教程

Kubernetes 文档的这一部分包含教程。每个教程展示了如何完成一个比单个 任务更大的目标。 通常一个教程有几个部分，每个部分都有一系列步骤。在浏览每个教程之前，您可能希望将标准化术语表页面添加到书签，供以后参考。

基础知识

- Kubernetes 基础知识是一个深入的 交互式教程，帮助您理解 Kubernetes 系统，并尝试一些基本的 Kubernetes 特性。
- 介绍 Kubernetes (edx)
- 你好 Minikube

配置

- 使用一个 ConfigMap 配置 Redis

无状态应用程序

- 公开外部 IP 地址访问集群中的应用程序
- 示例：使用 Redis 部署 PHP 留言板应用程序

有状态应用程序

- StatefulSet 基础
- 示例：WordPress 和 MySQL 使用持久卷
- 示例：使用有状态集部署 Cassandra
- 运行 ZooKeeper，CP 分布式系统

集群

- AppArmor

服务

- 使用源 IP

接下来

如果您想编写教程，请参阅内容页面类型 以获取有关教程页面类型的信息。

你好，Minikube

本教程向你展示如何使用 Minikube 和 Katacoda 在 Kubernetes 上运行一个应用示例。Katacoda 提供免费的浏览器内 Kubernetes 环境。

> **说明：** 如果你已在本地安装 Minikube，也可以按照本教程操作。 安装指南参阅 minikube start 。

教程目标

- 将一个示例应用部署到 Minikube。
- 运行应用程序。
- 查看应用日志

准备开始

本教程提供了容器镜像，使用 NGINX 来对所有请求做出回应：

创建 Minikube 集群

1. 点击 **启动终端**

   最后修改 January 12, 2021 at 4:09 PM PST: url miss. some lines are outdate. (9a33c7592)

   # 学习 Kubernetes 基础知识

   html

   ## Kubernetes 基础

   本教程介绍了 Kubernetes 集群编排系统的基础知识。每个模块包含关于 Kubernetes 主要特性和概念的一些背景信息，并包括一个在线互动教程。这些互动教程让您可以自己管理一个简单的集群及其容器化应用程序。

   使用互动教程，您可以学习：

   - 在集群上部署容器化应用程序
   - 弹性部署
   - 使用新的软件版本，更新容器化应用程序
   - 调试容器化应用程序

教程 Katacoda 在您的浏览器中运行一个虚拟终端，在浏览器中运行 Minikube，这是一个可在任何地方小规模本地部署的 Kubernetes 集群。不需要安装任何软件或进行任何配置；每个交互性教程都直接从您的网页浏览器上运行。

Kubernetes 可以为您做些什么?

通过现代的 Web 服务，用户希望应用程序能够 24/7 全天候使用，开发人员希望每天可以多次发布部署新版本的应用程序。 容器化可以帮助软件包达成这些目标，使应用程序能够以简单快速的方式发布和更新，而无需停机。Kubernetes 帮助您确保这些容器化的应用程序在您想要的时间和地点运行，并帮助应用程序找到它们需要的资源和工具。Kubernetes 是一个可用于生产的开源平台，根据 Google 容器集群方面积累的经验，以及来自社区的最佳实践而设计。

Kubernetes 基础模块

1. 创建一个 Kubernetes 集群

2. 部署应用程序

3. 应用程序探索

4. 应用外部可见

5. 应用可扩展

6. 应用更新

创建集群

使用 Minikube 创建集群

交互式教程 - 创建集群

使用 Minikube 创建集群

html

目标

- 了解 Kubernetes 集群。
- 了解 Minikube 。
- 使用在线终端开启一个 Kubernetes 集群。

Kubernetes 集群

Kubernetes 协调一个高可用计算机集群，每个计算机作为独立单元互相连接工作。 Kubernetes 中的抽象允许您将容器化的应用部署到集群，而无需将它们绑定到某个特定的独立计算机。为了使用这种新的部署模型，应用需要以将应用与单个主机分离的方式打包：它们需要被容器化。与过去的那种应用直接以包的方式深度与主机集成的部署模型相比，容器化应用更灵活、更可用。 **Kubernetes 以更高效的方式跨集群自动分发和调度应用容器。** Kubernetes 是一个开源平台，并且可应用于生产环境。

一个 Kubernetes 集群包含两种类型的资源:

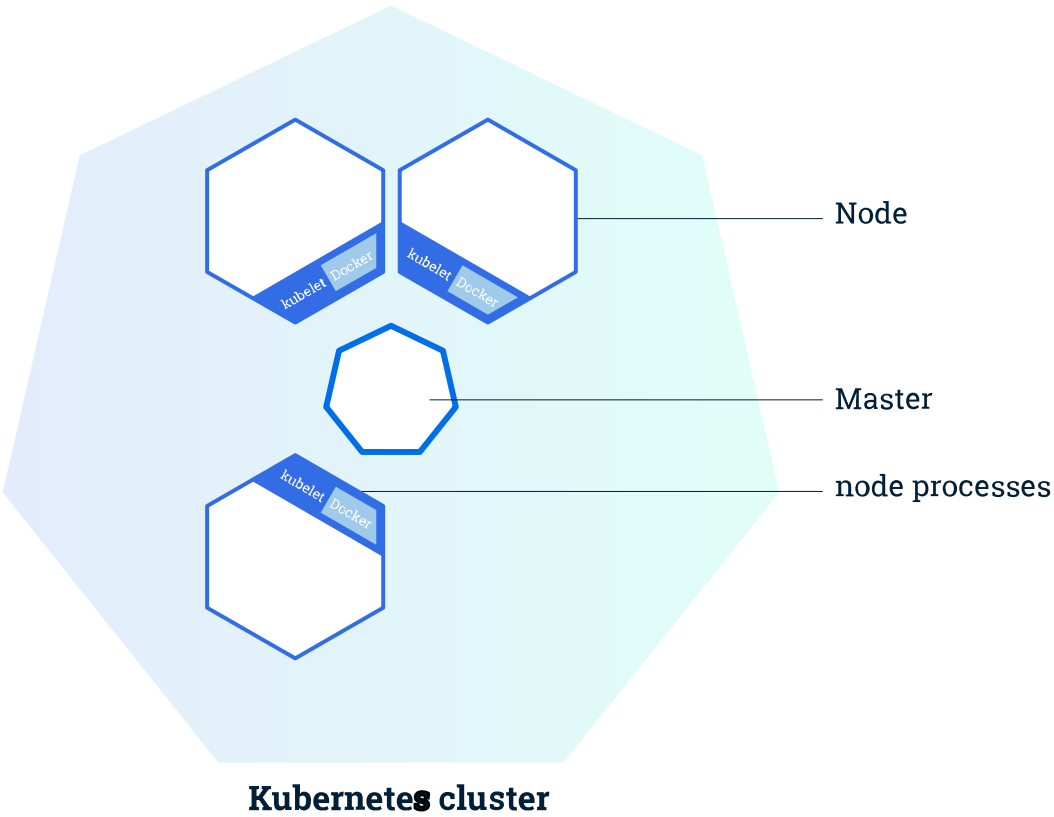
- **Master** 调度整个集群
- **Nodes** 负责运行应用

总结:

- Kubernetes 集群
- Minikube

Kubernetes 是一个生产级别的开源平台，可协调在计算机集群内和跨计算机集群的应用容器的部署（调度）和执行.

集群图



Master 负责管理整个集群。 Master 协调集群中的所有活动，例如调度应用、维护应用的所需状态、应用扩容以及推出新的更新。

Node 是一个虚拟机或者物理机，它在 Kubernetes 集群中充当工作机器的角色
每个Node都有 Kubelet，它管理 Node 而且是 Node 与 Master 通信的代理。Node 还应该具有用于处理容器操作的工具，例如 Docker 或 rkt 。处理生产级流量的 Kubernetes 集群至少应具有三个 Node 。

Master 管理集群，Node 用于托管正在运行的应用。

在 Kubernetes 上部署应用时，您告诉 Master 启动应用容器。 Master 就编排容器在集群的 Node 上运行。 **Node 使用 Master 暴露的 Kubernetes API 与 Master 通信。**终端用户也可以使用 Kubernetes API 与集群交互。

Kubernetes 既可以部署在物理机上也可以部署在虚拟机上。您可以使用 Minikube 开始部署 Kubernetes 集群。 Minikube 是一种轻量级的 Kubernetes 实现，可在本地计算机上创建 VM 并部署仅包含一个节点的简单集群。 Minikube 可用于 Linux ， macOS 和 Windows 系统。Minikube CLI 提供了用于引导集群工作的多种操作，包括启动、停止、查看状态和删除。在本教程里，您可以使用预装有 Minikube 的在线终端进行体验。

既然您已经知道 Kubernetes 是什么，让我们转到在线教程并启动我们的第一个 Kubernetes 集群！

启动交互教程 ›

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 24, 2020 at 5:12 PM PST: Update cluster-intro.html (aadf52491)

交互式教程 - 创建集群

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

部署应用

使用 kubectl 创建 Deployment

交互式教程 - 部署应用

使用 kubectl 创建 Deployment

html

目标

- 学习了解应用的部署
- 使用 kubectl 在 Kubernetes 上部署第一个应用

Kubernetes 部署

一旦运行了 Kubernetes 集群，就可以在其上部署容器化应用程序。 为此，您需要创建 Kubernetes **Deployment** 配置。Deployment 指挥 Kubernetes 如何

创建和更新应用程序的实例。创建 Deployment 后，Kubernetes master 将应用程序实例调度到集群中的各个节点上。

创建应用程序实例后，Kubernetes Deployment 控制器会持续监视这些实例。如果托管实例的节点关闭或被删除，则 Deployment 控制器会将该实例替换为群集中另一个节点上的实例。 **这提供了一种自我修复机制来解决机器故障维护问题。**

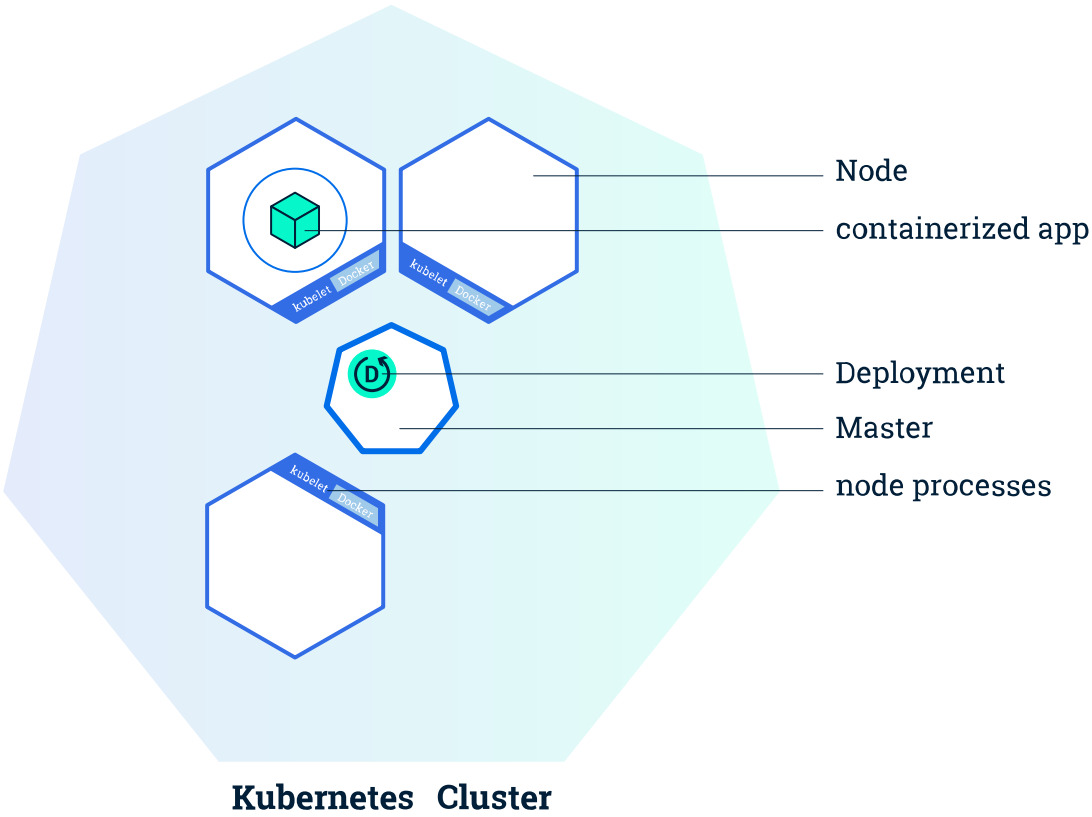
在没有 Kubernetes 这种编排系统之前，安装脚本通常用于启动应用程序，但它们不允许从机器故障中恢复。通过创建应用程序实例并使它们在节点之间运行，Kubernetes Deployments 提供了一种与众不同的应用程序管理方法。

总结:

- Deployments
- Kubectl

Deployment 负责创建和更新应用程序的实例

部署你在 Kubernetes 上的第一个应用程序



您可以使用 Kubernetes 命令行界面 **Kubectl** 创建和管理 Deployment。Kubectl 使用 Kubernetes API 与集群进行交互。在本单元中，您将学习创建在 Kubernetes 集群上运行应用程序的 Deployment 所需的最常见的 Kubectl 命令。

创建 Deployment 时，您需要指定应用程序的容器映像以及要运行的副本数。您可以稍后通过更新 Deployment 来更改该信息; 模块 5 和 6 讨论了如何扩展和更新 Deployments。

应用程序需要打包成一种受支持的容器格式，以便部署在 Kubernetes 上

对于我们的第一次部署，我们将使用打包在 Docker 容器中的 Node.js 应用程序。 要创建 Node.js 应用程序并部署 Docker 容器，请按照 你好 Minikube 教程.

现在您已经了解了 Deployment 的内容，让我们转到在线教程并部署我们的第一个应用程序！

开始交互式教程 ›

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 12:58 AM PST: Update deploy-intro.html (f1de32954)

交互式教程 - 部署应用

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

了解你的应用

查看 pod 和工作节点

交互式教程-了解你的应用

查看 pod 和工作节点

html

目标

- 了解 Kubernetes Pod。
- 了解 Kubernetes 工作节点。
- 对已部署的应用故障排除。

Kubernetes Pods

在模块 2创建 Deployment 时, Kubernetes 添加了一个 **Pod** 来托管你的应用实例。Pod 是 Kubernetes 抽象出来的，表示一组一个或多个应用程序容器（如 Docker），以及这些容器的一些共享资源。这些资源包括:

- 共享存储，当作卷
- 网络，作为唯一的集群 IP 地址
- 有关每个容器如何运行的信息，例如容器映像版本或要使用的特定端口。

Pod 为特定于应用程序的"逻辑主机"建模，并且可以包含相对紧耦合的不同应用容器。例如，Pod 可能既包含带有 Node.js 应用的容器，也包含另一个不同的容器，用于提供 Node.js 网络服务器要发布的数据。Pod 中的容器共享 IP 地址和端口，始终位于同一位置并且共同调度，并在同一工作节点上的共享上下文中运行。

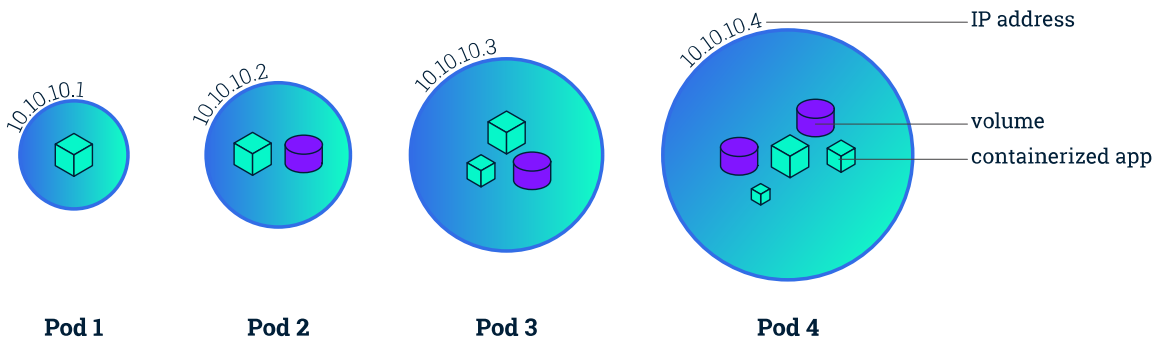
Pod是 Kubernetes 平台上的原子单元。 当我们在 Kubernetes 上创建 Deployment 时，该 Deployment 会在其中创建包含容器的 Pod （而不是直接创建容器）。每个 Pod 都与调度它的工作节点绑定，并保持在那里直到终止（根据重启策略）或删除。 如果工作节点发生故障，则会在群集中的其他可用工作节点上调度相同的 Pod。

总结:

- Pods
- 工作节点
- Kubectl 主要命令

Pod 是一组一个或多个应用程序容器（例如 Docker），包括共享存储（卷), IP 地址和有关如何运行它们的信息。

Pod 概览



工作节点

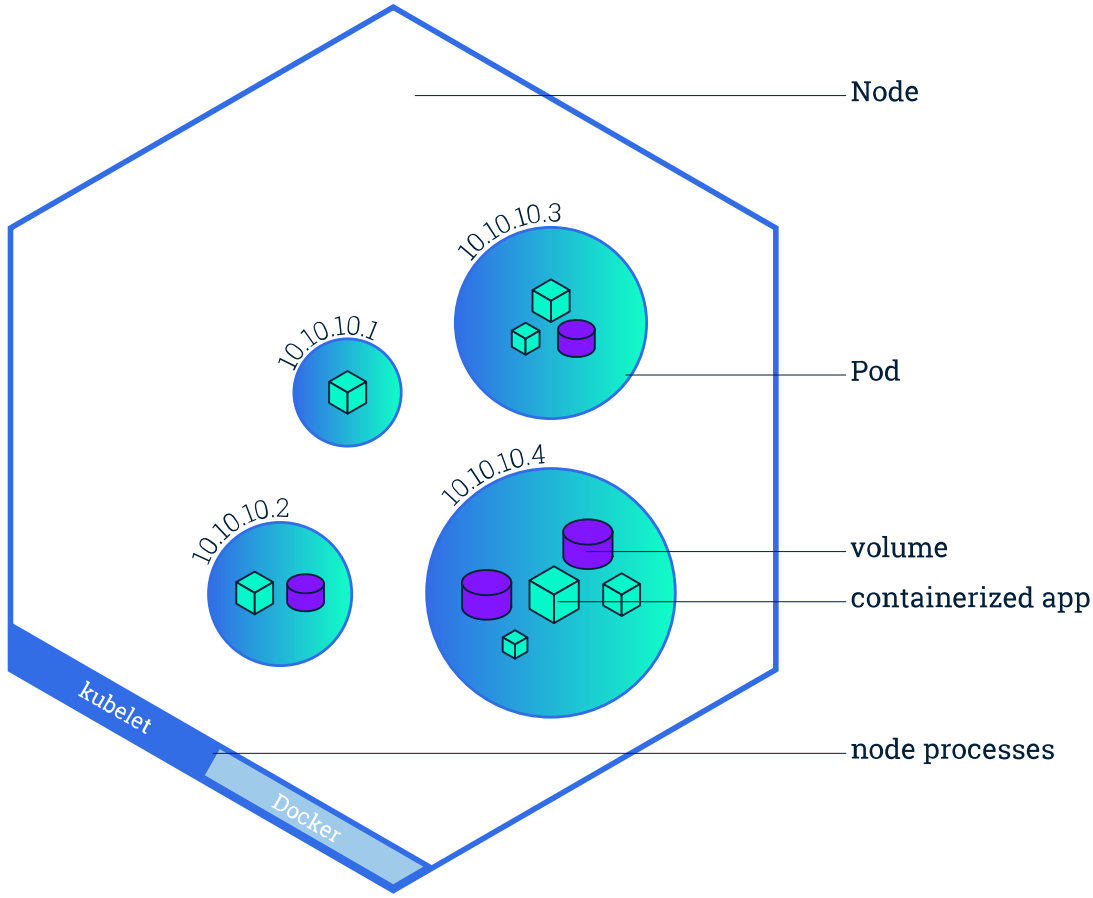
一个 pod 总是运行在 **工作节点**。工作节点是 Kubernetes 中的参与计算的机器，可以是虚拟机或物理计算机，具体取决于集群。每个工作节点由主节点管理。工作节点可以有多个 pod ，Kubernetes 主节点会自动处理在群集中的工作节点上调度 pod 。 主节点的自动调度考量了每个工作节点上的可用资源。

每个 Kubernetes 工作节点至少运行:

- Kubelet，负责 Kubernetes 主节点和工作节点之间通信的过程; 它管理 Pod 和机器上运行的容器。
- 容器运行时（如 Docker）负责从仓库中提取容器镜像，解压缩容器以及运行应用程序。

如果它们紧耦合并且需要共享磁盘等资源，这些容器应在一个 Pod 中编排。

工作节点概览



使用 kubectl 进行故障排除

在模块 2,您使用了 Kubectl 命令行界面。 您将继续在第3单元中使用它来获取有关已部署的应用程序及其环境的信息。 最常见的操作可以使用以下 kubectl 命令完成：

- **kubectl get** - 列出资源
- **kubectl describe** - 显示有关资源的详细信息
- **kubectl logs** - 打印 pod 和其中容器的日志
- **kubectl exec** - 在 pod 中的容器上执行命令

您可以使用这些命令查看应用程序的部署时间，当前状态，运行位置以及配置。

现在我们了解了有关集群组件和命令行的更多信息，让我们来探索一下我们的应用程序。

工作节点是 Kubernetes 中的负责计算的机器，可能是VM或物理计算机，具体取决于群集。多个 Pod 可以在一个工作节点上运行。

开始交互式教程 ›

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 2:37 PM PST: [zh] modify links to en-docs in tutorials to zh-docs (eba0555d1)

交互式教程-了解你的应用

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

公开地暴露你的应用

使用 Service 暴露您的应用

交互式教程 - 暴露你的应用

使用 Service 暴露您的应用

html

目标

- 了解 Kubernetes 中的 Service
- 了解 标签(Label) 和 标签选择器(Label Selector) 对象如何与 Service 关联
- 在 Kubernetes 集群外用 Service 暴露应用

Kubernetes Service 总览

Kubernetes Pod 是转瞬即逝的。 Pod 实际上拥有 生命周期。 当一个工作 Node 挂掉后, 在 Node 上运行的 Pod 也会消亡。 ReplicaSet 会自动地通过创建新的 Pod 驱动集群回到目标状态，以保证应用程序正常运行。 换一个例子，考虑一个具有3个副本数的用作图像处理的后端程序。这些副本是可替换的; 前端系统不应该关

心后端副本，即使 Pod 丢失或重新创建。也就是说，Kubernetes 集群中的每个 Pod (即使是在同一个 Node 上的 Pod)都有一个惟一的 IP 地址，因此需要一种方法自动协调 Pod 之间的变更，以便应用程序保持运行。

Kubernetes 中的服务(Service)是一种抽象概念，它定义了 Pod 的逻辑集和访问 Pod 的协议。Service 使从属 Pod 之间的松耦合成为可能。 和其他 Kubernetes 对象一样, Service 用 YAML (更推荐) 或者 JSON 来定义. Service 下的一组 Pod 通常由 *LabelSelector* (请参阅下面的说明为什么您可能想要一个 spec 中不包含selector的服务)来标记。

尽管每个 Pod 都有一个唯一的 IP 地址，但是如果没有 Service ，这些 IP 不会暴露在群集外部。Service 允许您的应用程序接收流量。Service 也可以用在 ServiceSpec 标记type的方式暴露

- *ClusterIP* (默认) - 在集群的内部 IP 上公开 Service 。这种类型使得 Service 只能从集群内访问。
- *NodePort* - 使用 NAT 在集群中每个选定 Node 的相同端口上公开 Service 。使用<NodeIP>:<NodePort> 从集群外部访问 Service。是 ClusterIP 的超集。
- *LoadBalancer* - 在当前云中创建一个外部负载均衡器(如果支持的话)，并为 Service 分配一个固定的外部IP。是 NodePort 的超集。
- *ExternalName* - 通过返回带有该名称的 CNAME 记录，使用任意名称(由 spec 中的externalName指定)公开 Service。不使用代理。这种类型需要kube-dns的v1.7或更高版本。

更多关于不同 Service 类型的信息可以在使用源 IP 教程。 也请参阅 连接应用程序和 Service 。

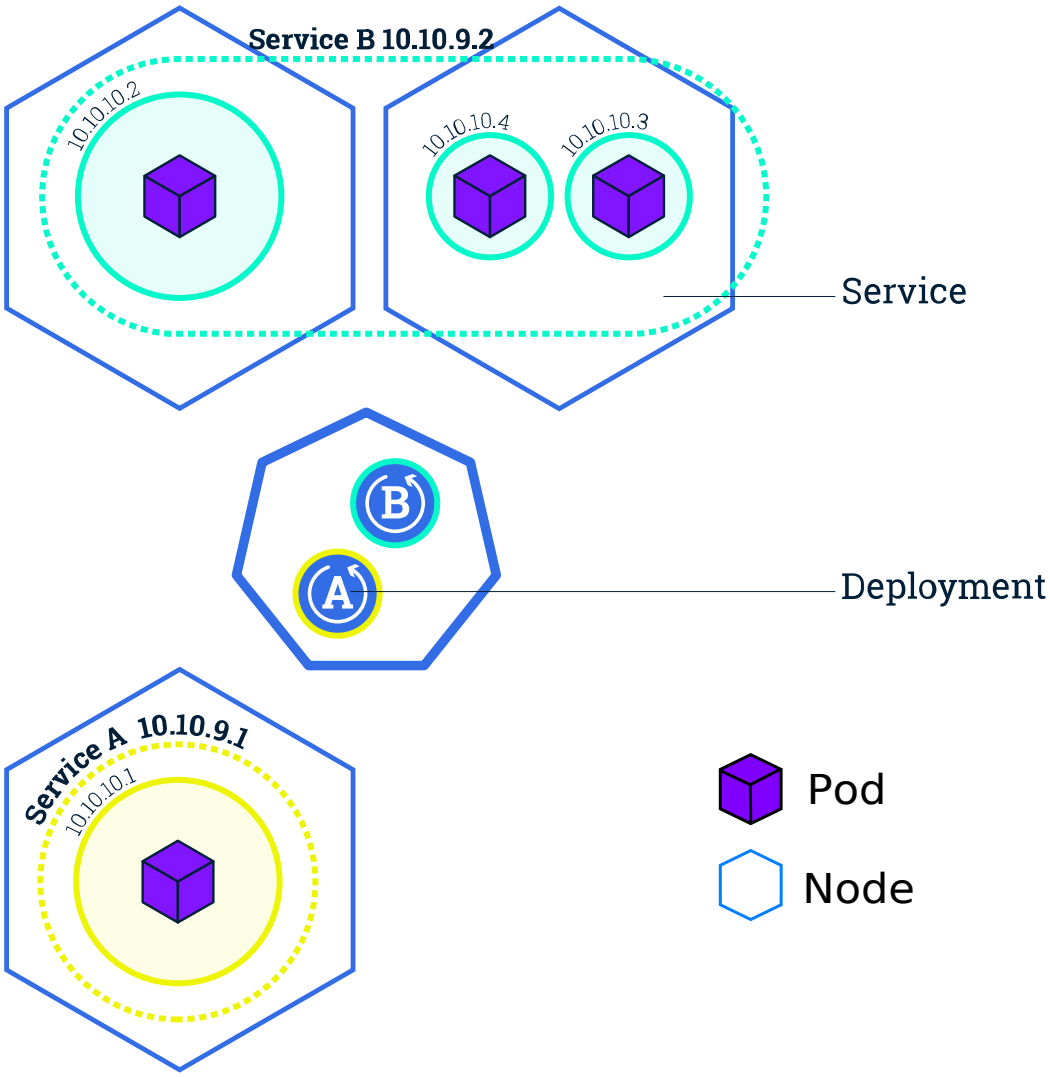
另外，需要注意的是有一些 Service 的用例没有在 spec 中定义selector。 一个没有selector创建的 Service 也不会创建相应的端点对象。这允许用户手动将服务映射到特定的端点。没有 selector 的另一种可能是您严格使用type: ExternalName 来标记。

总结

- 将 Pod 暴露给外部通信
- 跨多个 Pod 的负载均衡
- 使用标签(Label)

