**密级： 保密期限：**

xm 拷贝

**硕士学位论文**



**题目：边缘计算平台中计算卸载与镜像缓存方法研究与实现**

**学 号： 2016110762**

**姓 名： 邹晟**

**专 业： 计算机科学与技术**

**导 师： 刘亮**

**学 院： 计算机学院**

**年 月 日**



**Thesis for Master Degree**

**Title: RESEARCH AND IMPLEMENTATION OF COMPUTATION OFFLOADING AND IMAGE CACHING IN EDGE COMPUTING PLATFORM**

**Student ID：**

**Candidate：**

**Major：**

**Supervisor：**

**Institute：**

**February 27th, 2019**

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

学位论文作者完全了解北京邮电大学有关保留和使用学位论文的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属北京邮电大学。学校有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许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，可以允许采用影印、缩印或其它复制手段保存、汇编学位论文。（保密的学位论文在解密后遵守此规定）

保密论文注释：本学位论文属于保密在年解密后适用本授权书。

非保密论文注释：本学位论文不属于保密范围，适用本授权书。

本人签名：日期：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

导师签名：日期：\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_

**边缘计算平台中计算卸载与镜像缓存方法研究与实现**

**摘 要**

近年来，随着信息时代的高速发展，海量的数据逐渐涌入人们的日常生活中，给传统的云计算模式带来了新的挑战。为此，边缘计算模式应运而生，与传统的云计算模式互补，形成了云-边一体化的计算模式。然而，由于边缘设备轻量化的缘故，导致其资源相对有限的现状，尤其体现在计算资源和存储资源上。因此，如何有效地解决边缘计算资源和存储资源不足的问题，成为边缘计算亟需解决的重要挑战。

对于计算资源不足的问题，本文采用计算卸载的思想，将本地边缘的计算任务选择性地卸载到中心云或者其余边缘上。从而提出了基于双边博弈的计算卸载方法，进而设计实现了边缘计算云平台的调度器模块。对于存储资源不足的问题，本文则采用缓存优化的思想，将云平台上的镜像资源进行缓存替换。从而提出了基于流行度预测的镜像缓存方法。其中，采用基于GRU的镜像流行度预测策略，得出流行度的预测结果。再以此结果作为依据，进行缓存的替换。进而设计实现了边缘计算云平台的缓存器模块。本文还在硬件平台上实现了一套基于Kubernetes的边缘计算系统，用以检验调度器和缓存器的有效性，并通过实际测试验证了它们的性能。

**关键词：**边缘计算 计算卸载 流行度预测 缓存替换

RESEARCH AND IMPLEMENTATION OF COMPUTATION OFFLOADING AND IMAGE CACHING IN EDGE COMPUTING PLATFORM

ABSTRACT

In recent years, with the rapid development of the information age, massive data has gradually poured into people's daily lives, bringing new challenges to the traditional cloud computing model. To this end, the edge computing model emerges and complements the traditional cloud computing model, forming a cloud-edge integrated computing model. However, due to the relatively limited edge resources caused by the light weighting of edge devices, it is especially reflected in computing resources and storage resources. Therefore, how to effectively solve the problem of insufficient computing resources and storage resources on the edge has become an important challenge that edge computing needs to solve.

For the problem of insufficient computing resources, this paper uses the idea of computation offloading to selectively offload the local edge computing tasks to the core cloud or the remaining edges. Therefore, a computation offloading method based on bilateral game is proposed, and then the scheduler module of the system is designed and implemented. For the problem of insufficient storage resources, this paper adopts the idea of cache optimization to cache the image resources on the cloud platform. Therefore, an image cache method based on popularity prediction is proposed. Among them, the GRU-based image popularity prediction strategy is used to obtain the prediction result of popularity. Therefore, based on this result, the replacement of the cache is performed. Furthermore, the buffer module of the system is designed and implemented. This paper also implements a Kubernetes-based edge computing system on the hardware platform, which verifies the effectiveness of the scheduler and the buffer, and understands their performance through actual testing.

**KEY WORDS:** edge computing; computation offloading; popularity prediction; cache replacement

**目 录**

[第一章 绪论 1](#_Toc2687089)

[1.1 研究背景和意义 1](#_Toc2687090)

[1.2 国内外研究现状 3](#_Toc2687091)

[1.3 论文的主要研究内容 4](#_Toc2687092)

[1.4 论文的组织结构 6](#_Toc2687093)

[第二章 相关技术简介 8](#_Toc2687094)

[2.1 边缘计算 8](#_Toc2687095)

[2.2 Docker 9](#_Toc2687096)

[2.3 Kubernetes 10](#_Toc2687097)

[2.4 RNN、LSTM与GRU 12](#_Toc2687098)

[2.5 本章小节 14](#_Toc2687099)

[第三章 基于双边博弈的计算卸载方法 15](#_Toc2687100)

[3.1 边缘计算卸载架构 15](#_Toc2687101)

[3.1.1 通信模型 15](#_Toc2687102)

[3.1.2 计算模型 16](#_Toc2687103)

[3.2 计算任务的划分 17](#_Toc2687104)

[3.3 边缘计算卸载方法的目标 17](#_Toc2687105)

[3.4 基于双边博弈的计算卸载方法设计 19](#_Toc2687106)

[3.4.1 集群聚合构建 19](#_Toc2687107)

[3.4.2 多对一稳定匹配博弈 20](#_Toc2687108)

[3.5 本章小结 22](#_Toc2687109)

[第四章 基于流行度预测的镜像缓存方法 23](#_Toc2687110)

[4.1 基于GRU的镜像流行度预测 23](#_Toc2687111)

[4.1.1 数据收集 23](#_Toc2687112)

[4.1.2 影响镜像流行度的因素 23](#_Toc2687113)

[4.1.3 数据的预处理及输入的特征选取 26](#_Toc2687114)

[4.1.4 流行度预测模型的训练 27](#_Toc2687115)

[4.2 基于流行度预测的镜像缓存更替策略 28](#_Toc2687116)

[4.3 本章小结 31](#_Toc2687117)

[第五章 系统设计与实现 32](#_Toc2687118)

[5.1 传统云计算平台介绍 32](#_Toc2687119)

[5.2 边缘计算平台总体架构 33](#_Toc2687120)

[5.2.1 基于Kubernetes 的边缘计算云平台架构设计 33](#_Toc2687121)

[5.2.2 Kubernetes Federation简介 35](#_Toc2687122)

[5.2.3 工作流程 35](#_Toc2687123)

[5.3 基础环境 36](#_Toc2687124)

[5.4 边缘平台的基础搭建 37](#_Toc2687125)

[5.4.1 边缘集群上Kubernetes的搭建 37](#_Toc2687126)

[5.4.2 负载均衡器的实现 38](#_Toc2687127)

[5.4.3 Kubernetes Federation的搭建 39](#_Toc2687128)

[5.5 边缘计算云平台调度器的设计与实现 39](#_Toc2687129)

[5.5.1 调度的工作流程 39](#_Toc2687130)

[5.5.2 调度器模块的设计 41](#_Toc2687131)

[5.5.3 调度器模块的实现 41](#_Toc2687132)

[5.6 边缘计算云平台缓存器的设计与实现 45](#_Toc2687133)

[5.6.1 数据爬虫子模块 45](#_Toc2687134)

[5.6.2 特征预处理子模块 46](#_Toc2687135)

[5.6.3 模型训练子模块 47](#_Toc2687136)

[5.6.4 缓存替换子模块 48](#_Toc2687137)

[5.7 本章小结 48](#_Toc2687138)

[第六章 实验分析与系统测试 49](#_Toc2687139)

[6.1 实验环境与数据集 49](#_Toc2687140)

[6.2 基于双边博弈的计算卸载方法的实验分析 50](#_Toc2687141)

[6.3 基于GRU的镜像流行度预测方法的实验分析 53](#_Toc2687142)

[6.4 基于流行度预测的镜像缓存替换策略的实验分析 55](#_Toc2687143)

[6.5 本章小结 58](#_Toc2687144)

[第七章 总结与展望 59](#_Toc2687145)

[7.1 本文总结 59](#_Toc2687146)

[7.2 工作展望 60](#_Toc2687147)

[参考文献 61](#_Toc2687148)

[致谢 62](#_Toc2687149)

[攻读学位期间取得的研究成果 63](#_Toc2687150)

1. 绪论
   1. 研究背景和意义

自云计算（Cloud Computing）[1] [2]提出以来，人们的生活、工作、学习等方式就逐渐开始改变。生活中经常用到的QQ、京东等软件提供的服务就是典型的应用场景；并且，可伸缩的基础设施和能够支持云服务的处理引擎也对商业模式产生了巨大的影响。从而，带动了如AWS，阿里云，百度云等云服务商的高速发展。

同时，随着即将迈入万物互联（Internet of everything, IoE）[3]时代，海量的数据开始涌入人们的日常生活。据思科[4]估计，到2021年将会有近500亿台无线设备连接到互联网，而全球数据中心总数据流量预计将达到20ZB的海量级别。

对此，传统的云计算模式难免存在一些问题[5]。例如，1）海量数据的增长导致传统集中式云计算已无法满足其需求，；2）数据传输的增加也造成了网络带宽的大量消耗；3）物联网应用可能会要求极快的响应时间，数据的私密性。诸如此类问题，使得万物互联时代海量数据的处理迫在眉睫。

针对上述问题，催生了边缘式大数据处理模式，即边缘计算模式[5]。边缘计算模式，是指使用网络的边缘结点去处理、分析数据，从而使得计算模型更加高效。例如，移动边缘计算[6]、雾计算[7]、Cloudlet[8]、海云计算[9]、微数据中心[1]。边缘计算模式的提出，并非需要取代现有的传统云计算模式，而是与其相结合，即云边协同，运用云边各自不同的性能特性，进行协同处理。

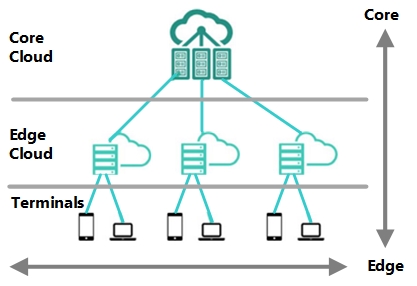


图1-1 云-边协同架构

如图1-1所示，对于边缘计算与传统云计算融合，即实现一个三层架构：中心云—边缘云—终端。第一层，终端，即智能终端。包括一些智能手机，传感器采集节点等。当为智能手机等自身具有数据处理能力的终端时，自身也能进行一部分的数据存储、处理，以及分析；当为采集节点时，则必须将数据传输到上层云进行下一步的数据处理。第二层，边缘云。为满足低时延的特性，边缘云一般部署在靠近终端的物理位置上。同时，边缘云为满足边缘云作为终端与中心云的中间层，将起到承上启下的作用，即与中心云和其余边缘云之间的任务协同。边缘云一般可为，一台至多台的小型服务器或PC，相比于传统的云中心，体现出边缘云的轻量化。第三层，中心云，即传统的云计算中心。一般而言，中心云拥有大型的服务器集群，并且对于用户终端一般时延高，带宽有限。同时，过多的用户申请服务，也使得中心云的计算负载过大。

基于此，考虑到边缘云的轻量化所带来的资源相对有限的现状，尤其是计算资源和存储资源。因此，以下两个问题也将不可避免地成为急需解决的问题。

1 ) 如何解决计算资源不足的问题。对于计算资源不足，不由联想到采用计算卸载的方式。通过将本地的任务传输给其余空闲的边缘云处理，有效地缓解边缘云间负载不均衡。然而，传统的边缘计算卸载，仅仅是存在于单一边缘与中心云之间的卸载。并且，过多的边缘云与中心云之间的卸载，也将会加重中心云的服务压力。与此同时，我们不难发现，边缘云之间设备的配置也存在差异。一些边缘云，由于任务较少，或者配置较高，存在资源剩余的现象；而一些边缘云，由于任务较多，或者配置较低，则存在资源不足的现象。因此，如图1-2所示，如何进行合理的计算卸载，高效地协同边缘云与中心云，以及边缘云与边缘云之间的卸载，得以充分利用闲置的边缘资源，均衡通信以及计算压力，从而制定出相应的卸载策略，进而最优化特定目标，就成为了一个有益的研究方向。

2）如何解决存储资源不足的问题。对于存储资源不足，通常采用的方法为对存储对象进行选择，保存一些常用的数据，而将一些冗余甚至不常用的数据进行清除，从而节省存储空间。同时，对于云平台，除了传统的应用数据，虚拟机或者容器的镜像，也是一种不可忽略的数据。镜像所占用的空间，对于本身存储空间就不充足的边缘而言，非常可观。一些镜像，被容器使用的频率较高，而一些镜像则被容器使用的频率较低，并且镜像本身大小的差异性同样也需要被作为选择性存储的考量因素。另外，鉴于边缘存储空间有限以及镜像的使用频率高，因此可以将镜像的存储视为一种缓存。同时，考虑到网络因素对于镜像下载的时间制约，如果能事先预测出未来所需的镜像，进行预先存储，则不仅节省了存储空间，更节约了由下载带来的传输时间，并降低了网络拥塞的可能性。因此，如何进行合理的镜像缓存，预先清除冗余镜像，甚至预测即将使用的镜像，进行预先存储，也成为了一个有益的研究方向。

* 1. 国内外研究现状

本节分为两个部分，分别介绍现有计算卸载技术和缓存替换技术在国内外的发展现状以及边缘场景下的应用，然后分析现阶段技术在边缘计算平台上使用的局限性。

1）计算卸载技术的研究现状

近年来，随着智能终端设备的大量普及，使得网络带宽资源越来越紧张，但用户对网络性能的要求却越来越高。同时，也随着终端设备性能的不断提升，终端开始充当更加重要的角色。但随之而来，应用程序对设备性能的需求也越来越高，同时移动终端设备也受制于自身能耗的局限性，不能长时间连续的运行。为此，计算卸载的概念应运而生。计算卸载技术[10][11][12]是指受资源约束的设备完全或部分地将计算密集型任务卸载到资源充足的云环境中。通常，计算卸载可以分为卸载决策和资源分配两部分。卸载决策，是对用户端是否卸载、卸载量以及卸载对象的决策；资源分配，则是资源卸载目的地选择的问题。一般的流程为，优化目标影响卸载决策，接着进行相应的资源分配，最后得到具体的计算卸载方案。

通常，计算卸载的优化目标分为，降低时延[13~18]、降低能耗[19~27]、以及权衡能耗和时延[28~31]，同时需要明确卸载目的地为单一节点还是多个节点。Zhang K等人[13]考虑到在计算资源有限的情况下，采用Stackelberg博弈论解决了以降低时延为目标的多用户计算卸载方案。Jia M等人[16]为了使得时延最小化，考虑到不同类型的任务对计算卸载有不同的需求特点，进而采取差异化的计算卸载方法。Xu J等人[55]提出了基于增强学习的方法管理资源，进而得出了最佳的动态卸载方案。

以上对于计算卸载的问题，更多的是考虑终端与边缘云之间的计算卸载。其实，考虑到边缘设备计算能力的局限性，以及地区间任务数量的差异可能导致边缘云之间负载不均衡的问题。因此，边缘计算的计算卸载，也应该考虑边缘云与边缘云之间的计算卸载。

2）缓存替换技术的研究现状

如何更合理地利用存储资源极度有限的边缘服务器进行数据的缓存，进而提高缓存的命中率，一直受到研究者的关注。对于缓存优化的作用，主要是实现在有限的缓存空间条件下，通过合理高效地缓存算法，提高缓存命中率，从而降低网络带宽的开销和使用的时延，进而提高用户的使用体验。通常，缓存优化技术主要分为两部分：首先，数据通过缓存决策策略选择合适的存储节点进行缓存；接着，倘若选择好的节点存储空间不足，则需要通过缓存替换策略将原先存储在节点上的数据替换掉。然而，鉴于边缘云与CDN等网络需求的差异性，因此对于边缘云上存储镜像的优化，只需要考虑缓存的替换算法。

传统的缓存的替换算法，可以分为：a）基于时间特性，例如FIFO（First in First out，先进先出）算法，是指如果一个数据最先进入缓存中，则应该最先被淘汰更换。b）基于访问时间特性，例如LRU（Least Recently Used，最近最久未使用）算法，通过记录数据最近一次被访问的时间，当缓存空间不足时，将与时间间隔最久没被访问的数据替换。c）基于访问频率特性，例如LFU（Least Frequently Used，最近最少使用）算法，基于假定在最近的一段时间内被访问次数少的数据，那么在将来的一段时间内被访问的可能性也小。因此，通过记录该段时间内，数据的被访问的次数，将访问次数最少的数据替换。d）基于内容大小，例如SIZE算法，将内容大的数据优先替换出去，进而获得更多的缓存空间，但缺乏考虑大量内容小的数据可能访问数也低的问题。

