计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(8):86−94 [doi: 10.15888/j.cnki.csa.009182]
©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 的 AI 调度引擎平台①

刘　祥[1], 胡瑞敏[2], 王海滨[3]

[1](西安电子科技大学 广州研究院, 广州 510555)
[2](西安电子科技大学 杭州研究院, 杭州 311231)
[3](厦门市美亚柏科信息股份有限公司, 厦门 361008)
通信作者: 胡瑞敏, E-mail: rmhu@xidian.edu.cn

摘　要: 文中介绍了基于 Kubernetes 的 AI 调度引擎平台的设计与实现, 针对当前人工智能调度系统中存在的服务配置复杂, 集群中各节点计算资源利用率不均衡以及系统运维成本高等问题, 本文提出了基于 Kubernetes 实现容器调度和服务管理的解决方案. 结合 AI 调度引擎平台的需求, 从功能实现和平台架构等方面设计该平台的各个模块. 同时, 针对 Kubernetes 无法感知 GPU 资源的问题, 引入 device plugin 收集集群中每个节点上的 GPU 信息并上报给调度器. 此外, 针对 Kubernetes 调度策略中优选算法只考虑节点本身的资源使用率和均衡度, 未考虑不同类型的应用对节点资源的需求差异, 提出了基于皮尔逊相关系数 (Pearson correlation coefficient, PCC) 的优选算法, 通过计算容器资源需求量与节点资源使用率的互补度来决定 Pod 的调度, 从而保证调度完成后各节点的资源均衡性.

关键词: Kubernetes; 容器; 调度; 皮尔逊相关系数

引用格式: 刘祥,胡瑞敏,王海滨.基于 Kubernetes 的 AI 调度引擎平台.计算机系统应用,2023,32(8):86–94. http://www.c-s-a.org.cn/1003-3254/9182.html

AI Scheduling Engine Platform Based on Kubernetes

LIU Xiang[1], HU Rui-Min[2], WANG Hai-Bin[3]

[1](Guangzhou Institution of Technology, Xidian University, Guangzhou 510555, China)
[2](Hangzhou Institution of Technology, Xidian University, Hangzhou 311231, China)
[3](Xiamen Meiya Baike Information Co. Ltd., Xiamen 361008, China)

Abstract: The design and realization of the AI scheduling engine platform based on Kubernetes is introduced in this paper. To tackle the problems of complex service configuration, the unbalanced utilization rate of computing resources of each node in the cluster and the high cost of system operation and maintenance in the current AI scheduling system, this study proposes a solution based on Kubernetes to implement container scheduling and service management. Combined with the requirements of the AI scheduling engine platform, the various modules of the platform are designed from such aspects as function implementation and platform architecture. At the same time, given the problem that Kubernetes cannot perceive GPU resources, Device Plugin is introduced to collect GPU information on each node in the cluster and report it to the scheduler. In addition, as priority algorithms in Kubernetes scheduling strategy only considers the resource utilization rate and balance degree of the node itself, disregarding the differences in the demand of different types of applications for node resources, priority algorithms based on Pearson correlation coefficient (PCC) is put forward. The scheduling of Pod is determined by calculating the complementary degree of container resources demand and node resource utilization rate, thus ensuring the resource balance of each node after the scheduling.

Key words: Kubernetes; container; schedule; Pearson correlation coefficient (PCC)

① 收稿时间: 2023-01-09; 修改时间: 2023-02-09; 采用时间: 2023-03-03; csa 在线出版时间: 2023-05-22
CNKI 网络首发时间: 2023-05-24

© 中国科学院软件研究所 http://www.c-s-a.org.cn

随着人工智能的快速发展和普及, 越来越多的用户希望通过人工智能相关服务来提升工作效率和质量, 然而随着人工智能服务应用数量的飞速增加, 如何进行合理的服务应用调度来充分利用服务器资源和简化平台的运维流程成为一个摆在企业级软件面前的现实问题. 美亚柏科作为深耕公安大数据领域的龙头企业, 承建全国超三分之一的省级公安大数据平台, 急需研发适配自身业务需求的 AI 调度引擎平台对图片取证、票据信息提取、图片敏感信息识别等服务应用进行合理调度和运维.

近年来, 云计算中的容器化技术和容器编排技术发展迅速, 特别是以 Docker 为代表的容器化已逐渐取代传统使用虚拟机进行操作系统级别的虚拟, 成为虚拟化技术中新的代表技术[1]. 容器通过将物理机器或裸机划分为数百个容器将资源虚拟化[2], 实现了进程隔离和资源限制. 然而, 随着容器数量的快速增长, 容器化平台的运营和管理复杂性必然会急剧上升, 因此, 与容器化技术对应的容器编排技术也得到了逐步提高, 特别是谷歌于 2014 年开源的 Kubernetes 可以承载以微服务为中心的应用程序, 可用于自动化部署、操作和扩展容器应用程序[3], 是一个完备的分布式系统支持平台, 具有超强集群管理能力[4], 已成为容器管理领域的事实标准[1], VMware 发布的《The State of Kubernetes 2022》[5] 指出 99% 的受访者已经从部署 Kubernetes 中获益, 最大的两个好处是提高资源利用率和简化应用程序升级和维护. 因此, 基于 Kubernetes 改进人工智能开发应用中的容器调度和服务管理具有较强的研究价值和可行性.

