2023届硕士专业学位研究生学位论文

分类号：_____ 学校代码：10269

密　级：_____ 学　　号：71205902097

East China Normal University

硕士专业学位论文

Master’s Degree Thesis (Professional)

论文题目：基于 Kubernetes 的微服务主动式扩缩与负载均衡算法的研究与应用

院　　系：软件工程学院

专业学位类别：电子信息

专业学位领域：软件工程

指 导 教 师：朱明华 教授

学 位 申 请 人：曹培胜

2023 年 4 月 25 日

Thesis for Master's Degree (Professional) in 2023

University code:10269

Student ID: 71205902097

East China Normal University

Title: Research and Application of Active Scaling and Load Balancing Algorithm for Micro-services based on Kubernetes

Department/School: Software Engineering Institute

Category: Electronic and Information Engineering

Field: Software Engineering

Supervisor: Professor. Minghua Zhu

Candidate: Peisheng Cao

April, 2023

摘 要

近年来，微服务架构被广泛使用。目前以 Kubernetes 为代表的容器管理系统均采用反应式方法对微服务的容器进行扩缩容，这些反应式的方法很容易因扩缩容不及时导致服务负载过高；并且目前微服务架构的负载均衡算法在高负载情况下很容易产生负载不均的情况。

这两方面的问题都会导致服务质量下降，从而违反 SLA（Service Level Agreement，服务水平协议）。针对这两个问题，本文的主要研究内容如下：

(1) 基于 Informer 模型搭建服务负载预测模型，实现 Kubernetes 环境下微服务的主动式扩缩。本文在社区类型的项目中优化了负载预测模型，将项目中的活动事件编码并加入模型，提高了负载预测的准确度；选用服务每分钟被调用次数这一业务相关的指标作为负载预测的指标，避免了其它系统级指标因服务配置变动而导致预测错误的问题；优化了自动扩缩算法，引入重试任务队列，提升了主动式扩缩的可靠性。

(2) 通过改进熵值法提出了一种客户端的阶段式的动态权重负载均衡算法。本文引入权重计算服务和 Redis，计算并传递权重数据，并加入不同指标的可容忍程度对指标权重系数进行加权，使计算出来的指标权重更适用于负载均衡的场景。该算法有效地解决了微服务架构中现有的客户端负载均衡算法容易产生负载不均的问题。

测试结果表明，在社区类型的项目下，本文的负载预测模型比其他模型的预测精度更高，本文的主动式扩缩很大程度上缓解了反应式扩缩的问题。同时与其它算法相比，本文的负载均衡算法可以显著提高服务吞吐量并降低服务请求的响应时间。上述工作有效地减少服务违反 SLA 的情况，并使服务运行得更稳定。

关键词：微服务，Kubernetes，主动式扩缩，Informer，负载均衡

Abstract

In recent years, microservices architecture is widely used. The current container management systems represented by Kubernetes all use reactive methods to scale up and down the containers of microservices, and these reactive methods can easily lead to high service load due to untimely scaling; and the load balancing algorithms of current microservice architectures can easily generate uneven load under high load situations.

Both of these problems can lead to service quality degradation and thus SLA(Service Level Agreement) violation. For these two problems, the main research of this thesis is as follows.

(1) Build a service load prediction model based on Informer model to achieve active scaling of microservices in Kubernetes environment. In this thesis, the load prediction model in a community-type project are optimized by coding and adding the active events in the project into the model to improve the accuracy of load prediction; the business-related metric of the number of service invocations per minute is chosen as the metric for load prediction to avoid the problem of other system-level metrics causing prediction errors due to service configuration changes; the automatic scaling algorithm is optimized and the retry task queue is introduced to improve the reliability of active scaling.

(2) A client-side phased dynamic weight load balancing algorithm is proposed by improving the entropy value method. This thesis introduces weight calculation service and Redis to calculate and transfer weight data, and adds the tolerability of different indicators to weight the indicator weight coefficients, so that the calculated indicator weights are more applicable to the load balancing. The algorithm effectively solves the problem that the existing client-side load balancing algorithm in microservice architecture tends to generate uneven load.

The test results show that the load prediction model in this thesis has higher prediction accuracy than other models under community type projects, and the active scaling in this thesis largely alleviates the problem of reactive scaling. Meanwhile the load balancing algorithm in this thesis can significantly improve the service throughput and reduce the response time of service requests compared with other algorithms. The above work effectively reduces the service SLA violation and makes the service operation more stable.

Keywords: Microservice, Kubernetes, Active Scaling, Informer, Load Balance

目 录

第一章 绪论

1.1 研究背景及意义

随着云计算和容器技术的广泛应用，越来越多的企业将业务应用程序迁移到云端，并采用微服务架构来构建高度可扩展、高可用性和高性能的应用程序。微服务架构通过将应用程序拆分成小型、自治的服务单元，以实现高度可扩展性、可维护性和灵活性，从而更好地满足不断变化的业务需求。

然而，随着企业应用程序的规模不断增长，微服务架构中的服务数量也在不断增加，如何确保微服务架构的高可用性和高性能成为了一个重要的问题。在这种情况下，自动化的扩缩容和负载均衡算法变得越来越重要，以确保微服务架构的稳定性和可靠性。

Kubernetes 作为一个开源的容器编排平台，可以自动管理和调度大规模的容器化应用程序，已经成为了自动化和管理容器化应用程序的首选工具之一。Kubernetes 通过提供自动化的扩缩容和负载均衡功能来帮助管理微服务应用程序，使得微服务应用程序可以自动调整其资源使用量，以满足不断变化的业务需求。Kubernetes 还提供了各种灵活的 API 和工具，以便开发人员能够更加轻松地管理其微服务应用程序，从而降低了开发和运维的难度。如今，大多数互联网公司都在使用 Kubernetes 部署他们的服务，各个云服务厂商也都对外提供了 Kubernetes 服务以方便公司或机构项目的运维和部署。

在 Kubernetes 中，通过 HPA（Horizontal Pod Autoscaler，Pod 水平自动扩缩）功能，可以实现对微服务的自动扩缩容。HPA 是一种反应式的扩缩容机制，通过监视 Pod 的资源使用情况自动调整 Pod 副本的数量。这种反应式的方法更适用于单体架构而在微服务架构下表现不佳。首先，微服务架构下，服务请求的调用链可能很长。由阿里巴巴开源的阿里巴巴集群追踪数据[1]显示，最长的微服务调用链包含一百多个微服务。调用链末端的微服务相比于调用链首端的微服务需要更长的时间才能感知到工作负载的变化，这就导致末端的微服务不能够及时地进

行扩展。其次，每个微服务的扩展需要从容器镜像库中下载镜像、启动容器、启动服务等过程，整个过程需要几秒甚至十几秒来完成。然而服务的 SLA（Service Level Agreement，服务水平协议）通常定义在几百毫秒[2]，反应式的扩缩容机制很容易导致违反 SLA。

在生产集群中，由于常规用户活动，多数的微服务工作负载呈现周期性模式，比如小红书、豆瓣、知乎、B 站这些社区类型的网站，每天的中午和下班后是用户访问的高峰时段，并且晚间时段的负载要比白天低很多。同时这些社区类型的项目受到项目中各种活动事件的影响，在活动期间服务的负载明显比平时更多。在负载呈周期性变化的应用项目中，本文通过加入主动式的扩缩方案来缓解反应式扩缩的问题。主动式扩缩是在即将有大量请求来临之前得到对服务负载的预测，提前对服务进行扩容，当大量请求来临时服务容器已经扩容完毕，也就减少了因负载容量不够而导致服务波动的风险。

与此同时，虽然 Kubernetes 提供了服务注册与发现和负载均衡的功能，但是因为使用 Kubernetes 提供的这些功能可定制性不足、服务代码需要与 Kubernetes 耦合、开发调试门槛高等原因，在实际的生产中，多数公司还是会选择通过注册中心实现服务注册与发现和客户端的负载均衡方式。当前微服务架构下的客户端的负载均衡算法为了简单大都是静态的或者是伪动态的负载均衡算法，由于微服务架构中服务调用关系复杂，静态的客户端式的负载均衡算法的负载均衡效果不佳，在高负载情况下，服务资源很难被充分利用。

主动式的扩缩方案虽然可以很大程度上减少服务处于高负载的情况，但是主动式的扩缩依赖于负载的预测，预测的负载数据不会百分百准确，当实际负载高于预测的负载数据时，服务依然可能会处于高负载的状态。

本文通过设计一种微服务架构下的客户端的动态负载均衡算法，以在负载预测错误时或在服务器资源不足时保证负载的均衡，使得在有限服务资源的情况下，拥有更高的服务资源的利用率和服务的质量。并且相比于静态的负载均衡算法，动态的负载均衡算法在服务扩缩后可以更快的负载均衡，从而提高微服务架构下服务的性能和稳定性。

1.2 国内外研究现状

1.2.1 容器自动扩缩容的研究现状

微服务的扩缩容其实就是对微服务的容器进行扩缩容，目前，容器的自动扩缩容的方法主要有两种：一种是基于阈值的反应式方法，另一种是基于预测的主动式方法。反应式方法是根据实时的资源使用情况，自动调整容器数量和规模，这种方法适用于应用程序负载变化不太频繁的情况。基于预测的方法则是利用机器学习等算法，根据历史数据和趋势来预测未来的资源需求，并自动调整容器数量和规模，这种方法适用于负载变化比较频繁的情况。

目前，市面上已经有了很多实现了容器自动扩缩容的框架和工具，其中比较流行的有 Kubernetes、Docker Swarm、Apache Mesos 等。这些工具和框架可以通过集群管理和容器编排来实现容器的自动扩缩容。此外，还有一些云平台提供了容器自动扩缩容的服务，如 AWS 的 Auto Scaling[3]和 Google Cloud 的 Autoscaler[4]等。上述的自动扩缩容都是基于计划或阈值的反应式方法，这种方式简单且容易实现。当前很多自动扩缩容的研究都是基于阈值的方法[5][6][7]，但是基于阈值的方法很难选择适当的阈值，尤其是在处理动态工作负载时。

与反应式的方法相比，主动式的方法通过预测微服务的工作负载，在负载波动到来前执行扩缩，这使得主动式的方法在时效性和准确性上要比反应式的方法更优秀。当前大量研究都通过时间序列预测算法实现对服务负载的预测从而实现容器主动式的扩缩，服务的负载数据也是一种时间序列的数据，当前基于时间序列的预测模型主要有两大类：基于传统概率统计学和基于神经网络模型的方法[8][9]。

Kan[10]使用自回归移动平均(ARMA)统计模型预测请求率，然后根据预测的请求率估计满足未来工作负载需求所需的容器数量，但是这个模型的对时间序列的预测能力并不理想。Ye 等人还使用二阶 ARMA 模型作为 Kubernetes 编排工具上的容器资源（CPU）利用率预测算法[11]。Ciptaningtyas 等人通过使用差分自回归移动平均（ARIMA）模型预测计划阶段期间的请求数[12]，作者选择 ARIMA 模型的原因是它对统计相关时间序列的短期预测精度较高，其预测能力要强于

ARMA，但是该模型的准确性可能会有较大的波动，尤其是在处理不同数据集的时候。

随着人工智能技术的不断进步，越来越多的机器学习和人工智能算法被用于时间序列预测。最初提出的循环神经网络(RNN)和长短期记忆(LSTM)模型被广泛认为适用于序列生成等任务，此后出现了众多基于这些结构的神经网络模型，这些机器学习模型也被运用到了服务的负载预测和容器自动扩缩容的研究中。文献[13]通过 LSTM 模型，预测容器运行时的动态工作负载变化，实现了对 Docker 容器进行自动扩缩，并且在预测未来工作负载方面的性能优于 ARIMA 模型。文献[14]通过双向长短期记忆（Bi-LSTM）模型对未来 HTTP 工作负载的数量进行预测，实现 Kubernetes 环境下的主动式扩缩，与 LSTM 模型相比 Bi-LSTM 模型在预测精度和弹性加速方面表现更好，并且这种主动式的扩缩容的表现要优于 Kubernetes 的 HPA 方式。文献[15]在 Bi-LSTM 模型的基础上，提出了多元 Bi-LSTM 模型，该模型能够更好地对多元的时间序列数据进行预测，在对服务工作负载的预测上有着更好的表现。郭杨虎[16]使用 GRU 模型对服务负载进行预测，实现了微服务在 Docker 环境下的调度。上述的机器学习模型虽然能够对服务负载提供有效的预测，但是与 Transformer[17]、Informer[18]等加入了注意力机制的模型相比预测的精度还是有待提高。

1.2.2 负载均衡算法的研究现状

负载均衡算法可分为两类：服务端的负载均衡和客户端的负载均衡[19]。国内外的负载均衡算法相关的研究有很多，且研究主要集中在服务端实现负载均衡。

在国内，聂世强等人提出了一种基于跳跃哈希的对象分布算法，该算法能够有效地将海量数据分布到存储节点上，从而解决了一致性哈希算法在负载均衡方面存在的负载倾斜问题[20]。汪佳文等人基于轮询和权重算法，通过收集服务的各项性能指标计算每个服务器的权重，提出了一种动态自适应权重轮询随机负载均衡算法，这种算法在异构集群中表现良好[21]。王诚等人对传统的一致性哈希算法进行改进，提出了基于贪心算法的分配策略，提高了负载均衡策略的效果[22]。张开琦等人在 CHWBL 算法的基础上进行改进，提出了 WCHVN（基于虚拟节点的

加权有界负载一致哈希）算法，解决了 CHWBL 算法无法根据节点性能的差异对各节点进行负载分配的问题[23]。吴俊鹏等人提出了一种新的负载均衡算法，该算法可以动态更新服务器各项性能指标权重与服务器权值。算法能更好地实现集群的负载均衡[24]。葛钰等人对 Nginx 的加权最小连接算法进行了优化，通过定期读取后端服务器的运行状态参数来确定下一个周期内的权重分配，并动态调整权值以提高负载均衡效率，这种方法有效地解决了原算法存在的问题[25]。刘瑞奇等人提出了概率随机的调度策略 resource-Pick__kx 和确定的平滑加权轮询算法将负载均衡的性能提升了 10%-20%[26]。

国外在负载均衡方面也有较多的研究成果。文献[27]阐述了当前微服务架构中主流负载均衡技术的方法和特征，并指出了负载均衡技术的未来发展方向。文献[28]通过收集服务的 CPU 利用率、磁盘利用率、内存利用率和连接数作为服务负载评估的参数实现了一种微服务架构下的动态权重负载均衡算法，该算法让服务保持较大的吞吐量和较短的响应时间。Vahab Mirrokni 等人提出了一种名为 CHWBL(Consistent Hashing with Bounded Loads)的算法，该算法在一致性哈希算法的基础上，对各个节点的负载上限进行限制，有效地解决了由于一致性哈希算法负载倾斜而导致节点过载的问题[29]。文献[30]提出了 DWLOAD(Dynamic Weight Loading Algorithm)算法，该算法将虚拟节点和动态权重相结合，通过不同服务副本最新请求的响应时间和吞吐量数据计算每个服务的实时权重数据，使得服务 QPS 处理能力提高了约 50%，并大幅减少了服务的响应时间。

1.3 研究内容

在 Kubernetes 框架下，为了应对 Kubernetes 提供的 HPA 的局限，并提升微服务的可用性，本文通过深度学习模型进行负载预测以实现容器主动式扩缩，并针对微服务中客户端的负载均衡算法产生的负载不均的问题，提出了可以更好适应服务负载变化的客户端的动态权重的负载均衡算法。本文具体的研究内容包括以下几点：

(1) 收集 Kubernetes 中服务的节点性能指标，通过改进 Informer 模型，在本

文社区类型项目的情况下实现更高准确率的长时间序列的服务负载预测，并通过优化后的自动扩缩算法实现 Kubernetes 环境下微服务容器的主动式扩缩。

(2) 实现了微服务中客户端的阶段式动态权重的负载均衡算法。通过改进熵值法计算不同服务副本的权重数据，并通过 Redis 中间件传递权重数据，实现权重计算与负载均衡算法的解耦。

(3) 利用监控工具和 JMeter 测试工具对上述改进进行监控和测试，验证上述改进的预期效果。

1.4 本文的组织结构

本文共有六个章节，每个章节的主要内容如下：

第一章，绪论。

第二章，相关理论与技术。本章对微服务、容器技术、管理微服务容器的 Kubernetes、Prometheus 监控系统、微服务容器的动态扩缩容、时间序列预测和负载均衡技术进行了概述和分析，为后续的研究提供有效的理论技术支持。

第三章，微服务负载预测模型与主动式扩缩。本章通过对比主动式和反应式扩缩，说明了为什么要实现主动式的扩缩；选用 Informer 模型对微服务的负载进行预测，并结合本文的社区项目对 Informer 模型做了部分改进，使模型在预测服务负载时准确率更高；设计了自动扩缩算法防止频繁的扩缩容影响服务质量。

第四章，微服务的客户端式动态负载均衡设计。本章借助 Redis 传递服务中的负载信息，通过对熵值法的改进，实现了客户端式的动态权重的负载均衡算法。

第五章，微服务主动式扩缩与负载均衡算法的部署与测试。本章介绍了本文的系统架构和系统的环境搭建，并对本文在微服务主动式扩缩和负载均衡算法上面的研究内容进行测试，验证了本文工作内容的有效性。

第六章，总结与展望。

第二章 相关理论与技术

2.1 微服务与容器技术

微服务是一种软件开发和部署架构模式，它将一个大型的单体应用程序拆分成多个小型的服务，每个服务都是独立的、可独立部署、可独立运行、可独立维护、可独立扩展，并通过网络进行通信，形成一个分布式系统。每个服务只需关注自己的业务逻辑，不需要考虑整个应用程序的逻辑和依赖关系。微服务架构通常采用轻量级的协议和通信方式，如 REST、HTTP、JSON 等，以便服务之间的通信和集成。

与传统的软件架构相比，微服务架构的优势主要体现在以下几个方面：

灵活性和可扩展性：由于每个微服务都是独立的，因此可以针对不同的需求和场景进行独立扩展和优化。如果某个服务需要更高的并发量或更高的性能，只需增加其实例数量或者优化其实现即可，而不会影响其他服务的运行。

故障隔离和容错能力：微服务架构可以通过服务之间的隔离和独立运行，实现对故障的隔离和容错。如果某个服务出现故障或不可用，其他服务不会受到影响，整个应用程序也可以继续运行。

更快的开发和交付速度：由于每个微服务都是独立的，因此可以采用不同的技术栈和开发流程，以适应不同的需求和场景。开发人员可以更加专注于服务的开发和测试，而不用关注整个应用程序的开发和测试。这可以使得应用程序的开发和交付速度更快，同时也可以降低开发和测试的成本。

