计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(9):152−160 [doi: 10.15888/j.cnki.csa.008062]

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

基于遗传算法的 Kubernetes 资源调度算法①

胡程鹏, 薛　涛

(西安工程大学 计算机科学学院, 西安 710048)
通讯作者: 薛　涛, E-mail: 2843053453@qq.com

摘　要: Kubernetes 在优选阶段仅根据节点 CPU 和内存的利用率来决定节点的分值, 这只能保证单节点的资源利用率, 无法保证集群资源的负载均衡. 针对该问题, 提出一种基于遗传算法的 Kubernetes 资源调度算法, 该算法加入了网络带宽和磁盘 IO 两项评价指标, 同时为评价指标赋予不同权重值, 并且引入校验字典校验并修复遗传算法生成的新种群中不符合配置的个体. 实验结果表明, 与 Kubernetes 默认资源调度策略相比, 该算法考虑了集群中的所有节点的资源利用率, 在保证集群负载均衡方面有着更好的效果.

关键词: Kubernetes; 遗传算法; 资源调度; 云平台; 云计算

引用格式: 胡程鹏,薛涛.基于遗传算法的 Kubernetes 资源调度算法.计算机系统应用,2021,30(9):152–160. http://www.c-s-a.org.cn/1003-3254/8062.html

Kubernetes Resource Scheduling Algorithm Based on Genetic Algorithm

HU Cheng-Peng, XUE Tao

(School of Computer Science, Xi'an Polytechnic University, Xi'an 710048, China)

Abstract: In the optimization stage, Kubernetes determines the score of a node only according to its utilization of CPU and memory. This can only guarantee the resource utilization of a single node but fails to achieve the load balancing of cluster resources. In response to this problem, a genetic algorithm-based Kubernetes resource scheduling algorithm is proposed. In the algorithm, two evaluation indicators, i.e., network bandwidth and disk IO, are added and assigned with different weights. In addition, a check dictionary is introduced to check and repair the individuals that do not meet the configuration in the new population generated by the genetic algorithm. Experimental results show that compared with the Kubernetes default resource scheduling strategy, this algorithm takes into account the resource utilization of all nodes in the cluster and performs better in ensuring cluster load balancing.

Key words: Kubernetes; genetic algorithm; resource scheduling; cloud platform; cloud computing

1　引言

随着互联网规模的扩大, 服务器产生的海量数据促使云计算[1] 的快速发展. 云计算是指将大量用网络连接的计算资源进行统一管理和调度, 构成一个计算资源池向用户提供服务. 通过云计算, 可以合并组织现有的公司信息资源, 并为其员工和合作伙伴提供通用的远程访问权限[2]. 云计算的问题之一是对资源的利用不均. 传统云计算架构中基础设施及服务层 (IaaS) 的资源分配是以虚拟机为基本单位进行调度, 但是虚拟机的调度是一种粗粒度的资源调度, 会出现调度缓慢等问题.

2013 年, 随着 Docker[3] 等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. 与虚拟机相比, 容器技术具有镜像

① 基金项目: 陕西省技术创新引导专项 (2020CGXNG-012)
Foundation item: Technology Innovation Guiding Project of Shaanxi Province (2020CGXNG-012)
收稿时间: 2020-11-30; 修改时间: 2020-12-28; 采用时间: 2021-01-13; csa 在线出版时间: 2021-09-02

小、资源消耗少、应用部署灵活和启动速度快等优点[4]. 大量的容器依托容器编排工具进行管理和控制, 其决定容器之间如何进行交互. 以 Kubernetes[5] 为代表的容器编排工具渐渐成为云原生的事实标准, 越来越多的微服务使用 Kubernetes 进行部署和管理. Kubernetes 自动化部署的功能可以使开发者轻松部署应用, 自动化管理的功能可以让定义好的服务一直按照用户期望的状态运行, 自动扩容\缩容的功能让服务器拥有的副本数量随着用户访问负载的变化而增减成为可能.

Kubernetes 的资源调度技术是云计算中的关键技术, 良好的资源调度策略可以使云服务提供商提高其资源利用效率, 节约软硬件成本, 同时也可以让用户得到更优质的云服务体验.目前, 云计算资源调度领域的研究集中在两个方向上[6]: 一是根据各种指标寻找最佳调度方案, 并根据其周期性对收集的服务器负载统计信息进行分析. 此方法提供对云平台的连续监视和对其工作量的定期评估, 基于此评估可以决定是否需要重新调度; 二是根据服务器负载的周期性, 使用机器学习方法和时间序列分析, 如 LSTM[7], 通过分析收集的重要时期负载数据, 从而确定调度时机. 何龙等[8] 实现了一种基于应用历史记录的 Kubernetes 调度算法, 该方法实现简单, 对集群资源利用率有一定提高, 但是其只考虑了 CPU 和内存的资源利用率, 诸如网络和磁盘资源等方面的研究不足. 常旭征等[9] 针对 Kubernetes 仅考虑了 CPU 和内存, 加入了磁盘和网络两个指标, 提高了集群资源利用率, 但是并未考虑节点本身的性能指标. Kong 等[10] 通过将时间和可靠性作为资源调度的目标, 以模糊规则作为预测模型, 提出了一种基于模糊规则预测的调度算法. 李天宇[11] 提出了一种基于强化学习的云计算虚拟机资源调度问题的解决方案和策略, 采用 Q 值[12] 强化学习机制实现了虚拟机资源调度策略. 虽然强化学习算法能动态调整自身参数和网络结构, 拥有良好控制策略, 但是模型训练本身需要占用大量资源. Kang[13] 提出了容器代理系统. 该系统使用 k-medoid 算法和分段算法来实现容器工作负载感知和节能, 同时还保证了集群的资源利用率和负载均衡能力, 但是此方法未考虑节点的网络和磁盘等资源利用情况. Rao 等[14] 提出一种分布式学习机制, 将云资源分配视为一种分布式学习任务, 开发了一种增强学习算法并在 iBallon 系统中对分布式学习算法进行了原型设计, 但是该方法仅考虑单个虚拟机资源, 忽略了集群的整体资源性能. Tsoumakos 等[15] 提出了 TIRAMOLA 这一支持云的开源框架, 它可以根据用户定义的策略对 NoSQL 集群进行自动调整大小, 并且这一过程是实时进行的, 但是此框架只对 NoSQL 集群效果显著.