对于传统的缓存的替换算法，其设计的考虑因素不免有些片面。相比之下，Das S 等人[32]提出了一种LFRU的算法，通过将LFU与LRU结合，综合考虑了访问频率与最近访问时间的因素，但由于平衡系数为固定值不能动态地进行修正，因此依旧属于没有考虑流行度的方法。Pei C等人[33]提出的GDS（Greedy Dual-Size）算法，综合考虑了对象大小、缓存成本以及运行时间，通过这三个指标作为评判标准对每个对象打分，每次替换分数最低的对象，不过该算法缺少对过去访问次数的考虑。之后，Ludmila Cherkasova[34]，与Ma T[35]提出的GDSF(Greedy Dual-Size Frequency)和WGDSF算法用以提高性能。Chootong S等人[37]提出了基于内容流行度的缓存替换策略，该策略的流行度通过数据的缓存命中率、最近请求时间的间隔和请求次数等因素确定，再每次对流行度最低的数据进行替换。黄丹等[38]也提出了基于内容价值的替换策略。

* 1. 论文的主要研究内容

本文在上述背景下，针对边缘计算平台下计算资源和存储资源不足的问题，采用基于Kubernetes Federation的云-边一体化边缘计算云平台，通过用以计算卸载的调度器模块和用以镜像存储的缓存器模块的设计、实现以及验证，得以进一步提升平台的资源利用率。因此，本文主要研究内容有以下三点：

1）边缘计算中计算卸载方法研究

本研究点为，如何进行合理的计算卸载，高效地协同边缘与中心云，以及边缘与边缘之间的卸载，得以充分利用闲置的边缘资源，从而最优化总体时间。具体的主要研究内容包括：

通常，计算卸载主要分为两部分，全卸载与部分卸载。为了更好地充分利用计算资源，因此本项目采用部分卸载。对于部分卸载，则需要设计算法去决定应用应该如何划分，以及卸载去处。首先，由于对于每一个应用，都可以分解成多个前后关联的任务。因此，倘若相关联的任务分散分布，会使得产生多余的不必要的数据传输时间。另外，对于每一个任务，同时也需要考虑到用户数据隐私性等安全问题。基于此，将决定此任务是否具有可卸载性。倘若此任务为不可卸载任务，则只能在本地边缘进行计算处理；否则，进行卸载去处的选择。其次，计算卸载的本质其实就是任务的卸载。也就是将不同的任务从本地边缘卸载到其余的边缘或者中心云上。在任务卸载时，需要考虑待选去处的一些性能指标，例如CPU、Memory、Storage，去处的这些指标必须满足任务的需求。另外，对于在其余的边缘或者中心云是否本地也还有任务需要处理，因此带来的等待时间开销，也必须作为一个考量因素。在本项目中，我们的目标为充分利用周边资源富足的边缘，选择最优的卸载策略，最优化总时间开销。因此，在进行卸载时，将面临以下挑战：a）应用划分生成的任务，所具备的前后关联性。因此，后继任务必须等待前驱任务完成才能开始执行；b）同一个边缘或者中心云可以同时处理多个任务。因此，如何合理的分配任务给不同的边缘或者云中心，使得总时间的减少。

鉴于这种情况，本项目提出了基于双边博弈的计算卸载方法。通过先将划分好的任务进行集群聚合构建，把具有强联通特性的任务节点合并、等同为一个节点。同时，考虑到用户数据隐私性等安全问题，当集群内任意任务节点为不可分发时，此任务集群也不可分发；再通过任务集群端与所有边缘和中心云组成的服务端，进行使双边偏好度最大化的多对一双边稳定匹配方法，从而得出稳定的匹配关系，即符合条件的相对最优卸载方案。

2）边缘计算中镜像缓存方法研究

根据云平台需要存储镜像的特性，本项目中的镜像存储即为镜像的缓存。基于此，本项目研究点为，如何进行合理的镜像缓存，预先清除冗余和不常用镜像，同时预测即将可能使用的镜像，进行预先下载存储，以节省传输时间。具体的主要研究内容包括：

首先，考虑到传统处理类似问题时，通常所采用的LRU（Least Recently Used，最近最少使用）算法，LFU（Least Frequently Used，最不经常使用）算法，FIFO（First Input First Output，先入先出）算法等。然而，镜像的流行度是动态变化的，受时间和种类的影响明显，部分镜像的流行度随着周中的日期变化剧烈，而一些镜像则几乎不受影响。因此，这些传统方法也就不适用于当前场景。同时，基于平台通用性的考量，用作预测的数据量大小和来源，也变得重要起来。因此，在进行预测时，将面临以下挑战：a）进行训练的数据采集，以及数据量的大小；b）如何更合理地预测出未来所需镜像；c）如何提高镜像存储算法的响应时间。

鉴于这种情况，本项目提出了基于流行度预测的镜像缓存方法。首先，针对平台通用性的特点，因此可以由百度指数的搜索量来间接体现镜像的使用量。采集百度指数上，不同地区（北上广深杭）不同镜像（40种）的使用量，将近五十多万条数据。再将这些数据进行预处理，清除一些噪声数据，并将数据转换成预测模块中所需的特征输入，进行基于GRU的流行度预测模型训练。之后，使用GRU训练出的模型，对第二天的流行度进行预测。基于流行度预测的结果，对本地边缘的镜像按流行度排序，按逆序排名置换冗余或者不常用镜像。

3）平台系统实现与性能分析

鉴于当前尚未有开源的边缘计算平台，因此将基于原生的传统Kubernetes云计算平台，采用Kubernetes Federation，实现完成支持云-边一体化的边缘计算云平台。其中，该计算平台包括，一个中心云和三个边缘云，其中，每个云都有三个主机。另外，由于设备资源的不足，因此通过使用在服务器上的VMware虚拟化实现部分边缘设备，并通过桥接模式保证所有设备处于相同的子网中。之后，在原平台的基础上，采用基于双边博弈的计算卸载方法，设计并实现了边缘计算云平台的调度器；采用基于流行度预测的镜像缓存方法，设计并实现了边缘计算云平台的缓存器。

对于本文所提出方法的性能分析，一方面，调度器模块里的基于双边博弈的计算卸载方法，利用云-边一体化平台，分别从边缘云数量、输入应用大小、以及输入应用的数量三个角度，通过对比实验以及对平均完成时间的结果进行性能分析，从而证明本文提出的计算卸载方法能够有效节约完成时间；另一方面，缓存器模块里的基于流行度预测的镜像缓存方法，先对预测方法进行验证集里Loss损失函数和值的效果分析，以及测试集中真实值与预测值的比较分析，进而验证预测模型的有效性，再利用云-边一体化平台，通过与LRU和LFU缓存命中率的对比实验以及结果分析，从而证明本文提出的镜像缓存方法能够在存储空间有限的条件下，尽可能地满足高的缓存命中率。最后，再将这两种经过验证通过的方法，通过容器化的方式接入系统，实现系统的计算卸载调度器以及镜像存储缓存器，进而有效地缓解了边缘计算平台上计算资源和存储资源不足的问题。

* 1. 论文的组织结构

本文根据现有边缘计算平台中存在的计算资源、以及存储资源不足的问题，提出了基于双边博弈的计算卸载方法，和基于流行度预测的镜像缓存方法。进而，在Kubernetes Federation的基础上，分别设计实现了边缘计算云平台的调度器和缓存器。本文总共分为七章，各章的内容组织如下：

第一章：绪论。本章首先介绍了本文的研究意义与背景，然后介绍了计算卸载技术以及缓存替换技术在国内外的研究现状，最后介绍了本文主要的研究内容。

第二章：相关技术简介。本章介绍了与本文相关的一些技术，包括云计算与边缘计算、Docker、Kubernetes以及深度学习里的RNN、LSTM和GRU。

第三章：基于双边博弈的计算卸载方法。本章首先分析了边缘计算环境下的通信、计算模型，并明确了卸载的优化目标为最小化完成时间；然后，基于效用函数的研究，发现本问题是一个混合整数的非线性规划；接着提出了基于双边博弈的两阶段计算卸载方法。

第四章：基于流行度预测的镜像缓存方法。本章首先分析与镜像流行度相关的因素；然后，采用基于GRU的镜像流行度预测，将提取的特征因素放入模型训练；最后，提出基于流行度预测的缓存替换策略，使得充分利用边缘的存储空间。

第五章：系统设计与实现。本章通过分析现有的平台，并结合边缘轻量化的特点，首先提出采用Kubernetes Federation作为边缘计算云平台的基础框架。然后，基于双边博弈的计算卸载方法设计并实现了边缘计算云平台的调度器模块；接着，基于流行度预测的镜像缓存方法设计并实现了边缘计算云平台的缓存器模块。

第六章：实验分析与系统测试。本章基于平台功能实现后的系统，对系统分别依次进行基于双边博弈的计算卸载方法、基于GRU的镜像流行度预测和基于流行度预测的镜像替换方法的实验分析。其中，计算卸载方法和镜像替换方法采用了对比实验。通过验证这三种方法的有效性，也表明了系统调度器和缓存器的性能到达了预期的目标。

第七章：总结与展望。本章通过对本文完成的工作进行总结，提出了下一步研究的建议，以及对今后工作的展望。

1. 相关技术简介

本章主要介绍了本文后续会涉及一些技术知识，包括首先，介绍了云计算和边缘计算的相关概念，然后，对平台构建相关的Docker和Kubernetes，介绍了它们的基本概念和原理。特别地，对于Kubernetes的调度器原理进行了讲解，以便于后续设计自定义调度器。最后，介绍了RNN、LSTM和GRU，便于后续选用训练模型时作为参考。

* 1. 云计算与边缘计算

云计算（Cloud Computing）的初衷是为了对资源进行更好的管理利用，其管理的对象主要为：计算资源、网络资源、以及存储资源。基于此，为了能更加合理的管理利用资源，进而提出了资源虚拟化的概念，包括计算资源虚拟化、网络资源虚拟化、以及存储资源虚拟化等。从某种角度而言，云计算的核心就是虚拟化技术，其本质也就是，从资源到架构的全面弹性灵活化。云计算平台通过这些虚拟化技术，将分散的物理资源进行整合、优化、重组，提供给用户使用。从用户的角度而言，使用云资源主要是为面向具体的服务需求，通过统一的云计算平台入口，进而达到对于物理资源设备的透明化，简化了物理资源部署所带来的诸多不便。另外，从云服务提供商的角度而言，使用云平台能够有效地解决资源分布不均，以及资源紧张和资源闲置的局面，进而得以最大化的利用资源，为企业带来创收。

然而，随着万物互联时代的来临，终端设备的数量呈现爆炸式的增长，进而导致了海量数据的产生。面对随之而来的海量数据处理，传统的云计算集中处理模式已经不能很好的应对一些问题[5]：

1. 网络带宽。鉴于大量数据的产生，倘若都使用传统的云计算集中处理模式，则必然导致当下已经资源紧张的网络带宽资源，承载更大的压力。并且，由于设备的泛滥，从而也加剧了冗余数据的产生，这些冗余的数据同时也将极度降低网络带宽的利用率。

2. 实时性。伴随万物互联时代的来临，同时也意味着终端设备需要能够更加迅速地处理一些实时信息，倘若依照传统的云计算集中处理模式，除了传输带来的时间开销以外，大量的数据处理一起涌入云端，则必然导致一些数据处理的时间延迟。

3. 隐私保护。随着终端设备的不断增加，一些设备也开始收集一些用户的隐私信息。例如，位置信息、图片信息、身体健康信息等等。使用传统的云计算集中处理模式，这些私人隐私数据的大量传输以及存储在云端设备中，都大大增加了这些数据泄露的风险。

4. 能耗。传统的云计算集中处理模式的能耗问题一直是一个研究的热点。同时，终端设备往云中发送数据，也导致了终端的能耗问题。倘若使部分任务从云中心迁移到终端上，或者近终端的边缘中心。这样，不仅可以缓解云中心的数据处理压力，同时也能有效的降低能耗。

基于以上的一些原因，边缘计算（Edge Computing）[39] [40]逐渐成为研究的热点领域。广义而言，边缘计算就是一种分散式的运算架构，通过将传统的需要传输到云中心的任务进行分解、拆分，切割成更小的单元，然后分散到更加靠近用户终端的边缘节点上，进而实现更加高效的数据处理。

同时，一些相近的概念也不断涌现。例如，思科于2012年提出的雾计算（Fog Computing）[7]，欧洲电信标准协会主推的多接入边缘计算（Multi-Access Edge Computing，MEC）[6]也就是原先的移动边缘计算（Mobile Edge Computing），以及由卡内基梅隆大学Elijah项目衍生出的Cloudlet[8]等等。通过分析这些不同的名词概念，不难发现无论是边缘计算、雾计算、多接入边缘计算，还是Cloudlet，其实其本质都是通过拉进传统云中心与终端设备之间的物理距离，从而在降低网络时延的同时，有效地处理网络带宽、隐私性，以及能耗的问题。

另外，需要注意的是，边缘计算的推出并不是为了取代传统的集中式处理的云计算模式，而是作为其有益的补充，得以构建云-边一体化的计算生态。

* 1. Docker

Docker 是 PaaS 提供商 dotCloud 开源的一个基于Linux 容器（LXC）的高级应用容器引擎，其基于Golang语言并遵从Apache2.0协议开源。并且，容器是完全使用沙箱机制，相互之间不会有任何接口。[41 [42]

Docker自推出以来，一直受到使用者的热捧，因为其有效解决了平台间程序的迁移问题。用户可以通过Docker的镜像（Image）在不同的操作系统平台下，诸如Windows、Ubuntu、CentOS等，一键快速部署环境。迁移时，也只需将Docker打包成镜像发布，直接在迁移的目的地拉取所需的对应镜像，进而优化原先冗杂繁琐的部署过程。

Docker作为容器的一种，也是基于软件的平台的虚拟化技术，因此不免与虚拟机（Virtual Machine，VM）进行一些比较。如图2-1所示，其中，对于虚拟机，其属于物理层面的完全虚拟化，也就是从底层物理硬件设备层面实现虚拟化的技术。例如，常见的OpenStack、VirtualBox和VMWare WorkStations等都属于使用虚拟机的虚拟化软件。而对于Docker这种容器，则属于操作系统层面的虚拟化，也就是直接通过内核创建虚拟的操作系统实例（内核和库），进而隔离不同的进程和资源。从某种意义上而言，Docker相当于操作系统中的一个进程。

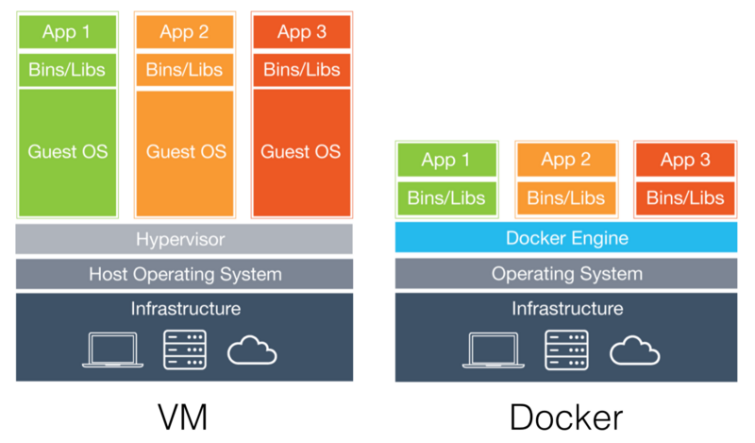


图2-1 虚拟机与Docker容器的对比[43]

Docker实现的底层主要由namespace、cgroup、以及AUFS三部分构成。其中，namespace，即命名空间，是隔离技术的第一层，确保 Docker 容器内的进程看不到也影响不到 Docker 外部的进程；cgroup，即Control Groups，是LXC技术的关键组件，实现了对资源的配额和度量，用于进行运行时的资源限制；AUFS (AnotherUnionFS)，即一种UnionFS文件系统，是容器的构件块，创建抽象层，从而实现 Docker 的轻量级和运行快速的特性。另外，得益于AUFS的特性, 每一个对只读层（readonly）文件/目录的修改都只会存在于上层的可写层（writable）中。这样由于不存在竞争, 多个容器就可以共享可读层。

对于Docker的使用，除了基本的操作语法之外，还需要知道容器（Container）、镜像（Image）以及Dockerfile这三个概念。镜像，就是上述的只读的 Docker 容器模板，简言之就是系统镜像文件。容器，即用于运行应用程序的容器，包含操作系统、用户文件和元数据，相当于镜像Images的一个运行实例，也就是上述的只读层。另外，用户可以在原镜像的基础上生成Docker容器，再将此容器打包成新的包含修改的镜像文件。对于构建镜像，除了在Docker容器基础上打包外，还可以采用DockerFile的形式，编辑镜像创建的指令文件。

* 1. Kubernetes

Kubernetes[44]是谷歌基于Borg推出的开源的大规模集群管理编排系统，其面向的编排对象为容器（Container），主要就是Docker容器。在Kubernetes中，调度的基本单位并不是容器，而是Pod。Pod是一组紧密关联的容器集合，它们共享了PID、IPC、Network和UTS namespace，也就是说其内的多个容器共享网络和文件系统。因此，Pod可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。所以，Kubernetes是通过使用Pod对容器进行管理。同时，因为Pod的生命周期短暂，当重启Pod时其IP地址极有可能改变，使得Kubernetes提供的服务难以持续关联。鉴于这种情况，Kubernetes通过引入服务（Service）机制解决这个问题。具体地，服务是定义一系列Pod以及访问这些Pod的策略的一层抽象。Kubernetes先将服务与Pod的绑定信息存储在Etcd中，之后需求通过定位服务的地址，进而实现与具体Pod的连接。