Kubernetes 的 Service 是一个抽象层，它定义了一组 Pod 的逻辑集，并为这些 Pod 支持外部流量暴露、负载平衡和服务发现。

Service 和 Label

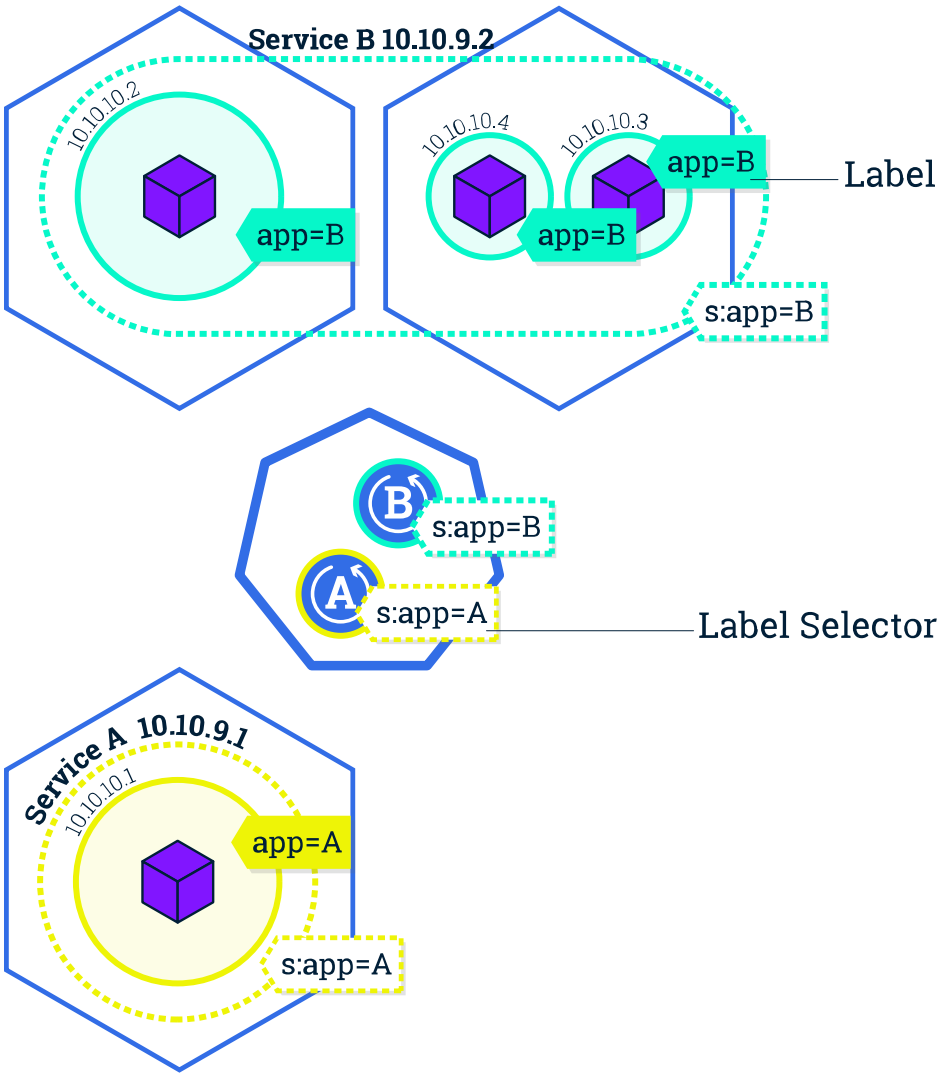


Service 通过一组 Pod 路由通信。Service 是一种抽象，它允许 Pod 死亡并在 Kubernetes 中复制，而不会影响应用程序。在依赖的 Pod (如应用程序中的前端和后端组件)之间进行发现和路由是由Kubernetes Service 处理的。

Service 匹配一组 Pod 是使用 标签(Label)和选择器(Selector), 它们是允许对 Kubernetes 中的对象进行逻辑操作的一种分组原语。标签(Label)是附加在对象上的键/值对，可以以多种方式使用:

- 指定用于开发，测试和生产的对象
- 嵌入版本标签
- 使用 Label 将对象进行分类

你也可以在创建 Deployment 的同时用 `--expose`创建一个 Service 。



标签(Label)可以在创建时或之后附加到对象上。他们可以随时被修改。现在使用 Service 发布我们的应用程序并添加一些 Label 。

开始交互式教程›

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 2:37 PM PST: [zh] modify links to en-docs in tutorials to zh-docs (eba0555d1)

交互式教程 - 暴露你的应用

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

缩放你的应用

运行应用程序的多个实例

html

目的

- 用 kubectl 扩缩应用程序

扩缩应用程序

在之前的模块中，我们创建了一个 Deployment，然后通过 Service让其可以开放访问。Deployment 仅为跑这个应用程序创建了一个 Pod。 当流量增加时，我们需要扩容应用程序满足用户需求。

扩缩 是通过改变 Deployment 中的副本数量来实现的。

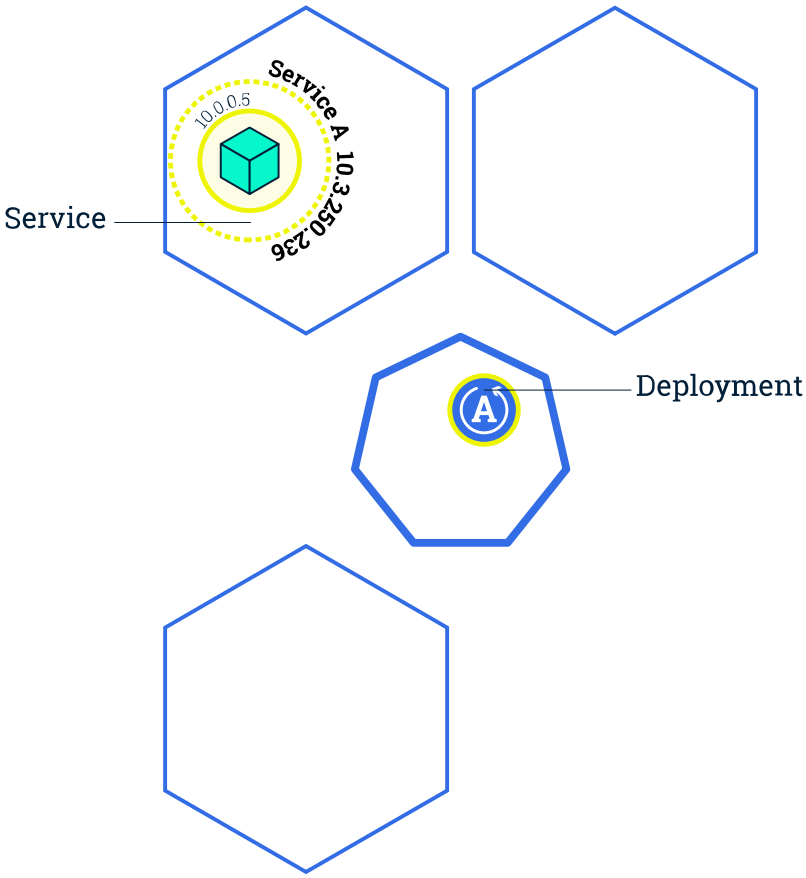
小结:

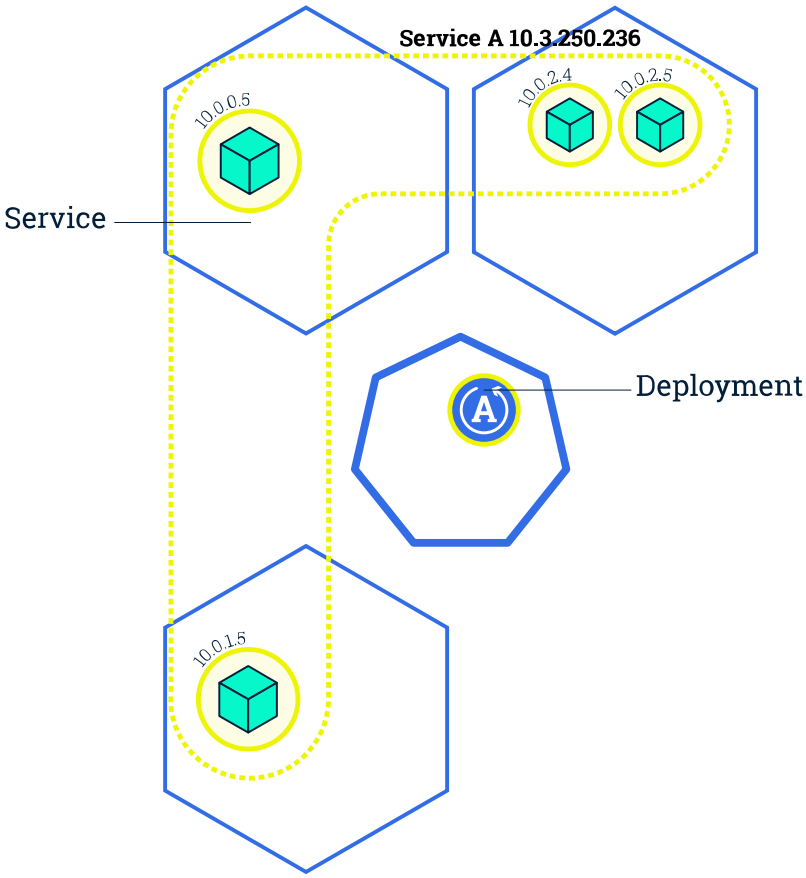
- 扩缩一个 Deployment

在运行 kubectl run 命令时，你可以通过设置 --replicas 参数来设置 Deployment 的副本数。

扩缩概述

1.
2.





Previous Next

扩展 Deployment 将创建新的 Pods，并将资源调度请求分配到有可用资源的节点上，收缩 会将 Pods 数量减少至所需的状态。Kubernetes 还支持 Pods 的自动缩放，但这并不在本教程的讨论范围内。将 Pods 数量收缩到0也是可以的，但这会终止 Deployment 上所有已经部署的 Pods。

运行应用程序的多个实例需要在它们之间分配流量。服务 (Service)有一种负载均衡器类型，可以将网络流量均衡分配到外部可访问的 Pods 上。服务将会一直通过端点来监视 Pods 的运行，保证流量只分配到可用的 Pods 上。

扩缩是通过改变 Deployment 中的副本数量来实现的。

一旦有了多个应用实例，就可以没有宕机地滚动更新。我们将会在下面的模块中介绍这些。现在让我们使用在线终端来体验一下应用程序的扩缩过程。

开始互动教程 ›

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 2:37 PM PST: [zh] modify links to en-docs in tutorials to zh-docs (eba0555d1)

交互教程 - 缩放你的应用

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

更新你的应用

执行滚动更新

交互式教程 - 更新你的应用

执行滚动更新

html

Objectives

- 使用 kubectl 执行滚动更新。

更新应用程序

用户希望应用程序始终可用，而开发人员则需要每天多次部署它们的新版本。在 Kubernetes 中，这些是通过滚动更新（Rolling Updates）完成的。 **滚动更新** 允许通过使用新的实例逐步更新 Pod 实例，零停机进行 Deployment 更新。新的 Pod 将在具有可用资源的节点上进行调度。

在前面的模块中，我们将应用程序扩展为运行多个实例。这是在不影响应用程序可用性的情况下执行更新的要求。默认情况下，更新期间不可用的 pod 的最大值和可以创建的新 pod 数都是 1。这两个选项都可以配置为（pod）数字或百分比。 在 Kubernetes 中，更新是经过版本控制的，任何 Deployment 更新都可以恢复到以前的（稳定）版本。

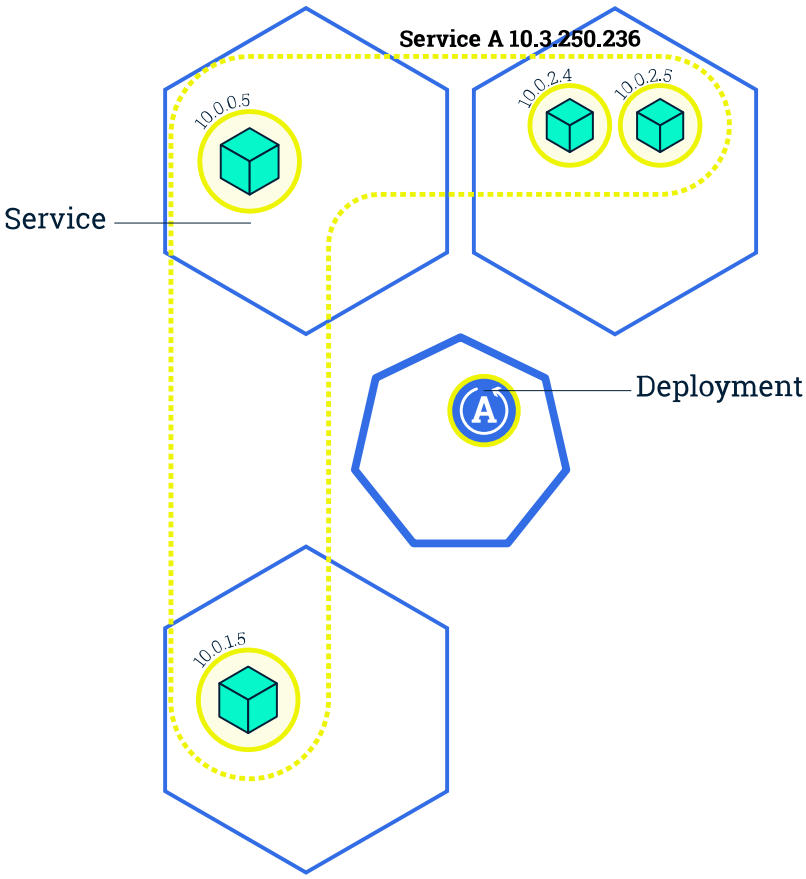
摘要：

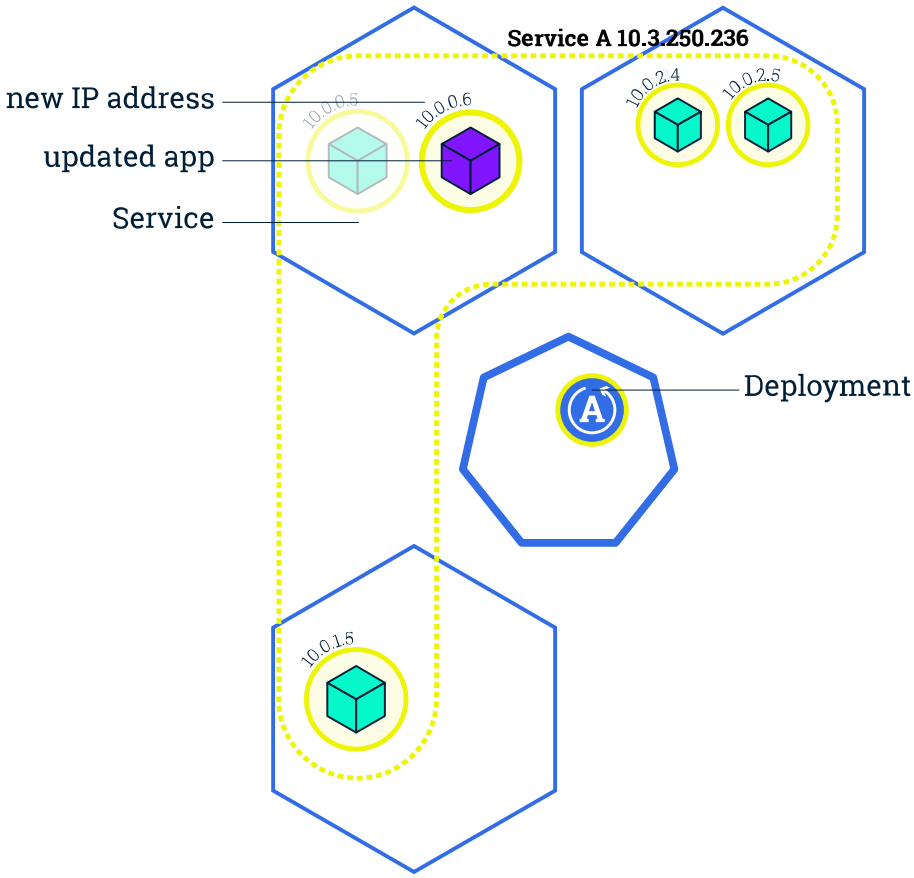
- 更新应用

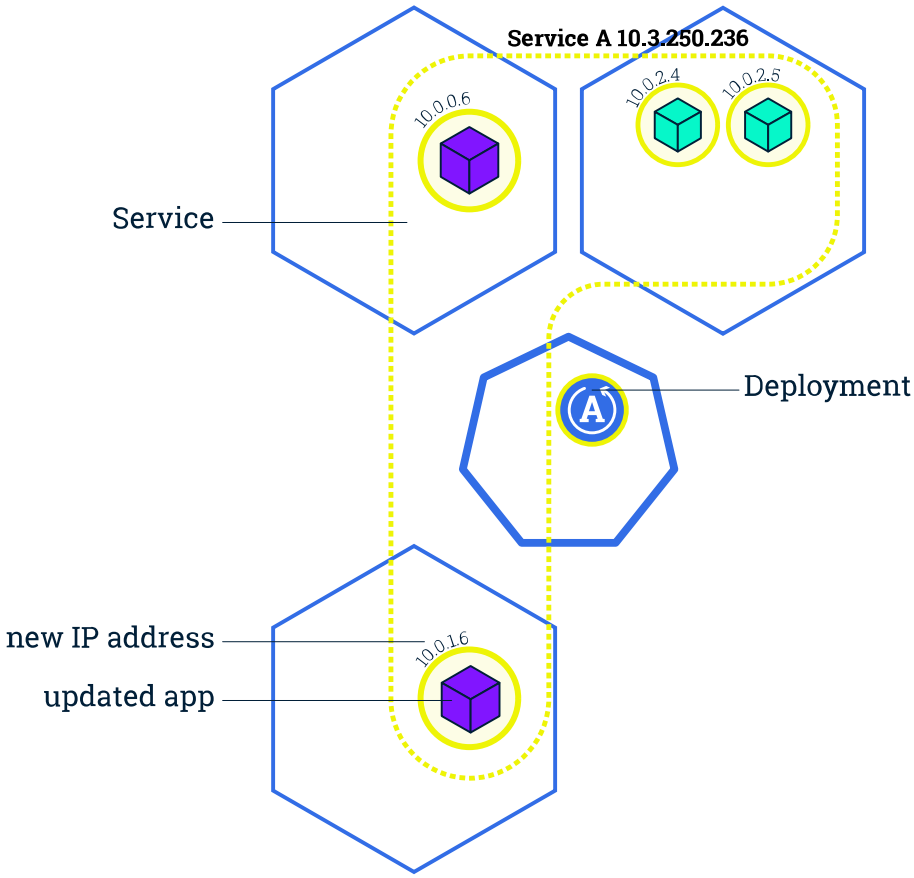
滚动更新允许通过使用新的实例逐步更新 Pod 实例从而实现 Deployments 更新，停机时间为零。

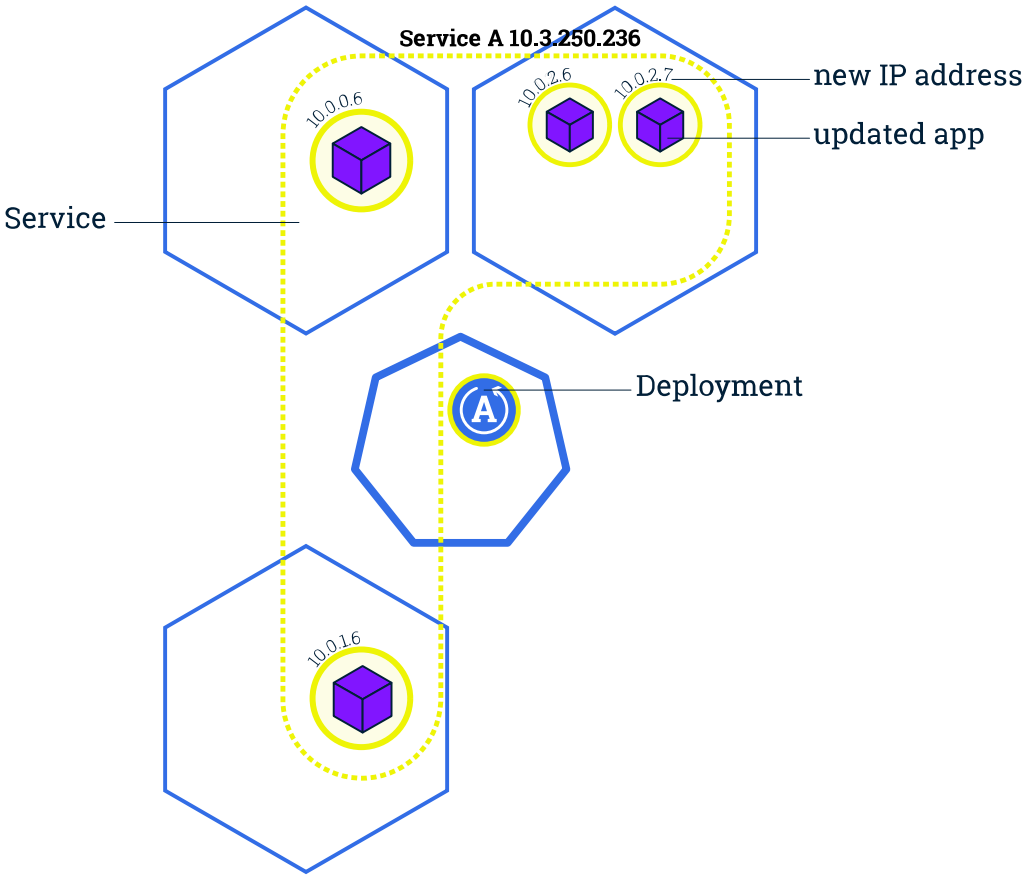
滚动更新概述

1.
2.
3.
4.




Service A 10.3.250.236
new IP address
updated app
Service
10.0.0.5
10.0.0.6
10.0.2.4
10.0.2.5
Deployment
10.0.1.5


Service A 10.3.250.236
10.0.0.6
10.0.2.4
10.0.2.5
Service
Deployment
new IP address
updated app
10.0.1.6



Previous Next

与应用程序扩展类似，如果公开了 Deployment，服务将在更新期间仅对可用的 pod 进行负载均衡。可用 Pod 是应用程序用户可用的实例。

滚动更新允许以下操作：

- 将应用程序从一个环境提升到另一个环境（通过容器镜像更新）
- 回滚到以前的版本
- 持续集成和持续交付应用程序，无需停机

如果 Deployment 是公开的，则服务将仅在更新期间对可用的 pod 进行负载均衡。

在下面的交互式教程中，我们将应用程序更新为新版本，并执行回滚。

启动交互教程›

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 04, 2020 at 8:05 PM PST: fix: zh doc missleading to en link. (3811e0c6e)

交互式教程 - 更新你的应用

html
最后修改 September 09, 2020 at 8:53 AM PST: Remove Roboto reference from interactive translated pages (3fa95f0c9)

配置

示例：配置 java 微服务

使用 ConfigMap 来配置 Redis

示例：配置 java 微服务

使用 MicroProfile、ConfigMaps、Secrets 实现外部化应用配置

互动教程 - 配置 java 微服务

使用 MicroProfile、ConfigMaps、Secrets 实现外部化应用配置

在本教程中，你会学到如何以及为什么要实现外部化微服务应用配置。 具体来说，你将学习如何使用 Kubernetes ConfigMaps 和 Secrets 设置环境变量， 然后在 MicroProfile config 中使用它们。

准备开始

创建 Kubernetes ConfigMaps 和 Secrets

在 Kubernetes 中，为 docker 容器设置环境变量有几种不同的方式，比如：Dockerfile、kubernetes.yml、Kubernetes ConfigMaps、和 Kubernetes Secrets。 在本教程中，你将学到怎么用后两个方式去设置你的环境变量，而环境变量的值将注入到你的微服务里。 使用 ConfigMaps 和 Secrets 的一个好处是他们能在多个容器间复用， 比如赋值给不同的容器中的不同环境变量。

ConfigMaps 是存储非机密键值对的 API 对象。 在互动教程中，你会学到如何用 ConfigMap 来保存应用名字。 ConfigMap 的更多信息，你可以在这里找到文档。

Secrets 尽管也用来存储键值对，但区别于 ConfigMaps 的是：它针对机密/敏感数据，且存储格式为 Base64 编码。 secrets 的这种特性使得它适合于存储证书、密钥、令牌，上述内容你将在交互教程中实现。 Secrets 的更多信息，你可以在这里找到文档。

从代码外部化配置

外部化应用配置之所以有用处，是因为配置常常根据环境的不同而变化。 为了实现此功能，我们用到了 Java 上下文和依赖注入（Contexts and Dependency Injection, CDI）、MicroProfile 配置。 MicroProfile config 是 MicroProfile 的功能特性， 是一组开放 Java 技术，用于开发、部署云原生微服务。

CDI 提供一套标准的依赖注入能力，使得应用程序可以由相互协作的、松耦合的 beans 组装而成。 MicroProfile Config 为 app 和微服务提供从各种来源，比如应用、运行时、环境，获取配置参数的标准方法。 基于来源定义的优先级，属性可以自动的合并到单独一组应用可以通过 API 访问到的属性。 CDI & MicroProfile 都会被用在互动教程中， 用来从 Kubernetes ConfigMaps 和 Secrets 获得外部提供的属性，并注入应用程序代码中。

很多开源框架、运行时支持 MicroProfile Config。 对于整个互动教程，你都可以使用开放的库、灵活的开源 Java 运行时，去构建并运行云原生的 apps 和微服务。 然而，任何 MicroProfile 兼容的运行时都可以用来做替代品。

教程目标

- 创建 Kubernetes ConfigMap 和 Secret
- 使用 MicroProfile Config 注入微服务配置

示例：使用 MicroProfile、ConfigMaps、Secrets 实现外部化应用配置

启动互动教程

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 15, 2020 at 11:21 PM PST: [zh] translate tutorials configure-java-microservice fix according to tengqm's comment (33a994238)

互动教程 - 配置 java 微服务

html
最后修改 November 15, 2020 at 11:21 PM PST: [zh] translate tutorials configure-java-microservice fix according to tengqm's comment (33a994238)

使用 ConfigMap 来配置 Redis

这篇文档基于使用 ConfigMap 来配置 Containers 这个任务，提供了一个使用 ConfigMap 来配置 Redis 的真实案例。

教程目标

- ■ 创建一个包含以下内容的 kustomization.yaml 文件：
  - ■ 一个 ConfigMap 生成器
  - ■ 一个使用 ConfigMap 的 Pod 资源配置
- 使用 kubectl apply -k ./ 应用整个路径的配置
- 验证配置已经被正确应用。

准备开始

- 你必须拥有一个 Kubernetes 的集群，同时你的 Kubernetes 集群必须带有 kubectl 命令行工具。 如果你还没有集群，你可以通过 Minikube 构建一 个你自己的集群，或者你可以使用下面任意一个 Kubernetes 工具构建：
  - Katacoda
  - 玩转 Kubernetes

  要获知版本信息，请输入 kubectl version.
- 此页面上显示的示例适用于 kubectl 1.14和在其以上的版本。
- 理解使用ConfigMap来配置Containers。

真实世界的案例：使用 ConfigMap 来配置 Redis

按照下面的步骤，您可以使用ConfigMap中的数据来配置Redis缓存。

1. 根据docs/user-guide/configmap/redis/redis-config来创建一个ConfigMap：

pods/config/redis-config

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

```
curl -OL https://k8s.io/examples/pods/config/redis-config

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-redis-config
  files:
  - redis-config
EOF
```

将 pod 的资源配置添加到 kustomization.yaml 文件中：

pods/config/redis-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis:5.0.4
    command:
```

```
    - redis-server
    - "/redis-master/redis.conf"
  env:
  - name: MASTER
    value: "true"
  ports:
  - containerPort: 6379
  resources:
    limits:
      cpu: "0.1"
  volumeMounts:
  - mountPath: /redis-master-data
    name: data
  - mountPath: /redis-master
    name: config
 volumes:
  - name: data
    emptyDir: {}
  - name: config
    configMap:
      name: example-redis-config
      items:
      - key: redis-config
        path: redis.conf
```

```
curl -OL https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/pods/config/redis-pod.yaml

cat <<EOF >>./kustomization.yaml
resources:
- redis-pod.yaml
EOF
```

应用整个 kustomization 文件夹以创建 ConfigMap 和 Pod 对象：

```
kubectl apply -k .
```

使用以下命令检查创建的对象

```
> kubectl get -k .
NAME                                    DATA   AGE
configmap/example-redis-config-dgh9dg555m   1      52s

NAME        READY   STATUS    RESTARTS   AGE
pod/redis   1/1     Running   0          52s
```

在示例中，配置卷挂载在 /redis-master 下。 它使用 path 将 redis-config 密钥添加到名为 redis.conf 的文件中。 因此，redis配置的文件路径为 /redis-master/redis.conf。 这是镜像将在其中查找 redis master 的配置文件的位置。

使用 kubectl exec 进入 pod 并运行 redis-cli 工具来验证配置已正确应用：

```
kubectl exec -it redis -- redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

删除创建的 pod：

```
kubectl delete pod redis
```

接下来

- 了解有关 ConfigMaps的更多信息。

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 22, 2020 at 10:46 PM PST: update exec command (ca71d9142)