更好的可维护性和可扩展性：由于每个微服务都是独立的，因此可以更加容易地进行维护和更新。如果某个服务需要进行升级或者改进，只需对该服务进行修改和测试即可，而不用修改整个应用程序。

更好的组织结构和协作：微服务架构可以使得组织结构和协作更加灵活和高效。每个服务可以由一个小团队来负责开发和维护，团队之间可以更加松散地协作和交流，以适应不同的需求和场景。

同时，微服务架构也可以更好地适应云计算和容器技术的发展趋势，因为容器技术可以提供更加轻量级的服务部署和运行环境，并可以更加容易地进行自动化部署和管理。

容器技术是一种轻量级的虚拟化技术，它通过隔离应用程序的运行环境来提供更高的灵活性和可移植性。容器技术可以将应用程序及其依赖项打包成一个可移植的容器镜像，容器镜像可以在不同的平台上运行，包括本地开发机、云服务器、容器编排平台等。容器技术最常用的实现是 Docker，它通过 Dockerfile 和 Docker 镜像的方式，使得应用程序的部署变得更加简单、可重复和可移植。

微服务和容器技术的结合可以带来以下优势：

更好的可移植性和可扩展性：容器技术可以使得微服务更加易于部署和管理，并且可以轻松地扩展和缩减服务的数量，以适应应用程序的变化需求。

更好的开发效率：微服务和容器技术可以使得开发者更加专注于应用程序的业务逻辑和特性实现，而不用考虑底层的环境和基础设施。

更好的可靠性和故障恢复能力：微服务和容器技术可以使得应用程序更加健壮和容错，每个服务都可以独立运行和管理，即使某个服务出现故障，也不会影响整个应用程序的运行。

总之，微服务和容器技术都是现代化应用程序开发和部署的重要组成部分，它们可以使得应用程序更加灵活、可靠和高效，有助于企业实现数字化转型和业务的快速发展。

2.2 Kubernetes

Kubernetes 是一个开源的容器编排和管理平台，它是 Google 内部使用的 Borg 系统的开源实现[31]。Kubernetes 最初由 Google 开发，现在由云原生计算基金会（CNCF）进行维护。它的目标是简化容器的部署、扩展和管理，并提供更高级的自动化和可靠性。

Kubernetes 提供了一种便捷的方法来管理容器，并通过多种方式实现高可用性和可扩展性，例如在多个节点上复制应用程序实例和动态调整容器数量以应对负载变化。Kubernetes 的主要特点有：

服务发现和负载均衡：Kubernetes 可以根据用户需求和流量情况自动地为容器分配 IP 地址和 DNS 名称，并且可以在它们之间提供负载均衡。通过服务发现功能，服务可以通过服务名找到它所依赖的服务。

存储编排：Kubernetes 可以自动地将用户选择的存储系统挂载到容器中。

自动化部署和回滚：Kubernetes 可以根据部署配置自动替换和重启故障容器，以实现应用程序的无缝更新。

自我修复：Kubernetes 可以检测并重新启动那些失败了的容器，杀死那些未通过用户定义的健康检查的容器，并且在准备好之前不会将其广播给其他节点。

密钥与配置管理：Kubernetes 可以管理密钥、令牌、证书等敏感信息，并且可以在不重建镜像或暴露密钥的情况下更新应用程序配置。

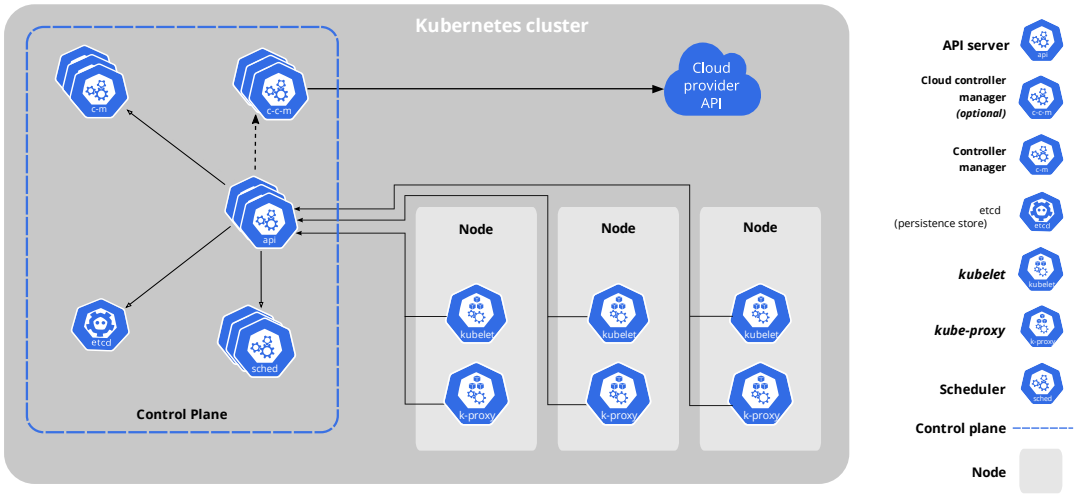

图 2-1 Kubernetes 集群中的组件

图 2-1 展示了 Kubernetes 集群中使用到的组件，主要包括：

(1) Master 节点：运行着各种控制平面组件，负责管理整个 Kubernetes 集群，其中包含的几个重要组件如下：

API Server：所有对 Kubernetes 系统的操作都需要通过 API Server 进行访问和管理。

Scheduler：它负责将 Pod 调度到合适的节点上运行。当用户创建一个 Pod 时，它会被提交给 Scheduler，然后 Scheduler 将根据一定的算法和策略，选择一个最合适的节点来运行该 Pod。

Controller manager：负责运行各种控制器，监视和管理 Kubernetes 中的资源对象。控制器通过监听 API Server 中资源对象的变化，然后自动触发一系列的操作来维持资源对象的状态。例如，Replication Controller 控制器负责确保集群中的 Pod 数量始终保持在一个预期的范围内；Deployment 控制器负责确保集群中的 Pod 根据所定义的 Deployment 策略进行滚动更新；Job 控制器负责运行一次性任务，并确保任务的执行状态等等。

etcd：一个分布式的键值存储库，它负责存储 Kubernetes 集群中的所有状态信息和配置数据。

(2) Node 节点：是运行应用程序实例的计算资源，可以是物理服务器、虚拟机或云实例，每个 node 节点都运行着以下几个核心组件：

kubelet：提供管理容器的生命周期、监控容器的健康状态、容器资源管理、节点状态报告等功能。

kube-proxy：一个网络代理，负责在集群内部为服务提供负载均衡和服务发现功能。

(3) Pod：是 Kubernetes 中最小的部署单元，它包含一个或多个容器及其共享的网络和存储资源。

(4) Service：是一种逻辑网络抽象，用于将多个 Pod 组合成一个可访问的服务，可以通过单个 IP 地址和 DNS 名称公开。

(5) Volume：是一种持久化存储技术，它将数据存储在独立于 Pod 的容器之外的存储卷中。

此外，Kubernetes 提供了丰富的 API 和 CLI，使得管理集群和部署应用程序变得非常方便。Kubernetes 还支持多种编排工具和平台，如 Helm、Istio 和 Knative 等，以便更轻松地部署和管理应用程序。

总之，Kubernetes 是一个强大的容器编排和管理平台，可以帮助开发人员和运维人员更轻松地管理应用程序和容器，并以高效、可靠和可扩展的方式运行它们。

2.3 Prometheus 容器监控系统

Prometheus 是一款开源的监控系统，它最初由 SoundCloud 开发并在 2012 年开源，目前由 CNCF 维护。它旨在通过对分布式系统的监控来帮助解决云原生应用程序的监控需求，支持多维数据模型和查询语言，可以实时地记录和查询应用程序的度量指标数据，提供了可视化的仪表板和警报功能，帮助用户对系统的状态进行监控和管理。

Prometheus 的核心设计思想是“多维度数据模型”，它采用一种称为“指标”（Metric）的数据结构来表示监控数据，一个指标由一个标识符（Metric name）和一个或多个键值对（Label）组成，它们描述了监控数据的名称、类型、维度和值等信息。Prometheus 还提供了一种查询语言——PromQL，可以支持多维度的指标查询和聚合，并提供了灵活的警报机制，可以通过配置警报规则来监控系统状态并发送通知。

Prometheus 的架构如图 2-2 所示，其体系结构包括以下几个核心组件：

Prometheus Server：负责收集、存储和查询指标数据，支持通过 HTTP 或者其他协议获取指标数据，并可以将指标数据存储到本地磁盘或者远程存储系统中。

Exporter：负责采集指标数据并将其暴露给 Prometheus Server，Prometheus 支持多种 Exporter，如 Node Exporter（用于采集主机指标）、Blackbox Exporter（用于采集网络指标）、MySQL Exporter（用于采集数据库指标）等。

Alertmanager：负责处理警报通知，当警报触发时，Alertmanager 会根据警报规则进行分组和处理，并将警报通知发送给指定的接收方。

Client Library：Prometheus 提供了多种客户端库，如 Go、Java、Python、Ruby 等，可以方便地在应用程序中集成指标的采集和上报功能。

服务发现：Prometheus 通过服务发现模块让 Prometheus 可以自动发现 Kubernetes 中的服务并自动采集信息，其中 file_sd 让 Prometheus 可以通过配置文件来实现服务的自动发现。

PromQL：与 SQL 相比，PromQL 查询和统计时序数据更高效更简单。通过 API clients 的 query 和 query_range 接口外部程序也可以向接口传入指定的

PromQL 查询语句从而获取 Prometheus 监控到的数据。也可以通过在 Grafana 中配置 PromQL 查询语句显示图形化的监控数据。

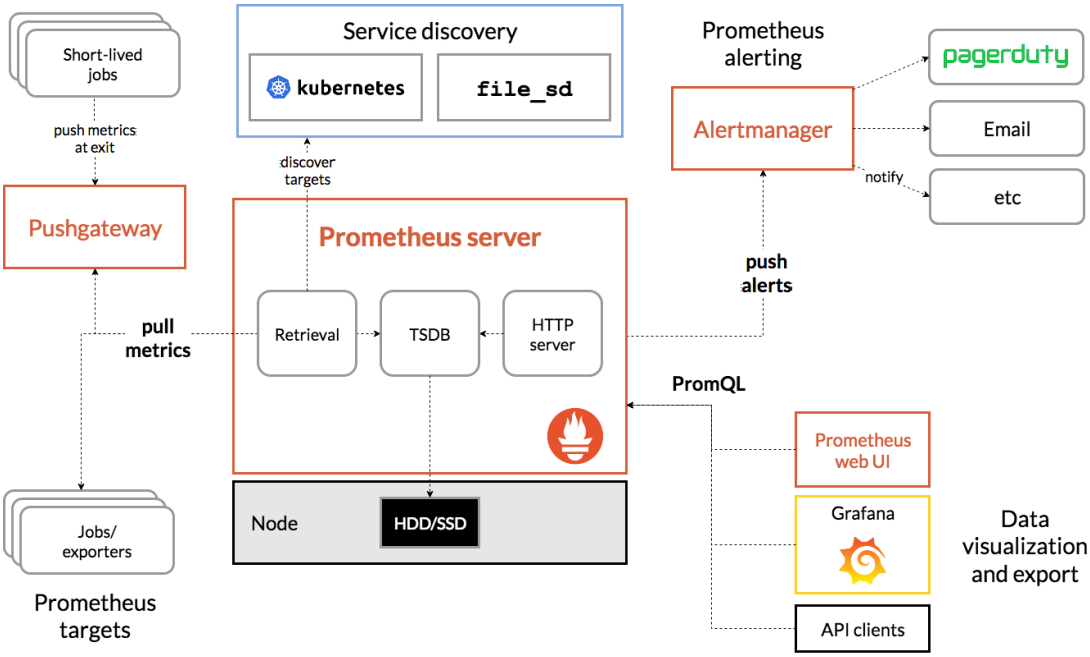

图 2-2 Prometheus 架构图

2.4 微服务容器的扩缩容技术

对一个服务的扩缩可分为两种方式：垂直扩缩和水平扩缩。垂直扩缩通过增加单个服务的资源数量实现，比如增加服务的 CPU 核心频率或数量、增加服务的内存大小、增加服务的带宽等，而水平扩缩容通过增加服务的数量实现对服务的扩展。垂直扩缩和水平扩缩各有优缺点，垂直扩缩可以节省资源，但是有资源上限和重启成本，且垂直扩缩对服务性能的提升是有限的，水平扩缩可以提高服务的可用性和弹性，但是有网络开销和协调成本，但是通常来说服务的数量越多服务的性能越好，水平扩缩的性能提升上限要高很多。

在微服务中通常都使用水平扩缩的方式对服务的容器进行扩缩，其中 Docker Swarm 和 Kubernetes 中的 HPA（Horizontal Pod Autoscaler）是主流的对微服务进行水平扩缩的方式。

2.4.1 Docker Swarm

Docker Swarm 是一种容器编排工具，可以在多个 Docker 节点上管理和部署

容器应用程序[32]。Docker Swarm 支持动态扩缩容，即根据应用程序的负载情况自动调整容器数量，以确保应用程序始终具有足够的资源并能够快速响应负载。

在 Docker Swarm 中，动态扩缩容是通过服务（Service）来实现的。服务是一组相同配置的容器的集合，它们共同提供同一应用程序的功能。Docker Swarm 通过调整服务的副本数，就可以实现动态扩缩容。

要启用动态扩缩容，需要定义以下参数：

(1) 副本数：即服务中容器的初始数量。

(2) CPU 和内存限制：即每个容器可使用的 CPU 和内存数量。

(3) 负载均衡策略：即如何将流量分配给不同的容器。

(4) 自动伸缩策略：即何时启动或停止容器。

Docker Swarm 支持两种自动伸缩策略：基于 CPU 利用率和基于请求计数。

基于 CPU 利用率：根据 CPU 利用率来调整容器数量。当 CPU 利用率超过预定义的阈值时，Docker Swarm 将启动新容器；当 CPU 利用率低于另一个阈值时，Docker Swarm 将停止一些容器。

基于请求计数：根据传入请求的数量来调整容器数量。当请求数量超过预定义的阈值时，Docker Swarm 将启动新容器；当请求数量低于另一个阈值时，Docker Swarm 将停止一些容器。

2.4.2 HPA

Kubernetes 中的 HPA（Horizontal Pod Autoscaler）是一种自动水平扩缩机制，它能够根据应用程序的资源使用情况自动调整 Pod 的数量，以满足应用程序对资源的需求[33]。

HPA 的工作原理如下：

(1) HPA 定期获取指标（如 CPU 利用率或内存利用率）。

(2) 如果指标超出了预先设置的阈值，HPA 就会自动调整副本数，以增加或减少 Pod 数量。

(3) 如果需要添加或删除 Pod，HPA 将根据预定义的模板自动创建或删除 Pod。

(4) HPA 能够确保应用程序始终具有足够的资源，并在需要时自动进行扩展

或缩减。这有助于提高应用程序的可靠性和性能，并减少成本浪费。

要创建 HPA，需要定义以下参数：

(1) 监测对象：即要监测的 Pod 或 Controller。

(2) 监测指标：即要监测的指标，如 CPU 利用率或内存利用率。

(3) 监测周期：即监测指标的时间间隔。

(4) 最小 Pod 数：即最小的 Pod 数量，即使负载很低，也必须保留的 Pod 数量。

(5) 最大 Pod 数：即最大的 Pod 数量，即使负载很高，也不能超过的 Pod 数量。

在实际应用中，可以根据应用程序的特性和负载情况来选择不同的监测指标和监测周期，以获得最佳的性能和可靠性。同时，也需要根据应用程序的资源需求和预算限制来设置最小和最大 Pod 数量。

2.5 时间序列预测

时间序列预测是一种用于预测未来时间点或一段时间内的数值或趋势变化的方法。它主要应用于时间序列数据的预测，例如股票价格、气象数据、交通流量、销售数据等。时间序列预测可以帮助企业和个人做出更好的决策，例如预测销售趋势、优化供应链管理、预测交通拥堵等等。本文为了实现微服务的主动式扩缩，需要对服务的负载数据进行预测，负载数据就是典型的时间序列数据。

时间序列预测方法可以分为两类：基于统计学模型的方法和基于机器学习模型的方法。

2.5.1 基于统计学模型的方法

基于统计学模型的时间序列预测方法主要是利用历史数据中的统计信息来预测未来的数据。其中比较常用的方法有 ARIMA 模型、ETS 模型和 VAR 模型。

ARIMA (Auto-Regressive Integrated Moving Averages)模型是一种基于时间序列差分和自回归移动平均的模型，它适用于没有明显趋势和季节性的时间序列[34]。ETS 模型是一种基于指数平滑的模型，它适用于具有明显趋势和季节性的时

间序列。VAR 模型是一种多元时间序列预测方法，它适用于多个变量之间存在相互关联的情况。

基于统计学模型的时间序列预测方法的优点如下：

(1) 可解释性强：基于统计学模型的方法通常有明确的数学模型和参数，能够很好地解释模型的预测结果。

(2) 稳定性高：能够很好地处理稳定和平稳的时间序列数据。

(3) 高精度：对于稳定和平稳的时间序列数据，基于统计学模型的方法通常能够得到较高的预测精度。

缺点如下：

(1) 受限于数据质量：基于统计学模型的方法对数据质量有较高的要求，需要有充足的历史数据，并且数据需要具有一定的规律性。

(2) 无法处理非线性问题：基于统计学模型的方法通常是基于线性关系的假设，对于非线性问题处理能力较差。

(3) 参数估计的困难：对于复杂的统计学模型，参数的估计需要比较复杂的数学方法和算法，因此对于一些数据量较大或者特征较复杂的时间序列数据，基于统计学模型的方法的训练和预测效率会比较低。

受用户活动的影响，服务的负载数据是会经常波动的，虽然负载数据有一定的规律性但是数据还是非线性的，使用统计学模型对负载数据进行预测的效果不佳。

2.5.2 GBM 类回归算法

GBM（Gradient Boosting Machine）是一种基于决策树的集成学习算法，通常用于回归问题。GBM 类算法的核心思想是通过不断地迭代来训练多个弱分类器（通常为决策树），并将它们组合成一个强分类器，以提高模型的预测性能。

