

任务

Kubernetes 文档这一部分包含的一些页面展示如何去完成单个任务。 每个任务页面是一般通过给出若干步骤展示如何执行完成某事。

如果你希望编写一个任务页面，参考 [创建一个文档拉取请求](#)。

[安装工具](#)

在你的计算机上设置 Kubernetes 工具。

[管理集群](#)

了解管理集群的常见任务。

[配置 Pods 和容器](#)

对 Pod 和容器执行常见的配置任务。

[管理 Kubernetes 对象](#)

用声明式和命令式范型与 Kubernetes API 交互。

[管理 Secrets](#)

使用 Secrets 管理机密配置数据。

[给应用注入数据](#)

给你的工作负载 Pod 指定配置和其他数据。

[运行应用](#)

运行和管理无状态和有状态的应用程序。

[运行 Jobs](#)

使用并行处理运行 Jobs。

[访问集群中的应用程序](#)

配置负载平衡、端口转发或设置防火墙或 DNS 配置，以访问集群中的应用程序。

[监控、日志和排错](#)

设置监视和日志记录以对集群进行故障排除或调试容器化应用。

[扩展 Kubernetes](#)

了解针对工作环境需要来调整 Kubernetes 集群的进阶方法。

[TLS](#)

了解如何使用传输层安全性（ TLS ）保护集群中的流量。

[管理集群守护进程](#)

执行 DaemonSet 管理的常见任务，例如执行滚动更新。

[安装服务目录](#)

安装服务目录扩展 API。

[网络](#)

了解如何为你的集群配置网络。

[用插件扩展 kubectl](#)

通过创建和安装 kubectl 插件扩展 kubectl。

[管理巨页 \(HugePages \)](#)

将大页配置和管理为集群中的可调度资源。

[调度 GPUs](#)

配置和调度 GPU 成一类资源以供集群中节点使用。

安装工具

在你的计算机上设置 Kubernetes 工具。

kubectl

Kubernetes 命令行工具，kubectl，使得你可以对 Kubernetes 集群运行命令。 你可以使用 kubectl 来部署应用、监测和管理集群资源以及查看日志。

关于如何下载和安装 kubectl 并配置其访问你的集群，可参阅 [安装和配置 kubectl](#)。

[查看 kubectl 安装和配置指南](#)

你也可以阅读 [kubectl 参考文档](#).

kind

[kind](#) 让你能够在本地计算机上运行 Kubernetes。 kind 要求你安装并配置好 [Docker](#)。

kind [快速入门](#) 页面展示了开始使用 kind 所需要完成的操作。

[查看 kind 的快速入门指南](#)

minikube

与 kind 类似，[minikube](#) 是一个工具，能让你在本地运行 Kubernetes。 minikube 在你本地的个人计算机（包括 Windows、macOS 和 Linux PC）运行一个单节点的 Kubernetes 集群，以便你来尝试 Kubernetes 或者开展每天的开发工作。

如果你关注如何安装此工具，可以按官方的 [Get Started!](#) 指南操作。

[查看 minikube 快速入门指南](#)

当你拥有了可工作的 minikube 时，就可以用它来 [运行示例应用](#) 了。

kubeadm

你可以使用 [kubeadm](#) 工具来 创建和管理 Kubernetes 集群。该工具能够执行必要的动作并用一种用户友好的方式启动一个可用的、安全的集群。

[安装 kubeadm](#) 展示了如何安装 kubeadm 的过程。一旦安装了 kubeadm，你就可以使用它来 [创建一个集群](#)。

[查看 kubeadm 安装指南](#)

安装并配置 kubectl

使用 Kubernetes 命令行工具 [kubectl](#)，你可以在 Kubernetes 上运行命令。使用 kubectl，你可以部署应用、检查和管理集群资源、查看日志。要了解 kubectl 操作的完整列表，请参阅 [kubectl 概览](#)。

准备开始

你必须使用与集群小版本号差别为一的 kubectl 版本。例如，1.2 版本的客户端应该与 1.1 版本、1.2 版本和 1.3 版本的主节点一起使用。使用最新版本的 kubectl 有助于避免无法预料的问题。

在 Linux 上安装 kubectl

在 Linux 上使用 curl 安装 kubectl 可执行文件

1. 使用下面命令下载最新的发行版本：

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"
```

要下载特定版本，`$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)` 部分替换为指定版本。

例如，要下载 Linux 上的版本 v1.20.0，输入：

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.20.0/bin/linux/amd64/kubectl
```

1. 标记 kubectl 文件为可执行：

```
chmod +x ./kubectl
```

1. 将文件放到 PATH 路径下：

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

1. 测试你所安装的版本是最新的：

```
kubectl version --client
```

使用原生包管理器安装

- [Ubuntu、Debian 或 HypriotOS](#)
- [CentOS、RHEL 或 Fedora](#)

```
sudo apt-get update && sudo apt-get install -y apt-transport-https gnupg2 curl  
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/  
apt/sources.list.d/kubernetes.list  
sudo apt-get update  
sudo apt-get install -y kubectl
```

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo  
[kubernetes]  
name=Kubernetes  
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
```

```
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://
packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
yum install -y kubectl
```

使用其他包管理器安装

- [Snap](#)
- [Homebrew](#)

如果你使用 Ubuntu 或者其他支持 [snap](#) 包管理器的 Linux 发行版，kubectl 可以作为 [Snap](#) 应用来安装：

```
sudo snap install kubectl --classic
```

```
kubectl version --client
```

如果你在使用 Linux 且使用 [Homebrew](#) 包管理器，kubectl 也可以用这种方式[安装](#)。

```
brew install kubectl
```

```
kubectl version --client
```

在 macOS 上安装 kubectl

在 macOS 上使用 curl 安装 kubectl 可执行文件

1. 下载最新发行版本：

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl"
```

要下载特定版本，可将上面命令中的\$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt) 部分替换成你想要的版本。

例如，要在 macOS 上安装版本 v1.20.0，输入：

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.20.0/bin/darwin/amd64/kubectl
```

将二进制文件标记为可执行：

```
chmod +x ./kubectl
```

1. 将二进制文件放入 PATH 目录下：

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

1. 测试以确保所安装的版本是最新的：

```
kubectl version --client
```

在 macOS 上使用 Homebrew 安装

如果你使用的是 macOS 系统且使用 [Homebrew](#) 包管理器，你可以使用 Homebrew 来安装 kubectl。

1. 运行安装命令：

```
brew install kubectl
```

或者

```
brew install kubernetes-cli
```

1. 测试以确保你安装的版本是最新的：

```
kubectl version --client
```

在 macOS 上用 Macports 安装 kubectl

如果你使用的是 macOS 系统并使用 [Macports](#) 包管理器，你可以通过 Macports 安装 kubectl。

1. 运行安装命令：

```
sudo port selfupdate  
sudo port install kubectl
```

1. 测试以确保你安装的版本是最新的：

```
kubectl version --client
```

在 Windows 上安装 kubectl

在 Windows 上使用 curl 安装 kubectl 二进制文件

1. 从此[链接](#) 下载最新发行版本 v1.20.0。

或者如何你安装了 curl，使用下面的命令：

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.20.0/  
bin/windows/amd64/kubectl.exe
```

要了解最新的稳定版本（例如，出于脚本编写目的），可查看 <https://storage.googleapis.com/kubernetes-release/release/stable.txt>。

1. 将可执行文件放到 PATH 目录下。
1. 测试以确定所下载的 kubectl 版本是正确的：

```
kubectl version --client
```

说明： Docker Desktop for Windows 会在 PATH 中添加自己的 kubectl 程序。如果你之前安装过 Docker Desktop，你可能需要将新安装的 PATH 项放到 Docker Desktop 安装程序所添加的目录之前，或者干脆删除 Docker Desktop 所安装的 kubectl。

使用 PowerShell 从 PSGallery 安装 kubectl

如果你使用的是 Windows 系统并使用 [Powershell Gallery](#) 软件包管理器，你可以使用 Powershell 安装和更新 kubectl。

1. 运行安装命令（确保指定 DownloadLocation）：

```
Install-Script -Name 'install-kubectl' -Scope CurrentUser -Force  
install-kubectl.ps1 [-DownloadLocation <path>]
```

说明： 如果你没有指定 DownloadLocation，那么 kubectl 将安装在用户的临时目录中。

安装程序创建 \$HOME/.kube 目录，并指示它创建配置文件

1. 测试以确保你安装的版本是最新的：

```
kubectl version --client
```

说明： 通过重新运行步骤 1 中列出的两个命令可以更新安装。

在 Windows 系统上用 Chocolatey 或者 Scoop 安装

1. 要在 Windows 上用 [Chocolatey](#) 或者 [Scoop](#) 命令行安装程序安装 kubectl：

- [choco](#)
- [scoop](#)

```
choco install kubernetes-cli
```

```
scoop install kubectl
```

1. 测试以确保你安装的版本是最新的：

```
kubectl version --client
```

1. 切换到你的 HOME 目录：

```
# 如果你在使用 cmd.exe , 运行 cd %USERPROFILE%
cd ~
```

1. 创建 .kube 目录 :

```
mkdir .kube
```

1. 进入到刚刚创建的 .kube 目录 :

```
cd .kube
```

1. 配置 kubectl 以使用远程 Kubernetes 集群 :

```
New-Item config -type file
```

说明 : 使用你喜欢的文本编辑器 , 例如 Notepad , 编辑此配置文件。

将 kubectl 作为 Google Cloud SDK 的一部分下载

kubectl 可以作为 Google Cloud SDK 的一部分进行安装。

1. 安装 [Google Cloud SDK](#)。

2. 运行以下命令安装 kubectl :

```
gcloud components install kubectl
```

1. 测试以确保你安装的版本是最新的 :

```
kubectl version --client
```

验证 kubectl 配置

kubectl 需要一个 [kubeconfig 配置文件](#) 使其找到并访问 Kubernetes 集群。当你使用 [kube-up.sh](#) 创建 Kubernetes 集群或者使用已经部署好的 Minikube 集群时 , 会自动生成 kubeconfig 配置文件。默认情况下 , kubectl 的配置文件位于 ~/.kube/config。

通过获取集群状态检查 kubectl 是否被正确配置 :

```
kubectl cluster-info
```

如果你看到一个 URL 被返回 , 那么 kubectl 已经被正确配置 , 能够正常访问你的 Kubernetes 集群。

如果你看到类似以下的信息被返回 , 那么 kubectl 没有被正确配置 , 无法正常访问你的 Kubernetes 集群。

```
The connection to the server <server-name:port> was refused - did you specify
the right host or port?
```

例如，如果你打算在笔记本电脑（本地）上运行 Kubernetes 集群，则需要首先安装 minikube 等工具，然后重新运行上述命令。

如果 kubectl cluster-info 能够返回 URL 响应，但你无法访问你的集群，可以使用下面的命令检查配置是否正确：

```
kubectl cluster-info dump
```

可选的 kubectl 配置

启用 shell 自动补全功能

kubectl 为 Bash 和 Zsh 支持自动补全功能，可以节省大量输入！

下面是设置 Bash 与 Zsh 下自动补齐的过程（包括 Linux 与 macOS 的差异）。

- [Linux 上的 Bash](#)
- [macOS 上的 Bash](#)
- [Zsh](#)

介绍

用于 Bash 的 kubectl 自动补齐脚本可以用 kubectl completion bash 命令生成。在 Shell 环境中引用自动补齐脚本就可以启用 kubectl 自动补齐。

不过，补齐脚本依赖于 [bash-completion](#) 软件包，这意味着你必须先安装 bash-completion（你可以通过运行 type _init_completion 来测试是否你已经安装了这个软件）。

安装 bash-completion

很多包管理器都提供 bash-completion（参见[这里](#)）。你可以通过 apt-get install bash-completion 或 yum install bash-completion 来安装。

上述命令会创建 /usr/share/bash-completion/bash_completion，也就是 bash-completion 的主脚本。取决于所用的包管理器，你可能必须在你的 ~/.bashrc 中通过 source 源引此文件。

要搞清楚这一点，可以重新加载你的 Shell 并运行 type _init_completion。如果命令成功，一切就绪；否则你就需要将下面的内容添加到你的 ~/.bashrc 文件中：

```
source /usr/share/bash-completion/bash_completion
```

之后，重新加载你的 Shell 并运行 type _init_completion 来检查 bash-completion 是否已正确安装。

启用 kubectl 自动补齐

你现在需要确定在你的所有 Shell 会话中都源引了 kubectl 自动补齐脚本。 实现这点有两种方式：

- 在 `~/.bashrc` 文件中源引自动补齐脚本

```
echo 'source <(kubectl completion bash)' >> ~/.bashrc
```

- 将自动补齐脚本添加到目录 `/etc/bash_completion.d` :

```
kubectl completion bash >/etc/bash_completion.d/kubectl
```

如果你为 `kubectl` 命令设置了别名 (alias) , 你可以扩展 Shell 补齐 , 使之能够与别名一起使用 :

```
echo 'alias k=kubectl' >> ~/.bashrc
echo 'complete -F __start_kubectl k' >> ~/.bashrc
```