无状态应用程序

公开外部 IP 地址以访问集群中应用程序

示例：使用 Redis 部署 PHP 留言板应用程序

示例: 添加日志和指标到 PHP / Redis Guestbook 案例

公开外部 IP 地址以访问集群中应用程序

此页面显示如何创建公开外部 IP 地址的 Kubernetes 服务对象。

准备开始

- 安装 kubectl.
- 使用 Google Kubernetes Engine 或 Amazon Web Services 等云供应商创建 Kubernetes 集群。 本教程创建了一个外部负载均衡器， 需要云供应商。
- 配置 kubectl 与 Kubernetes API 服务器通信。有关说明，请参阅云供应商文档。

教程目标

- 运行 Hello World 应用程序的五个实例。
- 创建一个公开外部 IP 地址的 Service 对象。
- 使用 Service 对象访问正在运行的应用程序。

为一个在五个 pod 中运行的应用程序创建服务

1. 在集群中运行 Hello World 应用程序：

service/load-balancer-example.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: load-balancer-example
  name: hello-world
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: load-balancer-example
  template:
    metadata:
      labels:
        app.kubernetes.io/name: load-balancer-example
    spec:
      containers:
      - image: gcr.io/google-samples/node-hello:1.0
        name: hello-world
```

```
    ports:
    - containerPort: 8080
```

```
kubectl apply -f https://k8s.io/examples/service/load-balancer-example.yaml
```

前面的命令创建一个 Deployment 对象和一个关联的 ReplicaSet 对象。ReplicaSet 有五个 Pods，每个都运行 Hello World 应用程序。

1. 显示有关 Deployment 的信息：

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

1. 显示有关 ReplicaSet 对象的信息：

```
kubectl get replicasets
kubectl describe replicasets
```

1. 创建公开 Deployment 的 Service 对象：

```
kubectl expose deployment hello-world --type=LoadBalancer --name=my-service
```

1. 显示有关 Service 的信息：

```
kubectl get services my-service
```

输出类似于：

```
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)    AGE
my-service   LoadBalancer   10.3.245.137   104.198.205.71   8080/TCP   54s
```

提示：type=LoadBalancer 服务由外部云服务提供商提供支持，本例中不包含此部分，详细信息请参考此页

提示：如果外部 IP 地址显示为 <pending>，请等待一分钟再次输入相同的命令。

1. 显示有关 Service 的详细信息：

```
kubectl describe services my-service
```

输出类似于：

```
Name:           my-service
Namespace:      default
Labels:         app.kubernetes.io/name=load-balancer-example
Annotations:    <none>
Selector:       app.kubernetes.io/name=load-balancer-example
```

```
Type:           LoadBalancer
IP:             10.3.245.137
LoadBalancer Ingress:   104.198.205.71
Port:           <unset> 8080/TCP
NodePort:       <unset> 32377/TCP
Endpoints:      10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...
Session Affinity:   None
Events:         <none>
```

记下服务公开的外部 IP 地址（LoadBalancer Ingress）。 在本例中，外部 IP 地址是 104.198.205.71。还要注意 Port 和 NodePort 的值。 在本例中，Port 是 8080，NodePort 是32377。

1. 在前面的输出中，您可以看到服务有几个端点： 10.0.0.6:8080、10.0.1.6:8080、10.0.1.7:8080 和另外两个， 这些都是正在运行 Hello World 应用程序的 pod 的内部地址。 要验证这些是 pod 地址，请输入以下命令：

```
kubectl get pods --output=wide
```

输出类似于：

```
NAME                     ...  IP         NODE
hello-world-2895499144-1jaz9 ...  10.0.1.6   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-2e5uh ...  10.0.1.8   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-9m4h1 ...  10.0.0.6   gke-cluster-1-default-pool-e0b8d269-5v7a
hello-world-2895499144-o4z13 ...  10.0.1.7   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-segjf ...  10.0.2.5   gke-cluster-1-default-pool-e0b8d269-cpuc
```

1. 使用外部 IP 地址（LoadBalancer Ingress）访问 Hello World 应用程序:

```
curl http://<external-ip>:<port>
```

其中 <external-ip> 是您的服务的外部 IP 地址（LoadBalancer Ingress），<port> 是您的服务描述中的 port 的值。 如果您正在使用 minikube，输入 minikube service my-service 将在浏览器中自动打开 Hello World 应用程序。

成功请求的响应是一条问候消息：

```
Hello Kubernetes!
```

清理现场

要删除服务，请输入以下命令：

```
kubectl delete services my-service
```

要删除正在运行 Hello World 应用程序的 Deployment，ReplicaSet 和 Pod，请输入以下命令：

```
kubectl delete deployment hello-world
```

接下来

进一步了解将应用程序与服务连接。

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 06, 2021 at 2:44 AM PST: update (04c997736)

示例：使用 Redis 部署 PHP 留言板应用程序

本教程向您展示如何使用 Kubernetes 和 Docker 构建和部署 一个简单的多层 web 应用程序。本例由以下组件组成：

- 单实例 Redis 主节点保存留言板条目
- 多个从 Redis 节点用来读取数据
- 多个 web 前端实例

教程目标

- 启动 Redis 主节点。
- 启动 Redis 从节点。
- 启动留言板前端。
- 公开并查看前端服务。
- 清理。

准备开始

你必须拥有一个 Kubernetes 的集群，同时你的 Kubernetes 集群必须带有 kubectl 命令行工具。 如果你还没有集群，你可以通过 Minikube 构建一 个你自己的集群，或者你可以使用下面任意一个 Kubernetes 工具构建：

- Katacoda
- 玩转 Kubernetes

要获知版本信息，请输入 kubectl version.

启动 Redis 主节点

留言板应用程序使用 Redis 存储数据。它将数据写入一个 Redis 主实例，并从多个 Redis 读取数据。

创建 Redis 主节点的 Deployment

下面包含的清单文件指定了一个 Deployment 控制器，该控制器运行一个 Redis 主节点 Pod 副本。

application/guestbook/redis-master-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: k8s.gcr.io/redis:e2e  # or just image: redis
```

```
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
    ports:
    - containerPort: 6379
```

1. 在下载清单文件的目录中启动终端窗口。
2. 从 redis-master-deployment.yaml 文件中应用 Redis 主 Deployment：

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-master-deployment.yaml
   ```

1. 查询 Pod 列表以验证 Redis 主节点 Pod 是否正在运行：

   ```
   kubectl get pods
   ```

响应应该与此类似：

```shell
NAME                           READY   STATUS    RESTARTS   AGE
redis-master-1068406935-3lswp  1/1     Running   0          28s
```

1. 运行以下命令查看 Redis 主节点 Pod 中的日志：

   ```
   kubectl logs -f POD-NAME
   ```

   说明：

   将 POD-NAME 替换为您的 Pod 名称。

创建 Redis 主节点的服务

留言板应用程序需要往 Redis 主节点中写数据。因此，需要创建 Service 来代理 Redis 主节点 Pod 的流量。Service 定义了访问 Pod 的策略。

application/guestbook/redis-master-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
    tier: backend
```

```
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

1. 使用下面的 redis-master-service.yaml 文件创建 Redis 主节点的服务：

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-master-service.yaml
   ```

1. 查询服务列表验证 Redis 主节点服务是否正在运行：

   ```
   kubectl get service
   ```

响应应该与此类似：

```shell
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     ClusterIP   10.0.0.1     <none>        443/TCP    1m
redis-master   ClusterIP   10.0.0.151   <none>        6379/TCP   8s
```

说明：

这个清单文件创建了一个名为 Redis-master 的 Service，其中包含一组与前面定义的标签匹配的标签，因此服务将网络流量路由到 Redis 主节点 Pod 上。

启动 Redis 从节点

尽管 Redis 主节点是一个单独的 pod，但是您可以通过添加 Redis 从节点的方式来使其高可用性，以满足流量需求。

创建 Redis 从节点 Deployment

Deployments 根据清单文件中设置的配置进行伸缩。在这种情况下，Deployment 对象指定两个副本。

如果没有任何副本正在运行，则此 Deployment 将启动容器集群上的两个副本。相反， 如果有两个以上的副本在运行，那么它的规模就会缩小，直到运行两个副本为止。

application/guestbook/redis-slave-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-slave
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your cluster to
          # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
          # service launched automatically. However, if the cluster you are using
          # does not have a built-in DNS service, you can instead
          # access an environment variable to find the master
          # service's host. To do so, comment out the 'value: dns' line above,
and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 6379
```

1. 从 redis-slave-deployment.yaml 文件中应用 Redis Slave Deployment：

```
kubectl apply -f https://k8s.io/examples/application/guestbook/redis-slave-deployment.yaml
```

1. 查询 Pod 列表以验证 Redis Slave Pod 正在运行：

```
kubectl get pods
```

响应应该与此类似：

```shell
NAME                           READY     STATUS              RESTARTS   AGE
redis-master-1068406935-3lswp  1/1       Running             0          1m
redis-slave-2005841000-fpvqc   0/1       ContainerCreating   0          6s
redis-slave-2005841000-phfv9   0/1       ContainerCreating   0          6s
```

创建 Redis 从节点的 Service

留言板应用程序需要从 Redis 从节点中读取数据。 为了便于 Redis 从节点可发现， 您需要设置一个 Service。Service 为一组 Pod 提供负载均衡。

application/guestbook/redis-slave-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

1. 从以下 redis-slave-service.yaml 文件应用 Redis Slave 服务：

```
kubectl apply -f https://k8s.io/examples/application/guestbook/redis-slave-service.yaml
```

1. 查询服务列表以验证 Redis 在服务是否正在运行：

```
kubectl get services
```

响应应该与此类似：

```
NAME          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes    ClusterIP   10.0.0.1     <none>        443/TCP    2m
redis-master  ClusterIP   10.0.0.151   <none>        6379/TCP   1m
redis-slave   ClusterIP   10.0.0.223   <none>        6379/TCP   6s
```

设置并公开留言板前端

留言板应用程序有一个 web 前端，服务于用 PHP 编写的 HTTP 请求。 它被配置为连接到写请求的 redis-master 服务和读请求的 redis-slave 服务。

创建留言板前端 Deployment

application/guestbook/frontend-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
```

```
      - name: GET_HOSTS_FROM
        value: dns
        # Using `GET_HOSTS_FROM=dns` requires your cluster to
        # provide a dns service. As of Kubernetes 1.3, DNS is a built-in
        # service launched automatically. However, if the cluster you are using
        # does not have a built-in DNS service, you can instead
        # access an environment variable to find the master
        # service's host. To do so, comment out the 'value: dns' line above,
and
        # uncomment the line below:
        # value: env
      ports:
      - containerPort: 80
```

1. 从 frontend-deployment.yaml 应用前端 Deployment 文件：

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

1. 查询 Pod 列表，验证三个前端副本是否正在运行：

   ```
   kubectl get pods -l app=guestbook -l tier=frontend
   ```

响应应该与此类似：

```
NAME                       READY     STATUS    RESTARTS   AGE
frontend-3823415956-dsvc5  1/1       Running   0          54s
frontend-3823415956-k22zn  1/1       Running   0          54s
frontend-3823415956-w9gbt  1/1       Running   0          54s
```

创建前端服务

应用的 redis-slave 和 redis-master 服务只能在容器集群中访问，因为服务的默认类型是 ClusterIP。ClusterIP 为服务指向的 Pod 集提供一个 IP 地址。这个 IP 地址只能在集群中访问。

如果您希望客人能够访问您的留言板，您必须将前端服务配置为外部可见的，以便客户机可以从容器集群之外请求服务。Minikube 只能通过 NodePort 公开服务。

> **说明：**
>
> 一些云提供商，如 Google Compute Engine 或 Google Kubernetes Engine，支持外部负载均衡器。如果您的云提供商支持负载均衡器，并且您希望使用它， 只需删除或注释掉 type: NodePort，并取消注释 type: LoadBalancer 即可。

application/guestbook/frontend-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # comment or delete the following line if you want to use a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to automatically
create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

1. 从 frontend-service.yaml 文件中应用前端服务：

```
kubectl apply -f https://k8s.io/examples/application/guestbook/
frontend-service.yaml
```

1. 查询服务列表以验证前端服务正在运行:

```
kubectl get services
```

响应应该与此类似：

```
NAME           TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)       AGE
frontend       NodePort    10.0.0.112    <none>        80:31323/TCP  6s
kubernetes     ClusterIP   10.0.0.1      <none>        443/TCP       4m
redis-master   ClusterIP   10.0.0.151    <none>        6379/TCP      2m
redis-slave    ClusterIP   10.0.0.223    <none>        6379/TCP      1m
```

通过 NodePort 查看前端服务

如果您将此应用程序部署到 Minikube 或本地集群，您需要找到 IP 地址来查看您的留言板。

1. 运行以下命令获取前端服务的 IP 地址。

   ```
   minikube service frontend --url
   ```

响应应该与此类似：

```
http://192.168.99.100:31323
```

1. 复制 IP 地址，然后在浏览器中加载页面以查看留言板。

通过 LoadBalancer 查看前端服务

如果您部署了 frontend-service.yaml。你需要找到 IP 地址来查看你的留言板。

1. 运行以下命令以获取前端服务的 IP 地址。

   ```
   kubectl get service frontend
   ```

响应应该与此类似：

```
NAME       TYPE        CLUSTER-IP      EXTERNAL-IP       PORT(S)        AGE
frontend   ClusterIP   10.51.242.136   109.197.92.229    80:32372/TCP   1m
```

1. 复制外部 IP 地址，然后在浏览器中加载页面以查看留言板。

扩展 Web 前端

伸缩很容易是因为服务器本身被定义为使用一个 Deployment 控制器的 Service。

1. 运行以下命令扩展前端 Pod 的数量：

   ```
   kubectl scale deployment frontend --replicas=5
   ```

1. 查询 Pod 列表验证正在运行的前端 Pod 的数量：

   ```
   kubectl get pods
   ```

响应应该类似于这样：

```
NAME                          READY     STATUS    RESTARTS   AGE
frontend-3823415956-70qj5     1/1       Running   0          5s
frontend-3823415956-dsvc5     1/1       Running   0          54m
frontend-3823415956-k22zn     1/1       Running   0          54m
frontend-3823415956-w9gbt     1/1       Running   0          54m
frontend-3823415956-x2pld     1/1       Running   0          5s
redis-master-1068406935-3lswp   1/1       Running   0          56m
redis-slave-2005841000-fpvqc    1/1       Running   0          55m
redis-slave-2005841000-phfv9    1/1       Running   0          55m
```

1. 运行以下命令缩小前端 Pod 的数量：

   ```
   kubectl scale deployment frontend --replicas=2
   ```

1. 查询 Pod 列表验证正在运行的前端 Pod 的数量：

   ```
   kubectl get pods
   ```

响应应该类似于这样：

```
NAME                          READY     STATUS    RESTARTS   AGE
frontend-3823415956-k22zn     1/1       Running   0          1h
frontend-3823415956-w9gbt     1/1       Running   0          1h
redis-master-1068406935-3lswp   1/1       Running   0          1h
redis-slave-2005841000-fpvqc    1/1       Running   0          1h
redis-slave-2005841000-phfv9    1/1       Running   0          1h
```

清理现场

删除 Deployments 和服务还会删除正在运行的 Pod。使用标签用一个命令删除多个资源。

1. 运行以下命令以删除所有 Pod，Deployments 和 Services。

   ```
   kubectl delete deployment -l app=redis
   kubectl delete service -l app=redis
   kubectl delete deployment -l app=guestbook
   kubectl delete service -l app=guestbook
   ```

响应应该是：

```
deployment.apps "redis-master" deleted
deployment.apps "redis-slave" deleted
```

```
service "redis-master" deleted
service "redis-slave" deleted
deployment.apps "frontend" deleted
service "frontend" deleted
```

1. 查询 Pod 列表，确认没有 Pod 在运行：

   ```
   kubectl get pods
   ```

响应应该是：

```
No resources found.
```

接下来

- 为 Guestbook 应用添加 ELK 日志与监控
- 完成 Kubernetes Basics 交互式教程
- 使用 Kubernetes 创建一个博客，使用 MySQL 和 Wordpress 的持久卷
- 阅读更多关于连接应用程序
- 阅读更多关于管理资源

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 05, 2021 at 4:36 AM PST: output is incorret. (9a80a9e36)

示例: 添加日志和指标到 PHP / Redis Guestbook 案例

本教程建立在 使用 Redis 部署 PHP Guestbook 教程之上。 *Beats*，是 Elastic 出品的开源的轻量级日志、指标和网络数据采集器， 将和 Guestbook 一同部署在 Kubernetes 集群中。 Beats 收集、分析、索引数据到 Elasticsearch，使你可以用 Kibana 查看并分析得到的运营信息。 本示例由以下内容组成：

- 带 Redis 的 PHP Guestbook 教程 的一个实例部署
- Elasticsearch 和 Kibana

- Filebeat
- Metricbeat
- Packetbeat

教程目标

- 启动用 Redis 部署的 PHP Guestbook。
- 安装 kube-state-metrics。
- 创建 Kubernetes secret。
- 部署 Beats。
- 用仪表板查看日志和指标。

准备开始

你必须拥有一个 Kubernetes 的集群，同时你的 Kubernetes 集群必须带有 kubectl 命令行工具。 如果你还没有集群，你可以通过 Minikube 构建一 个你自己的集群，或者你可以使用下面任意一个 Kubernetes 工具构建：

- Katacoda
- 玩转 Kubernetes

要获知版本信息，请输入 `kubectl version`.

此外，你还需要:

- 依照教程使用 Redis 的 PHP Guestbook得到的一套运行中的部署环境。
- 一套运行中的 Elasticsearch 和 Kibana 部署环境。你可以使用 Elastic 云中的Elasticsearch 服务、在工作站或者服务器上运行此下载文件、或运行 Elastic Helm Charts。

启动用 Redis 部署的 PHP Guestbook

本教程建立在 使用 Redis 部署 PHP Guestbook 之上。 如果你已经有一个运行的 Guestbook 应用程序，那就监控它。 如果还没有，那就按照说明先部署 Guestbook ，但不要执行**清理**的步骤。 当 Guestbook 运行起来后，再返回本页。

添加一个集群角色绑定

创建一个集群范围的角色绑定， 以便你可以在集群范围（在 kube-system 中）部署 kube-state-metrics 和 Beats。

```
kubectl create clusterrolebinding cluster-admin-binding \
 --clusterrole=cluster-admin --user=<your email associated with the k8s provider account>
```

安装 kube-state-metrics

Kubernetes *kube-state-metrics* 是一个简单的服务，它侦听 Kubernetes API 服务器并生成对象状态的指标。 Metricbeat 报告这些指标。 添加 kube-state-metrics 到运行 Guestbook 的 Kubernetes 集群。

```
git clone https://github.com/kubernetes/kube-state-metrics.git kube-state-metrics
kubectl apply -f kube-state-metrics/examples/standard
```

检查 kube-state-metrics 是否正在运行

```
kubectl get pods --namespace=kube-system -l app.kubernetes.io/name=kube-state-metrics
```

输出：

```
NAME                                  READY   STATUS    RESTARTS   AGE
kube-state-metrics-89d656bf8-vdthm   1/1     Running   0          21s
```

从 GitHub 克隆 Elastic examples 库

```
git clone https://github.com/elastic/examples.git
```

后续命令将引用目录 examples/beats-k8s-send-anywhere 中的文件， 所以把目录切换过去。

```
cd examples/beats-k8s-send-anywhere
```

创建 Kubernetes Secret

Kubernetes Secret 是包含少量敏感数据（类似密码、令牌、秘钥等）的对象。 这类信息也可以放在 Pod 规格定义或者镜像中； 但放在 Secret 对象中，能更好的控制它的使用方式，也能减少意外泄露的风险。

> **说明：** 这里有两套步骤，一套用于*自管理*的 Elasticsearch 和 Kibana（运行在你的服务器上或使用 Helm Charts）， 另一套用于在 Elastic 云服务中 *Managed service* 的 Elasticsearch 服务。 在本教程中，只需要为 Elasticsearch 和 Kibana 系统创建 secret。

- 自管理
- Managed service

自管理系统

如果你使用 Elastic 云中的 Elasticsearch 服务，切换到 **Managed service** 标签页。

设置凭据

当你使用自管理的 Elasticsearch 和 Kibana（对比托管于 Elastic 云中的 Elasticsearch 服务，自管理更有效率），创建 k8s secret 需要准备四个文件。这些文件是：

1. ELASTICSEARCH_HOSTS
2. ELASTICSEARCH_PASSWORD
3. ELASTICSEARCH_USERNAME
4. KIBANA_HOST

为你的 Elasticsearch 集群和 Kibana 主机设置这些信息。这里是一些例子（另见*此配置*）

ELASTICSEARCH_HOSTS

1. 来自于 Elastic Elasticsearch Helm Chart 的节点组：

   ```
   ["http://elasticsearch-master.default.svc.cluster.local:9200"]
   ```

2. Mac 上的单节点的 Elasticsearch，Beats 运行在 Mac 的容器中：

   ```
   ["http://host.docker.internal:9200"]
   ```

3. 运行在虚拟机或物理机上的两个 Elasticsearch 节点

   ```
   ["http://host1.example.com:9200", "http://host2.example.com:9200"]
   ```

编辑 ELASTICSEARCH_HOSTS

```
vi ELASTICSEARCH_HOSTS
```

ELASTICSEARCH_PASSWORD

只有密码；没有空格、引号、< 和 >：

```
<yoursecretpassword>
```

编辑 ELASTICSEARCH_PASSWORD：

```
vi ELASTICSEARCH_PASSWORD
```

ELASTICSEARCH_USERNAME

只有用名；没有空格、引号、< 和 >：

```
<为 Elasticsearch 注入的用户名>
```

编辑 ELASTICSEARCH_USERNAME：

```
vi ELASTICSEARCH_USERNAME
```

KIBANA_HOST

1. 从 Elastic Kibana Helm Chart 安装的 Kibana 实例。子域 `default` 指默认的命名空间。如果你把 Helm Chart 指定部署到不同的命名空间，那子域会不同：

   ```
   "kibana-kibana.default.svc.cluster.local:5601"
   ```

2. Mac 上的 Kibana 实例，Beats 运行于 Mac 的容器：

   ```
   "host.docker.internal:5601"
   ```

3. 运行于虚拟机或物理机上的两个 Elasticsearch 节点：

   ```
   "host1.example.com:5601"
   ```

编辑 KIBANA_HOST：

```
vi KIBANA_HOST
```

创建 Kubernetes secret

在上面编辑完的文件的基础上，本命令在 Kubernetes 系统范围的命名空间（kube-system）创建一个 secret。

```
kubectl create secret generic dynamic-logging \
  --from-file=./ELASTICSEARCH_HOSTS \
  --from-file=./ELASTICSEARCH_PASSWORD \
  --from-file=./ELASTICSEARCH_USERNAME \
  --from-file=./KIBANA_HOST \
  --namespace=kube-system
```

Managed service

本标签页只用于 Elastic 云 的 Elasticsearch 服务，如果你已经为自管理的 Elasticsearch 和 Kibana 创建了secret，请继续部署 Beats并继续。

设置凭据

在 Elastic 云中的托管 Elasticsearch 服务中，为了创建 k8s secret，你需要先编辑两个文件。它们是：

1. ELASTIC_CLOUD_AUTH
2. ELASTIC_CLOUD_ID

当你完成部署的时候，Elasticsearch 服务控制台会提供给你一些信息，用这些信息完成设置。 这里是一些示例：

ELASTIC_CLOUD_ID

```
devk8s:ABC123def456ghi789jkl123mno456pqr789stu123vwx456yza789bcd012efg345hijj678klm901nop345zEwOTJjMTc5YWQ0YzQ5OThlN2U5MjAwYTg4NTIzZQ==
```

ELASTIC_CLOUD_AUTH

只要用户名；没有空格、引号、< 和 >：

```
elastic:VFxJJf9Tjwer90wnfTghsn8w
```

编辑要求的文件

```
vi ELASTIC_CLOUD_ID
vi ELASTIC_CLOUD_AUTH
```

创建 Kubernetes secret

基于上面刚编辑过的文件，在 Kubernetes 系统范围命名空间（kube-system）中，用下面命令创建一个的secret：

```
kubectl create secret generic dynamic-logging \
  --from-file=./ELASTIC_CLOUD_ID \
  --from-file=./ELASTIC_CLOUD_AUTH \
  --namespace=kube-system
```

部署 Beats

为每一个 Beat 提供 清单文件。清单文件使用已创建的 secret 接入 Elasticsearch 和 Kibana 服务器。

关于 Filebeat

Filebeat 收集日志，日志来源于 Kubernetes 节点以及这些节点上每一个 Pod 中的容器。Filebeat 部署为 DaemonSet。 Filebeat 支持自动发现 Kubernetes 集群中的应用。 在启动时，Filebeat 扫描存量的容器，并为它们提供适当的配置， 然后开始监听新的启动/中止信号。

下面是一个自动发现的配置，它支持 Filebeat 定位并分析来自于 Guestbook 应用部署的 Redis 容器的日志文件。 下面的配置片段来自文件 filebeat-kubernetes.yaml：

```
- condition.contains:
    kubernetes.labels.app: redis
  config:
    - module: redis
      log:
```

```
    input:
      type: docker
      containers.ids:
        - ${data.kubernetes.container.id}
  slowlog:
    enabled: true
    var.hosts: ["${data.host}:${data.port}"]
```

这样配置 Filebeat，当探测到容器拥有 app 标签，且值为 redis，那就启用 Filebeat 的 redis 模块。 redis 模块可以根据 docker 的输入类型（在 Kubernetes 节点上读取和 Redis 容器的标准输出流关联的文件），从容器收集 log 流。 另外，此模块还可以使用容器元数据中提供的配置信息，连到 Pod 适当的主机和端口，收集 Redis 的 slowlog 。

部署 Filebeat

```
kubectl create -f filebeat-kubernetes.yaml
```

验证

```
kubectl get pods -n kube-system -l k8s-app=filebeat-dynamic
```

关于 Metricbeat

Metricbeat 自动发现的配置方式与 Filebeat 完全相同。 这里是针对 Redis 容器的 Metricbeat 自动发现配置。 此配置片段来自于文件 metricbeat-kubernetes.yaml：

```
- condition.equals:
    kubernetes.labels.tier: backend
  config:
    - module: redis
      metricsets: ["info", "keyspace"]
      period: 10s

      # Redis hosts
      hosts: ["${data.host}:${data.port}"]
```

配置 Metricbeat，在探测到标签 tier 的值等于 backend 时，应用 Metricbeat 模块 redis。 redis 模块可以获取容器元数据，连接到 Pod 适当的主机和端口，从 Pod 中收集指标 info 和 keyspace。

部署 Metricbeat

```
kubectl create -f metricbeat-kubernetes.yaml
```

验证

```
kubectl get pods -n kube-system -l k8s-app=metricbeat
```

关于 Packetbeat

Packetbeat 的配置方式不同于 Filebeat 和 Metricbeat。 相比于匹配容器标签的模式，它的配置基于相关协议和端口号。 下面展示的是端口号的一个子集：

> **说明：** 如果你的服务运行在非标准的端口上，那就打开文件 filebeat.yaml，把这个端口号添加到合适的类型中，然后删除/启动 Packetbeat 的守护进程。

```
packetbeat.interfaces.device: any

packetbeat.protocols:
- type: dns
  ports: [53]
  include_authorities: true
  include_additionals: true

- type: http
  ports: [80, 8000, 8080, 9200]

- type: mysql
  ports: [3306]

- type: redis
  ports: [6379]

packetbeat.flows:
  timeout: 30s
  period: 10s
```

部署 Packetbeat

```
kubectl create -f packetbeat-kubernetes.yaml
```

验证

```
kubectl get pods -n kube-system -l k8s-app=packetbeat-dynamic
```

在 kibana 中浏览

在浏览器中打开 kibana，再打开 **Dashboard。** 在搜索栏中键入 Kubernetes，再点击 Metricbeat 的 Kubernetes Dashboard。 此 Dashboard 展示节点状态、应用部署等。

在 Dashboard 页面，搜索 Packetbeat，并浏览 Packetbeat 概览信息。

同样地，浏览 Apache 和 Redis 的 Dashboard。 可以看到日志和指标各自独立 Dashboard。 Apache Metricbeat Dashboard 是空的。 找到 Apache Filebeat Dashboard，拉到最下面，查看 Apache 的错误日志。 日志会揭示出没有 Apache 指标的原因。

要让 metricbeat 得到 Apache 的指标，需要添加一个包含模块状态配置文件的 ConfigMap，并重新部署 Guestbook。

缩放部署规模，查看新 Pod 已被监控

列出现有的 deployments：

```
kubectl get deployments
```

输出：

```
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
frontend       3/3     3            3           3h27m
redis-master   1/1     1            1           3h27m
redis-slave    2/2     2            2           3h27m
```

