概念

概念部分可以帮助你了解 Kubernetes 的各个组成部分以及 Kubernetes 用来表示集群的一些抽象概念，并帮助你更加深入的理解 Kubernetes 是如何工作的。

* [**概述**](https://kubernetes.io/zh/docs/concepts/#%e6%a6%82%e8%bf%b0)
* [**Kubernetes 对象**](https://kubernetes.io/zh/docs/concepts/#kubernetes-%e5%af%b9%e8%b1%a1)
* [**Kubernetes 控制面**](https://kubernetes.io/zh/docs/concepts/#kubernetes-%e6%8e%a7%e5%88%b6%e9%9d%a2)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

概述

要使用 Kubernetes，你需要用 *Kubernetes API 对象* 来描述集群的 *预期状态（desired state）* ：包括你需要运行的应用或者负载，它们使用的镜像、副本数，以及所需网络和磁盘资源等等。你可以使用命令行工具 kubectl 来调用 Kubernetes API 创建对象，通过所创建的这些对象来配置预期状态。你也可以直接调用 Kubernetes API 和集群进行交互，设置或者修改预期状态。

一旦你设置了你所需的目标状态，*Kubernetes 控制面（control plane）* 会通过 Pod 生命周期事件生成器( PLEG )，促成集群的当前状态符合其预期状态。为此，Kubernetes 会自动执行各类任务，比如运行或者重启容器、调整给定应用的副本数等等。Kubernetes 控制面由一组运行在集群上的进程组成：

* **Kubernetes 主控组件（Master）** 包含三个进程，都运行在集群中的某个节上，通常这个节点被称为 master 节点。这些进程包括：[kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/)、[kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/) 和 [kube-scheduler](https://kubernetes.io/docs/admin/kube-scheduler/)。
* 集群中的每个非 master 节点都运行两个进程：
  + [**kubelet**](https://kubernetes.io/docs/admin/kubelet/)，和 master 节点进行通信。
  + [**kube-proxy**](https://kubernetes.io/docs/admin/kube-proxy/)，一种网络代理，将 Kubernetes 的网络服务代理到每个节点上。

Kubernetes 对象

Kubernetes 包含若干抽象用来表示系统状态，包括：已部署的容器化应用和负载、与它们相关的网络和磁盘资源以及有关集群正在运行的其他操作的信息。这些抽象使用 Kubernetes API 对象来表示。参阅 [Kubernetes 对象概述](https://kubernetes.io/docs/concepts/abstractions/overview/)以了解详细信息。

基本的 Kubernetes 对象包括：

* [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/)
* [Service](https://kubernetes.io/docs/concepts/services-networking/service/)
* [Volume](https://kubernetes.io/docs/concepts/storage/volumes/)
* [Namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)

另外，Kubernetes 包含大量的被称作*控制器（controllers）* 的高级抽象。控制器基于基本对象构建并提供额外的功能和方便使用的特性。具体包括：

* [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)
* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)
* [Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)

Kubernetes 控制面

关于 Kubernetes 控制平面的各个部分，（如 Kubernetes 主控组件和 kubelet 进程），管理着 Kubernetes 如何与你的集群进行通信。控制平面维护着系统中所有的 Kubernetes 对象的状态记录，并且通过连续的控制循环来管理这些对象的状态。在任意的给定时间点，控制面的控制环都能响应集群中的变化，并且让系统中所有对象的实际状态与你提供的预期状态相匹配。

比如， 当你通过 Kubernetes API 创建一个 Deployment 对象，你就为系统增加了一个新的目标状态。Kubernetes 控制平面记录着对象的创建，并启动必要的应用然后将它们调度至集群某个节点上来执行你的指令，以此来保持集群的实际状态和目标状态的匹配。

Kubernetes Master 节点

Kubernetes master 节点负责维护集群的目标状态。当你要与 Kubernetes 通信时，使用如 kubectl 的命令行工具，就可以直接与 Kubernetes master 节点进行通信。

“master” 是指管理集群状态的一组进程的集合。通常这些进程都跑在集群中一个单独的节点上，并且这个节点被称为 master 节点。master 节点也可以扩展副本数，来获取更好的可用性及冗余。

Kubernetes Node 节点

集群中的 node 节点（虚拟机、物理机等等）都是用来运行你的应用和云工作流的机器。Kubernetes master 节点控制所有 node 节点；你很少需要和 node 节点进行直接通信。

对象元数据

* [注解](https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/)

接下来

如果你想编写一个概念页面，请参阅[使用页面模板](https://kubernetes.io/docs/home/contribute/page-templates/)获取更多有关概念页面类型和概念模板的信息。

# Kubernetes 架构

## 节点

在 Kubernetes 中，节点（Node）是执行工作的机器，以前叫做 minion。根据你的集群环境，节点可以是一个虚拟机或者物理机器。每个节点都包含用于运行 [pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) 的必要服务，并由主控组件管理。节点上的服务包括 [容器运行时](https://kubernetes.io/docs/concepts/overview/components/#node-components)、kubelet 和 kube-proxy。查阅架构设计文档中 [Kubernetes 节点](https://git.k8s.io/community/contributors/design-proposals/architecture/architecture.md#the-kubernetes-node) 一节获取更多细节。

* [**节点状态**](https://kubernetes.io/zh/docs/concepts/architecture/#%e8%8a%82%e7%82%b9%e7%8a%b6%e6%80%81)
* [**管理**](https://kubernetes.io/zh/docs/concepts/architecture/#%e7%ae%a1%e7%90%86)
* [**节点拓扑**](https://kubernetes.io/zh/docs/concepts/architecture/#%e8%8a%82%e7%82%b9%e6%8b%93%e6%89%91)
* [**API 对象**](https://kubernetes.io/zh/docs/concepts/architecture/#api-%e5%af%b9%e8%b1%a1)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/architecture/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

节点状态

一个节点的状态包含以下信息:

* [地址](https://kubernetes.io/zh/docs/concepts/architecture/#addresses)
* [条件](https://kubernetes.io/zh/docs/concepts/architecture/#condition)
* [容量与可分配](https://kubernetes.io/zh/docs/concepts/architecture/#capacity)
* [信息](https://kubernetes.io/zh/docs/concepts/architecture/#info)

可以使用以下命令显示节点状态和有关节点的其他详细信息：

kubectl describe node <insert-node-name-here>

下面对每个章节进行详细描述。

地址

这些字段组合的用法取决于你的云服务商或者裸机配置。

* HostName：由节点的内核指定。可以通过 kubelet 的 --hostname-override 参数覆盖。
* ExternalIP：通常是可以外部路由的节点 IP 地址（从集群外可访问）。
* InternalIP：通常是仅可在集群内部路由的节点 IP 地址。

条件

conditions 字段描述了所有 Running 节点的状态。条件的示例包括：

| 节点条件 | 描述 |
| --- | --- |
| OutOfDisk | True 表示节点的空闲空间不足以用于添加新 pods, 否则为 False |
| Ready | 表示节点是健康的并已经准备好接受 pods；False 表示节点不健康而且不能接受 pods；Unknown 表示节点控制器在最近 40 秒内没有收到节点的消息 |
| MemoryPressure | True 表示节点存在内存压力 – 即节点内存用量低，否则为 False |
| PIDPressure | True 表示节点存在进程压力 – 即进程过多；否则为 False |
| DiskPressure | True 表示节点存在磁盘压力 – 即磁盘用量低，否则为 False |
| NetworkUnavailable | True 表示节点网络配置不正确；否则为 False |

节点条件使用一个 JSON 对象表示。例如，下面的响应描述了一个健康的节点。

"conditions": [

{

**"type"**: "Ready",

**"status"**: "True",

**"reason"**: "KubeletReady",

**"message"**: "kubelet is posting ready status",

**"lastHeartbeatTime"**: "2019-06-05T18:38:35Z",

**"lastTransitionTime"**: "2019-06-05T11:41:27Z"

}

]

如果 Ready 条件处于状态 Unknown 或者 False 的时间超过了 pod-eviction-timeout（一个传递给 [kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/) 的参数），节点上的所有 Pods 都会被节点控制器计划删除。默认的删除超时时长为**5 分钟**。某些情况下，当节点不可访问时，apiserver 不能和其上的 kubelet 通信。删除 pods 的决定不能传达给 kubelet，直到它重新建立和 apiserver 的连接为止。与此同时，被计划删除的 pods 可能会继续在分区节点上运行。

在 1.5 版本之前的 Kubernetes 里，节点控制器会将不能访问的 pods 从 apiserver 中[强制删除](https://kubernetes.io/docs/concepts/workloads/pods/pod/#force-deletion-of-pods)。但在 1.5 或更高的版本里，在节点控制器确认这些 pods 已经在集群停止运行前不会强制删除它们。你可以看到这些处于 Terminating 或者 Unknown 状态的 pods 可能在无法访问的节点上运行。为了防止 kubernetes 不能从底层基础设施中推断出一个节点是否已经永久的离开了集群，集群管理员可能需要手动删除这个节点对象。从 Kubernetes 删除节点对象将导致 apiserver 删除节点上所有运行的 Pod 对象并释放它们的名字。

节点生命周期控制器会自动创建代表条件的[污点](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)。 当调度器将 Pod 分配给节点时，调度器会考虑节点上的污点，但是 Pod 可以容忍的污点除外。

容量与可分配

描述节点上的可用资源：CPU、内存和可以调度到节点上的 pods 的最大数量。

capacity 块中的字段指示节点拥有的资源总量。allocatable 块指示节点上可供普通 Pod 消耗的资源量。

可以在学习如何在节点上[保留计算资源](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable)的同时阅读有关容量和可分配资源的更多信息。

信息

关于节点的通用信息，例如内核版本、Kubernetes 版本（kubelet 和 kube-proxy 版本）、Docker 版本（如果使用了）和操作系统名称。这些信息由 kubelet 从节点上搜集而来。

管理

与 [pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) 和 [services](https://kubernetes.io/docs/concepts/services-networking/service/) 不同，节点并不是在 Kubernetes 内部创建的：它是被外部的云服务商创建，例如 Google Compute Engine 或者你的集群中的物理或者虚拟机。这意味着当 Kubernetes 创建一个节点时，它其实仅仅创建了一个对象来代表这个节点。创建以后，Kubernetes 将检查这个节点是否可用。例如，如果你尝试使用如下内容创建一个节点：

{

**"kind"**: "Node",

**"apiVersion"**: "v1",

**"metadata"**: {

**"name"**: "10.240.79.157",

**"labels"**: {

**"name"**: "my-first-k8s-node"

}

}

}

Kubernetes 会在内部创一个 Node 对象（用以表示节点），并基于 metadata.name 字段执行健康检查，对节点进行验证。如果节点可用，意即所有必要服务都已运行，它就符合了运行一个 pod 的条件；否则它将被所有的集群动作忽略直到变为可用。

**注意：** Kubernetes 保留无效节点的对象，并继续检查它是否有效。必须显式删除 Node 对象以停止此过程。

当前，有 3 个组件同 Kubernetes 节点接口交互：节点控制器、kubelet 和 kubectl。

节点控制器

节点控制器是一个 Kubernetes master 组件，管理节点的方方面面。

节点控制器在节点的生命周期中扮演了多个角色。第一个是当节点注册时为它分配一个 CIDR block（如果打开了 CIDR 分配）。

第二个是使用云服务商提供了可用节点列表保持节点控制器内部的节点列表更新。如果在云环境下运行，任何时候当一个节点不健康时节点控制器将询问云服务节点的虚拟机是否可用。如果不可用，节点控制器会将这个节点从它的节点列表删除。

第三个是监控节点的健康情况。节点控制器负责在节点不能访问时（也即是节点控制器因为某些原因没有收到心跳，例如节点宕机）将它的 NodeStatus 的 NodeReady 状态更新为 ConditionUnknown。后续如果节点持续不可访问，节点控制器将删除节点上的所有 pods（使用优雅终止）。（默认情况下 40s 开始报告 ConditionUnknown，在那之后 5m 开始删除 pods。）节点控制器每隔 --node-monitor-period 秒检查每个节点的状态。

心跳机制

Kubernetes 节点发送的心跳有助于确定节点的可用性。 心跳有两种形式：NodeStatus 和 [Lease 对象](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#lease-v1-coordination-k8s-io)。 每个节点在 kube-node-lease[命名空间](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces) 中都有一个关联的 Lease 对象。 Lease 是一种轻量级的资源，可在集群扩展时提高节点心跳机制的性能。

kubelet 负责创建和更新 NodeStatus 和 Lease 对象。

* 当状态发生变化时，或者在配置的时间间隔内没有更新时，kubelet 会更新 NodeStatus。 NodeStatus 更新的默认间隔为 5 分钟（比无法访问的节点的 40 秒默认超时时间长很多）。
* kubelet 会每 10 秒（默认更新间隔时间）创建并更新其 Lease 对象。Lease 更新独立于 NodeStatus 更新而发生。

可靠性

在 Kubernetes 1.4 中我们更新了节点控制器逻辑以更好地处理大批量节点访问 master 出问题的情况（例如 master 的网络出了问题）。从 1.4 开始，节点控制器在决定删除 pod 之前会检查集群中所有节点的状态。

大部分情况下，节点控制器把驱逐频率限制在每秒 --node-eviction-rate 个（默认为 0.1）。这表示它每 10 秒钟内之多从一个节点驱逐 Pods。

当一个可用区域中的节点变为不健康时，它的驱逐行为将发生改变。节点控制器会同时检查可用区域中不健康（NodeReady 状态为 ConditionUnknown 或 ConditionFalse）的节点的百分比。如果不健康节点的部分超过 --unhealthy-zone-threshold （默认为 0.55），驱逐速率将会减小：如果集群较小（意即小于等于 --large-cluster-size-threshold 个 节点 - 默认为 50），驱逐操作将会停止，否则驱逐速率将降为每秒 --secondary-node-eviction-rate 个（默认为 0.01）。在单个可用区域实施这些策略的原因是当一个可用区域可能从 master 分区时其它的仍然保持连接。如果你的集群没有跨越云服务商的多个可用区域，那就只有一个可用区域整个集群）。

在多个可用区域分布你的节点的一个关键原因是当整个可用区域故障时，工作负载可以转移到健康的可用区域。因此，如果一个可用区域中的所有节点都不健康时，节点控制器会以正常的速率 --node-eviction-rate 进行驱逐操作。在所有的可用区域都不健康（也即集群中没有健康节点）的极端情况下，节点控制器将假设 master 的连接出了某些问题，它将停止所有驱逐动作直到一些连接恢复。

从 Kubernetes 1.6 开始，NodeController 还负责驱逐运行在拥有 NoExecute 污点的节点上的 pods，如果这些 pods 没有容忍这些污点。此外，作为一个默认禁用的 alpha 特性，NodeController 还负责根据节点故障（例如节点不可访问或没有 ready）添加污点。请查看[这个文档](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#taints-and-tolerations-beta-feature)了解关于 NoExecute 污点和这个 alpha 特性。

从版本 1.8 开始，可以使节点控制器负责创建代表节点条件的污点。这是版本 1.8 的 Alpha 功能。

节点自注册

当 kubelet 标志 --register-node 为 true （默认）时，它会尝试向 API 服务注册自己。这是首选模式，被绝大多数发行版选用。

对于自注册模式，kubelet 使用下列参数启动：

* --kubeconfig - 用于向 apiserver 验证自己的凭据路径。
* --cloud-provider - 如何从云服务商读取关于自己的元数据。
* --register-node - 自动向 API 服务注册。
* --register-with-taints - 使用 taints 列表（逗号分隔的 <key>=<value>:<effect>）注册节点。当 register-node 为 false 时无效。
* --node-ip - 节点 IP 地址。
* --node-labels - 在集群中注册节点时要添加的标签（请参阅 [NodeRestriction 准入插件](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction) 在 1.13+ 中实施的标签限制）。
* --node-status-update-frequency - 指定 kubelet 向 master 发送状态的频率。

启用[节点授权模式](https://kubernetes.io/docs/reference/access-authn-authz/node/) 和 [NodeRestriction 准入插件](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction)时，仅授权小组件创建或修改其自己的节点资源。

手动节点管理

集群管理员可以创建及修改节点对象。

如果管理员希望手动创建节点对象，请设置 kubelet 标记 --register-node=false。

管理员可以修改节点资源（忽略 --register-node 设置）。修改包括在节点上设置 labels 及标记它为不可调度。

节点上的 labels 可以和 pods 的节点 selectors 一起使用来控制调度，例如限制一个 pod 只能在一个符合要求的节点子集上运行。

标记一个节点为不可调度的将防止新建 pods 调度到那个节点之上，但不会影响任何已经在它之上的 pods。这是重启节点等操作之前的一个有用的准备步骤。例如，标记一个节点为不可调度的，执行以下命令：

kubectl cordon $NODENAME

**注意：**

请注意，被 daemonSet 控制器创建的 pods 将忽略 Kubernetes 调度器，且不会遵照节点上不可调度的属性。这个假设基于守护程序属于节点机器，即使在准备重启而隔离应用的时候。

节点容量

节点的容量（cpu 数量和内存容量）是节点对象的一部分。通常情况下，在创建节点对象时，它们会注册自己并报告自己的容量。如果你正在执行[手动节点管理](https://kubernetes.io/zh/docs/concepts/architecture/#manual-node-administration)，那么你需要在添加节点时手动设置节点容量。

Kubernetes 调度器保证一个节点上有足够的资源供其上的所有 pods 使用。它会检查节点上所有容器要求的总和不会超过节点的容量。这包括由 kubelet 启动的所有容器，但不包括由 [container runtime](https://kubernetes.io/docs/concepts/overview/components/#node-components) 直接启动的容器，也不包括在容器外部运行的任何进程。

如果要为非 Pod 进程显式保留资源。请按照本教程[为系统守护程序保留资源](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#system-reserved)。

节点拓扑

**FEATURE STATE:** Kubernetes v1.17 [alpha](https://kubernetes.io/zh/docs/concepts/architecture/)

如果启用了 TopologyManager [功能开关](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/)，则 kubelet 可以在做出资源分配决策时使用拓扑提示。

API 对象

节点是 Kubernetes REST API 的顶级资源。更多关于 API 对象的细节可以在这里找到：[节点 API 对象](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#node-v1-core)。

* 了解有关[节点组件](https://kubernetes.io/docs/concepts/overview/components/#node-components)的信息。
* 阅读有关节点级拓扑的信息：[控制节点上的拓扑管理策略](https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/)。

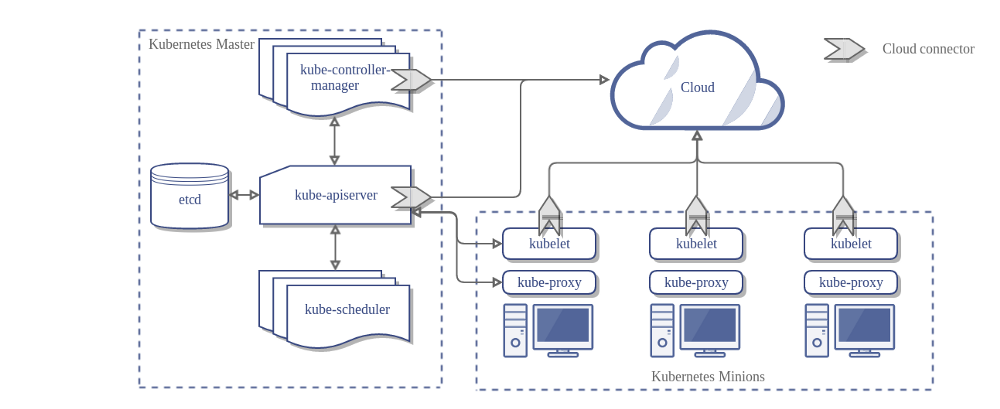
云控制器管理器的基础概念

云控制器管理器（cloud controller manager，CCM）这个概念 （不要与二进制文件混淆）创建的初衷是为了让特定的云服务供应商代码和 Kubernetes 核心相互独立演化。云控制器管理器与其他主要组件（如 Kubernetes 控制器管理器，API 服务器和调度程序）一起运行。它也可以作为 Kubernetes 的插件启动，在这种情况下，它会运行在 Kubernetes 之上。

云控制器管理器基于插件机制设计，允许新的云服务供应商通过插件轻松地与 Kubernetes 集成。目前已经有在 Kubernetes 上加入新的云服务供应商计划，并为云服务供应商提供从原先的旧模式迁移到新 CCM 模式的方案。

本文讨论了云控制器管理器背后的概念，并提供了相关功能的详细信息。

这是没有云控制器管理器的 Kubernetes 集群的架构：



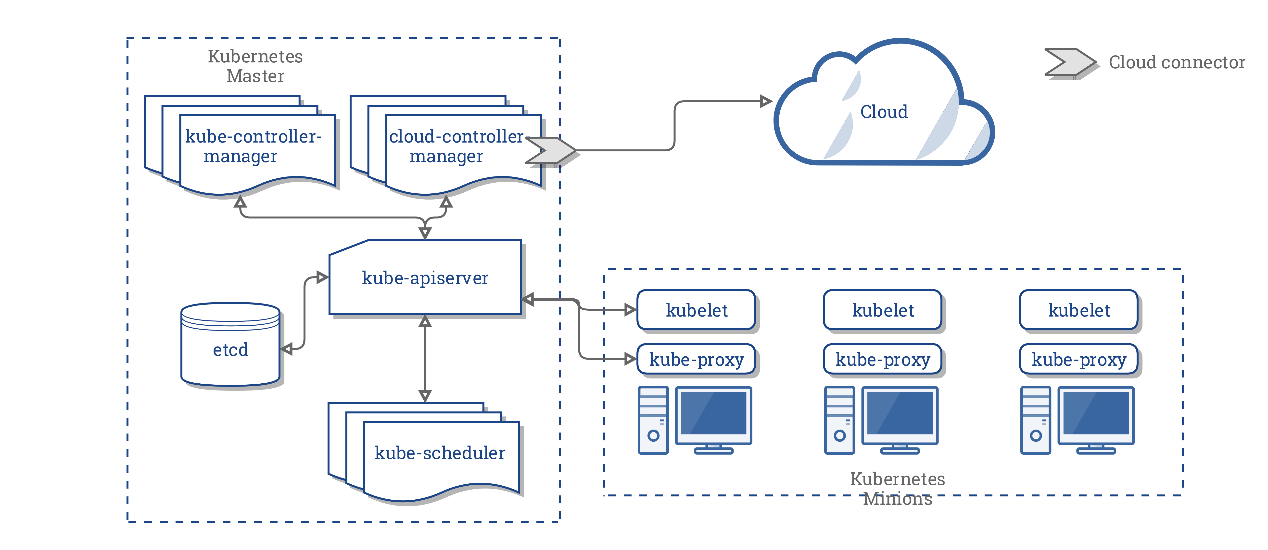
* [**设计**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#%e8%ae%be%e8%ae%a1)
* [**CCM 的组成部分**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#ccm-%e7%9a%84%e7%bb%84%e6%88%90%e9%83%a8%e5%88%86)
* [**CCM 的功能**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#ccm-%e7%9a%84%e5%8a%9f%e8%83%bd)
* [**插件机制**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#%e6%8f%92%e4%bb%b6%e6%9c%ba%e5%88%b6)
* [**授权**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#%e6%8e%88%e6%9d%83)
* [**供应商实施**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#%e4%be%9b%e5%ba%94%e5%95%86%e5%ae%9e%e6%96%bd)
* [**群集管理**](https://kubernetes.io/zh/docs/concepts/architecture/cloud-controller/#%e7%be%a4%e9%9b%86%e7%ae%a1%e7%90%86)

设计

在上图中，Kubernetes 和云服务供应商通过几个不同的组件进行了集成，分别是：

* Kubelet
* Kubernetes 控制管理器
* Kubernetes API 服务器

CCM 整合了前三个组件中的所有依赖于云的逻辑，以创建与云的单一集成点。CCM 的新架构如下所示：



CCM 的组成部分

CCM 打破了 Kubernetes 控制器管理器（KCM）的一些功能，并将其作为一个单独的进程运行。具体来说，它打破了 KCM 中依赖于云的控制器。KCM 具有以下依赖于云的控制器：

* 节点控制器
* 卷控制器
* 路由控制器
* 服务控制器

在 1.9 版本中，CCM 运行前述列表中的以下控制器：

* 节点控制器
* 路由控制器
* 服务控制器

**注意：**

注意卷控制器不属于 CCM，由于其中涉及到的复杂性和对现有供应商特定卷的逻辑抽象，因此决定了卷控制器不会被移动到 CCM 之中。

使用 CCM 支持 volume 的最初计划是使用 Flex volume 来支持可插拔卷，但是现在正在计划一项名为 CSI 的项目以取代 Flex。

考虑到这些正在进行中的变化，在 CSI 准备就绪之前，我们决定停止当前的工作。

CCM 的功能

CCM 从依赖于云提供商的 Kubernetes 组件继承其功能，本节基于这些组件组织。

1. Kubernetes 控制器管理器

CCM 的大多数功能都来自 KCM，如上一节所述，CCM 运行以下控制器。

* 节点控制器
* 路由控制器
* 服务控制器

节点控制器

节点控制器负责通过从云提供商获取有关在集群中运行的节点的信息来初始化节点，节点控制器执行以下功能：

1. 使用特定于云的域（zone）/区（region）标签初始化节点；
2. 使用特定于云的实例详细信息初始化节点，例如，类型和大小；
3. 获取节点的网络地址和主机名；
4. 如果节点无响应，请检查云以查看该节点是否已从云中删除。如果已从云中删除该节点，请删除 Kubernetes 节点对象。

路由控制器

Route 控制器负责适当地配置云中的路由，以便 Kubernetes 集群中不同节点上的容器可以相互通信。route 控制器仅适用于 Google Compute Engine 群集。

服务控制器

2. Kubelet

节点控制器包含 kubelet 中依赖于云的功能，在引入 CCM 之前，kubelet 负责使用特定于云的详细信息（如 IP 地址，域/区标签和实例类型信息）初始化节点。CCM 的引入已将此初始化操作从 kubelet 转移到 CCM 中。

在这个新模型中，kubelet 初始化一个没有特定于云的信息的节点。但是，它会为新创建的节点添加污点，使节点不可调度，直到 CCM 使用特定于云的信息初始化节点后，才会清除这种污点，便得该节点可被调度。

在这个新模型中，kubelet 初始化一个没有特定于云的信息的节点。但是，它会为新创建的节点添加污点，使节点不可调度，直到 CCM 使用特定于云的信息初始化节点后，才会清除这种污点，便得该节点可被调度。

插件机制

云控制器管理器使用 Go 接口允许插入任何云的实现。具体来说，它使用[此处](https://github.com/kubernetes/cloud-provider/blob/9b77dc1c384685cb732b3025ed5689dd597a5971/cloud.go#L42-L62)定义的 CloudProvider 接口。

上面强调的四个共享控制器的实现，以及一些辅助设施（scaffolding）和共享的 cloudprovider 接口，将被保留在 Kubernetes 核心中。但特定于云提供商的实现将在核心之外构建，并实现核心中定义的接口。

有关开发插件的更多信息，请参阅[开发云控制器管理器](https://kubernetes.io/docs/tasks/administer-cluster/developing-cloud-controller-manager/)。

授权

本节分解了 CCM 执行其操作时各种 API 对象所需的访问权限。

节点控制器

Node 控制器仅适用于 Node 对象，它需要完全访问权限来获取、列出、创建、更新、修补、监视和删除 Node 对象。

v1/Node:

* Get
* List
* Create
* Update
* Patch
* Watch
* Delete

路由控制器

路由控制器侦听 Node 对象创建并适当地配置路由，它需要访问 Node 对象。

v1/Node:

* Get

服务控制器

服务控制器侦听 Service 对象创建、更新和删除事件，然后适当地为这些服务配置端点。

要访问服务，它需要列表和监视访问权限。要更新服务，它需要修补和更新访问权限。

要为服务设置端点，需要访问 create、list、get、watch 和 update。

v1/Service:

* List
* Get
* Watch
* Patch
* Update

其它

CCM 核心的实现需要访问权限以创建事件，并且为了确保安全操作，它需要访问权限以创建服务账户。

v1/Event:

* Create
* Patch
* Update

v1/ServiceAccount:

* Create

针对 CCM 的 RBAC ClusterRole 看起来像这样：

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: cloud-controller-manager

rules:

- apiGroups:

- ""

resources:

- events

verbs:

- create

- patch

- update

- apiGroups:

- ""

resources:

- nodes

verbs:

- '\*'

- apiGroups:

- ""

resources:

- nodes/status

verbs:

- patch

- apiGroups:

- ""

resources:

- services

verbs:

- list

- patch

- update

- watch

- apiGroups:

- ""

resources:

- serviceaccounts

verbs:

- create

- apiGroups:

- ""

resources:

- persistentvolumes

verbs:

- get

- list

- update

- watch

- apiGroups:

- ""

resources:

- endpoints

verbs:

- create

- get

- list

- watch

- update

供应商实施

以下云服务提供商已实现了 CCM：

* [AWS](https://github.com/kubernetes/cloud-provider-aws)
* [Azure](https://github.com/kubernetes/cloud-provider-azure)
* [BaiduCloud](https://github.com/baidu/cloud-provider-baiducloud)
* [Digital Ocean](https://github.com/digitalocean/digitalocean-cloud-controller-manager)
* [GCP](https://github.com/kubernetes/cloud-provider-gcp)
* [Linode](https://github.com/linode/linode-cloud-controller-manager)
* [OpenStack](https://github.com/kubernetes/cloud-provider-openstack)
* [Oracle](https://github.com/oracle/oci-cloud-controller-manager)

群集管理

[这里](https://kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller/#cloud-controller-manager)提供了有关配置和运行 CCM 的完整说明。

控制器

在机器人技术和自动化中，控制环是一个控制系统状态的不终止的循环。

这是一个控制环的例子：房间里的温度自动调节器。

当你设置了温度，告诉了温度自动调节器你的\*期望状态\*。房间的实际温度是\*当前状态\*。通过对设备的开关控制，温度自动调节器让其当前状态接近期望状态。

控制器通过 [apiserver](https://kubernetes.io/docs/reference/generated/kube-apiserver/) 监控集群的公共状态，并致力于将当前状态转变为期望的状态。

* [**控制器模式**](https://kubernetes.io/zh/docs/concepts/architecture/controller/#controller-pattern)
* [**期望状态与当前状态**](https://kubernetes.io/zh/docs/concepts/architecture/controller/#desired-vs-current)
* [**Design**](https://kubernetes.io/zh/docs/concepts/architecture/controller/#design)
* [**运行控制器的方式**](https://kubernetes.io/zh/docs/concepts/architecture/controller/#running-controllers)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/architecture/controller/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

控制器模式

一个控制器至少追踪一种类型的 Kubernetes 资源。这些[对象](https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/)有一个代表期望状态的指定字段。正对这种资源的控制器就是要使他的当前状态接近与期望状态。

控制器可能会自行执行操作；在 Kubernetes 中更常见的是一个控制器会发送信息给 [API 服务器](https://kubernetes.io/docs/reference/generated/kube-apiserver/)，这会有副作用。看下面这个例子。

通过 API 服务器来控制

[Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion) 控制器是一个 Kubernetes 内置控制器的例子。内置控制器通过和集群 API 服务器交互来管理状态。

Job 是一种 Kubernetes 资源，它运行一个 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/)，或者可能是多个 Pod，来执行一个任务然后停止。

（一旦[被调度了](https://kubernetes.io/docs/concepts/scheduling/)），对 kubelet 来说 Pod 对象就会变成了期望状态的一部分。

在集群中，当 Job 控制器拿到新任务时，它会保证一组 Node 节点上的 kubelet 可以运行正确数量的 Pod 来完成工作。 Job 控制器不会自己运行任何的 Pod 或者容器。Job 控制器是通知 API 服务器来创建或者移除 Pod。 [控制平面](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-control-plane)中的其它组件根据新的消息而反应（调度新的 Pod 并且运行它）并且最终完成工作。

创建新 Job 后，所期望的状态就是完成这个 Job。Job 控制器会让 Job 的当前状态不断接近期望状态：创建为 Job 要完成工作所需要的 Pod，使 Job 的状态接近完成。

控制器也会更新配置对象。例如：一旦 Job 的工作完成了，Job 控制器会更新 Job 对象的状态为 Finished。

（这有点像温度自动调节器关闭了一个灯，以此来告诉你房间的温度现在到你设定的值了）。

直接控制

相比 Job 控制器，有些控制器需要对集群外的一些东西进行修改。

例如，如果你使用一个控制环来保证集群中有足够的[节点](https://kubernetes.io/docs/concepts/architecture/nodes/)，那么控制就需要当前集群外的一些服务在需要时创建新节点。

和外部状态交互的控制器从 API 服务器获取到它想要的状态，然后直接和外部系统进行通信并使当前状态更接近期望状态。

（实际上有一个控制器可以水平地扩展集群中的节点。请看[集群自动扩缩容](https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#cluster-autoscaling)）。

期望状态与当前状态

Kubernetes 采用了系统的云原生视图，并且可以处理持续的变化。

在任务执行时，集群随时都可能被修改，并且控制环会自动的修复故障。这意味着很可能集群永远不会达到稳定状态。

只要集群中控制器的在运行并且进行有效的修改，整体状态的稳定与否是无关紧要的。 <!–

Design

As a tenet of its design, Kubernetes uses lots of controllers that each manage a particular aspect of cluster state. Most commonly, a particular control loop (controller) uses one kind of resource as its desired state, and has a different kind of resource that it manages to make that desired state happen.

It’s useful to have simple controllers rather than one, monolithic set of control loops that are interlinked. Controllers can fail, so Kubernetes is designed to allow for that.

For example: a controller for Jobs tracks Job objects (to discover new work) and Pod object (to run the Jobs, and then to see when the work is finished). In this case something else creates the Jobs, whereas the Job controller creates Pods.

**注意：**

There can be several controllers that create or update the same kind of object. Behind the scenes, Kubernetes controllers make sure that they only pay attention to the resources linked to their controlling resource.

For example, you can have Deployments and Jobs; these both create Pods. The Job controller does not delete the Pods that your Deployment created, because there is information ([labels](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels)) the controllers can use to tell those Pods apart.

--> ## 设计 {#design} 作为设计的一个原则，Kubernetes 使用了很多控制器，每个控制器管理集群状态的一个特定方面。最常见的一个特定的控制器使用一种类型的资源作为它的期望状态，控制器管理控制另外一种类型的资源向它的期望状态发展。 使用简单的控制器而不是一组相互连接的单体控制环是很有用的。控制器会失败，所以 Kubernetes 的设计是考虑到了这一点。 例如：为 Job 追踪 Job 对象（发现新工作）和 Pod 对象（运行 Job，并且等工作完成）的控制器。在本例中，其它东西创建作业，而作业控制器创建 Pod。

**注意：**

可以有多个控制器来创建或者更新相同类型的对象。在这之后，Kubernetes 控制器确保他们只关心和它们控制资源相关联的资源。

例如，你可以有 Deployments 和 Jobs；它们都可以创建 Pod。Job 控制器不删除 Deployment 创建的 Pod，因为有信息([标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels))让控制器可以区分这些 Pod。

运行控制器的方式

Kubernetes 自带有一组内置的控制器，运行在 [kube-controller-manager](https://kubernetes.io/docs/reference/generated/kube-controller-manager/) 内。这些内置的控制器提供了重要的核心功能。

Deployment 控制器和 Job 控制器是 Kubernetes 内置控制器的典型例子。Kubernetes 运行一个弹性的控制平面，所以如果任意内置控制器失败了，控制平面的另外一部分会接替它的工作。

你会发现控制平面外面运行的控制器，扩展了 Kubernetes 的能力。或者，如果你愿意，你也可以写一个新控制器。你可以以一组 Pod 来运行你的控制器，或者运行在 Kubernetes 外面。什么是最合适的控制器，这将取决于特定控制器的功能。

* 请阅读 [Kubernetes 控制平面](https://kubernetes.io/docs/concepts/#kubernetes-control-plane)
* 了解一些基本的 [Kubernetes 对象](https://kubernetes.io/docs/concepts/#kubernetes-objects)
* 学习更多的 [Kubernetes API](https://kubernetes.io/docs/concepts/overview/kubernetes-api/)
* 如果你想写自己的控制器，请看 Kubernetes 的[扩展模式](https://kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster/#extension-patterns)。

Master 节点通信

* + [**概览**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#概览)
  + [**Cluster -> Master**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#cluster-master)
  + [**Master -> Cluster**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#master-cluster)
  + [**apiserver -> kubelet**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#apiserver-kubelet)
  + [**apiserver -> nodes, pods, and services**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#apiserver-nodes-pods-and-services)
  + [**SSH 隧道**](https://kubernetes.io/zh/docs/concepts/architecture/master-node-communication/#ssh-隧道)

概览

本文对 Master 节点（确切说是 apiserver）和 Kubernetes 集群之间的通信路径进行了分类。目的是为了让用户能够自定义他们的安装，对网络配置进行加固，使得集群能够在不可信的网络上（或者在一个云服务商完全公共的 IP 上）运行。

Cluster -> Master

所有从集群到 master 的通信路径都终止于 apiserver（其它 master 组件没有被设计为可暴露远程服务）。在一个典型的部署中，apiserver 被配置为在一个安全的 HTTPS 端口（443）上监听远程连接并启用一种或多种形式的客户端[身份认证](https://kubernetes.io/docs/admin/authentication/)机制。一种或多种客户端[身份认证](https://kubernetes.io/docs/admin/authentication/)机制应该被启用，特别是在允许使用 [匿名请求](https://kubernetes.io/docs/admin/authentication/#anonymous-requests) 或 [service account tokens](https://kubernetes.io/docs/admin/authentication/#service-account-tokens) 的时候。

应该使用集群的公共根证书开通节点，如此它们就能够基于有效的客户端凭据安全的连接 apiserver。例如：在一个默认的 GCE 部署中，客户端凭据以客户端证书的形式提供给 kubelet。请查看 [kubelet TLS bootstrapping](https://kubernetes.io/docs/admin/kubelet-tls-bootstrapping/) 获取如何自动提供 kubelet 客户端证书。

想要连接到 apiserver 的 Pods 可以使用一个 service account 安全的进行连接。这种情况下，当 Pods 被实例化时 Kubernetes 将自动的把公共根证书和一个有效的不记名令牌注入到 pod 里。kubernetes service （所有 namespaces 中）都配置了一个虚拟 IP 地址，用于转发（通过 kube-proxy）请求到 apiserver 的 HTTPS endpoint。

Master 组件通过非安全（没有加密或认证）端口和集群的 apiserver 通信。这个端口通常只在 master 节点的 localhost 接口暴露，这样，所有在相同机器上运行的 master 组件就能和集群的 apiserver 通信。一段时间以后，master 组件将变为使用带身份认证和权限验证的安全端口（查看[#13598](https://github.com/kubernetes/kubernetes/issues/13598)）。

这样的结果使得从集群（在节点上运行的 nodes 和 pods）到 master 的缺省连接操作模式默认被保护，能够在不可信或公网中运行。

Master -> Cluster

从 master（apiserver）到集群有两种主要的通信路径。第一种是从 apiserver 到集群中每个节点上运行的 kubelet 进程。第二种是从 apiserver 通过它的代理功能到任何 node、pod 或者 service。

apiserver -> kubelet

从 apiserver 到 kubelet 的连接用于获取 pods 日志、连接（通过 kubectl）运行中的 pods，以及使用 kubelet 的端口转发功能。这些连接终止于 kubelet 的 HTTPS endpoint。

默认的，apiserver 不会验证 kubelet 的服务证书，这会导致连接遭到中间人攻击，因而在不可信或公共网络上是不安全的。

为了对这个连接进行认证，请使用 --kubelet-certificate-authority 标记给 apiserver 提供一个根证书捆绑，用于 kubelet 的服务证书。

如果这样不可能，又要求避免在不可信的或公共的网络上进行连接，请在 apiserver 和 kubelet 之间使用 [SSH 隧道](https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/)。

最后，应该启用 [Kubelet 用户认证和/或权限认证](https://kubernetes.io/docs/admin/kubelet-authentication-authorization/)来保护 kubelet API。

apiserver -> nodes, pods, and services

从 apiserver 到 node、pod 或者 service 的连接默认为纯 HTTP 方式，因此既没有认证，也没有加密。他们能够通过给 API URL 中的 node、pod 或 service 名称添加前缀 https: 来运行在安全的 HTTPS 连接上。但他们即不会认证 HTTPS endpoint 提供的证书，也不会提供客户端证书。这样虽然连接是加密的，但它不会提供任何完整性保证。这些连接**目前还不能安全的**在不可信的或公共的网络上运行。

SSH 隧道

[Google Kubernetes Engine](https://cloud.google.com/kubernetes-engine/docs/) 使用 SSH 隧道保护 Master -> Cluster 通信路径。在这种配置下，apiserver 发起一个到集群中每个节点的 SSH 隧道（连接到在 22 端口监听的 ssh 服务）并通过这个隧道传输所有到 kubelet、node、pod 或者 service 的流量。这个隧道保证流量不会在集群运行的私有 GCE 网络之外暴露。

# 计算、存储和网络扩展

集群管理概述

集群管理概述面向任何创建和管理 Kubernetes 集群的读者人群。 我们假设你对[用户指南](https://kubernetes.io/docs/user-guide/)中的概念大概了解。

* [**规划集群**](https://kubernetes.io/zh/docs/concepts/cluster-administration/#%e8%a7%84%e5%88%92%e9%9b%86%e7%be%a4)
* [**管理集群**](https://kubernetes.io/zh/docs/concepts/cluster-administration/#%e7%ae%a1%e7%90%86%e9%9b%86%e7%be%a4)
* [**集群安全**](https://kubernetes.io/zh/docs/concepts/cluster-administration/#%e9%9b%86%e7%be%a4%e5%ae%89%e5%85%a8)
* [**可选集群服务**](https://kubernetes.io/zh/docs/concepts/cluster-administration/#%e5%8f%af%e9%80%89%e9%9b%86%e7%be%a4%e6%9c%8d%e5%8a%a1)

规划集群

查阅 [安装](https://kubernetes.io/docs/setup/) 中的指导，获取如何规划、建立以及配置 Kubernetes 集群的示例。本文所列的文章称为*发行版* 。

在选择一个指南前，有一些因素需要考虑：

* 你是打算在你的电脑上尝试 Kubernetes，还是要构建一个高可用的多节点集群？请选择最适合你需求的发行版。
* **如果你正在设计一个高可用集群**，请了解[在多个 zones 中配置集群](https://kubernetes.io/docs/concepts/cluster-administration/federation/)。
* 您正在使用 类似 [Google Kubernetes Engine](https://cloud.google.com/kubernetes-engine/) 这样的**被托管的Kubernetes集群**, 还是**管理您自己的集群**?
* 你的集群是在**本地**还是**云（IaaS）**上？ Kubernetes 不能直接支持混合集群。作为代替，你可以建立多个集群。
* **如果你在本地配置 Kubernetes**，需要考虑哪种[网络模型](https://kubernetes.io/docs/concepts/cluster-administration/networking/)最适合。
* 你的 Kubernetes 在 **裸金属硬件** 还是 **虚拟机（VMs）**上运行？
* 你**只想运行一个集群**，还是打算**活动开发 Kubernetes 项目代码**？如果是后者，请选择一个活动开发的发行版。某些发行版只提供二进制发布版，但提供更多的选择。
* 让你自己熟悉运行一个集群所需的[组件](https://kubernetes.io/docs/admin/cluster-components) 。

请注意：不是所有的发行版都被积极维护着。请选择测试过最近版本的 Kubernetes 的发行版。

管理集群

* [管理集群](https://kubernetes.io/docs/concepts/cluster-administration/cluster-management/)叙述了和集群生命周期相关的几个主题：创建一个新集群、升级集群的 master 和 worker 节点、执行节点维护（例如内核升级）以及升级活动集群的 Kubernetes API 版本。
* 学习如何 [管理节点](https://kubernetes.io/docs/concepts/nodes/node/).
* 学习如何设定和管理集群共享的 [资源配额](https://kubernetes.io/docs/concepts/policy/resource-quotas/) 。

集群安全

* [Certificates](https://kubernetes.io/docs/concepts/cluster-administration/certificates/) 描述了使用不同的工具链生成证书的步骤。
* [Kubernetes 容器环境](https://kubernetes.io/docs/concepts/containers/container-environment-variables/) 描述了 Kubernetes 节点上由 Kubelet 管理的容器的环境。
* [控制到 Kubernetes API 的访问](https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/)描述了如何为用户和 service accounts 建立权限许可。
* [用户认证](https://kubernetes.io/docs/reference/access-authn-authz/authentication/)阐述了 Kubernetes 中的认证功能，包括许多认证选项。
* [授权](https://kubernetes.io/docs/admin/authorization)从认证中分离出来，用于控制如何处理 HTTP 请求。
* [使用 Admission Controllers](https://kubernetes.io/docs/admin/admission-controllers) 阐述了在认证和授权之后拦截到 Kubernetes API 服务的请求的插件。
* [在 Kubernetes Cluster 中使用 Sysctls](https://kubernetes.io/docs/concepts/cluster-administration/sysctl-cluster/) 描述了管理员如何使用 sysctl 命令行工具来设置内核参数。
* [审计](https://kubernetes.io/docs/tasks/debug-application-cluster/audit/)描述了如何与 Kubernetes 的审计日志交互。

保护 kubelet

* [Master 节点通信](https://kubernetes.io/docs/concepts/cluster-administration/master-node-communication/)
* [TLS 引导](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/)
* [Kubelet 认证/授权](https://kubernetes.io/docs/admin/kubelet-authentication-authorization/)

可选集群服务

* [DNS 与 SkyDNS 集成](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)描述了如何将一个 DNS 名解析到一个 Kubernetes service。
* [记录和监控集群活动](https://kubernetes.io/docs/concepts/cluster-administration/logging/)阐述了 Kubernetes 的日志如何工作以及怎样实现。

证书

当使用客户端证书进行认证时，用户可以使用现有部署脚本，或者通过 easyrsa、openssl 或 cfssl 手动生成证书。

* [**分发自签名 CA 证书**](https://kubernetes.io/zh/docs/concepts/cluster-administration/certificates/#%e5%88%86%e5%8f%91%e8%87%aa%e7%ad%be%e5%90%8d-ca-%e8%af%81%e4%b9%a6)
* [**证书 API**](https://kubernetes.io/zh/docs/concepts/cluster-administration/certificates/#%e8%af%81%e4%b9%a6-api)

easyrsa

使用 **easyrsa** 能够手动地为集群生成证书。

1. 下载、解压并初始化 easyrsa3 的补丁版本。
2. curl -LO https://storage.googleapis.com/kubernetes-release/easy-rsa/easy-rsa.tar.gz
3. tar xzf easy-rsa.tar.gz
4. cd easy-rsa-master/easyrsa3
5. ./easyrsa init-pki
6. 生成 CA（通过 --batch 参数设置自动模式。 通过 --req-cn 设置默认使用的 CN）
7. ./easyrsa --batch "--req-cn=${MASTER\_IP}@`date +%s`" build-ca nopass
8. 生成服务器证书和密钥。 参数 --subject-alt-name 设置了访问 API 服务器时可能使用的 IP 和 DNS 名称。 MASTER\_CLUSTER\_IP 通常为 --service-cluster-ip-range 参数中指定的服务 CIDR 的 首个 IP 地址，--service-cluster-ip-range 同时用于 API 服务器和控制器管理器组件。 --days 参数用于设置证书的有效期限。 下面的示例还假设用户使用 cluster.local 作为默认的 DNS 域名。
9. ./easyrsa --subject-alt-name="IP:${MASTER\_IP},"\
10. "IP:${MASTER\_CLUSTER\_IP},"\
11. "DNS:kubernetes,"\
12. "DNS:kubernetes.default,"\
13. "DNS:kubernetes.default.svc,"\
14. "DNS:kubernetes.default.svc.cluster,"\
15. "DNS:kubernetes.default.svc.cluster.local" \
16. --days=10000 \
17. build-server-full server nopass
18. 拷贝 pki/ca.crt、 pki/issued/server.crt 和 pki/private/server.key 至您的目录。
19. 填充并在 API 服务器的启动参数中添加以下参数：
20. --client-ca-file=/yourdirectory/ca.crt
21. --tls-cert-file=/yourdirectory/server.crt
22. --tls-private-key-file=/yourdirectory/server.key

openssl

使用 **openssl** 能够手动地为集群生成证书。

1. 生成密钥位数为 2048 的 ca.key：
2. openssl genrsa -out ca.key 2048
3. 依据 ca.key 生成 ca.crt （使用 -days 参数来设置证书有效时间）：
4. openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER\_IP}" -days 10000 -out ca.crt
5. 生成密钥位数为 2048 的 server.key：
6. openssl genrsa -out server.key 2048
7. 创建用于生成证书签名请求（CSR）的配置文件。 确保在将其保存至文件（如 csr.conf）之前将尖括号标记的值（如 <MASTER\_IP>） 替换为你想使用的真实值。 注意：MASTER\_CLUSTER\_IP 是前面小节中描述的 API 服务器的服务集群 IP (service cluster IP)。 下面的示例也假设用户使用 cluster.local 作为默认的 DNS 域名。
8. [ req ]
9. default\_bits = 2048
10. prompt = no
11. default\_md = sha256
12. req\_extensions = req\_ext
13. distinguished\_name = dn
14. [ dn ]
15. C = <country>
16. ST = <state>
17. L = <city>
18. O = <organization>
19. OU = <organization unit>
20. CN = <MASTER\_IP>
21. [ req\_ext ]
22. subjectAltName = @alt\_names
23. [ alt\_names ]
24. DNS.1 = kubernetes
25. DNS.2 = kubernetes.default
26. DNS.3 = kubernetes.default.svc
27. DNS.4 = kubernetes.default.svc.cluster
28. DNS.5 = kubernetes.default.svc.cluster.local
29. IP.1 = <MASTER\_IP>
30. IP.2 = <MASTER\_CLUSTER\_IP>
31. [ v3\_ext ]
32. authorityKeyIdentifier=keyid,issuer:always
33. basicConstraints=CA:FALSE
34. keyUsage=keyEncipherment,dataEncipherment
35. extendedKeyUsage=serverAuth,clientAuth
36. subjectAltName=@alt\_names
37. 基于配置文件生成证书签名请求：
38. openssl req -new -key server.key -out server.csr -config csr.conf
39. 使用 ca.key、ca.crt 和 server.csr 生成服务器证书：
40. openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
41. -CAcreateserial -out server.crt -days 10000 \
42. -extensions v3\_ext -extfile csr.conf
43. 查看证书：
44. openssl x509 -noout -text -in ./server.crt

最后，添加同样的参数到 API 服务器的启动参数中。

cfssl

**cfssl** 是另一种用来生成证书的工具。

1. 按如下所示的方式下载、解压并准备命令行工具。 注意：你可能需要基于硬件架构和你所使用的 cfssl 版本对示例命令进行修改。
2. curl -L https://pkg.cfssl.org/R1.2/cfssl\_linux-amd64 -o cfssl
3. chmod +x cfssl
4. curl -L https://pkg.cfssl.org/R1.2/cfssljson\_linux-amd64 -o cfssljson
5. chmod +x cfssljson
6. curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo\_linux-amd64 -o cfssl-certinfo
7. chmod +x cfssl-certinfo
8. 创建目录来存放物料，并初始化 cfssl：
9. mkdir cert
10. cd cert
11. ../cfssl print-defaults config > config.json
12. ../cfssl print-defaults csr > csr.json
13. 创建用来生成 CA 文件的 JSON 配置文件，例如 ca-config.json：
14. {
15. "signing": {
16. "default": {
17. "expiry": "8760h"
18. },
19. "profiles": {
20. "kubernetes": {
21. "usages": [
22. "signing",
23. "key encipherment",
24. "server auth",
25. "client auth"
26. ],
27. "expiry": "8760h"
28. }
29. }
30. }
31. }
32. 创建用来生成 CA 证书签名请求（CSR）的 JSON 配置文件，例如 ca-csr.json。 确保将尖括号标记的值替换为你想使用的真实值。
33. {
34. "CN": "kubernetes",
35. "key": {
36. "algo": "rsa",
37. "size": 2048
38. },
39. "names":[{
40. "C": "<country>",
41. "ST": "<state>",
42. "L": "<city>",
43. "O": "<organization>",
44. "OU": "<organization unit>"
45. }]
46. }
47. 生成 CA 密钥（ca-key.pem）和证书（ca.pem）：
48. ../cfssl gencert -initca ca-csr.json | ../cfssljson -bare ca
49. 按如下所示的方式创建用来为 API 服务器生成密钥和证书的 JSON 配置文件。 确保将尖括号标记的值替换为你想使用的真实值。 MASTER\_CLUSTER\_IP 是前面小节中描述的 API 服务器的服务集群 IP。 下面的示例也假设用户使用 cluster.local 作为默认的 DNS 域名。
50. {
51. "CN": "kubernetes",
52. "hosts": [
53. "127.0.0.1",
54. "<MASTER\_IP>",
55. "<MASTER\_CLUSTER\_IP>",
56. "kubernetes",
57. "kubernetes.default",
58. "kubernetes.default.svc",
59. "kubernetes.default.svc.cluster",
60. "kubernetes.default.svc.cluster.local"
61. ],
62. "key": {
63. "algo": "rsa",
64. "size": 2048
65. },
66. "names": [{
67. "C": "<country>",
68. "ST": "<state>",
69. "L": "<city>",
70. "O": "<organization>",
71. "OU": "<organization unit>"
72. }]
73. }
74. 为 API 服务器生成密钥和证书，生成的秘钥和证书分别默认保存在文件 server-key.pem 和 server.pem 中：
75. ../cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
76. --config=ca-config.json -profile=kubernetes \
77. server-csr.json | ../cfssljson -bare server

分发自签名 CA 证书

客户端节点可能拒绝承认自签名 CA 证书有效。 对于非生产环境的部署，或运行在企业防火墙后的部署，用户可以向所有客户端分发自签名 CA 证书， 并刷新本地的有效证书列表。

在每个客户端上执行以下操作：

sudo cp ca.crt /usr/local/share/ca-certificates/kubernetes.crt

sudo update-ca-certificates

Updating certificates in /etc/ssl/certs...

1 added, 0 removed; done.

Running hooks in /etc/ca-certificates/update.d....

done.

证书 API

您可以按照[这里](https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster)记录的方式， 使用 certificates.k8s.io API 来准备 x509 证书，用于认证。

## 云驱动

本文介绍了如何管理运行在特定云驱动上的 Kubernetes 集群。

* [**Azure**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#azure)
* [**CloudStack**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#cloudstack)
* [**GCE**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#gce)
* [**OpenStack**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#openstack)
* [**OVirt**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#ovirt)
* [**Photon**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#photon)
* [**VSphere**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#vsphere)
* [**IBM Cloud Kubernetes Service**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#ibm-cloud-kubernetes-service)
* [**百度云容器引擎**](https://kubernetes.io/zh/docs/concepts/cluster-administration/cloud-providers/#%e7%99%be%e5%ba%a6%e4%ba%91%e5%ae%b9%e5%99%a8%e5%bc%95%e6%93%8e)

### kubeadm

[kubeadm](https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm/) 是创建 kubernetes 集群的一种流行选择。 kubeadm 通过提供配置选项来指定云驱动的配置信息。例如，一个典型的适用于“树内”云驱动的 kubeadm 配置如下：

apiVersion: kubeadm.k8s.io/v1beta2

kind: InitConfiguration

nodeRegistration:

kubeletExtraArgs:

cloud-provider: "openstack"

cloud-config: "/etc/kubernetes/cloud.conf"

---

apiVersion: kubeadm.k8s.io/v1beta2

kind: ClusterConfiguration

kubernetesVersion: v1.13.0

apiServer:

extraArgs:

cloud-provider: "openstack"

cloud-config: "/etc/kubernetes/cloud.conf"

extraVolumes:

- name: cloud

hostPath: "/etc/kubernetes/cloud.conf"

mountPath: "/etc/kubernetes/cloud.conf"

controllerManager:

extraArgs:

cloud-provider: "openstack"

cloud-config: "/etc/kubernetes/cloud.conf"

extraVolumes:

- name: cloud

hostPath: "/etc/kubernetes/cloud.conf"

mountPath: "/etc/kubernetes/cloud.conf"

“树内”的云驱动通常需要在命令行中为 [kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/)、[kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/) 和 [kubelet](https://kubernetes.io/docs/admin/kubelet/) 指定 --cloud-provider 和 --cloud-config。在 --cloud-config 中为每个供应商指定的文件的内容也同样需要写在下面。 对于所有外部云驱动，请遵循独立云存储库的说明，或浏览[所有版本库清单](https://github.com/kubernetes?q=cloud-provider-&type=&language=)

### AWS

本节介绍在 Amazon Web Services 上运行 Kubernetes 时可以使用的所有配置。 如果希望使用此外部云驱动，其代码库位于 [kubernetes/cloud-provider-aws](https://github.com/kubernetes/cloud-provider-aws#readme)

### 节点名称

云驱动 AWS 使用 AWS 实例的私有 DNS 名称作为 Kubernetes 节点对象的名称。

### 负载均衡器

用户可以通过配置注解（annotations）来设置 [外部负载均衡器](https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/)，以在 AWS 中使用特定功能，如下所示：

apiVersion: v1

kind: Service

metadata:

name: example

namespace: kube-system

labels:

run: example

annotations:

service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:xx-xxxx-x:xxxxxxxxx:xxxxxxx/xxxxx-xxxx-xxxx-xxxx-xxxxxxxxx *#replace this value*

service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http

spec:

type: LoadBalancer

ports:

- port: 443

targetPort: 5556

protocol: TCP

selector:

app: example

可以使用 注解 将不同的设置应用于 AWS 中的负载均衡器服务。下面描述了 AWS ELB 所支持的注解：

* service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval：用于指定访问日志的间隔。
* service.beta.kubernetes.io/aws-load-balancer-access-log-enabled：用于在服务中启用或禁用访问日志。
* service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name：用于指定访问日志的 S3 桶名称。
* service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix：用于指定访问日志的 S3 桶前缀。
* service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags：用于在服务中指定一个逗号分隔的键值对列表，它将作为附加标签被记录在 ELB 中。例如： "Key1=Val1,Key2=Val2,KeyNoVal1=,KeyNoVal2"。
* service.beta.kubernetes.io/aws-load-balancer-backend-protocol：用于在服务中指定监听器后端（pod）所使用的协议。如果指定 http（默认）或 https，将创建一个终止连接和解析头的 HTTPS 监听器。 如果设置为 ssl 或 tcp，将会使用 “原生的” SSL 监听器。如果设置为 http且不使用 aws-load-balancer-ssl-cert，将使用 HTTP 监听器。
* service.beta.kubernetes.io/aws-load-balancer-ssl-cert：用于在服务中请求安全监听器，其值为合法的证书 ARN（Amazon Resource Name）。更多内容，请参考 [ELB 监听器配置](http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/elb-listener-config.html)。证书 ARN 是 IAM（身份和访问管理）或 CM（证书管理）类型的 ARN，例如 arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012。
* service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled：用于在服务中启用或禁用连接耗尽（connection draining）。
* service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout：用于在服务中指定连接耗尽超时时间。
* service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout：用于在服务中指定空闲连接超时时间。
* service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled：用于在服务中启用或禁用跨区域负载平衡。
* service.beta.kubernetes.io/aws-load-balancer-extra-security-groups：用于在服务中指定要添加到创建的 ELB 中的其他安全组。
* service.beta.kubernetes.io/aws-load-balancer-internal：用于在服务中表明需要内部 ELB。
* service.beta.kubernetes.io/aws-load-balancer-proxy-protocol：用于在 ELB 上启用代理协议。 当前仅接受 \*值，也就是在所有 ELB 后端启用代理协议。将来可能进行调整，只允许特定的后端设置代理协议。
* service.beta.kubernetes.io/aws-load-balancer-ssl-ports：用于在服务中指定一个逗号分隔的端口列表，这些端口会使用 SSL/HTTPS 监听器。默认为 \*（全部）

AWS 相关的注解信息取自 [aws.go](https://github.com/kubernetes/cloud-provider-aws/blob/master/pkg/cloudprovider/providers/aws/aws.go) 文件的注释。

### Azure

如果希望使用此外部云驱动，其代码库位于 [kubernetes/cloud-provider-azure](https://github.com/kubernetes/cloud-provider-azure#readme)

### 节点名称

云驱动 Azure 使用节点的主机名（由 kubelet 决定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意 Kubernetes 节点名必须与 Azure 虚拟机的名称匹配。

### CloudStack

如果希望使用此外部云驱动，其代码库位于 [apache/cloudstack-kubernetes-provider](https://github.com/apache/cloudstack-kubernetes-provider)。

### 节点名称

云驱动 CloudStack 使用节点的主机名（由 kubelet 决定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意 Kubernetes 节点名必须与 CloudStack 虚拟机名匹配。

### GCE

如果希望使用此外部云驱动，其代码库位于 [kubernetes/cloud-provider-gcp](https://github.com/kubernetes/cloud-provider-gcp#readme)

### 节点名称

GCE 云驱动使用节点的主机名（由 kubelet 确定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意，Kubernetes 节点名的第一个字段必须匹配 GCE 实例名(例如，名为 kubernetes-node-2.c.my-proj.internal 的节点必须对应于一个名为 kubernetes-node-2 的实例)。

### OpenStack

本节介绍了使用 OpenStack 运行 Kubernetes 时所有可用的配置。 如果希望使用此外部云驱动，其代码库位于 [kubernetes/cloud-provider-openstack](https://github.com/kubernetes/cloud-provider-openstack#readme)

### 节点名称

OpenStack 云驱动使用实例名（由 OpenStack 元数据确定）作为 Kubernetes 节点对象的名称。 请注意，实例名必须是一个有效的 Kubernetes 节点名，以便 kubelet 成功注册其节点对象。

### 服务

Kubernetes 的 OpenStack 云驱动实现支持从底层云使用这些 OpenStack 服务：

| 服务 | API 版本 | 必需 |
| --- | --- | --- |
| 块存储 (Cinder) | V1†, V2, V3 | No |
| 计算 (Nova) | V2 | No |
| 身份认证 (Keystone) | V2‡, V3 | Yes |
| 负载均衡器 (Neutron) | V1§, V2 | No |
| 负载均衡器 (Octavia) | V2 | No |

† Block Storage V1 版本的 API 被弃用，从 Kubernetes 1.9 版本开始加入了 Block Storage V3 版本 API。

‡ Identity V2 API 支持已被弃用，将在未来的版本中从供应商中移除。从 “Queens” 版本开始，OpenStack 将不再支持 Identity V2 版本的 API。

§ Kubernetes 1.9 中取消了对 V1 版本 Load Balancing API 的支持。

服务发现是通过使用供应商配置中提供的 auth-url 所列出 OpenStack 身份认证（Keystone）管理的服务目录来实现的。 当除 Keystone 外的 OpenStack 服务不可用时，供应商将优雅地降低功能，并简单地放弃对受影响特性的支持。 某些功能还可以根据 Neutron 在底层云中发布的扩展列表启用或禁用。

### cloud.conf

Kubernetes 知道如何通过 cloud.conf 文件与 OpenStack 交互。该文件将为 Kubernetes 提供 OpenStack 验证端点的凭据和位置。 用户可在创建 cloud.conf 文件时指定以下信息：

#### 典型配置

下面是一个典型配置的例子，它涉及到最常设置的值。它将供应商指向 OpenStack 云的 Keystone 端点，提供如何使用它进行身份验证的细节，并配置负载均衡器:

[Global]

username=user

password=pass

auth-url=https://<keystone\_ip>/identity/v3

tenant-id=c869168a828847f39f7f06edd7305637

domain-id=2a73b8f597c04551a0fdc8e95544be8a

[LoadBalancer]

subnet-id=6937f8fa-858d-4bc9-a3a5-18d2c957166a

##### 全局配置

这些配置选项属于 OpenStack 驱动的全局配置，并且应该出现在 cloud.conf 文件中的 [global] 部分:

* auth-url (必需): 用于认证的 keystone API 的 URL。在 OpenStack 控制面板中，这可以在“访问和安全（Access and Security）> API 访问（API Access）> 凭证（Credentials）”中找到。
* username (必需): 指 keystone 中一个有效用户的用户名。
* password (必需): 指 keystone 中一个有效用户的密码。
* tenant-id (必需): 用于指定要创建资源的租户 ID。
* tenant-name (可选): 用于指定要在其中创建资源的租户的名称。
* trust-id (可选): 用于指定用于授权的信任的标识符。信任表示用户（委托人）将角色委托给另一个用户(受托人)的授权，并可选的允许受托人模仿委托人。可用的信任可以在 Keystone API 的 /v3/OS-TRUST/trusts 端点下找到。
* domain-id (可选): 用于指定用户所属域的 ID。
* domain-name (可选): 用于指定用户所属域的名称。
* region (可选): 用于指定在多区域 OpenStack 云上运行时使用的区域标识符。区域是 OpenStack 部署的一般性划分。虽然区域没有严格的地理含义，但部署可以使用地理名称表示区域标识符，如 us-east。可用区域位于 Keystone API 的 /v3/regions 端点之下。
* ca-file (可选): 用于指定自定义 CA 文件的路径。

当使用 Keystone V3 时(它将tenant更改为project)，tenant-id 值会自动映射到 API 中的项目。

#### 负载均衡器

这些配置选项属于 OpenStack 驱动的全局配置，并且应该出现在 cloud.conf 文件中的 [LoadBalancer] 部分:

* lb-version (可选): 用于覆盖自动版本检测。有效值为 v1 或 v2。如果没有提供值，则自动选择底层 OpenStack 云所支持的最高版本。
* use-octavia (可选): 用于确定是否查找和使用 Octavia LBaaS V2 服务目录端点。有效值是 true 或 false。 如果指定了“true”，并且无法找到 Octaiva LBaaS V2 入口，则提供者将退回并尝试寻找一个 Neutron LBaaS V2 端点。默认值是 false。
* subnet-id (可选): 用于指定要在其上创建负载均衡器的子网的 ID。 可以在 “Network > Networks” 上找到。 单击相应的网络以获得其子网。
* floating-network-id (可选): 如果指定，将为负载均衡器创建一个浮动 IP。
* lb-method (可选): 用于指定将负载分配到负载均衡器池成员的算法。值可以是 ROUND\_ROBIN、LEAST\_CONNECTIONS 或 SOURCE\_IP。如果没有指定，默认行为是 ROUND\_ROBIN。
* lb-provider (可选): 用于指定负载均衡器的提供程序。如果没有指定，将使用在 Neutron 中配置的默认提供者服务。
* create-monitor (可选): 指定是否为 Neutron 负载均衡器创建健康监视器。有效值是 true 和 false。 默认为 false。当指定 true 时，还必须设置 monitor-delay、monitor-timeout 和 monitor-max-retries。
* monitor-delay (可选): 向负载均衡器的成员发送探测之间的时间间隔。 确保您指定了一个有效的时间单位。 有效时间单位为 ns、us (或 µs)、ms、s、m、h。
* monitor-timeout (可选): 在超时之前，监视器等待 ping 响应的最长时间。该值必须小于延迟值。确保您指定了一个有效的时间单位。有效时间单位为 ns、 us (或 µs)、ms、s、m、 h。
* monitor-max-retries (可选): 在将负载均衡器成员的状态更改为非活动之前，允许 ping 失败的次数。 必须是 1 到 10 之间的数字。
* manage-security-groups (可选): 确定负载均衡器是否应自动管理安全组规则。有效值是 true 和 false。默认为 false。当指定 true 时，还必须提供 node-security-group。
* node-security-group (可选): 要管理的安全组的 ID。

##### 块存储

这些配置选项属于 OpenStack 驱动的全局配置，并且应该出现在 cloud.conf 文件中的 [BlockStorage] 部分：

* bs-version (可选): 指所使用的块存储 API 版本。其合法值为 v1、v2、v3和 auto。 auto 为默认值，将使用底层 Openstack 所支持的块存储 API 的最新版本。
* trust-device-path (可选): 在大多数情况下，块设备名称由 Cinder 提供（例如：/dev/vda）不可信任。此布尔值切换此行为。将其设置为 true 将导致信任 Cinder 提供的块设备名称。默认值 false 会根据设备序列号和 /dev/disk/by-id 映射发现设备路径，推荐这种方法。
* ignore-volume-az (可选): 用于在附加 Cinder 卷时影响可用区使用。 当 Nova 和 Cinder 有不同的可用区域时，应该将其设置为 true。 最常见的情况是，有许多 Nova 可用区，但只有一个 Cinder 可用区。 默认值是 false，以保持在早期版本中使用的行为，但是将来可能会更改。
* node-volume-attach-limit (可选): 可连接到节点的最大卷数，对于 Cinder 默认为 256。

如果在 OpenStack 上部署 Kubernetes <= 1.8 的版本，同时使用路径而不是端口来区分端点（Endpoints），那么可能需要显式设置 bs-version 参数。 基于路径的端点形如 http://foo.bar/volume，而基于端口的的端点形如 http://foo.bar:xxx。

在使用基于路径的端点，并且 Kubernetes 使用较旧的自动检索逻辑的环境中，尝试卷卸载（Detachment）会返回 BS API version autodetection failed. 错误。为了解决这个问题，可以通过添加以下内容到云驱动配置中，来强制使用 Cinder API V2 版本。

[BlockStorage]

bs-version=v2

##### 元数据

这些配置选项属于 OpenStack 提供程序的全局配置，并且应该出现在 cloud.conf 文件中的 [Metadata] 部分：

* search-order (可选): 此配置键影响提供者检索与其运行的实例相关的元数据的方式。 configDrive，metadataService 的默认值导致供应商首先从配置驱动器中检索与实例相关的元数据（如果可用的话），然后检索元数据服务。 他们的替代值：
  + configDrive - 仅从配置驱动器检索实例元数据。
  + metadataService - 仅从元数据服务检索实例元数据。
  + metadataService,configDrive - 如果可用，首先从元数据服务检索实例元数据，然后从配置驱动器检索。

影响这种行为可能是可取的，因为配置驱动器上的元数据可能会随着时间的推移而变得陈旧，而元数据服务总是提供最新的数据视图。并不是所有的 OpenStack 云都同时提供配置驱动和元数据服务，可能只有一个或另一个可用，这就是为什么默认情况下要同时检查两个。

##### 路由

这些配置选项属于 OpenStack 驱动为 Kubernetes 网络插件 [kubenet](https://kubernetes.io/docs/concepts/cluster-administration/network-plugins/#kubenet) 提供的设置，并且应该出现在 cloud.conf 文件中的 [Route] 部分:

* router-id (可选)：如果底层云的 Neutron 部署支持 extraroutes 扩展，则使用 router-id 指定要添加路由的路由器。选择的路由器必须跨越包含集群节点的私有网络（通常只有一个节点网络，这个值应该是节点网络的默认路由器）。在 OpenStack 上使用 [kubenet](https://kubernetes.io/docs/concepts/cluster-administration/network-plugins/#kubenet) 时需要这个值。

### OVirt

### 节点名称

OVirt 云驱动使用节点的主机名（由 kubelet 确定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意 Kubernetes 节点名必须与 VM FQDN 匹配（OVirt 在<vm><guest\_info><fqdn>...</fqdn></guest\_info></vm>中说明）。

### Photon

### 节点名称

Photon 云驱动使用节点的主机名（由 kubelet 决定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意，Kubernetes 节点名必须与 Photon VM名匹配（或者，如果在 --cloud-config 中将 overrideIP 设置为 true，则 Kubernetes 节点名必须与 Photon VM IP 地址匹配）。

### VSphere

### 节点名称

VSphere 云驱动使用节点检测到的主机名(由 kubelet 确定)作为 Kubernetes 节点对象的名称。 VSphere 云驱动会忽略 --hostname-override 参数。

### IBM Cloud Kubernetes Service

### 计算节点

通过使用 IBM Cloud Kubernetes Service 驱动，您可以在单个区域或跨区域的多个区（Region）中创建虚拟和物理（裸金属）节点的集群。 有关更多信息，请参见[规划您的集群和工作节点设置](https://cloud.ibm.com/docs/containers?topic=containers-plan_clusters#plan_clusters)。

Kubernetes 节点对象的名称是 IBM Cloud Kubernetes Services 工作节点实例的私有IP地址。

### 网络

IBM Cloud Kubernetes Services 驱动提供 VLAN，用于提供高质量的网络性能和节点间的网络隔离。您可以设置自定义防火墙和 Calico 网络策略来为您的集群添加额外的安全层，或者通过 VPN 将您的集群连接到自有数据中心。有关更多信息，请参见[规划集群内和私有网络](https://cloud.ibm.com/docs/containers?topic=containers-cs_network_cluster#cs_network_cluster)。

要向公众或集群内部公开应用程序，您可以利用 NodePort、LoadBalancer 或 Ingress 服务。您还可以使用注释自定义 Ingress 应用程序负载均衡器。有关更多信息，请参见[计划使用外部网络公开您的应用程序](https://cloud.ibm.com/docs/containers?topic=containers-cs_network_planning#cs_network_planning)。

### 存储

IBM Cloud Kubernetes Services 驱动利用 Kubernetes 原生的持久卷，使用户能够将文件、块和云对象存储装载到他们的应用程序中。还可以使用 database-as-a-service 和第三方附加组件来持久存储数据。有关更多信息，请参见[规划高可用性持久存储](https://cloud.ibm.com/docs/containers?topic=containers-storage_planning#storage_planning)。

### 百度云容器引擎

### 节点名称

Baidu 云驱动使用节点的私有 IP 地址（由 kubelet 确定，或者用 --hostname-override 覆盖）作为 Kubernetes 节点对象的名称。 注意 Kubernetes 节点名必须匹配百度 VM 的私有 IP。

管理资源

您已经部署了应用并通过服务暴露它。然后呢？Kubernetes 提供了一些工具来帮助管理您的应用部署，包括缩扩容和更新。我们将更深入讨论的特性包括[配置文件](https://kubernetes.io/docs/concepts/configuration/overview/)和[标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)。

* [**组织资源配置**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e7%bb%84%e7%bb%87%e8%b5%84%e6%ba%90%e9%85%8d%e7%bd%ae)
* [**kubectl 中的批量操作**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#kubectl-%e4%b8%ad%e7%9a%84%e6%89%b9%e9%87%8f%e6%93%8d%e4%bd%9c)
* [**有效地使用标签**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e6%9c%89%e6%95%88%e5%9c%b0%e4%bd%bf%e7%94%a8%e6%a0%87%e7%ad%be)
* [**金丝雀部署**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e9%87%91%e4%b8%9d%e9%9b%80%e9%83%a8%e7%bd%b2)
* [**更新标签**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e6%9b%b4%e6%96%b0%e6%a0%87%e7%ad%be)
* [**更新注解**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e6%9b%b4%e6%96%b0%e6%b3%a8%e8%a7%a3)
* [**缩扩您的应用**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e7%bc%a9%e6%89%a9%e6%82%a8%e7%9a%84%e5%ba%94%e7%94%a8)
* [**就地更新资源**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e5%b0%b1%e5%9c%b0%e6%9b%b4%e6%96%b0%e8%b5%84%e6%ba%90)
* [**破坏性的更新**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e7%a0%b4%e5%9d%8f%e6%80%a7%e7%9a%84%e6%9b%b4%e6%96%b0)
* [**在不中断服务的情况下更新应用**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e5%9c%a8%e4%b8%8d%e4%b8%ad%e6%96%ad%e6%9c%8d%e5%8a%a1%e7%9a%84%e6%83%85%e5%86%b5%e4%b8%8b%e6%9b%b4%e6%96%b0%e5%ba%94%e7%94%a8)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/cluster-administration/manage-deployment/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

组织资源配置

许多应用需要创建多个资源，例如 Deployment 和 Service。可以通过将多个资源组合在同一个文件中（在 YAML 中以 --- 分隔）来简化对它们的管理。例如：

| [application/nginx-app.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/application/nginx-app.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-nginx-svc  labels:  app: nginx  spec:  type: LoadBalancer  ports:  - port: 80  selector:  app: nginx  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: my-nginx  labels:  app: nginx  spec:  replicas: 3  selector:  matchLabels:  app: nginx  template:  metadata:  labels:  app: nginx  spec:  containers:  - name: nginx  image: nginx:1.7.9  ports:  - containerPort: 80 |

可以用创建单个资源相同的方式来创建多个资源：

kubectl apply -f https://k8s.io/examples/application/nginx-app.yaml

service/my-nginx-svc created

deployment.apps/my-nginx created

资源将按照它们在文件中的顺序创建。因此，最好先指定服务，这样在控制器（例如 Deployment）创建 Pod 时能够确保调度器可以将与服务关联的多个 Pod 分散到不同节点。

kubectl create 也接受多个 -f 参数:

kubectl apply -f https://k8s.io/examples/application/nginx/nginx-svc.yaml -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml

还可以指定目录路径，而不用添加多个单独的文件：

kubectl apply -f https://k8s.io/examples/application/nginx/

kubectl 将读取任何后缀为 .yaml，.yml 或者 .json 的文件。

建议的做法是，将同一个微服务或同一应用层相关的资源放到同一个文件中，将同一个应用相关的所有文件按组存放到同一个目录中。如果应用的各层使用 DNS 相互绑定，那么您可以简单地将堆栈的所有组件一起部署。

还可以使用 URL 作为配置源，便于直接使用已经提交到 Github 上的配置文件进行部署：

kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/application/nginx/nginx-deployment.yaml

deployment.apps/my-nginx created

kubectl 中的批量操作

资源创建并不是 kubectl 可以批量执行的唯一操作。kubectl 还可以从配置文件中提取资源名，以便执行其他操作，特别是删除您之前创建的资源：

kubectl delete -f https://k8s.io/examples/application/nginx-app.yaml

deployment.apps "my-nginx" deleted

service "my-nginx-svc" deleted

在仅有两种资源的情况下，可以使用”资源类型/资源名”的语法在命令行中同时指定这两个资源：

kubectl delete deployments/my-nginx services/my-nginx-svc

对于资源数目较大的情况，您会发现使用 -l 或 --selector 指定的筛选器（标签查询）能很容易根据标签筛选资源：

kubectl delete deployment,services -l app=nginx

deployment.apps "my-nginx" deleted

service "my-nginx-svc" deleted

由于 kubectl 用来输出资源名称的语法与其所接受的资源名称语法相同，所以很容易使用 $() 或 xargs 进行链式操作：

kubectl get **$(**kubectl create -f docs/concepts/cluster-administration/nginx/ -o name | grep service**)**

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

my-nginx-svc LoadBalancer 10.0.0.208 <pending> 80/TCP 0s

上面的命令中，我们首先使用 examples/application/nginx/ 下的配置文件创建资源，并使用 -o name 的输出格式（以”资源/名称”的形式打印每个资源）打印所创建的资源。然后，我们通过 grep 来过滤 “service”，最后再打印 kubectl get 的内容。

如果您碰巧在某个路径下的多个子路径中组织资源，那么也可以递归地在所有子路径上执行操作，方法是在 --filename,-f 后面指定 --recursive 或者 -R。

例如，假设有一个目录路径为 project/k8s/development，它保存开发环境所需的所有清单，并按资源类型组织：

project/k8s/development

├── configmap

│   └── my-configmap.yaml

├── deployment

│   └── my-deployment.yaml

└── pvc

└── my-pvc.yaml

默认情况下，对 project/k8s/development 执行的批量操作将停止在目录的第一级，而不是处理所有子目录。 如果我们试图使用以下命令在此目录中创建资源，则会遇到一个错误：

kubectl apply -f project/k8s/development

error: you must provide one or more resources by argument or filename (.json|.yaml|.yml|stdin)

然而，在 --filename,-f 后面标明 --recursive 或者 -R 之后：

kubectl apply -f project/k8s/development --recursive

configmap/my-config created

deployment.apps/my-deployment created

persistentvolumeclaim/my-pvc created

--recursive 可以用于接受 --filename,-f 参数的任何操作，例如：kubectl {create,get,delete,describe,rollout} 等。

有多个 -f 参数出现的时候，--recursive 参数也能正常工作：

kubectl apply -f project/k8s/namespaces -f project/k8s/development --recursive

namespace/development created

namespace/staging created

configmap/my-config created

deployment.apps/my-deployment created

persistentvolumeclaim/my-pvc created

如果您有兴趣学习更多关于 kubectl 的内容，请阅读 [kubectl 概述](https://kubernetes.io/docs/reference/kubectl/overview/)。

有效地使用标签

到目前为止我们使用的示例中的资源最多使用了一个标签。在许多情况下，应使用多个标签来区分集合。

例如，不同的应用可能会为 app 标签设置不同的值。 但是，类似 [guestbook 示例](https://github.com/kubernetes/examples/tree/master/guestbook/) 这样的多层应用，还需要区分每一层。前端可以带以下标签：

labels:

app: guestbook

tier: frontend

Redis 的主节点和从节点会有不同的 tier 标签，甚至还有一个额外的 role 标签：

labels:

app: guestbook

tier: backend

role: master

以及

labels:

app: guestbook

tier: backend

role: slave

标签允许我们按照标签指定的任何维度对我们的资源进行切片和切块：

kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml

kubectl get pods -Lapp -Ltier -Lrole

NAME READY STATUS RESTARTS AGE APP TIER ROLE

guestbook-fe-4nlpb 1/1 Running 0 1m guestbook frontend <none>

guestbook-fe-ght6d 1/1 Running 0 1m guestbook frontend <none>

guestbook-fe-jpy62 1/1 Running 0 1m guestbook frontend <none>

guestbook-redis-master-5pg3b 1/1 Running 0 1m guestbook backend master

guestbook-redis-slave-2q2yf 1/1 Running 0 1m guestbook backend slave

guestbook-redis-slave-qgazl 1/1 Running 0 1m guestbook backend slave

my-nginx-divi2 1/1 Running 0 29m nginx <none> <none>

my-nginx-o0ef1 1/1 Running 0 29m nginx <none> <none>

kubectl get pods -lapp=guestbook,role=slave

NAME READY STATUS RESTARTS AGE

guestbook-redis-slave-2q2yf 1/1 Running 0 3m

guestbook-redis-slave-qgazl 1/1 Running 0 3m

金丝雀部署

另一个需要多标签的场景是用来区分同一组件的不同版本或者不同配置的多个部署。常见的做法是部署一个使用\*金丝雀发布\*来部署新应用版本（在 pod 模板中通过镜像标签指定），保持新旧版本应用同时运行，这样，新版本在完全发布之前也可以接收实时的生产流量。

例如，您可以使用 track 标签来区分不同的版本。

主要稳定的发行版将有一个 track 标签，其值为 stable：

name: frontend

replicas: 3

...

labels:

app: guestbook

tier: frontend

track: stable

...

image: gb-frontend:v3

然后，您可以创建 guestbook 前端的新版本，让这些版本的 track 标签带有不同的值（即 canary），以便两组 pod 不会重叠：

name: frontend-canary

replicas: 1

...

labels:

app: guestbook

tier: frontend

track: canary

...

image: gb-frontend:v4

前端服务通过选择标签的公共子集（即忽略 track 标签）来覆盖两组副本，以便流量可以转发到两个应用：

selector:

app: guestbook

tier: frontend

您可以调整 stable 和 canary 版本的副本数量，以确定每个版本将接收实时生产流量的比例(在本例中为 3:1)。一旦有信心，您就可以将新版本应用的 track 标签的值从 canary 替换为 stable，并且将老版本应用删除。

想要了解更具体的示例，请查看 [Ghost 部署教程](https://github.com/kelseyhightower/talks/tree/master/kubecon-eu-2016/demo#deploy-a-canary)。

更新标签

有时，现有的 pod 和其它资源需要在创建新资源之前重新标记。这可以用 kubectl label 完成。 例如，如果想要将所有 nginx pod 标记为前端层，只需运行：

kubectl label pods -l app=nginx tier=fe

pod/my-nginx-2035384211-j5fhi labeled

pod/my-nginx-2035384211-u2c7e labeled

pod/my-nginx-2035384211-u3t6x labeled

首先用标签 “app=nginx” 过滤所有的 pod，然后用 “tier=fe” 标记它们。想要查看您刚才标记的 pod，请运行：

kubectl get pods -l app=nginx -L tier

NAME READY STATUS RESTARTS AGE TIER

my-nginx-2035384211-j5fhi 1/1 Running 0 23m fe

my-nginx-2035384211-u2c7e 1/1 Running 0 23m fe

my-nginx-2035384211-u3t6x 1/1 Running 0 23m fe

这将输出所有 “app=nginx” 的 pod，并有一个额外的描述 pod 的 tier 的标签列（用参数 -L 或者 --label-columns 标明）。

想要了解更多信息，请参考 [标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) 和 [kubectl label](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#label)。

更新注解

有时，您可能希望将注解附加到资源中。注解是 API 客户端（如工具、库等）用于检索的任意非标识元数据。这可以通过 kubectl annotate 来完成。例如：

kubectl annotate pods my-nginx-v4-9gw19 description='my frontend running nginx'

kubectl get pods my-nginx-v4-9gw19 -o yaml

apiVersion: v1

kind: pod

metadata:

annotations:

description: my frontend running nginx

...

想要了解更多信息，请参考 [注解](https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/) 和 [kubectl annotate](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#annotate) 文档。

缩扩您的应用

当应用上的负载增长或收缩时，使用 kubectl 能够轻松实现规模的缩扩。例如，要将 nginx 副本的数量从 3 减少到 1，请执行以下操作：

kubectl scale deployment/my-nginx --replicas=1

deployment.extensions/my-nginx scaled

现在，您的 deployment 管理的 pod 只有一个了。

kubectl get pods -l app=nginx

NAME READY STATUS RESTARTS AGE

my-nginx-2035384211-j5fhi 1/1 Running 0 30m

想要让系统自动选择需要 nginx 副本的数量，范围从 1 到 3，请执行以下操作：

kubectl autoscale deployment/my-nginx --min=1 --max=3

horizontalpodautoscaler.autoscaling/my-nginx autoscaled

现在，您的 nginx 副本将根据需要自动地增加或者减少。

想要了解更多信息，请参考 [kubectl scale](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#scale), [kubectl autoscale](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#autoscale) 和 [pod 水平自动伸缩](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) 文档。

就地更新资源

有时，有必要对您所创建的资源进行小范围、无干扰地更新。

kubectl apply

建议在源代码管理中维护一组配置文件（参见[配置即代码](http://martinfowler.com/bliki/InfrastructureAsCode.html)），这样，它们就可以和应用代码一样进行维护和版本管理。然后，您可以用 [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#apply) 将配置变更应用到集群中。

这个命令将会把推送的版本与以前的版本进行比较，并应用您所做的更改，但是不会自动覆盖任何你没有指定更改的属性。

kubectl apply -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml

deployment.apps/my-nginx configured

注意，kubectl apply 将为资源增加一个额外的注解，以确定自上次调用以来对配置的更改。当调用它时，kubectl apply 会在以前的配置、提供的输入和资源的当前配置之间找出三方差异，以确定如何修改资源。

目前，新创建的资源是没有这个注解的，所以，第一次调用 kubectl apply 将使用提供的输入和资源的当前配置双方之间差异进行比较。在第一次调用期间，它无法检测资源创建时属性集的删除情况。因此，不会删除它们。

所有后续调用 kubectl apply 以及其它修改配置的命令，如 kubectl replace 和 kubectl edit，都将更新注解，并允许随后调用的 kubectl apply 使用三方差异进行检查和执行删除。

**注意：** 想要使用 apply，请始终使用 kubectl apply 或 kubectl create --save-config 创建资源。

kubectl edit

或者，您也可以使用 kubectl edit 更新资源：

kubectl edit deployment/my-nginx

这相当于首先 get 资源，在文本编辑器中编辑它，然后用更新的版本 apply 资源：

kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml

vi /tmp/nginx.yaml

*# do some edit, and then save the file*

kubectl apply -f /tmp/nginx.yaml

deployment.apps/my-nginx configured

rm /tmp/nginx.yaml

这使您可以更加容易地进行更重大的更改。请注意，可以使用 EDITOR 或 KUBE\_EDITOR 环境变量来指定编辑器。

想要了解更多信息，请参考 [kubectl edit](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#edit) 文档。

kubectl patch

您可以使用 kubectl patch 来更新 API 对象。此命令支持 JSON patch，JSON merge patch，以及 strategic merge patch。 请参考 [使用 kubectl patch 更新 API 对象](https://kubernetes.io/docs/tasks/run-application/update-api-object-kubectl-patch/) 和 [kubectl patch](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#patch).

破坏性的更新

在某些情况下，您可能需要更新某些初始化后无法更新的资源字段，或者您可能只想立即进行递归更改，例如修复 Deployment 创建的不正常的 Pod。若要更改这些字段，请使用 replace --force，它将删除并重新创建资源。在这种情况下，您可以简单地修改原始配置文件：

kubectl replace -f https://k8s.io/examples/application/nginx/nginx-deployment.yaml --force

deployment.apps/my-nginx deleted

deployment.apps/my-nginx replaced

在不中断服务的情况下更新应用

在某些时候，您最终需要更新已部署的应用，通常都是通过指定新的镜像或镜像标签，如上面的金丝雀发布的场景中所示。kubectl 支持几种更新操作，每种更新操作都适用于不同的场景。

我们将指导您通过 Deployment 如何创建和更新应用。

假设您正运行的是 1.7.9 版本的 nginx：

kubectl run my-nginx --image=nginx:1.7.9 --replicas=3

deployment.apps/my-nginx created

要更新到 1.9.1 版本，只需使用我们前面学到的 kubectl 命令将 .spec.template.spec.containers[0].image 从 nginx:1.7.9 修改为 nginx:1.9.1。

kubectl edit deployment/my-nginx

没错，就是这样！Deployment 将在后台逐步更新已经部署的 nginx 应用。它确保在更新过程中，只有一定数量的旧副本被开闭，并且只有一定基于所需 pod 数量的新副本被创建。想要了解更多细节，请参考 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)。

Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by [pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) and localhost communications.
2. Pod-to-Pod communications: this is the primary focus of this document.
3. Pod-to-Service communications: this is covered by [services](https://kubernetes.io/docs/concepts/services-networking/service/).
4. External-to-Service communications: this is covered by [services](https://kubernetes.io/docs/concepts/services-networking/service/).

* [**The Kubernetes network model**](https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model)
* [**How to implement the Kubernetes networking model**](https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model)
* [**What's next**](https://kubernetes.io/docs/concepts/cluster-administration/networking/#what-s-next)

Kubernetes is all about sharing machines between applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

The Kubernetes network model

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

* pods on a node can communicate with all pods on all nodes without NAT
* agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support Pods running in the host network (e.g. Linux):

* pods in the host network of a node can communicate with all pods on all nodes without NAT

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address. This means that containers within a Pod can all reach each other’s ports on localhost. This also means that containers within a Pod must coordinate port usage, but this is no different from processes in a VM. This is called the “IP-per-pod” model.

How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the Node itself which forward to your Pod (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The Pod itself is blind to the existence or non-existence of host ports.

How to implement the Kubernetes networking model

There are a number of ways that this network model can be implemented. This document is not an exhaustive study of the various methods, but hopefully serves as an introduction to various technologies and serves as a jumping-off point.

The following networking options are sorted alphabetically - the order does not imply any preferential status.

ACI

[Cisco Application Centric Infrastructure](https://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html) offers an integrated overlay and underlay SDN solution that supports containers, virtual machines, and bare metal servers. [ACI](https://www.github.com/noironetworks/aci-containers) provides container networking integration for ACI. An overview of the integration is provided [here](https://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/solution-overview-c22-739493.pdf).

Antrea

Project [Antrea](https://github.com/vmware-tanzu/antrea) is an opensource Kubernetes networking solution intended to be Kubernetes native. It leverages Open vSwitch as the networking data plane. Open vSwitch is a high-performance programmable virtual switch that supports both Linux and Windows. Open vSwitch enables Antrea to implement Kubernetes Network Policies in a high-performance and efficient manner. Thanks to the “programmable” characteristic of Open vSwitch, Antrea is able to implement an extensive set of networking and security features and services on top of Open vSwitch.

AOS from Apstra

[AOS](http://www.apstra.com/products/aos/) is an Intent-Based Networking system that creates and manages complex datacenter environments from a simple integrated platform. AOS leverages a highly scalable distributed design to eliminate network outages while minimizing costs.

The AOS Reference Design currently supports Layer-3 connected hosts that eliminate legacy Layer-2 switching problems. These Layer-3 hosts can be Linux servers (Debian, Ubuntu, CentOS) that create BGP neighbor relationships directly with the top of rack switches (TORs). AOS automates the routing adjacencies and then provides fine grained control over the route health injections (RHI) that are common in a Kubernetes deployment.

AOS has a rich set of REST API endpoints that enable Kubernetes to quickly change the network policy based on application requirements. Further enhancements will integrate the AOS Graph model used for the network design with the workload provisioning, enabling an end to end management system for both private and public clouds.

AOS supports the use of common vendor equipment from manufacturers including Cisco, Arista, Dell, Mellanox, HPE, and a large number of white-box systems and open network operating systems like Microsoft SONiC, Dell OPX, and Cumulus Linux.

Details on how the AOS system works can be accessed here: <http://www.apstra.com/products/how-it-works/>

AWS VPC CNI for Kubernetes

The [AWS VPC CNI](https://github.com/aws/amazon-vpc-cni-k8s) offers integrated AWS Virtual Private Cloud (VPC) networking for Kubernetes clusters. This CNI plugin offers high throughput and availability, low latency, and minimal network jitter. Additionally, users can apply existing AWS VPC networking and security best practices for building Kubernetes clusters. This includes the ability to use VPC flow logs, VPC routing policies, and security groups for network traffic isolation.

Using this CNI plugin allows Kubernetes pods to have the same IP address inside the pod as they do on the VPC network. The CNI allocates AWS Elastic Networking Interfaces (ENIs) to each Kubernetes node and using the secondary IP range from each ENI for pods on the node. The CNI includes controls for pre-allocation of ENIs and IP addresses for fast pod startup times and enables large clusters of up to 2,000 nodes.

Additionally, the CNI can be run alongside [Calico for network policy enforcement](https://docs.aws.amazon.com/eks/latest/userguide/calico.html). The AWS VPC CNI project is open source with [documentation on GitHub](https://github.com/aws/amazon-vpc-cni-k8s).

Big Cloud Fabric from Big Switch Networks

[Big Cloud Fabric](https://www.bigswitch.com/container-network-automation) is a cloud native networking architecture, designed to run Kubernetes in private cloud/on-premises environments. Using unified physical & virtual SDN, Big Cloud Fabric tackles inherent container networking problems such as load balancing, visibility, troubleshooting, security policies & container traffic monitoring.

With the help of the Big Cloud Fabric’s virtual pod multi-tenant architecture, container orchestration systems such as Kubernetes, RedHat OpenShift, Mesosphere DC/OS & Docker Swarm will be natively integrated alongside with VM orchestration systems such as VMware, OpenStack & Nutanix. Customers will be able to securely inter-connect any number of these clusters and enable inter-tenant communication between them if needed.

BCF was recognized by Gartner as a visionary in the latest [Magic Quadrant](http://go.bigswitch.com/17GatedDocuments-MagicQuadrantforDataCenterNetworking_Reg.html). One of the BCF Kubernetes on-premises deployments (which includes Kubernetes, DC/OS & VMware running on multiple DCs across different geographic regions) is also referenced [here](https://portworx.com/architects-corner-kubernetes-satya-komala-nio/).

Cilium

[Cilium](https://github.com/cilium/cilium) is open source software for providing and transparently securing network connectivity between application containers. Cilium is L7/HTTP aware and can enforce network policies on L3-L7 using an identity based security model that is decoupled from network addressing.

CNI-Genie from Huawei

[CNI-Genie](https://github.com/Huawei-PaaS/CNI-Genie) is a CNI plugin that enables Kubernetes to [simultaneously have access to different implementations](https://github.com/Huawei-PaaS/CNI-Genie/blob/master/docs/multiple-cni-plugins/README.md#what-cni-genie-feature-1-multiple-cni-plugins-enables) of the [Kubernetes network model](https://git.k8s.io/website/docs/concepts/cluster-administration/networking.md#kubernetes-model) in runtime. This includes any implementation that runs as a [CNI plugin](https://github.com/containernetworking/cni#3rd-party-plugins), such as [Flannel](https://github.com/coreos/flannel#flannel), [Calico](http://docs.projectcalico.org/), [Romana](http://romana.io/), [Weave-net](https://www.weave.works/products/weave-net/).

CNI-Genie also supports [assigning multiple IP addresses to a pod](https://github.com/Huawei-PaaS/CNI-Genie/blob/master/docs/multiple-ips/README.md#feature-2-extension-cni-genie-multiple-ip-addresses-per-pod), each from a different CNI plugin.

cni-ipvlan-vpc-k8s

[cni-ipvlan-vpc-k8s](https://github.com/lyft/cni-ipvlan-vpc-k8s) contains a set of CNI and IPAM plugins to provide a simple, host-local, low latency, high throughput, and compliant networking stack for Kubernetes within Amazon Virtual Private Cloud (VPC) environments by making use of Amazon Elastic Network Interfaces (ENI) and binding AWS-managed IPs into Pods using the Linux kernel’s IPvlan driver in L2 mode.

The plugins are designed to be straightforward to configure and deploy within a VPC. Kubelets boot and then self-configure and scale their IP usage as needed without requiring the often recommended complexities of administering overlay networks, BGP, disabling source/destination checks, or adjusting VPC route tables to provide per-instance subnets to each host (which is limited to 50-100 entries per VPC). In short, cni-ipvlan-vpc-k8s significantly reduces the network complexity required to deploy Kubernetes at scale within AWS.

Contiv

[Contiv](https://github.com/contiv/netplugin) provides configurable networking (native l3 using BGP, overlay using vxlan, classic l2, or Cisco-SDN/ACI) for various use cases. [Contiv](http://contiv.io/) is all open sourced.

Contrail / Tungsten Fabric

[Contrail](http://www.juniper.net/us/en/products-services/sdn/contrail/contrail-networking/), based on [Tungsten Fabric](https://tungsten.io/), is a truly open, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with various orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide different isolation modes for virtual machines, containers/pods and bare metal workloads.

DANM

[DANM](https://github.com/nokia/danm) is a networking solution for telco workloads running in a Kubernetes cluster. It’s built up from the following components:

* A CNI plugin capable of provisioning IPVLAN interfaces with advanced features
* An in-built IPAM module with the capability of managing multiple, cluster-wide, discontinuous L3 networks and provide a dynamic, static, or no IP allocation scheme on-demand
* A CNI metaplugin capable of attaching multiple network interfaces to a container, either through its own CNI, or through delegating the job to any of the popular CNI solution like SRI-OV, or Flannel in parallel
* A Kubernetes controller capable of centrally managing both VxLAN and VLAN interfaces of all Kubernetes hosts
* Another Kubernetes controller extending Kubernetes’ Service-based service discovery concept to work over all network interfaces of a Pod

With this toolset DANM is able to provide multiple separated network interfaces, the possibility to use different networking back ends and advanced IPAM features for the pods.

Flannel

[Flannel](https://github.com/coreos/flannel#flannel) is a very simple overlay network that satisfies the Kubernetes requirements. Many people have reported success with Flannel and Kubernetes.

Google Compute Engine (GCE)

For the Google Compute Engine cluster configuration scripts, [advanced routing](https://cloud.google.com/vpc/docs/routes) is used to assign each VM a subnet (default is /24 - 254 IPs). Any traffic bound for that subnet will be routed directly to the VM by the GCE network fabric. This is in addition to the “main” IP address assigned to the VM, which is NAT’ed for outbound internet access. A linux bridge (called cbr0) is configured to exist on that subnet, and is passed to docker’s --bridge flag.

Docker is started with:

DOCKER\_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"

This bridge is created by Kubelet (controlled by the --network-plugin=kubenet flag) according to the Node’s .spec.podCIDR.

Docker will now allocate IPs from the cbr-cidr block. Containers can reach each other and Nodes over the cbr0 bridge. Those IPs are all routable within the GCE project network.

GCE itself does not know anything about these IPs, though, so it will not NAT them for outbound internet traffic. To achieve that an iptables rule is used to masquerade (aka SNAT - to make it seem as if packets came from the Node itself) traffic that is bound for IPs outside the GCE project network (10.0.0.0/8).

iptables -t nat -A POSTROUTING ! -d 10.0.0.0/8 -o eth0 -j MASQUERADE

Lastly IP forwarding is enabled in the kernel (so the kernel will process packets for bridged containers):

sysctl net.ipv4.ip\_forward=1

The result of all this is that all Pods can reach each other and can egress traffic to the internet.

Jaguar

[Jaguar](https://gitlab.com/sdnlab/jaguar) is an open source solution for Kubernetes’s network based on OpenDaylight. Jaguar provides overlay network using vxlan and Jaguar CNIPlugin provides one IP address per pod.

k-vswitch

[k-vswitch](https://github.com/k-vswitch/k-vswitch) is a simple Kubernetes networking plugin based on [Open vSwitch](https://www.openvswitch.org/). It leverages existing functionality in Open vSwitch to provide a robust networking plugin that is easy-to-operate, performant and secure.

Knitter

[Knitter](https://github.com/ZTE/Knitter/) is a network solution which supports multiple networking in Kubernetes. It provides the ability of tenant management and network management. Knitter includes a set of end-to-end NFV container networking solutions besides multiple network planes, such as keeping IP address for applications, IP address migration, etc.

Kube-OVN

[Kube-OVN](https://github.com/alauda/kube-ovn) is an OVN-based kubernetes network fabric for enterprises. With the help of OVN/OVS, it provides some advanced overlay network features like subnet, QoS, static IP allocation, traffic mirroring, gateway, openflow-based network policy and service proxy.

Kube-router

[Kube-router](https://github.com/cloudnativelabs/kube-router) is a purpose-built networking solution for Kubernetes that aims to provide high performance and operational simplicity. Kube-router provides a Linux [LVS/IPVS](http://www.linuxvirtualserver.org/software/ipvs.html)-based service proxy, a Linux kernel forwarding-based pod-to-pod networking solution with no overlays, and iptables/ipset-based network policy enforcer.

L2 networks and linux bridging

If you have a “dumb” L2 network, such as a simple switch in a “bare-metal” environment, you should be able to do something similar to the above GCE setup. Note that these instructions have only been tried very casually - it seems to work, but has not been thoroughly tested. If you use this technique and perfect the process, please let us know.

Follow the “With Linux Bridge devices” section of [this very nice tutorial](http://blog.oddbit.com/2014/08/11/four-ways-to-connect-a-docker/) from Lars Kellogg-Stedman.

Multus (a Multi Network plugin)

[Multus](https://github.com/Intel-Corp/multus-cni) is a Multi CNI plugin to support the Multi Networking feature in Kubernetes using CRD based network objects in Kubernetes.

Multus supports all [reference plugins](https://github.com/containernetworking/plugins) (eg. [Flannel](https://github.com/containernetworking/plugins/tree/master/plugins/meta/flannel), [DHCP](https://github.com/containernetworking/plugins/tree/master/plugins/ipam/dhcp), [Macvlan](https://github.com/containernetworking/plugins/tree/master/plugins/main/macvlan)) that implement the CNI specification and 3rd party plugins (eg. [Calico](https://github.com/projectcalico/cni-plugin), [Weave](https://github.com/weaveworks/weave), [Cilium](https://github.com/cilium/cilium), [Contiv](https://github.com/contiv/netplugin)). In addition to it, Multus supports [SRIOV](https://github.com/hustcat/sriov-cni), [DPDK](https://github.com/Intel-Corp/sriov-cni), [OVS-DPDK & VPP](https://github.com/intel/vhost-user-net-plugin) workloads in Kubernetes with both cloud native and NFV based applications in Kubernetes.

NSX-T

[VMware NSX-T](https://docs.vmware.com/en/VMware-NSX-T/index.html) is a network virtualization and security platform. NSX-T can provide network virtualization for a multi-cloud and multi-hypervisor environment and is focused on emerging application frameworks and architectures that have heterogeneous endpoints and technology stacks. In addition to vSphere hypervisors, these environments include other hypervisors such as KVM, containers, and bare metal.

[NSX-T Container Plug-in (NCP)](https://docs.vmware.com/en/VMware-NSX-T/2.0/nsxt_20_ncp_kubernetes.pdf) provides integration between NSX-T and container orchestrators such as Kubernetes, as well as integration between NSX-T and container-based CaaS/PaaS platforms such as Pivotal Container Service (PKS) and OpenShift.

Nuage Networks VCS (Virtualized Cloud Services)

[Nuage](http://www.nuagenetworks.net/) provides a highly scalable policy-based Software-Defined Networking (SDN) platform. Nuage uses the open source Open vSwitch for the data plane along with a feature rich SDN Controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes Pods and non-Kubernetes environments (VMs and bare metal servers). Nuage’s policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications.The platform’s real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

OpenVSwitch

[OpenVSwitch](https://www.openvswitch.org/) is a somewhat more mature but also complicated way to build an overlay network. This is endorsed by several of the “Big Shops” for networking.

OVN (Open Virtual Networking)

OVN is an opensource network virtualization solution developed by the Open vSwitch community. It lets one create logical switches, logical routers, stateful ACLs, load-balancers etc to build different virtual networking topologies. The project has a specific Kubernetes plugin and documentation at [ovn-kubernetes](https://github.com/openvswitch/ovn-kubernetes).

Project Calico

[Project Calico](http://docs.projectcalico.org/) is an open source container networking provider and network policy engine.

Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet, for both Linux (open source) and Windows (proprietary - available from [Tigera](https://www.tigera.io/essentials/)). Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking. Calico also provides fine-grained, intent based network security policy for Kubernetes pods via its distributed firewall.

Calico can also be run in policy enforcement mode in conjunction with other networking solutions such as Flannel, aka [canal](https://github.com/tigera/canal), or native GCE, AWS or Azure networking.

Romana

[Romana](http://romana.io/) is an open source network and security automation solution that lets you deploy Kubernetes without an overlay network. Romana supports Kubernetes [Network Policy](https://kubernetes.io/docs/concepts/services-networking/network-policies/) to provide isolation across network namespaces.

Weave Net from Weaveworks

[Weave Net](https://www.weave.works/products/weave-net/) is a resilient and simple to use network for Kubernetes and its hosted applications. Weave Net runs as a [CNI plug-in](https://www.weave.works/docs/net/latest/cni-plugin/) or stand-alone. In either version, it doesn’t require any configuration or extra code to run, and in both cases, the network provides one IP address per pod - as is standard for Kubernetes.

What's next

日志架构

应用和系统日志可以让您了解集群内部的运行状况。日志对调试问题和监控集群活动非常有用。大部分现代化应用都有某种日志记录机制；同样地，大多数容器引擎也被设计成支持某种日志记录机制。针对容器化应用，最简单且受欢迎的日志记录方式就是写入标准输出和标准错误流。

但是，由容器引擎或 runtime 提供的原生功能通常不足以满足完整的日志记录方案。例如，如果发生容器崩溃、pod 被逐出或节点宕机等情况，您仍然想访问到应用日志。因此，日志应该具有独立的存储和生命周期，与节点、pod 或容器的生命周期相独立。这个概念叫 *集群级的日志* 。集群级日志方案需要一个独立的后台来存储、分析和查询日志。Kubernetes 没有为日志数据提供原生存储方案，但是您可以集成许多现有的日志解决方案到 Kubernetes 集群中。

* [**Kubernetes 中的基本日志记录**](https://kubernetes.io/zh/docs/concepts/cluster-administration/logging/#kubernetes-%e4%b8%ad%e7%9a%84%e5%9f%ba%e6%9c%ac%e6%97%a5%e5%bf%97%e8%ae%b0%e5%bd%95)
* [**节点级日志记录**](https://kubernetes.io/zh/docs/concepts/cluster-administration/logging/#%e8%8a%82%e7%82%b9%e7%ba%a7%e6%97%a5%e5%bf%97%e8%ae%b0%e5%bd%95)
* [**集群级日志架构**](https://kubernetes.io/zh/docs/concepts/cluster-administration/logging/#%e9%9b%86%e7%be%a4%e7%ba%a7%e6%97%a5%e5%bf%97%e6%9e%b6%e6%9e%84)

集群级日志架构假定在集群内部或者外部有一个日志后台。如果您对集群级日志不感兴趣，您仍会发现关于如何在节点上存储和处理日志的描述对您是有用的。

Kubernetes 中的基本日志记录

本节，您会看到一个kubernetes 中生成基本日志的例子，该例子中数据被写入到标准输出。 这里通过一个特定的 [pod 规约](https://kubernetes.io/examples/debug/counter-pod.yaml) 演示创建一个容器，并令该容器每秒钟向标准输出写入数据。

| [debug/counter-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/debug/counter-pod.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: counter  spec:  containers:  - name: count  image: busybox  args: [/bin/sh, -c,  'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done'] |

用下面的命令运行 pod：

kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml

输出结果为：

pod/counter created

使用 kubectl logs 命令获取日志:

kubectl logs counter

输出结果为：

0: Mon Jan 1 00:00:00 UTC 2001

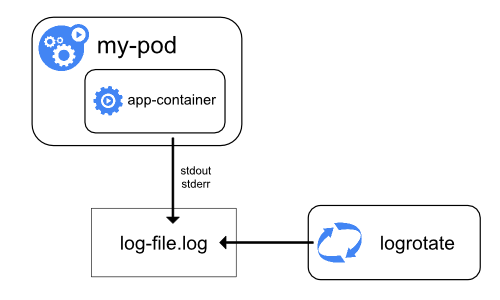
1: Mon Jan 1 00:00:01 UTC 2001

2: Mon Jan 1 00:00:02 UTC 2001

...

