深入理解Kubernetes资源限制：内存

Mark Betz 容器时代 2018-11-07

2018上海KubeCon

Kubernetes的全球盛会KubeCon将于11月13日~11月15日在中国上海隆重举行，此论坛汇集了众多在开源和云原生领域有卓越贡献的应用人员和技术专家。大会吸引了超过5000名行业精英前来参会，大家齐聚一堂相互分享经验，聚焦创新，并讨论云原生计算的未来。KubeCon + CloudNativeCon中国论坛将召开100多个分组会议，包括技术会议、深度学习、案例研究等。现在通过容器时代专属报名通道报名可以享受超大折扣哦，详情请戳此处链接：【容器时代粉丝专属福利】KubeCon + CloudNativeCon门票惊喜折扣

写在前面

当我开始大范围使用Kubernetes的时候，我开始考虑一个我做实验时没有遇到的问题：当集群里的节点没有足够资源的时候，Pod会卡在Pending状态。你是没有办法给节点增加CPU或者内存的，那么你该怎么做才能将这个Pod从这个节点拿走？最简单的办法是添加另一个节点，我承认我总是这么干。最终这个策略无法发挥出Kubernetes最重要的一个能力：即它优化计算资源使用的能力。这些场景里面实际的问题并不是节点太小，而是我们没有仔细为Pod计算过资源限制。

资源限制是我们可以向Kubernetes提供的诸多配置之一，它意味着两点：工作负载运行需要哪些资源；最多允许消费多少资源。第一点对于调度器而言十分重要，因为它要以此选择合适的节点。第二点对于Kubelet非常重要，每个节点上的守护进程Kubelet负责Pod的运行健康状态。大多数本文的读者可能对资源限制有一定的了解，实际上这里面有很多有趣的细节。在这个系列的两篇文章中我会先仔细分析内存资源限制，然后第二篇文章中分析CPU资源限制。

资源限制

资源限制是通过每个容器containerSpec的`resources`字段进行设置的，它是v1版本的ResourceRequirements类型的API对象。每个指定了"limits"和"requests"的对象都可以控制对应的资源。目前只有CPU和内存两种资源。第三种资源类型，持久化存储仍然是beta版本，我会在以后的博客里进行分析。大多数情况下，deployment、statefulset、daemonset的定义里都包含了podSpec和多个containerSpec。这里有个完整的v1资源对象的yaml格式配置：

```
resources:
    requests:
        cpu: 50m
        memory: 50Mi
  limits:
        cpu: 100m
        memory: 100Mi
```

这个对象可以这么理解：这个容器通常情况下，需要5%的CPU时间和50MiB的内存（requests），同时最多允许它使用10%的CPU时间和100MiB的内存（limits）。我会对requests和limits的区别做进一步讲解，但是一般来说，在调度的时候requests比较重要，在运行时limits比较重要。尽管资源限制配置在每个容器上，你可以认为Pod的资源限制就是它里面容器的资源限制之和，我们可以从系统的视角观察到这种关系。

内存限制

通常情况下分析内存要比分析CPU简单一些，所以我从这里开始着手。我的一个目标是给大家展示内存在系统中是如何实现的，也就是Kubernetes对容器运行时（docker/containerd）所做的工作，容器运行时对Linux内核所做的工作。从分析内存资源限制开始也为后面分析CPU打好了基础。首先，让我们回顾一下前面的例子：

```
resources:
    requests:
        memory: 50Mi
    limits:
        memory: 100Mi
```

单位后缀 `Mi` 表示的是MiB，所以这个资源对象定义了这个容器需要50MiB并且最多能使用100MiB的内存。当然还有其他单位可以进行表示。为了了解如何用这些值是来控制容器进程，我们首先创建一个没有配置内存限制的Pod:

```
$ kubectl run limit-test --image=busybox --command -- /bin/sh -c "while true; do sleep 2; done"
deployment.apps "limit-test" created
```