Kubernetes 默认调度机制只考虑了单节点的资源利用率, 未考虑整个集群的负载情况. 因此, 本文针对这一问题提出了一种基于遗传算法的 Kubernetes 资源调度算法. 在种群初始阶段, 通过随机方式初始化种群, 引入校验字典对种群中的个体进行校验, 同时修复不符合配置要求的个体; 在计算适应度值阶段, 将集群平均负载的标准差作为目标函数值, 标准差越小则表示集群负载越均衡; 使用轮盘赌选择方法选择优良个体进入下一代, 在交叉、变异操作后使用校验字典再次对种群中的个体进行校验和修复.

2 默认调度算法分析

Kubernetes Scheduler 是 Kubernetes 资源调度的核心组件, 其职责是将 API Server 或 Controller Manager 新建的待调度 Pod 根据指定的调度算法与集群中的某个合适的工作节点进行绑定, 并将 Pod 和节点的绑定信息写入 ETCD 中[16]. 而后工作节点通过守护进程 Kubelet 监听到 Kubernetes 调度器发出的 Pod 和节点的绑定信息, 并从 ETCD 中获取 Pod 配置文件, 最后根据配置文件完成容器应用的启动.

Kubernetes 调度器中的默认调度算法分为 3 个阶段: 预选阶段 (Predicates)、优选阶段 (Priority) 和选定阶段 (Select). Predicates 阶段的工作是查询集群中的所有节点, 根据 Predicates 的算法选择适用的节点完成初筛. Priority 阶段的工作是根据 Priority 中的算法给 Predicates 阶段初筛的节点进行评分, 挑选出得分最高的节点作为调度的目标节点.

Predciates 阶段要求满足条件的节点必须通过所有筛选策略. 下面介绍几种策略的筛选内容:

1) PodFitsHostPorts: 检查 Pod 请求的端口是否空闲;

2) PodFitsHost: 检查 Pod 是否通过其主机名指定了特定的 Node;

3) PodFitsResources: 检查节点是否有空闲资源 (例如 CPU 和内存) 来满足 Pod 的要求;

4) MatchNodeSelector: 检查 Pod 的节点选择器是否匹配节点的标签;

5) CheckNodeMemoryPressure: 如果节点正在报告内存压力, 并且没有配置的异常, 则不会再此处分配 Pod;

6) CheckNodeCondition: 节点可以报告具有完全完整的文件系统, 网络不可用或者 Kubelet 尚未准备好运行的 Pod. 如果为节点设置了这样的条件, 并且没有配置的异常, 则不会在此处分配 Pod.

优选阶段会根据优选策略对每个节点进行打分, 最终把 Pod 调度到分值最高的节点. Kube-Scheduler 用一组优先级函数处理每个通过预选的节点, 每个函数返回 0–10 的分数, 各个函数有不同权重, 最终得分是所有优先级函数的加权和, 即节点得分的计算公式为:

$$FinalScoreNode = \sum_{i=1}^{n} weight_i * priorityFunc_i \quad (1)$$

式 (1) 中, $FinalScoreNode$ 表示最终得分, $weight_i$ 表示各个函数的权重值, $priorityFunc_i$ 表示具体的优先级函数得分.

在 Priority 阶段的算法有: Least RequestedPriority、NodeAffinityPriority 等. 下面说明这两种算法:

1) Least RequestedPriority 算法默认权重为 1, 此算法尽量将 Pod 调度到计算资源占用比较小的节点上. 此算法设计两种计算资源: CPU 和内存. 计算公式如下:

$$cpuScore = \frac{cpuCapacity - \sum cpuRequest}{cpuCapacity} * 10 \quad (2)$$

$$memoryScore = \frac{memoryCapacity - \sum memoryRequest}{memoryCapacity} * 10 \quad (3)$$

$$leastScore = \frac{cpuScore + memoryScore}{2} \quad (4)$$

式 (2) 中, $cpuCapacity$ 表示候选节点 CPU 资源的总量, $cpuRequest$ 表示 Pod 需要的 CPU 资源量, $cpuScore$ 表示根据 CPU 指标计算的得分. 式 (3) 中, $memoryCapacity$ 表示候选节点内存资源的总量, $memoryRequest$ 表示 Pod 需要的内存资源量, $memoryScore$ 表示根据内存指标计算的得分. 式 (4) 中, $leastScore$ 表示使用 Least RequestedPriority 算法获得的得分.

