基于 Savitzky-Golay 滤波、小波分解与随机配置网络 SCN 的组合预测模型 SGW-S，从而对下一时间段的工作负载进行预测，经过实验验证，该预测模型 SGW-S 的预测效果比 ARIMA 和 BPNN 模型的效果要好 [24]。D Saxena 等人针对云计算资源弹性伸缩可能导致的问题构建了一个基于自适应神经网络模型的在线资源预测系统 OP-MLB，实验结果证实 OP-MLB 预测框架在提升资源利用率以及降低 SLA 风险上有较好的表现 [25]。Z Chen 针对现有方法不能对高维度、高度可变的云服务工作负载进行有效预测的问题，首先构建了用于从高维负载数据中抽取工作负载表征的顶部自动编码器 TSA，之后将 TSA 与 GRU 集成到 RNN 中，进而得到了一种基于深度学习的云服务负载预测算法 L-PAW。经验证，该负载预测算法 L-PAW 在不同预测步长下对不同类型的工作负载进行预测均能保证较好的负载预测自适应性和预测结果准确性 [26]。M Xu 等人针对回归模型和循环神经网络模型无法捕捉工作负载长期变化的问题，提出一种基于高效监督学习的深层神经网络模型 esDNN，对云计算任务负载进行预测。其中，esDNN 模型在进行预测的过程中，依次使用滑动窗口、深度学习模型与 GRU 模型进行写协作，实验结果表明，esDNN 模型能够显著降低均方误差，并且预测效果较好 [27]。RN Calheiros 等人考虑到包含 QoS、服务响应时间和服务拒绝率参数的服务请求，构建了基于 ARIMA 模型的服务预测模型，并且最终把服务 Qos 与资源利用率作为预测模型的评价指标 [28]。

在云计算场景中，各种应用对于云资源的需求并不是线性的，而是存在一定的波动，但是这种波动趋势会不断演变，在时间维度上存在一定相关性。此外，这种资源需求波动趋势还会被空间维拓的组织结构所影响，比如当集群中某个节点上资源、负载出现波动时，其他节点上的调度可能也会受影响 [29–31]。A Khan 等人使用一种共同聚类技术按照工作负载模式对这些虚拟机进行分组，并且使用隐马尔可夫模型确定虚拟机之间的相关性，进而预测工作负载模式的变化 [32]。Y Zhang 等人针对云计算场景中的 Qos 值进行预测，提出一种基于领域的预测方法

Cloud-Pred。Cloud-Pred 方法可以对不同云组件的 Qos 值进行个性化预测，提升了云计算场景下 Qos 预测的准确度 [33]。F Farahnakian 等人提出一种基于 BFD 算法改进的算法 UP-BFD，该算法为虚拟机在节约资源与降低 SLA 风险之间找到了一个平衡，并且 UP-BFD 算法通过使用 K 近邻回归算法，能够基于历史资源使用数据预测虚拟机上的 CPU 资源利用率 [34]。P Liljeberg 等人针对云数据中心的降低能耗问题，构建了一种负载预测方法 LiRCUP, 该方法基于线性回归技术通过 CPU 资源历史使用率预测当前值，可以保证降低 SLA 风险与节约能源 [35]。Z Zheng 等人提出一个对云服务 QoS 排名进行预测的框架 CloudRank，其能够通过用户的个人偏好或者用户过去的使用经验生成云服务排名，有助于辅助构建高质量云服务 [36]。Z Zhao 等人构建了一个基于 Savitzky-Golay 滤波与 LSTM 模型的组合预测模型，该组合模型使用梯度裁剪的方式消除梯度爆炸问题，对于任务负载的预测表现较好 [37]。

### 1.2.2 Kubernetes 调度

Kubernetes 中默认调度机制在调度时仅考虑节点上的 CPU 和内存资源，衡量指标比较单一，而事实上 Kubernetes 集群上业务在不断变化，较为单一的资源指标无法准确衡量集群整体负载。此外，Kubernetes 默认使用静态调度机制，调度器无法根据集群负载情况及时作出动态调整，不利于实现集群整体的负载均衡 [38–40]。因此，很多学者在 Kubernetes 的伸缩机制与调度机制上做了很多研究工作。Jakub Latusek 等人提出一种基于动态网络指标的新型 Kubernetes 调度策略 NetMARKS, 该策略使用 lstio 服务网格收集网络指标。此调度策略将应用响应时间减少为原来的 37%, 并且节省了 50% 的节点带宽 [41]。G Kaddoum 等人在边缘云场景中, 构建了一个 Kubernetes 边缘云调度框架 KaiS，其能根据集群规模与请求类型自学习调度策略, 提升了系统平均吞吐率, 降低了调度成本 [42]。C Jiang 等人在使用 Kubernetes 对 PoS blockchain 负载进行管理的场景中提出一种高效的调度策略，并且构建了离线最优方案与在线动态整合方案，工作节点的平均使用率降低了 13%[43]。在边缘

计算场景中微服务调度存在一定网络延迟，因此 Thomas Pusztai 等人针对微服务的调度问题构建了一个边缘感知 Kubernetes 调度器 Pogonip，其通过边缘感知启发式算法能够处理好微服务异步应用在调度时的节点选择问题 [44]。Y Yang 等人通过结合灰色预测方法与 LSTM 预测方法构建了一种优化 Kubernetes 调度机制的调度方法，经验证，该调度方法减少了集群中的碎片，并且提高了集群资源的利用率 [45]。T Menouer 等人提出一种基于 TOPSIS 算法的新型容器调度策略，其通过折中多种资源标准进而选择最合适的节点进行容器的调度 [46]。Z Zhong 等人设计了一个异质资源管理策略，其不仅支持异构作业的调度，也支持集群的弹性伸缩，并且还支持容器的动态重调度。实验验证，此策略节省了 23% 到 32% 的成本 [47]。C Zhiyong 等人基于 Kubernetes 默认调度机制提出一种新的调度方案，其在调度时把节点存储和网络带宽考虑在内，该调度策略使得集群负载更加均衡 [48]。M Wei 等人提出一种基于机器学习的容器调度策略，其通过预测下一个时间窗口的容器数量，提前进行容器数量的调整，该策略提高了系统性能、平衡了集群负载压力 [49]。SR Yang 等人构建了一种用于 Kubernetes 集群内部的资源监控机制，该机制对系统资源利用率、应用 Qos 指标进行收集，为调度决策提供参考 [50]。Xin xu[51] 等人构建了一个资源管理模型，其使用博弈论实现容器到服务器节点的最佳映射，经实验验证，该资源管理模型提升了资源利用率。Zhang Q, Zhani M.F[52] 等人基于容器需求与服务器节点资源之间的类似特征，使用 K 均值聚类方法对任务进行聚类操作。Gandhi Ad[53] 等人构建了一种动态管理策略 Autoscale，该策不仅能够保守地缩减服务器数量，还能自动保持应对突发负载的服务器数量，因此对服务请求规模和服务器效率均有较好的鲁棒性。

在国内，华为云开源了 CNCF 下唯一一个基于 Kubernetes 的面向高性能计算的容器批量计算平台——Volcano，其主要针对人工智能以及大数据作业等场景。Volcano 不仅提供 CPU、GPU 等资源在内的异构资源混合调度能力，还提出丰富的调度策略，比如成组调度、资源公平调度以及基于拓扑感知的调度策略等，在高性能计算场景下表现较好。此外，阿里云构建了针对不同任务负载的统一调度系统，

并且能够实现异构资源的精细化调度与批量计算调度等。其中实现了 GPU/CPU 拓扑感知调度、负载感知调度、CPU Burst 动态调度优化策略等，提升了集群资源利用率，并减少了服务延迟。

## 1.3 研究内容

本文以 Kubernetes v1.20.0 为研究对象, 并把 Kubernetes 中弹性伸缩机制与资源调度机制作为研究重点。本文对 Kubernetes 整体架构进行改进，在 Kubernetes 基础之上添加了监控模块、预测模块、弹性伸缩模块与动态调度模块。其中，针对 Kubernetes 中默认伸缩机制与调度机制所存在的问题，本文从以下三个方面对 Kubernetes 弹性伸缩机制与调度机制进行改进、优化。

**研究内容一: 基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略**

容器编排系统 Kubernetes 默认使用水平伸缩控制器 (Horizontal Pod Autoscaling, HPA) 来对 Kubernetes 基本部署单元 Pod 进行扩缩容操作。但是基于水平伸缩控制器 (HPA) 所实现的扩缩容机制是一种被动的响应式扩缩容机制，所以在面临突发性负载时，该扩缩容机制存在响应不及时的问题。并且，使用水平伸缩控制器 (HPA) 对 Pod 进行伸缩配置时，需要较高专业熟练度，难度较高。因此，本文在 Kubernetes 集群中构建了一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略，该策略能够在负载来临之前提前进行 Kubernetes 基本部署单元 Pod 的扩缩容。本文首先使用阿里云公开集群数据集 cluster-trace-v2018 [①] 对自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型进行训练，等到模型训练、调优过程完成后，将目标 Pod 的历史资源使用数据作为预测模型的输入，得到 ARIMA-GRU 组合预测模型的输出结果。然后，结合水平伸缩控制器 (HPA) 中所配置的最大副本数、最小副本数、容忍系数等字段计算目标 Pod 的期望副本数，通过 Kubernetes

[①]https://github.com/alibaba/clusterdata/blob/v2018/cluster-trace-v2018/trace_2018.md

REST API 进行 Pod 副本扩缩容。在 Kubernetes 中使用本文所提出的自定义弹性扩缩容策略时，其能够在集群高负载来临之前，提前扩容 Pod 副本，降低了用户请求的平均响应时间，提升了服务质量。在集群低负载来临之前，该策略提前缩容 Pod 副本，从而减少了集群中不必要的资源浪费。相比 Kubernetes 默认伸缩机制，本文所构建的弹性伸缩策略降低了集群中不必要的资源消耗以及提升了服务质量。

**研究内容二: 基于 Scheduling Framework 的扩展调度器以及一种基于集群负载均衡度的调度策略**

Kubernetes 调度机制就是把 Kubernetes 最小管理单元 Pod 运行在最合适的节点上。但是在对 Pod 进行调度时，Kubernetes 默认调度策略仅考量了 CPU 与内存两种资源，并没有考虑集群中其他类型的资源，比如网络带宽、网络 I/O 资源等，这可能导致没有被考虑到的其他类型资源出现性能瓶颈问题以及集群负载不均衡问题。本文基于 Scheduling Framework 构建 Kubernetes 扩展调度器，该扩展调度器在调度时，会考虑多种类型资源指标，比如 CPU、内存、网络 I/O、磁盘 I/O、磁盘容量等资源指标。并且，本文还提出一种基于集群负载均衡度的调度策略，在节点打分阶段，首先划分 Pod 类型，将 Pod 划分为计算型 Pod、I/O 型 Pod、网络型 Pod 等, 然后基于 Pod 类型为待筛选节点进行资源倾向性打分，如果节点上各种资源的均衡度分数越高，那么该节点的分数越高，该节点越适合调度。其中，使用层次分析法 (AHP) 计算节点上各类资源均衡度分数在节点分数中的占比。相比 Kubernetes 默认调度策略，本文所提出的基于集群负载均衡度的调度策略提升了集群负载均衡，保证了集群负载性能，并且有效缓解了节点上某种类型资源的性能瓶颈问题。

**研究内容三: 一种动态调整 Kubernetes 基本部署单元 Pod 资源限额的方案**

在 Kubernetes 集群中，开发者为保障应用稳定运行，会为 Kubernetes 最小管理单元 Pod 时请求较多资源。因此，通常情况下，Pod 在创建时对资源的请求量远大于其在实际运行时对资源的使用量。并且在运行过程中 Pod 对资源的使用量可能会不断变化，但是 Kubenetes 中默认的调度机制是一种静态调度，无法根据 Pod 的实际运行状况动态调整 Pod 的资源参数。这会导致 Kubenetes 集群为 Pod 预留了

很多可能用不到的资源，存在一定的资源浪费。因此本文提出一种基于 Kubernetes 最小管理单元 Pod 的实际运行状态而动态调整 Pod 资源限额的方案，该方案能够在不重新构建 Pod 的情况下实现 Pod 资源限额的动态调整。该方案运行时，首先查询 Kubernetes 中基本部署单位 Pod 的历史资源使用量，进而计算 Pod 在未来一段时间内的资源需求。然后使用 Linux 控制群组 (Cgroup) 直接对运行中 Pod 进行调整，调整后的 Pod 资源配置立即生效。相比 Kubernetes 默认静态调度机制，本文所提出的动态调整 Kubernetes 基本部署单元 Pod 资源限额的方案有效提升了集群资源的利用率，并且减少了 Pod 预留资源的浪费。

## 1.4 论文主要贡献

针对 Kubernetes 中默认弹性伸缩机制与调度机制所存在的问题，本文提出相应的优化方案并进行实施、验证，主要贡献如下:

(1) **基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略:** 本文构建了一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合负载预测模型和水平伸缩控制器 (HPA) 配置的弹性伸缩策略。相比 Kubernetes 默认伸缩机制所存在的扩缩容响应不及时的问题，本文所提出的弹性伸缩策略能够在集群负载来临之前提前对 Kubernetes 基本部署单位 Pod 进行扩缩容操作，降低了集群资源不必要的消耗以及减少了服务响应时间、提升了服务质量。

(2) **基于 Scheduling Framework 的扩展调度器以及一种基于集群负载均衡度的调度策略:** 本文基于 Scheduling Framework 构建了 Kubernetes 扩展调度器，并提出一种基于集群负载均衡度的调度策略。相比 Kubernetes 默认调度策略可能会导致集群负载不均衡，本文所提出的自定义调度策略在调度时考虑了集群中多种资源指标以及集群资源均衡度，更能实现集群负载均衡，提升了集群负载性能。

(3) **一种动态调整 Kubernetes 基本部署单元 Pod 资源限额的方案:** 本文基于 Kubernetes 基本部署单元 Pod 的历史资源使用量与运行状况提出一种动态调整 Pod 资源限额的方案，而 Kubernetes 默认调度方案是一种静态调度方案，在 Pod 运行期间 Kubenetes 默认不能根据 Pod 的实际运行状况对 Pod 进行动态调整。因此，本文所提出的动态调整 Pod 资源限额方案提升了集群资源的利用率，减少了 Pod 预留资源的浪费。

## 1.5 论文章节安排

本文一共分为六章，接下来介绍每一章节的主要工作内容:

第一章，绪论。首先对本文研究背景进行说明，接着分析国内外学者、研究人员在云计算场景下负载预测与调度领域中的研究现状，然后介绍本文研究内容以及工作贡献，最后阐述此论文章节安排。

第二章，背景知识介绍与说明。首先阐述 Kubernetes 功能特点、基本概念，然后介绍 Kubernete 基于水平伸缩控制器 (HPA) 所实现的默认调度机制，并分析默认调度机制存在的不足。接下来着重阐述 Kubernetes 默认调度策略以及其所包含的相关调度算法。最后介绍目前的时间序列预测模型分类，并详细说明自回归差分移动平均 (ARIMA) 模型与门控循环单元 (GRU) 模型。

第三章，基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略设计与实现。本章构建了一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略。针对 Kubenetes 默认伸缩机制与调度机制所存在的不足，本章首先介绍改进后的 Kubernetes 架构设计，之后说明架构设计中的各个模块，阐述监控模块中的组件构成与组件安装。然后分别介绍预测模块中自回归差分移动平均 (ARIMA) 模型与门控循环单元 (GRU) 模型的构建流程，详细介绍 ARIMA-GRU 组合预测模型的训练流程以及实际预测流程，最后详尽说明基于 ARIMA-GRU 组合预测模型与 HPA 配置的弹性伸缩策略的设计思路与执行流程。

第四章，Kubernetes 动态调度策略设计与实现。首先介绍 Kubernetes 调度器的工作流程，然后说明目前 Kubernetes 的扩展调度器方案。本文详细阐述了基于 Scheduling Framework 构建 Kubernetes 扩展调度器的流程，并详细说明本文所提出的基于集群负载均衡度的调度策略。最后，详细介绍本文所提出的动态调整 Kubernetes 基本部署单位 Pod 资源限额的方案，并对该方案的落地实践进行说明。

第五章, 实验结果与分析。本章介绍本文实验环境的节点配置、软件环境，对本文所构建的负载预测模型进行对比实验，并统计和分析实验结果。并且，针对本文所提出的弹性伸缩策略和动态调度策略分别进行实验、验证，之后分析实验结果，进而验证弹性伸缩策略和动态调度策略的可行性与有效性。

第六章，总结与展望。首先总结本文所进行的研究工作与工作内容，然后分析目前研究工作以及实践环节的不足，展望后续的研究与优化方向。

# 第二章　背景知识介绍与说明

## 2.1　Kubernetes 介绍与基本概念

Kubernetes 是谷歌公司所开源的、应用于云计算场景中的容器集群管理系统，起源于谷歌公司内部已经运行多年的大规模统一容器管理系统 Borg，之后 Kubernetes 被捐赠给 CNCF 基金会。目前 Kubernetes 在容器应用的部署、迁移与管理以及容器服务的弹性伸缩方面有较多 [54, 55]。Kubernetes 是一个支持移植和扩展的开源容器编排系统，允许开发者使用声明式配置文件构建、管理容器应用。Kubernetes 能够为容器应用提供统一编排管理、资源监控、资源管理调度以及服务负载等功能，具有自动化程度高、可移植性强和扩展性良好的特点 [56]。

目前 Kubernetes 已经是被广泛使用与认可的容器编排系统，包含以下功能:

- 自我恢复，在集群节点出现问题的时候，Kubernetes 能够重启、替换或者重新部署问题节点上的容器，把问题节点上的容器迁移到其他可用节点上，保证为容器定义好的预期副本数量。当容器健康检查失败后，Kubernetes 关闭容器并重新创建容器。

- 弹性伸缩，Kubernetes 支持通过 Kubectl 命令工具和相关 Kubernetes 管理界面通过设置 CPU、内存资源阈值实现 Kubernetes 最小管理单元 Pod 副本的扩缩容。弹性伸缩机制既能确在访问高峰时，保证应用服务的可用性；也能在业务低谷时，删除 Pod 应用副本，回收资源，减少不必要的资源浪费。

- 自动发布和回滚，Kubernetes 使用滚动更新策略更新 Kubernetes 最小管理单元 Pod，支持灰度更新 Pod 应用或者 Pod 应用相关的配置信息。Kubernetes 不会在同一时刻删除 Pod 应用的所有副本，并且会持续监控应用更新。如果 Pod 副本更新时出现问题，则马上进行回滚，确保正在进行升级的应用 Pod 能正常提供服务。

- 存储编排，在 Kubernetes 集群中开发者可以根据业务场景使用多种类型的存储系统，可以将 Kubernetes 最小管理单元 Pod 的数据卷挂载到本地存储、公有云存储或者网络存储。此外，Kubernetes 也允许开发者基于应用场景灵活组合使用多种类型存储系统。

- 服务发现和负载均衡，Kubernetes 基于 CoreDNS 组件 (也可以称为 KubeDNS 组件) 在集群内部内置了服务发现功能，并且为集群中每个 Service 分配 DNS 名称，允许集群内客户端直接通过该 DNS 名称向对应 Service 发送请求。此外，基于 iptables 和 ipvs，Service 内置了负载均衡功能。

Kubernetes 包含许多资源对象，大多数资源对象能够通过 REST API 来执行 CRUD 操作，操作结果会被持久化存储到 etcd 中 [57]。接下来，对 Kubernetes 中相关资源对象进行介绍。

1. Pod: 运行在 Kubernetes 集群中的 Node 节点上，是集群管理和调度的最小单位，或者说是最小粒度工作单元。并且在每一个 Pod 中存在许多容器，相同 Pod 中所有容器能够共同使用 Pod 中的网络和存储等资源。

2. Label: 这是一个 Key/Value 键值对，其可以用在多种类型资源对象中，比如能够为 Node、Service、RC、Pod 等资源对象添加 Label。同一个 Label 允许在多个资源对象中使用，并且一个资源对象也可以包含多个 Label，Kubernetes 使用 Label Selector 查询和筛选对应的资源对象。

3. Replication Controller(RC): 用于管理 Pod 副本，确保 Kubernetes 集群包含指定数量的 Pod 副本。比如，如果 Kubernetes 集群中 Pod 副本数量比指定的 Pod 副本数量要多，则 RC 回收多余 Pod 副本；如果集群中 Pod 副本数量比指定的 Pod 副本数量要少，那么 RC 构建新的 Pod 副本。RC 是 Kubernetes 集群进行弹性伸缩和滚动升级等业务的关键。

4. Deployment: 相当于 Replication Controller 的升级版本，对容器管理有着更完善的功能。

5. Service: 定义 Pod 的业务逻辑集合以及该逻辑集合的访问策略，对真实服务进行抽象。Service 为一组 Pod 副本提供单一的、稳定的名称和服务，Service 使得这一组 Pod 副本可以通过统一服务入口被访问。其中，用户对 Service 的访问流量被转发给 Pod，由 Pod 具体处理这些访问请求，并且 Service 为这些 Pod 的访问配置了负载均衡。

6. HPA(HorizontalPodAutoscaler): 通过监控分析 Deployment 或者 StatefulSet 等资源集合中 Pod 的负载情况来对 Pod 副本进行伸缩，实现应用 Pod 副本数量针对某些指标的自适应。

虽然 Kubernetes 基于水平伸缩控制器 (HPA) 可以实现 Pod 副本的扩缩容，但是该扩缩容机制存在一定滞后性。因此本文考虑构建基于负载预测模型的弹性伸缩策略，一定程度上可以保证集群中服务的响应速度。

## 2.2 Kubernetes 弹性伸缩技术

Kubernetes 集群中弹性伸缩机制是指 Kubernetes 中最小管理单位 Pod 在运行过程中，Kubernetes 能够根据负载请求的变化动态调整 Pod 副本数量，从而减少业务等待时间，节约集群资源。Kubernetes 中最小管理单位 Pod 的扩缩容操作默认是由水平伸缩控制器 (Horizontal Pod Autoscaling, HPA) 来完成的，在水平伸缩控制器的配置文件中可以配置 Statefulset、RC 等资源对象所能够使用的资源阈值，以及定义 Kubernetes 最小管理单位 Pod 的最小副本数、最大副本数等。其中，水平伸缩控制器 (HPA) 对应着 Kubernetes 中的某个 Controller, 该 Controller 会间隔执行，并且循环检查 HPA 中所定义的指标是否能够触发伸缩条件。一旦 Statefulset、RC 等资源对象触发了水平伸缩控制器中所定义的触发伸缩条件，Controller 会向 Kubernetes API Server 发送扩缩容请求，进而对 Statefulset、RC 等资源对象集合中最小管理单位 Pod 执行动态伸缩操作 [58, 59]。

但是 Kubernetes 默认伸缩机制是基于水平伸缩控制器 (HPA) 所实现的，这是一种被动的响应式扩缩容机制，并且存在一定的滞后性。当集群负载突然发生较大

变化时，水平伸缩控制器 (HPA) 可能由于扩容不及时，从而导致最小管理单位 Pod 所提供的服务出现问题, 影响服务质量。并且，在集群资源紧张的情况下，可能因为水平伸缩控制器 (HPA) 缩容响应不及时，导致集群中一些资源被空占用，致使相关最小管理单位 Pod 不能成功调度。此外，不同应用 Pod 对资源需求的差异性较大，需要开发者具有很高的专业熟练度，才能在水平伸缩控制器 (HPA) 配置中为不同 Pod 设置相对合理的弹性伸缩指标，难度太大。

## 2.3 Kubernetes 资源调度策略

Kubernetes Scheduler 是 Kubernetes 中的默认调度器, Kubernetes 通过 Scheduler 组件进行 Kubenetes 最小管理单位 Pod 的调度，Scheduler 组件负责为 Pod 寻找合适调度的 Node 节点。在 Kubernetes 集群中，客户端通过 Kubelete 或者 Kubenetes REST API 发送请求后，API Server 接收到请求信息，Controller Manager 将资源对象配置信息写入到 etcd 中。Scheduler 通过 List&Watch 机制监控 API Server 资源变化，进而进行任务调度。其中，具体调度流程是 Scheduler 不断访问 API Server 获取还没有进行调度的 Pod 以及 Node 节点集合，通过调度策略选择一个合适调度的节点与待调度 Pod 进行绑定，并通过节点上的 Kubelet 向 API Server 发送请求把目标 Pod 与该节点的绑定信息存储到 etcd 组件中 [60]。Pod 调度过程主要分为两个阶段和三个过程，分别是调度阶段 (包含预选过程和优选过程) 和绑定阶段 (绑定过程)。

