# 密级： 保密期限：



**硕士学位论文**



**题目： 基于Docker的视频监控云平台**

**资源预测与配置研究**

**学 号： 2014140420**

**姓 名： 姜 哲**

**专 业： 计算机技术**

**导 师： 刘 亮**

**学 院： 计算机学院**

**2017 年 3 月 9 日**

独创性（或创新性）声明

本人声明所呈交的论文是本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京邮电大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

申请学位论文与资料若有不实之处，本人承担一切相关责任。

本人签名： 日期：

关于论文使用授权的说明

本人完全了解并同意北京邮电大学有关保留、使用学位论文的规定，即：北京邮电大学拥有以下关于学位论文的无偿使用权，具体包括：学校有权保留并向国家有关部门或机构送交学位论文，有权允许学位论文被查阅和借阅；学校可以公布学位论文的全部或部分内容，有权允许采用影印、缩印或其它复制手段保存、汇编学位论文，将学位论文的全部或部分内容编入有关数据库进行检索。（保密的学位论文在解密后遵守此规定）

本人签名： 日期：

导师签名： 日期：

基于Docker的视频监控云平台资源预测与配置研究

摘 要

随着视频监控设备的大规模部署，传统的视频监控系统已无法对海量的视频监控数据进行有效的计算分析，伴随云计算的发展，视频监控系统与云计算融合所形成的视频监控云成了近年来的研究热点。当前主流的视频监控云平台基本都是以虚拟机的形式为用户提供服务，资源分配粒度大且存在性能损耗。目前云平台中对资源配置基本是在初始阶段采取静态分配方式，然后在运行期再根据监控预警或负载预测的方法进行动态水平伸缩。然而监控预警的方式没有考虑资源需求的实时性，是一种被动伸缩方式，容易违反服务等级协议SLA；负载预测方式虽然具有预动性，但目前仍没有一种预测方法能够对视频监控云平台中的服务负载做出准确的预测。

如何提高视频监控云平台的资源利用率，实现一个高效的可以动态弹性伸缩的视频监控云平台是本文的研究重点。首先，本文通过对典型视频监控服务的负载特性进行分析，提出了适用于视频监控云平台的资源预测模型，该模型的构建可分为两个阶段，在第一阶段可以根据视频服务的属性特征来预测其初始资源需求量，第二阶段基于资源需求时间序列相似性对工作负载进行预测。其次，针对视频监控云平台中无法对资源及时准确的进行动态调整这一问题，本文对现有基于容器技术的视频监控云平台进行了优化，在云平台中添加了资源预测模块并调整了资源配置策略，使之可以根据预测结果通过热迁移及垂直伸缩的方式对容器的资源进行重配置，从而提高资源利用率。最后，本文实现了基于Docker的视频监控云计算平台的优化工作，对所提出的资源预测模型及调整后的资源配置策略进行了性能测试，实验结果表明本文所提出的资源预测模型有更高的准确率，所调整资源配置策略可以有效提高视频监控云平台的资源利用率。

关键字：云计算，Docker，视频监控云，资源预测，资源配置

RESEARCH ON RESOURCE PREDICTION AND ALLOCATION OF VIDEO SURVEILLANCE CLOUD PLATFORM BASED ON DOCKER

ABSTRACT

Nowadays, with the large-scale deployment of video surveillance equipment, the traditional video surveillance system is unable to deal with the massive amounts of video surveillance data for effective calculation and analysis. In recent years, cloud computing technology is developing, and the related researches focus on the video surveillance system which is integrated with the cloud computing technology. Basically, the current mainstream video surveillance cloud platform provides services for user in the form of virtual machine, which results in the large resource fragmentation and the performance loss. The cloud platform allocates the resources by performing a static allocation in the initial stage, and then in the running period, it adaptively rebalances resources according to monitoring alarm or load predication strategy. However, the way of monitoring alarm does not consider the instantaneity of the resource requirements, and it adopts a passive telescopic way that will violate the SLA easily. Although the current load prediction method is proactive, it is unsuitable for video surveillance cloud platform to make the accurate prediction about the load of video services.

This paper focuses on how to improve the resource utilization of the video surveillance cloud platform and realize an efficient and flexible scalable video surveillance cloud platform. Firstly, this paper analyzes the load characteristics of the video surveillance services, and proposes a resource prediction model for video surveillance cloud platform. The model construction process can be divided into two phases. In the first phase, we predict the initial resource demand based on the performance of video services. In the second phase, we predict the video workload based on the resource demand time series similarity analysis. Moreover, as for lackness in re-allocating resources adaptively and accurately in video surveillance cloud platform, this paper gives a strategy to optimize the existed video surveillance cloud platform by adding a resource prediction module and adjusting the resource allocation strategy to improve the resource utilization rate. At the end of this paper, we realize the optimization strategy of the video surveillance cloud computing platform based on Docker, and conduct the extensive performance experiments. The experimental results show that the resource prediction model proposed in this paper has higher accuracy and the adjusted resource allocation strategy can effectively improve the resource utilization of the video surveillance cloud platform.

**KEY WORDS**: cloud computing, Docker, video surveillance cloud, resource prediction, resource allocation

目 录

[第一章 绪论 1](#_Toc477781095)

[1.1 研究背景和意义 1](#_Toc477781096)

[1.2 国内外研究现状 2](#_Toc477781097)

[1.3 论文的主要研究内容 4](#_Toc477781098)

[1.4 论文的组织结构 5](#_Toc477781099)

[第二章 相关技术介绍 7](#_Toc477781100)

[2.1 Docker相关技术 7](#_Toc477781101)

[2.1.1 Docker实现原理 8](#_Toc477781102)

[2.1.2 Docker与虚拟机对比 10](#_Toc477781103)

[2.1.3 容器迁移技术 11](#_Toc477781104)

[2.2 微服务 14](#_Toc477781105)

[2.2.1 单体式架构 14](#_Toc477781106)

[2.2.2 微服务架构 15](#_Toc477781107)

[2.3 云平台资源预测模型 17](#_Toc477781108)

[2.3.1 时间序列预测模型 17](#_Toc477781109)

[2.3.2 神经网络模型 19](#_Toc477781110)

[2.4 云平台资源调度策略 20](#_Toc477781111)

[2.4.1 Openstack平台中的资源调度策略 20](#_Toc477781112)

[2.4.2 Docker集群中的资源调度策略 21](#_Toc477781113)

[2.5 本章小结 22](#_Toc477781114)

[第三章 监控视频云计算平台资源预测方法 23](#_Toc477781115)

[3.1 负载及云资源特性分析 23](#_Toc477781116)

[3.1.1 负载特性 23](#_Toc477781117)

[3.1.2 资源特性 24](#_Toc477781118)

[3.2 云资源预测方法 25](#_Toc477781119)

[3.2.1 问题描述 25](#_Toc477781120)

[3.2.2 算法描述 25](#_Toc477781121)

[3.3 本章小结 28](#_Toc477781122)

[第四章 监控视频云计算平台资源配置与优化 31](#_Toc477781123)

[4.1 监控视频云计算平台介绍 31](#_Toc477781124)

[4.2 监控视频云计算平台的资源配置方法 34](#_Toc477781125)

[4.3 监控视频云计算平台资源配置优化 34](#_Toc477781126)

[4.3.1 工作流程 35](#_Toc477781127)

[4.3.2 资源监控模块 36](#_Toc477781128)

[4.3.3 资源预测模块 37](#_Toc477781129)

[4.3.4 云资源管理模块 38](#_Toc477781130)

[4.4 基于预测的云资源配置方法 41](#_Toc477781131)

[4.4.1 问题描述 41](#_Toc477781132)

[4.4.2 算法描述 41](#_Toc477781133)

[4.5 本章小结 44](#_Toc477781134)

[第五章 系统实现与测试分析 45](#_Toc477781135)

[5.1 系统硬件配置和软件配置 45](#_Toc477781136)

[5.1.1 系统硬件配置 45](#_Toc477781137)

[5.1.2 软件配置 45](#_Toc477781138)

[5.2 系统功能实现 46](#_Toc477781139)

[5.2.1 资源监控模块实现 46](#_Toc477781140)

[5.2.2 资源预测及管理模块实现 48](#_Toc477781141)

[5.2.3 典型视频微服务实现 51](#_Toc477781142)

[5.2.4 页面实现 52](#_Toc477781143)

[5.3 资源预测与配置方法性能测试 55](#_Toc477781144)

[5.3.1 资源预测方法性能测试 55](#_Toc477781145)

[5.3.2 资源配置方法性能测试 57](#_Toc477781146)

[5.4 本章小结 58](#_Toc477781147)

[第六章 总结与展望 59](#_Toc477781148)

[6.1 总结 59](#_Toc477781149)

[6.2 展望 60](#_Toc477781150)

[参考文献 61](#_Toc477781151)

[致谢 65](#_Toc477781152)

[攻读硕士学位期间的主要研究成果 67](#_Toc477781153)

# 第一章 绪论

## 1.1 研究背景和意义

现今社会，视频监控在我们的生活中已变的不再陌生，各式各样的监控设备充斥在道路、学校、小区等场所。视频监控系统越来越多地用于提供智能视频分析和基于事件的自动实时警报，如入侵检测，车辆检索和视频浓缩等，以加强城市管理[1][2]。但是随着监控设施的增多，监控规模的扩大，传统的视频监控方式针对每天产生的海量视频数据无法提供给充足的计算资源进行有效的智能分析，已不能满足人们的服务需求。并且这种传统的集中式的视频数据处理方式过多的依靠单服务器的处理能力，出现专机专用的现象，资源之间独立无法共享的模式也是传统计算模式的一大诟病。近年来，随着云计算的出现，智能视频分析技术与云计算的结合被广泛接受为解决大规模视频监控分析的有效解决方案[3][4][5]，视频监控云也迅速成为近年来研究的热点。云计算技术可以将底层分散异构的计算资源整合起来资源池化，统一进行管理、按需分配。然而面对视频服务资源需求的动态性及异构性等特点，如何实现一个高效的可以动态弹性扩展的视频监控系统云平台是很有挑战性的难题。

资源管理是所有云平台需要解决的关键问题[6]，视频监控云平台亦是如此。当前主流的视频监控云平台都是将智能视频应用程序及所需要的资源封装成虚拟机的形式来对外提供服务[5][7][8][9]，然而这种静态分配资源的方式存在一些问题。在视频监控云平台中，每个视频应用程序可能具有彼此不同的资源需求，此外，每个视频微服务的负载都是动态变化的，如果按照负载的峰值为其分配资源，那么在别的非峰值期间就会造成资源浪费；如果为视频微服务分配的资源不足，就有可能违反应用服务等级协议（SLA）。因此，我们可以实时地根据服务资源需求进行动态调整资源分配量，在资源需求增加时，为其提升资源配置，当资源需求降低时，回收其部分资源，这样就可以提高资源利用率。但是相对于回收资源，增加资源具有更高的要求，必须具有预动性，因为分配的资源需要一定的时延方可使用，如果当服务需求变大时再去扩展就很有可能违反SLA。因此，为了解决在这种复杂云环境下虚拟化资源分配不当可能导致资源浪费和应用程序性能下降的问题，这就需要对视频微服务的资源需求进行准确的预测，为资源的动态调整做好准备。根据预测结果及时的动态调整视频微服务的资源配置，能够为云计算资源的合理调度和云服务的稳定运行提供决策支持，可以在保证服务质量的同时，提高资源利用率。因此，寻找适合视频微服务负载模式的预测模型以及基于预测模型的资源配置研究对于视频监控云平台而言具有很重要的意义。

此外，更细粒度的资源管理可以提高资源利用率，为了更充分利用服务器的资源降低成本，我们可以考虑更轻量级、更细粒度的容器技术[10]。容器技术比虚拟机更有效率，最近的文献[11][12][13]表明Docker[14][15]容器与基于KVM的虚拟机相比只消耗了6%的CPU和16%的内存，并且启动时间也只是虚拟机的60%，重启时间也只有虚拟机的2%，所以容器技术比虚拟机更有效率，因此使用这种新的容器技术构建视频云监控平台是很有必要的。

本文针对视频监控云计算平台中的资源管理方面的问题，提出了适用于视频微服务的资源预测模型，并将预测模型与资源管理相结合，从而优化了视频监控云平台中资源配置策略，解决了构建一个高效的可以动态弹性扩展的视频监控系统云平台所遇到的难题。

## 1.2 国内外研究现状

自2006年亚马逊（Amazon）推出弹性计算云（EC2）服务[17]开始，云计算技术以其独特的优势得到大家的青睐迅速得到推广。随着云计算生态链的日益完善，多种技术融合的多元环境也已形成。相对于虚拟机的资源消耗大、启动时间长等缺点，容器技术以其细粒度的资源分配、提供开始构建、发布、运行分布式应用的方式迅速成为云计算领域的新贵，已逐渐成为主流的云计算技术之一，而Docker对容器技术的普及和发展具有不可磨灭的推动作用。从DockerCon 2016大会[18]上的数据可以看到，越来越多的使用场景和使用方式出现，例如Docker推出的面向企业的Container as a Service (CaaS)以及多平台（Linux、Windows、Solaris等）的支持；越来越多的公司参与到Docker的开发、使用中，如Amazon（AWS ECS）、Google（GCE）、Microsft（Azure Container Service）等，可以看出Docker已经成为了众多厂商交付软件的新选择[19]。今年双11最大的变化是支撑双11的所有交易核心应用都跑在了Docker容器中，几十万Docker容器撑起了双11交易17.5万笔每秒的下单峰值[20]。不仅是阿里，百度也使用Docker技术全部重构了BAE平台，另外腾讯、京东、华为、美团等公司也都运行着大规模的容器集群来为公司运营提供支撑。

除了目前流行的Docker容器技术，以LibOS为基础的更加高效的 Unikernel[21]技术同样引起大家的关注，Unikernel 将应用及其依赖的运行时环境全部运行在“内核态”。Unikernel也有其劣势，由于其为上层应用的定制化程度较高，因此难以像传统操作系统一样用于通用的应用目的。目前 Unikernel 仍处在研究阶段，主要的项目包括 ClickOS、Clive、 MirageOS 等，商业化应用为时尚早[22]。可见在视频监控云平台领域采用轻量级的容器技术是个发展趋势，本文就是基于Docker技术对视频监控云平台进行优化实现。



图1-1 从虚拟机到Unikernel的演化[22]

近些年，国内外在云平台中负载预测技术研究方面也逐渐热了起来，Liu等人采用云模型来研究云环境中负载预测的问题，提出一种基于改进云模型的负载预测方法PMCM，通过提出亲密云和重叠云来进行概念跃升，挖掘CPU与内存之间的关联，形成规则发生器，由此建立预测模型[23]。Jia等人[24]采用云模型相似度的度量方法与反向传播（BP）网络结合的方式来进行预测，云模型是通过转换定性和定量表达式而获得的一种模型，其表示了语言概念中包含的模糊性和随机性，反向传播网络验证了所提出方法的有效性。Y.Wu等[25]提出了应对网格计算环境的自适应混合资源预测方法，通过将静态AR模型扩展到动态窗口，并且通过自适应机制提高预测精度。 S.MALLICK等[26]通过监控云平台中的虚拟机，按照CPU负载划分状态构成，建立在状态之间转移的Markov模型，通过该模型预测虚拟机在未来一段时间的负载。Xiao等人[27]提出一种基于指数加权移动平均（EWMA）预测CPU负载的预测方法。上述资源预测的共同之处在于它们仅用于CPU负载预测。Y.Tan等[28]认为实际应用的多重特性（如CPU负载，内存闲置水平，磁盘使用率，网络流量等）共同决定了系统的不规则状态，他们建立了基于马尔可夫链的包含多个属性的资源预测模型，以预测物理服务器资源的使用。Calheiros等人[29]提出了基于差分自回归移动平均（ARIMA）模型来预测云负载。随着神经网络和机器学习技术的流行，Qiu[30]等提出了一种基于深度学习的虚拟机工作负载预测的新方法。深度学习预测模型包含一个由多层限制玻尔兹曼机（RBM）组成的深度信念网络（DBN）和一个回归层。DBN用于从所有VM工作负载数据提取高级功能，回归层用于预测未来VM的工作负载。Kousiouris 等人[31]提出一种基于通过遗传算法优化的人工神经网络（ANN）的应用工作负荷预测方法。 Islam 等人[32]应用ANN和线性回归预测应用所需的资源。Shen[33]提出了基于支持向量机（SVM）的虚拟资源调度预测算法来处理复杂、动态、变化环境的云平台。通过重建相位空间重建虚拟资源序列将重建序列用作SVM中的输入以用于训练和预测。现有的资源预测模型有很多，但是基本都是对整个云平台的工作负载或网络流量进行预测，还没有适用于容器中视频微服务的负载预测模型。本文第三章提出的基于服务相似性和最近邻回归的预测模型充分考虑了视频微服务的负载特性，进一步提高了针对视频服务负载的预测性能。