国内外学者对 Kubernetes 在人工智能领域的应用做了许多研究. Wu 等人[6] 针对大规模人工智能训练中出现的资源利率低, 机器负载分布不均匀的问题, 引入 Kubernetes 实现自动化应用程序的部署、扩展和操作, 极大程度简化了应用的配置流程, 减少了系统的运维成本. Menouer[7] 在容器调度策略中综合考虑节点的 CPU 利用率、内存利用率、磁盘利用率、功耗、容器数量和传输时间 6 个指标来选择最佳调度节点. Chang 等人 [8] 通过节点资源利用率和应用程序 QoS 指标决定资源的动态分发. 文献 [9–11] 针对 Kubernetes 未能实现对 GPU 资源的访问, 引入 NVIDIA device plugin 插件将 GPU 作为扩展资源进行访问. Yeh 等人[12] 设计的 KubeShare 扩展了 Kubernetes以支持 GPU 共享和细粒度分配. 文献 [13,14] 利用K-均值和皮尔逊相关系数等聚类思想将虚拟机放置于与其资源互补的物理节点上, 充分利用了节点资源. 但是其仍只考虑节点中的 CPU 和内存两种资源, 未考虑 GPU 资源的使用情况, 使得该算法不适用于人工智能等需要 GPU 资源的业务中.

通过上述研究, 针对企业在容器调度过程中遇到的困难, 本文提出了一种基于 Kubernetes 实现的 AI 调度引擎平台解决方案, 该平台通过 device plugin 实现对 GPU 资源的感知和调度; 同时研究了 AI 调度引擎平台的运行流程和实现细节, 实现了 AI 服务构建的全流程管理; 最后在李想和黄纬等人的基础上完善了基于皮尔逊相关系数的优选策略, 综合节点上 GPU 资源的使用情况决定调度结果, 成功减少了服务器中资源碎片的产生, 提高了调度平台的整体性能和节点的资源均衡性.

1 Kubernetes 架构[15]

Kubernetes (k8s) 是一个容器编排平台, 用于容器调度、自动化部署、管理和扩展容器化应用. 如今, k8s 和更广泛的容器生态系统逐渐融合, 正在形成一个通用的计算平台和生态系统, 为企业提供高生产力的平台即服务 (PaaS), 解决围绕云原生开发、基础架构和运维相关的多个任务及问题[16]. Kubernetes 的集群由 Master 节点和 Node 节点构成, 其结构如图 1 所示.

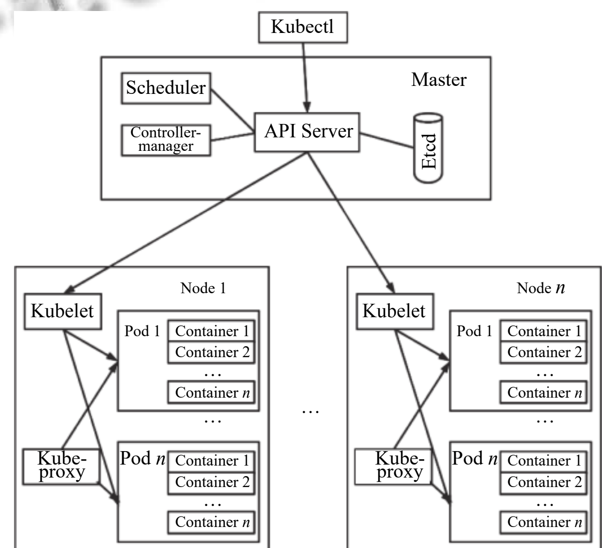

图 1 Kubernetes 集群架构图

© 中国科学院软件研究所 http://www.c-s-a.org.cn

Master 节点负责整个集群的管理和控制, 是 k8s 中的集群控制节点. k8s 的所有控制命令都将发送到 Master 节点, 由它负责具体的执行过程. Master 节点上运行着 API Server, controller manager, scheduler, ETCD 等组件. 其中 scheduler 是 Kubernetes 的调度器, 会根据特定的调度算法和调度策略将 Pod 调度到合适的 Node 节点上去.

Node 节点是 k8s 集群中的工作负载节点, 每个 Node 都会被 Master 分配一些工作负载, 当某个 Node 节点故障时, 其上的工作负载会被 Master 自动转移到其他节点上去. Node 节点维护着运行的 Pod 并提供 Kubernetes 运行时环境, 运行着 kubelet, kube-proxy 等组件.

2 AI 调度引擎平台设计

2.1 模块设计

该平台围绕 AI 镜像的封装和调度, 将 AI 模型以镜像的形式部署到 Kubernetes 所管理的平台中并新建 Pod, 然后对 Pod 配置接口地址等信息, 以服务的形式供外部用户使用, 最后对服务进行安全管控, 防止恶意请求.

经过对平台的需求分析, 为了保证各模块之间符合低耦合高内聚的要求, 将整个平台分为算子模块、算子实例模块、服务模块和安全管控模块, 各模块之间的交互关系如图 2 所示.

