# Kubernetes Node资源预留

Original YP小站 [YP小站](https://mp.weixin.qq.com/javascript:void(0);) 2020-01-10

<https://mp.weixin.qq.com/s?__biz=MzA4MzIwNTc4NQ%3D%3D&chksm=9ffb4923a88cc0355b57005ab7b8deef658c750114445fc7852a0048a9ab8b1811b4cf39e31e&idx=1&mid=2247484117&scene=21&sn=21bafae148363e3c269a6ed3d79767a1#wechat_redirect>

<https://blog.csdn.net/qq_24794401/article/details/106626042>

## **Node 为什么需要资源预留?**

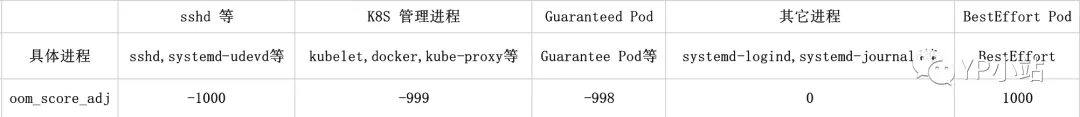
Kubernetes 的节点可以按照 Capacity 调度。默认情况下 pod 能够使用节点全部可用容量。这是个问题，因为节点自己通常运行了不少驱动 OS 和 Kubernetes 的系统守护进程（system daemons）。除非为这些系统守护进程留出资源，否则它们将与 pod 争夺资源并导致节点资源短缺问题。

按照是否为 Pod，可以把计算节点的进程分为两类：

* Pod 类进程：容器内部的进程，这些容器由 K8S 创建
* 非 Pod 类进程：系统进程，如内核，systemd 等；K8S 管理进程，如 Docker, Kubelet, Kube-proxy 等

如果没有资源预留，K8S 认为宿主机上所有的资源(RAM, CPU)都是可以分配给 Pod 类进程。因为非 Pod 类进程也需要占用一定的资源，当 Pod 创建很多时，就有可能出现资源不足的情况。宿主机中 kubelet、kube-proxy等进程被kill掉，最后导致整个 Node 节点不可用。

我们知道，当 Pod 里面内存不足时，会触发 Cgroup 把 Pod 里面的进程杀死；当系统内存不足时，就有可能触发系统 OOM，这时候根据 oom score 来确定优先杀死哪个进程，而 oom\_score\_adj 又是影响 oom score 的重要参数，其值越低，表示 oom 的优先级越低。在计算节点中，进程的 oom\_score\_adj 如下：



所以，很大概率上，OOM 的优先级如下：

BestEffort Pod > 其它进程 > Guarantee Pod > kubelet,docker,kube-proxy等 > sshd 等

那么问题来了，如果节点没有 BestEffort 类型的 pod，那么其它进程就有可能被 OOM，包括系统进程等，后果可想而知。所以，预留一定的资源给系统和 K8S 管理服务，非常有必要。

## **预留多少资源**

K8S 1.5 支持 CPU 和 RAM 两种资源的预留，更高版本支持 Disk 资源的预留。

以下参考设置是个人建议

* CPU：作为可压缩资源，超配的后果是运行变慢，影响较小，为了充分发挥节点性能，CPU 不预留
* RAM：8GB < RAM <= 16GB 预留 3GB; RAM > 16G 预留 4GB; (以上机器只跑k8s服务，没有额外应用服务)
* Disk：磁盘可预留 5% 至 10% 左右

## **如何预留**

kubelet 公开了一个名为 Node Allocatable 的特性，有助于为系统守护进程预留计算资源。Kubernetes 推荐集群管理员按照每个节点上的工作负载密度配置 Node Allocatable。

Node Allocatable（节点可分配资源）：

Node Capacity---------------------------| kube-reserved ||-------------------------|| system-reserved ||-------------------------|| eviction-threshold ||-------------------------|| || allocatable || (available for pods) || || |---------------------------

Kubernetes 节点上的 Allocatable 被定义为 pod 可用计算资源量。调度器不会超额申请 Allocatable。目前支持 CPU, memory 和 storage 这几个参数。

Node Allocatable 暴露为 API 中 v1.Node 对象的一部分，也是 CLI 中 kubectl describe node 的一部分。

在 kubelet 中，可以为两类系统守护进程预留资源。

启用 QoS 和 Pod 级别的 cgroups

为了恰当的在节点范围实施 node allocatable，您必须通过 --cgroups-per-qos 标志启用新的 cgroup 层次结构。这个标志是默认启用的。启用后，kubelet 将在其管理的 cgroup 层次结构中创建所有终端用户的 pod。