传统的固额分配资源的模式已很难适应云平台的发展，围绕如何有效的调整资源配置，提高资源利用率这一问题，国内外也展开了相关的研究。Hossain [9]提出了一种基于云端视频监控平台的服务组合的线性编程方法的动态资源分配方法。Yang等人[5]，提出了一种用于视频监控云的多资源虚拟机分配算法，以主导资源优先的调度策略提高了集群的资源利用率。文献[35]在资源分配方式中引入了控制机制，其动态分配CPU资源的方法是通过将卡尔曼过滤技术融入到反馈控制器中实现的。目前在云环境中采用预测技术对资源进行动态分配是主流方法。Li等人[36]，通过回归分析和KMP算法进行负载预测，然后根据预测结果进行动态的调整。Zhang等人[37]在视频监控云平台中通过时间序列相似性对视频服务负载进行预测，然后根据预测结果对容器的资源配置进行动态调整。Lee等[38]提出了一种两级资源分配方法：首先通过数学模型估算任务的资源需求量，将最大资源量分配给任务，然后根据HMM的预测结果动态的调整资源。本文在第四章给出了对视频监控云平台的资源配置优化方案，根据第三章所提的资源预测模型，结合容器热迁移等技术对视频微服务的资源配置进行垂直伸缩，从而提高整个云平台的资源利用率。

## 1.3 论文的主要研究内容

视频监控云平台的主要功能是为上层的视频应用提供弹性的云计算能力，在为用户提供视频处理服务时既可以保证服务质量避免违背服务级别协议（SLA），又可以提高资源利用率。为了解决平台的高可伸缩性，本文通过基于云资源预测技术的细粒度资源重配置研究，优化并实现了基于Docker的视频监控云平台。主要的研究工作包括以下几个部分：

（1）面向视频监控应用的云平台资源预测模型。当前主流基于虚拟机的视频监控云平台在为用户提供服务时都是生成一个固定配额的虚拟机在上面部署相应的应用程序，在运行的过程中一般不会再去动态的调整资源的大小因此会导致当前的资源不能很好的适应负载的变化。通过分析应用负载的特性，考虑视频监控服务的资源需求的动态性和随机性，提出了一种基于视频服务特征相似性和时间序列最近邻回归TSNNR相结合的云资源预测模型。该模型可以准确的预测在未来一段时间视频监控服务所需要资源数量，为动态伸缩视频微服务的云资源供应提供依据，这对于云资源及时分配和回收具有重要的指导意义。通过实验验证，该模型具有较高的准确率。

（2）基于资源预测模型的细粒度资源配置。当前的云平台在资源动态配置方面要么是采用监控预警的方式进行伸缩要么根据流量预测结果进行伸缩，所采用的伸缩方式基本都是水平伸缩即通过增加或减少虚拟机的数量来满足用户的服务请求。这对无状态的应用来说是很有效的，但是对于有状态的服务来说是很难的。我们通过获得的资源预测结果，根据优化的调度算法动态迁移容器到最优的目的节点，然后进行垂直伸缩，以最大化提高资源利用率。

（3）基于Docker技术的视频监控云平台的优化与实现。本文在现有的视频监控云平台的基础上增加了资源预测模块，并将资源配置模块进行了优化，使得容器的资源可以根据预测结果进行动态的伸缩。并基于微服务架构实现了依托于云平台的视频监控系统，验证了云平台的可用性，进一步提高了资源的利用率。

## 1.4 论文的组织结构

本文共分为六个章节，各章节的内容组织如下：

第一章：绪论。本章首先介绍了视频监控云平台的研究背景和意义以及需要解决的一些问题；其次，分析了现阶段国内外对云计算方面的一些发展现状及国内外学者对云资源预测和资源配置方面的研究方法进行总结；最后，简述了本课题的主要研究内容和相关工作。

第二章：相关技术介绍。本章首先介绍了构建及优化云平台方面的一些背景知识，包括Docker及迁移技术等知识；其次对当前云平台中所常用到的资源预测模型以及资源配置过程中所用到的调度算法进行了详细介绍。本章为后续的研究和平台的开发奠定了理论基础。

第三章：监控视频云计算平台资源预测方法。首先分析了视频监控服务的负载特性以及云平台中底层物理资源和服务请求资源的特性。然后通过分析国内外相关预测技术在视频监控云平台中应用的不足之处，提出了适用于视频服务的资源预测模型，包括服务相似性匹配和时间序列最近邻回归（TSNNR）算法。该算法能够准确的预测出服务在未来一段时间的资源需求，为下一阶段的资源配置提供依据。

第四章：监控视频云平台资源配置与优化。本章开始部分总结了当前视频监控云平台的不足之处，然后介绍了加入预测模块及优化了资源配置的云平台架构及工作流程。接着分别介绍了几个重点模块的设计，后面对视频监控云平台的资源重配置给出了解决方案。经过优化的云平台具有高可用性、高可伸缩性等特点。

第五章：系统实现与测试分析。本章从实现的角度详细介绍了各部分功能的具体实现过程，展示了所实现系统的功能界面，并对所提出的预测算法进行对比实验，通过实验证明了所提算法的优势。最后对优化的资源配置方案进行相应的性能测试，验证了资源伸缩可以有效提高云平台的资源利用率。

第六章：总结与展望。对本文所做的工作进行回顾总结，并对今后进一步的工作进行展望。

# 第二章 相关技术介绍

本章介绍了与后面章节相关的一些技术知识。包括与本文所优化的视频监控云平台相关的云环境知识，以及当前云平台中比较流行的资源预测模型及常用云资源管理中的调度算法。

## 2.1 Docker相关技术

Docker是2013年PaaS服务提供商dotCloud公司的一个开源项目，仅仅三年的时间就成为时下最流行的技术，为云计算注入了新的活力。随着Docker技术的不断发展，Docker生态圈的不断完善，Docker不再只是一个公司的名字，而是渐渐变成轻量级虚拟化的代名词。官方给出的定义：Docker是以Docker容器为资源分割和调度的单位，封装整个软件运行时环境，为开发者和系统管理员设计的，用于构建、发布和运行分布式应用的平台[16]。

Docker的底层是基于LXC（Linux Container）技术（主要是Namespace和CGroup）来实现容器的资源管理等功能，它是一种操作系统级虚拟化技术。Docker的主要目标是“Build，Ship and Run any app，anywhere”[39]，即通过对相应应用生命周期的管理，它能够让应用的分发、部署和管理都变得前所未有的高效和轻松。Docker及其生态系统主要带来了以下几个方面的好处。

（1）自动化测试和持续集成。Docker可以将应用封装在一个完整的文件系统中，该文件包含应用运行所需的所有内容，消除了开发、部署等之间的环境差异。开发人员只需要关注应用功能的开发，而不需要考虑环境依赖问题，同时减轻了测试和运维人员的工作压力，简化了开发、测试、部署的过程。

（2）跨平台的可移植性。Docker具有良好的兼容性，几乎可以运行在任意平台之上，而不局限于某一固定的云平台，这种兼容性可以使得应用部署在不同的云平台上并可以在多平台间轻松迁移。当前，支持Docker的云平台也是逐渐增多，包括Openstack、微软云平台Azure、亚马逊的AWS及Google的GCP等。

（3）高效的资源利用及隔离。Docker是操作系统级的虚拟化，相对于全虚拟化的虚拟机而言，没有Hypervisor等额外的开销，与底层共享内核，因此具有更高的性能同时对资源的额外消耗更低。Docker使用内核进行资源的隔离和控制，可以更直接的、更细粒度的分配和管理内存、CPU等资源，并保证容器之间互不影响。

（4）组件可复用。Docker采用AUFS联合文件系统使得镜像变得可分层、可编程，镜像之间不再是相互隔离的，不同的镜像可以组合成一个新的镜像。每个镜像都带有一个唯一的标签，通过标签可以实现镜像的回溯，加载指定的镜像。镜像的制作和修改过程也变得很简单，只需编辑Dockerfile文件重新创建即可。

### 2.1.1 Docker实现原理

Docker底层所采用的关键技术主要包括Linux内核提供的Namespace（命名空间）和Cgroup（控制组）以及Union File Systems（联合文件系统）等，本节将简单介绍一下这些相关的基本概念。

（1）Namespace

Namespace（命名空间）提供了虚拟化的轻量级方式，通过它可以实现进程之间的隔离。其原理是通过将内核中的各项全局资源进行抽象并做封装，这样可以使每个Namespace都有自己独立的资源，在同一Namespace中的进程是可以看到彼此的，而对其它的进程是不可见的，因此不同Namespace中的进程对资源的使用是互不影响的，从而可以实现资源的隔离。

图2-1所示是Namespace之间怎么对进程隔离。可以看出所有的Namespace都运行在同一系统内核上，每个命名空间中都有两个进程，命名空间1中的进程1-1可以看到同一空间中的进程1-2，但是别的空间中的进程如进程2-1等对其是透明的。每个Namespace中的进程都认为它们独占整个系统。



图2-1 命名空间实现进程隔离

不同的命名空间封装了不同的资源，在Linux内核中目前共实现了6种Namespace，它们基本上满足了容器所需要做的隔离。表2-1给出了这6种隔离项。

表2-1 namespace的6种隔离项

|  |  |  |
| --- | --- | --- |
| namespace | 系统调用参数 | 隔离内容 |
| UTS | CLONE\_NEWUTS | 主机名与域名 |
| IPC | CLONE\_NEWIPC | 信号量、消息队列和共享内存 |
| PID | CLONE\_NEWPID | 进程ID |
| Network | CLONE\_NEWNET | 网络设备、网络栈、端口 |
| Mount | CLONE\_NEWNS | 文件系统挂载点 |
| User | CLONE\_NEWUSER | 用户ID和用户组ID |

（2）Cgroup

Cgroup（control group）也称作控制组，是容器底层的另一个关键工具，它是Linux内核所提供的可以统计、限制和隔离一组进程所使用系统资源（包括CPU、内存、I/O等）的一种机制。Cgroup为系统资源管理提供了一个统一的框架，可以依据不同的条件，来对进程进行任意的分组。本质上来说，Cgroup是一组由内核附加到程序上的钩子（hooks），这些钩子可以在程序的执行过程中通过调度相应资源来触发，从而实现对资源的控制[16]。Cgroup的目标是实现一个标准化的接口以便对不同层次的资源进行管理。目前，Cgroup适用的场景逐渐增多，从单一任务的资源控制到操作系统层次的虚拟化实现。其主要提供了以下几个方面的功能特性：

1）资源限制：Cgroup可以限制应用运行过程中可使用的资源总量。例如，可以为应用设置一个内存使用上限，如果应用在运行过程中内存的使用量超过了其资源配额，那么就会出现内存溢出异常。

2）资源审计：可以记录系统的资源使用量，如CPU使用时间、内存的使用量等，可用于实现计费功能。

3）优先级控制：可以为某个应用分配特定的CPU时间片及网络I/O的大小，以实现对应用优先级的控制。

4）进程组控制：可以实现进程的挂起及恢复等操作。

（3）UnionFS

UnionFS（联合文件系统）是一种轻量级的分层文件系统，支持将文件系统中的修改信息作为提交，并层层叠加，同时可以将不同的目录挂载到同一虚拟文件系统下[40]。

UnionFS是实现Docker镜像的基础，使得镜像可以进行分层继承。如用户可以通过在基础镜像上添加修改来制作新的镜像，这些镜像都拥有同一个底层的基础镜像，这样可以节省磁盘空间。修改镜像实际上是在原有镜像上添加了一个新的层，而不需要替换整个镜像。用户下发镜像时也只需下发所变动的新层部分，这使得对镜像的管理变得很轻量级和便捷。AUFS就是Docker中所使用的一种UnionFS。它可以为每个成员目录设置只读、读写、写出这三种权限，同时AUFS里也有类似分层的概念。

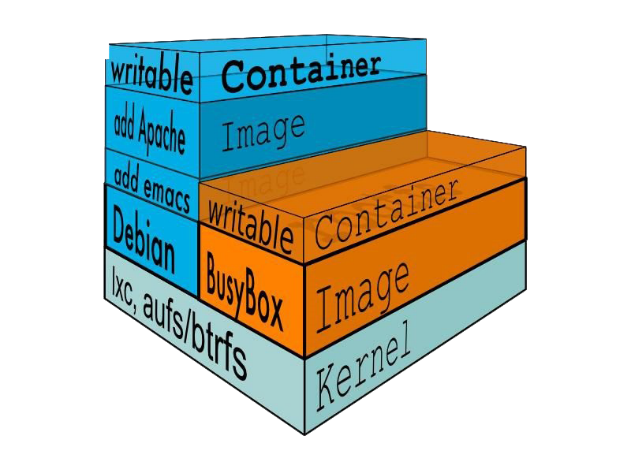


图2-2 联合文件系统[40]

当Docker启动容器时，会根据镜像来分配文件系统，因为镜像是只读的不保存用户状态，因此会给容器挂载一个新的可读写的层，所有的修改都写入最上层的writeable层中，从而在这个文件系统中创建容器，并将挂载的可读写的层覆盖在镜像上，如图2-2所示。

### 2.1.2 Docker与虚拟机对比

前一小节中对Docker容器技术进行了简单的介绍，在本节中对Docker和传统的虚拟机进行比较，关于两者架构见图2-3所示，左图为虚拟机的架构图，右图是Docker的架构图。

传统的虚拟机技术通过Hypervisor即虚拟机监视器（Virtual Machine Monitor，VMM）将计算机中的各种物理资源（服务器、内存、网络、存储等）进行抽象转换，然后供上层虚拟机使用。每个虚拟机都包含完整的硬件虚拟层、客户机操作系统（Guest OS）、公共库（Bins/Libs）及运行在其上的应用程序等组件。其本身的操作系统就会占用一定的资源，并且与硬件之间的交互都是通过Hypervisor代理，因此会带来一定的资源损耗及性能损失。

而Docker容器是基于操作系统级的虚拟化技术，用户所看到的容器只是宿主机操作系统（Host OS）上的一个进程，利用底层LXC的Namespace来进行资源隔离，Cgroup来做资源限制，并通过联合文件系统AUFS来提高系统的资源利用率。因为Docker容器是共享宿主机的内核，没有Hypervisor层中间开销也不需要生成自己的操作系统，通过Docker Engine实现系统调用等功能，相比虚拟机具有轻量级的优点，因此资源利用率及系统的性能都较高。



图2-3 虚拟机与Docker架构对比图[39]

此外，Docker容器相对虚拟机还有另外一个优势是层级镜像，即镜像可分层叠加，容器可以在只读镜像的基础上添加新的内容制作新的镜像，新的镜像又可以作为基础镜像被上层的镜像使用。这种特性可以大大节省磁盘空间从而提高资源利用率，并且多个容器可以使用统一镜像，这样也可以减少内存的占用量。本文对容器与虚拟机的对比总结见表2-2。