2) NodeAffinityPriority 默认权重为 1, 此算法尽量调度到标签匹配 Pod 属性要求的节点, 判断行为与预选中的 MatchNodeSelector 类似. 该算法提供两种选择器: requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution. 计算公式如下:

$$nodeAffinityScore = 10 * \frac{\sum_{i=1}^{n} weight_i}{CountWeight} \quad (5)$$

式 (5) 中, $weight_i$ 表示节点满足 requiredDuringSchedulingIgnoredDuringExecution 中所有满足条件的规则的权值, $CountWeight$ 表示 preferredDuringSchedulingIgnoredDuringExecution 中所有规则的权值总和. $nodeAffinityScore$ 表示使用 NodeAffinityPriority 算法获得的得分.

在获取了两种算法对候选节点的评分后, Kubernetes 调度器取两种算法的加权平均值作为各个节点的最终评分:

$$FinalScore = leastScore \times 1 + nodeAffinityScore \times 1 \quad (6)$$

通过分析 Kubernetes 调度器的默认算法可以发现其默认算法对节点的评价指标中只考虑了 CPU 和内存的使用率, 没有考虑磁盘 IO 和网络带宽的使用. 并且 Kubernetes 调度器是一种静态调度策略, 该调度机制虽然简单, 但是缺乏灵活性. 且只能在 Pod 初次部署时进行资源配置. 同时, 默认调度策略只能保证单节点的服务质量, 未考虑整个集群的负载情况. 从而导致集群资源利用率低.

3 Pod 分配模型与遗传算法

Scheduler 根据调度策略将 Pod 部署到不同的节点中, Pod 在节点之间的分配模型如图 1 所示, 其可以说明 Pod 与节点之间的相互关系.

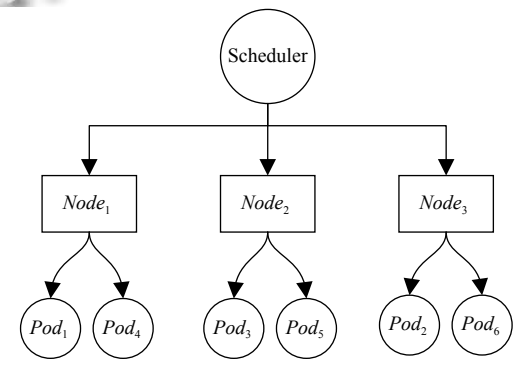

图 1 Pod 在 Node 的分配模型

假设 N 是系统中所有节点的集合, 即 $N=\{Node_1, Node_2, Node_3, \cdots, Node_n\}$, n 表示节点总数, 单一节点表示为 $Node_i$, i 表示节点编号, $i \in [1, n]$. 假设 P 是系统中所有 Pod 的集合, 即 $P=\{Pod_1, Pod_2, Pod_3, \cdots, Pod_m\}$,

m 表示 Pod 总数, 单一 Pod 表示为 Pod_j, j 表示 Pod 编号, $j\in[1, m]$. 假设在某个节点$Node_i$上有一组 Pod, 使用 C 表示节点中 Pod 的分配情况, $C_i=\{Pod_1, Pod_2, Pod_3, \cdots, Pod_k\}$, k 表示$Node_i$上 Pod 的数量. 因此图 1 中的分配情况表示为:$C_1=\{Pod_1, Pod_4\}$, $C_2=\{Pod_3, Pod_5\}$, $C_3=\{Pod_2, Pod_6\}$.

以上模型描述了 Pod 在节点上的分配结果, 默认资源调度模型尽力将待调度 Pod 分配在性能最好的节点上运行, 这会导致集群节点的负载不均衡, 集群整体服务质量下降. 通过分析 Pod 与节点之间的分配关系, 若将 m 个 Pod 分配到 n 个节点中, 根据排列组合可得出共有n^m种情况[17], 这是一个 NP Hard 问题, 在解决这类问题上, 如果问题的固有知识不能被用来减少搜索空间, 很容易产生搜索的组合爆炸.

针对 NP Hard 问题, 常用启发式算法来解决, 其中遗传算法 (GA)[18] 是由 Holland 教授于 1975 年提出的一种模拟生物进化的全局搜索算法. 遗传算法具有智能、并行、自组织、可扩展和易于使用的特点. 遗传算法作为一种启发式智能优化搜索算法, 经常用来解决多目标优化的群体搜索问题.

4 改进的遗传算法

针对目前 Kubernetes 调度算法只考虑单节点资源利用率, 并未考虑集群整体负载均衡的问题, 本文选择遗传算法作为资源调度算法: 在计算适应值阶段, 通过引入磁盘 IO、网络带宽评价指标, 并赋予指标权重值, 使用标准差衡量集群的负载均衡情况. 在 Kubernetes 的 yaml 配置文件中可以配置 Pod 运行的最低要求, 如内存至少为 4 GB, 最大为 16 GB; 磁盘空间至少为 1 TB, 最大为 2 TB 等. 同时, Pod 拥有 NodeAffinity 和 Taint 两种属性. 在标准遗传算法种群初始化阶段、选择阶段、交叉阶段和变异阶段会产生不符合配置文件的个体, 如将一个最低内存配置为 8 GB 的 Pod 分配给了内存为 4 GB 的节点. 对这些个体进行适应度计算和下一阶段操作都是无意义的, 所以针对此问题对标准遗传算法进行了改进: 在初始种群生成阶段、选择阶段、交叉阶段和变异阶段, 通过引入校验字典对不符合配置的个体进行校验和修复.