一旦发生容器崩溃，您可以使用命令 kubectl logs 和参数 --previous 检索之前的容器日志。 如果 pod 中有多个容器，您应该向该命令附加一个容器名以访问对应容器的日志。 详见 [kubectl logs 文档](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#logs)。

节点级日志记录



容器化应用写入 stdout 和 stderr 的任何数据，都会被容器引擎捕获并被重定向到某个位置。 例如，Docker 容器引擎将这两个输出流重定向到某个 [日志驱动](https://docs.docker.com/engine/admin/logging/overview) ， 该日志驱动在 Kubernetes 中配置为以 json 格式写入文件。

**注意：** Docker json 日志驱动将日志的每一行当作一条独立的消息。该日志驱动不直接支持多行消息。您需要在日志代理级别或更高级别处理多行消息。

默认情况下，如果容器重启，kubelet 会保留被终止的容器日志。 如果 pod 在工作节点被驱逐，该 pod 中所有的容器也会被驱逐，包括容器日志。

节点级日志记录中，需要重点考虑实现日志的轮转，以此来保证日志不会消耗节点上所有的可用空间。 Kubernetes 当前并不负责轮转日志，而是通过部署工具建立一个解决问题的方案。 例如，在 Kubernetes 集群中，用 kube-up.sh 部署一个每小时运行的工具 [logrotate](https://linux.die.net/man/8/logrotate)。 您也可以设置容器 runtime 来自动地轮转应用日志，比如使用 Docker 的 log-opt 选项。 在 kube-up.sh 脚本中，使用后一种方式来处理 GCP 上的 COS 镜像，而使用前一种方式来处理其他环境。 这两种方式，默认日志超过 10MB 大小时都会触发日志轮转。

例如，您可以找到关于 kube-up.sh 为 GCP 环境的 COS 镜像设置日志的详细信息， 相应的脚本在 [这里](https://github.com/kubernetes/kubernetes/blob/master/cluster/gce/gci/configure-helper.sh)。

当运行 [kubectl logs](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#logs) 时， 节点上的 kubelet 处理该请求并直接读取日志文件，同时在响应中返回日志文件内容。

**注意：**

当前，如果有其他系统机制执行日志轮转，那么 kubectl logs 仅可查询到最新的日志内容。 比如，一个 10MB 大小的文件，通过logrotate 执行轮转后生成两个文件，一个 10MB 大小，一个为空，所以 kubectl logs 将返回空。

系统组件日志

系统组件有两种类型：在容器中运行的和不在容器中运行的。例如：

* 在容器中运行的 kube-scheduler 和 kube-proxy。
* 不在容器中运行的 kubelet 和容器运行时（例如 Docker。

在使用 systemd 机制的服务器上，kubelet 和容器 runtime 写入日志到 journald。 如果没有 systemd，他们写入日志到 /var/log 目录的 .log 文件。 容器中的系统组件通常将日志写到 /var/log 目录，绕过了默认的日志机制。他们使用 [klog](https://github.com/kubernetes/klog) 日志库。 您可以在[日志开发文档](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-instrumentation/logging.md)找到这些组件的日志告警级别协议。

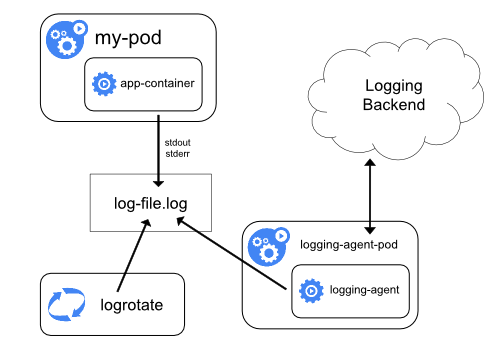
和容器日志类似，/var/log 目录中的系统组件日志也应该被轮转。 通过脚本 kube-up.sh 启动的 Kubernetes 集群中，日志被工具 logrotate 执行每日轮转，或者日志大小超过 100MB 时触发轮转。

集群级日志架构

虽然Kubernetes没有为集群级日志记录提供原生的解决方案，但您可以考虑几种常见的方法。以下是一些选项：

* 使用在每个节点上运行的节点级日志记录代理。
* 在应用程序的 pod 中，包含专门记录日志的 sidecar 容器。
* 将日志直接从应用程序中推送到日志记录后端。

使用节点级日志代理



您可以通过在每个节点上使用 *节点级的日志记录代理* 来实现群集级日志记录。日志记录代理是一种用于暴露日志或将日志推送到后端的专用工具。通常，日志记录代理程序是一个容器，它可以访问包含该节点上所有应用程序容器的日志文件的目录。

由于日志记录代理必须在每个节点上运行，它可以用 DaemonSet 副本，Pod 或 本机进程来实现。然而，后两种方法被弃用并且非常不别推荐。

对于 Kubernetes 集群来说，使用节点级的日志代理是最常用和被推荐的方式，因为在每个节点上仅创建一个代理，并且不需要对节点上的应用做修改。 但是，节点级的日志 \_仅适用于应用程序的标准输出和标准错误输出\_。

Kubernetes 并不指定日志代理，但是有两个可选的日志代理与 Kubernetes 发行版一起发布。 [Stackdriver 日志](https://kubernetes.io/docs/user-guide/logging/stackdriver) 适用于 Google Cloud Platform，和 [Elasticsearch](https://kubernetes.io/docs/user-guide/logging/elasticsearch)。 您可以在专门的文档中找到更多的信息和说明。两者都使用 [fluentd](http://www.fluentd.org/) 与自定义配置作为节点上的代理。

使用 sidecar 容器和日志代理

您可以通过以下方式之一使用 sidecar 容器：

* sidecar 容器将应用程序日志传送到自己的标准输出。
* sidecar 容器运行一个日志代理，配置该日志代理以便从应用容器收集日志。

传输数据流的 sidecar 容器

利用 sidecar 容器向自己的 stdout 和 stderr 传输流的方式，您就可以利用每个节点上的 kubelet 和日志代理来处理日志。 sidecar 容器从文件，socket 或 journald 读取日志。每个 sidecar 容器打印其自己的 stdout 和 stderr 流。

这种方法允许您将日志流从应用程序的不同部分分离开，其中一些可能缺乏对写入 stdout 或 stderr 的支持。重定向日志背后的逻辑是最小的，因此它的开销几乎可以忽略不计。 另外，因为 stdout、stderr 由 kubelet 处理，你可以使用内置的工具 kubectl logs。

考虑接下来的例子。pod 的容器向两个文件写不同格式的日志，下面是这个 pod 的配置文件:

| [admin/logging/two-files-counter-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/admin/logging/two-files-counter-pod.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: counter  spec:  containers:  - name: count  image: busybox  args:  - /bin/sh  - -c  - *>*  *i=0;*  while **true**;  do  echo "$i: $(date)" >> /var/log/1.log;  echo "$(date) INFO $i" >> /var/log/2.log;  i=$((i+1));  sleep 1;  done  volumeMounts:  - name: varlog  mountPath: /var/log  volumes:  - name: varlog  emptyDir: {} |

在同一个日志流中有两种不同格式的日志条目，这有点混乱，即使您试图重定向它们到容器的 stdout 流。 取而代之的是，您可以引入两个 sidecar 容器。 每一个 sidecar 容器可以从共享卷跟踪特定的日志文件，并重定向文件内容到各自的 stdout 流。

这是运行两个 sidecar 容器的 pod 文件。

| [admin/logging/two-files-counter-pod-streaming-sidecar.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/admin/logging/two-files-counter-pod-streaming-sidecar.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: counter  spec:  containers:  - name: count  image: busybox  args:  - /bin/sh  - -c  - *>*  *i=0;*  while **true**;  do  echo "$i: $(date)" >> /var/log/1.log;  echo "$(date) INFO $i" >> /var/log/2.log;  i=$((i+1));  sleep 1;  done  volumeMounts:  - name: varlog  mountPath: /var/log  - name: count-log-1  image: busybox  args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']  volumeMounts:  - name: varlog  mountPath: /var/log  - name: count-log-2  image: busybox  args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']  volumeMounts:  - name: varlog  mountPath: /var/log  volumes:  - name: varlog  emptyDir: {} |

现在当您运行这个 pod 时，您可以分别地访问每一个日志流，运行如下命令：

kubectl logs counter count-log-1

0: Mon Jan 1 00:00:00 UTC 2001

1: Mon Jan 1 00:00:01 UTC 2001

2: Mon Jan 1 00:00:02 UTC 2001

...

kubectl logs counter count-log-2

Mon Jan 1 00:00:00 UTC 2001 INFO 0

Mon Jan 1 00:00:01 UTC 2001 INFO 1

Mon Jan 1 00:00:02 UTC 2001 INFO 2

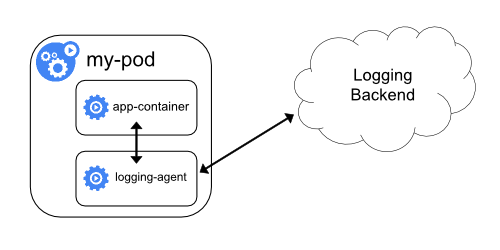
...

集群中安装的节点级代理会自动获取这些日志流，而无需进一步配置。如果您愿意，您可以配置代理程序来解析源容器的日志行。

注意，尽管 CPU 和内存使用率都很低（以多个 cpu millicores 指标排序或者按内存的兆字节排序）， 向文件写日志然后输出到 stdout 流仍然会成倍地增加磁盘使用率。 如果您的应用向单一文件写日志，通常最好设置 /dev/stdout 作为目标路径，而不是使用流式的 sidecar 容器方式。

应用本身如果不具备轮转日志文件的功能，可以通过 sidecar 容器实现。 该方式的一个例子是运行一个定期轮转日志的容器。 然而，还是推荐直接使用 stdout 和 stderr，将日志的轮转和保留策略交给 kubelet。

具有日志代理功能的 sidecar 容器



如果节点级日志记录代理程序对于你的场景来说不够灵活，您可以创建一个带有单独日志记录代理程序的 sidecar 容器，将代理程序专门配置为与您的应用程序一起运行。

**注意：** 在 sidecar 容器中使用日志代理会导致严重的资源损耗。此外，您不能使用 kubectl logs 命令访问日志，因为日志并没有被 kubelet 管理。

例如，您可以使用 [Stackdriver](https://kubernetes.io/docs/tasks/debug-application-cluster/logging-stackdriver/)，它使用fluentd作为日志记录代理。 以下是两个可用于实现此方法的配置文件。 第一个文件包含配置 fluentd 的[ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/)。

| [admin/logging/fluentd-sidecar-config.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/admin/logging/fluentd-sidecar-config.yaml) |
| --- |
| apiVersion: v1  kind: ConfigMap  metadata:  name: fluentd-config  data:  fluentd.conf: *|*  *<source>*  type tail  format none  path /var/log/1.log  pos\_file /var/log/1.log.pos  tag count.format1  </source*>*  *<source>*  type tail  format none  path /var/log/2.log  pos\_file /var/log/2.log.pos  tag count.format2  </source*>*  *<match \*\*>*  type google\_cloud  </match> |

**注意：** 配置fluentd超出了本文的范围。要知道更多的关于如何配置fluentd，请参考[fluentd 官方文档](http://docs.fluentd.org/).

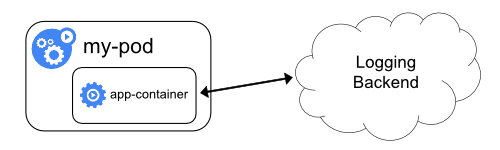
第二个文件描述了运行 fluentd sidecar 容器的 pod 。flutend 通过 pod 的挂载卷获取它的配置数据。

| [admin/logging/two-files-counter-pod-agent-sidecar.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/admin/logging/two-files-counter-pod-agent-sidecar.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: counter  spec:  containers:  - name: count  image: busybox  args:  - /bin/sh  - -c  - *>*  *i=0;*  while **true**;  do  echo "$i: $(date)" >> /var/log/1.log;  echo "$(date) INFO $i" >> /var/log/2.log;  i=$((i+1));  sleep 1;  done  volumeMounts:  - name: varlog  mountPath: /var/log  - name: count-agent  image: k8s.gcr.io/fluentd-gcp:1.30  env:  - name: FLUENTD\_ARGS  value: -c /etc/fluentd-config/fluentd.conf  volumeMounts:  - name: varlog  mountPath: /var/log  - name: config-volume  mountPath: /etc/fluentd-config  volumes:  - name: varlog  emptyDir: {}  - name: config-volume  configMap:  name: fluentd-config |

一段时间后，您可以在 Stackdriver 界面看到日志消息。

记住，这只是一个例子，事实上您可以用任何一个日志代理替换 fluentd ，并从应用容器中读取任何资源。

从应用中直接暴露日志目录



通过暴露或推送每个应用的日志，您可以实现集群级日志记录；然而，这种日志记录机制的实现已超出 Kubernetes 的范围。

配置 kubelet 垃圾回收策略

垃圾回收是 kubelet 的一个有用功能，它将清理未使用的镜像和容器。

Kubelet 将每分钟对容器执行一次垃圾回收，每五分钟对镜像执行一次垃圾回收。

不建议使用外部垃圾收集工具，因为这些工具可能会删除原本期望存在的容器进而破坏 kubelet 的行为。

* [**镜像回收**](https://kubernetes.io/zh/docs/concepts/cluster-administration/kubelet-garbage-collection/#%e9%95%9c%e5%83%8f%e5%9b%9e%e6%94%b6)
* [**容器回收**](https://kubernetes.io/zh/docs/concepts/cluster-administration/kubelet-garbage-collection/#%e5%ae%b9%e5%99%a8%e5%9b%9e%e6%94%b6)
* [**用户配置**](https://kubernetes.io/zh/docs/concepts/cluster-administration/kubelet-garbage-collection/#%e7%94%a8%e6%88%b7%e9%85%8d%e7%bd%ae)
* [**弃用**](https://kubernetes.io/zh/docs/concepts/cluster-administration/kubelet-garbage-collection/#%e5%bc%83%e7%94%a8)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/cluster-administration/kubelet-garbage-collection/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

镜像回收

Kubernetes 借助于 cadvisor 通过 imageManager 来管理所有镜像的生命周期。

镜像垃圾回收策略只考虑两个因素：HighThresholdPercent 和 LowThresholdPercent。

磁盘使用率超过上限阈值（HighThresholdPercent）将触发垃圾回收。

垃圾回收将删除最近最少使用的镜像，直到磁盘使用率满足下限阈值（LowThresholdPercent）。

容器回收

容器垃圾回收策略考虑三个用户定义变量。MinAge 是容器可以被执行垃圾回收的最小生命周期。MaxPerPodContainer 是每个 pod 内允许存在的死亡容器的最大数量。 MaxContainers 是全部死亡容器的最大数量。可以分别独立地通过将 MinAge 设置为 0，以及将 MaxPerPodContainer 和 MaxContainers 设置为小于 0 来禁用这些变量。

Kubelet 将处理无法辨识的、已删除的以及超出前面提到的参数所设置范围的容器。最老的容器通常会先被移除。 MaxPerPodContainer 和 MaxContainer 在某些场景下可能会存在冲突，例如在保证每个 pod 内死亡容器的最大数量（MaxPerPodContainer）的条件下可能会超过允许存在的全部死亡容器的最大数量（MaxContainer）。 MaxPerPodContainer 在这种情况下会被进行调整：最坏的情况是将 MaxPerPodContainer 降级为 1，并驱逐最老的容器。 此外，pod 内已经被删除的容器一旦年龄超过 MinAge 就会被清理。

不被 kubelet 管理的容器不受容器垃圾回收的约束。

用户配置

用户可以使用以下 kubelet 参数调整相关阈值来优化镜像垃圾回收：

1. image-gc-high-threshold，触发镜像垃圾回收的磁盘使用率百分比。默认值为 85%。
2. image-gc-low-threshold，镜像垃圾回收试图释放资源后达到的磁盘使用率百分比。默认值为 80%。

我们还允许用户通过以下 kubelet 参数自定义垃圾收集策略：

1. minimum-container-ttl-duration，完成的容器在被垃圾回收之前的最小年龄，默认是 0 分钟，这意味着每个完成的容器都会被执行垃圾回收。
2. maximum-dead-containers-per-container，每个容器要保留的旧实例的最大数量。默认值为 1。
3. maximum-dead-containers，要全局保留的旧容器实例的最大数量。默认值是 -1，这意味着没有全局限制。

容器可能会在其效用过期之前被垃圾回收。这些容器可能包含日志和其他对故障诊断有用的数据。 强烈建议为 maximum-dead-containers-per-container 设置一个足够大的值，以便每个预期容器至少保留一个死亡容器。 由于同样的原因，maximum-dead-containers 也建议使用一个足够大的值。

查阅 [这个问题](https://github.com/kubernetes/kubernetes/issues/13287) 获取更多细节。

弃用

这篇文档中的一些 kubelet 垃圾收集（Garbage Collection）功能将在未来被 kubelet 驱逐回收（eviction）所替代。

包括:

| 现存参数 | 新参数 | 解释 |
| --- | --- | --- |
| --image-gc-high-threshold | --eviction-hard 或 --eviction-soft | 现存的驱逐回收信号可以触发镜像垃圾回收 |
| --image-gc-low-threshold | --eviction-minimum-reclaim | 驱逐回收实现相同行为 |
| --maximum-dead-containers |  | 一旦旧日志存储在容器上下文之外，就会被弃用 |
| --maximum-dead-containers-per-container |  | 一旦旧日志存储在容器上下文之外，就会被弃用 |
| --minimum-container-ttl-duration |  | 一旦旧日志存储在容器上下文之外，就会被弃用 |
| --low-diskspace-threshold-mb | --eviction-hard or eviction-soft | 驱逐回收将磁盘阈值泛化到其他资源 |
| --outofdisk-transition-frequency | --eviction-pressure-transition-period | 驱逐回收将磁盘压力转换到其他资源 |

控制器管理器指标

控制器管理器指标为控制器管理器的性能和健康提供了重要的观测手段。

* [**什么是控制器管理器度量**](https://kubernetes.io/zh/docs/concepts/cluster-administration/controller-metrics/#%e4%bb%80%e4%b9%88%e6%98%af%e6%8e%a7%e5%88%b6%e5%99%a8%e7%ae%a1%e7%90%86%e5%99%a8%e5%ba%a6%e9%87%8f)
* [**配置**](https://kubernetes.io/zh/docs/concepts/cluster-administration/controller-metrics/#%e9%85%8d%e7%bd%ae)

什么是控制器管理器度量

控制器管理器指标为控制器管理器的性能和健康提供了重要的观测手段。 这些度量包括常见的 Go 语言运行时度量，比如 go\_routine 计数，以及控制器特定的度量，比如 etcd 请求延迟或 云提供商（AWS、GCE、OpenStack）的 API 延迟，这些参数可以用来测量集群的健康状况。

从 Kubernetes 1.7 版本开始，详细的云提供商指标可用于 GCE、AWS、Vsphere 和 OpenStack 的存储操作。 这些度量可用于监视持久卷操作的健康状况。

例如，在 GCE 中这些指标叫做：

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "instance\_list"}

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "disk\_insert"}

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "disk\_delete"}

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "attach\_disk"}

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "detach\_disk"}

cloudprovider\_gce\_api\_request\_duration\_seconds { request = "list\_disk"}

配置

在集群中，控制器管理器指标可从它所在的主机上的 http://localhost:10252/metrics 中获得。

这些指标是以 [prometheus 格式](https://prometheus.io/docs/instrumenting/exposition_formats/) 发出的，是人类可读的。

在生产环境中，您可能想配置 prometheus 或其他一些指标收集工具，以定期收集这些指标数据，并将它们应用到某种时间序列数据库中。

反馈

Kubernetes 中的代理

本文讲述了 Kubernetes 中所使用的代理。

* [**代理**](https://kubernetes.io/zh/docs/concepts/cluster-administration/proxies/#%e4%bb%a3%e7%90%86)
* [**请求重定向**](https://kubernetes.io/zh/docs/concepts/cluster-administration/proxies/#%e8%af%b7%e6%b1%82%e9%87%8d%e5%ae%9a%e5%90%91)

代理

用户在使用 Kubernetes 的过程中可能遇到几种不同的代理（proxy）：

1. [kubectl proxy](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#directly-accessing-the-rest-api)：
   * 运行在用户的桌面或 pod 中
   * 从本机地址到 Kubernetes apiserver 的代理
   * 客户端到代理使用 HTTP 协议
   * 代理到 apiserver 使用 HTTPS 协议
   * 指向 apiserver
   * 添加认证头信息
2. [apiserver proxy](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#discovering-builtin-services)：
   * 是一个建立在 apiserver 内部的“堡垒”
   * 将集群外部的用户与群集 IP 相连接，这些IP是无法通过其他方式访问的
   * 运行在 apiserver 进程内
   * 客户端到代理使用 HTTPS 协议 (如果配置 apiserver 使用 HTTP 协议，则使用 HTTP 协议)
   * 通过可用信息进行选择，代理到目的地可能使用 HTTP 或 HTTPS 协议
   * 可以用来访问 Node、 Pod 或 Service
   * 当用来访问 Service 时，会进行负载均衡
3. [kube proxy](https://kubernetes.io/docs/concepts/services-networking/service/#ips-and-vips)：
   * 在每个节点上运行
   * 代理 UDP 和 TCP
   * 不支持 HTTP
   * 提供负载均衡能力
   * 只用来访问 Service
4. apiserver 之前的代理/负载均衡器：
   * 在不同集群间的存在形式和实现不同 (如 nginx)
   * 位于所有客户端和一个或多个 apiserver 之间
   * 存在多个 apiserver 时，扮演负载均衡器的角色
5. 外部服务的云负载均衡器：
   * 由一些云供应商提供 (如AWS ELB、 Google Cloud Load Balancer)
   * Kubernetes service 为 LoadBalancer 类型时自动创建
   * 只使用 UDP/TCP 协议
   * 不同云供应商的实现不同。

Kubernetes 用户通常只需要关心前两种类型的代理，集群管理员通常需要确保后面几种类型的代理设置正确。

请求重定向

代理已经取代重定向功能，重定向已被弃用。

反馈

安装扩展（Addons）

Add-ons 扩展了 Kubernetes 的功能。

本文列举了一些可用的 add-ons 以及到它们各自安装说明的链接。

每个 add-ons 按字母顺序排序 - 顺序不代表任何优先地位。

* [**网络和网络策略**](https://kubernetes.io/zh/docs/concepts/cluster-administration/addons/#%e7%bd%91%e7%bb%9c%e5%92%8c%e7%bd%91%e7%bb%9c%e7%ad%96%e7%95%a5)
* [**服务发现**](https://kubernetes.io/zh/docs/concepts/cluster-administration/addons/#%e6%9c%8d%e5%8a%a1%e5%8f%91%e7%8e%b0)
* [**可视化管理**](https://kubernetes.io/zh/docs/concepts/cluster-administration/addons/#%e5%8f%af%e8%a7%86%e5%8c%96%e7%ae%a1%e7%90%86)
* [**基础设施**](https://kubernetes.io/zh/docs/concepts/cluster-administration/addons/#%e5%9f%ba%e7%a1%80%e8%ae%be%e6%96%bd)
* [**遗留 Add-ons**](https://kubernetes.io/zh/docs/concepts/cluster-administration/addons/#%e9%81%97%e7%95%99-add-ons)

网络和网络策略

* [ACI](https://www.github.com/noironetworks/aci-containers) 通过 Cisco ACI 提供集成的容器网络和安全网络。
* [Calico](https://docs.projectcalico.org/v3.11/getting-started/kubernetes/installation/calico) 是一个安全的 L3 网络和网络策略提供者。
* [Canal](https://github.com/tigera/canal/tree/master/k8s-install) 结合 Flannel 和 Calico， 提供网络和网络策略。
* [Cilium](https://github.com/cilium/cilium) 是一个 L3 网络和网络策略插件， 能够透明的实施 HTTP/API/L7 策略。 同时支持路由（routing）和叠加/封装（ overlay/encapsulation）模式。
* [CNI-Genie](https://github.com/Huawei-PaaS/CNI-Genie) 使 Kubernetes 无缝连接到一种 CNI 插件，例如：Flannel、Calico、Canal、Romana 或者 Weave。
* [Contiv](http://contiv.github.io/) 为多种用例提供可配置网络（使用 BGP 的原生 L3，使用 vxlan 的 overlay，经典 L2 和 Cisco-SDN/ACI）和丰富的策略框架。Contiv 项目完全[开源](http://github.com/contiv)。[安装工具](http://github.com/contiv/install)同时提供基于和不基于 kubeadm 的安装选项。
* [Contrail](http://www.juniper.net/us/en/products-services/sdn/contrail/contrail-networking/), based on [Tungsten Fabric](https://tungsten.io/), is an open source, multi-cloud network virtualization and policy management platform. Contrail and Tungsten Fabric are integrated with orchestration systems such as Kubernetes, OpenShift, OpenStack and Mesos, and provide isolation modes for virtual machines, containers/pods and bare metal workloads.
* [Flannel](https://github.com/coreos/flannel/blob/master/Documentation/kube-flannel.yml) 是一个可以用于 Kubernetes 的 overlay 网络提供者。
* [Knitter](https://github.com/ZTE/Knitter/) 是为 kubernetes 提供复合网络解决方案的网络组件.
* [Multus](https://github.com/Intel-Corp/multus-cni) 是一个多插件，可在 Kubernetes 中提供多种网络支持，以支持所有 CNI 插件（例如 Calico，Cilium，Contiv，Flannel）is a Multi plugin for multiple network support in Kubernetes to support all CNI plugins (e.g. Calico, Cilium, Contiv, Flannel)，而且包含了在Kubernetes中基于 SRIOV，DPDK，OVS-DPDK 和 VPP 的工作负载.
* [NSX-T](https://docs.vmware.com/en/VMware-NSX-T/2.0/nsxt_20_ncp_kubernetes.pdf) 容器插件（ NCP ）提供了 VMware NSX-T 与容器协调器（例如 Kubernetes）之间的集成，以及 NSX-T 与基于容器的 CaaS / PaaS 平台（例如关键容器服务（ PKS ）和 OpenShift ）之间的集成。
* [Nuage](https://github.com/nuagenetworks/nuage-kubernetes/blob/v5.1.1-1/docs/kubernetes-1-installation.rst) 是一个SDN平台，可在Kubernetes Pods和非Kubernetes环境之间提供基于策略的联网，并具有可视化和安全监控。
* [Romana](http://romana.io/) 是一个 pod 网络的层 3 解决方案，并且支持 [NetworkPolicy API](https://kubernetes.io/docs/concepts/services-networking/network-policies/)。Kubeadm add-on 安装细节可以在[这里](https://github.com/romana/romana/tree/master/containerize)找到。
* [Weave Net](https://www.weave.works/docs/net/latest/kube-addon/) 提供了在网络分组两端参与工作的网络和网络策略，并且不需要额外的数据库。

服务发现

* [CoreDNS](https://coredns.io/) 是一种灵活的，可扩展的 DNS 服务器，可以 [安装](https://github.com/coredns/deployment/tree/master/kubernetes) 为集群内的 Pod 提供 DNS 服务。

可视化管理

* [Dashboard](https://github.com/kubernetes/dashboard#kubernetes-dashboard) 是一个 Kubernetes 的 web 控制台界面。
* [Weave Scope](https://www.weave.works/documentation/scope-latest-installing/#k8s) 是一个图形化工具，用于查看你的 containers、 pods、services 等。 请和一个 [Weave Cloud account](https://cloud.weave.works/) 一起使用，或者自己运行 UI。

基础设施

* [KubeVirt](https://kubevirt.io/user-guide/docs/latest/administration/intro.html#cluster-side-add-on-deployment) 是可以让 Kubernetes 运行虚拟机的 add-ons 。通常运行在裸机群集上。

遗留 Add-ons

还有一些其它 add-ons 归档在已废弃的 [cluster/addons](https://git.k8s.io/kubernetes/cluster/addons) 路径中。

维护完善的 add-ons 应该被链接到这里。欢迎提出 PRs！

反馈

联邦

本页面阐明了为何以及如何使用联邦创建Kubernetes集群。

* [**为何使用联邦**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e4%b8%ba%e4%bd%95%e4%bd%bf%e7%94%a8%e8%81%94%e9%82%a6)
* [**建立联邦**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e5%bb%ba%e7%ab%8b%e8%81%94%e9%82%a6)
* [**API资源**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#api%e8%b5%84%e6%ba%90)
* [**级联删除**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e7%ba%a7%e8%81%94%e5%88%a0%e9%99%a4)
* [**单个集群的范围**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e5%8d%95%e4%b8%aa%e9%9b%86%e7%be%a4%e7%9a%84%e8%8c%83%e5%9b%b4)
* [**选择合适的集群数**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e9%80%89%e6%8b%a9%e5%90%88%e9%80%82%e7%9a%84%e9%9b%86%e7%be%a4%e6%95%b0)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

为何使用联邦

联邦可以使多个集群的管理简单化。它提供了两个主要构件模块：

* 跨集群同步资源：联邦能够让资源在多个集群中同步。例如，你可以确保在多个集群中存在同样的部署。
* 跨集群发现：联邦能够在所有集群的后端自动配置DNS服务和负载均衡。例如，通过多个集群的后端，你可以确保全局的VIP或DNS记录可用。

联邦技术的其他应用场景：

* 高可用性：通过跨集群分摊负载，自动配置DNS服务和负载均衡，联邦将集群失败所带来的影响降到最低。
* 避免供应商锁定：跨集群使迁移应用程序变得更容易，联邦服务避免了供应商锁定。

只有在多个集群的场景下联邦服务才是有帮助的。这里列出了一些你会使用多个集群的原因：

* 降低延迟：在多个区域含有集群，可使用离用户最近的集群来服务用户，从而最大限度降低延迟。
* 故障隔离：对于故障隔离，也许有多个小的集群比有一个大的集群要更好一些（例如：一个云供应商的不同可用域里有多个集群）。详细信息请参阅[多集群指南](https://kubernetes.io/docs/admin/multi-cluster)。
* 可伸缩性：对于单个kubernetes集群是有伸缩性限制的（但对于大多数用户来说并非如此。更多细节参考[Kubernetes扩展和性能目标](https://git.k8s.io/community/sig-scalability/goals.md)）。
* [混合云](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#混合云的能力)：可以有多个集群，它们分别拥有不同的云供应商或者本地数据中心。

注意事项

虽然联邦有很多吸引人的场景，但这里还是有一些需要关注的事项：

* 增加网络的带宽和损耗：联邦控制面会监控所有的集群，来确保集群的当前状态与预期一致。那么当这些集群运行在一个或者多个云提供者的不同区域中，则会带来重大的网络损耗。
* 降低集群的隔离：当联邦控制面中存在一个故障时，会影响所有的集群。把联邦控制面的逻辑降到最小可以缓解这个问题。 无论何时，它都是kubernetes集群里控制面的代表。设计和实现也使其变得更安全,避免多集群运行中断。
* 完整性：联邦项目相对较新，还不是很成熟。不是所有资源都可用，且很多资源才刚刚开始。[Issue 38893](https://github.com/kubernetes/kubernetes/issues/38893) 列举了一些团队正忙于解决的系统已知问题。

混合云的能力

Kubernetes集群里的联邦包括运行在不同云供应商上的集群（例如，谷歌云、亚马逊），和本地部署的集群（例如，OpenStack）。只需在适当的云供应商和/或位置创建所需的所有集群，并将每个集群的API endpoint和凭据注册到您的联邦API服务中（详情参考[联邦管理指南](https://kubernetes.io/docs/admin/federation/)）。

在此之后，您的[API资源](https://kubernetes.io/zh/docs/concepts/cluster-administration/federation/#api资源)就可以跨越不同的集群和云供应商。

建立联邦

若要能联合多个集群，首先需要建立一个联邦控制面。参照[安装指南](https://kubernetes.io/docs/tutorials/federation/set-up-cluster-federation-kubefed/) 建立联邦控制面。

API资源

控制面建立完成后，就可以开始创建联邦API资源了。 以下指南详细介绍了一些资源：

* [Cluster](https://kubernetes.io/docs/tasks/administer-federation/cluster/)
* [ConfigMap](https://kubernetes.io/docs/tasks/administer-federation/configmap/)
* [DaemonSets](https://kubernetes.io/docs/tasks/administer-federation/daemonset/)
* [Deployment](https://kubernetes.io/docs/tasks/administer-federation/deployment/)
* [Events](https://kubernetes.io/docs/tasks/administer-federation/events/)
* [Ingress](https://kubernetes.io/docs/tasks/administer-federation/ingress/)
* [Namespaces](https://kubernetes.io/docs/tasks/administer-federation/namespaces/)
* [ReplicaSets](https://kubernetes.io/docs/tasks/administer-federation/replicaset/)
* [Secrets](https://kubernetes.io/docs/tasks/administer-federation/secret/)
* [Services](https://kubernetes.io/docs/concepts/cluster-administration/federation-service-discovery/)

[API参考文档](https://kubernetes.io/docs/reference/federation/)列举了联邦API服务支持的所有资源。

级联删除

Kubernetes1.6版本支持联邦资源级联删除。使用级联删除，即当删除联邦控制面的一个资源时，也删除了所有底层集群中的相应资源。

当使用REST API时，级联删除功能不是默认开启的。若使用REST API从联邦控制面删除一个资源时，要开启级联删除功能，即需配置选项 DeleteOptions.orphanDependents=false。使用kubectl delete使级联删除功能默认开启。使用kubectl delete --cascade=false禁用级联删除功能。

注意：Kubernetes1.5版本开始支持联邦资源子集的级联删除。

单个集群的范围

对于IaaS供应商如谷歌计算引擎或亚马逊网络服务，一个虚拟机存在于一个[域](https://cloud.google.com/compute/docs/zones)或[可用域](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html)中。 我们建议一个Kubernetes集群里的所有虚机应该在相同的可用域里，因为：

* 与单一的全局Kubernetes集群对比，该方式有较少的单点故障。
* 与跨可用域的集群对比，该方式更容易推断单区域集群的可用性属性。
* 当Kubernetes开发者设计一个系统(例如，对延迟、带宽或相关故障进行假设)，他们也会假设所有的机器都在一个单一的数据中心，或者以其他方式紧密相连。

每个可用区域里包含多个集群当然是可以的，但是总的来说我们认为集群数越少越好。 偏爱较少集群数的原因是：

* 在某些情况下，在一个集群里有更多的节点，可以改进Pods的装箱问题（更少的资源碎片）。
* 减少操作开销（尽管随着OPS工具和流程的成熟而降低了这块的优势）。
* 为每个集群的固定资源花费降低开销，例如，使用apiserver的虚拟机（但是在全体集群开销中，中小型集群的开销占比要小的多）。

多集群的原因包括：

* 严格的安全性策略要求隔离一类工作与另一类工作（但是，请参见下面的集群分割）。
* 测试集群或其他集群软件直至最优的新Kubernetes版本发布。

选择合适的集群数

Kubernetes集群数量选择也许是一个相对静止的选择，因为对其重新审核的情况很少。相比之下，一个集群中的节点数和一个服务中的pods数可能会根据负载和增长频繁变化。

选择集群的数量，首先，需要决定哪些区域对于将要运行在Kubernetes上的服务，可以有足够的时间到达所有的终端用户（如果使用内容分发网络，则不需要考虑CDN-hosted内容的延迟需求）。法律问题也可能影响这一点。例如，拥有全球客户群的公司可能会对于在美国、欧盟、亚太和南非地区拥有集群起到决定权。使用R代表区域的数量。

其次，决定有多少集群在同一时间不可用，而一些仍然可用。使用U代表不可用的数量。如果不确定，最好选择1。

如果允许负载均衡在集群故障发生时将通信引导到任何区域，那么至少需要较大的R或U + 1集群。若非如此（例如，若要在集群故障发生时确保所有用户的低延迟），则需要R \* (U + 1)集群(在每一个R区域里都有U + 1)。在任何情况下，尝试将每个集群放在不同的区域中。

最后，如果你的集群需求超过一个Kubernetes集群推荐的最大节点数，那么你可能需要更多的集群。Kubernetes1.3版本支持多达1000个节点的集群规模。

反馈

扩展 Kubernetes

扩展 Kubernetes 集群

Kubernetes 是高度可配置和可扩展的。因此，极少需要分发或提交补丁代码给 Kubernetes 项目。

本文档介绍自定义 Kubernetes 集群的选项。本文档的目标读者 [cluster operators](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-cluster-operator) 是希望了解如何使 Kubernetes 集群满足其业务环境需求的集群运维人员。Kubernetes 项目的贡献者 [Contributors](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-contributor) 或潜在的平台开发人员 [Platform Developers](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-platform-developer) 也可以从本文找到有用的信息，如对已存在扩展点和模式的介绍，以及它们的权衡和限制。

* [**概述**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e6%a6%82%e8%bf%b0)
* [**配置**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e9%85%8d%e7%bd%ae)
* [**扩展程序**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e6%89%a9%e5%b1%95%e7%a8%8b%e5%ba%8f)
* [**扩展模式**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e6%89%a9%e5%b1%95%e6%a8%a1%e5%bc%8f)
* [**扩展点**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e6%89%a9%e5%b1%95%e7%82%b9)
* [**API 扩展**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#api-%e6%89%a9%e5%b1%95)
* [**基础设施扩展**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e5%9f%ba%e7%a1%80%e8%ae%be%e6%96%bd%e6%89%a9%e5%b1%95)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

概述

定制方法可以大致分为 *配置* 和 *扩展* 。*配置* 只涉及更改标志参数、本地配置文件或 API 资源；*扩展* 涉及运行额外的程序或服务。本文档主要内容是关于扩展。

配置

关于 *配置文件* 和 *标志* 的说明文档位于在线文档的参考部分，按照二进制组件各自描述：

* [kubelet](https://kubernetes.io/docs/admin/kubelet/)
* [kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/)
* [kube-controller-manager](https://kubernetes.io/docs/admin/kube-controller-manager/)
* [kube-scheduler](https://kubernetes.io/docs/admin/kube-scheduler/).

在托管的 Kubernetes 服务或受控安装的 Kubernetes 版本中，标志和配置文件可能并不总是可以更改的。而且当它们可以进行更改时，它们通常只能由集群管理员进行更改。此外，标志和配置文件在未来的 Kubernetes 版本中可能会发生变化，并且更改设置后它们可能需要重新启动进程。出于这些原因，只有在没有其他选择的情况下才使用它们。

*内置策略 API* ，例如 [ResourceQuota](https://kubernetes.io/docs/concepts/policy/resource-quotas/)、[PodSecurityPolicy](https://kubernetes.io/docs/concepts/policy/pod-security-policy/)、[NetworkPolicy](https://kubernetes.io/docs/concepts/services-networking/network-policies/) 和基于角色的权限控制 ([RBAC](https://kubernetes.io/docs/reference/access-authn-authz/rbac/))，是内置的 Kubernetes API。API 通常与托管的 Kubernetes 服务和受控的 Kubernetes 安装一起使用。 它们是声明性的，并使用与其他 Kubernetes 资源（如 Pod ）相同的约定，所以新的集群配置可以重复使用，并以与应用程序相同的方式进行管理。而且，当他们变稳定后，他们和其他 Kubernetes API 一样享受[定义支持政策](https://kubernetes.io/docs/reference/deprecation-policy/)。出于这些原因，在合适的情况下它们优先于 *配置文件* 和 *标志* 被使用。

扩展程序

扩展程序是指对 Kubernetes 进行扩展和深度集成的软件组件。它们适合用于支持新的类型和新型硬件。

大多数集群管理员会使用托管的或统一分发的 Kubernetes 实例。因此，大多数 Kubernetes 用户需要安装扩展程序，而且还有少部分用户甚至需要编写新的扩展程序。

扩展模式

Kubernetes 的设计是通过编写客户端程序来实现自动化的。任何读和（或）写 Kubernetes API 的程序都可以提供有用的自动化工作。*自动化* 程序可以运行在集群之中或之外。按照本文档的指导，您可以编写出高可用的和健壮的自动化程序。自动化程序通常适用于任何 Kubernetes 集群，包括托管集群和受管理安装的集群。

*控制器* 模式是编写适合 Kubernetes 的客户端程序的一种特定模式。控制器通常读取一个对象的 .spec 字段，可能做出一些处理，然后更新对象的 .status 字段。

一个控制器是 Kubernetes 的一个客户端。而当 Kubernetes 作为客户端调用远程服务时，它被称为 *Webhook* ，远程服务称为 *Webhook* 后端。 和控制器类似，Webhooks 增加了一个失败点。

在 webhook 模型里，Kubernetes 向远程服务发送一个网络请求。在 *二进制插件* 模型里，Kubernetes 执行一个二进制（程序）。二进制插件被 kubelet（如 [Flex 卷插件](https://github.com/kubernetes/community/blob/master/contributors/devel/flexvolume.md)和[网络插件](https://kubernetes.io/docs/concepts/cluster-administration/network-plugins/))和 kubectl 所使用。

下图显示了扩展点如何与 Kubernetes 控制平面进行交互。

扩展点

下图显示了 Kubernetes 系统的扩展点。

1. 用户通常使用 kubectl 与 Kubernetes API 进行交互。[kubectl 插件](https://kubernetes.io/docs/tasks/extend-kubectl/kubectl-plugins/)扩展了 kubectl 二进制程序。它们只影响个人用户的本地环境，因此不能执行站点范围的策略。
2. apiserver 处理所有请求。apiserver 中的几种类型的扩展点允许对请求进行身份认证或根据其内容对其进行阻止、编辑内容以及处理删除操作。这些内容在[API 访问扩展](https://kubernetes.io/docs/concepts/overview/extending#api-access-extensions)小节中描述。 <!–
3. The apiserver serves various kinds of *resources*. *Built-in resource kinds*, like pods, are defined by the Kubernetes project and can’t be changed. You can also add resources that you define, or that other projects have defined, called *Custom Resources*, as explained in the [Custom Resources](https://kubernetes.io/docs/concepts/overview/extending#user-defined-types) section. Custom Resources are often used with API Access Extensions.
4. The Kubernetes scheduler decides which nodes to place pods on. There are several ways to extend scheduling. These are described in the [Scheduler Extensions](https://kubernetes.io/docs/concepts/overview/extending#scheduler-extensions) section. –>
5. apiserver 提供各种 *资源* 。 *内置的资源种类* ，如 pods，由 Kubernetes 项目定义，不能更改。您还可以添加您自己定义的资源或其他项目已定义的资源，称为 自定义资源，如[自定义资源](https://kubernetes.io/docs/concepts/overview/extending#user-defined-types)部分所述。自定义资源通常与 API 访问扩展一起使用。
6. Kubernetes 调度器决定将 Pod 放置到哪个节点。有几种方法可以扩展调度器。这些内容在 [Scheduler Extensions](https://kubernetes.io/docs/concepts/overview/extending#scheduler-extensions) 小节中描述。 <!–
7. Much of the behavior of Kubernetes is implemented by programs called Controllers which are clients of the API-Server. Controllers are often used in conjunction with Custom Resources.
8. The kubelet runs on servers, and helps pods appear like virtual servers with their own IPs on the cluster network. [Network Plugins](https://kubernetes.io/docs/concepts/overview/extending#network-plugins) allow for different implementations of pod networking.
9. The kubelet also mounts and unmounts volumes for containers. New types of storage can be supported via [Storage Plugins](https://kubernetes.io/docs/concepts/overview/extending#storage-plugins). –>
10. Kubernetes 的大部分行为都是由称为控制器的程序实现的，这些程序是 API-Server 的客户端。控制器通常与自定义资源一起使用。
11. kubelet 在主机上运行，并帮助 pod 看起来就像在集群网络上拥有自己的 IP 的虚拟服务器。[网络插件](https://kubernetes.io/docs/concepts/overview/extending#network-plugins)让您可以实现不同的 pod 网络。
12. kubelet 也挂载和卸载容器的卷。新的存储类型可以通过[存储插件](https://kubernetes.io/docs/concepts/overview/extending#storage-plugins)支持。

如果您不确定从哪里开始扩展，此流程图可以提供帮助。请注意，某些解决方案可能涉及多种类型的扩展。

API 扩展

用户自定义类型

如果您想定义新的控制器、应用程序配置对象或其他声明式 API，并使用 Kubernetes 工具（如 kubectl）管理它们，请考虑为 Kubernetes 添加一个自定义资源。

不要使用自定义资源作为应用、用户或者监控数据的数据存储。

有关自定义资源的更多信息，请查看[自定义资源概念指南](https://kubernetes.io/docs/concepts/api-extension/custom-resources/)。

将新的 API 与自动化相结合

自定义资源 API 和控制循环的组合称为 [操作者模式](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/)。操作者模式用于管理特定的，通常是有状态的应用程序。这些自定义 API 和控制循环还可用于控制其他资源，例如存储或策略。

改变内置资源

当您通过添加自定义资源来扩展 Kubernetes API 时，添加的资源始终属于新的 API 组。您不能替换或更改已有的 API 组。添加 API 不会直接影响现有 API（例如 Pod ）的行为，但是 API 访问扩展可以。

API 访问扩展

当请求到达 Kubernetes API Server 时，它首先被要求进行用户认证，然后要进行授权检查，接着受到各种类型的准入控制的检查。有关此流程的更多信息，请参阅 [Kubernetes API访问控制](https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/)。

上述每个步骤都提供了扩展点。

Kubernetes 有几个它支持的内置认证方法。它还可以位于身份验证代理之后，并将授权 header 中的令牌发送给远程服务进行验证（webhook）。所有这些方法都在[身份验证文档](https://kubernetes.io/docs/reference/access-authn-authz/authentication/)中介绍。

身份认证

[身份认证](https://kubernetes.io/docs/reference/access-authn-authz/authentication/)将所有请求中的 header 或证书映射为发出请求的客户端的用户名。

Kubernetes 提供了几种内置的身份认证方法，如果这些方法不符合您的需求，可以使用[身份认证 webhook](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#webhook-token-authentication) 方法。

授权

[授权](https://kubernetes.io/docs/reference/access-authn-authz/webhook/)决定特定用户是否可以对 API 资源执行读取、写入以及其他操作。它只是在整个资源的层面上工作 – 它不基于任意的对象字段进行区分。如果内置授权选项不能满足您的需求，[授权 webhook](https://kubernetes.io/docs/reference/access-authn-authz/webhook/) 允许调用用户提供的代码来作出授权决定。

动态准入控制

在请求被授权之后，如果是写入操作，它还将进入[准入控制](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/)步骤。除了内置的步骤之外，还有几个扩展：

* [镜像策略 webhook](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#imagepolicywebhook) 限制了哪些镜像可以在容器中运行。
* 为了进行灵活的准入控制决策，可以使用通用的 [Admission webhook](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks)。Admission Webhooks 可以拒绝创建或更新操作。

基础设施扩展

存储插件

[Flex Volumes](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/storage/flexvolume-deployment.md) 允许用户挂载无内置插件支持的卷类型，它通过 Kubelet 调用一个二进制插件来挂载卷。

设备插件

设备插件允许节点通过[设备插件](https://kubernetes.io/docs/concepts/cluster-administration/device-plugins/)发现新的节点资源（除了内置的 CPU 和内存之外）。

网络插件

不同的网络结构可以通过节点级的[网络插件](https://kubernetes.io/docs/admin/network-plugins/)支持。

调度器扩展

调度器是一种特殊类型的控制器，用于监视 pod 并将其分配到节点。默认的调度器可以完全被替换，而继续使用其他 Kubernetes 组件，或者可以同时运行[多个调度器](https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/)。

这是一个重要的任务，几乎所有的 Kubernetes 用户都发现他们不需要修改调度器。

调度器也支持 [webhook](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md)，它允许一个 webhook 后端（调度器扩展程序）为 pod 筛选节点和确定节点的优先级。

* 详细了解[自定义资源](https://kubernetes.io/docs/concepts/api-extension/custom-resources/)
* 了解[动态准入控制](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/)
* 详细了解基础设施扩展
  + [网络插件](https://kubernetes.io/docs/concepts/cluster-administration/network-plugins/)
  + [设备插件](https://kubernetes.io/docs/concepts/cluster-administration/device-plugins/)
* 了解 [kubectl 插件](https://kubernetes.io/docs/tasks/extend-kubectl/kubectl-plugins/)
* 了解[操作者模式](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/)

扩展 Kubernetes API

通过聚合层扩展 Kubernetes API

聚合层允许 Kubernetes 通过额外的 API 进行扩展，而不局限于 Kubernetes 核心 API 提供的功能。

* [**概述**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/api-extension/#%e6%a6%82%e8%bf%b0)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/api-extension/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

概述

聚合层使您的集群可以安装其他 Kubernetes 风格的 API。这些 API 可以是预编译的、第三方的解决方案提供的例如[service-catalog](https://github.com/kubernetes-incubator/service-catalog/blob/master/README.md)、或者用户创建的类似[apiserver-builder](https://github.com/kubernetes-incubator/apiserver-builder/blob/master/README.md)一样的API可以帮助你上手。

聚合层在 kube-apiserver 进程内运行。在扩展资源注册之前，聚合层不做任何事情。要注册 API，用户必须添加一个 APIService 对象，用它来申领 Kubernetes API 中的 URL 路径。自此以后，聚合层将会把发给该 API 路径的所有内容（例如 /apis/myextension.mycompany.io/v1/…）代理到已注册的 APIService。

正常情况下，APIService 会实现为运行于集群中某 Pod 内的 extension-apiserver。如果需要对增加的资源进行动态管理，extension-apiserver 经常需要和一个或多个控制器一起使用。因此，apiserver-builder 同时提供用来管理新资源的 API 框架和控制器框架。另外一个例子，当安装了 service-catalog 时，它会为自己提供的服务提供 extension-apiserver 和控制器。

extension-apiserver 与 kube-apiserver 之间的连接应具有低延迟。 特别是，发现请求需要在五秒钟或更短的时间内从 kube-apiserver 往返。 如果您的部署无法实现此目的，则应考虑如何进行更改。目前，在 kube-apiserver 上设置 EnableAggregatedDiscoveryTimeout=false 功能开关将禁用超时限制。它将在将来的版本中被删除。

Custom Resources

*Custom resources* are extensions of the Kubernetes API. This page discusses when to add a custom resource to your Kubernetes cluster and when to use a standalone service. It describes the two methods for adding custom resources and how to choose between them.

* [**Custom resources**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#custom-resources)
* [**Custom controllers**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#custom-controllers)
* [**Should I add a custom resource to my Kubernetes Cluster?**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#should-i-add-a-custom-resource-to-my-kubernetes-cluster)
* [**Should I use a configMap or a custom resource?**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#should-i-use-a-configmap-or-a-custom-resource)
* [**Adding custom resources**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#adding-custom-resources)
* [**CustomResourceDefinitions**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions)
* [**API server aggregation**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#api-server-aggregation)
* [**Choosing a method for adding custom resources**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#choosing-a-method-for-adding-custom-resources)
* [**Preparing to install a custom resource**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#preparing-to-install-a-custom-resource)
* [**Accessing a custom resource**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#accessing-a-custom-resource)
* [**What's next**](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#what-s-next)

Custom resources

A *resource* is an endpoint in the [Kubernetes API](https://kubernetes.io/docs/reference/using-api/api-overview/) that stores a collection of [API objects](https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/) of a certain kind. For example, the built-in *pods* resource contains a collection of Pod objects.

A *custom resource* is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using [kubectl](https://kubernetes.io/docs/user-guide/kubectl-overview/), just as they do for built-in resources like *Pods*.

Custom controllers

On their own, custom resources simply let you store and retrieve structured data. When you combine a custom resource with a *custom controller*, custom resources provide a true *declarative API*.

A [declarative API](https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/#understanding-kubernetes-objects) allows you to *declare* or specify the desired state of your resource and tries to keep the current state of Kubernetes objects in sync with the desired state. The controller interprets the structured data as a record of the user’s desired state, and continually maintains this state.

You can deploy and update a custom controller on a running cluster, independently of the cluster’s own lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The [Operator pattern](https://coreos.com/blog/introducing-operators.html) combines custom resources and custom controllers. You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Should I add a custom resource to my Kubernetes Cluster?

When creating a new API, consider whether to [aggregate your API with the Kubernetes cluster APIs](https://kubernetes.io/docs/concepts/api-extension/apiserver-aggregation/) or let your API stand alone.

| Consider API aggregation if: | Prefer a stand-alone API if: |
| --- | --- |
| Your API is [Declarative](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#declarative-apis). | Your API does not fit the [Declarative](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#declarative-apis) model. |
| You want your new types to be readable and writable using kubectl. | kubectl support is not required |
| You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types. | Kubernetes UI support is not required. |
| You are developing a new API. | You already have a program that serves your API and works well. |
| You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the [API Overview](https://kubernetes.io/docs/concepts/overview/kubernetes-api/).) | You need to have specific REST paths to be compatible with an already defined REST API. |
| Your resources are naturally scoped to a cluster or to namespaces of a cluster. | Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths. |
| You want to reuse [Kubernetes API support features](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#common-features). | You don’t need those features. |

Declarative APIs

In a Declarative API, typically:

* Your API consists of a relatively small number of relatively small objects (resources).
* The objects define configuration of applications or infrastructure.
* The objects are updated relatively infrequently.
* Humans often need to read and write the objects.
* The main operations on the objects are CRUD-y (creating, reading, updating and deleting).
* Transactions across objects are not required: the API represents a desired state, not an exact state.

Imperative APIs are not declarative. Signs that your API might not be declarative include:

* The client says “do this”, and then gets a synchronous response back when it is done.
* The client says “do this”, and then gets an operation ID back, and has to check a separate Operation object to determine completion of the request.
* You talk about Remote Procedure Calls (RPCs).
* Directly storing large amounts of data (e.g. > a few kB per object, or >1000s of objects).
* High bandwidth access (10s of requests per second sustained) needed.
* Store end-user data (such as images, PII, etc) or other large-scale data processed by applications.
* The natural operations on the objects are not CRUD-y.
* The API is not easily modeled as objects.
* You chose to represent pending operations with an operation ID or an operation object.

Should I use a configMap or a custom resource?

Use a ConfigMap if any of the following apply:

* There is an existing, well-documented config file format, such as a mysql.cnf or pom.xml.
* You want to put the entire config file into one key of a configMap.
* The main use of the config file is for a program running in a Pod on your cluster to consume the file to configure itself.
* Consumers of the file prefer to consume via file in a Pod or environment variable in a pod, rather than the Kubernetes API.
* You want to perform rolling updates via Deployment, etc, when the file is updated.

**Note:** Use a [secret](https://kubernetes.io/docs/concepts/configuration/secret/) for sensitive data, which is similar to a configMap but more secure.

Use a custom resource (CRD or Aggregated API) if most of the following apply:

* You want to use Kubernetes client libraries and CLIs to create and update the new resource.
* You want top-level support from kubectl (for example: kubectl get my-object object-name).
* You want to build new automation that watches for updates on the new object, and then CRUD other objects, or vice versa.
* You want to write automation that handles updates to the object.
* You want to use Kubernetes API conventions like .spec, .status, and .metadata.
* You want the object to be an abstraction over a collection of controlled resources, or a summarization of other resources.

Adding custom resources

Kubernetes provides two ways to add custom resources to your cluster:

* CRDs are simple and can be created without any programming.
* [API Aggregation](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/) requires programming, but allows more control over API behaviors like how data is stored and conversion between API versions.

Kubernetes provides these two options to meet the needs of different users, so that neither ease of use nor flexibility is compromised.

Aggregated APIs are subordinate APIServers that sit behind the primary API server, which acts as a proxy. This arrangement is called [API Aggregation](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/) (AA). To users, it simply appears that the Kubernetes API is extended.

CRDs allow users to create new types of resources without adding another APIserver. You do not need to understand API Aggregation to use CRDs.

Regardless of how they are installed, the new resources are referred to as Custom Resources to distinguish them from built-in Kubernetes resources (like pods).

CustomResourceDefinitions

The [CustomResourceDefinition](https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/) API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource.

This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with [API server aggregation](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#api-server-aggregation).

Refer to the [custom controller example](https://github.com/kubernetes/sample-controller) for an example of how to register a new custom resource, work with instances of your new resource type, and use a controller to handle events.

API server aggregation

Usually, each resource in the Kubernetes API requires code that handles REST requests and manages persistent storage of objects. The main Kubernetes API server handles built-in resources like *pods* and *services*, and can also handle custom resources in a generic way through [CRDs](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#customresourcedefinitions).

The [aggregation layer](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/) allows you to provide specialized implementations for your custom resources by writing and deploying your own standalone API server. The main API server delegates requests to you for the custom resources that you handle, making them available to all of its clients.

Choosing a method for adding custom resources

CRDs are easier to use. Aggregated APIs are more flexible. Choose the method that best meets your needs.

Typically, CRDs are a good fit if:

* You have a handful of fields
* You are using the resource within your company, or as part of a small open-source project (as opposed to a commercial product)

Comparing ease of use

CRDs are easier to create than Aggregated APIs.

| CRDs | Aggregated API |
| --- | --- |
| Do not require programming. Users can choose any language for a CRD controller. | Requires programming in Go and building binary and image. Users can choose any language for a CRD controller. |
| No additional service to run; CRs are handled by API Server. | An additional service to create and that could fail. |
| No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades. | May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated APIserver. |
| No need to handle multiple versions of your API. For example: when you control the client for this resource, you can upgrade it in sync with the API. | You need to handle multiple versions of your API, for example: when developing an extension to share with the world. |

Advanced features and flexibility

Aggregated APIs offer more advanced API features and customization of other features, for example: the storage layer.

| Feature | Description | CRDs | Aggregated API |
| --- | --- | --- | --- |
| Validation | Help users prevent errors and allow you to evolve your API independently of your clients. These features are most useful when there are many clients who can’t all update at the same time. | Yes. Most validation can be specified in the CRD using [OpenAPI v3.0 validation](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/#validation). Any other validations supported by addition of a [Validating Webhook](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#validatingadmissionwebhook-alpha-in-1-8-beta-in-1-9). | Yes, arbitrary validation checks |
| Defaulting | See above | Yes, either via [OpenAPI v3.0 validation](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/#defaulting) default keyword (GA in 1.17), or via a [Mutating Webhook](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#mutatingadmissionwebhook) (though this will not be run when reading from etcd for old objects) | Yes |
| Multi-versioning | Allows serving the same object through two API versions. Can help ease API changes like renaming fields. Less important if you control your client versions. | [Yes](https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definition-versioning) | Yes |
| Custom Storage | If you need storage with a different performance mode (for example, time-series database instead of key-value store) or isolation for security (for example, encryption secrets or different | No | Yes |
| Custom Business Logic | Perform arbitrary checks or actions when creating, reading, updating or deleting an object | Yes, using [Webhooks](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks). | Yes |
| Scale Subresource | Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource | [Yes](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/#scale-subresource) | Yes |
| Status Subresource | * Finer-grained access control: user writes spec section, controller writes status section. * Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource) | [Yes](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/#status-subresource) | Yes |
| Other Subresources | Add operations other than CRUD, such as “logs” or “exec”. | No | Yes |
| strategic-merge-patch | The new endpoints support PATCH with Content-Type: application/strategic-merge-patch+json. Useful for updating objects that may be modified both locally, and by the server. For more information, see [“Update API Objects in Place Using kubectl patch”](https://kubernetes.io/docs/tasks/run-application/update-api-object-kubectl-patch/) | No | Yes |
| Protocol Buffers | The new resource supports clients that want to use Protocol Buffers | No | Yes |
| OpenAPI Schema | Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (in other words, don’t put an int in a string field?) | Yes, based on the [OpenAPI v3.0 validation](https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/#validation) schema (GA in 1.16) | Yes |

Common Features

When you create a custom resource, either via a CRDs or an AA, you get many features for your API, compared to implementing it outside the Kubernetes platform:

| Feature | What it does |
| --- | --- |
| CRUD | The new endpoints support CRUD basic operations via HTTP and kubectl |
| Watch | The new endpoints support Kubernetes Watch operations via HTTP |
| Discovery | Clients like kubectl and dashboard automatically offer list, display, and field edit operations on your resources |
| json-patch | The new endpoints support PATCH with Content-Type: application/json-patch+json |
| merge-patch | The new endpoints support PATCH with Content-Type: application/merge-patch+json |
| HTTPS | The new endpoints uses HTTPS |
| Built-in Authentication | Access to the extension uses the core apiserver (aggregation layer) for authentication |
| Built-in Authorization | Access to the extension can reuse the authorization used by the core apiserver (e.g. RBAC) |
| Finalizers | Block deletion of extension resources until external cleanup happens. |
| Admission Webhooks | Set default values and validate extension resources during any create/update/delete operation. |
| UI/CLI Display | Kubectl, dashboard can display extension resources. |
| Unset vs Empty | Clients can distinguish unset fields from zero-valued fields. |
| Client Libraries Generation | Kubernetes provides generic client libraries, as well as tools to generate type-specific client libraries. |
| Labels and annotations | Common metadata across objects that tools know how to edit for core and custom resources. |

Preparing to install a custom resource

There are several points to be aware of before adding a custom resource to your cluster.

Third party code and new points of failure

While creating a CRD does not automatically add any new points of failure (for example, by causing third party code to run on your API server), packages (for example, Charts) or other installation bundles often include CRDs as well as a Deployment of third-party code that implements the business logic for a new custom resource.

Installing an Aggregated APIserver always involves running a new Deployment.

Storage

Custom resources consume storage space in the same way that ConfigMaps do. Creating too many custom resources may overload your API server’s storage space.

Aggregated API servers may use the same storage as the main API server, in which case the same warning applies.

Authentication, authorization, and auditing

CRDs always use the same authentication, authorization, and audit logging as the built-in resources of your API Server.

If you use RBAC for authorization, most RBAC roles will not grant access to the new resources (except the cluster-admin role or any role created with wildcard rules). You’ll need to explicitly grant access to the new resources. CRDs and Aggregated APIs often come bundled with new role definitions for the types they add.

Aggregated API servers may or may not use the same authentication, authorization, and auditing as the primary API server.

Accessing a custom resource

Kubernetes [client libraries](https://kubernetes.io/docs/reference/using-api/client-libraries/) can be used to access custom resources. Not all client libraries support custom resources. The go and python client libraries do.

When you add a custom resource, you can access it using:

* kubectl
* The kubernetes dynamic client.
* A REST client that you write.
* A client generated using [Kubernetes client generation tools](https://github.com/kubernetes/code-generator) (generating one is an advanced undertaking, but some projects may provide a client along with the CRD or AA).

What's next

* Learn how to [Extend the Kubernetes API with the aggregation layer](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/).
* Learn how to [Extend the Kubernetes API with CustomResourceDefinition](https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/).

Feedback

Operator pattern

Operators are software extensions to Kubernetes that make use of [custom resources](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) to manage applications and their components. Operators follow Kubernetes principles, notably the [control loop](https://kubernetes.io/docs/concepts/#kubernetes-control-plane).

* [**Motivation**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#motivation)
* [**Operators in Kubernetes**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#operators-in-kubernetes)
* [**An example Operator**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#example)
* [**Deploying Operators**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#deploying-operators)
* [**Using an Operator**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#using-operators)
* [**Writing your own Operator**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#writing-operator)
* [**What's next**](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#what-s-next)

Motivation

The Operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The Operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides.

Operators in Kubernetes

Kubernetes is designed for automation. Out of the box, you get lots of built-in automation from the core of Kubernetes. You can use Kubernetes to automate deploying and running workloads, *and* you can automate how Kubernetes does that.

Kubernetes’ [controllers](https://kubernetes.io/docs/concepts/architecture/controller/) concept lets you extend the cluster’s behaviour without modifying the code of Kubernetes itself. Operators are clients of the Kubernetes API that act as controllers for a [Custom Resource](https://kubernetes.io/docs/concepts/api-extension/custom-resources/).

An example Operator

Some of the things that you can use an operator to automate include:

* deploying an application on demand
* taking and restoring backups of that application’s state
* handling upgrades of the application code alongside related changes such as database schemas or extra configuration settings
* publishing a Service to applications that don’t support Kubernetes APIs to discover them
* simulating failure in all or part of your cluster to test its resilience
* choosing a leader for a distributed application without an internal member election process

What might an Operator look like in more detail? Here’s an example in more detail:

1. A custom resource named SampleDB, that you can configure into the cluster.
2. A Deployment that makes sure a Pod is running that contains the controller part of the operator.
3. A container image of the operator code.
4. Controller code that queries the control plane to find out what SampleDB resources are configured.
5. The core of the Operator is code to tell the API server how to make reality match the configured resources.
   * If you add a new SampleDB, the operator sets up PersistentVolumeClaims to provide durable database storage, a StatefulSet to run SampleDB and a Job to handle initial configuration.
   * If you delete it, the Operator takes a snapshot, then makes sure that the StatefulSet and Volumes are also removed.
6. The operator also manages regular database backups. For each SampleDB resource, the operator determines when to create a Pod that can connect to the database and take backups. These Pods would rely on a ConfigMap and / or a Secret that has database connection details and credentials.
7. Because the Operator aims to provide robust automation for the resource it manages, there would be additional supporting code. For this example, code checks to see if the database is running an old version and, if so, creates Job objects that upgrade it for you.

Deploying Operators

The most common way to deploy an Operator is to add the Custom Resource Definition and its associated Controller to your cluster. The Controller will normally run outside of the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane), much as you would run any containerized application. For example, you can run the controller in your cluster as a Deployment.

Using an Operator

Once you have an Operator deployed, you’d use it by adding, modifying or deleting the kind of resource that the Operator uses. Following the above example, you would set up a Deployment for the Operator itself, and then:

kubectl get SampleDB *# find configured databases*

kubectl edit SampleDB/example-database *# manually change some settings*

…and that’s it! The Operator will take care of applying the changes as well as keeping the existing service in good shape.

Writing your own Operator

If there isn’t an Operator in the ecosystem that implements the behavior you want, you can code your own. In [What’s next](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/#what-s-next) you’ll find a few links to libraries and tools you can use to write your own cloud native Operator.

You also implement an Operator (that is, a Controller) using any language / runtime that can act as a [client for the Kubernetes API](https://kubernetes.io/docs/reference/using-api/client-libraries/).

What's next

* Learn more about [Custom Resources](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)
* Find ready-made operators on [OperatorHub.io](https://operatorhub.io/) to suit your use case
* Use existing tools to write your own operator, eg:
  + using [KUDO](https://kudo.dev/) (Kubernetes Universal Declarative Operator)
  + using [kubebuilder](https://book.kubebuilder.io/)
  + using [Metacontroller](https://metacontroller.app/) along with WebHooks that you implement yourself
  + using the [Operator Framework](https://github.com/operator-framework/getting-started)
* [Publish](https://operatorhub.io/) your operator for other people to use
* Read [CoreOS’ original article](https://coreos.com/blog/introducing-operators.html) that introduced the Operator pattern
* Read an [article](https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-building-kubernetes-operators-and-stateful-apps) from Google Cloud about best practices for building Operators

Feedback

## 计算、存储和网络扩展

**FEATURE STATE:** Kubernetes v1.17 [alpha](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/compute-storage-net/)

**警告：** Alpha 特性迅速变化。

Kubernetes中的网络插件有几种类型：

* CNI 插件： 遵守 appc/CNI 规约，为互操作性设计。
* Kubenet 插件：使用 bridge 和 host-local CNI 插件实现了基本的 cbr0。
* [**安装**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/compute-storage-net/#%e5%ae%89%e8%a3%85)
* [**网络插件要求**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/compute-storage-net/#%e7%bd%91%e7%bb%9c%e6%8f%92%e4%bb%b6%e8%a6%81%e6%b1%82)
* [**使用总结**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/compute-storage-net/#%e4%bd%bf%e7%94%a8%e6%80%bb%e7%bb%93)

安装

kubelet 有一个单独的默认网络插件，以及一个对整个集群通用的默认网络。 它在启动时探测插件，记住找到的内容，并在 pod 生命周期的适当时间执行所选插件（这仅适用于 Docker，因为 rkt 管理自己的 CNI 插件）。 在使用插件时，需要记住两个 Kubelet 命令行参数：

* cni-bin-dir： Kubelet 在启动时探测这个目录中的插件
* network-plugin： 要使用的网络插件来自 cni-bin-dir。它必须与从插件目录探测到的插件报告的名称匹配。对于 CNI 插件，其值为 “cni”。

网络插件要求

除了提供[NetworkPlugin 接口](https://github.com/kubernetes/kubernetes/tree/v1.17.0/pkg/kubelet/dockershim/network/plugins.go)来配置和清理 pod 网络之外，该插件还可能需要对 kube-proxy 的特定支持。 iptables 代理显然依赖于 iptables，插件可能需要确保 iptables 能够监控容器的网络通信。 例如，如果插件将容器连接到 Linux 网桥，插件必须将 net/bridge/bridge-nf-call-iptables 系统参数设置为1，以确保 iptables 代理正常工作。 如果插件不使用 Linux 网桥（而是类似于 Open vSwitch 或者其它一些机制），它应该确保为代理对容器通信执行正确的路由。

默认情况下，如果未指定 kubelet 网络插件，则使用 noop 插件，该插件设置 net/bridge/bridge-nf-call-iptables=1，以确保简单的配置（如带网桥的 Docker ）与 iptables 代理正常工作。

CNI

通过给 Kubelet 传递 --network-plugin=cni 命令行选项来选择 CNI 插件。 Kubelet 从 --cni-conf-dir （默认是 /etc/cni/net.d） 读取文件并使用该文件中的 CNI 配置来设置每个 pod 的网络。 CNI 配置文件必须与 [CNI 规约](https://github.com/containernetworking/cni/blob/master/SPEC.md#network-configuration)匹配，并且配置引用的任何所需的 CNI 插件都必须存在于 --cni-bin-dir（默认是 /opt/cni/bin）。

如果这个目录中有多个 CNI 配置文件，则使用按文件名的字典顺序排列的第一个配置文件。

除了配置文件指定的 CNI 插件外，Kubernetes 还需要标准的 CNI [lo](https://github.com/containernetworking/plugins/blob/master/plugins/main/loopback/loopback.go) 插件，最低版本是0.2.0。

支持 hostPort

CNI 网络插件支持 hostPort。 您可以使用官方 [portmap](https://github.com/containernetworking/plugins/tree/master/plugins/meta/portmap) 插件，它由 CNI 插件团队提供，或者使用您自己的带有 portMapping 功能的插件。

如果你想要启动 hostPort 支持，则必须在 cni-conf-dir 指定 portMappings capability。 例如：

{

**"name"**: "k8s-pod-network",

**"cniVersion"**: "0.3.0",

**"plugins"**: [

{

**"type"**: "calico",

**"log\_level"**: "info",

**"datastore\_type"**: "kubernetes",

**"nodename"**: "127.0.0.1",

**"ipam"**: {

**"type"**: "host-local",

**"subnet"**: "usePodCidr"

},

**"policy"**: {

**"type"**: "k8s"

},

**"kubernetes"**: {

**"kubeconfig"**: "/etc/cni/net.d/calico-kubeconfig"

}

},

{

**"type"**: "portmap",

**"capabilities"**: {**"portMappings"**: **true**}

}

]

}

支持流量整形

CNI 网络插件还支持 pod 入口和出口流量整形。 您可以使用 CNI 插件团队提供的 [bandwidth](https://github.com/containernetworking/plugins/tree/master/plugins/meta/bandwidth) 插件， 也可以使用您自己的具有带宽控制功能的插件。

如果您想要启用流量整形支持，你必须将 bandwidth 插件添加到 CNI 配置文件 （默认是 /etc/cni/net.d）。

{

**"name"**: "k8s-pod-network",

**"cniVersion"**: "0.3.0",

**"plugins"**: [

{

**"type"**: "calico",

**"log\_level"**: "info",

**"datastore\_type"**: "kubernetes",

**"nodename"**: "127.0.0.1",

**"ipam"**: {

**"type"**: "host-local",

**"subnet"**: "usePodCidr"

},

**"policy"**: {

**"type"**: "k8s"

},

**"kubernetes"**: {

**"kubeconfig"**: "/etc/cni/net.d/calico-kubeconfig"

}

},

{

**"type"**: "bandwidth",

**"capabilities"**: {**"bandwidth"**: **true**}

}

]

}

现在，您可以将 kubernetes.io/ingress-bandwidth 和 kubernetes.io/egress-bandwidth 注解添加到 pod 中。 例如：

apiVersion: v1

kind: Pod

metadata:

annotations:

kubernetes.io/ingress-bandwidth: 1M

kubernetes.io/egress-bandwidth: 1M

...

kubenet

Kubenet 是一个非常基本的、简单的网络插件，仅适用于 Linux。 它本身并不实现更高级的功能，如跨节点网络或网络策略。 它通常与云驱动一起使用，云驱动为节点间或单节点环境中的通信设置路由规则。

Kubenet 创建名为 cbr0 的网桥，并为每个 pod 创建了一个 veth 对，每个 pod 的主机端都连接到 cbr0。 这个 veth 对的 pod 端会被分配一个 IP 地址，该 IP 地址隶属于节点所被分配的 IP 地址范围内。节点的 IP 地址范围则通过配置或控制器管理器来设置。 cbr0 被分配一个 MTU，该 MTU 匹配主机上已启用的正常接口的最小 MTU。

使用此插件还需要一些其他条件：

* 需要标准的 CNI bridge、lo 以及 host-local 插件，最低版本是0.2.0。Kubenet 首先在 /opt/cni/bin 中搜索它们。 指定 cni-bin-dir 以提供其它的搜索路径。首次找到的匹配将生效。
* Kubelet 必须和 --network-plugin=kubenet 参数一起运行，才能启用该插件。
* Kubelet 还应该和 --non-masquerade-cidr=<clusterCidr> 参数一起运行，以确保超出此范围的 IP 流量将使用 IP 伪装。
* 节点必须被分配一个 IP 子网，通过kubelet 命令行的 --pod-cidr 选项或控制器管理器的命令行选项 --allocate-node-cidrs=true --cluster-cidr=<cidr> 来设置。

自定义 MTU（使用 kubenet）

要获得最佳的网络性能，必须确保 MTU 的取值配置正确。 网络插件通常会尝试推断出一个合理的 MTU，但有时候这个逻辑不会产生一个最优的 MTU。 例如，如果 Docker 网桥或其他接口有一个小的 MTU, kubenet 当前将选择该 MTU。 或者如果您正在使用 IPSEC 封装，则必须减少 MTU，并且这种计算超出了大多数网络插件的能力范围。

如果需要，您可以使用 network-plugin-mtu kubelet 选项显式的指定 MTU。 例如：在 AWS 上 eth0 MTU 通常是 9001，因此您可以指定 --network-plugin-mtu=9001。 如果您正在使用 IPSEC ，您可以减少它以允许封装开销，例如 --network-plugin-mtu=8873。

此选项会传递给网络插件； 当前 **仅 kubenet 支持 network-plugin-mtu**。

使用总结

* --network-plugin=cni 用来表明我们要使用 cni 网络插件，实际的 CNI 插件可执行文件位于 --cni-bin-dir（默认是 /opt/cni/bin）下， CNI 插件配置位于 --cni-conf-dir（默认是 /etc/cni/net.d）下。
* --network-plugin=kubenet 用来表明我们要使用 kubenet 网络插件，CNI bridge 和 host-local 插件位于 /opt/cni/bin 或 cni-bin-dir 中。
* --network-plugin-mtu=9001 指定了我们使用的 MTU，当前仅被 kubenet 网络插件使用。

反馈

Device Plugins

**FEATURE STATE:** Kubernetes v1.10 [beta](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/)

Kubernetes provides a [device plugin framework](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md) that you can use to advertise system hardware resources to the [Kubelet](https://kubernetes.io/docs/reference/generated/kubelet).

Instead of customizing the code for Kubernetes itself, vendors can implement a device plugin that you deploy either manually or as a [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset). The targeted devices include GPUs, high-performance NICs, FPGAs, InfiniBand adapters, and other similar computing resources that may require vendor specific initialization and setup.

* [**Device plugin registration**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#device-plugin-registration)
* [**Device plugin implementation**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#device-plugin-implementation)
* [**Device plugin deployment**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#device-plugin-deployment)
* [**API compatibility**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#api-compatibility)
* [**Monitoring Device Plugin Resources**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#monitoring-device-plugin-resources)
* [**Device Plugin integration with the Topology Manager**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#device-plugin-integration-with-the-topology-manager)
* [**Device plugin examples**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#examples)
* [**What's next**](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/#what-s-next)

Device plugin registration

The kubelet exports a Registration gRPC service:

service Registration {

rpc Register(RegisterRequest) returns (Empty) {}

}

A device plugin can register itself with the kubelet through this gRPC service. During the registration, the device plugin needs to send:

* The name of its Unix socket.
* The Device Plugin API version against which it was built.
* The ResourceName it wants to advertise. Here ResourceName needs to follow the [extended resource naming scheme](https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#extended-resources) as vendor-domain/resourcetype. (For example, an NVIDIA GPU is advertised as nvidia.com/gpu.)

Following a successful registration, the device plugin sends the kubelet the list of devices it manages, and the kubelet is then in charge of advertising those resources to the API server as part of the kubelet node status update. For example, after a device plugin registers hardware-vendor.example/foo with the kubelet and reports two healthy devices on a node, the node status is updated to advertise that the node has 2 “Foo” devices installed and available.

Then, users can request devices in a [Container](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#container-v1-core) specification as they request other types of resources, with the following limitations:

* Extended resources are only supported as integer resources and cannot be overcommitted.
* Devices cannot be shared among Containers.

Suppose a Kubernetes cluster is running a device plugin that advertises resource hardware-vendor.example/foo on certain nodes. Here is an example of a pod requesting this resource to run a demo workload:

---

apiVersion: v1

kind: Pod

metadata:

name: demo-pod

spec:

containers:

- name: demo-container-1

image: k8s.gcr.io/pause:2.0

resources:

limits:

hardware-vendor.example/foo: 2

*#*

*# This Pod needs 2 of the hardware-vendor.example/foo devices*

*# and can only schedule onto a Node that's able to satisfy*

*# that need.*

*#*

*# If the Node has more than 2 of those devices available, the*

*# remainder would be available for other Pods to use.*

Device plugin implementation

The general workflow of a device plugin includes the following steps:

* Initialization. During this phase, the device plugin performs vendor specific initialization and setup to make sure the devices are in a ready state.
* The plugin starts a gRPC service, with a Unix socket under host path /var/lib/kubelet/device-plugins/, that implements the following interfaces:
* service DevicePlugin {
* // ListAndWatch returns a stream of List of Devices
* // Whenever a Device state change or a Device disappears, ListAndWatch
* // returns the new list
* rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}
* // Allocate is called during container creation so that the Device
* // Plugin can run device specific operations and instruct Kubelet
* // of the steps to make the Device available in the container
* rpc Allocate(AllocateRequest) returns (AllocateResponse) {}

}

* The plugin registers itself with the kubelet through the Unix socket at host path /var/lib/kubelet/device-plugins/kubelet.sock.
* After successfully registering itself, the device plugin runs in serving mode, during which it keeps monitoring device health and reports back to the kubelet upon any device state changes. It is also responsible for serving Allocate gRPC requests. During Allocate, the device plugin may do device-specific preparation; for example, GPU cleanup or QRNG initialization. If the operations succeed, the device plugin returns an AllocateResponse that contains container runtime configurations for accessing the allocated devices. The kubelet passes this information to the container runtime.

Handling kubelet restarts

A device plugin is expected to detect kubelet restarts and re-register itself with the new kubelet instance. In the current implementation, a new kubelet instance deletes all the existing Unix sockets under /var/lib/kubelet/device-plugins when it starts. A device plugin can monitor the deletion of its Unix socket and re-register itself upon such an event.

Device plugin deployment

You can deploy a device plugin as a DaemonSet, as a package for your node’s operating system, or manually.

The canonical directory /var/lib/kubelet/device-plugins requires privileged access, so a device plugin must run in a privileged security context. If you’re deploying a device plugin as a DaemonSet, /var/lib/kubelet/device-plugins must be mounted as a [Volume](https://kubernetes.io/docs/concepts/storage/volumes/) in the plugin’s [PodSpec](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#podspec-v1-core).

If you choose the DaemonSet approach you can rely on Kubernetes to: place the device plugin’s Pod onto Nodes, to restart the daemon Pod after failure, and to help automate upgrades.

API compatibility

Kubernetes device plugin support is in beta. The API may change before stabilization, in incompatible ways. As a project, Kubernetes recommends that device plugin developers:

* Watch for changes in future releases.
* Support multiple versions of the device plugin API for backward/forward compatibility.

If you enable the DevicePlugins feature and run device plugins on nodes that need to be upgraded to a Kubernetes release with a newer device plugin API version, upgrade your device plugins to support both versions before upgrading these nodes. Taking that approach will ensure the continuous functioning of the device allocations during the upgrade.

Monitoring Device Plugin Resources

**FEATURE STATE:** Kubernetes v1.15 [beta](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/)

In order to monitor resources provided by device plugins, monitoring agents need to be able to discover the set of devices that are in-use on the node and obtain metadata to describe which container the metric should be associated with. [Prometheus](https://prometheus.io/) metrics exposed by device monitoring agents should follow the [Kubernetes Instrumentation Guidelines](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-instrumentation/instrumentation.md), identifying containers using pod, namespace, and container prometheus labels.

The kubelet provides a gRPC service to enable discovery of in-use devices, and to provide metadata for these devices:

// PodResourcesLister is a service provided by the kubelet that provides information about the

// node resources consumed by pods and containers on the node

service PodResourcesLister {

rpc List(ListPodResourcesRequest) returns (ListPodResourcesResponse) {}

}

The gRPC service is served over a unix socket at /var/lib/kubelet/pod-resources/kubelet.sock. Monitoring agents for device plugin resources can be deployed as a daemon, or as a DaemonSet. The canonical directory /var/lib/kubelet/pod-resources requires privileged access, so monitoring agents must run in a privileged security context. If a device monitoring agent is running as a DaemonSet, /var/lib/kubelet/pod-resources must be mounted as a [Volume](https://kubernetes.io/docs/concepts/storage/volumes/) in the plugin’s [PodSpec](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#podspec-v1-core).

Support for the “PodResources service” requires KubeletPodResources [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to be enabled. It is enabled by default starting with Kubernetes 1.15.

Device Plugin integration with the Topology Manager

**FEATURE STATE:** Kubernetes v1.17 [alpha](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/)

The Topology Manager is a Kubelet component that allows resources to be co-ordintated in a Topology aligned manner. In order to do this, the Device Plugin API was extended to include a TopologyInfo struct.

message TopologyInfo {

repeated NUMANode nodes = 1;

}

message NUMANode {

int64 ID = 1;

}

Device Plugins that wish to leverage the Topology Manager can send back a populated TopologyInfo struct as part of the device registration, along with the device IDs and the health of the device. The device manager will then use this information to consult with the Topology Manager and make resource assignment decisions.

TopologyInfo supports a nodes field that is either nil (the default) or a list of NUMA nodes. This lets the Device Plugin publish that can span NUMA nodes.

An example TopologyInfo struct populated for a device by a Device Plugin:

pluginapi.Device{ID: "25102017", Health: pluginapi.Healthy, Topology:&pluginapi.TopologyInfo{Nodes: []\*pluginapi.NUMANode{&pluginapi.NUMANode{ID: 0,},}}}

Device plugin examples

Here are some examples of device plugin implementations:

* The [AMD GPU device plugin](https://github.com/RadeonOpenCompute/k8s-device-plugin)
* The [Intel device plugins](https://github.com/intel/intel-device-plugins-for-kubernetes) for Intel GPU, FPGA and QuickAssist devices
* The [KubeVirt device plugins](https://github.com/kubevirt/kubernetes-device-plugins) for hardware-assisted virtualization
* The [NVIDIA GPU device plugin](https://github.com/NVIDIA/k8s-device-plugin)
  + Requires [nvidia-docker](https://github.com/NVIDIA/nvidia-docker) 2.0, which allows you to run GPU-enabled Docker containers.
* The [NVIDIA GPU device plugin for Container-Optimized OS](https://github.com/GoogleCloudPlatform/container-engine-accelerators/tree/master/cmd/nvidia_gpu)
* The [RDMA device plugin](https://github.com/hustcat/k8s-rdma-device-plugin)
* The [Solarflare device plugin](https://github.com/vikaschoudhary16/sfc-device-plugin)
* The [SR-IOV Network device plugin](https://github.com/intel/sriov-network-device-plugin)
* The [Xilinx FPGA device plugins](https://github.com/Xilinx/FPGA_as_a_Service/tree/master/k8s-fpga-device-plugin/trunk) for Xilinx FPGA devices

What's next

* Learn about [scheduling GPU resources](https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/) using device plugins
* Learn about [advertising extended resources](https://kubernetes.io/docs/tasks/administer-cluster/extended-resource-node/) on a node
* Read about using [hardware acceleration for TLS ingress](https://kubernetes.io/blog/2019/04/24/hardware-accelerated-ssl/tls-termination-in-ingress-controllers-using-kubernetes-device-plugins-and-runtimeclass/) with Kubernetes
* Learn about the [Topology Manager](https://kubernetes.io/docs/tasks/adminster-cluster/topology-manager/)

服务目录

服务目录（Service Catalog）是一种扩展 API，它能让 Kubernetes 集群中运行的应用易于使用外部托管的的软件服务，例如云供应商提供的数据仓库服务。

服务目录可以检索、供应、和绑定由 [服务代理人（Service Brokers）](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-service-broker) 提供的外部 [托管服务](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-managed-service)，而无需知道那些服务具体是怎样创建和托管的。

服务代理是由[开放服务代理 API 规范](https://github.com/openservicebrokerapi/servicebroker/blob/v2.13/spec.md)定义的一组托管服务的终结点，由第三方提供并维护，其中的第三方可以是 AWS，GCP 或 Azure 等云服务提供商。 托管服务的一些示例是 Microsoft Azure Cloud Queue，Amazon Simple Queue Service 和 Google Cloud Pub/Sub，但它们是可以使用应用程序的任何软件产品。

使用服务目录，集群操作者可以浏览其提供的托管服务列表，提供托管服务实例并与之绑定，以使其可以被 Kubernetes 集群中的应用程序使用。

* [**示例用例**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/service-catalog/#%e7%a4%ba%e4%be%8b%e7%94%a8%e4%be%8b)
* [**架构**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/service-catalog/#%e6%9e%b6%e6%9e%84)
* [**API 资源**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/service-catalog/#api-%e8%b5%84%e6%ba%90)
* [**使用方式**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/service-catalog/#%e4%bd%bf%e7%94%a8%e6%96%b9%e5%bc%8f)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/extend-kubernetes/service-catalog/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

示例用例

应用开发者希望使用消息队列作为其在 Kubernetes 集群中运行的应用程序的一部分。 但是，它们不想承受建立这种服务的开销，也不想自行管理。幸运的是，有一家云服务提供商通过它们的服务代理将消息队列作为托管服务提供。

集群运维人员可以设置服务目录并使用它与云服务提供商的服务代理 通信，以此提供消息队列服务的实例并使其对 Kubernetes 中的应用程序可用。 因此，应用开发者可以不用关心消息队列的实现细节，也不用对其进行管理。它们的应用程序可以简单的将其作为服务使用。

架构

服务目录使用[开放服务代理 API](https://github.com/openservicebrokerapi/servicebroker) 与服务代理进行通信，并作为 Kubernetes API Server 的中介，以便协商首要规定并获取应用程序使用托管服务的必要凭据。

它被实现为一个扩展 API 服务和一个控制器管理器，使用 Etcd 作为存储。它还使用了 Kubernetes 1.7+ 版本中提供的 [aggregation layer](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/) 来呈现其 API。

API 资源

服务目录安装 servicecatalog.k8s.io API 并提供以下 Kubernetes 资源：

* ClusterServiceBroker：服务目录的集群内代表，封装了它的服务连接细节。集群运维人员创建和管理这些资源，并希望使用该代理服务在集群中提供新类型的托管服务。
* ClusterServiceClass：由特定服务代理提供的托管服务。当新的 ClusterServiceBroker 资源被添加到集群时，服务目录控制器将连接到服务代理以获取可用的托管服务列表。然后为每个托管服务创建对应的新 ClusterServiceClass 资源。
* ClusterServicePlan：托管服务的特定产品。例如托管服务可能有不同的计划可用，如免费版本和付费版本，或者可能有不同的配置选项，例如使用 SSD 存储或拥有更多资源。与 ClusterServiceClass 类似，当一个新的 ClusterServiceBroker 被添加到集群时，服务目录会为每个托管服务的每个可用服务计划创建对应的新 ClusterServicePlan 资源。
* ServiceInstance：ClusterServiceClass 提供的示例。由集群运维人员创建，以使托管服务的特定实例可供一个或多个集群内应用程序使用。当创建一个新的 ServiceInstance 资源时，服务目录控制器将连接到相应的服务代理并指示它调配服务实例。
* ServiceBinding：ServiceInstance 的访问凭据。由希望其应用程序使用服务 ServiceInstance 的集群运维人员创建。创建之后，服务目录控制器将创建一个 Kubernetes Secret，其中包含服务实例的连接细节和凭据，可以挂载到 Pod 中。

认证

服务目录支持这些认证方法：

* 基础认证（用户名/密码）
* [OAuth 2.0 不记名令牌](https://tools.ietf.org/html/rfc6750)

使用方式

集群运维人员可以使用服务目录 API 资源来提供托管服务并使其在 Kubernetes 集群内可用。涉及的步骤有：

1. 列出服务代理提供的托管服务和服务计划。
2. 配置托管服务的新实例。
3. 绑定到托管服务，它将返回连接凭证。
4. 将连接凭证映射到应用程序中。

列出托管服务和服务计划

首先，集群运维人员在 servicecatalog.k8s.io 组内创建一个 ClusterServiceBroker 资源。此资源包含访问服务代理终结点所需的 URL 和连接详细信息。

这是一个 ClusterServiceBroker 资源的例子：

apiVersion: servicecatalog.k8s.io/v1beta1

kind: ClusterServiceBroker

metadata:

name: cloud-broker

spec:

*# Points to the endpoint of a service broker. (This example is not a working URL.)*

url: https://servicebroker.somecloudprovider.com/v1alpha1/projects/service-catalog/brokers/default

*#####*

*# Additional values can be added here, which may be used to communicate*

*# with the service broker, such as bearer token info or a caBundle for TLS.*

*#####*

下面的顺序图展示了从一个服务代理列出可用托管服务和计划所有涉及的步骤：

1. 一旦 ClusterServiceBroker 资源被添加到了服务目录之后，将会触发一个到外部服务代理的 List Services 调用。
2. 服务代理返回可用的托管服务和服务计划列表，这些列表将本地缓存在 ClusterServiceClass 和 ClusterServicePlan 资源中。
3. 然后集群运维人员可以使用以下命令获取可用托管服务的列表：
4. kubectl get clusterserviceclasses -o=custom-columns=SERVICE\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName

它应该输出一个和以下格式类似的服务名称列表：

SERVICE NAME EXTERNAL NAME

4f6e6cf6-ffdd-425f-a2c7-3c9258ad2468 cloud-provider-service

... ...

他们还可以使用以下命令查看可用的服务计划：

kubectl get clusterserviceplans -o=custom-columns=PLAN\ NAME:.metadata.name,EXTERNAL\ NAME:.spec.externalName

它应该输出一个和以下格式类似的服务计划列表：

PLAN NAME EXTERNAL NAME

86064792-7ea2-467b-af93-ac9694d96d52 service-plan-name

... ...

配置一个新实例

集群运维人员 可以通过创建一个 ServiceInstance 资源来启动一个新实例的配置。

这是一个 ServiceInstance 资源的例子：

apiVersion: servicecatalog.k8s.io/v1beta1

kind: ServiceInstance

metadata:

name: cloud-queue-instance

namespace: cloud-apps

spec:

*# References one of the previously returned services*

clusterServiceClassExternalName: cloud-provider-service

clusterServicePlanExternalName: service-plan-name

*#####*

*# Additional parameters can be added here,*

*# which may be used by the service broker.*

*#####*

以下顺序图展示了配置托管服务新实例所涉及的步骤：

1. 当创建 ServiceInstance 资源时，服务目录将启动一个到外部服务代理的配置实例调用。
2. 服务代理创建一个托管服务的新实例并返回 HTTP 响应。
3. 然后集群运维人员可以检查实例的状态是否就绪。

绑定到托管服务

在设置新实例之后，集群运维人员必须绑定到托管服务才能获取应用程序使用服务所需的连接凭据和服务账户的详细信息。该操作通过创建一个 ServiceBinding 资源完成。

以下是 ServiceBinding 资源的示例：

apiVersion: servicecatalog.k8s.io/v1beta1

kind: ServiceBinding

metadata:

name: cloud-queue-binding

namespace: cloud-apps

spec:

instanceRef:

name: cloud-queue-instance

*#####*

*# Additional information can be added here, such as a secretName or*

*# service account parameters, which may be used by the service broker.*

*#####*

以下顺序图展示了绑定到托管服务实例的步骤：

1. 在创建 ServiceBinding 之后，服务目录调用外部服务代理，请求绑定服务实例所需的信息。
2. 服务代理为相应服务账户启用应用权限/角色。
3. 服务代理返回连接和访问托管服务示例所需的信息。这是由提供商和服务特定的，故返回的信息可能因服务提供商和其托管服务而有所不同。

映射连接凭据

完成绑定之后的最后一步就是将连接凭据和服务特定的信息映射到应用程序中。这些信息存储在 secret 中，集群中的应用程序可以访问并使用它们直接与托管服务进行连接。

Pod 配置文件

执行此映射的一种方法是使用声明式 Pod 配置。

以下示例描述了如何将服务账户凭据映射到应用程序中。名为 sa-key 的密钥保存在一个名为 provider-cloud-key 的卷中，应用程序会将该卷挂载在 /var/secrets/provider/key.json 路径下。环境变量 PROVIDER\_APPLICATION\_CREDENTIALS 将映射为挂载文件的路径。

...

spec:

volumes:

- name: provider-cloud-key

secret:

secretName: sa-key

containers:

...

volumeMounts:

- name: provider-cloud-key

mountPath: /var/secrets/provider

env:

- name: PROVIDER\_APPLICATION\_CREDENTIALS

value: "/var/secrets/provider/key.json"

以下示例描述了如何将 secret 值映射为应用程序的环境变量。在这个示例中，消息队列的主题名从 secret provider-queue-credentials 中名为 topic 的 key 项映射到环境变量 TOPIC 中。

...

env:

- name: "TOPIC"

valueFrom:

secretKeyRef:

name: provider-queue-credentials

key: topic

Poseidon-Firmament - An alternate scheduler

**Current release of Poseidon-Firmament scheduler is an alpha release.**

Poseidon-Firmament scheduler is an alternate scheduler that can be deployed alongside the default Kubernetes scheduler.

* [**Introduction**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#introduction)
* [**Key Advantages**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#key-advantages)
* [**Poseidon-Firmament Scheduler - How it works**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#poseidon-firmament-scheduler-how-it-works)
* [**Possible Use Case Scenarios - When to use it**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#possible-use-case-scenarios-when-to-use-it)
* [**Current Project Stage**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#current-project-stage)
* [**Features Comparison Matrix**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#features-comparison-matrix)
* [**Installation**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#installation)
* [**Development**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#development)
* [**Latest Throughput Performance Testing Results**](https://kubernetes.io/docs/concepts/extend-kubernetes/poseidon-firmament-alternate-scheduler/#latest-throughput-performance-testing-results)

Introduction

Poseidon is a service that acts as the integration glue for the [Firmament scheduler](https://github.com/Huawei-PaaS/firmament) with Kubernetes. Poseidon-Firmament scheduler augments the current Kubernetes scheduling capabilities. It incorporates novel flow network graph based scheduling capabilities alongside the default Kubernetes Scheduler. Firmament scheduler models workloads and clusters as flow networks and runs min-cost flow optimizations over these networks to make scheduling decisions.

It models the scheduling problem as a constraint-based optimization over a flow network graph. This is achieved by reducing scheduling to a min-cost max-flow optimization problem. The Poseidon-Firmament scheduler dynamically refines the workload placements.

Poseidon-Firmament scheduler runs alongside the default Kubernetes Scheduler as an alternate scheduler, so multiple schedulers run simultaneously.

Key Advantages

Flow graph scheduling based Poseidon-Firmament scheduler provides the following key advantages:

* Workloads (pods) are bulk scheduled to enable scheduling at massive scale..
* Based on the extensive performance test results, Poseidon-Firmament scales much better than the Kubernetes default scheduler as the number of nodes increase in a cluster. This is due to the fact that Poseidon-Firmament is able to amortize more and more work across workloads.
* Poseidon-Firmament Scheduler outperforms the Kubernetes default scheduler by a wide margin when it comes to throughput performance numbers for scenarios where compute resource requirements are somewhat uniform across jobs (Replicasets/Deployments/Jobs). Poseidon-Firmament scheduler end-to-end throughput performance numbers, including bind time, consistently get better as the number of nodes in a cluster increase. For example, for a 2,700 node cluster (shown in the graphs [here](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/benchmark/README.md)), Poseidon-Firmament scheduler achieves a 7X or greater end-to-end throughput than the Kubernetes default scheduler, which includes bind time.
* Availability of complex rule constraints.
* Scheduling in Poseidon-Firmament is dynamic; it keeps cluster resources in a global optimal state during every scheduling run.
* Highly efficient resource utilizations.

Poseidon-Firmament Scheduler - How it works

As part of the Kubernetes multiple schedulers support, each new pod is typically scheduled by the default scheduler. Kubernetes can be instructed to use another scheduler by specifying the name of another custom scheduler (“poseidon” in our case) in the **schedulerName** field of the PodSpec at the time of pod creation. In this case, the default scheduler will ignore that Pod and allow Poseidon scheduler to schedule the Pod on a relevant node.

apiVersion: v1

kind: Pod

...

spec:

schedulerName: poseidon

**Note:** For details about the design of this project see the [design document](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/design/README.md).

Possible Use Case Scenarios - When to use it

As mentioned earlier, Poseidon-Firmament scheduler enables an extremely high throughput scheduling environment at scale due to its bulk scheduling approach versus Kubernetes pod-at-a-time approach. In our extensive tests, we have observed substantial throughput benefits as long as resource requirements (CPU/Memory) for incoming Pods are uniform across jobs (Replicasets/Deployments/Jobs), mainly due to efficient amortization of work across jobs.

Although, Poseidon-Firmament scheduler is capable of scheduling various types of workloads, such as service, batch, etc., the following are a few use cases where it excels the most:

1. For “Big Data/AI” jobs consisting of large number of tasks, throughput benefits are tremendous.
2. Service or batch jobs where workload resource requirements are uniform across jobs (Replicasets/Deployments/Jobs).

Current Project Stage

* **Alpha Release - Incubation repo.** at <https://github.com/kubernetes-sigs/poseidon>.
* Currently, Poseidon-Firmament scheduler **does not provide support for high availability**, our implementation assumes that the scheduler cannot fail. The [design document](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/design/README.md) describes possible ways to enable high availability, but we leave this to future work.
* We are **not aware of any production deployment** of Poseidon-Firmament scheduler at this time.
* Poseidon-Firmament is supported from Kubernetes release 1.6 and works with all subsequent releases.
* Release process for Poseidon and Firmament repos are in lock step. The current Poseidon release can be found [here](https://github.com/kubernetes-sigs/poseidon/releases) and the corresponding Firmament release can be found [here](https://github.com/Huawei-PaaS/firmament/releases).

Features Comparison Matrix

| Feature | Kubernetes Default Scheduler | Poseidon-Firmament Scheduler | Notes |
| --- | --- | --- | --- |
| Node Affinity/Anti-Affinity | Y | Y |  |
| Pod Affinity/Anti-Affinity - including support for pod anti-affinity symmetry | Y | Y | Currently, the default scheduler outperforms the Poseidon-Firmament scheduler pod affinity/anti-affinity functionality. We are working towards resolving this. |
| Taints & Tolerations | Y | Y |  |
| Baseline Scheduling capability in accordance to available compute resources (CPU & Memory) on a node | Y | Y\*\* | Not all Predicates & Priorities are supported at this time. |
| Extreme Throughput at scale | Y\*\* | Y | Bulk scheduling approach scales or increases workload placement. Substantial throughput benefits using Firmament scheduler as long as resource requirements (CPU/Memory) for incoming Pods is uniform across Replicasets/Deployments/Jobs. This is mainly due to efficient amortization of work across Replicasets/Deployments/Jobs . 1) For “Big Data/AI” jobs consisting of large no. of tasks, throughput benefits are tremendous. 2) Substantial throughput benefits also for service or batch job scenarios where workload resource requirements are uniform across Replicasets/Deployments/Jobs. |
| Optimal Scheduling | Pod-by-Pod scheduler, processes one pod at a time (may result into sub-optimal scheduling) | Bulk Scheduling (Optimal scheduling) | Pod-by-Pod Kubernetes default scheduler may assign tasks to a sub-optimal machine. By contrast, Firmament considers all unscheduled tasks at the same time together with their soft and hard constraints. |
| Colocation Interference Avoidance | N | N\*\* | Planned in Poseidon-Firmament. |
| Priority Pre-emption | Y | N\*\* | Partially exists in Poseidon-Firmament versus extensive support in Kubernetes default scheduler. |
| Inherent Re-Scheduling | N | Y\*\* | Poseidon-Firmament scheduler supports workload re-scheduling. In each scheduling run it considers all the pods, including running pods, and as a result can migrate or evict pods – a globally optimal scheduling environment. |
| Gang Scheduling | N | Y |  |
| Support for Pre-bound Persistence Volume Scheduling | Y | Y |  |
| Support for Local Volume & Dynamic Persistence Volume Binding Scheduling | Y | N\*\* | Planned. |
| High Availability | Y | N\*\* | Planned. |
| Real-time metrics based scheduling | N | Y\*\* | Initially supported using Heapster (now deprecated) for placing pods using actual cluster utilization statistics rather than reservations. Plans to switch over to “metric server”. |
| Support for Max-Pod per node | Y | Y | Poseidon-Firmament scheduler seamlessly co-exists with Kubernetes default scheduler. |
| Support for Ephemeral Storage, in addition to CPU/Memory | Y | Y |  |

Installation

For in-cluster installation of Poseidon, please start at the [Installation instructions](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/install/README.md).

Development

For developers, please refer to the [Developer Setup instructions](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/devel/README.md).

Latest Throughput Performance Testing Results

Pod-by-pod schedulers, such as the Kubernetes default scheduler, typically process one pod at a time. These schedulers have the following crucial drawbacks:

1. The scheduler commits to a pod placement early and restricts the choices for other pods that wait to be placed.
2. There is limited opportunities for amortizing work across pods because they are considered for placement individually.

These downsides of pod-by-pod schedulers are addressed by batching or bulk scheduling in Poseidon-Firmament scheduler. Processing several pods in a batch allows the scheduler to jointly consider their placement, and thus to find the best trade-off for the whole batch instead of one pod. At the same time it amortizes work across pods resulting in much higher throughput.

**Note:** Please refer to the [latest benchmark results](https://github.com/kubernetes-sigs/poseidon/blob/master/docs/benchmark/README.md) for detailed throughput performance comparison test results between Poseidon-Firmament scheduler and the Kubernetes default scheduler.