配置 cgroup 驱动

kubelet 支持在主机上使用 cgroup 驱动操作 cgroup 层次结构。驱动通过 --cgroup-driver 标志配置。

支持的参数值如下：

取决于相关容器运行时（container runtime）的配置，操作员可能需要选择一个特定的 cgroup 驱动来保证系统正常运行。例如如果操作员使用 docker 运行时提供的 cgroup 驱动时，必须配置 kubelet 使用 systemd cgroup 驱动。

* + cgroupfs 是默认的驱动，在主机上直接操作 cgroup 文件系统以对 cgroup 沙箱进行管理。
  + systemd 是可选的驱动，使用 init 系统支持的资源的瞬时切片管理 cgroup 沙箱。

K8S 把计算节点资源分为 4 个部分：

* Kube Reserved：预留给 K8S 管理进程的资源，如 Kubelet，Docker Daemon 等
* System Reserved：预留给系统资源，如 kernel，sshd，udev 等
* Eviction Thresholds：驱逐（Eviction）的阈值，只支持 memory 和 storage。
* Allocatable(available for pods)：pods 可以使用的资源

官方文档示例：

这是一个用于说明节点分配计算方式的示例：

* 节点拥有 32Gi 内存，16 核 CPU 和 100Gi 存储
* --kube-reserved 设置为 cpu=1,memory=2Gi,ephemeral-storage=1Gi
* --system-reserved 设置为 cpu=500m,memory=1Gi,ephemeral-storage=1Gi
* --eviction-hard 设置为 memory.available<500Mi,nodefs.available<10%

在这个场景下，Allocatable 将会是 14.5 CPUs、28.5Gi 内存以及 88Gi 本地存储。调度器保证这个节点上的所有 pod 请求的内存总量不超过 28.5Gi，存储不超过 88Gi。当 pod 的内存使用总量超过 28.5Gi或者磁盘使用总量超过 88Gi 时，Kubelet 将会驱逐它们。如果节点上的所有进程都尽可能多的使用 CPU，则 pod 加起来不能使用超过 14.5 CPUs 的资源。

当没有执行 kube-reserved 或 system-reserved 且系统守护进程使用量超过其预留时，如果节点内存用量高于 31.5Gi 或存储大于 90Gi，kubelet 将会驱逐 pod。

小结：为了简化管理，建议不对 kube-reserved/system-reserved 做区分，直接使用 --system-reserved做系统预留。在 kubelet 配置文件中设置如下参数：

--system-reserved=memory=3Gi,storage=5Gi \--eviction-hard=memory.available<1Gi,nodefs.available<10Gi,imagefs.available<15Gi \--eviction-soft=memory.available<1.5Gi,nodefs.available<15Gi,imagefs.available<20Gi \--eviction-soft-grace-period=memory.available=2m,nodefs.available=2m,imagefs.available=2m \--eviction-max-pod-grace-period=30 \--eviction-minimum-reclaim=memory.available=200Mi,nodefs.available=5Gi,imagefs.available=5Gi \

使用 kubectl describe nodes node-name 查询设置效果。

例如下面效果：

Capacity: cpu: 12 ephemeral-storage: 2683368944Ki hugepages-1Gi: 0 hugepages-2Mi: 0 memory: 65701752Ki pods: 110Allocatable: cpu: 12 ephemeral-storage: 2262397424Ki hugepages-1Gi: 0 hugepages-2Mi: 0 memory: 55215992Ki pods: 110

## 其它

Kubelet 还支持使用 cgroup 从物理上限制预留资源，这需要给 K8S 管理进程和系统进程分别设置 cgroup，复杂度相对较高，并且不好维护，个人不推荐这种方法。

## 参考链接

* https://kubernetes.io/zh/docs/tasks/administer-cluster/out-of-resource/#
* https://kubernetes.io/zh/docs/tasks/administer-cluster/reserve-compute-resources/
* http://wsfdl.com/kubernetes/2017/12/18/k8s%E8%B5%84%E6%BA%90%E9%A2%84%E7%95%99.html