GBM 类算法的训练过程可以分为以下几个步骤：

(1) 初始化：将训练集的每个样本的权重设置为相等值。

(2) 训练弱分类器：训练第一个弱分类器，通常为一个决策树。在训练过程中，样本的权重会不断被更新，使得分类器能够更好地拟合数据。

(3) 计算残差：计算每个样本的残差，即真实值与当前模型预测值的差异。

(4) 更新样本权重：根据残差的大小更新每个样本的权重，使得对于残差较大的样本，其权重更大，对下一个分类器的训练影响更大。

(5) 训练下一个弱分类器：基于更新后的样本权重，训练下一个弱分类器。与第一个分类器不同的是，第二个分类器会根据更新后的权重对数据进行加权，以便更好地拟合残差。

(6) 组合弱分类器：将所有训练出来的弱分类器组合起来，形成一个强分类器。这个强分类器的预测值是每个弱分类器的预测值的加权和。

(7) 迭代训练：重复上述过程，直到达到指定的迭代次数或者预测误差已经达到满意的水平。

GBM 类算法的优点是能够在训练过程中不断地优化模型的预测性能，并且对于一些非线性和高维度的数据，也能够有较好的拟合效果。然而，它也存在一些缺点，例如对于噪声数据和异常值比较敏感，容易出现过拟合的情况，以及需要进行较多的参数调整和模型优化。

2.5.3 基于 RNN 的 LSTM 模型

RNN（循环神经网络）是一种常用的神经网络模型，主要用于处理序列数据的建模和预测。与传统的前馈神经网络不同，RNN 模型具有反馈连接，可以接收并处理上一个时刻的输出作为当前时刻的输入，如图 2-3 所示。

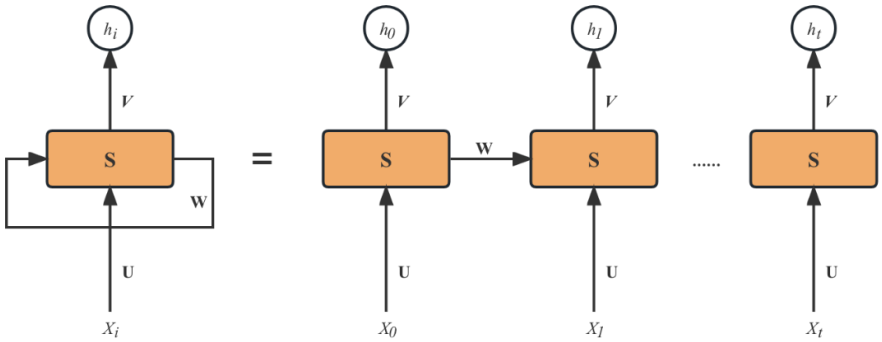

图 2-3 RNN 模型结构

RNN 模型的核心思想是利用时间维度的信息来增强模型的表达能力，从而

更好地适应序列数据的特征和模式。具体来说，RNN 模型通过引入一个隐状态（hidden state）来表示历史输入数据的影响，随着时间的推移，这个隐状态会不断地被更新和传递。

然而，传统的 RNN 模型存在着长时间依赖问题，即模型难以有效地处理与当前时刻相隔较远的输入信息。为了解决这个问题，出现了 LSTM（Long short-term memory,长短期记忆）模型等变体，可以更好地处理长序列数据[35]。

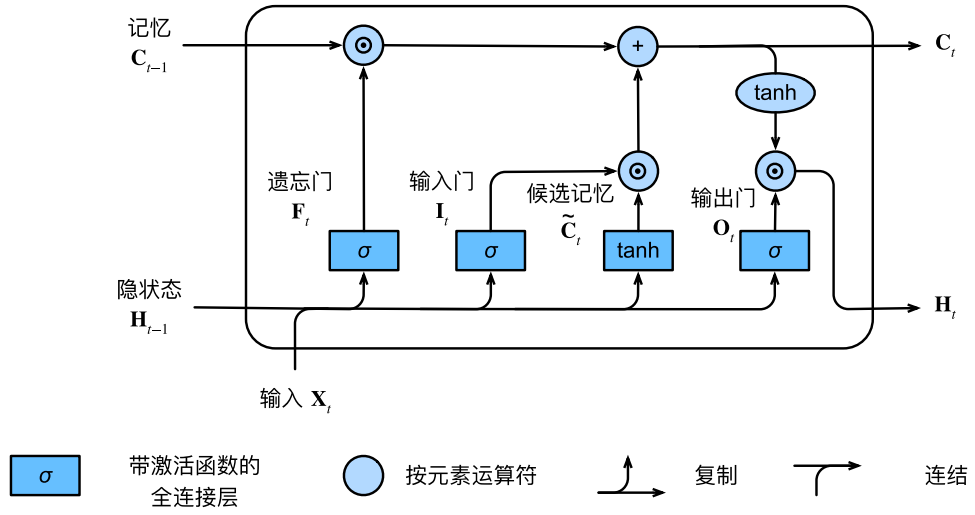

图 2-4 LSTM 模型中的记忆单元

LSTM 模型的核心思想是引入“记忆单元”，用于存储和控制信息流的传递。LSTM 记忆单元的结构如图 2-4 所示，这个记忆单元包含了三个门（输入门、输出门和遗忘门），通过对输入数据进行加权和控制，可以决定信息在记忆单元中的存储和传递。

输入门可以控制新的输入数据对记忆单元的影响，输出门可以控制记忆单元中的信息如何被输出，而遗忘门则可以控制旧的信息如何被遗忘。这三个门的组合使得 LSTM 模型能够有效地处理长序列数据，并且可以避免长时间依赖问题。

但是 LSTM 模型还存在一些问题：

计算量较大：LSTM 模型需要大量的计算资源，尤其是当输入序列较长时，还是会出现梯度消失或爆炸的问题，导致模型难以训练。

需要手动设置参数：LSTM 模型需要手动设置很多参数，如隐藏状态的大小、学习率、学习率衰减率等，这些参数的设置需要经验和尝试，比较困难。

容易出现过拟合：LSTM 模型在处理复杂的任务时，容易出现过拟合的问题，需要采取一些正则化方法来缓解这个问题。

难以处理长期依赖：虽然 LSTM 模型可以处理长期依赖关系，但是当序列长度较长时，它仍然难以捕捉到序列中的长期依赖。

预测结果可能不准确：由于 LSTM 模型是一种基于历史信息预测未来信息的模型，因此在预测结果时，如果历史信息不足或者有噪声，预测结果可能不准确。

2.5.4 注意力机制

深度学习模型中的注意力机制是用于控制信息流向的一种机制，它允许模型以一种有意识的方式对输入信息进行选择，以此来确定最重要的信息，这有助于减少冗余信息的影响，并有助于提高模型的准确性和效率[36]。

注意力机制在深度学习中的应用主要分为两种：点注意力和序列注意力。点注意力是指在单个样本中对单个元素进行注意力分配的机制，例如图像分类问题中的对单个像素的注意力分配。序列注意力是指在多个样本中对整个序列进行注意力分配的机制，例如语言模型中的对单词序列的注意力分配，又比如时间序列预测模型中对时间序列的注意力分配。

注意力机制在深度学习中的具体实现方式主要有以下几种：基于神经网络的注意力机制，基于内存的注意力机制和基于自注意力的注意力机制。基于神经网络的注意力机制是指在神经网络中增加一层注意力层，通过该层对输入信息进行注意力分配。基于内存的注意力机制是指使用一个内存矩阵来记录输入信息的权重，并通过该内存矩阵来进行注意力分配。基于自注意力的注意力机制是指在序列数据中，通过一种自注意力机制来对整个序列进行注意力分配。

注意力机制通常和其它技术结合使用，比如在 LSTM 模型中加入注意力机制，让 LSTM 在编解码时打破了依赖于内部一个固定长度向量的限制，使得 LSTM 模型在长序列数据的处理上有了更好表现。

2.5.5 基于 encoder-decoder 的 Transformer 模型

Transformer 模型是一种基于注意力机制（attention mechanism）的神经网络

模型，由 Vaswani 等人在 2017 年提出，被广泛应用于自然语言处理领域，特别是机器翻译任务中[17]。与传统的循环神经网络（RNN）和卷积神经网络（CNN）相比，Transformer 模型不需要通过循环或卷积操作处理输入序列，而是直接利用注意力机制来捕捉输入序列中的相关信息。

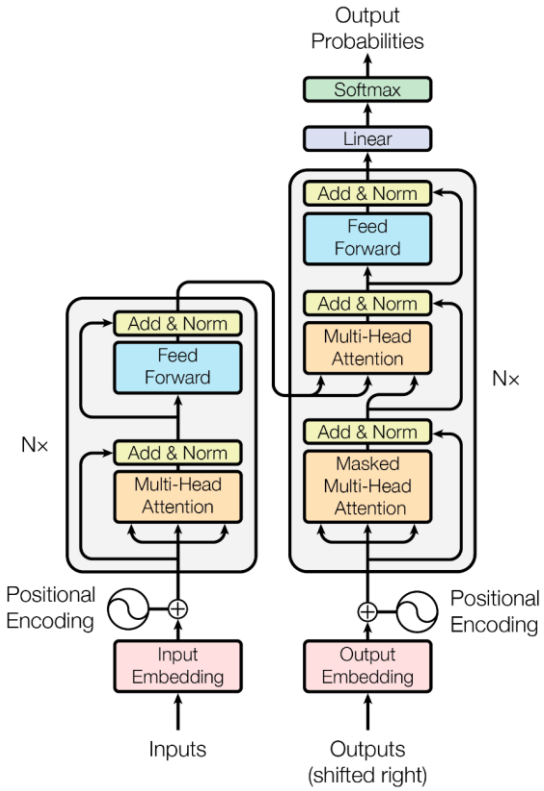

图 2-5 Transformer 模型架构

如图 2-5 所示，Transformer 模型主要由两个组件构成，分别是编码器（encoder）和解码器（decoder）。编码器将输入序列转换为一个高维的表示，而解码器则将这个表示转换为目标序列。两个组件均由多个相同的层组成，每层中包含两个子层，一个是多头自注意力机制，另一个是前馈网络。多头自注意力机制用于将输入序列中的信息相互关联起来，前馈网络则用于对每个位置上的向量进行非线性变换。

(1) 位置编码

在处理序列数据时，循环神经网络具有处理序列顺序的内置机制，但是由于 Transformer 中的 self-attention 是不考虑输入序列的位置的，并且架构中没有使用递归和卷积的结构，每个数据点视为独立于其他数据点。为了保留有序列数据中

顺序的信息，Transformer 模型引入了位置编码（Positional Encoding）的概念，位置编码公式如下。

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \quad (2-1)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \quad (2-2)$$

其中$PE_{(pos,2i)}$和$PE_{(pos,2i+1)}$分别表示在位置pos和维度$2i$或$2i+1$上的位置编码，d_{model}表示嵌入的维度，其中$i \in [0,(d_{model}-1)/2]$，三角函数的频率沿向量维度减小，且$d_{model}$可被 2 整除。该位置编码将输入数据转换为$d_{model}$维度的位置向量，这个向量隐含了数据在序列中的相对位置关系。这种位置编码方法可以保证不同位置之间的位置编码是不同的，从而帮助模型学习序列中的位置关系。

位置编码的使用有助于 Transformer 模型更好地处理序列数据，并且不存在长期依赖的问题，这是传统的循环神经网络在处理序列数据时的一个主要困难。因此，Transformer 模型可以更快地收敛，并且在许多任务中取得了很好的效果。

(2) 自注意力机制

自注意力机制是 Transformer 的核心，自注意力机制是一种计算输入序列中每个元素与所有其他元素的关联性的机制，这种机制可以帮助模型捕捉到序列中的长期依赖关系。

自注意力先将输入矩阵经过不同的线性变换生成三个不同的矩阵：查询矩阵Q、键矩阵K和值矩阵V，Q,K,V每行的维度都为d_k，正因为Q,K,V矩阵都来自同一组输入，所以叫做自注意力。自注意力将Q和K矩阵做点积得到得分矩阵 score，为了梯度的稳定，Transformer 使用了 score 归一化，将得分矩阵除以$\sqrt{d_k}$进行缩放操作，最后通过$softmax$函数转化为概率矩阵再与V矩阵做点积得到输出，计算公式如下：

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2-3)$$

Transformer 中的多头注意力机制，是自注意力机制的一种扩展形式，它同时

计算多个注意力向量，通过学习不同的行为来提高模型的表达能力和泛化能力。

如图 2-6 所示，多头注意力机制可以分为以下三个步骤：

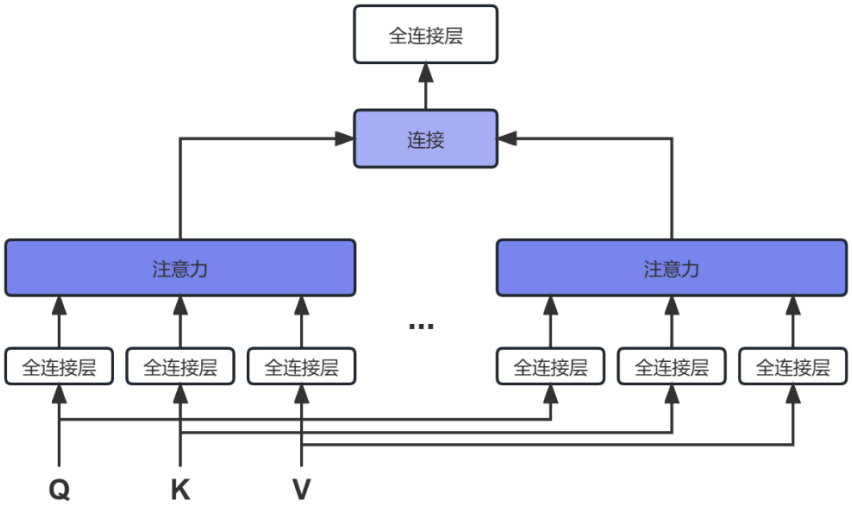

图 2-6 多头自注意力机制

多头切分：将输入序列切分成多个子序列，每个子序列都由一个全连接层实现一个特定的权重矩阵。

注意力计算：对于每个子序列，计算其与所有其他子序列之间的相似度，得到一个注意力分数向量。这个相似度可以通过点积或其他方式计算。

多头融合：将每个子序列的注意力向量加权求和，得到一个综合的注意力向量。这个综合的注意力向量可以用于对输入序列进行加权求和，得到输入序列的一个表示。

虽然计算量相对于 LSTM 更多，但是 Transformer 的自注意力机制可以很容易的实现并行计算，Transformer 的速度要比 LSTM 的更快。

2.5.6 Informer 模型

Informer 模型通过对 Transformer 的改进，使得模型在处理长序列时间序列问题(LSTF)上，可以预测得更快并且预测的准确度更高[18]。

现有的模型，比如 LSTM 模型、Transformer 模型，在短序列预测上表现很好，比如预测 48 个点或更少，而更长的序列长度则会使得模型的预测能力受到很大影响。

原始的 Transformer 模型在解决 LSTF 问题时主要面临三点限制：

(1) L^2的计算开销。

(2) 长序列输入时堆叠层数带来的内存瓶颈。

(3) 预测长序列输出时速度会比较缓慢。

图 2-7 展示了 Informer 模型的架构，Informer 针对上述三点的限制，做出了如下改进：

(1) 提出 ProbSparse Self-attention，筛选出最重要的 query，降低计算复杂度，时间复杂度为$O(L * \log L)$。

(2) 提出了 self-attention 蒸馏机制来缩短每一层的输入序列长度，减少维度和网络参数量，降低了模型的计算量和存储量。

(3) 提出 Generative Style Decoder，一步得到所有预测结果，将预测时间复杂度从$O(N)$降到了$O(1)$。

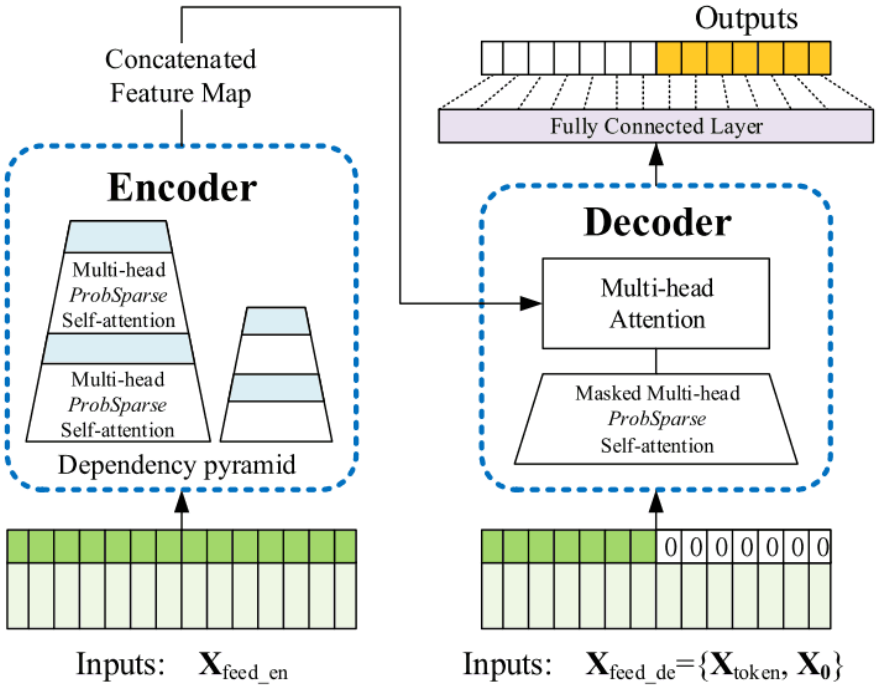

图 2-7 Informer 模型架构

除此之外，与 Transformer 模型相比，Informer 还对输入数据中的时间数据进行编码，这使得 Informer 能够学习到时间对数据的影响，这在 LSTF 问题上是一个很重要的操作。

2.6 微服务中的负载均衡技术

在微服务架构中，负载均衡技术是非常重要的一环，它可以将流量分发到不同的服务实例中，实现高可用、高性能的服务访问。按负载均衡器所在位置划分，负载均衡可以分为服务端的负载均衡和客户端的负载均衡[19]。

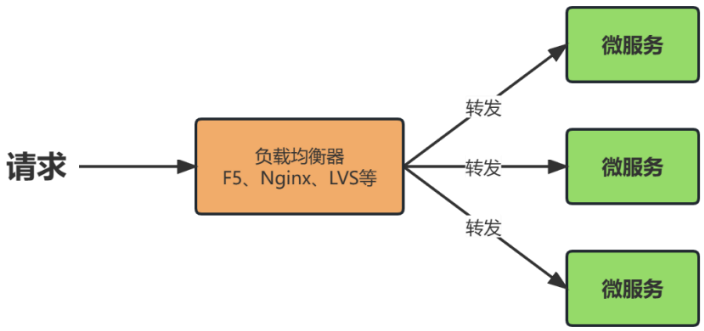

图 2-8 服务端的负载均衡架构

服务端负载均衡是指在服务提供者端进行负载均衡，如图 2-8 所示，通常会在服务端设置一个负载均衡器，负载均衡器接收服务请求，并根据一定的规则将请求分发到不同的服务实例中。常用的服务端负载均衡方式包括：

基于 DNS 的负载均衡：通过 DNS 服务器将请求解析成多个 IP 地址，实现请求分发到多个后端服务实例的目的。但是 DNS 负载均衡存在一些局限性，如 DNS 缓存、解析延迟等问题。

