计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2020,29(7):256−259 [doi: 10.15888/j.cnki.csa.007545]

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

Kubernetes 资源调度算法的改进与实现①

常旭征[1,2], 焦文彬[1]

[1](中国科学院 计算机网络信息中心, 北京 100190)
[2](中国科学院大学, 北京 100149)
通讯作者: 焦文彬, E-mail: wbjiao@cnic.cn

摘　要: Kubernetes 是 Google 主导的容器编排引擎, 其资源调度算法分为预选和优选两个过程. 针对预选过程要遍历所有节点比较耗时的问题, 改进的资源调度算法提出在选出满足条件的节点数量时直接进行优选而无需遍历所有节点, 从而提高资源调度效率; 针对优选过程只考虑了 pod 自身申请的 CPU 和内存使用情况, 并且未考虑节点本身的资源利用率的问题, 改进的资源调度算法综合考虑 CPU、内存、网络、IO 指标, 通过实验验证了改进算法能适应更加复杂的互联网应用环境, 进而提高集群的负载均衡效率.

关键词: 云计算; Kubernetes; 资源调度

引用格式: 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. http://www.c-s-a.org.cn/1003-3254/7545.html

Improvement and Implementation of Kubernetes Resource Scheduling Algorithm

CHANG Xu-Zheng[1,2], JIAO Wen-Bin[1]

[1](Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100149, China)

Abstract: Kubernetes is Google's leading container orchestration engine. Its resource scheduling algorithm consists of two processes: pre-selection and optimization. The pre-selection process needs to traverse all nodes, which is time-consuming, the improved scheduling algorithm proposes to directly optimize the number of nodes that meet the conditions without having to traverse all, which is expected to improve the scheduling efficiency. For the optimization process, only the CPU and memory usage applied by the pod itself are considered, and the resource utilization of the node itself is not considered. The improved resource scheduling comprehensively considers CPU, memory, network, IO indicators. The improved algorithm is verified through experiments, which can adapt to more complex internet application environments, thereby improving the load balancing efficiency of the cluster.

Key words: cloud computing; Kubernetes; resource scheduling

1　引言

云计算经过十几年的发展, 其底层的虚拟化技术已逐渐成熟, 而 Docker 虚拟化技术由于启动速度快、灵活、轻便等诸多优点逐渐成为云计算领域的热点[1]. 各大云计算厂商也纷纷建立了自己的 Docker 技术平台[2], 与此同时以 Kubernetes[3]为代表的容器编排工具逐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 最早来源于 Google 的 Borg 论文, 理论基础较为完善丰富, 同时扩展性强、自动化程度高, 其主要设计目标是使部署和管理复杂的分布式系统变得容易[3], 它为每个容器分配 IP, 并且可以在集群中的任意位置进行访问, 从而提供了一种分布式的环境[4].

Kubernetes 由一个 master 节点和若干 worker 节点

① 收稿时间: 2019-12-23; 修改时间: 2020-01-20, 2020-02-13, 2020-03-03; 采用时间: 2020-03-11; csa 在线出版时间: 2020-07-03

组成, master 是整个 Kubernetes 集群中的控制节点, 包括 api server、kube-scheduler、controller-manager 三个组件, 其中 api server 负责接收客户端请求, kube-scheduler 负责对 Kubernetes 中的原子调度单元 pod 进行调度, controller-manager 对 Kubernetes 中的控制器进行管理, worker 节点是工作节点, 用来运行具体的 pod. Kubernetes 负责将用户的应用打包为的异构的分布式应用程序, 从而使其在一组虚拟机上可用, 因此, 在这种情况下要解决的一个重要的问题就是调度或放置这些容器化应用程序到一组主机上. 在提交部署应用程序时, 编排系统需要考虑应用程序的特定约束, 尽可能充分的利用各种计算资源, 提高集群的负载均衡, 从而降低企业的成本[5].

Kubernetes 的资源调度算法由于扩展性较好, 逐渐成为国内外学者研究的热点. 以往的资源调度算法仅仅关注优选过程, 对预选过程关注较少, 且优选过程未考虑网络利用率资源指标, 不能很好的提高集群的负载均衡效率, 且资源利用率的计算方法也较为复杂. 本文对 Kubernetes 资源调度算法中的预选和优选过程都进行了改进, 且在优选过程中使用了新的资源利用率计算方式, 并且加入了网络利用率这一新的指标, 以最大程度的提高整个集群的负载均衡效率.