用Kubectl命令我们可以验证这个Pod是没有资源限制的：

```
$ kubectl get pods limit-test-7cff9996fc-zpjps -o=jsonpath='{.spec.containers[0].resources}'
map[]
```

Kubernetes最酷的一点是你可以跳到系统以外的角度来观察每个构成部分，所以我们登录到运行Pod的节点，看看Docker是如何运行这个容器的：

```
$ docker ps | grep busy | cut -d' ' -f1
5c3af3101afb
$ docker inspect 5c3af3101afb -f "{{.HostConfig.Memory}}"
0
```

这个容器的 `.HostConfig.Memory` 域对应了 `docker run` 时的 `--memory` 参数，0值表示未设定。Docker会对这个值做什么？为了控制容器进程能够访问的内存数量，Docker配置了一组control group，或者叫cgroup。Cgroup在2008年1月时合并到Linux 2.6.24版本的内核。它是一个很重要的话题。我们说cgroup是容器的一组用来控制内核如何运行进程的相关属性集合。针对内存、CPU和各种设备都有对应的cgroup。Cgroup是具有层级的，这意味着每个cgroup拥有一个它可以继承属性的父亲，往上一直直到系统启动时创建的root cgroup。

Cgroup可以通过/proc和/sys伪文件系统轻松查看到，所以检查容器如何配置内存的cgroup就很简单了。在容器的Pid namespace里，根进程的pid为1，但是namespace以外它呈现的是系统级pid，我们可以用来查找它的cgroups：

```
$ ps ax | grep /bin/sh
   9513 ?        Ss     0:00 /bin/sh -c while true; do sleep 2; done
$ sudo cat /proc/9513/cgroup
...
6:memory:/kubepods/burstable/podfbc202d3-da21-11e8-ab5e-42010a80014b/0a1b22ec1361a97c3511db37a4bae932d41b22264e5b9761174
```

我列出了内存cgroup，这正是我们所关注的。你在路径里可以看到前面提到的cgroup层级。一些比较重要的点是：首先，这个路径是以 `kubepods` 开始的cgroup，所以我们的进程继承了这个group的每个属性，还有 `burstable` 的属性（Kubernetes将Pod设置为burstable QoS类别）和一组用于审计的Pod表示。最后一段路径是我们进程实际使用的cgroup。我们可以把它追加到 `/sys/fs/cgroups/memory` 后面查看更多信息：

```
$ ls -l /sys/fs/cgroup/memory/kubepods/burstable/podfbc202d3-da21-11e8-ab5e-42010a80014b/0a1b22ec1361a97c3511db37a4bae93
...
-rw-r--r-- 1 root root 0 Oct 27 19:53 memory.limit_in_bytes
-rw-r--r-- 1 root root 0 Oct 27 19:53 memory.soft_limit_in_bytes
```

再一次，我只列出了我们所关心的记录。我们暂时不关注 `memory.soft_limit_in_bytes`，而将重点转移到 `memory.limit_in_bytes` 属性，它设置了内存限制。它等价于Docker命令中的 `--memory` 参数，也就是Kubernetes里的内存资源限制。我们看看：

```
$ sudo cat /sys/fs/cgroup/memory/kubepods/burstable/podfbc202d3-da21-11e8-ab5e-42010a80014b/0a1b22ec1361a97c3511db37a4ba
9223372036854771712
```

这是没有设置资源限制时我的节点上显示的情况。这里有对它的一个简单的解释(https://unix.stackexchange.com/questions/420906/what-is-the-value-for-the-cgroups-limit-in-bytes-if-the-memory-is-not-restricte)。所以我们看到如果没有在Kubernetes里设置内存限制的话，会导致Docker设置 `HostConfig.Memory` 值为0，并进一步导致容器进程被放置在默认值为"no limit"的 `memory.limit_in_bytes` 内存cgroup下。我们现在创建使用100MiB内存限制的Pod：

```
$ kubectl run limit-test --image=busybox --limits "memory=100Mi" --command -- /bin/sh -c "while true; do sleep 2; done"
deployment.apps "limit-test" created
```

