计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(7):226−239 [doi: 10.15888/j.cnki.csa.009180]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

容器编排工具中部署工作节点的资源优化①

谢兆贤, 张文静, 徐　娅, 王若冰, 倪冰雪

(曲阜师范大学 网络空间安全学院, 曲阜 273165)
通信作者: 张文静, E-mail: 2336505499@qq.com

摘　要: 随着云计算飞速发展, 以 Docker 为代表的容器技术逐渐被重视. 目前, 3 种常见的容器编排工具有 Kubernetes、Docker Swarm 和 Rancher. 然而, 现有的容器编排工具在所有工作节点的总容量超标时, 将会有响应时间长和资源占用较多等问题. 因此, 本文设计 LSD (least space unused) 算法以及 LRU-SD (least recently used and space unused) 算法, 并应用于 3 种编排工具中. 当总容量超出上限时, 则选择删除不工作的节点并且增加新的工作节点. 做法上, LSD 算法是删除剩余空间最少的工作节点, LRU-SD 算法先考虑删除最久未使用的节点, 当有多个符合要求的节点时, 则删除剩余空间最少的工作节点. 实验部分, 分析与比较使用不同算法对 3 种容器编排工具的影响, 包含响应时间、CPU 和内存. 实验结果发现, LSD 算法、LRU-SD 算法和 LRU 算法不仅能够提高编排工具的响应时间, 还可以增加资源的使用率. 同时, 在提升 CPU 的使用率方面, LRU-SD 算法的效果最好.

关键词: LRU; 容器编排; Docker; Kubernetes; Docker Swarm; Rancher

引用格式: 谢兆贤,张文静,徐娅,王若冰,倪冰雪.容器编排工具中部署工作节点的资源优化.计算机系统应用,2023,32(7):226–239. http://www.c-s-a.org.cn/1003-3254/9180.html

Resource Optimization of Deployment Working Nodes in Container Orchestration Tools

XIE Zhao-Xian, ZHANG Wen-Jing, XU Ya, WANG Ruo-Bing, NI Bing-Xue

(School of Cyber Science and Engineering, Qufu Normal University, Qufu 273165, China)

Abstract: As cloud computing rapidly develops, container technology, represented by Docker, has been gradually paid attention to. At present, three common container orchestration tools are Kubernetes, Docker Swarm, and Rancher. However, when the total capacity of all working nodes exceeds the limit, the existing container orchestration tools will have problems such as long response time and large resource occupation. Therefore, the least space unused (LSD) algorithm and least recently used and space unused (LRU-SD) algorithm are designed in this study and applied to three kinds of orchestration tools. When the total capacity exceeds the upper limit, the non-working nodes are deleted and new working nodes are added. In practice, the LSD algorithm deletes the working node with the least remaining space, while the LRU-SD algorithm first considers deleting the longest unused node. When there are multiple qualified nodes, the working node with the least remaining space is deleted. In the experiment part, the impacts of different algorithms on three container orchestration tools are analyzed and compared in terms of response time, CPU, and memory. The experimental results show that the LSD algorithm, the LRU-SD algorithm, and the LRU algorithm can not only improve the response time of the orchestration tools but also increase the utilization of resources. At the same time, the LRU-SD algorithm is the most effective in improving CPU utilization.

Key words: least recently used (LRU); container orchestration; Docker; Kubernetes; Docker Swarm; Rancher

① 基金项目: 山东省自然科学基金面上项目 (ZR2020MF048)
收稿时间: 2023-01-03; 修改时间: 2023-02-03, 2023-02-27; 采用时间: 2023-03-03; csa 在线出版时间: 2023-05-24
CNKI 网络首发时间: 2023-05-25

在虚拟化技术[1]的应用上, 存在移植性不方便和部署繁琐等问题, 容器技术的易迁移、轻量级和高效率等特性可以有效地改善问题[2]. 同时, 容器在持续部署和微服务等业务中利用率高, 可以提高科学计算领域的生产力[3]. Docker 是容器技术的一个应用, 它成功地解决 LXC (Linux container) 集成度低和自动化水平低等问题[4]. 随着 Docker 的发展[5,6] 不少公司广泛地使用 Docker 技术, 例如: Redhat 、Google 等[7], 成为现在最热门的开源应用容器引擎.

容器和组件一样需要管理和监控, 比如 Docker 命令行接口 (command line interface, CLI) 管理单个主机上的容器, 但是要满足在多个主机上部署管理容器则必须使用容器编排工具. 容器编排过程是指在大量机器集群上管理容器化应用程序, 一般涉及自动化应用管理的工具, 通过容器编排技术可以轻松完成部署容器、部署服务、扩展和调度. 使用容器编排工具不仅可以提高用户在开发、测试和部署时的效率, 还可以在管理容器、部署服务等业务上呈现出方便与高效等优点. 根据不同的架构环境, 越来越多的编排工具展现在大众面前, 其中有不少工具已经获得用户的认可和采用, 例如: OpenShift、Docker Swarm、Nomad、Open Nebula、Marathon、Kubernetes、Rancher 和 Cloudify. 其中以 Kubernetes、Docker Swarm 和 Rancher 这 3 个编排工具的使用率最高. Coasta 等[8] 研究容器编排系统中软件老化对 Kubernetes 的影响, Piedade 等[9] 为 Docker 的编排提供一个完整的视觉符号, 可以减少工作量、错误率和开发时间. 也可以利用最现代的技术和架构模式, 设计高效的云编译器. Heidari 等[10] 提出一种云编译器的架构设计, 它与 Kubernetes 等编排技术完全兼容, 提供更高层次的可扩展性、可靠性、安全性和可维护性. Rovnyagin 等[11] 提出一种系统性能的动态编排方法, 以增加数据处理系统的效能.

虽然现有的容器编排工具已经发展较为成熟, 它仍旧存在功能不够完善、部署繁琐、响应速度不够快和资料占用较多等问题. 当编排工具容量达到最大值时, 但是还有新工作节点想要继续进行工作, 那么需要删除其余某个不工作的节点, 释放一定的空间, 来达到新工作节点的空间需求. 算法上, Docker Swarm 和 Rancher 默认使用传统式算法, 即删除最后一个插入的工作节点, 再插入新的工作节点. Kubernetes 默认使用 LRU (least recently used) 算法来释放不工作的节点, 规则是删除最久未使用的工作节点. 传统式算法可能会导致删除的工作节点是最频繁使用的工作节点, 而真正需要释放的工作节点没有被删除. 例如: 插入工作节点 1 和工作节点 2 后已经达到了最大容量, 想要插入工作节点 3 时, 则需要默认删除工作节点 2, 再插入工作节点 3. 如果插入工作节点 3 后又需要使用工作节点 2, 那么默认是再删除工作节点 3, 插入工作节点 2. 此过程繁冗, 一直在做插入工作节点和删除工作节点的重复工作, 浪费了资源, 而空闲的工作节点 1 才是应该被释放的. 如果一开始先删除了工作节点 1, 那么集群内剩余了工作节点 2 和工作节点 3 可以交互工作, 省去了不必要的重复操作. 这种传统式做法, 不仅使得工作效率低下, 还浪费资源. 想要删除正确的工作节点, 必须制作相应的规则, 按照规则选择需要删除的工作节点. 为了改善这些问题, 提高开发人员和管理人员的工作效率, 实现更好的容器化, 本文提出将 LRU 算法融入 Docker Swarm 和 Rancher 中, 并且设计了 LSD (least remaining space) 和 LRU-SD 算法, LSD 算法是删除剩余空间最少的工作节点, LRU-SD 算法同时考虑了最近最久未使用和剩余空间两个指标, 然后将这两个算法融入 Kubernetes、Docker Swarm 和 Rancher 等 3 种编排工具. 本文做出的贡献有:

(1) 设计 LSD 算法和 LRU-SD 算法, 优化 Kubernetes、Docker Swarm 和 Rancher 的释放节点过程.

(2) 首次提出将 LRU 算法引用到 Docker Swarm 和 Rancher, 改善编排工具的释放节点过程.

(3) 比较 LRU-SD 算法、LSD 算法和 LRU 算法, 发现三者都可以提高工作效率及资源使用率. LRU-SD 算法的工作效率优于 LSD 算法和 LRU 算法.

首先在引言部分介绍容器技术、容器编排技术及工具, 提出目前编排工具存在的问题. 然后在相关工作部分, 详细介绍 Docker、Kubernetes、Docker Swarm 和 Rancher. 在编排系统部分, 描述 3 种编排工具的部署过程, 列出传统式、LSD 算法、LRU 算法和 LRU-SD 算法的具体内容. 在实验分析部分, 比较不同算法在 3 种编排工具下的工作效率及资源使用率, 对实验结果进行分析与总结. 最后, 归纳本文与展望未来.