# [重平衡工具Descheduler](https://cloud.tencent.com/developer/article/javascript:;" \o "重平衡工具Descheduler)

<https://cloud.tencent.com/developer/article/1671811>

<https://blog.csdn.net/qq_24794401/article/details/106626079>

<https://github.com/kubernetes-sigs/descheduler>

# [k8s节点资源预留与 pod 驱逐](https://segmentfault.com/a/1190000021402192)

<https://segmentfault.com/a/1190000021402192>

# Kubernetes 服务部署最佳实践(一) 如何合理利用资源

**Roc <https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/>**

Jun 16, 2020  2 分钟阅读时长

目录

* [引言](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%BC%95%E8%A8%80)
* [Request 与 Limit 怎么设置才好](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "request-%E4%B8%8E-limit-%E6%80%8E%E4%B9%88%E8%AE%BE%E7%BD%AE%E6%89%8D%E5%A5%BD)
  + [所有容器都应该设置 request](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E6%89%80%E6%9C%89%E5%AE%B9%E5%99%A8%E9%83%BD%E5%BA%94%E8%AF%A5%E8%AE%BE%E7%BD%AE-request)
  + [老是忘记设置怎么办](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E8%80%81%E6%98%AF%E5%BF%98%E8%AE%B0%E8%AE%BE%E7%BD%AE%E6%80%8E%E4%B9%88%E5%8A%9E)
  + [重要的线上应用改如何设置](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E9%87%8D%E8%A6%81%E7%9A%84%E7%BA%BF%E4%B8%8A%E5%BA%94%E7%94%A8%E6%94%B9%E5%A6%82%E4%BD%95%E8%AE%BE%E7%BD%AE)
  + [怎样设置才能提高资源利用率](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E6%80%8E%E6%A0%B7%E8%AE%BE%E7%BD%AE%E6%89%8D%E8%83%BD%E6%8F%90%E9%AB%98%E8%B5%84%E6%BA%90%E5%88%A9%E7%94%A8%E7%8E%87)
  + [尽量避免使用过大的 request 与 limit](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%B0%BD%E9%87%8F%E9%81%BF%E5%85%8D%E4%BD%BF%E7%94%A8%E8%BF%87%E5%A4%A7%E7%9A%84-request-%E4%B8%8E-limit)
  + [避免测试 namespace 消耗过多资源影响生产业务](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E9%81%BF%E5%85%8D%E6%B5%8B%E8%AF%95-namespace-%E6%B6%88%E8%80%97%E8%BF%87%E5%A4%9A%E8%B5%84%E6%BA%90%E5%BD%B1%E5%93%8D%E7%94%9F%E4%BA%A7%E4%B8%9A%E5%8A%A1)
* [如何让资源得到更合理的分配](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%A6%82%E4%BD%95%E8%AE%A9%E8%B5%84%E6%BA%90%E5%BE%97%E5%88%B0%E6%9B%B4%E5%90%88%E7%90%86%E7%9A%84%E5%88%86%E9%85%8D)
  + [使用亲和性](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E4%BD%BF%E7%94%A8%E4%BA%B2%E5%92%8C%E6%80%A7)
  + [使用污点与容忍](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E4%BD%BF%E7%94%A8%E6%B1%A1%E7%82%B9%E4%B8%8E%E5%AE%B9%E5%BF%8D)
* [弹性伸缩](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%BC%B9%E6%80%A7%E4%BC%B8%E7%BC%A9)
  + [如何支持流量突发型业务](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%A6%82%E4%BD%95%E6%94%AF%E6%8C%81%E6%B5%81%E9%87%8F%E7%AA%81%E5%8F%91%E5%9E%8B%E4%B8%9A%E5%8A%A1)
  + [如何节约成本](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%A6%82%E4%BD%95%E8%8A%82%E7%BA%A6%E6%88%90%E6%9C%AC)
  + **[无法水平扩容的服务怎么办](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E6%97%A0%E6%B3%95%E6%B0%B4%E5%B9%B3%E6%89%A9%E5%AE%B9%E7%9A%84%E6%9C%8D%E5%8A%A1%E6%80%8E%E4%B9%88%E5%8A%9E)**
* [参考资料](https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/" \l "%E5%8F%82%E8%80%83%E8%B5%84%E6%96%99)

## 引言

业务容器化后，如何将其部署在 K8S 上？如果仅仅是将它跑起来，很简单，但如果是上生产，我们有许多地方是需要结合业务场景和部署环境进行方案选型和配置调优的。比如，如何设置容器的 Request 与 Limit、如何让部署的服务做到高可用、如何配置健康检查、如何进行弹性伸缩、如何更好的进行资源调度、如何选择持久化存储、如何对外暴露服务等。

对于这一系列高频问题，这里将会出一个 Kubernetes 服务部署最佳实践的系列的文章来为大家一一作答，本文将先围绕如何合理利用资源的主题来进行探讨。

## Request 与 Limit 怎么设置才好

如何为容器配置 Request 与 Limit? 这是一个即常见又棘手的问题，这个根据服务类型，需求与场景的不同而不同，没有固定的答案，这里结合生产经验总结了一些最佳实践，可以作为参考。