2 Kubernetes 默认的资源调度算法[6]

Kube-scheduler 是 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运行在其上面[7], 这就是调度 pod 的过程. Kubernetes 的调度过程分为两部分, 分别是预选和优选过程.

预选过程是根据用户提交的 yaml 文件, 遍历所有 node, 过滤掉不符合用户定义要求的节点, 例如用户定义的 pod 标签要求使用带有 SSD 硬盘的节点, 那么显然 kube-scheduler 就不会把该 pod 调度到非 SSD 节点上. 通过分析 Kubernetes 源码可知, 其内置的部分预选函数有: CheckNodeCondition, 即检查节点是否正常, NoDiskConflict, 即检测磁盘不能冲突, PodToleratesNodeTaints, 即检查 pod 是否可以容忍节点上的污点等等.

优选过程则是为预选过程选择出的每个节点打分, 本文基于 Least Requested Priority 和 Balanced Resource Allocation 两种原始的算法进行, 这两种算法得分为[0, 10]中的整数, 并且每种算法都有一个权重, 默认为 1[8]. 其中 Least Requested Priority 的设计思想为: node 上的 CPU 和 memory 空闲比例的和越大, 则当前节点得分越高, 这不难理解, 因为 CPU 和内存空闲比例越大代表当前节点可以容纳更多 pod, 不会挤占资源占用率高的节点, 有利于提高整个集群的资源利用率和负载均衡, 其计算公式为:

$$score = \left[\frac{(Ccpu - Rcpu)}{Ccpu} * 10 + \frac{(Cmem - Rmem)}{Cmem} * 10\right] / 2 \tag{1}$$

而 Balanced Resource Allocation 设计思想为计算 CPU 和内存利用率之间的差值, 差值越小, 说明 CPU 和内存使用越均衡, 从而得分越高, 其计算公式为:

$$score = \left[1 - abs\left(\frac{Rcpu}{Ccpu} - \frac{Rmem}{Cmem}\right)\right] * 10 \tag{2}$$

其中, *Rcpu* 代表节点上已申请的 CPU 使用量与当前 pod 申请的 CPU 容量之和, *Rmem* 代表节点上已申请的内存使用量与当前 pod 申请的内存容量之和, *Ccpu* 和 *Cmem* 分别代表节点总的 CPU 容量和内存容量.

3 改进的资源调度算法

从以上分析可以看出, Kubernetes 默认的算法的预选过程要遍历所有节点, 当节点过多会导致预选耗时严重, 因此可以在预选过程中只选出满足条件的节点个数即可, 而无需轮询所有节点, 因为节点的资源利用率是随时变化的, 我们关注的是集群的最终均衡效率, 而不是中间的某一个状态, 轮询所有节点后再打分并不代表最终结果最优. 同时优选过程只考虑了 CPU 个和内存利用率, 而未关注节点本身的网络利用率, 磁盘利用率等指标, 而当今的互联网应用纷繁复杂, 仅靠 CPU 和内存指标显然无法反应出集群的整体情况. 下文将对优选过程进行改进, 对节点的 CPU、内存、网络、磁盘等指标综合考量, 以改进现有的优选过程算法模型.

3.1 算法设计

针对优选过程, 改进的资源调度算法整体思想是合理调度 pod 以提高整个集群的负载均衡效率, 而对于负载均衡效率的计算, 张玉芳等学者[9]在考虑节点本身性能的前提下, 提出了基于负载权值的计算方法, 使用该文献的方法进行负载的计算. 同时谭莉等[8]提出的考虑节点的性能的算法虽然加上了节点性能的考虑, 但对于互联网应用来说, 网络利用率也是不可忽略的因素, 因此本文在原有的负载均衡计算上加入网络利

用率指标, 把 CPU、内存、网络、磁盘 IO 四个维度数据综合考量计算最终得分. 计算公式为:

$$score = \left(1 - \frac{L(i)}{S(i)}\right) * 10 \tag{3}$$

其中, $L(i)$ 表示节点负载, $S(i)$ 表示节点性能.

