# 40 | Kubernetes的资源模型与资源管理

# 40 | Kubernetes的资源模型与资源管理

张磊 2018-11-23

10:25

讲述：张磊 大小：4.78M

你好，我是张磊。今天我和你分享的主题是：Kubernetes 的资源模型与资源管理。

作为一个容器集群编排与管理项目，Kubernetes 为用户提供的基础设施能力，不仅包括了我在前面为你讲述的应用定义和描述的部分，还包括了对应用的资源管理和调度的处理。那么，从今天这篇文章开始，我就来为你详细讲解一下后面这部分内容。

而作为 Kubernetes 的资源管理与调度部分的基础，我们要从它的资源模型开始说起。

我在前面的文章中已经提到过，在 Kubernetes 里，Pod 是最小的原子调度单位。这也就意味着，所有跟调度和资源管理相关的属性都应该是属于 Pod 对象的字段。而这其中最重要的部分，就是 Pod 的 CPU 和内存配置，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
```

```
    name: frontend
    spec:
    containers:
    - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
    value: "password"
    resources:
    requests:
    memory: "64Mi"
    cpu: "250m"
    limits:
    memory: "128Mi"
    cpu: "500m"
    - name: wp
    image: wordpress
    resources:
    requests:
    memory: "64Mi"
    cpu: "250m"
    limits:
    memory: "128Mi"
    cpu: "500m"
```

□复制代码

> *备注：关于哪些属性属于 Pod 对象，而哪些属性属于 Container，你可以在回顾一下第 14 篇文章《深入解析 Pod 对象（一）：基本概念》中的相关内容。*

在 Kubernetes 中，像 CPU 这样的资源被称作“可压缩资源”（compressible resources）。它的典型特点是，当可压缩资源不足时，Pod 只会“饥饿”，但不会退出。

而像内存这样的资源，则被称作“不可压缩资源（incompressible resources）。当不可压缩资源不足时，Pod 就会因为 OOM（Out-Of-Memory）被内核杀掉。

而由于 Pod 可以由多个 Container 组成，所以 CPU 和内存资源的限额，是要配置在每个 Container 的定义上的。这样，Pod 整体的资源配置，就由这些 Container 的配置值累加得到。

其中，Kubernetes 里为 CPU 设置的单位是“CPU 的个数”。比如，cpu=1 指的就是，这个 Pod 的 CPU 限额是 1 个 CPU。当然，具体“1 个 CPU”在宿主机上如何解释，是 1 个 CPU 核心，还是 1 个 vCPU，还是 1 个 CPU 的超线程（Hyperthread），完全取决于宿主机的 CPU 实现方式。Kubernetes 只负责保证 Pod 能够使用到“1 个 CPU”的计算能力。

此外，Kubernetes 允许你将 CPU 限额设置为分数，比如在我们的例子里，CPU limits 的值就是 500m。所谓 500m，指的就是 500 millicpu，也就是 0.5 个 CPU 的意思。这样，这个 Pod 就会被分配到 1 个 CPU 一半的计算能力。

当然，**你也可以直接把这个配置写成 cpu=0.5。但在实际使用时，我还是推荐你使用 500m 的写法，毕竟这才是 Kubernetes 内部通用的 CPU 表示方式。**

而对于内存资源来说，它的单位自然就是 bytes。Kubernetes 支持你使用 Ei、Pi、Ti、Gi、Mi、Ki（或者 E、P、T、G、M、K）的方式来作为 bytes 的值。比如，在我们的例子里，Memory requests 的值就是 64MiB (2 的 26 次方 bytes)。这里要注意区分 MiB（mebibyte）和 MB（megabyte）的区别。

> *备注：1Mi=1024*1024；1M=1000*1000*

此外，不难看到，**Kubernetes 里 Pod 的 CPU 和内存资源，实际上还要分为 limits 和 requests 两种情况**，如下所示：