Kubernetes 调度最小部署单位 Pod 的核心机制是两个控制循环，第一个控制循环负责调度队列的管理，当一个 Pod 被声明后等待调度时，调度器就会把这个 Pod 添加到待调度 Pod 队列中。待调度 Pod 队列的结构是一个优先队列, 该队列中 Pod 按照优先级排序。第二个调度控制循环是管理 Pod 具体调度的控制循环，主要作用是为等待调度的 Pod 绑定最合适的 Node 节点。

在预选 (Predicate) 过程中，主要是根据预选策略把不符合条件的节点过滤掉，得到符合条件的节点集合，从而完成节点的预选。

Kubernetes 中内置的预选策略有以下几种:

1、PodFitsHost: 如果待调度 Pod 的配置文件中特别定义了 NodeName 字段, 那么在进行节点筛选的时候会对主机节点名称与 Pod 配置中定义的 NodeName 进行对比。即检查两者是否匹配,检查节点名称与 Pod 资源对象配置文件中”spec.nodeName”字段值是否相同。

2、PodFitsResource: 检查节点上空闲资源是否能够满足当前待调度 Pod 的资源请求量，一般检查节点上的 CPU 资源和内存资源。

3、CheckNodeCondition: 检查在节点磁盘、网络设备没准备好或者不可用时能否将 Pod 调度到这个节点上。

4、NoDiskConflict: 查询待调度 Pod 与节点上存在的 Pod 在使用磁盘卷时是否会有冲突。比如，在同一节点上不允许两个不同 Pod 挂载相同的 GCEPersistentDisk 或者 AWSElasticBlockStore。

Kubernetes 中内置的优选策略有以下几种,

- ImageLocalityPriority: 如果节点上已经存在了待调度 Pod 所需要的镜像，镜像体积越大，则该节点的分数越高，即基于节点上是否存在待调度 Pod 所需要的镜像以及镜像体积进行打分。

- LeastRequestedPriority: 计算目前节点上 CPU 资源与内存资源的空闲率, 两类资源空闲率越高，该节点分数越高，即节点分数由节点上 CPU 与内存空闲资源与节点总资源的比值决定。计算公式为,

$$score = \frac{cpu\left(\frac{(Totalcapacity_i - \sum Need_i)*10}{Totalcapacity_i}\right) + memory\left(\frac{(Totalcapacity_i - \sum Need_i)*10}{Totalcapacity_i}\right)}{2} \quad (2.1)$$

- BalancedResourceAllocation: 节点上 CPU 资源和内存资源的使用率越接近, 那么节点资源越均衡，则节点的分数越高，该策略会把待调度 Pod 尽量部署到资源使用比较平衡的机器上。但是该预选策略不可以独立使用，必须和 LeastRequestedPriority 策略组合使用。计算公式为,

$$score = int\left(10 - math.Abs\left(\frac{totalMilliCPU}{nodeCpuCapacity} - \frac{totalMemory}{nodeMemoryCapacity}\right) * 10\right) \quad (2.2)$$

- SelectorSpreadPriority: 为保证不同 Pod 副本之间实现容灾，同一个 Service、Deployment 或者 RC 等资源对象中的多个 Pod 副本应该尽可能分散在不同节点上。检查节点上有无与待调度 Pod 同属于相同 Service 或者控制器的 Pod，该节点上此类 Pod 越少则节点分数越高。

尽管 Kubernetes 调度器内置了丰富的调度策略，但是默认调度策略仅仅针对 CPU 资源和内存资源进行节点选择，所考量的资源指标太过单一，在复杂应用场景下不能实现集群资源的均衡使用。因此本文构建了 Kubernetes 扩展调度器，在进行 Pod 调度时综合考虑 Node 节点上 CPU、内存、网络 I/O、磁盘 I/O、磁盘空间这五种资源，保证调度完成后集群资源的均衡性。此外，Kubernetes 调度默认调度机制是静态调度，没有考虑到集群中应用 Pod 的动态变化问题。因为集群中不断进行 Pod 的创建和删除，不同节点上会出现较多资源碎片。所以，集群运行较久之后可能会出现集群负载、资源非常不平衡的情况，因此本文提出了一种动态调整 Pod 资源限额的方案，能够实现集群资源的动态均衡效果。

## 2.4 基于时间序列的预测技术概述

负载预测是对资源进行优化分配的基础 [61]，通过收集应用负载的历史观测值并将其作为原始时间序列进行预测，可以预测应用负载在未来一定时间内的变化趋势，从而能为之后的资源调度提供决策支持 [62, 63]。在云计算场景下，运行在 Kubernetes 集群中的每个应用在每个时刻都使用着节点上各种资源，其中包含 CPU、内存、网络 I/O、磁盘 I/O、磁盘容量等资源。将 Kubernetes 中最小管理单元 Pod 对资源的请求量数据与其对资源的实际使用量数据按照时间节点进行收集，这就构成了集群负载时间数据序列。选择并建立时间序列预测模型之后，该时间数据序列就可以作为预测模型的输入，之后就可以预测在未来一段时间内 Pod 对节点资源的请求量与使用量。这样就能够在 Kubernetes 集群负载来临之前，提前扩缩容 Pod 副本。

其中，时间序列数据是每隔一段时间进行收集的数据，其在一定程度上可以显

现出数据随时间的变化趋势、变化程度。时间序列数据按照时间节点进行排序，时间序列预测其实就是基于历史数据使用合适的预测模型对未来一段时间内的可能值进行预测 [64, 65]。在经济学中，时间序列数据可以是国内生产总值、失业率和消费者物价指数等；在社会学中，时间序列数据可以是出生率、人口增长率等。

现有时间预测工具可以分为两大类，分别是基于统计学的传统时间序列预测模型和基于机器学习的时间序列预测模型。对于传统的时间序列预测模型，首先明确时间序列参数模型，然后对预测模型参数进行计算求解，最后根据参数确定完成后的预测模型进行预测。传统时间序列预测模型主要有：均值回归模型、隐马尔科夫模型和指数平滑模型等。这类传统时间序列预测模型能够对某一预测对象进行预测，具有复杂度低、计算速度快的特点，但是传统时间序列预测模型的学习自由度与泛化性差一些。对于基于机器学习的时间序列预测模型，可分为树模型和神经网络模型两种类型。在使用树模型进行时间序列的预测时，主要使用 xgboost 或者 lightgbm 方法，其核心点在于如何从历史数据中抽取出特征。在使用神经网络模型进行时间序列的预测时，使用的方法主要有 CNN、RNN、LSTM 和 GRU 等方法。相比于传统的时间序列预测模型，神经网络时间序列预测模型的训练要求大量数据作为输入。这些机器学习时间序列模型预测精度较高，学习自由度高，但是计算复杂度也更高一些 [66–68]。

### 2.4.1 自回归差分移动平均模型 (ARIMA) 模型

时间序列是将所要预测对象的历史数据按照时间节点进行排序得到的一组数值序列，时间序列预测方法就是根据预测对象的这些历史观测值，对其进行统计分析从而预测出未来一段时间内的值。ARIMA 模型属于传统的时间序列预测方法，是最常见的时间序列预测分析方法。ARIMA 模型的全称为自回归差分移动平均模型，因该模型是一种自回归模型，因此 ARIMA 模型只需要自变量就可以预测后续的值 [28]。其中，ARIMA 模型要求时间序列本身就是平稳的，或者经过差分运算后变为平稳序列。ARIMA 模型包含 AR(p) 模型、MA(q) 模型以及 ARMA(p, q) 模型等特殊形式的 ARIMA 模型，与 ARIMA 模型相关的时间序列模型有以下几种：

**1、AR(p) 自回归模型**

AR 模型称为自回归模型 (Auto Regressive model)，其被定义为一个 p 阶自回归模型。它描述了当前数据和历史数据的关系，其通过时间序列中历史时间点的线形组合加上白噪声来预测当前时间点，该模型基于目标变量的历史观测数据对变量自身执行预测操作。

一般的 p 阶自回归模型 AR 即 AR(p) 模型，如下所示:

$$X_t = \phi_0 + \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} + \varepsilon_t \tag{2.3}$$

公式中，称 $\{\phi\}$ 是 AR(p) 模型中的自回归系数，是当前值与历史值之间的自相关系数；$\varepsilon_t$ 是该时间序列的白噪声值，可理解为是时间序列数值的波动，这些波动的均值为 0；p 为自回归阶数，表示历史观测值需要 p 步才能确定当前值。

当 $\phi_0 = 0$ 时，此时的 AR(p) 模型就被称为中心化 AR(p) 模型。非中心化 AR(p) 模型可以通过下式转化为中心化 AR(p) 模型，

$$y_t = x_t - \frac{a_0}{1 - \sum_{i=1}^{p} a_i} \tag{2.4}$$

此外, AR(p) 模型对 ACF(自相关函数) 拖尾 (衰减趋于 0), 对 PACF(偏自相关函数) 截尾。所谓拖尾是指 ACF、PACF 不满足在某阶后均为 0，截尾是指 ACF、PACF 满足在某阶后均为 0。

**2、MA(q) 移动平均模型**

MA 模型称为移动平均模型 (moving average model), 它并不基于历史时序数据进行回归，而是基于历史误差值 (即历史的残差项) 构建一个类似回归的预测模型, 该预测模型用于预测当前值。相比于 AR(p) 模型是历史时序值的线性组合，MA(q) 则是历史白噪声的线形组合。

MA(q) 模型的 q 阶公式为,

$$y_t = c + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q} \tag{2.5}$$

其中，$y_t$ 可以被认为是历史预测误差值的加权移动平均值；c 是时间序列均值；$\{\theta\}$ 是 MA(q) 模型中的移动平均系数；$\{\varepsilon\}$ 是白噪声残差序列；q 为移动平均阶数, 表示历史残差项需要走 q 步才能确定当前值。

当 c=0 时，MA(q) 模型是中心化 MA(q) 模型。

此外，MA(q) 模型对 ACF(自相关函数) 截尾，对 PACF(偏自相关函数) 拖尾。

**3、ARMA(p , q) 自回归移动平均模型**

ARMA(p , q) 模型是把 AR(p) 模型与 MA(q) 模型进行了组合，该模型是历史时序值与历史白噪声数据的线性组合。该模型的表达公式为，

$$r_t = \phi_0 + \sum_{i=1}^{p} \phi_i r_{t-i} + a_t + \sum_{j=1}^{q} \theta_j a_{t-j} \tag{2.6}$$

公式中，p 为 AR(p) 自回归模型的阶数；q 为 MA(q) 移动平均模型的阶数；a 为白噪声残差序列。

当 $\phi_0 = 0$ 且 $a_0 = 0$ 时，ARMA(p , q) 模型就变成了中心化 ARMA(p , q) 模型。

此外，ARMA(p , q) 模型对 ACF 和 PACF 均拖尾。

**4、ARIMA(p , d , q) 自回归差分移动平均模型**

对于以上所提到的三个模型，它们的运用对象都是平稳的时间序列，而在实际场景下，某些时间序列不是平稳的。因此在 ARMA(p , q) 自回归移动平均模型的基础上引入了差分操作，这种把 ARMA(p,q) 模型和差分操作组合之后的模型就是 ARIMA(p,d,q) 模型，ARIMA(p,d,q) 模型的运用对象是差分后平稳的时间序列。

差分操作能够把非平稳时间序列转变为平稳时间序列，其可以计算相邻历史观测值之间的差值。简单地说，也可以把 ARIMA(p , d , q) 模型看作是对非平稳时间序列进行 d 次差分操作与 ARMA(p , q) 模型的组合。

$$\begin{cases} \varphi(B)\nabla^d x_t = \theta(B)\varepsilon_t \\ E(\varepsilon_t) = 0, Var(\varepsilon_t) = \sigma_\varepsilon^2, E(\varepsilon_t \varepsilon_s) = 0, s \neq t \\ E(x_s \varepsilon_t) = 0, \forall s < t \end{cases} \tag{2.7}$$

公式中，$\{\varepsilon\}$ 是白噪声差值序列，期望为 0；其中，$\nabla^d = (1-B)^d, \varphi(B) = 1 - \varphi_1 B - \cdots - \varphi_p B^p$ 是 ARMA(p, d) 模型的自回归部分；$\theta(B) = 1 - \theta_1 B - \cdots - \theta_q B^q$ 是 ARMA(p, d) 模型的移动平均部分。

d 阶差分后的时间序列为 $\nabla^d x_t = \sum_{j=0}^{d} (-1)^j C_d^j x_{t-j}$ , 其中 $C_d^j = \frac{d!}{i!(d-i)!}$ , 所以 ARIMA(p, d, q) 模型也可以表示为 $\nabla^d x_t = \frac{\theta(B)}{\varphi(B)} \varepsilon_t$。

该公式表示 ARIMA(p, d ,q) 模型就是基于 ARMA(p,q) 模型对每个原始时间序列执行差分操作，把非平稳时间序列转变为平稳时间序列。

特别地，当 d 取 0 值时，即没有进行差分操作，则此时的输入时间序列即为平稳时间序列，ARIMA(p, d ,q) 模型转化为 ARMA(p , d) 自回归移动平均模型；

当 d 取 0 值并且 p 也取 0 值时，ARIMA(p, d ,q) 模型转化为 MA(p , d) 移动平均模型；

当 d 取 0 值并且 q 也取 0 值时，ARIMA(p, d ,q) 模型转化为 AR(p) 自回归模型。

除了以上的表示方法，也可以将 ARIMA(p, d, q) 模型表示为，

$$ARIMA(p,d,q) = AR(p) + Difference(d) + MA(q) \tag{2.8}$$

公式中，AR(p) 项表示自回归模型，p 代表 AR(p) 模型中的阶数；MA(q) 项表示移动平均模型，q 代表 MA(q) 模型中的阶数；d 表示对非平稳原始时间序列执行多少次差分才能使其平稳化。因此，可以将 ARIMA(p, d ,q) 模型看作是已经做了 d 阶差分处理后的 ARMA(p, d) 自回归移动平均模型。

ARIMA(p, d ,q) 时间序列模型判别子模型的基本准则如下表2.1所示，

表 2.1: ARIMA 时间序列模型的基本准则表

| **模型** | **ACF** | **PACF** |
|---|---|---|
| AR(p) | 拖尾 | p 阶截尾 |
| MR(q) | q 阶截尾 | 拖尾 |
| ARIMA(p,q) | 拖尾 | 拖尾 |

### 2.4.2 门控循环单元 (GRU) 模型

循环神经网络 (RNN) 模型的设计目的就是为了对时序列数据进行处理，这也是它与传统神经网络模型最大的不同。由于引入了状态变量，所以 RNN 模型可以保存历史数据信息，历史数据信息对后边结点的结果输出有一定的影响。也可以说，RNN 模型隐藏层中的各个结点是连接起来的，并不是孤立的，所以隐藏层的输入不单单包含输入层的输出状态，上一时刻隐藏层的输出状态也可能会作为此

时隐藏层的输入 [69, 70]。但是 RNN 模型存在着长期依赖的问题，因为在反向传播过程中时间序列太长会导致循环神经网络模型出现梯度消失或者梯度爆炸的问题，因此基本的循环神经网络 RNN 模型在为长距离的依赖关系进行建模时比较困难，其根本原因还是记忆的时间序列太长了 [71, 72]。

长短期记忆神经网络模型 LSTM 属于一类比较特殊的 RNN 模型，是基础 RNN 模型的变体。与基础 RNN 模型相比，LSTM 模型在捕捉较长时间序列之间的语义关联上表现更好，从而对梯度消失以及梯度爆炸的情况能有一定的缓解作用 [73]。基于 RNN 模型，LSTM 模型中增加了“门控”机制，它们是输入门、输出门以及遗忘门。这种机制支持有选择地保存或者删除信息，所以相比 RNN 模型，LSTM 模型能较好处理长期依赖的问题。但是，由于 LSTM 模型内部结构复杂，所以在同等算力下该模型的计算效率甚至比基础 RNN 模型还要低 [74]。

而门控循环单元 (GRU) 同样也是基础循环神经网络模型的变体，GRU 模型不仅能像 LSTM 模型一样对基础 RNN 模型中所存在的梯度消失以及梯度爆炸情况有一定缓解作用 [75, 76]。而且，它的结构构造比 LSTM 模型简单，复杂度相比 LSTM 模型也更低，因此 GRU 模型在计算时效率更高，并且训练模型用时更短，同时它所要的计算资源更少。

## 2.5 本章小结

本章首先介绍 Kubernetes 的功能、特性以及资源对象，接下说明 Kubernetes 中默认的弹性伸缩机制及其工作流程，并分析默认弹性伸缩机制所存在的不足。本章还分析了 Pod 的调度流程，研究 Kubernetes 默认调度机制中的节点预选策略和优选策略，以及分析 Kubernetes 默认调度机制目前所存在的问题。并且针对 Kubernetes 默认调度机制所存在的问题，提出改进意见。本章重点介绍了自回归差分移动平均模型 (ARIMA) 模型以及门控循环单元 (GRU) 模型，比较分析了 LSTM 模型与 GRU 模型之间的差异。

# 第三章　基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略设计与实现

Kubernetes 集群中默认伸缩机制与调度机制均存在一些不足之处，其中，Kubernetes 默认伸缩机制是基于水平伸缩控制器 (HPA) 所实现的一种被动的响应式扩缩容机制，在某些场景下可能存在扩缩容响应不及时的问题，会影响服务质量、浪费集群资源。此外，Kubernetes 默认调度机制在进行调度时，仅考虑 CPU 资源与内存资源，并没有考虑集群中其他类型资源，所以调度之后可能导致集群负载不均衡。此外，Kubernetes 默认调度机制是一种静态调度机制，默认不能根据 Kubernetes 最小管理单元 Pod 的运行状况对其进行动态调整，这可能导致 Kubernetes 为最小管理单元 Pod 预留资源过多，可能存在一定资源浪费。因此本文针对 Kubernetes 集群中默认伸缩机制与调度机制所存在的问题进行优化和改进，在现有 Kubernetes 的基础上增加了监控模块、预测模块、弹性伸缩模块和资源调度模块。

本章主要对 Kubernetes 整体架构设计进行改进，自构建了多个模块，并对监控模块、预测模块与弹性伸缩模块的设计思路与构建流程进行了介绍。首先介绍 Kubernetes 集群的整体架构设计，并对说明各个模块的作用；之后介绍监控模块中的各个组件，以及监控组件的安装等。本章在预测模块中构建了一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 的组合预测模型，并详细阐述了该组合预测模型的构建流程、预测流程以及组合模型的性能评价指标。最后在弹性伸缩模块中提出一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合模型预测结果与 Kubernetes 水平伸缩控制器 (HPA) 配置的弹性伸缩策略，并详细说明了该弹性伸缩策略的构建流程与工作流程。资源调度模块的相关内容会在下一章进行具体介绍。

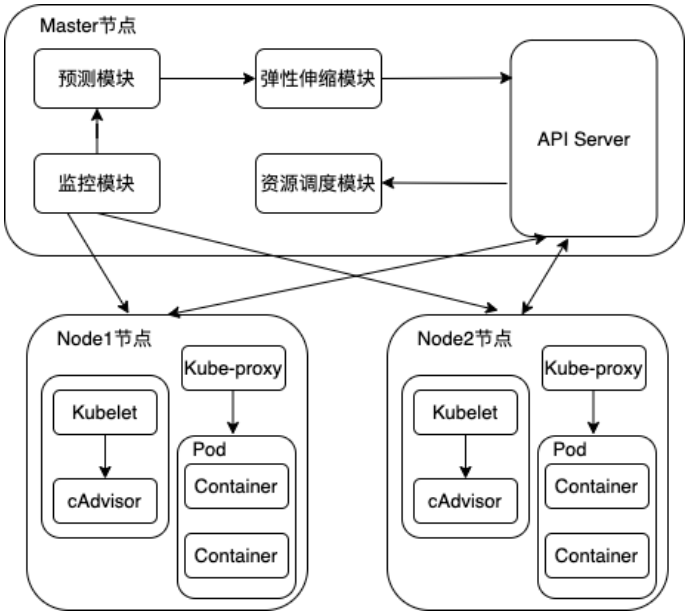



图 3.1: 系统架构图

## 3.1 Kubernetes 整体架构设计

为实现基于 Kubernetes 集群负载预测的 Kubernetes 弹性伸缩机制以及动态资源调度机制，本文在 Kubernetes 原有架构的基础上构建了监控模块、预测模块、弹性伸缩模块与资源调度模块，改进后的架构设计如上图3.1所示。接下来根据上图的架构设计，对改进后的 Kubernetes 架构中的相关模块进行解释说明，

1、监控模块: 监控整个 Kubernetes 集群中所有资源对象的运行状态，在集群中每一个 Node 节点上以容器的形式部署运行 cAdvisor, cAdvisor 容器采集 Node 节点与 Pod 的资源信息，包含 CPU、内存、网络 I/O、磁盘 I/O、磁盘容量等信息。将 cAdvisor 容器所收集到的数据整合完成后存储到 Prometheus 中，之后在预测模块与调度模块中使用到这些资源指标信息时直接使用 PromQL 语言对 Prometheus 进行查询即可获取相关数据信息。为了可视化集群整体负载信息，将 Prometheus 作为数据源对接到可视化工具 Grafana 中，这样就可以通过 Grafana 中监控面板方便、直观地查看集群状态。

2、预测模块: 使用 PromQL 从 Prometheus 查询 Node 节以及 Pod 的历史负载数据与资源使用量数据，将该部分数据作为预测模型的输入。然后基于 ARIMA-GRU 组合预测模型对未来一定时间内的集群负载进行预测，此预测模块的预测结果以

API 的形式进行封装并提供给弹性伸缩模块，为之后扩缩容操作提供决策参考。

3、弹性伸缩模块: 基于预测模块所预测出的集群负载数据计算 Pod 副本预测数量, 将根据预测数据所计算出的 Pod 副本的预测数量与水平伸缩控制器 (HPA) 设置的阈值等字段进行组合计算，最终确定 Pod 副本的期望数量。之后通过 Kubernetes REST API 对 Deployment 等资源对象进行 Patch 操作，从而对 Pod 副本实现扩缩容操作。其中，Pod 副本扩缩容操作需要服从水平伸缩控制器 (HPA) 中已经设置的容器资源阈值以及 Pod 副本扩缩容速率等规则。

4、资源调度模块: 监听 API Server，当弹性伸缩模块中的 Pod 副本进行扩缩容时或者新请求中 Pod 需要调度时，资源调度模块会为 Pod 选择合适的 Node 节点进行调度。在该模块中，基于 Scheduling Framework 构建自定义调度器，并且设计了基于资源负载均衡度的自定义调度策略，在 Kubernetes 默认调度策略的基础上进行了扩展和优化。

上述 Kubernetes 架构设计图3.1中各模块之间的交互关系如下图3.2所示,

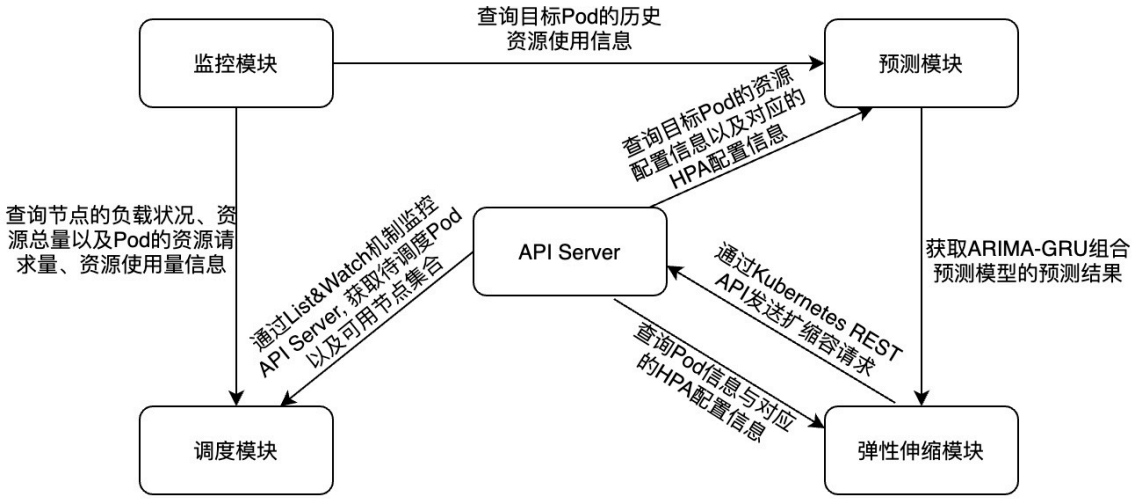



图 3.2: Kubernetes 架构中各模块交互图

## 3.2 监控模块

监控模块用来监控 Kubernetes 集群中的负载情况以及各种资源对象的资源使用情况,所收集的监控数据存储在 Prometheus 中,其他模块可以使用 PromQL 语言组合查询监控数据。具体来说, 监控模块主要监控并收集各个节点的资源状态、负载情