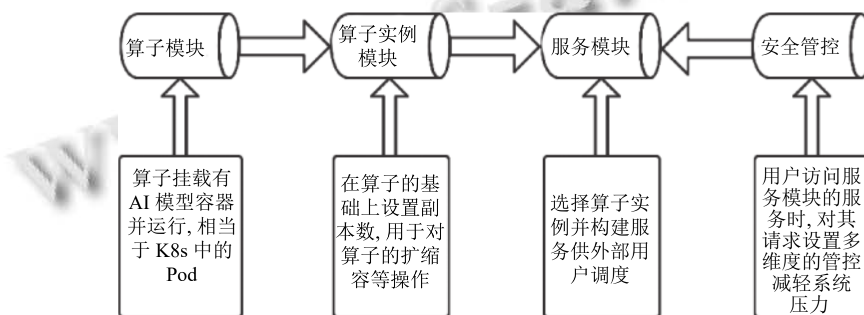

图 2 调度引擎平台模块间的交互关系

该 AI 调度引擎平台的模块主要有:

(1) 算子模块: 算子是具备例如 AI 能力、逻辑运算、数据接入、数据输出等功能的数据处理逻辑单元. 用户通过为算子绑定相关 AI 模型镜像使其具备对应的 AI 能力. 该模块可以实现算子的新增、上线、下线等操作.

(2) 算子实例模块: 算子实例是在算子的基础上设置实例名称和副本数. 该模块可以实现算子实例的上线、下线、扩缩容等操作. 算子实例的上线即向 Kubernetes 的 API Server 发送请求创建 deployment, 然后由 deployment 创建和管理 Pod, 并将 Pod 调度到合理的工作节点上运行.

(3) 服务模块: 当 Pod 调度到工作节点后, AI 应用便已部署到服务器上并分配所需资源, 但此时外部用户并不能简易的访问该 AI 应用, 可以先在算子实例的基础上进行对应的配置构成服务, 用户通过服务调度对应的 AI 能力.

(4) 安全管控: 安全管控模块可对服务模块构建的服务提供多维的请求控制, 如支持流量控制、黑白名单、中断控制等功能, 保证系统的安全性和稳定性.

2.2 AI 调度引擎平台架构实现

在该调度平台中, 资源对象主要为 Pod、deployment、service 这 3 类, 三者构成了该平台的具体架构, 如图 3 所示.

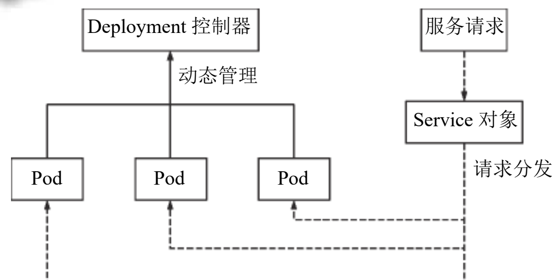

图 3 平台架构实现

架构最底层的 Pod 承载有容器镜像, 挂载了对应的 AI 模型镜像文件, 并运行着对应的程序, 是 Kubernetes 中执行功能的最基本单位, 对应于该平台中的算子. 在本平台中, 一个 Pod 可以包含一个或多个容器, 以便将多个紧耦合的应用容器作为一个实体来管理, 并为每

© 中国科学院软件研究所 http://www.c-s-a.org.cn

个容器设置应用存活探针和应用业务探针定时检测容器是否启动完成和是否处于就绪状态.

每个 Pod 都运行着给定应用程序的单个实例, 若想水平扩展应用程序则会带来较多的运维成本, 本平台只需在算子实例模块为 Pod 设置副本数量, 即可通过 deployment 控制器实现对服务的扩缩容, 避免了运维人员的繁琐操作.

Service 对象实现了对 Pod 的代理, 保证了客户端请求时, 上游服务不受 Pod 的变动而影响. 为了使 service 对集群外的用户提供服务, 将其类型设置为 NodePort, 这样, service 就定义了一个服务的访问入口地址, 用户的服务请求通过这个入口地址访问其背后的一组由 Pod 副本组成的集群实例, 并负载均衡到后端的各个容器应用上.

该平台架构中的每台服务器都会作为 Node 节点提供底层的物理机计算资源. 每个 Pod 创建后会根据 Kubernetes 的预选和优选算法将其调度到某个 Node 节点上.

2.3 Kubernetes 实现对 GPU 资源管理

GPU 作为一种最初专门针对图形计算而设计的处理器, 逐渐被设计成专门做类似运算的处理器, 现在也广泛应用于人工智能中以加速 AI 计算. 然而 Kubernetes 本地识别和分配的计算资源只有 CPU 和内存[12], 未能实现对 GPU 资源的感知和调度, 因此本平台利用 Kubernetes 热插拔的特性, 通过插件扩展的机制来访问和管理 GPU 资源, 使 Kubernetes 可根据用户的 GPU 请求来决定 Pod 的调度结果并为其分配所需资源, 扩展了其应用场景.

本平台通过 device plugin 使 Kubernetes 根据约束将 Pod 正确调度到具有 GPU 资源的节点上, 并实现后续的资源分配, 但是目前 Kubernetes 仅支持卡级别的调度, 即用户 Pod 可以申请独占一张或者多张 GPU 卡[17]. 用户只需在平台中的算子模块为容器组中的每个容器设置 GPU 卡数, 便会通过生成的 YAML 文件向系统申请对应数量的 GPU 资源, 无需再通过命令行进行申请, 降低了平台使用门槛. Device plugin 与 Kubernetes 交互机制如图 4 所示.

当 Pod 需要使用 GPU 资源时, 其工作流程如下.

(1) Device plugin 以容器的方式运行在节点上, 通过 gRPC 协议向 kubelet 内部集成的 device manager 发起注册请求.