说明 : `bash-completion` 会自动源引 `/etc/bash_completion.d` 下的所有自动补齐脚本。

两种方法是等价的。重新加载 Shell 之后 , `kubectl` 的自动补齐应该能够使用了。

介绍

用于 Bash 的 `kubectl` 自动补齐脚本可以用 `kubectl completion bash` 命令生成。在 Shell 环境中引用自动补齐脚本就可以启用 `kubectl` 自动补齐。不过 , 补齐脚本依赖于 [bash-completion](#) 软件包 , 你必须预先安装。

警告 : `bash-completion` 有两个版本 , v1 和 v2。v1 是用于 Bash 3.2 版本的 (macOS 上的默认配置) , v2 是用于 Bash 4.1 以上版本的。 `kubectl` 补齐脚本 无法 在 v1 版本的 `bash-completion` 和 Bash 3.2 上使用 , 需要 **bash-completion v2** 和 **Bash 4.1 以上版本**。因此 , 为了在 macOS 上正常使用 `kubectl` 自动补齐 , 你需要安装并使用 Bash 4.1+ 版本 ([相关指南](#))。下面的指令假定你在使用 Bash 4.1+ (也就是说 Bash 4.1 及以上版本) 。

升级 Bash

这里的命令假定你使用的是 Bash 4.1+。你可以通过下面的命令来检查 Bash 版本 :

```
echo $BASH_VERSION
```

如果版本很老 , 你可以使用 Homebrew 来安装或升级 :

```
brew install bash
```

重新加载 Shell 并验证你使用的版本是期望的版本 :

```
echo $BASH_VERSION $SHELL
```

Homebrew 通常安装 Bash 到 /usr/local/bin/bash。

安装 bash-completion

说明：如前所述，这里的指令假定你使用的是 Bash 4.1+，这意味着你会安装 bash-completion 的 v2 版本（与此相对，在 Bash 3.2 版本中的 bash-completion v1 是 kubectl 无法使用的）。

你可以通过输入 type _init_completion 来测试是否 bash-completion v2 已经安装。如果没有，可以用 Homebrew 来安装：

```
brew install bash-completion@2
```

就像命令的输出所提示的，你应该将下面的内容添加到 ~/.bash_profile 文件中：

```
export BASH_COMPLETION_COMPAT_DIR="/usr/local/etc/bash_completion.d"
[[ -r "/usr/local/etc/profile.d/bash_completion.sh" ]] && . "/usr/local/etc/profile.d/
bash_completion.sh"
```

重新加载你的 Shell 并运行 type _init_completion，验证 bash-completion v2 被正确安装。

启用 kubectl 自动补齐

你现在需要确保在你的所有 Shell 会话中都源引了 kubectl 自动补齐脚本。实现这点有两种方式：

- 在 ~/.bash_profile 文件中源引自动补齐脚本

```
echo 'source <(kubectl completion bash)' >> ~/.bash_profile
```

- 将自动补齐脚本添加到目录 /usr/local/etc/bash_completion.d：

```
kubectl completion bash >/usr/local/etc/bash_completion.d/kubectl
```

- 如果你为 kubectl 命令设置了别名（alias），你可以扩展 Shell 补齐，使之能够与别名一起使用：

```
echo 'alias k=kubectl' >> ~/.bash_profile
echo 'complete -F __start_kubectl k' >> ~/.bash_profile
```

- 如果你是所有 Homebrew 来安装 kubectl（如[前文](#)所述），kubectl 补齐脚本应该已经位于 /usr/local/etc/bash_completion.d/kubectl 目录下。在这种情况下，你就不用做任何操作了。

说明：Homebrew 安装 bash-completion v2 时会源引 BASH_COMPLETION_COMPAT_DIR 目录下的所有文件，这是为什么后面两种方法也可行的原因。

在任何一种情况下，重新加载 Shell 之后，kubectl 的自动补齐应该可以工作了。

Zsh 的 kubectl 补齐脚本可通过 `kubectl completion zsh` 命令来生成。在 Shell 环境中引用自动补齐脚本就可以启用 kubectl 自动补齐。

```
source <(kubectl completion zsh)
```

如果你为 kubectl 命令设置了别名（ alias ），你可以扩展 Shell 补齐，使之能够与别名一起使用：

```
echo 'alias k=kubectl' >> ~/.zshrc
echo 'complete -F __start_kubectl k' >> ~/.zshrc
```

重新加载 Shell 之后，kubectl 的自动补齐应该可以工作了。

如果你看到类似 `complete:13: command not found: compdef` 这种错误信息，可以将下面的命令添加到你的 `~/.zshrc` 文件的文件头：

```
autoload -Uz compinit
compinit
```

接下来

- [安装 Minikube](#)
- 参阅[入门指南](#)，了解创建集群相关的信息
- 了解如何[启动和暴露你的应用](#)
- 如果你需要访问别人创建的集群，参考 [共享集群访问文档](#)
- 阅读 [kubectl 参考文档](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 13, 2021 at 9:47 AM PST: [some are incorrect and some is outdated. \(024e3c0f6\)](#)

管理集群

了解管理集群的常见任务。

[用 kubeadm 进行管理](#)

[管理内存，CPU 和 API 资源](#)

[安装网络规则驱动](#)

[IP Masquerade Agent 用户指南](#)

[Kubernetes 云管理控制器](#)

[为 Kubernetes 运行 etcd 集群](#)

[为系统守护进程预留计算资源](#)

[为节点发布扩展资源](#)

[使用 CoreDNS 进行服务发现](#)

[使用 KMS 驱动进行数据加密](#)

[使用 Kubernetes API 访问集群](#)

[保护集群安全](#)

[关键插件 Pod 的调度保证](#)

[升级集群](#)

[名字空间演练](#)

[启用 EndpointSlices](#)

[启用/禁用 Kubernetes API](#)

[在 Kubernetes 集群中使用 NodeLocal DNSCache](#)

[在 Kubernetes 集群中使用 sysctl](#)

[在运行中的集群上重新配置节点的 kubelet](#)

[声明网络策略](#)

[安全地清空一个节点](#)

[开发云控制器管理器](#)

[开启服务拓扑](#)

[控制节点上的 CPU 管理策略](#)

[控制节点上的拓扑管理策略](#)

[搭建高可用的 Kubernetes Masters](#)

[改变默认 StorageClass](#)

[更改 PersistentVolume 的回收策略](#)

[自动扩缩集群 DNS 服务](#)

[自定义 DNS 服务](#)

[访问集群上运行的服务](#)

[调试 DNS 问题](#)

[通过名字空间共享集群](#)

[通过配置文件设置 Kubelet 参数](#)

[配置 API 对象配额](#)

[配置资源不足时的处理方式](#)

[限制存储消耗](#)

[静态加密 Secret 数据](#)

用 kubeadm 进行管理

[使用 kubeadm 进行证书管理](#)

[升级 kubeadm 集群](#)

[添加 Windows 节点](#)

[升级 Windows 节点](#)

使用 kubeadm 进行证书管理

FEATURE STATE: Kubernetes v1.15 [stable]

由 [kubeadm](#) 生成的客户端证书在 1 年后到期。 本页说明如何使用 kubeadm 管理证书续订。

准备开始

你应该熟悉 [Kubernetes 中的 PKI 证书和要求](#)。

使用自定义的证书

默认情况下, kubeadm 会生成运行一个集群所需的全部证书。 你可以通过提供你自己的证书来改变这个行为策略。

如果要这样做, 你必须将证书文件放置在通过 --cert-dir 命令行参数或者 kubeadm 配置中的 CertificatesDir 配置项指明的目录中。 默认的值是 /etc/kubernetes/pki。

如果在运行 kubeadm init 之前存在给定的证书和私钥对, kubeadm 将不会重写它们。 例如, 这意味着您可以将现有的 CA 复制到 /etc/kubernetes/pki/ca.crt 和 /etc/kubernetes/pki/ca.key 中, 而 kubeadm 将使用此 CA 对其余证书进行签名。

外部 CA 模式

只提供了 ca.crt 文件但是不提供 ca.key 文件也是可以的 (这只对 CA 根证书可用, 其它证书不可用)。 如果所有的其它证书和 kubeconfig 文件已就绪, kubeadm 检测到满足以上条件就会激活 "外部 CA" 模式。 kubeadm 将会在没有 CA 密钥文件的情况下继续执行。

否则, kubeadm 将独立运行 controller-manager, 附加一个 --controllers=csrsigner 的参数, 并且指明 CA 证书和密钥。

[PKI证书和要求](#)包括集群使用外部CA的设置指南。

[PKI 证书和要求](#)包括关于用外部 CA 设置集群的指南。

检查证书是否过期

你可以使用 check-expiration 子命令来检查证书何时过期

```
kubeadm alpha certs check-expiration
```

输出类似于以下内容 :

CERTIFICATE AUTHORITY	EXPIRES	RESIDUAL TIME	CERTIFICATE
EXTERNALLY MANAGED			
admin.conf	Dec 30, 2020 23:36 UTC	364d	no
apiserver	Dec 30, 2020 23:36 UTC	364d	ca no
apiserver-etcd-client	Dec 30, 2020 23:36 UTC	364d	etcd-ca
no			
apiserver-kubelet-client	Dec 30, 2020 23:36 UTC	364d	ca no
controller-manager.conf	Dec 30, 2020 23:36 UTC	364d	no
etcd-healthcheck-client	Dec 30, 2020 23:36 UTC	364d	etcd-ca
no			
etcd-peer	Dec 30, 2020 23:36 UTC	364d	etcd-ca no
etcd-server	Dec 30, 2020 23:36 UTC	364d	etcd-ca no
front-proxy-client	Dec 30, 2020 23:36 UTC	364d	front-proxy-ca

no			
scheduler.conf	Dec 30, 2020 23:36 UTC	364d	no
CERTIFICATE AUTHORITY EXPIRES RESIDUAL TIME EXTERNALLY			
MANAGED			
ca	Dec 28, 2029 23:36 UTC	9y	no
etcd-ca	Dec 28, 2029 23:36 UTC	9y	no
front-proxy-ca	Dec 28, 2029 23:36 UTC	9y	no

该命令显示 /etc/kubernetes/pki 文件夹中的客户端证书以及 kubeadm (admin.conf, controller-manager.conf 和 scheduler.conf) 使用的 KUBECONFIG 文件中嵌入的客户端证书的到期时间/剩余时间。

另外，kubeadm 会通知用户证书是否由外部管理；在这种情况下，用户应该小心的手动/使用其他工具来管理证书更新。

警告： kubeadm 不能管理由外部 CA 签名的证书

说明： 上面的列表中没有包含 kubelet.conf 因为 kubeadm 将 kubelet 配置为自动更新证书。

警告：

在通过 kubeadm init 创建的节点上，在 kubeadm 1.17 版本之前有一个 [缺陷](#)，该缺陷使得你必须手动修改 kubelet.conf 文件的内容。kubeadm init 操作结束之后，你必须更新 kubelet.conf 文件将 client-certificate-data 和 client-key-data 改为如下所示的内容以便使用轮换后的 kubelet 客户端证书：

client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
client-key: /var/lib/kubelet/pki/kubelet-client-current.pem

自动更新证书

kubeadm 会在控制面 [升级](#) 的时候更新所有证书。

这个功能旨在解决最简单的用例；如果你对此类证书的更新没有特殊要求，并且定期执行 Kubernetes 版本升级（每次升级之间的间隔时间少于 1 年），则 kubeadm 将确保你的集群保持最新状态并保持合理的安全性。

说明： 最佳的做法是经常升级集群以确保安全。

如果你对证书更新有更复杂的需求，则可通过将 --certificate-renewal=false 传递给 kubeadm upgrade apply 或者 kubeadm upgrade node，从而选择不采用默认行为。

警告： kubeadm 在 1.17 版本之前有一个 [缺陷](#)，该缺陷导致 kubeadm update node 执行时 --certificate-renewal 的默认值被设置为 false。在这种情况下，你需要显式地设置 --certificate-renewal=true。

手动更新证书

你能随时通过 `kubeadm alpha certs renew` 命令手动更新你的证书。

此命令用 CA (或者 front-proxy-CA) 证书和存储在 `/etc/kubernetes/pki` 中的密钥执行更新。

警告： 如果你运行了一个 HA 集群，这个命令需要在所有控制面板节点上执行。

说明： `certs renew` 使用现有的证书作为属性 (Common Name、Organization、SAN 等) 的权威来源，而不是 `kubeadm-config ConfigMap`。强烈建议使它们保持同步。

`kubeadm certs renew` 提供以下选项：

Kubernetes 证书通常在一年后到期。

- `--csr-only` 可用于经过一个外部 CA 生成的证书签名请求来更新证书（无需实际替换更新证书）；更多信息请参见下节。
- 可以更新单个证书而不是全部证书。

用 Kubernetes 证书 API 更新证书

本节提供有关如何使用 Kubernetes 证书 API 执行手动证书更新的更多详细信息。

注意： 这些是针对需要将其组织的证书基础结构集成到 `kubeadm` 构建的集群中的用户的高级主题。如果默认的 `kubeadm` 配置满足了你的需求，则应让 `kubeadm` 管理证书。