$L(i)$ 的计算公式为:

$$L(i) = L(c) + L(m) + L(n) + L(d) \tag{4}$$

其中, $L(c)$ 表示 CPU 利用率, $L(m)$ 表示内存利用率, $L(n)$ 表示网络利用率, $L(d)$ 表示磁盘利用率.

$S(i)$ 的计算公式为:

$$S(i) = n * S(c) + S(m) + S(n) + S(d) \tag{5}$$

其中, n 为 CPU 核数, S(c) 表示 CPU 频率, $S(m)$ 表示内存容量, $S(n)$ 表示网络速率, $S(d)$ 表示磁盘 IO 速率.

3.2 算法流程

3.2.1 获取节点资源指标

在本实验中, 使用两个脚本 computeL.sh 和 computeS.sh 分别统计 worker 节点的 CPU、内存、网络、磁盘利用率和 CPU 核数、内存容量、网络速率、磁盘读写速率, 分别表示资源利用率和节点性能.

3.2.2 实现 Kubernetes 自定义调度器

在 Kubernetes 官方文档中, 给出了 3 种实现自定义调度器的方法[10]: (1) 修改原有的 scheduler 模块并重新编译; (2) 重新实现自己的调度器模块; (3) 实现一个称为“scheduler extender”的调度接口, 供调度器调用决策. 本文使用第 2 种方式并根据官方例子实现一个简易的调度器, 这里假设只有两个节点, 多个节点同理, 其流程如图 1 所示.

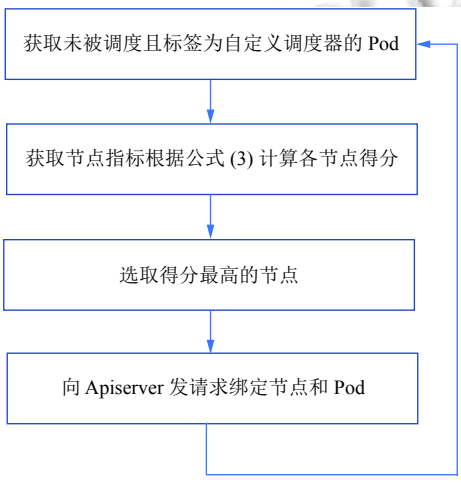

图 1 自定义调度器执行流程

自定义调度器的运行逻辑为一个轮询过程: 首先获取所有标签为 my-scheduler 并且尚未绑定节点的 pod, 其次在所有 worker 节点上分别运行 computeL.sh 和 computeS.sh 两个脚本获取资源利用率和节点性能, 然后计算得到每个 worker 节点得分, 最后将当前调度的 pod 与得分最高的 worker 节点进行绑定, 这个绑定过程通过向 api server 发送 post 请求完成, 从而完成最终的调度. 其伪代码如下:

Input: 各个节点的 CPU、内存、IO、网络利用率及 CPU 核数和频率、内存容量、网络带宽和磁盘容量.

Output: 节点与 pod 的绑定信息.

```
CHOSEN<— null
SERVER<—api server
1:  while true do
2:      for PODNAME in unscheduled where pod.name==my-scheduler do
3:          for node in nodes do
4:              L<— computeL()
5:              S<— computeS()
6:              score=(1-L/S)*10
7:              CHOSEN<— node[max(score)]
8:          end for
9:      curl --data {name: $CHOSEN} http://$SERVER/api/v1/namespaces
10:         /default/pods/$PODNAME/binding/
11:         printf "binding $PODENAME to $CHOSEN "
12:     end for
13:     sleep 1
14: end while
```

其中伪代码第 4 行和第 5 行的两个函数 computeL() 和 computeS() 分别为根据 Linux 命令获取节点性能指标的两个 shell 函数.

4 实验对比与分析

本实验采用 Kubernetes 1.11 版本, 通过虚拟机的方式部署在台式机上, 集群中共有 1 个 master 和 3 个 worker 共 4 个节点, 配置如表 1 所示.