图2-2 Kubernetes架构原理

如图2-2所示，为Kubernetes架构原理。对于Kubernetes集群，是由一组安装了Kubernetes的物理服务器或者虚拟机组成的节点群。在一个完整的集群内，包含了分布式存储Etcd和物理节点。其中，物理节点分为Master节点和Node节点。通常而言，集群有一个Mater节点和若干个Node节点。对于Master节点，一般是一台独立的服务器。但是，当在生产环境中需要保证高可用的情况下，则需要Master节点为三台服务器。

Master节点，即为控制节点，是Kubernetes集群内的控制中心，对集群进行调度管理。例如，容器的调度、维护资源的状态、自动扩展以及滚动更新等。具体地，Master节点包含的核心组件为kube-apiserver、kube-controller-manager和kube-scheduler。其中，kube-apiserver提供了资源操作的唯一入口，并也是集群控制的入口进程；kube-controller-manager则是Kubernetes里所有资源对象的控制中心，责维护集群的状态；kube-scheduler则是负责资源调度的进程，将 Pod按照预定的调度策略调度到相应的节点。

Node节点，则为计算节点，是Kubernetes集群内负责具体操作计算或者存储的节点，并且管理镜像和容器以及集群内的服务发现和负载均衡。具体地，Node节点包含的核心组件为kubelet、kube-proxy和Container runtime。其中，kubelet负责Pod对应容器的创建、启停等任务，同时也负责Volume（CVI）和网络（CNI）的管理；kube-proxy则是负责为服务提供集群内部的服务发现和负载均衡；Container runtime则是负责镜像管理以及Pod和容器的真正运行（CRI）。

Etcd集群，则主要保存了集群的状态，包括服务（Service）和pod间的绑定信息等。通常而言，将Etcd集群与Master节点关联，使得Master节点能够使用Etcd上存储的集群状态。

由于本文需要实现自定义的调度器，因此需要对于Kubernetes内的原生调度器（Scheduler），进行介绍。首先，调度器的作用为监听新创建没有分配到Node的Pod，并为此Pod选择一个Node进行绑定（bind），表明该 pod 应该放到的节点，之后将绑定信息更新到Etcd中进行存储。同时，在调度时需要考虑诸如公平、效率、灵活以及资源利用率高等因素。对于默认的调度器，其调度过程如下：

1. 将不满足条件的节点过滤，即predicate过程；

2. 对满足条件的节点再依照优先级顺序进行排序，即priority过程；

3. 选择优先级最高的节点。

倘若中间任意一步存在问题，则直接返回错误。

其中，对于predicate过程，平台自带了一些筛选因素的算法。例如，PodFitsResources是要求节点上剩余的资源满足Pod所需的资源；PodFitsHostPorts则是Pod申请的port端口是否在节点上已经被占用等等。另外，在predicate过程中，如果没有合适的节点，则pod会一直处于pending的状态，从而不断地进行新的调度过程，直到有节点满足条件为止。对于priority过程，平台也给出了优先级评判的标准。例如，LeastRequestedPriority是考虑CPU和Memory使用率的因素。

* 1. RNN、LSTM与GRU

鉴于本文在基于流行度预测的镜像缓存方法处，对镜像流行度的预测需要采用GRU的网络结构，因此此处对RNN、LSTM以及GRU作些简要介绍。

RNN[46]（Recurrent Neural Network，即循环神经网络）通常应用于具有时序特性的数据处理中，例如天气预测、股价走势、以及自然语言处理这种由有序词汇组成且前后具有关联性的数据。因为，RNN是一个自连接成环的神经网络单元，如图2-3所示，因此，可以将自循环的网络看成是由若干个相同的无环网络单元组合。正是由于循环次数的不确定性，使得RNN不同于前馈神经网络，可以处理任意时序的输入序列。因此，RNN不仅具有前馈连接，也还具有自身的反馈连接，使得RNN在能够在预测与时序相关的数据时，比前馈神经网络有更好的效果。然而，由于RNN自身网络结构在每个神经元内中只有一个tanh层激活函数的特性，使得当输入的序列过长时，RNN不能进行很好的学习，进而导致反馈消失的现象。因此，RNN不能处理有长期依赖关系的数据，也即RNN拥有长期依赖问题【】。

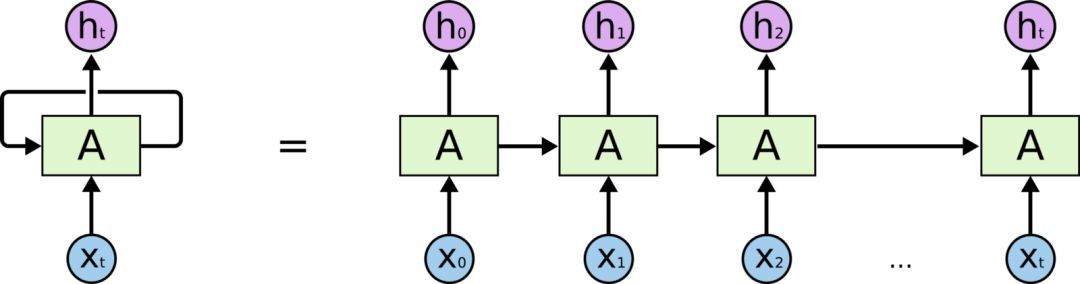


图2-3 展开的RNN[45]

对于RNN的长期依赖问题，LSTM[47] [48] [49]（Long-Short Term Memory，即长短期记忆网络）则能很好的处理。其实，LSTM是RNN的一种变种形式，但与RNN完全记忆不同，LSTM对时序信息采用选择性记忆的方式。LSTM新引入了两个概念，细胞状态以及“门”结构，具体的流程如下：

首先，经过遗忘门。遗忘门的作用，是通过sigmoid函数将一些不重要的信息从细胞状态下删除。其次，经过输入门。通过将前一时刻的隐藏状态与此刻的输入信息进行融合，再用sigmoid对刚刚融合的信息决定是否保留。然后，计算细胞状态。将上一层的细胞状态与遗忘向量逐点相乘，然后再将该值与输入门的输出值逐点相加，将神经网络发现的新信息更新到细胞状态中去。至此，就得到了更新后的细胞状态。最后，通过输出门。将 tanh 的输出与 sigmoid 的输出相乘，以确定隐藏状态应携带的信息。因此，遗忘门确定前一个步长中哪些相关的信息需要被保留；输入门确定当前输入中哪些信息是重要的，需要被添加的；输出门确定下一个隐藏状态应该是什么。

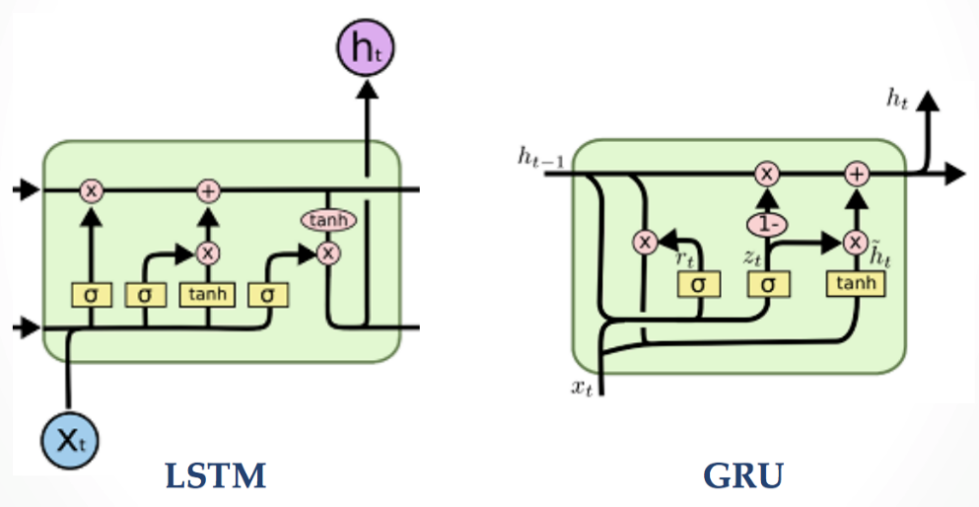


图2-4 LSTM与GRU的对比[45]

如图2-4所示，GRU[50]（Gate Recurrent Unit，即门控循环单元）也是RNN的一种变体和LSTM之间也有所区别。相比于LSTM的结构，GRU则只包含两个门，也就是更新门和重置门。更新门的作用类似于 LSTM 中的遗忘门和输入门。它决定了要忘记哪些信息以及哪些新信息需要被添加。重置门用于决定遗忘先前信息的程度。

但由于考虑到GRU 的张量运算较少，因此其所需时间也更低，所需数据量也更少。因此，在本文中拟采用GRU算法进行镜像流行度的预测。

* 1. 本章小节

本章通过介绍与本文相关的一些技术，包括云计算与边缘计算、Docker、Kubernetes以及深度学习里的RNN、LSTM和GRU。通过云计算与边缘计算、Docker以及Kubernetes的介绍，为后文Kubernetes Federation作为底层平台，以及使用Kubernetes的内部机制去实现本文所需的调度器做好铺垫；通过对RNN、LSTM和GRU的介绍，则方便于基于GRU的镜像流行度预测工作的开展。

1. 基于双边博弈的计算卸载方法

本章将设计基于双边博弈的计算卸载方法，首先，通过介绍边缘计算的计算卸载架构，了解边缘计算的通信模型和计算模型。接着，对一个应用程序内的多个计算任务进行可否卸载的重新划分，并得到任务间前后依赖的关系。之后，对提出了本章计算卸载的总目标为满足最小化时间成本的开销。最后，基于设定的目标，设计了两阶段的基于双边博弈的计算卸载方法，分为集群聚合构建阶段，以及多对一稳定匹配博弈阶段。

* 1. 边缘计算卸载架构

通常，边缘计算是与传统云计算协同工作而不是取代云计算的中间层。 因此，典型的三层架构即为：中心云-边缘云-终端。第一层，终端即智能终端，包括智能手机和传感器采集节点等设备。第二层，边缘云，主要为了降低云计算中心的计算负担，减缓网络带宽的压力，提高数据的处理效率。同时，为了满足低延迟特性，边缘云通常部署在终端的物理位置附近。第三层，中心云即传统的云计算中心。在本方法中，主要关注的是第二层和第三层之间的计算卸载。另外，对于边缘计算卸载，需要了解其如图3-1所示的架构模型。因此，本节将从通信模型和计算模型两个角度介绍卸载的架构模型。

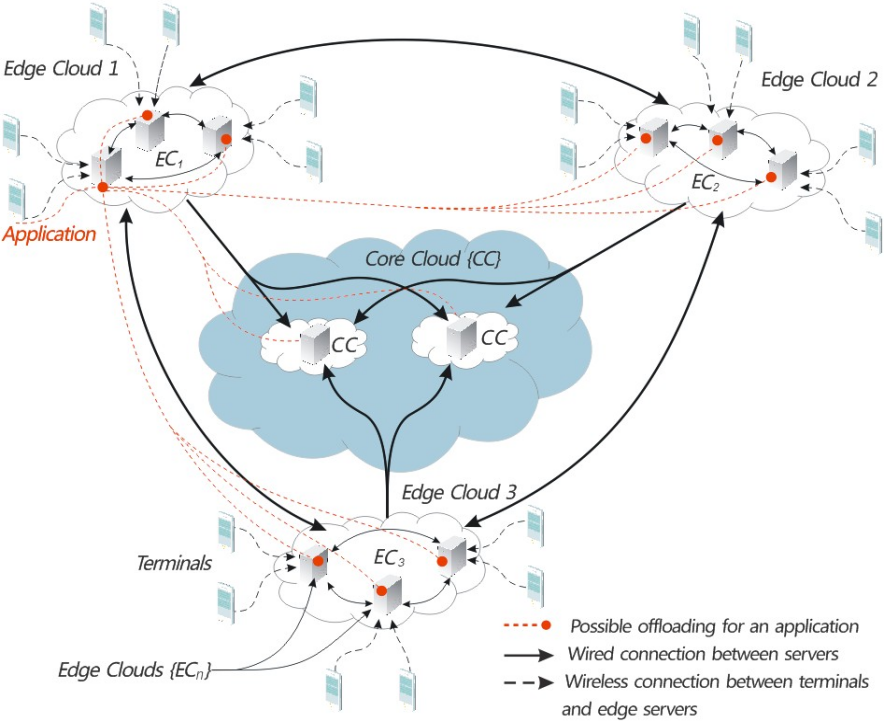


图3-1 云-边协同计算卸载架构

* + 1. 通信模型

通常而言，边缘计算设备通过有线传输数据，并且这些设备往往是服务器或者个人计算机。一方面，对于边缘云之间的传输，由于传输的数据量不大，因此上行数据速率可看成固定值。另一方面，在边缘云和中心云之间，由于中心云连接了大量边缘，则将中心云与边缘云之间的上行链路数据速率记为。

* + 1. 计算模型

首先，表3-1中列出了用于问题定义的一些标记符号。接下来，本章关注本地边缘计算，异地边缘计算和中心云计算的相应的应用完成时间。

表3-1计算模型中的一些符号标记

|  |  |
| --- | --- |
| **标记** | **边缘n上任务k对应所消耗的完成时间** |
|  | 在本地边缘执行 |
|  | 传输到其余边缘 |
|  | 传输到中心云 |
|  | 在异地边缘执行 |
|  | 在中心云上执行 |

1. 本地边缘计算

令表示任务m在边缘n上的计算能力（即CPU芯片的时钟频率）。在这里，本章允许边缘云上不同的边缘设备以不同的时钟频率执行任务。因此，可以得出，边缘n上的任务m在本地边缘n执行时，其计算执行时间为，

(2-1)

另外，在任务m开始执行之前，所有与其相关的前驱任务必须已经执行完成。然后，我们考虑任务m的等待时间为，

(2-2)

其中，表示任务m的相关前驱任务的集合。

1. 异地边缘计算和中心云计算

对于异地边缘计算或中心云计算，初始边缘n将其计算任务m卸载给它们。然后，异地边缘或中心云将执行计算任务并将结果返回到初始边缘。忽略异地边缘或中心云将计算结果返回到初始边缘所需要的时间。具体地，这个过程对应有两个阶段：

**传输阶段**：异地边缘计算和中心云计算的传输时间分别为，

和 (2-3)

**计算阶段**：边缘n的任务m在两处执行的计算执行时间分别为，

和 (2-4)

与上述类似，本章需要考虑异地边缘或中心云上任务m的等待时间，分别表示为和，即

(2-5)

(2-6)

* 1. 计算任务的划分

如图3-1所示，本地边缘的任务根据其容量和延迟要求分配给边缘服务器或中心云服务器。在本章中，我们假设一个由m个任务组成的应用程序。因此，应用程序用m个顶点表示加权图。图中每个顶点对应于应用程序中的一个任务。同时，考虑到任务相应数据的隐私性，我们使用布尔类型变量来标记是否为可卸载任务。图的每条边表示图中两个任务之间的交互。箭头的方向表示优先关系，它是任务和之间的优先约束，这样任务在其先前任务完成之前无法开始执行。如图3-2所示。

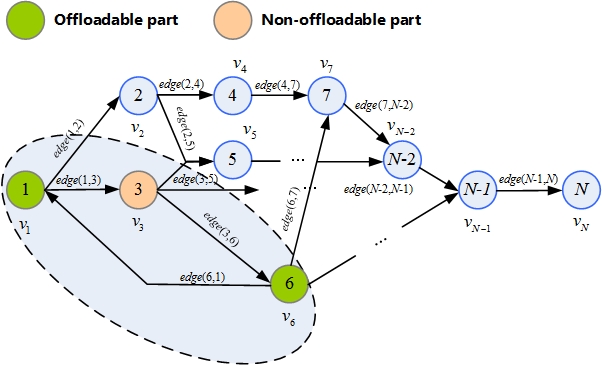


图3-2 应用划分任务图

另外，每个任务都有两个属性，即和。对于任务m，表示边缘n上计算任务m的计算输入数据的大小。表示完成边缘n上的任务m所需的计算工作量。特别地，对于输入数据，包括程序代码和输入参数。此外，计算工作负载是CPU周期的总数。

* 1. 边缘计算卸载方法的目标

对于计算卸载任务，本章中，其目标是要使得总体的完成时间开销最小化。基于此，首先要得到每个部分的效用函数（utility function），即完成时间。然后，将它们集成到总系统的计算开销中。

1）本地边缘计算

由于下行链路数据量明显小于上行链路数据量，因此在本章中，忽略了它的时间。基于此假设，本地边缘计算执行完成时间可以重写为，

, (2-7)

并且，上式也需满足，，。因此，在边缘n上的任务m在本地执行的完成时间是：本地边缘计算中本地执行时间和等待时间的总和。然后，我们可以获得本地边缘计算的效用函数，如下所示：

(2-8)

2）异地边缘计算

只有在任务完全卸载到同一个异地边缘或者任务m的所有相关前驱任务已完全在边缘执行后，异地边缘才能开始执行任务m，即

, (2-9)

