计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(4):129−140 [doi: 10.15888/j.cnki.csa.009023]
©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

# 集群环境中微服务容器资源特征分析及优化①

姚庆安, 刘力鸣, 张　鑫, 金镇君, 冯云丛, 赵　健

(长春工业大学 计算机科学与工程学院, 长春 130102)
通信作者: 金镇君, E-mail: 34712384@qq.com

**摘　要**: 在集群环境中部署微服务已经成为微服务部署的重要方式. 由于不同种类服务对于 CPU、内存、磁盘等资源的需求不同, 导致集群中的节点产生资源碎片、出现资源消耗倾斜. 如何提高集群资源利用率、降低集群能耗, 成为继保障服务级别协议 (service level agreement, SLA) 之后的重大挑战. 本文以阿里巴巴集团 2021 年发布的近两万个微服务的详细跟踪为数据样本, 从容器资源使用情况、节点部署特征和资源消耗偏好等多个维度出发, 分析其集群资源消耗特征, 发现集群中出现了资源消耗倾斜的情况. 通过进一步分析节点中容器部署情况发现容器资源分配不合理加剧了这一现象. 基于此我们提出了一种使用深度双 Q 网络的模型, 依据上游服务资源需求的实时变化, 对容器资源分配进行优化. 对比实验结果表明该方法可以在保证服务 SLA 的情况下有效提高容器资源利用率, 改善节点资源消耗倾斜的情况.

**关键词**: 微服务; 负载特性; 容器调度; 云计算; 深度强化学习

引用格式: 姚庆安,刘力鸣,张鑫,金镇君,冯云丛,赵健.集群环境中微服务容器资源特征分析及优化.计算机系统应用,2023,32(4):129–140. http://www.c-s-a.org.cn/1003-3254/9023.html

## Analysis and Optimization of Microservice Container Resources in Cluster Environment

YAO Qing-An, LIU Li-Ming, ZHANG Xin, JIN Zhen-Jun, FENG Yun-Cong, ZHAO Jian

(School of Computer Science and Technology, Changchun University of Technology, Changchun 130102, China)

**Abstract**: Microservice deployment in a cluster environment has become an essential way. As different kinds of services have different demands on resources such as CPU, memory, and disk, it makes the nodes in the cluster produce resource fragments and become biased in resource consumption. How to enhance cluster resource utilization and reduce cluster energy consumption has become a major challenge after the service level agreement (SLA) is guaranteed. This study takes the detailed tracking of nearly 20 000 microservices released by Alibaba Group in 2021 as data samples and analyzes their cluster resource consumption characteristics from multiple dimensions, such as container resource utilization, node deployment characteristics, and resource consumption preferences. As a result, the study finds that biased resource consumption occurs in the cluster. Further analysis of container deployment in the nodes reveals that this phenomenon is exacerbated by the inappropriate allocation of container resources. In view of this, this study proposes a model based on a deep dual-Q network to optimize container resource allocation according to the real-time changes of upstream service resource demands. The experimental results are compared, and it is shown that the method can effectively increase container resource utilization and ameliorate the bias of node resource consumption while serving SLA.

**Key words**: microservice; load characteristic; container schedule; cloud computing; deep reinforcement learning

① 基金项目: 吉林省科技发展规划重点研发项目 (20200401076GX); 吉林省教育厅“十三五”科学技术研究规划项目 (JJKH20200678KJ); 符号计算与知识工程教育部重点实验室 2020 年度开放基金 (93K172020K05)
收稿时间: 2022-09-03; 修改时间: 2022-09-30; 采用时间: 2022-10-14; csa 在线出版时间: 2023-03-01
CNKI 网络首发时间: 2023-03-02



© 中国科学院软件研究所 http://www.c-s-a.org.cn

微服务 (microservice, MS) 是面向服务系统结构 (SOA)[1] 的一种变体架构方式, 它将复杂臃肿的单一服务根据业务领域或架构等原因拆分成耦合性较低、可单独部署的较小组件或服务, 通过同步或异步的方式进行通讯[2]. 由于微服务结构拥有降低故障影响、满足快速变化的业务需求、提高开发和运维效率等优势, 其已被业界广泛用. 随着电子商务浪潮的兴起, 企业用户越来越多的选择把服务部署在互联网数据中心 (Internet data center, IDC) 以此节约成本、提高效率.

由于不同服务对于资源维度的需求存在多样性, 多个微服务部署于同一物理节点时可能出现资源消耗倾斜的现象. 例如某些数据操作型业务会对磁盘 I/O 和网络带宽造成大量消耗, 对于内存和 CPU 资源的需求却低很多. 如图 1 为两个裸机节点 CPU 和内存资源利用率情况, 其中图 1(a) 所示节点资源消耗均衡, 而图 1(b) 上的资源消耗存在明显倾斜, 这种情况产生的资源碎片对集群的资源造成了浪费. 为了提高经济效益, 企业通常通过高效的调度算法进行动态资源管理和智能化容器部署方案, 尽量避免资源消耗倾斜, 让节点利用率处在最佳经济效益的状态区间.

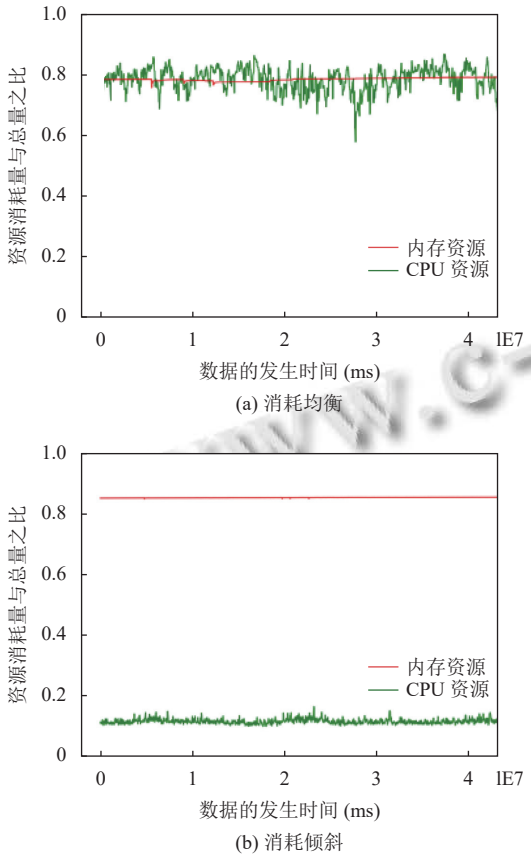



图 1 某节点内存-CPU 资源消耗情况

在提供 IaaS 服务的公共云中, 已有研究者针对容器资源自动管理问题进行研究[3–6]. Vinay 等人[7] 提出使用结合按需点播和 spot instance 提出了一种新的混合自动缩放策略, 并未针对在线服务需求变化频繁的特点进行优化. Habib 等人[8] 提出应用强化学习模型训练资源仲裁器和集群监视器的方式, 动态调整实例运行中的容器资源上限, 但这种方法没有考虑到微服务对资源消耗存在的特征, 使用过程中会频繁出现资源调整增加计算负担. John 等人[9] 提出一种将工作负荷预测与 Q 学习相结合的优化算法. Schuler 等人[10] 提出针对 Kubernetes 容器下 serverless服务, 使用 Q-learning 算法优化资源伸缩. 但该算法未使用深度神经网络对历史负荷进行感知, 且未针对容器资源利用情况进行优化[11–14].