1 相关工作

1.1 Docker 及容器编排技术

Docker 获得越来越多的用户喜爱, 性能高于虚拟

机[12], 然而, 即便 Docker 容器具备启动时间短的优势, 也仍存在隔离性弱等安全威胁的困扰[13]. Docker 采用 C/S (client/server) 模式的架构, 即客户端/服务器模式的架构. 其主要模块有 Docker 客户端、Docker 主机和 Docker 镜像注册中心.

随着需要管理的节点和容器的增多, 手动管理便不能满足多个容器的管理. 所以, 需要一种集结容器、自动化和规模化于一体的集群管理平台[14], 即容器编排工具. 文献 [15] 通过虚拟机、容器结合 YAML (YAML ain't a markup language) 和容器结合可视化等 3 种方式, 分别对部署的时间、服务启停的速度和扩展的速度进行测试, 结果发现容器编排可视化的方式效率最高, 甚至比传统虚拟机的方式提高一个到两个时间级别, 给开发人员带来极大的便利. 容器编排技术是组织多个容器的流程, 将部署、调度和管理等方面都成为自动化处理, 这样不只可以提升管理的效率, 还可以解决容器在管理和调度不便的问题, 并且提高安全性[16].

1.2 Kubernetes

Kubernetes, 人们称为 k8s, 数字 8 代替中间 8 个字符而形成. 它是由 Google 开发的开源容器编排工具[17], 用于容器云集群中自动部署与管理容器应用的编排系统[18]. Kubernetes 有很多利于用户使用的优点, 首先它拥有许多的自动化操作, 表现在可以自动部署、自动弹性伸缩和自动扩展等. 并且, 还支持公有云、私有云和混合云等具备易于移植的特性[19]. Kubernetes 主要由主节点和从节点所构成[20]. 节点即为主机, 可以是虚拟机或物理机.

Pod 是最小的工作节点[21], 按照调度策略将 pod 部署到运行状况良好与最适宜的节点; 若没有合适的节点, 则将 pod 置于挂起状态, 直到出现合适的节点. 而 controller manager 会检测要控制的节点及容器的当前状态, 负责维护集群的状态. 当有容器停止运行时, 它可以进行故障检测或回滚等方式解决问题. 当容器创建完成后, Apiserver 会自动调用 Kubelet 组件来启动容器, Kubelet 还可以监视一个或多个容器组合在一起的共享资源 pod, 并且负责维护容器的生命周期和节点间的通信[22].

1.3 Docker Swarm

为了使 Docker 容器发挥更大的作用, 需要构建集群, 以便管理和强化功能. 作为编排工具的 Docker Swarm 即可管理集群中的镜像和容器. Docker Swarm 包括 Manager 节点和 Worker 节点. Manager 节点负责控制集群, 将请求从外部传到指定的 Worker 节点[23], Worker 节点接收到任务并执行服务. Docker Swarm 通过 API (application programming interface) 接口与外界联系, 而且轻量和操作简单, 所以有利于用户使用. 内置的策略有 Spread、Random、和 Binpack, Spread 为默认的调度策略[24].

1.4 Rancher

Rancher 是一个企业级的、开源的容器编排工具, 它能够简单快捷的部署管理主机及容器. Rancher 除了支持内置的 Cattle 编排环境之外, 还支持 Kubernetes、Docker Swarm 和 Mesos. 用户可以使用它构建 Kubernetes 集群, 甚至直接导入已经存在的 Kubernetes 集群, 因此 Rancher 既能管理容器又能管理 Kubernetes 集群, 还可以对它们的健康状况进行监控. Rancher 还提供可视化接口、内部包含应用商店、支持弹性伸缩、易于添加管理主机. 相较于其他的工具, Rancher 不仅在添加单个主机时操作方便, 更在创建集群时节省很多的步骤. 缺点是, Rancher 对 Docker 的安装版本要求严格, 不同版本的 Rancher 需要安装相对应的 CentOS 版本和 Docker 版本.

2 编排系统

2.1 编排框架

本文提出一种新的想法, 在 Docker Swarm 和 Rancher 中运用 LRU 算法, 提高这两种工具在有限容量与数据较多时的响应速度, 并且设计 LSD 算法和 LRU-SD 算法, 并将这两种算法使用于 3 种容器编排工具. 编排整体架构图如图 1 所示. 用户通过 API 发送指令, 主节点中的 controller manager 和 scheduler 分别是维护集群的状态和负责资源的调度.

2.2 编排工具部署

图 2 展示 3 个工具运行使用的主要步骤. 首先将所需的软件包通过 FileZilla 上传到虚拟机指定目录下, 并且在所有节点都安装 Docker. 其次在主节点创建集群, 并将从节点加入集群. 然后在主节点和从节点拉取相应镜像, 并启动对应容器. 最后可通过浏览器访问容器编排工具的可视化界面, 在此界面可以对容器及节点做相应操作.

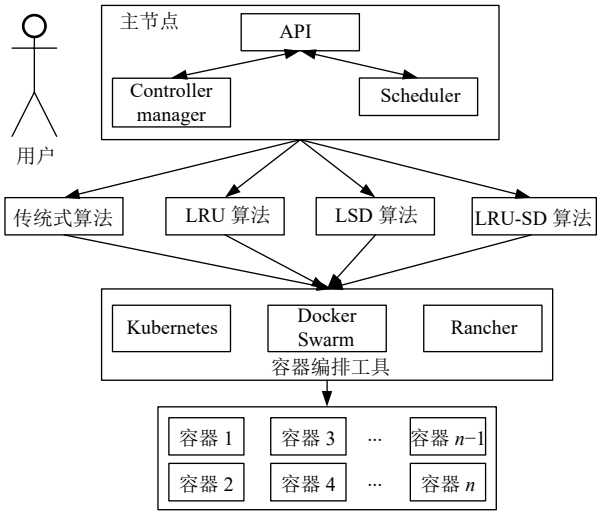


图 1 编排系统架构图

下面分别是 3 种编排工具的详细部署步骤.

1) Kubernetes. 首先, 安装软件包. 在主节点和从节点上使用 yum install <packages>命令安装 etcd、Kubernetes、flannel 软件包. 其次, 修改相应的配置文件. 然后, 构建集群. 成功部署集群后输入 kubectl get nodes 命令可查看集群中节点 IP 及节点状态. 第四, 通过启动 pod 部署可视化界面. 先创建 yaml 文件, 进行服务名称 name、镜像 image 和 IP 地址等的设置, 之后使用 Docker pull 命令拉取所需镜像, 并且启动 yaml 文件. 最后, 启动可视化界面. 通过 kubectl get pod -o wide --all-namespaces 命令可查看 pod 启动状态, 启动状态为 running 时, 可通过浏览器访问可视化界面.

2) Docker Swarm. 首先, 在 Manager 节点创建 Swarm 集群[25]. 其次, 在 Worker 节点上输入加入集群的命令, 将其加入 Swarm 集群. 然后, 输入 Docker node ls 可查看集群中节点名称及节点状态, 集群搭建成功. 最后, 部署 Portainer 可视化界面, 拉取 Portainer 镜像后启动一个容器, 启动成功后可通过浏览器访问 Portainer 可视化界面.

3) Rancher. 首先, 拉取 Rancher 镜像. 然后, 成功启动一个容器. 最后, 通过浏览器访问可视化界面, 可将浏览器内自动生成的命令复制到虚拟机上运行, 从而在 Rancher 工具中成功添加主机.