忽略任务m的结果的接收时间，源自边缘n的任务m的异地边缘执行的完成时间是异地边缘计算中的执行时间和等待时间的总和。然后，我们可以得到异地边缘计算中的效用函数，即

(2-10)

1. 中心云计算

和异地边缘计算类似，我们可以获得中心云计算中的效用函数，即

(2-11)

然后，鉴于我们关注于最小化系统范围的完成时间开销，因此其目标函数可以被定义为，

(2-12)

其中，该式的约束条件为和，，。

鉴于上面的目标函数，我们将此问题表示为混合整数优化。具体地，这是一个混合整数非线性优化问题，如下所示：

(2-13)

该式的约束条件为：

(2-14)

, (2-15)

, (2-16)

, (2-17)

, , (2-18)

(2-19)

约束(2-14)确保总时间小于所需的最大完成时间。约束(2-15)确保任务m只能在其所有相关前驱任务完全执行完毕后才能开始执行。约束(2-16)和(2-17)表示任务优先级要求约束，表明任务m只有在任务完全卸载到中心云或异地边缘后才能开始在中心云端或异地边缘云端执行，或者任务m的所有相关前驱任务已完全在中心云端或异地边缘云端执行完成。约束(2-18)表示，和是二进制整数变量。约束(2-19)保证计算任务在本地边缘云服务器或异地边缘云服务器或中心云服务器中执行。

* 1. 基于双边博弈的计算卸载方法设计

显然，整数约束，，使得这个优化问题为NP难的。因此，解决这项NP难优化问题是一个挑战。受一对多匹配的启发，本章提出了一种新颖的基于双边博弈的两阶段计算卸载方法，包括任务重建阶段和稳定匹配阶段。

对于任务重建阶段，主要为任务集群的重新聚合构建，将相关任务进行重组；对于稳定匹配阶段，主要为任务侧与服务器侧之间形成稳定的多对一匹配关系。下面详细介绍了提出的方法：

* + 1. 集群聚合构建

如图3-2所示，鉴于任务之间的关系是相互依赖的，为了减少不必要的传输时间，我们将原始图形通过使用基于强连通分量（SCC）的集群聚合构建算法，使得图重构为。具体为，首先，找到图中所有可能的SCC。接着，将每个SCC视为新的顶点。然后，重置新顶点的值并将其设置为等于来自此集群中原始顶点的值的总和。此外，考虑到用户数据隐私等安全因素，一些原始顶点因此是不可卸载的。此时，只要某个新顶点中有一个原始顶点是不可卸载的，则该新顶点是不可卸载的。以下是该阶段方法的实现伪代码：

|  |
| --- |
| **算法1** 集群聚合算法 |
| 1：设置所有节点的访问状态标签为未访问，即visited = false；  2：设置栈stackPath和stackRoot；  3：**while**在节点集合V内存在visited == false的节点时**do**  4： 使当下的这个节点的访问状态标签为访问，即visited = true，并且将这个节点放入栈stackPath和stackRoot中；  5： **for**对节点v的任意一个邻接节点u **do**  6： **if**节点u的访问状态标签为未访问，即visited = false **then**  7： 回退到步骤3，且v = u；  8： **end if**  9： **if**节点u的访问状态标签为访问，即visited = true且未被删除**then**  10： **while** stackRoot.peek() != u **do**  11： stackRoot.pop()；  12： **end while**  13： **end if**  14： **end for**  15： **if** stackRoot.peek() == v **then**  16： 输出对应的强联通分量；  17： 设置新节点v’，使v’.memory = 0，v’.core = 0，v’.storage = 0，和v’.privacy = false；  18： **while**栈stackRoot是非空**do**  19： 使u = stackRoot.pop()；  20： v’.memory = v’.memory + u.memory；  21： v’.core = v’.core + u.core；  22： v’.storage = v’.storage + u.storage；  23： **if** v’.privacy == false 并且u.privacy = true **then**  24： v’.privacy = true；  25： **end if**  26： 将边关系edge(u,y)改为edge(v’,y)，其中y为图中的其它边；  27： **end while**  28： **end if** |
| 29：**end while** |

* + 1. 多对一稳定匹配博弈

首先，给出多对一匹配博弈的定义。

**定义1**：给定两个不相交的集合，和，定义一对多匹配函数，其中，对于和有，

1. 且；
2. 且；
3. 。

第一个条件表明每个任务最多可以分配给一个服务器。第二个条件意味着每个服务器最多可以执行计算任务。第三个条件意味着如果任务与服务器匹配，那么服务器也与任务匹配。

将被定义为服务器的任务的效用函数。由此可得，

(2-20)

其中，和是两个加权参数，是或，并且。如果任务在服务器上执行，有更高的卸载率和计算资源相比于执行在服务器上，显然，相比于服务器，任务更偏好服务器。

类似地，被定义为任务的服务器的效用函数。由此可得，

。 (2-21)

其中，是或。显然，如果服务器执行任务的计算开销高于任务，则服务器更偏好任务，对比任务。

基于此两效用函数可得各自的偏好度，进而得出多对一匹配算法的具体步骤，如图3-3所示。

**步骤1：**初始化：建立初始的匹配状态。

**步骤2：** 每个服务器都计算各自的效用函数，并根据效用值来更新自己的偏好列表。然后，再向自己最偏好的任务发出请求，并将其从偏好列表中去除。

**步骤3：** 对每个任务，对所有提出可卸载请求的服务器，以及暂时决定卸载的服务器，计算效用函数，从而更新自己的偏好列表。

**步骤4：** 随机选择任一任务，从暂时决定卸载的服务器上，去除此任务。然后，将任务分配给自己最偏好的服务器，再更新匹配状态。

**步骤5：** 返回步骤1，直到系统中不存在满足以下条件的任务，即此任务无偏好服务器，而又有服务器的最偏好任务为此任务。从而得到稳定的匹配。

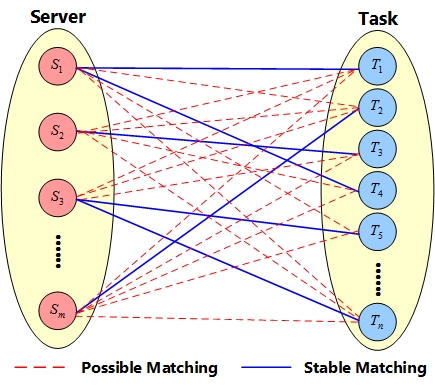


图3-3 多对一双边博弈匹配

以下是本阶段多对一稳定匹配算法的实现伪代码。

|  |
| --- |
| **算法2** 多对一稳定匹配算法 |
| 1：设置初始匹配状态函数；  2：**while**存在不能达到最佳匹配状态的任务时**do**  3： **for**服务器集合中的任意一个服务器n **do**  4： 为任务集合中的每一个任务，计算服务器n与之对应的效用函数；  5： 采用效用函数的值去更新此服务器n的偏好列表；  6： 接着，此服务器n再向自己最偏好的任务发出请求，并将其从偏好列表中去除；  7： **end for**  8： **for**任务集合中的任意一个任务m **do**  9： 对所有提出可卸载请求的服务器，计算此任务m与之对应效用函数；  10： 采用效用函数的值去更新此任务m的偏好列表；  11： **end for**  12： 随机选取任意一个任务m，接着从暂时决定卸载的服务器上，去除此任务；  13： 然后，将任务m分配给自己最偏好的服务器；  14： 更新匹配状态函数； |
| 15：**end while** |

* 1. 本章小结

本章通过首先分析边缘计算环境下的通信、计算模型，并明确了卸载的优化目标为最小化完成时间；然后，基于效用函数的研究，发现本问题是一个混合整数的非线性规划；接着提出了基于双边博弈的两阶段计算卸载方法。具体为，先通过强联通分量的集群聚合，再进行多对一稳定的匹配博弈。

1. 基于流行度预测的镜像缓存方法

本章将设计基于流行度预测的镜像缓存方法，利用公开数据集百度指数上相应镜像的热度指数，进行镜像流行度的预测。之后，基于流行度预测的数据，对边缘端服务器上的镜像进行存储替换。具体分为两部分，首先，是基于GRU的镜像流行度预测方法，包括神经网络的结构设计，数据的预处理，以及最后神经网络的训练和调优；其次，基于流行度的替换策略，对边缘云上的镜像进行更加优化的存储，达到在节约存储空间的同时，减少容器或者虚拟机可优化的响应时间。

* 1. 基于GRU的镜像流行度预测

本节将介绍基于GRU的镜像流行度预测，首先为数据集的收集，将百度指数作为镜像流行度的数据；其次，分析影响镜像流行度的相关因素，进而对数据进行输入特征的选取以及预处理操作；最后，则是对预测的训练模型的网络结构进行确定。

* + 1. 数据收集

本项目研究对象为边缘云平台中的Docker镜像，因此，可以将平台在某段时间窗口内镜像的使用量作为流行度的参考依据。但鉴于统计该方面数据缺乏普适和权威性，同时考虑到本项目的基础平台为通用性平台，因此我们使用百度指数中的Docker镜像对应的搜索指数作为镜像流行度的度量。因此，本文数据采集的是百度指数中2011年1月1日到2018年5月14日共计2690天的搜索热度指数。考虑到目前中国国内互联网技术公司主要分布在北京、上海、广州、深圳、和杭州，这五个城市，因此，采集数据的城市选择也为此五个城市。之后，参考Docker官方数据[55]以及CenturyLink实验室的调研[56]，选取了比较流行的40个Docker基础镜像作为边缘端存储镜像的预选存储镜像，例如，busybox，tomcat，nginx等。然后，本项目开发了一个爬虫程序去百度网站爬取以上介绍的数据，并存入到数据库中。每条数据记录了对应数据的日期、地区名、镜像类型、整体搜索趋势。

* + 1. 影响镜像流行度的因素

由于镜像本身特点的原因，其流行度的差异变化远不如一些音乐或者视频节目对生产商、制作者、以及推出时间等因素的存在明显的依赖程度。因此，对于镜像流行度因素的分析，通过采用对比流行度数值变化的趋势进行。

1. 流行度与时间的关系

在选取的两千多天的观察期内，随机选取了2018年1月1日至2018年4月1日这段时间内北京地区TensorFlow镜像的指数，如图4-1所示。另外，需要注意的是，2018年2月15日至2018年2月21日是国家法定节假日—春节假期。通过图4-1所示，可以明显看出，该类型的指数以周为单位存在着明显的不同程度的周期性变化。其中，周中工作日的指数明显高于周末的指数。此外，从图中还可以看出，2月14日起，指数开始出现明显的下跌趋势；而2月15日至2月20日这段时间内，指数出现明显的一直在低谷中；到2月21日之后，指数开始出现逐渐上升的趋势，直到恢复正常的周期性变化。对于这种现象，结合国家法定节假日的放假时间分析，可知这段时间内镜像的使用量明显减少是由于假期放假所致。基于此，我们又对比了其余节假日的情况，发现节假日时期，指数普遍出现了不同程度的下降，而且，与节假日的长度也存在某种关联。因此，可知镜像的流行度与时间存在相关关系。

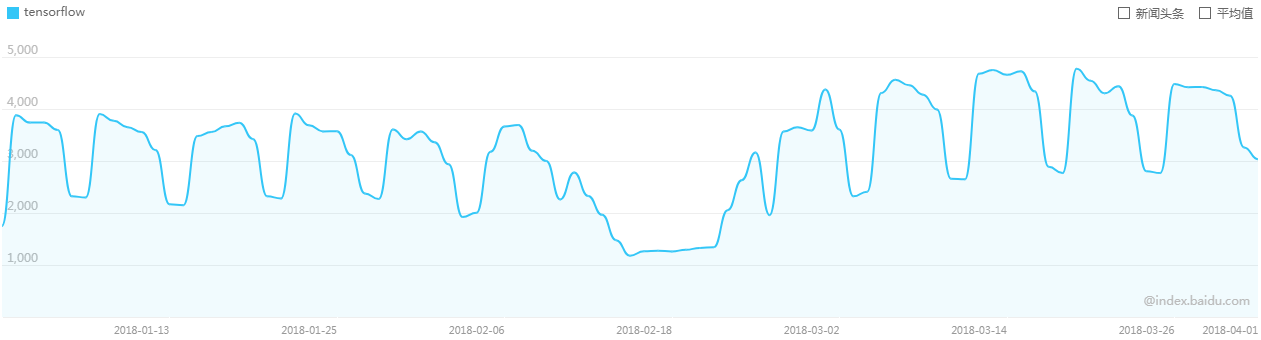


图4-1 流行度与时间关系

1. 流行度与镜像类型的关系

接着，我们又在选取的观察期内，随机选取了在2015年3月30日至2018年1月1日这段时间内，四种镜像（centOS、busybox、Nginx、和Redis）在全国范围内的指数分布图，如图4-2所示。通过采用全国的数据，可以有效剔除地区对于数据的影响性。从图中不难发现，对于一些镜像，虽然总体的指数平均值存在明显差异，但是总体的流行度指数变化一直具有相对的同步性。但是，对于另一些镜像而言，与其余的镜像使用则缺少相关关系。因此，由图可知镜像的流行度与其余的某些镜像也同样存在相关关系。

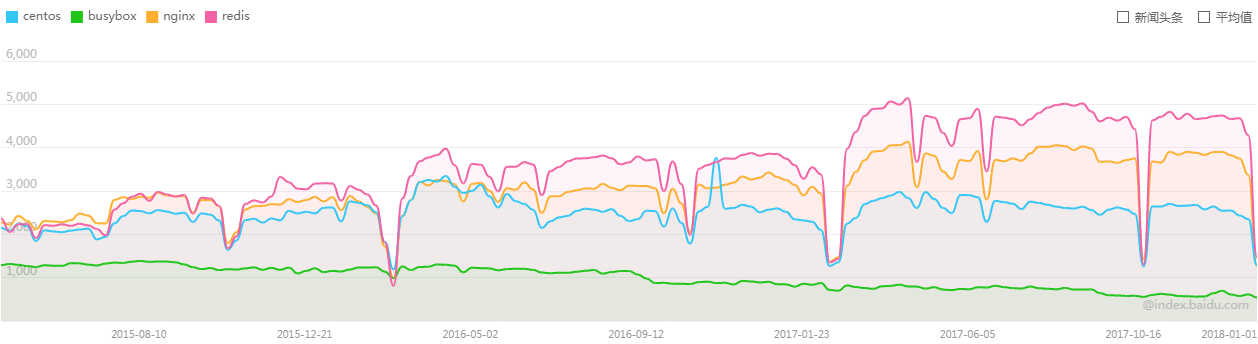


图4-2 流行度与镜像类型的关系

1. 流行度与地区的关系

同一时间内，同样的镜像类型，虽然在不同地地区，由于所处地区的产业、主要工作目标对象，以及地区人口等因素的不同，指数数值因此会呈现出明显的不同。然而，由于同种镜像的缘故，因而总体的变化趋势却呈现几近一致的趋势。之后，在选取的观察期内，本文随机选取了在2018年1月1日至2018年1月14日这14天时间内，MySQL镜像在北京、上海、广州、深圳和杭州五个地区的指数分布图，如图4-3所示。从中，不难发现，镜像流行度与地区也存在相关关系。

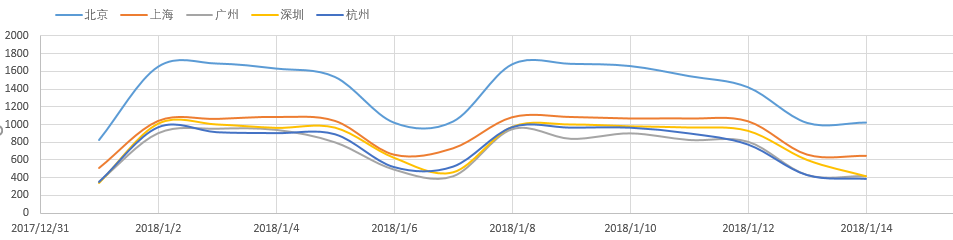


图4-3 流行度与地区的关系

4）流行度与热点（新闻头条）的关系

对于镜像流行度的预测，往往存在在某一特定天数，某种镜像的使用量出现突然增加的问题。对于这种情况，通过结合当下的资讯新闻，不难发现，通常是由于在当天有与该对应镜像的相关资讯出现，用户出于“尝鲜”的目的，导致用户对于该镜像的使用量增加。同样地，在选取的观察期内，随机选取了在2018年1月1日至2018年4月1日这段时间内，TensorFlow镜像在北京地区的指数分布图，如图4-4所示。在图中的A、B、C、D、E五点中分别存在不同程度的指数升高异常。结合当下的资讯可知，A点时，为2018年1月8日，当天谷歌发布TensorFlow 1.5；C点时，为2018年2月6日，前一天某用户在GitHub网站上发布了颇具热度的TensorFlow项目源码；D点时，为2018年3月1日，当天TensorFlow 1.6.0正式发布。因此，对于这些新闻热点，或可提前预知发布时间，或可通过当下突发时间，对之后出现的指数异常突增作好及时的预测，以提高预测的精确度。所以，可知流行度与热点（新闻头条）也存在相关关系。