容器

镜像

创建 Docker 镜像并将其推送到仓库，然后在 Kubernetes pod 中引用它。

容器的 image 属性支持与 docker 命令相同的语法，包括私有仓库和标签。

* [**升级镜像**](https://kubernetes.io/zh/docs/concepts/containers/#%e5%8d%87%e7%ba%a7%e9%95%9c%e5%83%8f)
* [**使用清单（manifest）构建多架构镜像**](https://kubernetes.io/zh/docs/concepts/containers/#%e4%bd%bf%e7%94%a8%e6%b8%85%e5%8d%95-manifest-%e6%9e%84%e5%bb%ba%e5%a4%9a%e6%9e%b6%e6%9e%84%e9%95%9c%e5%83%8f)
* [**使用私有仓库**](https://kubernetes.io/zh/docs/concepts/containers/#%e4%bd%bf%e7%94%a8%e7%a7%81%e6%9c%89%e4%bb%93%e5%ba%93)

升级镜像

默认的镜像拉取策略是 IfNotPresent，在镜像已经存在的情况下，kubelet 将不再去拉取镜像。如果总是想要拉取镜像，您可以执行以下操作：

* 设置容器的 imagePullPolicy 为 Always。
* 省略 imagePullPolicy，并使用 :latest 作为要使用的镜像的标签。
* 省略 imagePullPolicy 和要使用的镜像标签。
* 启用 [AlwaysPullImages](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#alwayspullimages) 准入控制器（admission controller）。

注意应避免使用 :latest 标签，参见[配置镜像最佳实践](https://kubernetes.io/docs/concepts/configuration/overview/#container-images) 获取更多信息。

使用清单（manifest）构建多架构镜像

Docker CLI 现在支持以下命令 docker manifest 以及 create、annotate、push 等子命令。这些命令可用于构建和推送清单。您可以使用 docker manifest inspect 来查看清单。

请在此处查看 docker 清单文档： <https://docs.docker.com/edge/engine/reference/commandline/manifest/>

查看有关如何在构建工具中使用清单的示例： <https://cs.k8s.io/?q=docker%20manifest%20(create%7Cpush%7Cannotate)&i=nope&files=&repos=>

这些命令依赖于 Docker CLI 并仅在 Docker CLI 上实现。需要编辑 $HOME/.docker/config.json 并将 experimental 设置为 enabled，或者仅在调用 CLI 命令时将 DOCKER\_CLI\_EXPERIMENTAL 环境变量设置为 enabled。

**注意：**

请使用 Docker \*18.06 或更高版本\*，低版本存在错误或不支持实验性命令行选项。导致容器问题示例 [https://github.com/docker/cli/issues/1135。](https://github.com/docker/cli/issues/1135%E3%80%82)

如果在上传旧清单时遇到麻烦，只需删除 $HOME/.docker/manifests 中旧的清单即可重新开始。

对于 Kubernetes，通常使用带有后缀 -$(ARCH) 的镜像。为了向后兼容，请生成带有后缀的旧镜像。想法是生成具有所有 arch(es) 清单的 pause 镜像，并生成 pause-amd64 镜像，该镜像向后兼容较早的配置或者可能已对带有后缀的镜像进行硬编码的 YAML 文件。

使用私有仓库

从私有仓库读取镜像时可能需要密钥。 凭证可以用以下方式提供:

- 使用 Google Container Registry - 每个集群 - 在 Google Compute Engine 或 Google Kubernetes Engine 上自动配置 - 所有 Pod 均可读取项目的私有仓库 - 使用 Amazon Elastic Container Registry（ECR） - 使用 IAM 角色和策略来控制对 ECR 仓库的访问 - 自动刷新 ECR 登录凭据 - 使用 Oracle Cloud Infrastructure Registry（OCIR） - 使用 IAM 角色和策略来控制对 OCIR 仓库的访问 - 使用 Azure Container Registry (ACR) - 使用 IBM Cloud Container Registry - 配置节点用于私有仓库进行身份验证 - 所有 Pod 均可读取任何已配置的私有仓库 - 需要集群管理员配置节点 - 预拉镜像 - 所有 Pod 都可以使用节点上缓存的任何镜像 - 需要所有节点的 root 访问权限才能进行设置 - 在 Pod 上指定 ImagePullSecrets - 只有提供自己密钥的 Pod 才能访问私有仓库

下面将详细描述每一项。

使用 Google Container Registry

Kuberetes 运行在 Google Compute Engine (GCE) 时原生支持 [Google Container Registry (GCR)](https://cloud.google.com/tools/container-registry/)。如果 kubernetes 集群运行在 GCE 或者 Google Kubernetes Engine，使用镜像全名(例如 gcr.io/my\_project/image:tag) 即可。

集群中所有 pod 都会有读取这个仓库镜像的权限。

kubelet 将使用实例的 Google service account 向 GCR 认证。实例的 Google service account 拥有 https://www.googleapis.com/auth/devstorage.read\_only，所以它可以从项目的 GCR 拉取，但不能推送。

使用 Amazon Elastic Container Registry

当 Node 是 AWS EC2 实例时，Kubernetes 原生支持 [Amazon Elastic Container Registry](https://aws.amazon.com/ecr/)。

在 pod 定义中，使用镜像全名即可 (例如 ACCOUNT.dkr.ecr.REGION.amazonaws.com/imagename:tag)

集群中所有可以创建 Pod 的用户都将能够运行使用 ECR 仓库中任何镜像的 Pod。

kubelet 将获取并定期刷新 ECR 凭据。它需要以下权限才能执行此操作：

* ecr:GetAuthorizationToken
* ecr:BatchCheckLayerAvailability
* ecr:GetDownloadUrlForLayer
* ecr:GetRepositoryPolicy
* ecr:DescribeRepositories
* ecr:ListImages
* ecr:BatchGetImage

要求：

* 必须使用 kubelet v1.2.0 及以上版本。（例如 运行 /usr/bin/kubelet --version=true）。
* 如果 Node 在区域 A，而镜像仓库在另一个区域 B，需要 v1.3.0 及以上版本。
* 区域中必须提供 ECR。

故障排除：

* 验证是否满足以上要求。
* 获取工作站的 $REGION (例如 us-west-2) 凭证，使用凭证 SSH 到主机手动运行 Docker。它行得通吗？
* 验证 kubelet 是否使用参数 --cloud-provider=aws 运行。
* 检查 kubelet 日志(例如 journalctl -u kubelet)是否有类似的行：
  + plugins.go:56] Registering credential provider: aws-ecr-key
  + provider.go:91] Refreshing cache for provider: \*aws\_credentials.ecrProvider

使用 Azure Container Registry (ACR)

当使用 [Azure Container Registry](https://azure.microsoft.com/en-us/services/container-registry/) 时，可以使用管理员用户或者 service principal 进行身份验证。任何一种情况，认证都通过标准的 Docker 授权完成。本指南假设使用 [azure-cli](https://github.com/azure/azure-cli) 命令行工具。

首先，需要创建仓库并获取凭证，完整的文档请参考 [Azure container registry 文档](https://docs.microsoft.com/en-us/azure/container-registry/container-registry-get-started-azure-cli)。

创建好容器仓库后，可以使用以下凭证登录：

* DOCKER\_USER : service principal，或管理员用户名称
* DOCKER\_PASSWORD: service principal 密码，或管理员用户密码
* DOCKER\_REGISTRY\_SERVER: ${some-registry-name}.azurecr.io
* DOCKER\_EMAIL: ${some-email-address}

填写以上变量后，就可以 [配置 Kubernetes Secret 并使用它来部署 Pod](https://kubernetes.io/docs/concepts/containers/images/#specifying-imagepullsecrets-on-a-pod)。

使用 IBM Cloud Container Registry

IBM Cloud Container Registry 提供了一个多租户私有镜像仓库，可以使用它来安全地存储和共享 Docker 仓库。默认情况下，集成的 Vulnerability Advisor 会扫描私有仓库中的镜像，以检测安全问题和潜在的漏洞。IBM Cloud 帐户中的用户可以访问您的镜像，也可以创建令牌来授予对仓库命名空间的访问权限。

要安装 IBM Cloud Container Registry CLI 插件并为镜像创建命名空间，请参阅 [IBM Cloud Container Registry 入门](https://cloud.ibm.com/docs/services/Registry?topic=registry-index#index)。

可以使用 IBM Cloud Container Registry 将容器从 [IBM Cloud 公共镜像](https://cloud.ibm.com/docs/services/Registry?topic=registry-public_images#public_images) 和私有镜像部署到 IBM Cloud Kubernetes Service 集群的默认命名空间。要将容器部署到其他命名空间，或使用来自其他 IBM Cloud Container 的仓库区域或 IBM Cloud 帐户的镜像，请创建 Kubernetes imagePullSecret。有关更多信息，请参阅[从镜像构建容器](https://cloud.ibm.com/docs/containers?topic=containers-images#images)。

配置 Node 对私有仓库认证

**注意：**

如果在 Google Kubernetes Engine 上运行集群，每个节点上都会有 .dockercfg 文件，它包含 Google Container Registry 的凭证。不需要使用以下方法。

**注意：**

如果在 AWS EC2 上运行集群且准备使用 EC2 Container Registry (ECR)，每个 node 上的 kubelet 会管理和更新 ECR 的登录凭证。不需要使用以下方法。

**注意：**

该方法适用于能够对节点进行配置的情况。该方法在 GCE 及在其它能自动配置节点的云平台上并不适合。

**注意：**

截至目前，Kubernetes 仅支持 docker config 的 auths 和 HttpHeaders 部分。这意味着不支持凭据助手（credHelpers 或 credsStore）。

Docker 将私有仓库的密钥存放在 $HOME/.dockercfg 或 $HOME/.docker/config.json 文件中。Kubelet 上，docker 会使用 root 用户 $HOME 路径下的密钥。

* {--root-dir:-/var/lib/kubelet}/config.json
* {cwd of kubelet}/config.json
* ${HOME}/.docker/config.json
* /.docker/config.json
* {--root-dir:-/var/lib/kubelet}/.dockercfg
* {cwd of kubelet}/.dockercfg
* ${HOME}/.dockercfg
* /.dockercfg

**注意：**

可能必须在环境变量文件中为 kubelet 显式设置 HOME=/root。

推荐如下步骤来为 node 配置私有仓库。以下示例在 PC 或笔记本电脑中操作：

1. 对于想要使用的每一种凭证，运行 docker login [server]，它会更新 $HOME/.docker/config.json。
2. 使用编辑器查看 $HOME/.docker/config.json，保证文件中包含了想要使用的凭证。
3. 获取 node 列表，例如
   * 如果想要 node 名称，nodes=$(kubectl get nodes -o jsonpath='{range.items[\*].metadata}{.name} {end}')
   * 如果想要 node IP ，nodes=$(kubectl get nodes -o jsonpath='{range .items[\*].status.addresses[?(@.type=="ExternalIP")]}{.address} {end}')
4. 将本地的 .docker/config.json 拷贝到每个节点 root 用户目录下
   * 例如： for n in $nodes; do scp ~/.docker/config.json root@$n:/root/.docker/config.json; done

创建使用私有仓库的 pod 来验证，例如：

kubectl apply -f - <<EOF

apiVersion: v1

kind: Pod

metadata:

name: private-image-test-1

spec:

containers:

- name: uses-private-image

image: $PRIVATE\_IMAGE\_NAME

imagePullPolicy: Always

command: [ "echo", "SUCCESS" ]

EOF

pod/private-image-test-1 created

如果一切正常，一段时间后，可以看到:

kubectl logs private-image-test-1

SUCCESS

如果失败，则可以看到：

kubectl describe pods/private-image-test-1 | grep "Failed"

Fri, 26 Jun 2015 15:36:13 -0700 Fri, 26 Jun 2015 15:39:13 -0700 19 {kubelet node-i2hq} spec.containers{uses-private-image} failed Failed to pull image "user/privaterepo:v1": Error: image user/privaterepo:v1 not found

必须保证集群中所有的节点都有相同的 .docker/config.json 文件。否则, pod 会在一些节点上正常运行而在另一些节点上无法启动。例如，如果使用 node 自动缩放，那么每个实例模板都需要包含 .docker/config.json，或者挂载一个包含这个文件的驱动器。

在 .docker/config.json 中配置了私有仓库密钥后，所有 pod 都会能读取私有仓库中的镜像。

提前拉取镜像

**注意：**

如果在 Google Kubernetes Engine 上运行集群，每个节点上都会有 .dockercfg 文件，它包含 Google Container Registry 的凭证。不需要使用以下方法。

**注意：**

该方法适用于能够对节点进行配置的情况。该方法在 GCE 及在其它能自动配置节点的云平台上并不适合。

默认情况下，kubelet 会尝试从指定的仓库拉取每一个镜像。但是，如果容器属性 imagePullPolicy 设置为 IfNotPresent或者 Never，则会使用本地镜像（优先、唯一、分别）。

如果依赖提前拉取镜像代替仓库认证，必须保证集群所有的节点提前拉取的镜像是相同的。

可以用于提前载入指定的镜像以提高速度，或者作为私有仓库认证的一种替代方案。

所有的 pod 都可以使用 node 上缓存的镜像。

在 pod 上指定 ImagePullSecrets

**注意：**

Google Kubernetes Engine、GCE 及其他自动创建 node 的云平台上，推荐使用本方法。

Kubernetes 支持在 pod 中指定仓库密钥。

使用 Docker Config 创建 Secret

运行以下命令，将大写字母代替为合适的值：

kubectl create secret docker-registry <name> --docker-server=DOCKER\_REGISTRY\_SERVER --docker-username=DOCKER\_USER --docker-password=DOCKER\_PASSWORD --docker-email=DOCKER\_EMAIL

如果已经有 Docker 凭证文件，则可以将凭证文件作为 Kubernetes secret 导入而不是使用上面的命令。[根据现有 Docker 凭证创建 Secret](https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/#registry-secret-existing-credentials) 解释了如何安装。如果使用多个私有容器仓库，这将特别有用，因为 kubectl create secret docker-registry 创建了一个仅适用于单个私有仓库的 Secret。

**注意：**

Pod 只能引用和它相同命名空间的 ImagePullSecrets，所以需要为每一个命名空间做配置。

引用 Pod 上的 imagePullSecrets

现在，在创建 pod 时，可以在 pod 定义中增加 imagePullSecrets 部分来引用 secret。

cat <<EOF > pod.yaml

apiVersion: v1

kind: Pod

metadata:

name: foo

namespace: awesomeapps

spec:

containers:

- name: foo

image: janedoe/awesomeapp:v1

imagePullSecrets:

- name: myregistrykey

EOF

cat <<EOF >> ./kustomization.yaml

resources:

- pod.yaml

EOF

对每一个使用私有仓库的 pod，都需要做以上操作。

但是，可以通过在 [serviceAccount](https://kubernetes.io/docs/user-guide/service-accounts) 资源中设置 imagePullSecrets 来自动设置 imagePullSecrets。检查 [将 ImagePullSecrets 添加 Service Account](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#add-imagepullsecrets-to-a-service-account) 以获取详细说明。

可以将其与每个节点 .docker/config.json 结合使用。凭据将被合并。这种方法适用于 Google Kubernetes Engine。

使用场景

配置私有仓库有多种方案，以下是一些常用场景和建议的解决方案。

1. 集群运行非专有（例如 开源镜像）镜像。镜像不需要隐藏。
   * 使用 Docker hub 上的公有镜像
   * 无需配置
   * 在 GCE/GKE 上会自动使用高稳定性和高速的 Docker hub 的本地 mirror <!–
2. Cluster running some proprietary images which should be hidden to those outside the company, but visible to all cluster users.
   * Use a hosted private [Docker registry](https://docs.docker.com/registry/).
     + It may be hosted on the [Docker Hub](https://hub.docker.com/signup), or elsewhere.
     + Manually configure .docker/config.json on each node as described above.
   * Or, run an internal private registry behind your firewall with open read access.
     + No Kubernetes configuration is required.
   * Or, when on GCE/Google Kubernetes Engine, use the project’s Google Container Registry.
     + It will work better with cluster autoscaling than manual node configuration.
   * Or, on a cluster where changing the node configuration is inconvenient, use imagePullSecrets. –>
3. 集群运行一些专有镜像，这些镜像对外部公司需要隐藏，对集群用户可见
   * 使用自主的私有 [Docker registry](https://docs.docker.com/registry/)。
     + 可以放置在 [Docker Hub](https://hub.docker.com/account/signup/),或者其他地方。
     + 按照上面的描述，在每个节点手动配置 .docker/config.json。
   * 或者，在防火墙内运行一个内置的私有仓库，并开放读取权限。
     + 不需要配置 Kubenretes。
   * 或者，在 GCE/GKE 上时，使用项目的 Google Container Registry。
     + 使用集群自动伸缩比手动配置 node 工作的更好。
   * 或者，在更改集群 node 配置不方便时，使用 imagePullSecrets。 <!–
4. Cluster with proprietary images, a few of which require stricter access control.
   * Ensure [AlwaysPullImages admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#alwayspullimages) is active. Otherwise, all Pods potentially have access to all images.
   * Move sensitive data into a “Secret” resource, instead of packaging it in an image. –>
5. 使用专有镜像的集群，有更严格的访问控制。
   * 保证开启 [AlwaysPullImages admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#alwayspullimages)。否则，所有的 pod 都可以使用镜像。
   * 将敏感数据存储在 “Secret” 资源中，而不是打包在镜像里。 <!–
6. A multi-tenant cluster where each tenant needs own private registry.
   * Ensure [AlwaysPullImages admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#alwayspullimages) is active. Otherwise, all Pods of all tenants potentially have access to all images.
   * Run a private registry with authorization required.
   * Generate registry credential for each tenant, put into secret, and populate secret to each tenant namespace.
   * The tenant adds that secret to imagePullSecrets of each namespace. –>
7. 多租户集群下，每个租户需要自己的私有仓库。
   * 开启保证 [AlwaysPullImages admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#alwayspullimages)。否则，所有租户的所有的 pod 都可以使用镜像。
   * 私有仓库开启认证。
   * 为每个租户获取仓库凭证，放置在 secret 中，并发布到每个租户的命名空间下。
   * 租户将 secret 增加到每个命名空间下的 imagePullSecrets 中。

如果需要访问多个仓库，则可以为每个仓库创建一个 secret。Kubelet 将任何 imagePullSecrets 合并为单个虚拟 .docker/config.json 文件。

反馈

容器运行时类(Runtime Class)

**FEATURE STATE:** Kubernetes v1.14 [beta](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/)

本页面描述了 RuntimeClass 资源和运行时的选择机制。

**警告：** RuntimeClass 特性在 v1.14 版本升级为 beta 特性时引入了不兼容的改变。 如果你在 v1.14 以前的版本中使用 RuntimeClass，请查阅 [Upgrading RuntimeClass from Alpha to Beta](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/#upgrading-runtimeclass-from-alpha-to-beta)。

* [**Runtime Class**](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/#runtime-class)
* [**Motivation**](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/#motivation)

Runtime Class

RuntimeClass 是用于选择容器运行时配置的特性，容器运行时配置用于运行 Pod 中的容器。

Motivation

您可以在不同的 pod 之间设置不同的 RuntimeClass，以提供性能与安全性之间的平衡。 例如，如果您的部分工作负载需要高级别的信息安全保证，那么您可以选择性地调度这些 pod， 使它们在使用硬件虚拟化的容器运行时中运行。 然后，您将从可选运行时的额外隔离中获益，代价是一些额外的开销。

您还可以使用 RuntimeClass 运行具有相同容器运行时但具有不同设置的pod。

设置

确保 RuntimeClass 特性开关处于开启状态（默认为开启状态）。 关于特性开关的详细介绍，请查阅 [Feature Gates](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/)。 RuntimeClass 特性开关必须在 apiservers 和 kubelets 同时开启。

1. 在节点上配置 CRI 的实现（取决于所选用的运行时）
2. 创建相应的 RuntimeClass 资源

1. 在节点上配置 CRI 实现

RuntimeClass 的配置依赖于 运行时接口（CRI）的实现。 根据你使用的 CRI 实现，查阅相关的文档（[下方](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/#cri-configuration)）来了解如何配置。

**注意：** RuntimeClass 假设集群中的节点配置是同构的 （换言之，所有的节点在容器运行时方面的配置是相同的）。 如果需要支持异构节点，配置方法请参阅下面的 [Scheduling](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/#scheduling)。

所有这些配置都具有相应的 handler 名，并被 RuntimeClass 引用。 handler 必须符合 DNS-1123 命名规范（字母、数字、或 -）。

2. 创建相应的 RuntimeClass 资源

在上面步骤 1 中，每个配置都需要有一个用于标识配置的 handler。 针对每个 handler 需要创建一个 RuntimeClass 对象。

RuntimeClass 资源当前只有两个重要的字段：RuntimeClass 名 (metadata.name) 和 handler (handler)。 对象定义如下所示：

apiVersion: node.k8s.io/v1beta1 *# RuntimeClass is defined in the node.k8s.io API group*

kind: RuntimeClass

metadata:

name: myclass *# The name the RuntimeClass will be referenced by*

*# RuntimeClass is a non-namespaced resource*

handler: myconfiguration *# The name of the corresponding CRI configuration*

**注意：** 建议将 RuntimeClass 写操作（create、update、patch 和 delete）限定于集群管理员使用。 通常这是默认配置。参阅[授权概述](https://kubernetes.io/docs/reference/access-authn-authz/authorization/)了解更多信息。

使用说明

一旦完成集群中 RuntimeClasses 的配置，使用起来非常方便。 在 Pod spec 中指定 runtimeClassName 即可。例如:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

runtimeClassName: myclass

*# ...*

这一设置会告诉 Kubelet 使用所指的 RuntimeClass 来运行该 pod。 如果所指的 RuntimeClass 不存在或者 CRI 无法运行相应的 handler，那么 pod 将会进入 Failed 终止[阶段](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-phase)。 你可以查看相应的[事件](https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application-introspection/)，获取出错信息。

如果未指定 runtimeClassName ，则将使用默认的 RuntimeHandler，相当于禁用 RuntimeClass 功能特性。

CRI Configuration

关于如何安装 CRI 运行时，请查阅[CRI installation](https://kubernetes.io/docs/setup/production-environment/container-runtimes/)。

dockershim

Kubernetes 内置 dockershim CRI 不支持配置运行时 handler。

[containerd](https://containerd.io/)

通过 containerd 的 /etc/containerd/config.toml 配置文件来配置运行时 handler。 handler 需要配置在 runtimes 块中：

[plugins.cri.containerd.runtimes.${HANDLER\_NAME}]

更详细信息，请查阅 containerd 配置文档： <https://github.com/containerd/cri/blob/master/docs/config.md>

[cri-o](https://cri-o.io/)

通过 cri-o 的 /etc/crio/crio.conf 配置文件来配置运行时 handler。 handler 需要配置在[crio.runtime 表](https://github.com/kubernetes-sigs/cri-o/blob/master/docs/crio.conf.5.md#crioruntime-table) 下方：

[crio.runtime.runtimes.${HANDLER\_NAME}]

runtime\_path = "${PATH\_TO\_BINARY}"

更详细信息，请查阅 containerd 配置文档： <https://github.com/kubernetes-sigs/cri-o/blob/master/cmd/crio/config.go>

Scheduling

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/)

在 Kubernetes v1.16 版本里，RuntimeClass 特性引入了 scheduling 字段来支持异构集群。 通过该字段，可以确保 pod 被调度到支持指定运行时的节点上。 该调度支持，需要确保 RuntimeClass [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/) 处于开启状态（1.16 版本默认开启）。

为了确保 pod 会被调度到支持指定运行时的 node 上，每个 node 需要设置一个通用的 label 用于被 runtimeclass.scheduling.nodeSelector 挑选。在 admission 阶段，RuntimeClass 的 nodeSelector 将会于 pod 的 nodeSelector 合并，取二者的交集。如果有冲突，pod 将会被拒绝。

如果 node 需要阻止某些需要特定 RuntimeClass 的 pod，可以在 tolerations 中指定。 与 nodeSelector 一样，tolerations 也在 admission 阶段与 pod 的 tolerations 合并，取二者的并集。

更多有关 node selector 和 tolerations 的配置信息，请查阅 [Assigning Pods to Nodes](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/)。

Pod Overhead

**FEATURE STATE:** Kubernetes v1.16 [alpha](https://kubernetes.io/zh/docs/concepts/containers/runtime-class/)

在 Kubernetes v1.16 版本中，RuntimeClass 开始支持 pod 的 overhead，作为 [PodOverhead](https://kubernetes.io/docs/concepts/configuration/pod-overhead) 特性的一部分。 若要使用 PodOverhead 特性，你需要确保 PodOverhead 特性开关处于开启状态（默认为关闭状态）。

Pod 的 overhead 在 RuntimeClass 的 Overhead 字段定义，该字段用于指定使用 RuntimeClass 特性时带来的 overhead。

Upgrading RuntimeClass from Alpha to Beta

RuntimeClass Beta 特性包含如下几个改变：

* node.k8s.io API 组和 runtimeclasses.node.k8s.io 资源已从 CRD 中迁移到内置的 API 中；
* spec 被放置到 RuntimeClass 中（例如，没有 RuntimeClassSpec 了）；
* runtimeHandler 字段重命名为 handler；
* handler 字段需要在所有版本的 API 提供，这意味着 runtimeHandler 字段在 Alpha API 中也需要提供；
* handler 字段必须是一个合法的 DNS 标识([RFC 1123](https://tools.ietf.org/html/rfc1123))， 这意味着不可以包含 . 字符。合法的 handler 必须满足如下规则：^[a-z0-9]([-a-z0-9]\*[a-z0-9])?$。

**Action Required:** RuntimeClass 特性从 alpha 版本升级到 beta 版本，需要做如下动作：

* RuntimeClass 资源必须在升级到 v1.14 *之后* 再创建，并且 CRD 资源 runtimeclasses.node.k8s.io 必须要手动删除：
* kubectl delete customresourcedefinitions.apiextensions.k8s.io runtimeclasses.node.k8s.io
* RuntimeClasses 中未指定或为空的 runtimeHandler 和 使用包含 . 符号的 handler 将不再合法， 必须迁移成合法的 handler 配置（见上）。

Further Reading

* [RuntimeClass Design](https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/runtime-class.md)
* [RuntimeClass Scheduling Design](https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/runtime-class-scheduling.md)
* Read about the [Pod Overhead](https://kubernetes.io/docs/concepts/configuration/pod-overhead/) concept
* [PodOverhead Feature Design](https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/20190226-pod-overhead.md)

反馈

容器生命周期钩子

这个页面描述了 kubelet 管理的容器如何使用容器生命周期钩子框架来运行在其管理生命周期中由事件触发的代码。

* [**概述**](https://kubernetes.io/zh/docs/concepts/containers/container-lifecycle-hooks/#%e6%a6%82%e8%bf%b0)
* [**容器钩子**](https://kubernetes.io/zh/docs/concepts/containers/container-lifecycle-hooks/#%e5%ae%b9%e5%99%a8%e9%92%a9%e5%ad%90)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/containers/container-lifecycle-hooks/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

概述

类似于许多具有生命周期钩子组件的编程语言框架，例如Angular，Kubernetes为容器提供了生命周期钩子。 钩子使容器能够了解其管理生命周期中的事件，并在执行相应的生命周期钩子时运行在处理程序中实现的代码。

容器钩子

有两个钩子暴露在容器中:

PostStart

这个钩子在创建容器之后立即执行。 但是，不能保证钩子会在容器入口点之前执行。 没有参数传递给处理程序。

PreStop

在容器终止之前是否立即调用此钩子，取决于 API 的请求或者管理事件，类似活动探针故障、资源抢占、资源竞争等等。 如果容器已经完全处于终止或者完成状态，则对 preStop 钩子的调用将失败。 它是阻塞的，同时也是同步的，因此它必须在删除容器的调用之前完成。 没有参数传递给处理程序。

有关终止行为的更详细描述，请参见[终止 Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods)。

钩子处理程序的实现

容器可以通过实现和注册该钩子的处理程序来访问该钩子。 针对容器，有两种类型的钩子处理程序可供实现：

* Exec - 执行一个特定的命令，例如 pre-stop.sh，在容器的 cgroups 和名称空间中。 命令所消耗的资源根据容器进行计算。
* HTTP - 对容器上的特定端点执行 HTTP 请求。

钩子处理程序执行

当调用容器生命周期管理钩子时，Kubernetes 管理系统在为该钩子注册的容器中执行处理程序。

钩子处理程序调用在包含容器的 Pod 上下文中是同步的。 这意味着对于 PostStart 钩子，容器入口点和钩子异步触发。 但是，如果钩子运行或挂起的时间太长，则容器无法达到 running 状态。

行为与 PreStop 钩子的行为类似。 如果钩子在执行过程中挂起，Pod 阶段将保持在 Terminating 状态，并在 Pod 结束的 terminationGracePeriodSeconds 之后被杀死。 如果 PostStart 或 PreStop 钩子失败，它会杀死容器。

用户应该使他们的钩子处理程序尽可能的轻量级。 但也需要考虑长时间运行的命令也很有用的情况，比如在停止容器之前保存状态。

钩子寄送保证

钩子的寄送应该是\*至少一次\*，这意味着对于任何给定的事件，例如 PostStart 或 PreStop，钩子可以被调用多次。 如何正确处理，是钩子实现所要考虑的问题。

通常情况下，只会进行单次寄送。 例如，如果 HTTP 钩子接收器宕机，无法接收流量，则不会尝试重新发送。 然而，偶尔也会发生重复寄送的可能。 例如，如果 kubelet 在发送钩子的过程中重新启动，钩子可能会在 kubelet 恢复后重新发送。

调试钩子处理程序

钩子处理程序的日志不会在 Pod 事件中公开。 如果处理程序由于某种原因失败，它将播放一个事件。 对于 PostStart，这是 FailedPostStartHook 事件，对于 PreStop，这是 FailedPreStopHook 事件。 您可以通过运行 kubectl describe pod <pod\_name> 命令来查看这些事件。 下面是运行这个命令的一些事件输出示例:

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {default-scheduler } Normal Scheduled Successfully assigned test-1730497541-cq1d2 to gke-test-cluster-default-pool-a07e5d30-siqd

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Pulling pulling image "test:1.0"

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Created Created container with docker id 5c6a256a2567; Security:[seccomp=unconfined]

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Pulled Successfully pulled image "test:1.0"

1m 1m 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Started Started container with docker id 5c6a256a2567

38s 38s 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Killing Killing container with docker id 5c6a256a2567: PostStart handler: Error executing in Docker Container: 1

37s 37s 1 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Normal Killing Killing container with docker id 8df9fdfd7054: PostStart handler: Error executing in Docker Container: 1

38s 37s 2 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} Warning FailedSync Error syncing pod, skipping: failed to "StartContainer" for "main" with RunContainerError: "PostStart handler: Error executing in Docker Container: 1"

1m 22s 2 {kubelet gke-test-cluster-default-pool-a07e5d30-siqd} spec.containers{main} Warning FailedPostStartHook

反馈

容器环境变量

本文介绍容器环境中对容器可用的资源。

* [**容器环境**](https://kubernetes.io/zh/docs/concepts/containers/container-environment-variables/#%e5%ae%b9%e5%99%a8%e7%8e%af%e5%a2%83)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/containers/container-environment-variables/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

容器环境

Kubernetes 容器环境为容器提供了几类重要的资源：

* 一个文件系统，其中包含一个[镜像](https://kubernetes.io/docs/concepts/containers/images/)和一个或多个[卷](https://kubernetes.io/docs/concepts/storage/volumes/)。
* 容器本身相关的信息。
* 集群中其他对象相关的信息。

容器信息

容器的 *hostname* 是容器所在的 Pod 名称。 可以通过 hostname 命令或调用 libc 中的 [gethostname](http://man7.org/linux/man-pages/man2/gethostname.2.html) 函数来获取。

Pod 名称和名字空间可以通过 [downward API](https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/) 以环境变量方式访问。

与 Docker 镜像中静态指定的环境变量一样，Pod 中用户定义的环境变量也可用于容器。

集群信息

容器创建时运行的所有服务的列表都会作为环境变量提供给容器。 这些环境变量与 Docker 链接语法相匹配。

对一个名为 *foo* ，映射到名为 *bar* 的容器端口的服务， 会定义如下变量：

FOO\_SERVICE\_HOST=<服务所在的主机地址>

FOO\_SERVICE\_PORT=<服务所启用的端口>

服务具有专用 IP 地址，如果启用了 [DNS 插件](http://releases.k8s.io/master/cluster/addons/dns/)，还可以在容器中通过 DNS 进行访问。

反馈

工作负载

Pods

本节提供了 Pod 的概览信息，Pod 是最小可部署的 Kubernetes 对象模型。

* [**理解 Pod**](https://kubernetes.io/zh/docs/concepts/workloads/pods/#%e7%90%86%e8%a7%a3-pod)
* [**使用 Pod**](https://kubernetes.io/zh/docs/concepts/workloads/pods/#%e4%bd%bf%e7%94%a8-pod)
* [**Pod 模板**](https://kubernetes.io/zh/docs/concepts/workloads/pods/#pod-%e6%a8%a1%e6%9d%bf)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/pods/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

理解 Pod

*Pod* 是 Kubernetes 的基本构建块，它是 Kubernetes 对象模型中创建或部署的最小和最简单的单元。 Pod 表示集群上正在运行的进程。

Pod 封装了应用程序容器（或者在某些情况下封装多个容器）、存储资源、唯一网络 IP 以及控制容器应该如何运行的选项。 Pod 表示部署单元：\*Kubernetes 中应用程序的单个实例\*，它可能由单个容器或少量紧密耦合并共享资源的容器组成。

[Docker](https://www.docker.com/) 是 Kubernetes Pod 中最常用的容器运行时，但 Pod 也能支持其他的[容器运行时](https://kubernetes.io/docs/setup/production-environment/container-runtimes/)。

Kubernetes 集群中的 Pod 可被用于以下两个主要用途：

* **运行单个容器的 Pod**。”每个 Pod 一个容器”模型是最常见的 Kubernetes 用例；在这种情况下，可以将 Pod 看作单个容器的包装器，并且 Kubernetes 直接管理 Pod，而不是容器。
* **运行多个协同工作的容器的 Pod**。 Pod 可能封装由多个紧密耦合且需要共享资源的共处容器组成的应用程序。 这些位于同一位置的容器可能形成单个内聚的服务单元——一个容器将文件从共享卷提供给公众，而另一个单独的“挂斗”容器则刷新或更新这些文件。 Pod 将这些容器和存储资源打包为一个可管理的实体。

[Kubernetes 博客](http://kubernetes.io/blog) 上有一些其他的 Pod 用例信息。更多信息请参考：

* [分布式系统工具包：复合容器的模式](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns)
* [容器设计模式](https://kubernetes.io/blog/2016/06/container-design-patterns)

每个 Pod 表示运行给定应用程序的单个实例。如果希望横向扩展应用程序（例如，运行多个实例），则应该使用多个 Pod，每个实例使用一个。 在 Kubernetes 中，这通常被称为 \_副本\_。 通常使用一个称为控制器的抽象来创建和管理一组被复制的 Pod。 更多信息请参见 [POD 和控制器](https://kubernetes.io/zh/docs/concepts/workloads/pods/#pods-and-controllers)。

Pod 怎样管理多个容器

Pod 被设计成支持形成内聚服务单元的多个协作过程（作为容器）。 Pod 中的容器被自动的安排到集群中的同一物理或虚拟机上，并可以一起进行调度。 容器可以共享资源和依赖、彼此通信、协调何时以及何种方式终止它们。

注意，在单个 Pod 中将多个并置和共同管理的容器分组是一个相对高级的使用方式。 只在容器紧密耦合的特定实例中使用此模式。 例如，您可能有个充当共享卷中文件的 Web 服务器的容器，以及从远程源更新这些文件的单独的”挂斗”容器，如下图所示：

pod diagram

Pod 为其组成容器提供了两种共享资源：*网络* 和 \*存储\*。

联网

每个 Pod 分配一个唯一的 IP 地址。 Pod 中的每个容器共享网络命名空间，包括 IP 地址和网络端口。 *Pod 内的容器* 可以使用 localhost 互相通信。 当 Pod 中的容器与 *Pod 之外* 的实体通信时，它们必须协调如何使用共享的网络资源（例如端口）。

存储

一个 Pod 可以指定一组共享存储 \*卷\*。 Pod 中的所有容器都可以访问共享卷，允许这些容器共享数据。 卷还允许 Pod 中的持久数据在重启 Pod 中的某个容器时存活。 有关 Kubernetes 如何在 Pod 中实现共享存储的更多信息，请参考 [卷](https://kubernetes.io/docs/concepts/storage/volumes/)。

使用 Pod

你很少在 Kubernetes 中直接创建单独的 Pod，甚至是单个存在的 Pod。 这是因为 Pod 被设计成了相对短暂的一次性的实体。 当 Pod 由您创建或者间接地由控制器创建时，它被调度在集群中的节点上运行。 Pod 会保持在该节点上运行，直到进程被终止、Pod 对象被删除、Pod 因资源不足而被 *驱逐* 或者节点失效为止。

**注意：**

重启 Pod 中的容器不应与重启 Pod 混淆。Pod 本身不运行，而是作为容器运行的环境，并且一直保持到被删除为止。

Pod 本身并不能自愈。 如果 Pod 被调度到失败的节点，或者如果调度操作本身失败，则删除该 Pod；同样，由于缺乏资源或进行节点维护，Pod 在被驱逐后将不再生存。 Kubernetes 使用了一个更高级的称为 *控制器* 的抽象，由它处理相对可丢弃的 Pod 实例的管理工作。 因此，虽然可以直接使用 Pod，但在 Kubernetes 中，更为常见的是使用控制器管理 Pod。 有关 Kubernetes 如何使用控制器实现 Pod 伸缩和愈合的更多信息，请参考 [Pod 和控制器](https://kubernetes.io/zh/docs/concepts/workloads/pods/#pods-and-controllers)。

Pod 和控制器

控制器可以为您创建和管理多个 Pod，管理副本和上线，并在集群范围内提供自修复能力。 例如，如果一个节点失败，控制器可以在不同的节点上调度一样的替身来自动替换 Pod。

包含一个或多个 Pod 的控制器一些示例包括：

* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)

控制器通常使用您提供的 Pod 模板来创建它所负责的 Pod。

Pod 模板

Pod 模板是包含在其他对象中的 Pod 规范，例如 [Replication Controllers](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/)、 [Jobs](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/)、和 [DaemonSets](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)。 控制器使用 Pod 模板来制作实际使用的 Pod。 下面的示例是一个简单的 Pod 清单，它包含一个打印消息的容器。

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox

command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']

Pod 模板就像饼干切割器，而不是指定所有副本的当前期望状态。 一旦饼干被切掉，饼干就与切割器没有关系。 没有“量子纠缠”。 随后对模板的更改或甚至切换到新的模板对已经创建的 Pod 没有直接影响。 类似地，由副本控制器创建的 Pod 随后可以被直接更新。 这与 Pod 形成有意的对比，Pod 指定了属于 Pod 的所有容器的当前期望状态。 这种方法从根本上简化了系统语义，增加了原语的灵活性。

Pod 概览

本节提供了 Pod 的概览信息，Pod 是最小可部署的 Kubernetes 对象模型。

* [**理解 Pod**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#%e7%90%86%e8%a7%a3-pod)
* [**使用 Pod**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#%e4%bd%bf%e7%94%a8-pod)
* [**Pod 模板**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#pod-%e6%a8%a1%e6%9d%bf)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

理解 Pod

*Pod* 是 Kubernetes 的基本构建块，它是 Kubernetes 对象模型中创建或部署的最小和最简单的单元。 Pod 表示集群上正在运行的进程。

Pod 封装了应用程序容器（或者在某些情况下封装多个容器）、存储资源、唯一网络 IP 以及控制容器应该如何运行的选项。 Pod 表示部署单元：\*Kubernetes 中应用程序的单个实例\*，它可能由单个容器或少量紧密耦合并共享资源的容器组成。

[Docker](https://www.docker.com/) 是 Kubernetes Pod 中最常用的容器运行时，但 Pod 也能支持其他的[容器运行时](https://kubernetes.io/docs/setup/production-environment/container-runtimes/)。

Kubernetes 集群中的 Pod 可被用于以下两个主要用途：

* **运行单个容器的 Pod**。”每个 Pod 一个容器”模型是最常见的 Kubernetes 用例；在这种情况下，可以将 Pod 看作单个容器的包装器，并且 Kubernetes 直接管理 Pod，而不是容器。
* **运行多个协同工作的容器的 Pod**。 Pod 可能封装由多个紧密耦合且需要共享资源的共处容器组成的应用程序。 这些位于同一位置的容器可能形成单个内聚的服务单元——一个容器将文件从共享卷提供给公众，而另一个单独的“挂斗”容器则刷新或更新这些文件。 Pod 将这些容器和存储资源打包为一个可管理的实体。

[Kubernetes 博客](http://kubernetes.io/blog) 上有一些其他的 Pod 用例信息。更多信息请参考：

* [分布式系统工具包：复合容器的模式](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns)
* [容器设计模式](https://kubernetes.io/blog/2016/06/container-design-patterns)

每个 Pod 表示运行给定应用程序的单个实例。如果希望横向扩展应用程序（例如，运行多个实例），则应该使用多个 Pod，每个实例使用一个。 在 Kubernetes 中，这通常被称为 \_副本\_。 通常使用一个称为控制器的抽象来创建和管理一组被复制的 Pod。 更多信息请参见 [POD 和控制器](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#pods-and-controllers)。

Pod 怎样管理多个容器

Pod 被设计成支持形成内聚服务单元的多个协作过程（作为容器）。 Pod 中的容器被自动的安排到集群中的同一物理或虚拟机上，并可以一起进行调度。 容器可以共享资源和依赖、彼此通信、协调何时以及何种方式终止它们。

注意，在单个 Pod 中将多个并置和共同管理的容器分组是一个相对高级的使用方式。 只在容器紧密耦合的特定实例中使用此模式。 例如，您可能有个充当共享卷中文件的 Web 服务器的容器，以及从远程源更新这些文件的单独的”挂斗”容器，如下图所示：

pod diagram

Pod 为其组成容器提供了两种共享资源：*网络* 和 \*存储\*。

联网

每个 Pod 分配一个唯一的 IP 地址。 Pod 中的每个容器共享网络命名空间，包括 IP 地址和网络端口。 *Pod 内的容器* 可以使用 localhost 互相通信。 当 Pod 中的容器与 *Pod 之外* 的实体通信时，它们必须协调如何使用共享的网络资源（例如端口）。

存储

一个 Pod 可以指定一组共享存储 \*卷\*。 Pod 中的所有容器都可以访问共享卷，允许这些容器共享数据。 卷还允许 Pod 中的持久数据在重启 Pod 中的某个容器时存活。 有关 Kubernetes 如何在 Pod 中实现共享存储的更多信息，请参考 [卷](https://kubernetes.io/docs/concepts/storage/volumes/)。

使用 Pod

你很少在 Kubernetes 中直接创建单独的 Pod，甚至是单个存在的 Pod。 这是因为 Pod 被设计成了相对短暂的一次性的实体。 当 Pod 由您创建或者间接地由控制器创建时，它被调度在集群中的节点上运行。 Pod 会保持在该节点上运行，直到进程被终止、Pod 对象被删除、Pod 因资源不足而被 *驱逐* 或者节点失效为止。

**注意：**

重启 Pod 中的容器不应与重启 Pod 混淆。Pod 本身不运行，而是作为容器运行的环境，并且一直保持到被删除为止。

Pod 本身并不能自愈。 如果 Pod 被调度到失败的节点，或者如果调度操作本身失败，则删除该 Pod；同样，由于缺乏资源或进行节点维护，Pod 在被驱逐后将不再生存。 Kubernetes 使用了一个更高级的称为 *控制器* 的抽象，由它处理相对可丢弃的 Pod 实例的管理工作。 因此，虽然可以直接使用 Pod，但在 Kubernetes 中，更为常见的是使用控制器管理 Pod。 有关 Kubernetes 如何使用控制器实现 Pod 伸缩和愈合的更多信息，请参考 [Pod 和控制器](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-overview/#pods-and-controllers)。

Pod 和控制器

控制器可以为您创建和管理多个 Pod，管理副本和上线，并在集群范围内提供自修复能力。 例如，如果一个节点失败，控制器可以在不同的节点上调度一样的替身来自动替换 Pod。

包含一个或多个 Pod 的控制器一些示例包括：

* [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)
* [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)

控制器通常使用您提供的 Pod 模板来创建它所负责的 Pod。

Pod 模板

Pod 模板是包含在其他对象中的 Pod 规范，例如 [Replication Controllers](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/)、 [Jobs](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/)、和 [DaemonSets](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)。 控制器使用 Pod 模板来制作实际使用的 Pod。 下面的示例是一个简单的 Pod 清单，它包含一个打印消息的容器。

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox

command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']

Pod 模板就像饼干切割器，而不是指定所有副本的当前期望状态。 一旦饼干被切掉，饼干就与切割器没有关系。 没有“量子纠缠”。 随后对模板的更改或甚至切换到新的模板对已经创建的 Pod 没有直接影响。 类似地，由副本控制器创建的 Pod 随后可以被直接更新。 这与 Pod 形成有意的对比，Pod 指定了属于 Pod 的所有容器的当前期望状态。 这种方法从根本上简化了系统语义，增加了原语的灵活性。

Pods

*Pods* 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。

* [**Pod 是什么？**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#pod-%e6%98%af%e4%bb%80%e4%b9%88)
* [**Pod 的动机**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#pod-%e7%9a%84%e5%8a%a8%e6%9c%ba)
* [**使用 Pod**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#%e4%bd%bf%e7%94%a8-pod)
* [**备选方案的思考**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#%e5%a4%87%e9%80%89%e6%96%b9%e6%a1%88%e7%9a%84%e6%80%9d%e8%80%83)
* [**Pod 的持久性（或稀缺性）**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#pod-%e7%9a%84%e6%8c%81%e4%b9%85%e6%80%a7-%e6%88%96%e7%a8%80%e7%bc%ba%e6%80%a7)
* [**Pod 的终止**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#pod-%e7%9a%84%e7%bb%88%e6%ad%a2)
* [**Pod 容器的特权模式**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#pod-%e5%ae%b9%e5%99%a8%e7%9a%84%e7%89%b9%e6%9d%83%e6%a8%a1%e5%bc%8f)
* [**API Object**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#api-object)
* [**API 对象**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod/#api-%e5%af%b9%e8%b1%a1)

Pod 是什么？

*pod* （就像在鲸鱼荚或者豌豆荚中）是一组（一个或多个）容器（例如 Docker 容器），这些容器有共享存储、网络、以及怎样运行这些容器的声明。Pod 的内容总是共同定位和共同调度，并且在共享的上下文中运行。 Pod 以特定于应用的“逻辑主机”为模型，它包含一个或多个应用容器，这些容器是相对紧密的耦合在一起；在容器出现之前，在相同的物理机或虚拟机上运行意味着在相同的逻辑主机上运行。

虽然 Kubernetes 支持多种容器运行时，但 Docker 是最常见的一种运行时，它有助于描述 Docker 术语中的 Pod。

Pod 的共享上下文是一组 Linux 命名空间、cgroups、以及其他潜在的资源隔离相关的因素，这些相同的东西也隔离了 Docker 容器。在 Pod 的上下文中，单个应用程序可能还会应用进一步的子隔离。

Pod 中的所有容器共享一个 IP 地址和端口空间，并且可以通过 localhost 互相发现。他们也能通过标准的进程间通信（如 SystemV 信号量或 POSIX 共享内存）方式进行互相通信。不同 Pod 中的容器的 IP 地址互不相同，没有 [特殊配置](https://kubernetes.io/docs/concepts/policy/pod-security-policy/) 就不能使用 IPC 进行通信。这些容器之间经常通过 Pod IP 地址进行通信。

Pod 中的应用也能访问共享卷，共享卷是 Pod 定义的一部分，可被用来挂载到每个应用的文件系统上。

在 [Docker](https://www.docker.com/) 体系的术语中，Pod 被建模为一组具有共享命名空间和共享 [卷](https://kubernetes.io/docs/concepts/storage/volumes/) 的 Docker 容器。

与单个应用程序容器一样，Pod被认为是相对短暂的（而不是持久的）实体。如 [Pod 的生命](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/) 所讨论的那样：Pod 被创建、给它指定一个唯一 ID （UID）、被调度到节点、在节点上存续直到终止（取决于重启策略）或被删除。如果节点宕机，调度到该节点上的 Pod 会在一个超时周期后被安排删除。给定 Pod （由 UID 定义）不会重新调度到新节点；相反，它会被一个完全相同的 Pod 替换掉，如果需要甚至连 Pod 名称都可以一样，除了 UID 是新的(更多信息请查阅 [副本控制器（replication controller）](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/))。

当某些东西被说成与 Pod（如卷）具有相同的生存期时，这意味着只要pod（具有该UID）存在，它就存在。如果那个 Pod 因为某种原因被删除了，即使创建了相同的 Pod 做替换，Pod 的相关的资源（如卷等）也会被销毁进行重新创建。

pod diagram

包含多个容器的 Pod 包含文件拉取器和 web 服务器，使用了持久卷在容器之间进行存储共享。

pod diagram

Pod 的动机

管理

Pod 是形成内聚服务单元的多个协作过程模式的模型。它们提供了一个比它们的应用组成集合更高级的抽象，从而简化了应用的部署和管理。Pod 可以用作部署单元、水平缩放和复制。在 Pod中，多个容器的共同定位（协同调度）、共享命运（例如，终止）、协调复制、资源共享和依赖项管理都是自动处理的。

资源共享和通信

Pod 使它的组成容器间能够进行数据共享和通信。

Pod 中的应用都使用相同的网络命名空间（相同 IP 和 端口空间），而且能够互相“发现”并使用 localhost 进行通信。因此，在 Pod 中的应用必须协调它们的端口使用情况。每个 Pod 在扁平的共享网络空间中具有一个IP地址，该空间能与整个网络上的其他物理机或虚拟机进行完全通信。

主机名被设置为 Pod 内的应用程序容器的 Pod 名称。请参考[组网的更多信息](https://kubernetes.io/docs/concepts/cluster-administration/networking/)。

Pod 除了定义了 Pod 中运行的应用程序容器之外，Pod 还指定了一组共享存储卷。该共享存储卷能使数据在容器重新启动后继续保留，并能在 Pod 内的应用程序之间共享。

使用 Pod

Pod 可以用于托管垂直集成的应用程序栈（例如，LAMP），但最主要的动机是支持位于同一位置的、共同管理的帮助程序，例如：

* 内容管理系统、文件和数据加载器、本地缓存管理器等。
* 日志和检查点备份、压缩、旋转、快照等。
* 数据更改监视器、日志跟踪器、日志和监视适配器、事件发布器等。
* 代理、桥接器和适配器
* 控制器、管理器、配置器和更新器

通常，单个 Pod 并不打算运行同一应用程序的多个实例。

更多解释，请参考 [分布式系统工具包：复合容器的模式](https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns)

备选方案的思考

*为什么不在单个（Docker）容器中运行多个程序？*

1. 透明度。Pod 内的容器对基础设施可见，使得基础设施能够向这些容器提供服务，例如流程管理和资源监控。这为用户提供了许多便利。
2. 解耦软件依赖关系。可以独立地对单个容器进行版本控制、重新构建和重新部署。Kubernetes 有一天甚至可能支持单个容器的实时更新。
3. 易用性。用户不需要运行他们自己的进程管理器、也不用担心信号和退出代码传播等。
4. 效率。因为基础结构承担了更多的责任，所以容器可以变得更加轻量化。

*为什么不支持基于亲和性的容器联合调度？*

这种处理方法尽管可以提供同址，但不能提供 Pod 的大部分好处，如资源共享、IPC、有保证的命运共享和简化的管理。

Pod 的持久性（或稀缺性）

Pod 并没想被视为持久的实体。它们无法在调度失败、节点故障或其他驱逐策略（例如由于缺乏资源或在节点维护的情况下）中生存。

一般来说，用户不需要直接创建 Pod。他们几乎都是使用控制器进行创建，即使对于单例的创建也一样使用控制器，例如 [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)。 控制器提供集群范围的自修复以及复制和滚动管理。 像 [StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset.md) 这样的控制器还可以提供支持有状态的 Pod。

在集群调度系统中，使用 API 合集作为面向用户的主要原语是比较常见的，包括 [Borg](https://research.google.com/pubs/pub43438.html)、[Marathon](https://mesosphere.github.io/marathon/docs/rest-api.html)、[Aurora](http://aurora.apache.org/documentation/latest/reference/configuration/#job-schema)、和 [Tupperware](https://www.slideshare.net/Docker/aravindnarayanan-facebook140613153626phpapp02-37588997)。

Pod 暴露为原语是为了便于：

* 调度器和控制器可插拔性
* 支持 Pod 级别的操作，而不需要通过控制器 API “代理” 它们
* Pod 生命与控制器生命的解耦，如自举
* 控制器和服务的解耦；端点控制器只监视 Pod
* Kubelet 级别的功能与集群级别功能的清晰组合；Kubelet 实际上是 “Pod 控制器”
* 高可用性应用程序期望在 Pod 终止之前并且肯定要在 Pod 被删除之前替换 Pod，例如在计划驱逐或镜像预先提取的情况下。

Pod 的终止

因为 Pod 代表在集群中的节点上运行的进程，所以当不再需要这些进程时（与被 KILL 信号粗暴地杀死并且没有机会清理相比），允许这些进程优雅地终止是非常重要的。 用户应该能够请求删除并且知道进程何时终止，但是也能够确保删除最终完成。当用户请求删除 Pod 时，系统会记录在允许强制删除 Pod 之前所期望的宽限期，并向每个容器中的主进程发送 TERM 信号。一旦过了宽限期，KILL 信号就发送到这些进程，然后就从 API 服务器上删除 Pod。如果 Kubelet 或容器管理器在等待进程终止时发生重启，则终止操作将以完整的宽限期进行重试。

流程示例：

1. 用户发送命令删除 Pod，使用的是默认的宽限期（30秒）
2. API 服务器中的 Pod 会随着宽限期规定的时间进行更新，过了这个时间 Pod 就会被认为已 “死亡”。
3. 当使用客户端命令查询 Pod 状态时，Pod 显示为 “Terminating”。
4. （和第3步同步进行）当 Kubelet 看到 Pod 由于步骤2中设置的时间而被标记为 terminating 状态时，它就开始执行关闭 Pod 流程。
   1. 如果 Pod 定义了 [preStop 钩子](https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/#hook-details)，就在 Pod 内部调用它。如果宽限期结束了 preStop 钩子还在运行，那么就用小的（2秒）扩展宽限期调用步骤2。
   2. 给 Pod 内的进程发送 TERM 信号。
5. （和第3步同步进行）从服务的端点列表中删除 Pod，Pod也不再被视为副本控制器的运行状态的 Pod 集的一部分。当负载平衡器（如服务代理）将 Pod 从轮换中移除时，关闭迟缓的 Pod 将不能继续为流量服务。
6. 当宽限期结束后，Pod 中运行的任何进程都将被用 SIGKILL 杀死。
7. Kubelet 将通过设置宽限期为0（立即删除）来完成删除 API 服务器上的 Pod。Pod 从 API 中消失，从客户端也不再可见。

默认情况下，所有删除在30秒内都是优雅的。kubectl delete 命令支持 --grace-period=<seconds> 选项，允许用户覆盖默认值并声明他们自己的宽限期。设置为 ‘0’ 会[强制删除](https://kubernetes.io/docs/concepts/workloads/pods/pod/#force-deletion-of-pods) Pod。在 kubectl 1.5以后的版本（含1.5版本）中，为了执行强制删除，您还必须为 --grace-period=0 声明一个额外的 --force 标志。

Pod 的强制删除

强制删除pod定义为从集群状态和 etcd 中立即删除 Pod。当执行强制删除时，apiserver 并不会等待 kubelet 的确认信息，该 Pod 已在所运行的节点上被终止了。强制删除操作从 API 中立即清除了 Pod， 因此可以用相同的名称创建一个新的 Pod。在节点上，设置为立即终止的 Pod 还是会在被强制删除前设置一个小的宽限期。

强制删除对某些 Pod 可能具有潜在危险，因此应该谨慎地执行。对于 StatefulSet 管理的 Pod，请参考 [从 StatefulSet 中删除 Pod](https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/)的任务文档。

Pod 容器的特权模式

从 Kubernetes v1.1 开始，Pod 内的任何容器都可以开启特权模式，开启的方法是在容器声明中的 SecurityContext 域使用 privileged 标志。这对于希望使用linux功能（如操作网络堆栈和访问设备）的容器非常有用。容器内的进程获得的特权与容器外的进程几乎相同。使用特权模式，将网络和卷插件编写为不需要编译到 kubelet 中的独立的 Pod 应该更容易。

如果主节点的 Kubernetes 版本是 v1.1 或更高，而工作节点的版本低于 v1.1，新的特权 Pod 将被 API 服务器接受，但不会被启动。它们将处于 pending 状态。如果用户使用命令 kubectl describe pod FooPodName，就可以看到 Pod 处于 pending 状态的原因。describe 命令的输出事件列表将显示： Error validating pod "FooPodName"."FooPodNamespace" from api, ignoring: spec.containers[0].securityContext.privileged: forbidden '<\*>(0xc2089d3248)true'

如果主节点的版本低于 v1.1，那么特权 Pod 就不能被创建。如果用户尝试创建有特权容器的 Pod，将会得到下面的错误信息： The Pod "FooPodName" is invalid. spec.containers[0].securityContext.privileged: forbidden '<\*>(0xc20b222db0)true'

<!–

API Object

Pod is a top-level resource in the Kubernetes REST API. More details about the API object can be found at: [Pod API object](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#pod-v1-core).

–>

API 对象

Pod 是 Kubernetes REST API 中的顶级资源。关于该 API 对象的更详细信息请参考 [Pod API 对象](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#pod-v1-core)。

反馈

Pod Preset

本文提供了 PodPreset 的概述。 在 Pod 创建时，用户可以使用 PodPreset 对象将特定信息注入 Pod 中，这些信息可以包括 secret、 卷、卷挂载和环境变量。

* [**理解 Pod Preset**](https://kubernetes.io/zh/docs/concepts/workloads/pods/podpreset/#%e7%90%86%e8%a7%a3-pod-preset)
* [**PodPreset 如何工作**](https://kubernetes.io/zh/docs/concepts/workloads/pods/podpreset/#podpreset-%e5%a6%82%e4%bd%95%e5%b7%a5%e4%bd%9c)
* [**启用 Pod Preset**](https://kubernetes.io/zh/docs/concepts/workloads/pods/podpreset/#%e5%90%af%e7%94%a8-pod-preset)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/pods/podpreset/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

理解 Pod Preset

Pod Preset 是一种 API 资源，在 Pod 创建时，用户可以用它将额外的运行时需求信息注入 Pod。 使用[标签选择器（label selector）](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)来指定 Pod Preset 所适用的 Pod。

使用 Pod Preset 使得 Pod 模板编写者不必显式地为每个 Pod 设置信息。 这样，使用特定服务的 Pod 模板编写者不需要了解该服务的所有细节。

了解更多的相关背景信息，请参考 [PodPreset 设计提案](https://git.k8s.io/community/contributors/design-proposals/service-catalog/pod-preset.md)。

PodPreset 如何工作

Kubernetes 提供了准入控制器 (PodPreset)，该控制器被启用时，会将 Pod Preset 应用于接收到的 Pod 创建请求中。 当出现 Pod 创建请求时，系统会执行以下操作：

1. 检索所有可用 PodPresets 。
2. 检查 PodPreset 的标签选择器与要创建的 Pod 的标签是否匹配。
3. 尝试合并 PodPreset 中定义的各种资源，并注入要创建的 Pod。
4. 发生错误时抛出事件，该事件记录了 pod 信息合并错误，同时在 *不注入* PodPreset 信息的情况下创建 Pod。
5. 为改动的 Pod spec 添加注解，来表明它被 PodPreset 所修改。 注解形如： podpreset.admission.kubernetes.io/podpreset-<pod-preset name>": "<resource version>"。

一个 Pod 可能不与任何 Pod Preset 匹配，也可能匹配多个 Pod Preset。 同时，一个 PodPreset 可能不应用于任何 Pod，也可能应用于多个 Pod。 当 PodPreset 应用于一个或多个 Pod 时，Kubernetes 修改 pod spec。 对于 Env、 EnvFrom 和 VolumeMounts 的改动， Kubernetes 修改 pod 中所有容器的规格，对于卷的改动，Kubernetes 修改 Pod spec。

**注意：**

Pod Preset 能够在适当的时候修改 Pod spec 的 spec.containers 字段，但是不会应用于 initContainers 字段。

为特定 Pod 禁用 Pod Preset

在一些情况下，用户不希望 Pod 被 Pod Preset 所改动，这时，用户可以在 Pod spec 中添加形如 podpreset.admission.kubernetes.io/exclude: "true" 的注解。

启用 Pod Preset

为了在集群中使用 Pod Preset，必须确保以下几点：

1. 已启用 API 类型 settings.k8s.io/v1alpha1/podpreset。 这可以通过在 API 服务器的 --runtime-config 配置项中包含 settings.k8s.io/v1alpha1=true 来实现。
2. 已启用准入控制器 PodPreset。 启用的一种方式是在 API 服务器的 --admission-control 配置项中包含 PodPreset 。
3. 已经通过在相应的命名空间中创建 PodPreset 对象，定义了 Pod Preset。

* [使用 PodPreset 将信息注入 Pod](https://kubernetes.io/docs/tasks/inject-data-application/podpreset/)

Pod 拓扑扩展约束

**FEATURE STATE:** Kubernetes v1.16 [alpha](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-topology-spread-constraints/)

可以使用\_拓扑扩展约束\_来控制 [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 在集群内故障域（例如地区，区域，节点和其他用户自定义拓扑域）之间的分布。这可以帮助实现高可用以及提升资源利用率。

* [**先决条件**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-topology-spread-constraints/#%e5%85%88%e5%86%b3%e6%9d%a1%e4%bb%b6)
* [**Pod 的拓扑约束**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-topology-spread-constraints/#pod-%e7%9a%84%e6%8b%93%e6%89%91%e7%ba%a6%e6%9d%9f)
* [**与 PodAffinity/PodAntiAffinity 相比较**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-topology-spread-constraints/#%e4%b8%8e-podaffinity-podantiaffinity-%e7%9b%b8%e6%af%94%e8%be%83)
* [**已知局限性**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-topology-spread-constraints/#%e5%b7%b2%e7%9f%a5%e5%b1%80%e9%99%90%e6%80%a7)

先决条件

启用功能

确保 EvenPodsSpread 功能已开启（在 1.16 版本中该功能默认关闭）。阅读[功能选项](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/)了解如何开启该功能。EvenPodsSpread 必须在 [API Server](https://kubernetes.io/docs/reference/generated/kube-apiserver/) **和** [scheduler](https://kubernetes.io/docs/reference/generated/kube-scheduler/) 中都要开启。

节点标签

拓扑扩展约束依赖于节点标签来标识每个节点所在的拓扑域。例如，一个节点可能具有标签：node=node1,zone=us-east-1a,region=us-east-1

假设你拥有一个具有以下标签的 4 节点集群：

NAME STATUS ROLES AGE VERSION LABELS

node1 Ready <none> 4m26s v1.16.0 node=node1,zone=zoneA

node2 Ready <none> 3m58s v1.16.0 node=node2,zone=zoneA

node3 Ready <none> 3m17s v1.16.0 node=node3,zone=zoneB

node4 Ready <none> 2m43s v1.16.0 node=node4,zone=zoneB

然后从逻辑上看集群如下：

+---------------+---------------+

| zoneA | zoneB |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

可以复用在大多数集群上自动创建和填充的[知名标签](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/)，而不是手动添加标签。

Pod 的拓扑约束

API

pod.spec.topologySpreadConstraints 字段定义如下所示：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

topologySpreadConstraints:

- maxSkew: <integer*>*

*topologyKey: <string>*

whenUnsatisfiable: <string*>*

*labelSelector: <object>*

可以定义一个或多个 topologySpreadConstraint 来指示 kube-scheduler 如何将每个传入的 Pod 根据与现有的 Pod 的关联关系在集群中部署。字段包括：

* **maxSkew** 描述 pod 分布不均的程度。这是给定拓扑类型中任意两个拓扑域中匹配的 pod 之间的最大允许差值。它必须大于零。
* **topologyKey** 是节点标签的键。如果两个节点使用此键标记并且具有相同的标签值，则调度器会将这两个节点视为处于同一拓扑中。调度器试图在每个拓扑域中放置数量均衡的 pod。
* **whenUnsatisfiable** 指示如果 pod 不满足扩展约束时如何处理：
  + DoNotSchedule（默认）告诉调度器不用进行调度。
  + ScheduleAnyway 告诉调度器在对最小化倾斜的节点进行优先级排序时仍对其进行调度。
* **labelSelector** 用于查找匹配的 pod。匹配此标签的 pod 将被统计，以确定相应拓扑域中 pod 的数量。有关详细信息，请参考[标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)。

执行 kubectl explain Pod.spec.topologySpreadConstraints 命令了解更多关于 topologySpreadConstraints 的信息。

例子：单个拓扑扩展约束

假设你拥有一个 4 节点集群，其中标记为 foo:bar 的 3 个 pod 分别位于 node1，node2 和 node3 中（P 表示 pod）：

+---------------+---------------+

| zoneA | zoneB |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

| P | P | P | |

+-------+-------+-------+-------+

如果希望传入的 pod 均匀散布在现有的 pod 区域，则可以指定字段如下：

| [pods/topology-spread-constraints/one-constraint.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/topology-spread-constraints/one-constraint.yaml) |
| --- |
| kind: Pod  apiVersion: v1  metadata:  name: mypod  labels:  foo: bar  spec:  topologySpreadConstraints:  - maxSkew: 1  topologyKey: zone  whenUnsatisfiable: DoNotSchedule  labelSelector:  matchLabels:  foo: bar  containers:  - name: pause  image: k8s.gcr.io/pause:3.1 |

topologyKey: zone 意味着均匀分布将只应用于存在标签对为 “zone:” 的节点上。whenUnsatisfiable: DoNotSchedule 告诉调度器，如果传入的 pod 不满足约束，则让它保持挂起状态。

如果调度器将传入的 pod 放入 “zoneA”，pod 分布将变为 [3, 1]，因此实际的倾斜为 2（3 - 1）。这违反了 maxSkew: 1。此示例中，传入的 pod 只能放置在 “zoneB” 上：

+---------------+---------------+ +---------------+---------------+

| zoneA | zoneB | | zoneA | zoneB |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

| node1 | node2 | node3 | node4 | OR | node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

| P | P | P | P | | P | P | P P | |

+-------+-------+-------+-------+ +-------+-------+-------+-------+

可以调整 pod 规格以满足各种要求：

* 将 maxSkew 更改为更大的值，比如 “2”，这样传入的 pod 也可以放在 “zoneA” 上。
* 将 topologyKey 更改为 “node”，以便将 pod 均匀分布在节点上而不是区域中。在上面的例子中，如果 maxSkew 保持为 “1”，那么传入的 pod 只能放在 “node4” 上。
* 将 whenUnsatisfiable: DoNotSchedule 更改为 whenUnsatisfiable: ScheduleAnyway，以确保传入的 pod 始终可以调度（假设满足其他的调度 API）。但是，最好将其放置在具有较少匹配 pod 的拓扑域中。（请注意，此优先性与其他内部调度优先级（如资源使用率等）一起进行标准化。）

例子：多个拓扑扩展约束

下面的例子建立在前面例子的基础上。假设你拥有一个 4 节点集群，其中 3 个标记为 foo:bar 的 pod 分别位于 node1，node2 和 node3 上（P 表示 pod）：

+---------------+---------------+

| zoneA | zoneB |

+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 |

+-------+-------+-------+-------+

| P | P | P | |

+-------+-------+-------+-------+

可以使用 2 个拓扑扩展约束来控制 pod 在 区域和节点两个维度上进行分布：

| [pods/topology-spread-constraints/two-constraints.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/topology-spread-constraints/two-constraints.yaml) |
| --- |
| kind: Pod  apiVersion: v1  metadata:  name: mypod  labels:  foo: bar  spec:  topologySpreadConstraints:  - maxSkew: 1  topologyKey: zone  whenUnsatisfiable: DoNotSchedule  labelSelector:  matchLabels:  foo: bar  - maxSkew: 1  topologyKey: node  whenUnsatisfiable: DoNotSchedule  labelSelector:  matchLabels:  foo: bar  containers:  - name: pause  image: k8s.gcr.io/pause:3.1 |

在这种情况下，为了匹配第一个约束，传入的 pod 只能放置在 “zoneB” 中；而在第二个约束中，传入的 pod 只能放置在 “node4” 上。然后两个约束的结果加在一起，因此唯一可行的选择是放置在 “node4” 上。

多个约束可能导致冲突。假设有一个跨越 2 个区域的 3 节点集群：

+---------------+-------+

| zoneA | zoneB |

+-------+-------+-------+

| node1 | node2 | nod3 |

+-------+-------+-------+

| P P | P | P P |

+-------+-------+-------+

如果对集群应用 “two-constraints.yaml”，会发现 “mypod” 处于 Pending 状态。这是因为：为了满足第一个约束，”mypod” 只能放在 “zoneB” 中，而第二个约束要求 “mypod” 只能放在 “node2” 上。pod 调度无法满足两种约束。

为了克服这种情况，可以增加 maxSkew 或修改其中一个约束，让其使用 whenUnsatisfiable: ScheduleAnyway。

约定

这里有一些值得注意的隐式约定：

* 只有与传入 pod 具有相同命名空间的 pod 才能作为匹配候选者。
* 没有 topologySpreadConstraints[\*].topologyKey 的节点将被忽略。这意味着：
  1. 位于这些节点上的 pod 不影响 maxSkew 的计算。在上面的例子中，假设 “node1” 没有标签 “zone”，那么 2 个 pod 将被忽略，因此传入的 pod 将被调度到 “zoneA” 中。
  2. 传入的 pod 没有机会被调度到这类节点上。在上面的例子中，假设一个带有标签 {zone-typo: zoneC} 的 “node5” 加入到集群，它将由于没有标签键 “zone” 而被忽略。

注意，如果传入 pod 的 topologySpreadConstraints[\*].labelSelector 与自身的标签不匹配，将会发生什么。在上面的例子中，如果移除传入 pod 的标签，pod 仍然可以调度到 “zoneB”，因为约束仍然满足。然而，在调度之后，集群的不平衡程度保持不变。zoneA 仍然有 2 个带有 {foo:bar} 标签的 pod，zoneB 有 1 个带有 {foo:bar} 标签的 pod。因此，如果这不是你所期望的，建议工作负载的 topologySpreadConstraints[\*].labelSelector 与其自身的标签匹配。

* 如果传入的 pod 定义了 spec.nodeSelector 或 spec.affinity.nodeAffinity，则将忽略不匹配的节点。

假设有一个从 zonea 到 zonec 的 5 节点集群：

```

+---------------+---------------+-------+

| zoneA | zoneB | zoneC |

+-------+-------+-------+-------+-------+

| node1 | node2 | node3 | node4 | node5 |

+-------+-------+-------+-------+-------+

| P | P | P | | |

+-------+-------+-------+-------+-------+

```

你知道 “zoneC” 必须被排除在外。在这种情况下，可以按如下方式编写 yaml，以便将 “mypod” 放置在 “zoneB” 上，而不是 “zoneC” 上。同样，spec.nodeSelector 也要一样处理。

| [pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/topology-spread-constraints/one-constraint-with-nodeaffinity.yaml) |
| --- |
| kind: Pod  apiVersion: v1  metadata:  name: mypod  labels:  foo: bar  spec:  topologySpreadConstraints:  - maxSkew: 1  topologyKey: zone  whenUnsatisfiable: DoNotSchedule  labelSelector:  matchLabels:  foo: bar  affinity:  nodeAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  nodeSelectorTerms:  - matchExpressions:  - key: zone  operator: NotIn  values:  - zoneC  containers:  - name: pause  image: k8s.gcr.io/pause:3.1 |

与 PodAffinity/PodAntiAffinity 相比较

在 Kubernetes 中，与 “Affinity” 相关的指令控制 pod 的调度方式（更密集或更分散）。

* 对于 PodAffinity，可以尝试将任意数量的 pod 打包到符合条件的拓扑域中。
* 对于 PodAntiAffinity，只能将一个 pod 调度到单个拓扑域中。

“EvenPodsSpread” 功能提供灵活的选项来将 pod 均匀分布到不同的拓扑域中，以实现高可用性或节省成本。这也有助于滚动更新工作负载和平滑扩展副本。有关详细信息，请参考[动机](https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/20190221-even-pods-spreading.md#motivation)。

已知局限性

1.16 版本（此功能为 alpha）存在下面的一些限制：

* Deployment 的缩容可能导致 pod 分布不平衡。
* pod 匹配到污点节点是允许的。参考 [Issue 80921](https://github.com/kubernetes/kubernetes/issues/80921)。

干扰

本指南针对的是希望构建高可用性应用程序的应用所有者，他们有必要了解可能发生在 pod 上的干扰类型。

文档同样适用于想要执行自动化集群操作（例如升级和自动扩展集群）的集群管理员。

* [**自愿干扰和非自愿干扰**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e8%87%aa%e6%84%bf%e5%b9%b2%e6%89%b0%e5%92%8c%e9%9d%9e%e8%87%aa%e6%84%bf%e5%b9%b2%e6%89%b0)
* [**处理干扰**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e5%a4%84%e7%90%86%e5%b9%b2%e6%89%b0)
* [**干扰预算工作原理**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e5%b9%b2%e6%89%b0%e9%a2%84%e7%ae%97%e5%b7%a5%e4%bd%9c%e5%8e%9f%e7%90%86)
* [**PDB 例子**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#pdb-%e4%be%8b%e5%ad%90)
* [**分离集群所有者和应用所有者角色**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e5%88%86%e7%a6%bb%e9%9b%86%e7%be%a4%e6%89%80%e6%9c%89%e8%80%85%e5%92%8c%e5%ba%94%e7%94%a8%e6%89%80%e6%9c%89%e8%80%85%e8%a7%92%e8%89%b2)
* [**如何在集群上执行干扰操作**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e5%a6%82%e4%bd%95%e5%9c%a8%e9%9b%86%e7%be%a4%e4%b8%8a%e6%89%a7%e8%a1%8c%e5%b9%b2%e6%89%b0%e6%93%8d%e4%bd%9c)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

自愿干扰和非自愿干扰

Pod 不会消失，除非有人（用户或控制器）将其销毁，或者出现了不可避免的硬件或软件系统错误。

我们把这些不可避免的情况称为应用的\*非自愿干扰\*。例如：

* 节点下层物理机的硬件故障
* 集群管理员错误地删除虚拟机（实例）
* 云提供商或虚拟机管理程序中的故障导致的虚拟机消失
* 内核错误
* 节点由于集群网络隔离从集群中消失
* 由于节点[资源不足](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/)导致 pod 被驱逐。

除了资源不足的情况，大多数用户应该都熟悉这些情况；它们不是特定于 Kubernetes 的。

我们称其他情况为\*自愿干扰\*。包括由应用程序所有者发起的操作和由集群管理员发起的操作。典型的应用程序所有者的 作包括：

* 删除 deployment 或其他管理 pod 的控制器
* 更新了 deployment 的 pod 模板导致 pod 重启
* 直接删除 pod（例如，因为误操作）

集群管理员操作包括：

* [排空（drain）节点](https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/)进行修复或升级。
* 从集群中排空节点以缩小集群（了解[集群自动扩缩](https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#cluster-autoscaler)）。
* 从节点中移除一个 pod，以允许其他 pod 使用该节点。

这些操作可能由集群管理员直接执行，也可能由集群管理员所使用的自动化工具执行，或者由集群托管提供商自动执行。

咨询集群管理员或联系云提供商，或者查询发布文档，以确定是否为集群启用了任何资源干扰源。如果没有启用，可以不用创建 Pod Disruption Budgets（Pod 干扰预算）

**警告:**

并非所有的自愿干扰都会受到 pod 干扰预算的限制。例如，删除 deployment 或 pod 的删除操作就会跳过 pod 干扰预算检查。

处理干扰

以下是减轻非自愿干扰的一些方法：

* 确保 pod[请求所需资源](https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-ram-container)。
* 如果需要更高的可用性，请复制应用程序。（了解有关运行多副本的[无状态](https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/)和[有状态](https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/)应用程序的信息。）
* 为了在运行复制应用程序时获得更高的可用性，请跨机架（使用[反亲和性](https://kubernetes.io/docs/user-guide/node-selection/#inter-pod-affinity-and-anti-affinity-beta-feature)）或跨区域（如果使用[多区域集群](https://kubernetes.io/docs/setup/multiple-zones)）扩展应用程序。

自愿干扰的频率各不相同。在一个基本的 Kubernetes 集群中，根本没有自愿干扰。然而，集群管理 或托管提供商可能运行一些可能导致自愿干扰的额外服务。例如，节点软 更新可能导致自愿干扰。另外，集群（节点）自动缩放的某些 现可能导致碎片整理和紧缩节点的自愿干扰。集群 理员或托管提供商应该已经记录了各级别的自愿干扰（如果有的话）。

Kubernetes 提供特性来满足在出现频繁自愿干扰的同时运行高可用的应用程序。我们称这些特性为*干扰预算*

干扰预算工作原理

应用程序所有者可以为每个应用程序创建 PodDisruptionBudget 对象（PDB）。PDB 将限制在同一时间因自愿干扰导致的复制应用程序中宕机的 pod 数量。例如，基于定额的应用程序希望确保运行的副本数 永远不会低于仲裁所需的数量。Web 前端可能希望确保提供负载的副本数量永远不会低于总数的某个百分比。

集群管理员和托管提供商应该使用遵循 Pod Disruption Budgets 的接口（通过调用[驱逐 API](https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/#the-eviction-api)），而不是直接删除 pod 或 deployment。示例包括 kubectl drain 命令和 Kubernetes-on-GCE 集群升级脚本（cluster/gce/upgrade.sh）。

当集群管理员想排空一个节点时，可以使用 kubectl drain 命令。该命令试图驱逐机器上的所有 pod。驱逐请求可能会暂时被拒绝，且该工具定时重试失败的请求直到所有的 pod 都被终止，或者达到配置的超时时间。

PDB 指定应用程序可以容忍的副本数量（相当于应该有多少副本）。例如，具有 .spec.replicas: 5 的 deployment 在任何时间都应该有 5 个 pod。如果 PDB 允许其在某一时刻有 4 个副本，那么驱逐 API 将允许同一时刻仅有一个而不是两个 pod 自愿干扰。

使用标签选择器来指定构成应用程序的一组 pod，这与应用程序的控制器（deployment，stateful-set 等）选择 pod 的逻辑一样。

Pod 控制器的 .spec.replicas 计算“预期的” pod 数量。根据 pod 对象的 .metadata.ownerReferences 字段来发现控制器。

PDB 不能阻止[非自愿干扰](https://kubernetes.io/zh/docs/concepts/workloads/pods/disruptions/#voluntary-and-involuntary-disruptions)的发生，但是确实会计入 算。

由于应用程序的滚动升级而被删除或不可用的 pod 确实会计入干扰预算，但是控制器（如 deployment 和 stateful-set）在进行滚动升级时不受 PDB 的限制。应用程序更新期间的故障处理是在控制器的 spec 中配置的。（了解[更新 deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment)。）

当使用驱逐 API 驱逐 pod 时，pod 会被优雅地终止（参考 [PodSpec](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#podspec-v1-core) 中的 terminationGracePeriodSeconds）。

PDB 例子

假设集群有 3 个节点，node-1 到 node-3。集群上运行了一些应用。其中一个应用有 3 个副本，分别是 pod-a，pod-b 和 pod-c。另外，还有一个不带 PDB 的无关 pod pod-x 也同样显示。最初，所有的 pod 分布如下：

| node-1 | node-2 | node-3 |
| --- | --- | --- |
| pod-a *available* | pod-b *available* | pod-c *available* |
| pod-x *available* |  |  |

3 个 pod 都是 deployment 的一部分，并且共同拥有同一个 PDB，要求 3 个 pod 中至少有 2 个 pod 始终处于可用状态。

例如，假设集群管理员想要重启系统，升级内核版本来修复内核中的 bug。集群管理员首先使用 kubectl drain 命令尝试排空 node-1 节点。命令尝试驱逐 pod-a 和 pod-x。操作立即就成功了。两个 pod 同时进入 terminating 状态。这时的集群处于下面的状态：

| node-1 *draining* | node-2 | node-3 |
| --- | --- | --- |
| pod-a *terminating* | pod-b *available* | pod-c *available* |
| pod-x *terminating* |  |  |

Deployment 控制器观察到其中一个 pod 正在终止，因此它创建了一个替代 pod pod-d。由于 node-1 被封锁（cordon），pod-d 落在另一个节点上。同样其他控制器也创建了 pod-y 作为 pod-x 的替代品。

（注意：对于 StatefulSet 来说，pod-a（也称为 pod-0）需要在替换 pod 创建之前完全终止，替代它的也称为 pod-0，但是具有不同的 UID。反之，样例也适用于 StatefulSet。）

当前集群的状态如下：

| node-1 *draining* | node-2 | node-3 |
| --- | --- | --- |
| pod-a *terminating* | pod-b *available* | pod-c *available* |
| pod-x *terminating* | pod-d *starting* | pod-y |

在某一时刻，pod 被终止，集群如下所示：

| node-1 *drained* | node-2 | node-3 |
| --- | --- | --- |
|  | pod-b *available* | pod-c *available* |
|  | pod-d *starting* | pod-y |

此时，如果一个急躁的集群管理员试图排空（drain）node-2 或 node-3，drain 命令将被阻塞，因为对于 deployment 来说只有 2 个可用的 pod，并且它的 PDB 至少需要 2 个。经过一段时间，pod-d 变得可用。

集群状态如下所示：

| node-1 *drained* | node-2 | node-3 |
| --- | --- | --- |
|  | pod-b *available* | pod-c *available* |
|  | pod-d *available* | pod-y |

现在，集群管理员试图排空（drain）node-2。drain 命令将尝试按照某种顺序驱逐两个 pod，假设先是 pod-b，然后是 pod-d。命令成功驱逐 pod-b，但是当它尝试驱逐 pod-d时将被拒绝，因为对于 deployment 来说只剩一个可用的 pod 了。

Deployment 创建 pod-b 的替代 pod pod-e。因为集群中没有足够的资源来调度 pod-e，drain 命令再次阻塞。集群最终将是下面这种状态：

| node-1 *drained* | node-2 | node-3 | *no node* |
| --- | --- | --- | --- |
|  | pod-b *available* | pod-c *available* | pod-e *pending* |
|  | pod-d *available* | pod-y |  |

此时，集群管理员需要增加一个节点到集群中以继续升级操作。

可以看到 Kubernetes 如何改变干扰发生的速率，根据：

* 应用程序需要多少个副本
* 优雅关闭应用实例需要多长时间
* 启动应用新实例需要多长时间
* 控制器的类型
* 集群的资源能力

分离集群所有者和应用所有者角色

通常，将集群管理者和应用所有者视为彼此了解有限的独立角色是很有用的。这种责任分离在下面这些场景下是有意义的：

* 当有许多应用程序团队共用一个 Kubernetes 集群，并且有自然的专业角色
* 当第三方工具或服务用于集群自动化管理

Pod 干扰预算通过在角色之间提供接口来支持这种分离。

如果你的组织中没有这样的责任分离，则可能不需要使用 Pod 干扰预算。

如何在集群上执行干扰操作

如果你是集群管理员，并且需要对集群中的所有节点执行干扰操作，例如节点或系统软件升级，则可以使用以下选项

* 接受升级期间的停机时间。
* 故障转移到另一个完整的副本集群。
  + 没有停机时间，但是对于重复的节点和人工协调成本可能是昂贵的。
* 编写可容忍干扰的应用程序和使用 PDB。
  + 没有停机 间。
  + 最小的资源重复。
  + 允许更多的集群管理自动化。
  + 编写可容忍干扰的应用程序是棘手的，但对于支持容忍自愿干扰所做的工作，和支持自动扩缩和容忍非 愿干扰所做工作相比，有大量的重叠

临时容器

**FEATURE STATE:** Kubernetes v1.17 [alpha](https://kubernetes.io/zh/docs/concepts/workloads/pods/ephemeral-containers/)

此页面概述了临时容器：一种特殊的容器，该容器在现有 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 中临时运行，为了完成用户启动的操作，例如故障排查。使用临时容器来检查服务，而不是构建应用程序。

**警告：** 临时容器处于早期的 alpha 阶段，不适用于生成环境集群。应该预料到临时容器在某些情况下不起作用，例如在定位容器的名称命名空间。根据 [Kubernetes 弃用政策](https://kubernetes.io/docs/reference/using-api/deprecation-policy/)，该 alpha 功能将来可能发生重大变化或完全删除。

* [**了解临时容器**](https://kubernetes.io/zh/docs/concepts/workloads/pods/ephemeral-containers/#%e4%ba%86%e8%a7%a3%e4%b8%b4%e6%97%b6%e5%ae%b9%e5%99%a8)
* [**临时容器的用途**](https://kubernetes.io/zh/docs/concepts/workloads/pods/ephemeral-containers/#%e4%b8%b4%e6%97%b6%e5%ae%b9%e5%99%a8%e7%9a%84%e7%94%a8%e9%80%94)

了解临时容器

[Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 是 Kubernetes 应用程序的基本构建块。由于 pod 是一次性且可替换的，因此一旦 Pod 创建，就无法将容器加入到 Pod 中。取而代之的是，通常使用 [deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 以受控的方式来删除并替换 Pod。

有时有必要检查现有 Pod 的状态，例如，对于难以复现的故障进行排查。在这些场景中，可以在现有 Pod 中运行临时容器来检查其状态并运行任意命令。

什么是临时容器？

临时容器与其他容器的不同之处在于，它们缺少对资源或执行的保证，并且永远不会自动重启，因此不适用于构建应用程序。临时容器使用与常规容器相同的 ContainerSpec 段进行描述，但许多字段是不相容且不允许的。

* 临时容器没有端口配置，因此像 ports，livenessProbe，readinessProbe 这样的字段是不允许的。
* Pod 资源分配是不可变的，因此 resources 配置是不允许的。
* 有关允许字段的完整列表，请参见[临时容器参考文档](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#ephemeralcontainer-v1-core)。

临时容器是使用 API 中的一种特殊的 ephemeralcontainers 处理器进行创建的，而不是直接添加到 pod.spec 段，因此无法使用 kubectl edit 来添加一个临时容器。

与常规容器一样，将临时容器添加到 Pod 后，将不能更改或删除临时容器。

临时容器的用途

当容器崩溃或容器镜像中不包含调试工具时，使用 kubectl exec 命令进行故障排查不足，临时容器对于交互式故障处理很有用。

尤其是，[distroless 镜像](https://github.com/GoogleContainerTools/distroless)能够使得部署最小的容器镜像，从而减少攻击面并减少故障和漏洞的暴露。由于 distroless 镜像不包含 shell 或任何的调试工具，因此很难单独使用 kubectl exec 命令进行故障排查。

使用临时容器时，启用[进程命名空间共享](https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/)很有帮助，可以查看其他容器中的进程。

例子

**注意：** 本节中的示例要求启用 EphemeralContainers [特性](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/)，并且 kubernetes 客户端和服务端版本要求为 v1.16 或更高版本。

本节中的示例演示了临时容器如何在 API 中出现。用户通常会使用 kubectl 插件来进行故障排查，从而自动化执行这些步骤。

临时容器是使用 Pod 的 ephemeralcontainers 子资源创建的，可以使用 kubectl --raw 命令进行显示。首先描述临时容器被添加为一个 EphemeralContainers 列表：

{

**"apiVersion"**: "v1",

**"kind"**: "EphemeralContainers",

**"metadata"**: {

**"name"**: "example-pod"

},

**"ephemeralContainers"**: [{

**"command"**: [

"sh"

],

**"image"**: "busybox",

**"imagePullPolicy"**: "IfNotPresent",

**"name"**: "debugger",

**"stdin"**: **true**,

**"tty"**: **true**,

**"terminationMessagePolicy"**: "File"

}]

}

使用如下命令更新已运行的临时容器 example-pod：

kubectl replace --raw /api/v1/namespaces/default/pods/example-pod/ephemeralcontainers -f ec.json

这将返回临时容器的新列表：

{

**"kind"**:"EphemeralContainers",

**"apiVersion"**:"v1",

**"metadata"**:{

**"name"**:"example-pod",

**"namespace"**:"default",

**"selfLink"**:"/api/v1/namespaces/default/pods/example-pod/ephemeralcontainers",

**"uid"**:"a14a6d9b-62f2-4119-9d8e-e2ed6bc3a47c",

**"resourceVersion"**:"15886",

**"creationTimestamp"**:"2019-08-29T06:41:42Z"

},

**"ephemeralContainers"**:[

{

**"name"**:"debugger",

**"image"**:"busybox",

**"command"**:[

"sh"

],

**"resources"**:{

},

**"terminationMessagePolicy"**:"File",

**"imagePullPolicy"**:"IfNotPresent",

**"stdin"**:**true**,

**"tty"**:**true**

}

]

}

可以使用以下命令查看新创建的临时容器的状态：

kubectl describe pod example-pod

...

Ephemeral Containers:

debugger:

Container ID: docker://cf81908f149e7e9213d3c3644eda55c72efaff67652a2685c1146f0ce151e80f

Image: busybox

Image ID: docker-pullable://busybox@sha256:9f1003c480699be56815db0f8146ad2e22efea85129b5b5983d0e0fb52d9ab70

Port: <none>

Host Port: <none>

Command:

sh

State: Running

Started: Thu, 29 Aug 2019 06:42:21 +0000

Ready: False

Restart Count: 0

Environment: <none>

Mounts: <none>

...

可以使用以下命令连接到新的临时容器：

kubectl attach -it example-pod -c debugger

如果启用了进程命名空间共享，则可以从该 Pod 的所有容器中查看进程。如下所示：

/ *# ps auxww*

PID USER TIME COMMAND

1 root 0:00 /pause

6 root 0:00 nginx: master process nginx -g daemon off;

11 101 0:00 nginx: worker process

12 101 0:00 nginx: worker process

13 101 0:00 nginx: worker process

14 101 0:00 nginx: worker process

15 101 0:00 nginx: worker process

16 101 0:00 nginx: worker process

17 101 0:00 nginx: worker process

18 101 0:00 nginx: worker process

19 root 0:00 /pause

24 root 0:00 sh

29 root 0:00 ps auxww

反馈

Init 容器

本页提供了 Init 容器的概览，它是一种专用的容器，在[Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/)内的应用容器启动之前运行，并包括一些应用镜像中不存在的实用工具和安装脚本。

* [**理解 Init 容器**](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#%e7%90%86%e8%a7%a3-init-%e5%ae%b9%e5%99%a8)
* [**Init 容器能做什么？**](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#init-%e5%ae%b9%e5%99%a8%e8%83%bd%e5%81%9a%e4%bb%80%e4%b9%88)
* [**具体行为**](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#%e5%85%b7%e4%bd%93%e8%a1%8c%e4%b8%ba)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

理解 Init 容器

[Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 可以包含多个容器，应用运行在这些容器里面，同时 Pod 也可以有一个或多个先于应用容器启动的 Init 容器。

Init 容器与普通的容器非常像，除了如下两点：

* 它们总是运行到完成。
* 每个都必须在下一个启动之前成功完成。

如果 Pod 的 Init 容器失败，Kubernetes 会不断地重启该 Pod，直到 Init 容器成功为止。然而，如果 Pod 对应的 restartPolicy 值为 Never，它不会重新启动。

指定容器为 Init 容器，需要在 Pod 的 spec 中添加 initContainers 字段， 该字段內以[Container](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#container-v1-core) 类型对象数组的形式组织，和应用的 containers 数组同级相邻。 Init 容器的状态在 status.initContainerStatuses 字段中以容器状态数组的格式返回（类似 status.containerStatuses 字段）。

与普通容器的不同之处

Init 容器支持应用容器的全部字段和特性，包括资源限制、数据卷和安全设置。 然而，Init 容器对资源请求和限制的处理稍有不同，在下面 [资源](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#资源) 处有说明。

同时 Init 容器不支持 Readiness Probe，因为它们必须在 Pod 就绪之前运行完成。

如果为一个 Pod 指定了多个 Init 容器，这些容器会按顺序逐个运行。每个 Init 容器必须运行成功，下一个才能够运行。当所有的 Init 容器运行完成时，Kubernetes 才会为 Pod 初始化应用容器并像平常一样运行。

Init 容器能做什么？

因为 Init 容器具有与应用容器分离的单独镜像，其启动相关代码具有如下优势：

* Init 容器可以包含一些安装过程中应用容器中不存在的实用工具或个性化代码。例如，没有必要仅为了在安装过程中使用类似 sed、 awk、 python 或 dig 这样的工具而去FROM 一个镜像来生成一个新的镜像。
* Init 容器可以安全地运行这些工具，避免这些工具导致应用镜像的安全性降低。
* 应用镜像的创建者和部署者可以各自独立工作，而没有必要联合构建一个单独的应用镜像。
* Init 容器能以不同于Pod内应用容器的文件系统视图运行。因此，Init容器可具有访问 [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/) 的权限，而应用容器不能够访问。
* 由于 Init 容器必须在应用容器启动之前运行完成，因此 Init 容器提供了一种机制来阻塞或延迟应用容器的启动，直到满足了一组先决条件。一旦前置条件满足，Pod内的所有的应用容器会并行启动。

示例

下面是一些如何使用 Init 容器的想法：

* 等待一个 Service 完成创建，通过类似如下 shell 命令：
* for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; exit 1
* 注册这个 Pod 到远程服务器，通过在命令中调用 API，类似如下：
* curl -X POST http://$MANAGEMENT\_SERVICE\_HOST:$MANAGEMENT\_SERVICE\_PORT/register -d 'instance=$(<POD\_NAME>)&ip=$(<POD\_IP>)'
* 在启动应用容器之前等一段时间，使用类似命令：
* `sleep 60`
* 克隆 Git 仓库到 [Volume](https://kubernetes.io/docs/concepts/storage/volumes/)。
* 将配置值放到配置文件中，运行模板工具为主应用容器动态地生成配置文件。例如，在配置文件中存放 POD\_IP 值，并使用 Jinja 生成主应用配置文件。

使用 Init 容器

下面的例子定义了一个具有 2 个 Init 容器的简单 Pod。 第一个等待 myservice 启动，第二个等待 mydb 启动。 一旦这两个 Init容器 都启动完成，Pod 将启动spec区域中的应用容器。

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: busybox:1.28

command: ['sh', '-c', 'echo The app is running! && sleep 3600']

initContainers:

- name: init-myservice

image: busybox:1.28

command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']

- name: init-mydb

image: busybox:1.28

command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']

下面的 yaml 文件展示了 mydb 和 myservice 两个 Service：

kind: Service

apiVersion: v1

metadata:

name: myservice

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

---

kind: Service

apiVersion: v1

metadata:

name: mydb

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9377

要启动这个 Pod，可以执行如下命令：

kubectl apply -f myapp.yaml

pod/myapp-pod created

要检查其状态：

kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

myapp-pod 0/1 Init:0/2 0 6m

如需更详细的信息：

kubectl describe -f myapp.yaml

Name: myapp-pod

Namespace: default

[...]

Labels: app=myapp

Status: Pending

[...]

Init Containers:

init-myservice:

[...]

State: Running

[...]

init-mydb:

[...]

State: Waiting

Reason: PodInitializing

Ready: False

[...]

Containers:

myapp-container:

[...]

State: Waiting

Reason: PodInitializing

Ready: False

[...]

Events:

FirstSeen LastSeen Count From SubObjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

16s 16s 1 {default-scheduler } Normal Scheduled Successfully assigned myapp-pod to 172.17.4.201

16s 16s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Pulling pulling image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Pulled Successfully pulled image "busybox"

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Created Created container with docker id 5ced34a04634; Security:[seccomp=unconfined]

13s 13s 1 {kubelet 172.17.4.201} spec.initContainers{init-myservice} Normal Started Started container with docker id 5ced34a04634

如需查看Pod内 Init 容器的日志，请执行:

$ kubectl logs myapp-pod -c init-myservice # Inspect the first init container

$ kubectl logs myapp-pod -c init-mydb # Inspect the second init container

在这一刻，Init 容器将会等待至发现名称为mydb和myservice的 Service。

如下为创建这些 Service 的配置文件：

---

apiVersion: v1

kind: Service

metadata:

name: myservice

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

---

apiVersion: v1

kind: Service

metadata:

name: mydb

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9377

创建mydb和myservice的 service 命令：

$ kubectl create -f services.yaml

service "myservice" created

service "mydb" created

这样你将能看到这些 Init容器 执行完毕，随后my-app的Pod转移进入 Running 状态：

$ kubectl get -f myapp.yaml

NAME READY STATUS RESTARTS AGE

myapp-pod 1/1 Running 0 9m

一旦我们启动了 mydb 和 myservice 这两个 Service，我们能够看到 Init 容器完成，并且 myapp-pod 被创建：

这个简单的例子应该能为你创建自己的 Init 容器提供一些启发。 [What’s next](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#what-s-next) 部分提供了更详细例子的链接。

具体行为

在 Pod 启动过程中，每个Init 容器在网络和数据卷初始化之后会按顺序启动。每个 Init容器 成功退出后才会启动下一个 Init容器。 如果因为运行或退出时失败引发容器启动失败，它会根据 Pod 的 restartPolicy 策略进行重试。 然而，如果 Pod 的 restartPolicy 设置为 Always，Init 容器失败时会使用 restartPolicy 的 OnFailure 策略。

在所有的 Init 容器没有成功之前，Pod 将不会变成 Ready 状态。 Init 容器的端口将不会在 Service 中进行聚集。 正在初始化中的 Pod 处于 Pending 状态，但会将条件 Initializing 设置为 true。

如果 Pod [重启](https://kubernetes.io/zh/docs/concepts/workloads/pods/init-containers/#pod-restart-reasons)，所有 Init 容器必须重新执行。

对 Init 容器 spec 的修改仅限于容器的 image 字段。 更改 Init 容器的 image 字段，等同于重启该 Pod。

因为 Init 容器可能会被重启、重试或者重新执行，所以 Init 容器的代码应该是幂等的。 特别地，基于 EmptyDirs 写文件的代码，应该对输出文件可能已经存在做好准备。

Init 容器具有应用容器的所有字段。 然而 Kubernetes 禁止使用 readinessProbe，因为 Init 容器不能定义不同于完成（completion）的就绪（readiness）。 这一点会在校验时强制执行。

在 Pod 上使用 activeDeadlineSeconds和在容器上使用 livenessProbe 可以避免 Init 容器一直重复失败。 activeDeadlineSeconds 时间包含了 Init 容器启动的时间。

在 Pod 中的每个应用容器和 Init 容器的名称必须唯一；与任何其它容器共享同一个名称，会在校验时抛出错误。

资源

给定Init 容器的执行顺序下，资源使用适用于如下规则：

* 所有 Init 容器上定义的任何特定资源的 limit 或 request 的最大值，作为 Pod *有效初始 request/limit*
* Pod 对资源的 \*有效 limit/request \* 是如下两者的较大者：
  + 所有应用容器对某个资源的 limit/request 之和
  + 对某个资源的有效初始 limit/request
* 基于有效 limit/request 完成调度，这意味着 Init 容器能够为初始化过程预留资源，这些资源在 Pod 生命周期过程中并没有被使用。
* Pod 的 *有效 QoS 层* ，与 Init 容器和应用容器的一样。

配额和限制适用于有效 Pod的 limit/request。 Pod 级别的 cgroups 是基于有效 Pod 的 limit/request，和调度器相同。

Pod 重启的原因

Pod重启导致 Init 容器重新执行，主要有如下几个原因：

* 用户更新 Pod 的 Spec 导致 Init 容器镜像发生改变。Init 容器镜像的变更会引起 Pod 重启. 应用容器镜像的变更仅会重启应用容器。
* Pod 的基础设施容器 (译者注：如 pause 容器) 被重启。 这种情况不多见，必须由具备 root 权限访问 Node 的人员来完成。
* 当 restartPolicy 设置为 Always，Pod 中所有容器会终止而强制重启，由于垃圾收集导致 Init 容器的完成记录丢失。

你可以在Pod的规格信息中与containers数组同级的位置指定 Init 容器。

反馈

Pod 的生命周期

该页面将描述 Pod 的生命周期。

* [**Pod phase**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#pod-phase)
* [**Pod 状态**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#pod-%e7%8a%b6%e6%80%81)
* [**容器探针**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#%e5%ae%b9%e5%99%a8%e6%8e%a2%e9%92%88)
* [**Pod 和容器状态**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#pod-%e5%92%8c%e5%ae%b9%e5%99%a8%e7%8a%b6%e6%80%81)
* [**重启策略**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#%e9%87%8d%e5%90%af%e7%ad%96%e7%95%a5)
* [**Pod 的生命**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#pod-%e7%9a%84%e7%94%9f%e5%91%bd)
* [**示例**](https://kubernetes.io/zh/docs/concepts/workloads/pods/pod-lifecycle/#%e7%a4%ba%e4%be%8b)

Pod phase

Pod 的 status 定义在 [PodStatus](https://kubernetes.io/docs/resources-reference/v1.7/#podstatus-v1-core) 对象中，其中有一个 phase 字段。

Pod 的运行阶段（phase）是 Pod 在其生命周期中的简单宏观概述。该阶段并不是对容器或 Pod 的综合汇总，也不是为了做为综合状态机。

Pod 相位的数量和含义是严格指定的。除了本文档中列举的内容外，不应该再假定 Pod 有其他的 phase 值。

下面是 phase 可能的值：

* 挂起（Pending）：Pod 已被 Kubernetes 系统接受，但有一个或者多个容器镜像尚未创建。等待时间包括调度 Pod 的时间和通过网络下载镜像的时间，这可能需要花点时间。
* 运行中（Running）：该 Pod 已经绑定到了一个节点上，Pod 中所有的容器都已被创建。至少有一个容器正在运行，或者正处于启动或重启状态。
* 成功（Succeeded）：Pod 中的所有容器都被成功终止，并且不会再重启。
* 失败（Failed）：Pod 中的所有容器都已终止了，并且至少有一个容器是因为失败终止。也就是说，容器以非0状态退出或者被系统终止。
* 未知（Unknown）：因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败。

Pod 状态

Pod 有一个 PodStatus 对象，其中包含一个 [PodCondition](https://kubernetes.io/docs/resources-reference/v1.7/#podcondition-v1-core) 数组。 PodCondition 数组的每个元素都有一个 type 字段和一个 status 字段。type 字段是字符串，可能的值有 PodScheduled、Ready、Initialized 和 Unschedulable。status 字段是一个字符串，可能的值有 True、False 和 Unknown。

容器探针

[探针](https://kubernetes.io/docs/resources-reference/v1.7/#probe-v1-core) 是由 [kubelet](https://kubernetes.io/docs/admin/kubelet/) 对容器执行的定期诊断。要执行诊断，kubelet 调用由容器实现的 [Handler](https://godoc.org/k8s.io/kubernetes/pkg/api/v1#Handler)。有三种类型的处理程序：

* [ExecAction](https://kubernetes.io/docs/resources-reference/v1.7/#execaction-v1-core)：在容器内执行指定命令。如果命令退出时返回码为 0 则认为诊断成功。
* [TCPSocketAction](https://kubernetes.io/docs/resources-reference/v1.7/#tcpsocketaction-v1-core)：对指定端口上的容器的 IP 地址进行 TCP 检查。如果端口打开，则诊断被认为是成功的。
* [HTTPGetAction](https://kubernetes.io/docs/resources-reference/v1.7/#httpgetaction-v1-core)：对指定的端口和路径上的容器的 IP 地址执行 HTTP Get 请求。如果响应的状态码大于等于200 且小于 400，则诊断被认为是成功的。

每次探测都将获得以下三种结果之一：

* 成功：容器通过了诊断。
* 失败：容器未通过诊断。
* 未知：诊断失败，因此不会采取任何行动。

Kubelet 可以选择是否执行在容器上运行的两种探针执行和做出反应：

* livenessProbe：指示容器是否正在运行。如果存活探测失败，则 kubelet 会杀死容器，并且容器将受到其 [重启策略](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) 的影响。如果容器不提供存活探针，则默认状态为 Success。
* readinessProbe：指示容器是否准备好服务请求。如果就绪探测失败，端点控制器将从与 Pod 匹配的所有 Service 的端点中删除该 Pod 的 IP 地址。初始延迟之前的就绪状态默认为 Failure。如果容器不提供就绪探针，则默认状态为 Success。

该什么时候使用存活（liveness）和就绪（readiness）探针?

如果容器中的进程能够在遇到问题或不健康的情况下自行崩溃，则不一定需要存活探针; kubelet 将根据 Pod 的restartPolicy 自动执行正确的操作。

如果您希望容器在探测失败时被杀死并重新启动，那么请指定一个存活探针，并指定restartPolicy 为 Always 或 OnFailure。

如果要仅在探测成功时才开始向 Pod 发送流量，请指定就绪探针。在这种情况下，就绪探针可能与存活探针相同，但是 spec 中的就绪探针的存在意味着 Pod 将在没有接收到任何流量的情况下启动，并且只有在探针探测成功后才开始接收流量。

如果您希望容器能够自行维护，您可以指定一个就绪探针，该探针检查与存活探针不同的端点。

请注意，如果您只想在 Pod 被删除时能够排除请求，则不一定需要使用就绪探针；在删除 Pod 时，Pod 会自动将自身置于未完成状态，无论就绪探针是否存在。当等待 Pod 中的容器停止时，Pod 仍处于未完成状态。

Pod 和容器状态

有关 Pod 容器状态的详细信息，请参阅 [PodStatus](https://kubernetes.io/docs/resources-reference/v1.7/#podstatus-v1-core) 和 [ContainerStatus](https://kubernetes.io/docs/resources-reference/v1.7/#containerstatus-v1-core)。请注意，报告的 Pod 状态信息取决于当前的 [ContainerState](https://kubernetes.io/docs/resources-reference/v1.7/#containerstatus-v1-core)。

重启策略

PodSpec 中有一个 restartPolicy 字段，可能的值为 Always、OnFailure 和 Never。默认为 Always。 restartPolicy 适用于 Pod 中的所有容器。restartPolicy 仅指通过同一节点上的 kubelet 重新启动容器。失败的容器由 kubelet 以五分钟为上限的指数退避延迟（10秒，20秒，40秒…）重新启动，并在成功执行十分钟后重置。如 [Pod 文档](https://kubernetes.io/docs/user-guide/pods/#durability-of-pods-or-lack-thereof) 中所述，一旦绑定到一个节点，Pod 将永远不会重新绑定到另一个节点。

Pod 的生命

一般来说，Pod 不会消失，直到人为销毁他们。这可能是一个人或控制器。这个规则的唯一例外是成功或失败的 phase 超过一段时间（由 master 确定）的Pod将过期并被自动销毁。

有三种可用的控制器：

* 使用 [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) 运行预期会终止的 Pod，例如批量计算。Job 仅适用于重启策略为 OnFailure 或 Never 的 Pod。
* 对预期不会终止的 Pod 使用 [ReplicationController](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/)、[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 和 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) ，例如 Web 服务器。 ReplicationController 仅适用于具有 restartPolicy 为 Always 的 Pod。
* 提供特定于机器的系统服务，使用 [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) 为每台机器运行一个 Pod 。

所有这三种类型的控制器都包含一个 PodTemplate。建议创建适当的控制器，让它们来创建 Pod，而不是直接自己创建 Pod。这是因为单独的 Pod 在机器故障的情况下没有办法自动复原，而控制器却可以。

如果节点死亡或与集群的其余部分断开连接，则 Kubernetes 将应用一个策略将丢失节点上的所有 Pod 的 phase 设置为 Failed。

示例

高级 liveness 探针示例

存活探针由 kubelet 来执行，因此所有的请求都在 kubelet 的网络命名空间中进行。

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-http

spec:

containers:

- args:

- /server

image: k8s.gcr.io/liveness

livenessProbe:

httpGet:

*# 当没有定义 "host" 时，使用 "PodIP"*

*# host: my-host*

*# 当没有定义 "scheme" 时，使用 "HTTP" scheme 只允许 "HTTP" 和 "HTTPS"*

*# scheme: HTTPS*

path: /healthz

port: 8080

httpHeaders:

- name: X-Custom-Header

value: Awesome

initialDelaySeconds: 15

timeoutSeconds: 1

name: liveness

状态示例

* Pod 中只有一个容器并且正在运行。容器成功退出。
  + 记录完成事件。
  + 如果 restartPolicy 为：
  + Always：重启容器；Pod phase 仍为 Running。
  + OnFailure：Pod phase 变成 Succeeded。
  + Never：Pod phase 变成 Succeeded。
* Pod 中只有一个容器并且正在运行。容器退出失败。
  + 记录失败事件。
  + 如果 restartPolicy 为：
  + Always：重启容器；Pod phase 仍为 Running。
  + OnFailure：重启容器；Pod phase 仍为 Running。
  + Never：Pod phase 变成 Failed。
* Pod 中有两个容器并且正在运行。有一个容器退出失败。
  + 记录失败事件。
  + 如果 restartPolicy 为：
  + Always：重启容器；Pod phase 仍为 Running。
  + OnFailure：重启容器；Pod phase 仍为 Running。
  + Never：不重启容器；Pod phase 仍为 Running。
  + 如果有一个容器没有处于运行状态，并且两个容器退出：
  + 记录失败事件。
  + 如果 restartPolicy 为：
    - Always：重启容器；Pod phase 仍为 Running。
    - OnFailure：重启容器；Pod phase 仍为 Running。
    - Never：Pod phase 变成 Failed。
* Pod 中只有一个容器并处于运行状态。容器运行时内存超出限制：
  + 容器以失败状态终止。
  + 记录 OOM 事件。
  + 如果 restartPolicy 为：
  + Always：重启容器；Pod phase 仍为 Running。
  + OnFailure：重启容器；Pod phase 仍为 Running。
  + Never: 记录失败事件；Pod phase 仍为 Failed。
* Pod 正在运行，磁盘故障：
  + 杀掉所有容器。
  + 记录适当事件。
  + Pod phase 变成 Failed。
  + 如果使用控制器来运行，Pod 将在别处重建。
* Pod 正在运行，其节点被分段。
  + 节点控制器等待直到超时。
  + 节点控制器将 Pod phase 设置为 Failed。
  + 如果是用控制器来运行，Pod 将在别处重建。

反馈

控制器

ReplicaSet

ReplicaSet 是下一代的 Replication Controller。 *ReplicaSet* 和 [*Replication Controller*](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/) 的唯一区别是选择器的支持。ReplicaSet 支持新的基于集合的选择器需求，这在[标签用户指南](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)中有描述。而 Replication Controller 仅支持基于相等选择器的需求。

* [**怎样使用 ReplicaSet**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#%e6%80%8e%e6%a0%b7%e4%bd%bf%e7%94%a8-replicaset)
* [**什么时候使用 ReplicaSet**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#%e4%bb%80%e4%b9%88%e6%97%b6%e5%80%99%e4%bd%bf%e7%94%a8-replicaset)
* [**示例**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#%e7%a4%ba%e4%be%8b)
* [**编写 ReplicaSet Spec**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#%e7%bc%96%e5%86%99-replicaset-spec)
* [**使用 ReplicaSets 的具体方法**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#%e4%bd%bf%e7%94%a8-replicasets-%e7%9a%84%e5%85%b7%e4%bd%93%e6%96%b9%e6%b3%95)
* [**ReplicaSet 的替代方案**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#replicaset-%e7%9a%84%e6%9b%bf%e4%bb%a3%e6%96%b9%e6%a1%88)

怎样使用 ReplicaSet

大多数支持 Replication Controllers 的[kubectl](https://kubernetes.io/docs/user-guide/kubectl/)命令也支持 ReplicaSets。但[rolling-update](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#rolling-update) 命令是个例外。如果您想要滚动更新功能请考虑使用 Deployment。[rolling-update](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#rolling-update) 命令是必需的，而 Deployment 是声明性的，因此我们建议通过 [rollout](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#rollout)命令使用 Deployment。

虽然 ReplicaSets 可以独立使用，但今天它主要被[Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 用作协调 Pod 创建、删除和更新的机制。 当您使用 Deployment 时，您不必担心还要管理它们创建的 ReplicaSet。Deployment 会拥有并管理它们的 ReplicaSet。

什么时候使用 ReplicaSet

ReplicaSet 确保任何时间都有指定数量的 Pod 副本在运行。 然而，Deployment 是一个更高级的概念，它管理 ReplicaSet，并向 Pod 提供声明式的更新以及许多其他有用的功能。 因此，我们建议使用 Deployment 而不是直接使用 ReplicaSet，除非您需要自定义更新业务流程或根本不需要更新。

这实际上意味着，您可能永远不需要操作 ReplicaSet 对象：而是使用 Deployment，并在 spec 部分定义您的应用。

示例

| [controllers/frontend.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/frontend.yaml) |
| --- |
| apiVersion: apps/v1  kind: ReplicaSet  metadata:  name: frontend  labels:  app: guestbook  tier: frontend  spec:  *# modify replicas according to your case*  replicas: 3  selector:  matchLabels:  tier: frontend  matchExpressions:  - {key: tier, operator: In, values: [frontend]}  template:  metadata:  labels:  app: guestbook  tier: frontend  spec:  containers:  - name: php-redis  image: gcr.io/google\_samples/gb-frontend:v3  resources:  requests:  cpu: 100m  memory: 100Mi  env:  - name: GET\_HOSTS\_FROM  value: dns  *# If your cluster config does not include a dns service, then to*  *# instead access environment variables to find service host*  *# info, comment out the 'value: dns' line above, and uncomment the*  *# line below.*  *# value: env*  ports:  - containerPort: 80 |

将此清单保存到 frontend.yaml 中，并将其提交到 Kubernetes 集群，应该就能创建 yaml 文件所定义的 ReplicaSet 及其管理的 Pod。

$ kubectl create -f http://k8s.io/examples/controllers/frontend.yaml

replicaset.apps/frontend created

$ kubectl describe rs/frontend

Name: frontend

Namespace: default

Selector: tier=frontend,tier in (frontend)

Labels: app=guestbook

tier=frontend

Annotations: <none>

Replicas: 3 current / 3 desired

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

Pod Template:

Labels: app=guestbook

tier=frontend

Containers:

php-redis:

Image: gcr.io/google\_samples/gb-frontend:v3

Port: 80/TCP

Requests:

cpu: 100m

memory: 100Mi

Environment:

GET\_HOSTS\_FROM: dns

Mounts: <none>

Volumes: <none>

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-qhloh

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-dnjpy

1m 1m 1 {replicaset-controller } Normal SuccessfulCreate Created pod: frontend-9si5l

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

frontend-9si5l 1/1 Running 0 1m

frontend-dnjpy 1/1 Running 0 1m

frontend-qhloh 1/1 Running 0 1m

编写 ReplicaSet Spec

与所有其他 Kubernetes API 对象一样，ReplicaSet 也需要 apiVersion、kind、和 metadata 字段。有关使用清单的一般信息，请参见 [使用 kubectl 管理对象](https://kubernetes.io/docs/concepts/overview/object-management-kubectl/overview/)。

ReplicaSet 也需要 [.spec](https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status) 部分。

Pod 模版

.spec.template 是 .spec 唯一需要的字段。.spec.template 是 [Pod 模版](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates)。它和 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/) 的语法几乎完全一样，除了它是嵌套的并没有 apiVersion 和 kind。

除了所需的 Pod 字段之外，ReplicaSet 中的 Pod 模板必须指定适当的标签和适当的重启策略。

对于标签，请确保不要与其他控制器重叠。更多信息请参考 [Pod 选择器](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#pod-selector)。

对于 [重启策略](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy)，.spec.template.spec.restartPolicy 唯一允许的取值是 Always，这也是默认值.