设置一个签名者 (Signer)

Kubernetes 证书颁发机构不是开箱即用。你可以配置外部签名者，例如 [cert-manager](#)，也可以使用内置签名者。内置签名者是 [kube-controller-manager](#) 的一部分。要激活内置签名者，请传递 `--cluster-signing-cert-file` 和 `--cluster-signing-key-file` 参数。

如果你正在创建一个新的集群，你可以使用 `kubeadm` 的 [配置文件](#)。

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    cluster-signing-cert-file: /etc/kubernetes/pki/ca.crt
    cluster-signing-key-file: /etc/kubernetes/pki/ca.key
```

创建证书签名请求 (CSR)

你可以用 `kubeadm alpha certs renew --use-api` 为 Kubernetes 证书 API 创建一个证书签名请求。

如果你设置例如 [cert-manager](#) 等外部签名者，证书签名请求 (CSRs) 会被自动批准。否则，你必须使用 [kubectl certificate](#) 命令手动批准证书。以下 kubeadm 命令输出要批准的证书名称，然后阻塞等待批准发生：

```
sudo kubeadm alpha certs renew apiserver --use-api &
```

输出类似于以下内容：

```
[1] 2890  
[certs] certificate request "kubeadm-cert-kube-apiserver-Id526" created
```

批准证书签名请求 (CSR)

如果你设置了一个外部签名者，证书签名请求 (CSRs) 会自动被批准。

否则，你必须用 [kubectl certificate](#) 命令手动批准证书，例如：

```
kubectl certificate approve kubeadm-cert-kube-apiserver-Id526
```

输出类似于以下内容：

```
certificatesigningrequest.certificates.k8s.io/kubeadm-cert-kube-apiserver-Id526  
approved
```

你可以使用 `kubectl get csr` 查看待处理证书列表。

通过外部 CA 更新证书

本节提供有关如何使用外部 CA 执行手动更新证书的更多详细信息。

为了更好的与外部 CA 集成，kubeadm 还可以生成证书签名请求 (CSR)。CSR 表示向 CA 请求客户的签名证书。在 kubeadm 术语中，通常由磁盘 CA 签名的任何证书都可以作为 CSR 生成。但是，CA 不能作为 CSR 生成。

创建证书签名请求 (CSR)

你可以通过 `kubeadm alpha certs renew --csr-only` 命令创建证书签名请求。

CSR 和随附的私钥都在输出中给出。你可以传入一个带有 `--csr-dir` 的目录，将 CRS 输出到指定位置。如果未指定 `--csr-dir`，则使用默认证书目录 (`/etc/kubernetes/pki`)。

证书可以通过 `kubeadm certs renew --csr-only` 来续订。和 `kubeadm init` 一样，可以使用 `--csr-dir` 标志指定一个输出目录。

CSR 签署证书后，必须将证书和私钥复制到 PKI 目录（默认情况下为 /etc/kubernetes/pki ）。

CSR 中包含一个证书的名字，域和 IP，但是未指定用法。颁发证书时，CA 有责任指定 [正确的证书用法](#)

- 在 openssl 中，这是通过 [openssl ca 命令](#) 来完成的。
- 在 cfssl 中，这是通过 [在配置文件中指定用法](#) 来完成的。

使用首选方法对证书签名后，必须将证书和私钥复制到 PKI 目录（默认为 /etc/kubernetes/pki ）。

证书机构 (CA) 轮换

kubeadm 并不直接支持对 CA 证书的轮换或者替换。

关于手动轮换或者置换 CA 的更多信息，可参阅 [手动轮换 CA 证书](#)。

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 11, 2020 at 10:52 AM PST: [zh-trans Update kubeadm-certs.md \(32063d9c9\)](#)

升级 kubeadm 集群

本页介绍如何将 kubeadm 创建的 Kubernetes 集群从 1.18.x 版本升级到 1.19.x 版本，或者从版本 1.19.x 升级到 1.19.y，其中 $y > x$ 。

要查看 kubeadm 创建的有关旧版本集群升级的信息，请参考以下页面：

- [将 kubeadm 集群从 1.17 升级到 1.18](#)
- [将 kubeadm 集群从 1.16 升级到 1.17](#)
- [将 kubeadm 集群从 1.15 升级到 1.16](#)
- [将 kubeadm 集群从 1.14 升级到 1.15](#)
- [将 kubeadm 集群从 1.13 升级到 1.14](#)

升级工作的基本流程如下：

1. 升级主控制平面节点
2. 升级其他控制平面节点
3. 升级工作节点

准备开始

- 你需要有一个由 kubeadm 创建并运行着 1.18.0 或更高版本的 Kubernetes 集群。
- 禁用交换分区。**
- 集群应使用静态的控制平面和 etcd Pod 或者 外部 etcd。
- 务必仔细认真阅读[发行说明](#)。
- 务必备份所有重要组件，例如存储在数据库中应用层面的状态。 kubeadm upgrade 不会影响你的工作负载，只会涉及 Kubernetes 内部的组件，但备份终究是好的。

附加信息

- 升级后，因为容器规约的哈希值已更改，所有容器都会被重新启动。
- 你只能从一个次版本升级到下一个次版本，或者在次版本相同时升级补丁版本。也就是说，升级时不可以跳过次版本。例如，你只能从 1.y 升级到 1.y+1，而不能从 from 1.y 升级到 1.y+2。

确定要升级到哪个版本

找到最新的稳定版 1.19：

- [Ubuntu、Debian 或 HypriotOS](#)
- [CentOS、RHEL 或 Fedora](#)

```
apt update  
apt-cache policy kubeadm  
# 在列表中查找最新的 1.19 版本  
# 它看起来应该是 1.19.x-00，其中 x 是最新的补丁
```

```
yum list --showduplicates kubeadm --disableexcludes=kubernetes  
# 在列表中查找最新的 1.19 版本  
# 它看起来应该是 1.19.x-0，其中 x 是最新的补丁版本
```

升级控制平面节点

升级第一个控制面节点

- 在第一个控制平面节点上，升级 kubeadm：
 - [Ubuntu、Debian 或 HypriotOS](#)
 - [CentOS、RHEL 或 Fedora](#)

```
# 用最新的修补程序版本替换 1.19.x-00 中的 x  
apt-mark unhold kubeadm && \  
apt-get update && apt-get install -y kubeadm=1.19.x-00 && \  
apt-mark hold kubeadm
```

```
# 从 apt-get 1.1 版本起，你也可以使用下面的方法  
apt-get update && \  
apt-get install -y --allow-change-held-packages kubeadm=1.19.x-00
```

```
# 用最新的修补程序版本替换 1.19.x-0 中的 x  
yum install -y kubeadm-1.19.x-0 --disableexcludes=kubernetes
```

- 验证下载操作正常，并且 kubeadm 版本正确：

```
kubeadm version
```

- 腾空控制平面节点：

```
# 将 <cp-node-name> 替换为你自己的控制面节点名称  
kubectl drain <cp-node-name> --ignore-daemonsets
```

- 在控制面节点上，运行：

```
sudo kubeadm upgrade plan
```

你应该可以看到与下面类似的输出：

```
[upgrade/config] Making sure the configuration is correct:  
[upgrade/config] Reading configuration from the cluster...  
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'  
[preflight] Running pre-flight checks.  
[upgrade] Running cluster health checks  
[upgrade] Fetching available versions to upgrade to  
[upgrade/versions] Cluster version: v1.18.4  
[upgrade/versions] kubeadm version: v1.19.0  
[upgrade/versions] Latest stable version: v1.19.0  
[upgrade/versions] Latest version in the v1.18 series: v1.18.4
```

Components that must be upgraded manually after you have upgraded the control plane with 'kubeadm upgrade apply':

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.18.4	v1.19.0

Upgrade to the latest version in the v1.18 series:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.18.4	v1.19.0
Controller Manager	v1.18.4	v1.19.0
Scheduler	v1.18.4	v1.19.0
Kube Proxy	v1.18.4	v1.19.0
CoreDNS	1.6.7	1.7.0

Etcdb	3.4.3-0	3.4.7-0
-------	---------	---------

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.19.0
```

The table below shows the current state of component configs as understood by this version of kubeadm.

Configs that have a "yes" mark in the "MANUAL UPGRADE REQUIRED" column require manual config upgrade or resetting to kubeadm defaults before a successful upgrade can be performed. The version to manually upgrade to is denoted in the "PREFERRED VERSION" column.

API GROUP	CURRENT VERSION	PREFERRED VERSION	
MANUAL UPGRADE REQUIRED			
kubeproxy.config.k8s.io	v1alpha1	v1alpha1	no
kubelet.config.k8s.io	v1beta1	v1beta1	no

此命令检查你的集群是否可以升级，并可以获取到升级的版本。其中也显示了组件配置版本状态的表格。

说明： kubeadm upgrade 也会自动对它在此节点上管理的证书进行续约。如果选择不对证书进行续约，可以使用标志 --certificate-renewal=false。关于更多细节信息，可参见[证书管理指南](#)。

说明：

如果 kubeadm upgrade plan 显示有任何组件配置需要手动升级，则用户必须通过命令行参数 --config 给 kubeadm upgrade apply 操作 提供带有替换配置的配置文件。

- 选择要升级到的版本，然后运行相应的命令。例如：

```
# 将 x 替换为你为此次升级所选的补丁版本号  
sudo kubeadm upgrade apply v1.19.x
```

你应该可以看见与下面类似的输出：

```
[upgrade/config] Making sure the configuration is correct:  
[upgrade/config] Reading configuration from the cluster...  
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'  
[preflight] Running pre-flight checks.  
[upgrade] Running cluster health checks
```

```
[upgrade/version] You have chosen to change the cluster version to  
"v1.19.0"  
[upgrade/versions] Cluster version: v1.18.4  
[upgrade/versions] kubeadm version: v1.19.0  
[upgrade/confirm] Are you sure you want to proceed with the upgrade? [y/  
N]: y  
[upgrade/prepull] Pulling images required for setting up a Kubernetes  
cluster  
[upgrade/prepull] This might take a minute or two, depending on the speed  
of your internet connection  
[upgrade/prepull] You can also perform this action in beforehand using  
'kubeadm config images pull'  
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version  
"v1.19.0"...  
Static pod: kube-apiserver-kind-control-plane hash:  
b4c8effe84b4a70031f9a49a20c8b003  
Static pod: kube-controller-manager-kind-control-plane hash:  
9ac092f0ca813f648c61c4d5fcfb39f2  
Static pod: kube-scheduler-kind-control-plane hash:  
7da02f2c78da17af7c2bf1533ecf8c9a  
[upgrade/etcd] Upgrading to TLS for etcd  
Static pod: etcd-kind-control-plane hash:  
171c56cd0e81c0db85e65d70361ceddf  
[upgrade/staticpods] Preparing for "etcd" upgrade  
[upgrade/staticpods] Renewing etcd-server certificate  
[upgrade/staticpods] Renewing etcd-peer certificate  
[upgrade/staticpods] Renewing etcd-healthcheck-client certificate  
[upgrade/staticpods] Moved new manifest to "/etc/kubernetes/manifests/  
etcd.yaml" and backed up old manifest to "/etc/kubernetes/tmp/kubeadm-  
backup-manifests-2020-07-13-16-24-16/etcd.yaml"  
[upgrade/staticpods] Waiting for the kubelet to restart the component  
[upgrade/staticpods] This might take a minute or longer depending on the  
component/version gap (timeout 5m0s)  
Static pod: etcd-kind-control-plane hash:  
171c56cd0e81c0db85e65d70361ceddf  
Static pod: etcd-kind-control-plane hash:  
171c56cd0e81c0db85e65d70361ceddf  
Static pod: etcd-kind-control-plane hash:  
59e40b2aab1cd7055e64450b5ee438f0  
[apiclient] Found 1 Pods for label selector component=etcd  
[upgrade/staticpods] Component "etcd" upgraded successfully!  
[upgrade/etcd] Waiting for etcd to become available  
[upgrade/staticpods] Writing new Static Pod manifests to "/etc/kubernetes/  
tmp/kubeadm-upgraded-manifests999800980"  
[upgrade/staticpods] Preparing for "kube-apiserver" upgrade  
[upgrade/staticpods] Renewing apiserver certificate
```

```
[upgrade/staticpods] Renewing apiserver-kubelet-client certificate
[upgrade/staticpods] Renewing front-proxy-client certificate
[upgrade/staticpods] Renewing apiserver-etcd-client certificate
[upgrade/staticpods] Moved new manifest to "/etc/kubernetes/manifests/
kube-apiserver.yaml" and backed up old manifest to "/etc/kubernetes/tmp/
kubeadm-backup-manifests-2020-07-13-16-24-16/kube-apiserver.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[upgrade/staticpods] This might take a minute or longer depending on the
component/version gap (timeout 5m0s)
Static pod: kube-apiserver-kind-control-plane hash:
b4c8effe84b4a70031f9a49a20c8b003
Static pod: kube-apiserver-kind-control-plane hash:
f717874150ba572f020dcd89db8480fc
[apiclient] Found 1 Pods for label selector component=kube-apiserver
[upgrade/staticpods] Component "kube-apiserver" upgraded successfully!
[upgrade/staticpods] Preparing for "kube-controller-manager" upgrade
[upgrade/staticpods] Renewing controller-manager.conf certificate
[upgrade/staticpods] Moved new manifest to "/etc/kubernetes/manifests/
kube-controller-manager.yaml" and backed up old manifest to "/etc/
kubernetes/tmp/kubeadm-backup-manifests-2020-07-13-16-24-16/kube-
controller-manager.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[upgrade/staticpods] This might take a minute or longer depending on the
component/version gap (timeout 5m0s)
Static pod: kube-controller-manager-kind-control-plane hash:
9ac092f0ca813f648c61c4d5fcfb39f2
Static pod: kube-controller-manager-kind-control-plane hash:
b155b63c70e798b806e64a866e297dd0
[apiclient] Found 1 Pods for label selector component=kube-controller-
manager
[upgrade/staticpods] Component "kube-controller-manager" upgraded
successfully!
[upgrade/staticpods] Preparing for "kube-scheduler" upgrade
[upgrade/staticpods] Renewing scheduler.conf certificate
[upgrade/staticpods] Moved new manifest to "/etc/kubernetes/manifests/
kube-scheduler.yaml" and backed up old manifest to "/etc/kubernetes/tmp/
kubeadm-backup-manifests-2020-07-13-16-24-16/kube-scheduler.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[upgrade/staticpods] This might take a minute or longer depending on the
component/version gap (timeout 5m0s)
```

```
Static pod: kube-scheduler-kind-control-plane hash:  
7da02f2c78da17af7c2bf1533ecf8c9a  
Static pod: kube-scheduler-kind-control-plane hash:  
260018ac854dbf1c9fe82493e88aec31  
[apiclient] Found 1 Pods for label selector component=kube-scheduler  
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!  
[upload-config] Storing the configuration used in ConfigMap "kubeadm-  
config" in the "kube-system" Namespace  
[kubelet] Creating a ConfigMap "kubelet-config-1.19" in namespace kube-  
system with the configuration for the kubelets in the cluster  
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/  
config.yaml"  
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to  
get nodes  
[bootstrap-token] configured RBAC rules to allow Node Bootstrap tokens to  
post CSRs in order for nodes to get long term certificate credentials  
[bootstrap-token] configured RBAC rules to allow the csapprover controller  
automatically approve CSRs from a Node Bootstrap Token  
[bootstrap-token] configured RBAC rules to allow certificate rotation for all  
node client certificates in the cluster  
W0713 16:26:14.074656 2986 dns.go:282] the CoreDNS Configuration will  
not be migrated due to unsupported version of CoreDNS. The existing  
CoreDNS Corefile configuration and deployment has been retained.  
[addons] Applied essential addon: CoreDNS  
[addons] Applied essential addon: kube-proxy
```