在实例部署到节点的过程中, 需要为容器申请资源. 虽然节点中的容器可以通过 cGroup[15] 等技术共享内存和 CPU 等资源, 但是仍需要对容器使用上限进行限制. 在一条调用链中, 上游无状态服务的调用量激烈变化将对下游服务的调用量造成明显影响. 如果能感知上游服务的资源需求变化, 并提前合理的对下游高权重路径上的容器进行扩容/缩容, 将会极大提升调用链上的资源利用率. 如何合理地分配容器资源, 在实例 Qos 得到满足的条件下, 将节点的资源碎片率降到最低成为提升集群经济效益的关键问题之一.

深度强化学习具备高维数据特征感知能力、较强的自主决策能力, 常被用来解决复杂状态空间的决策问题, 常常在智能决策、实时仿真、游戏 AI 等领域发挥重要作用. 在本文研究中, 我们尝试使用深度双 Q 网络来决定容器资源分配策略, 让模型决定接下来一段时间内的容器资源上限, 进而达到释放节点资源碎片的目的.

本文使用阿里巴巴 2021 年发布的微服务交易跟踪作为主要数据来源, 通过对容器在裸机节点上资源消耗情况进行分析和对微服务在容器内的资源利用率进行整理挖掘, 我们发现容器资源分配存在不合理的情况, 进而导致裸机节点上的资源消耗倾斜, 产生资源碎片. 我们提出一种基于深度强化学习的调度模型, 通过对上游服务状态变化和节点资源消耗情况进行感知, 调整容器资源分配为手段, 有效提升了容器的资源利用率并缓解了节点资源消耗倾斜的情况.



© 中国科学院软件研究所 http://www.c-s-a.org.cn

## 1 跟踪信息

### 1.1 跟踪轨迹概览

尽管云计算资源管理和作业调度领域的研究受到越来越多的关注, 但公开可用的集群工作负载数据集仍然很少. 长期以来的主要开源数据集来源是: 谷歌集群跟踪[16] 和 Facebook 发布的 SWIM 跟踪[17] 等. 然而由于不同来源集群数据的异构性和特殊性使得其负载特征存在巨大差异[18]. 阿里巴巴集团 2018 年至今发布更新一系列跟踪轨迹, 包含多种领域的标准化、多维度集群运行信息和任务特征信息[19–22].

2021 年发布的 cluster–trace-microservices-v2021 (记为 CTM21)[23]记录了阿里巴巴大于一万个裸机节点上部署的近两万个微服务在 12 h 内的调用关系、运行状态、部署情况等详细信息. CTM21 采用 Kubernetes[24] 管理裸机云[25], 微服务在容器中运行, 由 Kubernetes 直接调度. 本文采用 CTM21 作为数据分析来源, 为了方便读者阅读本文其余部分, 此处对本文用到的部分数据结构和内在关联进行简述.

(1) 节点资源

节点表 (NodeTable.csv) 记录了裸机节点的实时运行信息, 每条数据都含有时间戳 (timestamp)、节点标识 ID (nodeid) 以及内存和 CPU 利用率. 某些节点未部署服务时处在关闭状态.

(2) 容器资源

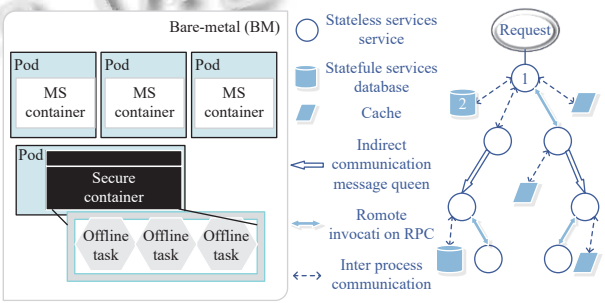
微服务以实例 (instance) 为最小单位部署在容器中, 通常一个微服务需要多个实例同时部署以满足业务的需要. 实例所处容器在节点上的部署情况、容器的资源消耗情况被详细记录在微服务资源表 (MS_Resource_Table.csv) 中. 其以时间戳 (timestamp) 对齐, 通过微服务名 (msname) 和服务实例名 (msinstanceid) 可获得实例与微服务的关系和实例运行资源消耗情况. 节点 ID (nodeid)、容器 CPU 利用率和内存利用率获取实例的运行对节点的影响. 值得注意的是, 由于有状态服务在其他专用集群中运行, 此处记载的实例均为无状态服务.

(3) 服务调用关系

在响应时间/调用率表 (MS_RT_Qps_Table.csv, RTQps) 和调用图表 (MS_CallGraph_Table.csv) 中记录了上下游服务的详细调用信息. 为了让研究人员更好的分析微服务的调用关系特征, 调用图表记录了多于 MS_Resource_Table 和 RTQps 中的微服务运行记录.

### 1.2 微服务体系结构

不同的服务类型对资源的需求不同, CTM21 采用在线服务和离线服务混合部署的方式, 提高裸机集群整体资源利用率. 图 2(a) 为 CTM21 集群体系结构, 裸机节点由 Kubernetes 分配 POD 进行资源管理, 其中 POD 部署了两种容器 (container): 在线微服务容器和专门为离线任务提供服务的安全容器. 安全容器相当于一个轻量级的虚拟机, 通过将离线任务容器调度到“安全容器”中实现与在线任务容器的资源隔离, 从而有效降低彼此之间的互相干扰, 保证在线任务容器的服务质量 (QoS).




(a) 微服务与离线任务混合部署结构 (b) 微服务调用图

图 2 微服务体系结构和调用图

在 CTM21 中根据状态不同, 微服务可分为无状态服务和有状态服务, 分别对应图 2(b) 中的圆形和平行四边形或圆柱体. 无状态服务对请求信息不予保存、服务无差别处理各个请求. 用户请求往往需要通过会话 (session)、内存缓存 (memCache)、数据库 (database) 等持久化层获取业务的上下文数据, 从而获得状态信息.

微服务之间的通信方式主要有 3 种: 进程间通信、远程 RPC 调用和异步调用. 当一次请求开始后, 将会触发涉及本业务的多个微服务之间一系列调用, 这些包含层级关系的调用被称为调用图 (call graph). 两个服务之间存在调用和被调用关系, 调用发起者称之为上游微服务 (upstream microservice, UM) , 调用承接者称为下游微服务 (downstream microservice, DM). 图 2(b) 中所展示了一次请求的完整调用图, 它揭示了有状态服务与无状态服务之间的调用关系和调用方式. 图中的 1 号无状态服务调用 2 号持久化服务过程中, 服务 1 作为 UM, 服务 2 作为 DM. 在 1 号服务接收请求后, 通过与 cache 和 database 的通信进行数据读写, 在本服务处理完成后向下一层业务所需的无状态服务传递请求, 在所有业务完成处理后结束本次请求.



© 中国科学院软件研究所 http://www.c-s-a.org.cn

## 2 资源消耗特征分析

### 2.1 指标计算

#### 2.1.1 相关性计算

皮尔森相关性系数 (Pearson product-moment correlation coefficient, $r$)[26], 是最常用的线性相关系数之一, 它常用来衡量两组变量的线性相关程度. $r$ 值介于 −1 到 1 之间, 绝对值越大表明相关性越强. 其具体关系如下:

$$\begin{cases} \text{strong correlation}, & 0.8 < |r| \leqslant 1 \\ \text{correlation}, & 0.3 \leqslant |r| \leqslant 0.8 \\ \text{weak or no correlation}, & |r| < 0.3 \end{cases} \tag{1}$$

为了理解多组指标数据之间的逻辑关系并衡量影响程度, 我们将相关性系数作为一种分析手段, 用来分析不同维度的资源消耗关系.