4.1 实验过程

1) 用默认的调度器搭建完集群后, 在集群中起 35 个 pod, 分别执行不同的任务, 然后使用文献[9]中的

方法计算集群的均衡负载; 2) 删除原来的 pod, 使用文献[8]的资源调度算法, 重新起与之前一样的 35 个 pod, 重新计算集群的负载均衡效率; 3) 删除原来的 pod, 使用自定义调度器, 重新起与之前一样的 35 个 pod, 重新计算集群的负载均衡效率; 4) 对比 3 次的实验结果找出差异. 其中, 集群负载均衡效率计算方法为:

$$H(i)=\frac{L(i)/L(avg)}{S(i)/S(avg)}i=1,2,3 \tag{6}$$

其中, $L(avg)$ 代表节点的平均负载, $S(avg)$ 代表节点节点平均性能, $H(i)$ 越大, 说明集群负载均衡越好, 从而资源调度算法也就越好.

表 1 实验环境

虚拟机名称	操作系统	CPU 核数	内存 (GB)	磁盘容量 (GB)	网络带宽 (Mbps)
Master	CentOS 7.5	2	2	20	1000
Worker 1	CentOS 7.5	4	3	20	500
Worker 2	CentOS 7.5	2	2	40	1000
Worker 3	CentOS 7.5	2	2	20	1000

4.2 实验结果

根据式 (6) 分别计算使用默认调度器、文献[8]的调度器 (记为调度器 1) 和本文实现的自定义调度器 (记为调度器 2) 后集群总体的性能差异, 得到对比结果如表 2 所示.

表 2 实验结果 (单位: %)

调度器	Worker 1	Worker 2	Worker 3
默认调度器	73.32	84.34	90.45
调度器 1[8]	79.57	86.28	89.52
调度器 2	82.03	96.95	98.83

图 2 为使用默认调度器、调度器 1 和调度器 2 的情况下集群负载均衡效率柱状图对比, 从实验结果来看, 3 个 worker 节点的集群负载均衡效率对比默认调度器和调度器 1 均有不同程度的提升, 从而验证了资源调度策略的有效性.

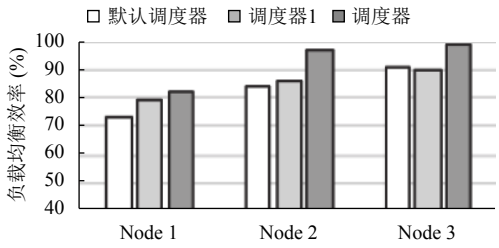

图 2 实验结果

5 结束语

Kubernetes 作为微服务架构最核心的编排工具, 其默认资源调度算法的预选过程由于要遍历所有节点, 节点数量较多时比较耗时, 改进的资源调度算法对预选过程做了优化, 提出关注集群最终的资源利用率, 而不关注中间状态, 因此无需遍历所有节点进而提升资源调度效率. 同时针对优选过程仅仅考虑了 CPU 和内存两个指标, 且只是 pod 申请的 CPU 和内存, 并未考虑节点本身的性能指标的问题, 本文在现有的资源调度算法研究基础上, 加入了除 CPU、内存和磁盘外的网络利用率和速率, 并实现了自定义调度器来应用改进的资源调度算法, 以期提升集群的整体负载均衡效率. 最后通过实验对比分析验证了算法的有效性.

参考文献

1 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]

2 彭丽苹, 吕晓丹, 蒋朝惠, 等. 基于 Docker 的云资源弹性调度策略. 计算机应用, 2018, 38(2): 557–562.

3 Burns B, Grant B, Oppenheimer D, *et al*. Borg, omega, and kubernetes. Communications of the ACM, 2016, 59(5): 50–57. [doi: 10.1145/2890784]

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

5 Rodriguez M, Buyya R. Container Orchestration With Cost-Efficient Autoscaling in Cloud Computing Environments. Handbook of Research on Multimedia Cyber Security. IGI Global, 2020. 190–213.

6 Kubernetes Scheduler. https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/#default-policies. [2019-12-06].

7 左灿, 刘晓洁. 一种改进的 Kubernetes 动态资源调度方法. 数据通信, 2019, (2): 50–54. [doi: 10.3969/j.issn.1002-5057.2019.02.012]

8 谭莉, 陶宏才. 一种基于负载均衡的 Kubernetes 调度改进算法. 成都信息工程大学学报, 2019, 34(3): 228–231.

9 张玉芳, 魏钦磊, 赵膺. 基于负载权值的负载均衡算法. 计算机应用研究, 2012, 29(12): 4711–4713. [doi: 10.3969/j.issn.1001-3695.2012.12.080]

10 Scheduler extender. https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md. [2019-12-06].