文章编号：2096-1618(2019)03-0228-04

一种基于负载均衡的 Kubernetes 调度改进算法

谭 莉，陶宏才
(西南交通大学信息科学与技术学院，四川 成都 611756)

摘要：Kubernetes 是云计算领域中的容器技术编排工具和集群管理系统，默认加载的预选算法和优选算法能将 Pod 对象调度到集群中合适的节点中运行。但 Kubernetes 调度算法使用的资源模型仅包括了 CPU 和内存，未考虑节点的性能。此外，在优选过程中，对于未设置 CPU 或内存下限的容器，无论节点的性能如何，Kubernetes 都采用相同的默认值。针对上述不足，基于负载均衡对 Kubernetes 调度算法进行了改进，实验结果表明改进算法能提高 Kubernetes 集群的均衡效率。

关 键 词：云计算；Kubernetes；容器技术；负载均衡
中图分类号：TP391.9 **文献标志码**：A
doi：10.16836/j.cnki.jcuit.2019.03.003

0 引言

众所周知，实现云计算最关键、最核心的技术是虚拟化技术。起初，云计算主要使用虚拟机技术来隔离不同资源以达到应用独立的目的。然而，虚拟机技术过多的抽象给用户带来了不便[1]。2013 年，随着 Docker 容器技术的快速发展，基于容器的虚拟化技术迅速成为各大云计算厂商和云计算开发者的首选。对比虚拟机，容器技术具有镜像小、资源消耗少、应用部署灵活和启动速度快等优点[2]，但在实际生产环境中需要管理的容器数量庞大，且各容器之间的关系也可能较复杂。实际应用中，一般使用某个工具对这些数量庞大的容器实施编排部署。目前，针对容器的编排部署工具有 Docker Compose、Docker Swarm 和 Kubernetes 等。

Kubernetes 是谷歌内部使用的集群管理系统 Borg 的开源版本，是一个基于容器技术的分布式架构和集群管理系统。Kubernetes 集群中的节点分为 Master 和 Node 两种，这些节点可以是物理机，也可以是虚拟机。在 Kubernetes 中，Service(服务)是核心，而服务最终会被映射到一组容器中。为了将提供服务的容器隔离，Kubernetes 设计了 Pod 对象。Pod 中可以运行一个或多个容器。Pod 运行在 Node 节点上，它同样是 Kubernetes 的最小调度单位和最小管理单位。文中主要研究的是 Kubernetes 调度 Pod 时使用的算法，使用的容器技术为 Docker。

1 相关工作

1.1 Kubernetes 资源模型

目前，对于 Docker 容器，在 Kubernetes 集群中创建一个 Pod 时，用户可以为 Pod 中的每个容器指定运行时所需的 CPU 和内存的上下限。对应的配置字段为

- spec.containers[].resource.requests.cpu
- spec.containers[].resource.requests.memory
- spec.containers[].resource.limits.cpu
- spec.containers[].resource.limits.memory

其中，requests 和 limits 分别是资源的上限和下限。对于 CPU 而言，基本单位是核数(Cores)；对于内存，基本单位是字节数(Bytes)。requests 和 limits 都是可选的。

1.2 Kubernetes Scheduler 调度算法

Kubernetes Scheduler 是运行在 Master 节点上的进程，负责将 Pod 调度到集群中某个合适的 Node 节点上运行，并将调度结果写入 etcd 中。Kubernetes 调度 Pod 的依据是 CPU 和内存的 requests。如果 Pod 中容器未设置 requests 或者 requests 为 0，那么就按照集群设置的默认值进行计算。Kubernetes Scheduler 的设计是“插件化”的，开发者能够根据自己的需求实现自定义调度算法并注册到 Kubernetes 中。Kubernetes 中的调度算法分为预选算法和优选算法，向 Kubernetes 提供注册自定义算法时，只需提供算法名称、算法函数，如果是优选算法，则还需提供一个整数类型的权重值。

目前，Kubernetes Scheduler 的调度流程分为预选过程和优选过程。在预选过程中，Scheduler 会遍历集群中的所有 Node 节点，筛选出符合预选算法的 Node 节点作为候选节点。文中使用的预选算法有 No Disk Conflict、Pod Fits Ports、Pod Fits Resources 和 Check Node Memory Pressure[3] 等。优选过程则是使用优选算法计算每个候选节点的得分，最后将 Pod 调度到得分最高的 Node 节点上。如果存在多个得分最高者，则

收稿日期：2019-02-05
基金项目：国家自然科学基金资助项目(61505168)