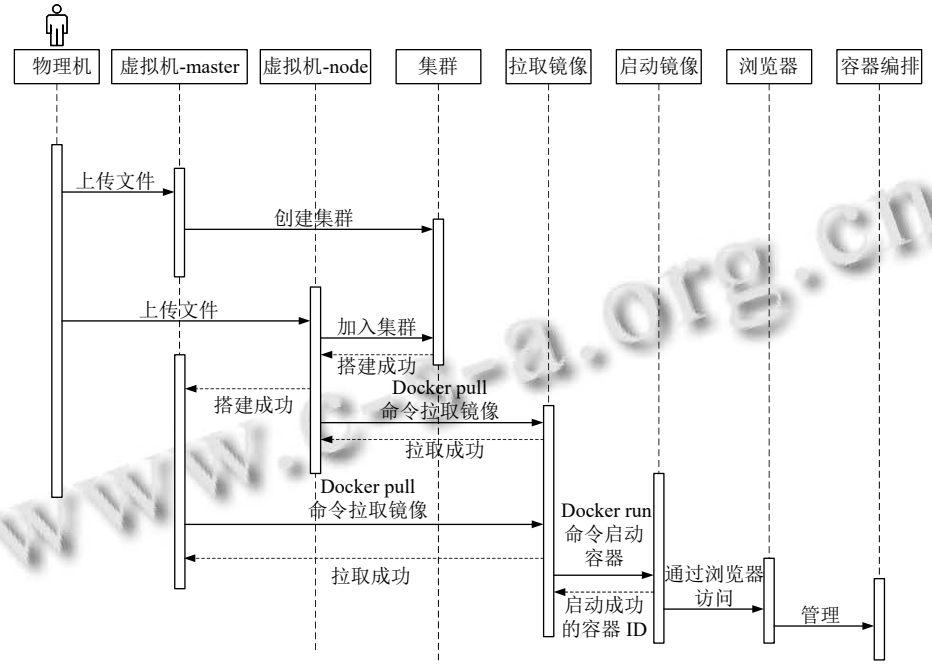


图 2 容器编排工具部署序列图

2.3 符号及公式定义

本文涉及的符号及定义如表 1.

定义 1. 设插入工作节点的个数为ϑ, 设删除工作节点的个数为Ω, δ_{all}为当前链表中工作节点总个数, τ是一个标记数, 默认为 1.

根据定义 1 可以计算链表中工作节点总个数, 如式 (1):

$$\delta_{\text{all}} = \sum_{1}^{\vartheta} \tau(x) - \sum_{1}^{\Omega} \tau(x) \tag{1}$$

定义 2. 设多个工作节点的总容量为Φ(工作点1), …,

Φ(工作点i), …, Φ(工作点n), 设当前使用量为Φ_{current}(工作点1), …, Φ_{current}(工作点i), …, Φ_{current}(工作点n).

表 1 符号定义表

符号	定义
Ω	链表中删除工作节点的个数
ϑ	链表中插入工作节点的个数
δ_{all}	当前链表中工作节点总个数
Φ	工作节点的总容量
Φ_{current}	工作节点的当前使用空间量
Φ_{re}	工作节点的剩余空间量
γ	剩余空间集中的最小值
τ	标记数, 默认为1
β	选中的要删除的工作节点

根据定义 2, 可以得出Φ_{re}的计算式 (2):

$$\Phi_{\text{re}}(\text{工作点}i)=\Phi(\text{工作点}i)-\Phi_{\text{current}}(\text{工作点}i) \quad (2)$$

定义 3. 设γ为剩余空间集中的最小值.

结合定义 2 的结果, 可以得出最小剩余空间的值, 如式 (3):

$$\begin{aligned}\gamma&=\min(\Phi_{\text{re}}(\text{工作点1}),\cdots,\Phi_{\text{re}}(\text{工作点}n))\\&=\min(\Phi-\Phi_{\text{current}}(\text{工作点1}),\cdots,\Phi-\Phi_{\text{current}}(\text{工作点}n))\end{aligned} \quad (3)$$

2.4 算法设计

当容器满时, 传统算法选择删除最后一个进入容器的节点. LRU 算法选择最近最久未使用的节点进行删除, LSD 算法选择当前剩余容量最小的节点进行删除, LRU-SD 算法先考虑最近最久未使用的节点. 如果存在多个满足要求的节点, 则计算这些满足要求的节点的剩余容量, 选择剩余容量最小的进行删除. 在这个过程发现, 传统算法对删除容器的选择并没有采用较优的策略, LRU 算法根据节点使用的频次进行待删除节点的选择, 将使用频次低的节点删除. 理论上, 被删除的节点在后面再次使用的概率低, 因此会降低换页率. LSD 算法根据节点的剩余容量进行待删除节点的选择, 优先删除剩余容量少的节点. 留下的节点因为剩余容量大, 再次需要被删除的概率变低, 因此降低换页率. LRU 算法和 LSD 算法分别从节点的调用频次和剩余空间进行考虑. 同时, 本文结合 LRU 算法和 LSD 算法, 提出 LRU-SD 算法, 该算法同时考虑调用频次和剩余空间, 有效降低换页率. 经过实验验证, 它优于传统算法、LRU 算法和 LSD 算法.

2.4.1 传统式算法

在 Docker Swarm 和 Rancher 中, 发现工作节点的空间不足, 就需要释放空间以便后续的工作. 默认的做法是释放最后一次插入的工作节点.

图 3 显示传统式算法, 当有足够容量满足需要时, 则可以直接插入新的工作节点, 如情况 1 到情况 2 的过程. 当超出最大容量时, 想要插入新的工作节点, 则先删除最后插入的工作节点, 然后再插入新的工作节点, 如情况 3-1 到情况 4-1 的过程. 具体情况介绍如下.

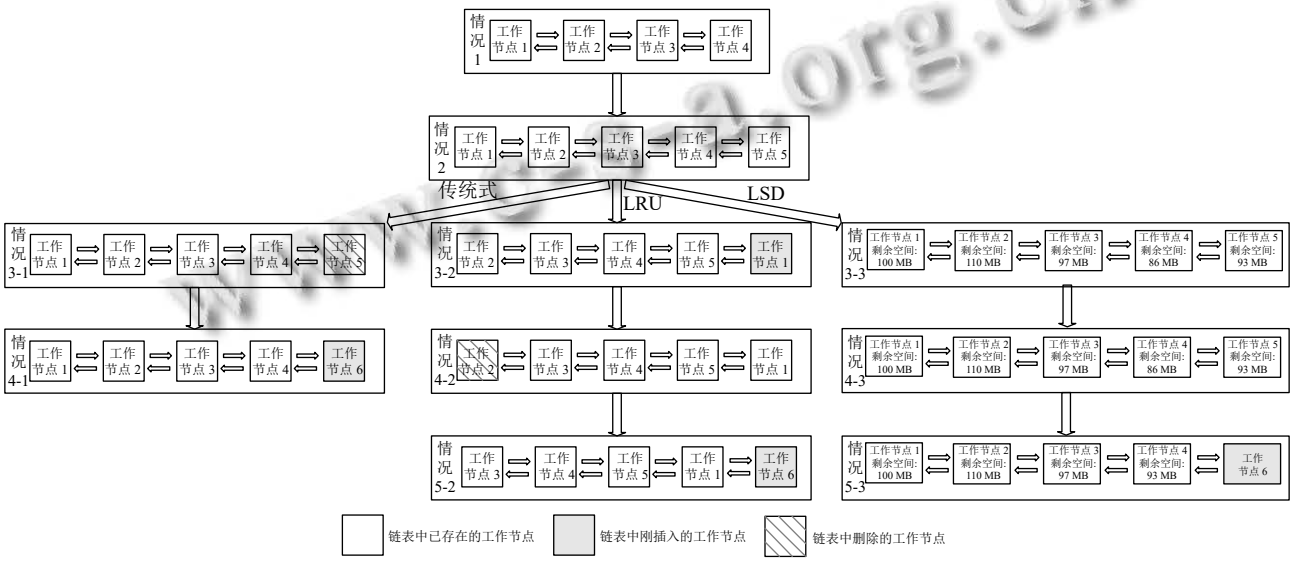


图 3 传统式、LRU、和 LSD 算法过程解释图

情况 1. 链表中存在 4 个正常工作的节点.

情况 2. 将工作节点 5 插入此链表中.

情况 3-1. 将工作节点 5 删除, 其余 4 个工作节点正常工作.

情况 4-1. 工作节点 5 删除后, 再插入工作节点 6.

2.4.2 LRU 算法

当超出最大容量时, Kubernetes 默认使用的是 LRU 算法, 释放最久未使用的工作节点, 来满足新任务的需求. LRU 算法是利用双向链表和哈希表组成的数据结构, 首先利用双向链表进行存储, 其优势是通过 prev 指针和 next 指针可以得出当前元素的前驱元素和后继元素, 方便对元素增加和删除等操作. 其次是利用哈希表进行查找, 其优势是能够快速根据 key 值直接访问 value 的地址. LRU 算法将两者融合后, 可以快速地实现查询、插入或删除的操作, 时间复杂度为 O(1). 图 3 的 LRU 算法, 具体情况介绍如下.

情况 1. 链表中存在 4 个正常工作的节点.

情况 2. 将工作节点 5 插入此链表中.

情况 3-2. 再次调用工作节点 1, 相当于先删除工作节点 1, 然后将工作节点 1 再插入链表中.

情况 4-2. 将最久未使用的工作节点删除, 即删除工作节点 2.

情况 5-2. 插入工作节点 6.