况以及节点上各种资源对象的资源使用量等数据。本文使用 cAdvisor、Prometheus、Grafana 构建 Kubernetes 集群监控平台，设计如下图3.3,

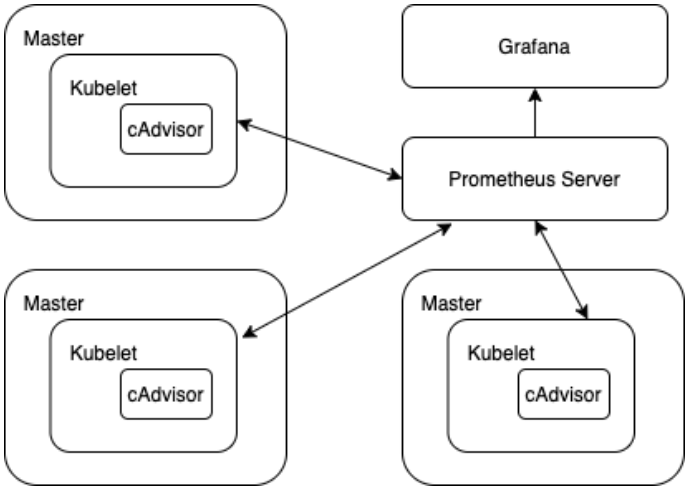



图 3.3: 监控模块架构图

- cAdvisor 是 Google 公司所开源的容器监控工具，并且也是被内置在 Kubelet 中的容器资源收集工具。在 Kubernetes 集群中可以使用 cAdvisor 工具收集 Kubernetes 集群上的节点状态信息、节点资源信息以及最小管理单位 Pod 对 CPU、内存、网络带宽、磁盘空间等资源的请求量与使用量数据，并且设置 cAdvisor 通过 Kubelet Metrics API 把数据暴露出来。

- Prometheus 是 Google 公司所开源的完整监控解决方案,可以通过使用 Prometheus Server 对接 Kubernetes 集群中已经安装的 cAdvisor 工具，对 cAdvisor 监控工具所监控的资源对象进行数据收集、处理, 并且通过 prometheus.yml 配置文件中配置 Kubernetes 集群资源对象的数据采集周期，接下来 Prometheus Server 就能够从 cAdvisor 中周期性拉取数据了，并能持久化存储监控数据。之后，预测模块与调度模块可以通过使用 PromQL 查询语言对 Prometheus 中所存储的 Kubernetes 资源对象数据进行查询。

- Grafana 是一个开源数据可视化工具，本文将其作为监控模块的监控大屏，能够在 Grafana 监控面板上查看到 Kubernetes 集群上的资源使用情况、节点状态、节点上 Kubernetes 最小部署单位 Pod 的运行状况等。

其中，可以基于 google/cadvisor:latest 镜像手动部署 cAdvisor 容器，并且使用 publish 参数设置 cAdvisor 容器暴露出 Kubentest 集群对象数据端口 Port，从而 Prometheus 能够通过 http://HostIP:Port/metrics 地址访问 cAdvisor 容器所收集的监控数据。prometheus.yml 文件就是用于配置 Prometheus Server 的文件，在对接 cAdvisor 与 Prometheus 的时候，在该文件中配置 cAdvisor 作为采集目标中的一个 Metrics Exporter。并且在该文件中配置数据采集周期以及定义报警插件等，这样当 cAdvisor 对 Kubernetes 集群的采集任务出现问题时，邮箱能够及时收到报警反馈。

## 3.3 基于 ARIMA-GRU 组合预测模型的预测模块

目前的时间序列预测方法也可以划分为线性预测方法与非线性预测方法，在 Kubernetes 集群中工作负载的影响因素较多，并且集群负载不仅包含线性部分的工作负载，还有非线性部分的工作负载。传统时间序列预测模型对线性部分的工作负载有较好的预测结果，但是对于非线性部分的工作负载无法准确预测。而在非线性负载的预测领域，神经网络时间序列预测模型的预测表现较好，因此本文构建 ARIMA-GRU 组合预测模型来预测 Kubernetes 集群负载。

### 3.3.1 ARIMA 模型构建流程

本文构建 ARIMA 时间序列预测模型的过程如下图3.4,

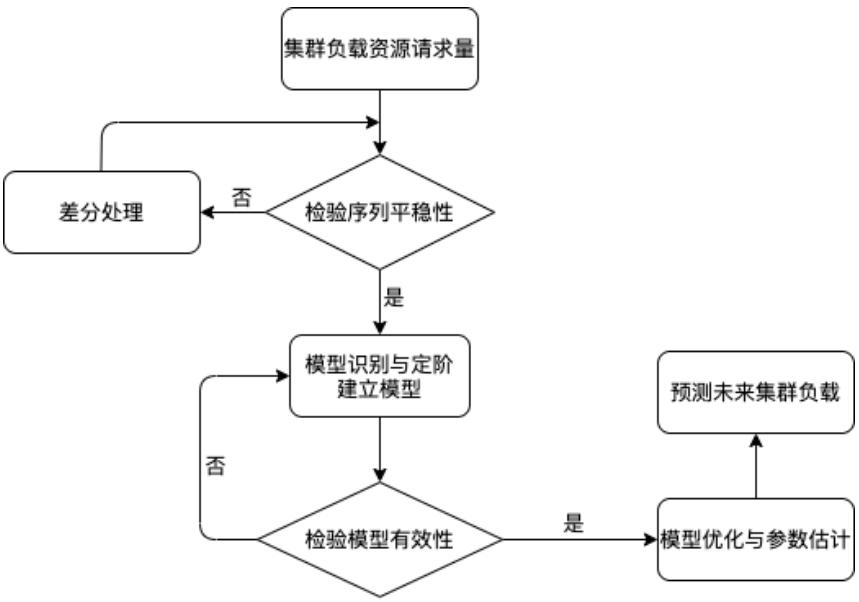



图 3.4: ARIMA 模型建立过程图

如上图3.4所示，本文建立 ARIMA 时间序列预测模型的过程包含以下步骤，

**1、平稳性检验**

容器资源请求量数据需要通过平稳性检验才能作为资源时间序列预测模型的输入，如果输入的是非平稳时间序列，则资源预测模型的预测结果不佳，落后于训练时的模型预测准确度。平稳时间序列的性质不随观测时间的变化而改变，可以通过时序图、自相关图和偏自相关图、以及单位根检验来判断时间序列的平稳性。

**2、平稳化处理非平稳数据序列**

对非平稳时间序列进行差分处理使其平稳化，在此期间，为避免过度差分需要对每次差分后的时间序列进行判断，这样通过差分的次数就可以得到 ARIMA(p, d ,q) 中的参数 d。其中，可以通过观察 ACF 图的自相关系数3.1或者 PACF 图的偏自相关系数3.2来判定输入的时间序列是否为平稳时间序列。如果自相关系数与偏自相关系数的值均趋于 0，则表示此差分时间序列已经比较平稳。要注意的是，有的非平稳时间序列要执行多次差分操作。

一阶差分表达式为: $\Delta y_t = y_{t+1} - y_t$;

二阶差分表达式为: $\Delta^2 y_t = y_{t+2} - 2y_{t+1} + y_t$;

n 阶差分表达式为: $\Delta^n y_t = \Delta\left(\Delta^{n-1} y_t\right) = \Delta^{n-1} y_{t+1} - \Delta^{n-1} y_t$;

上述差分表达式中 t 为时间序列中的时间节点，$y_t$ 对应时间序列中时刻 t 下的观测值。

$$ACF(k) = \sum_{t=k+1}^{N} \frac{\left(x_t - \overline{x}\right)\left(x_{t-k} - \overline{x}\right)}{\sum_{t=1}^{N}\left(x_t - \overline{x}\right)^2} \tag{3.1}$$

其中，$x_t$ 为时间序列中时刻 t 下的观测值，$x_{t-k}$ 为时间序列中时刻 t-k 下的观测值，$\overline{x}$ 为时间序列中观测值的均值, N 为时间序列的长度。

$$\begin{aligned} PACF(k) &= \frac{E\left(x_t - Ex_t\right)\left(x_{t-k} - Ex_{t-k}\right)}{\sqrt{E\left(x_t - Ex_t\right)^2}\sqrt{E\left(x_{t-k} - Ex_{t-k}\right)^2}} \\ &= \frac{cov\left[\left(x_t - \overline{x_t}\right), \left(x_{t-k} - \overline{x_{t-k}}\right)\right]}{\sqrt{var\left(x_t - \overline{x_t}\right)}\sqrt{var\left(x_{t-k} - \overline{x_{t-k}}\right)}} \end{aligned} \tag{3.2}$$

其中，$x_t$ 为时间序列中时刻 t 下的观测值，E 为均值函数，cov 为协方差函数。

**3、模型识别与定阶**

此步骤即为确定 p , q 阶数的环节。通过观察时间序列 ACF(自相关系数) 图和 PACF(偏相关系数) 图的截尾性等特征确定此预测模型的形式。比如，如果输入时间序列的 ACF 是拖尾的，而 PACF 是截尾的，那么此输入时间序列适合 AR 模型。如果输入时间序列的 ACF 是截尾的，而 PACF 是拖尾的，那么此输入时间序列适合 MA 模型。如果输入时间序列的 ACF 和 PACF 均是拖尾的，那么输入时间序列适合 ARMA 模型。但是，识别后的结果模型可能有多个，所以为确定最优预测模型，本文使用 AIC3.3和 SC3.4最小准则来求解模型的最优阶数，需要反复进行实验才能最终确定 ARIMA 模型的最佳阶数 p、q。

$$AIC = 2k - 2ln(L) \tag{3.3}$$

$$SC = kln(n) - 2ln(L) \tag{3.4}$$

公式3.3与公式3.4中 k 是待拟合预测模型中参数的个数，L 为似然函数，n 为时间序列中的样本数量。其中，k 值越小，代表模型的复杂度越低；L 值越大，代表模型的预测准确度越高。

**4、参数估计**

在确定预测模型以及模型中 d, p, q 的阶数之后，就能够估计预测模型中剩余的未知参数。其中，未知参数包含了自回归模型系数 φ 与移动平均模型系数 θ 等参数。此步骤中，使用极大似然估计方法来对剩余的未知参数进行参数估计。

**5、模型检验**

除了需要对模型中的残差序列进行自相关性检验操作之外，此步骤还要执行正态性检验操作，并且，还应该对模型残差进行异方差检验。如果模型通过了这些检验，则预测模型建立合理。

**6、模型预测**

利用该 ARIMA 模型基于本文 Kubernetes 集群负载原始时间序列数据预测，计算预测结果与原始负载数据之间的相对误差，将计算误差作为 ARIMA 模型预测精度的评价标准。如果该模型的预测误差较小，则该 ARIMA 模型预测结果较好。

### 3.3.2 GRU 模型构建流程

从 GRU 模型单元图3.5可以看出，GRU 模型把 LSTM 模型中的输入门和遗忘门合并为更新门，并且 LSTM 模型中的输出门对应 GRU 模型中的重置门。其中，更新门能够管理前一时刻对应的状态信息中有多少能够写入到当前状态中。正是这两处的改变，使得 GRU 模型训练速度更快，运行效率更高。

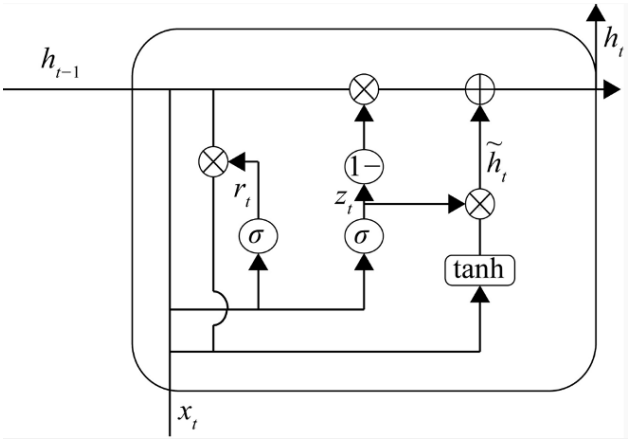



图 3.5: GRU 模型单元

按照 GRU 模型单元图3.5对 GRU 模型构建过程中的四个基本阶段进行说明，

**1、更新门** 更新信号 $z_t$ 用于确定多少的 $h_{t-1}$ 能够向前传递到下一个隐藏状态。比如，如果 $z_t$ 约等于 1，那么 $h_{t-1}$ 几乎完全都可以向前传递到下一个状态；如果 $z_t$ 约等于 0，那么大部分新记忆 $\tilde{h_t}$ 能够传递到下一个状态。公式为，

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \tag{3.5}$$

公式中 $x_t$ 表示第 t 个时间步 (时刻) 所对应的的输入向量。

**2、重置门** 更新信号 $z_t$ 用于确定 $h_{t-1}$ 对 $\tilde{h_t}$ 的影响程度。如果 $r_t$ 的值接近 1，那么就保留历史状态 $h_{t-1}$；如果 $r_t$ 的值接近 0，那么就丢弃历史状态 $h_{t-1}$。换句话说，如果能够确定 $h_{t-1}$ 与新记忆的计算没有关系，那么重置门可以消除过去的隐藏状态。公式为，

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \tag{3.6}$$

**3、新记忆** 一个新记忆 $\tilde{h_t}$ 的生成是通过新的输入向量 $x_t$ 和过去的隐藏历史状态 $h_{t-1}$ 共同计算出来的，此处是把新的输入向量 $x_t$ 和过去的隐藏历史状态 $h_{t-1}$ 进

行结合，根据过去的状态计算新记忆 $\tilde{h_t}$。

$$\tilde{h_t} = \tanh(r_t \cdot U h_{t-1} + W x_t) \tag{3.7}$$

**4、隐状态** 根据更新门的建议，结合历史隐藏输入 $h_{t-1}$ 和新记忆 $\tilde{h_t}$ 得到隐藏状态 $h_t$。公式为，

$$h_t = (1 - z_t) \cdot \tilde{h_t} + z_t \cdot h_{t-1} \tag{3.8}$$

上述公式中，$W^{(z)}$ 代表更新门输入权重矩阵，$W^{(r)}$ 代表重置门的输入权重矩阵; 更新门对应的记忆内容权重矩阵记作为 $U^{(z)}$，上一时刻重置门对应的记忆内容权重矩阵为 $U^{(r)}$。其中，$\cdot$ 代表矩阵运算中的 Hadamard 积，$\pm$ 代表常规的加减操作，$\sigma$ 代表 Sigmoid 函数，tanh 为双曲正切函数; 输入为 $x_t$, 更新为 $z_t$, 重置门为 $r_t$。上一时刻记忆内容为 $h_{t-1}$, 当下时间步记忆为 $h_t$。

### 3.3.3 建立 ARIMA-GRU 组合预测模型

尽管 ARIMA 模型与 GRU 模型都能够进行时间序列预测，但这两种预测模型有各自的适用场景，并且两者在时间序列预测上都有不足之处。此外，相较于组合时间序列预测模型，单一时间序列预测模型只能在特定场景下有较好的预测结果，而在复杂的云计算场景中，单一预测模型的预测准确度较低。构建组合预测模型可以结合每个单一预测模型的优点，并且能够在复杂场景下获得比较好的预测效果。

在 Kubernetes 集群负载的预测场景中，时间序列数据不仅包含线性部分序列数据，而且还包含非线性部分序列数据。传统时间序列预测模型 ARIMA 模型仅对时间序列中的线性部分有较好的预测效果，对非线性部分预测时会有预测结果的缺失；而 GRU 模型在时间序列中的非线性部上的中预测表现较好，但是难以挖掘出时间序列中的非线性部分。因此，本文提出 ARIMA-GRU 组合时间序列预测模型来对 Kubernetes 集群中的负载请求进行预测。在构建组合时间序列预测模型时，主要有串联组合时间序列预测模型和并联组合时间序列模型两种方式。鉴于串联组合时间序列预测模型能够将多个单一预测模型的性能进行聚合，可以提升预测模型的预测精度，因此本文选用串联组合预测模型，构建方式如下图3.6所示。

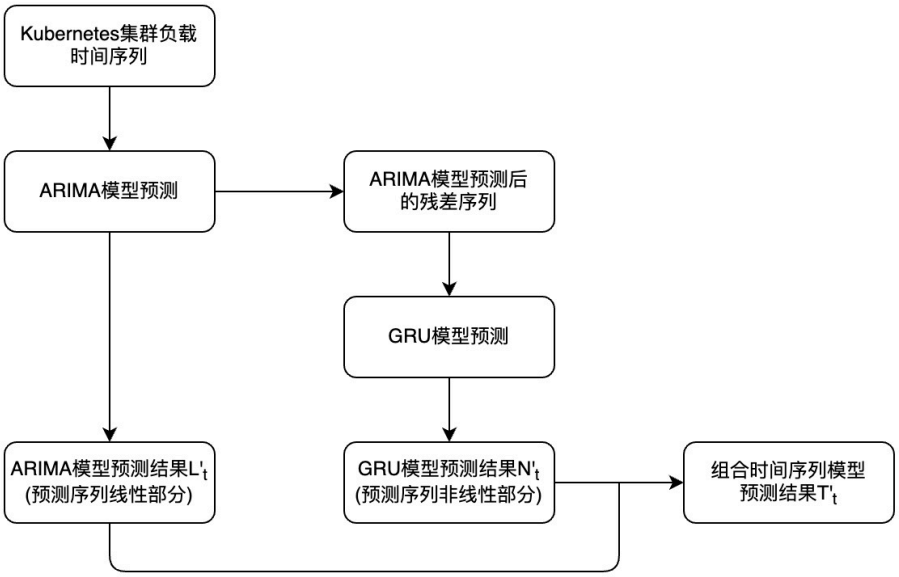



图 3.6: 串联组合时间序列预测模型

本文 ARIMA-GRU 组合预测模型的具体预测流程为，首先使用 ARIMA 模型预测拟合原始时间序列，获取时间序列线性部分的预测结果以及残差序列，负载时间序列的非线性部分就包含在残差序列中。然后，使用 GRU 模型对负载时间序列的非线性部分进行预测，得到负载时间序列非线性部分的预测结果。最后，将两部分预测结果进行求和就可以得到组合预测模型的预测结果。预测流程设计如下，

假设 Kubernetes 集群负载时间序列 $T_t$ 由线性部分和非线形部分组成，表示为，

$$T_t = L_t + N_t \tag{3.9}$$

其中，$L_t$ 和 $N_t$ 分别表示 Kubernetes 集群负载时间序列中的线性部分和非线性部分，接下来介绍 ARIMA-GRU 组合预测模型进行负载预测的具体步骤，

**步骤一:** 利用 ARIMA 模型拟合 Kubernetes 集群负载时间序列，得到线性部分的预测结果 $L'_t$ 以及残差序列 $e_t$。因为 ARIMA 模型无法预测负载时间序列的非线性部分，所以一定存在残差值，残差序列的表达式为，

$$e_t = T_t - L'_t \tag{3.10}$$

**步骤二:** 使用 GRU 模型对残差序列 $e_t$ 建模，预测 Kubernetes 集群负载时间序列中的非线性部分。使用非线性函数 $f$ 当作 GRU 模型所建立的关系函数，且该神

经网络模型有 n 个输入，得到最终非线性部分的预测值 $N_t'$，表达式为，

$$N_t' = f\left(e_{t-1}, e_{t-2}, \ldots, e_{t-n}\right) \tag{3.11}$$

**步骤三：** 分别得到 Kubernetes 集群负载时间序列中线性部分与非线性部分对应的负载预测值之后，两者相加，即可得到集群负载预测结果 $T_t'$，表达式为，

$$T_t' = L_t' + N_t' \tag{3.12}$$

**步骤四：** 将集群负载预测结果 $T_t'$ 封装在 API 中供弹性伸缩模块查询、使用。

在上述 ARIMA-GRU 组合预测模型进行实际负载预测之前，需要对 ARIMA-GRU 组合预测模型进行训练、调优，ARIMA-GRU 组合预测模型的训练流程可总结为下图3.7，

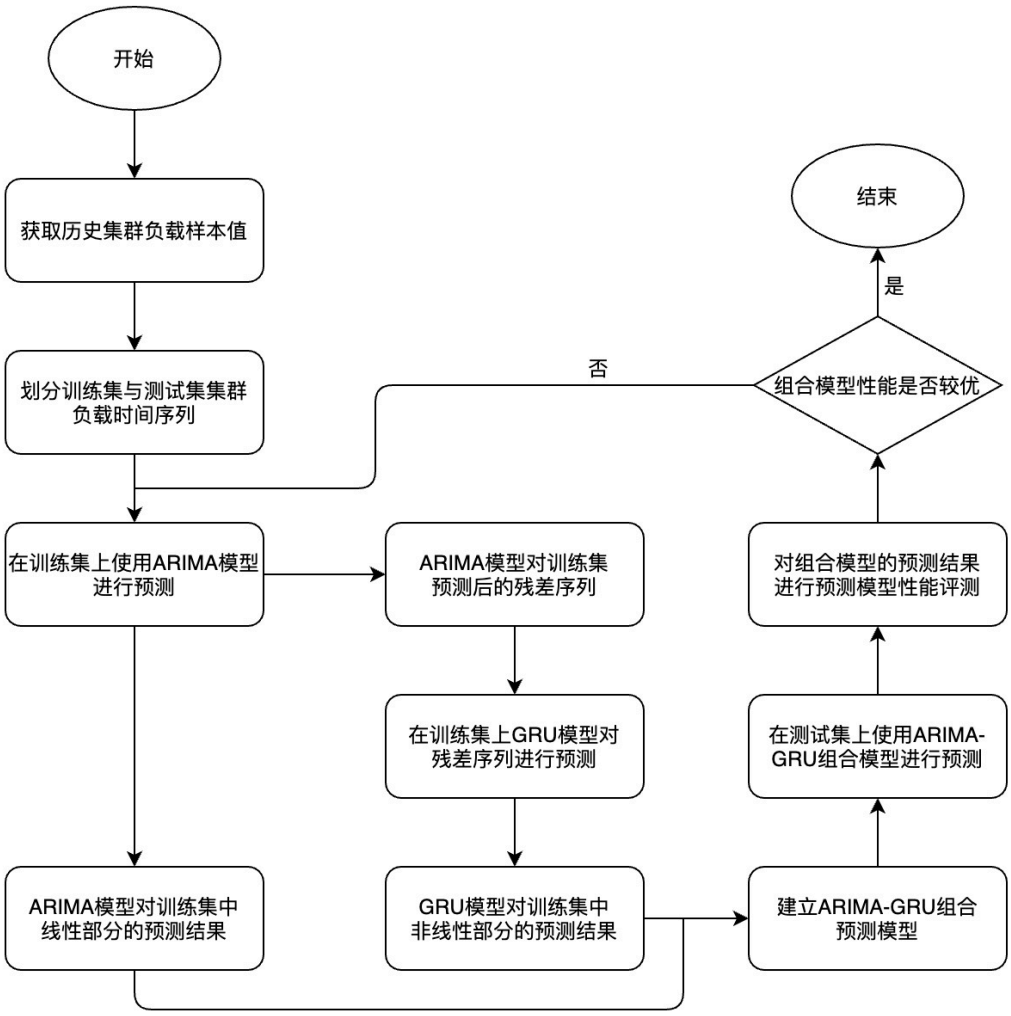



图 3.7: ARIMA-GRU 组合模型训练流程示意图

按照上图3.7对 ARIMA-GRU 组合预测模型进行训练、调优的具体流程如下，

**步骤一：** 查询存储在 Prometheus 中的 Kubernetes 集群资源对象数据并构建数据集。监控模块所收集的 Kubernetes 集群上资源对象的相关数据存放在 Prometheus 中，预测模块使用 PromQL 查询语言按照一定的查询指标、时间间隔从 Prometheus 中查询近一定时间周期内的 Kubernetes 集群负载、资源历史使用量等数据，其中还可以通过 PromQL 中的内置函数、聚合函数进行聚合查询。数据查询完成后，进行数据清洗操作，构建对 Kubernetes 集群进行负载预测所需的数据集。

**步骤二：** 从数据集中划分训练集和测试集。将步骤一所构建的 Kubernetes 负载数据集分为多组，将每组数据中的 70% 当作训练集，余下 30% 当作测试集。

**步骤三：** 自回归差分移动平均 (ARIMA) 模型拟合预测训练数据集。在训练集中，首先使用 ARIMA 模型拟合训练集负载时间序列，得到训练集中负载时间序列线性部分的预测结果以及残差序列。

**步骤四：** 门控循环单元 (GRU) 模型对步骤三中的残差序列进行预测。使用 GRU 模型对 ARIMA 模型预测之后的残差序列再次进行预测，得到训练集中负载时间序列非线性部分的预测结果。

**步骤五：** 获取自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型的预测结果。将步骤三中自回归差分移动平均 (ARIMA) 模型对训练集中负载时间序列线性部分的预测结果与步骤四中门控循环单元 (GRU) 模型对训练集中负载时间序列非线性部分的预测结果进行相加，得到针对训练集负载时间序列的组合预测结果。

