

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

获得 Kubernetes 及其构件的高层次概要。

Kubernetes 架构

Kubernetes 背后的架构概念。

容器

打包应用及其运行依赖环境的技术。

工作负载

理解 Pods，Kubernetes 中可部署的最小计算对象，以及辅助它运行它们的高层抽象对象。

服务、负载均衡和联网

Kubernetes 网络背后的概念和资源。

存储

为集群中的 Pods 提供长期和临时存储的方法。

配置

Kubernetes 为配置 Pods 提供的资源。

安全

确保云原生工作负载安全的一组概念。

策略

可配置的、可应用到一组资源的策略。

[调度和驱逐 \(Scheduling and Eviction\)](#)

在Kubernetes中，调度 (Scheduling) 指的是确保 Pods 匹配到合适的节点，以便 kubelet 能够运行它们。驱逐 (Eviction) 是在资源匮乏的节点上，主动让一个或多个 Pods 失效的过程。

[集群管理](#)

关于创建和管理 Kubernetes 集群的底层细节。

[扩展 Kubernetes](#)

改变你的 Kubernetes 集群的行为的若干方法。

概述

获得 Kubernetes 及其构件的高层次概要。

[Kubernetes 是什么？](#)

Kubernetes 是一个可移植的，可扩展的开源平台，用于管理容器化的工作负载和服务，方便了声明式配置和自动化。它拥有一个庞大且快速增长的生态系统。Kubernetes 的服务，支持和工具广泛可用。

[Kubernetes 组件](#)

Kubernetes 集群由代表控制平面的组件和一组称为节点的机器组成。

[Kubernetes API](#)

Kubernetes API 使你可以查询和操纵 Kubernetes 中对象的状态。Kubernetes 控制平面的核心是 API 服务器和它暴露的 HTTP API。用户、集群的不同部分以及外部组件都通过 API 服务器相互通信。

[使用 Kubernetes 对象](#)

Kubernetes 对象是 Kubernetes 系统中的持久性实体。Kubernetes 使用这些实体表示您的集群状态。了解 Kubernetes 对象模型以及如何使用这些对象。

Kubernetes 是什么？

Kubernetes 是一个可移植的，可扩展的开源平台，用于管理容器化的工作负载和服务，方便了声明式配置和自动化。它拥有一个庞大且快速增长的生态系统。Kubernetes 的服务，支持和工具广泛可用。

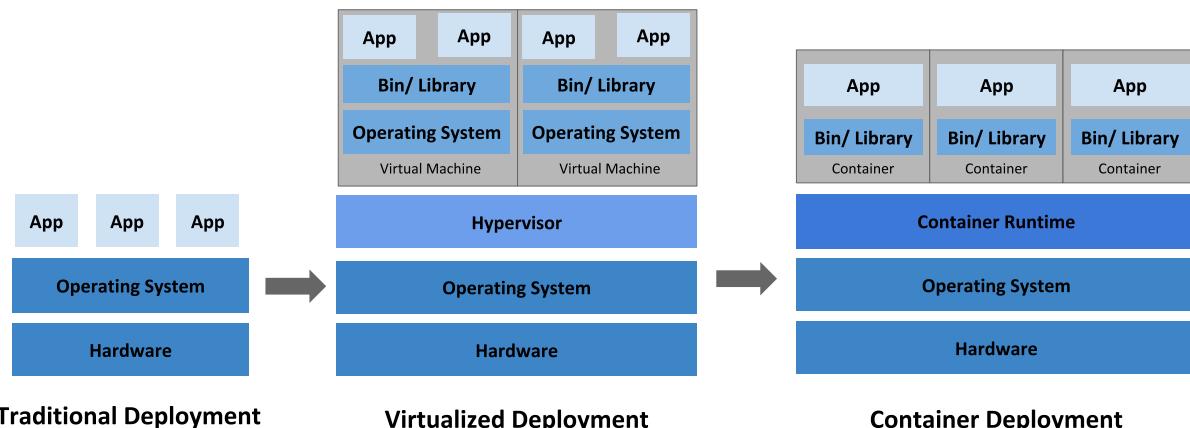
此页面是 Kubernetes 的概述。

Kubernetes 是一个可移植的、可扩展的开源平台，用于管理容器化的工作负载和服务，可促进声明式配置和自动化。Kubernetes 拥有一个庞大且快速增长的生态系统。Kubernetes 的服务、支持和工具广泛可用。

名称 **Kubernetes** 源于希腊语，意为“舵手”或“飞行员”。Google 在 2014 年开源了 Kubernetes 项目。Kubernetes 建立在 [Google 在大规模运行生产工作负载方面拥有十几年的经验](#) 的基础上，结合了社区中最好的想法和实践。

时光回溯

让我们回顾一下为什么 Kubernetes 如此有用。



传统部署时代：

早期，组织在物理服务器上运行应用程序。无法为物理服务器中的应用程序定义资源边界，这会导致资源分配问题。例如，如果在物理服务器上运行多个应用程序，则可能会出现一个应用程序占用大部分资源的情况，结果可能导致其他应用程序的性能下降。一种解决方案是在不同的物理服务器上运行每个应用程序，但是由于资源利用不足而无法扩展，并且组织维护许多物理服务器的成本很高。

虚拟化部署时代：

作为解决方案，引入了虚拟化。虚拟化技术允许你在单个物理服务器的 CPU 上运行多个虚拟机（VM）。虚拟化允许应用程序在 VM 之间隔离，并提供一定程度的安全，因为一个应用程序的信息不能被另一应用程序随意访问。

虚拟化技术能够更好地利用物理服务器上的资源，并且因为可轻松地添加或更新应用程序而可以实现更好的可伸缩性，降低硬件成本等等。

每个 VM 是一台完整的计算机，在虚拟化硬件之上运行所有组件，包括其自己的操作系统。

容器部署时代：

容器类似于 VM，但是它们具有被放宽的隔离属性，可以在应用程序之间共享操作系统（OS）。因此，容器被认为是轻量级的。容器与 VM 类似，具有自己的文件系统、CPU、内存、进程空间等。由于它们与基础架构分离，因此可以跨云和 OS 发行版本进行移植。

容器因具有许多优势而变得流行起来。下面列出的是容器的一些好处：

- 敏捷应用程序的创建和部署：与使用 VM 镜像相比，提高了容器镜像创建的简便性和效率。
- 持续开发、集成和部署：通过快速简单的回滚（由于镜像不可变性），支持可靠且频繁的容器镜像构建和部署。
- 关注开发与运维的分离：在构建/发布时而不是在部署时创建应用程序容器镜像，从而将应用程序与基础架构分离。
- 可观察性不仅可以显示操作系统级别的信息和指标，还可以显示应用程序的运行状况和其他指标信号。
- 跨开发、测试和生产的环境一致性：在便携式计算机上与在云中相同地运行。
- 跨云和操作系统发行版本的可移植性：可在 Ubuntu、RHEL、CoreOS、本地、Google Kubernetes Engine 和其他任何地方运行。
- 以应用程序为中心的管理：提高抽象级别，从在虚拟硬件上运行 OS 到使用逻辑资源在 OS 上运行应用程序。
- 松散耦合、分布式、弹性、解放的微服务：应用程序被分解成较小的独立部分，并且可以动态部署和管理 - 而不是在一台大型单机上整体运行。
- 资源隔离：可预测的应用程序性能。
- 资源利用：高效率和高密度。

为什么需要 Kubernetes，它能做什么？

容器是打包和运行应用程序的好方式。在生产环境中，你需要管理运行应用程序的容器，并确保不会停机。例如，如果一个容器发生故障，则需要启动另一个容器。如果系统处理此行为，会不会更容易？

这就是 Kubernetes 来解决这些问题的方法！Kubernetes 为你提供了一个可弹性运行分布式系统的框架。Kubernetes 会满足你的扩展要求、故障转移、部署模式等。例如，Kubernetes 可以轻松管理系统的 Canary 部署。

Kubernetes 为你提供：

- **服务发现和负载均衡**

Kubernetes 可以使用 DNS 名称或自己的 IP 地址公开容器，如果进入容器的流量很大，Kubernetes 可以负载均衡并分配网络流量，从而使部署稳定。

- **存储编排**

Kubernetes 允许你自动挂载你选择的存储系统，例如本地存储、公共云提供商等。

- **自动部署和回滚**

你可以使用 Kubernetes 描述已部署容器的所需状态，它可以以受控的速率将实际状态更改为期望状态。例如，你可以自动化 Kubernetes 来为你的部署创建新容器，删除现有容器并将它们的所有资源用于新容器。

- **自动完成装箱计算**

Kubernetes 允许你指定每个容器所需 CPU 和内存 (RAM)。当容器指定了资源请求时，Kubernetes 可以做出更好的决策来管理容器的资源。

- **自我修复**

Kubernetes 重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器，并且在准备好服务之前不将其通告给客户端。

- **密钥与配置管理**

Kubernetes 允许你存储和管理敏感信息，例如密码、OAuth 令牌和 ssh 密钥。你可以在不重建容器镜像的情况下部署和更新密钥和应用程序配置，也无需在堆栈配置中暴露密钥。

Kubernetes 不是什么

Kubernetes 不是传统的、包罗万象的 PaaS (平台即服务) 系统。由于 Kubernetes 在容器级别而不是在硬件级别运行，它提供了 PaaS 产品共有的一些普遍适用的功能，例如部署、扩展、负载均衡、日志记录和监视。但是，Kubernetes 不是单体系统，默认解决方案都是可选和可插拔的。Kubernetes 提供了构建开发人员平台的基础，但是在重要的地方保留了用户的选择和灵活性。

Kubernetes :

- 不限制支持的应用程序类型。Kubernetes 旨在支持极其多种多样的工作负载，包括无状态、有状态和数据处理工作负载。如果应用程序可以在容器中运行，那么它应该可以在 Kubernetes 上很好地运行。
- 不部署源代码，也不构建你的应用程序。持续集成(CI)、交付和部署 (CI/CD) 工作流取决于组织的文化和偏好以及技术要求。
- 不提供应用程序级别的服务作为内置服务，例如中间件（例如，消息中间件）、数据处理框架（例如，Spark）、数据库（例如，mysql）、缓存、集群存储系统（例如，Ceph）。这样的组件可以在 Kubernetes 上运行，并且/或者可以由运行在 Kubernetes 上的应用程序通过可移植机制（例如，[开放服务代理](#)）来访问。
- 不要求日志记录、监视或警报解决方案。它提供了一些集成作为概念证明，并提供了收集和导出指标的机制。
- 不提供或不要求配置语言/系统（例如 jsonnet），它提供了声明性 API，该声明性 API 可以由任意形式的声明性规范所构成。
- 不提供也不采用任何全面的机器配置、维护、管理或自我修复系统。
- 此外，Kubernetes 不仅仅是一个编排系统，实际上它消除了编排的需要。编排的技术定义是执行已定义的工作流程：首先执行 A，然后执行 B，再执行 C。相比之下，Kubernetes 包含一组独立的、可组合的控制过程，这些过程连续地将当前状态驱动到所提供的所需状态。如何从 A 到 C 的方式无关紧要，也不需要集中控制，这使得系统更易于使用且功能更强大、系统更健壮、更为弹性和可扩展。

接下来

- 查阅 [Kubernetes 组件](#)
- 开始 [Kubernetes 入门](#)？

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 12, 2020 at 4:39 PM PST: [\[zh\] Sync changes from English site \(2\) \(8ae2209de\)](#)

Kubernetes 组件

Kubernetes 集群由代表控制平面的组件和一组称为节点的机器组成。

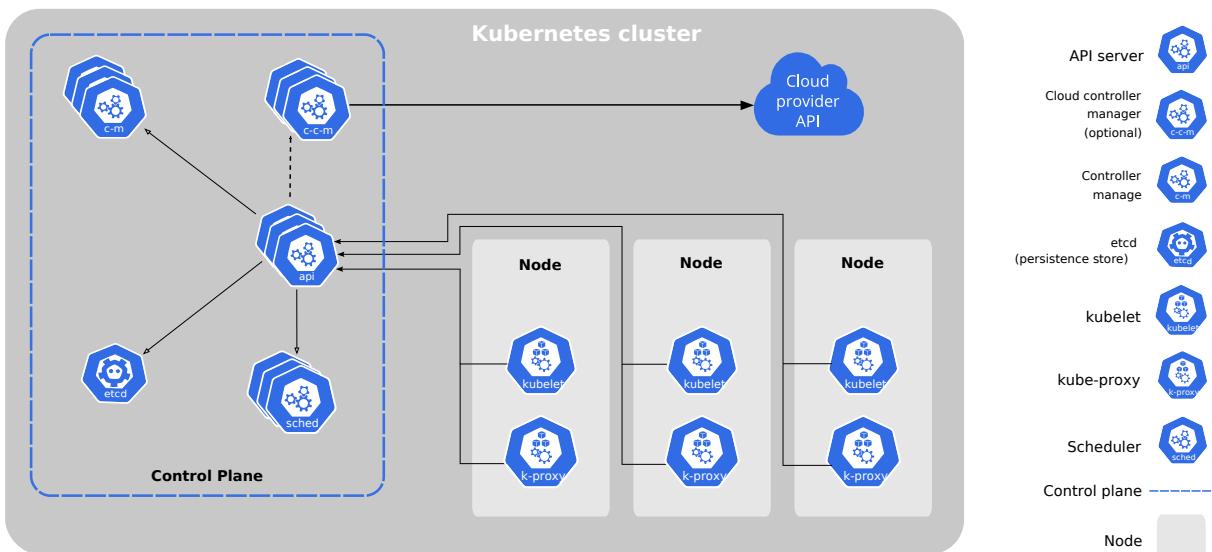
当你部署完 Kubernetes，即拥有了一个完整的集群。

一个 Kubernetes 集群包含 集群由一组被称作节点的机器组成。这些节点上运行 Kubernetes 所管理的容器化应用。集群具有至少一个工作节点。

工作节点托管作为应用负载的组件的 Pod。控制平面管理集群中的工作节点和 Pod。为集群提供故障转移和高可用性，这些控制平面一般跨多主机运行，集群跨多个节点运行。

本文档概述了交付正常运行的 Kubernetes 集群所需的各种组件。

这张图表展示了包含所有相互关联组件的 Kubernetes 集群。



控制平面组件 (Control Plane Components)

控制平面的组件对集群做出全局决策(比如调度) , 以及检测和响应集群事件 (例如 , 当不满足部署的 replicas 字段时 , 启动新的 [pod](#)) 。

控制平面组件可以在集群中的任何节点上运行。然而 , 为了简单起见 , 设置脚本通常会在同一个计算机上启动所有控制平面组件 , 并且不会在此计算机上运行用户容器。请参阅[构建高可用性集群](#) 中对于多主机 VM 的设置示例。

kube-apiserver

API 服务器是 Kubernetes [控制面](#)的组件 , 该组件公开了 Kubernetes API。 API 服务器是 Kubernetes 控制面的前端。

Kubernetes API 服务器的主要实现是 [kube-apiserver](#)。 kube-apiserver 设计上考虑了水平伸缩 , 也就是说 , 它可通过部署多个实例进行伸缩。你可以运行 kube-apiserver 的多个实例 , 并在这些实例之间平衡流量。

etcd

etcd 是兼具一致性和高可用性的键值数据库 , 可以作为保存 Kubernetes 所有集群数据的后台数据库。

您的 Kubernetes 集群的 etcd 数据库通常需要有个备份计划。

要了解 etcd 更深层次的信息 , 请参考 [etcd 文档](#)。

kube-scheduler

控制平面组件 , 负责监视新创建的、未指定运行[节点 \(node \)](#)的 [Pods](#) , 选择节点让 Pod 在上面运行。

调度决策考虑的因素包括单个 Pod 和 Pod 集合的资源需求、硬件/软件/策略约束、亲和性和反亲和性规范、数据位置、工作负载间的干扰和最后时限。

kube-controller-manager

在主节点上运行 [控制器](#) 的组件。

从逻辑上讲 , 每个[控制器](#)都是一个单独的进程 , 但是为了降低复杂性 , 它们都被编译到同一个可执行文件 , 并在一个进程中运行。

这些控制器包括:

- 节点控制器 (Node Controller) : 负责在节点出现故障时进行通知和响应。
- 副本控制器 (Replication Controller) : 负责为系统中的每个副本控制器对象维护正确数量的 Pod。
- 端点控制器 (Endpoints Controller) : 填充端点(Endpoints)对象(即加入 Service 与 Pod)。

- 服务帐户和令牌控制器 (Service Account & Token Controllers) : 为新的命名空间创建默认帐户和 API 访问令牌。

cloud-controller-manager

云控制器管理器是指嵌入特定云的控制逻辑的 [控制平面](#) 组件。 云控制器管理器允许您链接聚合到云提供商的应用编程接口中， 并分离出相互作用的组件与您的集群交互的组件。

cloud-controller-manager 仅运行特定于云平台的控制回路。 如果你在自己的环境中运行 Kubernetes， 或者在本地计算机中运行学习环境， 所部署的环境中不需要云控制器管理器。

与 kube-controller-manager 类似， cloud-controller-manager 将若干逻辑上独立的控制回路组合到同一个可执行文件中， 供你以同一进程的方式运行。 你可以对其执行水平扩容（运行不止一个副本）以提升性能或者增强容错能力。

下面的控制器都包含对云平台驱动的依赖：

- 节点控制器 (Node Controller) : 用于在节点终止响应后检查云提供商以确定节点是否已被删除
- 路由控制器 (Route Controller) : 用于在底层云基础架构中设置路由
- 服务控制器 (Service Controller) : 用于创建、更新和删除云提供商负载均衡器

Node 组件

节点组件在每个节点上运行， 维护运行的 Pod 并提供 Kubernetes 运行环境。

kubelet

一个在集群中每个[节点 \(node \)](#) 上运行的代理。 它保证[容器 \(containers \)](#) 都运行在 [Pod](#) 中。

kubelet 接收一组通过各类机制提供给它的 PodSpecs， 确保这些 PodSpecs 中描述的容器处于运行状态且健康。 kubelet 不会管理不是由 Kubernetes 创建的容器。

kube-proxy

[kube-proxy](#) 是集群中每个节点上运行的网络代理， 实现 Kubernetes [服务 \(Service \)](#) 概念的一部分。

kube-proxy 维护节点上的网络规则。 这些网络规则允许从集群内部或外部的网络会话与 Pod 进行网络通信。

如果操作系统提供了数据包过滤层并可用的话， kube-proxy 会通过它来实现网络规则。 否则， kube-proxy 仅转发流量本身。

容器运行时 (Container Runtime)

容器运行环境是负责运行容器的软件。

Kubernetes 支持多个容器运行环境: [Docker](#)、[containerd](#)、[CRI-O](#) 以及任何实现 [Kubernetes CRI \(容器运行环境接口\)](#)。

插件 (Addons)

插件使用 Kubernetes 资源 ([DaemonSet](#)、[Deployment](#)等) 实现集群功能。因为这些插件提供集群级别的功能，插件中命名空间域的资源属于 `kube-system` 命名空间。

下面描述众多插件中的几种。有关可用插件的完整列表，请参见 [插件 \(Addons \)](#)。

DNS

尽管其他插件都并非严格意义上的必需组件，但几乎所有 Kubernetes 集群都应该有[集群 DNS](#)，因为很多示例都需要 DNS 服务。

集群 DNS 是一个 DNS 服务器，和环境中的其他 DNS 服务器一起工作，它为 Kubernetes 服务提供 DNS 记录。

Kubernetes 启动的容器自动将此 DNS 服务器包含在其 DNS 搜索列表中。

Web 界面 (仪表盘)

[Dashboard](#) 是Kubernetes 集群的通用的、基于 Web 的用户界面。 它使用户可以管理集群中运行的应用程序以及集群本身并进行故障排除。

容器资源监控

[容器资源监控](#) 将关于容器的一些常见的时间序列度量值保存到一个集中的数据库中，并提供用于浏览这些数据的界面。

集群层面日志

[集群层面日志](#) 机制负责将容器的日志数据 保存到一个集中的日志存储中，该存储能够提供搜索和浏览接口。

接下来

- 进一步了解[节点](#)
- 进一步了解[控制器](#)
- 进一步了解[kube-scheduler](#)
- 阅读 etcd 官方[文档](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 18, 2020 at 11:47 AM PST: [\[zh\] fix content error \(920e55190\)](#)

Kubernetes API

Kubernetes API 使你可以查询和操纵 Kubernetes 中对象的状态。Kubernetes 控制平面的核心是 API 服务器和它暴露的 HTTP API。用户、集群的不同部分以及外部组件都通过 API 服务器相互通信。

Kubernetes [控制面](#) 的核心是 [API 服务器](#)。API 服务器负责提供 HTTP API，以供用户、集群中的不同部分和集群外部组件相互通信。

Kubernetes API 使你可以查询和操纵 Kubernetes API 中对象（例如：Pod、Namespace、ConfigMap 和 Event）的状态。

大部分操作都可以通过 [kubectl](#) 命令行接口或 [类似 kubeadm](#) 这类命令行工具来执行，这些工具在背后也是调用 API。不过，你也可以使用 REST 调用来自访问这些 API。

如果你正在编写程序来访问 Kubernetes API，可以考虑使用 [客户端库](#)之一。

OpenAPI 规范

完整的 API 细节是用 [OpenAPI](#) 来表述的。

Kubernetes API 服务器通过 /openapi/v2 末端提供 OpenAPI 规范。你可以按照下表所给的请求头部，指定响应的格式：

OpenAPI v2 查询请求的合法头部值

头部	可选值	说明
Accept-Encoding	gzip	不指定此头部也是可以的
Accept	application/com.github.proto-openapi.spec.v2@v1.0+protobuf	主要用于集群内部
	application/json	默认值
	*	提供application/json

Kubernetes 为 API 实现了一种基于 Protobuf 的序列化格式，主要用于集群内部通信。关于此格式的详细信息，可参考 [Kubernetes Protobuf 序列化](#) 设计提案。每种模式对应的接口描述语言 (IDL) 位于定义 API 对象的 Go 包中。

API 变更

任何成功的系统都要随着新的使用案例的出现和现有案例的变化来成长和变化。为此，Kubernetes 的功能特性设计考虑了让 Kubernetes API 能够持续变更和成长的因素。

Kubernetes 项目的目标是 不要 引发现有客户端的兼容性问题，并在一定的时期内 维持这种兼容性，以便其他项目有机会作出适应性变更。

一般而言，新的 API 资源和新的资源字段可以被频繁地添加进来。删除资源或者字段则要遵从 [API 废弃策略](#)。

关于什么是兼容性的变更、如何变更 API 等详细信息，可参考 [API 变更](#)。

API 组和版本

为了简化删除字段或者重构资源表示等工作，Kubernetes 支持多个 API 版本，每一个版本都在不同 API 路径下，例如 /api/v1 或 /apis/rbac.authorization.k8s.io/v1alpha1。

版本化是在 API 级别而不是在资源或字段级别进行的，目的是为了确保 API 为系统资源和行为提供清晰、一致的视图，并能够控制对已废止的和/或实验性 API 的访问。

为了便于演化和扩展其 API，Kubernetes 实现了 可被[启用或禁用的 API 组](#)。

API 资源之间靠 API 组、资源类型、名字空间（对于名字空间作用域的资源而言）和 名字来相互区分。API 服务器可能通过多个 API 版本来向外提供相同的下层数据，并透明地完成不同 API 版本之间的转换。所有这些不同的版本实际上都是同一资源的（不同）表现形式。例如，假定同一资源有 v1 和 v1beta1 版本，使用 v1beta1 创建的对象则可以使用 v1beta1 或者 v1 版本来读取、更改 或者删除。

关于 API 版本级别的详细定义，请参阅 [API 版本参考](#)。

API 扩展

有两种途径来扩展 Kubernetes API：

1. 你可以使用[自定义资源](#) 来以声明式方式定义 API 服务器如何提供你所选择的资源 API。
2. 你也可以选择实现自己的 [聚合层](#) 来扩展 Kubernetes API。

接下来

- 了解如何通过添加你自己的 [CustomResourceDefinition](#) 来扩展 Kubernetes API。
- [控制 Kubernetes API 访问](#) 页面描述了集群如何针对 API 访问管理身份认证和鉴权。
- 通过阅读 [API 参考](#) 了解 API 端点、资源类型以及示例。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 21, 2020 at 1:35 AM PST: [\[zh\] sync kubernetes-api.md \(1141c3b8a\)](#)

使用 Kubernetes 对象

Kubernetes 对象是 Kubernetes 系统中的持久性实体。Kubernetes 使用这些实体表示您的集群状态。了解 Kubernetes 对象模型以及如何使用这些对象。

[理解 Kubernetes 对象](#)

[Kubernetes 对象管理](#)

[对象名称和 IDs](#)

[名字空间](#)

[标签和选择算符](#)

[注解](#)

[字段选择器](#)

[推荐使用的标签](#)

理解 Kubernetes 对象

本页说明了 Kubernetes 对象在 Kubernetes API 中是如何表示的，以及如何在 .yaml 格式的文件中表示。

理解 Kubernetes 对象

在 Kubernetes 系统中，*Kubernetes 对象* 是持久化的实体。Kubernetes 使用这些实体去表示整个集群的状态。特别地，它们描述了如下信息：

- 哪些容器化应用在运行（以及在哪些节点上）
- 可以被应用使用的资源
- 关于应用运行时表现的策略，比如重启策略、升级策略，以及容错策略

Kubernetes 对象是 "目标性记录" —— 一旦创建对象，Kubernetes 系统将持续工作以确保对象存在。通过创建对象，本质上是在告知 Kubernetes 系统，所需要的集群工作负载看起来是什么样子的，这就是 Kubernetes 集群的 **期望状态 (Desired State)**

。

操作 Kubernetes 对象 —— 无论是创建、修改，或者删除 —— 需要使用 [Kubernetes API](#)。比如，当使用 kubectl 命令行接口时，CLI 会执行必要的 Kubernetes API 调用，也可以在程序中使用 [客户端库](#)直接调用 Kubernetes API。

对象规约 (Spec) 与状态 (Status)

几乎每个 Kubernetes 对象包含两个嵌套的对象字段，它们负责管理对象的配置：对象 *spec* (规约) 和对象 *status* (状态)。对于具有 *spec* 的对象，你必须在创建对象时设置其内容，描述你希望对象所具有的特征：期望状态 (*Desired State*)。

status 描述了对象的 *当前状态* (*Current State*)，它是由 Kubernetes 系统和组件设置并更新的。在任何时刻，Kubernetes [控制平面](#) 都一直积极地管理着对象的实际状态，以使之与期望状态相匹配。

例如，Kubernetes 中的 Deployment 对象能够表示运行在集群中的应用。当创建 Deployment 时，可能需要设置 Deployment 的 *spec*，以指定该应用需要有 3 个副本运行。Kubernetes 系统读取 Deployment 规约，并启动我们所期望的应用的 3 个实例 —— 更新状态以与规约相匹配。如果这些实例中有的失败了（一种状态变更），Kubernetes 系统通过执行修正操作 来响应规约和状态间的不一致 —— 在这里意味着它会启动一个新的实例来替换。

关于对象 *spec*、*status* 和 *metadata* 的更多信息，可参阅 [Kubernetes API 约定](#)。

描述 Kubernetes 对象

创建 Kubernetes 对象时，必须提供对象的规约，用来描述该对象的期望状态，以及关于对象的一些基本信息（例如名称）。当使用 Kubernetes API 创建对象时（或者直接创建，或者基于kubectl），API 请求必须在请求体中包含 JSON 格式的信息。**大多数情况下，需要在 .yaml 文件中为 kubectl 提供这些信息。** kubectl 在发起 API 请求时，将这些信息转换成 JSON 格式。

这里有一个 .yaml 示例文件，展示了 Kubernetes Deployment 的必需字段和对象规约：

[application/deployment.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
```

```
labels:  
  app: nginx  
spec:  
  containers:  
    - name: nginx  
      image: nginx:1.14.2  
      ports:  
        - containerPort: 80
```

使用类似于上面的 .yaml 文件来创建 Deployment 的一种方式是使用 kubectl 命令行接口 (CLI) 中的 [kubectl apply](#) 命令，将 .yaml 文件作为参数。下面是一个示例：

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

输出类似如下这样：

```
deployment.apps/nginx-deployment created
```

必需字段

在想要创建的 Kubernetes 对象对应的 .yaml 文件中，需要配置如下的字段：

- apiVersion - 创建该对象所使用的 Kubernetes API 的版本
- kind - 想要创建的对象的类别
- metadata - 帮助唯一性标识对象的一些数据，包括一个 name 字符串、UID 和可选的 namespace

你也需要提供对象的 spec 字段。对象 spec 的精确格式对每个 Kubernetes 对象来说是不同的，包含了特定于该对象的嵌套字段。[Kubernetes API 参考](#) 能够帮助我们找到任何我们想创建的对象的 spec 格式。例如，可以从 [core/v1 PodSpec](#) 查看 Pod 的 spec 格式，并且可以从 [apps/v1 DeploymentSpec](#) 查看 Deployment 的 spec 格式。

接下来

- 了解最重要的 Kubernetes 基本对象，例如 [Pod](#)。
- 了解 Kubernetes 中的[控制器](#)。
- [使用 Kubernetes API](#) 一节解释了一些 API 概念。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

Kubernetes 对象管理

kubectl 命令行工具支持多种不同的方式来创建和管理 Kubernetes 对象。本文档概述了不同的方法。阅读 [Kubectl book](#) 来了解 kubectl 管理对象的详细信息。

管理技巧

警告：

应该只使用一种技术来管理 Kubernetes 对象。混合和匹配技术作用在同一对象上将导致未定义行为。

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1 +	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate
Declarative object configuration	Directories of files	Production projects	1 +	Highest

命令式命令

使用命令式命令时，用户可以在集群中的活动对象上进行操作。用户将操作传给 kubectl 命令作为参数或标志。

这是开始或者在集群中运行一次性任务的最简单方法。因为这个技术直接在活动对象上操作，所以它不提供以前配置的历史记录。

例子

通过创建 Deployment 对象来运行 nginx 容器的实例：

```
kubectl run nginx --image nginx
```

使用不同的语法来达到同样的上面的效果：

```
kubectl create deployment nginx --image nginx
```

权衡

与对象配置相比的优点：

- 命令简单，易学且易于记忆。
- 命令仅需一步即可对集群进行更改。

与对象配置相比的缺点：

- 命令不与变更审查流程集成。
- 命令不提供与更改关联的审核跟踪。
- 除了实时内容外，命令不提供记录源。
- 命令不提供用于创建新对象的模板。

命令式对象配置

在命令式对象配置中，kubectl 命令指定操作（创建，替换等），可选标志和至少一个文件名。指定的文件必须包含 YAML 或 JSON 格式的对象的完整定义。

有关对象定义的详细信息，请查看 [API 参考](#)。

警告：

replace 命令式命令将现有规范替换为新提供的规范，并删除对配置文件中缺少的对象的所有更改。此方法不应与规范独立于配置文件进行更新的资源类型一起使用。比如类型为 LoadBalancer 的服务，它的 externalIPs 字段就是独立于集群配置进行更新。

例子

创建配置文件中定义的对象：

```
kubectl create -f nginx.yaml
```

删除两个配置文件中定义的对象：

```
kubectl delete -f nginx.yaml -f redis.yaml
```

通过覆盖活动配置来更新配置文件中定义的对象：

```
kubectl replace -f nginx.yaml
```

权衡

与命令式命令相比的优点：

- 对象配置可以存储在源控制系统中，比如 Git。
- 对象配置可以与流程集成，例如在推送和审计之前检查更新。
- 对象配置提供了用于创建新对象的模板。

与命令式命令相比的缺点：

- 对象配置需要对对象架构有基本的了解。
- 对象配置需要额外的步骤来编写 YAML 文件。

与声明式对象配置相比的优点：

- 命令式对象配置行为更加简单易懂。

- 从 Kubernetes 1.5 版本开始，命令式对象配置更加成熟。

与声明式对象配置相比的缺点：

- 命令式对象配置更适合文件，而非目录。
- 对活动对象的更新必须反映在配置文件中，否则会在下一次替换时丢失。

声明式对象配置

使用声明式对象配置时，用户对本地存储的对象配置文件进行操作，但是用户未定义要对该文件执行的操作。kubectl 会自动检测每个文件的创建、更新和删除操作。这使得配置可以在目录上工作，根据目录中配置文件对不同的对象执行不同的操作。

说明：

声明式对象配置保留其他编写者所做的修改，即使这些更改并未合并到对象配置文件中。可以通过使用 patch API 操作仅写入观察到的差异，而不是使用 replace API 操作来替换整个对象配置来实现。

例子

处理 configs 目录中的所有对象配置文件，创建并更新活动对象。可以首先使用 diff 子命令查看将要进行的更改，然后在进行应用：

```
kubectl diff -f configs/  
kubectl apply -f configs/
```

递归处理目录：

```
kubectl diff -R -f configs/  
kubectl apply -R -f configs/
```

权衡

与命令式对象配置相比的优点：

- 对活动对象所做的更改即使未合并到配置文件中，也会被保留下。
- 声明性对象配置更好地支持对目录进行操作并自动检测每个文件的操作类型（创建，修补，删除）。

与命令式对象配置相比的缺点：

- 声明式对象配置难于调试并且出现异常时结果难以理解。
- 使用 diff 产生的部分更新会创建复杂的合并和补丁操作。

接下来

- [使用命令式命令管理 Kubernetes 对象](#)
- [使用对象配置管理 Kubernetes 对象（命令式）](#)

- [使用对象配置管理 Kubernetes 对象（声明式）](#)
- [使用 Kustomize（声明式）管理 Kubernetes 对象](#)
- [Kubectl 命令参考](#)
- [Kubectl Book](#)
- [Kubernetes API 参考](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 October 12, 2020 at 10:25 AM PST: [Modify links from en version to zh verison \(6d0a20a4c\)](#)

对象名称和 IDs

集群中的每一个对象都有一个[名称](#)来标识在同类资源中的唯一性。

每个 Kubernetes 对象也有一个[UID](#)来标识在整个集群中的唯一性。

比如，在同一个[名字空间](#)中有一个名为 myapp-1234 的 Pod, 但是可以命名一个 Pod 和一个 Deployment 同为 myapp-1234.

对于用户提供的非唯一性的属性，Kubernetes 提供了[标签 \(Labels\)](#)和[注解 \(Annotation\)](#)机制。

名称

客户端提供的字符串，引用资源 url 中的对象，如/api/v1/pods/some name。

某一时刻，只能有一个给定类型的对象具有给定的名称。但是，如果删除该对象，则可以创建同名的新对象。

以下是比较常见的三种资源命名约束。

DNS 子域名

很多资源类型需要可以用作 DNS 子域名的名称。DNS 子域名的定义可参见 [RFC 1123](#)。这一要求意味着名称必须满足如下规则：

- 不能超过253个字符
- 只能包含字母数字，以及'-' 和 '.'
- 须以字母数字开头
- 须以字母数字结尾

DNS 标签名

某些资源类型需要其名称遵循 [RFC 1123](#) 所定义的 DNS 标签标准。也就是命名必须满足如下规则：

- 最多63个字符
- 只能包含字母数字，以及'-'
- 须以字母数字开头
- 须以字母数字结尾

路径分段名称

某些资源类型要求名称能被安全地用作路径中的片段。 换句话说，其名称不能是 .、..，也不可以包含 / 或 % 这些字符。

下面是一个名为nginx-demo的 Pod 的配置清单：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

说明：某些资源类型可能具有额外的命名约束。

UIDs

Kubernetes 系统生成的字符串，唯一标识对象。

在 Kubernetes 集群的整个生命周期中创建的每个对象都有一个不同的 uid，它旨在区分类似实体的历史事件。

Kubernetes UIDs 是全局唯一标识符（也叫 UUIDs）。UUIDs 是标准化的，见 ISO/IEC 9834-8 和 ITU-T X.667。

接下来

- 进一步了解 Kubernetes [标签](#)
- 参阅 [Kubernetes 标识符和名称](#)的设计文档

反馈

此页是否对您有帮助？

是否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 September 25, 2020 at 5:46 PM PST: [docs: lost "有" word \(571b8e1bc\)](#)

名字空间

Kubernetes 支持多个虚拟集群，它们底层依赖于同一个物理集群。这些虚拟集群被称为名字空间。

何时使用多个名字空间

名字空间适用于存在很多跨多个团队或项目的用户的场景。对于只有几到几十个用户的集群，根本不需要创建或考虑名字空间。当需要名称空间提供的功能时，请开始使用它们。

名字空间为名称提供了一个范围。资源的名称需要在名字空间内是唯一的，但不能跨名字空间。名字空间不能相互嵌套，每个 Kubernetes 资源只能在一个名字空间中。

名字空间是在多个用户之间划分集群资源的一种方法（通过[资源配置](#)）。

不需要使用多个名字空间来分隔轻微不同的资源，例如同一软件的不同版本：使用[标签](#)来区分同一名字空间中的不同资源。

使用名字空间

名字空间的创建和删除在[名字空间的管理指南文档](#)描述。

说明：避免使用前缀 kube- 创建名字空间，因为它是为 Kubernetes 系统名字空间保留的。

查看名字空间

你可以使用以下命令列出集群中现存的名字空间：

```
kubectl get namespace
```

NAME	STATUS	AGE
default	Active	1d
kube-node-lease	Active	1d
kube-system	Active	1d
kube-public	Active	1d

Kubernetes 会创建四个初始名字空间：

- default 没有指明使用其它名字空间的对象所使用的默认名字空间

- kube-system Kubernetes 系统创建对象所使用的名字空间
- kube-public 这个名字空间是自动创建的，所有用户（包括未经过身份验证的用户）都可以读取它。这个名字空间主要用于集群使用，以防某些资源在整个集群中应该是可见和可读的。这个名字空间的公共方面只是一种约定，而不是要求。
- kube-node-lease 此名字空间用于与各个节点相关的租期（Lease）对象；此对象的设计使得集群规模很大时节点心跳检测性能得到提升。

为请求设置名字空间

要为当前请求设置名字空间，请使用 --namespace 参数。

例如：

```
kubectl run nginx --image=nginx --namespace=<名字空间名称>
kubectl get pods --namespace=<名字空间名称>
```

设置名字空间偏好

你可以永久保存名字空间，以用于对应上下文中所有后续 kubectl 命令。

```
kubectl config set-context --current --namespace=<名字空间名称>
# 验证之
kubectl config view | grep namespace:
```

名字空间和 DNS

当你创建一个[服务](#)时，Kubernetes 会创建一个相应的[DNS 条目](#)。

该条目的形式是 <服务名称>.<名字空间名称>.svc.cluster.local，这意味着如果容器只使用 <服务名称>，它将被解析到本地名字空间的服务。这对于跨多个名字空间（如开发、分级和生产）使用相同的配置非常有用。如果你希望跨名字空间访问，则需要使用完全限定域名（FQDN）。

并非所有对象都在名字空间中

大多数 kubernetes 资源（例如 Pod、Service、副本控制器等）都位于某些名字空间中。但是名字空间资源本身并不在名字空间中。而且底层资源，例如[节点](#)和持久化卷不属于任何名字空间。

查看哪些 Kubernetes 资源在名字空间中，哪些不在名字空间中：

```
# 位于名字空间中的资源
kubectl api-resources --namespaced=true

# 不在名字空间中的资源
kubectl api-resources --namespaced=false
```

接下来

- 进一步了解[建立新的名字空间](#)。
- 进一步了解[删除名字空间](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 January 02, 2021 at 11:54 AM PST: [Update namespaces.md \(c888beb2c\)](#)

标签和选择算符

标签 (*Labels*) 是附加到 Kubernetes 对象（比如 Pods）上的键值对。标签旨在用于指定对用户有意义且相关的对象的标识属性，但不直接对核心系统有语义含义。标签可以用于组织和选择对象的子集。标签可以在创建时附加到对象，随后可以随时添加和修改。每个对象都可以定义一组键/值标签。每个键对于给定对象必须是唯一的。

```
"metadata": {
  "labels": {
    "key1": "value1",
    "key2": "value2"
  }
}
```

标签能够支持高效的查询和监听操作，对于用户界面和命令行是很理想的。应使用[注解](#)记录非识别信息。

动机

标签使用户能够以松散耦合的方式将他们自己的组织结构映射到系统对象，而无需客户端存储这些映射。

服务部署和批处理流水线通常是多维实体（例如，多个分区或部署、多个发行序列、多个层，每层多个微服务）。管理通常需要交叉操作，这打破了严格的层次表示的封装，特别是由基础设施而不是用户确定的严格的层次结构。

示例标签：

- "release" : "stable", "release" : "canary"
- "environment" : "dev", "environment" : "qa", "environment" : "production"

- "tier" : "frontend", "tier" : "backend", "tier" : "cache"
- "partition" : "customerA", "partition" : "customerB"
- "track" : "daily", "track" : "weekly"

这些只是常用标签的例子；你可以任意制定自己的约定。请记住，对于给定对象标签的键必须是唯一的。

语法和字符集

标签 是键值对。有效的标签键有两个段：可选的前缀和名称，用斜杠（/）分隔。名称段是必需的，必须小于等于 63 个字符，以字母数字字符（[a-z0-9A-Z]）开头和结尾，带有破折号（-），下划线（_），点（.）和之间的字母数字。前缀是可选的。如果指定，前缀必须是 DNS 子域：由点（.）分隔的一系列 DNS 标签，总共不超过 253 个字符，后跟斜杠（/）。

如果省略前缀，则假定标签键对用户是私有的。向最终用户对象添加标签的自动系统组件（例如 kube-scheduler、kube-controller-manager、kube-apiserver、kubectl 或其他第三方自动化工具）必须指定前缀。

kubernetes.io/ 前缀是为 Kubernetes 核心组件保留的。

有效标签值必须为 63 个字符或更少，并且必须为空或以字母数字字符（[a-z0-9A-Z]）开头和结尾，中间可以包含破折号（-）、下划线（_）、点（.）和字母或数字。

标签选择算符

与[名称和 UID](#) 不同，标签不支持唯一性。通常，我们希望许多对象携带相同的标签。

通过 标签选择算符，客户端/用户可以识别一组对象。标签选择算符是 Kubernetes 中的核心分组原语。

API 目前支持两种类型的选择算符：[基于等值的](#) 和 [基于集合的](#)。标签选择算符可以由逗号分隔的多个 需求 组成。在多个需求的情况下，必须满足所有要求，因此逗号分隔符充当逻辑 与（&&）运算符。

空标签选择算符或者未指定的选择算符的语义取决于上下文，支持使用选择算符的 API 类别应该将算符的合法性和含义用文档记录下来。

说明：对于某些 API 类别（例如 ReplicaSet）而言，两个实例的标签选择算符不得在命名空间内重叠，否则它们的控制器将互相冲突，无法确定应该存在的副本个数。

注意：对于基于等值的和基于集合的条件而言，不存在逻辑或（||）操作符。你要确保你的过滤语句按合适的方式组织。

基于等值的需求

基于等值 或 基于不等值 的需求允许按标签键和值进行过滤。 匹配对象必须满足所有指定的标签约束，尽管它们也可能具有其他标签。 可接受的运算符有=、== 和 != 三种。 前两个表示 相等（并且只是同义词），而后者表示 不相等。 例如：

```
environment = production
tier != frontend
```

前者选择所有资源，其键名等于 environment，值等于 production。 后者选择所有资源，其键名等于 tier，值不同于 frontend，所有资源都没有带有 tier 键的标签。 可以使用逗号运算符来过滤 production 环境中的非 frontend 层资源：environment=production,tier!=frontend。

基于等值的标签要求的一种使用场景是 Pod 要指定节点选择标准。 例如，下面的示例 Pod 选择带有标签 "accelerator=nvidia-tesla-p100"。

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

基于集合的需求

基于集合 的标签需求允许你通过一组值来过滤键。 支持三种操作符：in、notin 和 exists（只可以用在键标识符上）。 例如：

```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

- 第一个示例选择了所有键等于 environment 并且值等于 production 或者 qa 的资源。
- 第二个示例选择了所有键等于 tier 并且值不等于 frontend 或者 backend 的资源，以及所有没有 tier 键标签的资源。
- 第三个示例选择了所有包含了有 partition 标签的资源；没有校验它的值。

- 第四个示例选择了所有没有 partition 标签的资源；没有校验它的值。类似地，逗号分隔符充当与运算符。因此，使用 partition 键（无论为何值）和 environment 不同于 qa 来过滤资源可以使用 partition, environment notin (qa) 来实现。

基于集合 的标签选择算符是相等标签选择算符的一般形式，因为 environment=production 等同于 environment in (production) ; != 和 notin 也是类似的。

基于集合 的要求可以与基于 相等 的要求混合使用。例如：partition in (customerA, customerB),environment!=qa。

API

LIST 和 WATCH 过滤

LIST and WATCH 操作可以使用查询参数指定标签选择算符过滤一组对象。两种需求都是允许的。（这里显示的是它们出现在 URL 查询字符串中）

- 基于等值 的需求: ?labelSelector=environment%3Dproduction,tier%3Dfrontend
- 基于集合 的需求: ?labelSelector=environment+in+ %28production%2Cqa%29%2Ctier+in+%28frontend%29

两种标签选择算符都可以通过 REST 客户端用于 list 或者 watch 资源。例如，使用 kubectl 定位 apiserver，可以使用 基于等值 的标签选择算符可以这么写：

```
kubectl get pods -l environment=production,tier=frontend
```

或者使用 基于集合 的需求：

```
kubectl get pods -l 'environment in (production),tier in (frontend)'
```

正如刚才提到的，基于集合 的需求更具有表达力。例如，它们可以实现值的 或 操作：

```
kubectl get pods -l 'environment in (production, qa)'
```

或者通过 exists 运算符限制不匹配：

```
kubectl get pods -l 'environment,environment notin (frontend)'
```

在 API 对象中设置引用

一些 Kubernetes 对象，例如 [services](#) 和 [replicationcontrollers](#)，也使用了标签选择算符去指定了其他资源的集合，例如 [pods](#)。

Service 和 ReplicationController

一个 Service 指向的一组 Pods 是由标签选择算符定义的。同样，一个 ReplicationController 应该管理的 pods 的数量也是由标签选择算符定义的。

两个对象的标签选择算符都是在 json 或者 yaml 文件中使用映射定义的，并且只支持 基于等值需求的选择算符：

```
"selector": {  
    "component": "redis",  
}
```

或者

```
selector:  
  component: redis
```

这个选择算符(分别在 json 或者 yaml 格式中) 等价于 component=redis 或 component in (redis)。

支持基于集合需求的资源

比较新的资源，例如 [Job](#)、[Deployment](#)、[Replica Set](#) 和 [DaemonSet](#)，也支持 基于集合的需求。

```
selector:  
  matchLabels:  
    component: redis  
  matchExpressions:  
    - {key: tier, operator: In, values: [cache]}  
    - {key: environment, operator: NotIn, values: [dev]}
```

matchLabels 是由 {key,value} 对组成的映射。 matchLabels 映射中的单个 {key,value} 等同于 matchExpressions 的元素，其 key 字段为 "key"，operator 为 "In"，而 values 数组仅包含 "value"。 matchExpressions 是 Pod 选择算符需求的列表。有效的运算符包括 In、NotIn、Exists 和 DoesNotExist。在 In 和 NotIn 的情况下，设置的值必须是非空的。来自 matchLabels 和 matchExpressions 的所有要求都按逻辑与的关系组合到一起 -- 它们必须都满足才能匹配。

选择节点集

通过标签进行选择的一个用例是确定节点集，方便 Pod 调度。有关更多信息，请参阅[选择节点](#)文档。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

注解

你可以使用 Kubernetes 注解为对象附加任意的非标识的元数据。客户端程序（例如工具和库）能够获取这些元数据信息。

为对象附加元数据

你可以使用标签或注解将元数据附加到 Kubernetes 对象。标签可以用来选择对象和查找满足某些条件的对象集合。相反，注解不用于标识和选择对象。注解中的元数据，可以很小，也可以很大，可以是结构化的，也可以是非结构化的，能够包含标签不允许的字符。

注解和标签一样，是键/值对：

```
"metadata": {  
  "annotations": {  
    "key1": "value1",  
    "key2": "value2"  
  }  
}
```

以下是一些例子，用来说明哪些信息可以使用注解来记录：

- 由声明性配置所管理的字段。将这些字段附加为注解，能够将它们与客户端或服务端设置的默认值、自动生成的字段以及通过自动调整大小或自动伸缩系统设置的字段区分开来。
- 构建、发布或镜像信息（如时间戳、发布 ID、Git 分支、PR 数量、镜像哈希、仓库地址）。
- 指向日志记录、监控、分析或审计仓库的指针。
- 可用于调试目的的客户端库或工具信息：例如，名称、版本和构建信息。
- 用户或者工具/系统的来源信息，例如来自其他生态系统组件的相关对象的 URL。
- 轻量级上线工具的元数据信息：例如，配置或检查点。
- 负责人员的电话或呼机号码，或指定在何处可以找到该信息的目录条目，如团队网站。
- 从用户到最终运行的指令，以修改行为或使用非标准功能。

你可以将这类信息存储在外部数据库或目录中而不使用注解，但这样做就使得开发人员很难生成用于部署、管理、自检的客户端共享库和工具。

语法和字符集

注解 (*Annotations*) 存储的形式是键/值对。有效的注解键分为两部分：可选的前缀和名称，以斜杠 (/) 分隔。名称段是必需项，并且必须在63个字符以内，以字母数字字符 ([a-z0-9A-Z]) 开头和结尾，并允许使用破折号 (-)、下划线 (_)、点 (.) 和字母数字。前缀是可选的。如果指定，则前缀必须是DNS子域：一系列由点 (.) 分隔的DNS标签，总计不超过253个字符，后跟斜杠 (/)。如果省略前缀，则假定注解键对用户是私有的。由系统组件添加的注解（例如，kube-scheduler，kube-controller-manager，kube-apiserver，kubectl 或其他第三方组件），必须为终端用户添加注解前缀。

kubernetes.io/ 和 k8s.io/ 前缀是为Kubernetes核心组件保留的。

例如，下面是一个 Pod 的配置文件，其注解中包含 `imageregistry: https://hub.docker.com/`：

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

接下来

- 进一步了解[标签和选择算符](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 12, 2020 at 4:39 PM PST: [\[zh\] Sync changes from English site \(2\) \(8ae2209de\)](#)

字段选择器

字段选择器 (*Field selectors*) 允许你根据一个或多个资源字段的值 [筛选 Kubernetes 资源](#)。下面是一些使用字段选择器查询的例子：

- metadata.name=my-service
- metadata.namespace!=default
- status.phase=Pending

下面这个 kubectl 命令将筛选出 [status.phase](#) 字段值为 Running 的所有 Pod：

```
kubectl get pods --field-selector status.phase=Running
```

说明：

字段选择器本质上是资源过滤器 (*Filters*)。默认情况下，字段选择器/过滤器是未被应用的，这意味着指定类型的所有资源都会被筛选出来。这使得以下的两个 kubectl 查询是等价的：

```
kubectl get pods  
kubectl get pods --field-selector ""
```

支持的字段

不同的 Kubernetes 资源类型支持不同的字段选择器。所有资源类型都支持 `metadata.name` 和 `metadata.namespace` 字段。使用不被支持的字段选择器会产生错误。例如：

```
kubectl get ingress --field-selector foo.bar=baz
```

```
Error from server (BadRequest): Unable to find "ingresses" that match label  
selector "", field selector "foo.bar=baz": "foo.bar" is not a known field selector:  
only "metadata.name", "metadata.namespace"
```

支持的操作符

你可在字段选择器中使用 `=`、`==` 和 `!=` (`=` 和 `==` 的意义是相同的) 操作符。例如，下面这个 kubectl 命令将筛选所有不属于 default 命名空间的 Kubernetes 服务：

```
kubectl get services --all-namespaces --field-selector metadata.namespace!=def  
ault
```

链式选择器

同[标签](#)和其他选择器一样，字段选择器可以通过使用逗号分隔的列表组成一个选择链。下面这个 kubectl 命令将筛选 `status.phase` 字段不等于 `Running` 同时 `spec.restartPolicy` 字段等于 `Always` 的所有 Pod：

```
kubectl get pods --field-selector=status.phase!=Running,spec.restartPolicy=Always
```

多种资源类型

你能够跨多种资源类型来使用字段选择器。下面这个 kubectl 命令将筛选出所有不在 default 命名空间中的 StatefulSet 和 Service：

```
kubectl get statefulsets,services --all-namespaces --field-selector metadata.namespace!=default
```

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 August 05, 2020 at 2:37 PM PST: [\[zh\] Fix links in concepts section \(6\) \(d38315f27\)](#)

推荐使用的标签

除了 kubectl 和 dashboard 之外，您可以使用其他工具来可视化和管理 Kubernetes 对象。一组通用的标签可以让多个工具之间相互操作，用所有工具都能理解的通用方式描述对象。

除了支持工具外，推荐的标签还以一种可以查询的方式描述了应用程序。

元数据围绕 *应用* (*application*) 的概念进行组织。Kubernetes 不是 平台即服务 (PaaS)，没有或强制执行正式的应用程序概念。相反，应用程序是非正式的，并使用元数据进行描述。应用程序包含的定义是松散的。

说明：

这些是推荐的标签。它们使管理应用程序变得更容易但不是任何核心工具所必需的。

共享标签和注解都使用同一个前缀：app.kubernetes.io。没有前缀的标签是用户私有的。共享前缀可以确保共享标签不会干扰用户自定义的标签。

标签

为了充分利用这些标签，应该在每个资源对象上都使用它们。

键	描述	示例	类型
app.kubernetes.io/name	应用程序的名称	mysql	字符串
app.kubernetes.io/instance	用于唯一确定应用实例的名称	mysql-abcxyz	字符串
app.kubernetes.io/version	应用程序的当前版本（例如，语义版本，修订版哈希等）	5.7.21	字符串
app.kubernetes.io/component	架构中的组件	database	字符串
app.kubernetes.io/part-of	此级别的更高级别应用程序的名称	wordpress	字符串
app.kubernetes.io/managed-by	用于管理应用程序的工具	helm	字符串

为说明这些标签的实际使用情况，请看下面的 StatefulSet 对象：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  labels:
    app.kubernetes.io/name: mysql
    app.kubernetes.io/instance: mysql-abcxyz
    app.kubernetes.io/version: "5.7.21"
    app.kubernetes.io/component: database
    app.kubernetes.io/part-of: wordpress
    app.kubernetes.io/managed-by: helm
```

应用和应用实例

应用可以在 Kubernetes 集群中安装一次或多次。在某些情况下，可以安装在同一命名空间中。例如，可以不止一次地为不同的站点安装不同的 WordPress。

应用的名称和实例的名称是分别记录的。例如，某 WordPress 实例的 app.kubernetes.io/name 为 wordpress，而其实例名称表现为 app.kubernetes.io/instance 的属性值 wordpress-abcxyz。这使应用程序和应用程序的实例成为可能是可识别的。应用程序的每个实例都必须具有唯一的名称。

示例

为了说明使用这些标签的不同方式，以下示例具有不同的复杂性。

一个简单的无状态服务

考虑使用 Deployment 和 Service 对象部署的简单无状态服务的情况。以下两个代码段表示如何以最简单的形式使用标签。

下面的 Deployment 用于监督运行应用本身的 pods。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
...
```

下面的 Service 用于暴露应用。

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: myservice
    app.kubernetes.io/instance: myservice-abcxzy
...
...
```

带有一个数据库的 Web 应用程序

考虑一个稍微复杂的应用：一个使用 Helm 安装的 Web 应用（WordPress），其中 使用了数据库（MySQL）。以下代码片段说明用于部署此应用程序的对象的开始。

以下 Deployment 的开头用于 WordPress：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
    app.kubernetes.io/managed-by: helm
    app.kubernetes.io/component: server
    app.kubernetes.io/part-of: wordpress
...
...
```

这个 Service 用于暴露 WordPress：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/name: wordpress
    app.kubernetes.io/instance: wordpress-abcxzy
    app.kubernetes.io/version: "4.9.4"
```

```
app.kubernetes.io/managed-by: helm
app.kubernetes.io/component: server
app.kubernetes.io/part-of: wordpress
```

...

MySQL 作为一个 StatefulSet 暴露，包含它和它所属的较大应用程序的元数据：

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
labels:
  app.kubernetes.io/name: mysql
  app.kubernetes.io/instance: mysql-abcxyz
  app.kubernetes.io/version: "5.7.21"
  app.kubernetes.io/managed-by: helm
  app.kubernetes.io/component: database
  app.kubernetes.io/part-of: wordpress
```

...

Service 用于将 MySQL 作为 WordPress 的一部分暴露：

```
apiVersion: v1
kind: Service
metadata:
labels:
  app.kubernetes.io/name: mysql
  app.kubernetes.io/instance: mysql-abcxyz
  app.kubernetes.io/version: "5.7.21"
  app.kubernetes.io/managed-by: helm
  app.kubernetes.io/component: database
  app.kubernetes.io/part-of: wordpress
```

...

使用 MySQL StatefulSet 和 Service，您会注意到有关 MySQL 和 Wordpress 的信息，包括更广泛的应用程序。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 January 07, 2021 at 12:47 AM PST: [\[zh\] corrected instance of WordPress written incorrectly \(695ad01aa\)](#)

Kubernetes 架构

Kubernetes 背后的架构概念。

节点

[控制面到节点通信](#)

控制器

[云控制器管理器的基础概念](#)

节点

Kubernetes 通过将容器放入在节点 (Node) 上运行的 Pod 中来执行你的工作负载。节点可以是一个虚拟机或者物理机器，取决于所在的集群配置。每个节点都包含用于运行 Pod 所需要的服务，这些服务由 [控制面](#)负责管理。

通常集群中会有若干个节点；而在一个学习用或者资源受限的环境中，你的集群中也可能只有一个节点。

节点上的[组件](#)包括 [kubelet](#)、[容器运行时](#)以及 [kube-proxy](#)。

管理

向 [API 服务器](#)添加节点的方式主要有两种：

1. 节点上的 kubelet 向控制面执行自注册；
2. 你，或者别的什么人，手动添加一个 Node 对象。

在你创建了 Node 对象或者节点上的 kubelet 执行了自注册操作之后，控制面会检查新的 Node 对象是否合法。例如，如果你使用下面的 JSON 对象来创建 Node 对象：

```
{  
  "kind": "Node",  
  "apiVersion": "v1",  
  "metadata": {  
    "name": "10.240.79.157",  
    "labels": {  
      "name": "my-first-k8s-node"  
    }  
  }  
}
```

Kubernetes 会在内部创建一个 Node 对象作为节点的表示。Kubernetes 检查 kubelet 向 API 服务器注册节点时使用的 metadata.name 字段是否匹配。如果节点是健康的

(即所有必要的服务都在运行中) , 则该节点可以用来运行 Pod。否则 , 直到该节点变为健康之前 , 所有的集群活动都会忽略该节点。

说明 : Kubernetes 会一直保存着非法节点对应的对象 , 并持续检查该节点是否已经 变得健康。你 , 或者某个[控制器](#)必需显式地 删除该 Node 对象以停止健康检查操作。

Node 对象的名称必须是合法的 [DNS 子域名](#)。

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 API 服务器表明身份的凭据路径。
- --cloud-provider - 与某[云驱动](#) 进行通信以读取与自身相关的元数据的方式。
- --register-node - 自动向 API 服务注册。
- --register-with-taints - 使用所给的污点列表 (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](#)。 (参见 [NodeRestriction 准入控制插件](#) 所实施的标签限制) 。
- --node-status-update-frequency - 指定 kubelet 向控制面发送状态的频率。

启用[节点授权模式](#)和 [NodeRestriction 准入插件](#) 时 , 仅授权 kubelet 创建或修改其自己的节点资源。

手动节点管理

你可以使用 [kubectl](#) 来创建和修改 Node 对象。

如果你希望手动创建节点对象时 , 请设置 kubelet 标志 --register-node=false。

你可以修改 Node 对象 (忽略 --register-node 设置) 。例如 , 修改节点上的标签或标记其为不可调度。

你可以结合使用节点上的标签和 Pod 上的选择算符来控制调度。例如 , 你可以限制某 Pod 只能在符合要求的节点子集上运行。

如果标记节点为不可调度 (unschedulable) , 将阻止新 Pod 调度到该节点之上 , 但不会 影响任何已经在其上的 Pod。这是重启节点或者执行其他维护操作之前的一个有用的准备步骤。

要标记一个节点为不可调度 , 执行以下命令 :

```
kubectl cordon $NODENAME
```

说明：被 [DaemonSet](#) 控制器创建的 Pod 能够容忍节点的不可调度属性。 DaemonSet 通常提供节点本地的服务，即使节点上的负载应用已经被腾空，这些服务也仍需运行在节点之上。

节点状态

一个节点的状态包含以下信息：

- [地址](#)
- [状况](#)
- [容量与可分配](#)
- [信息](#)

你可以使用 kubectl 来查看节点状态和其他细节信息：

```
kubectl describe node <节点名称>
```

下面对每个部分进行详细描述。

地址

这些字段的用法取决于你的云服务商或者物理机配置。

- HostName：由节点的内核设置。可以通过 kubelet 的 --hostname-override 参数覆盖。
- ExternalIP：通常是节点的可外部路由（从集群外可访问）的 IP 地址。
- InternalIP：通常是节点的仅可在集群内部路由的 IP 地址。

状况

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

节点状况	描述
Ready	如节点是健康的并已经准备好接收 Pod 则为 True；False 表示节点不健康而且不能接收 Pod；Unknown 表示节点控制器在最近 node-monitor-grace-period 期间（默认 40 秒）没有收到节点的消息
DiskPressure	True 表示节点的空闲空间不足以用于添加新 Pod, 否则为 False
MemoryPressure	True 表示节点存在内存压力，即节点内存可用量低，否则为 False
PIDPressure	True 表示节点存在进程压力，即节点上进程过多；否则为 False
NetworkUnavailable	True 表示节点网络配置不正确；否则为 False

说明：如果使用命令行工具来打印已保护（Cordoned）节点的细节，其中的 Condition 字段可能包括 SchedulingDisabled。SchedulingDisabled 不是 Kubernetes API 中定义的 Condition，被保护起来的节点在其规约中被标记为不可调度（Unschedulable）。

节点条件使用 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 Unknown 或者 False 状态的时间超过了 pod-eviction-timeout 值，（一个传递给 [kube-controller-manager](#) 的参数），节点上的所有 Pod 都会被节点控制器计划删除。默认的逐出超时时长为 **5 分钟**。某些情况下，当节点不可达时，API 服务器不能和其上的 kubelet 通信。删除 Pod 的决定不能传达给 kubelet，直到它重新建立和 API 服务器的连接为止。与此同时，被计划删除的 Pod 可能会继续在游离的节点上运行。

节点控制器在确认 Pod 在集群中已经停止运行前，不会强制删除它们。你可以看到这些可能在无法访问的节点上运行的 Pod 处于 Terminating 或者 Unknown 状态。如果 kubernetes 不能基于下层基础设施推断出某节点是否已经永久离开了集群，集群管理员可能需要手动删除该节点对象。从 Kubernetes 删除节点对象将导致 API 服务器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状况的 [污点](#)。当调度器将 Pod 指派给某节点时，会考虑节点上的污点。Pod 则可以通过容忍度（Toleration）表达所能容忍的污点。

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 Pod 的个数上限。

capacity 块中的字段标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[预留计算资源](#)的时候了解有关容量和可分配资源的更多信息。

信息

关于节点的一般性信息，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。

节点控制器

节点[控制器](#)是 Kubernetes 控制面组件，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多个角色。第一个是当节点注册时为它分配一个 CIDR 区段（如果启用了 CIDR 分配）。

第二个是保持节点控制器内的节点列表与云服务商所提供的可用机器列表同步。如果在云环境下运行，只要某节点不健康，节点控制器就会询问云服务是否节点的虚拟机仍可用。如果不可用，节点控制器会将该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可达（即，节点控制器因为某些原因没有收到心跳，例如节点宕机）时，将节点状态的 NodeReady 状况更新为 "Unknown"。如果节点接下来持续处于不可达状态，节点控制器将逐出节点上的所有 Pod（使用体面终止）。默认情况下 40 秒后开始报告 "Unknown"，在那之后 5 分钟开始逐出 Pod。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（Heartbeats）有助于确定节点的可用性。心跳有两种形式：NodeStatus 和 [Lease 对象] ([/docs/reference/generated/kubernetes-api/v1.20/#lease-v1-coordination-k8s-io](#))。每个节点在 `kube-node-lease` 名字空间中都有一个与之关联的 Lease 对象。Lease 是一种轻量级的资源，可在集群规模扩大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事件时，`kubelet` 会更新 NodeStatus。NodeStatus 更新的默认间隔为 5 分钟（比不可达节点的 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。如果 Lease 的更新操作失败，`kubelet` 会采用指数回退机制，从 200 毫秒开始重试，最长重试间隔为 7 秒钟。

可靠性

大部分情况下，节点控制器把逐出速率限制在每秒 --node-eviction-rate 个（默认为 0.1）。这表示它每 10 秒钟内至多从一个节点驱逐 Pod。

当一个可用区域（Availability Zone）中的节点变为不健康时，节点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 状况为 Unknown 或 False）的节点的百分比。如果不健康节点的比例超过 --unhealthy-zone-threshold（默认为 0.55），驱逐速率将会降低：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控制面脱离时其它可用区域可能仍然保持连接。如果你的集群没有跨越云服务商的多个可用区域，那（整个集群）就只有一个可用区域。

跨多个可用区域部署你的节点的一个关键原因是当某个可用区域整体出现故障时，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，节点控制器将假设控制面节点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。

节点控制器还负责驱逐运行在拥有 NoExecute 污点的节点上的 Pod，除非这些 Pod 能够容忍此污点。节点控制器还负责根据节点故障（例如节点不可访问或没有就绪）为其添加 [污点](#)。这意味着调度器不会将 Pod 调度到不健康的节点上。

注意： kubectl cordon 会将节点标记为“不可调度（Unschedulable）”。此操作的副作用是，服务控制器会将该节点从负载均衡器中之前的目标节点列表中移除，从而使得来自负载均衡器的网络请求不会到达被保护起来的节点。

节点容量

Node 对象会跟踪节点上资源的容量（例如可用内存和 CPU 数量）。通过[自注册](#)机制生成的 Node 对象会在注册期间报告自身容量。如果你[手动](#)添加了 Node，你就需要在添加节点时手动设置节点容量。

Kubernetes [调度器](#)保证节点上有足够的资源供其上的所有 Pod 使用。它会检查节点上所有容器的请求的总和不会超过节点的容量。总的请求包括由 kubelet 启动的所有容器，但不包括由容器运行时直接启动的容器，也不包括不受 kubelet 控制的其他进程。

说明： 如果要为非 Pod 进程显式保留资源。请参考 [为系统守护进程预留资源](#)。

节点拓扑

FEATURE STATE: Kubernetes v1.16 [alpha]

如果启用了 TopologyManager [特性门控](#)，kubelet 可以在作出资源分配决策时使用拓扑提示。参考[控制节点上拓扑管理策略](#) 了解详细信息。

节点体面关闭

FEATURE STATE: Kubernetes v1.20 [alpha]

如果你启用了 GracefulNodeShutdown [特性门控](#)，那么 kubelet 尝试检测节点的系统关闭事件并终止在节点上运行的 Pod。在节点终止期间，kubelet 保证 Pod 遵从常规的 [Pod 终止流程](#)。

当启用了 GracefulNodeShutdown 特性门控时，kubelet 使用 [systemd 抑制器锁](#) 在给定的期限内延迟节点关闭。在关闭过程中，kubelet 分两个阶段终止 Pod：

1. 终止在节点上运行的常规 Pod。
2. 终止在节点上运行的[关键 Pod](#)。

节点体面关闭的特性对应两个 [KubeletConfiguration](#) 选项：

- ShutdownGracePeriod：
 - 指定节点应延迟关闭的总持续时间。此时间是 Pod 体面终止的时间总和，不分常规 Pod 还是 [关键 Pod](#)。

- ShutdownGracePeriodCriticalPods：
 - 在节点关闭期间指定用于终止 [关键 Pod](#) 的持续时间。该值应小于 Shutdown GracePeriod。

例如，如果设置了 ShutdownGracePeriod=30s 和 ShutdownGracePeriodCriticalPods=10s，则 kubelet 将延迟 30 秒关闭节点。在关闭期间，将保留前 20 (30 - 10) 秒用于体面终止常规 Pod，而保留最后 10 秒用于终止 [关键 Pod](#)。

接下来

- 了解有关节点[组件](#)
- 阅读[节点的 API 定义](#)
- 阅读架构设计文档中有关[节点](#)的章节
- 了解[污点和容忍度](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 18, 2021 at 11:05 AM PST: [sync content/zh/docs/concepts/architecture/nodes.md \(d6ad8266b\)](#)

控制面到节点通信

本文列举控制面节点（确切说是 API 服务器）和 Kubernetes 集群之间的通信路径。目的是为了让用户能够自定义他们的安装，以实现对网络配置的加固，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开的 IP 上）运行。

节点到控制面

Kubernetes 采用的是中心辐射型（Hub-and-Spoke）API 模式。所有从集群（或所运行的 Pods）发出的 API 调用都终止于 apiserver（其它控制面组件都没有被设计为可暴露远程服务）。apiserver 被配置为在一个安全的 HTTPS 端口（443）上监听远程连接请求，并启用一种或多种形式的客户端[身份认证](#)机制。一种或多种客户端[鉴权机制](#)应该被启用，特别是在允许使用[匿名请求](#)或[服务账号令牌](#)的时候。

应该使用集群的公共根证书开通节点，这样它们就能够基于有效的客户端凭据安全地连接 apiserver。一种好的方法是以客户端证书的形式将客户端凭据提供给 kubelet。请查看 [kubelet TLS 启动引导](#) 以了解如何自动提供 kubelet 客户端证书。

想要连接到 apiserver 的 Pod 可以使用服务账号安全地进行连接。当 Pod 被实例化时，Kubernetes 自动把公共根证书和一个有效的持有者令牌注入到 Pod 里。kubernetes

tes 服务（位于所有名字空间中）配置了一个虚拟 IP 地址，用于（通过 kube-proxy）转发请求到 apiserver 的 HTTPS 末端。

控制面组件也通过安全端口与集群的 apiserver 通信。

这样，从集群节点和节点上运行的 Pod 到控制面的连接的缺省操作模式即是安全的，能够在不可信的网络或公网上运行。

控制面到节点

从控制面（apiserver）到节点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 通过它的代理功能连接到任何节点、Pod 或者服务。

API 服务器到 kubelet

从 apiserver 到 kubelet 的连接用于：

- 获取 Pod 日志
- 挂接（通过 kubectl）到运行中的 Pod
- 提供 kubelet 的端口转发功能。

这些连接终止于 kubelet 的 HTTPS 末端。默认情况下，apiserver 不检查 kubelet 的服务证书。这使得此类连接容易受到中间人攻击，在非受信网络或公开网络上运行也是 **不安全的**。

为了对这个连接进行认证，使用 `--kubelet-certificate-authority` 标志给 apiserver 提供一个根证书包，用于 kubelet 的服务证书。

如果无法实现这点，又要求避免在非受信网络或公共网络上进行连接，可在 apiserver 和 kubelet 之间使用 [SSH 隧道](#)。

最后，应该启用 [kubelet 用户认证和/或鉴权](#) 来保护 kubelet API。

apiserver 到节点、Pod 和服务

从 apiserver 到节点、Pod 或服务的连接默认为纯 HTTP 方式，因此既没有认证，也没有加密。这些连接可通过给 API URL 中的节点、Pod 或服务名称添加前缀 https：来运行在安全的 HTTPS 连接上。不过这些连接既不会验证 HTTPS 末端提供的证书，也不会提供客户端证书。因此，虽然连接是加密的，仍无法提供任何完整性保证。这些连接 **目前还不能安全地** 在非受信网络或公共网络上运行。

SSH 隧道

Kubernetes 支持使用 SSH 隧道来保护从控制面到节点的通信路径。在这种配置下，apiserver 建立一个到集群中各节点的 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、节点、Pod 或服务的请求。这一隧道保证通信不会被暴露到集群节点所运行的网络之外。

SSH 隧道目前已被废弃。除非你了解个中细节，否则不应使用。Konnectivity 服务是对此通信通道的替代品。

Konnectivity 服务

FEATURE STATE: Kubernetes v1.18 [beta]

作为 SSH 隧道的替代方案，Konnectivity 服务提供 TCP 层的代理，以便支持从控制面到集群的通信。Konnectivity 服务包含两个部分：Konnectivity 服务器和 Konnectivity 代理，分别运行在控制面网络和节点网络中。Konnectivity 代理建立并维持到 Konnectivity 服务器的网络连接。启用 Konnectivity 服务之后，所有控制面到节点的通信都通过这些连接传输。

请浏览 [Konnectivity 服务任务](#) 在你的集群中配置 Konnectivity 服务。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

控制器

在机器人技术和自动化领域，控制回路（Control Loop）是一个非终止回路，用于调节系统状态。

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（Desired State）。房间的实际温度是当前状态（Current State）。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。

在 Kubernetes 中，控制器通过监控[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

一个控制器至少追踪一种类型的 Kubernetes 资源。这些[对象](#)有一个代表期望状态的 spec 字段。该资源的控制器负责确保其当前状态接近期望状态。

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](#)，这会有副作用。具体可参看后文的例子。

通过 API 服务器来控制

[Job](#) 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，它运行一个或者多个 [Pod](#)，来执行一个任务然后停止。
(一旦被调度了，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。控制面中的其它组件根据新的消息作出反应(调度并运行新 Pod)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回路来保证集群中有足够的 [节点](#)，那么控制器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。请参阅

这里，很重要的一点是，控制器做出了一些变更以使得事物更接近你的期望状态，之后将当前状态报告给集群的 API 服务器。其他控制回路可以观测到所汇报的数据的这种变化并采取其各自的行动。

在温度计的例子中，如果房间很冷，那么某个控制器可能还会启动一个防冻加热器。就 Kubernetes 集群而言，控制面间接地与 IP 地址管理工具、存储服务、云驱动 APIs 以及其他服务协作，通过[扩展 Kubernetes](#) 来实现这点。

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，并且控制回路会自动修复故障。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。

设计

作为设计原则之一，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。最常见的是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演化。

使用简单的控制器而不是一组相互连接的单体控制回路是很有用的。控制器会失败，所以 Kubernetes 的设计正是考虑到了这一点。

说明：

可以有多个控制器来创建或者更新相同类型的对象。在后台，Kubernetes 控制器确保它们只关心与其控制资源相关联的资源。

例如，你可以创建 Deployment 和 Job；它们都可以创建 Pod。Job 控制器不会删除 Deployment 所创建的 Pod，因为有信息（[标签](#)）让控制器可以区分这些 Pod。

运行控制器的方式

Kubernetes 内置一组控制器，运行在 [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 允许你运行一个稳定的控制平面，这样即使某些内置控制器失败了，控制平面的其他部分会接替它们的工作。

你会遇到某些控制器运行在控制面之外，用以扩展 Kubernetes。或者，如果你愿意，你也可以自己编写新控制器。你可以以一组 Pod 来运行你的控制器，或者运行在 Kubernetes 之外。最合适方案取决于控制器所要执行的功能是什么。

接下来

- 阅读 [Kubernetes 控制平面组件](#)
- 了解 [Kubernetes 对象](#) 的一些基本知识
- 进一步学习 [Kubernetes API](#)
- 如果你想编写自己的控制器，请看 Kubernetes 的 [扩展模式](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

云控制器管理器的基础概念

FEATURE STATE: Kubernetes v1.11 [beta]

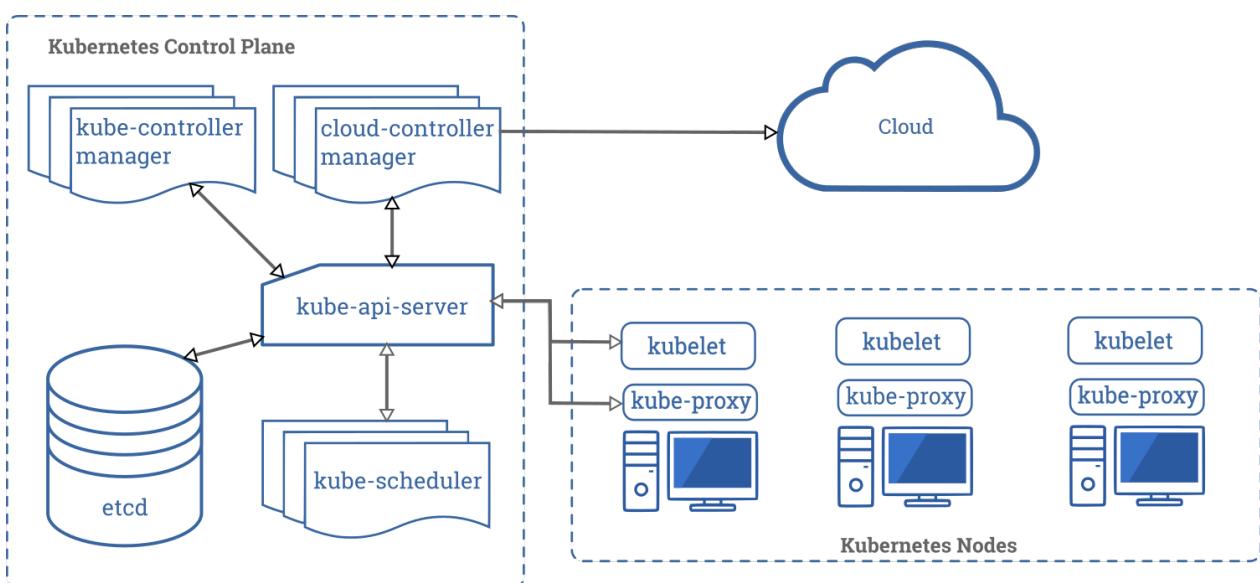
使用云基础设施技术，你可以在公有云、私有云或者混合云环境中运行 Kubernetes。Kubernetes 的信条是基于自动化的、API 驱动的基础设施，同时避免组件间紧密耦合。

组件 cloud-controller-manager 是云控制器管理器是指嵌入特定云的控制逻辑的 [控制平面](#) 组件。云控制器管理器允许您链接聚合到云提供商的应用编程接口中，并分离出相互作用的组件与您的集群交互的组件。

通过分离 Kubernetes 和底层云基础设施之间的互操作性逻辑，云控制器管理器组件使云提供商能够以不同于 Kubernetes 主项目的速度进行发布新特征。

cloud-controller-manager 组件是基于一种插件机制来构造的，这种机制使得不同的云厂商都能将其平台与 Kubernetes 集成。

设计



云控制器管理器以一组多副本的进程集合的形式运行在控制面中，通常表现为 Pod 中的容器。每个 cloud-controller-manager 在同一进程中实现多个 [控制器](#)。

说明：你也可以以 Kubernetes [插件](#) 的形式而不是控制面中的一部分来运行云控制器管理器。

云控制器管理器的功能

云控制器管理器中的控制器包括：

节点控制器

节点控制器负责在云基础设施中创建了新服务器时为之 创建 [节点 \(Node\)](#) 对象。 节点控制器从云提供商获取当前租户中主机的信息。 节点控制器执行以下功能：

1. 针对控制器通过云平台驱动的 API 所发现的每个服务器初始化一个 Node 对象；
2. 利用特定云平台的信息为 Node 对象添加注解和标签，例如节点所在的 区域（Region）和所具有的资源（CPU、内存等等）；
3. 获取节点的网络地址和主机名；
4. 检查节点的健康状况。如果节点无响应，控制器通过云平台 API ll 查看该节点是否已从云中禁用、删除或终止。如果节点已从云中删除，则控制器从 Kubernetes 集群中删除 Node 对象。

某些云驱动实现中，这些任务被划分到一个节点控制器和一个节点生命周期控制器中。

路由控制器

Route 控制器负责适当地配置云平台中的路由，以便 Kubernetes 集群中不同节点上的容器之间可以相互通信。

取决于云驱动本身，路由控制器可能也会为 Pod 网络分配 IP 地址块。

服务控制器

[服务 \(Service\)](#) 与受控的负载均衡器、IP 地址、网络包过滤、目标健康检查等云基础设施组件集成。服务控制器与云驱动的 API 交互，以配置负载均衡器和其他基础设施组件。你所创建的 Service 资源会需要这些组件服务。

鉴权

本节分别讲述云控制器管理器为了完成自身工作而产生的对各类 API 对象的访问需求。

节点控制器

节点控制器只操作 Node 对象。它需要读取和修改 Node 对象的完全访问权限。

v1/Node:

- Get
- List
- Create
- Update
- Patch
- Watch
- Delete

路由控制器

路由控制器会监听 Node 对象的创建事件，并据此配置路由设施。它需要读取 Node 对象的 Get 权限。

v1/Node:

- Get

服务控制器

服务控制器监测 Service 对象的 Create、Update 和 Delete 事件，并配置对应服务的 Endpoints 对象。为了访问 Service 对象，它需要 List、Watch 访问权限；为了更新 Service 对象，它需要 Patch 和 Update 访问权限。为了能够配置 Service 对应的 Endpoints 资源，它需要 Create、List、Get、Watch 和 Update 等访问权限。

v1/Service:

- List
- Get
- Watch
- Patch
- Update

其他

云控制器管理器的实现中，其核心部分需要创建 Event 对象的访问权限以及 创建 ServiceAccount 资源以保证操作安全性的权限。

v1/Event:

- Create
- Patch
- Update

v1/ServiceAccount:

- Create

用于云控制器管理器 [RBAC](#) 的 ClusterRole 如下例所示：

```
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

接下来

[云控制器管理器的管理](#) 给出了运行和管理云控制器管理器的指南。

想要了解如何实现自己的云控制器管理器，或者对现有项目进行扩展么？

云控制器管理器使用 Go 语言的接口，从而使得针对各种云平台的具体实现都可以接入。其中使用了在 [kubernetes/cloud-provider](#) 项目中 [cloud.go](#) 文件所定义的 CloudProvider 接口。

本文中列举的共享控制器（节点控制器、路由控制器和服务控制器等）的实现以及 其他一些生成具有 CloudProvider 接口的框架的代码，都是 Kubernetes 的核心代码。特定于云驱动的实现虽不是 Kubernetes 核心成分，仍要实现 CloudProvider 接口。

关于如何开发插件的详细信息，可参考 [开发云控制器管理器](#) 文档。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 17, 2020 at 12:32 PM PST: [【ZH】typo : 云控制器管理器的基础概念 #25060 \(9fe5011ec\)](#)

容器

打包应用及其运行依赖环境的技术。

每个运行的容器都是可重复的；包含依赖环境在内的标准，意味着无论您在哪里运行它，您都会得到相同的行为。

容器将应用程序从底层的主机设施中解耦。这使得在不同的云或 OS 环境中部署更加容易。

容器镜像

[容器镜像](#)是一个随时可以运行的软件包，包含运行应用程序所需的一切：代码和它需要的所有运行时、应用程序和系统库，以及一些基本设置的默认值。

根据设计，容器是不可变的：你不能更改已经运行的容器的代码。如果有一个容器化的应用程序需要修改，则需要构建包含更改的新镜像，然后再基于新构建的镜像重新运行容器。

容器运行时

容器运行环境是负责运行容器的软件。

Kubernetes 支持多个容器运行环境: [Docker](#)、[containerd](#)、[CRI-O](#) 以及任何实现 [Kubernetes CRI \(容器运行环境接口\)](#)。

接下来

- 进一步阅读[容器镜像](#)
- 进一步阅读[Pods](#)

镜像

容器镜像 (Image) 所承载的是封装了应用程序及其所有软件依赖的二进制数据。容器镜像是可执行的软件包，可以单独运行；该软件包对所处的运行时环境具有 良定 (Well Defined) 的假定。

你通常会创建应用的容器镜像并将其推送到某仓库，然后在 [Pod](#) 中引用它。

本页概要介绍容器镜像的概念。

镜像名称

容器镜像通常会被赋予 pause、example/mycontainer 或者 kube-apiserver 这类的名称。镜像名称也可以包含所在仓库的主机名。例如：fictional.registry.example/imagename。还可以包含仓库的端口号，例如：fictional.registry.example:10443/imagename。

如果你不指定仓库的主机名，Kubernetes 认为你在使用 Docker 公共仓库。

在镜像名称之后，你可以添加一个 标签 (Tag)（就像在 docker 或 podman 中也在用的那样）。使用标签能让你辨识同一镜像序列中的不同版本。

镜像标签可以包含小写字母、大写字符、数字、下划线 (_)、句点 (.) 和连字符 (-)。关于在镜像标签中何处可以使用分隔字符 (_)、(-) 和 (.) 还有一些额外的规则。如果你不指定标签，Kubernetes 认为你想使用标签 latest。

注意：

你要避免在生产环境中使用 latest 标签，因为这会使得跟踪所运行的镜像版本变得 非常困难，同时也很难回滚到之前运行良好的版本。

正确的做法恰恰相反，你应该指定一个有意义的标签，如 v1.42.0。

更新镜像

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。如果希望强制总是拉取镜像，你可以执行以下操作之一：

- 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](#) 准入控制器（Admission Controller）。

如果 imagePullPolicy 未被定义为特定的值，也会被设置为 Always。

带镜像索引的多架构镜像

除了提供二进制的镜像之外，容器仓库也可以提供 [容器镜像索引](#)。镜像索引可以根据特定于体系结构版本的容器指向镜像的多个 [镜像清单](#)。这背后的理念是让你可以为镜像命名（例如：pause、example/mycontainer、kube-apiserver）的同时，允许不同的系统基于它们所使用的机器体系结构取回正确的二进制镜像。

Kubernetes 自身通常在命名容器镜像时添加后缀 -\$(ARCH)。为了向前兼容，请在生成较老的镜像时也提供后缀。这里的理念是为某镜像（如 pause）生成针对所有平台都适用的清单时，生成 pause-amd64 这类镜像，以便较老的配置文件或者将镜像后缀影编码到其中的 YAML 文件也能兼容。

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：

- 配置节点向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所有镜像
 - 需要所有节点的 root 访问权限才能进行设置
- 在 Pod 中设置 ImagePullSecrets
 - 只有提供自己密钥的 Pod 才能访问私有仓库
- 特定于厂商的扩展或者本地扩展
 - 如果你在使用定制的节点配置，你（或者云平台提供商）可以实现让节点向容器仓库认证的机制

下面将详细描述每一项。

配置 Node 对私有仓库认证

如果你在节点上运行的是 Docker，你可以配置 Docker 容器运行时来向私有容器仓库认证身份。

此方法适用于能够对节点进行配置的场合。

说明：Kubernetes 默认仅支持 Docker 配置中的 auths 和 HttpHeaders 部分，不支持 Docker 凭据辅助程序（ credHelpers 或 credsStore ）。

Docker 将私有仓库的密钥保存在 \$HOME/.dockercfg 或 \$HOME/.docker/config.json 文件中。如果你将相同的文件放在下面所列的搜索路径中，kubelet 会在拉取镜像时将其用作凭据 数据来源：

- {--root-dir:-/var/lib/kubelet}/config.json
- {kubelet 当前工作目录}/config.json
- \${HOME}/.docker/config.json
- /.docker/config.json
- {--root-dir:-/var/lib/kubelet}/.dockercfg
- {kubelet 当前工作目录}/.dockercfg
- \${HOME}/.dockercfg
- /.dockercfg

说明：你可能不得不为 kubelet 进程显式地设置 HOME=/root 环境变量。

推荐采用如下步骤来配置节点以便访问私有仓库。以下示例中，在 PC 或笔记本电脑中操作：

1. 针对你要使用的每组凭据，运行 docker login [服务器] 命令。这会更新你本地环境中的 \$HOME/.docker/config.json 文件。
2. 在编辑器中打开查看 \$HOME/.docker/config.json 文件，确保其中仅包含你要使用的凭据信息。
3. 获得节点列表；例如：
 - 如果想要节点名称：nodes=\$(kubectl get nodes -o jsonpath='{range.items[*].metadata}{.name} {end}')
 - 如果想要节点 IP，nodes=\$(kubectl get nodes -o jsonpath='{range .items[*].status.addresses[?(@.type == "ExternalIP")]}{.address} {end}')
4. 将本地的 .docker/config.json 拷贝到所有节点，放入如上所列的目录之一：
 - 例如，可以试一下：for n in \$nodes; do scp ~/.docker/config.json root@"\$n":/var/lib/kubelet/config.json; done

说明：对于产品环境的集群，可以使用配置管理工具来将这些设置应用到你所期望的节点上。

创建使用私有镜像的 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
```

输出类似于：

```
pod/private-image-test-1 created
```

如果一切顺利，那么一段时间后你可以执行：

```
kubectl logs private-image-test-1
```

然后可以看到命令的输出：

```
SUCCESS
```

如果你怀疑命令失败了，你可以运行：

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

如果命令确实失败，输出类似于：

```
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
```

你必须确保集群中所有节点的 `.docker/config.json` 文件内容相同。否则，Pod 会能在一些节点上正常运行而无法在另一些节点上启动。例如，如果使用节点自动扩缩，那么每个实例模板都需要包含 `.docker/config.json`，或者挂载一个包含该文件的驱动器。

在 `.docker/config.json` 中配置了私有仓库密钥后，所有 Pod 都将能读取私有仓库中的镜像。

提前拉取镜像

说明：该方法适用于你能够控制节点配置的场合。如果你的云供应商负责管理节点并自动置换节点，这一方案无法可靠地工作。

默认情况下，`kubelet` 会尝试从指定的仓库拉取每个镜像。但是，如果容器属性 `imagePullPolicy` 设置为 `IfNotPresent` 或者 `Never`，则会优先使用（对应 `IfNotPresent`）或者一定使用（对应 `Never`）本地镜像。

如果你希望使用提前拉取镜像的方法代替仓库认证，就必须保证集群中所有节点提前拉取的镜像是相同的。

这一方案可以用来提前载入指定的镜像以提高速度，或者作为向私有仓库执行身份认证的一种替代方案。

所有的 Pod 都可以使用节点上提前拉取的镜像。

在 Pod 上指定 ImagePullSecrets

说明：运行使用私有仓库中镜像的容器时，建议使用这种方法。

Kubernetes 支持在 Pod 中设置容器镜像仓库的密钥。

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 <名称> \
--docker-server=DOCKER_REGISTRY_SERVER \
--docker-username=DOCKER_USER \
--docker-password=DOCKER_PASSWORD \
--docker-email=DOCKER_EMAIL
```

如果你已经有 Docker 凭据文件，则可以将凭据文件导入为 Kubernetes [Secret](#)，而不是执行上面的命令。[基于已有的 Docker 凭据创建 Secret](#) 解释了如何完成这一操作。

如果你在使用多个私有容器仓库，这种技术将特别有用。原因是 kubectl create secret docker-registry 创建的是仅适用于某个私有仓库的 Secret。

说明：Pod 只能引用位于自身所在名字空间中的 Secret，因此需要针对每个名字空间 重复执行上述过程。

在 Pod 中引用 ImagePullSecrets

现在，在创建 Pod 时，可以在 Pod 定义中增加 imagePullSecrets 部分来引用该 Secret。

例如：

```
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
```

你需要对使用私有仓库的每个 Pod 执行以上操作。不过，设置该字段的过程也可以通过为 [服务账号](#) 资源设置 imagePullSecrets 来自动完成。有关详细指令可参见 [将 ImagePullSecrets 添加到服务账号](#)。

你也可以将此方法与节点级别的 .docker/config.json 配置结合使用。来自不同来源的凭据会被合并。

使用案例

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. 集群运行非专有镜像（例如，开源镜像）。镜像不需要隐藏。
 - 使用 Docker hub 上的公开镜像
 - 无需配置
 - 某些云厂商会自动为公开镜像提供高速缓存，以便提升可用性并缩短拉取镜像所需时间
1. 集群运行一些专有镜像，这些镜像需要对公司外部隐藏，对所有集群用户可见
 - 使用托管的私有 [Docker 仓库](#)。
 - 可以托管在 [Docker Hub](#) 或者其他地方
 - 按照上面的描述，在每个节点上手动配置 .docker/config.json 文件
 - 或者，在防火墙内运行一个组织内部的私有仓库，并开放读取权限
 - 不需要配置 Kubenretes
 - 使用控制镜像访问的托管容器镜像仓库服务
 - 与手动配置节点相比，这种方案能更好地处理集群自动扩缩容
 - 或者，在不方便更改节点配置的集群中，使用 imagePullSecrets
1. 集群使用专有镜像，且有些镜像需要更严格的访问控制
 - 确保 [AlwaysPullImages 准入控制器](#)被启用。否则，所有 Pod 都可以使用所有镜像。
 - 确保将敏感数据存储在 Secret 资源中，而不是将其打包在镜像里
1. 集群是多租户的并且每个租户需要自己的私有仓库
 - 确保 [AlwaysPullImages 准入控制器](#)。否则，所有租户的所有的 Pod 都可以使用所有镜像。
 - 为私有仓库启用鉴权
 - 为每个租户生成访问仓库的凭据，放置在 Secret 中，并将 Secret 发布到各租户的命名空间下。

- 租户将 Secret 添加到每个名字空间中的 imagePullSecrets

如果你需要访问多个仓库，可以为每个仓库创建一个 Secret。 kubelet 将所有 imagePullSecrets 合并为一个虚拟的 .docker/config.json 文件。

接下来

- 阅读 [OCI Image Manifest 规范](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 28, 2020 at 7:15 AM PST: [update \(1f4b65209\)](#)

容器环境

本页描述了在容器环境里容器可用的资源。

容器环境

Kubernetes 的容器环境给容器提供了几个重要的资源：

- 文件系统，其中包含一个[镜像](#) 和一个或多个的[卷](#)
- 容器自身的[信息](#)
- 集群中其他对象的[信息](#)

容器信息

容器的 `hostname` 是它所运行在的 pod 的名称。它可以通过 `hostname` 命令或者调用 `libc` 中的 [gethostname](#) 函数来获取。

Pod 名称和命名空间可以通过 [下行 API](#) 转换为环境变量。

Pod 定义中的用户所定义的环境变量也可在容器中使用，就像在 Docker 镜像中静态指定的任何环境变量一样。

集群信息

创建容器时正在运行的所有服务的列表都可用作该容器的环境变量。这些环境变量与 Docker 链接的语法匹配。

对于名为 *foo* 的服务，当映射到名为 *bar* 的容器时，以下变量是被定义了的：

`FOO_SERVICE_HOST`=<the host the service is running on>
`FOO_SERVICE_PORT`=<the port the service is running on>

服务具有专用的 IP 地址。如果启用了 [DNS插件](#)，可以在容器中通过 DNS 来访问服务。

接下来

- 学习更多有关[容器生命周期回调](#)的知识
- 动手[为容器生命周期事件添加处理程序](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 August 05, 2020 at 2:37 PM PST: [\[zh\] Fix links in concepts section \(6\) \(d38315f27\)](#)

容器运行时类(Runtime Class)

FEATURE STATE: Kubernetes v1.20 [stable]

本页面描述了 RuntimeClass 资源和运行时的选择机制。

RuntimeClass 是一个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。

动机

你可以在不同的 Pod 设置不同的 RuntimeClass，以提供性能与安全性之间的平衡。例如，如果你的部分工作负载需要高级别的信息安全保证，你可以决定在调度这些 Pod 时尽量使它们在使用硬件虚拟化的容器运行时中运行。这样，你将从这些不同运行时所提供的额外隔离中获益，代价是一些额外的开销。

你还可以使用 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，请参阅 [特性门控](#)。RuntimeClass 特性开关必须在 API 服务器和 kubelet 端同时开启。

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](#)）来了解如何配置。

说明：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 [调度](#)。

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/v1 # RuntimeClass 定义于 node.k8s.io API 组
kind: RuntimeClass
metadata:
  name: myclass # 用来引用 RuntimeClass 的名字
  # RuntimeClass 是一个集群层面的资源
handler: myconfiguration # 对应的 CRI 配置的名称
```

说明：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](#)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。你可以查看相应的[事件](#)，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 配置

关于如何安装 CRI 运行时，请查阅 [CRI 安装](#)。

dockershim

Kubernetes 内置的 dockershim CRI 不支持配置运行时 handler。

containerd

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

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](#) 下面：

```
[crio.runtime.runtimes.${HANDLER_NAME}]
  runtime_path = "${PATH_TO_BINARY}"
```

更详细信息，请查阅 [CRI-O 配置文档](#)。

调度

FEATURE STATE: Kubernetes v1.16 [beta]

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 准入控制器](#) 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 [将 Pod 分派到节点](#)。

Pod 开销

FEATURE STATE: Kubernetes v1.18 [beta]

你可以指定与运行 Pod 相关的 **开销** 资源。声明开销即允许集群（包括调度器）在决策 Pod 和资源时将其考虑在内。若要使用 Pod 开销特性，你必须确保 PodOverhead 特性门控 处于启用状态（默认为启用状态）。

Pod 开销通过 RuntimeClass 的 overhead 字段定义。通过使用这些字段，你可以指定使用该 RuntimeClass 运行 Pod 时的开销并确保 Kubernetes 将这些开销计算在内。

接下来

- [RuntimeClass 设计](#)
- [RuntimeClass 调度设计](#)
- 阅读关于 [Pod 开销](#) 的概念
- [PodOverhead 特性设计](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 12, 2021 at 12:16 AM PST: [\[zh\] fix some typo \(5a9fc4ebc\)](#)

容器生命周期回调

这个页面描述了 kubelet 管理的容器如何使用容器生命周期回调框架，藉由其管理生命周期中的事件触发，运行指定代码。

概述

类似于许多具有生命周期回调组件的编程语言框架，例如 Angular、Kubernetes 为容器提供了生命周期回调。回调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回调时运行在处理程序中实现的代码。

容器回调

有两个回调暴露给容器：

PostStart

这个回调在容器被创建之后立即被执行。但是，不能保证回调会在容器入口点（ENTRYPOINT）之前执行。没有参数传递给处理程序。

PreStop

在容器因 API 请求或者管理事件（诸如存活态探针失败、资源抢占、资源竞争等）而被终止之前，此回调会被调用。如果容器已经处于终止或者完成状态，则对 preStop 回调的调用将失败。此调用是阻塞的，也是同步调用，因此必须在发出删除容器的信号之前完成。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](#)。