缩放前端到两个 Pod：

```
kubectl scale --replicas=2 deployment/frontend
```

输出：

```
deployment.extensions/frontend scaled
```

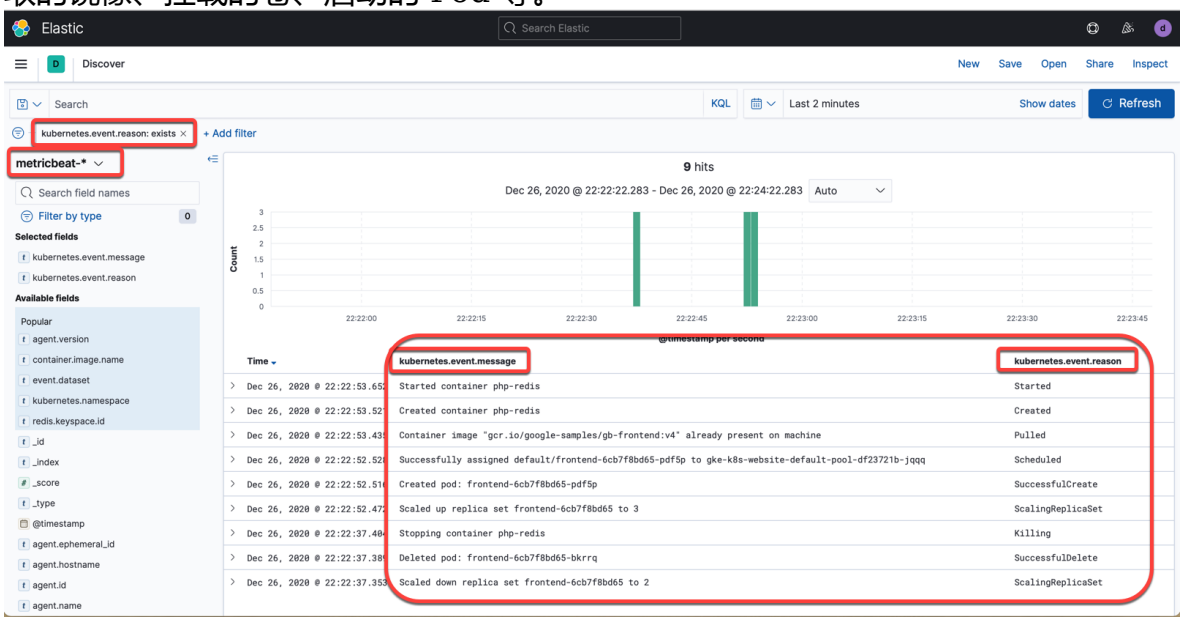
将前端应用缩放回三个 Pod：

```
kubectl scale --replicas=3 deployment/frontend
```

在 Kibana 中查看变化

参见屏幕截图，添加指定的过滤器，然后将列添加到视图。 你可以看到，ScalingReplicaSet 被做了标记，从标记的点开始，到消息列表的顶部，展示了拉

取的镜像、挂载的卷、启动的 Pod 等。



清理现场

删除 Deployments 和 Services， 删除运行的 Pod。 用标签功能在一个命令中删除多个资源。

1. 执行下列命令，删除所有的 Pod、Deployment 和 Services。

   ```
   kubectl delete deployment -l app=redis
   kubectl delete service -l app=redis
   kubectl delete deployment -l app=guestbook
   kubectl delete service -l app=guestbook
   kubectl delete -f filebeat-kubernetes.yaml
   kubectl delete -f metricbeat-kubernetes.yaml
   kubectl delete -f packetbeat-kubernetes.yaml
   kubectl delete secret dynamic-logging -n kube-system
   ```

2. 查询 Pod，以核实没有 Pod 还在运行：

   ```
   kubectl get pods
   ```

   响应应该是这样：

   ```
   No resources found.
   ```

接下来

- 了解监控资源的工具
- 进一步阅读日志体系架构
- 进一步阅读应用内省和调试
- 进一步阅读应用程序的故障排除

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 10, 2020 at 7:32 PM PST: Update guestbook-logs-metrics-with-elk.md (#25463) (ed649646d)

有状态的应用

示例：使用 Persistent Volumes 部署 WordPress 和 MySQL

本示例描述了如何通过 Minikube 在 Kubernetes 上安装 WordPress 和 MySQL。这两个应用都使用 PersistentVolumes 和 PersistentVolumeClaims 保存数据。

PersistentVolume（PV）是一块集群里由管理员手动提供，或 kubernetes 通过 StorageClass 动态创建的存储。 PersistentVolumeClaim（PVC）是一个满足对 PV 存储需要的请求。PersistentVolumes 和 PersistentVolumeClaims 是独立于 Pod 生命周期而在 Pod 重启，重新调度甚至删除过程中保存数据。

> **警告：**
>
> deployment 在生产场景中并不适合，它使用单实例 WordPress 和 MySQL Pods。考虑使用 WordPress Helm Chart 在生产场景中部署 WordPress。
>
> **说明：**
>
> 本教程中提供的文件使用 GA Deployment API，并且特定于 kubernetes 1.9 或更高版本。如果您希望将本教程与 Kubernetes 的

早期版本一起使用，请相应地更新 API 版本，或参考本教程的早期版本。

教程目标

- 创建 PersistentVolumeClaims 和 PersistentVolumes
- 创建 kustomization.yaml 使用
  - Secret 生成器
  - MySQL 资源配置
  - WordPress 资源配置
- 应用整个 kustomization 目录 `kubectl apply -k ./`
- 清理

准备开始

你必须拥有一个 Kubernetes 的集群，同时你的 Kubernetes 集群必须带有 kubectl 命令行工具。 如果你还没有集群，你可以通过 Minikube 构建一 个你自己的集群，或者你可以使用下面任意一个 Kubernetes 工具构建：

- Katacoda
- 玩转 Kubernetes

要获知版本信息，请输入 `kubectl version`.

此例在kubectl 1.14 或者更高版本有效。

下载下面的配置文件：

1. mysql-deployment.yaml
2. wordpress-deployment.yaml

创建 PersistentVolumeClaims 和 PersistentVolumes

MySQL 和 Wordpress 都需要一个 PersistentVolume 来存储数据。他们的 PersistentVolumeClaims 将在部署步骤中创建。

许多群集环境都安装了默认的 StorageClass。如果在 PersistentVolumeClaim 中未指定 StorageClass，则使用群集的默认 StorageClass。

创建 PersistentVolumeClaim 时，将根据 StorageClass 配置动态设置 PersistentVolume。

> **警告：**
>
> 在本地群集中，默认的 StorageClass 使用hostPath供应器。 hostPath卷仅适用于开发和测试。使用 hostPath 卷，您的数据位于 Pod 调度

到的节点上的/tmp中，并且不会在节点之间移动。如果 Pod 死亡并被调度到群集中的另一个节点，或者该节点重新启动，则数据将丢失。

说明：

如果要建立需要使用hostPath设置程序的集群，则必须在 controller-manager 组件中设置--enable-hostpath-provisioner标志。

说明：

如果你已经有运行在 Google Kubernetes Engine 的集群，请参考 this guide。

创建 kustomization.yaml

创建 Secret 生成器

A Secret 是存储诸如密码或密钥之类的敏感数据的对象。从 1.14 开始，kubectl 支持使用 kustomization 文件管理 Kubernetes 对象。您可以通过kustomization.yaml中的生成器创建一个 Secret。

通过以下命令在kustomization.yaml中添加一个 Secret 生成器。您需要用您要使用的密码替换YOUR_PASSWORD。

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: mysql-pass
  literals:
  - password=YOUR_PASSWORD
EOF
```

补充 MySQL 和 WordPress 的资源配置

以下 manifest 文件描述了单实例 MySQL 部署。MySQL 容器将 PersistentVolume 挂载在/var/lib/mysql。 MYSQL_ROOT_PASSWORD环境变量设置来自 Secret 的数据库密码。

application/wordpress/mysql-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
```

```
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
```

```
            name: mysql-pass
            key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

以下 manifest 文件描述了单实例 WordPress 部署。WordPress 容器将网站数据文件位于/var/www/html的 PersistentVolume。WORDPRESS_DB_HOST环境变量集上面定义的 MySQL Service 的名称，WordPress 将通过 Service 访问数据库。WORDPRESS_DB_PASSWORD环境变量设置从 Secret kustomize 生成的数据库密码。

application/wordpress/wordpress-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
```

```
    requests:
      storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:4.8-apache
        name: wordpress
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 80
          name: wordpress
        volumeMounts:
        - name: wordpress-persistent-storage
          mountPath: /var/www/html
      volumes:
      - name: wordpress-persistent-storage
        persistentVolumeClaim:
          claimName: wp-pv-claim
```

1. 下载 MySQL deployment 配置文件。

```
curl -LO https://k8s.io/examples/application/wordpress/mysql-deployment.yaml
```

2. 下载 WordPress 配置文件。

```
curl -LO https://k8s.io/examples/application/wordpress/wordpress-deployment.yaml
```

3. 补充到 kustomization.yaml 文件。

```
cat <<EOF >>./kustomization.yaml
resources:
  - mysql-deployment.yaml
  - wordpress-deployment.yaml
EOF
```

应用和验证

kustomization.yaml包含用于部署 WordPress 网站的所有资源以及 MySQL 数据库。您可以通过以下方式应用目录

```
kubectl apply -k ./
```

现在，您可以验证所有对象是否存在。

1. 通过运行以下命令验证 Secret 是否存在：

   ```
   kubectl get secrets
   ```

   响应应如下所示：

   ```
   NAME                    TYPE                                  DATA   AGE
   mysql-pass-c57bb4t7mf   Opaque                                1      9s
   ```

2. 验证是否已动态配置 PersistentVolume：

   ```
   kubectl get pvc
   ```

   > **说明：** 设置和绑定 PV 可能要花费几分钟。

   响应应如下所示：

   ```
   NAME             STATUS    VOLUME                                     CAPACITY
   ACCESS MODES   STORAGECLASS      AGE
   mysql-pv-claim   Bound     pvc-8cbd7b2e-4044-11e9-
   b2bb-42010a800002   20Gi       RWO            standard       77s
   wp-pv-claim      Bound     pvc-8cd0df54-4044-11e9-
   b2bb-42010a800002   20Gi       RWO            standard       77s
   ```

3. 通过运行以下命令来验证 Pod 是否正在运行：

```
kubectl get pods
```

说明： 等待 Pod 状态变成RUNNING可能会花费几分钟。

响应应如下所示：

```
NAME                               READY     STATUS    RESTARTS   AGE
wordpress-mysql-1894417608-x5dzt   1/1       Running   0          40s
```

4. 通过运行以下命令来验证 Service 是否正在运行：

```
kubectl get services wordpress
```

响应应如下所示：

```
NAME        TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
wordpress   ClusterIP   10.0.0.89    <pending>     80:32406/TCP   4m
```

说明： Minikube 只能通过 NodePort 公开服务。EXTERNAL-IP 始终处于挂起状态

5. 运行以下命令以获取 WordPress 服务的 IP 地址：

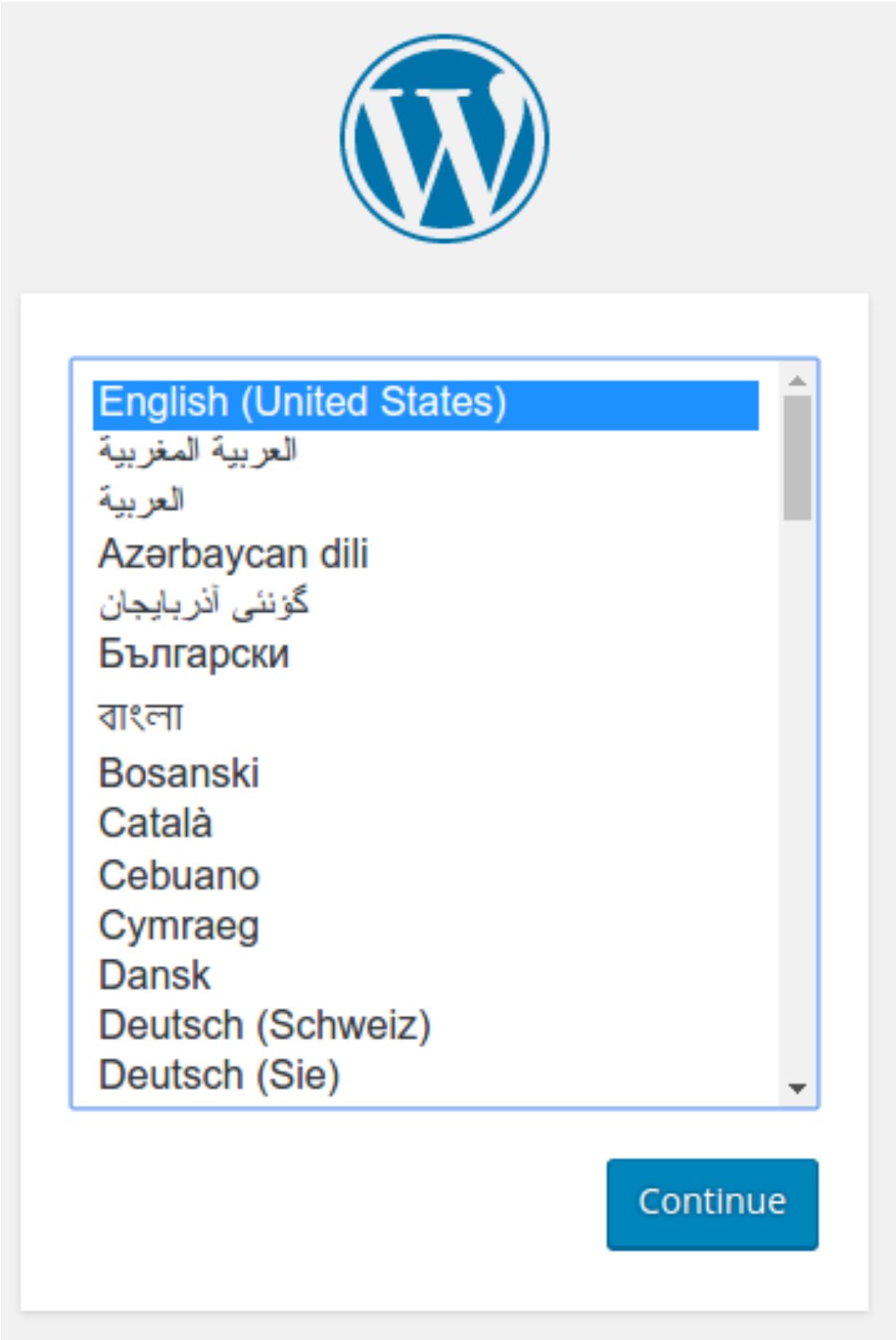
```
minikube service wordpress --url
```

响应应如下所示：

```
http://1.2.3.4:32406
```

6. 复制 IP 地址，然后将页面加载到浏览器中来查看您的站点。

   您应该看到类似于以下屏幕截图的 WordPress 设置页面。



警告：

不要在此页面上保留 WordPress 安装。如果其他用户找到了它，他们可以在您的实例上建立一个网站并使用它来提供恶意内容。

通过创建用户名和密码来安装 WordPress 或删除您的实例。

清理现场

1. 运行一下命令删除您的 Secret，Deployments，Services and PersistentVolumeClaims：

   ```
   kubectl delete -k ./
   ```

接下来

- 了解更多关于 Introspection and Debugging
- 了解更多关于 Jobs
- 了解更多关于 Port Forwarding
- 了解如何 Get a Shell to a Container

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 27, 2020 at 1:55 PM PST: [zh] Fix links on docs home (073ce649b)

StatefulSet 基础

本教程介绍如何了使用 StatefulSets 来管理应用。演示了如何创建、删除、扩容/缩容和更新 StatefulSets 的 Pods。

准备开始

在开始本教程之前，你应该熟悉以下 Kubernetes 的概念：

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- PersistentVolume Provisioning
- StatefulSets
- kubectl CLI

本教程假设你的集群被配置为动态的提供 PersistentVolumes。如果没有这样配置，在开始本教程之前，你需要手动准备 2 个 1 GiB 的存储卷。

教程目标

StatefulSets 旨在与有状态的应用及分布式系统一起使用。然而在 Kubernetes 上管理有状态应用和分布式系统是一个宽泛而复杂的话题。为了演示 StatefulSet 的基本特性，并且不使前后的主题混淆，你将会使用 StatefulSet 部署一个简单的 web 应用。

在阅读本教程后，你将熟悉以下内容：

- 如何创建 StatefulSet
- StatefulSet 怎样管理它的 Pods
- 如何删除 StatefulSet
- 如何对 StatefulSet 进行扩容/缩容
- 如何更新一个 StatefulSet 的 Pods

创建 StatefulSet

作为开始，使用如下示例创建一个 StatefulSet。它和 StatefulSets 概念中的示例相似。它创建了一个 Headless Service nginx 用来发布 StatefulSet web 中的 Pod 的 IP 地址。

application/web/web.yaml

```yaml
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:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
```

```
    app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    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" ]
    resources:
      requests:
        storage: 1Gi
```

下载上面的例子并保存为文件 web.yaml。

你需要使用两个终端窗口。在第一个终端中，使用 kubectl get 来查看 StatefulSet 的 Pods 的创建情况。

```
kubectl get pods -w -l app=nginx
```

在另一个终端中，使用 kubectl apply来创建定义在 web.yaml 中的 Headless Service 和 StatefulSet。

```
kubectl apply -f web.yaml
service/nginx created
statefulset.apps/web created
```

上面的命令创建了两个 Pod，每个都运行了一个 NGINX web 服务器。获取 nginx Service 和 web StatefulSet 来验证是否成功的创建了它们。

```
kubectl get service nginx
NAME      TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     ClusterIP    None         <none>        80/TCP    12s

kubectl get statefulset web
```

```
NAME    DESIRED   CURRENT   AGE
web     2         1         20s
```

顺序创建 Pod

对于一个拥有 N 个副本的 StatefulSet，Pod 被部署时是按照 {0 …… N-1} 的序号顺序创建的。在第一个终端中使用 kubectl get 检查输出。这个输出最终将看起来像下面的样子。

```
kubectl get pods -w -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   0/1     Pending   0          0s
web-0   0/1     Pending   0          0s
web-0   0/1     ContainerCreating   0          0s
web-0   1/1     Running   0          19s
web-1   0/1     Pending   0          0s
web-1   0/1     Pending   0          0s
web-1   0/1     ContainerCreating   0          0s
web-1   1/1     Running   0          18s
```

请注意在 web-0 Pod 处于 Running和Ready 状态后 web-1 Pod 才会被启动。

StatefulSet 中的 Pod

StatefulSet 中的 Pod 拥有一个唯一的顺序索引和稳定的网络身份标识。

检查 Pod 的顺序索引

获取 StatefulSet 的 Pod。

```
kubectl get pods -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          1m
web-1   1/1     Running   0          1m
```

如同 StatefulSets 概念中所提到的，StatefulSet 中的 Pod 拥有一个具有黏性的、独一无二的身份标志。这个标志基于 StatefulSet 控制器分配给每个 Pod 的唯一顺序索引。Pod 的名称的形式为<statefulset name>-<ordinal index>。webStatefulSet 拥有两个副本，所以它创建了两个 Pod：web-0和web-1。

使用稳定的网络身份标识

每个 Pod 都拥有一个基于其顺序索引的稳定的主机名。使用kubectl exec在每个 Pod 中执行hostname。

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1
```

使用 kubectl run 运行一个提供 nslookup 命令的容器，该命令来自于 dnsutils 包。通过对 Pod 的主机名执行 nslookup，你可以检查他们在集群内部的 DNS 地址。

```
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.6

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.6
```

headless service 的 CNAME 指向 SRV 记录（记录每个 Running 和 Ready 状态的 Pod）。SRV 记录指向一个包含 Pod IP 地址的记录表项。

在一个终端中查看 StatefulSet 的 Pod。

```
kubectl get pod -w -l app=nginx
```

在另一个终端中使用 kubectl delete 删除 StatefulSet 中所有的 Pod。

```
kubectl delete pod -l app=nginx
pod "web-0" deleted
pod "web-1" deleted
```

等待 StatefulSet 重启它们，并且两个 Pod 都变成 Running 和 Ready 状态。

```
kubectl get pod -w -l app=nginx
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0          0s
web-1     1/1       Running   0          34s
```

使用 kubectl exec 和 kubectl run 查看 Pod 的主机名和集群内部的 DNS 表项。

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
web-0
web-1

kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm /
bin/sh
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.7

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.8
```

Pod 的序号、主机名、SRV 条目和记录名称没有改变，但和 Pod 相关联的 IP 地址可能发生了改变。在本教程中使用的集群中它们就改变了。这就是为什么不要在其他应用中使用 StatefulSet 中的 Pod 的 IP 地址进行连接，这点很重要。

如果你需要查找并连接一个 StatefulSet 的活动成员，你应该查询 Headless Service 的 CNAME。和 CNAME 相关联的 SRV 记录只会包含 StatefulSet 中处于 Running 和 Ready 状态的 Pod。

如果你的应用已经实现了用于测试 liveness 和 readiness 的连接逻辑，你可以使用 Pod 的 SRV 记录（web-0.nginx.default.svc.cluster.local， web-1.nginx.default.svc.cluster.local）。因为他们是稳定的，并且当你的 Pod 的状态变为 Running 和 Ready 时，你的应用就能够发现它们的地址。

写入稳定的存储

获取 web-0 和 web-1 的 PersistentVolumeClaims。

```
kubectl get pvc -l app=nginx
NAME        STATUS    VOLUME                                     CAPACITY
ACCESSMODES   AGE
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002
1Gi         RWO           48s
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002
1Gi         RWO           48s
```

StatefulSet 控制器创建了两个 PersistentVolumeClaims，绑定到两个 PersistentVolumes。由于本教程使用的集群配置为动态提供 PersistentVolume，所有的 PersistentVolume 都是自动创建和绑定的。

NGINX web 服务器默认会加载位于 /usr/share/nginx/html/index.html 的 index 文件。StatefulSets spec 中的 volumeMounts 字段保证了 /usr/share/nginx/html 文件夹由一个 PersistentVolume 支持。

将 Pod 的主机名写入它们的index.html文件并验证 NGINX web 服务器使用该主机名提供服务。

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'echo $(hostname) > /usr/share/nginx/html/index.html'; done

for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

> **说明：**
>
> 请注意，如果你看见上面的 curl 命令返回了 403 Forbidden 的响应，你需要像这样修复使用 volumeMounts（due to a bug when using hostPath volumes）挂载的目录的权限：
>
> ```
> for i in 0 1; do kubectl exec web-$i -- chmod 755 /usr/share/nginx/html; done
> ```
>
> 在你重新尝试上面的 curl 命令之前。

在一个终端查看 StatefulSet 的 Pod。

```
kubectl get pod -w -l app=nginx
```

在另一个终端删除 StatefulSet 所有的 Pod。

```
kubectl delete pod -l app=nginx
pod "web-0" deleted
pod "web-1" deleted
```

在第一个终端里检查 kubectl get 命令的输出，等待所有 Pod 变成 Running 和 Ready 状态。

```
kubectl get pod -w -l app=nginx
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
```

```
web-1   0/1   ContainerCreating   0   0s
web-1   1/1   Running   0   34s
```

验证所有 web 服务器在继续使用它们的主机名提供服务。

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

虽然 web-0 和 web-1 被重新调度了，但它们仍然继续监听各自的主机名，因为和它们的 PersistentVolumeClaim 相关联的 PersistentVolume 被重新挂载到了各自的 volumeMount 上。不管 web-0 和 web-1 被调度到了哪个节点上，它们的 PersistentVolumes 将会被挂载到合适的挂载点上。

扩容/缩容 StatefulSet

扩容/缩容 StatefulSet 指增加或减少它的副本数。这通过更新 replicas 字段完成。你可以使用kubectl scale 或者kubectl patch来扩容/缩容一个 StatefulSet。

扩容

在一个终端窗口观察 StatefulSet 的 Pod。

```
kubectl get pods -w -l app=nginx
```

在另一个终端窗口使用 kubectl scale 扩展副本数为 5。

```
kubectl scale sts web --replicas=5
statefulset.apps/web scaled
```

在第一个 终端中检查 kubectl get 命令的输出，等待增加的 3 个 Pod 的状态变为 Running 和 Ready。

```
kubectl get pods -w -l app=nginx
NAME    READY   STATUS    RESTARTS   AGE
web-0   1/1     Running   0          2h
web-1   1/1     Running   0          2h
NAME    READY   STATUS    RESTARTS   AGE
web-2   0/1     Pending   0          0s
web-2   0/1     Pending   0          0s
web-2   0/1     ContainerCreating   0   0s
web-2   1/1     Running   0          19s
web-3   0/1     Pending   0          0s
web-3   0/1     Pending   0          0s
web-3   0/1     ContainerCreating   0   0s
web-3   1/1     Running   0          18s
web-4   0/1     Pending   0          0s
web-4   0/1     Pending   0          0s
```

```
web-4     0/1       ContainerCreating   0          0s
web-4     1/1       Running   0          19s
```

StatefulSet 控制器扩展了副本的数量。如同[创建 StatefulSet] 所述，StatefulSet 按序号索引顺序的创建每个 Pod，并且会等待前一个 Pod 变为 Running 和 Ready 才会启动下一个 Pod。

缩容

在一个终端观察 StatefulSet 的 Pod。

```
kubectl get pods -w -l app=nginx
```

在另一个终端使用 kubectl patch 将 StatefulSet 缩容回三个副本。

```
kubectl patch sts web -p '{"spec":{"replicas":3}}'
statefulset.apps/web patched
```

等待 web-4 和 web-3 状态变为 Terminating。

```
kubectl get pods -w -l app=nginx
NAME      READY     STATUS            RESTARTS   AGE
web-0     1/1       Running           0          3h
web-1     1/1       Running           0          3h
web-2     1/1       Running           0          55s
web-3     1/1       Running           0          36s
web-4     0/1       ContainerCreating   0        18s
NAME      READY     STATUS    RESTARTS   AGE
web-4     1/1       Running   0          19s
web-4     1/1       Terminating   0        24s
web-4     1/1       Terminating   0        24s
web-3     1/1       Terminating   0        42s
web-3     1/1       Terminating   0        42s
```

顺序终止 Pod

控制器会按照与 Pod 序号索引相反的顺序每次删除一个 Pod。在删除下一个 Pod 前会等待上一个被完全关闭。

获取 StatefulSet 的 PersistentVolumeClaims。

```
kubectl get pvc -l app=nginx
NAME        STATUS    VOLUME                                     CAPACITY
ACCESSMODES   AGE
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002
1Gi         RWO           13h
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002
```

```
1Gi        RWO         13h
www-web-2   Bound     pvc-e1125b27-b508-11e6-932f-42010a800002
1Gi        RWO         13h
www-web-3   Bound     pvc-e1176df6-b508-11e6-932f-42010a800002
1Gi        RWO         13h
www-web-4   Bound     pvc-e11bb5f8-b508-11e6-932f-42010a800002
1Gi        RWO         13h
```

五个 PersistentVolumeClaims 和五个 PersistentVolumes 仍然存在。查看 Pod 的 稳定存储，我们发现当删除 StatefulSet 的 Pod 时，挂载到 StatefulSet 的 Pod 的 PersistentVolumes 不会被删除。当这种删除行为是由 StatefulSet 缩容引起时也是一样的。

更新 StatefulSet

Kubernetes 1.7 版本的 StatefulSet 控制器支持自动更新。更新策略由 StatefulSet API Object 的spec.updateStrategy 字段决定。这个特性能够用来更新一个 StatefulSet 中的 Pod 的 container images，resource requests，以及 limits，labels 和 annotations。RollingUpdate滚动更新是 StatefulSets 默认策略。

Rolling Update 策略

RollingUpdate 更新策略会更新一个 StatefulSet 中所有的 Pod，采用与序号索引相反的顺序并遵循 StatefulSet 的保证。

Patch web StatefulSet 来执行 RollingUpdate 更新策略。

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":
{"type":"RollingUpdate"}}}'
statefulset.apps/web patched
```

在一个终端窗口中 patch web StatefulSet 来再次的改变容器镜像。

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/
spec/template/spec/containers/0/image", "value":"gcr.io/google_containers/
nginx-slim:0.8"}]'
statefulset.apps/web patched
```

在另一个终端监控 StatefulSet 中的 Pod。

```
kubectl get po -l app=nginx -w
NAME     READY     STATUS    RESTARTS   AGE
web-0    1/1       Running   0          7m
web-1    1/1       Running   0          7m
web-2    1/1       Running   0          8m
```

```
web-2   1/1   Terminating   0   8m
web-2   1/1   Terminating   0   8m
web-2   0/1   Terminating   0   8m
web-2   0/1   Terminating   0   8m
web-2   0/1   Terminating   0   8m
web-2   0/1   Terminating   0   8m
web-2   0/1   Pending   0   0s
web-2   0/1   Pending   0   0s
web-2   0/1   ContainerCreating   0   0s
web-2   1/1   Running   0   19s
web-1   1/1   Terminating   0   8m
web-1   0/1   Terminating   0   8m
web-1   0/1   Terminating   0   8m
web-1   0/1   Terminating   0   8m
web-1   0/1   Pending   0   0s
web-1   0/1   Pending   0   0s
web-1   0/1   ContainerCreating   0   0s
web-1   1/1   Running   0   6s
web-0   1/1   Terminating   0   7m
web-0   1/1   Terminating   0   7m
web-0   0/1   Terminating   0   7m
web-0   0/1   Terminating   0   7m
web-0   0/1   Terminating   0   7m
web-0   0/1   Terminating   0   7m
web-0   0/1   Pending   0   0s
web-0   0/1   Pending   0   0s
web-0   0/1   ContainerCreating   0   0s
web-0   1/1   Running   0   10s
```