**步骤六：** 验证组合预测模型的性能。在测试集上基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型进行负载预测，得到预测结果，使用预测模型性能评测指标评价预测模型的预测准确度。如果组合预测模型的预测准确度较低，则不断调整、优化自回归差分移动平均模型 (ARIMA) 模型与门控循环单元 (GRU) 模型，使得 ARIMA-GRU 组合预测模型有较好的预测精度。

其中，预测模块中所使用的部分 PromQL 查询语句如下所示，

- 查询一分钟内不同节点实例上的 CPU 资源使用率

  (1 - sum(rate(node_cpu_seconds_totalmode=”idle”[1m])) by (instance) / sum(rate(node_cpu_seconds_total[1m])) by (instance) ) * 100

- 查询五分钟内不同节点实例上的磁盘 I/O

  rate(node_disk_reads_completed_total[5m]) +rate(node_disk_writes_ completed_total[5m])

- 查询五分钟内不同节点实例上的网络 I/O

  sum by(instance) (irate(node_network_receive_bytes_totaldevice! ”bond.*?|lo”[5m]) + irate(node_network_transmit_bytesdevice! ”bond.*?|lo”[5m]))

### 3.3.4 组合预测模型性能评价指标

完成构建 Kubernetes 集群负载时间序列组合预测模型之后，为了检验组合预测模型的预测效果，需要对组合预测模型的预测准确度进行合理的评价。准确度表示的是实际值与预测值的拟合程度, 误差可以直接反应拟合程度，误差越小则预测结果的准确度就越高。本文选择三种常见的预测模型结果评价模型，分别是:

1、平均绝对误差，即 MAE, 其对应的公式为,

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \overline{y_i}| \tag{3.13}$$

2、平均绝对百分比误差，即 MAPE，其对应的公式为,

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \overline{y_i}}{y_i}\right| \tag{3.14}$$

3、均方根误差，即 RMSE，其对应的公式为,

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y_i})^2} \tag{3.15}$$

## 3.4 基于集群负载预测结果与 HPA 配置的弹性伸缩模块

Kubernetes 中基于水平伸缩控制器 (HPA) 所实现的默认扩缩容机制存在响应不及时的问题, 并可能进一步导致服务质量下降、浪费集群资源等问题。针对这些问题，本文提出一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略。其中，根据 Kubernetes 集群负载预测模型的输出预测结果计算 Kubernetes 中最小管理单位 Pod 的副本数量预测值，结合水平伸缩控制器 (HPA) 所设置的资源阈值、最小副本数、最大副本数、容忍系数等参数，计算出目标 Pod 副本期望值，协同实现 Pod 的提前扩缩容，从而避免因 Pod 扩缩容不及时所可能导致的各种问题。本文弹性伸缩模块包含本文所提出的基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略。

弹性伸缩模块工作流程图如下图3.8所示,

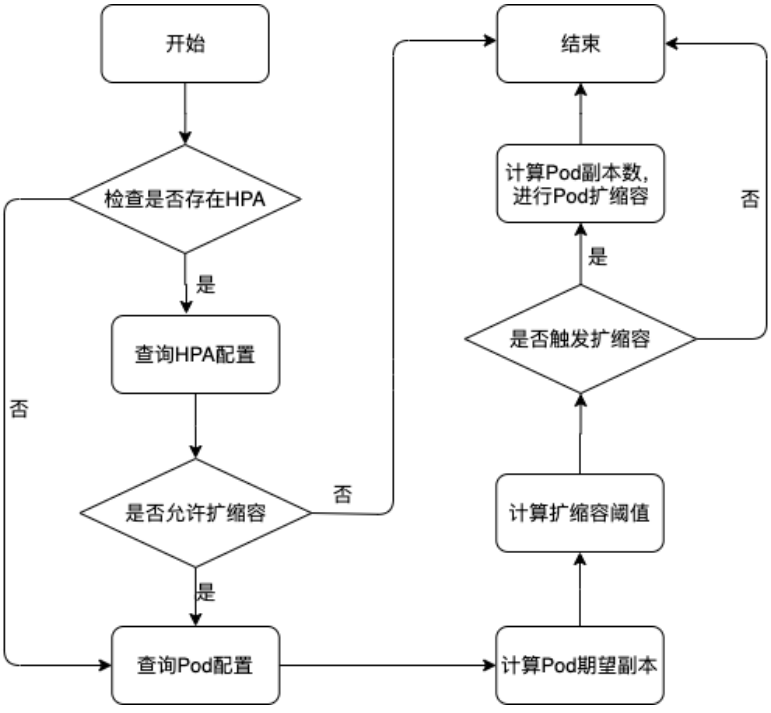



图 3.8: 弹性伸缩模块工作流程图

本文所使用的 Kubernetes v1.20.0 中水平伸缩控制器 (HPA) 目前仅支持 CPU 使用率和内存使用率作为判断 Kubernetes 最小管理单位 Pod 是否要进行伸缩的资源

指标。因此，为避免本文所提出的弹性伸缩策略与水平伸缩控制器 (HPA) 中已经定义的扩缩容规则产生冲突，本文弹性伸缩策略也根据目标 Pod 当前 CPU 使用率和内存使用率指标进行设计。弹性伸缩策略执行过程如下:

**步骤一:** 查询待伸缩 Pod 是否设置对应的水平伸缩控制器 (HPA)，如果存在对应水平伸缩控制器 (HPA)，则获取水平伸缩控制器 (HPA) 声明文件中 Pod 最小副本数 minReplicas、Pod 最大副本数 maxReplicas、伸缩资源指标 TargetMetrics、是否禁用扩缩容等字段值。如果待伸缩 Pod 没有设置对应的水平伸缩控制器 (HPA), 则忽略该步骤。如果目标 Pod 所对应的水平伸缩控制器 (HPA) 不允许进行扩缩容, 则后续步骤不再执行，流程结束。

**步骤二:** 查询目标 Pod 的直接控制器，比如 Deployment 资源对象，获取目标 Pod 对 CPU 资源和内存资源的申请量 Requested，以及目标 Pod 当前副本数量 CR。

**步骤三:** 计算目标 Pod 副本预测值。监控模块已经收集目标 Pod 对 CPU 和内存资源的历史使用量数据，预测模块预测目标 Pod 之后一段时间内对 CPU、内存资源的需求量，然后计算 Pod 副本预测值。公式如下,

$$DR = \max\left(ceil\left(CR * \left(\frac{Used_i}{Requested_i}\right)\right)\right) \tag{3.16}$$

其中，i 为资源编号，i=1 时表示 CPU 资源，i=2 时表示内存资源；DR 为目标 Pod 副本期望数值；CR 为目标 Pod 当前副本数量；$Used_i$ 为负载预测模型目所预测的在未来一段时间内标 Pod 对资源 i 所需的资源量；$Requested_i$ 为目标 Pod 在创建时对资源 i 的资源申请量；ceil 函数为向上取整函数。

**步骤四:** 计算目标 Pod 副本扩缩容阈值。水平伸缩控制器 (HPA) 的 Yaml 文件中定义了目标 Pod 副本伸缩阈值，以及最大副本数 maxReplicas 与最小副本数 minReplicas 字段值。HPA 控制器为避免非必要 Pod 副本扩缩容所带来的开销以及保证集群稳定性，设置了容忍系数 t, 其默认值为 0.1。HPA 控制器中 Pod 副本扩缩容阈值可通过下式3.17计算得出，最大扩容阈值为 maxR，最小扩容阈值为 minR。

$$\begin{cases} maxR = (1+t) * maxReplicas \\ minR = (1-t) * minReplicas \end{cases} \tag{3.17}$$

**步骤五:** 计算目标 Pod 副本期望值。比较 Pod 副本预测值与当前目标 Pod 副本扩缩容阈值之间的大小，如果 Pod 副本预测值小于最大扩容阈值 maxR，则触发副本扩容操作，转入步骤六；如果 Pod 副本期望值大于最小扩容阈值 minR，则触发副本缩容操作，转入步骤六。否则，流程中止。

**步骤六:** 根据 Pod 副本期望值与水平伸缩控制器 (HPA) 扩缩容阈值确定目标副本的期望值，最后通过 Kubernetes REST API 与调度模块更新 Pod 副本。其中，maxR 为包含了容忍系数的目标 Pod 最大副本数，minR 为包含了容忍系数的目标 Pod 最小副本数，将公式3.16中目标 Pod 期望副本数 DR 与目标 Pod 最大副本数 maxR、目标 Pod 最小副本数 minR、目标 Pod 当前副本数 CR 进行比较。如果目标 Pod 期望副本数 DR 大于目标 Pod 当前副本数，并且目标 Pod 期望副本数 DR 小于目标 Pod 最大副本数 maxR，则对 Pod 副本按照目标 Pod 期望副本数 DR 进行扩容。如果如果目标 Pod 期望副本数 DR 小于目标 Pod 当前副本数，并且目标 Pod 期望副本数 DR 大于目标 Pod 最小副本数 minR，则对 Pod 副本按照目标 Pod 期望副本数 DR 进行缩容。公式表达如下，

$$Replicas = \begin{cases} DR, CR \leq DR \leq maxR \\ maxR, CR \leq DR \wedge maxR \leq DR \\ DR, minR \leq DR \leq CR \\ minR, DR \leq CR \wedge DR \leq minR \end{cases} \tag{3.18}$$

其中，目标 Pod 副本调整过程需要服从水平伸缩控制器 (HPA) 中所定义的扩缩容速率字段值来进行扩缩容操作。并且，开发者还可以通过 HPA 控制器中的 behavior 字段定义多个扩缩容策略，从而满足不同用户的 Pod 副本扩缩容需求。

此外，由于 Pod 副本的扩缩容均需要一定时间，所以如果在短时间内多次扩缩容，会影响 Pod 副本期望结果。因此水平伸缩控制器 (HPA) 设置了默认冷却时间，扩容操作冷却周期为 3 分钟，即上一次扩容操作声明后 3 分钟内，不能再次声明新的扩容操作；缩容操作冷却周期为 5 分钟，即上一次缩容操作声明后 5 分钟内，不能再次声明新的缩容操作。

## 3.5 本章小结

本章针对 Kubernetes 默认伸缩机制所存在的一些问题，提出一些改进与优化方案，并对改进方案进行设计与实施。首先，本章介绍改进后的 Kubernetes 整体架构设计，分别说明架构中各个模块，并且阐述各个模块之间的交互流程，接下来还说明了监控模块中组件构成以及组件安装、使用。然后，本章介绍预测模块中自回归差分移动平均 (ARIMA) 模型与门控循环单元 (GRU) 模型的构建流程，并且详细阐述自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型的训练流程以及实际负载预测流程，并介绍用于评价本文时间序列预测模型性能的评价指标。最后在弹性伸缩模块中构建基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略，详细说明弹性伸缩策略的具体执行过程。

本章已经介绍了改进后 Kubernetes 的整体架构设计，以及监控模块、预测模块与弹性伸缩模块的设计思路与构建流程，但是没有介绍调度模块。当弹性伸缩模块通过 Kubernetes REST API 向 Kubernetes API Server 发送扩缩容请求之后，本文所设计的调度模块将进行具体调度，调度模块将在下一章详细阐述。

# 第四章　Kubernetes 动态调度策略设计与实现

第三章介绍了改进后的 Kubernetes 整体架构设计，并且说明了监控模块、预测模块与弹性伸缩模块的设计思路与构建流程，当弹性伸缩模块向 Kubernetes API Server 发送扩缩容请求或者出现新的 Kubernetes 基本部署单位 Pod 的部署请求时，本文所设计的动态调度模块将进行具体调度。本章将详细介绍本文所构建的 Kubernetes 扩展调度器、基于集群负载均衡度的调度策略以及动态调整 Kubernetes 基本部署单位 Pod 资源限额的方案。

针对 Kubernetes 默认调度机制的不足，本章首先介绍目前 Kubernetes 调度器的扩展方案，然后对比不同方案的优劣，选择最合适的方案。接下来详细介绍基于 Scheduling Framework 构建 Kubernetes 扩展调度器的流程，然后提出一种基于集群负载均衡度的调度策略，并详细阐述该调度策略的设计流程。最后，对本文所提出的动态调整 Kubernetes 基本部署单位 Pod 资源限额方案的设计思路与实践方式进行详细说明。

## 4.1　资源调度模块

根据预测模块的负载预测结果与水平伸缩控制器 (HPA) 配置计算目标 Pod 期望副本数量，预测模块把目标 Pod 副本数量更新需求提交到 API Server 之后，接下来进行 Kubernetes 基本部署单位 Pod 的具体调度流程。调度过程中，Kubernetes Scheduler 组件与集群中其他组件的交互如下图 4.1所示，Kubernetes Scheduler 的调度流程可以分为以下几个步骤:

- **步骤一:** 用户可以通过使用 Kubectl 命令工具或者调用 Kubernetes REST API 来声明、创建或者更新 Deployment、ReplicaSet、Pod 等资源对象。
- **步骤二:** API Server 监听资源请求，将相应的资源对象配置信息存储在 Etcd 中。其中，Controller Manager 组件通过 API Server 监控 Kubernetes 集群中资

源对象的运行状态，保证集群中资源对象符合预期声明的工作状态，并且负责资源对象的自动扩展、滚动升级等。

- **步骤三:** Scheduler 通过 List&Watch 机制监控 API Server，获取待调度 Pod 以及可用 Node 节点集合，按照调度策略进行调度。调度流程包含预选阶段与优选阶段，在预选阶段，按照预选策略筛选符合条件的节点。在优选阶段，对预选阶段得到的 Node 节点集合按照优选策略计算各节点的得分，节点分数范围为 0-10, 节点分数越高，则说明待调度 Pod 越适合调度到该节点上。如果有多个分数相同的节点，则随机选择其中一个节点执行调度操作。
- **步骤四:** 优选阶段中得分最高的节点即为最适合调度的节点，对该节点与待调度 Pod 执行绑定操作，使用 Kubelet 向 API Server 发送请求把目标节点与待调度 Pod 的绑定信息存储到 Etcd 中。

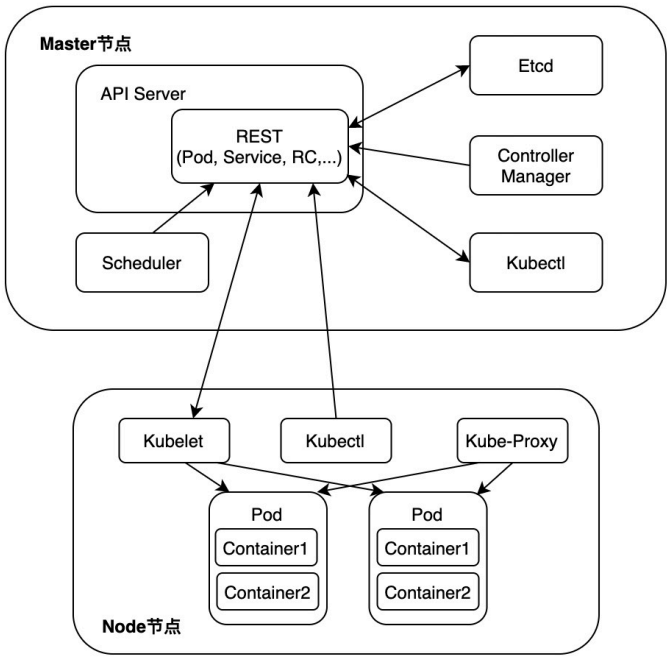



图 4.1: 调度过程中组件交互图

上图4.1展示了 Kubernetes 基本部署单位 Pod 的调度流程，本文基于 Scheduling Framework 所构建的 Kubernetes 扩展调度器主要用于替换图4.1中的 Kubernetes Scheduler 组件，并且本文所构建的 Kubernetes 扩展调度器的工作流程需要和上图4.1中的相关步骤保持一致。

## 4.2 Kubernetes 扩展调度器方案研究

Kubernetes 默认调度机制存在一定缺陷与不足，其在调度时资源指标考虑单一，不能全面考虑 Kubernetes 集群中多种资源指标，这可能会造成某种类型资源的性能瓶颈问题。目前，在云原生领域有多种方案可以扩展 Kubernetes 调度器，允许用户改进 Kubernetes 默认调度机制。经过多种扩展方案的比较，本文最终选择基于 Scheduling Framework 构建扩展调度器。

目前主要有四种方案可以扩展 Kubernetes 调度器，介绍与比较如下。

1. **直接下载 Kubernetes 源代码并改进，**在用户需要改进的位置直接修改 Kubernetes 源代码，允许开发者添加、实现自定义调度算。之后将自构建的调度逻辑注册到调度流程中，对源代码重新进行编译，编译完成后重新部署修改后的调度器。但是这种方式非常不建议使用，因为 Kubernetes 上游代码会变化，当以这种方式自定义调度器时，需要耗费大量的时间、精力去和上游变化代码保持一致。
2. **配置多个调度器，**这种方案允许 Kubernetes 默认调度器与自定义调度器同时在 Kubernetes 集群中运行。实际使用时，两个调度器的调度流程各自独立，用户可以通过设置 Pod 的 spec.schedulerName 字段值来使用不同的调度器，如果在 Pod 声明文件没有设置 spec.schedulerName 字段值，则使用 Kubernetes 默认调度器。但是，当多个调度器同时存在于 Kubernetes 集群中时，会出现一些问题。比如当两个调度器准备将不同 Pod 调度到同一个 Node 节点上时，很有可能第一个调度器已经把一个 Pod 调度到该 Node 节点上，此时第二个调度器即将完成另一个 Pod 的调度。对于第二个 Pod 而言，该 Node 节点可能已经不是最合适的调度节点，因为第一个 Pod 改变了节点状态。甚至，在第一个 Pod 调度完成后，该 Node 节点上的资源不再能支持第二个 Pod 的运行。所以，在多调度器共存的时，维护高质量的自定义调度器困难较多。
3. **使用调度扩展程序构建自定义调度器，**该方案基于 Kubernetes Scheduler Extender 机制实现自定义调度器。该机制的实质就是通过 Scheduler 所提供的

webhook 机制实现自定义调度算法，可以实现和上游程序的兼容。换句话说，此调度扩展程序的本质就是一个可进行配置的 webhook。但是，自定义调度扩展程序与 Kubernetes 默认调度程序之间是通过 https 的方式进行通信、传输数据的，其中要对两者之间的传输数据执行序列化与反序列化操作，所以存在一定的网络开销。

4. **通过调度框架 (Scheduling Framework) 构建自定义调度器，**在 Kubernetes v1.15 版本中，Kubernetes 引入可插拔插件式的调度框架结构。在 Kubernetes 默认调度流程中，已经设置了大量扩展点接口，这使得开发者自定义调度器变得简单。该调度框架结构允许用户为现有调度器编写多个扩展插件，这些插件能够在不同调度阶段作为现有调度器的扩展，并且这些自定义插件可以和现有调度器源代码一块编译到调度程序中。目前，这种方式是 Kubernetes 官方目前最为推崇的扩展调度器方案，之后也会是构建自定义调度器的首选方案。并且，在 Kubernetes v1.16 版本中，方案 3 中 Scheduler Extender 机制已经被废弃、不再维护。

表 4.1: 扩展调度器方案总结

| 方案 | 优点 | 缺陷 |
|---|---|---|
| 方案一: 直接修改源码 | 直接、改动小 | 维护难度大 |
| 方案二: 多调度器运行 | 不改动源码 | 存在调度竞争 |
| 方案三: 调度扩展程序 | 不改动源码 | 官方不再维护、存在网络耗时 |
| 方案四: 调度框架 | 轻量、扩展方便 | 代码难度大 |

以上对几种常见 Kubernetes 扩展调度器构建方案进行了介绍，将它们各自的优缺点总结于上表4.1中。并且，通过该表了解到基于 Scheduling Framework 构建自定义调度器的方式是目前主流方式。相比其他方案，该方案没有明显短板，也是 Kubernetes 官方推荐的方式，因此本文基于 Scheduling Framework 构建扩展调度器。

## 4.3 基于 Scheduling Framework 构建扩展调度器

Kubernetes 社区最初在 Kubernetes v1.16 中推出一种 Kubernetes 调度框架，也就是后来为开发者所熟知的 Kubernetes Scheduling Framework。该调度框架设置了

大量扩展点接口，实现扩展点接口的过程中开发者能够添加自定义调度逻辑，从而完成扩展调度插件构建，之后扩展会被注册到扩展点上。这样的话，在调度流程中，当执行到调度框架中相应扩展点时，就会使用之前开发者自定义的扩展插件。调度框架最初在设计扩展点时考虑了开发者的各种需求，比如通过修改某些扩展点逻辑能够自定义程序决策策略，而某些扩展点上的扩展逻辑仅用于发送通知。总的来说，开发者在使用 Scheduling Framework 构建扩展调度器时，需要在扩展点接口位置扩展逻辑，然后注册扩展插件以及相关插件配置，这样就把自定义调度逻辑集成到了调度框架中。调度框架的设计目标是为了提升 Kubernetes 原生内置调度器的扩展性，并允许将原生调度器的相关功能以插件化的形式进行拓展，从而实现开发者的自定义调度逻辑，这样简化了原生调度器的调度逻辑与扩展逻辑。

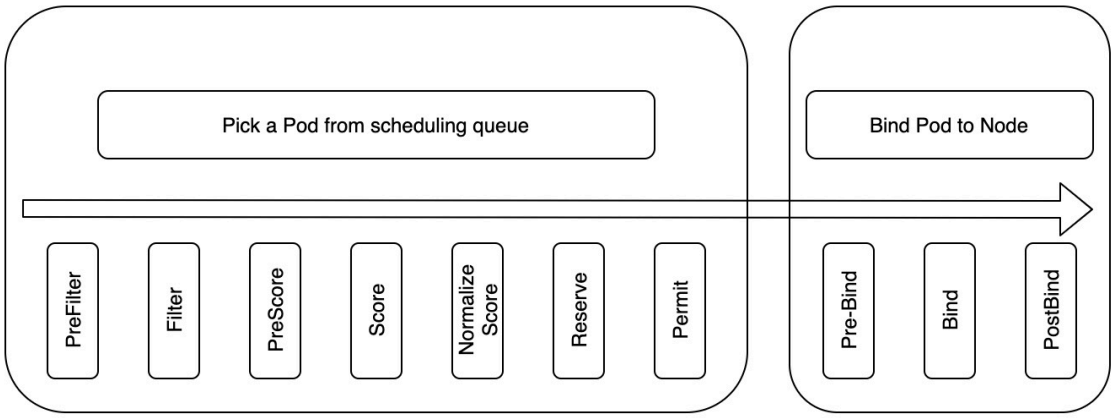



图 4.2: Scheduler Framework 扩展点示意图

对 Kuberbetes 最小管理单位 Pod 的调度分为两个过程，分别是调度 (scheduling) 过程和绑定 (binding) 过程，两者结合后被统称为调度上下文 (scheduling context)。调度过程中相关操作同步执行，即在同一时刻，有且仅有一个 Pod 能被调度。而绑定过程中相关操作可以异步进行，即在同一时刻，多个 Pod 能够同时进行绑定操作。调度框架扩展点结构如上图4.2所示，该图展示了调度框架中的调度上下文，并且途中标注了调度框架中扩展点的位置。每一个调度器扩展插件可以注册其中多个扩展点，在某些复杂场景下这是必要的。

接下来对调度框架图4.2中相关扩展点 (Extension Points) 进行说明，

1. Pre-filter 扩展对 Pod 的相关信息执行预处理操作，或者核实 Pod 被调度前需

要符合的一些条件。假如 Pre-filter 的返回值为 error，那么就要停止调度过程。

2. Filter 扩展会排除掉不支持运行待调度 Pod 的节点，并且针对集群中每一个节点，都需要遵循一定顺序执行多个 Filter 扩展。假如某节点被其中任意一个 Filter 扩展判定为不可选，那么不再对该节点进行其他 Filter 扩展操作。其中，调度器能够对多个 Node 节点在同一时刻进行 Filter 扩展操作。
3. Post-filter 扩展是执行通知任务的扩展点，Filter 阶段完成后 Node 节点列表作为该扩展的输入参数，该扩展能够生成日志信息或者指标信息，以及更新集群内部状态信息。
4. Scoring 扩展计算所有通过筛选的节点的分数，调度器对所有 Node 节点使用 Scoring 扩展，将节点分数限定在一定范围内。在 Normalize scoring 阶段，调度器将所有参与打分的 Scoring 扩展对节点的打分结果与该 Scoring 扩展权重值进行加权计算，以此作为该节点最终打分结果。