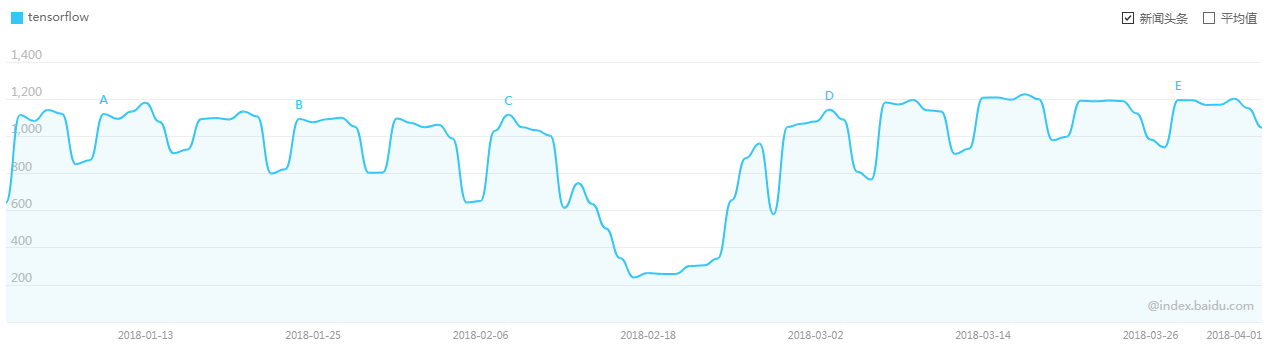


图4-4 流行度与热点（新闻头条）的关系

* + 1. 数据的预处理及输入的特征选取

从上一小节的分析可知，对于镜像流行度而言，与所选取的镜像特征具有相关关系。因此，本项目对于镜像流行度的预测，采用有监督的深度学习方法来对镜像的历史数据进行分析、学习，进而提取出与镜像流行度相关特征的相关性强弱，从而构建出镜像流行度随时间变化的模型。

对于百度指数中的数据，由于未知的原因，会出现一些不可预测且未知原因的数据突变式下滑，而对于镜像的使用而言，未知原因的使用量大幅下滑明显是出于数据的异常。同时，在数据爬取过程中，也会因为一些外部因素，例如爬取代码的异常等原因造成一些数据的缺失或者异常。为了使现有的数据能够更好的表达实际情况中流行度的趋势，所以需要在对数据的进行详细分析，以及模型训练之前，对采集来的原始数据进行预处理。

对于，爬虫过程中的数据异常或缺失，可以通过及时的对数据进行散点成图的操作，对于缺失或异常的数据，得以及时发现，并进而通过对比百度指数的数据进行更新。

而对于，百度指数中的一些异常、或者缺失的数据，则需要采用填充法对数据进行相应的填充，即采用上下相关的值去填充或者更新该值。常用的方法有，中位数填充法、均值填充法，和线性回归填充法等。对于本项目所研究的镜像的指数而言，由于指数数值上下之间存在着时序关系，因此在此种数据处理的过程中，如果遇到某天存在数据异常的情况，一般采用前后均值补全，最后利用处理后的数据作为训练数据集进行流行度预测模型的训练。

通过对原始数据的预处理后，接着需要提取训练模型的输入特征，并进行对应的数据预处理操作。在上一小节中，已经说明了四种与镜像流行度相关的特征，但对于这些特征也还需进行一些数据的预处理才能作为算法模型的输入。

1）时间。通过之前大量数据的数据研究不难发现，镜像的流行度与时间的强相关性。特别地，周末、节假日对使用量带来的影响。另外，不难发现，节假日的长度也会对使用量产生显著的影响。例如，正常两天的周末，使用量的下降明显没有春节以及十一这种七天长假带来的下降明显。因此，对于时间，其特征可以选取星期、节假日、节假日长度、节假日始末这四项指标表示。对于星期指标，采用独热（One-Hot）编码，先将其转换为长度为七的二进制位，接着使得星期对应的位数用数字1表示，其余位则使用数字0表示。对于节假日，采用一位的状态码区分；而对于节假日始末，也采用独热编码转换为三位的二进制数，分别对应节假日第一天，节假日最后一天以及平常日期。对于节假日长度，则直接采用对应数值表示。

2）地区。从之前的分析同样可以得出，同地区间存在不同的镜像，由于镜像功能和关联、互补，进而影响彼此流行度的现象。因此，它们的流行度间也可以作为互相的特征。出于对不同镜像功能光联的不确定性，本文采用将同一地区内其余镜像的流行度也都作为训练的输入特征。由于流行度数值的区间过大，不利于进行相应的机器学习。因此，对于流行度的数值需要进行归一化的操作。

3）镜像类型。同样，从之前的分析得出，同种镜像在不同地区也有相似的变化规律。因此，同样也将其余地区的同种类型镜像作为训练的输入特征。同样地，也需要对此流行度数值进行归一化操作。

4）热点（新闻头条）。显而易见，热点咨询的产生极易影响镜像的流行度升高。通过考虑可预见的热点咨询，例如镜像新版本号的发布等，能使得对于镜像的预测更加准确。但是，对于热度程度，并不能进行有效的评估。为此，本文通过将热点进行人工评估热度，分为5种程度，依次从1至5，热度逐渐递增。

* + 1. 流行度预测模型的训练



图4-5 预测模型的GRU结构

Cho, et al.[50]在2014年提出了GRU算法。作为循环神经网络RNN的一种变种，GRU与LSTM类似，也有效地解决了传统RNN不能处理长期依赖的问题。本文选用GRU模型的原因有以下三点：

1） 根据以上的分析以及实际经验易知，流行度的变化具有一定的连续性，并且与时间的关系密切。因此，当前日期的镜像流行度也与之前多个日期的流行度存在关联。

2） 由于本模型是应用于计算资源能力有限的边缘端上。因此，对于模型的选择，除了倾向于选择计算量更少，结构足够简单的模型，也要能使得计算耗时尽可能的减少，从而最大化的避免对边缘端其余服务的影响。

3）由于存在一些镜像，推出时间较晚，进而导致其流行度的数据量不够充分。因此，也希望训练模型所需的样本数量能相应的减少。

进而，需要对预测的训练模型进行网络结构的确定。考虑到边缘端计算资源能力有限的情况，本文采用尽量简单的网络结构，以实现流行度的预测模型。本文通过构建一个一层的GRU模型，将处理好的某一地区某一种类型的镜像的特征数据，按13天的时间序列顺序放入GRU模型中进行训练。之后，通过全连接层，输出预测值。如图4-5，为设计的GRU模型结构。

* 1. 基于流行度预测的镜像缓存更替策略

在Kubernetes中，当用户启动pod时，对于镜像的使用请求，可以采取以下的方式：首先，在本地的镜像缓存中进行查找是否存在符合条件的对应镜像，倘若符合需求，则直接使用该镜像；否则，需要向上级镜像库中一级一级地查询镜像，直到查询成功，之后再从相应的镜像库中下载该镜像到本地节点服务器上。

对于常规的云平台而言，由于拥有足够的缓存空间，并且Docker镜像相对而言的轻量化。因此，在缓存Docker镜像到本地节点服务器时，节点服务器一般不需要考虑空间的存储问题，只需在到达一定存储量阈值时，人工实现镜像的冗余删除与更替。然而，对于边缘计算云平台的边缘节点而言，由于其存储空间的约束性，即使Docker镜像所需的缓存空间已经足够轻量化，边缘平台也还需实现镜像缓存更新的自动化，以缓解边缘节点存储空间不足所导致的pod启动延迟问题。

因此，对于边缘计算云平台启动新pod而言，通过预先采用有效的镜像缓存的更替策略去决定该镜像是否预先缓存到本地的边缘节点服务器。使用这种方法，当本地不存在对应镜像时，得以有效地降低启动pod时需要拉取镜像而导致的网络带宽消耗。并且，同时也可以有效地缩短启动pod的响应时间，降低服务的延迟，从而更好地提高用户的使用体验。鉴于，边缘计算云平台边缘节点的缓存空间实在十分紧张。因此，需要从镜像仓库中选择预计使用量最高的镜像存储到边缘节点上，同时也需要将使用量不高的镜像，从对应的边缘节点上清除冗余，得以缓解缓存资源的紧张，进而能够最大化的存储使用量高的镜像到边缘节点服务器上。

良好的缓存替换策略能够使得缓存存储空间有限的边缘节点尽可能多的缓存不同的镜像类型。通常而言，常见的缓存替换算法有，FIFO、LFU、LRU。通常而言，当使用的镜像有使用量区别时，使用FIFO算法进行缓存替代，不能很好的体现出镜像访问使用的差异，该方法从某种意义上而言会使得镜像的缓存命中率保持在一个比较低的水平。因此，对于镜像最普通的去处冗余的办法，就是采用此种基于时间的淘汰规则。与FIFO算法不同，LFU算法和LRU算法都考虑到了镜像访问使用的差异，倘若镜像的使用量变化是相对静态，没有大规模的变化的话，再加上选择的时间范围足够长，在理论上这两种方法都能作为良好的缓存策略。然而，不幸的是，对于镜像的使用量而言，从之前采集的指数数据也能看出，其变化是高度动态的。同时，LRU算法和LFU算法，也无法有效预测镜像未来流行度的变化。并且，由于一些热点新闻的缘故，会导致第二天使用量的增加波动，对此，LRU和LFU也同样无法作出相应的变化。

基于此，通过采用基于镜像流行度预测模型的数据，作为缓存更替策略的依据。通过此方法，可以有效地提升缓存命中率，进而提升用户的使用体验。具体的缓存替换策略如下：

使用之前所述训练出的GRU流行度预测模型，对第二天的流行度进行预测，并按流行度对镜像进行排名。另外，设置镜像存储空间大小W，并将本地边缘节点服务器内的镜像，按预测出的镜像流行度也进行从高到低排序。之后，按流行度从高到低的顺序，将预存镜像，如图4-6所示，依次进行如下过程。

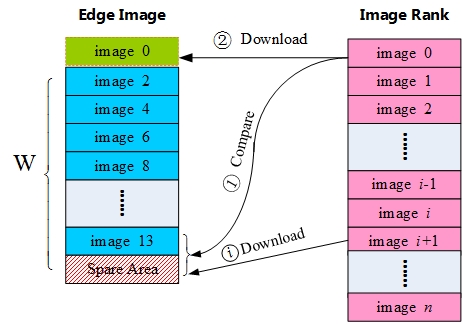


图4-6 基于流行度的缓存替换步骤

**步骤1：**如果存储空间足够，则直接存储；

**步骤2：** 倘若空间不足，则执行镜像置换策略，即从本地镜像排序的末尾起，统计本地流行度末尾且流行度低于预存镜像的镜像，直到找到的镜像大小总和超过预存镜像，将它们清除以存放预存镜像；

**步骤3：** 持续步骤二的操作，直到本地最末尾的镜像流行度，低于准备预存的镜像；

**步骤4：** 如若存储空间还有剩余，则继续按流行度高低，依次存入空间足够的镜像，直至遍历完成所有镜像。

显然，通过以上操作可以有效地减少比较次数，提高镜像缓存替换的响应时间。具体伪代码如下所示：

|  |
| --- |
| **算法3** 基于镜像流行度的缓存替换算法 |
| 1：设置本地镜像缓存存储空间大小W，本地缓存镜像集L，总体镜像集A，镜像缓存替换集R，镜像缓存添加集P；  2：对总体镜像集A进行排名结果的降序排序，得到已排序总体镜像集A’；  3：**for** i按A’的排序顺序 **do**  4： 对本地缓存镜像集L进行排名结果的升序排序，得到已排序本地缓存镜像集L’；  5： **for** j按L’的排序顺序**do**  6： **if** ( rank of i ) >= ( rank of j )  7： **if** i属于L’中  8： **continue；**  9： **else**  10： **if** ( size of L’ ) + ( size of i )  11： L’ = L’ + i；  12： **else**  13： 清空R，往R中从头按序添加L’中的镜像，直到( size of R ) >= ( size of i )；  14： L’ = L’ - R；  15： L’ = L’ + i；  16: **end if**  17: **break;**  18: **end if**  19: **else if** ( size of L’ ) < W  20: 清空P，往P中按此刻的i开始依序添加A’中的镜像，直到( size of P ) + ( size of L’ ) >=W；  21: L’ = L’ +P；  22: **break；**  23： **end if**  24： **end for** |
| 25：**end for** |

* 1. 本章小结

本章首先分析与镜像流行度相关的因素；然后，采用基于GRU的镜像流行度预测，将提取的特征因素放入模型训练；最后，提出基于流行度预测的缓存替换策略，使得充分利用边缘的存储空间。

1. 系统设计与实现

本章主要介绍基于Kubernetes和Docker技术的边缘计算云平台的设计与实现过程。首先，介绍了当前传统云计算平台应用于边缘计算存在的一些不足之处，然后，提出了基于Kubernetes Federation的边缘计算云平台，并给出具体的架构设计，以及工作的流程。最后，先简要介绍了开发使用的软硬件环境，再详细描述了各模块的设计与实现。

* 1. 传统云计算平台介绍

对于传统的云计算平台而言，无论是OpenStack还是Kubernetes，其设计定位都是在单一集群内，进行调度和计算存储操作。然而，对于边缘计算云平台而言，基于其自身的特性，除了存在中心云之外，还有若干边缘云。对于每一个边缘云，实际上也都是一个完整独立的云集群，只是由于边缘的设计定位，使得边缘云的资源相对于传统云中心显得十分的轻量化。因此，对于边缘计算云平台而言，需要满足云平台能够跨集群操作的问题。此时，仅使用一般的OpenStack或者Kubernetes显然不能满足平台的需求。

* 1. 边缘计算平台总体架构

对于边缘计算平台，除了需要满足跨集群的调度问题，还需要兼顾边缘节点资源轻量化的特点。因此，考虑到平台更加面向于提供具体的软件平台服务，以及Docker容器相比于虚拟机也更加轻量化的特性，本文选择使用Kubernetes作为基础的底层云平台。之后，本小节先介绍边缘计算云平台的架构设计，接着描述了其中使用到的Kubernetes Federation的简要情况，最后说明了本平台具体的工作流程。

* + 1. 基于Kubernetes 的边缘计算云平台架构设计

首先，相比于传统云计算平台都是由控制节点向计算节点发送任务请求的自上而下的模式。对于边缘计算的云平台设计，一直存在两种可行的选择方案。一是，与传统云计算模式相同的，用户先向中心云计算平台发送任务请求，之后再通过统一的特定的选择机制，由中心云计算平台向选定的边缘计算平台转发任务请求的自上而下的模式；而另一种就是，与之相反的自下而上的模式，用户先将任务请求发送到就近的边缘计算平台，之后再由边缘计算平台通过特定的选择机制，选择是否将该任务请求转发到中心云计算平台，或者是其余的边缘计算平台进行处理。考虑到用户与就近边缘计算平台直接进行任务请求的发送，比传统的自上而下模式，更能减少不必要的网络带宽消耗。因此，本文边缘计算云平台架构设计采用自下而上的任务请求分发模式。

其次，本文基于Kubernetes的容器编排系统，设计了基于Kubernetes 的边缘计算云平台。对于边缘计算云平台系统，由于每个边缘云平台，从某种意义上而言就是一个单独的云平台，只是相比于传统的云平台，其计算节点数，计算能力，存储能力都较之有大幅度的性能下降。因此，对于边缘计算云平台的设计，首先需要解决的就是上面所述的跨集群问题。对于Kubernetes的跨集群问题，由于Kubernetes自1.3版本开始引入集群联邦（cluster federation）的概念，用于对多个Kubernetes集群进行统一管理。因此，本文采用了Kubernetes Federation的方案来解决跨集群这个棘手的问题。

基于此，基于Kubernetes 的边缘计算云平台的底层设计基本完成。接着，在底层设计的基础之上，本文需要完成之前介绍的两种方法，即基于双边博弈的计算卸载方法和基于流行度预测的镜像缓存方法，并将其接入总体系统的设计。

对于基于双边博弈的计算卸载方法，其实也就是需要在总体体统中的实现一个自定义的Kubernetes调度器；而对于基于流行度预测的镜像缓存方法，同样也可以理解成需要在总体体统中的实现一个额外扩展的Kubernetes缓存器。其中，由于Kubernetes平台本身自带了默认的调度器，并对平台中的一些组件存在一些关联。因此，对于调度器的设计，除了需要认真学习Kubernetes的原理之外，还需要对其平台实现的原码进行学习，进而在了解默认调度器的原理和实现之后，实现自定义的基于双边博弈的调度器。相比于自定义调度器的设计，额外扩展的缓存器的设计明显简单不少。由于缓存器是Kubernetes本身不具备的功能，因此设计上只需将其当成Kubernetes的新插件，用以辅助总系统提高性能。



图5-1 边缘计算云平台架构

基于如上所述，本文基于Kubernetes 的边缘计算云平台架构设计可以分成以下三部分，如图5-1所示。一是，Kubernetes Federation平台；二是，边缘计算云平台调度器模块；三是，边缘计算云平台缓存器模块。其中，在缓存器模块内，又分为数据爬虫子模块、特征预处理子模块、模型训练子模块以及缓存替换子模块。

* + 1. Kubernetes Federation简介

联邦（Federation）[52]是Kubernetes自v1.3开始引入的概念，用以实现多Kubernetes集群统一管理的简易化。为此，其提供了两个主要特性用以实现多集群的管理。