StatefulSet 里的 Pod 采用和序号相反的顺序更新。在更新下一个 Pod 前，StatefulSet 控制器终止每个 Pod 并等待它们变成 Running 和 Ready。请注意，虽然在顺序后继者变成 Running 和 Ready 之前 StatefulSet 控制器不会更新下一个 Pod，但它仍然会重建任何在更新过程中发生故障的 Pod，使用的是它们当前的版本。已经接收到更新请求的 Pod 将会被恢复为更新的版本，没有收到请求的 Pod 则会被恢复为之前的版本。像这样，控制器尝试继续使应用保持健康并在出现间歇性故障时保持更新的一致性。

获取 Pod 来查看他们的容器镜像。

```
for p in 0 1 2; do kubectl get po web-$p --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
k8s.gcr.io/nginx-slim:0.8
k8s.gcr.io/nginx-slim:0.8
k8s.gcr.io/nginx-slim:0.8
```

StatefulSet 中的所有 Pod 现在都在运行之前的容器镜像。

小窍门：你还可以使用 kubectl rollout status sts/<name> 来查看 rolling update 的状态。

分段更新

你可以使用 RollingUpdate 更新策略的 partition 参数来分段更新一个 StatefulSet。分段的更新将会使 StatefulSet 中的其余所有 Pod 保持当前版本的同时仅允许改变 StatefulSet 的 .spec.template。

Patch web StatefulSet 来对 updateStrategy 字段添加一个分区。

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":3}}}}'
statefulset.apps/web patched
```

再次 Patch StatefulSet 来改变容器镜像。

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/
spec/template/spec/containers/0/image", "value":"k8s.gcr.io/nginx-slim:
0.7"}]'
statefulset.apps/web patched
```

删除 StatefulSet 中的 Pod。

```
kubectl delete po web-2
pod "web-2" deleted
```

等待 Pod 变成 Running 和 Ready。

```
kubectl get po -lapp=nginx -w
NAME     READY     STATUS            RESTARTS   AGE
web-0    1/1       Running           0          4m
web-1    1/1       Running           0          4m
web-2    0/1       ContainerCreating   0          11s
web-2    1/1       Running   0         18s
```

获取 Pod 的容器。

```
kubectl get po web-2 --template '{{range $i, $c := .spec.containers}}
{{$c.image}}{{end}}'
k8s.gcr.io/nginx-slim:0.8
```

请注意，虽然更新策略是 RollingUpdate，StatefulSet 控制器还是会使用原始的容器恢复 Pod。这是因为 Pod 的序号比 updateStrategy 指定的 partition 更小。

灰度发布

你可以通过减少 上文指定的 partition 来进行灰度发布，以此来测试你的程序的改动。

通过 patch 命令修改 StatefulSet 来减少分区。

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":2}}}}'
statefulset.apps/web patched
```

等待 web-2 变成 Running 和 Ready。

```
kubectl get po -lapp=nginx -w
NAME     READY     STATUS            RESTARTS   AGE
web-0    1/1       Running           0          4m
web-1    1/1       Running           0          4m
web-2    0/1       ContainerCreating 0          11s
web-2    1/1       Running   0       18s
```

获取 Pod 的容器。

```
kubectl get po web-2 --template '{{range $i, $c := .spec.containers}}
{{$c.image}}{{end}}'
k8s.gcr.io/nginx-slim:0.7
```

当你改变 partition 时，StatefulSet 会自动的更新 web-2 Pod，这是因为 Pod 的序号小于或等于 partition。

删除 web-1 Pod。

```
kubectl delete po web-1
pod "web-1" deleted
```

等待 web-1 变成 Running 和 Ready。

```
kubectl get po -lapp=nginx -w
NAME     READY     STATUS       RESTARTS   AGE
web-0    1/1       Running      0          6m
web-1    0/1       Terminating  0          6m
web-2    1/1       Running      0          2m
web-1    0/1       Terminating  0          6m
web-1    0/1       Terminating  0          6m
web-1    0/1       Terminating  0          6m
web-1    0/1       Pending   0       0s
web-1    0/1       Pending   0       0s
web-1    0/1       ContainerCreating  0       0s
web-1    1/1       Running   0       18s
```

获取 web-1 Pod 的容器。

```
kubectl get po web-1 --template '{{range $i, $c := .spec.containers}}
{{$c.image}}{{end}}'
k8s.gcr.io/nginx-slim:0.8
```

web-1 被按照原来的配置恢复，因为 Pod 的序号小于分区。当指定了分区时，如果更新了 StatefulSet 的 .spec.template，则所有序号大于或等于分区的 Pod 都将被更新。如果一个序号小于分区的 Pod 被删除或者终止，它将被按照原来的配置恢复。

分阶段的发布

你可以使用类似灰度发布的方法执行一次分阶段的发布（例如一次线性的、等比的或者指数形式的发布）。要执行一次分阶段的发布，你需要设置 partition 为希望控制器暂停更新的序号。

分区当前为2。请将分区设置为0。

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":0}}}}'
statefulset.apps/web patched
```

等待 StatefulSet 中的所有 Pod 变成 Running 和 Ready。

```
kubectl get po -lapp=nginx -w
NAME     READY     STATUS              RESTARTS   AGE
web-0    1/1       Running             0          3m
web-1    0/1       ContainerCreating   0          11s
web-2    1/1       Running             0          2m
web-1    1/1       Running   0         18s
web-0    1/1       Terminating   0         3m
web-0    1/1       Terminating   0         3m
web-0    0/1       Terminating   0         3m
web-0    0/1       Terminating   0         3m
web-0    0/1       Terminating   0         3m
web-0    0/1       Terminating   0         3m
web-0    0/1       Pending   0         0s
web-0    0/1       Pending   0         0s
web-0    0/1       ContainerCreating   0         0s
web-0    1/1       Running   0         3s
```

获取 Pod 的容器。

```
for p in 0 1 2; do kubectl get po web-$p --template '{{range $i,
$c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
k8s.gcr.io/nginx-slim:0.7
k8s.gcr.io/nginx-slim:0.7
```

```
k8s.gcr.io/nginx-slim:0.7
```

将 partition 改变为 0 以允许 StatefulSet 控制器继续更新过程。

On Delete 策略

OnDelete 更新策略实现了传统（1.7 之前）行为，它也是默认的更新策略。当你选择这个更新策略并修改 StatefulSet 的 .spec.template 字段时，StatefulSet 控制器将不会自动的更新 Pod。

删除 StatefulSet

StatefulSet 同时支持级联和非级联删除。使用非级联方式删除 StatefulSet 时，StatefulSet 的 Pod 不会被删除。使用级联删除时，StatefulSet 和它的 Pod 都会被删除。

非级联删除

在一个终端窗口查看 StatefulSet 中的 Pod。

```
kubectl get pods -w -l app=nginx
```

使用 kubectl delete 删除 StatefulSet。请确保提供了 --cascade=false 参数给命令。这个参数告诉 Kubernetes 只删除 StatefulSet 而不要删除它的任何 Pod。

```
kubectl delete statefulset web --cascade=false
statefulset.apps "web" deleted
```

获取 Pod 来检查他们的状态。

```
kubectl get pods -l app=nginx
NAME     READY    STATUS    RESTARTS   AGE
web-0    1/1      Running   0          6m
web-1    1/1      Running   0          7m
web-2    1/1      Running   0          5m
```

虽然 web 已经被删除了，但所有 Pod 仍然处于 Running 和 Ready 状态。 删除 web-0。

```
kubectl delete pod web-0
pod "web-0" deleted
```

获取 StatefulSet 的 Pod。

```
kubectl get pods -l app=nginx
NAME     READY    STATUS    RESTARTS   AGE
web-1    1/1      Running   0          10m
web-2    1/1      Running   0          7m
```

由于 web StatefulSet 已经被删除，web-0没有被重新启动。

在一个终端监控 StatefulSet 的 Pod。

```
kubectl get pods -w -l app=nginx
```

在另一个终端里重新创建 StatefulSet。请注意，除非你删除了 nginx Service（你不应该这样做），你将会看到一个错误，提示 Service 已经存在。

```
kubectl apply -f web.yaml
statefulset.apps/web created
service/nginx unchanged
```

请忽略这个错误。它仅表示 kubernetes 进行了一次创建 nginx Headless Service 的尝试，尽管那个 Service 已经存在。

在第一个终端中运行并检查 kubectl get 命令的输出。

```
kubectl get pods -w -l app=nginx
NAME     READY    STATUS    RESTARTS   AGE
web-1    1/1      Running   0          16m
web-2    1/1      Running   0          2m
NAME     READY    STATUS    RESTARTS   AGE
web-0    0/1      Pending   0          0s
web-0    0/1      Pending   0         0s
web-0    0/1      ContainerCreating   0       0s
web-0    1/1      Running   0         18s
web-2    1/1      Terminating   0       3m
web-2    0/1      Terminating   0       3m
web-2    0/1      Terminating   0       3m
web-2    0/1      Terminating   0       3m
```

当重新创建 web StatefulSet 时，web-0被第一个重新启动。由于 web-1 已经处于 Running 和 Ready 状态，当 web-0 变成 Running 和 Ready 时，StatefulSet 会直接接收这个 Pod。由于你重新创建的 StatefulSet 的 replicas 等于 2，一旦 web-0 被重新创建并且 web-1 被认为已经处于 Running 和 Ready 状态时，web-2将会被终止。

让我们再看看被 Pod 的 web 服务器加载的 index.html 的内容。

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

尽管你同时删除了 StatefulSet 和 web-0 Pod，但它仍然使用最初写入 index.html 文件的主机名进行服务。这是因为 StatefulSet 永远不会删除和一个 Pod 相关联的 PersistentVolumes。当你重建这个 StatefulSet 并且重新启动了 web-0 时，它原本的 PersistentVolume 会被重新挂载。

级联删除

在一个终端窗口观察 StatefulSet 里的 Pod。

```
kubectl get pods -w -l app=nginx
```

在另一个窗口中再次删除这个 StatefulSet。这次省略 --cascade=false 参数。

```
kubectl delete statefulset web
statefulset.apps "web" deleted
```

在第一个终端检查 kubectl get 命令的输出，并等待所有的 Pod 变成 Terminating 状态。

```
kubectl get pods -w -l app=nginx
NAME     READY    STATUS    RESTARTS   AGE
web-0    1/1      Running   0          11m
web-1    1/1      Running   0          27m
NAME     READY    STATUS       RESTARTS   AGE
web-0    1/1      Terminating  0          12m
web-1    1/1      Terminating  0          29m
web-0    0/1      Terminating  0          12m
web-0    0/1      Terminating  0          12m
web-0    0/1      Terminating  0          12m
web-1    0/1      Terminating  0          29m
web-1    0/1      Terminating  0          29m
web-1    0/1      Terminating  0          29m
```

如同你在缩容一节看到的，Pod 按照和他们序号索引相反的顺序每次终止一个。在终止一个 Pod 前，StatefulSet 控制器会等待 Pod 后继者被完全终止。

请注意，虽然级联删除会删除 StatefulSet 和它的 Pod，但它并不会删除和 StatefulSet 关联的 Headless Service。你必须手动删除nginx Service。

```
kubectl delete service nginx
service "nginx" deleted
```

再一次重新创建 StatefulSet 和 Headless Service。

```
kubectl apply -f web.yaml
service/nginx created
statefulset.apps/web created
```

当 StatefulSet 所有的 Pod 变成 Running 和 Ready 时，获取它们的 index.html 文件的内容。

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

即使你已经删除了 StatefulSet 和它的全部 Pod，这些 Pod 将会被重新创建并挂载它们的 PersistentVolumes，并且 web-0 和 web-1 将仍然使用它们的主机名提供服务。

最后删除 nginx service...

```
kubectl delete service nginx
```

```
service "nginx" deleted
```

... 并且删除 web StatefulSet:

```
kubectl delete statefulset web
```

```
statefulset "web" deleted
```

Pod 管理策略

对于某些分布式系统来说，StatefulSet 的顺序性保证是不必要和/或者不应该的。这些系统仅仅要求唯一性和身份标志。为了解决这个问题，在 Kubernetes 1.7 中我们针对 StatefulSet API Object 引入了 .spec.podManagementPolicy。

OrderedReady Pod 管理策略

OrderedReady pod 管理策略是 StatefulSets 的默认选项。它告诉 StatefulSet 控制器遵循上文展示的顺序性保证。

Parallel Pod 管理策略

Parallel pod 管理策略告诉 StatefulSet 控制器并行的终止所有 Pod，在启动或终止另一个 Pod 前，不必等待这些 Pod 变成 Running 和 Ready 或者完全终止状态。

application/web/web-parallel.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
```

```yaml
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  podManagementPolicy: "Parallel"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      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" ]
      resources:
        requests:
          storage: 1Gi
```

下载上面的例子并保存为 web-parallel.yaml。

这份清单和你在上文下载的完全一样，只是 web StatefulSet 的 .spec.podManagementPolicy 设置成了 Parallel。

在一个终端窗口查看 StatefulSet 中的 Pod。

```
kubectl get po -lapp=nginx -w
```

在另一个终端窗口创建清单中的 StatefulSet 和 Service。

```
kubectl apply -f web-parallel.yaml
service/nginx created
statefulset.apps/web created
```

查看你在第一个终端中运行的 kubectl get 命令的输出。

```
kubectl get po -lapp=nginx -w
NAME    READY   STATUS    RESTARTS   AGE
web-0   0/1     Pending   0          0s
web-0   0/1     Pending   0          0s
web-1   0/1     Pending   0          0s
web-1   0/1     Pending   0          0s
web-0   0/1     ContainerCreating   0          0s
web-1   0/1     ContainerCreating   0          0s
web-0   1/1     Running   0          10s
web-1   1/1     Running   0          10s
```

StatefulSet 控制器同时启动了 web-0 和 web-1。

保持第二个终端打开，并在另一个终端窗口中扩容 StatefulSet。

```
kubectl scale statefulset/web --replicas=4
statefulset.apps/web scaled
```

在 kubectl get 命令运行的终端里检查它的输出。

```
web-3   0/1     Pending   0          0s
web-3   0/1     Pending   0          0s
web-3   0/1     Pending   0          7s
web-3   0/1     ContainerCreating   0          7s
web-2   1/1     Running   0          10s
web-3   1/1     Running   0          26s
```

StatefulSet 控制器启动了两个新的 Pod，而且在启动第二个之前并没有等待第一个变成 Running 和 Ready 状态。

保持这个终端打开，并在另一个终端删除 web StatefulSet。

```
kubectl delete sts web
```

在另一个终端里再次检查 kubectl get 命令的输出。

```
web-3   1/1     Terminating   0          9m
web-2   1/1     Terminating   0          9m
```

```
web-3   1/1   Terminating   0   9m
web-2   1/1   Terminating   0   9m
web-1   1/1   Terminating   0   44m
web-0   1/1   Terminating   0   44m
web-0   0/1   Terminating   0   44m
web-3   0/1   Terminating   0   9m
web-2   0/1   Terminating   0   9m
web-1   0/1   Terminating   0   44m
web-0   0/1   Terminating   0   44m
web-2   0/1   Terminating   0   9m
web-2   0/1   Terminating   0   9m
web-2   0/1   Terminating   0   9m
web-1   0/1   Terminating   0   44m
web-1   0/1   Terminating   0   44m
web-1   0/1   Terminating   0   44m
web-0   0/1   Terminating   0   44m
web-0   0/1   Terminating   0   44m
web-0   0/1   Terminating   0   44m
web-3   0/1   Terminating   0   9m
web-3   0/1   Terminating   0   9m
web-3   0/1   Terminating   0   9m
```

StatefulSet 控制器将并发的删除所有 Pod，在删除一个 Pod 前不会等待它的顺序后继者终止。

关闭 kubectl get 命令运行的终端并删除nginx Service。

```
kubectl delete svc nginx
```

清理现场

你需要删除本教程中用到的 PersistentVolumes 的持久化存储介质。基于你的环境、存储配置和提供方式，按照必须的步骤保证回收所有的存储。

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 1:50 PM PST: [zh] Incorrect url (495d6859d)

示例：使用 Stateful Sets 部署 Cassandra

目录

下文描述了在 Kubernetes 上部署一个_云原生_ Cassandra 的过程。当我们说_云原生_时，指的是一个应用能够理解它运行在一个集群管理器内部，并且使用这个集群的管理基础设施来帮助实现这个应用。特别的，本例使用了一个自定义的 Cassandra SeedProvider 帮助 Cassandra 发现新加入集群 Cassandra 节点。

本示例也使用了Kubernetes的一些核心组件：

- *Pods*
- *Services*
- *Replication Controllers*
- *Stateful Sets*
- *Daemon Sets*

准备工作

本示例假设你已经安装运行了一个 Kubernetes集群（版本 >=1.2），并且还在某个路径下安装了 kubectl 命令行工具。请查看 getting started guides 获取关于你的平台的安装说明。

本示例还需要一些代码和配置文件。为了避免手动输入，你可以 git clone Kubernetes 源到你本地。

Cassandra Docker 镜像

Pod 使用来自 Google 容器仓库 的 gcr.io/google-samples/cassandra:v12 镜像。这个 docker 镜像基于 debian:jessie 并包含 OpenJDK 8。该镜像包含一个

从 Apache Debian 源中安装的标准 Cassandra。你可以通过使用环境变量改变插入到 cassandra.yaml 文件中的参数值。

ENV VAR	DEFAULT VALUE
CASSANDRA_CLUSTER_NAME	'Test Cluster'
CASSANDRA_NUM_TOKENS	32
CASSANDRA_RPC_ADDRESS	0.0.0.0

快速入门

application/cassandra/cassandra-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

如果你希望直接跳到我们使用的命令，以下是全部步骤：

```
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-service.yaml
```

application/cassandra/cassandra-statefulset.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
```

```
    app: cassandra
template:
  metadata:
    labels:
      app: cassandra
  spec:
    terminationGracePeriodSeconds: 1800
    containers:
    - name: cassandra
      image: gcr.io/google-samples/cassandra:v13
      imagePullPolicy: Always
      ports:
      - containerPort: 7000
        name: intra-node
      - containerPort: 7001
        name: tls-intra-node
      - containerPort: 7199
        name: jmx
      - containerPort: 9042
        name: cql
      resources:
        limits:
          cpu: "500m"
          memory: 1Gi
        requests:
          cpu: "500m"
          memory: 1Gi
      securityContext:
        capabilities:
          add:
            - IPC_LOCK
      lifecycle:
        preStop:
          exec:
            command:
            - /bin/sh
            - -c
            - nodetool drain
      env:
        - name: MAX_HEAP_SIZE
          value: 512M
        - name: HEAP_NEWSIZE
          value: 100M
        - name: CASSANDRA_SEEDS
          value: "cassandra-0.cassandra.default.svc.cluster.local"
        - name: CASSANDRA_CLUSTER_NAME
```

```
          value: "K8Demo"
        - name: CASSANDRA_DC
          value: "DC1-K8Demo"
        - name: CASSANDRA_RACK
          value: "Rack1-K8Demo"
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
      readinessProbe:
        exec:
          command:
          - /bin/bash
          - -c
          - /ready-probe.sh
        initialDelaySeconds: 15
        timeoutSeconds: 5
      # These volume mounts are persistent. They are like inline claims,
      # but not exactly because the names need to match exactly one of
      # the stateful pod volumes.
      volumeMounts:
      - name: cassandra-data
        mountPath: /cassandra_data
  # These are converted to volume claims by the controller
  # and mounted at the paths mentioned above.
  # do not use these in production until ssd GCEPersistentDisk or other ssd
pd
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: fast
      resources:
        requests:
          storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: k8s.io/minikube-hostpath
parameters:
  type: pd-ssd
```

```
# 创建 statefulset
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-statefulset.yaml

# 验证 Cassandra 集群。替换一个 pod 的名称。
kubectl exec -ti cassandra-0 -- nodetool status

# 清理
grace=$(kubectl get po cassandra-0 -o=jsonpath='{.spec.terminationGracePeriodSeconds}') \
  && kubectl delete statefulset,po -l app=cassandra \
  && echo "Sleeping $grace" \
  && sleep $grace \
  && kubectl delete pvc -l app=cassandra

#
# 资源控制器示例
#

# 创建一个副本控制器来复制 cassandra 节点
kubectl create -f cassandra/cassandra-controller.yaml

# 验证 Cassandra 集群。替换一个 pod 的名称。
kubectl exec -ti cassandra-xxxxx -- nodetool status

# 扩大 Cassandra 集群
kubectl scale rc cassandra --replicas=4

# 删除副本控制器
kubectl delete rc cassandra

#
# 创建一个 DaemonSet，在每个 kubernetes 节点上放置一个 cassandra 节点
#

kubectl create -f cassandra/cassandra-daemonset.yaml --validate=false

# 资源清理
kubectl delete service -l app=cassandra
kubectl delete daemonset cassandra
```

步骤 1：创建 Cassandra Headless Service

Kubernetes *Service* 描述一组执行同样任务的 *Pod*。在 Kubernetes 中，一个应用的原子调度单位是一个 Pod：一个或多个_必须_调度到相同主机上的容器。

这个 Service 用于在 Kubernetes 集群内部进行 Cassandra 客户端和 Cassandra Pod 之间的 DNS 查找。

以下为这个 service 的描述：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
    - port: 9042
  selector:
    app: cassandra
```

Download cassandra-service.yaml and cassandra-statefulset.yaml

为 StatefulSet 创建 service

```
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-service.yaml
```

以下命令显示了 service 是否被成功创建。

```
$ kubectl get svc cassandra
```

命令的响应应该像这样：

```
NAME        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
cassandra   None         <none>        9042/TCP   45s
```

如果返回错误则表示 service 创建失败。

步骤 2：使用 StatefulSet 创建 Cassandra Ring 环

StatefulSets（以前叫做 PetSets）特性在 Kubernetes 1.5 中升级为一个 *Beta* 组件。在集群环境中部署类似于 Cassandra 的有状态分布式应用是一项具有挑战性的工作。我们实现了 StatefulSet，极大的简化了这个过程。本示例使用了 StatefulSet 的多个特性，但其本身超出了本文的范围。请参考 StatefulSet 文档。

以下是 StatefulSet 的清单文件，用于创建一个由三个 pod 组成的 Cassandra ring 环。

本示例使用了 GCE Storage Class，请根据你运行的云平台做适当的修改。

```yaml
apiVersion: "apps/v1beta1"
kind: StatefulSet
metadata:
  name: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v12
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
           cpu: "500m"
           memory: 1Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "PID=$(pidof java) && kill $PID &&
while ps -p $PID > /dev/null; do sleep 1; done"]
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
```

```
        - name: CASSANDRA_SEEDS
          value: "cassandra-0.cassandra.default.svc.cluster.local"
        - name: CASSANDRA_CLUSTER_NAME
          value: "K8Demo"
        - name: CASSANDRA_DC
          value: "DC1-K8Demo"
        - name: CASSANDRA_RACK
          value: "Rack1-K8Demo"
        - name: CASSANDRA_AUTO_BOOTSTRAP
          value: "false"
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
      readinessProbe:
        exec:
          command:
          - /bin/bash
          - -c
          - /ready-probe.sh
        initialDelaySeconds: 15
        timeoutSeconds: 5
      # These volume mounts are persistent. They are like inline claims,
      # but not exactly because the names need to match exactly one of
      # the stateful pod volumes.
      volumeMounts:
      - name: cassandra-data
        mountPath: /cassandra_data
  # These are converted to volume claims by the controller
  # and mounted at the paths mentioned above.
  # do not use these in production until ssd GCEPersistentDisk or other ssd
pd
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
      annotations:
        volume.beta.kubernetes.io/storage-class: fast
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
```

```
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

创建 Cassandra StatefulSet 如下：

```
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-statefulset.yaml
```

步骤 3：验证和修改 Cassandra StatefulSet

这个 StatefulSet 的部署展示了 StatefulSets 提供的两个新特性：

1. Pod 的名称已知
2. Pod 以递增顺序部署

首先，运行下面的 kubectl 命令，验证 StatefulSet 已经被成功部署。

```
$ kubectl get statefulset cassandra
```

这个命令的响应应该像这样：

```
NAME       DESIRED   CURRENT   AGE
cassandra   3         3         13s
```

接下来观察 Cassandra pod 以一个接一个的形式部署。StatefulSet 资源按照数字序号的模式部署 pod：1，2，3 等。如果在 pod 部署前执行下面的命令，你就能够看到这种顺序的创建过程。

```
$ kubectl get pods -l="app=cassandra"
NAME         READY     STATUS              RESTARTS   AGE
cassandra-0   1/1       Running           0          1m
cassandra-1   0/1       ContainerCreating   0          8s
```

上面的示例显示了三个 Cassandra StatefulSet pod 中的两个已经部署。一旦所有的 pod 都部署成功，相同的命令会显示一个完整的 StatefulSet。

```
$ kubectl get pods -l="app=cassandra"
NAME         READY     STATUS    RESTARTS   AGE
cassandra-0   1/1       Running   0          10m
cassandra-1   1/1       Running   0          9m
cassandra-2   1/1       Running   0          8m
```

运行 Cassandra 工具 nodetool 将显示 ring 环的状态。

```
$ kubectl exec cassandra-0 -- nodetool status
Datacenter: DC1-K8Demo
======================
```

```
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host
ID                               Rack
UN  10.4.2.4  65.26 KiB  32           63.7%
a9d27f81-6783-461d-8583-87de2589133e  Rack1-K8Demo
UN  10.4.0.4  102.04 KiB  32           66.7%             5559a58c-8b03-47ad-bc32-
c621708dc2e4  Rack1-K8Demo
UN  10.4.1.4  83.06 KiB  32           69.6%             9dce943c-581d-4c0e-9543-
f519969cc805  Rack1-K8Demo
```

你也可以运行 cqlsh 来显示集群的 keyspaces。

```
$ kubectl exec cassandra-0 -- cqlsh -e 'desc keyspaces'

system_traces  system_schema  system_auth  system  system_distributed
```

你需要使用 kubectl edit 来增加或减小 Cassandra StatefulSet 的大小。你可以在文档 中找到更多关于 edit 命令的信息。

使用以下命令编辑 StatefulSet。

```
$ kubectl edit statefulset cassandra
```

这会在你的命令行中创建一个编辑器。你需要修改的行是 replicas。这个例子没有包含终端窗口的所有内容，下面示例中的最后一行就是你希望改变的 replicas 行。

```
# 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: apps/v1beta1
kind: StatefulSet
metadata:
  creationTimestamp: 2016-08-13T18:40:58Z
  generation: 1
  labels:
    app: cassandra
  name: cassandra
  namespace: default
  resourceVersion: "323"
  uid: 7a219483-6185-11e6-a910-42010a8a0fc0
spec:
  replicas: 3
```

按下面的示例修改清单文件并保存。

```
spec:
  replicas: 4
```

这个 StatefulSet 现在将包含四个 pod。

```
$ kubectl get statefulset cassandra
```

这个command的响应应该像这样：

```
NAME       DESIRED   CURRENT   AGE
cassandra   4         4         36m
```

对于 Kubernetes 1.5 发布版，beta StatefulSet 资源没有像 Deployment, ReplicaSet, Replication Controller 或者 Job 一样，包含 kubectl scale 功能，

步骤 4：删除 Cassandra StatefulSet

删除或者缩容 StatefulSet 时不会删除与之关联的 volumes。这样做是为了优先保证安全。你的数据比其它会被自动清除的 StatefulSet 关联资源更宝贵。删除 Persistent Volume Claims 可能会导致关联的 volumes 被删除，这种行为依赖 storage class 和 reclaim policy。永远不要期望能在 claim 删除后访问一个 volume。

使用如下命令删除 StatefulSet。

```
$ grace=$(kubectl get po cassandra-0 -
o=jsonpath='{.spec.terminationGracePeriodSeconds}') \
  && kubectl delete statefulset -l app=cassandra \
  && echo "Sleeping $grace" \
  && sleep $grace \
  && kubectl delete pvc -l app=cassandra
```

步骤 5：使用 Replication Controller 创建 Cassandra 节点 pod

Kubernetes *Replication Controller* 负责复制一个完全相同的 pod 集合。像 Service 一样，它具有一个 selector query，用来识别它的集合成员。和 Service 不一样的是，它还具有一个期望的副本数，并且会通过创建或删除 Pod 来保证 Pod 的数量满足它期望的状态。