基于网络设备的负载均衡：通过网络设备（如 F5、Array 等）的负载均衡功能，将请求分发到不同的后端服务实例上。这种负载均衡技术具有较好的性能和可靠性，但需要昂贵的硬件和专业的网络知识。

基于软件的负载均衡：通过反向代理服务器（如 Apache、HAProxy、Nginx 等）将请求转发到后端服务实例，实现负载均衡和高可用性的目的。反向代理服务器可以根据多种策略（如轮询、随机、IP 哈希等）进行请求分发，还可以提供路由、过滤和安全控制等功能。

此外在 Kubernetes 中的负载均衡也是服务端的负载均衡，其实现是由 kube-proxy 组件完成的。kube-proxy 会监视 Service 和 Endpoint 对象的变化，以便动态更新负载均衡规则。

客户端负载均衡是指在服务消费者端进行负载均衡，将请求发送到不同的服务实例中。常用的客户端负载均衡技术包括：

本地负载均衡：在客户端实现一个负载均衡的模块，通过配置列表、路由规则等信息来选择服务实例处理请求，可以灵活地控制请求分配策略，但需要实现负载均衡模块，对客户端有一定的影响。

服务发现：通过服务注册中心来发现可用的服务实例，并将它们的信息缓存到本地，在每次请求时选择可用的服务实例处理请求，可以避免硬编码服务实例信息的问题，但对服务注册中心有一定的依赖。

DNS 解析：将服务实例的域名注册到 DNS 服务器中，客户端通过域名解析来获取服务实例的 IP 地址，可以实现简单的负载均衡和服务发现，但对于服务实例动态变化的情况，需要实现定时更新 DNS 记录。

微服务中的负载均衡与传统的负载均衡有所不同，主要是由于微服务架构中的服务实例数量更多、更动态，而且服务实例可能分布在多个数据中心、区域、甚至云平台上。因此，微服务架构中的负载均衡需要具备更高的智能和自适应性，Kubernetes 中提供的负载均衡功能可以很好地满足这些需求，但是国内的大部分微服务应用还在使用客户端式的负载均衡方式，对于多数企业来说将现有的微服务中的负载均衡处理转交给 Kubernetes 处理的意义不大，并且还需要承担改变架构带来的风险。

因为 Spring Cloud 被广泛应用于微服务架构的开发，Spring Cloud 使用客户端的负载均衡方式，因此当前的大部分微服务都使用客户端的负载均衡。在 Spring Cloud Alibaba 中，Nacos 作为微服务架构下的注册中心，每个微服务中的 Nacos Client 会维护一个定时任务通过持续调用服务端的接口更新心跳时间，保证自己处于存活状态，防止服务端将服务剔除，Nacos 默认 5 秒向服务端发送一次，通过请求服务端接口/instance/beat 发送心跳。在做负载均衡时，客户端从 Nacos 获取指定的服务名称的服务集群的 IP 地址、端口号、权重值和其他的相关元数据，通过相应的客户端的负载均衡算法选择其中一个服务进行调用。具体过程如图 2-9 所示。

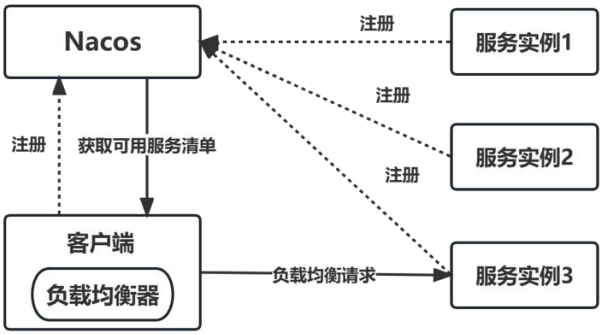

图 2-9 客户端的负载均衡流程

在 Spring Cloud 中，Ribbon 与 Spring Cloud LoadBalancer 是常用的客户端的负载均衡器，但是在 Spring Cloud2021 版本 Ribbon 就被移除了，改用 Spring Cloud LoadBalancer 作为默认的负载均衡器。Ribbon 与 Spring Cloud LoadBalancer 中提供许多负载均衡策略可供选择，具体策略如下：

随机负载均衡：随机选择一个服务实例处理请求，简单、易实现，但无法保证请求的均衡分配。

轮询负载均衡：依次轮询选择每个服务实例处理请求，可以实现请求的均衡分配，适用于服务实例相对平均、负载均衡要求不高的场景，但对于请求响应时间不一致的情况，可能会导致性能问题。

加权轮询负载均衡：在轮询的基础上，通过权重值来调整服务实例的请求处理比例，可以更精确地控制请求的分配比例。

最少连接数负载均衡：选择当前连接数最少的服务实例处理请求，可以有效地避免请求集中在某一个服务实例上的问题，适用于服务响应时间不同、服务实例数量较多的场景，但需要实时统计连接数，对性能有一定的影响。

最短响应时间负载均衡：选择响应时间最短的服务实例来处理请求，适用于服务响应时间相差较大的场景。

基于哈希值的负载均衡：通过对请求数据的哈希值进行计算，将同一个哈希值的请求分配给同一个服务实例，适用于需要保持请求和响应的一致性的场景。

总之，负载均衡通常是微服务的重要组成部分，一个好的负载均衡方式可以提高应用程序的可用性和性能，减少服务器故障和停机时间，从而降低业务风险。

2.7 本章小结

本章节针对本论文基于 Kubernetes 的微服务主动式扩缩与负载均衡算法研究中使用到的相关理论和技术进行了阐述和说明。为后文对微服务主动式扩缩与负载均衡算法的研究做准备。

第三章 微服务负载预测模型与主动式扩缩

微服务架构下通常为了应对峰值请求会过度配置峰值资源以满足服务水平协议（Service Level Agreement，SLA）[2]。但是过度配置不可避免的造成了服务资源的浪费，为了缓解这个问题，可以在工作负载随时间变化时使用被动或主动方法自动扩缩每个微服务的容器数量。反应式方法根据当前工作负载和系统的性能反馈调整资源；而主动式的方法通过预测工作负载的变化，提前对微服务进行扩缩，以解决绪论中提到的反应式方法的问题。

主动式的方法通过收集服务的负载数据，采用某种合适的预测方法预测未来一段时间的负载数据，当预测的工作负载超过当前服务副本数量所能承受的程度之前就将服务副本数量进行扩充，或预测工作负载降低时主动释放空闲的服务副本，减少资源的浪费。主动式的方法大幅度减少了服务处于高工作负载的时间，从而使得主动式的方法与反应式方法相比更能满足 SLA。

但是主动式方法的运行依赖于准确的服务工作负载预测和合理的扩缩算法。本章节基于 Informer 搭建针对本文社区类型项目的工作负载预测模型，并在预测模型的基础上设计扩缩算法，实现了微服务的主动式扩缩。

3.1 基于 Informer 的负载预测模型

Informer 模型在时序序列预测中表现出众，在模型训练和预测时速度更快、对硬件要求更少，且在长时序序列预测时的精准度更高，所以本文选用 Informer 模型对服务的负载数据进行预测。为了能够更准确地预测出项目中各种活动对服务负载的影响，本小节通过对活动事件进行编码并加入模型，使得模型在对负载数据进行预测时，能有更好的预测结果。

3.1.1 负载指标的选择

在 Kubernetes 中，每个容器的微服务工作负载与容器的操作系统级指标（如 CPU 和内存利用率）密切相关，每个容器的操作系统级指标达到一定的值之后都会导致服务质量受到影响。所以通过对容器的操作系统级指标的预测，并扩展微

服务的容器副本的数量，使容器的各项操作系统级的指标保持在设定的阈值以下，从而保证微服务的工作负载能够稳定运行。

容器在运行时需要运行整个的操作系统环境，容器中的微服务只是容器中正在运行的其中一个进程，这就导致即使是在稳定运行的情况下，CPU 利用率和内存利用率这些操作系统级的指标也会在小幅度范围内波动，并且在容器负载变高时这些指标的波动范围更大。一个微服务容器的副本可能有几个至几十个不等，极端情况下，这些指标数据的波峰和波谷叠加会使得获取的指标数据产生很大的误差，这对负载的预测结果产生影响。

经过对阿里巴巴公开对微服务集群数据研究发现，绝大多数的微服务都是 CPU 型或是内存型的微服务，且微服务需要处理的每分钟被调用次数(calls per minute，CPM)与容器的操作系统级指标呈线性关系[37]。经过验证，本文的社区项目中各个服务都是 CPU 型或是内存型微服务，且服务的 CPM 与 CPU 利用率和内存利用率呈线性关系。所以，本文同时收集 CPU 利用率、内存利用率、CPM 这三项指标，通过 CPM 与 CPU 和内存利用率的线性关系，由服务中每个容器 CPU 和内存利用率的阈值推导出每个容器的 CPM 阈值。通过对 CPM 数据的预测，即可确定需要对 Kubernetes 中微服务的容器副本进行扩缩的数量。

3.1.2 Informer 模型的特征输入

在特征输入上，LTSF 问题中，时序建模不仅需要局部时序信息还需要层次时序信息，如星期、月和年等，以及突发事件或某些节假日等。经典自注意力机制很难直接适配，因此 Informer 提出了三层特征输入形式，如图 3-1 所示。

Informer 的三层特征输入：

(1) 数据 embedding。将原始的输入数据做一维卷积，将输入数据从$C_{in}(=7)$维映射为$d_{model}(=512)$维。

(2) 位置编码。这里的位置编码和 Transformer 模型的位置编码相同。在 RNN 模型中，由于 RNN 本身就是按照序列的位置顺序处理数据，需要处理完前一个数据才能处理当前数据。与 RNN 模型不同，在 Informer 和 Transformer 中，输入的序列元素是一起处理的，这样做加快了模型的速度但是忽略了序列中元素的先

后关系。通过位置编码，Informer 和 Transformer 模型便能知道数据在序列中的位置信息。

(3) 时间戳编码。首先 Informer 将输入的时间转换为由年、月、周、日、小时、分钟的数组数据以供对时间戳进行编码。Informer 提供了两种时间戳的编码方式：TemporalEmbedding 和 TimeFeatureEmbedding，前者使用 month_embed、day_embed、weekday_embed、hour_embed 和 minute_embed(可选)多个 embedding 层处理输入的时间戳，将结果相加；后者直接使用一个全连接层将输入的时间戳映射到 512 维的 embedding。Informer 默认使用 TimeFeatureEmbedding 的编码方式。

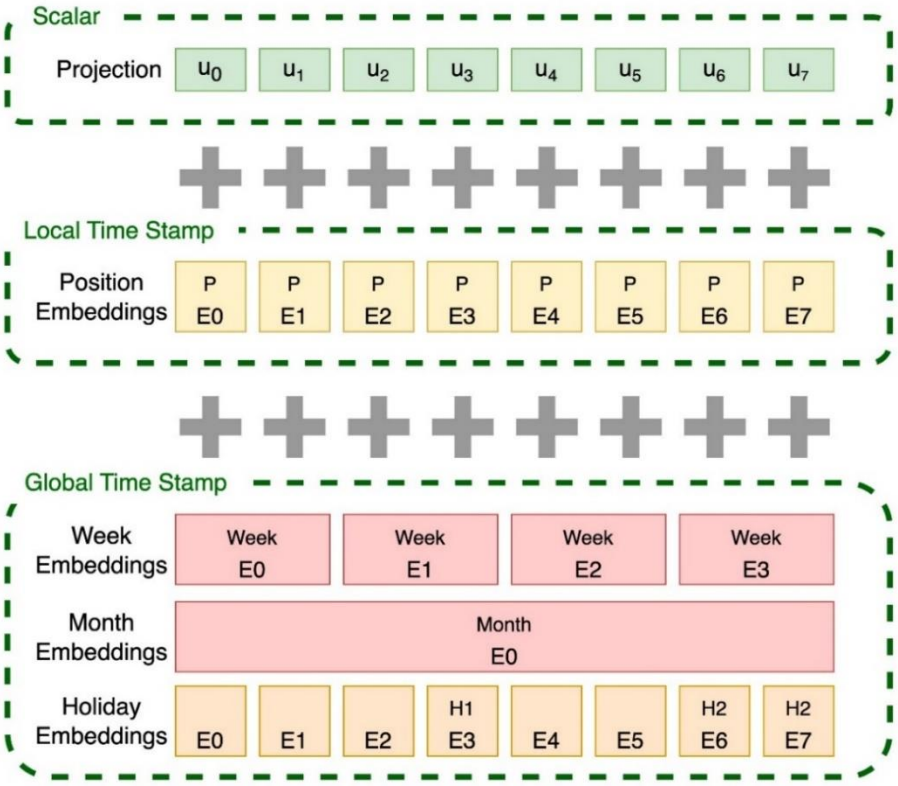

图 3-1 Informer 模型的嵌入

得益于位置编码和时间戳编码的加入，Informer 在解码时，即使不依赖自回归结构也能有不错的预测效果。如图 3-2 所示，输入序列的时间从 t0 到 t1，预测序列的输入从 t2 到 t3，不用通过预测 t1 到 t2 的数据就可直接获取到 t2 到 t3 的预测数据。

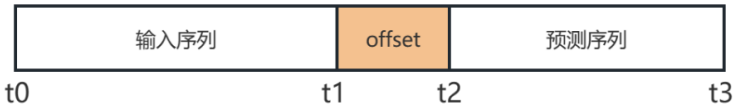

图 3-2 Informer 间隔预测

3.1.3 受活动事件影响的特征输入

本文的社区项目因为会不定期进行推广和折扣活动，每次在推广和折扣活动期间，项目的访问请求都会有不同程度地提升。Informer 模型在对输入序列数据进行 embedding 时添加了时间戳编码，使得模型在训练和预测时将时间这一特征考虑在内，可以更好地反映出不同月份、不同日期、工作日和周末的负载区别，但是这仍然无法学习到项目中的各种活动对服务负载的影响。本文在原先 Informer 的 embedding 操作的基础上，将当前时间是否在活动期间这一特征也加入 embedding 操作中，以提高预测的精准度，如图 3-3 所示。

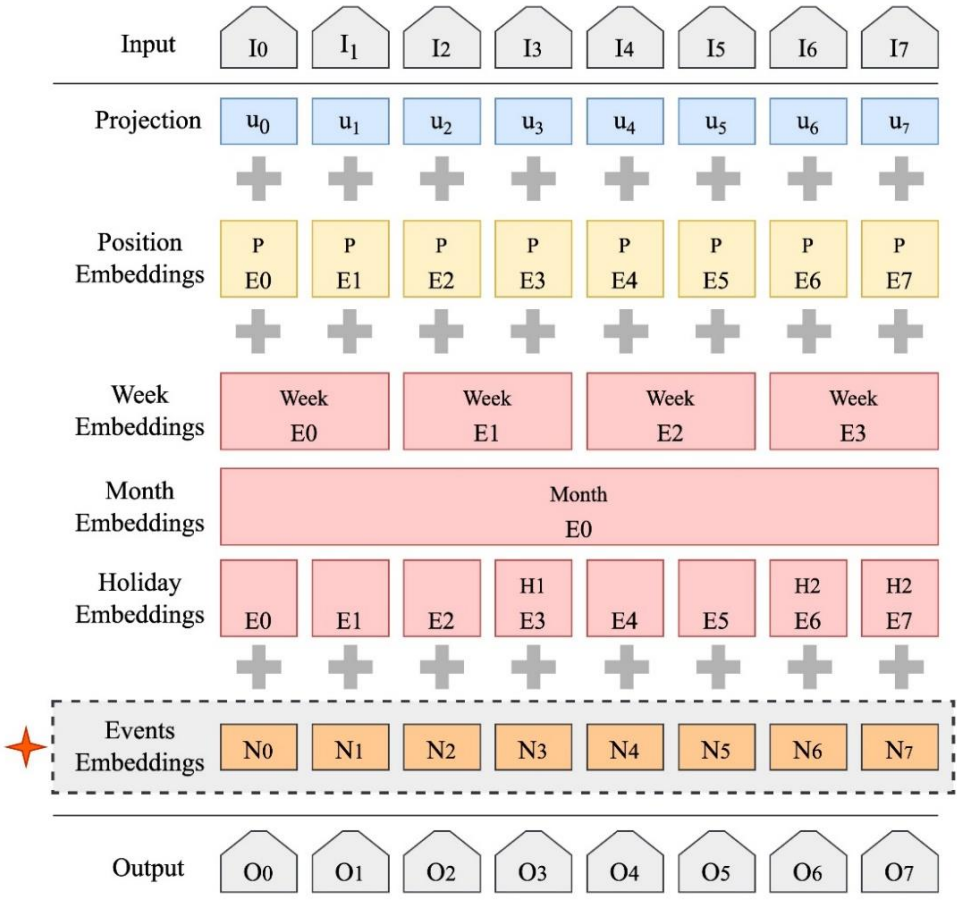

图 3-3 活动事件嵌入

本文项目中的活动分为几种不同的类型：广告推广、站内活动、折扣活动。

通过观测，每种活动对不同服务的负载影响也不同，比如在折扣活动时，项目中的售卖服务的负载明显升高，而在站内活动时，售卖服务的负载变化不大。在同一时间不同的活动可能会同时进行，也可能同一时间只有一个活动正在进行。如果只通过单一的变量反映当前时间是否在活动状态，模型就无法区分出当前正在进行的是什么活动，也就无法准确学习到活动对服务负载的影响。因为本文的活动类型有 3 种，所以在对处理活动数据做 embedding 时需要 3 维的输入。

当前时间，某一活动类型正在进行则输入为 1，否则输入为 0。比如，当只有站内活动正在进行时，输入的数据为[0,1,0]，站内活动和折扣活动同时进行时，输入的数据为[0,1,1]。Events Embeddings 通过一个全连接层将输入数据从$C_{in}(=3)$维映射为$d_{model}(=512)$维。

如公式 3-1 所示，将活动事件数据 embedding 后的结果加到 Informer 原先的 embedding 操作中，其中x为负载指标的输入数据，x_mark为编码后的时间数据，x_event为编码后的活动数据。

$$\begin{aligned} x = {} & value\_embedding(x) + position\_embedding(x) \\ & + temporal\_embedding(x\_mark) \\ & + events\_embedding(x\_event) \end{aligned} \qquad (3-1)$$

3.1.4 模型的训练与参数调整

本文服务负载数据以一天为一个周期，白天用户请求量大，服务负载较高，夜间用户请求量很小，服务负载较低。为了能够更好地让模型学习到训练数据的周期性，本文设定训练数据的时间间隔为 15 分钟。在做模型训练时，模型中编码器的输入序列长度为一天的数据量，有 96 条数据。部分输入序列的数据需要作为解码器的输入，设定为半天的数据量有 48 条。最后，每次获取的预测数据为一天的数据量，有 96 条数据。