表2-2 虚拟机与Docker容器比较

|  |  |  |
| --- | --- | --- |
| 比较项 | 虚拟机 | Docker容器 |
| 创建时间 | 分钟级 | 秒级 |
| 启动时间 | 分钟级 | 毫秒级 |
| 硬盘使用 | 一般为GB | 一般为MB |
| 额外开销 | 有 | 无 |
| 是否共享内核 | 否 | 是 |
| 性能 | 弱于原生 | 接近原生 |
| 系统支持量 | 单机一般不多于几十个 | 单机支持上千个 |
| 实例间资源共享 | 一般不能 | 可以 |
| 资源分配粒度 | 粗粒度 | 细粒度 |
| 通信 | 通过以太网设备 | 标准的IPC机制 |

### 2.1.3 容器迁移技术

容器迁移是指将原宿主机上运行着的某个容器移动到目标主机，并能够在目标主机上正常运行，迁移的内容包括容器中运行的服务、运行状态等，如图2-4所示。由于云平台中负载是动态变化的，当容器调整资源配置时有可能出现本地资源剩余不足的情况，这就需要对容器进行迁移，为其选择一个资源充沛的节点以便伸缩。因此，在云平台中实现容器资源动态伸缩的一种重要技术就是微服务迁移技术。



图2-4 容器迁移实例

容器的迁移是个复杂的过程，包括目标宿主机的选择、迁移的时间点的确定等一系列步骤。可以根据迁移时容器的运行状态可以将迁移过程分为冷迁移（离线迁移）和热迁移（在线迁移）[41]。

（1）冷迁移

冷迁移是指把处于暂停或关闭状态的容器从源宿主机迁移到目的宿主机。在这个过程中，需要先使容器处于暂停状态，然后通过网络将容器的内存状态、配置等数据拷贝到目的主机中。因为容器的基础是镜像文件，所以接着将容器关闭，复制镜像文件通过网络将其保存到目的宿主机上，在目的宿主机上根据镜像重新创建容器，恢复其内存等状态，使原有服务继续运行。冷迁移的过程需要关闭容器，这就会造成在一段时间内服务不可用的状况，所以不适用于实时性要求较高的场景，因为操作简单，常用于对服务的要求不那么高的场景。

（2）热迁移

相对于冷迁移，在热迁移的过程中不需要暂停或关闭容器，能够保证容器持续的提供服务，这个过程对用户来说是透明的。热迁移的主要过程是保持容器的正常运行，然后将容器的CPU、内存运行状态等信息复制到目的主机，然后在一个恰当的时刻经过一个短暂的切换，在目的主机上启动容器替代源宿主机中的容器以提供服务，完成迁移过程。切换过程耗时非常小，用户基本察觉不到服务的中断，所以热迁移通常使用于对服务可用性要求较高的场景。

对于容器的冷迁移我们可以简单的通过Docker提供的export和import命令实现，export命令会将容器打包成一个tar文件，里面包含容器的所有数据，然后在目的主机上通过tar文件用import命令将其还原成一个容器。为了提高服务的性能，我们在资源重配置过程中的微服务迁移采用热迁移方式。

我们知道容器实质上是一个隔离的进程组实例，对容器的迁移本质上是对进程的迁移，容器可以通过checkpoint和restore实现热迁移。Checkpoint是指将容器的所有状态等信息都保存在文件中，restore是指根据保存文件中的信息恢复容器的执行。

Checkpoint包含以下几个阶段：

1）收集进程信息并冻结进程（Collect process tree and freeze）：从 /proc目录下收集文件描述信息、内存映射等信息，并将容器中所有进程在某个已知的状态挂起，同时暂停网络I/O。

2）复制容器（dump the container）：获取容器的完整状态包括CPU运行状态、内存状态等，然后将该容器状态保存在checkpoint文件中。

3）停止容器（cleanup）：当所有的转储完成后，结束容器中的进程，将容器中的文件系统卸载。

Restore是通过所dump出的checkpoint文件进行恢复的过程，其包括以下几个阶段：

1）重建容器（rebuild the container）：根据从源宿主机拷贝来的checkpoint文件，重新创建一个容器使其具有相同的状态。

2）重启进程（restart processes）：多次调用fork()来重新创建需要恢复的进程，并依据checkpoint文件将每个进程的内存状态及执行任务所使用的资源等逐一恢复。

3）恢复容器（resume the container）：恢复线程的执行，恢复网络及I/O的状态等，以便容器继续执行之前的任务。

因为是热迁移，容器在迁移过程中处于运行状态，CPU状态等数据比较少容易同步，但是内存状态是在时刻变化着的，如何对内存进行同步热迁移目前有两种解决方案被普遍认可，分别是pre-copy（预拷贝）和post-copy（后拷贝）[42][43]。下面分别介绍这两种方案，流程图分别见图2-5和图2-6：

Pre-copy：该方法是不论是在学术界还是工业界都是目前最成熟的热迁移方案，被广泛应用于Xen、KVM等平台。在迁移时，会在源节点上跟踪内存，将所有的内存复制到目标节点上，因为在复制内存的过程中，虚拟机仍在持续运行，所以会不断地修改内存页面，需要记录下当前被修改的内存页，然后迭代pre-copy。以后每次执行pre-copy，都只会复制上一轮的脏页，直到脏页到达某一阈值，源节点上该容器内存信息和目的节点上的内存备份差异变得最小为止。之后，它冻结容器，获取其余的状态（如剩余脏页、CPU状态等），将其迁移到目标节点，然后在目标节点恢复和解冻容器，最后从源宿主机上将该容器清除。如果在容器迁移的过程中发生任何故障，那么立即在源节点上恢复容器的运行。



图2-5 迭代pre-copy热迁移方案流程图[42]

Post-copy：该方案是首先将源宿主机上要迁移的容器挂起，然后将该容器的CPU运行状态、I/O状态等拷贝到目的节点，然后在目的主机上恢复容器的运行，将服务迁移到目的主机上，然后将源宿主机上容器的内存页按需同步到目的主机上。待所有的内存页拷贝完毕才可将源宿主机上的容器关机。该方法可以缩短停机所用的时间，但是因为会频繁的缺页造成总迁移时间增加。



图2-6 迭代post-copy热迁移方案流程图[42]

## 2.2 微服务

### 2.2.1 单体式架构

单体式架构（Monolithic Architecture）也叫集中式架构，它是指将所有的功能集成在一个应用中，各个模块中间通过相互调用的方式交互，通常按照软件的三层架构模式（表示层、业务逻辑层、数据访问层）去构建分层式应用，每一层都还可以根据具体的业务进一步细化使功能模块化[44]。虽然应用实现了业务分层功能逻辑模块化，但是在部署的时候还是打包成一个整体运行在同一进程中。对于单体式应用来说，当应用不是很大时，由于其开发方式、测试、部署乃至水平伸缩都比较容易，所以在软件领域中这种架构长期成为了主流模式[46]。但是随着业务规模的不断扩大、应用变的很复杂时，这种软件架构的不足就越发明显，CloudFoundry.com的创始人Chris Richardson总结了传统的单体式应用具有以下不足[47]:

（1）功能扩展维护困难。一个简单的应用会随着业务功能的增加而变得越来越庞大，应用太复杂，开发过程中代码质量下降也越来越难以理解，所以后续很难有单个开发人员对全局功能有深入的理解，这就使功能的扩展、错误的修复变得很难。

（2）持续部署困难。对应用即使有细微的修改或添加都需要重新构建和部署整个应用。单体式应用开发的过程相当于流水线，可能出现开发人员等待别的其他开发人员代码的提交，并且随着应用的庞大，整个部署流程花费的时间就相应的增加，不利于进行频繁持续部署，也不利于软件的快速迭代。

（3）可伸缩性差。若将整个应用的模块打包放置在一台服务器上，当用户的访问量增加服务器某一种资源供应出现瓶颈时，很难对应用程序进行扩展。虽然通过投资提高服务器的性能进行垂直扩展比较容易，但是随着业务量的增加，垂直扩展会变的越来越难，成本越来越高。通常的做法是进行水平扩展，通过将打包的应用重复拷贝到这些新增加的集群节点中，然后通过负载均衡将用户的请求分散到不同的节点上。不同模块的资源倾向可能不同，有的可能是CPU密集型，有的可能是内存密集型，单体式架构并不能对各个模块进行伸缩，只能作用于整个应用，这会造成资源的大量浪费。

（4）可靠性低。由于应用的所有功能模块作为一个整体在同一个进程中运行，其中任何一个模块出现故障，比如内存泄露或运行时异常等，都会造成整个应用程序的瘫痪。除此之外，因为所有的模块在整个应用中都是唯一存在的，模块中的任何异常都会影响整个应用的可靠性。

（5）受技术栈限制。单体式应用倾向于在整个开发过程中使用统一的技术或方案。计算机行业，技术更新迭代较快，随着应用越来越复杂，如果后续希望尝试新的技术或框架风险代价会比较高。

从单体式架构的弊端我们可以知道，单体式架构不适于结合云计算技术构建云服务。它不仅难以用于构建分布式应用，还难以在云平台进行调度，容易造成资源浪费。因此，在云平台中，如今微服务的架构正在逐渐取代单体式架构。微服务架构将会在下面进行详细介绍。

### 2.2.2 微服务架构

随着云计算技术的普及发展，互联网新技术的不断涌现，采用传统的单体式架构来开发应用带来了太多的问题，使之催生出了一种新的软件架构风格即微服务架构（Microservices Architecture）。近几年微服务架构伴随着Docker容器技术成为人们热议的话题，其主流趋势是通过将庞大的系统进行有效拆分成多个细粒度化的应用实例，每个实例运行在一个Docker容器中来构建分布式的云上服务。微服务架构与传统的SOA（面向服务架构）模式相似，可以看做是在其基础上去掉中心化发展而来的。目前对微服务的理解是仁者见仁智者见智，Martin Fowler等[45]这样定义微服务架构：“微服务架构是一种将单个应用程序开发为一组小型服务的方法，每个小型服务不拘泥于使用同一编程语言及存储技术进行开发，可以独立运行在自己的进程空间中，相互之间通过轻量级机制（REST、RPC）进行通信。这些服务围绕业务功能构建，可通过全自动部署机制进行独立部署实现去中心化、集中式管理最小化[44]。”

简而言之，微服务架构是一种全新的软件架构风格，将传统的整体式应用分割成一系列微小的服务，每个服务都具有单一职责，各个服务都可以进行独立部署运行，相互之间通过轻量级通信机制进行交互协作来实现完整的应用，满足业务需求。图2-7是微服务架构和单体式架构应用部署对比图。

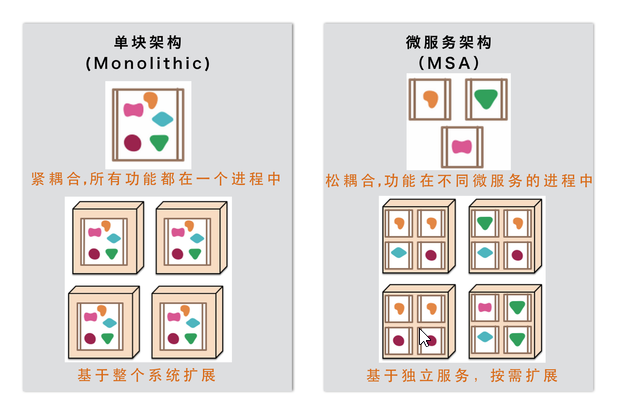


图2-7 单体式架构和微服务架构[48]

下面简要介绍下微服务的特征：

（1）自治性。一个微服务就是一个独立的实体，服务的运行不需要依赖其他的任何服务，彼此之间互相隔离通过网络调用进行通信。微服务自身就能够提供一个独立完整的服务，并有一套独立的运行机制和外部通信的标准化应用编程接口，能够独立的部署、升级、扩展等。

（2）单一职责原则。每个微服务负责一个应用中单独的功能，这是SOLID原则之一。

（3）轻量化。在软件架构上采用去中心化，服务之间交互采用轻量级通信协议（RTSP、AMQP等）。

（4）可组合性。每个微服务是相互独立的，在功能上具有单一性，因此一个微服务并不能构成一个完整的应用系统。每个微服务向外提供标准统一的接口，可以根据自己的需求将具有相应功能的微服务组合来构建自己的系统，具有高内聚低耦合的特点。并且对系统的重构也是非常方便的。

从上面的特征，我们可以总结出微服务所具有的优点，每个服务彼此之间相互独立相互隔离易于开发、维护；对局部功能进行修改不需要重新编译整个系统，有利于系统的扩展交付；能实现故障隔离，一个服务出现问题不会造成整个系统的崩溃；不同微服务可以使用不同的编程语言进行开发，不会受到技术栈的限制，可以提高开发效率提升系统的性能等。结合微服务的这些优点可以看出微服务架构适合构建云应用。除了上面所述的优点，微服务架构的细粒度也使得云平台中的资源管理、负载均衡实现等变得非常容易。

当然，微服务架构并不是没有缺点，通过将整个应用分解成一个个的微服务，这就加大了应用部署的困难，由原来的只需要对一个打包应用进行部署变成了对多个微服务进行部署。并且每个微服务的运行都需要相关的依赖环境，为避免资源浪费要尽量使环境只包含与服务相关的配置。可以通过将微服务部署在虚拟机或容器中来解决上述问题，但是虚拟机相比容器具有粒度太大、性能有损耗等缺点。所以近几年来热门的Docker容器技术与微服务组合的模式是一个完美的解决方案。

## 2.3 云平台资源预测模型

资源预测分析是通过对历史数据进行趋势分析，找到其潜在的变化规律，从而可以准确的预测未来的资源需求量。目前比较流行的预测算法有：时间序列预测、神经网络预测、逻辑回归预测等[49]。每种方法都有各自的优缺点，下面在本文中主要选取时间序列预测模型和神经网络模型进行简要介绍。

### 2.3.1 时间序列预测模型

时间序列[49]可以看成是按照时间的顺序排列的随机变量序列。通过目标的历史信息来预测其未来的发展趋势是时间序列预测模型的主题思想，该模型的基本特征就是前后数据之间具有相关性，并且该模型具有动态性，可用于云平台的动态预测。

时间序列一般是由系统性及非系统性两个部分组成。系统性部分一般由水平、趋势和周期性三种成分构成，非系统性部分是指由测量误差或其它因素引起的随机噪声。水平是指时间序列的平均值，趋势成分是指在一个长时期内时间序列受基本因素影响随时间变化的趋势，周期性成分是指在一个长时期内时间序列重复出现的波动。对于每个时间序列来说，水平成分是其必备组成元素，但是并不一定具备趋势或周期性成分。通过合理的差分变换等方式可以使任意的时间序列包含趋势、周期性及随机噪声等成分。典型的线性时间序列模型包括MA（滑动平均模型）、AR（自回归模型）、ARMA（自回归滑动平均模型）以及ARIMA（差分自回归移动平均模型）等[50]。其中AR、MA及ARMA都只能处理平稳的数据序列，而ARIMA可以通过差分将非平稳序列转换成平稳序列后再使用ARMA模型进行建模。理论上，ARIMA(p,d,q)模型是最常用的时间序列预测模型。ARIMA(p,d,q)模型可以抽象成公式2-1表示，

(2-1)

对于ARIMA(p,d,q)而言，AR代表自回归，p是自回归阶数，为自回归参数；MA代表移动平均，q为移动平均项数，为移动平均参数；d为将时间序列变成平稳序列所需要的差分次数，是差分算子，其中B是滞后值。上面式子中表示工作负载是平稳、正态、均值为零的时间序列，表示见公式2-2，是白噪声序列。基于白噪声序列和历史负载的时间序列以及滞后值来预测在t时刻的资源使用量表示为。

(2-2)