本文基于 Scheduling Framework 构建扩展调度器，主要对上图4.2中 Filter 扩展点与 Score 扩展点进行扩展，该扩展调度器使用多维资源指标调度来进行 Pod 的调度。开发者如果要在默认调度器的基础上实现自定义调度插件，需要向调度框架进行插件的注册，并且还要对插件进行配置。此外，开发者还需要实现调度框架中相应扩展点接口。该扩展调度器具体实现过程如下，

**步骤一:** 在构建自定义插件之前，需要进行一定的准备工作。需要提前搭建一个 Kubernetes 集群，之后使用自构建扩展调度器替换 Kubernetes 内置调度器。并且对于 Kubernetes 官方项目 Kubernetes-sigs/scheduler-plugins，将其 fork 到开发者自己的仓库，然后将该仓库克隆到自己本地，进行具体开发工作。

**步骤二:** 本文构建一个名称为 MultiMetricsPlugin 的过滤 (Filter Plugin)-评分插件 (Score Plugin)，该插件综合考虑多种资源指标，比如 CPU、内存、网络 I/O、磁盘 I/O 和磁盘容量指标信息。因为监控模块的监控数据存储在 Prometheus 中，所以 MultiMetricsPlugin 插件的输入参数需要从 Prometheus 中获取，其中要在 scheduler-plugins/pkg 目录下定义插件结构体以及插件与 Prometheus 的交互逻辑。

**步骤三:** 实现 Filter 扩展接口，重写 Filter 函数，使用多维资源指标进行节点过滤。此外，还要实现 Scoring 扩展接口，把 Prometheus 返回数据信息作为 Scoring 扩展接口中 Score 函数的参数。然后使用多维资源指标计算节点分数，并对 NormalizeScore 函数进行重写，将节点最终分数限制为 0-10 之间。其中，New 函数需要服从调度框架 PluginFactory 的接口规范。此外，调度框架中扩展点接口存放在 Kubernetes/pkg/scheduler/framework/interface.go 文件中。

**步骤四:** 在 scheduler-plugins/pkg/apis 文件夹下进行 MultiMetricsPlugin 插件的配置,配置文件为 scheduler-plugins/pkg/apis/config/types.go 和 scheduler-plugins/pkg/apis/config/v1beta1/types.go。这两个文件中结构体必须遵循 <Plugin Name>Args 命名规范，否则这两个配置文件不能被正确解析。然后运行 scheduler-plugins/hack/update-codegen.sh 文件，该文件用于进行代码生成。通过使用代码生成器，才能构建与 Kubernetes 内置资源控制器工作原理相同的控制器。

**步骤五:** 注册自定义插件与其配置,主要对 scheduler-plugins/pkg /apis/config/register.go 文件与 scheduler-plugins/pkg/apis/config/v1beta1/register.go 文件进行修改。

至此, MultiMetricsPlugin 插件的代码逻辑、配置注册, 以及自定义扩展调度器镜像的构建都已经完成, 接下来就可以在 Kubernetes 集群中使用自定义扩展调度器替换 Kubernetes 内置调度器。其中, 为了集群稳定性与安全性, 可以在 Kubernetes 集群中为自定义扩展调度器构建独立 ServiceAccount、ClusterRole 和 ClusterRoleBinding 等资源对象，从而对扩展调度器相关权限进行限制。至此，就完成了 Kubernetes 自定义扩展调度器的部署安装。

本文中基于 Scheduling Framework 所构建的自定义扩展调度器是对调度框架中 Filter 和 Scoring 扩展点进行扩展，Pod 调度过程中预选阶段与优选阶段均使用了多维资源指标策略。与 Kubernetes 集群默认调度机制相比，此种调度策略更全面地考量指标、更能实现集群资源负载均衡。并且，此自定义扩展调度器扩展性较强，能够在调度上下文中多个扩展点位置进行插件式扩展，在之后的使用场景中方便继续进行更新、迭代。

## 4.4 基于集群负载均衡度的调度策略

Kubernetes 默认调度机制在节点调度过程中仅考虑了 CPU 资源与内存资源，这种调度机制在考量调度指标时具有一定片面性。这是因为 Kubernetes 集群中除了 CPU 资源与内存资源，还有很多其他类型资源。在进行调度时，仅考虑这两类资源而不考虑其他类型资源可能会导致集群中其他类型资源的使用不均衡。比如，Kubernetes 默认调度机制对磁盘 I/O 资源不敏感，在某些场景下或许会把多个磁盘 I/O 密集型 Pod 调度到同一个 Node 节点上，这样会造成该 Node 节点上磁盘 I/O 资源的性能瓶颈问题，严重影响节点上其他应用 Pod 的稳定运行。此外，Kubernetes 默认调度机制也忽略了不同任务类型 Pod 对资源需求的不均匀问题，有些应用 Pod 对 CPU 资源的需求量可能比对其他资源的需求量更大，那么此 Pod 就是一个计算型 Pod。在节点优选阶段中进行节点打分时，节点上 CPU 资源指标在节点分数中权重就应该更高一些，这样才更符合应用 Pod 本身的资源需求特征。

针对上述 Kubernetes 原生调度机制所存在的问题，本文构建了基于集群资源均衡度的调度策略，在进行 Pod 调度的时候，自定义扩展调度器将多维资源指标作为输入，比如 CPU、内存、网络 I/O、磁盘 I/O 和磁盘容量资源的数据信息，在资源指标方面考虑的比较全面。在进行节点优选时，对于不同类型 Pod，比如计算型 Pod、I/O 型 Pod、网络型 Pod，针对应用 Pod 资源需求特征在节点评分时为其主导需求资源设置较大权重，实现引用 Pod 差异化调度，优化节点优选策略。本文中基于集群资源均衡度的调度策略也是一种针对 Kubernetes 负载的动态调度策略，每当进行 Pod 调度时, 该调度策略会考虑当该 Pod 调度到集群中不同节点上时，集群中整体资源负载的各种变化情况，从而就能够选取出最有利于集群负载均衡的节点来运行目标 Pod。

本文所构建的基于集群负载均衡度调度的算法，即 ClusterLoadBalancePriority (CLBP) 算法。CLBP 算法作为集群负载动态调度算法，考虑了集群中全部节点上所有负载的实时运行情况，并且集群资源均衡度同时也考虑了集群中全部节点自身的性能。除此之外，CLBP 策略在对节点进行打分之前，会根据应用 Pod 资源请

求量的差异把应用 Pod 划分为不同类型。从而在节点打分过程中，根据 Pod 类型为其主导需求资源设置较大节点分数权重。

假设某 Pod 对 CPU、内存、网络 I/O、磁盘 I/O 和磁盘容量资源的需求量分别为 $R_c$、$R_m$、$R_{nio}$、$R_{dio}$、$R_d$；某 Node 节点上当前 CPU、内存、网络 I/O、磁盘 I/O 和磁盘容量等资源的负载 (使用量) 为 $N_c$、$N_m$、$N_{nio}$、$N_{dio}$、$N_d$；该 Node 节点上 CPU、内存、网络 I/O、磁盘 I/O 和磁盘容量资源的最大负载为 $T_c$、$T_m$、$T_{nio}$、$T_{dio}$、$T_d$。调度过程中，CLBP 调度策略的输入参数为应用 Pod 与 Node 节点的相关资源数据，均从 Prometheus 中通过 PromQL 语句查询得到。

根据某些研究，如果某 Node 节点上相关资源负载 $R_i$ 大于等于该 Node 节点上资源 $R_i$ 最大负载的 80%，则将此节点定义为高负载节点，在这种情况下一般不将应用 Pod 调度到该节点。并且，为该 Node 节点打上 NoSchedule 污点。直到该节点上主要类别资源负载 $R_j$ 都低于该 Node 节点上资源 $R_j$ 最大负载的 80% 时，才删除该 Node 节点上的 NoSchedule 污点。

在调度应用 Pod 时，首先进入节点预选阶段，Kubernetes 默认预选调度策略仅考虑 CPU 资源与内存资源能否满足应用 Pod 的资源请求量，这种调度策略比较片面。一方面，考量的资源指标不够全面；另一方面，没有考虑该应用 Pod 调度到该 Node 节点上之后 Node 节点上资源负载情况。因此，CLBP 算法在预选阶段为应用 Pod 过滤 Node 节点时，考虑了多维资源指标以及调度完成后 Node 节点负载情况。在预选阶段，Pod 资源请求量与节点负载量、节点资源总量在满足以下条件时，才认为该节点能够支持目标 Pod 的运行，否则该节点会被过滤掉。

$$R_c + N_c < T_c * 0.8 \tag{4.1a}$$

$$R_m + N_m < T_m * 0.8 \tag{4.1b}$$

$$R_{nio} + N_{nio} < T_{nio} * 0.8 \tag{4.1c}$$

$$R_{dio} + N_{dio} < T_{dio} * 0.8 \tag{4.1d}$$

$$R_d + N_d < T_d * 0.8 \tag{4.1e}$$

上述 (4.1a)(4.1b)(4.1c)(4.1d)(4.1e) 式必须同时满足，该 Node 节点才能进入到

优选调度阶段。该组公式表示节点不仅能提供目标 Pod 运行所需要的资源，还要能保证目标 Pod 调度完成后节点上仍有一定的资源余量，这样的 Node 节点才能通过预选阶段，并将此处经过预选阶段得到的 Node 节点集合称为 Filter-Node 集合。

接下来进入调度过程中优选调度阶段，在优选调度阶段，会根据应用 Pod 资源请求特征将其划分为不同类型 Pod，从而在为节点打分时，能够根据 Pod 类型，为 Pod 主导需求资源赋予较高节点分数权重，使得该类资源均衡度分数在节点分数构成中占比较高。现在将应用 Pod 分为三种类型: 计算密集型 Pod、I/O 密集型 (磁盘 I/O 密集型)Pod、网络密集型 Pod，在节点打分过程中，如果应用 Pod 属于计算密集型 Pod，则 CPU 资源权重更高一些。Pod 类型按照以下公式进行判断，

$$\frac{R_c}{T_c - N_c} \geq 0.15 \tag{4.2a}$$

$$\frac{R_{dio}}{T_{dio} - N_{dio}} \geq 0.15 \tag{4.2b}$$

$$\frac{R_{nio}}{T_{nio} - N_{nio}} \geq 0.15 \tag{4.2c}$$

当式中仅 (4.2a) 式满足时，则该 Pod 为计算密集型 Pod；如果仅 (4.2b) 式满足，则该 Pod 为 I/O 密集型 Pod；如果仅式 (4.2c) 满足，则该 Pod 为网络密集型 Pod。

但是在 Pod 调度时可能还存在一类情况，就是某个 Pod 所请求的资源种类比较多，并且对 CPU 资源、磁盘 I/O 资源等均有较大需求，这个时候需要进一步判断 Pod 类型。

$$\left\{\frac{R_c}{T_c - N_c}\right\} / \left\{\frac{R_{dio}}{T_{dio} - N_{dio}}\right\} \geq 1 \tag{4.3a}$$

$$\left\{\frac{R_c}{T_c - N_c}\right\} / \left\{\frac{R_{nio}}{T_{nio} - N_{nio}}\right\} \geq 1 \tag{4.3b}$$

$$\left\{\frac{R_{dio}}{T_{dio} - N_{dio}}\right\} / \left\{\frac{R_{nio}}{T_{nio} - N_{nio}}\right\} \geq 1 \tag{4.3c}$$

按照上述公式，若式 (4.2a) 与式 (4.2b) 均满足，则进行式 (4.3a) 的判定，如果满足式 (4.3a)，则该 Pod 为 CPU 密集型 Pod，若不满足式 (4.3a)，则该 Pod 为 I/O 密集型 Pod；若式 (4.2a) 与式 (4.2c) 均满足，则进行式 (4.3b) 的判定，如果满足式 (4.3b)，则该 Pod 为 CPU 密集型，否则，该 Pod 为网络密集型 Pod；若式 (4.2b) 与式 (4.2c) 均满足，如果满足式 (4.3c)，则该 Pod 为 I/O 密集型 Pod，否则该 Pod 为网

络密集型 Pod。即根据待调度 Pod 所申请的某类型资源占据节点上该类型空闲资源的最大比重，来决定应用 Pod 类型。应用 Pod 类型的判定总结如下表4.2。

在实际场景中，虽然大多数情况下可以按照下表4.2进行应用 Pod 类型的判定，但还需注意,如果待调度 Pod 与某 Node 节点之间的资源关系对公式 (4.2a)(4.2b)(4.2c) 均不满足，那么该 Pod 即为普通类型 Pod。在进行节点打分时，无需为节点上特定资源类型分配较高权重, 但又因为在 Kubernetes 中 CPU 资源和内存资源被认定为计算资源，所以 CPU 和内存资源在节点分数中权重稍微比其余类型资源稍高一些。

表 4.2: Pod 类型判定对照表

| **Pod 申请资源与节点空闲资源关系** | **Pod 类型判定结果** |
|---|---|
| 仅满足公式 (4.2a) | 计算密集型 Pod |
| 仅满足公式 (4.2b) | I/O 密集型 Pod |
| 仅满足公式 (4.2c) | 网络密集型 Pod |
| 同时满足公式 (4.2a)(4.2b) | 满足公式 (4.3a)，则为 CPU 密集型 Pod |
| 同时满足公式 (4.2a)(4.2b) | 不满足公式 (4.3a)，则为 I/O 密集型 Pod |
| 同时满足公式 (4.2a)(4.2c) | 满足公式 (4.3b)，则为 CPU 密集型 Pod |
| 同时满足公式 (4.2a)(4.2c) | 不满足公式 (4.3b)，则为网络密集型 Pod |
| 同时满足公式 (4.2b)(4.2c) | 满足公式 (4.3c)，则为 I/O 密集型 Pod |
| 同时满足公式 (4.2b)(4.2c) | 不满足公式 (4.3c)，则为网络密集型 Pod |

本文所提出的 CLBP 节点调度策略基于 CPU 资源、内存资源、网络 I/O 资源、磁盘 I/O 资源和磁盘容量这五种资源指标进行节点过滤与节点打分操作，要从多资源指标维度评估 Node 节点负载情况，并且需要计算节点上每种资源的负载均衡程度。由于这几种资源指标的权重值应该根据 Pod 类型有所倾斜，因此需要定义一定的权重规则为按照 Pod 类型为 Pod 所需资源赋予不同权重。

接下来以 CPU 资源指标为例，计算某节点上 CPU 资源在该节点上的均衡度，计算过程如下。

1、假设在对 Pod 进行调度时, 通过节点过滤阶段所得到的 Filter-Node 集合中共有 N 个节点，即共有 N 个节点进入到节点优选阶段。假设某待调度 Pod 对 CPU 资源的请求量为 $R_{cpu}$，使用 $N_{i-cpu}$ 代表集群中节点编号为 i 的节点上当前 CPU 资源负载, 使用 $T_{i-cpu}$ 代表该节点 i 上 CPU 资源最大负载。接下来就可以按照公式4.4与公式4.5计算得出节点 i 上 CPU 资源使用率为 $P_{i-cpu}$, 以及这 N 个节点上平均 CPU

资源使用率 $Avg_{cpu}$。

$$P_{i-cpu} = \frac{N_{i-cpu}}{T_{i-cpu}} \tag{4.4}$$

$$Avg_{cpu} = \frac{\sum_{i=1}^{N} P_{i-cpu}}{N} \tag{4.5}$$

公式4.5中 $Avg_{cpu}$ 表示 Kubernetes 集群中所有节点上 CPU 资源的平均使用率，本文通过节点之间 CPU 资源使用率标准差来衡量集群中 CPU 资源均衡度。

2、假设待调度 Pod 对 CPU 资源的请求量为 $R_{cpu}$，并假设该应用 Pod 调度到节点 j(1≤j≤N) 上之后，节点 j 上 CPU 资源负载为 $N'_{j-cpu}$，此时节点 j 上 CPU 资源使用率为 $P'_{j-cpu}$，N 个节点上平均 CPU 资源使用率为 $Avg'_{cpu}$，公式如下。

$$N'_{j-cpu} = N_{j-cpu} + R_{cpu} \tag{4.6}$$

$$P'_{j-cpu} = \frac{N'_{j-cpu}}{T_{j-cpu}} \tag{4.7}$$

$$Avg'_{cpu} = \frac{\sum_{i=1}^{N} P_{i-cpu} - P_{j-cpu} + P'_{j-cpu}}{N} \tag{4.8}$$

3、假设目标 Pod 调度到节点 j 上时，使用集群中所有节点上 CPU 资源使用率标准差来表示此时集群中 CPU 资源负载之间的差异度 $D_{j-cpu}$，计算公式如下。

$$D_j = \sqrt{\frac{\sum_{i=1}^{N} \left(P_{i-cpu} - Avg'_{cpu}\right)^2 - \left(P_{j-cpu} - Avg'_{cpu}\right)^2 + \left(P'_{j-cpu} - Avg'_{cpu}\right)^2}{N}} \tag{4.9}$$

4、在调度目标 Pod 时，假设目标 Pod 被调度到节点 j 上时，计算此时节点 j 上 CPU 资源负载均衡度分数，即把节点 j 上 CPU 资源均衡度进行量化。其中 $D_{max}$ 代表集群中 CPU 资源负载最大差异度，$D_{min}$ 代表集群中 CPU 资源负载最小差异度。其中，当目标 Pod 调度到节点 j 上时，节点 j 上 CPU 资源负载均衡度分数表示为 $S_{j-cpu}$，计算公式如4.10，$S_{j-cpu}$ 范围为 0-10。

$$S_{j-cpu} = 10 - 10 * \frac{D_{j-cpu} - D_{min}}{D_{max} - D_{min}} \tag{4.10}$$

如果 $D_{j-cpu}$ 值越大，则说明，如果目标 Pod 调度到节点 j 上，此时集群中所有节点之间 CPU 资源负载差距较大。这不利于 Kubernetes 集群中 CPU 资源的均衡使用，此时节点 j 分数就会越低，不建议目标 Pod 调度到节点 j 上. 如果 $D_{j-cpu}$ 值越

小，则说明，如果目标 Pod 调度到了节点 j 上，此时集群中所有节点之间 CPU 资源负载差距较小。这保证了目标 Pod 调度完成后 Kubernetes 集群中 CPU 资源的均衡使用，此时节点 j 分数就越高，建议目标 Pod 调度到节点 j 上。

至此，在把目标 Pod 调度到具体节点上之前，在步骤 4 中已经得到每个节点 i 上 CPU 资源负载均衡度分数为 $S_{i-cpu}$ , 同理可以计算每个节点 i 上内存资源、网络 I/O 资源、磁盘 I/O 资源和磁盘容量的负载均衡度分数分别为 $S_{i-mem}$ 、 $S_{i-nio}$ 、 $S_{i-dio}$ 、 $S_{i-d}$。本文通过以下公式4.11计算基于多维资源指标的节点 i 资源均衡度，即节点优选阶段中节点 i 最终分数，分数范围为 0-10 之间。优选阶段中节点打分综合考虑了节点上 CPU 资源、内存资源、网络 I/O 资源、磁盘 I/O 资源和磁盘容量这五种资源的均衡度分数。基于集群资源均衡度的节点打分公式如4.11，其中，权重系数满足 $\sum_{i=1}^{5} W_i = 1(0 \leq W_i \leq 1)$。

$$S_i = W_1 * S_{i-cpu} + W_2 * S_{i-mem} + W_3 * S_{i-nio} + W_4 * S_{i-dio} + W_5 * S_{i-d} \tag{4.11}$$

公式4.11中，$W_1$、$W_2$、$W_3$、$W_4$ 和 $W_5$ 是节点 i 上 CPU 资源、内存资源、网络 I/O 资源、磁盘 I/O 资源和磁盘容量均衡度分数在节点 i 分数组成中的权重。其中需要在优选调度阶段最开始的时候为 Pod 分类，从而在节点打分时为特定资源类型赋予较高权重。在大多数负载均衡场景下，开发者通常会依据经验主义来设置权值参数。本文使用运筹学理论 AHP 层次分析法构建权重模型，从而确定权重参数 $W_1$、$W_2$、$W_3$、$W_4$ 和 $W_5$ 的值，并且 AHP 方法支持在部分依靠经验主义的基础上进行主观评价，进而设置权重参数值会更贴合实际。

AHP 层次分析法是一类对难以完全定量的问题进行决策的层次权重决策方法，其原理为先将待分析问题进行层次划分。即依据问题的性质和期望目标，把问题划分为各种组成因素，然后依据不同组成因素之间所存在的相关性和隶属关系把各个组成因素按照不同层次执行聚合操作，从而构建一个多层次分析模型，最终该问题就可以转化为最低层次因素相对最高层次因素重要程度的权值问题。其中，决策者可以通过已有历史经验对各个组成因素之间相对重要程度进行判定，建立 AHP 判断矩阵，之后通过最小二乘法或者方根法计算 AHP 判断矩阵特征向量，将其作

为原始问题中各组成因素权重系数值，最后还要对权重系数结果进行一致性检验。若通过检验，则说明权重系数结果合理、自洽，问题求解完成。

根据 AHP 层次分析法模型求解复杂问题的流程如图4.3所示,

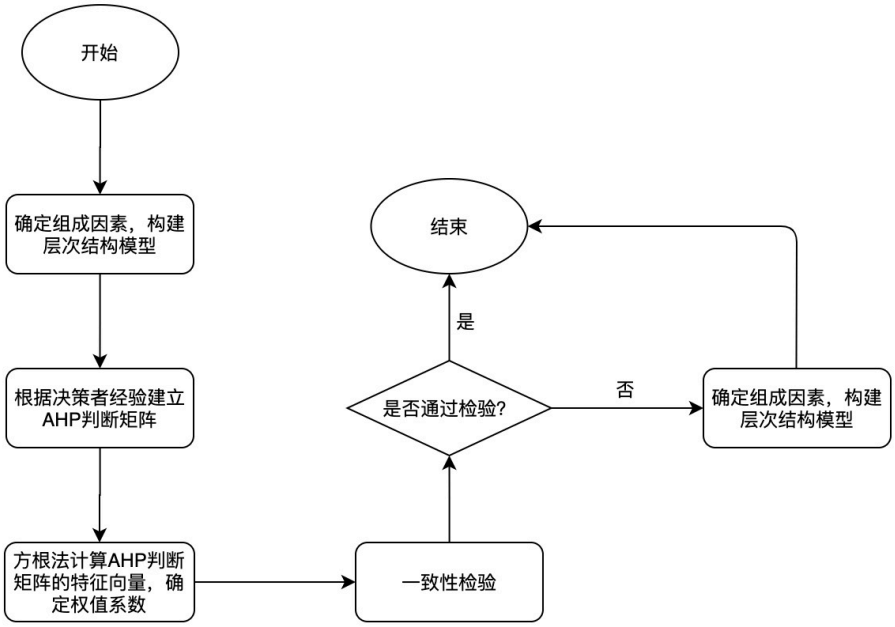



图 4.3: AHP 层次分析法流程

在本文中，层次分析模型中组成因素集合为: $E = \{e_1, e_2, \ldots e_n\}$, 其中 $e_i$ 表示节点上资源负载均衡度因子。$e_1$ 代表节点 i 上 CPU 资源均衡度分数，$e_2$ 代表节点 i 上内存资源均衡度分数，$e_3$ 代表节点 i 上网络 I/O 资源均衡度分数，$e_4$ 代表节点 i 上磁盘 I/O 资源均衡度分数，$e_5$ 代表节点 i 上磁盘空间资源均衡度分数。接下来构建 AHP 判断矩阵 (也被称为成对比较矩阵)，该 AHP 判断矩阵用于判定层次分析模型中 5 个负载均衡度因子之间对于目标问题的相对优劣程度。AHP 判断矩阵是本层所有因素与上一层某个因素之间相对重要程度的比较，矩阵中元素 $e_{ij}$ 表示相对于目标问题，第 i 个因素与第 j 个因素之间重要程度的比较结果。本文中采用 $1 \sim 9$ 标度法，将 AHP 判断矩阵中 5 个资源负载均衡度因子的相对重要程度进行量化，其形式如下,

$$E = \begin{bmatrix} e_{11} & e_{12} & e_{13} & e_{14} & e_{15} \\ e_{21} & e_{22} & e_{23} & e_{24} & e_{25} \\ e_{31} & e_{32} & e_{33} & e_{34} & e_{35} \\ e_{41} & e_{42} & e_{43} & e_{44} & e_{45} \\ e_{51} & e_{52} & e_{53} & e_{54} & e_{55} \end{bmatrix} \tag{4.12}$$