4.1 种群初始化

4.1.1 编码的确定

编码是应用遗传算法时要解决的首要问题, 也是设计遗传算法时的一个关键步骤. 针对一个具体应用问题, 设计一种完美的编码方案一直是遗传算法的应用难点之一, 也是遗传算法的一个重要研究方向. 由于遗传算法应用的广泛性, 迄今为止人们已经提出了许多种不同的编码方法, 可以分为 3 大类: 二进制编码方法、符号编码方法和浮点数编码方法[19].

根据 Pod 在节点上的分配模型即可得出这是一种 01 背包问题, 1 表示 Pod 分配在此节点上, 反之不分配. 并且, 二进制编码方法是遗传算法中最重要的一种编码方法, 它使用的编码符号集是由二进制符号 0 和 1 所组成的二值符号集{0,1}, 它所构成的个体基因型是一个二进制编码符号. 因此本文选用二进制编码作为编码方案.

4.1.2 校验字典

确定编码后随机产生 N 个初始串结构数据, 每个串结构数据成为一个个体, N 个个体构成了一个种群. 遗传算法以这 N 个串结构作为初始点开始迭代. 设置进化代数计数器; 设置最大进化代数 T; 随机生成 M 个个体作为初始群体 $P(0)$.

本文的染色体 (Chromosome) 编码采用二进制编码方法, 每个个体的基因编码是一个二进制编码符号, 即 0 和 1, 采用二维数组存储个体染色体, 行代表 Pod, 列代表 Node.Chromosome[i][j]=1, 表示要在编号为 j 的节点上分配第 i 个 Pod. 例如 i=1, j=3 表示第 1 个 Pod 被分配在第 3 个节点上. 如果生成了以下染色体{100,001,010}, 通过解码可以知道第 1 个 Pod 在 $Node_1$ 上, 第 2 个 Pod 在 $Node_2$ 上, 第 3 个 Pod 在 $Node_3$ 上. 如表 1 所示为在一次随机初始化种群中产生的一个染色体编码.

表 1 随机初始化个体

Pod	$Node_1$	$Node_2$	$Node_3$
Pod_1	1	0	0
Pod_2	0	1	0
Pod_3	0	0	1
Pod_4	1	0	0
Pod_5	0	1	0
Pod_6	1	0	0
Pod_7	1	0	0

表 1 表示随机初始化种群中的一个个体, 编码为:{100, 010, 001, 100, 010, 100, 100}.

解决 Kubernetes 资源调度问题的目的是找到合理的资源分配方案, 但是由于 Kubernetes 的 Pod 中拥有 NodeAffinity (亲和性) 和 Taint (污点) 两种属性, 使得种群中的个体很可能不符合配置:

1) NodeAffinity 属性从 1.4 版本开始引入. 如果在具有标签 X 的 Node 上运行了一个或者多个符合条件 Y 的 Pod, 那么 Pod 应该运行在这个节点上.

2) Taint 属性让 Pod 避开那些不合适的节点, 在节点上设置一个或者多个 Taint 之后, 除非 Pod 明确声明能够容忍这些污点, 否则无法在这些节点上.

为解决在随机初始化种群和后续选择操作、交叉操作和变异操作中产生不符合用户配置的个体这一问题, 根据配置文件生成校验字典对生成的个体进行校验操作. 如表 2 表示为一个校验字典.

表 2 校验字典

节点	类型	Pod
$Node_1$	Affine	1, 2
	Forbid	3, 4
$Node_2$	Affine	1, 2, 3, 4
	Forbid	5
$Node_3$	Affine	5, 6
	Forbid	4, 7

表 2 中, Type 为 Affine 表示 Pod 倾向于运行在此节点上, Type 为 Forbid 表示 Pod 禁止运行在此节点上. 通过此校验字典对上文随机生成的个体编码进行校验并修复后所得编码为: {100, 010, 001, 010, 011, 100, 100}.

4.2 多维度加权评价指标

在 Kubernetes 默认调度算法的优选阶段, 只考虑了节点的 CPU 和内存利用率, 但是对于互联网应用, 磁盘 IO 和网络利用率也是十分重要的因素. 因此本文在原有评价指标上加入了磁盘 IO 和网络利用率指标.

对于某一节点, 其单项资源利用率为该节点上所有 Pod 对应该项资源分配综合除以节点该项资源值. 例如使用式 (7) 表示某一节点上 CPU 的资源利用率 $L(Cpu_i)$:

$$L(Cpu_i)=\frac{\sum_{j=1}^{N}CpuPod_{ij}}{PodNum_i}\times 100\% \tag{7}$$

式 (7) 中, $CpuPod_{ij}$ 表示编号为 i 的 Node 节点上编号为 j 的 Pod 的 CPU 使用量, $\sum_{j=1}^{N}CpuPod_{ij}$ 表示编号为 i 的 Node 节点上 CPU 使用总量, $PodNum_i$ 表示 $Node_i$ 上 Pod 的总数. 同理可求得该 Node 上内存、磁盘 IO 和网络的利用率 $L(Mem_i)$、$L(Disk_i)$、$L(Net_i)$. 进而可求得该 Node 上的资源平均利用率:

$$L(N_i)=\frac{L(Cpu_i)+L(Mem_i)+L(Disk_i)+L(Net_i)}{4} \tag{8}$$

在实际应用部署中, 不同节点对资源的倾向性不同, 如计算型节点更倾向于 CPU 的使用, 磁盘类型为 SSD 的节点更倾向于磁盘 IO 的使用等. 针对不同类型的节点对评价指标的侧重点不同, 引入权重 $weight$ 决定各评价指标对节点负载的影响:

$$L(N_i)=\frac{Weight(Cpu_i)\times L(Cpu_i)+Weight(Mem_i)\times L(Mem_i)+Weight(Disk_i)\times L(Disk_i)+Weight(Net_i)*L(Net_i)}{Count(Weight)} \tag{9}$$

式 (9) 中, $Weight(Cpu_i)$、$Weight(Mem_i)$、$Weight(Disk_i)$、$Weight(Net_i)$ 分别表示赋予 CPU、内存、磁盘、网络的权重值. 不同于式 (9) 中分母为 4, 使用 $Count(Weight)$ 根据 $weight$ 决定分母的值, 若 $weight$ 为 0 则不考虑此指标.

