117Kubernetes 系列（一一零）Kubernetes 资源、容量和可分配简介

介绍

Kubernetes Pod 可以通过资源请求/限制来控制其部署和运行条件。 Request用于帮助找到资源充足的节点，Limit用于保证Pod的资源使用不超过上限。

通过使用 `kubectl describe node` ，您可以观察每个节点上的资源分配情况。与资源分配相关的有两个重要概念：容量(capacity)和可分配(allocatable)。

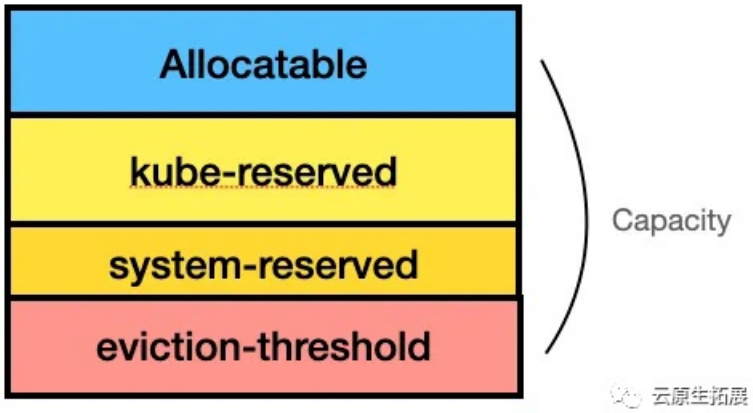
本文将探讨这两个概念的区别以及实际应用中需要注意的事项。

对于 Kubernetes 节点来说，可用资源通常包括 CPU、内存、临时存储等。节点上可用的资源总量称为“Capacity”，而可以分配给 Kubernetes Pod 的总量称为“Allocatable”。

您可以通过 `kubectl describe node` 访问此信息。例如我的虚拟机中使用kubeadm安装的默认集群信息如下：

```
Capacity:
  cpu:                2
  ephemeral-storage:  64800356Ki
  hugepages-2Mi:      0
  memory:             4039556Ki
  pods:               110
Allocatable:
  cpu:                2
  ephemeral-storage:  59720007991
  hugepages-2Mi:      0
  memory:             3937156Ki
  pods:               110
```

上述信息由各节点上的 kubelet 维护并返回。对于kubelet来说，Capacity和Allocatable的关系如下：



系统保留 （System-Reserved）

作为 Kubernetes 节点，系统本身除了运行 Kubernetes 应用的各种容器外，还会运行一些服务，比如 sshd、dhclient 等系统服务。

在CPU方面，如果节点上的所有CPU都分配给Kubernetes Pod，是否有可能没有足够的CPU来维持sshd等系统应用的基本操作？

系统保留就是针对这种情况而设计的。它的主要目的是让kubelet知道为系统相关的服务预留一些系统资源，而不是将所有节点资源分配给Pod。

- 默认值：除非指定，否则默认不启用
- 配置：kubelet通过“--system-reserved”设置不同的资源量

kube 保留 (Kube-Reserved)

Kube-Reserved 的概念与 System-Reserved 完全相同，但 Kube-Reserved 是为任何与 Kubernetes 交互的应用程序而设计的。最简单的例子是 kubelet 应用程序。

- 默认值：除非指定，否则默认不启用
- 配置：kubelet通过“--kube-reserved”设置不同的资源量

驱逐门槛 （Eviction-Thresholds）

例如，当节点上的内存使用率过高时，可能会导致 OOM 情况并导致系统上正在运行的 K8s Pod 被内核删除。

为了减少出现这种问题的可能性，kubelet 在发现节点资源不足时会开始踢出正在运行的 Pod，以便 Scheduler 尝试将 Pod 调度到另一个资源更丰富的节点上。