3.1.5 获取 CPM 与各指标的线性关系

在 3.1.1 节中提到一个服务的 CPM 数据与该服务的 CPU 使用率和内存使用率呈线性关系。本文通过线性回归模型获取每个服务 CPM 与各指标的线性关系。

$$g_i^{cpu}(x) = a_i^{cpu} \times x + b_i^{cpu} \qquad (3-2)$$

$$g_i^{mem}(x) = a_i^{mem} \times x + b_i^{mem} \quad (3-3)$$

其中，x为每个服务平均分配的 CPM 数据，$g_i^{cpu}(x)$为服务i在 CPM 为x时对应的平均 CPU 使用率，$g_i^{mem}(x)$为服务i在 CPM 为x时对应的平均内存使用率。a_i和b_i分别对应着 CPM 与指标的线性关系的斜率和偏移，偏移值代表着服务容器在没有调用请求的情况下，操作系统和容器内其他进程占用各项指标的值。

每个服务经过线性回归都会获得a_i^{cpu}、b_i^{cpu}、a_i^{mem}、b_i^{mem}这 4 个指标的值，保存这 4 个指标的值，供预测时使用。

3.2 预测与扩缩算法

系统设定每隔一天对后面一天的服务负载数据进行预测。每次预测获取最新的 96 条以 15 分钟为间隔的 CPM 数据传入预测程序，得到的数据包括预测的 96 条 CPM 数据和该服务的 CPU 使用率与内存使用率和 CPM 指标经过线性回归计算后的线性关系。在得到这些数据之后，还需要通过相应的算法计算并自动扩缩容器的数量。

3.2.1 计算预测的扩缩目标副本数量

首先，扩缩后的容器要求容器的 CPU 使用率和内存使用率不能超过阈值，这里阈值设置为 70%，通过 CPM 指标与 CPU 和内存使用率的线性关系可得到每个微服务i的每个容器的 CPM 阈值$threshold_cpm_i$

$$CPM_i^{cpu} = \left\lceil \frac{70 - b_i^{cpu}}{a_i^{cpu}} \right\rceil \quad (3-4)$$

$$CPM_i^{mem} = \left\lceil \frac{70 - b_i^{mem}}{a_i^{mem}} \right\rceil \quad (3-5)$$

$$threshold_i^{CPM} = min\{CPM_i^{cpu}, CPM_i^{mem}\} \quad (3-6)$$

其中a_i^{cpu}和b_i^{cpu}为 CPM 数据与 CPU 使用率之间线性关系的斜率与偏置项，a_i^{mem}和b_i^{mem}为 CPM 数据与 CPU 使用率之间线性关系的斜率与偏置项。

每个微服务i在时间t的容器数量$c_i(t)$可由 CPM 的阈值和t时刻的预测值$P_i(t)$得到：

$$c_i(t) = \left\lceil \frac{P_i(t)}{threshold_i^{CPM}} \right\rceil \quad (3-7)$$

最后，将计算得到的每个服务在将来一天内的 96 个容器数量的预测值存入数据库，由自动扩缩算法处理。

3.2.2 自动扩缩算法

微服务的工作负载在连续的时间间隔之间波动可能会很大，有可能当前时刻负载较高，而下一时刻的负载较低，在下一时刻的负载又会很高。如果简单地按照当前时刻的预测值进行扩缩，可能会导致微服务的容器频繁的扩展和缩减。为了解决这一问题，需要对自动扩缩的算法进行优化。

本文通过每一个小时一次的定时任务运行自动扩缩算法。每次算法需要获取当前时刻微服务i正在运行的容器数量$Replicas_i$、当前服务真实的 CPM 数据T_i、当前时刻及以后的一个半小时的容器数量的最大预测值C_i。

$$C_i = max\{c_i(t)\} \quad (3-8)$$

首先计算得到当前微服务所需副本的最小数量M_i。

$$M_i = max\left\{\left\lceil \frac{T_i}{threshold_i^{CPM}} \right\rceil, 1\right\} \quad (3-9)$$

再比较$Replicas_i$、C_i、M_i三者的差值，从而判断如何进行自动扩缩。

当$C_i \geq M_i$且$C_i \neq Replicas_i$时，需要进行扩容，需要将微服务的容器数量扩容为C_i。

当$C_i < M_i$且$Replicas_i < M_i$时，需要进行扩容，需要将微服务的容器数量扩容为M_i，并且在 15 分钟后对该服务再次运行本算法。此时的预测的负载数据小于当前实际的负载，预测出现偏差，为了保证服务的稳定运行需要先将服务扩容到阈值内的负载水平，并在 15 分钟后再次判断服务负载是否回到预测的水平。

当$C_i < M_i < Replicas_i$时，不需要进行扩缩容，并且在 15 分钟后对该服务再次运行本算法。

其他情况不需要进行扩缩容操作。

3.3 微服务主动式扩缩的实现

3.3.1 模块划分

要实现服务主动式的扩缩，需要获取并处理负载指标数据、训练预测模型、获取预测数据、计算扩缩数量、执行扩缩任务等步骤。根据微服务架构的思想，本文将不同的任务拆分成单独的模块，每个模块作为单独的服务运行，不同的业务功能之间互相解耦，不同的功能模块各司其职。这样即使某一个模块的服务出现问题也不会影响其他模块的正常运行，同时，不同的功能模块可以用不同的语言开发，比如这里训练模型和预测数据使用 python 语言开发，其他模块使用 Java 语言开发。

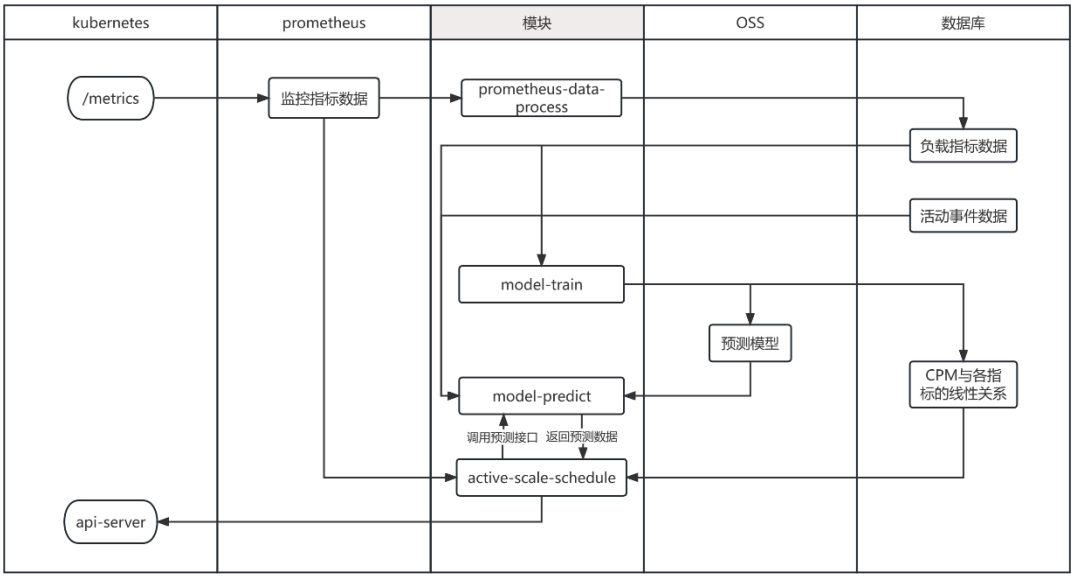

图 3-4 主动式扩缩架构泳道图

如图 3-4 所示，这里将执行微服务主动式扩缩的程序划分为以下几个模块：

(1) 负载指标数据获取与处理模块（prometheus-data-process）。每次模型的训练都需要大量的数据，如果数据都直接从 Prometheus 中获取数据，会对 Prometheus 造成很大的压力，并且由于 Prometheus 返回的数据格式是 JSON 格式不如结构化的数据方便处理。为了处理这一问题，该模块负责周期性地从 Prometheus 中获取监控到的负载指标数据，并将 JSON 数据转为结构化的数据保存到数据库中，以便后续使用。

(2) 模型训练模块（model-train）。该模块负责负载预测模型的训练，训练数据从数据库中获得，其数据包括了负载指标数据获取与处理模块保存的负载指标数据和活动事件数据。模型训练完成后，将训练好的模型保存至 OSS 中，同时计算各服务 CPM 指标与其他指标的线性关系并将结果保存到数据库中供预测时使用。

(3) 负载预测模块（model-predict）。该模块提供预测接口供调用，该接口从 OSS 中获取相应的模型数据，并从数据库中获取最新负载指标数据以及活动事件数据进行负载指标的预测，并返回对负载数据的预测结果。

(4) 主动扩缩调度模块（active-scale-schedule）。该模块调用负载预测模块获取预测数据，从 Prometheus 中获取服务当前的状态，并根据 3.2.2 节中描述的算法计算扩缩数量，然后调用 Kubernetes 的 api-server 接口执行扩缩。

3.3.2 负载指标数据的监控

本文通过 Prometheus 监控 Kubernetes 中的微服务工作负载的 CPU 利用率、内存利用率和 CPM 指标。

Kubernetes 使用 cAdvisor 收集容器的性能指标和资源利用率数据，例如 CPU、内存、磁盘、网络等，并将这些数据存储在 Kubernetes 集群中的 etcd 数据库中。这些数据可以用于监测和管理容器的资源使用情况，并支持自动伸缩、故障恢复和容器调度等功能[38]。Prometheus 可以通过 kubelet 的/metrics/cadvisor 路径获取 cAdvisor 采集到的 CPU 和内存数据，这里采集的频率设置为 5 秒一次，设置为 5 秒也为后续负载均衡算法的实现做准备。

在一个服务中暴露了 exporter 接口之后， Prometheus 就可以通过该接口获得服务的被调用次数。本文的社区项目使用 Java 语言进行微服务的开发，为了实现 exporter 的接口的暴露，需要加入 spring-boot-starter-actuator 依赖和 micrometer-registry-prometheus 依赖，如图 3-5 所示，并需要将相关配置加入到 spring-boot 的配置文件中，暴露 Prometheus 相关的接口，如图 3-6 所示。

最后需要在 Prometheus 中部署配置服务发现，使得 Prometheus 可以自动发现服务暴露的 exporter 接口，部署文件内容如图 3-7 所示。部署文件的配置内容

为：Prometheus 每 60 秒从目标服务中“拉”取一次数据，过短的时间间隔会影响服务的性能；exporter 接口路径需要设置为“/actuator/prometheus”，否则 Prometheus 寻找默认的接口路径“/metrics”；只会发现服务端口名配置为“svcport”的服务，并且只会选择服务部署配置中配置了含有标签“k8s.kuboard.cn/layer: svc”的服务，这两处的配置可根据业务需要灵活配置，这样可以灵活配置哪些服务需要被监控哪些服务不用。将部署文件部署到 Kubernetes 后 Prometheus 即可监控到服务中暴露 exporter 接口的数据。

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

图 3-5 spring boot 添加 Prometheus 依赖

```yaml
management:
  endpoints:
    web:
      exposure:
        # 打开 Prometheus 的 Web 访问 Path
        include: prometheus
  metrics:
    # 下面选项建议打开，以监控 http 请求的 P99/P95 等，具体的时间分布可以根据实际情况设置
    distribution:
      sla:
        http:
          server:
            requests: 1ms,5ms,10ms,50ms,100ms,200ms,500ms,1s,5s
    # 在 Prometheus 中添加特别的 Labels，以便于在 Grafana 中进行过滤
    tags:
      application: ${spring.application.name}
```

图 3-6 spring boot 暴露 Prometheus 接口配置

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: springboot-monitor
  namespace: default
spec:
  endpoints:
    - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
      interval: 60s
      path: /actuator/prometheus
      port: svcport
  namespaceSelector:
    any: true
  selector:
    matchLabels:
      k8s.kuboard.cn/layer: svc
```

图 3-7 Prometheus 配置自动发现 spring boot

服务暴露的监控数据中，包含 http_server_requests_seconds_count 指标，该指标统计了服务的工作负载中不同接口被调用的总次数，CPM 数据通过计算该指标在一分钟间隔的差值即可得到。

3.3.3 负载指标数据获取与处理

Prometheus 的 API Clients 对外提供了两种获取监控数据的 http 接口，一种是瞬时值的查询接口“/api/v1/query”，另一种是范围值的查询接口“/api/v1/query_range”。接口需要传递的参数有：promQL 语句、查询起始时间、查询结束时间、查询时间的步长，获取每个指标数据的 promQL 语句如下：

服务的 CPM 指标：

sum by (application) (increase(http_server_requests_seconds_count{uri!="/actuator/prometheus"}[2m]) / 2)

Pod 的 CPM 指标：

sum by (application, pod) (increase(http_server_requests_seconds_count{uri!="/actuator/prometheus"}[2m]) / 2)

CPU 使用率指标：

sum(irate(container_cpu_usage_seconds_total{container =~"thesis-.*"}[2m])) by (container, pod) / (sum(container_spec_cpu_quota{container =~"thesis-.*"}/100000) by (container, pod)) * 100

内存使用率指标：

sum (container_memory_working_set_bytes{container =~"thesis-.*"}) by (container, pod) / sum(container_spec_memory_limit_bytes{container =~"thesis-.*"}) by (container, pod) * 100 < 200

API Clients 接口返回 JSON 格式的数据，为了方便数据的读取和操作，本文将 JSON 格式的数据转为结构化数据存入数据库中，表结构如表 3-1 所示。其中 record_type 用于区分当前记录的 record_value 字段值的类型：CPM、CPU 使用率或内存使用率；service_name 用于区分当前记录的是哪个服务的数据；instance 用于区分当前记录的是某一服务的具体哪个实例的数据；record_value

为当前指标的记录值；record_time 代表当前指标的记录时间。

表 3-1 t_prometheus_record 表结构

序号	列名	类型	备注
1	id	int	自增主键
2	record_type	varchar(20)	记录数据类型
3	service_name	varchar(100)	服务名称
4	instance	varchar(100)	实例名称
5	record_value	double	记录值
6	record_time	timestamp	记录时间

3.3.4 活动事件的记录与获取

在 3.1.3 节中提到本文项目中的活动分为广告推广、站内活动、折扣活动这 3 种类型。为了方便记录和获取活动事件，需要将活动事件维护到数据库中的活动记录表中，表结构如表 3-2 所示。其中 event_type 字段用来区分活动的类型，这里设定该字段值对应的类型：1-广告推广、2-站内活动、3-折扣活动。

表 3-2 t_event 表结构

序号	列名	类型	备注
1	id	int	自增主键
2	event_type	timestamp	活动类型
3	start_time	timestamp	活动启始时间
4	end_time	timestamp	活动结束时间
5	create_time	timestamp	记录创建时间

在每次做模型训练和预测时，训练和预测的数据都是一段连续的时间序列数据，为了判断这段时间内每条数据是否有活动正在进行，可通过 SQL 语句查询 t_event 表获取：select * from t_event where start_time >= #{start_time} and end_time <= #{end_time}。其中#{start_time}为需要查询的起始时间，时间的值与每次训练和预测的时候从 t_prometheus_record 表中获取数据的起始时间相同；#{end_time}为需要查询的结束时间，如果是模型训练，则时间的值与每次训练和预测的时候从 t_prometheus_record 表中获取数据的结束时间相同，如果是预测，则时间的值需要在查询 t_prometheus_record 表传入的结束时间的基础上再增加一天的时间，因为每次预测需要预测后面一天的数据，模型在做预测时也需要通过预测的时间是否处在活动期间这一特征进行预测。

3.3.5 模型的训练

模型需要以 15 分钟为间隔的指标数据，而监控得到的指标数据时间间隔为 1 分钟，在训练和预测的时候，首先需要将监控到的指标数据每 15 条数据聚合成一条。其中 CPM 指标聚合数据为 15 个 CPM 数据中的最大值，CPU 使用率和内存使用率指标的聚合数据取与 CPM 最大值对应数据的 CPU 使用率和内存使用率，时间指标取 15 条数据中最后的时间。

模型每一周将所有需要预测的服务负载数据都训练一次，每次训练将上一周监控到的负载指标数据作为训练数据，一次训练一个服务，一个服务保存为一个模型，直到将所有需要预测的服务都训练完毕。每次训练好的模型保存到阿里云的 OSS 中并以服务名称命名模型文件，以区分不同服务的预测模型。

表 3-3 t_metrics_linear_relation 表结构

序号	列名	类型	备注
1	id	int	自增主键
2	service_name	varchar(50)	服务名称
3	cpu_slope	double	cpm 与 cpu 使用率之间的线性关系的斜率
4	cpu_offset	double	cpm 与 cpu 使用率之间的线性关系的偏移值
5	memory_slope	double	cpm 与内存使用率之间的线性关系的斜率
6	memory_offset	double	cpm 与内存使用率之间的线性关系的偏移值
7	create_time	timestamp	记录创建时间
8	update_time	timestamp	记录更新时间

同时，在训练预测模型之外，还需要通过线性回归计算每个服务的 CPM 指标与 CPU 使用率和内存使用率的线性关系，并将线性关系中的斜率和偏移值保存到数据库，表结构如表 3-3 所示。

3.3.6 负载预测模块

为了方便获取负载预测数据，预测模块通过 Flask 搭建 web 应用服务，对外提供获取负载预测数据的 http 接口。Flask 是一个 Python 的轻量级 Web 应用程序框架，提供了构建 Web 应用程序的基本组件和工具，较其他同类型框架更为

灵活、轻便、安全且容易上手[39]。

预测程序接口接收服务名作为参数，以判断需要预测的是哪个服务。首先，预测程序需要根据传入的服务名从 OSS 中获取指定的预测模型，并从数据库中获取当前时间的半天前到一天后期间存在的活动事件数据和与传入的服务名所对应服务的最近半天的负载指标数据，并将指标数据聚合成以 15 分钟为间隔的共 48 条数据。将负载指标数据和活动时间数据传入预测模型，即可得到服务的负载预测数据，最后接口将得到的预测数据返回。

3.3.7 操作 Kubernetes

主动扩缩调度模块由 Java 语言开发，通过 Java 操作 Kubernetes 的 api-server 一般有两种方式：通过配置证书或通过 token，本文使用配置证书的方式。首先本文选择使用 fabric8io 的 kubernetes-client 包对 api-server 进行操作，并将 Kubernetes 的证书配置到项目中，证书可从 Kubernetes 的 master 节点中获取，证书配置和证书路径的对应关系如下：

client-crt:/etc/kubernetes/pki/apiserver-kubelet-client.crt

client-key:/etc/kubernetes/pki/apiserver-kubelet-client.key

ca-crt:/etc/kubernetes/pki/ca.crt

需要注意的是，在创建与 api-server 的连接时，传入的 client-crt 必须先用 Base64 对证书内容加密，否则会提示 input null。

正确配置之后，通过 KubernetesClient 即可对 Kubernetes 进行操作，比如获取或改变当前指定服务的 pod 数量，如下代码所示：

获取 pod 数量：

kubernetesClient.apps().deployments().inNamespace(namespace).withName(name).scale().getSpec().getReplicas();

改变 pod 数量：

kubernetesClient.apps().deployments().inNamespace(namespace).withName(containerName).scale(scale, true);

3.3.8 主动扩缩任务的实现

主动扩缩模块运行两个定时任务，第一个定时任务每一个小时执行一次，负责对所有需要实现主动式扩缩的服务执行主动扩缩算法。该任务首先获取所有需要实现主动式扩缩的服务名集合，再遍历所有的服务名依次执行主动扩缩算法，具体过程如算法 3-1 和算法 3-2 所示，算法原理由 3.2 节所述。其中需要说明的是，如果当前预测需要的副本数量少于当前服务负载所需最少副本数时，为保证服务正常运行，将该服务名加入主动扩缩重试任务队列，重试任务队列由第二个定时任务处理。

算法 3-1 主动扩缩定时任务

算法 3-1 主动扩缩定时任务

1. 获取所有需要主动扩缩的服务名
2. 将 Prometheus 中最新数据保存到数据库
3. For（服务名）
4. 　　主动扩缩算法（服务名）
5. Endfor

算法 3-2 主动扩缩算法

算法 3-2 主动扩缩算法

1. 预测 CPM = 获取服务 CPM 预测数据（服务名）
2. 线性关系值 = 获取服务 CPM 与各指标的线性关系（服务名）
3. C_i = 计算预测需要的副本数量（预测 CPM，线性关系值）
4. $Replicas_i$ = 获取服务当前的副本数（服务名）
5. M_i = 计算当前服务负载所需最小副本数（服务名）
6. If $C_i \geq M_i$ & $C_i \neq Replicas_i$
7. 　　调整服务副本数量为C_i
8. Elseif $C_i < M_i$ & $Replicas_i < M_i$
9. 　　调整服务副本数量为M_i
10. 　　当前服务加入主动扩缩重试任务队列
11. Elseif $C_i < M_i < Replicas_i$
12. 　　当前服务加入主动扩缩重试任务队列
13. Else
14. 　　不需要进行操作

第二个定时任务每十五分钟执行一次，负责对主动扩缩重试任务队列中的服务再次执行主动扩缩算法。该任务首先会判断主动扩缩重试任务队列是否为空，如果为空则跳过本次定时任务，如果不为空则重试任务队列出队，执行主动扩缩

算法，直到队列为空，具体操作如算法 3-3 所示。添加此重试机制的原因是在主动扩缩定时任务执行时，可能因为服务请求突然变多或其他原因导致服务当前的负载高于模型预测的负载，为了保证服务正常运行，不能对服务进行缩容，但可能这种情况可能只持续几分钟或更短的时间，后续服务的负载又会回到正常水平，通过重试机制，使得本文的主动扩缩算法可以一定程度地适应服务负载突变的情况，并在负载突变之后及时释放多余的资源供其他服务使用。

算法 3-3 主动扩缩重试定时任务

算法 3-3 主动扩缩重试定时任务