当使用某个工作节点之后, 便会直接被移到链表最后端, 如情况 3-2 所示. 此时, 链表前端便只剩下最久未使用的工作节点. 当容量不足时, 需要删除最前端的工作节点, 释放最久未使用的工作节点, 然后再插入新的工作节点, 如情况 4-2 到情况 5-2 所示.

经过对 LRU 算法过程的归纳, 伪代码如算法 1.

算法 1. LRU 算法

输入: LRU(key, value, hash, list2, Max)
//数据关键字 key; 数据值 value; 数据查询集 map; 数据删除插入集 list2; 最大容量 Max; node 节点
输出: LRU 结构体 Tree

```
1. Procedure list2(key, value)
2. {
3.    node=get(key, value);
4.   //调用 get 函数, 查找数据是否存在, 若存在返回相应的节点地址;
5.    if exist (node != null){
6.       hash.delete(node) //删除 node 节点, 节点个数减 1;
7.       hash.insert(node) //插入 node 节点到尾部, 节点个数加 1;
8.    }else {
9.       node = Node.create(key, value) //定义一个新的 Node;
10.       All.sum //利用式 (1) 计算当前链表中总个数 num;
11.       if (num>Max){
12.          hash.delete //删除头部的工作节点, 节点个数减 1;
13.          hash.insert(node) //插入新的工作节点到尾部, 节点个数加 1;
14.       }
15.       else
16.          hash.insert(node) //直接插入新的工作节点到尾部, 节点个数加 1;
17. }
18. End list2
19. Function put(key, value)
20.    node = hash.get(key)
21.    if (key != null){
22.       Node.value = value;
23.       return node;
24.    }
25.    else
26.       return null;
27. End put
```

2.4.3 LSD 算法

LSD 算法的设计思想为删除剩余空间最少的工作节点, 释放此工作节点, 来满足新任务的空间需求. 此算法使用双向链表方便数据的查询、插入和删除, 并且时间复杂度为 O(1), 容易实现. LSD 算法解释如图 3 所示, 具体情况描述如下.

情况 1. 链表中存在 4 个正常工作的节点, 使用双链表存储这 4 个工作节点.

情况 2. 将工作节点 5 插入此链表中, 此时链表未满, 插入成功. 将工作节点 6 插入链表中, 此时链表已满, 插入失败, 采用 LSD 算法进行节点删除后, 再进行插入.

情况 3-3. 计算链表中 5 个工作节点的剩余空间, 5 个工作节点的剩余空间分别是 100 MB、110 MB、97 MB、86 MB、93 MB.

情况 4-3. 删除 5 个工作节点中剩余空间最小的节点, 即工作节点 4.

情况 5-3. 将新的工作节点 6 插入链表中, 此时插入成功.

当超出最大容量时, 需要先计算各个工作节点的剩余空间, 然后选出所有工作节点中剩余空间最少的工作节点, 并且删除此工作节点释放空间, 最后再插入新的工作节点即可, 如情况 3-3 到情况 5-3 的过程.

根据对 LSD 算法过程的归纳总结, 伪代码如算法 2.

算法 2. LSD 算法

输入: LSD(key, value, hash, list2, Max)
//数据关键字 key; 数据值 value; 数据查询集 map; 数据删除插入集 list2; 最大容量 Max; node 节点
输出: LSD 结构体 Tree

```
1. Procedure list3(key, value)
2. {
3.    node=get(key, value);
4.   //调用 get 函数, 查找数据是否存在, 若存在返回相应的节点地址;
5.    if exist (node != null){
6.       hash.delete(node) //删除 node 节点, 节点个数减 1;
7.       hash.insert(node) //插入 node 节点到尾部, 节点个数加 1;
8.    }else {
9.       node = Node.create(key, value) //定义一个新的 Node;
10.      All.sum //利用式 (1) 计算当前链表中总个数 num;
11.      if(num>Max){
12.         nodeF=re(hash);hash.delete(nodeF) //删除 nodeF 节点, 节点个数减 1;
13.         hash.insert(nodeF) //在删除的位置上插入新节点, +1, 节点个数加 1;
14.   }
15.   else
16.         hash.insert(node) //直接插入新的工作节点到尾部, 节点个数加 1;
17.   }
18. End list3
19. Function put(key, value)
20.   node = hash.get(key)
21.   if (key != null){
22.      Node.value = value;
23.      return node;
24.   }
25.   else
26.      return null;
27. End put
28. Function re(key)
29.   All.remainder //利用式 (2) 求出节点对应的剩余空间;
30.   All.least //利用式 (3) 选出最少剩余空间的节点 key;
31.   return least
32. End re
```

2.4.4 LRU-SD 算法

LRU-SD 算法的设计思想同时考虑了节点的剩余空间和最近使用情况. 当没用空闲工作节点时, 首先选择最久未使用的节点释放, 当出现多个最久未使用的节点时, 考虑这些节点当前剩余的工作空间, 释放剩余工作空间最小的节点, 如式 (4) 所示:

$$\beta = \begin{cases} \alpha,\ num(\alpha_i) = 1 \\ \alpha_i,\ \text{if } (\varphi_{re}(\alpha_i) = \min(\varphi_{re}(\alpha_1), \varphi_{re}(\alpha_2), \varphi_{re}(\alpha_3)\cdots)) \end{cases} \tag{4}$$

其中, α_i 表示最近最久未使用的工作节点, $num(\alpha)$ 表示最近最久未使用的工作节点的数量. β 表示要删除的节点. 此算法使用双向链表方便数据的查询、插入和删除, 并且时间复杂度为 O(1), 容易实现. LRU-SD 算法解释如图 4 所示, 具体情况描述如下.

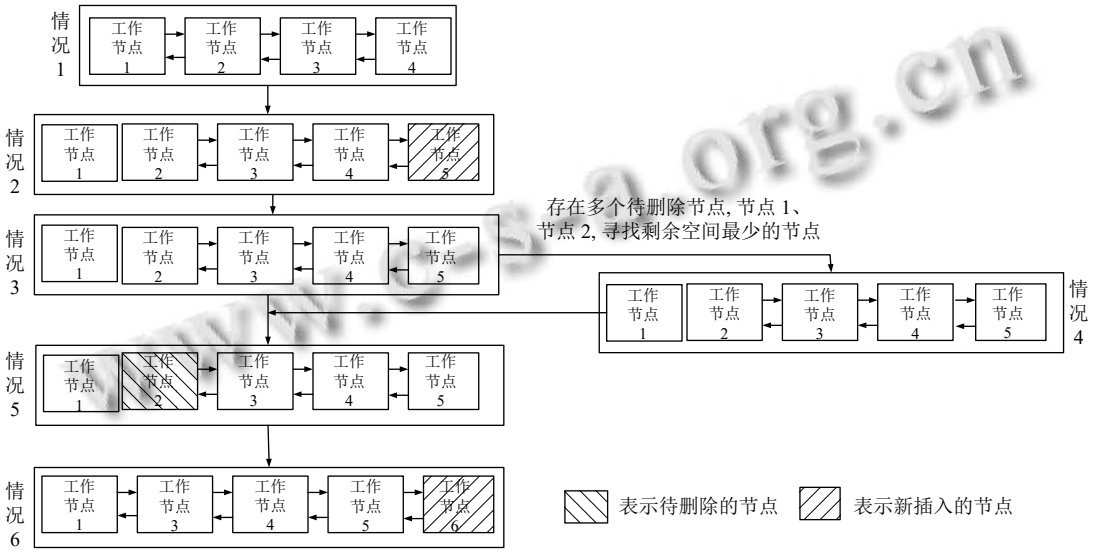


图 4 LRU-SD 算法解释图

情况 1. 链表中存在 4 个正常工作的节点, 使用双链表存储这 4 个工作节点.

情况 2. 将工作节点 5 插入此链表中. 此时, 链表处于未满状态, 可以直接插入.

情况 3. 此时要插入工作节点 6, 求出当前最久未使用的工作节点, 若存在多个工作节点, 进入情况 4; 若只存在一个当前最久未使用的工作节点, 选中该节点, 进入情况 5.

情况 4. 计算所有当前最久未使用的工作节点的剩余空间, 并选中所有当前最久未使用的工作节点中剩余空间最小的节点. 在图 4 中, 存在多个最近最久未使用的节点, 即节点 1 和节点 2. 然后分别计算节点 1 和节点 2 的剩余空间, 节点 2 的剩余空间小于节点 1, 则删除节点 2.

情况 5. 删除选中的节点.

情况 6. 将新的工作节点 6 插入链表中.