对于本地容器重新启动，ReplicaSet 委托给了节点上的代理去执行，例如[Kubelet](https://kubernetes.io/docs/admin/kubelet/) 或 Docker 去执行。

Pod 选择器

.spec.selector 字段是[标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)。ReplicaSet 管理所有标签匹配与标签选择器的 Pod。它不区分自己创建或删除的 Pod 和其他人或进程创建或删除的pod。这允许在不影响运行中的 Pod 的情况下替换副本集。

.spec.template.metadata.labels 必须匹配 .spec.selector，否则它将被 API 拒绝。

Kubernetes 1.9 版本中，API 版本 apps/v1 中的 ReplicaSet 类型的版本是当前版本并默认开启。API 版本 apps/v1beta2 被弃用。

另外，通常您不应该创建标签与此选择器匹配的任何 Pod，或者直接与另一个 ReplicaSet 或另一个控制器（如 Deployment）标签匹配的任何 Pod。 如果你这样做，ReplicaSet 会认为它创造了其他 Pod。Kubernetes 并不会阻止您这样做。

如果您最终使用了多个具有重叠选择器的控制器，则必须自己负责删除。

Replicas

通过设置 .spec.replicas 您可以指定要同时运行多少个 Pod。 在任何时间运行的 Pod 数量可能高于或低于 .spec.replicas 指定的数量，例如在副本刚刚被增加或减少后、或者 Pod 正在被优雅地关闭、以及替换提前开始。

如果您没有指定 .spec.replicas, 那么默认值为 1。

使用 ReplicaSets 的具体方法

删除 ReplicaSet 和它的 Pod

要删除 ReplicaSet 和它的所有 Pod，使用[kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete) 命令。 默认情况下，[垃圾收集器](https://kubernetes.io/docs/concepts/workloads/controllers/garbage-collection/) 自动删除所有依赖的 Pod。

当使用 REST API 或 client-go 库时，您必须在删除选项中将 propagationPolicy 设置为 Background 或 Foreground。例如：

kubectl proxy --port=8080

curl -X DELETE 'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/frontend' **\**

> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' **\**

> -H "Content-Type: application/json"

只删除 ReplicaSet

您可以只删除 ReplicaSet 而不影响它的 Pod，方法是使用[kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete) 命令并设置 --cascade=false 选项。

当使用 REST API 或 client-go 库时，您必须将 propagationPolicy 设置为 Orphan。例如：

kubectl proxy --port=8080

curl -X DELETE 'localhost:8080/apis/extensions/v1beta1/namespaces/default/replicasets/frontend' **\**

> -d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' **\**

> -H "Content-Type: application/json"

一旦删除了原来的 ReplicaSet，就可以创建一个新的来替换它。 由于新旧 ReplicaSet 的 .spec.selector 是相同的，新的 ReplicaSet 将接管老的 Pod。 但是，它不会努力使现有的 Pod 与新的、不同的 Pod 模板匹配。 若想要以可控的方式将 Pod 更新到新的 spec，就要使用 [滚动更新](https://kubernetes.io/zh/docs/concepts/workloads/controllers/#rolling-updates)的方式。

将 Pod 从 ReplicaSet 中隔离

可以通过改变标签来从 ReplicaSet 的目标集中移除 Pod。这种技术可以用来从服务中去除 Pod，以便进行排错、数据恢复等。 以这种方式移除的 Pod 将被自动替换（假设副本的数量没有改变）。

缩放 RepliaSet

通过更新 .spec.replicas 字段，ReplicaSet 可以被轻松的进行缩放。ReplicaSet 控制器能确保匹配标签选择器的数量的 Pod 是可用的和可操作的。

ReplicaSet 作为水平的 Pod 自动缩放器目标

ReplicaSet 也可以作为 [水平的 Pod 缩放器 (HPA)](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/) 的目标。也就是说，ReplicaSet 可以被 HPA 自动缩放。 以下是 HPA 以我们在前一个示例中创建的副本集为目标的示例。

| [controllers/hpa-rs.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/hpa-rs.yaml) |
| --- |
| apiVersion: autoscaling/v1  kind: HorizontalPodAutoscaler  metadata:  name: frontend-scaler  spec:  scaleTargetRef:  kind: ReplicaSet  name: frontend  minReplicas: 3  maxReplicas: 10  targetCPUUtilizationPercentage: 50 |

将这个列表保存到 hpa-rs.yaml 并提交到 Kubernetes 集群，就能创建它所定义的 HPA，进而就能根据复制的 Pod 的 CPU 利用率对目标 ReplicaSet进行自动缩放。

kubectl create -f https://k8s.io/examples/controllers/hpa-rs.yaml

或者，可以使用 kubectl autoscale 命令完成相同的操作。 (而且它更简单！)

kubectl autoscale rs frontend

ReplicaSet 的替代方案

Deployment （推荐）

[Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 是一个高级 API 对象，它以 kubectl rolling-update 的方式更新其底层副本集及其Pod。 如果您需要滚动更新功能，建议使用 Deployment，因为 Deployment 与 kubectl rolling-update 不同的是：它是声明式的、服务器端的、并且具有其他特性。 有关使用 Deployment 来运行无状态应用的更多信息，请参阅 [使用 Deployment 运行无状态应用](https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/)。

裸 Pod

与用户直接创建 Pod 的情况不同，ReplicaSet 会替换那些由于某些原因被删除或被终止的 Pod，例如在节点故障或破坏性的节点维护（如内核升级）的情况下。 因为这个好处，我们建议您使用 ReplicaSet，即使应用程序只需要一个 Pod。 想像一下，ReplicaSet 类似于进程监视器，只不过它在多个节点上监视多个 Pod，而不是在单个节点上监视单个进程。 ReplicaSet 将本地容器重启的任务委托给了节点上的某个代理（例如，Kubelet 或 Docker）去完成。

Job

使用[Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) 代替ReplicaSet，可以用于那些期望自行终止的 Pod。

DaemonSet

对于管理那些提供主机级别功能（如主机监控和主机日志）的容器，就要用[DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) 而不用 ReplicaSet。 这些 Pod 的寿命与主机寿命有关：这些 Pod 需要先于主机上的其他 Pod 运行，并且在机器准备重新启动/关闭时安全地终止。

反馈

ReplicationController

**注意：**

现在推荐使用配置 [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 的 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 来建立副本管理机制。

*ReplicationController* 确保在任何时候都有特定数量的 pod 副本处于运行状态。 换句话说，ReplicationController 确保一个 pod 或一组同类的 pod 总是可用的。

* [**ReplicationController 如何工作**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#replicationcontroller-%e5%a6%82%e4%bd%95%e5%b7%a5%e4%bd%9c)
* [**运行一个示例 ReplicationController**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e8%bf%90%e8%a1%8c%e4%b8%80%e4%b8%aa%e7%a4%ba%e4%be%8b-replicationcontroller)
* [**编写一个 ReplicationController Spec**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e7%bc%96%e5%86%99%e4%b8%80%e4%b8%aa-replicationcontroller-spec)
* [**使用 ReplicationController**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e4%bd%bf%e7%94%a8-replicationcontroller)
* [**常见的使用模式**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e5%b8%b8%e8%a7%81%e7%9a%84%e4%bd%bf%e7%94%a8%e6%a8%a1%e5%bc%8f)
* [**编写多副本的应用**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e7%bc%96%e5%86%99%e5%a4%9a%e5%89%af%e6%9c%ac%e7%9a%84%e5%ba%94%e7%94%a8)
* [**ReplicationController 的职责**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#replicationcontroller-%e7%9a%84%e8%81%8c%e8%b4%a3)
* [**API 对象**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#api-%e5%af%b9%e8%b1%a1)
* [**ReplicationController 的替代方案**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#replicationcontroller-%e7%9a%84%e6%9b%bf%e4%bb%a3%e6%96%b9%e6%a1%88)
* [**更多信息**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#%e6%9b%b4%e5%a4%9a%e4%bf%a1%e6%81%af)

ReplicationController 如何工作

当 pods 数量过多时，ReplicationController 会终止多余的 pods。当 pods 数量太少时，ReplicationController 将会启动新的 pods。 与手动创建的 pod 不同，由 ReplicationController 创建的 pods 在失败、被删除或被终止时会被自动替换。 例如，在中断性维护（如内核升级）之后，您的 pod 会在节点上重新创建。 因此，即使您的应用程序只需要一个 pod，您也应该使用一个 ReplicationController。 ReplicationController 类似于进程管理器，但是 ReplicationController 不是监控单个节点上的单个进程，而是监控跨多个节点的多个 pods。

在讨论中，ReplicationController 通常缩写为 “rc” 或 “rcs”，并作为 kubectl 命令的快捷方式。

一个简单的例子是创建一个 ReplicationController 对象来可靠地无限期地运行 Pod 的一个实例。 更复杂的用例是运行一个多副本服务（如 web 服务器）的若干相同副本。

运行一个示例 ReplicationController

这个示例 ReplicationController 配置运行 nginx web 服务器的三个副本。

| [controllers/replication.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/replication.yaml) |
| --- |
| apiVersion: v1  kind: ReplicationController  metadata:  name: nginx  spec:  replicas: 3  selector:  app: nginx  template:  metadata:  name: nginx  labels:  app: nginx  spec:  containers:  - name: nginx  image: nginx  ports:  - containerPort: 80 |

通过下载示例文件并运行以下命令来运行示例任务:

kubectl apply -f https://k8s.io/examples/controllers/replication.yaml

replicationcontroller/nginx created

使用以下命令检查 ReplicationController 的状态:

kubectl describe replicationcontrollers/nginx

Name: nginx

Namespace: default

Selector: app=nginx

Labels: app=nginx

Annotations: <none>

Replicas: 3 current / 3 desired

Pods Status: 0 Running / 3 Waiting / 0 Succeeded / 0 Failed

Pod Template:

Labels: app=nginx

Containers:

nginx:

Image: nginx

Port: 80/TCP

Environment: <none>

Mounts: <none>

Volumes: <none>

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- ---- ------ -------

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-qrm3m

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-3ntk0

20s 20s 1 {replication-controller } Normal SuccessfulCreate Created pod: nginx-4ok8v

此时，创建了三个 pod，但是还没有运行，可能正在拉取镜像。 稍后，相同的命令可能显示：

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

要以机器可读的形式列出属于 ReplicationController 的所有 pod，可以使用如下命令：

pods=**$(**kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name}**)**

echo $pods

nginx-3ntk0 nginx-4ok8v nginx-qrm3m

这里，选择器与 ReplicationController 的选择器相同（参见 kubectl describe 输出），并以不同的形式出现在 replication.yaml 中。 --output=jsonpath 选项指定了一个表达式，只从返回列表中的每个 pod 中获取名称。

编写一个 ReplicationController Spec

与所有其它 Kubernetes 配置一样，ReplicationController 需要 apiVersion ，kind 和 metadata 字段。 有关使用配置文件的常规信息，参考[对象管理](https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/)。

ReplicationController 也需要一个 [.spec 部分](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status)。

Pod 模板

.spec.template 是 .spec 的唯一必需字段。

.spec.template 是一个 [pod 模板](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates)。它的模式与 [pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/) 完全相同，只是它是嵌套的，并且没有 apiVersion 或 kind。

除了 Pod 所需的字段外，ReplicationController 中的 pod 模板必须指定适当的标签和适当的重新启动策略。 对于标签，请确保不与其他控制器重叠。参考 [pod 选择器](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#pod-选择器)。

只允许 [.spec.template.spec.restartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) 等于 Always，如果没有指定，这是默认值。

对于本地容器重启，ReplicationController 委托给节点上的代理， 例如 [Kubelet](https://kubernetes.io/docs/admin/kubelet/) 或 Docker。

ReplicationController 上的标签

ReplicationController 本身可以有标签 （.metadata.labels）。 通常，您可以将这些设置为 .spec.template.metadata.labels； 如果没有指定 .metadata.labels 那么它默认为 .spec.template.metadata.labels。  
但是，Kubernetes 允许它们是不同的，.metadata.labels 不会影响 ReplicationController 的行为。

Pod 选择器

.spec.selector 字段是一个[标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)。 ReplicationController 管理标签与选择器匹配的所有 Pods。 它不区分它创建或删除的 pod 以及另一个人或进程创建或删除的 pod。 这允许在不影响正在运行的 pod 的情况下替换 ReplicationController。

如果指定了 .spec.template.metadata.labels，它必须和 .spec.selector 相同，否则它将被 API 拒绝。 如果没有指定 .spec.selector，它将默认为 .spec.template.metadata.labels。

此外，用户通常不应创建标签与此选择器匹配的任何其他 Pods，无论是直接创建、使用另一个 ReplicationController 来创建或者别的控制器（如 Job ）来创建，都是不允许的。 如果这样做， ReplicationController 会认为这些 Pods 是由它自己创建的。 Kubernetes 并没有阻止你这样做。

如果您的确创建了多个控制器并且其选择器之间存在重叠，那么您将不得不自己管理删除操作（参考[后文](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#使用-replicationcontrollers)）。

多个副本

你可以通过设置 .spec.replicas 来指定应该同时运行多少个 Pods。 在任何时候，处于运行状态的 Pods 个数都可能高于或者低于设定值。例如，副本个数刚刚被增加或减少时，或者一个 pod 处于体面终止过程中而其替代副本已经提前开始创建时。

如果你没有指定 .spec.replicas ，那么它默认是1。

使用 ReplicationController

删除一个 ReplicationController 以及它的 Pod

要删除一个 ReplicationController 以及它的 Pod，使用 [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete)。 Kubectl 将 ReplicationController 缩放为0并等待以便在删除 ReplicationController 本身之前删除每个 pod。 如果这个 kubectl 命令被中断，可以重新启动它。

当使用 REST API 或 go 客户端库时，您需要明确地执行这些步骤（缩放副本为0、 等待 Pod 删除，之后删除 ReplicationController 资源）。

只删除 ReplicationController

你可以删除一个 ReplicationController 而不影响它的任何 pod。

使用 kubectl ，为 [kubectl delete](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#delete) 指定 --cascade=false 选项。

当使用 REST API 或 go 客户端库时， 只需删除 ReplicationController 对象。

一旦原始对象被删除，你可以创建一个新的 ReplicationController 来替换它。 只要新的和旧的 .spec.selector 相同，那么新的控制器将领养旧的 Pods。 但是，它不会做出任何努力使现有的 pod 匹配新的、不同的 pod 模板。 如果希望以受控方式更新 Pods 以使用新的 spec，请执行[滚动更新](https://kubernetes.io/zh/docs/concepts/workloads/controllers/replicationcontroller/#滚动更新)操作。

从 ReplicationController 中隔离 pod

通过更改 Pod 的标签，可以从 ReplicationController 的目标中删除 pod。 此技术可用于从服务中删除 pod 以进行调试、数据恢复等。以这种方式删除的 pod 将自动替换（假设复制副本的数量也没有更改）。

常见的使用模式

重新调度

如上所述，无论您想要继续运行1个 pod 还是1000个，一个 ReplicationController 都将确保存在指定数量的 pod，即使在节点故障或 pod 终止(例如，由于另一个控制代理的操作)的情况下也是如此。 <!–

Scaling

The ReplicationController makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the replicas field. –>

扩缩容

通过简单地更新 replicas 字段，ReplicationController 可以方便地横向扩容或缩容副本的数量，或手动或通过自动缩放控制代理。

滚动更新

ReplicationController 的设计目的是通过逐个替换 pod 以方便滚动更新服务。

如[#1353](http://issue.k8s.io/1353)所述，建议的方法是使用1个副本创建一个新的 ReplicationController，逐个缩放新的（+1）和旧的（-1）控制器，然后在旧的控制器达到0个副本后将其删除。这一方法能够实现可控的 Pods 集合更新，即使存在意外失效的状况。

理想情况下，滚动更新控制器将考虑应用程序的就绪情况，并确保在任何给定时间都有足够数量的 Pods 有效地提供服务。

这两个 ReplicationController 将需要创建至少具有一个不同标签的 pod，比如 pod 主要容器的镜像标签，因为通常是镜像更新触发滚动更新。

滚动更新是在客户端工具 [kubectl rolling-update](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#rolling-update)中实现的。 访问 [kubectl rolling-update 任务](https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/)以获得更多的具体示例。

多个版本跟踪

除了在滚动更新过程中运行应用程序的多个版本之外，通常还会使用多个版本跟踪来长时间，甚至持续运行多个版本。这些跟踪将根据标签加以区分。

例如，一个服务可能把具有 tier in (frontend), environment in (prod) 的所有 pod 作为目标。 现在假设您有10个副本的 pod 组成了这个层。但是你希望能够’金丝雀部署’这个组件的新版本。 您可以为大部分副本设置一个 ReplicationController，其中 replicas 设置为9，标签为 tier=frontend, environment=prod, track=stable 而为 canary 设置另一个 ReplicationController，其中 replicas 设置为1，标签为 tier=frontend, environment=prod, track=canary。 现在这个服务覆盖了 canary 和非 canary pod。但您可以单独处理 ReplicationController，以测试、监控结果等。

和服务一起使用 ReplicationController

多个 ReplicationController 可以位于一个服务的后面，例如，一部分流量流向旧版本，一部分流量流向新版本。

一个 ReplicationController 永远不会自行终止，但它不会像服务那样长寿。 服务可以由多个 ReplicationController 控制的 pod 组成，并且在服务的生命周期内（例如，为了执行 pod 更新而运行服务），可以创建和销毁许多 ReplicationController。 服务本身和它们的客户端都应该忽略负责维护服务 Pod 的 ReplicationController 的存在。

编写多副本的应用

由 ReplicationController 创建的 Pod 是可替换的，语义上是相同的，尽管随着时间的推移，它们的配置可能会变得异构。 这显然适合于多副本的无状态服务器，但是 ReplicationController 也可以用于维护主选、分片和工作池应用程序的可用性。 这样的应用程序应该使用动态的工作分配机制，例如 [RabbitMQ工作队列](https://www.rabbitmq.com/tutorials/tutorial-two-python.html)，而不是静态的/一次性定制每个 pod 的配置，这被认为是一种反模式。 执行的任何 pod 定制，例如资源的垂直自动调整大小(例如，cpu 或内存)，都应该由另一个在线控制器进程执行，这与 ReplicationController 本身没什么不同。

ReplicationController 的职责

ReplicationController 只需确保所需的 pod 数量与其标签选择器匹配，并且是可操作的。 目前，它的计数中只排除终止的 pod。 未来，可能会考虑系统提供的[就绪状态](http://issue.k8s.io/620)和其他信息，我们可能会对替换策略添加更多控制，我们计划发出事件，这些事件可以被外部客户端用来实现任意复杂的替换和/或缩减策略。

ReplicationController 永远被限制在这个狭隘的职责范围内。 它本身既不执行就绪态探测，也不执行活跃性探测。 它不负责执行自动缩放，而是由外部自动缩放器控制（如[#492](http://issue.k8s.io/492)中所述），后者负责更改其 replicas 字段取值。 我们不会向 ReplicationController 添加调度策略(例如，[spreading](http://issue.k8s.io/367#issuecomment-48428019))。 它也不应该验证所控制的 pod 是否与当前指定的模板匹配，因为这会阻碍自动调整大小和其他自动化过程。 类似地，完成期限、整理依赖关系、配置扩展和其他特性也属于其他地方。 我们甚至计划考虑批量创建 pod 的机制（[#170](http://issue.k8s.io/170)）。

ReplicationController 旨在成为可组合的构建基元。 我们希望在它和其他补充原语的基础上构建更高级别的 API 和/或工具，以便于将来的用户使用。 Kubectl 目前支持的“宏”操作（运行、缩放、滚动更新）就是这方面的概念示例。 例如，我们可以想象类似于 [Asgard](http://techblog.netflix.com/2012/06/asgaard-web-based-cloud-management-and.html) 的东西管理 ReplicationController、自动定标器、服务、调度策略、 canary 等。

API 对象

在 Kubernetes REST API 中 Replication controller 是顶级资源。 更多关于 API 对象的详细信息可以在 [ReplicationController API 对象](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#replicationcontroller-v1-core)找到。

ReplicationController 的替代方案

ReplicaSet

[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 是下一代 ReplicationController ，支持新的[基于集合的标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement)。 它主要被 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 用来作为一种编排 pod 创建、删除及更新的机制。 请注意，我们推荐使用 Deployment 而不是直接使用 ReplicaSet，除非您需要自定义更新编排或根本不需要更新。

Deployment （推荐）

[Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 是一种更高级别的 API 对象，它以类似于 kubectl rolling-update 的方式更新其底层 ReplicaSet 及其 Pod。 如果您想要这种滚动更新功能，那么推荐使用 Deployment，因为与 kubectl rolling-update 不同，它们是声明式的、服务端的，并且具有其它特性。

裸 Pods

与用户直接创建 pod 的情况不同，ReplicationController 能够替换因某些原因被删除或被终止的 pod ，例如在节点故障或中断节点维护的情况下，例如内核升级。 因此，我们建议您使用 ReplicationController，即使您的应用程序只需要一个 pod。 可以将其看作类似于进程管理器，它只管理跨多个节点的多个 pod ，而不是单个节点上的单个进程。 ReplicationController 将本地容器重启委托给节点上的某个代理(例如，Kubelet 或 Docker)。

Job

对于预期会自行终止的 pod (即批处理任务)，使用 [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) 而不是 ReplicationController。

DaemonSet

对于提供机器级功能（例如机器监控或机器日志记录）的 pod ，使用 [DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) 而不是 ReplicationController。 这些 pod 的生命期与机器的生命期绑定：它们需要在其他 pod 启动之前在机器上运行，并且在机器准备重新启动/关闭时安全地终止。

更多信息

请阅读[运行无状态的 Replication Controller](https://kubernetes.io/docs/tutorials/stateless-application/run-stateless-ap-replication-controller/)。

StatefulSets

StatefulSet 是用来管理有状态应用的工作负载 API 对象。

**注意：**

**注意** StatefulSets 是在 1.9 版本中正式发布的。

StatefulSet 用来管理 Deployment 和扩展一组 Pod，并且能为这些 Pod 提供\*序号和唯一性保证\*。

和 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 相同的是，StatefulSet 管理了基于相同容器定义的一组 Pod。但和 Deployment 不同的是，StatefulSet 为它们的每个 Pod 维护了一个固定的 ID。这些 Pod 是基于相同的声明来创建的，但是不能相互替换：无论怎么调度，每个 Pod 都有一个永久不变的 ID。

StatefulSet 和其他控制器使用相同的工作模式。你在 StatefulSet *对象* 中定义你期望的状态，然后 StatefulSet 的 *控制器* 就会通过各种更新来达到那种你想要的状态。

* [**使用 StatefulSets**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e4%bd%bf%e7%94%a8-statefulsets)
* [**限制**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e9%99%90%e5%88%b6)
* [**组件**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e7%bb%84%e4%bb%b6)
* [**Pod 选择器**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#pod-%e9%80%89%e6%8b%a9%e5%99%a8)
* [**Pod 的身份**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#pod-%e7%9a%84%e8%ba%ab%e4%bb%bd)
* [**部署和伸缩保证**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e9%83%a8%e7%bd%b2%e5%92%8c%e4%bc%b8%e7%bc%a9%e4%bf%9d%e8%af%81)
* [**更新策略**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e6%9b%b4%e6%96%b0%e7%ad%96%e7%95%a5)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

使用 StatefulSets

StatefulSets 对于需要满足以下一个或多个需求的应用程序很有价值：

* 稳定的、唯一的网络标识符。
* 稳定的、持久的存储。
* 有序的、优雅的部署和缩放。
* 有序的、自动的滚动更新。

在上面，稳定意味着 Pod 调度或重调度的整个过程是有持久性的。如果应用程序不需要任何稳定的标识符或有序的部署、删除或伸缩，则应该使用由一组无状态的副本控制器提供的工作负载来部署应用程序，比如 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 或者 [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 可能更适用于您的无状态需要。

限制

* 给定 Pod 的存储必须由 [PersistentVolume 驱动](https://github.com/kubernetes/examples/tree/master/staging/persistent-volume-provisioning/README.md) 基于所请求的 storage class 来提供，或者由管理员预先提供。
* 删除和／或收缩 StatefulSet 并\*不会\*删除它关联的存储卷。这样做是为了保证数据安全，它通常比自动清除 StatefulSet 所有相关的资源更有价值。
* StatefulSet 当前需要 [headless 服务](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) 来负责 Pod 的网络标识。您需要负责创建此服务。
* 当删除 StatefulSets 时，StatefulSet 不提供任何终止 Pod 的保证。为了实现 StatefulSet 中的 Pod 可以有序和优雅的终止，可以在删除之前将 StatefulSet 缩放为 0。
* 在默认 [Pod 管理策略](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#pod-management-policies)(OrderedReady) 时使用 [滚动更新](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#rolling-updates)，可能进入需要 [人工干预](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#forced-rollback) 才能修复的损坏状态。

组件

下面的示例演示了 StatefulSet 的组件。

* 名为 nginx 的无头服务用来控制网络域。
* 名为 web 的 StatefulSet 有一个 Spec，它表明将在单个 Pod 中启动 nginx 容器的 3 个副本。
* volumeClaimTemplates 将通过 [PersistentVolumes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) 驱动提供的 PersistentVolume 来提供稳定的存储。
* apiVersion: v1
* kind: Service
* metadata:
* name: nginx
* labels:
* app: nginx
* spec:
* ports:
* - port: 80
* name: web
* clusterIP: None
* selector:
* app: nginx
* ---
* apiVersion: apps/v1
* kind: StatefulSet
* metadata:
* name: web
* spec:
* selector:
* matchLabels:
* app: nginx *# has to match .spec.template.metadata.labels*
* serviceName: "nginx"
* replicas: 3 *# by default is 1*
* template:
* metadata:
* labels:
* app: nginx *# has to match .spec.selector.matchLabels*
* spec:
* terminationGracePeriodSeconds: 10
* containers:
* - name: nginx
* image: k8s.gcr.io/nginx-slim:0.8
* ports:
* - containerPort: 80
* name: web
* volumeMounts:
* - name: www
* mountPath: /usr/share/nginx/html
* volumeClaimTemplates:
* - metadata:
* name: www
* spec:
* accessModes: [ "ReadWriteOnce" ]
* storageClassName: "my-storage-class"
* resources:
* requests:

storage: 1Gi

Pod 选择器

您必须设置 StatefulSet 的 .spec.selector 字段来匹配它的.spec.template.metadata.labels标签。在 Kubernetes 1.8 版本之前，忽略.spec.selector 字段会提供默认设置。在 1.8 和以后的版本中，指定的 Pod 选择器匹配失败将在创建 StatefulSet 期间导致验证错误。

Pod 的身份

StatefulSet Pod 具有唯一的标识，该标识包括顺序标识、稳定的网络标识和稳定的存储。该标识和 Pod 是绑定的，不管它被调度在哪个节点上。

有序索引

对于具有 N 个副本的 StatefulSet，StatefulSet 中的每个 Pod 将被分配一个整数序号，从 0 到 N-1，该序号在 StatefulSet 上是唯一的。

稳定的网络 ID

StatefulSet 中的每个 Pod 根据 StatefulSet 的名称和 Pod 的序号派生出它的主机名。组合主机名的格式为$(statefulset name)-$(ordinal)。上例将会创建三个名称为web-0,web-1,web-2的三个 Pod。 StatefulSet 可以使用 [headless 服务](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) 控制它的 Pod 的网络域。管理域的这个服务的格式为： $(service name).$(namespace).svc.cluster.local，其中cluster.local是集群域。 一旦每个 Pod 创建成功，就会得到一个匹配的 DNS 子域，格式为：$(podname).$(governing service domain)，其中管理服务由 StatefulSet 的 serviceName 域来定义。

下面给出一些选择集群域、服务名、StatefulSet 名、及其怎样影响 StatefulSet 的 Pod 上的 DNS 名称的示例：

| Cluster Domain | Service (ns/name) | StatefulSet (ns/name) | StatefulSet Domain | Pod DNS | Pod Hostname |
| --- | --- | --- | --- | --- | --- |
| cluster.local | default/nginx | default/web | nginx.default.svc.cluster.local | web-{0..N-1}.nginx.default.svc.cluster.local | web-{0..N-1} |
| cluster.local | foo/nginx | foo/web | nginx.foo.svc.cluster.local | web-{0..N-1}.nginx.foo.svc.cluster.local | web-{0..N-1} |
| kube.local | foo/nginx | foo/web | nginx.foo.svc.kube.local | web-{0..N-1}.nginx.foo.svc.kube.local | web-{0..N-1} |

**注意：**

**注意** 集群域会被设置为 cluster.local 除非有[其他配置](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/#how-it-works)。

稳定的存储

Kubernetes 为每个 VolumeClaimTemplate 创建一个 [PersistentVolume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)。在上面的 nginx 示例中，每个 Pod 将会得到基于 StorageClass my-storage-class 提供的 1 Gib 的 PersistentVolume。如果没有声明 StorageClass，就会使用默认的 StorageClass。当一个 Pod 被安排到节点上时，它的 volumeMounts 挂载了与其 PersistentVolumeClaims 相关联的 PersistentVolume。请注意，当 Pod 或者 StatefulSet 被删除时，与 PersistentVolumeClaims 相关联的 PersistentVolume 并不会被删除。要删除它必须通过手动方式来完成。

Pod 名称标签

当 StatefulSet [控制器](https://kubernetes.io/docs/admin/kube-controller-manager/) 创建 Pod 时，它会添加一个标签 statefulset.kubernetes.io/pod-name，该标签设置为 Pod 名称。这个标签允许您给 StatefulSet 中的特定 Pod 绑定一个 Service。

部署和伸缩保证

* 对于包含 N 个 副本的 StatefulSet，当部署 Pod 时，它们是依次创建的，顺序为0..N-1。
* 当删除 Pod 时，它们是逆序终止的，顺序为N-1..0。
* 在将缩放操作应用到 Pod 之前，它前面的所有 Pod 必须是 Running 和 Ready 状态。
* 在 Pod 终止之前，所有的继任者必须完全关闭。

StatefulSet 不应该声明 pod.Spec.TerminationGracePeriodSeconds 为 0。这种做法是不安全的，要强烈阻止。更多的解释请参考 [强制删除 StatefulSet Pods](https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/)。

在上面的 nginx 示例被创建后，会按照 web-0、web-1、web-2 的顺序部署三个 Pod。在 web-0 进入 [Running 和 Ready](https://kubernetes.io/docs/user-guide/pod-states/) 状态前不会部署 web-1。在 web-1 进入 Running 和 Ready 状态前不会部署 web-2。如果 web-1 已经处于 Running 和 Ready 状态，而 web-2 尚未部署，在此期间发生了 web-0 运行失败，那么 web-2 将不会被部署，要等到 web-0 部署完成并进入 Running 和 Ready 状态后，才会部署 web-2。

如果用户想将示例中的 StatefulSet 收缩为 replicas=1，首先被终止的是 web-2。在 web-2 没有被完全停止和删除前，web-1 不会被终止。当 web-2 已被终止和删除、web-1 尚未被终止，如果在此期间发生 web-0 运行失败，那么就不会终止 web-1，必须等到 web-0 进入 Running 和 Ready 状态后才会终止 web-1。

Pod 管理策略

在 Kubernetes 1.7 及以后的版本中，StatefulSet 允许您放松其排序保证，同时通过它的 .spec.podManagementPolicy 域保持其唯一性和身份保证。

有序的 Pod 管理

有序的 Pod 管理是 StatefulSet 的默认功能。它实现了 [上面](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#deployment-and-scaling-guarantees)描述的行为。

并行的 Pod 管理

并行 Pod 管理让 StatefulSet 控制器并行的启动或终止所有的 Pod，启动或者终止其他 Pod 前，无需等待 Pod 进入 Running 和 ready 或者完全停止状态。

更新策略

在 Kubernetes 1.7 及以后的版本中，StatefulSet 的 .spec.updateStrategy 字段让您可以配置和禁用掉自动滚动更新 Pod 的容器、标签、资源请求／限制、以及注解。

On Delete

OnDelete 更新策略实现了 1.6 及以前版本的历史遗留行为。当 StatefulSet 的 .spec.updateStrategy.type 设置为 OnDelete 时，它的控制器将不会自动更新 StatefulSet 中的 Pod。用户必须手动删除 Pod 以便让控制器创建新的 Pod，以此来对 StatefulSet 的 .spec.template 的变动作出反应。

滚动更新

RollingUpdate 更新策略对 StatefulSet 中的 Pod 执行自动的滚动更新。在没有声明.spec.updateStrategy时，RollingUpdate是默认配置。 当 StatefulSet 的 .spec.updateStrategy.type 被设置为 RollingUpdate 时，StatefulSet 控制器会删除和重建 StatefulSet 中的每个 Pod。 它将按照与 Pod 终止相同的顺序（从最大序号到最小序号）进行，每次更新一个 Pod。它会等到被更新的 Pod 进入 Running 和 Ready 状态，然后再更新其前身。

分区

通过声明 .spec.updateStrategy.rollingUpdate.partition的方式，RollingUpdate更新策略可以实现分区。如果声明了一个分区，当 StatefulSet 的.spec.template 被更新时，所有序号大于等于该分区序号的 Pod 都会被更新。所有序号小于该分区序号的 Pod 都不会被更新，并且，即使他们被删除也会依据之前的版本进行重建。如果 StatefulSet 的 .spec.updateStrategy.rollingUpdate.partition 大于它的 .spec.replicas，对它的 .spec.template 的更新将不会传递到它的 Pod。 在大多数情况下，您不需要使用分区，但如果您希望进行阶段更新、执行金丝雀或执行分阶段展开，则这些分区会非常有用。

强制回滚

在默认 [Pod 管理策略](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#pod-management-policies)(OrderedReady) 时使用 [滚动更新](https://kubernetes.io/zh/docs/concepts/workloads/controllers/statefulset/#rolling-updates) ，可能进入需要人工干预才能修复的损坏状态。

如果更新后 Pod 模板配置进入无法运行或就绪的状态（例如，由于错误的二进制文件或应用程序级配置错误），StatefulSet 将停止回滚并等待。

在这种状态下，仅将 Pod 模板还原为正确的配置是不够的。由于 [已知问题](https://github.com/kubernetes/kubernetes/issues/67250)， StatefulSet 将继续等待损坏状态的 Pod 准备就绪（永远不会发生），然后再尝试将其恢复为正常工作配置。

恢复模板后，还必须删除 StatefulSet 尝试使用错误的配置来运行的 Pod。这样，StatefulSet 才会开始使用被还原的模板来重新创建 Pod。

接下来

* 示例一：[部署有状态应用](https://kubernetes.io/docs/tutorials/stateful-application/basic-stateful-set/)。
* 示例二：[使用 StatefulSet 部署 Cassandra](https://kubernetes.io/docs/tutorials/stateful-application/cassandra/)。

反馈

DaemonSet

*DaemonSet* 确保全部（或者某些）节点上运行一个 Pod 的副本。当有节点加入集群时， 也会为他们新增一个 Pod 。当有节点从集群移除时，这些 Pod 也会被回收。删除 DaemonSet 将会删除它创建的所有 Pod。

* 运行集群存储 daemon，例如在每个节点上运行 glusterd、ceph。
* 在每个节点上运行日志收集 daemon，例如fluentd、logstash。
* 在每个节点上运行监控 daemon，例如 [Prometheus Node Exporter](https://github.com/prometheus/node_exporter)、[Sysdig 代理](https://sysdigdocs.atlassian.net/wiki/spaces/Platform)、collectd、[Dynatrace OneAgent](https://www.dynatrace.com/technologies/kubernetes-monitoring/)、[AppDynamics 代理](https://docs.appdynamics.com/display/CLOUD/Container+Visibility+with+Kubernetes)、[Datadog 代理](https://docs.datadoghq.com/agent/kubernetes/daemonset_setup/)、[New Relic 代理](https://docs.newrelic.com/docs/integrations/kubernetes-integration/installation/kubernetes-installation-configuration)，Ganglia gmond 或 [Instana 代理](https://www.instana.com/supported-integrations/kubernetes-monitoring/)。

一个简单的用法是在所有的节点上都启动一个 DaemonSet，将被作为每种类型的 daemon 使用。

一个稍微复杂的用法是单独对每种 daemon 类型使用多个 DaemonSet，但具有不同的标志， 和/或对不同硬件类型具有不同的内存、CPU要求。

* [**编写 DaemonSet Spec**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#%e7%bc%96%e5%86%99-daemonset-spec)
* [**如何调度 Daemon Pods**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#%e5%a6%82%e4%bd%95%e8%b0%83%e5%ba%a6-daemon-pods)
* [**与 Daemon Pods 通信**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#%e4%b8%8e-daemon-pods-%e9%80%9a%e4%bf%a1)
* [**更新 DaemonSet**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#%e6%9b%b4%e6%96%b0-daemonset)
* [**DaemonSet 的可替代选择**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#daemonset-%e7%9a%84%e5%8f%af%e6%9b%bf%e4%bb%a3%e9%80%89%e6%8b%a9)

编写 DaemonSet Spec

创建 DaemonSet

您可以在 YAML 文件中描述 DaemonSet。例如，下面的 daemonset.yaml 文件描述了一个运行 fluentd-elasticsearch Docker 镜像的 DaemonSet：

| [controllers/daemonset.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/daemonset.yaml) |
| --- |
| apiVersion: apps/v1  kind: DaemonSet  metadata:  name: fluentd-elasticsearch  namespace: kube-system  labels:  k8s-app: fluentd-logging  spec:  selector:  matchLabels:  name: fluentd-elasticsearch  template:  metadata:  labels:  name: fluentd-elasticsearch  spec:  tolerations:  - key: node-role.kubernetes.io/master  effect: NoSchedule  containers:  - name: fluentd-elasticsearch  image: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2  resources:  limits:  memory: 200Mi  requests:  cpu: 100m  memory: 200Mi  volumeMounts:  - name: varlog  mountPath: /var/log  - name: varlibdockercontainers  mountPath: /var/lib/docker/containers  readOnly: **true**  terminationGracePeriodSeconds: 30  volumes:  - name: varlog  hostPath:  path: /var/log  - name: varlibdockercontainers  hostPath:  path: /var/lib/docker/containers |

* 基于 YAML 文件创建 DaemonSet:
* kubectl apply -f https://k8s.io/examples/controllers/daemonset.yaml

必需字段

和其它所有 Kubernetes 配置一样，DaemonSet 需要 apiVersion、kind 和 metadata 字段。有关配置文件的基本信息，详见文档 [部署应用](https://kubernetes.io/docs/user-guide/deploying-applications/)、[配置容器](https://kubernetes.io/docs/tasks/) 和 [使用kubectl进行对象管理](https://kubernetes.io/docs/concepts/overview/object-management-kubectl/overview/)。

DaemonSet 也需要一个 [.spec](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status) 配置段。

Pod 模板

.spec 唯一必需的字段是 .spec.template。

.spec.template 是一个 [Pod 模板](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates)。它与 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/) 具有相同的 schema，除了它是嵌套的，而且不具有 apiVersion 或 kind 字段。

除了 Pod 必需字段外，在 DaemonSet 中的 Pod 模板必须指定合理的标签（查看 [Pod Selector](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/#pod-selector)）。

在 DaemonSet 中的 Pod 模板必须具有一个值为 Always 的 [RestartPolicy](https://kubernetes.io/docs/user-guide/pod-states)，或者未指定它的值，默认是 Always。

Pod Selector

.spec.selector 字段表示 Pod Selector，它与 [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/) 的 .spec.selector 的作用是相同的。

从 Kubernetes 1.8开始，您必须指定与 .spec.template 的标签匹配的 pod selector。当不配置时，pod selector 将不再有默认值。selector 默认与 kubectl apply 不兼容。 此外，一旦创建了 DaemonSet，它的 .spec.selector 就不能修改。修改 pod selector 可能导致成为 孤儿Pod，并且这对用户来说是困惑的。

spec.selector 表示一个对象，它由如下两个字段组成：

* matchLabels - 与 [ReplicationController](https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/) 的 .spec.selector 的作用相同。
* matchExpressions - 允许构建更加复杂的 Selector，可以通过指定 key、value 列表 ，以及与 key 和 value 列表相关的操作符。

当上述两个字段都指定时，结果表示的是 AND 关系。

如果指定了 .spec.selector，必须与 .spec.template.metadata.labels 相匹配。如果与它们配置的不匹配，则会被 API 拒绝。

此外，您通常不应该创建任何 pods，它们的 label 与 selector 匹配，或者直接创建，或者通过另一个 DaemonSet、或者其他控制器，比如 ReplicaSet。否则，DaemonSet 控制器会认为这些 Pod 是由它创建的。Kubernetes 不会阻止你这样做。 您可能希望这样做的一种情况是在节点上手动创建具有不同值的 Pod 以进行测试。

仅在某些节点上运行 Pod

如果指定了 .spec.template.spec.nodeSelector，DaemonSet Controller 将在能够与 [Node Selector](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/) 匹配的节点上创建 Pod。类似这种情况，可以指定 .spec.template.spec.affinity，然后 DaemonSet Controller 将在能够与 [node Affinity](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/) 匹配>的节点上创建 Pod。 如果根本就没有指定，则 DaemonSet Controller 将在所有节点上创建 Pod。

如何调度 Daemon Pods

通过默认 scheduler 调度

**FEATURE STATE:** Kubernetes v1.17 [稳定](https://kubernetes.io/zh/docs/concepts/workloads/controllers/daemonset/)

DaemonSet 确保所有符合条件的节点都运行一个 Pod 的副本。通常，运行 Pod 的节点由 Kubernetes scheduler 选择。然而，DaemonSet pods 由 DaemonSet controller 创建和调度。这将引入以下问题：

* Pod 行为的不一致性：等待调度的正常 Pod 已被创建并处于 Pending 状态，但 DaemonSet pods 未在 Pending 状态下创建。 这使用户感到困惑。
* [Pod preemption](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/)由默认 scheduler 处理。 启用抢占后，DaemonSet 控制器将在不考虑 pod 优先级和抢占的情况下制定调度决策。

ScheduleDaemonSetPods 允许您使用默认 scheduler 而不是 DaemonSet 控制器来调度 DaemonSets，方法是将 NodeAffinity 添加到 DaemonSet pods，而不是 .spec.nodeName。 然后使用默认 scheduler 将 pod 绑定到目标主机。 如果 DaemonSet pod的亲和节点已存在，则替换它。 DaemonSet 控制器仅在创建或修改 DaemonSet pods 时执行这些操作，并且不对 DaemonSet的 spec.template 进行任何更改。

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchFields:

- key: metadata.name

operator: In

values:

- target-host-name

此外，node.kubernetes.io/unschedulable：NoSchedule toleration 会自动添加到 DaemonSet Pods。 在调度DaemonSet Pod 时，默认调度器会忽略 unschedulable节点。

Taints and Tolerations

尽管 Daemon Pods 尊重[taints and tolerations](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration)，根据相关特性，会自动将以下 tolerations 添加到 DaemonSet Pods 中。

| Toleration Key | Effect | Version | Description |
| --- | --- | --- | --- |
| node.kubernetes.io/not-ready | NoExecute | 1.13+ | DaemonSet pods will not be evicted when there are node problems such as a network partition. |
| node.kubernetes.io/unreachable | NoExecute | 1.13+ | DaemonSet pods will not be evicted when there are node problems such as a network partition. |
| node.kubernetes.io/disk-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/memory-pressure | NoSchedule | 1.8+ |  |
| node.kubernetes.io/unschedulable | NoSchedule | 1.12+ | DaemonSet pods tolerate unschedulable attributes by default scheduler. |
| node.kubernetes.io/network-unavailable | NoSchedule | 1.12+ | DaemonSet pods, who uses host network, tolerate network-unavailable attributes by default scheduler. |

与 Daemon Pods 通信

与 DaemonSet 中的 Pod 进行通信，几种可能的模式如下：

* **Push**：配置 DaemonSet 中的 Pod 向其它 Service 发送更新，例如统计数据库。它们没有客户端。
* **NodeIP 和已知端口**：DaemonSet 中的 Pod 可以使用 hostPort，从而可以通过节点 IP 访问到 Pod。客户端能通过某种方法知道节点 IP 列表，并且基于此也可以知道端口 。
* **DNS**：创建具有相同 Pod Selector 的 [Headless Service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services)，然后通过使用 endpoints 资源或从 DNS 检索到多个 A 记录来发现 DaemonSet。
* **Service**：创建具有相同 Pod Selector 的 Service，并使用该 Service 随机访问到某个节点上的 daemon（没有办法访问到特定节点）。

更新 DaemonSet

如果修改了节点标签（Label），DaemonSet 将立刻向新匹配上的节点添加 Pod，同时删除不能够匹配的节点上的 Pod。

您可以修改 DaemonSet 创建的 Pod。然而，不允许对 Pod 的所有字段进行更新。当下次 节点（即使具有相同的名称）被创建时，DaemonSet Controller 还会使用最初的模板。

您可以删除一个 DaemonSet。如果使用 kubectl 并指定 --cascade=false 选项，则 Pod 将被保留在节点上。然后可以创建具有不同模板的新 DaemonSet。具有不同模板的新 DaemonSet 将能够通过标签匹配并识别所有已经存在的 Pod。 如果有任何 Pod 需要替换，则 DaemonSet 根据它的 updateStrategy 来替换。

DaemonSet 的可替代选择

init 脚本

我们很可能希望直接在一个节点上启动 daemon 进程（例如，使用 init、upstartd、或 systemd）。这非常好，但基于 DaemonSet 来运行这些进程有如下一些好处：

* 像对待应用程序一样，具备为 daemon 提供监控和管理日志的能力。
* 为 daemon 和应用程序使用相同的配置语言和工具（如 Pod 模板、kubectl）。
* 在资源受限的容器中运行 daemon，能够增加 daemon 和应用容器的隔离性。然而，这也实现了在容器中运行 daemon，但却不能在 Pod 中运行（例如，直接基于 Docker 启动）。

裸 Pod

可能要直接创建 Pod，同时指定其运行在特定的节点上。然而，DaemonSet 替换了由于任何原因被删除或终止的 Pod，例如节点失败、例行节点维护、内核升级。由于这个原因，我们应该使用 DaemonSet 而不是单独创建 Pod。

静态 Pod

可能需要通过在一个指定目录下编写文件来创建 Pod，该目录受 Kubelet 所监视。这些 Pod 被称为 [静态 Pod](https://kubernetes.io/docs/concepts/cluster-administration/static-pod/)。 不像 DaemonSet，静态 Pod 不受 kubectl 和其它 Kubernetes API 客户端管理。静态 Pod 不依赖于 apiserver，这使得它们在集群启动的情况下非常有用。而且，未来静态 Pod 可能会被废弃掉。

Deployments

DaemonSet 与 [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 非常类似，它们都能创建 Pod，这些 Pod 对应的进程都不希望被终止掉（例如，Web 服务器、存储服务器）。 为无状态的 Service 使用 Deployments，比如前端 Frontend 服务，实现对副本的数量进行扩缩容、平滑升级，比基于精确控制 Pod 运行在某个主机上要重要得多。 需要 Pod 副本总是运行在全部或特定主机上，并需要先于其他 Pod 启动，当这被认为非常重要时，应该使用 Daemon Controller。

反馈

已完成资源的 TTL 控制器

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/zh/docs/concepts/workloads/controllers/ttlafterfinished/)

TTL 控制器提供了一种 TTL 机制来限制已完成执行的资源对象的生命周期。TTL 控制器目前只处理[作业](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)，可能以后会扩展以处理将完成执行的其他资源，例如 Pod 和自定义资源。

Alpha 免责声明：此功能目前是 alpha 版，可以通过[feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) TTLAfterFinished 启用。

* [**TTL 控制器**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/ttlafterfinished/#ttl-%e6%8e%a7%e5%88%b6%e5%99%a8)
* [**警告**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/ttlafterfinished/#%e8%ad%a6%e5%91%8a)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/ttlafterfinished/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

TTL 控制器

TTL 控制器现在只支持 Jobs。集群操作员可以通过指定 Job 的 .spec.ttlSecondsAfterFinished 字段来自动清理已结束的作业（“完成”或“失败”），就像下边的[示例](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#clean-up-finished-jobs-automatically)。 TTL 控制器假设资源能在执行完成后的 TTL 秒内被清理，也就是当 TTL 过期后。当 TTL 控制器清理资源时，它将做级联删除，即删除资源对象的同时也删除其依赖对象。注意，当资源被删除时，由该资源的生命周期保证其终结器（finalizers）等被执行。

可以随时设置 TTL 秒。以下是设置 Job 的 .spec.ttlSecondsAfterFinished 字段的一些示例：

* 在资源清单（manifest）中指定此字段，以便作业在完成后的某个时间被自动清除。
* 将此字段设置为存在的、已完成的资源，以采用此新功能。
* 在资源创建时使用 [mutating admission webhook](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks) 动态设置该字段。集群管理员可以使用它对完成的资源强制执行 TTL 策略。
* 使用 [mutating admission webhook](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#admission-webhooks) 在资源完成后动态设置该字段，并根据资源状态、标签等选择不同的 TTL 值。

警告

更新 TTL 秒

请注意，在创建资源后或已经执行结束后，仍可以修改其 TTL 周期，例如作业的 .spec.ttlSecondsAfterFinished 字段。但是，一旦作业变为可被删除状态（当其 TTL 已过期时），即使您通过 API 扩展其 TTL 时长得到了成功的响应，系统也不保证作业将被保留。

时间偏差

由于 TTL 控制器使用存储在 Kubernetes 资源中的时间戳来确定 TTL 是否已过期，因此该功能对集群中的时间偏差很敏感，这可能导致 TTL 控制器在错误的时间清理资源对象。

在 Kubernetes 中，需要在所有节点上运行 NTP（参见[#6159](https://github.com/kubernetes/kubernetes/issues/6159#issuecomment-93844058)）以避免时间偏差。时钟并不总是如此正确，但差异应该很小。设置非零 TTL 时请注意这种风险。

接下来

[自动清理作业](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#clean-up-finished-jobs-automatically)

[设计文档](https://github.com/kubernetes/enhancements/blob/master/keps/sig-apps/0026-ttl-after-finish.md)

反馈

Jobs - Run to Completion

A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.

A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

You can also use a Job to run multiple Pods in parallel.

* [**Running an example Job**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#running-an-example-job)
* [**Writing a Job Spec**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#writing-a-job-spec)
* [**Handling Pod and Container Failures**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#handling-pod-and-container-failures)
* [**Job Termination and Cleanup**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-termination-and-cleanup)
* [**Clean Up Finished Jobs Automatically**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#clean-up-finished-jobs-automatically)
* [**Job Patterns**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-patterns)
* [**Advanced Usage**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#advanced-usage)
* [**Alternatives**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#alternatives)
* [**Cron Jobs**](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#cron-jobs)

Running an example Job

Here is an example Job config. It computes π to 2000 places and prints it out. It takes around 10s to complete.

| [controllers/job.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/controllers/job.yaml) |
| --- |
| apiVersion: batch/v1  kind: Job  metadata:  name: pi  spec:  template:  spec:  containers:  - name: pi  image: perl  command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]  restartPolicy: Never  backoffLimit: 4 |

You can run the example with this command:

kubectl apply -f https://k8s.io/examples/controllers/job.yaml

job.batch/pi created

Check on the status of the Job with kubectl:

kubectl describe jobs/pi

Name: pi

Namespace: default

Selector: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

Labels: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

job-name=pi

Annotations: kubectl.kubernetes.io/last-applied-configuration:

{"apiVersion":"batch/v1","kind":"Job","metadata":{"annotations":{},"name":"pi","namespace":"default"},"spec":{"backoffLimit":4,"template":...

Parallelism: 1

Completions: 1

Start Time: Mon, 02 Dec 2019 15:20:11 +0200

Completed At: Mon, 02 Dec 2019 15:21:16 +0200

Duration: 65s

Pods Statuses: 0 Running / 1 Succeeded / 0 Failed

Pod Template:

Labels: controller-uid=c9948307-e56d-4b5d-8302-ae2d7b7da67c

job-name=pi

Containers:

pi:

Image: perl

Port: <none>

Host Port: <none>

Command:

perl

-Mbignum=bpi

-wle

print bpi(2000)

Environment: <none>

Mounts: <none>

Volumes: <none>

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal SuccessfulCreate 14m job-controller Created pod: pi-5rwd7

To view completed Pods of a Job, use kubectl get pods.

To list all the Pods that belong to a Job in a machine readable form, you can use a command like this:

pods=**$(**kubectl get pods --selector=job-name=pi --output=jsonpath='{.items[\*].metadata.name}'**)**

echo $pods

pi-5rwd7

Here, the selector is the same as the selector for the Job. The --output=jsonpath option specifies an expression that just gets the name from each Pod in the returned list.

View the standard output of one of the pods:

kubectl logs $pods

The output is similar to this:

3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051185480744623799627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922796892589235420199561121290219608640344181598136297747713099605187072113499999983729780499510597317328160963185950244594553469083026425223082533446850352619311881710100031378387528865875332083814206171776691473035982534904287554687311595628638823537875937519577818577805321712268066130019278766111959092164201989380952572010654858632788659361533818279682303019520353018529689957736225994138912497217752834791315155748572424541506959508295331168617278558890750983817546374649393192550604009277016711390098488240128583616035637076601047101819429555961989467678374494482553797747268471040475346462080466842590694912933136770289891521047521620569660240580381501935112533824300355876402474964732639141992726042699227967823547816360093417216412199245863150302861829745557067498385054945885869269956909272107975093029553211653449872027559602364806654991198818347977535663698074265425278625518184175746728909777727938000816470600161452491921732172147723501414419735685481613611573525521334757418494684385233239073941433345477624168625189835694855620992192221842725502542568876717904946016534668049886272327917860857843838279679766814541009538837863609506800642251252051173929848960841284886269456042419652850222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326459958133904780275901

Writing a Job Spec

As with all other Kubernetes config, a Job needs apiVersion, kind, and metadata fields.

A Job also needs a [.spec section](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status).

Pod Template

The .spec.template is the only required field of the .spec.

The .spec.template is a [pod template](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates). It has exactly the same schema as a [pod](https://kubernetes.io/docs/user-guide/pods), except it is nested and does not have an apiVersion or kind.

In addition to required fields for a Pod, a pod template in a Job must specify appropriate labels (see [pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#pod-selector)) and an appropriate restart policy.

Only a [RestartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) equal to Never or OnFailure is allowed.

Pod Selector

The .spec.selector field is optional. In almost all cases you should not specify it. See section [specifying your own pod selector](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#specifying-your-own-pod-selector).

Parallel Jobs

There are three main types of task suitable to run as a Job:

1. Non-parallel Jobs
   * normally, only one Pod is started, unless the Pod fails.
   * the Job is complete as soon as its Pod terminates successfully.
2. Parallel Jobs with a *fixed completion count*:
   * specify a non-zero positive value for .spec.completions.
   * the Job represents the overall task, and is complete when there is one successful Pod for each value in the range 1 to .spec.completions.
   * **not implemented yet:** Each Pod is passed a different index in the range 1 to .spec.completions.
3. Parallel Jobs with a *work queue*:
   * do not specify .spec.completions, default to .spec.parallelism.
   * the Pods must coordinate amongst themselves or an external service to determine what each should work on. For example, a Pod might fetch a batch of up to N items from the work queue.
   * each Pod is independently capable of determining whether or not all its peers are done, and thus that the entire Job is done.
   * when *any* Pod from the Job terminates with success, no new Pods are created.
   * once at least one Pod has terminated with success and all Pods are terminated, then the Job is completed with success.
   * once any Pod has exited with success, no other Pod should still be doing any work for this task or writing any output. They should all be in the process of exiting.

For a *non-parallel* Job, you can leave both .spec.completions and .spec.parallelism unset. When both are unset, both are defaulted to 1.

For a *fixed completion count* Job, you should set .spec.completions to the number of completions needed. You can set .spec.parallelism, or leave it unset and it will default to 1.

For a *work queue* Job, you must leave .spec.completions unset, and set .spec.parallelism to a non-negative integer.

For more information about how to make use of the different types of job, see the [job patterns](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-patterns) section.

Controlling Parallelism

The requested parallelism (.spec.parallelism) can be set to any non-negative value. If it is unspecified, it defaults to 1. If it is specified as 0, then the Job is effectively paused until it is increased.

Actual parallelism (number of pods running at any instant) may be more or less than requested parallelism, for a variety of reasons:

* For *fixed completion count* Jobs, the actual number of pods running in parallel will not exceed the number of remaining completions. Higher values of .spec.parallelism are effectively ignored.
* For *work queue* Jobs, no new Pods are started after any Pod has succeeded – remaining Pods are allowed to complete, however.
* If the Job [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) has not had time to react.
* If the Job controller failed to create Pods for any reason (lack of ResourceQuota, lack of permission, etc.), then there may be fewer pods than requested.
* The Job controller may throttle new Pod creation due to excessive previous pod failures in the same Job.
* When a Pod is gracefully shut down, it takes time to stop.

Handling Pod and Container Failures

A container in a Pod may fail for a number of reasons, such as because the process in it exited with a non-zero exit code, or the container was killed for exceeding a memory limit, etc. If this happens, and the .spec.template.spec.restartPolicy = "OnFailure", then the Pod stays on the node, but the container is re-run. Therefore, your program needs to handle the case when it is restarted locally, or else specify .spec.template.spec.restartPolicy = "Never". See [pod lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#example-states) for more information on restartPolicy.

An entire Pod can also fail, for a number of reasons, such as when the pod is kicked off the node (node is upgraded, rebooted, deleted, etc.), or if a container of the Pod fails and the .spec.template.spec.restartPolicy = "Never". When a Pod fails, then the Job controller starts a new Pod. This means that your application needs to handle the case when it is restarted in a new pod. In particular, it needs to handle temporary files, locks, incomplete output and the like caused by previous runs.

Note that even if you specify .spec.parallelism = 1 and .spec.completions = 1 and .spec.template.spec.restartPolicy = "Never", the same program may sometimes be started twice.

If you do specify .spec.parallelism and .spec.completions both greater than 1, then there may be multiple pods running at once. Therefore, your pods must also be tolerant of concurrency.

Pod backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set .spec.backoffLimit to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s …) capped at six minutes. The back-off count is reset if no new failed Pods appear before the Job’s next status check.

**Note:** Issue [#54870](https://github.com/kubernetes/kubernetes/issues/54870) still exists for versions of Kubernetes prior to version 1.12

**Note:** If your job has restartPolicy = "OnFailure", keep in mind that your container running the Job will be terminated once the job backoff limit has been reached. This can make debugging the Job’s executable more difficult. We suggest setting restartPolicy = "Never" when debugging the Job or using a logging system to ensure output from failed Jobs is not lost inadvertently.

Job Termination and Cleanup

When a Job completes, no more Pods are created, but the Pods are not deleted either. Keeping them around allows you to still view the logs of completed pods to check for errors, warnings, or other diagnostic output. The job object also remains after it is completed so that you can view its status. It is up to the user to delete old jobs after noting their status. Delete the job with kubectl (e.g. kubectl delete jobs/pi or kubectl delete -f ./job.yaml). When you delete the job using kubectl, all the pods it created are deleted too.

By default, a Job will run uninterrupted unless a Pod fails (restartPolicy=Never) or a Container exits in error (restartPolicy=OnFailure), at which point the Job defers to the .spec.backoffLimit described above. Once .spec.backoffLimit has been reached the Job will be marked as failed and any running Pods will be terminated.

Another way to terminate a Job is by setting an active deadline. Do this by setting the .spec.activeDeadlineSeconds field of the Job to a number of seconds. The activeDeadlineSeconds applies to the duration of the job, no matter how many Pods are created. Once a Job reaches activeDeadlineSeconds, all of its running Pods are terminated and the Job status will become type: Failed with reason: DeadlineExceeded.

Note that a Job’s .spec.activeDeadlineSeconds takes precedence over its .spec.backoffLimit. Therefore, a Job that is retrying one or more failed Pods will not deploy additional Pods once it reaches the time limit specified by activeDeadlineSeconds, even if the backoffLimit is not yet reached.

Example:

apiVersion: batch/v1

kind: Job

metadata:

name: pi-with-timeout

spec:

backoffLimit: 5

activeDeadlineSeconds: 100

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

Note that both the Job spec and the [Pod template spec](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#detailed-behavior) within the Job have an activeDeadlineSeconds field. Ensure that you set this field at the proper level.

Clean Up Finished Jobs Automatically

Finished Jobs are usually no longer needed in the system. Keeping them around in the system will put pressure on the API server. If the Jobs are managed directly by a higher level controller, such as [CronJobs](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/), the Jobs can be cleaned up by CronJobs based on the specified capacity-based cleanup policy.

TTL Mechanism for Finished Jobs

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/)

Another way to clean up finished Jobs (either Complete or Failed) automatically is to use a TTL mechanism provided by a [TTL controller](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/) for finished resources, by specifying the .spec.ttlSecondsAfterFinished field of the Job.

When the TTL controller cleans up the Job, it will delete the Job cascadingly, i.e. delete its dependent objects, such as Pods, together with the Job. Note that when the Job is deleted, its lifecycle guarantees, such as finalizers, will be honored.

For example:

apiVersion: batch/v1

kind: Job

metadata:

name: pi-with-ttl

spec:

ttlSecondsAfterFinished: 100

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

The Job pi-with-ttl will be eligible to be automatically deleted, 100 seconds after it finishes.

If the field is set to 0, the Job will be eligible to be automatically deleted immediately after it finishes. If the field is unset, this Job won’t be cleaned up by the TTL controller after it finishes.

Note that this TTL mechanism is alpha, with feature gate TTLAfterFinished. For more information, see the documentation for [TTL controller](https://kubernetes.io/docs/concepts/workloads/controllers/ttlafterfinished/) for finished resources.

Job Patterns

The Job object can be used to support reliable parallel execution of Pods. The Job object is not designed to support closely-communicating parallel processes, as commonly found in scientific computing. It does support parallel processing of a set of independent but related *work items*. These might be emails to be sent, frames to be rendered, files to be transcoded, ranges of keys in a NoSQL database to scan, and so on.

In a complex system, there may be multiple different sets of work items. Here we are just considering one set of work items that the user wants to manage together — a *batch job*.

There are several different patterns for parallel computation, each with strengths and weaknesses. The tradeoffs are:

* One Job object for each work item, vs. a single Job object for all work items. The latter is better for large numbers of work items. The former creates some overhead for the user and for the system to manage large numbers of Job objects.
* Number of pods created equals number of work items, vs. each Pod can process multiple work items. The former typically requires less modification to existing code and containers. The latter is better for large numbers of work items, for similar reasons to the previous bullet.
* Several approaches use a work queue. This requires running a queue service, and modifications to the existing program or container to make it use the work queue. Other approaches are easier to adapt to an existing containerised application.

The tradeoffs are summarized here, with columns 2 to 4 corresponding to the above tradeoffs. The pattern names are also links to examples and more detailed description.

| Pattern | Single Job object | Fewer pods than work items? | Use app unmodified? | Works in Kube 1.1? |
| --- | --- | --- | --- | --- |
| [Job Template Expansion](https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/) |  |  | ✓ | ✓ |
| [Queue with Pod Per Work Item](https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/) | ✓ |  | sometimes | ✓ |
| [Queue with Variable Pod Count](https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/) | ✓ | ✓ |  | ✓ |
| Single Job with Static Work Assignment | ✓ |  | ✓ |  |

When you specify completions with .spec.completions, each Pod created by the Job controller has an identical [spec](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status). This means that all pods for a task will have the same command line and the same image, the same volumes, and (almost) the same environment variables. These patterns are different ways to arrange for pods to work on different things.

This table shows the required settings for .spec.parallelism and .spec.completions for each of the patterns. Here, W is the number of work items.

| Pattern | .spec.completions | .spec.parallelism |
| --- | --- | --- |
| [Job Template Expansion](https://kubernetes.io/docs/tasks/job/parallel-processing-expansion/) | 1 | should be 1 |
| [Queue with Pod Per Work Item](https://kubernetes.io/docs/tasks/job/coarse-parallel-processing-work-queue/) | W | any |
| [Queue with Variable Pod Count](https://kubernetes.io/docs/tasks/job/fine-parallel-processing-work-queue/) | 1 | any |
| Single Job with Static Work Assignment | W | any |

Advanced Usage

Specifying your own pod selector

Normally, when you create a Job object, you do not specify .spec.selector. The system defaulting logic adds this field when the Job is created. It picks a selector value that will not overlap with any other jobs.

However, in some cases, you might need to override this automatically set selector. To do this, you can specify the .spec.selector of the Job.

Be very careful when doing this. If you specify a label selector which is not unique to the pods of that Job, and which matches unrelated Pods, then pods of the unrelated job may be deleted, or this Job may count other Pods as completing it, or one or both Jobs may refuse to create Pods or run to completion. If a non-unique selector is chosen, then other controllers (e.g. ReplicationController) and their Pods may behave in unpredictable ways too. Kubernetes will not stop you from making a mistake when specifying .spec.selector.

Here is an example of a case when you might want to use this feature.

Say Job old is already running. You want existing Pods to keep running, but you want the rest of the Pods it creates to use a different pod template and for the Job to have a new name. You cannot update the Job because these fields are not updatable. Therefore, you delete Job old but *leave its pods running*, using kubectl delete jobs/old --cascade=false. Before deleting it, you make a note of what selector it uses:

kubectl get job old -o yaml

kind: Job

metadata:

name: old

...

spec:

selector:

matchLabels:

controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002

...

Then you create a new Job with name new and you explicitly specify the same selector. Since the existing Pods have label controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002, they are controlled by Job new as well.

You need to specify manualSelector: true in the new Job since you are not using the selector that the system normally generates for you automatically.

kind: Job

metadata:

name: new

...

spec:

manualSelector: true

selector:

matchLabels:

controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002

...

The new Job itself will have a different uid from a8f3d00d-c6d2-11e5-9f87-42010af00002. Setting manualSelector: true tells the system to that you know what you are doing and to allow this mismatch.

Alternatives

Bare Pods

When the node that a Pod is running on reboots or fails, the pod is terminated and will not be restarted. However, a Job will create new Pods to replace terminated ones. For this reason, we recommend that you use a Job rather than a bare Pod, even if your application requires only a single Pod.

Replication Controller

Jobs are complementary to [Replication Controllers](https://kubernetes.io/docs/user-guide/replication-controller). A Replication Controller manages Pods which are not expected to terminate (e.g. web servers), and a Job manages Pods that are expected to terminate (e.g. batch tasks).

As discussed in [Pod Lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/), Job is *only* appropriate for pods with RestartPolicy equal to OnFailure or Never. (Note: If RestartPolicy is not set, the default value is Always.)

Single Job starts Controller Pod

Another pattern is for a single Job to create a Pod which then creates other Pods, acting as a sort of custom controller for those Pods. This allows the most flexibility, but may be somewhat complicated to get started with and offers less integration with Kubernetes.

One example of this pattern would be a Job which starts a Pod which runs a script that in turn starts a Spark master controller (see [spark example](https://github.com/kubernetes/examples/tree/master/staging/spark/README.md)), runs a spark driver, and then cleans up.

An advantage of this approach is that the overall process gets the completion guarantee of a Job object, but complete control over what Pods are created and how work is assigned to them.

Cron Jobs

You can use a [CronJob](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/) to create a Job that will run at specified times/dates, similar to the Unix tool cron.

Feedback

CronJob

**FEATURE STATE:** Kubernetes v1.8 [beta](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/)

A *Cron Job* creates [Jobs](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/) on a time-based schedule.

One CronJob object is like one line of a *crontab* (cron table) file. It runs a job periodically on a given schedule, written in [Cron](https://en.wikipedia.org/wiki/Cron) format.

**Note:** All **CronJob** schedule: times are based on the timezone of the master where the job is initiated.

For instructions on creating and working with cron jobs, and for an example of a spec file for a cron job, see [Running automated tasks with cron jobs](https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs).

* [**Cron Job Limitations**](https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/#cron-job-limitations)

Cron Job Limitations

A cron job creates a job object *about* once per execution time of its schedule. We say “about” because there are certain circumstances where two jobs might be created, or no job might be created. We attempt to make these rare, but do not completely prevent them. Therefore, jobs should be *idempotent*.

If startingDeadlineSeconds is set to a large value or left unset (the default) and if concurrencyPolicy is set to Allow, the jobs will always run at least once.

For every CronJob, the CronJob [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) checks how many schedules it missed in the duration from its last scheduled time until now. If there are more than 100 missed schedules, then it does not start the job and logs the error

Cannot determine if job needs to be started. Too many missed start time (> 100). Set or decrease .spec.startingDeadlineSeconds or check clock skew.

It is important to note that if the startingDeadlineSeconds field is set (not nil), the controller counts how many missed jobs occurred from the value of startingDeadlineSeconds until now rather than from the last scheduled time until now. For example, if startingDeadlineSeconds is 200, the controller counts how many missed jobs occurred in the last 200 seconds.

A CronJob is counted as missed if it has failed to be created at its scheduled time. For example, If concurrencyPolicy is set to Forbid and a CronJob was attempted to be scheduled when there was a previous schedule still running, then it would count as missed.

For example, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its startingDeadlineSeconds field is not set. If the CronJob controller happens to be down from 08:29:00 to 10:21:00, the job will not start as the number of missed jobs which missed their schedule is greater than 100.

To illustrate this concept further, suppose a CronJob is set to schedule a new Job every one minute beginning at 08:30:00, and its startingDeadlineSeconds is set to 200 seconds. If the CronJob controller happens to be down for the same period as the previous example (08:29:00 to 10:21:00,) the Job will still start at 10:22:00. This happens as the controller now checks how many missed schedules happened in the last 200 seconds (ie, 3 missed schedules), rather than from the last scheduled time until now.

The CronJob is only responsible for creating Jobs that match its schedule, and the Job in turn is responsible for the management of the Pods it represents.

Feedback

Deployments

A *Deployment* controller provides declarative updates for [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) and [ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/).

You describe a *desired state* in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

**注意：** You should not manage ReplicaSets owned by a Deployment. All the use cases should be covered by manipulating the Deployment object. Consider opening an issue in the main Kubernetes repository if your use case is not covered below.

* [**Use Case**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#use-case)
* [**Creating a Deployment**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#creating-a-deployment)
* [**Updating a Deployment**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#updating-a-deployment)
* [**Rolling Back a Deployment**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment)
* [**Scaling a Deployment**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment)
* [**Pausing and Resuming a Deployment**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment)
* [**Deployment status**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#deployment-status)
* [**Clean up Policy**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#clean-up-policy)
* [**Use Cases**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#use-cases)
* [**Writing a Deployment Spec**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#writing-a-deployment-spec)
* [**Alternative to Deployments**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#alternative-to-deployments)

Use Case

The following are typical use cases for Deployments:

* [Use Case](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#use-case)
* [Creating a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#creating-a-deployment)
  + [Pod-template-hash label](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#pod-template-hash-label)
* [Updating a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#updating-a-deployment)
  + [Rollover (aka multiple updates in-flight)](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rollover-aka-multiple-updates-in-flight)
  + [Label selector updates](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#label-selector-updates)
* [Rolling Back a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment)
  + [Checking Rollout History of a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#checking-rollout-history-of-a-deployment)
  + [Rolling Back to a Previous Revision](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rolling-back-to-a-previous-revision)
* [Scaling a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment)
  + [Proportional scaling](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#proportional-scaling)
* [Pausing and Resuming a Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment)
* [Deployment status](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#deployment-status)
  + [Progressing Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#progressing-deployment)
  + [Complete Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#complete-deployment)
  + [Failed Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#failed-deployment)
  + [Operating on a failed deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#operating-on-a-failed-deployment)
* [Clean up Policy](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#clean-up-policy)
* [Use Cases](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#use-cases)
  + [Canary Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#canary-deployment)
* [Writing a Deployment Spec](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#writing-a-deployment-spec)
  + [Pod Template](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#pod-template)
  + [Replicas](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#replicas)
  + [Selector](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#selector)
  + [Strategy](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#strategy)
  + [Recreate Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#recreate-deployment)
  + [Rolling Update Deployment](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment)
    - [Max Unavailable](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#max-unavailable)
    - [Max Surge](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#max-surge)
  + [Progress Deadline Seconds](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds)
  + [Min Ready Seconds](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#min-ready-seconds)
  + [Rollback To](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rollback-to)
  + [Revision](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#revision)
  + [Revision History Limit](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#revision-history-limit)
  + [Paused](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#paused)
* [Alternative to Deployments](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#alternative-to-deployments)
  + [kubectl rolling update](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#kubectl-rolling-update)

Creating a Deployment

Here is an example Deployment. It creates a ReplicaSet to bring up three nginx Pods.

| [controllers/nginx-deployment.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/nginx-deployment.yaml) |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: nginx-deployment  labels:  app: nginx  spec:  replicas: 3  selector:  matchLabels:  app: nginx  template:  metadata:  labels:  app: nginx  spec:  containers:  - name: nginx  image: nginx:1.15.4  ports:  - containerPort: 80 |

Run the example by downloading the example file and then running this command:

kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml

Setting the kubectl flag --record to true allows you to record current command in the annotations of the resources being created or updated. It is useful for future introspection: for example, to see the commands executed in each Deployment revision.

Then running get immediately will give:

$ kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 0 0 0 1s

This indicates that the Deployment’s number of desired replicas is 3 (according to deployment’s .spec.replicas), the number of current replicas (.status.replicas) is 0, the number of up-to-date replicas (.status.updatedReplicas) is 0, and the number of available replicas (.status.availableReplicas) is also 0.

To see the Deployment rollout status, run:

$ kubectl rollout status deployment/nginx-deployment

Waiting **for** rollout to finish: 2 out of 3 new replicas have been updated...

deployment "nginx-deployment" successfully rolled out

Running the get again a few seconds later should give:

$ kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 18s

This indicates that the Deployment has created all three replicas, and all replicas are up-to-date (contains the latest pod template) and available (pod status is ready for at least Deployment’s .spec.minReadySeconds). Running kubectl get rs and kubectl get pods will show the ReplicaSet (RS) and Pods created.

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-2035384211 3 3 3 18s

You may notice that the name of the ReplicaSet is always <the name of the Deployment>-<hash value of the pod template>.

$ kubectl get pods --show-labels

NAME READY STATUS RESTARTS AGE LABELS

nginx-deployment-2035384211-7ci7o 1/1 Running 0 18s app=nginx,pod-template-hash=2035384211

nginx-deployment-2035384211-kzszj 1/1 Running 0 18s app=nginx,pod-template-hash=2035384211

nginx-deployment-2035384211-qqcnn 1/1 Running 0 18s app=nginx,pod-template-hash=2035384211

The created ReplicaSet ensures that there are three nginx Pods at all times.

**注意：** You must specify an appropriate selector and pod template labels in a Deployment (in this case, app = nginx). That is, don’t overlap with other controllers (including other Deployments, ReplicaSets, StatefulSets, etc.). Kubernetes doesn’t stop you from overlapping, and if multiple controllers have overlapping selectors, those controllers may fight with each other and won’t behave correctly.

Pod-template-hash label

**注意：** Do not change this label.

Note the pod-template-hash label in the example output in the pod labels above. This label is added by the Deployment controller to every ReplicaSet that a Deployment creates or adopts. Its purpose is to make sure that child ReplicaSets of a Deployment do not overlap. It is computed by hashing the PodTemplate of the ReplicaSet and using the resulting hash as the label value that will be added in the ReplicaSet selector, pod template labels, and in any existing Pods that the ReplicaSet may have.

Updating a Deployment

**注意：** A Deployment’s rollout is triggered if and only if the Deployment’s pod template (that is, .spec.template) is changed, for example if the labels or container images of the template are updated. Other updates, such as scaling the Deployment, do not trigger a rollout.

Suppose that we now want to update the nginx Pods to use the nginx:1.9.1 image instead of the nginx:1.7.9 image.

$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1

deployment "nginx-deployment" image updated

Alternatively, we can edit the Deployment and change .spec.template.spec.containers[0].image from nginx:1.7.9 to nginx:1.9.1:

$ kubectl edit deployment/nginx-deployment

deployment "nginx-deployment" edited

To see the rollout status, run:

$ kubectl rollout status deployment/nginx-deployment

Waiting **for** rollout to finish: 2 out of 3 new replicas have been updated...

deployment "nginx-deployment" successfully rolled out

After the rollout succeeds, you may want to get the Deployment:

$ kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 36s

The number of up-to-date replicas indicates that the Deployment has updated the replicas to the latest configuration. The current replicas indicates the total replicas this Deployment manages, and the available replicas indicates the number of current replicas that are available.

We can run kubectl get rs to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-1564180365 3 3 3 6s

nginx-deployment-2035384211 0 0 0 36s

Running get pods should now show only the new Pods:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-deployment-1564180365-khku8 1/1 Running 0 14s

nginx-deployment-1564180365-nacti 1/1 Running 0 14s

nginx-deployment-1564180365-z9gth 1/1 Running 0 14s

Next time we want to update these Pods, we only need to update the Deployment’s pod template again.

Deployment can ensure that only a certain number of Pods may be down while they are being updated. By default, it ensures that at least 1 less than the desired number of Pods are up (1 max unavailable).

Deployment can also ensure that only a certain number of Pods may be created above the desired number of Pods. By default, it ensures that at most 1 more than the desired number of Pods are up (1 max surge).

In a future version of Kubernetes, the defaults will change from 1-1 to 25%-25%.

For example, if you look at the above Deployment closely, you will see that it first created a new Pod, then deleted some old Pods and created new ones. It does not kill old Pods until a sufficient number of new Pods have come up, and does not create new Pods until a sufficient number of old Pods have been killed. It makes sure that the number of available Pods is at least 2 and the number of total Pods is at most 4.

$ kubectl describe deployments

Name: nginx-deployment

Namespace: default

CreationTimestamp: Tue, 15 Mar 2016 12:01:06 -0700

Labels: app=nginx

Selector: app=nginx

Replicas: 3 updated | 3 total | 3 available | 0 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

OldReplicaSets: <none>

NewReplicaSet: nginx-deployment-1564180365 (3/3 replicas created)

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

36s 36s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-2035384211 to 3

23s 23s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 1

23s 23s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 2

23s 23s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 2

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 0

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

Here we see that when we first created the Deployment, it created a ReplicaSet (nginx-deployment-2035384211) and scaled it up to 3 replicas directly. When we updated the Deployment, it created a new ReplicaSet (nginx-deployment-1564180365) and scaled it up to 1 and then scaled down the old ReplicaSet to 2, so that at least 2 Pods were available and at most 4 Pods were created at all times. It then continued scaling up and down the new and the old ReplicaSet, with the same rolling update strategy. Finally, we’ll have 3 available replicas in the new ReplicaSet, and the old ReplicaSet is scaled down to 0.

Rollover (aka multiple updates in-flight)

Each time a new deployment object is observed by the deployment controller, a ReplicaSet is created to bring up the desired Pods if there is no existing ReplicaSet doing so. Existing ReplicaSet controlling Pods whose labels match .spec.selector but whose template does not match .spec.template are scaled down. Eventually, the new ReplicaSet will be scaled to .spec.replicas and all old ReplicaSets will be scaled to 0.

If you update a Deployment while an existing rollout is in progress, the Deployment will create a new ReplicaSet as per the update and start scaling that up, and will roll over the ReplicaSet that it was scaling up previously – it will add it to its list of old ReplicaSets and will start scaling it down.

For example, suppose you create a Deployment to create 5 replicas of nginx:1.7.9, but then updates the Deployment to create 5 replicas of nginx:1.9.1, when only 3 replicas of nginx:1.7.9 had been created. In that case, Deployment will immediately start killing the 3 nginx:1.7.9 Pods that it had created, and will start creating nginx:1.9.1 Pods. It will not wait for 5 replicas of nginx:1.7.9 to be created before changing course.

Label selector updates

It is generally discouraged to make label selector updates and it is suggested to plan your selectors up front. In any case, if you need to perform a label selector update, exercise great caution and make sure you have grasped all of the implications.

* Selector additions require the pod template labels in the Deployment spec to be updated with the new label too, otherwise a validation error is returned. This change is a non-overlapping one, meaning that the new selector does not select ReplicaSets and Pods created with the old selector, resulting in orphaning all old ReplicaSets and creating a new ReplicaSet.
* Selector updates – that is, changing the existing value in a selector key – result in the same behavior as additions.
* Selector removals – that is, removing an existing key from the Deployment selector – do not require any changes in the pod template labels. No existing ReplicaSet is orphaned, and a new ReplicaSet is not created, but note that the removed label still exists in any existing Pods and ReplicaSets.

Rolling Back a Deployment

Sometimes you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping. By default, all of the Deployment’s rollout history is kept in the system so that you can rollback anytime you want (you can change that by modifying revision history limit).

**注意：** A Deployment’s revision is created when a Deployment’s rollout is triggered. This means that the new revision is created if and only if the Deployment’s pod template (.spec.template) is changed, for example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision, so that we can facilitate simultaneous manual- or auto-scaling. This means that when you roll back to an earlier revision, only the Deployment’s pod template part is rolled back.

Suppose that we made a typo while updating the Deployment, by putting the image name as nginx:1.91 instead of nginx:1.9.1:

$ kubectl set image deployment/nginx-deployment nginx=nginx:1.91

deployment "nginx-deployment" image updated

The rollout will be stuck.

$ kubectl rollout status deployments nginx-deployment

Waiting **for** rollout to finish: 2 out of 3 new replicas have been updated...

Press Ctrl-C to stop the above rollout status watch. For more information on stuck rollouts, [read more here](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#deployment-status).

You will also see that both the number of old replicas (nginx-deployment-1564180365 and nginx-deployment-2035384211) and new replicas (nginx-deployment-3066724191) are 2.

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-1564180365 2 2 0 25s

nginx-deployment-2035384211 0 0 0 36s

nginx-deployment-3066724191 2 2 2 6s

Looking at the Pods created, you will see that the 2 Pods created by new ReplicaSet are stuck in an image pull loop.

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

nginx-deployment-1564180365-70iae 1/1 Running 0 25s

nginx-deployment-1564180365-jbqqo 1/1 Running 0 25s

nginx-deployment-3066724191-08mng 0/1 ImagePullBackOff 0 6s

nginx-deployment-3066724191-eocby 0/1 ImagePullBackOff 0 6s

**注意：** The Deployment controller will stop the bad rollout automatically, and will stop scaling up the new ReplicaSet. This depends on the rollingUpdate parameters (maxUnavailable specifically) that you have specified. Kubernetes by default sets the value to 1 and spec.replicas to 1 so if you haven’t cared about setting those parameters, your Deployment can have 100% unavailability by default! This will be fixed in Kubernetes in a future version.

$ kubectl describe deployment

Name: nginx-deployment

Namespace: default

CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700

Labels: app=nginx

Selector: app=nginx

Replicas: 2 updated | 3 total | 2 available | 2 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

OldReplicaSets: nginx-deployment-1564180365 (2/2 replicas created)

NewReplicaSet: nginx-deployment-3066724191 (2/2 replicas created)

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

1m 1m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-2035384211 to 3

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 1

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 2

22s 22s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 2

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 0

21s 21s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 1

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-1564180365 to 2

13s 13s 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 2

To fix this, we need to rollback to a previous revision of Deployment that is stable.

Checking Rollout History of a Deployment

First, check the revisions of this deployment:

$ kubectl rollout history deployment/nginx-deployment

deployments "nginx-deployment"

REVISION CHANGE-CAUSE

1 kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml

2 kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1

3 kubectl set image deployment/nginx-deployment nginx=nginx:1.91

Because we recorded the command while creating this Deployment using --record, we can easily see the changes we made in each revision.

To further see the details of each revision, run:

$ kubectl rollout history deployment/nginx-deployment --revision=2

deployments "nginx-deployment" revision 2

Labels: app=nginx

pod-template-hash=1159050644

Annotations: kubernetes.io/change-cause=kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1

Containers:

nginx:

Image: nginx:1.9.1

Port: 80/TCP

QoS Tier:

cpu: BestEffort

memory: BestEffort

Environment Variables: <none>

No volumes.

Rolling Back to a Previous Revision

Now we’ve decided to undo the current rollout and rollback to the previous revision:

$ kubectl rollout undo deployment/nginx-deployment

deployment "nginx-deployment" rolled back

Alternatively, you can rollback to a specific revision by specifying it with --to-revision:

$ kubectl rollout undo deployment/nginx-deployment --to-revision=2

deployment "nginx-deployment" rolled back

For more details about rollout related commands, read [kubectl rollout](https://kubernetes.io/docs/user-guide/kubectl/v1.17/#rollout).

The Deployment is now rolled back to a previous stable revision. As you can see, a DeploymentRollback event for rolling back to revision 2 is generated from Deployment controller.

$ kubectl get deployment

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 3 3 3 3 30m

$ kubectl describe deployment

Name: nginx-deployment

Namespace: default

CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700

Labels: app=nginx

Selector: app=nginx

Replicas: 3 updated | 3 total | 3 available | 0 unavailable

StrategyType: RollingUpdate

MinReadySeconds: 0

RollingUpdateStrategy: 1 max unavailable, 1 max surge

OldReplicaSets: <none>

NewReplicaSet: nginx-deployment-1564180365 (3/3 replicas created)

Events:

FirstSeen LastSeen Count From SubobjectPath Type Reason Message

--------- -------- ----- ---- ------------- -------- ------ -------

30m 30m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-2035384211 to 3

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 1

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 2

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 2

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-2035384211 to 0

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 2

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-3066724191 to 1

29m 29m 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-1564180365 to 2

2m 2m 1 {deployment-controller } Normal ScalingReplicaSet Scaled down replica set nginx-deployment-3066724191 to 0

2m 2m 1 {deployment-controller } Normal DeploymentRollback Rolled back deployment "nginx-deployment" to revision 2

29m 2m 2 {deployment-controller } Normal ScalingReplicaSet Scaled up replica set nginx-deployment-1564180365 to 3

Scaling a Deployment

You can scale a Deployment by using the following command:

$ kubectl scale deployment nginx-deployment --replicas=10

deployment "nginx-deployment" scaled

Assuming [horizontal pod autoscaling](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/) is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.

$ kubectl autoscale deployment nginx-deployment --min=10 --max=15 --cpu-percent=80

deployment "nginx-deployment" autoscaled

Proportional scaling

RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in progress or paused), then the Deployment controller will balance the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called *proportional scaling*.

For example, you are running a Deployment with 10 replicas, [maxSurge](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#max-surge)=3, and [maxUnavailable](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#max-unavailable)=2.

$ kubectl get deploy

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 10 10 10 10 50s

You update to a new image which happens to be unresolvable from inside the cluster.

$ kubectl set image deploy/nginx-deployment nginx=nginx:sometag

deployment "nginx-deployment" image updated

The image update starts a new rollout with ReplicaSet nginx-deployment-1989198191, but it’s blocked due to the maxUnavailable requirement that we mentioned above.

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-1989198191 5 5 0 9s

nginx-deployment-618515232 8 8 8 1m

Then a new scaling request for the Deployment comes along. The autoscaler increments the Deployment replicas to 15. The Deployment controller needs to decide where to add these new 5 replicas. If we weren’t using proportional scaling, all 5 of them would be added in the new ReplicaSet. With proportional scaling, we spread the additional replicas across all ReplicaSets. Bigger proportions go to the ReplicaSets with the most replicas and lower proportions go to ReplicaSets with less replicas. Any leftovers are added to the ReplicaSet with the most replicas. ReplicaSets with zero replicas are not scaled up.

In our example above, 3 replicas will be added to the old ReplicaSet and 2 replicas will be added to the new ReplicaSet. The rollout process should eventually move all replicas to the new ReplicaSet, assuming the new replicas become healthy.

$ kubectl get deploy

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx-deployment 15 18 7 8 7m

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-deployment-1989198191 7 7 0 7m

nginx-deployment-618515232 11 11 11 7m

Pausing and Resuming a Deployment

You can pause a Deployment before triggering one or more updates and then resume it. This will allow you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts.

For example, with a Deployment that was just created:

$ kubectl get deploy

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

nginx 3 3 3 3 1m

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-2142116321 3 3 3 1m

Pause by running the following command:

$ kubectl rollout pause deployment/nginx-deployment

deployment "nginx-deployment" paused

Then update the image of the Deployment:

$ kubectl set image deploy/nginx-deployment nginx=nginx:1.9.1

deployment "nginx-deployment" image updated

Notice that no new rollout started:

$ kubectl rollout history deploy/nginx-deployment

deployments "nginx"

REVISION CHANGE-CAUSE

1 <none>

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-2142116321 3 3 3 2m

You can make as many updates as you wish, for example, update the resources that will be used:

$ kubectl set resources deployment nginx -c=nginx --limits=cpu=200m,memory=512Mi

deployment "nginx" resource requirements updated

The initial state of the Deployment prior to pausing it will continue its function, but new updates to the Deployment will not have any effect as long as the Deployment is paused.

Eventually, resume the Deployment and observe a new ReplicaSet coming up with all the new updates:

$ kubectl rollout resume deploy/nginx-deployment

deployment "nginx" resumed

$ kubectl get rs -w

NAME DESIRED CURRENT READY AGE

nginx-2142116321 2 2 2 2m

nginx-3926361531 2 2 0 6s

nginx-3926361531 2 2 1 18s

nginx-2142116321 1 2 2 2m

nginx-2142116321 1 2 2 2m

nginx-3926361531 3 2 1 18s

nginx-3926361531 3 2 1 18s

nginx-2142116321 1 1 1 2m

nginx-3926361531 3 3 1 18s

nginx-3926361531 3 3 2 19s

nginx-2142116321 0 1 1 2m

nginx-2142116321 0 1 1 2m

nginx-2142116321 0 0 0 2m

nginx-3926361531 3 3 3 20s

^C

$ kubectl get rs

NAME DESIRED CURRENT READY AGE

nginx-2142116321 0 0 0 2m

nginx-3926361531 3 3 3 28s

**注意：** You cannot rollback a paused Deployment until you resume it.

Deployment status

A Deployment enters various states during its lifecycle. It can be [progressing](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#progressing-deployment) while rolling out a new ReplicaSet, it can be [complete](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#complete-deployment), or it can [fail to progress](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#failed-deployment).

Progressing Deployment

Kubernetes marks a Deployment as *progressing* when one of the following tasks is performed:

* The Deployment creates a new ReplicaSet.
* The Deployment is scaling up its newest ReplicaSet.
* The Deployment is scaling down its older ReplicaSet(s).
* New Pods become ready or available (ready for at least [MinReadySeconds](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#min-ready-seconds)).

You can monitor the progress for a Deployment by using kubectl rollout status.

Complete Deployment

Kubernetes marks a Deployment as *complete* when it has the following characteristics:

* All of the replicas associated with the Deployment have been updated to the latest version you’ve specified, meaning any updates you’ve requested have been completed.
* All of the replicas associated with the Deployment are available.
* No old replicas for the Deployment are running.

You can check if a Deployment has completed by using kubectl rollout status. If the rollout completed successfully, kubectl rollout status returns a zero exit code.

$ kubectl rollout status deploy/nginx-deployment

Waiting **for** rollout to finish: 2 of 3 updated replicas are available...

deployment "nginx" successfully rolled out

$ echo $?

0

Failed Deployment

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

* Insufficient quota
* Readiness probe failures
* Image pull errors
* Insufficient permissions
* Limit ranges
* Application runtime misconfiguration

One way you can detect this condition is to specify a deadline parameter in your Deployment spec: ([spec.progressDeadlineSeconds](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#progress-deadline-seconds)). spec.progressDeadlineSeconds denotes the number of seconds the Deployment controller waits before indicating (in the Deployment status) that the Deployment progress has stalled.

The following kubectl command sets the spec with progressDeadlineSeconds to make the controller report lack of progress for a Deployment after 10 minutes:

$ kubectl patch deployment/nginx-deployment -p '{"spec":{"progressDeadlineSeconds":600}}'

"nginx-deployment" patched

Once the deadline has been exceeded, the Deployment controller adds a DeploymentCondition with the following attributes to the Deployment’s status.conditions:

* Type=Progressing
* Status=False
* Reason=ProgressDeadlineExceeded

See the [Kubernetes API conventions](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#typical-status-properties) for more information on status conditions.

**注意：** Kubernetes will take no action on a stalled Deployment other than to report a status condition with Reason=ProgressDeadlineExceeded. Higher level orchestrators can take advantage of it and act accordingly, for example, rollback the Deployment to its previous version.

**注意：** If you pause a Deployment, Kubernetes does not check progress against your specified deadline. You can safely pause a Deployment in the middle of a rollout and resume without triggering the condition for exceeding the deadline.

You may experience transient errors with your Deployments, either due to a low timeout that you have set or due to any other kind of error that can be treated as transient. For example, let’s suppose you have insufficient quota. If you describe the Deployment you will notice the following section:

$ kubectl describe deployment nginx-deployment

<...>

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True ReplicaSetUpdated

ReplicaFailure True FailedCreate

<...>

If you run kubectl get deployment nginx-deployment -o yaml, the Deployment status might look like this:

status:

availableReplicas: 2

conditions:

- lastTransitionTime: 2016-10-04T12:25:39Z

lastUpdateTime: 2016-10-04T12:25:39Z

message: Replica set "nginx-deployment-4262182780" is progressing.

reason: ReplicaSetUpdated

status: "True"

type: Progressing

- lastTransitionTime: 2016-10-04T12:25:42Z

lastUpdateTime: 2016-10-04T12:25:42Z

message: Deployment has minimum availability.

reason: MinimumReplicasAvailable

status: "True"

type: Available

- lastTransitionTime: 2016-10-04T12:25:39Z

lastUpdateTime: 2016-10-04T12:25:39Z

message: 'Error creating: pods "nginx-deployment-4262182780-" is forbidden: exceeded quota:

object-counts, requested: pods=1, used: pods=3, limited: pods=2'

reason: FailedCreate

status: "True"

type: ReplicaFailure

observedGeneration: 3

replicas: 2

unavailableReplicas: 2

Eventually, once the Deployment progress deadline is exceeded, Kubernetes updates the status and the reason for the Progressing condition:

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing False ProgressDeadlineExceeded

ReplicaFailure True FailedCreate

You can address an issue of insufficient quota by scaling down your Deployment, by scaling down other controllers you may be running, or by increasing quota in your namespace. If you satisfy the quota conditions and the Deployment controller then completes the Deployment rollout, you’ll see the Deployment’s status update with a successful condition (Status=True and Reason=NewReplicaSetAvailable).

Conditions:

Type Status Reason

---- ------ ------

Available True MinimumReplicasAvailable

Progressing True NewReplicaSetAvailable

Type=Available with Status=True means that your Deployment has minimum availability. Minimum availability is dictated by the parameters specified in the deployment strategy. Type=Progressing with Status=True means that your Deployment is either in the middle of a rollout and it is progressing or that it has successfully completed its progress and the minimum required new replicas are available (see the Reason of the condition for the particulars - in our case Reason=NewReplicaSetAvailable means that the Deployment is complete).

You can check if a Deployment has failed to progress by using kubectl rollout status. kubectl rollout status returns a non-zero exit code if the Deployment has exceeded the progression deadline.

$ kubectl rollout status deploy/nginx-deployment

Waiting **for** rollout to finish: 2 out of 3 new replicas have been updated...

error: deployment "nginx" exceeded its progress deadline

$ echo $?

1

Operating on a failed deployment

All actions that apply to a complete Deployment also apply to a failed Deployment. You can scale it up/down, roll back to a previous revision, or even pause it if you need to apply multiple tweaks in the Deployment pod template.

Clean up Policy

You can set .spec.revisionHistoryLimit field in a Deployment to specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, all revision history will be kept. In a future version, it will default to switch to 2.

**注意：** Explicitly setting this field to 0, will result in cleaning up all the history of your Deployment thus that Deployment will not be able to roll back.

Use Cases

Canary Deployment

If you want to roll out releases to a subset of users or servers using the Deployment, you can create multiple Deployments, one for each release, following the canary pattern described in [managing resources](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#canary-deployments).

Writing a Deployment Spec

As with all other Kubernetes configs, a Deployment needs apiVersion, kind, and metadata fields. For general information about working with config files, see [deploying applications](https://kubernetes.io/docs/tutorials/stateless-application/run-stateless-application-deployment/), configuring containers, and [using kubectl to manage resources](https://kubernetes.io/docs/tutorials/object-management-kubectl/object-management/) documents.

A Deployment also needs a [.spec section](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status).

Pod Template

The .spec.template is the only required field of the .spec.

The .spec.template is a [pod template](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#pod-templates). It has exactly the same schema as a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/), except it is nested and does not have an apiVersion or kind.

In addition to required fields for a Pod, a pod template in a Deployment must specify appropriate labels and an appropriate restart policy. For labels, make sure not to overlap with other controllers. See [selector](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#selector)).

Only a [.spec.template.spec.restartPolicy](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy) equal to Always is allowed, which is the default if not specified.

Replicas

.spec.replicas is an optional field that specifies the number of desired Pods. It defaults to 1.

Selector

.spec.selector is an optional field that specifies a [label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) for the Pods targeted by this deployment.

If specified, .spec.selector must match .spec.template.metadata.labels, or it will be rejected by the API. If .spec.selector is unspecified, .spec.selector.matchLabelsdefaults to .spec.template.metadata.labels.

A Deployment may terminate Pods whose labels match the selector if their template is different from .spec.template or if the total number of such Pods exceeds .spec.replicas. It brings up new Pods with .spec.template if the number of Pods is less than the desired number.

**注意：** You should not create other pods whose labels match this selector, either directly, by creating another Deployment, or by creating another controller such as a ReplicaSet or a ReplicationController. If you do so, the first Deployment thinks that it created these other pods. Kubernetes does not stop you from doing this.

If you have multiple controllers that have overlapping selectors, the controllers will fight with each other and won’t behave correctly.

Strategy

.spec.strategy specifies the strategy used to replace old Pods by new ones. .spec.strategy.type can be “Recreate” or “RollingUpdate”. “RollingUpdate” is the default value.

Recreate Deployment

All existing Pods are killed before new ones are created when .spec.strategy.type==Recreate.

Rolling Update Deployment

The Deployment updates Pods in a [rolling update](https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/) fashion when .spec.strategy.type==RollingUpdate. You can specify maxUnavailable and maxSurge to control the rolling update process.

Max Unavailable

.spec.strategy.rollingUpdate.maxUnavailable is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if .spec.strategy.rollingUpdate.maxSurge is 0. The default value is 25%.

For example, when this value is set to 30%, the old ReplicaSet can be scaled down to 70% of desired Pods immediately when the rolling update starts. Once new Pods are ready, old ReplicaSet can be scaled down further, followed by scaling up the new ReplicaSet, ensuring that the total number of Pods available at all times during the update is at least 70% of the desired Pods.

Max Surge

.spec.strategy.rollingUpdate.maxSurge is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if MaxUnavailable is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

For example, when this value is set to 30%, the new ReplicaSet can be scaled up immediately when the rolling update starts, such that the total number of old and new Pods does not exceed 130% of desired Pods. Once old Pods have been killed, the new ReplicaSet can be scaled up further, ensuring that the total number of Pods running at any time during the update is at most 130% of desired Pods.

Progress Deadline Seconds

.spec.progressDeadlineSeconds is an optional field that specifies the number of seconds you want to wait for your Deployment to progress before the system reports back that the Deployment has [failed progressing](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#failed-deployment) - surfaced as a condition with Type=Progressing, Status=False. and Reason=ProgressDeadlineExceeded in the status of the resource. The deployment controller will keep retrying the Deployment. In the future, once automatic rollback will be implemented, the deployment controller will roll back a Deployment as soon as it observes such a condition.

If specified, this field needs to be greater than .spec.minReadySeconds.

Min Ready Seconds

.spec.minReadySeconds is an optional field that specifies the minimum number of seconds for which a newly created Pod should be ready without any of its containers crashing, for it to be considered available. This defaults to 0 (the Pod will be considered available as soon as it is ready). To learn more about when a Pod is considered ready, see [Container Probes](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes).

Rollback To

.spec.rollbackTo is an optional field with the configuration the Deployment should roll back to. Setting this field triggers a rollback, and this field will be cleared by the server after a rollback is done.

Because this field will be cleared by the server, it should not be used declaratively. For example, you should not perform kubectl apply with a manifest with .spec.rollbackTo field set.

Revision

.spec.rollbackTo.revision is an optional field specifying the revision to roll back to. Setting to 0 means rolling back to the last revision in history; otherwise, means rolling back to the specified revision. This defaults to 0 when [spec.rollbackTo](https://kubernetes.io/zh/docs/concepts/workloads/controllers/deployment/#rollback-to) is set.

Revision History Limit

A Deployment’s revision history is stored in the replica sets it controls.

.spec.revisionHistoryLimit is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. Its ideal value depends on the frequency and stability of new Deployments. All old ReplicaSets will be kept by default, consuming resources in etcd and crowding the output of kubectl get rs, if this field is not set. The configuration of each Deployment revision is stored in its ReplicaSets; therefore, once an old ReplicaSet is deleted, you lose the ability to rollback to that revision of Deployment.

More specifically, setting this field to zero means that all old ReplicaSets with 0 replica will be cleaned up. In this case, a new Deployment rollout cannot be undone, since its revision history is cleaned up.

Paused

.spec.paused is an optional boolean field for pausing and resuming a Deployment. The only difference between a paused Deployment and one that is not paused, is that any changes into the PodTemplateSpec of the paused Deployment will not trigger new rollouts as long as it is paused. A Deployment is not paused by default when it is created.

Alternative to Deployments

kubectl rolling update

[Kubectl rolling update](https://kubernetes.io/docs/user-guide/kubectl/v1.17/#rolling-update) updates Pods and ReplicationControllers in a similar fashion. But Deployments are recommended, since they are declarative, server side, and have additional features, such as rolling back to any previous revision even after the rolling update is done.

反馈

垃圾收集

Kubernetes 垃圾收集器的作用是删除某些曾经拥有所有者（owner）但现在不再拥有所有者的对象。

* [**所有者和附属**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/garbage-collection/#%e6%89%80%e6%9c%89%e8%80%85%e5%92%8c%e9%99%84%e5%b1%9e)
* [**控制垃圾收集器删除附属者**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/garbage-collection/#%e6%8e%a7%e5%88%b6%e5%9e%83%e5%9c%be%e6%94%b6%e9%9b%86%e5%99%a8%e5%88%a0%e9%99%a4%e9%99%84%e5%b1%9e%e8%80%85)
* [**已知的问题**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/garbage-collection/#%e5%b7%b2%e7%9f%a5%e7%9a%84%e9%97%ae%e9%a2%98)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/garbage-collection/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)
* [**控制垃圾收集器删除附属者**](https://kubernetes.io/zh/docs/concepts/workloads/controllers/garbage-collection/#%e6%8e%a7%e5%88%b6%e5%9e%83%e5%9c%be%e6%94%b6%e9%9b%86%e5%99%a8%e5%88%a0%e9%99%a4%e9%99%84%e5%b1%9e%e8%80%85)

所有者和附属

某些 Kubernetes 对象是其它一些对象的所有者。例如，一个 ReplicaSet 是一组 Pod 的所有者。 具有所有者的对象被称为是所有者的\*附属\*。 每个附属对象具有一个指向其所属对象的 metadata.ownerReferences 字段。

有时，Kubernetes 会自动设置 ownerReference 的值。 例如，当创建一个 ReplicaSet 时，Kubernetes 自动设置 ReplicaSet 中每个 Pod 的 ownerReference 字段值。 在 1.6 版本，Kubernetes 会自动为某些对象设置 ownerReference 的值，这些对象是由 ReplicationController、ReplicaSet、StatefulSet、DaemonSet 和 Deployment 所创建或管理。 也可以通过手动设置 ownerReference 的值，来指定所有者和附属之间的关系。

这里有一个配置文件，表示一个具有 3 个 Pod 的 ReplicaSet：

| [controllers/replicaset.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/controllers/replicaset.yaml) |
| --- |
| apiVersion: apps/v1  kind: ReplicaSet  metadata:  name: my-repset  spec:  replicas: 3  selector:  matchLabels:  pod-is-for: garbage-collection-example  template:  metadata:  labels:  pod-is-for: garbage-collection-example  spec:  containers:  - name: nginx  image: nginx |

如果创建该 ReplicaSet，然后查看 Pod 的 metadata 字段，能够看到 OwnerReferences 字段：

kubectl apply -f https://k8s.io/examples/controllers/replicaset.yaml

kubectl get pods --output=yaml

输出显示了 Pod 的所有者是名为 my-repset 的 ReplicaSet：

apiVersion: v1

kind: Pod

metadata:

...

ownerReferences:

- apiVersion: apps/v1

controller: true

blockOwnerDeletion: true

kind: ReplicaSet

name: my-repset

uid: d9607e19-f88f-11e6-a518-42010a800195

...

**注意：**

根据设计，kubernetes 不允许跨命名空间指定所有者。这意味着：

1）命名空间范围的附属只能在相同的命名空间中指定所有者，并且只能指定集群范围的所有者。

2）集群范围的附属只能指定集群范围的所有者，不能指定命名空间范围的。

控制垃圾收集器删除附属者

当删除对象时，可以指定是否该对象的附属者也自动删除掉。 自动删除 Dependent 也称为 *级联删除* 。 Kubernetes 中有两种 *级联删除* 的模式：*background* 模式和 *foreground* 模式。

如果删除对象时，不自动删除它的附属者，这些附属者被称作是原对象的*孤儿* 。

显式级联删除

在\*显式级联删除\*模式下，根对象首先进入 deletion in progress 状态。在 deletion in progress 状态会有如下的情况：

* 对象仍然可以通过 REST API 可见。
* 会设置对象的 deletionTimestamp 字段。
* 对象的 metadata.finalizers 字段包含了值 foregroundDeletion。

一旦对象被设置为 deletion in progress 状态，垃圾收集器会删除对象的所有附属。 垃圾收集器在删除了所有 Blocking 状态的附属（对象的 ownerReference.blockOwnerDeletion=true）之后，它会删除拥有者对象。

注意，在 foregroundDeletion 模式下，只有设置了 ownerReference.blockOwnerDeletion 值的附属者才能阻止删除拥有者对象。 在 Kubernetes 1.7 版本中将增加准入控制器（Admission Controller），基于拥有者对象上的删除权限来控制用户去设置 blockOwnerDeletion 的值为 true，所以未授权的附属者不能够延迟拥有者对象的删除。

如果一个对象的 ownerReferences 字段被一个 Controller（例如 Deployment 或 ReplicaSet）设置，blockOwnerDeletion 会被自动设置，不需要手动修改这个字段。

隐式级联删除

在 *隐式级联删除除* 模式下，Kubernetes 会立即删除拥有者对象，然后垃圾收集器会在后台删除这些附属。

设置级联删除策略

通过为拥有者对象设置 deleteOptions.propagationPolicy 字段，可以控制级联删除策略。 可能的取值包括：orphan、Foreground或Background。

对很多 Controller 资源，包括 ReplicationController、ReplicaSet、StatefulSet、DaemonSet 和 Deployment，默认的垃圾收集策略是 orphan。 因此，除非指定其它的垃圾收集策略，否则所有附属对象使用的都是 orphan 策略。

下面是一个在后台删除 Dependent 对象的例子：

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Background"}' **\**

-H "Content-Type: application/json"

下面是一个在前台删除附属对象的例子：

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}' **\**

-H "Content-Type: application/json"

这里是一个孤儿附属的例子：

kubectl proxy --port=8080

curl -X DELETE localhost:8080/apis/apps/v1/namespaces/default/replicasets/my-repset **\**

-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Orphan"}' **\**

-H "Content-Type: application/json"

kubectl 也支持级联删除。 通过设置 --cascade 为 true，可以使用 kubectl 自动删除附属对象。设置 --cascade 为 false，会使附属对象成为孤儿附属对象。--cascade 的默认值是 true。

下面是一个例子，使一个 ReplicaSet 的附属对象成为孤儿附属：

kubectl delete replicaset my-repset --cascade=false

Deployment 的其他说明

在 1.7 之前的版本中，当在 Deployment 中使用级联删除时，您必须*使用* propagationPolicy:Foreground。这样不仅删除所创建的 ReplicaSet，还删除其 Pod。如果不使用这种类型的 propagationPolicy，则将只删除 ReplicaSet，而 Pod 被孤立。

更多信息，请参考 [kubeadm/#149](https://github.com/kubernetes/kubeadm/issues/149#issuecomment-284766613)。

已知的问题

跟踪 [#26120](https://github.com/kubernetes/kubernetes/issues/26120)

接下来

控制垃圾收集器删除附属者

[设计文档 1](https://git.k8s.io/community/contributors/design-proposals/api-machinery/garbage-collection.md)

[设计文档 2](https://git.k8s.io/community/contributors/design-proposals/api-machinery/synchronous-garbage-collection.md)

存储

## Volumes

容器中的文件在磁盘上是临时存放的，这给容器中运行的特殊应用程序带来一些问题。 首先，当容器崩溃时，kubelet 将重新启动容器，容器中的文件将会丢失——因为容器会以干净的状态重建。 其次，当在一个 Pod 中同时运行多个容器时，常常需要在这些容器之间共享文件。 Kubernetes 抽象出 Volume 对象来解决这两个问题。

阅读本文前建议您熟悉一下 [Pods](https://kubernetes.io/docs/user-guide/pods)。

* [**背景**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#%e8%83%8c%e6%99%af)
* [**Volume 的类型**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#volume-%e7%9a%84%e7%b1%bb%e5%9e%8b)
* [**使用 subPath**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#%e4%bd%bf%e7%94%a8-subpath)
* [**资源**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#%e8%b5%84%e6%ba%90)
* [**Out-of-Tree 卷插件**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#out-of-tree-%e5%8d%b7%e6%8f%92%e4%bb%b6)
* [**挂载卷的传播**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#%e6%8c%82%e8%bd%bd%e5%8d%b7%e7%9a%84%e4%bc%a0%e6%92%ad)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/storage/volumes/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 背景

Docker 也有 [Volume](https://docs.docker.com/engine/admin/volumes/) 的概念，但对它只有少量且松散的管理。 在 Docker 中，Volume 是磁盘上或者另外一个容器内的一个目录。 直到最近，Docker 才支持对基于本地磁盘的 Volume 的生存期进行管理。 虽然 Docker 现在也能提供 Volume 驱动程序，但是目前功能还非常有限（例如，截至 Docker 1.7，每个容器只允许有一个 Volume 驱动程序，并且无法将参数传递给卷）。

另一方面，Kubernetes 卷具有明确的生命周期——与包裹它的 Pod 相同。 因此，卷比 Pod 中运行的任何容器的存活期都长，在容器重新启动时数据也会得到保留。 当然，当一个 Pod 不再存在时，卷也将不再存在。也许更重要的是，Kubernetes 可以支持许多类型的卷，Pod 也能同时使用任意数量的卷。

卷的核心是包含一些数据的目录，Pod 中的容器可以访问该目录。 特定的卷类型可以决定这个目录如何形成的，并能决定它支持何种介质，以及目录中存放什么内容。

使用卷时, Pod 声明中需要提供卷的类型 (.spec.volumes 字段)和卷挂载的位置 (.spec.containers.volumeMounts 字段).