#### 2.1.2 资源碎片率计算

在理想的状态下, 假设宿主机同时拥有 $M$ 种可用资源. 在某段时间之中, 由于其第 $i$ 种资源到达部署上限, 而无法继续分配新任务时, 宿主机此时的资源碎片率 (debris rate, $D$) 应满足:

$$D = \frac{\sum_{i=1}^{n}\left(1 - \frac{\lambda_i \times U_i}{T_i}\right)}{M} \times 100\% \tag{2}$$

其中, $U_i$ 表示第 $i$ 种资源的使用量, $T_i$ 表示第 $i$ 种资源的总量, $\lambda_i$ 为第 $i$ 种资源的权重.

资源碎片率可用来衡量资源使用情况, 碎片率越高意味着节点中被浪费的资源越多, 可优化空间越大. 在实际工作环境中对于容器而言, 为了保证服务 QoS, 服务申请到的容器资源必须保证一段时间段内的峰值资源需求可以被满足. 为了满足其低频率出现的资源消耗峰值, 而申请远高于常态需求的资源, 会产生大量资源碎片. 对于裸机节点而言, 除了上述碎片来源外还有另一种情况: 节点在已部署部分容器后, 由于某个维度的资源已被容器预定达到阈值的而无法部署新的容器, 此时未被占满的其他种类资源均为碎片.

#### 2.1.3 内存/CPU 利用率差平均值计算

将跟踪轨迹的内存利用率和 CPU 利用率根据其时间戳对齐. 在长度为 $T$ 的内存/CPU 观测对序列中, 第 $i$ 对观测值的内存利用率记为$M_i$, CPU 利用率记为 $C_i$, 则观测周期内的利用率差平均值 $G$ 应满足:

$$G = \frac{\sum_{i=1}^{T}|C_i - M_i|}{T} \times 100\% \tag{3}$$

通过计算内存/CPU 利用率差平均值, 我们能够直观理解当前内存与 CPU 资源消耗的差距.

### 2.2 利用率变化原因分析

结合对部分节点中容器部署情况与服务调用图数据, 分析得到结论如表 1. 集群资源利用率变化的主要原因是上游服务请求的处理和微服务部署变动. 不同类型的上游服务请求对于 CPU 与内存利用率波动分别存在不同程度的影响[27], 而在没有请求需要处理的情况下 MS 的部署或释放对于内存的影响更加明显.

表 1 资源利用率变化原因

| 事件/影响 | 利用率上升原因 | 利用率下降原因 |
|---|---|---|
| UM请求 | 到达并执行 | 执行完毕释放资源 |
| MS调度 | 新的MS部署 | MS停止运行 |

### 2.3 资源利用率相关性分析

#### 2.3.1 数据处理与感知

将所有节点在全时段内的 CPU、内存利用率按照时间对齐, 可整理为三元组结构. 对每个节点计算 CPU 和内存利用率的相关性系数, 以分析节点中的内存资源和 CPU 资源消耗关系. 将所有节点的结果按照相关性系数值从小到大排序后绘制得到图 3(a). 图中纵坐标为相关性系数, 横坐标为其 rank 值. 由此将曲线划分为 3 个区域: 内存与 CPU 利用率呈负相关性区 ($r$<−0.3, 记为 A 区)、若相关性或无关区 (−0.3≤$r$<0.3, 记为 B 区)、正相关性 ($r$≥0.3, 记为 C 区).

表 2 记录了各区域所含节点信息. 由其可初步得出以下结论: 在观测样本中, 内存利用率波动情况比 CPU 更稳定. 在线请求往往对 CPU 的要求更高, 利用率波动也更为激烈. 当无状态服务部署达到稳定后, 内存使用情况普遍比较稳定, 只有在处理新计算型请求时会明显增加、处理完毕后触发内存回收才有可能出现大幅下降.

#### 2.3.2 相关性区域特征表达

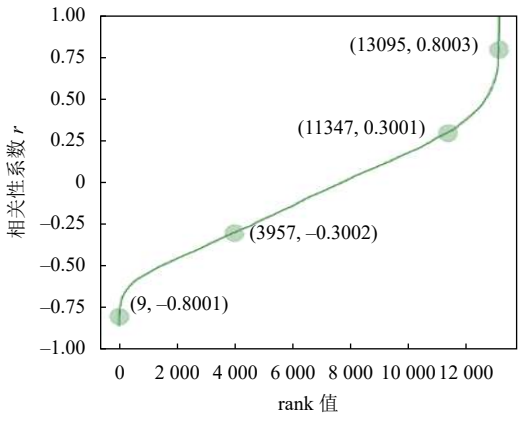
(1) 强相关性区域

在 A 区域中, 存在部分节点的内存、CPU 利用率具有强烈或明显负相关性将这类节点中部分具有代表性的 CPU、内存利用率按照时间戳绘制成图得到图 4(a), 由其结合对具体节点数据的分析, CPU 利用率波动主要出现在以下两种情况.

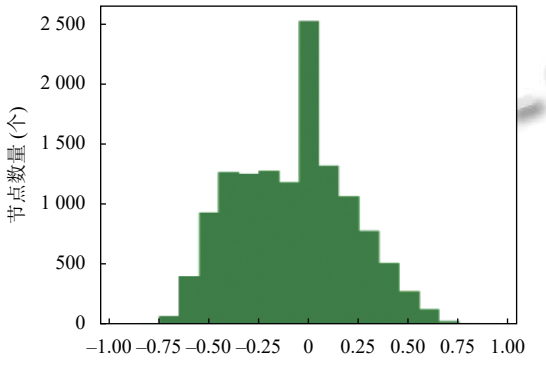
1) 利用率在多个相邻时间戳中出现较大幅度的周期性的增减, 且整体呈现下降趋势.