```
spec.containers[].resources.limits.cpu
spec.containers[].resources.limits.memory
spec.containers[].resources.requests.cpu
spec.containers[].resources.requests.memory
```

复制代码

这两者的区别其实非常简单：在调度的时候，kube-scheduler 只会按照 requests 的值进行计算。而在真正设置 Cgroups 限制的时候，kubelet 则会按照 limits 的值来进行设置。

更确切地说，当你指定了 requests.cpu=250m 之后，相当于将 Cgroups 的 cpu.shares 的值设置为 (250/1000)*1024。而当你没有设置 requests.cpu 的时候，cpu.shares 默认则是 1024。这样，Kubernetes 就通过 cpu.shares 完成了对 CPU 时间的按比例分配。

而如果你指定了 limits.cpu=500m 之后，则相当于将 Cgroups 的 cpu.cfs_quota_us 的值设置为 (500/1000)*100ms，而 cpu.cfs_period_us 的值始终是 100ms。这样，Kubernetes 就为你设置了这个容器只能用到 CPU 的 50%。

而对于内存来说，当你指定了 limits.memory=128Mi 之后，相当于将 Cgroups 的 memory.limit_in_bytes 设置为 128 * 1024 * 1024。而需要注意的是，在调度的时候，调度器只会使用 requests.memory=64Mi 来进行判断。

**Kubernetes 这种对 CPU 和内存资源限额的设计，实际上参考了 Borg 论文中对“动态资源边界”的定义**，既：容器化作业在提交时所设置的资源边界，并不一

定是调度系统所必须严格遵守的，这是因为在实际场景中，大多数作业使用到的资源其实远小于它所请求的资源限额。

基于这种假设，Borg 在作业被提交后，会主动减小它的资源限额配置，以便容纳更多的作业、提升资源利用率。而当作业资源使用量增加到一定阈值时，Borg 会通过“快速恢复”过程，还原作业原始的资源限额，防止出现异常情况。

而 Kubernetes 的 requests+limits 的做法，其实就是上述思路的一个简化版：用户在提交 Pod 时，可以声明一个相对较小的 requests 值供调度器使用，而 Kubernetes 真正设置给容器 Cgroups 的，则是相对较大的 limits 值。不难看到，这跟 Borg 的思路相通的。

在理解了 Kubernetes 资源模型的设计之后，我再来和你谈谈 Kubernetes 里的 QoS 模型。在 Kubernetes 中，不同的 requests 和 limits 的设置方式，其实会将这个 Pod 划分到不同的 QoS 级别当中。

**当 Pod 里的每一个 Container 都同时设置了 requests 和 limits，并且 requests 和 limits 值相等的时候，这个 Pod 就属于 Guaranteed 类别**，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
name: qos-demo
namespace: qos-example
spec:
containers:
- name: qos-demo-ctr
image: nginx
resources:
limits:
memory: "200Mi"
cpu: "700m"
requests:
memory: "200Mi"
cpu: "700m"
```

复制代码

当这个 Pod 创建之后，它的 qosClass 字段就会被 Kubernetes 自动设置为 Guaranteed。需要注意的是，当 Pod 仅设置了 limits 没有设置 requests 的时候，Kubernetes 会自动为它设置与 limits 相同的 requests 值，所以，这也属于 Guaranteed 情况。

**而当 Pod 不满足 Guaranteed 的条件，但至少有一个 Container 设置了 requests。那么这个 Pod 就会被划分到 Burstable 类别。**比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
name: qos-demo-2
namespace: qos-example
spec:
containers:
- name: qos-demo-2-ctr
image: nginx
resources:
limits
memory: "200Mi"
requests:
memory: "100Mi"
```

复制代码

**而如果一个 Pod 既没有设置 requests，也没有设置 limits，那么它的 QoS 类别就是 BestEffort。**比如下面这个例子：

```
apiVersion: v1
kind: Pod
metadata:
name: qos-demo-3
namespace: qos-example
spec:
containers:
- name: qos-demo-3-ctr
image: nginx
```

复制代码

那么，Kubernetes 为 Pod 设置这样三种 QoS 类别，具体有什么作用呢？

实际上，**QoS 划分的主要应用场景，是当宿主机资源紧张的时候，kubelet 对 Pod 进行 Eviction（即资源回收）时需要用到的。**

具体地说，当 Kubernetes 所管理的宿主机上不可压缩资源短缺时，就有可能触发 Eviction。比如，可用内存（memory.available）、可用的宿主机磁盘空间（nodefs.available），以及容器运行时镜像存储空间（imagefs.available）等等。

目前，Kubernetes 为你设置的 Eviction 的默认阈值如下所示：