容器中的进程能看到由它们的 Docker 镜像和卷组成的文件系统视图。 [Docker 镜像](https://docs.docker.com/userguide/dockerimages/) 位于文件系统层次结构的根部，并且任何 Volume 都挂载在镜像内的指定路径上。 卷不能挂载到其他卷，也不能与其他卷有硬链接。 Pod 中的每个容器必须独立地指定每个卷的挂载位置。

### Volume 的类型

Kubernetes 支持下列类型的卷：

* [awsElasticBlockStore](https://kubernetes.io/zh/docs/concepts/storage/volumes/#awselasticblockstore)
* [azureDisk](https://kubernetes.io/zh/docs/concepts/storage/volumes/#azuredisk)
* [azureFile](https://kubernetes.io/zh/docs/concepts/storage/volumes/#azurefile)
* [cephfs](https://kubernetes.io/zh/docs/concepts/storage/volumes/#cephfs)
* [cinder](https://kubernetes.io/zh/docs/concepts/storage/volumes/#cinder)
* [configMap](https://kubernetes.io/zh/docs/concepts/storage/volumes/#configmap)
* [csi](https://kubernetes.io/zh/docs/concepts/storage/volumes/#csi)
* [downwardAPI](https://kubernetes.io/zh/docs/concepts/storage/volumes/#downwardapi)
* [emptyDir](https://kubernetes.io/zh/docs/concepts/storage/volumes/#emptydir)
* [fc (fibre channel)](https://kubernetes.io/zh/docs/concepts/storage/volumes/#fc)
* [flexVolume](https://kubernetes.io/zh/docs/concepts/storage/volumes/#flexVolume)
* [flocker](https://kubernetes.io/zh/docs/concepts/storage/volumes/#flocker)
* [gcePersistentDisk](https://kubernetes.io/zh/docs/concepts/storage/volumes/#gcepersistentdisk)
* [gitRepo (deprecated)](https://kubernetes.io/zh/docs/concepts/storage/volumes/#gitrepo)
* [glusterfs](https://kubernetes.io/zh/docs/concepts/storage/volumes/#glusterfs)
* [hostPath](https://kubernetes.io/zh/docs/concepts/storage/volumes/#hostpath)
* [iscsi](https://kubernetes.io/zh/docs/concepts/storage/volumes/#iscsi)
* [local](https://kubernetes.io/zh/docs/concepts/storage/volumes/#local)
* [nfs](https://kubernetes.io/zh/docs/concepts/storage/volumes/#nfs)
* [persistentVolumeClaim](https://kubernetes.io/zh/docs/concepts/storage/volumes/#persistentvolumeclaim)
* [projected](https://kubernetes.io/zh/docs/concepts/storage/volumes/#projected)
* [portworxVolume](https://kubernetes.io/zh/docs/concepts/storage/volumes/#portworxvolume)
* [quobyte](https://kubernetes.io/zh/docs/concepts/storage/volumes/#quobyte)
* [rbd](https://kubernetes.io/zh/docs/concepts/storage/volumes/#rbd)
* [scaleIO](https://kubernetes.io/zh/docs/concepts/storage/volumes/#scaleio)
* [secret](https://kubernetes.io/zh/docs/concepts/storage/volumes/#secret)
* [storageos](https://kubernetes.io/zh/docs/concepts/storage/volumes/#storageos)
* [vsphereVolume](https://kubernetes.io/zh/docs/concepts/storage/volumes/#vspherevolume)

我们欢迎大家贡献其他的卷类型支持。

#### awsElasticBlockStore

awsElasticBlockStore 卷将 Amazon Web服务（AWS）[EBS 卷](http://aws.amazon.com/ebs/) 挂载到您的 Pod 中。 与 emptyDir 在删除 Pod 时会被删除不同，EBS 卷的内容在删除 Pod 时会被保留，卷只是被卸载掉了。 这意味着 EBS 卷可以预先填充数据，并且可以在 Pod 之间传递数据。

**警告:**

您在使用 EBS 卷之前必须先创建它，可以使用 aws ec2 create-volume 命令进行创建；也可以使用 AWS API 进行创建。

使用 awsElasticBlockStore 卷时有一些限制：

* Pod 正在运行的节点必须是 AWS EC2 实例。
* 这些实例需要与 EBS 卷在相同的地域（region）和可用区（availability-zone）。
* EBS 卷只支持被挂载到单个 EC2 实例上。

#### 创建 EBS 卷

在将 EBS 卷用到 Pod 上之前，您首先要创建它。

aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2

确保该区域与您的群集所在的区域相匹配。（也要检查卷的大小和 EBS 卷类型都适合您的用途！）

#### AWS EBS 配置示例

apiVersion: v1

kind: Pod

metadata:

name: test-ebs

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-ebs

name: test-volume

volumes:

- name: test-volume

*# This AWS EBS volume must already exist.*

awsElasticBlockStore:

volumeID: <volume-id*>*

*fsType: ext4*

#### azureDisk

azureDisk 用来在 Pod 上挂载 Microsoft Azure [数据盘（Data Disk）](https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-about-disks-vhds/) . 更多详情请参考[这里](https://github.com/kubernetes/examples/tree/master/staging/volumes/azure_disk/README.md)。

#### CSI迁移

**FEATURE STATE:** Kubernetes v1.15 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

启用azureDisk的CSI迁移功能后，它会将所有插件操作从现有的内建插件填添加disk.csi.azure.com容器存储接口（CSI）驱动程序中。 为了使用此功能，必须在群集上安装 [Azure磁盘CSI驱动程序](https://github.com/kubernetes-sigs/azuredisk-csi-driver)， 并且 CSIMigration 和 CSIMigrationAzureDisk Alpha功能 必须启用。

#### azureFile

azureFile 用来在 Pod 上挂载 Microsoft Azure 文件卷（File Volume） (SMB 2.1 和 3.0)。 更多详情请参考[这里](https://github.com/kubernetes/examples/tree/master/staging/volumes/azure_file/README.md)。

#### CSI迁移

**FEATURE STATE:** Kubernetes v1.15 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

启用azureFile的CSI迁移功能后，它会将所有插件操作从现有的内建插件填添加file.csi.azure.com容器存储接口（CSI）驱动程序中。 为了使用此功能，必须在群集上安装 [Azure文件CSI驱动程序](https://github.com/kubernetes-sigs/azurefile-csi-driver)， 并且 CSIMigration 和 CSIMigrationAzureFile Alpha功能 必须启用。

#### cephfs

cephfs 允许您将现存的 CephFS 卷挂载到 Pod 中。不像 emptyDir 那样会在删除 Pod 的同时也会被删除，cephfs 卷的内容在删除 Pod 时会被保留，卷只是被卸载掉了。 这意味着 CephFS 卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。CephFS 卷可同时被多个写者挂载。

**警告:**

在您使用 Ceph 卷之前，您的 Ceph 服务器必须正常运行并且要使用的 share 被导出（exported）。

更多信息请参考 [CephFS 示例](https://github.com/kubernetes/examples/tree/master/volumes/cephfs/)。

#### cinder

**注意：**

先决条件：配置了OpenStack Cloud Provider 的 Kubernetes。 有关 cloudprovider 配置，请参考 [cloud provider openstack](https://kubernetes.io/docs/concepts/cluster-administration/cloud-providers/#openstack)。

cinder 用于将 OpenStack Cinder 卷安装到 Pod 中。

#### Cinder Volume示例配置

apiVersion: v1

kind: Pod

metadata:

name: test-cinder

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-cinder-container

volumeMounts:

- mountPath: /test-cinder

name: test-volume

volumes:

- name: test-volume

*# This OpenStack volume must already exist.*

cinder:

volumeID: <volume-id*>*

*fsType: ext4*

#### CSI迁移

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

启用Cinder的CSI迁移功能后，它会将所有插件操作从现有的内建插件填添加 cinder.csi.openstack.org 容器存储接口（CSI）驱动程序中。 为了使用此功能，必须在群集上安装 [Openstack Cinder CSI驱动程序](https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/using-cinder-csi-plugin.md)， 并且 CSIMigration 和 CSIMigrationOpenStack Alpha功能 必须启用。

#### configMap

[configMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) 资源提供了向 Pod 注入配置数据的方法。 ConfigMap 对象中存储的数据可以被 configMap 类型的卷引用，然后被应用到 Pod 中运行的容器化应用。

当引用 configMap 对象时，你可以简单的在 Volume 中通过它名称来引用。 还可以自定义 ConfigMap 中特定条目所要使用的路径。 例如，要将名为 log-config 的 ConfigMap 挂载到名为 configmap-pod 的 Pod 中，您可以使用下面的 YAML：

apiVersion: v1

kind: Pod

metadata:

name: configmap-pod

spec:

containers:

- name: test

image: busybox

volumeMounts:

- name: config-vol

mountPath: /etc/config

volumes:

- name: config-vol

configMap:

name: log-config

items:

- key: log\_level

path: log\_level

log-config ConfigMap 是以卷的形式挂载的， 存储在 log\_level 条目中的所有内容都被挂载到 Pod 的 “/etc/config/log\_level” 路径下。 请注意，这个路径来源于 Volume 的 mountPath 和 log\_level 键对应的 path。

**警告:**

在使用 [ConfigMap](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/) 之前您首先要创建它。

**注意：**

容器以 [subPath](https://kubernetes.io/zh/docs/concepts/storage/volumes/#using-subpath) 卷挂载方式使用 ConfigMap 时，将无法接收 ConfigMap 的更新。

#### downwardAPI

downwardAPI 卷用于使 downward API 数据对应用程序可用。 这种卷类型挂载一个目录并在纯文本文件中写入请求的数据。

**注意：**

容器以挂载 [subPath](https://kubernetes.io/zh/docs/concepts/storage/volumes/#using-subpath) 卷的方式使用 downwardAPI 时，将不能接收到它的更新。

更多详细信息请参考 [downwardAPI 卷示例](https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/)。

#### emptyDir

当 Pod 指定到某个节点上时，首先创建的是一个 emptyDir 卷，并且只要 Pod 在该节点上运行，卷就一直存在。 就像它的名称表示的那样，卷最初是空的。 尽管 Pod 中的容器挂载 emptyDir 卷的路径可能相同也可能不同，但是这些容器都可以读写 emptyDir 卷中相同的文件。 当 Pod 因为某些原因被从节点上删除时，emptyDir 卷中的数据也会永久删除。

**注意：**

容器崩溃并不会导致 Pod 被从节点上移除，因此容器崩溃时 emptyDir 卷中的数据是安全的。

emptyDir 的一些用途：

* 缓存空间，例如基于磁盘的归并排序。
* 为耗时较长的计算任务提供检查点，以便任务能方便地从崩溃前状态恢复执行。
* 在 Web 服务器容器服务数据时，保存内容管理器容器获取的文件。

默认情况下， emptyDir 卷存储在支持该节点所使用的介质上；这里的介质可以是磁盘或 SSD 或网络存储，这取决于您的环境。 但是，您可以将 emptyDir.medium 字段设置为 "Memory"，以告诉 Kubernetes 为您安装 tmpfs（基于 RAM 的文件系统）。 虽然 tmpfs 速度非常快，但是要注意它与磁盘不同。 tmpfs 在节点重启时会被清除，并且您所写入的所有文件都会计入容器的内存消耗，受容器内存限制约束。

#### Pod 示例

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /cache

name: cache-volume

volumes:

- name: cache-volume

emptyDir: {}

#### fc (光纤通道)

fc 卷允许将现有的光纤通道卷挂载到 Pod 中。 可以使用卷配置中的参数 targetWWNs 来指定单个或多个目标 WWN。 如果指定多个 WWN，targetWWNs 期望这些 WWN 来自多路径连接。

**警告:**

您必须配置 FC SAN Zoning，以便预先向目标 WWN 分配和屏蔽这些 LUN（卷），这样 Kubernetes 主机才可以访问它们。

更多详情请参考 [FC 示例](https://github.com/kubernetes/examples/tree/master/staging/volumes/fibre_channel)。

#### flocker

[Flocker](https://github.com/ClusterHQ/flocker) 是一个开源的、集群化的容器数据卷管理器。 Flocker 提供了由各种存储后备支持的数据卷的管理和编排。

flocker 卷允许将一个 Flocker 数据集挂载到 Pod 中。 如果数据集在 Flocker 中不存在，则需要首先使用 Flocker CLI 或 Flocker API 创建数据集。 如果数据集已经存在，那么 Flocker 将把它重新附加到 Pod 被调度的节点。 这意味着数据可以根据需要在 Pod 之间 “传递”。

**警告:**

您在使用 Flocker 之前必须先安装运行自己的 Flocker。

更多详情请参考 [Flocker 示例](https://github.com/kubernetes/examples/tree/master/staging/volumes/flocker)。

#### gcePersistentDisk

gcePersistentDisk 卷能将谷歌计算引擎 (GCE) [持久盘（PD）](http://cloud.google.com/compute/docs/disks) 挂载到您的 Pod 中。 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，持久盘卷的内容在删除 Pod 时会被保留，卷只是被卸载掉了。 这意味着持久盘卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。

**警告:**

您在使用 PD 前，必须使用 gcloud 或者 GCE API 或 UI 创建它。

使用 gcePersistentDisk 时有一些限制：

* 运行 Pod 的节点必须是 GCE VM
* 那些 VM 必须和持久盘属于相同的 GCE 项目和区域（zone）

PD 的一个特点是它们可以同时被多个消费者以只读方式挂载。 这意味着您可以用数据集预先填充 PD，然后根据需要并行地在尽可能多的 Pod 中提供该数据集。 不幸的是，PD 只能由单个使用者以读写模式挂载——即不允许同时写入。

在由 ReplicationController 所管理的 Pod 上使用 PD 将会失败，除非 PD 是只读模式或者副本的数量是 0 或 1。

#### 创建持久盘（PD）

在 Pod 中使用 GCE 持久盘之前，您首先要创建它。

gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk

#### Pod 示例

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-pd

name: test-volume

volumes:

- name: test-volume

*# This GCE PD must already exist.*

gcePersistentDisk:

pdName: my-data-disk

fsType: ext4

#### 区域持久盘（Regional Persistent Disks）

**FEATURE STATE:** Kubernetes v1.10 [beta](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

[区域持久盘](https://cloud.google.com/compute/docs/disks/#repds) 功能允许您创建能在同一区域的两个可用区中使用的持久盘。 要使用这个功能，必须以持久盘的方式提供卷；Pod 不支持直接引用这种卷。

#### 手动供应基于区域 PD 的 PersistentVolume

使用 [为 GCE PD 定义的存储类](https://kubernetes.io/docs/concepts/storage/storage-classes/#gce) 也可以动态供应。 在创建 PersistentVolume 之前，您首先要创建 PD。

gcloud beta compute disks create --size=500GB my-data-disk

--region us-central1

--replica-zones us-central1-a,us-central1-b

PersistentVolume 示例：

apiVersion: v1

kind: PersistentVolume

metadata:

name: test-volume

labels:

failure-domain.beta.kubernetes.io/zone: us-central1-a\_\_us-central1-b

spec:

capacity:

storage: 400Gi

accessModes:

- ReadWriteOnce

gcePersistentDisk:

pdName: my-data-disk

fsType: ext4

#### CSI迁移

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

启用 GCE PD 的 CSI 迁移功能后，它会将所有插件操作从现有的内建插件填添加 pd.csi.storage.gke.io 容器存储接口（ CSI ）驱动程序中。 为了使用此功能，必须在群集上安装 [GCE PD CSI驱动程序](https://github.com/kubernetes-sigs/gcp-compute-persistent-disk-csi-driver)， 并且 CSIMigration 和 CSIMigrationGCE Alpha功能 必须启用。

#### gitRepo (已弃用)

**警告：**

gitRepo 卷类型已经被废弃。如果需要在容器中提供 git 仓库，请将一个 [EmptyDir](https://kubernetes.io/zh/docs/concepts/storage/volumes/#emptydir) 卷挂载到 InitContainer 中，使用 git 命令完成仓库的克隆操作，然后将 [EmptyDir](https://kubernetes.io/zh/docs/concepts/storage/volumes/#emptydir) 卷挂载到 Pod 的容器中。

gitRepo 卷是一个卷插件的例子。 该卷类型挂载了一个空目录，并将一个 Git 代码仓库克隆到这个目录中供您使用。 将来，这种卷可能被移动到一个更加解耦的模型中，而不是针对每个应用案例扩展 Kubernetes API。

下面给出一个 gitRepo 卷的示例：

apiVersion: v1

kind: Pod

metadata:

name: server

spec:

containers:

- image: nginx

name: nginx

volumeMounts:

- mountPath: /mypath

name: git-volume

volumes:

- name: git-volume

gitRepo:

repository: "git@somewhere:me/my-git-repository.git"

revision: "22f1d8406d464b0c0874075539c1f2e96c253775"

#### glusterfs

glusterfs 卷能将 [Glusterfs](http://www.gluster.org/) (一个开源的网络文件系统) 挂载到您的 Pod 中。 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，glusterfs 卷的内容在删除 Pod 时会被保存，卷只是被卸载掉了。 这意味着 glusterfs 卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。GlusterFS 可以被多个写者同时挂载。

**警告:**

在使用前您必须先安装运行自己的 GlusterFS。

更多详情请参考 [GlusterFS 示例](https://github.com/kubernetes/examples/tree/master/volumes/glusterfs)。

#### hostPath

hostPath 卷能将主机节点文件系统上的文件或目录挂载到您的 Pod 中。 虽然这不是大多数 Pod 需要的，但是它为一些应用程序提供了强大的逃生舱。

例如，hostPath 的一些用法有：

* 运行一个需要访问 Docker 引擎内部机制的容器；请使用 hostPath 挂载 /var/lib/docker 路径。
* 在容器中运行 cAdvisor 时，以 hostPath 方式挂载 /sys。
* 允许 Pod 指定给定的 hostPath 在运行 Pod 之前是否应该存在，是否应该创建以及应该以什么方式存在。

除了必需的 path 属性之外，用户可以选择性地为 hostPath 卷指定 type。

支持的 type 值如下：

| 取值 | 行为 |
| --- | --- |
|  | 空字符串（默认）用于向后兼容，这意味着在安装 hostPath 卷之前不会执行任何检查。 |
| DirectoryOrCreate | 如果在给定路径上什么都不存在，那么将根据需要创建空目录，权限设置为 0755，具有与 Kubelet 相同的组和所有权。 |
| Directory | 在给定路径上必须存在的目录。 |
| FileOrCreate | 如果在给定路径上什么都不存在，那么将在那里根据需要创建空文件，权限设置为 0644，具有与 Kubelet 相同的组和所有权。 |
| File | 在给定路径上必须存在的文件。 |
| Socket | 在给定路径上必须存在的 UNIX 套接字。 |
| CharDevice | 在给定路径上必须存在的字符设备。 |
| BlockDevice | 在给定路径上必须存在的块设备。 |

当使用这种类型的卷时要小心，因为：

* 具有相同配置（例如从 podTemplate 创建）的多个 Pod 会由于节点上文件的不同而在不同节点上有不同的行为。
* 当 Kubernetes 按照计划添加资源感知的调度时，这类调度机制将无法考虑由 hostPath 使用的资源。
* 基础主机上创建的文件或目录只能由 root 用户写入。您需要在 [特权容器](https://kubernetes.io/docs/user-guide/security-context) 中以 root 身份运行进程，或者修改主机上的文件权限以便容器能够写入 hostPath 卷。

#### Pod 示例

apiVersion: v1

kind: Pod

metadata:

name: test-pd

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-pd

name: test-volume

volumes:

- name: test-volume

hostPath:

*# directory location on host*

path: /data

*# this field is optional*

type: Directory

#### iscsi

iscsi 卷能将 iSCSI (基于 IP 的 SCSI) 挂载到您的 Pod 中。 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，持久盘 卷的内容在删除 Pod 时会被保存，卷只是被卸载掉了。 这意味着 iscsi 卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。

**警告:**

在您使用 iSCSI 卷之前，您必须拥有自己的 iSCSI 服务器，并在上面创建卷。

iSCSI 的一个特点是它可以同时被多个用户以只读方式挂载。 这意味着您可以用数据集预先填充卷，然后根据需要在尽可能多的 Pod 上提供它。不幸的是，iSCSI 卷只能由单个使用者以读写模式挂载——不允许同时写入。

更多详情请参考 [iSCSI 示例](https://github.com/kubernetes/examples/tree/master/volumes/iscsi)。

#### local

**FEATURE STATE:** Kubernetes v1.14 [稳定](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

**注意：**

alpha 版本的 PersistentVolume NodeAffinity 注释已被取消，将在将来的版本中废弃。 用户必须更新现有的使用该注解的 PersistentVolume，以使用新的 PersistentVolume NodeAffinity 字段。

local 卷指的是所挂载的某个本地存储设备，例如磁盘、分区或者目录。

local 卷只能用作静态创建的持久卷。尚不支持动态配置。

相比 hostPath 卷，local 卷可以以持久和可移植的方式使用，而无需手动将 Pod 调度到节点，因为系统通过查看 PersistentVolume 所属节点的亲和性配置，就能了解卷的节点约束。

然而，local 卷仍然取决于底层节点的可用性，并不是适合所有应用程序。 如果节点变得不健康，那么local 卷也将变得不可访问，并且使用它的 Pod 将不能运行。 使用 local 卷的应用程序必须能够容忍这种可用性的降低，以及因底层磁盘的耐用性特征而带来的潜在的数据丢失风险。

下面是一个使用 local 卷和 nodeAffinity 的持久卷示例：

apiVersion: v1

kind: PersistentVolume

metadata:

name: example-pv

spec:

capacity:

storage: 100Gi

*# volumeMode field requires BlockVolume Alpha feature gate to be enabled.*

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Delete

storageClassName: local-storage

local:

path: /mnt/disks/ssd1

nodeAffinity:

required:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- example-node

使用 local 卷时，需要使用 PersistentVolume 对象的 nodeAffinity 字段。 它使 Kubernetes 调度器能够将使用 local 卷的 Pod 正确地调度到合适的节点。

现在，可以将 PersistentVolume 对象的 volumeMode 字段设置为 “Block”（而不是默认值 “Filesystem”），以将 local 卷作为原始块设备暴露出来。 volumeMode 字段需要启用 Alpha 功能 BlockVolume。

当使用 local 卷时，建议创建一个 StorageClass，将 volumeBindingMode 设置为 WaitForFirstConsumer。 请参考 [示例](https://kubernetes.io/docs/concepts/storage/storage-classes/#local)。 延迟卷绑定操作可以确保 Kubernetes 在为 PersistentVolumeClaim 作出绑定决策时，会评估 Pod 可能具有的其他节点约束，例如：如节点资源需求、节点选择器、Pod 亲和性和 Pod 反亲和性。

您可以在 Kubernetes 之外单独运行静态驱动以改进对 local 卷的生命周期管理。 请注意，此驱动不支持动态配置。 有关如何运行外部 local 卷驱动的示例，请参考 [local 卷驱动用户指南](https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner)。

**注意：**

如果不使用外部静态驱动来管理卷的生命周期，则用户需要手动清理和删除 local 类型的持久卷。

#### nfs

nfs 卷能将 NFS (网络文件系统) 挂载到您的 Pod 中。 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，nfs 卷的内容在删除 Pod 时会被保存，卷只是被卸载掉了。 这意味着 nfs 卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。

**警告:**

在您使用 NFS 卷之前，必须运行自己的 NFS 服务器并将目标 share 导出备用。

要了解更多详情请参考 [NFS 示例](https://github.com/kubernetes/examples/tree/master/staging/volumes/nfs)。

#### persistentVolumeClaim

persistentVolumeClaim 卷用来将[持久卷](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)（PersistentVolume）挂载到 Pod 中。 持久卷是用户在不知道特定云环境细节的情况下”申领”持久存储（例如 GCE PersistentDisk 或者 iSCSI 卷）的一种方法。

更多详情请参考[持久卷示例](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

#### projected

projected 卷类型能将若干现有的卷来源映射到同一目录上。

目前，可以映射的卷来源类型如下：

* [secret](https://kubernetes.io/zh/docs/concepts/storage/volumes/#secret)
* [downwardAPI](https://kubernetes.io/zh/docs/concepts/storage/volumes/#downwardapi)
* [configMap](https://kubernetes.io/zh/docs/concepts/storage/volumes/#configmap)
* serviceAccountToken

所有的卷来源需要和 Pod 处于相同的命名空间。 更多详情请参考[一体化卷设计文档](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/all-in-one-volume.md)。

服务帐户令牌的映射是 Kubernetes 1.11 版本中引入的一个功能，并在 1.12 版本中被提升为 Beta 功能。 若要在 1.11 版本中启用此特性，需要显式设置 TokenRequestProjection [功能开关](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) 为 True。

#### 包含 secret、downwardAPI 和 configmap 的 Pod 示例如下：

apiVersion: v1

kind: Pod

metadata:

name: volume-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: all-in-one

mountPath: "/projected-volume"

readOnly: **true**

volumes:

- name: all-in-one

projected:

sources:

- secret:

name: mysecret

items:

- key: username

path: my-group/my-username

- downwardAPI:

items:

- path: "labels"

fieldRef:

fieldPath: metadata.labels

- path: "cpu\_limit"

resourceFieldRef:

containerName: container-test

resource: limits.cpu

- configMap:

name: myconfigmap

items:

- key: config

path: my-group/my-config

带有非默认许可模式设置的多个 secret 的 Pod 示例如下：

apiVersion: v1

kind: Pod

metadata:

name: volume-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: all-in-one

mountPath: "/projected-volume"

readOnly: **true**

volumes:

- name: all-in-one

projected:

sources:

- secret:

name: mysecret

items:

- key: username

path: my-group/my-username

- secret:

name: mysecret2

items:

- key: password

path: my-group/my-password

mode: 511

每个被投射的卷来源都在 spec 中的 sources 内列出。 参数几乎相同，除了两处例外：

* 对于 secret，secretName 字段已被变更为 name 以便与 ConfigMap 命名一致。
* defaultMode 只能根据投射级别指定，而不是针对每个卷来源指定。不过，如上所述，您可以显式地为每个投射项设置 mode 值。

当开启 TokenRequestProjection 功能时，可以将当前 [服务帐户](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#service-account-tokens)的令牌注入 Pod 中的指定路径。 下面是一个例子：

apiVersion: v1

kind: Pod

metadata:

name: sa-token-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: token-vol

mountPath: "/service-account"

readOnly: **true**

volumes:

- name: token-vol

projected:

sources:

- serviceAccountToken:

audience: api

expirationSeconds: 3600

path: token

示例 Pod 具有包含注入服务帐户令牌的映射卷。 例如，这个令牌可以被 Pod 容器用来访问 Kubernetes API服务器。 audience 字段包含令牌的预期受众。 令牌的接收者必须使用令牌的受众中指定的标识符来标识自己，否则应拒绝令牌。 此字段是可选的，默认值是 API 服务器的标识符。

expirationSeconds 是服务帐户令牌的有效期。 默认值为 1 小时，必须至少 10 分钟（600 秒）。 管理员还可以通过指定 API 服务器的 --service-account-max-token-expiration选项来限制其最大值。 path 字段指定相对于映射卷的挂载点的相对路径。

**注意：**

使用投射卷源作为 [subPath](https://kubernetes.io/zh/docs/concepts/storage/volumes/#using-subpath) 卷挂载的容器将不会接收这些卷源的更新。

#### portworxVolume

portworxVolume 是一个可伸缩的块存储层，能够以超聚合（hyperconverged）的方式与 Kubernetes 一起运行。 [Portworx](https://portworx.com/use-case/kubernetes-storage/) 支持对服务器上存储的指纹处理、基于存储能力进行分层以及跨多个服务器整合存储容量。 Portworx 可以以 in-guest 方式在虚拟机中运行，也可以在裸金属 Linux 节点上运行。

portworxVolume 类型的卷可以通过 Kubernetes 动态创建，也可以在 Kubernetes Pod 内预先供应和引用。 下面是一个引用预先配置的 PortworxVolume 的示例 Pod：

apiVersion: v1

kind: Pod

metadata:

name: test-portworx-volume-pod

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /mnt

name: pxvol

volumes:

- name: pxvol

*# This Portworx volume must already exist.*

portworxVolume:

volumeID: "pxvol"

fsType: "<fs-type>"

**警告:**

在 Pod 中使用 portworxVolume 之前，请确保有一个名为 pxvol 的 PortworxVolume 存在。

更多详情和示例可以在[这里](https://github.com/kubernetes/examples/tree/master/staging/volumes/portworx/README.md)找到。

#### quobyte

quobyte 卷允许将现有的 [Quobyte](http://www.quobyte.com/) 卷挂载到您的 Pod 中。

**警告:**

在使用 Quobyte 卷之前，您首先要进行安装并创建好卷。

Quobyte 支持[容器存储接口](https://kubernetes.io/docs/concepts/storage/volumes/#csi)。 推荐使用 CSI 插件以在 Kubernetes 中使用 Quobyte 卷。 Quobyte 的 GitHub 项目具有[说明](https://github.com/quobyte/quobyte-csi#quobyte-csi)以及使用示例来部署 CSI 的 Quobyte。

#### rbd

rbd 卷允许将 [Rados 块设备](http://ceph.com/docs/master/rbd/rbd/) 卷挂载到您的 Pod 中. 不像 emptyDir 那样会在删除 Pod 的同时也会被删除，rbd 卷的内容在删除 Pod 时会被保存，卷只是被卸载掉了。 这意味着 rbd 卷可以被预先填充数据，并且这些数据可以在 Pod 之间”传递”。

**警告:**

在使用 RBD 之前，您必须安装运行 Ceph。

RBD 的一个特点是它可以同时被多个用户以只读方式挂载。 这意味着您可以用数据集预先填充卷，然后根据需要从尽可能多的 Pod 中并行地提供卷。 不幸的是，RBD 卷只能由单个使用者以读写模式安装——不允许同时写入。

更多详情请参考 [RBD 示例](https://github.com/kubernetes/examples/tree/master/volumes/rbd)。

#### scaleIO

ScaleIO 是基于软件的存储平台，可以使用现有硬件来创建可伸缩的、共享的而且是网络化的块存储集群。 scaleIO 卷插件允许部署的 Pod 访问现有的 ScaleIO 卷（或者它可以动态地为持久卷申领提供新的卷，参见[ScaleIO 持久卷](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#scaleio))。

**警告:**

在使用前，您必须有个安装完毕且运行正常的 ScaleIO 集群，并且创建好了存储卷。

下面是配置了 ScaleIO 的 Pod 示例：

apiVersion: v1

kind: Pod

metadata:

name: pod-0

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: pod-0

volumeMounts:

- mountPath: /test-pd

name: vol-0

volumes:

- name: vol-0

scaleIO:

gateway: https://localhost:443/api

system: scaleio

protectionDomain: sd0

storagePool: sp1

volumeName: vol-0

secretRef:

name: sio-secret

fsType: xfs

更多详情，请参考 [ScaleIO 示例](https://github.com/kubernetes/examples/tree/master/staging/volumes/scaleio)。

#### secret

secret 卷用来给 Pod 传递敏感信息，例如密码。您可以将 secret 存储在 Kubernetes API 服务器上，然后以文件的形式挂在到 Pod 中，无需直接与 Kubernetes 耦合。 secret 卷由 tmpfs（基于 RAM 的文件系统）提供存储，因此它们永远不会被写入非易失性（持久化的）存储器。

**警告:**

使用前您必须在 Kubernetes API 中创建 secret。

**注意：**

容器以 [subPath](https://kubernetes.io/zh/docs/concepts/storage/volumes/#using-subpath) 卷的方式挂载 Secret 时，它将感知不到 Secret 的更新。

Secret 的更多详情请参考[这里](https://kubernetes.io/docs/user-guide/secrets)。

#### storageOS

storageos 卷允许将现有的 [StorageOS](https://www.storageos.com/) 卷挂载到您的 Pod 中。

StorageOS 在 Kubernetes 环境中以容器的形式运行，这使得应用能够从 Kubernetes 集群中的任何节点访问本地或关联的存储。 为应对节点失效状况，可以复制数据。 若需提高利用率和降低成本，可以考虑瘦配置（Thin Provisioning）和数据压缩。

作为其核心能力之一，StorageOS 为容器提供了可以通过文件系统访问的块存储。

StorageOS 容器需要 64 位的 Linux，并且没有其他的依赖关系。 StorageOS 提供免费的开发者授权许可。

**警告:**

您必须在每个希望访问 StorageOS 卷的或者将向存储资源池贡献存储容量的节点上运行 StorageOS 容器。 有关安装说明，请参阅 [StorageOS 文档](https://docs.storageos.com/)。

apiVersion: v1

kind: Pod

metadata:

labels:

name: redis

role: master

name: test-storageos-redis

spec:

containers:

- name: master

image: kubernetes/redis:v1

env:

- name: MASTER

value: "true"

ports:

- containerPort: 6379

volumeMounts:

- mountPath: /redis-master-data

name: redis-data

volumes:

- name: redis-data

storageos:

*# The `redis-vol01` volume must already exist within StorageOS in the `default` namespace.*

volumeName: redis-vol01

fsType: ext4

更多关于动态供应和持久卷申领的信息请参考 [StorageOS 示例](https://github.com/kubernetes/examples/blob/master/volumes/storageos)。

#### vsphereVolume

**注意：**

前提条件：配备了 vSphere 云驱动的 Kubernetes。云驱动的配置方法请参考 [vSphere 使用指南](https://vmware.github.io/vsphere-storage-for-kubernetes/documentation/)。

vsphereVolume 用来将 vSphere VMDK 卷挂载到您的 Pod 中。 在卸载卷时，卷的内容会被保留。 vSphereVolume 卷类型支持 VMFS 和 VSAN 数据仓库。

**警告:**

在挂载到 Pod 之前，您必须用下列方式之一创建 VMDK。

#### 创建 VMDK 卷

选择下列方式之一创建 VMDK。

* [使用 vmkfstools 创建](https://kubernetes.io/zh/docs/concepts/storage/volumes/#tabs-volumes-0)

#### vSphere VMDK 配置示例

apiVersion: v1

kind: Pod

metadata:

name: test-vmdk

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-vmdk

name: test-volume

volumes:

- name: test-volume

*# This VMDK volume must already exist.*

vsphereVolume:

volumePath: "[DatastoreName] volumes/myDisk"

fsType: ext4

更多示例可以在[这里](https://github.com/kubernetes/examples/tree/master/staging/volumes/vsphere)找到。

### 使用 subPath

有时，在单个 Pod 中共享卷以供多方使用是很有用的。 volumeMounts.subPath 属性可用于指定所引用的卷内的子路径，而不是其根路径。

下面是一个使用同一共享卷的、内含 LAMP 栈（Linux Apache Mysql PHP）的 Pod 的示例。 HTML 内容被映射到卷的 html 文件夹，数据库将被存储在卷的 mysql 文件夹中：

apiVersion: v1

kind: Pod

metadata:

name: my-lamp-site

spec:

containers:

- name: mysql

image: mysql

env:

- name: MYSQL\_ROOT\_PASSWORD

value: "rootpasswd"

volumeMounts:

- mountPath: /var/lib/mysql

name: site-data

subPath: mysql

- name: php

image: php:7.0-apache

volumeMounts:

- mountPath: /var/www/html

name: site-data

subPath: html

volumes:

- name: site-data

persistentVolumeClaim:

claimName: my-lamp-site-data

#### 使用带有扩展环境变量的 subPath

**FEATURE STATE:** Kubernetes v1.15 [beta](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

使用 subPathExpr 字段从 Downward API 环境变量构造 subPath 目录名。 在使用此特性之前，必须启用 VolumeSubpathEnvExpansion 功能开关。 subPath 和 subPathExpr 属性是互斥的。

在这个示例中，Pod 基于 Downward API 中的 Pod 名称，使用 subPathExpr 在 hostPath 卷 /var/log/pods 中创建目录 pod1。 主机目录 /var/log/pods/pod1 挂载到了容器的 /logs 中。

apiVersion: v1

kind: Pod

metadata:

name: pod1

spec:

containers:

- name: container1

env:

- name: POD\_NAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: metadata.name

image: busybox

command: [ "sh", "-c", "while [ true ]; do echo 'Hello'; sleep 10; done | tee -a /logs/hello.txt" ]

volumeMounts:

- name: workdir1

mountPath: /logs

subPathExpr: $(POD\_NAME)

restartPolicy: Never

volumes:

- name: workdir1

hostPath:

path: /var/log/pods

### 资源

emptyDir 卷的存储介质（磁盘、SSD 等）是由保存 kubelet 根目录（通常是 /var/lib/kubelet）的文件系统的介质确定。 emptyDir 卷或者 hostPath 卷可以消耗的空间没有限制，容器之间或 Pod 之间也没有隔离。

将来，我们希望 emptyDir 卷和 hostPath 卷能够使用 [resource](https://kubernetes.io/docs/user-guide/computeresources) 规范来请求一定量的空间，并且能够为具有多种介质类型的集群选择要使用的介质类型。

### Out-of-Tree 卷插件

Out-of-Tree 卷插件包括容器存储接口（CSI）和 FlexVolume。 它们使存储供应商能够创建自定义存储插件，而无需将它们添加到 Kubernetes 代码仓库。

在引入 CSI 和 FlexVolume 之前，所有卷插件（如上面列出的卷类型）都是 “in-tree” 的，这意味着它们是与 Kubernetes 的核心组件一同构建、链接、编译和交付的，并且这些插件都扩展了 Kubernetes 的核心 API。 这意味着向 Kubernetes 添加新的存储系统（卷插件）需要将代码合并到 Kubernetes 核心代码库中。

CSI 和 FlexVolume 都允许独立于 Kubernetes 代码库开发卷插件，并作为扩展部署（安装）在 Kubernetes 集群上。

对于希望创建 out-of-tree 卷插件的存储供应商，请参考[这个 FAQ](https://github.com/kubernetes/community/blob/master/sig-storage/volume-plugin-faq.md)。

#### CSI

**FEATURE STATE:** Kubernetes v1.10 [beta](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

[容器存储接口](https://github.com/container-storage-interface/spec/blob/master/spec.md) (CSI) 为容器编排系统（如 Kubernetes）定义标准接口，以将任意存储系统暴露给它们的容器工作负载。

更多详情请阅读 [CSI 设计方案](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/storage/container-storage-interface.md)。

CSI 的支持在 Kubernetes v1.9 中作为 alpha 特性引入，在 Kubernetes v1.10 中转为 beta 特性，并在 Kubernetes v1.13 正式 GA。

**注意：**

Kubernetes v1.13中不支持 CSI 规范版本0.2和0.3，并将在以后的版本中删除。

**注意：**

CSI驱动程序可能并非在所有Kubernetes版本中都兼容。 请查看特定CSI驱动程序的文档，以获取每个 Kubernetes 版本所支持的部署步骤以及兼容性列表。

一旦在 Kubernetes 集群上部署了 CSI 兼容卷驱动程序，用户就可以使用 csi 卷类型来关联、挂载 CSI 驱动程序暴露出来的卷。

csi 卷类型不支持来自 Pod 的直接引用，只能通过 PersistentVolumeClaim 对象在 Pod 中引用。

存储管理员可以使用以下字段来配置 CSI 持久卷：

* driver：指定要使用的卷驱动程序名称的字符串值。 这个值必须与 CSI 驱动程序在 GetPluginInfoResponse 中返回的值相对应；该接口定义在 [CSI 规范](https://github.com/container-storage-interface/spec/blob/master/spec.md#getplugininfo)中。 Kubernetes 使用所给的值来标识要调用的 CSI 驱动程序；CSI 驱动程序也使用该值来辨识哪些 PV 对象属于该 CSI 驱动程序。
* volumeHandle：唯一标识卷的字符串值。 该值必须与CSI 驱动程序在 CreateVolumeResponse 的 volume\_id 字段中返回的值相对应；接口定义在 [CSI spec](https://github.com/container-storageinterface/spec/blob/master/spec.md#createvolume) 中。 在所有对 CSI 卷驱动程序的调用中，引用该 CSI 卷时都使用此值作为 volume\_id 参数。
* readOnly：一个可选的布尔值，指示通过 ControllerPublished 关联该卷时是否设置该卷为只读。 默认值是 false。 该值通过 ControllerPublishVolumeRequest 中的 readonly 字段传递给 CSI 驱动程序。
* fsType：如果 PV 的 VolumeMode 为 Filesystem，那么此字段指定挂载卷时应该使用的文件系统。 如果卷尚未格式化，并且支持格式化，此值将用于格式化卷。 此值可以通过 ControllerPublishVolumeRequest、NodeStageVolumeRequest 和 NodePublishVolumeRequest 的 VolumeCapability 字段传递给 CSI 驱动。
* volumeAttributes：一个字符串到字符串的映射表，用来设置卷的静态属性。 该映射必须与 CSI 驱动程序返回的 CreateVolumeResponse 中的 volume.attributes 字段的映射相对应；[CSI 规范](https://github.com/container-storage-interface/spec/blob/master/spec.md#createvolume) 中有相应的定义。 该映射通过ControllerPublishVolumeRequest、NodeStageVolumeRequest、和 NodePublishVolumeRequest 中的 volume\_attributes 字段传递给 CSI 驱动。
* controllerPublishSecretRef：对包含敏感信息的 secret 对象的引用；该敏感信息会被传递给 CSI 驱动来完成 CSI ControllerPublishVolume 和 ControllerUnpublishVolume 调用。 此字段是可选的；在不需要 secret 时可以是空的。 如果 secret 对象包含多个 secret，则所有的 secret 都会被传递。
* nodeStageSecretRef：对包含敏感信息的 secret 对象的引用，以传递给 CSI 驱动来完成 CSI NodeStageVolume 调用。 此字段是可选的，如果不需要 secret，则可能是空的。 如果 secret 对象包含多个 secret，则传递所有 secret。
* nodePublishSecretRef：对包含敏感信息的 secret 对象的引用，以传递给 CSI 驱动来完成 CSI `NodePublishVolume 调用。 此字段是可选的，如果不需要 secret，则可能是空的。 如果 secret 对象包含多个 secret，则传递所有 secret。

##### CSI 原始块卷支持

**FEATURE STATE:** Kubernetes v1.14 [beta](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

从 1.11 版本开始，CSI 引入了对原始块卷的支持。该特性依赖于在 Kubernetes 的之前版本中引入的原始块卷（Raw Block Volume）功能。 该特性将使具有外部 CSI 驱动程序的供应商能够在 Kubernetes 工作负载中实现原始块卷支持。

CSI块卷支持功能已启用，但默认情况下启用。必须为此功能启用的两个功能是“ BlockVolume”和“ CSIBlockVolume”。

--feature-gates=BlockVolume=true,CSIBlockVolume=true

学习怎样[安装您的带有块卷支持的 PV/PVC](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#raw-block-volume-support)。

##### CSI临时卷

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

此功能使 CSI 卷可以直接嵌入 Pod 规范中，而不是 PersistentVolume 中。 以这种方式指定的卷是临时的，不会在 Pod 重新启动后持续存在。

实例：

kind: Pod

apiVersion: v1

metadata:

name: my-csi-app

spec:

containers:

- name: my-frontend

image: busybox

volumeMounts:

- mountPath: "/data"

name: my-csi-inline-vol

command: [ "sleep", "1000000" ]

volumes:

- name: my-csi-inline-vol

csi:

driver: inline.storage.kubernetes.io

volumeAttributes:

foo: bar

此功能需要启用 CSIInlineVolume 功能门。 从Kubernetes 1.16开始默认启用它。

CSI 临时卷仅由一部分 CSI 驱动程序支持。 请在[此处](https://kubernetes-csi.github.io/docs/drivers.html)查看 CSI 驱动程序列表。

＃开发人员资源 有关如何开发 CSI 驱动程序的更多信息，请参考[kubernetes-csi文档](https://kubernetes-csi.github.io/docs/)

##### 从 in-tree 插件迁移到 CSI 驱动程序

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volumes/)

启用 CSI 迁移功能后，会将针对现有 in-tree 插件的操作定向到相应的 CSI 插件（应安装和配置）。 该功能实现了必要的转换逻辑和填充以无缝方式重新路由操作。 因此，操作员在过渡到取代树内插件的CSI驱动程序时，无需对现有存储类，PV 或 PVC（指 in-tree 插件)进行任何配置更改。 在 Alpha 状态下，受支持的操作和功能包括供应/删除，附加/分离，安装/卸载和调整卷大小。 上面的 “卷类型” 部分列出了支持 CSI 迁移并已实现相应 CSI 驱动程序的树内插件。

#### FlexVolume

FlexVolume 是一个自 1.2 版本（在 CSI 之前）以来在 Kubernetes 中一直存在的 out-of-tree 插件接口。 它使用基于 exec 的模型来与驱动程序对接。 用户必须在每个节点（在某些情况下是主节点）上的预定义卷插件路径中安装 FlexVolume 驱动程序可执行文件。

Pod 通过 flexvolume in-tree 插件与 Flexvolume 驱动程序交互。 更多详情请参考[这里](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-storage/flexvolume.md)。

### 挂载卷的传播

挂载卷的传播能力允许将容器安装的卷共享到同一 Pod 中的其他容器，甚至共享到同一节点上的其他 Pod。

卷的挂载传播特性由 Container.volumeMounts 中的 mountPropagation 字段控制。 它的值包括：

* None - 此卷挂载将不会感知到主机后续在此卷或其任何子目录上执行的挂载变化。 类似的，容器所创建的卷挂载在主机上是不可见的。这是默认模式。 该模式等同于 [Linux 内核文档](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt)中描述的 private 挂载传播选项。
* HostToContainer - 此卷挂载将会感知到主机后续针对此卷或其任何子目录的挂载操作。

换句话说，如果主机在此挂载卷中挂载任何内容，容器将能看到它被挂载在那里。

类似的，配置了 Bidirectional 挂载传播选项的 Pod 如果在同一卷上挂载了内容，挂载传播设置为 HostToContainer 的容器都将能看到这一变化。

该模式等同于 [Linux 内核文档](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt) 中描述的 rslave 挂载传播选项。

* Bidirectional - 这种卷挂载和 HostToContainer 挂载表现相同。

另外，容器创建的卷挂载将被传播回至主机和使用同一卷的所有 Pod 的所有容器。

该模式等同于 [Linux 内核文档](https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt) 中描述的 rshared 挂载传播选项。

**警告:**

Bidirectional 形式的挂载传播可能比较危险。 它可以破坏主机操作系统，因此它只被允许在特权容器中使用。 强烈建议您熟悉 Linux 内核行为。 此外，由 Pod 中的容器创建的任何卷挂载必须在终止时由容器销毁（卸载）。

#### 配置

在某些部署环境中，挂载传播正常工作前，必须在 Docker 中正确配置挂载共享（mount share），如下所示。

编辑您的 Docker systemd 服务文件，按下面的方法设置 MountFlags：

MountFlags=shared

或者，如果存在 MountFlags=slave 就删除掉。然后重启 Docker 守护进程：

sudo systemctl daemon-reload

sudo systemctl restart docker

### 接下来

* 参考[使用持久卷部署 WordPress 和 MySQL](https://kubernetes.io/zh/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/) 示例。

### 反馈

## Persistent Volumes

This document describes the current state of PersistentVolumes in Kubernetes. Familiarity with [volumes](https://kubernetes.io/docs/concepts/storage/volumes/) is suggested.

* [**Introduction**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#introduction)
* [**Lifecycle of a volume and claim**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#lifecycle-of-a-volume-and-claim)
* [**Types of Persistent Volumes**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes)
* [**Persistent Volumes**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistent-volumes)
* [**PersistentVolumeClaims**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims)
* [**Claims As Volumes**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#claims-as-volumes)
* [**Raw Block Volume Support**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#raw-block-volume-support)
* [**Volume Snapshot and Restore Volume from Snapshot Support**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#volume-snapshot-and-restore-volume-from-snapshot-support)
* [**Volume Cloning**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#volume-cloning)
* [**Writing Portable Configuration**](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#writing-portable-configuration)

### Introduction

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using [Storage Classes](https://kubernetes.io/docs/concepts/storage/storage-classes/). It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted once read/write or many times read-only).

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the StorageClass resource.

See the [detailed walkthrough with working examples](https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/).

### Lifecycle of a volume and claim

PVs are resources in the cluster. PVCs are requests for those resources and also act as claim checks to the resource. The interaction between PVs and PVCs follows this lifecycle:

#### Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

#### Static

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.

#### Dynamic

When none of the static PVs the administrator created match a user’s PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses: the PVC must request a [storage class](https://kubernetes.io/docs/concepts/storage/storage-classes/) and the administrator must have created and configured that class for dynamic provisioning to occur. Claims that request the class "" effectively disable dynamic provisioning for themselves.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the DefaultStorageClass [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#defaultstorageclass) on the API server. This can be done, for example, by ensuring that DefaultStorageClass is among the comma-delimited, ordered list of values for the --enable-admission-plugins flag of the API server component. For more information on API server command-line flags, check [kube-apiserver](https://kubernetes.io/docs/admin/kube-apiserver/) documentation.

#### Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes. A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC. Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested. Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping.

Claims will remain unbound indefinitely if a matching volume does not exist. Claims will be bound as matching volumes become available. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

#### Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod. For volumes that support multiple access modes, the user specifies which mode is desired when using their claim as a volume in a Pod.

Once a user has a claim and that claim is bound, the bound PV belongs to the user for as long as they need it. Users schedule Pods and access their claimed PVs by including a persistentVolumeClaim in their Pod’s volumes block. [See below for syntax details](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#claims-as-volumes).

#### Storage Object in Use Protection

The purpose of the Storage Object in Use Protection feature is to ensure that Persistent Volume Claims (PVCs) in active use by a Pod and Persistent Volume (PVs) that are bound to PVCs are not removed from the system, as this may result in data loss.

**Note:** PVC is in active use by a Pod when a Pod object exists that is using the PVC.

If a user deletes a PVC in active use by a Pod, the PVC is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods. Also, if an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC.

You can see that a PVC is protected when the PVC’s status is Terminating and the Finalizers list includes kubernetes.io/pvc-protection:

kubectl describe pvc hostpath

Name: hostpath

Namespace: default

StorageClass: example-hostpath

Status: Terminating

Volume:

Labels: <none>

Annotations: volume.beta.kubernetes.io/storage-class=example-hostpath

volume.beta.kubernetes.io/storage-provisioner=example.com/hostpath

Finalizers: [kubernetes.io/pvc-protection]

...

You can see that a PV is protected when the PV’s status is Terminating and the Finalizers list includes kubernetes.io/pv-protection too:

kubectl describe pv task-pv-volume

Name: task-pv-volume

Labels: type=local

Annotations: <none>

Finalizers: [kubernetes.io/pv-protection]

StorageClass: standard

Status: Available

Claim:

Reclaim Policy: Delete

Access Modes: RWO

Capacity: 1Gi

Message:

Source:

Type: HostPath (bare host directory volume)

Path: /tmp/data

HostPathType:

Events: <none>

##### Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be Retained, Recycled, or Deleted.

#### Retain

The Retain reclaim policy allows for manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered “released”. But it is not yet available for another claim because the previous claimant’s data remains on the volume. An administrator can manually reclaim the volume with the following steps.

1. Delete the PersistentVolume. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
2. Manually clean up the data on the associated storage asset accordingly.
3. Manually delete the associated storage asset, or if you want to reuse the same storage asset, create a new PersistentVolume with the storage asset definition.

#### Delete

For volume plugins that support the Delete reclaim policy, deletion removes both the PersistentVolume object from Kubernetes, as well as the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume. Volumes that were dynamically provisioned inherit the [reclaim policy of their StorageClass](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#reclaim-policy), which defaults to Delete. The administrator should configure the StorageClass according to users’ expectations; otherwise, the PV must be edited or patched after it is created. See [Change the Reclaim Policy of a PersistentVolume](https://kubernetes.io/docs/tasks/administer-cluster/change-pv-reclaim-policy/).

#### Recycle

**Warning:** The Recycle reclaim policy is deprecated. Instead, the recommended approach is to use dynamic provisioning.

If supported by the underlying volume plugin, the Recycle reclaim policy performs a basic scrub (rm -rf /thevolume/\*) on the volume and makes it available again for a new claim.

However, an administrator can configure a custom recycler Pod template using the Kubernetes controller manager command line arguments as described [here](https://kubernetes.io/docs/admin/kube-controller-manager/). The custom recycler Pod template must contain a volumes specification, as shown in the example below:

apiVersion: v1

kind: Pod

metadata:

name: pv-recycler

namespace: default

spec:

restartPolicy: Never

volumes:

- name: vol

hostPath:

path: /any/path/it/will/be/replaced

containers:

- name: pv-recycler

image: "k8s.gcr.io/busybox"

command: ["/bin/sh", "-c", "test -e /scrub && rm -rf /scrub/..?\* /scrub/.[!.]\* /scrub/\* && test -z \"$(ls -A /scrub)\" || exit 1"]

volumeMounts:

- name: vol

mountPath: /scrub

However, the particular path specified in the custom recycler Pod template in the volumes part is replaced with the particular path of the volume that is being recycled.

#### Expanding Persistent Volumes Claims

**FEATURE STATE:** Kubernetes v1.11 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Support for expanding PersistentVolumeClaims (PVCs) is now enabled by default. You can expand the following types of volumes:

* gcePersistentDisk
* awsElasticBlockStore
* Cinder
* glusterfs
* rbd
* Azure File
* Azure Disk
* Portworx
* FlexVolumes
* CSI

You can only expand a PVC if its storage class’s allowVolumeExpansion field is set to true.

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: gluster-vol-default

provisioner: kubernetes.io/glusterfs

parameters:

resturl: "http://192.168.10.100:8080"

restuser: ""

secretNamespace: ""

secretName: ""

allowVolumeExpansion: **true**

To request a larger volume for a PVC, edit the PVC object and specify a larger size. This triggers expansion of the volume that backs the underlying PersistentVolume. A new PersistentVolume is never created to satisfy the claim. Instead, an existing volume is resized.

#### CSI Volume expansion

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Support for expanding CSI volumes is enabled by default but it also requires a specific CSI driver to support volume expansion. Refer to documentation of the specific CSI driver for more information.

#### Resizing a volume containing a file system

You can only resize volumes containing a file system if the file system is XFS, Ext3, or Ext4.

When a volume contains a file system, the file system is only resized when a new Pod is using the PersistentVolumeClaim in ReadWrite mode. File system expansion is either done when a Pod is starting up or when a Pod is running and the underlying file system supports online expansion.

FlexVolumes allow resize if the driver is set with the RequiresFSResize capability to true. The FlexVolume can be resized on Pod restart.

#### Resizing an in-use PersistentVolumeClaim

**FEATURE STATE:** Kubernetes v1.15 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

**Note:** Expanding in-use PVCs is available as beta since Kubernetes 1.15, and as alpha since 1.11. The ExpandInUsePersistentVolumes feature must be enabled, which is the case automatically for many clusters for beta features. Refer to the [feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) documentation for more information.

In this case, you don’t need to delete and recreate a Pod or deployment that is using an existing PVC. Any in-use PVC automatically becomes available to its Pod as soon as its file system has been expanded. This feature has no effect on PVCs that are not in use by a Pod or deployment. You must create a Pod that uses the PVC before the expansion can complete.

Similar to other volume types - FlexVolume volumes can also be expanded when in-use by a Pod.

**Note:** FlexVolume resize is possible only when the underlying driver supports resize.

**Note:** Expanding EBS volumes is a time-consuming operation. Also, there is a per-volume quota of one modification every 6 hours.

### Types of Persistent Volumes

PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:

* GCEPersistentDisk
* AWSElasticBlockStore
* AzureFile
* AzureDisk
* CSI
* FC (Fibre Channel)
* FlexVolume
* Flocker
* NFS
* iSCSI
* RBD (Ceph Block Device)
* CephFS
* Cinder (OpenStack block storage)
* Glusterfs
* VsphereVolume
* Quobyte Volumes
* HostPath (Single node testing only – local storage is not supported in any way and WILL NOT WORK in a multi-node cluster)
* Portworx Volumes
* ScaleIO Volumes
* StorageOS

### Persistent Volumes

Each PV contains a spec and status, which is the specification and status of the volume.

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv0003

spec:

capacity:

storage: 5Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Recycle

storageClassName: slow

mountOptions:

- hard

- nfsvers=4.1

nfs:

path: /tmp

server: 172.17.0.2

#### Capacity

Generally, a PV will have a specific storage capacity. This is set using the PV’s capacity attribute. See the Kubernetes [Resource Model](https://git.k8s.io/community/contributors/design-proposals/scheduling/resources.md) to understand the units expected by capacity.

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

#### Volume Mode

**FEATURE STATE:** Kubernetes v1.13 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Prior to Kubernetes 1.9, all volume plugins created a filesystem on the persistent volume. Now, you can set the value of volumeMode to block to use a raw block device, or filesystem to use a filesystem. filesystem is the default if the value is omitted. This is an optional API parameter.

#### Access Modes

A PersistentVolume can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV’s access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV’s capabilities.

The access modes are:

* ReadWriteOnce – the volume can be mounted as read-write by a single node
* ReadOnlyMany – the volume can be mounted read-only by many nodes
* ReadWriteMany – the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

* RWO - ReadWriteOnce
* ROX - ReadOnlyMany
* RWX - ReadWriteMany

**Important!** A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCEPersistentDisk can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes, but not at the same time.

| Volume Plugin | ReadWriteOnce | ReadOnlyMany | ReadWriteMany |
| --- | --- | --- | --- |
| AWSElasticBlockStore | ✓ | - | - |
| AzureFile | ✓ | ✓ | ✓ |
| AzureDisk | ✓ | - | - |
| CephFS | ✓ | ✓ | ✓ |
| Cinder | ✓ | - | - |
| CSI | depends on the driver | depends on the driver | depends on the driver |
| FC | ✓ | ✓ | - |
| FlexVolume | ✓ | ✓ | depends on the driver |
| Flocker | ✓ | - | - |
| GCEPersistentDisk | ✓ | ✓ | - |
| Glusterfs | ✓ | ✓ | ✓ |
| HostPath | ✓ | - | - |
| iSCSI | ✓ | ✓ | - |
| Quobyte | ✓ | ✓ | ✓ |
| NFS | ✓ | ✓ | ✓ |
| RBD | ✓ | ✓ | - |
| VsphereVolume | ✓ | - | - (works when Pods are collocated) |
| PortworxVolume | ✓ | - | ✓ |
| ScaleIO | ✓ | ✓ | - |
| StorageOS | ✓ | - | - |

#### Class

A PV can have a class, which is specified by setting the storageClassName attribute to the name of a [StorageClass](https://kubernetes.io/docs/concepts/storage/storage-classes/). A PV of a particular class can only be bound to PVCs requesting that class. A PV with no storageClassName has no class and can only be bound to PVCs that request no particular class.

In the past, the annotation volume.beta.kubernetes.io/storage-class was used instead of the storageClassName attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

#### Reclaim Policy

Current reclaim policies are:

* Retain – manual reclamation
* Recycle – basic scrub (rm -rf /thevolume/\*)
* Delete – associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted

Currently, only NFS and HostPath support recycling. AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

#### Mount Options

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

**Note:** Not all Persistent Volume types support mount options.

The following volume types support mount options:

* AWSElasticBlockStore
* AzureDisk
* AzureFile
* CephFS
* Cinder (OpenStack block storage)
* GCEPersistentDisk
* Glusterfs
* NFS
* Quobyte Volumes
* RBD (Ceph Block Device)
* StorageOS
* VsphereVolume
* iSCSI

Mount options are not validated, so mount will simply fail if one is invalid.

In the past, the annotation volume.beta.kubernetes.io/mount-options was used instead of the mountOptions attribute. This annotation is still working; however, it will become fully deprecated in a future Kubernetes release.

#### Node Affinity

**Note:** For most volume types, you do not need to set this field. It is automatically populated for [AWS EBS](https://kubernetes.io/docs/concepts/storage/volumes/#awselasticblockstore), [GCE PD](https://kubernetes.io/docs/concepts/storage/volumes/#gcepersistentdisk) and [Azure Disk](https://kubernetes.io/docs/concepts/storage/volumes/#azuredisk) volume block types. You need to explicitly set this for [local](https://kubernetes.io/docs/concepts/storage/volumes/#local) volumes.

A PV can specify [node affinity](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#volumenodeaffinity-v1-core) to define constraints that limit what nodes this volume can be accessed from. Pods that use a PV will only be scheduled to nodes that are selected by the node affinity.

#### Phase

A volume will be in one of the following phases:

* Available – a free resource that is not yet bound to a claim
* Bound – the volume is bound to a claim
* Released – the claim has been deleted, but the resource is not yet reclaimed by the cluster
* Failed – the volume has failed its automatic reclamation

The CLI will show the name of the PVC bound to the PV.

### PersistentVolumeClaims

Each PVC contains a spec and status, which is the specification and status of the claim.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: myclaim

spec:

accessModes:

- ReadWriteOnce

volumeMode: Filesystem

resources:

requests:

storage: 8Gi

storageClassName: slow

selector:

matchLabels:

release: "stable"

matchExpressions:

- {key: environment, operator: In, values: [dev]}

#### Access Modes

Claims use the same conventions as volumes when requesting storage with specific access modes.

#### Volume Modes

Claims use the same convention as volumes to indicate the consumption of the volume as either a filesystem or block device.

#### Resources

Claims, like Pods, can request specific quantities of a resource. In this case, the request is for storage. The same [resource model](https://git.k8s.io/community/contributors/design-proposals/scheduling/resources.md) applies to both volumes and claims.

#### Selector

Claims can specify a [label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors) to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim. The selector can consist of two fields:

* matchLabels - the volume must have a label with this value
* matchExpressions - a list of requirements made by specifying key, list of values, and operator that relates the key and values. Valid operators include In, NotIn, Exists, and DoesNotExist.

All of the requirements, from both matchLabels and matchExpressions, are ANDed together – they must all be satisfied in order to match.

#### Class

A claim can request a particular class by specifying the name of a [StorageClass](https://kubernetes.io/docs/concepts/storage/storage-classes/) using the attribute storageClassName. Only PVs of the requested class, ones with the same storageClassName as the PVC, can be bound to the PVC.

PVCs don’t necessarily have to request a class. A PVC with its storageClassName set equal to "" is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to ""). A PVC with no storageClassName is not quite the same and is treated differently by the cluster, depending on whether the [DefaultStorageClass admission plugin](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#defaultstorageclass) is turned on.

* If the admission plugin is turned on, the administrator may specify a default StorageClass. All PVCs that have no storageClassName can be bound only to PVs of that default. Specifying a default StorageClass is done by setting the annotation storageclass.kubernetes.io/is-default-class equal to true in a StorageClass object. If the administrator does not specify a default, the cluster responds to PVC creation as if the admission plugin were turned off. If more than one default is specified, the admission plugin forbids the creation of all PVCs.
* If the admission plugin is turned off, there is no notion of a default StorageClass. All PVCs that have no storageClassName can be bound only to PVs that have no class. In this case, the PVCs that have no storageClassName are treated the same way as PVCs that have their storageClassName set to "".

Depending on installation method, a default StorageClass may be deployed to a Kubernetes cluster by addon manager during installation.

When a PVC specifies a selector in addition to requesting a StorageClass, the requirements are ANDed together: only a PV of the requested class and with the requested labels may be bound to the PVC.

**Note:** Currently, a PVC with a non-empty selector can’t have a PV dynamically provisioned for it.

In the past, the annotation volume.beta.kubernetes.io/storage-class was used instead of storageClassName attribute. This annotation is still working; however, it won’t be supported in a future Kubernetes release.

### Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the Pod using the claim. The cluster finds the claim in the Pod’s namespace and uses it to get the PersistentVolume backing the claim. The volume is then mounted to the host and into the Pod.

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: myfrontend

image: nginx

volumeMounts:

- mountPath: "/var/www/html"

name: mypd

volumes:

- name: mypd

persistentVolumeClaim:

claimName: myclaim

#### A Note on Namespaces

PersistentVolumes binds are exclusive, and since PersistentVolumeClaims are namespaced objects, mounting claims with “Many” modes (ROX, RWX) is only possible within one namespace.

### Raw Block Volume Support

**FEATURE STATE:** Kubernetes v1.13 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

The following volume plugins support raw block volumes, including dynamic provisioning where applicable:

* AWSElasticBlockStore
* AzureDisk
* FC (Fibre Channel)
* GCEPersistentDisk
* iSCSI
* Local volume
* RBD (Ceph Block Device)
* VsphereVolume (alpha)

**Note:** Only FC and iSCSI volumes supported raw block volumes in Kubernetes 1.9. Support for the additional plugins was added in 1.10.

#### Persistent Volumes using a Raw Block Volume

apiVersion: v1

kind: PersistentVolume

metadata:

name: block-pv

spec:

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

volumeMode: Block

persistentVolumeReclaimPolicy: Retain

fc:

targetWWNs: ["50060e801049cfd1"]

lun: 0

readOnly: **false**

#### Persistent Volume Claim requesting a Raw Block Volume

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: block-pvc

spec:

accessModes:

- ReadWriteOnce

volumeMode: Block

resources:

requests:

storage: 10Gi

#### Pod specification adding Raw Block Device path in container

apiVersion: v1

kind: Pod

metadata:

name: pod-with-block-volume

spec:

containers:

- name: fc-container

image: fedora:26

command: ["/bin/sh", "-c"]

args: [ "tail -f /dev/null" ]

volumeDevices:

- name: data

devicePath: /dev/xvda

volumes:

- name: data

persistentVolumeClaim:

claimName: block-pvc

**Note:** When adding a raw block device for a Pod, you specify the device path in the container instead of a mount path.

#### Binding Block Volumes

If a user requests a raw block volume by indicating this using the volumeMode field in the PersistentVolumeClaim spec, the binding rules differ slightly from previous releases that didn’t consider this mode as part of the spec. Listed is a table of possible combinations the user and admin might specify for requesting a raw block device. The table indicates if the volume will be bound or not given the combinations: Volume binding matrix for statically provisioned volumes:

| PV volumeMode | PVC volumeMode | Result |
| --- | --- | --- |
| unspecified | unspecified | BIND |
| unspecified | Block | NO BIND |
| unspecified | Filesystem | BIND |
| Block | unspecified | NO BIND |
| Block | Block | BIND |
| Block | Filesystem | NO BIND |
| Filesystem | Filesystem | BIND |
| Filesystem | Block | NO BIND |
| Filesystem | unspecified | BIND |

**Note:** Only statically provisioned volumes are supported for alpha release. Administrators should take care to consider these values when working with raw block devices.

### Volume Snapshot and Restore Volume from Snapshot Support

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Volume snapshot feature was added to support CSI Volume Plugins only. For details, see [volume snapshots](https://kubernetes.io/docs/concepts/storage/volume-snapshots/).

To enable support for restoring a volume from a volume snapshot data source, enable the VolumeSnapshotDataSource feature gate on the apiserver and controller-manager.

#### Create Persistent Volume Claim from Volume Snapshot

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: restore-pvc

spec:

storageClassName: csi-hostpath-sc

dataSource:

name: new-snapshot-test

kind: VolumeSnapshot

apiGroup: snapshot.storage.k8s.io

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

### Volume Cloning

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)

Volume clone feature was added to support CSI Volume Plugins only. For details, see [volume cloning](https://kubernetes.io/docs/concepts/storage/volume-pvc-datasource/).

To enable support for cloning a volume from a PVC data source, enable the VolumePVCDataSource feature gate on the apiserver and controller-manager.

#### Create Persistent Volume Claim from an existing pvc

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: cloned-pvc

spec:

storageClassName: my-csi-plugin

dataSource:

name: existing-src-pvc-name

kind: PersistentVolumeClaim

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 10Gi

### Writing Portable Configuration

If you’re writing configuration templates or examples that run on a wide range of clusters and need persistent storage, it is recommended that you use the following pattern:

* Include PersistentVolumeClaim objects in your bundle of config (alongside Deployments, ConfigMaps, etc).
* Do not include PersistentVolume objects in the config, since the user instantiating the config may not have permission to create PersistentVolumes.
* Give the user the option of providing a storage class name when instantiating the template.
  + If the user provides a storage class name, put that value into the persistentVolumeClaim.storageClassName field. This will cause the PVC to match the right storage class if the cluster has StorageClasses enabled by the admin.
  + If the user does not provide a storage class name, leave the persistentVolumeClaim.storageClassName field as nil. This will cause a PV to be automatically provisioned for the user with the default StorageClass in the cluster. Many cluster environments have a default StorageClass installed, or administrators can create their own default StorageClass.
* In your tooling, watch for PVCs that are not getting bound after some time and surface this to the user, as this may indicate that the cluster has no dynamic storage support (in which case the user should create a matching PV) or the cluster has no storage system (in which case the user cannot deploy config requiring PVCs).

#### Feedback

## 卷快照

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/)

本文档描述 Kubernetes 中 VolumeSnapshots 的当前状态。建议先熟悉[持久卷](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)。

* [**介绍**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/#%e4%bb%8b%e7%bb%8d)
* [**卷快照和卷快照内容的生命周期**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/#%e5%8d%b7%e5%bf%ab%e7%85%a7%e5%92%8c%e5%8d%b7%e5%bf%ab%e7%85%a7%e5%86%85%e5%ae%b9%e7%9a%84%e7%94%9f%e5%91%bd%e5%91%a8%e6%9c%9f)
* [**卷快照内容**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/#%e5%8d%b7%e5%bf%ab%e7%85%a7%e5%86%85%e5%ae%b9)
* [**VolumeSnapshots**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/#volumesnapshots)
* [**从快照供应卷**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshots/#%e4%bb%8e%e5%bf%ab%e7%85%a7%e4%be%9b%e5%ba%94%e5%8d%b7)

### 介绍

与 PersistentVolume 和 PersistentVolumeClaim 两个 API 资源用于给用户和管理员提供卷类似，VolumeSnapshotContent 和 VolumeSnapshot 两个 API 资源用于给用户和管理员创建卷快照。

VolumeSnapshotContent 是从管理员已提供的集群中的卷获取的快照。就像持久卷是集群的资源一样，它也是集群中的资源。

VolumeSnapshot 是用户对于卷的快照的请求。它类似于持久卷声明。

VolumeSnapshots 允许用户消费抽象的存储资源，集群管理员需要能够提供多种 VolumeSnapshotContents，又不会向用户暴露这些应该供应的卷快照的细节。为了实现这些需求，就需要 VolumeSnapshotClass 资源。

当使用该功能时，用户需要注意以下几点：

* API 对象 VolumeSnapshot，VolumeSnapshotContent 和 VolumeSnapshotClass 是 CRD，不是核心 API 的部分。
* VolumeSnapshot 支持仅可用于 CSI 驱动。
* 作为部署过程的一部分，Kubernetes 团队为快照控制器提供了一个名为 external-snapshotter 的 sidecar 帮助容器。它监视 VolumeSnapshot 对象然后向 CSI 端点触发 CreateSnapshot 和 DeleteSnapshot 操作。
* CSI 驱动可能实现，也可能没有实现卷快照功能。CSI 驱动可能会使用 external-snapshotter 来提供对卷快照的支持。
* 支持卷快照的 CSI 驱动将自动安装 用于定义卷快照定义的 CRD。

### 卷快照和卷快照内容的生命周期

VolumeSnapshotContents 是集群中的资源。VolumeSnapshots 是对于这些资源的请求。VolumeSnapshotContents 和 VolumeSnapshots 之间的交互遵循以下生命周期：

#### 供应卷快照

快照可以通过两种方式进行供应：静态或动态。

##### 静态的

集群管理员创建多个 VolumeSnapshotContents。它们带有实际存储的详细信息，可以供集群用户使用。它们存在于 Kubernetes API 中，并且能够被使用。

##### 动态的

当管理员创建的静态 VolumeSnapshotContents 都不能匹配用户的 VolumeSnapshot，集群可能会尝试专门为 VolumeSnapshot 对象供应一个卷快照。此供应基于 VolumeSnapshotClasses：VolumeSnapshot 必须请求[卷快照类](https://kubernetes.io/docs/concepts/storage/volume-snapshot-classes/)并且管理员必须已经创建并配置了该类，才能进行动态供应。

#### 绑定

用户创建，或在动态供应场景下已经创建了的 VolumeSnapshot 具有特定数量的存储请求和特定的访问模式。一个控制循环监视新的 VolumeSnapshots，找到匹配的 VolumeSnapshotContent（如果可能），并把它们绑定到一起。如果 VolumeSnapshotContent 是给动态供应给一个新的VolumeSnapshot，循环将依然绑定 VolumeSnapshotContent 到 VolumeSnapshot。一旦绑定，无论是如何绑定的，VolumeSnapshot 绑定都是排他的。VolumeSnapshot 到 VolumeSnapshotContent 的绑定是一对一的映射。

如果不存在匹配的 VolumeSnapshotContent，VolumeSnapshots 将永远保持未绑定状态。当匹配的 VolumeSnapshotContents 可用时，将绑定 VolumeSnapshots。

#### 使用保护的持久性卷声明

使用保护的持久性卷声明功能的目的是确保使用中的 PVC API 对象不会从系统中被删除（因为这可能会导致数据丢失）。

如果一个 PVC 正在被快照用来作为源进行快照创建，则该 PVC 是使用中的。如果用户删除正作为快照源的 PVC API 对象，则 PVC 对象不会立即被删除掉。相反，PVC 对象的删除将推迟到任何快照不在主动使用它为止。当快照的 Status 中的 ReadyToUse值为 true 时，PVC 将不再用作快照源。

#### 删除

删除操作会从 Kubernetes API 中删除 VolumeSnapshotContent 对象，同时删除它在外部基础架构中关联存储资产。

### 卷快照内容

每个 VolumeSnapshotContent 包含一个 spec，用来表示卷快照的规格。

apiVersion: snapshot.storage.k8s.io/v1alpha1

kind: VolumeSnapshotContent

metadata:

name: new-snapshot-content-test

spec:

snapshotClassName: csi-hostpath-snapclass

source:

name: pvc-test

kind: PersistentVolumeClaim

volumeSnapshotSource:

csiVolumeSnapshotSource:

creationTime: 1535478900692119403

driver: csi-hostpath

restoreSize: 10Gi

snapshotHandle: 7bdd0de3-aaeb-11e8-9aae-0242ac110002

#### 类

VolumeSnapshotContent 可以具有一个类，该类通过设置 snapshotClassName 属性为 [VolumeSnapshotClass](https://kubernetes.io/docs/concepts/storage/volume-snapshot-classes/) 的名称来指定。一个特定类的 VolumeSnapshotContent 只能够绑定到请求该类的 VolumeSnapshots。没有 snapshotClassName 的 VolumeSnapshotContent 没有类，并且只能绑定到不要求特定类的 VolumeSnapshots。

### VolumeSnapshots

每个 VolumeSnapshot 对象包含 spec 和 status，用来表示卷快照的规格和状态。

apiVersion: snapshot.storage.k8s.io/v1alpha1

kind: VolumeSnapshot

metadata:

name: new-snapshot-test

spec:

snapshotClassName: csi-hostpath-snapclass

source:

name: pvc-test

kind: PersistentVolumeClaim

#### 类

通过使用 snapshotClassName 属性来指定 [VolumeSnapshotClass](https://kubernetes.io/docs/concepts/storage/volume-snapshot-classes/) 的名称，卷快照可以请求特定的类。 只有所请求类（与 VolumeSnapshot 有相同的 snapshotClassName）的 VolumeSnapshotContents 才可以绑定到 VolumeSnapshot。

### 从快照供应卷

你可以供应一个新卷，该卷预填充了快照中的数据，在 持久卷声明 对象中使用 dataSource 字段。

更多详细信息，请参阅 [卷快照和从快照还原卷](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#volume-snapshot-and-restore-volume-from-snapshot-support)。

#### 反馈

## CSI 卷克隆

**FEATURE STATE:** Kubernetes v1.16 [beta](https://kubernetes.io/zh/docs/concepts/storage/volume-pvc-datasource/)

本文档描述 Kubernetes 中克隆现有 CSI 卷的概念。建议先熟悉[卷](https://kubernetes.io/docs/concepts/storage/volumes)。

此功能需要启动 VolumePVCDataSource 功能门：

--feature-gates=VolumePVCDataSource=true

* [**介绍**](https://kubernetes.io/zh/docs/concepts/storage/volume-pvc-datasource/#%e4%bb%8b%e7%bb%8d)
* [**供应**](https://kubernetes.io/zh/docs/concepts/storage/volume-pvc-datasource/#%e4%be%9b%e5%ba%94)
* [**用法**](https://kubernetes.io/zh/docs/concepts/storage/volume-pvc-datasource/#%e7%94%a8%e6%b3%95)

### 介绍

[CSI](https://kubernetes.io/docs/concepts/storage/volumes/#csi) 卷克隆功能增加了在 dataSource 字段指定现有的 [PVC](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)s，来表示用户想要克隆的 [卷](https://kubernetes.io/docs/concepts/storage/volumes/)。

克隆定义为已有 Kubernetes 卷的副本，可以像任何标准卷一样被使用。唯一的区别就是配置后，后端设备将创建指定卷的精确副本，而不是创建一个“新的”空卷。

从 Kubernetes API 的角度看，克隆的实现只是在创建新的 PVC 时，增加了指定一个现有未绑定 PVC 作为数据源的能力。

用户在使用该功能时，需要注意以下事项：

* 克隆支持（VolumePVCDataSource）仅适用于 CSI 驱动。
* 克隆支持仅适用于 动态供应器。
* CSI 驱动可能实现，也可能未实现卷克隆功能。
* 仅当 PVC 与目标 PVC 存在于同一命名空间（源和目标 PVC 必须在相同的命名空间）时，才可以克隆 PVC。
* 仅在同一存储类中支持克隆。
  + 目标卷必须和源卷具有相同的存储类
  + 可以使用默认的存储类并且 storageClassName 字段在规格中忽略了

### 供应

克隆卷与其他任何 PVC 一样配置，除了需要增加 dataSource 来引用同一命名空间中现有的 PVC。

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: clone-of-pvc-1

namespace: myns

spec:

accessModes:

- ReadWriteOnce

storageClassName: cloning

resources:

requests:

storage: 5Gi

dataSource:

kind: PersistentVolumeClaim

name: pvc-1

结果是一个名称为 clone-of-pvc-1 的新 PVC 与指定的源 pvc-1 拥有相同的内容。

### 用法

一旦新的 PVC 可用，被克隆的 PVC 项其他 PVC 一样被使用。可以预期的是，新创建的 PVC 是一个独立的对象。可以独立使用，克隆，快照或删除它，而不需要考虑它的原始数据源 PVC。这也意味着，源没有以任何方式链接到新创建的 PVC，它也可以被修改或删除，而不会影响到新创建的克隆。

### 反馈

## Storage Classes

本文描述了 Kubernetes 中 StorageClass 的概念。建议先熟悉 [卷](https://kubernetes.io/docs/concepts/storage/volumes/) 和 [持久卷](https://kubernetes.io/docs/concepts/storage/persistent-volumes) 的概念。

* [**介绍**](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#%e4%bb%8b%e7%bb%8d)
* [**StorageClass 资源**](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#storageclass-%e8%b5%84%e6%ba%90)
* [**参数**](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#%e5%8f%82%e6%95%b0)

### 介绍

StorageClass 为管理员提供了描述存储 "类" 的方法。 不同的类型可能会映射到不同的服务质量等级或备份策略，或是由群集管理员制定的任意策略。 Kubernetes 本身并不清楚各种类代表的什么。这个类的概念在其他存储系统中有时被称为”配置文件”。

### StorageClass 资源

每个 StorageClass 都包含 provisioner、parameters 和 reclaimPolicy 字段， 这些字段会在StorageClass需要动态分配 PersistentVolume 时会使用到。

StorageClass 对象的命名很重要，用户使用这个命名来请求生成一个特定的类。 当创建 StorageClass 对象时，管理员设置 StorageClass 对象的命名和其他参数，一旦创建了对象就不能再对其更新。

管理员可以为没有申请绑定到特定 StorageClass 的 PVC 指定一个默认的类 ： 更多详情请参阅 [PersistentVolumeClaim 章节](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#persistentvolumeclaims)。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: standard

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2

reclaimPolicy: Retain

allowVolumeExpansion: **true**

mountOptions:

- debug

volumeBindingMode: Immediate

#### 存储分配器

StorageClass 有一个分配器，用来决定使用哪个卷插件分配持久化卷申领。该字段必须指定。

| 卷插件 | 提供厂商 | 配置例子 |
| --- | --- | --- |
| AWSElasticBlockStore | ✓ | [AWS EBS](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#aws-ebs) |
| AzureFile | ✓ | [Azure File](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#azure-file) |
| AzureDisk | ✓ | [Azure Disk](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#azure-disk) |
| CephFS | - | - |
| Cinder | ✓ | [OpenStack Cinder](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#openstack-cinder) |
| FC | - | - |
| FlexVolume | - | - |
| Flocker | ✓ | - |
| GCEPersistentDisk | ✓ | [GCE PD](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#gce-pd) |
| Glusterfs | ✓ | [Glusterfs](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#glusterfs) |
| iSCSI | - | - |
| Quobyte | ✓ | [Quobyte](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#quobyte) |
| NFS | - | - |
| RBD | ✓ | [Ceph RBD](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#ceph-rbd) |
| VsphereVolume | ✓ | [vSphere](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#vsphere) |
| PortworxVolume | ✓ | [Portworx Volume](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#portworx-volume) |
| ScaleIO | ✓ | [ScaleIO](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#scaleio) |
| StorageOS | ✓ | [StorageOS](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#storageos) |
| Local | - | [Local](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#local) |

您不限于指定此处列出的”内置”分配器（其名称前缀为 kubernetes.io 并打包在 Kubernetes 中）。 您还可以运行和指定外部分配器，这些独立的程序遵循由 Kubernetes 定义的 [规范](https://git.k8s.io/community/contributors/design-proposals/storage/volume-provisioning.md)。 外部供应商的作者完全可以自由决定他们的代码保存于何处、打包方式、运行方式、使用的插件（包括 Flex）等。 代码仓库 [kubernetes-sigs/sig-storage-lib-external-provisioner](https://github.com/kubernetes-sigs/sig-storage-lib-external-provisioner) 包含一个用于为外部分配器编写功能实现的类库。可以通过下面的代码仓库，查看外部分配器列表。

[kubernetes-incubator/external-storage](https://github.com/kubernetes-incubator/external-storage).

例如，NFS 没有内部分配器，但可以使用外部分配器。 也有第三方存储供应商提供自己的外部分配器。

#### 回收策略

由 StorageClass 动态创建的持久化卷会在的 reclaimPolicy 字段中指定回收策略，可以是 Delete 或者 Retain。如果 StorageClass 对象被创建时没有指定 reclaimPolicy ，它将默认为 Delete。

通过 StorageClass 手动创建并管理的 Persistent Volume 会使用它们被创建时指定的回收政策。

#### 允许卷扩展

**FEATURE STATE:** Kubernetes v1.11 [beta](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

永久卷可以配置为可扩展。将此功能设置为 true 时，允许用户通过编辑相应的PVC对象来调整卷大小。

当基础存储类的 allowVolumeExpansion 字段设置为true时，以下类型的卷支持卷扩展。

* gcePersistentDisk
* awsElasticBlockStore
* Cinder
* glusterfs
* rbd
* Azure File
* Azure Disk
* Portworx
* FlexVolumes
* CSI

**FEATURE STATE:** Kubernetes v1.14 [alpha](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

**注意：**

此功能不能用于缩小卷。

#### 挂载选项

由 StorageClass 动态创建的 Persistent Volume 将使用类中 mountOption 字段指定的挂载选项。

如果卷插件不支持挂载选项，却指定了该选项，则分配操作会失败。 挂载选项在 StorageClass 和持久卷上都不会做验证，所以如果挂载选项无效，那么这个 PV 就会失败。

#### 卷绑定模式

**FEATURE STATE:** Kubernetes v1.12 [beta](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

**注意：** 这个功能特性需要启用 VolumeScheduling 参数才能使用。

volumeBindingMode 字段控制了 [卷绑定和动态分配](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#provisioning) 应该发生在什么时候。

默认情况下，Immediate 模式表示一旦创建了 PersistentVolumeClaim 也就完成了卷绑定和动态分配。 对于由于拓扑限制而非集群所有节点可达的存储后端，PersistentVolume 会在不知道 Pod 调度要求的情况下绑定或者分配。

集群管理员可以通过指定 WaitForFirstConsumer 模式来解决此问题。 该模式将延迟 PersistentVolume 的绑定和分配，直到使用该 PersistentVolumeClaim 的 Pod 被创建。 PersistentVolume 会根据 Pod 调度约束指定的拓扑来选择或分配。这些包括但不限于 [资源需求](https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container)， [节点筛选器](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#nodeselector)， [pod 亲和性和互斥性](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity), 以及 [污点和容忍度](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration).

以下插件支持动态分配的 WaitForFirstConsumer 模式:

* [AWSElasticBlockStore](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#aws-ebs)
* [GCEPersistentDisk](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#gce-pd)
* [AzureDisk](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#azure-disk)

以下插件支持预创建绑定 PersistentVolume 的 WaitForFirstConsumer 模式：

* All of the above
* [Local](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#local)

**FEATURE STATE:** Kubernetes 1.14 [beta](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

动态配置和预先创建的PVs也支持 [CSI卷](https://kubernetes.io/docs/concepts/storage/volumes/#csi)， 但是您需要查看特定CSI驱动程序的文档以查看其支持的拓扑密钥和例子。 必须启用 CSINodeInfo 特性。

#### 允许的拓扑结构

**FEATURE STATE:** Kubernetes v1.12 [beta](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

**注意：** 这个特性需要开启 VolumeScheduling 特性开关。

当集群操作人员使用了 WaitForFirstConsumer 的卷绑定模式，在大部分情况下就没有必要将配置限制为特定的拓扑结构。 然而，如果还有需要的话，可以使用 allowedTopologies。

这个例子描述了如何将分配卷限的拓扑限制在特定的区域，在使用时应该根据插件支持情况替换 zone 和 zones 参数。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: standard

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-standard

volumeBindingMode: WaitForFirstConsumer

allowedTopologies:

- matchLabelExpressions:

- key: failure-domain.beta.kubernetes.io/zone

values:

- us-central1-a

- us-central1-b

### 参数

Storage class 具有描述属于卷的参数。取决于分配器，可以接受不同的参数。 例如，参数 type 的值 io1 和参数 iopsPerGB 特定于 EBS PV。当参数被省略时，会使用默认值。

#### AWS EBS

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: slow

provisioner: kubernetes.io/aws-ebs

parameters:

type: io1

iopsPerGB: "10"

fsType: ext4

* type：io1，gp2，sc1，st1。详细信息参见 [AWS 文档](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html)。默认值：gp2。
* zone(弃用)：AWS 区域。如果没有指定 zone 和 zones，通常卷会在 Kubernetes 集群节点所在的活动区域中轮询调度分配。zone 和 zones 参数不能同时使用。
* zones(弃用)：以逗号分隔的 AWS 区域列表。如果没有指定 zone 和 zones，通常卷会在 Kubernetes 集群节点所在的活动区域中轮询调度分配。zone和zones参数不能同时使用。
* iopsPerGB：只适用于 io1 卷。每 GiB 每秒 I/O 操作。AWS 卷插件将其与请求卷的大小相乘以计算 IOPS 的容量，并将其限制在 20 000 IOPS（AWS 支持的最高值，请参阅 [AWS 文档](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html)。 这里需要输入一个字符串，即 "10"，而不是 10。
* fsType：受 Kubernetes 支持的文件类型。默认值："ext4"。
* encrypted：指定 EBS 卷是否应该被加密。合法值为 "true" 或者 "false"。这里需要输入字符串，即 "true", 而非 true。
* kmsKeyId：可选。加密卷时使用密钥的完整 Amazon 资源名称。如果没有提供，但 encrypted 值为 true，AWS 生成一个密钥。关于有效的 ARN 值，请参阅 AWS 文档。

**注意：**

zone 和 zones 已被弃用并被 [允许的拓扑结构](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#allowed-topologies) 取代。

#### GCE PD

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: slow

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-standard

replication-type: none

* type：pd-standard 或者 pd-ssd。默认：pd-standard
* zone(弃用)：GCE 区域。如果没有指定 zone 和 zones，通常卷会在 Kubernetes 集群节点所在的活动区域中轮询调度分配。zone 和 zones 参数不能同时使用。
* zones(弃用)：逗号分隔的 GCE 区域列表。如果没有指定 zone 和 zones，通常卷会在 Kubernetes 集群节点所在的活动区域中轮询调度（round-robin）分配。zone 和 zones 参数不能同时使用。
* replication-type：none 或者 regional-pd。默认值：none。

如果 replication-type 设置为 none，会分配一个常规（当前区域内的）持久化磁盘。

如果 replication-type 设置为 regional-pd，会分配一个 [区域性持久化磁盘（Regional Persistent Disk）](https://cloud.google.com/compute/docs/disks/#repds)。在这种情况下，用户必须使用 zones 而非 zone 来指定期望的复制区域（zone）。如果指定来两个特定的区域，区域性持久化磁盘会在这两个区域里分配。如果指定了多于两个的区域，Kubernetes 会选择其中任意两个区域。如果省略了 zones 参数，Kubernetes 会在集群管理的区域中任意选择。

**注意：**

zone 和 zones 已被弃用并被 [allowedTopologies](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/#allowed-topologies) 取代。

#### Glusterfs

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: slow

provisioner: kubernetes.io/glusterfs

parameters:

resturl: "http://127.0.0.1:8081"

clusterid: "630372ccdc720a92c681fb928f27b53f"

restauthenabled: "true"

restuser: "admin"

secretNamespace: "default"

secretName: "heketi-secret"

gidMin: "40000"

gidMax: "50000"

volumetype: "replicate:3"

* resturl：分配 gluster 卷的需求的 Gluster REST 服务/Heketi 服务 url。 通用格式应该是 IPaddress:Port，这是 GlusterFS 动态分配器的必需参数。 如果 Heketi 服务在 openshift/kubernetes 中安装并暴露为可路由服务，则可以使用类似于 http://heketi-storage-project.cloudapps.mystorage.com 的格式，其中 fqdn 是可解析的 heketi 服务网址。
* restauthenabled：Gluster REST 服务身份验证布尔值，用于启用对 REST 服务器的身份验证。如果此值为 ‘true’，则必须填写 restuser 和 restuserkey 或 secretNamespace + secretName。此选项已弃用，当在指定 restuser，restuserkey，secretName 或 secretNamespace 时，身份验证被启用。
* restuser：在 Gluster 可信池中有权创建卷的 Gluster REST服务/Heketi 用户。
* restuserkey：Gluster REST 服务/Heketi 用户的密码将被用于对 REST 服务器进行身份验证。此参数已弃用，取而代之的是 secretNamespace + secretName。 <!–
* secretNamespace, secretName : Identification of Secret instance that contains user password to use when talking to Gluster REST service. These parameters are optional, empty password will be used when both secretNamespace and secretName are omitted. The provided secret must have type "kubernetes.io/glusterfs", e.g. created in this way:
* kubectl create secret generic heketi-secret \
* --type="kubernetes.io/glusterfs" --from-literal=key='opensesame' \
* --namespace=default

Example of a secret can be found in [glusterfs-provisioning-secret.yaml](https://github.com/kubernetes/examples/tree/master/staging/persistent-volume-provisioning/glusterfs/glusterfs-secret.yaml). –>

* secretNamespace，secretName：Secret 实例的标识，包含与 Gluster REST 服务交互时使用的用户密码。 这些参数是可选的，secretNamespace 和 secretName 都省略时使用空密码。所提供的 Secret 必须将类型设置为 “kubernetes.io/glusterfs”，例如以这种方式创建：
* kubectl create secret generic heketi-secret \
* --type="kubernetes.io/glusterfs" --from-literal=key='opensesame' \
* --namespace=default

secret 的例子可以在 [glusterfs-provisioning-secret.yaml](https://github.com/kubernetes/examples/tree/master/staging/persistent-volume-provisioning/glusterfs/glusterfs-secret.yaml) 中找到。 <!–

* clusterid: 630372ccdc720a92c681fb928f27b53f is the ID of the cluster which will be used by Heketi when provisioning the volume. It can also be a list of clusterids, for example: "8452344e2becec931ece4e33c4674e4e,42982310de6c63381718ccfa6d8cf397". This is an optional parameter.
* gidMin, gidMax : The minimum and maximum value of GID range for the storage class. A unique value (GID) in this range ( gidMin-gidMax ) will be used for dynamically provisioned volumes. These are optional values. If not specified, the volume will be provisioned with a value between 2000-2147483647 which are defaults for gidMin and gidMax respectively. –>
* clusterid：630372ccdc720a92c681fb928f27b53f 是集群的 ID，当分配卷时，Heketi 将会使用这个文件。它也可以是一个 clusterid 列表，例如： "8452344e2becec931ece4e33c4674e4e,42982310de6c63381718ccfa6d8cf397"。这个是可选参数。
* gidMin，gidMax：storage class GID 范围的最小值和最大值。在此范围（gidMin-gidMax）内的唯一值（GID）将用于动态分配卷。这些是可选的值。如果不指定，卷将被分配一个 2000-2147483647 之间的值，这是 gidMin 和 gidMax 的默认值。 <!–
* volumetype : The volume type and its parameters can be configured with this optional value. If the volume type is not mentioned, it’s up to the provisioner to decide the volume type.

For example:

* + Replica volume: volumetype: replicate:3 where ‘3’ is replica count.
  + Disperse/EC volume: volumetype: disperse:4:2 where ‘4’ is data and ‘2’ is the redundancy count.
  + Distribute volume: volumetype: none

For available volume types and administration options, refer to the [Administration Guide](https://access.redhat.com/documentation/en-US/Red_Hat_Storage/3.1/html/Administration_Guide/part-Overview.html).

For further reference information, see [How to configure Heketi](https://github.com/heketi/heketi/wiki/Setting-up-the-topology).

When persistent volumes are dynamically provisioned, the Gluster plugin automatically creates an endpoint and a headless service in the name gluster-dynamic-<claimname>. The dynamic endpoint and service are automatically deleted when the persistent volume claim is deleted. –>

* volumetype：卷的类型及其参数可以用这个可选值进行配置。如果未声明卷类型，则由分配器决定卷的类型。

例如： ‘Replica volume’: volumetype: replicate:3 其中 ‘3’ 是 replica 数量. ‘Disperse/EC volume’: volumetype: disperse:4:2 其中 ‘4’ 是数据，’2’ 是冗余数量. ‘Distribute volume’: volumetype: none

有关可用的卷类型和管理选项，请参阅 [管理指南](https://access.redhat.com/documentation/en-US/Red_Hat_Storage/3.1/html/Administration_Guide/part-Overview.html)。

更多相关的参考信息，请参阅 [如何配置 Heketi](https://github.com/heketi/heketi/wiki/Setting-up-the-topology)。

当动态分配持久卷时，Gluster 插件自动创建名为 gluster-dynamic-<claimname> 的端点和 headless service。在 PVC 被删除时动态端点和 headless service 会自动被删除。

#### OpenStack Cinder

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: gold

provisioner: kubernetes.io/cinder

parameters:

availability: nova

* availability：可用区域。如果没有指定，通常卷会在 Kubernetes 集群节点所在的活动区域中轮询调度分配。

**注意：**

**FEATURE STATE:** Kubernetes 1.11 [废弃](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

OpenStack 的内部驱动程序已经被弃用。请使用 [OpenStack 的外部驱动程序](https://github.com/kubernetes/cloud-provider-openstack)。

#### vSphere

1. 使用用户指定的磁盘格式创建一个 StorageClass。
2. apiVersion: storage.k8s.io/v1
3. kind: StorageClass
4. metadata:
5. name: fast
6. provisioner: kubernetes.io/vsphere-volume
7. parameters:

diskformat: zeroedthick

`diskformat`: `thin`, `zeroedthick` 和 `eagerzeroedthick`。默认值: `"thin"`。

1. 在用户指定的数据存储上创建磁盘格式的 StorageClass。
2. apiVersion: storage.k8s.io/v1
3. kind: StorageClass
4. metadata:
5. name: fast
6. provisioner: kubernetes.io/vsphere-volume
7. parameters:
8. diskformat: zeroedthick

datastore: VSANDatastore

`datastore`：用户也可以在 StorageClass 中指定数据存储。卷将在 storage class 中指定的数据存储上创建，在这种情况下是 `VSANDatastore`。该字段是可选的。如果未指定数据存储，则将在用于初始化 vSphere Cloud Provider 的 vSphere 配置文件中指定的数据存储上创建该卷。

1. Kubernetes 中的存储策略管理

\* 使用现有的 vCenter SPBM 策略

vSphere 用于存储管理的最重要特性之一是基于策略的管理。基于存储策略的管理（SPBM）是一个存储策略框架，提供单一的统一控制平面的跨越广泛的数据服务和存储解决方案。 SPBM 使能 vSphere 管理员克服先期的存储配置挑战，如容量规划，差异化服务等级和管理容量空间。

SPBM 策略可以在 StorageClass 中使用 `storagePolicyName` 参数声明。

\* Kubernetes 内的 Virtual SAN 策略支持

Vsphere Infrastructure（VI）管理员将能够在动态卷配置期间指定自定义 Virtual SAN 存储功能。您现在可以定义存储需求，例如性能和可用性，当动态卷供分配时会以存储功能的形式提供。存储功能需求会转换为 Virtual SAN 策略，然后当 persistent volume（虚拟磁盘）在创建时，会将其推送到 Virtual SAN 层。虚拟磁盘分布在 Virtual SAN 数据存储中以满足要求。

更多有关 persistent volume 管理的存储策略的详细信息，

您可以参考 [基于存储策略的动态分配卷管理](https://vmware.github.io/vsphere-storage-for-kubernetes/documentation/policy-based-mgmt.html)。

有几个 [vSphere 例子](https://github.com/kubernetes/examples/tree/master/staging/volumes/vsphere) 供您在 Kubernetes for vSphere 中尝试进行 persistent volume 管理。

#### Ceph RBD

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/rbd

parameters:

monitors: 10.16.153.105:6789

adminId: kube

adminSecretName: ceph-secret

adminSecretNamespace: kube-system

pool: kube

userId: kube

userSecretName: ceph-secret-user

userSecretNamespace: default

fsType: ext4

imageFormat: "2"

imageFeatures: "layering"

* monitors：Ceph monitor，逗号分隔。该参数是必需的。
* adminId：Ceph 客户端 ID，用于在池 ceph 池中创建映像。默认是 “admin”。
* adminSecret：adminId 的 Secret 名称。该参数是必需的。 提供的 secret 必须有值为 “kubernetes.io/rbd” 的 type 参数。
* adminSecretNamespace：adminSecret 的命名空间。默认是 “default”。
* pool: Ceph RBD 池. 默认是 “rbd”。
* userId：Ceph 客户端 ID，用于映射 RBD 镜像。默认与 adminId 相同。 <!–
* userSecretName: The name of Ceph Secret for userId to map RBD image. It must exist in the same namespace as PVCs. This parameter is required. The provided secret must have type “kubernetes.io/rbd”, e.g. created in this way:
* kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" **\**
* --from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ==' **\**

--namespace=kube-system

–> \* userSecretName：用于映射 RBD 镜像的 userId 的 Ceph Secret 的名字。 它必须与 PVC 存在于相同的 namespace 中。该参数是必需的。 提供的 secret 必须具有值为 “kubernetes.io/rbd” 的 type 参数，例如以这样的方式创建：

```shell

kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \

--from-literal=key='QVFEQ1pMdFhPUnQrSmhBQUFYaERWNHJsZ3BsMmNjcDR6RFZST0E9PQ==' \

--namespace=kube-system

```

* userSecretNamespace：userSecretName 的命名空间。
* fsType：Kubernetes 支持的 fsType。默认："ext4"。
* imageFormat：Ceph RBD 镜像格式，”1” 或者 “2”。默认值是 “1”。
* imageFeatures：这个参数是可选的，只能在你将 imageFormat 设置为 “2” 才使用。 目前支持的功能只是 layering。默认是 ““，没有功能打开。

#### Quobyte

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: slow

provisioner: kubernetes.io/quobyte

parameters:

quobyteAPIServer: "http://138.68.74.142:7860"

registry: "138.68.74.142:7861"

adminSecretName: "quobyte-admin-secret"

adminSecretNamespace: "kube-system"

user: "root"

group: "root"

quobyteConfig: "BASE"

quobyteTenant: "DEFAULT"

* quobyteAPIServer：Quobyte API 服务器的格式是 "http(s)://api-server:7860"
* registry：用于挂载卷的 Quobyte registry。你可以指定 registry 为 <host>:<port> 或者如果你想指定多个 registry，你只需要在他们之间添加逗号，例如 <host1>:<port>,<host2>:<port>,<host3>:<port>。 主机可以是一个 IP 地址，或者如果您有正在运行的 DNS，您也可以提供 DNS 名称。
* adminSecretNamespace：adminSecretName的 namespace。 默认值是 “default”。
* adminSecretName：保存关于 Quobyte 用户和密码的 secret，用于对 API 服务器进行身份验证。 提供的 secret 必须有值为 “kubernetes.io/quobyte” 的 type 参数 和 user 与 password 的键值， 例如以这种方式创建：
* kubectl create secret generic quobyte-admin-secret **\**
* --type="kubernetes.io/quobyte" --from-literal=key='opensesame' **\**

--namespace=kube-system

* user：对这个用户映射的所有访问权限。默认是 “root”。
* group：对这个组映射的所有访问权限。默认是 “nfsnobody”。
* quobyteConfig：使用指定的配置来创建卷。您可以创建一个新的配置，或者，可以修改 Web console 或 quobyte CLI 中现有的配置。默认是 “BASE”。
* quobyteTenant：使用指定的租户 ID 创建/删除卷。这个 Quobyte 租户必须已经于 Quobyte。 默认是 “DEFAULT”。

### Azure 磁盘

#### Azure Unmanaged Disk Storage Class（非托管磁盘存储类）

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: slow

provisioner: kubernetes.io/azure-disk

parameters:

skuName: Standard\_LRS

location: eastus

storageAccount: azure\_storage\_account\_name

* skuName：Azure 存储帐户 Sku 层。默认为空。
* location：Azure 存储帐户位置。默认为空。
* storageAccount：Azure 存储帐户名称。如果提供存储帐户，它必须位于与集群相同的资源组中，并且 location 是被忽略的。如果未提供存储帐户，则会在与群集相同的资源组中创建新的存储帐户。

#### 新的 Azure 磁盘 Storage Class（从 v1.7.2 开始）

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: slow

provisioner: kubernetes.io/azure-disk

parameters:

storageaccounttype: Standard\_LRS

kind: Shared

* storageaccounttype：Azure 存储帐户 Sku 层。默认为空。
* kind：可能的值是 shared（默认）、dedicated 和 managed。 当 kind 的值是 shared 时，所有非托管磁盘都在集群的同一个资源组中的几个共享存储帐户中创建。 当 kind 的值是 dedicated 时，将为在集群的同一个资源组中新的非托管磁盘创建新的专用存储帐户。
* Premium VM 可以同时添加 Standard\_LRS 和 Premium\_LRS 磁盘，而 Standard 虚拟机只能添加 Standard\_LRS 磁盘。
* 托管虚拟机只能连接托管磁盘，非托管虚拟机只能连接非托管磁盘。

### Azure 文件

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: azurefile

provisioner: kubernetes.io/azure-file

parameters:

skuName: Standard\_LRS

location: eastus

storageAccount: azure\_storage\_account\_name

* skuName：Azure 存储帐户 Sku 层。默认为空。
* location：Azure 存储帐户位置。默认为空。
* storageAccount：Azure 存储帐户名称。默认为空。 如果不提供存储帐户，会搜索所有与资源相关的存储帐户，以找到一个匹配 skuName 和 location 的账号。 如果提供存储帐户，它必须存在于与集群相同的资源组中，skuName 和 location 会被忽略。
* secretNamespace：包含 Azure 存储帐户名称和密钥的密钥的名称空间。 默认值与 Pod 相同。
* secretName：包含 Azure 存储帐户名称和密钥的密钥的名称。 默认值为 azure-storage-account-<accountName>-secret
* readOnly：指示是否将存储安装为只读的标志。默认为 false，表示 读/写 挂载。 该设置也会影响VolumeMounts中的 ReadOnly 设置。

在存储分配期间，为挂载凭证创建一个名为 secretName 的 secret。如果集群同时启用了 [RBAC](https://kubernetes.io/docs/admin/authorization/rbac/) 和 [Controller Roles](https://kubernetes.io/docs/admin/authorization/rbac/#controller-roles)， 为 system:controller:persistent-volume-binder 的 clusterrole 添加 secret 资源的 create 权限。

在多租户上下文中，强烈建议显式设置 secretNamespace 的值，否则其他用户可能会读取存储帐户凭据。

### Portworx 卷

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: portworx-io-priority-high

provisioner: kubernetes.io/portworx-volume

parameters:

repl: "1"

snap\_interval: "70"

io\_priority: "high"

* fs：选择的文件系统：none/xfs/ext4（默认：ext4）。
* block\_size：以 Kbytes 为单位的块大小（默认值：32）。
* repl：同步副本数量，以复制因子 1..3（默认值：1）的形式提供。 这里需要填写字符串，即，"1" 而不是 1。
* io\_priority：决定是否从更高性能或者较低优先级存储创建卷 high/medium/low（默认值：low）。
* snap\_interval：触发快照的时钟/时间间隔（分钟）。快照是基于与先前快照的增量变化，0 是禁用快照（默认：0）。 这里需要填写字符串，即，是 "70" 而不是 70。
* aggregation\_level：指定卷分配到的块数量，0 表示一个非聚合卷（默认：0）。 这里需要填写字符串，即，是 "0" 而不是 0。
* ephemeral：指定卷在卸载后进行清理还是持久化。 emptyDir 的使用场景可以将这个值设置为 true ， persistent volumes 的使用场景可以将这个值设置为 false（例如 Cassandra 这样的数据库）true/false（默认为 false）。这里需要填写字符串，即，是 "true" 而不是 true。

### ScaleIO

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: slow

provisioner: kubernetes.io/scaleio

parameters:

gateway: https://192.168.99.200:443/api

system: scaleio

protectionDomain: pd0

storagePool: sp1

storageMode: ThinProvisioned

secretRef: sio-secret

readOnly: **false**

fsType: xfs

* provisioner：属性设置为 kubernetes.io/scaleio
* gateway 到 ScaleIO API 网关的地址（必需）
* system：ScaleIO 系统的名称（必需）
* protectionDomain：ScaleIO 保护域的名称（必需）
* storagePool：卷存储池的名称（必需）
* storageMode：存储提供模式：ThinProvisioned（默认）或 ThickProvisioned
* secretRef：对已配置的 Secret 对象的引用（必需）
* readOnly：指定挂载卷的访问模式（默认为 false）
* fsType：卷的文件系统（默认是 ext4）

ScaleIO Kubernetes 卷插件需要配置一个 Secret 对象。 secret 必须用 kubernetes.io/scaleio 类型创建，并与引用它的 PVC 所属的名称空间使用相同的值 如下面的命令所示：

kubectl create secret generic sio-secret --type="kubernetes.io/scaleio" **\**

--from-literal=username=sioadmin --from-literal=password=d2NABDNjMA== **\**

--namespace=default

### StorageOS

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/storageos

parameters:

pool: default

description: Kubernetes volume

fsType: ext4

adminSecretNamespace: default

adminSecretName: storageos-secret

* pool：分配卷的 StorageOS 分布式容量池的名称。如果未指定，则使用通常存在的 default 池。
* description：分配给动态创建的卷的描述。所有卷描述对于 storage class 都是相同的， 但不同的 storage class 可以使用不同的描述，以区分不同的使用场景。 默认为 Kubernetas volume。
* fsType：请求的默认文件系统类型。请注意，在 StorageOS 中用户定义的规则可以覆盖此值。默认为 ext4
* adminSecretNamespace：API 配置 secret 所在的命名空间。如果设置了 adminSecretName，则是必需的。
* adminSecretName：用于获取 StorageOS API 凭证的 secret 名称。如果未指定，则将尝试默认值。

StorageOS Kubernetes 卷插件可以使 Secret 对象来指定用于访问 StorageOS API 的端点和凭据。 只有当默认值已被更改时，这才是必须的。 secret 必须使用 kubernetes.io/storageos 类型创建，如以下命令：

kubectl create secret generic storageos-secret **\**

--type="kubernetes.io/storageos" **\**

--from-literal=apiAddress=tcp://localhost:5705 **\**

--from-literal=apiUsername=storageos **\**

--from-literal=apiPassword=storageos **\**

--namespace=default

用于动态分配卷的 Secret 可以在任何名称空间中创建，并通过 adminSecretNamespace 参数引用。 预先配置的卷使用的 Secret 必须在与引用它的 PVC 在相同的名称空间中。

### 本地

**FEATURE STATE:** Kubernetes v1.14 [稳定](https://kubernetes.io/zh/docs/concepts/storage/storage-classes/)

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: local-storage

provisioner: kubernetes.io/no-provisioner

volumeBindingMode: WaitForFirstConsumer

本地卷还不支持动态分配，然而还是需要创建 StorageClass 以延迟卷绑定，直到完成 pod 的调度。这是由 WaitForFirstConsumer 卷绑定模式指定的。

延迟卷绑定使得调度器在为 PersistentVolumeClaim 选择一个合适的 PersistentVolume 时能考虑到所有 pod 的调度限制。

#### 反馈

## 卷快照类

本文档描述了 Kubernetes 中 VolumeSnapshotClass 的概念。 建议熟悉[卷快照（Volume Snapshots）](https://kubernetes.io/docs/concepts/storage/volume-snapshots/)和[存储类（Storage Class）](https://kubernetes.io/docs/concepts/storage/storage-classes)。

* [**介绍**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshot-classes/#%e4%bb%8b%e7%bb%8d)
* [**VolumeSnapshotClass 资源**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshot-classes/#volumesnapshotclass-%e8%b5%84%e6%ba%90)
* [**参数**](https://kubernetes.io/zh/docs/concepts/storage/volume-snapshot-classes/#%e5%8f%82%e6%95%b0)

### 介绍

就像 StorageClass 为管理员提供了一种在配置卷时描述存储“类”的方法，VolumeSnapshotClass 提供了一种在配置卷快照时描述存储“类”的方法。

### VolumeSnapshotClass 资源

每个 VolumeSnapshotClass 都包含 snapshotter 和 parameters 字段，当需要动态配置属于该类的 VolumeSnapshot 时使用。

VolumeSnapshotClass 对象的名称很重要，是用户可以请求特定类的方式。 管理员在首次创建 VolumeSnapshotClass 对象时设置类的名称和其他参数，对象一旦创建就无法更新。

管理员可以为不请求任何特定类绑定的 VolumeSnapshots 指定默认的 VolumeSnapshotClass。

apiVersion: snapshot.storage.k8s.io/v1alpha1

kind: VolumeSnapshotClass

metadata:

name: csi-hostpath-snapclass

snapshotter: csi-hostpath

parameters:

#### 快照生成器（Snapshotter）

卷快照类具有一个快照生成器，用于确定配置 VolumeSnapshot 的 CSI 卷插件。 必须指定此字段。

### 参数

卷快照类具有描述属于卷快照类的卷快照参数。 可根据 snapshotter 接受不同的参数。

#### 反馈

## 动态卷供应

动态卷供应允许按需创建存储卷。 如果没有动态供应，集群管理员必须手动地联系他们的云或存储提供商来创建新的存储卷， 然后在 Kubernetes 集群创建 [PersistentVolume 对象](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)来表示这些卷。 动态供应功能消除了集群管理员预先配置存储的需要。 相反，它在用户请求时自动供应存储。

* [**背景**](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#%e8%83%8c%e6%99%af)
* [**启用动态卷供应**](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#%e5%90%af%e7%94%a8%e5%8a%a8%e6%80%81%e5%8d%b7%e4%be%9b%e5%ba%94)
* [**使用动态卷供应**](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#%e4%bd%bf%e7%94%a8%e5%8a%a8%e6%80%81%e5%8d%b7%e4%be%9b%e5%ba%94)
* [**默认行为**](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#%e9%bb%98%e8%ae%a4%e8%a1%8c%e4%b8%ba)
* [**拓扑感知**](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#%e6%8b%93%e6%89%91%e6%84%9f%e7%9f%a5)

### 背景

动态卷供应的实现基于 storage.k8s.io API 组中的 StorageClass API 对象。 集群管理员可以根据需要定义多个 StorageClass 对象，每个对象指定一个\*卷插件\*（又名 \*provisioner\*）， 卷插件向卷供应商提供在创建卷时需要的数据卷信息及相关参数。

集群管理员可以在集群中定义和公开多种存储（来自相同或不同的存储系统），每种都具有自定义参数集。 该设计也确保终端用户不必担心存储供应的复杂性和细微差别，但仍然能够从多个存储选项中进行选择。

点击[这里](https://kubernetes.io/docs/concepts/storage/storage-classes/)查阅有关存储类的更多信息。

### 启用动态卷供应

要启用动态供应功能，集群管理员需要为用户预先创建一个或多个 StorageClass 对象。 StorageClass 对象定义在进行动态卷供应时应使用哪个卷供应商，以及应该将哪些参数传递给该供应商。 以下清单创建了一个存储类 “slow”，它提供类似标准磁盘的永久磁盘。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: slow

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-standard

以下清单创建了一个 “fast” 存储类，它提供类似 SSD 的永久磁盘。

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-ssd

### 使用动态卷供应

用户通过在 PersistentVolumeClaim 中包含存储类来请求动态供应的存储。 在 Kubernetes v1.6 之前，这通过 volume.beta.kubernetes.io/storage-class 注解实现。然而，这个注解自 v1.6 起就不被推荐使用了。 用户现在能够而且应该使用 PersistentVolumeClaim 对象的 storageClassName 字段。 这个字段的值必须能够匹配到集群管理员配置的 StorageClass 名称（见[下面](https://kubernetes.io/zh/docs/concepts/storage/dynamic-provisioning/#enabling-dynamic-provisioning)）。

例如，要选择 “fast” 存储类，用户将创建如下的 PersistentVolumeClaim：

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: claim1

spec:

accessModes:

- ReadWriteOnce

storageClassName: fast

resources:

requests:

storage: 30Gi

该声明会自动供应一块类似 SSD 的永久磁盘。 在删除该声明后，这个卷也会被销毁。

### 默认行为

可以在群集上启用动态卷供应，以便在未指定存储类的情况下动态设置所有声明。 集群管理员可以通过以下方式启用此行为：

* 标记一个 StorageClass 为 \*默认\*；
* 确保 [DefaultStorageClass 准入控制器](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#defaultstorageclass)在 API 服务端被启用。

管理员可以通过向其添加 storageclass.kubernetes.io/is-default-class 注解来将特定的 StorageClass 标记为默认。 当集群中存在默认的 StorageClass 并且用户创建了一个未指定 storageClassName 的 PersistentVolumeClaim 时， DefaultStorageClass 准入控制器会自动向其中添加指向默认存储类的 storageClassName 字段。

请注意，群集上最多只能有一个 默认 存储类，否则无法创建没有明确指定 storageClassName 的 PersistentVolumeClaim。

### 拓扑感知

在[多区域](https://kubernetes.io/docs/setup/multiple-zones)集群中，Pod 可以被分散到多个区域。 单区域存储后端应该被供应到 Pod 被调度到的区域。 这可以通过设置[卷绑定模式](https://kubernetes.io/docs/concepts/storage/storage-classes/#volume-binding-mode)来实现。

### 反馈

## 特定于节点的卷数限制

此页面描述了各个云供应商可关联至一个节点的最大卷数。

谷歌、亚马逊和微软等云供应商通常对可以关联到节点的卷数量进行限制。 Kubernetes 需要尊重这些限制。 否则，在节点上调度的 Pod 可能会卡住去等待卷的关联。

* [**Kubernetes 的默认限制**](https://kubernetes.io/zh/docs/concepts/storage/storage-limits/#kubernetes-%e7%9a%84%e9%bb%98%e8%ae%a4%e9%99%90%e5%88%b6)
* [**自定义限制**](https://kubernetes.io/zh/docs/concepts/storage/storage-limits/#%e8%87%aa%e5%ae%9a%e4%b9%89%e9%99%90%e5%88%b6)
* [**动态卷限制**](https://kubernetes.io/zh/docs/concepts/storage/storage-limits/#%e5%8a%a8%e6%80%81%e5%8d%b7%e9%99%90%e5%88%b6)

### Kubernetes 的默认限制

The Kubernetes 调度器对关联于一个节点的卷数有默认限制：

|  |  |
| --- | --- |
| 云服务 | 每节点最大卷数 |
| [Amazon Elastic Block Store (EBS)](https://aws.amazon.com/ebs/) | 39 |
| [Google Persistent Disk](https://cloud.google.com/persistent-disk/) | 16 |
| [Microsoft Azure Disk Storage](https://azure.microsoft.com/en-us/services/storage/main-disks/) | 16 |

### 自定义限制

您可以通过设置 KUBE\_MAX\_PD\_VOLS 环境变量的值来设置这些限制，然后再启动调度器。 CSI 驱动程序可能具有不同的过程，关于如何自定义其限制请参阅相关文档。

如果设置的限制高于默认限制，请谨慎使用。请参阅云提供商的文档以确保节点可支持您设置的限制。

此限制应用于整个集群，所以它会影响所有节点。

### 动态卷限制

**FEATURE STATE:** Kubernetes v1.17 [稳定](https://kubernetes.io/zh/docs/concepts/storage/storage-limits/)

以下卷类型支持动态卷限制。

* Amazon EBS
* Google Persistent Disk
* Azure Disk
* CSI

对于由内建插件管理的卷，Kubernetes 会自动确定节点类型并确保节点上可关联的卷数目合规。 例如：

* 在 [Google Compute Engine](https://cloud.google.com/compute/)环境中, [根据节点类型](https://cloud.google.com/compute/docs/disks/#pdnumberlimits)最多可以将128个卷关联到节点。
* 对于 M5、C5、R5、T3 和 Z1D 类型实例的 Amazon EBS 磁盘，Kubernetes 仅允许 25 个卷关联到节点。 对于 ec2 上的其他实例类型 [Amazon Elastic Compute Cloud (EC2)](https://aws.amazon.com/ec2/), Kubernetes 允许 39 个卷关联至节点。
* 在 Azure 环境中, 根据节点类型，最多 64 个磁盘可以关联至一个节点。 更多详细信息，请参阅[Azure 虚拟机的数量大小](https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes)。
* 如果 CSI 存储驱动程序（使用 NodeGetInfo ）为节点通告卷数上限，则 [kube-scheduler](https://kubernetes.io/docs/reference/generated/kube-scheduler/) 将遵守该限制值。 参考 [CSI 规范](https://github.com/container-storage-interface/spec/blob/master/spec.md#nodegetinfo) 获取更多详细信息。
* 对于由已迁移到 CSI 驱动程序的树内插件管理的卷，最大卷数将是 CSI 驱动程序报告的卷数。

# 配置

## 扩展资源的资源箱打包

**FEATURE STATE:** Kubernetes 1.16 [alpha](https://kubernetes.io/zh/docs/concepts/configuration/)

可以将 kube-scheduler 配置为使用 RequestedToCapacityRatioResourceAllocation 优先级函数启用资源箱打包以及扩展资源。 优先级函数可用于根据自定义需求微调 kube-scheduler 。

* [**使用 RequestedToCapacityRatioResourceAllocation 启用装箱**](https://kubernetes.io/zh/docs/concepts/configuration/#%e4%bd%bf%e7%94%a8-requestedtocapacityratioresourceallocation-%e5%90%af%e7%94%a8%e8%a3%85%e7%ae%b1)

### 使用 RequestedToCapacityRatioResourceAllocation 启用装箱

在 Kubernetes 1.15 之前，Kube-scheduler 用于允许根据主要资源，如 CPU 和内存对容量之比的请求对节点进行评分。 Kubernetes 1.16 在优先级函数中添加了一个新参数，该参数允许用户指定资源以及每个资源的权重，以便根据容量之比的请求为节点评分。 这允许用户通过使用适当的参数来打包扩展资源，从而提高了大型集群中稀缺资源的利用率。 RequestedToCapacityRatioResourceAllocation 优先级函数的行为可以通过名为 requestedToCapacityRatioArguments 的配置选项进行控制。 这个论证由两个参数 shape 和 resources 组成。 Shape 允许用户根据 utilization 和 score 值将功能调整为要求最少或要求最高的功能。 资源由 name 和 weight 组成，name 指定评分时要考虑的资源，weight 指定每种资源的权重。

以下是一个配置示例，该配置将 requestedToCapacityRatioArguments 设置为扩展资源 intel.com/foo 和 intel.com/bar 的装箱行为

{

**"kind"** : "Policy",

**"apiVersion"** : "v1",

...