(2) Kubelet 将设备数量和 Node 状态上报到 API Server, 以便后续调度器根据这些信息进行调度.

(3) Kubelet 会建立一个到 device plugin 的 ListAndWatch 长链接来发现该 Node 上 GPU 设备的 ID 和健康状态, 当某个设备不健康时会主动通知 kubelet, 并将这个设备从可调度的 GPU 列表中移除.

(4) 当 Pod 需要申请一个 GPU 资源时, 只需在 Pod 的 limits 字段声明: nvidia.com/gpu: 1, Kubernetes 的调度器便会寻找 GPU 数量满足条件的 Node, 完成 Pod 与 Node 的绑定.

(5) 当 kubelet 发现调度成功的 Pod 请求 GPU 资源时, kubelet 就会向 device plugin 发起 allocate 请求为 Pod 分配 GPU 资源.

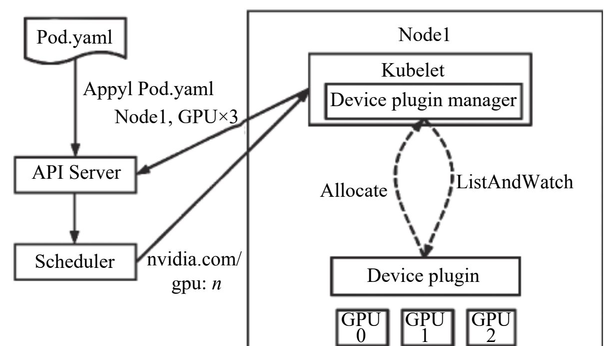

图 4 Device plugin 与 Kubernetes 交互机制

3 AI 调度引擎平台的运行

3.1 算子的创建

在实现调度平台模块功能的过程中, 首先需要结合实际业务场景从镜像仓库中选择对应的镜像, 然后在镜像的基础上设置所需的计算资源、启动命令等封装为容器组, 最后设置算子的基础信息即可完成模型算子的构建; 考虑到平台实际应用中可能调用第三方的 AI 镜像实现既定功能, 因此该模块也可以通过 URL 使用第三方的容器镜像完成三方算子的构建. 算子创建后即可向 Kubernetes 的 API Server 发起请求创建 Pod. 创建算子的流程如图 5 所示.

3.2 算子实例的构建和调度流程

当算子创建完成后, 即可向 Kubernetes 的 API Server 请求创建 Pod 并调度到合适的节点上运行. 为了提高系统的可用性, 平台并不直接创建 Pod, 而是按照前文图 3 所示结构通过 deployment 实现 Pod 的创

© 中国科学院软件研究所 http://www.c-s-a.org.cn

建, 并设置对应 Pod 的副本数构成算子实例, 算子实例对应 Kubernetes 中的 deployment.

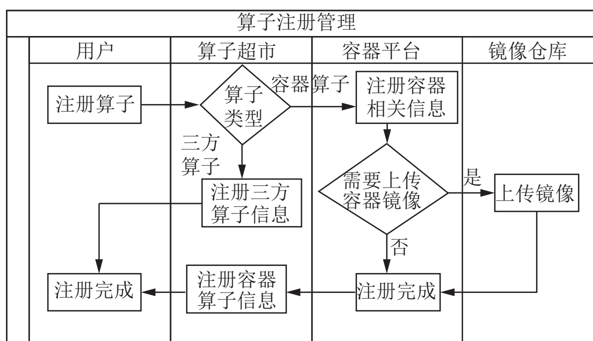

图 5　算子注册流程

当算子实例创建并完成调度后, 集群会根据 deployment 设置的约束生成相应数量的 Pod 供用户调用. 然而, 此时 Pod 的 IP 是不固定的, 当 Pod 重启后 IP 等状态信息可能会变动, 因此需要按照图 3 所示构建 service, 外部请求通过 service 负载均衡到 Pod.

由于 Kubernetes 原生不支持对 GPU 等异构资源的访问, 而 AI 应用往往需要 GPU 资源进行计算, 因此需要为平台引入 device plugin, 使 Kubernetes 实现将服务调度到多节点的 GPU 上. 当用户请求创建 Pod 时, kubelet 会根据 YAML 文件向 API Server 发送一个请求命令, 然后待创建的 Pod 会被放入到一个 FIFO 队列中. 调度器 scheduler 每次从这个队列中取出一个 Pod, 根据相应的预选算法过滤掉不满足条件的节点, 然后对预选出来的节点根据优选算法进行打分, 将 Pod 与最优节点进行绑定完成调度. 调度流程如图 6 所示. 调度算法作为调度平台的核心, 合理的调度策略和算法, 可以提升集群的资源使用率, 减少资源的浪费, 提升集群运行的稳定性和效率[18].