1. 跨集群资源同步：联邦提供了多集群之间保持资源同步的能力。比如，联邦可以确保在联邦集群下，通过Deployment部署的服务在多个不同的集群中始终同时存在，并且保持一致。

2. 跨集群服务发现：联邦提供了在所有集群的后端自动配置DNS服务和负载均衡的能力。通过此能力，确保通过DNS记录可以访问同一联邦集群下，不同Kubernetes集群里的后端服务。

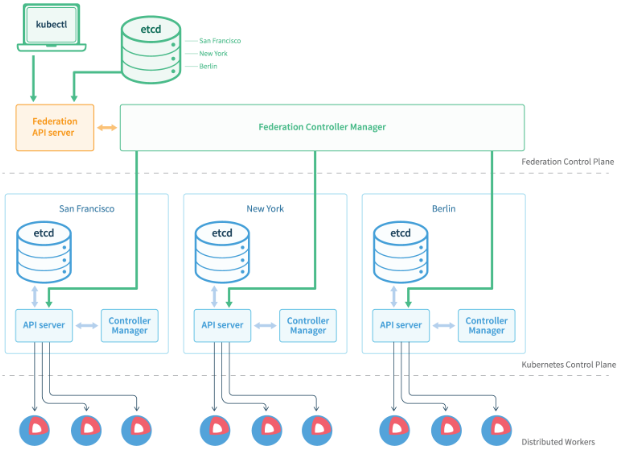


图5-2Kubernetes Federation架构模型[51]

基于此，联邦的设计思路采用了解耦和分层的思想。如图5-2所示，联邦的控制平台位于联邦集群内所有Kubernetes集群之上，同时所有的Kubernetes集群独立运行，互不牵连。Kubernetes集群相互间也互不知道对方的存在，甚至联邦控制平台的存在，从而实现了解除耦合的能力。而每个Kubernetes内部还是高耦合的状态。从某种角度上看，其实可以将联邦中控制平台与Kubernetes集群的关系，理解成Kubernetes集群内控制节点Master和计算节点Node之间的关系。用户通过使用“控制节点”，进而将任务分配给其下属的“计算节点”去执行。因此，Kubernetes联邦也可以看成是一种两层云结构的平台，只是上层之间耦合度较低而已。

* + 1. 工作流程

通过以上介绍基于Kubernetes的边缘计算云平台架构设计，以及Kubernetes联邦，如图5-4所示，我们可以很容易的得出基于Kubernetes 的边缘计算云平台的工作流程。



图5-4 边缘计算云平台工作流程

首先，用户将所需要的任务请求发送给离用户最近的边缘计算平台的服务器；接着，本地的边缘计算平台的服务器通过其自身的联邦控制平台，通过基于双边博弈的计算卸载方法设计的自定义调度器，将该任务请求，转发给中心云计算平台或者其余的某一边缘计算平台，亦或是保留在本地的边缘计算平台；之后，对应的边缘计算平台再将其分发到具体的某一边缘计算的节点服务器上；同时，通过基于流行度预测的镜像缓存方法设计的边缘计算云平台的缓存器，对相应的边缘计算的节点服务器进行镜像的缓存替换，使得节点服务器能缓存最合理的镜像；最后，在该相应的边缘计算的节点服务器上，更加便捷迅速地启动相应要求的pod，以满足用户的使用需求。

* 1. 基础环境

操作系统： Windows 7 Professional x64， CentOS 7.3

Ubuntu14.04.4 LTS桌面版

开发工具： JetBrains PyCharm 2017.2.3 x64, LiteIDE X32.1, Eclipse Java Oxygen

开发语言： Golang1.9，Python3.5，Java 8

第三方库： TensorFlow1.8.0， Matplotlib3.0.3

Numpy1.14.3， pandas0.22.0，sklearn0.19.1

* 1. 边缘平台的基础搭建

本项目边缘计算云平台的搭建，采用基于Kubernetes的方式，即在Kubernetes源码的基础上进行开发。因此，首先需要在每个边缘的云集群上，搭建Kubernetes。之后再搭建Kubernetes联邦以实现跨集群的功能。

* + 1. 边缘集群上Kubernetes的搭建

1. 修改系统配置

关闭防火墙服务与禁用开机启动项，以及关闭SELINUX选项。

# systemctl stop firewalld

# systemctl disable firewalld

# setenforce 0

# sed -i 's/SELINUX=enforcing/SELINUX=disabled/g'/etc/sysconfig/selinux

1. 安装Docker

$ yum install docker

1. 安装Kuberadm与Kubelet

在每个节点安装kubeadm和kubelet；

$ cat <<EOF > /etc/yum.repos.d/kubernetes.repo

[kubernetes]

name=Kubernetes

baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86\_64

enabled=1

gpgcheck=1

repo\_gpgcheck=1

gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg

https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg

EOF

安装kubeadm, kubelet, kubectl, kubernets-cni，并为kubelet服务设置开机启动。

$ yum install -y kubelet kubeadm kubectl kubernetes-cni

$ systemctl enable kubelet.service

1. 初始化集群

使用kubeadm初始化集群，选择其中一个Node作为Master。

1. 安装Pod网络组件

使用flannel 网络模式；

$ wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel-rbac.yml

$ wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

部署网络组件。

$ kubectl create -f kube-flannel-rbac.yml

$ kubectl apply -f kube-flannel.yml

* + 1. 负载均衡器的实现

Kubernetes的服务发现为Kubernetes集群中具有相同功能的容器应用提供了一个统一的入口地址，以便于当Pod出现故障，需要重启启动相同功能的新Pod时，Pod启动到与原先不同的物理节点而造成的需要变更访问地址。具备服务（Service）功能之后，访问相关的Pod应用开始使用服务的地址，由于服务的地址不会变更，因此在与之相关的容器中也可实现不需要经常更改地址的操作。至于，Pod所在物理节点的变更进而导致的与服务地址的关联，则由系统后台在Etcd上实现了相关绑定关系的变更。从而，实现了集群外部访问，可以通过服务动态查找到Pod的功能。

对于服务的访问方式，通常有三种类型：ExternalIP、NodePort和LoadBalancer。对于ExternalIP和NodePort，都是直接通过人工设定的方式，将服务暴露出去，使得集群外部得以访问。与之不同的是，LoadBalancer则是通过设置LoadBalancer映射到云服务商提供的LoadBalancer地址。通常，这种方法仅用于在公有云服务提供商的云平台上设置服务的场景。

因此，显而易见，类型为LoadBalancer的服务在Kubernetes中并没有直接支持。然而，在Kubernetes联邦的搭建过程中，又需要为每一个Kubernetes集群实现支持 LoadBalancer 服务，从而实现DNS服务发现，进而满足联邦的跨集群服务发现功能。

对于本地的负载均衡实现，本文经过调研，决定采用Metalb[53]实现。对于Metalb，首先，其本身是运行在Kubernetes内部，通过监听服务对象的变化，一旦察觉有新的LoadBalancer服务运行，并且没有可申请的负载均衡器之后，就会完成地址分配和地址广播两部分的工作。

MetalLB的部署很简单，支持helm和yaml两种安装方法。本文直接使用 yaml文件部署：

$ kubectl apply -f https://raw.githubusercontent.com/google/metallb/v0.7.1/manifests/metallb.yaml

部署完成后需要为LoadBalancer服务选择一个特定的IP地址池，此时可以通过采用ConfigMap来创建。

* + 1. Kubernetes Federation的搭建

1. 安装kubefed

首先，选择一个集群作为本地边缘集群，这个集群将运行组成联邦控制面板的所有组件，再进行下载与解压。

$ curl -LO https://storage.cloud.google.com/kubernetes-federation-release/release/v1.9/federation-client-linux-amd64.tar.gz

$ tar -xzvf federation-client-linux-amd64.tar.gz

1. 配置context

在准备配置联邦集群的本地边缘集群的Master中配置剩余集群的 context。使此节点能通过切换 context 连接到不同的集群；

再创建本地集群的 kubeconfig 文件，之后将剩余集群的kubeconfig 文件与之合并。

1. 部署Etcd

先安装helm，再下载helm charts仓库，再部署etcd-operator

$ helm install --name etcd-operator stable/etcd-operator --set rbac.install=true,rbac.apiVersion=v1,customResources.createEtcdClusterCRD=true

1. 设置CoreDNS作为集群联邦的DNS提供商

$ helm install --name coredns -f Values.yaml stable/coredns

…

1. 初始化Federation控制平面

$ kubefed init federation \ # 联邦的名字

--host-cluster-context=kubernetes \ # 主集群的context名字

--dns-provider=coredns \ # DNS服务提供商

…

* 1. 边缘计算云平台调度器的设计与实现

对于Kubernetes中自定义调度器的设计与实现，首先我们介绍了其内部的调度的工作流程。之后，基于这个工作流程，我们设计了基于双边博弈匹配的自定义调度器。最后，采用Golang的开发语言，并采用Docker的形式，用Pod实现了该自定义调度器。

* + 1. 调度的工作流程

由于Kubernetes调度程序，甚至Kubernetes体系结构中的几乎所有组件都是非常松散耦合的。因此，在设计Kubernetes的自定义调度器时，不必要去考虑在新创建一个pod时，是否需要一些API去调用其余的用来满足创建pod的组件。基于此，对于设计Kubernetes的自定义调度器，只需满足以下的需求：为集群中的所有pod找到对应的节点，并让Kubernetes的apiserver知道对应的绑定关系。之后，apiserver和kubelet将负责将剩下的工作，即启动工作的容器。

因此，如图5-1所示，我们可知调度工作流程的具体情况为：

1. 通过REST API创建一个pod，并通过apiserver将其处于未绑定的状态保存到etcd；
2. 调度器通过监听检测发现一个新的尚未绑定节点的pod；
3. 调度器开始调度，从而找到最适合此pod的节点；
4. 将pod绑定到的对应节点通过apiserver通知给etcd，进而将新的状态保存到etcd；
5. 特定节点上的Kubelets监听检测到有新的pod调度过来，通过container runtime运行该Pod；
6. Kubelets通过container runtime取到Pod状态，并更新到apiserver中。



图5-1 调度的工作流程

* + 1. 调度器模块的设计

通过以上的分析，不难得出，调度器的设计需要满足如下三步核心步骤，如图5-2所示：



图5-2 调度器模块设计图

**步骤1：** 通过apiserver watch API，来监控发现群集中未绑定的 pod；

**步骤2：** 通过之前介绍的基于双边博弈的计算卸载方法，找到pod的最佳匹配节点；

**步骤3：**通过apiserver将绑定的信息发送给Etcd。

* + 1. 调度器模块的实现

对于调度器的实现，本项目采用了Golang作为开发语言。首先，介绍调度器的基本操作信息。主要信息如下：

//将调度器的名称定为matching-scheduler，便于后续识别

const schedulerName = "matching-scheduler"

//根据传递的内容更新pod的条件PodCondition

type PodConditionUpdater interface {

Update(pod \*v1.Pod, podCondition \*v1.PodCondition) error

}

// 删除pod，以及更新pod注释

type PodPreemptor interface {

GetUpdatedPod(pod \*v1.Pod) (\*v1.Pod, error)

DeletePod(pod \*v1.Pod) error

UpdatePodAnnotations(pod \*v1.Pod, annots map[string]string) error

}

//删除pod，之后根据传递的pod创建一个新pod

type PodWriter interface {

UpdatePod(oldPod, newPod \*v1.Pod) error

}

// 调度器监视新的尚未安排的pod，其试图找到适合的节点并将绑定写回api服务器

type Scheduler struct {

clientset \*kubernetes.Clientset

}

然后，对应设计中的三步骤可知，调度器的第一个重要部分是，通过连续的循环去监听观察新创建的pod。此时，我们可以在pod上使用Kubernetes client-go SDK的Watch方法。该方法为我们提供了一系列足以覆盖（range）所有监听范围的事件（events）。

func (s \*Scheduler) SchedulePods() error { //在连续循环中观察新创建的pod

watch, err := s.clientset.CoreV1().Pods("").Watch(metav1.ListOptions{

FieldSelector: fmt.Sprintf("spec.schedulerName=%s, status.phase = ‘Pending’ ", schedulerName),

})

…

for event := range watch.ResultChan() {

if event.Type != "ADDED" {

continue

}

p, ok := event.Object.(\*v1.Pod)

…

fmt.Println("found a pod to schedule:", p.Namespace, "/", p.Name)

…

}

如上所示，Watch函数将FieldSelector作为输入参数。接着，我们在该选择器中添加了两个部分。status.phase = ‘Pending’表明，它在等待一个包含Pending状态对象的新事件。倘若status.phase不是Pending状态，那就意味着该pod已经绑定好了。另一部分，则是一个需要我们规范的scheduler\_name，即spec.schedulerName=matching-scheduler。由于Kubernetes能够在集群中使用多个调度器，并且pod规范允许包含调度器的名称。因此，如果设置了调度器名称，则通过相应的期望调度器将该pod绑定到对应节点。同时，如果没有设置调度器名称，则表示使用Kubernetes默认的官方调度器。基于此，需要对scheduler\_name进行规范。同时，当通过watch函数提供的方法进行覆盖所有的事件时，由于我们只想关注新添加的pod，因此还需要过滤出ADDED事件。

func (s \*Scheduler) findFit() (\*v1.Node, error) { //寻找符合的最佳节点

nodes, err := s.clientset.CoreV1().Nodes().List(metav1.ListOptions{})

… //基于双边博弈的计算卸载方法

}

接下来的部分是为pod找到一个最佳的节点。在我们的设计中，我们将从节点列表中进行筛选节点，从而与pod进行双边博弈匹配。需要注意的是，此处与基于双边博弈的计算卸载方法中提到的概念名词稍有不同，此处的pod即对应任务，而节点即对应服务器。

func (s \*Scheduler) bindPod(p \*v1.Pod, randomNode \*v1.Node) error { //让apiserver获知绑定信息

return s.clientset.CoreV1().Pods(p.Namespace).Bind(&v1.Binding{

…

})

}

接着，在我们为我们的pod找到一个节点之后，剩下的唯一重要的事情就是让apiserver获知绑定的信息。此时，我们通过Bind方法完成它，如上图所示。

然后，在Kubernetes集群中部署该调度器。鉴于，Docker容器容易维护的特性，本项目采用包含调度器代码的容器创建标准部署。首先，基于上述的实现，采用Dockerfile构建Docker镜像；然后，通过编写部署的yaml文件，如下图所示，编写deployment.yaml文件。

apiVersion: apps/v1

kind: Deployment

metadata:

name: matching-scheduler

labels:

app: matching-scheduler

spec:

replicas: 1

selector:

matchLabels:

app: matching-scheduler

template:

metadata:

labels:

app: matching-scheduler

spec:

serviceAccount: matching-scheduler

containers:

- name: matching-scheduler

image: bupt632edgecomputingmatching/matchingscheduler

imagePullPolicy: Always

最后，通过kubectl的命令操作，运行带有基于双边博弈匹配matching-scheduler调度器的pod。

$ kubectl apply -f deployment.yaml

当该调度器pod正常运行之后，采用此调度器的新pod需要在其生成的yaml文件中标明使用matching-scheduler调度器。

spec:

# 选择使用项目自定义的调度器 matching-scheduler

schedulerName: matching-scheduler

* 1. 边缘计算云平台缓存器的设计与实现

对于边缘计算云平台缓存器的设计与实现，由于Kubernetes本身不具备类似节点缓存器的功能，因此该节点缓存器对于原生的Kubernetes平台而言是类似插件的工具。接着，基于这个原因，本节设计并实现了基于流行度预测的节点缓存器。

* + 1. 数据爬虫子模块



图5-3 爬虫子模块设计图

该子模块流程图如图5-3所示。该子模块负责爬取百度指数上一段时间内，特定地区、特定对象的搜索指数。由于搜索指数最早的数据日期为2011年1月1日，因此，爬取的数据不能早于这个时间点。倘若，开始时间超过最早的数据日期，则会导致数据的不准确。使用时，通过在python文件中写入需要爬取的时间段，以及城市的相应编码，然后向指定的URL（即百度指数）发送请求。接着，获取其返回的JSON文件，通过解析得到每项的具体数据。然而，由于关键词对应的每日指数数据，是加密的密文。所以，需要对相应的密文进行解密，进而获得真实的数据。最后，将数据导入到指定的csv文件中。

另外，由于本项目服务于Kubernetes平台，为了更好的匹配平台的功能特性。因此，将爬虫程序写入了基础镜像中，进而制作了专门服务于本项目的爬虫镜像buptspider。之后，通过在本地边缘节点启动pod完成功能的实现。

* + 1. 特征预处理子模块



图5-4 特征预处理子模块设计图

由模块的设计可知，该子模块流程图如图5-4所示。该子模块先获取由爬虫子模块采集而来的csv文件，对异常数据进行前后均值替换。之后，将处理过后的数据通过pandas导入到Python程序中，对日期数据进行特征的预处理操作，也就是相应的进行One-Hot编码。其次，由于在同一时间不同地区不同镜像的流行度数据分散存储，因此需要对其镜像数据的拼接工作。通过Numpy将异常数据处理后的csv文件的所有数据导入，再对数据进行二维数组操作，得以实现数据的拼接。最后，对数据中的所有镜像流行度数据，通过sklearn的MinMaxScaler进行最大-最小归一化（Normalization）操作。从而，得以得到的输入为57维的数据。同样，本子模块的预处理程序也写入了基础镜像中，制作出了相应的镜像文件buptdatapreprocessing。