### **所有容器都应该设置 request**

request 的值并不是指给容器实际分配的资源大小，它仅仅是给调度器看的，调度器会 “观察” 每个节点可以用于分配的资源有多少，也知道每个节点已经被分配了多少资源。被分配资源的大小就是节点上所有 Pod 中定义的容器 request 之和，它可以计算出节点剩余多少资源可以被分配(可分配资源减去已分配的 request 之和)。如果发现节点剩余可分配资源大小比当前要被调度的 Pod 的 reuqest 还小，那么就不会考虑调度到这个节点，反之，才可能调度。所以，如果不配置 request，那么调度器就不能知道节点大概被分配了多少资源出去，调度器得不到准确信息，也就无法做出合理的调度决策，很容易造成调度不合理，有些节点可能很闲，而有些节点可能很忙，甚至 NotReady。

所以，建议是给所有容器都设置 request，让调度器感知节点有多少资源被分配了，以便做出合理的调度决策，让集群节点的资源能够被合理的分配使用，避免陷入资源分配不均导致一些意外发生。

### **老是忘记设置怎么办**

有时候我们会忘记给部分容器设置 request 与 limit，其实我们可以使用 LimitRange 来设置 namespace 的默认 request 与 limit 值，同时它也可以用来限制最小和最大的 request 与 limit。 示例:

apiVersion: v1kind: LimitRangemetadata:

name: mem-limit-range

namespace: testspec:

limits:

- default:

memory: 512Mi

cpu: 500m

defaultRequest:

memory: 256Mi

cpu: 100m

type: Container

### **重要的线上应用改如何设置**

节点资源不足时，会触发自动驱逐，将一些低优先级的 Pod 删除掉以释放资源让节点自愈。没有设置 request，limit 的 Pod 优先级最低，容易被驱逐；request 不等于 limit 的其次； request 等于 limit 的 Pod 优先级较高，不容易被驱逐。所以如果是重要的线上应用，不希望在节点故障时被驱逐导致线上业务受影响，就建议将 request 和 limit 设成一致。

### **怎样设置才能提高资源利用率**

如果给给你的应用设置较高的 request 值，而实际占用资源长期远小于它的 request 值，导致节点整体的资源利用率较低。当然这对时延非常敏感的业务除外，因为敏感的业务本身不期望节点利用率过高，影响网络包收发速度。所以对一些非核心，并且资源不长期占用的应用，可以适当减少 request 以提高资源利用率。

如果你的服务支持水平扩容，单副本的 request 值一般可以设置到不大于 1 核，CPU 密集型应用除外。比如 coredns，设置到 0.1 核就可以，即 100m。

### **尽量避免使用过大的 request 与 limit**

如果你的服务使用单副本或者少量副本，给很大的 request 与 limit，让它分配到足够多的资源来支撑业务，那么某个副本故障对业务带来的影响可能就比较大，并且由于 request 较大，当集群内资源分配比较碎片化，如果这个 Pod 所在节点挂了，其它节点又没有一个有足够的剩余可分配资源能够满足这个 Pod 的 request 时，这个 Pod 就无法实现漂移，也就不能自愈，加重对业务的影响。

相反，建议尽量减小 request 与 limit，通过增加副本的方式来对你的服务支撑能力进行水平扩容，让你的系统更加灵活可靠。

### **避免测试 namespace 消耗过多资源影响生产业务**

若生产集群有用于测试的 namespace，如果不加以限制，可能导致集群负载过高，从而影响生产业务。可以使用 ResourceQuota 来限制测试 namespace 的 request 与 limit 的总大小。 示例:

apiVersion: v1kind: ResourceQuotametadata:

name: quota-test

namespace: testspec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

## 如何让资源得到更合理的分配

设置 Request 能够解决让 Pod 调度到有足够资源的节点上，但无法做到更细致的控制。如何进一步让资源得到合理的使用？我们可以结合亲和性、污点与容忍等高级调度技巧，让 Pod 能够被合理调度到合适的节点上，让资源得到充分的利用。

### **使用亲和性**

* 对节点有特殊要求的服务可以用节点亲和性 (Node Affinity) 部署，以便调度到符合要求的节点，比如让 MySQL 调度到高 IO 的机型以提升数据读写效率。
* 可以将需要离得比较近的有关联的服务用 Pod 亲和性 (Pod Affinity) 部署，比如让 Web 服务跟它的 Redis 缓存服务都部署在同一可用区，实现低延时。
* 也可使用 Pod 反亲和 (Pod AntiAffinity) 将 Pod 进行打散调度，避免单点故障或者流量过于集中导致的一些问题。