4.3 种群初始化

在遗传算法中使用适应度 (fitness) 来度量群体中各个个体在优化计算中能达到或接近于或有助于找到最优解的优良程度. 适应度函数 (fitness function) 也称为评价函数, 是根据目标函数确定的用于区分群体中个体好坏的标准, 是算法演化的驱动力, 也是进行自然选择的唯一依据. 适应度较高的个体遗传到下一代的概率较大; 而适应度较低的个体遗传到下一代的概率较小[20].

在概率论中常用方差来度量随机变量和均值之间的偏离程度, 它刻画了一个随机变量值的分布范围. 方差越大, 表示数据的波动越大, 方差越小, 表示数据的波动就越小. 为了使 Kubernetes 集群负载均衡, 因此将可行解的平均资源利用率标准差作为适应度的判定条件.

首先, 计算每一种分配方案的资源平均利用率 $L(Clu_k)$, 式 (10) 中 N 表示符合要求的可行解数量:

$$L(Clu_k)=\frac{\sum_{i=1}^{N}L(N_i)}{N} \tag{10}$$

其次, 通过每一 Node 上的资源评价利用率 $L(N_i)$ 和每一种分配方案的资源平均利用率 $L(Clu_k)$ 可以很方便的计算出每一种分配方案的标准差:

$$\sigma(Clu_k)=\sqrt{\frac{\sum_{i=1}^{N}(L(N_i)-L(Clu_k))^2}{N}} \tag{11}$$

在遗传算法中, 各个个体被遗传到下一代的种群中的概率是由该个体的适应度来确定的. 适应度越高的个体遗传到下一代的概率就越大. 为了使得集群负载均衡, 标准差越小越好, 所以我们无法直接使用标准差作为适应度值, 通常对于求解最小值的问题我们需要进行转换操作, 因此适应度函数如下所示:

$$Fit(\sigma(Clu_k))=\begin{cases}Const_{\max}-\sigma(Clu_k), & Const_{\max}>\sigma(Clu_k)\\ 0, & \text{其他}\end{cases} \tag{12}$$

式 (12) 中, $Const_{\max}$ 为一个适当的相对较大的值, 可以是标准差的最大估计.

5 实验结果与分析

为验证改进算法在 Kubernetes 集群上的有效性, 本文分别设计 3 组实验, 实验采用 Kubernetes1.17 版本, 通过虚拟机的方式部署在宿主机上. 宿主机配置为 64 位 Windows10 系统, CPU 为 i7-8750H, 内存为 32 GB.

实验 1. 验证引入校验字典的改进遗传算法有更少的迭代次数. 实验结果如图 2 所示. 标准遗传算法从第 190 代开始目标函数值趋于最小值, 使用校验字典的改进遗传算法在第 110 代开始目标函数值趋于最小值. 因此, 与标准遗传算法相比, 改进的遗传算法获得相同的目标函数值所需的迭代次数更少.

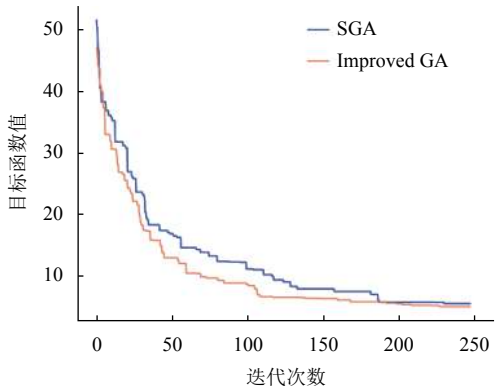

图 2 两种算法迭代次数对比

实验 2. 对 Kubernetes Scheduler 默认资源调度器与基于改进遗传算法的自定义调度器 (custom scheduler) 在保证集群整体负载均衡能力上进行对比. 实验结果如图 3 所示. 由图可知, 使用 Kubernetes 默认调度器的集群负载比使用改进遗传算法自定义调度器的集群的平均负载更高, 前者标准差为 3.421 93, 后者标准差为 1.102 72, 说明本文算法更能保证集群的负载均衡.