**def** prepare\_data(): #实现数据的预处理

dataset = read\_csv(DATAPATH) # 加载数据集

enc = preprocessing.OneHotEncoder() #One-Hot编码

scaler = MinMaxScaler(feature\_range=(-1, 1)) # 最大-最小归一化特征函数

* + 1. 模型训练子模块



图5-5 基于GRU的流行度预测网络结构图

在基于GRU的流行度预测模型中，本文设计了一层的GRU网络模型，利用输入时间而转换得到的星期、是否节假日、当前节假日长度以及在节假日中所处位置这些特征，进而得到该镜像的流行度值。如图所示，通过TensorFlow实现GRU网络和全连接层（Fully Connected Layer）。其伪代码实现如下：

**def** gru\_layer(self, input, dim) #实现GRU网络结构

#实现全连接层

**def** fc\_layer(self, inputs, in\_dim, out\_dim, keep\_prob=1.0, initializer\_range=0.1)

**def** build\_net(self) #实现网络结构

**def** train(self) #实现训练

表5-1 输出层超参数设置

|  |  |  |
| --- | --- | --- |
| **参数名称** | **参数作用** | **参数值** |
| **optimizer** | 优化器 | Adam |
| **loss** | 损失函数 | MSE（Mean Squared Error，均方误差） |
| **learning \_rate** | 学习率 | 0.01 |
| **n\_epoch** | 迭代次数 | 50 |
| **batch\_size** | mini-batch每批次大小 | 64 |

最后，同样将本子模块基于GRU的流行度预测模型代码整合进基础镜像，制作出相应的镜像文件buptgrutrain。

* + 1. 缓存替换子模块

缓存替换子模块的主要功能就是利用从模型训练子模块（基于GRU的镜像流行度预测算法）中获得的流行度预测值更新本地的镜像缓存。关于基于流行度的缓存替换算法的具体实现已经在第四章中给出了详细的说明，因此此处不再赘述。为了便于缓存替换子模块在各个节点的部署，本模块采用Dockerfile文件进行构建对应的镜像，其代码如下：

FROM java:8

MAINTAINER sheng

RUN mkdir -p /usr /src /cacheReplacement

COPY . /usr /src /cacheReplacement

WORKDIR /usr /src /cacheReplacement

RUN javac cacheReplacementGRU.java #为缓存替换算法的实现文件

EXPOSE 8080

CMD ["java", " cacheReplacementGRU "]

* 1. 本章小结

本章通过分析现有的平台，并结合边缘轻量化的特点，首先提出采用Kubernetes Federation作为边缘计算云平台的基础框架。然后，基于双边博弈的计算卸载方法设计并实现了边缘计算云平台的调度器模块；接着，基于流行度预测的镜像缓存方法设计并实现了边缘计算云平台的缓存器模块。

1. 实验分析与系统测试

本章主要通过对系统两大模块内基于双边博弈的计算卸载方法，基于GRU的镜像流行度预测方法，以及基于流行度预测的镜像缓存策略的实验分析与结果研究，从而实现对本边缘计算云平台调度器模块和缓存器模块的系统功能测试及性能评估。具体地，本章首先介绍了本边缘计算云平台的搭建环境，也即实验环境。其次，通过三个角度对基于双边博弈的计算卸载方法进行效果的衡量。然后，通过在验证集中Loss损失函数和随迭代次数的变化曲线，以及在测试集中将预测值与真实值进行对比的方法，评估基于GRU的镜像流行度预测方法的有效性。最后，再将基于流行度预测的镜像缓存策略与传统的缓存替换算法LRU、LFU进行命中率的比较，用以检测方法的性能。

* 1. 实验环境与数据集

本章实验以搭建本文所述的边缘计算云平台为基础，为此构建一个基于Kubernetes的云-边一体化的边缘计算云平台。其中，平台包括一个中心云，与三个边缘云，每个云内又由三个终端组成。由于边缘终端设备的资源紧张，而实验中又有对边缘云数量的更高需求。为此，本项目在原本三个边缘云的基础上，通过将富余服务器进行VMWare虚拟化，从而得到多个边缘终端设备，再将这些虚拟化设备通过桥接的网络模式获得外部的IP地址，进而实现了多个边缘云以满足实验变量的需求。具体地，表6-1列出了组成系统的设备配置情况。其中包括了云平台比较关注的CPU、以及内存的详情。

表6-1 系统的设备配置情况

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **CPU** | **内存** | **数量** |
| 1 | 32核，2.60GHz | 64GB | 2 |
| 2 | 8核，3.40GHz | 8 GB | 8 |
| 3 | 8核，3.60GHz | 8 GB | 3 |
| 4 | 16核，2.00GHz | 24 GB | 2 |
| 5 | 2核，2.13GHz | 2 GB | 1 |
| 6 | 8核，3.40GHz | 16 GB | 1 |

鉴于本边缘计算云平台为通用云平台的设计定位，以及百度指数数据的权威性和考虑影响因素的全面性，本章在进行基于GRU的镜像流行度预测方法和基于流行度预测的镜像缓存策略时，都采用百度指数的数据作为实验数据。将北京、上海、广州、深圳和杭州视为五个边缘云各自所在的范围，再通过对Docker镜像中流行度较高的镜像进行调研统计[55][56]得出40种使用量较高的镜像，故而在百度指数上获取北上广深杭五个地区，自2011年1月1日以来这40种镜像的指数数据，作为流行度的数值依据以及镜像请求数的数值依据。

* 1. 基于双边博弈的计算卸载方法的实验分析

因为基于双边博弈的计算卸载方法的设计目标是为了最大化的减少任务的完成时间，所以，本节主要通过基于双边博弈的计算卸载方法与其余的计算卸载方法在不同的场景、条件下任务完成时间的长度进行对比分析。为了减少测量时间的偶然性误差，本节实验将重复进行相同的五组实验，然后将五组实验中测量所得的完成时间进行均值处理，得到平均完成时间，进而以此作为卸载性能的衡量标准。

对于输入的测试应用程序，本实验选择了一个常见的机器学习应用，其可以分解成多个前后相关的子任务进行处理。同时，作为对比的参照系，本实验选取了四种计算卸载方法。具体为，仅本地边缘、本地边缘与中心云、随机选择以及非聚合匹配。而对于本文所提出的基于双边博弈的计算卸载方法，在本节中则简化称之为聚合匹配。

另外，考虑到集群内部还存在计算卸载的情况，因此本实验将简化边缘云平台，即每个边缘云只拥有一个计算节点。在前文所述的实验环境搭建完毕之后，经检测，此时的网络带宽为100Mbps。接下来，将分别以边缘云数量、应用输入大小以及应用数量的角度观察其对计算卸载方案的影响，并对五种计算卸载方案进行对比分析。

首先，本节将评估边缘云数量对平均完成时间带来的影响。为了消除不同类型设备的差异可能导致的结果误差，此处实验选择表6-1中类型为2的设备作为边缘云的类型，应用输入大小为431MB，而应用的数量仅为一个。

图6-1 边缘云数量对计算卸载方法的影响

在图6-1中，当边缘云数量为1时，意味着系统内只有中心云和一个边缘云；以此类推，当边缘云数量为2时，则意味着系统内有中心云和两个边缘云。从图6-1可知，仅本地边缘和本地边缘与中心云这两种卸载方法不受边缘云数量的影响，一直保持不变。其余的三种方法则都呈平均完成时间下降的趋势。这表明，随着边缘云数量的增加，能有效的减少时间消耗。这是因为随着边缘云的数量增多，中心云的计算压力能更容易的被边缘云缓解。然而，当边缘云的数量达到一定数量后，则平均完成时间不再随着边缘云数量的增加而减少。这是由于当前边缘云的数量已经足够缓解计算压力，继续增加的边缘云只能充当闲置资源。同时，对比图中呈下降趋势的三种计算卸载方法，可以明显看出，聚合匹配通过预先整合相关任务的分配，减少了更多不必要的通信时间花销。因此可知，在一定限度下，随着边缘的增加，聚合匹配最能有效的节省平均完成时间。

其次，考虑应用程序输入数据的大小对计算卸载方案的影响。本实验选择三台类型2和两台类型3，其余类型各一个的边缘设备组合。并且，应用的数量不变，还有一个程序。应用程序输入的大小，则依次为51MB、245MB、431MB、623MB、809MB等。

图6-2 应用输入大小对计算卸载方法的影响

图6-2表明，所有的计算卸载方案都与应用输入的大小呈正相关。并且随着输入量的增加，各方法的差异性变得更大。特别地，当输入量到达某一值时，仅本地与中心云的增长趋势会变得平缓。而其余四种方法，将保持相同的完成时间。这是由于当应用的输入到达一定限度后，任务则必须卸载到中心云进行处理。并且，不难发现，在到达该特定值之前，聚合匹配也还一直是五种中最佳的卸载方案。

然后，本文再考虑相同的应用，随着数量增多对计算卸载方法产生的影响。本实验服务器的选择于上一实验一致。同时，应用程序输入的大小为431MB，并且，边缘云的数量为1。

图6-3 应用输量对计算卸载方法的影响

正如图6-3所示，仅本地边缘需要比别的方案更长的时间。对于本地边缘与中心云，起初增长的相对比较快。然而，过了一段时间后，当达到了某一特定值时，其逐渐有靠近非聚合匹配和聚合匹配的趋势。这是由于一开始只有中心云能缓解本地边缘的计算压力。对比其它拥有边缘云的卸载方案，其显然会使得平均完成的增加。然而，随着任务数量的增长，剩余的边缘云也将会有计算压力。此时，边缘云之间的计算卸载将不再拥有减少平均完成时间的优势。此时，聚合匹配仍处于效果最优的转态。另外，从图6-2与图6-3可知，虽然总体的趋势相近，但是对于直接增加应用输入的大小与增加应用的数量相比，还有存在显著的差异。

通过以上分析三种变量对于五种计算卸载方法的影响，不难发现，本文提出的基于双边博弈匹配的计算卸载方法，也即聚合匹配，总是能实现最短的平均完成时间。基于此，该计算卸载方案的有效性被证实。从而，也表明了系统平台调度器其性能满足设计需求，即有效的缓解了边缘计算中计算资源不足的问题。

* 1. 基于GRU的镜像流行度预测方法的实验分析

本节通过对基于GRU的镜像流行度预测方法，观察验证集中Loss损失函数和随迭代次数的变化曲线，以及在测试集中将预测值与真实值进行对比的方法，进而评估模型的准确性。

首先，本方法的数据集是2011年1月1日至2018年5月14日之间，百度指数上五种地区40种镜像类型的指数数据538200条，对于不用地区不同镜像类型则有2691条。之后，再将按8:1:1的比例将数据分为训练集、验证集和测试集。

在通过TensorFlow实现的了该基于GRU的镜像预测模型后，对于某一种特定的镜像，按13天预测1天的规律，输入预处理过后的流行度特征数据到模型中，经过50次的迭代次数，可以得到图6-4所示的北京地区MySQL镜像的Loss函数随迭代次数变化效果和对应的图6-5左上角的随迭代次数变化效果。

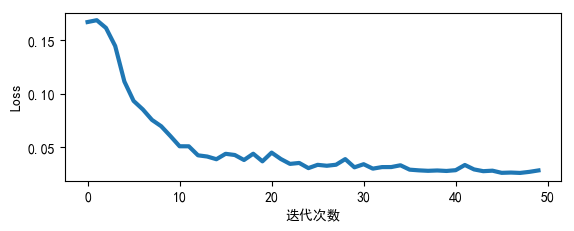


图6-4 Loss函数随迭代次数变化效果

对于Loss函数，采用的是均方误差（mean square error，MSE）作为衡量预测模型的依据。可以看出随迭代次数的增加，Loss值逐步地下降，当到达14次左右时，Loss函数下降趋势减缓，开始逐渐稳定下来。

同时，本文也采用（确定系数）[54]作为衡量预测模型正确率的依据。的表达公式如下：

(6-1)

其范围为0~1之间，当为1时说明完全拟合，越接近1说明拟合效果越好。从图6-5(a)的曲线可以看出，拟合数值逐渐稳定在0.8的附近。结合两图，说明预测结果在验证集验证中的效果不错。其余地区、镜像类型的效果也与此效果类似。

之后，再将模型与测试集中的数据进行对比分析效果。如图6-5(b)(c)(d)所示，依次为北京地区MySQL镜像，北京地区CentOS镜像，广州地区Nginx镜像的真实值与预测值数据对比。图中，蓝色实线为真实值，黄色虚线为预测值，X轴为测试样本，Y轴为镜像流行度数据。

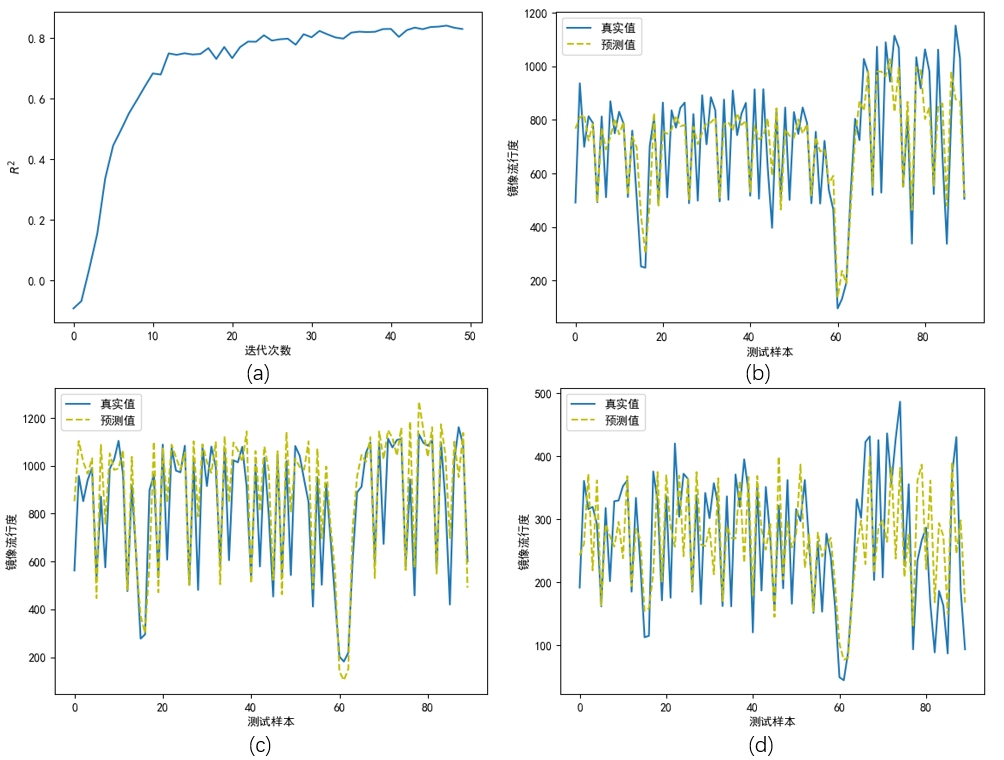


图6-5 R^2随迭代次数变化效果，以及测试集内真实值与预测值的比较

其中，测试样本是从测试集中按随机顺序选择的若干数据样本。从图中可以看出，本预测模型能够较好地预测出镜像流行度的变化趋势，因此为后续的基于流行度预测的镜像缓存替换策略提供了良好的替换依据。

* 1. 基于流行度预测的镜像缓存替换策略的实验分析

通过上节的实验分析，验证了基于GRU的镜像流行度预测方法的良好性能。本节将采用基于GRU预测的流行度结果作为参考标准，与传统的缓存替换策略LRU、LFU进行缓存命中率的比较。其中，缓存命中率（Cache Hit Ratio）是指本地镜像请求的命中率。本实验采用14天作为计算缓存命中率的周期。

图6-6 不同算法缓存命中率和缓存容量的对比

实验采取测量多处时间内，缓存命中率的数值，进而通过均值处理，得到平均的缓存命中率作为衡量性能的依据。实验结果如图6-6所示，从图中可以看出，当缓存空间较小时，增大缓存空间对于缓存命中率的提升效果明显。但当达到一定程度时，如图6-6中缓存容量比为0.005时，继续增大缓存空间则对于提升缓存命中率的效果降低。

并且，从图中可以看出，在相同缓存容量比的情况下，基于流行度预测的镜像缓存方法（即图中标注为b-GRU的方法）的缓存命中率要明显优于LRU和LFU的缓存命中率。同时，当缓存命中率相同时，基于流行度预测的镜像缓存方法则要比LRU和LFU更加的节省缓存空间，进而降低对设备需求的标准。

同时，为了更好的利用边缘的存储空间资源，本实验将选取缓存容量比为0.005的缓存命中率作为比较参考的依据，对比具体日期内缓存命中率的变化情况。

基于2690天的的数据观测，对b-GRU、LRU和LFU算法的缓存命中率对比，发现了其中存在在一定范围内的周期波动。特别地，对于常规时间，和带有长假期时间分别作了对比分析。为了更好地观测例如春节和十一这种7天的长假期，本实验采取观测两周的缓存命中率的变化情况。