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.19.0".
Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed
with upgrading your kubelets if you haven't already done so.

- 手动升级你的 CNI 驱动插件。

你的容器网络接口 (CNI) 驱动应该提供了程序自身的升级说明。参阅[插件](#)页面查
找你 CNI 所提供的程序，并查看是否需要其他升级步骤。

如果 CNI 提供程序作为 DaemonSet 运行，则在其他控制平面节点上不需要此步
骤。

- 取消对控制面节点的保护

```
# 将 <cp-node-name> 替换为你的控制面节点名称  
kubectl uncordon <cp-node-name>
```

升级其他控制面节点

与第一个控制面节点类似，不过使用下面的命令：

```
sudo kubeadm upgrade node
```

而不是：

```
sudo kubeadm upgrade apply
```

同时，也不需要执行 sudo kubeadm upgrade plan。

升级 kubelet 和 kubectl

- [Ubuntu、Debian 或 HypriotOS](#)
- [CentOS、RHEL 或 Fedora](#)

```
# 用最新的补丁版本替换 1.19.x-00 中的 x
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.19.x-00 kubectl=1.19.x-00 && \
apt-mark hold kubelet kubectl
```

从 apt-get 的 1.1 版本开始，你也可以使用下面的方法：

```
apt-get update && \
apt-get install -y --allow-change-held-packages kubelet=1.19.x-00 kubectl=1.19.x-00
```

用最新的补丁版本替换 1.19.x-00 中的 x

```
yum install -y kubelet-1.19.x-0 kubectl-1.19.x-0 --disableexcludes=kubernetes
```

重启 kubelet

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

升级工作节点

工作节点上的升级过程应该一次执行一个节点，或者一次执行几个节点，以不影响运行工作负载所需的最小容量。

升级 kubeadm

- 在所有工作节点升级 kubeadm:
- [Ubuntu、Debian 或 HypriotOS](#)
- [CentOS、RHEL 或 Fedora](#)

```
# 将 1.19.x-00 中的 x 替换为最新的补丁版本  
apt-mark unhold kubeadm && \  
apt-get update && apt-get install -y kubeadm=1.19.x-00 && \  
apt-mark hold kubeadm
```

从 apt-get 的 1.1 版本开始，你也可以使用下面的方法：

```
apt-get update && \  
apt-get install -y --allow-change-held-packages kubeadm=1.19.x-00
```

```
# 用最新的补丁版本替换 1.19.x-00 中的 x  
yum install -y kubeadm-1.19.x-0 --disableexcludes=kubernetes
```

腾空节点

- 通过将节点标记为不可调度并逐出工作负载，为维护做好准备。运行：

```
# 将 <node-to-drain> 替换为你正在腾空的节点的名称  
kubectl drain <node-to-drain> --ignore-daemonsets
```

你应该可以看见与下面类似的输出：

```
node/ip-172-31-85-18 cordoned  
WARNING: ignoring DaemonSet-managed Pods: kube-system/kube-proxy-  
dj7d7, kube-system/weave-net-z65qx  
node/ip-172-31-85-18 drained
```

升级 kubelet 配置

- 升级 kubelet 配置：

```
sudo kubeadm upgrade node
```

升级 kubelet 与 kubectl

- 在所有工作节点上升级 kubelet 和 kubectl：
 - [Ubuntu、Debian 或 HypriotOS](#)
 - [CentOS, RHEL or Fedora](#)

```
# 将 1.19.x-00 中的 x 替换为最新的补丁版本  
apt-mark unhold kubelet kubectl && \  
apt-get update && apt-get install -y kubelet=1.19.x-00 kubectl=1.19.x-00 && \  
apt-mark hold kubelet kubectl
```

从 apt-get 的 1.1 版本开始，你也可以使用下面的方法：

```
apt-get update && \  
apt-get install -y --allow-change-held-packages kubelet=1.19.x-00 kubectl=1.19.x-00
```

```
apt-get install -y --allow-change-held-packages kubelet=1.19.x-00 kubectl=1.19.x-00
```

将 1.18.x-00 中的 x 替换为最新的补丁版本

```
yum install -y kubelet-1.19.x-0 kubectl-1.19.x-0 --disableexcludes=kubernetes
```

- 重启 kubelet

```
sudo systemctl daemon-reload  
sudo systemctl restart kubelet
```

取消对节点的保护

- 通过将节点标记为可调度，让节点重新上线：

```
# 将 <node-to-drain> 替换为当前节点的名称  
kubectl uncordon <node-to-drain>
```

验证集群的状态

在所有节点上升级 kubelet 后，通过从 kubectl 可以访问集群的任何位置运行以下命令，验证所有节点是否再次可用：

```
kubectl get nodes
```

STATUS 应显示所有节点为 Ready 状态，并且版本号已经被更新。

从故障状态恢复

如果 kubeadm upgrade 失败并且没有回滚，例如由于执行期间意外关闭，你可以再次运行 kubeadm upgrade。此命令是幂等的，并最终确保实际状态是你声明的所需状态。要从故障状态恢复，你还可以运行 kubeadm upgrade --force 而不去更改集群正在运行的版本。

在升级期间，kubeadm 向 /etc/kubernetes/tmp 目录下的如下备份文件夹写入数据：

- kubeadm-backup-etcd-<date>-<time>
- kubeadm-backup-manifests-<date>-<time>

kubeadm-backup-etcd 包含当前控制面节点本地 etcd 成员数据的备份。如果 etcd 升级失败并且自动回滚也无法修复，则可以将此文件夹中的内容复制到 /var/lib/etcd 进行手工修复。如果使用的是外部的 etcd，则此备份文件夹为空。

kubeadm-backup-manifests 包含当前控制面节点的静态 Pod 清单文件的备份版本。如果升级失败并且无法自动回滚，则此文件夹中的内容可以复制到 /etc/kubernetes/manifests 目录实现手工恢复。如果由于某些原因，在升级前后某个组件的清单未发生变化，则 kubeadm 也不会为之生成备份版本。

工作原理

kubeadm upgrade apply 做了以下工作：

- 检查你的集群是否处于可升级状态：
 - API 服务器是可访问的
 - 所有节点处于 Ready 状态
 - 控制面是健康的
- 强制执行版本偏差策略。
- 确保控制面的镜像是可用的或可拉取到服务器上。
- 如果组件配置要求版本升级，则生成替代配置与/或使用用户提供的覆盖版本配置。
- 升级控制面组件或回滚（如果其中任何一个组件无法启动）。
- 应用新的 kube-dns 和 kube-proxy 清单，并强制创建所有必需的 RBAC 规则。
- 如果旧文件在 180 天后过期，将创建 API 服务器的新证书和密钥文件并备份旧文件。

kubeadm upgrade node 在其他控制平节点上执行以下操作：

- 从集群中获取 kubeadm ClusterConfiguration。
- 可选地备份 kube-apiserver 证书。
- 升级控制平面组件的静态 Pod 清单。
- 为本节点升级 kubelet 配置

kubeadm upgrade node 在工作节点上完成以下工作：

- 从集群取回 kubeadm ClusterConfiguration。
- 为本节点升级 kubelet 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 November 23, 2020 at 4:41 PM PST: [\[zh\] Resync kubeadm-upgrade \(6c447c928\)](#)

添加 Windows 节点

FEATURE STATE: Kubernetes v1.18 [beta]

你可以使用 Kubernetes 来混合运行 Linux 和 Windows 节点，这样你就可以混合使用运行于 Linux 上的 Pod 和运行于 Windows 上的 Pod。本页面展示如何将 Windows 节点注册到你的集群。

准备开始

您的 Kubernetes 服务器版本必须不低于版本 1.17。要获知版本信息，请输入 `kubectl version`。

- 获取 [Windows Server 2019 或更高版本的授权](#) 以便配置托管 Windows 容器的 Windows 节点。如果你在使用 VXLAN/覆盖 (Overlay) 联网设施，则你还必须安装 [KB4489899](#)。
- 一个利用 `kubeadm` 创建的基于 Linux 的 Kubernetes 集群；你能访问该集群的控制面（参见[使用 `kubeadm` 创建一个单控制面的集群](#)）。

教程目标

- 将一个 Windows 节点注册到集群上
- 配置网络，以便 Linux 和 Windows 上的 Pod 和 Service 之间能够相互通信。

开始行动：向你的集群添加一个 Windows 节点

联网配置

一旦你有了一个基于 Linux 的 Kubernetes 控制面节点，你就可以为其选择联网方案。出于简单考虑，本指南展示如何使用 VXLAN 模式的 Flannel。

配置 Flannel

1. 为 Flannel 准备 Kubernetes 的控制面

在我们的集群中，建议对 Kubernetes 的控制面进行少许准备处理。建议在使用 Flannel 时为 iptables 链启用桥接方式的 IPv4 流处理，必须在所有 Linux 节点上执行如下命令：

```
sudo sysctl net.bridge.bridge-nf-call-iptables=1
```

1. 下载并配置 Linux 版本的 Flannel

下载最新的 Flannel 清单文件：

```
wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

修改 Flannel 清单中的 `net-conf.json` 部分，将 VNI 设置为 4096，并将 Port 设置为 4789。结果看起来像下面这样：

```
net-conf.json: |
{
  "Network": "10.244.0.0/16",
  "Backend": {
```

```
        "Type": "vxlan",
        "VNI": 4096,
        "Port": 4789
    }
}
```

说明：在 Linux 节点上 VNI 必须设置为 4096，端口必须设置为 4789，这样才能令其与 Windows 上的 Flannel 互操作。关于这些字段的详细说明，请参见 [VXLAN 文档](#)。

说明：如要使用 L2Bridge/Host-gateway 模式，则可将 Type 值设置为 "host-gw"，并忽略 VNI 和 Port 的设置。

1. 应用 Flannel 清单并验证

首先应用 Flannel 配置：

```
kubectl apply -f kube-flannel.yml
```

几分钟之后，如果 Flannel Pod 网络被正确部署，你应该会看到所有 Pods 都处于运行中状态。