**"priorities"** : [

...

{

**"name"**: "RequestedToCapacityRatioPriority",

**"weight"**: 2,

**"argument"**: {

**"requestedToCapacityRatioArguments"**: {

**"shape"**: [

{**"utilization"**: 0, **"score"**: 0},

{**"utilization"**: 100, **"score"**: 10}

],

**"resources"**: [

{**"name"**: "intel.com/foo", **"weight"**: 3},

{**"name"**: "intel.com/bar", **"weight"**: 5}

]

}

}

}

],

}

**默认情况下禁用此功能**

#### 调整 RequestedToCapacityRatioResourceAllocation 优先级函数

shape 用于指定 RequestedToCapacityRatioPriority 函数的行为。

{"utilization": 0, "score": 0},

{"utilization": 100, "score": 10}

上面的参数在利用率为 0% 时给节点评分为0，在利用率为 100% 时给节点评分为10，因此启用了装箱行为。 要启用最少请求，必须按如下方式反转得分值。

{"utilization": 0, "score": 100},

{"utilization": 100, "score": 0}

resources 是一个可选参数，默认情况下设置为：

"resources": [

{"name": "CPU", "weight": 1},

{"name": "Memory", "weight": 1}

]

它可以用来添加扩展资源，如下所示：

"resources": [

{"name": "intel.com/foo", "weight": 5},

{"name": "CPU", "weight": 3},

{"name": "Memory", "weight": 1}

]

weight 参数是可选的，如果未指定，则设置为1。 同样， weight 不能设置为负值。

#### RequestedToCapacityRatioResourceAllocation 优先级函数如何对节点评分

本部分适用于希望了解此功能的内部细节的人员。 以下是如何针对给定的一组值计算节点得分的示例。

Requested Resources

intel.com/foo : 2

Memory: 256MB

CPU: 2

Resource Weights

intel.com/foo : 5

Memory: 1

CPU: 3

FunctionShapePoint {{0, 0}, {100, 10}}

Node 1 Spec

Available:

intel.com/foo : 4

Memory : 1 GB

CPU: 8

Used:

intel.com/foo: 1

Memory: 256MB

CPU: 1

Node Score:

intel.com/foo = resourceScoringFunction((2+1),4)

= (100 - ((4-3)\*100/4)

= (100 - 25)

= 75

= rawScoringFunction(75)

= 7

Memory = resourceScoringFunction((256+256),1024)

= (100 -((1024-512)\*100/1024))

= 50

= rawScoringFunction(50)

= 5

CPU = resourceScoringFunction((2+1),8)

= (100 -((8-3)\*100/8))

= 37.5

= rawScoringFunction(37.5)

= 3

NodeScore = (7 \* 5) + (5 \* 1) + (3 \* 3) / (5 + 1 + 3)

= 5

Node 2 Spec

Available:

intel.com/foo: 8

Memory: 1GB

CPU: 8

Used:

intel.com/foo: 2

Memory: 512MB

CPU: 6

Node Score:

intel.com/foo = resourceScoringFunction((2+2),8)

= (100 - ((8-4)\*100/8)

= (100 - 25)

= 50

= rawScoringFunction(50)

= 5

Memory = resourceScoringFunction((256+512),1024)

= (100 -((1024-768)\*100/1024))

= 75

= rawScoringFunction(75)

= 7

CPU = resourceScoringFunction((2+6),8)

= (100 -((8-8)\*100/8))

= 100

= rawScoringFunction(100)

= 10

NodeScore = (5 \* 5) + (7 \* 1) + (10 \* 3) / (5 + 1 + 3)

= 7

#### 反馈

## 配置最佳实践

本文档重点介绍并整合了整个用户指南、入门文档和示例中介绍的配置最佳实践。

这是一份活文件。 如果您认为某些内容不在此列表中但可能对其他人有用，请不要犹豫，提交问题或提交 PR。

* [**一般配置提示**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#%e4%b8%80%e8%88%ac%e9%85%8d%e7%bd%ae%e6%8f%90%e7%a4%ba)
* [**“Naked”Pods 与 ReplicaSet，Deployment 和 Jobs**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#naked-pods-%e4%b8%8e-replicaset-deployment-%e5%92%8c-jobs)
* [**服务**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#%e6%9c%8d%e5%8a%a1)
* [**使用标签**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#%e4%bd%bf%e7%94%a8%e6%a0%87%e7%ad%be)
* [**容器镜像**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#%e5%ae%b9%e5%99%a8%e9%95%9c%e5%83%8f)
* [**使用 kubectl**](https://kubernetes.io/zh/docs/concepts/configuration/overview/#%e4%bd%bf%e7%94%a8-kubectl)

### 一般配置提示

* 定义配置时，请指定最新的稳定 API 版本。
* 在推送到集群之前，配置文件应存储在版本控制中。 这允许您在必要时快速回滚配置更改。 它还有助于集群重新创建和恢复。
* 使用 YAML 而不是 JSON 编写配置文件。虽然这些格式几乎可以在所有场景中互换使用，但 YAML 往往更加用户友好。
* 只要有意义，就将相关对象分组到一个文件中。 一个文件通常比几个文件更容易管理。 请参阅[guestbook-all-in-one.yaml](https://github.com/kubernetes/examples/tree/master/guestbook/all-in-one/guestbook-all-in-one.yaml) 文件作为此语法的示例。
* 另请注意，可以在目录上调用许多kubectl命令。 例如，你可以在配置文件的目录中调用kubectl apply。
* 不要不必要地指定默认值：简单的最小配置会降低错误的可能性。
* 将对象描述放在注释中，以便更好地进行内省。

### “Naked”Pods 与 ReplicaSet，Deployment 和 Jobs

* 如果您能避免，不要使用 naked Pods（即，Pod 未绑定到[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 或[Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)）。 如果节点发生故障，将不会重新安排 Naked Pods。

部署，它创建一个 ReplicaSet 以确保所需数量的 Pod 始终可用，并指定替换 Pod 的策略(例如 [RollingUpdate](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment))，除了一些显式的[restartPolicy: Never](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy)场景之外，几乎总是优先考虑直接创建 Pod。 [Job](https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/) 也可能是合适的。

### 服务

* 在其相应的后端工作负载（Deployment 或 ReplicaSet）之前，以及在需要访问它的任何工作负载之前创建[服务](https://kubernetes.io/docs/concepts/services-networking/service/)。 当 Kubernetes 启动容器时，它提供指向启动容器时正在运行的所有服务的环境变量。 例如，如果存在名为foo当服务，则所有容器将在其初始环境中获取以下变量。
* FOO\_SERVICE\_HOST=<the host the Service is running on>

FOO\_SERVICE\_PORT=<the port the Service is running on>

这确实意味着订购要求 - 必须在Pod本身之前创建Pod想要访问的任何Service，否则将不会填充环境变量。 DNS没有此限制。

* 一个可选（尽管强烈推荐）[cluster add-on](https://kubernetes.io/docs/concepts/cluster-administration/addons/)是 DNS 服务器。DNS 服务器为新的Services监视 Kubernetes API，并为每个创建一组 DNS 记录。 如果在整个集群中启用了 DNS，则所有Pods应该能够自动对Services进行名称解析。
* 除非绝对必要，否则不要为 Pod 指定hostPort。 将 Pod 绑定到hostPort时，它会限制 Pod 可以调度的位置数，因为每个<hostIP, hostPort, protocol>组合必须是唯一的。如果您没有明确指定hostIP和protocol，Kubernetes将使用0.0.0.0作为默认hostIP和TCP作为默认protocol。

如果您只需要访问端口以进行调试，则可以使用[apiserver proxy](https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/#manually-constructing-apiserver-proxy-urls)或[kubectl port-forward](https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/)。

如果您明确需要在节点上公开 Pod 的端口，请在使用hostPort之前考虑使用[NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) 服务。

* 避免使用hostNetwork，原因与hostPort相同。
* 当您不需要kube-proxy负载平衡时，使用 [无头服务](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) (具有None的ClusterIP)以便于服务发现。

### 使用标签

* 定义并使用[标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)来识别应用程序或部署的**semantic attributes**，例如{ app: myapp, tier: frontend, phase: test, deployment: v3 }。 您可以使用这些标签为其他资源选择合适的 Pod；例如，一个选择所有tier: frontend Pod 的服务，或者app: myapp的所有phase: test组件。 有关此方法的示例，请参阅[留言板](https://github.com/kubernetes/examples/tree/master/guestbook/) 。

通过从选择器中省略特定发行版的标签，可以使服务跨越多个部署。 [部署](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)可以在不停机的情况下轻松更新正在运行的服务。

部署描述了对象的期望状态，并且如果对该规范的更改是\_applied\_，则部署控制器以受控速率将实际状态改变为期望状态。

* 您可以操纵标签进行调试。
* 由于 Kubernetes 控制器（例如 ReplicaSet）和服务使用选择器标签与 Pod 匹配，因此从 Pod 中删除相关标签将阻止其被控制器考虑或由服务提供服务流量。 如果删除现有 Pod 的标签，其控制器将创建一个新的 Pod 来取代它。 这是在”隔离”环境中调试先前”实时”Pod 的有用方法。 要以交互方式删除或添加标签，请使用[kubectl label](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#label)。

### 容器镜像

当 [kubelet](https://kubernetes.io/docs/admin/kubelet/)尝试拉取指定的镜像时，[imagePullPolicy]（/ docs / concepts / containers / images / #updating-images）和镜像的标签会生效。

* imagePullPolicy: IfNotPresent：仅当镜像在本地不存在时镜像才被拉取。
* imagePullPolicy: Always：每次启动 pod 的时候都会拉取镜像。
* 省略imagePullPolicy，镜像标签为:latest或被省略，Always被应用。
* imagePullPolicy被省略，并且镜像的标签被指定且不是:latest，IfNotPresent被应用。
* imagePullPolicy: Never：镜像被假设存在于本地。 没有尝试拉取镜像。

**注意：** 要确保容器始终使用相同版本的镜像，你可以指定其 [摘要](https://docs.docker.com/engine/reference/commandline/pull/#pull-an-image-by-digest-immutable-identifier), 例如sha256:45b23dee08af5e43a7fea6c4cf9c25ccf269ee113168c19722f87876677c5cb2。 摘要唯一地标识出镜像的指定版本，因此除非您更改摘要值，否则 Kubernetes 永远不会更新它。

**注意：** 在生产中部署容器时应避免使用 :latest 标记，因为更难跟踪正在运行的镜像版本，并且更难以正确回滚。

**注意：** 底层镜像提供程序的缓存语义甚至使 imagePullPolicy: Always变得高效。 例如，对于 Docker，如果镜像已经存在，则拉取尝试很快，因为镜像层都被缓存并且不需要镜像下载。

### 使用 kubectl

* 使用kubectl apply -f <directory>。 它在<directory>中的所有.yaml，.yml和.json文件中查找 Kubernetes 配置，并将其传递给apply。
* 使用标签选择器进行get和delete操作，而不是特定的对象名称。
* 请参阅[标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-selectors)和[有效使用标签]部分(/docs/concepts/cluster-administration/manage-deployment/#using-labels-effectively)。
* 使用kubectl run和kubectl expose来快速创建单容器部署和服务。 有关示例，请参阅[使用服务访问集群中的应用程序](https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/)。

### 反馈

## Pod 开销

**FEATURE STATE:** Kubernetes v1.16 [alpha](https://kubernetes.io/zh/docs/concepts/configuration/pod-overhead/)

在节点上运行 Pod 时，Pod 本身占用大量系统资源。这些资源是运行 Pod 内容器所需资源的附加资源。 POD 开销 是一个特性，用于计算 Pod 基础设施在容器请求和限制之上消耗的资源。

* [**Pod 开销**](https://kubernetes.io/zh/docs/concepts/configuration/pod-overhead/#pod-%e5%bc%80%e9%94%80)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/configuration/pod-overhead/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### Pod 开销

在 Kubernetes 中，Pod 的开销是根据与 Pod 的 [RuntimeClass](https://kubernetes.io/docs/concepts/containers/runtime-class/) 相关联的开销在[准入](https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#what-are-admission-webhooks)时设置的。

当启用 Pod 开销时，在调度 Pod 时，除了考虑容器资源请求的总和外，还要考虑 Pod 开销。类似地，Kubelet 将在确定 pod cgroup 的大小和执行 Pod 驱逐排序时包含 Pod 开销。

#### 设置

您需要确保在集群中启用了 PodOverhead [特性门](https://kubernetes.io/zh/docs/concepts/configuration/pod-overhead/%E9%BB%98%E8%AE%A4%E6%83%85%E5%86%B5%E4%B8%8B%E6%98%AF%E5%85%B3%E9%97%AD%E7%9A%84)。这意味着：

* 在 [kube-scheduler](https://kubernetes.io/docs/reference/generated/kube-scheduler/)
* 在 [kube-apiserver](https://kubernetes.io/docs/reference/generated/kube-apiserver/)
* 在每一个 Node 的 [kubelet](https://kubernetes.io/docs/reference/generated/kubelet)
* 在任何使用特性门的自定义api服务器中

**注意：**

能够写入运行时类资源的用户能够对工作负载性能产生集群范围的影响。可以使用 Kubernetes 访问控制来限制对此功能的访问。 有关详细信息，请参见[授权概述](https://kubernetes.io/docs/reference/access-authn-authz/authorization/)。

### 接下来

### 反馈

## 为容器管理计算资源

当您定义 [Pod](https://kubernetes.io/docs/user-guide/pods) 的时候可以选择为每个容器指定需要的 CPU 和内存（RAM）大小。当为容器指定了资源请求后，调度器就能够更好的判断出将容器调度到哪个节点上。如果您还为容器指定了资源限制，Kubernetes 就可以按照指定的方式来处理节点上的资源竞争。关于资源请求和限制的不同点和更多资料请参考 [Resource QoS](https://git.k8s.io/community/contributors/design-proposals/resource-qos.md)。

* [**资源类型**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e8%b5%84%e6%ba%90%e7%b1%bb%e5%9e%8b)
* [**Pod 和 容器的资源请求和限制**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#pod-%e5%92%8c-%e5%ae%b9%e5%99%a8%e7%9a%84%e8%b5%84%e6%ba%90%e8%af%b7%e6%b1%82%e5%92%8c%e9%99%90%e5%88%b6)
* [**CPU 的含义**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#cpu-%e7%9a%84%e5%90%ab%e4%b9%89)
* [**内存的含义**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e5%86%85%e5%ad%98%e7%9a%84%e5%90%ab%e4%b9%89)
* [**具有资源请求的 Pod 如何调度**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e5%85%b7%e6%9c%89%e8%b5%84%e6%ba%90%e8%af%b7%e6%b1%82%e7%9a%84-pod-%e5%a6%82%e4%bd%95%e8%b0%83%e5%ba%a6)
* [**具有资源限制的 Pod 如何运行**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e5%85%b7%e6%9c%89%e8%b5%84%e6%ba%90%e9%99%90%e5%88%b6%e7%9a%84-pod-%e5%a6%82%e4%bd%95%e8%bf%90%e8%a1%8c)
* [**监控计算资源使用**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e7%9b%91%e6%8e%a7%e8%ae%a1%e7%ae%97%e8%b5%84%e6%ba%90%e4%bd%bf%e7%94%a8)
* [**疑难解答**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e7%96%91%e9%9a%be%e8%a7%a3%e7%ad%94)
* [**我的容器被终止了**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e6%88%91%e7%9a%84%e5%ae%b9%e5%99%a8%e8%a2%ab%e7%bb%88%e6%ad%a2%e4%ba%86)
* [**本地临时存储**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e6%9c%ac%e5%9c%b0%e4%b8%b4%e6%97%b6%e5%ad%98%e5%82%a8)
* [**拓展资源**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e6%8b%93%e5%b1%95%e8%b5%84%e6%ba%90)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 资源类型

CPU 和\*内存\*都是\*资源类型\*。资源类型具有基本单位。CPU 的单位是核心数，内存的单位是字节。

CPU和内存统称为\*计算资源\*，也可以称为\*资源\*。计算资源的数量是可以被请求、分配、消耗和可测量的。它们与 [API 资源](https://kubernetes.io/docs/concepts/overview/kubernetes-api/) 不同。 API 资源（如 Pod 和 [Service](https://kubernetes.io/docs/concepts/services-networking/service/)）是可通过 Kubernetes API server 读取和修改的对象。

### Pod 和 容器的资源请求和限制

Pod 中的每个容器都可以指定以下的一个或者多个值：

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

尽管只能在个别容器上指定请求和限制，但是我们可以方便地计算出 Pod 资源请求和限制。特定资源类型的Pod 资源请求/限制是 Pod 中每个容器的该类型的资源请求/限制的总和。

### CPU 的含义

CPU 资源的限制和请求以 cpu 为单位。

Kubernetes 中的一个 cpu 等于：

* 1 AWS vCPU
* 1 GCP Core
* 1 Azure vCore
* 1 Hyperthread 在带有超线程的裸机 Intel 处理器上

允许浮点数请求。具有 spec.containers[].resources.requests.cpu 为 0.5 的容器保证了一半 CPU 要求 1 CPU的一半。表达式 0.1 等价于表达式 100m，可以看作 “100 millicpu”。有些人说成是“一百毫 cpu”，其实说的是同样的事情。具有小数点（如 0.1）的请求由 API 转换为100m，精度不超过 1m。因此，可能会优先选择 100m 的形式。

CPU 总是要用绝对数量，不可以使用相对数量；0.1 的 CPU 在单核、双核、48核的机器中的意义是一样的。

### 内存的含义

内存的限制和请求以字节为单位。您可以使用以下后缀之一作为平均整数或定点整数表示内存：E，P，T，G，M，K。您还可以使用两个字母的等效的幂数：Ei，Pi，Ti ，Gi，Mi，Ki。例如，以下代表大致相同的值：

128974848, 129e6, 129M, 123Mi

下面是个例子。

以下 Pod 有两个容器。每个容器的请求为 0.25 cpu 和 64MiB（226 字节）内存，每个容器的限制为 0.5 cpu 和 128MiB 内存。您可以说该 Pod 请求 0.5 cpu 和 128 MiB 的内存，限制为 1 cpu 和 256MiB 的内存。

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"

### 具有资源请求的 Pod 如何调度

当您创建一个 Pod 时，Kubernetes 调度程序将为 Pod 选择一个节点。每个节点具有每种资源类型的最大容量：可为 Pod 提供的 CPU 和内存量。调度程序确保对于每种资源类型，调度的容器的资源请求的总和小于节点的容量。请注意，尽管节点上的实际内存或 CPU 资源使用量非常低，但如果容量检查失败，则调度程序仍然拒绝在该节点上放置 Pod。当资源使用量稍后增加时，例如在请求率的每日峰值期间，这可以防止节点上的资源短缺。

### 具有资源限制的 Pod 如何运行

当 kubelet 启动一个 Pod 的容器时，它会将 CPU 和内存限制传递到容器运行时。

当使用 Docker 时：

* spec.containers[].resources.requests.cpu 的值将转换成 millicore 值，这是个浮点数，并乘以 1024，这个数字中的较大者或 2 用作 docker run 命令中的[--cpu-shares](https://docs.docker.com/engine/reference/run/#/cpu-share-constraint) 标志的值。
* spec.containers[].resources.limits.cpu 被转换成 millicore 值。被乘以 100000 然后 除以 1000。这个数字用作 docker run 命令中的 [--cpu-quota](https://docs.docker.com/engine/reference/run/#/cpu-quota-constraint) 标志的值。[--cpu-quota ] 标志被设置成了 100000，表示测量配额使用的默认100ms 周期。如果 [--cpu-cfs-quota] 标志设置为 true，则 kubelet 会强制执行 cpu 限制。从 Kubernetes 1.2 版本起，此标志默认为 true。

**注意：** 默认配额限制为 100 毫秒。 CPU配额的最小单位为 1 毫秒。

* spec.containers[].resources.limits.memory 被转换为整型，作为 docker run 命令中的 [--memory](https://docs.docker.com/engine/reference/run/#/user-memory-constraints) 标志的值。

如果容器超过其内存限制，则可能会被终止。如果可重新启动，则与所有其他类型的运行时故障一样，kubelet 将重新启动它。

如果一个容器超过其内存请求，那么当节点内存不足时，它的 Pod 可能被逐出。

容器可能被允许也可能不被允许超过其 CPU 限制时间。但是，由于 CPU 使用率过高，不会被杀死。

要确定容器是否由于资源限制而无法安排或被杀死，请参阅[疑难解答](https://kubernetes.io/zh/docs/concepts/configuration/manage-compute-resources-container/#troubleshooting) 部分。

### 监控计算资源使用

Pod 的资源使用情况被报告为 Pod 状态的一部分。

如果为集群配置了 [可选监控](http://releases.k8s.io/master/cluster/addons/cluster-monitoring/README.md)，则可以从监控系统检索 Pod 资源的使用情况。

### 疑难解答

#### 我的 Pod 处于 pending 状态且事件信息显示 failedScheduling

如果调度器找不到任何该 Pod 可以匹配的节点，则该 Pod 将保持不可调度状态，直到找到一个可以被调度到的位置。每当调度器找不到 Pod 可以调度的地方时，会产生一个事件，如下所示：

kubectl describe pod frontend | grep -A 3 Events

Events:

FirstSeen LastSeen Count From Subobject PathReason Message

36s 5s 6 {scheduler } FailedScheduling Failed for reason PodExceedsFreeCPU and possibly others

在上述示例中，由于节点上的 CPU 资源不足，名为 “frontend” 的 Pod 将无法调度。由于内存不足（PodExceedsFreeMemory），类似的错误消息也可能会导致失败。一般来说，如果有这种类型的消息而处于 pending 状态，您可以尝试如下几件事情：

* 向集群添加更多节点。
* 终止不需要的 Pod，为待处理的 Pod 腾出空间。
* 检查 Pod 所需的资源是否大于所有节点的资源。 例如，如果全部节点的容量为cpu：1，那么一个请求为 cpu：1.1的 Pod 永远不会被调度。

您可以使用 kubectl describe nodes 命令检查节点容量和分配的数量。 例如：

kubectl describe nodes e2e-test-node-pool-4lw4

Name: e2e-test-node-pool-4lw4

[ ... lines removed for clarity ...]

Capacity:

cpu: 2

memory: 7679792Ki

pods: 110

Allocatable:

cpu: 1800m

memory: 7474992Ki

pods: 110

[ ... lines removed for clarity ...]

Non-terminated Pods: (5 in total)

Namespace Name CPU Requests CPU Limits Memory Requests Memory Limits

--------- ---- ------------ ---------- --------------- -------------

kube-system fluentd-gcp-v1.38-28bv1 100m (5%) 0 (0%) 200Mi (2%) 200Mi (2%)

kube-system kube-dns-3297075139-61lj3 260m (13%) 0 (0%) 100Mi (1%) 170Mi (2%)

kube-system kube-proxy-e2e-test-... 100m (5%) 0 (0%) 0 (0%) 0 (0%)

kube-system monitoring-influxdb-grafana-v4-z1m12 200m (10%) 200m (10%) 600Mi (8%) 600Mi (8%)

kube-system node-problem-detector-v0.1-fj7m3 20m (1%) 200m (10%) 20Mi (0%) 100Mi (1%)

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

CPU Requests CPU Limits Memory Requests Memory Limits

------------ ---------- --------------- -------------

680m (34%) 400m (20%) 920Mi (12%) 1070Mi (14%)

在上面的输出中，您可以看到如果 Pod 请求超过 1120m CPU 或者 6.23Gi 内存，节点将无法满足。

通过查看 Pods 部分，您将看到哪些 Pod 占用的节点上的资源。

Pod 可用的资源量小于节点容量，因为系统守护程序使用一部分可用资源。 [NodeStatus](https://kubernetes.io/docs/resources-reference/v1.17/#nodestatus-v1-core) 的 allocatable 字段给出了可用于 Pod 的资源量。 有关更多信息，请参阅 [节点可分配资源](https://git.k8s.io/community/contributors/design-proposals/node-allocatable.md)。

可以将 [资源配额](https://kubernetes.io/docs/concepts/policy/resource-quotas/) 功能配置为限制可以使用的资源总量。如果与 namespace 配合一起使用，就可以防止一个团队占用所有资源。

### 我的容器被终止了

您的容器可能因为资源枯竭而被终止了。要查看容器是否因为遇到资源限制而被杀死，请在相关的 Pod 上调用 kubectl describe pod：

kubectl describe pod simmemleak-hra99

Name: simmemleak-hra99

Namespace: default

Image(s): saadali/simmemleak

Node: kubernetes-node-tf0f/10.240.216.66

Labels: name=simmemleak

Status: Running

Reason:

Message:

IP: 10.244.2.75

Replication Controllers: simmemleak (1/1 replicas created)

Containers:

simmemleak:

Image: saadali/simmemleak

Limits:

cpu: 100m

memory: 50Mi

State: Running

Started: Tue, 07 Jul 2015 12:54:41 -0700

Last Termination State: Terminated

Exit Code: 1

Started: Fri, 07 Jul 2015 12:54:30 -0700

Finished: Fri, 07 Jul 2015 12:54:33 -0700

Ready: False

Restart Count: 5

Conditions:

Type Status

Ready False

Events:

FirstSeen LastSeen Count From SubobjectPath Reason Message

Tue, 07 Jul 2015 12:53:51 -0700 Tue, 07 Jul 2015 12:53:51 -0700 1 {scheduler } scheduled Successfully assigned simmemleak-hra99 to kubernetes-node-tf0f

Tue, 07 Jul 2015 12:53:51 -0700 Tue, 07 Jul 2015 12:53:51 -0700 1 {kubelet kubernetes-node-tf0f} implicitly required container POD pulled Pod container image "k8s.gcr.io/pause:0.8.0" already present on machine

Tue, 07 Jul 2015 12:53:51 -0700 Tue, 07 Jul 2015 12:53:51 -0700 1 {kubelet kubernetes-node-tf0f} implicitly required container POD created Created with docker id 6a41280f516d

Tue, 07 Jul 2015 12:53:51 -0700 Tue, 07 Jul 2015 12:53:51 -0700 1 {kubelet kubernetes-node-tf0f} implicitly required container POD started Started with docker id 6a41280f516d

Tue, 07 Jul 2015 12:53:51 -0700 Tue, 07 Jul 2015 12:53:51 -0700 1 {kubelet kubernetes-node-tf0f} spec.containers{simmemleak} created Created with docker id 87348f12526a

在上面的例子中，Restart Count: 5 意味着 Pod 中的 simmemleak 容器被终止并重启了五次。

您可以使用 kubectl get pod 命令加上 -o go-template=... 选项来获取之前终止容器的状态。

kubectl get pod -o go-template='{{range.status.containerStatuses}}{{"Container Name: "}}{{.name}}{{"\r\nLastState: "}}{{.lastState}}{{end}}' simmemleak-hra99

Container Name: simmemleak

LastState: map[terminated:map[exitCode:137 reason:OOM Killed startedAt:2015-07-07T20:58:43Z finishedAt:2015-07-07T20:58:43Z containerID:docker://0e4095bba1feccdfe7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22fad8b2]]

您可以看到容器因为 reason:OOM killed 被终止，OOM 表示 Out Of Memory。

### 本地临时存储

Kubernetes版本1.8引入了新资源\_ephemeral-storage\_，用于管理本地临时存储。 在每个Kubernetes节点中，kubelet的根目录（默认为 /var/lib/kubelet）和日志目录（ /var/log ）存储在节点的根分区上。 Pods还通过emptyDir卷，容器日志，镜像层和容器可写层共享和使用此分区。

该分区是“临时”分区，应用程序无法从该分区获得任何性能SLA（例如磁盘IOPS）。 本地临时存储管理仅适用于根分区。 图像层和可写层的可选分区超出范围。

**注意：** 如果使用可选的运行时分区，则根分区将不保存任何镜像层或可写层。

### 本地临时存储的请求和限制设置

Pod 的每个容器可以指定以下一项或多项：

* spec.containers[].resources.limits.ephemeral-storage
* spec.containers[].resources.requests.ephemeral-storage

对“临时存储”的限制和请求以字节为单位。您可以使用以下后缀之一将存储表示为纯整数或小数形式：E，P，T，G，M，K。您还可以使用2的幂次方：Ei，Pi，Ti，Gi，Mi，Ki。例如，以下内容表示的值其实大致相同：

128974848, 129e6, 129M, 123Mi

例如，以下Pod具有两个容器。每个容器都有一个2GiB的本地临时存储请求。每个容器的本地临时存储限制为4GiB。因此，该Pod要求本地临时存储空间为4GiB，存储空间限制为8GiB。

apiVersion: v1

kind: Pod

metadata:

name: frontend

spec:

containers:

- name: db

image: mysql

env:

- name: MYSQL\_ROOT\_PASSWORD

value: "password"

resources:

requests:

ephemeral-storage: "2Gi"

limits:

ephemeral-storage: "4Gi"

- name: wp

image: wordpress

resources:

requests:

ephemeral-storage: "2Gi"

limits:

ephemeral-storage: "4Gi"

### 如何调度临时存储请求的 Pod

创建Pod时，Kubernetes调度程序会选择一个节点来运行Pod。每个节点都可以为Pod提供最大数量的本地临时存储。 有关更多信息，请参见[节点可分配](https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable)。

调度程序会确保调度的容器的资源请求的总和小于节点的容量。

### 具有临时存储限制的 Pod 如何运行

对于容器级隔离，如果容器的可写层和日志使用量超出其存储限制，则将驱逐Pod。对于 pod 级别的隔离，如果来自所有容器的本地临时存储使用量以及 Pod 的 emptyDir 卷的总和超过限制，则将驱逐Pod。

### 监控临时存储消耗

使用本地临时存储时，kubelet 会持续对本地临时存储时进行监视。 通过定期扫描，来监视每个 emptyDir 卷，日志目录和可写层。 从Kubernetes 1.15开始，作为集群操作员的一个选项，可以通过[项目配额](http://xfs.org/docs/xfsdocs-xml-dev/XFS_User_Guide/tmp/en-US/html/xfs-quotas.html) 来管理 emptyDir 卷（但是不包括日志目录或可写层）。 项目配额最初是在XFS中实现的，最近又被移植到ext4fs中。 项目配额可用于监视和执行； 从Kubernetes 1.15开始，它们可用作Alpha功能仅用于监视。

配额比目录扫描更快，更准确。 将目录分配给项目时，在该目录下创建的所有文件都将在该项目中创建，内核仅需跟踪该项目中的文件正在使用多少块。 如果创建并删除了文件，但是文件描述符已打开，它将继续占用空间。 该空间将由配额跟踪，但目录扫描不会检查。

Kubernetes使用从1048576开始的项目ID。正在使用的ID注册于 /etc/projects 和 /etc/projid。 如果此范围内的项目ID用于系统上的其他目的，则这些项目ID必须在 /etc/projects 和 /etc/projid 中注册，以防止Kubernetes使用它们。

要启用项目配额，集群操作员必须执行以下操作：

* 在kubelet配置中启用 LocalStorageCapacityIsolationFSQuotaMonitoring = true 功能。 在Kubernetes 1.15中默认为 false，因此必须显式设置为 true。
* 确保根分区（或可选的运行时分区）是在启用项目配额的情况下构建的。 所有 XFS 文件系统都支持项目配额，但是 ext4 文件系统必须专门构建。
* 确保在启用了项目配额的情况下挂载了根分区（或可选的运行时分区）。

#### 在启用项目配额的情况下构建和挂载文件系统

XFS文件系统在构建时不需要任何特殊操作; 它们是在启用项目配额的情况下自动构建的。

Ext4fs文件系统必须在启用了配额的情况下构建，然后必须在文件系统中启用它们：

% sudo mkfs.ext4 other\_ext4fs\_args... -E quotatype=prjquota /dev/block\_device

% sudo tune2fs -O project -Q prjquota /dev/block\_device

要挂载文件系统，ext4fs 和 XFS 都需要在 /etc/fstab 中设置 prjquota 选项：

/dev/block\_device /var/kubernetes\_data defaults,prjquota 0 0

### 拓展资源

拓展资源是 kubernetes.io 域名之外的标准资源名称。它们允许集群管理员做分发，而且用户可以使用非Kubernetes内置资源。 使用扩展资源需要两个步骤。 首先，集群管理员必须分发拓展资源。 其次，用户必须在 Pod 中请求拓展资源。

curl --header "Content-Type: application/json-patch+json" **\**

--request PATCH **\**

--data '[{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"}]' **\**

http://k8s-master:8080/api/v1/nodes/k8s-node-1/status

### 管理拓展资源

#### 节点级拓展资源

节点级拓展资源绑定到节点。

##### 设备插件托管资源

有关如何在每个节点上分发设备插件托管资源的信息，请参阅[设备插件](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/)。

##### 其他资源

为了发布新的节点级拓展资源，集群操作员可以向API服务器提交 PATCH HTTP 请求， 以在 status.capacity 中为集群中的节点指定可用数量。 完成此操作后，节点的 status.capacity 将包含新资源。 由kubelet异步使用新资源自动更新 status.allocatable 字段。 请注意，由于调度程序在评估Pod适合性时使用节点的状态 status.allocatable 值， 因此在用新资源修补节点容量和请求在该节点上调度资源的第一个Pod之间可能会有短暂的延迟。

**示例:**

这是一个示例，显示了如何使用 curl 进行HTTP请求，该请求在主节点为 k8s-master 的子节点 k8s-node-1 上通告五个 example.com/foo 资源。

curl --header "Content-Type: application/json-patch+json" **\**

--request PATCH **\**

--data '[{"op": "add", "path": "/status/capacity/example.com~1foo", "value": "5"}]' **\**

http://k8s-master:8080/api/v1/nodes/k8s-node-1/status

**注意：**

在前面的请求中，~1 是 Patch 路径中字符 / 的编码。 JSON-Patch中的操作路径值被解释为JSON-Pointer。 有关更多详细信息，请参见 [IETF RFC 6901, section 3](https://tools.ietf.org/html/rfc6901#section-3).

#### 集群级扩展资源

群集级扩展资源不绑定到节点。 它们通常由调度程序扩展程序管理，这些程序处理资源消耗和资源配额。

您可以在[调度程序策略配置](https://github.com/kubernetes/kubernetes/blob/release-1.10/pkg/scheduler/api/v1/types.go#L31)中指定由调度程序扩展程序处理的扩展资源。

**示例:**

通过调度程序策略的以下配置，指示群集级扩展资源 “example.com/foo” 由调度程序扩展程序处理。

* 仅当Pod请求 “example.com/foo” 时，调度程序才会将 Pod 发送到调度程序扩展程序。
* ignoredByScheduler 字段指定调度程序不在其 PodFitsResources 字段中检查 “example.com/foo” 资源。
* {
* **"kind"**: "Policy",
* **"apiVersion"**: "v1",
* **"extenders"**: [
* {
* **"urlPrefix"**:"<extender-endpoint>",
* **"bindVerb"**: "bind",
* **"managedResources"**: [
* {
* **"name"**: "example.com/foo",
* **"ignoredByScheduler"**: **true**
* }
* ]
* }
* ]

}

### 消耗扩展资源

就像 CPU 和内存一样，用户可以使用 Pod 的扩展资源。 调度程序负责核算资源，因此不会同时将过多的可用资源分配给 Pod。

**注意：**

扩展资源取代了 Opaque 整数资源。 用户可以使用保留字 kubernetes.io 以外的任何域名前缀。

要在Pod中使用扩展资源，请在容器规范的 spec.containers[].resources.limits 映射中包含资源名称作为键。

**注意：**

扩展资源不能过量使用，因此如果容器规范中存在请求和限制，则它们必须一致。

仅当满足所有资源请求(包括 CPU ，内存和任何扩展资源)时，才能调度 Pod。 只要资源请求无法满足，则 Pod 保持在 PENDING 状态。

**示例:**

下面的 Pod 请求2个 CPU 和1个”example.com/foo”(扩展资源)。

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

containers:

- name: my-container

image: myimage

resources:

requests:

cpu: 2

example.com/foo: 1

limits:

example.com/foo: 1

### 接下来

* 获取将 [分配内存资源给容器和 Pod](https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/) 的实践经验
* 获取将 [分配 CPU 资源给容器和 Pod](https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/) 的实践经验
* [容器](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#container-v1-core)
* [资源需求](https://kubernetes.io/docs/resources-reference/v1.17/#resourcerequirements-v1-core)

### 反馈

## 将 Pod 分配给节点

你可以约束一个 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 只能在特定的 [Node(s)](https://kubernetes.io/docs/concepts/architecture/nodes/) 上运行，或者优先运行在特定的节点上。有几种方法可以实现这点，推荐的方法都是用[标签选择器](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)来进行选择。通常这样的约束不是必须的，因为调度器将自动进行合理的放置（比如，将 pod 分散到节点上，而不是将 pod 放置在可以资源不足的节点上等等），但在某些情况下，你可以需要更多控制 pod 停靠的节点，例如，确保 pod 最终落在连接了 SSD 的机器上，或者将来自两个不通的服务且有大量通信的 pod 放置在同一个可用区。

* [**nodeSelector**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#nodeselector)
* [**插曲：内置的节点标签**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#%e5%86%85%e7%bd%ae%e7%9a%84%e8%8a%82%e7%82%b9%e6%a0%87%e7%ad%be)
* [**节点隔离/限制**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#%e8%8a%82%e7%82%b9%e9%9a%94%e7%a6%bb-%e9%99%90%e5%88%b6)
* [**亲和与反亲和**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#%e4%ba%b2%e5%92%8c%e4%b8%8e%e5%8f%8d%e4%ba%b2%e5%92%8c)
* [**nodeName**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#nodename)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### nodeSelector

nodeSelector 是节点选择约束的最简单推荐形式。nodeSelector 是 PodSpec 的一个字段。它指定键值对的映射。为了使 pod 可以在节点上运行，节点必须具有每个指定的键值对作为标签（它也可以具有其他标签）。最常用的是一对键值对。

让我们来看一个使用 nodeSelector 的例子。

### 步骤零：先决条件

本示例假设你已基本了解 Kubernetes 的 pod 并且已经[建立一个 Kubernetes 集群](https://kubernetes.io/docs/setup/)。

### 步骤一：添加标签到节点

执行 kubectl get nodes 命令获取集群的节点名称。选择一个你要增加标签的节点，然后执行 kubectl label nodes <node-name> <label-key>=<label-value> 命令将标签添加到你所选择的节点上。例如，如果你的节点名称为 ‘kubernetes-foo-node-1.c.a-robinson.internal’ 并且想要的标签是 ‘disktype=ssd’，则可以执行 kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd 命令。

你可以通过重新运行 kubectl get nodes --show-labels 并且查看节点当前具有了一个标签来验证它是否有效。你也可以使用 kubectl describe node "nodename" 命令查看指定节点的标签完整列表。

### 步骤二：添加 nodeSelector 字段到 pod 配置中

拿任意一个你想运行的 pod 的配置文件，并且在其中添加一个 nodeSelector 部分。例如，如果下面是我的 pod 配置：

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

然后像下面这样添加 nodeSelector：

| [pods/pod-nginx.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/pod-nginx.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: nginx  labels:  env: test  spec:  containers:  - name: nginx  image: nginx  imagePullPolicy: IfNotPresent  nodeSelector:  disktype: ssd |

当你之后运行 kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml 命令，pod 将会调度到将标签添加到的节点上。你可以通过运行 kubectl get pods -o wide 并查看分配给 pod 的 “NODE” 来验证其是否有效。

### 插曲：内置的节点标签

除了你[附加](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#添加标签到节点)的标签外，节点还预先填充了一组标准标签。这些标签是

* [kubernetes.io/hostname](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#kubernetes-io-hostname)
* [failure-domain.beta.kubernetes.io/zone](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#failure-domain-beta-kubernetes-io-zone)
* [failure-domain.beta.kubernetes.io/region](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#failure-domain-beta-kubernetes-io-region)
* [beta.kubernetes.io/instance-type](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#beta-kubernetes-io-instance-type)
* [kubernetes.io/os](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#kubernetes-io-os)
* [kubernetes.io/arch](https://kubernetes.io/docs/reference/kubernetes-api/labels-annotations-taints/#kubernetes-io-arch)

**注意：**

这些标签的值特定于云供应商的，因此不能保证可靠。例如，kubernetes.io/hostname 的值在某些环境中可能与节点名称相同，但在其他环境中可能是一个不同的值。

### 节点隔离/限制

向 Node 对象添加标签可以将 pod 定位到特定的节点或节点组。这可以用来确保指定的 pod 只能运行在具有一定隔离性，安全性或监管属性的节点上。当为此目的使用标签时，强烈建议选择节点上的 kubelet 进程无法修改的标签键。这可以防止受感染的节点使用其 kubelet 凭据在自己的 Node 对象上设置这些标签，并影响调度器将工作负载调度到受感染的节点。

NodeRestriction 准入插件防止 kubelet 使用 node-restriction.kubernetes.io/ 前缀设置或修改标签。要使用该标签前缀进行节点隔离：

1. 检查是否在使用 Kubernetes v1.11+，以便 NodeRestriction 功能可用。
2. 确保你在使用[节点授权](https://kubernetes.io/docs/reference/access-authn-authz/node/)并且已经启用 [NodeRestriction 准入插件](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#noderestriction)。
3. 将 node-restriction.kubernetes.io/ 前缀下的标签添加到 Node 对象，然后在节点选择器中使用这些标签。例如，example.com.node-restriction.kubernetes.io/fips=true 或 example.com.node-restriction.kubernetes.io/pci-dss=true。

### 亲和与反亲和

nodeSelector 提供了一种非常简单的方法来将 pod 约束到具有特定标签的节点上。亲和/反亲和功能极大地扩展了你可以表达约束的类型。关键的增强点是

1. 语言更具表现力（不仅仅是“完全匹配的 AND”）
2. 你可以发现规则是“软”/“偏好”，而不是硬性要求，因此，如果调度器无法满足该要求，仍然调度该 pod
3. 你可以使用节点上（或其他拓扑域中）的 pod 的标签来约束，而不是使用节点本身的标签，来允许哪些 pod 可以或者不可以被放置在一起。

亲和功能包含两种类型的亲和，即“节点亲和”和“pod 间亲和/反亲和”。节点亲和就像现有的 nodeSelector（但具有上面列出的前两个好处），然而 pod 间亲和/反亲和约束 pod 标签而不是节点标签（在上面列出的第三项中描述，除了具有上面列出的第一和第二属性）。

#### 节点亲和

节点亲和概念上类似于 nodeSelector，它使你可以根据节点上的标签来约束 pod 可以调度到哪些节点。

目前有两种类型的节点亲和，分别为 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution。你可以视它们为“硬”和“软”，意思是，前者指定了将 pod 调度到一个节点上\*必须\*满足的规则（就像 nodeSelector 但使用更具表现力的语法），后者指定调度器将尝试执行单不能保证的\*偏好\*。名称的“IgnoredDuringExecution”部分意味着，类似于 nodeSelector 的工作原理，如果节点的标签在运行时发生变更，从而不再满足 pod 上的亲和规则，那么 pod 将仍然继续在该节点上运行。将来我们计划提供 requiredDuringSchedulingRequiredDuringExecution，它将类似于 requiredDuringSchedulingIgnoredDuringExecution，除了它会将 pod 从不再满足 pod 的节点亲和要求的节点上驱逐。

因此，requiredDuringSchedulingIgnoredDuringExecution 的示例将是“仅将 pod 运行在具有 Intel CPU 的节点上”，而 preferredDuringSchedulingIgnoredDuringExecution 的示例为“尝试将这组 pod 运行在 XYZ 故障区域，如果这不可能的话，则允许一些 pod 在其他地方运行”。

节点亲和通过 PodSpec 的 affinity 字段下的 nodeAffinity 字段进行指定。

下面是一个使用节点亲和的 pod 的实例：

| [pods/pod-with-node-affinity.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/pod-with-node-affinity.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: with-node-affinity  spec:  affinity:  nodeAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  nodeSelectorTerms:  - matchExpressions:  - key: kubernetes.io/e2e-az-name  operator: In  values:  - e2e-az1  - e2e-az2  preferredDuringSchedulingIgnoredDuringExecution:  - weight: 1  preference:  matchExpressions:  - key: another-node-label-key  operator: In  values:  - another-node-label-value  containers:  - name: with-node-affinity  image: k8s.gcr.io/pause:2.0 |

此节点亲和规则表示，pod 只能放置在具有标签键为 kubernetes.io/e2e-az-name 且 标签值为 e2e-az1 或 e2e-az2 的节点上。另外，在满足这些标准的节点中，具有标签键为 another-node-label-key 且标签值为 another-node-label-value 的节点应该优先使用。

你可以在上面的例子中看到 In 操作符的使用。新的节点亲和语法支持下面的操作符： In，NotIn，Exists，DoesNotExist，Gt，Lt。你可以使用 NotIn 和 DoesNotExist 来实现节点反亲和行为，或者使用[节点污点](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)将 pod 从特定节点中驱逐。

如果你同时指定了 nodeSelector 和 nodeAffinity，\*两者\*必须都要满足，才能将 pod 调度到候选节点上。

如果你指定了多个与 nodeAffinity 类型关联的 nodeSelectorTerms，则**如果其中一个** nodeSelectorTerms 满足的话，pod将可以调度到节点上。

如果你指定了多个与 nodeSelectorTerms 关联的 matchExpressions，则**只有当所有** matchExpressions 满足的话，pod 才会可以调度到节点上。

如果你修改或删除了 pod 所调度到的节点的标签，pod 不会被删除。换句话说，亲和选择只在 pod 调度期间有效。

preferredDuringSchedulingIgnoredDuringExecution 中的 weight 字段值的范围是 1-100。对于每个符合所有调度要求（资源请求，RequiredDuringScheduling 亲和表达式等）的节点，调度器将遍历该字段的元素来计算总和，并且如果节点匹配对应的MatchExpressions，则添加“权重”到总和。然后将这个评分与该节点的其他优先级函数的评分进行组合。总分最高的节点是最优选的。

#### pod 间亲和与反亲和

pod 间亲和与反亲和使你可以\*基于已经在节点上运行的 pod 的标签\*来约束 pod 可以调度到的节点，而不是基于节点上的标签。规则的格式为“如果 X 节点上已经运行了一个或多个 满足规则 Y 的pod，则这个 pod 应该（或者在非亲和的情况下不应该）运行在 X 节点”。Y 表示一个具有可选的关联命令空间列表的 LabelSelector；与节点不同，因为 pod 是命名空间限定的（因此 pod 上的标签也是命名空间限定的），因此作用于 pod 标签的标签选择器必须指定选择器应用在哪个命名空间。从概念上讲，X 是一个拓扑域，如节点，机架，云供应商地区，云供应商区域等。你可以使用 topologyKey 来表示它，topologyKey 是节点标签的键以便系统用来表示这样的拓扑域。请参阅上面[插曲：内置的节点标签](https://kubernetes.io/zh/docs/concepts/configuration/assign-pod-node/#内置的节点标签)部分中列出的标签键。

**注意：**

Pod 间亲和与反亲和需要大量的处理，这可能会显著减慢大规模集群中的调度。我们不建议在超过数百个节点的集群中使用它们。

**注意：**

Pod 反亲和需要对节点进行一致的标记，即集群中的每个节点必须具有适当的标签能够匹配 topologyKey。如果某些或所有节点缺少指定的 topologyKey 标签，可能会导致意外行为。

与节点亲和一样，当前有两种类型的 pod 亲和与反亲和，即 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution，分表表示“硬性”与“软性”要求。请参阅前面节点亲和部分中的描述。requiredDuringSchedulingIgnoredDuringExecution 亲和的一个示例是“将服务 A 和服务 B 的 pod 放置在同一区域，因为它们之间进行大量交流”，而 preferredDuringSchedulingIgnoredDuringExecution 反亲和的示例将是“将此服务的 pod 跨区域分布”（硬性要求是说不通的，因为你可能拥有的 pod 数多于区域数）。

Pod 间亲和通过 PodSpec 中 affinity 字段下的 podAffinity 字段进行指定。而 pod 间反亲和通过 PodSpec 中 affinity 字段下的 podAntiAffinity 字段进行指定。

### Pod 使用 pod 亲和 的示例：

| [pods/pod-with-pod-affinity.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/pods/pod-with-pod-affinity.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: with-pod-affinity  spec:  affinity:  podAffinity:  requiredDuringSchedulingIgnoredDuringExecution:  - labelSelector:  matchExpressions:  - key: security  operator: In  values:  - S1  topologyKey: failure-domain.beta.kubernetes.io/zone  podAntiAffinity:  preferredDuringSchedulingIgnoredDuringExecution:  - weight: 100  podAffinityTerm:  labelSelector:  matchExpressions:  - key: security  operator: In  values:  - S2  topologyKey: failure-domain.beta.kubernetes.io/zone  containers:  - name: with-pod-affinity  image: k8s.gcr.io/pause:2.0 |

在这个 pod 的 affinity 配置定义了一条 pod 亲和规则和一条 pod 反亲和规则。在此示例中，podAffinity 配置为 requiredDuringSchedulingIgnoredDuringExecution，然而 podAntiAffinity 配置为 preferredDuringSchedulingIgnoredDuringExecution。pod 亲和规则表示，仅当节点和至少一个已运行且有键为“security”且值为“S1”的标签的 pod 处于同一区域时，才可以将该 pod 调度到节点上。（更确切的说，如果节点 N 具有带有键 failure-domain.beta.kubernetes.io/zone 和某个值 V 的标签，则 pod 有资格在节点 N 上运行，以便集群中至少有一个节点具有键 failure-domain.beta.kubernetes.io/zone 和值为 V 的节点正在运行具有键“security”和值“S1”的标签的 pod。）pod 反亲和规则表示，如果节点已经运行了一个具有键“security”和值“S2”的标签的 pod，则该 pod 不希望将其调度到该节点上。（如果 topologyKey 为 failure-domain.beta.kubernetes.io/zone，则意味着当节点和具有键“security”和值“S2”的标签的 pod 处于相同的区域，pod 不能被调度到该节点上。）查阅[设计文档](https://git.k8s.io/community/contributors/design-proposals/scheduling/podaffinity.md)来获取更多 pod 亲和与反亲和的样例，包括 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution 两种配置。

Pod 亲和与反亲和的合法操作符有 In，NotIn，Exists，DoesNotExist。

原则上，topologyKey 可以是任何合法的标签键。然而，出于性能和安全原因，topologyKey 受到一些限制：

1. 对于亲和与 requiredDuringSchedulingIgnoredDuringExecution 要求的 pod 反亲和，topologyKey 不允许为空。
2. 对于 requiredDuringSchedulingIgnoredDuringExecution 要求的 pod 反亲和，准入控制器 LimitPodHardAntiAffinityTopology 被引入来限制 topologyKey 不为 kubernetes.io/hostname。如果你想使它可用于自定义拓扑结构，你必须修改准入控制器或者禁用它。
3. 对于 preferredDuringSchedulingIgnoredDuringExecution 要求的 pod 反亲和，空的 topologyKey 被解释为“所有拓扑结构”（这里的“所有拓扑结构”限制为 kubernetes.io/hostname，failure-domain.beta.kubernetes.io/zone 和 failure-domain.beta.kubernetes.io/region 的组合）。
4. 除上述情况外，topologyKey 可以是任何合法的标签键。

除了 labelSelector 和 topologyKey，你也可以指定表示命名空间的 namespaces 队列，labelSelector 也应该匹配它（这个与 labelSelector 和 topologyKey 的定义位于相同的级别）。如果忽略或者为空，则默认为 pod 亲和/反亲和的定义所在的命名空间。

所有与 requiredDuringSchedulingIgnoredDuringExecution 亲和与反亲和关联的 matchExpressions 必须满足，才能将 pod 调度到节点上。

#### 更实际的用例

Pod 间亲和与反亲和在与更高级别的集合（例如 ReplicaSets，StatefulSets，Deployments 等）一起使用时，它们可能更加有用。可以轻松配置一组应位于相同定义拓扑（例如，节点）中的工作负载。

##### 始终放置在相同节点上

在三节点集群中，一个 web 应用程序具有内存缓存，例如 redis。我们希望 web 服务器尽可能与缓存放置在同一位置。

下面是一个简单 redis deployment 的 yaml 代码段，它有三个副本和选择器标签 app=store。Deployment 配置了 PodAntiAffinity，用来确保调度器不会将副本调度到单个节点上。

apiVersion: apps/v1

kind: Deployment

metadata:

name: redis-cache

spec:

selector:

matchLabels:

app: store

replicas: 3

template:

metadata:

labels:

app: store

spec:

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- store

topologyKey: "kubernetes.io/hostname"

containers:

- name: redis-server

image: redis:3.2-alpine

下面 webserver deployment 的 yaml 代码段中配置了 podAntiAffinity 和 podAffinity。这将通知调度器将它的所有副本与具有 app=store 选择器标签的 pod 放置在一起。这还确保每个 web 服务器副本不会调度到单个节点上。

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-server

spec:

selector:

matchLabels:

app: web-store

replicas: 3

template:

metadata:

labels:

app: web-store

spec:

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- web-store

topologyKey: "kubernetes.io/hostname"

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- store

topologyKey: "kubernetes.io/hostname"

containers:

- name: web-app

image: nginx:1.12-alpine

如果我们创建了上面的两个 deployment，我们的三节点集群将如下表所示。

| node-1 | node-2 | node-3 |
| --- | --- | --- |
| webserver-1 | webserver-2 | webserver-3 |
| cache-1 | cache-2 | cache-3 |

如你所见，web-server 的三个副本都按照预期那样自动放置在同一位置。

kubectl get pods -o wide

输出类似于如下内容：

NAME READY STATUS RESTARTS AGE IP NODE

redis-cache-1450370735-6dzlj 1/1 Running 0 8m 10.192.4.2 kube-node-3

redis-cache-1450370735-j2j96 1/1 Running 0 8m 10.192.2.2 kube-node-1

redis-cache-1450370735-z73mh 1/1 Running 0 8m 10.192.3.1 kube-node-2

web-server-1287567482-5d4dz 1/1 Running 0 7m 10.192.2.3 kube-node-1

web-server-1287567482-6f7v5 1/1 Running 0 7m 10.192.4.3 kube-node-3

web-server-1287567482-s330j 1/1 Running 0 7m 10.192.3.2 kube-node-2

##### 永远不放置在相同节点

上面的例子使用 PodAntiAffinity 规则和 topologyKey: "kubernetes.io/hostname" 来部署 redis 集群以便在同一主机上没有两个实例。参阅 [ZooKeeper 教程](https://kubernetes.io/docs/tutorials/stateful-application/zookeeper/#tolerating-node-failure)，以获取配置反亲和来达到高可用性的 StatefulSet 的样例（使用了相同的技巧）。

### nodeName

nodeName 是节点选择约束的最简单方法，但是由于其自身限制，通常不使用它。nodeName 是 PodSpec 的一个字段。如果它不为空，调度器将忽略 pod，并且运行在它指定节点上的 kubelet 进程尝试运行该 pod。因此，如果 nodeName 在 PodSpec 中指定了，则它优先于上面的节点选择方法。

使用 nodeName 来选择节点的一些限制：

* 如果指定的节点不存在，
* 如果指定的节点没有资源来容纳 pod，pod 将会调度失败并且其原因将显示为，比如 OutOfmemory 或 OutOfcpu。
* 云环境中的节点名称并非总是可预测或稳定的。

下面的是使用 nodeName 字段的 pod 配置文件的例子：

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx

nodeName: kube-01

上面的 pod 将运行在 kube-01 节点上。

### 接下来

[污点](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)允许节点\*排斥\*一组 pod。

[节点亲和](https://git.k8s.io/community/contributors/design-proposals/scheduling/nodeaffinity.md)与 [pod 间亲和/反亲和](https://git.k8s.io/community/contributors/design-proposals/scheduling/podaffinity.md)的设计文档包含这些功能的其他背景信息。

一旦 pod 分配给 节点，kubelet 应用将运行该 pod 并且分配节点本地资源。[拓扑管理](https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/)

### 反馈

## Taint 和 Toleration

节点亲和性（详见[这里](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#node-affinity-beta-feature)），是 pod 的一种属性（偏好或硬性要求），它使 pod 被吸引到一类特定的节点。Taint 则相反，它使 节点 能够 排斥 一类特定的 pod。

Taint 和 toleration 相互配合，可以用来避免 pod 被分配到不合适的节点上。每个节点上都可以应用一个或多个 taint ，这表示对于那些不能容忍这些 taint 的 pod，是不会被该节点接受的。如果将 toleration 应用于 pod 上，则表示这些 pod 可以（但不要求）被调度到具有匹配 taint 的节点上。

* [**概念**](https://kubernetes.io/zh/docs/concepts/configuration/taint-and-toleration/#%e6%a6%82%e5%bf%b5)
* [**使用例子**](https://kubernetes.io/zh/docs/concepts/configuration/taint-and-toleration/#%e4%bd%bf%e7%94%a8%e4%be%8b%e5%ad%90)
* [**基于 taint 的驱逐**](https://kubernetes.io/zh/docs/concepts/configuration/taint-and-toleration/#%e5%9f%ba%e4%ba%8e-taint-%e7%9a%84%e9%a9%b1%e9%80%90)
* [**基于节点状态添加 taint**](https://kubernetes.io/zh/docs/concepts/configuration/taint-and-toleration/#%e5%9f%ba%e4%ba%8e%e8%8a%82%e7%82%b9%e7%8a%b6%e6%80%81%e6%b7%bb%e5%8a%a0-taint)

### 概念

您可以使用命令 [kubectl taint](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#taint) 给节点增加一个 taint。比如，

kubectl taint nodes node1 key=value:NoSchedule

给节点 node1 增加一个 taint，它的 key 是 key，value 是 value，effect 是 NoSchedule。这表示只有拥有和这个 taint 相匹配的 toleration 的 pod 才能够被分配到 node1 这个节点。您可以在 PodSpec 中定义 pod 的 toleration。下面两个 toleration 均与上面例子中使用 kubectl taint 命令创建的 taint 相匹配，因此如果一个 pod 拥有其中的任何一个 toleration 都能够被分配到 node1 ：

想删除上述命令添加的 taint ，您可以运行：

kubectl taint nodes node1 key:NoSchedule-

您可以在PodSpec中为容器设定容忍标签。以下两个容忍标签都与上面的 kubectl taint 创建的污点“匹配”， 因此具有任一容忍标签的Pod都可以将其调度到“ node1”上：

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

tolerations:

- key: "key"

operator: "Exists"

effect: "NoSchedule"

一个 toleration 和一个 taint 相“匹配”是指它们有一样的 key 和 effect ，并且：

* 如果 operator 是 Exists （此时 toleration 不能指定 value），或者
* 如果 operator 是 Equal ，则它们的 value 应该相等

**注意：**

存在两种特殊情况：

* 如果一个 toleration 的 key 为空且 operator 为 Exists ，表示这个 toleration 与任意的 key 、 value 和 effect 都匹配，即这个 toleration 能容忍任意 taint。
* tolerations:

- operator: "Exists"

* 如果一个 toleration 的 effect 为空，则 key 值与之相同的相匹配 taint 的 effect 可以是任意值。
* tolerations:
* - key: "key"

operator: "Exists"

上述例子使用到的 effect 的一个值 NoSchedule，您也可以使用另外一个值 PreferNoSchedule。这是“优化”或“软”版本的 NoSchedule ——系统会\*尽量\*避免将 pod 调度到存在其不能容忍 taint 的节点上，但这不是强制的。effect 的值还可以设置为 NoExecute ，下文会详细描述这个值。

您可以给一个节点添加多个 taint ，也可以给一个 pod 添加多个 toleration。Kubernetes 处理多个 taint 和 toleration 的过程就像一个过滤器：从一个节点的所有 taint 开始遍历，过滤掉那些 pod 中存在与之相匹配的 toleration 的 taint。余下未被过滤的 taint 的 effect 值决定了 pod 是否会被分配到该节点，特别是以下情况：

* 如果未被过滤的 taint 中存在一个以上 effect 值为 NoSchedule 的 taint，则 Kubernetes 不会将 pod 分配到该节点。
* 如果未被过滤的 taint 中不存在 effect 值为 NoSchedule 的 taint，但是存在 effect 值为 PreferNoSchedule 的 taint，则 Kubernetes 会\*尝试\*将 pod 分配到该节点。
* 如果未被过滤的 taint 中存在一个以上 effect 值为 NoExecute 的 taint，则 Kubernetes 不会将 pod 分配到该节点（如果 pod 还未在节点上运行），或者将 pod 从该节点驱逐（如果 pod 已经在节点上运行）。

例如，假设您给一个节点添加了如下的 taint

kubectl taint nodes node1 key1=value1:NoSchedule

kubectl taint nodes node1 key1=value1:NoExecute

kubectl taint nodes node1 key2=value2:NoSchedule

然后存在一个 pod，它有两个 toleration

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoSchedule"

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoExecute"

在这个例子中，上述 pod 不会被分配到上述节点，因为其没有 toleration 和第三个 taint 相匹配。但是如果在给节点添加 上述 taint 之前，该 pod 已经在上述节点运行，那么它还可以继续运行在该节点上，因为第三个 taint 是三个 taint 中唯一不能被这个 pod 容忍的。

通常情况下，如果给一个节点添加了一个 effect 值为 NoExecute 的 taint，则任何不能忍受这个 taint 的 pod 都会马上被驱逐，任何可以忍受这个 taint 的 pod 都不会被驱逐。但是，如果 pod 存在一个 effect 值为 NoExecute 的 toleration 指定了可选属性 tolerationSeconds 的值，则表示在给节点添加了上述 taint 之后，pod 还能继续在节点上运行的时间。例如，

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoExecute"

tolerationSeconds: 3600

这表示如果这个 pod 正在运行，然后一个匹配的 taint 被添加到其所在的节点，那么 pod 还将继续在节点上运行 3600 秒，然后被驱逐。如果在此之前上述 taint 被删除了，则 pod 不会被驱逐。

### 使用例子

通过 taint 和 toleration ，可以灵活地让 pod \*避开\*某些节点或者将 pod 从某些节点驱逐。下面是几个使用例子：

* **专用节点**：如果您想将某些节点专门分配给特定的一组用户使用，您可以给这些节点添加一个 taint（即， kubectl taint nodes nodename dedicated=groupName:NoSchedule），然后给这组用户的 pod 添加一个相对应的 toleration（通过编写一个自定义的[admission controller](https://kubernetes.io/docs/admin/admission-controllers/)，很容易就能做到）。拥有上述 toleration 的 pod 就能够被分配到上述专用节点，同时也能够被分配到集群中的其它节点。如果您希望这些 pod 只能被分配到上述专用节点，那么您还需要给这些专用节点另外添加一个和上述 taint 类似的 label （例如：dedicated=groupName），同时 还要在上述 admission controller 中给 pod 增加节点亲和性要求上述 pod 只能被分配到添加了 dedicated=groupName 标签的节点上。
* **配备了特殊硬件的节点**：在部分节点配备了特殊硬件（比如 GPU）的集群中，我们希望不需要这类硬件的 pod 不要被分配到这些特殊节点，以便为后继需要这类硬件的 pod 保留资源。要达到这个目的，可以先给配备了特殊硬件的节点添加 taint（例如 kubectl taint nodes nodename special=true:NoSchedule or kubectl taint nodes nodename special=true:PreferNoSchedule)，然后给使用了这类特殊硬件的 pod 添加一个相匹配的 toleration。和专用节点的例子类似，添加这个 toleration 的最简单的方法是使用自定义 [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/)。比如，我们推荐使用 [Extended Resources](https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#extended-resources) 来表示特殊硬件，给配置了特殊硬件的节点添加 taint 时包含 extended resource 名称，然后运行一个 [ExtendedResourceToleration](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#extendedresourcetoleration) admission controller。此时，因为节点已经被 taint 了，没有对应 toleration 的 Pod 会被调度到这些节点。但当你创建一个使用了 extended resource 的 Pod 时，ExtendedResourceToleration admission controller 会自动给 Pod 加上正确的 toleration ，这样 Pod 就会被自动调度到这些配置了特殊硬件件的节点上。这样就能够确保这些配置了特殊硬件的节点专门用于运行 需要使用这些硬件的 Pod，并且您无需手动给这些 Pod 添加 toleration。
* **基于 taint 的驱逐 （beta 特性）**: 这是在每个 pod 中配置的在节点出现问题时的驱逐行为，接下来的章节会描述这个特性

### 基于 taint 的驱逐

前文我们提到过 taint 的 effect 值 NoExecute ，它会影响已经在节点上运行的 pod \* 如果 pod 不能忍受effect 值为 NoExecute 的 taint，那么 pod 将马上被驱逐 \* 如果 pod 能够忍受effect 值为 NoExecute 的 taint，但是在 toleration 定义中没有指定 tolerationSeconds，则 pod 还会一直在这个节点上运行。 \* 如果 pod 能够忍受effect 值为 NoExecute 的 taint，而且指定了 tolerationSeconds，则 pod 还能在这个节点上继续运行这个指定的时间长度。

此外，Kubernetes 1.6 已经支持（alpha阶段）节点问题的表示。换句话说，当某种条件为真时，node controller会自动给节点添加一个 taint。当前内置的 taint 包括：

* node.kubernetes.io/not-ready：节点未准备好。这相当于节点状态 Ready 的值为 “False“。
* node.kubernetes.io/unreachable：node controller 访问不到节点. 这相当于节点状态 Ready 的值为 “Unknown“。
* node.kubernetes.io/out-of-disk：节点磁盘耗尽。
* node.kubernetes.io/memory-pressure：节点存在内存压力。
* node.kubernetes.io/disk-pressure：节点存在磁盘压力。
* node.kubernetes.io/network-unavailable：节点网络不可用。
* node.kubernetes.io/unschedulable: 节点不可调度。
* node.cloudprovider.kubernetes.io/uninitialized：如果 kubelet 启动时指定了一个 “外部” cloud provider，它将给当前节点添加一个 taint 将其标志为不可用。在 cloud-controller-manager 的一个 controller 初始化这个节点后，kubelet 将删除这个 taint。

在版本1.13中，TaintBasedEvictions 功能已升级为Beta，并且默认启用，因此污点会自动给节点添加这类 taint，上述基于节点状态 Ready 对 pod 进行驱逐的逻辑会被禁用。

**注意：**

注意：为了保证由于节点问题引起的 pod 驱逐[rate limiting](https://kubernetes.io/docs/concepts/architecture/nodes/)行为正常，系统实际上会以 rate-limited 的方式添加 taint。在像 master 和 node 通讯中断等场景下，这避免了 pod 被大量驱逐。

使用这个 beta 功能特性，结合 tolerationSeconds ，pod 就可以指定当节点出现一个或全部上述问题时还将在这个节点上运行多长的时间。

比如，一个使用了很多本地状态的应用程序在网络断开时，仍然希望停留在当前节点上运行一段较长的时间，愿意等待网络恢复以避免被驱逐。在这种情况下，pod 的 toleration 可能是下面这样的：

tolerations:

- key: "node.kubernetes.io/unreachable"

operator: "Exists"

effect: "NoExecute"

tolerationSeconds: 6000

注意，Kubernetes 会自动给 pod 添加一个 key 为 node.kubernetes.io/not-ready 的 toleration 并配置 tolerationSeconds=300，除非用户提供的 pod 配置中已经已存在了 key 为 node.kubernetes.io/not-ready 的 toleration。同样，Kubernetes 会给 pod 添加一个 key 为 node.kubernetes.io/unreachable 的 toleration 并配置 tolerationSeconds=300，除非用户提供的 pod 配置中已经已存在了 key 为 node.kubernetes.io/unreachable 的 toleration。

这种自动添加 toleration 机制保证了在其中一种问题被检测到时 pod 默认能够继续停留在当前节点运行 5 分钟。这两个默认 toleration 是由 [DefaultTolerationSeconds admission controller](https://git.k8s.io/kubernetes/plugin/pkg/admission/defaulttolerationseconds)添加的。

[DaemonSet](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/) 中的 pod 被创建时，针对以下 taint 自动添加的 NoExecute 的 toleration 将不会指定 tolerationSeconds：

* node.kubernetes.io/unreachable
* node.kubernetes.io/not-ready

这保证了出现上述问题时 DaemonSet 中的 pod 永远不会被驱逐，这和 TaintBasedEvictions 这个特性被禁用后的行为是一样的。

### 基于节点状态添加 taint

Node 生命周期控制器会自动创建与 Node 条件相对应的污点。 同样，调度器不检查节点条件，而是检查节点污点。这确保了节点条件不会影响调度到节点上的内容。用户可以通过添加适当的 Pod 容忍度来选择忽略某些 Node 的问题(表示为 Node 的调度条件)。 注意，TaintNodesByCondition 只会污染具有 NoSchedule 设定的节点。 NoExecute 效应由 TaintBasedEviction 控制， TaintBasedEviction 是 Beta 版功能，自 Kubernetes 1.13 起默认启用。

* node.kubernetes.io/memory-pressure
* node.kubernetes.io/disk-pressure
* node.kubernetes.io/out-of-disk (只适合 critical pod)
* node.kubernetes.io/unschedulable (1.10 或更高版本)
* node.kubernetes.io/network-unavailable (只适合 host network)

添加上述 toleration 确保了向后兼容，您也可以选择自由的向 DaemonSet 添加 toleration。

### 反馈

## Secret

Secret 对象类型用来保存敏感信息，例如密码、OAuth 令牌和 ssh key。 将这些信息放在 secret 中比放在 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 的定义或者 [容器镜像](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-image) 中来说更加安全和灵活。 参阅 [Secret 设计文档](https://git.k8s.io/community/contributors/design-proposals/auth/secrets.md) 获取更多详细信息。

* [**Secret 概览**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#secret-%e6%a6%82%e8%a7%88)
* [**使用 Secret**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#%e4%bd%bf%e7%94%a8-secret)
* [**详细**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#%e8%af%a6%e7%bb%86)
* [**使用案例**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#%e4%bd%bf%e7%94%a8%e6%a1%88%e4%be%8b)
* [**最佳实践**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#%e6%9c%80%e4%bd%b3%e5%ae%9e%e8%b7%b5)
* [**安全属性**](https://kubernetes.io/zh/docs/concepts/configuration/secret/#%e5%ae%89%e5%85%a8%e5%b1%9e%e6%80%a7)

### Secret 概览

Secret 是一种包含少量敏感信息例如密码、token 或 key 的对象。这样的信息可能会被放在 Pod spec 中或者镜像中；将其放在一个 secret 对象中可以更好地控制它的用途，并降低意外暴露的风险。

用户可以创建 secret，同时系统也创建了一些 secret。

要使用 secret，pod 需要引用 secret。Pod 可以用两种方式使用 secret：作为 [volume](https://kubernetes.io/docs/concepts/storage/volumes/) 中的文件被挂载到 pod 中的一个或者多个容器里，或者当 kubelet 为 pod 拉取镜像时使用。

#### 内置 secret

#### Service Account 使用 API 凭证自动创建和附加 secret

Kubernetes 自动创建包含访问 API 凭据的 secret，并自动修改您的 pod 以使用此类型的 secret。

如果需要，可以禁用或覆盖自动创建和使用API凭据。但是，如果您需要的只是安全地访问 apiserver，我们推荐这样的工作流程。

参阅 [Service Account](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/) 文档获取关于 Service Account 如何工作的更多信息。

#### 创建您自己的 Secret

#### 使用 kubectl 创建 Secret

假设有些 pod 需要访问数据库。这些 pod 需要使用的用户名和密码在您本地机器的 ./username.txt 和 ./password.txt 文件里。

*# Create files needed for rest of example.*

echo -n 'admin' > ./username.txt

echo -n '1f2d1e2e67df' > ./password.txt

kubectl create secret 命令将这些文件打包到一个 Secret 中并在 API server 中创建了一个对象。

kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt

secret "db-user-pass" created

**注意：**

特殊字符（例如 $, \\* 和 ! ）需要转义。 如果您使用的密码具有特殊字符，则需要使用 \\ 字符对其进行转义。 例如，如果您的实际密码是 S!B\\*d$zDsb ，则应通过以下方式执行命令： kubectl create secret generic dev-db-secret –from-literal=username=devuser –from-literal=password=S\!B\\\*d\$zDsb 您无需从文件中转义密码中的特殊字符（ --from-file ）。

您可以这样检查刚创建的 secret：

kubectl get secrets

NAME TYPE DATA AGE

db-user-pass Opaque 2 51s

kubectl describe secrets/db-user-pass

Name: db-user-pass

Namespace: default

Labels: <none>

Annotations: <none>

Type: Opaque

Data

====

password.txt: 12 bytes

username.txt: 5 bytes

**注意：**

默认情况下，kubectl get和kubectl describe避免显示密码的内容。 这是为了防止机密被意外地暴露给旁观者或存储在终端日志中。

请参阅 [解码 secret](https://kubernetes.io/zh/docs/concepts/configuration/secret/#解码-secret) 了解如何查看它们的内容。

#### 手动创建 Secret

您也可以先以 json 或 yaml 格式在文件中创建一个 secret 对象，然后创建该对象。 [密码](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.12/#secret-v1-core)包含两中类型，数据和字符串数据。 数据字段用于存储使用base64编码的任意数据。 提供stringData字段是为了方便起见，它允许您将机密数据作为未编码的字符串提供。

例如，要使用数据字段将两个字符串存储在 Secret 中，请按如下所示将它们转换为 base64：

echo -n 'admin' | base64

YWRtaW4=

echo -n '1f2d1e2e67df' | base64

MWYyZDFlMmU2N2Rm

现在可以像这样写一个 secret 对象：

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

data:

username: YWRtaW4=

password: MWYyZDFlMmU2N2Rm

使用 [kubectl apply](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#apply) 创建 secret：

kubectl apply -f ./secret.yaml

secret "mysecret" created

对于某些情况，您可能希望改用 stringData 字段。 此字段允许您将非 base64 编码的字符串直接放入 Secret 中， 并且在创建或更新 Secret 时将为您编码该字符串。

下面的一个实践示例提供了一个参考，您正在部署使用密钥存储配置文件的应用程序，并希望在部署过程中填补齐配置文件的部分内容。

如果您的应用程序使用以下配置文件：

apiUrl: "https://my.api.com/api/v1"

username: "user"

password: "password"

您可以使用以下方法将其存储在Secret中：

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

stringData:

config.yaml: |*-*

*apiUrl: "https://my.api.com/api/v1"*

username: {{username}}

password: {{password}}

然后，您的部署工具可以在执行 kubectl apply 之前替换模板的 {{username}} 和 {{password}} 变量。 stringData 是只写的便利字段。 检索 Secrets 时永远不会被输出。 例如，如果您运行以下命令：

kubectl get secret mysecret -o yaml

输出将类似于：

apiVersion: v1

kind: Secret

metadata:

creationTimestamp: 2018-11-15T20:40:59Z

name: mysecret

namespace: default

resourceVersion: "7225"

uid: c280ad2e-e916-11e8-98f2-025000000001

type: Opaque

data:

config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19

如果在 data 和 stringData 中都指定了字段，则使用 stringData 中的值。 例如，以下是 Secret 定义：

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

data:

username: YWRtaW4=

stringData:

username: administrator

secret 中的生成结果：

apiVersion: v1

kind: Secret

metadata:

creationTimestamp: 2018-11-15T20:46:46Z

name: mysecret

namespace: default

resourceVersion: "7579"

uid: 91460ecb-e917-11e8-98f2-025000000001

type: Opaque

data:

username: YWRtaW5pc3RyYXRvcg==

YWRtaW5pc3RyYXRvcg== 转换成了 administrator。

data和stringData的键必须由字母数字字符 ‘-’, ‘\_’ 或者 ‘.’ 组成。

\*\* 编码注意：\*\* 秘密数据的序列化 JSON 和 YAML 值被编码为base64字符串。 换行符在这些字符串中无效，因此必须省略。 在 Darwin / macOS 上使用 base64 实用程序时，用户应避免使用 -b 选项来分隔长行。 相反，Linux用户 应该 在 base64 命令中添加选项 -w 0， 或者，如果-w选项不可用的情况下， 执行 base64 | tr -d '\n'。

#### 从生成器创建 Secret

Kubectl 从1.14版本开始支持 [使用 Kustomize 管理对象](https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/) 使用此新功能，您还可以从生成器创建一个 Secret，然后将其应用于在 Apiserver 上创建对象。 生成器应在目录内的“ kustomization.yaml”中指定。

例如，从文件 ./username.txt 和 ./password.txt 生成一个 Secret。

*# Create a kustomization.yaml file with SecretGenerator*

cat <<EOF >./kustomization.yaml

secretGenerator:

- name: db-user-pass

files:

- username.txt

- password.txt

EOF

应用 kustomization 目录创建 Secret 对象。

$ kubectl apply -k .

secret/db-user-pass-96mffmfh4k created

您可以检查 secret 是否是这样创建的：

$ kubectl get secrets

NAME TYPE DATA AGE

db-user-pass-96mffmfh4k Opaque 2 51s

$ kubectl describe secrets/db-user-pass-96mffmfh4k

Name: db-user-pass

Namespace: default

Labels: <none>

Annotations: <none>

Type: Opaque

Data

====

password.txt: 12 bytes

username.txt: 5 bytes

例如，要从文字 username=admin 和 password=secret 生成秘密，可以在 kustomization.yaml 中将秘密生成器指定为

*# Create a kustomization.yaml file with SecretGenerator*

$ cat <<EOF >./kustomization.yaml

secretGenerator:

- name: db-user-pass

literals:

- username=admin

- password=secret

EOF

Apply the kustomization directory to create the Secret object.

$ kubectl apply -k .

secret/db-user-pass-dddghtt9b5 created

**注意：**

通过对内容进行序列化后，生成一个后缀作为 Secrets 的名称。 这样可以确保每次修改内容时都会生成一个新的Secret。

#### 解码 Secret

可以使用 kubectl get secret 命令获取 secret。例如，获取在上一节中创建的 secret：

kubectl get secret mysecret -o yaml

apiVersion: v1

kind: Secret

metadata:

creationTimestamp: 2016-01-22T18:41:56Z

name: mysecret

namespace: default

resourceVersion: "164619"

uid: cfee02d6-c137-11e5-8d73-42010af00002

type: Opaque

data:

username: YWRtaW4=

password: MWYyZDFlMmU2N2Rm

解码密码字段：

echo 'MWYyZDFlMmU2N2Rm' | base64 --decode

1f2d1e2e67df

#### 编辑 Secret

可以通过下面的命令编辑一个已经存在的 secret 。

kubectl edit secrets mysecret

这将打开默认配置的编辑器，并允许更新 data 字段中的base64编码的 secret：

# Please edit the object below. Lines beginning with a '#' will be ignored,

# and an empty file will abort the edit. If an error occurs while saving this file will be

# reopened with the relevant failures.

#

apiVersion: v1

data:

username: YWRtaW4=

password: MWYyZDFlMmU2N2Rm

kind: Secret

metadata:

annotations:

kubectl.kubernetes.io/last-applied-configuration: { ... }

creationTimestamp: 2016-01-22T18:41:56Z

name: mysecret

namespace: default

resourceVersion: "164619"

uid: cfee02d6-c137-11e5-8d73-42010af00002

type: Opaque

### 使用 Secret

Secret 可以作为数据卷被挂载，或作为[环境变量](https://kubernetes.io/docs/concepts/containers/container-environment-variables.md) 暴露出来以供 pod 中的容器使用。它们也可以被系统的其他部分使用，而不直接暴露在 pod 内。 例如，它们可以保存凭据，系统的其他部分应该用它来代表您与外部系统进行交互。

### 在 Pod 中使用 Secret 文件

在 Pod 中的 volume 里使用 Secret：

1. 创建一个 secret 或者使用已有的 secret。多个 pod 可以引用同一个 secret。
2. 修改您的 pod 的定义在 spec.volumes[] 下增加一个 volume。可以给这个 volume 随意命名，它的 spec.volumes[].secret.secretName 必须等于 secret 对象的名字。
3. 将 spec.containers[].volumeMounts[] 加到需要用到该 secret 的容器中。指定 spec.containers[].volumeMounts[].readOnly = true 和 spec.containers[].volumeMounts[].mountPath 为您想要该 secret 出现的尚未使用的目录。
4. 修改您的镜像并且／或者命令行让程序从该目录下寻找文件。Secret 的 data 映射中的每一个键都成为了 mountPath 下的一个文件名。

这是一个在 pod 中使用 volume 挂在 secret 的例子：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

readOnly: **true**

volumes:

- name: foo

secret:

secretName: mysecret

您想要用的每个 secret 都需要在 spec.volumes 中指明。

如果 pod 中有多个容器，每个容器都需要自己的 volumeMounts 配置块，但是每个 secret 只需要一个 spec.volumes。

您可以打包多个文件到一个 secret 中，或者使用的多个 secret，怎样方便就怎样来。

**向特性路径映射 secret 密钥**

我们还可以控制 Secret key 映射在 volume 中的路径。您可以使用 spec.volumes[].secret.items 字段修改每个 key 的目标路径：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

readOnly: **true**

volumes:

- name: foo

secret:

secretName: mysecret

items:

- key: username

path: my-group/my-username

将会发生什么呢：

* username secret 存储在 /etc/foo/my-group/my-username 文件中而不是 /etc/foo/username 中。
* password secret 没有被映射

如果使用了 spec.volumes[].secret.items，只有在 items 中指定的 key 被映射。要使用 secret 中所有的 key，所有这些都必须列在 items 字段中。所有列出的密钥必须存在于相应的 secret 中。否则，不会创建卷。

**Secret 文件权限**

您还可以指定 secret 将拥有的权限模式位文件。如果不指定，默认使用 0644。您可以为整个保密卷指定默认模式，如果需要，可以覆盖每个密钥。

例如，您可以指定如下默认模式：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

volumes:

- name: foo

secret:

secretName: mysecret

defaultMode: 256

然后，secret 将被挂载到 /etc/foo 目录，所有通过该 secret volume 挂载创建的文件的权限都是 0400。

请注意，JSON 规范不支持八进制符号，因此使用 256 值作为 0400 权限。如果您使用 yaml 而不是 json 作为 pod，则可以使用八进制符号以更自然的方式指定权限。

您还可以使用映射，如上一个示例，并为不同的文件指定不同的权限，如下所示：

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

volumes:

- name: foo

secret:

secretName: mysecret

items:

- key: username

path: my-group/my-username

mode: 511

在这种情况下，导致 /etc/foo/my-group/my-username 的文件的权限值为 0777。由于 JSON 限制，必须以十进制格式指定模式。

请注意，如果稍后阅读此权限值可能会以十进制格式显示。

**从 Volume 中消费 secret 值**

在挂载的 secret volume 的容器内，secret key 将作为文件，并且 secret 的值使用 base-64 解码并存储在这些文件中。这是在上面的示例容器内执行的命令的结果：

ls /etc/foo/

username

password

cat /etc/foo/username

admin

cat /etc/foo/password

1f2d1e2e67df

容器中的程序负责从文件中读取 secret。

**挂载的 secret 被自动更新**

当已经在 volume 中被消费的 secret 被更新时，被映射的 key 也将被更新。 Kubelet 在周期性同步时检查被挂载的 secret 是不是最新的。 但是，它正在使用其本地缓存来获取 Secret 的当前值。

缓存的类型可以使用 (ConfigMapAndSecretChangeDetectionStrategy 中的 [KubeletConfiguration 结构](https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/kubelet/config/v1beta1/types.go)). 它可以通过基于 ttl 的 watch(默认)传播，也可以将所有请求直接重定向到直接kube-apiserver。 结果，从更新密钥到将新密钥投射到 Pod 的那一刻的总延迟可能与 kubelet 同步周期 + 缓存传播延迟一样长，其中缓存传播延迟取决于所选的缓存类型。 (它等于观察传播延迟，缓存的ttl或相应为0)

**注意：**

使用 Secret 作为[子路径](https://kubernetes.io/docs/concepts/storage/volumes#using-subpath)卷安装的容器将不会收到 Secret 更新。

#### Secret 作为环境变量

将 secret 作为 pod 中的[环境变量](https://kubernetes.io/docs/concepts/containers/container-environment-variables.md)使用：

1. 创建一个 secret 或者使用一个已存在的 secret。多个 pod 可以引用同一个 secret。
2. 修改 Pod 定义，为每个要使用 secret 的容器添加对应 secret key 的环境变量。消费secret key 的环境变量应填充 secret 的名称，并键入 env[x].valueFrom.secretKeyRef。
3. 修改镜像并／或者命令行，以便程序在指定的环境变量中查找值。

这是一个使用 Secret 作为环境变量的示例：

apiVersion: v1

kind: Pod

metadata:

name: secret-env-pod

spec:

containers:

- name: mycontainer

image: redis

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: mysecret

key: username

- name: SECRET\_PASSWORD

valueFrom:

secretKeyRef:

name: mysecret

key: password

restartPolicy: Never

**消费环境变量里的 Secret 值**

在一个消耗环境变量 secret 的容器中，secret key 作为包含 secret 数据的 base-64 解码值的常规环境变量。这是从上面的示例在容器内执行的命令的结果：

echo $SECRET\_USERNAME

admin

echo $SECRET\_PASSWORD

1f2d1e2e67df

#### 使用 imagePullSecret

imagePullSecret 是将包含 Docker（或其他）镜像注册表密码的 secret 传递给 Kubelet 的一种方式，因此可以代表您的 pod 拉取私有镜像。

**手动指定 imagePullSecret**

imagePullSecret 的使用在 [镜像文档](https://kubernetes.io/docs/concepts/containers/images/#specifying-imagepullsecrets-on-a-pod) 中说明。

### 安排 imagePullSecrets 自动附加

您可以手动创建 imagePullSecret，并从 serviceAccount 引用它。使用该 serviceAccount 创建的任何 pod 和默认使用该 serviceAccount 的 pod 将会将其的 imagePullSecret 字段设置为服务帐户的 imagePullSecret 字段。有关该过程的详细说明，请参阅 [将 ImagePullSecrets 添加到服务帐户](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#adding-imagepullsecrets-to-a-service-account)。

#### 自动挂载手动创建的 Secret

手动创建的 secret（例如包含用于访问 github 帐户的令牌）可以根据其服务帐户自动附加到 pod。请参阅 [使用 PodPreset 向 Pod 中注入信息](https://kubernetes.io/docs/tasks/run-application/podpreset/) 以获取该进程的详细说明。

### 详细

### 限制

验证 secret volume 来源确保指定的对象引用实际上指向一个类型为 Secret 的对象。因此，需要在依赖于它的任何 pod 之前创建一个 secret。

Secret API 对象驻留在命名空间中。它们只能由同一命名空间中的 pod 引用。

每个 secret 的大小限制为1MB。这是为了防止创建非常大的 secret 会耗尽 apiserver 和 kubelet 的内存。然而，创建许多较小的 secret 也可能耗尽内存。更全面得限制 secret 对内存使用的功能还在计划中。

Kubelet 仅支持从 API server 获取的 Pod 使用 secret。这包括使用 kubectl 创建的任何 pod，或间接通过 replication controller 创建的 pod。它不包括通过 kubelet --manifest-url 标志，其 --config 标志或其 REST API 创建的pod（这些不是创建 pod 的常用方法）。

必须先创建 secret，除非将它们标记为可选项，否则必须在将其作为环境变量在 pod 中使用之前创建 secret。对不存在的 secret 的引用将阻止其启动。

使用 secretKeyRef ，引用指定的 secret 中的不存在的 key ，这会阻止 pod 的启动。

对于通过 envFrom 填充环境变量的 secret，这些环境变量具有被认为是无效环境变量名称的 key 将跳过这些键。该 pod 将被允许启动。将会有一个事件，其原因是 InvalidVariableNames，该消息将包含被跳过的无效键的列表。该示例显示一个 pod，它指的是包含2个无效键，1badkey 和 2alsobad 的默认/mysecret ConfigMap。

kubectl get events

LASTSEEN FIRSTSEEN COUNT NAME KIND SUBOBJECT TYPE REASON

0s 0s 1 dapi-test-pod Pod Warning InvalidEnvironmentVariableNames kubelet, 127.0.0.1 Keys [1badkey, 2alsobad] from the EnvFrom secret default/mysecret were skipped since they are considered invalid environment variable names.

### Secret 与 Pod 生命周期的联系

通过 API 创建 Pod 时，不会检查应用的 secret 是否存在。一旦 Pod 被调度，kubelet 就会尝试获取该 secret 的值。如果获取不到该 secret，或者暂时无法与 API server 建立连接，kubelet 将会定期重试。Kubelet 将会报告关于 pod 的事件，并解释它无法启动的原因。一旦获取到 secret，kubelet将创建并装载一个包含它的卷。在所有 pod 的卷被挂载之前，都不会启动 pod 的容器。

#### 使用案例

##### 使用案例：包含 ssh 密钥的 pod

创建一个包含 ssh key 的 secret：

kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/path/to/.ssh/id\_rsa --from-file=ssh-publickey=/path/to/.ssh/id\_rsa.pub

secret "ssh-key-secret" created

**警告:**

发送自己的 ssh 密钥之前要仔细思考：集群的其他用户可能有权访问该密钥。使用您想要共享 Kubernetes 群集的所有用户可以访问的服务帐户，如果它们遭到入侵，可以撤销。

现在我们可以创建一个使用 ssh 密钥引用 secret 的pod，并在一个卷中使用它：

apiVersion: v1

kind: Pod

metadata:

name: secret-test-pod

labels:

name: secret-test

spec:

volumes:

- name: secret-volume

secret:

secretName: ssh-key-secret

containers:

- name: ssh-test-container

image: mySshImage

volumeMounts:

- name: secret-volume

readOnly: **true**

mountPath: "/etc/secret-volume"

当容器中的命令运行时，密钥的片段将可在以下目录：

/etc/secret-volume/ssh-publickey

/etc/secret-volume/ssh-privatekey

然后容器可以自由使用密钥数据建立一个 ssh 连接。

##### 使用案例：包含 prod/test 凭据的 pod

下面的例子说明一个 pod 消费一个包含 prod 凭据的 secret，另一个 pod 使用测试环境凭据消费 secret。

通过秘钥生成器制作 kustomization.yaml

kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11

secret "prod-db-secret" created

kubectl create secret generic test-db-secret --from-literal=username=testuser --from-literal=password=iluvtests

secret "test-db-secret" created

**注意：**

特殊字符（例如 $, \\*, 和 !）需要转义。 如果您使用的密码具有特殊字符，则需要使用 \\ 字符对其进行转义。 例如，如果您的实际密码是 S!B\\*d$zDsb，则应通过以下方式执行命令：

kubectl create secret generic dev-db-secret --from-literal=username=devuser --from-literal=password=S**\\**!B**\\\\***d**\\**$zDsb

您无需从文件中转义密码中的特殊字符（ --from-file ）。

创建 pod ：

$ cat <<EOF > pod.yaml

apiVersion: v1

kind: List

items:

- kind: Pod

apiVersion: v1

metadata:

name: prod-db-client-pod

labels:

name: prod-db-client

spec:

volumes:

- name: secret-volume

secret:

secretName: prod-db-secret

containers:

- name: db-client-container

image: myClientImage

volumeMounts:

- name: secret-volume

readOnly: true

mountPath: "/etc/secret-volume"

- kind: Pod

apiVersion: v1

metadata:

name: test-db-client-pod

labels:

name: test-db-client

spec:

volumes:

- name: secret-volume

secret:

secretName: test-db-secret

containers:

- name: db-client-container

image: myClientImage

volumeMounts:

- name: secret-volume

readOnly: true

mountPath: "/etc/secret-volume"

EOF

加入 Pod 到同样的 kustomization.yaml 文件

$ cat <<EOF >> kustomization.yaml

resources:

- pod.yaml

EOF

部署所有的对象通过下面的命令

kubectl apply -k .

这两个容器将在其文件系统上显示以下文件，其中包含每个容器环境的值：

/etc/secret-volume/username

/etc/secret-volume/password

请注意，两个 pod 的 spec 配置中仅有一个字段有所不同；这有助于使用普通的 pod 配置模板创建具有不同功能的 pod。

您可以使用两个 service account 进一步简化基本 pod spec：一个名为 prod-user 拥有 prod-db-secret ，另一个称为 test-user 拥有 test-db-secret 。然后，pod spec 可以缩短为，例如：

apiVersion: v1

kind: Pod

metadata:

name: prod-db-client-pod

labels:

name: prod-db-client

spec:

serviceAccount: prod-db-client

containers:

- name: db-client-container

image: myClientImage

##### 使用案例：Secret 卷中以点号开头的文件

为了将数据“隐藏”起来（即文件名以点号开头的文件），简单地说让该键以一个点开始。例如，当如下 secret 被挂载到卷中：

apiVersion: v1

kind: Secret

metadata:

name: dotfile-secret

data:

.secret-file: dmFsdWUtMg0KDQo=

---

apiVersion: v1

kind: Pod

metadata:

name: secret-dotfiles-pod

spec:

volumes:

- name: secret-volume

secret:

secretName: dotfile-secret

containers:

- name: dotfile-test-container

image: k8s.gcr.io/busybox

command:

- ls

- "-l"

- "/etc/secret-volume"

volumeMounts:

- name: secret-volume

readOnly: **true**

mountPath: "/etc/secret-volume"

Secret-volume 将包含一个单独的文件，叫做 .secret-file，dotfile-test-container 的 /etc/secret-volume/.secret-file 路径下将有该文件。

**注意：**

以点号开头的文件在 ls -l 的输出中被隐藏起来了；列出目录内容时，必须使用 ls -la 才能查看它们。

##### 使用案例：Secret 仅对 pod 中的一个容器可见

考虑以下一个需要处理 HTTP 请求的程序，执行一些复杂的业务逻辑，然后使用 HMAC 签署一些消息。因为它具有复杂的应用程序逻辑，所以在服务器中可能会出现一个未被注意的远程文件读取漏洞，这可能会将私钥暴露给攻击者。

这可以在两个容器中分为两个进程：前端容器，用于处理用户交互和业务逻辑，但无法看到私钥；以及可以看到私钥的签名者容器，并且响应来自前端的简单签名请求（例如通过本地主机网络）。

使用这种分割方法，攻击者现在必须欺骗应用程序服务器才能进行任意的操作，这可能比使其读取文件更难。

### 最佳实践

#### 客户端使用 Secret API

当部署与 secret API 交互的应用程序时，应使用 [授权策略](https://kubernetes.io/docs/reference/access-authn-authz/authorization/)， 例如 [RBAC](https://kubernetes.io/docs/reference/access-authn-authz/rbac/) 来限制访问。

Secret 中的值对于不同的环境来说重要性可能不同，例如对于 Kubernetes 集群内部（例如 service account 令牌）和集群外部来说就不一样。即使一个应用程序可以理解其期望的与之交互的 secret 有多大的能力，但是同一命名空间中的其他应用程序却可能不这样认为。

由于这些原因，在命名空间中 watch 和 list secret 的请求是非常强大的功能，应该避免这样的行为，因为列出 secret 可以让客户端检查所有 secret 是否在该命名空间中。在群集中watch 和 list 所有 secret 的能力应该只保留给最有特权的系统级组件。

需要访问 secrets API 的应用程序应该根据他们需要的 secret 执行 get 请求。这允许管理员限制对所有 secret 的访问， 同时设置 [白名单访问](https://kubernetes.io/docs/reference/access-authn-authz/rbac/#referring-to-resources) 应用程序需要的各个实例。

为了提高循环获取的性能，客户端可以设计引用 secret 的资源，然后 watch 资源，在引用更改时重新请求 secret。 此外，还提出了一种 [”批量监控“ API](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/bulk_watch.md) 来让客户端 watch 每个资源，该功能可能会在将来的 Kubernetes 版本中提供。

### 安全属性

#### 保护

因为 secret 对象可以独立于使用它们的 pod 而创建，所以在创建、查看和编辑 pod 的流程中 secret 被暴露的风险较小。系统还可以对 secret 对象采取额外的预防措施，例如避免将其写入到磁盘中可能的位置。

只有当节点上的 pod 需要用到该 secret 时，该 secret 才会被发送到该节点上。它不会被写入磁盘，而是存储在 tmpfs 中。一旦依赖于它的 pod 被删除，它就被删除。

同一节点上的很多个 pod 可能拥有多个 secret。但是，只有 pod 请求的 secret 在其容器中才是可见的。因此，一个 pod 不能访问另一个 Pod 的 secret。

Pod 中有多个容器。但是，pod 中的每个容器必须请求其挂载卷中的 secret 卷才能在容器内可见。 这可以用于 [在 Pod 级别构建安全分区](https://kubernetes.io/zh/docs/concepts/configuration/secret/#使用案例secret-仅对-pod-中的一个容器可见)。

在大多数 Kubernetes 项目维护的发行版中，用户与 API server 之间的通信以及从 API server 到 kubelet 的通信都受到 SSL/TLS 的保护。通过这些通道传输时，secret 受到保护。

**FEATURE STATE:** Kubernetes v1.13 [beta](https://kubernetes.io/zh/docs/concepts/configuration/secret/)

你可以为 secret 数据开启[静态加密](https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/)，这样秘密信息就不会以明文形式存储到[etcd](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/)。

#### 风险

* API server 的 secret 数据以纯文本的方式存储在 etcd 中，因此：
  + 管理员应该为集群数据开启静态加密(需求 v1.13 或者更新)。
  + 管理员应该限制 admin 用户访问 etcd；
  + API server 中的 secret 数据位于 etcd 使用的磁盘上；管理员可能希望在不再使用时擦除/粉碎 etcd 使用的磁盘
  + 如果 etcd 运行在集群内，管理员应该确保 etcd 之间的通信使用 SSL/TLS 进行加密。
* 如果您将 secret 数据编码为 base64 的清单（JSON 或 YAML）文件，共享该文件或将其检入代码库，这样的话该密码将会被泄露。 Base64 编码不是一种加密方式，一样也是纯文本。
* 应用程序在从卷中读取 secret 后仍然需要保护 secret 的值，例如不会意外记录或发送给不信任方。
* 可以创建和使用 secret 的 pod 的用户也可以看到该 secret 的值。即使 API server 策略不允许用户读取 secret 对象，用户也可以运行暴露 secret 的 pod。
* 目前，任何节点的 root 用户都可以通过模拟 kubelet 来读取 API server 中的任何 secret。只有向实际需要它们的节点发送 secret 才能限制单个节点的根漏洞的影响，该功能还在计划中。

### 反馈

## 使用 kubeconfig 文件组织集群访问

使用 kubeconfig 文件来组织有关集群、用户、命名空间和身份认证机制的信息。kubectl 命令行工具使用 kubeconfig 文件来查找选择集群所需的信息，并与集群的 API 服务器进行通信。

**注意：** 注意：用于配置集群访问的文件称为 \*kubeconfig 文件\*。这是引用配置文件的通用方法。这并不意味着有一个名为 kubeconfig 的文件

默认情况下，kubectl 在 $HOME/.kube 目录下查找名为 config 的文件。您可以通过设置 KUBECONFIG 环境变量或者设置[--kubeconfig](https://kubernetes.io/docs/reference/generated/kubectl/kubectl/)参数来指定其他 kubeconfig 文件。

有关创建和指定 kubeconfig 文件的分步说明，请参阅[配置对多集群的访问](https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters)。

* [**支持多集群、用户和身份认证机制**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#%e6%94%af%e6%8c%81%e5%a4%9a%e9%9b%86%e7%be%a4-%e7%94%a8%e6%88%b7%e5%92%8c%e8%ba%ab%e4%bb%bd%e8%ae%a4%e8%af%81%e6%9c%ba%e5%88%b6)
* [**上下文（Context）**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#%e4%b8%8a%e4%b8%8b%e6%96%87-context)
* [**KUBECONFIG 环境变量**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#kubeconfig-%e7%8e%af%e5%a2%83%e5%8f%98%e9%87%8f)
* [**合并 kubeconfig 文件**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#%e5%90%88%e5%b9%b6-kubeconfig-%e6%96%87%e4%bb%b6)
* [**文件引用**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#%e6%96%87%e4%bb%b6%e5%bc%95%e7%94%a8)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/configuration/organize-cluster-access-kubeconfig/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 支持多集群、用户和身份认证机制

假设您有多个集群，并且您的用户和组件以多种方式进行身份认证。比如：

* 正在运行的 kubelet 可能使用证书在进行认证。
* 用户可能通过令牌进行认证。
* 管理员可能拥有多个证书集合提供给各用户。

使用 kubeconfig 文件，您可以组织集群、用户和命名空间。您还可以定义上下文，以便在集群和命名空间之间快速轻松地切换。

### 上下文（Context）

通过 kubeconfig 文件中的 context 元素，使用简便的名称来对访问参数进行分组。每个上下文都有三个参数：cluster、namespace 和 user。默认情况下，kubectl 命令行工具使用 当前上下文 中的参数与集群进行通信。

选择当前上下文

kubectl config use-context

### KUBECONFIG 环境变量

KUBECONFIG 环境变量包含一个 kubeconfig 文件列表。对于 Linux 和 Mac，列表以冒号分隔。对于 Windows，列表以分号分隔。KUBECONFIG 环境变量不是必要的。如果 KUBECONFIG 环境变量不存在，kubectl 使用默认的 kubeconfig 文件，$HOME/.kube/config。

如果 KUBECONFIG 环境变量存在，kubectl 使用 KUBECONFIG 环境变量中列举的文件合并后的有效配置。

### 合并 kubeconfig 文件

要查看配置，输入以下命令：

kubectl config view

如前所述，输出可能来自 kubeconfig 文件，也可能是合并多个 kubeconfig 文件的结果。

以下是 kubectl 在合并 kubeconfig 文件时使用的规则。

1. 如果设置了 --kubeconfig 参数，则仅使用指定的文件。不进行合并。此参数只能使用一次。

否则，如果设置了 KUBECONFIG 环境变量，将它用作应合并的文件列表。根据以下规则合并 KUBECONFIG 环境变量中列出的文件：

* 忽略空文件名。
* 对于内容无法反序列化的文件，产生错误信息。
* 第一个设置特定值或者映射键的文件将生效。
* 永远不会更改值或者映射键。示例：保留第一个文件的上下文以设置 current-context。示例：如果两个文件都指定了 red-user，则仅使用第一个文件的 red-user 中的值。即使第二个文件在 red-user 下有非冲突条目，也要丢弃它们。

有关设置 KUBECONFIG 环境变量的示例，请参阅[设置 KUBECONFIG 环境变量](https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/#set-the-kubeconfig-environment-variable)。

否则，使用默认的 kubeconfig 文件， $HOME/.kube/config，不进行合并。

1. 根据此链中的第一个匹配确定要使用的上下文。
   1. 如果存在，使用 --context 命令行参数。
   2. 使用合并的 kubeconfig 文件中的 current-context。

这种场景下允许空上下文。

1. 确定集群和用户。此时，可能有也可能没有上下文。根据此链中的第一个匹配确定集群和用户，这将运行两次：一次用于用户，一次用于集群。
   1. 如果存在，使用命令行参数：--user 或者 --cluster。
   2. 如果上下文非空，从上下文中获取用户或集群。

这种场景下用户和集群可以为空。

1. 确定要使用的实际集群信息。此时，可能有也可能没有集群信息。基于此链构建每个集群信息；第一个匹配项会被采用：
   1. 如果存在：--server、--certificate-authority 和 --insecure-skip-tls-verify，使用命令行参数。
   2. 如果合并的 kubeconfig 文件中存在集群信息属性，则使用它们。
   3. 如果没有 server 配置，则配置无效。
2. 确定要使用的实际用户信息。使用与集群信息相同的规则构建用户信息，但每个用户只允许一种身份认证技术：
   1. 如果存在：--client-certificate、--client-key、--username、--password 和 --token，使用命令行参数。
   2. 使用合并的 kubeconfig 文件中的 user 字段。
   3. 如果存在两种冲突技术，则配置无效。
3. 对于仍然缺失的任何信息，使用其对应的默认值，并可能提示输入身份认证信息。

### 文件引用

kubeconfig 文件中的文件和路径引用是相对于 kubeconfig 文件的位置。命令行上的文件引用是相当对于当前工作目录的。在 $HOME/.kube/config 中，相对路径按相对路径存储，绝对路径按绝对路径存储。

### 接下来

* [配置对多集群的访问](https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/)
* [kubectl config](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#config)

### 反馈

## Pod Priority and Preemption

**FEATURE STATE:** Kubernetes 1.14 [stable](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/)

[Pods](https://kubernetes.io/docs/user-guide/pods) can have priority. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

In Kubernetes 1.9 and later, Priority also affects scheduling order of Pods and out-of-resource eviction ordering on the Node.