当存在差分阶数d时，必须进行相应的转换才能得到原始序列的预测结果，可表示为公式2-3，

(2-3)

根据公式可以看出，当d=0时，ARIMA(p,0,q)就退化成ARMA(p,q)模型；当p=0并且d=0时，ARIMA(0,0,q)就变成MA(q)模型了；当d=0,q=0时，ARIMA(p,0,0)转化为AR(p)模型。ARIMA模型相较其他三种模型来说，具有更小的资源预测误差，具有更稳定的特点，这三种模型都适合进行短期预测，预测步数越少，精确度越高。ARIMA可以对非稳定时间序列进行差分处理，变换成稳定序列，但是随着差分次数的增多会造成预测误差变大。

### 2.3.2 神经网络模型

近几年神经网络被广泛应用于云资源预测领域，相对于ARIMA算法其不仅在线性时间序列中适用也适用于非线性时间序列。神经网络用于获取预测变量和目标变量之间的复杂关系，通过一系列层将预测变量和目标变量关联起来[49]。在每一层中，都会对输入信息进行一系列操作，生成该层的输出变量（衍生变量），该层的输出又将作为下层的输入变量。神经网络包含三类层，如图2-8所示。

（1）Input Layer（输入层）：用来输入变量值。

（2）Hidden Layer（隐藏层）：可以有一个或多个，负责接收上一层的输出数据作为本层的输入，并对输入数据进行相关计算，生成输出变量。

（3）Output Layer（输出层）：接收最后一层隐藏层的数据，生成预测结果。

从图中可以看到每一层的节点数都不是固定的，每个节点表示一个向量。输入层的节点是原始的输入变量，隐藏层的节点是中间生成的衍生变量，输出层的变量是最终的目标变量。它们是输入变量的某个单调函数值或输入变量的加权和。常见的隐藏层函数有线性函数、指数函数或s型函数（如logit函数、双曲正切函数）。



图2-8 神经网络模型示意图

对于一组输入,节点j的输出就是输入的加权和，即, 其中是权重，它们的初始值可以随机设置，随着神经网络的自学习过程来调整它们的取值。负责控制节点j的水平。随后在该加权和上应用核函数来生成节点j的输出，即。如图2-8中，节点3计算节点1和节点2的加权和,然后在其上应用核函数。节点6同样是节点3、4、5的加权和后的核函数值。神经网络模型可以有多个输出层，可以往前多步预测。如图2-8中的模型是基于滚动预测的，即先预测出，然后基于往前滚动预测。要想通过神经网络模型进行准确的预测，一般常常需要添加外部信息参数。对于神经网络模型来说，其输入参数的范围在[0,1]或[-1,1]之间才能达到最好的效果，因此需要对时间序列数据进行参数转换预处理操作。神经网络模型提供精准预测的同时，存在计算资源耗费严重、实时性较差、训练样本集较大等缺点。

## 2.4 云平台资源调度策略

在云平台中，视频服务一般是以虚拟机或容器的方式提供服务，所以对视频服务的调度可以归结为是针对虚拟机或容器的调度。下面我们分别看看Openstack和Docker中分别对虚拟机和容器以怎样的调度策略来进行资源配置。

### 2.4.1 Openstack平台中的资源调度策略

Openstack中实现的调度策略有以下四种：ChanceScheduler（随机调度器）、SimpleScheduler（简单调度器）、FilterScheduler（过滤调度器）和MultiScheduler（多重调度器）。其中ChanceScheduler是随机的从节点中选择一台与虚拟机请求映射，SimpleScheduler总是从集群中选择负载最小（实例最少、Volume使用最少）的节点。因为这两种方法太简单，没有很好的效果，一般不会在生产环境中使用。Openstack默认的是MultiScheduler，但其本身并不包含具体的调度算法，只是多种子调度器的集合体，所以主要用到的是FilterScheduler。它主要的步骤可以分为条件过滤和权值计算，图2-9是其工作流程图。



图2-9 FilterScheduler工作流程图

（1）条件过滤

该步骤通过根据指定的过滤规则将不满足指定条件的节点从列表集合中删除，得到满足条件的宿主机列表。其已实现多种过滤策略，默认使用的过滤器如表2-3所示。

（2）权值计算

对上面满足过滤条件的宿主机列表进行权重计算，然后对集合进行排序，就得到一个按照权重从高到低的排序列表。目前Openstack中weighter仅考虑内存指标，内存剩余量越大，该宿主机拥有的权重越高。

表2-3 相关过滤器介绍

|  |  |
| --- | --- |
| 过滤器 | 说明 |
| RamFilter | 根据内存剩余量进行过滤，剩余量不足的过滤掉 |
| ComputeFilter | 所有能提供计算能力的计算节点都能通过 |
| AvailabilityZoneFilter | 根据可用区域过滤，选择与指定虚拟机位置相同的宿主机 |
| CoreFilter | 根据CPU的使用率过滤，过滤掉CPU核使用过多的宿主机 |
| ImagePropertiesFilter | 基于镜像属性过滤，选择支持虚拟机指定的镜像属性的宿主机 |

Yang[51]针对视频监控云服务及宿主机资源的多维性和异构性，提出了一种主导资源优先的分配算法，该算法充分考虑了视频服务的资源需求特点，有效的提高了整个云平台的资源利用率。

### 2.4.2 Docker集群中的资源调度策略

Swarm是Docker的原生集群管理工具，它在给容器分配最佳节点时，也是通过两个阶段。首先是根据用户的过滤条件筛选出满足条件的节点集合，然后通过相应的调度策略选择最优的节点部署。其主要支持三种调度策略：Random、Spread和Binpack。

（1）Random，从满足过滤条件的节点集合中随机选择一个节点，不需考虑节点上的资源使用状况等。

（2）Spread，通过计算节点上的CPU、内存等剩余资源量和节点上容器的数量将容器调度到负载低的节点上，这样可以使系统达到较好的负载均衡，从而提高系统的高可用性。

（3）Binpack,与Spread相反，它是将容器调度到负载最高的节点上，可以使容器部署集中，减少宿主机的数量，从而降低能源消耗。

## 2.5 本章小结

本章首先介绍了当前云平台中所常用的资源预测模型及资源调度算法，然后介绍了云环境的相关知识，包括Docker的原理等及资源配置所用到的迁移技术和开发视频服务会采用到的微服务架构等。本章主要是为后面的章节打下理论基础。

# 第三章 监控视频云计算平台资源预测方法

在监控视频云计算平台中，视频微服务的负载是动态变化的，为了动态调整资源的大小以降低违背SLA的风险及提高资源利用率，这就需要我们能够有效的预测出视频微服务的负载变化趋势，并在负载变化前做好相应的资源伸缩准备，以防止对业务的运行造成影响。在本章中，我们通过分析视频服务的负载特性及资源特性，并分析了当前的云资源预测算法在视频监控云平台中的不足之处，提出了适应于视频监控服务的资源预测算法。文章后半部分详细介绍了所提算法的具体内容及步骤。

## 3.1 负载及云资源特性分析

### 3.1.1 负载特性

为了选择合适的算法模型以确保对服务资源需求预测的准确性，这就需要我们对应用负载的特性有一定的了解。Peter A. Dinda历经两年从多台不同的机器（集群、计算服务器、桌面主机）及这些机器上面的应用程序取得了大量的动态负载信息。通过详细的汇总分析所得到的这些负载信息，可以将负载的特性总结为以下几个方面[53]：

（1）负载具有随机性。其变化的过程是随机的没有固定的模式可循。

（2）负载具有波动性。通常在大部分时间内都在低水平范围内，但不是持续保持平稳状态而是有一定的波动变化。

（3）负载的绝对波动性不仅表现在平均负载值低的应用集群中，其对平均高负载的应用也具有通用性，即负载的最值之间相差较大。不过对于相对波动量而言，负载均值低的要比负载均值高的更具有波动性。负载均值和最大值之间的相关性表明为了具有更高的价值可以将负载预测应用于重负载的机器。

（4）负载值的分布具有复杂性，以非常复杂的方式在大范围内变化。负载值的复合多样性与负载均值两者成正相关，这意味着对负载值的分析预测不能只是简单的通过分析负载的分布曲线而应该采用数据驱动的方法。

（5）负载的变化与时间具有强相关性，即未来的负载值依赖于历史的负载观测值。这意味着基于负载的历史数据来预测未来的负载这个方案是行得通的，并且适合用时间序列模型来对将来的负载进行预测。

（6）负载的变化具有高度的自相似性，有相对高的赫斯特参数，就是在所有的时间尺度下负载的变化既复杂又具有长期的依赖性，因此负载的模型化和预测是困难的[54]。

（7）负载的变化呈现突变性。负载值可能在某一个时刻会突然增加或突然降低造成比较大的变化，然后趋势慢慢变的平缓在长时间段内保持相对稳定。这种负载变化的特点意味着负载预测模型需要具备自动修正误差以及适应这种变化的机制。

### 3.1.2 资源特性

在云平台中，云资源管理应该是其主要的内容。其通过虚拟化技术将底层物理资源等生成弹性资源池，通过对宿主机上生成的容器进行监控、分配、迁移和回收等操作来实现对虚拟资源的统一管理。但是为了提高资源的利用率，我们应该充分考虑分析资源的特性，不仅包括物理机的资源还包括视频服务的资源特性，以便更合理的进行配置调度。概况来说云资源主要有以下几个方面的特性[51]：

（1）多维性。资源的多维性应该是我们首先考虑的问题，每个物理机都包括CPU、内存、磁盘、带宽等多维资源，在其上运行的容器所拥有的资源一样也具有多维性。每个容器或虚拟机为用户提供服务都是多种资源协同的结果，所以在资源配置的过程中需要综合考虑各种资源的合理分配，以免造成资源浪费。

（2）动态性。其次资源的分配应该是个动态的过程，虽然物理机的固有资源量是不变的，但是随着容器的创建、销毁都会涉及到资源的分配回收，并且容器中的服务的资源需求量也是动态改变的，因此其资源利用率是不断变化的。如果固额分配有可能会造成多度分配造成资源浪费，也有可能某段时间的资源需求量超过资源分配量违背SLA协议。所以资源的动态性在资源的重新配置方面是个需要考虑的因素。

（3）异构性。是指同一物理机或容器中不同资源间所呈现的不均衡关系，资源的多维性必然导致异构性。视频服务不仅需要非常高的计算性能，而且不同的服务在资源需求量上也有显著的不同。比如有的视频服务就是CPU密集型的任务需要大量的计算，如视频浓缩、车牌识别等；有的服务是I/O密集型的，不需要大量计算但需要处理大量的I/O请求，如视频直播等。随着容器的不断创建，硬件设备的更新换代，这些情况都会造成物理机的异构性。如何将容器与物理机进行一一映射，这是一个复杂的多维装箱问题。

## 3.2 云资源预测方法

从上文中，我们可以看到云平台中的视频服务负载变化具有随机性、突变性等特征，传统的预测算法如移动平均法、指数平滑法、灰色模型等无法准确的拟合负载的变化，预测的误差较高；人工神经网络算法其需要大量的训练样本，收敛速度较慢，并且需要更多的计算和存储成本，不适合实时性的预测。目前并没有发现有针对视频服务的特性进行预测的算法，这些传统的预测算法大部分都是针对网络流量进行预测。从负载的特性可知，视频应用负载的变化与时间具有强相关性，即可以将视频服务的负载看成一种时间序列，并且与自身趋势有高度相似性，因此对视频服务的负载预测可以根据时间序列的预测方式执行。所以我们基于视频服务的自相似性、时间相关性、随机性及波动性等特点提出了适用于视频监控服务的资源预测模型。下面详细介绍我们所提出的预测模型。

### 3.2.1 问题描述

为了表述的方便性，首先将视频监控云平台中所涉及到的和资源预测相关的内容以及属性进行数值化表示。假设视频监控云平台中视频服务集合为 ，其中 表示第*i*个视频服务。每个视频服务可以表示为，其中代表任务的第*k*个属性，每一个视频任务包含*n*个特征属性（分辨率、码率、任务类型等）。视频任务在CIVU中执行时的资源使用量描述为，其中分别代表CPU、内存、上行带宽和下行带宽的资源使用量。因为每个资源使用量的表示类似，这里以CPU的资源使用量为例，每个CIVU从服务开始到当前时间*t*的CPU资源使用量可以表示为，*t*表示时间序列的下标，例如*t*=1代表时间序列的第一个时期。我们将CIVU中历史资源使用数据，按照固定的时间周期划分成一个个时间段，形成一个时间段序列，根据来预测未来的资源负载{}, 其中表示第*t+*1时期的预测值，*p*表示前往预测的步数。

### 3.2.2 算法描述

因为我们是针对容器的动态资源需求进行预测，这就需要我们有一定的历史数据供我们建立模型，所以根据历史数据量将预测方法分为了两个阶段：初始资源预测阶段和运行时持续资源需求预测阶段。

**（1）初始资源预测阶段**

因为在这个阶段，根据用户的请求在服务部署过程中，数据库中是没有任何该服务的历史运行数据，所以我们就没法通过时序回归的方法去对其资源需求进行预测。但是在数据库中有很多其他服务所分配的*R*个CIVU的资源使用数据。对于用户所提交的服务请求，我们怎么才能给这个CIVU更加准确的分配合适的初始资源量，避免其造成资源严重浪费的同时又要保证不违背SLA协议。我们可以通过将用户请求的服务和数据库中已存在的服务历史数据进行匹配，数据库中不仅包含服务的资源使用量信息，还有该服务的属性标签（如服务类型、码率等）。我们从之前执行的众多服务中找到一个和用户所请求服务最相似的一个，用那个相似服务的资源使用峰值（为了尽量避免SLA）作为初始的资源分配量。基于服务相似性资源预测算法如算法3-1所示。

首先将服务的属性值进行归一化操作，然后通过归一化的属性向量对所请求的视频监控服务和数据库已有的视频服务的属性特征用公式3-1求其L2范数

(3-1)

使DISTANCE最小的那个服务就是和用户所请求的服务是最相似的，为了避免降低服务质量，我们选择该服务的资源使用最大值作为分配量，见公式3-2。

(3-2)

|  |
| --- |
| **算法3-1 ：**基于服务相似性预测算法 |
| **输入**：用户请求的视频服务taski，数据库中所存在的容器化智能处理单元集合数据CIVU\_Set，其中每个CIVU负责处理一个task  **输出**：预测初始资源需求量initialV  1 minDistance ← ∞;  2 if CIVU\_Set不为空 then  3 xi ← normalize(taski.attributeVector)；  4 for ∀CIVUj ∈ CIVU\_Set do  5 taskj ←CIVUj.execute()；//获取第*j*个CIVU中运行的服务  6 xj ← normalize(taskj.attributeVector)；//将视频服务的属性归一化  7 distance ← DISTANCE(xi, xj)；//计算任务*i*和任务*j*之间的相似性  8 if minDistance > distance then  9 minDistance ← distance；  10 initialV ← max(CIVU j.resource) //将最相似任务j的峰值资源作为初始资源量  11 end if  12 end for  13 else  14 initialV determined by experience;  15 end if |

**（2）运行时资源预测阶段**

在这个阶段，我们提出了一个时间序列的最近邻回归（TSNNR）算法，利用历史数据时间序列的趋势相似性来预测未来一段时间的资源需求。TSNNR算法的伪代码如算法3-2所示，其具体步骤主要有以下几步。

首先，我们从存储在数据库中该任务的历史时间序列数据中找到与当前时间*t*最近*q*个时间段数据最相似的一个时间段序列。这个时间序列数据可以通过使用L2范数，用公式3-3表示。

(3-3)

得到,其中表示实例存储在数据库中的过去时间，*t*表示当前时间，*q*表示我们使用当前时间后面的多少时间段作为一个比较序列，即表示一个预测周期的时间段步长。上述步骤递归地执行*k*次，就会得到*k*个最近邻时间序列集合。