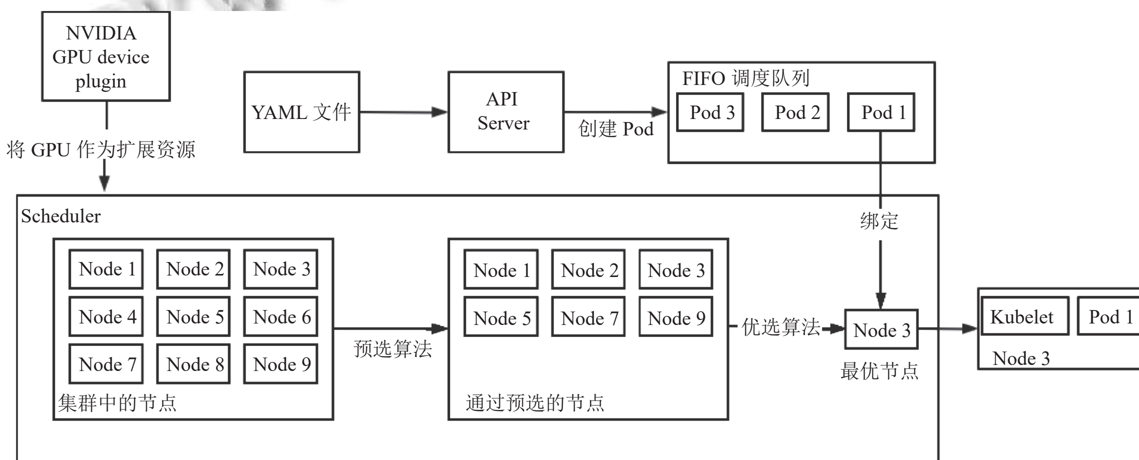

图 6　Pod 调度流程

3.3　服务创建与安全管控

当 Pod 完成调度后, 绑定的 Node 节点便已为其分配所需的各种资源. 但直接通过 Pod 或 service 调用对应的 AI 能力仍需一定的技术门槛且操作繁琐, 这对普通用户来说是不现实的. 服务创建模块实现了通过选择算子来创建对应的 AI 服务, 并选择协议类型和认证方式, 然后设置接口地址、连接超时时间等配置, 最后为服务设置服务描述等基本信息, 形成服务接口供用户调用. 用户可通过已创建的服务接口直接使用对应的 AI 能力.

当服务构建完成后, 为防止过量请求对服务器造成危害, 可对服务的访问请求设置多维度的访问控制, 平台共实现 4 种请求管控类型, 分别是请求管控、流量管控、中断管控、黑白名单管控, 各类安全管控的功能描述如表 1 所示.

表 1　安全管控类型

管控类型	功能
请求管控	配置具体服务的调用次数、频率, 减少服务提供方的系统压力, 超过上限将被限制调用
流量管控	限制请求流量, 防止非预期的大流量请求压垮业务系统
中断管控	当一个服务出现问题时, 停掉该服务并返回指定代码和指定消息, 防止系统的雪崩效应
黑白名单管控	设置服务调用的黑白名单, 根据请求客户端的IP或者IP段来限制服务可否调用的状态

© 中国科学院软件研究所 http://www.c-s-a.org.cn

4 AI 调度引擎平台的调度算法优化

4.1 Kubernetes 默认调度策略

Scheduler 作为 Kubernetes 的集群调度器, 它根据用户创建的 Pod 请求为 Pod 找到一个合适的节点并运行在其上[19]. 前文中提到, Pod 在调度过程中需要经过预选策略和优选策略后与最优节点进行绑定.

预选是根据用户提交的 YAML 文件, 遍历集群中的 Node 节点, 过滤掉不符合用户定义要求的节点. 优选过程则是为通过预选的 Node 节点进行打分, 选出评分最高的 Node 节点与 Pod 绑定. Scheduler 默认采用的优选算法为 LeastRequestedPriority 算法和 BalanceResourceAllocation 算法. LeastRequestedPriority 算法的目标是在通过预选的 Node 节点上选取一个资源 (内存和 CPU) 占比最小的节点, 其计算公式为:

$$score=\frac{(cpuNodeCap-sum(cpuPodReq))\times 10}{2\times cpuNodeCap}+\frac{(memoryNodeCap-sum(memoryPodReq))\times 10}{2\times memoryNodeCap} \tag{1}$$

其中, $cpuNodeCap$ 和 $memoryNodeCap$ 分别表示节点可分配的 CPU 和内存, $cpuPodReq$ 和 $memoryPodReq$ 分别表示待调度 Pod 的 CPU 和内存的请求量.

BalanceResourceAllocation 算法的目标是挑选 CPU 和内存使用率更加接近的节点, 该算法平衡了节点各种资源使用率的均衡性. 其计算公式为:

$$score=10-\left|\frac{cpuPodReq}{cpuNodeCap}-\frac{memoryPodReq}{memoryNodeCap}\right|\times 10 \tag{2}$$

4.2 基于皮尔逊相关系数的优选调度算法

在一个拥有多种资源的系统中往往需要面对资源平均分配的问题, 每个用户对每个资源都有不同的需求, 如果某一节点上某种资源接近满载, 而其他资源使用率却很低, 就会导致利用率低的资源由于该节点无法调度 Pod 而被浪费掉. 上述 Kubernetes 采取的默认优选算法虽然在一定程度上维护了 Node 节点资源使用率的均衡, 但是也存在两处不足: (1) BalanceResourceAllocation 算法仅通过 CPU 和内存使用率的差值来判定节点的资源均衡性, 该算法相对比较简单, 不能较好的衡量节点资源使用率的均衡性. 且该算法不能满足不同类型应用对资源需求的差异, 如内存敏感型应用对内存的需求较高, 从而造成资源碎片浪费节点资源; (2) 默认调度算法只考虑了 CPU 和内存两种资源指标, 无法满足各异的应用需求[20]. 而本文所实现的 AI 调度引擎平台应用于人工智能领域, 默认调度算法无法满足平台的业务需求.