首先，在2690天中选取连续的两周的常规时间进行观察测量，结果如图6-7所示。可以看出，在观测时间内，b-GRU的缓存命中率，一直维持在80%上下浮动，没有较大的差别。而LRU和LFU则都出现了较大幅度的“W”型震荡。通过观察可知，LRU和LFU缓存命中率较低的时间都为周六和周一，这种工作日与非工作日交替变化的时间。通过分析可知，这种时间下，采用LRU和LFU这种静态的镜像流行度预测，不能很好的体现出流行度的动态变化差异。

图6-7 正常双休日的两周内，缓存命中率的对比

其次，在2690天中选取带有长假期的连续两周进行观察测量，本实验采用2017年1月26日至2017年2月8日的这段时间，其中1月27日为春节假期的第一天，测量结果如图6-8所示。由图像可知，在观测时间内，b-GRU的缓存命中率，依旧一直维持在0.8上下浮动，没有较大的差别。而LRU和LFU，则相比于正常双休日的两周内“W”型震荡有了一些区别。首先，从图中可知，出现较大预测误差的时间依旧为工作日与非工作日交替的第一天。而且，相比于正常的双休日误差，图中周五的误差更加严重，其中LFU的缓存命中率已经跌至20%以下，在下一个周五，LRU的缓存命中率更是已经接近10%的水平。因此，不难看出，LRU和LFU对于流行度动态变化比较强烈时，其缓存更替方法的依据不能作为很好的参考标准。

图6-8 存在长假期的两周内，缓存命中率的对比

基于以上两个实验的结果，表明基于流行度预测的镜像缓存方法能有效地提高镜像的缓存命中率。在满足相同镜像缓存命中率的条件下，能够很好的节约缓存资源。同时，也表明了系统平台缓存器其性能满足设计需求，即有效的缓解了边缘计算中存储资源不足的问题。

* 1. 本章小结

本章基于平台功能实现后的系统，对系统分别依次进行基于双边博弈的计算卸载方法、基于GRU的镜像流行度预测和基于流行度预测的镜像替换方法的实验分析。其中，计算卸载方法和镜像替换方法采用了对比实验。通过验证这三种方法的有效性，也表明了系统调度器和缓存器的性能到达了预期的目标。

1. 总结与展望
   1. 本文总结



图7-1 本文设计流程

本文针对边缘计算场景下，计算资源和存储资源不足的现状，分别利用计算卸载和镜像缓存的思想，提出了基于双边博弈的计算卸载方法和基于流行度预测的镜像缓存方法。再通过与Kubernetes Federation平台结合，设计并实现了相应的边缘计算云平台的调度器和缓存器，从而实现了云-边一体化的边缘计算云平台，如图7-1所示。本文的主要创新点为：

首先，提出了基于双边博弈的计算卸载方法。该方法增加了多对一双边博弈匹配算法在其余边缘云和中心云上进行选择，相比于传统的单纯往中心云卸载的方法，有效的节约了时间成本，充分利用了周边空闲的边缘资源。通过缓解本地计算资源紧张，以及中心云的计算压力大的局面，有效地节约了时间成本。同时，将具有强联通特性的任务聚合在一起，更好地节约了任务等待时间。

其次，提出了基于流行度预测的镜像缓存方法。该方法相比于传统的采用LRU以及LFU的替换算法，使用基于流行度预测的模型，用于镜像的缓存，有效地节约了边缘上的存储资源，缓解了存储压力。通过定性分析与镜像流度关系紧密若干因素的关系，并根据这些关系选取使用时间（日期，星期，节假日）、镜像种类，以及使用地区（北上广深杭）作为特征，基于GRU算法，构建了镜像流行度的预测算法，并采用基于镜像流行度的缓存替换策略进行直接存储或置换，在满足高命中率的同时，有效地节约了存储空间。

* 1. 工作展望

本文提出了基于双边博弈的计算卸载方法和基于流行度预测的镜像缓存方法，进而设计并实现了边缘计算云平台的调度器和缓存器。通过第六章各自的对比实验分析，证明了通过使用本文提出的计算卸载方案，能有效地减少任务的完成时间；使用本文提出的镜像缓存方案，则能在保证高命中率的条件下，有效地节省边缘的存储空间。但是，本文也还存在一些可以改进的地方：

1）本文在基于流行度的镜像缓存方法中，采用的数据集是以天为单位进行采集的。因此，缺乏更加细粒度化的预测，在未来的工作中，可以考虑以小时为单位的采集，进而能使得预测更加精准，也能在存储资源有限的前提下，更好地提高用户的使用体验。

2）本文在缓存器模块里，还未实现完全的自动化镜像替换。由于爬虫程序尚未实现定期的自动爬取数据的操作，并且底层的数据存储也尚未实现数据库的方式。因而，在接下来的工作中，可以一方面实现底层数据库，另一方面，实现定期的爬虫机制。

参考文献

1. JoSEP A D, KAtz R A D, KonWinSKi A D, et al. A view of cloud computing[J]. Communications of the ACM, 2010, 53(4): 50-58.
2. Mell P, Grance T. The NIST definition of cloud computing[J]. Communications of the ACM, 2011, 53(6): 1-7.
3. Culler D E. The once and future Internet of everything[J]. GetMobile: Mobile Computing and Communications, 2017, 20(3): 5-11.
4. Anon. Cisco Global Cloud Index: Forecast and Methodology [EB/OL]. 2016–2021 White Paper, 2019-2-28.
5. 施巍松, 孙辉, 曹杰, 等. 边缘计算: 万物互联时代新型计算模型[J]. 计算机研究与发展, 2017: 1.
6. Hu Y C, Patel M, Sabella D, et al. Mobile edge computing—A key technology towards 5G[J]. ETSI white paper, 2015, 11(11): 1-16.
7. Bonomi F, Milito R, Zhu J, et al. Fog computing and its role in the internet of things[C]//Proceedings of the first edition of the MCC workshop on Mobile cloud computing. ACM, 2012: 13-16.
8. Gai K, Qiu M, Zhao H, et al. Dynamic energy-aware cloudlet-based mobile cloud computing model for green computing[J]. Journal of Network and Computer Applications, 2016, 59: 46-54.
9. Xu Z W. Cloud-sea computing systems: Towards thousand-fold improvement in performance per watt for the coming zettabyte era[J]. Journal of Computer Science and Technology, 2014, 29(2): 177-181.
10. 谢人超, 廉晓飞, 贾庆民, 等. 移动边缘计算卸载技术综述[J]. 通信学报, 2018, 39(11): 138-155.
11. Mach P, Becvar Z. Mobile edge computing: A survey on architecture and computation offloading[J]. arXiv preprint arXiv:1702.05309, 2017.
12. Mao Y, You C, Zhang J, et al. Mobile edge computing: Survey and research outlook[J]. arXiv preprint arXiv, 2017, 1701.
13. Liu J, Mao Y, Zhang J, et al. Delay-optimal computation task scheduling for mobile-edge computing systems[C]. //2016 IEEE International Symposium on Information Theory (ISIT). IEEE, 2016: 1451-1455.
14. Mao Y, Zhang J, Letaief K B. Dynamic computation offloading for mobile-edge computing with energy harvesting devices[J]. IEEE Journal on Selected Areas in Communications, 2016, 34(12): 3590-3605.
15. Zhang K, Mao Y, Leng S, et al. Optimal delay constrained offloading for vehicular edge computing networks[C]. //2017 IEEE International Conference on Communications (ICC). IEEE, 2017: 1-6.
16. Jia M, Cao J, Yang L. Heuristic offloading of concurrent tasks for computation-intensive applications in mobile cloud computing[C]. //2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2014: 352-357.
17. Kao Y H, Krishnamachari B, Ra M R, et al. Hermes: Latency optimal task assignment for resource-constrained mobile computing[J]. IEEE Transactions on Mobile Computing, 2017, 16(11): 3056-3069.
18. Yang L, Cao J, Cheng H, et al. Multi-user computation partitioning for latency sensitive mobile cloud applications[J]. IEEE Transactions on Computers, 2015, 64(8): 2253-2266.
19. Kamoun M, Labidi W, Sarkiss M. Joint resource allocation and offloading strategies in cloud enabled cellular networks[C]. //2015 IEEE International Conference on Communications (ICC). IEEE, 2015: 5529-5534.
20. Labidi W, Sarkiss M, Kamoun M. Energy-optimal resource scheduling and computation offloading in small cell networks[C]. //2015 22nd International Conference on Telecommunications (ICT). IEEE, 2015: 313-318.
21. Zhang H, Guo J, Yang L, et al. Computation offloading considering fronthaul and backhaul in small-cell networks integrated with MEC[C]. //2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2017: 115-120.
22. You C, Huang K. Multiuser resource allocation for mobile-edge computation offloading[C]. //2016 IEEE Global Communications Conference (GLOBECOM). IEEE, 2016: 1-6.
23. You C, Huang K, Chae H, et al. Energy-efficient resource allocation for mobile-edge computation offloading[J]. IEEE Transactions on Wireless Communications, 2017, 16(3): 1397-1411.
24. Lin X, Wang Y, Xie Q, et al. Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment[J]. IEEE Transactions on Services Computing, 2015, 8(2): 175-186.
25. Huang D, Wang P, Niyato D. A dynamic offloading algorithm for mobile computing[J]. IEEE Transactions on Wireless Communications, 2012, 11(6): 1991-1995.
26. Chen M H, Dong M, Liang B. Joint offloading decision and resource allocation for mobile cloud with computing access point[C]. //Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on. IEEE, 2016: 3516-3520.
27. Lin X, Wang Y, Xie Q, et al. Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment[J]. IEEE Transactions on Services Computing, 2015, 8(2): 175-186.
28. Munoz O, Pascual-Iserte A, Vidal J. Optimization of radio and computational resources for energy efficiency in latency-constrained application offloading[J]. IEEE Transactions on Vehicular Technology, 2015, 64(10): 4738-4755.
29. Nan Y, Li W, Bao W, et al. Adaptive energy-aware computation offloading for cloud of things systems[J]. IEEE Access, 2017, 5: 23947-23957.
30. Wang W, Zhou W. Computational offloading with delay and capacity constraints in mobile edge[C]. //2017 IEEE International Conference on Communications (ICC). IEEE, 2017: 1-6.
31. Liu L, Chang Z, Guo X, et al. Multi-objective optimization for computation offloading in mobile-edge computing[C]. //2017 IEEE Symposium on Computers and Communications (ISCC). IEEE, 2017: 832-837.
32. Das S , Banerjee A . An arbitration on cache replacements based on frequency — Recency product values[C]. // International Conference on Vlsi Systems. IEEE, 2016.
33. Pei C, Irani S. Cost-aware WWW proxy caching algorithms[C]. // Usenix Symposium on Internet Technologies & Systems on Usenix Symposium on Internet Technologies & Systems. 1997.
34. Cherkasova L. Improving WWW proxies performance with greedy-dual-size-frequency caching policy[M]. Hewlett-Packard Laboratories, 1998.
35. Ma T , Qu J , Shen W , et al. Weighted Greedy Dual Size Frequency based Caching Replacement Algorithm[J]. IEEE Access, 2018, PP(99):1-1.
36. Chootong S , Thaenthong J . Cache replacement mechanism with Content Popularity for Vehicular Content-Centric Networks (VCCN)[C]. // International Joint Conference on Computer Science & Software Engineering. IEEE, 2017.
37. 黄丹, 宋荣方. 基于内容价值的缓存替换策略[J]. 电信科学, 2018, 34(11): 59-66.
38. Shi W, Cao J, Zhang Q, et al. Edge computing: Vision and challenges[J]. IEEE Internet of Things Journal, 2016, 3(5): 637-646.
39. Satyanarayanan M. The emergence of edge computing[J]. Computer, 2017, 50(1): 30-39.
40. Bernstein D. Containers and cloud: From lxc to docker to kubernetes[J]. IEEE Cloud Computing, 2014, 1(3): 81-84.
41. 浙江大学SEL实验室. 容器与容器云[M]. 2. 人民邮电出版社, 2016.
42. Anon. Docker Birthday #3-Intro to Docker Slides [EB/OL]. https://www.slidesh- are.net/Docker/docker-birthday-3-intro-to-docker-slides, 2019-2-27.
43. 龚京, 吴治辉, 王伟, 等. Kubernetes权威指南：从Docker到Kubernetes实践全接触[M]. 电子工业出版社, 2017.
44. Anon. Understanding LSTM Networks [EB/OL]. [http://colah.github.io/posts/20- 15-08-Understanding-LSTMs/](http://colah.github.io/posts/20-%2015-08-Understanding-LSTMs/), 2019-1-27.
45. Graves A. Supervised sequence labelling with recurrent neural networks. 2012[J]. URL http://books. google. com/books.
46. Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.
47. Lipton Z C, Berkowitz J, Elkan C. A critical review of recurrent neural networks for sequence learning[J]. arXiv preprint arXiv:1506.00019, 2015.
48. Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures[J]. Neural Networks, 2005, 18(5-6): 602-610.
49. Cho K, Van Merriënboer B, Gulcehre C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation[J]. arXiv preprint arXiv:1406.1078, 2014.
50. Anon. Federation [EB/OL]. [https://kubernetes.feisky.xyz/he-xin-yuan-li/index-1/ feder- ation](https://kubernetes.feisky.xyz/he-xin-yuan-li/index-1/%20feder-%20ation), 2019-3-3.
51. Anon. Federation [EB/OL]. [https://kubernetes.io/docs/concepts/cluster- adminis tration /federation/#](https://kubernetes.io/docs/concepts/cluster-%20adminis%20tration%20/federation/#), 2019-3-3.
52. Anon. METALLB [EB/OL]. <https://metallb.universe.tf/>, 2019-3-3.
53. Anon. coefficient of determination [EB/OL]. <https://zh.Wikipedia.org/wiki>, 2019-3-4.
54. Xu J , Chen L , Ren S . Online Learning for Offloading and Autoscaling in Energy Harvesting Mobile Edge Computing[J]. IEEE Transactions on Cognitive Communications and Networking, 2017:1-1.
55. Anon. DockerHub [EB/OL]. <https://hub.docker.com/search/?q=&type=image>, 2019-3-4.
56. Anon. Docker Hub Top 10 [EB/OL]. [https://www.ctl.io/developers/blog/post/do- cker-hub-top-10/](https://www.ctl.io/developers/blog/post/do-%20%20%20cker-hub-top-10/), 2019-3-4.

致谢

时光荏苒，岁月匆匆。不知不觉间，三年的研究生生涯即将落下帷幕。清楚的记得，2015年9月底，那个与实验室结下不解之缘的起点。很荣幸的获得刘老师以及实验室诸位老师们的认可和青睐，从一名北邮的本科生，顺利成为了实验室大家庭的一份子、一名研0的学生，拉开了这三年的美好时光序幕。三年里虽然也曾有过怅然失落的刹那，但欢声笑语却更加的相随。更重要的是，自己在这三年里，在受周围优秀的老师、舍友以及同窗们的耳濡目染之下，向优秀的他们学习，逐渐地从当初那个稚嫩的少年开始真正的蜕变，开始学会规划自己的人生，开始学会对自己负责。告别曾经那个不成熟、怠惰的自己，迎来更加积极奋发的自我。而且，学会了在遇到问题、困难、挫折时，敢于披荆斩棘，直面问题。不再和以前一样唯唯诺诺，向困难妥协。这一路走来有太多的人，太多的事儿想要感谢。

感谢我的导师刘亮老师。首先感谢您给我来到实验室的机会，让我接触到周围这么多的优秀的小伙伴；其次，感谢您一直以来对我学业和生活上的指导，每一次和您交流学术问题，都被您知识的渊博，思维的活跃，以及求真务实，矜矜业业的态度佩服得五体投地。激励着我务必需要更加严谨求真，不能有丝毫的懈怠。

感谢对我进行指导的张海涛老师。感谢您对我在研究方向上所作的进行的指导，在与您的接触中，深感您对工作和科研的一丝不苟，做事有计划、有条理、不拖拉。这种做事风格也深深影响着我。告诫我要改掉拖沓的坏毛病。

然后，感谢实验室的马华东老师，李文生老师、段鹏瑞老师，周安福老师、傅慧源老师、以及赵东老师和郑霄龙老师，感谢对三年来对我学习和生活上的真诚照顾与关心。

还要感谢实验室里的诸多小伙伴，宋烨承、徐少青、胡彬轩、张政、刘澍、康宇超、薛玉磊、唐炳昌、杨宁、高文慧、夏丹、孙明欣、吴蕾蕾。平日里积极的互动让枯燥的工作生活，有了更多的欢声笑语。 在美好的年华与诸君相遇，使我受益匪浅，也希望大家都前程似锦。

最后，衷心感谢我的父母和家人，每当遇到挫折时，都在给我最大的支持和理解。让我有一往无前的勇气和不轻言放弃的支持。

最后，万分感谢各位评审老师和专家在百忙之中抽出宝贵的时间审阅我的文章，谢谢您们给出的宝贵意见和建议！

，

攻读学位期间取得的研究成果