第二步，我们通过前面找到的*k*个最相似时间序列数据来估计未来的一段时间的资源需求量。我们认为历史时间序列数据中与当前时间紧邻的前一个时间序列数据的趋势相似，那么的下一个时间序列和的下一个时间序列也有相似的趋势。具体的方式就是我们用*k*个时间序列的下一个时间序列数据与其的均差值与当前序列的数据值相加作为下一阶段的预测值，用公式3-4描述为

(3-4)

上面我们只是往前预测一步，如果我们想往前预测*p*步，我们可以将预测值作为历史数据，寻找与数据序列最相似的历史序列数据，求出下一阶段的预测值。如此的重复执行上面的步骤*p*次，就可以得到未来*p*个时间周期的资源负载预测值集合。

最后，为了保证服务质量，我们选择预测资源负载的峰值加上一个预测误差*e*作为最终的资源预测分配值，见公式3-5。

(3-5)

*e*为历史预测误差的平均值，通过公式3-6求得。

(3-6)

这样我们就预测出在未来一段时间内CPU的资源需求量，同理对于内存和带宽的预测方式是相同的。

|  |
| --- |
| **算法3-2 ：**最近邻回归（TSNNR）算法 |
| **输入**：该任务的历史时间序列数据historicalData，预测周期的时间段步长*q*, 递归执行次数*k，*超前预测步长*p*  **输出**：资源需求预测值predictionV  1 DataSet ← segmente(historicalData);  2 recentData ← get(DataSet, q);  3 for steps < p do //往前预测p步  4 for num<k do //找到k个最近邻时间序列  5 minDistance ← ∞;  6 for ∀pastData∈DataSet do  7 distance ← DISTANCE(rencentData, pastData);  8 if minDistance>distance then  9 minDistance ← distance;  10 similarV ← pastData.next;  11 end if  12 end for  13 D\_valueSet ← sub(similarV-similarV.pre);  14 end for  15 predictionSet ← avg(D\_valueSet)+recentData+e  16 end for  17 predictionV← max(predictionSet);  18 return predictionV; |

## 3.3 本章小结

本章首先分析了云平台中宿主机及其上运行应用的负载和资源的特性，然后指出当前预测技术存在的不足之处。后面我们根据视频服务负载的动态性、自相关性、与时间相关性等特性结合资源的多维性、异构性、动态性等特征提出了基于服务相似性的预测算法和基于时间序列的预测算法TSNNR，从而为下一阶段的资源配置提供准确的预测值。

# 第四章 监控视频云计算平台资源配置与优化

本章首先介绍了当前主流的视频监控云平台，分析了其采用的资源配置方式存在的不足，从而对现有的基于Docker的视频监控云平台进行优化，在平台中添加了资源预测模块并对资源管理模块进行优化使其可以根据预测结果对容器进行资源的动态调整。接着介绍了云平台中的几个重点模块设计。

## 4.1 监控视频云计算平台介绍

为了表述方便，视频架构图中部分功能模块命名参考了ITU-T标准[55]，图中命名参考表4-1所示。

表4-1 视频监控云计算平台命名参考

|  |  |  |
| --- | --- | --- |
| 缩写 | 英文名称 | 中文意义 |
| CRM | Cloud Resource Manager | 云资源管理 |
| VCSM | Video Cloud Service Manager | 视频云服务管理 |
| CIVU | Containerized Intelligent Video Unit | 容器化的智能视频处理单元 |
| VS | Visual Surveillance | 视频监控系统 |
| PU | Premises Unit | 前端设备 |
| CU | Customer Unit | 用户单元 |

（1）基于虚拟机的视频监控云平台

当前大部分的视频监控云平台都是通过Openstack等开源项目来实现云基础架构服务（IaaS），在其上通过在生成的虚拟机上部署视频监控应用的方式来为用户提供服务。传统视频监控云平台架构如图4-1所示，VS视频监控系统负责用户和云平台之间的交互；CRM云资源管理组件负责为用户的请求进行虚拟机的调度和资源的管理；VM上面搭载了视频监控应用；Monitor负责监控虚拟机及宿主机的资源使用状况，并以可视化的方式展示给平台管理员。

虽然这种基于虚拟机的方式解决了传统的将一个系统固定的部署在单个服务器上带来的部署困难、难以扩展、资源浪费严重等一系列问题，但是其还是存在一些不足之处。我们在第二章总结过虚拟机相较容器存在的一些差别，可以看出将视频监控应用服务放置在虚拟机上会有一定的性能损耗，并且虚拟机的资源分配粒度大，导致资源利用率低；并且在虚拟机上部署的应用一般都是采用单体式架构，高内聚低耦合的方式容易造成系统崩溃，在视频监控云平台中一般是将视频应用打包成一个镜像，方便启动部署以降低部署开销。但是制作镜像比较困难，并且镜像不具有可编程的特性，视频应用稍加改动都需要重新制作镜像，这就造成服务维护、扩展困难。



图4-1 基于虚拟机的视频监控云平台架构图

（2）基于Docker的视频监控云平台

综上所述，基于虚拟机的视频监控云平台存在些许不足，当前采用比较热门的Docker容器技术来架构新的视频监控云平台也越来越流行，并将其上提供的视频服务采用微服务的架构进行微服务化使之与Docker容器结合更好的为用户提供服务。当前基于Docker的视频监控云平台的总体架构如图4-2所示（不包括红框内的资源预测模块）。



图4-2 基于Docker的视频监控云平台架构图

通过系统架构图，可看到视频云监控平台包含多个模块，下面介绍一下每个部分。

（1）云资源池：云资源池是云视频监控系统的核心组成，主要包括很多服务器主机，平台的物理计算资源，每个物理机上面均部署Docker引擎负责创建和销毁容器，其主要功能是为视频微服务提供资源。

（2）视频云服务管理模块（VCSM）：主要管理视频微服务，例如视频转码、视频浓缩、车辆检索等。当用户通过web平台提交一个服务请求时，VCSM将会触发云资源管理模块配置分配资源并控制容器引擎生成一个容器化的智能处理单元为用户提供服务。主要任务是接收并分析从VS发来的服务请求；管理一系列的CIVU，并负责各个服务组件之间的信息交互；收集视频服务处理结果并将结果返回给用户。

（3）容器化的智能视频处理单元（CIVU）：是一个可以完成指定任务的编程化的Docker容器，可处理视频转发、视频编解码、车辆检索等。可通过从镜像仓库中加载相关功能的镜像生成该容器，在云资源池中每个CIVU有唯一的ID，每个CIVU的配置是独立的并且根据不同应用的需求而各异，每个CIVU可毫秒级启动。

（4）资源监控模块：用来实现实时的监控每个处于运行态的CIVU和宿主机的资源使用状况。它主要包含数据采集子模块和数据库，数据采集子模块监控每个运行着的CIVU及宿主机并收集其CPU、内存、I/O等使用率信息将其存放在数据库中。

（5）云资源管理模块（CRM）：负责将分散的、异构的基础设施资源统一管理起来。

（6）镜像仓库：是视频微服务镜像的存储中心，保存着平台中所有已构建的视频微服务镜像，同时当创建新的视频微服务时为计算节点提供镜像服务。

（7）视频监控系统web平台（VS）：该平台是用户和视频监控平台的交互界面，普通用户通过该web页面可以使用该监控云平台所提供的的视频应用服务上传视频进行处理，并且可以直观化的查看视频任务的处理进度等。管理员用户通过该web端可以对视频微服务进行管理，比如查看微服务列表，添加新的微服务等。

（8）视频采集设备（PU）：包括网络摄像头、NVR等设备，负责采集视频信息，以便传输给视频监控云平台处理。

（9）用户终端设备（CU）：用户通过终端设备接入视频监控系统提交服务和接收处理结果，主要包括个人PC、手机等。

## 4.2 监控视频云计算平台的资源配置方法

当前大部分云平台对用户请求所提供服务的虚拟机或容器都是固额分配，因为负载是动态变化的，这就会造成如图4-3所示的情况，资源配额过高造成资源浪费，资源配额过低不能提供高质量服务。针对资源的自动伸缩有的采用资源负载监测机制，即通过设定两个阈值分别代表最高负载限和最低负载限，当监测模块监测到负载到达所设的阈值时就会相应的自动伸缩，然而这种方式是被动伸缩，即只有在当前资源不满足需求时才去扩展，容易违背SLA协议。

解决这个问题的方式是可以运用云资源预测技术通过预测结果进行主动伸缩，这样可以既能提高资源利用率又不影响应用的运行性能。无论是通过负载监测机制被动伸缩还是基于资源预测方法的资源重配置都要依靠调度策略来实现负载均衡或提高资源利用率。

图4-3 典型的业务负载变化及传统的资源分配方式[52]

## 4.3 监控视频云计算平台资源配置优化

上文中已经总结了当前主流的视频监控云平台的资源配置存在的一些问题，为了解决这种问题，我们对基于Docker的视频监控云平台进行了优化，架构图见图4-2所示，所做的优化见红色区域，主要添加了资源预测模块及对云资源管理模块进行了优化调整。与之前的平台相比所做优化体现在下面两个模块中：

（1）资源预测模块（Resource prediction）：这一模块是我们系统设计的重点之一，该模块主要实现了对资源监控模块收集到的历史数据进行处理并训练预测模型，在服务开始启动阶段以及运行过程中周期性的预测服务未来一个或几个阶段的资源需求量，以便资源管理模块能够动态的对容器的资源进行垂直伸缩。

（2）云资源管理模块（CRM，Cloud Resource Manager）：负责将分散的、异构的基础设施资源统一管理起来。对该模块的优化主要是使其根据预测模块的预测值去分配资源，并且优化了其调度策略。当创建新的视频微服务时或对视频微服务的资源进行调整时，资源管理器根据调度策略选择合适的宿主机进行迁移，然后根据预测模块提供的下一阶段容器资源预测值动态的伸缩容器的资源量。一旦CIVU被销毁它所占用的资源立马被回收。

### 4.3.1 工作流程

相较于传统的视频监控云平台来说，架构上的差异也体现在工作流程上。在本小节中，我们介绍一个该平台的工作流程，主要包括服务部署、服务迁移、服务销毁三个阶段：

（1）服务部署：用户通过客户端浏览器接入视频监控系统VS，提交自己的视频监控任务，VS将用户的请求转发给VCSM；然后VCSM从镜像仓库中查找对应的视频服务镜像，并且将资源请求发送到CRM；CRM接收到请求后调用资源预测模块获得该服务的资源需求量，然后根据相应的调度策略选择合适的宿主机，并调度当前主机中的容器引擎基于前面查找到的视频微服务镜像生成容器实例CIVU。创建完成后向VCSM发送确认消息，交给VCSM进行管理，CIVU启动后，CIVU中的视频处理程序将自动运行。

（2）服务迁移：在CIVU运行阶段，CIVU的资源使用情况由资源监控模块进行收集存储。资源预测模块利用存储在数据库中的历史资源使用数据周期性地预测下一阶段CIVU的资源需求量。CRM会周期性的从数据库中获取资源预测结果，然后会根据预测结果对CIVU按照相应的资源重配置策略进行动态的垂直伸缩。

（3）服务销毁：对于短期任务来说，一旦CIVU中的视频处理任务完成结束或用户终止视频处理任务，CIVU将处理结果返回给用户，然后自动销毁并将消息传递给VCSM使之在服务注册中删除该服务信息，并自动释放资源。

相较于传统的基于虚拟机的云平台而言，因为Docker容器是轻量级的，其创建和销毁都是开销极小的，所以承载视频服务的容器都是即时即用的，不需要实现生成大量的CIVU闲置以备使用，这样也避免资源的浪费。为了减小每次创建都要从镜像仓库拉取镜像造成的开销，容器在第一次拉取镜像后将镜像保存在本地，而不随该镜像的销毁而消亡，这样可以在下次同类服务创建时直接使用，提高速度降低开销。

### 4.3.2 资源监控模块

在生产环境中，我们对云平台中运行着的容器微服务不能放任不管，而是应该时刻知道应用运行的状态，以便应用在发生故障时及时知晓。资源监控模块主要做的工作就是对容器以及容器所在宿主机的各种资源使用情况（如CPU的使用率、内存的使用量、磁盘及网络I/O的流量使用等）进行监测，并将这些数据保存到数据库中以供预测模块使用。并且可以把所监控的信息以可视化的方式展示出来。本系统中的资源监控模块的架构如图4-4所示。



图4-4 Docker集群监控架构图

从图中我们有可以看到该监控模块又涉及到如下几个主要部分：Client 、DBase、Monitor Server、Docker Deamon。其中Client主要负责监控数据的可视化展示；DBase是数据库负责保存数据；Server起关键作用，主要负责对获取数据的处理；Docker Deamon是提供REST API以便调用获取Docker容器监控数据。对容器资源的进行监控的流程如下所示：

（1）Monitor Server定期的调用Docker Deamon提供的stats接口来获得容器的资源使用数据。

（2）待收到返回的容器监控数据后，Server负责对数据进行处理，因为数据是以json的格式返回的，然后将处理后的数据存入数据库中。

（3）Client从数据库中拿取数据进行可视化展示。

通过Docker所提供的REST API stats接口我们可以获取容器的CPU、内存、网络、磁盘等监控数据，但是数据是以JSON的格式返回的，因此需要对数据处理后才能存储到数据库中。数据格式如下所示：

|  |
| --- |
| HTTP/1.1 200 OK  Content-Type: application/json  {  "read" : "2016-12-10T22:57:31.547920715Z",  "network" : {  "rx\_bytes" : 648,  "tx\_bytes" : 648  …… },  "memory\_stats" : {  "max\_usage" : 6651904,  "usage" : 6537216,  ……},  "blkio\_stats" : {  "io\_service\_bytes\_recursive":  [{"major":8, "minor":0, "op":"Read", "value":844887 },  …… },  "cpu\_stats" : {  "cpu\_usage" : {  "total\_usage" : 100215355,  …… },……},  } |

Docker所提供的stats接口返回的结果只是容器的资源使用状况，并没有宿主机的资源信息，所以我们另外实现针对宿主机的监控，并提供标准的REST API供Monitor Server调用，我们会在后面的实现部分具体介绍。

### 4.3.3 资源预测模块

预测模块主要是从数据库中获取监控模块所采集的容器及宿主机的资源使用数据及视频服务的属性特征，预测出初始阶段或下一阶段的CIVU的资源需求值，然后将这一预测结果传给资源管理模块同时把结果保存到数据库中。本文已在第三章提出了针对视频服务的云资源预测算法，该算法是本模块的核心。该模块主要具备数据预处理、预测匹配两个方面的功能。其架构如图4-5所示。

（1）数据预处理：在数据采集传输的过程中，并不能保证数据的可靠性，所以保存到数据库中的信息必然包含一些噪声，比如数据丢失、数据奇点等。因为我们采用的是时间序列预测算法，这就需要保证我们的数据具有连续性，如果存在一些数据丢失的情况下，我们就需要采取一定的措施进行数据的修复，否则无法用来进行模型的训练。主要采用的方法可以是简单的采用和前一个数据相同的数据来填补。视频服务的属性特征有的并不是数字化数据，需要进行规范化到[0~1]区间，方便进行计算。



图4-5 预测模块架构图

（2）模式匹配：数据库中数据不仅包含容器和宿主机的资源信息，还包含CIVU中所处理视频服务的视频特征（任务类型、码率等）。本模块不是对整个平台的流量或负载进行预测，而是针对于单个的容器根据其历史负载情况来预测其未来的负载，然后以便对该容器进行资源的动态伸缩。所以其预测流程就分为两个阶段，在初始创建容器时根据服务相似性预测算法进行预测，在运行一段时间后根据最近邻回归算法进行预测。所以根据阶段不同，从数据库中所取的数据也不相同。