和我们刚才定义的 Service 一起，Replication Controller 能够让我们轻松的构建一个复制的、可扩展的 Cassandra 集群。

让我们创建一个具有两个初始副本的 replication controller。

```
apiVersion: v1
kind: ReplicationController
```

```yaml
metadata:
  name: cassandra
  # The labels will be applied automatically
  # from the labels in the pod template, if not set
  # labels:
    # app: cassandra
spec:
  replicas: 2
  # The selector will be applied automatically
  # from the labels in the pod template, if not set.
  # selector:
     # app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      containers:
        - command:
            - /run.sh
          resources:
            limits:
              cpu: 0.5
          env:
            - name: MAX_HEAP_SIZE
              value: 512M
            - name: HEAP_NEWSIZE
              value: 100M
            - name: CASSANDRA_SEED_PROVIDER
              value: "io.k8s.cassandra.KubernetesSeedProvider"
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
          image: gcr.io/google-samples/cassandra:v12
          name: cassandra
          ports:
            - containerPort: 7000
              name: intra-node
            - containerPort: 7001
              name: tls-intra-node
            - containerPort: 7199
```

```
        name: jmx
      - containerPort: 9042
        name: cql
      volumeMounts:
      - mountPath: /cassandra_data
        name: data
  volumes:
    - name: data
      emptyDir: {}
```

下载示例

在这个描述中需要注意几件事情。

selector 属性包含了控制器的 selector query。它能够被显式指定，或者在没有设置时，像此处一样从 pod 模板中的 labels 中自动应用。

Pod 模板的标签 app:cassandra 匹配步骤1中的 Service selector。这就是 Service 如何选择 replication controller 创建的 pod 的原理。

replicas 属性指明了期望的副本数量，在本例中最开始为 2。我们很快将要扩容更多数量。

创建 Replication Controller：

```
$ kubectl create -f cassandra/cassandra-controller.yaml
```

你可以列出新建的 controller：

```
$ kubectl get rc -o wide
NAME        DESIRED   CURRENT   AGE       CONTAINER(S)
IMAGE(S)                          SELECTOR
cassandra   2         2         11s       cassandra     gcr.io/google-samples/
cassandra:v12   app=cassandra
```

现在，如果你列出集群中的 pod，并且使用 app=cassandra 标签过滤，你应该能够看到两个 Cassandra pod。（wide 参数使你能够看到 pod 被调度到了哪个 Kubernetes 节点上）

```
$ kubectl get pods -l="app=cassandra" -o wide
NAME              READY     STATUS    RESTARTS   AGE       NODE
cassandra-21qyy   1/1       Running   0          1m        kubernetes-minion-
b286
cassandra-q6sz7   1/1       Running   0          1m        kubernetes-minion-9ye5
```

因为这些 pod 拥有 app=cassandra 标签，它们被映射给了我们在步骤 1 中创建的 service。

你可以使用下面的 service endpoint 查询命令来检查 Pod 是否对 Service 可用。

```
$ kubectl get endpoints cassandra -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2015-06-21T22:34:12Z
  labels:
    app: cassandra
  name: cassandra
  namespace: default
  resourceVersion: "944373"
  uid: a3d6c25f-1865-11e5-a34e-42010af01bcc
subsets:
- addresses:
  - ip: 10.244.3.15
    targetRef:
      kind: Pod
      name: cassandra
      namespace: default
      resourceVersion: "944372"
      uid: 9ef9895d-1865-11e5-a34e-42010af01bcc
  ports:
  - port: 9042
    protocol: TCP
```

为了显示 SeedProvider 逻辑是按设想在运行，你可以使用 nodetool 命令来检查 Cassandra 集群的状态。为此，请使用 kubectl exec 命令，这样你就能在一个 Cassandra pod 上运行 nodetool。同样的，请替换 cassandra-xxxxx 为任意一个 pods的真实名字。

```
$ kubectl exec -ti cassandra-xxxxx -- nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens  Owns (effective)  Host ID
Rack
UN  10.244.0.5  74.09 KB   256     100.0%            86feda0f-f070-4a5b-
bda1-2eeb0ad08b77  rack1
```

```
UN  10.244.3.3  51.28 KB   256     100.0%           dafe3154-1d67-42e1-
ac1d-78e7e80dce2b  rack1
```

步骤 6：Cassandra 集群扩容

现在，让我们把 Cassandra 集群扩展到 4 个 pod。我们通过告诉 Replication Controller 现在我们需要 4 个副本来完成。

```
$ kubectl scale rc cassandra --replicas=4
```

你可以看到列出了新的 pod：

```
$ kubectl get pods -l="app=cassandra" -o wide
NAME              READY     STATUS    RESTARTS   AGE       NODE
cassandra-21qyy   1/1       Running   0          6m        kubernetes-minion-
b286
cassandra-81m2l   1/1       Running   0          47s       kubernetes-minion-
b286
cassandra-8qoyp   1/1       Running   0          47s       kubernetes-minion-9ye5
cassandra-q6sz7   1/1       Running   0          6m        kubernetes-minion-9ye5
```

一会儿你就能再次检查 Cassandra 集群的状态，你可以看到新的 pod 已经被自定义的 SeedProvider 检测到：

```
$ kubectl exec -ti cassandra-xxxxx -- nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens  Owns (effective)  Host ID
Rack
UN  10.244.0.6  51.67 KB   256     48.9%
d07b23a5-56a1-4b0b-952d-68ab95869163  rack1
UN  10.244.1.5  84.71 KB   256     50.7%            e060df1f-faa2-470c-923d-
ca049b0f3f38  rack1
UN  10.244.1.6  84.71 KB   256     47.0%
83ca1580-4f3c-4ec5-9b38-75036b7a297f  rack1
UN  10.244.0.5  68.2 KB    256     53.4%            72ca27e2-
c72c-402a-9313-1e4b61c2f839  rack1
```

步骤 7：删除 Replication Controller

在你开始步骤 5 之前， __删除__你在上面创建的 **replication controller。**

```
$ kubectl delete rc cassandra
```

步骤 8：使用 DaemonSet 替换 Replication Controller

在 Kubernetes中，*DaemonSet* 能够将 pod 一对一的分布到 Kubernetes 节点上。和 *ReplicationController* 相同的是它也有一个用于识别它的集合成员的 selector query。但和 *ReplicationController* 不同的是，它拥有一个节点 selector，用于限制基于模板的 pod 可以调度的节点。并且 pod 的复制不是基于一个设置的数量，而是为每一个节点分配一个 pod。

示范用例：当部署到云平台时，预期情况是实例是短暂的并且随时可能终止。Cassandra 被搭建成为在各个节点间复制数据以便于实现数据冗余。这样的话，即使一个实例终止了，存储在它上面的数据却没有，并且集群会通过重新复制数据到其它运行节点来作为响应。

DaemonSet 设计为在 Kubernetes 集群中的每个节点上放置一个 pod。那样就会给我们带来数据冗余度。让我们创建一个 DaemonSet 来启动我们的存储集群：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  labels:
    name: cassandra
  name: cassandra
spec:
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      # Filter to specific nodes:
      # nodeSelector:
      #  app: cassandra
      containers:
        - command:
            - /run.sh
          env:
            - name: MAX_HEAP_SIZE
              value: 512M
```

```
        - name: HEAP_NEWSIZE
          value: 100M
        - name: CASSANDRA_SEED_PROVIDER
          value: "io.k8s.cassandra.KubernetesSeedProvider"
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
      image: gcr.io/google-samples/cassandra:v12
      name: cassandra
      ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
          # If you need it, it will go away in C* 4.0.
          #- containerPort: 9160
          #  name: thrift
      resources:
        requests:
          cpu: 0.5
      volumeMounts:
        - mountPath: /cassandra_data
          name: data
  volumes:
    - name: data
      emptyDir: {}
```

下载示例

这个 DaemonSet 绝大部分的定义和上面的 ReplicationController 完全相同；它只是简单的给 daemonset 一个创建新的 Cassandra pod 的方法，并且以集群中所有的 Cassandra 节点为目标。

不同之处在于 `nodeSelector` 属性，它允许 DaemonSet 以全部节点的一个子集为目标（你可以向其他资源一样标记节点），并且没有 `replicas` 属性，因为它使用1对1的 node-pod 关系。

创建这个 DaemonSet：

```
$ kubectl create -f cassandra/cassandra-daemonset.yaml
```

你可能需要禁用配置文件检查，像这样：

```
$ kubectl create -f cassandra/cassandra-daemonset.yaml --validate=false
```

你可以看到 DaemonSet 已经在运行：

```
$ kubectl get daemonset
NAME        DESIRED   CURRENT   NODE-SELECTOR
cassandra   3         3         <none>
```

现在，如果你列出集群中的 pods，并且使用 app=cassandra 标签过滤，你应该能够看到你的网络中的每一个节点上都有一个（且只有一个）新的 cassandra pod。

```
$ kubectl get pods -l="app=cassandra" -o wide
NAME              READY     STATUS    RESTARTS   AGE       NODE
cassandra-ico4r   1/1       Running   0          4s        kubernetes-minion-rpo1
cassandra-kitfh   1/1       Running   0          1s        kubernetes-minion-9ye5
cassandra-tzw89   1/1       Running   0          2s        kubernetes-minion-b286
```

为了证明这是按设想的在工作，你可以再次使用 nodetool 命令来检查集群的状态。为此，请使用 kubectl exec 命令在任何一个新建的 cassandra pod 上运行 nodetool。

```
$ kubectl exec -ti cassandra-xxxxx -- nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens  Owns (effective)  Host ID
Rack
UN  10.244.0.5  74.09 KB   256     100.0%            86feda0f-f070-4a5b-
bda1-2eeb0ad08b77  rack1
UN  10.244.4.2  32.45 KB   256     100.0%            0b1be71a-6ffb-4895-ac3e-
b9791299c141  rack1
UN  10.244.3.3  51.28 KB   256     100.0%            dafe3154-1d67-42e1-
ac1d-78e7e80dce2b  rack1
```

注意：这个示例让你在创建 DaemonSet 前删除了 cassandra 的 Replication Controller。这是因为为了保持示例的简单，RC 和 DaemonSet 使用了相同的 app=cassandra 标签（如此它们的 pod 映射到了我们创建的 service，这样 SeedProvider 就能识别它们）。

如果我们没有预先删除 RC，这两个资源在需要运行多少 pod 上将会发生冲突。如果希望的话，我们可以使用额外的标签和 selectors 来支持同时运行它们。

步骤 9：资源清理

当你准备删除你的资源时，按以下执行：

```
$ kubectl delete service -l app=cassandra
$ kubectl delete daemonset cassandra
```

Seed Provider Source

我们使用了一个自定义的 SeedProvider 来在 Kubernetes 之上运行 Cassandra。仅当你通过 replication control 或者 daemonset 部署 Cassandra 时才需要使用自定义的 seed provider。在 Cassandra 中，SeedProvider 引导 Cassandra 使用 gossip 协议来查找其它 Cassandra 节点。Seed 地址是被视为连接端点的主机。Cassandra 实例使用 seed 列表来查找彼此并学习 ring 环拓扑。KubernetesSeedProvider 通过 Kubernetes API 发现 Cassandra seeds IP 地址，那些 Cassandra 实例在 Cassandra Service 中定义。

请查阅自定义 seed provider 的 README 文档，获取 KubernetesSeedProvider 进阶配置。对于本示例来说，你应该不需要自定义 Seed Provider 的配置。

查看本示例的 image 目录，了解如何构建容器的 docker 镜像及其内容。

你可能还注意到我们设置了一些 Cassandra 参数（MAX_HEAP_SIZE和HEAP_NEWSIZE），并且增加了关于 namespace 的信息。我们还告诉 Kubernetes 容器暴露了 CQL 和 Thrift API 端口。最后，我们告诉集群管理器我们需要 0.1 cpu（0.1 核）。

!Analytics]()

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 April 10, 2020 at 10:50 AM PST: replace zh to /zh in content/zh/docs/tutorials/ directory (4b334e536)

运行 ZooKeeper， 一个 CP 分布式系统

本教程展示了在 Kubernetes 上使用 StatefulSets，PodDisruptionBudgets 和 PodAntiAffinity 特性运行 Apache Zookeeper。

准备开始

在开始本教程前，你应该熟悉以下 Kubernetes 概念。

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- PersistentVolume Provisioning
- StatefulSets
- PodDisruptionBudgets
- PodAntiAffinity
- kubectl CLI

你需要一个至少包含四个节点的集群，每个节点至少 2 CPUs 和 4 GiB 内存。在本教程中你将会 cordon 和 drain 集群的节点。**这意味着集群节点上所有的 Pods 将会被终止并移除。这些节点也会暂时变为不可调度。**在本教程中你应该使用一个独占的集群，或者保证你造成的干扰不会影响其它租户。

本教程假设你的集群配置为动态的提供 PersistentVolumes。如果你的集群没有配置成这样，在开始本教程前，你需要手动准备三个 20 GiB 的卷。

教程目标

在学习本教程后，你将熟悉下列内容。

- 如何使用 StatefulSet 部署一个 ZooKeeper ensemble。
- 如何一致性配置 ensemble。
- 如何在 ensemble 中 分布 ZooKeeper 服务器的部署。
- 如何在计划维护中使用 PodDisruptionBudgets 确保服务可用性。

ZooKeeper 基础

Apache ZooKeeper 是一个分布式的开源协调服务，用于分布式系统。ZooKeeper 允许你读取、写入数据和发现数据更新。数据按层次结构组织在文件系统中，并复制到 ensemble（一个 ZooKeeper 服务器的集合） 中所有的 ZooKeeper 服务器。对数据的所有操作都是原子的和顺序一致的。ZooKeeper 通

过 Zab 一致性协议在 ensemble 的所有服务器之间复制一个状态机来确保这个特性。

ensemble 使用 Zab 协议选举一个 leader，在选举出 leader 前不能写入数据。一旦选举出了 leader，ensemble 使用 Zab 保证所有写入被复制到一个 quorum，然后这些写入操作才会被确认并对客户端可用。如果没有遵照加权 quorums，一个 quorum 表示包含当前 leader 的 ensemble 的多数成员。例如，如果 ensemble 有3个服务器，一个包含 leader 的成员和另一个服务器就组成了一个 quorum。如果 ensemble 不能达成一个 quorum，数据将不能被写入。

ZooKeeper 在内存中保存它们的整个状态机，但是每个改变都被写入一个在存储介质上的持久 WAL（Write Ahead Log）。当一个服务器故障时，它能够通过回放 WAL 恢复之前的状态。为了防止 WAL 无限制的增长，ZooKeeper 服务器会定期的将内存状态快照保存到存储介质。这些快照能够直接加载到内存中，所有在这个快照之前的 WAL 条目都可以被安全的丢弃。

创建一个 ZooKeeper Ensemble

下面的清单包含一个 Headless Service，一个 Service，一个 PodDisruptionBudget，和一个 StatefulSet。

application/zookeeper/zookeeper.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: zk-hs
  labels:
    app: zk
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
  clusterIP: None
  selector:
    app: zk
---
apiVersion: v1
kind: Service
metadata:
  name: zk-cs
  labels:
    app: zk
```

```yaml
spec:
  ports:
  - port: 2181
    name: client
  selector:
    app: zk
---
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  selector:
    matchLabels:
      app: zk
  maxUnavailable: 1
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: zk
spec:
  selector:
    matchLabels:
      app: zk
  serviceName: zk-hs
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: zk
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - zk
              topologyKey: "kubernetes.io/hostname"
      containers:
```

```
      - name: kubernetes-zookeeper
        imagePullPolicy: Always
        image: "k8s.gcr.io/kubernetes-zookeeper:1.0-3.4.10"
        resources:
          requests:
            memory: "1Gi"
            cpu: "0.5"
        ports:
        - containerPort: 2181
          name: client
        - containerPort: 2888
          name: server
        - containerPort: 3888
          name: leader-election
        command:
        - sh
        - -c
        - "start-zookeeper \
          --servers=3 \
          --data_dir=/var/lib/zookeeper/data \
          --data_log_dir=/var/lib/zookeeper/data/log \
          --conf_dir=/opt/zookeeper/conf \
          --client_port=2181 \
          --election_port=3888 \
          --server_port=2888 \
          --tick_time=2000 \
          --init_limit=10 \
          --sync_limit=5 \
          --heap=512M \
          --max_client_cnxns=60 \
          --snap_retain_count=3 \
          --purge_interval=12 \
          --max_session_timeout=40000 \
          --min_session_timeout=4000 \
          --log_level=INFO"
        readinessProbe:
          exec:
            command:
            - sh
            - -c
            - "zookeeper-ready 2181"
          initialDelaySeconds: 10
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
```

```
        - sh
        - -c
        - "zookeeper-ready 2181"
      initialDelaySeconds: 10
      timeoutSeconds: 5
    volumeMounts:
    - name: datadir
      mountPath: /var/lib/zookeeper
  securityContext:
    runAsUser: 1000
    fsGroup: 1000
volumeClaimTemplates:
- metadata:
    name: datadir
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 10Gi
```

打开一个命令行终端，使用 kubectl apply 创建这个清单。

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

这个操作创建了 zk-hs Headless Service、zk-cs Service、zk-pdb PodDisruptionBudget 和 zk StatefulSet。

```
service/zk-hs created
service/zk-cs created
poddisruptionbudget.policy/zk-pdb created
statefulset.apps/zk created
```

使用 kubectl get 查看 StatefulSet 控制器创建的 Pods。

```
kubectl get pods -w -l app=zk
```

一旦 zk-2 Pod 变成 Running 和 Ready 状态，使用 CRTL-C 结束 kubectl。

```
NAME    READY   STATUS    RESTARTS   AGE
zk-0    0/1     Pending   0          0s
zk-0    0/1     Pending   0          0s
zk-0    0/1     ContainerCreating   0          0s
zk-0    0/1     Running   0          19s
zk-0    1/1     Running   0          40s
zk-1    0/1     Pending   0          0s
zk-1    0/1     Pending   0          0s
```

```
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

StatefulSet 控制器创建了3个 Pods，每个 Pod 包含一个 ZooKeeper 服务器。

促成 Leader 选举

由于在匿名网络中没有用于选举 leader 的终止算法，Zab 要求显式的进行成员关系配置，以执行 leader 选举。Ensemble 中的每个服务器都需要具有一个独一无二的标识符，所有的服务器均需要知道标识符的全集，并且每个标识符都需要和一个网络地址相关联。

使用 kubectl exec 获取 zk StatefulSet 中 Pods 的主机名。

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname; done
```

StatefulSet 控制器基于每个 Pod 的序号索引为它们各自提供一个唯一的主机名。主机名采用 <statefulset name>-<ordinal index> 的形式。由于 zk StatefulSet 的 replicas 字段设置为3，这个 Set 的控制器将创建3个 Pods，主机名为：zk-0、zk-1 和 zk-2。

```
zk-0
zk-1
zk-2
```

ZooKeeper ensemble 中的服务器使用自然数作为唯一标识符，每个服务器的标识符都保存在服务器的数据目录中一个名为 myid 的文件里。

检查每个服务器的 myid 文件的内容。

```
for i in 0 1 2; do echo "myid zk-$i";kubectl exec zk-$i -- cat /var/lib/
zookeeper/data/myid; done
```

由于标识符为自然数并且序号索引是非负整数，你可以在序号上加 1 来生成一个标识符。

```
myid zk-0
1
myid zk-1
2
myid zk-2
3
```

获取 zk StatefulSet 中每个 Pod 的 FQDN (Fully Qualified Domain Name，正式域名)。

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname -f; done
```

zk-hs Service 为所有 Pods 创建了一个 domain：zk-hs.default.svc.cluster.local。

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

Kubernetes DNS 中的 A 记录将 FQDNs 解析成为 Pods 的 IP 地址。如果 Pods 被调度，这个 A 记录将会使用 Pods 的新 IP 地址更新，但 A 记录的名称不会改变。

ZooKeeper 在一个名为 zoo.cfg 的文件中保存它的应用配置。使用 kubectl exec 在 zk-0 Pod 中查看 zoo.cfg 文件的内容。

```
kubectl exec zk-0 -- cat /opt/zookeeper/conf/zoo.cfg
```

文件底部为 server.1、server.2 和 server.3，其中的 1、2和3分别对应 ZooKeeper 服务器的 myid 文件中的标识符。它们被设置为 zk StatefulSet 中的 Pods 的 FQDNs。

```
clientPort=2181
dataDir=/var/lib/zookeeper/data
dataLogDir=/var/lib/zookeeper/log
tickTime=2000
initLimit=10
syncLimit=2000
maxClientCnxns=60
minSessionTimeout= 4000
maxSessionTimeout= 40000
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
server.1=zk-0.zk-hs.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:3888
```

达成一致

一致性协议要求每个参与者的标识符唯一。在 Zab 协议里任何两个参与者都不应该声明相同的唯一标识符。对于让系统中的进程协商哪些进程已经提交了哪些数据而言，这是必须的。如果有两个 Pods 使用相同的序号启动，这两个 ZooKeeper 服务器会将自己识别为相同的服务器。

```
kubectl get pods -w -l app=zk
```

```
NAME     READY     STATUS    RESTARTS   AGE
zk-0     0/1       Pending   0          0s
zk-0     0/1       Pending   0          0s
zk-0     0/1       ContainerCreating   0          0s
zk-0     0/1       Running   0          19s
zk-0     1/1       Running   0          40s
zk-1     0/1       Pending   0          0s
zk-1     0/1       Pending   0          0s
zk-1     0/1       ContainerCreating   0          0s
zk-1     0/1       Running   0          18s
zk-1     1/1       Running   0          40s
zk-2     0/1       Pending   0          0s
zk-2     0/1       Pending   0          0s
zk-2     0/1       ContainerCreating   0          0s
zk-2     0/1       Running   0          19s
zk-2     1/1       Running   0          40s
```

每个 Pod 的 A 记录仅在 Pod 变成 Ready状态时被录入。因此，ZooKeeper 服务器的 FQDNs 只会解析到一个 endpoint，而那个 endpoint 将会是一个唯一的 ZooKeeper 服务器，这个服务器声明了配置在它的 myid 文件中的标识符。

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

这保证了 ZooKeepers 的 zoo.cfg 文件中的 servers 属性代表了一个正确配置的 ensemble。

```
server.1=zk-0.zk-hs.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:3888
```

当服务器使用 Zab 协议尝试提交一个值的时候，它们会达成一致并成功提交这个值（如果 leader 选举成功并且至少有两个 Pods 处于 Running 和 Ready状态），或者将会失败（如果没有满足上述条件中的任意一条）。当一个服务器承认另一个服务器的代写时不会有状态产生。

Ensemble 健康检查

最基本的健康检查是向一个 ZooKeeper 服务器写入一些数据，然后从另一个服务器读取这些数据。

使用 zkCli.sh 脚本在 zk-0 Pod 上写入 world 到路径 /hello。

```
kubectl exec zk-0 zkCli.sh create /hello world
```

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:None path:null
Created /hello
```

从 zk-1 Pod 获取数据。

```
kubectl exec zk-1 zkCli.sh get /hello
```

你在 zk-0 创建的数据在 ensemble 中所有的服务器上都是可用的。

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

准备持久存储

如同在 ZooKeeper 基础 一节所提到的，ZooKeeper 提交所有的条目到一个持久 WAL，并周期性的将内存快照写入存储介质。对于使用一致性协议实现一个复制状态机的应用来说，使用 WALs 提供持久化是一种常用的技术，对于普通的存储应用也是如此。

使用 kubectl delete 删除 zk StatefulSet。

```
kubectl delete statefulset zk
```

```
statefulset.apps "zk" deleted
```

观察 StatefulSet 中的 Pods 变为终止状态。

```
kubectl get pods -w -l app=zk
```

当 zk-0 完全终止时，使用 CRTL-C 结束 kubectl。

```
zk-2      1/1       Terminating   0         9m
zk-0      1/1       Terminating   0         11m
zk-1      1/1       Terminating   0         10m
zk-2      0/1       Terminating   0         9m
zk-2      0/1       Terminating   0         9m
```

```
zk-2      0/1       Terminating   0         9m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
```

重新应用 zookeeper.yaml 中的代码清单。

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

zk StatefulSet 将会被创建。由于清单中的其他 API 对象已经存在，所以它们不会被修改。

观察 StatefulSet 控制器重建 StatefulSet 的 Pods。

```
kubectl get pods -w -l app=zk
```

一旦 zk-2 Pod 处于 Running 和 Ready 状态，使用 CRTL-C 停止 kubectl命令。

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

从 zk-2 Pod 中获取你在健康检查中输入的值。

```
kubectl exec zk-2 zkCli.sh get /hello
```

尽管 zk StatefulSet 中所有的 Pods 都已经被终止并重建过，ensemble 仍然使用原来的数值提供服务。

```
WATCHER::
```

```
WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

zk StatefulSet 的 spec 中的 volumeClaimTemplates 字段标识了将要为每个 Pod 准备的 PersistentVolume。

```
volumeClaimTemplates:
  - metadata:
      name: datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 20Gi
```

StatefulSet 控制器为 StatefulSet 中的每个 Pod 生成一个 PersistentVolumeClaim。

获取 StatefulSet 的 PersistentVolumeClaims。

```
kubectl get pvc -l app=zk
```

当 StatefulSet 重新创建它的 Pods时，Pods 的 PersistentVolumes 会被重新挂载。

```
NAME          STATUS    VOLUME                                     CAPACITY
ACCESSMODES   AGE
datadir-zk-0  Bound     pvc-bed742cd-bcb1-11e6-994f-42010a800002
20Gi      RWO         1h
datadir-zk-1  Bound     pvc-bedd27d2-bcb1-11e6-994f-42010a800002
20Gi      RWO         1h
datadir-zk-2  Bound     pvc-bee0817e-bcb1-11e6-994f-42010a800002
20Gi      RWO         1h
```

StatefulSet 的容器 template 中的 volumeMounts 一节使得 PersistentVolumes 被挂载到 ZooKeeper 服务器的数据目录。

```
volumeMounts:
      - name: datadir
        mountPath: /var/lib/zookeeper
```

当 zk StatefulSet 中的一个 Pod 被（重新）调度时，它总是拥有相同的 PersistentVolume，挂载到 ZooKeeper 服务器的数据目录。即使在 Pods 被重新调度时，所有对 ZooKeeper 服务器的 WALs 的写入和它们的全部快照都仍然是持久的。

确保一致性配置

如同在 促成 leader 选举 和 达成一致 小节中提到的，ZooKeeper ensemble 中的服务器需要一致性的配置来选举一个 leader 并形成一个 quorum。它们还需要 Zab 协议的一致性配置来保证这个协议在网络中正确的工作。在这次的样例中，我们通过直接将配置写入代码清单中来达到该目的。

获取 zk StatefulSet。

```
kubectl get sts zk -o yaml
```

```
...
command:
      - sh
      - -c
      - "start-zookeeper \
        --servers=3 \
        --data_dir=/var/lib/zookeeper/data \
        --data_log_dir=/var/lib/zookeeper/data/log \
        --conf_dir=/opt/zookeeper/conf \
        --client_port=2181 \
        --election_port=3888 \
        --server_port=2888 \
        --tick_time=2000 \
        --init_limit=10 \
        --sync_limit=5 \
        --heap=512M \
        --max_client_cnxns=60 \
        --snap_retain_count=3 \
        --purge_interval=12 \
        --max_session_timeout=40000 \
        --min_session_timeout=4000 \
        --log_level=INFO"
...
```

用于启动 ZooKeeper 服务器的命令将这些配置作为命令行参数传给了 ensemble。你也可以通过环境变量来传入这些配置。

配置日志

zkGenConfig.sh 脚本产生的一个文件控制了 ZooKeeper 的日志行为。ZooKeeper 使用了 Log4j 并默认使用基于文件大小和时间的滚动文件追加器作为日志配置。

从 zk StatefulSet 的一个 Pod 中获取日志配置。

```
kubectl exec zk-0 cat /usr/etc/zookeeper/log4j.properties
```

下面的日志配置会使 ZooKeeper 进程将其所有的日志写入标志输出文件流中。

```
zookeeper.root.logger=CONSOLE
zookeeper.console.threshold=INFO
log4j.rootLogger=${zookeeper.root.logger}
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=${zookeeper.console.threshold}
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} [myid:
%X{myid}] - %-5p [%t:%C{1}@%L] - %m%n
```

这是在容器里安全记录日志的最简单的方法。由于应用的日志被写入标准输出，Kubernetes 将会为你处理日志轮转。Kubernetes 还实现了一个智能保存策略，保证写入标准输出和标准错误流的应用日志不会耗尽本地存储媒介。

使用 kubectl logs 从一个 Pod 中取回最后几行日志。

```
kubectl logs zk-0 --tail 20
```

使用 kubectl logs 或者从 Kubernetes Dashboard 可以查看写入到标准输出和标准错误流中的应用日志。