```
kubectl get pods -n kube-system
```

输出中应该包含处于运行中状态的 Linux Flannel DaemonSet：

NAMESPACE	NAME	READY	STATUS
RESTARTS	AGE		
...			
kube-system	kube-flannel-ds-54954	1/1	Running
0	1m		

1. 添加 Windows Flannel 和 kube-proxy DaemonSet

现在你可以添加 Windows 兼容版本的 Flannel 和 kube-proxy。为了确保你能获得兼容 版本的 kube-proxy，你需要替换镜像中的标签。下面的例子中展示的是针对 Kubernetes v1.20.0 版本的用法，不过你应该根据你自己的集群部署调整其中的版本号。

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/
latest/download/kube-proxy.yml | sed 's/VERSION/v1.20.0/g' | kubectl apply
-f -
kubectl apply -f https://github.com/kubernetes-sigs/sig-windows-tools/
releases/latest/download/flannel-overlay.yml
```

说明：如果你在使用 host-gateway 模式，则应该使用 <https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel-host-gw.yml> 这一清单。

说明：

如果你在 Windows 节点上使用的不是以太网（即，"Ethernet0 2"）接口，你需要修改 flannel-host-gw.yml 或 flannel-overlay.yml 文件中的下面这行：

```
wins cli process run --path /k/flannel/setup.exe --args "--mode=overlay --interface=Ethernet"
```

在其中根据情况设置要使用的网络接口。

Example

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel-overlay.yml | sed 's/Ethernet/Ethernet0 2/g' | kubectl apply -f -
```

加入 Windows 工作节点 {joining-a-windows-worker-node}

你必须安装 Containers 功能特性并安装 Docker 工具。相关的指令可以在 [Install Docker Engine - Enterprise on Windows Servers](#) 处找到。

Windows 节的所有代码片段都需要在 PowerShell 环境中执行，并且要求在 Windows 工作节点上具有提升的权限（Administrator）。

1. 安装 wins、kubelet 和 kubeadm

```
curl.exe -LO https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/PrepareNode.ps1  
.\\PrepareNode.ps1 -KubernetesVersion v1.20.0
```

1. 运行 kubeadm 添加节点

当你在控制面主机上运行 kubeadm init 时，输出了一个命令。现在运行这个命令。如果你找不到这个命令，或者命令中对应的令牌已经过期，你可以（在一个控制面主机上）运行 kubeadm token create --print-join-command 来生成新的令牌和 join 命令。

检查你的安装

你现在应该能够通过运行下面的命令来查看集群中的 Windows 节点了：

```
kubectl get nodes -o wide
```

如果你的新节点处于 NotReady 状态，很可能的原因是系统仍在下载 Flannel 镜像。你可以像之前一样，通过检查 kube-system 名字空间中的 Flannel Pods 来了解安装进度。

```
kubectl -n kube-system get pods -l app=flannel
```

一旦 Flannel Pod 运行起来，你的节点就应该能进入 Ready 状态并可用来处理负载。

接下来

- [升级 kubeadm 安装的 Windows 节点](#)

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 6:26 PM PST: [Update adding-windows-nodes.md \(4e2959c82\)](#)

升级 Windows 节点

FEATURE STATE: Kubernetes v1.18 [beta]

本页解释如何升级[用 kubeadm 创建的](#) Windows 节点。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 1.17. 要获知版本信息，请输入 kubectl version.

- 熟悉[更新 kubeadm 集群中的其余组件](#)。在升级你的 Windows 节点之前你会想要升级控制面节点。

升级工作节点

升级 kubeadm

1. 在 Windows 节点上升级 kubeadm：

```
# 将 v1.20.0 替换为你希望的版本
curl.exe -Lo C:\k\kubeadm.exe https://dl.k8s.io/v1.20.0/bin/windows/amd64/
kubeadm.exe
```

腾空节点

1. 在一个能访问到 Kubernetes API 的机器上，将 Windows 节点标记为不可调度并驱逐其上的所有负载，以便准备节点维护操作：

```
# 将 <要腾空的节点> 替换为你要腾空的节点的名称
kubectl drain <要腾空的节点> --ignore-daemonsets
```

你应该会看到类似下面的输出：

```
node/ip-172-31-85-18 cordoned
node/ip-172-31-85-18 drained
```

升级 kubelet 配置

1. 在 Windows 节点上，执行下面的命令来同步新的 kubelet 配置：

```
kubeadm upgrade node
```

升级 kubelet

1. 在 Windows 节点上升级并重启 kubelet：

```
stop-service kubelet
curl.exe -Lo C:\k\kubelet.exe https://dl.k8s.io/v1.20.0/bin/windows/amd64/
kubelet.exe
restart-service kubelet
```

对节点执行 uncordon 操作

1. 从一台能够访问到 Kubernetes API 的机器上，通过将节点标记为可调度，使之重新上线：

```
# 将 <要腾空的节点> 替换为你的节点名称
kubectl uncordon <要腾空的节点>
```

升级 kube-proxy

1. 在一台可访问 Kubernetes API 的机器上和，将 v1.20.0 替换成你期望的版本后再次执行下面的命令：

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/
latest/download/kube-proxy.yaml | sed 's/VERSION/v1.20.0/g' | kubectl apply
-f -
```

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 19, 2020 at 12:01 PM PST: [\[zh\] Translate upgrading-windows-nodes \(3a19c6b7b\)](#)

管理内存，CPU 和 API 资源

[为命名空间配置默认的内存请求和限制](#)

[为命名空间配置默认的 CPU 请求和限制](#)

[配置命名空间的最小和最大内存约束](#)

[为命名空间配置 CPU 最小和最大约束](#)

[为命名空间配置内存和 CPU 配额](#)

[配置命名空间下 Pod 配额](#)

为命名空间配置默认的内存请求和限制

本文介绍怎样给命名空间配置默认的内存请求和限制。如果在一个有默认内存限制的命名空间创建容器，该容器没有声明自己的内存限制时，将会被指定默认内存限制。Kubernetes 还为某些情况指定了默认的内存请求，本章后面会进行介绍。

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.

你的集群中的每个节点必须至少有 2 GiB 的内存。

创建命名空间

创建一个命名空间，以便本练习中所建的资源与集群的其余资源相隔离。

```
kubectl create namespace default-mem-example
```

创建 LimitRange 和 Pod

这里给出了一个限制范围对象的配置文件。该配置声明了一个默认的内存请求和一个默认的内存限制。

[admin/resource/memory-defaults.yaml](#)



```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
  type: Container
```

在 default-mem-example 命名空间创建限制范围：

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults.yaml
--namespace=default-mem-example
```

现在，如果在 default-mem-example 命名空间创建容器，并且该容器没有声明自己的内存请求和限制值，它将被指定默认的内存请求 256 MiB 和默认的内存限制 512 MiB。

下面是具有一个容器的 Pod 的配置文件。容器未指定内存请求和限制。

[admin/resource/memory-defaults-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo
spec:
  containers:
  - name: default-mem-demo-ctr
    image: nginx
```

创建 Pod

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-
pod.yaml --namespace=default-mem-example
```

查看 Pod 的详情：

```
kubectl get pod default-mem-demo --output=yaml --namespace=default-mem-example
```

输出内容显示该 Pod 的容器有 256 MiB 的内存请求和 512 MiB 的内存限制。这些都是 LimitRange 设置的默认值。

```
containers:  
- image: nginx  
  imagePullPolicy: Always  
  name: default-mem-demo-ctr  
resources:  
  limits:  
    memory: 512Mi  
  requests:  
    memory: 256Mi
```

删除你的 Pod：

```
kubectl delete pod default-mem-demo --namespace=default-mem-example
```

声明容器的限制而不声明它的请求会怎么样？

这里给出了包含一个容器的 Pod 的配置文件。该容器声明了内存限制，而没有声明内存请求：

[admin/resource/memory-defaults-pod-2.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: default-mem-demo-2  
spec:  
  containers:  
    - name: default-mem-demo-2-ctr  
      image: nginx  
      resources:  
        limits:  
          memory: "1Gi"
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod-2.yaml --namespace=default-mem-example
```

查看 Pod 的详情：

```
kubectl get pod default-mem-demo-2 --output=yaml --namespace=default-mem-example
```

输出结果显示容器的内存请求被设置为它的内存限制相同的值。注意该容器没有被指定默认的内存请求值 256MiB。

```
resources:  
  limits:  
    memory: 1Gi  
  requests:  
    memory: 1Gi
```

声明容器的内存请求而不声明内存限制会怎么样？

这里给出了一个包含一个容器的 Pod 的配置文件。该容器声明了内存请求，但没有内存限制：

[admin/resource/memory-defaults-pod-3.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: default-mem-demo-3  
spec:  
  containers:  
  - name: default-mem-demo-3-ctr  
    image: nginx  
    resources:  
      requests:  
        memory: "128Mi"
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod-3.yaml --namespace=default-mem-example
```

查看 Pod 声明：

```
kubectl get pod default-mem-demo-3 --output=yaml --namespace=default-mem-example
```

输出结果显示该容器的内存请求被设置为了容器配置文件中声明的数值。容器的内存限制被设置为 512MiB，即命名空间的默认内存限制。

```
resources:  
  limits:  
    memory: 512Mi
```

```
requests:  
  memory: 128Mi
```

设置默认内存限制和请求的动机

如果你的命名空间有资源配置，那么默认内存限制是很有帮助的。下面是一个例子，通过资源配置为命名空间设置两项约束：

- 运行在命名空间中的每个容器必须有自己的内存限制。
- 命名空间中所有容器的内存使用量之和不能超过声明的限制值。

如果一个容器没有声明自己的内存限制，会被指定默认限制，然后它才会被允许在限定了配额的命名空间中运行。

清理

删除你的命名空间：

```
kubectl delete namespace default-mem-example
```

接下来

集群管理员参考

- [为命名空间配置默认的 CPU 请求和限制](#)
- [为命名空间配置最小和最大内存限制](#)
- [为命名空间配置最小和最大 CPU 限制](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)
- [为 Pod 配置服务质量](#)

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 19, 2020 at 6:27 PM PST: [Update memory-default-namespace.md \(cb901fdeb\)](#)

为命名空间配置默认的 CPU 请求和限制

本章介绍怎样为命名空间配置默认的 CPU 请求和限制。一个 Kubernetes 集群可被划分为多个命名空间。如果在配置了 CPU 限制的命名空间创建容器，并且该容器没有声明自己的 CPU 限制，那么这个容器会被指定默认的 CPU 限制。Kubernetes 在一些特定情况还会指定 CPU 请求，本文后续章节将会对其进行解释。

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.

创建命名空间

创建一个命名空间，以便本练习中创建的资源和集群的其余部分相隔离。

```
kubectl create namespace default-cpu-example
```

创建 LimitRange 和 Pod

这里给出了 LimitRange 对象的配置文件。该配置声明了一个默认的 CPU 请求和一个默认的 CPU 限制。

[admin/resource/cpu-defaults.yaml](#)



```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
    cpu: 1
  defaultRequest:
    cpu: 0.5
  type: Container
```

在命名空间 default-cpu-example 中创建 LimitRange 对象：

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults.yaml --namespace=default-cpu-example
```

现在如果在 default-cpu-example 命名空间创建一个容器，该容器没有声明自己的 CPU 请求和限制时，将会给它指定默认的 CPU 请求0.5和默认的 CPU 限制值1.

这里给出了包含一个容器的 Pod 的配置文件。该容器没有声明 CPU 请求和限制。

[admin/resource/cpu-defaults-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo
spec:
  containers:
    - name: default-cpu-demo-ctr
      image: nginx
```

创建 Pod。

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-pod.yaml --namespace=default-cpu-example
```

查看该 Pod 的声明：

```
kubectl get pod default-cpu-demo --output=yaml --namespace=default-cpu-example
```

输出显示该 Pod 的容器有一个500 millicpus的 CPU 请求和一个1 cpu的 CPU 限制。这些是 LimitRange 声明的默认值。

```
containers:
- image: nginx
  imagePullPolicy: Always
  name: default-cpu-demo-ctr
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 500m
```

你只声明容器的限制，而不声明请求会怎么样？

这是包含一个容器的 Pod 的配置文件。该容器声明了 CPU 限制，而没有声明 CPU 请求。

[admin/resource/cpu-defaults-pod-2.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
  resources:
    limits:
      cpu: "1"
```

创建 Pod

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-
pod-2.yaml --namespace=default-cpu-example
```

查看 Pod 的声明：

```
kubectl get pod default-cpu-demo-2 --output=yaml --namespace=default-cpu-
example
```

输出显示该容器的 CPU 请求和 CPU 限制设置相同。注意该容器没有被指定默认的 CPU 请求值0.5 cpu。

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: "1"
```

你只声明容器的请求，而不声明它的限制会怎么样？

这里给出了包含一个容器的 Pod 的配置文件。该容器声明了 CPU 请求，而没有声明 CPU 限制。

[admin/resource/cpu-defaults-pod-3.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-3
spec:
```

```
containers:
- name: default-cpu-demo-3-ctr
  image: nginx
  resources:
    requests:
      cpu: "0.75"
```

创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-pod-3.yaml --namespace=default-cpu-example
```

查看 Pod 的规约 :

```
kubectl get pod default-cpu-demo-3 --output=yaml --namespace=default-cpu-example
```

结果显示该容器的 CPU 请求被设置为容器配置文件中声明的数值。 容器的CPU限制被设置为 1 CPU , 即该命名空间的默认 CPU 限制值。

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 750m
```

默认 CPU 限制和请求的动机

如果你的命名空间有一个 [资源配置](#) , 那么有一个默认的 CPU 限制是有帮助的。这里有资源配置强加给命名空间的两条限制 :

- 命名空间中运行的每个容器必须有自己的 CPU 限制。
- 命名空间中所有容器使用的 CPU 总和不能超过一个声明值。

如果容器没有声明自己的 CPU 限制 , 将会给它一个默认限制 , 这样它就能被允许运行在一个有配额限制的命名空间中。

清理

删除你的命名空间 :

```
kubectl delete namespace constraints-cpu-example
```

接下来

集群管理员参考

- [为命名空间配置默认内存请求和限制](#)

- [为命名空间配置内存限制的最小值和最大值](#)
- [为命名空间配置 CPU 限制的最小值和最大值](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)
- [为 Pod 配置服务质量](#)

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 11, 2020 at 9:53 AM PST: [Update cpu-default-namespace.md \(7e998a73c\)](#)

配置命名空间的最小和最大内存约束

本页介绍如何设置在命名空间中运行的容器使用的内存的最小值和最大值。你可以在 [LimitRange](#) 对象中指定最小和最大内存值。如果 Pod 不满足 LimitRange 施加的约束，则无法在命名空间中创建它。

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.