Pod priority and preemption graduated to beta in Kubernetes 1.11 and to GA in Kubernetes 1.14. They have been enabled by default since 1.11.

In Kubernetes versions where Pod priority and preemption is still an alpha-level feature, you need to explicitly enable it. To use these features in the older versions of Kubernetes, follow the instructions in the documentation for your Kubernetes version, by going to the documentation archive version for your Kubernetes version.

| Kubernetes Version | Priority and Preemption State | Enabled by default |
| --- | --- | --- |
| 1.8 | alpha | no |
| 1.9 | alpha | no |
| 1.10 | alpha | no |
| 1.11 | beta | yes |
| 1.14 | stable | yes |

**Warning:** In a cluster where not all users are trusted, a malicious user could create pods at the highest possible priorities, causing other pods to be evicted/not get scheduled. To resolve this issue,[ResourceQuota](https://kubernetes.io/docs/concepts/policy/resource-quotas/) is augmented to support Pod priority. An admin can create ResourceQuota for users at specific priority levels, preventing them from creating pods at high priorities. This feature is in beta since Kubernetes 1.12.

* [**How to use priority and preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#how-to-use-priority-and-preemption)
* [**How to disable preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#how-to-disable-preemption)
* [**PriorityClass**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#priorityclass)
* [**Pod priority**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#pod-priority)
* [**Preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#preemption)
* [**Debugging Pod Priority and Preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#debugging-pod-priority-and-preemption)
* [**Interactions of Pod priority and QoS**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#interactions-of-pod-priority-and-qos)

### How to use priority and preemption

To use priority and preemption in Kubernetes 1.11 and later, follow these steps:

1. Add one or more [PriorityClasses](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#priorityclass).
2. Create Pods with[priorityClassName](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#pod-priority) set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add priorityClassNameto the Pod template of a  collection object like a Deployment.

Keep reading for more information about these steps.

If you try the feature and then decide to disable it, you must remove the PodPriority command-line flag or set it to false, and then restart the API server and scheduler. After the feature is disabled, the existing Pods keep their priority fields, but preemption is disabled, and priority fields are ignored. If the feature is disabled, you cannot set priorityClassName in new Pods.

### How to disable preemption

**Note:** In Kubernetes 1.12+, critical pods rely on scheduler preemption to be scheduled when a cluster is under resource pressure. For this reason, it is not recommended to disable preemption.

**Note:** In Kubernetes 1.15 and later, if the feature NonPreemptingPriority is enabled, PriorityClasses have the option to set preemptionPolicy: Never. This will prevent pods of that PriorityClass from preempting other pods.

In Kubernetes 1.11 and later, preemption is controlled by a kube-scheduler flag disablePreemption, which is set to false by default. If you want to disable preemption despite the above note, you can set disablePreemption to true.

This option is available in component configs only and is not available in old-style command line options. Below is a sample component config to disable preemption:

apiVersion: kubescheduler.config.k8s.io/v1alpha1

kind: KubeSchedulerConfiguration

algorithmSource:

provider: DefaultProvider

...

disablePreemption: **true**

### PriorityClass

A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the name field of the PriorityClass object’s metadata. The value is specified in the required value field. The higher the value, the higher the priority.

A PriorityClass object can have any 32-bit integer value smaller than or equal to 1 billion. Larger numbers are reserved for critical system Pods that should not normally be preempted or evicted. A cluster admin should create one PriorityClass object for each such mapping that they want.

PriorityClass also has two optional fields: globalDefault and description. The globalDefault field indicates that the value of this PriorityClass should be used for Pods without a priorityClassName. Only one PriorityClass with globalDefault set to true can exist in the system. If there is no PriorityClass with globalDefault set, the priority of Pods with no priorityClassName is zero.

The description field is an arbitrary string. It is meant to tell users of the cluster when they should use this PriorityClass.

#### Notes about PodPriority and existing clusters

* If you upgrade your existing cluster and enable this feature, the priority of your existing Pods is effectively zero.
* Addition of a PriorityClass with globalDefault set to true does not change the priorities of existing Pods. The value of such a PriorityClass is used only for Pods created after the PriorityClass is added.
* If you delete a PriorityClass, existing Pods that use the name of the deleted PriorityClass remain unchanged, but you cannot create more Pods that use the name of the deleted PriorityClass.

#### Example PriorityClass

apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

name: high-priority

value: 1000000

globalDefault: **false**

description: "This priority class should be used for XYZ service pods only."

#### Non-preempting PriorityClasses (alpha)

1.15 adds the PreemptionPolicy field as an alpha feature. It is disabled by default in 1.15, and requires the NonPreemptingPriority[feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to be enabled.

Pods with PreemptionPolicy: Never will be placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled will stay in the scheduling queue, until sufficient resources are free, and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries these pods and they cannot be scheduled, they will be retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods may still be preempted by other, high-priority pods.

PreemptionPolicy defaults to PreemptLowerPriority, which will allow pods of that PriorityClass to preempt lower-priority pods (as is existing default behavior). If PreemptionPolicy is set to Never, pods in that PriorityClass will be non-preempting.

An example use case is for data science workloads. A user may submit a job that they want to be prioritized above other workloads, but do not wish to discard existing work by preempting running pods. The high priority job with PreemptionPolicy: Never will be scheduled ahead of other queued pods, as soon as sufficient cluster resources “naturally” become free.

##### Example Non-preempting PriorityClass

apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

name: high-priority-nonpreempting

value: 1000000

preemptionPolicy: Never

globalDefault: **false**

description: "This priority class will not cause other pods to be preempted."

### Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the priorityClassName field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

priorityClassName: high-priority

### Effect of Pod priority on scheduling order

In Kubernetes 1.9 and later, when Pod priority is enabled, scheduler orders pending Pods by their priority and a pending Pod is placed ahead of other pending Pods with lower priority in the scheduling queue. As a result, the higher priority Pod may be scheduled sooner than Pods with lower priority if its scheduling requirements are met. If such Pod cannot be scheduled, scheduler will continue and tries to schedule other lower priority Pods.

### Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let’s call the pending Pod P. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than P would enable P to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get evicted from the Node. After the Pods are gone, P can be scheduled on the Node.

### User exposed information

When Pod P preempts one or more Pods on Node N, nominatedNodeName field of Pod P’s status is set to the name of Node N. This field helps scheduler track resources reserved for Pod P and also gives users information about preemptions in their clusters.

Please note that Pod P is not necessarily scheduled to the “nominated Node”. After victim Pods are preempted, they get their graceful termination period. If another node becomes available while scheduler is waiting for the victim Pods to terminate, scheduler will use the other node to schedule Pod P. As a result nominatedNodeName and nodeName of Pod spec are not always the same. Also, if scheduler preempts Pods on Node N, but then a higher priority Pod than Pod P arrives, scheduler may give Node N to the new higher priority Pod. In such a case, scheduler clears nominatedNodeName of Pod P. By doing this, scheduler makes Pod P eligible to preempt Pods on another Node.

### Limitations of preemption

#### Graceful termination of preemption victims

When Pods are preempted, the victims get their [graceful termination period](https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods). They have that much time to finish their work and exit. If they don’t, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod (P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue. Therefore, there is usually a time gap between the point that scheduler preempts victims and the time that Pod P is scheduled. In order to minimize this gap, one can set graceful termination period of lower priority Pods to zero or a small number.

#### PodDisruptionBudget is supported, but not guaranteed!

A [Pod Disruption Budget (PDB)](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/) allows application owners to limit the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes 1.9 supports PDB when preempting Pods, but respecting PDB is best effort. The Scheduler tries to find victims whose PDB are not violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

#### Inter-Pod affinity on lower-priority Pods

A Node is considered for preemption only when the answer to this question is yes: “If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending Pod be scheduled on the Node?”

**Note:** Preemption does not necessarily remove all lower-priority Pods. If the pending Pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority Pods.

#### Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here’s an example:

* Pod P is being considered for Node N.
* Pod Q is running on another Node in the same Zone as Node N.
* Pod P has Zone-wide anti-affinity with Pod Q (topologyKey: failure-domain.beta.kubernetes.io/zone).
* There are no other cases of anti-affinity between Pod P and other Pods in the Zone.
* In order to schedule Pod P on Node N, Pod Q can be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the Pod anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if there is enough demand and if we find an algorithm with reasonable performance.

### Debugging Pod Priority and Preemption

Pod Priority and Preemption is a major feature that could potentially disrupt Pod scheduling if it has bugs.

### Potential problems caused by Priority and Preemption

The followings are some of the potential problems that could be caused by bugs in the implementation of the feature. This list is not exhaustive.

#### Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If a user gives high priorities to certain Pods by mistake, these unintentional high priority Pods may cause preemption in the cluster. As mentioned above, Pod priority is specified by setting the priorityClassName field of podSpec. The integer value of priority is then resolved and populated to the priority field of podSpec.

To resolve the problem, priorityClassName of the Pods must be changed to use lower priority classes or should be left empty. Empty priorityClassName is resolved to zero by default.

When a Pod is preempted, there will be events recorded for the preempted Pod. Preemption should happen only when a cluster does not have enough resources for a Pod. In such cases, preemption happens only when the priority of the pending Pod (preemptor) is higher than the victim Pods. Preemption must not happen when there is no pending Pod, or when the pending Pods have equal or lower priority than the victims. If preemption happens in such scenarios, please file an issue.

#### Pods are preempted, but the preemptor is not scheduled

When pods are preempted, they receive their requested graceful termination period, which is by default 30 seconds, but it can be any different value as specified in the PodSpec. If the victim Pods do not terminate within this period, they are force-terminated. Once all the victims go away, the preemptor Pod can be scheduled.

While the preemptor Pod is waiting for the victims to go away, a higher priority Pod may be created that fits on the same node. In this case, the scheduler will schedule the higher priority Pod instead of the preemptor.

In the absence of such a higher priority Pod, we expect the preemptor Pod to be scheduled after the graceful termination period of the victims is over.

#### Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod and if no node is found, it tries to remove Pods with lower priority from one node to make room for the pending pod. If a node with low priority Pods is not feasible to run the pending Pod, the scheduler may choose another node with higher priority Pods (compared to the Pods on the other node) for preemption. The victims must still have lower priority than the preemptor Pod.

When there are multiple nodes available for preemption, the scheduler tries to choose the node with a set of Pods with lowest priority. However, if such Pods have PodDisruptionBudget that would be violated if they are preempted then the scheduler may choose another node with higher priority Pods.

When multiple nodes exist for preemption and none of the above scenarios apply, we expect the scheduler to choose a node with the lowest priority. If that is not the case, it may indicate a bug in the scheduler.

### Interactions of Pod priority and QoS

Pod priority and [QoS](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/resource-qos.md) are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler’s preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by PodDisruptionBudget.

The only component that considers both QoS and Pod priority is [Kubelet out-of-resource eviction](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/). The kubelet ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by Priority, and then by the consumption of the starved compute resource relative to the Pods’ scheduling requests. See [Evicting end-user pods](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#evicting-end-user-pods) for more details. Kubelet out-of-resource eviction does not evict Pods whose usage does not exceed their requests. If a Pod with lower priority is not exceeding its requests, it won’t be evicted. Another Pod with higher priority that exceeds its requests may be evicted.

### Feedback

### Pod Priority and Preemption

**FEATURE STATE:** Kubernetes 1.14 [stable](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/)

[Pods](https://kubernetes.io/docs/user-guide/pods) can have priority. Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

In Kubernetes 1.9 and later, Priority also affects scheduling order of Pods and out-of-resource eviction ordering on the Node.

Pod priority and preemption graduated to beta in Kubernetes 1.11 and to GA in Kubernetes 1.14. They have been enabled by default since 1.11.

In Kubernetes versions where Pod priority and preemption is still an alpha-level feature, you need to explicitly enable it. To use these features in the older versions of Kubernetes, follow the instructions in the documentation for your Kubernetes version, by going to the documentation archive version for your Kubernetes version.

| Kubernetes Version | Priority and Preemption State | Enabled by default |
| --- | --- | --- |
| 1.8 | alpha | no |
| 1.9 | alpha | no |
| 1.10 | alpha | no |
| 1.11 | beta | yes |
| 1.14 | stable | yes |

**Warning:** In a cluster where not all users are trusted, a malicious user could create pods at the highest possible priorities, causing other pods to be evicted/not get scheduled. To resolve this issue,[ResourceQuota](https://kubernetes.io/docs/concepts/policy/resource-quotas/) is augmented to support Pod priority. An admin can create ResourceQuota for users at specific priority levels, preventing them from creating pods at high priorities. This feature is in beta since Kubernetes 1.12.

* [**How to use priority and preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#how-to-use-priority-and-preemption)
* [**How to disable preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#how-to-disable-preemption)
* [**PriorityClass**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#priorityclass)
* [**Pod priority**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#pod-priority)
* [**Preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#preemption)
* [**Debugging Pod Priority and Preemption**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#debugging-pod-priority-and-preemption)
* [**Interactions of Pod priority and QoS**](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#interactions-of-pod-priority-and-qos)

### How to use priority and preemption

To use priority and preemption in Kubernetes 1.11 and later, follow these steps:

1. Add one or more [PriorityClasses](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#priorityclass).
2. Create Pods with[priorityClassName](https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/#pod-priority) set to one of the added PriorityClasses. Of course you do not need to create the Pods directly; normally you would add priorityClassNameto the Pod template of a  collection object like a Deployment.

Keep reading for more information about these steps.

If you try the feature and then decide to disable it, you must remove the PodPriority command-line flag or set it to false, and then restart the API server and scheduler. After the feature is disabled, the existing Pods keep their priority fields, but preemption is disabled, and priority fields are ignored. If the feature is disabled, you cannot set priorityClassName in new Pods.

### How to disable preemption

**Note:** In Kubernetes 1.12+, critical pods rely on scheduler preemption to be scheduled when a cluster is under resource pressure. For this reason, it is not recommended to disable preemption.

**Note:** In Kubernetes 1.15 and later, if the feature NonPreemptingPriority is enabled, PriorityClasses have the option to set preemptionPolicy: Never. This will prevent pods of that PriorityClass from preempting other pods.

In Kubernetes 1.11 and later, preemption is controlled by a kube-scheduler flag disablePreemption, which is set to false by default. If you want to disable preemption despite the above note, you can set disablePreemption to true.

This option is available in component configs only and is not available in old-style command line options. Below is a sample component config to disable preemption:

apiVersion: kubescheduler.config.k8s.io/v1alpha1

kind: KubeSchedulerConfiguration

algorithmSource:

provider: DefaultProvider

...

disablePreemption: **true**

### PriorityClass

A PriorityClass is a non-namespaced object that defines a mapping from a priority class name to the integer value of the priority. The name is specified in the name field of the PriorityClass object’s metadata. The value is specified in the required value field. The higher the value, the higher the priority.

A PriorityClass object can have any 32-bit integer value smaller than or equal to 1 billion. Larger numbers are reserved for critical system Pods that should not normally be preempted or evicted. A cluster admin should create one PriorityClass object for each such mapping that they want.

PriorityClass also has two optional fields: globalDefault and description. The globalDefault field indicates that the value of this PriorityClass should be used for Pods without a priorityClassName. Only one PriorityClass with globalDefault set to true can exist in the system. If there is no PriorityClass with globalDefault set, the priority of Pods with no priorityClassName is zero.

The description field is an arbitrary string. It is meant to tell users of the cluster when they should use this PriorityClass.

### Notes about PodPriority and existing clusters

* If you upgrade your existing cluster and enable this feature, the priority of your existing Pods is effectively zero.
* Addition of a PriorityClass with globalDefault set to true does not change the priorities of existing Pods. The value of such a PriorityClass is used only for Pods created after the PriorityClass is added.
* If you delete a PriorityClass, existing Pods that use the name of the deleted PriorityClass remain unchanged, but you cannot create more Pods that use the name of the deleted PriorityClass.

### Example PriorityClass

apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

name: high-priority

value: 1000000

globalDefault: **false**

description: "This priority class should be used for XYZ service pods only."

### Non-preempting PriorityClasses (alpha)

1.15 adds the PreemptionPolicy field as an alpha feature. It is disabled by default in 1.15, and requires the NonPreemptingPriority[feature gate](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/) to be enabled.

Pods with PreemptionPolicy: Never will be placed in the scheduling queue ahead of lower-priority pods, but they cannot preempt other pods. A non-preempting pod waiting to be scheduled will stay in the scheduling queue, until sufficient resources are free, and it can be scheduled. Non-preempting pods, like other pods, are subject to scheduler back-off. This means that if the scheduler tries these pods and they cannot be scheduled, they will be retried with lower frequency, allowing other pods with lower priority to be scheduled before them.

Non-preempting pods may still be preempted by other, high-priority pods.

PreemptionPolicy defaults to PreemptLowerPriority, which will allow pods of that PriorityClass to preempt lower-priority pods (as is existing default behavior). If PreemptionPolicy is set to Never, pods in that PriorityClass will be non-preempting.

An example use case is for data science workloads. A user may submit a job that they want to be prioritized above other workloads, but do not wish to discard existing work by preempting running pods. The high priority job with PreemptionPolicy: Never will be scheduled ahead of other queued pods, as soon as sufficient cluster resources “naturally” become free.

#### Example Non-preempting PriorityClass

apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

name: high-priority-nonpreempting

value: 1000000

preemptionPolicy: Never

globalDefault: **false**

description: "This priority class will not cause other pods to be preempted."

### Pod priority

After you have one or more PriorityClasses, you can create Pods that specify one of those PriorityClass names in their specifications. The priority admission controller uses the priorityClassName field and populates the integer value of the priority. If the priority class is not found, the Pod is rejected.

The following YAML is an example of a Pod configuration that uses the PriorityClass created in the preceding example. The priority admission controller checks the specification and resolves the priority of the Pod to 1000000.

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

priorityClassName: high-priority

### Effect of Pod priority on scheduling order

In Kubernetes 1.9 and later, when Pod priority is enabled, scheduler orders pending Pods by their priority and a pending Pod is placed ahead of other pending Pods with lower priority in the scheduling queue. As a result, the higher priority Pod may be scheduled sooner than Pods with lower priority if its scheduling requirements are met. If such Pod cannot be scheduled, scheduler will continue and tries to schedule other lower priority Pods.

### Preemption

When Pods are created, they go to a queue and wait to be scheduled. The scheduler picks a Pod from the queue and tries to schedule it on a Node. If no Node is found that satisfies all the specified requirements of the Pod, preemption logic is triggered for the pending Pod. Let’s call the pending Pod P. Preemption logic tries to find a Node where removal of one or more Pods with lower priority than P would enable P to be scheduled on that Node. If such a Node is found, one or more lower priority Pods get evicted from the Node. After the Pods are gone, P can be scheduled on the Node.

### User exposed information

When Pod P preempts one or more Pods on Node N, nominatedNodeName field of Pod P’s status is set to the name of Node N. This field helps scheduler track resources reserved for Pod P and also gives users information about preemptions in their clusters.

Please note that Pod P is not necessarily scheduled to the “nominated Node”. After victim Pods are preempted, they get their graceful termination period. If another node becomes available while scheduler is waiting for the victim Pods to terminate, scheduler will use the other node to schedule Pod P. As a result nominatedNodeName and nodeName of Pod spec are not always the same. Also, if scheduler preempts Pods on Node N, but then a higher priority Pod than Pod P arrives, scheduler may give Node N to the new higher priority Pod. In such a case, scheduler clears nominatedNodeName of Pod P. By doing this, scheduler makes Pod P eligible to preempt Pods on another Node.

### Limitations of preemption

#### Graceful termination of preemption victims

When Pods are preempted, the victims get their [graceful termination period](https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods). They have that much time to finish their work and exit. If they don’t, they are killed. This graceful termination period creates a time gap between the point that the scheduler preempts Pods and the time when the pending Pod (P) can be scheduled on the Node (N). In the meantime, the scheduler keeps scheduling other pending Pods. As victims exit or get terminated, the scheduler tries to schedule Pods in the pending queue. Therefore, there is usually a time gap between the point that scheduler preempts victims and the time that Pod P is scheduled. In order to minimize this gap, one can set graceful termination period of lower priority Pods to zero or a small number.

#### PodDisruptionBudget is supported, but not guaranteed!

A [Pod Disruption Budget (PDB)](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/) allows application owners to limit the number of Pods of a replicated application that are down simultaneously from voluntary disruptions. Kubernetes 1.9 supports PDB when preempting Pods, but respecting PDB is best effort. The Scheduler tries to find victims whose PDB are not violated by preemption, but if no such victims are found, preemption will still happen, and lower priority Pods will be removed despite their PDBs being violated.

#### Inter-Pod affinity on lower-priority Pods

A Node is considered for preemption only when the answer to this question is yes: “If all the Pods with lower priority than the pending Pod are removed from the Node, can the pending Pod be scheduled on the Node?”

**Note:** Preemption does not necessarily remove all lower-priority Pods. If the pending Pod can be scheduled by removing fewer than all lower-priority Pods, then only a portion of the lower-priority Pods are removed. Even so, the answer to the preceding question must be yes. If the answer is no, the Node is not considered for preemption.

If a pending Pod has inter-pod affinity to one or more of the lower-priority Pods on the Node, the inter-Pod affinity rule cannot be satisfied in the absence of those lower-priority Pods. In this case, the scheduler does not preempt any Pods on the Node. Instead, it looks for another Node. The scheduler might find a suitable Node or it might not. There is no guarantee that the pending Pod can be scheduled.

Our recommended solution for this problem is to create inter-Pod affinity only towards equal or higher priority Pods.

#### Cross node preemption

Suppose a Node N is being considered for preemption so that a pending Pod P can be scheduled on N. P might become feasible on N only if a Pod on another Node is preempted. Here’s an example:

* Pod P is being considered for Node N.
* Pod Q is running on another Node in the same Zone as Node N.
* Pod P has Zone-wide anti-affinity with Pod Q (topologyKey: failure-domain.beta.kubernetes.io/zone).
* There are no other cases of anti-affinity between Pod P and other Pods in the Zone.
* In order to schedule Pod P on Node N, Pod Q can be preempted, but scheduler does not perform cross-node preemption. So, Pod P will be deemed unschedulable on Node N.

If Pod Q were removed from its Node, the Pod anti-affinity violation would be gone, and Pod P could possibly be scheduled on Node N.

We may consider adding cross Node preemption in future versions if there is enough demand and if we find an algorithm with reasonable performance.

### Debugging Pod Priority and Preemption

Pod Priority and Preemption is a major feature that could potentially disrupt Pod scheduling if it has bugs.

### Potential problems caused by Priority and Preemption

The followings are some of the potential problems that could be caused by bugs in the implementation of the feature. This list is not exhaustive.

#### Pods are preempted unnecessarily

Preemption removes existing Pods from a cluster under resource pressure to make room for higher priority pending Pods. If a user gives high priorities to certain Pods by mistake, these unintentional high priority Pods may cause preemption in the cluster. As mentioned above, Pod priority is specified by setting the priorityClassName field of podSpec. The integer value of priority is then resolved and populated to the priority field of podSpec.

To resolve the problem, priorityClassName of the Pods must be changed to use lower priority classes or should be left empty. Empty priorityClassName is resolved to zero by default.

When a Pod is preempted, there will be events recorded for the preempted Pod. Preemption should happen only when a cluster does not have enough resources for a Pod. In such cases, preemption happens only when the priority of the pending Pod (preemptor) is higher than the victim Pods. Preemption must not happen when there is no pending Pod, or when the pending Pods have equal or lower priority than the victims. If preemption happens in such scenarios, please file an issue.

#### Pods are preempted, but the preemptor is not scheduled

When pods are preempted, they receive their requested graceful termination period, which is by default 30 seconds, but it can be any different value as specified in the PodSpec. If the victim Pods do not terminate within this period, they are force-terminated. Once all the victims go away, the preemptor Pod can be scheduled.

While the preemptor Pod is waiting for the victims to go away, a higher priority Pod may be created that fits on the same node. In this case, the scheduler will schedule the higher priority Pod instead of the preemptor.

In the absence of such a higher priority Pod, we expect the preemptor Pod to be scheduled after the graceful termination period of the victims is over.

#### Higher priority Pods are preempted before lower priority pods

The scheduler tries to find nodes that can run a pending Pod and if no node is found, it tries to remove Pods with lower priority from one node to make room for the pending pod. If a node with low priority Pods is not feasible to run the pending Pod, the scheduler may choose another node with higher priority Pods (compared to the Pods on the other node) for preemption. The victims must still have lower priority than the preemptor Pod.

When there are multiple nodes available for preemption, the scheduler tries to choose the node with a set of Pods with lowest priority. However, if such Pods have PodDisruptionBudget that would be violated if they are preempted then the scheduler may choose another node with higher priority Pods.

When multiple nodes exist for preemption and none of the above scenarios apply, we expect the scheduler to choose a node with the lowest priority. If that is not the case, it may indicate a bug in the scheduler.

### Interactions of Pod priority and QoS

Pod priority and [QoS](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/resource-qos.md) are two orthogonal features with few interactions and no default restrictions on setting the priority of a Pod based on its QoS classes. The scheduler’s preemption logic does not consider QoS when choosing preemption targets. Preemption considers Pod priority and attempts to choose a set of targets with the lowest priority. Higher-priority Pods are considered for preemption only if the removal of the lowest priority Pods is not sufficient to allow the scheduler to schedule the preemptor Pod, or if the lowest priority Pods are protected by PodDisruptionBudget.

The only component that considers both QoS and Pod priority is [Kubelet out-of-resource eviction](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/). The kubelet ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by Priority, and then by the consumption of the starved compute resource relative to the Pods’ scheduling requests. See [Evicting end-user pods](https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/#evicting-end-user-pods) for more details. Kubelet out-of-resource eviction does not evict Pods whose usage does not exceed their requests. If a Pod with lower priority is not exceeding its requests, it won’t be evicted. Another Pod with higher priority that exceeds its requests may be evicted.

### Feedback

## 调度框架

**FEATURE STATE:** Kubernetes 1.15 [alpha](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/)

调度框架是 Kubernetes Scheduler 的一种新的可插入架构，可以简化调度器的自定义。它向现有的调度器增加了一组新的“插件” API。插件被编译到调度器程序中。这些 API 允许大多数调度功能以插件的形式实现，同时使调度“核心”保持简单且可维护。请参考[调度框架的设计提案](https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/20180409-scheduling-framework.md)获取框架设计的更多技术信息。

* [**调度周期和绑定周期**](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#%e8%b0%83%e5%ba%a6%e5%91%a8%e6%9c%9f%e5%92%8c%e7%bb%91%e5%ae%9a%e5%91%a8%e6%9c%9f)
* [**扩展点**](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#%e6%89%a9%e5%b1%95%e7%82%b9)
* [**插件 API**](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#%e6%8f%92%e4%bb%b6-api)

### 框架工作流程

调度框架定义了一些扩展点。调度器插件注册后在一个或多个扩展点处被调用。这些插件中的一些可以改变调度决策，而另一些仅用于提供信息。

每次调度一个 pod 的尝试都分为两个阶段，即**调度周期**和**绑定周期**。

### 调度周期和绑定周期

调度周期为 pod 选择一个节点，绑定周期将该决策应用于集群。调度周期和绑定周期一起被称为“调度上下文”。

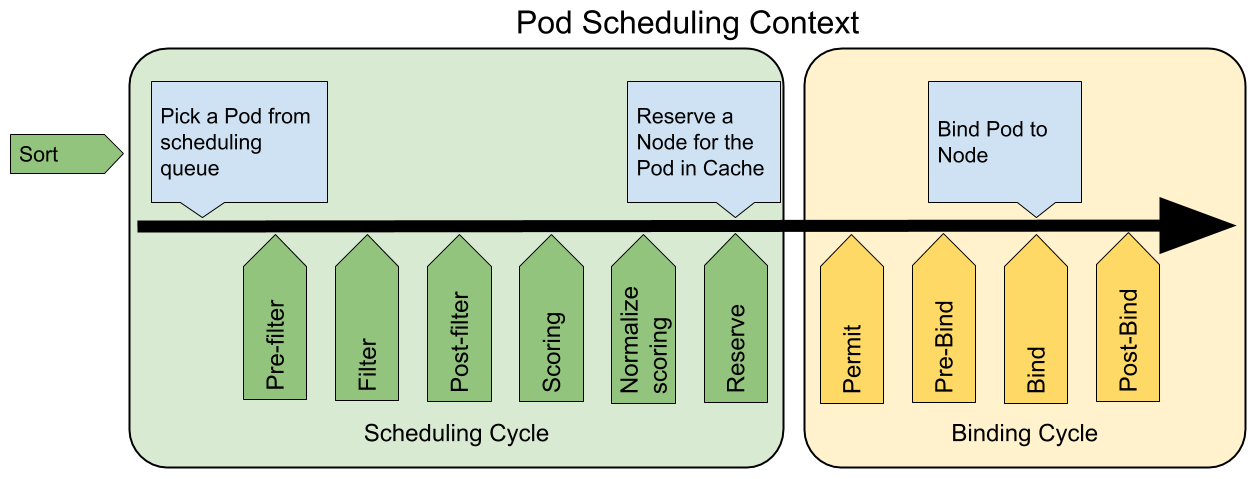
调度周期是串行运行的，而绑定周期可能是同时运行的。

如果确定 pod 不可调度或者存在内部错误，则可以终止调度周期或绑定周期。Pod 将返回队列并重试。

### 扩展点

下图显示了一个 pod 的调度上下文以及调度框架公开的扩展点。在此图片中，“过滤器”等同于“断言”，“评分”相当于“优先级函数”。

一个插件可以在多个扩展点处注册，以执行更复杂或有状态的任务。



#### 调度框架扩展点

###　队列排序

队列排序插件用于对调度队列中的 pod 进行排序。队列排序插件本质上将提供 “less(Pod1, Pod2)” 函数。一次只能启动一个队列插件。

#### 前置过滤

前置过滤插件用于预处理 pod 的相关信息，或者检查集群或 pod 必须满足的某些条件。如果前置过滤插件返回错误，则调度周期将终止。

#### 过滤

过滤插件用于过滤出不能运行该 pod 的节点。对于每个节点，调度器将按照其配置顺序调用这些过滤插件。如果任何过滤插件将节点标记为不可行，则不会为该节点调用剩下的过滤插件。节点可以被同时进行评估。

#### 后置过滤

后置过滤是一个信息性的扩展点。通过过滤阶段的节点列表将调用这些后置过滤。插件将使用这些数据来更新内部的状态或者生成日志/指标。

**注意：**希望执行“预评分”工作的插件应该使用后置过滤扩展点。

#### 评分

评分插件用于对通过过滤阶段的节点进行排名。调度器将为每个节点调用每个评分插件。将有一个定义明确的整数范围，代表最小和最大分数。在[标准化评分](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#标准化评分)阶段之后，调度器将根据配置的插件权重合并所有插件的节点分数。

#### 标准化评分

标准化评分插件用于在调度器计算节点的排名之前修改分数。在此扩展点注册的插件将使用同一插件的[评分](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#评分)结果被调用。每个插件在每个调度周期调用一次。

例如，假设一个 BlinkingLightScorer 插件基于具有的闪烁指示灯数量来对节点进行排名。

**func** ScoreNode(\_ \*v1.pod, n \*v1.Node) (**int**, **error**) {

**return** getBlinkingLightCount(n)

}

然而，最大的闪烁灯个数值可能比 NodeScoreMax 小。要解决这个问题，BlinkingLightScorer 插件还应该注册该扩展点。

**func** NormalizeScores(scores **map**[**string**]**int**) {

highest := 0

**for** \_, score := **range** scores {

highest = max(highest, score)

}

**for** node, score := **range** scores {

scores[node] = score\*NodeScoreMax/highest

}

}

如果任何标准化评分插件返回错误，则调度阶段将终止。

**注意：**希望执行“预保留”工作的插件应该使用标准化评分扩展点。

#### 保留

保留是一个信息性的扩展点。管理运行时状态的插件（也成为“有状态插件”）应该使用此扩展点，以便调度器在节点给指定 pod 预留了资源时能够通知该插件。这是在调度器真正将 pod 绑定到节点之前发生的，并且它存在是为了防止在调度器等待绑定成功时发生竞争情况。

这个是调度周期的最后一步。一旦 pod 处于保留状态，它将在绑定周期结束时触发[不保留](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#不保留)插件（失败时）或 [绑定后](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#绑定后)插件（成功时）。

注意：此概念曾被称为“假设”。

#### 允许

允许插件用于防止或延迟 pod 的绑定。一个允许插件可以做以下三件事之一。

1. **批准**  
   一旦所有允许插件批准 pod 后，该 pod 将被发送以进行绑定。
2. **拒绝**  
   如果任何允许插件拒绝 pod，则该 pod 将被返回到调度队列。这将触发[不保留](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#不保留)插件。
3. **等待**（带有超时）  
   如果一个允许插件返回“等待”结果，则 pod 将保持在允许阶段，直到插件批准它。如果超时发生，**等待**变成**拒绝**，并且 pod 将返回调度队列，从而触发[不保留](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#不保留)插件。

**批准 pod 绑定**

尽管任何插件可以从缓存中访问“等待”状态的 pod 列表并批准它们。我们希望只有允许插件可以批准处于“等待”状态的 预留 pod 的绑定。一旦 pod 被批准了，它将发送到预绑定阶段。

#### 预绑定

预绑定插件用于执行 pod 绑定前所需的任何工作。例如，一个预绑定插件可能需要提供网络卷并且在允许 pod 运行在该节点之前将其挂载到目标节点上。

如果任何预绑定插件返回错误，则 pod 将被[拒绝](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#不保留)并且返回到调度队列中。

#### 绑定

绑定插件用于将 pod 绑定到节点上。直到所有的预绑定插件都完成，绑定插件才会被调用。每个绑定插件按照配置顺序被调用。绑定插件可以选择是否处理指定的 pod。如果绑定插件选择处理 pod，**剩余的绑定插件将被跳过**。

#### 绑定后

这是个信息性的扩展点。绑定后插件在 pod 成功绑定后被调用。这是绑定周期的结尾，可用于清理相关的资源。

#### 不保留

这是个信息性的扩展点。如果 pod 被保留，然后在后面的阶段中被拒绝，则不保留插件将被通知。不保留插件应该清楚保留 pod 的相关状态。

使用此扩展点的插件通常也使用[保留](https://kubernetes.io/zh/docs/concepts/configuration/scheduling-framework/#保留)。

### 插件 API

插件 API 分为两个步骤。首先，插件必须注册并配置，然后才能使用扩展点接口。扩展点接口具有以下形式。

**type** Plugin **interface** {

Name() **string**

}

**type** QueueSortPlugin **interface** {

Plugin

Less(\*v1.pod, \*v1.pod) **bool**

}

**type** PreFilterPlugin **interface** {

Plugin

PreFilter(PluginContext, \*v1.pod) **error**

}

// ...

#### 插件配置

可以在调度器配置中启用插件。另外，默认的插件可以在配置中禁用。在 1.15 版本，调度框架没有默认的插件。

调度器配置也可以包含插件的配置。这些配置在调度器初始化插件时传给插件。配置是一个任意值。接收插件应该解码并处理配置信息。

下面的例子显示一个调度器配置，该配置在 reserve 和 preBind 扩展点启用了一些插件并且禁用了一个插件。它还提供了 foo 插件的配置。

apiVersion: kubescheduler.config.k8s.io/v1alpha1

kind: KubeSchedulerConfiguration

...

plugins:

reserve:

enabled:

- name: foo

- name: bar

disabled:

- name: baz

preBind:

enabled:

- name: foo

disabled:

- name: baz

pluginConfig:

- name: foo

args: *>*

*Arbitrary set of args to plugin foo*

当配置省略扩展点时，将使用该扩展点的默认插件。当存在扩展掉并且配置为 enabled，则 enabled 的插件将和默认插件一同调用。首先调用默认插件，然后以配置中指定的顺序来调用其他已启用的插件。如果希望以不同的顺序来调用默认插件，默认插件必须 disabled，然后以期望的顺序 enabled。

假设在 reserve 扩展点有一个默认的 foo 插件，且添加 bar 插件并且希望在 foo 插件之前执行，我们应该禁用 foo 并且按顺序启用 bar 和 foo。下面的例子显示了实现此目的的配置：

apiVersion: kubescheduler.config.k8s.io/v1alpha1

kind: KubeSchedulerConfiguration

...

plugins:

reserve:

enabled:

- name: bar

- name: foo

disabled:

- name: foo

#### 反馈

# 服务、负载均衡和联网

## Endpoint Slices

**FEATURE STATE:** Kubernetes v1.17 [beta](https://kubernetes.io/zh/docs/concepts/services-networking/)

Endpoint Slices 提供了一种简单的方法来跟踪 Kubernetes 集群中的网络端点（network endpoints）。它们为 Endpoints 提供了一种可伸缩和可拓展的替代方案。

* [**Endpoint Slice 资源**](https://kubernetes.io/zh/docs/concepts/services-networking/#endpointslice-resource)
* [**地址类型**](https://kubernetes.io/zh/docs/concepts/services-networking/#%e5%9c%b0%e5%9d%80%e7%b1%bb%e5%9e%8b)
* [**动机**](https://kubernetes.io/zh/docs/concepts/services-networking/#%e5%8a%a8%e6%9c%ba)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### Endpoint Slice 资源

在 Kubernetes 中，EndpointSlice 包含对一组网络端点的引用。指定选择器后，EndpointSlice 控制器会自动为 Kubernetes 服务创建 EndpointSlice。这些 EndpointSlice 将包含对与服务选择器匹配的所有 Pod 的引用。EndpointSlice 通过唯一的服务和端口组合将网络端点组织在一起。

例如，这里是 Kubernetes服务 example 的示例 EndpointSlice 资源。

apiVersion: discovery.k8s.io/v1beta1

kind: EndpointSlice

metadata:

name: example-abc

labels:

kubernetes.io/service-name: example

addressType: IPv4

ports:

- name: http

protocol: TCP

port: 80

endpoints:

- addresses:

- "10.1.2.3"

conditions:

ready: **true**

hostname: pod-1

topology:

kubernetes.io/hostname: node-1

topology.kubernetes.io/zone: us-west2-a

默认情况下，由 EndpointSlice 控制器管理的 Endpoint Slice 将有不超过 100 个 endpoints。低于此比例时，Endpoint Slices 应与 Endpoints 和服务进行 1：1 映射，并具有相似的性能。

当涉及如何路由内部流量时，Endpoint Slices 可以充当 kube-proxy 的真实来源。启用该功能后，在服务的 endpoints 规模庞大时会有可观的性能提升。

### 地址类型

EndpointSlice 支持三种地址类型：

* IPv4
* IPv6
* FQDN (完全合格的域名)

### 动机

Endpoints API 提供了一种简单明了的方法在 Kubernetes 中跟踪网络端点。不幸的是，随着 Kubernetes 集群与服务的增长，该 API 的局限性变得更加明显。最值得注意的是，这包含了扩展到更多网络端点的挑战。

由于服务的所有网络端点都存储在单个 Endpoints 资源中，因此这些资源可能会变得很大。这影响了 Kubernetes 组件（尤其是主控制平面）的性能，并在 Endpoints 发生更改时导致大量网络流量和处理。Endpoint Slices 可帮助您缓解这些问题并提供可扩展的 附加特性（例如拓扑路由）平台。

### 接下来

* [启用 Endpoint Slices](https://kubernetes.io/docs/tasks/administer-cluster/enabling-endpoint-slices)
* 阅读 [Connecting Applications with Services](https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/)

### 反馈

## Service Topology

**FEATURE STATE:** Kubernetes v1.17 [alpha](https://kubernetes.io/docs/concepts/services-networking/service-topology/)

Service Topology enables a service to route traffic based upon the Node topology of the cluster. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.

* [**Introduction**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#introduction)
* [**Prerequisites**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#prerequisites)
* [**Enable Service Topology**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#enable-service-topology)
* [**Using Service Topology**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#using-service-topology)
* [**Constraints**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#constraints)
* [**What's next**](https://kubernetes.io/docs/concepts/services-networking/service-topology/#what-s-next)

### Introduction

By default, traffic sent to a ClusterIP or NodePort Service may be routed to any backend address for the Service. Since Kubernetes 1.7 it has been possible to route “external” traffic to the Pods running on the Node that received the traffic, but this is not supported for ClusterIP Services, and more complex topologies — such as routing zonally — have not been possible. The Service Topology feature resolves this by allowing the Service creator to define a policy for routing traffic based upon the Node labels for the originating and destination Nodes.

By using Node label matching between the source and destination, the operator may designate groups of Nodes that are “closer” and “farther” from one another, using whatever metric makes sense for that operator’s requirements. For many operators in public clouds, for example, there is a preference to keep service traffic within the same zone, because interzonal traffic has a cost associated with it, while intrazonal traffic does not. Other common needs include being able to route traffic to a local Pod managed by a DaemonSet, or keeping traffic to Nodes connected to the same top-of-rack switch for the lowest latency.

### Prerequisites

The following prerequisites are needed in order to enable topology aware service routing:

* Kubernetes 1.17 or later
* Kube-proxy running in iptables mode or IPVS mode
* Enable [Endpoint Slices](https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/)

### Enable Service Topology

To enable service topology, enable the ServiceTopology feature gate for kube-apiserver and kube-proxy:

--feature-gates="ServiceTopology=true"

### Using Service Topology

If your cluster has Service Topology enabled, you can control Service traffic routing by specifying the topologyKeys field on the Service spec. This field is a preference-order list of Node labels which will be used to sort endpoints when accessing this Service. Traffic will be directed to a Node whose value for the first label matches the originating Node’s value for that label. If there is no backend for the Service on a matching Node, then the second label will be considered, and so forth, until no labels remain.

If no match is found, the traffic will be rejected, just as if there were no backends for the Service at all. That is, endpoints are chosen based on the first topology key with available backends. If this field is specified and all entries have no backends that match the topology of the client, the service has no backends for that client and connections should fail. The special value "\*" may be used to mean “any topology”. This catch-all value, if used, only makes sense as the last value in the list.

If topologyKeys is not specified or empty, no topology constraints will be applied.

Consider a cluster with Nodes that are labeled with their hostname, zone name, and region name. Then you can set the topologyKeys values of a service to direct traffic as follows.

* Only to endpoints on the same node, failing if no endpoint exists on the node: ["kubernetes.io/hostname"].
* Preferentially to endpoints on the same node, falling back to endpoints in the same zone, followed by the same region, and failing otherwise: ["kubernetes.io/hostname", "topology.kubernetes.io/zone", "topology.kubernetes.io/region"]. This may be useful, for example, in cases where data locality is critical.
* Preferentially to the same zone, but fallback on any available endpoint if none are available within this zone: ["topology.kubernetes.io/zone", "\*"].

### Constraints

* Service topology is not compatible with externalTrafficPolicy=Local, and therefore a Service cannot use both of these features. It is possible to use both features in the same cluster on different Services, just not on the same Service.
* Valid topology keys are currently limited to kubernetes.io/hostname, topology.kubernetes.io/zone, and topology.kubernetes.io/region, but will be generalized to other node labels in the future.
* Topology keys must be valid label keys and at most 16 keys may be specified.
* The catch-all value, "\*", must be the last value in the topology keys, if it is used.

### What's next

* Read about [enabling Service Topology](https://kubernetes.io/docs/tasks/administer-cluster/enabling-service-topology)
* Read [Connecting Applications with Services](https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/)

### Feedback

## Services

将运行在一组 [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 上的应用程序公开为网络服务的抽象方法。

使用Kubernetes，您无需修改应用程序即可使用不熟悉的服务发现机制。 Kubernetes为Pods提供自己的IP地址和一组Pod的单个DNS名称，并且可以在它们之间进行负载平衡。

* [**动机**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e5%8a%a8%e6%9c%ba)
* [**Service 资源**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-resource)
* [**定义 Service**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e5%ae%9a%e4%b9%89-service)
* [**VIP 和 Service 代理**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#vip-%e5%92%8c-service-%e4%bb%a3%e7%90%86)
* [**多端口 Service**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e5%a4%9a%e7%ab%af%e5%8f%a3-service)
* [**选择自己的 IP 地址**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e9%80%89%e6%8b%a9%e8%87%aa%e5%b7%b1%e7%9a%84-ip-%e5%9c%b0%e5%9d%80)
* [**服务发现**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e6%9c%8d%e5%8a%a1%e5%8f%91%e7%8e%b0)
* [**Headless Services**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#headless-services)
* [**发布服务 —— 服务类型**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#publishing-services-service-types)
* [**不足之处**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e4%b8%8d%e8%b6%b3%e4%b9%8b%e5%a4%84)
* [**虚拟IP实施**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#the-gory-details-of-virtual-ips)
* [**API Object**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#api-object)
* [**Supported protocols**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#protocol-support)
* [**未来工作**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e6%9c%aa%e6%9d%a5%e5%b7%a5%e4%bd%9c)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/service/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 动机

Kubernetes [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 是有生命周期的。他们可以被创建，而且销毁不会再启动。 如果您使用 [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) 来运行您的应用程序，则它可以动态创建和销毁 Pod。

每个 Pod 都有自己的 IP 地址，但是在 Deployment 中，在同一时刻运行的 Pod 集合可能与稍后运行该应用程序的 Pod 集合不同。

这导致了一个问题： 如果一组 Pod（称为“后端”）为群集内的其他 Pod（称为“前端”）提供功能，那么前端如何找出并跟踪要连接的 IP 地址，以便前端可以使用工作量的后端部分？

进入 \_Services\_。

### Service 资源

Kubernetes Service 定义了这样一种抽象：逻辑上的一组 Pod，一种可以访问它们的策略 —— 通常称为微服务。 这一组 Pod 能够被 Service 访问到，通常是通过 [selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) （查看[下面](https://kubernetes.io/zh/docs/concepts/services-networking/service/#services-without-selectors)了解，为什么你可能需要没有 selector 的 Service）实现的。

举个例子，考虑一个图片处理 backend，它运行了3个副本。这些副本是可互换的 —— frontend 不需要关心它们调用了哪个 backend 副本。 然而组成这一组 backend 程序的 Pod 实际上可能会发生变化，frontend 客户端不应该也没必要知道，而且也不需要跟踪这一组 backend 的状态。 Service 定义的抽象能够解耦这种关联。

#### 云原生服务发现

如果您能够在应用程序中使用 Kubernetes 接口进行服务发现，则可以查询 [API server](https://kubernetes.io/docs/reference/generated/kube-apiserver/) 的 endpoint 资源，只要服务中的Pod集合发生更改，端点就会更新。

对于非本机应用程序，Kubernetes提供了在应用程序和后端Pod之间放置网络端口或负载平衡器的方法。

### 定义 Service

一个 Service 在 Kubernetes 中是一个 REST 对象，和 Pod 类似。 像所有的 REST 对象一样， Service 定义可以基于 POST 方式，请求 API server 创建新的实例。

例如，假定有一组 Pod，它们对外暴露了 9376 端口，同时还被打上 app=MyApp 标签。

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

上述配置创建一个名称为 “my-service” 的 Service 对象，它会将请求代理到使用 TCP 端口 9376，并且具有标签 "app=MyApp" 的 Pod 上。 Kubernetes 为该服务分配一个 IP 地址（有时称为 “集群IP” ），该 IP 地址由服务代理使用。 (请参见下面的 [虚拟 IP 和服务代理](https://kubernetes.io/zh/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies)). 服务选择器的控制器不断扫描与其选择器匹配的 Pod，然后将所有更新发布到也称为 “my-service” 的Endpoint对象。

**注意：**

需要注意的是， Service 能够将一个接收 port 映射到任意的 targetPort。 默认情况下，targetPort 将被设置为与 port 字段相同的值。

Pod中的端口定义具有名称字段，您可以在服务的 targetTarget 属性中引用这些名称。 即使服务中使用单个配置的名称混合使用 Pod，并且通过不同的端口号提供相同的网络协议，此功能也可以使用。 这为部署和发展服务提供了很大的灵活性。 例如，您可以更改Pods在新版本的后端软件中公开的端口号，而不会破坏客户端。

服务的默认协议是TCP；默认协议是TCP。 您还可以使用任何其他 [受支持的协议](https://kubernetes.io/zh/docs/concepts/services-networking/service/#protocol-support)。

由于许多服务需要公开多个端口，因此 Kubernetes 在服务对象上支持多个端口定义。 每个端口定义可以具有相同的 protocol，也可以具有不同的协议。

#### 没有 selector 的 Service

服务最常见的是抽象化对 Kubernetes Pod 的访问，但是它们也可以抽象化其他种类的后端。 实例:

* 希望在生产环境中使用外部的数据库集群，但测试环境使用自己的数据库。
* 希望服务指向另一个 [命名空间](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces) 中或其它集群中的服务。
* 您正在将工作负载迁移到 Kubernetes。 在评估该方法时，您仅在 Kubernetes 中运行一部分后端。

在任何这些场景中，都能够定义没有 selector 的 Service。 实例:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

由于此服务没有选择器，因此 不会 自动创建相应的 Endpoint 对象。 您可以通过手动添加 Endpoint 对象，将服务手动映射到运行该服务的网络地址和端口：

apiVersion: v1

kind: Endpoints

metadata:

name: my-service

subsets:

- addresses:

- ip: 192.0.2.42

ports:

- port: 9376

**注意：**

端点 IPs 必须不可以 : 环回( IPv4 的 127.0.0.0/8 , IPv6 的 ::1/128 ）或本地链接（IPv4 的 169.254.0.0/16 和 224.0.0.0/24，IPv6 的 fe80::/64)。 端点 IP 地址不能是其他 Kubernetes Services 的群集 IP，因为 [kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) 不支持将虚拟 IP 作为目标。

访问没有 selector 的 Service，与有 selector 的 Service 的原理相同。 请求将被路由到用户定义的 Endpoint， YAML中为: 192.0.2.42:9376 (TCP)。

ExternalName Service 是 Service 的特例，它没有 selector，也没有使用 DNS 名称代替。 有关更多信息，请参阅本文档后面的[ExternalName](https://kubernetes.io/zh/docs/concepts/services-networking/service/#externalname)。

#### Endpoint 切片

**FEATURE STATE:** Kubernetes v1.16 [alpha](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

Endpoint 切片是一种 API 资源，可以为 Endpoint 提供更可扩展的替代方案。 尽管从概念上讲与 Endpoint 非常相似，但 Endpoint 切片允许跨多个资源分布网络端点。 默认情况下，一旦到达100个 Endpoint，该 Endpoint 切片将被视为“已满”，届时将创建其他 Endpoint 切片来存储任何其他 Endpoint。

Endpoint 切片提供了附加的属性和功能，这些属性和功能在 [Endpoint 切片](https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/)中进行了详细描述。

### VIP 和 Service 代理

在 Kubernetes 集群中，每个 Node 运行一个 kube-proxy 进程。kube-proxy 负责为 Service 实现了一种 VIP（虚拟 IP）的形式，而不是 [ExternalName](https://kubernetes.io/zh/docs/concepts/services-networking/service/#externalname) 的形式。

#### 为什么不使用 DNS 轮询？

时不时会有人问道，就是为什么 Kubernetes 依赖代理将入站流量转发到后端。 那其他方法呢？ 例如，是否可以配置具有多个A值（或IPv6为AAAA）的DNS记录，并依靠轮询名称解析？

使用服务代理有以下几个原因：

* DNS 实现的历史由来已久，它不遵守记录 TTL，并且在名称查找结果到期后对其进行缓存。
* 有些应用程序仅执行一次 DNS 查找，并无限期地缓存结果。
* 即使应用和库进行了适当的重新解析，DNS 记录上的 TTL 值低或为零也可能会给 DNS 带来高负载，从而使管理变得困难。

#### 版本兼容性

从Kubernetes v1.0开始，您已经可以使用 [用户空间代理模式](https://kubernetes.io/zh/docs/concepts/services-networking/service/#proxy-mode-userspace)。 Kubernetes v1.1添加了 iptables 模式代理，在 Kubernetes v1.2 中，kube-proxy 的 iptables 模式成为默认设置。 Kubernetes v1.8添加了 ipvs 代理模式。

#### userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes master 对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会在本地 Node 上打开一个端口（随机选择）。 任何连接到“代理端口”的请求，都会被代理到 Service 的backend Pods 中的某个上面（如 Endpoints 所报告的一样）。 使用哪个 backend Pod，是 kube-proxy 基于 SessionAffinity 来确定的。

最后，它安装 iptables 规则，捕获到达该 Service 的 clusterIP（是虚拟 IP）和 Port 的请求，并重定向到代理端口，代理端口再代理请求到 backend Pod。

默认情况下，用户空间模式下的kube-proxy通过循环算法选择后端。

默认的策略是，通过 round-robin 算法来选择 backend Pod。

#### iptables 代理模式

这种模式，kube-proxy 会监视 Kubernetes 控制节点对 Service 对象和 Endpoints 对象的添加和移除。 对每个 Service，它会安装 iptables 规则，从而捕获到达该 Service 的 clusterIP 和端口的请求，进而将请求重定向到 Service 的一组 backend 中的某个上面。 对于每个 Endpoints 对象，它也会安装 iptables 规则，这个规则会选择一个 backend 组合。

默认的策略是，kube-proxy 在 iptables 模式下随机选择一个 backend。

使用 iptables 处理流量具有较低的系统开销，因为流量由 Linux netfilter 处理，而无需在用户空间和内核空间之间切换。 这种方法也可能更可靠。

如果 kube-proxy 在 iptable s模式下运行，并且所选的第一个 Pod 没有响应，则连接失败。 这与用户空间模式不同：在这种情况下，kube-proxy 将检测到与第一个 Pod 的连接已失败，并会自动使用其他后端 Pod 重试。

您可以使用 Pod [readiness 探测器](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes) 验证后端 Pod 可以正常工作，以便 iptables 模式下的 kube-proxy 仅看到测试正常的后端。 这样做意味着您避免将流量通过 kube-proxy 发送到已知已失败的Pod。

#### IPVS 代理模式

**FEATURE STATE:** Kubernetes v1.11 [稳定](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

在 ipvs 模式下，kube-proxy监视Kubernetes服务和端点，调用 netlink 接口相应地创建 IPVS 规则， 并定期将 IPVS 规则与 Kubernetes 服务和端点同步。 该控制循环可确保　IPVS　状态与所需状态匹配。 访问服务时，IPVS　将流量定向到后端Pod之一。

IPVS代理模式基于类似于 iptables 模式的 netfilter 挂钩函数，但是使用哈希表作为基础数据结构，并且在内核空间中工作。 这意味着，与 iptables 模式下的 kube-proxy 相比，IPVS 模式下的 kube-proxy 重定向通信的延迟要短，并且在同步代理规则时具有更好的性能。与其他代理模式相比，IPVS 模式还支持更高的网络流量吞吐量。

IPVS提供了更多选项来平衡后端Pod的流量。 这些是：

* rr: round-robin
* lc: least connection (smallest number of open connections)
* dh: destination hashing
* sh: source hashing
* sed: shortest expected delay
* nq: never queue

**注意：**

要在 IPVS 模式下运行 kube-proxy，必须在启动 kube-proxy 之前使 IPVS Linux 在节点上可用。

当 kube-proxy 以 IPVS 代理模式启动时，它将验证 IPVS 内核模块是否可用。 如果未检测到 IPVS 内核模块，则 kube-proxy 将退回到以 iptables 代理模式运行。

在这些代理模型中，绑定到服务IP的流量：在客户端不了解Kubernetes或服务或Pod的任何信息的情况下，将Port代理到适当的后端。 如果要确保每次都将来自特定客户端的连接传递到同一Pod，则可以通过将 service.spec.sessionAffinity 设置为 “ClientIP” (默认值是 “None”)，来基于客户端的IP地址选择会话关联。

您还可以通过适当设置 service.spec.sessionAffinityConfig.clientIP.timeoutSeconds 来设置最大会话停留时间。 （默认值为 10800 秒，即 3 小时）。

### 多端口 Service

对于某些服务，您需要公开多个端口。 Kubernetes允许您在Service对象上配置多个端口定义。 为服务使用多个端口时，必须提供所有端口名称，以使它们无歧义。 例如：

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- name: http

protocol: TCP

port: 80

targetPort: 9376

- name: https

protocol: TCP

port: 443

targetPort: 9377

**注意：**

与一般的Kubernetes名称一样，端口名称只能包含 小写字母数字字符 和 -。 端口名称还必须以字母数字字符开头和结尾。

例如，名称 123\_abc 和 web 有效，但是 123\_abc 和 -web 无效。

### 选择自己的 IP 地址

在 Service 创建的请求中，可以通过设置 spec.clusterIP 字段来指定自己的集群 IP 地址。 比如，希望替换一个已经已存在的 DNS 条目，或者遗留系统已经配置了一个固定的 IP 且很难重新配置。

用户选择的 IP 地址必须合法，并且这个 IP 地址在 service-cluster-ip-range CIDR 范围内，这对 API Server 来说是通过一个标识来指定的。 如果 IP 地址不合法，API Server 会返回 HTTP 状态码 422，表示值不合法。

### 服务发现

Kubernetes 支持2种基本的服务发现模式 —— 环境变量和 DNS。

#### 环境变量

当 Pod 运行在 Node 上，kubelet 会为每个活跃的 Service 添加一组环境变量。 它同时支持 [Docker links兼容](https://docs.docker.com/userguide/dockerlinks/) 变量（查看 [makeLinkVariables](http://releases.k8s.io/master/pkg/kubelet/envvars/envvars.go#L49)）、简单的 {SVCNAME}\_SERVICE\_HOST 和 {SVCNAME}\_SERVICE\_PORT 变量，这里 Service 的名称需大写，横线被转换成下划线。

举个例子，一个名称为 "redis-master" 的 Service 暴露了 TCP 端口 6379，同时给它分配了 Cluster IP 地址 10.0.0.11，这个 Service 生成了如下环境变量：

REDIS\_MASTER\_SERVICE\_HOST=10.0.0.11

REDIS\_MASTER\_SERVICE\_PORT=6379

REDIS\_MASTER\_PORT=tcp://10.0.0.11:6379

REDIS\_MASTER\_PORT\_6379\_TCP=tcp://10.0.0.11:6379

REDIS\_MASTER\_PORT\_6379\_TCP\_PROTO=tcp

REDIS\_MASTER\_PORT\_6379\_TCP\_PORT=6379

REDIS\_MASTER\_PORT\_6379\_TCP\_ADDR=10.0.0.11

**注意：**

当您具有需要访问服务的Pod时，并且您正在使用环境变量方法将端口和群集IP发布到客户端Pod时，必须在客户端Pod出现 之前 创建服务。 否则，这些客户端Pod将不会设定其环境变量。

如果仅使用DNS查找服务的群集IP，则无需担心此设定问题。

#### DNS

您可以（几乎总是应该）使用[附加组件](https://kubernetes.io/docs/concepts/cluster-administration/addons/)为Kubernetes集群设置DNS服务。

支持群集的DNS服务器（例如CoreDNS）监视 Kubernetes API 中的新服务，并为每个服务创建一组 DNS 记录。 如果在整个群集中都启用了 DNS，则所有 Pod 都应该能够通过其 DNS 名称自动解析服务。

例如，如果您在 Kubernetes 命名空间 "my-ns" 中有一个名为 "my-service" 的服务， 则控制平面和DNS服务共同为 "my-service.my-ns" 创建 DNS 记录。 "my-ns" 命名空间中的Pod应该能够通过简单地对 my-service 进行名称查找来找到它（ "my-service.my-ns" 也可以工作）。

其他命名空间中的Pod必须将名称限定为 my-service.my-ns 。 这些名称将解析为为服务分配的群集IP。

Kubernetes 还支持命名端口的 DNS SRV（服务）记录。 如果 "my-service.my-ns" 服务具有名为 "http"　的端口，且协议设置为TCP， 则可以对 \_http.\_tcp.my-service.my-ns 执行DNS SRV查询查询以发现该端口号, "http"以及IP地址。

Kubernetes DNS 服务器是唯一的一种能够访问 ExternalName 类型的 Service 的方式。 更多关于 ExternalName 信息可以查看[DNS Pod 和 Service](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)。

### Headless Services

有时不需要或不想要负载均衡，以及单独的 Service IP。 遇到这种情况，可以通过指定 Cluster IP（spec.clusterIP）的值为 "None" 来创建 Headless Service。

您可以使用 headless Service 与其他服务发现机制进行接口，而不必与 Kubernetes 的实现捆绑在一起。

对这 headless Service 并不会分配 Cluster IP，kube-proxy 不会处理它们，而且平台也不会为它们进行负载均衡和路由。 DNS 如何实现自动配置，依赖于 Service 是否定义了 selector。

#### 配置 Selector

对定义了 selector 的 Headless Service，Endpoint 控制器在 API 中创建了 Endpoints 记录，并且修改 DNS 配置返回 A 记录（地址），通过这个地址直接到达 Service 的后端 Pod 上。

#### 不配置 Selector

对没有定义 selector 的 Headless Service，Endpoint 控制器不会创建 Endpoints 记录。 然而 DNS 系统会查找和配置，无论是：

* ExternalName 类型 Service 的 CNAME 记录
* 记录：与 Service 共享一个名称的任何 Endpoints，以及所有其它类型

### 发布服务 —— 服务类型

对一些应用（如 Frontend）的某些部分，可能希望通过外部Kubernetes 集群外部IP 地址暴露 Service。

Kubernetes ServiceTypes 允许指定一个需要的类型的 Service，默认是 ClusterIP 类型。

Type 的取值以及行为如下： \* ClusterIP：通过集群的内部 IP 暴露服务，选择该值，服务只能够在集群内部可以访问，这也是默认的 ServiceType。 \* [NodePort](https://kubernetes.io/zh/docs/concepts/services-networking/service/#nodeport)：通过每个 Node 上的 IP 和静态端口（NodePort）暴露服务。NodePort 服务会路由到 ClusterIP 服务，这个 ClusterIP 服务会自动创建。通过请求 <NodeIP>:<NodePort>，可以从集群的外部访问一个 NodePort 服务。 \* [LoadBalancer](https://kubernetes.io/zh/docs/concepts/services-networking/service/#loadbalancer)：使用云提供商的负载局衡器，可以向外部暴露服务。外部的负载均衡器可以路由到 NodePort 服务和 ClusterIP 服务。 \* [ExternalName](https://kubernetes.io/zh/docs/concepts/services-networking/service/#externalname)：通过返回 CNAME 和它的值，可以将服务映射到 externalName 字段的内容（例如， foo.bar.example.com）。 没有任何类型代理被创建。

**注意：** 您需要 CoreDNS 1.7 或更高版本才能使用 ExternalName 类型。

您也可以使用 [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) 来暴露自己的服务。 Ingress 不是服务类型，但它充当集群的入口点。 它可以将路由规则整合到一个资源中，因为它可以在同一IP地址下公开多个服务。

#### NodePort 类型

如果将 type 字段设置为 NodePort，则 Kubernetes 控制平面将在 --service-node-port-range 标志指定的范围内分配端口（默认值：30000-32767）。 每个节点将那个端口（每个节点上的相同端口号）代理到您的服务中。 您的服务在其 .spec.ports[\*].nodePort 字段中要求分配的端口。

如果您想指定特定的IP代理端口，则可以将 kube-proxy 中的 --nodeport-addresses 标志设置为特定的IP块。从Kubernetes v1.10开始支持此功能。

该标志采用逗号分隔的IP块列表（例如10.0.0.0/8、192.0.2.0/25）来指定 kube-proxy 应该认为是此节点本地的IP地址范围。

例如，如果您使用 --nodeport-addresses=127.0.0.0/8 标志启动 kube-proxy，则 kube-proxy 仅选择 NodePort Services 的环回接口。 --nodeport-addresses 的默认值是一个空列表。 这意味着 kube-proxy 应该考虑 NodePort 的所有可用网络接口。 （这也与早期的Kubernetes版本兼容）。

如果需要特定的端口号，则可以在 nodePort 字段中指定一个值。 控制平面将为您分配该端口或向API报告事务失败。 这意味着您需要自己注意可能发生的端口冲突。 您还必须使用有效的端口号，该端口号在配置用于NodePort的范围内。

使用 NodePort 可以让您自由设置自己的负载平衡解决方案，配置 Kubernetes 不完全支持的环境，甚至直接暴露一个或多个节点的IP。

需要注意的是，Service 能够通过 <NodeIP>:spec.ports[\*].nodePort 和 spec.clusterIp:spec.ports[\*].port 而对外可见。

#### LoadBalancer 类型

使用支持外部负载均衡器的云提供商的服务，设置 type 的值为 "LoadBalancer"，将为 Service 提供负载均衡器。 负载均衡器是异步创建的，关于被提供的负载均衡器的信息将会通过 Service 的 status.loadBalancer 字段被发布出去。 实例:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

clusterIP: 10.0.171.239

loadBalancerIP: 78.11.24.19

type: LoadBalancer

status:

loadBalancer:

ingress:

- ip: 146.148.47.155

来自外部负载均衡器的流量将直接打到 backend Pod 上，不过实际它们是如何工作的，这要依赖于云提供商。

在这些情况下，将根据用户设置的 loadBalancerIP 来创建负载均衡器。 某些云提供商允许设置 loadBalancerIP。如果没有设置 loadBalancerIP，将会给负载均衡器指派一个临时 IP。 如果设置了 loadBalancerIP，但云提供商并不支持这种特性，那么设置的 loadBalancerIP 值将会被忽略掉。

**注意：**

如果您使用的是 SCTP，请参阅下面有关 LoadBalancer 服务类型的 [caveat](https://kubernetes.io/zh/docs/concepts/services-networking/service/#caveat-sctp-loadbalancer-service-type)。

**注意：**

在 **Azure** 上，如果要使用用户指定的公共类型 loadBalancerIP ，则首先需要创建静态类型的公共IP地址资源。 此公共IP地址资源应与群集中其他自动创建的资源位于同一资源组中。 例如，MC\_myResourceGroup\_myAKSCluster\_eastus。

将分配的IP地址指定为loadBalancerIP。 确保您已更新云提供程序配置文件中的securityGroupName。 有关对 CreatingLoadBalancerFailed 权限问题进行故障排除的信息， 请参阅 [与Azure Kubernetes服务（AKS）负载平衡器一起使用静态IP地址](https://docs.microsoft.com/en-us/azure/aks/static-ip)或[通过高级网络在AKS群集上创建LoadBalancerFailed](https://github.com/Azure/AKS/issues/357)。

##### 内部负载均衡器

在混合环境中，有时有必要在同一(虚拟)网络地址块内路由来自服务的流量。

在水平分割 DNS 环境中，您需要两个服务才能将内部和外部流量都路由到您的 endpoints。 您可以通过向服务添加以下注释之一来实现此目的。 要添加的注释取决于您使用的云服务提供商。

* [Default](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-0)
* [GCP](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-1)
* [AWS](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-2)
* [Azure](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-3)
* [OpenStack](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-4)
* [Baidu Cloud](https://kubernetes.io/zh/docs/concepts/services-networking/service/#service-tabs-5)

选择一个标签

#### AWS TLS 支持

为了对在AWS上运行的集群提供部分TLS / SSL支持，您可以向 LoadBalancer 服务添加三个注释：

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-ssl-cert: arn:aws:acm:us-east-1:123456789012:certificate/12345678-1234-1234-1234-123456789012

第一个指定要使用的证书的ARN。 它可以是已上载到 IAM 的第三方颁发者的证书，也可以是在 AWS Certificate Manager 中创建的证书。

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-backend-protocol: (https|http|ssl|tcp)

第二个注释指定 Pod 使用哪种协议。 对于 HTTPS 和 SSL，ELB 希望 Pod 使用证书通过加密连接对自己进行身份验证。

HTTP 和 HTTPS 选择第7层代理：ELB 终止与用户的连接，解析标头，并在转发请求时向 X-Forwarded-For 标头注入用户的 IP 地址（Pod 仅在连接的另一端看到 ELB 的 IP 地址）。

TCP 和 SSL 选择第4层代理：ELB 转发流量而不修改报头。

在某些端口处于安全状态而其他端口未加密的混合使用环境中，可以使用以下注释：

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-backend-protocol: http

service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443,8443"

从Kubernetes v1.9起可以使用 [预定义的 AWS SSL 策略](http://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-security-policy-table.html) 为您的服务使用HTTPS或SSL侦听器。 要查看可以使用哪些策略，可以使用 aws 命令行工具：

aws elb describe-load-balancer-policies --query 'PolicyDescriptions[].PolicyName'

然后，您可以使用 “service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy“ 注解; 例如：

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-ssl-negotiation-policy: "ELBSecurityPolicy-TLS-1-2-2017-01"

##### AWS上的PROXY协议支持

为了支持在AWS上运行的集群，启用 [PROXY协议](https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt), 您可以使用以下服务注释：

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "\*"

从1.3.0版开始，此注释的使用适用于 ELB 代理的所有端口，并且不能进行其他配置。

##### AWS上的ELB访问日志

有几个注释可用于管理AWS上ELB服务的访问日志。

注释 service.beta.kubernetes.io/aws-load-balancer-access-log-enabled 控制是否启用访问日志。

注解 service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval 控制发布访问日志的时间间隔（以分钟为单位）。 您可以指定5分钟或60分钟的间隔。

注释 service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name 控制存储负载均衡器访问日志的Amazon S3存储桶的名称。

注释 service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix 指定为Amazon S3存储桶创建的逻辑层次结构。

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-access-log-enabled: "true"

*# Specifies whether access logs are enabled for the load balancer*

service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval: "60"

*# The interval for publishing the access logs. You can specify an interval of either 5 or 60 (minutes).*

service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name: "my-bucket"

*# The name of the Amazon S3 bucket where the access logs are stored*

service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix: "my-bucket-prefix/prod"

*# The logical hierarchy you created for your Amazon S3 bucket, for example `my-bucket-prefix/prod`*

##### AWS上的连接排空

可以将注释 service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled 设置为 "true" 的值来管理 ELB 的连接消耗。 注释 service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout 也可以用于设置最大时间（以秒为单位），以保持现有连接在注销实例之前保持打开状态。

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled: "true"

service.beta.kubernetes.io/aws-load-balancer-connection-draining-timeout: "60"

##### 其他ELB注释

还有其他一些注释，用于管理经典弹性负载均衡器，如下所述。

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-connection-idle-timeout: "60"

*# The time, in seconds, that the connection is allowed to be idle (no data has been sent over the connection) before it is closed by the load balancer*

service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled: "true"

*# Specifies whether cross-zone load balancing is enabled for the load balancer*

service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags: "environment=prod,owner=devops"

*# A comma-separated list of key-value pairs which will be recorded as*

*# additional tags in the ELB.*

service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold: ""

*# The number of successive successful health checks required for a backend to*

*# be considered healthy for traffic. Defaults to 2, must be between 2 and 10*

service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold: "3"

*# The number of unsuccessful health checks required for a backend to be*

*# considered unhealthy for traffic. Defaults to 6, must be between 2 and 10*

service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "20"

*# The approximate interval, in seconds, between health checks of an*

*# individual instance. Defaults to 10, must be between 5 and 300*

service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout: "5"

*# The amount of time, in seconds, during which no response means a failed*

*# health check. This value must be less than the service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval*

*# value. Defaults to 5, must be between 2 and 60*

service.beta.kubernetes.io/aws-load-balancer-extra-security-groups: "sg-53fae93f,sg-42efd82e"

*# A list of additional security groups to be added to the ELB*

**FEATURE STATE:** Kubernetes v1.15 [beta](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

metadata:

name: my-service

annotations:

service.beta.kubernetes.io/aws-load-balancer-type: "nlb"

**注意：**

NLB 仅适用于某些实例类。 有关受支持的实例类型的列表，请参见 Elastic Load Balancing 上的 [AWS文档](http://docs.aws.amazon.com/elasticloadbalancing/latest/network/target-group-register-targets.html#register-deregister-targets)。

与经典弹性负载平衡器不同，网络负载平衡器（NLB）将客户端的 IP 地址转发到该节点。 如果服务的 .spec.externalTrafficPolicy 设置为 Cluster ，则客户端的IP地址不会传达到终端 Pod。

通过将 .spec.externalTrafficPolicy 设置为 Local，客户端IP地址将传播到终端 Pod，但这可能导致流量分配不均。 没有针对特定 LoadBalancer 服务的任何 Pod 的节点将无法通过自动分配的 .spec.healthCheckNodePort 进行 NLB 目标组的运行状况检查，并且不会收到任何流量。

为了获得平均流量，请使用DaemonSet或指定 [pod anti-affinity](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity)使其不在同一节点上。

您还可以将NLB服务与 [内部负载平衡器](https://kubernetes.io/docs/concepts/services-networking/service/#internal-load-balancer)批注一起使用。

为了使客户端流量能够到达 NLB 后面的实例，使用以下 IP 规则修改了节点安全组：

| Rule | Protocol | Port(s) | IpRange(s) | IpRange Description |
| --- | --- | --- | --- | --- |
| Health Check | TCP | NodePort(s) (.spec.healthCheckNodePort for .spec.externalTrafficPolicy = Local) | VPC CIDR | kubernetes.io/rule/nlb/health=<loadBalancerName> |
| Client Traffic | TCP | NodePort(s) | .spec.loadBalancerSourceRanges (defaults to 0.0.0.0/0) | kubernetes.io/rule/nlb/client=<loadBalancerName> |
| MTU Discovery | ICMP | 3,4 | .spec.loadBalancerSourceRanges (defaults to 0.0.0.0/0) | kubernetes.io/rule/nlb/mtu=<loadBalancerName> |

为了限制哪些客户端IP可以访问网络负载平衡器，请指定 loadBalancerSourceRanges。

spec:

loadBalancerSourceRanges:

- "143.231.0.0/16"

**注意：**

如果未设置 .spec.loadBalancerSourceRanges ，则 Kubernetes 允许从 0.0.0.0/0 到节点安全组的流量。 如果节点具有公共 IP 地址，请注意，非 NLB 流量也可以到达那些修改后的安全组中的所有实例。

#### 类型ExternalName

类型为 ExternalName 的服务将服务映射到 DNS 名称，而不是典型的选择器，例如 my-service 或者 cassandra。 您可以使用 spec.externalName 参数指定这些服务。

例如，以下 Service 定义将 prod 名称空间中的 my-service 服务映射到 my.database.example.com：

apiVersion: v1

kind: Service

metadata:

name: my-service

namespace: prod

spec:

type: ExternalName

externalName: my.database.example.com

**注意：**

ExternalName 接受 IPv4 地址字符串，但作为包含数字的 DNS 名称，而不是 IP 地址。 类似于 IPv4 地址的外部名称不能由 CoreDNS 或 ingress-nginx 解析，因为外部名称旨在指定规范的 DNS 名称。 要对 IP 地址进行硬编码，请考虑使用 [headless Services](https://kubernetes.io/zh/docs/concepts/services-networking/service/#headless-services)。

当查找主机 my-service.prod.svc.cluster.local 时，群集DNS服务返回 CNAME 记录，其值为 my.database.example.com。 访问 my-service 的方式与其他服务的方式相同，但主要区别在于重定向发生在 DNS 级别，而不是通过代理或转发。 如果以后您决定将数据库移到群集中，则可以启动其 Pod，添加适当的选择器或端点以及更改服务的类型。

**注意：**

本部分感谢 [Alen Komljen](https://akomljen.com/)的 [Kubernetes Tips - Part1](https://akomljen.com/kubernetes-tips-part-1/) 博客文章。

#### 外部 IP

如果外部的 IP 路由到集群中一个或多个 Node 上，Kubernetes Service 会被暴露给这些 externalIPs。 通过外部 IP（作为目的 IP 地址）进入到集群，打到 Service 的端口上的流量，将会被路由到 Service 的 Endpoint 上。 externalIPs 不会被 Kubernetes 管理，它属于集群管理员的职责范畴。

根据 Service 的规定，externalIPs 可以同任意的 ServiceType 来一起指定。 在上面的例子中，my-service 可以在 “80.11.12.10:80”(externalIP:port) 上被客户端访问。

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- name: http

protocol: TCP

port: 80

targetPort: 9376

externalIPs:

- 80.11.12.10

### 不足之处

为 VIP 使用 userspace 代理，将只适合小型到中型规模的集群，不能够扩展到上千 Service 的大型集群。 查看 [最初设计方案](http://issue.k8s.io/1107) 获取更多细节。

使用 userspace 代理，隐藏了访问 Service 的数据包的源 IP 地址。 这使得一些类型的防火墙无法起作用。 iptables 代理不会隐藏 Kubernetes 集群内部的 IP 地址，但却要求客户端请求必须通过一个负载均衡器或 Node 端口。

Type 字段支持嵌套功能 —— 每一层需要添加到上一层里面。 不会严格要求所有云提供商（例如，GCE 就没必要为了使一个 LoadBalancer 能工作而分配一个 NodePort，但是 AWS 需要 ），但当前 API 是强制要求的。

### 虚拟IP实施

对很多想使用 Service 的人来说，前面的信息应该足够了。 然而，有很多内部原理性的内容，还是值去理解的。

#### 避免冲突

Kubernetes 最主要的哲学之一，是用户不应该暴露那些能够导致他们操作失败、但又不是他们的过错的场景。 这种场景下，让我们来看一下网络端口 —— 用户不应该必须选择一个端口号，而且该端口还有可能与其他用户的冲突。 这就是说，在彼此隔离状态下仍然会出现失败。

为了使用户能够为他们的 Service 选择一个端口号，我们必须确保不能有2个 Service 发生冲突。 我们可以通过为每个 Service 分配它们自己的 IP 地址来实现。

为了保证每个 Service 被分配到一个唯一的 IP，需要一个内部的分配器能够原子地更新 etcd 中的一个全局分配映射表，这个更新操作要先于创建每一个 Service。 为了使 Service 能够获取到 IP，这个映射表对象必须在注册中心存在，否则创建 Service 将会失败，指示一个 IP 不能被分配。 一个后台 Controller 的职责是创建映射表（从 Kubernetes 的旧版本迁移过来，旧版本中是通过在内存中加锁的方式实现），并检查由于管理员干预和清除任意 IP 造成的不合理分配，这些 IP 被分配了但当前没有 Service 使用它们。

#### Service IP 地址

不像 Pod 的 IP 地址，它实际路由到一个固定的目的地，Service 的 IP 实际上不能通过单个主机来进行应答。 相反，我们使用 iptables（Linux 中的数据包处理逻辑）来定义一个虚拟IP地址（VIP），它可以根据需要透明地进行重定向。 当客户端连接到 VIP 时，它们的流量会自动地传输到一个合适的 Endpoint。 环境变量和 DNS，实际上会根据 Service 的 VIP 和端口来进行填充。

kube-proxy支持三种代理模式: 用户空间，iptables和IPVS；它们各自的操作略有不同。

##### Userspace

作为一个例子，考虑前面提到的图片处理应用程序。 当创建 backend Service 时，Kubernetes master 会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 Service 的端口是 1234，该 Service 会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会打开一个新的端口，建立一个从该 VIP 重定向到新端口的 iptables，并开始接收请求连接。

当一个客户端连接到一个 VIP，iptables 规则开始起作用，它会重定向该数据包到 Service代理 的端口。 Service代理 选择一个 backend，并将客户端的流量代理到 backend 上。

这意味着 Service 的所有者能够选择任何他们想使用的端口，而不存在冲突的风险。 客户端可以简单地连接到一个 IP 和端口，而不需要知道实际访问了哪些 Pod。

##### iptables

再次考虑前面提到的图片处理应用程序。 当创建 backend Service 时，Kubernetes 控制面板会给它指派一个虚拟 IP 地址，比如 10.0.0.1。 假设 Service 的端口是 1234，该 Service 会被集群中所有的 kube-proxy 实例观察到。 当代理看到一个新的 Service， 它会安装一系列的 iptables 规则，从 VIP 重定向到 per-Service 规则。 该 per-Service 规则连接到 per-Endpoint 规则，该 per-Endpoint 规则会重定向（目标 NAT）到 backend。

当一个客户端连接到一个 VIP，iptables 规则开始起作用。一个 backend 会被选择（或者根据会话亲和性，或者随机），数据包被重定向到这个 backend。 不像 userspace 代理，数据包从来不拷贝到用户空间，kube-proxy 不是必须为该 VIP 工作而运行，并且客户端 IP 是不可更改的。 当流量打到 Node 的端口上，或通过负载均衡器，会执行相同的基本流程，但是在那些案例中客户端 IP 是可以更改的。

##### IPVS

在大规模集群（例如10,000个服务）中，iptables 操作会显着降低速度。 IPVS 专为负载平衡而设计，并基于内核内哈希表。 因此，您可以通过基于 IPVS 的 kube-proxy 在大量服务中实现性能一致性。 同时，基于 IPVS 的 kube-proxy 具有更复杂的负载平衡算法（最小连接，局部性，加权，持久性）。

### API Object

Service 是Kubernetes REST API中的顶级资源。 您可以在以下位置找到有关API对象的更多详细信息： [Service 对象 API](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#service-v1-core).

### Supported protocols

#### TCP

**FEATURE STATE:** Kubernetes v1.0 [稳定](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

您可以将TCP用于任何类型的服务，这是默认的网络协议。

#### UDP

**FEATURE STATE:** Kubernetes v1.0 [稳定](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

您可以将UDP用于大多数服务。 对于 type=LoadBalancer 服务，对 UDP 的支持取决于提供此功能的云提供商。

#### HTTP

**FEATURE STATE:** Kubernetes v1.1 [稳定](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

如果您的云提供商支持它，则可以在 LoadBalancer 模式下使用服务来设置外部 HTTP/HTTPS 反向代理，并将其转发到该服务的 Endpoints。

**注意：**

您还可以使用 [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) 代替 Service 来公开HTTP / HTTPS服务。

#### PROXY 协议

**FEATURE STATE:** Kubernetes v1.1 [稳定](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

如果您的云提供商支持它(例如, [AWS](https://kubernetes.io/docs/concepts/cluster-administration/cloud-providers/#aws))， 则可以在 LoadBalancer 模式下使用 Service 在 Kubernetes 本身之外配置负载均衡器，该负载均衡器将转发前缀为 [PROXY协议][PROXY protocol](https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt) 的连接。

负载平衡器将发送一系列初始字节，描述传入的连接，类似于此示例

PROXY TCP4 192.0.2.202 10.0.42.7 12345 7\r\n

接下来是来自客户端的数据。

#### SCTP

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/zh/docs/concepts/services-networking/service/)

Kubernetes 支持 SCTP 作为 Service，Endpoint，NetworkPolicy 和 Pod 定义中的 协议 值作为alpha功能。 要启用此功能，集群管理员需要在apiserver上启用 SCTPSupport 功能门，例如 --feature-gates = SCTPSupport = true，…。

启用功能门后，您可以将服务，端点，NetworkPolicy或Pod的 protocol 字段设置为 SCTP。 Kubernetes相应地为 SCTP 关联设置网络，就像为 TCP 连接一样。

##### 警告

##### 支持多宿主SCTP关联

**警告：**

对多宿主 SCTP 关联的支持要求CNI插件可以支持将多个接口和 IP 地址分配给 Pod。 用于多宿主 SCTP 关联的 NAT 在相应的内核模块中需要特殊的逻辑。

##### Service 类型为 LoadBalancer 的服务

**警告：**

如果云提供商的负载平衡器实现支持将 SCTP 作为协议，则只能使用 类型 LoadBalancer 加上 协议 SCTP 创建服务。 否则，服务创建请求将被拒绝。 当前的云负载平衡器提供商（Azure，AWS，CloudStack，GCE，OpenStack）都缺乏对 SCTP 的支持。

##### Windows

**警告：**

基于Windows的节点不支持SCTP。

##### Userspace kube-proxy

**警告：**

当 kube-proxy 处于用户空间模式时，它不支持 SCTP 关联的管理。

### 未来工作

未来我们能预见到，代理策略可能会变得比简单的 round-robin 均衡策略有更多细微的差别，比如 master 选举或分片。 我们也能想到，某些 Service 将具有 “真正” 的负载均衡器，这种情况下 VIP 将简化数据包的传输。

Kubernetes 项目打算为 L7（HTTP）Service 改进我们对它的支持。

Kubernetes 项目打算为 Service 实现更加灵活的请求进入模式，这些 Service 包含当前 ClusterIP、NodePort 和 LoadBalancer 模式，或者更多。

### 接下来

* 阅读 [Connecting Applications with Services](https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/)
* 阅读 [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/)
* 阅读 [Endpoint Slices](https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/)

### 反馈

## Pod 与 Service 的 DNS

该页面概述了Kubernetes对DNS的支持。

* [**介绍**](https://kubernetes.io/zh/docs/concepts/services-networking/dns-pod-service/#%e4%bb%8b%e7%bb%8d)
* [**怎样获取 DNS 名字?**](https://kubernetes.io/zh/docs/concepts/services-networking/dns-pod-service/#%e6%80%8e%e6%a0%b7%e8%8e%b7%e5%8f%96-dns-%e5%90%8d%e5%ad%97)
* [**支持的 DNS 模式**](https://kubernetes.io/zh/docs/concepts/services-networking/dns-pod-service/#%e6%94%af%e6%8c%81%e7%9a%84-dns-%e6%a8%a1%e5%bc%8f)
* [**Pods**](https://kubernetes.io/zh/docs/concepts/services-networking/dns-pod-service/#pods)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/dns-pod-service/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 介绍

Kubernetes DNS 在群集上调度 DNS Pod 和服务，并配置 kubelet 以告知各个容器使用 DNS 服务的 IP 来解析 DNS 名称。

### 怎样获取 DNS 名字?

在集群中定义的每个 Service（包括 DNS 服务器自身）都会被指派一个 DNS 名称。 默认，一个客户端 Pod 的 DNS 搜索列表将包含该 Pod 自己的 Namespace 和集群默认域。 通过如下示例可以很好地说明：

假设在 Kubernetes 集群的 Namespace bar 中，定义了一个Service foo。 运行在Namespace bar 中的一个 Pod，可以简单地通过 DNS 查询 foo 来找到该 Service。 运行在 Namespace quux 中的一个 Pod 可以通过 DNS 查询 foo.bar 找到该 Service。

以下各节详细介绍了受支持的记录类型和支持的布局。 其中代码部分的布局，名称或查询命令均被视为实现细节，如有更改，恕不另行通知。 有关最新规范请查看 [Kubernetes 基于 DNS 的服务发现](https://github.com/kubernetes/dns/blob/master/docs/specification.md).

### 支持的 DNS 模式

下面各段详细说明支持的记录类型和布局。 如果任何其它的布局、名称或查询，碰巧也能够使用，这就需要研究下它们的实现细节，以免后续修改它们又不能使用了。

#### Service

##### A 记录

“正常” Service（除了 Headless Service）会以 my-svc.my-namespace.svc.cluster-domain.example 这种名字的形式被指派一个 DNS A 记录。 这会解析成该 Service 的 Cluster IP。

“Headless” Service（没有Cluster IP）也会以 my-svc.my-namespace.svc.cluster-domain.example 这种名字的形式被指派一个 DNS A 记录。 不像正常 Service，它会解析成该 Service 选择的一组 Pod 的 IP。 希望客户端能够使用这一组 IP，否则就使用标准的 round-robin 策略从这一组 IP 中进行选择。

##### SRV 记录

命名端口需要创建 SRV 记录，这些端口是正常 Service或 [Headless Services](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services) 的一部分。 对每个命名端口，SRV 记录具有 \_my-port-name.\_my-port-protocol.my-svc.my-namespace.svc.cluster-domain.example 这种形式。 对普通 Service，这会被解析成端口号和 CNAME：my-svc.my-namespace.svc.cluster-domain.example。 对 Headless Service，这会被解析成多个结果，Service 对应的每个 backend Pod 各一个， 包含 auto-generated-name.my-svc.my-namespace.svc.cluster-domain.example 这种形式 Pod 的端口号和 CNAME。

### Pods

#### Pod的 hostname 和 subdomain 字段

当前，创建 Pod 后，它的主机名是该 Pod 的 metadata.name 值。

在 v1.2 版本中，用户可以配置 Pod annotation， 通过 pod.beta.kubernetes.io/hostname 来设置 Pod 的主机名。 如果为 Pod 配置了 annotation，会优先使用 Pod 的名称作为主机名。 例如，给定一个 Pod，它具有 annotation pod.beta.kubernetes.io/hostname: my-pod-name，该 Pod 的主机名被设置为 “my-pod-name”。

在 v1.3 版本中，PodSpec 具有 hostname 字段，可以用来指定 Pod 的主机名。这个字段的值优先于 annotation pod.beta.kubernetes.io/hostname。 在 v1.2 版本中引入了 beta 特性，用户可以为 Pod 指定 annotation，其中 pod.beta.kubernetes.io/subdomain 指定了 Pod 的子域名。 最终的域名将是 “...svc.”。 举个例子，Pod 的主机名 annotation 设置为 “foo”，子域名 annotation 设置为 “bar”，在 Namespace “my-namespace” 中对应的 FQDN 为 “foo.bar.my-namespace.svc.cluster.local”。

在 v1.3 版本中，PodSpec 具有 subdomain 字段，可以用来指定 Pod 的子域名。 这个字段的值优先于 annotation pod.beta.kubernetes.io/subdomain 的值。

实例:

apiVersion: v1

kind: Service

metadata:

name: default-subdomain

spec:

selector:

name: busybox

clusterIP: None

ports:

- name: foo *# Actually, no port is needed.*

port: 1234

targetPort: 1234

---

apiVersion: v1

kind: Pod

metadata:

name: busybox1

labels:

name: busybox

spec:

hostname: busybox-1

subdomain: default-subdomain

containers:

- image: busybox:1.28

command:

- sleep

- "3600"

name: busybox

---

apiVersion: v1

kind: Pod

metadata:

name: busybox2

labels:

name: busybox

spec:

hostname: busybox-2

subdomain: default-subdomain

containers:

- image: busybox:1.28

command:

- sleep

- "3600"

name: busybox

如果 Headless Service 与 Pod 在同一个 Namespace 中，它们具有相同的子域名，集群的 KubeDNS 服务器也会为该 Pod 的完整合法主机名返回 A 记录。 例如，在同一个 Namespace 中，给定一个主机名为 “busybox-1” 的 Pod，子域名设置为 “default-subdomain”，名称为 “default-subdomain” 的 Headless Service ，Pod 将看到自己的 FQDN 为 “busybox-1.default-subdomain.my-namespace.svc.cluster.local”。 DNS 会为那个名字提供一个 A 记录，指向该 Pod 的 IP。 “busybox1” 和 “busybox2” 这两个 Pod 分别具有它们自己的 A 记录。

端点对象可以为任何端点地址及其 IP 指定 hostname。

**注意：**

因为没有为 Pod 名称创建A记录，所以要创建 Pod 的 A 记录需要 hostname 。

没有 hostname 但带有 subdomain 的 Pod 只会为指向Pod的IP地址的 headless 服务创建 A 记录(default-subdomain.my-namespace.svc.cluster-domain.example)。 另外，除非在服务上设置了 publishNotReadyAddresses=True，否则 Pod 需要准备好 A 记录。

* “Default“: Pod从运行所在的节点继承名称解析配置。 参考 [相关讨论](https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/#inheriting-dns-from-the-node) 获取更多信息。
* “ClusterFirst“: 与配置的群集域后缀不匹配的任何DNS查询(例如 “www.kubernetes.io” )都将转发到从节点继承的上游名称服务器。 群集管理员可能配置了额外的存根域和上游DNS服务器。 See [相关讨论](https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/#impacts-on-pods) 获取如何 DNS 的查询和处理信息的相关资料。
* “ClusterFirstWithHostNet“: 对于与 hostNetwork 一起运行的 Pod，应显式设置其DNS策略 “ClusterFirstWithHostNet“。
* “None“: 它允许 Pod 忽略 Kubernetes 环境中的 DN S设置。 应该使用 Pod Spec 中的 dnsConfig 字段提供所有 DNS 设置。

**注意：**

“Default” 不是默认的 DNS 策略。 如果未明确指定 dnsPolicy，则使用 “ClusterFirst”。

下面的示例显示了一个Pod，其DNS策略设置为 “ClusterFirstWithHostNet“，因为它已将 hostNetwork 设置为 true。

apiVersion: v1

kind: Pod

metadata:

name: busybox

namespace: default

spec:

containers:

- image: busybox:1.28

command:

- sleep

- "3600"

imagePullPolicy: IfNotPresent

name: busybox

restartPolicy: Always

hostNetwork: **true**

dnsPolicy: ClusterFirstWithHostNet

#### Pod 的 DNS 设定

Pod 的 DNS 配置可让用户对 Pod 的 DNS 设置进行更多控制。

dnsConfig 字段是可选的，它可以与任何 dnsPolicy 设置一起使用。 但是，当 Pod 的 dnsPolicy 设置为 “None” 时，必须指定 dnsConfig 字段。

用户可以在 dnsConfig 字段中指定以下属性：

* nameservers: 将用作于 Pod 的 DNS 服务器的 IP 地址列表。最多可以指定3个 IP 地址。 当 Pod 的 dnsPolicy 设置为 “None” 时，列表必须至少包含一个IP地址，否则此属性是可选的。列出的服务器将合并到从指定的 DNS 策略生成的基本名称服务器，并删除重复的地址。
* searches: 用于在 Pod 中查找主机名的 DNS 搜索域的列表。此属性是可选的。指定后，提供的列表将合并到根据所选 DNS 策略生成的基本搜索域名中。 重复的域名将被删除。    Kubernetes最多允许6个搜索域。
* options: 对象的可选列表，其中每个对象可能具有 name 属性（必需）和 value 属性（可选）。 此属性中的内容将合并到从指定的 DNS 策略生成的选项。 重复的条目将被删除。

以下是具有自定义DNS设置的Pod示例：

| [service/networking/custom-dns.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/custom-dns.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  namespace: default  name: dns-example  spec:  containers:  - name: test  image: nginx  dnsPolicy: "None"  dnsConfig:  nameservers:  - 1.2.3.4  searches:  - ns1.svc.cluster-domain.example  - my.dns.search.suffix  options:  - name: ndots  value: "2"  - name: edns0 |

创建上面的Pod后，容器 test 会在其 /etc/resolv.conf 文件中获取以下内容：

nameserver 1.2.3.4

search ns1.svc.cluster-domain.example my.dns.search.suffix

options ndots:2 edns0

对于IPv6设置，搜索路径和名称服务器应按以下方式设置：

kubectl exec -it dns-example -- cat /etc/resolv.conf

有以下输出：

nameserver fd00:79:30::a

search default.svc.cluster-domain.example svc.cluster-domain.example cluster-domain.example

options ndots:5

#### 可用功能

Pod DNS 配置和 DNS 策略 “None” 的版本对应如下所示。

| k8s version | Feature support |
| --- | --- |
| 1.14 | Stable |
| 1.10 | Beta (on by default) |
| 1.9 | Alpha |

### 接下来

有关管理 DNS 配置的指导，请查看 [配置 DNS 服务](https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/)

### 反馈

## 应用连接到 Service

### Kubernetes 连接容器模型

既然有了一个持续运行、可复制的应用，我们就能够将它暴露到网络上。 在讨论 Kubernetes 网络连接的方式之前，非常值得与 Docker 中 “正常” 方式的网络进行对比。

默认情况下，Docker 使用私有主机网络连接，只能与同在一台机器上的容器进行通信。 为了实现容器的跨节点通信，必须在机器自己的 IP 上为这些容器分配端口，为容器进行端口转发或者代理。

多个开发人员之间协调端口的使用很难做到规模化，那些难以控制的集群级别的问题，都会交由用户自己去处理。 Kubernetes 假设 Pod 可与其它 Pod 通信，不管它们在哪个主机上。 我们给 Pod 分配属于自己的集群私有 IP 地址，所以没必要在 Pod 或映射到的容器的端口和主机端口之间显式地创建连接。 这表明了在 Pod 内的容器都能够连接到本地的每个端口，集群中的所有 Pod 不需要通过 NAT 转换就能够互相看到。 文档的剩余部分将详述如何在一个网络模型之上运行可靠的服务。

该指南使用一个简单的 Nginx server 来演示并证明谈到的概念。同样的原则也体现在一个更加完整的 [Jenkins CI 应用](http://kubernetes.io/blog/2015/07/strong-simple-ssl-for-kubernetes.html) 中。

* [**在集群中暴露 Pod**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#%e5%9c%a8%e9%9b%86%e7%be%a4%e4%b8%ad%e6%9a%b4%e9%9c%b2-pod)
* [**创建 Service**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#%e5%88%9b%e5%bb%ba-service)
* [**访问 Service**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#%e8%ae%bf%e9%97%ae-service)
* [**Service 安全**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#service-%e5%ae%89%e5%85%a8)
* [**暴露 Service**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#%e6%9a%b4%e9%9c%b2-service)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/connect-applications-service/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 在集群中暴露 Pod

我们在之前的示例中已经做过，然而再让我重试一次，这次聚焦在网络连接的视角。 创建一个 Nginx Pod，指示它具有一个容器端口的说明：

| [service/networking/run-my-nginx.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/run-my-nginx.yaml) |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: my-nginx  spec:  selector:  matchLabels:  run: my-nginx  replicas: 2  template:  metadata:  labels:  run: my-nginx  spec:  containers:  - name: my-nginx  image: nginx  ports:  - containerPort: 80 |

这使得可以从集群中任何一个节点来访问它。检查节点，该 Pod 正在运行：

kubectl apply -f ./run-my-nginx.yaml

kubectl get pods -l run=my-nginx -o wide

NAME READY STATUS RESTARTS AGE IP NODE

my-nginx-3800858182-jr4a2 1/1 Running 0 13s 10.244.3.4 kubernetes-minion-905m

my-nginx-3800858182-kna2y 1/1 Running 0 13s 10.244.2.5 kubernetes-minion-ljyd

检查 Pod 的 IP 地址：

kubectl get pods -l run=my-nginx -o yaml | grep podIP

podIP: 10.244.3.4

podIP: 10.244.2.5

应该能够通过 ssh 登录到集群中的任何一个节点上，使用 curl 也能调通所有 IP 地址。 需要注意的是，容器不会使用该节点上的 80 端口，也不会使用任何特定的 NAT 规则去路由流量到 Pod 上。 这意味着可以在同一个节点上运行多个 Pod，使用相同的容器端口，并且可以从集群中任何其他的 Pod 或节点上使用 IP 的方式访问到它们。 像 Docker 一样，端口能够被发布到主机节点的接口上，但是出于网络模型的原因应该从根本上减少这种用法。

如果对此好奇，可以获取更多关于 [如何实现网络模型](https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-achieve-this) 的内容。

### 创建 Service

我们有 Pod 在一个扁平的、集群范围的地址空间中运行 Nginx 服务，可以直接连接到这些 Pod，但如果某个节点死掉了会发生什么呢？ Pod 会终止，Deployment 将创建新的 Pod，且使用不同的 IP。这正是 Service 要解决的问题。

Kubernetes Service 从逻辑上定义了运行在集群中的一组 Pod，这些 Pod 提供了相同的功能。 当每个 Service 创建时，会被分配一个唯一的 IP 地址（也称为 clusterIP）。 这个 IP 地址与一个 Service 的生命周期绑定在一起，当 Service 存在的时候它也不会改变。 可以配置 Pod 使它与 Service 进行通信，Pod 知道与 Service 通信将被自动地负载均衡到该 Service 中的某些 Pod 上。

可以使用 kubectl expose 命令为 2个 Nginx 副本创建一个 Service：

kubectl expose deployment/my-nginx

service/my-nginx exposed

这等价于使用 kubectl create -f 命令创建，对应如下的 yaml 文件：

| [service/networking/nginx-svc.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/nginx-svc.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-nginx  labels:  run: my-nginx  spec:  ports:  - port: 80  protocol: TCP  selector:  run: my-nginx |

上述规约将创建一个 Service，对应具有标签 run: my-nginx 的 Pod，目标 TCP 端口 80，并且在一个抽象的 Service 端口（targetPort：容器接收流量的端口；port：抽象的 Service 端口，可以使任何其它 Pod 访问该 Service 的端口）上暴露。 查看 [Service API 对象](https://kubernetes.io/docs/api-reference/v1.17/#service-v1-core) 了解 Service 定义支持的字段列表。 查看你的 Service 资源:

kubectl get svc my-nginx

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

my-nginx ClusterIP 10.0.162.149 <none> 80/TCP 21s

正如前面所提到的，一个 Service 由一组 backend Pod 组成。这些 Pod 通过 endpoints 暴露出来。 Service Selector 将持续评估，结果被 POST 到一个名称为 my-nginx 的 Endpoint 对象上。 当 Pod 终止后，它会自动从 Endpoint 中移除，新的能够匹配上 Service Selector 的 Pod 将自动地被添加到 Endpoint 中。 检查该 Endpoint，注意到 IP 地址与在第一步创建的 Pod 是相同的。

kubectl describe svc my-nginx

Name: my-nginx

Namespace: default

Labels: run=my-nginx

Annotations: <none>

Selector: run=my-nginx

Type: ClusterIP

IP: 10.0.162.149

Port: <unset> 80/TCP

Endpoints: 10.244.2.5:80,10.244.3.4:80

Session Affinity: None

Events: <none>

kubectl get ep my-nginx

NAME ENDPOINTS AGE

my-nginx 10.244.2.5:80,10.244.3.4:80 1m

现在，能够从集群中任意节点上使用 curl 命令请求 Nginx Service <CLUSTER-IP>:<PORT> 。 注意 Service IP 完全是虚拟的，它从来没有走过网络，如果对它如何工作的原理感到好奇， 可以阅读更多关于 [服务代理](https://kubernetes.io/docs/user-guide/services/#virtual-ips-and-service-proxies) 的内容。

### 访问 Service

Kubernetes支持两种查找服务的主要模式: 环境变量和DNS。 前者开箱即用，而后者则需要[CoreDNS集群插件] [CoreDNS 集群插件](http://releases.k8s.io/master/cluster/addons/dns/coredns).

**注意：**

如果不需要服务环境变量（因为可能与预期的程序冲突，可能要处理的变量太多，或者仅使用DNS等），则可以通过在 [pod spec](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#pod-v1-core)上将 enableServiceLinks 标志设置为 false 来禁用此模式。

#### 环境变量

当 Pod 在 Node 上运行时，kubelet 会为每个活跃的 Service 添加一组环境变量。 这会有一个顺序的问题。想了解为何，检查正在运行的 Nginx Pod 的环境变量（Pod 名称将不会相同）：

kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE

KUBERNETES\_SERVICE\_HOST=10.0.0.1

KUBERNETES\_SERVICE\_PORT=443

KUBERNETES\_SERVICE\_PORT\_HTTPS=443

注意，还没有谈及到 Service。这是因为创建副本先于 Service。 这样做的另一个缺点是，调度器可能在同一个机器上放置所有 Pod，如果该机器宕机则所有的 Service 都会挂掉。 正确的做法是，我们杀掉 2 个 Pod，等待 Deployment 去创建它们。 这次 Service 会 先于 副本存在。这将实现调度器级别的 Service，能够使 Pod 分散创建（假定所有的 Node 都具有同样的容量），以及正确的环境变量：

kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-nginx --replicas=2;

kubectl get pods -l run=my-nginx -o wide

NAME READY STATUS RESTARTS AGE IP NODE

my-nginx-3800858182-e9ihh 1/1 Running 0 5s 10.244.2.7 kubernetes-minion-ljyd

my-nginx-3800858182-j4rm4 1/1 Running 0 5s 10.244.3.8 kubernetes-minion-905m

可能注意到，Pod 具有不同的名称，因为它们被杀掉后并被重新创建。

kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE

KUBERNETES\_SERVICE\_PORT=443

MY\_NGINX\_SERVICE\_HOST=10.0.162.149

KUBERNETES\_SERVICE\_HOST=10.0.0.1

MY\_NGINX\_SERVICE\_PORT=80

KUBERNETES\_SERVICE\_PORT\_HTTPS=443

#### DNS

Kubernetes 提供了一个 DNS 插件 Service，它使用 skydns 自动为其它 Service 指派 DNS 名字。 如果它在集群中处于运行状态，可以通过如下命令来检查：

kubectl get services kube-dns --namespace=kube-system

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kube-dns ClusterIP 10.0.0.10 <none> 53/UDP,53/TCP 8m

如果没有在运行，可以 [启用它](http://releases.k8s.io/master/cluster/addons/dns/kube-dns/README.md#how-do-i-configure-it)。 本段剩余的内容，将假设已经有一个 Service，它具有一个长久存在的 IP（my-nginx），一个为该 IP 指派名称的 DNS 服务器（kube-dns 集群插件），所以可以通过标准做法，使在集群中的任何 Pod 都能与该 Service 通信（例如：gethostbyname）。 让我们运行另一个 curl 应用来进行测试：

kubectl run curl --image=radial/busyboxplus:curl -i --tty

Waiting for pod default/curl-131556218-9fnch to be running, status is Pending, pod ready: false

Hit enter for command prompt

然后，按回车并执行命令 nslookup my-nginx：

[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx

Server: 10.0.0.10

Address 1: 10.0.0.10

Name: my-nginx

Address 1: 10.0.162.149

### Service 安全

到现在为止，我们只在集群内部访问了 Nginx server。在将 Service 暴露到 Internet 之前，我们希望确保通信信道是安全的。对于这可能需要：

* https 自签名证书（除非已经有了一个识别身份的证书）
* 使用证书配置的 Nginx server
* 使证书可以访问 Pod 的[秘钥](https://kubernetes.io/docs/user-guide/secrets)

可以从 [Nginx https 示例](https://github.com/kubernetes/kubernetes/tree/master/examples/https-nginx/) 获取所有上述内容，简明示例如下：

make keys KEY=/tmp/nginx.key CERT=/tmp/nginx.crt

kubectl create secret tls nginxsecret --key /tmp/nginx.key --cert /tmp/nginx.crt

secret/nginxsecret created

kubectl get secrets

NAME TYPE DATA AGE

default-token-il9rc kubernetes.io/service-account-token 1 1d

nginxsecret Opaque 2 1m

以下是您在运行make时遇到问题时要遵循的手动步骤（例如，在Windows上）：

*#create a public private key pair*

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/tmp/nginx.key -out /d/tmp/nginx.crt -subj "/CN=my-nginx/O=my-nginx"

*#convert the keys to base64 encoding*

cat /d/tmp/nginx.crt | base64

cat /d/tmp/nginx.key | base64

使用前面命令的输出来创建yaml文件，如下所示。 base64编码的值应全部放在一行上。

apiVersion: "v1"

kind: "Secret"

metadata:

name: "nginxsecret"

namespace: "default"

data:

nginx.crt: ""

nginx.key: ""

现在使用文件创建 secrets：

kubectl apply -f nginxsecrets.yaml

kubectl get secrets

NAME TYPE DATA AGE

default-token-il9rc kubernetes.io/service-account-token 1 1d

nginxsecret Opaque 2 1m

现在修改 Nginx 副本，启动一个使用在秘钥中的证书的 https 服务器和 Servcie，都暴露端口（80 和 443）：

| [service/networking/nginx-secure-app.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/nginx-secure-app.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-nginx  labels:  run: my-nginx  spec:  type: NodePort  ports:  - port: 8080  targetPort: 80  protocol: TCP  name: http  - port: 443  protocol: TCP  name: https  selector:  run: my-nginx  ---  apiVersion: apps/v1  kind: Deployment  metadata:  name: my-nginx  spec:  selector:  matchLabels:  run: my-nginx  replicas: 1  template:  metadata:  labels:  run: my-nginx  spec:  volumes:  - name: secret-volume  secret:  secretName: nginxsecret  containers:  - name: nginxhttps  image: bprashanth/nginxhttps:1.0  ports:  - containerPort: 443  - containerPort: 80  volumeMounts:  - mountPath: /etc/nginx/ssl  name: secret-volume |

关于 nginx-secure-app manifest 值得注意的点如下：

* 它在相同的文件中包含了 Deployment 和 Service 的规格
* [Nginx server](https://github.com/kubernetes/kubernetes/tree/master/examples/https-nginx/default.conf) 处理 80 端口上的 http 流量，以及 443 端口上的 https 流量，Nginx Service 暴露了这两个端口。
* 每个容器访问挂载在 /etc/nginx/ssl 卷上的秘钥。这需要在 Nginx server 启动之前安装好。

kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml

这时可以从任何节点访问到 Nginx server。

kubectl get pods -o yaml | grep -i podip

podIP: 10.244.3.5

node $ curl -k https://10.244.3.5

...

