计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(10):213−218 [doi: 10.15888/j.cnki.csa.007106]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 Kubernetes 应用的弹性伸缩策略①

陈　雁, 黄嘉鑫

(西南石油大学 计算机科学学院, 成都 610500)
通讯作者: 陈　雁, E-mail: carly.chenyan@gmail.com

摘　要: 弹性伸缩是云计算的关键特征, 它可以根据应用程序工作负载及时扩展计算资源以实现在高并发请求下应用的负载均衡. 基于容器的微服务更应具有弹性伸缩功能从而在不同的工作负载条件下稳定运行. 目前广泛使用的容器编排工具 Kubernetes 的弹性伸缩算法灵活性差, 应对突发流量 Pod 会频繁进行扩展, 并且扩展程度不能满足当前负载要求, 会造成系统不稳定. 针对这一问题, 本文提出了一种自动缩放机制, 将响应式扩展与弹性伸缩容忍度相结合, 确保了系统的可靠性, 大大提高了系统的灵活性, 并具有很强的应用负载能力. 实验测试表明, 当系统面临大流量、高并发请求时, 通过本文的方法实施弹性伸缩以后, 失败请求率下降 97.83%, 保证了系统稳定性, 能够很好的实现应用的负载均衡.

关键词: 云计算; docker; Kubernetes; 自动伸缩算法; 弹性负载均衡

引用格式: 陈雁,黄嘉鑫.基于 Kubernetes 应用的弹性伸缩策略.计算机系统应用,2019,28(10):213–218. http://www.c-s-a.org.cn/1003-3254/7106.html

Elastic Scaling Strategy Based on Kubernetes Application

CHEN Yan, HUANG Jia-Xin

(School of Computer Science, Southwest Petroleum University, Chendu 610500, China)

Abstract: Autoscaling is a key feature of cloud computing. It can expand computing resources in time according to application workload and achieve load balancing under high concurrent requests. Container-based micro-services should also have the function of autoscaling so as to have stably performance under different workloads. The elastic scaling algorithm of Kubernetes, a widely used container layout tool, has unsatifactory flexibility. Pod will expand frequently to deal with sudden traffic, and the scaling degree can not meet the current load requirements, which will make a system instability. To solve this problem, an automatic scaling mechanism is proposed, which combines the response expansion with the elastic scaling tolerance, and ensures the reliability of the system. Our method greatly improves the flexibility of the system, and is also competent when facing high application load. Experiments results show that when the system meet with heavy traffic and high concurrent requests, the failure request rate can decrease by 97.83% after carrying out the proposed method. So our method can ensure the stability of the system and realizes the load balancing of the application well.

Key words: cloud computing; docker; Kubernetes; autoscaling algorithm; elastic load balancing

随着计算机技术的更新迭代, 传统的应用正在变得越来越复杂, 需要支持更多的用户、更强的计算能力, 而且还需要更加稳定和安全, 为了支撑这些不断增长的需求, 一种新型的计算模式“云计算”应运而生. 云

① 基金项目: 国家自然科学基金青年基金项目 (61503312)
Foundation item: Young Scientists Fund of National Natural Science Foundation of China (61503312)
收稿时间: 2019-03-14; 修改时间: 2019-04-04, 2019-04-18; 采用时间: 2019-04-23; csa 在线出版时间: 2019-10-15

© 中国科学院软件研究所 http://www.c-s-a.org.cn

计算是一种新的基于互联网的计算方式和资源应用平台, 从计算方式上来说, "云计算"是通过网络将庞大复杂的数据交由多台服务器组成的集群系统进行存储、计算、分析后, 将处理结果回传给用户的一种计算模式[1].

虚拟化是云计算的基石, 传统虚拟化技术是在硬件资源级别的虚拟化, 需要有虚拟机管理程序和虚拟机操作系统. 随着 Docker[2]的出现, 容器技术正在成为打包和部署应用程序的新趋势, Docker 是直接建立在操作系统上的虚拟化, 它直接复用本地操作系统, 更加轻量级. 使用 Docker 比使用虚拟机 (VM) 部署应用启动更快, 系统的开销更小, 能够更有效地支持应用程序的快速迭代[3]. 目前, Docker 容器已在云基础架构中得到广泛部署, 例如 Amazon Ec2 Container Service, Google Container Engine, Rackspace, Docker Data Center[4].