针对以上两点不足, 为改进默认调度算法存在的缺点, 本文基于皮尔逊相关系数实现了新的优选调度算法, 与默认算法相比显著提高了集群中各节点资源使用率的均衡度. 此外该算法将 GPU 使用情况纳入 Pod 调度的影响因素, 从而满足人工智能等业务的应用需求.

该改进算法的设计思路是通过皮尔逊相关系数法计算 Pod 资源请求量与各 Node 节点资源使用率的相关性, 由于具有负相关性的虚拟机与物理节点之间一般具有更多的资源互补[14], 从而由相关性计算出每一个待调度的 Pod 与 Node 节点之间的互补度, 互补度越高表示 Pod 越合适调度到该 Node 节点上.

本文在量化资源需求时, 将考虑节点的 CPU、内存、GPU 这 3 个指标, 每一个 Pod 的资源请求量的特征矩阵表示如下:

$$p_i=(r_{i1},r_{i2},r_{i3}) \tag{3}$$

其中, p_i 表示第 i 个 Pod 资源需求特征矩阵, r_{i1} 表示的是该 Pod 的 CPU 请求量, r_{i2} 表示的是该 Pod 的内存请求量, r_{i3} 表示的是该 Pod 的 GPU 请求量.

集群中每一个 Node 节点的资源利用率也将从 CPU、内存、GPU 这 3 个指标考虑, 其特征矩阵如下:

$$n_j=\left(d_{j1},d_{j2},d_{j3}\right) \tag{4}$$

其中, n_j 表示第 j 个 Node 节点资源利用率的特征矩阵, d_{j1} 表示该节点的 CPU 使用率, d_{j2} 表示该节点的内存使用率, d_{j3} 表示该节点的 GPU 利用率.

然后通过皮尔逊相关系数法来计算 Pod 资源请求量与集群中各节点资源使用率的相关性, 其计算公式如下所示:

$$\rho_{i,j}=\frac{cov\left(p_i,n_j\right)}{\sigma_{p_i}\sigma_{n_j}}=\frac{\sum_{k=1}^{3}(p_{ik}-\overline{p_i})\left(n_{jk}-\overline{n_j}\right)}{\sqrt{\sum_{k=1}^{3}(p_{ik}-\overline{p_i})^2}\sqrt{\sum_{k=1}^{3}\left(n_{jk}-\overline{n_j}\right)^2}} \tag{5}$$

其中, $\rho_{i,j}$ 表示第 i 个 Pod 和第 j 个 Node 节点的相似度, 其值为 Pod 资源请求量与 Node 节点资源使用率两个变量之间协方差和标准差乘积的比值. 其值介于−1 与 1 之间. 其值越大表明 Pod 和 Node 节点的相关性越高, 值越低表明相关性越低, 值为正数表示正相关, 负数表

© 中国科学院软件研究所 http://www.c-s-a.org.cn

示负相关. $\bar{p}_i$表示第 i 个 Pod 各维度资源请求量的平均值, $\bar{n}_j$表示第 j 个 Node 节点各维度资源使用率的平均值, 其计算公式如下:

$$\bar{p}_i = \frac{1}{3}\sum_{k=1}^{3} p_{ik} \tag{6}$$

$$\bar{n}_j = \frac{1}{3}\sum_{k=1}^{3} p_{jk} \tag{7}$$

通过式 (5) 计算出 Pod 与每一个 Node 节点间的相似度, 最终使用式 (8) 来表示每一个待调度 Pod 与集群中所以 Node 节点之间的互补度.

$$comp_{i,j} = \left(1-\left|\rho_{i,j}\right|\right)\times 10 \tag{8}$$

其中, $comp_{i,j}$ 表示第 i 个 Pod 和第 j 个 Node 节点间的互补度, 其值越大说明第 i 个 Pod 的资源请求量与第 j 个 Node 节点的资源越互补, 从而使调度后的 Node 节点各维度资源越均衡, 因此从维护节点资源均衡度的角度看, Pod 更适合调度到该 Node 节点上.

本文实现的基于皮尔逊相关系数的优选调度算法是将待调度 Pod 调度到与其资源互补度最大的 Node 节点上, 从而保证集群中节点各种资源的均衡度. 同时实现了对 GPU 资源的感知和使用, 使得该算法可应用于人工智能等业务场景. 改进后的 scheduler 调度器执行流程如图 7 所示.

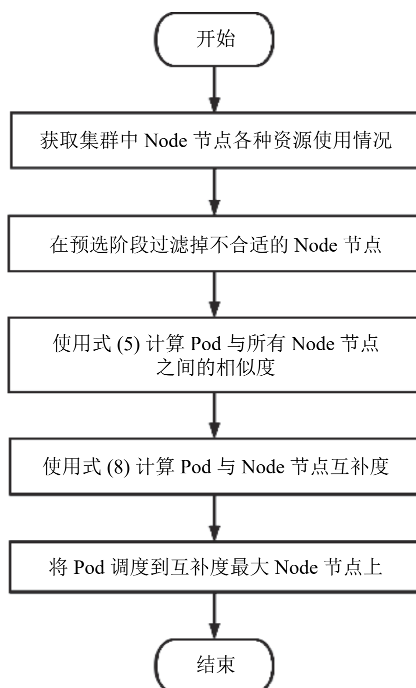

图 7　改进后的调度器执行流程

5　实验结果与分析

5.1　测试环境

本平台是在 3 台采用 Kubernetes 1.18.0 的服务器上进行应用部署和调度实验, 3 台服务器上均安装有 Docker 18.09.7、NVIDIA docker 2、NVIDIA device plugin. 1 台服务器作为 Master 节点, 另外 2 台服务器作为 Node 节点, 为充分利用 Master 节点资源, 将其设置成 Pod 可调度型, 即 Pod 也可分配到 Master 节点并运行. 3 台服务器的配置如表 2 所示.