因此，Eviction Thresholds 是一个资源阈值。当系统资源低于此阈值时，将触发驱逐机制。

- 默认值：默认启用
  - 内存为100MiB
  - 临时存储为 10%
- 配置：kubelet通过“--eviction-hard”设置不同的资源量

```
Capacity:
  cpu:                2
  ephemeral-storage:  64800356Ki
  hugepages-2Mi:      0
  memory:             4039556Ki
  pods:               110
Allocatable:
  cpu:                2
  ephemeral-storage:  59720007991
  hugepages-2Mi:      0
  memory:             3937156Ki
  pods:               110
```

我们再回顾一下上面的资源吧！ 我们在这里分析几个重要的节点资源：

容量 = 可分配 + 系统保留 + Kube 保留 + 逐出阈值

如前所述，System-Reserved 和 Kube-Reserved 默认情况下处于禁用状态，因此:

容量 = 可分配 + 0 + 0 + 驱逐阈值

我们可以得到如下方程

驱逐阈值 = 容量 — 可分配

- CPU：在这种情况下没有区别。
- 内存：4039556Ki-3937156Ki，相差102400Ki，也就是100Mi
- Ephemeral-Storage: 64800356Ki — 59720007991 Byte，首先将前者乘以 12 转换为 Byte，得到 66355564544，因此 59720007991/66355564544 约为 0.8999999851，即 0.9 因此，实际可分配的容量只有90%。

注意：Mi、Ki、Gi 的基数为 1024，而不是 1000

上述值与我们在驱逐阈值部分中提到的默认值完全相同。

实验

将以下内容添加到 /var/lib/kubelet/config.yaml，针对 Kube-Reserved、System-Reserved 和 evictionHard 参数：

注意：Eviction Threshold 设置实际上是通过 evictionHard 参数配置的。

```
systemReserved:
  memory: 500Mi
  cpu: 250m
kubeReserved:
  memory: 1Gi
  cpu: 500m
evictionHard:
  memory.available: 200Mi
  nodefs.available: 20Gi
```

计算我们的配置将使用多少系统资源，如下表所示。注意Memory有Gi和Mi，1Gi就是1024 Mi，所以总量是1724 Mi。

```
| Type         | System   | Reserved | Evictionhard | Total |
| -------------| ------------------ |------------ |-------|
| CPU          | 250m     | 500m     | 0            | 750m  |
| Memory       | 500Mi    | 1Gi      | 200Mi        | 1724Mi|
| Storage      | 0        | 0        | 20Gi         | 20Gi  |
```

并使用 sudo systemctl restart kubelet 重新启动 kubelet 以加载新设置。一切完成后，使用kubectl描述节点来观察变化。

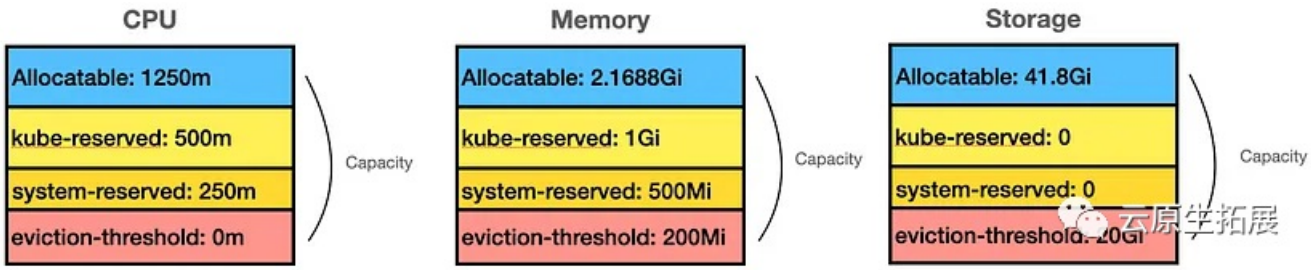
```
Capacity:
  cpu:                2
  ephemeral-storage:  64800356Ki
  hugepages-2Mi:      0
  memory:             4039556Ki
  pods:               110
Allocatable:
  cpu:                1250m
  ephemeral-storage:  43828836Ki
  hugepages-2Mi:      0
  memory:             2274180Ki
  pods:               110
```