当超出最大容量并存在多个最久未使用的工作节点时, 计算所有当前最久未使用的工作节点的剩余空间, 并选中所有当前最久未使用的工作节点中剩余空间最小的节点, 删除该节点.

当在同一时间有多个节点进入, 并在之后的时间里没有被使用时, 这多个节点会同时成为最久未使用的节点. 在上面 3 个算法中, 由于存储节点的数据结构的局限性, 无法找出这多个节点, 所以本文重新定义一个数据结构来存储同时进入链表的节点, 即节点集 nodes, 在节点集中可以存储多个节点, 在同一个节点集中的节点就是同时进入的节点. 其他操作同上述算法一样. LRU-SD 算法的流程图如图 5 所示.

根据对 LRU-SD 算法过程的归纳总结, 伪代码如算法 3.

算法 3. LRU-SD 算法

输入: LRU-SD(key, value, hash, list4, Max)
//数据关键字 key; 数据值 value; 数据查询集 map; 数据删除插入集 list4; 最大容量 Max; node 节点
输出: LRU-SD 结构体 Tree

```
1. Procedure list4(key, value)
2. {
3.   node=get(key, value);
4.   //调用 get 函数, 查找数据是否存在, 若存在返回相应的节点地址;
5.   if exist (node != null){
6.       hash.delete(node) //删除 node 节点, 节点个数减 1;
7.       hash.insert(node) //插入 node 节点到尾部, 节点个数加 1;
8.   }else {
9.     node = Node.create(key, value) //定义一个新的 Node;
10.    All.sum //利用式 (1) 计算当前链表中总个数 num;
11.    if (num>Max){
12.      if (nodes.num>1){//当前最久未使用的节点有多个
13.        nodeF=re(hash);//找当前最久未使用的节点中剩余空间最少的节点
14.        node.delete(nodeF) //删除 keyF 节点, −1;
15.        node.insert(ndoe) //在删除的位置上插入新节点, +1;}
16.      else{//只有一个最久未使用节点, 直接删除该节点
17.        hash.delete //删除头部的工作节点, −1;
18.        hash.insert(node) //插入新的工作节点到尾部, +1;} }
19.    else
20.      hash.insert(node) //直接插入新的工作节点到尾部, 节点个数加 1;
21. }
22. End list4
23. Function put(key, value)
24.   node = hash.get(key)
25.   if (key != null){
26.      Node.value = value;
27.      return node;
28.   }
29.   else
30.      return null;
31. End put
32. Function re(key)
33.   All.remainder //利用式 (2) 求出节点对应的剩余空间;
34.   All.least //利用式 (3) 选出最少剩余空间的节点 key;
35.   return key;
36. End re
```

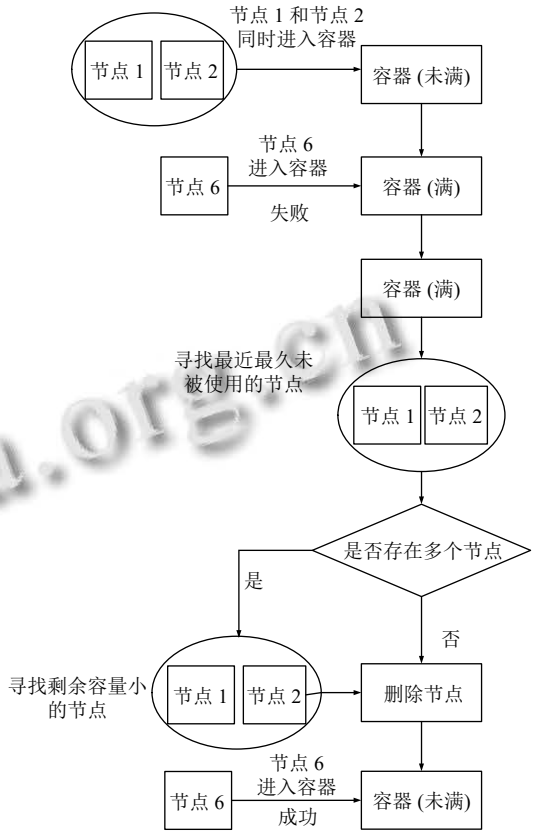


图 5 LRU-SD 算法流程图

3 实验与分析

本节使用传统式算法、LSD 算法、LRU 算法和 LRU-SD 算法运行程序后, 然后对 Kubernetes、Docker

Swarm 和 Rancher 产生的影响做实验与分析. 实验包括对响应时间、CPU 使用率和内存占用率的测试, 以此分析和比较传统式、LSD 算法、LRU 算法和 LRU-SD 算法的区别. 表 2 为相关的实验环境及软件版本.

表 2 实验环境及软件版本说明

名称	版本
虚拟机	VMware Workstation 12 pro
操作系统	CentOS Linux release 7.9.2009 (Core)
Docker	18.06.2.ce
Kubernetes	V1.5.2
Rancher	V1.6.30

3.1 测试实验

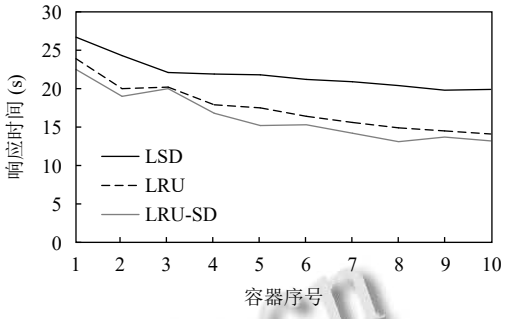
测试实验分成 3 个部分, 第 1 部分是 3 个工具在传统式、LSD 算法、LRU 算法和 LRU-SD 算法启动容器的时间响应测试. 依照 cAdvisor[26] 有收集、处理和导出资源使用和网络统计等关于容器信息的功能[27]. 第 2 部分和第 3 部分是利用 cAdvisor 对 Kubernetes、Docker Swarm 和 Rancher 等 3 种工具进行监控, 获取 CPU 和内存的占用情况. 通过 CPU 和内存的测试, 对比这 3 种编排工具的资源占用情况, 针对使用传统式、LSD 算法、LRU 算法和 LRU-SD 算法对编排工具资源使用率的影响.

3.1.1 响应时间测试

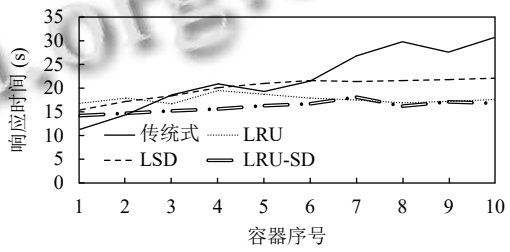
分别使用 3 个工具来连续启动 10 个容器, 比较这 3 个工具在传统式、LSD 算法、LRU 算法和 LRU-SD 算法下启动容器所花费的时间. 实验中启动容器的同时, 勾选启动前自动拉取镜像的选项. 成功启动容器后, 在 3 个虚拟机终端输入查看容器详细信息的命令, 可以查看对应容器的详细信息, 其中包括创建完成时间和开始运行时间. 查看 Docker Swarm 和 Rancher 启动的容器信息需要输入 Docker inspect <容器 ID>命令, 而查看 Kubernetes 启动的容器信息则要输入 kubectl describe pods <容器 NAME>. 由于响应时间即为开始运行时间和创建完成时间的差值, 能够轻易地比较 3 个工具在不同算法启动容器的速度快慢. 对应的测试结果如图 6 所示.

依据测试实验, 得出各个工具在启动相同数量容器时的响应时间. 结果表明, 使用传统式算法时, Docker Swarm 和 Rancher 的图线呈上升趋势. 使用 LSD 算法和 LRU-SD 算法时, Kubernetes 的图线呈下降趋势, Docker Swarm 和 Rancher 的图线较平稳. 使用 LRU 算法时, Kubernetes 的图线比使用 LSD 算法下降幅度更大, Docker Swarm 和 Rancher 的图线整体呈平稳状态, 并且比使用 LSD 算法时的整体响应时间更短. 当启动的容器数目较少时, Kubernetes 的响应时间比其他两种工具的响应时间更长. 使用传统式算法时, 随着启动容器的数目越来越多, Docker Swarm 和 Rancher 的响应速度却越来越慢. 而使用 LSD 算法和 LRU 算法时, 当启动容器的数目增多, Kubernetes 的响应速度越来越快, Docker Swarm 和 Rancher 的响应速度基本保持不变. 因此, Kubernetes 更适宜于在容器或主机多的情况下使用, 而 Docker Swarm 和 Rancher 则在容器或主机较少的情况下表现更出色. 通过式 (5) 可以计算出 3 种工具分别在使用不同算法时的平均响应时间.