Docker 是一个相对底层的容器引擎, 在大规模的服务器集群中, 需要一个集中的控制器来进行任务的编排, 调度和控制. 而 Kubernetes[5]提供了大规模运行容器的编排系统和管理平台, 它实现了包括应用部署、高可用管理和弹性伸缩在内的一系列基础功能, 并封装成为一整套完整、简单易用的 RESTful API 对外提供服务.

Kubernetes 的弹性扩展功能是由 Horizontal Pod Autoscaler (HPA)[6]控制器来实现的, HPA 会根据自定义指标控制副本集中的容器数量, 这种自动扩展伸缩机制使用固定阈值的规则, 并且 Kubernetes 的扩展和收缩 Pod 数量是根据同一套算法, 同一个检测指标来判定是否需要扩容或者收缩.

由于 Kubernetes 的扩容和收缩决策判断基于同一个指标, 这就会产生频繁扩容缩容的问题; 在应对大规模突发流量时, Kubernetes 的扩容幅度不足以应对大规模流量, 就会造成访问中断; 在访问流量骤降的时候, Kubernetes 内置算法则会迅速收缩 Pod 的数量, 收缩速度太快, 当出现第二次突发流量, 收缩后的 Pod 还来不及扩展, 就会导致应用负载过大, 造成应用崩溃.

针对 Kubernetes 弹性扩展的调度策略, Casalicchio 等[7]对 CPU 密集型的工作负载, 提出 KHPA-A 算法, 该算法采用绝对度量指标来做扩展决策, 使用绝对指标来触发容器弹性伸缩和确定 Pod 的数量, 将度量指标保持在设定阈值以下. Al-Dhuraibi 等[8]在 Docker 层面提出 ELASTICDOCKER 算法, 该算法是垂直弹性扩展容器的 CPU 和内存大小, 当应用程序工作负载上升/下降时, ELASTICDOCKER将分配给每个容器的 CPU 和内存向上/下弹性伸缩. 垂直弹性仅受限于主机容量, 当所有主机资源都已分配给容器时, 容器将会执行从源主机到目标主机的动态迁移.

以上研究是针对于特殊场景的 CPU 密集型计算和纵向扩展增加容器的 CPU 和内存, 而如何在通用常规情况下, 使用方便简单的方法保证系统平稳、稳定运行则是更需要解决的问题. 因此, 本文提出步长容忍度算法水平伸缩的控制器, 它能在当应用服务器中业务负载上升的时候, 迅速创建多个新的 Pod 来保证业务系统稳定运行, 当 Pod 中业务负载下降的时候, 在保证系统稳定运行的前提下逐步销毁 Pod 来提高资源利用率. 对于应用, 需要保障能够稳定平滑的运行, 而不只是尽可能的节约成本. 步长容忍度算法在扩展的时候, 保证应用服务器能够承担不断增长的连接数, 做到快速扩展, 甚至过度扩展. 在收缩的时候采取较为宽松的收缩策略, 达到逐渐收缩的目的.

1 Kubernetes 内置算法

HPA 是 Kubernetes 里面 Pod 弹性伸缩的实现, 它在 Kubernetes 中被设计为一个 controller, 能根据设置的监控阀值进行 Pod 的弹性扩缩容[9]. HPA Controller 默认 30 s 轮询一次, 查询指定的 resource 中 Deployment/RS 的资源使用率[10], 并且与创建时设定的值和指标做对比, 从而实现自动伸缩的功能. HPA 控制器工作原理如图 1 所示.