```
If 重试任务队列为空
    Return
将 Prometheus 中最新数据保存到数据库
While（重试任务队列非空）
    服务名 = 重试任务队列出队
    主动扩缩算法（服务名）
Endwhile
```

3.4 本章小结

本章首先介绍了为了满足 SLA，微服务的两种水平扩缩的方式：反应式和主动式，并对比了反应式方法和主动式方法的优劣，本章选用主动式的方法对微服务进行扩缩。实现主动式的扩缩需要解决两个问题，一是需要对服务负载数据准确地预测，二是需要扩缩算法基于负载预测的数据对微服务进行合理的扩缩。本章经过对时间序列预测的研究，选用 Informer 实现对长时间负载数据的准确预测，提出了将 CPM 作为反应服务负载的指标，并结合本文的社区项目中活动事件对服务负载的影响，在 Informer 对输入的 embedding 操作中加入对活动事件的嵌入，提升了模型对项目的预测准确度，同时为防止服务的频繁波动提出了本文的自动扩缩算法，最后，对主动式扩缩的实现的过程进行了说明。

第四章 微服务的客户端式动态负载均衡设计

尽管采用主动式扩缩技术可以有效地降低服务处于高负载状态的风险，但是模型预测的数据仍然存在偏差，而且模型也无法准确预测超出负载规律的突发请求所带来的高负载情况。在服务副本数量没有增加且服务处于高负载的情况下，一个好的负载均衡算法可以更加充分地利用服务资源，让服务在有限的服务器资源的情况下拥有更高的并发量，处理更多的用户请求。

负载均衡是微服务架构中重要且不可或缺的一部分，本文的项目在高并发场景监测下，经常会出现服务集群负载不均的情况。本章节研究了微服务架构下客户端的负载均衡方式，并且通过改进熵值法提出一种微服务架构下客户端的动态权重的负载均衡算法，该算法可以有效解决本文社区项目中在高并发场景下负载不均的情况。

4.1 动态负载均衡的架构设计

在 2.6 节中介绍了当前微服务架构下常用的客户端的负载均衡算法，这些常用的算法都是静态的或者伪动态的负载均衡算法，与动态的负载均衡算法相比，静态的负载平衡虽然简单但是效果要差许多。当前的算法使用的数据都是从注册中心获取，而服务发送给注册中心的心跳包中并不包含 CPU 使用率、内存使用率等服务的状态数据，想要实现动态的负载均衡算法，就需要先获取到这些数据。

本文的负载均衡算法设计为客户端的动态权重的负载均衡算法，算法的整体架构如图 4-1 所示。与服务端的负载均衡不同，在客户端的负载均衡中每个客户端都运行一个负载均衡算法，为了避免重复计算也为了减小权重计算过程对服务性能的影响，本文将权重计算单独作为一个服务运行。

上文提到 Prometheus 获取的服务相关数据中已经包含了各容器的 CPU 使用数据和内存使用数据，权重计算服务将从 Prometheus 中获取服务的动态数据并计算每个服务不同副本的权重数据，并将数据保存到 Redis 中供负载均衡算法使用。当消费者服务需要调用其他生产者服务时，负载均衡算法从注册中心中取出

消费者服务不同副本的 IP 地址和端口号，并由服务名、IP 地址和端口号从 Redis 中获取相应的权重数据，得到权重数据后负载均衡算法便可通过动态权重算法选择消费者服务的副本发送调用请求。

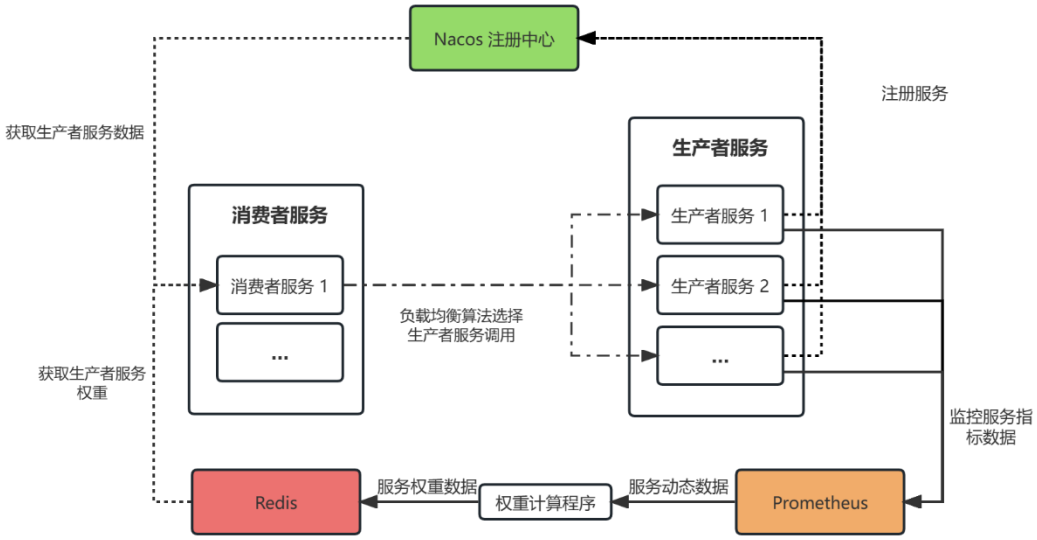

图 4-1 通过 Redis 实现客户端的负载均衡

Redis 是完全开源的，遵守 BSD 协议，是一个高性能的 key-value 数据库，得益于 IO 多路复用机制，Redis 有着非常好的性能，其读的速度可达到 10000 次/s，写的速度也可达到 81000 次/s[40]。同时相比于 memcache，Redis 提供了更丰富的数据类型，在数据存储和读取服务状态时也更加方便。这里使用 Redis 作为中间件保存服务的权重数据不仅让负载均衡算法与权重计算服务解耦也减少了负载均衡算法的性能损耗。

在微服务架构下，与微服务的响应时间相比，CPU 和内存利用率与微服务工作负载的相关性更强，且本文的社区项目服务都是 CPU 密集型和内存密集型的服务，且服务的工作负载与网络吞吐量和磁盘利用率等其他服务状态数据关系不大，所以这里选用 CPU 和内存指标计算服务权重。

需要注意的是，因为服务运行在容器中，由于容器的特性，在容器中的服务代码只能获取到宿主机的 CPU 和内容使用情况，所以这里无法通过服务代码获取服务所在容器的 CPU 利用率和内存利用率，需要通过 Prometheus 或其他监控方式获取。如果是传统的 SOA 或其他架构，服务代码直接运行在宿主机上可以

由服务代码上报服务的相关动态数据，可以减小 Prometheus 的压力。

4.2 基于熵值法的动态权重计算

熵值法（Entropy method）是一种客观赋权法，是信息论方法的一种，它根据各项指标的信息熵的大小来确定指标的权重，信息熵越小，代表指标的离散程度越大，该指标对综合评价的影响越大，权重也就越大[41]。负载均衡算法的目的就是要让不同服务的指标数据尽量均衡，对于 CPU 和内存指标数据来说，指标的熵值越小代表该指标越不均衡，算法就需要更优先平衡该指标。负载均衡算法可通过熵值法得到不同指标的熵值，由各指标的熵值计算出不同服务的权重，通过权重将请求分配到不同的服务副本使目标服务集群达到负载均衡的目的。

设x_{ij}为第i个服务副本的第j个指标的数据，熵值法计算服务权重的步骤如下：

(1) 原始数据归一化

因为只有 CPU 可用率和内存可用率两个指标，并且这两个指标只可能为正数且都不是逆向指标，这里可以省略指标数据归一化的步骤。

(2) 计算第 j 项指标下第i个服务副本的指标值与该指标总值的比率

$$p_{ij} = \frac{x_{ij}}{\sum_{i=1}^{n} x_{ij}} \tag{4-1}$$

(3) 计算第 j 项指标的熵值

$$e_j = -k \sum_{i=1}^{n} p_{ij} ln(p_{ij}) \tag{4-2}$$

其中 k=1/ln(n)>0 满足$e_j > 0$。

(4) 计算指标信息熵值的冗余度

$$d_j = 1 - e_j \tag{4-3}$$

(5) 计算各项指标的权重系数

$$w_j = \frac{d_j}{\sum_{j=1}^{m} d_j} \tag{4-4}$$

上述熵值法计算步骤获取的权重系数w_j由不同指标数据的离散程度获得，但

是这里的权重系数并不适用于计算服务的动态权重，比如当服务的 CPU 平均可用率只有 10%，而内存平均可用率还有 50%时，这个时候内存可用率指标对不均衡的可容忍程度要比 CPU 可用率对不均衡的可容忍程度高，也就是说 CPU 可用率指标的重要程度比内存可用率指标要大，权重应该更偏向于 CPU 使用率的指标。因此本文通过当前指标的可容忍程度对指标权重系数进行加权。

(6) 获取每项指标的加权权重

$$a_j = \frac{\sum_{i=1}^{n}(100-x_{ij})}{n \times 100} \quad (4-5)$$

将每个指标的平均使用率作为加权的权重，也就是将 CPU 平均使用率和内存平均使用率作为评价指标，使用率越高指标的偏移权重越大，从而数值越高的指标对不平衡的可容忍程度越低，数值越低的指标对不平衡的可容忍程度越高。因为 CPU 可用率和内存可用率都是 0 至 100 范围内的数值，分母乘 100 后，可将a_j限制为 0 至 1 范围的数值。

(7) 重新计算各项指标的权重系数

$$w_j = \frac{w_j \times a_j}{\sum_{j=1}^{m} w_j \times a_j} \quad (4-6)$$

(8) 计算每个服务副本的权重

$$w_i = n \times \sum_{j=1}^{m} \frac{x_{ij}}{\sum_{i=1}^{n} x_{ij}} \times w_j \quad (4-7)$$

计算完毕后，每个服务各副本的权重和应该等于当前服务的副本总数$\sum_{i=1}^{n} w_i = n$。

4.3 客户端的动态权重负载均衡算法的实现

4.3.1 算法模块架构的设计

本文的社区项目由多个微服务组成，每个业务服务都需要配置为本文的负载均衡算法。为了减少本文的负载均衡算法对服务的业务代码的侵入，也为了代码重用，这里把配置本文定义的负载均衡算法和配置 Redis 的操作抽取出来作为单

独的模块，需要使用这些功能的微服务将需要的模块加入项目依赖即可，模块的依赖关系如图 4-2 所示。这样不仅简化了项目的配置过程，还减少了大量重复代码，而且当后续对算法进行修改测试升级时也更加方便，只用修改一个地方即可。

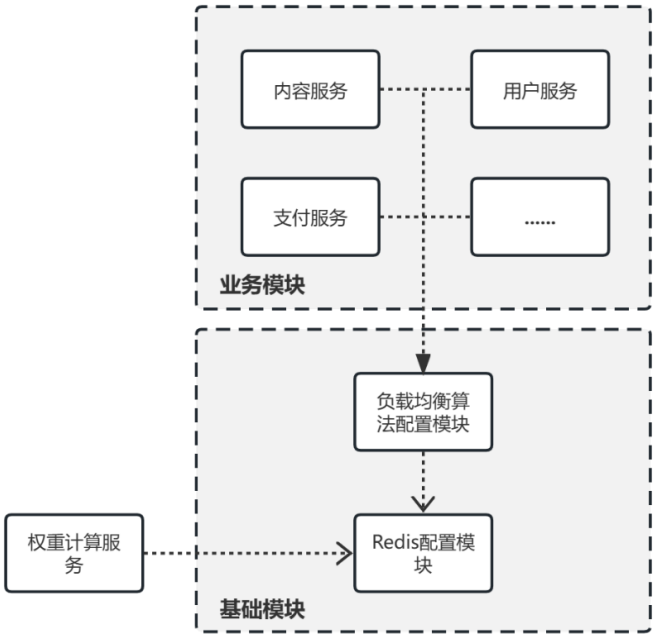

图 4-2 项目模块依赖图

4.3.2 权重计算服务的实现

权重计算服务只运行着一个定时任务，该定时任务每 5 秒执行一次，每次任务执行都将获取所有服务的负载数据，并根据本文改进的熵值法计算每个服务下不同副本的权重值，最后将计算好的权重值保存到 Redis 中，供负载均衡算法使用。

为了节约计算资源，权重计算程序只对副本数量大于 1 对服务计算权重，并且对负载数据设置了阈值，CPU 使用率和内存使用率的阈值都设置为 70%。如果判断所有的服务副本的 CPU 使用率和内存使用率都低于 70%，则不再通过熵值法计算每个副本的权重，直接向 Redis 中添加服务没有超过阈值的标记，这个标记被下文的负载均衡算法所使用。其中保存到 Redis 中的 key 由指定前缀拼接服务名组成，value 由服务名、服务 IP 地址、服务端口号、权重值、权重计算时间和是否超过阈值的标识组成。具体的执行内容如算法 4-1 所示。

算法 4-1 权重计算程序

算法 4-1 权重计算程序

```
所有服务副本的负载数据 = 从 Prometheus 中获取服务的负载数据
map<服务名，负载数据集合> = 按服务名进行分组(所有服务副本的负载数据)
For map
    If 负载数据集合长度 > 1
        redisKey = 生成服务权重的 redisKey(服务名)
        If 负载没有超过阈值(负载数据集合)
            标记 Redis 中对应的权重数据没有超过阈值(redisKey)
        Elseif
            每个副本的权重值 = 熵值法计算权重(负载数据集合)
            保存权重数据到 Redis(redisKey, 每个副本的权重值)
Endfor
```

4.3.3 负载均衡算法的实现

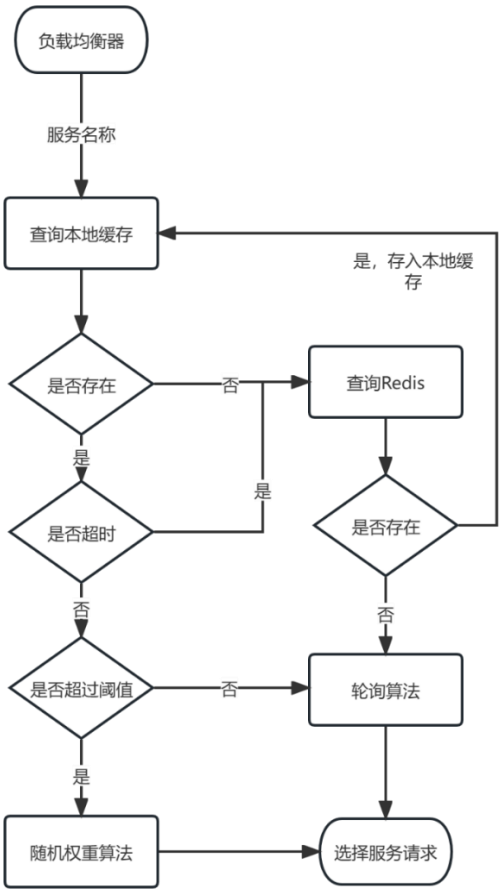

图 4-3 阶段式动态随机权重算法流程图

与默认的轮询算法相比，权重算法要经历产生随机数、遍历权重列表等步骤，会占用相对更多的计算资源。在目标服务各个副本的负载都不高的情况下，服务

负载的不均衡对服务质量的影响不大。本文的负载均衡算法设计为分段式的算法，通过指定前缀拼接目标服务的服务名组成 Redis 的 Key，用这个 Key 从 Redis 中获取目标服务各副本的权重数据和是否超过阈值标识，当是否超过阈值标识为 false 时，使用默认的轮询算法，否则使用动态权重算法。

在做负载均衡时，为了防止频繁地请求 Redis，本文的负载均衡算法会从 Redis 中同步目标服务的权重数据到本地缓存，并优先使用本地缓存的权重数据，如果本地缓存中数据的权重计算时间与当前时间的差值大于 5 秒则需要再次同步 Redis 中的数据。具体流程如图 4-3 所示。

本文的基于改进熵值法的随机动态权重算法采用 Spring Cloud LoadBalancer 实现，Spring Cloud LoadBalancer 默认使用轮询算法，通过相关的配置即可实现自定义的负载均衡算法。首先，需要在 RestTemplate 方法上添加@LoadBalanced 注解开启负载均衡功能，然后在负载均衡策略配置类中配置自定义的负载均衡策略，如图 4-4 和图 4-5 所示。MyWeightedRule 类即为本文的负载均衡算法实现类。