现在我们再计算一下这个值

系统保留 + Kube 保留 + 逐出阈值 = 容量 - 可分配

- CPU：2 个 2000m 单元，因此 2000m–1250m = 750m。
- 内存：4039556Ki-2274180Ki = 1765376Ki，则1765376Ki/1024 = 1724 Mi。
- 存储：64800356Ki-43828836Ki = 20971520Ki，则 20971520Ki/1024/1024 = 20 Gi

计算结果与预期相符。下图总结了上述概念。



强制节点可分配

了解了Capacity和Allocatable的基本概念和计算方法后，下一步就是了解更详细的应用控制。

实际上，system-reserved 和 kube-reserved 参数的含义是“根据 system-reserved 和 kube-reserved 的参数请求 kubelet 预留系统资源，以防止 Kubernetes Pod 占用过多资源”。

现在，我们提出几个问题来思考：

- 如果系统应用程序或 Kubernetes 相关应用程序使用的系统资源多于设置的（system-reserved、kube-reserved）参数，会发生什么情况？
- 什么类型的应用程序属于系统保留？
- 什么类型的应用程序属于 kube-reserved？

- 自主开发的应用程序是否可以添加到其中一个类别中？

默认配置下，上述问题的答案是：

- 什么都不会发生，也不会有任何后果。
- 因为超过限度不会产生任何后果，所以对任何人进行分类是没有意义的。
- 因为超过限制不会产生任何后果，所以管理自己的应用程序是没有意义的。

如果您希望 system-reserverd 或 kube-systemted 应用程序在使用过多时被删除，您应该怎么做？

此时就需要用到kubelet的另一个参数--enforce-node-allocatable，它有三个可以组合的参数，分别是pods、system-reserved、kube-reserved。

该参数的含义是“哪种类型的资源超过使用量，需要被系统杀死”。

默认值是Pods，这就是为什么Pod如果设置了request/limit，超过了limit，可能会触发OOM，直接被系统kill掉。默认情况下，system-reserved 和 kube-reserved 没有设置，所以超出限制使用也没有问题。

在实现中，kubelet 不参与任何监控或删除应用程序的决策。它只是依靠cgroup（控制组）来根据设置处理所有事情，并让内核帮助处理。

因此，如果你阅读文档，它会告诉你，如果你想在enforce-node-allocatable中设置system-reserved和kube-reserved，你还必须设置参数--kube-reserved-cgroup和--system-保留-cgroup。

将相关配置添加到 /var/lib/kubelet/config.yaml 中，如下所示。

```
Capacity:
  cpu:                2
  ephemeral-storage:  64800356Ki
  hugepages-2Mi:      0
  memory:             4039556Ki
  pods:               110
Allocatable:
  cpu:                1250m
  ephemeral-storage:  43828836Ki
  hugepages-2Mi:      0
  memory:             2274180Ki
  pods:               110
systemReservedCgroup: /system.slice
enforceNodeAllocatable:
  - pods
  - system-reserved
```