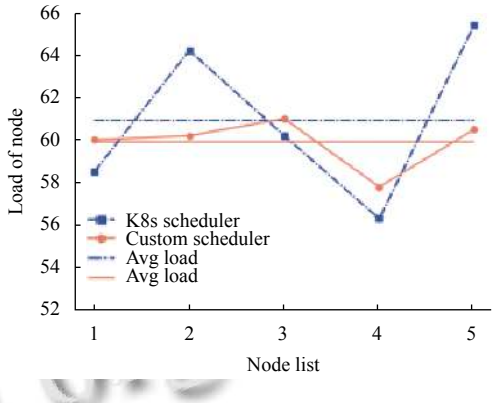

图 3 集群节点负载

实验 3. 首先验证引入网络和磁盘 IO 指标对集群带宽和磁盘 IO 的影响, 权重值都为 1. 如图 4、图 5 分别为使用网络指标和使用磁盘指标后 K8s 集群的网络和磁盘 IO 负载情况. 由实验结果可知, 使用 K8s 默认资源调度算法的网络带宽利用率标准差为 0.304 89, 磁盘利用率标准差为 0.263 25. 使用本文算法的网络带宽利用率标准差为 0.1387, 磁盘利用率标准差为 0.174 34. 本文算法的两个指标对应的标准差都比 K8s 默认调度算法小, 说明本文算法在保证带宽和磁盘负载均衡方面优于 K8s 默认资源调度算法.

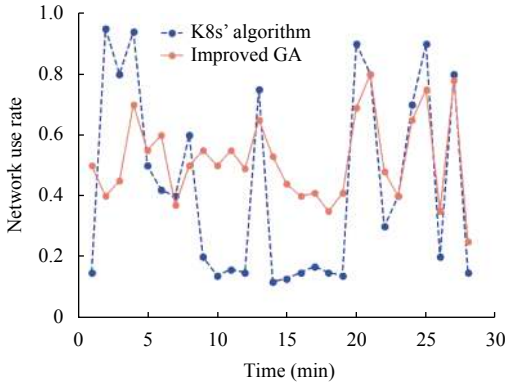

图 4 网络使用率

其次, 为验证权重对不同类型节点的影响, 针对 CPU, 内存, 带宽和磁盘分别创建高性能和低性能两种类型节点进行分组实验, 每组实验 3 个节点, 其中 $Node_1$ 为高性能节点, $Node_2$ 和 $Node_3$ 为低性能节点.各节点对应资源权重值分别设置为: 0.5, 1, 1.

如图 6~图 9 所示为 3 个节点在 120 s 时间内对应资源的使用率情况.

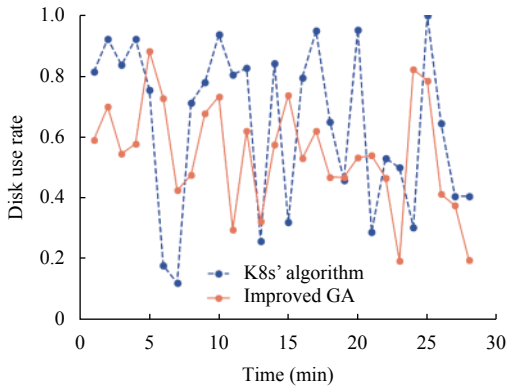

图 5 磁盘使用率

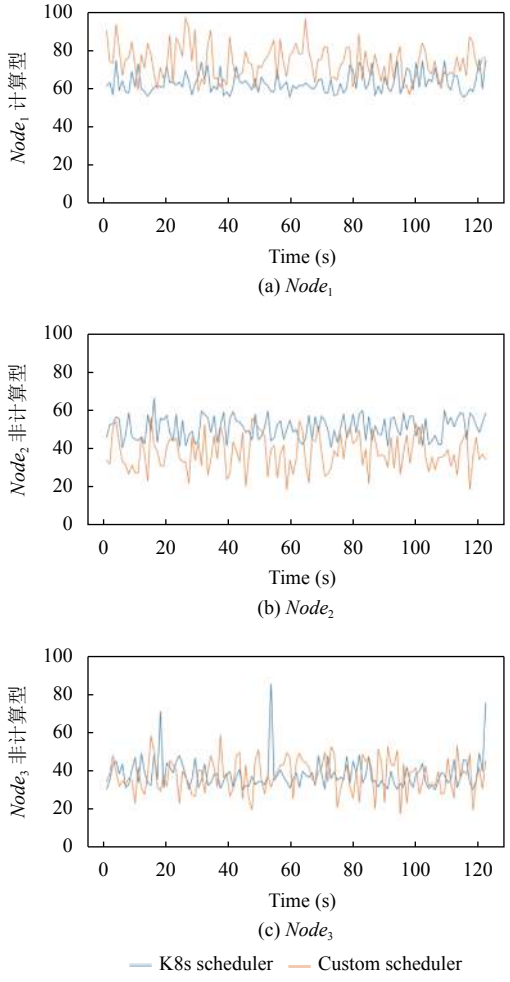

图 6 CPU 权重实验

统计 3 个节点在 120 s 内对应资源的使用率, 对比 Kubernetes 默认资源调度算法和本文算法在不同性能节点上资源使用率均值, 将数据归纳如表 3 所示.

由表 3 可知, 使用本文算法后高性能节点将承受更高的负载, 如高性能内存节点 $Node_1$ 使用 K8s 调度算法使用率为 70.08%, 使用本文算法后使用率为 78.51%, 这是因为赋予高性能节点更低的权重值降低节点负载从而将更多的 Pod 调度至此节点. 但是低性能内存节点 $Node_2$ 使用 K8s 调度算法使用率为 39.61%, 使用本文算法后使用率为 35.32%, 说明本文算法降低了低性能节点负载. 虽然高性能节点负载提高了, 但是由于节点的性能优势并不会给节点带来严重负担.所以, 与 K8s 默认资源调度算法相比, 本文算法结合节点优势特性提高了集群负载能力.