回调处理程序的实现

容器可以通过实现和注册该回调的处理程序来访问该回调。针对容器，有两种类型的回调处理程序可供实现：

- Exec - 在容器的 cgroups 和名称空间中执行特定的命令（例如 pre-stop.sh）。命令所消耗的资源计入容器的资源消耗。
- HTTP - 对容器上的特定端点执行 HTTP 请求。

回调处理程序执行

当调用容器生命周期管理回调时，Kubernetes 管理系统根据回调动作执行其处理程序，exec 和 tcpSocket 在容器内执行，而 httpGet 则由 kubelet 进程执行。

回调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 回调，容器入口点和回调异步触发。但是，如果回调运行或挂起的时间太长，则容器无法达到 running 状态。

PreStop 回调并不会与停止容器的信号处理程序异步执行；回调必须在可以发送信号之前完成执行。如果 PreStop 回调在执行期间停滞不前，Pod 的阶段会变成 Terminating 并且一致处于该状态，直到其 terminationGracePeriodSeconds 耗尽为止，这时 Pod 会被杀死。这一宽限期是针对 PreStop 回调的执行时间及容器正常停止时间的总和而言的。例如，如果 terminationGracePeriodSeconds 是 60，回调函数花了 55 秒钟完成执行，而容器在收到信号之后花了 10 秒钟来正常结束，那么容器会在其能够正常结束之

前即被杀死，因为 terminationGracePeriodSeconds 的值 小于后面两件事情所花费的总时间 (55 + 10)。

如果 PostStart 或 PreStop 回调失败，它会杀死容器。

用户应该使他们的回调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。

回调递送保证

回调的递送应该是 至少一次，这意味着对于任何给定的事件，例如 PostStart 或 PreStop，回调可以被调用多次。 如何正确处理被多次调用的情况，是回调实现所要考虑的问题。

通常情况下，只会进行单次递送。 例如，如果 HTTP 回调接收器宕机，无法接收流量，则不会尝试重新发送。 然而，偶尔也会发生重复递送的可能。 例如，如果 kubelet 在发送回调的过程中重新启动，回调可能会在 kubelet 恢复后重新发送。

调试回调处理程序

回调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
```

接下来

- 进一步了解[容器环境](#)
- 动手实践，[为容器生命周期事件添加处理程序](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 12, 2020 at 9:03 PM PST: [\[zh\] Sync changes from English site \(3\) \(95ab5ac19\)](#)

工作负载

理解 Pods，Kubernetes 中可部署的最小计算对象，以及辅助它运行它们的高层抽象对象。

工作负载是在 Kubernetes 上运行的应用程序。

无论你的负载是单一组件还是由多个一同工作的组件构成，在 Kubernetes 中你可以在一组 [Pods](#) 中运行它。在 Kubernetes 中，Pod 代表的是集群上处于运行状态的一组 [容器](#)。

Kubernetes Pods 有[确定的生命周期](#)。例如，一旦某 Pod 在你的集群中运行，Pod 运行所在的 [节点](#) 出现致命错误时，所有该节点上的 Pods 都会失败。Kubernetes 将这类失败视为最终状态：即使该节点后来恢复正常运行，你也需要创建新的 Pod 来恢复应用。

不过，为了让用户的日子略微好过一些，你并不需要直接管理每个 Pod。相反，你可以使用 [负载资源](#) 来替你管理一组 Pods。这些资源配置 [控制器](#) 来确保合适类型的、处于运行状态的 Pod 个数是正确的，与你所指定的状态相一致。

Kubernetes 提供若干种内置的工作负载资源：

- [Deployment](#) 和 [ReplicaSet](#)（替换原来的资源 [ReplicationController](#)）。 Deployment 很适合用来管理你的集群上的无状态应用，Deployment 中的所有 Pod 都是相互等价的，并且在需要的时候被换掉。
- [StatefulSet](#) 让你能够运行一个或者多个以某种方式跟踪应用状态的 Pods。例如，如果你的负载会将数据作持久存储，你可以运行一个 StatefulSet，将每个 Pod 与某个 [PersistentVolume](#) 对应起来。你在 StatefulSet 中各个 Pod 内运行的代码可以将数据复制到同一 StatefulSet 中的其它 Pod 中以提高整体的服务可靠性。
- [DaemonSet](#) 定义提供节点本地支撑设施的 Pods。这些 Pods 可能对于你的集群的运维是非常重要的，例如作为网络链接的辅助工具或者作为网络 [插件](#) 的一部分等等。每次你向集群中添加一个新节点时，如果该节点与某 DaemonSet 的规约匹配，则控制面会为该 DaemonSet 调度一个 Pod 到该新节点上运行。
- [Job](#) 和 [CronJob](#)。定义一些一直运行到结束并停止的任务。Job 用来表达的是一次性的任务，而 CronJob 会根据其时间规划反复运行。

在庞大的 Kubernetes 生态系统中，你还可以找到一些提供额外操作的第三方 工作负载资源。通过使用 [定制资源定义 \(CRD\)](#)，你可以添加第三方工作负载资源，以完成原本不是 Kubernetes 核心功能的工作。例如，如果你希望运行一组 Pods，但要求所有 Pods 都可用时才执行操作（比如针对某种高吞吐量的分布式任务），你可以实现一个能够满足这一需求的扩展，并将其安装到集群中运行。

接下来

除了阅读了解每类资源外，你还可以了解与这些资源相关的任务：

- [使用 Deployment 运行一个无状态的应用](#)
- [以单实例 或者多副本集合 的形式运行有状态的应用](#)；
- [使用 CronJob 运行自动化的任务](#)

要了解 Kubernetes 将代码与配置分离的实现机制，可参阅 [配置部分](#)。

关于 Kubernetes 如何为应用管理 Pods，还有两个支撑概念能够提供相关背景信息：

- [垃圾收集](#)机制负责在 对象的 属主资源 被删除时在集群中清理这些对象。
- [Time-to-Live 控制器](#)会在 Job 结束之后的指定时间间隔之后删除它们。

一旦你的应用处于运行状态，你就可能想要以 [Service](#) 的形式使之可在互联网上访问；或者对于 Web 应用而言，使用 [Ingress](#) 资源将其暴露到互联网上。

[Pods](#)

[工作负载资源](#)

Pods

Pod 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。

Pod (就像在鲸鱼荚或者豌豆荚中)是一组(一个或多个) [容器](#)；这些容器共享存储、网络、以及怎样运行这些容器的声明。Pod 中的内容总是并置 (colocated) 的并且一同调度，在共享的上下文中运行。Pod 所建模的是特定于应用的“逻辑主机”，其中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的。在非云环境中，在相同的物理机或虚拟机上运行的应用类似于在同一逻辑主机上运行的云应用。

除了应用容器，Pod 还可以包含在 Pod 启动期间运行的 [Init 容器](#)。你也可以在集群中支持[临时性容器](#)的情况下，为调试的目的注入临时性容器。

什么是 Pod？

说明：除了 Docker 之外，Kubernetes 支持很多其他[容器运行时](#)，[Docker](#) 是最有名的运行时，使用 Docker 的术语来描述 Pod 会很有帮助。

Pod 的共享上下文包括一组 Linux 名字空间、控制组 (cgroup) 和可能一些其他的隔离方面，即用来隔离 Docker 容器的技术。在 Pod 的上下文中，每个独立的应用可能会进一步实施隔离。

就 Docker 概念的术语而言，Pod 类似于共享名字空间和文件系统卷的一组 Docker 容器。

使用 Pod

通常你不需要直接创建 Pod，甚至单实例 Pod。相反，你会使用诸如 [Deployment](#) 或 [Job](#) 这类工作负载资源来创建 Pod。如果 Pod 需要跟踪状态，可以考虑 [StatefulSet](#) 资源。

Kubernetes 集群中的 Pod 主要有两种用法：

- **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”(sidecar)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。

说明：将多个并置、同管的容器组织到一个 Pod 中是一种相对高级的使用场景。只有在一些场景中，容器之间紧密关联时你才应该使用这种模式。

每个 Pod 都旨在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，运行多个实例以提供更多的资源)，则应该使用多个 Pod，每个实例使用一个 Pod。在

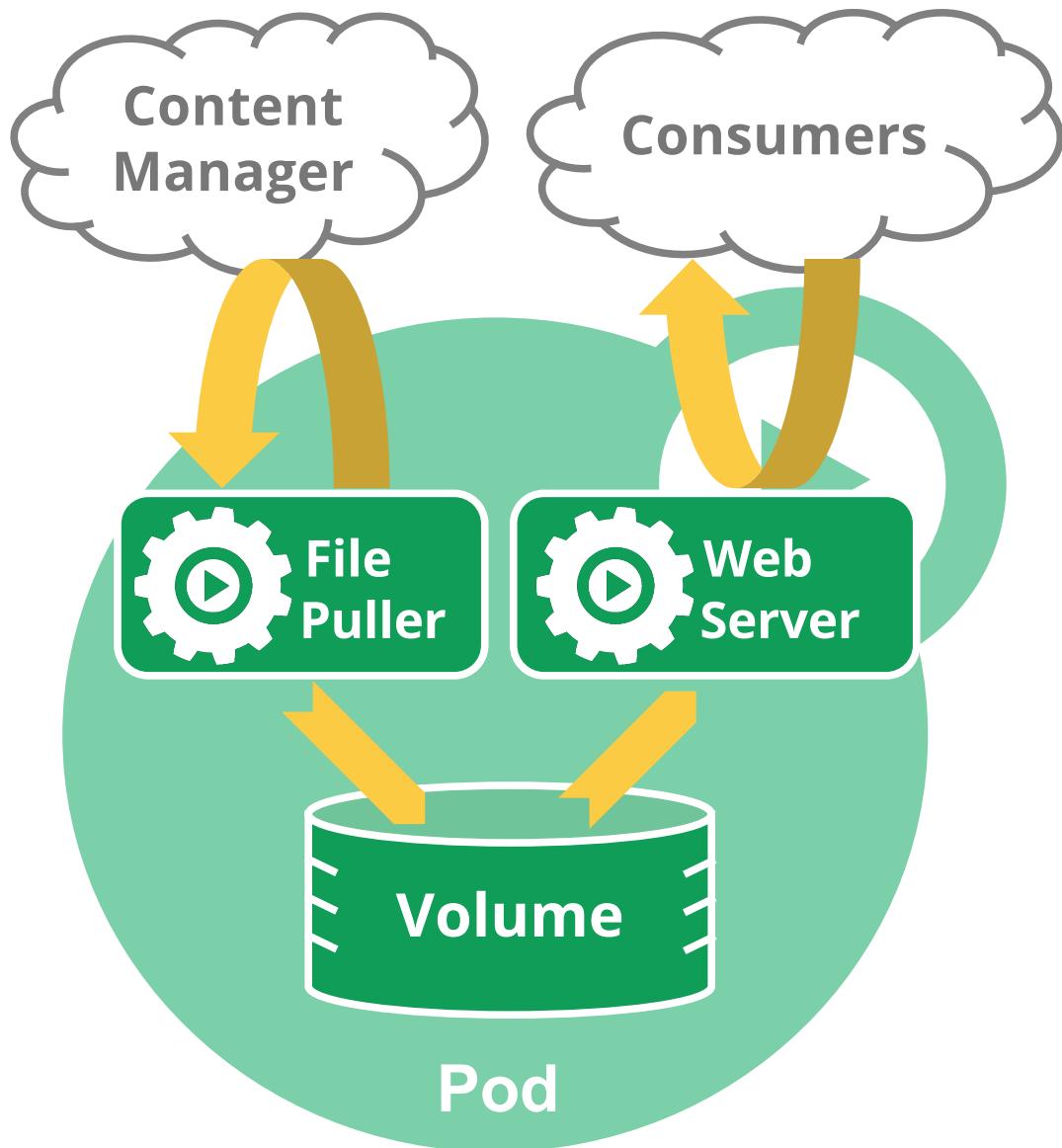
Kubernetes 中，这通常被称为 **副本** (*Replication*)。通常使用一种工作负载资源及其[控制器](#)来创建和管理一组 Pod 副本。

参见[Pod 和控制器](#)以了解 Kubernetes 如何使用工作负载资源及其控制器以实现应用的扩缩和自动修复。

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（形式为容器）。Pod 中的容器被自动安排到集群中的同一物理机或虚拟机上，并可以一起进行调度。容器之间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自身。

例如，你可能有一个容器，为共享卷中的文件提供 Web 服务器支持，以及一个单独的“sidecar（挂斗）”容器负责从远端更新这些文件，如下图所示：



有些 Pod 具有 [Init 容器](#) 和 [应用容器](#)。Init 容器会在启动应用容器之前运行并完成。

Pod 天生地为其成员容器提供了两种共享资源：[网络](#)和[存储](#)。

使用 Pod

你很少在 Kubernetes 中直接创建一个个的 Pod，甚至是单实例（ Singleton ）的 Pod。这是因为 Pod 被设计成了相对临时性的、用后即抛的一次性实体。当 Pod 由你或者间接地由 [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 Pod 结束执行、Pod 对象被删除、Pod 因资源不足而被 驱逐 或者节点失效为止。

说明：重启 Pod 中的容器不应与重启 Pod 混淆。Pod 不是进程，而是容器运行的环境。在被删除之前，Pod 会一直存在。

当你为 Pod 对象创建清单时，要确保所指定的 Pod 名称是合法的 [DNS 子域名](#)。

Pod 和控制器

你可以使用工作负载资源来创建和管理多个 Pod。资源的控制器能够处理副本的管理、上线，并在 Pod 失效时提供自愈能力。例如，如果一个节点失败，控制器注意到该节点上的 Pod 已经停止工作，就可以创建替换性的 Pod。调度器会将替身 Pod 调度到一个健康的节点执行。

下面是一些管理一个或者多个 Pod 的工作负载资源的示例：

- [Deployment](#)
- [StatefulSet](#)
- [DaemonSet](#)

Pod 模版

[负载](#)资源的控制器通常使用 *Pod 模板* (*Pod Template*) 来替你创建 Pod 并管理它们。

Pod 模板是包含在工作负载对象中的规范，用来创建 Pod。这类负载资源包括 [Deployment](#)、[Job](#) 和 [DaemonSets](#) 等。

工作负载的控制器会使用负载对象中的 PodTemplate 来生成实际的 Pod。PodTemplate 是你用来运行应用时指定的负载资源的目标状态的一部分。

下面的示例是一个简单的 Job 的清单，其中的 template 指示启动一个容器。该 Pod 中的容器会打印一条消息之后暂停。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # 这里是 Pod 模版
    spec:
      containers:
        - name: hello
```

```
image: busybox
command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600']
restartPolicy: OnFailure
# 以上为 Pod 模版
```

修改 Pod 模版或者切换到新的 Pod 模版都不会对已经存在的 Pod 起作用。Pod 不会直接收到模版的更新。相反，新的 Pod 会被创建出来，与更改后的 Pod 模版匹配。

例如，Deployment 控制器针对每个 Deployment 对象确保运行中的 Pod 与当前的 Pod 模版匹配。如果模版被更新，则 Deployment 必须删除现有的 Pod，基于更新后的模版 创建新的 Pod。每个工作负载资源都实现了自己的规则，用来处理对 Pod 模版的更新。

在节点上，[kubelet](#)并不直接监测或管理与 Pod 模版相关的细节或模版的更新，这些细节都被抽象出来。这种抽象和关注点分离简化了整个系统的语义，并且使得用户可以在不改变现有代码的前提下就能扩展集群的行为。

Pod 更新与替换

正如前面章节所述，当某工作负载的 Pod 模板被改变时，控制器会基于更新的模板 创建新的 Pod 对象而不是对现有 Pod 执行更新或者修补操作。

Kubernetes 并不禁止你直接管理 Pod。对运行中的 Pod 的某些字段执行就地更新操作还是可能的。不过，类似 [patch](#) 和 [replace](#) 这类更新操作有一些限制：

- Pod 的绝大多数元数据都是不可变的。例如，你不可以改变其 namespace、name、uid 或者 creationTimestamp 字段；generation 字段是比较特别的，如果更新该字段，只能增加字段取值而不能减少。
- 如果 metadata.deletionTimestamp 已经被设置，则不可以向 metadata.finalizers 列表中添加新的条目。
- Pod 更新不可以改变除 spec.containers[*].image、spec.initContainers[*].image、spec.activeDeadlineSeconds 或 spec.tolerations 之外的字段。对于 spec.tolerations，你只被允许添加新的条目到其中。
- 在更新spec.activeDeadlineSeconds 字段时，以下两种更新操作是被允许的：
 1. 如果该字段尚未设置，可以将其设置为一个正数；
 2. 如果该字段已经设置为一个正数，可以将其设置为一个更小的、非负的整数。

资源共享和通信

Pod 使它的成员容器间能够进行数据共享和通信。

Pod 中的存储

一个 Pod 可以设置一组共享的存储卷。Pod 中的所有容器都可以访问该共享卷，从而允许这些容器共享数据。卷还允许 Pod 中的持久数据保留下，即使其中的容器需要重新

启动。有关 Kubernetes 如何在 Pod 中实现共享存储并将其提供给 Pod 的更多信息，请参考[卷](#)。

Pod 联网

每个 Pod 都在每个地址族中获得一个唯一的 IP 地址。Pod 中的每个容器共享网络名字空间，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）。

在同一个 Pod 内，所有容器共享一个 IP 地址和端口空间，并且可以通过 localhost 发现对方。他们也能通过如 SystemV 信号量或 POSIX 共享内存这类标准的进程间通信方式互相通信。不同 Pod 中的容器的 IP 地址互不相同，没有[特殊配置](#)就不能使用 IPC 进行通信。如果某容器希望与运行于其他 Pod 中的容器通信，可以通过 IP 联网的方式实现。

Pod 中的容器所看到的系统主机名与为 Pod 配置的 name 属性值相同。[网络](#)部分提供了更多有关此内容的信息。

容器的特权模式

Pod 中的任何容器都可以使用容器规约中的[安全性上下文](#)中的 privileged 参数启用特权模式。这对于想要使用使用操作系统管理权能（Capabilities，如操纵网络堆栈和访问设备）的容器很有用。容器内的进程几乎可以获得与容器外的进程相同的特权。

说明：你的[容器运行时](#)必须支持 特权容器的概念才能使用这一配置。

静态 Pod

静态 Pod (*Static Pod*) 直接由特定节点上的 kubelet 守护进程管理，不需要[API 服务器](#)看到它们。尽管大多数 Pod 都是通过控制面（例如，[Deployment](#)）来管理的，对于静态 Pod 而言，kubelet 直接监控每个 Pod，并在其失效时重启之。

静态 Pod 通常绑定到某个节点上的 [kubelet](#)。其主要用途是运行自托管的控制面。在自托管场景中，使用 kubelet 来管理各个独立的[控制面组件](#)。

kubelet 自动尝试为每个静态 Pod 在 Kubernetes API 服务器上创建一个[镜像 Pod](#)。这意味着在节点上运行的 Pod 在 API 服务器上是可见的，但不可以通过 API 服务器来控制。

接下来

要了解为什么 Kubernetes 会在其他资源（如[StatefulSet](#) 或 [Deployment](#)）封装通用的 Pod API，相关的背景信息可以在前人的研究中找到。其中包括：

- [Aurora](#)
- [Borg](#)
- [Marathon](#)

- [Omega](#)
- [Tupperware](#).

Pod 的生命周期

本页面讲述 Pod 的生命周期。Pod 遵循一个预定义的生命周期，起始于 Pending 阶段，如果至少其中一个主要容器正常启动，则进入 Running，之后取决于 Pod 中是否有容器以失败状态结束而进入 Succeeded 或者 Failed 阶段。

在 Pod 运行期间，kubelet 能够重启容器以处理一些失效场景。在 Pod 内部，Kubernetes 跟踪不同容器的[状态](#)并确定使 Pod 重新变得健康所需要采取的动作。

在 Kubernetes API 中，Pod 包含规约部分和实际状态部分。Pod 对象的状态包含了一组 [Pod 状况 \(Conditions\)](#)。如果应用需要的话，你也可以向其中注入[自定义的就绪性信息](#)。

Pod 在其生命周期中只会被调度一次。一旦 Pod 被调度（分派）到某个节点，Pod 会一直在该节点运行，直到 Pod 停止或者被[终止](#)。

Pod 生命期

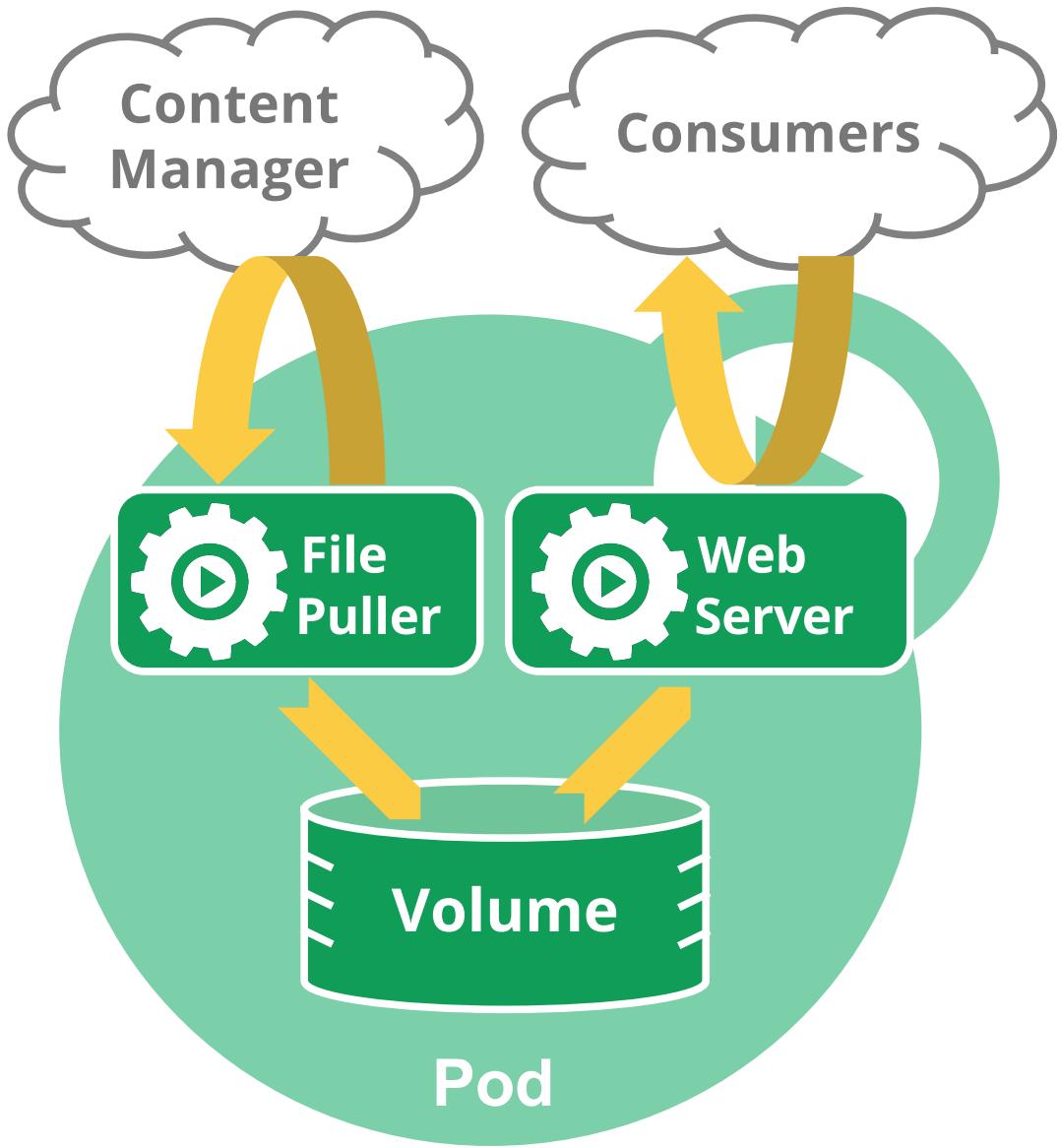
和一个个独立的应用容器一样，Pod 也被认为是相对临时性（而不是长期存在）的实体。Pod 会被创建、赋予一个唯一的 ID ([UID](#))，并被调度到节点，并在终止（根据重启策略）或删除之前一直运行在该节点。

如果一个[节点](#)死掉了，调度到该节点的 Pod 也被计划在给定超时期限结束后[删除](#)。

Pod 自身不具有自愈能力。如果 Pod 被调度到某[节点](#)而该节点之后失效，或者调度操作本身失效，Pod 会被删除；与此类似，Pod 无法在节点资源耗尽或者节点维护期间继续存活。Kubernetes 使用一种高级抽象，称作 [控制器](#)，来管理这些相对而言可随时丢弃的 Pod 实例。

任何给定的 Pod（由 UID 定义）从不会被“重新调度（rescheduled）”到不同的节点；相反，这一 Pod 可以被一个新的、几乎完全相同的 Pod 替换掉。如果需要，新 Pod 的名字可以不变，但是其 UID 会不同。

如果某物声称其生命期与某 Pod 相同，例如存储[卷](#)，这就意味着该对象在此 Pod（UID 亦相同）存在期间也一直存在。如果 Pod 因为任何原因被删除，甚至某完全相同的替代 Pod 被创建时，这个相关的对象（例如这里的卷）也会被删除并重建。



Pod 结构图例

一个包含多个容器的 Pod 中包含一个用来拉取文件的程序和一个 Web 服务器，均使用持久卷作为容器间共享的存储。

Pod 阶段

Pod 的 status 字段是一个 [PodStatus](#) 对象，其中包含一个 phase 字段。

Pod 的阶段 (Phase) 是 Pod 在其生命周期中所处位置的简单宏观概述。该阶段并不是对容器或 Pod 状态的综合汇总，也不是为了成为完整的状态机。

Pod 阶段的数量和含义是严格定义的。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：

取值	描述
Pending (悬决)	Pod 已被 Kubernetes 系统接受 , 但有一个或者多个容器尚未创建亦未运行。此阶段包括等待 Pod 被调度的时间和通过网络下载镜像的时间 ,
Running (运行中)	Pod 已经绑定到了某个节点 , Pod 中所有的容器都已被创建。至少有一个容器仍在运行 , 或者正处于启动或重启状态。
Succeeded (成功)	Pod 中的所有容器都已成功终止 , 并且不会再重启。
Failed (失败)	Pod 中的所有容器都已终止 , 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。
Unknown (未知)	因为某些原因无法取得 Pod 的状态。这种情况通常是因为与 Pod 所在主机通信失败。

如果某节点死掉或者与集群中其他节点失联 , Kubernetes 会实施一种策略 , 将失去的节点上运行的所有 Pod 的 phase 设置为 Failed。

容器状态

Kubernetes 会跟踪 Pod 中每个容器的状态 , 就像它跟踪 Pod 总体上的[阶段](#)一样。你可以使用[容器生命周期回调](#) 来在容器生命周期中的特定时间点触发事件。

一旦[调度器](#)将 Pod 分派给某个节点 , kubelet 就通过 [容器运行时](#) 开始为 Pod 创建容器。容器的状态有三种 : Waiting (等待) 、 Running (运行中) 和 Terminated (已终止) 。

要检查 Pod 中容器的状态 , 你可以使用 kubectl describe pod <pod 名称>。 其输出中包含 Pod 中每个容器的状态。

每种状态都有特定的含义 :

Waiting (等待)

如果容器并不处在 Running 或 Terminated 状态之一 , 它就处在 Waiting 状态。处于 Waiting 状态的容器仍在运行它完成启动所需要的操作 : 例如 , 从某个容器镜像仓库拉取容器镜像 , 或者向容器应用 [Secret](#) 数据等等。 当你使用 kubectl 来查询包含 Waiting 状态的容器的 Pod 时 , 你也会看到一个 Reason 字段 , 其中给出了容器处于等待状态的原因。

Running (运行中)

Running 状态表明容器正在执行状态并且没有问题发生。 如果配置了 postStart 回调 , 那么该回调已经执行且已完成。 如果你使用 kubectl 来查询包含 Running 状态的容器的 Pod 时 , 你也会看到关于容器进入 Running 状态的信息。

Terminated (已终止)

处于 Terminated 状态的容器已经开始执行并且或者正常结束或者因为某些原因失败。如果你使用 kubectl 来查询包含 Terminated 状态的容器的 Pod 时，你会看到容器进入此状态的原因、退出代码以及容器执行期间的起止时间。

如果容器配置了 preStop 回调，则该回调会在容器进入 Terminated 状态之前执行。

容器重启策略

Pod 的 spec 中包含一个 restartPolicy 字段，其可能取值包括 Always、OnFailure 和 Never。默认值是 Always。

restartPolicy 适用于 Pod 中的所有容器。restartPolicy 仅针对同一节点上 kubelet 的容器重启动作。当 Pod 中的容器退出时，kubelet 会按指数回退方式计算重启的延迟（10s、20s、40s、...），其最长延迟为 5 分钟。一旦某容器执行了 10 分钟并且没有出现问题，kubelet 对该容器的重启动回退计时器执行 重置操作。

Pod 状况

Pod 有一个 PodStatus 对象，其中包含一个 [PodConditions](#) 数组。Pod 可能通过也可能未通过其中的一些状况测试。

- PodScheduled : Pod 已经被调度到某节点；
- ContainersReady : Pod 中所有容器都已就绪；
- Initialized : 所有的 [Init 容器](#) 都已成功启动；
- Ready : Pod 可以为请求提供服务，并且应该被添加到对应服务的负载均衡池中。

字段名称	描述
type	Pod 状况的名称
status	表明该状况是否适用，可能的取值有 "True", "False" 或 "Unknown"
lastProbeTime	上次探测 Pod 状况时的时间戳
lastTransitionTime	Pod 上次从一种状态转换到另一种状态时的时间戳
reason	机器可读的、驼峰编码 (UpperCamelCase) 的文字，表述上次状况变化的原因
message	人类可读的消息，给出上次状态转换的详细信息

Pod 就绪态

FEATURE STATE: Kubernetes v1.14 [stable]

你的应用可以向 PodStatus 中注入额外的反馈或者信号：Pod Readiness (Pod 就绪态)。要使用这一特性，可以设置 Pod 规约中的 readinessGates 列表，为 kubelet 提供一组额外的状况供其评估 Pod 就绪态时使用。

就绪态门控基于 Pod 的 status.conditions 字段的当前值来做决定。如果 Kubernetes 无法在 status.conditions 字段中找到某状况，则该状况的状态值默认为 "False"。

这里是一个例子：

```
kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready          # 内置的 Pod 状况
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
    - type: "www.example.com/feature-1"      # 额外的 Pod 状况
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2018-01-01T00:00:00Z
  containerStatuses:
    - containerID: docker://abcd...
      ready: true
...

```

你所添加的 Pod 状况名称必须满足 Kubernetes [标签键名格式](#)。

Pod 就绪态的状态

命令 kubectl patch 不支持修改对象的状态。如果需要设置 Pod 的 status.conditions，应用或者 [Operators](#) 需要使用 PATCH 操作。你可以使用 [Kubernetes 客户端库](#) 之一来编写代码，针对 Pod 就绪态设置定制的 Pod 状况。

对于使用定制状况的 Pod 而言，只有当下面的陈述都适用时，该 Pod 才会被评估为就绪：

- Pod 中所有容器都已就绪；
- readinessGates 中的所有状况都为 True 值。

当 Pod 的容器都已就绪，但至少一个定制状况没有取值或者取值为 False，kubelet 将 Pod 的状况设置为 ContainersReady。

容器探针

[Probe](#) 是由 [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](#)（处理器）。有三种类型的处理程序：

- [ExecAction](#)：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](#)：对容器的 IP 地址上的指定端口执行 TCP 检查。如果端口打开，则诊断被认为是成功的。
- [HTTPGetAction](#)：对容器的 IP 地址上指定端口和路径执行 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：

- Success（成功）：容器通过了诊断。
- Failure（失败）：容器未通过诊断。
- Unknown（未知）：诊断失败，因此不会采取任何行动。

针对运行中的容器，kubelet 可以选择是否执行以下三种探针，以及如何针对探测结果作出反应：

- livenessProbe：指示容器是否正在运行。如果存活态探测失败，则 kubelet 会杀死容器，并且容器将根据其[重启策略](#)决定未来。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为请求提供服务。如果就绪态探测失败，端点控制器将从与 Pod 匹配的所有服务的端点列表中删除该 Pod 的 IP 地址。初始延迟之前的就绪态的状态值默认为 Failure。如果容器不提供就绪态探针，则默认状态为 Success。
- startupProbe：指示容器中的应用是否已经启动。如果提供了启动探针，则所有其他探针都会被禁用，直到此探针成功为止。如果启动探测失败，kubelet 将杀死容器，而容器依其[重启策略](#)进行重启。如果容器没有提供启动探测，则默认状态为 Success。

如欲了解如何设置存活态、就绪态和启动探针的进一步细节，可以参阅[配置存活态、就绪态和启动探针](#)。

何时该使用存活态探针？

FEATURE STATE: Kubernetes v1.0 [stable]

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，则不一定需要存活态探针；kubelet 将根据 Pod 的 restartPolicy 自动执行修复操作。

如果你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态探针，并指定 restartPolicy 为 "Always" 或 "OnFailure"。

何时该使用就绪态探针？

FEATURE STATE: Kubernetes v1.0 [stable]

如果要仅在探测成功时才开始向 Pod 发送请求流量，请指定就绪态探针。在这种情况下，就绪态探针可能与存活态探针相同，但是规约中的就绪态探针的存在意味着 Pod 将在启动阶段不接收任何数据，并且只有在探针探测成功后才开始接收数据。

如果你的容器需要加载大规模的数据、配置文件或者在启动期间执行迁移操作，可以添加一个就绪态探针。

如果你希望容器能够自行进入维护状态，也可以指定一个就绪态探针，检查某个特定于就绪态的因此不同于存活态探测的端点。

说明：请注意，如果你只是想在 Pod 被删除时能够排空请求，则不一定需要使用就绪态探针；在删除 Pod 时，Pod 会自动将自身置于未就绪状态，无论就绪态探针是否存在。等待 Pod 中的容器停止期间，Pod 会一直处于未就绪状态。

何时该使用启动探针？

FEATURE STATE: Kubernetes v1.18 [beta]

对于所包含的容器需要较长时间才能启动就绪的 Pod 而言，启动探针是有用的。你不再需要配置一个较长的存活态探测时间间隔，只需要设置另一个独立的配置选定，对启动期间的容器执行探测，从而允许使用远远超出存活态时间间隔所允许的时长。

如果你的容器启动时间通常超出 `initialDelaySeconds + failureThreshold × periodSeconds` 总值，你应该设置一个启动探测，对存活态探针所使用的同一端点执行检查。`periodSeconds` 的默认值是 10 秒。你应该将其 `failureThreshold` 设置得足够高，以便容器有充足的时间完成启动，并且避免更改存活态探针所使用的默认值。这一设置有助于减少死锁状况的发生。

Pod 的终止

由于 Pod 所代表的是在集群中节点上运行的进程，当不再需要这些进程时允许其体面地终止是很重要的。一般不应武断地使用 KILL 信号终止它们，导致这些进程没有机会完成清理操作。

设计的目标是令你能够请求删除进程，并且知道进程何时被终止，同时也能够确保删除操作终将完成。当你请求删除某个 Pod 时，集群会记录并跟踪 Pod 的体面终止周期，而不是直接强制地杀死 Pod。在存在强制关闭设施的前提下，[kubelet](#) 会尝试体面地终止 Pod。

通常情况下，容器运行时会发送一个 TERM 信号到每个容器中的主进程。很多容器运行时都能够注意到容器镜像中 `STOPSIGAL` 的值，并发送该信号而不是 TERM。一旦超出了体面终止限期，容器运行时会向所有剩余进程发送 KILL 信号，之后 Pod 就会被从[API 服务器](#) 上移除。如果 kubelet 或者容器运行时的管理服务在等待进程终止期间被重启，集群会从头开始重试，赋予 Pod 完整的体面终止限期。

下面是一个例子：

1. 你使用 kubectl 工具手动删除某个特定的 Pod，而该 Pod 的体面终止限期是默认值（30 秒）。
2. API 服务器中的 Pod 对象被更新，记录涵盖体面终止限期在内 Pod 的最终死期，超出所计算时间点则认为 Pod 已死（dead）。如果你使用 kubectl describe 来查验你正在删除的 Pod，该 Pod 会显示为 "Terminating"（正在终止）。在 Pod 运行所在的节点上：kubelet 一旦看到 Pod 被标记为正在终止（已经设置了体面终止限期），kubelet 即开始本地的 Pod 关闭过程。

1. 如果 Pod 中的容器之一定义了 preStop 回调，kubelet 开始在容器内运行该回调逻辑。如果超出体面终止限期时，preStop 回调逻辑仍在运行，kubelet 会请求给予该 Pod 的宽限期一次性增加 2 秒钟。

说明：如果 preStop 回调所需要的时间长于默认的体面终止限期，你必须修改 terminationGracePeriodSeconds 属性值来使其正常工作。

2. kubelet 接下来触发容器运行时发送 TERM 信号给每个容器中的进程 1。

说明：Pod 中的容器会在不同时刻收到 TERM 信号，接收顺序也是不确定的。如果关闭的顺序很重要，可以考虑使用 preStop 回调逻辑来协调。

1. 与此同时，kubelet 启动体面关闭逻辑，控制面会将 Pod 从对应的端点列表（以及端点切片列表，如果启用了的话）中移除，过滤条件是 Pod 被对应的 服务 以某 选择算符 选定。ReplicaSets 和其他工作负载资源 不再将关闭进程中的 Pod 视为合法的、能够提供服务的副本。关闭动作很慢的 Pod 也无法继续处理请求数据，因为负载均衡器（例如服务代理）已经在终止宽限期开始的时候 将其从端点列表中移除。

1. 超出终止宽限期限时，kubelet 会触发强制关闭过程。容器运行时会向 Pod 中所有容器内 仍在运行的进程发送 SIGKILL 信号。kubelet 也会清理隐藏的 pause 容器，如果容器运行时使用了这种容器的话。
2. kubelet 触发强制从 API 服务器上删除 Pod 对象的逻辑，并将体面终止限期设置为 0（这意味着马上删除）。
3. API 服务器删除 Pod 的 API 对象，从任何客户端都无法再看到该对象。

强制终止 Pod

注意：对于某些工作负载及其 Pod 而言，强制删除很可能带来某种破坏。

默认情况下，所有的删除操作都会附有 30 秒钟的宽限期。kubectl delete 命令支持 --grace-period=<seconds> 选项，允许你重载默认值，设定自己希望的期限值。

将宽限期强制设置为 0 意味着立即从 API 服务器删除 Pod。如果 Pod 仍然运行于某节点上，强制删除操作会触发 kubelet 立即执行清理操作。

说明：你必须在设置 `--grace-period=0` 的同时额外设置 `--force` 参数才能发起强制删除请求。

执行强制删除操作时，API 服务器不再等待来自 kubelet 的、关于 Pod 已经在原来运行的节点上终止执行的确认消息。API 服务器直接删除 Pod 对象，这样新的与之同名的 Pod 即可以被创建。在节点侧，被设置为立即终止的 Pod 仍然会在被强行杀死之前获得一点点的宽限时间。

如果你需要强制删除 StatefulSet 的 Pod，请参阅 [从 StatefulSet 中删除 Pod](#) 的任务文档。

失效 Pod 的垃圾收集

对于已失败的 Pod 而言，对应的 API 对象仍然会保留在集群的 API 服务器上，直到用户或者[控制器](#)进程显式地将其删除。

控制面组件会在 Pod 个数超出所配置的阈值（根据 `kube-controller-manager` 的 `terminated-pod-gc-threshold` 设置）时删除已终止的 Pod（阶段值为 `Succeeded` 或 `Failed`）。这一行为会避免随着时间演进不断创建和终止 Pod 而引起的资源泄露问题。

接下来

- 动手实践[为容器生命周期时间关联处理程序](#)。
- 动手实践[配置存活态、就绪态和启动探针](#)。
- 进一步了解[容器生命周期回调](#)。
- 关于 API 中定义的有关 Pod/容器的详细规范信息，可参阅 [PodStatus](#) 和 [ContainerStatus](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 January 13, 2021 at 1:08 AM PST: [\[zh\] update pod-lifecycle.md \(ce6a73de1\)](#)

Init 容器

本页提供了 Init 容器的概览。Init 容器是一种特殊容器，在 [Pod](#) 内的应用容器启动之前运行。Init 容器可以包括一些应用镜像中不存在的实用工具和安装脚本。

你可以在 Pod 的规约中与用来描述应用容器的 `containers` 数组平行的位置指定 Init 容器。

理解 Init 容器

每个 [Pod](#) 中可以包含多个容器，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，kubelet 会不断地重启该 Init 容器直到该容器成功为止。然而，如果 Pod 对应的 restartPolicy 值为 "Never"，Kubernetes 不会重新启动 Pod。

为 Pod 设置 Init 容器需要在 Pod 的 spec 中添加 initContainers 字段，该字段以 [Container](#) 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](#)节有说明。

同时 Init 容器不支持 lifecycle、livenessProbe、readinessProbe 和 startupProbe，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。

使用 Init 容器

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 容器可以访问应用容器不能访问的 [Secret](#) 的权限。
- 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=$()&ip=$()'
```

- 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[卷](#)中。

- 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 容器的情况

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 节中的应用容器。

```
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.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox:1.28
```

```
command: ['sh', '-c', "until nslookup mydb.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for mydb; sleep 2; done"]
```

你通过运行下面的命令启动 Pod :

```
kubectl apply -f myapp.yaml
```

```
pod/myapp-pod created
```

使用下面的命令检查其状态 :

```
kubectl get -f myapp.yaml
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-pod	0/1	Init:0/2	0	6m

或者查看更多详细信息 :

```
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 # 查看第一个 Init 容器  
kubectl logs myapp-pod -c init-mydb # 查看第二个 Init 容器
```

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 服务的命令：

```
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 进入 Running 状态：

```
kubectl get -f myapp.yaml
```

NAME	READY	STATUS	RESTARTS	AGE
myapp-pod	1/1	Running	0	9m

这个简单例子应该能为你创建自己的 Init 容器提供一些启发。 [接下来](#)节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 容器会在网络和数据卷初始化之后按顺序启动。 kubelet 运行依据 Init 容器在 Pod 规约中的出现顺序依次运行之。

每个 Init 容器成功退出后才会启动下一个 Init 容器。 如果某容器因为容器运行时的原因无法启动，或以错误状态退出，kubelet 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，但会将状况 Initializing 设置为 false。

如果 Pod [重启](#)，所有 Init 容器必须重新执行。

对 Init 容器规约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态（Completion）的就绪态（Readiness）。 Kubernetes 会在校验时强制执行此检查。

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

在给定的 Init 容器执行顺序下，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 的请求和限制值。Pod 级别的 cgroups 是基于有效 Pod 的请求和限制值，和调度器相同。

Pod 重启的原因

Pod 重启会导致 Init 容器重新执行，主要有如下几个原因：

- 用户更新 Pod 的规约导致 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 权限访问节点的人员来完成。
- 当 restartPolicy 设置为 "Always"，Pod 中所有容器会终止而强制重启。由于垃圾收集机制的原因，Init 容器的完成记录将会丢失。

接下来

- 阅读[创建包含 Init 容器的 Pod](#)
- 学习如何[调试 Init 容器](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 08, 2021 at 3:54 AM PST: [fix init container status describe fault \(18e7b9076\)](#)

Pod 拓扑分布约束

FEATURE STATE: Kubernetes v1.19 [stable]

你可以使用 *拓扑分布约束 (Topology Spread Constraints)* 来控制 [Pods](#) 在集群内故障域之间的分布，例如区域 (Region)、可用区 (Zone)、节点和其他用户自定义拓扑域。这样做有助于实现高可用并提升资源利用率。

说明：在 v1.19 之前的 Kubernetes 版本中，如果要使用 Pod 拓扑扩展约束，你必须在 [API 服务器](#) 和 [调度器](#) 中启用 EvenPodsSpread [特性门控](#)。

先决条件

节点标签

拓扑分布约束依赖于节点标签来标识每个节点所在的拓扑域。例如，某节点可能具有标签：node=node1,zone=us-east-1a,region=us-east-1

假设你拥有具有以下标签的一个 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：

[JavaScript must be [enabled](#) to view content]

你可以复用在大多数集群上自动创建和填充的 [常用标签](#)，而不是手动添加标签。

Pod 的分布约束

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>
```

你可以定义一个或多个 topologySpreadConstraint 来指示 kube-scheduler 如何根据与现有的 Pod 的关联关系将每个传入的 Pod 部署到集群中。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。取决于 whenUnsatisfiable 的 取值，其语义会有不同。
 - 当 whenUnsatisfiable 等于 "DoNotSchedule" 时，maxSkew 是目标拓扑 域 中匹配的 Pod 数与全局最小值之间可存在的差异。
 - 当 whenUnsatisfiable 等于 "ScheduleAnyway" 时，调度器会更为偏向能 够降低 偏差值的拓扑域。
- **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域中。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 不满足分布约束时如何处理：
 - DoNotSchedule (默认) 告诉调度器不要调度。
 - ScheduleAnyway 告诉调度器仍然继续调度，只是根据如何能将偏差最小化 来对 节点进行排序。
- **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[标签选择算符](#)。

你可以执行 kubectl explain Pod.spec.topologySpreadConstraints 命令以 了解关于 topologySpreadConstraints 的更多信息。

例子：单个 TopologySpreadConstraint

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：

[JavaScript must be [enabled](#) to view content]

如果希望新来的 Pod 均匀分布在现有的可用区域，则可以按如下设置其规约：

[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键值对为 "zone:<any value>" 的节点。 whenUnsatisfiable: DoNotSchedule 告诉调度器如果新的 Pod 不满足约束，则让它保持悬决状态。

如果调度器将新的 Pod 放入 "zoneA"，Pods 分布将变为 [3, 1]，因此实际的偏差为 2 (3 - 1)。这违反了 maxSkew: 1 的约定。此示例中，新 Pod 只能放置在 "zoneB" 上：

[JavaScript must be [enabled](#) to view content]

或者

[JavaScript must be [enabled](#) to view content]

你可以调整 Pod 规约以满足各种要求：

- 将 maxSkew 更改为更大的值，比如 "2"，这样新的 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，以确保新的 Pod 始终可以被调度（假设满足其他的调度 API）。但是，最好将其放置在匹配 Pod 数量较少的拓扑域中。（请注意，这一优先判定会与其他内部调度优先级（如资源使用率等）排序准则一起进行标准化。）

例子：多个 TopologySpreadConstraints

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：

[JavaScript must be [enabled](#) to view content]

可以使用 2 个 TopologySpreadConstraint 来控制 Pod 在 区域和节点两个维度上的分布：

[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，新的 Pod 只能放置在 "zoneB" 中；而在第二个约束中，新的 Pod 只能放置在 "node4" 上。最后两个约束的结果加在一起，唯一可行的选择是放置在 "node4" 上。

多个约束之间可能存在冲突。假设有一个跨越 2 个区域的 3 节点集群：

[JavaScript must be [enabled](#) to view content]

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，你可以增加 maxSkew 或修改其中一个约束，让其使用 whenUnsatifiable: ScheduleAnyway。

约定

这里有一些值得注意的隐式约定：

- 只有与新的 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. 新的 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-type: zoneC} 的 "node5" 加入到集群，它将由于没有标签键 "zone" 而被忽略。
- 注意，如果新 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，如果移除新 Pod 上的标签，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。
- 如果新 Pod 定义了 spec.nodeSelector 或 spec.affinity.nodeAffinity，则不匹配的节点会被忽略。

假设你有一个跨越 zoneA 到 zoneC 的 5 节点集群：

[JavaScript must be [enabled](#) to view content]

[JavaScript must be [enabled](#) to view content]

而且你知道 "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](#)
[View]

```
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
```

集群级别的默认约束

为集群设置默认的拓扑分布约束也是可能的。默认拓扑分布约束在且仅在以下条件满足时才会应用到 Pod 上：

- Pod 没有在其 .spec.topologySpreadConstraints 设置任何约束；
- Pod 隶属于某个服务、副本控制器、ReplicaSet 或 StatefulSet。

你可以在 [调度方案 \(Schedulingg Profile \)](#) 中将默认约束作为 PodTopologySpread 插件参数的一部分来设置。约束的设置采用如前所述的 [API](#)，只是 labelSelector 必须为空。选择算符是根据 Pod 所属的服务、副本控制器、ReplicaSet 或 StatefulSet 来设置的。

配置的示例可能看起来像下面这个样子：

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - pluginConfig:
      - name: PodTopologySpread
args:
  defaultConstraints:
    - maxSkew: 1
      topologyKey: topology.kubernetes.io/zone
```

```
whenUnsatisfiable: ScheduleAnyway  
defaultingType: List
```

说明：

默认调度约束所生成的评分可能与 [SelectorSpread 插件](#) 所生成的评分有冲突。建议你在为 PodTopologySpread 设置默认约束是禁用调度方案中的该插件。

内部默认约束

FEATURE STATE: Kubernetes v1.20 [beta]

当你使用了默认启用的 DefaultPodTopologySpread 特性门控时，原来的 SelectorSpread 插件会被禁用。kube-scheduler 会使用下面的默认拓扑约束作为 PodTopologySpread 插件的配置：

```
defaultConstraints:  
- maxSkew: 3  
  topologyKey: "kubernetes.io/hostname"  
  whenUnsatisfiable: ScheduleAnyway  
- maxSkew: 5  
  topologyKey: "topology.kubernetes.io/zone"  
  whenUnsatisfiable: ScheduleAnyway
```

此外，原来用于提供等同行为的 SelectorSpread 插件也会被禁用。

说明：

如果你的节点不会 **同时** 设置 kubernetes.io/hostname 和 topology.kubernetes.io/zone 标签，你应该定义自己的约束而不是使用 Kubernetes 的默认约束。

插件 PodTopologySpread 不会为未设置分布约束中所给拓扑键的节点评分。

如果你不想为集群使用默认的 Pod 分布约束，你可以通过设置 defaultingType 参数为 List 和将 PodTopologySpread 插件配置中的 defaultConstraints 参数置空来禁用默认 Pod 分布约束。

```
apiVersion: kubescheduler.config.k8s.io/v1beta1  
kind: KubeSchedulerConfiguration
```

```
profiles:  
- pluginConfig:  
  - name: PodTopologySpread  
    args:
```

```
defaultConstraints: []
defaultingType: List
```

与 PodAffinity/PodAntiAffinity 相比较

在 Kubernetes 中，与“亲和性”相关的指令控制 Pod 的调度方式（更密集或更分散）。

- 对于 PodAffinity，你可以尝试将任意数量的 Pod 集中到符合条件的拓扑域中。
- 对于 PodAntiAffinity，只能将一个 Pod 调度到某个拓扑域中。

要实现更细粒度的控制，你可以设置拓扑分布约束来将 Pod 分布到不同的拓扑域下，从而实现高可用性或节省成本。这也有助于工作负载的滚动更新和平稳地扩展副本规模。有关详细信息，请参考 [动机](#) 文档。

已知局限性

- Deployment 缩容操作可能导致 Pod 分布不平衡。
- 具有污点的节点上的 Pods 也会被统计。参考 [Issue 80921](#)。

接下来

- [博客: PodTopologySpread介绍](#) 详细解释了 maxSkew，并给出了一些高级的使用示例。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 January 08, 2021 at 1:22 AM PST: [feature-state version number is outdated and some lines are missing. \(6eeb41f68\)](#)

干扰 (Disruptions)

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

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** (*Involuntary Disruptions*)。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](#)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** (*Voluntary Disruptions*)。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操作包括：

- 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](#)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](#)）。
- 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)

注意：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 在请求中给出[所需资源](#)。
- 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](#) 和[有状态](#) 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](#)）或跨区域（如果使用[多区域集群](#)）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）自动缩放的某些实现可能导致碎片整理和紧缩节点的自愿干扰。集群管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** (*Disruption Budget*)。

干扰预算

FEATURE STATE: Kubernetes v1.5 [beta]

即使你会经常引入自愿性干扰，Kubernetes 也能够支持你运行高度可用的应用。

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，基于票选机制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 [Eviction API](#)），而不是直接删除 Pod 或 Deployment。

例如，kubectl drain 命令可以用来标记某个节点即将停止服务。运行 kubectl drain 命令时，工具会尝试驱逐机器上的所有 Pod。kubectl 所提交的驱逐请求可能会暂时被拒绝，所以该工具会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, StatefulSet 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](#)的发生，但是确实会计入预算。

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 StatefulSet）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方式是在对应的工作负载资源的 spec 中配置的。

当使用驱逐 API 驱逐 Pod 时，Pod 会被体面地[终止](#)，期间会参考 [PodSpec](#) 中的 terminationGracePeriodSeconds 配置值。

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 也同样显示出来。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权限。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。除此之外，此示例也适用于 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性操作

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。
 - 不停机。
 - 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自愿干扰所做工作相比，有大量的重叠

接下来

- 参考[配置 Pod 干扰预算](#)中的方法来保护你的应用。
- 进一步了解[排空节点](#)的信息。
- 了解[更新 Deployment](#)的过程，包括如何在其进程中维持应用的可用性

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 7:38 PM PST: [fix disruptions.md typo](#) ([e4d5725f6](#))

临时容器

FEATURE STATE: Kubernetes v1.16 [alpha]

本页面概述了临时容器：一种特殊的容器，该容器在现有 [Pod](#) 中临时运行，以便完成用户发起的操作，例如故障排查。你会使用临时容器来检查服务，而不是用它来构建应用程序。

警告：临时容器处于早期的 alpha 阶段，不适用于生产环境集群。应该预料到临时容器在某些情况下不起作用，例如在定位容器的命名空间时。根据 [Kubernetes 弃用政策](#)，此 alpha 功能将来可能发生重大变化或被完全删除。

了解临时容器

[Pod](#) 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 [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 节来描述，但许多字段是不兼容和不允许的。

- 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 [EphemeralContainer 参考文档](#)。

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

当由于容器崩溃或容器镜像不包含调试工具而导致 kubectl exec 无用时，临时容器对于交互式故障排查很有用。

尤其是，[distroless 镜像](#) 允许用户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[进程名字空间共享](#) 很有帮助，可以查看其他容器中的进程。

示例

说明：本节中的示例要求启用 EphemeralContainers [特性门控](#)，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出现在 API 中。通常，你可以使用 kubectl 插件进行故障排查，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
```

输出为：

```
...
Ephemeral Containers:
  debugger:
    Container ID: docker://cf81908f149e7e9213d3c3644eda55c72efaff67652a2685c1146f0ce151e80f
    Image:      busybox
    Image ID:   docker-pullable://busybox@sha256:9f1003c480699be56815db0f8146ad2e22fea85129b5b5983d0e0fb52d9ab70
    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，则可以查看该 Pod 所有容器中的进程。例如，运行上述 attach 操作后，在调试器容器中运行 ps 操作：

```
# 在 "debugger" 临时容器内中运行此 shell 命令
ps auxww
```

运行命令后，输出类似于：

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

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

工作负载资源

[Deployments](#)

[ReplicaSet](#)

[StatefulSets](#)

[DaemonSet](#)

[Jobs](#)

[垃圾收集](#)

[已完成资源的 TTL 控制器](#)

[CronJob](#)

[ReplicationController](#)

Deployments

一个 Deployment 为 [Pods](#) 和 [ReplicaSets](#) 提供声明式的更新能力。

你负责描述 Deployment 中的 [目标状态](#)，而 Deployment [控制器 \(Controller \)](#) 以受控速率更改实际状态，使其变为期望状态。你可以定义 Deployment 以创建新的 ReplicaSet，或删除现有 Deployment，并通过新的 Deployment 收养其资源。

说明：不要管理 Deployment 所拥有的 ReplicaSet。如果存在下面未覆盖的使用场景，请考虑在 Kubernetes 仓库中提出 Issue。

用例

以下是 Deployments 的典型用例：

- [创建 Deployment 以将 ReplicaSet 上线](#)。ReplicaSet 在后台创建 Pods。检查 ReplicaSet 的上线状态，查看其是否成功。
- 通过更新 Deployment 的 PodTemplateSpec，[声明 Pod 的新状态](#)。新的 ReplicaSet 会被创建，Deployment 以受控速率将 Pod 从旧 ReplicaSet 迁移到新 ReplicaSet。每个新的 ReplicaSet 都会更新 Deployment 的修订版本。
- 如果 Deployment 的当前状态不稳定，[回滚到较早的 Deployment 版本](#)。每次回滚都会更新 Deployment 的修订版本。
- [扩大 Deployment 规模以承担更多负载](#)。
- [暂停 Deployment 以应用对 PodTemplateSpec 所作的多项修改](#)，然后恢复其执行以启动新的上线版本。
- [使用 Deployment 状态 来判定上线过程是否出现停滞](#)。
- [清理较旧的不再需要的 ReplicaSet](#)。

创建 Deployment

下面是 Deployment 示例。其中创建了一个 ReplicaSet，负责启动三个 nginx Pods：

[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:1.14.2
```

```
ports:  
- containerPort: 80
```

在该例中：

- 创建名为 nginx-deployment (由 .metadata.name 字段标明) 的 Deployment。
- 该 Deployment 创建三个 (由 replicas 字段标明) Pod 副本。
- selector 字段定义 Deployment 如何查找要管理的 Pods。在这里，你只需选择在 Pod 模板中定义的标签 (app: nginx)。不过，更复杂的选择规则是也可能的，只要 Pod 模板本身满足所给规则即可。

说明：matchLabels 字段是 {key,value} 偶对的映射。在 matchLabels 映射中的单个 {key,value} 映射等效于 matchExpressions 中的一个元素，即其 key 字段是 "key"，operator 为 "In"，value 数组仅包含 "value"。在 matchLabels 和 matchExpressions 中给出的所有条件都必须满足才能匹配。

- template 字段包含以下子字段：
 - Pod 被使用 labels 字段打上 app: nginx 标签。
 - Pod 模板规约 (即 .template.spec 字段) 指示 Pods 运行一个 nginx 容器，该容器运行版本为 1.14.2 的 nginx [Docker Hub](#) 镜像。
 - 创建一个容器并使用 name 字段将其命名为 nginx。

开始之前，请确保的 Kubernetes 集群已启动并运行。按照以下步骤创建上述 Deployment：

1. 通过运行以下命令创建 Deployment：

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

说明：你可以设置 --record 标志将所执行的命令写入资源注解 kubernetes.io/change-cause 中。这对于以后的检查是有用的。例如，要查看针对每个 Deployment 修订版本所执行过的命令。

1. 运行 kubectl get deployments 检查 Deployment 是否已创建。如果仍在创建 Deployment，则输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	0	0	0	1s

在检查集群中的 Deployment 时，所显示的字段有：

- NAME 列出了集群中 Deployment 的名称。
- READY 显示应用程序的可用的副本数。显示的模式是"就绪个数/期望个数"。
- UP-TO-DATE 显示为了打到期望状态已经更新的副本数。
- AVAILABLE 显示应用可供用户使用的副本数。
- AGE 显示应用程序运行的时间。

请注意期望副本数是根据 `.spec.replicas` 字段设置 3。

- 要查看 Deployment 上线状态，运行 `kubectl rollout status deployment/nginx-deployment`。

输出类似于：

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "nginx-deployment" successfully rolled out
```

- 几秒钟后再次运行 `kubectl get deployments`。输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	18s

注意 Deployment 已创建全部三个副本，并且所有副本都是最新的（它们包含最新的 Pod 模板）并且可用。

- 要查看 Deployment 创建的 ReplicaSet (rs)，运行 `kubectl get rs`。输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-75675f5897	3	3	3	18s

ReplicaSet 输出中包含以下字段：

- NAME 列出名字空间中 ReplicaSet 的名称；
- DESIRED 显示应用的期望副本个数，即在创建 Deployment 时所定义的值。此为期望状态；
- CURRENT 显示当前运行状态中的副本个数；
- READY 显示应用中有多少副本可以为用户提供服务；
- AGE 显示应用已经运行的时间长度。

注意 ReplicaSet 的名称始终被格式化为[Deployment名称]-[随机字符串]。其中的随机字符串是使用 `pod-template-hash` 作为种子随机生成的。

- 要查看每个 Pod 自动生成的标签，运行 `kubectl get pods --show-labels`。返回以下输出：

NAME	READY	STATUS	RESTARTS	AGE	LABELS
nginx-deployment-75675f5897-7ci7o	1/1	Running	0	18s	app=nginx,pod-template-hash=3123191453
nginx-deployment-75675f5897-kzszej	1/1	Running	0	18s	app=nginx,pod-template-hash=3123191453
nginx-deployment-75675f5897-qqcnn	1/1	Running	0	18s	app=nginx,pod-template-hash=3123191453

所创建的 ReplicaSet 确保总是存在三个 nginx Pod。

说明：你必须在 Deployment 中指定适当的选择算符和 Pod 模板标签（在本例中为 app: nginx）。标签或者选择算符不要与其他控制器（包括其他 Deployment 和 StatefulSet）重叠。Kubernetes 不会阻止你这样做，但是如果多个控制器具有重叠的选择算符，它们可能会发生冲突 执行难以预料的操作。

Pod-template-hash 标签

说明：不要更改此标签。

Deployment 控制器将 pod-template-hash 标签添加到 Deployment 所创建或收留的每个 ReplicaSet。

此标签可确保 Deployment 的子 ReplicaSets 不重叠。标签是通过对 ReplicaSet 的 PodTemplate 进行哈希处理。所生成的哈希值被添加到 ReplicaSet 选择算符、Pod 模板标签，并存在于在 ReplicaSet 可能拥有的任何现有 Pod 中。

更新 Deployment

说明：仅当 Deployment Pod 模板（即 .spec.template）发生改变时，例如模板的标签或容器镜像被更新，才会触发 Deployment 上线。其他更新（如对 Deployment 执行扩缩容的操作）不会触发上线动作。

按照以下步骤更新 Deployment：

1. 先来更新 nginx Pod 以使用 nginx:1.16.1 镜像，而不是 nginx:1.14.2 镜像。

```
kubectl --record deployment.apps/nginx-deployment set image \
deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
```

或者使用下面的命令：

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 --record
```

输出类似于：

```
deployment.apps/nginx-deployment image updated
```

或者，可以 edit Deployment 并将 .spec.template.spec.containers[0].image 从 nginx:1.14.2 更改至 nginx:1.16.1。

```
kubectl edit deployment.v1.apps/nginx-deployment
```

输出类似于：

```
deployment.apps/nginx-deployment edited
```

1. 要查看上线状态，运行：

```
kubectl rollout status deployment/nginx-deployment
```

输出类似于：

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

或者

```
deployment "nginx-deployment" successfully rolled out
```

获取关于已更新的 Deployment 的更多信息：

- 在上线成功后，可以通过运行 `kubectl get deployments` 来查看 Deployment：
输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	36s

- 运行 `kubectl get rs` 以查看 Deployment 通过创建新的 ReplicaSet 并将其扩容到 3 个副本并将旧 ReplicaSet 缩容到 0 个副本完成了 Pod 的更新操作：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1564180365	3	3	3	6s
nginx-deployment-2035384211	0	0	0	36s

- 现在运行 `get pods` 应仅显示新的 Pods:

```
kubectl get pods
```

输出类似于：

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

下次要更新这些 Pods 时，只需再次更新 Deployment Pod 模板即可。

Deployment 可确保在更新时仅关闭一定数量的 Pod。默认情况下，它确保至少所需 Pods 75% 处于运行状态（最大不可用比例为 25%）。

Deployment 还确保仅所创建 Pod 数量只可能比期望 Pods 数高一点点。默认情况下，它可确保启动的 Pod 个数比期望个数最多多出 25%（最大峰值 25%）。

例如，如果仔细查看上述 Deployment，将看到它首先创建了一个新的 Pod，然后删除了一些旧的 Pods，并创建了新的 Pods。它不会杀死老 Pods，直到有足够的

的数量新的 Pods 已经出现。在足够数量的旧 Pods 被杀死前并没有创建新 Pods。它确保至少 2 个 Pod 可用，同时最多总共 4 个 Pod 可用。

- 获取 Deployment 的更多信息

```
kubectl describe deployments
```

输出类似于：

```
Name:          nginx-deployment
Namespace:     default
CreationTimestamp: Thu, 30 Nov 2017 10:56:25 +0000
Labels:         app=nginx
Annotations:   deployment.kubernetes.io/revision=2
Selector:       app=nginx
Replicas:      3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image:    nginx:1.16.1
      Port:     80/TCP
      Environment: <none>
      Mounts:    <none>
      Volumes:   <none>
  Conditions:
    Type  Status  Reason
    ----  -----  -----
    Available  True  MinimumReplicasAvailable
    Progressing  True  NewReplicaSetAvailable
  OldReplicaSets: <none>
  NewReplicaSet:  nginx-deployment-1564180365 (3/3 replicas created)
  Events:
    Type  Reason  Age  From           Message
    ----  -----  ---  ----
    Normal  ScalingReplicaSet  2m  deployment-controller  Scaled up
replica set nginx-deployment-2035384211 to 3
    Normal  ScalingReplicaSet  24s  deployment-controller  Scaled up replica
set nginx-deployment-1564180365 to 1
    Normal  ScalingReplicaSet  22s  deployment-controller  Scaled down
replica set nginx-deployment-2035384211 to 2
    Normal  ScalingReplicaSet  22s  deployment-controller  Scaled up replica
set nginx-deployment-1564180365 to 2
    Normal  ScalingReplicaSet  19s  deployment-controller  Scaled down
```

```
replica set nginx-deployment-2035384211 to 1
  Normal ScalingReplicaSet 19s deployment-controller Scaled up replica
  set nginx-deployment-1564180365 to 3
  Normal ScalingReplicaSet 14s deployment-controller Scaled down
  replica set nginx-deployment-2035384211 to 0
```

可以看到，当第一次创建 Deployment 时，它创建了一个 ReplicaSet (nginx-deployment-2035384211) 并将其直接扩容至 3 个副本。更新 Deployment 时，它创建了一个新的 ReplicaSet (nginx-deployment-1564180365)，并将其扩容为 1，然后将旧 ReplicaSet 缩容到 2，以便至少有 2 个 Pod 可用且最多创建 4 个 Pod。然后，它使用相同的滚动更新策略继续对新的 ReplicaSet 扩容并对旧的 ReplicaSet 缩容。最后，你将有 3 个可用的副本在新的 ReplicaSet 中，旧 ReplicaSet 将缩容到 0。

翻转 (多 Deployment 动态更新)

Deployment 控制器每次注意到新的 Deployment 时，都会创建一个 ReplicaSet 以启动所需的 Pods。如果更新了 Deployment，则控制标签匹配 .spec.selector 但模板不匹配 .spec.template 的 Pods 的现有 ReplicaSet 被缩容。最终，新的 ReplicaSet 缩放为 .spec.replicas 个副本，所有旧 ReplicaSets 缩放为 0 个副本。

当 Deployment 正在上线时被更新，Deployment 会针对更新创建一个新的 ReplicaSet 并开始对其扩容，之前正在被扩容的 ReplicaSet 会被翻转，添加到旧 ReplicaSets 列表 并开始缩容。

例如，假定你在创建一个 Deployment 以生成 nginx:1.14.2 的 5 个副本，但接下来更新 Deployment 以创建 5 个 nginx:1.16.1 的副本，而此时只有 3 个 nginx:1.14.2 副本已创建。在这种情况下，Deployment 会立即开始杀死 3 个 nginx:1.14.2 Pods，并开始创建 nginx:1.16.1 Pods。它不会等待 nginx:1.14.2 的 5 个副本都创建完成 后才开始执行变更动作。

更改标签选择算符

通常不鼓励更新标签选择算符。建议你提前规划选择算符。在任何情况下，如果需要更新标签选择算符，请格外小心，并确保自己了解 这背后可能发生的所有事情。

说明：在 API 版本 apps/v1 中，Deployment 标签选择算符在创建后是不可变的。

- 添加选择算符时要求使用新标签更新 Deployment 规约中的 Pod 模板标签，否则将返回验证错误。此更改是非重叠的，也就是说新的选择算符不会选择使用旧选择算符所创建的 ReplicaSet 和 Pod，这会导致创建新的 ReplicaSet 时所有旧 ReplicaSet 都会被孤立。
- 选择算符的更新如果更改了某个算符的键名，这会导致与添加算符时相同的行为。
- 删除选择算符的操作会删除从 Deployment 选择算符中删除现有算符。此操作不需要更改 Pod 模板标签。现有 ReplicaSet 不会被孤立，也不会因此创建新的 ReplicaSet，但请注意已删除的标签仍然存在于现有的 Pod 和 ReplicaSet 中。

回滚 Deployment

有时，你可能想要回滚 Deployment；例如，当 Deployment 不稳定时（例如进入反复崩溃状态）。默认情况下，Deployment 的所有上线记录都保留在系统中，以便可以随时回滚（你可以通过修改修订历史记录限制来更改这一约束）。

说明：Deployment 被触发上线时，系统就会创建 Deployment 的新的修订版本。这意味着仅当 Deployment 的 Pod 模板 (.spec.template) 发生更改时，才会创建新修订版本 -- 例如，模板的标签或容器镜像发生变化。其他更新，如 Deployment 的扩缩容操作不会创建 Deployment 修订版本。这是为了方便同时执行手动缩放或自动缩放。换言之，当你回滚到较早的修订版本时，只有 Deployment 的 Pod 模板部分会被回滚。

- 假设你在更新 Deployment 时犯了一个拼写错误，将镜像名称命名设置为 nginx:1.161 而不是 nginx:1.16.1：

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.161 --record=true
```

输出类似于：

```
deployment.apps/nginx-deployment image updated
```

- 此上线进程会出现停滞。你可以通过检查上线状态来验证：

```
kubectl rollout status deployment/nginx-deployment
```

输出类似于：

```
Waiting for rollout to finish: 1 out of 3 new replicas have been updated...
```

- 按 Ctrl-C 停止上述上线状态观测。有关上线停滞的详细信息，[参考这里](#)。
- 你可以看到旧的副本有两个（nginx-deployment-1564180365 和 nginx-deployment-2035384211），新的副本有 1 个（nginx-deployment-3066724191）：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1564180365	3	3	3	25s
nginx-deployment-2035384211	0	0	0	36s
nginx-deployment-3066724191	1	1	0	6s

- 查看所创建的 Pod，你会注意到新 ReplicaSet 所创建的 1 个 Pod 卡顿在镜像拉取循环中。

```
kubectl get pods
```

输出类似于：

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1564180365-70iae	1/1	Running	0	25s
nginx-deployment-1564180365-jbqyo	1/1	Running	0	25s
nginx-deployment-1564180365-hysrc	1/1	Running	0	25s
nginx-deployment-3066724191-08mng	0/1	ImagePullBackOff	0	6s

说明： Deployment 控制器自动停止有问题的上线过程，并停止对新的 ReplicaSet 扩容。这行为取决于所指定的 rollingUpdate 参数（具体为 maxUnavailable）。默认情况下，Kubernetes 将此值设置为 25%。

- 获取 Deployment 描述信息：

```
kubectl describe deployment
```

输出类似于：

```
Name:      nginx-deployment
Namespace:  default
CreationTimestamp: Tue, 15 Mar 2016 14:48:04 -0700
Labels:     app=nginx
Selector:   app=nginx
Replicas:   3 desired | 1 updated | 4 total | 3 available | 1 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
    nginx:
      Image:  nginx:1.91
      Port:   80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts:  <none>
      Volumes: <none>
  Conditions:
    Type      Status  Reason
    ----      ----  -----
    Available  True   MinimumReplicasAvailable
    Progressing True   ReplicaSetUpdated
  OldReplicaSets:  nginx-deployment-1564180365 (3/3 replicas created)
  NewReplicaSet:   nginx-deployment-3066724191 (1/1 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 1			
21s	21s	1	{deployment-controller}		Normal
ScalingReplicaSet	Scaled up replica	set nginx-deployment-1564180365 to 3			
13s	13s	1	{deployment-controller}		Normal
ScalingReplicaSet	Scaled down replica	set nginx-deployment-2035384211 to 0			
13s	13s	1	{deployment-controller}		Normal
ScalingReplicaSet	Scaled up replica	set nginx-deployment-3066724191 to 1			

要解决此问题，需要回滚到以前稳定的 Deployment 版本。

检查 Deployment 上线历史

按照如下步骤检查回滚历史：

- 首先，检查 Deployment 修订历史：

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

输出类似于：

```
deployments "nginx-deployment"
REVISION  CHANGE-CAUSE
1        kubectl apply --filename=https://k8s.io/examples/controllers/nginx-deployment.yaml --record=true
2        kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true
3        kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.9.1 --record=true
```

CHANGE-CAUSE 的内容是从 Deployment 的 kubernetes.io/change-cause 注解复制过来的。复制动作发生在修订版本创建时。你可以通过以下方式设置 CHANGE-CAUSE 消息：

- 使用 kubectl annotate deployment.v1.apps/nginx-deployment kubernetes.io/change-cause="image updated to 1.9.1" 为 Deployment 添加注解。
- 追加 --record 命令行标志以保存正在更改资源的 kubectl 命令。
- 手动编辑资源的清单。

1. 要查看修订历史的详细信息，运行：

```
kubectl rollout history deployment.v1.apps/nginx-deployment --revision=2
```

输出类似于：

```
deployments "nginx-deployment" revision 2
Labels:    app=nginx
           pod-template-hash=1159050644
Annotations: kubernetes.io/change-cause=kubectl set image
deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1 --record=true
Containers:
nginx:
Image:    nginx:1.16.1
Port:     80/TCP
QoS Tier:
cpu:      BestEffort
memory:   BestEffort
Environment Variables: <none>
No volumes.
```

回滚到之前的修订版本

按照下面给出的步骤将 Deployment 从当前版本回滚到以前的版本（即版本 2）。

1. 假定现在你已决定撤消当前上线并回滚到以前的修订版本：

```
kubectl rollout undo deployment.v1.apps/nginx-deployment
```

输出类似于：

```
deployment.apps/nginx-deployment
```

或者，你也可以通过使用 --to-revision 来回滚到特定修订版本：

```
kubectl rollout undo deployment.v1.apps/nginx-deployment --to-revision=2
```

输出类似于：

```
deployment.apps/nginx-deployment
```

与回滚相关的指令的更详细信息，请参考 [kubectl rollout](#)。

现在，Deployment 正在回滚到以前的稳定版本。正如你所看到的，Deployment 控制器生成了回滚到修订版本 2 的 DeploymentRollback 事件。

1. 检查回滚是否成功以及 Deployment 是否正在运行，运行：

```
kubectl get deployment nginx-deployment
```

输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3	3	3	3	30m

1. 获取 Deployment 描述信息：

```
kubectl describe deployment nginx-deployment
```

输出类似于：

```
Name:           nginx-deployment
Namespace:      default
CreationTimestamp: Sun, 02 Sep 2018 18:17:55 -0500
Labels:          app=nginx
Annotations:    deployment.kubernetes.io/revision=4
                  kubernetes.io/change-cause=kubectl set image
deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1 --record=true
Selector:        app=nginx
Replicas:        3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image:    nginx:1.16.1
      Port:     80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts:    <none>
      Volumes:   <none>
  Conditions:
    Type      Status  Reason
    ----      ----  -----
    Available  True   MinimumReplicasAvailable
```

```
Progressing True NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: nginx-deployment-c4747d96c (3/3 replicas created)
Events:
  Type Reason          Age From             Message
  ---- ----          ---- ----
  Normal ScalingReplicaSet 12m deployment-controller Scaled up
replica set nginx-deployment-75675f5897 to 3
  Normal ScalingReplicaSet 11m deployment-controller Scaled up
replica set nginx-deployment-c4747d96c to 1
  Normal ScalingReplicaSet 11m deployment-controller Scaled down
replica set nginx-deployment-75675f5897 to 2
  Normal ScalingReplicaSet 11m deployment-controller Scaled up
replica set nginx-deployment-c4747d96c to 2
  Normal ScalingReplicaSet 11m deployment-controller Scaled down
replica set nginx-deployment-75675f5897 to 1
  Normal ScalingReplicaSet 11m deployment-controller Scaled up
replica set nginx-deployment-c4747d96c to 3
  Normal ScalingReplicaSet 11m deployment-controller Scaled down
replica set nginx-deployment-75675f5897 to 0
  Normal ScalingReplicaSet 11m deployment-controller Scaled up
replica set nginx-deployment-595696685f to 1
  Normal DeploymentRollback 15s deployment-controller Rolled back
deployment "nginx-deployment" to revision 2
  Normal ScalingReplicaSet 15s deployment-controller Scaled down
replica set nginx-deployment-595696685f to 0
```

缩放 Deployment

你可以使用如下指令缩放 Deployment :

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10
```

输出类似于 :

```
deployment.apps/nginx-deployment scaled
```

假设集群启用了[Pod 的水平自动缩放](#)，你可以为 Deployment 设置自动缩放器，并基于现有 Pods 的 CPU 利用率选择要运行的 Pods 个数下限和上限。

```
kubectl autoscale deployment.v1.apps/nginx-deployment --min=10 --max=15 --
cpu-percent=80
```

输出类似于 :

```
deployment.apps/nginx-deployment scaled
```

比例缩放

RollingUpdate 的 Deployment 支持同时运行应用程序的多个版本。当自动缩放器缩放处于上线进程（仍在进行中或暂停）中的 RollingUpdate Deployment 时，Deployment 控制器会平衡现有的活跃状态的 ReplicaSets（含 Pods 的 ReplicaSets）中的额外副本，以降低风险。这称为 **比例缩放** (*Proportional Scaling*)。

例如，你正在运行一个 10 个副本的 Deployment，其 [maxSurge=3](#)，[maxUnavailable=2](#)。

- 确保 Deployment 的这 10 个副本都在运行。

```
kubectl get deploy
```

输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	10	10	10	10	50s

- 更新 Deployment 使用新镜像，碰巧该镜像无法从集群内部解析。

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:some-tag
```

输出类似于：

```
deployment.apps/nginx-deployment image updated
```

- 镜像更新使用 ReplicaSet nginx-deployment-1989198191 启动新的上线过程，但由于上面提到的 maxUnavailable 要求，该进程被阻塞了。检查上线状态：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1989198191	5	5	0	9s
nginx-deployment-618515232	8	8	8	1m

- 然后，出现了新的 Deployment 扩缩请求。自动缩放器将 Deployment 副本增加到 15。Deployment 控制器需要决定在何处添加 5 个新副本。如果未使用比例缩放，所有 5 个副本都将添加到新的 ReplicaSet 中。使用比例缩放时，可以将额外的副本分布到所有 ReplicaSet。较大比例的副本会被添加到拥有最多副本的 ReplicaSet，而较低比例的副本会进入到副本较少的 ReplicaSet。所有剩下的副本都会添加到副本最多的 ReplicaSet。具有零副本的 ReplicaSets 不会被扩容。

在上面的示例中，3 个副本被添加到旧 ReplicaSet 中，2 个副本被添加到新 ReplicaSet。假定新的副本都很健康，上线过程最终应将所有副本迁移到新的 ReplicaSet 中。要确认这一点，请运行：

```
kubectl get deploy
```

输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	15	18	7	8	7m

上线状态确认了副本是如何被添加到每个 ReplicaSet 的。

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-deployment-1989198191	7	7	0	7m
nginx-deployment-618515232	11	11	11	7m

暂停、恢复 Deployment

你可以在触发一个或多个更新之前暂停 Deployment，然后再恢复其执行。这样做使得你能够在暂停和恢复执行之间应用多个修补程序，而不会触发不必要的上线操作。

- 例如，对于一个刚刚创建的 Deployment：获取 Deployment 信息：

```
kubectl get deploy
```

输出类似于：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx	3	3	3	3	1m

获取上线状态：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	3	3	3	1m

- 使用如下指令暂停运行：

```
kubectl rollout pause deployment.v1.apps/nginx-deployment
```

输出类似于：

```
deployment.apps/nginx-deployment paused
```

- 接下来更新 Deployment 镜像：

```
kubectl set image deployment.v1.apps/nginx-deployment nginx=nginx:1.16.1
```

输出类似于：

```
deployment.apps/nginx-deployment image updated
```

- 注意没有新的上线被触发：

```
kubectl rollout history deployment.v1.apps/nginx-deployment
```

输出类似于：

```
deployments "nginx"
REVISION CHANGE-CAUSE
1 <none>
```

- 获取上线状态确保 Deployment 更新已经成功：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	3	3	3	2m

- 你可以根据需要执行很多更新操作，例如，可以要使用的资源：

```
kubectl set resources deployment.v1.apps/nginx-deployment -c=nginx --limits=cpu=200m,memroy=512Mi
```

输出类似于：

```
deployment.apps/nginx-deployment resource requirements updated
```

暂停 Deployment 之前的初始状态将继续发挥作用，但新的更新在 Deployment 被 暂停期间不会产生任何效果。

- 最终，恢复 Deployment 执行并观察新的 ReplicaSet 的创建过程，其中包含了所应用的所有更新：

```
kubectl rollout resume deployment.v1.apps/nginx-deployment
```

输出：

```
deployment.apps/nginx-deployment resumed
```

- 观察上线的状态，直到完成。

```
kubectl get rs -w
```

输出类似于：

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

- 获取最近上线的状态：

```
kubectl get rs
```

输出类似于：

NAME	DESIRED	CURRENT	READY	AGE
nginx-2142116321	0	0	0	2m
nginx-3926361531	3	3	3	28s

说明：你不可回滚处于暂停状态的 Deployment，除非先恢复其执行状态。

Deployment 状态

Deployment 的生命周期中会有许多状态。上线新的 ReplicaSet 期间可能处于 [Progressing \(进行中\)](#)，可能是 [Complete \(已完成\)](#)，也可能是 [Failed \(失败\)](#) 以至于无法继续进行。

进行中的 Deployment

执行下面的任务期间，Kubernetes 标记 Deployment 为 [进行中 \(Progressing\)](#)：

- Deployment 创建新的 ReplicaSet
- Deployment 正在为其最新的 ReplicaSet 扩容
- Deployment 正在为其旧有的 ReplicaSet(s) 缩容
- 新的 Pods 已经就绪或者可用（就绪至少持续了 [MinReadySeconds](#) 秒）。

你可以使用 kubectl rollout status 监视 Deployment 的进度。

完成的 Deployment

当 Deployment 具有以下特征时，Kubernetes 将其标记为 完成 (*Complete*) :

- 与 Deployment 关联的所有副本都已更新到指定的最新版本，这意味着之前请求的所有更新都已完成。
- 与 Deployment 关联的所有副本都可用。
- 未运行 Deployment 的旧副本。

你可以使用 `kubectl rollout status` 检查 Deployment 是否已完成。如果上线成功完成，`kubectl rollout status` 返回退出代码 0。

```
kubectl rollout status deployment/nginx-deployment
```

输出类似于：

```
Waiting for rollout to finish: 2 of 3 updated replicas are available...
deployment "nginx-deployment" successfully rolled out
$ echo $?
0
```

失败的 Deployment

你的 Deployment 可能会在尝试部署其最新的 ReplicaSet 受挫，一直处于未完成状态。造成此情况一些可能因素如下：

- 配额 (Quota) 不足
- 就绪探测 (Readiness Probe) 失败
- 镜像拉取错误
- 权限不足
- 限制范围 (Limit Ranges) 问题
- 应用程序运行时的配置错误

检测此状况的一种方法是在 Deployment 规约中指定截止时间参数： (`[.spec.progressDeadlineSeconds]` (#progress-deadline-seconds))。 `.spec.progressDeadlineSeconds` 给出的是一个秒数值，Deployment 控制器在 (通过 Deployment 状态) 标示 Deployment 进展停滞之前，需要等待所给的时长。

以下 `kubectl` 命令设置规约中的 `progressDeadlineSeconds`，从而告知控制器在 10 分钟后报告 Deployment 没有进展：

```
kubectl patch deployment.v1.apps/nginx-deployment -p '{"spec": {"progressDeadlineSeconds":600}}'
```

输出类似于：

```
deployment.apps/nginx-deployment patched
```

超过截止时间后，Deployment 控制器将添加具有以下属性的 DeploymentCondition 到 Deployment 的 .status.conditions 中：

- Type=Progressing
- Status=False
- Reason=ProgressDeadlineExceeded

参考 [Kubernetes API 约定](#) 获取更多状态状况相关的信息。

说明：

除了报告 Reason=ProgressDeadlineExceeded 状态之外，Kubernetes 对已停止的 Deployment 不执行任何操作。更高级别的编排器可以利用这一设计并相应地采取行动。例如，将 Deployment 回滚到其以前的版本。

如果你暂停了某个 Deployment，Kubernetes 不再根据指定的截止时间检查 Deployment 进展。你可以在上线过程中间安全地暂停 Deployment 再恢复其执行，这样做不会导致超出最后时限的问题。

Deployment 可能会出现瞬时性的错误，可能因为设置的超时时间过短，也可能因为其他可认为是临时性的问题。例如，假定所遇到的问题是配额不足。如果描述 Deployment，你将会注意到以下部分：

```
kubectl describe deployment nginx-deployment
```

输出类似于：

```
<...>
Conditions:
Type      Status  Reason
----      ----- -----
Available  True    MinimumReplicasAvailable
Progressing  True   ReplicaSetUpdated
ReplicaFailure  True  FailedCreate
<...>
```

如果运行 kubectl get deployment nginx-deployment -o yaml，Deployment 状态输出 将类似于这样：

```
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
```

最终，一旦超过 Deployment 进度限期，Kubernetes 将更新状态和进度状况的原因：

Conditions:

Type	Status	Reason
Available	True	MinimumReplicasAvailable
Progressing	False	ProgressDeadlineExceeded
ReplicaFailure	True	FailedCreate

可以通过缩容 Deployment 或者缩容其他运行状态的控制器，或者直接在命名空间中增加配额 来解决配额不足的问题。如果配额条件满足，Deployment 控制器完成了 Deployment 上线操作，Deployment 状态会更新为成功状况（Status=True and Reason>NewReplicaSetAvailable）。

Conditions:

Type	Status	Reason
Available	True	MinimumReplicasAvailable
Progressing	True	NewReplicaSetAvailable

Type=Available 加上 Status=True 意味着 Deployment 具有最低可用性。最低可用性由 Deployment 策略中的参数指定。Type=Progressing 加上 Status=True 表示 Deployment 处于上线过程中，并且正在运行，或者已成功完成进度，最小所需新副本处于可用。请参阅对应状况的 Reason 了解相关细节。在我们的案例中 Reason>NewReplicaSetAvailable 表示 Deployment 已完成。

你可以使用 kubectl rollout status 检查 Deployment 是否未能取得进展。如果 Deployment 已超过进度限期，kubectl rollout status 返回非零退出代码。

```
kubectl rollout status deployment/nginx-deployment
```

输出类似于：

```
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
error: deployment "nginx" exceeded its progress deadline
```

kubectl rollout 命令的退出状态为 1（表明发生了错误）：

```
$ echo $?
```

```
1
```

对失败 Deployment 的操作

可应用于已完成的 Deployment 的所有操作也适用于失败的 Deployment。你可以对其执行扩缩容、回滚到以前的修订版本等操作，或者在需要对 Deployment 的 Pod 模板应用多项调整时，将 Deployment 暂停。

清理策略

你可以在 Deployment 中设置 `.spec.revisionHistoryLimit` 字段以指定保留此 Deployment 的多少个旧有 ReplicaSet。其余的 ReplicaSet 将在后台被垃圾回收。默认情况下，此值为 10。

说明：显式将此字段设置为 0 将导致 Deployment 的所有历史记录被清空，因此 Deployment 将无法回滚。

金丝雀部署

如果要使用 Deployment 向用户子集或服务器子集上线版本，则可以遵循 [资源管理](#) 所描述的金丝雀模式，创建多个 Deployment，每个版本一个。

编写 Deployment 规约

同其他 Kubernetes 配置一样，Deployment 需要 `apiVersion`，`kind` 和 `metadata` 字段。有关配置文件的其他信息，请参考 [部署 Deployment](#)、[配置容器](#)和 [使用 kubectl 管理资源](#)等相关文档。

Deployment 对象的名称必须是合法的 [DNS 子域名](#)。Deployment 还需要 [.spec 部分](#)。

Pod 模板

`.spec` 中只有 `.spec.template` 和 `.spec.selector` 是必需的字段。

`.spec.template` 是一个 [Pod 模板](#)。它和 [Pod](#) 的语法规则完全相同。只是这里它是嵌套的，因此不需要 `apiVersion` 或 `kind`。

除了 Pod 的必填字段外，Deployment 中的 Pod 模板必须指定适当的标签和适当的新启动策略。对于标签，请确保不要与其他控制器重叠。请参考[选择算符](#)。

只有 [.spec.template.spec.restartPolicy](#) 等于 Always 才是被允许的，这也是在没有指定时的默认设置。

副本

.spec.replicas 是指定所需 Pod 的可选字段。它的默认值是1。

选择算符

.spec.selector 是指定本 Deployment 的 Pod [标签选择算符](#)的必需字段。

.spec.selector 必须匹配 .spec.template.metadata.labels，否则请求会被 API 拒绝。

在 API apps/v1版本中，.spec.selector 和 .metadata.labels 如果没有设置的话，不会被默认设置为 .spec.template.metadata.labels，所以需要明确进行设置。同时在 apps /v1版本中，Deployment 创建后 .spec.selector 是不可变的。

当 Pod 的标签和选择算符匹配，但其模板和 .spec.template 不同时，或者此类 Pod 的总数超过 .spec.replicas 的设置时，Deployment 会终结之。如果 Pods 总数未达到期望值，Deployment 会基于 .spec.template 创建新的 Pod。

说明：你不应直接创建、或者通过创建另一个 Deployment，或者创建类似 ReplicaSet 或 ReplicationController 这类控制器来创建标签与此选择算符匹配的 Pod。如果这样做，第一个 Deployment 会认为它创建了这些 Pod。Kubernetes 不会阻止你这么做。

如果有多个控制器的选择算符发生重叠，则控制器之间会因冲突而无法正常工作。

策略

.spec.strategy 策略指定用于用新 Pods 替换旧 Pods 的策略。.spec.strategy.type 可以是 "Recreate" 或 "RollingUpdate"。"RollingUpdate" 是默认值。

重新创建 Deployment

如果 .spec.strategy.type==Recreate，在创建新 Pods 之前，所有现有的 Pods 会被杀死。

滚动更新 Deployment

Deployment 会在 .spec.strategy.type==RollingUpdate时，采取 滚动更新的方式更新 Pods。你可以指定 maxUnavailable 和 maxSurge 来控制滚动更新 过程。

最大不可用

.spec.strategy.rollingUpdate.maxUnavailable 是一个可选字段，用来指定 更新过程中不可用的 Pod 的个数上限。该值可以是绝对数字（例如，5），也可以是 所需 Pods 的

百分比（例如，10%）。百分比值会转换成绝对数并去除小数部分。如果 `.spec.strategy.rollingUpdate.maxSurge` 为 0，则此值不能为 0。默认值为 25%。

例如，当此值设置为 30% 时，滚动更新开始时会立即将旧 ReplicaSet 缩容到期望 Pod 个数的 70%。新 Pod 准备就绪后，可以继续缩容旧有的 ReplicaSet，然后对新的 ReplicaSet 扩容，确保在更新期间 可用的 Pods 总数在任何时候都至少为所需的 Pod 个数的 70%。

最大峰值

`.spec.strategy.rollingUpdate.maxSurge` 是一个可选字段，用来指定可以创建的超出 期望 Pod 个数的 Pod 数量。此值可以是绝对数（例如，5）或所需 Pods 的百分比（例如，10%）。如果 `MaxUnavailable` 为 0，则此值不能为 0。百分比值会通过向上取整 转换为绝对数。此字段的默认值为 25%。

例如，当此值为 30% 时，启动滚动更新后，会立即对新的 ReplicaSet 扩容，同时保证 新旧 Pod 的总数不超过所需 Pod 总数的 130%。一旦旧 Pods 被杀死，新的 ReplicaSet 可以进一步扩容，同时确保更新期间的任何时候运行中的 Pods 总数最多为 所需 Pods 总数的 130%。

进度期限秒数

`.spec.progressDeadlineSeconds` 是一个可选字段，用于指定系统在报告 Deployment [进展失败](#) 之前等待 Deployment 取得进展的秒数。这类报告会在资源状态中体现为 `Type=Progressing`、`Status=False`、`Reason=ProgressDeadlineExceeded`。 Deployment 控制器将持续重试 Deployment。将来，一旦实现了自动回滚， Deployment 控制器将在探测到这样的条件时立即回滚 Deployment。

如果指定，则此字段值需要大于 `.spec.minReadySeconds` 取值。

最短就绪时间

`.spec.minReadySeconds` 是一个可选字段，用于指定新创建的 Pod 在没有任意容器崩溃情况下的最小就绪时间，只有超出这个时间 Pod 才被视为可用。默认值为 0（Pod 在准备就绪后立即将被视为可用）。要了解何时 Pod 被视为就绪，可参考[容器探针](#)。

修订历史限制

Deployment 的修订历史记录存储在它所控制的 ReplicaSets 中。

`.spec.revisionHistoryLimit` 是一个可选字段，用来设定出于会滚目的所要保留的旧 ReplicaSet 数量。这些旧 ReplicaSet 会消耗 etcd 中的资源，并占用 `kubectl get rs` 的输出。每个 Deployment 修订版本的配置都存储在其 ReplicaSets 中；因此，一旦 删除了旧的 ReplicaSet，将失去回滚到 Deployment 的对应修订版本的能力。默认情况下，系统保留 10 个旧 ReplicaSet，但其理想值取决于新 Deployment 的频率和稳定性。

更具体地说，将此字段设置为 0 意味着将清理所有具有 0 个副本的旧 ReplicaSet。在这种情况下，无法撤消新的 Deployment 上线，因为它的修订历史被清除了。

paused (暂停的)

.spec.paused 是用于暂停和恢复 Deployment 的可选布尔字段。暂停的 Deployment 和未暂停的 Deployment 的唯一区别是，Deployment 处于暂停状态时，PodTemplateSpec 的任何修改都不会触发新的上线。Deployment 在创建时是默认不会处于暂停状态。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 11, 2021 at 10:33 AM PST: [\[zh\] Resync concepts/workloads/controllers/deployment.md \(d5a626678\)](#)

ReplicaSet

ReplicaSet 的目的是维护一组在任何时候都处于运行状态的 Pod 副本的稳定集合。因此，它通常用来保证给定数量的、完全相同的 Pod 的可用性。

ReplicaSet 的工作原理

ReplicaSet 是通过一组字段来定义的，包括一个用来识别可获得的 Pod 的集合的选择算符、一个用来标明应该维护的副本个数的数值、一个用来指定应该创建新 Pod 以满足副本个数条件时要使用的 Pod 模板等等。每个 ReplicaSet 都通过根据需要创建和删除 Pod 以使得副本个数达到期望值，进而实现其存在价值。当 ReplicaSet 需要创建新的 Pod 时，会使用所提供的 Pod 模板。

ReplicaSet 通过 Pod 上的 [metadata.ownerReferences](#) 字段连接到附属 Pod，该字段给出当前对象的属主资源。ReplicaSet 所获得的 Pod 都在其 ownerReferences 字段中包含了属主 ReplicaSet 的标识信息。正是通过这一连接，ReplicaSet 知道它所维护的 Pod 集合的状态，并据此计划其操作行为。

ReplicaSet 使用其选择算符来辨识要获得的 Pod 集合。如果某个 Pod 没有 OwnerReference 或者其 OwnerReference 不是一个 [控制器](#)，且其匹配到某 ReplicaSet 的选择算符，则该 Pod 立即被此 ReplicaSet 获得。

何时使用 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，除非你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，你可能永远不需要操作 ReplicaSet 对象：而是使用 Deployment，并在 spec 部分定义你的应用。

示例

[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
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

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 apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

你可以看到当前被部署的 ReplicaSet：

```
kubectl get rs
```

并看到你所创建的前端：

NAME	DESIRED	CURRENT	READY	AGE
frontend	3	3	3	6s

你也可以查看 ReplicaSet 的状态：

```
kubectl describe rs/frontend
```

你会看到类似如下的输出：

```
Name:      frontend
Namespace: default
Selector: tier=frontend
Labels:    app=guestbook
           tier=frontend
Annotations: kubectl.kubernetes.io/last-applied-configuration:
             {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"annotations":{},"labels":{"app":"guestbook","tier":"frontend"},"name":"frontend",...}}
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  tier=frontend
  Containers:
    php-redis:
      Image:   gcr.io/google_samples/gb-frontend:v3
      Port:    <none>
      Host Port: <none>
      Environment: <none>
      Mounts:   <none>
      Volumes:  <none>
  Events:
    Type  Reason        Age   From          Message
    ----  -----        --   --            --
    Normal  SuccessfulCreate  117s  replicaset-controller  Created pod: frontend-wtsmm
    Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-b2zdv
    Normal  SuccessfulCreate  116s  replicaset-controller  Created pod: frontend-vcmts
```

最后可以查看启动了的 Pods：

```
kubectl get pods
```

你会看到类似如下的 Pod 信息：

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	6m36s
frontend-vcmts	1/1	Running	0	6m36s
frontend-wtsmm	1/1	Running	0	6m36s

你也可以查看 Pods 的属主引用被设置为前端的 ReplicaSet。要实现这点，可取回运行中的 Pods 之一的 YAML：

```
kubectl get pods frontend-b2zdv -o yaml
```

输出将类似这样，frontend ReplicaSet 的信息被设置在 metadata 的 ownerReferences 字段中：

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-02-12T07:06:16Z"
  generateName: frontend-
  labels:
    tier: frontend
  name: frontend-b2zdv
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: frontend
    uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
...
...
```

非模板 Pod 的获得

尽管你完全可以直接创建裸的 Pods，强烈建议你确保这些裸的 Pods 并不包含可能与你的某个 ReplicaSet 的选择算符相匹配的标签。原因在于 ReplicaSet 并不仅限于拥有在其模板中设置的 Pods，它还可以像前面小节中所描述的那样获得其他 Pods。

[pods/pod-rs.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    tier: frontend
spec:
  containers:
  - name: hello1
    image: gcr.io/google-samples/hello-app:2.0
...