上面的示例指示 kubelet 对属于系统保留组的应用程序强制实施资源限制，该组定义为 cgroup 路径 /system.slice 下的所有应用程序。

在我的Ubuntu 18.04环境中，就CPU使用率而言，/system.slice下存在以下应用程序。

```
○ → lscgroup cpu:/system.slice
cpu,cpuacct:/system.slice/
cpu,cpuacct:/system.slice/irqbalance.service
cpu,cpuacct:/system.slice/systemd-update-utmp.service
cpu,cpuacct:/system.slice/vboxadd-service.service
cpu,cpuacct:/system.slice/lvm2-monitor.service
cpu,cpuacct:/system.slice/systemd-journal-flush.service
cpu,cpuacct:/system.slice/containerd.service
cpu,cpuacct:/system.slice/systemd-sysctl.service
cpu,cpuacct:/system.slice/systemd-networkd.service
cpu,cpuacct:/system.slice/systemd-udevd.service
cpu,cpuacct:/system.slice/lxd-containers.service
cpu,cpuacct:/system.slice/cron.service
cpu,cpuacct:/system.slice/sys-fs-fuse-connections.mount
cpu,cpuacct:/system.slice/networking.service
cpu,cpuacct:/system.slice/sys-kernel-config.mount
cpu,cpuacct:/system.slice/docker.service
cpu,cpuacct:/system.slice/polkit.service
cpu,cpuacct:/system.slice/systemd-remount-fs.service
cpu,cpuacct:/system.slice/networkd-dispatcher.service
cpu,cpuacct:/system.slice/sys-kernel-debug.mount
cpu,cpuacct:/system.slice/accounts-daemon.service
cpu,cpuacct:/system.slice/systemd-tmpfiles-setup.service
cpu,cpuacct:/system.slice/kubelet.service
cpu,cpuacct:/system.slice/console-setup.service
cpu,cpuacct:/system.slice/vboxadd.service
cpu,cpuacct:/system.slice/systemd-journald.service
cpu,cpuacct:/system.slice/atd.service
cpu,cpuacct:/system.slice/systemd-udev-trigger.service
cpu,cpuacct:/system.slice/lxd.socket
cpu,cpuacct:/system.slice/ssh.service
cpu,cpuacct:/system.slice/dev-mqueue.mount
cpu,cpuacct:/system.slice/ufw.service
cpu,cpuacct:/system.slice/systemd-random-seed.service
cpu,cpuacct:/system.slice/snapd.seeded.service
cpu,cpuacct:/system.slice/rsyslog.service
cpu,cpuacct:/system.slice/systemd-modules-load.service
cpu,cpuacct:/system.slice/blk-availability.service
cpu,cpuacct:/system.slice/systemd-tmpfiles-setup-dev.service
cpu,cpuacct:/system.slice/rpcbind.service
cpu,cpuacct:/system.slice/lxcfs.service
cpu,cpuacct:/system.slice/grub-common.service
cpu,cpuacct:/system.slice/ebtables.service
cpu,cpuacct:/system.slice/snapd.socket
cpu,cpuacct:/system.slice/kmod-static-nodes.service
cpu,cpuacct:/system.slice/run-rpc_pipefs.mount
cpu,cpuacct:/system.slice/lvm2-lvmetad.service
cpu,cpuacct:/system.slice/docker.socket
cpu,cpuacct:/system.slice/apport.service
cpu,cpuacct:/system.slice/apparmor.service
cpu,cpuacct:/system.slice/systemd-resolved.service
cpu,cpuacct:/system.slice/system-lvm2\x2dpvscan.slice
cpu,cpuacct:/system.slice/dev-hugepages.mount
cpu,cpuacct:/system.slice/dbus.service
cpu,cpuacct:/system.slice/system-getty.slice
cpu,cpuacct:/system.slice/keyboard-setup.service
cpu,cpuacct:/system.slice/systemd-user-sessions.service
cpu,cpuacct:/system.slice/systemd-logind.service
cpu,cpuacct:/system.slice/setvtrgb.service
```

从上面的文件名可以看出，有很多系统服务。

修改完kubelet后，重启就会发现kubelet启动失败。您将在日志中收到一条错误消息，指示 /system.slice 路径不存在。

kubelet.go:1391] “Failed to start ContainerManager” err=”Failed to enforce System Reserved Cgroup Limits on “/system.slice”: [“system.slice”] cgroup does not exist