其中，AHP 判断矩阵的标度定义如下表4.3,

表 4.3: AHP 矩阵标度定义表

| **标度 $e_{ij}$** | **标度 $e_{ij}$ 定义 (i,j 为负载均衡度因素)** |
|---|---|
| 1 | 前者因素 $e_i$ 与后者因素 $e_j$ 同等重要 |
| 3 | 前者因素 $e_i$ 与后者因素 $e_j$ 稍微重要 |
| 5 | 前者因素 $e_i$ 与后者因素 $e_j$ 明显重要 |
| 7 | 前者因素 $e_i$ 与后者因素 $e_j$ 非常重要 |
| 9 | 前者因素 $e_i$ 与后者因素 $e_j$ 极端重要 |
| 2，4，6，8 | 介于上述两个因素标度之间的中间量 |
| 倒数 | 前者因素 $e_i$ 相比后者因素 $e_j$ 的重要程度为 $e_{ij}$,<br>则后者因素 $e_j$ 相比后者因素 $e_i$ 的重要程度为 $e_{ji}$, 其中 $e_{ji} = 1/e_{ij}$ |

通过上述对 AHP 判断矩阵的介绍以及对矩阵标度的定义，接下来以 CPU 密集型 Pod 为例，对 AHP 判断矩阵中节点资源均衡度因子两两之间进行对比分析，从而构建对应 AHP 判断矩阵。判断矩阵构造结果如下表4.4所示,

表 4.4: AHP 判断矩阵中节点资源均衡度因子比较

| **因素** | **CPU** | **内存** | **网络 I/O** | **磁盘 I/O** | **磁盘空间** |
|---|---|---|---|---|---|
| **CPU** | 1 | 3 | 3 | 5 | 7 |
| **内存** | 1/3 | 1 | 2 | 4 | 7 |
| **网络 I/O** | 1/3 | 1/2 | 1 | 3 | 4 |
| **磁盘 I/O** | 1/5 | 1/4 | 1/3 | 1 | 3 |
| **磁盘空间** | 1/7 | 1/7 | 1/4 | 1/3 | 1 |

以矩阵形式表示 AHP 判断矩阵的话，可以表示为,

$$E = \begin{bmatrix} 1 & 3 & 3 & 5 & 7 \\ 1/3 & 1 & 2 & 4 & 7 \\ 1/3 & 1/2 & 1 & 3 & 4 \\ 1/5 & 1/4 & 1/3 & 1 & 3 \\ 1/7 & 1/7 & 1/4 & 1/3 & 1 \end{bmatrix} \tag{4.13}$$

根据上述 AHP 判断矩阵，使用方根法对 AHP 判断矩阵的权重向量进行计算，具体的计算过程和结果如下，

首先，按照行顺序对矩阵中行元素进行求积运算，然后再求 1/n 次幂。具体来说就是针对 AHP 判断矩阵中每一行所有元素标度值进行连乘操作，之后对该连乘操作结果开 n 次方根 (n 为 AHP 判断矩阵的维度)，从而得到向量 $W' = (w'_1, w'_2, \ldots, w'_n)^T$。其中，$w'_i$ 的计算公式为，

$$w'_i = n\sqrt{\prod_{j=1}^{n} e_{ij}} \tag{4.14}$$

然后，对向量 $W'$ 进行归一化处理，最后得到权重向量 $W = (w_1, w_2, \ldots, w_n)^T$。其中，$w_i$ 的计算公式为，

$$w_i = \frac{w'_i}{\sum_{i=j}^{n} w'_j} \tag{4.15}$$

通过以上计算得到权重向量 $W = (0.46, 0.26, 0.13, 0.10, 0.05)^T$, 向量 W 中每一个 $w_i$ 对应具体的资源均衡度因子。AHP 层次模型对问题的分析结果如下表4.5,

表 4.5: AHP 层次模型分析结果表

| 项 | 特征向量 | 权重系数 (%) | 最大特征值 | CI 值 |
|---|---|---|---|---|
| 指标 1 | 3.16 | 46.426 | | |
| 指标 2 | 1.796 | 26.382 | | |
| 指标 3 | 0.871 | 12.791 | 5.17 | 0.042 |
| 指标 4 | 0.684 | 10.054 | | |
| 指标 5 | 0.296 | 4.347 | | |

最后需要对分析结果执行一致性检验操作，本文中该 AHP 判断矩阵最大特征根 λ = 5.17。基于公式4.16对 CI 值进行计算，其值为 0.042，查询 RI 表得知对应 RI 值是 1.11，从而按照公式4.17进一步计算得到 CR 值为 0.038。因为 CR 值为 0.038, 小于 0.1, 所以 AHP 判断矩阵的结果逻辑自洽，通过一次性检验，上述计算过程所得到的权重系数向量有效。

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{4.16}$$

$$CR = \frac{CI}{RI} \tag{4.17}$$

以上实践是假定应用 Pod 为计算密集型 Pod 所计算得到的权重系数结果，即各类资源均衡度分数在节点 i 分数构成中所占的权重，此时公式4.11中权重系数向量为 $W = (0.46, 0.26, 0.13, 0.10, 0.05)^T$。同理，可以得到，当待调度应用 Pod 为内存型 Pod 时，公式4.11中的权重系数向量为 $W = (0.26, 0.46, 0.13, 0.10, 0.05)^T$。当待调度应用 Pod 为网络型 Pod 时，公式4.11中网络型 Pod 所对应的权重系数向量为 $W = (0.26, 0.13, 0.46, 0.10, 0.05)^T$。此外，需要注意，如果待调度 Pod 是一个普通 Pod, 即该 Pod 不是计算密集型 Pod、内存型 Pod 或者网络型 Pod, 此时权重系数向量为 $W = (0.30, 0.30, 0.20, 0.10, 0.10)^T$。至此，已经根据应用 Pod 类型完成了优选阶段中资源倾向性节点打分。在选定分数最高节点之后，目标 Pod 与目标 Node 将完成绑定，绑定信息通过 API Server 写入 Etcd 中，目标 Pod 完成调度，然后目标 Pod 在节点上部署、运行。

Pod 调度完成后，为了评估调度算法的性能，需要计算 Kubernetes 集群中每个节点上同一类型资源使用率之间的标准差。标准差越小，则表示该资源在集群中的使用越均衡，集群负载就越均衡。因此，本文使用不同节点上同一种类型资源使用率的标准差作为 Kubernetes 自定义调度算法性能评价标准。

## 4.5 动态调整 Pod 资源限额

通常情况下用户为保证应用 Pod 能够稳定运行，会为应用 Pod 申请较多资源，但是大多数时候应用 Pod 实际运行所消耗的资源远远低于用户为该 Pod 所申请的计算资源，这会导致资源浪费。除此之外，由于频繁创建、删除 Pod，Kubernetes 集群中不同节点上会留下大量资源碎片，新创建、新部署的 Pod 可能无法使用这部分资源碎片。此外，Kubernetes 默认调度机制在完成 Pod 调度之后，就不再更新、调整运行中 Pod 的计算资源。但是 Pod 在运行过程中资源需求会动态变化，而 Kubernetes 默认调度机制是一种静态调度，不能实现动态调整 Pod 资源限额。并且，Pod 运行过程中，内存资源使用率可能不断升高，此时如果不提升该 Pod 内存资源限额，可能会触发 OOM(Out Of Memory) 问题，导致该 Pod 被删除，服务终止。

因此，本文从节约集群资源、管理集群资源碎片、提升集群资源负载均衡以及保障服务稳定性的角度出发，使用 Cgroups 控制组机制更新 Pod, 调整 Pod 所申请 CPU、内存以及磁盘资源的资源限额。在 Pod 运行期间，Kubernetes 支持使用 Cgroups 控制组机制动态修改 Pod 资源参数。在 Docker 容器运行场景中，开发者可以在/sys/fs/cgroup 控制组下子系统中根据 Container ID 直接调整 CPU、内存以及磁盘资源限额；而在 Kubernetes 集群中，用户在/sys/fs/cgroup 控制组下子系统中需要通过 kubepods.slice 调整资源限额。接下来以动态调整 Pod 所使用的 CPU 资源限额为例，使用 Cgroups 控制组机制调整运行中 Pod 的 CPU 资源限额的过程如下，

1. 按照一定周期 T 从 Prometheus 中查询目标 Pod 的 CPU 资源使用量数据，历史 CPU 资源使用量分别为 $U_T$、$U_{2T}$、$U_{3T}$、$U_{4T}$、$U_{5T}$。为避免频繁调整资源限所带来的额外开销，将 T 设置为 1min，即每隔一分钟采集目标 Pod 的 CPU 资源使用量数据，收集当前点时刻前五分钟内该 Pod 的 CPU 资源用量数据。

2. 计算目标 Pod 的期望 CPU 使用量 $H_c$。为避免目标 Pod 申请资源远大于目标 Pod 实际使用资源, 本文基于所采集到的资源使用量历史数据计算平均值, 然后结合自定义资源弹性系数 $\lambda$ 即可得到 $H_c$。如果前五分钟内 Pod 平均资源使用量上升，则使用公式4.18提升 CPU 使用量限额；如果前五分钟内 Pod 平均资源使用量降低，则使用公式4.19降低 CPU 使用量限额。公式如下，其中使用 $\lambda$ 表示资源弹性系数，本文取 $\lambda$ 为 0.2, 也可以根据实际场景调整。

$$H_c = \frac{\sum_{i=1}^{5} U_{i*T}}{5} * (1 + \lambda) \tag{4.18}$$

$$H_c = \frac{\sum_{i=1}^{5} U_{i*T}}{5} * \frac{1}{(1 + \lambda)} \tag{4.19}$$

3. 通过 Kubernetes API 查询目标 Pod 的 uid 信息，进而获取 Pod 中业务容器 id 信息，那么就可以得到待更新 Pod 中业务容器所对应的 Cgroups 控制组路径，即/sys/fs/cgroup/cpu/kubepods.slice/kubepods-pod<pod_uid>.slice/docker-<container_id>.scope。

4. 得到目标 Pod 所对应的 Cgroups 控制组路径之后，就可以通过 echo 命令直接把公式4.18或者4.19计算得出的 CPU 资源限额 $H_c$ 写入到 Cgroups 控制文件中，从而实现了对目标 Pod 中容器所使用的 CPU 资源限额的动态调整。

同理，可以通过收集目标 Pod 其余类型资源使用量的历史数据，计算目标 Pod 对某类型资源的期望使用量，进而通过目标 Pod 所对应的 Cgroups 控制文件来进行内存资源、磁盘 I/O 资源的动态限额调整。Cgroups 控制组中内存资源相关 Cgroups 子系统路径为/sys/fs/cgroup/memory , 磁盘 I/O 资源相关 Cgroups 子系统路径为/sys/fs/cgroup/blkio。在实际使用场景中，可以在 Kubernetes 集群中为动态调整 Pod 资源限额方案设置定期执行的触发方式，即每隔一段时间 (默认值为 10 分钟) 就对集群中各 Node 节点上资源占用较多的前十个 Pod 动态调整资源限额，以缓解 Node 节点负载压力，实现集群资源的均衡利用。

## 4.6 本章小结

本章针对 Kubernetes 默认调度机制所存在的缺陷, 构建 Kubernetes 扩展调度器，提出一种基于集群负载均衡度的调度策略，并且还提出一种动态调整 Pod 资源限额方案。首先，介绍 Kubernetes 调度器工作流程，然后对比分析目前 Kubernetes 扩展调度器方案。接下来详细阐述基于 Scheduling Framework 构建 Kubernetes 扩展调度器的流程，之后设计基于集群节点负载均衡度的调度算法并具体阐述其设计理念。最后，详细说明本文所提出的动态调整 Kubernetes 基本部署单位 Pod 资源限额的方案，以及该方案的触发方式。

# 第五章　实验结果与分析

## 5.1　实验环境搭建

本章基于阿里云 2018 年公开容器数据集 Cluster-trace-v2018[①] 对本文所提出的 ARIMA-GRU 组合负载预测模型进行训练、构建与调优。由于 [43][77–80] 等研究工作在 Kubernetes 弹性伸缩机制研究与调度机制研究中均使用模拟仿真实验来验证所提出的优化方案，因此本文为保证实验效果的直观性，也通过模拟仿真实验来对本文所改进的弹性伸缩机制与动态调度机制进行验证。

本文需要在两个 Kubernetes 集群上进行实验，以此构建对比实验，将其中一个 Kubernetes 集群作为实验组，将另一个 Kubernetes 集群作为对照组。实验组 Kubernetes 集群使用本文中自构建的多个模块进行搭建，而对照组中 Kubernetes 集群则基于官方默认模块进行搭建。为控制变量，两个集群基础配置信息需要一致。两个集群的集群配置如下，下表5.1展示了集群节点配置信息；下表5.2展示了集群版本以及 Docker 引擎版本等信息。

表 5.1: 集群节点配置信息

| 节点 | CPU 核心 | 内存容量 | 网络带宽 | 磁盘吞吐量 | 磁盘容量 |
|---|---|---|---|---|---|
| Master | 8 | 16GB | 50Mbps | 160MB/s | 200GB |
| Node1 | 4 | 8GB | 30Mbps | 120MB/s | 100GB |
| Node1 | 4 | 8GB | 30Mbps | 120MB/s | 100GB |
| Node1 | 4 | 8GB | 30Mbps | 120MB/s | 100GB |

表 5.2: 软件环境

| 软件名称 | 软件版本 |
|---|---|
| 操作系统 | CentOS 7.6 64 位 |
| Kubernetes | V1.20.0 |
| Docker | V20.10.10 |

[①]https://github.com/alibaba/clusterdata/blob/v2018/cluster-trace-v2018/trace_2018.md

## 5.2 负载预测实验

### 5.2.1 实验设计

使用本文所提出的自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型对集群负载进行预测时，保证预测准确度非常重要。本文使用多组数据训练 ARIMA-GRU 组合预测模型，并使用多种预测模型评价指标评价该预测模型。

此负载预测实验的实验数据主要来自 2018 年阿里云公开容器数据集 Cluster-trace-v2018[②]。该数据集包含阿里云大规模容器集群 8 天内在 4000 台机器上的容器数据，阿里云对集群中所有节点、Pod、集群中其他资源对象的负载情况与资源请求情况等进行收集。其中包含节点上资源总量信息、资源使用量信息，比如节点 CPU 资源总量、CPU 利用率、内存资源总量、内存利用率、网络带宽负载等数据；应用 Pod 资源请求量、资源使用量信息，比如 CPU 请求量、CPU 使用量、网络带宽占用等信息。阿里云数据集 Cluster-trace-v2018 的数据体量很大，解压缩前数据集体量为 48G, 解压缩后数据集体量为 280G，本文选取数据集中部分数据进行实验。阿里云 cluster-trace-v2018 公开数据集包含 6 张数据表，分别用于存储容器、节点和批工作负载的数据信息。对 cluster-trace-v2018 数据集中的数据表介绍如下表5.3,

表 5.3: cluster-trace-v2018 数据表说明

| 数据表名称 | 数据存储说明 |
|---|---|
| machine_meta.csv | 存储每台机器节点上元信息与活动信息 |
| machine_usage.csv | 存储每台机器节点上 CPU、内存等用量信息 |
| container_meta.csv | 存储集群中所有容器元信息与活动信息 |
| container_usage.csv | 存储集群中每个容器的 CPU、内存等用量信息 |
| batch_instance.csv | 存储批处理工作负载信息 |
| batch_task.csv | 存储批处理工作负载中每个节点实例信息 |

针对负载预测实验，本文基于 Cluster-trace-v2018 数据集中 container_usage.csv 数据表数据，即阿里云容器集群中节点上容器资源用量数据作为 ARIMA-GRU 模

[②]https://github.com/alibaba/clusterdata/blob/v2018/cluster-trace-v2018/trace_2018.md

型数据集。集群负载数据的存储格式如下表5.4所示，

表 5.4: 集群负载数据格式表

| 数据列名 | 标度数据类型 | 数据字段说明 |
|---|---|---|
| container_id | string | 容器的 ID 信息，每个容器具有独一无二的 UID |
| machine_id | string | 承载容器运行的机器节点 uid |
| time_stamp | double | 时间戳 |
| cpu_util_percent | bigint | 容器对宿主机节点上 CPU 资源的使用率，取值范围为 [0, 100] |
| mem_util_percent | bigint | 容器对宿主机节点上内存资源的使用率，取值范围为 [0, 100] |
| net_in | double | 传入网络数据包的数量，归一化处理后，其取值范围为 [0, 100] |
| net_out | double | 发出网络数据包的数量，行归一化处理后，其取值范围为 [0, 100] |
| disk_io_percent | double | 容器对宿主机节点上磁盘 I/O 资源的使用率，取值范围为 [0, 100] |

基于 container_usage.csv 数据表中 70% 数据构建实验训练集，基于余下 30% 数据构建实验测试集。因为 ARIMA-GRU 组合预测模型的预测结果与真实值之间可能会存在误差，因此该预测实验使用常用的平均绝对误差、平均绝对百分比误差与均方根误差这三个评估指标来衡量本文组合预测模型的预测准确度，从而不断反馈、迭代、优化 ARIMA-GRU 组合预测模型，使其预测效果达到相对较优水平。

### 5.2.2 实验结果与分析

从 container_usage.csv 数据表中挑选 6 组实验数据，每组数据量为 1w 条，基于其中 70% 数据构建训练集，并且基于余下 30% 数据构建测试集。首先，分别使用 ARIMA 模型、GRU 模型在训练集上进行单一模型预测，然后使用本文所提出的 ARIMA-GRU 组合预测模型在训练集中进行组合模型预测，并且在训练过程中不断优化预测模型。之后，在测试集上分别计算 ARIMA 模型、GRU 模型与 ARIMA-GRU 组合预测模型在 6 组测试数据上预测结果与真实值之间的差异程度，使用 MAE、MAPE 与 RMSE 表示预测差异度。针对第一组数据集的预测结果评价如下表所示，从下表5.5和下表5.6可以总结出，在第一组数据集上，本文 ARIMA-GRU 组合预测模型相比单一 ARIMA 模型与 GRU 模型均有较好预测精度，并且组合模型对 CPU 资源和内存资源用量的预测准确率平均达到 93% 以上，预测准确度较高。

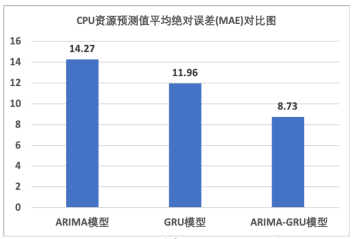
表 5.5: CPU 资源预测值准确度评价表

| 预测模型 | ARIMA | GRU | ARIMA-GRU |
|---|---|---|---|
| 平均绝对误差 (MAE) | 13.28 | 11.65 | 7.66 |
| 平均绝对百分比误差 (MAPE) | 12.72 | 9.43 | 5.18 |
| 均方根误差 (RMSE) | 15.26 | 10.78 | 7.92 |

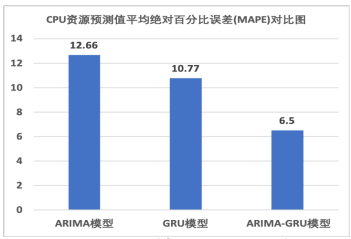
表 5.6: 内存资源预测值准确度评价表

| 预测模型 | ARIMA | GRU | ARIMA-GRU |
|---|---|---|---|
| 平均绝对误差 (MAE) | 14.26 | 12.29 | 5.98 |
| 平均绝对百分比误差 (MAPE) | 13.69 | 10.04 | 4.93 |
| 均方根误差 (RMSE) | 18.66 | 12.38 | 8.77 |

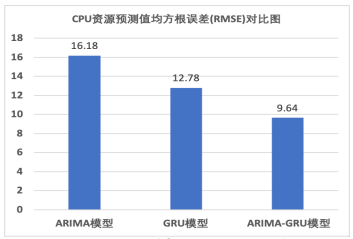
为避免单组实验的偶然性，本实验使用 3 种不同预测模型在 6 组实验数据上分别预测，对 6 组数据平均预测结果的评价如下图5.1，



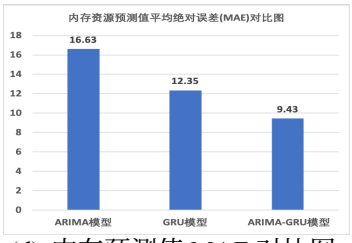

(a) CPU 预测值 MAE 对比图



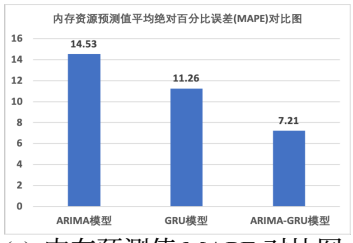

(b) CPU 预测值 MAPE 对比图



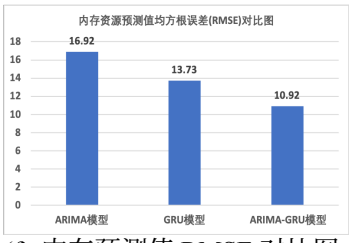

(c) CPU 预测值 RMSE 对比图




(d) 内存预测值 MAE 对比图




(e) 内存预测值 MAPE 对比图




(f) 内存预测值 RMSE 对比图

图 5.1: 多组实验下 CPU 与内存资源预测平均值结果评价图

根据上图5.1中一系列预测误差对比图可知，在 6 组数据上的预测实验中，单一预测模型比如 ARIMA 模型、GRU 模型的预测误差均比本文所提出的 ARIMA-GRU 组合预测模型的预测误差大。由于本文所提出的 ARIMA-GRU 组合预测模型能分别对集群负载时间序列中线性部分与非线性部分进行预测，因此该组合模型

对 CPU 资源使用量以及对内存资源使用值的预测效果都明显好于单一模型。所以本文使用 ARIMA-GRU 组合预测模型预测集群负载有一定实际意义，并且之后能为集群中弹性伸缩机制提供支持。

在云计算负载预测场景中，选择预测模型时不仅要考虑预测模型的预测精准度，还需要考虑预测模型训练时间。本文使用 ARIMA 模型、GRU 模型、本文所提出的 ARIMA-GRU 组合预测模型分别在上述 6 组实验数据上进行预测，统计 6 组预测实验中每个预测模型的平均训练时间，三种预测模型基于容器集群负载数据集的平均训练时间如下表5.7所示。

表 5.7: 统计不同预测模型的训练时间

| **预测模型** | **预测模型平均训练时间** |
|---|---|
| ARIMA 模型 | 132s |
| GRU 模型 | 309s |
| ARIMA-GRU 模型 | 368s |

从上表5.7可以看出, ARIMA 模型训练时间最短, GRU 模型训练时间比 ARIMA 模型长些，其中 ARIMA-GRU 模型训练时间最长；但是由于 ARIMA-GRU 组合模型的预测精度最高，因此 ARIMA-GRU 组合模型综合表现较好。并且在 Kubernetes 集群上负载波动幅度相对较小的场景中，ARIMA-GRU 组合预测模型一旦训练完成，在未来一段时间内是可以一直使用的，通过收集过去一段时间内 Kubernetes 集群上负载数据就可以训练本文所提出的 ARIMA-GRU 组合预测模型。综合考虑之下，ARIMA-GRU 组合模型适合 Kubernetes 集群负载预测工作。

## 5.3 Kubernetes 弹性伸缩实验

### 5.3.1 实验设计

本实验主要检验本文所提出的基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略的运行效果。实验内容包含两个方面，一方面是测试自定义弹性伸缩策略能否在集群中实现正常 Pod 副本扩缩容功能，即检验该策略是否能正常工

作。另一方面是验证自定义弹性伸缩策略与 HPA 控制器所实现的默认伸缩机制相比，该策略能否减少服务响应时间、提升服务质量以及节约集群资源，即检验该策略的有效性。此外，考虑到本文 Kubernetes 版本中 HPA 控制器仅能为 Pod 设置 CPU 资源阈值与内存资源阈值，因此为避免与 HPA 控制器中现有默认扩缩容策略冲突，本文自构建弹性伸缩策略也基于 CPU 与内存资源计算 Pod 副本期望数量。

具体实验流程如下，

1、构建一个 Web 应用，该 Web 应用在运行时会消耗一定的集群节点资源，把该 Web 应用封装在 Docker 镜像中，并将该 Web 应用以 Pod 应用的形式运行在 Kubernetes 集群中。并且为该 Pod 配置 Deployment 控制器，通过 Service 把 Pod 中该 Web 服务暴露出来。

2、分别在实验组 Kubernetes 集群、对照组 Kubernetes 集群中同时部署、运行该 Web 应用 Pod，其中实验组集群使用本文所提出的弹性伸缩策略实现 Pod 副本的扩缩容，对照组集群则默认基于 HPA 控制器实现 Pod 副本的扩缩容。

3、为 Web 应用 Pod 构建水平伸缩控制器 (HPA)，在 HPA Yaml 配置文件中，将 Web 应用 Pod 最小副本数 minReplicas 设置为 2，最大副本数 maxReplicas 设置为 8，Kubernetes 集群将在此副本数量范围内对该 Pod 进行扩缩容。Kubernetes 集群中水平伸缩控制器 (HPA) 的扩容时间间隔为 3min，缩容时间间隔为 5min。为提升实验效率，本文通过修改集群中 Controller Manager 启动参数来更改扩缩容冷却时间。其中，通过调整 horizontal-pod-autoscaler-downscale-delay 参数可以修改扩容冷却时间，通过调整 horizontal-pod-autoscaler-upscale-delay 参数可以修改缩容冷却时间。实验所构建的 Web 应用 Pod 在 30s 内可以完成部署运行或者删除清理操作，所以本文将水平伸缩控制器 (HPA) 扩缩容时间间隔均设置为 30s。