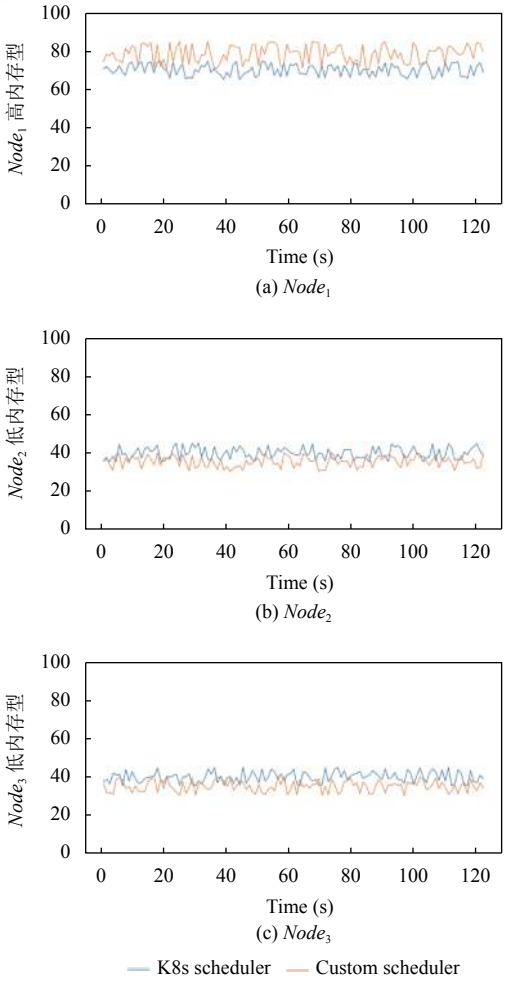

图 7 内存权重实验

6 结语

本文为了优化 Kubernetes 集群平台中受资源调度

影响的负载均衡, 降低集群的平均负载, 提出了基于改进遗传算法的 Kubernetes 资源调度算法, 该算法根据 Kubernetes 中 Pod 拥有的 NodeAffinity 和 Taint 属性, 为降低在随机初始化种群、选择操作、交叉操作和变异操作过程中会产生不符合用户配置的个体对结果的影响, 引入校验字典对种群中的个体进行校验及修复, 实验表明校验字典的引入可以减少算法的迭代次数, 提高算法运行效率. 同时, Kubernetes Scheduler 默认资源调度算法只考虑了 Node 的 CPU 和内存利用率, 结合节点特性提出了多维度加权评价指标, 并使用标准差作为适应度函数, 降低了集群的平均负载且维持了集群的负载均衡. 但与默认资源调度算法相比本文算法对 CPU 的占用过高, 进而影响 Master 节点中其他 Kubernetes 组件的运行, 因此下一阶段工作任务将针对此问题进行研究.

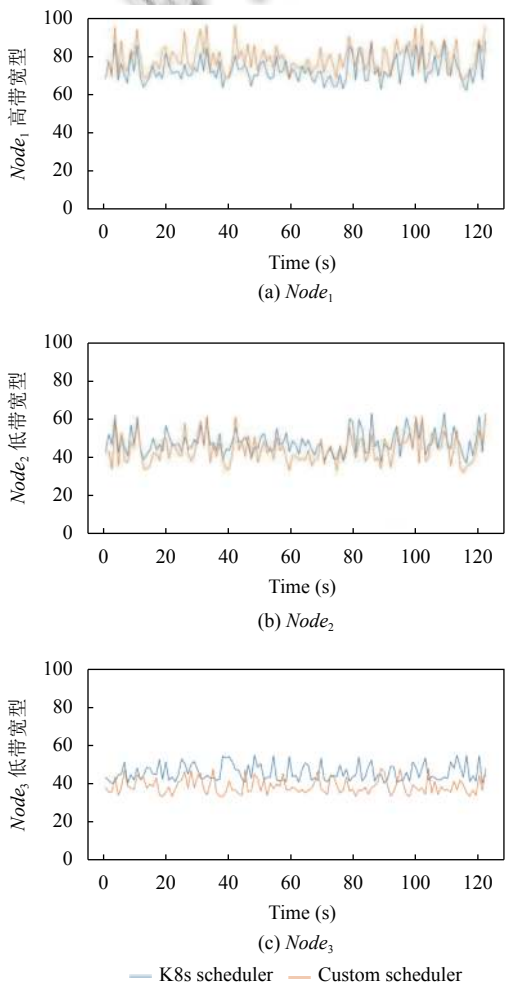

图 8 网络权重实验

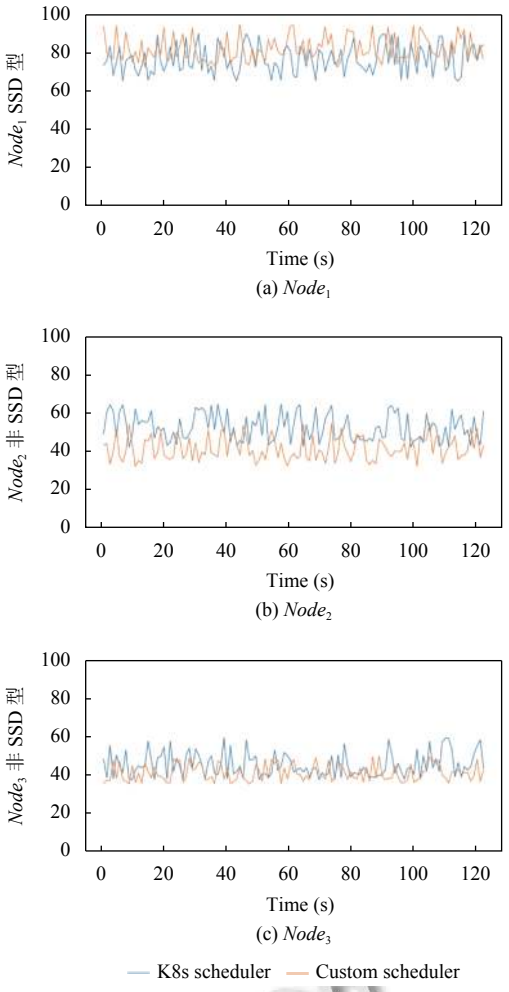

图 9 磁盘权重实验

表 3 不同资源使用两种算法的资源使用率 (%)