表 2　实验环境配置

节点	操作系统	CPU核数	内存 (GB)	GPU卡数
Master	Ubuntu 16.04.1	48	503	4
Node 1	Ubuntu 16.04.1	32	125	4
Node 2	Ubuntu 16.04.1	32	125	4

5.2　实验设计

为了验证本文提出的基于皮尔逊相关系数改进的优选调度算法的可行性和有效性, 实验共设置 4 次对照实验, 每次实验设置一个实验组和对照组, 实验组和对照组均创建 6 个资源需求完全相同的 Pod. 对照组采用 Kubernetes 默认的优选算法, 实验组的优选算法采用上述的皮尔逊相关系数算法 (PCC). 为了检测不同场景下皮尔逊相关系数算法的效果, 4 次实验的 Pod 分别设置成 GPU 需求比 CPU 和内存高 (GPU 型)、内存需求比 CPU 和 GPU 高 (内存型)、CPU 需求比内存和 GPU 高 (CPU 型), CPU 与内存和 GPU 需求差不多 (均衡型).

在每次实验中, 分别采用 Kubernetes 的默认优选调度算法和基于皮尔逊相关系数 (PCC) 的优选调度算法对 6 个 Pod 进行调度, 统计每个节点上的 CPU、内存、GPU 使用率, 然后计算每个节点上各维度资源使用率的方差, 用集群中节点方差的平均值来表示集群中节点资源的均衡度, 通过对比说明皮尔逊相关系数 (PCC) 算法能显著提高集群中节点资源的均衡度.

5.3　实验结果

第 1 组实验数据如表 3 所示, 该组实验的 Pod 为 GPU 型, 即 GPU 资源的需求量高于内存和 CPU. 实验数据显示默认优选算法调度后, 集群中各节点资源使用率的平均方差为 230, 采用 PCC 算法调度后平均方差为 217, 表明对于此类 Pod, 基于皮尔逊相关系数的优选算法能明显提高节点各种资源的均衡性.

© 中国科学院软件研究所　http://www.c-s-a.org.cn

表 3 第 1 组实验数据

节点类型	资源类型	默认算法资源使用率	PCC算法资源使用率
Master	CPU	24%	14%
	内存	4%	2%
	GPU	50%	25%
Node 1	CPU	38%	53%
	内存	18%	26%
	GPU	50%	75%
Node 2	CPU	38%	38%
	内存	19%	19%
	GPU	50%	50%
节点平均方差	—	230	217

第 2 组实验数据如表 4 所示, 该组实验的 Pod 为内存型, 即内存资源的需求量高于 CPU 和 GPU. 实验数据显示 PCC 算法调度后的各节点资源使用率的方差均值为 60, 明显小于默认算法的方差 98, 表明对于此类 Pod, 基于皮尔逊相关系数的优选算法能明显提高节点各种资源的均衡性.

表 4 第 2 组实验数据

节点类型	资源类型	默认算法资源使用率	PCC算法资源使用率
Master	CPU	37%	20%
	内存	15%	8%
	GPU	50%	25%
Node 1	CPU	56%	56%
	内存	66%	66%
	GPU	50%	50%
Node 2	CPU	56%	81%
	内存	66%	97%
	GPU	50%	75%
节点平均方差	—	98	60

第 3 组实验数据如表 5 所示, 该组实验的 Pod 为 CPU 型, 即 CPU 资源的需求量高于内存和 GPU. 实验数据显示 PCC 算法调度后集群资源使用率的平均方差为 196, 明显小于默认优选算法的 318. 表明对于此类 Pod, 基于皮尔逊相关系数的优选算法能明显提高节点各种资源的均衡性.

第 4 组实验数据如表 6 所示, 该组实验的 Pod 为均衡存型, 即内存、CPU 和 GPU 资源的需求量类似. 实验结果表明对于此类 Pod, 基于皮尔逊相关系数的优选算法能明显提高节点各种资源的均衡性.

5.4 实验结论

上述 4 次实验分别展示了 GPU 型、内存性、CPU 型和均衡型 4 类 Pod 在 Kubernetes 默认优选算法和基于皮尔逊相关系数优选算法下集群中各节点资源使用率的情况, 并以各节点资源使用率的方差平均值作为衡量均衡度的指标, 并将其绘制成柱状图, 如图 8 所示. 实验证明针对 4 种不同类型的 Pod, 相比于 Kubernetes 默认的优选算法, 本文提出的基于皮尔逊相关系数的优选算法能明显降低集群中节点各维度资源使用率的方差, 提高了调度后节点资源的均衡度, 有效减少了资源碎片的产生, 提高集群整体负载能力.