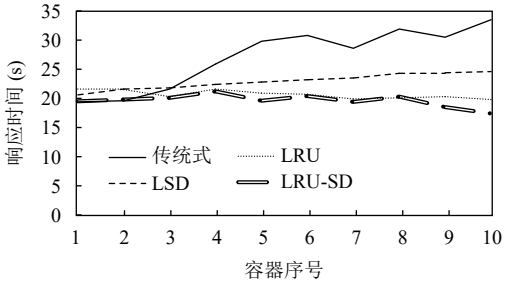
$$\bar{t} = \frac{\sum_{i=1}^{10} t_i}{10} \tag{5}$$



(a) Kubernetes 的响应时间记录



(b) Docker Swarm 的响应时间记录图



(c) Rancher 的响应时间记录图

图 6 启动容器的响应时间

式 (5) 计算 Kubernetes 使用 LRU-SD 算法、LSD 算法和 LRU 算法的$\bar{t}$分别为 15.6 s、21.9 s 和 17.5 s. Docker Swarm 使用传统式和 LSD 算法的$\bar{t}$均为 21 s 左右, LRU 算法的$\bar{t}$为 17.8 s, LRU-SD 算法的最低. Rancher 与 Docker Swarm 使用 LRU-SD 算法、LSD 算法和 LRU 算法的差距较小, 呈现平稳的特性, 但是传统式的$\bar{t}$为测试结果中最大的值, 到达 27.1 s. 由此发现, Docker Swarm 和 Rancher 使用传统式算法的$\bar{t}$最长且逐渐上升, 使用 LRU 算法的$\bar{t}$最短且最平稳. Kubernetes 使用 LRU-SD 算法比使用 LSD 算法时的$\bar{t}$短. 无论是使用 LSD 算法还是 LRU 算法, Kubernetes的响应速度最快, Rancher 的响应速度最慢.

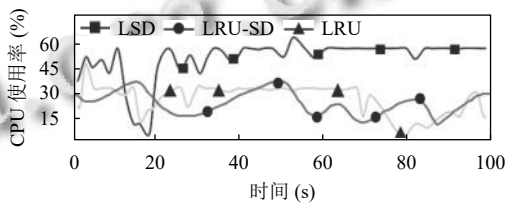
3.1.2 CPU 测试

图 7 展示 3 种工具在不同算法时的 CPU 测试结果. 结果显示, 使用传统式算法时, 图 7(b) 的 Docker Swarm 在 CPU 的使用率为 15% 左右时会趋于平稳; 图 7(c) 的 Rancher 在 CPU 的使用率为图 7(b) 的 Docker Swarm 的 2 倍左右趋于平稳. 使用 LSD 算法时, 图 7(a) 的 Kubernetes 有较大的波动, 后在 45% 左右趋于平稳, 总体占用较高; 图 7(b) 的 Docker Swarm在 CPU 使用率波动介于 5%–15% 之间, 总体占用较少. Rancher 的 CPU 使用率 12%–36% 范围内波动. 使用 LRU 算法和 LRU-SD 算法时, Kubernetes 整体在 20%–30% 之间浮动, 基本平稳. Docker Swarm 开始在 5%–15% 范围内波动, 后面在 7% 左右趋于平稳. Rancher 整体稳定在 12%.

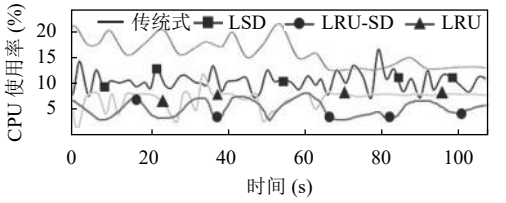
根据式 (6) 求出 Kubernetes、Docker Swarm 和 Rancher 这 3 个工具, 使用不同算法时的 CPU 平均使用率.

$$\bar{n}=\frac{1}{5}\left(\sum_{T=0}^{4}\frac{\sum_{t=20T+1}^{20T+20}n_t}{20}\right) \tag{6}$$

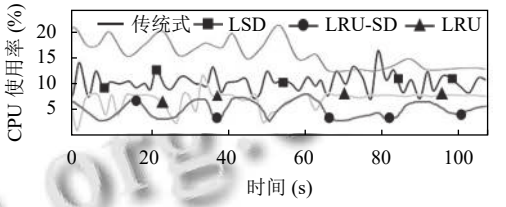
根据式 (6) 求出 Kubernetes 在使用 LRU-SD 算法、LSD 算法和 LRU 算法的 CPU 平均使用率分别为 23.2%、44.8% 和 25.2%, 使用 LSD 算法比使用 LRU 算法多 1 倍; Docker Swarm 在使用传统式、LSD 算法、LRU-SD 算法和 LRU 算法的 CPU 平均使用率分别是 15.1%、11.6%、9.1% 和 8.2%, CPU 平均使用率逐渐降低, 总体差距不大; Rancher 在使用传统式、LSD 算法、LRU-SD 算法和 LRU 算法的 CPU 平均使用率分别是 39.7%、25.6%、15.3% 和 14.1%, CPU 平均使用率逐渐降低, 传统式算法和 LRU-SD 算法差距最大. 根据 CPU 测试以及比对均值使用率可知, 无论是使用哪种算法, Kubernetes 的 CPU 平均使用率最大, Docker Swarm 的 CPU 平均使用率最小. 并且 Kubernetes 使用 LRU 算法比使用 LSD 算法的 CPU 使用率更低. Docker Swarm 和 Rancher 使用 LSD 算法和 LRU 算法的 CPU 使用率都比传统式低, 其中使用 LRU 算法的 CPU 使用率最低.



(a) Kubernetes 的 CPU 使用测试



(b) Docker Swarm 的 CPU 使用测试



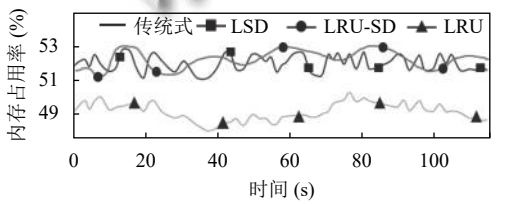
(c) Rancher 的 CPU 使用测试

图 7 CPU 使用率图

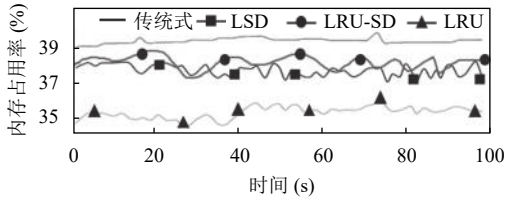
3.1.3 内存测试

图 8 展示 3 种工具在使用不同算法时的内存占用测试. 结果显示, 使用传统式算法时, 图 8(b) Docker Swarm 的内存占用在 39% 处基本稳定, 小范围内波动; 图 8(c) Rancher 的内存占用在 45% 处上下波动, 与图 8(b) 的 Docker Swarm 差距不大. 使用 LSD 算法时, 图 8(a) Kubernetes 和图 8(b) Docker Swarm 的内存占用存在上下浮动; 图 8(c) 的 Rancher 的内存变化平稳. 使用 LRU 算法时, 图 8(a) Kubernetes 的内存变化介于 48% 和 51%; 图 8(b) 的 Docker Swarm 在 36% 左右, 小范围内波动; 图 8(c) Rancher 在 42% 左右, 小范围内波动.

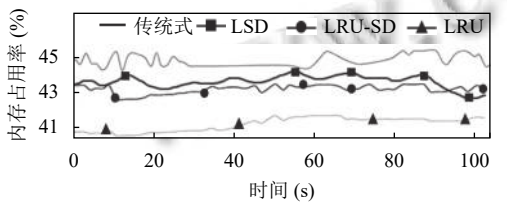
此实验设定的内存为 1 024 MB. 根据式 (5) 求出 Kubernetes、Docker Swarm 和 Rancher 这 3 个工具在使用不同算法时的内存平均占用率. Kubernetes 在使用 LSD 算法、LRU-SD 算法和 LRU 算法时的内存平均占用率都在 50% 左右; Docker Swarm 在使用传统式、LSD 算法、LRU-SD 算法和 LRU 算法时的内存平均占用率介于 35% 和 39%, Rancher 在使用传统式、LSD 算法、LRU-SD 算法和 LRU 算法均比 Docker Swarm 大 6% 左右. 根据内存测试以及比对内存均值占用率可知, 无论是使用哪种算法, Kubernetes 的内存平均占用率最大, Docker Swarm 的内存平均占用率最小. 并且 Kubernetes 使用 LRU 算法比使用 LSD 算法的内存平均占用率更低. Docker Swarm 和 Rancher 使用 LSD 算法和 LRU 算法的内存平均占用率都比传统式低, 其中使用 LRU 算法的内存平均占用率最低.