```
2016-12-06 19:34:16,236 [myid:1] - INFO  [NIOServerCxn.Factory:
0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command
from /127.0.0.1:52740
2016-12-06 19:34:16,237 [myid:1] - INFO
[Thread-1136:NIOServerCnxn@1008] - Closed socket connection for client /
127.0.0.1:52740 (no session established for client)
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:
0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket
connection from /127.0.0.1:52749
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:
0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command
from /127.0.0.1:52749
2016-12-06 19:34:26,156 [myid:1] - INFO
[Thread-1137:NIOServerCnxn@1008] - Closed socket connection for client /
```

127.0.0.1:52749 (no session established for client)
2016-12-06 19:34:26,222 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from /127.0.0.1:52750
2016-12-06 19:34:26,222 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command from /127.0.0.1:52750
2016-12-06 19:34:26,226 [myid:1] - INFO [Thread-1138:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52750 (no session established for client)
2016-12-06 19:34:36,151 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from /127.0.0.1:52760
2016-12-06 19:34:36,152 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command from /127.0.0.1:52760
2016-12-06 19:34:36,152 [myid:1] - INFO [Thread-1139:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52760 (no session established for client)
2016-12-06 19:34:36,230 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from /127.0.0.1:52761
2016-12-06 19:34:36,231 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command from /127.0.0.1:52761
2016-12-06 19:34:36,231 [myid:1] - INFO [Thread-1140:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52761 (no session established for client)
2016-12-06 19:34:46,149 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from /127.0.0.1:52767
2016-12-06 19:34:46,149 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command from /127.0.0.1:52767
2016-12-06 19:34:46,149 [myid:1] - INFO [Thread-1141:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52767 (no session established for client)
2016-12-06 19:34:46,230 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket connection from /127.0.0.1:52768
2016-12-06 19:34:46,230 [myid:1] - INFO [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command from /127.0.0.1:52768
2016-12-06 19:34:46,230 [myid:1] - INFO [Thread-1142:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52768 (no session established for client)

Kubernetes 支持与 Stackdriver 和 Elasticsearch and Kibana 的整合以获得复杂但更为强大的日志功能。 对于集群级别的日志输出与整合，可以考虑部署一个 sidecar 容器。

配置非特权用户

在容器中允许应用以特权用户运行这条最佳实践是值得商讨的。如果你的组织要求应用以非特权用户运行，你可以使用 SecurityContext 控制运行容器入口点的用户。

zk StatefulSet 的 Pod 的 template 包含了一个 SecurityContext。

```
securityContext:
  runAsUser: 1000
  fsGroup: 1000
```

在 Pods 的容器内部，UID 1000 对应用户 zookeeper，GID 1000对应用户组 zookeeper。

从 zk-0 Pod 获取 ZooKeeper 进程信息。

```
kubectl exec zk-0 -- ps -elf
```

由于 securityContext 对象的 runAsUser 字段被设置为1000而不是 root，ZooKeeper 进程将以 zookeeper 用户运行。

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S zookeep+     1     0  0  80   0 -  1127 -      20:46 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.sh start-foreground
0 S zookeep+    27     1  0  80   0 - 1155556 -    20:46 ?        00:00:19 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -Dzookeeper.log.dir=/var/log/zookeeper -Dzookeeper.root.logger=INFO,CONSOLE -cp /usr/bin/../build/classes:/usr/bin/../build/lib/*.jar:/usr/bin/../share/zookeeper/zookeeper-3.4.9.jar:/usr/bin/../share/zookeeper/slf4j-log4j12-1.6.1.jar:/usr/bin/../share/zookeeper/slf4j-api-1.6.1.jar:/usr/bin/../share/zookeeper/netty-3.10.5.Final.jar:/usr/bin/../share/zookeeper/log4j-1.2.16.jar:/usr/bin/../share/zookeeper/jline-0.9.94.jar:/usr/bin/../src/java/lib/*.jar:/usr/bin/../etc/zookeeper: -Xmx2G -Xms2G -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=false org.apache.zookeeper.server.quorum.QuorumPeerMain /usr/bin/../etc/zookeeper/zoo.cfg
```

默认情况下，当 Pod 的 PersistentVolume 被挂载到 ZooKeeper 服务器的数据目录时，它只能被 root 用户访问。这个配置将阻止 ZooKeeper 进程写入它的 WAL 及保存快照。

在 zk-0 Pod 上获取 ZooKeeper 数据目录的文件权限。

```
kubectl exec -ti zk-0 -- ls -ld /var/lib/zookeeper/data
```

由于 securityContext 对象的 fsGroup 字段设置为1000，Pods 的 PersistentVolumes 的所有权属于 zookeeper 用户组，因而 ZooKeeper 进程能够成功的读写数据。

```
drwxr-sr-x 3 zookeeper zookeeper 4096 Dec  5 20:45 /var/lib/zookeeper/
data
```

管理 ZooKeeper 进程

ZooKeeper documentation 文档指出"你将需要一个监管程序用于管理每个 ZooKeeper 服务进程（JVM）"。在分布式系统中，使用一个看门狗（监管程序）来重启故障进程是一种常用的模式。

更新 Ensemble

zk StatefulSet 的更新策略被设置为了 RollingUpdate。

你可以使用 kubectl patch 更新分配给每个服务器的 cpus 的数量。

```
kubectl patch sts zk --type='json' -p='[{"op": "replace", "path": "/spec/
template/spec/containers/0/resources/requests/cpu", "value":"0.3"}]'
```

```
statefulset.apps/zk patched
```

使用 kubectl rollout status 观测更新状态。

```
kubectl rollout status sts/zk
```

```
waiting for statefulset rolling update to complete 0 pods at revision
zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 1 pods at revision
zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 2 pods at revision
zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
statefulset rolling update complete 3 pods at revision zk-5db4499664...
```

这项操作会逆序地依次终止每一个 Pod，并用新的配置重新创建。这样做确保了在滚动更新的过程中 quorum 依旧保持工作。

使用 kubectl rollout history 命令查看历史或先前的配置。

```
kubectl rollout history sts/zk
```

```
statefulsets "zk"
REVISION
1
2
```

使用 kubectl rollout undo 命令撤销这次的改动。

```
kubectl rollout undo sts/zk
```

```
statefulset.apps/zk rolled back
```

处理进程故障

Restart Policies 控制 Kubernetes 如何处理一个 Pod 中容器入口点的进程故障。对于 StatefulSet 中的 Pods 来说，Always 是唯一合适的 RestartPolicy，这也是默认值。你应该**绝不**覆盖 stateful 应用的默认策略。

检查 zk-0 Pod 中运行的 ZooKeeper 服务器的进程树。

```
kubectl exec zk-0 -- ps -ef
```

作为容器入口点的命令的 PID 为 1，Zookeeper 进程是入口点的子进程，PID 为 27。

```
UID        PID  PPID  C STIME TTY          TIME CMD
zookeep+     1     0  0 15:03 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.sh start-foreground
zookeep+    27     1  0 15:03 ?        00:00:03 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -Dzookeeper.log.dir=/var/log/zookeeper -Dzookeeper.root.logger=INFO,CONSOLE -cp /usr/bin/../build/classes:/usr/bin/../build/lib/*.jar:/usr/bin/../share/zookeeper/zookeeper-3.4.9.jar:/usr/bin/../share/zookeeper/slf4j-log4j12-1.6.1.jar:/usr/bin/../share/zookeeper/slf4j-api-1.6.1.jar:/usr/bin/../share/zookeeper/netty-3.10.5.Final.jar:/usr/bin/../share/zookeeper/log4j-1.2.16.jar:/usr/bin/../share/zookeeper/jline-0.9.94.jar:/usr/bin/../src/java/lib/*.jar:/usr/bin/../etc/zookeeper: -Xmx2G -Xms2G -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=false org.apache.zookeeper.server.quorum.QuorumPeerMain /usr/bin/../etc/zookeeper/zoo.cfg
```

在一个终端观察 zk StatefulSet 中的 Pods。

```
kubectl get pod -w -l app=zk
```

在另一个终端杀掉 Pod zk-0 中的 ZooKeeper 进程。

```
 kubectl exec zk-0 -- pkill java
```

ZooKeeper 进程的终结导致了它父进程的终止。由于容器的 RestartPolicy 是 Always，父进程被重启。

```
NAME    READY    STATUS    RESTARTS   AGE
zk-0    1/1      Running   0          21m
zk-1    1/1      Running   0          20m
zk-2    1/1      Running   0          19m
NAME    READY    STATUS    RESTARTS   AGE
zk-0    0/1      Error     0          29m
zk-0    0/1      Running   1          29m
zk-0    1/1      Running   1          29m
```

如果你的应用使用一个脚本（例如 zkServer.sh）来启动一个实现了应用业务逻辑的进程，这个脚本必须和子进程一起结束。这保证了当实现应用业务逻辑的进程故障时，Kubernetes 会重启这个应用的容器。

存活性测试

你的应用配置为自动重启故障进程，但这对于保持一个分布式系统的健康来说是不够的。许多场景下，一个系统进程可以是活动状态但不响应请求，或者是不健康状态。你应该使用 liveness probes 来通知 Kubernetes 你的应用进程处于不健康状态，需要被重启。

zk StatefulSet 的 Pod 的 template 一节指定了一个存活探针。

```
livenessProbe:
        exec:
          command:
          - sh
          - -c
          - "zookeeper-ready 2181"
        initialDelaySeconds: 15
        timeoutSeconds: 5
```

这个探针调用一个简单的 bash 脚本，使用 ZooKeeper 的四字缩写 ruok 来测试服务器的健康状态。

```
OK=$(echo ruok | nc 127.0.0.1 $1)
if [ "$OK" == "imok" ]; then
    exit 0
else
    exit 1
fi
```

在一个终端窗口观察 zk StatefulSet 中的 Pods。

```
kubectl get pod -w -l app=zk
```

在另一个窗口中，从 Pod zk-0 的文件系统中删除 zookeeper-ready 脚本。

```
kubectl exec zk-0 -- rm /usr/bin/zookeeper-ready
```

当 ZooKeeper 进程的存活探针探测失败时，Kubernetes 将会为你自动重启这个进程，从而保证 ensemble 中不健康状态的进程都被重启。

```
kubectl get pod -w -l app=zk
```

```
NAME     READY    STATUS    RESTARTS   AGE
zk-0     1/1      Running   0          1h
zk-1     1/1      Running   0          1h
zk-2     1/1      Running   0          1h
NAME     READY    STATUS    RESTARTS   AGE
zk-0     0/1      Running   0          1h
zk-0     0/1      Running   1         1h
zk-0     1/1      Running   1         1h
```

就绪性测试

就绪不同于存活。如果一个进程是存活的，它是可调度和健康的。如果一个进程是就绪的，它应该能够处理输入。存活是就绪的必要非充分条件。在许多场景下，特别是初始化和终止过程中，一个进程可以是存活但没有就绪的。

如果你指定了一个就绪探针，Kubernetes将保证在就绪检查通过之前，你的应用不会接收到网络流量。

对于一个 ZooKeeper 服务器来说，存活即就绪。因此 zookeeper.yaml 清单中的就绪探针和存活探针完全相同。

```
readinessProbe:
  exec:
    command:
    - sh
    - -c
    - "zookeeper-ready 2181"
  initialDelaySeconds: 15
  timeoutSeconds: 5
```

虽然存活探针和就绪探针是相同的，但同时指定它们两者仍然重要。这保证了 ZooKeeper ensemble 中只有健康的服务器能接收网络流量。

容忍节点故障

ZooKeeper 需要一个 quorum 来提交数据变动。对于一个拥有 3 个服务器的 ensemble来说，必须有两个服务器是健康的，写入才能成功。在基于 quorum 的系统里，成员被部署在故障域之间以保证可用性。为了防止由于某台机器断连引起服务中断，最佳实践是防止应用的多个示例在相同的机器上共存。

默认情况下，Kubernetes 可以把 StatefulSet 的 Pods 部署在相同节点上。对于你创建的 3 个服务器的 ensemble 来说，如果有两个服务器并存于相同的节点上

并且该节点发生故障时，ZooKeeper 服务将中断，直至至少一个 Pods 被重新调度。

你应该总是提供额外的容量以允许关键系统进程在节点故障时能够被重新调度。如果你这样做了，服务故障就只会持续到 Kubernetes 调度器重新调度某个 ZooKeeper 服务器为止。但是，如果希望你的服务在容忍节点故障时无停服时间，你应该设置 podAntiAffinity。

获取 zk Stateful Set 中的 Pods 的节点。

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo "
"; done
```

zk StatefulSe 中所有的 Pods 都被部署在不同的节点。

```
kubernetes-node-cxpk
kubernetes-node-a5aq
kubernetes-node-2g2d
```

这是因为 zk StatefulSet 中的 Pods 指定了 PodAntiAffinity。

```
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: "app"
                  operator: In
                  values:
                  - zk
            topologyKey: "kubernetes.io/hostname"
```

requiredDuringSchedulingIgnoredDuringExecution 告诉 Kubernetes 调度器，在以 topologyKey 指定的域中，绝对不要把带有键为 app，值为 zk 的标签的两个 Pods 调度到相同的节点。topologyKey kubernetes.io/hostname 表示这个域是一个单独的节点。使用不同的 rules、labels 和 selectors，你能够通过这种技术把你的 ensemble 分布在不同的物理、网络和电力故障域之间。

存活管理

在本节中你将会 cordon 和 drain 节点。如果你是在一个共享的集群里使用本教程，请保证不会影响到其他租户

上一小节展示了如何在节点之间分散 Pods 以在计划外的节点故障时保证服务存活。但是你也需要为计划内维护引起的临时节点故障做准备。

获取你集群中的节点。

```
kubectl get nodes
```

使用 kubectl cordon cordon 你的集群中除4个节点以外的所有节点。

```
kubectl cordon <node-name>
```

获取 zk-pdb PodDisruptionBudget。

```
kubectl get pdb zk-pdb
```

max-unavailable 字段指示 Kubernetes 在任何时候，zk StatefulSet 至多有一个 Pod 是不可用的。

```
NAME      MIN-AVAILABLE   MAX-UNAVAILABLE   ALLOWED-DISRUPTIONS   AGE
zk-pdb    N/A             1                 1
```

在一个终端观察 zk StatefulSet 中的 Pods。

```
kubectl get pods -w -l app=zk
```

在另一个终端获取 Pods 当前调度的节点。

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo "
"; done
```

```
kubernetes-node-pb41
kubernetes-node-ixsl
kubernetes-node-i4c4
```

使用 kubectl drain 来 cordon 和 drain zk-0 Pod 调度的节点。

```
kubectl drain $(kubectl get pod zk-0 --template {{.spec.nodeName}}) --
ignore-daemonsets --force --delete-local-data
```

```
node "kubernetes-node-pb41" cordoned

WARNING: Deleting pods not managed by ReplicationController,
ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-
pb41, kube-proxy-kubernetes-node-pb41; Ignoring DaemonSet-managed
pods: node-problem-detector-v0.1-o5elz
pod "zk-0" deleted
node "kubernetes-node-pb41" drained
```

由于你的集群中有4个节点, kubectl drain 执行成功，`zk-0 被调度到其它节点。

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS      RESTARTS   AGE
```

```
zk-0      1/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
```

在第一个终端持续观察 StatefulSet 的 Pods并 drain zk-1 调度的节点。

```
kubectl drain $(kubectl get pod zk-1 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-local-data "kubernetes-node-ixsl"
cordoned
```

```
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-ixsl, kube-proxy-kubernetes-node-ixsl; Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-voc74
pod "zk-1" deleted
node "kubernetes-node-ixsl" drained
```

zk-1 Pod 不能被调度。由于 zk StatefulSet 包含了一个防止 Pods 共存的 PodAntiAffinity 规则，而且只有两个节点可用于调度，这个 Pod 将保持在 Pending 状态。

```
kubectl get pods -w -l app=zk
```

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2         1h
zk-1      1/1       Running   0         1h
zk-2      1/1       Running   0         1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
zk-1      1/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
```

```
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
```

继续观察 stateful set 的 Pods 并 drain zk-2 调度的节点。

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-local-data
```

```
node "kubernetes-node-i4c4" cordoned

WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-i4c4, kube-proxy-kubernetes-node-i4c4; Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog
WARNING: Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog; Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-i4c4, kube-proxy-kubernetes-node-i4c4
There are pending pods when an error occurred: Cannot evict pod as it would violate the pod's disruption budget.
pod/zk-2
```

使用 CRTL-C 终止 kubectl。

你不能 drain 第三个节点，因为删除 zk-2 将和 zk-budget 冲突。然而这个节点仍然保持 cordoned。

使用 zkCli.sh 从 zk-0 取回你的健康检查中输入的数值。

```
kubectl exec zk-0 zkCli.sh get /hello
```

由于遵守了 PodDisruptionBudget，服务仍然可用。

```
WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x200000002
ctime = Wed Dec 07 00:08:59 UTC 2016
mZxid = 0x200000002
mtime = Wed Dec 07 00:08:59 UTC 2016
pZxid = 0x200000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

使用 kubectl uncordon 来取消对第一个节点的隔离。

```
kubectl uncordon kubernetes-node-pb41
```

```
node "kubernetes-node-pb41" uncordoned
```

zk-1 被重新调度到了这个节点。等待 zk-1 变为 Running 和 Ready 状态。

```
kubectl get pods -w -l app=zk
```

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Terminating   2          2h
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0          0s
zk-0      0/1       ContainerCreating   0          0s
zk-0      0/1       Running   0          51s
zk-0      1/1       Running   0          1m
zk-1      1/1       Terminating   0          2h
zk-1      0/1       Terminating   0          2h
zk-1      0/1       Terminating   0          2h
zk-1      0/1       Terminating   0          2h
zk-1      0/1       Pending   0          0s
zk-1      0/1       Pending   0          0s
zk-1      0/1       Pending   0          12m
zk-1      0/1       ContainerCreating   0          12m
zk-1      0/1       Running   0          13m
zk-1      1/1       Running   0          13m
```

尝试 drain zk-2 调度的节点。

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-local-data
```

输出：

```
node "kubernetes-node-i4c4" already cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-i4c4, kube-proxy-kubernetes-node-i4c4; Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog
pod "heapster-v1.2.0-2604621511-wht1r" deleted
pod "zk-2" deleted
node "kubernetes-node-i4c4" drained
```

这次 kubectl drain 执行成功。

Uncordon 第二个节点以允许 zk-2 被重新调度。

```
kubectl uncordon kubernetes-node-ixsl
```

```
node "kubernetes-node-ixsl" uncordoned
```

你可以同时使用 kubectl drain 和 PodDisruptionBudgets 来保证你的服务在维护过程中仍然可用。如果使用了 drain 来隔离节点并在节点离线之前排出了 pods，那么表达了 disruption budget 的服务将会遵守该 budget。你应该总是为关键服务分配额外容量，这样它们的 Pods 就能够迅速的重新调度。

清理现场

- 使用 kubectl uncordon 解除你集群中所有节点的隔离。
- 你需要删除在本教程中使用的 PersistentVolumes 的持久存储媒介。请遵循必须的步骤，基于你的环境、存储配置和准备方法，保证回收所有的存储。

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 06, 2021 at 1:38 AM PST: Incorrect url redirect. (b48dfd7f1)

集群

使用 Seccomp 限制容器的系统调用

AppArmor

使用 Seccomp 限制容器的系统调用

FEATURE STATE: Kubernetes v1.19 [stable]

Seccomp 代表安全计算模式，自 2.6.12 版本以来一直是 Linux 内核的功能。 它可以用来对进程的特权进行沙盒处理，从而限制了它可以从用户空间向内核进行的调用。 Kubernetes 允许你将加载到节点上的 seccomp 配置文件自动应用于 Pod 和容器。

确定工作负载所需的特权可能很困难。在本教程中，你将了解如何将 seccomp 配置文件 加载到本地 Kubernetes 集群中，如何将它们应用到 Pod，以及如何开始制作仅向容器 进程提供必要特权的配置文件。

教程目标

- 了解如何在节点上加载 seccomp 配置文件
- 了解如何将 seccomp 配置文件应用于容器
- 观察由容器进程进行的系统调用的审核
- 观察当指定了一个不存在的配置文件时的行为
- 观察违反 seccomp 配置的情况
- 了解如何创建精确的 seccomp 配置文件
- 了解如何应用容器运行时默认 seccomp 配置文件

准备开始

为了完成本教程中的所有步骤，你必须安装 kind 和 kubectl。本教程将显示同时具有 alpha（v1.19 之前的版本）和通常可用的 seccomp 功能的示例，因此请确保为所使用的版本正确配置了集群。

创建 Seccomp 文件

这些配置文件的内容将在以后进行探讨，但现在继续进行，并将其下载到名为 profiles/ 的目录中，以便可以将其加载到集群中。

- audit.json
- violation.json
- fine-grained.json

pods/security/seccomp/profiles/audit.json

```
{
    "defaultAction": "SCMP_ACT_LOG"
}
```

pods/security/seccomp/profiles/violation.json

```
{
    "defaultAction": "SCMP_ACT_ERRNO"
}
```

pods/security/seccomp/profiles/fine-grained.json

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "accept4",
                "epoll_wait",
                "pselect6",
                "futex",
                "madvise",
                "epoll_ctl",
                "getsockname",
                "setsockopt",
                "vfork",
                "mmap",
                "read",
                "write",
                "close",
                "arch_prctl",
                "sched_getaffinity",
                "munmap",
                "brk",
                "rt_sigaction",
                "rt_sigprocmask",
                "sigaltstack",
                "gettid",
                "clone",
                "bind",
                "socket",
                "openat",
                "readlinkat",
                "exit_group",
                "epoll_create1",
                "listen",
                "rt_sigreturn",
                "sched_yield",
                "clock_gettime",
                "connect",
                "dup2",
                "epoll_pwait",
                "execve",
```

```
                "exit",
                "fcntl",
                "getpid",
                "getuid",
                "ioctl",
                "mprotect",
                "nanosleep",
                "open",
                "poll",
                "recvfrom",
                "sendto",
                "set_tid_address",
                "setitimer",
                "writev"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

使用 Kind 创建一个本地 Kubernetes 集群

为简单起见，可以使用 kind 创建一个已经加载 seccomp 配置文件的单节点集群。 Kind 在 Docker 中运行 Kubernetes，因此集群的每个节点实际上只是一个容器。这允许将文件挂载到每个容器的文件系统中， 就像将文件挂载到节点上一样。

pods/security/seccomp/kind.yaml

```yaml
apiVersion: kind.x-k8s.io/v1alpha4
kind: Cluster
nodes:
- role: control-plane
  extraMounts:
  - hostPath: "./profiles"
    containerPath: "/var/lib/kubelet/seccomp/profiles"
```

下载上面的这个示例，并将其保存为 kind.yaml。然后使用这个配置创建集群。

```
kind create cluster --config=kind.yaml
```

一旦这个集群已经就绪，找到作为单节点集群运行的容器：

```
docker ps
```

你应该看到输出显示正在运行的容器名称为 kind-control-plane。

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS                       NAMES
6a96207fed4b        kindest/node:v1.18.2   "/usr/local/bin/entr..."   27 seconds ago      Up 24 seconds       127.0.0.1:42223->6443/tcp   kind-control-plane
```

如果观察该容器的文件系统，则应该看到 profiles/ 目录已成功加载到 kubelet 的默认 seccomp 路径中。 使用 docker exec 在 Pod 中运行命令：

```
docker exec -it 6a96207fed4b ls /var/lib/kubelet/seccomp/profiles
```

```
audit.json  fine-grained.json  violation.json
```

使用 Seccomp 配置文件创建 Pod 以进行系统调用审核

首先，将 audit.json 配置文件应用到新的 Pod 中，该配置文件将记录该进程的所有系统调用。

为你的 Kubernetes 版本下载正确的清单：

- v1.19 或更新版本（GA）
- v1.19之前版本（alpha）

pods/security/seccomp/ga/audit-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

pods/security/seccomp/alpha/audit-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: localhost/profiles/audit.json
spec:
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

在集群中创建 Pod：

```
kubectl apply -f audit-pod.yaml
```

这个配置文件并不限制任何系统调用，所以这个 Pod 应该会成功启动。

```
kubectl get pod/audit-pod
```

```
NAME        READY   STATUS    RESTARTS   AGE
audit-pod   1/1     Running   0          30s
```

为了能够与该容器公开的端点进行交互，请创建一个 NodePort 服务，该服务允许从 kind 控制平面容器内部访问该端点。

```
kubectl expose pod/audit-pod --type NodePort --port 5678
```

检查这个服务在这个节点上被分配了什么端口。

```
kubectl get svc/audit-pod
```

```
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
audit-pod   NodePort   10.111.36.142   <none>        5678:32373/TCP   72s
```

现在你可以使用 curl 命令从 kind 控制平面容器内部通过该服务暴露出来的端口来访问这个端点。

```
docker exec -it 6a96207fed4b curl localhost:32373
```

```
just made some syscalls!
```

你可以看到该进程正在运行，但是实际上执行了哪些系统调用？因为该 Pod 是在本地集群中运行的， 你应该可以在 /var/log/syslog 日志中看到这些。打开一个新的终端窗口，使用 tail 命令来 查看来自 http-echo 的调用输出：

```
tail -f /var/log/syslog | grep 'http-echo'
```

你应该已经可以看到 http-echo 发出的一些系统调用日志， 如果你在控制面板容器内 curl 了这个端点，你会看到更多的日志。

```
Jul  6 15:37:40 my-machine kernel: [369128.669452] audit: type=1326
audit(1594067860.484:14536): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=51 compat=0 ip=0x46fe1f code=0x7ffc0000
Jul  6 15:37:40 my-machine kernel: [369128.669453] audit: type=1326
audit(1594067860.484:14537): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=54 compat=0 ip=0x46fdba code=0x7ffc0000
Jul  6 15:37:40 my-machine kernel: [369128.669455] audit: type=1326
audit(1594067860.484:14538): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=202 compat=0 ip=0x455e53 code=0x7ffc0000
Jul  6 15:37:40 my-machine kernel: [369128.669456] audit: type=1326
audit(1594067860.484:14539): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=288 compat=0 ip=0x46fdba code=0x7ffc0000
Jul  6 15:37:40 my-machine kernel: [369128.669517] audit: type=1326
audit(1594067860.484:14540): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=0 compat=0 ip=0x46fd44 code=0x7ffc0000
Jul  6 15:37:40 my-machine kernel: [369128.669519] audit: type=1326
audit(1594067860.484:14541): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=270 compat=0 ip=0x4559b1 code=0x7ffc0000
Jul  6 15:38:40 my-machine kernel: [369188.671648] audit: type=1326
audit(1594067920.488:14559): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=270 compat=0 ip=0x4559b1 code=0x7ffc0000
Jul  6 15:38:40 my-machine kernel: [369188.671726] audit: type=1326
audit(1594067920.488:14560): auid=4294967295 uid=0 gid=0
ses=4294967295 pid=29064 comm="http-echo" exe="/http-echo" sig=0
arch=c000003e syscall=202 compat=0 ip=0x455e53 code=0x7ffc0000
```

通过查看每一行上的 syscall= 条目，你可以开始了解 http-echo 进程所需的系统调用。 尽管这些不太可能包含它使用的所有系统调用，但它可以作为该容器的 seccomp 配置文件的基础。

开始下一节之前，请清理该 Pod 和 Service：

```
kubectl delete pod/audit-pod
kubectl delete svc/audit-pod
```

使用导致违规的 Seccomp 配置文件创建 Pod

为了进行演示，请将不允许任何系统调用的配置文件应用于 Pod。

为你的 Kubernetes 版本下载正确的清单：

- v1.19 或更新版本（GA）
- v1.19 之前版本（alpha）

pods/security/seccomp/ga/violation-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: violation-pod
  labels:
    app: violation-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/violation.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

pods/security/seccomp/alpha/violation-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: violation-pod
  labels:
    app: violation-pod
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: localhost/profiles/
```

violation.json

```
spec:
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

在集群中创建 Pod：

```
kubectl apply -f violation-pod.yaml
```

如果你检查 Pod 的状态，你将会看到该 Pod 启动失败。

```
kubectl get pod/violation-pod
```

```
NAME            READY   STATUS             RESTARTS   AGE
violation-pod   0/1     CrashLoopBackOff   1          6s
```

如上例所示，http-echo 进程需要大量的系统调用。通过设置 "defaultAction": "SCMP_ACT_ERRNO"， 来指示 seccomp 在任何系统调用上均出错。这是非常安全的，但是会删除执行有意义的操作的能力。 你真正想要的只是给工作负载所需的特权。

开始下一节之前，请清理该 Pod 和 Service：

```
kubectl delete pod/violation-pod
kubectl delete svc/violation-pod
```

使用设置仅允许需要的系统调用的配置文件来创建 Pod

如果你看一下 fine-pod.json 文件，你会注意到在第一个示例中配置文件设置为 "defaultAction": "SCMP_ACT_LOG" 的一些系统调用。 现在，配置文件设置为 "defaultAction": "SCMP_ACT_ERRNO"，但是在 "action": "SCMP_ACT_ALLOW" 块中明确允许一组系统调用。 理想情况下，容器将成功运行，并且你将不会看到任何发送到 syslog 的消息。

为你的 Kubernetes 版本下载正确的清单：

- v1.19 或更新版本（GA）
- v1.19 之前版本（alpha）

pods/security/seccomp/ga/fine-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fine-pod
  labels:
    app: fine-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/fine-grained.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

pods/security/seccomp/alpha/fine-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fine-pod
  labels:
    app: fine-pod
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: localhost/profiles/fine-grained.json
spec:
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

在你的集群上创建Pod：