```java
@Configuration
@LoadBalancerClients(defaultConfiguration = {LoadBalancedConfig.class})
public class RestTemplateConfig {
    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }
}
```

图 4-4 RestTemplate 配置

```java
@Configuration
public class LoadBalancedConfig {
    @Bean
    ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(Environment environment,
                                                            LoadBalancerClientFactory loadBalancerClientFactory) {
        String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        // 返回内容为自定义负载均衡的配置类
        return new MyWeightedRule(loadBalancerClientFactory.getLazyProvider(name, ServiceInstanceListSupplier.class),
                name);
    }
}
```

图 4-5 自定义负载均衡算法配置

4.4 本章小结

本章节设计了一种在微服务架构下客户端式的动态权重的负载均衡算法。为了实现该负载均衡算法，选择了 CPU 可用率和内存可用率两个指标用于权重的

计算，并引入权重计算服务和 Redis 用于权重数据的计算、保存和获取。基于熵值法计算服务的权重数据，并实现了阶段式的动态权重的负载均衡算法。本文对熵值法进行了改进，通过添加各项指标的偏移权重使得计算的权重数据更适用于负载均衡的场景。

第五章 微服务主动式扩缩与负载均衡算法的部署与测试

为了验证本文工作内容的有效性，本章节将本文的研究内容部署到一个社区项目环境中，用本文的负载预测模型与其他预测模型的预测效果进行对比测试，验证预测模型预测精度的提升和主动式扩缩的效果，最后将本文的负载均衡算法与其他流行的客户端的负载均衡算法进行对比测试，验证本文负载均衡算法的有效性。

5.1 系统架构设计与环境的搭建

5.1.1 社区项目系统架构设计

本文将上述改进加入到本文的社区项目系统中，系统架构图如图 5-1 所示，系统的架构由以下几部分组成：

网关层（gateway）：系统对外唯一的入口，系统使用 Ingress 作为项目服务对外的负载均衡器，本文通过部署 Spring Cloud Gateway 的方式来构建 API 网关，网关通过对请求地址的辨识，将请求路由转发到相应的业务服务进行处理。同时，借助 Gateway，还能够对请求进行权限验证、监控、缓存等非业务功能的逻辑处理。为了不让网关成为系统的瓶颈，这里设置了至少 3 个副本同时处理外部请求。

业务服务层：社区项目中提供的服务都在这层，主要包括：内容服务、内容上传服务、商店服务、支付服务、用户服务、后台服务等。这些服务由于只会被内网调用，且项目使用客户端的负载均衡方式，这里可以不用对服务设置 service 或负载均衡器。这些服务的副本可能会有一个也可能会有多个，副本的数量由主动扩缩模块调配。此外为了实现本文的负载均衡算法，这里还部署了权重计算服务以保证算法的正常运行。

注册中心和配置中心：项目使用 Nacos 作为注册中心和配置中心。

数据存储：系统中使用 Mysql 和 OSS 存储项目生产出的数据，项目中的大部分数据都存储在 Mysql 中，OSS 中存储了用户上传的图片数据以及主动式扩

缩模块中的预测模型数据。

中间件：系统中使用了 Redis 和 RabbitMQ 作为项目的中间件。Redis 用于存放项目中的缓存信息、负载均衡算法产生的服务负载和服务的权重信息、分布式业务代码中的锁信息等。RabbitMQ 作为系统的消息中间件，用于实现服务之间的异步调用、发布订阅、流量削峰、项目解耦等。

监控和预警层：通过 Prometheus 和 Grafana 实现。Prometheus 通过收集存储 Kubernetes 中 pod 和服务的监控指标数据，为本文微服务主动式扩缩的实现提供了数据来源。通过 Grafana 将 Prometheus 中监控到的数据以图形化的形式展示出来，为本文的改进提供了直观的展示。

主动式扩缩模块：该模块的具体内容可见本文的 3.3 节。

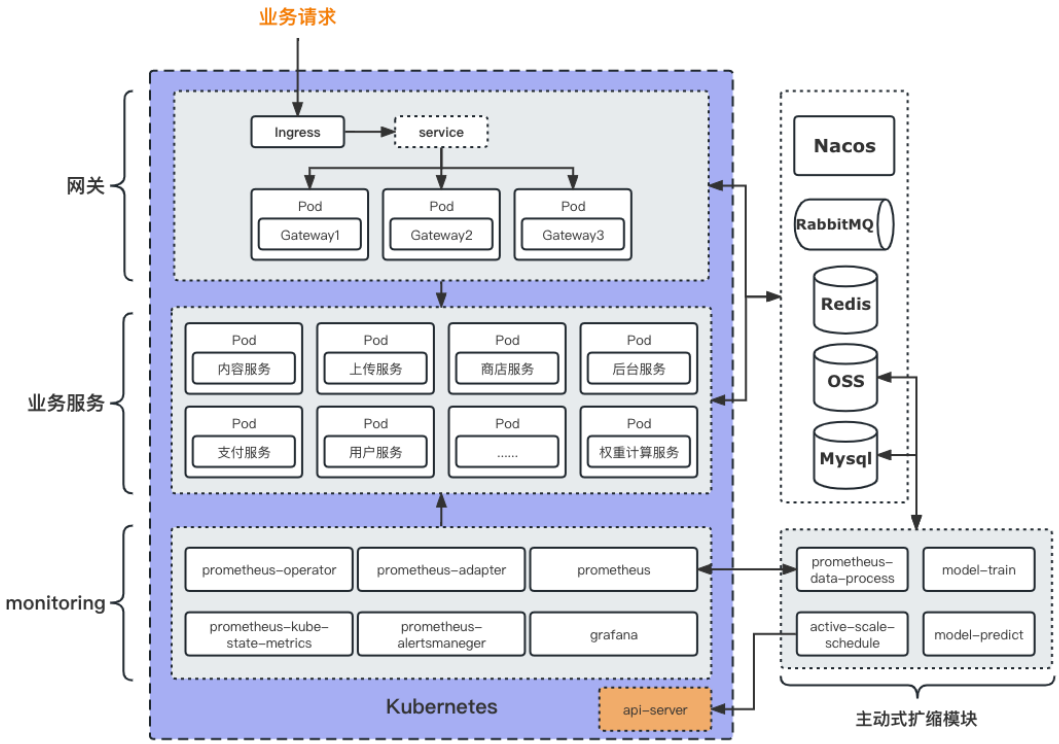

图 5-1 项目系统架构

5.1.2 系统环境搭建

本文的系统环境搭建在阿里云的环境中。Kubernetes 部署在 6 台服务器上，由 3 台 Master 节点和 3 台 Node 节点组成，如表 5-1 所示。通过 kubeadm 工具对 Kubernetes 进行部署，部署的 Kubernetes 版本为 1.23.6。

表 5-1 Kubernetes 集群环境节点配置

序号	Role	IP	CPU 型号	CPU 配置	内存配置
1	master	172.27.151.200	Intel(R) Xeon(R) Platinum 8269CY	4 核	8G
2	master	172.27.151.201	Intel(R) Xeon(R) Platinum 8269CY	4 核	8G
3	master	172.27.151.202	Intel(R) Xeon(R) Platinum 8269CY	4 核	8G
4	node	172.27.151.203	Intel(R) Xeon(R) Platinum 8269CY	8 核	16G
5	node	172.27.151.204	Intel(R) Xeon(R) Platinum 8269CY	8 核	16G
6	node	172.27.151.205	Intel(R) Xeon(R) Platinum 8269CY	8 核	16G

为了方便对项目服务的部署和管理，本文通过部署 Kuboard 工具以简化项目的部署过程。相较于 Kubernetes Dashboard 等其他 Kubernetes 管理界面，Kuboard 可以对微服务分层展示、对工作负载的展示更直观同时支持图形化界面对工作负载进行编辑，避免了对 YAML 文件的编辑。本文社区项目中的网关层和业务服务层中的工作负载都使用 Kuboard 进行部署和管理。

项目的系统架构中，网关、业务服务和 monitoring 模块部署在 Kubernetes 中，为了节省服务器资源，也为了减少项目的维护难度，Nacos、数据库、OSS 和其它的中间件都选择使用阿里云提供的对应服务。在选购时这些服务位置应和 Kubernetes 集群处于同一内网环境下，确保项目的服务质量不会受到连接这些服务资源产生的网络延迟的影响。

主动式扩缩模块搭建在本地的实验环境中，为了加快负载预测模型训练和预测的速度，其中 model-train 和 model-predict 模块需要 GPU 的支持。实验环境如表 5-2 所示：

表 5-2 主动式扩缩模块实验环境

序号	模块名	GPU 型号	CPU 型号	CPU	内存
1	model-train	NVIDIA GeForce RTX 3070 8G	Intel(R) Core(TM) i7-10700	8 核	32G
2	model-predict				
3	prometheus-data-process		Apple M1	8 核	16G
4	active-scale-schedule				

5.2 微服务主动式扩缩效果测试

5.2.1 评价指标

主动式扩缩的效果依赖于负载预测模型的性能，为了评价本文的负载预测模型，这里选用了两种常用的评价指标来衡量模型的预测精度：均方误差（Mean Squared Error，MSE）和平均绝对误差（Mean Absolute Error，MAE）。

MSE 的计算公式如下：

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \quad (5-1)$$

MAE 的计算公式如下：

$$MSE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n} \quad (5-2)$$

上述公式中，y_i代表第i个样本的真实值，$\hat{y}_i$代表第i个样本的预测值，n是样本数量。

5.2.2 对比分析

在 3.1.3 节中提到通过将项目中的活动事件编码加入到模型中以提升预测模型预测的准确性，为了验证这个改进的效果，这里选用 Informer 和 LSTM 模型作为基准模型，选用项目中的用户服务的负载数据进行模型训练，对比不同模型的 MSE 和 MAE 指标。

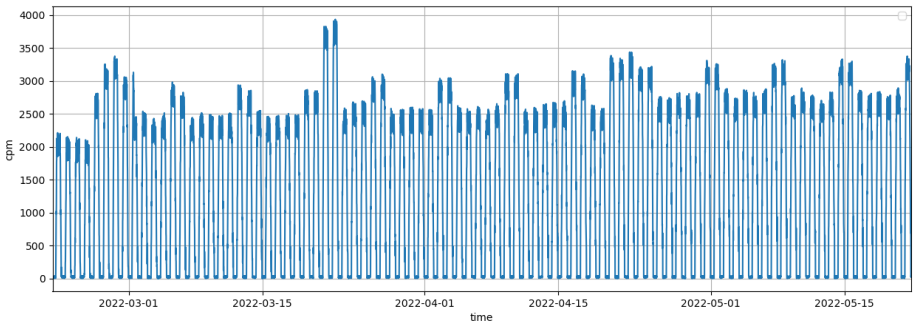

图 5-2 CPM 负载数据

这里选取了用户服务从 2022 年 2 月 21 号零点到 2022 年 5 月 22 号零点连

续 3 个月的负载数据进行模型的训练，训练的负载数据如图 5-2 所示。表 5-3 展示了这 3 个月时间内项目中的活动情况，包括了不同的活动类型和活动的开始和结束时间。可以发现，用户服务周末的负载要比工作日的高，并且在活动时间段内的服务的负载要高于平均水平，尤其在 3 月 21 号到 3 月 23 号期间同时存在广告推广活动和站内活动，两个活动同时进行使负载数据提升得更多。

表 5-3 活动事件表

活动类型	活动开始时间	活动结束时间
广告推广	2022-02-25 00:00:00	2022-03-01 12:00:00
广告推广	2022-03-21 00:00:00	2022-03-23 00:00:00
广告推广	2022-05-22 00:00:00	2022-05-24 12:00:00
站内活动	2022-03-21 00:00:00	2022-03-23 00:00:00
折扣活动	2022-04-20 00:00:00	2022-04-23 00:00:00

表 5-4 为本文改进后的预测模型与 Informer 模型和 LSTM 模型在预测序列长度分别为 48、96、192 和 384 的对比结果。可以看出，LSTM 模型在预测的序列长度较短时预测效果良好，当预测序列长度大于或等于 96 时预测精度明显下降。而本文的预测模型由于嵌入了经过编码后的活动事件数据，使得模型能够学习到活动事件对负载数据的影响，其 MSE 和 MAE 指标都明显优于其它两个模型，模型的预测精度得到提高。

表 5-4 预测模型对比结果

模型		L=48	L=96	L=192	L=384
LSTM	MSE	0.03819	0.12958	0.14144	0.29721
	MAE	0.13672	0.25135	0.24422	0.37144
Informer	MSE	0.03989	0.05195	0.065251	0.07138
	MAE	0.13668	0.15488	0.17163	0.17039
本文模型	MSE	**0.03299**	**0.04157**	**0.04644**	**0.05896**
	MAE	**0.12884**	**0.14039**	**0.14499**	**0.16035**

为了验证在活动事件的影响下，本文改进后的模型的预测数据与真实数据的拟合情况，这里将 5 月 21 号的数据作为三个模型的输入分别去预测 5 月 22 号的数据。由于 5 月 22 号是周六并且又是活动开始的第一天，其真实的负载要比工作日的高并且也要比没有活动的周末的负载高。图 5-3 为 LSTM 模型的预测结果，由于 LSTM 模型学习不到时间数据对预测数据的影响，模型的预测结果与真

实值相差很多。图 5-4 为 Informer 模型的预测结果，由于 Informer 的注意力机制并且加入了时间戳编码，Informer 学到了周末对用户服务负载的影响，但由于模型没有学到活动事件对负载的影响，模型的预测结果与真实值还有一定的差距。图 5-5 为本文改进后的模型，由于加入了活动事件编码，该模型的预测结果与真实数据基本拟合，这也证明了本文改进的有效性。

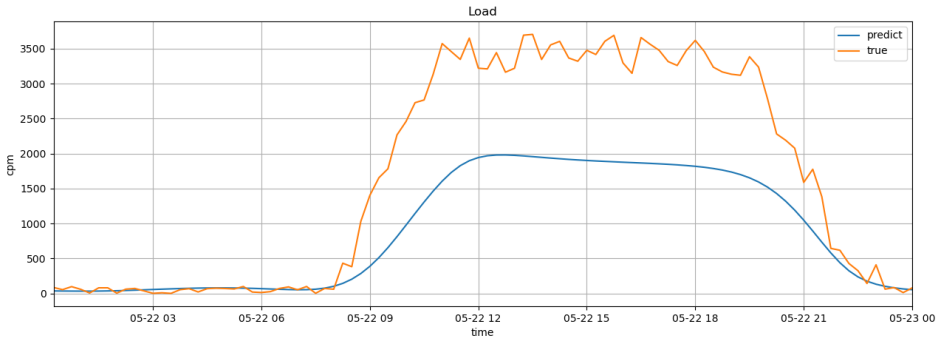

图 5-3 LSTM 模型的预测结果

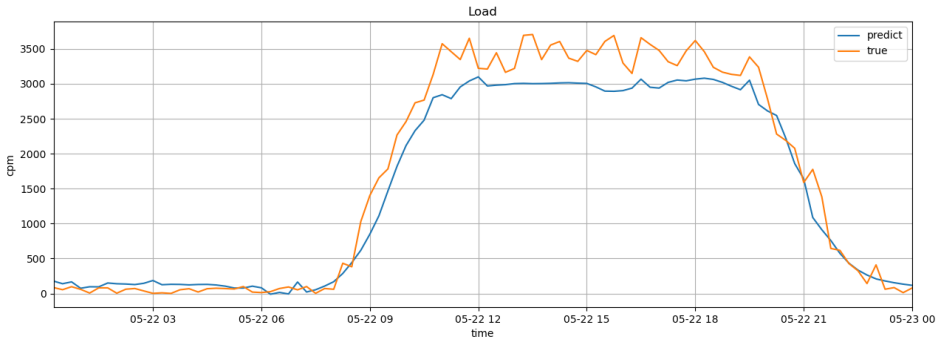

图 5-4 Informer 模型的预测结果

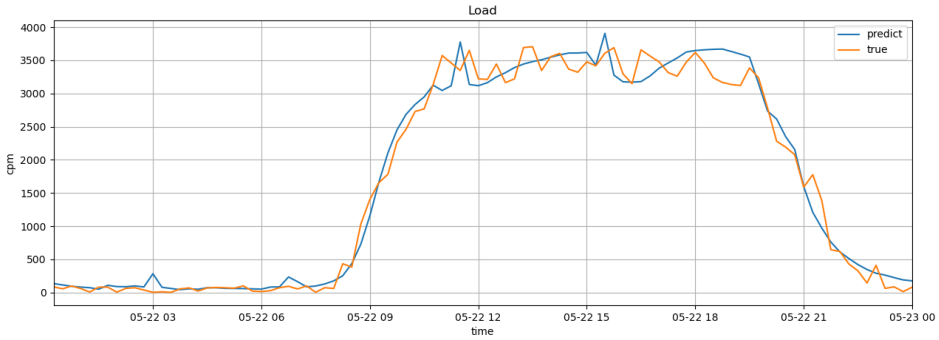

图 5-5 改进后模型的预测结果

图 5-6 展示了用户服务从 2022 年 4 月 23 号到 2022 年 6 月 22 号连续两个月的 CPU 负载数据，项目在 2022 年 5 月 23 号开启本文的主动式扩缩方案，图中

红色的竖线标明了具体的时间点。可以看出，在开启主动式扩缩方案之前经常出现因反应式扩缩的滞后导致的 CPU 负载的突增情况，在开启主动式扩缩方案之后，CPU 负载突增的情况明显减少，很大程度上缓解了反应式扩缩导致服务质量下降的问题。

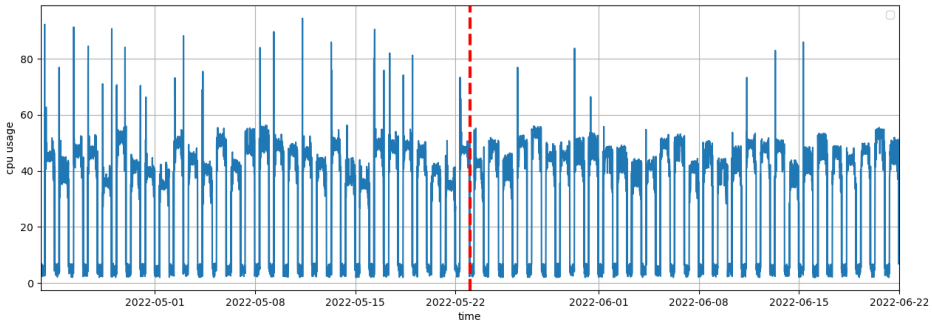

图 5-6 主动式扩缩改进效果

5.3 客户端的动态权重的负载均衡算法效果测试

本文选用 JMeter 对本文提出的客户端的动态权重的负载均衡算法进行测试。由于本文的算法是轮询算法和动态随机权重算法结合的算法，结合了两种算法的优势，这两种算法分别又是静态和动态的负载均衡算法，为了体现本文算法的效果，这里分别选取了常见的静态和动态的负载均衡算法与本文算法的负载效果进行对比，选取的算法包括：轮询算法、最小连接数算法和最短响应时间算法，同时选取了文献[42]中的 dyn 算法进行对比。

这里选取项目中的用户服务进行性能测试，设置用户服务的 Pod 副本数量为 3 个且副本分布在不同的 Node 节点上，并对容器资源限制为 2 核 1G。在无其它业务请求的情况下，测试请求调用网关，由网关执行负载均衡算法选择调用具体的用户服务。在该环境下，通过设置固定数量的并发请求测试用户服务，每次测试持续 5 分钟，并不断提升并发请求数量，获取每个算法的测试请求在不同并发数量情况下的平均响应时间和实际并发连接数，测试结果如图 5-7 和图 5-8 所示。

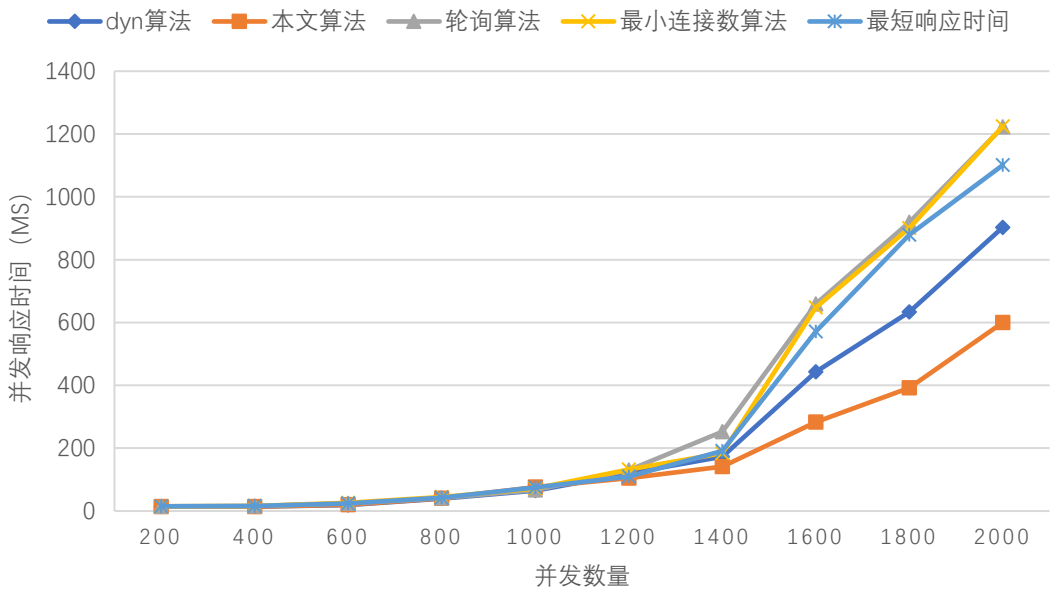

图 5-7 平均响应时间比较

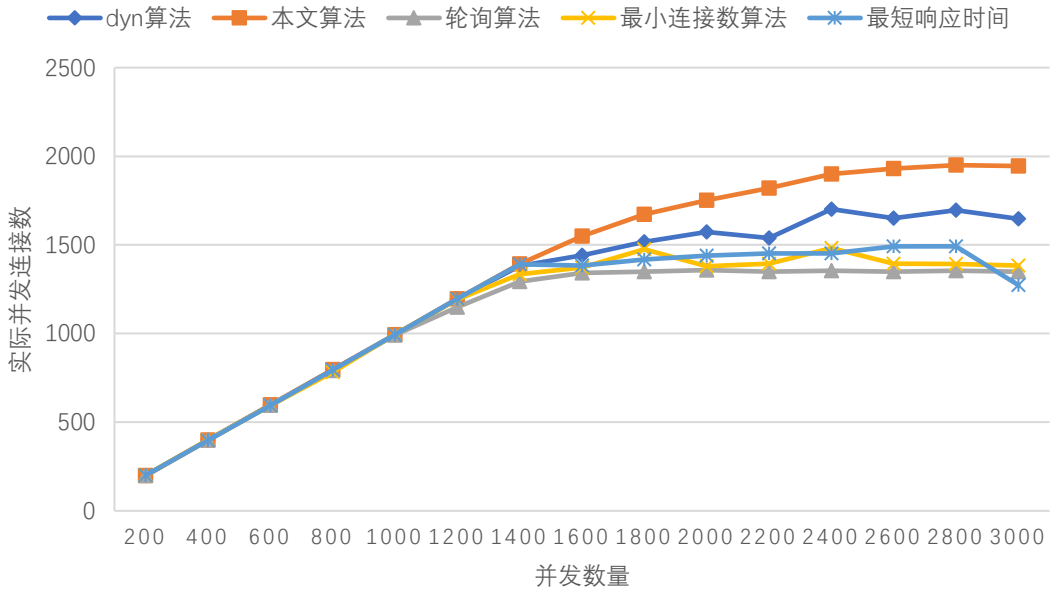

图 5-8 实际并发连接数量比较

由图 5-7 和图 5-8 可以看出，并发数量设置在 1000 以内时各负载均衡算法的平均响应时间差别不大，都能保持在 100 毫秒以内，并且各负载均衡算法的实际并发连接数都能大致等同于并发数量。并发数在 1000 以上时，随着并发数量的增加，用户服务集群的整体负载增大，各个负载均衡算法的平均响应时间都开始增加，且各算法的实际并发连接数开始趋于稳定。通过 Redis 中的数据观察到在并发数量为 1000 时用户服务的 CPU 使用率开始超过阈值，本文的负载均衡算

法进入到动态权重算法模式。在并发数量大于 1000 时，其它算法的实际并发连接数已经稳定不再增长，而本文的算法还有着部分增长，同时相比其它算法有着明显更低的平均响应时间和更高的实际并发连接数量。这也反应了与其它负载均衡算法相比，在高负载情况下，本文的负载均衡算法能够更大程度地利用服务机器的性能，提供更好的服务质量。

表 5-5 不同算法在不同并发程度下的平均响应时间（ms）

	200 并发	400 并发	600 并发	800 并发	1000 并发
本文算法	14	14	19	39	65
轮询算法	14	14	20	40	76
dyn 算法	15	16	24	42	67
最小连接数算法	15	16	26	45	72
最短响应时间	15	16	24	43	75

另外，表 5-5 显示了图 5-7 的并发数量在 1000 以内时的平均响应时间数据，其单位为毫秒。从表中数据可以看出，由于最小连接数算法、最短响应时间算法和 dyn 算法需要维护当前服务对目标服务的连接数、记录请求响应时间并判断最小值、计算权重等操作，算法的请求的响应时间上要大于其它算法。并且由于此时本文的算法还处于第一阶段，算法的平均响应时间与轮询算法基本相同。这也证明了本文的算法不仅在高负载情况下拥有更好的负载均衡效果，还保留了静态算法更短的响应时间的优势。