© 中国科学院软件研究所 http://www.c-s-a.org.cn




(a) $r$ 值 rank




(b) $r$ 值分布

图 3 CPU-内存利用率相关性分析结果图

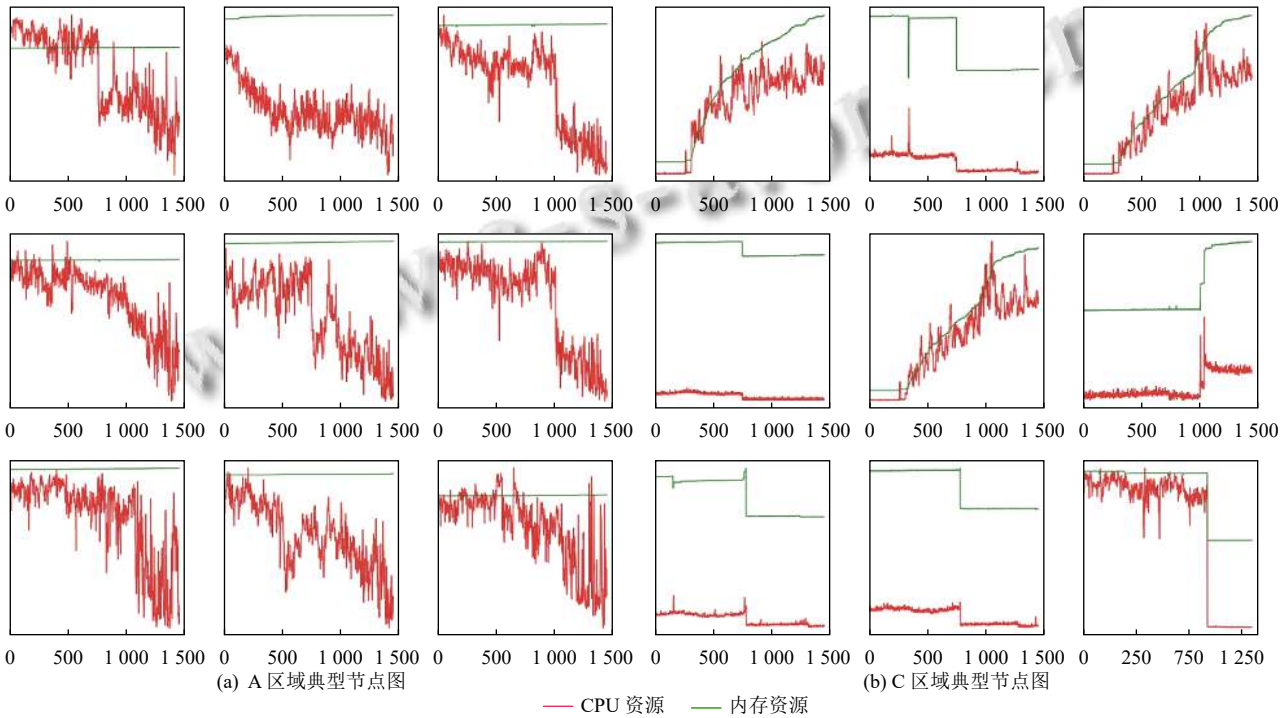
表 2 区域资源利用率详情

| 指标 | A区 | B区 | C区 |
|---|---|---|---|
| 总数 | 3 958 | 7 390 | 1 768 |
| CPU标准差均值 | 0.098 3 | 0.088 1 | 0.097 7 |
| 内存标准差均值 | 0.001 7 | 0.003 1 | 0.007 6 |
| CPU均值 | 0.636 7 | 0.640 2 | 0.606 0 |
| 内存均值 | 0.760 5 | 0.745 6 | 0.718 3 |
| CPU峰值数均值 | 0.799 4 | 0.796 7 | 0.779 1 |
| 内存峰值数均值 | 0.763 3 | 0.752 4 | 0.732 4 |
| $r$平均值 | −0.467 8 | −0.004 | 0.442 1 |
| $r$中位数 | −0.457 5 | −0.002 1 | 0.416 0 |

2) 利用率在某时刻开始崖式下跌. 由于内存利用率未随 CPU 利用率的整体下降而出现明显变化, 故其 $r$ 值表现出负相关性.

容器调度对内存和 CPU 影响具有同期性, 推测此类节点的利用率波动来源是执行上游服务的请求而非容器调度.可由此理解为调度器将部分 CPU 需求变化复杂、内存需求稳定型服务部署在此类节点中.

C 区域 CPU/内存利用率呈现出正相关选取部分具有代表性的节点 CPU/内存随时间变化的利用率曲线如图 4(b), 结合 instance 部署情况, 我们发现: C 区域中的 $r$ 值极高部分的节点利用率波动主要原因是节点中 instance 部署情况变化, 导致的资源回收或资源分配. 当部署变化后, 内存和 CPU 变化呈现同期性.




(a) A 区域典型节点图 (b) C 区域典型节点图

图 4 典型节点资源利用率图 (横坐标为时间周期序号, 纵坐标为某时刻对应资源消耗量与总量之比)



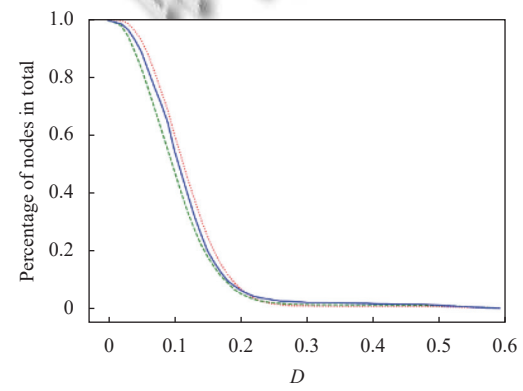
© 中国科学院软件研究所 http://www.c-s-a.org.cn

(2) 弱相关或无关区域

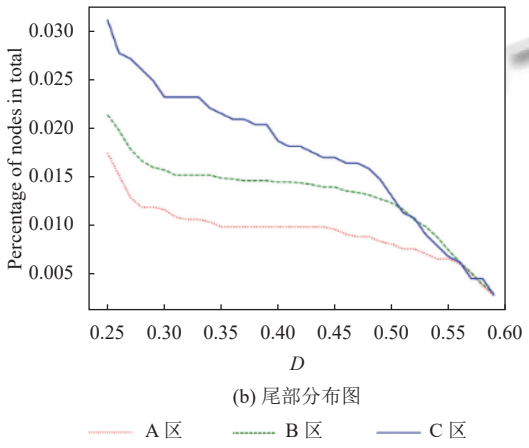
由表 2 可知, B 区域中稳定在较高的 CPU 消耗水平的节点数量较多, 观察其内存利用率均值、中位数和标准差均值可发现其内存利用率稳定在较高的利用率区间. 在 B 区观测时段内, 实例部署情况的变化较少.

### 2.4 利用率差分析

对第 2.3 节中得到的各区域 CPU/内存利用率差平均值 $D$ 进行计算分析, 发现 $D$ 值在部分区间的节点数占比满足长尾分布[28]. 尽管多数节点的 资源消耗较为平衡, 但在尾部仍有部分节点的资源消耗存在倾斜且不容忽视. 如图 5(a) 所示, 两张图横坐标是 $D$, 纵坐标是 $D$ 值对应节点数量占总数的百分比. 在图 5(b) 中展示了放大尾部的数据, 其节点数量占比随着 $r$ 值增加而增加. $D$ 值处于 0.5 附近的节点较多, 区域具有较高的优化价值. 由表 3 可知, $D$ 值大于等于 0.5 的节点仍有百余个, 提升其资源平衡度仍有较高的经济价值.




(a) 在全 $D$ 值区间上的节点分布图




(b) 尾部分布图

图 5 利用率差平均值图

在 CPU/内存相关性高的区域, 出现 CPU/内存差值均值大的概率就越高, 其资源消耗倾斜出现概率越大. 在优化容器调度算法时, 关注 $r$ 值较高的区域, 通过变更节点上的资源使用情况, 减少资源碎片从而提升整体利用率.

### 2.5 实例运行特征分析

在 MSResource 系列文件中记录了无状态服务的部署详情和微服务实例在每个时刻的具体资源消耗情况, 同时还包括微服务与实例的映射关系. 如图 6 所示, 一个微服务可能部署了多个实例, 他们共同协作以满足服务的功能需求. 实例部署依赖于集群的调度策略, 一个微服务的实例可能因节点状态和服务优先级等因素影响而部署到多个节点上. 然而从服务调度关系的角度出发, 某些上下游服务之间的调用率远高于平均水平, 将这类实例部署到同一个网域甚至同一节点上可以大幅提高响应效率[29].