资源类型	节点	K8s默认调度算法	本文算法
CPU	$Node_1$	62.79	73.23
	$Node_2$	50.67	37.86
	$Node_3$	38.16	37.52
内存	$Node_1$	70.08	78.51
	$Node_2$	39.61	35.32
	$Node_3$	39.78	34.97
带宽	$Node_1$	72.83	73.92
	$Node_2$	47.83	43.92
	$Node_3$	45.57	38.61
磁盘	$Node_1$	76.53	81.68
	$Node_2$	52.43	41.5
	$Node_3$	45.22	40.91

参考文献

1 张建勋, 古志民, 郑超. 云计算研究进展综述. 计算机应用研究, 2010, 27(2): 429–433. [doi: 10.3969/j.issn.1001-3695.

2010.02.007]
2 Sotomayor B, Keahey K, Foster I. Overhead matters: A model for virtual resource management. Proceedings of the 1st International Workshop on Virtualization Technology in Distributed Computing. Tampa, FL, USA. 2006. 5. [doi: 10.1109/VTDC.2006.9]
3 Sureshkumar M, Rajesh P. Optimizing the docker container usage based on load scheduling. Proceedings of the 2nd International Conference on Computing and Communications Technologies (ICCCT). Chennai, India. 2017.165–168. [doi: 10.1109/ICCCT2.2017.7972269]
4 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. [doi: 10.11772/j.issn.1001-9081.2017.04.0915]
5 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]
6 Volkova VN, Chemenkaya LV, Desyatirikova EN, *et al*. Load balancing in cloud computing. Proceedings of 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering. Moscow and St. Petersburg, Russia. 2018. 387–390.
7 Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735–1780. [doi: 10.1162/neco.1997.9.8.1735]
8 何龙, 刘晓洁. 一种基于应用历史记录的 Kubernetes 调度算法. 数据通信, 2019, (3): 33–36. [doi: 10.3969/j.issn.1002-5057.2019.03.009]
9 常旭征, 焦文彬. Kubernetes 资源调度算法的改进与实现. 计算机系统应用, 2020, 29(7): 256–259. [doi: 10.15888/j.cnki.csa.007545]
10 Kong XZ, Lin C, Jiang YX, *et al*. Efficient dynamic task scheduling in virtualized data centers with fuzzy prediction. Journal of Network and Computer Applications, 2011, 34(4): 1068–1077. [doi: 10.1016/j.jnca.2010.06.001]
11 李天宇. 基于强化学习的云计算资源调度策略研究. 上海电力学院学报, 2019, 35(4): 399–403.
12 Wu B, Feng YP. Policy reuse for learning and planning in partially observable Markov decision processes. Proceedings of the 4th International Conference on Information Science and Control Engineering (ICISCE). Changsha, China. 2017. 549–552. [doi: 10.1109/ICISCE.2017.120]
13 Kang DK, Choi GB, Kim SH, *et al*. Workload-aware resource management for energy efficient heterogeneous Docker containers. Proceedings of 2016 IEEE Region 10 Conference (TENCON). Singapore. 2016. 2428–2431. [doi: 10.1109/TENCON.2016.7848467]
14 Rao J, Bu XP, Xu CZ, *et al*. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. Proceedings of the IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems. Singapore. 2011. 45–54. [doi: 10.1109/MASCOTS.2011.47]
15 Tsoumakos D, Konstantinou I, Boumpouka C, *et al*. Automated, elastic resource provisioning for NoSQL clusters using TIRAMOLA. Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. Delft, the Netherlands. 2013. 34–41.
16 Liu D, Sui X, Li L. An energy-efficient virtual machine placement algorithm in cloud data center. Proceedings of the 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD). Changsha, China. 2016. 719–723.
17 Khodar A, Al-Afare HAF, Alkhayat I. New scheduling approach for virtual machine resources in cloud computing based on genetic algorithm. Proceedings of 2019 International Russian Automation Conference (RusAutoCon). Sochi, Russia. 2019. 1–5. [doi: 10.1109/RUSAUTOCON.2019.8867638]
18 葛继科, 邱玉辉, 吴春明, 等. 遗传算法研究综述. 计算机应用研究, 2008, 25(10): 2911–2916. [doi: 10.3969/j.issn.1001-3695.2008.10.008]
19 张国辉, 吴立辉. 求解柔性作业车间调度的 GATOC 混合方法. 计算机工程与应用, 2015, 51(23): 266–270. [doi: 10.3778/j.issn.1002-8331.1502-0171]
20 贾花萍. 智能算法在数学建模比赛中的应用. 计算机系统应用, 2016, 25(8): 149–154. [doi: 10.15888/j.cnki.csa.005272]