```

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

```
name: pod2
labels:
  tier: frontend
spec:
  containers:
    - name: hello2
      image: gcr.io/google-samples/hello-app:1.0
```

由于这些 Pod 没有控制器（Controller，或其他对象）作为其属主引用，并且其标签与 frontend ReplicaSet 的选择算符匹配，它们会立即被该 ReplicaSet 获取。

假定你在 frontend ReplicaSet 已经被部署之后创建 Pods，并且你已经在 ReplicaSet 中设置了其初始的 Pod 副本数以满足其副本计数需要：

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

新的 Pods 会被该 ReplicaSet 获取，并立即被 ReplicaSet 终止，因为它们的存在会使得 ReplicaSet 中 Pod 个数超出其期望值。

取回 Pods：

```
kubectl get pods
```

输出显示新的 Pods 或者已经被终止，或者处于终止过程中：

NAME	READY	STATUS	RESTARTS	AGE
frontend-b2zdv	1/1	Running	0	10m
frontend-vcmts	1/1	Running	0	10m
frontend-wtsmm	1/1	Running	0	10m
pod1	0/1	Terminating	0	1s
pod2	0/1	Terminating	0	1s

如果你先行创建 Pods：

```
kubectl apply -f https://kubernetes.io/examples/pods/pod-rs.yaml
```

之后再创建 ReplicaSet：

```
kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml
```

你会看到 ReplicaSet 已经获得了该 Pods，并仅根据其规约创建新的 Pods，直到新的 Pods 和原来的 Pods 的总数达到其预期个数。这时取回 Pods：

```
kubectl get pods
```

将会生成下面的输出：

NAME	READY	STATUS	RESTARTS	AGE
frontend-hmmj2	1/1	Running	0	9s

pod1	1/1	Running	0	36s
pod2	1/1	Running	0	36s

采用这种方式，一个 ReplicaSet 中可以包含异质的 Pods 集合。

编写 ReplicaSet 的 spec

与所有其他 Kubernetes API 对象一样，ReplicaSet 也需要 apiVersion、kind、和 metadata 字段。对于 ReplicaSets 而言，其 kind 始终是 ReplicaSet。在 Kubernetes 1.9 中，ReplicaSet 上的 API 版本 apps/v1 是其当前版本，且被默认启用。API 版本 apps/v1beta2 已被废弃。参考 frontend.yaml 示例的第一行。

ReplicaSet 对象的名称必须是合法的 [DNS 子域名](#)。

ReplicaSet 也需要 [spec](#) 部分。

Pod 模版

.spec.template 是一个[Pod 模版](#)，要求设置标签。在 frontend.yaml 示例中，我们指定了标签 tier: frontend。注意不要将标签与其他控制器的选择算符重叠，否则那些控制器会尝试收养此 Pod。

对于模板的[重启策略](#) 字段，.spec.template.spec.restartPolicy，唯一允许的取值是 Always，这也是默认值。

Pod 选择算符

.spec.selector 字段是一个[标签选择算符](#)。如前文中[所讨论的](#)，这些是用来标识要被获取的 Pods 的标签。在签名的 frontend.yaml 示例中，选择算符为：

matchLabels:

tier: frontend

在 ReplicaSet 中，.spec.template.metadata.labels 的值必须与 spec.selector 值相匹配，否则该配置会被 API 拒绝。

说明：

对于设置了相同的 .spec.selector，但 .spec.template.metadata.labels 和 .spec.template.spec 字段不同的两个 ReplicaSet 而言，每个 ReplicaSet 都会忽略被另一个 ReplicaSet 所创建的 Pods。

Replicas

你可以通过设置 .spec.replicas 来指定要同时运行的 Pod 个数。ReplicaSet 创建、删除 Pods 以与此值匹配。

如果你没有指定 .spec.replicas，那么默认值为 1。

使用 ReplicaSets

删除 ReplicaSet 和它的 Pod

要删除 ReplicaSet 和它的所有 Pod，使用 [kubectl delete](#) 命令。默认情况下，[垃圾收集器](#) 自动删除所有依赖的 Pod。

当使用 REST API 或 client-go 库时，你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/apps/v1/namespaces/default/replicasets/
frontend' \
-d '{"kind":"DeleteOptions","apiVersion":"v1","propagationPolicy":"Foreground"}'
-H "Content-Type: application/json"
```

只删除 ReplicaSet

你可以只删除 ReplicaSet 而不影响它的 Pod，方法是使用 [kubectl delete](#) 命令并设置 --cascade=false 选项。

当使用 REST API 或 client-go 库时，你必须将 propagationPolicy 设置为 Orphan。例如：

```
kubectl proxy --port=8080
curl -X DELETE 'localhost:8080/apis/apps/v1/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更新 Pod 的规约，可以使用 [Deployment](#) 资源，因为 ReplicaSet 并不直接支持滚动更新。

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 ReplicaSet

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 \(HPA\)](#) 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](#)



```
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 apply -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 --max=10 --min=3 --cpu-percent=50
```

ReplicaSet 的替代方案

Deployment (推荐)

[Deployment](#) 是一个可以拥有 ReplicaSet 并使用声明式方式在服务器端完成对 Pods 滚动更新的对象。尽管 ReplicaSet 可以独立使用，目前它们的主要用途是提供给 Deployment 作为编排 Pod 创建、删除和更新的一种机制。当使用 Deployment 时，你不必关心如何管理它所创建的 ReplicaSet，Deployment 拥有并管理其 ReplicaSet。因此，建议你在需要 ReplicaSet 时使用 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。因为这个原因，我们建议你使用 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](#) 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](#) 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。

ReplicationController

ReplicaSet 是[ReplicationController](#) 的后继者。二者目的相同且行为类似，只是 ReplicationController 不支持[标签用户指南](#) 中讨论的基于集合的选择算符需求。因此，相比于 ReplicationController，应优先考虑 ReplicaSet。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#). 在 GitHub 仓库上登记新的问题[报告问题](#) 或者[提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

StatefulSets

StatefulSet 是用来管理有状态应用的工作负载 API 对象。

StatefulSet 用来管理某[Pod](#) 集合的部署和扩缩，并为这些 Pod 提供持久存储和持久标识符。

和[Deployment](#) 类似，StatefulSet 管理基于相同容器规约的一组 Pod。但和 Deployment 不同的是，StatefulSet 为它们的每个 Pod 维护了一个有粘性的 ID。这些 Pod 是基于相同的规约来创建的，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。

如果希望使用存储卷为工作负载提供持久存储，可以使用 StatefulSet 作为解决方案的一部分。尽管 StatefulSet 中的单个 Pod 仍可能出现故障，但持久的 Pod 标识符使得将现有卷与替换已失败 Pod 的新 Pod 相匹配变得更加容易。

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。

在上面描述中，“稳定的”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](#) 或者 [ReplicaSet](#) 可能更适用于你的无状态应用部署需要。

限制

- 给定 Pod 的存储必须由 [PersistentVolume 驱动](#) 基于所请求的 storage class 来提供，或者由管理员预先提供。
- 删除或者收缩 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[无头服务](#)来负责 Pod 的网络标识。你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 可以有序地且体面地终止，可以在删除之前将 StatefulSet 缩放为 0。
- 在默认 [Pod 管理策略](#)(OrderedReady) 时使用 [滚动更新](#)，可能进入需要[人工干预](#)才能修复的损坏状态。

组件

下面的示例演示了 StatefulSet 的组件。

```
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

```

- 名为 nginx 的 Headless Service 用来控制网络域名。
- 名为 web 的 StatefulSet 有一个 Spec，它表明将在独立的 3 个 Pod 副本中启动 nginx 容器。
- volumeClaimTemplates 将通过 PersistentVolumes 驱动提供的 [PersistentVolumes](#) 来提供稳定的存储。

Pod 选择算符

你必须设置 StatefulSet 的 .spec.selector 字段，使之匹配其在 .spec.template.metadata.labels 中设置的标签。在 Kubernetes 1.8 版本之前，被忽略 .spec.selector 字段会获得默认设置值。在 1.8 和以后的版本中，未指定匹配的 Pod 选择器将在创建 StatefulSet 期间导致验证错误。

Pod 标识

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 名称)-\$(序号)。上例将会创建三个名称分别为 web-0、web-1、web-2 的 Pod。StatefulSet 可以使用 [无头服务](#) 控制它的 Pod 的网络域。管理域的这个服务的格式为：\$(服务名称).\$(命名空间).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：\$(pod 名称).\$(所属服务的 DNS 域名)，其中所属服务由 StatefulSet 的 serviceName 域来设定。

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：

集群域名	服务 (名字 空间/名 字)	StatefulSet (名 字空间/名字)	StatefulSet 域名	Pod DNS
cluster.local	default/ nginx	default/web	nginx.default.svc.cluster.local	web-{0..N-1}.nginx.defa
cluster.local	foo/ nginx	foo/web	nginx.foo.svc.cluster.local	web-{0..N-1}.nginx.foo.s
kube.local	foo/ nginx	foo/web	nginx.foo.svc.kube.local	web-{0..N-1}.nginx.

说明： 集群域会被设置为 cluster.local，除非有[其他配置](#)。

稳定的存储

Kubernetes 为每个 VolumeClaimTemplate 创建一个 [PersistentVolume](#)。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 被调度（重新调度）到节点上时，它的 volumeMounts 会挂载与其 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 \(Controller\)](#) 创建 Pod 时，它会添加一个标签 `statefulset.kubernetes.io/pod-name`，该标签值设置为 Pod 名称。这个标签允许你给 StatefulSet 中的特定 Pod 绑定一个 Service。

部署和扩缩保证

- 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 不应将 `pod.Spec.TerminationGracePeriodSeconds` 设置为 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 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](#) 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 允许你放宽其排序保证，同时通过它的 `.spec.podManagementPolicy` 域保持其唯一性和身份保证。

OrderedReady Pod 管理

OrderedReady Pod 管理是 StatefulSet 的默认设置。它实现了 [上面](#) 描述的功能。

并行 Pod 管理

Parallel 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` 字段让你可以配置和禁用掉自动滚动更新 Pod 的容器、标签、资源请求或限制、以及注解。

关于删除策略

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，你不需要使用分区，但如果你希望进行阶段更新、执行金丝雀或执行分阶段上线，则这些分区会非常有用。

强制回滚

在默认 [Pod 管理策略](#)(`OrderedReady`) 下使用 [滚动更新](#)，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](#)，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。](#)
- 示例二：[使用 StatefulSet 部署 Cassandra。](#)
- 示例三：[运行多副本的有状态应用程序。](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

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。

DaemonSet 的一些典型用法：

- 在每个节点上运行集群守护进程
- 在每个节点上运行日志收集守护进程
- 在每个节点上运行监控守护进程

一种简单的用法是为每种类型的守护进程在所有的节点上都启动一个 DaemonSet。一个稍微复杂的用法是为同一种守护进程部署多个 DaemonSet；每个具有不同的标志，并且对不同硬件类型具有不同的内存、CPU 要求。

编写 DaemonSet Spec

创建 DaemonSet

你可以在 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](#)



```
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:
        # this toleration is to have the daemonset runnable on master nodes
        # remove it if your masters can't run pods
        - 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

和所有其他 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，参见 [部署应用](#)、[配置容器](#)和[使用 kubectl 进行对象管理](#) 文档。

DaemonSet 对象的名称必须是一个合法的 [DNS 子域名](#)。

DaemonSet 也需要一个 [.spec](#) 配置段。

Pod 模板

.spec 中唯一必需的字段是 .spec.template。

.spec.template 是一个 [Pod 模板](#)。除了它是嵌套的，因而不具有 apiVersion 或 kind 字段之外，它与 [Pod](#) 具有相同的 schema。

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 选择算符](#)）。

在 DaemonSet 中的 Pod 模板必须具有一个值为 Always 的 [RestartPolicy](#)。当该值未指定时，默认是 Always。

Pod 选择算符

.spec.selector 字段表示 Pod 选择算符，它与 [Job](#) 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 选择算符。用户不指定 Pod 选择算符时，该字段不再有默认值。选择算符的默认值生成结果与 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 选择算符可能导致 Pod 意外悬浮，并且这对用户来说是费解的。

spec.selector 是一个对象，如下两个字段组成：

- matchLabels - 与 [ReplicationController](#) 的 .spec.selector 的作用相同。
- matchExpressions - 允许构建更加复杂的选择器，可以通过指定 key、value 列表以及将 key 和 value 列表关联起来的 operator。

当上述两个字段都指定时，结果会按逻辑与 (AND) 操作处理。

如果指定了 .spec.selector，必须与 .spec.template.metadata.labels 相匹配。如果与后者不匹配，则 DeamonSet 会被 API 拒绝。

仅在某些节点上运行 Pod

如果指定了 .spec.template.spec.nodeSelector，DaemonSet 控制器将在能够与 [Node 选择算符](#) 匹配的节点上创建 Pod。类似这种情况，可以指定 .spec.template.spec.affinity，之后 DaemonSet 控制器将在能够与[节点亲和性](#) 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。

Daemon Pods 是如何被调度的

通过默认调度器调度

FEATURE STATE: Kubernetes v1.20 [stable]

DaemonSet 确保所有符合条件的节点都运行该 Pod 的一个副本。通常，运行 Pod 的节点由 Kubernetes 调度器选择。不过，DaemonSet Pods 由 DaemonSet 控制器创建和调度。这就带来了以下问题：

- Pod 行为的不一致性：正常 Pod 在被创建后等待调度时处于 Pending 状态，DaemonSet Pods 创建后不会处于 Pending 状态下。这使用户感到困惑。
- **Pod 抢占** 由默认调度器处理。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 允许您使用默认调度器而不是 DaemonSet 控制器来调度 DaemonSets，方法是将 NodeAffinity 条件而不是 .spec.nodeName 条件添加到 DaemonSet Pods。默认调度器接下来将 Pod 绑定到目标主机。如果 DaemonSet Pod 的节点亲和性配置已存在，则被替换。DaemonSet 控制器仅在创建或修改 DaemonSet Pod 时执行这些操作，并且不会更改 DaemonSet 的 spec.template。

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchFields:

- **key**: metadata.name

operator: In

values:

- target-host-name

此外，系统会自动添加 node.kubernetes.io/unschedulable : NoSchedule 容忍度到 DaemonSet Pods。在调度 DaemonSet Pod 时，默认调度器会忽略 unschedulable 节点。

污点和容忍度

尽管 Daemon Pods 遵循[污点和容忍度](#) 规则，根据相关特性，控制器会自动将以下容忍度添加到 DaemonSet Pod：

容忍度键名	效果	版本	描述
node.kubernetes.io/not-ready	NoExecute	1.13+	当出现类似网络断开的情况导致节点问题时，DaemonSet Pod 不会被逐出。
node.kubernetes.io/unreachable	NoExecute	1.13+	当出现类似于网络断开的情况导致节点问题时，DaemonSet Pod 不会被逐出。
node.kubernetes.io/disk-pressure	NoSchedule	1.8+	

容忍度键名	效果	版本	描述
node.kubernetes.io/memory-pressure	NoSchedule	1.8+	
node.kubernetes.io/unschedulable	NoSchedule	1.12+	DaemonSet Pod 能够容忍默认调度器所设置的 unschedulable 属性。
node.kubernetes.io/network-unavailable	NoSchedule	1.12+	DaemonSet 在使用宿主网络时，能够容忍默认调度器所设置的 network-unavailable 属性。

与 Daemon Pods 通信

与 DaemonSet 中的 Pod 进行通信的几种可能模式如下：

- **推送 (Push)**：配置 DaemonSet 中的 Pod，将更新发送到另一个服务，例如统计数据库。这些服务没有客户端。
- **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获取节点 IP 列表，并且基于此也可以获取到相应的端口。
- **DNS**：创建具有相同 Pod 选择算符的 [无头服务](#)，通过使用 endpoints 资源或从 DNS 中检索到多个 A 记录来发现 DaemonSet。
- **Service**：创建具有相同 Pod 选择算符的服务，并使用该服务随机访问到某个节点上的守护进程（没有办法访问到特定节点）。

更新 DaemonSet

如果节点的标签被修改，DaemonSet 将立刻向新匹配上的节点添加 Pod，同时删除不匹配的节点上的 Pod。

你可以修改 DaemonSet 创建的 Pod。不过并非 Pod 的所有字段都可更新。下次当某节点（即使具有相同的名称）被创建时，DaemonSet 控制器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。接下来如果创建使用相同选择算符的新 DaemonSet，新的 DaemonSet 会收养已有的 Pod。如果有 Pod 需要被替换，DaemonSet 会根据其 updateStrategy 来替换。

你可以对 DaemonSet [执行滚动更新操作](#)。

DaemonSet 的替代方案

init 脚本

直接在节点上启动守护进程（例如使用 init、upstartd 或 systemd）的做法当然是可行的。不过，基于 DaemonSet 来运行这些进程有如下一些好处：

- 像所运行的其他应用一样，DaemonSet 具备为守护进程提供监控和日志管理的能力。
- 为守护进程和应用所使用的配置语言和工具（如 Pod 模板、kubectl）是相同的。
- 在资源受限的容器中运行守护进程能够增加守护进程和应用容器的隔离性。然而，这一点也可以通过在容器中运行守护进程但却不在 Pod 中运行之来实现。例如，直接基于 Docker 启动。

裸 Pod

直接创建 Pod 并指定其运行在特定的节点上也是可以的。然而，DaemonSet 能够替换由于任何原因（例如节点失败、例行节点维护、内核升级）而被删除或终止的 Pod。由于这个原因，你应该使用 DaemonSet 而不是单独创建 Pod。

静态 Pod

通过在一个指定的、受 kubelet 监视的目录下编写文件来创建 Pod 也是可行的。这类 Pod 被称为 [静态 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 API 服务器，这使得它们在启动引导新集群的情况下非常有用。此外，静态 Pod 在将来可能会被废弃。

Deployments

DaemonSet 与 [Deployments](#) 非常类似，它们都能创建 Pod，并且 Pod 中的进程都不希望被终止（例如，Web 服务器、存储服务器）。建议为无状态的服务使用 Deployments，比如前端服务。对这些服务而言，对副本的数量进行扩缩容、平滑升级，比精确控制 Pod 运行在某个主机上要重要得多。当需要 Pod 副本总是运行在全部或特定主机上，并需要它们先于其他 Pod 启动时，应该使用 DaemonSet。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 06, 2020 at 1:26 AM PST: [\[zh\] sync remove problems in DaemonSet fixed by controllerRef \(246ded8b5\)](#)

Jobs

Job 会创建一个或者多个 Pods，并确保指定数量的 Pods 成功终止。随着 Pods 成功结束，Job 跟踪记录成功完成的 Pods 个数。当数量达到指定的成功个数阈值时，任务（即 Job）结束。删除 Job 的操作会清除所创建的全部 Pods。

一种简单的使用场景下，你会创建一个 Job 对象以便以一种可靠的方式运行某 Pod 直到完成。当第一个 Pod 失败或者被删除（比如因为节点硬件失效或者重启）时，Job 对象会启动一个新的 Pod。

你也可以使用 Job 以并行的方式运行多个 Pod。

运行示例 Job

下面是一个 Job 配置示例。它负责计算 π 到小数点后 2000 位，并将结果打印出来。此计算大约需要 10 秒钟完成。

[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
```

你可以使用下面的命令来运行此示例：

```
kubectl apply -f https://kubernetes.io/examples/controllers/job.yaml
```

输出类似于：

```
job.batch/pi created
```

使用 kubectl 来检查 Job 的状态：

```
kubectl describe jobs/pi
```

输出类似于：

```
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
```

要查看 Job 对应的已完成的 Pods , 可以执行 kubectl get pods。

要以机器可读的方式列举隶属于某 Job 的全部 Pods , 你可以使用类似下面这条命令 :

```
pods=$(kubectl get pods --selector=job-name=pi --output=jsonpath='{.items[*].metadata.name}')
echo $pods
```

输出类似于 :

```
pi-5rwd7
```

这里，选择算符与 Job 的选择算符相同。--output=jsonpath 选项给出了一个表达式，用来从返回的列表中提取每个 Pod 的 name 字段。

查看其中一个 Pod 的标准输出：

```
kubectl logs $pods
```

输出类似于：

```
3.14159265358979323846264338327950288419716939937510582097494459230  
781640628620899862803482534211706798214808651328230664709384460955  
058223172535940812848111745028410270193852110555964462294895493038  
196442881097566593344612847564823378678316527120190914564856692346  
034861045432664821339360726024914127372458700660631558817488152092  
096282925409171536436789259036001133053054882046652138414695194151  
160943305727036575959195309218611738193261179310511854807446237996  
274956735188575272489122793818301194912983367336244065664308602139  
494639522473719070217986094370277053921717629317675238467481846766  
940513200056812714526356082778577134275778960917363717872146844090  
122495343014654958537105079227968925892354201995611212902196086403  
441815981362977477130996051870721134999999837297804995105973173281  
609631859502445945534690830264252230825334468503526193118817101000  
313783875288658753320838142061717766914730359825349042875546873115  
956286388235378759375195778185778053217122680661300192787661119590  
921642019893809525720106548586327886593615338182796823030195203530  
185296899577362259941389124972177528347913151557485724245415069595  
082953311686172785588907509838175463746493931925506040092770167113  
900984882401285836160356370766010471018194295559619894676783744944  
825537977472684710404753464620804668425906949129331367702898915210  
475216205696602405803815019351125338243003558764024749647326391419  
927260426992279678235478163600934172164121992458631503028618297455  
570674983850549458858692699569092721079750930295532116534498720275  
596023648066549911988183479775356636980742654252786255181841757467  
289097777279380008164706001614524919217321721477235014144197356854  
816136115735255213347574184946843852332390739414333454776241686251  
898356948556209921922218427255025425688767179049460165346680498862  
723279178608578438382796797668145410095388378636095068006422512520  
511739298489608412848862694560424196528502221066118630674427862203  
919494504712371378696095636437191728746776465757396241389086583264  
59958133904780275901
```

编写 Job 规约

与 Kubernetes 中其他资源的配置类似，Job 也需要 apiVersion、kind 和 metadata 字段。Job 的名字必须时合法的 [DNS 子域名](#)。

Job 配置还需要一个 [spec 节](#)。

Pod 模版

Job 的 .spec 中只有 .spec.template 是必需的字段。

字段 .spec.template 的值是一个 [Pod 模版](#)。其定义规范与 [Pod](#) 完全相同，只是其中不再需要 apiVersion 或 kind 字段。

除了作为 Pod 所必需的字段之外，Job 中的 Pod 模版必需设置合适的标签（参见[Pod 选择算符](#)）和合适的重启策略。

Job 中 Pod 的 [RestartPolicy](#) 只能设置为 Never 或 OnFailure 之一。

Pod 选择算符

字段 .spec.selector 是可选的。在绝大多数场合，你都不需要为其赋值。参阅[设置自己的 Pod 选择算符](#)。

Job 的并行执行

适合以 Job 形式来运行的任务主要有三种：

1. 非并行 Job

- 通常只启动一个 Pod，除非该 Pod 失败
- 当 Pod 成功终止时，立即视 Job 为完成状态

2. 具有 确定完成计数 的并行 Job

- .spec.completions 字段设置为非 0 的正数值
- Job 用来代表整个任务，当对应于 1 和 .spec.completions 之间的每个整数都存在一个成功的 Pod 时，Job 被视为完成
- 尚未实现：每个 Pod 收到一个介于 1 和 spec.completions 之间的不同索引值

3. 带 工作队列 的并行 Job

- 不设置 spec.completions，默认值为 .spec.parallelism
- 多个 Pod 之间必须相互协调，或者借助外部服务确定每个 Pod 要处理哪个工作条目。例如，任一 Pod 都可以从工作队列中取走最多 N 个工作条目。
- 每个 Pod 都可以独立确定是否其它 Pod 都已完成，进而确定 Job 是否完成
- 当 Job 中 任何 Pod 成功终止，不再创建新 Pod
- 一旦至少 1 个 Pod 成功完成，并且所有 Pod 都已终止，即可宣告 Job 成功完成
- 一旦任何 Pod 成功退出，任何其它 Pod 都不应再对此任务执行任何操作或生成任何输出。所有 Pod 都应启动退出过程。

对于 非并行 的 Job，你可以不设置 spec.completions 和 spec.parallelism。这两个属性都不设置时，均取默认值 1。

对于 确定完成计数 类型的 Job，你应该设置 .spec.completions 为所需要的完成个数。你可以设置 .spec.parallelism，也可以不设置。其默认值为 1。

对于一个 工作队列 Job，你不可以设置 .spec.completions，但要将.spec.parallelism 设置为一个非负整数。

关于如何利用不同类型的 Job 的更多信息，请参见 [Job 模式](#)一节。

控制并行性

并行性请求 (`.spec.parallelism`) 可以设置为任何非负整数。如果未设置，则默认为 1。如果设置为 0，则 Job 相当于启动之后便被暂停，直到此值被增加。

实际并行性（在任意时刻运行状态的 Pods 个数）可能比并行性请求略大或略小，原因如下：

- 对于 确定完成计数 Job，实际上并行执行的 Pods 个数不会超出剩余的完成数。如果 `.spec.parallelism` 值较高，会被忽略。
- 对于 工作队列 Job，有任何 Job 成功结束之后，不会有新的 Pod 启动。不过，剩下的 Pods 允许执行完毕。
- 如果 Job [控制器](#) 没有来得及作出响应，或者
- 如果 Job 控制器因为任何原因（例如，缺少 ResourceQuota 或者没有权限）无法创建 Pods。Pods 个数可能比请求的数目小。
- Job 控制器可能会因为之前同一 Job 中 Pod 失效次数过多而压制新 Pod 的创建。
- 当 Pod 处于体面终止进程中，需要一定时间才能停止。

处理 Pod 和容器失效

Pod 中的容器可能因为多种不同原因失效，例如因为其中的进程退出时返回值非零，或者容器因为超出内存约束而被杀死等等。如果发生这类事件，并且 `.spec.template.spec.restartPolicy = "OnFailure"`，Pod 则继续留在当前节点，但容器会被重新运行。因此，你的程序需要能够处理在本地被重启的情况，或者要设置 `.spec.template.spec.restartPolicy = "Never"`。关于 `restartPolicy` 的更多信息，可参阅 [Pod 生命周期](#)。

整个 Pod 也可能会失败，且原因各不相同。例如，当 Pod 启动时，节点失效（被升级、被重启、被删除等）或者其中的容器失败而 `.spec.template.spec.restartPolicy = "Never"`。当 Pod 失败时，Job 控制器会启动一个新的 Pod。这意味着，你的应用需要处理在一个新 Pod 中被重启的情况。尤其是应用需要处理之前运行所触碰或产生的临时文件、锁、不完整的输出等问题。

注意，即使你将 `.spec.parallelism` 设置为 1，且将 `.spec.completions` 设置为 1，并且 `.spec.template.spec.restartPolicy` 设置为 "Never"，同一程序仍然有可能被启动两次。

如果你确实将 `.spec.parallelism` 和 `.spec.completions` 都设置为比 1 大的值，那就有可能同时出现多个 Pod 运行的情况。为此，你的 Pod 也必须能够处理并发性问题。

Pod 回退失效策略

在有些情形下，你可能希望 Job 在经历若干次重试之后直接进入失败状态，因为这很可能意味着遇到了配置错误。为了实现这点，可以将 `.spec.backoffLimit` 设置为视 Job 为失败之前的重试次数。失效回退的限制值默认为 6。与 Job 相关的失效的 Pod 会被 Job 控制器重建，并且以指数型回退计算重试延迟（从 10 秒、20 秒到 40 秒，最多 6 分钟）。当 Job 的 Pod 被删除时，或者 Pod 成功时没有其它 Pod 处于失败状态，失效回退的次数也会被重置（为 0）。

说明：如果你的 Job 的 restartPolicy 被设置为 "OnFailure"，就要注意运行该 Job 的容器会在 Job 到达失效回退次数上限时自动被终止。这会使得调试 Job 中可执行文件的工作变得非常棘手。我们建议在调试 Job 时将 restartPolicy 设置为 "Never"，或者使用日志系统来确保失效 Jobs 的输出不会意外遗失。

Job 终止与清理

Job 完成时不会再创建新的 Pod，不过已有的 Pod 也不会被删除。保留这些 Pod 使得你可以查看已完成的 Pod 的日志输出，以便检查错误、警告或者其它诊断性输出。Job 完成时 Job 对象也一样被保留下来，这样你就可以查看它的状态。在查看了 Job 状态之后删除老的 Job 的操作留给了用户自己。你可以使用 kubectl 来删除 Job（例如，kubectl delete jobs/pi 或者 kubectl delete -f ./job.yaml）。当使用 kubectl 来删除 Job 时，该 Job 所创建的 Pods 也会被删除。

默认情况下，Job 会持续运行，除非某个 Pod 失败（restartPolicy=Never）或者某个容器出错退出（restartPolicy=OnFailure）。这时，Job 基于前述的 spec.backoffLimit 来决定是否以及如何重试。一旦重试次数到达 .spec.backoffLimit 所设的上限，Job 会被标记为失败，其中运行的 Pods 都会被终止。

终止 Job 的另一种方式是设置一个活跃期限。你可以为 Job 的 .spec.activeDeadlineSeconds 设置一个秒数值。该值适用于 Job 的整个生命期，无论 Job 创建了多少个 Pod。一旦 Job 运行时间达到 activeDeadlineSeconds 秒，其所有运行中的 Pod 都会被终止，并且 Job 的状态更新为 type: Failed 及 reason: DeadlineExceeded。

注意 Job 的 .spec.activeDeadlineSeconds 优先级高于其 .spec.backoffLimit 设置。因此，如果一个 Job 正在重试一个或多个失效的 Pod，该 Job 一旦到达 activeDeadlineSeconds 所设的时限即不再部署额外的 Pod，即使其重试次数还未达到 backoffLimit 所设的限制。

例如：

```
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
```

注意 Job 规约和 Job 中的 [Pod 模版规约](#) 都有 activeDeadlineSeconds 字段。请确保你在合适的层次设置正确的字段。

还要注意的是，restartPolicy 对应的是 Pod，而不是 Job 本身：一旦 Job 状态变为 type: Failed，就不会再发生 Job 重启的动作。换言之，由 .spec.activeDeadlineSeconds 和 .spec.backoffLimit 所触发的 Job 终结机制都会导致 Job 永久性的失败，而这类状态都需要手工干预才能解决。

自动清理完成的 Job

完成的 Job 通常不需要留存在系统中。在系统中一直保留它们会给 API 服务器带来额外的压力。如果 Job 由某种更高级别的控制器来管理，例如 [CronJobs](#)，则 Job 可以被 CronJob 基于特定的根据容量裁定的清理策略清理掉。

已完成 Job 的 TTL 机制

FEATURE STATE: Kubernetes v1.12 [alpha]

自动清理已完成 Job（状态为 Complete 或 Failed）的另一种方式是使用由 [TTL 控制器](#) 所提供的 TTL 机制。通过设置 Job 的 .spec.ttlSecondsAfterFinished 字段，可以让该控制器清理掉已结束的资源。

TTL 控制器清理 Job 时，会级联式地删除 Job 对象。换言之，它会删除所有依赖的对象，包括 Pod 及 Job 本身。注意，当 Job 被删除时，系统会考虑其生命周期保障，例如其 Finalizers。

例如：

```
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
```

Job pi-with-ttl 在结束 100 秒之后，可以成为被自动删除的标的。

如果该字段设置为 0，Job 在结束之后立即成为可被自动删除的对象。如果该字段没有设置，Job 不会在结束之后被 TTL 控制器自动清除。

注意这种 TTL 机制仍然是一种 Alpha 状态的功能特性，需要配合 TTLAfterFinished 特性门控使用。有关详细信息，可参考 [TTL 控制器](#) 的文档。

Job 模式

Job 对象可以用来支持多个 Pod 的可靠的并发执行。Job 对象不是设计用来支持相互通信的并行进程的，后者一般在科学计算中应用较多。Job 的确能够支持对一组相互独立而又有所关联的工作条目的并行处理。这类工作条目可能是要发送的电子邮件、要渲染的视频帧、要编解码的文件、NoSQL 数据库中要扫描的主键范围等等。

在一个复杂系统中，可能存在多个不同的工作条目集合。这里我们仅考虑用户希望一起管理的工作条目集合之一——批处理作业。

并行计算的模式有好多种，每种都有自己的强项和弱点。这里要权衡的因素有：

- 每个工作条目对应一个 Job 或者所有工作条目对应同一 Job 对象。后者更适合处理大量工作条目的场景；前者会给用户带来一些额外的负担，而且需要系统管理大量的 Job 对象。
- 创建与工作条目相等的 Pod 或者令每个 Pod 可以处理多个工作条目。前者通常不需要对现有代码和容器做较大改动；后者则更适合工作条目数量较大的场合，原因同上。
- 有几种技术都会用到工作队列。这意味着需要运行一个队列服务，并修改现有程序或容器使之能够利用该工作队列。与之比较，其他方案在修改现有容器化应用以适应需求方面可能更容易一些。

下面是对这些权衡的汇总，列 2 到 4 对应上面的权衡比较。模式的名称对应了相关示例和更详细描述的链接。

模式	单个 Job 对象	Pods 数少于工作条目数？	直接使用应用无需修改？	在 Kube 1.1 上可用？
Job 模版扩展			✓	✓
每工作条目一 Pod 的队列	✓		有时	✓
Pod 数量可变的队列	✓	✓		✓
静态工作分派的单个 Job	✓		✓	

当你使用 `.spec.completions` 来设置完成数时，Job 控制器所创建的每个 Pod 使用完全相同的 [spec](#)。这意味着任务的所有 Pod 都有相同的命令行，都使用相同的镜像和数据卷，甚至连环境变量都（几乎）相同。这些模式是让每个 Pod 执行不同工作的几种不同形式。

下表显示的是每种模式下 `.spec.parallelism` 和 `.spec.completions` 所需要的设置。其中，W 表示的是工作条目的个数。

模式	<code>.spec.completions</code>	<code>.spec.parallelism</code>
Job 模版扩展	1	应该为 1
每工作条目一 Pod 的队列	W	任意值

模式	.spec.completions	.spec.parallelism
Pod 个数可变的队列	1	任意值
基于静态工作分派的单一 Job	W	任意值

高级用法

指定你自己的 Pod 选择算符

通常，当你创建一个 Job 对象时，你不会设置 .spec.selector。系统的默认值填充逻辑会在创建 Job 时添加此字段。它会选择一个不会与任何其他 Job 重叠的选择算符设置。

不过，有些场合下，你可能需要重载这个自动设置的选择算符。为了实现这点，你可以手动设置 Job 的 spec.selector 字段。

做这个操作时请务必小心。如果你所设定的标签选择算符并不唯一针对 Job 对应的 Pod 集合，甚或该算符还能匹配 其他无关的 Pod，这些无关的 Job 的 Pod 可能会被删除。或者当前 Job 会将另外一些 Pod 当作是完成自身工作的 Pods，又或者两个 Job 之一或者二者同时都拒绝创建 Pod，无法运行至完成状态。如果所设置的算符不具有唯一性，其他控制器（如 RC 副本控制器）及其所管理的 Pod 集合可能会变得行为不可预测。Kubernetes 不会在你设置 .spec.selector 时尝试阻止你犯这类错误。

下面是一个示例场景，在这种场景下你可能会使用刚刚讲述的特性。

假定名为 old 的 Job 已经处于运行状态。你希望已有的 Pod 继续运行，但你希望 Job 接下来要创建的其他 Pod 使用一个不同的 Pod 模版，甚至希望 Job 的名字也发生变化。你无法更新现有的 Job，因为这些字段都是不可更新的。因此，你会删除 old Job，但 允许该 Job 的 Pod 集合继续运行。这是通过 kubectl delete jobs/old --cascade=false 实现的。在删除之前，我们先记下该 Job 所使用的算符。

```
kubectl get job old -o yaml
```

输出类似于：

```
kind: Job
metadata:
  name: old
...
spec:
  selector:
    matchLabels:
      controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
...
```

接下来你会创建名为 new 的新 Job，并显式地为其设置相同的选择算符。由于现有 Pod 都具有标签 controller-uid=a8f3d00d-c6d2-11e5-9f87-42010af00002，它们也会被名为 new 的 Job 所控制。

你需要在新 Job 中设置 manualSelector: true , 因为你并未使用系统通常自动为你生成的选择算符。

```
kind: Job
metadata:
  name: new
...
spec:
  manualSelector: true
  selector:
    matchLabels:
      controller-uid: a8f3d00d-c6d2-11e5-9f87-42010af00002
...

```

新的 Job 自身会有一个不同于 a8f3d00d-c6d2-11e5-9f87-42010af00002 的唯一 ID。设置 manualSelector: true 是在告诉系统你知道自己在干什么并要求系统允许这种不匹配的存在。

替代方案

裸 Pod

当 Pod 运行所在的节点重启或者失败，Pod 会被终止并且不会被重启。Job 会重新创建新的 Pod 来替代已经终止的 Pod。因为这个原因，我们建议你使用 Job 而不是独立的裸 Pod，即使你的应用仅需要一个 Pod。

副本控制器

Job 与[副本控制器](#)是彼此互补的。副本控制器管理的是那些不希望被终止的 Pod (例如，Web 服务器)，Job 管理的是那些希望被终止的 Pod (例如，批处理作业)。

正如在[Pod 生命期](#)中讨论的，Job 仅适合于 restartPolicy 设置为 OnFailure 或 Never 的 Pod。注意：如果 restartPolicy 未设置，其默认值是 Always。

单个 Job 启动控制器 Pod

另一种模式是用唯一的 Job 来创建 Pod，而该 Pod 负责启动其他 Pod，因此扮演了一种后启动 Pod 的控制器的角色。这种模式的灵活性更高，但是有时候可能会把事情搞得很复杂，很难入门，并且与 Kubernetes 的集成度很低。

这种模式的实例之一是用 Job 来启动一个运行脚本的 Pod，脚本负责启动 Spark 主控制器 (参见[Spark 示例](#))，运行 Spark 驱动，之后完成清理工作。

这种方法的优点之一是整个过程得到了 Job 对象的完成保障，同时维持了对创建哪些 Pod、如何向其分派工作的完全控制能力，

Cron Jobs

你可以使用 [CronJob](#) 创建一个在指定时间/日期运行的 Job，类似于 UNIX 系统上的 cron 工具。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 08, 2021 at 6:29 AM PST: [Ymal blocks miss 'yaml'](#) ([4a4aaaf866](#))

垃圾收集

Kubernetes 垃圾收集器的作用是删除某些曾经拥有属主（Owner）但现在不再拥有属主的对象。

属主和附属

某些 Kubernetes 对象是其它一些对象的属主。例如，一个 ReplicaSet 是一组 Pod 的属主。具有属主的对象被称为是属主的 **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 Kubernetes 1.8 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、Deployment、Job 和 CronJob 所创建或管理的。

你也可以通过手动设置 ownerReference 的值，来指定属主和附属之间的关系。

下面的配置文件中包含一个具有 3 个 Pod 的 ReplicaSet：

[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
```

如果你创建该 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 的属主是名为 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
...
...
```

说明：根据设计，kubernetes 不允许跨命名空间指定属主。这意味着：
1) 命名空间范围的附属只能指定同一的命名空间中的或者集群范围的属主。
2) 集群范围的附属只能指定集群范围的属主，不能指定命名空间范围的属主。

控制垃圾收集器删除附属

当你删除对象时，可以指定该对象的附属是否也自动删除。自动删除附属的行为也称为级联删除 (*Cascading Deletion*)。Kubernetes 中有两种 级联删除 模式：后台 (*Background*) 模式和 前台 (*Foreground*) 模式。

如果删除对象时，不自动删除它的附属，这些附属被称作 孤立对象 (*Orphaned*)。

前台级联删除

在 前台级联删除 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。
- 对象的 deletionTimestamp 字段被设置。
- 对象的 metadata.finalizers 字段包含值 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有阻塞能力的附属（对象的 ownerReference.blockOwnerDeletion=true）之后，删除属主对象。

注意，在 foregroundDeletion 模式下，只有设置了 ownerReference.blockOwnerDeletion 值的附属才能阻止删除属主对象。在 Kubernetes 1.7 版本增加了 [准入控制器](#)，基于属主对象上的删除权限来控制用户设置 blockOwnerDeletion 的值为 True，这样未经授权的附属不能够阻止属主对象的删除。

如果一个对象的 ownerReferences 字段被一个控制器（例如 Deployment 或 ReplicaSet）设置，blockOwnerDeletion 也会被自动设置，你不需要手动修改这个字段。

后台级联删除

在 后台级联删除 模式下，Kubernetes 会立即删除属主对象，之后垃圾收集器会在后台删除其附属对象。

设置级联删除策略

通过为属主对象设置 deleteOptions.propagationPolicy 字段，可以控制级联删除策略。可能的取值包括：Orphan、Foreground 或者 Background。

下面是一个在后台删除附属对象的示例：

```
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"
```

下面是一个在前台中删除附属对象的示例：

```
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"
```

下面是一个令附属成为孤立对象的示例：

```
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 命令也支持级联删除。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立附属对象。`--cascade` 的默认值是 `true`。

下面是一个例子，使一个 ReplicaSet 的附属对象成为孤立附属：

```
kubectl delete replicaset my-repset --cascade=false
```

Deployment 的附加说明

在 1.7 之前的版本中，当在 Deployment 中使用级联删除时，你 必须使用 `propagationPolicy:Foreground` 模式以便在删除所创建的 ReplicaSet 的同时，还删除其 Pod。如果不使用这种类型的 `propagationPolicy`，将只删除 ReplicaSet，而 Pod 被孤立。

有关信息请参考 [kubeadm/#149](#)。

已知的问题

跟踪 [#26120](#)

接下来

- [设计文档 1](#)
- [设计文档 2](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 16, 2020 at 12:56 PM PST: [\[zh\] Sync changes from English site \(6\) \(86ca2a9c1\)](#)

已完成资源的 TTL 控制器

FEATURE STATE: Kubernetes v1.12 [alpha]

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 [Job](#)，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，并且可以通过 kube-apiserver 和 kube-controller-manager 上的 [特性门控](#) TTLAfterFinished 启用。

TTL 控制器

TTL 控制器现在只支持 Job。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（Complete 或 Failed），如 [示例](#) 所示。

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，它将做级联删除操作，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 Job 在完成后的某个时间被自动清除。
- 将此字段设置为现有的、已完成的资源，以采用此新功能。
- 在创建资源时使用 [mutating admission webhook](#) 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](#) 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，在创建资源或已经执行结束后，仍可以修改其 TTL 周期，例如 Job 的 .spec.ttlSecondsAfterFinished 字段。但是一旦 Job 变为可被删除状态（当其 TTL 已过期时），即使您通过 API 增加其 TTL 时长得到了成功的响应，系统也不保证 Job 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](#)）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 时请注意避免这种风险。

接下来

- [自动清理 Job](#)
- [设计文档](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 August 12, 2020 at 7:26 PM PST: [\[zh\] fix links in concept section \(11cc9e007\)](#)

CronJob

FEATURE STATE: Kubernetes v1.8 [beta]

Cron Job 创建基于时间调度的 [Jobs](#).

一个 CronJob 对象就像 *crontab* (cron table) 文件中的一行。它用 [Cron](#) 格式进行编写，并周期性地在给定的调度时间执行 Job。

注意：

所有 **CronJob** 的 `schedule`: 时间都是基于 [kube-controller-manager](#). 的时区。

如果你的控制平面在 Pod 或是裸容器中运行了 `kube-controller-manager`，那么为该容器所设置的时区将会决定 Cron Job 的控制器所使用的时区。

为 CronJob 资源创建清单时，请确保所提供的名称是一个合法的 [DNS 子域名](#). 名称不能超过 52 个字符。这是因为 CronJob 控制器将自动在提供的 Job 名称后附加 11 个字符，并且存在一个限制，即 Job 名称的最大长度不能超过 63 个字符。

CronJob

CronJobs 对于创建周期性的、反复重复的任务很有用，例如执行数据备份或者发送邮件。CronJobs 也可以用来计划在指定时间来执行的独立任务，例如计划当集群看起来很空闲时 执行某个 Job。

示例

下面的 CronJob 示例清单会在每分钟打印出当前时间和问候消息：

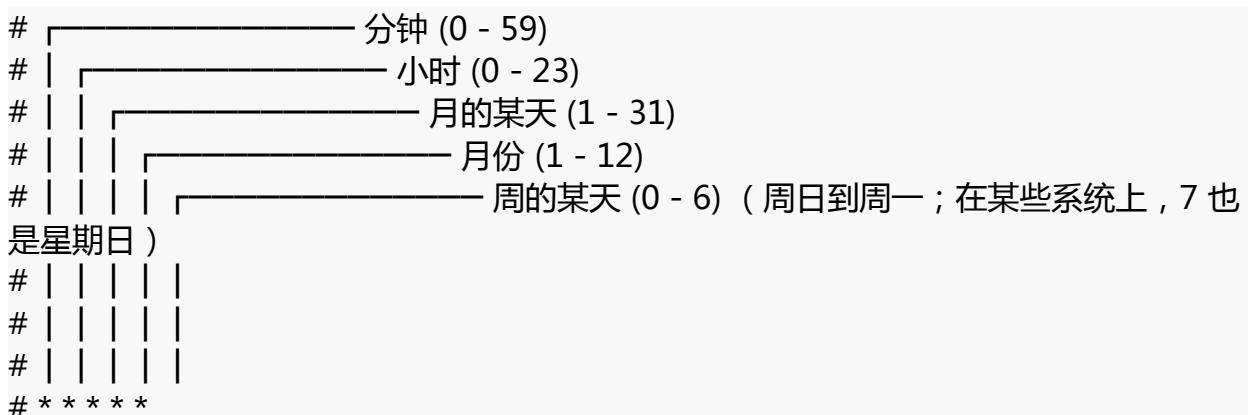
application/job/cronjob.yaml



```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              imagePullPolicy: IfNotPresent
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
        restartPolicy: OnFailure
```

[使用 CronJob 运行自动化任务](#) 一文会为你详细讲解此例。

Cron 时间表语法



输入	描述	相当于
@yearly (or @annually)	每年 1 月 1 日的午夜运行一次	0 0 1 1 *
@monthly	每月第一天的午夜运行一次	0 0 1 **
@weekly	每周的周日午夜运行一次	0 0 *** 0

输入	描述	相当于
@daily (or @midnight)	每天午夜运行一次	0 0 * * *
@hourly	每小时的开始一次	0 * * * *

例如，下面这行指出必须在每个星期五的午夜以及每个月 13 号的午夜开始任务：

0 0 13 * 5

要生成 CronJob 时间表表达式，你还可以使用 [crontab.guru](#) 之类的 Web 工具。

CronJob 限制

CronJob 根据其计划编排，在每次该执行任务的时候大约会创建一个 Job。我们之所以说“大约”，是因为在某些情况下，可能会创建两个 Job，或者不会创建任何 Job。我们试图使这些情况尽量少发生，但不能完全杜绝。因此，Job 应该是 幂等 的。

如果 startingDeadlineSeconds 设置为很大的数值或未设置（默认），并且 concurrencyPolicy 设置为 Allow，则作业将始终至少运行一次。

对于每个 CronJob，CronJob 控制器（Controller）检查从上一次调度的时间点到现在所错过了调度次数。如果错过的调度次数超过 100 次，那么它就不会启动这个任务，并记录这个错误：

```
Cannot determine if job needs to be started. Too many missed start time (> 100).  
Set or decrease .spec.startingDeadlineSeconds or check clock skew.
```

需要注意的是，如果 startingDeadlineSeconds 字段非空，则控制器会统计从 startingDeadlineSeconds 设置的值到现在而不是从上一个计划时间到现在错过了多少次 Job。例如，如果 startingDeadlineSeconds 是 200，则控制器会统计在过去 200 秒中错过了多少次 Job。

如果未能在调度时间内创建 CronJob，则计为错过。例如，如果 concurrencyPolicy 被设置为 Forbid，并且当前有一个调度仍在运行的情况下，试图调度的 CronJob 将被计算为错过。

例如，假设一个 CronJob 被设置为从 08:30:00 开始每隔一分钟创建一个新的 Job，并且它的 startingDeadlineSeconds 字段未被设置。如果 CronJob 控制器从 08:29:00 到 10:21:00 终止运行，则该 Job 将不会启动，因为其错过的调度次数超过了 100。

为了进一步阐述这个概念，假设将 CronJob 设置为从 08:30:00 开始每隔一分钟创建一个新的 Job，并将其 startingDeadlineSeconds 字段设置为 200 秒。如果 CronJob 控制器恰好在与上一个示例相同的时间段（08:29:00 到 10:21:00）终止运行，则 Job 仍将从 10:22:00 开始。造成这种情况的原因是控制器现在检查在最近 200 秒（即 3 个错过的调度）中发生了多少次错过的 Job 调度，而不是从现在为止的最后一个调度时间开始。

CronJob 仅负责创建与其调度时间相匹配的 Job，而 Job 又负责管理其代表的 Pod。

新控制器

CronJob 控制器有一个替代的实现，自 Kubernetes 1.20 开始以 alpha 特性引入。如果选择 CronJob 控制器的 v2 版本，请在 [kube-controller-manager](#) 中设置以下[特性门控](#) 标志。

```
--feature-gates="CronJobControllerV2=true"
```

接下来

- 进一步了解 [Cron 表达式的格式](#)，学习设置 CronJob schedule 字段
- 有关创建和使用 CronJob 的说明及示例规约文件，请参见 [使用 CronJob 运行自动化任务](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 25, 2020 at 5:10 PM PST: [zh-trans: update CronJob \(540d43d3b\)](#)

ReplicationController

说明：现在推荐使用配置 [ReplicaSet](#) 的 [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

当 Pod 数量过多时，ReplicationController 会终止多余的 Pod。当 Pod 数量太少时，ReplicationController 将会启动新的 Pod。与手动创建的 Pod 不同，由 ReplicationController 创建的 Pod 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，你的 Pod 会在节点上重新创建。因此，即使你的应用程序只需要一个 Pod，你也应该使用 ReplicationController 创建 Pod。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 Pod。

在讨论中，ReplicationController 通常缩写为 "rc"，并作为 kubectl 命令的快捷方式。

一个简单的示例是创建一个 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](#)



```
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				

在这里，创建了三个 Pod，但没有一个 Pod 正在运行，这可能是因为正在拉取镜像。稍后，相同的命令可能会显示：

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，选择算符与 ReplicationController 的选择算符相同（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](#)。

ReplicationController 也需要一个 [spec 部分](#)。

Pod 模板

.spec.template 是 .spec 的唯一必需字段。

.spec.template 是一个 [Pod 模板](#)。它的模式与 [Pod](#) 完全相同，只是它是嵌套的，没有 apiVersion 或 kind 属性。

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 选择算符](#)。

只允许 [.spec.template.spec.restartPolicy](#) 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](#) 或 Docker。

ReplicationController 上的标签

ReplicationController 本身可以有标签（.metadata.labels）。通常，你可以将这些设置为 .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 选择算符

.spec.selector 字段是一个 [标签选择算符](#)。ReplicationController 管理标签与选择算符匹配的所有 Pod。它不区分它创建或删除的 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。

另外，通常不应直接使用另一个 ReplicationController 或另一个控制器（例如 Job）来创建其标签与该选择算符匹配的任何 Pod。如果这样做，ReplicationController 会认为它创建了这些 Pod。Kubernetes 并没有阻止你这样做。

如果你的确创建了多个控制器并且其选择算符之间存在重叠，那么你将不得不自己管理删除操作（参考[后文](#)）。

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 Pod。在任何时候，处于运行状态的 Pod 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 处于优雅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](#)。 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，你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](#) 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 Pod。 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 Pod 以使用新的 spec，请执行[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，无论你想要继续运行 1 个 Pod 还是 1000 个 Pod，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。

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](#) PR 中所述，建议的方法是使用 1 个副本创建一个新的 ReplicationController，逐个扩容新的 (+1) 和缩容旧的 (-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 Pod 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 Pod 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](#) 中实现的。访问 [kubectl rolling-update 任务](#) 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。现在假设你有 10 个副本的 Pod 组成了这个层。但是你希望能够 canary (金丝雀) 发布这个组件的新版本。你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。但你可以单独处理 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时间存活。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](#)，而不是静态的或者一次性定制每个 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 数量与其标签选择算符匹配，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](#)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](#) 中所述），后者负责更改其 replicas 字段值。我们不会向

ReplicationController 添加调度策略（例如，[spreading](#)）。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（查阅[#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 或者工具，以便于将来的用户使用。kubectl 目前支持的“macro”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](#) 的东西管理 ReplicationController、自动定标器、服务、调度策略、金丝雀发布等。

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](#)找到。

ReplicationController 的替代方案

ReplicaSet

[ReplicaSet](#) 是下一代 ReplicationController，支持新的[基于集合的标签选择算符](#)。它主要被[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，除非你需要自定义更新编排或根本不需要更新。

Deployment (推荐)

[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。如果你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 Pod

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，我们建议你使用 ReplicationController，即使你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](#) 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](#) 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或者关闭时安全地终止。

更多信息

请阅读[运行无状态的 ReplicationController](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 16, 2020 at 12:56 PM PST: [\[zh\] Sync changes from English site \(6\) \(86ca2a9c1\)](#)

服务、负载均衡和联网

Kubernetes 网络背后的概念和资源。

Kubernetes 网络解决四方面的问题：

- 一个 Pod 中的容器之间通过本地回路 (loopback) 通信。
- 集群网络在不同 pod 之间提供通信。
- Service 资源允许你对外暴露 Pods 中运行的应用程序，以支持来自于集群外部的访问。
- 可以使用 Services 来发布仅供集群内部使用的服务。

服务

[服务拓扑 \(Service Topology \)](#)

[Pod 与 Service 的 DNS](#)

[使用 Service 连接到应用](#)

[端点切片 \(Endpoint Slices \)](#)

[Ingress](#)

[Ingress 控制器](#)

[网络策略](#)

[使用 HostAliases 向 Pod /etc/hosts 文件添加条目](#)

[IPv4/IPv6 双协议栈](#)

服务

将运行在一组 [Pods](#) 上的应用程序公开为网络服务的抽象方法。

使用 Kubernetes，你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，并为一组 Pod 提供相同的 DNS 名，并且可以在它们之间进行负载均衡。

动机

创建和销毁 Kubernetes [Pod](#) 以匹配集群状态。Pod 是非永久性资源。如果你使用 [Deployment](#) 来运行你的应用程序，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”）为集群内的其他 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。Service 所针对的 Pods 集合通常是通过[选择算符](#)来确定的。要了解定义服务端点的其他方法，请参阅[不带选择算符的服务](#)。

举个例子，考虑一个图片处理后端，它运行了 3 个副本。这些副本是可互换的——前端不需要关心它们调用了哪个后端副本。然而组成这一组后端程序的 Pod 实际上可能会发生变化，前端客户端不应该也没必要知道，而且也不需要跟踪这一组后端的状态。

Service 定义的抽象能够解耦这种关联。

云原生服务发现

如果你想要在应用程序中使用 Kubernetes API 进行服务发现，则可以查询 [API 服务器](#) 的 Endpoints 资源，只要服务中的 Pod 集合发生更改，Endpoints 就会被更新。

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均衡器的方法。

定义 Service

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。Service 对象的名称必须是合法的 [DNS 标签名称](#)。

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 [VIP 和 Service 代理](#)）。

服务选择算符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。

说明：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 中的端口定义是有名字的，你可以在服务的 targetPort 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，你可以更改 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；你还可以使用任何其他[受支持的协议](#)。

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。

没有选择算符的 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 [名字空间 \(Namespace\)](#) 中或其它集群中的服务。
- 你正在将工作负载迁移到 Kubernetes。在评估该方法时，你仅在 Kubernetes 中运行一部分后端。

在任何这些场景中，都能够定义没有选择算符的 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算符，因此 不会自动创建相应的 Endpoint 对象。你可以通过手动添加 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
```

Endpoints 对象的名称必须是合法的 [DNS 子域名](#)。

说明：

端点 IPs 必须不可以是：本地回路（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 服务的集群 IP，因为 [kube-proxy](#) 不支持将虚拟 IP 作为目标。

访问没有选择算符的 Service，与有选择算符的 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，它没有选择算符，但是使用 DNS 名称。有关更多信息，请参阅本文档后面的[ExternalName](#)。

EndpointSlice

FEATURE STATE: Kubernetes v1.17 [beta]

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 [EndpointSlices](#) 中有详细描述。

应用程序协议

FEATURE STATE: Kubernetes v1.20 [stable]

appProtocol 字段提供了一种为每个 Service 端口指定应用协议的方式。此字段的取值会被映射到对应的 Endpoints 和 EndpointSlices 对象。

该字段遵循标准的 Kubernetes 标签语法。其值可以是 [IANA 标准服务名称](#) 或以域名前缀的名称，如 mycompany.com/my-custom-protocol。

虚拟 IP 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](#) 的形式。

为什么不使用 DNS 轮询？

时不时会有人问到为什么 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：

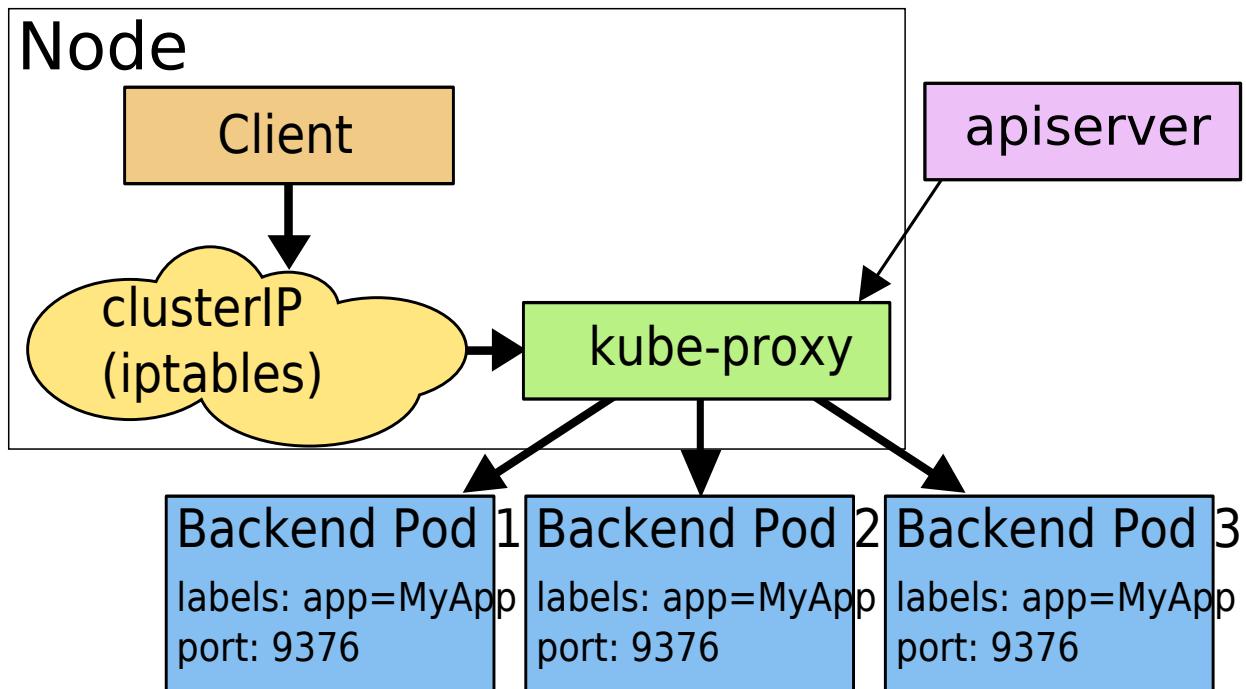
- DNS 实现的历史由来已久，它不遵守记录 TTL，并且在名称查找结果到期后对其进行缓存。
- 有些应用程序仅执行一次 DNS 查找，并无限期地缓存结果。
- 即使应用和库进行了适当的重新解析，DNS 记录上的 TTL 值低或为零也可能会给 DNS 带来高负载，从而使管理变得困难。

userspace 代理模式

这种模式，kube-proxy 会监视 Kubernetes 主控节点对 Service 对象和 Endpoints 对象的添加和移除操作。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 的后端 Pods 中的某个上面（如 Endpoints 所报告的一样）。使用哪个后端 Pod，是 kube-proxy 基于 Session Affinity 来确定的。

最后，它配置 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后端 Pod。

默认情况下，用户空间模式下的 kube-proxy 通过轮转算法选择后端。



iptables 代理模式

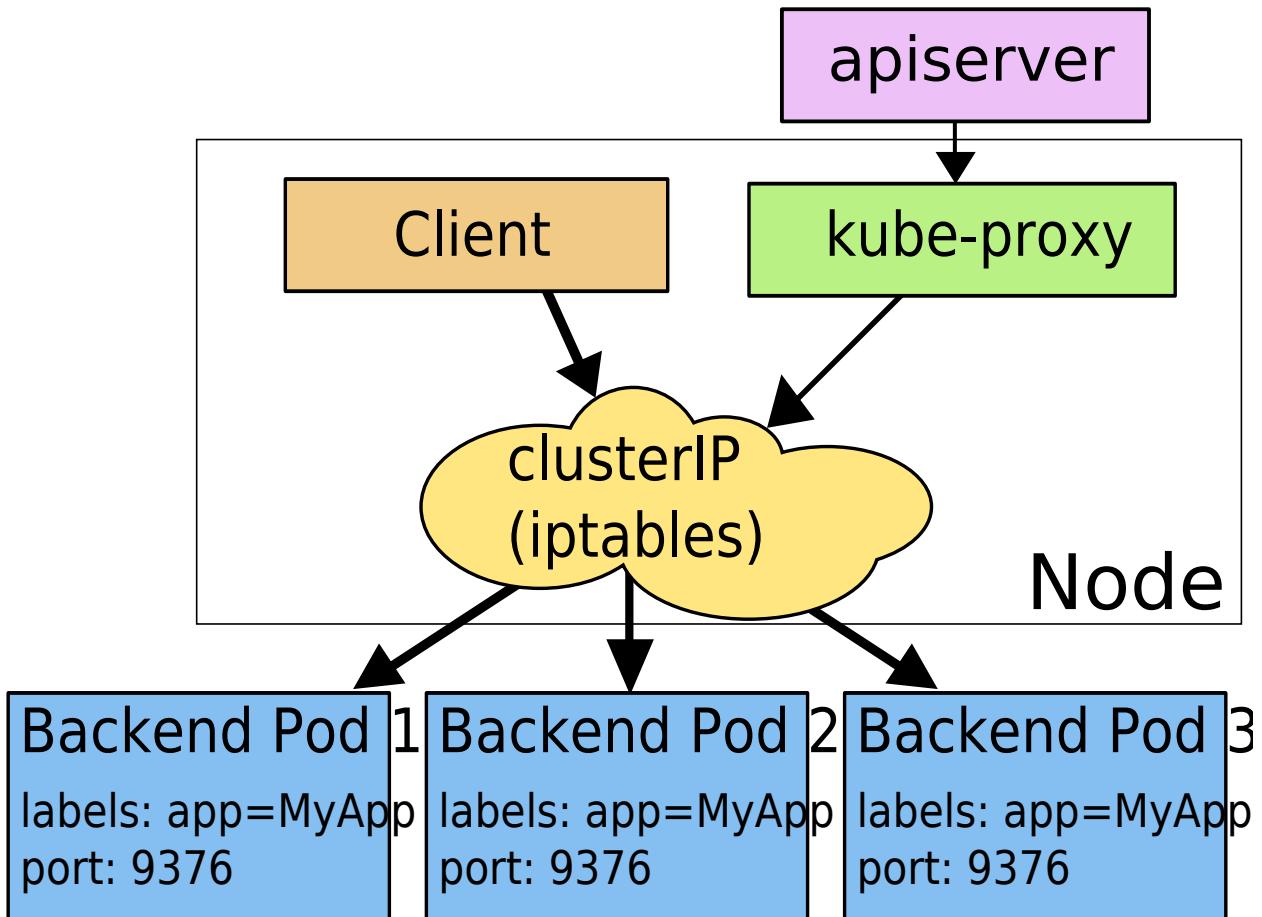
这种模式，kube-proxy 会监视 Kubernetes 控制节点对 Service 对象和 Endpoints 对象的添加和移除。对每个 Service，它会配置 iptables 规则，从而捕获到达该 Service 的 clusterIP 和端口的请求，进而将请求重定向到 Service 的一组后端中的某个 Pod 上面。对于每个 Endpoints 对象，它也会配置 iptables 规则，这个规则会选择一个后端组合。

默认的策略是，kube-proxy 在 iptables 模式下随机选择一个后端。

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 iptables 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。

你可以使用 Pod [就绪探测器](#) 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。这样做意味着你避免将流量通过 kube-proxy 发送到已知已失败的 Pod。



IPVS 代理模式

FEATURE STATE: Kubernetes v1.11 [stable]

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：

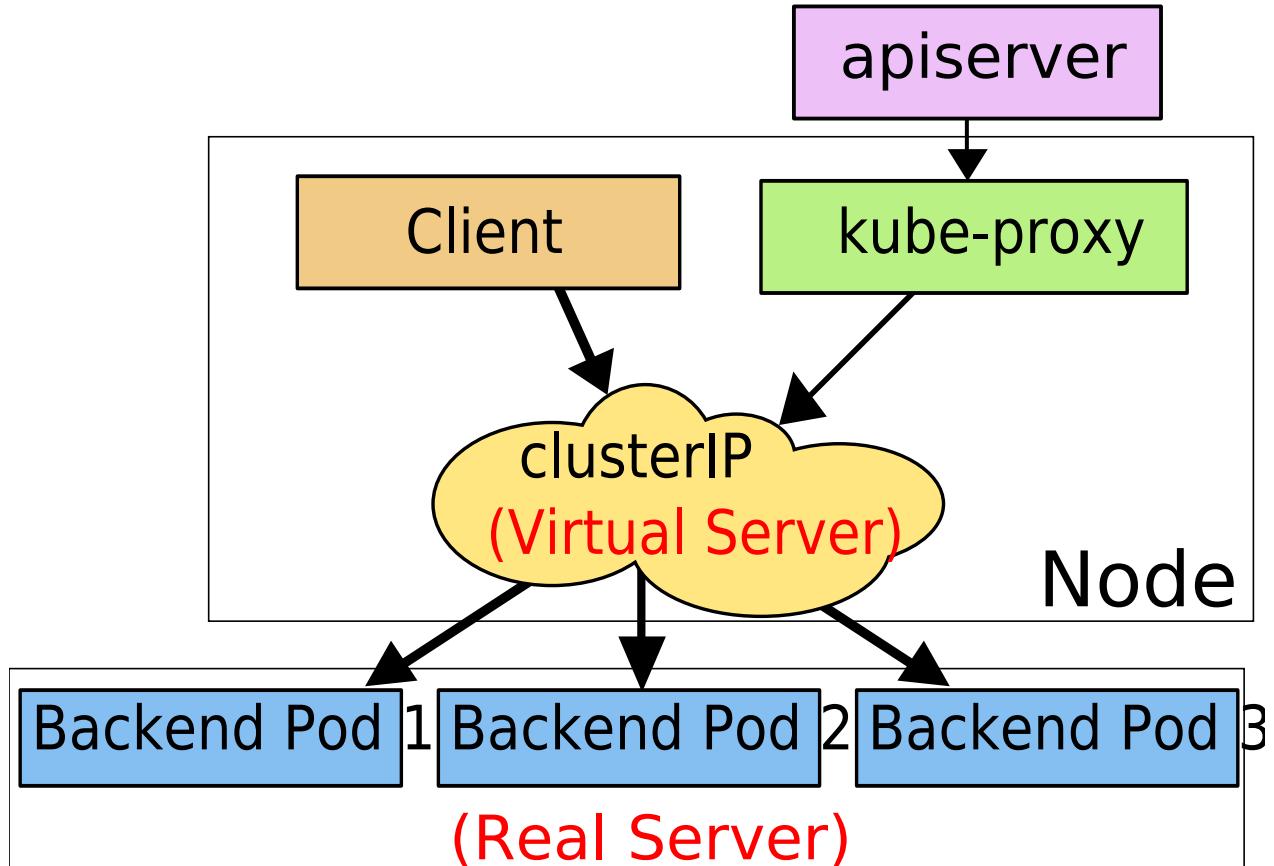
- rr : 轮替 (Round-Robin)
- lc : 最少链接 (Least Connection)，即打开链接数量最少者优先
- dh : 目标地址哈希 (Destination Hashing)
- sh : 源地址哈希 (Source Hashing)
- sed : 最短预期延迟 (Shortest Expected Delay)

- nq : 从不排队 (Never Queue)

说明 :

要在 IPVS 模式下运行 kube-proxy , 必须在启动 kube-proxy 之前使 IPVS 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。你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 , 你需要公开多个端口。Kubernetes 允许你在 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
```

说明：

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 服务器来说是通过一个标识来指定的。如果 IP 地址不合法，API 服务器会返回 HTTP 状态码 422，表示值不合法。

服务发现

Kubernetes 支持两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](#) 变量（查看 [makeLinkVariables](#)）、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
```

说明：

当你具有需要访问服务的 Pod 时，并且你正在使用环境变量方法将端口和集群 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 查找服务的集群 IP，则无需担心此设定问题。

DNS

你可以（几乎总是应该）使用[附加组件](#)为 Kubernetes 集群设置 DNS 服务。

支持集群的 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。如果在整个集群中都启用了 DNS，则所有 Pod 都应该能够通过其 DNS 名称自动解析服务。

例如，如果你在 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按名检索 my-service 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集群 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](#)。

无头服务（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。

你可以使用无头 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。

对这无头 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 是否定义了选择算符。

带选择算符的服务

对定义了选择算符的无头服务 , 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。

无选择算符的服务

对没有定义选择算符的无头服务 , 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 :

- 对于 [ExternalName](#) 类型的服务 , 查找其 CNAME 记录
- 对所有其他类型的服务 , 查找与 Service 名称相同的任何 Endpoints 的记录

发布服务 (服务类型)

对一些应用的某些部分 (如前端) , 可能希望将其暴露给 Kubernetes 集群外部的 IP 地址。

Kubernetes ServiceTypes 允许指定你所需要的 Service 类型 , 默认是 ClusterIP 。

Type 的取值以及行为如下 :

- ClusterIP : 通过集群的内部 IP 暴露服务 , 选择该值时服务只能够在集群内部访问。 这也是默认的 ServiceType 。
- [NodePort](#) : 通过每个节点上的 IP 和静态端口 (NodePort) 暴露服务。 NodePort 服务会路由到自动创建的 ClusterIP 服务。 通过请求 < 节点 IP > : < 节点端口 > , 你可以从集群的外部访问一个 NodePort 服务。
- [LoadBalancer](#) : 使用云提供商的负载均衡器向外部暴露服务。 外部负载均衡器可以将流量路由到自动创建的 NodePort 服务和 ClusterIP 服务上。
- [ExternalName](#) : 通过返回 CNAME 和对应值 , 可以将服务映射到 externalName 字段的内容 (例如 , foo.bar.example.com) 。 无需创建任何类型代理。

说明 : 你需要使用 kube-dns 1.7 及以上版本或者 CoreDNS 0.0.8 及以上版本才能使用 ExternalName 类型。

你也可以使用 [Ingress](#) 来暴露自己的服务。 Ingress 不是一种服务类型 , 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

如果你将 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）代理到你的服务中。你的服务在其 .spec.ports[*].nodePort 字段中要求分配的端口。

如果你想指定特定的 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，如果你使用 --nodeport-addresses=127.0.0.0/8 标志启动 kube-proxy，则 kube-proxy 仅选择 NodePort Services 的本地回路接口。--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，你可以在 nodePort 字段中指定一个值。控制平面将为你分配该端口或报告 API 事务失败。这意味着你需要自己注意可能发生的端口冲突。你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 可以让你自由设置自己的负载均衡解决方案，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（如果 kube-proxy 的 --nodeport-addresses 参数被设置了，将被过滤 NodeIP。）。

例如：

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: MyApp
  ports:
    # 默认情况下，为了方便起见，`targetPort` 被设置为与 `port` 字段相同的值。
    - port: 80
      targetPort: 80
      # 可选字段
      # 默认情况下，为了方便起见，Kubernetes 控制平面会从某个范围内分配一个端口号（默认：30000-32767）
      nodePort: 30007
```

LoadBalancer 类型

在使用支持外部负载均衡器的云提供商的服务时，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
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
      - ip: 192.0.2.127
```

来自外部负载均衡器的流量将直接重定向到后端 Pod 上，不过实际它们是如何工作的，这要依赖于云提供商。

某些云提供商允许设置 loadBalancerIP。在这些情况下，将根据用户设置的 loadBalancerIP 来创建负载均衡器。如果没有设置 loadBalancerIP 字段，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。

说明：

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 地址资源应与集群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 地址设置为 loadBalancerIP。确保你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](#) 或在 AKS 集群上使用高级联网时出现 [CreatingLoadBalancerFailed](#)。

混合协议类型的负载均衡器

FEATURE STATE: Kubernetes v1.20 [alpha]

默认情况下，对于 LoadBalancer 类型的服务，当定义了多个端口时，所有 端口必须具有相同的协议，并且该协议必须是受云提供商支持的协议。

如果为 kube-apiserver 启用了 MixedProtocolLBService 特性门控，则当定义了多个端口时，允许使用不同的协议。

说明：可用于 LoadBalancer 类型服务的协议集仍然由云提供商决定。

禁用负载均衡器节点端口分配

FEATURE STATE: Kubernetes v1.20 [alpha]

从 v1.20 版本开始，你可以通过设置 spec.allocateLoadBalancerNodePorts 为 false 对类型为 LoadBalancer 的服务禁用节点端口分配。这仅适用于直接将流量路由到 Pod 而不是使用节点端口的负载均衡器实现。默认情况下，spec.allocateLoadBalancerNodePorts 为 true，LoadBalancer 类型的服务继续分配节点端口。如果现有服务已被分配节点端口，将参数 spec.allocateLoadBalancerNodePorts 设置为 false 时，这些服务上已分配的节点端口不会被自动释放。你必须显式地在每个服务端口中删除 nodePorts 项以释放对应端口。你必须启用 ServiceLBNodePortControl 特性门控才能使用该字段。

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，你需要两个服务才能将内部和外部流量都路由到你的端点 (Endpoints)。

如要设置内部负载均衡器，请根据你所使用的云运营商，为服务添加以下注解之一。

- [Default](#)
- [GCP](#)
- [AWS](#)
- [Azure](#)
- [IBM Cloud](#)
- [OpenStack](#)
- [Baidu Cloud](#)
- [Tencent Cloud](#)
- [Alibaba Cloud](#)

选择一个标签

[...]

metadata:

name: my-service

annotations:

```
cloud.google.com/load-balancer-type: "Internal"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  name: my-service
```

```
  annotations:
```

```
    service.beta.kubernetes.io/aws-load-balancer-internal: "true"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  name: my-service
```

```
  annotations:
```

```
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  name: my-service
```

```
  annotations:
```

```
    service.kubernetes.io/ibm-load-balancer-cloud-provider-ip-type: "private"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  name: my-service
```

```
  annotations:
```

```
    service.beta.kubernetes.io/openstack-internal-load-balancer: "true"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  name: my-service
```

```
  annotations:
```

```
    service.beta.kubernetes.io/cce-load-balancer-internal-vpc: "true"
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  annotations:
```

```
    service.kubernetes.io/qcloud-loadbalancer-internal-subnetid: subnet-xxxxxx
```

```
[...]
```

```
[...]
```

```
metadata:
```

```
  annotations:
```

```
service.beta.kubernetes.io/alibaba-cloud-loadbalancer-address-type: "intranet"
[...]
```

AWS TLS 支持

为了对在 AWS 上运行的集群提供 TLS/SSL 部分支持，你可以向 LoadBalancer 服务添加三个注解：

```
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)
```

第二个注解指定 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，可以使用以下注解：

```
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"
```

在上例中，如果服务包含 80、443 和 8443 三个端口，那么 443 和 8443 将使用 SSL 证书，而 80 端口将仅仅转发 HTTP 数据包。

从 Kubernetes v1.9 起可以使用 [预定义的 AWS SSL 策略](#) 为你的服务使用 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，你可以使用 "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](#)。你可以使用以下服务注解：

```
metadata:  
  name: my-service  
  annotations:  
    service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
```

从 1.3.0 版开始，此注解的使用适用于 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

有几个注解可用于管理 AWS 上 ELB 服务的访问日志。

注解 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）。你可以指定 5 分钟或 60 分钟的间隔。

注解 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。

注解 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"  
    # 指定是否为负载均衡器启用访问日志  
    service.beta.kubernetes.io/aws-load-balancer-access-log-emit-interval: "60"  
    # 发布访问日志的时间间隔。你可以将其设置为 5 分钟或 60 分钟。  
    service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-name: "my-bucket"  
    # 用来存放访问日志的 Amazon S3 Bucket 名称  
    service.beta.kubernetes.io/aws-load-balancer-access-log-s3-bucket-prefix: "my-bucket-prefix/prod"
```

```
# 你为 Amazon S3 Bucket 所创建的逻辑层次结构，例如 `my-bucket-prefix/prod`
```

AWS 上的连接排空

可以将注解 `service.beta.kubernetes.io/aws-load-balancer-connection-draining-enabled` 设置为 "true" 来管理 ELB 的连接排空。注解 `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 注解

还有其他一些注解，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"  
      # 按秒计的时间，表示负载均衡器关闭连接之前连接可以保持空闲  
      # (连接上无数据传输) 的时间长度  
  
    service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled: "true"  
      # 指定该负载均衡器上是否启用跨区的负载均衡能力  
  
    service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags: "environment=prod,owner=devops"  
      # 逗号分隔列表值，每一项都是一个键-值对，会作为额外的标签记录于 ELB 中  
  
    service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold: ""  
      # 将某后端视为健康、可接收请求之前需要达到的连续成功健康检查次数。  
      # 默认为 2，必须介于 2 和 10 之间  
  
    service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold: "3"  
      # 将某后端视为不健康、不可接收请求之前需要达到的连续不成功健康检查次数。  
      # 默认为 6，必须介于 2 和 10 之间
```

```

service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "20"
# 对每个实例进行健康检查时，连续两次检查之间的大致间隔秒数
# 默认为 10，必须介于 5 和 300 之间

service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout: "5"
# 时长秒数，在此期间没有响应意味着健康检查失败
# 此值必须小于 service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval
# 默认值为 5，必须介于 2 和 60 之间

service.beta.kubernetes.io/aws-load-balancer-security-groups: "sg-53fae93f"
# 要添加到所创建的 ELB 之上的已有安全组列表。与注解
# service.beta.kubernetes.io/aws-load-balancer-extra-security-groups 不同，  

此
# 注解会替换掉之前指派给 ELB 的所有其他安全组

service.beta.kubernetes.io/aws-load-balancer-extra-security-groups: "sg-53fae93f,sg-42efd82e"
# 要添加到 ELB 上的额外安全组列表

service.beta.kubernetes.io/aws-load-balancer-target-node-labels: "ingress-gw,gw-name=public-api"
# 用逗号分隔的一个键-值偶对列表，用来为负载均衡器选择目标节点

```

AWS 上网络负载均衡器支持

FEATURE STATE: Kubernetes v1.15 [beta]

要在 AWS 上使用网络负载均衡器，可以使用注解 `service.beta.kubernetes.io/aws-load-balancer-type`，将其取值设为 `nlb`。

```

metadata:
  name: my-service
annotations:
  service.beta.kubernetes.io/aws-load-balancer-type: "nlb"

```

说明： 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 [AWS 文档](#) 中关于所支持的实例类型的 Elastic Load Balancing 说明。

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 地址不会传达到最终的 Pod。

通过将 `.spec.externalTrafficPolicy` 设置为 `Local`，客户端 IP 地址将传播到最终的 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。

为了获得均衡流量，请使用 DaemonSet 或指定 [Pod 反亲和性](#) 使其不在同一节点上。

你还可以将 NLB 服务与[内部负载平衡器](#)注解一起使用。

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)	Ip
Health Check	TCP	NodePort(s) (.spec.healthCheckNodePort for .spec.externalTrafficPolicy=Local)	VPC CIDR	ku he
Client Traffic	TCP	NodePort(s)	.spec.loadBalancerSourceRanges (defaults to 0.0.0.0/0)	ku cli
MTU Discovery	ICMP	3,4	.spec.loadBalancerSourceRanges (defaults to 0.0.0.0/0)	ku mi

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"
```

说明：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。

腾讯 Kubernetes 引擎 (TKE) 上的 CLB 注解

以下是在 TKE 上管理云负载均衡器的注解。

```
metadata:  
  name: my-service  
  annotations:  
    # 绑定负载均衡器到指定的节点。  
    service.kubernetes.io/qcloud-loadbalancer-backends-label: key in (value1,  
value2)  
  
    # 为已有负载均衡器添加 ID。  
    service.kubernetes.io/tke-existed-lbid : lb-6swtxxxx  
  
    # 负载均衡器 (LB) 的自定义参数尚不支持修改 LB 类型。  
    service.kubernetes.io/service.extensiveParameters: ""  
  
    # 自定义负载均衡监听器。  
    service.kubernetes.io/service.listenerParameters: ""
```

```
# 指定负载均衡类型。
# 可用参数: classic (Classic Cloud Load Balancer) 或 application (Application
Cloud Load Balancer)
service.kubernetes.io/loadbalance-type: xxxxx

# 指定公用网络带宽计费方法。
# 可用参数: TRAFFIC_POSTPAID_BY_HOUR(bill-by-traffic) 和
BANDWIDTH_POSTPAID_BY_HOUR (bill-by-bandwidth).
service.kubernetes.io/qcloud-loadbalancer-internet-charge-type: xxxxxxx

# 指定带宽参数 (取值范围 : [1,2000] Mbps).
service.kubernetes.io/qcloud-loadbalancer-internet-max-bandwidth-out: "10
"
# 当设置该注解时，负载平衡器将只注册正在运行 Pod 的节点，
# 否则所有节点将被注册。
service.kubernetes.io/local-svc-only-bind-node-with-pod: true
```

ExternalName 类型

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。你可以使用 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
```

说明：ExternalName 服务接受 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](#)。

当查找主机 my-service.prod.svc.cluster.local 时，集群 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。如果以后你决定将数据库移到集群中，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 type。

警告：

对于一些常见的协议，包括 HTTP 和 HTTPS，你使用 ExternalName 可能会遇到问题。如果你使用 ExternalName，那么集群内客户端使用的主机名与 ExternalName 引用的名称不同。

对于使用主机名的协议，此差异可能会导致错误或意外响应。HTTP 请求将具有源服务器无法识别的 Host: 标头；TLS 服务器将无法提供与客户端连接的主机名匹配的证书。

说明：本部分感谢 [Alen Komljen](#) 的 [Kubernetes Tips - Part1](#) 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 使用用户空间代理，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](#) 获取更多细节。

使用用户空间代理，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，还是值得去理解的。

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。对于 Service 资源的设计，这意味着如果用户的选择有可能与他人冲突，那就不要让用户自行选择端口号。这是一个隔离性的失败。

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。Kubernetes 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。

在控制平面中，一个后台 Controller 的职责是创建映射表（需要支持从使用了内存锁的 Kubernetes 的旧版本迁移过来）。同时 Kubernetes 会通过控制器检查不合理的分配（如管理员干预导致的）以及清理已被分配但不再被任何 Service 使用的 IP 地址。

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。当创建后端 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 "服务代理" 的端口。"服务代理" 选择一个后端，并将客户端的流量代理到后端上。

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。当创建后端 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，它会配置一系列的 iptables 规则，从 VIP 重定向到每个 Service 规则。该特定于服务的规则连接到特定于 Endpoint 的规则，而后者会重定向（目标地址转译）到后端。

当客户端连接到一个 VIP，iptables 规则开始起作用。一个后端会被选择（或者根据会话亲和性，或者随机），数据包被重定向到这个后端。不像用户空间代理，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 10000 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，你可以通过基于 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 具有更复杂的负载均衡算法（最小连接、局部性、加权、持久性）。

API 对象

Service 是 Kubernetes REST API 中的顶级资源。你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](#)。

受支持的协议

TCP

你可以将 TCP 用于任何类型的服务，这是默认的网络协议。

UDP

你可以将 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。

SCTP

FEATURE STATE: Kubernetes v1.20 [stable]

一旦你使用了支持 SCTP 流量的网络插件，你就可以使用 SCTP 于更多的服务。对于 type = LoadBalancer 的服务，SCTP 的支持取决于提供此设施的云供应商（多大数不支持）。

警告

支持多宿主 SCTP 关联

警告：

支持多宿主SCTP关联要求 CNI 插件能够支持为一个 Pod 分配多个接口和IP 地址。

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。

Windows

说明： 基于 Windows 的节点不支持 SCTP。

用户空间 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。

HTTP

如果你的云提供商支持它，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。

说明： 你还可以使用 [Ingres](#) 代替 Service 来公开 HTTP/HTTPS 服务。

PROXY 协议

如果你的云提供商支持它，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](#) 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
```

上述是来自客户端的数据。

接下来

- 阅读[使用服务访问应用](#)
- 阅读了解[Ingress](#)
- 阅读了解[端点切片 \(Endpoint Slices\)](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 18, 2020 at 11:06 AM PST: [Resync finshed, also I have reviewed the codes. \(db55c7c07\)](#)

服务拓扑 (Service Topology)

FEATURE STATE: Kubernetes v1.17 [alpha]

服务拓扑 (Service Topology) 可以让一个服务基于集群的 Node 拓扑进行流量路由。例如，一个服务可以指定流量是被优先路由到一个和客户端在同一个 Node 或者在同一可用区域的端点。

介绍

默认情况下，发往 ClusterIP 或者 NodePort 服务的流量可能会被路由到任意一个服务后端的地址上。从 Kubernetes 1.7 开始，可以将“外部”流量路由到节点上运行的 Pod 上，但不支持 ClusterIP 服务，更复杂的拓扑 — 比如分区路由 — 也还不支持。通过允许 Service 创建者根据源 Node 和目的 Node 的标签来定义流量路由策略，服务拓扑特性实现了服务流量的路由。

通过对源 Node 和目的 Node 标签的匹配，运营者可以使用任何符合运营者要求的度量值 来指定彼此“较近”和“较远”的节点组。例如，对于在公有云上的运营者来说，更偏向于把流量控制在同一区域内，因为区域间的流量是有费用成本的，而区域内的流量没有。其它常用需求还包括把流量路由到由 DaemonSet 管理的本地 Pod 上，或者 把保持流量在连接同一机架交换机的 Node 上，以获得低延时。

使用服务拓扑

如果集群启用了服务拓扑功能后，就可以在 Service 配置中指定 topologyKeys 字段，从而控制 Service 的流量路由。此字段是 Node 标签的优先顺序字段，将用于在访问这个 Service 时对端点进行排序。流量会被定向到第一个标签值和源 Node 标签值相匹配的 Node。如果这个 Service 没有匹配的后端 Node，那么第二个标签会被使用做匹配，以此类推，直到没有标签。

如果没有匹配到，流量会被拒绝，就如同这个 Service 根本没有后端。换言之，系统根据可用后端的第一个拓扑键来选择端点。如果这个字段被配置了而没有后端可以匹配客户端拓扑，那么这个 Service 对那个客户端是没有后端的，链接应该是失败的。这个字段配置为 “*” 意味着任意拓扑。这个通配符值如果使用了，那么只有作为配置值列表中的最后一个才有用。

如果 topologyKeys 没有指定或者为空，就没有启用这个拓扑约束。

一个集群中，其 Node 的标签被打为其主机名，区域名和地区名。那么就可以设置 Service 的 topologyKeys 的值，像下面的做法一样定向流量了。

- 只定向到同一个 Node 上的端点，Node 上没有端点存在时就失败：配置 ["kubernetes.io/hostname"]。
- 偏向定向到同一个 Node 上的端点，回退同一区域的端点上，然后是同一地区，其它情况下就失败：配置 ["kubernetes.io/hostname", "topology.kubernetes.io/zone", "topology.kubernetes.io/region"]。这或许很有用，例如，数据局部性很重要的情况下。
- 偏向于同一区域，但如果此区域中没有可用的终结点，则回退到任何可用的终结点：配置 ["topology.kubernetes.io/zone", "*"]。

约束条件

- 服务拓扑和 externalTrafficPolicy=Local 是不兼容的，所以 Service 不能同时使用这两种特性。但是在同一个集群的不同 Service 上是可以分别使用这两种特性的，只要不在同一个 Service 上就可以。
- 有效的拓扑键目前只有：kubernetes.io/hostname、topology.kubernetes.io/zone 和 topology.kubernetes.io/region，但是未来会推广到其它的 Node 标签。
- 拓扑键必须是有效的标签，并且最多指定16个。
- 通配符："*"，如果要用，则必须是拓扑键值的最后一个值。

示例

以下是使用服务拓扑功能的常见示例。

仅节点本地端点

仅路由到节点本地端点的一种服务。如果节点上不存在端点，流量则被丢弃：

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

```
topologyKeys:  
- "kubernetes.io/hostname"
```

首选节点本地端点

首选节点本地端点，如果节点本地端点不存在，则回退到集群范围端点的一种服务：

```
apiVersion: v1  
kind: Service  
metadata:  
  name: my-service  
spec:  
  selector:  
    app: my-app  
  ports:  
    - protocol: TCP  
      port: 80  
      targetPort: 9376  
  topologyKeys:  
    - "kubernetes.io/hostname"  
    - "*"
```

仅地域或区域端点

首选地域端点而不是区域端点的一种服务。如果以上两种范围内均不存在端点，流量则被丢弃。

```
apiVersion: v1  
kind: Service  
metadata:  
  name: my-service  
spec:  
  selector:  
    app: my-app  
  ports:  
    - protocol: TCP  
      port: 80  
      targetPort: 9376  
  topologyKeys:  
    - "topology.kubernetes.io/zone"  
    - "topology.kubernetes.io/region"
```

优先选择节点本地端点，地域端点，然后是区域端点

优先选择节点本地端点，地域端点，然后是区域端点，然后才是集群范围端点的一种服务。

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
    - "*"
```

接下来

- 阅读关于[启用服务拓扑](#)
- 阅读[用服务连接应用程序](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 17, 2020 at 3:43 PM PST: [Update service-topology.md \(9d6c1af32\)](#)

Pod 与 Service 的 DNS

本页面提供 Kubernetes 对 DNS 的支持的概述。

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。

哪些对象会有 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 自己的名字空间和集群默认域。如下示例是一个很好的说明：

假设在 Kubernetes 集群的名字空间 bar 中，定义了一个服务 foo。运行在名字空间 bar 中的 Pod 可以简单地通过 DNS 查询 foo 来找到该服务。运行在名字空间 quux 中的 Pod 可以通过 DNS 查询 foo.bar 找到该服务。

以下各节详细介绍了受支持的记录类型和支持的布局。其它布局、名称或者查询即使碰巧可以工作，也应视为实现细节，将来很可能被更改而且不会因此出现警告。有关最新规范请查看 [Kubernetes 基于 DNS 的服务发现](#)。

服务

A/AAAA 记录

"普通" 服务 (除了无头服务) 会以 my-svc.my-namespace.svc.cluster-domain.example 这种名字的形式被分配一个 DNS A 或 AAAA 记录，取决于服务的 IP 协议族。该名称会解析成对应服务的集群 IP。

"无头 (Headless)" 服务 (没有集群 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 或 AAAA 记录，具体取决于服务的 IP 协议族。与普通服务不同，这一记录会被解析成对应服务所选择的 Pod 集合的 IP。客户端要能够使用这组 IP，或者使用标准的轮转策略从这组 IP 中进行选择。

SRV 记录

Kubernetes 会为命名端口创建 SRV 记录，这些端口是普通服务或 [无头服务](#) 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。对普通服务，该记录会被解析成端口号和域名：my-svc.my-namespace.svc.cluster-domain.example。对无头服务，该记录会被解析成多个结果，服务对应的每个后端 Pod 各一个；其中包含 Pod 端口号和形为 auto-generated-name.my-svc.my-namespace.svc.cluster-domain.example 的域名。

Pods

A/AAAA 记录

一般而言，Pod 会对应如下 DNS 名字解析：

pod-ip-address.my-namespace.pod.cluster-domain.example

例如，对于一个位于 default 名字空间，IP 地址为 172.17.0.3 的 Pod，如果集群的域名为 cluster.local，则 Pod 会对应 DNS 名称：

172-17-0-3.default.pod.cluster.local.

Deployment 或通过 Service 暴露出来的 DaemonSet 所创建的 Pod 会有如下 DNS 解析名称可用：

pod-ip-address.deployment-name.my-namespace.svc.cluster-domain.example.

Pod 的 hostname 和 subdomain 字段

当前，创建 Pod 时其主机名取自 Pod 的 metadata.name 值。

Pod 规约中包含一个可选的 hostname 字段，可以用来指定 Pod 的主机名。当这个字段被设置时，它将优先于 Pod 的名字成为该 Pod 的主机名。举个例子，给定一个 host name 设置为 "my-host" 的 Pod，该 Pod 的主机名将被设置为 "my-host"。

Pod 规约还有一个可选的 subdomain 字段，可以用来指定 Pod 的子域名。举个例子，某 Pod 的 hostname 设置为 "foo"，subdomain 设置为 "bar"，在名字空间 "my-namespace" 中对应的完全限定域名 (FQDN) 为 "foo.bar.my-namespace.svc.cluster-domain.example"。

示例：

```
apiVersion: v1
kind: Service
metadata:
  name: default-subdomain
spec:
  selector:
    name: busybox
  clusterIP: None
  ports:
    - name: foo # 实际上不需要指定端口号
      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
```

如果某无头服务与某 Pod 在同一个名字空间中，且它们具有相同的子域名，集群的 DNS 服务器也会为该 Pod 的全限定主机名返回 A 记录或 AAAA 记录。例如，在同一个名字空间中，给定一个主机名为 "busybox-1"、子域名设置为 "default-subdomain" 的 Pod，和一个名称为 "default-subdomain" 的无头服务，Pod 将看到自己的 FQDN 为 "busybox-1.default-subdomain.my-namespace.svc.cluster-domain.example"。DNS 会为此名字提供一个 A 记录或 AAAA 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 或 AAAA 记录。

Endpoints 对象可以为任何端点地址及其 IP 指定 hostname。

说明：

因为没有为 Pod 名称创建 A 记录或 AAAA 记录，所以要创建 Pod 的 A 记录或 AAAA 记录需要 hostname。

没有设置 hostname 但设置了 subdomain 的 Pod 只会为无头服务创建 A 或 AAAA 记录 (default-subdomain.my-namespace.svc.cluster-domain.example) 指向 Pod 的 IP 地址。另外，除非在服务上设置了 publishNotReadyAddresses=True，否则只有 Pod 进入就绪状态 才会有与之对应的记录。

Pod 的 setHostnameAsFQDN 字段

FEATURE STATE: Kubernetes v1.19 [alpha]

前置条件：SetHostnameAsFQDN 特性门控 必须在 [API 服务器](#) 上启用。

当你在 Pod 规约中设置了 `setHostnameAsFQDN: true` 时，`kubelet` 会将 Pod 的全限定域名（ FQDN ）作为该 Pod 的主机名记录到 Pod 所在名字空间。在这种情况下，`hostname` 和 `hostname --fqdn` 都会返回 Pod 的全限定域名。

说明：

在 Linux 中，内核的主机名字段（ `struct utsname` 的 `nodename` 字段）限制最多 64 个字符。

如果 Pod 启用这一特性，而其 FQDN 超出 64 字符，Pod 的启动会失败。Pod 会一直处于 Pending 状态（通过 `kubectl` 所看到的 `ContainerCreating` ），并产生错误事件，例如 "Failed to construct FQDN from pod hostname and cluster domain, FQDN long-FQDN is too long (64 characters is the max, 70 characters requested)." （无法基于 Pod 主机名和集群域名构造 FQDN，FQDN long-FQDN 过长，至多 64 字符，请求字符数为 70）。对于这种场景而言，改善用户体验的一种方式是创建一个 [准入 Webhook 控制器](#)，在用户创建顶层对象（如 Deployment）的时候控制 FQDN 的长度。

Pod 的 DNS 策略

DNS 策略可以逐个 Pod 来设定。目前 Kubernetes 支持以下特定 Pod 的 DNS 策略。这些策略可以在 Pod 规约中的 `dnsPolicy` 字段设置：

- "Default": Pod 从运行所在的节点继承名称解析配置。参考 [相关讨论](#) 获取更多信息。
- "ClusterFirst": 与配置的集群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。集群管理员可能配置了额外的存根域和上游 DNS 服务器。参阅[相关讨论](#) 了解在这些场景中如何处理 DNS 查询的信息。
- "ClusterFirstWithHostNet" : 对于以 `hostNetwork` 方式运行的 Pod，应显式设置其 DNS 策略 "ClusterFirstWithHostNet"。
- "None": 此设置允许 Pod 忽略 Kubernetes 环境中的 DNS 设置。Pod 会使用其 `dnsConfig` 字段所提供的 DNS 设置。参见 [Pod 的 DNS 配置节](#)。

说明： "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 配置

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。所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。指定此属性时，所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`：可选的对象列表，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](#)



```
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
```

输出类似于

```
nameserver fd00:79:30::a
search default.svc.cluster-domain.example svc.cluster-domain.example cluster-
domain.example
options ndots:5
```

功能的可用性

Pod DNS 配置和 DNS 策略 "None" 的可用版本对应如下所示。

k8s 版本	特性支持
1.14	稳定
1.10	Beta (默认启用)
1.9	Alpha

接下来

有关管理 DNS 配置的指导，请查看 [配置 DNS 服务](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

使用 Service 连接到应用

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。

多个开发人员或是提供容器的团队之间协调端口的分配很难做到规模化，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。Kubernetes 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。

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](#)
□

```
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](#) 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](#)



```
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](#) 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，可以进一步阅读[服务代理](#) 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](#)。

说明：如果不希望服务环境变量（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](#) 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](#)。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。这里我们使用 CoreDNS 集群插件（应用名为 kube-dns），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()）。如果 CoreDNS 没有在运行，你可以参照[CoreDNS README](#) 或者[安装 CoreDNS](#) 来启用它。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
```

保护 Service

到现在为止，我们只在集群内部访问了 Nginx 服务器。在将 Service 暴露到因特网之前，我们希望确保通信信道是安全的。为实现这一目的，可能需要：

- 用于 HTTPS 的自签名证书（除非已经有了一个识别身份的证书）
- 使用证书配置的 Nginx 服务器

- 使证书可以访问 Pod 的 [Secret](#)