其实这个问题是 kubelet 试图从很多 cgroup 子系统中查找，只要有一个不存在，就视为错误。根据下面的源代码。

```go
func (m *cgroupManagerImpl) Exists(name CgroupName) bool {
 if libcontainercgroups.IsCgroup2UnifiedMode() {
  cgroupPath := m.buildCgroupUnifiedPath(name)
  neededControllers := getSupportedUnifiedControllers()
  enabledControllers, err := readUnifiedControllers(cgroupPath)
  if err != nil {
   return false
  }
  difference := neededControllers.Difference(enabledControllers)
  if difference.Len() > 0 {
   klog.V(4).InfoS("The cgroup has some missing controllers", "cgroupName", name, "controllers", difference)
   return false
  }
  return true
 }

 // Get map of all cgroup paths on the system for the particular cgroup
 cgroupPaths := m.buildCgroupPaths(name)

 // the presence of alternative control groups not known to runc confuses
 // the kubelet existence checks.
 // ideally, we would have a mechanism in runc to support Exists() logic
 // scoped to the set control groups it understands.  this is being discussed
 // in https://github.com/opencontainers/runc/issues/1440
 // once resolved, we can remove this code.
 allowlistControllers := sets.NewString("cpu", "cpuacct", "cpuset", "memory", "systemd", "pids")

 if _, ok := m.subsystems.MountPoints["hugetlb"]; ok {
  allowlistControllers.Insert("hugetlb")
 }
 var missingPaths []string
 // If even one cgroup path doesn't exist, then the cgroup doesn't exist.
 for controller, path := range cgroupPaths {
  // ignore mounts we don't care about
  if !allowlistControllers.Has(controller) {
   continue
  }
  if !libcontainercgroups.PathExists(path) {
   missingPaths = append(missingPaths, path)
  }
 }

 if len(missingPaths) > 0 {
  klog.V(4).InfoS("The cgroup has some missing paths", "cgroupName", name, "paths", missingPaths)
  return false
 }

 return true
}
```

kubelet 将搜索我系统中的 cpu、cpuacct、cpuset、内存、systemd、pids 和 Hugetlb 子系统，但不幸的是 cpuset、hugetlb 和 systemd 子系统不包含 /system.slice 路径。

因此，如果要使用该参数进行控制，需要正确配置目标cgroup路径，才能成功启动kubelet。

```
o → mkdir -p /sys/fs/cgroup/hugetlb/system.slice
o → mkdir -p /sys/fs/cgroup/cpuset/system.slice
o → mkdir -p /sys/fs/cgroup/systemd/system.slice
o → systemctl restart kubelet
```

一切执行成功后，我们可以使用cgroup命令来检查系统保留的资源是否已设置到相应的cgroup中。让我们回顾一下之前的表格：

```
| Type         | System   | Reserved | Evictionhard | Total |
| -------------| ------------------ |------------ |-------|
| CPU          | 250m     | 500m     | 0            | 750m  |
| Memory       | 500Mi    | 1Gi      | 200Mi        | 1724Mi|
| Storage      | 0        | 0        | 20Gi         | 20Gi  |
```

SystemReserved CPU为250m，内存为500Mi。

```
o → cat /sys/fs/cgroup/cpu/system.slice/cpu.shares
256
o → cat /sys/fs/cgroup/memory/system.slice/memory.limit_in_bytes
524288000
```

这里我们可以看到CPU是256，以1024为单位计算的话，就是25%，也就是250m。内存的单位是字节，524288000/1024/1024 = 500Mi。

总结

说完以上，我们回到三个问题：

- 如果系统应用程序或 Kubernetes 相关应用程序使用的系统资源多于集合（system-reserved、kube-reserved），会发生什么？
- 什么样的应用程序属于系统保留？哪些类型的应用程序属于 kube-reserved？
- 自主开发的应用程序可以添加到这些类别之一吗？

答案是：

- 这取决于你是否使用enforce-node-allocatable来要求kubelet帮助内核cgroup控制资源使用。
- 使用 system-reserved-cgroup 和 kube-reserved-cgroup 参数指定 cgroup 路径。
- 将您的应用程序添加到相应的cgroup组中，并记住不同的资源有不同的路径。

最后，如果您对这些设置感兴趣，并且相信它们能够更好地控制系统资源的使用，请务必进行测试并确保您了解cgroup的所有概念，以免以后调试时完全无能为力。

另外，对于system-reserved、kube-reserved等应用的系统使用情况，请先用监控系统长时间观察，有个概念后再进行设置。同时，如果启用这两个设置，请做好这些应用程序可能因OOM而被删除的心理准备，以免以后出现问题时毫无头绪。