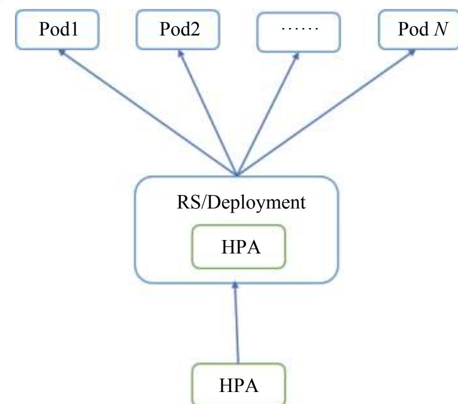

图 1 HPA 控制器原理图

HPA 使用式 (1) 中的方法计算

$$dP = ceil[cR*(cM/dM)] \tag{1}$$

© 中国科学院软件研究所 http://www.c-s-a.org.cn

其中, dP (desired Replicas) 为期待的副本数量, 即算法计算后得到的副本数量; cR (current Replicas) 为当前的副本数量, 探测器检测到的副本的数量; cM (current Metric value) 为当前设置的检测指标的值, 比如 CPU 利用率, 内存使用情况; dM (desired Metric value) 为期望的 Pod 的检测指标的值; *ceil* 为表示取大于或等于某数的最近一个整数.

HPA 中 Pod 水平弹性伸缩的实现是通过定期轮循查询 Pod 的状态 (默认轮询时间为 30 s) 通过式 (1) 计算. 在每个周期时间, 控制器管理器根据每个 HPA 定义中指定的度量标准查询资源利用率. 控制管理器从资源指标的 API 中获取度量, 获取到 Pod 的监控数据, 然后通过现有 Pod 使用率的平均值跟目标使用率进行比较, 计算出需要伸缩的具体值并进行操作. 在每次扩容和收缩都有一个窗口时间, 在执行伸缩操作后, 在这个窗口时间内, 不会再进行伸缩操作, 默认扩容为 3 min, 收缩为 5 min.

例如当前的副本数量为 1 个, 设置当前度量标准值 cM 是内存, 检测到当前 Pod 的内存为 150 MB, 而初始设定的期望为 100 MB, 则副本的数量将变为 2 个, 因为 1×(150/100) = 1.5 向上取整为 2, 所以 Pod 数量会弹性扩展到 2 个.

参考 Kubernates 源代码发现, 原生弹性伸缩的实现非常简单: (1) 计算所有 Pod 的度量指标使用率; (2) 根据弹性伸缩算法计算 Pod 的需求量; (3) 按照计算出的 Pod 数量进行扩容和伸缩. 这一自动缩放算法灵活性差, 扩容和收缩都是用同一个算法, 对突发流量会造成频繁扩容问题, 突发流量过后会造成频繁收缩的问题; 容器启动是需要一定的时间的, 在应对大规模突发流量时, 扩容还来不及扩展 Pod, 就可能会造成当前应用承受不了负载而崩溃, 应当为新扩容的容器实例预留启动时间, 给正在运行的 Pod 充足的资源应对负载; 逐渐收缩, 在保证访问量承载的前提下, 逐步递减收缩 Pod, 防止收缩太快低于系统的承载能力, 造成系统的不稳定, 针对这些问题本文提出了步长容忍度算法.

2 步长容忍度算法

设计步长容忍度算法的时候, 考虑到自动扩展的决策需要一段时间才会生效, 若当前 Pod 的 CPU 负荷过大, 在创建一个新 Pod 的过程中, 应用系统的 CPU 使用量可能会同样有一个攀升的过程. 所以在循环探测中, 在每一次作出决策后的一段时间内, 将不再进行扩展/收缩决策. 对于扩容而言, 这个时间段为 3 分钟, 缩容为 5 分钟. 因为需要尽可能满足 Pod 业务的正常使用, 所以扩容的优先级要大于缩容. 步长容忍度算法会通过调整副本数量使得检测指标使用率尽可能向期望值靠近, 而且不是完全相等. 步长容忍度算法在 HPA Controller 中引入一个 tolerance (容忍度) 的概念, 它允许在一定范围内使用量的不稳定, 设置容忍度为 15%, 这也是出于维护系统稳定性的考虑. 例如, 设定 HPA 调度策略为 CPU 使用率高于 60% 触发扩容, 那么只有当使用率大于 75% 或者小于 45% 才会触发伸缩活动, HPA 会尽力把 Pod 的使用率控制在这个范围之间. 自动伸缩的监测指标包括 CPU 平均使用量、内存平均使用量、用户自定义一些监控指标.