(a) Kubernetes 的内存占用测试图



(b) Docker Swarm 的内存占用测试图



(c) Rancher 的内存占用测试图

图 8 内存占用率图

3.2 实验分析与总结

3.2.1 内存测试

针对响应时间的测试实验做出分析, 由图 9 可以明显看出, Rancher 的响应速度最慢, 但是经过对算法的改进, 有效缩短 Rancher 与其他两种工具在响应速度上的差距. 使用 LSD 算法、LRU-SD 算法和 LRU 算法的响应速度都比使用传统式算法快, 其中以 LRU-SD 算法的速度最快. Kubernetes 使用 LRU-SD 算法比 LSD 算法的$\bar{t}$快 4.4 s; Docker Swarm 使用传统式、LSD 算法、LRU 算法的$\bar{t}$逐渐递减; Rancher 使用传统式时间最长, 相比之下 LRU 算法和 LSD 算法的$\bar{t}$更快, 同时, LRU 算法比 LSD 算法还要快 1.6 s. 综上说明使用 LSD 算法、LRU-SD 算法和 LRU 算法可以加快编排工具的响应速度, 进而可以提升工作效率, 其中 LRU-SD 算法的效果更好、提升的效率更高, 尤其是对 Rancher 的促进效果最明显.

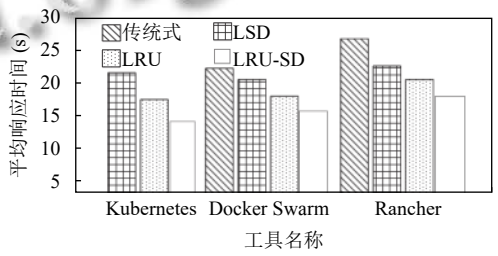


图 9 响应时间对比图

3.2.2 分析 CPU 使用率及内存占用率

根据各个工具在使用不同算法时的 CPU 使用率和内存占用率, 制成表 3, 由此看出 3 种工具中 Kubernetes 的 CPU 使用率和内存占用率都为最高, 说明 Kubernetes 资源占用率比其他两种工具多, 所以 Kubernetes 的成本比其他两种工具的成本高. 而 Docker Swarm 的资源占用最少, 成本最低.

表 3 性能对比表 (%)

工具	算法	CPU平均使用率	内存平均占用率
Kubernetes	LSD	44.8	51.9
	LRU	25.2	49.5
	LRU-SD	23.2	52.1
Docker Swarm	传统式	15.1	39.1
	LSD	11.6	37.8
	LRU	8.2	35.6
	LRU-SD	8.1	38.2
Rancher	传统式	39.7	45.1
	LSD	25.6	44.5
	LRU	14.1	41.9
	LRU-SD	13.8	44.3

表 3 显示, Kubernetes 使用 LRU 算法比 LSD 算法的 CPU 平均使用率大约低 1/5; Docker Swarm 使用传

统式算法、LSD 算法、和 LRU 算法的 CPU 平均使用率, 呈现逐渐递减趋势, 递减 3.5% 左右; Rancher 使用传统式算法、LSD 算法、和 LRU 算法的 CPU 平均使用率, 呈现逐渐递减趋势, 递减 13% 左右. 这说明 LSD 算法和 LRU 算法都可以提升 CPU 的使用率, 以 LRU 算法的提升效果较好, 对 Kubernetes 的 CPU 使用率影响最大, 对 Docker Swarm 的影响最小. 在 3 种编排工具中, LRU-SD 算法的 CPU 平均使用率仅仅略低于 LRU, 同时 LRU-SD 算法需要额外的空间存储节点间, 在内存使用率上比 LRU 和 LSD 算法略高.

从内存平均占用率来看, Kubernetes 使用 LRU 算法比 LSD 算法的内存平均占用率低 2.4%; Docker Swarm 使用传统式算法的内存平均占用率最高, 使用 LRU 算法的内存平均占用率最低, 使用 LRU 算法比 LSD 算法低 2.2%; Rancher 使用 LSD 算法和传统式算法的内存平均占用率相差不大, 使用 LRU 算法的内存平均占用率最低, 约为 3%. 说明 LSD 算法和 LRU 算法都可以提升内存使用率, 其中 LRU 算法提升效果更好, 但是对 3 种工具的影响效果相差不大.

经过上述实验发现, LSD 和 LRU 算法无论在速度上还是 CPU 和内存使用上均优于传统算法, 并且 LRU 算法作为一种经典算法, 在应用于容器时依然具有极大的优势. 但是, 本文考虑到当同一时刻有多个最近最久未使用的节点时, LRU 算法仅是随机删除一个节点, 但是这些节点之间的剩余容量并不相同, 总希望删除剩余容量最小的节点, 因为这样可以延缓节点满时需要添加新的节点的情况. 因此, 本文将新提出的 LSD 方法与 LRU 方法相结合, 当出现多个最近最久未使用的节点时, 删除剩余空间最少的节点, 以此提高容器的使用效率, 并且混合后的方法与 LSD 方法相比, 仅需要计算最近最久未使用的节点的剩余空间即可, 不需要计算全部节点的剩余空间, 减少了计算量.

本次实验的前提是同时有多个节点加入容器, 这样会产生多个最近最久未使用的节点, 在该前提下, 探究 LRU、LSD、和 LRU-SD 等 3 种方法的内存使用率、CPU 使用率和换页率. 与 LRU 方法和 LSD 方法相比, LRU-SD 方法需要记录同时进入容器的节点, 在本次实验中使用链表加标志的方式进行记录, 这就需要新的存储空间, 所以在内存使用率上 LRU-SD 方法明显高于 LRU 和 LSD 方法. 但是通过增加内存记忆同时进入容器的节点, 再从中选出剩余空间最小的节点进行删除, 可以降低换页率, 提高容器的使用效率.

综上所述, 使用 LSD 算法和使用 LRU 算法的编排工具比使用传统式算法的编排工具所占资源少, 其中使用 LRU 算法的编排工具降低的资源占用最大. 说明 LSD 算法和 LRU 算法都可以提高资源使用率, 而且 LRU 算法提升资源使用率效果最好, 尤其是对提高 CPU 的使用率最有效. LRU-SD 算法在内存占用率上比其他算法略高, 并且可以有效降低 CPU 的使用率.

3.2.3 功能分析

从工具的部署特性, Kubernetes 是应用的部署, Docker Swarm 是容器的部署, 而 Rancher 属于容器化组件. 在安装复杂度方面, Kubernetes 安装过程复杂, Docker Swarm 和 Rancher 的部署步骤简单. 从功能方面上看, Kubernetes、Docker Swarm 和 Rancher 有很多相同点和不同点. 首先 3 个工具都有自动化部署、支持负载均衡和监测容器健康状况的功能, 这些功能都为编排工作提供便利. 此外, 这 3 种工具也有很多的不同点, 表 4 显示它们的功能归纳与对比.

表 4 3 种编排工具功能对比

功能	Kubernetes	Docker Swarm	Rancher
灰度发布	支持并最适合	支持	不支持
弹性伸缩	手动和自动	手动	手动
回滚到特定版本	支持	不支持	支持
可视化界面	内置	手动配置	内置
应用模板	不提供	不提供	提供

功能对比显示, Rancher 不支持灰度发布, Kubernetes 和 Docker Swarm 支持灰度发布, 其中 Kubernetes 对灰度发布的支持度最高, 可以尽量避免新版本带来的风险. 然后 Kubernetes 和 Rancher 都可以回滚到特定的版本, 而 Docker Swarm 不支持此功能, 只能回滚到上一次的状态. 在弹性伸缩的功能中, 3 个都支持手动伸缩, 但是只有 Kubernetes 还支持自动伸缩. 特别的是, Docker Swarm 没有内置的可视化界面, 需要进行单独部署. Rancher 提供应用模板, 简化复杂应用. 整体上, Kubernetes 在功能性上较其他两种更有优势, 集群管理能力更强[28]. 并且在文献 [29] 中, 通过在 Kubernetes 和 Docker Swarm 上运行服务来对比, 也能够证明 Kubernetes 功能性确实更强.

3.2.4 实验总结

经过对使用不同算法的编排工具的比较, 验证 LSD 算法、LRU-SD 算法和 LRU 算法不仅能提高编

排工具在启动容器时的响应速度, 还可以提升编排工具的资源使用率, 主要包括 CPU 使用率和内存占用率. 其中 LRU-SD 算法对提高工作效率、提升资源使用率的效果最好. 并且 3 种算法的时间复杂度都为 O(1), 因此 3 种算法容易实现.