### **使用污点与容忍**

使用污点 (Taint) 与容忍 (Toleration) 可优化集群资源调度:

* 通过给节点打污点来给某些应用预留资源，避免其它 Pod 调度上来。
* 需要使用这些资源的 Pod 加上容忍，结合节点亲和性让它调度到预留节点，即可使用预留的资源。

## 弹性伸缩

### **如何支持流量突发型业务**

通常业务都会有高峰和低谷，为了更合理的利用资源，我们为服务定义 HPA，实现根据 Pod 的资源实际使用情况来对服务进行自动扩缩容，在业务高峰时自动扩容 Pod 数量来支撑服务，在业务低谷时，自动缩容 Pod 释放资源，以供其它服务使用（比如在夜间，线上业务低峰，自动缩容释放资源以供大数据之类的离线任务运行) 。

使用 HPA 前提是让 K8S 得知道你服务的实际资源占用情况(指标数据)，需要安装 resource metrics (metrics.k8s.io) 或 custom metrics (custom.metrics.k8s.io) 的实现，好让 hpa controller 查询这些 API 来获取到服务的资源占用情况。早期 HPA 用 resource metrics 获取指标数据，后来推出 custom metrics，可以实现更灵活的指标来控制扩缩容。官方有个叫 [metrics-server](https://github.com/kubernetes-sigs/metrics-server" \t "https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/_blank) 的实现，通常社区使用的更多的是基于 prometheus 的 实现 [prometheus-adapter](https://github.com/DirectXMan12/k8s-prometheus-adapter" \t "https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/_blank)，而云厂商托管的 K8S 集群通常集成了自己的实现，比如 TKE，实现了 CPU、内存、硬盘、网络等维度的指标，可以在网页控制台可视化创建 HPA，但最终都会转成 K8S 的 yaml，示例:

apiVersion: autoscaling/v2beta2kind: HorizontalPodAutoscalermetadata:

name: nginxspec:

scaleTargetRef:

apiVersion: apps/v1beta2

kind: Deployment

name: nginx

minReplicas: 1

maxReplicas: 10

metrics:

- type: Pods

pods:

metric:

name: k8s\_pod\_rate\_cpu\_core\_used\_request

target:

averageValue: "100"

type: AverageValue

### **如何节约成本**

HPA 能实现 Pod 水平扩缩容，但如果节点资源不够用了，Pod 扩容出来还是会 Pending。如果我们提前准备好大量节点，做好资源冗余，提前准备好大量节点，通常不会有 Pod Pending 的问题，但也意味着需要付出更高的成本。通常云厂商托管的 K8S 集群都会实现 [cluster-autoscaler](https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler" \t "https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/_blank)，即根据资源使用情况，动态增删节点，让计算资源能够被最大化的弹性使用，按量付费，以节约成本。在 TKE 上的实现叫做伸缩组，以及一个包含伸缩功能组但更高级的特性：节点池(正在灰度)

### **无法水平扩容的服务怎么办**

对于无法适配水平伸缩的单体应用，或者不确定最佳 request 与 limit 超卖比的应用，可以尝用 [VPA](https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler" \t "https://imroc.cc/post/202006/kubernetes-app-deployment-best-practice-1/_blank) 来进行垂直伸缩，即自动更新 request 与 limit，然后重启 pod。不过这个特性容易导致你的服务出现短暂的不可用，不建议在生产环境中大规模使用。

## 参考资料

* Understanding Kubernetes limits and requests by example: <https://sysdig.com/blog/kubernetes-limits-requests/>
* Understanding resource limits in kubernetes: cpu time: <https://medium.com/@betz.mark/understanding-resource-limits-in-kubernetes-cpu-time-9eff74d3161b>
* Understanding resource limits in kubernetes: memory: <https://medium.com/@betz.mark/understanding-resource-limits-in-kubernetes-memory-6b41e9a955f9>
* Kubernetes best practices: Resource requests and limits: <https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits>
* Kubernetes 资源分配之 Request 和 Limit 解析: <https://cloud.tencent.com/developer/article/1004976>
* Assign Pods to Nodes using Node Affinity: <https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/>
* Taints and Tolerations: <https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/>
* metrics-server: <https://github.com/kubernetes-sigs/metrics-server>
* prometheus-adapter: <https://github.com/DirectXMan12/k8s-prometheus-adapter>
* cluster-autoscaler: <https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler>
* VPA: <https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>