表 3 各区域利用率差平均值数量

| $D$值 | A区 | B区 | C区 |
|---|---|---|---|
| 0.2 | 365 | 501 | 142 |
| 0.3 | 47 | 118 | 44 |
| 0.4 | 39 | 108 | 36 |
| 0.5 | 33 | 94 | 26 |

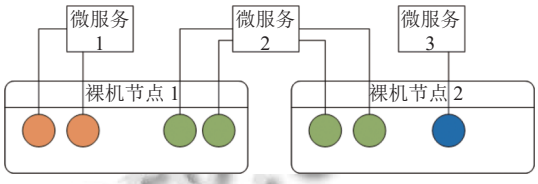



图 6 微服务实例部署关系图

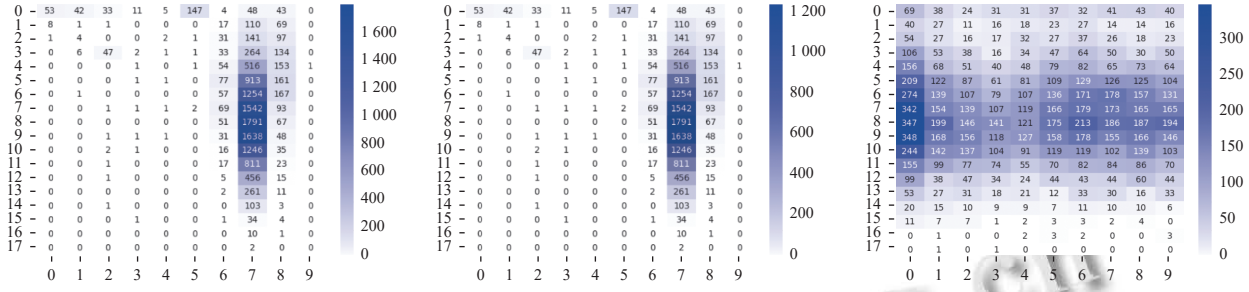
图 7 是容器数量与资源利用率热力图. 调度器在一个节点部署容器数量集中在 5–12 之间, 让节点内存利用率集中在 70% 左右, CPU 利用率集中在 60% 左右. 表 4 将节点和容器的资源消耗进行对比, 二者内存利用率差距并不明显, 然而 CPU 利用率却差别巨大. 容器的 CPU 使用均值和峰值比节点的对应指标低. 在部分容器 CPU 使用率上限为 200% 的情况下, 其峰值均值仍然徘徊在 30% 左右, 说明大量节点的 CPU 利用率处于低水平. 这也说明部分容器分配的资源不合理, 资源申请量高于周期内的实际使用量.

我们跟踪了节点中 CPU 利用率均值达到 70% 以上的 2 387 个节点, 和低于 30% 的 371 个节点. 发现节点 CPU 利用率高, 其部署的容器数量越多且其容器的 CPU 利用率均值越高. 我们还寻找了资源消耗更偏向



© 中国科学院软件研究所 http://www.c-s-a.org.cn

内存维度而造成 CPU 碎片率较高的 1 500 个节点, 发现他们部署的容器 CPU 利用率均值仅为 0.292 2, 标准差均值为 0.062 5, 这说明这类节点资源消耗倾斜与其部署的容器 CPU 利用率偏低存在关联关系.



(a) Memory-instance Number heatmap (b) CPU-instance Number heatmap (c) R-instance Number heatmap

图 7 容器数量热力图

表 4 节点与容器资源消耗对比

| 指标 | 节点 | 容器 |
|---|---|---|
| CPU均值 | 0.634 5 | 0.188 8 |
| CPU标准差均值 | 0.092 5 | 0.024 7 |
| CPU峰值均值 | 0.795 1 | 0.310 8 |
| 内存均值 | 0.746 4 | 0.655 9 |
| 内存标准差均值 | 0.003 3 | 0.008 9 |
| 内存峰值均值 | 0.753 0 | 0.669 5 |

## 3 容器资源分配优化研究

### 3.1 深度双 Q 网络

Q-learning 通过学习最优动作价值函数 $Q^*(s, a)$, 在给定状态 $s_t$ 下求得最优动作, 其核心思想是查询 Q 表中 $s$ 状态下所有可选动作的价值, 并选择价值最高的动作执行. 当状态和动作空间高维连续时 Q 表将变得十分巨大, 这对于 Q 表的维护和查询将变得极为困难. DQN 使用人工神经网络近似模拟 Q 表, 可以有效地对连续状态与动作空间进行特征提取, 使得特征相似的输入会得到相近的输出, 让 Q 学习泛化能力更强. 使用价值函数决策与策略梯度、actor-critic 等算法相比具有结构更简单、响应速度更快、可在线训练等特点, 更符合本文的应用环境.

在本文的问题中, 状态序列的前后关联性强. 为了减少这种关联性对训练造成的副作用, 我们使用了优先经验回放 (prioritized experience replay) 技术, 将状态序列顺序打乱, 并优先学习高误差值的经验. 深度双 Q 网络 (deep dual-Q network, DDQN) 是基于 DQN 改进而得来. 为了解决本文 DQN 训练中出现的非均匀高估值问题, 进一步加快训练速度和收敛速度而使用目标网络 (target network) , 使用一个 DQN 用来控制 agent 并收集经验 (transitions) , target network 用来提供价值估计 maxQ. 如式 (4) 描述了 DDQN 的目标值计算过程:

$$y_t = r_{t+1} + q \times (s_{t+1}, \arg\max_a q(s_{t+1}, a; w_e); w_t) \tag{4}$$

其中, $y_t$ 表示目标值, $\gamma$ 表示折扣系数, $w_e$ 表示评估网络参数, $w_t$ 表示目标网络参数. 在步长达到 $C$, 将评估网络权重整体更新至目标网络, 即使 $w_t$=$w_e$.

### 3.2 MDP 和人工神经网络结构

将若干连续时间视为一个决策周期, 智能体在获取到环境的状态反馈后按照一定策略进行下一步动作, 这类问题可以使用马尔可夫决策过程 (Markov decision process, MDP) 来进行建模. 本文方法中的 MDP 可以被定义为一个四元组, 用 $S$, $A$, $P$, $R$ 表示, 分别代表状态空间、动作空间、状态转移函数和奖励函数.

#### 3.2.1 状态空间

状态空间定义为一个三元组, 表示如下:

$$s = \{I_{\text{um}}, I_{\text{current}}, I_{\text{node}}\} \tag{5}$$

其中, $I_{\text{um}}$ 代表上游系列服务的性能指标, $I_{\text{current}}$ 代表容器当下的状态, $I_{\text{node}}$ 代表所部署节点的性能状态. 图 8 描述了状态转移的过程, $s_0$ 是初始状态, 经过某次动作 $a$ 后以概率 $p$ 到达 $s'$并最终在终止状态 $s_t$ 停止转移.