通过对伸缩、回滚功能、反应速度、CPU 和内存使用率的性能比较, 发现 Kubernetes 在灰度发布和弹性伸缩的功能性更强, 并且在开始时响应速率较慢, 而启动大规模容器时速率明显加快, 但是它占用的 CPU 和内存都相对较多. Docker Swarm 和 Rancher 比 Kubernetes 部署使用更简单[30], 且占用的 CPU 和内存较少, 其中 Docker Swarm 占用最少. 但是 Docker Swarm 和 Rancher 只在小规模容器时速率快、凸显优势. 3 个工具中只有 Rancher 可以直接创建复杂的应用, 节省时间. 综上所述, Kubernetes 擅长管理大规模和高可用的容器及集群, 虽然部署过程的复杂且耗时, 但是它的使用更广泛, 并且功能性更强[31]. Docker Swarm 和 Rancher 适合管理小规模集群, 安装简单, 启动速度快, 其中 Rancher 还更适合在执行复杂的服务时使用. 3 种工具各有优势, 用户应根据不同需要合理选择适宜的工具.

在实验的最后将 LRU 算法与 LSD 算法相结合, 借助经典算法的普适高效率以及 LSD 算法在特殊情况下具有的优势, 进一步完善 LRU 算法, 降低容器的换页率.

4 结语

容器编排工具和可视化界面极大地提高服务生成效率. 容器编排工具为容器的自动化和规模化提供保障. 本文将 LRU 算法引用到 Docker Swarm 和 Rancher 中, 提出 LSD 算法和 LRU-SD 算法并用于 3 种编排工具中, 来释放不工作的节点. 然后验证这种做法在有限空间中可以提高编排工具响应速度和资源使用率, 提升工作效率.

基于容器和编排的快速发展, 安全性问题俨然成为人们关心的焦点. 未来, 可以继续对提高编排工具自身安全性的研究. 此外, 延续微服务架构的模式, 更多地将容器化应用到大型自动化的制造产业, 发展快速更新软件的过程.

参考文献

1 Uhlig R, Neiger G, Rodgers D, *et al*. Intel virtualization technology. Computer, 2005, 38(5): 48–56. [doi: 10.1109/MC.2005.163]

2 彭博, 杨鹏, 马志程, 等. 基于 Docker 的 ARM 嵌入式平台性能评测与分析. 计算机应用, 2017, 37(S1): 325–330.

3 徐蕴琪, 黄荷, 金钟. 容器技术在科学计算中的应用研究. 计算机科学, 2021, 48(1): 319–325. [doi: 10.11896/jsjkx.191100111]

4 应毅, 刘亚军, 俞琰. 利用 Docker 容器技术构建大数据实验室. 实验室研究与探索, 2018, 37(2): 264–268. [doi: 10.3969/j.issn.1006-7167.2018.02.064]

5 Merkel D. Docker: Lightweight Linux containers for consistent development and deployment. Linux Journal, 2014, 2014(239): 2.

6 Kang DK, Choi GB, Kim SH, *et al*. Workload-aware resource management for energy efficient heterogeneous Docker containers. Proceedings of the 2016 IEEE Region 10 Conference. Singapore: IEEE, 2016. 2428–2431.

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

8 Costa J, Matos R, Araujo J, *et al*. Software aging effects on Kubernetes in container orchestration systems for digital twin cloud infrastructures of urban air mobility. Drones, 2023, 7(1): 35. [doi: 10.3390/drones7010035]

9 Piedade B, Dias JP, Correia FF. Visual notations in container orchestrations: An empirical study with Docker compose. Software and Systems Modeling, 2022, 21(5): 1983–2005. [doi: 10.1007/s10270-022-01027-8]

10 Heidari SM, Paznikov AA. Multipurpose cloud-based compiler based on microservice architecture and container orchestration. Symmetry, 2022, 14(9): 1818. [doi: 10.3390/sym14091818]

11 Rovnyagin MM, Shipugin VA, Ovchinnikov KA, *et al*. Intelligent container orchestration techniques for batch and micro-batch processing and data transfer. Procedia Computer Science, 2021, 190: 684–689. [doi: 10.1016/j.procs.2021.06.079]

12 邓伟健, 陈曦. 基于时变资源的容器化虚拟网络映射算法. 计算机应用, 2022, 42(2): 550–556.

13 边曼琳, 王利明. 云环境下 Docker 容器隔离脆弱性分析与研究. 信息网络安全, 2020, 20(7): 85–95. [doi: 10.3969/j.issn.1671-1122.2020.07.010]

14 王宝生, 张维琦, 邓文平. 面向大规模容器集群的网络控制技术. 国防科技大学学报, 2019, 41(1): 142–151. [doi: 10.11887/j.cn.201901020]

15 张丽敏, 高晶, 李务斌, 等. 微服务环境下容器编排可视化实践研究. 计算机工程与科学, 2019, 41(8): 1366–1373.

[doi: 10.3969/j.issn.1007-130X.2019.08.005]
16 李华东, 张学亮, 王晓磊, 等. Kubernetes 集群中多节点合作博弈负载均衡策略. 西安电子科技大学学报, 2021, 48(6): 16–22, 122. [doi: 10.19665/j.issn1001-2400.2021.06.003]
17 韩宁. 云计算虚拟化技术的发展与趋势. 电子技术与软件工程, 2018(13): 158.
18 孔德瑾, 姚晓玲. 面向 5G 边缘计算的 Kubernetes 资源调度策略. 计算机工程, 2021, 47(2): 32–38. [doi: 10.19678/j.issn.1000-3428.0058047]
19 谢晓兰, 张征征, 王建伟, 等. 基于三次指数平滑法和时间卷积网络的云资源预测模型. 通信学报, 2019, 40(8): 143–150. [doi: 10.11959/j.issn.1000-436x.2019172]
20 杨凯琪, 姚培, 赵玉龙, 等. 面向异构容器云的应用迁移方法. 计算机工程, 2019, 45(8): 42–47. [doi: 10.19678/j.issn.1000-3428.0051519]
21 Shamim SI, Bhuiyan FA, Rahman A. XI commandments of Kubernetes security: A systematization of knowledge related to Kubernetes security practices. Proceedings of the 2020 IEEE Secure Development. Atlanta: IEEE, 2020. 58–64.
22 Rattihalli G, Govindaraju M, Lu H, *et al*. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes. Proceedings of the 12th IEEE International Conference on Cloud Computing. Milan: IEEE, 2019. 33–40.
23 Bella MRM, Data M, Yahya W. Web server load balancing based on memory utilization using Docker Swarm. Proceedings of the 2018 International Conference on Sustainable Information Engineering and Technology. Malang: IEEE, 2018. 220–223.
24 林昊. 基于改进混合蛙跳算法的 Swarm 集群调度策略研究 [硕士学位论文]. 广州: 广东工业大学, 2020. 8.
25 王昭, 邓浩江, 胡琳琳, 等. 一种多服务端 HTML5 Web Worker 迁移系统设计与实现. 计算机应用与软件, 2018, 35(9): 27–31, 43. [doi: 10.3969/j.issn.1000-386x.2018.09.005]
26 卜尧, 吴斌, 陈玉峰, 等. BDAP——一个基于 Spark 的数据挖掘工具平台. 中国科学技术大学学报, 2017, 47(4): 358–368. [doi: 10.3969/j.issn.0253-2778.2017.04.011]
27 Cinque M, Corte RD, Pecchia A. Microservices monitoring with event logs and black box execution tracing. IEEE Transactions on Services Computing, 2022, 15(1): 294–307. [doi: 10.1109/TSC.2019.2940009]
28 许源佳, 吴恒, 杨晨, 等. 面向状态可变数据流的集群调度综述. 计算机学报, 2022, 45(5): 973–992.
29 Marathe N, Gandhi A, Shah JM. Docker Swarm and Kubernetes in cloud computing environment. Proceedings of the 3rd International Conference on Trends in Electronics and Informatics. Tirunelveli: IEEE, 2019. 179–184.
30 游贵荣, 陈杰, 乐宁莉. 轻量级网络安全攻防 PaaS 平台研究与实现. 实验室研究与探索, 2020, 39(12): 129–133, 162. [doi: 10.3969/j.issn.1006-7167.2020.12.026]
31 Kang B, Jeong J, Choo H. Docker Swarm and Kubernetes containers for smart home gateway. IT Professional, 2021, 23(4): 75–80. [doi: 10.1109/MITP.2020.3034116]

(校对责编: 牛欣悦)