集群中每个节点必须至少要有 1 GiB 的内存。

创建命名空间

创建一个命名空间，以便在此练习中创建的资源与群集的其余资源隔离。

```
kubectl create namespace constraints-mem-example
```

创建 LimitRange 和 Pod

下面是 LimitRange 的配置文件：

[admin/resource/memory-constraints.yaml](#)


```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
    - max:
        memory: 1Gi
      min:
        memory: 500Mi
    type: Container
```

创建 LimitRange：

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-
constraints.yaml --namespace=constraints-mem-example
```

查看 LimitRange 的详情：

```
kubectl get limitrange mem-min-max-demo-lr --namespace=constraints-mem-
example --output=yaml
```

输出显示预期的最小和最大内存约束。但请注意，即使你没有在 LimitRange 的配置文件中指定默认值，也会自动创建它们。

```
limits:
- default:
    memory: 1Gi
  defaultRequest:
    memory: 1Gi
  max:
    memory: 1Gi
  min:
    memory: 500Mi
  type: Container
```

现在，只要在 constraints-mem-example 命名空间中创建容器，Kubernetes 就会执行下面的步骤：

- 如果 Container 未指定自己的内存请求和限制，将为它指定默认的内存请求和限制。

验证 Container 的内存请求是否大于或等于 500 MiB。

- 验证 Container 的内存限制是否小于或等于 1 GiB。

这里给出了包含一个 Container 的 Pod 配置文件。Container 声明了 600 MiB 的内存请求和 800 MiB 的内存限制，这些满足了 LimitRange 施加的最小和最大内存约束。

[admin/resource/memory-constraints-pod.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo
spec:
  containers:
  - name: constraints-mem-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "600Mi"
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-
pod.yaml --namespace=constraints-mem-example
```

确认下 Pod 中的容器在运行：

```
kubectl get pod constraints-mem-demo --namespace=constraints-mem-example
```

查看 Pod 详情：

```
kubectl get pod constraints-mem-demo --output=yaml --namespace=constraints-
-mem-example
```

输出结果显示容器的内存请求为 600 MiB，内存限制为 800 MiB。这些满足了 LimitRange 设定的限制范围。

```
resources:
  limits:
    memory: 800Mi
  requests:
    memory: 600Mi
```

删除你创建的 Pod：

```
kubectl delete pod constraints-mem-demo --namespace=constraints-mem-example
```

尝试创建一个超过最大内存限制的 Pod

这里给出了包含一个容器的 Pod 的配置文件。容器声明了800 MiB 的内存请求和1.5 GiB 的内存限制。

[admin/resource/memory-constraints-pod-2.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-2
spec:
  containers:
  - name: constraints-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1.5Gi"
      requests:
        memory: "800Mi"
```

尝试创建 Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-2.yaml --namespace=constraints-mem-example
```

输出结果显示 Pod 没有创建成功，因为容器声明的内存限制太大了：

```
Error from server (Forbidden): error when creating "examples/admin/resource/memory-constraints-pod-2.yaml":
pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Container is 1Gi, but limit is 1536Mi.
```

尝试创建一个不满足最小内存请求的 Pod

这里给出了包含一个容器的 Pod 的配置文件。容器声明了100 MiB 的内存请求和800 MiB 的内存限制。

[admin/resource/memory-constraints-pod-3.yaml](#)


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

```
metadata:  
  name: constraints-mem-demo-3  
spec:  
  containers:  
    - name: constraints-mem-demo-3-ctr  
      image: nginx  
      resources:  
        limits:  
          memory: "800Mi"  
        requests:  
          memory: "100Mi"
```

尝试创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-3.yaml --namespace=constraints-mem-example
```

输出结果显示 Pod 没有创建成功，因为容器声明的内存请求太小了：

```
Error from server (Forbidden): error when creating "examples/admin/resource/memory-constraints-pod-3.yaml":  
pods "constraints-mem-demo-3" is forbidden: minimum memory usage per Container is 500Mi, but request is 100Mi.
```

创建一个没有声明内存请求和限制的 Pod

这里给出了包含一个容器的 Pod 的配置文件。容器没有声明内存请求，也没有声明内存限制。

[admin/resource/memory-constraints-pod-4.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: constraints-mem-demo-4  
spec:  
  containers:  
    - name: constraints-mem-demo-4-ctr  
      image: nginx
```

创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-4.yaml --namespace=constraints-mem-example
```

查看 Pod 详情：

```
kubectl get pod constraints-mem-demo-4 --namespace=constraints-mem-example --output=yaml
```

输出结果显示 Pod 的内存请求为 1 GiB，内存限制为 1 GiB。容器怎样获得哪些数值呢？

```
resources:  
  limits:  
    memory: 1Gi  
  requests:  
    memory: 1Gi
```

因为你的容器没有声明自己的内存请求和限制，它从 LimitRange 那里获得了 [默认的内存请求和限制](#)。

此时，你的容器可能运行起来也可能没有运行起来。回想一下我们本次任务的先决条件是你的每个节点都至少有 1 GiB 的内存。如果你的每个节点都只有 1 GiB 的内存，那将没有一个节点拥有足够的可分配内存来满足 1 GiB 的内存请求。

删除你的 Pod：

```
kubectl delete pod constraints-mem-demo-4 --namespace=constraints-mem-example
```

强制执行内存最小和最大限制

LimitRange 为命名空间设定的最小和最大内存限制只有在 Pod 创建和更新时才会强制执行。如果你更新 LimitRange，它不会影响此前创建的 Pod。

设置内存最小和最大限制的动因

做为集群管理员，你可能想规定 Pod 可以使用的内存总量限制。例如：

- 集群的每个节点有 2 GB 内存。你不想接受任何请求超过 2 GB 的 Pod，因为集群中没有节点可以满足。
- 集群由生产部门和开发部门共享。你希望允许产品部门的负载最多耗用 8 GB 内存，但是开发部门的负载最多可使用 512 MiB。这时，你可以为产品部门和开发部门分别创建名字空间，并为各个名字空间设置内存约束。

清理

删除你的命名空间：

```
kubectl delete namespace constraints-mem-example
```

接下来

集群管理员参考

- [为命名空间配置默认内存请求和限制](#)
- [为命名空间配置内存限制的最小值和最大值](#)
- [为命名空间配置 CPU 限制的最小值和最大值](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)
- [为 Pod 配置服务质量](#)

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 14, 2020 at 12:05 AM PST: [Update memory-constraint-namespace.md \(bfdeacdaa\)](#)

为命名空间配置 CPU 最小和最大约束

本页介绍如何为命名空间中容器和 Pod 使用的 CPU 资源设置最小和最大值。你可以通过 [LimitRange](#) 对象声明 CPU 的最小和最大值. 如果 Pod 不能满足 LimitRange 的限制，它就不能在命名空间中创建。

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.

你的集群中每个节点至少要有 1 个 CPU 可用才能运行本任务示例。

创建命名空间

创建一个命名空间，以便本练习中创建的资源和集群的其余资源相隔离。

```
kubectl create namespace constraints-cpu-example
```

创建 LimitRange 和 Pod

这里给出了 LimitRange 的配置文件：

[admin/resource/cpu-constraints.yaml](#)



```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-min-max-demo-lr
spec:
  limits:
  - max:
      cpu: "800m"
    min:
      cpu: "200m"
  type: Container
```

创建 LimitRange：

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints.yaml --namespace=constraints-cpu-example
```

查看 LimitRange 详情：

```
kubectl get limitrange cpu-min-max-demo-lr --output=yaml --namespace=constraints-cpu-example
```

输出结果显示 CPU 的最小和最大限制符合预期。但需要注意的是，尽管你在 LimitRange 的配置文件中你没有声明默认值，默认值也会被自动创建。

```
limits:
- default:
    cpu: 800m
defaultRequest:
  cpu: 800m
max:
  cpu: 800m
min:
```

```
cpu: 200m  
type: Container
```

现在不管什么时候在 constraints-cpu-example 命名空间中创建容器，Kubernetes 都会执行下面这些步骤：

- 如果容器没有声明自己的 CPU 请求和限制，将为容器指定默认 CPU 请求和限制。
- 核查容器声明的 CPU 请求确保其大于或者等于 200 millicpu。
- 核查容器声明的 CPU 限制确保其小于或者等于 800 millicpu。

说明：当创建 LimitRange 对象时，你也可以声明大页面和 GPU 的限制。当这些资源同时声明了 'default' 和 'defaultRequest' 参数时，两个参数值必须相同。

这里给出了包含一个容器的 Pod 的配置文件。该容器声明了 500 millicpu 的 CPU 请求和 800 millicpu 的 CPU 限制。这些参数满足了 LimitRange 对象规定的 CPU 最小和最大限制。

[admin/resource/cpu-constraints-pod.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: constraints-cpu-demo  
spec:  
  containers:  
    - name: constraints-cpu-demo-ctr  
      image: nginx  
      resources:  
        limits:  
          cpu: "800m"  
        requests:  
          cpu: "500m"
```

创建Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod.yaml --namespace=constraints-cpu-example
```

确认一下 Pod 中的容器在运行：

```
kubectl get pod constraints-cpu-demo --namespace=constraints-cpu-example
```

查看 Pod 的详情：

```
kubectl get pod constraints-cpu-demo --output=yaml --namespace=constraints-cpu-example
```

输出结果表明容器的 CPU 请求为 500 millicpu , CPU 限制为 800 millicpu。 这些参数满足 LimitRange 规定的限制范围。

```
resources:  
  limits:  
    cpu: 800m  
  requests:  
    cpu: 500m
```

删除 Pod

```
kubectl delete pod constraints-cpu-demo --namespace=constraints-cpu-example
```

尝试创建一个超过最大 CPU 限制的 Pod

这里给出了包含一个容器的 Pod 的配置文件。容器声明了 500 millicpu 的 CPU 请求和 1.5 CPU 的 CPU 限制。

[admin/resource/cpu-constraints-pod-2.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: constraints-cpu-demo-2  
spec:  
  containers:  
  - name: constraints-cpu-demo-2-ctr  
    image: nginx  
  resources:  
    limits:  
      cpu: "1.5"  
    requests:  
      cpu: "500m"
```

尝试创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-2.yaml --namespace=constraints-cpu-example
```

输出结果表明 Pod 没有创建成功，因为容器声明的 CPU 限制太大了：

```
Error from server (Forbidden): error when creating "examples/admin/resource/cpu-constraints-pod-2.yaml":
```

```
pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container  
is 800m, but limit is 1500m.
```

尝试创建一个不满足最小 CPU 请求的 Pod

这里给出了包含一个容器的 Pod 的配置文件。该容器声明了100 millicpu的 CPU 请求和800 millicpu的CPU 限制。

[admin/resource/cpu-constraints-pod-3.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: constraints-cpu-demo-3  
spec:  
  containers:  
    - name: constraints-cpu-demo-3-ctr  
      image: nginx  
      resources:  
        limits:  
          cpu: "800m"  
        requests:  
          cpu: "100m"
```

尝试创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-  
pod-3.yaml --namespace=constraints-cpu-example
```

输出结果显示 Pod 没有创建成功，因为容器声明的 CPU 请求太小了：

```
Error from server (Forbidden): error when creating "examples/admin/resource/  
cpu-constraints-pod-3.yaml":  
pods "constraints-cpu-demo-4" is forbidden: minimum cpu usage per Container  
is 200m, but request is 100m.
```

创建一个没有声明 CPU 请求和 CPU 限制的 Pod

这里给出了包含一个容器的 Pod 的配置文件。该容器没有设定 CPU 请求和 CPU 限制。

[admin/resource/cpu-constraints-pod-4.yaml](#)


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

```
metadata:  
  name: constraints-cpu-demo-4  
spec:  
  containers:  
    - name: constraints-cpu-demo-4-ctr  
      image: vish/stress
```

创建 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-4.yaml --namespace=constraints-cpu-example
```

查看 Pod 的详情 :

```
kubectl get pod constraints-cpu-demo-4 --namespace=constraints-cpu-example  
--output=yaml
```

输出结果显示 Pod 的容器有个 800 millicpu 的 CPU 请求和 800 millicpu 的 CPU 限制。 容器是怎样得到那些值的呢 ?