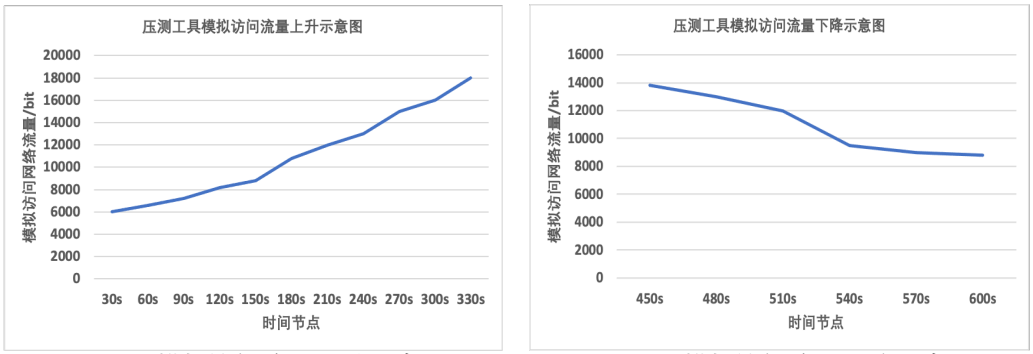
4、通过 Kubernetes Service 所暴露出来的服务端口，使用压测工具 Siege 模拟访问 Web 应用 Pod。在本实验中设置总访问时间为 10 分钟，总访问时长即为 600s，将 10 分钟模拟访问分为 3 个阶段。第一阶段是访问请求逐渐增加，第二阶段是访问请求保持平稳，第三阶段是访问请求逐渐下降，从而在不同时间节点模拟触发

Pod 副本的扩缩容。因为本实验将 Web 应用 Pod 副本的扩缩容间隔设置为 30s，因此将弹性伸缩模块中预测模型的预测步长设置为 30s，即等待上一次 Pod 副本扩缩容完成后，再进行下一次的负载预测与 Pod 副本预测值的计算。

5、通过统计不同时间节点下 Pod 副本数量以及该 Web 应用的服务响应时间，来评估基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略的运行效果。

### 5.3.2 实验结果与分析

本实验分为两组,第一组实验在实验组 Kubernetes 集群上进行,使用本文所提出的基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略来实现 Kubernetes 最小管理单位 Pod 的扩缩容操作。第二组实验在对照组 Kubernetes 集群上进行，使用基于水平伸缩控制器 (HPA) 的默认伸缩机制实现扩缩容操作。为保证实验的完整性，需要按照实验设计中步骤 4，通过在不同时间节点下调整压测工具 Siege 的模拟访问频率，从而在模拟触发实验组集群与对照组集群中的扩缩容操作。实验过程中压测工具 Siege 的模拟访问流量数据如下图5.2所示，在实验设计环节中需要为 Siege 工具提前设计好流量访问策略。




(a) Siege 模拟访问流量上升示意图　(b) Siege 模拟访问流量下降示意图

图 5.2: 压测工具 Siege 流量访问策略示意图

除了通过上图5.2中流量策略模拟触发集群中 Pod 副本的扩缩容过程，本实验还要模拟 Pod 稳定处理服务的过程，所以在 330s 到 420s 时间段内的模拟访问流量会比较平稳。通过采集压测工具的流量数据，得知在 330s，360s，390s，420s 位置压测工具 Siege 的模拟访问流量分别为 18000bit，18150bit，18000bit，18150bit。

使用压测工具 Siege 分别对实验组 Kubernetes 集群与对照组 Kubernetes 集群中的 Web 应用 Pod 进行模拟访问，两组实验中 Web 应用 Pod 副本的弹性伸缩结果以及服务响应时间如下图5.3所示，

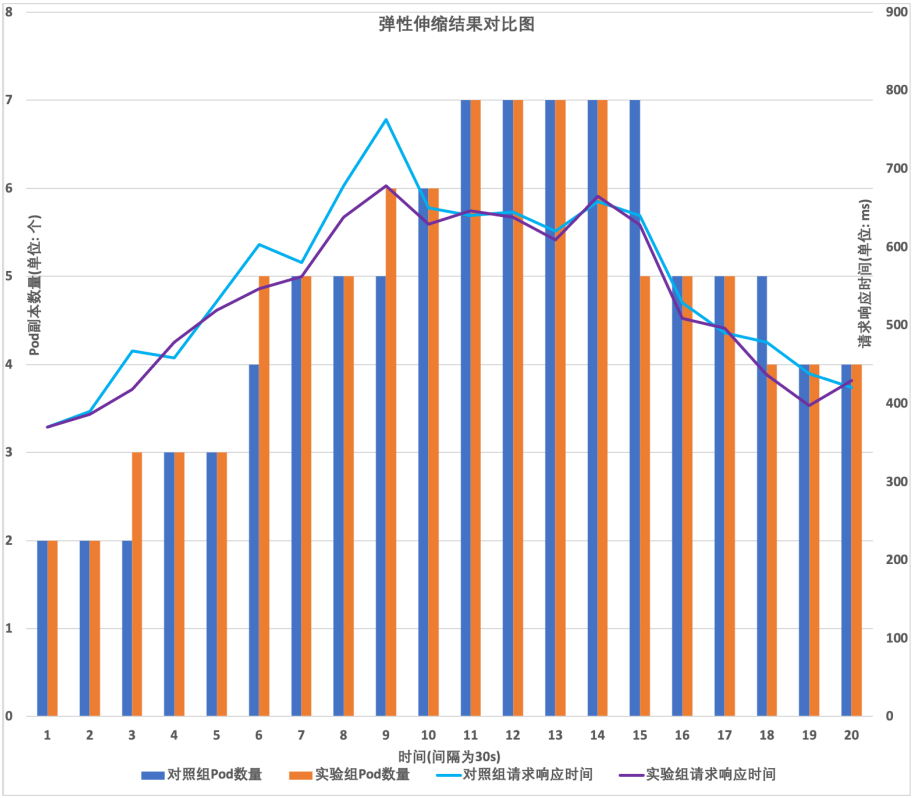



图 5.3: 弹性伸缩实验结果对比图

此外，实验组与对照组 Kubernetes 集群中 Pod 副本的数量变化趋势如下图5.4所示。通过观察弹性伸缩实验结果图5.3与 Pod 副本数量变化示意图5.4可知，实验组 Kubernetes 集群能够在负载来临之前基于自构建弹性伸缩策略提前进行 Pod 副本的扩缩容。而对照组 Kubernetes 集群中基于水平伸缩控制器 (HPA) 的被动式响应扩缩容策略，只有在负载到来的时候，才能被触发，所以对照组 Kubernetes 集群的扩缩容操作存在滞后性。在 90s 到 330s 期间，压测工具对 Web 应用 Pod 的访问量不断上升，在第 90s、180s、270s 处，实验组 Kubernetes 集群多次提前进行扩容。在这几个时间节点下，访问负载已经到来，而对照组 Kubernetes 集群这个时候才进行扩容，显然此时服务质量已经受到了影响，并且服务响应时间相比实验组集群中服务响应时间明显耗时更长。在这几个时间点下，实验组 Kubernetes 集群中服务

质量更高，服务响应速度更快，保证了服务的访问。在 450s 到 570s 之间，压测工具对 Web 应用 Pod 的访问量不断下降，而实验组 Kubernetes 集群在 450s、540s 就提前进行了缩容操作。因为在该时间段内网络访问量已经下降，所以即使在实验组 Kubernetes 集群中减少 Pod 副本数量，依然能够保证服务响应时间与服务质量，此时节省了集群资源。综上所述，实验组 Kubernetes 集群中基于 ARIMA-GRU 负载预测模型与 HPA 配置的弹性伸缩策略能够在集群负载到来之前，提前进行 Web 应用 Pod 的扩缩容，这在一定程度上提升了集群资源利用率，并且在保证服务质量的同时减少了不必要的资源消耗。

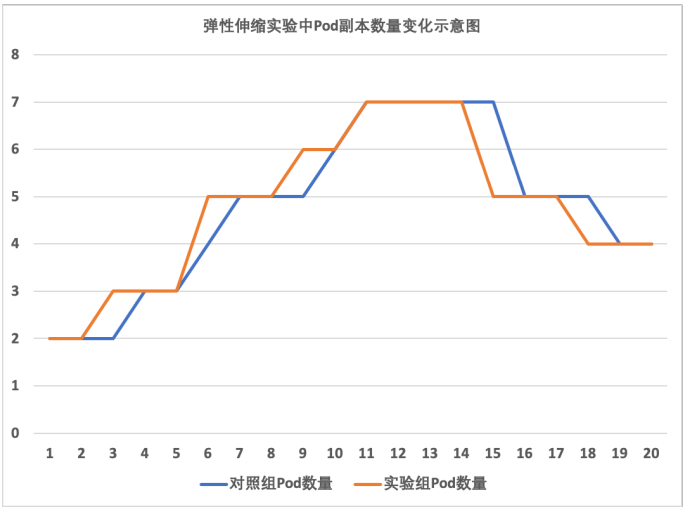



图 5.4: 弹性伸缩实验中 Pod 副本数量变化示意图

## 5.4 Kubernetes 调度实验

### 5.4.1 实验设计

本实验用于验证本文所提出的基于集群负载均衡度的调度算法的有效性，该实验分为两组。一组实验是在实验组 Kubernetes 集群中进行，使用本文所提出的基于集群负载均衡度的调度算法进行调度。另一组在对照组 Kubernetes 集群中进行，使用 Kubernetes 集群默认调度策略进行调度。为控制实验变量，两个集群中节点配置、任务类型与资源请求量完全一致，仅调度算法存在差别。具体实验步骤如下，

1、本文设计 3 种不同类型 Pod 进行实验，分别是计算密集型 Pod、I/O 密集型

Pod、网络密集型 Pod。3 种类型 Pod 对资源的需求如下表5.8所示，其中 Pod1 为计算密集型 Pod；Pod2 为网络密集型 Pod；Pod3 为 I/O 密集型 Pod。

表 5.8: 设计不同类型 Pod

| Pod 名称 | CPU 请求量 | 网络带宽请求量 | 磁盘 I/O 请求量 | 副本数量 |
|---|---|---|---|---|
| Pod1 | 600m | 2Mbps | 10MB/s | 5 |
| Pod2 | 200m | 5Mbps | 10MB/s | 5 |
| Pod3 | 200m | 2Mbps | 20MB/s | 5 |

2、通过编写 shell 脚本，定义不同类型 Pod 对宿主机资源的请求量。在 Dockerfile 中添加 shell 脚本，并构建容器镜像，该容器镜像之后用于构建计算密集型 Pod、网络密集型 Pod 和 I/O 密集型 Pod。

3、构建 Deployment 控制器对象，在 Yaml 配置文件中声明对步骤 2 中所构建容器镜像的使用，按照步骤 1 中不同类型 Pod 对资源的需求定义 Pod 资源请求量，并将 Deployment 中 Pod 副本数设置为 5。对这 3 类 Pod，共 15 个 Pod 依次进行编号，Pod1 副本编号为 1-5，Pod2 副本编号为 6-10，Pod3 副本编号为 10-15。

4、分别在实验组 Kubernetes 集群与对照组 Kubernetes 集群中部署、运行 3 种不同类型 Pod，收集两个集群中不同节点上的资源使用信息，进而计算两个 Kubernetes 集群的整体负载均衡度，评估本文自构建调度算法的有效性。

此外，为验证本文所提出的动态调整 Pod 资源限额方案的可用性与有效性，构建 Web 应用 Pod，包含 3 个 Pod 副本。该 Web 应用 Pod 对资源的请求量分别为 CPU600m，内存 400m，网络带宽请求量 5Mbps，磁盘 I/O 请求量 10MB/s，磁盘空间 10G。并且，分别在实验组与对照组 Kubernetes 集群中运行该 Pod，并且使用压测工具 Siege 模拟对该 Web 应用 Pod 的访问，访问时长设置为为 10min，总实验时长即为 600s。其中，Siege 在 30s 到 330s 内模拟增长的访问流量，在 330s 到 450s 模拟平稳的访问流量，在 450s 到 600s 模拟下降的访问流量。其中，设置动态调整 Pod 资源限额的时间周期 T 为 20s，资源弹性系数 $\lambda$ 为 0.2。在动态调整 Web 应用 Pod 资源限额的过程中，观察服务响应时间，如果实验组中服务响应时间与对照组中服务响应时间差异较小，则说明本文所提出的动态调整 Pod 资源限额在保证服务访问质量的同时提升了集群资源利用率。

### 5.4.2 实验结果与分析

在使用基于集群负载均衡度自定义调度算法的实验组 Kubernetes 集群中与使用默认调度算法的对照组 Kubernetes 集群中分别对 3 种不同类型的 Pod 进行调度，然后统计每个 Kubernetes 集群中资源的使用情况。本文通过计算集群中所有节点之间 CPU 资源、内存资源、网络 I/O 资源、磁盘 I/O 资源、磁盘容量资源使用率之间的标准差来衡量集群负载的均衡程度。

在对照组 Kubernetes 集群中，对 3 种不同类型 Pod 进行调度。其中，编号为 1，4，5，7，10，11 的 Pod 与 Node1 节点进行了绑定；编号为 2，6，12，14 的 Pod 与 Node2 节点进行了绑定；编号为 3，8，9，13，15 的 Pod 与 Node3 节点进行了绑定。等到所有 Pod 调度完成后，每个节点上 CPU 资源、内存资源、网络 I/O 资源、磁盘 I/O 资源和磁盘容量资源的使用情况如下表5.9所示。

表 5.9: 对照组 Kubernetes 集群调度结果表

| 节点 | Pod | CPU 用量 | 内存用量 | 网络带宽占用 | 磁盘 I/O 占用 | 磁盘空间占用 |
|---|---|---|---|---|---|---|
| Node1 | 1,4,5,7,10,11 | 63.6% | 33.8% | 62.9% | 63.2% | 32.4% |
| Node2 | 2,6,12,14 | 34.3% | 23.9% | 37.6% | 64.6% | 23.5% |
| Node3 | 3,8,9,13,15 | 42.2% | 27.7% | 53.8% | 73.1% | 27.4% |

在实验组 Kubernetes 集群中，对 3 种不同类型 Pod 进行调度。其中，编号为 3，7，10，12，14 的 Pod 与 Node1 节点进行了绑定；编号为 2，4，5，8，13 的 Pod 与 Node2 节点进行了绑定；编号为 1，6，9，11，15 的 Pod 与 Node3 节点进行了绑定。等到所有 Pod 调度完成后，集群中资源使用情况如下表5.10所示。

表 5.10: 实验组 Kubernetes 集群调度结果表

| 节点 | Pod | CPU 用量 | 内存用量 | 网络带宽占用 | 磁盘 I/O 占用 | 磁盘空间占用 |
|---|---|---|---|---|---|---|
| Node1 | 3,7,10,12,14 | 31.6% | 25.1% | 56.1% | 65.3% | 29.4% |
| Node2 | 2,4,5,8,13 | 57.3% | 30.2% | 47.8% | 63.4% | 28.5% |
| Node3 | 1,6,9,11,15 | 33.9% | 26.7% | 48.6% | 70.7% | 24.8% |

实验组 Kubernetes 集群中使用了基于集群负载均衡度的调度策略，调度时会考虑多种资源类型，并且还在节点优选过程中还会根据 Pod 类型对节点进行资源倾向性打分。因此调度完成后，实验组集群中所有节点间各种资源的使用率比对照组集群中所有节点间各种资源的使用率更加均衡。

上述实验中,所有 Pod 调度完成后,对照组 Kubernetes 集群与实验组 Kubernetes 集群中不同节点之间 CPU 资源、网络 I/O 资源、磁盘 I/O 资源使用率标准差对比结果如下图 5.5所示,

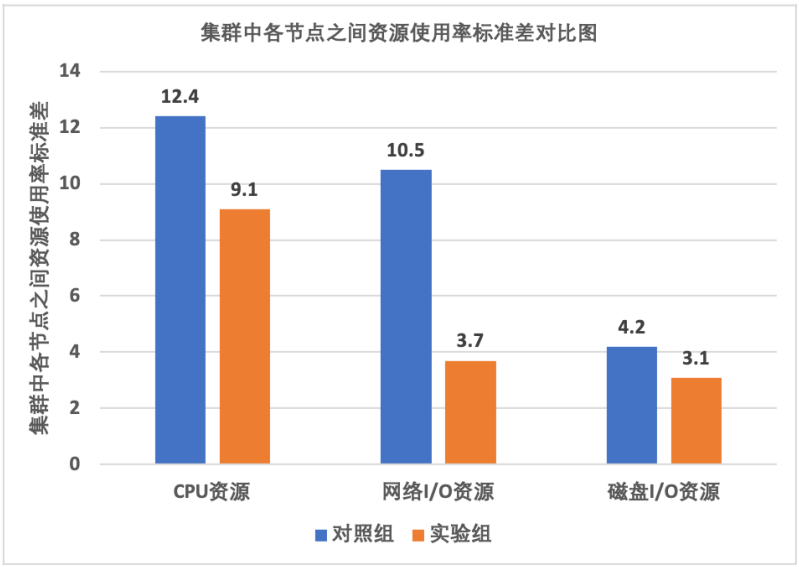
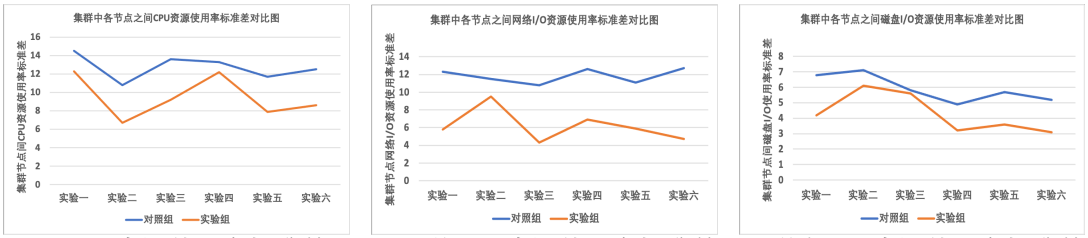



图 5.5: 集群中各节点之间资源使用率标准差对比图

如上图5.5所示，通过计算对照组 Kubernetes 集群与实验组 Kubernetes 集群中不同节点上同一种资源使用率之间的标准差，可得实验组 Kubernetes 集群中不同节点之间 CPU 资源、网络带宽资源、磁盘 I/O 资源使用率的标准差均小于对照组 Kubernetes 集群中对应资源使用率之间的标准差。这说明在使用基于负载均衡度的调度算法进行调度时，实验组 Kubernetes 集群中不同节点上 CPU、网络 I/O、磁盘 I/O 资源的使用率更均衡一些。

为避免实验的偶然性，重复该实验 6 次，调度完成后对照组和实验组集群中各节点之间的资源利用率标准差对比结果如下图5.6所示,

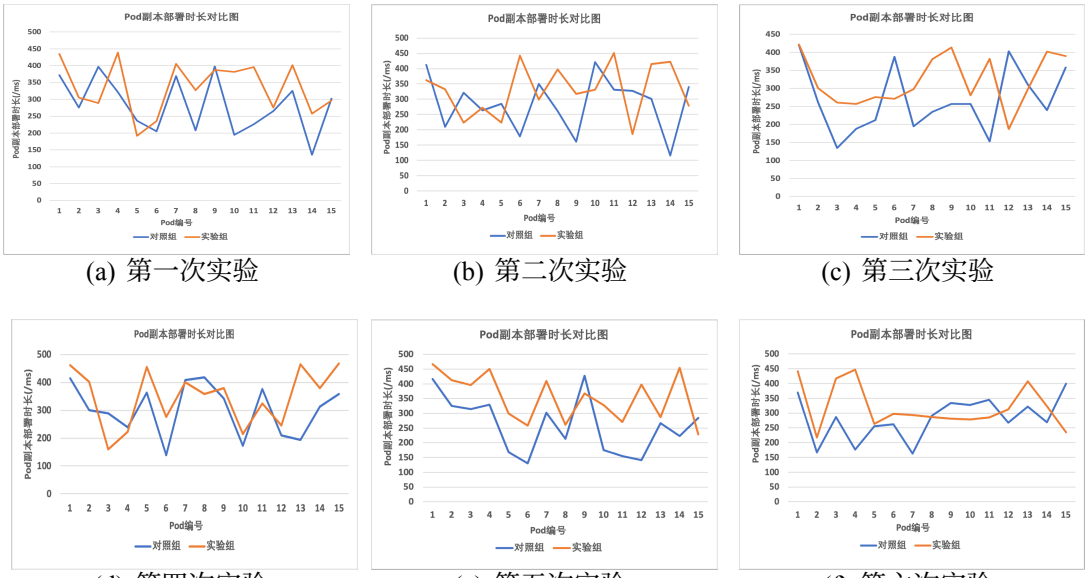



(a) CPU 资源使用率标准差 (b) 网络 I/O 资源使用率标准差 (c) 磁盘 I/O 资源使用率标准差

图 5.6: 多组实验下集群中各节点之间资源使用率标准差对比图

上图5.6中多次调度实验结果表明实验组 Kubernetes 集群中不同节点之间 CPU、网络 I/O、磁盘 I/O 资源使用率的标准差都是更小的，因此本文所提出的基于集群负载均衡度的调度策略相比 Kubernetes 默认调度策略可以更好地实现集群中资源的均衡使用，保证集群负载均衡。

在上述 6 次实验中，分别收集每一次实验下在对照组和实验组集群中进行 Pod 副本调度时每一个 Pod 副本的实际部署时间，单次实验中每个 Pod 副本的部署时间结果如下图5.7，以及同一实验中所有 Pod 副本的平均调度时间如下图5.8所示，




(a) 第一次实验　(b) 第二次实验　(c) 第三次实验

(d) 第四次实验　(e) 第五次实验　(f) 第六次实验

图 5.7: 多组实验下每个 Pod 副本调度时间结果图

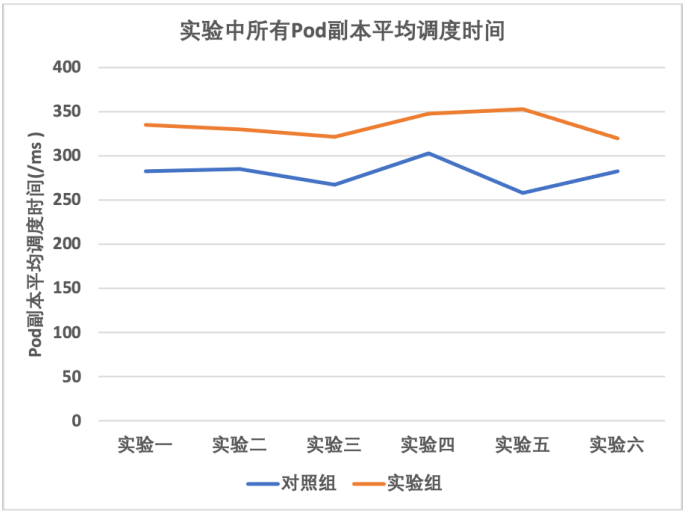



图 5.8: 多组实验下单个 Pod 副本平均调度时间结果图

观察上图5.7中每一次调度实验，在使用默认调度策略的对照组 Kubernetes 集群中，每个 Pod 副本调度时间大多在 150ms 到 400ms 之间；而在使用基于集群负载均衡度的调度策略的实验组 Kubernetes 集群中，每个 Pod 副本调度时间大多在 200ms 到 450ms 之间。在 6 次实验中，对照组 Kubernetes 集群对所有 Pod 副本的平均调度时间为 279.7ms, 实验组 Kubernetes 集群对所有 Pod 副本的平均调度时间为 334.4ms。这是因为相比 Kubernetes 默认调度策略，基于集群负载均衡度的自定义调度策略在进行节点选择时会考虑更多资源指标，并且需要从 Prometheus 存储或者本地缓存中查询资源指标数据，会有一定额外开销。此外，基于集群负载均衡度的自定义调度策略在每次进行 Pod 调度时，都会基于集群资源均衡度进行节点优选，因此调度时间相比对照组 Kubernetes 集群要更长。但是，从集群资源均衡度与调度耗时两方面考虑的话，在进行 Pod 调度时基于集群负载均衡度的调度策略比集群默认调度策略平均耗时多 50ms，但是较大提升了集群资源均衡度。因此该调度耗时在对时延不是非常敏感的场景下是能忍受的，改进后的 Kubernetes 调度机制适用于在线视频网站、在线课程网站等对服务时延不是非常敏感的场景。所以，相比 Kubernertes 默认调度机制，本文所提出的基于集群负载均衡度的自定义调度策略在综合表现较好。