我们再一次使用kubectl验证我们的资源配置：

```
$ kubectl get pods limit-test-5f5c7dc87d-8qtdx -o=jsonpath='{.spec.containers[0].resources}'
map[limits:map[memory:100Mi] requests:map[memory:100Mi]]
```

你会注意到除了我们设置的limits外，Pod还增加了requests。当你设置limits而没有设置requests时，Kubernetes默认让requests等于limits。如果你从调度器的角度看这是非常有意义的。我会在下面进一步讨论requests。当这个Pod启动后，我们可以看到Docker如何配置的容器以及这个进程的内存cgroup：

```
$ docker ps | grep busy | cut -d' ' -f1
8fec6c7b6119
$ docker inspect 8fec6c7b6119 --format '{{.HostConfig.Memory}}'
104857600
$ ps ax | grep /bin/sh
   29532 ?      Ss     0:00 /bin/sh -c while true; do sleep 2; done
$ sudo cat /proc/29532/cgroup
...
6:memory:/kubepods/burstable/pod88f89108-daf7-11e8-b1e1-42010a800070/8fec6c7b61190e74cd9f88286181dd5fa3bbf9cf33c947574eb
$ sudo cat /sys/fs/cgroup/memory/kubepods/burstable/pod88f89108-daf7-11e8-b1e1-42010a800070/8fec6c7b61190e74cd9f88286181
104857600
```

正如你所见，Docker基于我们的containerSpec正确地设置了这个进程的内存cgroup。但是这对于运行时意味着什么？Linux内存管理是一个复杂的话题，Kubernetes工程师需要知道的是：当一个宿主机遇到了内存资源压力时，内核可能会有选择性地杀死进程。如果一个使用了多于限制内存的进程会有更高几率被杀死。因为Kubernetes的任务是尽可能多地向这些节点上安排Pod，这会导致节点内存压力异常。如果你的容器使用了过多内存，那么它很可能会被oom-killed。如果Docker收到了内核的通知，Kubernetes会找到这个容器并依据设置尝试重启这个Pod。

所以Kubernetes默认创建的内存requests是什么？拥有一个100MiB的内存请求会影响到cgroup？可能它设置了我们之前看到的`memory.soft_limit_in_bytes`？让我们看看：

```
$ sudo cat /sys/fs/cgroup/memory/kubepods/burstable/pod88f89108-daf7-11e8-b1e1-42010a800070/8fec6c7b61190e74cd9f88286181
9223372036854771712
```