步长容忍度算法包含两个部分: 快速扩展算法和逐步收缩算法.

2.1 快速扩展算法

步长容忍度算法的快速扩展将 Pods 的目标期望利用率 T_{metric}(作为 *MetricValue* 指标的百分比)、前一个轮询控制周期 (T s) 中运行的 Pods 的 *ActPod* 集合、实例化的初始 Pod 扩容数量、设定的扩展步长 *UpSetup* 等作为输入, 输出是则要部署的 Pods 的目标数量 P.

系统启动之后, 每隔 T s(如 30 s) 循环检查一次所有 HPA, 检测结果若有一个 Pod 指标超过初始设置的期望值加上容忍度的值, 就以步长 (如 2) 执行扩展. 每次扩展完成后, 记录扩展时间间隔, 保证扩展操作间隔不小于 3 min 以确保系统的稳定性. Pod 的数量按照式 (1) 进行计算, 而利用率/期望的比例大于 1.15(容忍度为 15%), 才能进行扩容, 防止抖动.

系统每 T s(实验设置为 30 s), 算法收集 Pod 的 *MetricValue* 利用率, 用 cAdvisor(容器监控工具) 测量并将其存储在向量 U 中. 最后, 计算出 *DisredPods* 的目标数量 P, P 由式 (2) 计算得出.

$$P = \left\lceil \frac{\sum_{i \in Actpod} U_i}{T_{metric}} + S \right\rceil \tag{2}$$

算法 1 为步长容忍度算法中的快速扩展算法.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

算法 1. 步长容忍度算法的快速扩展算法

Inputs:
TMetric: 期望的检测指标的平均使用率;
AvgMetric: 检测指标的平均使用率;
ActPod: 正在运行的 Pod 的数量;
DisredPods: 期望 Pod 的数量;
Tolerance: 期望的容忍度;
UpSetup: 扩容的步长;
T: 定期轮询时间.

```
run Application(s)
init ActPod = 1
while true do
  if [AvgMetric > (T_metric + Tolerance)] {
    if (ActPod = = 1){
      DesiredPod = ActPod + UpSetup
        ActPod = DesiredPod
    }
  }
  else{
    DesiredPod=Ceil[ActPod*(AvgMetric/Tmetric)+UpSetup]
    ActPod = DesiredPod
  }
  wait(T)
  end while
```

假设 T_{metric} = 60%. 正在运行 3 个应用程序副本, 每个 Pod 的 CPU 利用率分别为 73%, 75% 和 82%. 在下一个控制周期, 步长容忍度算法确定应该部署新的 Pod $P = 6$. 这样快速扩展以应对大规模的数据流量, 防止应用崩溃. 负载将在 Pod 之间分配, 预计的每 Pod 调整利用率由式 (3) 计算出:

$$\frac{\sum_{i=1}^{p} Ui}{p} = 38.33 \tag{3}$$

在下一次扩容的时间间隔 3 min 内, 以便有足够的资源来负载访问流量, 达到快速扩容的目的.

2.2 逐步收缩算法

步长容忍度算法节点收缩采取的策略是当监测指标值低于所设定的缩容条件, 以默认步长 2 减少节点数量. 当监测节点数量为 2, 停止减少容器数量.

约束规则为缩容条件的可选范围是 (0%~45%). 缩容的时候当节点数<= 集群最小节点数 (设置为 2) 的时候, 不会进行缩容操作.

算法 2 为步长容忍度算法中的逐步收缩的算法.