### 4.3.4 云资源管理模块

云资源管理服务是整个云平台的核心服务，虽然对容器来说可以弹性的利用系统的资源，但是我们在使用的时候推荐是要限制容器的CPU、内存等使用量，否则有可能造成应用之间的资源竞争，甚至会出现某个应用会占用大量的资源从而导致其它应用“饥饿”的现象。所以我们在使用过程中对资源进行限制，这样就会使容器之间相互隔离不同实例之间是互不干扰的。我们在上面介绍过固额分配会出现的一些问题，所以在本模块我们采用基于预测模块所获得预测值对容器进行创建和在运行中进行资源的重配置。目前针对Docker集群的资源管理有很多开源项目，我们这里采用Docker原生支持的Swarm，架构如图4-6所示，我们主要对其默认的调度策略进行了修改。

为了解决资源分配过量或分配不足的问题，目前大部分实现弹性伸缩的解决方案都是进行水平伸缩。但是水平伸缩主要是以虚拟机或容器为调节单位，是一种粗粒度的调整方式，主要是通过在水平方向上进行增加或减少虚拟机数量，通过虚拟机迁移及负载均衡等策略来进行资源的调整。水平伸缩的方式容易造成资源的浪费，并且这种方式适合于无状态的视频应用，对于有状态的服务来说，在程序为用户提供服务的过程中根据负载进行资源调整的最好方式还是垂直伸缩。垂直伸缩相对水平伸缩而言是一种细粒度的资源调整，并且可以在服务不间断的过程中对服务的资源需求进行动态调整。所以我们在资源重配置时采用垂直伸缩的方式，主要是结合资源调度、容器热迁移和Docker Deamon所提供的 REST API update接口实现。



图4-6 Swarm 架构图

从图4-7资源重配置的流程图中可以看出资源重配置过程不是一次性的，而是一个周而复始的过程。其资源管理的大致流程如下：

（1）如果是对新的服务请求进行资源初始分配，那么服务端API接收服务请求后会调用预测单元，根据数据库DB中的历史数据得到预测结果。云资源管理模块则根据资源预测单元所生成的预测结果，通过调度策略选择一台合适的宿主机，然后在其上创建容器并为之分配资源以提供服务，该容器也被纳入监控范围。

（2）如果是对持续运行的容器进行资源重配置，资源统计分析模块会根据资源监控模块获取的当前宿主机、该容器的资源使用量及资源预测模块获得的预测值进行计算分析。根据分析结果判断是需要对容器进行资源扩展还是资源收缩，如果是资源收缩则直接在本宿主机上进行动态调整即可，这样可以减少迁移次数，降低资源损耗。如果是需要扩展资源，那就需要根据调度策略选择合适的宿主机进行热迁移，热迁移后再对资源进行调整。因为对资源的申请如果分配不当容易造成“木桶效应”，即有的资源剩余过多有的不足以分配，这会造成资源的浪费。因此，我们对资源的扩展需求先根据调度策略选择合适的宿主节点，而对资源的收缩需求只需在本地进行即可。资源的扩展及收缩这两种情况可以交给资源配置策略进行处理，策略的执行单元负责容器的热迁移以及调用Docker Daemon的remote API进行容器的配置修改。



图4-7 基于预测的动态伸缩

（3）在资源的动态调整过程中不仅依靠预测模块提供的预测值，还与负载告警触发相结合。可以设置负载调整阈值，如果根据预测结果调整资源之后，负载超过了当前资源量或超过设置的阈值，则进行被动伸缩，以防违背SLA协议，我们宁愿牺牲一点资源换取稳定的性能。

## 4.4 基于预测的云资源配置方法

对于资源调度来说，无论是OpenStack中的调度算法还是Docker中自带的调度策略都是只考虑了单一资源的使用量或随机性，而没有考虑资源的多维性及视频服务的资源倾向性等特点。因此，在本文中我们根据预测算法得到的预测值利用主导资源优先的调度算法来对容器的资源量进行动态伸缩。

### 4.4.1 问题描述

因为资源存在多维性及异构性的因素，所以在资源分配过程中很难将一个服务器中的资源完全分配出去，必然会存在一些资源碎片，如某种资源剩余太少或某种资源完全分配出去，这样就无法创建新的容器，剩下的资源只能备着用来对运行着的容器进行相应的扩展。要想提高整个集群的资源利用率，就要尽量减少资源碎片并且碎片尽量小。

资源的配置是个动态的过程，同样资源的剩余量也是动态变化的。传统的资源分配策略中，比较多的是依靠服务器的资源剩余量或资源使用率来进行选择宿主机，这样就没有考虑资源的多维性等特点。可以从全局的角度对资源分配进行决策而不是仅局限于其在单一服务器内部的资源剩余比例，而是应该考虑其在集群剩余资源中所占的比例。下面我们所要使用的主导资源优先分配算法（DRFA）[51]就解决了这种问题。

为了模拟视频监控云平台中的资源分配问题，我们假设在异构物理服务器中存在*m*个异构物理服务器所组成的集合 资源配置问题实际上就是服务的资源请求集合和物理服务器集群P之间的多维映射关系。假设服务器中的资源有*n*种，可以描述为,其中表示CPU、内存或带宽等。资源请求中的V中任意元素代表容器根据上述预测结果所创建或伸缩重新调整资源量的请求，可以表示为，其中表示服务在未来一段时间内所需要的资源的资源量。同时物理机也可以用多维向量表示,其中表示服务器所拥有的资源数量，如果将的请求映射到上，那么此时的资源量就变为

(4-1)

### 4.4.2 算法描述

主导资源是资源比例向量中数值最大的资源。服务器的主导资源能够代表服务器相对于其它资源的优越性，而服务请求的主导资源能够代表服务请求的热门资源。主导资源能够很好的把握服务器内资源的异构性。我们先描述一下算法中所用到的主导资源的求解过程。

首先，在服务器集群中，汇总每一台服务器上的某一种资源剩余量，得到某资源的剩余总量，可通过累加方式计算，计算公式如4-2所示

(4-2)

通过同样的计算方式将其它每种资源的剩余总量计算出来，就可以得到所有资源的剩余总量，可以表示为.

接下来计算容器对某种资源的请求在该资源剩余总量中所占的比例，可通过公式（4-3）计算得出

(4-3)

容器整体资源请求比例用向量表示为。

同理，可以计算出服务器中资源剩余比例向量，其中

(4-4)

根据主导资源的定义，我们就可以分别计算出容器和宿主机的主导资源。其中容器的主导资源可以表示为

(4-5)

宿主机的主导资源表示为

(4-6)

下面介绍基于资源预测模型改进的主导资源优先调度算法，如算法3-3所示。流程可以总结为：

（1）每次会根据先入先出（FIFO）的原则从视频微服务请求队列中取出任务*task*，计算*task*的主导资源。

（2）然后遍历服务器集合pmList，判断其上的资源剩余量是否满足*task*的请求，如果满足则计算出服务器*pm*的主导资源，如果两者主导资源相同就将该pm加入到候选列表中，如果不满足就过滤掉。

（3）如果候选集合为空，则随机选取一台满足请求的宿主机映射，如果不为空，那么计算集合中的每个宿主机和task的匹配距离，通过欧式距离计算

(4-7)

（4）选择具有最小距离的*pm*进行映射。

|  |
| --- |
| **算法3-3 ：基于资源预测模型改进的主导资源优先分配算法** |
| **输入**：服务器集群中可用的宿主机列表pmList，任务队列taskQueue，里面包含每个视频服务通过资源预测算法得到的资源需求预测值。  **输出**：按照预测结果调度后的最佳宿主机  1 while taskQueue不空 do  2 task← taskQueue.get(); //按照FIFO原则  3 r ← argmax(task.ProportionalVector)；  4 candidateSet ← ∅；  5 for ∀pm ∈ pmList do //遍历服务器集合，找到满足条件的候选集合  6 if pm meets task then  7 p ← argmax(pm.ProportionalVector)；  8 if r equals to p then  9 candidateSet = candidateSet ∪ pm；  10 end if  11 end if  12 end for  13 candidate ← ∅；  14 if candidateSet不为空 then  15 minDistance ← ∞;  16 for ∀pm ∈ candidateSet do //遍历候选集合，找到最佳宿主机  17 distance ← DISTANCE(pm, task)；  18 if minDistance > distance then  19 minDistance ← distance；  20 candidate ← pm；  21 end if  22 end for  23 else  24 candidate ← Random(task, pmList)；  25 end if  26 candidate allocates resource to task;//给任务分配资源的最佳宿主机  27 end for |

## 4.5 本章小结

本章首先总结了当前视频监控云平台架构及存在的一些不足之处，然后对基于Docker的视频监控云平台的架构进行了优化介绍，接着对视频监控云平台的工作流程分为三个阶段进行了描述。本章最后给出了几个主要模块的设计思路，下文会给出模块的具体实现及功能测试。

# 第五章 系统实现与测试分析

## 5.1 系统硬件配置和软件配置

### 5.1.1 系统硬件配置

实验室共有12台服务器，为了保证集群的HA，控制节点设置为3个，分别命名为controller1、controller2和controller3，剩下的几个节点作为计算节点，命名为node1~node9。另外还有3台云存储服务器用于存储视频元数据和微服务保存的数据，每台服务器都安装了36T的机械硬盘，集群通过NFS的方式挂载到计算服务器中。还有一些大华的NVR及网络摄像头，用于采集视频数据以供实验。本系统平台所使用硬件的具体配置见表5-1所示。

表5-1 硬件配置列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 类型 | 厂家 | 产品参数 | 数量 |
| 1 | 2U机架式服务器 | 浪潮 NF5260M3 | CPU:32cores,2.6GHZ  内存:32GB | 2 |
| 2 | 1U机架式服务器 | DELL | CPU:32cores,2.6GHZ  内存:32GB | 3 |
| 3 | 2U机架式服务器 | DELL PowerEdge R720 | CPU:32cores,2.6GHZ  内存:32GB | 1 |
| 4 | 塔式服务器 | 浪潮NF5588M3 | CPU:32cores,2.6GHZ  内存:32GB | 2 |
| f5 | 塔式服务器 | DELL PowerEdge T630 | CPU:24cores, 2.4GHZ  内存:32GB | 4 |
| 6 | 云存储服务器 | UIT UFS | 存储空间：36T | 3 |
| 7 | NVR | 大华 |  | 2 |
| 8 | 网络摄像头 | 大华 |  | 12 |

### 5.1.2 软件配置

因为Docker只能安装在64位的Linux操作系统上并且内核版本要求3.10以上，当然也可用通过虚拟化的桌面工具安装在Windows或MacOS上，在实验室环境中我们在服务器上统一安装了Ubuntu 14.04.4 (LTS) server amd64版本的操作系统,内核版本为3.13.0-96-generic。并在所有的服务器上安装版本为1.11.1版本的Docker Engine。系统平台的开发是在实验室的电脑上进行，具体的软件开发环境见表5-2。

表5-2 系统平台开发环境

|  |  |
| --- | --- |
| 软件 | 版本 |
| 开发环境 | Windows 10、Java JDK7 |
| 运行环境 | Ubuntu 14.04.4 LTS Server、Docker |
| 开发IDE | Eclipse Luna |
| Web服务器 | tomcat 7.0 |
| 数据库 | Mysql |
| 其他软件 | Vim、g++、Spring Framework4.0、Maven |

## 5.2 系统功能实现

Docker集群的安装按照Docker官网[39]上的步骤操作即可，这里不再赘述。下面我们介绍一下几个主要模块的实现。

### 5.2.1 资源监控模块实现

为了收集容器的资源信息，我们可以使用Docker Deamon所提供的Remote API，如通过RESTFul风格的接口获取容器列表，

|  |
| --- |
| **GET** /containers/json?all=1 HTTP/1.1  ArrayList<String> getContainerList(){ //获取容器列表ID集合  endpoint = <http://10.103.242.128:2377/>  result = request.get(endpoint+“/containers/json?all=1”)；//获得以Json封装的容器列表信息  for(int i=0; i<result.size(); i++) //保存列表中容器的ID  arraylist.add(result[i].ID);  return arraylist；//返回容器ID集合  } |

该模块的主要是先获取集群中的容器列表，扫描容器集合，对每个新生成的容器生成一个线程对其进行监控，该线程主要的任务是收集容器的基本信息，如所使用的镜像、所在宿主机、容器名等。然后通过调用stats接口，来获取容器的资源使用信息、包括CPU、内存、I/O等信息。每个线程设定一个收集周期，每隔固定的时间收集一次，然后将收集到的信息存放到mysql数据库中。所用到的主要方法如表5-3所示。

表5-3 资源监控所需方法列表

|  |  |  |  |
| --- | --- | --- | --- |
| 方法名 | 参数 | 返回类型 | 描述 |
| getContainerList | null | ArrayList<String> | 用来获取Container列表,返回ID集合 |
| getContanerInfo | String containerId | Container | 获取一个容器的一些详细信息,如所用镜像、容器名等，参数为容器的ID |
| getContainerResource | String containerId | Container | 获取容器资源使用信息，参数为容器的ID |
| newThread | String containerId,  long time | InstanceThread | 每个新容器生成一个线程监控，参数为容器ID和线程执行周期 |

程序主要流程如下：

|  |
| --- |
| ArrayList<String> list = getContainerList();  for(int i=0; i<list.size(); i++){  if(新生成的容器){  Container container = getContainerInfo(list.get(i));  if(container.state==“running”){ //运行态的容器  newThread(container.Id, time).run(){//为每个容器创建一个监控线程  container = getContainerInfo(containerId);  container = getContainerResource(containerId);  insert(container);  }  }  }  } |

将上述监控的资源信息等存储在开源的mysql数据库中，为每个容器单独创建一个表，以每个容器的ID命名，这样可以提高查询效率，为了避免整个表过大，我们可以设置一个数据存活的期限。下面是创建container-id表的语句，表中各字段的含义如表5-4所示。

|  |
| --- |
| CREATE TABLE container-id(  time data not null,  name varchar(40),  node varchar(40),  service\_name varchar(40),  memory\_percent varchar(40),  cpu\_percent varchar(40)，  ……  network\_tx\_bytes, varchar(40)  )ENGINE=MYISAM DEFAULT CHARSET=UTF8; |

表5-4 container-id表

|  |  |  |
| --- | --- | --- |
| 字段名 | 字段类型 | 描述 |
| time | data | 信息收集时间 |
| name | varchar(40) | 容器别名 |
| node | varchar(40) | 所在节点 |
| memory\_ percent | varchar(40) | 内存利用率 |
| cpu\_ percent | varchar(40) | CPU利用率 |
| …… | …… | …… |
| network\_tx\_bytes | varchar(40) | 上行流量 |

### 5.2.2 资源预测及管理模块实现

该模块主要是对TSNNR算法的实现，图5-1为上节中监控模块所采集的容器的数据表，我们主要的工作是对这个表中的数据进行预处理，然后通过对这些历史数据进行分析建立预测模型，来预测未来一段时间的资源需求量。

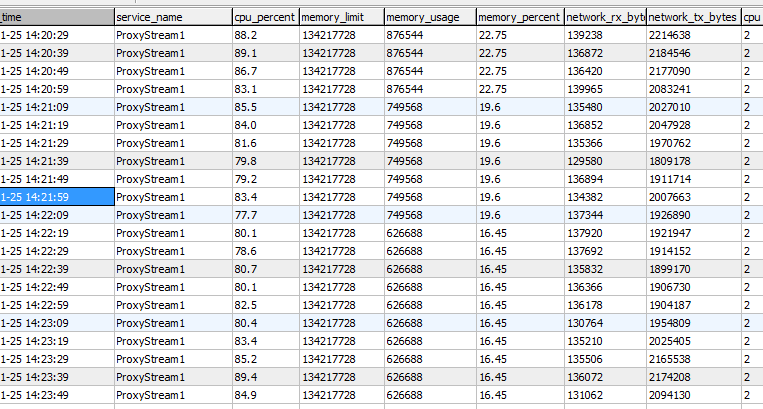


图5-1 容器数据信息采集表

先对数据进行预处理，因为可能含有缺失数据，所以需要对数据进行清洗后才可以作为预测模型的训练集。

|  |
| --- |
| public ArrayList<Resource> restore(ArrayList<Resource> queue){  ArrayList<Resource> lists = new ArrayList<Resource>();  lists.add(queue.get(0));  for(int i=1; i<queue.size(); i++){  if(queue.get(i)==null)  lists.add(queue.get(i-1);//如果存在缺失数据，用前一个数据的值填充  else  lists.add(queue.get(i));  }  return lists;  } |