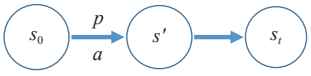



图 8 状态转移图

在模型的输入层包含两类数据: 智能体所处的状态信息$s_t$中环境给出的上游服务状态$I_{\text{um}_t}$, $I_{\text{um}_t}$是一个容



© 中国科学院软件研究所 http://www.c-s-a.org.cn

器的所有直接上游服务状态加权指标向量, 见式 (6). $CPU_{\text{avg\_}\theta}$, $CPU_{\text{max\_}\theta}$, $MEM_{\text{avg\_}\theta}$, $MEM_{\text{max\_}\theta}$ 分别代表 CPU 和内存在对应周期内的加权平均值和加权最大值, $G_\theta$ 代表 CPU 和内存利用率差值平均值 (见式 (3)), $T_\theta$ 则是上游服务吞吐量加权和.

$$I_{\text{um}_t}=(CPU_{\text{avg\_}\theta},CPU_{\text{max\_}\theta},MEM_{\text{avg\_}\theta},MEM_{\text{max\_}\theta},G_\theta,T_\theta) \tag{6}$$

例如, 图 9 所示的容器 $D$ 拥有 3 个直接上游服务, $D$ 在观测周期内的调用全部来自 $UM_a$、$UM_b$、$UM_c$, 其调用次数分别为 $a$, $b$, $c$. 则 $t$ 时段内三者所占权重分别是:

$$\theta_a=\frac{a}{a+b+c},\ \theta_b=\frac{b}{a+b+c},\ \theta_c=\frac{c}{a+b+c} \tag{7}$$

其计算方法见式 (8), 其中 $N_i$ 为第 $i$ 个上游服务对本服务的调用数量, $\sum N$是本服务的被调用数之和.

$$\theta_i=\frac{N_i}{\sum N} \tag{8}$$

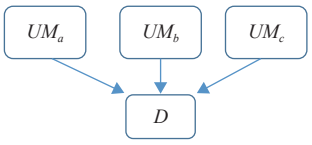



图 9 上游容器调用示例图

### 3.2.2 动作空间

在本模型中, 智能体决定在下一时间段内, 容器各维度资源分配上限变化情况. 动作空间的定义见式 (9):

$$A=\{add_i,reduce_j,keep_k\} \tag{9}$$

其中, $add_i$ 表示对容器中第 $i$ 个资源维度的上限进行扩容动作, $reduce_j$ 表示同时对第 $j$ 个资源维度的上限进行缩容动作, $keep_k$ 则代表了保持第 $k$ 个资源维度的上限不变. 在扩容和缩容动作中, 具体执行的扩缩规格应根据容器所在环境决定.

本文探讨 CPU 与内存两种资源维度情况下的优化问题. 以 CPU 内核分配数量$R_{CPU}$为例, $R_{CPU}\in(0,n]$ 根据可部署节点的最大可用内核数做调整. 将多个维度的资源上限离散化后做向量叉乘, 得到的结果向量即可作为动作空间, 这样可以保证每次的动作选择不会超出动作空间, 简化神经网络输出层的设计. 我们使用最大置信度上界 (UCB) 方法平衡探索和利用, 如式 (10) 所示, $q_a$ 是动作 $a$ 的初始评估价值, $n$ 是所有动作执行次数, $q_a$ 是 $a$ 动作执行的次数, $c$ 是平衡系数. 通过 UCB 算法, 我们期望智能体更趋近于探索从未选择过的动作.

$$q_{UCB}=q_a+c\times n^{\frac{1}{2}}\times n_a^{-1} \tag{10}$$

### 3.2.3 价值函数和更新策略

在本问题中, 智能体既要对容器本身的资源使用情况进行维护, 还要参考节点资源碎片和资源倾斜等情况. 智能体在 $S$ 状态下做出 $a$ 决策后, 奖励 $R$ 表达形式如式 (11):

$$R(S,a)=\begin{cases}\dfrac{\alpha_0\times\ln(K+1)}{R_{\text{container}}+R_{\text{node}}+\beta_0}, & \text{Not overflow}\\ -C, & \text{Overflow}\end{cases} \tag{11}$$

当周期内容器正常运行时, 根据式 (11) 计算奖励. 若某时段内的容器资源需求溢出, 将会给予负数奖励并立即结束当前动作周期, 开始新动作选择. 在公式中, $K$ 是所有维度连续选择 $keep$ 动作的周期数量, 我们称为动作惯性量. 我们鼓励智能体尽可能多的减少扩容和缩容行为, 原因在于资源上限的变更将加大容器被重新调度的概率, 这相当于增加了集群开销. 动作惯性系数 $\alpha\in(0,1)$, $C$、$\beta_0$ 是平衡常数.

$R_{\text{container}}$ 是根据容器运行指标综合评估而得, 具体见式 (12) . 其中, $CPU_{\text{avg}}$、$CPU_{\text{max}}$ 代表容器在 $t$ 时段内的 CPU 平均值和最大值. $MEM_{\text{avg}}$、$MEM_{\text{max}}$ 是内存平均值和最大值. $G_{\text{container}}$ 是对应时段容器中的利用率差值平均值, 计算方法见式 (3) . 通过 $R_{\text{container}}$ 衡量 $t$ 时段内动作 $a_t$ 对容器产生的影响. 我们鼓励容器在每个决策时段内的利用率都尽量接近所分配的资源上限, 并且必须保证节点内的资源倾斜程度尽量降低. $\gamma_i$ 是平衡系数.

$$R_{\text{container}}=\frac{\gamma_1\times CPU_{\text{avg}}+\gamma_2\times CPU_{\text{max}}+\gamma_3\times MEM_{\text{avg}}+\gamma_4\times MEM_{\text{max}}}{\gamma_5\times G_{\text{container}}} \tag{12}$$

$R_{\text{node}}$ 是容器所部署的节点运行指标综合评估而得, 具体见式 (13). $D_{\text{node}}$ 是节点在对应时段内的资源碎片率, 其计算方式见式 (2). $G_{\text{node}}$ 是节点在对应时段中的 CPU 和内存利用率差值平均值. 我们希望容器所部署的节点能够产生尽量少的资源碎片, 提升其资源消耗的平衡性. $\gamma_i$ 是平衡系数.

$$R_{\text{node}}=\gamma_6\times G_{\text{node}}+\gamma_7\times D_{\text{node}} \tag{13}$$

关于更新策略, 考虑到容器在某时刻出现的动作,



© 中国科学院软件研究所 http://www.c-s-a.org.cn

其原因可能是之前的动作不合理导致的. 如图 10 为了让智能体的动作拥有更加长远的考虑, 我们使用式 (14) 求得的 $t$ 时刻奖励 $r_t$ 的折扣值$r'_i$, 对之前 $m$ 次动作进行折扣更新, 其中 $i$ 代表被折扣的奖励.

$$r'_i=\begin{cases} r_t^{\frac{1}{t-i+1}}, & i<t \\ r_t, & i=t \end{cases} \tag{14}$$

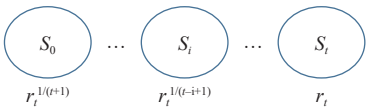



图 10 更新策略示意

图 11 所示, 对神经网络输入以下类数据: 环境反馈的容器当前状态、节点当前状态、上游服务加权状态向量, 经过 5 层 CNN 网络提取特征后, 进入 2 层全连接层, 最后是动作空间映射. 我们的实验中, 动作空间是离散的. 为了简化模型的实现, 我们把智能体在每个状态下的可选择的行动制作成了表格. 每个容器最多申请 4 个 CPU 核心和 4 GB 内存.

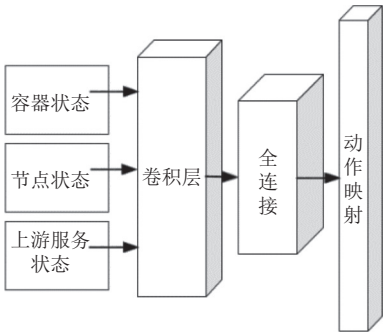



图 11 网络结构示意

本文使用 CloudSim 仿真框架[30] 作为实验平台, CloudSim 是一个用于在云计算中建模、模拟场景的工具包, 它可以模拟出机器的负载情况和容器调用情况. 我们对 CloudSim 进行了扩展, 重写了其中部分方法, 增加其作为智能体运行环境, 计算并反馈节点状态、容器状态、上游服务加权向量的能力. 我们以 JSON 格式在 Java 开发的 CloudSim 与 Python 训练的智能体进行数据交互, 在 CPU 和内存两个资源维度进行实验. 如表 5, 我们在 CloudSim 中设置了两种资源规格的节点, 以此降低由资源规格不合理对消耗倾斜的影响. 由于模型将节点的运行情况作为状态空间的一部分, 为了让模型更具普适性, 我们在容器的调度上选择了 2 种经典调度算法: 先到先匹配算法 (FF) 和最高降序调度算法 (FFD). FD 是选择每次放置容器后资源碎片率最低的节点进行调度. 两种容器调度算法各控制一半数量的节点.