从中随机选择一个 Node 节点进行调度。文中使用的优选算法有 Least Requested Priority、Balanced Resource Allocation[3] 等。每个优选算法都有一个权重,默认值都为 1。同时,每个算法的得分是[0, 10]的整数,0 表示最不合适,10 表示最合适。最后,当所有算法都应用完成后,根据设置的权重,对节点的各个算法的得分进行加权求和作为节点的最终得分。

Balanced Resource Allocation 算法是由杜军[4] 提出并实现,这个算法的目的是从候选节点中选出各项资源利用率最均衡的节点。其思想是计算 CPU 利用率和内存利用率的差值,差值越小,资源利用率越均衡。计算方式为

$$\text{score}=10-\left[\left|\frac{T_{\text{cpu}}}{C_{\text{cpu}}}-\frac{T_{\text{mem}}}{C_{\text{mem}}}\times 10\right|\right] \tag{1}$$

其中,C_{cpu}表示节点中 CPU 的总容量,T_{cpu}表示节点上运行的 Pod 和待调度 Pod 的总 CPU 占用量;同理,C_{mem}表示节点中内存的容量,T_{mem}表示节点上运行的 Pod 和待调度 Pod 的总内存占用量。

综合来看,Balanced Resource Allocation 算法还存在以下不足:第一,涉及的资源只有 CPU 和内存两种,未考虑到磁盘容量、I/O 读写速度等因素。第二,算法中只考虑了资源的利用率,未考虑节点本身的性能。另外,通过分析 Kubernetes 源码发现,对于 requests 为 0 的容器,Kubernetes Scheduler 在优选时使用 Default Milli CpuRequest 和 Default Memory Request 两个常量来进行计算,而这两个常量的设置也并未考虑节点本身的性能。因此,将针对上述 3 点不足进行改进。

2　优选算法的改进

2.1　算法改进思路

负载均衡可以优化计算机集群和网络链路等计算资源的工作负载、减少资源浪费,使资源利用率最大化[5]。负载均衡算法也能被应用到具有分布式特点的云计算环境中,Balanced Resource Allocation 算法正是基于文献[6]提出。Pradeep Kumar Tiwari 和 Sandeep Joshi 则提出了一种动态加权虚拟机实时迁移机制(DWLM)来管理云环境下的负载,其目的是使 CPU 的利用率最大化。DWLM 分为两部分,第一部分负责管理和迁移虚拟机,第二部分负责检查虚拟机的状态[7]。文献[8]提出了一种优化的 k-subset 负载均衡算法并应用与云计算。优化后的 k-subset 算法的目的是将任务分配的时间最小化。张玉芳等[9] 提出了一种基于负载权值的算法,充分地考虑了节点性能。

基于文献[9],文中将使用 CPU 利用率、内存利用率和磁盘使用率来衡量节点负载,而使用 CPU 核数、CPU 频率、内存容量以及 I/O 速率来衡量节点性能。同时,使用 $L(N_i)$ 表示节点的负载,使用 $S(N_i)$ 表示节点的性能,改进后的优选算法打分方式

$$\text{score}=10-\left[\frac{L(N_i)}{S(N_i)}\times 10\right] \tag{2}$$

$L(N_i)$ 的计算为

$$L(N_i)=L(C_i)+L(M_i)+L(D_i) \tag{3}$$

其中,$L(C_i)$ 是 CPU 利用率,$L(M_i)$ 是内存利用率,$L(D_i)$ 是磁盘使用率。

$S(N_i)$ 的计算公式为

$$S(N_i)=n\times S(C_i)+S(M_i)+S(D_i) \tag{4}$$

其中,n 是 CPU 核数,$S(C_i)$ 是 CPU 频率,$S(M_i)$ 是内存容量,$S(D_i)$ 是 I/O 速率。

2.2　改进算法的描述

改进算法分为:(1)根据候选节点名称获取节点的 CPU、内存和磁盘等数据;(2)根据待调度 Pod 中容器的 requests 是否为 0 判断是否需要使用默认 requests 值;(3)分别计算出节点的 $L(N_i)$、$S(N_i)$;(4)使用公式(2)计算节点的得分;(5)返回得分和节点名称。

改进算法中的 CPU 频率、磁盘使用率、I/O 速率并不能直接通过原有的接口获取。cAdvisor 是一个开源的容器资源使用率和性能特性的代理工具,它能监控节点的 CPU、内存、磁盘和网络等使用情况,用户可以通过其 Restful API 查询相关资源使用情况。目前,在 Kubernetes 项目中,cAdvisor 已经被集成到了 kubelet 组件的源码中。因此,在 Kubernetes 项目内部可以使用 Kubernetes Proxy API 访问相关数据。文中使用两个 Kubernetes Proxy API:

- /api/v1/proxy/nodes/%s/stats/
- /api/v1/proxy/nodes/%s/spec/

其中,%s 使用节点名称(Node 结构体中的 name 字段)替换。第一个 API 用于获取磁盘和 I/O 等信息,而第二个 API 则用于获取 CPU 信息。

2.3　改进算法的实现

2.3.1　设置默认 requests

当待调度 Pod 中某个容器的 requests 为 0 时,需要使用 Kubernetes 设置的默认 requests。阅读 Kubernetes 源码发现,DefaultMilliCpuRequest 是默认的 CPU 下限值,其值为 100,即0.1个 CPU 核数,DefaultMemoryRequest 是默认的内存下限值,其值为 200MB(200×1024×1024)。无论节点性能如何,这两个值都固定不变。文中改进算法根据 CPU 核数来设置 CPU 默认 requests,根据内存总容量来设置内存默认 requests,二者计算公式为

$$out_{CPU} = 100 \times n \quad (5)$$

$$out_{Memory} = 200 \times 1024 \times 1024 \times mem \quad (6)$$

out_{CPU}和 out_{Memory}为新的默认值,n 表示 CPU 核数,mem 为内存容量。需要注意的是,内存容量从 cAdvisor 获取时单位是字节,在这里需要将其转换为 GB,并向上取整。

2.3.2 计算磁盘容量、已使用量和 I/O 速率

节点的磁盘数据和 I/O 数据均可通过内置的"/api/v1/proxy/nodes/%s/stats/" API 获取,获取出的磁盘数据和 I/O 数据均被放在"stats"字段中。"stats"字段是一个数组,数组的每一项都带有时间戳,表示不同时间的数据。

节点的磁盘数据存放在"stats[ ]. filesystem"中。"stats[ ]. filesystem"也是一个数组,表示所有的磁盘分区。数组每一项都记录了一个分区的信息,包括设备名称、设备类型、分区容量和分区已使用量等。节点的 I/O 数据存放在"stats[ ]. diskio"中,"stats[ ]. diskio"是一个对象。其中,"io_service_bytes"字段记录着 I/O 操作的字节数,"io_service_time"字段记录着 I/O 操作时间。这两个字段均为数组,数组每一项表示了一个设备。计算磁盘容量、已使用量和 I/O 速率的伪代码描述如下:

Input:cInfo,cadvisorapi. ContainerInfo 结构体;
Output:
(1) capacity,磁盘容量;
(2) usage,已使用量;
(3) speed,I/O 速率

```
function computeFSInfo(cInfo)
capacity←0
usage←0
speed←0
statslen←len(cInfo. Stats)
for i=0→statslen - 1 do
total←0,time←0
fslen←len(cInfo. Stats[i]. Filesystem)
for j=0→fslen - 1 do
capacity←capacity+cInfo. Stats[i]. Filesystem[j]. Limit
usage←usage+cInfo. Stats[i]. Filesystem[j]. Usage
end for
bytes←cInfo. Stats[i]. DiskIo. IoServiceBytes
times←cInfo. Stats[i]. DiskIo. IoServiceTime
Iolen←len(bytes)
for j=0→Iolen - 1 do
total←total+bytes[j]. Stats["Total"]
time←time+times[j]. Stats["Total"]
end for
speed←speed+total/time
end for
speed←speed/statslen
capacity←capacity/statslen
usage←usage/statslen
return capacity,usage,speed
end function
```

3 实验分析

3.1 实验设计

将改进算法加入到 Kubernetes v1.5.2版本中,并在 Docker 容器中编译生成可执行的二进制文件。使用4 台腾讯云服务器搭建实验环境,各个云服务器的参数如表 1 所示。

表 1 实验环境

节点名称	操作系统	CPU	内存
master	CentOS7.2(64)	1 核	2G
node1	CentOS7.2(64)	2 核	4G
node2	CentOS7.2(64)	1 核	1G
node3	CentOS7.2(64)	1 核	1G

表中第一个节点为主节点,即 master 节点,其他节点为工作节点,即 Node 节点。通过部署两次 Kubernetes 集群来实施实验,实验过程如下:

(1)使用改进前的优选算法搭建一个 Kubernetes 集群,记作集群 1。集群中使用表 1 的 master 作为主节点,使用 node1、node2 和 node3 作为工作节点;(2)在集群中部署 30 个 Pod;(3)部署完成后,分别计算 node1、node2 和 node3 的均衡效率;(4)销毁集群 1,再使用改进算法重新搭建一个 Kubernetes 集群,记作集群 2。集群 2 中仍使用表 1 的 master 作为主节点,node1、node2 和 node3 为工作节点;(5)重复第(2)步;(6)再次 node1、node2 和 node3 的均衡效率。

根据文献[9],实验采用均衡效率作为评价指标。均衡效率计算公式为

$$H(N_i) = \frac{L(N_i)/L(N_{avg})}{S(N_i)/S(N_{avg})}, i = 1,2,3 \quad (7)$$

其中,$L(N_{avg})$是 3 个工作节点的负载均值,$S(N_{avg})$是3 个工作节点的性能均值。均衡效率比值越接近于 1 说明算法越好。

3.2 实验结果

根据公式(7),两个集群各个节点的均衡效率如

表 2 和图 1 所示。

表 2　实验结果　　%

集群名称	node1	node2	node3
集群 1	82.02	102.59	121.94
集群 2	94.15	104.89	106.63

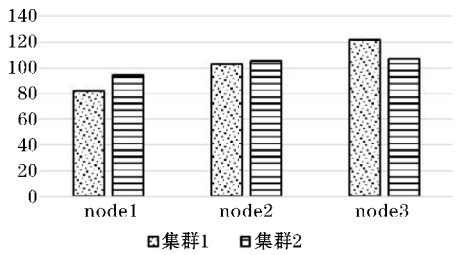

图 1　实验结果柱状图

从实验结果看,在集群 2 中,node1 和 node3 的均衡效率更加接近于 1,node2 比集群 1 中稍差一点,但总体来说,使用改进算法后集群总体的均衡效率较高。

4　结束语

针对 Kubernetes 优选算法中未考虑节点本身性能、使用的资源模型过小和设置默认 requests 不合理这 3 点不足,对算法进行改进。首先,在设置 CPU 默认 requests 时考虑了 CPU 核数,在设置内存默认 requests 考虑了内存总容量。其次,改进的优选算法考虑了节点本身的性能。最后,通过实验验证,改进后的优选算法使集群的资源更加均衡。

参考文献:

[1] Wes Felter, Alexandre Ferreira, Ram Rajamony, et al. An Updated Performance Comparison of Virtual Machine and Linux Container[C]. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015: 171-172.

[2] 武志学. 云计算虚拟化技术的发展与趋势[J]. 计算机应用,2017,37(4):915-923.

[3] 闫健勇,龚正,吴治辉,等. Kubernetes 权威指南[M]. 纪念版. 北京:电子工业出版社,2017.

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

[5] Sidra Aslam, Munam Ali Shah. Load balancing algorithms in cloud computing: A survey of modern techniques[C]. National Software Engineering Conference(NSEC). 2015:30-35.

[6] Wei Huang, Xin Li, Zhuzhong Qian. An Energy Efficient Virtual Machine Placement Algorithm with Balanced Resource Utilization[C]. Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, 2013:313-319.

[7] Pradeep Kumar Tiwari, Sandeep Joshi. Dynamic Weighted Virtual Machine Live Migration Mechanism to Manages Load Balancing in Cloud Computing[C]. IEEE International Conference on Computational Intelligence and Computing Research (ICCIC). 2016:1-5.

[8] Linlin Tang, Pingfei Ren, Jengshyang Pan. An Improved K-Subset Algorithm for Load Balance Problems in Cloud Computing[C]. IEEE 3rd International Conference on Cloud Computing and Intelligence Systems. 2014:175-179.

[9] 张玉芳,魏钦磊,赵膺. 基于负载权值的负载均衡算法[J]. 计算机应用研究,2012,29(12):4711-4713.

An Improved Kubernetes Priority Algorithm based on Load Balancing

TAN Li,　TAO Hongcai

(College of Information Science & Technology, Southwest Jiaotong University, Chengdu 611756, China)

Abstract: Kubernetes is an orchestration tool and a cluster management system based on container in cloud computing area. The default predicate algorithms and priority algorithms in Kubernetes scheduler can schedule pod to a suitable node to run. However, the resource model used by the default algorithms just contains CPU and memory, and doesn't take the performance of node into account either. In addition, during the priority process, for the containers without the requests of CPU or memory, Kubernetes applies the same default values. Aiming at the shortcomings above, this paper proposes an improved priority algorithm based on load balancing. The experiment results show that the improved priority algorithm can enhance equilibrium efficiency.

Keywords: cloud computing; Kubernetes; container; load balancing