数据清洗后，下一步就是对数据进行划分时间序列，然后查找相似的时间序列进行求值计算。

|  |
| --- |
| LinkedList<ArrayList<Resource>> init(int n)；//按n个数据为周期划分时间序列数据 |
| //往前预测一步  public static ArrayList<Resource> predict(LinkedList<ArrayList<Resource>> queue, int q, int k, int p)；  // 往前预测n段  public static int predictN(LinkedList<ArrayList<Resource>> queue, int n) {  for(int i=0; i<n; i++){  ArrayList<Resource> plist = predict(queue,q,k,p);  queue.addLast(plist); //将预测结果作为历史数据  }  return max(plist);  } |

对于资源的伸缩可以将新的分配资源封装成如下所示的JSON数据，然后调用Docker Deam的Remote API进行修改。

|  |
| --- |
| {  "BlkioWeight": 300,  "CpuShares": 512,  "Memory": 314572800,  ……  "MemorySwap": 514288000,  "MemoryReservation": 209715200,  "KernelMemory": 52428800,  } |

然后通过下面的REST API传递参数进行伸缩

|  |
| --- |
| POST /containers/${containerId}/update |

对于资源扩展我们还要执行容器调度，选择合适的宿主机进行热迁移，然后在进行资源伸缩。

|  |
| --- |
| /\*\*  \*传入的参数是预测模块传过来的资源预测值及主机集合  \*@return 返回最佳调度节点  \*/  public Node scheduling（double[] predictValue, ArrayList<Node> pmList）;  /\*\*  \*传入参数包括要迁移容器的ID，目的节点  \*@return 成功返回true，失败返回false  \*/  public boolean migration(Long containerId, Node destNode); |

对于资源伸缩过程的容器热迁移我们通过CRIU实现。CRIU主要由两个过程实现，checkpoint和restore。Checkpoint阶段负责保存容器进程的运行状态，然后增量备份根文件目录并将文件同步到目的主机。在restore阶段，是负责恢复容器根文件系统，利用Libnetwork恢复网络、Volume Driver恢复数据卷，最后恢复容器的进程。图5-2为checkpoint和restore的原理图。



图5-2 基于CRIU的热迁移

### 5.2.3 典型视频微服务实现

在视频监控云平台中，微服务主要是将所开发的模块打包可运行文件，如java的jar包，通过Dockerfile将应用程序制作成基础镜像并上传到镜像仓库中。当用户请求服务时，通过查找相应功能的镜像进行容器的创建，就可以为用户提供服务了。服务之间通过暴露出REST API进行交互。本文依托实验室已实现的视频处理算法来构造视频微服务镜像。

本实验室中视频浓缩的源码是通过Maven进行构建的，因此需要使用Maven来自动的构建源码。我们首先在Maven项目的根目录下创建一个Dockerfile文件，然后以Maven的官方镜像作为基础镜像，拷贝相关文件到镜像中，然后使用Maven package进行编译打包并用test进行测试。在镜像构建过程中，我们要注意将编译中间生成的文件删掉以减小镜像的大小。Dockerfile的代码如下：

|  |
| --- |
| FROM maven:3  MAINTAINER jiang [zhe.jiang90@gmail.com](mailto:zhe.jiang90@gmail.com)  RUN mkdir –p /build /input /output  WORKDIR /build  ENV TASK synopsis.jar  ADD pom.xml .  ADD src src  RUN mvn package && mvn test  RUN cp target/$TASK / && rm –rf /build && rm –rf ~/.m2/\*  VOLUME /output  CMD [“java”, “-jar”, “/synopsis.jar”, “$@”] |

Dockerfile编写完成就是制作镜像并上传到镜像仓库，可通过下面的命令进行：

|  |
| --- |
| # docker build –t controller1:5000/synopsis . //通过当前目录下的Dockerfile生成镜像为controller1:5000/synopsis,其中controller1:5000为视频服务的仓库地址。  # docker run –ti –rm controller1:5000/synopsis //运行测试看看制作的镜像是否可用  # docker push controller1:5000/synopsis //将镜像上传到镜像仓库 |

### 5.2.4 页面实现

本文对实验室现有的视频监控云平台进行了优化，并实现了相应的视频监控系统运行在云平台中，主要页面展示如下：

（1）服务请求界面

当用户在浏览器中输入地址http://10.103.242.97:8080/videocenter/ 时，系统就会跳转到登录界面，登录成功会调到如图5-3所示的服务请求页面。

在该页面中用户点击摄像头列表中的相应的通道，就会根据年月对各个视频通道的视频文件进行获取并显示（包括视频文件大小、文件名、视频的时间戳等）。用户可以批量选择特定的多个视频文件发起视频浓缩请求，一旦请求发起，后端的资源预测模块就会根据服务类型、码率等进行资源预测，然后根据预测结果调用swarm远程接口发起创建服务的请求，并根据优化的调度策略选择宿主机生成容器。

（2）任务处理界面

容器生成后，用户可以实时查看当前各个视频任务的处理进度，如图5-4所示，其中每一个任务底层由特定的容器实例负责运行，一旦任务处理结束、容器会自行销毁，swarm负责进行资源的回收。处理的结果被存放在存储服务器中，可以在线下载到你的本地。

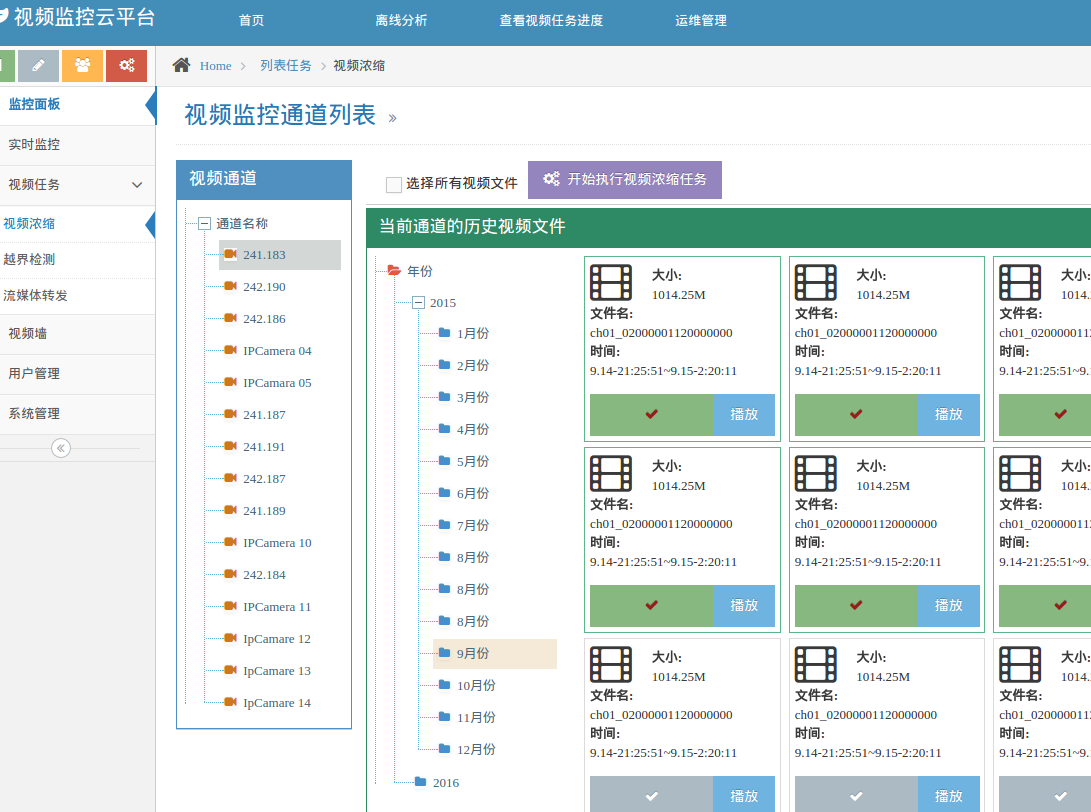


图5-3 服务请求页面



图5-4 任务处理列表

（3）容器列表

管理员用户可以通过在浏览器中输入地址[http://10.103.242.97:8080/ dockerVI/](http://10.103.242.97:8080/%20dockerVI/) 在登录界面输入用户名和密码登录到后台的管理界面。在后台的管理系统中，管理员可以对镜像、容器进行管理，管理员可以查看正在为用户提供服务的容器列表，如图5-5所示。



图5-5 容器列表

用户通过点击查看详情，就可以看到这个容器相关的一些详细信息，包括这个容器的资源配置、可视化的资源负载图、及服务的运行日志等，如图5-6.

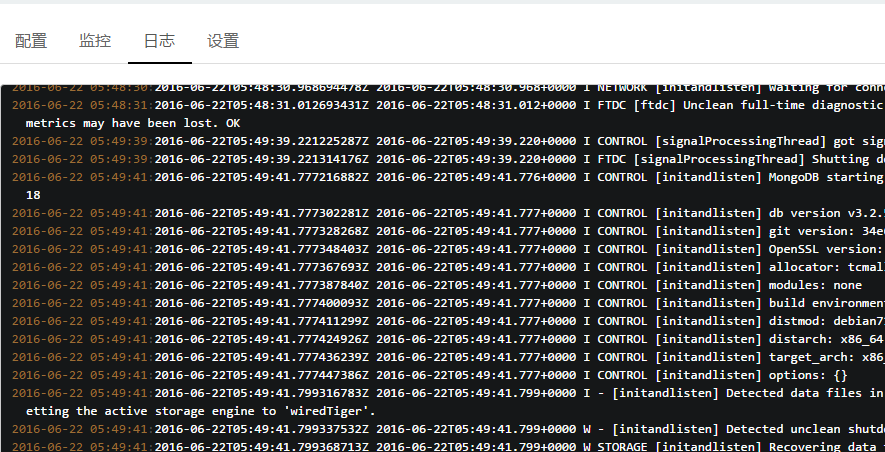


图5-6 容器运行日志

通过我们构建的云视频监控系统及云平台管理系统的良好运行，我们可以得出本文针对基于Docker的视频监控云平台结合了资源预测算法及资源的动态伸缩功能所做的优化是具有实用性的。并且，较之一般的视频监控云平台更有意义。

## 5.3 资源预测与配置方法性能测试

### 5.3.1 资源预测方法性能测试

在第三章我们以提高资源利用率为目标提出了时间序列最近邻回归算法（TSNNR），并根据资源预测值进行动态伸缩。通过对算法的分析可以知道，TSNNR算法主要是基于容器的历史资源使用量进行未来资源预测，我们通过从数据库中获取某一容器的历史数据进行对比实验。

在资源预测算法的对比试验中，我们对不同的视频微服务进行了测试。以视频转码微服务为例，我们选取了前100个小时的资源使用量进行实验，其中前10个小时作为初始训练集。我们将时间序列按10分钟为一个周期进行划分成一个个的子时间序列，TSNNR算法中的三个参数我们设置为了k=5,q=1,p=3(k=5代表找的5个最相邻序列数，q=1 表示我们使用当前时间后面的1个子时间序列作为一个比较序列，p=3表示以3个时间片数据作为预测周期)。在对比实验上本文选取了目前经典的运用较广的ARIMA算法以及神经网络（Neural network）预测方法进行对比。我们的实验对比结果如图5-7所示。

图5-7中蓝色的曲线代表实际的负载趋势，图（a）中黑色虚线代表了利用TSNNR算法所得到的预测趋势，图（b）中黑色虚线表示利用ARIMA算法得到的预测趋势，图（c）中黑色虚线表示通过神经网络模型得到的预测趋势。上图是对CPU进行预测的对比图，通过负载的趋势图来看这三种预测算法都和实际的负载趋势基本重合，说明TSNNR算法、ARIMA算法及神经网络模型对实际的预测都是比较准确的。我们以平均百分比绝对误差来作为衡量准确性的尺度，平均百分比绝对误差(mean absolute percentage error)的计算方式如公式（5-1）。

(5-1)

其中表示真实负载值，表示预测负载值。通过计算我们得到TSNNR的MAPE=7.06%，而通过ARIMA算法所得到的MAPE=8.71%，神经网络方法得到的预测结果其MAPE=8.13%，从数据我们可以看到针对视频微服务的负载预测而言，我们所提出的TSNNR算法要比ARIMA算法及神经网络模型略好。并且比较二者的算法复杂度发现，ARIMA的算法复杂度为O(*n*2)，TSNNR的算法复杂度为O(*n*)，*n*均为样本集的大小。而神经网络需要的样本量大，计算耗时多。通过比较我们可以总结出来我们的算法具有更小的复杂度、更有实时性并且具有更高的精确度。

图5-7 TSNNR、ARIMA及神经网络预测算法对比图

### 5.3.2 资源配置方法性能测试

资源的配置涉及到容器的热迁移，所以要考虑热迁移的性能，如图5-9所示，对容器进行热迁移所花费的总时间为9.9s左右，迁移过程中不可用时间仅为2秒左右，相比于虚拟机动辄十几分钟的时间损耗完全在可接受的范围内，在这个过程中用户完全感受不到服务的变化。并且迁移完成后容器可以正常持续的执行任务如图5-8所示。从而证明本文所提的资源配置方式是可行的。

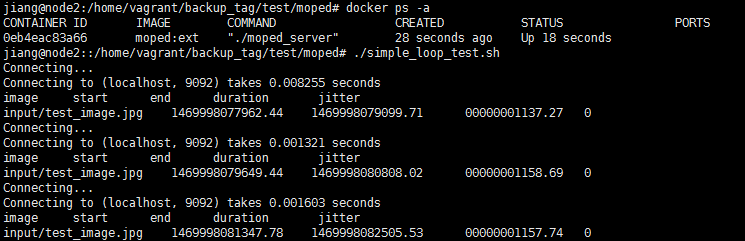


图5-8 容器热迁移后的状态

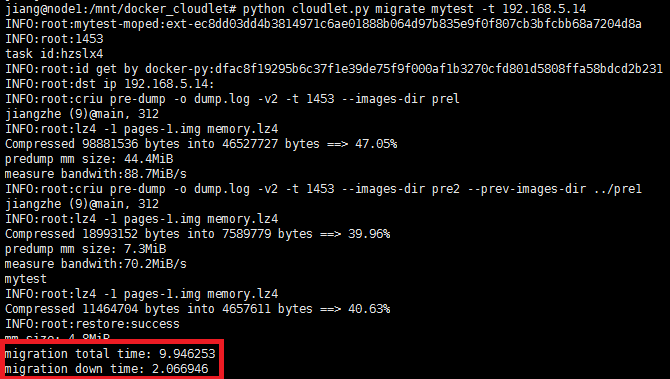


图5-9 容器热迁移

容器的垂直伸缩是毫秒级的，从图5-10中可以看出容器mytest在伸缩前后的资源使用状况，伸缩前CPU、内存的利用率都在50%左右，根据预测结果对其下一周期进行资源重配置后，可以看出CPU和内存的利用率都达到90%，这说明资源被充分利用，基于预测值的资源动态配置是可行的。