表 5 节点规格

| CPU | 内存 (GB) | 数量 |
|---|---|---|
| 64核 | 256 | 32 |
| 32核 | 64 | 64 |

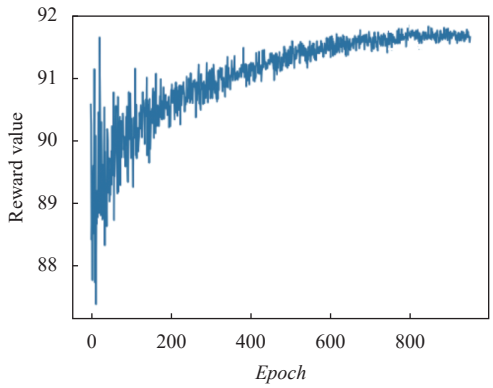
我们根据 CTM21 的调用关系和容器利用率信息, 整理出分布在 330 个拥有不同特征的完整调用链上, 共计 1 722 个容器的仿真数据. 每个容器利用率信息数据均包含 1 440 个时间周期, 每个周期内提供了本阶段的 CPU 需求量和内存需求量. 这些仿真数据共 2 个文件: 记录了调用关系信息的 calltable.csv, 其中字段包括了 timestamp、UM、DM 等. 记录容器运行状态的 containerState.csv, 其中字段包括 timestamp, containerID, cpuNeed, memNeed, 二者根据 timestamp 对齐, DM 是 containerID 的外键. 实验中我们使用 MySQL 存储实验数据, 以便于在 CloudSim 中快速查询状态信息等数据. 为了加快模型收敛速度, 我们在开始的阶段使用了小规模数据样本进行初始化训练. 将每一个 *epoch* 的长度控制在 64 步之内, 通过挑选出 30 组具有代表性的调用链数据, 我们在 11 个节点上部署了 225 个容器, 对智能体进行了 100 轮的初步训练.

如图 12(a) 显示了 *epoch*=1000 的训练结果. 由于采用了小样本训练初始化了网络参数, agent 在训练初期表现出比较好的水平. Agent 在经过 650 轮训练后, 奖励逐渐收敛于 91.5 分. 在前 200 轮训练中, agent 表现出的不稳定性主要来自于使用 UCB 算法对未知动作的探索. 此情况说明预训练好的网络模型对收敛速度起到了正向作用.

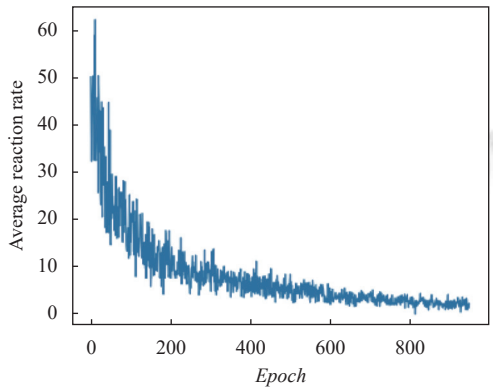
在实验过程中, 我们发现如果 $K$ 值设置过高, 将导致智能体更倾向于在前几个动作空间就做出较大的资源占用, 使整体资源利用率偏低, 而 $K$ 值设置过低则会出现容器 QoS 低下, 经常出现 CPU 占用超出设置上限的情况. 经过几轮调整得到的一组较为均衡的参数, 其 QoS 在训练过程中变化情况见图 12(b). 其中纵坐标为容器出现资源需求超出分配份额的决策周期占全部决策周期的比例. 在 *epoch* 达到 800 后, 出现容器分配资源不能满足需求的情况占到 3.32%, 容器的平均可用情况达到了 96.68%.



© 中国科学院软件研究所 http://www.c-s-a.org.cn




(a) 奖励 $r$ 随训练变化情况




(b) 平均重部署次数占比情况

图 12 训练情况分析

为了对比观察模型效果, 我们设置了 2 种其他虚拟机配置选择方案. 最小选择法 (MS) 是在所有规格的容器中默认选择最小配置进行初始部署, 当出现资源需求超出上限的情况时, 选择更大一号的资源配置. 随机部署法 (MN) , 根据 MS 方法运行后产生的配置数量生成选择概率. 新容器按概率选择一款配置, 当资源需求超出上限时, 选择更大一号的资源配置. 我们为这两种选择算法添加了 4 个不同的容器规格见表 6. 模型可以选择任意小于 4 核的 CPU 和小于 4 GB 的内存, 内存调整粒度是 1 GB.

表 6 对照组虚拟机规格

| 配置 | CPU | 内存 (GB) |
|---|---|---|
| 1 | 1核 | 2 |
| 2 | 2核 | 2 |
| 3 | 2核 | 4 |
| 4 | 4核 | 4 |

将训练好的 DDQN 模型和两种经典算法使用 170 组容器数据进行测试, 对比结果见表 7. 因 CTM21 是真实生产轨迹数据, 其集群使用了资源共享等技术, 并且其资源细分的粒度要精细于仿真数据, 故此处实验结果不能直接同第 3 节的轨迹分析结果直接对比. 我们参考 DDQN 与 MS、MN 的结果, 分析得出了以下两个结论.

1) 本文提出的 DDQN 模型, 其 CPU 利用率和内存利用率远高于其他两种经典算法, 原因在于 DDQN 能够根据状态 $s$ 动态调整容器资源上限, 当容器处于低效运行时, 模型将回收部分资源以达到贴合容器运行特征的效果. 而其他两种模型则仅有资源扩展, 没有资源回收功能.

2) 在容器资源使用维度观察, 本文的 DDQN 模型 CPU 利用率远远高于对比算法 MS 和 MN, 而内存利用率 DDQN 与 MS 相差并不明显. 由于 MN 倾向于初始就选择较高的配置, 所以其资源利用率最低. 得益于对容器的资源管理, DDQN 在节点维度的表现比较优异, 它有效缓解了节点资源消耗倾斜的情况, 相比于 MS 算法的明显倾斜, DQN 将节点利用率差值均值控制在了 11%.

表 7 对照实验结果

| 属性 | DDQN (Our) | MS | MN |
|---|---|---|---|
| 节点最高占用数 | 22 | 29 | 44 |
| 节点利用率差值均值 (%) | 11 | 37 | 29 |
| 节点资源碎片率均值 (%) | 7 | 11 | 15 |
| 容器CPU利用率均值 (%) | 73 | 22 | 19 |
| 容器内存利用率均值 (%) | 85 | 78 | 66 |
| 资源上限变更的周期比 (%) | 37 | 35 | 19 |
| 时段内容器QoS达标率 (%) | 91 | 66 | 78 |

由此可知, 与经典算法相比, 具备资源回收功能的 DDQN 方法在资源利用率方面和节点资源碎片率方面都拥有明显优势. 它的部署手段更加灵活、集群状态感知更加敏感. 虽然在 QoS 达标率方面, DDQN 获得了远高于其他两种方法的成绩, 但低于训练数据的 96.68%, 模型可能对训练数据拟合较高. 且其资源上限变更占比在所有方法中最高, 频繁变更容器上限将对容器在节点上的部署造成压力, 这些问题有待后续研究优化.