<h1>Welcome to nginx!</h1>

注意最后一步我们是如何提供 -k 参数执行 curl命令的，这是因为在证书生成时，我们不知道任何关于运行 Nginx 的 Pod 的信息，所以不得不在执行 curl 命令时忽略 CName 不匹配的情况。 通过创建 Service，我们连接了在证书中的 CName 与在 Service 查询时被 Pod使用的实际 DNS 名字。 让我们从一个 Pod 来测试（为了简化使用同一个秘钥，Pod 仅需要使用 nginx.crt 去访问 Service）：

| [service/networking/curlpod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/curlpod.yaml) |
| --- |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: curl-deployment  spec:  selector:  matchLabels:  app: curlpod  replicas: 1  template:  metadata:  labels:  app: curlpod  spec:  volumes:  - name: secret-volume  secret:  secretName: nginxsecret  containers:  - name: curlpod  command:  - sh  - -c  - while **true**; do sleep 1; done  image: radial/busyboxplus:curl  volumeMounts:  - mountPath: /etc/nginx/ssl  name: secret-volume |

kubectl apply -f ./curlpod.yaml

kubectl get pods -l app=curlpod

NAME READY STATUS RESTARTS AGE

curl-deployment-1515033274-1410r 1/1 Running 0 1m

kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert /etc/nginx/ssl/nginx.crt

...

<title>Welcome to nginx!</title>

...

### 暴露 Service

对我们应用的某些部分，可能希望将 Service 暴露在一个外部 IP 地址上。 Kubernetes 支持两种实现方式：NodePort 和 LoadBalancer。 在上一段创建的 Service 使用了 NodePort，因此 Nginx https 副本已经就绪，如果使用一个公网 IP，能够处理 Internet 上的流量。

kubectl get svc my-nginx -o yaml | grep nodePort -C 5

uid: 07191fb3-f61a-11e5-8ae5-42010af00002

spec:

clusterIP: 10.0.162.149

ports:

- name: http

nodePort: 31704

port: 8080

protocol: TCP

targetPort: 80

- name: https

nodePort: 32453

port: 443

protocol: TCP

targetPort: 443

selector:

run: my-nginx

kubectl get nodes -o yaml | grep ExternalIP -C 1

- address: 104.197.41.11

type: ExternalIP

allocatable:

--

- address: 23.251.152.56

type: ExternalIP

allocatable:

...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k

...

<h1>Welcome to nginx!</h1>

让我们重新创建一个 Service，使用一个云负载均衡器，只需要将 my-nginx Service 的 Type 由 NodePort 改成 LoadBalancer。

kubectl edit svc my-nginx

kubectl get svc my-nginx

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

my-nginx ClusterIP 10.0.162.149 162.222.184.144 80/TCP,81/TCP,82/TCP 21s

curl https://<EXTERNAL-IP> -k

...

<title>Welcome to nginx!</title>

在 EXTERNAL-IP 列指定的 IP 地址是在公网上可用的。CLUSTER-IP 只在集群/私有云网络中可用。

注意，在 AWS 上类型 LoadBalancer 创建一个 ELB，它使用主机名（比较长），而不是 IP。 它太长以至于不能适配标准 kubectl get svc 的输出，事实上需要通过执行 kubectl describe service my-nginx 命令来查看它。 可以看到类似如下内容：

kubectl describe service my-nginx

...

LoadBalancer Ingress: a320587ffd19711e5a37606cf4a74574-1142138393.us-east-1.elb.amazonaws.com

...

### 接下来

Kubernetes 也支持联合 Service，能够跨多个集群和云提供商，为 Service 提供逐步增强的可用性、更优的容错、更好的可伸缩性。 查看 [联合 Service 用户指南](https://kubernetes.io/docs/concepts/cluster-administration/federation-service-discovery/) 获取更进一步信息。

### 反馈

## Ingress

**FEATURE STATE:** Kubernetes v1.1 [beta](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/)

Ingress 是对集群中服务的外部访问进行管理的 API 对象，典型的访问方式是 HTTP。

Ingress 可以提供负载均衡、SSL 终结和基于名称的虚拟托管。

* [**专用术语**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e4%b8%93%e7%94%a8%e6%9c%af%e8%af%ad)
* [**Ingress 是什么？**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#ingress-%e6%98%af%e4%bb%80%e4%b9%88)
* [**环境准备**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e7%8e%af%e5%a2%83%e5%87%86%e5%a4%87)
* [**Ingress 资源**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#ingress-%e8%b5%84%e6%ba%90)
* [**更新 Ingress**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e6%9b%b4%e6%96%b0-ingress)
* [**跨可用区失败**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e8%b7%a8%e5%8f%af%e7%94%a8%e5%8c%ba%e5%a4%b1%e8%b4%a5)
* [**跨可用区失败**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e8%b7%a8%e5%8f%af%e7%94%a8%e5%8c%ba%e5%a4%b1%e8%b4%a5-1)
* [**未来工作**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e6%9c%aa%e6%9d%a5%e5%b7%a5%e4%bd%9c)
* [**替代方案**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e6%9b%bf%e4%bb%a3%e6%96%b9%e6%a1%88)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 专用术语

为了表达更加清晰，本指南定义了以下术语：

节点（Node）

Kubernetes 集群中其中一台工作机器，是集群的一部分。

<!– Cluster

A set of Nodes that run containerized applications managed by Kubernetes. For this example, and in most common Kubernetes deployments, nodes in the cluster are not part of the public internet. –> 集群（Cluster）

一组运行程序（这些程序是容器化的，被 Kubernetes 管理的）的节点。 在此示例中，和在大多数常见的Kubernetes部署方案，集群中的节点都不会是公共网络。

<!– Edge router

A router that enforces the firewall policy for your cluster. This could be a gateway managed by a cloud provider or a physical piece of hardware. –> 边缘路由器（Edge router）

在集群中强制性执行防火墙策略的路由器（router）。可以是由云提供商管理的网关，也可以是物理硬件。

<!– Cluster network

A set of links, logical or physical, that facilitate communication within a cluster according to the Kubernetes [networking model](https://kubernetes.io/docs/concepts/cluster-administration/networking/). –> 集群网络（Cluster network）

一组逻辑或物理的链接，根据 Kubernetes [网络模型](https://kubernetes.io/docs/concepts/cluster-administration/networking/) 在集群内实现通信。

<!– Service

A Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) that identifies a set of Pods using [label](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels) selectors. Unless mentioned otherwise, Services are assumed to have virtual IPs only routable within the cluster network. –> 服务（Service）： Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) 使用 [标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels) 选择器（selectors）标识的一组 Pod。除非另有说明，否则假定服务只具有在集群网络中可路由的虚拟 IP。

### Ingress 是什么？

Ingress公开了从集群外部到集群内 [services](https://kubernetes.io/docs/concepts/services-networking/service/) 的HTTP和HTTPS路由。 流量路由由Ingress资源上定义的规则控制。

internet

|

[ Ingress ]

--|-----|--

[ Services ]

可以将 Ingress 配置为提供服务外部可访问的 URL、负载均衡流量、终止 SSL / TLS 并提供基于名称的虚拟主机。[Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)通常负责通过负载均衡器来实现 Ingress，尽管它也可以配置边缘路由器或其他前端来帮助处理流量。

Ingress 不会公开任意端口或协议。 将 HTTP 和 HTTPS 以外的服务公开到 Internet 时，通常使用 [Service.Type=NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) 或者 [Service.Type=LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer) 类型的服务。

可以将Ingress配置为提供服务外部可访问的URL，负载均衡流量，终止 SSL / TLS 并提供基于名称的虚拟主机。 [Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)通常负责通过负载平衡器来实现入口，尽管它也可以配置边缘路由器或其他前端以帮助处理流量。

Ingress 不会公开任意端口或协议。 将 HTTP 和 HTTPS 以外的服务公开给 Internet 时，通常使用以下类型的服务 [Service.Type=NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport) 或者 [Service.Type=LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer).

### 环境准备

您必须具有 [ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)才能满足 Ingress 的要求。仅创建 Ingress 资源无效。

您可能需要部署 Ingress 控制器，例如 [ingress-nginx](https://kubernetes.github.io/ingress-nginx/deploy/)。您可以从许多[Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)中进行选择。

**注意：**

确保您查看了 Ingress 控制器的文档，以了解选择它的注意事项。

一定要检查一下这个控制器的 [beta 限制](https://github.com/kubernetes/ingress-gce/blob/master/LIMITATIONS.md)。 在 GCE／Google Kubernetes Engine 之外的环境中，需要将[控制器部署](https://git.k8s.io/ingress-nginx/README.md) 为 Pod。

**注意：**

确保您查看了 Ingress 控制器的文档，以了解选择它的注意事项。

### Ingress 资源

一个最小的 Ingress 资源示例：

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: test-ingress

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- http:

paths:

- path: /testpath

backend:

serviceName: test

servicePort: 80

与所有其他 Kubernetes 资源一样，Ingress 需要使用 apiVersion、kind 和 metadata 字段。 有关使用配置文件的一般信息，请参见[部署应用](https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/)、 [配置容器](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/)、[管理资源](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/)。 Ingress 经常使用注解（annotations）来配置一些选项，具体取决于 Ingress 控制器，例如 [rewrite-target annotation](https://github.com/kubernetes/ingress-nginx/blob/master/docs/examples/rewrite/README.md)。 不同的 [Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)支持不同的注解（annotations）。查看文档以供您选择 Ingress 控制器，以了解支持哪些注解（annotations）。

Ingress [规范](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status) 具有配置负载均衡器或者代理服务器所需的所有信息。最重要的是，它包含与所有传入请求匹配的规则列表。Ingress 资源仅支持用于定向 HTTP 流量的规则。

#### Ingress 规则

每个 HTTP 规则都包含以下信息：

* 可选主机。在此示例中，未指定主机，因此该规则适用于通过指定 IP 地址的所有入站 HTTP 通信。如果提供了主机（例如 foo.bar.com），则规则适用于该主机。
* 路径列表（例如，/testpath）,每个路径都有一个由 serviceName 和 servicePort 定义的关联后端。在负载均衡器将流量定向到引用的服务之前，主机和路径都必须匹配传入请求的内容。
* 后端是[服务文档](https://kubernetes.io/docs/concepts/services-networking/service/)中所述的服务和端口名称的组合。与规则的主机和路径匹配的对 Ingress 的HTTP（和HTTPS）请求将发送到列出的后端。

通常在 Ingress 控制器中配置默认后端，以服务任何不符合规范中路径的请求。

#### 默认后端

没有规则的 Ingress 将所有流量发送到单个默认后端。默认后端通常是 [Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)的配置选项，并且未在 Ingress 资源中指定。

如果没有主机或路径与 Ingress 对象中的 HTTP 请求匹配，则流量将路由到您的默认后端。

Ingress 类型

#### 单服务 Ingress

现有的 Kubernetes 概念允许您暴露单个 Service (查看[替代方案](https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#alternatives))，您也可以通过指定无规则的 默认后端 来对 Ingress 进行此操作。

| [service/networking/ingress.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/ingress.yaml) |
| --- |
| apiVersion: extensions/v1beta1  kind: Ingress  metadata:  name: test-ingress  spec:  backend:  serviceName: testsvc  servicePort: 80 |

如果使用 kubectl apply -f 创建它，则应该能够查看刚刚添加的 Ingress 的状态：

kubectl get ingress test-ingress

NAME HOSTS ADDRESS PORTS AGE

test-ingress \* 107.178.254.228 80 59s

其中 107.178.254.228 是由 Ingress 控制器分配以满足该 Ingress 的 IP。

**注意：**

Ingress 控制器和负载均衡器可能需要一两分钟才能分配 IP 地址。 在此之前，您通常会看到地址为 <pending>。

**注意：**

入口控制器和负载平衡器可能需要一两分钟才能分配IP地址。 在此之前，您通常会看到地址字段的值被设定为 <pending>。

#### 简单分列

一个分列配置根据请求的 HTTP URI 将流量从单个 IP 地址路由到多个服务。 Ingress 允许您将负载均衡器的数量降至最低。例如，这样的设置：

foo.bar.com -> 178.91.123.132 -> / foo service1:4200

/ bar service2:8080

将需要一个 Ingress，例如：

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: simple-fanout-example

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- host: foo.bar.com

http:

paths:

- path: /foo

backend:

serviceName: service1

servicePort: 4200

- path: /bar

backend:

serviceName: service2

servicePort: 8080

当您使用 kubectl apply -f 创建 Ingress 时：

kubectl describe ingress simple-fanout-example

Name: simple-fanout-example

Namespace: default

Address: 178.91.123.132

Default backend: default-http-backend:80 (10.8.2.3:8080)

Rules:

Host Path Backends

---- ---- --------

foo.bar.com

/foo service1:4200 (10.8.0.90:4200)

/bar service2:8080 (10.8.0.91:8080)

Annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ADD 22s loadbalancer-controller default/test

Ingress 控制器将提供实现特定的负载均衡器来满足 Ingress，只要 Service (s1，s2) 存在。 当它这样做了，你会在地址栏看到负载均衡器的地址。

**注意：**

根据您使用的 [Ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)，您可能需要创建默认 HTTP 后端 [Service](https://kubernetes.io/docs/concepts/services-networking/service/)。

#### 基于名称的虚拟托管

基于名称的虚拟主机支持将 HTTP 流量路由到同一 IP 地址上的多个主机名。

foo.bar.com --| |-> foo.bar.com service1:80

| 178.91.123.132 |

bar.foo.com --| |-> bar.foo.com service2:80

以下 Ingress 让后台负载均衡器基于[主机 header](https://tools.ietf.org/html/rfc7230#section-5.4) 路由请求。

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: name-virtual-host-ingress

spec:

rules:

- host: foo.bar.com

http:

paths:

- backend:

serviceName: service1

servicePort: 80

- host: bar.foo.com

http:

paths:

- backend:

serviceName: service2

servicePort: 80

如果您创建的 Ingress 资源没有规则中定义的任何主机，则可以匹配到您 Ingress 控制器 IP 地址的任何网络流量，而无需基于名称的虚拟主机。

例如，以下 Ingress 资源会将 first.bar.com 请求的流量路由到 service1，将 second.foo.com 请求的流量路由到 service2，而没有在请求中定义主机名的 IP 地址的流量路由（即，不提供请求标头）到 service3。

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: name-virtual-host-ingress

spec:

rules:

- host: first.bar.com

http:

paths:

- backend:

serviceName: service1

servicePort: 80

- host: second.foo.com

http:

paths:

- backend:

serviceName: service2

servicePort: 80

- http:

paths:

- backend:

serviceName: service3

servicePort: 80

#### TLS

您可以通过指定包含 TLS 私钥和证书的 secret [Secret](https://kubernetes.io/docs/concepts/configuration/secret/) 来加密 Ingress。 目前，Ingress 只支持单个 TLS 端口 443，并假定 TLS 终止。

如果 Ingress 中的 TLS 配置部分指定了不同的主机，那么它们将根据通过 SNI TLS 扩展指定的主机名（如果 Ingress 控制器支持 SNI）在同一端口上进行复用。 TLS Secret 必须包含名为 tls.crt 和 tls.key 的密钥，这些密钥包含用于 TLS 的证书和私钥，例如：

apiVersion: v1

kind: Secret

metadata:

name: testsecret-tls

namespace: default

data:

tls.crt: base64 encoded cert

tls.key: base64 encoded key

type: kubernetes.io/tls

在 Ingress 中引用此 Secret 将会告诉 Ingress 控制器使用 TLS 加密从客户端到负载均衡器的通道。您需要确保创建的 TLS secret 来自包含 sslexample.foo.com 的 CN 的证书。

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

name: tls-example-ingress

spec:

tls:

- hosts:

- sslexample.foo.com

secretName: testsecret-tls

rules:

- host: sslexample.foo.com

http:

paths:

- path: /

backend:

serviceName: service1

servicePort: 80

**注意：**

**注意：**

各种 Ingress 控制器所支持的 TLS 功能之间存在差异。请参阅有关文件 [nginx](https://git.k8s.io/ingress-nginx/README.md#https)、 [GCE](https://git.k8s.io/ingress-gce/README.md#frontend-https) 或者任何其他平台特定的 Ingress 控制器，以了解 TLS 如何在您的环境中工作。

#### 负载均衡

Ingress 控制器使用一些适用于所有 Ingress 的负载均衡策略设置进行自举，例如负载均衡算法、后端权重方案和其他等。更高级的负载均衡概念（例如，持久会话、动态权重）尚未通过 Ingress 公开。您可以通过用于服务的负载均衡器来获取这些功能。

值得注意的是，即使健康检查不是通过 Ingress 直接暴露的，但是在 Kubernetes 中存在并行概念，比如 [就绪检查](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/)，它允许您实现相同的最终结果。 请检查控制器特殊说明文档，以了解他们是怎样处理健康检查的 ( [nginx](https://git.k8s.io/ingress-nginx/README.md)， [GCE](https://git.k8s.io/ingress-gce/README.md#health-checks))。

### 更新 Ingress

要更新现有的 Ingress 以添加新的 Host，可以通过编辑资源来对其进行更新：

kubectl describe ingress test

Name: test

Namespace: default

Address: 178.91.123.132

Default backend: default-http-backend:80 (10.8.2.3:8080)

Rules:

Host Path Backends

---- ---- --------

foo.bar.com

/foo service1:80 (10.8.0.90:80)

Annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ADD 35s loadbalancer-controller default/test

kubectl edit ingress test

这将弹出具有 YAML 格式的现有配置的编辑器。 修改它来增加新的主机：

spec:

rules:

- host: foo.bar.com

http:

paths:

- backend:

serviceName: service1

servicePort: 80

path: /foo

- host: bar.baz.com

http:

paths:

- backend:

serviceName: service2

servicePort: 80

path: /foo

..

保存更改后，kubectl 将更新 API 服务器中的资源，该资源将告诉 Ingress 控制器重新配置负载均衡器。

验证：

kubectl describe ingress test

Name: test

Namespace: default

Address: 178.91.123.132

Default backend: default-http-backend:80 (10.8.2.3:8080)

Rules:

Host Path Backends

---- ---- --------

foo.bar.com

/foo service1:80 (10.8.0.90:80)

bar.baz.com

/foo service2:80 (10.8.0.91:80)

Annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ADD 45s loadbalancer-controller default/test

您可以通过 kubectl replace -f 命令调用修改后的 Ingress yaml 文件来获得同样的结果。

### 跨可用区失败

### 跨可用区失败

用于跨故障域传播流量的技术在云提供商之间是不同的。详情请查阅相关 Ingress 控制器的文档。 请查看相关[Ingress控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers)的文档以了解详细信息。 您还可以参考[联合文档](https://kubernetes.io/docs/concepts/cluster-administration/federation/)，以获取有关在联合集群中部署Ingress的详细信息。

### 未来工作

跟踪 [SIG 网络](https://github.com/kubernetes/community/tree/master/sig-network)以获得有关 Ingress 和相关资源演变的更多细节。您还可以跟踪 [Ingress 仓库](https://github.com/kubernetes/ingress/tree/master)以获取有关各种 Ingress 控制器的更多细节。

### 替代方案

不直接使用 Ingress 资源，也有多种方法暴露 Service：

* 使用 [Service.Type=LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer)
* 使用 [Service.Type=NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport)

### 接下来

* 了解更多 [ingress 控制器](https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/)
* [使用 NGINX 控制器在 Minikube 上安装 Ingress](https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube)

### 反馈

## Ingress 控制器

为了让 Ingress 资源工作，集群必须有一个正在运行的 Ingress 控制器。

与其他类型的控制器不同，它们是作为 kube-controller-manager 二进制文件的一部分运行的，而 Ingress 控制器不是随集群自动启动的。 通过此页面可选择最适合您的集群的 ingress 控制器实现。

Kubernetes 作为一个项目，目前支持和维护 [GCE](https://git.k8s.io/ingress-gce/README.md) 和 [nginx](https://git.k8s.io/ingress-nginx/README.md) 控制器。

* [**其他控制器**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress-controllers/#%e5%85%b6%e4%bb%96%e6%8e%a7%e5%88%b6%e5%99%a8)
* [**使用多个 Ingress 控制器**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress-controllers/#%e4%bd%bf%e7%94%a8%e5%a4%9a%e4%b8%aa-ingress-%e6%8e%a7%e5%88%b6%e5%99%a8)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/ingress-controllers/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 其他控制器

* [Ambassador](https://www.getambassador.io/) API 网关， 一个基于 [Envoy](https://www.envoyproxy.io/) 的 ingress 控制器，有着来自 [Datawire](https://www.datawire.io/) [社区](https://www.getambassador.io/docs)或[商业](https://www.getambassador.io/pro/)的支持。
* [AppsCode Inc.](https://appscode.com/) 为最广泛使用的基于 [HAProxy](http://www.haproxy.org/) 的 ingress 控制器 [Voyager](https://appscode.com/products/voyager) 提供支持和维护.
* [Contour](https://projectcontour.io/) 是一个基于 [Envoy](https://www.envoyproxy.io/) 的 ingress 控制器，它由 VMware 提供和支持。
* Citrix 为其硬件（MPX），虚拟化（VPX）和 [免费容器化 (CPX) ADC](https://www.citrix.com/products/citrix-adc/cpx-express.html) 提供了一个 [Ingress 控制器](https://github.com/citrix/citrix-k8s-ingress-controller)，用于[裸金属](https://github.com/citrix/citrix-k8s-ingress-controller/tree/master/deployment/baremetal)和[云](https://github.com/citrix/citrix-k8s-ingress-controller/tree/master/deployment)部署。
* F5 Networks 为 [用于 Kubernetes 的 F5 BIG-IP 控制器](http://clouddocs.f5.com/products/connectors/k8s-bigip-ctlr/latest)提供[支持和维护](https://support.f5.com/csp/article/K86859508)。
* [Gloo](https://gloo.solo.io/) 是一个开源的基于 [Envoy](https://www.envoyproxy.io/) 的 ingress 控制器，它提供了 API 网关功能，有着来自 [solo.io](https://www.solo.io/) 的企业级支持。
* [HAProxy Technologies](https://www.haproxy.com/) 为 [HAProxy Ingress Controller for Kubernetes](https://github.com/haproxytech/kubernetes-ingress). See the [official documentation](https://www.haproxy.com/documentation/hapee/1-9r1/traffic-management/kubernetes-ingress-controller/) 提供支持和运维服务。
* 基于 [Istio](https://istio.io/) 的 ingress 控制器[控制 Ingress 流量](https://istio.io/docs/tasks/traffic-management/ingress/)。
* [Kong](https://konghq.com/) 为[用于 Kubernetes 的 Kong Ingress 控制器](https://github.com/Kong/kubernetes-ingress-controller) 提供[社区](https://discuss.konghq.com/c/kubernetes)或[商业](https://konghq.com/kong-enterprise/)支持和维护。
* [NGINX, Inc.](https://www.nginx.com/) 为[用于 Kubernetes 的 NGINX Ingress 控制器](https://www.nginx.com/products/nginx/kubernetes-ingress-controller)提供支持和维护。
* [Skipper](https://opensource.zalando.com/skipper/kubernetes/ingress-controller/) HTTP路由器和反向代理，用于服务组合，包括诸如Kubernetes Ingress之类的用例，被设计为用于构建自定义代理的库。
* [Traefik](https://github.com/containous/traefik) 是一个全功能的 ingress 控制器 （[Let’s Encrypt](https://letsencrypt.org/)，secrets，http2，websocket），并且它也有来自 [Containous](https://containo.us/services) 的商业支持。

### 使用多个 Ingress 控制器

你可以在集群中部署[任意数量的 ingress 控制器](https://git.k8s.io/ingress-nginx/docs/user-guide/multiple-ingress.md#multiple-ingress-controllers)。 创建 ingress 时，应该使用适当的 [ingress.class](https://git.k8s.io/ingress-gce/docs/faq/README.md#how-do-i-run-multiple-ingress-controllers-in-the-same-cluster) 注解每个 ingress 以表明在集群中如果有多个 ingress 控制器时，应该使用哪个 ingress 控制器。

如果不定义 ingress.class，云提供商可能使用默认的 ingress 控制器。

理想情况下，所有 ingress 控制器都应满足此规范，但各种 ingress 控制器的操作略有不同。

**注意：**

确保您查看了 ingress 控制器的文档，以了解选择它的注意事项。

### 接下来

* 了解更多关于 [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/)。
* [在 Minikube 上使用 NGINX 控制器安装 Ingress](https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube)。

### 反馈

## 网络策略

网络策略（NetworkPolicy）是一种关于pod间及pod与其他网络端点间所允许的通信规则的规范。

NetworkPolicy 资源使用标签选择pod，并定义选定pod所允许的通信规则。

* [**前提**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#%e5%89%8d%e6%8f%90)
* [**隔离和非隔离的Pod**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#%e9%9a%94%e7%a6%bb%e5%92%8c%e9%9d%9e%e9%9a%94%e7%a6%bb%e7%9a%84pod)
* [**NetworkPolicy 资源**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#networkpolicy-%e8%b5%84%e6%ba%90)
* [**选择器 to 和 from 的行为**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#%e9%80%89%e6%8b%a9%e5%99%a8-to-%e5%92%8c-from-%e7%9a%84%e8%a1%8c%e4%b8%ba)
* [**默认策略**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#%e9%bb%98%e8%ae%a4%e7%ad%96%e7%95%a5)
* [**SCTP 支持**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#sctp-%e6%94%af%e6%8c%81)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 前提

网络策略通过网络插件来实现，所以用户必须使用支持 NetworkPolicy 的网络解决方案 - 简单地创建资源对象，而没有控制器来使它生效的话，是没有任何作用的。

### 隔离和非隔离的Pod

默认情况下，Pod是非隔离的，它们接受任何来源的流量。

Pod可以通过相关的网络策略进行隔离。一旦命名空间中有网络策略选择了特定的Pod，该Pod会拒绝网络策略所不允许的连接。 (命名空间下其他未被网络策略所选择的Pod会继续接收所有的流量)

### NetworkPolicy 资源

查看 [网络策略](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#networkpolicy-v1-networking-k8s-io) 来了解资源定义。

下面是一个 NetworkPolicy 的示例:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: test-network-policy

namespace: default

spec:

podSelector:

matchLabels:

role: db

policyTypes:

- Ingress

- Egress

ingress:

- from:

- ipBlock:

cidr: 172.17.0.0/16

except:

- 172.17.1.0/24

- namespaceSelector:

matchLabels:

project: myproject

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: TCP

port: 6379

egress:

- to:

- ipBlock:

cidr: 10.0.0.0/24

ports:

- protocol: TCP

port: 5978

除非选择支持网络策略的网络解决方案，否则将上述示例发送到API服务器没有任何效果。

**必填字段**: 与所有其他的Kubernetes配置一样，NetworkPolicy 需要 apiVersion、 kind和 metadata 字段。 关于配置文件操作的一般信息，请参考 [这里](https://kubernetes.io/docs/user-guide/simple-yaml)、 [这里](https://kubernetes.io/docs/user-guide/configuring-containers)和 [这里](https://kubernetes.io/docs/user-guide/working-with-resources)。

**spec**: NetworkPolicy [spec](https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status) 中包含了在一个命名空间中定义特定网络策略所需的所有信息

**podSelector**: 每个 NetworkPolicy 都包括一个 podSelector ，它对该策略所应用的一组Pod进行选择。因为 NetworkPolicy 目前只支持定义 ingress 规则，这里的 podSelector 本质上是为该策略定义 “目标pod” 。示例中的策略选择带有 “role=db” 标签的pod。空的 podSelector 选择命名空间下的所有pod。

**policyTypes**: Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. The policyTypes field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no policyTypes are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.

**ingress**: 每个 NetworkPolicy 包含一个 ingress 规则的白名单列表。（其中的）规则允许同时匹配 from 和 ports 部分的流量。示例策略中包含一条简单的规则： 它匹配一个单一的端口，来自两个来源中的一个， 第一个通过 namespaceSelector 指定，第二个通过 podSelector 指定。

**egress**: 每个 NetworkPolicy 包含一个 egress 规则的白名单列表。每个规则都允许匹配 to 和 port 部分的流量。该示例策略包含一条规则，该规则将单个端口上的流量匹配到 10.0.0.0/24 中的任何目的地。

所以，示例网络策略:

1. 隔离 “default” 命名空间下 “role=db” 的pod (如果它们不是已经被隔离的话)。
2. 允许从 “default” 命名空间下带有 “role=frontend” 标签的pod到 “default” 命名空间下的pod的6379 TCP端口的连接。
   * 标签为 “role=frontend” 的 “default” 名称空间中的任何Pod
   * 名称空间中带有标签 “project=myproject” 的任何pod
   * IP 地址范围为 172.17.0.0–172.17.0.255 和 172.17.2.0–172.17.255.255（即，除了 172.17.1.0/24 之外的所有 172.17.0.0/16）
3. 允许从带有 “project=myproject” 标签的命名空间下的任何 pod 到 “default” 命名空间下的 pod 的6379 TCP端口的连接。

查看 [网络策略入门指南](https://kubernetes.io/docs/getting-started-guides/network-policy/walkthrough) 了解更多示例。

### 选择器 to 和 from 的行为

可以在 ingress from 部分或 egress to 部分中指定四种选择器：

**podSelector**: 这将在与 NetworkPolicy 相同的名称空间中选择特定的 Pod，应将其允许作为入口源或出口目的地。

**namespaceSelector**: 这将选择特定的名称空间，应将所有 Pod 用作其输入源或输出目的地。

**namespaceSelector** 和 **podSelector**: 一个指定 namespaceSelector 和 podSelector 的 to/from 条目选择特定命名空间中的特定 Pod。注意使用正确的YAML语法；这项策略：

...

ingress:

- from:

- namespaceSelector:

matchLabels:

user: alice

podSelector:

matchLabels:

role: client

...

在 from 数组中仅包含一个元素，只允许来自标有 role = client 的 Pod 且该 Pod 所在的名称空间中标有user=alice的连接。这项策略：

...

ingress:

- from:

- namespaceSelector:

matchLabels:

user: alice

- podSelector:

matchLabels:

role: client

...

在 from 数组中包含两个元素，允许来自本地命名空间中标有 role = client 的 Pod 的连接，\*或\*来自任何名称空间中标有user = alice的任何Pod的连接。

如有疑问，请使用 kubectl describe 查看 Kubernetes 如何解释该策略。

**ipBlock**: 这将选择特定的 IP CIDR 范围以用作入口源或出口目的地。 这些应该是群集外部 IP，因为 Pod IP 存在时间短暂的且随机产生。

群集的入口和出口机制通常需要重写数据包的源 IP 或目标 IP。在发生这种情况的情况下，不确定在 NetworkPolicy 处理之前还是之后发生，并且对于网络插件，云提供商，Service 实现等的不同组合，其行为可能会有所不同。

在进入的情况下，这意味着在某些情况下，您可以根据实际的原始源 IP 过滤传入的数据包，而在其他情况下，NetworkPolicy 所作用的 源IP 则可能是 LoadBalancer 或 Pod的节点等。

对于出口，这意味着从 Pod 到被重写为集群外部 IP 的 Service IP 的连接可能会或可能不会受到基于 ipBlock 的策略的约束。

### 默认策略

默认情况下，如果名称空间中不存在任何策略，则所有进出该名称空间中的Pod的流量都被允许。以下示例使您可以更改该名称空间中的默认行为。

#### 默认拒绝所有入口流量

您可以通过创建选择所有容器但不允许任何进入这些容器的入口流量的 NetworkPolicy 来为名称空间创建 “default” 隔离策略。

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Ingress

这样可以确保即使容器没有选择其他任何 NetworkPolicy，也仍然可以被隔离。此策略不会更改默认的出口隔离行为。

#### 默认允许所有入口流量

如果要允许所有流量进入某个命名空间中的所有 Pod（即使添加了导致某些 Pod 被视为“隔离”的策略），则可以创建一个策略来明确允许该命名空间中的所有流量。

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-all

spec:

podSelector: {}

ingress:

- {}

policyTypes:

- Ingress

#### 默认拒绝所有出口流量

您可以通过创建选择所有容器但不允许来自这些容器的任何出口流量的 NetworkPolicy 来为名称空间创建 “default” egress 隔离策略。

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Egress

这样可以确保即使没有被其他任何 NetworkPolicy 选择的 Pod 也不会被允许流出流量。此策略不会更改默认的 ingress 隔离行为。

#### 默认允许所有出口流量

如果要允许来自命名空间中所有 Pod 的所有流量（即使添加了导致某些 Pod 被视为“隔离”的策略），则可以创建一个策略，该策略明确允许该命名空间中的所有出口流量。

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-all

spec:

podSelector: {}

egress:

- {}

policyTypes:

- Egress

#### 默认拒绝所有入口和所有出口流量

您可以为名称空间创建 “default” 策略，以通过在该名称空间中创建以下 NetworkPolicy 来阻止所有入站和出站流量。

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Ingress

- Egress

这样可以确保即使没有被其他任何 NetworkPolicy 选择的 Pod 也不会被允许进入或流出流量。

### SCTP 支持

**FEATURE STATE:** Kubernetes v1.12 [alpha](https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/)

Kubernetes 支持 SCTP 作为 NetworkPolicy 定义中的协议值作为 alpha 功能提供。要启用此功能，集群管理员需要在 apiserver 上启用 SCTPSupport 功能门，例如 “--feature-gates=SCTPSupport=true,...”。启用功能门后，用户可以将 NetworkPolicy 的 protocol 字段设置为 SCTP。 Kubernetes 相应地为 SCTP 关联设置网络，就像为 TCP 连接一样。

CNI插件必须在 NetworkPolicy 中将 SCTP 作为 protocol 值支持。

### 接下来

* 查看 [声明网络策略](https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/) 来进行更多的示例演练
* 有关NetworkPolicy资源启用的常见方案的更多信息，请参见 [Recipes](https://github.com/ahmetb/kubernetes-network-policy-recipes)。

### 反馈

## 使用 HostAliases 向 Pod /etc/hosts 文件添加条目

当 DNS 配置以及其它选项不合理的时候，通过向 Pod 的 /etc/hosts 文件中添加条目，可以在 Pod 级别覆盖对主机名的解析。在 1.7 版本，用户可以通过 PodSpec 的 HostAliases 字段来添加这些自定义的条目。

建议通过使用 HostAliases 来进行修改，因为该文件由 Kubelet 管理，并且可以在 Pod 创建/重启过程中被重写。

* [**默认 hosts 文件内容**](https://kubernetes.io/zh/docs/concepts/services-networking/add-entries-to-pod-etc-hosts-with-host-aliases/#%e9%bb%98%e8%ae%a4-hosts-%e6%96%87%e4%bb%b6%e5%86%85%e5%ae%b9)
* [**通过 HostAliases 增加额外的条目**](https://kubernetes.io/zh/docs/concepts/services-networking/add-entries-to-pod-etc-hosts-with-host-aliases/#%e9%80%9a%e8%bf%87-hostaliases-%e5%a2%9e%e5%8a%a0%e9%a2%9d%e5%a4%96%e7%9a%84%e6%9d%a1%e7%9b%ae)
* [**为什么 Kubelet 管理 hosts文件？**](https://kubernetes.io/zh/docs/concepts/services-networking/add-entries-to-pod-etc-hosts-with-host-aliases/#%e4%b8%ba%e4%bb%80%e4%b9%88-kubelet-%e7%ae%a1%e7%90%86-hosts%e6%96%87%e4%bb%b6)

### 默认 hosts 文件内容

让我们从一个 Nginx Pod 开始，给该 Pod 分配一个 IP：

kubectl run nginx --image nginx --generator=run-pod/v1

pod/nginx created

检查Pod IP：

kubectl get pods --output=wide

NAME READY STATUS RESTARTS AGE IP NODE

nginx 1/1 Running 0 13s 10.200.0.4 worker0

主机文件的内容如下所示：

kubectl exec nginx -- cat /etc/hosts

# Kubernetes-managed hosts file.

127.0.0.1 localhost

::1 localhost ip6-localhost ip6-loopback

fe00::0 ip6-localnet

fe00::0 ip6-mcastprefix

fe00::1 ip6-allnodes

fe00::2 ip6-allrouters

10.200.0.4 nginx

默认，hosts 文件只包含 ipv4 和 ipv6 的样板内容，像 localhost 和主机名称。

### 通过 HostAliases 增加额外的条目

除了默认的样板内容，我们可以向 hosts 文件添加额外的条目，将 foo.local、 bar.local 解析为127.0.0.1， 将 foo.remote、 bar.remote 解析为 10.1.2.3，我们可以在 .spec.hostAliases 下为 Pod 添加 HostAliases。

| [service/networking/hostaliases-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/hostaliases-pod.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: hostaliases-pod  spec:  restartPolicy: Never  hostAliases:  - ip: "127.0.0.1"  hostnames:  - "foo.local"  - "bar.local"  - ip: "10.1.2.3"  hostnames:  - "foo.remote"  - "bar.remote"  containers:  - name: cat-hosts  image: busybox  command:  - cat  args:  - "/etc/hosts" |

可以使用以下命令启动此Pod：

kubectl apply -f hostaliases-pod.yaml

pod/hostaliases-pod created

检查Pod IP 和状态：

kubectl get pod --output=wide

NAME READY STATUS RESTARTS AGE IP NODE

hostaliases-pod 0/1 Completed 0 6s 10.200.0.5 worker0

hosts 文件的内容看起来类似如下这样：

kubectl logs hostaliases-pod

# Kubernetes-managed hosts file.

127.0.0.1 localhost

::1 localhost ip6-localhost ip6-loopback

fe00::0 ip6-localnet

fe00::0 ip6-mcastprefix

fe00::1 ip6-allnodes

fe00::2 ip6-allrouters

10.200.0.5 hostaliases-pod

# Entries added by HostAliases.

127.0.0.1 foo.local bar.local

10.1.2.3 foo.remote bar.remote

在最下面额外添加了一些条目。

### 为什么 Kubelet 管理 hosts文件？

kubelet [管理](https://github.com/kubernetes/kubernetes/issues/14633) Pod 中每个容器的 hosts 文件，避免 Docker 在容器已经启动之后去 [修改](https://github.com/moby/moby/issues/17190) 该文件。

因为该文件是托管性质的文件，无论容器重启或 Pod 重新调度，用户修改该 hosts 文件的任何内容，都会在 Kubelet 重新安装后被覆盖。因此，不建议修改该文件的内容。

### 反馈

## IPv4/IPv6 双协议栈

**FEATURE STATE:** Kubernetes v1.16 [alpha](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/)

IPv4/IPv6 双协议栈能够将 IPv4 和 IPv6 地址分配给 [Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 和 [Services](https://kubernetes.io/docs/concepts/services-networking/service/)。

如果你为 Kubernetes 集群启用了 IPv4/IPv6 双协议栈网络，则该集群将支持同时分配 IPv4 和 IPv6 地址。

* [**支持的功能**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e6%94%af%e6%8c%81%e7%9a%84%e5%8a%9f%e8%83%bd)
* [**先决条件**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e5%85%88%e5%86%b3%e6%9d%a1%e4%bb%b6)
* [**启用 IPv4/IPv6 双协议栈**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e5%90%af%e7%94%a8-ipv4-ipv6-%e5%8f%8c%e5%8d%8f%e8%ae%ae%e6%a0%88)
* [**服务**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e6%9c%8d%e5%8a%a1)
* [**出口流量**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e5%87%ba%e5%8f%a3%e6%b5%81%e9%87%8f)
* [**已知问题**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e5%b7%b2%e7%9f%a5%e9%97%ae%e9%a2%98)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 支持的功能

在 Kubernetes 集群上启用 IPv4/IPv6 双协议栈可提供下面的功能：

* 双协议栈 pod 网络 (每个 pod 分配一个 IPv4 和 IPv6 地址)
* IPv4 和 IPv6 启用的服务 (每个服务必须是一个单独的地址族)
* Kubenet 多地址族支持（IPv4 和 IPv6）
* Pod 的集群外出口通过 IPv4 和 IPv6 路由

### 先决条件

为了使用 IPv4/IPv6 双栈的 Kubernetes 集群，需要满足以下先决条件：

* Kubernetes 1.16 版本及更高版本
* 提供商支持双协议栈网络（云提供商或其他提供商必须能够为 Kubernetes 节点提供可路由的 IPv4/IPv6 网络接口）
* Kubenet 网络插件
* Kube-proxy 运行在 IPVS 模式

### 启用 IPv4/IPv6 双协议栈

要启用 IPv4/IPv6 双协议栈，为集群的相关组件启用 IPv6DualStack [特性门控](https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/)，并且设置双协议栈的集群网络分配：

* kube-controller-manager:
  + --feature-gates="IPv6DualStack=true"
  + --cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR> 例如 --cluster-cidr=10.244.0.0/16,fc00::/24
  + --service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>
  + --node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6 对于 IPv4 默认为 /24，对于 IPv6 默认为 /64
* kubelet:
  + --feature-gates="IPv6DualStack=true"
* kube-proxy:
  + --proxy-mode=ipvs
  + --cluster-cidrs=<IPv4 CIDR>,<IPv6 CIDR>
  + --feature-gates="IPv6DualStack=true"

**警告:**

如果命令行的 --cluster-cidr 指定大于 /24 的 IPv6 地址块，地址分配将失败。

### 服务

如果你的集群启用了 IPv4/IPv6 双协议栈网络，则可以使用 IPv4 或 IPv6 地址来创建 [Services](https://kubernetes.io/docs/concepts/services-networking/service/)。你可以通过设置服务的 .spec.ipFamily 字段来选择服务的集群 IP 的地址族。你只能在创建新服务时设置该字段。.spec.ipFamily 字段的设置是可选的，并且仅当你计划在集群上启用 IPv4 和 IPv6 的 [Services](https://kubernetes.io/docs/concepts/services-networking/service/) 和 [Ingresses](https://kubernetes.io/docs/concepts/services-networking/ingress/)。对于[出口](https://kubernetes.io/zh/docs/concepts/services-networking/dual-stack/#出口流量)流量，该字段的配置不是必须的。

**注意：**

集群的默认地址族是第一个服务集群 IP 范围的地址族，该地址范围通过 kube-controller-manager 上的 --service-cluster-ip-range 标志设置。

你可以设置 .spec.ipFamily 为：

* IPv4：API 服务器将从 service-cluster-ip-range 中分配 ipv4 地址
* IPv6：API 服务器将从 service-cluster-ip-range 中分配 ipv6 地址

以下服务规约不包含 ipFamily 字段。Kubernetes 将从最初配置的 service-cluster-ip-range 范围内分配一个 IP 地址（也称作“集群 IP”）给该服务。

| [service/networking/dual-stack-default-svc.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/dual-stack-default-svc.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-service  spec:  selector:  app: MyApp  ports:  - protocol: TCP  port: 80  targetPort: 9376 |

以下服务规约不包含 ipFamily 字段。Kubernetes 将从已配置的 service-cluster-ip-range 范围内分配一个 IPv6 地址（也称作“集群 IP”）给该服务。

| [service/networking/dual-stack-ipv6-svc.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/dual-stack-ipv6-svc.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-service  spec:  ipFamily: IPv6  selector:  app: MyApp  ports:  - protocol: TCP  port: 80  targetPort: 9376 |

为了进行比较，将从已配置的 service-cluster-ip-range 向该服务分配以下 IPV4 地址（也称为“集群 IP”）。

| [service/networking/dual-stack-ipv4-svc.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/service/networking/dual-stack-ipv4-svc.yaml) |
| --- |
| apiVersion: v1  kind: Service  metadata:  name: my-service  spec:  ipFamily: IPv4  selector:  app: MyApp  ports:  - protocol: TCP  port: 80  targetPort: 9376 |

#### 负载均衡器类型

在支持启用了 IPv6 的外部服务均衡器的云驱动上，除了将 ipFamily 字段设置为 IPv6，将 type 字段设置为 LoadBalancer，为你的服务提供云负载均衡。

### 出口流量

公共路由和非公共路由的 IPv6 地址块的使用是可以的。提供底层 [CNI](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/#cni) 的提供程序可以实现这种传输。如果你拥有使用非公共路由 IPv6 地址的 Pod，并且希望该 Pod 到达集群外目的（比如，公共网络），你必须为出口流量和任何响应消息设置 IP 伪装。[ip-masq-agent](https://github.com/kubernetes-incubator/ip-masq-agent) 可以感知双栈，所以你可以在双栈集群中使用 ip-masq-agent 来进行 IP 伪装。

### 已知问题

* Kubenet 强制 IPv4，IPv6 的 IPs 位置报告 (–cluster-cidr)

### 接下来

* [验证 IPv4/IPv6 双协议栈](https://kubernetes.io/docs/tasks/network/validate-dual-stack)网络

### 反馈

# 安全

## Overview of Cloud Native Security

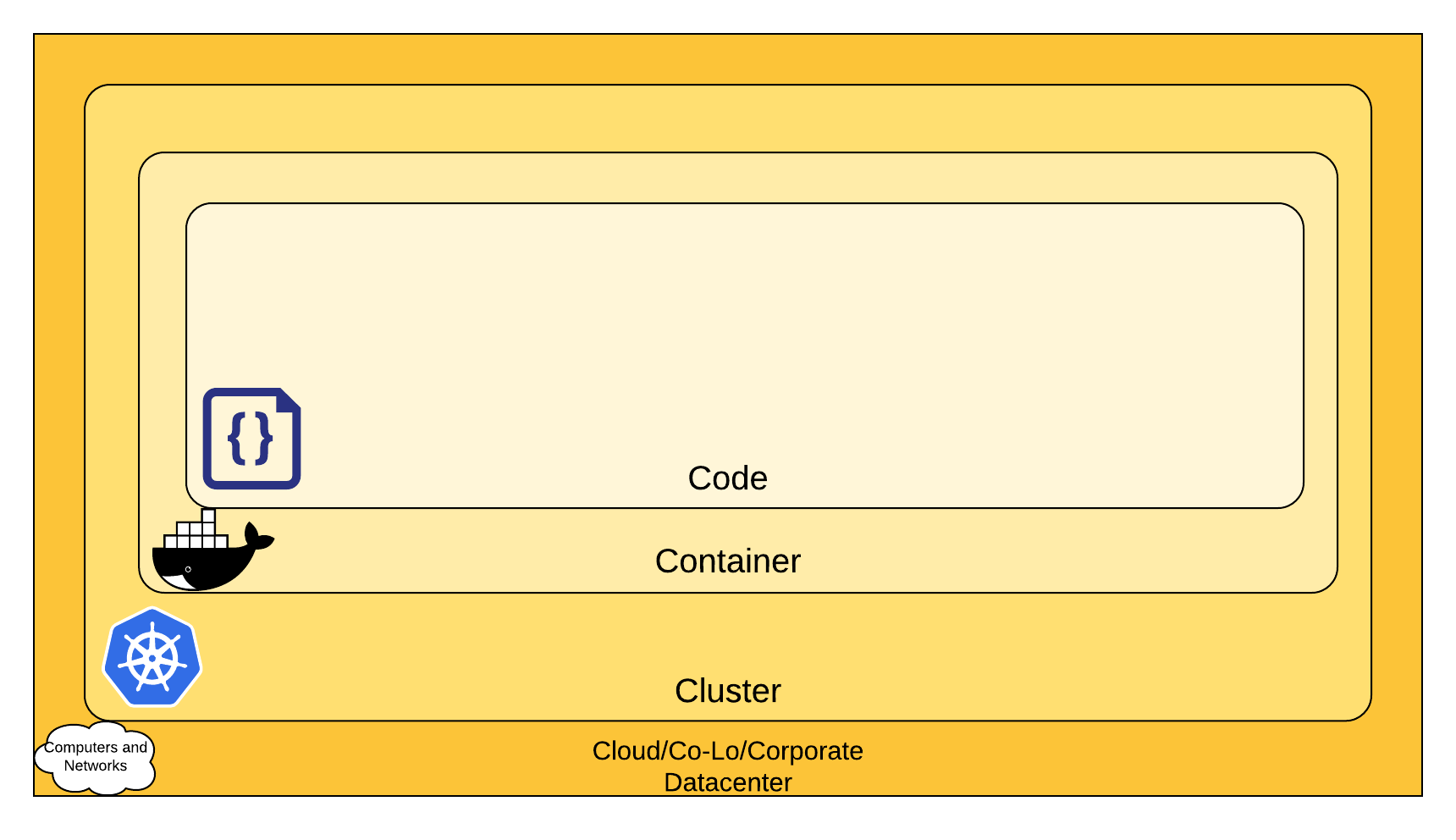
Kubernetes Security (and security in general) is an immense topic that has many highly interrelated parts. In today’s era where open source software is integrated into many of the systems that help web applications run, there are some overarching concepts that can help guide your intuition about how you can think about security holistically. This guide will define a mental model for some general concepts surrounding Cloud Native Security. The mental model is completely arbitrary and you should only use it if it helps you think about where to secure your software stack.

* [**The 4C’s of Cloud Native Security**](https://kubernetes.io/docs/concepts/security/overview/#the-4c-s-of-cloud-native-security)
* [**Cloud**](https://kubernetes.io/docs/concepts/security/overview/#cloud)
* [**Cluster**](https://kubernetes.io/docs/concepts/security/overview/#cluster)
* [**Container**](https://kubernetes.io/docs/concepts/security/overview/#container)
* [**Code**](https://kubernetes.io/docs/concepts/security/overview/#code)
* [**Robust automation**](https://kubernetes.io/docs/concepts/security/overview/#robust-automation)
* [**What's next**](https://kubernetes.io/docs/concepts/security/overview/#what-s-next)

### The 4C’s of Cloud Native Security

Let’s start with a diagram that may help you understand how you can think about security in layers.

**Note:** This layered approach augments the [defense in depth](https://en.wikipedia.org/wiki/Defense_in_depth_(computing)) approach to security, which is widely regarded as a best practice for securing software systems. The 4C’s are Cloud, Clusters, Containers, and Code.



##### The 4C's of Cloud Native Security

As you can see from the above figure, each one of the 4C’s depend on the security of the squares in which they fit. It is nearly impossibly to safeguard against poor security standards in Cloud, Containers, and Code by only addressing security at the code level. However, when these areas are dealt with appropriately, then adding security to your code augments an already strong base. These areas of concern will now be described in more detail below.

### Cloud

In many ways, the Cloud (or co-located servers, or the corporate datacenter) is the [trusted computing base](https://en.wikipedia.org/wiki/Trusted_computing_base) of a Kubernetes cluster. If these components themselves are vulnerable (or configured in a vulnerable way) then there’s no real way to guarantee the security of any components built on top of this base. Each cloud provider has extensive security recommendations they make to their customers on how to run workloads securely in their environment. It is out of the scope of this guide to give recommendations on cloud security since every cloud provider and workload is different. Here are some links to some of the popular cloud providers’ documentation for security as well as give general guidance for securing the infrastructure that makes up a Kubernetes cluster.

#### Cloud Provider Security Table

| IaaS Provider | Link |
| --- | --- |
| Alibaba Cloud | <https://www.alibabacloud.com/trust-center> |
| Amazon Web Services | <https://aws.amazon.com/security/> |
| Google Cloud Platform | <https://cloud.google.com/security/> |
| IBM Cloud | <https://www.ibm.com/cloud/security> |
| Microsoft Azure | <https://docs.microsoft.com/en-us/azure/security/azure-security> |
| VMWare VSphere | <https://www.vmware.com/security/hardening-guides.html> |

If you are running on your own hardware or a different cloud provider you will need to consult your documentation for security best practices.

#### General Infrastructure Guidance Table

| Area of Concern for Kubernetes Infrastructure | Recommendation |
| --- | --- |
| Network access to API Server (Masters) | Ideally all access to the Kubernetes Masters is not allowed publicly on the internet and is controlled by network access control lists restricted to the set of IP addresses needed to administer the cluster. |
| Network access to Nodes (Worker Servers) | Nodes should be configured to only accept connections (via network access control lists) from the masters on the specified ports, and accept connections for services in Kubernetes of type NodePort and LoadBalancer. If possible, these nodes should not be exposed on the public internet entirely. |
| Kubernetes access to Cloud Provider API | Each cloud provider will need to grant a different set of permissions to the Kubernetes Masters and Nodes, so this recommendation will be more generic. It is best to provide the cluster with cloud provider access that follows the [principle of least privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege) for the resources it needs to administer. An example for Kops in AWS can be found here: <https://github.com/kubernetes/kops/blob/master/docs/iam_roles.md#iam-roles> |
| Access to etcd | Access to etcd (the datastore of Kubernetes) should be limited to the masters only. Depending on your configuration, you should also attempt to use etcd over TLS. More info can be found here: <https://github.com/etcd-io/etcd/tree/master/Documentation#security> |
| etcd Encryption | Wherever possible it’s a good practice to encrypt all drives at rest, but since etcd holds the state of the entire cluster (including Secrets) its disk should especially be encrypted at rest. |

### Cluster

This section will provide links for securing workloads in Kubernetes. There are two areas of concern for securing Kubernetes:

* Securing the components that are configurable which make up the cluster
* Securing the components which run in the cluster

#### Components of the Cluster

If you want to protect your cluster from accidental or malicious access, and adopt good information practices, read and follow the advice about [securing your cluster](https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/).

#### Components in the Cluster (your application)

Depending on the attack surface of your application, you may want to focus on specific aspects of security. For example, if you are running a service (Service A) that is critical in a chain of other resources and a separate workload (Service B) which is vulnerable to a resource exhaustion attack, by not putting resource limits on Service B you run the risk of also compromising Service A. Below is a table of links of things to consider when securing workloads running in Kubernetes.

| Area of Concern for Workload Security | Recommendation |
| --- | --- |
| RBAC Authorization (Access to the Kubernetes API) | <https://kubernetes.io/docs/reference/access-authn-authz/rbac/> |
| Authentication | <https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/> |
| Application secrets management (and encrypting them in etcd at rest) | <https://kubernetes.io/docs/concepts/configuration/secret/> <https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/> |
| Pod Security Policies | <https://kubernetes.io/docs/concepts/policy/pod-security-policy/> |
| Quality of Service (and Cluster resource management) | <https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/> |
| Network Policies | <https://kubernetes.io/docs/concepts/services-networking/network-policies/> |
| TLS For Kubernetes Ingress | <https://kubernetes.io/docs/concepts/services-networking/ingress/#tls> |

### Container

In order to run software in Kubernetes, it must be in a container. Because of this, there are certain security considerations that must be taken into account in order to benefit from the workload security primitives of Kubernetes. Container security is also outside the scope of this guide, but here is a table of general recommendations and links for further exploration of this topic.

| Area of Concern for Containers | Recommendation |
| --- | --- |
| Container Vulnerability Scanning and OS Dependency Security | As part of an image build step or on a regular basis you should scan your containers for known vulnerabilities with a tool such as [CoreOS’s Clair](https://github.com/coreos/clair/) |
| Image Signing and Enforcement | Two other CNCF Projects (TUF and Notary) are useful tools for signing container images and maintaining a system of trust for the content of your containers. If you use Docker, it is built in to the Docker Engine as [Docker Content Trust](https://docs.docker.com/engine/security/trust/content_trust/). On the enforcement piece, [IBM’s Portieris](https://github.com/IBM/portieris) project is a tool that runs as a Kubernetes Dynamic Admission Controller to ensure that images are properly signed via Notary before being admitted to the Cluster. |
| Disallow privileged users | When constructing containers, consult your documentation for how to create users inside of the containers that have the least level of operating system privilege necessary in order to carry out the goal of the container. |

### Code

Finally moving down into the application code level, this is one of the primary attack surfaces over which you have the most control. This is also outside of the scope of Kubernetes but here are a few recommendations:

#### General Code Security Guidance Table

| Area of Concern for Code | Recommendation |
| --- | --- |
| Access over TLS only | If your code needs to communicate via TCP, ideally it would be performing a TLS handshake with the client ahead of time. With the exception of a few cases, the default behavior should be to encrypt everything in transit. Going one step further, even “behind the firewall” in our VPC’s it’s still a good idea to encrypt network traffic between services. This can be done through a process known as mutual or [mTLS](https://en.wikipedia.org/wiki/Mutual_authentication) which performs a two sided verification of communication between two certificate holding services. There are numerous tools that can be used to accomplish this in Kubernetes such as [Linkerd](https://linkerd.io/) and [Istio](https://istio.io/). |
| Limiting port ranges of communication | This recommendation may be a bit self-explanatory, but wherever possible you should only expose the ports on your service that are absolutely essential for communication or metric gathering. |
| 3rd Party Dependency Security | Since our applications tend to have dependencies outside of our own codebases, it is a good practice to ensure that a regular scan of the code’s dependencies are still secure with no CVE’s currently filed against them. Each language has a tool for performing this check automatically. |
| Static Code Analysis | Most languages provide a way for a snippet of code to be analyzed for any potentially unsafe coding practices. Whenever possible you should perform checks using automated tooling that can scan codebases for common security errors. Some of the tools can be found here: <https://www.owasp.org/index.php/Source_Code_Analysis_Tools> |
| Dynamic probing attacks | There are a few automated tools that are able to be run against your service to try some of the well known attacks that commonly befall services. These include SQL injection, CSRF, and XSS. One of the most popular dynamic analysis tools is the OWASP Zed Attack proxy <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project> |

### Robust automation

Most of the above mentioned suggestions can actually be automated in your code delivery pipeline as part of a series of checks in security. To learn about a more “Continuous Hacking” approach to software delivery, [this article](https://thenewstack.io/beyond-ci-cd-how-continuous-hacking-of-docker-containers-and-pipeline-driven-security-keeps-ygrene-secure/) provides more detail.

### What's next

* Read about [network policies for Pods](https://kubernetes.io/docs/concepts/services-networking/network-policies/)
* Read about [securing your cluster](https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/)
* Read about [API access control](https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/)
* Read about [data encryption in transit](https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/) for the control plane
* Read about [data encryption at rest](https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/)
* Read about [Secrets in Kubernetes](https://kubernetes.io/docs/concepts/configuration/secret/)

### Feedback

## 安全

# 调度

## Kubernetes 调度器

在 Kubernetes 中，调度 是指将 [Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/) 放置到合适的 [Node](https://kubernetes.io/docs/concepts/architecture/nodes/) 上，然后对应 Node 上的 [Kubelet](https://kubernetes.io/docs/reference/generated/kubelet) 才能够运行这些 pod。

* [**调度概览**](https://kubernetes.io/zh/docs/concepts/scheduling/kube-scheduler/#scheduling)
* [**kube-scheduler**](https://kubernetes.io/zh/docs/concepts/scheduling/kube-scheduler/#kube-scheduler)
* [**kube-scheduler 调度流程**](https://kubernetes.io/zh/docs/concepts/scheduling/kube-scheduler/#kube-scheduler-implementation)
* [**接下来**](https://kubernetes.io/zh/docs/concepts/scheduling/kube-scheduler/#%e6%8e%a5%e4%b8%8b%e6%9d%a5)

### 调度概览

调度器通过 kubernetes 的 watch 机制来发现集群中新创建且尚未被调度到 Node 上的 Pod。调度器会将发现的每一个未调度的 Pod 调度到一个合适的 Node 上来运行。调度器会依据下文的调度原则来做出调度选择。

如果你想要理解 Pod 为什么会被调度到特定的 Node 上，或者你想要尝试实现一个自定义的调度器，这篇文章将帮助你了解调度。

### kube-scheduler

[kube-scheduler](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/) 是 Kubernetes 集群的默认调度器，并且是集群 [控制面](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-control-plane) 的一部分。如果你真的希望或者有这方面的需求，kube-scheduler 在设计上是允许你自己写一个调度组件并替换原有的 kube-scheduler。

对每一个新创建的 Pod 或者是未被调度的 Pod，kube-scheduler 会选择一个最优的 Node 去运行这个 Pod。然而，Pod 内的每一个容器对资源都有不同的需求，而且 Pod 本身也有不同的资源需求。因此，Pod 在被调度到 Node 上之前，根据这些特定的资源调度需求，需要对集群中的 Node 进行一次过滤。

在一个集群中，满足一个 Pod 调度请求的所有 Node 称之为 \_可调度节点\_。如果没有任何一个 Node 能满足 Pod 的资源请求，那么这个 Pod 将一直停留在未调度状态直到调度器能够找到合适的 Node。

调度器先在集群中找到一个 Pod 的所有可调度节点，然后根据一系列函数对这些可调度节点打分，然后选出其中得分最高的 Node 来运行 Pod。之后，调度器将这个调度决定通知给 kube-apiserver，这个过程叫做 \_绑定\_。

在做调度决定时需要考虑的因素包括：单独和整体的资源请求、硬件/软件/策略限制、亲和以及反亲和要求、数据局域性、负载间的干扰等等。

### kube-scheduler 调度流程

kube-scheduler 给一个 pod 做调度选择包含两个步骤：

1. 过滤
2. 打分

过滤阶段会将所有满足 Pod 调度需求的 Node 选出来。例如，PodFitsResources 过滤函数会检查候选 Node 的可用资源能否满足 Pod 的资源请求。在过滤之后，得出一个 Node 列表，里面包含了所有可调度节点；通常情况下，这个 Node 列表包含不止一个 Node。如果这个列表是空的，代表这个 Pod 不可调度。

在打分阶段，调度器会为 Pod 从所有可调度节点中选取一个最合适的 Node。根据当前启用的打分规则，调度器会给每一个可调度节点进行打分。

最后，kube-scheduler 会将 Pod 调度到得分最高的 Node 上。如果存在多个得分最高的 Node，kube-scheduler 会从中随机选取一个。

#### 默认策略

kube-scheduler 有一系列的默认调度策略。

#### 过滤策略

* PodFitsHostPorts：如果 Pod 中定义了 hostPort 属性，那么需要先检查这个指定端口是否 已经被 Node 上其他服务占用了。
* PodFitsHost：若 pod 对象拥有 hostname 属性，则检查 Node 名称字符串与此属性是否匹配。
* PodFitsResources：检查 Node 上是否有足够的资源（如，cpu 和内存）来满足 pod 的资源请求。
* PodMatchNodeSelector：检查 Node 的 [标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels) 是否能匹配 Pod 属性上 Node 的 [标签](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels) 值。
* NoVolumeZoneConflict：检测 pod 请求的 [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/) 在 Node 上是否可用，因为某些存储卷存在区域调度约束。
* NoDiskConflict：检查 Pod 对象请求的存储卷在 Node 上是否可用，若不存在冲突则通过检查。
* MaxCSIVolumeCount：检查 Node 上已经挂载的 [CSI](https://kubernetes.io/docs/concepts/storage/volumes/#csi) 存储卷数量是否超过了指定的最大值。
* CheckNodeMemoryPressure：如果 Node 上报了内存资源压力过大，而且没有配置异常，那么 Pod 将不会被调度到这个 Node 上。
* CheckNodePIDPressure：如果 Node 上报了 PID 资源压力过大，而且没有配置异常，那么 Pod 将不会被调度到这个 Node 上。
* CheckNodeDiskPressure：如果 Node 上报了磁盘资源压力过大（文件系统满了或者将近满了）， 而且配置异常，那么 Pod 将不会被调度到这个 Node 上。
* CheckNodeCondition：Node 可以上报其自身的状态，如磁盘、网络不可用，表明 kubelet 未准备好运行 pod。 如果 Node 被设置成这种状态，那么 pod 将不会被调度到这个 Node 上。
* PodToleratesNodeTaints：检查 pod 属性上的 [tolerations](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/) 能否容忍 Node 的 [taints](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/)。
* CheckVolumeBinding：检查 Node 上已经绑定的和未绑定的 [PVCs](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) 能否满足 Pod 对象的存储卷需求。

#### 打分策略

* SelectorSpreadPriority：尽量将归属于同一个 [Service](https://kubernetes.io/docs/concepts/services-networking/service/)、[StatefulSet](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/) 或 [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) 的 Pod 资源分散到不同的 Node 上。
* InterPodAffinityPriority：遍历 Pod 对象的亲和性条目，并将那些能够匹配到给定 Node 的条目的权重相加，结果值越大的 Node 得分越高。
* LeastRequestedPriority：空闲资源比例越高的 Node 得分越高。换句话说，Node 上的 Pod 越多，并且资源被占用的越多，那么这个 Node 的得分就会越少。
* MostRequestedPriority：空闲资源比例越低的 Node 得分越高。这个调度策略将会把你所有的工作负载（Pod）调度到尽量少的 Node 上。
* RequestedToCapacityRatioPriority：为 Node 上每个资源占用比例设定得分值，给资源打分函数在打分时使用。
* BalancedResourceAllocation：优选那些使得资源利用率更为均衡的节点。
* NodePreferAvoidPodsPriority：这个策略将根据 Node 的注解信息中是否含有 scheduler.alpha.kubernetes.io/preferAvoidPods 来 计算其优先级。使用这个策略可以将两个不同 Pod 运行在不同的 Node 上。
* NodeAffinityPriority：基于 Pod 属性中 PreferredDuringSchedulingIgnoredDuringExecution 来进行 Node 亲和性调度。你可以通过这篇文章 [Pods 到 Nodes 的分派](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/) 来了解到更详细的内容。
* TaintTolerationPriority：基于 Pod 中对每个 Node 上污点容忍程度进行优先级评估，这个策略能够调整待选 Node 的排名。
* ImageLocalityPriority：Node 上已经拥有 Pod 需要的 [容器镜像](https://kubernetes.io/zh/docs/reference/glossary/?all=true#term-image) 的 Node 会有较高的优先级。
* ServiceSpreadingPriority：这个调度策略的主要目的是确保将归属于同一个 Service 的 Pod 调度到不同的 Node 上。如果 Node 上 没有归属于同一个 Service 的 Pod，这个策略更倾向于将 Pod 调度到这类 Node 上。最终的目的：即使在一个 Node 宕机之后 Service 也具有很强容灾能力。
* CalculateAntiAffinityPriorityMap：这个策略主要是用来实现[pod反亲和](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity)。
* EqualPriorityMap：将所有的 Node 设置成相同的权重为 1。

### 接下来

* 阅读关于 [调度器性能调优](https://kubernetes.io/docs/concepts/scheduling/scheduler-perf-tuning/)
* 阅读关于 [Pod 拓扑分布约束](https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/)
* 阅读关于 kube-scheduler 的 [参考文档](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/)
* 了解关于 [配置多个调度器](https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/) 的方式
* 了解关于 [拓扑结构管理策略](https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/)
* 了解关于 [Pod 额外开销](https://kubernetes.io/docs/concepts/configuration/pod-overhead/)

### 反馈

## 调度器性能调优

**FEATURE STATE:** Kubernetes 1.14 [beta](https://kubernetes.io/zh/docs/concepts/scheduling/scheduler-perf-tuning/)

作为 kubernetes 集群的默认调度器，kube-scheduler 主要负责将 Pod 调度到集群的 Node 上。

在一个集群中，满足一个 Pod 调度请求的所有 Node 称之为 可调度 Node。调度器先在集群中找到一个 Pod 的可调度 Node，然后根据一系列函数对这些可调度 Node打分，之后选出其中得分最高的 Node 来运行 Pod。最后，调度器将这个调度决定告知 kube-apiserver，这个过程叫做 \_绑定\_。

这篇文章将会介绍一些在大规模 Kubernetes 集群下调度器性能优化的方式。

* [**设置打分阶段 Node 数量占集群总规模的百分比**](https://kubernetes.io/zh/docs/concepts/scheduling/scheduler-perf-tuning/#%e8%ae%be%e7%bd%ae%e6%89%93%e5%88%86%e9%98%b6%e6%ae%b5-node-%e6%95%b0%e9%87%8f%e5%8d%a0%e9%9b%86%e7%be%a4%e6%80%bb%e8%a7%84%e6%a8%a1%e7%9a%84%e7%99%be%e5%88%86%e6%af%94)

### 设置打分阶段 Node 数量占集群总规模的百分比

在 Kubernetes 1.12 版本之前，kube-scheduler 会检查集群中所有节点的可调度性，并且给可调度节点打分。Kubernetes 1.12 版本添加了一个新的功能，允许调度器在找到一定数量的可调度节点之后就停止继续寻找可调度节点。该功能能提高调度器在大规模集群下的调度性能。这个数值是集群规模的百分比。这个百分比通过 percentageOfNodesToScore 参数来进行配置。其值的范围在 1 到 100 之间，最大值就是 100%。如果设置为 0 就代表没有提供这个参数配置。Kubernetes 1.14 版本又加入了一个特性，在该参数没有被用户配置的情况下，调度器会根据集群的规模自动设置一个集群比例，然后通过这个比例筛选一定数量的可调度节点进入打分阶段。该特性使用线性公式计算出集群比例，如在 100-node 的集群下取 50%。在 5000-node 的集群下取 10%。这个自动设置的参数的最低值是 5%。换句话说，调度器至少会对集群中 5% 的节点进行打分，除非用户将该参数设置的低于 5。

下面就是一个将 percentageOfNodesToScore 参数设置为 50% 的例子。

apiVersion: kubescheduler.config.k8s.io/v1alpha1

kind: KubeSchedulerConfiguration

algorithmSource:

provider: DefaultProvider

...

percentageOfNodesToScore: 50

**注意：** 当集群中的可调度节点少于 50 个时，调度器仍然会去检查所有的 Node，因为可调度节点太少，不足以停止调度器最初的过滤选择。

**如果想要关闭这个功能**，你可以将 percentageOfNodesToScore 值设置成 100。

#### 调节 percentageOfNodesToScore 参数

percentageOfNodesToScore 的值必须在 1 到 100 之间，而且其默认值是通过集群的规模计算得来的。另外，还有一个 50 个 Node 的数值是硬编码在程序里面的。设置这个值的作用在于：当集群的规模是数百个 Node 并且 percentageOfNodesToScore 参数设置的过低的时候，调度器筛选到的可调度节点数目基本不会受到该参数影响。当集群规模较小时，这个设置将导致调度器性能提升并不明显。然而在一个超过 1000 个 Node 的集群中，将调优参数设置为一个较低的值可以很明显的提升调度器性能。

值得注意的是，该参数设置后可能会导致只有集群中少数节点被选为可调度节点，很多 node 都没有进入到打分阶段。这样就会造成一种后果，一个本来可以在打分阶段得分很高的 Node 甚至都不能进入打分阶段。由于这个原因，这个参数不应该被设置成一个很低的值。通常的做法是不会将这个参数的值设置的低于 10。很低的参数值一般在调度器的吞吐量很高且对 node 的打分不重要的情况下才使用。换句话说，只有当你更倾向于在可调度节点中任意选择一个 Node 来运行这个 Pod 时，才使用很低的参数设置。

如果你的集群规模只有数百个节点或者更少，我们并不推荐你将这个参数设置得比默认值更低。因为这种情况下不太可能有效的提高调度器性能。

#### 调度器做调度选择的时候如何覆盖所有的 Node

如果你想要理解这一个特性的内部细节，那么请仔细阅读这一章节。

在将 Pod 调度到 Node 上时，为了让集群中所有 Node 都有公平的机会去运行这些 Pod，调度器将会以轮询的方式覆盖全部的 Node。你可以将 Node 列表想象成一个数组。调度器从数组的头部开始筛选可调度节点，依次向后直到可调度节点的数量达到 percentageOfNodesToScore 参数的要求。在对下一个 Pod 进行调度的时候，前一个 Pod 调度筛选停止的 Node 列表的位置，将会来作为这次调度筛选 Node 开始的位置。

如果集群中的 Node 在多个区域，那么调度器将从不同的区域中轮询 Node，来确保不同区域的 Node 接受可调度性检查。如下例，考虑两个区域中的六个节点：

Zone 1: Node 1, Node 2, Node 3, Node 4

Zone 2: Node 5, Node 6

调度器将会按照如下的顺序去评估 Node 的可调度性：

Node 1, Node 5, Node 2, Node 6, Node 3, Node 4

在评估完所有 Node 后，将会返回到 Node 1，从头开始。

### 反馈

# 策略

## Pod 安全策略

PodSecurityPolicy 类型的对象能够控制，是否可以向 Pod 发送请求，该 Pod 能够影响被应用到 Pod 和容器的 SecurityContext。 查看 [Pod 安全策略建议](https://git.k8s.io/community/contributors/design-proposals/security-context-constraints.md) 获取更多信息。

* + [**什么是 Pod 安全策略？**](https://kubernetes.io/zh/docs/concepts/policy/#什么是-pod-安全策略)
    - [**RunAsUser**](https://kubernetes.io/zh/docs/concepts/policy/#runasuser)
    - [**SELinux**](https://kubernetes.io/zh/docs/concepts/policy/#selinux)
    - [**SupplementalGroups**](https://kubernetes.io/zh/docs/concepts/policy/#supplementalgroups)
    - [**FSGroup**](https://kubernetes.io/zh/docs/concepts/policy/#fsgroup)
    - [**控制卷**](https://kubernetes.io/zh/docs/concepts/policy/#控制卷)
    - [**主机网络**](https://kubernetes.io/zh/docs/concepts/policy/#主机网络)
    - [**允许的主机路径**](https://kubernetes.io/zh/docs/concepts/policy/#允许的主机路径)
  + [**许可**](https://kubernetes.io/zh/docs/concepts/policy/#许可)
    - [**创建一个策略和一个 Pod**](https://kubernetes.io/zh/docs/concepts/policy/#创建一个策略和一个-pod)
  + [**获取 Pod 安全策略列表**](https://kubernetes.io/zh/docs/concepts/policy/#获取-pod-安全策略列表)
  + [**修改 Pod 安全策略**](https://kubernetes.io/zh/docs/concepts/policy/#修改-pod-安全策略)
  + [**删除 Pod 安全策略**](https://kubernetes.io/zh/docs/concepts/policy/#删除-pod-安全策略)
  + [**启用 Pod 安全策略**](https://kubernetes.io/zh/docs/concepts/policy/#启用-pod-安全策略)
  + [**使用 RBAC**](https://kubernetes.io/zh/docs/concepts/policy/#使用-rbac)

### 什么是 Pod 安全策略？

Pod 安全策略 是集群级别的资源，它能够控制 Pod 运行的行为，以及它具有访问什么的能力。 PodSecurityPolicy 对象定义了一组条件，指示 Pod 必须按系统所能接受的顺序运行。 它们允许管理员控制如下方面：

| 控制面 | 字段名称 |
| --- | --- |
| 已授权容器的运行 | privileged |
| 为容器添加默认的一组能力 | defaultAddCapabilities |
| 为容器去掉某些能力 | requiredDropCapabilities |
| 容器能够请求添加某些能力 | allowedCapabilities |
| 控制卷类型的使用 | [volumes](https://kubernetes.io/zh/docs/concepts/policy/#controlling-volumes) |
| 主机网络的使用 | [hostNetwork](https://kubernetes.io/zh/docs/concepts/policy/#host-network) |
| 主机端口的使用 | hostPorts |
| 主机 PID namespace 的使用 | hostPID |
| 主机 IPC namespace 的使用 | hostIPC |
| 主机路径的使用 | [allowedHostPaths](https://kubernetes.io/zh/docs/concepts/policy/#allowed-host-paths) |
| 容器的 SELinux 上下文 | [seLinux](https://kubernetes.io/zh/docs/concepts/policy/#selinux) |
| 用户 ID | [runAsUser](https://kubernetes.io/zh/docs/concepts/policy/#runasuser) |
| 配置允许的补充组 | [supplementalGroups](https://kubernetes.io/zh/docs/concepts/policy/#supplementalgroups) |
| 分配拥有 Pod 数据卷的 FSGroup | [fsGroup](https://kubernetes.io/zh/docs/concepts/policy/#fsgroup) |
| 必须使用一个只读的 root 文件系统 | readOnlyRootFilesystem |

Pod 安全策略 由设置和策略组成，它们能够控制 Pod 访问的安全特征。这些设置分为如下三类：

* 基于布尔值控制 ：这种类型的字段默认为最严格限制的值。
* 基于被允许的值集合控制 ：这种类型的字段会与这组值进行对比，以确认值被允许。
* 基于策略控制 ：设置项通过一种策略提供的机制来生成该值，这种机制能够确保指定的值落在被允许的这组值中。

#### RunAsUser

* MustRunAs - 必须配置一个 range。使用该范围内的第一个值作为默认值。验证是否不在配置的该范围内。
* MustRunAsNonRoot - 要求提交的 Pod 具有非零 runAsUser 值，或在镜像中定义了 USER 环境变量。不提供默认值。
* RunAsAny - 没有提供默认值。允许指定任何 runAsUser 。

#### SELinux

* MustRunAs - 如果没有使用预分配的值，必须配置 seLinuxOptions。默认使用 seLinuxOptions。验证 seLinuxOptions。
* RunAsAny - 没有提供默认值。允许任意指定的 seLinuxOptions ID。

#### SupplementalGroups

* MustRunAs - 至少需要指定一个范围。默认使用第一个范围的最小值。验证所有范围的值。
* RunAsAny - 没有提供默认值。允许任意指定的 supplementalGroups ID。

#### FSGroup

* MustRunAs - 至少需要指定一个范围。默认使用第一个范围的最小值。验证在第一个范围内的第一个 ID。
* RunAsAny - 没有提供默认值。允许任意指定的 fsGroup ID。

#### 控制卷

通过设置 PSP 卷字段，能够控制具体卷类型的使用。当创建一个卷的时候，与该字段相关的已定义卷可以允许设置如下值：

1. azureFile
2. azureDisk
3. flocker
4. flexVolume
5. hostPath
6. emptyDir
7. gcePersistentDisk
8. awsElasticBlockStore
9. gitRepo
10. secret
11. nfs
12. iscsi
13. glusterfs
14. persistentVolumeClaim
15. rbd
16. cinder
17. cephFS
18. downwardAPI
19. fc
20. configMap
21. vsphereVolume
22. quobyte
23. projected
24. portworxVolume
25. scaleIO
26. storageos
27. \* (allow all volumes)

对新的 PSP，推荐允许的卷的最小集合包括：configMap、downwardAPI、emptyDir、persistentVolumeClaim、secret 和 projected。

#### 主机网络

* HostPorts ， 默认为 empty。HostPortRange 列表通过 min(包含) and max(包含) 来定义，指定了被允许的主机端口。

#### 允许的主机路径

* AllowedHostPaths 是一个被允许的主机路径前缀的白名单。空值表示所有的主机路径都可以使用。

### 许可

包含 PodSecurityPolicy 的 \_许可控制\_，允许控制集群资源的创建和修改，基于这些资源在集群范围内被许可的能力。

许可使用如下的方式为 Pod 创建最终的安全上下文： 1. 检索所有可用的 PSP。 1. 生成在请求中没有指定的安全上下文设置的字段值。 1. 基于可用的策略，验证最终的设置。

如果某个策略能够匹配上，该 Pod 就被接受。如果请求与 PSP 不匹配，则 Pod 被拒绝。

Pod 必须基于 PSP 验证每个字段。

#### 创建一个策略和一个 Pod

在一个文件中定义 PodSecurityPolicy 对象实例。这里的策略只是用来禁止创建有特权 要求的 Pods。

| [policy/example-psp.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/zh/examples/policy/example-psp.yaml) |
| --- |
| apiVersion: policy/v1beta1  kind: PodSecurityPolicy  metadata:  name: example  spec:  privileged: **false** *# Don't allow privileged pods!*  *# The rest fills in some required fields.*  seLinux:  rule: RunAsAny  supplementalGroups:  rule: RunAsAny  runAsUser:  rule: RunAsAny  fsGroup:  rule: RunAsAny  volumes:  - '\*' |

使用 kubectl 执行创建操作：

kubectl-admin create -f example-psp.yaml

### 获取 Pod 安全策略列表

获取已存在策略列表，使用 kubectl get：

$ kubectl get psp

NAME PRIV CAPS SELINUX RUNASUSER FSGROUP SUPGROUP READONLYROOTFS VOLUMES

permissive false [] RunAsAny RunAsAny RunAsAny RunAsAny false [\*]

privileged true [] RunAsAny RunAsAny RunAsAny RunAsAny false [\*]

restricted false [] RunAsAny MustRunAsNonRoot RunAsAny RunAsAny false [emptyDir secret downwardAPI configMap persistentVolumeClaim projected]

### 修改 Pod 安全策略

通过交互方式修改策略，使用 kubectl edit：

$ kubectl edit psp permissive

该命令将打开一个默认文本编辑器，在这里能够修改策略。

### 删除 Pod 安全策略

一旦不再需要一个策略，很容易通过 kubectl 删除它：

$ kubectl delete psp permissive

podsecuritypolicy "permissive" deleted

### 启用 Pod 安全策略

为了能够在集群中使用 Pod 安全策略，必须确保满足如下条件：

1. 已经启用 API 类型 extensions/v1beta1/podsecuritypolicy（仅对 1.6 之前的版本）
2. 已经启用许可控制器 PodSecurityPolicy
3. 已经定义了自己的策略

### 使用 RBAC

在 Kubernetes 1.5 或更新版本，可以使用 PodSecurityPolicy 来控制，对基于用户角色和组的已授权容器的访问。访问不同的 PodSecurityPolicy 对象，可以基于认证来控制。基于 Deployment、ReplicaSet 等创建的 Pod，限制访问 PodSecurityPolicy 对象，[Controller Manager](https://kubernetes.io/docs/admin/kube-controller-manager/) 必须基于安全 API 端口运行，并且不能够具有超级用户权限。

PodSecurityPolicy 认证使用所有可用的策略，包括创建 Pod 的用户，Pod 上指定的服务账户（Service Account）。当 Pod 基于 Deployment、ReplicaSet 创建时，它是创建 Pod 的 Controller Manager，所以如果基于非安全 API 端口运行，允许所有的 PodSecurityPolicy 对象，并且不能够有效地实现细分权限。用户访问给定的 PSP 策略有效，仅当是直接部署 Pod 的情况。更多详情，查看 [PodSecurityPolicy RBAC 示例](https://git.k8s.io/kubernetes/examples/podsecuritypolicy/rbac/README.md)，当直接部署 Pod 时，应用 PodSecurityPolicy 控制基于角色和组的已授权容器的访问 。

### 反馈

## Limit Ranges

By default, containers run with unbounded [compute resources](https://kubernetes.io/docs/user-guide/compute-resources) on a Kubernetes cluster. With Resource quotas, cluster administrators can restrict the resource consumption and creation on a namespace basis. Within a namespace, a Pod or Container can consume as much CPU and memory as defined by the namespace’s resource quota. There is a concern that one Pod or Container could monopolize all of the resources. Limit Range is a policy to constrain resource by Pod or Container in a namespace.

* [**Enabling Limit Range**](https://kubernetes.io/docs/concepts/policy/limit-range/#enabling-limit-range)
* [**Limiting Container compute resources**](https://kubernetes.io/docs/concepts/policy/limit-range/#limiting-container-compute-resources)
* [**Limiting Pod compute resources**](https://kubernetes.io/docs/concepts/policy/limit-range/#limiting-pod-compute-resources)
* [**Limiting Storage resources**](https://kubernetes.io/docs/concepts/policy/limit-range/#limiting-storage-resources)
* [**Limits/Requests Ratio**](https://kubernetes.io/docs/concepts/policy/limit-range/#limits-requests-ratio)
* [**Examples**](https://kubernetes.io/docs/concepts/policy/limit-range/#examples)
* [**What's next**](https://kubernetes.io/docs/concepts/policy/limit-range/#what-s-next)

A limit range, defined by a LimitRange object, provides constraints that can:

* Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
* Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
* Enforce a ratio between request and limit for a resource in a namespace.
* Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

### Enabling Limit Range

Limit Range support is enabled by default for many Kubernetes distributions. It is enabled when the apiserver --enable-admission-plugins= flag has LimitRanger admission controller as one of its arguments.

A limit range is enforced in a particular namespace when there is a LimitRange object in that namespace.

#### Overview of Limit Range:

* The administrator creates one LimitRange in one namespace.
* Users create resources like Pods, Containers, and PersistentVolumeClaims in the namespace.
* The LimitRanger admission controller enforces defaults limits for all Pods and Container that do not set compute resource requirements and tracks usage to ensure it does not exceed resource minimum , maximum and ratio defined in any LimitRange present in the namespace.
* If creating or updating a resource (Pod, Container, PersistentVolumeClaim) violates a limit range constraint, the request to the API server will fail with HTTP status code 403 FORBIDDEN and a message explaining the constraint that would have been violated.
* If limit range is activated in a namespace for compute resources like cpu and memory, users must specify requests or limits for those values; otherwise, the system may reject pod creation.
* LimitRange validations occurs only at Pod Admission stage, not on Running pods.

Examples of policies that could be created using limit range are:

* In a 2 node cluster with a capacity of 8 GiB RAM, and 16 cores, constrain Pods in a namespace to request 100m and not exceeds 500m for CPU , request 200Mi and not exceed 600Mi
* Define default CPU limits and request to 150m and Memory default request to 300Mi for containers started with no cpu and memory requests in their spec.

In the case where the total limits of the namespace is less than the sum of the limits of the Pods/Containers, there may be contention for resources; The Containers or Pods will not be created.

Neither contention nor changes to limitrange will affect already created resources.

### Limiting Container compute resources

The following section discusses the creation of a LimitRange acting at Container Level. A Pod with 04 containers is first created; each container within the Pod has a specific spec.resource configuration each container within the pod is handled differently by the LimitRanger admission controller.

Create a namespace limitrange-demo using the following kubectl command:

kubectl create namespace limitrange-demo

To avoid passing the target limitrange-demo in your kubectl commands, change your context with the following command:

kubectl config set-context --current --namespace=limitrange-demo

Here is the configuration file for a LimitRange object:

| [admin/resource/limit-mem-cpu-container.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-mem-cpu-container.yaml) |
| --- |
| apiVersion: v1  kind: LimitRange  metadata:  name: limit-mem-cpu-per-container  spec:  limits:  - max:  cpu: "800m"  memory: "1Gi"  min:  cpu: "100m"  memory: "99Mi"  default:  cpu: "700m"  memory: "900Mi"  defaultRequest:  cpu: "110m"  memory: "111Mi"  type: Container |

This object defines minimum and maximum Memory/CPU limits, default cpu/Memory requests and default limits for CPU/Memory resources to be apply to containers.

Create the limit-mem-cpu-per-container LimitRange in the limitrange-demo namespace with the following kubectl command:

kubectl create -f https://k8s.io/examples/admin/resource/limit-mem-cpu-container.yaml -n limitrange-demo

kubectl describe limitrange/limit-mem-cpu-per-container -n limitrange-demo

Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio

---- -------- --- --- --------------- ------------- -----------------------

Container cpu 100m 800m 110m 700m -

Container memory 99Mi 1Gi 111Mi 900Mi -

Here is the configuration file for a Pod with 04 containers to demonstrate LimitRange features :

| [admin/resource/limit-range-pod-1.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-range-pod-1.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: busybox1  spec:  containers:  - name: busybox-cnt01  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt01; sleep 10;done"]  resources:  requests:  memory: "100Mi"  cpu: "100m"  limits:  memory: "200Mi"  cpu: "500m"  - name: busybox-cnt02  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt02; sleep 10;done"]  resources:  requests:  memory: "100Mi"  cpu: "100m"  - name: busybox-cnt03  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt03; sleep 10;done"]  resources:  limits:  memory: "200Mi"  cpu: "500m"  - name: busybox-cnt04  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt04; sleep 10;done"] |

Create the busybox1 Pod:

kubectl apply -f https://k8s.io/examples/admin/resource/limit-range-pod-1.yaml -n limitrange-demo

#### Container spec with valid CPU/Memory requests and limits

View the busybox-cnt01 resource configuration:

kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[0].resources"

{

**"limits"**: {

**"cpu"**: "500m",

**"memory"**: "200Mi"

},

**"requests"**: {

**"cpu"**: "100m",

**"memory"**: "100Mi"

}

}

* The busybox-cnt01 Container inside busybox Pod defined requests.cpu=100m and requests.memory=100Mi.
* 100m <= 500m <= 800m , The container cpu limit (500m) falls inside the authorized CPU limit range.
* 99Mi <= 200Mi <= 1Gi , The container memory limit (200Mi) falls inside the authorized Memory limit range.
* No request/limits ratio validation for CPU/Memory , thus the container is valid and created.

#### Container spec with a valid CPU/Memory requests but no limits

View the busybox-cnt02 resource configuration

kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[1].resources"

{

**"limits"**: {

**"cpu"**: "700m",

**"memory"**: "900Mi"

},

**"requests"**: {

**"cpu"**: "100m",

**"memory"**: "100Mi"

}

}

* The busybox-cnt02 Container inside busybox1 Pod defined requests.cpu=100m and requests.memory=100Mi but not limits for cpu and memory.
* The container do not have a limits section, the default limits defined in the limit-mem-cpu-per-container LimitRange object are injected to this container limits.cpu=700mi and limits.memory=900Mi.
* 100m <= 700m <= 800m , The container cpu limit (700m) falls inside the authorized CPU limit range.
* 99Mi <= 900Mi <= 1Gi , The container memory limit (900Mi) falls inside the authorized Memory limit range.
* No request/limits ratio set , thus the container is valid and created.

#### Container spec with a valid CPU/Memory limits but no requests

View the busybox-cnt03 resource configuration

kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[2].resources"

{

**"limits"**: {

**"cpu"**: "500m",

**"memory"**: "200Mi"

},

**"requests"**: {

**"cpu"**: "500m",

**"memory"**: "200Mi"

}

}

* The busybox-cnt03 Container inside busybox1 Pod defined limits.cpu=500m and limits.memory=200Mi but no requests for cpu and memory.
* The container do not define a request section, the defaultRequest defined in the limit-mem-cpu-per-container LimitRange is not used to fill its limits section but the limits defined by the container are set as requests limits.cpu=500m and limits.memory=200Mi.
* 100m <= 500m <= 800m , The container cpu limit (500m) falls inside the authorized CPU limit range.
* 99Mi <= 200Mi <= 1Gi , The container memory limit (200Mi) falls inside the authorized Memory limit range.
* No request/limits ratio set , thus the container is valid and created.

#### Container spec with no CPU/Memory requests/limits

View the busybox-cnt04 resource configuration:

kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[3].resources"

{

**"limits"**: {

**"cpu"**: "700m",

**"memory"**: "900Mi"

},

**"requests"**: {

**"cpu"**: "110m",

**"memory"**: "111Mi"

}

}

* The busybox-cnt04 Container inside busybox1 define neither limits nor requests.
* The container do not define a limit section, the default limit defined in the limit-mem-cpu-per-container LimitRange is used to fill its request limits.cpu=700m and limits.memory=900Mi .
* The container do not define a request section, the defaultRequest defined in the limit-mem-cpu-per-container LimitRange is used to fill its request section requests.cpu=110m and requests.memory=111Mi
* 100m <= 700m <= 800m , The container cpu limit (700m) falls inside the authorized CPU limit range.
* 99Mi <= 900Mi <= 1Gi , The container memory limit (900Mi) falls inside the authorized Memory limitrange .
* No request/limits ratio set , thus the container is valid and created.

All containers defined in the busybox Pod passed LimitRange validations, this the Pod is valid and create in the namespace.

### Limiting Pod compute resources

The following section discusses how to constrain resources at Pod level.

| [admin/resource/limit-mem-cpu-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-mem-cpu-pod.yaml) |
| --- |
| apiVersion: v1  kind: LimitRange  metadata:  name: limit-mem-cpu-per-pod  spec:  limits:  - max:  cpu: "2"  memory: "2Gi"  type: Pod |

Without having to delete busybox1 Pod, create the limit-mem-cpu-pod LimitRange in the limitrange-demo namespace:

kubectl apply -f https://k8s.io/examples/admin/resource/limit-mem-cpu-pod.yaml -n limitrange-demo

The limitrange is created and limits CPU to 2 Core and Memory to 2Gi per Pod:

limitrange/limit-mem-cpu-per-pod created

Describe the limit-mem-cpu-per-pod limit object using the following kubectl command:

kubectl describe limitrange/limit-mem-cpu-per-pod

Name: limit-mem-cpu-per-pod

Namespace: limitrange-demo

Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio

---- -------- --- --- --------------- ------------- -----------------------

Pod cpu - 2 - - -

Pod memory - 2Gi - - -

Now create the busybox2 Pod:

| [admin/resource/limit-range-pod-2.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-range-pod-2.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: busybox2  spec:  containers:  - name: busybox-cnt01  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt01; sleep 10;done"]  resources:  requests:  memory: "100Mi"  cpu: "100m"  limits:  memory: "200Mi"  cpu: "500m"  - name: busybox-cnt02  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt02; sleep 10;done"]  resources:  requests:  memory: "100Mi"  cpu: "100m"  - name: busybox-cnt03  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt03; sleep 10;done"]  resources:  limits:  memory: "200Mi"  cpu: "500m"  - name: busybox-cnt04  image: busybox  command: ["/bin/sh"]  args: ["-c", "while true; do echo hello from cnt04; sleep 10;done"] |

kubectl apply -f https://k8s.io/examples/admin/resource/limit-range-pod-2.yaml -n limitrange-demo

The busybox2 Pod definition is identical to busybox1 but an error is reported since Pod’s resources are now limited:

Error from server (Forbidden): error when creating "limit-range-pod-2.yaml": pods "busybox2" is forbidden: [maximum cpu usage per Pod is 2, but limit is 2400m., maximum memory usage per Pod is 2Gi, but limit is 2306867200.]

kubectl get po/busybox1 -n limitrange-demo -o json | jq ".spec.containers[].resources.limits.memory"

"200Mi"

"900Mi"

"200Mi"

"900Mi"

busybox2 Pod will not be admitted on the cluster since the total memory limit of its container is greater than the limit defined in the LimitRange. busybox1 will not be evicted since it was created and admitted on the cluster before the LimitRange creation.

### Limiting Storage resources

You can enforce minimum and maximum size of [storage resources](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) that can be requested by each PersistentVolumeClaim in a namespace using a LimitRange:

| [admin/resource/storagelimits.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/storagelimits.yaml) |
| --- |
| apiVersion: v1  kind: LimitRange  metadata:  name: storagelimits  spec:  limits:  - type: PersistentVolumeClaim  max:  storage: 2Gi  min:  storage: 1Gi |

Apply the YAML using kubectl create:

kubectl create -f https://k8s.io/examples/admin/resource/storagelimits.yaml -n limitrange-demo

limitrange/storagelimits created

Describe the created object:

kubectl describe limits/storagelimits

The output should look like:

Name: storagelimits

Namespace: limitrange-demo

Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio

---- -------- --- --- --------------- ------------- -----------------------

PersistentVolumeClaim storage 1Gi 2Gi - - -

| [admin/resource/pvc-limit-lower.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/pvc-limit-lower.yaml) |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: pvc-limit-lower  spec:  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 500Mi |

kubectl create -f https://k8s.io/examples/admin/resource//pvc-limit-lower.yaml -n limitrange-demo

While creating a PVC with requests.storage lower than the Min value in the LimitRange, an Error thrown by the server:

Error from server (Forbidden): error when creating "pvc-limit-lower.yaml": persistentvolumeclaims "pvc-limit-lower" is forbidden: minimum storage usage per PersistentVolumeClaim is 1Gi, but request is 500Mi.

Same behaviour is noted if the requests.storage is greater than the Max value in the LimitRange:

| [admin/resource/pvc-limit-greater.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/pvc-limit-greater.yaml) |
| --- |
| apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: pvc-limit-greater  spec:  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 5Gi |

kubectl create -f https://k8s.io/examples/admin/resource/pvc-limit-greater.yaml -n limitrange-demo

Error from server (Forbidden): error when creating "pvc-limit-greater.yaml": persistentvolumeclaims "pvc-limit-greater" is forbidden: maximum storage usage per PersistentVolumeClaim is 2Gi, but request is 5Gi.

### Limits/Requests Ratio

If LimitRangeItem.maxLimitRequestRatio is specified in the LimitRangeSpec, the named resource must have a request and limit that are both non-zero where limit divided by request is less than or equal to the enumerated value

The following LimitRange enforces memory limit to be at most twice the amount of the memory request for any pod in the namespace.

| [admin/resource/limit-memory-ratio-pod.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-memory-ratio-pod.yaml) |
| --- |
| apiVersion: v1  kind: LimitRange  metadata:  name: limit-memory-ratio-pod  spec:  limits:  - maxLimitRequestRatio:  memory: 2  type: Pod |

kubectl apply -f https://k8s.io/examples/admin/resource/limit-memory-ratio-pod.yaml

Describe the LimitRange with the following kubectl command:

$ kubectl describe limitrange/limit-memory-ratio-pod

Name: limit-memory-ratio-pod

Namespace: limitrange-demo

Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio

---- -------- --- --- --------------- ------------- -----------------------

Pod memory - - - - 2

Let’s create a pod with requests.memory=100Mi and limits.memory=300Mi:

| [admin/resource/limit-range-pod-3.yaml](https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/admin/resource/limit-range-pod-3.yaml) |
| --- |
| apiVersion: v1  kind: Pod  metadata:  name: busybox3  spec:  containers:  - name: busybox-cnt01  image: busybox  resources:  limits:  memory: "300Mi"  requests:  memory: "100Mi" |

kubectl apply -f https://k8s.io/examples/admin/resource/limit-range-pod-3.yaml

The pod creation failed as the ratio here (3) is greater than the enforced limit (2) in limit-memory-ratio-pod LimitRange

Error from server (Forbidden): error when creating "limit-range-pod-3.yaml": pods "busybox3" is forbidden: memory max limit to request ratio per Pod is 2, but provided ratio is 3.000000.

#### Clean up

Delete the limitrange-demo namespace to free all resources:

kubectl delete ns limitrange-demo

### Examples

* See [a tutorial on how to limit compute resources per namespace](https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-constraint-namespace/) .
* Check [how to limit storage consumption](https://kubernetes.io/docs/tasks/administer-cluster/limit-storage-consumption/#limitrange-to-limit-requests-for-storage).
* See a [detailed example on quota per namespace](https://kubernetes.io/docs/tasks/administer-cluster/quota-memory-cpu-namespace/).

### What's next

See [LimitRanger design doc](https://git.k8s.io/community/contributors/design-proposals/resource-management/admission_control_limit_range.md) for more information.

### Feedback

## 资源配额

* + [**启用资源配额**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#启用资源配额)
  + [**计算资源配额**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#计算资源配额)
  + [**存储资源配额**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#存储资源配额)
  + [**对象数量配额**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#对象数量配额)
  + [**配额作用域**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#配额作用域)
  + [**请求／约束**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#请求-约束)
  + [**查看和设置配额**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#查看和设置配额)
  + [**配额和集群容量**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#配额和集群容量)
  + [**示例**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#示例)
  + [**更多信息**](https://kubernetes.io/zh/docs/concepts/policy/resource-quotas/#更多信息)

当多个用户或团队共享具有固定数目节点的集群时，人们会担心有人使用的资源超出应有的份额。

资源配额是帮助管理员解决这一问题的工具。

资源配额， 通过 ResourceQuota 对象来定义， 对每个namespace的资源消耗总量提供限制。 它可以按类型限制namespace下可以创建的对象的数量，也可以限制可被该项目以资源形式消耗的计算资源的总量。

资源配额的工作方式如下：

* 不同的团队在不同的namespace下工作。 目前这是自愿的， 但计划通过ACL (Access Control List 访问控制列表) 使其变为强制性的。
* 管理员为每个namespace创建一个或多个资源配额对象。
* 用户在namespace下创建资源 (pods、 services等)，同时配额系统会跟踪使用情况，来确保其不超过 资源配额中定义的硬性资源限额。
* 如果资源的创建或更新违反了配额约束，则请求会失败，并返回 HTTP状态码 403 FORBIDDEN ，以及说明违反配额 约束的信息。
* 如果namespace下的计算资源 （如 cpu 和 memory）的配额被启用，则用户必须为这些资源设定请求值（request） 和约束值（limit），否则配额系统将拒绝Pod的创建。 提示: 可使用 LimitRange 准入控制器来为没有设置计算资源需求的Pod设置默认值。 作为示例，请参考 [演练](https://kubernetes.io/docs/tasks/administer-cluster/quota-memory-cpu-namespace/) 来避免这个问题。

下面是使用namespace和配额构建策略的示例：

* 在具有 32 GiB 内存 和 16 核CPU资源的集群中， 允许A团队使用 20 GiB 内存 和 10 核的CPU资源， 允许B团队使用 10GiB 内存和 4 核的CPU资源， 并且预留 2GiB 内存和 2 核的CPU资源供将来分配。
* 限制 “testing” namespace使用 1 核CPU资源和 1GiB 内存。 允许 “production” namespace使用任意数量。

在集群容量小于各namespace配额总和的情况下，可能存在资源竞争。 Kubernetes采用先到先服务的方式处理这类问题。

无论是资源竞争还是配额的变更都不会影响已经创建的资源。

### 启用资源配额

资源配额的支持在很多Kubernetes版本中是默认开启的。 当 apiserver 的 --admission-control= 参数中包含 ResourceQuota 时，资源配额会被启用。

当namespace中存在一个 ResourceQuota 对象时，该namespace即开始实施资源配额管理。 一个namespace中最多只应存在一个 ResourceQuota 对象

### 计算资源配额

用户可以对给定namespace下的 [计算资源](https://kubernetes.io/docs/user-guide/compute-resources) 总量进行限制。

配额机制所支持的资源类型：

| 资源名称 | 描述 |
| --- | --- |
| cpu | 所有非终止状态的Pod中，其CPU需求总量不能超过该值。 |
| limits.cpu | 所有非终止状态的Pod中，其CPU限额总量不能超过该值。 |
| limits.memory | 所有非终止状态的Pod中，其内存限额总量不能超过该值。 |
| memory | 所有非终止状态的Pod中，其内存需求总量不能超过该值。 |
| requests.cpu | 所有非终止状态的Pod中，其CPU需求总量不能超过该值。 |
| requests.memory | 所有非终止状态的Pod中，其内存需求总量不能超过该值。 |

### 存储资源配额

用户可以对给定namespace下的 [存储资源](https://kubernetes.io/docs/user-guide/persistent-volumes) 总量进行限制。

此外，还可以根据相关的存储类（Storage Class）来限制存储资源的消耗。

| 资源名称 | 描述 |
| --- | --- |
| requests.storage | 所有的PVC中，存储资源的需求不能超过该值。 |
| persistentvolumeclaims | namespace中所允许的 [PVC](https://kubernetes.io/docs/user-guide/persistent-volumes/#persistentvolumeclaims) 总量。 |
| <storage-class-name>.storageclass.storage.k8s.io/requests.storage | 所有该storage-class-name相关的PVC中， 存储资源的需求不能超过该值。 |
| <storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims | namespace中所允许的该storage-class-name相关的[PVC](https://kubernetes.io/docs/user-guide/persistent-volumes/#persistentvolumeclaims)的总量。 |

例如，如果一个操作人员针对 “黄金” 存储类型与 “铜” 存储类型设置配额，操作员可以 定义配额如下：

* gold.storageclass.storage.k8s.io/requests.storage: 500Gi
* bronze.storageclass.storage.k8s.io/requests.storage: 100Gi

### 对象数量配额

给定类型的对象数量可以被限制。 支持以下类型：

| 资源名称 | 描述 |
| --- | --- |
| configmaps | namespace下允许存在的configmap的数量。 |
| persistentvolumeclaims | namespace下允许存在的[PVC](https://kubernetes.io/docs/user-guide/persistent-volumes/#persistentvolumeclaims)的数量。 |
| pods | namespace下允许存在的非终止状态的pod数量。 如果pod 的 status.phase 为 Failed 或 Succeeded ， 那么其处于终止状态。 |
| replicationcontrollers | namespace下允许存在的replication controllers的数量。 |
| resourcequotas | namespace下允许存在的 [resource quotas](https://kubernetes.io/docs/admin/admission-controllers/#resourcequota) 的数量。 |
| services | namespace下允许存在的service的数量。 |
| services.loadbalancers | namespace下允许存在的load balancer类型的service的数量。 |
| services.nodeports | namespace下允许存在的node port类型的service的数量。 |
| secrets | namespace下允许存在的secret的数量。 |

例如 pods 配额统计并保证单个namespace下创建 pods 的最大数量。

用户可能希望在namespace中为pod设置配额，来避免有用户创建很多小的pod，从而耗尽集群提供的pod IP地址。

### 配额作用域

每个配额都有一组相关的作用域（scope），配额只会对作用域内的资源生效。

当一个作用域被添加到配额中后，它会对作用域相关的资源数量作限制。 如配额中指定了允许（作用域）集合之外的资源，会导致验证错误。

| 范围 | 描述 |
| --- | --- |
| Terminating | 匹配 spec.activeDeadlineSeconds >= 0 的pod。 |
| NotTerminating | 匹配 spec.activeDeadlineSeconds is nil 的pod。 |
| BestEffort | 匹配”尽力而为（best effort)“服务类型的pod。 |
| NotBestEffort | 匹配非”尽力而为（best effort)“服务类型的pod。 |

BestEffort 作用域限制配额跟踪以下资源： pods

Terminating、 NotTerminating 和 NotBestEffort 限制配额跟踪以下资源：

* cpu
* limits.cpu
* limits.memory
* memory
* pods
* requests.cpu
* requests.memory

### 请求／约束

分配计算资源时，每个容器可以为CPU或内存指定请求和约束。 也可以设置两者中的任何一个。

如果配额中指定了 requests.cpu 或 requests.memory 的值，那么它要求每个进来的容器针对这些资源有明确的请求。 如果配额中指定了 limits.cpu 或 limits.memory的值，那么它要求每个进来的容器针对这些资源指定明确的约束。

### 查看和设置配额

Kubectl 支持创建、更新和查看配额：

$ kubectl create namespace myspace

$ cat <<EOF > compute-resources.yaml

apiVersion: v1

kind: ResourceQuota

metadata:

name: compute-resources

spec:

hard:

pods: "4"

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

EOF

$ kubectl create -f ./compute-resources.yaml --namespace=myspace

$ cat <<EOF > object-counts.yaml

apiVersion: v1

kind: ResourceQuota

metadata:

name: object-counts

spec:

hard:

configmaps: "10"

persistentvolumeclaims: "4"

replicationcontrollers: "20"

secrets: "10"

services: "10"

services.loadbalancers: "2"

EOF

$ kubectl create -f ./object-counts.yaml --namespace=myspace

$ kubectl get quota --namespace=myspace

NAME AGE

compute-resources 30s

object-counts 32s

$ kubectl describe quota compute-resources --namespace=myspace

Name: compute-resources

Namespace: myspace

Resource Used Hard

-------- ---- ----

limits.cpu 0 2

limits.memory 0 2Gi

pods 0 4

requests.cpu 0 1

requests.memory 0 1Gi

$ kubectl describe quota object-counts --namespace=myspace

Name: object-counts

Namespace: myspace

Resource Used Hard

-------- ---- ----

configmaps 0 10

persistentvolumeclaims 0 4

replicationcontrollers 0 20

secrets 1 10

services 0 10

services.loadbalancers 0 2

### 配额和集群容量

配额对象是独立于集群容量的。它们通过绝对的单位来表示。 所以，为集群添加节点， 不会 自动赋予每个namespace消耗更多资源的能力。

有时可能需要更复杂的策略，比如：

* 在几个团队中按比例划分总的集群资源。
* 允许每个租户根据需要增加资源使用量，但要有足够的限制以防止意外资源耗尽。
* 在namespace中添加节点、提高配额的额外需求。

这些策略可以基于 ResourceQuota，通过编写一个检测配额使用，并根据其他信号调整各namespace下的配额硬性限制的 “控制器” 来实现。

注意：资源配额对集群资源总体进行划分，但它对节点没有限制：来自多个namespace的Pod可能在同一节点上运行。

### 示例

查看 [如何使用资源配额的详细示例](https://kubernetes.io/docs/tasks/administer-cluster/quota-api-object/)。

### 更多信息

查看 [资源配额设计文档](https://git.k8s.io/community/contributors/design-proposals/resource-management/admission_control_resource_quota.md) 了解更多信息。

### 反馈

# 概念模板示例