C:\Users\bupt911No2\Desktop\毕业论文\4.png



图5-10 容器资源伸缩前后对比图（上图为伸缩前，下图为伸缩后）

图5-11 基于预测模型的平台资源利用率

如图5-11所示，图中PDRFA代表采用基于本文预测模型改进的主导资源优先调度算法的资源动态配置方式，PFS和PS所表示的资源配置方法分别采用了基于预测模型改进型的FilterScheduler和Spread调度算法，FA表示采用固额分配被动伸缩的资源配置方式。在实验中我们分别对不同类型的服务请求提供服务，所采用的物理集群见表5-1所示，其中固额分配中我们设定的配额为2cores、4GB，重复10次试验后所得到的平均试验结果如图5-11所示。从图中可以看出相对于初始固额分配在运行中被动伸缩的方式，我们所采用的基于预测模型进行主动资源调整的资源配置方式使云平台拥有更高的资源利用率，并且本文采用的PDRFA更有优势。通过这两个实验数据可以看出基于本文所提出预测模型的资源配置方式能有效提高云平台的资源利用率。

## 5.4 本章小结

本章主要对设计的视频监控云平台的主要模块（监控模块、预测模块）进行了编码实现，并且搭建了一个简单的视频监控系统运行在云管理平台上，验证了平台的实用性。最后对第三章中所提的资源预测算法TSNNR进行了对比试验，通过实验结果表明该算法能够准确的预测未来一段时间的资源需求量，对系统的动弹伸缩具有指导意义。

# 第六章 总结与展望

## 6.1 总结

随着人们对视频监控系统需求的不断提升，视频监控系统与云计算融合所形成的视频监控云也成了近年来的研究热点。将底层异构的资源进行整合管理是视频监控云平台的一大特色，但是云平台中的资源毕竟是有限的，传统的云平台中对虚拟机的分配都是采取固额分配，通过监控负载告警实现被动伸缩，这导致云平台中资源浪费严重并且可能影响服务质量。如何提高视频监控云平台的资源利用率，实现一个高效的可以动态弹性伸缩的视频监控云平台是我们的研究重点。围绕这一目标，我们开展了以下几个方面的工作：

（1）基于服务属性特征及时间序列的资源预测模型构建。

当前主流的视频监控云平台都是通过生成虚拟机并在上面部署视频应用程序来为用户提供服务。然而对虚拟机分配资源的策略基本是分配一个固定的额度，在运行期间很难及时动态调整，因为视频监控的负载具有波动性，所以这种分配方式不能很好的适应负载的变化，有可能出现资源浪费或资源分配不足的情况。为了解决这种问题，就要求我们要能对负载的趋势变化进行有效的预测，并在负载变化之前做好准备，以防止资源严重浪费或对应用运行造成影响。为此我们通过分析监控视频的负载特性，提出了一种基于视频服务特征相似性和时间序列最近邻回归TSNNR相结合的模型，可以有效的对视频监控服务的未来一段时间的负载进行预测，为资源的及时伸缩提供依据。

（2）基于预测模型的细粒度资源重配置。

目前，云平台在资源伸缩方面的研究和应用都是通过水平伸缩这种方式，就是通过在水平层次上相应的增加或减少虚拟机的数量来控制负载均衡，这种调节是以虚拟机为调整粒度的，容易造成资源浪费，并且对有状态是视频服务来说是不适用的。所以我们的研究侧重通过资源预测结果来进行垂直伸缩，并考虑服务需求和物理资源的多维、异构等特性，通过优化的资源调度策略选择最优的宿主机进行动态迁移然后进行资源重配置，通过这种方式可以提高云平台的资源利用率。

（3）基于Docker技术的视频监控云平台的优化与实现

传统的视频监控云平台都是以虚拟机的方式提供服务，资源分配粒度大并且有一定的性能损耗等不足之处。基于Docker的云平台解决了上述缺陷，因此本文在实验室现有的视频监控云平台的基础上添加了资源预测模块并对资源配置模块进行了优化，经过验证优化后的平台能有效提高资源利用率。

## 6.2 展望

本文通过分析传统视频监控云平台中的不足之处，结合视频服务和资源的一些特性，提出了面向视频服务的云资源预测算法以及通过预测结果进行资源的重配置研究，最后实现了基于Docker的视频监控云平台的优化工作。经过测试，预测算法及平台功能等都达到了预期的效果，具备一定的实用性。但是本文还认为存在一些可以进一步研究的地方：

（1）垂直伸缩和水平伸缩结合。在本文中我们关注的是对不适于水平伸缩的有状态服务逐个进行垂直伸缩以提高资源利用率。并没有考虑那种无状态服务（如流媒体转发），在用户高流量请求的情况下是进行垂直伸缩还是通过负载均衡进行水平伸缩，亦或两者结合的方式，这是未来的一个研究点。

（2）云资源长期预测。本文所提出的时间序列最近邻回归TSNNR应该是针对单一的视频服务进行短期的预测。而没有对整个平台的网络流量进行负载预测，可以认为网络流量越高平台的负载越高，可以结合季节、节假日周期进行长期的云平台负载预测，使短期预测和长期的预测相结合。

（3）平台自适应负载均衡。本文中所采用的基于主导资源优先调度的策略考虑到资源的多维异构等特点，能很好的提高资源利用率。但是随着容器的创建销毁，平台会出现极度不均衡状况，我们只是在容器创建及资源调整时可以调整集群的负载，可以通过监控集群负载情况，当出现负载不均衡状况时，可以自动的利用热迁移技术调整集群负载。

# 参考文献

1. Zhao X M, Ma H D, Zhang H T, et al. Metadata extraction and correction for large-scale traffic surveillance videos[C]. // 2014 IEEE International Conference on Big Data. IEEE, 2014:412-420.
2. Feris R S, Siddiquie B, Petterson J, et al. Large-Scale Vehicle Detection, Indexing, and Search in Urban Surveillance Videos[J]. IEEE Transactions on Multimedia, 2012, 14(1):28-42.
3. Xiong Y H, Wan S Y, He Y, et al. Design and implementation of a prototype cloud video surveillance system[J]. Journal of Advanced Computational Intelligence & Intelligent Informatics, 2014, 18(1):40-47.
4. Zhao X M, Ma H D, Zhang H T, et al. HVPI: Extending Hadoop to Support Video Analytic Applications[C]. // IEEE International Conference on Cloud Computing. IEEE, 2015:789-796.
5. Yang X D, Zhang H T, Ma H D, et al. Multi-resource Allocation for Virtual Machine Placement in Video Surveillance Cloud[C]. //International Conference on Human Centered Computing(HCC2016). Springer, 2016: 544-555.
6. Chaisiri S, Lee B S, Niyato D. Optimization of resource provisioning cost in cloud computing[J]. IEEE Transactions on Services Computing, 2012, 5(2): 164-177.
7. Neal D, Rahman S M. Video surveillance in the cloud-computing?[C].//2012 7th International Conference on Electrical & Computer Engineering (ICECE), IEEE, 2012:58-61.
8. Zhao H, Pan M, Liu X, et al. Exploring fine-grained resource rental planning in cloud computing[J]. IEEE Transactions on Cloud Computing, 2015, 3(3): 304-317.
9. Hossain M S, Hassan M M, Al Qurishi M, et al. Resource allocation for service composition in cloud-based video surveillance platform[C]. // 2012 IEEE International Conference on Multimedia and Expo Workshops (ICMEW), 2012: 408-412.
10. Pahl C. Containerization and the PaaS cloud[J]. IEEE Cloud Computing, 2015 (3):24-31.
11. Li L, Tang T, Chou W. A REST Service Framework for Fine-Grained Resource Management in Container-Based Cloud[C]. //2015 IEEE 8th International Conference on Cloud Computing. IEEE, 2015: 645-652.
12. Joy A M. Performance comparison between linux containers and virtual machines[C]. //2015 International Conference on Advances in Computer Engineering and Applications (ICACEA), 2015: 342-346.
13. Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers[C]. //2015 IEEE International Symposium On Performance Analysis of Systems and Software (ISPASS), 2015: 171-172.
14. Bernstein D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE Cloud Computing, 2014, 1(3): 81-84.
15. 浙江大学SEL实验室. Docker:容器与容器云, 人民邮电出版社[M], 2015.
16. 仇臣. Docker容器的性能监控和日志服务的设计与实现[D]. 浙江大学, 2016.
17. Amazon EC2. <https://aws.amazon.com/ec2/>. accessed:12/2016.
18. Dockercon. <http://2016.dockercon.com/>. accessed:12/2016.
19. DockerCon 2016 Day1:Docker社区爆发增长. <https://yq.aliyun.com/articles/55866?spm=5176.8110234.375527.2.wK6B0u>. accessed:12/2016.
20. 阿里Docker化之路. <http://mt.sohu.com/20161130/n474583151.shtml>, accessed:12/2016.
21. Unikernel. <http://unikernel.org/> . accessed:12/2016.
22. 中国信息通信研究院. 云计算白皮书[J], 2016.
23. 刘晓艳, 王颖. 基于改进云模型的云计算负载预测[J]. 计算机应用研究, 2015, 32(10): 3124-3127.
24. Jia S, Xu X, Pang Y, et al. Similarity measurement based on cloud models for time series prediction[C]. //2016 Chinese Control and Decision Conference (CCDC), IEEE, 2016: 5138-5142.
25. Wu Y, Hwang K, Yuan Y, et al. Adaptive workload prediction of grid performance in confidence windows [J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21(7): 925-938.
26. Mallick S, Hains G, Deme C S. A resource prediction model for virtualization servers[C]. // 2012 International Conference on High Performance Computing and Simulation (HPCS), IEEE, 2012: 667-671.
27. Xiao Z, Song W, Chen Q. Dynamic resource allocation using virtual machines for cloud computing environment[J]. IEEE Transactions on parallel and distributed systems, 2013, 24(6): 1107-1117.
28. Tan Y, Gu X. On predictability of system anomalies in real world[C]. //2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, IEEE, 2010: 133-140.
29. 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, 2015, 3(4): 449-458.
30. Qiu F, Zhang B, Guo J. A deep learning approach for VM workload prediction in the cloud[C]. //2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), IEEE, 2016: 319-324.
31. Kousiouris G, Menychtas A, Kyriazis D, et al. Parametric design and performance analysis of a decoupled service-oriented prediction framework based on embedded numerical software[J]. IEEE Transactions on Services Computing, 2013, 6(4): 511-524.
32. Islam S, Keung J, Lee K, et al. Empirical prediction models for adaptive resource provisioning in the cloud[J]. Future Generation Computer Systems, 2012, 28(1): 155-162.
33. Shen Y. Virtual Resource Scheduling Prediction Based on a Support Vector Machine in Cloud Computing[C]. //2015 IEEE 8th International Symposium on Computational Intelligence and Design (ISCID), 2015, 1: 110-113.
34. Kalyvianaki E, Charalambous T, Hand S. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters[C].// International Conference on Autonomic Computing. ACM, 2009:117-126.
35. Kalman R E. A New Approach to Linear Filtering and Prediction Problems[J]. Transaction of the ASM&Journal of Based Engineering, 1960, 82(D) 35-45.
36. Li T, Wang J, Li W, et al. Load Prediction-Based Automatic Scaling Cloud Computing[C]. // 2016 International Conference on Networking and Network Applications (NaNA), IEEE, 2016: 330-335.
37. Zhang H T, Ma H D, Fu G P, et al. Container based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning[C]. // IEEE International Conference on Cloud Computing. IEEE, 2016: 758-765.
38. Lee W R, Teng H Y, Hwang R H. Optimization of cloud resource subscription policy[C]. //2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2012: 449-455.
39. Docker. <http://www.docker.com>. accessed: 12/2016.
40. 杨保华, 戴王建, 曹亚伦. Docker技术入门与实战. 机械工业出版社[M],2015.
41. Tchernykh A, Schwiegelshohn U, Yahyapour R, et al. On-line hierarchical job scheduling on grids with admissible allocation[J]. Journal of Scheduling, 2010, 13(5): 545-552.
42. Containers Live Migration: Behind the Scenes. // <https://www.infoq.com/articles/container-live-migration>, accessed: 12/2016.
43. Hines M R, Deshpande U, Gopalan K. Post-copy live migration of virtual machines[J]. ACM SIGOPS operating systems review, 2009, 43(3): 14-26.
44. 付广平，基于Docker技术的视频监控云服务管理平台研究与实现[D]. 北京邮电大学，2016.
45. Lewis J, Fowler M. Microservices[J]. 2014
46. Thönes J. Microservices[J]. IEEE Software, 2015, 32(1): 116-116.
47. Introduction to Microservices. [https://www.nginx.com/blog/introduction-to- microservices/](https://www.nginx.com/blog/introduction-to-%20microservices/), accessed: 12/2016.
48. 图解服务化架构演进. [http://ifeve.com/weixin-art/#more-24349](http://ifeve.com/weixin-art/" \l "more-24349), accessed: 12/2016.
49. 格蕾特, 李洪成. 时间序列预测实践教程[J]. 2012.
50. Box, George E P, Pierce, et al. Distribution of Residua] Autocorrelations in Autoregressive-Integrated Moving Average Time Series Models[J]. Journal of the American Statistical Association. 1970. 65(332): 1509-1526.
51. 杨贤达. 基于Openstack的视频监控云资源调度技术研究与应用[D], 北京邮电大学，2016.
52. 王庆波, 何乐. 虚拟化与云计算[M]. 电子工业出版社, 2009.
53. Dinda P A. The statistical properties of host load[J]. Scientific Programming, 1999, 7(3‐4): 211-229.
54. 杨 伟, 朱巧明, 李培峰, 等. 基于时间序列的服务器负载预测[J]. 计算机工程, 2006, 32(19): 143-145.
55. ITU-T H.626 standard, “Architectural requirements for visual of surveillance,” 2011.

# 致谢

时光荏苒，转眼之间两年半的研究生生涯即将结束，回首这两年多的时光，有欢声笑语也有艰辛困苦，收获更多的就是成长。马华东老师的严于律己、宽以待人的风范，认真负责、踏实勤恳的工作态度都潜移默化的影响着我，能在马老师的团队工作学习是一件非常荣幸的事。

在此，我要深深的感谢我的导师刘亮老师和张海涛老师，感谢刘老师和张老师这两年半的培养、教育和关怀。学业上刘老师和张老师的悉心指导使我的收获颇丰，勤奋能干的刘老师和温文儒雅的张老师对待学术的严谨态度及处事风范都深深打动了我。在临近毕业之际，毕业论文写作方面，刘老师和张老师也积极给予宝贵的建设性意见。在此，我谨向刘老师和张老师致以衷心的感谢！

感谢实验室一起工作学习的各位小伙伴，在这样一个温馨、朝气蓬勃、团结友爱的大家庭里，使我能够轻松快乐的学习和生活。感谢许彬、骆亮亮、赵梦琪同学，实验室整个项目的完成离不开大家的努力。其次感谢杨贤达、付广平学长在我初入实验室时给予的帮助，使我在科研的道路上轻松前行。同时感谢高阳阳师弟在项目开发过程中付出的努力。正是有了大家的共同努力，才营造了实验室良好的学习氛围，同时也带来了实验室的欢声笑语。同样要感谢我的室友安家琪、陈博、侯夏冰、齐恒、徐怀宇，是你们每天给我带来了欢声笑语，使我放松不至于负重前行，谢谢你们营造了一种温暖快乐的宿舍氛围。

最后，我将最诚挚的感激献给我的父母和姐姐，他们在生活和学业上给予我的鼓励和关心是我前进的最大动力。

# 攻读硕士学位期间的主要研究成果

[1] Haitao Zhang, Huadong Ma, Guangping Fu, Xianda Yang, **Zhe Jiang**, Yangyang Gao. Container based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning[C]. // IEEE International Conference on Cloud Computing. IEEE, 2016: 758-765.

[2] 曹宁, 胡豆豆, 张海涛, 姜哲. 基于OpenStack架构的视频监控服务平台[J]. 电信技术, 2015(12):51-55.