你可以从 [Nginx https 示例](#) 获取所有上述内容。你需要安装 go 和 make 工具。如果你不想安装这些软件，可以按照 后文所述的手动执行步骤执行操作。简要过程如下：

```
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	kubernetes.io/tls	2	1m

以下是 configmap：

```
kubectl create configmap nginxconfigmap --from-file=default.conf
```

```
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

NAME	DATA	AGE
nginxconfigmap	1	114s

以下是你在运行 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"
  type: kubernetes.io/tls
data:
  tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURIekNDQWdlZ0F3SUJBZ0IKQUp5M3IQK0pzMlpJTUEwR0NTcUdTSWIzRFFFQkJRVUFNQ1I4VRBUEJnTIYKQkFNVENHNW5hVzU0YzNaak1SRXdEd1IEVIFRS0V3aHVaMmx1ZUhOMII6QWVG
```

dzB4TnpFd01qWXdOekEzTVRKYQpGdzB4T0RFd01qWXdOekEzTVRKYU1DWXhFV
EFQQmdOVkJBTVRDRzVuYVc1NGMzWmpNUkV3RHdZRFZRUUtFd2h1CloybHVISE
4yWXpDQ0FTSXdEUVIKS29aSWh2Y05BUUVCQIFBRGdnRVBBRENDQVFvQ2dnRUJ
BSjFxSU1SOVdWM0IKMIZIQIRMRmtobDRONXijMEJxYUhIQktMSnJMcy8vdzZhU3
hRS29GbHIJSU94NGUrMIN5ajBFcndCLzIYTnBwbQppeW1CL3JkRldkOXg5UWhBQ
UxCZkVaTmNiV3NsTVFVcnhBZW50VWt1dk1vLzgvMHRpbGhjc3paenJEYVJ4NEo5C
i82UVRtVVI3a0ZTWUpOWTVQZkR3cGc3dlVvaDZmZ1Voam92VG42eHNVR0M2Q
URVODBpNXFlZWhNeVI1N2ImU2YKNHZpaXdIY3hnL3lZR1JBR9mRTRqakxCdmd
ONjc2SU90S01rZXV3R0ljNDFhd05tNnNTSzRqYUNGGeGpYSnZaZQp2by9kTIEybHh
HWCTKT2I3SEhXbXNhGp4WTRaNVk3R1ZoK0QrWnYvcW1mMFgvbVY0Rmo1Nz
V3ajFMWVBocWtsCmdhSXZYRyt4U1FVQ0F3RUFBYU5RTUU0d0hRWURWUjBPQkJ
ZRUZPNG9OWkI3YXc1OUlsYkROMzhIYkduYnhFVjcKTUI4R0ExVWRJd1FZTUjhQUZ
PNG9OWkI3YXc1OUlsYkROMzhIYkduYnhFVjdNQXdHQTFVZEV3UUZNQU1CQWY
4dwpEUVIKS29aSWh2Y05BUUVGQIFBRGdnRUJBRVhTMW9FU0lFaXdyMDhWcVA0
K2NwTHI3TW5FMTducDBvMm14alFvCjRGb0RvRjdRZnZqeE04Tzd2TjB0clcb2pGS
W0vWDE4ZnZaL3k4ZzVaWG40Vm8zc3hKVmRBcStNZC9jTStzUGEKNmJjTkNUekZq
eFpUV0UrKzE5NS9zb2dmOUZ3VDVDK3U2Q3B5N0M3MTZvUXRUakViV05VdEt4c
XI0Nk1OZWNCMApwRFhWZmdWQTRadkR4NFo3S2RiZDY5eXM3OVFHYmg5ZW1
PZ05NZFlsSUswSGt0ejF5WU4vbVpmK3FqTkJqbWZjCkNnMnlwbGQ0Wi8rUUNQZj
I3SkoybFIrY2FnT0R4elBWcGxNSEcybzgvTHFDdnh6elZPUDUxeXdLZEtxaUMwsVEK
Q0I5T2wwWW5scE9UNEh1b2hSuBPostlMm9KdFZsNUiyiczRpbDlhZ3RTVXFxUIU
9Ci0tLS0tRU5EIENFUJRkIDQVRFLS0tLS0K"

tls.key: "LS0tLS1CRUdJTiBQUklWQVRFIetFWS0tLS0tCk1JSUV2UUICQURBTkJna3F
oa2IHOXcwQkFRRUZBQVNDQktd2dnU2pBZ0VBQW9JQkFRQ2RhaURFZlZsZHdkb
FIKd1V5eFpJWmVEZWNuTkFhbWh4d1NpeWF5N1AvOE9ta3NVQ3FCWmNpQ0Rz
ZUh2dGtzbzICShBZi9WemFhWm9zcApnZjYzUlZuZmNmVUIRQUN3WHhHVfH
MXJKVEVGShRSHA3VkpMcnpLUC9QOUxZcFIYTE0yYzZ3MmtjZUNmZitrCkU1bEV
INUVbUNUV09UM3c4S1IPNzFLSWVuNEZJWTZMMDUrc2JGQmd1Z0ExUE5JdWF
ubm9UTWtIZTRuMG4rTDQKb3NCM01ZUDhtQmtRQIAzeE9JNHI3YjREZXUraURyU
2pKSHJzQmlIT05Xc0RadXJFaXVJMmdoY1kxeWIyWHI2UAozVFVOcGNSbC9pVG9z
QngxcHJHclk4V09HZVdPeGxZZmcvbWIVNnBuOUYvNWxIqlkrZStjSTITMkQ0YXBK
WUdpCkwxeHzzVWtGQWdNQkFBRUNnZ0VBZfCK0xkbk8ySEIOTGo5bWRsb25IU
GIHWWVzZ294RGQwci9hQ1Zkank4dIEKTjIwL3FQWkUxek1yall6Ry9kVGhTMmMw
c0QxaTBXSjdwR1IGb0xtdXIWTjItY0FXUTM5SjM0VHZaU2FFSWZWNgo5TE1jUhhN
TmFsNjRLMFRVbUFQZytGam9QSFhUUxLOERLOUtnNXNrSE5pOWNzMIY5ckd6V
WIVZWtBL0RBUIBTClI3L2ZjUFBacDRuRWVBZmI3WTk1R1llb1p5V21SU3VKdINybIB
ESgtUdW1vVIVWdkxMRHRzaG9reUxiTWVtN3oKMmJzVmpwSW1GTHjqbGtmQXI
pNHg0WjJrV3YyMFRrdWtsZU1jaVIMbjk4QWxiRi9DSmRLM3QraTRoMTVIR2ZQeg
poTnh3bk9QdIVTaDR2Qo03c2Q5TmtEUGJvS2JneVVHOXYamZhRGR2UVFLQmdR
RFFLM01nUkhkQ1pKNVFqZWFKCIFGdXF4cHdnNzhZTjQyL1NwenIUYmtGcVFoQW
tyczJxWGx1MDZBRzhrZzIzQkswaHkzaE9zSGgxcXRVK3NHZVAKOWRERHBsUWV0
ODZsY2FIR3hoc0V0L1R6cEdtNGFKSm5oNzVVA TVGZk9QTDhPTm1FZ3MxMVRhUI
dhNzZxeIRyMgphRlpjQ2pWV1g0YnRSTHVwSkgrMjZnY0FhUUtCZ1FEQmxVSUUzT
nNVOFBZEWyL25sQVB5VWs1T3IDdWc3dmVyCIUycXIrdXFzYnBkSi9hODViT1JhM0
5IVmpVM25uRGpHVHBWaE9JeXg5TEFrc2RwZEfjVmxvcG9HODhXYk9IMTAKMUD
qbnkySmdDK3JVWUZiRGtpUGx1K09IYnRnOXFYcGJMSHBzUVpsMGhucDBYSFNYV

```
m9CMULiQndnMGEyOFVadApCbFBtWmc2d1BRS0JnRHVIUVV2SDZHYTNDVUsxN  
FdmOFhIcFFnMU16M2VvWTBPQm5iSDRvZUZKZmcraEppSXInCm9RN3hqWldVR3  
BIc3AyblRtcHErQWISNzdyRVhsdlhtOEIVU2FsbkNiRGIKY01Pc29RdFBZNS9NczJMR  
m5LQTQKaENmL0pWb2FtZm1nZEN0ZGtFMXNINE9MR2IJVHdEbTRpb0dWZGIwM  
lInbzFyb2htNUpLMUI3MkpBb0dBUW01UQpHNDhXOTVhL0w1eSt5dCsyZ3YvUHM  
2VnBvMjZITzRNQ3IJazJVem9ZWE9IYnNkODJkaC8xT2sybGdHZI2K3VuCnc1YytZU  
XRSTHlhQmd3MUtpbGhFZDBKTWU3cGpUSVpnQWJ0LzVPbnIDak9OVXN2aDJjs2I  
rQ1Z2dTZsZIBjNkQKckliT2ZIaHhxV0RZK2Q1TGN1YSt2NzJ0RkxhenJsSIBsRzlOZHhr  
Q2dZRUf5elIzT3UyMDNRVVV6bUICRkwzZAp4Wm5XZ0JLSEo3TnNxcGFWb2RjL0d  
5aGVycjFDZzE2MmJaSjJDV2RsZkI0VEdtUjZZdmxTZEFOOFRwUWhFbUtKCnFBLzVz  
dHdxNWd0WGVLOVJmMWxXK29xNThRNTBxMmk1NVdUTThoSDZhTjlaMTItZ0FG  
dE5VdGNqQUx2dFYxdEYKWSs4WFjkSHJaRnBIWII2NWkwVW1VbGc9Ci0tLS0tRU5  
EIFBSSVZBVEUgS0VZLS0tLS0K"
```

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	kubernetes.io/tls	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 Service，暴露端口（80 和 443）：

[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
        - name: configmap-volume
          configMap:
            name: nginxconfigmap
      containers:
        - name: nginxhttps
          image: bprashanth/nginxhttps:1.0
          ports:
            - containerPort: 443
            - containerPort: 80
          volumeMounts:
            - mountPath: /etc/nginx/ssl
              name: secret-volume
            - mountPath: /etc/nginx/conf.d
              name: configmap-volume
```

关于 nginx-secure-app 清单，值得注意的几点如下：

- 它在相同的文件中包含了 Deployment 和 Service 的规约
- [Nginx 服务器](#) 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 服务器启动之前安装好。

```
kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
```

这时，你可以从任何节点访问到 Nginx 服务器。

```
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](#)



```
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/tls.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	LoadBalancer	10.0.162.149	xx.xxx.xxx.xxx	8080:30163/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

- 进一步了解如何[使用 Service 访问集群中的应用](#)
- 进一步了解如何[使用 Service 将前端连接到后端](#)
- 进一步了解如何[创建外部负载均衡器](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 04, 2021 at 10:35 AM PST: [Update connect-applications-service.md \(1602ec8ce\)](#)

端点切片 (Endpoint Slices)

FEATURE STATE: Kubernetes v1.17 [beta]

端点切片 (*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。

动机

Endpoints API 提供了在 Kubernetes 跟踪网络端点的一种简单而直接的方法。不幸的是，随着 Kubernetes 集群和 [服务](#) 逐渐开始为更多的后端 Pods 处理和发送请求，原来的 API 的局限性变得越来越明显。最重要的是那些因为要处理大量网络端点而带来的挑战。

由于任一服务的所有网络端点都保存在同一个 Endpoints 资源中，这类资源可能变得非常巨大，而这一变化会影响到 Kubernetes 组件（比如主控组件）的性能，并在 Endpoints 变化时需要处理大量的网络流量和处理。EndpointSlice 能够帮助你缓解这一问题，还能为一些诸如拓扑路由这类的额外功能提供一个可扩展的平台。

Endpoint Slice 资源

在 Kubernetes 中，EndpointSlice 包含对一组网络端点的引用。指定选择器后控制面会自动为设置了 [选择算符](#) 的 Kubernetes 服务创建 EndpointSlice。这些 EndpointSlice 将包含对与服务选择算符匹配的所有 Pod 的引用。EndpointSlice 通过唯一的协议、端口号和服务名称将网络端点组织在一起。EndpointSlice 的名称必须是合法的 [DNS 子域名](#)。

例如，下面是 Kubernetes 服务 example 的 EndpointSlice 资源示例。

```
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，控制面创建和管理的 EndpointSlice 将包含不超过 100 个端点。你可以使用 [kube-controller-manager](#) 的 `--max-endpoints-per-slice` 标志设置此值，最大值为 1000。

当涉及如何路由内部流量时，EndpointSlice 可以充当 [kube-proxy](#) 的决策依据。启用该功能后，在服务的端点数量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)

状况

EndpointSlice API 存储了可能对使用者有用的、有关端点的状况。这三个状况分别是 ready、serving 和 terminating。

Ready (就绪)

ready 状况是映射 Pod 的 Ready 状况的。处于运行中的 Pod，它的 Ready 状况被设置为 True，应该将此 EndpointSlice 状况也设置为 true。出于兼容性原因，当 Pod 处于终止过程中，ready 永远不会为 true。消费者应参考 serving 状况来检查处于终止中的 Pod 的就绪情况。该规则的唯一例外是将 spec.publishNotReadyAddresses 设置为 true 的服务。这些服务（Service）的端点将始终将 ready 状况设置为 true。

Serving (服务中)

FEATURE STATE: Kubernetes v1.20 [alpha]

serving 状况与 ready 状况相同，不同之处在于它不考虑终止状态。如果 EndpointSlice API 的使用者关心 Pod 终止时的就绪情况，就应检查此状况。

说明：

尽管 serving 与 ready 几乎相同，但是它是为防止破坏 ready 的现有含义而增加的。如果对于处于终止中的端点，ready 可能是 true，那么对于现有的客户端来说可能是有些意外的，因为从始至终，Endpoints 或 EndpointSlice API 从未包含处于终止中的端点。出于这个原因，ready 对于处于终止中的端点 总是 false，并且在 v1.20 中添加了新的状况 serving

，以便客户端可以独立于 ready 的现有语义来跟踪处于终止中的 Pod 的就绪情况。

Terminating (终止中)

FEATURE STATE: Kubernetes v1.20 [alpha]

Terminating 是表示端点是否处于终止中的状况。对于 Pod 来说，这是设置了删除时间戳的 Pod。

拓扑信息

FEATURE STATE: Kubernetes v1.20 [deprecated]

说明：EndpointSlices 中的 topology 字段已被弃用，并将在以后的版本中删除。将使用新的 nodeName 字段代替在 topology 中设置 kubernetes.io/hostname。可以确定的是，其他覆盖区和域的拓扑字段用 EndpointSlice 标签来表达更合适，该标签将应用于 EndpointSlice 内的所有端点。

EndpointSlice 中的每个端点都可以包含一定的拓扑信息。这一信息用来标明端点的位置，包含对应节点、可用区、区域的信息。当这些值可用时，控制面会为 EndpointSlice 设置如下拓扑标签：

- kubernetes.io/hostname - 端点所在的节点名称
- topology.kubernetes.io/zone - 端点所处的可用区
- topology.kubernetes.io/region - 端点所处的区域

这些标签的值时根据与切片中各个端点相关联的资源来生成的。标签 hostname 代表的是对应的 Pod 的 NodeName 字段的取值。zone 和 region 标签则代表的是对应的节点所拥有的同名标签的值。

管理

通常，控制面（尤其是端点切片的 [controller](#)）会创建和管理 EndpointSlice 对象。EndpointSlice 对象还有一些其他使用场景，例如作为服务网格（Service Mesh）的实现。这些场景都会导致有其他实体或者控制器负责管理额外的 EndpointSlice 集合。

为了确保多个实体可以管理 EndpointSlice 而且不会相互产生干扰，Kubernetes 定义了 [标签 endpointslice.kubernetes.io/managed-by](#)，用来标明哪个实体在管理某个 EndpointSlice。端点切片控制器会在自己所管理的所有 EndpointSlice 上将该标签值设置为 endpointslice-controller.k8s.io。管理 EndpointSlice 的其他实体也应该为此标签设置一个唯一值。

属主关系

在大多数场合下，EndpointSlice 都由某个 Service 所有，（因为）该端点切片正是为该服务跟踪记录其端点。这一属主关系是通过为每个 EndpointSlice 设置一个 属主

(owner) 引用，同时设置 kubernetes.io/service-name 标签来标明的，目的是方便查找隶属于某服务的所有 EndpointSlice。

EndpointSlice 镜像

在某些场合，应用会创建定制的 Endpoints 资源。为了保证这些应用不需要并发递更改 Endpoints 和 EndpointSlice 资源，集群的控制面将大多数 Endpoints 映射到对应的 EndpointSlice 之上。

控制面对 Endpoints 资源进行映射的例外情况有：

- Endpoints 资源上标签 endpointslice.kubernetes.io/skip-mirror 值为 true。
- Endpoints 资源包含标签 control-plane.alpha.kubernetes.io/leader。
- 对应的 Service 资源不存在。
- 对应的 Service 的选择算符不为空。

每个 Endpoints 资源可能会被翻译到多个 EndpointSlices 中去。当 Endpoints 资源中包含多个子网或者包含多个 IP 地址族（IPv4 和 IPv6）的端点时，就有可能发生这种状况。每个子网最多有 1000 个地址会被镜像到 EndpointSlice 中。

EndpointSlices 的分布问题

每个 EndpointSlice 都有一组端口值，适用于资源内的所有端点。当为服务使用命名端口时，Pod 可能会就同一命名端口获得不同的端口号，因而需要不同的 EndpointSlice。这有点像 Endpoints 用来对子网进行分组的逻辑。

控制面尝试尽量将 EndpointSlice 填满，不过不会主动地在若干 EndpointSlice 之间执行再平衡操作。这里的逻辑也是相对直接的：

1. 列举所有现有的 EndpointSlices，移除那些不再需要的端点并更新那些已经变化的端点。
2. 列举所有在第一步中被更改过的 EndpointSlices，用新增加的端点将其填满。
3. 如果还有新的端点未被添加进去，尝试将这些端点添加到之前未更改的切片中，或者创建新切片。

这里比较重要的是，与在 EndpointSlice 之间完成最佳的分布相比，第三步中更看重限制 EndpointSlice 更新的操作次数。例如，如果有 10 个端点待添加，有两个 EndpointSlice 中各有 5 个空位，上述方法会创建一个新的 EndpointSlice 而不是将现有的两个 EndpointSlice 都填满。换言之，与执行多个 EndpointSlice 更新操作相比较，方法会优先考虑执行一个 EndpointSlice 创建操作。

由于 kube-proxy 在每个节点上运行并监视 EndpointSlice 状态，EndpointSlice 的每次变更都变得相对代价较高，因为这些状态变化要传递到集群中每个节点上。这一方法尝试限制要发送到所有节点上的变更消息个数，即使这样做可能会导致有多个 EndpointSlice 没有被填满。

在实践中，上面这种并非最理想的分布是很少出现的。大多数被 EndpointSlice 控制器处理的变更都是足够小的，可以添加到某已有 EndpointSlice 中去的。并且，假使无法添加到已有的切片中，不管怎样都会快就会需要一个新的 EndpointSlice 对象。

Deployment 的滚动更新为重新为 EndpointSlice 打包提供了一个自然的机会，所有 Pod 及其对应的端点在这一期间都会被替换掉。

重复的端点

由于 EndpointSlice 变化的自身特点，端点可能会同时出现在不止一个 EndpointSlice 中。鉴于不同的 EndpointSlice 对象在不同时刻到达 Kubernetes 的监视/缓存中，这种情况的出现是很自然的。使用 EndpointSlice 的实现必须能够处理端点出现在多个切片中的状况。关于如何执行端点去重（deduplication）的参考实现，你可以在 kube-proxy 的 EndpointSlice 实现中找到。

接下来

- 了解[启用 EndpointSlice](#)
- 阅读[使用服务连接应用](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 19, 2021 at 3:59 PM PST: [resync content/zh/docs/concepts/services-networking/endpoint-slices.md \(786422bb5\)](#)

Ingress

FEATURE STATE: Kubernetes v1.19 [stable]

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。

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。
- 集群（Cluster）：一组运行由 Kubernetes 管理的容器化应用程序的节点。在此示例和在大多数常见的 Kubernetes 部署环境中，集群中的节点都不在公共网络中。
- 边缘路由器（Edge router）：在集群中强制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。

- 集群网络 (Cluster network) : 一组逻辑的或物理的连接，根据 Kubernetes 网络模型 在集群内实现通信。
- 服务 (Service) : Kubernetes 服务 使用 标签 选择算符 (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[服务](#)的 HTTP 和 HTTPS 路由。 流量路由由 Ingress 资源上定义的规则控制。

下面是一个将所有流量都发送到同一 Service 的简单 Ingress 示例：

[JavaScript must be [enabled](#) to view content]

可以将 Ingress 配置为服务提供外部可访问的 URL、负载均衡流量、终止 SSL/TLS，以及提供基于名称的虚拟主机等能力。[Ingress 控制器](#) 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](#) 或 [Service.Type=LoadBalancer](#) 类型的服务。

环境准备

你必须具有 [Ingress 控制器](#) 才能满足 Ingress 的要求。 仅创建 Ingress 资源本身没有任何效果。

你可能需要部署 Ingress 控制器，例如 [ingress-nginx](#)。 你可以从许多 [Ingress 控制器](#) 中进行选择。

理想情况下，所有 Ingress 控制器都应符合参考规范。但实际上，不同的 Ingress 控制器操作略有不同。

说明： 确保你查看了 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：

[service/networking/minimal-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
```

```
annotations:  
  nginx.ingress.kubernetes.io/rewrite-target: /  
spec:  
  rules:  
    - http:  
        paths:  
          - path: /testpath  
            pathType: Prefix  
        backend:  
          service:  
            name: test  
            port:  
              number: 80
```

与所有其他 Kubernetes 资源一样，Ingress 需要使用 apiVersion、kind 和 metadata 字段。 Ingress 对象的命名必须是合法的 [DNS 子域名名称](#)。有关使用配置文件的一般信息，请参见[部署应用](#)、[配置容器](#)、[管理资源](#)。 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 [重写目标注解](#)。不同的 [Ingress 控制器](#) 支持不同的注解。查看文档以供你选择 Ingress 控制器，以了解支持哪些注解。

Ingress 规约 提供了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，其中包含与所有传入请求匹配的规则列表。 Ingress 资源仅支持用于转发 HTTP 流量的规则。

Ingress 规则

每个 HTTP 规则都包含以下信息：

- 可选的 host。在此示例中，未指定 host，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 host（例如 foo.bar.com），则 rules 适用于该 host。
- 路径列表 paths（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。
- backend（后端）是 [Service 文档](#)中所述的服务和端口名称的组合。与规则的 host 和 path 匹配的对 Ingress 的 HTTP（和 HTTPS）请求将发送到列出的 backend。

通常在 Ingress 控制器中会配置 defaultBackend（默认后端），以服务于任何不符合规约中 path 的请求。

DefaultBackend

没有 rules 的 Ingress 将所有流量发送到同一个默认后端。 defaultBackend 通常是 [Ingress 控制器](#) 的配置选项，而非在 Ingress 资源中指定。

如果 hosts 或 paths 都没有与 Ingress 对象中的 HTTP 请求匹配，则流量将路由到默认后端。

资源后端

Resource 后端是一个 ObjectRef，指向同一名字空间中的另一个 Kubernetes，将其作为 Ingress 对象。Resource 与 Service 配置是互斥的，在二者均被设置时会无法通过合法性检查。Resource 后端的一种常见用法是将所有入站数据导向带有静态资产的对象存储后端。

[service/networking/ingress-resource-backend.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
      name: static-assets
  rules:
  - http:
      paths:
      - path: /icons
        pathType: ImplementationSpecific
        backend:
          resource:
            apiGroup: k8s.example.com
            kind: StorageBucket
            name: icon-assets
```

创建了如上的 Ingress 之后，你可以使用下面的命令查看它：

```
kubectl describe ingress ingress-resource-backend
```

```
Name:      ingress-resource-backend
Namespace: default
Address:
Default backend: APIGroup: k8s.example.com, Kind: StorageBucket, Name: static-assets
Rules:
Host      Path  Backends
----      ----  -----
*
```

```
/icons APIGroup: k8s.example.com, Kind: StorageBucket, Name: icon-  
assets  
Annotations: <none>  
Events: <none>
```

路径类型

Ingress 中的每个路径都需要有对应的路径类型 (Path Type)。未明确设置 pathType 的路径无法通过合法性检查。当前支持的路径类型有三种：

- ImplementationSpecific：对于这种路径类型，匹配方法取决于 IngressClass。具体实现可以将其作为单独的 pathType 处理或者与 Prefix 或 Exact 类型作相同处理。
- Exact：精确匹配 URL 路径，且区分大小写。
- Prefix：基于以 / 分隔的 URL 路径前缀匹配。匹配区分大小写，并且对路径中的元素逐个完成。路径元素指的是由 / 分隔符分隔的路径中的标签列表。如果每个 p 都是请求路径 p 的元素前缀，则请求与路径 p 匹配。

说明：如果路径的最后一个元素是请求路径中最后一个元素的子字符串，则不会匹配（例如：/foo/bar 匹配 /foo/bar/baz, 但不匹配 /foo/barbaz）。

示例

类型	路径	请求路径	匹配与否？
Prefix	/	(所有路径)	是
Exact	/foo	/foo	是
Exact	/foo	/bar	否
Exact	/foo	/foo/	否
Exact	/foo/	/foo	否
Prefix	/foo	/foo, /foo/	是
Prefix	/foo/	/foo, /foo/	是
Prefix	/aaa/bb	/aaa/bbb	否
Prefix	/aaa/bbb	/aaa/bbb	是
Prefix	/aaa/bbb/	/aaa/bbb	是，忽略尾部斜线
Prefix	/aaa/bbb	/aaa/bbb/	是，匹配尾部斜线
Prefix	/aaa/bbb	/aaa/bbb/ccc	是，匹配子路径
Prefix	/aaa/bbb	/aaa/bbbxyz	否，字符串前缀不匹配
Prefix	/, /aaa	/aaa/ccc	是，匹配 /aaa 前缀
Prefix	/, /aaa, /aaa/bbb	/aaa/bbb	是，匹配 /aaa/bbb 前缀
Prefix	/, /aaa, /aaa/bbb	/ccc	是，匹配 / 前缀
Prefix	/aaa	/ccc	否，使用默认后端

类型	路径	请求路径	匹配与否？
混合	/foo (Prefix), /foo (Exact)	/foo	是，优选 Exact 类型

多重匹配

在某些情况下，Ingress 中的多条路径会匹配同一个请求。这种情况下最长的匹配路径优先。如果仍然有两条同等的匹配路径，则精确路径类型优先于前缀路径类型。

主机名通配符

主机名可以是精确匹配（例如"foo.bar.com"）或者使用通配符来匹配（例如".foo.com"）。精确匹配要求 HTTP host 头部字段与 host 字段值完全匹配。通配符匹配则要求 HTTP host 头部字段与通配符规则中的后缀部分相同。

主机	host 头部	匹配与否？
*.foo.com	bar.foo.com	基于相同的后缀匹配
*.foo.com	baz.bar.foo.com	不匹配，通配符仅覆盖了一个 DNS 标签
*.foo.com	foo.com	不匹配，通配符仅覆盖了一个 DNS 标签

[service/networking/ingress-wildcard-host.yaml](#)
□

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
    - host: "foo.bar.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
            backend:
              service:
                name: service1
                port:
                  number: 80
    - host: "*.foo.com"
      http:
        paths:
          - pathType: Prefix
            path: "/foo"
            backend:
              service:
                name: service2
```

```
port:  
  number: 80
```

Ingress 类

Ingress 可以由不同的控制器实现，通常使用不同的配置。每个 Ingress 应当指定一个类，也就是一个对 IngressClass 资源的引用。IngressClass 资源包含额外的配置，其中包括应当实现该类的控制器名称。

[service/networking/external-lb.yaml](#)



```
apiVersion: networking.k8s.io/v1  
kind: IngressClass  
metadata:  
  name: external-lb  
spec:  
  controller: example.com/ingress-controller  
  parameters:  
    apiGroup: k8s.example.com  
    kind: IngressParameters  
    name: external-lb
```

IngressClass 资源包含一个可选的 parameters 字段，可用于为该类引用额外配置。

废弃的注解

在 Kubernetes 1.18 版本引入 IngressClass 资源和 ingressClassName 字段之前，Ingress 类是通过 Ingress 中的一个 kubernetes.io/ingress.class 注解来指定的。这个注解从未被正式定义过，但是得到了 Ingress 控制器的广泛支持。

Ingress 中新的 ingressClassName 字段是该注解的替代品，但并非完全等价。该注解通常用于引用实现该 Ingress 的控制器的名称，而这个新的字段则是对一个包含额外 Ingress 配置的 IngressClass 资源的引用，包括 Ingress 控制器的名称。

默认 Ingress 类

你可以将一个特定的 IngressClass 标记为集群默认 Ingress 类。将一个 IngressClass 资源的 ingressclass.kubernetes.io/is-default-class 注解设置为 true 将确保新的未指定 ingressClassName 字段的 Ingress 能够分配为这个默认的 IngressClass。

注意：如果集群中有多个 IngressClass 被标记为默认，准入控制器将阻止创建新的未指定 ingressClassName 的 Ingress 对象。解决这个问题只需确保集群中最多只能有一个 IngressClass 被标记为默认。

Ingress 类型

由单个 Service 来完成的 Ingress

现有的 Kubernetes 概念允许你暴露单个 Service (参见[替代方案](#))。 你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/test-ingress.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

如果使用 `kubectl apply -f` 创建此 Ingress，则应该能够查看刚刚添加的 Ingress 的状态：

```
kubectl get ingress test-ingress
```

NAME	CLASS	HOSTS	ADDRESS	POR	TS	AGE
test-ingress	external-lb	*	203.0.113.123	80	59s	

其中 203.0.113.123 是由 Ingress 控制器分配以满足该 Ingress 的 IP。

说明： 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，你通常会看到地址字段的值被设定为 `<pending>`。

简单扇出

一个扇出 (fanout) 配置根据请求的 HTTP URI 将来自同一 IP 地址的流量路由到多个 Service。 Ingress 允许你将负载均衡器的数量降至最低。例如，这样的设置：

[JavaScript must be [enabled](#) to view content]

将需要一个如下所示的 Ingress：

[service/networking/simple-fanout-example.yaml](#)



```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080

```

当你使用 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 (service1, service2) 存在。当它这样做时，你会在 Address 字段看到负载均衡器的地址。

说明：取决于你所使用的 [Ingress 控制器](#)，你可能需要创建默认 HTTP 后端服务。

基于名称的虚拟托管

基于名称的虚拟主机支持将针对多个主机名的 HTTP 流量路由到同一 IP 地址上。

[JavaScript must be [enabled](#) to view content]

以下 Ingress 让后台负载均衡器基于[host 头部字段](#) 来路由请求。

[service/networking/name-virtual-host-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
    - host: foo.bar.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service1
                port:
                  number: 80
    - host: bar.foo.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service2
                port:
                  number: 80
```

如果你创建的 Ingress 资源没有在 rules 中定义的任何 hosts，则可以匹配指向 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 会将针对 first.bar.com 的请求流量路由到 service1，将针对 second.foo.com 的请求流量路由到 service2，而针对该 IP 地址的、没有在请求中定义主机名的请求流量会被路由（即，不提供请求标头）到 service3。

[service/networking/name-virtual-host-ingress-no-third-host.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress-no-third-host
spec:
  rules:
    - host: first.bar.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service1
                port:
                  number: 80
    - host: second.bar.com
      http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service2
                port:
                  number: 80
    - http:
        paths:
          - pathType: Prefix
            path: "/"
            backend:
              service:
                name: service3
                port:
                  number: 80
```

TLS

你可以通过设定包含 TLS 私钥和证书的[Secret](#) 来保护 Ingress。 Ingress 只支持单个 TLS 端口 443，并假定 TLS 连接终止于 Ingress 节点（与 Service 及其 Pod 之间的流量都以明文传输）。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 的键名。这些数据包含用于 TLS 的证书和私钥。例如：

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 编码的 cert
  tls.key: base64 编码的 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。你需要确保创建的 TLS Secret 创建自包含 sslexample.foo.com 的公用名称（CN）的证书。这里的公共名称也被称为全限定域名（FQDN）。

说明：

注意，默认规则上无法使用 TLS，因为需要为所有可能的子域名发放证书。因此，tls 节区的 hosts 的取值需要域 rules 节区的 host 完全匹配。

[service/networking/tls-example-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
    - hosts:
        - https-example.foo.com
      secretName: testsecret-tls
  rules:
    - host: https-example.foo.com
      http:
        paths:
          - path: /
            pathType: Prefix
      backend:
```

```
service:  
  name: service1  
  port:  
    number: 80
```

说明：各种 Ingress 控制器所支持的 TLS 功能之间存在差异。请参阅有关 [nginx](#)、[GCE](#) 或者任何其他平台特定的 Ingress 控制器的文档，以了解 TLS 如何在你的环境中工作。

负载均衡

Ingress 控制器启动引导时使用一些适用于所有 Ingress 的负载均衡策略设置，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。你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，尽管健康检查不是通过 Ingress 直接暴露的，在 Kubernetes 中存在并行的概念，比如 [就绪检查](#)，允许你实现相同的目的。请检查特定控制器的说明文档（[nginx](#)，[GCE](#)）以了解它们是怎样处理健康检查的。

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
```

这一命令将打开编辑器，允许你以 YAML 格式编辑现有配置。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
        pathType: Prefix
    - host: bar.baz.com
      http:
        paths:
          - backend:
              serviceName: service2
              servicePort: 80
              path: /foo
              pathType: Prefix
...

```

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

```

你也可以通过 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

不同的云厂商使用不同的技术来实现跨故障域的流量分布。详情请查阅相关 Ingress 控制器的文档。请查看相关 [Ingress 控制器](#) 的文档以了解详细信息。

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](#)
- 使用 [Service.Type=NodePort](#)

接下来

- 进一步了解 [Ingress API](#)
- 进一步了解 [Ingress 控制器](#)
- [使用 NGINX 控制器在 Minikube 上安装 Ingress](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 13, 2020 at 1:46 PM PST: [\[zh\] Sync changes from English site \(5\) \(e97677e6d\)](#)

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。

与作为 kube-controller-manager 可执行文件的一部分运行的其他类型的控制器不同，Ingress 控制器不是随集群自动启动的。基于此页面，你可选择最适合你的集群的 ingress 控制器实现。

Kubernetes 作为一个项目，目前支持和维护 [AWS](#)，[GCE](#) 和 [nginx](#) Ingress 控制器。

其他控制器

注意：本部分链接到提供 Kubernetes 所需功能的第三方项目。

Kubernetes 项目作者不负责这些项目。此页面遵循[CNCF 网站指南](#)，按字

母顺序列出项目。要将项目添加到此列表中，请在提交更改之前阅读[内容指南](#)。

- [AKS 应用程序网关 Ingress 控制器](#) 是一个配置 [Azure 应用程序网关](#) 的 Ingress 控制器。
- [Ambassador API 网关](#)是一个基于 [Envoy](#) 的 Ingress 控制器。
- [Apache APISIX Ingress 控制器](#) 是一个基于 [Apache APISIX 网关](#) 的 Ingress 控制器。
- [Avi Kubernetes Operator](#) 使用 [VMware NSX Advanced Load Balancer](#) 提供第 4 到第 7 层的负载均衡。
- [Citrix Ingress 控制器](#) 可以用来与 Citrix Application Delivery Controller 一起使用。
- [Contour](#) 是一个基于 [Envoy](#) 的 Ingress 控制器。
- F5 BIG-IP 的 [用于 Kubernetes 的容器 Ingress 服务](#) 让你能够使用 Ingress 来配置 F5 BIG-IP 虚拟服务器。
- [Gloo](#) 是一个开源的、基于 [Envoy](#) 的 Ingress 控制器，能够提供 API 网关功能，
- [HAProxy Ingress](#) 针对 [HAProxy](#) 的 Ingress 控制器。
- [用于 Kubernetes 的 HAProxy Ingress 控制器](#) 也是一个针对 [HAProxy](#) 的 Ingress 控制器。
- [Istio Ingress](#) 是一个基于 [Istio](#) 的 Ingress 控制器。
- [用于 Kubernetes 的 Kong Ingress 控制器](#) 是一个用来驱动 [Kong Gateway](#) 的 Ingress 控制器。
- [用于 Kubernetes 的 NGINX Ingress 控制器](#) 能够与 [NGINX](#) Web 服务器（作为代理）一起使用。
- [Skipper](#) HTTP 路由器和反向代理可用于服务组装，支持包括 Kubernetes Ingress 这类使用场景，设计用来作为构造你自己的定制代理的库。
- [Traefik Kubernetes Ingress 提供程序](#) 是一个用于 [Traefik](#) 代理的 Ingress 控制器。
- [Voyager](#) 是一个针对 [HAProxy](#) 的 Ingress 控制器。

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](#)。创建 ingress 时，应该使用适当的 [ingress.class](#) 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。

说明：确保你查看了 ingress 控制器的文档，以了解选择它的注意事项。

接下来

- 进一步了解 [Ingress](#)。
- 在 [Minikube 上使用 NGINX 控制器安装 Ingress](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 20, 2021 at 12:31 AM PST: [\[zh\] fix NGINX Ingress Controller URL \(db90dce84\)](#)

网络策略

如果你希望在 IP 地址或端口层面（OSI 第 3 层或第 4 层）控制网络流量，则你可以考虑为集群中特定应用使用 Kubernetes 网络策略（NetworkPolicy）。

NetworkPolicy 是一种以应用为中心的结构，允许你设置如何允许 [Pod](#) 与网络上的各类网络“实体”（我们这里使用实体以避免过度使用诸如“端点”和“服务”这类常用术语，这些术语在 Kubernetes 中有特定含义）通信。

Pod 可以通信的 Pod 是通过如下三个标识符的组合来辨识的：

1. 其他被允许的 Pods（例外：Pod 无法阻塞对自身的访问）
2. 被允许的名字空间
3. IP 组块（例外：与 Pod 运行所在的节点的通信总是被允许的，无论 Pod 或节点的 IP 地址）

在定义基于 Pod 或名字空间的 NetworkPolicy 时，你会使用 [选择算符](#) 来设定哪些流量可以进入或离开与该算符匹配的 Pod。

同时，当基于 IP 的 NetworkPolicy 被创建时，我们基于 IP 组块（CIDR 范围）来定义策略。

前置条件

网络策略通过[网络插件](#)来实现。要使用网络策略，你必须使用支持 NetworkPolicy 的网络解决方案。创建一个 NetworkPolicy 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 在被某 NetworkPolicy 选中时进入被隔离状态。一旦名字空间中有 NetworkPolicy 选择了特定的 Pod，该 Pod 会拒绝该 NetworkPolicy 所不允许的连接。（名字空间下其他未被 NetworkPolicy 所选择的 Pod 会继续接受所有的流量）

网络策略不会冲突，它们是累积的。如果任何一个或多个策略选择了一个 Pod，则该 Pod 受限于这些策略的入站 (Ingress) /出站 (Egress) 规则的并集。因此评估的顺序并不会影响策略的结果。

NetworkPolicy 资源

参阅 [NetworkPolicy](#) 来了解资源的完整定义。

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
```

说明：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。

必需字段：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 [使用 ConfigMap 配置容器, 和对象管理](#)。

spec：NetworkPolicy 规约 中包含了在一个名字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，它对该策略所适用的一组 Pod 进行选择。示例中的策略选择带有 "role=db" 标签的 Pod。空的 podSelector 选择名字空间下的所有 Pod。

policyTypes: 每个 NetworkPolicy 都包含一个 policyTypes 列表，其中包含 Ingress 或 Egress 或两者兼具。policyTypes 字段表示给定的策略是应用于进入所选 Pod 的入站流量还是来自所选 Pod 的出站流量，或两者兼有。如果 NetworkPolicy 未指定 policyTypes 则默认情况下始终设置 Ingress；如果 NetworkPolicy 有任何出口规则的话则设置 Egress。

ingress: 每个 NetworkPolicy 可包含一个 ingress 规则的白名单列表。每个规则都允许同时匹配 from 和 ports 部分的流量。示例策略中包含一条简单的规则：它匹配某个特定端口，来自三个来源中的一个，第一个通过 ipBlock 指定，第二个通过 namespace Selector 指定，第三个通过 podSelector 指定。

egress: 每个 NetworkPolicy 可包含一个 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，该规则将指定端口上的流量匹配到 10.0.0.0/24 中的任何目的地。

所以，该网络策略示例：

1. 隔离 "default" 名字空间下 "role=db" 的 Pod（如果它们不是已经被隔离的话）。
2. (Ingress 规则) 允许以下 Pod 连接到 "default" 名字空间下的带有 "role=db" 标签的所有 Pod 的 6379 TCP 端口：
 - "default" 名字空间下带有 "role=frontend" 标签的所有 Pod
 - 带有 "project=myproject" 标签的所有名字空间中的 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. (Egress 规则) 允许从带有 "role=db" 标签的名字空间下的任何 Pod 到 CIDR 10.0.0.0/24 下 5978 TCP 端口的连接。

参阅[声明网络策略演练](#) 了解更多示例。

选择器 to 和 from 的行为

可以在 ingress 的 from 部分或 egress 的 to 部分中指定四种选择器：

podSelector: 此选择器将在与 NetworkPolicy 相同的名字空间中选择特定的 Pod , 应将其允许作为入站流量来源或出站流量目的地。

namespaceSelector : 此选择器将选择特定的名字空间 , 应将所有 Pod 用作其 入站流量来源或出站流量目的地。

namespaceSelector 和 podSelector : 一个指定 namespaceSelector 和 podSelector 的 to/from 条目选择特定名字空间中的特定 Pod。 注意使用正确的 YAML 语法 ; 下面的策略 :

```
...
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 所在的名字空间中标有 user=alice 的连接。但是 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 , 允许来自本地名字空间中标有 role=client 的 Pod 的连接 , 或 来自任何名字空间中标有 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: 此选择器将选择特定的 IP CIDR 范围以用作入站流量来源或出站流量目的地。 这些应该是集群外部 IP , 因为 Pod IP 存在时间短暂的且随机产生。

集群的入站和出站机制通常需要重写数据包的源 IP 或目标 IP。 在发生这种情况时 , 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。

对入站流量而言，这意味着在某些情况下，你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。

对于出站流量而言，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，如果名字空间中不存在任何策略，则所有进出该名字空间中 Pod 的流量都被允许。以下示例使你可以更改该名字空间中的默认行为。

默认拒绝所有入站流量

你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站流量的 NetworkPolicy 来为名字空间创建 "default" 隔离策略。

[service/networking/network-policy-default-deny-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
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。

默认允许所有入站流量

如果要允许所有流量进入某个名字空间中的所有 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名字空间中的所有流量。

[service/networking/network-policy-allow-all-ingress.yaml](#)


```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
```

```
- {}  
policyTypes:  
- Ingress
```

默认拒绝所有出站流量

你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站流量的 NetworkPolicy 来为名字空间创建 "default" egress 隔离策略。

[service/networking/network-policy-default-deny-egress.yaml](#)


```
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: default-deny-egress  
spec:  
  podSelector: {}  
  policyTypes:  
  - Egress
```

此策略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入站流量隔离行为。

默认允许所有出站流量

如果要允许来自名字空间中所有 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，该策略明确允许该名字空间中的所有出站流量。

[service/networking/network-policy-allow-all-egress.yaml](#)


```
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: allow-all-egress  
spec:  
  podSelector: {}  
  egress:  
  - {}  
  policyTypes:  
  - Egress
```

默认拒绝所有入口和所有出站流量

你可以为名字空间创建"默认"策略，以通过在该名字空间中创建以下 NetworkPolicy 来阻止所有入站和出站流量。



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

此策略可以确保即使没有被其他任何 NetworkPolicy 选择的 Pod 也不会被 允许入站或出站流量。

SCTP 支持

FEATURE STATE: Kubernetes v1.19 [beta]

作为一个 Beta 特性，SCTP 支持默认是被启用的。要在集群层面禁用 SCTP，你（或你的集群管理员）需要为 API 服务器指定 `--feature-gates=SCTPSupport=false,...` 来禁用 SCTPSupport 特性门控。启用该特性门控后，用户可以将 NetworkPolicy 的 protocol 字段设置为 SCTP。

说明：你必须使用支持 SCTP 协议网络策略的 [CNI](#) 插件。

你通过网络策略（至少目前还）无法完成的工作

到 Kubernetes v1.20 为止，NetworkPolicy API 还不支持以下功能，不过 你可能可以使用操作系统组件（如 SELinux、OpenVSwitch、IPTables 等等）或者第七层技术（Ingress 控制器、服务网格实现）或准入控制器来实现一些 替代方案。如果你对 Kubernetes 中的网络安全性还不太了解，了解使用 NetworkPolicy API 还无法实现下面的用户场景是很值得的。对这些用户场景中的一部分（而非全部）的讨论仍在进行，或许在将来 NetworkPolicy API 中会给出一定支持。

- 强制集群内部流量经过某公用网关（这种场景最好通过服务网格或其他代理来实现）；
- 与 TLS 相关的场景（考虑使用服务网格或者 Ingress 控制器）；
- 特定于节点的策略（你可以使用 CIDR 来表达这一需求不过你无法使用节点在 Kubernetes 中的其他标识信息来辩识目标节点）；
- 基于名字来选择名字空间或者服务（不过，你可以使用 [标签](#) 来选择目标 Pod 或名字空间，这也通常是一种可靠的替代方案）；
- 创建或管理由第三方来实际完成的“策略请求”；
- 实现适用于所有名字空间或 Pods 的默认策略（某些第三方 Kubernetes 发行版本或项目可以做到这点）；
- 高级的策略查询或者可达性相关工具；

- 在同一策略声明中选择目标端口范围的能力；
- 生成网络安全事件日志的能力（例如，被阻塞或接收的连接请求）；
- 显式地拒绝策略的能力（目前，NetworkPolicy 的模型默认采用拒绝操作，其唯一的能力是添加允许策略）；
- 禁止本地回路或指向宿主的网络流量（Pod 目前无法阻塞 localhost 访问，它们也无法禁止来自所在节点的访问请求）。

接下来

- 参阅[声明网络策略 演练](#)了解更多示例；
- 有关 NetworkPolicy 资源所支持的常见场景的更多信息，请参见 [此指南](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 13, 2020 at 4:24 PM PST: [\[zh\] Resync network-policies \(84df6d320\)](#)

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。你可以通过 PodSpec 的 HostAliases 字段来添加这些自定义条目。

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

让我们从一个 Nginx Pod 开始，该 Pod 被分配一个 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
```

默认情况下，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。例如，要将 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 配置 HostAliases。

[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"
```

你可以使用以下命令用此配置启动 Pod：

```
kubectl apply -f hostaliases-pod.yaml
```

```
pod/hostaliases-pod created
```

检查 Pod 详情，查看其 IPv4 地址和状态：

```
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](#) 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](#) 该文件。

注意：

请避免手工更改容器内的 hosts 文件内容。

如果你对 hosts 文件做了手工修改，这些修改都会在容器退出时丢失。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 13, 2020 at 1:46 PM PST: [\[zh\] Sync changes from English site \(5\) \(e97677e6d\)](#)

IPv4/IPv6 双协议栈

FEATURE STATE: Kubernetes v1.16 [alpha]

IPv4/IPv6 双协议栈能够将 IPv4 和 IPv6 地址分配给 [Pod](#) 和 [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

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
- 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 1.20 版本及更高版本，有关更早 Kubernetes 版本的使用双栈服务的信息，请参考那个版本的 Kubernetes 文档。
- 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）
- 支持双协议栈的网络插件（如 Kubenet 或 Calico）

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](#)，并且设置双协议栈的集群网络分配：

- kube-apiserver:
 - `--feature-gates="IPv6DualStack=true"`
 - `--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>`

- kube-controller-manager:
 - --feature-gates="IPv6DualStack=true"
 - --cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR> 例如 --cluster-cidr=10.244.0.0/16,fc00::/48
 - --service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR> 例如 --service-cluster-ip-range=10.0.0.0/16,fd00::/108
 - --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:
 - --cluster-cidr=<IPv4 CIDR>,<IPv6 CIDR>
 - --feature-gates="IPv6DualStack=true"

说明 :

IPv4 CIDR 的一个例子 : 10.244.0.0/16 (尽管你会提供你自己的地址范围)

IPv6 CIDR 的一个例子 : fdXY:IJKL:MNOP:15::/64 (这里演示的是格式而非有效地址 - 请看 [RFC 4193](#))

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 [Service](#)。服务的地址族默认为第一个服务集群 IP 范围的地址族 (通过 kube-controller-manager 的 --service-cluster-ip-range 参数配置) 当你定义服务时 , 可以选择将其配置为双栈。若要指定所需的行为 , 你可以设置 .spec.ipFamilyPolicy 字段为以下值之一 :

- SingleStack : 单栈服务。控制面使用第一个配置的服务集群 IP 范围为服务分配集群 IP。
- PreferDualStack :
 - 仅当集群启用了双栈时使用。为服务分配 IPv4 和 IPv6 集群 IP。
 - 如果集群没有启用双堆栈 , 则此设置与 SingleStack 行为相同。
- RequireDualStack : 从 IPv4 和 IPv6 的地址范围分配服务的 .spec.ClusterIPs
 - 从基于在 .spec.ipFamilies 数组中第一个元素的地址族的 .spec.ClusterIPs 列表中选择 .spec.ClusterIP
 - 集群必须配置双栈网络

如果你想要定义哪个 IP 族用于单栈或定义双栈 IP 族的顺序 , 可以通过设置服务上的可选字段 .spec.ipFamilies 来选择地址族。

说明 : .spec.ipFamilies 字段是不可变的 , 因为系统无法为已经存在的服务重新分配 .spec.ClusterIP 。如果你想改变 .spec.ipFamilies , 则需要删除并重新创建服务。

你可以设置 `.spec.ipFamily` 为以下任何数组值：

- `["IPv4"]`
- `["IPv6"]`
- `["IPv4", "IPv6"]` (双栈)
- `["IPv6", "IPv4"]` (双栈)

你所列出的第一个地址族用于原来的 `.spec.ClusterIP` 字段。

双栈服务配置场景

以下示例演示多种双栈服务配置场景下的行为。

新服务的双栈选项

1. 此服务规约中没有显式设定 `.spec.ipFamilyPolicy`。当你创建此服务时，Kubernetes 从所配置的第一个 `service-cluster-ip-range` 种为服务分配一个集群 IP，并设置 `.spec.ipFamilyPolicy` 为 `SingleStack`。（[无选择算符的服务](#)和[无头服务](#)的行为方式与此相同。）

[service/networking/dual-stack-default-svc.yaml](#)


```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

1. 此服务规约显式地将 `.spec.ipFamilyPolicy` 设置为 `PreferDualStack`。当你在双栈集群上创建此服务时，Kubernetes 会为该服务分配 IPv4 和 IPv6 地址。控制平面更新服务的 `.spec` 以记录 IP 地址分配。字段 `.spec.ClusterIPs` 是主要字段，包含两个分配的 IP 地址；`.spec.ClusterIP` 是次要字段，其取值从 `.spec.ClusterIPs` 计算而来。

[service/networking/dual-stack-preferred-svc.yaml](#)


```
apiVersion: v1
kind: Service
```

```
metadata:  
  name: my-service  
  labels:  
    app: MyApp  
spec:  
  ipFamilyPolicy: PreferDualStack  
  selector:  
    app: MyApp  
  ports:  
    - protocol: TCP  
      port: 80
```

- 下面的服务规约显式地在 `.spec.ipFamilies` 中指定 IPv6 和 IPv4，并将 `.spec.ipFamilyPolicy` 设定为 `PreferDualStack`。当 Kubernetes 为 `.spec.ClusterIPs` 分配一个 IPv6 和一个 IPv4 地址时，`.spec.ClusterIP` 被设置成 IPv6 地址，因为它是 `.spec.ClusterIPs` 数组中的第一个元素，覆盖其默认值。

[service/networking/dual-stack-preferred-ipfamilies-svc.yaml](#)


```
apiVersion: v1  
kind: Service  
metadata:  
  name: my-service  
  labels:  
    app: MyApp  
spec:  
  ipFamilyPolicy: PreferDualStack  
  ipFamilies:  
    - IPv6  
    - IPv4  
  selector:  
    app: MyApp  
  ports:  
    - protocol: TCP  
      port: 80
```

现有服务的双栈默认值

下面示例演示了在服务已经存在的集群上新启用双栈时的默认行为。

- 在集群上启用双栈时，控制面会将现有服务（无论是 IPv4 还是 IPv6）配置 `.spec.ipFamilyPolicy` 设置为 `SingleStack` 并设置 `.spec.ipFamilies` 为服务的当前地址族。

[service/networking/dual-stack-default-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

你可以通过使用 kubectl 检查现有服务来验证此行为。

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: MyApp
  name: my-service
spec:
  clusterIP: 10.0.197.123
  clusterIPs:
    - 10.0.197.123
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: MyApp
  type: ClusterIP
status:
  loadBalancer: {}
```

1. 在集群上启用双栈时，带有选择算符的现有 [无头服务](#) 由控制面设置 .spec.ipFamilyPolicy 为 SingleStack 并设置 .spec.ipFamilies 为第一个服务群集 IP 范围的地址族（通过配置 kube-controller-manager 的 --service-cluster-ip-range 参数），即使 .spec.ClusterIP 的设置值为 None 也如此。

[service/networking/dual-stack-default-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

你可以通过使用 kubectl 检查带有选择算符的现有无头服务来验证此行为。

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: MyApp
  name: my-service
spec:
  clusterIP: None
  clusterIPs:
    - None
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: MyApp
```

在单栈和双栈之间切换服务

服务可以从单栈更改为双栈，也可以从双栈更改为单栈。

1. 要将服务从单栈更改为双栈，根据需要将 .spec.ipFamilyPolicy 从 SingleStack 改为 PreferDualStack 或 RequireDualStack。当你将此服务从单栈更改为双栈时，

Kubernetes 将分配缺失的地址族，以便现在该服务具有 IPv4 和 IPv6 地址。编辑服务规约将 .spec.ipFamilyPolicy 从 SingleStack 改为 PreferDualStack。

之前：

```
spec:  
  ipFamilyPolicy: SingleStack
```

之后：

```
spec:  
  ipFamilyPolicy: PreferDualStack
```

1. 要将服务从双栈更改为单栈，请将 .spec.ipFamilyPolicy 从 PreferDualStack 或 RequireDualStack 改为 SingleStack。当你将此服务从双栈更改为单栈时，Kubernetes 只保留 .spec.ClusterIPs 数组中的第一个元素，并设置 .spec.ClusterIP 为那个 IP 地址，并设置 .spec.ipFamilies 为 .spec.ClusterIPs 地址族。

无选择算符的无头服务

对于[不带选择算符的无头服务](#)，若没有显式设置 .spec.ipFamilyPolicy，则 .spec.ipFamilyPolicy 字段默认设置为 RequireDualStack。

LoadBalancer 类型

要为你的服务提供双栈负载均衡器：

- 将 .spec.type 字段设置为 LoadBalancer
- 将 .spec.ipFamilyPolicy 字段设置为 PreferDualStack 或者 RequireDualStack

说明：为了使用双栈的负载均衡器类型服务，你的云驱动必须支持 IPv4 和 IPv6 的负载均衡器。

出站流量

如果你要启用出口流量，以便使用非公开路由 IPv6 地址的 Pod 到达集群外地址（例如公网），则需要通过透明代理或 IP 伪装等机制使 Pod 使用公共路由的 IPv6 地址。[ip-masq-agent](#)项目支持在双栈集群上进行 IP 伪装。

说明：确认你的 [CNI](#) 驱动支持 IPv6。

接下来

- [验证 IPv4/IPv6 双协议栈](#)网络

反馈

此页是否对您有帮助？

是否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 23, 2020 at 10:27 AM PST: [sync zh dual-stack.md \(9a0fc8307\)](#)

存储

为集群中的 Pods 提供长期和临时存储的方法。

[卷](#)

[卷快照](#)

[持久卷](#)

[CSI 卷克隆](#)

[卷快照类](#)

[存储类](#)

[动态卷供应](#)

[存储容量](#)

[临时卷](#)

[特定于节点的卷数限制](#)

卷

Container 中的文件在磁盘上是临时存放的，这给 Container 中运行的较重要的应用程序带来一些问题。问题之一是当容器崩溃时文件丢失。kubelet 会重新启动容器，但容器会以干净的状态重启。第二个问题会在同一 Pod 中运行多个容器并共享文件时出现。Kubernetes [卷 \(Volume\)](#) 这一抽象概念能够解决这两个问题。

阅读本文前建议你熟悉一下 [Pods](#)。

背景

Docker 也有 [卷 \(Volume\)](#) 的概念，但对它只有少量且松散的管理。Docker 卷是磁盘上或者另外一个容器内的一个目录。Docker 提供卷驱动程序，但是其功能非常有限。

Kubernetes 支持很多类型的卷。 [Pod](#) 可以同时使用任意数目的卷类型。 临时卷类型的生命周期与 Pod 相同，但持久卷可以比 Pod 的存活期长。 因此，卷的存在时间会超出 Pod 中运行的所有容器，并且在容器重新启动时数据也会得到保留。 当 Pod 不再存在时，卷也将不再存在。

卷的核心是包含一些数据的一个目录，Pod 中的容器可以访问该目录。 所采用的特定的卷类型将决定该目录如何形成的、使用何种介质保存数据以及目录中存放 的内容。

使用卷时，在 `.spec.volumes` 字段中设置为 Pod 提供的卷，并在 `.spec.containers[*].volumeMounts` 字段中声明卷在容器中的挂载位置。 容器中的进程看到的是由它们的 Docker 镜像和卷组成的文件系统视图。 [Docker 镜像](#) 位于文件系统层次结构的根部。 各个卷则挂载在镜像内的指定路径上。 卷不能挂载到其他卷之上，也不能与其他卷有硬链接。 Pod 配置中的每个容器必须独立指定各个卷的挂载位置。

卷类型

Kubernetes 支持下列类型的卷：

awsElasticBlockStore

`awsElasticBlockStore` 卷将 Amazon Web服务（ AWS ）[EBS 卷](#) 挂载到你的 Pod 中。 与 `emptyDir` 在 Pod 被删除时也被删除不同，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，并且该数据可以在 Pod 之间共享。

说明： 你在使用 EBS 卷之前必须使用 `aws ec2 create-volume` 命令或者 AWS API 创建该卷。

使用 `awsElasticBlockStore` 卷时有一些限制：

- Pod 运行所在的节点必须是 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，你首先要创建它。

```
aws ec2 create-volume --availability-zone=eu-west-1a --size=10 --volume-type=gp2
```

确保该区域与你的群集所在的区域相匹配。还要检查卷的大小和 EBS 卷类型都适合你的用途。

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
      # 此 AWS EBS 卷必须已经存在
      awsElasticBlockStore:
        volumeID: "<volume-id>"
        fsType: ext4
```

AWS EBS CSI 卷迁移

FEATURE STATE: Kubernetes v1.17 [beta]

如果启用了对 awsElasticBlockStore 的 CSIMigration 特性支持，所有插件操作都 不再指向树内插件（In-Tree Plugin），转而指向 ebs.csi.aws.com 容器存储接口（Container Storage Interface，CSI）驱动。为了使用此特性，必须在集群中安装 [AWS EBS CSI 驱动](#)，并确保 CSIMigration 和 CSIMigrationAWS Beta 功能特性被启用。

AWS EBS CSI 迁移结束

FEATURE STATE: Kubernetes v1.17 [alpha]

如欲禁止 awsElasticBlockStore 存储插件被控制器管理器和 kubelet 组件加载，可将 CSIMigrationAWSComplete 特性门控设置为 true。此特性要求在 集群中所有工作节点上安装 ebs.csi.aws.com 容器存储接口驱动。

azureDisk

azureDisk 卷类型用来在 Pod 上挂载 Microsoft Azure [数据盘 \(Data Disk \)](#)。若需了解更多详情，请参考 [azureDisk 卷插件](#)。

azureDisk 的 CSI 迁移

FEATURE STATE: Kubernetes v1.19 [beta]

启用 azureDisk 的 CSIMigration 功能后，所有插件操作从现有的树内插件重定向到 disk.csi.azure.com 容器存储接口（CSI）驱动程序。为了使用此功能，必须在集群中安装 [Azure 磁盘 CSI 驱动程序](#)，并且 CSIMigration 和 CSIMigrationAzureDisk 功能必须被启用。

azureFile

azureFile 卷类型用来在 Pod 上挂载 Microsoft Azure 文件卷 (File Volume) (SMB 2.1 和 3.0)。更多详情请参考 [azureFile 卷插件](#)。

CSI 迁移

FEATURE STATE: Kubernetes v1.15 [alpha]

启用 azureFile 的 CSIMigration 功能后，所有插件操作将从现有的树内插件重定向到 file.csi.azure.com 容器存储接口 (CSI) 驱动程序。要使用此功能，必须在集群中安装 [Azure 文件 CSI 驱动程序](#)，并且 CSIMigration 和 CSIMigrationAzureFile Alpha 功能特性必须被启用。

cephfs

cephfs 卷允许你将现存的 CephFS 卷挂载到 Pod 中。不像 emptyDir 那样会在 Pod 被删除的同时也会被删除，cephfs 卷的内容在 Pod 被删除时会被保留，只是卷被卸载了。这意味着 cephfs 卷可以被预先填充数据，且这些数据可以在 Pod 之间共享。同一 cephfs 卷可同时被多个写者挂载。

说明：在使用 Ceph 卷之前，你的 Ceph 服务器必须已经运行并将要使用的 share 导出 (exported)。

更多信息请参考 [CephFS 示例](#)。

cinder

说明： Kubernetes 必须配置了 OpenStack Cloud Provider。

cinder 卷类型用于将 OpenStack Cinder 卷挂载到 Pod 中。

Cinder 卷示例配置

```
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
```

```
# 此 OpenStack 卷必须已经存在  
cinder:  
  volumeID: "<volume-id>"  
  fsType: ext4
```

OpenStack CSI 迁移

FEATURE STATE: Kubernetes v1.18 [beta]

启用 Cinder 的 CSIMigration 功能后，所有插件操作会从现有的树内插件重定向到 cinder.csi.openstack.org 容器存储接口 (CSI) 驱动程序。为了使用此功能，必须在集群中安装 [Openstack Cinder CSI 驱动程序](#)，并且 CSIMigration 和 CSIMigrationOpenStack Beta 功能必须被启用。

configMap

[configMap](#) 卷提供了向 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 Pod 中运行的容器化应用使用。

引用 configMap 对象时，你可以在 volume 中通过它的名称来引用。你可以自定义 ConfigMap 中特定条目所要使用的路径。下面的配置显示了如何将名为 log-config 的 ConfigMap 挂载到名为 configmap-pod 的 Pod 中：

```
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 以卷的形式挂载，并且存储在 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，这个路径来源于卷的 mount Path 和 log_level 键对应的 path。

说明：

- 在使用 [ConfigMap](#) 之前你首先要创建它。
- 容器以 [subPath](#) 卷挂载方式使用 ConfigMap 时，将无法接收 ConfigMap 的更新。
- 文本数据挂载成文件时采用 UTF-8 字符编码。如果使用其他字符编码形式，可使用 binaryData 字段。

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所请求的数据。

说明：容器以 [subPath](#) 卷挂载方式使用 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](#)。

emptyDir

当 Pod 分派到某个 Node 上时，emptyDir 卷会被创建，并且在 Pod 在该节点上运行期间，卷一直存在。就像其名称表示的那样，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，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 卷中的数据也会被永久删除。

说明：容器崩溃并不会导致 Pod 被从节点上移除，因此容器崩溃期间 empty Dir 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。

取决于你的环境，emptyDir 卷存储在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。但是，你可以将 emptyDir.medium 字段设置为 "Memory"，以告诉 Kubernetes 为你挂载 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，并且你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。

emptyDir 配置示例

```
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 卷类型允许将现有的光纤通道块存储卷挂载到 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 (World Wide Names)。如果指定了多个 WWN , targetWWNs 期望这些 WWN 来自多路径连接。

注意：你必须配置 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](#)。

flocker (已弃用)

[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端所支持的数据卷的管理和编排。

使用 flocker 卷可以将一个 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 之间共享。

说明：在使用 Flocker 之前你必须先安装运行自己的 Flocker。

更多详情请参考 [Flocker 示例](#)。

gcePersistentDisk

gcePersistentDisk 卷能将谷歌计算引擎 (GCE) [持久盘 \(PD \)](#) 挂载到你的 Pod 中。不像 emptyDir 那样会在 Pod 被删除的同时也会被删除 , 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 之间共享。

注意：在使用 PD 前 , 你必须使用 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
- 这些 VM 必须和持久盘位于相同的 GCE 项目和区域 (zone)

GCE 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。这意味着你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 GCE 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 GCE 持久盘 (PD)

在 Pod 中使用 GCE 持久盘之前，你首先要创建它。

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

GCE 持久盘配置示例

```
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
      # 此 GCE PD 必须已经存在
      gcePersistentDisk:
        pdName: my-data-disk
        fsType: ext4
```

区域持久盘

[区域持久盘](#) 功能允许你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，必须以持久卷 (PersistentVolume) 的方式提供卷；直接从 Pod 引用这种卷是不可以的。

手动供应基于区域 PD 的 PersistentVolume

使用[为 GCE PD 定义的存储类](#)可以实现动态供应。在创建 PersistentVolume 之前，你首先要创建 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
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: failure-domain.beta.kubernetes.io/zone
              operator: In
              values:
                - us-central1-a
                - us-central1-b
```

GCE CSI 迁移

FEATURE STATE: Kubernetes v1.17 [beta]

启用 GCE PD 的 CSIMigration 功能后，所有插件操作将从现有的树内插件重定向到 pd.csi.storage.gke.io 容器存储接口（CSI）驱动程序。为了使用此功能，必须在集群中上安装 [GCE PD CSI 驱动程序](#)，并且 CSIMigration 和 CSIMigrationGCE Beta 功能必须被启用。

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](#) 卷挂载到 InitContainer 中，使用 git 命令完成仓库的克隆操作，然后将 [EmptyDir](#) 卷挂载到 Pod 的容器中。

gitRepo 卷是一个卷插件的例子。该卷挂载一个空目录，并将一个 Git 代码仓库克隆到这个目录中供 Pod 使用。

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](#) (一个开源的网络文件系统) 挂载到你的 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。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 之间共享。GlusterFS 可以被多个写者同时挂载。

说明：在使用前你必须先安装运行自己的 GlusterFS。

更多详情请参考 [GlusterFS 示例](#)。

hostPath

hostPath 卷能将主机节点文件系统上的文件或目录挂载到你的 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 内部机制的容器；可使用 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 相同的组和属主信息。
Directory	在给定路径上必须存在的目录。

取值	行为
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（例如基于同一 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。
- 下层主机上创建的文件或目录只能由 root 用户写入。你需要在 [特权容器](#) 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。

hostPath 配置示例：

```
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:
        # 宿主上目录位置
        path: /data
        # 此字段为可选
      type: Directory
```

注意： FileOrCreate 模式不会负责创建文件的父目录。如果欲挂载的文件的父目录不存在，Pod 启动会失败。为了确保这种模式能够工作，可以尝试把文件和它对应的目录分开挂载，如 [FileOrCreate 配置](#) 所示。

hostPath FileOrCreate 配置示例

```
apiVersion: v1
kind: Pod
metadata:
  name: test-webserver
```

```
spec:  
  containers:  
    - name: test-webserver  
      image: k8s.gcr.io/test-webserver:latest  
      volumeMounts:  
        - mountPath: /var/local/aaa  
          name: mydir  
        - mountPath: /var/local/aaa/1.txt  
          name: myfile  
  volumes:  
    - name: mydir  
      hostPath:  
        # 确保文件所在目录成功创建。  
        path: /var/local/aaa  
        type: DirectoryOrCreate  
    - name: myfile  
      hostPath:  
        path: /var/local/aaa/1.txt  
        type: FileOrCreate
```

iscsi

iscsi 卷能将 iSCSI (基于 IP 的 SCSI) 卷挂载到你的 Pod 中。不像 emptyDir 那样会在删除 Pod 的同时也会被删除，iscsi 卷的内容在删除 Pod 时会被保留，卷只是被卸载。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 之间共享。

注意：在使用 iSCSI 卷之前，你必须拥有自己的 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。这意味着你可以用数据集预先填充卷，然后根据需要在尽可能多的 Pod 上使用它。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](#)。

local

local 卷所代表的是某个被挂载的本地存储设备，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。

与 hostPath 卷相比，local 卷能够以持久和可移植的方式使用，而无需手动将 Pod 调度到节点。系统通过查看 PersistentVolume 的节点亲和性配置，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 卷也将变得不可被 Pod 访问。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: 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，你需要设置 PersistentVolume 对象的 nodeAffinity 字段。

Kubernetes 调度器使用 PersistentVolume 的 nodeAffinity 信息来将使用 local 卷的 Pod 调度到正确的节点。

PersistentVolume 对象的 volumeMode 字段可被设置为 "Block"（而不是默认值 "Filesystem"），以将 local 卷作为原始块设备暴露出来。

使用 local 卷时，建议创建一个 StorageClass 并将其 volumeBindingMode 设置为 WaitForFirstConsumer。要了解更多详细信息，请参考 [local StorageClass 示例](#)。延迟卷绑定的操作可以确保 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。

你可以在 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，此驱动尚不支持动态配置。有关如何运行外部 local 卷驱动，请参考 [local 卷驱动用户指南](#)。

说明：如果不使用外部静态驱动来管理卷的生命周期，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) 挂载到你的 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。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 之间共享。

注意：在使用 NFS 卷之前，你必须运行自己的 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](#)。

persistentVolumeClaim

persistentVolumeClaim 卷用来将持久卷 (PersistentVolume) 挂载到 Pod 中。持久卷申领 (PersistentVolumeClaim) 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](#)。

portworxVolume

portworxVolume 是一个可伸缩的块存储层，能够以超融合 (hyperconverged) 的方式与 Kubernetes 一起运行。[Portworx](#) 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，也可以预先配备并在 Kubernetes Pod 内引用。下面是一个引用预先配备的 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
      # 此 Portworx 卷必须已经存在
      portworxVolume:
        volumeID: "pxvol"
        fsType: "<fs-type>"
```

说明：

在 Pod 中使用 portworxVolume 之前，你要确保有一个名为 pxvol 的 PortworxVolume 存在。

更多详情可以参考 [Portworx 卷](#)。

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](#)
- [downwardAPI](#)
- [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](#)。

包含 Secret、downwardAPI 和 configMap 的 Pod 示例

```
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
```

下面是一个带有非默认访问权限设置的多个 secret 的 Pod 示例：

```
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规约中的 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 只能在整个投射卷级别指定，而无法针对每个卷来源指定。不过，如上所述，你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 [服务帐号](#) 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。该令牌可以被 Pod 中的容器用来自访问 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时长。默认值为 1 小时，必须至少 10 分钟（600 秒）。管理员还可以通过设置 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。

说明：

使用投射卷源作为 [subPath](#) 卷挂载的容器将不会接收这些卷源的更新。

quobyte

quobyte 卷允许将现有的 [Quobyte](#) 卷挂载到你的 Pod 中。

说明：在使用 Quobyte 卷之前，你首先要进行安装 Quobyte 并创建好卷。

Quobyte 支持[容器存储接口（CSI）](#)。推荐使用 CSI 插件以在 Kubernetes 中使用 Quobyte 卷。Quobyte 的 GitHub 项目包含以 CSI 形式部署 Quobyte 的 [说明](#) 及使用示例。

rbd

rbd 卷允许将 [Rados 块设备](#) 卷挂载到你的 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。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 之间共享。

注意： 在使用 RBD 之前，你必须安装运行 Ceph。

RBD 的一个特性是它可以同时被多个用户以只读方式挂载。这意味着你可以用数据集预先填充卷，然后根据需要在尽可能多的 Pod 中并行地使用卷。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](#)。

scaleIO (已弃用)

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](#)）。

说明： 在使用前，你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](#)。

secret

secret 卷用来给 Pod 传递敏感信息，例如密码。你可以将 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。

说明： 使用前你必须在 Kubernetes API 中创建 secret。

说明： 容器以 [subPath](#) 卷挂载方式挂载 Secret 时，将感知不到 Secret 的更新。

更多详情请参考[配置 Secrets](#)。

storageOS

storageos 卷允许将现有的 [StorageOS](#) 卷挂载到你的 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的或挂接的存储。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。

注意： 你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](#)。

```
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:
# `redis-vol01` 卷必须在 StorageOS 中存在，并位于 `default` 名字空间内
volumeName: redis-vol01
fsType: ext4
```

关于 StorageOS 的进一步信息、动态供应和持久卷申领等等，请参考 [StorageOS 例子](#)。

vSphereVolume

说明：你必须配置 Kubernetes 的 vSphere 云驱动。云驱动的配置方法请参考 [vSphere 使用指南](#)。

vSphereVolume 用来将 vSphere VMDK 卷挂载到你的 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。

注意：在挂载到 Pod 之前，你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](#)
- [使用 vmware-vdiskmanager 创建](#)

首先 ssh 到 ESX，然后使用下面的命令来创建 VMDK：

```
vmkfstools -c 2G /vmfs/volumes/DatastoreName/volumes/myDisk.vmdk
```

使用下面的命令创建 VMDK：

```
vmware-vdiskmanager -c -t 0 -s 40GB -a lsilogic myDisk.vmdk
```

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  
  # 此 VMDK 卷必须已经存在  
  vsphereVolume:  
    volumePath: "[DatastoreName] volumes/myDisk"  
    fsType: ext4
```

进一步信息可参考 [vSphere 卷](#)。

vSphere CSI 迁移

FEATURE STATE: Kubernetes v1.19 [beta]

当 vsphereVolume 的 CSIMigration 特性被启用时，所有插件操作都被从树内插件重定向到 csi.vsphere.vmware.com [CSI](#) 驱动。为了使用此功能特性，必须在集群中安装 [vSphere CSI 驱动](#)，并启用 CSIMigration 和 CSIMigrationvSphere [特性门控](#)。

此特性还要求 vSphere vCenter/ESXi 的版本至少为 7.0u1，且 HW 版本至少为 VM version 15。

说明：

vSphere CSI 驱动不支持内置 vsphereVolume 的以下 StorageClass 参数：

- diskformat
- hostfailures tolerated
- forceprovisioning
- cachereservation
- diskstripes
- objectspacereservation
- iopslimit

使用这些参数创建的现有卷将被迁移到 vSphere CSI 驱动，不过使用 vSphere CSI 驱动所创建的新卷都不会理会这些参数。

vSphere CSI 迁移完成

FEATURE STATE: Kubernetes v1.19 [beta]

为了避免控制器管理器和 kubelet 加载 vsphereVolume 插件，你需要将 CSIMigration VSphereComplete 特性设置为 true。你还必须在所有工作节点上安装 csi.vsphere.vmware.com [CSI](#) 驱动。

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。

下面例子展示了如何配置某包含 LAMP 堆栈（Linux Apache MySQL PHP）的 Pod 使用同一共享卷。此示例中的 `subPath` 配置不建议在生产环境中使用。PHP 应用的代码和相关数据映射到卷的 `html` 文件夹，MySQL 数据库存储在卷的 `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 v1.17 [stable]

使用 `subPathExpr` 字段可以基于 Downward API 环境变量来构造 `subPath` 目录名。`subPath` 和 `subPathExpr` 属性是互斥的。

在这个示例中，Pod 使用 `subPathExpr` 来 hostPath 卷 `/var/log/pods` 中创建目录 `pod1`。hostPath 卷采用来自 downwardAPI 的 Pod 名称生成目录名。宿主目录 `/var/log/pods/pod1` 被挂载到容器的 `/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 数据的根目录（通常是 /var/lib/kubelet）的文件系统的介质确定。Kubernetes 对 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。

要了解如何使用资源规约来请求空间，可参考 [如何管理资源](#)。

树外 (Out-of-Tree) 卷插件

Out-of-Tree 卷插件包括 [容器存储接口 \(CSI \)](#) (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。

以前，所有卷插件（如上面列出的卷类型）都是“树内 (In-Tree) ”的。“树内”插件是与 Kubernetes 的核心组件一同构建、链接、编译和交付的。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。

对于希望创建树外 (Out-Of-Tree) 卷插件的存储供应商，请参考 [卷插件常见问题](#)。

CSI

[容器存储接口](#) (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](#)。

说明：Kubernetes v1.13 废弃了对 CSI 规范版本 0.2 和 0.3 的支持，并将在以后的版本中删除。

说明：CSI 驱动可能并非兼容所有的 Kubernetes 版本。请查看特定 CSI 驱动的文档，以了解各个 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 卷类型来挂接、挂载 CSI 驱动所提供的卷。

csi 卷可以在 Pod 中以三种方式使用：

- 通过 PersistentVolumeClaim (#persistentvolumeclaim) 对象引用
- 使用[一般性的临时卷](#) (Alpha 特性)
- 使用[CSI 临时卷](#)，前提是驱动支持这种用法 (Beta 特性)

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](#) 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 驱动在 CreateVolumeResponse 的 volume_id 字段中返回的值相对应；接口定义在 [CSI spec](#) 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。
- 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](#) 中有相应的定义。该映射通过 ControllerPublishVo

lumeRequest、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 条目，则所有的 Secret 条目都会被传递。
- nodeStageSecretRef：对包含敏感信息的 Secret 对象的引用。该信息会传递给 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 条目，则传递所有 Secret 条目。
- nodePublishSecretRef：对包含敏感信息的 Secret 对象的引用。该信息传递给 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 条目，则传递所有 Secret 条目。

CSI 原始块卷支持

FEATURE STATE: Kubernetes v1.18 [stable]

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。

你可以和以前一样，安装自己的 [带有原始块卷支持的 PV/PVC](#)，采用 CSI 对此过程没有影响。

CSI 临时卷

FEATURE STATE: Kubernetes v1.16 [beta]

你可以直接在 Pod 规约中配置 CSI 卷。采用这种方式配置的卷都是临时卷，无法在 Pod 重新启动后继续存在。进一步的信息可参阅[临时卷](#)。

有关如何开发 CSI 驱动的更多信息，请参考 [kubernetes-csi 文档](#)。

从树内插件迁移到 CSI 驱动程序

FEATURE STATE: Kubernetes v1.17 [beta]

启用 CSIMigration 功能后，针对现有树内插件的操作会被重定向到相应的 CSI 插件（应已安装和配置）。因此，操作员在过渡期到取代树内插件的 CSI 驱动时，无需对现有存储类、PV 或 PVC（指树内插件）进行任何配置更改。

所支持的操作和功能包括：配备（Provisioning）/删除、挂接（Attach）/解挂（Detach）、挂载（Mount）/卸载（Unmount）和调整卷大小。

上面的[卷类型](#)节列出了支持 CSIMigration 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 中一直存在的树外插件接口。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（在某些情况下是主控节点）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 树内插件与 Flexvolume 驱动程序交互。更多详情请参考 [FlexVolume 示例](#)。

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](#) 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](#) 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](#) 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。强烈建议你熟悉 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。

编辑你的 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](#)示例。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#). 在 GitHub 仓库上登记新的问题[报告问题](#) 或者[提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

卷快照

FEATURE STATE: Kubernetes 1.17 [beta]

在 Kubernetes 中，卷快照是一个存储系统上卷的快照，本文假设你已经熟悉了 Kubernetes 的[持久卷](#)。

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 是一种快照，从管理员已提供的集群中的卷获取。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。

VolumeSnapshotClass 允许指定属于 VolumeSnapshot 的不同属性。在从存储系统的相同卷上获取的快照之间，这些属性可能有所不同，因此不能通过使用与 PersistentVolumeClaim 相同的 StorageClass 来表示。

卷快照能力为 Kubernetes 用户提供了一种标准的方式来在指定时间点 复制卷的内容， 并且不需要创建全新的卷。例如，这一功能使得数据库管理员 能够在执行编辑或删除之类的修改之前对数据库执行备份。

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 [CRDs](#)， 不属于核心 API。
- VolumeSnapshot 支持仅可用于 CSI 驱动。
- 作为 VolumeSnapshot 部署过程的一部分，Kubernetes 团队提供了一个部署于控制平面的快照控制器， 并且提供了一个叫做 csi-snapshotter 的边车 (Sidecar) 辅助容器，和 CSI 驱动程序一起部署。快照控制器监视 VolumeSnapshot 和 VolumeSnapshotContent 对象， 并且负责创建和删除 VolumeSnapshotContent 对象。边车 csi-snapshotter 监视 VolumeSnapshotContent 对象， 并且触发针对 CSI 端点的 CreateSnapshot 和 DeleteSnapshot 的操作。
- 还有一个验证性质的 Webhook 服务器，可以对快照对象进行更严格的验证。Kubernetes 发行版应将其与快照控制器和 CRD (而非 CSI 驱动程序) 一起安装。此服务器应该安装在所有启用了快照功能的 Kubernetes 集群中。
- 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 csi-snapshotter 来提供对卷快照的支持。详见 [CSI 驱动程序文档](#)
- Kubernetes 负责 CRDs 和快照控制器的安装。

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配置：预配置或动态配置。

预配置

集群管理员创建多个 VolumeSnapshotContents。它们带有存储系统上实际卷快照的详细信息，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

可以从 PersistentVolumeClaim 中动态获取快照，而不用使用已经存在的快照。在获取快照时，[卷快照类](#) 指定要用的特定于存储提供程序的参数。

绑定

在预配置和动态配置场景下，快照控制器处理绑定 VolumeSnapshot 对象和其合适的 VolumeSnapshotContent 对象。绑定关系是一对一的。

在预配置快照绑定场景下，VolumeSnapshotContent 对象创建之后，才会和 VolumeSnapshot 进行绑定。

快照源的持久性卷声明保护

这种保护的目的是确保在从系统中获取快照时，不会将正在使用的 [PersistentVolumeClaim API](#) 对象从系统中删除（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。

当从 PersistentVolumeClaim 中生成快照时，PersistentVolumeClaim 就在被使用了。如果删除一个作为快照源的 PersistentVolumeClaim 对象，这个 PersistentVolumeClaim 对象不会立即被删除的。相反，删除 PersistentVolumeClaim 对象的动作会被放弃，或者推迟到快照的 Status 为 ReadyToUse 时再执行。

删除

删除 VolumeSnapshot 对象触发删除 VolumeSnapshotContent 操作，并且 Deletion Policy 会紧跟着执行。如果 DeletionPolicy 是 Delete，那么底层存储快照会和 Volume SnapshotContent 一起被删除。如果 DeletionPolicy 是 Retain，那么底层快照和 VolumeSnapshotContent 都会被保留。

卷快照

每个 VolumeSnapshot 包含一个 spec 和一个状态。

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

persistentVolumeClaimName 是 PersistentVolumeClaim 数据源对快照的名称。这个字段是动态配置快照中的必填字段。

卷快照可以通过指定 [VolumeSnapshotClass](#) 使用 volumeSnapshotClassName 属性来请求特定类。如果没有设置，那么使用默认类（如果有）。

如下面例子所示，对于预配置的快照，需要给快照指定 volumeSnapshotContentName 来作为源。对于预配置的快照 source 中的 volumeSnapshotContentName 字段是必填的。

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: test-snapshot
spec:
  source:
    volumeSnapshotContentName: test-content
```

每个 VolumeSnapshotContent 对象包含 spec 和 status。在动态配置时，快照通用控制器创建 VolumeSnapshotContent 对象。下面是例子：

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: snapcontent-72d9a349-aacd-42d2-a240-d775650d2455
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    volumeHandle: ee0cfb94-f8d4-11e9-b2d8-0242ac110002
    volumeSnapshotClassName: csi-hostpath-snapclass
    volumeSnapshotRef:
      name: new-snapshot-test
      namespace: default
      uid: 72d9a349-aacd-42d2-a240-d775650d2455
```

volumeHandle 是存储后端创建卷的唯一标识符，在卷创建期间由 CSI 驱动程序返回。动态设置快照需要此字段。它指出了快照的卷源。

对于预配置快照，你（作为集群管理员）要按如下命令来创建 VolumeSnapshotContent 对象。

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotContent
metadata:
  name: new-snapshot-content-test
spec:
  deletionPolicy: Delete
  driver: hostpath.csi.k8s.io
  source:
    snapshotHandle: 7bdd0de3-aaeb-11e8-9aae-0242ac110002
    volumeSnapshotRef:
      name: new-snapshot-test
      namespace: default
```

snapshotHandle 是存储后端创建卷的唯一标识符。对于预设置快照，这个字段是必须的。它指定此 VolumeSnapshotContent 表示的存储系统上的 CSI 快照 id。

从快照供应卷

你可以配置一个新卷，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](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 18, 2020 at 8:36 AM PST: [Update volume-snapshots.md \(e3e199f41\)](#)

持久卷

本文描述 Kubernetes 中 *持久卷 (Persistent Volume)* 的当前状态。建议先熟悉[卷 \(Volume\)](#)的概念。

介绍

存储的管理是一个与计算实例的管理完全不同的问题。PersistentVolume 子系统为用户和管理员提供了一组 API，将存储如何供应的细节从其如何被使用中抽象出来。为了实现这点，我们引入了两个新的 API 资源：PersistentVolume 和 PersistentVolumeClaim。

持久卷 (PersistentVolume , PV) 是集群中的一块存储，可以由管理员事先供应，或者使用[存储类 \(Storage Class\)](#) 来动态供应。持久卷是集群资源，就像节点也是集群资源一样。PV 持久卷和普通的 Volume 一样，也是使用卷插件来实现的，只是它们拥有独立于任何使用 PV 的 Pod 的生命周期。此 API 对象中记述了存储的实现细节，无论其背后是 NFS、iSCSI 还是特定于云平台的存储系统。

持久卷申领 (PersistentVolumeClaim , PVC) 表达的是用户对存储的请求。概念上与 Pod 类似。Pod 会耗用节点资源，而 PVC 申领会耗用 PV 资源。Pod 可以请求特定数量的资源 (CPU 和内存)；同样 PVC 申领也可以请求特定的大小和访问模式 (例如，可以要求 PV 卷能够以 ReadWriteOnce、ReadOnlyMany 或 ReadWriteMany 模式之一来挂载，参见[访问模式](#))。

尽管 PersistentVolumeClaim 允许用户消耗抽象的存储资源，常见的原因是针对不同的问题用户需要的是具有不同属性 (如，性能) 的 PersistentVolume 卷。集群管理员需要能够提供不同性质的 PersistentVolume，并且这些 PV 卷之间的差别不仅限于卷大小和访问模式，同时又不能将卷是如何实现的这些细节暴露给用户。为了满足这类需求，就有了[存储类 \(StorageClass\)](#) 资源。

[参见基于运行示例的详细演练。](#)

卷和申领的生命周期

PV 卷是集群中的资源。PVC 申领是对这些资源的请求，也被用来执行对资源的申领检查。PV 卷和 PVC 申领之间的互动遵循如下生命周期：

供应

PV 卷的供应有两种方式：静态供应或动态供应。

静态供应

集群管理员创建若干 PV 卷。这些卷对象带有真实存储的细节信息，并且对集群 用户可用（可见）。PV 卷对象存在于 Kubernetes API 中，可供用户消费（使用）。

动态供应

如果管理员所创建的所有静态 PV 卷都无法与用户的 PersistentVolumeClaim 匹配，集群可以尝试为该 PVC 申领动态供应一个存储卷。这一供应操作是基于 StorageClass 来实现的：PVC 申领必须请求某个 [存储类](#)，同时集群管理员必须 已经创建并配置了该类，这样动态供应卷的动作才会发生。如果 PVC 申领指定存储类为 ""，则相当于为自身禁止使用动态供应的卷。

为了基于存储类完成动态的存储供应，集群管理员需要在 API 服务器上启用 DefaultStorageClass [准入控制器](#)。举例而言，可以通过保证 DefaultStorageClass 出现在 API 服务器组件的 --enable-admission-plugins 标志值中实现这点；该标志的值可以是逗号分隔的有序列表。关于 API 服务器标志的更多信息，可以参考 [kube-apiserver](#) 文档。

绑定

用户创建一个带有特定存储容量和特定访问模式需求的 PersistentVolumeClaim 对象；在动态供应场景下，这个 PVC 对象可能已经创建完毕。主控节点中的控制回路监测新的 PVC 对象，寻找与之匹配的 PV 卷（如果可能的话），并将二者绑定到一起。如果为了新的 PVC 申领动态供应了 PV 卷，则控制回路总是将该 PV 卷绑定到这一 PVC 申领。否则，用户总是能够获得他们所请求的资源，只是所获得的 PV 卷可能会超出所请求的配置。一旦绑定关系建立，则 PersistentVolumeClaim 绑定就是排他性的，无论该 PVC 申领是如何与 PV 卷建立的绑定关系。PVC 申领与 PV 卷之间的绑定是一种一对一的映射，实现上使用 ClaimRef 来记述 PV 卷与 PVC 申领间的双向绑定关系。

如果找不到匹配的 PV 卷，PVC 申领会无限期地处于未绑定状态。当与之匹配的 PV 卷可用时，PVC 申领会被绑定。例如，即使某集群上供应了很多 50 Gi 大小的 PV 卷，也无法与请求 100 Gi 大小的存储的 PVC 匹配。当新的 100 Gi PV 卷被加入到集群时，该 PVC 才有可能被绑定。

使用

Pod 将 PVC 申领当做存储卷来使用。集群会检视 PVC 申领，找到所绑定的卷，并为 Pod 挂载该卷。对于支持多种访问模式的卷，用户要在 Pod 中以卷的形式使用申领时指定期望的访问模式。

一旦用户有了申领对象并且该申领已经被绑定，则所绑定的 PV 卷在用户仍然需要它期间一直属于该用户。用户通过在 Pod 的 volumes 块中包含 persistentVolumeClaim 节区来调度 Pod，访问所申领的 PV 卷。相关细节可参阅[使用申领作为卷](#)。

保护使用中的存储对象

保护使用中的存储对象（Storage Object in Use Protection）这一功能特性的目的 是确保仍被 Pod 使用的 PersistentVolumeClaim（PVC）对象及其所绑定的 PersistentVolume（PV）对象在系统中不会被删除，因为这样做可能会引起数据丢失。

说明：当使用某 PVC 的 Pod 对象仍然存在时，认为该 PVC 仍被此 Pod 使用。

如果用户删除被某 Pod 使用的 PVC 对象，该 PVC 申领不会被立即移除。PVC 对象的移除会被推迟，直至其不再被任何 Pod 使用。此外，如果管理员删除已绑定到某 PVC 申领的 PV 卷，该 PV 卷也不会被立即移除。PV 对象的移除也要推迟到该 PV 不再绑定到 PVC。

你可以看到当 PVC 的状态为 Terminating 且其 Finalizers 列表中包含 kubernetes.io/pvc-protection 时，PVC 对象是处于被保护状态的。

```
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]
...
```

你也可以看到当 PV 对象的状态为 Terminating 且其 Finalizers 列表中包含 kubernetes.io/pv-protection 时，PV 对象是处于被保护状态的。

```
kubectl describe pv task-pv-volume
```

```
Name:      task-pv-volume
Labels:    type=local
Annotations: <none>
Finalizers: [kubernetes.io/pv-protection]
```

```
StorageClass: standard
Status: Terminating
Claim:
Reclaim Policy: Delete
Access Modes: RWO
Capacity: 1Gi
Message:
Source:
  Type: HostPath (bare host directory volume)
  Path: /tmp/data
  HostPathType:
Events: <none>
```

回收

当用户不再使用其存储卷时，他们可以从 API 中将 PVC 对象删除，从而允许该资源被回收再利用。PersistentVolume 对象的回收策略告诉集群，当其被从申领中释放时如何处理该数据卷。目前，数据卷可以被 Retained (保留)、Recycled (回收) 或 Deleted (删除)。

保留 (Retain)

回收策略 Retain 使得用户可以手动回收资源。当 PersistentVolumeClaim 对象被删除时，PersistentVolume 卷仍然存在，对应的数据卷被视为“已释放 (released)”。由于卷上仍然存在这前一申领人的数据，该卷还不能用于其他申领。管理员可以通过下面的步骤来手动回收该卷：

1. 删除 PersistentVolume 对象。与之相关的、位于外部基础设施中的存储资产（例如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）在 PV 删除之后仍然存在。
2. 根据情况，手动清除所关联的存储资产上的数据。
3. 手动删除所关联的存储资产；如果你希望重用该存储资产，可以基于存储资产的定义创建新的 PersistentVolume 卷对象。

删除 (Delete)

对于支持 Delete 回收策略的卷插件，删除动作会将 PersistentVolume 对象从 Kubernetes 中移除，同时也会从外部基础设施（如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）中移除所关联的存储资产。动态供应的卷会继承[其 StorageClass 中设置的回收策略](#)，该策略默认为 Delete。管理员需要根据用户的期望来配置 StorageClass；否则 PV 卷被创建之后必须要被编辑或者修补。参阅[更改 PV 卷的回收策略](#)。

回收 (Recycle)

警告：回收策略 Recycle 已被废弃。取而代之的建议方案是使用动态供应。

如果下层的卷插件支持，回收策略 Recycle 会在卷上执行一些基本的擦除（`rm -rf /thevolume/*`）操作，之后允许该卷用于新的 PVC 申领。

不过，管理员可以按 [参考资料](#) 中所述，使用 Kubernetes 控制器管理器命令行参数来配置一个定制的回收器（Recycler）Pod 模板。此定制的回收器 Pod 模板必须包含一个 volumes 规约，如下例所示：

```
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
```

定制回收器 Pod 模板中在 volumes 部分所指定的特定路径要替换为 正被回收的卷的路径。

预留 PersistentVolume

通过在 PersistentVolumeClaim 中指定 PersistentVolume，你可以声明该特定 PV 与 PVC 之间的绑定关系。如果该 PersistentVolume 存在且未被通过其 `claimRef` 字段预留给 PersistentVolumeClaim，则该 PersistentVolume 会和该 PersistentVolumeClaim 绑定到一起。

绑定操作不会考虑某些卷匹配条件是否满足，包括节点亲和性等等。控制面仍然会检查 [存储类](#)、访问模式和所请求的 存储尺寸都是合法的。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: foo-pvc
  namespace: foo
spec:
  storageClassName: "" # 此处须显式设置空字符串，否则会被设置为默认的StorageClass
```

```
volumeName: foo-pv
```

```
...
```

此方法无法对 PersistentVolume 的绑定特权做出任何形式的保证。如果有其他 PersistentVolumeClaim 可以使用你所指定的 PV，则你应该首先预留 该存储卷。你可以将 PV 的 claimRef 字段设置为相关的 PersistentVolumeClaim 以确保其他 PVC 不会绑定到该 PV 卷。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: foo-pv
spec:
  storageClassName: ""
  claimRef
    name: foo-pvc
    namespace: foo
```

```
...
```

如果你想要使用 claimPolicy 属性设置为 Retain 的 PersistentVolume 卷时，包括你希望复用现有的 PV 卷时，这点是很有用的

扩充 PVC 申领

FEATURE STATE: Kubernetes v1.11 [beta]

现在，对扩充 PVC 申领的支持默认处于被启用状态。你可以扩充以下类型的卷：

- gcePersistentDisk
- awsElasticBlockStore
- Cinder
- glusterfs
- rbd
- Azure File
- Azure Disk
- Portworx
- FlexVolumes [CSI](#)

只有当 PVC 的存储类中将 allowVolumeExpansion 设置为 true 时，你才可以扩充该 PVC 申领。

```
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
```

如果要为某 PVC 请求较大的存储卷，可以编辑 PVC 对象，设置一个更大的尺寸值。这一编辑操作会触发为下层 PersistentVolume 提供存储的卷的扩充。Kubernetes 不会创建新的 PV 卷来满足此申领的请求。与之相反，现有的卷会被调整大小。

CSI 卷的扩充

FEATURE STATE: Kubernetes v1.16 [beta]

对 CSI 卷的扩充能力默认是被启用的，不过扩充 CSI 卷要求 CSI 驱动支持 卷扩充操作。可参阅特定 CSI 驱动的文档了解更多信息。

重设包含文件系统的卷的大小

只有卷中包含的文件系统是 XFS、Ext3 或者 Ext4 时，你才可以重设卷的大小。

当卷中包含文件系统时，只有在 Pod 使用 ReadWrite 模式来使用 PVC 申领的情况下才能重设其文件系统的大小。文件系统扩充的操作或者是在 Pod 启动期间完成，或者在下层文件系统支持在线 扩充的前提下在 Pod 运行期间完成。

如果 FlexVolumes 的驱动将 RequiresFSResize 能力设置为 true，则该 FlexVolume 卷可以在 Pod 重启期间调整大小。

重设使用中 PVC 申领的大小

FEATURE STATE: Kubernetes v1.15 [beta]

说明： Kubernetes 从 1.15 版本开始将调整使用中 PVC 申领大小这一能力作为 Beta 特性支持；该特性在 1.11 版本以来处于 Alpha 阶段。ExpandInUsePersistentVolumes 特性必须被启用；在很多集群上，与此类似的 Beta 阶段的特性是自动启用的。可参考[特性门控](#) 文档了解更多信息。

在这种情况下，你不需要删除和重建正在使用某现有 PVC 的 Pod 或 Deployment。所有使用中的 PVC 在其文件系统被扩充之后，立即可供其 Pod 使用。此功能特性对于没有被 Pod 或 Deployment 使用的 PVC 而言没有效果。你必须在执行扩展操作之前创建一个使用该 PVC 的 Pod。

与其他卷类型类似，FlexVolume 卷也可以在被 Pod 使用期间执行扩充操作。

说明： FlexVolume 卷的重设大小只能在下层驱动支持重设大小的时候才可进行。

说明： 扩充 EBS 卷的操作非常耗时。同时还存在另一个配额限制：每 6 小时只能执行一次（尺寸）修改操作。

处理扩充卷过程中的失败

如果扩充下层存储的操作失败，集群管理员可以手动地恢复 PVC 申领的状态并取消重设大小的请求。否则，在没有管理员干预的情况下，控制器会反复重试 重设大小的操作。

1. 将绑定到 PVC 申领的 PV 卷标记为 Retain 回收策略；
2. 删除 PVC 对象。由于 PV 的回收策略为 Retain，我们不会在重建 PVC 时丢失数据。
3. 删除 PV 规约中的 claimRef 项，这样新的 PVC 可以绑定到该卷。这一操作会使 PV 卷变为 "可用 (Available) "。
4. 使用小于 PV 卷大小的尺寸重建 PVC，设置 PVC 的 volumeName 字段为 PV 卷的名称。这一操作将把新的 PVC 对象绑定到现有的 PV 卷。
5. 不要忘记恢复 PV 卷上设置的回收策略。