```
resources:  
  limits:  
    cpu: 800m  
  requests:  
    cpu: 800m
```

因为你的 Container 没有声明自己的 CPU 请求和限制 , LimitRange 给它指定了 [默认的 CPU 请求和限制](#)

此时 , 你的容器可能运行也可能没有运行。 回想一下 , 本任务的先决条件是你的节点要有 1 个 CPU。 如果你的每个节点仅有 1 个 CPU , 那么可能没有任何一个节点可以满足 800 millicpu 的 CPU 请求。 如果你在用的节点恰好有两个 CPU , 那么你才可能有足够的 CPU 来满足 800 millicpu 的请求。

```
kubectl delete pod constraints-cpu-demo-4 --namespace=constraints-cpu-example
```

CPU 最小和最大限制的强制执行

只有当 Pod 创建或者更新时 , LimitRange 为命名空间规定的 CPU 最小和最大限制才会被强制执行。 如果你对 LimitRange 进行修改 , 那不会影响此前创建的 Pod。

最小和最大 CPU 限制范围的动机

作为集群管理员 , 你可能想设定 Pod 可以使用的 CPU 资源限制。 例如 :

- 集群中的每个节点有两个 CPU。 你不想接受任何请求超过 2 个 CPU 的 Pod , 因为集群中没有节点可以支持这种请求。

- 你的生产和开发部门共享一个集群。你想允许生产工作负载消耗 3 个 CPU，而开发部门工作负载的消耗限制为 1 个 CPU。你可以为生产和开发创建不同的命名空间，并且为每个命名空间都应用 CPU 限制。

清理

删除你的命名空间：

```
kubectl delete namespace constraints-cpu-example
```

接下来

集群管理员参考：

- [为命名空间配置默认内存请求和限制](#)
- [为命名空间配置内存限制的最小值和最大值](#)
- [为命名空间配置 CPU 限制的最小值和最大值](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)

应用开发者参考：

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)
- [为 Pod 配置服务质量](#)

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 11, 2020 at 6:26 PM PST: [Update cpu-constraint-namespace.md \(415286ff1\)](#)

为命名空间配置内存和 CPU 配额

本文介绍怎样为命名空间设置容器可用的内存和 CPU 总量。你可以通过 [ResourceQuota](#) 对象设置配额.

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.

集群中每个节点至少有 1 GiB 的内存。

创建命名空间

创建一个命名空间，以便本练习中创建的资源和集群的其余部分相隔离。

```
kubectl create namespace quota-mem-cpu-example
```

创建 ResourceQuota

这里给出一个 ResourceQuota 对象的配置文件：

[admin/resource/quota-mem-cpu.yaml](#)



```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

创建 ResourceQuota

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu.yaml --namespace=quota-mem-cpu-example
```

查看 ResourceQuota 详情：

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=yaml
```

ResourceQuota 在 quota-mem-cpu-example 命名空间中设置了如下要求：

- 每个容器必须有内存请求和限制，以及 CPU 请求和限制。
- 所有容器的内存请求总和不能超过1 GiB。

- 所有容器的内存限制总和不能超过2 GiB。
- 所有容器的 CPU 请求总和不能超过1 cpu。
- 所有容器的 CPU 限制总和不能超过2 cpu。

创建 Pod

这里给出 Pod 的配置文件：

[admin/resource/quota-mem-cpu-pod.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo
spec:
  containers:
  - name: quota-mem-cpu-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
        cpu: "800m"
      requests:
        memory: "600Mi"
        cpu: "400m"
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-
pod.yaml --namespace=quota-mem-cpu-example
```

检查下 Pod 中的容器在运行：

```
kubectl get pod quota-mem-cpu-demo --namespace=quota-mem-cpu-example
```

再查看 ResourceQuota 的详情：

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-
example --output=yaml
```

输出结果显示了配额以及有多少配额已经被使用。你可以看到 Pod 的内存和 CPU 请求值及限制值没有超过配额。

```
status:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
```

```
requests.cpu: "1"
requests.memory: 1Gi
used:
limits.cpu: 800m
limits.memory: 800Mi
requests.cpu: 400m
requests.memory: 600Mi
```

尝试创建第二个 Pod

这里给出了第二个 Pod 的配置文件：

[admin/resource/quota-mem-cpu-pod-2.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo-2
spec:
  containers:
  - name: quota-mem-cpu-demo-2-ctr
    image: redis
    resources:
      limits:
        memory: "1Gi"
        cpu: "800m"
      requests:
        memory: "700Mi"
        cpu: "400m"
```

配置文件中，你可以看到 Pod 的内存请求为 700 MiB。请注意新的内存请求与已经使用的内存请求之和超过了内存请求的配额。 $600 \text{ MiB} + 700 \text{ MiB} > 1 \text{ GiB}$ 。

尝试创建 Pod：

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-pod-2.yaml --namespace=quota-mem-cpu-example
```

第二个 Pod 不能被创建成功。输出结果显示创建第二个 Pod 会导致内存请求总量超过内存请求配额。

```
Error from server (Forbidden): error when creating "examples/admin/resource/quota-mem-cpu-pod-2.yaml":
pods "quota-mem-cpu-demo-2" is forbidden: exceeded quota: mem-cpu-demo,
requested: requests.memory=700Mi, used: requests.memory=600Mi, limited:
requests.memory=1Gi
```

讨论

如你在本练习中所见，你可以用 ResourceQuota 限制命名空间中所有容器的内存请求总量。同样你也可以限制内存限制总量、CPU 请求总量、CPU 限制总量。

如果你想对单个容器而不是所有容器进行限制，就请使用 [LimitRange](#)。

清理

删除你的命名空间：

```
kubectl delete namespace quota-mem-cpu-example
```

接下来

集群管理员参考

- [为命名空间配置默认内存请求和限制](#)
- [为命名空间配置内存限制的最小值和最大值](#)
- [为命名空间配置 CPU 限制的最小值和最大值](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配CPU资源](#)
- [为 Pod 配置服务质量](#)

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 14, 2020 at 12:07 AM PST: [Update quota-memory-cpu-namespace.md \(e8cb50356\)](#)

配置命名空间下 Pod 配额

本文主要描述如何配置一个命名空间下可运行的 Pod 个数配额。你可以使用 [ResourceQuota](#) 对象来配置配额。

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.

创建一个命名空间

首先创建一个命名空间，这样可以将本次操作中创建的资源与集群其他资源隔离开来。

```
kubectl create namespace quota-pod-example
```

创建 ResourceQuota

下面是一个 ResourceQuota 的配置文件：

[admin/resource/quota-pod.yaml](#)



```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "2"
```

创建这个 ResourceQuota：

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod.yaml --namespace=quota-pod-example
```

查看资源配置的详细信息：

```
kubectl get resourcequota pod-demo --namespace=quota-pod-example --output=yaml
```

从输出的信息我们可以看到，该命名空间下 Pod 的配额是 2 个，目前创建的 Pod 数为 0，配额使用率为 0。

```
spec:
  hard:
    pods: "2"
```

```
status:  
  hard:  
    pods: "2"  
  used:  
    pods: "0"
```

下面是一个 Deployment 的配置文件：

[admin/resource/quota-pod-deployment.yaml](#)


```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: pod-quota-demo  
spec:  
  selector:  
    matchLabels:  
      purpose: quota-demo  
  replicas: 3  
  template:  
    metadata:  
      labels:  
        purpose: quota-demo  
    spec:  
      containers:  
      - name: pod-quota-demo  
        image: nginx
```

在配置文件中，replicas: 3 告诉 Kubernetes 尝试创建三个 Pods，且运行相同的应用。

创建这个 Deployment：

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod-deployment.yaml --namespace=quota-pod-example
```

查看 Deployment 的详细信息：

```
kubectl get deployment pod-quota-demo --namespace=quota-pod-example --output=yaml
```

从输出的信息我们可以看到，尽管尝试创建三个 Pod，但是由于配额的限制，只有两个 Pod 能被成功创建。

```
spec:  
  ...  
  replicas: 3  
  ...
```

```
status:  
  availableReplicas: 2  
...  
lastUpdateTime: 2017-07-07T20:57:05Z  
  message: 'unable to create pods: pods "pod-quota-demo-1650323038-" is  
forbidden:  
    exceeded quota: pod-demo, requested: pods=1, used: pods=2, limited: pods  
=2'
```

清理

删除命名空间：

```
kubectl delete namespace quota-pod-example
```

接下来

集群管理人员参考

- [为命名空间配置默认的内存请求和限制](#)
- [为命名空间配置默认的 CPU 请求和限制](#)
- [为命名空间配置内存的最小值和最大值约束](#)
- [为命名空间配置 CPU 的最小值和最大值约束](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为 API 对象的设置配额](#)

应用开发人员参考

- [为容器和 Pod 分配内存资源](#)
- [给容器和 Pod 分配 CPU 资源](#)
- [配置 Pod 的服务质量](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 15, 2020 at 11:30 AM PST: [\[zh\] Tidy up and fix links in tasks section \(7/10\) \(30423d108\)](#)

安装网络规则驱动

[使用 Calico 提供 NetworkPolicy](#)

[使用 Cilium 提供 NetworkPolicy](#)

[使用 kube-router 提供 NetworkPolicy](#)

[使用 Romana 提供 NetworkPolicy](#)

[使用 Weave Net 提供 NetworkPolicy](#)

使用 Calico 提供 NetworkPolicy

本页展示了几种在 Kubernetes 上快速创建 Calico 集群的方法。

准备开始

确定你想部署一个[云版本](#)还是[本地版本](#)的集群。

在 Google Kubernetes Engine (GKE) 上创建一个 Calico 集群

先决条件: [gcloud](#)

- 启动一个带有 Calico 的 GKE 集群，只需加上参数 --enable-network-policy。

语法

```
gcloud container clusters create [CLUSTER_NAME] --enable-network-policy
```

示例

```
gcloud container clusters create my-calico-cluster --enable-network-policy
```

- 使用如下命令验证部署是否正确。

```
kubectl get pods --namespace=kube-system
```

Calico 的 pods 名以 calico 打头，检查确认每个 pods 状态为 Running。

使用 kubeadm 创建一个本地 Calico 集群

使用 kubeadm 在 15 分钟内得到一个本地单主机 Calico 集群，请参考 [Calico 快速入门](#)。

接下来

集群运行后，您可以按照[声明网络策略](#)去尝试使用 Kubernetes NetworkPolicy。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 18, 2020 at 8:22 PM PST: [Update calico-network-policy.md \(de69cbc9e\)](#)

使用 Cilium 提供 NetworkPolicy

本页展示如何使用 Cilium 提供 NetworkPolicy。

关于 Cilium 的背景知识，请阅读 [Cilium 介绍](#)。

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.

在 Minikube 上部署 Cilium 用于基本测试

为了轻松熟悉 Cilium 你可以根据 [Cilium Kubernetes 入门指南](#) 在 minikube 中执行一个 cilium 的基本 DaemonSet 安装。

要启动 minikube，需要的最低版本为 1.3.1，使用下面的参数运行：

```
minikube version
```

```
minikube version: v1.3.1
```

```
minikube start --network-plugin=cni --memory=4096
```

挂载 BPF 文件系统：

```
minikube ssh -- sudo mount bpffs -t bpf /sys/fs/bpf
```

在 minikube 环境中，你可以部署下面的“一体化” YAML 文件，其中包含 Cilium 的 DaemonSet 配置以及适当的 RBAC 配置：

```
kubectl create -f https://raw.githubusercontent.com/cilium/cilium/v1.8/install/kubernetes/quick-install.yaml
```

```
configmap/cilium-config created
serviceaccount/cilium created
serviceaccount/cilium-operator created
clusterrole.rbac.authorization.k8s.io/cilium created
clusterrole.rbac.authorization.k8s.io/cilium-operator created
clusterrolebinding.rbac.authorization.k8s.io/cilium created
clusterrolebinding.rbac.authorization.k8s.io/cilium-operator created
daemonset.apps/cilium create
deployment.apps/cilium-operator created
```

入门指南其余的部分用一个示例应用说明了如何强制执行 L3/L4 (即 IP 地址+端口) 的安全策略 以及L7 (如 HTTP) 的安全策略。

部署 Cilium 用于生产用途

关于部署 Cilium 用于生产的详细说明，请见 [Cilium Kubernetes 安装指南](#) 此文档包括详细的需求、说明和生产用途 DaemonSet 文件示例。

了解 Cilium 组件

部署使用 Cilium 的集群会添加 Pods 到 kube-system 命名空间。要查看 Pod 列表，运行：

```
kubectl get pods --namespace=kube-system
```

你将看到像这样的 Pods 列表：

NAME	READY	STATUS	RESTARTS	AGE
cilium-6rxbd	1/1	Running	0	1m
...				