假设 T_{metric} = 60%. 正在运行 6 个应用程序副本, 每个 Pod 的 CPU 利用率分别为 50%, 45%, 47%, 52%, 43% 和 40%. 在下一个控制周期, 步长容忍度算法确定收缩节点的数量 P=4. 这样以步长为 2 逐步收缩. 负载将在 Pod 之间分配, 预计的每 Pod 调整利用率由式 (4) 计算出变为:

$$\frac{\sum_{i=1}^{p} Ui}{p} = 59.25 \tag{4}$$

算法 2. 步长容忍度算法的逐步收缩算法

Inputs:
TMetric: 期望的检测指标的平均使用率
AvgMetric: 检测指标的平均使用率
ActPod: 正在运行的 Pod 的数量
DisredPods: 期望 Pod 的数量
Tolerance: 期望的容忍度
DownSetup: 收缩的步长
T: 定期轮询时间

```
run Application(s)
init ActPod = 1
while true do
  if [AvgMetric < (Tmetric-Tolerance)] {
    if (ActPod = = 2){
      ActPod = 2
    }
  }
  else{
    DesiredPod = ActPod -DownSetup]
    ActPod = DesiredPod
  }
  wait(T)
end while
```

在下一次收缩的时间间隔 5 min 内, 保持系统的稳定性, 达到逐步收缩的目的.

3 实验结果与分析

3.1 实验环境

在实验中, 为了验证步长容忍度算法, 实验环境采用 2 套相同的 Kubernetes 实验环境做对比测试, 一套 Kubernetes 实验环境使用原生的弹性伸缩算法, 对比实验采用步长容忍度算法. Kubernetes 实验环境都是 1 个 master 节点和 2 个 node 节点, Kubernetes 版本 1.11.3, docker 版本为 17.03.3-ce. 在实验中使用的环境配置如表 1 所示.

表 1 实验环境配置

特征	OS	CPU(s)	CPU (MHz)	内存 (MB)	网卡 (Mbps)
Master 节点	CentOS 7.2	4	2600×4	8192	1000
Node 节点	CentOS 7.2	4	2600×4	8192	1000

© 中国科学院软件研究所 http://www.c-s-a.org.cn

设定了一个在进行压力测试的情况下检验 Pod 个数的变换以及每个 Pod 的 CPU 负载的变换情况.

3.2 实验步骤

实验包含两个部分: (1) 使用 ApacheBench 进行压力测试, 在压力测试的时候检验 pod 的变化情况, 在压力测试完成后, 观察 Pod 随即的数量变化状况, 验证在应对突发流量时扩容/收缩情况. (2) 使用不同的并发请求数, 请求 5 000 000 个连接, 在高压高并发的情况下检测服务的吞吐率、用户平均请求等待时间、服务器平均处理时间, 验证步长容忍度算法是否提高系统的稳定性. 每个实验重复进行 5 次迭代, 取实验结果的平均值.

实验设定: 创建部署 2 个一主两从的 Kubernetes 集群. 建一个包含 nginx-stress 容器的 deployment, 对这个 deployment 持续进行并发请求, 模拟用户在客户端频繁地访问 web 应用, 应用会将访问流量转发到后端一个 Pod 上, 继而增加这个 Pod 的压力, Kubernetes 系统检测到 Pod 内存和 CPU 在不断升高, 到达扩容的临界值就会扩展 Pod, 来应对这些流量.

实验表明, 启动一个 Pod 容器实例所需的时间大约是 6 s. 在实验中, Kubernetes 资源探测的时间间隔定义为 30 s, 虽然也可以将监视间隔指定为更短的时间, 但是将其设置为 30 s 可以减少通信流量负载和测量的监视开销. 在实验中, 本文把阈值被设置为 65%, 这样步长容忍度算法将有足够的时间对工作负载中的运行时变化作出反应, 而不是阈值非常接近 100% 才进行反应, 从而防止压力过大应用崩溃的问题. 扩容间隔时间设置为 3 min, 可以防止运行容器实例数量的过于频繁的变化. 收缩时间间隔为 5 min, 避免收缩过快引发系统不稳定性.

对应用进行持续不断的请求, 进行压力测试, 图 2 显示了在工作负载急剧上升的场景下内置算法和步长容忍度算法的 Pod 的数量变换情况.