## 4 结论与展望

本文通过对 CTM21 轨迹数据的分析, 发现大规模集群中的节点存在资源消耗倾斜的情况. 深入挖掘其原因, 我们发现容器资源分配过量是导致节点资源消



© 中国科学院软件研究所 http://www.c-s-a.org.cn

耗倾斜的原因之一. 由此我们提出了一种基于深度双 Q 网络的容器资源优化模型, 通过对上游微服务的运行情况和容器状态进行感知, 让智能体对容器资源分配进行自动维护, 以达到优化容器资源利用率、改善集群节点资源消耗倾斜的情况. 随后使用 CTM21 数据制作的仿真数据在 CloudSim 仿真环境中对模型进行了验证. 经过验证, 本文提出的模型可以在较大规模的集群环境中, 针对复杂的服务调用环境提供有效的容器 CPU——内存资源管理、并对节点的资源倾斜起到调整的作用, 提高了节点运行效率, 能够有效节约集群运行成本.

考虑到影响节点资源消耗平衡因素的多样性, 后续工作将从优化容器部署角度和增加资源维度的方面入手. 尝试通过对调度时机选择、调度对象的选择以及加入容器网络带宽、硬盘 I/O 等资源的管理, 进一步优化节点的资源消耗情况, 提升集群整体经济效益.

## 参考文献


1 魏东, 陈晓江, 房鼎益. 基于 SOA 体系结构的软件开发方法研究. 微电子学与计算机, 2005, 22(6): 73–76. [doi: 10.3969/j.issn.1000-7180.2005.06.020]

2 Birrell AD, Nelson BJ. Implementing remote procedure calls. ACM Transactions on Computer Systems, 1984, 2(1): 39–59. [doi: 10.1145/2080.357392]

3 马武彬, 王锐, 王威超, 等. 基于进化多目标优化的微服务组合部署与调度策略. 系统工程与电子技术, 2020, 42(1): 90–100. [doi: 10.3969/j.issn.1001-506X.2020.01.13]

4 Qiu HR, Banerjee SS, Jha S, *et al*. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020. 805–825.

5 单好民. 基于改进蚁群算法和粒子群算法的云计算资源调度. 计算机系统应用, 2017, 26(6): 187–192. [doi: 10.15888/j.cnki.csa.005870]

6 李轲, 窦亮, 杨静. 面向 Kubernetes 的多集群资源监控方案. 计算机系统应用, 2022, 31(7): 77–84. [doi: 10.15888/j.cnki.csa.008565]

7 Vinay K, Kumar SMD. Virtual machine based hybrid auto-scaling for large scale scientific workflows in cloud computing. Proceedings of the 3rd International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC). Palladam: IEEE, 2019. 526–530. [doi: 10.1109/I-SMAC47947.2019.9032507]

8 Habib A, Khan MI. Reinforcement learning based autonomic virtual machine management in clouds. Proceedings of the 5th International Conference on Informatics, Electronics and Vision (ICIEV). Dhaka: IEEE, 2016. 1083–1088. [doi: 10.1109/iciev.2016.7760166]

9 John I, Sreekantan A, Bhatnagar S. Auto-scaling resources for cloud applications using reinforcement learning. Proceedings of the 2019 Grace Hopper Celebration India (GHCI). Bangalore: IEEE, 2019. 1–5. [doi: 10.1109/ghci47972.2019.9071835]

10 Schuler L, Jamil S, Kühl N. AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). Melbourne: IEEE, 2021. 804–811. [doi: 10.1109/ccgrid51090.2021.00098]

11 Yazdanov L, Fetzer C. VScaler: Autonomic virtual machine scaling. Proceedings of the 2013 IEEE 6th International Conference on Cloud Computing. Santa Clara: IEEE, 2013. 212–219. [doi: 10.1109/cloud.2013.142]

12 傅德泉, 杨立坚, 陈哲毅. 基于深度强化学习的云软件服务自适应资源分配方法. 计算机应用, 2022, 42(S1): 201–207. [doi: 10.11772/j.issn.1001-9081.2021111993]

13 Tian HS, Zheng YC, Wang W. Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud. Proceedings of the ACM Symposium on Cloud Computing. Santa Cruz: ACM, 2019. 139–151. [doi: 10.1145/3357223.3362710]

14 Liu QX, Yu ZB. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace. Proceedings of the ACM Symposium on Cloud Computing. Carlsbad: ACM, 2018. 347–360. [doi: 10.1145/3267809.3267830]

15 ArchWiki. cGroups. https://wiki.archlinux.org/title/Cgroups. (2022-06-25)[2022-09-20].

16 Reiss C, Wilkes J, Hellerstein JL. Google cluster-usage traces: Format + schema. Google Inc., 2011. 1–17.

17 Chen YP, Alspaugh S, Katz R. Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads. Proceedings of the VLDB Endowment, 2012, 5(12): 1802–1813. [doi: 10.14778/2367502.2367519]

18 Amvrosiadis G, Park JW, Ganger GR, *et al*. On the diversity of cluster workloads and its impact on research results. Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference. Boston: USENIX




© 中国科学院软件研究所 http://www.c-s-a.org.cn

Association, 2018. 533–546.

19 Zhang Z, Li C, Tao YY, *et al*. Fuxi: A fault-tolerant resource management and job scheduling system at Internet scale. Proceedings of the VLDB Endowment, 2014, 7(13): 1393–1404. [doi: 10.14778/2733004.2733012]

20 Felidsche. cluster-trace-v2017. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2017. (2021-06-23) [2022-09-20].

21 MyGodItsFull0fStars. cluster-trace-v2018. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018. (2022-07-27)[2022-09-20].

22 Qzweng. cluster-trace-gpu-v2020. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-gpu-v2020. (2022-09-17) [2022-09-20].

23 Niewuya. cluster-trace-microservices-v2021. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021. (2022-04-05)[2022-09-20].

24 Brewer EA. Kubernetes and the path to cloud native. Proceedings of the 6th ACM Symposium on Cloud Computing. Kohala Coast: ACM, 2015. 167.

25 Zhang XT, Zheng X, Wang Z, *et al*. High-density multi-tenant bare-metal cloud. Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 483–495. [doi: 10.1145/3373376.3378507]

26 Lee Rodgers J, Nicewander WA. Thirteen ways to look at the correlation coefficient. The American Statistician, 1988, 42(1): 59–66. [doi: 10.1080/00031305.1988.10475524]

27 Luo ST, Xu HL, Lu CZ, *et al*. Characterizing microservice dependency and performance: Alibaba trace analysis. Proceedings of the ACM Symposium on Cloud Computing. Seattle: ACM, 2021. 412–426.

28 Downey AB. Evidence for long-tailed distributions in the Internet. Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement. San Francisco: ACM, 2001. 229–241. [doi: 10.1145/505202.505230]

29 Esparrachiari S, Reilly T, Rentz A. Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. Queue, 2018, 16(4): 44–65. [doi: 10.1145/3277539.3277541]

30 Calheiros RN, Ranjan R, Beloglazov A, *et al*. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience, 2011, 41(1): 23–50. [doi: 10.1002/spe.995]

(校对责编: 牛欣悦)

www.c-s-a.org.cn



© 中国科学院软件研究所 http://www.c-s-a.org.cn