5.4 本章小结

本章节对本文提出的微服务的主动式扩缩与负载均衡算法进行了部署与测试。首先介绍了本文社区项目的架构与部署环境。然后通过对比 MSE 和 MAE 两个模型评价指标，并对比真实预测结果与实际数据的误差，证明了本文的负载预测模型与其它模型相比有着更高的预测精度，通过监控数据发现，使用本文的主动式扩缩可以大幅度减少服务违反 SLA 的情况。最后分别对本文的和其它主流的客户端负载均衡算法进行压力测试，实验结果表明本文的负载均衡算法能够让服务拥有更低的响应时间和更高的吞吐量。

第六章 总结与展望

6.1 总结

Kubernetes 被广泛应用于部署和管理微服务架构，本文通过观察社区类的项目在 Kubernetes 环境中的运行，发现服务资源的过度配置导致集群资源利用率低，同时 Kubernetes 的反应式扩缩方法也无法很好地满足服务水平协议(SLA)。为了解决这些问题，本文通过搭建负载预测模型，对服务的工作负载进行预测，实现了服务副本的主动式扩缩。另外，由于当前微服务架构多数还在使用客户端式的负载均衡，但是这些算法的效果大都不佳，为了尽可能提高服务的资源利用率，提升服务的质量，本文通过对服务负载数据的权重计算改进了现有的客户端的负载均衡算法。在本文的研究工作中，取得了如下成果：

(1) 基于 Informer 模型实现了微服务主动式的扩缩。本文研究了 Kubernetes 环境下 HPA 这种反应式扩缩的不足之处，为了在 Kubernetes 环境下实现主动式的扩缩，本文研究了当前时间序列预测相关的文献与相关技术，选用 Informer 模型作为服务负载预测的模型，通过对微服务未来一段时间负载的预测实现微服务的主动式扩缩。针对本文的社区项目，本文在主动式扩缩的流程中做了如下改进：选用 CPM 作为服务负载的预测指标，减少了 CPU 和内存使用率指标波动叠加对预测结果的影响；将项目的活动事件数据编码加入预测模型，提高了模型预测的准确程度；优化了自动扩缩算法，借助重试任务队列，提升了主动式扩缩的可靠性。

(2) 为了缓解服务负载预测失败导致服务负载过高、服务质量下降的情况，本文通过改进熵值法实现微服务中客户端的阶段式的动态权重负载均衡算法以增加高负载情况下服务的可用性。为了实现该算法，本文引入权重计算服务和 Redis，计算并传递服务的权重数据。并且本文在原先熵值法的基础上，考虑不同指标的可容忍程度对指标权重系数进行加权，使计算出来的指标权重更适用于负载均衡的场景。

(3) 将上述的微服务主动式扩缩与负载均衡算法部署到本文的社区项目中，通过与其他的时间序列预测模型和其他的客户端负载均衡算法对比，验证了本文工作的有效性。结果表明本文改进后的预测模型在社区类型的项目中预测精度更高，并且监控结果表明本文的主动式扩缩方式有效地缓解了反应式扩缩导致服务质量下降的问题。同时通过压力测试结果表明，本文的负载均衡算法可让服务拥有更短的响应时间，更高的系统吞吐量。最后结果表明上述工作有效地减少服务违反 SLA 的情况，并使服务运行得更稳定。

6.2 展望

本文基于 Informer 模型，对服务负载数据进行预测，实现了微服务的主动式扩缩，同时引入了客户端的动态的负载均衡算法提高了服务资源的利用率。这两部分的内容大部分情况下都表现良好，但是还存在着一些问题需要后续解决和改善。

(1) 预测模型对周期性的负载数据能够很好地预测，但是模型很难预测出周期之外的负载波动，这种情况下，本文的主动扩缩算法无法对此类负载波动做出反应。

(2) 本文的主动式扩缩算法和负载均衡算法都只考虑了 CPU 和内存使用率两个指标，这只适用于项目中都是 CPU 和内存密集型服务的情况，后续可以将网络、IO 等指标纳入算法考量范围，使得算法可以适用于更多场景的服务。

(3) Kubernetes 本身提供了负载均衡的功能，只是大部分的微服务架构都没有使用，后续可将项目中的负载均衡交由 Kubernetes 处理，降低项目微服务架构的复杂程度。

参考文献

[1] alibaba/clusterdata[EB/OL]. [2023-02-20]. https://github.com/alibaba/clusterdata.

[2] 刘伟,石冰心. 服务水平协议(SLA)——Internet 服务业的新趋势[J]. 电信科学,2000,16(11):5-8. DOI:10.3969/j.issn.1000-0801.2000.11.002.

[3] Amazon Auto Scaling[EB/OL]. [2023-02-07]. https://aws.amazon.com/cn/autoscaling/.

[4] Google Cloud Autoscaler[EB/OL]. [2023-02-07]. https://cloud.google.com/compute/docs/autoscaler.

[5] Liu Z, Zhu Z, Gao J, et al. Forecast methods for time series data: a survey[J]. IEEE Access, 2021, 9: 91896-91912.

[6] Dama F, Sinoquet C. Time Series Analysis and Modeling to Forecast: a Survey[J]. arXiv preprint arXiv:2104.00164, 2021.

[7] Klinaku F, Frank M, Becker S. CAUS: an elasticity controller for a containerized microservice[C]//Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. 2018: 93-98.

[8] Taherizadeh S, Stankovski V. Dynamic multi-level auto-scaling rules for containerized applications[J]. The Computer Journal, 2019, 62(2): 174-197.

[9] Zhang F, Tang X, Li X, et al. Quantifying cloud elasticity with container-based autoscaling[J]. Future Generation Computer Systems, 2019, 98: 672-681.

[10] Kan C. DoCloud: An elastic cloud platform for Web applications based on Docker[C]//2016 18th international conference on advanced communication technology (ICACT). IEEE, 2016: 478-483.

[11] Ye T, Guangtao X, Shiyou Q, et al. An auto-scaling framework for containerized elastic applications[C]//2017 3rd international conference on big data computing and communications (BIGCOM). IEEE, 2017: 422-430.

[12] Ciptaningtyas H T, Santoso B J, Razi M F. Resource elasticity controller for Docker-based web applications[C]//2017 11th International Conference on Information & Communication Technology and System (ICTS). IEEE, 2017: 193-196.

[13] Imdoukh M, Ahmad I, Alfailakawi M G. Machine learning-based auto-scaling for containerized applications[J]. Neural Computing and Applications, 2020, 32: 9745-9760.

[14] Dang-Quang N M, Yoo M. Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes[J]. Applied Sciences, 2021, 11(9): 3835.

[15] Dang-Quang N M, Yoo M. Multivariate deep learning model for workload prediction in cloud computing[C]//2021 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, 2021: 858-862.

[16] 郭杨虎. 微服务环境下 docker 容器调度策略的研究与实现[D]. 北京邮电大学, 2018.

[17] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[18] Zhou H, Zhang S, Peng J, et al. Informer: Beyond efficient transformer for long sequence time-series forecasting[C]//Proceedings of the AAAI conference on artificial intelligence. 2021, 35(12): 11106-11115.

[19] Rabiu S, Yong C H, Mohamad S M S. A Cloud-Based Container Microservices: A Review on Load-Balancing and Auto-Scaling Issues[J]. International Journal of Data Science, 2022, 3(2): 80-92.

[20] 聂世强,伍卫国,张兴军等.一种基于跳跃 hash 的对象分布算法[J].软件学报,2017,28(08):1929-1939.DOI:10.13328/j.cnki.jos.005200.

[21] 汪佳文,王书培,徐立波等.基于权重轮询负载均衡算法的优化[J].计算机系统应用,2018,27(04):138-144.DOI:10.15888/j.cnki.csa.006284.

[22] 王诚,李奇源.基于贪心算法的一致性哈希负载均衡优化[J].南京邮电大学学报(自然科学版),2018,38(03):89-97.DOI:10.14132/j.cnki.1673-5439.2018.03.014.

[23] 张开琦. 微服务架构负载均衡及服务容错研究[D].昆明理工大学,2021.DOI:10.27200/d.cnki.gkmlu.2021.000777.

[24] 吴俊鹏,刘晓东.一种基于集群的动态负载均衡算法研究[J].电子设计工程,2021,29(16):75-78.DOI:10.14022/j.issn1674-6236.2021.16.016.

[25] 葛钰,李洪赭,李赛飞.一种 web 服务器集群自适应动态负载均衡设计与实现[J].计算机与数字工程,2020,48(12):3002-3007.

[26] 刘瑞奇,李博扬,高玉金等.新型分布式计算系统中的异构任务调度框架[J].软件学报,2022,33(03):1005-1017.DOI:10.13328/j.cnki.jos.006451.

[27] Wang H, Wang Y, Liang G, et al. Research on load balancing technology for microservice architecture[C]//MATEC Web of Conferences. EDP Sciences, 2021, 336: 08002.

[28] Yi C, Zhang X, Cao W. Dynamic weight based load balancing for microservice cluster[C]//Proceedings of the 2nd International Conference on Computer Science and Application Engineering. 2018: 1-7.

[29] Vahab Mirrokni,Mikkel Thorup,Morteza Zadimoghaddam. Consistent hashingwith bounded loads[P]. Discrete Algorithms,2018.

[30] Li J, Yi G, Wu B, et al. Research and Improvement of Dynamic Highly Available Load Balancing Algorithm for Microservices[C]//2022 International Conference on Artificial Intelligence and Computer Information Technology (AICIT). IEEE, 2022: 1-6.

[31] Kubernetes Overview [EB/OL]. [2023-02-19]. https://kubernetes.io/docs/concepts/overview/.

[32] Swarm mode overview[EB/OL]. [2023-02-20]. https://docs.docker.com/engine/swarm/.

[33] Kubernetes Horizontal Pod Autoscaler[EB/OL]. [2023-02-20]. https://kubernetes.io/docs/tasks/run-application/horizontal-podautoscale/.

[34] Khan S, Alghulaiakh H. ARIMA model for accurate time series stocks forecasting[J]. International Journal of Advanced Computer Science and Applications, 2020, 11(7).

[35] Van Houdt G, Mosquera C, Nápoles G. A review on the long short-term memory model[J]. Artificial Intelligence Review, 2020, 53: 5929-595.

[36] 任欢, 王旭光. 注意力机制综述[J]. 计算机应用, 2021, 41(S01): 1-6.

[37] Luo S, Xu H, Ye K, et al. The power of prediction: microservice auto scaling via workload learning[C]//Proceedings of the 13th Symposium on Cloud Computing. 2022: 355-369.

[38] Makai J. New solutions in it monitoring: cadvisor and collectd[R]. 2015.

[39] Grinberg M. Flask web development: developing web applications with python[M]. " O'Reilly Media, Inc.", 2018.

[40] Redis Documentation[EB/OL]. [2023-02-20]. https://redis.io/docs/.

[41] 裘炜毅,闫博.基于熵值法和 TOPSIS 法高铁新城发展水平评价的指标体系构建方法[J].交通与运输,2023,39(01):32-38.

[42] 张娜,马琳.基于 Nginx 负载均衡的动态分配技术研究[J].齐齐哈尔大学学报(自然科学版),2019,35(01):27-30.