持久卷的类型

PV 持久卷是用插件的形式来实现的。Kubernetes 目前支持以下插件：

- [awsElasticBlockStore](#) - AWS 弹性块存储 (EBS)
- [azureDisk](#) - Azure Disk
- [azureFile](#) - Azure File
- [cephfs](#) - CephFS volume
- [cinder](#) - Cinder (OpenStack 块存储) (**弃用**)
- [csi](#) - 容器存储接口 (CSI)
- [fc](#) - Fibre Channel (FC) 存储
- [flexVolume](#) - FlexVolume
- [flocker](#) - Flocker 存储
- [gcePersistentDisk](#) - GCE 持久化盘
- [glusterfs](#) - Glusterfs 卷
- [hostPath](#) - HostPath 卷 (仅供单节点测试使用；不适用于多节点集群；请尝试使用 local 卷作为替代)
- [iscsi](#) - iSCSI (SCSI over IP) 存储
- [local](#) - 节点上挂载的本地存储设备
- [nfs](#) - 网络文件系统 (NFS) 存储
- [photonPersistentDisk](#) - Photon 控制器持久化盘。 (这个卷类型已经因对应的云提供商被移除而被弃用)。
- [portworxVolume](#) - Portworx 卷
- [quobyte](#) - Quobyte 卷
- [rbd](#) - Rados 块设备 (RBD) 卷
- [scaleIO](#) - ScaleIO 卷 (**弃用**)
- [storageos](#) - StorageOS 卷
- [vsphereVolume](#) - vSphere VMDK 卷

持久卷

每个 PV 对象都包含 spec 部分和 status 部分，分别对应卷的规约和状态。 PersistentVolume 对象的名称必须是合法的 [DNS 子域名](#)。

```
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
```

说明：在集群中使用持久卷存储通常需要一些特定于具体卷类型的辅助程序。在这个例子中，PersistentVolume 是 NFS 类型的，因此需要辅助程序 /sbin/mount.nfs 来支持挂载 NFS 文件系统。

容量

一般而言，每个 PV 卷都有确定的存储容量。容量属性是使用 PV 对象的 capacity 属性来设置的。参考 Kubernetes [资源模型 \(Resource Model\)](#) 设计提案，了解 capacity 字段可以接受的单位。

目前，存储大小是可以设置和请求的唯一资源。未来可能会包含 IOPS、吞吐量等属性。

卷模式

FEATURE STATE: Kubernetes v1.18 [stable]

针对 PV 持久卷，Kubernetes 支持两种卷模式 (volumeModes)：Filesystem (文件系统) 和 Block (块)。volumeMode 是一个可选的 API 参数。如果该参数被省略，默认的卷模式是 Filesystem。

volumeMode 属性设置为 Filesystem 的卷会被 Pod 挂载 (Mount) 到某个目录。如果卷的存储来自某块设备而该设备目前为空，Kubernetes 会在第一次挂载卷之前在设备上创建文件系统。

你可以将 volumeMode 设置为 Block，以便将卷作为原始块设备来使用。这类卷以块设备的方式交给 Pod 使用，其上没有任何文件系统。这种模式对于为 Pod 提供一种使用最快可能方式来访问卷而言很有帮助，Pod 和 卷之间不存在文件系统层。另外，Pod 中运行的应用必须知道如何处理原始块设备。关于如何在 Pod 中使用 volumeMode: Block 的卷，可参阅 [原始块卷支持](#)。

访问模式

PersistentVolume 卷可以用资源提供者所支持的任何方式挂载到宿主系统上。如下表所示，提供者（驱动）的能力不同，每个 PV 卷的访问模式都会设置为 对应卷所支持的模式值。例如，NFS 可以支持多个读写客户，但是某个特定的 NFS PV 卷可能在服务器上以只读的方式导出。每个 PV 卷都会获得自身的访问模式集合，描述的是 特定 PV 卷的能力。

访问模式有：

- ReadWriteOnce -- 卷可以被一个节点以读写方式挂载；
- ReadOnlyMany -- 卷可以被多个节点以只读方式挂载；
- ReadWriteMany -- 卷可以被多个节点以读写方式挂载。

在命令行接口（CLI）中，访问模式也使用以下缩写形式：

- RWO - ReadWriteOnce
- ROX - ReadOnlyMany
- RWX - ReadWriteMany

重要提醒！ 每个卷只能同一时刻只能以一种访问模式挂载，即使该卷能够支持 多种访问模式。例如，一个 GCEPersistentDisk 卷可以被某节点以 ReadWriteOnce 模式挂载，或者被多个节点以 ReadOnlyMany 模式挂载，但不可以同时以两种模式 挂载。

卷插件	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWSElasticBlockStore	✓	-	-
AzureFile	✓	✓	✓
AzureDisk	✓	-	-
CephFS	✓	✓	✓
Cinder	✓	-	-
CSI	取决于驱动	取决于驱动	取决于驱动
FC	✓	✓	-
FlexVolume	✓	✓	取决于驱动
Flocker	✓	-	-
GCEPersistentDisk	✓	✓	-
Glusterfs	✓	✓	✓
HostPath	✓	-	-
iSCSI	✓	✓	-
Quobyte	✓	✓	✓
NFS	✓	✓	✓
RBD	✓	✓	-

卷插件	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
VsphereVolume	✓	-	- (Pod 运行于同一节点上时可行)
PortworxVolume	✓	-	✓
ScaleIO	✓	✓	-
StorageOS	✓	-	-

类

每个 PV 可以属于某个类 (Class) , 通过将其 storageClassName 属性设置为某个 [StorageClass](#) 的名称来指定。 特定类的 PV 卷只能绑定到请求该类存储卷的 PVC 申领。 未设置 storageClassName 的 PV 卷没有类设定 , 只能绑定到那些没有指定特定 存储类的 PVC 申领。

早前 , Kubernetes 使用注解 `volume.beta.kubernetes.io/storage-class` 而不是 `storageClassName` 属性。 这一注解目前仍然起作用 , 不过在将来的 Kubernetes 发布版本 中该注解会被彻底废弃。

回收策略

目前的回收策略有 :

- Retain -- 手动回收
- Recycle -- 基本擦除 (`rm -rf /thevolume/*`)
- Delete -- 诸如 AWS EBS、GCE PD、Azure Disk 或 OpenStack Cinder 卷这 类关联存储资产也被删除

目前 , 仅 NFS 和 HostPath 支持回收 (Recycle) 。 AWS EBS、GCE PD、 Azure Disk 和 Cinder 卷都支持删除 (Delete) 。

挂载选项

Kubernetes 管理员可以指定持久卷被挂载到节点上时使用的附加挂载选项。

说明 : 并非所有持久卷类型都支持挂载选项。

以下卷类型支持挂载选项 :

- AWSElasticBlockStore
- AzureDisk
- AzureFile
- CephFS
- Cinder (OpenStack 块存储)
- GCEPersistentDisk
- Glusterfs
- NFS
- Quobyte 卷
- RBD (Ceph 块设备)
- StorageOS
- VsphereVolume

- iSCSI

Kubernetes 不对挂载选项执行合法性检查，因此非法的挂载选项只是会导致挂载失败。

早前，Kubernetes 使用注解 `volume.beta.kubernetes.io/mount-options` 而不是 `mountOptions` 属性。这一注解目前仍然起作用，不过在将来的 Kubernetes 发布版本中该注解会被彻底废弃。

节点亲和性

每个 PV 卷可以通过设置 [节点亲和性](#) 来定义一些约束，进而限制从哪些节点上可以访问此卷。使用这些卷的 Pod 只会被调度到节点亲和性规则所选择的节点上执行。

说明： 对大多数类型的卷而言，你不需要设置节点亲和性字段。[AWS EBS](#)、[GCE PD](#) 和 [Azure Disk](#) 卷类型都能自动设置相关字段。你需要为 [local](#) 卷显式地设置此属性。

阶段

每个卷会处于以下阶段（Phase）之一：

- Available (可用) -- 卷是一个空闲资源，尚未绑定到任何申领；
- Bound (已绑定) -- 该卷已经绑定到某申领；
- Released (已释放) -- 所绑定的申领已被删除，但是资源尚未被集群回收；
- Failed (失败) -- 卷的自动回收操作失败。

命令行接口能够显示绑定到某 PV 卷的 PVC 对象。

PersistentVolumeClaims

每个 PVC 对象都有 spec 和 status 部分，分别对应申领的规约和状态。PersistentVolumeClaim 对象的名称必须是合法的 [DNS 子域名](#)。

```
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]}

访问模式

申领在请求具有特定访问模式的存储时，使用与卷相同的访问模式约定。

卷模式

申领使用与卷相同的约定来表明是将卷作为文件系统还是块设备来使用。

资源

申领和 Pod 一样，也可以请求特定数量的资源。在这个上下文中，请求的资源是存储。卷和申领都使用相同的 [资源模型](#)。

选择算符

申领可以设置[标签选择算符](#) 来进一步过滤卷集合。只有标签与选择算符相匹配的卷能够绑定到申领上。选择算符包含两个字段：

- matchLabels - 卷必须包含带有此值的标签
- matchExpressions - 通过设定键 (key)、值列表和操作符 (operator) 来构造的需求。合法的操作符有 In、NotIn、Exists 和 DoesNotExist。

来自 matchLabels 和 matchExpressions 的所有需求都按逻辑与的方式组合在一起。这些需求都必须被满足才被视为匹配。

类

申领可以通过为 storageClassName 属性设置 [StorageClass](#) 的名称来请求特定的存储类。只有所请求的类的 PV 卷，即 storageClassName 值与 PVC 设置相同的 PV 卷，才能绑定到 PVC 申领。

PVC 申领不必一定要请求某个类。如果 PVC 的 storageClassName 属性值设置为 ""，则被视为要请求的是没有设置存储类的 PV 卷，因此这一 PVC 申领只能绑定到未设置存储类的 PV 卷（未设置注解或者注解值为 "" 的 PersistentVolume (PV) 对象在系统中不会被删除，因为这样做可能会引起数据丢失。未设置 storageClassName 的 PVC 与此大不相同，也会被集群作不同处理。具体筛查方式取决于 [DefaultStorageClass 准入控制器插件](#) 是否被启用。

- 如果准入控制器插件被启用，则管理员可以设置一个默认的 StorageClass。所有未设置 storageClassName 的 PVC 都只能绑定到隶属于默认存储类的 PV 卷。设置默认 StorageClass 的工作是通过将对应 StorageClass 对象的注解 storageclass.kubernetes.io/is-default-class 赋值为 true 来完成的。如果管理员未设置默认存储类，集群对 PVC 创建的处理方式与未启用准入控制器插件时相同。如果设置的默认存储类不止一个，准入控制插件会禁止所有创建 PVC 操作。

- 如果准入控制器插件被关闭，则不存在默认 StorageClass 的说法。所有未设置 storageClassName 的 PVC 都只能绑定到未设置存储类的 PV 卷。在这种情况下，未设置 storageClassName 的 PVC 与 storageClassName 设置为 "" 的 PVC 的处理方式相同。

取决于安装方法，默认的 StorageClass 可能在集群安装期间由插件管理器（Addon Manager）部署到集群中。

当某 PVC 除了请求 StorageClass 之外还设置了 selector，则这两种需求会按逻辑与关系处理：只有隶属于所请求类且带有所请求标签的 PV 才能绑定到 PVC。

说明：目前，设置了非空 selector 的 PVC 对象无法让集群为其动态供应 PV 卷。

早前，Kubernetes 使用注解 volume.beta.kubernetes.io/storage-class 而不是 storageClassName 属性。这一注解目前仍然起作用，不过在将来的 Kubernetes 发布版本中该注解会被彻底废弃。

使用申领作为卷

Pod 将申领作为卷来使用，并藉此访问存储资源。申领必须位于使用它的 Pod 所在的同一名字空间内。集群在 Pod 的名字空间中查找申领，并使用它来获得申领所使用的 PV 卷。之后，卷会被挂载到宿主上并挂载到 Pod 中。

```
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
```

关于名字空间的说明

PersistentVolume 卷的绑定是排他性的。由于 PersistentVolumeClaim 是名字空间作用域的对象，使用 "Many" 模式（ROX、RWX）来挂载申领的操作只能在同一名字空间内进行。

原始块卷支持

FEATURE STATE: Kubernetes v1.18 [stable]

以下卷插件支持原始块卷，包括其动态供应（如果支持的话）的卷：

- AWSElasticBlockStore
- AzureDisk
- CSI
- FC (光纤通道)
- GCEPersistentDisk
- iSCSI
- Local 卷
- OpenStack Cinder
- RBD (Ceph 块设备)
- VsphereVolume

使用原始块卷的持久卷

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cf1"]
    lun: 0
    readOnly: false
```

申请原始块卷的 PVC 申领

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
```

```
requests:  
storage: 10Gi
```

在容器中添加原始块设备路径的 Pod 规约

```
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
```

说明：向 Pod 中添加原始块设备时，你要在容器内设置设备路径而不是挂载路径。

绑定块卷

如果用户通过 PersistentVolumeClaim 规约的 volumeMode 字段来表明对原始块设备的请求，绑定规则与之前版本中未在规约中考虑此模式的实现略有不同。下面列举的表格是用户和管理员可以为请求原始块设备所作设置的组合。此表格表明在不同的组合下卷是否会被绑定。

静态供应卷的卷绑定矩阵：

PV volumeMode	PVC volumeMode	Result
未指定	未指定	绑定
未指定	Block	不绑定
未指定	Filesystem	绑定
Block	未指定	不绑定
Block	Block	绑定
Block	Filesystem	不绑定
Filesystem	Filesystem	绑定
Filesystem	Block	不绑定
Filesystem	未指定	绑定

说明：Alpha 发行版本中仅支持静态供应的卷。 管理员需要在处理原始块设备时小心处理这些值。

对卷快照及从卷快照中恢复卷的支持

FEATURE STATE: Kubernetes v1.17 [beta]

卷快照 (Volume Snapshot) 功能的添加仅是为了支持 CSI 卷插件。 有关细节可参阅 [卷快照文档](#)。

要启用从卷快照数据源恢复数据卷的支持，可在 API 服务器和控制器管理器上启用 VolumeSnapshotDataSource 特性门控。

基于卷快照创建 PVC 申领

```
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
```

卷克隆

[卷克隆](#)功能特性仅适用于 CSI 卷插件。

基于现有 PVC 创建新的 PVC 申领

```
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
```

编写可移植的配置

如果你要编写配置模板和示例用来在很多集群上运行并且需要持久性存储，建议你使用以下模式：

- 将 PersistentVolumeClaim 对象包含到你的配置包 (Bundle) 中，和 Deployment 以及 ConfigMap 等放在一起。
- 不要在配置中包含 PersistentVolume 对象，因为对配置进行实例化的用户很可能没有创建 PersistentVolume 的权限。
- 为用户提供在实例化模板时指定存储类名称的能力。
 - 仍按用户提供存储类名称，将该名称放到 persistentVolumeClaim.storageClassName 字段中。这样会使得 PVC 在集群被管理员启用了存储类支持时能够匹配到正确的存储类，
 - 如果用户未指定存储类名称，将 persistentVolumeClaim.storageClassName 留空 (nil)。这样，集群会使用默认 StorageClass 为用户自动供应一个存储卷。很多集群环境都配置了默认的 StorageClass，或者管理员也可以自行创建默认的 StorageClass。
- 在你的工具链中，监测经过一段时间后仍未被绑定的 PVC 对象，要让用户知道这些对象，因为这可能意味着集群没有动态存储支持（因而用户必须先创建一个匹配的 PV），或者 集群没有配置存储系统（因而用户无法配置需要 PVC 的工作负载配置）。

接下来

- 进一步了解[创建持久卷](#)。
- 进一步学习[创建 PVC 申领](#)。
- 阅读[持久存储的设计文档](#)。

参考

- [PersistentVolume](#)
- [PersistentVolumeSpec](#)
- [PersistentVolumeClaim](#)
- [PersistentVolumeClaimSpec](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 18, 2021 at 12:29 AM PST: [\[zh\] add tooltip for CSI in persistent volume page \(f0b4f2e6f\)](#)

CSI 卷克隆

本文档介绍 Kubernetes 中克隆现有 CSI 卷的概念。阅读前建议先熟悉[卷](#)。

介绍

[CSI 卷克隆功能](#)增加了通过在 `dataSource` 字段中指定存在的 [PVC](#)，来表示用户想要克隆的 [卷 \(Volume\)](#)。

克隆，意思是为已有的 Kubernetes 卷创建副本，它可以像任何其它标准卷一样被使用。唯一的区别就是配置后，后端设备将创建指定完全相同的副本，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 PVC 作为数据源的能力。源 PVC 必须是 bound 状态且可用的（不在使用中）。

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
- 克隆只能在两个使用相同 `VolumeMode` 设置的卷中进行（如果请求克隆一个块存储模式的卷，源卷必须也是块存储模式）。

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
```

说明：你必须为 spec.resources.requests.storage 指定一个值，并且你指定的值必须大于或等于源卷的值。

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 像其他 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

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 July 31, 2020 at 1:57 PM PST: [\[zh\] Fix links in concepts section \(9f7414e73\)](#)

卷快照类

本文档描述了 Kubernetes 中 VolumeSnapshotClass 的概念。建议熟悉 [卷快照 \(Volume Snapshots\)](#) 和 [存储类 \(Storage Class\)](#)。

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 driver、deletionPolicy 和 parameters 字段，在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。

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
parameters:
```

管理员可以为未请求任何特定类绑定的 VolumeSnapshots 指定默认的 VolumeSnapshotClass，方法是设置注解 snapshot.storage.kubernetes.io/is-default-class: "true"：

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshotClass
metadata:
  name: csi-hostpath-snapclass
  annotations:
    snapshot.storage.kubernetes.io/is-default-class: "true"
driver: hostpath.csi.k8s.io
deletionPolicy: Delete
parameters:
```

驱动程序

卷快照类有一个驱动程序，用于确定配置 VolumeSnapshot 的 CSI 卷插件。此字段必须指定。

删除策略

卷快照类具有 deletionPolicy 属性。用户可以配置当所绑定的 VolumeSnapshot 对象将被删除时，如何处理 VolumeSnapshotContent 对象。卷快照的这个策略可以是 Retain 或者 Delete。这个策略字段必须指定。

如果删除策略是 Delete，那么底层的存储快照会和 VolumeSnapshotContent 对象一起删除。如果删除策略是 Retain，那么底层快照和 VolumeSnapshotContent 对象都会被保留。

参数

卷快照类具有描述属于该卷快照类的卷快照的参数，可根据 driver 接受不同的参数。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 08, 2021 at 7:52 AM PST: [snapshot.storage.k8s.io/v1beta1](#)
--> [apiVersion: snapshot.storage.k8s.io/v1 \(4d1862831\)](#)

存储类

本文描述了 Kubernetes 中 StorageClass 的概念。建议先熟悉 [卷](#) 和 [持久卷](#) 的概念。

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，或是由集群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 指定一个默认的存储类：更多详情请参阅 [PersistentVolumeClaim 章节](#)。

```
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
```

存储制备器

每个 StorageClass 都有一个制备器（ Provisioner ），用来决定使用哪个卷插件制备 PV。该字段必须指定。

卷插件	内置制备器	配置例子
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

你不限于指定此处列出的“内置”制备器（其名称前缀为“kubernetes.io”并打包在 Kubernetes 中）。你还可以运行和指定外部制备器，这些独立的程序遵循由 Kubernetes 定义的[规范](#)。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](#) 包含一个用于为外部制备器编写功能实现的类库。你可以访问代码仓库[kubernetes-sigs/sig-storage-lib-external-provisioner](#) 了解外部驱动列表。

例如，NFS 没有内部制备器，但可以使用外部制备器。也有第三方存储供应商提供自己的外部制备器。

回收策略

由 StorageClass 动态创建的 PersistentVolume 会在类的 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 PersistentVolume 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]

PersistentVolume 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。

当下层 StorageClass 的 allowVolumeExpansion 字段设置为 true 时，以下类型的卷支持卷扩展。

卷类型	Kubernetes 版本要求
gcePersistentDisk	1.11
awsElasticBlockStore	1.11
Cinder	1.11
glusterfs	1.11
rbd	1.11
Azure File	1.11
Azure Disk	1.11
Portworx	1.11
FlexVolume	1.13
CSI	1.14 (alpha), 1.16 (beta)

说明：此功能仅可用于扩容卷，不能用于缩小卷。

挂载选项

由 StorageClass 动态创建的 PersistentVolume 将使用类中 mountOptions 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，则制备操作会失败。挂载选项在 StorageClass 和 PV 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

volumeBindingMode 字段控制了[卷绑定和动态制备](#) 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制备。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制备。

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 的绑定和制备，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制备。这些包括但不限于[资源需求](#)、[节点筛选器](#)、[pod 亲和性和互斥性](#)、以及[污点和容忍度](#)。

以下插件支持动态供应的 WaitForFirstConsumer 模式：

- [AWSElasticBlockStore](#)
- [GCEPersistentDisk](#)
- [AzureDisk](#)

以下插件支持预创建绑定 PersistentVolume 的 WaitForFirstConsumer 模式：

- 上述全部
- [Local](#)

FEATURE STATE: Kubernetes v1.17 [stable]

动态配置和预先创建的 PV 也支持 [CSI卷](#)，但是你需要查看特定 CSI 驱动程序的文档以查看其支持的拓扑键名和例子。

允许的拓扑结构

FEATURE STATE: Kubernetes v1.12 [beta]

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制备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供应卷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 Classes 的参数描述了存储类的卷。取决于制备器，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。

一个 StorageClass 最多可以定义 512 个参数。这些参数对象的总长度不能 超过 256 KiB, 包括参数的键和值。

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](#)。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 20000 IOPS (AWS 支持的最高值 , 请参阅 [AWS 文档](#)。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。

说明 :

zone 和 zones 已被弃用并被 [允许的拓扑结构](#) 取代。

GCE PD

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

```
fstype: ext4
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。
- fstype: ext4 或 xfs。 默认: ext4。宿主机操作系统必须支持所定义的文件系统类型。
- replication-type : none 或者 regional-pd。默认值 : none。

如果 replication-type 设置为 none , 会制备一个常规 (当前区域内的) 持久化磁盘。

如果 replication-type 设置为 regional-pd , 会制备一个 [区域性持久化磁盘 \(Regional Persistent Disk \)](#)。

强烈建议设置 volumeBindingMode: WaitForFirstConsumer , 这样设置后 , 当你创建一个 Pod , 它使用的 PersistentVolumeClaim 使用了这个 StorageClass , 区域性持久化磁盘会在两个区域里制备。其中一个区域是 Pod 所在区域。另一个区域是会在集群管理的区域中任意选择。磁盘区域可以通过 allowedTopologies 加以限制。

说明 : zone 和 zones 已被弃用并被 [allowedTopologies](#) 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 : 制备 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 动态制备器的必需参数。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` : 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](#) 中找到。

- `clusterid` : `630372ccdc720a92c681fb928f27b53f` 是集群的 ID，当制备卷时，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` : StorageClass GID 范围的最小值和最大值。在此范围 (`gidMin-gidMax`) 内的唯一值 (GID) 将用于动态制备卷。这些是可选的值。如果不指定，所制备的卷为一个 2000-2147483647 之间的值，这是 `gidMin` 和 `gidMax` 的默认值。
- `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，则由制备器决定卷的类型。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](#)。

更多相关的参考信息，请参阅 [如何配置 Heketi](#)。

当动态制备持久卷时，Gluster 插件自动创建名为 `gluster-dynamic-<claimname>` 的端点和无头服务。在 PVC 被删除时动态端点和无头服务会自动被删除。

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转调度。

说明：

FEATURE STATE: Kubernetes 1.11 [deprecated]

OpenStack 的内部驱动已经被弃用。请使用 [OpenStack 的外部云驱动](#)。

vSphere

vSphere 存储类有两种制备器

- [CSI 制备器](#): csi.vsphere.vmware.com
- [vCP 制备器](#): kubernetes.io/vsphere-volume

树内制备器已经被 [弃用](#)。更多关于 CSI 制备器的详情，请参阅 [Kubernetes vSphere CSI 驱动](#) 和 [vSphereVolume CSI 迁移](#)。

CSI 制备器

vSphere CSI StorageClass 制备器在 Tanzu Kubernetes 集群下运行。示例请参[vSphere CSI 仓库](#)。

vCP 制备器

以下示例使用 VMware Cloud Provider (vCP) StorageClass 调度器

1. 使用用户指定的磁盘格式创建一个 StorageClass。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  diskformat: zeroedthick
```

`diskformat`: thin, zeroedthick 和 eagerzeroedthick。默认值: "thin"。

- 在用户指定的数据存储上创建磁盘格式的 StorageClass。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/vsphere-volume
parameters:
  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。你现在可以在动态制备卷期间以存储能力的形式定义存储需求，例如性能和可用性。存储能力需求会转换为 Virtual SAN 策略，之后当持久卷（虚拟磁盘）被创建时，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。

你可以参考[基于存储策略的动态制备卷管理](#)，进一步了解有关持久卷管理的存储策略的详细信息。

有几个 [vSphere 例子](#) 供你在 Kubernetes for vSphere 中尝试进行持久卷管理。

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 : 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 :

```
kubectl create secret generic ceph-secret --type="kubernetes.io/rbd" \
--from-literal=key='QVFEQ1pMdFhPUUnQrSmhBQUFYaERWNHJsZ3BsMmNj
cDR6RFZST0E9PQ==' \
--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 , 或者如果你有正在运行的 DNS , 你也可以提供 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。你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。

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。
- resourceGroup: 指定要创建 Azure 磁盘所属的资源组。必须是已存在的资源组名称。若未指定资源组，磁盘会默认放入与当前 Kubernetes 集群相同的资源组中。
- 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。

在存储制备期间，为挂载凭证创建一个名为 `secretName` 的 Secret。如果集群同时启用了 [RBAC](#) 和 [控制器角色](#)，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"
  priority_io: "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 : 制备卷的 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 : 指定给动态创建的卷的描述。所有卷描述对于存储类而言都是相同的，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 Kubernetes 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
```

用于动态制备卷的 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 [stable]

```
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制备，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

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 December 17, 2020 at 4:44 PM PST: [\[zh\] update content to match en master \(fbbd5b4f7\)](#)

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](#) 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

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，但仍能够从多个存储选项中进行选择。

点击[这里](#)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 对象定义当动态供应被调用时，哪一个驱动将被使用和哪些参数将被传递给驱动。以下清单创建了一个 StorageClass 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](#)）。

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。

设置默认值的行为

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](#) 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](#)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](#)来实现。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 16, 2020 at 2:16 PM PST: [\[zh\] Sync changes from English site \(7\) \(089040daa\)](#)

存储容量

存储容量是有限的，并且会因为运行 Pod 的节点不同而变化：网络存储可能并非所有节点都能够访问，或者对于某个节点存储是本地的。

FEATURE STATE: Kubernetes v1.19 [alpha]

本页面描述了 Kubernetes 如何跟踪存储容量以及调度程序如何为了余下的尚未挂载的卷使用该信息将 Pod 调度到能够访问到足够存储容量的节点上。如果没有跟踪存储容量，调度程序可能会选择一个没有足够容量来提供卷的节点，并且需要多次调度重试。

[容器存储接口](#) (CSI) 驱动程序支持跟踪存储容量，并且在安装 CSI 驱动程序时[需要启用](#)该功能。

API

这个特性有两个 API 扩展接口：

- CSISecurityCapacity 对象：这些对象由 CSI 驱动程序在安装驱动程序的命名空间中产生。每个对象都包含一个存储类的容量信息，并定义哪些节点可以访问该存储。
- [CSIDriverSpec.StorageCapacity 字段](#)：设置为 true 时，Kubernetes 调度程序将考虑使用 CSI 驱动程序的卷的存储容量。

调度

如果有以下情况，存储容量信息将会被 Kubernetes 调度程序使用：

- CSISecurityCapacity 特性门控被设置为 true，
- Pod 使用的卷还没有被创建，
- 卷使用引用了 CSI 驱动的 [StorageClass](#)，并且使用了 [WaitForFirstConsumer 卷绑定模式](#)，
- 驱动程序的 CSIDriver 对象的 StorageCapacity 被设置为 true。

在这种情况下，调度程序仅考虑将 Pod 调度到有足够存储容量的节点上。这个检测非常简单，仅将卷的大小与 CSIStrageCapacity 对象中列出的容量进行比较，并使用包含该节点的拓扑。

对于具有 Immediate 卷绑定模式的卷，存储驱动程序将决定在何处创建该卷，而不取决于将使用该卷的 Pod。然后，调度程序将 Pod 调度到创建卷后可使用该卷的节点上。

对于 [CSI 临时卷](#)，调度总是在不考虑存储容量的情况下进行。这是基于这样的假设：该卷类型仅由节点本地的特殊 CSI 驱动程序使用，并且不需要大量资源。

重新调度

当为带有 WaitForFirstConsumer 的卷的 Pod 来选择节点时，该决定仍然是暂定的。下一步是要求 CSI 存储驱动程序创建卷，并提示该卷在被选择的节点上可用。

因为 Kubernetes 可能会根据已经过时的存储容量信息来选择一个节点，因此可能无法真正创建卷。然后就会重置节点选择，Kubernetes 调度器会再次尝试为 Pod 查找节点。

限制

存储容量跟踪增加了调度器第一次尝试即成功的机会，但是并不能保证这一点，因为调度器必须根据可能过期的信息来进行决策。通常，与没有任何存储容量信息的调度相同的重试机制可以处理调度失败。

当 Pod 使用多个卷时，调度可能会永久失败：一个卷可能已经在拓扑段中创建，而该卷又没有足够的容量来创建另一个卷，要想从中恢复，必须要进行手动干预，比如通过增加存储容量或者删除已经创建的卷。需要[进一步工作](#)来自动处理此问题。

开启存储容量跟踪

存储容量跟踪是一个 *alpha* 特性，只有当 CSIStrageCapacity [特性门控](#) 和 storage.k8s.io/v1alpha1 [API 组](#) 启用时才能启用。更多详细信息，可以查看--feature-gates 和 --runtime-config [kube-apiserver 参数](#)。

快速检查 Kubernetes 集群是否支持这个特性，可以通过下面命令列出 CSIStrageCapacity 对象：

```
kubectl get csistoragecapacities --all-namespaces
```

如果集群支持 CSIStrageCapacity，就会返回 CSIStrageCapacity 的对象列表或者：

```
No resources found
```

如果不支持，下面这个错误就会被打印出来：

```
error: the server doesn't have a resource type "csistoragecapacities"
```

除了在集群中启用该功能外，CSI 驱动程序还必须支持它。有关详细信息，请参阅驱动程序的文档。

接下来

- 想要获得更多该设计的信息，查看 [Storage Capacity Constraints for Pod Scheduling KEP](#)。
- 有关此功能的进一步开发信息，查看 [enhancement tracking issue #1472](#)。
- 学习 [Kubernetes 调度器](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 17, 2020 at 4:07 PM PST: [Update storage-capacity.md \(39c6521f1\)](#)

临时卷

本文档描述 Kubernetes 中的 *临时卷* (*Ephemeral Volume*)。建议先了解[卷](#)，特别是 PersistentVolumeClaim 和 PersistentVolume。

有些应用程序需要额外的存储，但并不关心数据在重启后仍然可用，既是否被持久地保存。例如，缓存服务经常受限于内存大小，将不常用的数据转移到比内存慢、但对总体性能的影响很小的存储中。

另有些应用程序需要以文件形式注入的只读数据，比如配置数据或密钥。

临时卷 就是为此类用例设计的。因为卷会遵从 Pod 的生命周期，与 Pod 一起创建和删除，所以停止和重新启动 Pod 时，不会受持久卷在何处可用的限制。

临时卷在 Pod 规范中以 *内联* 方式定义，这简化了应用程序的部署和管理。

临时卷的类型

Kubernetes 为了不同的目的，支持几种不同类型的临时卷：

- [emptyDir](#)：Pod 启动时为空，存储空间来自本地的 kubelet 根目录（通常是根磁盘）或内存
- [configMap](#)、[downwardAPI](#)、[secret](#)：将不同类型的 Kubernetes 数据注入到 Pod 中
- [CSI 临时卷](#)：类似于前面的卷类型，但由专门[支持此特性](#) 的指定 [CSI 驱动程序](#) 提供
- [通用临时卷](#)：它可以由所有支持持久卷的存储驱动程序提供

`emptyDir`、`configMap`、`downwardAPI`、`secret` 是作为 [本地临时存储](#) 提供的。它们由各个节点上的 `kubelet` 管理。

CSI 临时卷 必须 由第三方 CSI 存储驱动程序提供。

通用临时卷 可以 由第三方 CSI 存储驱动程序提供，也可以由支持动态配置的任何其他存储驱动程序提供。一些专门为 CSI 临时卷编写的 CSI 驱动程序，不支持动态供应：因此这些驱动程序不能用于通用临时卷。

使用第三方驱动程序的优势在于，它们可以提供 Kubernetes 本身不支持的功能，例如，与 `kubelet` 管理的磁盘具有不同运行特征的存储，或者用来注入不同的数据。

CSI 临时卷

FEATURE STATE: Kubernetes v1.16 [beta]

该特性需要启用参数 `CSIIInlineVolume` [特性门控 \(feature gate \)](#)。该参数从 Kubernetes 1.16 开始默认启用。

说明：只有一部分 CSI 驱动程序支持 CSI 临时卷。Kubernetes [CSI 驱动程序列表](#) 显示了支持临时卷的驱动程序。

从概念上讲，CSI 临时卷类似于 `configMap`、`downwardAPI` 和 `secret` 类型的卷：其存储在每个节点本地管理，并在将 Pod 调度到节点后与其他本地资源一起创建。在这个阶段，Kubernetes 没有重新调度 Pods 的概念。卷创建不太可能失败，否则 Pod 启动将会受阻。特别是，这些卷 不支持[感知存储容量的 Pod 调度](#)。它们目前也没包括在 Pod 的存储资源使用限制中，因为 `kubelet` 只能对它自己管理的存储强制执行。

下面是使用 CSI 临时存储的 Pod 的示例清单：

```
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
```

`volumeAttributes` 决定驱动程序准备什么样的卷。这些属性特定于每个驱动程序，且没有实现标准化。有关进一步的说明，请参阅每个 CSI 驱动程序的文档。

作为一个集群管理员，你可以使用 [PodSecurityPolicy](#) 来控制在 Pod 中可以使用哪些 CSI 驱动程序，具体则是通过 [allowedCSIDrivers 字段](#) 指定。

通用临时卷

FEATURE STATE: Kubernetes v1.19 [alpha]

这个特性需要启用 `GenericEphemeralVolume` [特性门控](#)。因为这是一个alpha特性，默禁用。

通用临时卷类似于 `emptyDir` 卷，但更加灵活：

- 存储可以是本地的，也可以是网络连接的。
- 卷可以有固定的大小，pod不能超量使用。
- 卷可能有一些初始数据，这取决于驱动程序和参数。
- 当驱动程序支持，卷上的典型操作将被支持，包括（[快照、克隆、调整大小](#)和[存储容量跟踪](#)）。

示例：

```
kind: Pod
apiVersion: v1
metadata:
  name: my-app
spec:
  containers:
    - name: my-frontend
      image: busybox
      volumeMounts:
        - mountPath: "/scratch"
          name: scratch-volume
      command: [ "sleep", "1000000" ]
  volumes:
    - name: scratch-volume
      ephemeral:
        volumeClaimTemplate:
          metadata:
            labels:
              type: my-frontend-volume
          spec:
            accessModes: [ "ReadWriteOnce" ]
            storageClassName: "scratch-storage-class"
            resources:
              requests:
                storage: 1Gi
```

生命周期和 PersistentVolumeClaim

关键的设计思想是在 Pod 的卷来源中允许使用 [卷申领的参数](#)。

PersistentVolumeClaim 的标签、注解和整套字段集均被支持。创建这样一个 Pod 后，临时卷控制器在 Pod 所属的命名空间中创建一个实际的 PersistentVolumeClaim 对象，并确保删除 Pod 时，同步删除 PersistentVolumeClaim。

如上设置将触发卷的绑定与/或准备操作，相应动作或者在 [StorageClass](#) 使用即时卷绑定时立即执行，或者当 Pod 被暂时性调度到某节点时执行 (WaitForFirstConsumer 卷绑定模式)。对于常见的临时卷，建议采用后者，这样调度器就可以自由地为 Pod 选择合适的节点。对于即时绑定，调度器则必须选出一个节点，使得在卷可用时，能立即访问该卷。

就[资源所有权](#)而言，拥有通用临时存储的 Pod 是提供临时存储 (ephemeral storage) 的 PersistentVolumeClaim 的所有者。当 Pod 被删除时，Kubernetes 垃圾收集器会删除 PVC，然后 PVC 通常会触发卷的删除，因为存储类的默认回收策略是删除卷。你可以使用带有 retain 回收策略的 StorageClass 创建准临时 (quasi-ephemeral) 本地存储：该存储比 Pod 寿命长，在这种情况下，你需要确保单独进行卷清理。

当这些 PVC 存在时，它们可以像其他 PVC 一样使用。特别是，它们可以被引用作为批量克隆或快照的数据源。PVC 对象还保持着卷的当前状态。

PersistentVolumeClaim 的命名

自动创建的 PVCs 的命名是确定的：此名称是 Pod 名称和卷名称的组合，中间由连字符 (-) 连接。在上面的示例中，PVC 将命名为 my-app-scratch-volume。这种确定性命名方式使得与 PVC 交互变得更容易，因为一旦知道 Pod 名称和卷名，就不必搜索它。

这种确定性命名方式也引入了潜在的冲突，比如在不同的 Pod 之间（名为 "Pod-a" 的 Pod 挂载名为 "scratch" 的卷，名为 "pod" 的 Pod 挂载名为 "a-scratch" 的卷，这两者均会生成名为 "pod-a-scratch" 的 PVC），或者在 Pod 和手工创建的 PVC 之间。

以下冲突会被检测到：如果 PVC 是为 Pod 创建的，那么它只用于临时卷。此检测基于所有权关系。现有的 PVC 不会被覆盖或修改。但这并不能解决冲突，因为如果没有正确的 PVC，Pod 就无法启动。

注意：当命名 Pods 和卷出现在同一个命名空间中时，要小心，以防止发生此类冲突。

安全

启用 GenericEphemeralVolume 特性会导致那些没有 PVCs 创建权限的用户，在创建 Pods 时，被允许间接的创建 PVCs。集群管理员必须意识到这一点。如果这不符合他们的安全模型，他们有两种选择：

- 通过特性门控显式禁用该特性，可以避免将来的 Kubernetes 版本默认启用时带来混乱。
- 当卷列表不包含 ephemeral 卷类型时，使用 [Pod 安全策略](#)。

在一个命名空间中，用于 PVCs 的常规命名空间配额仍然适用，因此即使允许用户使用这种新机制，他们也不能使用它来规避其他策略。

接下来

kubelet 管理的临时卷

参阅[本地临时存储](#)。

CSI 临时卷

- 有关设计的更多信息，参阅 [Ephemeral Inline CSI volumes KEP](#)。
- 本特性下一步开发的更多信息，参阅 [enhancement tracking issue #596](#)。

通用临时卷

- 有关设计的更多信息，参阅 [Generic ephemeral inline volumes KEP](#)。
- 本特性下一步开发的更多信息，参阅 [enhancement tracking issue #1698](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 30, 2020 at 1:53 AM PST: [sync link to CSI ephemeral volumes \(zh\) \(c4edcaaba\)](#)

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

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)	39
Google Persistent Disk	16
Microsoft Azure Disk Storage	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 [stable]

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](#) 环境中，根据节点类型最多可以将 127 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 \(EC2\)](#)，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](#)。
- 如果 CSI 存储驱动程序（使用 NodeGetInfo）为节点通告卷数上限，则 [kube-scheduler](#) 将遵守该限制值。参考 [CSI 规范](#) 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。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 June 01, 2020 at 9:23 AM PST: [add zh pages \(4b35d4d40\)](#)

配置

Kubernetes 为配置 Pods 提供的资源。

[配置最佳实践](#)

[ConfigMap](#)

[Secret](#)

[为容器管理资源](#)

[使用 kubeconfig 文件组织集群访问](#)

[Pod 优先级与抢占](#)

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。

这是一份不断改进的文件。 如果您认为某些内容缺失但可能对其他人有用，请不要犹豫，提交 Issue 或提交 PR。

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](#) 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。
- 除非必要，否则不指定默认值：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

- 如果可能，不要使用独立的 Pods（即，未绑定到 [ReplicaSet](#) 或 [Deployment](#) 的 Pod）。如果节点发生故障，将不会重新调度独立的 Pods。

Deployment 既可以创建一个 ReplicaSet 来确保预期个数的 Pod 始终可用，也可以指定替换 Pod 的策略（例如 [RollingUpdate](#)）。除了一些显式的 [restartPolicy: Never](#) 场景外，Deployment 通常比直接创建 Pod 要好得多。[Job](#) 也可能是合适的选择。

服务

- 在创建相应的后端工作负载（Deployment 或 ReplicaSet），以及在需要访问它的任何工作负载之前创建 [服务](#)。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 的服务，则所有容器将在其初始环境中获得以下变量。

```
FOO_SERVICE_HOST=<the host the Service is running on>
FOO_SERVICE_PORT=<the port the Service is running on>
```

这确实意味着在顺序上的要求 - 必须在 Pod 本身被创建之前创建 Pod 想要访问的任何 Service，否则将环境变量不会生效。DNS 没有此限制。

- 一个可选（尽管强烈推荐）的[集群插件](#)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](#) 或 [kubectl port-forward](#)。

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](#) 服务。

- 避免使用 hostNetwork，原因与 hostPort 相同。
- 当您不需要 kube-proxy 负载均衡时，使用 [无头服务](#)（ClusterIP 被设置为 None）以便于服务发现。

使用标签

- 定义并使用[标签](#)来识别应用程序或 Deployment 的语义属性，例如 { app: myapp, tier: frontend, phase: test, deployment: v3 }。你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[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 Deployment。[Deployment](#) 可以在不停机的情况下轻松更新正在运行的服务。

Deployment 描述了对象的期望状态，并且如果对该规范的更改被成功应用，则 Deployment 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来匹配 Pod，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“活跃”的 Pod 的有用方法。要以交互方式删除或添加标签，请使用[kubectl label](#)。

容器镜像

[imagePullPolicy](#) 和镜像标签会影响 [kubelet](#) 何时尝试拉取指定的镜像。

- imagePullPolicy: IfNotPresent：仅当镜像在本地不存在时才被拉取。
- imagePullPolicy: Always：每次启动 Pod 的时候都会拉取镜像。
- imagePullPolicy 省略时，镜像标签为 :latest 或不存在，使用 Always 值。
- imagePullPolicy 省略时，指定镜像标签并且不是 :latest，使用 IfNotPresent 值。
- imagePullPolicy: Never：假设镜像已经存在本地，不会尝试拉取镜像。

说明：要确保容器始终使用相同版本的镜像，你可以指定其 [摘要](#)，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。

说明：在生产中部署容器时应避免使用 :latest 标记，因为这样更难跟踪正在运行的镜像版本，并且更难以正确回滚。

说明：底层镜像驱动程序的缓存语义能够使即便 imagePullPolicy: Always 的配置也很高效。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](#)和[有效使用标签](#)部分。
- 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](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 December 03, 2020 at 7:23 PM PST: [update content/zh/docs/concepts/configuration/overview.md \(d3c0fae62\)](#)

ConfigMap

ConfigMap 是一种 API 对象，用来将非机密性的数据保存到键值对中。使用时，[Pods](#) 可以将其用作环境变量、命令行参数或者存储卷中的配置文件。

ConfigMap 将您的环境配置信息和 [容器镜像](#) 解耦，便于应用配置的修改。

注意：

ConfigMap 并不提供保密或者加密功能。如果你想存储的数据是机密的，请使用 [Secret](#)，或者使用其他第三方工具来保证你的数据的私密性，而不是用 ConfigMap。

动机

使用 ConfigMap 来将你的配置数据和应用程序代码分开。

比如，假设你正在开发一个应用，它可以在你自己的电脑上（用于开发）和在云上（用于实际流量）运行。你的代码里有一段是用于查看环境变量 DATABASE_HOST，在本地运行时，你将这个变量设置为 localhost，在云上，你将其设置为引用 Kubernetes 集群中的 [公开数据库组件的服务](#)。

这让你可以获取在云中运行的容器镜像，并且如果有需要的话，在本地调试完全相同的代码。

ConfigMap 在设计上不是用来保存大量数据的。在 ConfigMap 中保存的数据不可超过 1 MiB。如果你需要保存超出此尺寸限制的数据，你可能希望考虑挂载存储卷或者使用独立的数据库或者文件服务。

ConfigMap 对象

ConfigMap 是一个 API [对象](#)，让你可以存储其他对象所需要使用的配置。和其他 Kubernetes 对象都有一个 spec 不同的是，ConfigMap 使用 data 和 binaryData 字段。这些字段能够接收键-值对作为其取值。data 和 binaryData 字段都是可选的。data 字段设计用来保存 UTF-8 字节序列，而 binaryData 则被设计用来保存二进制数据。

ConfigMap 的名字必须是一个合法的 [DNS 子域名](#)。

data 或 binaryData 字段下面的每个键的名称都必须由字母数字字符或者 -、_ 或 . 组成。在 data 下保存的键名不可以与在 binaryData 下出现的键名有重叠。

从 v1.19 开始，你可以添加一个 immutable 字段到 ConfigMap 定义中，创建 [不可变更的 ConfigMap](#)。

ConfigMaps 和 Pods

你可以写一个引用 ConfigMap 的 Pod 的 spec，并根据 ConfigMap 中的数据 在该 Pod 中配置容器。这个 Pod 和 ConfigMap 必须要在同一个 [名字空间](#) 中。

这是一个 ConfigMap 的示例，它的一些键只有一个值，其他键的值看起来像是 配置的片段格式。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # 类属性键；每一个键都映射到一个简单的值
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"

  # 类文件键
  game.properties: |
    enemy.types=aliens,monsters
    player.maximum-lives=5
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

你可以使用四种方式来使用 ConfigMap 配置 Pod 中的容器：

1. 在容器命令和参数内
2. 容器的环境变量
3. 在只读卷里面添加一个文件，让应用来读取
4. 编写代码在 Pod 中运行，使用 Kubernetes API 来读取 ConfigMap

这些不同的方法适用于不同的数据使用方式。对前三个方法，[kubelet](#) 使用 ConfigMap 中的数据在 Pod 中启动容器。

第四种方法意味着你必须编写代码才能读取 ConfigMap 和它的数据。然而，由于你是直接使用 Kubernetes API，因此只要 ConfigMap 发生更改，你的 应用就能够通过订阅来获取更新，并且在这样的情况发生的时候做出反应。通过直接进入 Kubernetes API，这个技术也可以让你能够获取到不同的名字空间 里的 ConfigMap。

下面是一个 Pod 的示例，它通过使用 game-demo 中的值来配置一个 Pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      env:
        # 定义环境变量
        - name: PLAYER_INITIAL_LIVES # 请注意这里和 ConfigMap 中的键名是不一样的
          valueFrom:
            configMapKeyRef:
              name: game-demo      # 这个值来自 ConfigMap
              key: player_initial_lives # 需要取值的键
        - name: UI_PROPERTIES_FILE_NAME
          valueFrom:
            configMapKeyRef:
              name: game-demo
              key: ui_properties_file_name
  volumeMounts:
    - name: config
      mountPath: "/config"
      readOnly: true
  volumes:
    # 你可以在 Pod 级别设置卷，然后将其挂载到 Pod 内的容器中
    - name: config
      configMap:
        # 提供你想要挂载的 ConfigMap 的名字
        name: game-demo
        # 来自 ConfigMap 的一组键，将被创建为文件
      items:
        - key: "game.properties"
          path: "game.properties"
        - key: "user-interface.properties"
          path: "user-interface.properties"
```

ConfigMap 不会区分单行属性值和多行类似文件的值，重要的是 Pods 和其他对象 如何使用这些值。

上面的例子定义了一个卷并将它作为 /config 文件夹挂载到 demo 容器内，创建两个文件，/config/game.properties 和 /config/user-interface.properties，尽管 ConfigMap 中包含了四个键。这是因为 Pod 定义中在 volumes 节指定了一个 items

数组。如果你完全忽略 items 数组，则 ConfigMap 中的每个键都会变成一个与该键同名的文件，因此你会得到四个文件。

使用 ConfigMap

ConfigMap 可以作为数据卷挂载。ConfigMap 也可被系统的其他组件使用，而不一定直接暴露给 Pod。例如，ConfigMap 可以保存系统中其他组件要使用的配置数据。

ConfigMap 最常见的用法是为同一命名空间里某 Pod 中运行的容器执行配置。你也可以单独使用 ConfigMap。

比如，你可能会遇到基于 ConfigMap 来调整其行为的 [插件](#) 或者 [operator](#)。

在 Pod 中将 ConfigMap 当做文件使用

1. 创建一个 ConfigMap 对象或者使用现有的 ConfigMap 对象。多个 Pod 可以引用同一个 ConfigMap。
2. 修改 Pod 定义，在 spec.volumes[] 下添加一个卷。为该卷设置任意名称，之后将 spec.volumes[].configMap.name 字段设置为对你的 ConfigMap 对象的引用。
3. 为每个需要该 ConfigMap 的容器添加一个 .spec.containers[].volumeMounts[]。设置 .spec.containers[].volumeMounts[].readOnly=true 并将 .spec.containers[].volumeMounts[].mountPath 设置为一个未使用的目录名，ConfigMap 的内容将出现在该目录中。
4. 更改你的镜像或者命令行，以便程序能够从该目录中查找文件。ConfigMap 中的每个 data 键会变成 mountPath 下面的一个文件名。

下面是一个将 ConfigMap 以卷的形式进行挂载的 Pod 示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      configMap:
        name: myconfigmap
```

你希望使用的每个 ConfigMap 都需要在 spec.volumes 中被引用到。

如果 Pod 中有多个容器，则每个容器都需要自己的 volumeMounts 块，但针对每个 ConfigMap，你只需要设置一个 spec.volumes 块。

被挂载的 ConfigMap 内容会被自动更新

当卷中使用的 ConfigMap 被更新时，所投射的键最终也会被更新。 kubelet 组件会在每次周期性同步时检查所挂载的 ConfigMap 是否为最新。不过，kubelet 使用的是其本地的高速缓存来获得 ConfigMap 的当前值。高速缓存的类型可以通过 [KubeletConfiguration 结构](#) 的 ConfigMapAndSecretChangeDetectionStrategy 字段来配置。

ConfigMap 既可以通过 watch 操作实现内容传播（默认形式），也可实现基于 TTL 的缓存，还可以直接将所有请求重定向到 API 服务器。因此，从 ConfigMap 被更新的那一刻算起，到新的主键被投射到 Pod 中去，这一时间跨度可能与 kubelet 的同步周期加上高速缓存的传播延迟相等。这里的传播延迟取决于所选的高速缓存类型（分别对应 watch 操作的传播延迟、高速缓存的 TTL 时长或者 0）。

以环境变量方式使用的 ConfigMap 数据不会被自动更新。更新这些数据需要重新启动 Pod。

不可变更的 ConfigMap

FEATURE STATE: Kubernetes v1.19 [beta]

Kubernetes Beta 特性 不可变更的 Secret 和 ConfigMap 提供了一种将各个 Secret 和 ConfigMap 设置为不可变更的选项。对于大量使用 ConfigMap 的集群（至少有数万个各不相同的 ConfigMap 给 Pod 挂载）而言，禁止更改 ConfigMap 的数据有以下好处：

- 保护应用，使之免受意外（不想要的）更新所带来的负面影响。
- 通过大幅降低对 kube-apiserver 的压力提升集群性能，这是因为系统会关闭对已标记为不可变更的 ConfigMap 的监视操作。

此功能特性由 ImmutableEphemeralVolumes [特性门控](#) 来控制。你可以通过将 immutable 字段设置为 true 创建不可变更的 ConfigMap。例如：

```
apiVersion: v1
kind: ConfigMap
metadata:
...
data:
...
immutable: true
```

一旦某 ConfigMap 被标记为不可变更，则无法逆转这一变化，也无法更改 data 或 binaryData 字段的内容。你只能删除并重建 ConfigMap。因为现有的 Pod 会维护一个对已删除的 ConfigMap 的挂载点，建议重新创建这些 Pods。

接下来

- 阅读 [Secret](#)。
- 阅读 [配置 Pod 来使用 ConfigMap](#)。
- 阅读 [Twelve-Factor 应用](#) 来了解将代码和配置分开的动机。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 26, 2020 at 5:24 AM PST: [\[zh\] sync configmap.md \(75c231d19\)](#)

Secret

Secret 对象类型用来保存敏感信息，例如密码、OAuth 令牌和 SSH 密钥。将这些信息放在 secret 中比放在 [Pod](#) 的定义或者 [容器镜像](#) 中来说更加安全和灵活。参阅 [Secret 设计文档](#) 获取更多详细信息。

Secret 是一种包含少量敏感信息例如密码、令牌或密钥的对象。这样的信息可能会被放在 Pod 规约中或者镜像中。用户可以创建 Secret，同时系统也创建了一些 Secret。

注意：

Kubernetes Secret 默认情况下存储为 base64-编码的、非加密的字符串。默认情况下，能够访问 API 的任何人，或者能够访问 Kubernetes 下层数据存储（etcd）的任何人都可以以明文形式读取这些数据。为了能够安全地使用 Secret，我们建议你（至少）：

1. 为 Secret [启用静态加密](#)；
2. [启用 RBAC 规则来限制对 Secret 的读写操作](#)。要注意，任何被允许创建 Pod 的人都默认地具有读取 Secret 的权限。

Secret 概览

要使用 Secret，Pod 需要引用 Secret。Pod 可以用三种方式之一来使用 Secret：

- 作为挂载到一个或多个容器上的 [卷](#) 中的[文件](#)。
- 作为[容器的环境变量](#)
- 由 [kubelet 在为 Pod 拉取镜像时使用](#)

Secret 对象的名称必须是合法的 [DNS 子域名](#)。在为创建 Secret 编写配置文件时，你可以设置 data 与/或 stringData 字段。data 和 stringData 字段都是可选的。data 字

段中所有键值都必须是 base64 编码的字符串。如果不希望执行这种 base64 字符串的转换操作，你可以选择设置 `stringData` 字段，其中可以使用任何字符串作为其取值。

Secret 的类型

在创建 Secret 对象时，你可以使用 [Secret](#) 资源的 `type` 字段，或者与其等价的 `kubectl` 命令行参数（如果有的话）为其设置类型。Secret 的类型用来帮助编写程序处理 Secret 数据。

Kubernetes 提供若干种内置的类型，用于一些常见的使用场景。针对这些类型，Kubernetes 所执行的合法性检查操作以及对其所实施的限制各不相同。

内置类型	用法
Opaque	用户定义的任意数据
<code>kubernetes.io/service-account-token</code>	服务账号令牌
<code>kubernetes.io/dockercfg</code>	<code>~/.dockercfg</code> 文件的序列化形式
<code>kubernetes.io/dockerconfigjson</code>	<code>~/.docker/config.json</code> 文件的序列化形式
<code>kubernetes.io/basic-auth</code>	用于基本身份认证的凭据
<code>kubernetes.io/ssh-auth</code>	用于 SSH 身份认证的凭据
<code>kubernetes.io/tls</code>	用于 TLS 客户端或者服务器端的数据
<code>bootstrap.kubernetes.io/token</code>	启动引导令牌数据

通过为 Secret 对象的 `type` 字段设置一个非空的字符串值，你也可以定义并使用自己 Secret 类型。如果 `type` 值为空字符串，则被视为 Opaque 类型。Kubernetes 并不对类型的名称作任何限制。不过，如果你要使用内置类型之一，则你必须满足为该类型所定义的所有要求。

Opaque Secret

当 Secret 配置文件中未作显式设定时，默认的 Secret 类型是 Opaque。当你使用 `kubectl` 来创建一个 Secret 时，你会使用 `generic` 子命令来标明 要创建的是一个 Opaque 类型 Secret。例如，下面的命令会创建一个空的 Opaque 类型 Secret 对象：

```
kubectl create secret generic empty-secret  
kubectl get secret empty-secret
```

输出类似于

```
NAME      TYPE    DATA  AGE  
empty-secret  Opaque  0    2m6s
```

`DATA` 列显示 Secret 中保存的数据条目个数。在这个例子中，0 意味着我们刚刚创建了一个空的 Secret。

服务账号令牌 Secret

类型为 kubernetes.io/service-account-token 的 Secret 用来存放标识某 服务账号的令牌。使用这种 Secret 类型时，你需要确保对象的注解 kubernetes.io/service-account-name 被设置为某个已有的服务账号名称。某个 Kubernetes 控制器会填写 Secret 的其它字段，例如 kubernetes.io/service-account.uid 注解以及 data 字段中的 token 键值，使之包含实际的令牌内容。

下面的配置实例声明了一个服务账号令牌 Secret：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  # 你可以像 Opaque Secret 一样在这里添加额外的键/值偶对
  extra: YmFyCg==
```

Kubernetes 在创建 Pod 时会自动创建一个服务账号 Secret 并自动修改你的 Pod 以使用该 Secret。该服务账号令牌 Secret 中包含了访问 Kubernetes API 所需要的凭据。

如果需要，可以禁止或者重载这种自动创建并使用 API 凭据的操作。不过，如果你仅仅是希望能够安全地访问 API 服务器，这是建议的工作方式。

参考 [ServiceAccount](#) 文档了解服务账号的工作原理。你也可以查看 [Pod](#) 资源中的 automountServiceAccountToken 和 serviceAccountName 字段文档，了解从 Pod 中引用服务账号。

Docker 配置 Secret

你可以使用下面两种 type 值之一来创建 Secret，用以存放访问 Docker 仓库 来下载镜像的凭据。

- kubernetes.io/dockercfg
- kubernetes.io/dockerconfigjson

kubernetes.io/dockercfg 是一种保留类型，用来存放 `~/.dockercfg` 文件的序列化形式。该文件是配置 Docker 命令行的一种老旧形式。使用此 Secret 类型时，你需要确保 Secret 的 data 字段中包含名为 `.dockercfg` 的主键，其对应键值是用 base64 编码的某 `~/.dockercfg` 文件的内容。

类型 kubernetes.io/dockerconfigjson 被设计用来保存 JSON 数据的序列化形式，该 JSON 也遵从 `~/.docker/config.json` 文件的格式规则，而后者是 `~/.dockercfg` 的新版本格式。使用此 Secret 类型时，Secret 对象的 data 字段必须包含 `.dockerconfigjson` 键，其键值为 base64 编码的字符串包含 `~/.docker/config.json` 文件的内容。

下面是一个 kubernetes.io/dockercfg 类型 Secret 的示例：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-dockercfg
type: kubernetes.io/dockercfg
data:
  .dockercfg: |
    "<base64 encoded ~/.dockercfg file>"
```

说明：

如果你不希望执行 base64 编码转换，可以使用 stringData 字段代替。

当你使用清单文件来创建这两类 Secret 时，API 服务器会检查 data 字段中是否 存在所期望的主键，并且验证其中所提供的键值是否是合法的 JSON 数据。不过，API 服务器不会检查 JSON 数据本身是否是一个合法的 Docker 配置文件内容。

```
kubectl create secret docker-registry secret-tiger-docker \
--docker-username=tiger \
--docker-password=pass113 \
--docker-email=tiger@acme.com
```

上面的命令创建一个类型为 kubernetes.io/dockerconfigjson 的 Secret。如果你对 data 字段中的 .dockerconfigjson 内容进行转储，你会得到下面的 JSON 内容，而这一内容是一个合法的 Docker 配置文件。

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "username": "tiger",
      "password": "pass113",
      "email": "tiger@acme.com",
      "auth": "dGlnZXI6cGFzcExMw=="
    }
  }
}
```

基本身份认证 Secret

kubernetes.io/basic-auth 类型用来存放用于基本身份认证所需的凭据信息。 使用这种 Secret 类型时，Secret 的 data 字段必须包含以下两个键：

- username: 用于身份认证的用户名；
- password: 用于身份认证的密码或令牌。

以上两个键的键值都是 base64 编码的字符串。当然你也可以在创建 Secret 时使用 stringData 字段来提供明文形式的内容。下面的 YAML 是基本身份认证 Secret 的一个示例清单：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin
  password: t0p-Secret
```

提供基本身份认证类型的 Secret 仅仅是出于用户方便性考虑。你也可以使用 Opaque 类型来保存用于基本身份认证的凭据。不过，使用内置的 Secret 类型的有助于对凭据格式进行归一化处理，并且 API 服务器确实会检查 Secret 配置中是否提供了所需要的主键。

SSH 身份认证 Secret

Kubernetes 所提供的内置类型 kubernetes.io/ssh-auth 用来存放 SSH 身份认证中所需要的凭据。使用这种 Secret 类型时，你就必须在其 data（或 stringData）字段中提供一个 ssh-privatekey 键值对，作为要使用的 SSH 凭据。

下面的 YAML 是一个 SSH 身份认证 Secret 的配置示例：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # 此例中的实际数据被截断
  ssh-privatekey: |
```

MIIEpQIBAAKCAQEaUqb/Y ...

提供 SSH 身份认证类型的 Secret 仅仅是出于用户方便性考虑。你也可以使用 Opaque 类型来保存用于 SSH 身份认证的凭据。不过，使用内置的 Secret 类型的有助于对凭据格式进行归一化处理，并且 API 服务器确实会检查 Secret 配置中是否提供了所需要的主键。

注意：SSH 私钥自身无法建立 SSH 客户端与服务器端之间的可信连接。需要其它方式来建立这种信任关系，以缓解“中间人（Man In The Middle）”攻击，例如向 ConfigMap 中添加一个 known_hosts 文件。

TLS Secret

Kubernetes 提供一种内置的 kubernetes.io/tls Secret 类型，用来存放证书 及其相关密钥（通常用在 TLS 场合）。此类数据主要提供给 Ingress 资源，用以终结 TLS 链接，不过也可以用于其他 资源或者负载。当使用此类型的 Secret 时，Secret 配置中的 data（或 stringData）字段必须包含 tls.key 和 tls.crt 主键，尽管 API 服务器实际上并不会对每个键的取值作进一步的合法性检查。

下面的 YAML 包含一个 TLS Secret 的配置示例：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # 此例中的数据被截断
  tls.crt: |
    MIIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
    MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...
```

提供 TLS 类型的 Secret 仅仅是出于用户方便性考虑。你也可以使用 Opaque 类型来保存用于 TLS 服务器与/或客户端的凭据。不过，使用内置的 Secret 类型的有助于对凭据格式进行归一化处理，并且 API 服务器确实会检查 Secret 配置中是否提供了所需要的主键。

当使用 kubectl 来创建 TLS Secret 时，你可以像下面的例子一样使用 tls 子命令：

```
kubectl create secret tls my-tls-secret \
--cert=path/to/cert/file \
--key=path/to/key/file
```

这里的公钥/私钥对都必须事先已存在。用于 --cert 的公钥证书必须是 .PEM 编码的（Base64 编码的 DER 格式），且与 --key 所给定的私钥匹配。私钥必须是通常所说的 PEM 私钥格式，且未加密。对这两个文件而言，PEM 格式数据的第一行和最后一行（例如，证书所对应的 -----BEGIN CERTIFICATE----- 和 -----END CERTIFICATE----）都不会包含在其中。

启动引导令牌 Secret

通过将 Secret 的 type 设置为 bootstrap.kubernetes.io/token 可以创建 启动引导令牌类型的 Secret。这种类型的 Secret 被设计用来支持节点的启动引导过程。其中包含用来为周知的 ConfigMap 签名的令牌。

启动引导令牌 Secret 通常创建于 kube-system 名字空间内，并以 bootstrap-token-<令牌 ID> 的形式命名；其中 <令牌 ID> 是一个由 6 个字符组成 的字符串，用作令牌的标识。

以 Kubernetes 清单文件的形式，某启动引导令牌 Secret 可能看起来像下面这样：

```
apiVersion: v1
kind: Secret
metadata:
  name: bootstrap-token-5emitj
  namespace: kube-system
type: bootstrap.kubernetes.io/token
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZWFrkbTpkZWZhdWx0
LW5vZGUtdG9rZW4=
  expiration: MjAyMC0wOS0xM1QwNDozOToxMFo=
  token-id: NWVtaXRq
  token-secret: a3E0Z2lodnN6emduMXAwcg==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
```

启动引导令牌类型的 Secret 会在 data 字段中包含如下主键：

- token-id：由 6 个随机字符组成的字符串，作为令牌的标识符。必需。
- token-secret：由 16 个随机字符组成的字符串，包含实际的令牌机密。必需。
- description：供用户阅读的字符串，描述令牌的用途。可选。
- expiration：一个使用 RFC3339 来编码的 UTC 绝对时间，给出令牌要过期的时间。可选。
- usage-bootstrap-<usage>：布尔类型的标志，用来标明启动引导令牌的其他用途。
- auth-extra-groups：用逗号分隔的组名列表，身份认证时除被认证为 system:bootstrappers 组之外，还会被添加到所列的用户组中。

上面的 YAML 文件可能看起来令人费解，因为其中的数值均为 base64 编码的字符串。实际上，你完全可以使用下面的 YAML 来创建一个一模一样的 Secret：

```
apiVersion: v1
kind: Secret
metadata:
  # 注意 Secret 的命名方式
  name: bootstrap-token-5emitj
  # 启动引导令牌 Secret 通常位于 kube-system 名字空间
  namespace: kube-system
type: bootstrap.kubernetes.io/token
stringData:
  auth-extra-groups: "system:bootstrappers:kubeadm:default-node-token"
  expiration: "2020-09-13T04:39:10Z"
  # 此令牌 ID 被用于生成 Secret 名称
  token-id: "5emitj"
  token-secret: "kq4gihvszzgn1p0r"
  # 此令牌还可用于 authentication (身份认证)
```

```
usage-bootstrap-authentication: "true"
# 且可用于 signing (证书签名)
usage-bootstrap-signing: "true"
```

创建 Secret

有几种不同的方式来创建 Secret :

- [使用 kubectl 命令创建 Secret](#)
- [使用配置文件来创建 Secret](#)
- [使用 kustomize 来创建 Secret](#)

编辑 Secret

你可以通过下面的命令编辑现有的 Secret :

```
kubectl edit secrets mysecret
```

这一命令会打开默认的编辑器，允许你更新 data 字段中包含的 base64 编码的 Secret 值：

```
# 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: MWYyZDFIMmU2N2Rm
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](#) 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，系统的其他部分将用它来代表你与外部系统进行交互。

在 Pod 中使用 Secret 文件

在 Pod 中使用存放在卷中的 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. 修改你的 Pod 定义，在 spec.volumes[] 下增加一个卷。可以给这个卷随意命名，它的 spec.volumes[].secret.secretName 必须是 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 为你想要该 Secret 出现的尚未使用的目录。
4. 修改你的镜像并且 / 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对应 mountPath 下的一个文件名。

这是一个在 Pod 中使用存放在挂载卷中 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 中引用。

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。

将 Secret 键名映射到特定路径

我们还可以控制 Secret 键名在存储卷中映射的的路径。你可以使用 spec.volumes[].secret.items 字段修改每个键对应的目标路径：

```
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 中指定的键会被映射。要使用 Secret 中所有键，就必须将它们都列在 items 字段中。所有列出的键名必须存在于相应的 Secret 中。否则，不会创建卷。

Secret 文件权限

你还可以指定 Secret 将拥有的权限模式位。如果不指定，默认使用 0644。你可以为整个 Secret 卷指定默认模式；如果需要，可以为每个密钥设定重载值。

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
```

之后，Secret 将被挂载到 /etc/foo 目录，而所有通过该 Secret 卷挂载 所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。如果你使用 YAML 而不是 JSON，则可以使用八进制符号以更自然的方式指定权限。

注意，如果你通过 kubectl exec 进入到 Pod 中，你需要沿着符号链接来找到 所期望的文件模式。例如，下面命令检查 Secret 文件的访问模式：

```
kubectl exec mypod -it sh  
  
cd /etc/foo  
ls -l
```

输出类似于：

```
total 0  
lrwxrwxrwx 1 root root 15 May 18 00:18 password -> ..data/password  
lrwxrwxrwx 1 root root 15 May 18 00:18 username -> ..data/username
```

沿着符号链接，可以查看文件的访问模式：

```
cd /etc/foo/..data  
ls -l
```

输出类似于：

```
total 8  
-r----- 1 root root 12 May 18 00:18 password  
-r----- 1 root root 5 May 18 00:18 username
```

你还可以使用映射，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
```

在这里，位于 /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，即 511。

请注意，如果稍后读取此权限值，可能会以十进制格式显示。

使用来自卷中的 Secret 值

在挂载了 Secret 卷的容器内，Secret 键名显示为文件名，并且 Secret 的值 使用 base-64 解码后存储在这些文件中。这是在上面的示例容器内执行的命令的结果：

```
ls /etc/foo/
```

输出类似于：

```
username  
password
```

```
cat /etc/foo/username
```

输出类似于：

```
admin
```

```
cat /etc/foo/password
```

输出类似于：

```
1f2d1e2e67df
```

容器中的程序负责从文件中读取 secret。

挂载的 Secret 会被自动更新

当已经存储于卷中被使用的 Secret 被更新时，被映射的键也将终将被更新。组件 kubelet 在周期性同步时检查被挂载的 Secret 是不是最新的。但是，它会使用其本地缓存的数值作为 Secret 的当前值。

缓存的类型可以使用 [KubeletConfiguration 结构](#) 中的 ConfigMapAndSecretChange DetectionStrategy 字段来配置。它可以通过 watch 操作来传播（默认），基于 TTL 来刷新，也可以 将所有请求直接重定向到 API 服务器。因此，从 Secret 被更新到将新 Secret 被投射到 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。对于不同的缓存类型，该延迟或者等于 watch 传播延迟，或者等于缓存的 TTL，或者为 0。

说明： 使用 Secret 作为[子路径](#)卷挂载的容器 不会收到 Secret 更新。

以环境变量的形式使用 Secrets

将 Secret 作为 Pod 中的[环境变量](#)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 键的环境变量。使用 Secret 键的环境变量应在 env[x].valueFrom.secretKeyRef 中指定要包含的 Secret 名称和键名。
3. 更改镜像并 / 或者命令行，以便程序在指定的环境变量中查找值。

这是一个使用来自环境变量中的 Secret 值的 Pod 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
```

使用来自环境变量的 Secret 值

在一个以环境变量形式使用 Secret 的容器中，Secret 键表现为常规的环境变量，其中包含 Secret 数据的 base-64 解码值。这是从上面的示例在容器内执行的命令的结果：

```
echo $SECRET_USERNAME
```

输出类似于：

```
admin
```

```
echo $SECRET_PASSWORD
```

输出类似于：

```
1f2d1e2e67df
```

Secret 更新之后对应的环境变量不会被更新

如果某个容器已经在通过环境变量使用某 Secret，对该 Secret 的更新不会被 容器马上看见，除非容器被重启。有一些第三方的解决方案能够在 Secret 发生变化时触发容器重启。

不可更改的 Secret

FEATURE STATE: Kubernetes v1.19 [beta]

Kubernetes 的 alpha 特性 不可变的 Secret 和 ConfigMap 提供了一种可选配置，可以设置各个 Secret 和 ConfigMap 为不可变的。对于大量使用 Secret 的集群（至少有成千上万各不相同的 Secret 供 Pod 挂载），禁止变更它们的数据有下列好处：

- 防止意外（或非预期的）更新导致应用程序中断
- 通过将 Secret 标记为不可变来关闭 kube-apiserver 对其的监视，从而显著降低 kube-apiserver 的负载，提升集群性能。

这个特性通过 `ImmutableEmphemeralVolumes` [特性门控](#) 来控制，从 v1.19 开始默认启用。你可以通过将 Secret 的 `immutable` 字段设置为 `true` 创建不可更改的 Secret。例如：

```
apiVersion: v1
kind: Secret
metadata:
...
data:
...
immutable: true
```

说明：

一旦一个 Secret 或 ConfigMap 被标记为不可更改，撤销此操作或者更改 `data` 字段的内容都是不可能的。只能删除并重新创建这个 Secret。现有的 Pod 将维持对已删除 Secret 的挂载点 - 建议重新创建这些 Pod。

使用 `imagePullSecret`

`imagePullSecrets` 字段中包含一个列表，列举对同一名字空间中的 Secret 的引用。你可以使用 `imagePullSecrets` 将包含 Docker（或其他）镜像仓库密码的 Secret 传递给 kubelet。kubelet 使用此信息来替你的 Pod 拉取私有镜像。关于 `imagePullSecrets` 字段的更多信息，请参考 [PodSpec API](#) 文档。

手动指定 imagePullSecret

你可以阅读[容器镜像文档](#) 以了解如何设置 imagePullSecrets。

设置自动附加 imagePullSecrets

您可以手动创建 imagePullSecret，并在 ServiceAccount 中引用它。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 值。有关该过程的详细说明，请参阅[将 ImagePullSecrets 添加到服务帐户](#)。

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 帐户令牌的 Secret) 可以根据其服务帐户自动附加到 Pod。

详细说明

限制

Kubernetes 会验证 Secret 作为卷来源时所给的对象引用确实指向一个类型为 Secret 的对象。因此，Secret 需要先于任何依赖于它的 Pod 创建。

Secret API 对象处于某[名字空间](#) 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 导致 API 服务器和 kubelet 的内存耗尽。然而，创建过多较小的 Secret 也可能耗尽内存。更全面得限制 Secret 内存用量的功能还在计划中。

kubelet 仅支持从 API 服务器获得的 Pod 使用 Secret。这包括使用 kubectl 创建的所有 Pod，以及间接通过副本控制器创建的 Pod。它不包括通过 kubelet --manifest-url 标志，--config 标志或其 REST API 创建的 Pod (这些不是创建 Pod 的常用方法)。

以环境变量形式在 Pod 中使用 Secret 之前必须先创建 Secret，除非该环境变量被标记为可选的。Pod 中引用不存在的 Secret 时将无法启动。

使用 secretKeyRef 时，如果引用了指定 Secret 不存在的键，对应的 Pod 也无法启动。

对于通过 envFrom 填充环境变量的 Secret，如果 Secret 中包含的键名无法作为合法的环境变量名称，对应的键会被跳过，该 Pod 将被允许启动。不过这时会产生一个事件，其原因为 InvalidVariableNames，其消息中包含被跳过的无效键的列表。下面的示例显示一个 Pod，它引用了包含 2 个无效键 1badkey 和 2alsobad。

```
kubectl get events
```

输出类似于：

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 生命周期的关系

通过 API 创建 Pod 时，不会检查引用的 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 服务器建立连接，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 将创建并挂载一个包含它的卷。在 Pod 的所有卷被挂载之前，Pod 中的容器不会启动。

使用案例

案例：以环境变量的形式使用 Secret

创建一个 Secret 定义：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  USER_NAME: YWRtaW4=
  PASSWORD: MWYyZDFIMmU2N2Rm
```

生成 Secret 对象：

```
kubectl apply -f mysecret.yaml
```

使用 envFrom 将 Secret 的所有数据定义为容器的环境变量。Secret 中的键名称为 Pod 中的环境变量名称：

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
        - secretRef:
```

```
  name: mysecret
restartPolicy: Never
```

案例：包含 SSH 密钥的 Pod

创建一个包含 SSH 密钥的 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
```

输出类似于：

```
secret "ssh-key-secret" created
```

你也可以创建一个带有包含 SSH 密钥的 secretGenerator 字段的 kustomization.yaml 文件。

注意：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。你可以使用一个服务帐户，分享给 Kubernetes 集群中合适的用户，这些用户是你要分享的。如果服务账号遭到侵犯，可以将其收回。

现在我们可以创建一个 Pod，令其引用包含 SSH 密钥的 Secret，并通过存储卷来使用它：

```
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"
```

容器中的命令运行时，密钥的片段可以在以下目录找到：

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

然后容器可以自由使用 Secret 数据建立一个 SSH 连接。

案例：包含生产/测试凭据的 Pod

下面的例子展示的是两个 Pod。一个 Pod 使用包含生产环境凭据的 Secret，另一个 Pod 使用包含测试环境凭据的 Secret。

你可以创建一个带有 secretGenerator 字段的 kustomization.yaml 文件，或者执行 kubectl create secret：

```
kubectl create secret generic prod-db-secret \
--from-literal=username=produser \
--from-literal=password=Y4nys7f11
```

输出类似于：

```
secret "prod-db-secret" created
```

```
kubectl create secret generic test-db-secret \
--from-literal=username=testuser \
--from-literal=password=iluvtests
```

输出类似于：

```
secret "test-db-secret" created
```

说明：

特殊字符（例如 \$、\、*、= 和 !）会被你的 [Shell](#) 解释，因此需要转义。在大多数 Shell 中，对密码进行转义的最简单方式是用单引号（'）将其括起来。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=d
evuser --from-literal=password='S!B\*d$zDsb='
```

您无需对文件中的密码（--from-file）中的特殊字符进行转义。

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
```

将 Pod 添加到同一个 kustomization.yaml 文件

```
$ cat <<EOF >> kustomization.yaml
resources:
- pod.yaml
EOF
```

通过下面的命令应用所有对象

```
kubectl apply -k .
```

两个容器都会在其文件系统上存在以下文件，其中包含容器对应的环境的值：

```
/etc/secret-volume/username  
/etc/secret-volume/password
```

请注意，两个 Pod 的规约配置中仅有一个字段不同；这有助于使用共同的 Pod 配置模板创建具有不同能力的 Pod。

您可以使用两个服务账号进一步简化基本的 Pod 规约：

1. 名为 prod-user 的服务账号拥有 prod-db-secret
2. 名为 test-user 的服务账号拥有 test-db-secret

然后，Pod 规约可以缩短为：

```
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
```

案例：Secret 卷中以句点号开头的文件

你可以通过定义以句点开头的键名，将数据“隐藏”起来。例如，当如下 Secret 被挂载到 secret-volume 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"

卷中将包含唯一的叫做 .secret-file 的文件。容器 dotfile-test-container 中，该文件处于 /etc/secret-volume/.secret-file 路径下。

说明：以点号开头的文件在 ls -l 的输出中会被隐藏起来；列出目录内容时，必须使用 ls -la 才能看到它们。

案例：Secret 仅对 Pod 中的一个容器可见

考虑一个需要处理 HTTP 请求、执行一些复杂的业务逻辑，然后使用 HMAC 签署一些消息的应用。因为应用程序逻辑复杂，服务器中可能会存在一个未被注意的远程文件读取漏洞，可能会将私钥暴露给攻击者。

解决的办法可以是将应用分为两个进程，分别运行在两个容器中：前端容器，用于处理用户交互和业务逻辑，但无法看到私钥；签名容器，可以看到私钥，响应来自前端（例如通过本地主机网络）的简单签名请求。

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 [鉴权策略](#)，例如 [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。很多 Secret 都可能导致 Kubernetes 集群内部的权限越界（例如服务账号令牌）甚至逃逸到集群外部。即使某一个应用程序可以就所交互的 Secret 的能力作出正确抉择，但是同一命名空间中的其他应用程序却可能不这样做。

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能力，是应该避免的行为。列出 Secret 的操作可以让客户端检查该命名空间中存在的所有 Secret。在群集中 watch 和 list 所有 Secret 的能力应该只保留在特权最高的系统级组件。

需要访问 Secret API 的应用程序应该针对所需要的 Secret 执行 get 请求。这样，管理员就能限制对所有 Secret 的访问，同时为应用所需要的 [实例设置访问允许清单](#)。

为了获得高于轮询操作的性能，客户端设计资源时，可以引用 Secret，然后对资源执行 watch 操作，在引用更改时重新检索 Secret。此外，社区还存在一种 "[批量监控 API](#)" 的提案，允许客户端 watch 独立的资源，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性保护措施，例如，在可能的情况下避免将其写到磁盘。

只有当某节点上的 Pod 需要用到某 Secret 时，该 Secret 才会被发送到该节点上。Secret 不会被写入磁盘，而是被 kubelet 存储在 tmpfs 中。一旦依赖于它的 Pod 被删除，Secret 数据的本地副本就被删除。

同一节点上的很多个 Pod 可能拥有多个 Secret。但是，只有 Pod 所请求的 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。

同一个 Pod 中可能有多个容器。但是，Pod 中的每个容器必须通过 volumeMounts 请求挂载 Secret 卷才能使卷中的 Secret 对容器可见。这一实现可以用于在 Pod 级别[构建安全分区](#)。

在大多数 Kubernetes 发行版中，用户与 API 服务器之间的通信以及从 API 服务器到 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]

你可以为 Secret 数据开启[静态加密](#)，这样 Secret 数据就不会以明文形式存储到[etcd](#) 中。

风险

- API 服务器上的 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（要求 v1.13 或者更高版本）。
 - 管理员应该限制只有 admin 用户能访问 etcd；
 - API 服务器中的 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，该密码将会被泄露。Base64 编码不是一种加密方式，应该视同纯文本。
- 应用程序在从卷中读取 Secret 后仍然需要保护 Secret 的值，例如不会意外将其写入日志或发送给不信任方。
- 可以创建使用 Secret 的 Pod 的用户也可以看到该 Secret 的值。即使 API 服务器策略不允许用户读取 Secret 对象，用户也可以运行 Pod 导致 Secret 暴露。

- 目前，任何节点的 root 用户都可以通过模拟 kubelet 来读取 API 服务器中的任何 Secret。仅向实际需要 Secret 的节点发送 Secret 数据才能限制节点的 root 账号漏洞的影响，该功能还在计划中。

接下来

- 学习如何[使用 kubectl 管理 Secret](#)
- 学习如何[使用配置文件管理 Secret](#)
- 学习如何[使用 kustomize 管理 Secret](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 18, 2021 at 10:22 AM PST: [\[zh\] Clean PodPreset related pages/examples \(b9265d377\)](#)

为容器管理资源

当你定义 [Pod](#) 时可以选择性地为每个 [容器](#) 设定所需要的资源数量。最常见的可设定资源是 CPU 和内存 (RAM) 大小；此外还有其他类型的资源。

当你为 Pod 中的 Container 指定了资源 [请求](#) 时，调度器就利用该信息决定将 Pod 调度到哪个节点上。当你还为 Container 指定了资源 [约束](#) 时，kubelet 就可以确保运行的容器不会使用超出所设约束的资源。kubelet 还会为容器预留所 [请求](#) 数量的系统资源，供其使用。

请求和约束

如果 Pod 运行所在的节点具有足够的可用资源，容器可能（且可以）使用超出对应资源 `request` 属性所设置的资源量。不过，容器不可以使用超出其资源 `limit` 属性所设置的资源量。

例如，如果你将容器的 `memory` 的请求量设置为 256 MiB，而该容器所处的 Pod 被调度到一个具有 8 GiB 内存的节点上，并且该节点上没有其他 Pods 运行，那么该容器就可以尝试使用更多的内存。

如果你将某容器的 `memory` 约束设置为 4 GiB，kubelet（和 [容器运行时](#)）就会确保该约束生效。容器运行时会禁止容器使用超出所设置资源约束的资源。例如：当容器中进程尝试使用超出所允许内存量的资源时，系统内核会将尝试申请内存的进程终止，并引发内存不足 (OOM) 错误。

约束值可以以被动方式来实现（系统会在发现违例时进行干预），或者通过强制生效的方式实现（系统会避免容器用量超出约束值）。不同的容器运行时采用不同方式来实现相同的限制。

说明：

如果某 Container 设置了自己的内存限制但未设置内存请求，Kubernetes 自动为其设置与内存限制相匹配的请求值。类似的，如果某 Container 设置了 CPU 限制值但未设置 CPU 请求值，则 Kubernetes 自动为其设置 CPU 请求并使之与 CPU 限制值匹配。

资源类型

CPU 和内存都是资源类型。每种资源类型具有其基本单位。CPU 表达的是计算处理能力，其单位是 [Kubernetes CPUs](#)。内存的单位是字节。如果你使用的是 Kubernetes v1.14 或更高版本，则可以指定巨页（Huge Page）资源。巨页是 Linux 特有的功能，节点内核在其中分配的内存块比默认页大小大得多。

例如，在默认页面大小为 4KiB 的系统上，你可以指定约束 hugepages-2Mi: 80Mi。如果容器尝试分配 40 个 2MiB 大小的巨页（总共 80 MiB），则分配请求会失败。

说明：

你不能过量使用 hugepages-* 资源。这与 memory 和 cpu 资源不同。

CPU 和内存统称为计算资源，或简称为资源。计算资源的数量是可测量的，可以被请求、被分配、被消耗。它们与 [API 资源](#) 不同。API 资源（如 Pod 和 [Service](#)）是可通过 Kubernetes API 服务器读取和修改的对象。

Pod 和 容器的资源请求和约束

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.limits.hugepages-<size>
- spec.containers[].resources.requests.cpu
- spec.containers[].resources.requests.memory
- spec.containers[].resources.requests.hugepages-<size>

尽管请求和限制值只能在单个容器上指定，我们仍可方便地计算出 Pod 的资源请求和约束。Pod 对特定资源类型的请求/约束值是 Pod 中各容器对该类型资源的请求/约束值的总和。

Kubernetes 中的资源单位

CPU 的含义

CPU 资源的约束和请求以 *cpu* 为单位。

Kubernetes 中的一个 *cpu* 等于云平台上的 **1 个 vCPU/核** 和裸机 Intel 处理器上的 ****1 个超线程 ****。

你也可以表达带小数 CPU 的请求。spec.containers[].resources.requests.cpu 为 0.5 的 Container 肯定能够获得请求 1 CPU 的容器的一半 CPU 资源。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；最大精度是 1m。因此，或许你应该优先考虑使用 100m 的形式。

CPU 总是按绝对数量来请求的，不可以使用相对数量；0.1 的 CPU 在单核、双核、48 核的机器上的意义是一样的。

内存的含义

内存的约束和请求以字节为单位。你可以使用以下后缀之一以一般整数或定点数字形式来表示内存：E、P、T、G、M、K。你也可以使用对应的 2 的幂数：Ei、Pi、Ti、Gi、Mi、Ki。例如，以下表达式所代表的是大致相同的值：

128974848、129e6、129M、123Mi

下面是个例子。

以下 Pod 有两个 Container。每个 Container 的请求为 0.25 cpu 和 64MiB (2^{26} 字节) 内存，每个容器的资源约束为 0.5 cpu 和 128MiB 内存。你可以认为该 Pod 的资源请求为 0.5 cpu 和 128 MiB 内存，资源限制为 1 cpu 和 256MiB 内存。

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
```

```
limits:  
  memory: "128Mi"  
  cpu: "500m"  
- name: log-aggregator  
  image: images.my-company.example/log-aggregator:v6  
  resources:  
    requests:  
      memory: "64Mi"  
      cpu: "250m"  
    limits:  
      memory: "128Mi"  
      cpu: "500m"
```

带资源请求的 Pod 如何调度

当你创建一个 Pod 时，Kubernetes 调度程序将为 Pod 选择一个节点。每个节点对每种资源类型都有一个容量上限：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，所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，如果容量检查失败，调度程序仍会拒绝在该节点上放置 Pod。当稍后节点上资源用量增加，例如到达请求率的每日峰值区间时，节点上也不会出现资源不足的问题。

带资源约束的 Pod 如何运行

当 kubelet 启动 Pod 中的 Container 时，它会将 CPU 和内存约束信息传递给容器运行时。

当使用 Docker 时：

- spec.containers[].resources.requests.cpu 先被转换为可能是小数的基础值，再乘以 1024。这个数值和 2 的较大者用作 docker run 命令中的 [--cpu-shares](#) 标志的值。
- spec.containers[].resources.limits.cpu 先被转换为 millicore 值，再乘以 100。其结果就是每 100 毫秒容器可以使用的 CPU 时间总量。在此期间（100ms），容器所使用的 CPU 时间不会超过它被分配的时间。

说明：默认的配额（Quota）周期为 100 毫秒。CPU 配额的最小精度为 1 毫秒。

- spec.containers[].resources.limits.memory 被转换为整数值，作为 docker run 命令中的 [--memory](#) 参数值。

如果 Container 超过其内存限制，则可能会被终止。如果容器可重新启动，则与所有其他类型的运行时失效一样，kubelet 将重新启动容器。

如果一个 Container 内存用量超过其内存请求值，那么当节点内存不足时，容器所处的 Pod 可能被逐出。

每个 Container 可能被允许也可能不被允许使用超过其 CPU 约束的处理时间。但是，容器不会由于 CPU 使用率过高而被杀死。

要确定 Container 是否会由于资源约束而无法调度或被杀死，请参阅[疑难解答](#) 部分。

监控计算和内存资源用量

Pod 的资源使用情况是作为 Pod 状态的一部分来报告的。

如果为集群配置了可选的 [监控工具](#)，则可以直接从 [指标 API](#) 或者监控工具获得 Pod 的资源使用情况。

本地临时存储

FEATURE STATE: Kubernetes v1.10 [beta]

节点通常还可以具有本地的临时性存储，由本地挂接的可写入设备或者有时也用 RAM 来提供支持。“临时（Ephemeral）”意味着对所存储的数据不提供长期可用性的保证。

Pods 通常可以使用临时性本地存储来实现缓冲区、保存日志等功能。kubelet 可以为使用本地临时存储的 Pods 提供这种存储空间，允许后者使用 [emptyDir](#) 类型的 [卷](#) 将其挂载到容器中。

kubelet 也使用此类存储来保存 [节点层面的容器日志](#)，容器镜像文件、以及运行中容器的可写入层。

注意：如果节点失效，存储在临时性存储中的数据会丢失。你的应用不能对本地临时性存储的性能 SLA（例如磁盘 IOPS）作任何假定。

作为一种 beta 阶段功能特性，Kubernetes 允许你跟踪、预留和限制 Pod 可消耗的临时性本地存储数量。

本地临时性存储的配置

Kubernetes 有两种方式支持节点上配置本地临时性存储：

- [单一文件系统](#)
- [双文件系统](#)

采用这种配置时，你会把所有类型的临时性本地数据（包括 emptyDir 卷、可写入容器层、容器镜像、日志等）放到同一个文件系统中。作为最有效的 kubelet 配置方式，这意味着该文件系统是专门提供给 Kubernetes（kubelet）来保存数据的。

kubelet 也会生成 [节点层面的容器日志](#)，并按临时性本地存储的方式对待之。

kubelet 会将日志写入到所配置的日志目录（默认为 /var/log）下的文件中；还会针对其他本地存储的数据使用同一个基础目录（默认为 /var/lib/kubelet）。

通常，/var/lib/kubelet 和 /var/log 都是在系统的根文件系统中。kubelet 的设计也考虑到这一点。

你的集群节点当然可以包含其他的、并非用于 Kubernetes 的很多文件系统。

你使用节点上的某个文件系统来保存运行 Pods 时产生的临时性数据：日志和 emptyDir 卷等。你可以使用这个文件系统来保存其他数据（例如：与 Kubernetes 无关的其他系统日志）；这个文件系统还可以是根文件系统。

kubelet 也将 [节点层面的容器日志](#) 写入到第一个文件系统中，并按临时性本地存储的方式对待之。

同时你使用另一个由不同逻辑存储设备支持的文件系统。在这种配置下，你会告诉 kubelet 将容器镜像层和可写层保存到这第二个文件系统上的某个目录中。

第一个文件系统中不包含任何镜像层和可写层数据。

当然，你的集群节点上还可以有很多其他与 Kubernetes 没有关联的文件系统。

kubelet 能够度量其本地存储的用量。实现度量机制的前提是：

- LocalStorageCapacityIsolation [特性门控](#) 被启用（默认状态），并且
- 你已经对节点进行了配置，使之使用所支持的本地临时性储存配置方式之一

如果你的节点配置不同于以上预期，kubelet 就无法对临时性本地存储的资源约束实施限制。

说明： kubelet 会将 tmpfs emptyDir 卷的用量当作容器内存用量，而不是本地临时性存储来统计。

为本地临时性存储设置请求和约束值

你可以使用 *ephemeral-storage* 来管理本地临时性存储。Pod 中的每个 Container 可以设置以下属性：

- spec.containers[].resources.limits.ephemeral-storage
- spec.containers[].resources.requests.ephemeral-storage

ephemeral-storage 的请求和约束值是按字节计量的。你可以使用一般整数或者定点数字 加上下面的后缀来表达存储量：E、P、T、G、M、K。你也可以使用对应的 2 的幂级数来表达：Ei、Pi、Ti、Gi、Mi、Ki。例如，下面的表达式所表达的大致是同一个值：

```
128974848, 129e6, 129M, 123Mi
```

在下面的例子中，Pod 包含两个 Container。每个 Container 请求 2 GiB 大小的本地临时性存储。每个 Container 都设置了 4 GiB 作为其本地临时性存储的约束值。因此，整个 Pod 的本地临时性存储请求是 4 GiB，且其本地临时性存储的约束为 8 GiB。

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
```

```
containers:
- name: app
  image: images.my-company.example/app:v4
  resources:
    requests:
      ephemeral-storage: "2Gi"
    limits:
      ephemeral-storage: "4Gi"
- name: log-aggregator
  image: images.my-company.example/log-aggregator:v6
  resources:
    requests:
      ephemeral-storage: "2Gi"
    limits:
      ephemeral-storage: "4Gi"
```

带临时性存储的 Pods 的调度行为

当你创建一个 Pod 时，Kubernetes 调度器会为 Pod 选择一个节点来运行之。每个节点都有一个本地临时性存储的上限，是其可提供给 Pods 使用的总量。欲了解更多信息，可参考 [节点可分配资源](#) 节。

调度器会确保所调度的 Containers 的资源请求总和不会超出节点的资源容量。

临时性存储消耗的管理

如果 kubelet 将本地临时性存储作为资源来管理，则 kubelet 会度量以下各处的存储用量：

- emptyDir 卷，除了 *tmpfs* emptyDir 卷
- 保存节点层面日志的目录
- 可写入的容器镜像层

如果某 Pod 的临时存储用量超出了你所允许的范围，kubelet 会向其发出逐出（eviction）信号，触发该 Pod 被逐出所在节点。

就容器层面的隔离而言，如果某容器的可写入镜像层和日志用量超出其存储约束，kubelet 也会将所在的 Pod 标记为逐出候选。

就 Pod 层面的隔离而言，kubelet 会将 Pod 中所有容器的约束值相加，得到 Pod 存储约束的总值。如果所有容器的本地临时性存储用量总和加上 Pod 的 emptyDir 卷的用量超出 Pod 存储约束值，kubelet 也会将该 Pod 标记为逐出候选。

注意：

如果 kubelet 没有度量本地临时性存储的用量，即使 Pod 的本地存储用量超出其约束值也不会被逐出。

不过，如果用于可写入容器镜像层、节点层面日志或者 emptyDir 卷的文件系统中可用空间太少，节点会为自身设置本地存储不足的污点 标签。这一污点会触发对那些无法容忍该污点的 Pods 的逐出操作。

关于临时性本地存储的配置信息，请参考[这里](#)

kubelet 支持使用不同方式来度量 Pod 的存储用量：

- [周期性扫描](#)
- [文件系统项目配额](#)

kubelet 按预定周期执行扫描操作，检查 emptyDir 卷、容器日志目录以及可写入容器镜像层。

这一扫描会度量存储空间用量。

说明：

在这种模式下，kubelet 并不检查已删除文件所对应的、仍处于打开状态的文件描述符。

如果你（或者容器）在 emptyDir 卷中创建了一个文件，写入一些内容之后再次打开该文件并执行了删除操作，所删除文件对应的 inode 仍然存在，直到你关闭该文件为止。kubelet 不会将该文件所占用的空间视为已使用空间。

FEATURE STATE: Kubernetes v1.15 [alpha]

项目配额（Project Quota）是一个操作系统层的功能特性，用来管理文件系统中的存储用量。在 Kubernetes 中，你可以启用项目配额以监视存储用量。你需要确保节点上为 emptyDir 提供存储的文件系统支持项目配额。例如，XFS 和 ext4fs 文件系统都支持项目配额。

说明：项目配额可以帮你监视存储用量，但无法对存储约束执行限制。

Kubernetes 所使用的项目 ID 始于 1048576。所使用的 IDs 会注册在 /etc/projects 和 /etc/projid 文件中。如果该范围中的项目 ID 已经在系统中被用于其他目的，则已占用的项目 IDs 也必须注册到 /etc/projects 和 /etc/projid 中，这样 Kubernetes 才不会使用它们。

配额方式与目录扫描方式相比速度更快，结果更精确。当某个目录被分配给某个项目时，该目录下所创建的所有文件都属于该项目，内核只需要跟踪该项目中的文件所使用的存储块个数。如果某文件被创建后又被删除，但对应文件描述符仍处于打开状态，该文件会继续耗用存储空间。配额跟踪技术能够精确第记录对应存储空间的状态，而目录扫描方式会忽略被删除文件所占用的空间。

如果你希望使用项目配额，你需要：

- 在 kubelet 配置中启用 LocalStorageCapacityIsolationFSQuotaMonitoring=true 特性门控。

- 确保根文件系统（或者可选的运行时文件系统）启用了项目配额。所有 XFS 文件系统都支持项目配额。对 extf 文件系统而言，你需要在文件系统尚未被挂载时启用项目配额跟踪特性：

```
# 对 ext4 而言，在 /dev/block-device 尚未被挂载时执行下面操作  
sudo tune2fs -O project -Q prjquota /dev/block-device
```

- 确保根文件系统（或者可选的运行时文件系统）在挂载时项目配额特性是被启用的。对于 XFS 和 ext4fs 而言，对应的挂载选项称作 prjquota。

扩展资源 (Extended Resources)

扩展资源是 kubernetes.io 域名之外的标准资源名称。它们使得集群管理员能够颁布非 Kubernetes 内置资源，而用户可以使用它们。

使用扩展资源需要两个步骤。首先，集群管理员必须颁布扩展资源。其次，用户必须在 Pod 中请求扩展资源。

管理扩展资源

节点级扩展资源

节点级扩展资源绑定到节点。

设备插件管理的资源

有关如何颁布在各节点上由设备插件所管理的资源，请参阅 [设备插件](#)。

其他资源

为了颁布新的节点级扩展资源，集群操作员可以向 API 服务器提交 PATCH HTTP 请求，以在集群中节点的 status.capacity 中为其配置可用数量。完成此操作后，节点的 status.capacity 字段中将包含新资源。kubelet 会异步地对 status.allocatable 字段执行自动更新操作，使之包含新资源。请注意，由于调度器在评估 Pod 是否适合在某节点上执行时会使用节点的 status.allocatable 值，在更新节点容量使之包含新资源之后和请求该资源的第一个 Pod 被调度到该节点之间，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 构造 HTTP 请求，公告主节点为 k8s-master 的节点 k8s-node-1 上存在五个 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
```