表 5 第 3 组实验数据

节点类型	资源类型	默认算法资源使用率	PCC算法资源使用率
Master	CPU	63%	23%
	内存	12%	4%
	GPU	75%	25%
Node 1	CPU	36%	95%
	内存	19%	50%
	GPU	25%	75%
Node 2	CPU	66%	66%
	内存	35%	35%
	GPU	50%	50%
节点平均方差	—	318	196

表 6 第 4 组实验数据

节点类型	资源类型	默认算法资源使用率	PCC算法资源使用率
Master	CPU	37%	20%
	内存	13%	6%
	GPU	50%	25%
Node 1	CPU	56%	56%
	内存	55%	55%
	GPU	50%	50%
Node 2	CPU	56%	81%
	内存	56%	82%
	GPU	50%	75%
节点平均方差	—	83	27

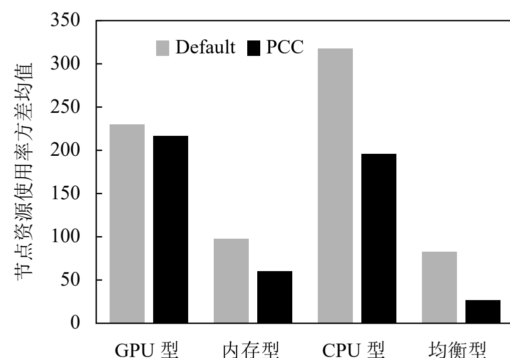

图 8 集群中节点资源使用率方差均值

6 结束语

本文采用 Kubernetes 容器编排技术, 搭建了基于

© 中国科学院软件研究所 http://www.c-s-a.org.cn

容器的 AI 调度引擎平台, 实现了从 AI 镜像到 AI 服务的全流程管理, 用户仅需简易操作便可定制稳定可靠的 AI 服务, 此外本文引入 device plugin 使 Kubernetes 实现了对异构资源 GPU 的访问和管理, 同时优化了 Kubernetes 的优选调度算法, 根据皮尔逊相关系数算法计算待调度 Pod 与集群中节点资源的互补性进行调度, 提高了集群中节点各维度资源的均衡度, 减少了资源碎片的产生. 应用实例证明, 该 AI 调度引擎平台可完成 AI 容器的调度和 AI 服务的创建, 并完成模块设计中的各项功能.

参考文献

1 赵宇峰, 雷晟, 张国钢, 等. 基于容器技术的电力设备仿真云平台设计与开发. 计算机工程, 2021, 47(9): 171–177, 184.

2 Lawrence JJ, Prakash E, Hewage C. Kubernetes: Essential for cloud transformation. Cardiff: Cardiff Metropolitan University, 2021. [doi: 10.25401/cardiffmet.14612397.v1]

3 Fazio M, Celesti A, Ranjan R, *et al*. Open issues in scheduling microservices in the cloud. IEEE Cloud Computing, 2016, 3(5): 81–88. [doi: 10.1109/MCC.2016.112]

4 王艺颖. 基于 Kubernetes 的 TensorFlow 分布式模型训练平台的设计与实现 [硕士学位论文]. 哈尔滨: 哈尔滨工业大学, 2018.

5 VMware. The state of Kubernetes 2022. https://tanzu.vmware.com/content/ebooks/stateofkubernetes2022-ebook.

6 Wu CY, Haihong E, Song MN. An automatic artificial intelligence training platform based on Kubernetes. Proceedings of the 2nd International Conference on Big Data Engineering and Technology. Singapore: Association for Computing Machinery, 2020. 58–62.

7 Menouer T. KCSS: Kubernetes container scheduling strategy. The Journal of Supercomputing, 2020, 77(5): 4267–4293.

8 Chang CC, Yang SR, Yeh EH, *et al*. A Kubernetes-based monitoring platform for dynamic cloud resource provisioning. Proceedings of the 2017 IEEE Global Communications Conference. Singapore: IEEE, 2017. 1–6.

9 El Haj Ahmed G, Gil-Castiñeira F, Costa-Montenegro E. KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters. Software: Practice and Experience, 2021, 51(2): 213–234. [doi: 10.1002/spe.2898]

10 郭效杨. 基于 Kubernetes 的深度学习任务管理平台的设计与实现 [硕士学位论文]. 西安: 西安电子科技大学, 2020.

11 陈培, 王超, 段国栋, 等. Kubernetes 集群上深度学习负载优化. 计算机系统应用, 2022, 31(9): 114–126. [doi: 10.15888/j.cnki.csa.008672]

12 Yeh TA, Chen HH, Chou J. KubeShare: A framework to manage GPUs as first-class and shared resources in container cloud. Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing. Stockholm: ACM, 2020. 173–184.

13 李想. 基于 Kubernetes 容器云的资源调度算法研究 [硕士学位论文]. 西安: 西安电子科技大学, 2020.

14 黄纬, 温志萍, 程初. 云计算中基于 K-均值聚类的虚拟机调度算法研究. 南京理工大学学报, 2013, 37(6): 807–812.

15 Kubernetes. Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/. (2023-01-02).

16 浅谈 AI 任务调度. https://view.inews.qq.com/a/20211015A0A3YY00. (2023-01-02).

17 刘志彬, 黄秋兰, 胡庆宝, 等. Kubernetes 异构资源细粒度调度策略的设计与实现. 计算机工程, 2023, 49(2): 31–36, 45.

18 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.

19 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]

20 张文辉, 王子辰. 基于组合权重 TOPSIS 的 Kubernetes 调度算法. 计算机系统应用, 2022, 31(1): 195–203. [doi: 10.15888/j.cnki.csa.008251]

(校对责编: 牛欣悦)

© 中国科学院软件研究所 http://www.c-s-a.org.cn