由图 2 可见, 在应用收到大规模流量请求时, 内置算法扩容的比例比步长容忍度算法小的多, 从应用的负载情况来看, 步长容忍度算法能够很好的支持大规模访问. 步长容忍度算法几乎没有失败请求数, 而内置算法出现了不少的失败请求数 (见表 2), 由于工作量增加导致系统过载, 内置算法的应用提供了一段时间的慢响应时间. 步长容忍度算法这种快速扩展能够很好支撑大规模流量访问.

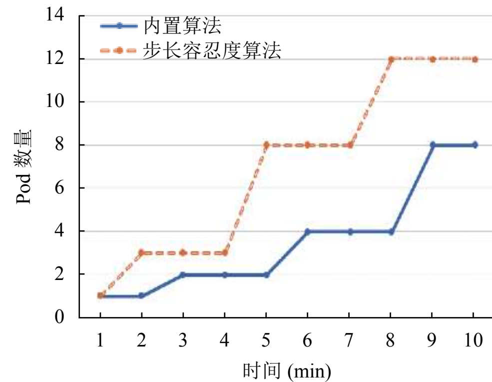

图 2 Pod 增长变化图

表 2 内置算法并发请求测试

并发请求数	测试持续时间 (s)	失败请求数 (个)	吞吐率 (/s)	用户请求等待时间 (ms)	服务器平均处理时间 (ms)
125	852.34	60	5859.36	21.32	0.174
250	535.52	108	9339.5	26.58	0.106
500	413.68	162	12 091.82	41.16	0.082
1000	378.66	342	13 189.4	75.90	0.076

当工作负载密度下降到 1 个请求, Pod 的数量会根据执行时到达的请求的数量而减少. 图 3 显示了在工作负载下降的场景的 Pod 的数量变化情况.

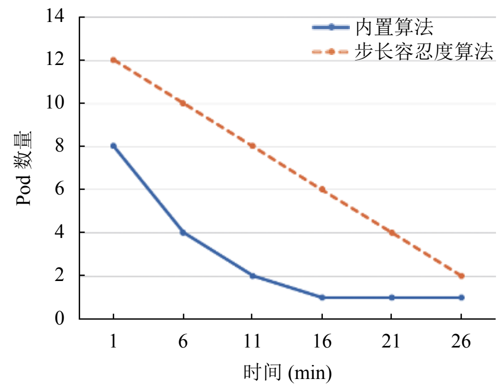

图 3 Pod 收缩变化图

步长容忍度算法和内置算法的自动弹性伸缩方法都不会立即停止在集群中运行的容器实例, Pod 数量是呈现逐渐减少的状态. 与内置算法的自动缩放方法相比, 步长容忍度算法分配的 Pod 数量明显多于内置算法, 是递减收缩. 在工作负载突然增加时, 步长容忍度算法在开始及启动时有足够的 Pod 实例, 比内置算法自动扩展方法更快. 因此, 对于急剧变化的工作负载模式, 步长容忍度算法有更好的负载能力和维持系统的稳定性能.

© 中国科学院软件研究所 http://www.c-s-a.org.cn

本文也对在弹性扩展中 Pod 的性能做了测试, 在进行 5 000 000 个连接数情况下对应用的失败请求数、吞吐率、用户平均请求等待时间和服务器平均处理时间进行了测试, 表 2 为内置算法的并发请求性能测试, 表 3 为步长容忍度算法的并发请求性能测试.

表 3　步长容忍度算法并发请求测试

并发请求数	测试持续时间 (s)	失败请求数 (个)	吞吐率 (/s)	用户请求等待时间 (ms)	服务器平均处理时间 (ms)
125	858.4	0	5816.2	21.48	0.172
250	502.14	0	9954	24.12	0.101
500	403.28	6	13 212.29	39.32	0.075
1000	364.18	17	13 404.23	71.94	0.075

对比内置算法和步长容忍度算法并发请求测试的结果, 在并发请求数比较少的情况, 内置算法和步长容忍度算法表现都差不多, 在失败请求数这个指标, 内置算法由于持续压力测试当前启动的 Pod 不能负载流量, 启动新的 Pod 需要一定的启动时间, 造成应用的不稳定, 产生较多的失败请求. 在并发请求数比较大的时候, 步长容忍度算法的优势就体现出来了, 应用能在更短的时间内处理请求, 响应客户端的速度也更快, 服务器的吞吐率也更大.