```
kubectl apply -f fine-pod.yaml
```

Pod 应该被成功启动。

```
kubectl get pod/fine-pod
```

```
NAME       READY   STATUS    RESTARTS   AGE
fine-pod   1/1     Running   0          30s
```

打开一个新的终端窗口，使用 tail 命令查看来自 http-echo 的调用的输出：

```
tail -f /var/log/syslog | grep 'http-echo'
```

使用 NodePort 服务为该 Pod 开一个端口：

```
kubectl expose pod/fine-pod --type NodePort --port 5678
```

检查服务在该节点被分配了什么端口：

```
kubectl get svc/fine-pod
```

```
NAME       TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
fine-pod   NodePort   10.111.36.142   <none>        5678:32373/TCP   72s
```

使用 curl 命令从 kind 控制面板容器内部请求这个端点：

```
docker exec -it 6a96207fed4b curl localhost:32373
```

```
just made some syscalls!
```

你会看到 syslog 中没有任何输出，因为这个配置文件允许了所有需要的系统调用，并指定如果有发生列表之外的系统调用将发生错误。从安全角度来看，这是理想的情况， 但是在分析程序时需要多付出一些努力。如果有一种简单的方法无需花费太多精力就能更接近此安全性，那就太好了。

开始下一节之前，请清理该 Pod 和 Service：

```
kubectl delete pod/fine-pod
kubectl delete svc/fine-pod
```

使用容器运行时默认的 Seccomp 配置文件创建 Pod

大多数容器运行时都提供一组允许或不允许的默认系统调用。通过使用 runtime/default 注释 或将 Pod 或容器的安全上下文中的 seccomp 类型设置为 RuntimeDefault，可以轻松地在 Kubernetes 中应用默认值。

为你的 Kubernetes 版本下载正确的清单：

- v1.19 或更新版本（GA）
- v1.19 之前版本（alpha）

pods/security/seccomp/ga/default-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

pods/security/seccomp/alpha/default-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-pod
  labels:
    app: default-pod
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
spec:
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

默认的 seccomp 配置文件应该为大多数工作负载提供足够的权限。

接下来

额外的资源：

- Seccomp 概要

- Seccomp 在 Docker 中的安全配置

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 30, 2020 at 9:23 AM PST: Translate Restrict a Container's Syscalls with Seccomp into Chinese (0dfed51bc)

AppArmor

FEATURE STATE: Kubernetes v1.4 [beta]

Apparmor 是一个 Linux 内核安全模块，它补充了标准的基于 Linux 用户和组的安全模块将程序限制为有限资源集的权限。AppArmor 可以配置为任何应用程序减少潜在的攻击面，并且提供更加深入的防御。AppArmor 是通过配置文件进行配置的，这些配置文件被调整为报名单，列出了特定程序或者容器所需要的访问权限，如 Linux 功能、网络访问、文件权限等。每个配置文件都可以在*强制*模式(阻止访问不允许的资源)或*投诉*模式(仅报告冲突)下运行。

教程目标

- 查看如何在节点上加载配置文件示例
- 了解如何在 Pod 上强制执行配置文件
- 了解如何检查配置文件是否已加载
- 查看违反配置文件时会发生什么情况
- 查看无法加载配置文件时会发生什么情况

准备开始

务必：

1. Kubernetes 版本至少是 v1.4 -- AppArmor 在 Kubernetes v1.4 版本中才添加了对 AppArmor 的支持。早于 v1.4 版本的 Kubernetes 组件不知道新的 AppArmor 注释，并且将会 **默认忽略** 提供的任何 AppArmor 设置。为了确保您的 Pods 能够得到预期的保护，必须验证节点的 Kubelet 版本：

```
kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}:
{@.status.nodeInfo.kubeletVersion}\n{end}'
```

```
gke-test-default-pool-239f5d02-gyn2: v1.4.0
gke-test-default-pool-239f5d02-x1kf: v1.4.0
gke-test-default-pool-239f5d02-xwux: v1.4.0
```

1. AppArmor 内核模块已启用 -- 要使 Linux 内核强制执行 AppArmor 配置文件，必须安装并且启动 AppArmor 内核模块。默认情况下，有几个发行版支持该模块，如 Ubuntu 和 SUSE，还有许多发行版提供可选支持。要检查模块是否已启用，请检查 /sys/module/apparmor/parameters/enabled 文件：

```
cat /sys/module/apparmor/parameters/enabled
Y
```

如果 Kubelet 包含 AppArmor 支持(>=v1.4)，如果内核模块未启用，它将拒绝运行带有 AppArmor 选项的 Pod。

> **说明：** Ubuntu 携带了许多没有合并到上游 Linux 内核中的 AppArmor 补丁，包括添加附加钩子和特性的补丁。Kubernetes 只在上游版本中测试过，不承诺支持其他特性。

1. Docker 作为容器运行环境 -- 目前，支持 Kubernetes 运行的容器中只有 Docker 也支持 AppArmor。随着更多的运行时添加 AppArmor 的支持，可选项将会增多。您可以使用以下命令验证节点是否正在运行 Docker:

```
kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}:
{@.status.nodeInfo.containerRuntimeVersion}\n{end}'
```

```
gke-test-default-pool-239f5d02-gyn2: docker://1.11.2
gke-test-default-pool-239f5d02-x1kf: docker://1.11.2
gke-test-default-pool-239f5d02-xwux: docker://1.11.2
```

如果 Kubelet 包含 AppArmor 支持(>=v1.4)，如果运行环境不是 Docker，它将拒绝运行带有 AppArmor 选项的 Pod。

1. 配置文件已加载 -- 通过指定每个容器都应使用 AppArmor 配置文件，AppArmor 应用于 Pod。如果指定的任何配置文件尚未加载到内核，Kubelet (>=v1.4) 将拒绝 Pod。通过检查 /sys/kernel/security/apparmor/profiles 文件，可以查看节点加载了哪些配置文件。例如:

```
ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/
security/apparmor/profiles | sort"
```

```
apparmor-test-deny-write (enforce)
apparmor-test-audit-write (enforce)
docker-default (enforce)
k8s-nginx (enforce)
```

有关在节点上加载配置文件的详细信息，请参见使用配置文件设置节点。

只要 Kubelet 版本包含 AppArmor 支持(>=v1.4)，如果不满足任何先决条件，Kubelet 将拒绝带有 AppArmor 选项的 Pod。您还可以通过检查节点就绪状况消息来验证节点上的 AppArmor 支持(尽管这可能会在以后的版本中删除)：

```
kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {.status.conditions[?(@.reason=="KubeletReady")].message}\n{end}'
```

```
gke-test-default-pool-239f5d02-gyn2: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-x1kf: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-xwux: kubelet is posting ready status. AppArmor enabled
```

保护 Pod

> **说明：**
>
> AppArmor 目前处于测试阶段，因此选项被指定为注释。一旦 AppArmor 被授予支持通用，注释将替换为首要的字段(更多详情参见 升级到 GA 的途径)。

AppArmor 配置文件被指定为 *per-container*。要指定要用其运行 Pod 容器的 AppArmor 配置文件，请向 Pod 的元数据添加注释：

```
container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_ref>
```

`<container_name>` 的名称是容器的简称，用以描述简介，并且简称为 `<profile_ref>` 。`<profile_ref>` 可以作为其中之一：

- runtime/default 应用运行时的默认配置
- localhost/<profile_name> 应用在名为 <profile_name> 的主机上加载的配置文件
- unconfined 表示不加载配置文件

有关注释和配置文件名称格式的详细信息，请参阅API 参考。

Kubernetes AppArmor 强制执行方式首先通过检查所有先决条件都已满足，然后将配置文件选择转发到容器运行时进行强制执行。如果未满足先决条件， Pod 将被拒绝，并且不会运行。

要验证是否应用了配置文件，可以查找容器创建事件中列出的 AppArmor 安全选项：

```
kubectl get events | grep Created
```

```
22s 22s 1 hello-apparmor Pod spec.containers{hello} Normal Created {kubelet e2e-test-stclair-node-pool-31nt} Created
```

```
container with docker id 269a53b202d3; Security:[seccomp=unconfined apparmor=k8s-apparmor-example-deny-write]
```

您还可以通过检查容器的 proc attr，直接验证容器的根进程是否以正确的配置文件运行：

```
kubectl exec <pod_name> cat /proc/1/attr/current
```

```
k8s-apparmor-example-deny-write (enforce)
```

举例

本例假设您已经使用 AppArmor 支持设置了一个集群。

首先，我们需要将要使用的配置文件加载到节点上。我们将使用的配置文件仅拒绝所有文件写入：

```
#include <tunables/global>
profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>
  file,
  # Deny all file writes.
  deny /** w,
}
```

由于我们不知道 Pod 将被安排在那里，我们需要在所有节点上加载配置文件。在本例中，我们将只使用 SSH 来安装概要文件，但是在使用配置文件设置节点中讨论了其他方法。

```
NODES=(
    # The SSH-accessible domain names of your nodes
    gke-test-default-pool-239f5d02-gyn2.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-x1kf.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-xwux.us-central1-a.my-k8s)
for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_parser -q <<EOF
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>

  file,

  # Deny all file writes.
  deny /** w,
}
EOF'
done
```

接下来，我们将运行一个带有拒绝写入配置文件的简单 "Hello AppArmor" pod：

pods/security/hello-apparmor.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    # Tell Kubernetes to apply the AppArmor profile "k8s-apparmor-example-deny-write".
    # Note that this is ignored if the Kubernetes node is not running version 1.4 or greater.
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-deny-write
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

```
kubectl create -f ./hello-apparmor.yaml
```

如果我们查看 pod 事件，我们可以看到 pod 容器是用 AppArmor 配置文件 "k8s-apparmor-example-deny-write" 所创建的：

```
kubectl get events | grep hello-apparmor
```

```
14s       14s       1         hello-apparmor   Pod                                 Normal    Scheduled   {default-scheduler }                                 Successfully assigned hello-apparmor to gke-test-default-pool-239f5d02-gyn2
14s       14s       1         hello-apparmor   Pod       spec.containers{hello}    Normal    Pulling     {kubelet gke-test-default-pool-239f5d02-gyn2}   pulling image "busybox"
13s       13s       1         hello-apparmor   Pod       spec.containers{hello}    Normal    Pulled      {kubelet gke-test-default-pool-239f5d02-gyn2}   Successfully pulled image "busybox"
13s       13s       1         hello-apparmor   Pod       spec.containers{hello}    Normal    Created     {kubelet gke-test-default-pool-239f5d02-gyn2}   Created container with docker id 06b6cd1c0989; Security: [seccomp=unconfined apparmor=k8s-apparmor-example-deny-write]
13s       13s       1         hello-apparmor   Pod       spec.containers{hello}    Normal    Started     {kubelet gke-test-default-pool-239f5d02-gyn2}   Started container with docker id 06b6cd1c0989
```

我们可以通过检查该配置文件的 proc attr 来验证容器是否实际使用该配置文件运行：

```
kubectl exec hello-apparmor cat /proc/1/attr/current
```

```
k8s-apparmor-example-deny-write (enforce)
```

最后，我们可以看到如果试图通过写入文件来违反配置文件，会发生什么情况：

```
kubectl exec hello-apparmor touch /tmp/test
```

```
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero exit code: Error executing in Docker Container: 1
```

最后，让我们看看如果我们试图指定一个尚未加载的配置文件会发生什么：

```
kubectl create -f /dev/stdin <<EOF
```

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor-2
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-allow-write
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
EOF
pod/hello-apparmor-2 created
```

```
kubectl describe pod hello-apparmor-2
```

```
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kubernetes.io/hello=localhost/k8s-apparmor-example-allow-write
Status:        Pending
Reason:        AppArmor
Message:       Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" is not loaded
IP:
```

```
Controllers:  <none>
Containers:
  hello:
    Container ID:
    Image:    busybox
    Image ID:
    Port:
    Command:
      sh
      -c
      echo 'Hello AppArmor!' && sleep 1h
    State:          Waiting
      Reason:       Blocked
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-
dnz7v (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-dnz7v:
    Type:   Secret (a volume populated by a Secret)
    SecretName:   default-token-dnz7v
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen    LastSeen    Count    From                    SubobjectPath
Type        Reason        Message
  ---------    --------    -----    ----                    -------------    --------
------       -------
  23s          23s         1        {default-scheduler }                    Normal
Scheduled     Successfully assigned hello-apparmor-2 to e2e-test-stclair-
node-pool-t1f5
  23s          23s         1        {kubelet e2e-test-stclair-node-pool-t1f5}
Warning       AppArmor    Cannot enforce AppArmor: profile "k8s-
apparmor-example-allow-write" is not loaded
```

注意 pod 呈现失败状态，并且显示一条有用的错误信息：Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" 未加载。还用相同的消息记录了一个事件。

管理

使用配置文件设置节点

Kubernetes 目前不提供任何本地机制来将 AppArmor 配置文件加载到节点上。有很多方法可以设置配置文件，例如：

- 通过在每个节点上运行 Pod 的DaemonSet确保加载了正确的配置文件。可以找到一个示例实现这里。
- 在节点初始化时，使用节点初始化脚本(例如 Salt 、Ansible 等)或镜像。
- 通过将配置文件复制到每个节点并通过 SSH 加载它们，如示例。

调度程序不知道哪些配置文件加载到哪个节点上，因此必须将全套配置文件加载到每个节点上。另一种方法是为节点上的每个配置文件(或配置文件类)添加节点标签，并使用[节点选择器](/zh/docs/concepts/configuration/assign pod node/)确保 Pod 在具有所需配置文件的节点上运行。

使用 PodSecurityPolicy 限制配置文件

如果启用了 PodSecurityPolicy 扩展，则可以应用群集范围的 AppArmor 限制。要启用 PodSecurityPolicy，必须在"apiserver"上设置以下标志：

```
--enable-admission-plugins=PodSecurityPolicy[,others...]
```

AppArmor 选项可以指定为 PodSecurityPolicy 上的注释：

```
apparmor.security.beta.kubernetes.io/defaultProfileName: <profile_ref>
apparmor.security.beta.kubernetes.io/allowedProfileNames: <profile_ref>[,others...]
```

默认配置文件名选项指定默认情况下在未指定任何配置文件时应用于容器的配置文件。节点允许配置文件名选项指定允许 Pod 容器运行时的配置文件列表。配置文件的指定格式与容器上的相同。完整规范见API 参考。

禁用 AppArmor

如果您不希望 AppArmor 在集群上可用，可以通过命令行标志禁用它：

```
--feature-gates=AppArmor=false
```

禁用时，任何包含 AppArmor 配置文件的 Pod 都将因 "Forbidden" 错误而导致验证失败。注意，默认情况下，docker 总是在非特权 pods 上启用 "docker-default" 配置文件(如果 AppArmor 内核模块已启用)，并且即使功能门已禁用，也将继续启用该配置文件。当 AppArmor 应用于通用(GA)时，禁用 Apparmor 的选项将被删除。

使用 AppArmor 升级到 Kubernetes v1.4

不需要对 AppArmor 执行任何操作即可将集群升级到 v1.4。但是，如果任何现有的 pods 有一个 AppArmor 注释，它们将不会通过验证(或 PodSecurityPolicy 认证)。如果节点上加载了许可配置文件，恶意用户可以预先应用许可配置文件，将 pod 权限提升到 docker-default 权限之上。如果存在这个问题，建议清除包含 apparmor.security.beta.kubernetes.io 注释的任何 pods 的集群。

升级到一般可用性的途径

当 Apparmor 准备升级到通用(GA)时，当前指定的选项通过注释将转换为字段。通过转换支持所有升级和降级路径是非常微妙的，并将在转换发生时详细解释。我们将承诺在至少两个版本中同时支持字段和注释，并在之后的至少两个版本中显式拒绝注释。

编写配置文件

获得正确指定的 AppArmor 配置文件可能是一件棘手的事情。幸运的是，有一些工具可以帮助您做到这一点：

- aa-genprof and aa-logprof 通过监视应用程序的活动和日志并承认它所采取的操作来生成配置文件规则。更多说明由AppArmor 文档提供。
- bane是一个用于 Docker的 AppArmor 档案生成器，它使用简化的档案语言。

建议在开发工作站上通过 Docker 运行应用程序以生成配置文件，但是没有什么可以阻止在运行 Pod 的 Kubernetes 节点上运行工具。

想要调试 AppArmor 的问题，您可以检查系统日志，查看具体拒绝了什么。AppArmor 将详细消息记录到 dmesg ，错误通常可以在系统日志中或通过 journalctl 找到。更多详细信息见AppArmor 失败。

API 参考

Pod 注释

指定容器将使用的配置文件：

- **key**: container.apparmor.security.beta.kubernetes.io/<container_name> 中的 <container_name> 匹配 Pod 中的容器名称。 可以为 Pod 中的每个容器指定单独的配置文件。
- **value**: 配置文件参考，如下所述

配置文件参考

- runtime/default: 指默认运行时配置文件。
  - 等同于不指定配置文件(没有 PodSecurityPolicy 默认值)，除非它仍然需要启用 AppArmor。

        - 对于 Docker，这将解析为非特权容器的Docker default配置文件，特权容器的配置文件为未定义(无配置文件)。
    - localhost/<profile_name>: 指按名称加载到节点(localhost)上的配置文件。
        - 可能的配置文件名在 核心策略参考。
    - unconfined: 这有效地禁用了容器上的 AppArmor 。

任何其他配置文件引用格式无效。

PodSecurityPolicy 注解

指定在未提供容器时应用于容器的默认配置文件：

- **key**: apparmor.security.beta.kubernetes.io/defaultProfileName
- **value**: 如上述文件参考所述

上面描述的指定配置文件， Pod 容器列表的配置文件引用允许指定：

- **key**: apparmor.security.beta.kubernetes.io/allowedProfileNames
- **value**: 配置文件引用的逗号分隔列表(如上所述)
    - 尽管转义逗号是配置文件名中的合法字符，但此处不能显式允许。

接下来

其他资源

- Apparmor 配置文件语言快速指南
- Apparmor 核心策略参考

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 June 01, 2020 at 9:23 AM PST: add zh pages (4b35d4d40)

Services

使用 Source IP

使用 Source IP

Kubernetes 集群中运行的应用通过 Service 抽象来互相查找、通信和与外部世界沟通。本文介绍被发送到不同类型 Services 的数据包源 IP 的变化过程，你可以根据你的需求改变这些行为。

准备开始

你必须拥有一个 Kubernetes 的集群，同时你的 Kubernetes 集群必须带有 kubectl 命令行工具。 如果你还没有集群，你可以通过 Minikube 构建一 个你自己的集群，或者你可以使用下面任意一个 Kubernetes 工具构建：

- Katacoda
- 玩转 Kubernetes

要获知版本信息，请输入 kubectl version.

术语表

本文使用了下列术语：

- NAT: 网络地址转换
- Source NAT: 替换数据包的源 IP, 通常为节点的 IP
- Destination NAT: 替换数据包的目的 IP, 通常为 Pod 的 IP
- VIP: 一个虚拟 IP, 例如分配给每个 Kubernetes Service 的 IP
- Kube-proxy: 一个网络守护程序，在每个节点上协调 Service VIP 管理

准备工作

你必须拥有一个正常工作的 Kubernetes 1.5 集群来运行此文档中的示例。该示例使用一个简单的 nginx webserver，通过一个HTTP消息头返回它接收到请求的源IP。你可以像下面这样创建它：

```
kubectl run source-ip-app --image=k8s.gcr.io/echoserver:1.4
```

输出结果为

```
deployment.apps/source-ip-app created
```

教程目标

- 通过多种类型的 Services 暴露一个简单应用
- 理解每种 Service 类型如何处理源 IP NAT
- 理解保留源IP所涉及的折中

Type=ClusterIP 类型 Services 的 Source IP

如果你的 kube-proxy 运行在 iptables 模式下，从集群内部发送到 ClusterIP 的包永远不会进行源地址 NAT，这从 Kubernetes 1.2 开始是默认选项。Kube-proxy 通过一个 proxyMode endpoint 暴露它的模式。

```
kubectl get nodes
```

输出结果与以下结果类似:

```
NAME                     STATUS    ROLES    AGE     VERSION
kubernetes-node-6jst   Ready     <none>   2h      v1.13.0
kubernetes-node-cx31   Ready     <none>   2h      v1.13.0
kubernetes-node-jj1t   Ready     <none>   2h      v1.13.0
```

从其中一个节点中得到代理模式

```
kubernetes-node-6jst $ curl localhost:10249/proxyMode
```

输出结果为：

```
iptables
```

你可以通过在source IP应用上创建一个Service来测试源IP保留。

```
kubectl expose deployment source-ip-app --name=clusterip --port=80 --target-port=8080
```

输出结果为：

```
service/clusterip exposed
```

```
kubectl get svc clusterip
```

输出结果与以下结果类似：

```
NAME        TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
clusterip   ClusterIP   10.0.170.92   <none>        80/TCP    51s
```

从相同集群中的一个 pod 访问这个 ClusterIP：

```
kubectl run busybox -it --image=busybox --restart=Never --rm
```

输出结果与以下结果类似：

```
Waiting for pod default/busybox to be running, status is Pending, pod ready: false
If you don't see a command prompt, try pressing enter.

# ip addr
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.8/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::188a:84ff:feb0:26a5/64 scope link
       valid_lft forever preferred_lft forever

# wget -qO - 10.0.170.92
CLIENT VALUES:
client_address=10.244.3.8
command=GET
...
```

无论客户端 pod 和 服务端 pod 是否在相同的节点上，client_address 始终是客户端 pod 的 IP 地址。

Type=NodePort 类型 Services 的 Source IP

从 Kubernetes 1.5 开始，发送给类型为 Type=NodePort Services 的数据包默认进行源地址 NAT。你可以通过创建一个 NodePort Service 来进行测试：

```
kubectl expose deployment source-ip-app --name=nodeport --port=80 --target-port=8080 --type=NodePort
```

输出结果为：

```
service/nodeport exposed
```

```
NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].nodePort}" services nodeport)
NODES=$(kubectl get nodes -o jsonpath='{ $.items[*].status.addresses[?(@.type=="InternalIP")].address }')
```

如果你的集群运行在一个云服务上，你可能需要为上面报告的 nodes:nodeport 开启一条防火墙规则。 现在，你可以通过上面分配的节点端口从外部访问这个 Service。

```
for node in $NODES; do curl -s $node:$NODEPORT | grep -i client_address; done
```

输出结果与以下结果类似：

```
client_address=10.180.1.1
client_address=10.240.0.5
client_address=10.240.0.3
```

请注意，这些并不是正确的客户端 IP，它们是集群的内部 IP。这是所发生的事情：

- 客户端发送数据包到 node2:nodePort
- node2 使用它自己的 IP 地址替换数据包的源 IP 地址（SNAT）
- node2 使用 pod IP 地址替换数据包的目的 IP 地址
- 数据包被路由到 node 1，然后交给 endpoint
- Pod 的回复被路由回 node2
- Pod 的回复被发送回给客户端

用图表示：

```
       client
         \ ^
          \ \
           v \
  node 1 <--- node 2
   | ^   SNAT
   | |   --->
   v |
endpoint
```

为了防止这种情况发生，Kubernetes 提供了一个特性来保留客户端的源 IP 地址(点击此处查看可用特性)。设置 service.spec.externalTrafficPolicy 的值为 Local，请求就只会被代理到本地 endpoints 而不会被转发到其它节点。这样就保留了最初的源 IP 地址。如果没有本地 endpoints，发送到这个节点的数据包将会被丢弃。这样在应用到数据包的任何包处理规则下，你都能依赖这个正确的 source-ip 使数据包通过并到达 endpoint。

设置 service.spec.externalTrafficPolicy 字段如下：

```
kubectl patch svc nodeport -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

输出结果为：

```
service/nodeport patched
```

现在，重新运行测试：

```
for node in $NODES; do curl --connect-timeout 1 -s $node:$NODEPORT | grep -i client_address; done
```

输出结果为：

```
client_address=104.132.1.79
```

请注意，你只从 endpoint pod 运行的那个节点得到了一个回复，这个回复有*正确的*客户端 IP。

这是发生的事情：

- 客户端发送数据包到 node2:nodePort，它没有任何 endpoints
- 数据包被丢弃
- 客户端发送数据包到 node1:nodePort，它*有*endpoints
- node1 使用正确的源 IP 地址将数据包路由到 endpoint

用图表示：

```
      client
     ^ /   \
    / /     \
   / v       X
  node 1     node 2
   ^ |
   | |
   | v
endpoint
```

Type=LoadBalancer 类型 Services 的 Source IP

从Kubernetes1.5开始，发送给类型为 Type=LoadBalancer Services 的数据包默认进行源地址 NAT，这是因为所有处于 Ready 状态的可调度 Kubernetes 节点对于负载均衡的流量都是符合条件的。所以如果数据包到达一个没有 endpoint 的节点，系统将把这个包代理到*有* endpoint 的节点，并替换数据包的源 IP 为节点的 IP（如前面章节所述）。

你可以通过在一个 loadbalancer 上暴露这个 source-ip-app 来进行测试。

```
kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-port=8080 --type=LoadBalancer
```

输出结果为：

```
service/loadbalancer exposed
```

打印Service的IPs：

```
kubectl get svc loadbalancer
```

输出结果与以下结果类似：

```
NAME           TYPE           CLUSTER-IP    EXTERNAL-IP       PORT(S)   AGE
loadbalancer   LoadBalancer   10.0.65.118   104.198.149.140   80/TCP    5m
```

```
curl 104.198.149.140
```

输出结果与以下结果类似：

```
CLIENT VALUES:
client_address=10.240.0.5
...
```

然而，如果你的集群运行在 Google Kubernetes Engine/GCE 上，可以通过设置 service.spec.externalTrafficPolicy 字段值为 Local ，故意导致健康检查失败来强制使没有 endpoints 的节点把自己从负载均衡流量的可选节点列表中删除。

用图表示：

```
                client
                  |
                lb VIP
               / ^
              v /
health check --->   node 1   node 2 <--- health check
      200  <---    ^ |           ---> 500
               | V
              endpoint
```

你可以设置 annotation 来进行测试：

```
kubectl patch svc loadbalancer -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

你应该能够立即看到 Kubernetes 分配的 service.spec.healthCheckNodePort 字段：

```
kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort
```

输出结果与以下结果类似：

```
  healthCheckNodePort: 32122
```

service.spec.healthCheckNodePort 字段指向每个节点在 /healthz 路径上提供的用于健康检查的端口。你可以这样测试：

```
kubectl get pod -o wide -l run=source-ip-app
```

输出结果与以下结果类似：

```
NAME                          READY     STATUS    RESTARTS   AGE       IP
NODE
source-ip-app-826191075-qehz4   1/1       Running   0          20h
10.180.1.136   kubernetes-node-6jst
```

使用 curl 命令发送请求到每个节点的 /healthz 路径。

```
kubernetes-node-6jst $ curl localhost:32122/healthz
```

输出结果与以下结果类似：

```
1 Service Endpoints found
```

```
kubernetes-node-jj1t $ curl localhost:32122/healthz
```

输出结果与以下结果类似：

```
No Service Endpoints Found
```

主节点运行的 service 控制器负责分配 cloud loadbalancer。在这样做的同时，它也会分配指向每个节点的 HTTP 健康检查的 port/path。等待大约 10 秒钟之后，没有 endpoints 的两个节点的健康检查会失败，然后 curl 负载均衡器的 ip：

```
curl 104.198.149.140
```

输出结果与以下结果类似：

```
CLIENT VALUES:
client_address=104.132.1.79
...
```

跨平台支持

从 Kubernetes 1.5 开始，通过类型为 Type=LoadBalancer 的 Services 进行源 IP 保存的支持仅在一部分 cloudproviders 中实现（GCP and Azure）。你的集群运行的 cloudprovider 可能以某些不同的方式满足 loadbalancer 的要求：

1. 使用一个代理终止客户端连接并打开一个到你的 nodes/endpoints 的新连接。在这种情况下，源 IP 地址将永远是云负载均衡器的地址而不是客户端的。

2. 使用一个包转发器，因此从客户端发送到负载均衡器 VIP 的请求在拥有客户端源 IP 地址的节点终止，而不被中间代理。

第一类负载均衡器必须使用一种它和后端之间约定的协议来和真实的客户端 IP 通信，例如 HTTP X-FORWARDED-FOR 头，或者 proxy 协议。 第二类负载均衡器可以通过简单的在保存于 Service 的 service.spec.healthCheckNodePort 字段上创建一个 HTTP 健康检查点来使用上面描述的特性。

清理现场

删除服务：

```
$ kubectl delete svc -l run=source-ip-app
```

删除 Deployment、ReplicaSet 和 Pod：

```
$ kubectl delete deployment source-ip-app
```

接下来

- 学习更多关于 通过 services 连接应用
- 学习更多关于 负载均衡

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 25, 2020 at 6:49 PM PST: fix svc type error (75b8566c9)