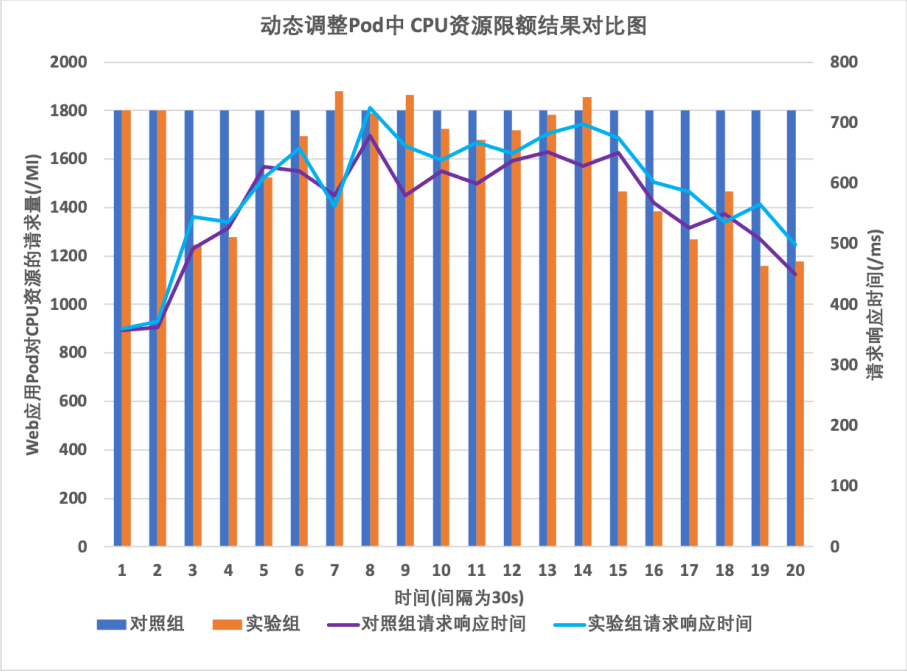



图 5.9: 动态调整 Pod 中 CPU 资源限额结果对比图

此外，在验证本文所提出的动态调整容器资源限额方案有效性的实验中，在对照组与实验组 Kubernetes 集群上均运行 3 副本 Web 应用 Pod，并使用压测工具 Siege 模拟请求访问 Web 应用服务。实验过程中保证 Siege 压测策略的一致性，多次重复该实验，对实验结果进行平均取值后，本文所提出的动态调整 Pod 资源限额方案的表现如上图5.9，

根据实验结果图5.9可知，本文所提出的动态调整 Pod 资源限额的方案能够根据 Kubernetes 最小单位 Pod 对 CPU 资源的实际使用情况动态调整 Pod 的 CPU 资源限额。实验表明，在大多数时刻，此应用 Pod 对 CPU 资源的申请量超出 CPU 资源的实际使用量较多。因此通过动态调整 Web 应用 Pod 中 CPU 资源限额既保证了服务质量，也减少了集群资源不必要的浪费，提升了集群资源的利用率。

## 5.5 本章小结

本章首先介绍实验组与对照组集群配置，然后验证本文对 Kuberbetes 默认弹性伸缩机制与调度机制进行的改进的有效性。首先，进行负载预测实验，结果显示，相较于单一预测模型，ARIMA-GRU 组合预测模型的准确度更高、误差更小。之后，对本文所提出的基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略进行验证，验证其能否正常执行伸缩功能以及是否具有较好性能。实验结果表明，本文所构建的扩缩容策略能够在负载来临之前提前进行扩缩容，相比水平伸缩控制器 (HPA) 所实现的 Kubernetes 被动响应式扩缩容策略，所提出的扩缩容策略能减少服务响应时间，提升服务质量以及节约集群资源。然后，在使用基于集群负载均衡度的调度策略的实验组 Kubernetes 集群与使用默认调度策略的对照组 Kuberbetes 集群中进行实验，验证基于集群负载均衡度的调度策略的有效性。实验结果显示，基于集群负载均衡度的自定义调度策略在调度时更加考虑集群中各种资源的均衡性，该调度策略实现了集群负载均衡。最后验证动态调整 Pod 资源限额的方案，实验结果表示，相比 Kubernetes 默认静态调度机制，本文所提出的动态调整 Pod 资源限额方案在保证服务质量的同时，提高了集群资源利用率。

# 第六章　总结与展望

## 6.1　总结

本文以 Kubernetes 为研究对象，针对 Kubernetes 默认伸缩机制与调度机制中所存在的一些问题进行深入研究。其中，主要分析 Kubernetes 原生伸缩机制与调度机制在实际运行时的缺陷，并且提出相应的优化、改进方案，然后按照优化方案进行具体实践以及测试。

首先，本文构建一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与水平伸缩控制器 (HPA) 配置的弹性伸缩策略。本文针对 Kubernetes 默认伸缩机制所存在的响应不及时以及其所可能导致的资源浪费、服务质量下降等问题，提出一种基于自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型与 HPA 配置的弹性伸缩策略。该策略能够在 Kubernetes 集群负载到来之前提前进行 Kubernetes 最小部署单位 Pod 的扩缩容操作，可以较好地应对突发性的集群负载，并且提升了服务质量以及减少了集群资源的浪费。

然后，本文基于 Scheduling Framework 构建 Kubernetes 自定义扩展调度器，并且提出一种基于集群负载均衡度的调度策略。针对 Kubernetes 默认调度机制在进行调度时考量资源太过片面 (仅考虑 CPU 资源和内存资源) 所可能导致的集群其他资源性能瓶颈问题以及集群负载不均衡等问题，本文基于 Scheduling Framework 构建 Kubernetes 扩展调度器。该自定义扩展调度器在调度时会考虑更多集群资源指标，缓解了集群中其他类型资源的性能瓶颈问题。并且，本文还提出了一种基于集群负载均衡度的调度策略，其能够在进行节点选择时，充分考虑任务特征，从而选择最合适节点进行调度，提升了集群资源负载均衡，提高了集群资源利用率。

最后，本文提出一种动态调整 Kubernetes 最小部署单位 Pod 资源限额的方案。由于 Kubernetes 默认静态调度机制不能在 Kubernetes 最小部署单位 Pod 运行期间根据其运行状态对其进行动态调整, 这可能会导致资源浪费。因此, 本文基于 Pod 实

际运行状态提出一种动态调整 Pod 资源限额的方案，其通过收集、计算 Kubernetes 最小部署单位 Pod 的历史资源使用数据，计算目标 Pod 的实际资源需求。之后使用 Linux 控制群组机制 (Cgroup) 直接调整目标 Pod 资源配置，配置修改后立即生效，从而在不重新构建 Pod 的情况下，实现了目标 Pod 的动态调整。该方案提升了集群资源的利用率，并减少了 Pod 预留资源的浪费。

## 6.2 未来工作与展望

虽然本文针对 Kubernetes 默认伸缩机制与调度机制所存在的一些问题进行了优化、改进，但仍然存在一定的研究和完善空间。

1、本文所构建的自回归差分移动平均模型-门控循环单元 (ARIMA-GRU) 组合预测模型的预测效果较好，但随着预测周期的延长或者时间序列的较大波动，此预测模型的预测准确度存在一定程度的下降，因此构建自适应预测模型是未来需要进一步进行的工作。

2、本文中的监控模块将监控数据收集、存储在 Prometheus 中，但是当预测模块与调度模块在查询 Prometheus 中 Kubernetes 资源对象数据时，存在一定的网络开销。虽然本文使用了缓存在一定程度上减少了部分网络请求的压力，但是在大规模 Kubernetes 集群中某些应用对数据延迟要求较高，此时该部分网络请求所带来的网络开销不可忽视，之后的研究应该考虑如何降低网络数据传输延迟。

3、虽然本文所构建的基于集群负载均衡度的调度策略在调度时考虑了 CPU、内存、网络 I/O、磁盘 I/O、磁盘空间等资源，但是在复杂的场景中，还应该考虑容器镜像下载时间、容器与持久化系统之间的传输速度等资源指标。此外，在进行 Kubernetes 最小管理单位 Pod 的调度时，本文仅考虑了应用 Pod 与节点之间的资源匹配度，并没有考虑应用 Pod 的优先级问题。但是在某些场景下，需要保证高优先级 Pod 能够优先运行。因此在复杂任务场景下，资源指标考虑的全面性以及 Kubernetes 最小管理单位 Pod 的优先级问题也是后续研究中应该考虑的重点。

4、因实验条件有限，本文改进后的 Kubernetes 伸缩机制与调度机制并没有在

大规模 Kubernetes 集群上进行实践，在之后的研究中，应该调整 Kubernetes 集群规模，在大规模 Kubernetes 集群中验证本文改进工作的有效性与可行性。此外，后续研究可以对本文所改进的伸缩机制与调度机制进一步优化，将 GPU 资源等异构资源考虑在内，并把机器学习与大数据作业等任务纳入实验范围，进而检验本文改进工作在高性能计算场景下的表现。

# 参考文献

[1] GONG C, LIU J, ZHANG Q, et al. The characteristics of cloud computing[C]//2010 39th International Conference on Parallel Processing Workshops. : IEEE, 2010: 275–279.

[2] CHARD K, CATON S, RANA O, et al. Social cloud: Cloud computing in social networks[C]//2010 IEEE 3rd International Conference on Cloud Computing. : IEEE, 2010: 99–106.

[3] DILLON T, WU C, CHANG E. Cloud computing: issues and challenges[C]//2010 24th IEEE international conference on advanced information networking and applications. : IEEE, 2010: 27–33.

[4] HWANG K, KULKARENI S, HU Y. Cloud security with virtualized defense and reputation-based trust mangement[C]//2009 Eighth IEEE international conference on dependable, autonomic and secure computing. : IEEE, 2009: 717–722.

[5] PATIDAR S, RANE D, JAIN P. A survey paper on cloud computing[C]//2012 second international conference on advanced computing & communication technologies. : IEEE, 2012: 394–398.

[6] HAY B, NANCE K, BISHOP M. Storm clouds rising: security challenges for iaas cloud computing[C]//2011 44th Hawaii International Conference on System Sciences. : IEEE, 2011: 1–7.

[7] ARMBRUST M, FOX A, GRIFFITH R, et al. A view of cloud computing[J]. Communications of the ACM, 2010, 53(4): 50–58.

[8] SELLAMI M, YANGUI S, MOHAMED M, et al. Paas-independent provisioning and management of applications in the cloud[C]//2013 IEEE Sixth International Conference on Cloud Computing. : IEEE, 2013: 693–700.

[9] CUSUMANO M. Cloud computing and saas as new computing platforms[J]. Communications of the ACM, 2010, 53(4): 27–29.

[10] GODSE M, MULIK S. An approach for selecting software-as-a-service (saas) product[C]//2009 IEEE International Conference on Cloud Computing. : IEEE, 2009: 155–158.

[11] XIAO Z, SONG W, CHEN Q. Dynamic resource allocation using virtual machines for cloud computing environment[J]. IEEE transactions on parallel and distributed systems, 2012, 24(6): 1107–1117.

[12] BERNSTEIN D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE cloud computing, 2014, 1(3): 81–84.

[13] HAQUE M U, IWAYA L H, BABAR M A. Challenges in docker development: A large-scale study using stack overflow[C]//Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2020: 1–11.

[14] SONG M P, GU G C. Research on particle swarm optimization: a review[C]// Proceedings of 2004 international conference on machine learning and cybernetics (IEEE Cat. No. 04EX826): volume 4. : IEEE, 2004: 2236–2241.

[15] VAYGHAN L A, SAIED M A, TOEROE M, et al. Deploying microservice based applications with kubernetes: Experiments and lessons learned[C]//2018 IEEE 11th international conference on cloud computing (CLOUD). : IEEE, 2018: 970–973.

[16] BALLA D, SIMON C, MALIOSZ M. Adaptive scaling of kubernetes pods[C]// NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium. : IEEE, 2020: 1–5.

[17] SAHA P, BELTRE A, GOVINDARAJU M. Exploring the fairness and resource distribution in an apache mesos environment[C]//2018 IEEE 11th International Conference on Cloud Computing (CLOUD). : IEEE, 2018: 434–441.

[18] VERMA A, PEDROSA L, KORUPOLU M, et al. Large-scale cluster management at google with borg[C]//Proceedings of the Tenth European Conference on Computer Systems. 2015: 1–17.

[19] GUO Y, YAO W. A container scheduling strategy based on neighborhood division in micro service[C]//NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium. : IEEE, 2018: 1–6.

[20] CHUNG A, PARK J W, GANGER G R. Stratus: Cost-aware container scheduling in the public cloud[C]//Proceedings of the ACM symposium on cloud computing. 2018: 121–134.

[21] JADEJA Y, MODI K. Cloud computing-concepts, architecture and challenges[C]// 2012 international conference on computing, electronics and electrical technologies (ICCEET). : IEEE, 2012: 877–880.

[22] XIE Y, JIN M, ZOU Z, et al. Real-time prediction of docker container resource load based on a hybrid model of arima and triple exponential smoothing[J]. IEEE Transactions on Cloud Computing, 2020, 10(2): 1386–1401.

[23] GAO J, WANG H, SHEN H. Machine learning based workload prediction in cloud computing[C]//2020 29th international conference on computer communications and networks (ICCCN). : IEEE, 2020: 1–9.

[24] BI J, YUAN H, ZHOU M. Temporal prediction of multiapplication consolidated workloads in distributed clouds[J]. IEEE Transactions on Automation Science and Engineering, 2019, 16(4): 1763–1773.

[25] SAXENA D, SINGH A K, BUYYA R. Op-mlb: An online vm prediction-based multi-objective load balancing framework for resource management at cloud data center[J]. IEEE Transactions on Cloud Computing, 2021, 10(4): 2804–2816.

[26] CHEN Z, HU J, MIN G, et al. Towards accurate prediction for high-dimensional and highly-variable cloud workloads with deep learning[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(4): 923–934. DOI: 10.1109/TPDS.2019.2953745.

[27] XU M, SONG C, WU H, et al. esdnn: deep neural network based multivariate workload prediction in cloud computing environments[J]. ACM Transactions on Internet Technology (TOIT), 2022, 22(3): 1–24.

[28] CALHEIROS R N, MASOUMI E, RANJAN R, et al. Workload prediction using arima model and its impact on cloud applications' qos[J]. IEEE transactions on cloud computing, 2014, 3(4): 449–458.

[29] AL-HAIDARI F, SQALLI M, SALAH K. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources[C]//2013 IEEE 5th International Conference on Cloud Computing Technology and Science: volume 2. : IEEE, 2013: 256–261.

[30] CALHEIROS R N, RANJAN R, BUYYA R. Virtual machine provisioning based on analytical performance and qos in cloud computing environments[C]//2011 International Conference on Parallel Processing. : IEEE, 2011: 295–304.

[31] JUVE G, DEELMAN E. Automating application deployment in infrastructure clouds[C]//2011 IEEE Third International Conference on Cloud Computing Technology and Science. : IEEE, 2011: 658–665.

[32] KHAN A, YAN X, TAO S, et al. Workload characterization and prediction in the cloud: A multiple time series approach[C]//2012 IEEE Network Operations and Management Symposium. : IEEE, 2012: 1287–1294.

[33] ZHANG Y, ZHENG Z, LYU M R. Exploring latent features for memory-based qos prediction in cloud computing[C]//2011 IEEE 30th international symposium on reliable distributed systems. : IEEE, 2011: 1–10.

[34] FARAHNAKIAN F, PAHIKKALA T, LILJEBERG P, et al. Utilization prediction aware vm consolidation approach for green cloud computing[C]//2015 IEEE 8th International Conference on Cloud Computing. : IEEE, 2015: 381–388.

[35] FARAHNAKIAN F, LILJEBERG P, PLOSILA J. Lircup: Linear regression based cpu usage prediction algorithm for live migration of virtual machines in data centers [C]//2013 39th Euromicro conference on software engineering and advanced applications. : IEEE, 2013: 357–364.

[36] ZHENG Z, WU X, ZHANG Y, et al. Qos ranking prediction for cloud services[J]. IEEE transactions on parallel and distributed systems, 2012, 24(6): 1213–1222.

[37] BI J, LI S, YUAN H, et al. Deep neural networks for predicting task time series in cloud computing systems[C]//2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC). : IEEE, 2019: 86–91.

[38] WEI-GUO Z, XI-LIN M, JIN-ZHONG Z. Research on kubernetes' resource scheduling scheme[C]//Proceedings of the 8th International Conference on Communication and Network Security. 2018: 144–148.

[39] COPIL G, MOLDOVAN D, TRUONG H L, et al. rsybl: a framework for specifying and controlling cloud services elasticity[J]. ACM Transactions on Internet Technology (TOIT), 2016, 16(3): 1–20.

[40] REISS C, TUMANOV A, GANGER G R, et al. Heterogeneity and dynamicity of clouds at scale: Google trace analysis[C]//Proceedings of the third ACM symposium on cloud computing. 2012: 1–13.

[41] WOJCIECHOWSKI Ł, OPASIAK K, LATUSEK J, et al. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh[C]//IEEE INFOCOM 2021-IEEE Conference on Computer Communications. : IEEE, 2021: 1–9.

[42] HAN Y, SHEN S, WANG X, et al. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system[C]//IEEE INFOCOM 2021-IEEE Conference on Computer Communications. : IEEE, 2021: 1–10.

[43] SHI Z, JIANG C, JIANG L, et al. Hpks: High performance kubernetes scheduling for dynamic blockchain workloads in cloud computing[C]//2021 IEEE 14th International Conference on Cloud Computing (CLOUD). : IEEE, 2021: 456–466.

[44] PUSZTAI T, ROSSI F, DUSTDAR S. Pogonip: Scheduling asynchronous applications on the edge[C]//2021 IEEE 14th International Conference on Cloud Computing (CLOUD). : IEEE, 2021: 660–670.

[45] YANG Y, CHEN L. Design of kubernetes scheduling strategy based on lstm and grey model[C]//2019 IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering (ISKE). : IEEE, 2019: 701–707.

[46] MENOUER T, DARMON P. New scheduling strategy based on multi-criteria decision algorithm[C]//2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). : IEEE, 2019: 101–107.

[47] ZHONG Z, BUYYA R. A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources[J]. ACM Transactions on Internet Technology (TOIT), 2020, 20(2): 1–24.

[48] ZHIYONG C, XIAOLAN X. An improved container cloud resource scheduling strategy[C]//Proceedings of the 2019 4th International Conference on Intelligent Information Processing. 2019: 383–387.

[49] LV J, WEI M, YU Y. A container scheduling strategy based on machine learning in microservice architecture[C]//2019 IEEE International Conference on Services Computing (SCC). : IEEE, 2019: 65–71.

[50] CHANG C C, YANG S R, YEH E H, et al. A kubernetes-based monitoring platform for dynamic cloud resource provisioning[C]//GLOBECOM 2017-2017 IEEE Global Communications Conference. : IEEE, 2017: 1–6.

[51] XU X, YU H, PEI X. A novel resource scheduling approach in container based clouds[C]//2014 IEEE 17th international conference on computational science and engineering. : IEEE, 2014: 257–264.

[52] ZHANG Q, ZHANI M F, BOUTABA R, et al. Dynamic heterogeneity-aware resource provisioning in the cloud[J]. IEEE transactions on cloud computing, 2014, 2 (1): 14–28.

[53] GANDHI A, HARCHOL-BALTER M, RAGHUNATHAN R, et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers[J]. ACM Transactions on Computer Systems (TOCS), 2012, 30(4): 1–26.

[54] TOKA L, DOBREFF G, FODOR B, et al. Adaptive ai-based auto-scaling for kubernetes[C]//2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). : IEEE, 2020: 599–608.

[55] LIU P, BRAVO-ROCCA G, GUITART J, et al. Scanflow-k8s: Agent-based framework for autonomic management and supervision of ml workflows in kubernetes clusters[C]//2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). : IEEE, 2022: 376–385.

[56] TAMIRU M A, TORDSSON J, ELMROTH E, et al. An experimental evaluation of the kubernetes cluster autoscaler in the cloud[C]//2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). : IEEE, 2020: 17–24.

[57] KIM E, LEE K, YOO C. On the resource management of kubernetes[C]//2021 International Conference on Information Networking (ICOIN). : IEEE, 2021: 154–158.

[58] RATTIHALLI G, GOVINDARAJU M, LU H, et al. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes[C]//2019 IEEE 12th International Conference on Cloud Computing (CLOUD). : IEEE, 2019: 33–40.

[59] GAWEL M, ZIELINSKI K. Analysis and evaluation of kubernetes based nfv management and orchestration[C]//2019 IEEE 12th International Conference on Cloud Computing (CLOUD). : IEEE, 2019: 511–513.

[60] FIORI S, ABENI L, CUCINOTTA T. Rt-kubernetes: containerized real-time cloud computing[C]//Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. 2022: 36–39.

[61] WU Y, YUAN Y, YANG G, et al. Load prediction using hybrid model for computational grid[C]//2007 8th IEEE/ACM International Conference on Grid Computing. : IEEE, 2007: 235–242.

[62] YUAN Y, WU Y, YANG G, et al. Adaptive hybrid model for long term load prediction in computational grid[C]//2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID). : IEEE, 2008: 340–347.

[63] SINGH A K, SAXENA D, KUMAR J, et al. A quantum approach towards the adaptive prediction of cloud workloads[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(12): 2893–2905.

[64] DING D, ZHANG M, PAN X, et al. Modeling extreme events in time series prediction[C]//Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019: 1114–1122.

[65] LIU C, HOI S C, ZHAO P, et al. Online arima algorithms for time series prediction [C]//Proceedings of the AAAI conference on artificial intelligence: volume 30. 2016.

[66] WICHARD J D, OGORZALEK M. Time series prediction with ensemble models [C]//2004 IEEE international joint conference on neural networks (IEEE Cat. No. 04CH37541): volume 2. : IEEE, 2004: 1625–1630.

[67] MARTINETZ T M, BERKOVICH S G, SCHULTEN K J. 'neural-gas' network for vector quantization and its application to time-series prediction[J]. IEEE transactions on neural networks, 1993, 4(4): 558–569.

[68] XU M, HAN M, CHEN C P, et al. Recurrent broad learning systems for time series prediction[J]. IEEE transactions on cybernetics, 2018, 50(4): 1405–1417.

[69] GHODSI M, LIU X, APFEL J, et al. Rnn-transducer with stateless prediction network[C]//ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). : IEEE, 2020: 7049–7053.

[70] AL-MOLEGI A, JABREEL M, GHALEB B. Stf-rnn: Space time features-based recurrent neural network for predicting people next location[C]//2016 IEEE Symposium Series on Computational Intelligence (SSCI). : IEEE, 2016: 1–7.

[71] GUO K, HU Y, QIAN Z, et al. Optimized graph convolution recurrent neural network for traffic prediction[J]. IEEE Transactions on Intelligent Transportation Systems, 2020, 22(2): 1138–1149.

[72] LI C, SONG D, TAO D. Multi-task recurrent neural networks and higher-order markov random fields for stock price movement prediction: Multi-task rnn and higer-order mrfs for stock price classification[C]//Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. 2019: 1141–1151.

[73] XUE H, HUYNH D Q, REYNOLDS M. Ss-lstm: A hierarchical lstm model for pedestrian trajectory prediction[C]//2018 IEEE Winter Conference on Applications of Computer Vision (WACV). : IEEE, 2018: 1186–1194.

[74] ALTCHÉ F, DE LA FORTELLE A. An lstm network for highway trajectory prediction[C]//2017 IEEE 20th international conference on intelligent transportation systems (ITSC). : IEEE, 2017: 353–359.

[75] ZHENG W, CHEN G. An accurate gru-based power time-series prediction approach with selective state updating and stochastic optimization[J]. IEEE Transactions on Cybernetics, 2021, 52(12): 13902–13914.

[76] LIT Z, CAI S, WANG X, et al. Multiple object tracking with gru association and kalman prediction[C]//2021 International Joint Conference on Neural Networks (IJCNN). : IEEE, 2021: 1–8.

[77] FUNARI L, PETRUCCI L, DETTI A. Storage-saving scheduling policies for clusters running containers[J]. IEEE Transactions on Cloud Computing, 2021.

[78] WANG S, DING Z, JIANG C. Elastic scheduling for microservice applications in clouds[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 32(1): 98–115.

[79] WANG S, GONZALEZ O J, ZHOU X, et al. An efficient and non-intrusive gpu scheduling framework for deep learning training systems[C]//SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. : IEEE, 2020: 1–13.

[80] LONG S, WEN W, LI Z, et al. A global cost-aware container scheduling strategy in cloud data centers[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 33(11): 2752–2766.