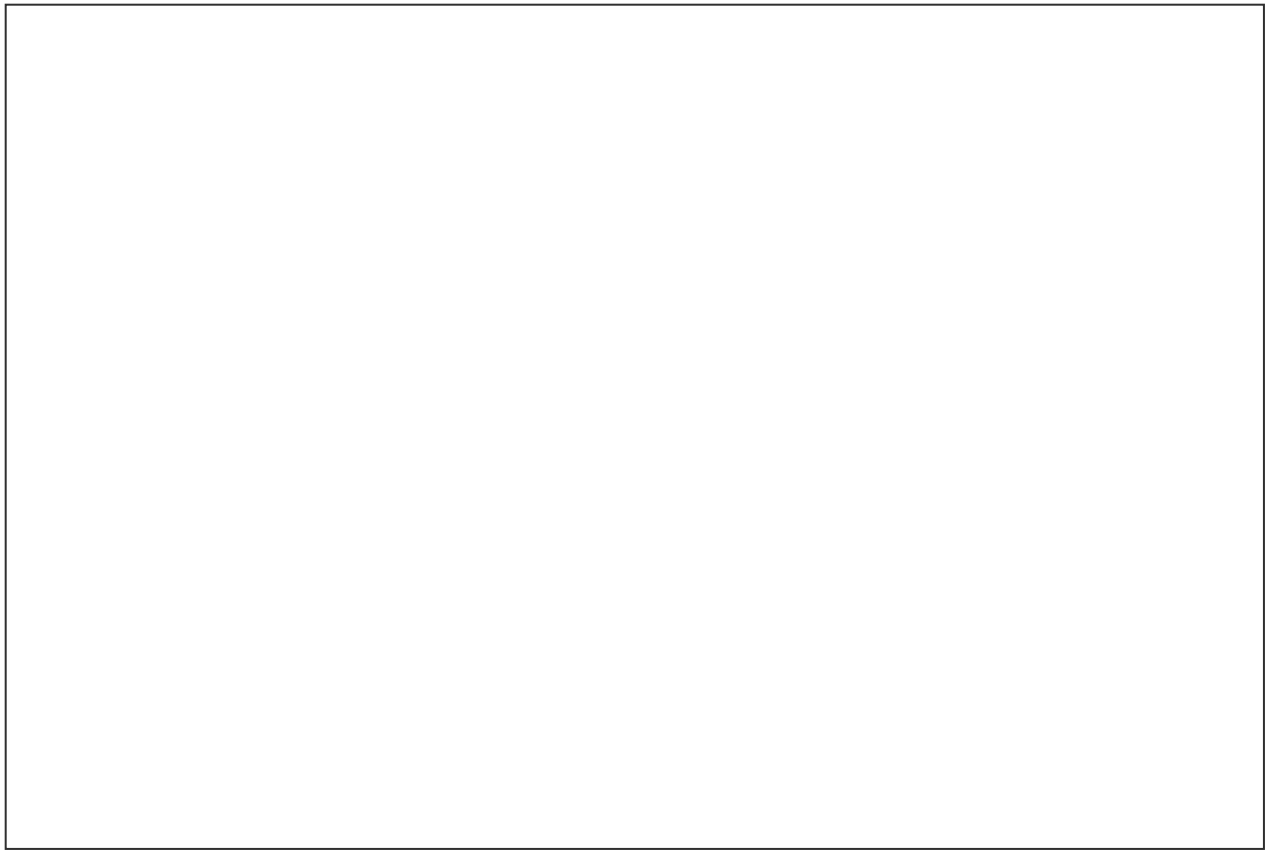
你可以看到软限制仍然被设置为默认值“no limit”。即使Docker支持通过参数`--memory-reservation`进行设置，但Kubernetes并不支持这个参数。这是否意味着为你的容器指定内存requests并不重要？不，不是的。requests要比limits更重要。limits告诉Linux内核什么时候你的进程可以为了清理空间而被杀死。requests帮助Kubernetes调度找到合适的节点运行Pod。如果不设置它们，或者设置得非常低，那么可能会有不好的影响。

例如，假设你没有配置内存requests来运行Pod，而配置了一个较高的limits。正如我们所知道的Kubernetes默认会把requests的值指向limits，如果没有合适的资源的节点的话，Pod可能会调度失败，即使它实际需要的资源并没有那么多。另一方面，如果你运行了一个配置了较低requests值的Pod，你其实是在鼓励内核oom-kill掉它。为什么？假设你的Pod通常使用100MiB内存，你却只为它配置了50MiB内存requests。如果你有一个拥有75MiB内存空间的节点，那么这个Pod会被调度到这个节点。当Pod内存消耗扩大到100MiB时，会让这个节点压力变大，这个时候内核可能会选择杀掉你的进程。所以我们要正确配置Pod的内存requests和limits。

希望这篇文章能够帮助说明Kubernetes容器内存限制是如何设置和实现的，以及为什么你需要正确设置这些值。如果你为Kubernetes提供了它所需要的足够信息，它可以智能地调度你的任务并最大化使用你的云计算资源。在下一篇博文里我们会讨论CPU限制是如何运作的，并且简单讨论如何按照namespace设置默认的requests和limits。

原文链接：https://medium.com/@betz.mark/understanding-resource-limits-in-kubernetes-memory-6b41e9a955f9

点击阅读原文即可加入，加入之后还有神秘福利等着你呦～



译者： 立尧
校对： Ghoul
编辑： 立尧

阅读原文