说明：在前面的请求中，~1 是在 patch 路径中对字符 / 的编码。JSON-Patch 中的操作路径的值被视为 JSON-Pointer 类型。有关更多详细信息，请参见 [IETF RFC 6901 第 3 节](#)。

集群层面的扩展资源

集群层面的扩展资源并不绑定到具体节点。它们通常由调度器扩展程序（Scheduler Extenders）管理，这些程序处理资源消耗和资源配置。

你可以在[调度器策略配置](#) 中指定由调度器扩展程序处理的扩展资源。

示例：

下面的调度器策略配置标明集群层扩展资源 "example.com/foo" 由调度器扩展程序处理。

- 仅当 Pod 请求 "example.com/foo" 时，调度器才会将 Pod 发送到调度器扩展程序。
- ignoredByScheduler 字段指定调度器不要在其 PodFitsResources 断言中检查 "example.com/foo" 资源。

```
{  
  "kind": "Policy",  
  "apiVersion": "v1",  
  "extenders": [  
    {  
      "urlPrefix": "<extender-endpoint>",  
      "bindVerb": "bind",  
      "managedResources": [  
        {  
          "name": "example.com/foo",  
          "ignoredByScheduler": true  
        }  
      ]  
    }  
  ]  
}
```

使用扩展资源

就像 CPU 和内存一样，用户可以在 Pod 的规约中使用扩展资源。调度器负责资源的核算，确保同时分配给 Pod 的资源总量不会超过可用数量。

说明：扩展资源取代了非透明整数资源（Opaque Integer Resources，OIR）。用户可以使用 kubernetes.io（保留）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。

说明：扩展资源不能过量使用，因此如果容器规范中同时存在请求和约束，则它们的取值必须相同。

仅当所有资源请求（包括 CPU、内存和任何扩展资源）都被满足时，Pod 才能被调度。在资源请求无法满足时，Pod 会保持在 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
```

PID 限制

进程 ID (PID) 限制允许对 kubelet 进行配置，以限制给定 Pod 可以消耗的 PID 数量。有关信息，请参见 [PID 限制](#)。

疑难解答

我的 Pod 处于悬决状态且事件信息显示 failedScheduling

如果调度器找不到该 Pod 可以匹配的任何节点，则该 Pod 将保持未被调度状态，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 无法被调度。由于内存不足（PodExceedsFreeMemory）而导致失败时，也有类似的错误消息。一般来说，如果 Pod 处于悬决状态且有这种类型的消息时，你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，为悬决的 Pod 腾出空间。
- 检查 Pod 所需的资源是否超出所有节点的资源容量。例如，如果所有节点的容量都是cpu : 1，那么一个请求为 cpu: 1.1 的 Pod 永远不会被调度。

你可以使用 kubectl describe nodes 命令检查节点容量和已分配的资源数量。例如：

```
kubectl describe nodes e2e-test-node-pool-4lw4
```

```
Name:           e2e-test-node-pool-4lw4
[ ... 这里忽略了若干行以便阅读 ...]
Capacity:
cpu:            2
memory:         7679792Ki
pods:           110
Allocatable:
cpu:           1800m
memory:        7474992Ki
pods:          110
[ ... 这里忽略了若干行以便阅读 ...]
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，你可以看到如果 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，你将看到哪些 Pod 占用了节点上的资源。

可供 Pod 使用的资源量小于节点容量，因为系统守护程序也会使用一部分可用资源。[NodeStatus](#) 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](#)。

可以配置 [资源配置](#) 功能特性以限制可以使用的资源总量。如果与名字空间配合一起使用，就可以防止一个团队占用所有资源。

我的容器被终止了

你的容器可能因为资源紧张而被终止。要查看容器是否因为遇到资源限制而被杀死，请针对相关的 Pod 执行 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:          None
Message:         None
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。

你可以使用 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://
0e4095bba1feccdf7ef9fb6ebffe972b4b14285d5acdec6f0d3ae8a22fad8b2]]
```

你可以看到容器因为 reason:OOM killed 而被终止，OOM 表示内存不足（Out Of Memory）。

接下来

- 获取[分配内存资源给容器和 Pod 的实践经验](#)
- 获取[分配 CPU 资源给容器和 Pod 的实践经验](#)
- 关于请求和约束之间的区别，细节信息可参见[服务质量](#)
- 阅读 API 参考文档中 [Container](#) 部分。
- 阅读 API 参考文档中 [ResourceRequirements](#) 部分。
- 阅读 XFS 中关于[项目配额](#) 的文档。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 18, 2020 at 4:57 AM PST: [\[zh\] sync manage-resources-containers.md \(7dcba3fab\)](#)

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。

说明：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](#) 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](#)。

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](#)。

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。
1. 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。
1. 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](#)
- [kubectl config](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 July 28, 2020 at 3:27 PM PST: [\[zh\] Fix links in concepts section \(1\)](#) ([8d7fd3f43](#))

Pod 优先级与抢占

FEATURE STATE: Kubernetes v1.14 [stable]

[Pods](#) 可以有优先级 (*Priority*)。优先级体现的是当前 Pod 与其他 Pod 相比的重要程度。如果 Pod 无法被调度，则调度器会尝试抢占（逐出）低优先级的 Pod，从而使得悬决的 Pod 可被调度。

警告：

在一个并非所有用户都可信任的集群中，一个有恶意的用户可能创建优先级最高的 Pod，从而导致其他 Pod 被逐出或者无法调度。管理员可以使用 ResourceQuota 来避免用户创建高优先级的 Pod。

参考[限制默认使用的优先级类](#)以了解更多细节。

如何使用优先级和抢占

要使用优先级和抢占特性：

1. 添加一个或多个 [PriorityClasses](#) 对象
2. 创建 Pod 时设置其 [priorityClassName](#) 为所添加的 PriorityClass 之一。当然你也不必一定要直接创建 Pod；通常你会在一个集合对象（如 Deployment）的 Pod 模板中添加 priorityClassName。

关于这些步骤的详细信息，请继续阅读。

说明：Kubernetes 发行时已经带有两个 PriorityClasses：system-cluster-critical 和 system-node-critical。这些优先级类是公共的，用来 [确保关键组件总是能够先被调度](#)。

如何禁用抢占

注意：关键 Pod 依赖调度器抢占机制以在集群资源压力较大时得到调度。因此，不建议禁用抢占。

说明：在 Kubernetes 1.15 及之后版本中，如果特性门控 NonPreemptingPriority 被启用，则 PriorityClass 对象可以选择设置 preemptionPolicy: Never。这样就会避免属于该 PriorityClass 的 Pod 抢占其他 Pod。

抢占能力是通过 kube-scheduler 的标志 disablePreemption 来控制的，该标志默认为 false。如果你在了解上述提示的前提下仍希望禁用抢占，可以将 disablePreemption 设置为 true。

这一选项只能通过组件配置来设置，无法通过命令行选项这种较老的形式设置。下面是禁用抢占的组件配置示例：

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider
```

...

```
disablePreemption: true
```

PriorityClass

PriorityClass 是一种不属于任何名字空间的对象，定义的是从优先级类名向优先级整数值的映射。优先级类名称用 PriorityClass 对象的元数据的 name 字段指定。优先级整数值在必须提供的 value 字段中指定。优先级值越大，优先级越高。PriorityClass 对象的名称必须是合法的 [DNS 子域名](#) 且不可包含 system- 前缀。

PriorityClass 对象可以设置数值小于等于 10 亿的 32 位整数。更大的数值保留给那些通常不可被抢占或逐出的系统 Pod。集群管理员应该为每个优先级值映射创建一个 PriorityClass 对象。

PriorityClass 对象还有两个可选字段：globalDefault 和 description。前者用来表明此 PriorityClass 的数值应该用于未设置 priorityClassName 的 Pod。系统中只能存在一个 globalDefault 设为真的 PriorityClass 对象。如果没有 PriorityClass 对象的 globalDefault 被设置，则未设置 priorityClassName 的 Pod 的优先级为 0。

description 字段可以设置任意字符串值。其目的是告诉用户何时该使用该 PriorityClass。

关于 Pod 优先级与现有集群的说明

- 如果你要升级一个不支持 Pod 优先级的集群，现有 Pod 的有效优先级都被视为 0。
- 向集群中添加 globalDefault 设置为 true 的 PriorityClass 不会改变现有 Pod 的优先级。新添加的 PriorityClass 值仅适用于 PriorityClass 被添加之后 新建的 Pod。
- 如果你要删除 PriorityClass，则使用所删除的 PriorityClass 名称的现有 Pod 都不会受影响，但是你不可以再创建使用该 PriorityClass 名称的新 Pod。

PriorityClass 示例

```
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."
```

非抢占式的 PriorityClass

FEATURE STATE: Kubernetes v1.15 [alpha]

配置 `preemptionPolicy: Never` 的 Pod 在调度队列中会被放在低优先级的 Pod 的前面，但是它们不可以抢占其他 Pod。非抢占 Pod 会在调度队列中等待调度，直到有足够的空闲资源时才被调度。非抢占 Pod 与其他 Pod 一样，也受调度器回退（Back-off）机制影响。换言之，如果调度器尝试调度这些 Pod 时发现它们无法调度，它们会被再次尝试，并且重试的频率会被降低，这样可以使得其他优先级较低的 Pod 有机会在它们之前被调度。

非抢占 Pod 仍有可能被其他高优先级的 Pod 抢占。

`preemptionPolicy` 默认取值为 `PreemptLowerPriority`，这会使得该 `PriorityClass` 的 Pod 能够抢占低优先级的 Pod（这也是当前的默认行为）。如果 `preemptionPolicy` 被设置为 `Never`，则该 `PriorityClass` 下的 Pod 都是非抢占的。

使用 `preemptionPolicy` 字段要求启用 [NonPreemptingPriority 特性门控](#)。

一种示例应用场景是数据科学负载。用户可能希望所提交的 Job 比其他负载的优先级都高，但又不希望因为抢占运行中的 Pod 而丢弃现有工作。只要集群中“自然地”释放出足够的资源，配置了 `preemptionPolicy: Never` 的高优先级 Job 可以在队列中其他 Pod 之前获得调度机会。

非抢占 PriorityClass 示例

```
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 优先级

在已经创建了一个或多个 `PriorityClass` 对象之后，你就可以创建 Pod 并在其规约中指定这些 `PriorityClass` 的名字之一。优先级准入控制器使用 `priorityClassName` 字段来填充优先级整数值。如果所指定优先级类不存在，则 Pod 被拒绝。

下面的 YAML 是一个 Pod 配置，使用了前面例子中创建的 `PriorityClass`。优先级准入控制器检查 Pod 的规约并将 Pod 优先级解析为 1000000。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```
labels:  
  env: test  
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      imagePullPolicy: IfNotPresent  
  priorityClassName: high-priority
```

优先级对 Pod 调度顺序的影响

当集群启用了 Pod 优先级时，调度器会基于 Pod 的优先级来排序悬决的 Pod。新 Pod 会被放在调度队列中较低优先级的其他悬决 Pod 前面。因此，优先级较高的 Pod 在其调度需求被满足的前提下会比优先级低的 Pod 先被调度。如果优先级较高的 Pod 无法被调度，调度器会继续尝试调度其他较低优先级的 Pod。

抢占

Pod 被创建时会被放入一个队列中等待调度。调度器从队列中选择 Pod，尝试将其调度到某 Node 上。如果找不到能够满足 Pod 所设置需求的 Node，就会触发悬决 Pod 的抢占逻辑。假定 P 是悬决的 Pod，抢占逻辑会尝试找到一个这样的节点，在该节点上移除一个或者多个 优先级比 P 低的 Pod 后，P 就可以被调度到该节点。如果调度器能够找到这样的节点，该节点上的一个或者多个优先级较低的 Pod 就会被逐出。当被逐出的 Pod 从该节点上消失时，P 就可以调度到此节点。

暴露给用户的信息

当 Pod P 在节点 N 上抢占了一个或多个 Pod 时，Pod P 的状态中的`nominatedNodeName` 字段会被设置为节点 N 的名字。此字段有助于调度器跟踪为 P 所预留的资源，同时也给用户提供了其集群中发生的抢占的信息。

请注意，Pod P 不一定会被调度到其 "nominated node (提名节点) "。当选定的 Pod 被抢占时，它们都会有其体面终止时限 (Graceful Termination Period)。如果在调度器等待选定的 (被牺牲的) Pod 终止期间有新的节点可用，调度器会使用其他 节点来调度 Pod P。因此，Pod 中的 `nominatednodeName` 和 `nodeName` 并不总是相同。此外，如果调度器抢占了节点 N 上的 Pod，但接下来出现优先级比 P 还高的 Pod 要被调度，则调度器会把节点 N 让给新的优先级更高的 Pod。如果发生了这种情况，调度器会清除 Pod P 的 `nominatednodeName`。通过清除操作，调度器使得 Pod P 可以尝试抢占别的节点上的 Pod。

抢占的局限性

抢占牺牲者的体面终止期限

当 Pod 被抢占时，做出牺牲的 Pod 仍有各自的 [体面终止期限](#)。这些 Pod 可以在给定的期限内结束其工作并退出。如果它们不能及时退出则会被杀死。这一体面终止期限带来了一个时间空隙，跨度从调度器开始抢占 Pod 的那一刻到悬决 Pod (P) 可以被调度到节点 (N) 上的那一刻。与此同时，调度器还要继续调度其他悬决的 Pod。随着被抢占

的 Pod 退出或终止，调度器尝试继续尝试调度悬决队列中的 Pod。因此，从调度器抢占被牺牲的 Pod 到 Pod P 被调度，中间通常存在一个时间间隔。为了缩短此时间间隔，用户可以将低优先级的 Pod 的体面终止期限设置为 0 或者较小的数字。

PodDisruptionBudget 是被支持的，但不提供保证

[PodDisruptionBudget](#) (PDB) 的存在使得应用的属主能够限制多副本应用因主动干扰而同时离线的 Pod 的个数。Kubernetes 在抢占 Pod 时是可以支持 PDB 的，但对 PDB 的约束也仅限于尽力而为。调度器会尝试寻找不会因为抢占而违反其 PDB 约束的 Pod 作为牺牲品，不过如果找不到这样的待逐出 Pod，抢占行为仍会发生，低优先级的 Pod 仍会被逐出而不管是否违反其 PDB 约束。

低优先级 Pod 间的亲和性

只有对下面的问题的回答是肯定的时候，才会考虑在节点上执行抢占操作：“如果所有优先级低于悬决 Pod 的 Pod 都从节点上逐出，悬决 Pod 可以调度到此节点么？”

说明： 抢占操作不一定要逐出所有优先级较低的 Pod。如果少逐出几个 Pod 而不是逐出所有较低优先级的 Pod 即可令悬决 Pod 被调度，则优先级较低的 Pod 中只有一部分会被逐出。即便如此，对上述问题的回答仍须是肯定的。如果回答是否定的，Kubernetes 不会考虑在该节点上执行抢占操作。

如果悬决 Pod 与节点上一个或多个较低优先级的 Pod 之间存在 Pod 间亲和性关系，那些对应的低优先级 Pod 若被逐出则无法满足此亲和性规则。在这种场合下，调度器不会抢占节点上的任何 Pod。相反，它会尝试寻找其他节点。调度器可能能找到也可能找不到合适的节点。Kubernetes 并不保证悬决的 Pod 最终会被调度。

对此问题的一种解决方案是仅针对优先级相同或更高的 Pod 设置 Pod 间亲和性。

跨节点的抢占

假定当前正在考虑在节点 N 上执行抢占操作以便 Pod P 能够被调度到 N 上执行。可是只有当另一个节点上的某个 Pod 被抢占，P 才有可能在 N 上调度执行。例如：

- Pod P 正在考虑被调度到节点 N。
- Pod Q 正运行在节点 N 所处区域（Zone）的另一个节点上。
- Pod P 设置了区域范畴的与 Pod Q 的反亲和性（topologyKey: topology.kubernetes.io/zone）。
- Pod P 与区域中的其他 Pod 之间都不存在反亲和性关系。
- 为了将 P 调度到节点 N 上，Pod Q 可以被抢占，但是调度器不会执行跨节点的抢占操作。因此，Pod P 会被视为无法调度到节点 N 上执行。

如果 Pod Q 真的被从其节点上移除，Pod 间反亲和性的规则就会得到满足，Pod P 就有可能被调度到节点 N 上执行。

我们可能在将来版本中考虑添加跨节点的抢占能力。前提是在这方面有足够的需求，并且我们找到了性能可接受的算法。

故障排查

Pod 优先级和抢占机制可能产生一些不想看到的副作用。下面是一些可能存在的问题以及相应的处理方法。

Pod 被不必要地抢占

抢占操作会在集群中资源压力较大，进而无法为高优先级的悬决 Pod 腾出空间时发生。如果你不小心给某些 Pod 赋予了较高优先级，这些意外获得高优先级的 Pod 可能导致集群中出现抢占行为。Pod 优先级是通过在其规约中的 `priorityClassName` 来设定的。优先级的整数值被解析出来后会添加到 Pod 规约的 `priority` 字段。

要解决这一问题，你可以修改这些 Pod 的 `priorityClassName` 设置，使用优先级较低的优先级类，或者将该字段留空。空的 `priorityClassName` 默认解析为优先级 0。

Pod 被抢占时，被抢占的 Pod 会有对应的事件被记录下来。只有集群中无法为某 Pod 提供足够资源的时候才会发生抢占。在出现这种情况时，也只有悬决 Pod（抢占者）的优先级高于被牺牲的 Pod 的优先级时，才会发生抢占现象。当没有悬决 Pod，或者悬决 Pod 的优先级等于或者低于现有 Pod 时，都不应发生抢占行为。如果在这种条件下仍然发生了抢占，请登记一个 Issue。

Pod 被抢占但抢占者未被调度

当有 Pod 被抢占时，它们会得到各自的体面终止期限（默认为 30 秒）。如果被牺牲的 Pod 在此限期内未能终止，则 Pod 会被强制终止一旦所有被牺牲的 Pod 都已消失不见，抢占者 Pod 就可被调度。

在抢占者 Pod 等待被牺牲的 Pod 消失期间，可能有更高优先级的 Pod 被创建，且适合调度到同一节点。如果是这种情况，调度器会调度优先级更高的 Pod 而不是抢占者。

这是期望发生的行为：优先级更高的 Pod 应该取代优先级较低的 Pod。

高优先级的 Pod 比低优先级的 Pod 先被抢占

调度器尝试寻找可以运行悬决 Pod 的节点。如果找不到这样的节点，调度器会尝试从任一节点上逐出优先级较低的 Pod 以运行悬决 Pod。如果包含低优先级 Pod 的节点不适合用来运行悬决 Pod，调度器可能会选择其他的、运行着较高优先级（相对之前所评估的节点上的 Pod 而言）的 Pod 的节点来执行抢占操作。即使如此，被牺牲的 Pod 的优先级也必须比抢占者 Pod 的优先级低。

当有多个节点可供抢占时，调度器会选择 Pod 集合的优先级最低的节点。不过如果这些 Pod 上定义了 `PodDisruptionBudget` (PDB) 而且如果被抢占了的话就会违反 PDB，则调度器会选择另一个 Pod 集合优先级稍高的节点。

当存在多个节点可供抢占，但以上场景都不适用，则调度器会选择优先级最低的节点。

Pod 优先级与服务质量间关系

Pod 优先级与 [QoS 类](#) 是两个相互独立的功能特性，其间交互之处很少，并且不存在基于 Pod QoS 类来为其设置优先级方面的默认限制。调度器的抢占逻辑在选择抢占目标时不会考虑 QoS 因素。抢占考虑的是 Pod 优先级，并选择优先级最低的 Pod 作为抢占目标。只有移除最低优先级的 Pod 尚不足以允许调度器调度抢占者 Pod 或者最低优先级的 Pod 受到 Pod 干扰预算 (PDB) 保护时，才会考虑抢占优先级稍高的 Pod。

唯一同时考虑 QoS 和 Pod 优先级的组件是 kubelet，体现在其 [资源不足时的逐出操作](#)。kubelet 首先根据 Pod 对濒危资源的使用是否超出其请求值来选择要被逐出的 Pod，接下来对这些 Pod 按优先级排序，再按其相对 Pod 的调度请求所耗用的濒危资源的用量排序。更多细节可参阅 [逐出最终用户的 Pod](#)。

kubelet 资源不足时的逐出操作不会逐出 Pod 资源用量未超出其请求值的 Pod。如果优先级较低的 Pod 未超出其请求值，它们不会被逐出。其他优先级较高的且用量超出请求值的 Pod 则可能被逐出。

接下来

- 阅读结合 PriorityClass 来使用 ResourceQuota 的介绍：[限制默认可使用的优先级类](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

安全

确保云原生工作负载安全的一组概念。

[Pod 安全性标准](#)

[云原生安全概述](#)

[Kubernetes API 访问控制](#)

Pod 安全性标准

Pod 的安全性配置一般通过使用 [安全性上下文 \(Security Context \)](#) 来保证。安全性上下文允许用户逐个 Pod 地定义特权级及访问控制。

以前，对集群的安全性上下文的需求的实施及其基于策略的定义都通过使用 [Pod 安全性策略](#) 来实现。*Pod 安全性策略 (Pod Security Policy)* 是一种集群层面的资源，控制 Pod 规约中 安全性敏感的部分。

不过，新的策略实施方式不断涌现，或增强或替换 PodSecurityPolicy 的使用。本页的目的是详细介绍建议实施的 Pod 安全框架；这些内容与具体的实现无关。

策略类型

在进一步讨论整个策略谱系之前，有必要对基本的策略下个定义。策略可以是很严格的也可以是很宽松的：

- **Privileged** - 不受限制的策略，提供最大可能范围的权限许可。这些策略 允许已知的特权提升。
- **Baseline/Default** - 限制性最弱的策略，禁止已知的策略提升。 允许使用默认的（规定最少）Pod 配置。
- **Restricted** - 限制性非常强的策略，遵循当前的保护 Pod 的最佳实践。

策略

Privileged

Privileged 策略是有目的地开放且完全无限制的策略。此类策略通常针对由 特权较高、受信任的用户所管理的系统级或基础设施级负载。

Privileged 策略定义中限制较少。对于默认允许（Allow-by-default）实施机制（例如 gatekeeper），Privileged 框架可能意味着不应用任何约束而不是实施某策略实例。与此不同，对于默认拒绝（Deny-by-default）实施机制（如 Pod 安全策略）而言，Privileged 策略应该默认允许所有控制（即，禁止所有限制）。

Baseline/Default

Baseline/Default 策略的目标是便于常见的容器化应用采用，同时禁止已知的特权提升。此策略针对的是应用运维人员和非关键性应用的开发人员。下面列举的控制应该被实施（禁止）：

控制 (Control)	策略 (Policy)
-------------------	---------------

宿主名字空间	<p>必须禁止共享宿主名字空间。</p> <p>限制的字段： spec.hostNetwork spec.hostPID spec.hostIPC</p> <p>允许的值： false</p>
特权容器	<p>特权 Pod 禁用大多数安全性机制，必须被禁止。</p> <p>限制的字段： spec.containers[*].securityContext.privileged spec.initContainers[*].securityContext.privileged</p> <p>允许的值： false、未定义/nil</p>
权能	<p>必须禁止添加默认集合之外的权能。</p> <p>限制的字段： spec.containers[*].securityContext.capabilities.add spec.initContainers[*].securityContext.capabilities.add</p> <p>允许的值： 空 (或限定为一个已知列表)</p>
HostPath 卷	<p>必须禁止 HostPath 卷。</p> <p>限制的字段： spec.volumes[*].hostPath</p> <p>允许的值： 未定义/nil</p>
宿主端口	<p>应禁止使用宿主端口，或者至少限定为已知列表。</p> <p>限制的字段： spec.containers[*].ports[*].hostPort spec.initContainers[*].ports[*].hostPort</p> <p>允许的值： 0、未定义 (或限定为已知列表)</p>
AppArmor (可选)	<p>在受支持的宿主上，默认应用 'runtime/default' AppArmor Profile。默认策略应禁止重载或者禁用该策略，或将重载限定未所允许的 profile 集合。</p> <p>限制的字段： metadata.annotations['container.apparmor.security.beta.kubernetes.io/*']</p> <p>允许的值： 'runtime/default'、未定义</p>

SELinux (可选)	<p>应禁止设置定制的 SELinux 选项。</p> <p>限制的字段 : spec.securityContext.seLinuxOptions spec.containers[*].securityContext.seLinuxOptions spec.initContainers[*].securityContext.seLinuxOptions</p> <p>允许的值 : undefined/nil</p>
/proc 挂载类型	<p>要求使用默认的 /proc 掩码以减小攻击面。</p> <p>限制的字段 : spec.containers[*].securityContext.procMount spec.initContainers[*].securityContext.procMount</p> <p>允许的值 : 未定义/nil、'Default'</p>
Sysctls	<p>Sysctls 可以禁用安全机制或影响宿主上所有容器，因此除了若干『安全』的子集之外，应该被禁止。如果某 sysctl 是受容器或 Pod 的名字空间限制，且与节点上其他 Pod 或进程相隔离，可认为是安全的。</p> <p>限制的字段 : spec.securityContext.sysctls</p> <p>允许的值 : kernel.shm_rmid_forced net.ipv4.ip_local_port_range net.ipv4.tcp_syncookies net.ipv4.ping_group_range 未定义/空值</p>

Restricted

Restricted 策略旨在实施当前保护 Pod 的最佳实践，尽管这样作可能会牺牲一些兼容性。该类策略主要针对运维人员和安全性很重要的应用的开发人员，以及不太被信任的用户。下面列举的控制需要被实施（禁止）：

控制 (Control)	策略 (Policy)
<i>Default 策略的所有要求。</i>	

	<p>除了限制 HostPath 卷之外，此类策略还限制可以通过 PersistentVolumes 定义的非核心卷类型。</p> <p>限制的字段：</p> <pre>spec.volumes[*].hostPath spec.volumes[*].gcePersistentDisk spec.volumes[*].awsElasticBlockStore spec.volumes[*].gitRepo spec.volumes[*].nfs spec.volumes[*].iscsi spec.volumes[*].glusterfs spec.volumes[*].rbd spec.volumes[*].flexVolume spec.volumes[*].cinder spec.volumes[*].cephFS spec.volumes[*].flocker spec.volumes[*].fc spec.volumes[*].azureFile spec.volumes[*].vsphereVolume spec.volumes[*].quobyte spec.volumes[*].azureDisk spec.volumes[*].portworxVolume spec.volumes[*].scaleIO spec.volumes[*].storageos spec.volumes[*].csi</pre> <p>允许的值：未定义/nil</p>
特权提升	<p>禁止（通过 SetUID 或 SetGID 文件模式）获得特权提升。</p> <p>限制的字段：</p> <pre>spec.containers[*].securityContext.allowPrivilegeEscalation spec.initContainers[*].securityContext.allowPrivilegeEscalation</pre> <p>允许的值：false</p>
以非 root 账号运行	<p>必须要求容器以非 root 用户运行。</p> <p>限制的字段：</p> <pre>spec.securityContext.runAsNonRoot spec.containers[*].securityContext.runAsNonRoot spec.initContainers[*].securityContext.runAsNonRoot</pre> <p>允许的值：true</p>

非 root 组 (可选)	<p>禁止容器使用 root 作为主要或辅助 GID 来运行。</p> <p>限制的字段 :</p> <ul style="list-style-type: none"> spec.securityContext.runAsGroup spec.securityContext.supplementalGroups[*] spec.securityContext.fsGroup spec.containers[*].securityContext.runAsGroup spec.initContainers[*].securityContext.runAsGroup <p>允许的值 :</p> <ul style="list-style-type: none"> 非零值 未定义/nil (*.runAsGroup 除外)
Seccomp	<p>必须要求使用 RuntimeDefault seccomp profile 或者允许使用特定的 profiles。</p> <p>限制的字段 :</p> <ul style="list-style-type: none"> spec.securityContext.seccompProfile.type spec.containers[*].securityContext.seccompProfile spec.initContainers[*].securityContext.seccompProfile <p>允许的值 :</p> <ul style="list-style-type: none"> 'runtime/default' 未定义/nil

策略实例化

将策略定义从策略实例中解耦出来有助于形成跨集群的策略理解和语言陈述，以免绑定到特定的下层实施机制。

随着相关机制的成熟，这些机制会按策略分别定义在下面。特定策略的实施方法不在这里定义。

PodSecurityPolicy

- [Privileged](#)
- [Baseline](#)
- [Restricted](#)

常见问题

为什么策略类型定义在 Privileged 和 Default 之间

这里定义的三种策略框架有一个明晰的线性递进关系，从最安全 (Restricted) 到最不安全，并且覆盖了很大范围的工作负载。特权要求超出 Baseline 策略者通常是特定于应用的需求，所以我们没有在这个范围内提供标准框架。这并不意味着在这样的情形下仍然只能使用 Privileged 框架，只是说处于这个范围的策略需要因地制宜地定义。

SIG Auth 可能会在将来考虑这个范围的框架，前提是有关于其他框架的需求。

安全策略与安全上下文的区别是什么？

[安全上下文](#)在运行时配置 Pod 和容器。安全上下文是在 Pod 清单中作为 Pod 和容器规约的一部分来定义的，所代表的是传递给容器运行时的参数。

安全策略则是控制面用来对安全上下文以及安全性上下文之外的参数实施某种设置的机制。在 2020 年 2 月，目前实施这些安全性策略的原生解决方案是 [Pod 安全性策略](#) - 一种对集群中 Pod 的安全性策略进行集中控制的机制。Kubernetes 生态系统中还在开发一些其他的替代方案，例如 [OPA Gatekeeper](#)。

我应该为我的 Windows Pod 实施哪种框架？

Kubernetes 中的 Windows 负载与标准的基于 Linux 的负载相比有一些局限性和区别。尤其是 Pod SecurityContext 字段 [对 Windows 不起作用](#)。因此，目前没有对应的标准 Pod 安全性框架。

沙箱（Sandboxed）Pod 怎么处理？

现在还没有 API 标准来控制 Pod 是否被视作沙箱化 Pod。沙箱 Pod 可以通过其是否使用沙箱化运行时（如 gVisor 或 Kata Container）来辨别，不过目前还没有关于什么是沙箱化运行时的标准定义。

沙箱化负载所需要的保护可能彼此各不相同。例如，当负载与下层内核直接隔离开来时，限制特权化操作的许可就不那么重要。这使得那些需要更多许可权限的负载仍能被有效隔离。

此外，沙箱化负载的保护高度依赖于沙箱化的实现方法。因此，现在还没有针对所有沙箱化负载的建议策略。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 12, 2020 at 9:03 PM PST: [\[zh\] Sync changes from English site \(3\) \(95ab5ac19\)](#)

云原生安全概述

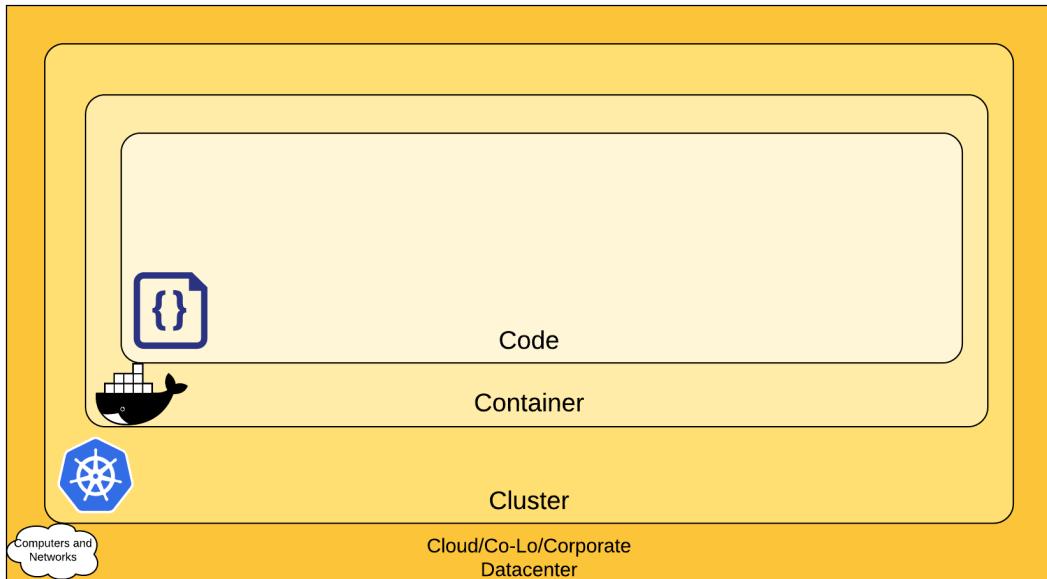
本概述定义了一个模型，用于在 Cloud Native 安全性上下文中考虑 Kubernetes 安全性。

警告：此容器安全模型只提供建议，而不是经过验证的信息安全策略。

云原生安全的 4 个 C

你可以分层去考虑安全性，云原生安全的 4 个 C 分别是云（Cloud）、集群（Cluster）、容器（Container）和代码（Code）。

说明：这种分层方法增强了[深度防护方法](#)在安全性方面的 防御能力，该方法被广泛认为是保护软件系统的最佳实践。



云原生安全的 4C

云原生安全模型的每一层都是基于下一个最外层，代码层受益于强大的基础安全层（云、集群、容器）。你无法通过在代码层解决安全问题来为基础层中糟糕的安全标准提供保护。

云

在许多方面，云（或者位于同一位置的服务器，或者是公司数据中心）是 Kubernetes 集群中的[可信计算基](#)。如果云层容易受到攻击（或者被配置成了易受攻击的方式），就不能保证在此基础之上构建的组件是安全的。每个云提供商都会提出安全建议，以在其环境中安全地运行工作负载。

云提供商安全性

如果您是在您自己的硬件或者其他不通的云提供商上运行 Kubernetes 集群，请查阅相关文档来获取最好的安全实践。

下面是一些比较流行的云提供商的安全性文档链接：

IaaS 提供商	链接
Alibaba Cloud	https://www.alibabacloud.com/trust-center

IaaS 提供商	链接
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

基础设施安全

关于在 Kubernetes 集群中保护你的基础设施的建议：

Kubernetes 基础架构关注领域	建议
通过网络访问 API 服务（控制平面）	所有对 Kubernetes 控制平面的访问不允许在 Internet 上公开，同时应由网络访问控制列表控制，该列表包含管理集群所需的 IP 地址集。
通过网络访问 Node（节点）	节点应配置为 仅能 从控制平面上通过指定端口来接受（通过网络访问控制列表）连接，以及接受 NodePort 和 LoadBalancer 类型的 Kubernetes 服务连接。如果可能的话，这些节点不应完全暴露在公共互联网上。
Kubernetes 访问云提供商的 API	每个云提供商都需要向 Kubernetes 控制平面和节点授予不同的权限集。为集群提供云提供商访问权限时，最好遵循对需要管理的资源的 最小特权原则 。 Kops 文档 提供有关 IAM 策略和角色的信息。
访问 etcd	对 etcd（Kubernetes 的数据存储）的访问应仅限于控制平面。根据配置情况，你应该尝试通过 TLS 来使用 etcd。更多信息可以在 etcd 文档 中找到。
etcd 加密	在所有可能的情况下，最好对所有驱动器进行静态数据加密，但是由于 etcd 拥有整个集群的状态（包括机密信息），因此其磁盘更应该进行静态数据加密。

集群

保护 Kubernetes 有两个方面需要注意：

- 保护可配置的集群组件
- 保护在集群中运行的应用程序

集群组件

如果想要保护集群免受意外或恶意的访问，采取良好的信息管理实践，请阅读并遵循有关[保护集群](#)的建议。

集群中的组件（您的应用）

根据您的应用程序的受攻击面，您可能需要关注安全性的特定面，比如：如果您正在运行中的一个服务（A 服务）在其他资源链中很重要，并且所运行的另一工作负载（服务 B）容易受到资源枯竭的攻击，则如果你不限制服务 B 的资源的话，损害服务 A 的风险就会很高。下表列出了安全性关注的领域和建议，用以保护 Kubernetes 中运行的工作负载：

工作负载安全性关注领域	建议
RBAC 授权(访问 Kubernetes API)	https://kubernetes.io/zh/docs/reference/access-authn-authz/rbac/
认证方式	https://kubernetes.io/zh/docs/reference/access-authn-authz/controlling-access/
应用程序 Secret 管理 (并在 etcd 中对其进行静态数据加密)	https://kubernetes.io/zh/docs/concepts/configuration/secret/ https://kubernetes.io/zh/docs/tasks/administer-cluster/encrypt-data/
Pod 安全策略	https://kubernetes.io/zh/docs/concepts/policy/pod-security-policy/
服务质量（和集群资源管理）	https://kubernetes.io/zh/docs/tasks/configure-pod-container/quality-service-pod/
网络策略	https://kubernetes.io/zh/docs/concepts/services-networking/network-policies/
Kubernetes Ingress 的 TLS 支持	https://kubernetes.io/zh/docs/concepts/services-networking/ingress/#tls

容器

容器安全性不在本指南的探讨范围内。下面是一些探索此主题的建议和连接：

容器关注领域	建议
容器漏洞扫描和操作系统依赖安全性	作为镜像构建的一部分，您应该扫描您的容器里的已知漏洞。
镜像签名和执行	对容器镜像进行签名，以维护对容器内容的信任。
禁止特权用户	构建容器时，请查阅文档以了解如何在具有最低操作系统特权级别的容器内部创建用户，以实现容器的目标。

代码

应用程序代码是您最能够控制的主要攻击面之一，虽然保护应用程序代码不在 Kubernetes 安全主题范围内，但以下是保护应用程序代码的建议：

代码安全性

代码关注领域	建议
仅通过 TLS 访问	如果您的代码需要通过 TCP 通信，请提前与客户端执行 TLS 握手。除少数情况外，请加密传输中的所有内容。更进一步，加密服务之间的网络流量是一个好主意。这可以通过被称为相互 LTS 或 mTLS 的过程来完成，该过程对两个证书持有服务之间的通信执行双向验证。
限制通信端口范围	此建议可能有点不言自明，但是在任何可能的情况下，你都只应公开服务上对于通信或度量收集绝对必要的端口。
第三方依赖性安全	最好定期扫描应用程序的第三方库以了解已知的安全漏洞。每种编程语言都有一个自动执行此检查的工具。
静态代码分析	大多数语言都提供了一种方法，来分析代码段中是否存在潜在的不安全的编码实践。只要有可能，你都应该使用自动工具执行检查，该工具可以扫描代码库以查找常见的安全错误，一些工具可以在以下连接中找到： https://owasp.org/www-community/Source_Code_Analysis_Tools
动态探测攻击	您可以对服务运行一些自动化工具，来尝试一些众所周知的服务攻击。这些攻击包括 SQL 注入、CSRF 和 XSS。 OWASP Zed Attack 代理工具是最受欢迎的动态分析工具之一。

接下来

学习了解相关的 Kubernetes 安全主题：

- [Pod 安全标准](#)
- [Pod 的网络策略](#)
- [控制对 Kubernetes API 的访问](#)
- [保护您的集群](#)
- 为控制面[加密通信中的数据](#)
- [加密静止状态的数据](#)
- [Kubernetes 中的 Secret](#)

反馈

此页是否对您有帮助？

是 否

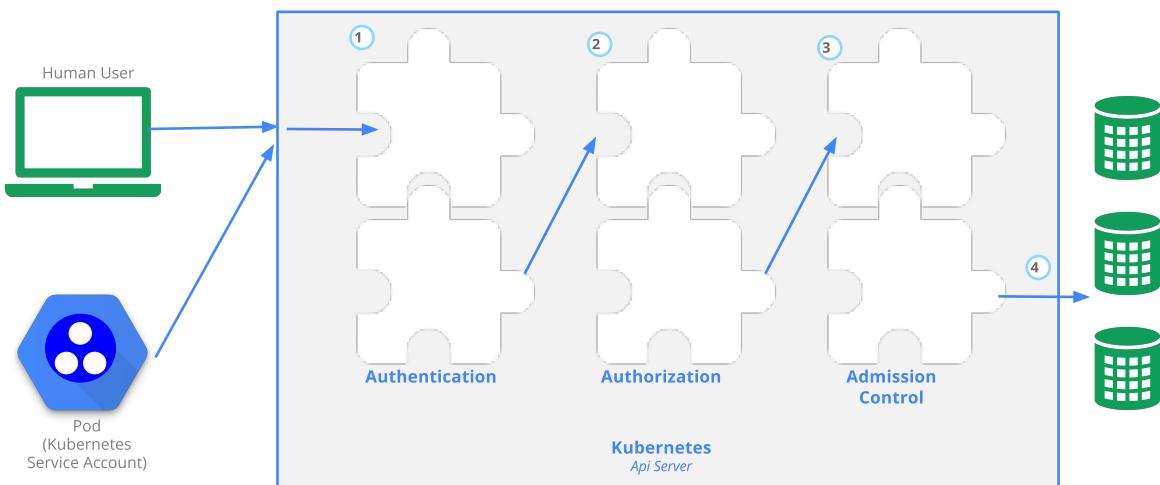
感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 12, 2020 at 9:03 PM PST: [\[zh\] Sync changes from English site \(3\) \(95ab5ac19\)](#)

Kubernetes API 访问控制

本页面概述了对 Kubernetes API 的访问控制。

用户使用 kubectl、客户端库或构造 REST 请求来访问 [Kubernetes API](#)。人类用户和 [Kubernetes 服务账户](#)都可以被鉴权访问 API。当请求到达 API 时，它会经历多个阶段，如下图所示：



传输安全

在典型的 Kubernetes 集群中，API 服务器在 443 端口上提供服务，受 TLS 保护。API 服务器出示证书。该证书可以使用私有证书颁发机构（CA）签名，也可以基于链接到公认的 CA 的公钥基础架构签名。

如果你的集群使用私有证书颁发机构，你需要在客户端的 `~/.kube/config` 文件中提供该 CA 证书的副本，以便你可以信任该连接并确认该连接没有被拦截。

你的客户端可以在此阶段出示 TLS 客户端证书。

认证

如上图步骤 1 所示，建立 TLS 后，HTTP 请求将进入认证（Authentication）步骤。集群创建脚本或者集群管理员配置 API 服务器，使之运行一个或多个身份认证组件。身份认证组件在[认证节](#)中有更详细的描述。

认证步骤的输入整个 HTTP 请求；但是，通常组件只检查头部或/和客户端证书。

认证模块包含客户端证书、密码、普通令牌、引导令牌和 JSON Web 令牌（JWT，用于服务账户）。

可以指定多个认证模块，在这种情况下，服务器依次尝试每个验证模块，直到其中一个成功。

如果请求认证不通过，服务器将以 HTTP 状态码 401 拒绝该请求。反之，该用户被认证为特定的 `username`，并且该用户名可用于后续步骤以在其决策中使用。部分验证器还提供用户的组成员身份，其他则不提供。

鉴权

如上图的步骤 2 所示，将请求验证为来自特定的用户后，请求必须被鉴权。

请求必须包含请求者的用户名、请求的行为以及受该操作影响的对象。如果现有策略声明用户有权完成请求的操作，那么该请求被鉴权通过。

例如，如果 Bob 有以下策略，那么他只能在 `projectCaribou` 名称空间中读取 Pod。

```
{  
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",  
  "kind": "Policy",  
  "spec": {  
    "user": "bob",  
    "namespace": "projectCaribou",  
    "resource": "pods",  
    "readonly": true  
  }  
}
```

如果 Bob 执行以下请求，那么请求会被鉴权，因为允许他读取 `projectCaribou` 名称空间中的对象。

```
{  
  "apiVersion": "authorization.k8s.io/v1beta1",  
  "kind": "SubjectAccessReview",  
  "spec": {  
    "resourceAttributes": {  
      "namespace": "projectCaribou",  
      "verb": "get",  
      "group": "unicorn.example.org",  
      "resource": "pods"  
    }  
  }  
}
```

如果 Bob 在 `projectCaribou` 名字空间中请求写（`create` 或 `update`）对象，其鉴权请求将被拒绝。如果 Bob 在诸如 `projectFish` 这类其它名字空间中请求读取（`get`）对象，其鉴权也会被拒绝。

Kubernetes 鉴权要求使用公共 REST 属性与现有的组织范围或云提供商范围的访问控制系统进行交互。 使用 REST 格式很重要，因为这些控制系统可能会与 Kubernetes API 之外的 API 交互。

Kubernetes 支持多种鉴权模块，例如 ABAC 模式、RBAC 模式和 Webhook 模式等。 管理员创建集群时，他们配置应在 API 服务器中使用的鉴权模块。 如果配置了多个鉴权模块，则 Kubernetes 会检查每个模块，任意一个模块鉴权该请求，请求即可继续；如果所有模块拒绝了该请求，请求将被拒绝（HTTP 状态码 403）。

要了解更多有关 Kubernetes 鉴权的更多信息，包括有关使用支持鉴权模块创建策略的详细信息，请参阅[鉴权](#)。

准入控制

准入控制模块是可以修改或拒绝请求的软件模块。 除鉴权模块可用的属性外，准入控制模块还可以访问正在创建或修改的对象的内容。

准入控制器对创建、修改、删除或（通过代理）连接对象的请求进行操作。 准入控制器不会对仅读取对象的请求起作用。 有多个准入控制器被配置时，服务器将依次调用它们。

这一操作如上图的步骤 3 所示。

与身份认证和鉴权模块不同，如果任何准入控制器模块拒绝某请求，则该请求将立即被拒绝。

除了拒绝对象之外，准入控制器还可以为字段设置复杂的默认值。

可用的准入控制模块在[准入控制器](#)中进行了描述。

请求通过所有准入控制器后，将使用检验例程检查对应的 API 对象，然后将其写入对象存储（如步骤 4 所示）。

API 服务器端口和 IP

前面的讨论适用于发送到 API 服务器的安全端口的请求（典型情况）。 API 服务器实际上可以在 2 个端口上提供服务：

默认情况下，Kubernetes API 服务器在 2 个端口上提供 HTTP 服务：

1. localhost 端口：

- 用于测试和引导，以及主控节点上的其他组件（调度器，控制器管理器）与 API 通信
- 没有 TLS
- 默认为端口 8080，使用 --insecure-port 进行更改
- 默认 IP 为 localhost，使用 --insecure-bind-address 进行更改
- 请求 绕过 身份认证和鉴权模块
- 由准入控制模块处理的请求
- 受需要访问主机的保护

"安全端口"：

2.

- 尽可能使用
- 使用 TLS。用 `--tls-cert-file` 设置证书，用 `--tls-private-key-file` 设置密钥
- 默认端口 6443，使用 `--secure-port` 更改
- 默认 IP 是第一个非本地网络接口，使用 `--bind-address` 更改
- 请求须经身份认证和鉴权组件处理
- 请求须经准入控制模块处理
- 身份认证和鉴权模块运行

接下来

阅读更多有关身份认证、鉴权和 API 访问控制的文档：

- [认证](#)
 - [使用 Bootstrap 令牌进行身份认证](#)
- [准入控制器](#)
 - [动态准入控制](#)
- [鉴权](#)
 - [基于角色的访问控制](#)
 - [基于属性的访问控制](#)
 - [节点鉴权](#)
 - [Webhook 鉴权](#)
- [证书签名请求](#)
 - 包括 [CSR 认证](#) 和 [证书签名](#)
- [服务账户](#)
 - [开发者指导](#)
 - [管理](#)

你可以了解

- Pod 如何使用 [Secrets](#) 获取 API 凭证.

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 25, 2020 at 3:07 AM PST: [error sentence. \(2e10cc75a\)](#)

策略

可配置的、可应用到一组资源的策略。

[资源配置](#)

[限制范围](#)

[Pod 安全策略](#)

[进程 ID 约束与预留](#)

资源配置

当多个用户或团队共享具有固定节点数目的集群时，人们会担心有人使用超过其基于公平原则所分配到的资源量。

资源配置是帮助管理员解决这一问题的工具。

资源配置，通过 ResourceQuota 对象来定义，对每个命名空间的资源消耗总量提供限制。它可以限制命名空间中某种类型的对象的总数目上限，也可以限制命令空间中的 Pod 可以使用的计算资源的总上限。

资源配置的工作方式如下：

- 不同的团队可以在不同的命名空间下工作，目前这是非约束性的，在未来的版本中可能会通过 ACL (Access Control List 访问控制列表) 来实现强制性约束。
- 集群管理员可以为每个命名空间创建一个或多个 ResourceQuota 对象。
- 当用户在命名空间下创建资源（如 Pod、Service 等）时，Kubernetes 的配额系统会 跟踪集群的资源使用情况，以确保使用的资源用量不超过 ResourceQuota 中定义的硬性资源限额。
- 如果资源创建或者更新请求违反了配额约束，那么该请求会报错（HTTP 403 FORBIDDEN），并在消息中给出有可能违反的约束。
- 如果命名空间下的计算资源（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 LimitRanger 准入控制器来为没有设置计算资源需求的 Pod 设置默认值。

若想避免这类问题，请参考 [演练示例](#)。

ResourceQuota 对象的名称必须是合法的 [DNS 子域名](#)。

下面是使用命名空间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 10 GiB 内存和 4 核的 CPU 资源，并且预留 2 GiB 内存和 2 核的 CPU 资源供将来分配。
- 限制 "testing" 命名空间使用 1 核 CPU 资源和 1GiB 内存。允许 "production" 命名空间使用任意数量。

在集群容量小于各命名空间配额总和的情况下，可能存在资源竞争。资源竞争时，Kubernetes 系统会遵循先到先得的原则。

不管是资源竞争还是配额的修改，都不会影响已经创建的资源使用对象。

启用资源配置

资源配置的支持在很多 Kubernetes 版本中是默认开启的。当 API 服务器的 `--enable-admission-plugins=` 参数中包含 `ResourceQuota` 时，资源配置会被启用。

当命名空间中存在一个 `ResourceQuota` 对象时，对于该命名空间而言，资源配置就是开启的。

计算资源配置

用户可以对给定命名空间下的可被请求的 [计算资源](#) 总量进行限制。

配额机制所支持的资源类型：

资源名称	描述
<code>limits.cpu</code>	所有非终止状态的 Pod，其 CPU 限额总量不能超过该值。
<code>limits.memory</code>	所有非终止状态的 Pod，其内存限额总量不能超过该值。
<code>requests.cpu</code>	所有非终止状态的 Pod，其 CPU 需求总量不能超过该值。
<code>requests.memory</code>	所有非终止状态的 Pod，其内存需求总量不能超过该值。
<code>hugepages-<size></code>	对于所有非终止状态的 Pod，针对指定尺寸的巨页请求总数不能超过此值。
<code>cpu</code>	与 <code>requests.cpu</code> 相同。
<code>memory</code>	与 <code>requests.memory</code> 相同。

扩展资源的资源配置

除上述资源外，在 Kubernetes 1.10 版本中，还添加了对 [扩展资源](#) 的支持。

由于扩展资源不可超量分配，因此没有必要在配额中为同一扩展资源同时指定 `requests` 和 `limits`。对于扩展资源而言，目前仅允许使用前缀为 `requests.` 的配额项。

以 GPU 拓展资源为例，如果资源名称为 `nvidia.com/gpu`，并且要将命名空间中请求的 GPU 资源总数限制为 4，则可以如下定义配额：

- `requests.nvidia.com/gpu: 4`

有关更多详细信息，请参阅[查看和设置配额](#)。

存储资源配置

用户可以对给定命名空间下的[存储资源](#) 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 PVC，存储资源的需求总量不能超过该值。
persistentvolumeclaims	在该命名空间中所允许的 PVC 总量。
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	在所有与 <storage-class-name> 相关的持久卷申领中，存储请求的总和不能超过该值。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	在与 storage-class-name 相关的所有持久卷申领中，命名空间中可以存在的持久卷申领总数。

例如，如果一个操作人员针对 gold 存储类型与 bronze 存储类型设置配额，操作人员可以定义如下配额：

- gold.storageclass.storage.k8s.io/requests.storage: 500Gi
- bronze.storageclass.storage.k8s.io/requests.storage: 100Gi

在 Kubernetes 1.8 版本中，本地临时存储的配额支持已经是 Alpha 功能：

资源名称	描述
requests.ephemeral-storage	在命名空间的所有 Pod 中，本地临时存储请求的总和不能超过此值。
limits.ephemeral-storage	在命名空间的所有 Pod 中，本地临时存储限制值的总和不能超过此值。
ephemeral-storage	与 requests.ephemeral-storage 相同。

对象数量配额

你可以使用以下语法对所有标准的、命名空间域的资源类型进行配额设置：

- count/<resource>.<group>：用于非核心（ core ）组的资源
- count/<resource>：用于核心组的资源

这是用户可能希望利用对象计数配额来管理的一组资源示例。

- count/persistentvolumeclaims
- count/services
- count/secrets
- count/configmaps
- count/replicationcontrollers
- count/deployments.apps
- count/replicasets.apps
- count/statefulsets.apps
- count/jobs.batch

- count/cronjobs.batch

相同语法也可用于自定义资源。例如，要对 example.com API 组中的自定义资源 widgets 设置配额，请使用 count/widgets.example.com。

当使用 count/* 资源配额时，如果对象存在于服务器存储中，则会根据配额管理资源。这些类型的配额有助于防止存储资源耗尽。例如，用户可能想根据服务器的存储能力来对服务器中 Secret 的数量进行配额限制。集群中存在过多的 Secret 实际上会导致服务器和控制器无法启动。用户可以选择对 Job 进行配额管理，以防止配置不当的 CronJob 在某命名空间中创建太多 Job 而导致集群拒绝服务。

对有限的一组资源上实施一般性的对象数量配额也是可能的。此外，还可以进一步按资源的类型设置其配额。

支持以下类型：

资源名称	描述
configmaps	在该命名空间中允许存在的 ConfigMap 总数上限。
persistentvolumeclaims	在该命名空间中允许存在的 PVC 的总数上限。
pods	在该命名空间中允许存在的非终止状态的 Pod 总数上限。Pod 终止状态等价于 Pod 的 .status.phase in (Failed, Succeeded) 为真。
replicationcontrollers	在该命名空间中允许存在的 ReplicationController 总数上限。
resourcequotas	在该命名空间中允许存在的 ResourceQuota 总数上限。
services	在该命名空间中允许存在的 Service 总数上限。
services.loadbalancers	在该命名空间中允许存在的 LoadBalancer 类型的 Service 总数上限。
services.nodeports	在该命名空间中允许存在的 NodePort 类型的 Service 总数上限。
secrets	在该命名空间中允许存在的 Secret 总数上限。

例如，pods 配额统计某个命名空间中所创建的、非终止状态的 Pod 个数并确保其不超过某上限值。用户可能希望在某命名空间中设置 pods 配额，以避免有用户创建很多小的 Pod，从而耗尽集群所能提供的 Pod IP 地址。

配额作用域

每个配额都有一组相关的 scope（作用域），配额只会对作用域内的资源生效。配额机制仅统计所列举的作用域的交集中的资源用量。

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。

作用域	描述
Terminating	匹配所有 spec.activeDeadlineSeconds 不小于 0 的 Pod。
NotTerminating	匹配所有 spec.activeDeadlineSeconds 是 nil 的 Pod。

作用域	描述
BestEffort	匹配所有 Qos 是 BestEffort 的 Pod。
NotBestEffort	匹配所有 Qos 不是 BestEffort 的 Pod。
PriorityClass	匹配所有引用了所指定的 优先级类 的 Pods。

BestEffort 作用域限制配额跟踪以下资源：

- pods

Terminating、NotTerminating、NotBestEffort 和 PriorityClass 这些作用域限制配额跟踪以下资源：

- pods
- cpu
- memory
- requests.cpu
- requests.memory
- limits.cpu
- limits.memory

需要注意的是，你不能在同一个配额对象中同时设置 Terminating 和 NotTerminating 作用域，你也不可以在同一个配额中同时设置 BestEffort 和 NotBestEffort 作用域。

scopeSelector 支持在 operator 字段中使用以下值：

- In
- NotIn
- Exists
- DoesNotExist

定义 scopeSelector 时，如果使用以下值之一作为 scopeName 的值，则对应的 operator 只能是 Exists。

- Terminating
- NotTerminating
- BestEffort
- NotBestEffort

如果 operator 是 In 或 NotIn 之一，则 values 字段必须至少包含一个值。例如：

```
scopeSelector:
  matchExpressions:
    - scopeName: PriorityClass
      operator: In
      values:
        - middle
```

如果 operator 为 Exists 或 DoesNotExist，则不可以设置 values 字段。

基于优先级类 (PriorityClass) 来设置资源配置

FEATURE STATE: Kubernetes v1.17 [stable]

Pod 可以创建为特定的[优先级](#)。通过使用配额规约中的 scopeSelector 字段，用户可以根据 Pod 的优先级控制其系统资源消耗。

仅当配额规范中的 scopeSelector 字段选择到某 Pod 时，配额机制才会匹配和计量 Pod 的资源消耗。

如果配额对象通过 scopeSelector 字段设置其作用域为优先级类，则配额对象只能 跟踪以下资源：

- pods
- cpu
- memory
- ephemeral-storage
- limits.cpu
- limits.memory
- limits.ephemeral-storage
- requests.cpu
- requests.memory
- requests.ephemeral-storage

本示例创建一个配额对象，并将其与具有特定优先级的 Pod 进行匹配。该示例的工作方式如下：

- 集群中的 Pod 可取三个优先级类之一，即 "low"、"medium"、"high"。
- 为每个优先级创建一个配额对象。

将以下 YAML 保存到文件 quota.yml 中。

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: pods-high
  spec:
    hard:
      cpu: "1000"
      memory: 200Gi
      pods: "10"
    scopeSelector:
      matchExpressions:
        - operator : In
          scopeName: PriorityClass
```

```
values: ["high"]
- apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-medium
spec:
  hard:
    cpu: "10"
    memory: 20Gi
    pods: "10"
  scopeSelector:
    matchExpressions:
      - operator : In
      scopeName: PriorityClass
      values: ["medium"]
- apiVersion: v1
kind: ResourceQuota
metadata:
  name: pods-low
spec:
  hard:
    cpu: "5"
    memory: 10Gi
    pods: "10"
  scopeSelector:
    matchExpressions:
      - operator : In
      scopeName: PriorityClass
      values: ["low"]
```

使用 kubectl create 命令运行以下操作。

```
kubectl create -f ./quota.yml
```

```
resourcequota/pods-high created
resourcequota/pods-medium created
resourcequota/pods-low created
```

使用 kubectl describe quota 操作验证配额的 Used 值为 0。

```
kubectl describe quota
```

```
Name:      pods-high
Namespace: default
Resource   Used Hard
-----
cpu        0    1k
```

```
memory 0 200Gi
pods   0 10
```

```
Name: pods-low
Namespace: default
Resource Used Hard
-----
cpu 0 5
memory 0 10Gi
pods 0 10
```

```
Name: pods-medium
Namespace: default
Resource Used Hard
-----
cpu 0 10
memory 0 20Gi
pods 0 10
```

创建优先级为 "high" 的 Pod。 将以下 YAML 保存到文件 high-priority-pod.yml 中。

```
apiVersion: v1
kind: Pod
metadata:
  name: high-priority
spec:
  containers:
  - name: high-priority
    image: ubuntu
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo hello; sleep 10;done"]
    resources:
      requests:
        memory: "10Gi"
        cpu: "500m"
      limits:
        memory: "10Gi"
        cpu: "500m"
  priorityClassName: high
```

使用 kubectl create 运行以下操作。

```
kubectl create -f ./high-priority-pod.yml
```

确认 "high" 优先级配额 pods-high 的 "Used" 统计信息已更改，并且其他两个配额未更改。

```
kubectl describe quota
```

```
Name:      pods-high
Namespace: default
Resource   Used Hard
-----
cpu        500m 1k
memory    10Gi 200Gi
pods       1    10
```

```
Name:      pods-low
Namespace: default
Resource   Used Hard
-----
cpu        0    5
memory    0    10Gi
pods       0    10
```

```
Name:      pods-medium
Namespace: default
Resource   Used Hard
-----
cpu        0    10
memory    0    20Gi
pods       0    10
```

请求与限制

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。配额可以针对二者之一进行设置。

如果配额中指定了 requests.cpu 或 requests.memory 的值，则它要求每个容器都显式给出对这些资源的请求。同理，如果配额中指定了 limits.cpu 或 limits.memory 的值，那么它要求每个容器都显式设定对应资源的限制。

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:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.nvidia.com/gpu: 4
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"
    pods: "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
requests.cpu       0   1
requests.memory    0   1Gi
requests.nvidia.com/gpu 0   4
```

```
kubectl describe quota object-counts --namespace=myspace
```

```
Name:          object-counts
Namespace:     myspace
Resource       Used   Hard
-----
configmaps    0     10
persistentvolumeclaims 0     4
pods          0     4
replicationcontrollers 0     20
secrets        1     10
services       0     10
services.loadbalancers 0     2
```

kubectl 还使用语法 `count/<resource>.<group>` 支持所有标准的、命名空间域的资源的对象计数配额：

```
kubectl create namespace myspace
```

```
kubectl create quota test --hard=count/deployments.apps=2,count/
replicasets.apps=4,count/pods=3,count/secrets=4 --namespace=myspace
```

```
kubectl create deployment nginx --image=nginx --namespace=myspace --
replicas=2
```

```
kubectl describe quota --namespace=myspace
```

```
Name:          test
Namespace:     myspace
Resource       Used   Hard
-----
count/deployments.apps   1   2
count/pods              2   3
count(replicasets.apps)  1   4
count/secrets            1   4
```

配额和集群容量

ResourceQuota 与集群资源总量是完全独立的。它们通过绝对的单位来配置。所以，为集群添加节点时，资源配置不会自动赋予每个命名空间消耗更多资源的能力。

有时可能需要资源配额支持更复杂的策略，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资源意外耗尽。
- 探测某个命名空间的需求，添加物理节点并扩大资源配额值。

这些策略可以通过将资源配额作为一个组成模块、手动编写一个控制器来监控资源使用情况，并结合其他信号调整命名空间上的硬性资源配额来实现。

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：来自不同命名空间的 Pod 可能在同一节点上运行。

默认情况下限制特定优先级的资源消耗

有时候可能希望当且仅当某名字空间中存在匹配的配额对象时，才可以创建特定优先级（例如 "cluster-services"）的 Pod。

通过这种机制，操作人员能够将限制某些高优先级类仅出现在有限数量的命名空间中，而并非每个命名空间默认情况下都能够使用这些优先级类。

要实现此目的，应设置 kube-apiserver 的标志 --admission-control-config-file 指向如下配置文件：

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: "ResourceQuota"
  configuration:
    apiVersion: apiserver.config.k8s.io/v1
    kind: ResourceQuotaConfiguration
    limitedResources:
    - resource: pods
      matchScopes:
      - scopeName: PriorityClass
        operator: In
        values: ["cluster-services"]
```

现在，仅当命名空间中存在匹配的 scopeSelector 的配额对象时，才允许使用 "cluster-services" Pod。

示例：

```
scopeSelector:
matchExpressions:
- scopeName: PriorityClass
  operator: In
  values: ["cluster-services"]
```

接下来

- 查看[资源配置设计文档](#)
- 查看[如何使用资源配置的详细示例](#)。
- 阅读[优先级类配额支持的设计文档](#)。 了解更多信息。
- 参阅 [LimitedResources](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

限制范围

默认情况下，Kubernetes 集群上的容器运行使用的[计算资源](#)没有限制。 使用资源配置，集群管理员可以以[名字空间](#)为单位，限制其资源的使用与创建。 在命名空间中，一个 Pod 或 Container 最多能够使用命名空间的资源配置所定义的 CPU 和内存用量。 有人担心，一个 Pod 或 Container 会垄断所有可用的资源。 LimitRange 是在命名空间内限制资源分配（给多个 Pod 或 Container）的策略对象。

一个 *LimitRange*（限制范围）对象提供的限制能够做到：

- 在一个命名空间中实施对每个 Pod 或 Container 最小和最大的资源使用量的限制。
- 在一个命名空间中实施对每个 PersistentVolumeClaim 能申请的最小和最大的存储空间大小的限制。
- 在一个命名空间中实施对一种资源的申请值和限制值的比值的控制。
- 设置一个命名空间中对计算资源的默认申请/限制值，并且自动的在运行时注入到多个 Container 中。

启用 LimitRange

对 LimitRange 的支持自 Kubernetes 1.10 版本默认启用。

LimitRange 支持在很多 Kubernetes 发行版本中也是默认启用的。

LimitRange 的名称必须是合法的 [DNS 子域名](#)。

限制范围总览

- 管理员在一个命名空间内创建一个 LimitRange 对象。

- 用户在命名空间内创建 Pod , Container 和 PersistentVolumeClaim 等资源。
- LimitRange 准入控制器对所有没有设置计算资源需求的 Pod 和 Container 设置默认值与限制值， 并跟踪其使用量以保证没有超出命名空间中存在的任意 LimitRange 对象中的最小、最大资源使用量以及使用量比值。
- 若创建或更新资源 (Pod、Container、PersistentVolumeClaim) 违反了 LimitRange 的约束， 向 API 服务器的请求会失败，并返回 HTTP 状态码 403 FORBIDDEN 与描述哪一项约束被违反的消息。
- 若命名空间中的 LimitRange 启用了对 cpu 和 memory 的限制， 用户必须指定这些值的需求使用量与限制使用量。否则，系统将会拒绝创建 Pod。
- LimitRange 的验证仅在 Pod 准入阶段进行，不对正在运行的 Pod 进行验证。

能够使用限制范围创建的策略示例有：

- 在一个有两个节点，8 GiB 内存与16个核的集群中，限制一个命名空间的 Pod 申请 100m 单位，最大 500m 单位的 CPU，以及申请 200Mi，最大 600Mi 的内存。
- 为 spec 中没有 cpu 和内存需求值的 Container 定义默认 CPU 限制值与需求值 150m，内存默认需求值 300Mi。

在命名空间的总限制值小于 Pod 或 Container 的限制值的总和的情况下，可能会产生资源竞争。在这种情况下，将不会创建 Container 或 Pod。

竞争和对 LimitRange 的改变都不会影响任何已经创建了的资源。

接下来

参阅 [LimitRanger 设计文档](#) 获取更多信息。

关于使用限值的例子，可参看

- [如何配置每个命名空间最小和最大的 CPU 约束。](#)
- [如何配置每个命名空间最小和最大的内存约束。](#)
- [如何配置每个命名空间默认的 CPU 申请值和限制值。](#)
- [如何配置每个命名空间默认的内存申请值和限制值。](#)
- [如何配置每个命名空间最小和最大存储使用量。](#)
- [配置每个命名空间的配额的详细例子。](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 August 06, 2020 at 4:49 PM PST: [\[zh\] Fix links/translation in concepts section \(7\) \(ed4cefba\)](#)

Pod 安全策略

FEATURE STATE: Kubernetes v1.20 [beta]

Pod 安全策略使得对 Pod 创建和更新进行细粒度的权限控制成为可能。

什么是 Pod 安全策略？

Pod 安全策略 (Pod Security Policy) 是集群级别的资源，它能够控制 Pod 规约中与安全性相关的各个方面。[PodSecurityPolicy](#) 对象定义了一组 Pod 运行时必须遵循的条件及相关字段的默认值，只有 Pod 满足这些条件 才会被系统接受。Pod 安全策略允许管理员控制如下方面：

控制的角度	字段名称
运行特权容器	privileged
使用宿主名字空间	hostPID, hostIPC
使用宿主的网络和端口	hostNetwork, hostPorts
控制卷类型的使用	volumes
使用宿主文件系统	allowedHostPaths
允许使用特定的 FlexVolume 驱动	allowedFlexVolumes
分配拥有 Pod 卷的 FSGroup 账号	fsGroup
以只读方式访问根文件系统	readOnlyRootFilesystem
设置容器的用户和组 ID	runAsUser, runAsGroup, supplementalGroups
限制 root 账号特权级提升	allowPrivilegeEscalation, defaultAllowPrivilegeEscalation
Linux 权能字 (Capabilities)	defaultAddCapabilities, requiredDropCapabilities, allowedCapabilities
设置容器的 SELinux 上下文	seLinux
指定容器可以挂载的 proc 类型	allowedProcMountTypes
指定容器使用的 AppArmor 模版	annotations
指定容器使用的 seccomp 模版	annotations
指定容器使用的 sysctl 模版	forbiddenSysctls, allowedUnsafeSysctls

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。

启用 Pod 安全策略

Pod 安全策略实现为一种可选（但是建议启用）的 [准入控制器](#)。启用了准入控制器 即可强制实施 Pod 安全策略，不过如果没有授权认可策略之前即启用 准入控制器 **将导致集群中无法创建任何 Pod**。

由于 Pod 安全策略 API (`policy/v1beta1/podsecuritypolicy`) 是独立于准入控制器 来启用的，对于现有集群而言，建议在启用准入控制器之前先添加策略并对其授权。

授权策略

`PodSecurityPolicy` 资源被创建时，并不执行任何操作。为了使用该资源，需要对发出请求的用户或者目标 Pod 的 [服务账号](#) 授权，通过允许其对策略执行 `use` 动词允许其使用该策略。

大多数 Kubernetes Pod 不是由用户直接创建的。相反，这些 Pod 是由 [Deployment](#)、[ReplicaSet](#) 或者经由控制器管理器模版化的控制器创建。赋予控制器访问策略的权限意味着对应控制器所创建的 **所有 Pod 都可访问策略**。因此，对策略进行授权的优先方案是为 Pod 的服务账号授予访问权限（参见[示例](#)）。

通过 RBAC 授权

[RBAC](#) 是一种标准的 Kubernetes 鉴权模式，可以很容易地用来授权策略访问。

首先，某 Role 或 ClusterRole 需要获得使用 `use` 访问目标策略的权限。访问授权的规则看起来像这样：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <Role 名称>
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:   ['use']
  resourceNames:
    - <要授权的策略列表>
```

接下来将该 Role（或 ClusterRole）绑定到授权的用户：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <绑定名称>
roleRef:
  kind: ClusterRole
  name: <角色名称>
  apiGroup: rbac.authorization.k8s.io
subjects:
# 授权特定的服务账号
- kind: ServiceAccount
  name: <要授权的服务账号名称>
  namespace: <authorized pod namespace>
# 授权特定的用户（不建议这样操作）
- kind: User
  apiGroup: rbac.authorization.k8s.io
  name: <要授权的用户名>
```

如果使用的是 RoleBinding（而不是 ClusterRoleBinding），授权仅限于与该 RoleBinding 处于同一名字空间中的 Pods。可以考虑将这种授权模式和系统组结合，对名字空间中的所有 Pod 授予访问权限。

```
# 授权该某名字空间中所有服务账号
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:serviceaccounts
# 或者与之等价，授权给某名字空间中所有被认证过的用户
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:authenticated
```

参阅[角色绑定示例](#) 查看 RBAC 绑定的更多实例。参阅[下文](#)，查看对 PodSecurityPolicy 进行授权的完整示例。

故障排查

- [控制器管理器组件](#) 必须运行在 [安全的 API 端口](#)，并且一定不能具有超级用户权限。否则其请求会绕过身份认证和鉴权模块控制，从而导致所有 PodSecurityPolicy 对象都被启用，用户亦能创建特权容器。关于配置控制器管理器鉴权相关的详细信息，可参阅 [控制器角色](#)。

策略顺序

除了限制 Pod 创建与更新，Pod 安全策略也可用来为所控制的很多字段 设置默认值。当存在多个策略对象时，Pod 安全策略控制器依据以下条件选择 策略：

1. 优先考虑中允许 Pod 不经修改地创建或更新的 PodSecurityPolicy，这些策略 不会更改 Pod 字段的默认值或者其他配置。这类非更改性质的 PodSecurityPolicy 对象之间的顺序无关紧要。
2. 如果必须要为 Pod 设置默认值或者其他配置，（按名称顺序）选择第一个允许 Pod 操作的 PodSecurityPolicy 对象。

说明：在更新操作期间（这时不允许更改 Pod 规约），仅使用非更改性质的 PodSecurityPolicy 来对 Pod 执行验证操作。

示例

本示例假定你已经有一个启动了 *PodSecurityPolicy* 准入控制器的集群并且 你拥有集群管理员特权。

配置

为运行此示例，配置一个名字空间和一个服务账号。我们将用这个服务账号来 模拟一个非管理员账号的用户。

```
kubectl create namespace psp-example  
kubectl create serviceaccount -n psp-example fake-user  
kubectl create rolebinding -n psp-example fake-editor --clusterrole=edit --  
serviceaccount=psp-example:fake-user
```

创建两个别名，以更清晰地展示我们所使用的用户账号，同时减少一些键盘输入：

```
alias kubectl-admin='kubectl -n psp-example'  
alias kubectl-user='kubectl --as=system:serviceaccount:psp-example:fake-user -  
n psp-example'
```

创建一个策略和一个 Pod

在一个文件中定义一个示例的 PodSecurityPolicy 对象。这里的策略只是用来禁止创建有特权要求的 Pods。PodSecurityPolicy 对象的名称必须是合法的 [DNS 子域名](#)。

[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
```

现在，作为一个非特权用户，尝试创建一个简单的 Pod：

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
Error from server (Forbidden): error when creating "STDIN": pods "pause" is
forbidden: unable to validate against any pod security policy: []
```

发生了什么？ 尽管 PodSecurityPolicy 被创建，Pod 的服务账号或者 fake-user 用户都没有使用该策略的权限。

```
kubectl-user auth can-i use podsecuritypolicy/example
```

```
no
```

创建角色绑定，赋予 fake-user 使用 use 访问示例策略的权限：

说明： 不建议使用这种方法！欲了解优先考虑的方法，请参见[下节](#)。

```
kubectl-admin create role psp:unprivileged \
  --verb=use \
  --resource=podsecuritypolicy \
  --resource-name=example
```

输出：

```
role "psp:unprivileged" created
```

```
kubectl-admin create rolebinding fake-user:psp:unprivileged \
--role=psp:unprivileged \
--serviceaccount=psp-example:fake-user
```

输出：

```
rolebinding "fake-user:psp:unprivileged" created
```

```
kubectl-user auth can-i use podsecuritypolicy/example
```

输出：

```
yes
```

现在重试创建 Pod：

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: pause
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
EOF
```

输出：

```
pod "pause" created
```

此次尝试不出所料地成功了！不过任何创建特权 Pod 的尝试还是会被拒绝：

```
kubectl-user create -f- <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: privileged
spec:
  containers:
    - name: pause
      image: k8s.gcr.io/pause
      securityContext:
        privileged: true
EOF
```

输出为：

```
Error from server (Forbidden): error when creating "STDIN": pods "privileged" is forbidden: unable to validate against any pod security policy:  
[spec.containers[0].securityContext.privileged: Invalid value: true: Privileged containers are not allowed]
```

继续此例之前先删除该 Pod :

```
kubectl-user delete pod pause
```

运行另一个 Pod

我们再试一次，稍微有些不同：

```
kubectl-user create deployment pause --image=k8s.gcr.io/pause
```

输出为：

```
deployment "pause" created
```

```
kubectl-user get pods
```

输出为：

```
No resources found.
```

```
kubectl-user get events | head -n 2
```

输出为：

LASTSEEN	FIRSTSEEN	COUNT	NAME	KIND
SUBOBJECT		TYPE	REASON	SOURCE
MESSAGE				
1m	2m	15	pause-7774d79b5	ReplicaSet
Warning	FailedCreate		replicaset-controller	Error creating: pods "pause-7774d79b5-" is forbidden: no providers available to validate pod request

发生了什么？ 我们已经为用户 fake-user 绑定了 psp:unprivileged 角色，为什么还会收到错误 Error creating: pods "pause-7774d79b5-" is forbidden: no providers available to validate pod request (创建错误：pods "pause-7774d79b5" 被禁止：没有可用来验证 pod 请求的驱动)？答案在于源文件 - replicaset-controller。fake-user 用户成功地创建了 Deployment，而后者也成功地创建了 ReplicaSet，不过当 ReplicaSet 创建 Pod 时，发现未被授权使用示例 PodSecurityPolicy 资源。

为了修复这一问题，将 psp:unprivileged 角色绑定到 Pod 的服务账号。在这里，因为我们没有给出服务账号名称，默认的服务账号是 default。

```
kubectl-admin create rolebinding default:psp:unprivileged \  
--role=psp:unprivileged \  
--serviceaccount=psp-example:default
```

输出为：

```
rolebinding "default:psp:unprivileged" created
```

现在如果你给 ReplicaSet 控制器一分钟的时间来重试，该控制器最终将能够成功地创建 Pod：

```
kubectl-user get pods --watch
```

输出类似于：

NAME	READY	STATUS	RESTARTS	AGE
pause-7774d79b5-qrgcb	0/1	Pending	0	1s
pause-7774d79b5-qrgcb	0/1	Pending	0	1s
pause-7774d79b5-qrgcb	0/1	ContainerCreating	0	1s
pause-7774d79b5-qrgcb	1/1	Running	0	2s

清理

删除名字空间即可清理大部分示例资源：

```
kubectl-admin delete ns psp-example
```

输出类似于：

```
namespace "psp-example" deleted
```

注意 PodSecurityPolicy 资源不是名字空间域的资源，必须单独清理：

```
kubectl-admin delete psp example
```

输出类似于：

```
podsecuritypolicy "example" deleted
```

示例策略

下面是一个你可以创建的约束性非常弱的策略，其效果等价于没有使用 Pod 安全 策略准入控制器：

[policy/privileged-psp.yaml](#)



```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: privileged
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: '*'
spec:
```

```
privileged: true
allowPrivilegeEscalation: true
allowedCapabilities:
- '*'
volumes:
- '*'
hostNetwork: true
hostPorts:
- min: 0
  max: 65535
hostIPC: true
hostPID: true
runAsUser:
  rule: 'RunAsAny'
seLinux:
  rule: 'RunAsAny'
supplementalGroups:
  rule: 'RunAsAny'
fsGroup:
  rule: 'RunAsAny'
```

下面是一个具有约束性的策略，要求用户以非特权账号运行，禁止可能的向 root 权限的升级，同时要求使用若干安全机制。

[policy/restricted-psp.yaml](#)



```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/
default, runtime/default'
    apparmor.security.beta.kubernetes.io/allowedProfileNames: 'runtime/default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName: 'runtime/default'
    apparmor.security.beta.kubernetes.io/defaultProfileName: 'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
  - ALL
  # Allow core volume types.
```

```
volumes:
  - 'configMap'
  - 'emptyDir'
  - 'projected'
  - 'secret'
  - 'downwardAPI'
  # Assume that persistentVolumes set up by the cluster admin are safe to use.
  - 'persistentVolumeClaim'
hostNetwork: false
hostIPC: false
hostPID: false
runAsUser:
  # Require the container to run without root privileges.
  rule: 'MustRunAsNonRoot'
seLinux:
  # This policy assumes the nodes are using AppArmor rather than SELinux.
  rule: 'RunAsAny'
supplementalGroups:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
fsGroup:
  rule: 'MustRunAs'
  ranges:
    # Forbid adding the root group.
    - min: 1
      max: 65535
readOnlyRootFilesystem: false
```

更多的示例可参考 [Pod 安全标准](#)。

策略参考

Privileged

Privileged - 决定是否 Pod 中的某容器可以启用特权模式。默认情况下，容器是不可以访问宿主上的任何设备的，不过一个“privileged（特权的）”容器则被授权访问宿主上所有设备。这种容器几乎享有宿主上运行的进程的所有访问权限。对于需要使用 Linux 权能字（如操控网络堆栈和访问设备）的容器而言是有用的。

宿主名字空间

HostPID - 控制 Pod 中容器是否可以共享宿主上的进程 ID 空间。注意，如果与 ptrace 相结合，这种授权可能被利用，导致向容器外的特权逃逸（默认情况下 ptrace 是被禁止的）。

HostIPC - 控制 Pod 容器是否可共享宿主上的 IPC 名字空间。

HostNetwork - 控制是否 Pod 可以使用节点的网络名字空间。此类授权将允许 Pod 访问本地回路（loopback）设备、在本地主机（localhost）上监听的服务、还可能用来监听同一节点上其他 Pod 的网络活动。

HostPorts - 提供可以在宿主网络名字空间中可使用的端口范围列表。该属性定义为一组 HostPortRange 对象的列表，每个对象中包含 min（含）与 max（含）值的设置。默认不允许访问宿主端口。

卷和文件系统

Volumes - 提供一组被允许的卷类型列表。可被允许的值对应于创建卷时可以设置的卷来源。卷类型的完整列表可参见 [卷类型](#)。此外，* 可以用来允许所有卷类型。

对于新的 Pod 安全策略设置而言，建议设置的卷类型的最小列表包含：

- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- secret
- projected

警告：PodSecurityPolicy 并不限制可以被 PersistentVolumeClaim 所引用的 PersistentVolume 对象的类型。此外 hostPath 类型的 PersistentVolume 不支持只读访问模式。应该仅赋予受信用户创建 PersistentVolume 对象的访问权限。

FSGroup - 控制应用到某些卷上的附加用户组。

- *MustRunAs* - 要求至少指定一个 range。使用范围中的最小值作为默认值。所有 range 值都会被用来执行验证。
- *MayRunAs* - 要求至少指定一个 range。允许不设置 FSGroups，且无默认值。如果 FSGroup 被设置，则所有 range 值都会被用来执行验证检查。
- *RunAsAny* - 不提供默认值。允许设置任意 fsGroup ID 值。

AllowedHostPaths - 设置一组宿主文件目录，这些目录项可以在 hostPath 卷中使用。列表为空意味着对所使用的宿主目录没有限制。此选项定义包含一个对象列表，表中对象包含 pathPrefix 字段，用来表示允许 hostPath 卷挂载以所指定前缀开头的路径。对象中还包含一个 readOnly 字段，用来表示对应的卷必须以只读方式挂载。例如：

allowedHostPaths:

```
# 下面的设置允许 "/foo"、"/foo/"、"/foo/bar" 等路径，但禁止  
# "/fool"、"/etc/foo" 这些路径。  
# "/foo/.." 总会被当作非法路径。  
- pathPrefix: "/foo"  
readOnly: true # 仅允许只读模式挂载
```

警告：

容器如果对宿主文件系统拥有不受限制的访问权限，就可以有很多种方式提升自己的特权，包括读取其他容器中的数据、滥用系统服务（如 kubelet）的凭据信息等。

由可写入的目录所构造的 hostPath 卷能够允许容器写入数据到宿主文件系统，并且在写入时避开 pathPrefix 所设置的目录限制。 readOnly: true 这一设置在 Kubernetes 1.11 版本之后可用。必须针对 allowedHostPaths 中的所有条目设置此属性才能有效地限制容器只能访问 pathPrefix 所指定的目录。

ReadOnlyRootFilesystem - 要求容器必须以只读方式挂载根文件系统来运行（即不允许存在可写入层）。

FlexVolume 驱动

此配置指定一个可以被 FlexVolume 卷使用的驱动程序的列表。空的列表或者 nil 值意味着对驱动没有任何限制。请确保[volumes](#) 字段包含了 flexVolume 卷类型，否则所有 FlexVolume 驱动都被禁止。

```
apiVersion: policy/v1beta1  
kind: PodSecurityPolicy  
metadata:  
  name: allow-flex-volumes  
spec:  
  # spec d的其他字段  
  volumes:  
    - flexVolume  
  allowedFlexVolumes:  
    - driver: example/lvm  
    - driver: example/cifs
```

用户和组

RunAsUser - 控制使用哪个用户 ID 来运行容器。

- *MustRunAs* - 必须配置一个 range。使用该范围内的第一个值作为默认值。所有 range 值都被用于验证检查。
- *MustRunAsNonRoot* - 要求提交的 Pod 具有非零 runAsUser 值，或在镜像中（使用 UID 数值）定义了 USER 环境变量。如果 Pod 既没有设置 runAsNonRoo

- t，也没有设置 runAsUser，则该 Pod 会被修改以设置 runAsNonRoot=true，从而要求容器通过 USER 指令给出非零的数值形式的用户 ID。此配置没有默认值。采用此配置时，强烈建议设置 allowPrivilegeEscalation=false。
- *RunAsAny* - 没有提供默认值。允许指定任何 runAsUser 配置。

RunAsGroup - 控制运行容器时使用的主用户组 ID。

- *MustRunAs* - 要求至少指定一个 range 值。第一个 range 中的最小值作为默认值。所有 range 值都被用来执行验证检查。
- *MayRunAs* - 不要求设置 RunAsGroup。不过，如果指定了 RunAsGroup 被设置，所设置值必须处于所定义的范围内。
- *RunAsAny* - 未指定默认值。允许 runAsGroup 设置任何值。

SupplementalGroups - 控制容器可以添加的组 ID。

- *MustRunAs* - 要求至少指定一个 range 值。第一个 range 中的最小值用作默认值。所有 range 值都被用来执行验证检查。
- *MayRunAs* - 要求至少指定一个 range 值。允许不指定 supplementalGroups 且不设置默认值。如果 supplementalGroups 被设置，则所有 range 值都被用来执行验证检查。
- *RunAsAny* - 未指定默认值。允许为 supplementalGroups 设置任何值。

特权提升

这一组选项控制容器的allowPrivilegeEscalation 属性。该属性直接决定是否为 容器进程设置 [no_new_privs](#) 参数。此参数会禁止 setuid 属性的可执行文件更改有效用户 ID (EUID)，并且 禁止启用额外权能的文件。例如，no_new_privs 会禁止使用 ping 工具。如果想有效地实施 MustRunAsNonRoot 控制，需要配置这一选项。

AllowPrivilegeEscalation - 决定是否用户可以将容器的安全上下文设置为 allowPrivilegeEscalation=true。默认设置下，这样做是允许的，目的是避免 造成现有的 setuid 应用无法运行。将此选项设置为 false 可以确保容器的所有 子进程都无法获得比父进程更多的特权。

DefaultAllowPrivilegeEscalation - 为 allowPrivilegeEscalation 选项设置 默认值。不设置此选项时的默认行为是允许特权提升，以便运行 setuid 程序。如果不希望运行 setuid 程序，可以使用此字段将选项的默认值设置为禁止，同时 仍然允许 Pod 显式地请求 allowPrivilegeEscalation。

权能字

Linux 权能字 (Capabilities) 将传统上与超级用户相关联的特权作了细粒度的分解。其中某些权能字可以用来提升特权，打破容器边界，可以通过 PodSecurityPolicy 来限制。关于 Linux 权能字的更多细节，可参阅 [capabilities\(7\)](#)。

下列字段都可以配置为权能字的列表。表中的每一项都是 ALL_CAPS 中的一个权能字 名称，只是需要去掉 CAP_ 前缀。

AllowedCapabilities - 给出可以被添加到容器的权能字列表。默认的权能字集合是被隐式允许的那些。空集合意味着只能使用默认权能字集合，不允许添加额外的权能字。* 可以用来设置允许所有权能字。

RequiredDropCapabilities - 必须从容器中去除的权能字。所给的权能字会从默认权能字集合中去除，并且一定不可以添加。RequiredDropCapabilities 中列举的权能字不能出现在 AllowedCapabilities 或 DefaultAddCapabilities 所给的列表中。

DefaultAddCapabilities - 默认添加到容器的权能字集合。这一集合是作为容器运行时所设值的补充。关于使用 Docker 容器运行引擎时默认的权能字列表，可参阅 [Docker 文档](#)。

SELinux

- *MustRunAs* - 要求必须配置 seLinuxOptions。默认使用 seLinuxOptions。针对 seLinuxOptions 所给值执行验证检查。
- *RunAsAny* - 没有提供默认值。允许任意指定的 seLinuxOptions 选项。

AllowedProcMountTypes

allowedProcMountTypes 是一组可以允许的 proc 挂载类型列表。空表或者 nil 值表示只能使用 DefaultProcMountType。

DefaultProcMount 使用容器运行时的默认值设置来决定 /proc 的只读挂载模式和路径屏蔽。大多数容器运行时都会屏蔽 /proc 下面的某些路径以避免特殊设备或信息被不小心暴露给容器。这一配置使所有 Default 字符串值来表示。

此外唯一的 ProcMountType 是 UnmaskedProcMount，意味着即将绕过容器运行时的路径屏蔽行为，确保新创建的 /proc 不会被容器修改。此配置用字符串 Unmasked 来表示。

AppArmor

通过 PodSecurityPolicy 上的注解来控制。详情请参阅 [AppArmor 文档](#)。

Seccomp

Pod 对 seccomp 模版的使用可以通过在 PodSecurityPolicy 上设置注解来控制。Seccomp 是 Kubernetes 的一项 alpha 阶段特性。

seccomp.security.alpha.kubernetes.io/defaultProfileName - 注解用来指定为容器配置默认的 seccomp 模版。可选值为：

- unconfined - 如果没有指定其他替代方案，Seccomp 不会被应用到容器进程上（Kubernets 中的默认设置）。
- runtime/default - 使用默认的容器运行时模版。
- docker/default - 使用 Docker 的默认 seccomp 模版。自 1.11 版本废弃。应改为使用 runtime/default。

- `localhost/<路径名>` - 指定节点上路径 `<seccomp_root>/<路径名>` 下的一个文件作为其模版。其中 `<seccomp_root>` 是通过 kubelet 的标志 `--seccomp-profile-root` 来指定的。

`seccomp.security.alpha.kubernetes.io/allowedProfileNames` - 指定可以为 Pod seccomp 注解配置的值的注解。取值为一个可用值的列表。表中每项可以是上述各值之一，还可以是 `*`，用来表示允许所有的模版。如果没有设置此注解，意味着默认的 seccomp 模版是不可更改的。

Sysctl

默认情况下，所有的安全的 sysctl 都是被允许的。

- `forbiddenSysctls` - 用来排除某些特定的 sysctl。你可以在此列表中禁止一些安全的或者不安全的 sysctl。此选项设置为 `*` 意味着禁止设置所有 sysctl。
- `allowedUnsafeSysctls` - 用来启用那些被默认列表所禁用的 sysctl，前提是所启用的 sysctl 没有被列在 `forbiddenSysctls` 中。

参阅 [Sysctl 文档](#)。

接下来

- 参阅[Pod 安全标准](#) 了解策略建议。
- 阅读 [Pod 安全策略参考](#)了解 API 细节。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

进程 ID 约束与预留

FEATURE STATE: Kubernetes v1.20 [stable]

Kubernetes 允许你限制一个 [Pod](#) 中可以使用的 进程 ID (PID) 数目。你也可以为每个 [节点](#) 预留一定数量的可分配的 PID，供操作系统和守护进程 (而非 Pod) 使用。

进程 ID (PID) 是节点上的一种基础资源。很容易就会在尚未超出其它资源约束的时候就 已经触及任务个数上限，进而导致宿主机器不稳定。

集群管理员需要一定的机制来确保集群中运行的 Pod 不会导致 PID 资源枯竭，甚而造成宿主机上的守护进程（例如 [kubelet](#) 或者 [kube-proxy](#) 乃至包括容器运行时本身）无法正常运行。此外，确保 Pod 中 PID 的个数受限对于保证其不会影响到同一节点上其它负载也很重要。

说明：

在某些 Linux 安装环境中，操作系统会将 PID 约束设置为一个较低的默认值，例如 32768。这时可以考虑提升 /proc/sys/kernel/pid_max 的设置值。

你可以配置 kubelet 限制给定 Pod 能够使用的 PID 个数。例如，如果你的节点上的宿主操作系统被设置为最多可使用 262144 个 PID，同时预期节点上会运行的 Pod 个数不会超过 250，那么你可以为每个 Pod 设置 1000 个 PID 的预算，避免耗尽该节点上可用 PID 的总量。如果管理员系统像 CPU 或内存那样允许对 PID 进行过量分配（Overcommit），他们也可以这样做，只是会有一些额外的风险。不管怎样，任何一个 Pod 都不可以将整个机器的运行状态破坏。这类资源限制有助于避免简单的派生炸弹（Fork Bomb）影响到整个集群的运行。

在 Pod 级别设置 PID 限制使得管理员能够保护 Pod 之间不会互相伤害，不过无法确保所有调度到该宿主机器上的所有 Pod 都不会影响到节点整体。Pod 级别的限制也无法保护节点代理任务自身不会受到 PID 耗尽的影响。

你也可以预留一定量的 PID，作为节点的额外开销，与分配给 Pod 的 PID 集合独立。这有点类似于在给操作系统和其它设施预留 CPU、内存或其它资源时所做的操作，这些任务都在 Pod 及其所包含的容器之外运行。

PID 限制是与[计算资源](#) 请求和限制相辅相成的一种机制。不过，你需要用一种不同的方式来设置这一限制：你需要将其设置到 kubelet 上而不是在 Pod 的 .spec 中为 Pod 设置资源限制。目前还不支持在 Pod 级别设置 PID 限制。

注意：

这意味着，施加在 Pod 之上的限制值可能因为 Pod 运行所在的节点不同而有差别。为了简化系统，最简单的方法是为所有节点设置相同的 PID 资源限制和预留值。

节点级别 PID 限制

Kubernetes 允许你为系统预留一定量的进程 ID。为了配置预留数量，你可以使用 kubelet 的 --system-reserved 和 --kube-reserved 命令行选项中的参数 pid=<number>。你所设置的参数值分别用来声明为整个系统和 Kubernetes 系统守护进程所保留的进程 ID 数目。

说明：

在 Kubernetes 1.20 版本之前，在节点级别通过 PID 资源限制预留 PID 的能力需要启用特性门控 SupportNodePidsLimit 才行。

Pod 级别 PID 限制

Kubernetes 允许你限制 Pod 中运行的进程个数。你可以在节点级别设置这一限制，而不是为特定的 Pod 来将其设置为资源限制。每个节点都可以有不同的 PID 限制设置。要设置限制值，你可以设置 kubelet 的命令行参数 `--pod-max-pids`，或者在 kubelet 的[配置文件](#)中设置 `PodPidsLimit`。

说明：

在 Kubernetes 1.20 版本之前，为 Pod 设置 PID 资源限制的能力需要启用[特性门控](#) `SupportNodePidsLimit` 才行。

基于 PID 的驱逐

你可以配置 kubelet 使之在 Pod 行为不正常或者消耗不正常数量资源的时候将其终止。这一特性称作驱逐。你可以针对不同的驱逐信号[配置资源不足的处理](#)。使用 `pid.available` 驱逐信号来配置 Pod 使用的 PID 个数的阈值。你可以设置硬性的和软性的驱逐策略。不过，即使使用硬性的驱逐策略，如果 PID 个数增长过快，节点仍然可能因为触及节点 PID 限制而进入一种不稳定状态。驱逐信号的取值是周期性计算的，而不是一直能够强制实施约束。

Pod 级别和节点级别的 PID 限制会设置硬性限制。一旦触及限制值，工作负载会在尝试获得新的 PID 时开始遇到问题。这可能会也可能不会导致 Pod 被重新调度，取决于工作负载如何应对这类失败以及 Pod 的存活性和就绪态探测是如何配置的。可是，如果限制值被正确设置，你可以确保其它 Pod 负载和系统进程不会因为某个 Pod 行为不正常而没有 PID 可用。

接下来

- 参阅[PID 约束改进文档](#)以了解更多信息。
- 关于历史背景，请阅读[Kubernetes 1.14 中限制进程 ID 以提升稳定性](#)的博文。
- 请阅读[为容器管理资源](#)。
- 学习如何[配置资源不足情况的处理](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#)。在 GitHub 仓库上登记新的问题[报告问题](#) 或者[提出改进建议](#)。

最后修改 January 17, 2021 at 9:01 PM PST: [\[zh\] Localize concepts/policy/pid-limiting.md \(55b68bd2a\)](#)

调度和驱逐 (Scheduling and Eviction)

在Kubernetes中，调度 (Scheduling) 指的是确保 Pods 匹配到合适的节点，以便 kubelet 能够运行它们。驱逐 (Eviction) 是在资源匮乏的节点上，主动让一个或多个 Pods 失效的过程。

[Pod 开销](#)

[污点和容忍度](#)

[Kubernetes 调度器](#)

[将 Pod 分配给节点](#)

[扩展资源的资源装箱](#)

[驱逐策略](#)

[调度框架](#)

[调度器性能调优](#)

Pod 开销

FEATURE STATE: Kubernetes v1.18 [beta]

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

在 Kubernetes 中，Pod 的开销是根据与 Pod 的 [RuntimeClass](#) 相关联的开销在 [准入](#) 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。

启用 Pod 开销

您需要确保在集群中启用了 PodOverhead [特性门控](#)（在 1.18 默认是开启的），以及一个用于定义 overhead 字段的 RuntimeClass。

使用示例

要使用 PodOverhead 特性，需要一个定义 overhead 字段的 RuntimeClass。作为例子，可以在虚拟机和寄宿操作系统中通过一个虚拟化容器运行时来定义 RuntimeClass 如下，其中每个 Pod 大约使用 120MiB:

```
---
kind: RuntimeClass
apiVersion: node.k8s.io/v1
metadata:
  name: kata-fc
handler: kata-fc
overhead:
  podFixed:
    memory: "120Mi"
    cpu: "250m"
```

通过指定 kata-fc RuntimeClass 处理程序创建的工作负载会将内存和 cpu 开销计入资源配额计算、节点调度以及 Pod cgroup 分级。

假设我们运行下面给出的工作负载示例 test-pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  runtimeClassName: kata-fc
  containers:
    - name: busybox-ctr
      image: busybox
      stdin: true
      tty: true
      resources:
        limits:
          cpu: 500m
          memory: 100Mi
    - name: nginx-ctr
      image: nginx
      resources:
        limits:
          cpu: 1500m
          memory: 100Mi
```

在准入阶段 RuntimeClass [准入控制器](#) 更新工作负载的 PodSpec 以包含 RuntimeClass 中定义的 overhead. 如果 PodSpec 中该字段已定义，该 Pod 将会被

拒绝。在这个例子中，由于只指定了 RuntimeClass 名称，所以准入控制器更新了 Pod，包含了一个 overhead。

在 RuntimeClass 准入控制器之后，可以检验一下已更新的 PodSpec：