```
memory.available<100Mi
nodefs.available<10%
```

```
nodefs.inodesFree<5%

imagefs.available<15%
```

□复制代码

当然，上述各个触发条件在 kubelet 里都是可配置的。比如下面这个例子：

```
kubelet --eviction-
hard=imagefs.available<10%,memory.available<500Mi,nodefs.available<5%,nodefs.inodesFr
--eviction-soft=imagefs.available<30%,nodefs.available<10% --eviction-soft-grace-
period=imagefs.available=2m,nodefs.available=2m --eviction-max-pod-grace-period=600
```

□复制代码

在这个配置中，你可以看到**Eviction 在 Kubernetes 里其实分为 Soft 和 Hard 两种模式。**

其中，Soft Eviction 允许你为 Eviction 过程设置一段“优雅时间”，比如上面例子里的 imagefs.available=2m，就意味着当 imagefs 不足的阈值达到 2 分钟之后，kubelet 才会开始 Eviction 的过程。

而 Hard Eviction 模式下，Eviction 过程就会在阈值达到之后立刻开始。

> *Kubernetes 计算 Eviction 阈值的数据来源，主要依赖于从 Cgroups 读取到的值，以及使用 cAdvisor 监控到的数据。*

当宿主机的 Eviction 阈值达到后，就会进入 MemoryPressure 或者 DiskPressure 状态，从而避免新的 Pod 被调度到这台宿主机上。

而当 Eviction 发生的时候，kubelet 具体会挑选哪些 Pod 进行删除操作，就需要参考这些 Pod 的 QoS 类别了。

首当其冲的，自然是 BestEffort 类别的 Pod。
其次，是属于 Burstable 类别、并且发生“饥饿”的资源使用量已经超出了 requests 的 Pod。
最后，才是 Guaranteed 类别。并且，Kubernetes 会保证只有当 Guaranteed 类别的 Pod 的资源使用量超过了其 limits 的限制，或者宿主机本身正处于 Memory Pressure 状态时，Guaranteed 的 Pod 才可能被选中进行 Eviction 操作。

当然，对于同 QoS 类别的 Pod 来说，Kubernetes 还会根据 Pod 的优先级来进行进一步地排序和选择。

在理解了 Kubernetes 里的 QoS 类别的设计之后，我再来为你讲解一下 Kubernetes 里一个非常有用的特性：cpuset 的设置。

我们知道，在使用容器的时候，你可以通过设置 cpuset 把容器绑定到某个 CPU 的核上，而不是像 cpushare 那样共享 CPU 的计算能力。

这种情况下，由于操作系统在 CPU 之间进行上下文切换的次数大大减少，容器里应用的性能会得到大幅提升。事实上，**cpuset 方式，是生产环境里部署在线应用类型的 Pod 时，非常常用的一种方式。**

可是，这样的需求在 Kubernetes 里又该如何实现呢？

其实非常简单。

> 首先，你的 Pod 必须是 Guaranteed 的 QoS 类型；
> 然后，你只需要将 Pod 的 CPU 资源的 requests 和 limits 设置为同一个相等的整数值即可。

比如下面这个例子：

```
spec:
containers:
- name: nginx
image: nginx
resources:
limits:
memory: "200Mi"
cpu: "2"
requests:
memory: "200Mi"
cpu: "2"
```

□复制代码

这时候，该 Pod 就会被绑定在 2 个独占的 CPU 核上。当然，具体是哪两个 CPU 核，是由 kubelet 为你分配的。

以上，就是 Kubernetes 的资源模型和 QoS 类别相关的主要内容。

# 总结

在本篇文章中，我先为你详细讲解了 Kubernetes 里对资源的定义方式和资源模型的设计。然后，我为你讲述了 Kubernetes 里对 Pod 进行 Eviction 的具体策略和实践方式。

正是基于上述讲述，在实际的使用中，我强烈建议你将 DaemonSet 的 Pod 都设置为 Guaranteed 的 QoS 类型。否则，一旦 DaemonSet 的 Pod 被回收，它又会立即在原宿主机上被重建出来，这就使得前面资源回收的动作，完全没有意义了。

# 思考题

为什么宿主机进入 MemoryPressure 或者 DiskPressure 状态后，新的 Pod 就不会被调度到这台宿主机上呢?