实验表明, 在并发请求数较大的情况下, 步长容忍度算法的 Kubernetes 集群处理能力比内置算法的集群应对突发流量的能力更强, 内置算法的集群对突发流量会出现承载过大, 出现较多失败请求, 对比使用不同并发请求数的请求测试, 步长容忍度算法的失败请求数比内置算法的失败请求数下降了 97.83%, 整个弹性伸缩系统更加稳定健壮. 步长容忍度算法集群的吞吐率比内置算法集群的吞吐率更大, 处理请求的速度越快, 使用步长容忍度算法的服务器的响应速度 (依据用户请求等待时间) 比内置算法算法的响应速度更快, 提高了 4.54%. 步长容忍度算法在其他方面的结果比内置算法结果好. 此实验通过用压力测试模拟用户请求触发阈值进而进行扩容验证了本文提出的步长容忍度算法的有效性, 是在应对大规模流量横向扩展的有效手段.

4　结论与展望

Kubernetes 内置的 Horizontal Pod 弹性缩放算法扩容和收缩采用相同的伸缩机制, 会造成频繁的扩容和收缩问题. 本文提出了一种新的自动缩放算法, 该算法能够实现快速扩容和逐步收缩以达到系统稳定运行的状态, 对于服务的稳定性是一个很好的提升. 本文提出的方案在并发请求数较大的情况下, 步长容忍度算法比内置弹性伸缩算法应对突发流量的能力更强.由于目前弹性伸缩策略是根据实时的使用率来进行扩容, 但其实对于很多应用来说, 很多业务高峰期是有规律的, 因此如果未来能做到提前预测出高峰期, 做适应的扩容, 那将会极大地提高资源使用率.

参考文献

1 危烽. 浅谈云计算在互联网中的应用. 电脑知识与技术, 2009, 5(3): 583–584, 670. [doi: 10.3969/j.issn.1009-3044.2009.03.029]

2 Boettiger C. An introduction to Docker for reproducible research. ACM Sigops Operating Systems Review, 2015, 49(1): 71–79. [doi: 10.1145/2723872]

3 肖俊. 基于 Docker 的跨主机容器集群自动伸缩设计与实现[硕士学位论文]. 西安: 西北大学, 2015.

4 Bernstein D. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 2014, 1(3): 81–84. [doi: 10.1109/MCC.2014.51]

5 Burns B, Grant B, Oppenheimer D, *et al*. Borg, omega, and Kubernetes. Communications of the ACM, 2016, 59(5): 50–57. [doi: 10.1145/2930840]

6 Vohra D. Using autoscaling. Kubernetes Management Design Patterns. Berkeley, CA: Apress, 2017. 299–308.

7 Casalicchio E, Perciballi V. Auto-scaling of containers: The impact of relative and absolute metrics. 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems. Tucson, AZ, USA. 2017. 207–214.

8 Al-Dhuraibi Y, Paraiso F, Djarallah N, *et al*. Autonomic vertical elasticity of docker containers with elasticdocker. 2017 IEEE 10th International Conference on Cloud Computing (CLOUD). Honolulu, CA, USA. 2017. 472–479.

9 Huang W, Zhang W, Zhang DY, *et al*. Elastic spatial query processing in openstack cloud computing environment for time-constraint data analysis. International Journal of Geo-Information, 2017, 6(3): 84. [doi: 10.3390/ijgi6030084]

10 Khazaei H, Ravichandiran R, Park B, *et al*. Elascale: Autoscaling and monitoring as a service. Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering. Markham, Ontario, Canada. 2017. 234–240.

© 中国科学院软件研究所 http://www.c-s-a.org.cn