```
kubectl get pod test-pod -o jsonpath='{.spec.overhead}'
```

输出：

```
map[cpu:250m memory:120Mi]
```

如果定义了 ResourceQuota，则容器请求的总量以及 overhead 字段都将计算在内。

当 kube-scheduler 决定在哪一个节点调度运行新的 Pod 时，调度器会兼顾该 Pod 的 overhead 以及该 Pod 的容器请求总量。在这个示例中，调度器将资源请求和开销相加，然后寻找具备 2.25 CPU 和 320 MiB 内存可用的节点。

一旦 Pod 调度到了某个节点，该节点上的 kubelet 将为该 Pod 新建一个 cgroup。底层容器运行时将在这个 pod 中创建容器。

如果该资源对每一个容器都定义了一个限制（定义了受限的 Guaranteed QoS 或者 Burstable QoS），kubelet 会为与该资源（CPU 的 cpu.cfs_quota_us 以及内存的 memory.limit_in_bytes）相关的 pod cgroup 设定一个上限。该上限基于容器限制总量与 PodSpec 中定义的 overhead 之和。

对于 CPU，如果 Pod 的 QoS 是 Guaranteed 或者 Burstable，kubelet 会基于容器请求总量与 PodSpec 中定义的 overhead 之和设置 cpu.shares。

请看这个例子，验证工作负载的容器请求：

```
kubectl get pod test-pod -o jsonpath='{.spec.containers[*].resources.limits}'
```

容器请求总计 2000m CPU 和 200MiB 内存：

```
map[cpu: 500m memory:100Mi] map[cpu:1500m memory:100Mi]
```

对照从节点观察到的情况来检查一下：

```
kubectl describe node | grep test-pod -B2
```

该输出显示请求了 2250m CPU 以及 320MiB 内存，包含了 PodOverhead 在内：

Namespace Requests	Name Memory Limits	AGE	CPU Requests	CPU Limits	Memory
-----	----	-----	-----	-----	-----
default (1%)	test-pod 320Mi (1%)	36m	2250m (56%)	2250m (56%)	320Mi

验证 Pod cgroup 限制

在工作负载所运行的节点上检查 Pod 的内存 cgroups. 在接下来的例子中，将在该节点上使用具备 CRI 兼容的容器运行时命令行工具 [cricl](#). 这是一个展示 PodOverhead 行为的进阶示例，用户并不需要直接在该节点上检查 cgroups.

首先在特定的节点上确定该 Pod 的标识符：

```
# 在该 Pod 调度的节点上执行如下命令：  
POD_ID=$(sudo crictl pods --name test-pod -q)"
```

可以依此判断该 Pod 的 cgroup 路径：

```
# 在该 Pod 调度的节点上执行如下命令：  
sudo crictl inspectp -o=json $POD_ID | grep cgroupsPath
```

执行结果的 cgroup 路径中包含了该 Pod 的 pause 容器。Pod 级别的 cgroup 即上面的一个目录。

```
"cgroupsPath": "/kubepods/podd7f4b509-cf94-4951-9417-  
d1087c92a5b2/7ccf55aee35dd16aca4189c952d83487297f3cd760f1bbf09620e206  
e7d0c27a"
```

在这个例子中，该 pod 的 cgroup 路径是 kubepods/podd7f4b509-cf94-4951-9417-d1087c92a5b2。验证内存的 Pod 级别 cgroup 设置：

```
# 在该 Pod 调度的节点上执行这个命令。  
# 另外，修改 cgroup 的名称以匹配为该 pod 分配的 cgroup。  
cat /sys/fs/cgroup/memory/kubepods/podd7f4b509-cf94-4951-9417-  
d1087c92a5b2/memory.limit_in_bytes
```

和预期的一样是 320 MiB

```
335544320
```

可观测性

在 [kube-state-metrics](#) 中可以通过 kube_pod_overhead 指标来协助确定何时使用 PodOverhead 以及协助观察以一个既定开销运行的工作负载的稳定性。该特性在 kube-state-metrics 的 1.9 发行版本中不可用，不过预计将在后续版本中发布。在此之前，用户需要从源代码构建 kube-state-metrics.

接下来

- [RuntimeClass](#)
- [PodOverhead 设计](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 30, 2020 at 9:49 AM PST: [outdate \(258f189e7\)](#)

污点和容忍度

节点亲和性（详见[这里](#)）是 [Pod](#) 的一种属性，它使 Pod 被吸引到一类特定的[节点](#)。这可能出于一种偏好，也可能是硬性要求。Taint（污点）则相反，它使节点能够排斥一类特定的 Pod。

容忍度（Tolerations）是应用于 Pod 上的，允许（但并不要求）Pod 调度到带有与之匹配的污点的节点上。

污点和容忍度（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污点，这表示对于那些不能容忍这些污点的 Pod，是不会被该节点接受的。

概念

您可以使用命令 [kubectl taint](#) 给节点增加一个污点。比如，

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

给节点 node1 增加一个污点，它的键名是 key1，键值是 value1，效果是 NoSchedule。这表示只有拥有和这个污点相匹配的容忍度的 Pod 才能够被分配到 node1 这个节点。

若要移除上述命令所添加的污点，你可以执行：

```
kubectl taint nodes node1 key:NoSchedule-
```

您可以在 PodSpec 中定义 Pod 的容忍度。下面两个容忍度均与上面例子中使用 kubectl taint 命令创建的污点相匹配，因此如果一个 Pod 拥有其中的任何一个容忍度都能够被分配到 node1：

```
tolerations:  
- key: "key1"  
  operator: "Equal"  
  value: "value1"  
  effect: "NoSchedule"
```

```
tolerations:  
- key: "key1"  
  operator: "Exists"  
  effect: "NoSchedule"
```

这里是一个使用了容忍度的 Pod :

[pods/pod-with-toleration.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx  
  labels:  
    env: test  
spec:  
  containers:  
  - name: nginx  
    image: nginx  
    imagePullPolicy: IfNotPresent  
  tolerations:  
  - key: "example-key"  
    operator: "Exists"  
    effect: "NoSchedule"
```

operator 的默认值是 Equal。

一个容忍度和一个污点相"匹配"是指它们有一样的键名和效果，并且：

- 如果 operator 是 Exists (此时容忍度不能指定 value)，或者
- 如果 operator 是 Equal，则它们的 value 应该相等

说明：

存在两种特殊情况：

如果一个容忍度的 key 为空且 operator 为 Exists，表示这个容忍度与任意的 key、value 和 effect 都匹配，即这个容忍度能容忍任意 taint。

如果 effect 为空，则可以与所有键名 key 的效果相匹配。

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污点的节点上，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污点，也可以给一个 Pod 添加多个容忍度设置。

Kubernetes 处理多个污点和容忍度的过程就像一个过滤器：从一个节点的所有污点开始

遍历，过滤掉那些 Pod 中存在与之相匹配的容忍度的污点。余下未被过滤的污点的 effect 值决定了 Pod 是否会被分配到该节点，特别是以下情况：

- 如果未被过滤的污点中存在至少一个 effect 值为 NoSchedule 的污点，则 Kubernetes 不会将 Pod 分配到该节点。
- 如果未被过滤的污点中不存在 effect 值为 NoSchedule 的污点，但是存在 effect 值为 PreferNoSchedule 的污点，则 Kubernetes 会尝试将 Pod 分配到该节点。
- 如果未被过滤的污点中存在至少一个 effect 值为 NoExecute 的污点，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污点

```
kubectl taint nodes node1 key1=value1:NoSchedule  
kubectl taint nodes node1 key1=value1:NoExecute  
kubectl taint nodes node1 key2=value2:NoSchedule
```

假定有一个 Pod，它有两个容忍度：

```
tolerations:  
- key: "key1"  
  operator: "Equal"  
  value: "value1"  
  effect: "NoSchedule"  
- key: "key1"  
  operator: "Equal"  
  value: "value1"  
  effect: "NoExecute"
```

在这种情况下，上述 Pod 不会被分配到上述节点，因为其没有容忍度和第三个污点相匹配。但是如果在给节点添加上述污点之前，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，因为第三个污点是三个污点中唯一不能被这个 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 的污点，则任何不能忍受这个污点的 Pod 都会马上被驱逐，任何可以忍受这个污点的 Pod 都不会被驱逐。但是，如果 Pod 存在一个 effect 值为 NoExecute 的容忍度指定了可选属性 tolerationSeconds 的值，则表示在给节点添加了上述污点之后，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，同时一个匹配的污点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。如果在此之前上述污点被删除了，则 Pod 不会被驱逐。

使用例子

通过污点和容忍度，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污点（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 [准入控制器](#)，很容易就能做到）。拥有上述容忍度的 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 污点类似的 label（例如：`dedicated=groupName`），同时 还要在上述准入控制器中给 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` 或 `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，添加这个容忍度的最简单的方法是使用自定义 [准入控制器](#)。比如，我们推荐使用[扩展资源](#)来表示特殊硬件，给配置了特殊硬件的节点添加污点时包含扩展资源名称，然后运行一个 [ExtendedResourceToleration](#) 准入控制器。此时，因为节点已经被设置污点了，没有对应容忍度的 Pod 会被调度到这些节点。但当你创建一个使用了扩展资源的 Pod 时，`ExtendedResourceToleration` 准入控制器会自动给 Pod 加上正确的容忍度，这样 Pod 就会被自动调度到这些配置了特殊硬件的节点上。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 添加容忍度。
- **基于污点的驱逐**：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。

基于污点的驱逐

FEATURE STATE: Kubernetes v1.18 [stable]

前文提到过污点的 effect 值 `NoExecute`会影响已经在节点上运行的 Pod

- 如果 Pod 不能忍受 effect 值为 `NoExecute` 的污点，那么 Pod 将马上被驱逐
- 如果 Pod 能够忍受 effect 值为 `NoExecute` 的污点，但是在容忍度定义中没有指定 `tolerationSeconds`，则 Pod 还会一直在该节点上运行。
- 如果 Pod 能够忍受 effect 值为 `NoExecute` 的污点，而且指定了 `tolerationSeconds`，则 Pod 还能在该节点上继续运行这个指定的时间长度。

当某种条件为真时，节点控制器会自动给节点添加一个污点。当前内置的污点包括：

- 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：节点控制器访问不到节点。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" 云平台驱动，它将给当前节点添加一个污点将其标志为不可用。在 cloud-controller-manager 的一个控制器初始化这个节点后，kubelet 将删除这个污点。

在节点被驱逐时，节点控制器或者 kubelet 会添加带有 NoExecute 效应的相关污点。如果异常状态恢复正常，kubelet 或节点控制器能够移除相关的污点。

说明：为了保证由于节点问题引起的 Pod 驱逐 [速率限制](#) 行为正常，系统实际上会以限定速率的方式添加污点。在像主控节点与工作节点间通信中断等场景下，这样做可以避免 Pod 被大量驱逐。

使用这个功能特性，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 的容忍度可能是下面这样的：

tolerations:

```
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 6000
```

说明：

Kubernetes 会自动给 Pod 添加一个 key 为 node.kubernetes.io/not-ready 的容忍度 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 的容忍度。

同样，Kubernetes 会给 Pod 添加一个 key 为 node.kubernetes.io/unreachable 的容忍度 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 的容忍度。

这种自动添加的容忍度意味着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。

[DaemonSet](#) 中的 Pod 被创建时，针对以下污点自动添加的 NoExecute 的容忍度将不会指定 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。

基于节点状态添加污点

Node 生命周期控制器会自动创建与 Node 条件相对应的带有 NoSchedule 效应的污点。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）。

DaemonSet 控制器自动为所有守护进程添加如下 NoSchedule 容忍度以防 DaemonSet 崩溃：

- node.kubernetes.io/memory-pressure
- node.kubernetes.io/disk-pressure
- node.kubernetes.io/out-of-disk (只适合关键 Pod)
- node.kubernetes.io/unschedulable (1.10 或更高版本)
- node.kubernetes.io/network-unavailable (只适合主机网络配置)

添加上述容忍度确保了向后兼容，您也可以选择自由向 DaemonSet 添加容忍度。

接下来

- 阅读[资源耗尽的处理](#)，以及如何配置其行为
- 阅读[Pod 优先级](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#)。在 GitHub 仓库上登记新的问题[报告问题](#)或者[提出改进建议](#)。

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

Kubernetes 调度器

在 Kubernetes 中，调度是指将[Pod](#) 放置到合适的[Node](#) 上，然后对应 Node 上的[Kubelet](#) 才能够运行这些 pod。

调度概览

调度器通过 kubernetes 的监测 (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](#) 是 Kubernetes 集群的默认调度器，并且是集群 [控制面](#) 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，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。

支持以下两种方式配置调度器的过滤和打分行为：

1. [调度策略](#) 允许你配置过滤的 [谓词\(Predicates\)](#) 和打分的 [优先级\(Priorities\)](#)。
2. [调度配置](#) 允许你配置实现不同调度阶段的插件，包括：QueueSort, Filter, Score, Bind, Reserve, Permit 等等。你也可以配置 kube-scheduler 运行不同的配置文件。

接下来

- 阅读关于 [调度器性能调优](#)
- 阅读关于 [Pod 拓扑分布约束](#)
- 阅读关于 kube-scheduler 的 [参考文档](#)
- 了解关于 [配置多个调度器](#) 的方式
- 了解关于 [拓扑结构管理策略](#)
- 了解关于 [Pod 额外开销](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

将 Pod 分配给节点

你可以约束一个 [Pod](#) 只能在特定的 [节点](#) 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 [标签选择算符](#) 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 放置在可用资源不足的节点上等等）。但在某些情况下，你可能需要进一步控制 Pod 停靠的节点，例如，确保 Pod 最终落在连接了 SSD 的机器上，或者将来自两个不同的服务 且有大量通信的 Pods 被放置在同一个可用区。

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。它包含键值对的映射。为了使 pod 可以在某个节点上运行，该节点的标签中 必须包含这里的每个键值对（它也可以具有其他标签）。最常见的用法的是一对键值对。

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](#)。

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，查看节点当前具有了所指定的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

选择任何一个你想运行的 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](#)



```
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[添加](#)的标签外，节点还预先填充了一组标准标签。这些标签有：

- [kubernetes.io/hostname](#)
- [failure-domain.beta.kubernetes.io/zone](#)
- [failure-domain.beta.kubernetes.io/region](#)
- [topology.kubernetes.io/zone](#)
- [topology.kubernetes.io/region](#)
- [beta.kubernetes.io/instance-type](#)
- [node.kubernetes.io/instance-type](#)
- [kubernetes.io/os](#)
- [kubernetes.io/arch](#)

说明：

这些标签的值是特定于云供应商的，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](#)并且已经[启用 NodeRestriction 准入插件](#)。
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`。

亲和性与反亲和性

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。 亲和性/反亲和性功能极大地扩展了你可以表达约束的类型。 关键的增强点包括：

1. 语言更具表现力（不仅仅是“对完全匹配规则的 AND”）
2. 你可以发现规则是“软需求”/“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。

亲和性功能包含两种类型的亲和性，即“节点亲和性”和“Pod 间亲和性/反亲和性”。 节点亲和性就像现有的 nodeSelector（但具有上面列出的前两个好处），然而 Pod 间亲和性/反亲和性约束 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）。

节点亲和性

节点亲和性概念上类似于 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性，分别为 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution。 你可以视它们为“硬需求”和“软需求”，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但不能保证的偏好。 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 上的亲和性规则，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 的节点亲和性要求的节点上驱逐。

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”。

节点亲和性通过 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性的 Pod 的实例：

[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
```

此节点亲和性规则表示，Pod 只能放置在具有标签键 `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。新的节点亲和性语法支持下面的操作符：`I n` , `NotIn` , `Exists` , `DoesNotExist` , `Gt` , `Lt`。你可以使用 `NotIn` 和 `DoesNotExist` 来实现节点反亲和性行为，或者使用 [节点污点](#) 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，亲和性选择只在 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` 亲和性表达式等）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。

逐个调度方案中设置节点亲和性

FEATURE STATE: Kubernetes v1.20 [beta]

在配置多个[调度方案](#)时，你可以将某个方案与节点亲和性关联起来，如果某个调度方案仅适用于某组特殊的节点时，这样做是很有用的。要实现这点，可以在[调度器配置](#)中为[NodeAffinity 插件](#)添加 addedAffinity 参数。例如：

```
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration

profiles:
  - schedulerName: default-scheduler
  - schedulerName: foo-scheduler
    pluginConfig:
      - name: NodeAffinity
        args:
          addedAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: scheduler-profile
                      operator: In
              values:
                - foo
```

这里的 addedAffinity 除遵从 Pod 规约中设置的节点亲和性之外，还适用于将 .spec.schedulerName 设置为 foo-scheduler。

说明： DaemonSet 控制器[为 DaemonSet 创建 Pods](#)，但该控制器不理会调度方案。因此，建议你保留一个调度方案，例如 default-scheduler，不要在其中设置 addedAffinity。这样，DaemonSet 的 Pod 模板将会使用此调度器名称。否则，DaemonSet 控制器所创建的某些 Pods 可能持续处于不可调度状态。

pod 间亲和性与反亲和性

Pod 间亲和性与反亲和性使你可以 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（或者在反亲和性的情况下不应该）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 标签的标签选择算符必须指定选择算符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、云供应商可用区、云供应商地理区域等。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](#)部分中列出的标签键。

说明：

Pod 间亲和性与反亲和性需要大量的处理，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。

说明：

Pod 反亲和性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。

与节点亲和性一样，当前有两种类型的 Pod 亲和性与反亲和性，即 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution，分别表示“硬性”与“软性”要求。请参阅前面节点亲和性部分中的描述。 required DuringSchedulingIgnoredDuringExecution 亲和性的一个示例是“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 反亲和性的示例将是“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 间亲和性通过 PodSpec 中 affinity 字段下的 podAffinity 字段进行指定。而 Pod 间反亲和性通过 PodSpec 中 affinity 字段下的 podAntiAffinity 字段进行指定。

Pod 使用 pod 亲和性 的示例：

[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: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: security
```

```
operator: In
values:
- S2
topologyKey: topology.kubernetes.io/zone
containers:
- name: with-pod-affinity
  image: k8s.gcr.io/pause:2.0
```

在这个 Pod 的亲和性配置定义了一条 Pod 亲和性规则和一条 Pod 反亲和性规则。在此示例中，podAffinity 配置为 requiredDuringSchedulingIgnoredDuringExecution，然而 podAntiAffinity 配置为 preferredDuringSchedulingIgnoredDuringExecution。Pod 亲和性规则表示，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 topology.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 topology.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 反亲和性规则表示，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 topology.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](#)以获取 Pod 亲和性与反亲和性的更多样例，包括 requiredDuringSchedulingIgnoredDuringExecution 和 preferredDuringSchedulingIgnoredDuringExecution 两种配置。

Pod 亲和性与反亲和性的合法操作符有 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. 对于亲和性与 requiredDuringSchedulingIgnoredDuringExecution 要求的 Pod 反亲和性，topologyKey 不允许为空。
2. 对于 requiredDuringSchedulingIgnoredDuringExecution 要求的 Pod 反亲和性，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 反亲和性，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, topology.kubernetes.io/zone 和 topology.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 亲和性/反亲和性的定义所在的命名空间。

所有与 requiredDuringSchedulingIgnoredDuringExecution 亲和性与反亲和性关联的 matchExpressions 必须满足，才能将 pod 调度到节点上。

更实际的用例

Pod 间亲和性与反亲和性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:1.16-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](#)，以获取配置反亲和性来达到高可用性的 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，并且给定节点上运行的 kubelet 进程尝试执行该 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](#) 允许节点排斥一组 Pod。

[节点亲和性与 pod 间亲和性/反亲和性](#) 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。 [拓扑管理器](#) 可以参与到节点级别的资源分配决定中。

反馈

此页是否对您有帮助 ?

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 January 11, 2021 at 3:18 PM PST: [Resync concepts/scheduling-eviction/assign-pod-node.md \(8c834f22c\)](#)

扩展资源的资源装箱

FEATURE STATE: Kubernetes 1.16 [alpha]

使用 RequestedToCapacityRatioResourceAllocation 优先级函数，可以将 kube-scheduler 配置为支持包含扩展资源在内的资源装箱操作。 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

在 Kubernetes 1.15 之前，Kube-scheduler 通常允许根据对主要资源（如 CPU 和内存）的请求数量和可用容量之比率对节点评分。 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类资源的权重，以便根据请求数量与可用容量之比率为节点评分。 这就使得用户可以通过使用适当的参数来对扩展资源执行装箱操作，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。 该标志由两个参数 shape 和 resources 组成。 shape 允许用户根据 utilization 和 score 值将函数调整为最少请求 (least requested) 或 最多请求 (most requested) 计算。 resources 包含由 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 设置为对扩展资源 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}  
          ]  
        }  
      }  
    },  
  ],  
}
```

默认情况下此功能处于被禁用状态

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 utilization 为 0% 时给节点评分为 0，在 utilization 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 (least requested) 模式，必须按如下方式反转得分值。

```
{"utilization": 0, "score": 10},  
{"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。同时，weight 不能设置为负值。

RequestedToCapacityRatioResourceAllocation 优先级函数如何对节点评分

本节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来计算节点得分的示例。

请求的资源

intel.com/foo: 2
Memory: 256MB
CPU: 2

资源权重

intel.com/foo: 5
Memory: 1
CPU: 3

FunctionShapePoint {{0, 0}, {100, 10}}

节点 Node 1 配置

可用：

intel.com/foo : 4
Memory : 1 GB
CPU: 8

已用：

intel.com/foo: 1
Memory: 256MB
CPU: 1

节点得分：

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

节点 Node 2 配置

可用：

intel.com/foo: 8
Memory: 1GB
CPU: 8

已用：

intel.com/foo: 2
Memory: 512MB
CPU: 6

节点得分：

intel.com/foo = resourceScoringFunction((2+2),8)
= $(100 - ((8-4)*100/8))$
= $(100 - 50)$
= 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

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 03, 2021 at 7:26 PM PST: [Update resource-bin-packing.md \(6122203b6\)](#)

驱逐策略

本页提供 Kubernetes 驱逐策略的概览。

驱逐策略

[Kubelet](#) 主动监测和防止 计算资源的全面短缺。在资源短缺时，kubelet 可以主动地结束一个或多个 Pod 以回收短缺的资源。当 kubelet 结束一个 Pod 时，它将终止 Pod 中的所有容器，而 Pod 的 Phase 将变为 Failed。如果被驱逐的 Pod 由 Deployment 管理，这个 Deployment 会创建另一个 Pod 给 Kubernetes 来调度。

接下来

- 阅读[配置资源不足的处理](#)， 进一步了解驱逐信号和阈值。

反馈

此页是否对您有帮助？

是否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 16, 2020 at 2:16 PM PST: [\[zh\] Sync changes from English site \(7\) \(089040daa\)](#)

调度框架

FEATURE STATE: Kubernetes 1.15 [alpha]

调度框架是 Kubernetes Scheduler 的一种可插入架构，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](#) 获取框架设计的更多技术信息。

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"。

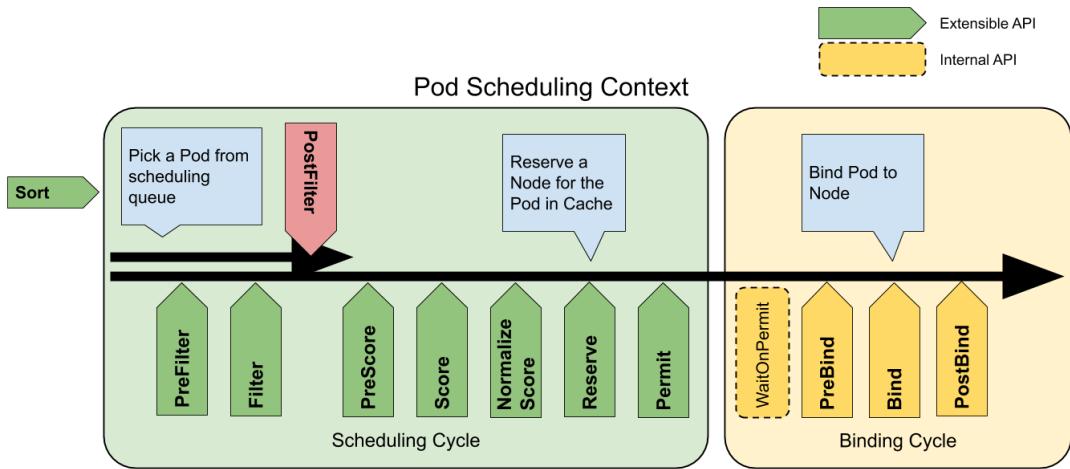
调度周期是串行运行的，而绑定周期可能是同时运行的。

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 PreFilter 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

这些插件在筛选阶段后调用，但仅在该 Pod 没有可行的节点时调用。插件按其配置的顺序调用。如果任何后过滤器插件标记节点为“可调度”，则其余的插件不会调用。典型的后筛选实现是抢占，试图通过抢占其他 Pod 的资源使该 Pod 可以调度。

前置评分

前置评分插件用于执行“前置评分”工作，即生成一个可共享状态供评分插件使用。如果 PreScore 插件返回错误，则调度周期将终止。

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](#)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](#)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 NormalizeScore 插件返回错误，则调度阶段将终止。

说明：希望执行“预保留”工作的插件应该使用 NormalizeScore 扩展点。

Reserve

Reserve 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](#)插件（失败时）或[绑定后](#)插件（成功时）。

Permit

Permit 插件在每个 Pod 调度周期的最后调用，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 Permit 插件批准 Pod 后，该 Pod 将被发送以进行绑定。

1. 拒绝

如果任何 Permit 插件拒绝 Pod，则该 Pod 将被返回到调度队列。这将触发 [Unreserve](#) 插件。

1. 等待 (带有超时)

如果一个 Permit 插件返回 "等待" 结果，则 Pod 将保持在一个内部的 "等待中" 的 Pod 列表，同时该 Pod 的绑定周期启动时即直接阻塞直到得到 [批准](#)。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 [Unreserve](#) 插件。

说明：尽管任何插件可以访问 "等待中" 状态的 Pod 列表并批准它们 (查看 [FrameworkHandle](#))。我们希望只有允许插件可以批准处于 "等待中" 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 PreBind 插件返回错误，则 Pod 将被[拒绝](#)并且退回到调度队列中。

Bind

Bind 插件用于将 Pod 绑定到节点上。直到所有的 PreBind 插件都完成，Bind 插件才会被调用。各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。

Unreserve

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 Unreserve 插件将被通知。Unreserve 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[Reserve](#)。

插件 API

插件 API 分为两个步骤。首先，插件必须完成注册并配置，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(context.Context, *framework.CycleState, *v1.Pod) error
}

// ...
```

插件配置

你可以在调度器配置中启用或禁用插件。如果你在使用 Kubernetes v1.18 或更高版本，大部分调度 [插件](#) 都在使用中且默认启用。

除了默认的插件，你还可以实现自己的调度插件并且将它们与默认插件一起配置。你可以访问[scheduler-plugins](#) 了解更多信息。

如果你正在使用 Kubernetes v1.18 或更高版本，你可以将一组插件设置为一个调度器配置文件，然后定义不同的配置文件来满足各类工作负载。了解更多关于[多配置文件](#)。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 December 04, 2020 at 2:10 PM PST: [\[zh\] Fix links in zh localization \(1\) \(482351ef2\)](#)

调度器性能调优

FEATURE STATE: Kubernetes 1.14 [beta]

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 (*Binding*)。

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。

在大规模集群中，你可以调节调度器的表现来平衡调度的延迟（新 Pod 快速就位）和精度（调度器很少做出糟糕的放置决策）。

你可以通过设置 kube-scheduler 的 `percentageOfNodesToScore` 来配置这个调优设置。 这个 `KubeSchedulerConfiguration` 设置决定了调度集群中节点的阈值。

设置阈值

`percentageOfNodesToScore` 选项接受从 0 到 100 之间的整数值。 0 值比较特殊，表示 kube-scheduler 应该使用其编译后的默认值。 如果你设置 `percentageOfNodesToScore` 的值超过了 100， kube-scheduler 的表现等价于设置值为 100。

要修改这个值，编辑 kube-scheduler 的配置文件（通常是 `/etc/kubernetes/config/kube-scheduler.yaml`），然后重启调度器。

修改完成后，你可以执行

```
kubectl get pods -n kube-system | grep kube-scheduler
```

来检查该 kube-scheduler 组件是否健康。

节点打分阈值

要提升调度性能，kube-scheduler 可以在找到足够的可调度节点之后停止查找。 在大规模集群中，比起考虑每个节点的简单方法相比可以节省时间。

你可以使用整个集群节点总数的百分比作为阈值来指定需要多少节点就足够。 kube-scheduler 会将它转换为节点数的整数值。 在调度期间，如果 kube-scheduler 已确认的可调度节点数足以超过了配置的百分比数量， kube-scheduler 将停止继续查找可调度节点并继续进行 [打分阶段](#)。

[调度器如何遍历节点](#) 详细介绍了这个过程。

默认阈值

如果你不指定阈值，Kubernetes 使用线性公式计算出一个比例，在 100-节点集群下取 50%，在 5000-节点的集群下取 10%。这个自动设置的参数的最低值是 5%。

这意味着，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。

如果你想让调度器对集群内所有节点进行打分，则将 `percentageOfNodesToScore` 设置为 100。

示例

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
```

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 的最小值是硬编码在程序中。

说明：

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。

同理，在小规模集群中，如果你将 `percentageOfNodesToScore` 设置为一个较低的值，则没有或者只有很小的效果。

如果集群只有几百个节点或者更少，请保持这个配置的默认值。改变基本不会对调度器的性能有明显的提升。

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，很多节点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节点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节点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节点来运行这个 Pod 时，才使用很低的参数设置。

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 调度到节点上时，为了让集群中所有节点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 percentage OfNodesToScore 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

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 16, 2020 at 2:16 PM PST: [\[zh\] Sync changes from English site \(7\) \(089040daa\)](#)

集群管理

关于创建和管理 Kubernetes 集群的底层细节。

集群管理概述面向任何创建和管理 Kubernetes 集群的读者人群。我们假设你大概了解一些核心的 Kubernetes [概念](#)。

规划集群

查阅[安装](#)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。

说明：并非所有发行版都是被积极维护的。请选择使用最近 Kubernetes 版本测试过的发行版。

在选择一个指南前，有一些因素需要考虑：

- 你是打算在你的计算机上尝试 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。
- 你正在使用类似 [Google Kubernetes Engine](#) 这样的**被托管的 Kubernetes 集群**，还是**管理你自己的集群**？
- 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](#)最适合。
- 你的 Kubernetes 在**裸金属硬件**上还是**虚拟机 (VMs)** 上运行？
- 你**只想运行一个集群**，还是打算**参与开发 Kubernetes 项目代码**？如果是后者，请选择一个处于开发状态的发行版。某些发行版只提供二进制发布版，但提供更多选择。
- 让你自己熟悉运行一个集群所需的[组件](#)。

管理集群

- 学习如何[管理节点](#)。
- 学习如何设定和管理集群共享的[资源配置](#)。

保护集群

- [证书](#) 节描述了使用不同的工具链生成证书的步骤。
- [Kubernetes 容器环境](#) 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](#) 描述了如何为用户和 service accounts 建立权限许可。
- [身份认证](#) 节阐述了 Kubernetes 中的身份认证功能，包括许多认证选项。
- [鉴权](#) 与身份认证不同，用于控制如何处理 HTTP 请求。
- [使用准入控制器](#) 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 集群中使用 Sysctls](#) 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](#) 描述了如何与 Kubernetes 的审计日志交互。

保护 kubelet

- [主控节点通信](#)
- [TLS 引导](#)
- [Kubelet 认证/授权](#)

可选集群服务

- [DNS 集成](#) 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](#) 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。

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。

```
curl -LO https://storage.googleapis.com/kubernetes-release/easy-rsa/easy-rsa.tar.gz  
tar xzf easy-rsa.tar.gz  
cd easy-rsa-master/easyrsa3  
. ./easyrsa init-pki
```

2. 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)

```
./easyrsa --batch "--req-cn=${MASTER_IP}@`date +%s`" build-ca nopass
```

3. 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。

```
./easyrsa --subject-alt-name="IP:${MASTER_IP},\"  
"IP:${MASTER_CLUSTER_IP},\"  
"DNS:kubernetes,"\  
"DNS:kubernetes.default,"\  
"DNS:kubernetes.default.svc,"\  
"DNS:kubernetes.default.svc.cluster,"\  
"DNS:kubernetes.default.svc.cluster.local" \  
--days=10000 \  
build-server-full server nopass
```

4. 拷贝 pki/ca.crt、pki/issued/server.crt 和 pki/private/server.key 至您的目录。
5. 填充并在 API 服务器的启动参数中添加以下参数：

```
--client-ca-file=/yourdirectory/ca.crt  
--tls-cert-file=/yourdirectory/server.crt  
--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 :

```
openssl genrsa -out ca.key 2048
```

2. 依据 ca.key 生成 ca.crt (使用 -days 参数来设置证书有效时间) :

```
openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER_IP}" -  
days 10000 -out ca.crt
```

3. 生成密钥位数为 2048 的 server.key :

```
openssl genrsa -out server.key 2048
```

4. 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。

```
[ req ]  
default_bits = 2048  
prompt = no  
default_md = sha256  
req_extensions = req_ext  
distinguished_name = dn  
  
[ dn ]  
C = <国家>  
ST = <州/省>  
L = <市>  
O = <组织>  
OU = <部门>  
CN = <MASTER_IP>  
  
[ req_ext ]  
subjectAltName = @alt_names  
  
[ alt_names ]  
DNS.1 = kubernetes  
DNS.2 = kubernetes.default  
DNS.3 = kubernetes.default.svc  
DNS.4 = kubernetes.default.svc.cluster  
DNS.5 = kubernetes.default.svc.cluster.local  
IP.1 = <MASTER_IP>
```

```
IP.2 = <MASTER_CLUSTER_IP>
```

```
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alt_names
```

5. 基于配置文件生成证书签名请求：

```
openssl req -new -key server.key -out server.csr -config csr.conf
```

6. 使用 ca.key、ca.crt 和 server.csr 生成服务器证书：

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 10000 \
-extensions v3_ext -extfile csr.conf
```

7. 查看证书：

```
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 是用来生成证书的另一种工具。

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。

```
curl -L https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 -o cfssl
chmod +x cfssl
curl -L https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 -o cfssljson
chmod +x cfssljson
curl -L https://pkg.cfssl.org/R1.2/cfssl-certinfo_linux-amd64 -o cfssl-certinfo
chmod +x cfssl-certinfo
```

2. 创建目录来存放物料，并初始化 cfssl：

```
mkdir cert
cd cert
./cfssl print-defaults config > config.json
./cfssl print-defaults csr > csr.json
```

3. 创建用来生成 CA 文件的 JSON 配置文件，例如 ca-config.json：

```
{
  "signing": {
```

```

    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
          "signing",
          "key encipherment",
          "server auth",
          "client auth"
        ],
        "expiry": "8760h"
      }
    }
  }
}

```

4. 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。

```

{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    "C": "<country>",
    "ST": "<state>",
    "L": "<city>",
    "O": "<organization>",
    "OU": "<organization unit>"
  ]
}

```

5. 生成 CA 密钥 (ca-key.pem) 和证书 (ca.pem) :

```
./cfssl gencert -initca ca-csr.json | ./cfssljson -bare ca
```

6. 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。

```
{
  "CN": "kubernetes",
  "hosts": [

```

```
"127.0.0.1",
"<MASTER_IP>",
"<MASTER_CLUSTER_IP>",
"kubernetes",
"kubernetes.default",
"kubernetes.default.svc",
"kubernetes.default.svc.cluster",
"kubernetes.default.svc.cluster.local"
],
"key": {
  "algo": "rsa",
  "size": 2048
},
"names": [
  "C": "<country>",
  "ST": "<state>",
  "L": "<city>",
  "O": "<organization>",
  "OU": "<organization unit>"
}]
}
```

7. 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：

```
..../cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
--config=ca-config.json -profile=kubernetes \
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](#)记录的方式，使用 certificates.k8s.io API 来准备 x509 证书，用于认证。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 19, 2021 at 1:18 AM PST: [\[zh\] upgrade cfssl installations \(009fd0f66\)](#)

管理资源

你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你的应用部署，包括扩缩容和更新。我们将更深入讨论的特性包括 [配置文件](#) 和 [标签](#)。

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](#)



```
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:1.14.2
      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，那么你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，特别是删除你之前创建的资源：

```
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，你会发现使用 -l 或 --selector 指定筛选器（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。

如果你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)
```

正确的做法是，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
```

如果你有兴趣进一步学习关于 kubectl 的内容，请阅读 [kubectl 概述](#)。

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](#) 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 (Canary Deployments)

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，你可以使用 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，你可以创建 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
```

你可以调整 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，你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](#)。

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。想要查看你刚才标记的 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](#) 和 [kubectl label](#) 命令文档。

更新注解

有时，你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](#) 和 [kubectl annotate](#) 命令文档。

扩缩你的应用

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扩缩。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，你的 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，你的 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](#) 命令文档、[kubectl autoscale](#) 命令文档和 [水平 Pod 自动伸缩](#) 文档。

就地更新资源

有时，有必要对你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](#)），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，你可以用 [kubectl 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，并应用你所做的更改，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。执行时，`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` 时 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，`kubectl` 不会删除它们。

所有后续的 `kubectl apply` 操作以及其他修改配置的命令，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。

说明：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，你也可以使用 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
```

这使你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 文档](#)。

kubectl patch

你可以使用 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](#) 和 [kubectl patch](#)。

破坏性的更新

在某些情况下，你可能需要更新某些初始化后无法更新的资源字段，或者你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，你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，你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。

我们将指导你通过 Deployment 如何创建和更新应用。

假设你正运行的是 1.14.2 版本的 nginx :

```
kubectl create deployment my-nginx --image=nginx:1.14.2
```

```
deployment.apps/my-nginx created
```

要更新到 1.16.1 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:1.14.2 修改为 nginx:1.16.1。

```
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关闭，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](#)。

接下来

- 学习[如何使用 kubectl 观察和调试应用](#)
- 阅读[配置最佳实践和技巧](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

集群网络系统

集群网络系统是 Kubernetes 的核心部分，但是想要准确了解它的工作原理可是个不小的挑战。下面列出的是网络系统的四个主要问题：

1. 高度耦合的容器间通信：这个已经被 [Pods](#) 和 localhost 通信解决了。
2. Pod 间通信：这个是本文档的重点要讲述的。
3. Pod 和服务间通信：这个已经在[服务](#) 里讲述过了。
4. 外部和服务间通信：这也已经在[服务](#) 讲述过了。

Kubernetes 的宗旨就是在应用之间共享机器。通常来说，共享机器需要两个应用之间不能使用相同的端口，但是在多个应用开发者之间去大规模地协调端口是件很困难的事情，尤其是还要让用户暴露在他们控制范围之外的集群级别的问题上。

动态分配端口也会给系统带来很多复杂度 - 每个应用都需要设置一个端口的参数，而 API 服务器还需要知道如何将动态端口数值插入到配置模块中，服务也需要知道如何找到对方等等。与其去解决这些问题，Kubernetes 选择了其他不同的方法。

Kubernetes 网络模型

每一个 Pod 都有它自己的IP地址，这就意味着你不需要显式地在每个 Pod 之间创建链接，你几乎不需要处理容器端口到主机端口之间的映射。这将创建一个干净的、向后兼容的模型，在这个模型里，从端口分配、命名、服务发现、负载均衡、应用配置和迁移的角度来看，Pod 可以被视作虚拟机或者物理主机。

Kubernetes 对所有网络设施的实施，都需要满足以下的基本要求（除非有设置一些特定的网络分段策略）：

- 节点上的 Pod 可以不通过 NAT 和其他任何节点上的 Pod 通信
- 节点上的代理（比如：系统守护进程、kubelet）可以和节点上的所有Pod通信

备注：仅针对那些支持 Pods 在主机网络中运行的平台(比如：Linux)：

- 那些运行在节点的主机网络里的 Pod 可以不通过 NAT 和所有节点上的 Pod 通信

这个模型不仅不复杂，而且还和 Kubernetes 的实现廉价的从虚拟机向容器迁移的初衷相兼容，如果你的工作开始是在虚拟机中运行的，你的虚拟机有一个 IP，这样就可以和其他的虚拟机进行通信，这是基本相同的模型。

Kubernetes 的 IP 地址存在于 Pod 范围内 - 容器共享它们的网络命名空间 - 包括它们的 IP 地址和 MAC 地址。这就意味着 Pod 内的容器都可以通过 localhost 到达各个端口。这也意味着 Pod 内的容器都需要相互协调端口的使用，但是这和虚拟机中的进程似乎没有什么不同，这也被称为"一个 Pod 一个 IP" 模型。

如何实现这一点是正在使用的容器运行时的特定信息。

也可以在 node 本身通过端口去请求你的 Pod（称之为宿主机端口），但这是一个很特殊的操作。转发方式如何实现也是容器运行时的细节。Pod 自己并不知道这些宿主机端口是否存在。

如何实现 Kubernetes 的网络模型

有很多种方式可以实现这种网络模型，本文档并不是对各种实现技术的详细研究，但是希望可以作为对各种技术的详细介绍，并且成为你研究的起点。

接下来的网络技术是按照首字母排序，顺序本身并无其他意义。

注意：本部分链接到提供 Kubernetes 所需功能的第三方项目。

Kubernetes 项目作者不负责这些项目。此页面遵循[CNCF 网站指南](#)，按字母顺序列出项目。要将项目添加到此列表中，请在提交更改之前阅读[内容指南](#)。

ACI

[Cisco Application Centric Infrastructure](#) 提供了一个集成覆盖网络和底层 SDN 的解决方案来支持容器、虚拟机和其他裸机服务器。[ACI](#) 为 ACI 提供了容器网络集成。[点击这里](#) 查看概述。

Antrea

[Antrea](#) 项目是一个开源的联网解决方案，旨在成为 Kubernetes 原生的网络解决方案。它利用 Open vSwitch 作为网络数据平面。Open vSwitch 是一个高性能可编程的虚拟交换机，支持 Linux 和 Windows 平台。Open vSwitch 使 Antrea 能够以高性能和高效的方式实现 Kubernetes 的网络策略。借助 Open vSwitch 可编程的特性，Antrea 能够在 Open vSwitch 之上实现广泛的联网、安全功能和服务。

Apstra 的 AOS

[AOS](#) 是一个基于意图的网络系统，可以通过一个简单的集成平台创建和管理复杂的数据中心环境。AOS 利用高度可扩展的分布式设计来消除网络中断，同时将成本降至最低。

AOS 参考设计当前支持三层连接的主机，这些主机消除了旧的两层连接的交换问题。这些三层连接的主机可以是 Linux (Debian、Ubuntu、CentOS) 系统，它们直接在机架式交换机 (TOR) 的顶部创建 BGP 邻居关系。AOS 自动执行路由邻接，然后提供对 Kubernetes 部署中常见的路由运行状况注入 (RHI) 的精细控制。

AOS 具有一组丰富的 REST API 端点，这些端点使 Kubernetes 能够根据应用程序需求快速更改网络策略。进一步的增强功能将用于网络设计的 AOS Graph 模型与工作负载供应集成在一起，从而为私有云和公共云提供端到端管理系统。

AOS 支持使用包括 Cisco、Arista、Dell、Mellanox、HPE 在内的制造商提供的通用供应商设备，以及大量白盒系统和开放网络操作系统，例如 Microsoft SONiC、Dell OPX 和 Cumulus Linux。

想要更详细地了解 AOS 系统是如何工作的可以点击这里：<https://www.apstra.com/products/how-it-works/>

Kubernetes 的 AWS VPC CNI

[AWS VPC CNI](#) 为 Kubernetes 集群提供了集成的 AWS 虚拟私有云 (VPC) 网络。该 CNI 插件提供了高吞吐量和可用性，低延迟以及最小的网络抖动。此外，用户可以使用现有的 AWS VPC 网络和安全最佳实践来构建 Kubernetes 集群。这包括使用 VPC 流日志、VPC 路由策略和安全组进行网络流量隔离的功能。

使用该 CNI 插件，可使 Kubernetes Pod 拥有与在 VPC 网络上相同的 IP 地址。CNI 将 AWS 弹性网络接口 (ENI) 分配给每个 Kubernetes 节点，并将每个 ENI 的辅助 IP 范围用于该节点上的 Pod。CNI 包含用于 ENI 和 IP 地址的预分配的控件，以便加快 Pod 的启动时间，并且能够支持多达 2000 个节点的大型集群。

此外，CNI 可以与 [用于执行网络策略的 Calico](#) 一起运行。AWS VPC CNI 项目是开源的，请查看 [GitHub 上的文档](#)。

Kubernetes 的 Azure CNI

[Azure CNI](#) 是一个开源插件，将 Kubernetes Pods 和 Azure 虚拟网络 (也称为 VNet) 集成在一起，可提供与 VM 相当的网络性能。Pod 可以通过 Express Route 或者 站点到站点的 VPN 来连接到对等的 VNet，也可以从这些网络来直接访问 Pod。Pod 可以访问受服务端点或者受保护链接的 Azure 服务，比如存储和 SQL。你可以使

用 VNet 安全策略和路由来筛选 Pod 流量。该插件通过利用在 Kubernetes 节点的网络接口上预分配的辅助 IP 池将 VNet 分配给 Pod。

Azure CNI 可以在 [Azure Kubernetes Service \(AKS\)](#) 中获得。

Big Switch Networks 的 Big Cloud Fabric

[Big Cloud Fabric](#) 是一个基于云原生的网络架构，旨在在私有云或者本地环境中运行 Kubernetes。它使用统一的物理和虚拟 SDN，Big Cloud Fabric 解决了固有的容器网络问题，比如负载均衡、可见性、故障排除、安全策略和容器流量监控。

在 Big Cloud Fabric 的虚拟 Pod 多租户架构的帮助下，容器编排系统（比如 Kubernetes、RedHat OpenShift、Mesosphere DC/OS 和 Docker Swarm）将与 VM 本地编排系统（比如 VMware、OpenStack 和 Nutanix）进行本地集成。客户将能够安全地互联任意数量的这些集群，并且在需要时启用他们之间的租户间通信。

在最新的 [Magic Quadrant](#) 上，BCF 被 Gartner 认为是非常有远见的。而 BCF 的一条关于 Kubernetes 的本地部署（其中包括 Kubernetes、DC/OS 和在不同地理区域的多个 DC 上运行的 VMware）也在[这里](#)被引用。

Calico

[Calico](#) 是一个开源的联网及网络安全方案，用于基于容器、虚拟机和本地主机的工作负载。Calico 支持多个数据面，包括：纯 Linux eBPF 的数据面、标准的 Linux 联网数据面以及 Windwos HNS 数据面。Calico 在提供完整的联网堆栈的同时，还可与 [云驱动 CNIs](#) 联合使用，以保证网络策略实施。

Cilium

[Cilium](#) 是一个开源软件，用于提供并透明保护应用容器间的网络连接。Cilium 支持 L7/HTTP，可以在 L3-L7 上通过使用与网络分离的基于身份的安全模型寻址来实施网络策略，并且可以与其他 CNI 插件结合使用。

华为的 CNI-Genie

[CNI-Genie](#) 是一个 CNI 插件，可以让 Kubernetes 在运行时使用不同的[网络模型](#)的实现同时被访问。这包括以 [CNI 插件](#)运行的任何实现，比如 [Flannel](#)、[Calico](#)、[Romana](#)、[Weave-net](#)。

CNI-Genie 还支持[将多个 IP 地址分配给 Pod](#)，每个都来自不同的 CNI 插件。

cni-ipvlan-vpc-k8s

[cni-ipvlan-vpc-k8s](#) 包含了一组 CNI 和 IPAM 插件来提供一个简单的、本地主机、低延迟、高吞吐量 以及通过使用 Amazon 弹性网络接口（ENI）并使用 Linux 内核的 IPv2 驱动程序以 L2 模式将 AWS 管理的 IP 绑定到 Pod 中，在 Amazon Virtual Private Cloud（VPC）环境中为 Kubernetes 兼容的网络堆栈。

这些插件旨在直接在 VPC 中进行配置和部署，Kubelets 先启动，然后根据需要进行自我配置和扩展它们的 IP 使用率，而无需经常建议复杂的管理 覆盖网络、BGP、禁用源/

目标检查或调整 VPC 路由表以向每个主机提供每个实例子网的 复杂性（每个 VPC 限制为50-100个条目）。简而言之，cni-ipvlan-vpc-k8s 大大降低了在 AWS 中大规模部署 Kubernetes 所需的网络复杂性。

Coil

[Coil](#) 是一个为易于集成、提供灵活的出站流量网络而设计的 CNI 插件。与裸机相比，Coil 的额外操作开销低，并允许针对外部网络的出站流量任意定义 NAT 网关。

Contiv

[Contiv](#) 为各种使用情况提供了一个可配置网络（使用了 BGP 的本地 L3，使用 vxlan、经典 L2 或 Cisco-SDN/ACI 的覆盖网络）。[Contiv](#) 是完全开源的。

Contrail/Tungsten Fabric

[Contrail](#) 是基于 [Tungsten Fabric](#) 的，真正开放的多云网络虚拟化和策略管理平台。Contrail 和 Tungsten Fabric 与各种编排系统集成在一起，例如 Kubernetes、OpenShift、OpenStack 和 Mesos，并为虚拟机、容器或 Pods 以及裸机工作负载提供了不同的隔离模式。

DANM

[DANM](#) 是一个针对在 Kubernetes 集群中运行的电信工作负载的网络解决方案。它以下几个组件构成：

- 能够配置具有高级功能的 IPVLAN 接口的 CNI 插件
- 一个内置的 IPAM 模块，能够管理多个、群集内的、不连续的 L3 网络，并按请求提供动态、静态或无 IP 分配方案
- CNI 元插件能够通过自己的 CNI 或通过将任务授权给其他任何流行的 CNI 解决方案（例如 SRI-OV 或 Flannel）来实现将多个网络接口连接到容器
- Kubernetes 控制器能够集中管理所有 Kubernetes 主机的 VxLAN 和 VLAN 接口
- 另一个 Kubernetes 控制器扩展了 Kubernetes 的基于服务的服务发现概念，以在 Pod 的所有网络接口上工作

通过这个工具集，DANM 可以提供多个分离的网络接口，可以为 Pod 使用不同的网络后端和高级 IPAM 功能。

Flannel

[Flannel](#) 是一个非常简单的能够满足 Kubernetes 所需要的覆盖网络。已经有许多人报告了使用 Flannel 和 Kubernetes 的成功案例。

Google Compute Engine (GCE)

对于 Google Compute Engine 的集群配置脚本，[高级路由器](#) 用于为每个虚机分配一个子网（默认是 /24 - 254个 IP），绑定到孩子网的任何流量都将通过 GCE 网络结构直接路由到虚机。这是除了分配给虚机的“主”IP 地址之外的一个补充，该 IP 地址经过

NAT 转换以用于访问外网。 Linux 网桥（称为“cbr0”）被配置为存在于孩子网中，并被传递到 Docker 的 --bridge 参数上。

Docker 会以这样的参数启动：

```
DOCKER_OPTS="--bridge=cbr0 --iptables=false --ip-masq=false"
```

这个网桥是由 Kubelet（由 --network-plugin=kubenet 参数控制）根据节点的 .spec.podCIDR 参数创建的。

Docker 将会从 cbr-cidr 块分配 IP。 容器之间可以通过 cbr0 网桥相互访问，也可以访问节点。 这些 IP 都可以在 GCE 的网络中被路由。 而 GCE 本身并不知道这些 IP，所以不会对访问外网的流量进行 NAT。 为了实现此目的，使用了 iptables 规则来伪装（又称为 SNAT，使数据包看起来好像是来自“节点”本身），将通信绑定到 GCE 项目网络（10.0.0.0/8）之外的 IP。

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

最后，在内核中启用了 IP 转发（因此内核将处理桥接容器的数据包）：

```
sysctl net.ipv4.ip_forward=1
```

所有这些的结果是所有 Pod 都可以互相访问，并且可以将流量发送到互联网。

Jaguar

[Jaguar](#) 是一个基于 OpenDaylight 的 Kubernetes 网络开源解决方案。 Jaguar 使用 vxlan 提供覆盖网络，而 Jaguar CNIPlugin 为每个 Pod 提供一个 IP 地址。

k-vswitch

[k-vswitch](#) 是一个基于 [Open vSwitch](#) 的简易 Kubernetes 网络插件。 它利用 Open vSwitch 中现有的功能来提供强大的网络插件，该插件易于操作，高效且安全。

Knitter

[Knitter](#) 是一个支持 Kubernetes 中实现多个网络系统的解决方案。 它提供了租户管理和网络管理的功能。 除了多个网络平面外，Knitter 还包括一组端到端的 NFV 容器网络解决方案，例如为应用程序保留 IP 地址、IP 地址迁移等。

Kube-OVN

[Kube-OVN](#) 是一个基于 OVN 的用于企业的 Kubernetes 网络架构。 借助于 OVN/OVS，它提供了一些高级覆盖网络功能，例如子网、QoS、静态 IP 分配、流量镜像、网关、基于 openflow 的网络策略和服务代理。

Kube-router

[Kube-router](#) 是 Kubernetes 的专用网络解决方案，旨在提供高性能和易操作性。Kube-router 提供了一个基于 Linux [LVS/IPVS](#) 的服务代理、一个基于 Linux 内核转发的无覆盖 Pod-to-Pod 网络解决方案和基于 iptables/ipset 的网络策略执行器。

L2 networks and linux bridging

如果你具有一个“哑”的L2网络，例如“裸机”环境中的简单交换机，则应该能够执行与上述GCE设置类似的操作。请注意，这些说明仅是非常简单的尝试过-似乎可行，但尚未经过全面测试。如果您使用此技术并完善了流程，请告诉我们。

根据 Lars Kellogg-Stedman 的这份非常不错的“Linux 网桥设备”[使用说明](#)来进行操作。

Multus (a Multi Network plugin)

[Multus](#) 是一个多 CNI 插件，使用 Kubernetes 中基于 CRD 的网络对象来支持实现 Kubernetes 多网络系统。

Multus 支持所有[参考插件](#)（比如：[Flannel](#)、[DHCP](#)、[Macvlan](#)）来实现 CNI 规范和第三方插件（比如：[Calico](#)、[Weave](#)、[Cilium](#)、[Contiv](#)）。除此之外，Multus 还支持[SRIOV](#)、[DPDK](#)、[OVS-DPDK & VPP](#)的工作负载，以及 Kubernetes 中基于云的本机应用程序和基于 NFV 的应用程序。

NSX-T

[VMware NSX-T](#) 是一个网络虚拟化和安全平台。NSX-T 可以为多云及多系统管理程序环境提供网络虚拟化，并专注于具有异构端点和技术堆栈的新兴应用程序框架和体系结构。除了 vSphere 管理程序之外，这些环境还包括其他虚拟机管理程序，例如 KVM、容器和裸机。

[NSX-T Container Plug-in \(NCP\)](#) 提供了 NSX-T 与容器协调器（例如 Kubernetes）之间的结合，以及 NSX-T 与基于容器的 CaaS/PaaS 平台（例如 Pivotal Container Service (PKS) 和 OpenShift）之间的集成。

Nuage Networks VCS (Virtualized Cloud Services)

[Nuage](#) 提供了一个高度可扩展的基于策略的软件定义网络 (SDN) 平台。Nuage 使用开源的 Open vSwitch 作为数据平面，以及基于开放标准构建具有丰富功能的 SDN 控制器。

Nuage 平台使用覆盖层在 Kubernetes Pod 和非 Kubernetes 环境 (VM 和裸机服务器) 之间提供基于策略的无缝联网。Nuage 的策略抽象模型在设计时就考虑到了应用程序，并且可以轻松声明应用程序的细粒度策略。该平台的实时分析引擎可为 Kubernetes 应用程序提供可见性和安全性监控。

OpenVSwitch

[OpenVSwitch](#) 是一个较为成熟的解决方案，但同时也增加了构建覆盖网络的复杂性。这也得到了几个网络系统的"大商店"的拥护。

OVN (开放式虚拟网络)

OVN 是一个由 Open vSwitch 社区开发的开源的网络虚拟化解决方案。它允许创建逻辑交换器、逻辑路由、状态 ACL、负载均衡等等来建立不同的虚拟网络拓扑。该项目有一个特定的Kubernetes插件和文档 [ovn-kubernetes](#)。

Romana

[Romana](#) 是一个开源网络和安全自动化解决方案。它可以让你在没有覆盖网络的情况下部署 Kubernetes。Romana 支持 Kubernetes [网络策略](#)，来提供跨网络命名空间的隔离。

Weaveworks 的 Weave Net

[Weave Net](#) 是 Kubernetes 及其 托管应用程序的弹性且易于使用的网络系统。Weave Net 可以作为 [CNI 插件](#) 运行或者独立运行。在这两种运行方式里，都不需要任何配置或额外的代码即可运行，并且在两种情况下，网络都为每个 Pod 提供一个 IP 地址 -- 这是 Kubernetes 的标准配置。

接下来

网络模型的早期设计、运行原理以及未来的一些计划，都在 [联网设计文档](#) 里有更详细的描述。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 17, 2020 at 3:46 AM PST: [\[zh\] sync networking.md \(4bd7d06de\)](#)

Kubernetes 系统组件指标

系统组件指标可以更好地了解系统内部发生的情况。指标对于构建仪表板和告警特别有用。

Kubernetes 组件以 [Prometheus 格式](#) 生成度量值。这种格式是结构化的纯文本，旨在使人和机器都可以阅读。

Kubernetes 中的指标

在大多数情况下，可以在 HTTP 服务器的 /metrics 端点上访问度量值。对于默认情况下不公开端点的组件，可以使用 --bind-address 标志启用。

这些组件的示例：

- [kube-controller-manager](#)
- [kube-proxy](#)
- [kube-apiserver](#)
- [kube-scheduler](#)
- [kubelet](#)

在生产环境中，你可能需要配置 [Prometheus 服务器](#) 或某些其他指标搜集器以定期收集这些指标，并使它们在某种时间序列数据库中可用。

请注意，[kubelet](#) 还会在 /metrics/cadvisor，/metrics/resource 和 /metrics/probes 端点中公开度量值。这些度量值的生命周期各不相同。

如果你的集群使用了 [RBAC](#)，则读取指标需要通过基于用户、组或 ServiceAccount 的鉴权，要求具有允许访问 /metrics 的 ClusterRole。例如：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - nonResourceURLs:
    - "/metrics"
    verbs:
    - get
```

指标生命周期

Alpha 指标 → 稳定的指标 → 弃用的指标 → 隐藏的指标 → 删除的指标

Alpha 指标没有稳定性保证。这些指标可以随时被修改或者删除。

稳定的指标可以保证不会改变。这意味着：

- 稳定的、不包含已弃用 (deprecated) 签名的指标不会被删除 (或重命名)
- 稳定的指标的类型不会被更改

已弃用的指标最终将被删除，不过仍然可用。这类指标包含注解，标明其被废弃的版本。

例如：

- 被弃用之前：

```
# HELP some_counter this counts things
# TYPE some_counter counter
some_counter 0
```

- 被启用之后：

```
# HELP some_counter (Deprecated since 1.15.0) this counts things
# TYPE some_counter counter
some_counter 0
```

隐藏的指标不会再被发布以供抓取，但仍然可用。要使用隐藏指标，请参阅[显式隐藏指标节](#)。

删除的指标不再被发布，亦无法使用。

显示隐藏指标

如上所述，管理员可以通过设置可执行文件的命令行参数来启用隐藏指标，如果管理员错过了上一版本中已经弃用的指标的迁移，则可以把这个用作管理员的逃生门。

`show-hidden-metrics-for-version` 标志接受版本号作为取值，版本号给出你希望显示该发行版本中已弃用的指标。版本表示为 `x.y`，其中 `x` 是主要版本，`y` 是次要版本。补丁程序版本不是必须的，即使指标可能会在补丁程序发行版中弃用，原因是指标弃用策略规定仅针对次要版本。

该参数只能使用前一个次要版本。如果管理员将先前版本设置为 `show-hidden-metrics-for-version`，则先前版本中隐藏的度量值会再度生成。不允许使用过旧的版本，因为那样会违反指标弃用策略。

以指标 A 为例，此处假设 A 在 1.n 中已弃用。根据指标弃用策略，我们可以得出以下结论：

- 在版本 1.n 中，这个指标已经弃用，且默认情况下可以生成。
- 在版本 1.n+1 中，这个指标默认隐藏，可以通过命令行参数 `show-hidden-metrics-for-version=1.n` 来再度生成。
- 在版本 1.n+2 中，这个指标就将被从代码中移除，不会再有任何逃生窗口。

如果你要从版本 1.12 升级到 1.13，但仍依赖于 1.12 中弃用的指标 A，则应通过命令行设置隐藏指标：`--show-hidden-metrics=1.12`，并记住在升级到 1.14 版本之前删除此指标依赖项。

禁用加速器指标

kubelet 通过 cAdvisor 收集加速器指标。为了收集这些指标，对于 NVIDIA GPU 之类的加速器，kubelet 在驱动程序上保持打开状态。这意味着为了执行基础结构更改（例如更新驱动程序），集群管理员需要停止 kubelet 代理。

现在，收集加速器指标的责任属于供应商，而不是 kubelet。供应商必须提供一个收集指标的容器，并将其公开给指标服务（例如 Prometheus）。

[DisableAcceleratorUsageMetrics 特性门控](#) 禁止由 kubelet 收集的指标。关于[何时会在默认情况下启用此功能也有一定规划](#)。

组件指标

kube-controller-manager 指标

控制器管理器指标可提供有关控制器管理器性能和运行状况的重要洞察。这些指标包括通用的 Go 语言运行时指标（例如 go_routine 数量）和控制器特定的度量指标，例如可用于评估集群运行状况的 etcd 请求延迟或云提供商（AWS、GCE、OpenStack）的 API 延迟等。

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。这些指标可用于监控持久卷操作的运行状况。

比如，对于 GCE，这些指标称为：

```
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"}
```

kube-scheduler 指标

FEATURE STATE: Kubernetes v1.20 [alpha]

调度器会暴露一些可选的指标，报告所有运行中 Pods 所请求的资源和期望的约束值。这些指标可用来构造容量规划监控面板、访问调度约束的当前或历史数据、快速发现因为缺少资源而无法被调度的负载，或者将 Pod 的实际资源用量与其请求值进行比较。

kube-scheduler 组件能够辨识各个 Pod 所配置的资源 [请求和约束](#)。在 Pod 的资源请求值或者约束值非零时，kube-scheduler 会以度量值时间序列的形式生成报告。该时间序列值包含以下标签：

- 名字空间
- Pod 名称
- Pod 调度所处节点，或者当 Pod 未被调度时用空字符串表示
- 优先级
- 为 Pod 所指派的调度器
- 资源的名称（例如，cpu）
- 资源的单位，如果知道的话（例如，cores）

一旦 Pod 进入完成状态（其 restartPolicy 为 Never 或 OnFailure，且其处于 Succeeded 或 Failed Pod 阶段，或者已经被删除且所有容器都具有 终止状态），该时间序列停止报告，因为调度器现在可以调度其它 Pod 来执行。这两个指标称作 kube_pod_resource_request 和 kube_pod_resource_limit。

指标暴露在 HTTP 端点 /metrics/resources，与调度器上的 /metrics 端点一样要求相同的访问授权。你必须使用 --show-hidden-metrics-for-version=1.20 标志才能暴露那些稳定性为 Alpha 的指标。

接下来

- 阅读有关指标的 [Prometheus 文本格式](#)
- 查看 [Kubernetes 稳定指标](#) 的列表
- 阅读有关 [Kubernetes 弃用策略](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 11, 2021 at 2:23 PM PST: [\[zh\] Resync concepts/cluster-administration/system-metrics.md \(028396366\)](#)

日志架构

应用日志可以让你了解应用内部的运行状况。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，由容器引擎或运行时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，你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，但是你可以集成许多现有的日志解决方案到 Kubernetes 集群中。

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。如果你对集群级日志不感兴趣，你仍会发现关于如何在节点上存储和处理日志的描述对你是有用的。

Kubernetes 中的基本日志记录

本节，你会看到一个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。这里的示例为包含一个容器的 Pod 规约，该容器每秒钟向标准输出写入数据。

[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:

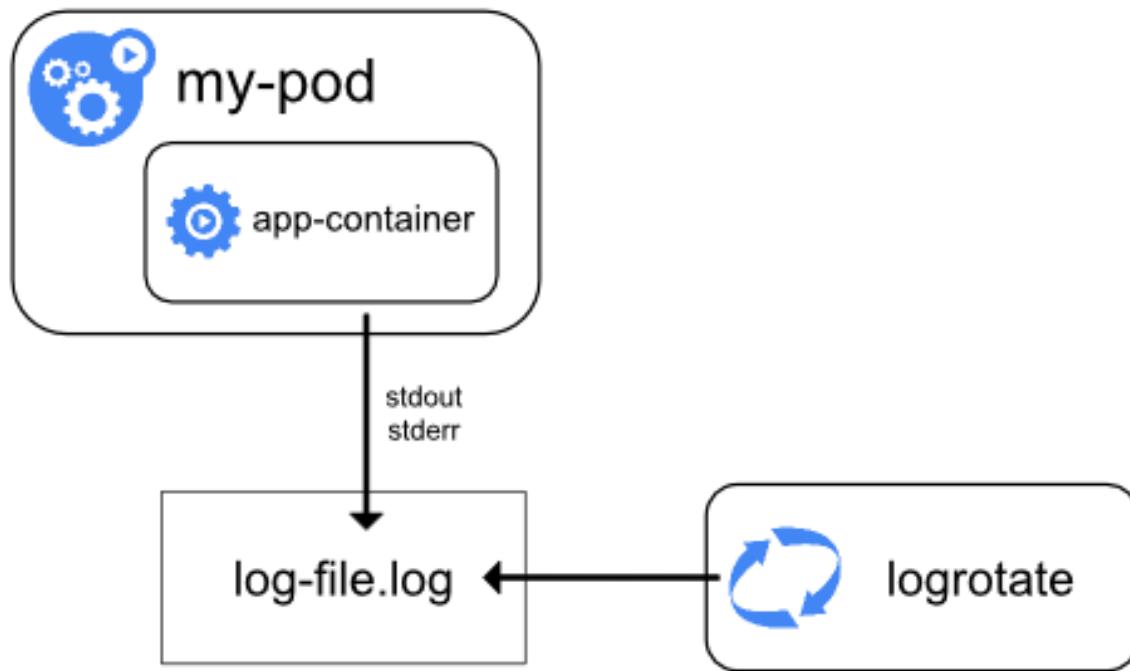
```
kubectl logs counter
```

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，你可以使用命令 kubectl logs 和参数 --previous 检索之前的容器日志。如果 pod 中有多个容器，你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](#)。

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](#)，该日志驱动在 Kubernetes 中配置为以 JSON 格式写入文件。

说明：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。你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](#)。你也可以设置容器 `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，你可以找到关于 `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](#)。

当运行 [kubectl 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。

说明：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](#) 日志库。你可以在[日志开发文档](#)找到这些组件的日志告警级别协议。

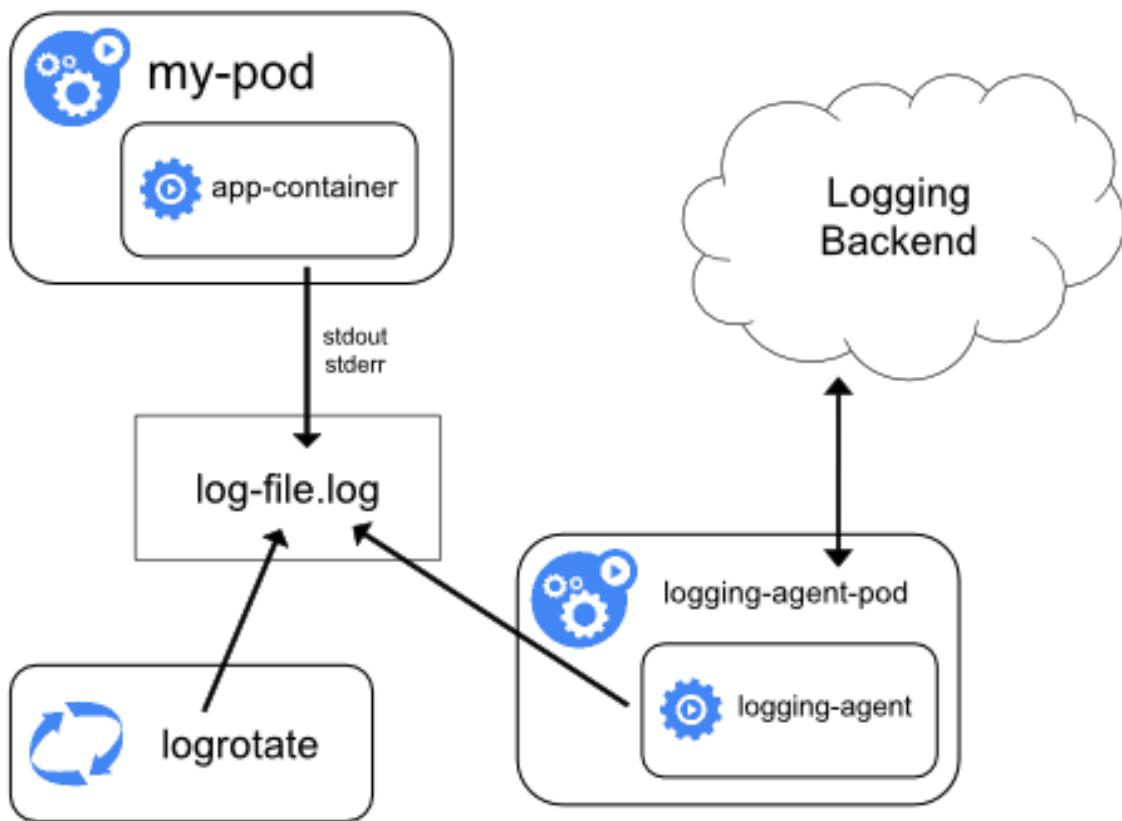
和容器日志类似，`/var/log` 目录中的系统组件日志也应该被轮转。通过脚本 `kube-up.sh` 启动的 Kubernetes 集群中，日志被工具 `logrotate` 执行每日轮转，或者日志大小超过 100MB 时触发轮转。

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，但你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



你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。

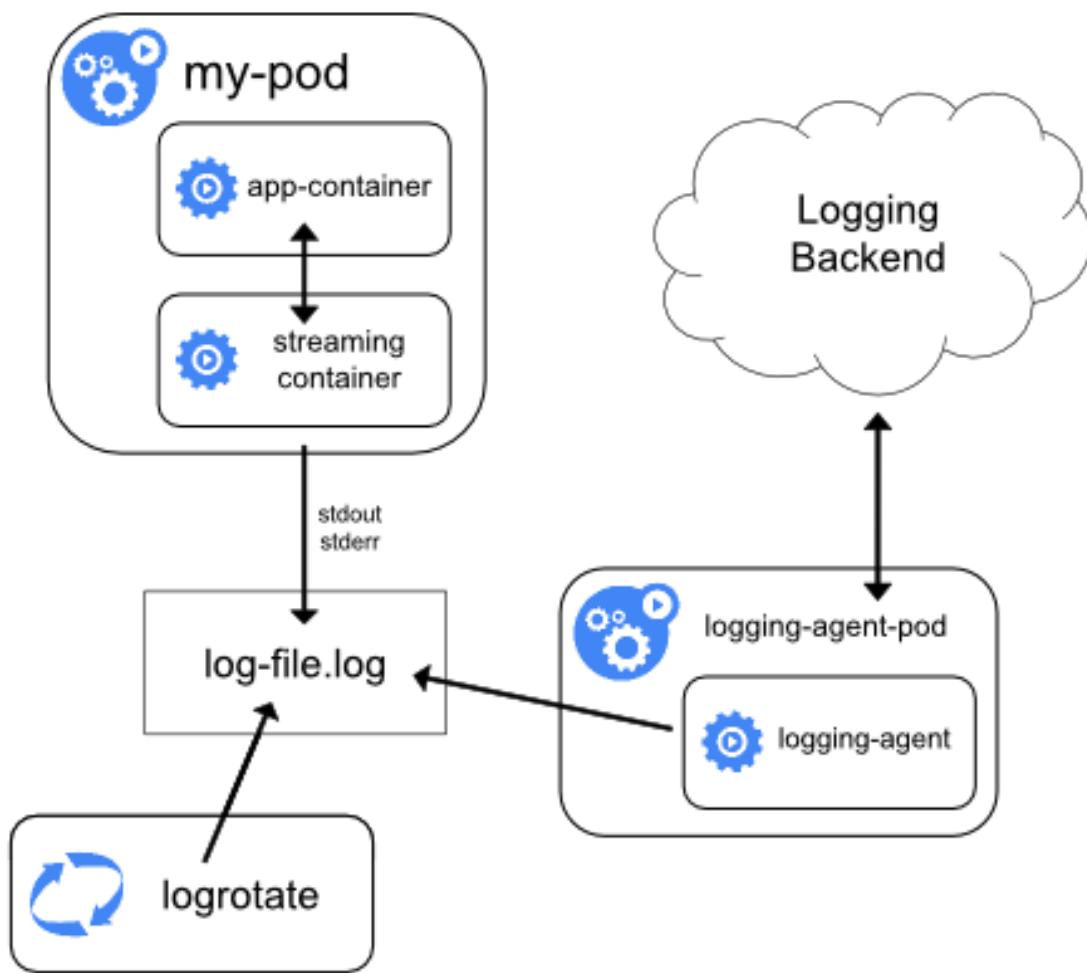
Kubernetes 并不指定日志代理，但是有两个可选的日志代理与 Kubernetes 发行版一起发布。[Stackdriver 日志](#) 适用于 Google Cloud Platform，和 [Elasticsearch](#)。你可以在专门的文档中找到更多的信息和说明。两者都使用 [fluentd](#) 与自定义配置作为节点上的代理。

使用 sidecar 容器和日志代理

你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，你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、套接字或 journald 读取日志。每个 sidecar 容器打印其自己的 stdout 和 stderr 流。

这种方法允许你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](#)
□

```
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，即使你试图重定向它们到容器的 stdout 流。取而代之的是，你可以引入两个 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](#)

```

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: {}
```

现在当你运行这个 Pod 时，你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
```

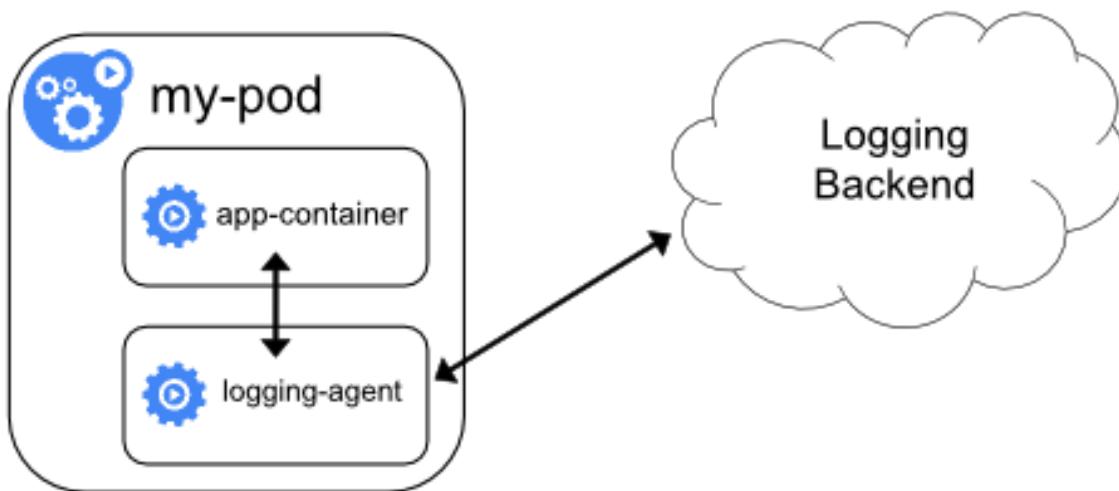
```
Mon Jan 1 00:00:00 UTC 2001 INFO 0
Mon Jan 1 00:00:01 UTC 2001 INFO 1
Mon Jan 1 00:00:02 UTC 2001 INFO 2
...
```

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。如果你愿意，你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。如果你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，你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你的应用程序一起运行。

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

例如，你可以使用 [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](#)。

[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>
```

说明：配置 fluentd 超出了本文的范围。要进一步了解如何配置 fluentd，请参考 [fluentd 官方文档](#).

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](#)



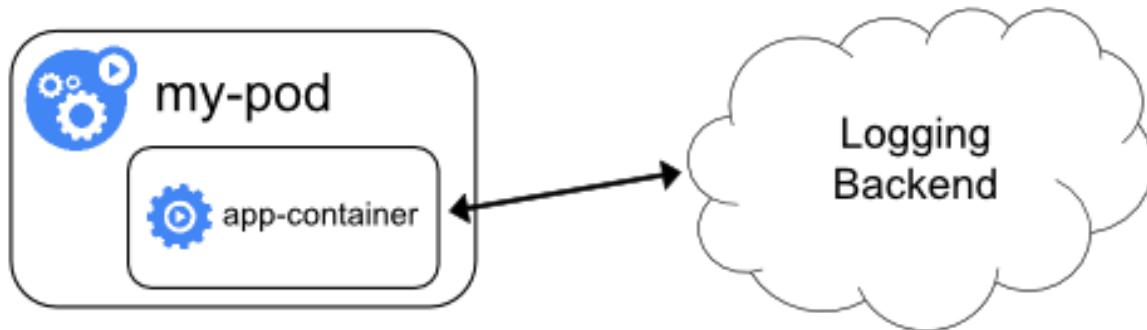
```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
      image: busybox
      args:
        - /bin/sh
        - -C
        - '>
          i=0;
          while true;
          do
            echo "$i: $(date)" >> /var/log/1.log;
            echo "$(date) INFO $i" >> /var/log/2.log;
            i=$((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，你可以在 Stackdriver 界面看到日志消息。

记住，这只是一个例子，事实上你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，你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。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 07, 2020 at 3:56 PM PST: [\[zh\] sync missing period in logging.md \(cfbe452d0\)](#)

系统日志

系统组件的日志记录集群中发生的事件，这对于调试非常有用。你可以配置日志的精细度，以展示更多或更少的细节。日志可以是粗粒度的，如只显示组件内的错误，也可以是细粒度的，如显示事件的每一个跟踪步骤（比如 HTTP 访问日志、pod 状态更新、控制器动作或调度器决策）。

Klog

klog 是 Kubernetes 的日志库。[klog](#) 为 Kubernetes 系统组件生成日志消息。

有关 klog 配置的更多信息，请参见[命令行工具参考](#)。

klog 原始格式的示例：

```
I1025 00:15:15.525108    1 httplog.go:79] GET /api/v1/namespaces/kube-system/pods/metrics-server-v0.3.1-57c75779f-9p8wg: (1.512ms) 200 [pod_nanny/v0.0.0 (linux/amd64) kubernetes/$Format 10.56.1.19:51756]
```

结构化日志

FEATURE STATE: Kubernetes v1.19 [alpha]

警告：

到结构化日志消息的迁移是一个持续的过程。在此版本中，并非所有日志消息都是结构化的。解析日志文件时，你也必须要处理非结构化日志消息。

日志格式和值的序列化可能会发生变化。

结构化日志记录旨在日志消息中引入统一结构，以方便提取信息，使日志的存储和处理更容易、成本更低。新的消息格式向后兼容，并默认启用。

结构化日志的格式：

```
<klog header> "<message>" <key1>=<value1> <key2>=<value2> ...
```

示例：

```
I1025 00:15:15.525108    1 controller_utils.go:116] "Pod status updated"  
pod="kube-system/kubedns" status="ready"
```

JSON 日志格式

FEATURE STATE: Kubernetes v1.19 [alpha]

警告：

JSON 输出并不支持太多标准 klog 参数。对于不受支持的 klog 参数的列表，请参见 [命令行工具参考](#)。

并不是所有日志都保证写成 JSON 格式（例如，在进程启动期间）。如果你打算解析日志，请确保可以处理非 JSON 格式的日志行。

字段名和 JSON 序列化可能会发生变化。

--logging-format=json 参数将日志格式从 klog 原生格式改为 JSON 格式。 JSON 日志格式示例（美化输出）：

```
{  
  "ts": 1580306777.04728,  
  "v": 4,  
  "msg": "Pod status updated",  
  "pod": {  
    "name": "nginx-1",  
    "namespace": "default"  
  },  
  "status": "ready"  
}
```

具有特殊意义的 key：

- ts - Unix 时间风格的时间戳（必选项，浮点值）
- v - 精细度（必选项，整数，默认值 0）
- err - 错误字符串（可选项，字符串）
- msg - 消息（必选项，字符串）

当前支持JSON格式的组件列表：

- [kube-controller-manager](#)
- [kube-apiserver](#)
- [kube-scheduler](#)
- [kubelet](#)

日志精细度级别

参数 -v 控制日志的精细度。增大该值会增大日志事件的数量。减小该值可以减小日志事件的数量。增大精细度会记录更多的不太严重的事件。精细度设置为 0 时只记录关键（critical）事件。

日志位置

有两种类型的系统组件：运行在容器中的组件和不运行在容器中的组件。例如：

- Kubernetes 调度器和 kube-proxy 在容器中运行。
- kubelet 和容器运行时，例如 Docker，不在容器中运行。

在使用 systemd 的系统中，kubelet 和容器运行时写入 journald。在别的系统中，日志写入 /var/log 目录下的 .log 文件中。容器中的系统组件总是绕过默认的日志记录机制，写入 /var/log 目录下的 .log 文件。与容器日志类似，你应该轮转 /var/log 目录下系统组件日志。在 kube-up.sh 脚本创建的 Kubernetes 集群中，日志轮转由 logrotate 工具配置。logrotate 工具，每天或者当日志大于 100MB 时，轮转日志。

接下来

- 阅读 [Kubernetes 日志架构](#)
- 阅读 [Structured Logging](#)
- 阅读 [Conventions for logging severity](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 September 21, 2020 at 6:39 PM PST: [\[zh\] addzh/docs/concepts/cluster-administration/system-logs.md, fix 24025 \(b1add796f\)](#)

容器镜像的垃圾收集

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

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）。MaxPerPod Container 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，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 Issue](#) 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
<code>--image-gc-high-threshold</code>	<code>--eviction-hard</code> 或 <code>--eviction-soft</code>	现存的驱逐回收信号可以触发镜像垃圾回收

现存参数	新参数	解释
--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

接下来

查阅[配置资源不足情况的处理](#)了解更多细节。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 22, 2020 at 3:17 AM PST: [\[zh\] sync kubelet-garbage-collection.md \(369b94cad\)](#)

Kubernetes 中的代理

本文讲述了 Kubernetes 中所使用的代理。

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](#) :

- 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

1. [apiserver proxy](#) :

- 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

1. [kube proxy](#) :

- 在每个节点上运行
- 代理 UDP、TCP 和 SCTP
- 不支持 HTTP
- 提供负载均衡能力
- 只用来访问 Service

1. apiserver 之前的代理/负载均衡器：

- 在不同集群中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 API 服务器之间
- 存在多个 API 服务器时，扮演负载均衡器的角色

1. 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 服务类型为 LoadBalancer 时自动创建
- 通常仅支持 UDP/TCP 协议
- SCTP 支持取决于云供应商的负载均衡器实现
- 不同云供应商的云负载均衡器实现不同

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，重定向功能已被弃用。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

API 优先级和公平性

FEATURE STATE: Kubernetes v1.20 [beta]

对于集群管理员来说，控制 Kubernetes API 服务器在过载情况下的行为是一项关键任务。[kube-apiserver](#) 有一些控件（例如：命令行标志 `--max-requests-inflight` 和 `--max-mutating-requests-inflight`），可以限制将要接受的未处理的请求，从而防止过量请求入站，潜在导致 API 服务器崩溃。但是这些标志不足以保证在高流量期间，最重要的请求仍能被服务器接受。

API 优先级和公平性（APF）是一种替代方案，可提升上述最大并发限制。APF 以更细粒度的方式对请求进行分类和隔离。它还引入了空间有限的排队机制，因此在非常短暂的突发情况下，API 服务器不会拒绝任何请求。通过使用公平排队技术从队列中分发请求，这样，一个行为不佳的 [控制器](#) 就不会饿死其他控制器（即使优先级相同）。

注意： 属于“长时间运行”类型的请求（主要是 `watch`）不受 API 优先级和公平性过滤器的约束。如果未启用 APF 特性，即便设置 `--max-requests-inflight` 标志，该类请求也不受约束。

启用/禁用 API 优先级和公平性

API 优先级与公平性（APF）特性由特性门控控制，默认情况下启用。有关特性门控的一般性描述以及如何启用和禁用特性门控，请参见[特性门控](#)。APF 的特性门控称为 API PriorityAndFairness。此特性也与某个 [API 组](#) 相关：(a) 一个 v1alpha1 版本，默认被禁用；(b) 一个 v1beta1 版本，默认被启用。你可以在启动 `kube-apiserver` 时，添加以下命令行标志来禁用此功能门控及 v1beta1 API 组：

```
 kube-apiserver \
--feature-gates=APIPriorityAndFairness=false \
--runtime-config=flowcontrol.apiserver.k8s.io/v1beta1=false \
# ...其他配置不变
```

或者，你也可以通过 `--runtime-config=flowcontrol.apiserver.k8s.io/v1beta1=true` 启用 API 组的 v1alpha1 版本。

命令行标志 `--enable-priority-fairness=false` 将彻底禁用 APF 特性，即使其他标志启用它也是无效。

概念

APF 特性包含几个不同的功能。传入的请求通过 *FlowSchema* 按照其属性分类，并分配优先级。每个优先级维护自定义的并发限制，加强了隔离度，这样不同优先级的请求，就不会相互饿死。在同一个优先级内，公平排队算法可以防止来自不同 *flow* 的请求

相互饿死。该算法将请求排队，通过排队机制，防止在平均负载较低时，通信量突增而导致请求失败。

优先级

如果未启用 APF，API 服务器中的整体并发量将受到 kube-apiserver 的参数 `--max-requests-inflight` 和 `--max-mutating-requests-inflight` 的限制。启用 APF 后，将对这些参数定义的并发限制进行求和，然后将总和分配到一组可配置的 优先级 中。每个传入的请求都会分配一个优先级；每个优先级都有各自的配置，设定允许分发的并发请求数。

例如，默认配置包括针对领导者选举请求、内置控制器请求和 Pod 请求都单独设置优先级。这表示即使异常的 Pod 向 API 服务器发送大量请求，也无法阻止领导者选举或内置控制器的操作执行成功。

排队

即使在同一优先级内，也可能存在大量不同的流量源。在过载情况下，防止一个请求流饿死其他流是非常有价值的（尤其是在一个较为常见的场景中，一个有故障的客户端会疯狂地向 kube-apiserver 发送请求，理想情况下，这个有故障的客户端不应对其他客户端产生太大的影响）。公平排队算法在处理具有相同优先级的请求时，实现了上述场景。每个请求都被分配到某个 流 中，该 流 由对应的 FlowSchema 的名字加上一个 流区分项 (*Flow Distinguisher*) 来标识。这里的流区分项可以是发出请求的用户、目标资源的名称空间或什么都不是。系统尝试为不同流中具有相同优先级的请求赋予近似相等的权重。

将请求划分到流中之后，APF 功能将请求分配到队列中。分配时使用一种称为 [混洗分片 \(Shuffle-Sharding\)](#) 的技术。该技术可以相对有效地利用队列隔离低强度流与高强度流。

排队算法的细节可针对每个优先等级进行调整，并允许管理员在内存占用、公平性（当总流量超标时，各个独立的流将都会取得进展）、突发流量的容忍度以及排队引发的额外延迟之间进行权衡。

豁免请求

某些特别重要的请求不受制于此特性施加的任何限制。这些豁免可防止不当的流控配置完全禁用 API 服务器。

默认值

APF 特性附带推荐配置，该配置对实验场景应该足够；如果你的集群有可能承受较大的负载，那么你应该考虑哪种配置最有效。推荐配置将请求分为五个优先级：

- system 优先级用于 `system:nodes` 组（即 Kubelets）的请求；kubelets 必须能连上 API 服务器，以便工作负载能够调度到其上。
- leader-election 优先级用于内置控制器的领导选举的请求（特别是来自 `kube-system` 名称空间中 `system:kube-controller-manager` 和 `system:kube-`

scheduler 用户和服务账号，针对 endpoints、configmaps 或 leases 的请求）。将这些请求与其他流量相隔离非常重要，因为领导者选举失败会导致控制器发生故障并重新启动，这反过来会导致新启动的控制器在同步信息时，流量开销更大。

- workload-high 优先级用于内置控制器的请求。
- workload-low 优先级适用于来自任何服务帐户的请求，通常包括来自 Pods 中运行的控制器的所有请求。
- global-default 优先级可处理所有其他流量，例如：非特权用户运行的交互式 kubectl 命令。

系统内置了两个 PriorityLevelConfiguration 和两个 FlowSchema，它们是不可重载的：

- 特殊的 exempt 优先级的请求完全不受流控限制：它们总是立刻被分发。特殊的 exempt FlowSchema 把 system:masters 组的所有请求都归入该优先级组。如果合适，你可以定义新的 FlowSchema，将其他请求定向到该优先级。
- 特殊的 catch-all 优先级与特殊的 catch-all FlowSchema 结合使用，以确保每个请求都分类。一般地，你不应该依赖于 catch-all 的配置，而应适当地创建自己的 catch-all FlowSchema 和 PriorityLevelConfigurations（或使用默认安装的 global-default 配置）。为了帮助捕获部分请求未分类的配置错误，强制要求 catch-all 优先级仅允许一个并发份额，并且不对请求进行排队，使得仅与 catch-all FlowSchema 匹配的流量被拒绝的可能性更高，并显示 HTTP 429 错误。

健康检查并发豁免

推荐配置没有为本地 kubelet 对 kube-apiserver 执行健康检查的请求进行任何特殊处理——它们倾向于使用安全端口，但不提供凭据。在推荐配置中，这些请求将分配 global-default FlowSchema 和 global-default 优先级，这样其他流量可以排除健康检查。

如果添加以下 FlowSchema，健康检查请求不受速率限制。

注意：进行此更改后，任何敌对方都可以发送与此 FlowSchema 匹配的任意数量的健康检查请求。如果你有 Web 流量过滤器或类似的外部安全机制保护集群的 API 服务器免受常规网络流量的侵扰，则可以配置规则，阻止所有来自集群外部的健康检查请求。

[priority-and-fairness/health-for-strangers.yaml](#)


```
apiVersion: flowcontrol.apiserver.k8s.io/v1alpha1
kind: FlowSchema
metadata:
  name: health-for-strangers
spec:
  matchingPrecedence: 1000
  priorityLevelConfiguration:
```

```
name: exempt
rules:
- nonResourceRules:
- nonResourceURLs:
  - "/healthz"
  - "/livez"
  - "/readyz"
verbs:
- "*"
subjects:
- kind: Group
  group:
    name: system:unauthenticated
```

资源

流控 API 涉及两种资源。[PriorityLevelConfigurations](#) 定义隔离类型和可处理的并发预算量，还可以微调排队行为。[FlowSchemas](#) 用于对每个入站请求进行分类，并与一个 PriorityLevelConfigurations 相匹配。此外同一 API 组还有一个 v1alpha1 版本，其中包含语法和语义都相同的资源类别。

PriorityLevelConfiguration

一个 PriorityLevelConfiguration 表示单个隔离类型。每个 PriorityLevelConfigurations 对未完成的请求数有各自的限制，对排队中的请求数也有限制。

PriorityLevelConfigurations 的并发限制不是指定请求绝对数量，而是在“并发份额”中指定。API 服务器的总并发量限制通过这些份额按例分配到现有 PriorityLevelConfigurations 中。集群管理员可以更改 --max-requests-inflight (或 --max-mutating-requests-inflight) 的值，再重新启动 kube-apiserver 来增加或减小服务器的总流量，然后所有的 PriorityLevelConfigurations 将看到其最大并发增加 (或减少) 了相同的比例。

注意：启用 APF 功能后，服务器的总并发量限制将设置为 --max-requests-inflight 和 --max-mutating-requests-inflight 之和。可变请求和不可变请求之间不再有任何区别；如果对于某种资源，你需要区别对待不同请求，请创建不同的 FlowSchema 分别匹配可变请求和不可变请求。

当入站请求的数量大于分配的 PriorityLevelConfigurations 中允许的并发级别时，type 字段将确定对额外请求的处理方式。Reject 类型，表示多余的流量将立即被 HTTP 429 (请求过多) 错误所拒绝。Queue 类型，表示对超过阈值的请求进行排队，将使用阈值分片和公平排队技术来平衡请求流之间的进度。

公平排队算法支持通过排队配置对优先级微调。可以在[增强建议](#)中阅读算法的详细信息，但总之：

- queues 递增能减少不同流之间的冲突概率，但代价是增加了内存使用量。值为 1 时，会禁用公平排队逻辑，但仍允许请求排队。
- queueLengthLimit 递增可以在不丢弃任何请求的情况下支撑更大的突发流量，但代价是增加了等待时间和内存使用量。
- 修改 handSize 允许你调整过载情况下不同流之间的冲突概率以及单个流可用的整体并发性。

说明： 较大的 handSize 使两个单独的流程发生碰撞的可能性较小（因此，一个流可以饿死另一个流），但是更有可能的是少数流可以控制 apiserver。较大的 handSize 还可能增加单个高并发流的延迟量。单个流中可能排队的请求的最大数量为 handSize *queueLengthLimit 。

下表显示了有趣的随机分片配置集合，每行显示给定的老鼠（低强度流）被不同数量的大象挤压（高强度流）的概率。表来源请参阅：<https://play.golang.org/p/Gi0PLgVHiUg>

随机分片	队列数	1 个大象	4 个大象	16 个大象
12	32	4.428838398950118e-09	0.11431348830099144	0.9935089607656024
10	32	1.550093439632541e-08	0.0626479840223545	0.9753101519027554
10	64	6.601827268370426e-12	0.00045571320990370776	0.49999929150089345
9	64	3.6310049976037345e-11	0.00045501212304112273	0.4282314876454858
8	64	2.25929199850899e-10	0.0004886697053040446	0.35935114681123076
8	128	6.994461389026097e-13	3.4055790161620863e-06	0.02746173137155063
7	128	1.0579122850901972e-11	6.960839379258192e-06	0.02406157386340147
7	256	7.597695465552631e-14	6.728547142019406e-08	0.0006709661542533682
6	256	2.7134626662687968e-12	2.9516464018476436e-07	0.0008895654642000348
6	512	4.116062922897309e-14	4.982983350480894e-09	2.26025764343413e-05
6	1024	6.337324016514285e-16	8.09060164312957e-11	4.517408062903668e-07

FlowSchema

FlowSchema 匹配一些入站请求，并将它们分配给优先级。每个入站请求都会对所有 FlowSchema 测试是否匹配，首先从 matchingPrecedence 数值最低的匹配开始（我们认为这是逻辑上匹配度最高），然后依次进行，直到首个匹配出现。

注意： 对一个请求来说，只有首个匹配的 FlowSchema 才有意义。如果一个入站请求与多个 FlowSchema 匹配，则将基于 matchingPrecedence 值最高的请求进行筛选。如果一个请求匹配多个 FlowSchema 且 matchingPrecedence 的值相同，则按 name 的字典序选择最小，但是最好不要依赖

它，而是确保不存在两个 FlowSchema 具有相同的 matchingPrecedence 值。

当给定的请求与某个 FlowSchema 的 rules 的其中一条匹配，那么就认为该请求与该 FlowSchema 匹配。判断规则与该请求是否匹配，**不仅要求**该条规则的 subjects 字段至少存在一个与该请求相匹配，**而且要求**该条规则的 resourceRules 或 nonResourceRules（取决于传入请求是针对资源URL还是非资源URL）字段至少存在一个与该请求相匹配。

对于 subjects 中的 name 字段和资源和非资源规则的 verbs，apiGroups，resources，namespaces 和 nonResourceURLs 字段，可以指定通配符 * 来匹配任意值，从而有效地忽略该字段。

FlowSchema 的 distinguisherMethod.type 字段决定了如何把与该模式匹配的请求分散到各个流中。可能是 ByUser，在这种情况下，一个请求用户将无法饿死其他容量的用户；或者是 ByNamespace，在这种情况下，一个名称空间中的资源请求将无法饿死其它名称空间的资源请求；或者它可以为空（或者可以完全省略 distinguisherMethod），在这种情况下，与此 FlowSchema 匹配的请求将被视为单个流的一部分。资源和你的特定环境决定了如何选择正确一个 FlowSchema。

问题诊断

启用了 APF 的 API 服务器，它每个 HTTP 响应都有两个额外的 HTTP 头：X-Kubernetes-PF-FlowSchema-UID 和 X-Kubernetes-PF-PriorityLevel-UID，注意与请求匹配的 FlowSchema 和已分配的优先级。如果请求用户没有查看这些对象的权限，则这些 HTTP 头中将不包含 API 对象的名称，因此在调试时，你可以使用类似如下的命令：

```
kubectl get flowschemas -o custom-columns="uid:{metadata.uid},name:{metadata.name}"
kubectl get prioritylevelconfigurations -o custom-columns="uid:{metadata.uid},name:{metadata.name}"
```

来获取 UID 到 FlowSchema 的名称和 UID 到 PriorityLevelConfigurations 的名称的映射。

可观察性

指标

说明：在 Kubernetes v1.20 之前的版本中，标签 flow_schema 和 priority_level 的命名有时被写作 flowSchema 和 priorityLevel，即存在不一致的情况。如果你在运行 Kubernetes v1.19 或者更早版本，你需要参考你所使用的集群版本对应的文档。

当你开启了 APF 后，kube-apiserver 会暴露额外指标。监视这些指标有助于判断你的配置是否不当地限制了重要流量，或者发现可能会损害系统健康的，行为不良的工作负载。

- `apiserver_flowcontrol_rejected_requests_total` 是一个计数器向量，记录被拒绝的请求数量（自服务器启动以来累计值），由标签 `flow_schema`（表示与请求匹配的 FlowSchema），`priority_level`（表示分配给该请求的优先级）和 `reason` 来区分。`reason` 标签将具有以下值之一：
 - `queue-full`，表明已经有太多请求排队，
 - `concurrency-limit`，表示将 PriorityLevelConfiguration 配置为 Reject 而不是 Queue，或者
 - `time-out`，表示在其排队时间超期的请求仍在队列中。
- `apiserver_flowcontrol_dispatched_requests_total` 是一个计数器向量，记录开始执行的请求数量（自服务器启动以来的累积值），由标签 `flow_schema`（表示与请求匹配的 FlowSchema）和 `priority_level`（表示分配给该请求的优先级）来区分。
- `apiserver_current_inqueue_requests` 是一个表向量，记录最近排队请求数量的高水位线，由标签 `request_kind` 分组，标签的值为 `mutating` 或 `readOnly`。这些高水位线表示在最近一秒内看到的最大数字。它们补充说明了老的表向量 `apiserver_current_inflight_requests`（该量保存了最后一个窗口中，正在处理的请求数量的高水位线）。
- `apiserver_flowcontrol_read_vs_write_request_count_samples` 是一个直方图向量，记录当前请求数量的观察值，由标签 `phase`（取值为 `waiting` 和 `executing`）和 `request_kind`（取值 `mutating` 和 `readOnly`）拆分。定期以高速率观察该值。
- `apiserver_flowcontrol_read_vs_write_request_count_watermarks` 是一个直方图向量，记录请求数量的高/低水位线，由标签 `phase`（取值为 `waiting` 和 `executing`）和 `request_kind`（取值为 `mutating` 和 `readOnly`）拆分；标签 `mark` 取值为 `high` 和 `low`。`apiserver_flowcontrol_read_vs_write_request_count_samples` 向量观察到有值新增，则该向量累积。这些水位线显示了样本值的范围。
- `apiserver_flowcontrol_current_inqueue_requests` 是一个表向量，记录包含排队中的（未执行）请求的瞬时数量，由标签 `priorityLevel` 和 `flowSchema` 拆分。
- `apiserver_flowcontrol_current_executing_requests` 是一个表向量，记录包含执行中（不在队列中等待）请求的瞬时数量，由标签 `priority_level` 和 `flow_schema` 进一步区分。
- `apiserver_flowcontrol_priority_level_request_count_samples` 是一个直方图向量，记录当前请求的观测值，由标签 `phase`（取值为 `waiting` 和 `executing`）和 `p`

`priority_level` 进一步区分。 每个直方图都会定期进行观察，直到相关类别的最后活动为止。观察频率高。

- `apiserver_flowcontrol_priority_level_request_count_watermarks` 是一个直方图向量，记录请求数的高/低水位线，由标签 `phase`（取值为 `waiting` 和 `executing`）和 `priority_level` 拆分；标签 `mark` 取值为 `high` 和 `low`。`apiserver_flowcontrol_priority_level_request_count_samples` 向量观察到有值新增，则该向量累积。这些水位线显示了样本值的范围。
- `apiserver_flowcontrol_request_queue_length_after_enqueue` 是一个直方图向量，记录请求队列的长度，由标签 `priority_level` 和 `flow_schema` 进一步区分。每个排队中的请求都会为其直方图贡献一个样本，并在添加请求后立即上报队列的长度。请注意，这样产生的统计数据与无偏调查不同。

说明： 直方图中的离群值在这里表示单个流（即，一个用户或一个名称空间的请求，具体取决于配置）正在疯狂地向 API 服务器发请求，并受到限制。相反，如果一个优先级的直方图显示该优先级的所有队列都比其他优先级的队列长，则增加 `PriorityLevelConfigurations` 的并发份额是比较合适的。

- `apiserver_flowcontrol_request_concurrency_limit` 是一个表向量，记录并发限制的计算值（基于 API 服务器的总并发限制和 `PriorityLevelConfigurations` 的并发份额），并按标签 `priority_level` 进一步区分。
- `apiserver_flowcontrol_request_wait_duration_seconds` 是一个直方图向量，记录请求排队的时间，由标签 `flow_schema`（表示与请求匹配的 `FlowSchema`），`priority_level`（表示分配该请求的优先级）和 `execute`（表示请求是否开始执行）进一步区分。

说明： 由于每个 `FlowSchema` 总会给请求分配 `PriorityLevelConfigurations`，因此你可以为一个优先级添加所有 `FlowSchema` 的直方图，以获取分配给该优先级的请求的有效直方图。

- `apiserver_flowcontrol_request_execution_seconds` 是一个直方图向量，记录请求实际执行需要花费的时间，由标签 `flow_schema`（表示与请求匹配的 `FlowSchema`）和 `priority_level`（表示分配给该请求的优先级）进一步区分。

调试端点

启用 APF 特性后，`kube-apiserver` 会在其 HTTP/HTTPS 端口提供以下路径：

- `/debug/api_priority_and_fairness/dump_priority_levels` —— 所有优先级及其当前状态的列表。你可以这样获取：

```
kubectl get --raw /debug/api_priority_and_fairness/dump_priority_levels
```

输出类似于：

```
PriorityLevelName, ActiveQueues, IsIdle, IsQuiescing, WaitingRequests,  
ExecutingRequests,  
workload-low, 0, true, false, 0, 0,  
global-default, 0, true, false, 0, 0,  
exempt, <none>, <none>, <none>, <none>, <none>,  
catch-all, 0, true, false, 0, 0,  
system, 0, true, false, 0, 0,  
leader-election, 0, true, false, 0, 0,  
workload-high, 0, true, false, 0, 0,
```

- `/debug/api_priority_and_fairness/dump_queues` —— 所有队列及其当前状态的列表。你可以这样获取：

```
kubectl get --raw /debug/api_priority_and_fairness/dump_queues
```

输出类似于：

```
PriorityLevelName, Index, PendingRequests, ExecutingRequests, VirtualStart,  
workload-high, 0, 0, 0, 0.0000,  
workload-high, 1, 0, 0, 0.0000,  
workload-high, 2, 0, 0, 0.0000,  
...  
leader-election, 14, 0, 0, 0.0000,  
leader-election, 15, 0, 0, 0.0000,
```

- `/debug/api_priority_and_fairness/dump_requests` —— 当前正在队列中等待的所有请求的列表。你可以这样获取：

```
kubectl get --raw /debug/api_priority_and_fairness/dump_requests
```

输出类似于：

```
PriorityLevelName, FlowSchemaName, QueueIndex, RequestIndexInQueue,  
FlowDistingsher, ArriveTime,  
exempt, <none>, <none>, <none>,  
<none>, <none>,  
system, system-nodes, 12, 0, system:node:127.0.0.1,  
2020-07-23T15:26:57.179170694Z,
```

针对每个优先级别，输出中还包含一条虚拟记录，对应豁免限制。

你可以使用以下命令获得更详细的清单：

```
kubectl get --raw '/debug/api_priority_and_fairness/dump_requests?  
includeRequestDetails=1'
```

输出类似于：

```
PriorityLevelName, FlowSchemaName, QueueIndex, RequestIndexInQueue,
FlowDistingsher,      ArriveTime,           UserName,        Verb,
APIPath,                  Namespace, Name, APIVersion,
Resource, SubResource,
system,    system-nodes, 12, 0,          system:node:127.0.0.1,
2020-07-23T15:31:03.583823404Z, system:node:127.0.0.1, create, /api/v1/
namespaces/scaletest/configmaps,
system,    system-nodes, 12, 1,          system:node:127.0.0.1,
2020-07-23T15:31:03.594555947Z, system:node:127.0.0.1, create, /api/v1/
namespaces/scaletest/configmaps,
```

接下来

有关API优先级和公平性的设计细节的背景信息， 请参阅[增强建议](#)。 你可以通过 [SIG API Machinery](#) 提出建议和特性请求。

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 January 10, 2021 at 4:47 PM PST: [\[zh\] Resync concepts/cluster-administration/flow-control.md \(10fccce49\)](#)

安装扩展 (Addons)

注意：本部分链接到提供 Kubernetes 所需功能的第三方项目。

Kubernetes 项目作者不负责这些项目。此页面遵循[CNCF 网站指南](#)，按字母顺序列出项目。要将项目添加到此列表中，请在提交更改之前阅读[内容指南](#)。

。

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

- [ACI](#) 通过 Cisco ACI 提供集成的容器网络和安全网络。
- [Calico](#) 是一个安全的 L3 网络和网络策略驱动。
- [Canal](#) 结合 Flannel 和 Calico，提供网络和网络策略。

- [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) 和覆盖/封装 (overlay/encapsulation) 模式。
- [CNI-Genie](#) 使 Kubernetes 无缝连接到一种 CNI 插件，例如：Flannel、Calico、Canal、Romana 或者 Weave。
- [Contiv](#) 为多种用例提供可配置网络（使用 BGP 的原生 L3，使用 vxlan 的覆盖网络，经典 L2 和 Cisco-SDN/ACI）和丰富的策略框架。Contiv 项目完全[开源](#)。[安装工具](#)同时提供基于和不基于 kubeadm 的安装选项。
- 基于 [Tungsten Fabric](#) 的 [Contrail](#) 是一个开源的多云网络虚拟化和策略管理平台，Contrail 和 Tungsten Fabric 与业务流程系统（例如 Kubernetes、OpenShift、OpenStack 和 Mesos）集成在一起，为虚拟机、容器或 Pod 以及裸机工作负载提供了隔离模式。
- [Flannel](#) 是一个可以用于 Kubernetes 的 overlay 网络提供者。
- [Knitter](#) 是为 kubernetes 提供复合网络解决方案的网络组件。
- [Multus](#) 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），而且包含了在 Kubernetes 中基于 SRIOV、DPDK、OVS-DPDK 和 VPP 的工作负载。
- [OVN-Kubernetes](#) 是一个 Kubernetes 网络驱动，基于 [OVN \(Open Virtual Network \)](#) 实现，是从 Open vSwitch (OVS) 项目衍生出来的虚拟网络实现。OVN-Kubernetes 为 Kubernetes 提供基于覆盖网络的网络实现，包括一个基于 OVS 实现的负载均衡器 和 网络策略。
- [OVN4NFV-K8S-Plugin](#) 是一个基于 OVN 的 CNI 控制器插件，提供基于云原生的服务功能链条（Service Function Chaining，SFC）、多种 OVN 覆盖网络、动态子网创建、动态虚拟网络创建、VLAN 驱动网络、直接驱动网络，并且可以 驳接其他的多网络插件，适用于基于边缘的、多集群联网的云原生工作负载。
- [NSX-T](#) 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](#) 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](#) 是一个 pod 网络的第三层解决方案，并支持 [NetworkPolicy API](#)。Kubeadm add-on 安装细节可以在[这里](#)找到。
- [Weave Net](#) 提供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](#) 是一种灵活的，可扩展的 DNS 服务器，可以 [安装](#)为集群内的 Pod 提供 DNS 服务。

可视化管理

- [Dashboard](#) 是一个 Kubernetes 的 Web 控制台界面。
- [Weave Scope](#) 是一个图形化工具，用于查看你的容器、Pod、服务等。请和一个 [Weave Cloud 账号](#) 一起使用，或者自己运行 UI。

基础设施

- [KubeVirt](#) 是可以让 Kubernetes 运行虚拟机的 add-ons。通常运行在裸机集群上。

遗留 Add-ons

还有一些其它 add-ons 归档在已废弃的 [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

此页是否对您有帮助 ?

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 12, 2020 at 3:32 PM PST: [\[zh\] Sync changes from English site \(1\) \(9dbaf53f8\)](#)

扩展 Kubernetes

改变你的 Kubernetes 集群的行为的若干方法。

Kubernetes 是高度可配置且可扩展的。因此，大多数情况下，你不需要 派生自己的 Kubernetes 副本或者向项目代码提交补丁。

本指南描述定制 Kubernetes 的可选方式。主要针对的读者是希望了解如何针对自身工作环境 需要来调整 Kubernetes 的**集群管理者**。对于那些充当**平台开发人员** 的开发人员或 Kubernetes 项目的**贡献者** 而言，他们也会在本指南中找到有用的介绍信息，了解系统中存在哪些扩展点和扩展模式，以及它们所附带的各种权衡和约束等等。

概述

定制化的方法主要可分为 **配置 (Configuration)** 和 **扩展 (Extensions)** 两种。前者主要涉及改变参数标志、本地配置文件或者 API 资源；后者则需要额外运行一些程序或服务。本文主要关注扩展。

Configuration

配置文件和参数标志的说明位于在线文档的参考章节，按可执行文件组织：

- [kubelet](#)
- [kube-apiserver](#)

- [kube-controller-manager](#)
- [kube-scheduler](#).

在托管的 Kubernetes 服务中或者受控安装的发行版本中，参数标志和配置文件不总是可以修改的。即使它们是可修改的，通常其修改权限也仅限于集群管理员。此外，这些内容在将来的 Kubernetes 版本中很可能发生变化，设置新参数或配置文件可能需要重启进程。有鉴于此，通常应该在没有其他替代方案时才应考虑更改参数标志和配置文件。

内置的策略 API，例如[ResourceQuota](#)、[PodSecurityPolicies](#)、[NetworkPolicy](#)和基于角色的访问控制（[RBAC](#)）等等都是内置的 Kubernetes API。API 通常用于托管的 Kubernetes 服务和受控的 Kubernetes 安装环境中。这些 API 是声明式的，与 Pod 这类其他 Kubernetes 资源遵从相同的约定，所以新的集群配置是可复用的，并且可以当作应用程序来管理。此外，对于稳定版本的 API 而言，它们与其他 Kubernetes API 一样，采纳的是一种[预定义的支持策略](#)。出于以上原因，在条件允许的情况下，基于 API 的方案应该优先于配置文件和参数标志。

扩展

扩展（Extensions）是一些扩充 Kubernetes 能力并与之深度集成的软件组件。它们调整 Kubernetes 的工作方式使之支持新的类型和新的硬件种类。

大多数集群管理员会使用一种托管的 Kubernetes 服务或者其某种发行版本。因此，大多数 Kubernetes 用户不需要安装扩展，至于需要自己编写新的扩展的情况就更少了。

扩展模式

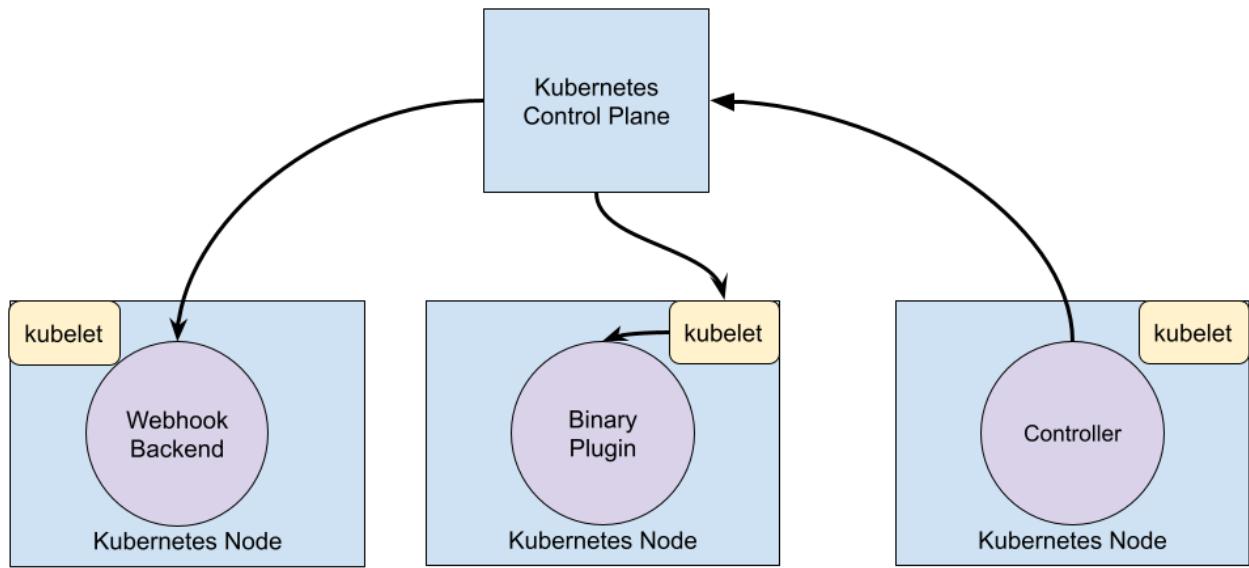
Kubernetes 从设计上即支持通过编写客户端程序来将其操作自动化。任何能够对 Kubernetes API 发出读写指令的程序都可以提供有用自动化能力。自动化组件可以运行在集群上，也可以运行在集群之外。通过遵从本文中的指南，你可以编写高度可用的、运行稳定的自动化组件。自动化组件通常可以用于所有 Kubernetes 集群，包括托管的集群和受控的安装环境。

编写客户端程序有一种特殊的*Controller（控制器）*模式，能够与 Kubernetes 很好地协同工作。控制器通常会读取某个对象的 `.spec`，或许还会执行一些操作，之后更新对象的 `.status`。

控制器是 Kubernetes 的客户端。当 Kubernetes 充当客户端，调用某远程服务时，对应的远程组件称作 *Webhook*。远程服务称作 *Webhook 后端*。与控制器模式相似，*Webhook* 也会在整个架构中引入新的失效点（Point of Failure）。

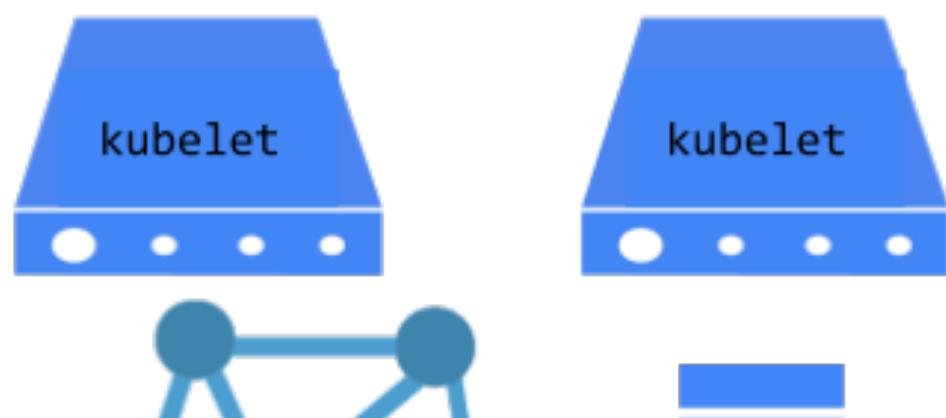
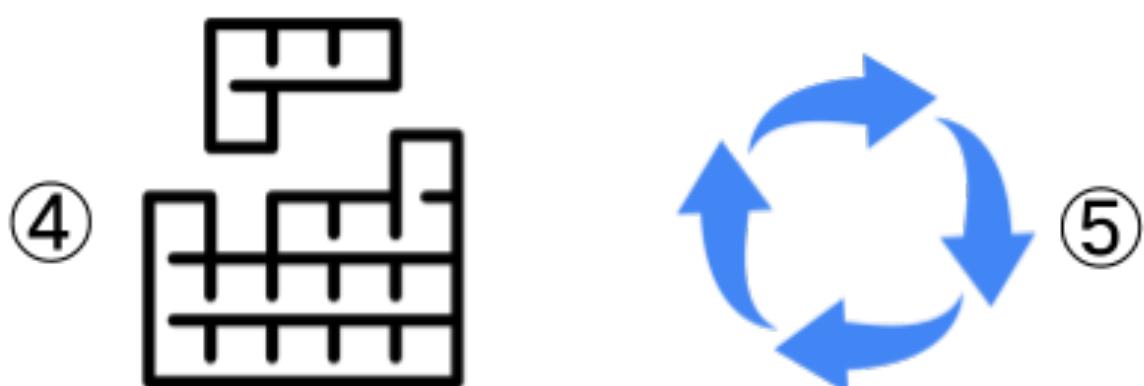
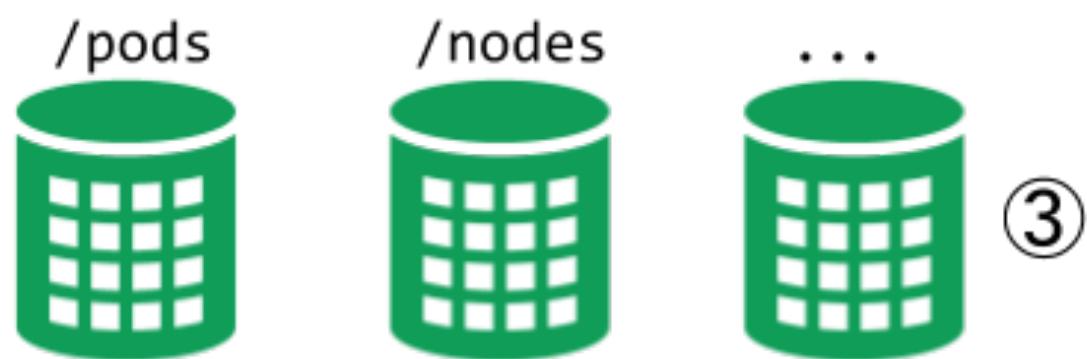
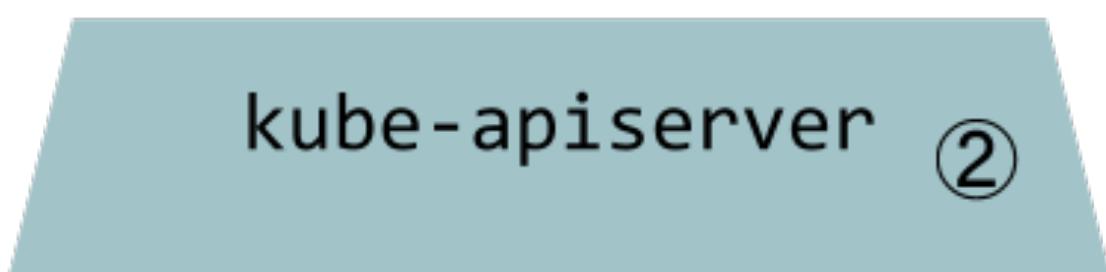
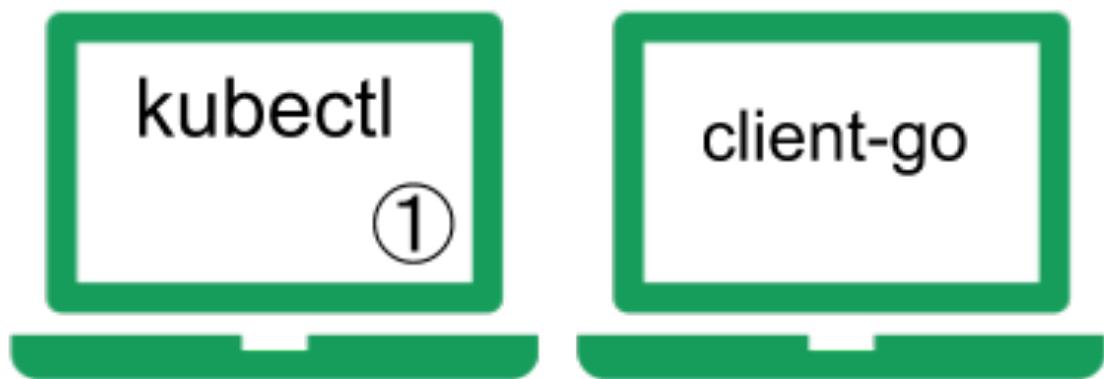
在 *Webhook* 模式中，Kubernetes 向远程服务发起网络请求。在*可执行文件插件（Binary Plugin）*模式中，Kubernetes 执行某个可执行文件（程序）。可执行文件插件在 `kubelet`（例如，[FlexVolume 插件](#) 和[网络插件](#)）和 `kubectl` 中使用。

下面的示意图中展示了这些扩展点如何与 Kubernetes 控制面交互。



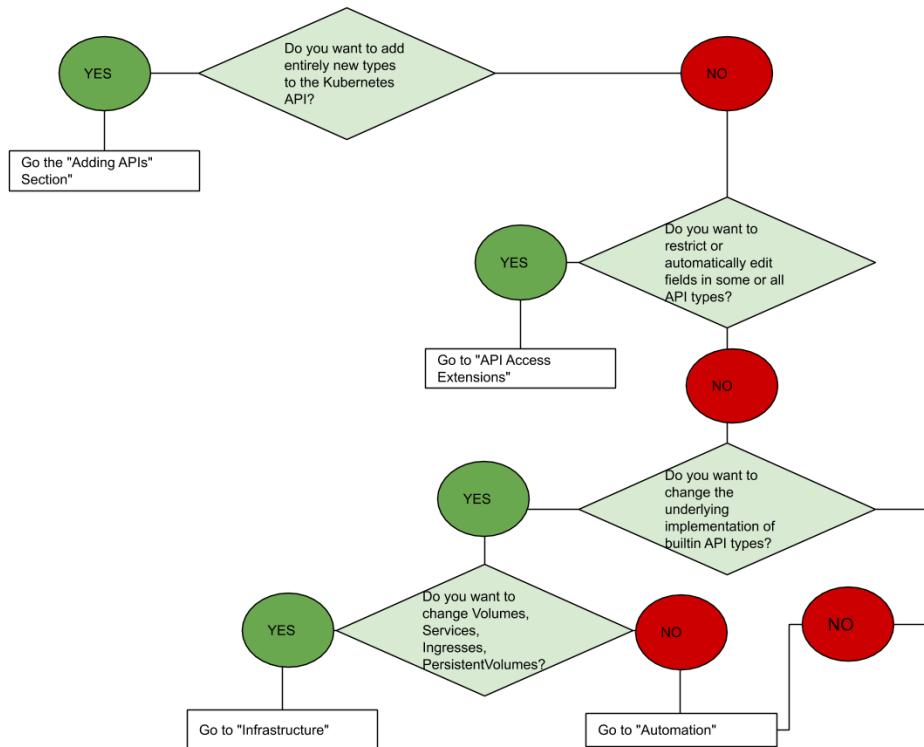
扩展点

此示意图显示的是 Kubernetes 系统中的扩展点。



- 用户通常使用 kubectl 与 Kubernetes API 交互。 [kubectl 插件](#)能够扩展 kubectl 程序的行为。这些插件只会影响到每个用户的本地环境，因此无法用来强制实施整个站点范围的策略。
2. API 服务器处理所有请求。API 服务器中的几种扩展点能够使用户对请求执行身份认证、基于其内容阻止请求、编辑请求内容、处理删除操作等等。这些扩展点在 [API 访问扩展](#) 节详述。
 3. API 服务器向外提供不同类型的资源 (*resources*)。内置的资源类型，如 pods，是由 Kubernetes 项目所定义的，无法改变。你也可以添加自己定义的或者其他项目所定义的称作自定义资源 (*Custom Resources*) 的资源，正如[自定义资源](#)节所描述的那样。自定义资源通常与 API 访问扩展点结合使用。
 4. Kubernetes 调度器负责决定 Pod 要放置到哪些节点上执行。有几种方式来扩展调度行为。这些方法将在 [调度器扩展](#) 节中展开。
 5. Kubernetes 中的很多行为都是通过称为控制器 (Controllers) 的程序来实现的，这些程序也都是 API 服务器的客户端。控制器常常与自定义资源结合使用。
 6. 组件 kubelet 运行在各个节点上，帮助 Pod 展现为虚拟的服务器并在集群网络中拥有自己的 IP。[网络插件](#)使得 Kubernetes 能够采用不同实现技术来连接 Pod 网络。
 7. 组件 kubelet 也会为容器增加或解除存储卷的挂载。通过[存储插件](#)，可以支持新的存储类型。

如果你无法确定从何处入手，下面的流程图可能对你有些帮助。注意，某些方案可能需要同时采用几种类型的扩展。



API 扩展

用户定义的类型

如果你想要定义新的控制器、应用配置对象或者其他声明式 API，并且使用 Kubernetes 工具（如 kubectl）来管理它们，可以考虑向 Kubernetes 添加自定义资源。

不要使用自定义资源来充当应用、用户或者监控数据的数据存储。

关于自定义资源的更多信息，可参见[自定义资源概念指南](#)。

结合使用新 API 与自动化组件

自定义资源 API 与控制回路的组合称作 [Operator 模式](#)。Operator 模式用来管理特定的、通常是有状态的应用。这些自定义 API 和控制回路也可用来控制其他资源，如存储或策略。

更改内置资源

当你通过添加自定义资源来扩展 Kubernetes 时，所添加的资源通常会被放在一个新的 API 组中。你不可以替换或更改现有的 API 组。添加新的 API 不会直接让你影响现有 API（如 Pods）的行为，不过 API 访问扩展能够实现这点。

API 访问扩展

当请求到达 Kubernetes API 服务器时，首先要经过身份认证，之后是鉴权操作，再之后要经过若干类型的准入控制器的检查。参见[控制 Kubernetes API 访问](#)以了解此流程的细节。

这些步骤中都存在扩展点。

Kubernetes 提供若干内置的身份认证方法。它也可以运行在某中身份认证代理的后面，并且可以将来自鉴权头部的令牌发送到某个远程服务（Webhook）来执行验证操作。所有这些方法都在[身份认证文档](#)中有详细论述。

身份认证

[身份认证](#)负责将所有请求中的头部或证书映射到发出该请求的客户端的用户名。

Kubernetes 提供若干种内置的认证方法，以及[认证 Webhook](#)方法以备内置方法无法满足你的要求。

鉴权

[鉴权](#)操作负责确定特定的用户是否可以读、写 API 资源或对其执行其他操作。此操作仅在整个资源集合的层面进行。换言之，它不会基于对象的特定字段作出不同的判决。如果内置的鉴权选项无法满足你的需要，你可以使用[鉴权 Webhook](#)来调用用户提供 的代码，执行定制的鉴权操作。

动态准入控制

请求的鉴权操作结束之后，如果请求的是写操作，还会经过[准入控制](#)处理步骤。除了内置的处理步骤，还存在一些扩展点：

- [Image Policy webhook](#) 能够限制容器中可以运行哪些镜像。
- 为了执行任意的准入控制，可以使用一种通用的[Admission webhook](#)机制。这类 Webhook 可以拒绝对象创建或更新请求。

基础设施扩展

存储插件

[FlexVolumes](#) 卷可以让用户挂载无需内建支持的卷类型，kubelet 会调用可执行文件插件来挂载对应的存储卷。

设备插件

使用[设备插件](#)，节点能够发现新的节点资源（除了内置的类似 CPU 和内存这类资源）。

网络插件

通过节点层面的[网络插件](#)，可以支持不同的网络设施。

调度器扩展

调度器是一种特殊的控制器，负责监视 Pod 变化并将 Pod 分派给节点。默认的调度器可以被整体替换掉，同时继续使用其他 Kubernetes 组件。或者也可以在同一时刻使用 [多个调度器](#)。

这是一项非同小可的任务，几乎绝大多数 Kubernetes 用户都会发现其实他们不需要修改调度器。

调度器也支持一种 [Webhook](#)，允许使用某种 Webhook 后端（调度器扩展）来为 Pod 可选的节点执行过滤和优先排序操作。

接下来

- 进一步了解[自定义资源](#)
- 了解[动态准入控制](#)
- 进一步了解基础设施扩展
 - [网络插件](#)
 - [设备插件](#)
- 了解 [kubectl 插件](#)
- 了解 [Operator 模式](#)

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。本文档的目标读者包括希望了解如何使 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](#)。或潜在的[平台开发人员](#) 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

定制方法可以大致分为 [配置](#) (*Configuration*) 和 [扩展](#) (*Extension*)。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，按照可执行文件组织：

- [kubelet](#)
- [kube-apiserver](#)
- [kube-controller-manager](#)
- [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](#)、[PodSecurityPolicy](#)、[NetworkPolicy](#) 和基于角色的权限控制 ([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，当它们变稳定后，也遵循和其他 Kubernetes API 一样的 [支持政策](#)。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

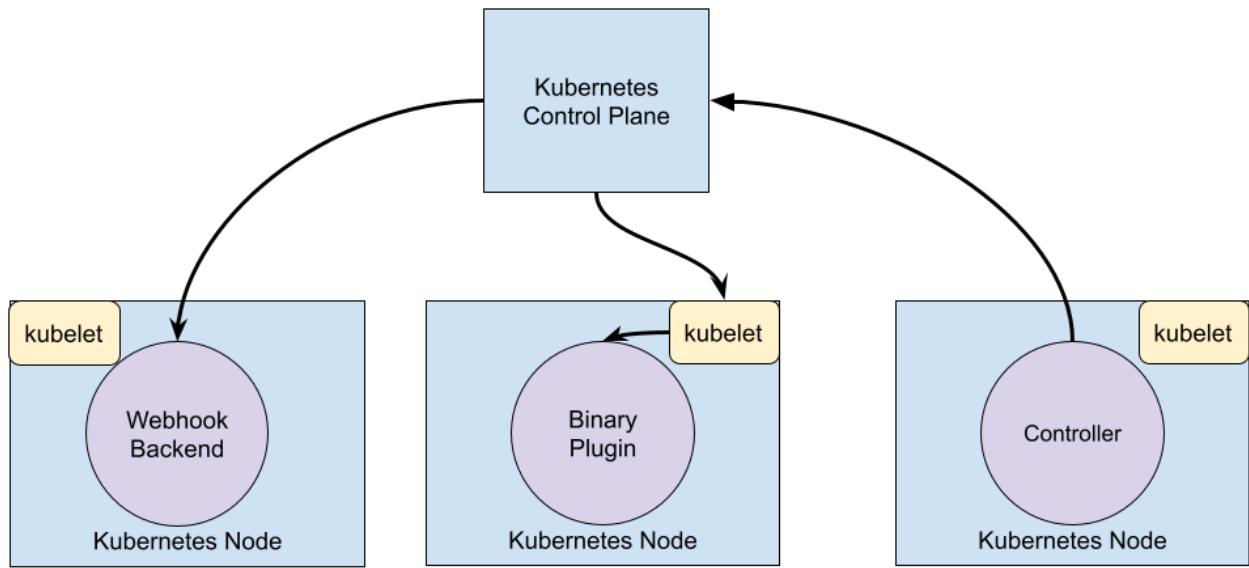
Kubernetes 的设计是通过编写客户端程序来实现自动化的。任何读和（或）写 Kubernetes API 的程序都可以提供有用的自动化工作。自动化程序可以运行在集群之中或之外。按照本文档的指导，你可以编写出高可用的和健壮的自动化程序。自动化程序通常适用于任何 Kubernetes 集群，包括托管集群和受管理安装的集群。

控制器（Controller）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。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。

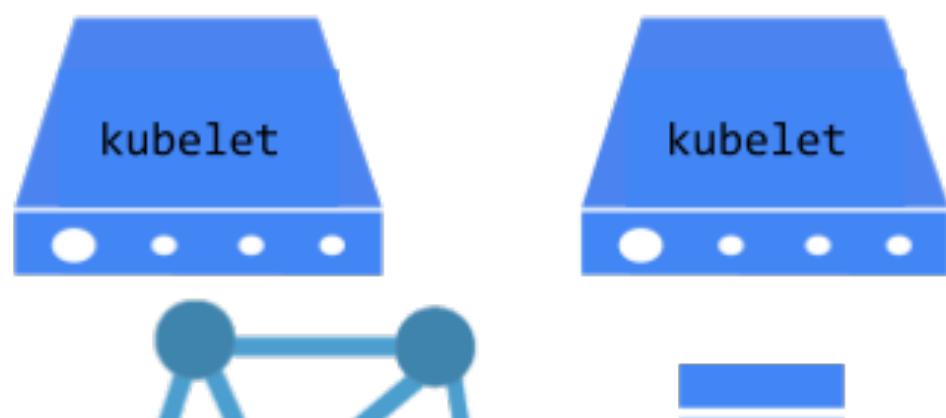
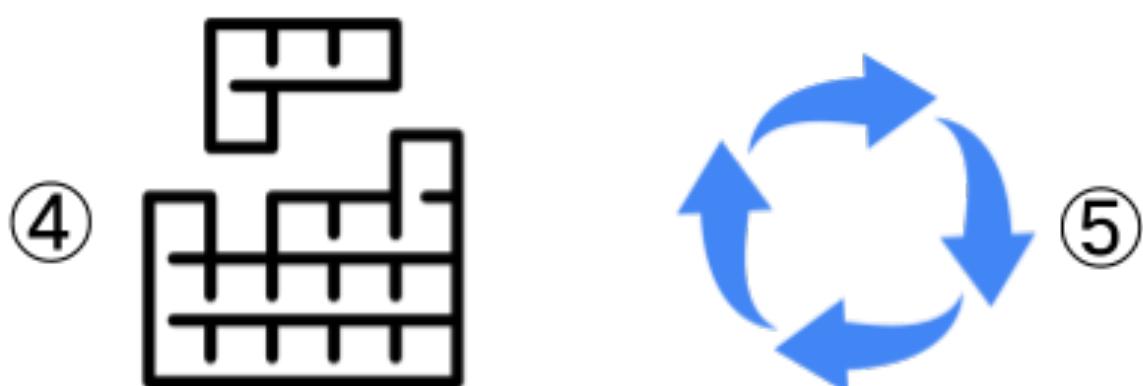
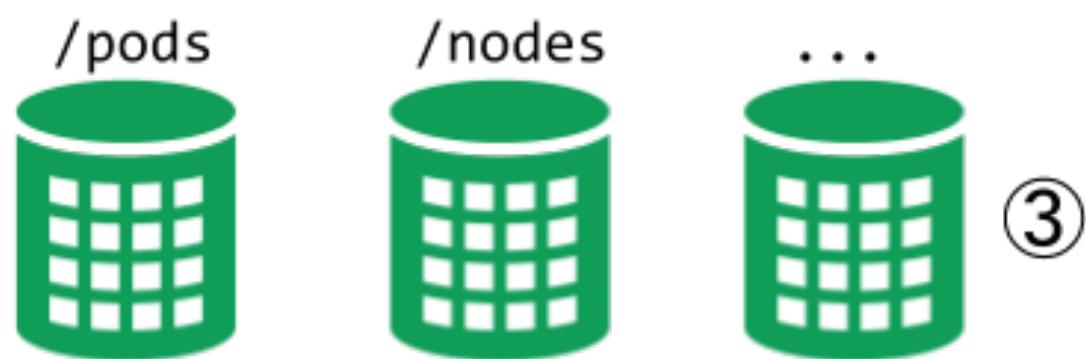
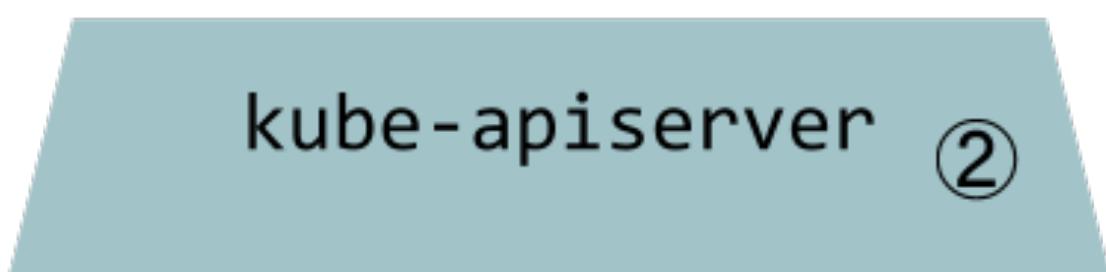
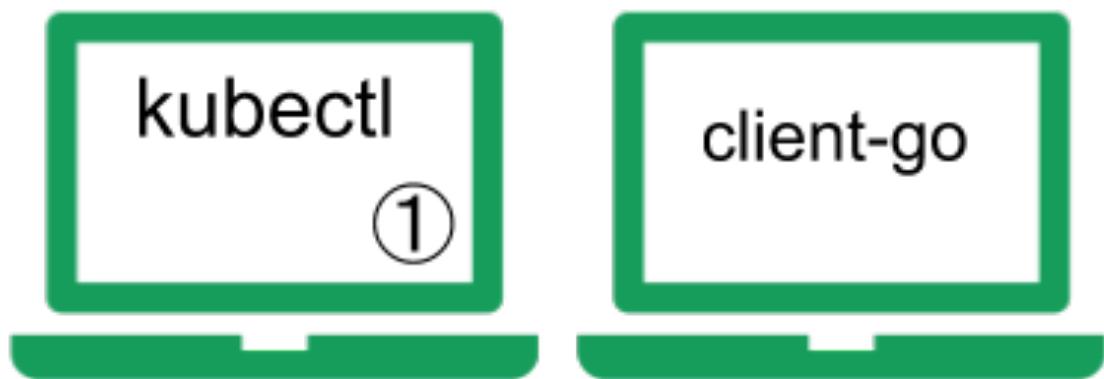
在 webhook 模型里，Kubernetes 向远程服务发送一个网络请求。在 可执行文件插件 模型里，Kubernetes 执行一个可执行文件（程序）。可执行文件插件被 kubelet（如 [Flex 卷插件](#) 和 [网络插件](#)）和 kubectl 所使用。

下图显示了扩展点如何与 Kubernetes 控制平面进行交互。



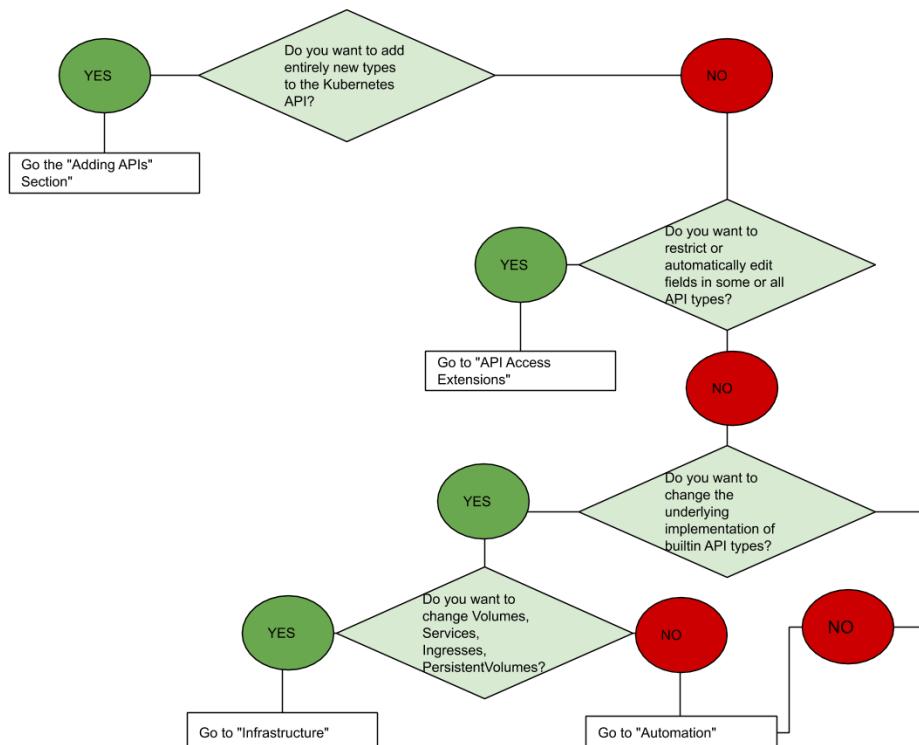
扩展点

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](#)扩展了 `kubectl` 可执行文件。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. API 服务器处理所有请求。API 服务器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](#) 小节中描述。
3. API 服务器提供各种 资源 (*Resource*)。内置的资源种类 (*Resource Kinds*)，如 `pods`，由 Kubernetes 项目定义，不能更改。你还可以添加你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** (*Custom Resource*)，如[自定义资源](#) 部分所述。自定义资源通常与 API 访问扩展一起使用。
4. 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[调度器扩展](#) 小节中描述。
5. Kubernetes 的大部分行为都是由称为控制器 (Controllers) 的程序实现的，这些程序是 API 服务器的客户端。控制器通常与自定义资源一起使用。
6. `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](#)让你可以实现不同的 pod 网络。
7. `kubelet` 也负责为容器挂载和卸载卷。新的存储类型可以通过[存储插件](#)支持。

如果你不确定从哪里开始扩展，下面流程图可以提供一些帮助。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

如果你想定义新的控制器、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](#)。

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（Operator）模式](#)。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

当你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。你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](#)。

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 Authorization 头部中的令牌发送给远程服务（webhook）进行验证。所有这些方法都在 [身份验证文档](#) 中介绍。

身份认证

[身份认证](#) 将所有请求中的头部字段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你的需求，可以使用 [身份认证 Webhook](#) 方法。

鉴权

[鉴权组件](#) 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你的需求，[鉴权 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](#) 步骤。除了内置的步骤之外，还有几个扩展：

- [镜像策略 Webhook](#) 限制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 [准入 Webhook](#)。准入 Webhooks 可以拒绝创建或更新操作。

基础设施扩展

存储插件

[Flex Volumes](#) 允许用户挂载无内置插件支持的卷类型，它通过 Kubelet 调用一个可执行文件插件来挂载卷。

设备插件

设备插件允许节点通过 [设备插件](#)，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](#) 得到支持。

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](#)。

这是一个不太轻松的任务，几乎所有的 Kubernetes 用户都会意识到他们并不需要修改调度器。

调度器也支持 [Webhook](#)，它允许使用一个 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。

接下来

- 详细了解[自定义资源](#)
- 了解[动态准入控制](#)
- 详细了解基础设施扩展
 - [网络插件](#)
 - [设备插件](#)
- 了解 [kubectl 插件](#)
- 了解[操作者模式](#)

反馈

此页是否对您有帮助？

是否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

扩展 Kubernetes API

[定制资源](#)

[通过聚合层扩展 Kubernetes API](#)

定制资源

定制资源 (*Custom Resource*) 是对 Kubernetes API 的扩展。本页讨论何时向 Kubernetes 集群添加定制资源，何时使用独立的服务。本页描述添加定制资源的两种方法以及怎样在二者之间做出抉择。

定制资源

资源 (*Resource*) 是 [Kubernetes API](#) 中的一个端点，其中存储的是某个类别的 [API 对象](#) 的一个集合。例如内置的 *pods* 资源包含一组 Pod 对象。

定制资源 (*Custom Resource*) 是对 Kubernetes API 的扩展，不一定在默认的 Kubernetes 安装中就可用。定制资源所代表的是对特定 Kubernetes 安装的一种定制。不过，很多 Kubernetes 核心功能现在都用定制资源来实现，这使得 Kubernetes 更加模块化。

定制资源可以通过动态注册的方式在运行中的集群内或出现或消失，集群管理员可以独立于集群更新定制资源。一旦某定制资源被安装，用户可以使用 [kubectl](#) 来创建和访问其中的对象，就像他们为 *pods* 这种内置资源所做的一样。

定制控制器

就定制资源本身而言，它只能用来存取结构化的数据。当你将定制资源与 [定制控制器](#) (*Custom Controller*) 相结合时，定制资源就能够 提供真正的 [声明式 API](#) (*Declarative API*)。

使用[声明式 API](#)，你可以 声明 或者设定你的资源的期望状态，并尝试让 Kubernetes 对象的当前状态 同步到其期望状态。控制器负责将结构化的数据解释为用户所期望状态的记录，并 持续地维护该状态。

你可以在一个运行中的集群上部署和更新定制控制器，这类操作与集群的生命周期无关。定制控制器可以用于任何类别的资源，不过它们与定制资源结合起来时最为有效。

[Operator 模式](#)就是将定制资源与定制控制器相结合的。你可以使用定制控制器来将特定于某应用的领域知识组织起来，以编码的形式构造对 Kubernetes API 的扩展。

我是否应该向我的 Kubernetes 集群添加定制资源？

在创建新的 API 时，请考虑是 [将你的 API 与 Kubernetes 集群 API 聚合起来](#) 还是让你的 API 独立运行。

考虑 API 聚合的情况	优选独立 API 的情况
你的 API 是声明式的。	你的 API 不符合声明式模型。
你希望可以是使用 kubectl 来读写你的新资源类别。	不要求 kubectl 支持。
你希望在 Kubernetes UI（如仪表板）中和其他内置类别一起查看你的新资源类别。	不需要 Kubernetes UI 支持。
你在开发新的 API。	你已经有一个提供 API 服务的程序并且工作良好。
你有意愿接受 Kubernetes 对 REST 资源路径所作的格式限制，例如 API 组和名字空间。 (参阅 API 概述)	你需要使用一些特殊的 REST 路径以便与已经定义的 REST API 保持兼容。
你的资源可以自然地界定为集群作用域或集群中某个名字空间作用域。	集群作用域或名字空间作用域这种二分法很不合适；你需要对资源路径的细节进行控制。
你希望复用 Kubernetes API 支持特性 。	你不需要这类特性。

声明式 APIs

典型地，在声明式 API 中：

- 你的 API 包含相对而言为数不多的、尺寸较小的对象（资源）。
- 对象定义了应用或者基础设施的配置信息。
- 对象更新操作频率较低。
- 通常需要人来读取或写入对象。
- 对象的主要操作是 CRUD 风格的（创建、读取、更新和删除）。
- 不需要跨对象的事务支持：API 对象代表的是期望状态而非确切实际状态。

命令式 API (Imperative API) 与声明式有所不同。以下迹象表明你的 API 可能不是声明式的：

- 客户端发出“做这个操作”的指令，之后在该操作结束时获得同步响应。
- 客户端发出“做这个操作”的指令，并获得一个操作 ID，之后需要检查一个 Operation（操作）对象来判断请求是否成功完成。
- 你会将你的 API 类比为远程过程调用 (Remote Procedure Call , RPCs) 。
- 直接存储大量数据；例如每个对象几 kB，或者存储上千个对象。
- 需要较高的访问带宽（长期保持每秒数十个请求）。
- 存储有应用来处理的最终用户数据（如图片、个人标识信息（PII）等）或者其他大规模数据。
- 在对象上执行的常规操作并非 CRUD 风格。
- API 不太容易用对象来建模。

- 你决定使用操作 ID 或者操作对象来表现悬决的操作。

我应该使用一个 ConfigMap 还是一个定制资源？

如果满足以下条件之一，应该使用 ConfigMap：

- 存在一个已有的、文档完备的配置文件格式约定，例如 mysql.cnf 或 pom.xml。
- 你希望将整个配置文件放到某 configMap 中的一个主键下面。
- 配置文件的主要用途是针对运行在集群中 Pod 内的程序，供后者依据文件数据配置自身行为。
- 文件的使用者期望以 Pod 内文件或者 Pod 内环境变量的形式来使用文件数据，而不是通过 Kubernetes API。
- 你希望当文件被更新时通过类似 Deployment 之类的资源完成滚动更新操作。

说明：请使用 [Secret](#) 来保存敏感数据。 Secret 类似于 configMap，但更为安全。

如果以下条件中大多数都被满足，你应该使用定制资源（CRD 或者 聚合 API）：

- 你希望使用 Kubernetes 客户端库和 CLI 来创建和更改新的资源。
- 你希望 kubectl 能够直接支持你的资源；例如，kubectl get my-object object-name。
- 你希望构造新的自动化机制，监测新对象上的更新事件，并对其他对象执行 CRUD 操作，或者监测后者更新前者。
- 你希望编写自动化组件来处理对对象的更新。
- 你希望使用 Kubernetes API 对诸如 .spec、.status 和 .metadata 等字段的约定。
- 你希望对象是对一组受控资源的抽象，或者对其他资源的归纳提炼。

添加定制资源

Kubernetes 提供了两种方式供你向集群中添加定制资源：

- CRD 相对简单，创建 CRD 可以不必编程。
- [API 聚合](#) 需要编程，但支持对 API 行为进行更多的控制，例如数据如何存储以及在不同 API 版本间如何转换等。

Kubernetes 提供这两种选项以满足不同用户的需求，这样就既不会牺牲易用性也不会牺牲灵活性。

聚合 API 指的是一些下位的 API 服务器，运行在主 API 服务器后面；主 API 服务器以代理的方式工作。这种组织形式称作 [API 聚合 \(API Aggregation , AA \)](#)。对用户而言，看起来仅仅是 Kubernetes API 被扩展了。

CRD 允许用户创建新的资源类别同时又不必添加新的 API 服务器。 使用 CRD 时，你并不需要理解 API 聚合。

无论以哪种方式安装定制资源，新的资源都会被当做定制资源，以便与内置的 Kubernetes 资源（如 Pods）相区分。

CustomResourceDefinitions

[CustomResourceDefinition](#) API 资源允许你定义定制资源。 定义 CRD 对象的操作会使用你所设定的名字和模式定义 (Schema) 创建一个新的定制资源， Kubernetes API 负责为你的定制资源提供存储和访问服务。 CRD 对象的名称必须是合法的 [DNS 子域名](#)。

CRD 使得你不必编写自己的 API 服务器来处理定制资源，不过其背后实现的通用性也意味着 你所获得的灵活性要比 [API 服务器聚合](#) 少很多。

关于如何注册新的定制资源、 使用新资源类别的实例以及如何使用控制器来处理事件，相关的例子可参见[定制控制器示例](#)。

API 服务器聚合

通常，Kubernetes API 中的每个都需要处理 REST 请求和管理对象持久性存储的代码。 Kubernetes API 主服务器能够处理诸如 *pods* 和 *services* 这些内置资源，也可以 按通用的方式通过 CRD {#customresourcedefinitions} 来处理定制资源。

[聚合层 \(Aggregation Layer \)](#) 使得你可以通过编写和部署你自己的独立的 API 服务器来为定制资源提供特殊的实现。 主 API 服务器将针对你要处理的定制资源的请求全部委托给你来处理，同时将这些资源 提供给其所有客户。

选择添加定制资源的方法

CRD 更为易用；聚合 API 则更为灵活。请选择最符合你的需要的方法。

通常，如何存在以下情况，CRD 可能更合适：

- 定制资源的字段不多；
- 你在组织内部使用该资源或者在一个小规模的开源项目中使用该资源，而不是 在商业产品中使用。

比较易用性

CRD 比聚合 API 更容易创建

CRDs	聚合 API
无需编程。 用户可选择任何语言来实现 CRD 控制器。	需要使用 Go 来编程，并构建可执行文件和镜像。
无需额外运行服务；CRD 由 API 服务器处理。	需要额外创建服务，且该服务可能失效。
一旦 CRD 被创建，不需要持续提供支持。 Kubernetes 主控节点升级过程中自动会带入缺陷修复。	可能需要周期性地从上游提取缺陷修复并更新聚合 API 服务器。
无需处理 API 的多个版本；例如，当你控制资源的客户端时，你可以更新它使之与 API 同步。	你需要处理 API 的多个版本；例如，在开发打算与很多人共享的扩展时。

高级特性与灵活性

聚合 API 可提供更多的高级 API 特性，也可对其他特性实行定制；例如，对存储层进行定制。

特性	描述	CRDs	聚合 API
合法性检查	帮助用户避免错误，允许你独立于客户端版本演化 API。这些特性对于由很多无法同时更新的客户端的场合。	可以。大多数验证可以使用 OpenAPI v3.0 合法性检查 来设定。其他合法性检查操作可以通过添加 合法性检查 Webhook 来实现。	可以，可执行任何合法性检查。
默认值设置	同上	可以。可通过 OpenAPI v3.0 合法性检查 的 default 关键词（自 1.17 正式发布）或 更改性 (Mutating) Webhook 来实现（不过从 etcd 中读取老的对象时不会执行这些 Webhook）。	可以。
多版本支持	允许通过两个 API 版本同时提供同一对象。可帮助简化类似字段更名这类 API 操作。如果你能控制客户端版本，这一特性将不再重要。	可以。	可以。
定制存储	支持使用具有不同性能模式的存储（例如，要使用时间序列数据库而不是键值存储），或者因安全性原因对存储进行隔离（例如对敏感信息执行加密）。	不可以。	可以。
定制业务逻辑	在创建、读取、更新或删除对象时，执行任意的检查或操作。	可以。要使用 Webhook 。	可以。
支持 scale 子资源	允许 HorizontalPodAutoscaler 和 PodDisruptionBudget 这类子系统与你的新资源交互。	可以。	可以。
支持 status 子资源	允许在用户写入 spec 部分而控制器写入 status 部分时执行细粒度的访问控制。允许在对定制资源的数据进行更改时增加对象的代际（Generation）；这需要资源对 spec 和 status 部分有明确划分。	可以。	可以。
其他子资源	添加 CRUD 之外的操作，例如 "logs" 或 "exec"。	不可以。	可以。

特性	描述	CRDs	聚合 API
strategic-merge-patch	新的端点要支持标记了 Content-Type: application/strategic-merge-patch+json 的 PATCH 操作。对于更新既可在本地更改也可在服务器端更改的对象而言是有用的。要了解更多信息，可参见 使用 kubectl patch 来更新 API 对象 。	不可以。	可以。
支持协议缓冲区	新的资源要支持想要使用协议缓冲区 (Protocol Buffer) 的客户端。	不可以。	可以。
OpenAPI Schema	是否存在新资源类别的 OpenAPI (Swagger) Schema 可供动态从服务器上读取？是否存在机制确保只能设置被允许的字段以避免用户犯字段拼写错误？是否实施了字段类型检查（换言之，不允许在 string 字段设置 int 值）？	可以，依据 OpenAPI v3.0 合法性检查 模式（1.16 中进入正式发布状态）。	可以。

公共特性

与在 Kubernetes 平台之外实现定制资源相比，无论是通过 CRD 还是通过聚合 API 来创建定制资源，你都会获得很多 API 特性：

功能特性	具体含义
CRUD	新的端点支持通过 HTTP 和 kubectl 发起的 CRUD 基本操作
监测 (Watch)	新的端点支持通过 HTTP 发起的 Kubernetes Watch 操作
发现 (Discovery)	类似 kubectl 和仪表盘 (Dashboard) 这类客户端能够自动提供列举、显示、在字段级编辑你的资源的操作
json-patch	新的端点支持带 Content-Type: application/json-patch+json 的 PATCH 操作
merge-patch	新的端点支持带 Content-Type: application/merge-patch+json 的 PATCH 操作
HTTPS	新的端点使用 HTTPS
内置身份认证	对扩展的访问会使用核心 API 服务器（聚合层）来执行身份认证操作
内置鉴权授权	对扩展的访问可以复用核心 API 服务器所使用的鉴权授权机制；例如，RBAC
Finalizers	在外部清除工作结束之前阻止扩展资源被删除
准入 Webhooks	在创建、更新和删除操作中对扩展资源设置默认值和执行合法性检查
UI/CLI 展示	kubectl 和仪表盘 (Dashboard) 可以显示扩展资源
区分未设置值和空值	客户端能够区分哪些字段是未设置的，哪些字段的值是被显式设置为零值的。
生成客户端库	Kubernetes 提供通用的客户端库，以及用来生成特定类别客户端库的工具
标签和注解	提供涵盖所有对象的公共元数据结构，且工具知晓如何编辑核心资源和定制资源的这些元数据

准备安装定制资源

在向你的集群添加定制资源之前，有些事情需要搞清楚。

第三方代码和新的失效点的问题

尽管添加新的 CRD 不会自动带来新的失效点（Point of Failure），例如导致第三方代码被在 API 服务器上运行，类似 Helm Charts 这种软件包或者其他安装包通常在提供 CRD 的同时还包含带有第三方 代码的 Deployment，负责实现新的定制资源的业务逻辑。

安装聚合 API 服务器时，也总会牵涉到运行一个新的 Deployment。

存储

定制资源和 ConfigMap 一样也会消耗存储空间。创建过多的定制资源可能会导致 API 服务器上的存储空间超载。

聚合 API 服务器可以使用主 API 服务器的同一存储。如果是这样，你也要注意此警告。

身份认证、鉴权授权以及审计

CRD 通常与 API 服务器上的内置资源一样使用相同的身份认证、鉴权授权 和审计日志机制。

如果你使用 RBAC 来执行鉴权授权，大多数 RBAC 角色都会授权对新资源的访问（除了 cluster-admin 角色以及使用通配符规则创建的其他角色）。你要显式地为新资源的访问授权。CRD 和聚合 API 通常在交付时会包含针对所添加的类别的新的角色定义。

聚合 API 服务器可能会使用主 API 服务器相同的身份认证、鉴权授权和审计 机制，也可能不会。

访问定制资源

Kubernetes [客户端库](#) 可用来访问定制资源。并非所有客户端库都支持定制资源。Go 和 Python 客户端库是支持的。

当你添加了新的定制资源后，可以用如下方式之一访问它们：

- kubectl
- Kubernetes 动态客户端
- 你所编写的 REST 客户端
- 使用 [Kubernetes 客户端生成工具](#) 所生成的客户端。生成客户端的工作有些难度，不过某些项目可能会随着 CRD 或 聚合 API 一起提供一个客户端

接下来

- 了解如何[使用聚合层扩展 Kubernetes API](#)
- 了解如何[使用 CustomResourceDefinition 来扩展 Kubernetes API](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#)。在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#)。

最后修改 November 13, 2020 at 11:03 AM PST: [\[zh\] Sync changes from English site \(4\) \(2d8e136e6\)](#)

通过聚合层扩展 Kubernetes API

使用聚合层（Aggregation Layer），用户可以通过额外的 API 扩展 Kubernetes，而不局限于 Kubernetes 核心 API 提供的功能。

这里的附加 API 可以是[服务目录](#)这类已经成熟的解决方案，也可以是你自己开发的 API。

聚合层不同于[定制资源 \(Custom Resources\)](#)。后者的目的是让 [kube-apiserver](#) 能够认识新的对象类别（Kind）。

聚合层

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/...）转发到已注册的 APIService。

APIService 的最常见实现方式是在集群中某 Pod 内运行 [扩展 API 服务器](#)。如果你在使用扩展 API 服务器来管理集群中的资源，该扩展 API 服务器（也被写成“extension-apiserver”）一般需要和一个或多个[控制器](#)一起使用。`apiserver-builder` 库同时提供构造扩展 API 服务器和控制器框架代码。

反应延迟

扩展 API 服务器与 kube-apiserver 之间需要存在低延迟的网络连接。发现请求需要在五秒钟或更短的时间内完成到 kube-apiserver 的往返。

如果你的扩展 API 服务器无法满足这一延迟要求，应考虑如何更改配置已满足需要。你也可以为 kube-apiserver 设置 `EnableAggregatedDiscoveryTimeout=false` [特性门控](#) 来禁用超时限制。此特性门控已经废弃，将在未来版本中被删除。

接下来

- 阅读[配置聚合层](#) 文档，了解如何在自己的环境中启用聚合器。
- 接下来，了解[安装扩展 API 服务器](#)，开始使用聚合层。

- 也可以学习怎样[使用自定义资源定义扩展 Kubernetes API](#)。
- 阅读[APIService](#) 的规范

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

Operator 模式

Operator 是 Kubernetes 的扩展软件，它利用[定制资源](#)管理应用及其组件。Operator 遵循 Kubernetes 的理念，特别是在[控制器](#)方面。

初衷

Operator 模式旨在捕获（正在管理一个或一组服务的）运维人员的关键目标。负责特定应用和 service 的运维人员，在系统应该如何运行、如何部署以及出现问题时如何处理等方面有深入的了解。

在 Kubernetes 上运行工作负载的人们都喜欢通过自动化来处理重复的任务。Operator 模式会封装你编写的（Kubernetes 本身提供功能以外的）任务自动化代码。

Kubernetes 上的 Operator

Kubernetes 为自动化而生。无需任何修改，你即可以从 Kubernetes 核心中获得许多内置的自动化功能。你可以使用 Kubernetes 自动化部署和运行工作负载，甚至可以自动化 Kubernetes 自身。

Kubernetes [控制器](#) 使你无需修改 Kubernetes 自身的代码，即可以扩展集群的行为。Operator 是 Kubernetes API 的客户端，充当[定制资源](#)的控制器。

Operator 示例

使用 Operator 可以自动化的事情包括：

- 按需部署应用
- 获取/还原应用状态的备份
- 处理应用代码的升级以及相关改动。例如，数据库 schema 或额外的配置设置
- 发布一个 service，要求不支持 Kubernetes API 的应用也能发现它
- 模拟整个或部分集群中的故障以测试其稳定性

- 在没有内部成员选举程序的情况下，为分布式应用选择首领角色

想要更详细的了解 Operator ? 这儿有一个详细的示例：

1. 有一个名为 SampleDB 的自定义资源，你可以将其配置到集群中。
2. 一个包含 Operator 控制器部分的 Deployment，用来确保 Pod 处于运行状态。
3. Operator 代码的容器镜像。
4. 控制器代码，负责查询控制平面以找出已配置的 SampleDB 资源。
5. Operator 的核心是告诉 API 服务器，如何使现实与代码里配置的资源匹配。
 - 如果添加新的 SampleDB，Operator 将设置 PersistentVolumeClaims 以提供持久化的数据库存储，设置 StatefulSet 以运行 SampleDB，并设置 Job 来处理初始配置。
 - 如果你删除它，Operator 将建立快照，然后确保 StatefulSet 和 Volume 已被删除。
6. Operator 也可以管理常规数据库的备份。对于每个 SampleDB 资源，Operator 会确定何时创建（可以连接到数据库并进行备份的）Pod。这些 Pod 将依赖于 ConfigMap 和/或具有数据库连接详细信息和凭据的 Secret。
7. 由于 Operator 旨在为其管理的资源提供强大的自动化功能，因此它还需要一些额外的支持性代码。在这个示例中，代码将检查数据库是否正运行在旧版本上，如果是，则创建 Job 对象为你升级数据库。

部署 Operator

部署 Operator 最常见的方法是将自定义资源及其关联的控制器添加到你的集群中。跟运行容器化应用一样，控制器通常会运行在 [控制平面](#) 之外。例如，你可以在集群中将控制器作为 Deployment 运行。

使用 Operator

部署 Operator 后，你可以对 Operator 所使用的资源执行添加、修改或删除操作。按照上面的示例，你将为 Operator 本身建立一个 Deployment，然后：

```
kubectl get SampleDB # 查找所配置的数据库
```

```
kubectl edit SampleDB/example-database # 手动修改某些配置
```

可以了！Operator 会负责应用所作的更改并保持现有服务处于良好的状态。

编写你自己的 Operator

如果生态系统中没可以实现你目标的 Operator，你可以自己编写代码。在 [接下来](#)一节中，你会找到编写自己的云原生 Operator 需要的库和工具的链接。

你还可以使用任何支持 [Kubernetes API 客户端](#) 的语言或运行时来实现 Operator（即控制器）。

接下来

- 详细了解[定制资源](#)
- 在[OperatorHub.io](#) 上找到现成的、适合你的 Operator
- 借助已有的工具来编写你自己的 Operator，例如：
 - [KUDO](#) (Kubernetes 通用声明式 Operator)
 - [kubebuilder](#)
 - [Metacontroller](#)，可与 Webhook 结合使用，以实现自己的功能。
 - [Operator Framework](#)
- [发布](#)你的 Operator，让别人也可以使用
- 阅读[CoreOS 原文](#)，其介绍了 Operator 介绍
- 阅读这篇来自谷歌云的关于构建 Operator 最佳实践的 [文章](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问[Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 December 04, 2020 at 2:50 PM PST: [\[zh\] Fix links in zh localization \(2\) \(35b632715\)](#)

计算、存储和网络扩展

[网络插件](#)

[设备插件](#)

使用 Kubernetes 设备插件框架来实现适用于 GPU、NIC、FPGA、InfiniBand 以及类似的需要特定于供应商设置的资源的插件。

网络插件

Kubernetes中的网络插件有几种类型：

- CNI 插件：遵守[容器网络接口 \(Container Network Interface , CNI \)](#) 规范，其设计上偏重互操作性。
 - Kubernetes 遵从 CNI 规范的 [v0.4.0](#) 版本。
- Kubenet 插件：使用 bridge 和 host-local CNI 插件实现了基本的 cbr0。

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 CRI 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](#) 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` 命令行选项可以选择 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 配置来设置各个 Pod 的网络。CNI 配置文件必须与 [CNI 规约](#) 匹配，并且配置所引用的所有所需的 CNI 插件都应存在于 `--cni-bin-dir`（默认是 `/opt/cni/bin`）下。

如果这个目录中有多个 CNI 配置文件，kubelet 将会使用按文件名的字典顺序排列的第一个作为配置文件。

除了配置文件指定的 CNI 插件外，Kubernetes 还需要标准的 CNI [lo](#) 插件，最低版本是 0.2.0。

支持 hostPort

CNI 网络插件支持 hostPort。你可以使用官方 [portmap](#) 插件，它由 CNI 插件团队提供，或者使用你自己的带有 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

实验功能

CNI 网络插件还支持 pod 入口和出口流量整形。你可以使用 CNI 插件团队提供的 [bandwidth](#) 插件，也可以使用你自己的具有带宽控制功能的插件。

如果你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）并保证该可执行文件包含在你的 CNI 的 bin 文件夹内（默认为 /opt/cni/bin）。

```
{
  "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，你可以将 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。 或者如果你正在使用 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，你可以使用 `--network-plugin-mtu` kubelet 选项显式的指定 MTU。 例如：在 AWS 上 `eth0` MTU 通常是 9001，因此你可以指定 `--network-plugin-mtu=9001`。 如果你正在使用 IPSEC，你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**。

用法总结

- `--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。

接下来

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 13, 2020 at 11:03 AM PST: [\[zh\] Sync changes from English site \(4\) \(2d8e136e6\)](#)

设备插件

使用 Kubernetes 设备插件框架来实现适用于 GPU、NIC、FPGA、InfiniBand 以及类似的需要特定于供应商设置的资源的插件。

FEATURE STATE: Kubernetes v1.10 [beta]

Kubernetes 提供了一个 [设备插件框架](#)，你可以用它来将系统硬件资源发布到 [Kubelet](#)。
。

供应商可以实现设备插件，由你手动部署或作为 [DaemonSet](#) 来部署，而不必定制 Kubernetes 本身的代码。目标设备包括 GPU、高性能 NIC、FPGA、InfiniBand 适配器以及其他类似的、可能需要特定于供应商的初始化和设置的计算资源。

注册设备插件

kubelet 提供了一个 Registration 的 gRPC 服务：

```
service Registration {
    rpc Register(RegisterRequest) returns (Empty) {}
}
```

设备插件可以通过此 gRPC 服务在 kubelet 进行注册。在注册期间，设备插件需要发送下面几样内容：

- 设备插件的 Unix 套接字。
- 设备插件的 API 版本。
- `ResourceName` 是需要公布的。这里 `ResourceName` 需要遵循 [扩展资源命名方案](#)，类似于 `vendor-domain/resourcetype`。（比如 NVIDIA GPU 就被公布为 `nvidia.com/gpu`。）

成功注册后，设备插件就向 kubelet 发送它所管理的设备列表，然后 kubelet 负责将这些资源发布到 API 服务器，作为 kubelet 节点状态更新的一部分。

比如，设备插件在 kubelet 中注册了 `hardware-vendor.example/foo` 并报告了节点上的两个运行状况良好的设备后，节点状态将更新以通告该节点已安装 2 个 "Foo" 设备并且是可用的。

然后用户需要请求其他类型的资源的时候，就可以在 [Container](#) 规范请求这类设备，但是有以下的限制：

- 扩展资源仅可作为整数资源使用，并且不能被过量使用
- 设备不能在容器之间共享

假设 Kubernetes 集群正在运行一个设备插件，该插件在一些节点上公布的资源为 `hardware-vendor.example/foo`。下面就是一个 Pod 示例，请求此资源以运行某演示负载：

```
---
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
```

```
#  
# 这个 pod 需要两个 hardware-vendor.example/foo 设备  
# 而且只能够调度到满足需求的节点上  
#  
# 如果该节点中有 2 个以上的设备可用，其余的可供其他 Pod 使用
```

设备插件的实现

设备插件的常规工作流程包括以下几个步骤：

- 初始化。在这个阶段，设备插件将执行供应商特定的初始化和设置，以确保设备处于就绪状态。
- 插件使用主机路径 /var/lib/kubelet/device-plugins/ 下的 Unix 套接字启动一个 gRPC 服务，该服务实现以下接口：

```
service DevicePlugin {  
    // ListAndWatch 返回 Device 列表构成的数据流。  
    // 当 Device 状态发生变化或者 Device 消失时，ListAndWatch  
    // 会返回新的列表。  
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}  
  
    // Allocate 在容器创建期间调用，这样设备插件可以运行一些特定于设备的操作，  
    // 并告诉 kubelet 如何令 Device 可在容器中访问的所需执行的具体步骤  
    rpc Allocate(AssignRequest) returns (AssignResponse) {}  
  
    // GetPreferredAllocation 从一组可用的设备中返回一些优选的设备用来分配，  
    // 所返回的优选分配结果不一定会是设备管理器的最终分配方案。  
    // 此接口的设计仅是为了让设备管理器能够在可能的情况下做出更有意义的决定。  
    rpc GetPreferredAllocation(PreferredAllocationRequest) returns  
(PreferredAllocationResponse) {}  
  
    // PreStartContainer 在设备插件注册阶段根据需要被调用，调用发生在容器启动之前。  
    // 在将设备提供给容器使用之前，设备插件可以运行一些诸如重置设备之类的特定于  
    // 具体设备的操作，  
    rpc PreStartContainer(PreStartContainerRequest) returns  
(PreStartContainerResponse) {}  
}
```

说明：

插件并非必须为 GetPreferredAllocation() 或 PreStartContainer() 提供有用的实现逻辑，调用 GetDevicePluginOptions() 时所返回的 DevicePluginOptions 消息中应该设置这些调用是否可用。kubelet 在真正调用这些函数之前，总会调用 GetDevicePluginOptions() 来查看是否存在这些可选的函数。

- 插件通过 Unix socket 在主机路径 /var/lib/kubelet/device-plugins/kubelet.sock 处向 kubelet 注册自身。
- 成功注册自身后，设备插件将以服务模式运行，在此期间，它将持续监控设备运行状况，并在设备状态发生任何变化时向 kubelet 报告。它还负责响应 Allocate gRPC 请求。在 Allocate 期间，设备插件可能还会做一些设备特定的准备；例如 GPU 清理或 QRNG 初始化。如果操作成功，则设备插件将返回 AllocateResponse，其中包含用于访问被分配的设备容器运行时的配置。kubelet 将此信息传递到容器运行时。

处理 kubelet 重启

设备插件应能监测到 kubelet 重启，并且向新的 kubelet 实例来重新注册自己。在当前实现中，当 kubelet 重启的时候，新的 kubelet 实例会删除 /var/lib/kubelet/device-plugins 下所有已经存在的 Unix 套接字。设备插件需要能够监控到它的 Unix 套接字被删除，并且当发生此类事件时重新注册自己。

设备插件部署

你可以将你的设备插件作为节点操作系统的软件包来部署、作为 DaemonSet 来部署或者手动部署。

规范目录 /var/lib/kubelet/device-plugins 是需要特权访问的，所以设备插件必须要在被授权的安全的上下文中运行。如果你将设备插件部署为 DaemonSet，/var/lib/kubelet/device-plugins 目录必须要在插件的 [PodSpec](#) 中声明作为 [卷 \(Volume\)](#) 被挂载到插件中。

如果你选择 DaemonSet 方法，你可以通过 Kubernetes 进行以下操作：将设备插件的 Pod 放置在节点上，在出现故障后重新启动守护进程 Pod，来进行自动升级。

API 兼容性

Kubernetes 设备插件支持还处于 beta 版本。所以在稳定版本出来之前 API 会以不兼容的方式进行更改。作为一个项目，Kubernetes 建议设备插件开发者：

- 注意未来版本的更改
- 支持多个版本的设备插件 API，以实现向后/向前兼容性。

如果你启用 DevicePlugins 功能，并在需要升级到 Kubernetes 版本来获得较新的设备插件 API 版本的节点上运行设备插件，请在升级这些节点之前先升级设备插件以支持这两个版本。采用该方法将确保升级期间设备分配的连续运行。

监控设备插件资源

FEATURE STATE: Kubernetes v1.15 [beta]

为了监控设备插件提供的资源，监控代理程序需要能够发现节点上正在使用的设备，并获取元数据来描述哪个指标与容器相关联。设备监控代理暴露给 [Prometheus](#) 的指标应该遵循 [Kubernetes Instrumentation Guidelines](#)，使用 pod、namespace 和 container 标签来标识容器。

kubelet 提供了 gRPC 服务来使得正在使用中的设备被发现，并且还为这些设备提供了元数据：

```
// 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) {}
}
```

gRPC 服务通过 /var/lib/kubelet/pod-resources/kubelet.sock 的 UNIX 套接字来提供服务。设备插件资源的监控代理程序可以部署为守护进程或者 DaemonSet。规范的路径 /var/lib/kubelet/pod-resources 需要特权来进入，所以监控代理程序必须要在获得授权的安全的上下文中运行。如果设备监控代理以 DaemonSet 形式运行，必须要在插件的 [PodSpec](#) 中声明将 /var/lib/kubelet/pod-resources 目录以 [卷](#) 的形式被挂载到容器中。

对“PodResources 服务”的支持要求启用 KubeletPodResources [特性门控](#)。从 Kubernetes 1.15 开始默认启用。

设备插件与拓扑管理器的集成

FEATURE STATE: Kubernetes v1.18 [beta]

拓扑管理器是 Kubelet 的一个组件，它允许以拓扑对齐方式来调度资源。为了做到这一点，设备插件 API 进行了扩展来包括一个 TopologyInfo 结构体。

```
message TopologyInfo {
    repeated NUMANode nodes = 1;
}

message NUMANode {
    int64 ID = 1;
}
```

设备插件希望拓扑管理器可以将填充的 TopologyInfo 结构体作为设备注册的一部分以及设备 ID 和设备的运行状况发送回去。然后设备管理器将使用此信息来咨询拓扑管理器并做出资源分配决策。

TopologyInfo 支持定义 nodes 字段，允许为 nil (默认) 或者是一个 NUMA 节点的列表。这样就可以使设备插件可以跨越 NUMA 节点去发布。

下面是一个由设备插件为设备填充 TopologyInfo 结构体的示例：

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

设备插件示例

下面是一些设备插件实现的示例：

- [AMD GPU 设备插件](#)
- [Intel 设备插件](#) 支持 Intel GPU、FPGA 和 QuickAssist 设备
- [KubeVirt 设备插件](#) 用于硬件辅助的虚拟化
- The [NVIDIA GPU 设备插件](#)
 - 需要 [nvidia-docker](#) 2.0，以允许运行 Docker 容器的时候启用 GPU。
- [为 Container-Optimized OS 所提供的 NVIDIA GPU 设备插件](#)
- [RDMA 设备插件](#)
- [Solarflare 设备插件](#)
- [SR-IOV 网络设备插件](#)
- [Xilinx FPGA 设备插件](#)

接下来

- 查看[调度 GPU 资源](#) 来学习使用设备插件
- 查看在上如何[公布节点上的扩展资源](#)
- 阅读如何在 Kubernetes 中使用 [TLS Ingress 的硬件加速](#)
- 学习[拓扑管理器](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 13, 2020 at 11:03 AM PST: [\[zh\] Sync changes from English site \(4\) \(2d8e136e6\)](#)

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 \(Service Brokers\)](#) 提供的外部托管服务 ([Managed Services](#))，而无需知道那些服务具体是怎样创建和托管的。

服务代理 (Service Broker) 是由[Open Service Broker API 规范](#)定义的一组托管服务的端点，这些服务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，但它们可以是应用程序能够使用的任何软件交付物。

使用服务目录，[集群操作员](#) 可以浏览某服务代理所提供的托管服务列表，供应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

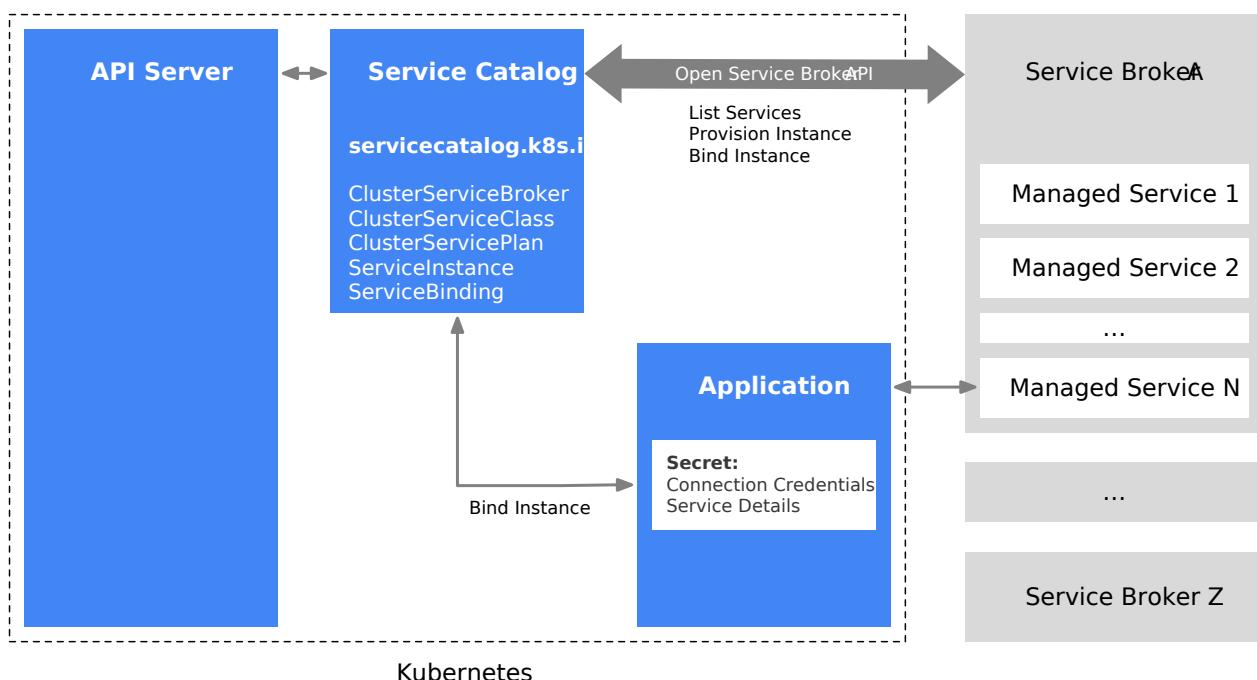
[应用开发人员](#)，希望使用消息队列，作为其在 Kubernetes 集群中运行的应用程序的一部分。但是，他们不想承受构造这种服务的开销，也不想自行管理。幸运的是，有一家云服务提供商通过其服务代理以托管服务的形式提供消息队列服务。

集群操作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，进而部署消息队列服务的实例 并使其对 Kubernetes 中的应用程序可用。应用开发者于是可以不关心消息队列的实现细节，也不用对其进行管理。他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[Open Service Broker API](#) 与服务代理进行通信，并作为 Kubernetes API 服务器的中介，以便协商启动部署和获取 应用程序使用托管服务时必须的凭据。

服务目录实现为一个扩展 API 服务器和一个控制器，使用 Etcd 提供存储。它还使用了 Kubernetes 1.7 之后版本中提供的 [聚合层](#) 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：服务目录的集群内表现形式，封装了其服务连接细节。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：

- 基本认证（用户名/密码）
- [OAuth 2.0 不记名令牌](#)

使用方式

集群运维人员可以使用服务目录 API 资源来供应托管服务并使其在 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。

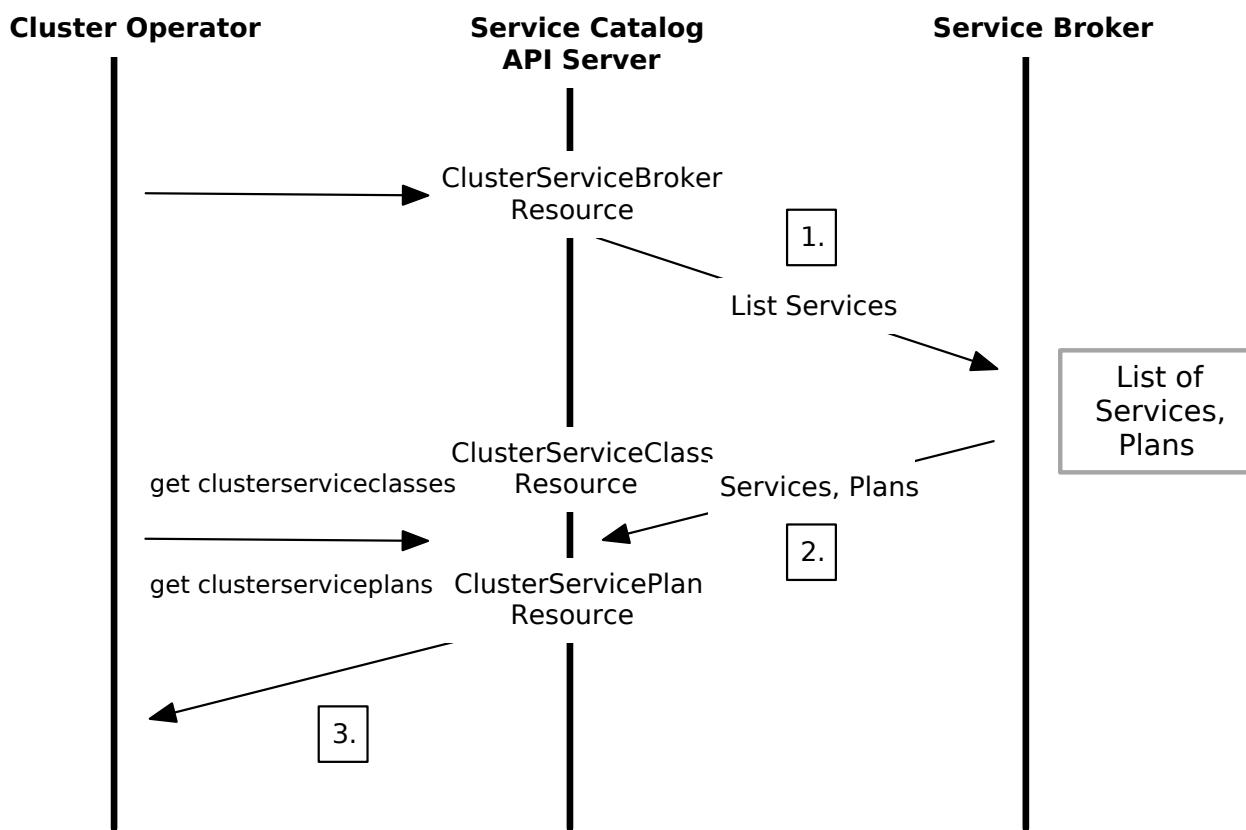
这是一个 ClusterServiceBroker 资源的例子：

```

apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceBroker
metadata:
  name: cloud-broker
spec:
  # 指向服务代理的末端。 (这里的 URL 是无法使用的)
  url: https://servicebroker.somecloudprovider.com/v1alpha1/projects/service-
catalog/brokers/default
  #####
  # 这里可以添加额外的用来与服务代理通信的属性值,
  # 例如持有者令牌信息或者 TLS 的 CA 包
  #####

```

下面的时序图展示了从服务代理列出可用托管服务和计划所涉及的各个步骤：



- 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 调用，以列举所有可用服务；
- 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。
- 集群运维人员接下来可以使用以下命令获取可用托管服务的列表：

```

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
...	...

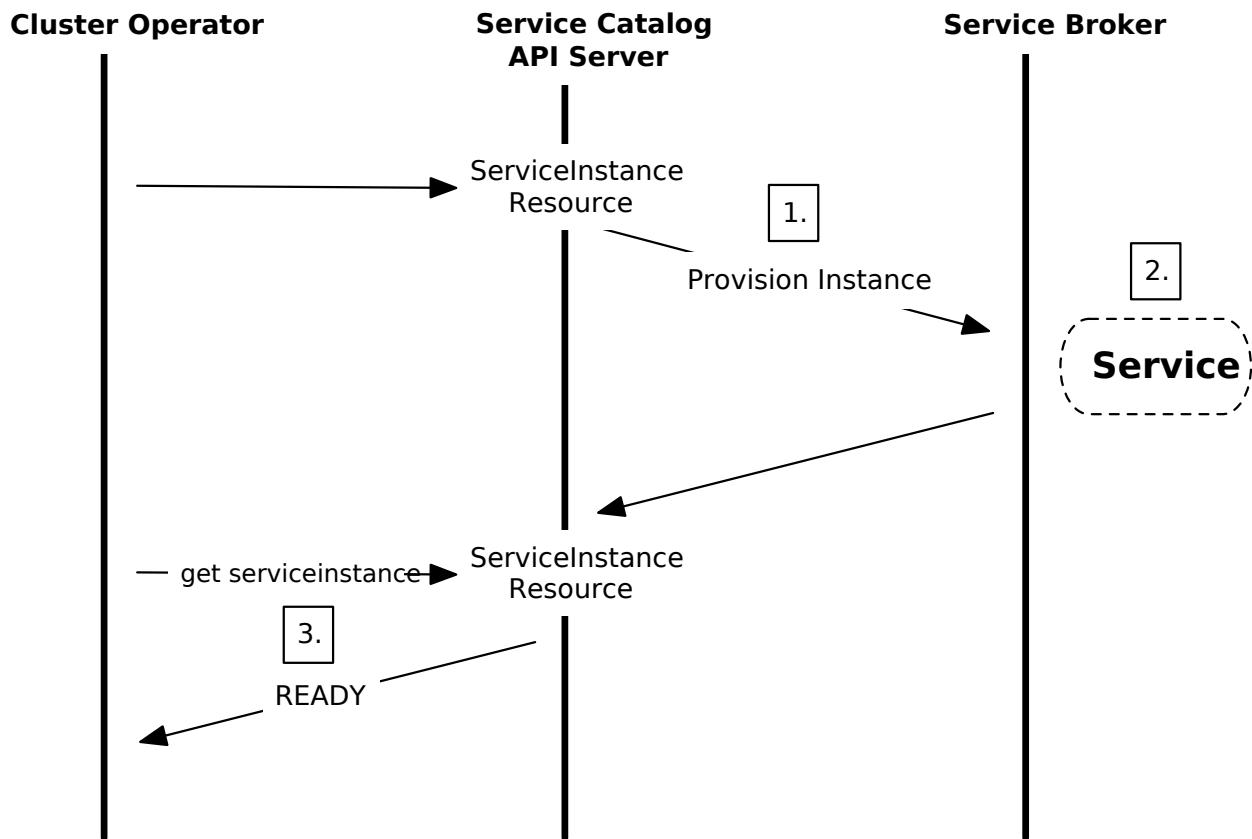
供应一个新实例

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。

下面是一个 ServiceInstance 资源的例子：

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceInstance
metadata:
  name: cloud-queue-instance
  namespace: cloud-apps
spec:
  # 引用之前返回的服务之一
  clusterServiceClassExternalName: cloud-provider-service
  clusterServicePlanExternalName: service-plan-name
  #####
  # 这里可添加额外的参数，供服务代理使用
  #####
```

以下时序图展示了配置托管服务新实例所涉及的步骤：



1. 创建 `ServiceInstance` 资源时，服务目录将启动一个到外部服务代理的调用，请求供应一个实例。
2. 服务代理创建一个托管服务的新实例并返回 HTTP 响应。
3. 接下来，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。

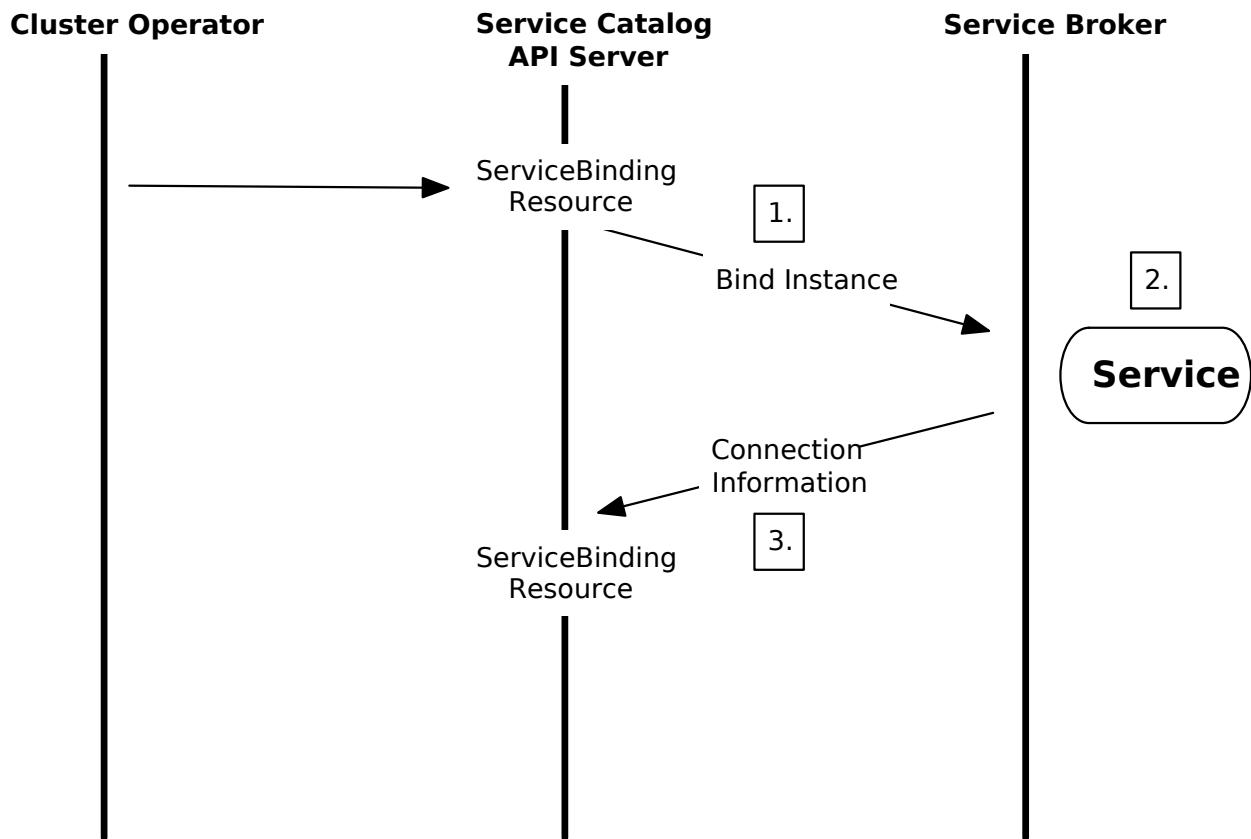
以下是 `ServiceBinding` 资源的示例：

```

apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: cloud-queue-binding
  namespace: cloud-apps
spec:
  instanceRef:
    name: cloud-queue-instance
    #####
    # 这里可以添加供服务代理使用的额外信息，例如 Secret 名称或者服务账号参数，
    #####

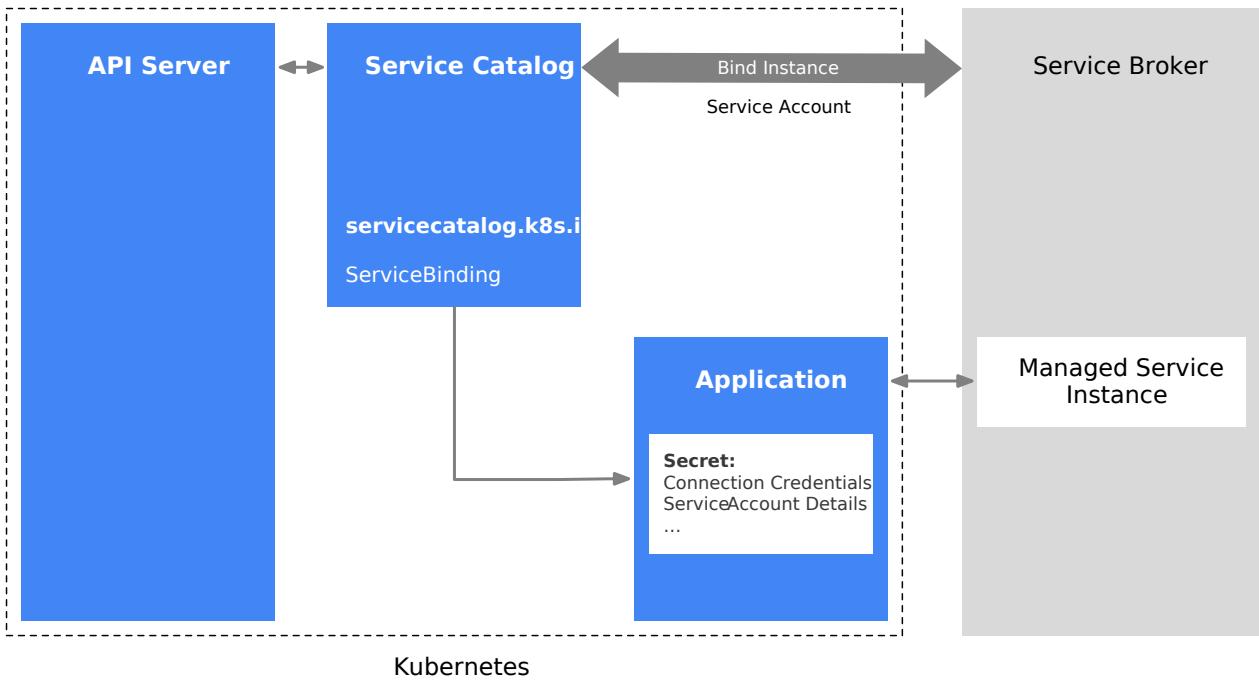
```

以下顺序图展示了绑定到托管服务实例的步骤：



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 的键映射到环境变量 TOPIC 中。

```
...
  env:
    - name: "TOPIC"
      valueFrom:
        secretKeyRef:
          name: provider-queue-credentials
          key: topic
```

接下来

- 如果你熟悉 [Helm Charts](#)，可以使用 [Helm 安装服务目录](#) 到 Kubernetes 集群中。或者，你可以 [使用 SC 工具安装服务目录](#)。
- 查看[服务代理示例](#)
- 浏览 [kubernetes-sigs/service-catalog](#) 项目
- 查看 [svc-cat.io](#)

反馈

此页是否对您有帮助？

是 否

感谢反馈。如果您有一个关于如何使用 Kubernetes 的特定的、需要答案的问题，可以访问 [Stack Overflow](#). 在 GitHub 仓库上登记新的问题 [报告问题](#) 或者 [提出改进建议](#).

最后修改 November 13, 2020 at 11:03 AM PST: [\[zh\] Sync changes from English site \(4\) \(2d8e136e6\)](#)