你的集群中的每个节点上都会运行一个 cilium Pod，通过使用 Linux BPF 针对该节点上的 Pod 的入站、出站流量实施网络策略控制。

接下来

集群运行后，你可以按照 [声明网络策略](#) 试用基于 Cilium 的 Kubernetes NetworkPolicy。 玩得开心，如果你有任何疑问，请到 [Cilium Slack 频道](#) 联系我们。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 13, 2021 at 11:39 AM PST: [Urls in correct. \(0046b7fa8\)](#)

使用 kube-router 提供 NetworkPolicy

本页展示如何使用 [Kube-router](#) 提供 NetworkPolicy。

准备开始

你需要拥有一个运行中的 Kubernetes 集群。如果你还没有集群，可以使用任意的集群安装程序如 Kops、Bootkube、Kubeadm 等创建一个。

安装 kube-router 插件

kube-router 插件自带一个网络策略控制器，监视来自于 Kubernetes API 服务器的 NetworkPolicy 和 Pod 的变化，根据策略指示配置 iptables 规则和 ipsets 来允许或阻止流量。请根据 [通过集群安装程序尝试 kube-router](#) 指南安装 kube-router 插件。

接下来

在你安装了 kube-router 插件后，可以参考 [声明网络策略](#) 去尝试使用 Kubernetes NetworkPolicy。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 16, 2020 at 11:05 AM PST: [\[zh\] Tidy up and fix links in tasks section \(8/10\) \(c2aae6890\)](#)

使用 Romana 提供 NetworkPolicy

本页展示如何使用 Romana 作为 NetworkPolicy。

准备开始

完成 [kubeadm 入门指南](#) 中的 1、2、3 步。

使用 kubeadm 安装 Romana

按照[容器化安装指南](#)， 使用 kubeadm 安装。

应用网络策略

使用以下的一种方式应用网络策略：

- [Romana 网络策略](#)
 - [Romana 网络策略例子](#)
- NetworkPolicy API

接下来

Romana 安装完成后，你可以按照 [声明网络策略](#) 去尝试使用 Kubernetes NetworkPolicy。

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 20, 2020 at 6:47 PM PST: [\[zh\] Sync changes from English site \(10\) \(f42b28f86\)](#)

使用 Weave Net 提供 NetworkPolicy

本页展示如何使用使用 Weave Net 提供 NetworkPolicy。

准备开始

你需要拥有一个 Kubernetes 集群。按照 [kubeadm 入门指南](#) 来启动一个。

安装 Weave Net 插件

按照[通过插件集成 Kubernetes](#) 指南执行安装。

Kubernetes 的 Weave Net 插件带有 [网络策略控制器](#)，可自动监控 Kubernetes 所有名字空间的 NetworkPolicy 注释，配置 iptables 规则以允许或阻止策略指示的流量。

测试安装

验证 weave 是否有效。

输入以下命令：

```
kubectl get po -n kube-system -o wide
```

输出类似这样：

NAME NODE	READY	STATUS	RESTARTS	AGE	IP
weave-net-1t1qg worknode3	2/2	Running	0	9d	192.168.2.10
weave-net-231d7 worknodegpu	2/2	Running	1	7d	10.2.0.17
weave-net-7nmwt 192.168.2.131 masternode	2/2	Running	3	9d	
weave-net-pmw8w 192.168.2.216 worknode2	2/2	Running	0	9d	

每个 Node 都有一个 weave Pod，所有 Pod 都是Running 和 2/2 READY。（2/2 表示每个 Pod 都有 weave 和 weave-npc）

接下来

安装 Weave Net 插件后，你可以参考 [声明网络策略](#) 来试用 Kubernetes NetworkPolicy。如果你有任何疑问，请通过 [Slack 上的 #weave-community 频道](#) 或者 [Weave 用户组](#) 联系我们。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

IP Masquerade Agent 用户指南

此页面展示如何配置和启用 ip-masq-agent。

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.

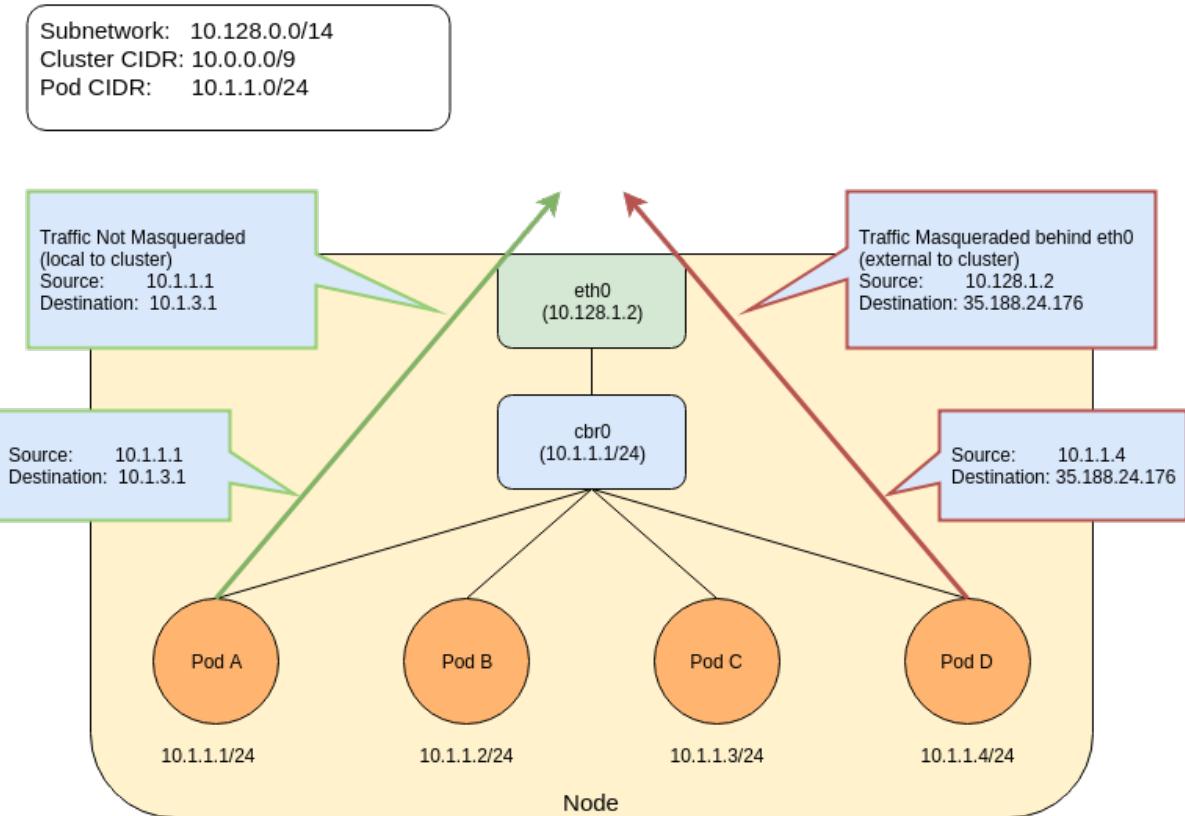
IP Masquerade Agent 用户指南

ip-masq-agent 配置 iptables 规则以隐藏位于集群节点 IP 地址后面的 Pod 的 IP 地址。这通常在将流量发送到集群的 Pod [CIDR](#) 范围之外的目的地时使用。

关键术语

- **NAT (网络地址转译)** 是一种通过修改 IP 地址头中的源和/或目标地址信息将一个 IP 地址重新映射 到另一个 IP 地址的方法。通常由执行 IP 路由的设备执行。
- **伪装 NAT** 的一种形式，通常用于执行多对一地址转换，其中多个源 IP 地址被隐藏在单个地址后面，该地址通常是执行 IP 路由的设备。在 Kubernetes 中，这是节点的 IP 地址。
- **CIDR (无类别域间路由)** 基于可变长度子网掩码，允许指定任意长度的前缀。CIDR 引入了一种新的 IP 地址表示方法，现在通常称为**CIDR表示法**，其中地址或路由前缀后添加一个后缀，用来表示前缀的位数，例如 192.168.2.0/24。
- **本地链路** 本地链路是仅对网段或主机所连接的广播域内的通信有效的网络地址。IPv4 的本地链路地址在 CIDR 表示法的地址块 169.254.0.0/16 中定义。

ip-masq-agent 配置 iptables 规则，以便在将流量发送到集群节点的 IP 和集群 IP 范围之外的目标时 处理伪装节点或 Pod 的 IP 地址。这本质上隐藏了集群节点 IP 地址后面的 Pod IP 地址。在某些环境中，去往“外部”地址的流量必须从已知的机器地址发出。例如，在 Google Cloud 中，任何到互联网的流量都必须来自 VM 的 IP。使用容器时，如 Google Kubernetes Engine，从 Pod IP 发出的流量将被拒绝出站。为了避免这种情况，我们必须将 Pod IP 隐藏在 VM 自己的 IP 地址后面 - 通常称为“伪装”。默认情况下，代理配置为将 [RFC 1918](#) 指定的三个私有 IP 范围视为非伪装 [CIDR](#)。这些范围是 10.0.0.0/8, 172.16.0.0/12 和 192.168.0.0/16。默认情况下，代理还将链路本地地址（169.254.0.0/16）视为非伪装 CIDR。代理程序配置为每隔 60 秒从 /etc/config/ip-masq-agent 重新加载其配置，这也是可修改的。



代理配置文件必须使用 YAML 或 JSON 语法编写，并且可能包含三个可选值：

- **nonMasqueradeCIDRs:** [CIDR](#) 表示法中的字符串列表，用于指定不需伪装的地址范围。
- **masqLinkLocal:** 布尔值 (true / false)，表示是否将流量伪装到本地链路前缀 169.254.0.0/16。默认为 false。
- **resyncInterval:** 代理尝试从磁盘重新加载配置的时间间隔。例如 '30s'，其中 's' 是秒，'ms' 是毫秒等...

10.0.0.0/8、172.16.0.0/12 和 192.168.0.0/16 范围内的流量不会被伪装。任何其他流量（假设是互联网）将被伪装。Pod 访问本地目的地的例子，可以是其节点的 IP 地址、另一节点的地址或集群的 IP 地址范围内的一个 IP 地址。默认情况下，任何其他流量都将伪装。以下条目展示了 ip-masq-agent 的默认使用的规则：

```
iptables -t nat -L IP-MASQ-AGENT
RETURN  all -- anywhere      169.254.0.0/16    /* ip-masq-agent: cluster-
local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-
type !LOCAL
RETURN  all -- anywhere      10.0.0.0/8       /* ip-masq-agent: cluster-
local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-
type !LOCAL
RETURN  all -- anywhere      172.16.0.0/12     /* ip-masq-agent: cluster-
local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-
type !LOCAL
```

```
RETURN all -- anywhere 192.168.0.0/16 /* ip-masq-agent: cluster-local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-type !LOCAL  
MASQUERADE all -- anywhere anywhere /* ip-masq-agent: outbound traffic should be subject to MASQUERADE (this match must come after cluster-local CIDR matches) */ ADDRTYPE match dst-type !LOCAL
```

默认情况下，从 Kubernetes 1.7.0 版本开始的 GCE/Google Kubernetes Engine 中，如果启用了网络策略，或者你使用的集群 CIDR 不在 10.0.0.0/8 范围内，则 ip-masq-agent 将在你的集群中运行。如果你在其他环境中运行，则可以将 ip-masq-agent [DaemonSet](#) 添加到你的集群：

创建 ip-masq-agent

通过运行以下 kubectl 指令创建 ip-masq-agent:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/ip-masq-agent/master/ip-masq-agent.yaml
```

你必须同时将适当的节点标签应用于集群中希望代理运行的任何节点。

```
kubectl label nodes my-node beta.kubernetes.io/masq-agent-ds-ready=true
```

更多信息可以通过 ip-masq-agent 文档 [这里](#) 找到。

在大多数情况下，默认的规则集应该足够；但是，如果你的群集不是这种情况，则可以创建并应用 [ConfigMap](#) 来自定义受影响的 IP 范围。例如，要允许 ip-masq-agent 仅作用于 10.0.0.0/8，你可以在一个名为 "config" 的文件中创建以下 [ConfigMap](#)。

说明：

重要的是，该文件之所以被称为 config，因为默认情况下，该文件将被用作 ip-masq-agent 查找的主键：

```
nonMasqueradeCIDRs:  
  - 10.0.0.0/8  
resyncInterval: 60s
```

运行以下命令将配置映射添加到你的集群：

```
kubectl create configmap ip-masq-agent --from-file=config --namespace=kube-system
```

这将更新位于 `/etc/config/ip-masq-agent` 的一个文件，该文件以 `resyncInterval` 为周期定期检查并应用于集群节点。重新同步间隔到期后，你应该看到你的更改在 iptables 规则中体现：

```
iptables -t nat -L IP-MASQ-AGENT  
Chain IP-MASQ-AGENT (1 references)
```

```
target  prot opt source          destination
RETURN  all  --  anywhere      169.254.0.0/16    /* ip-masq-agent: cluster-
local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-
type !LOCAL
RETURN  all  --  anywhere      10.0.0.0/8       /* ip-masq-agent: cluster-
local
MASQUERADE all  --  anywhere      anywhere        /* ip-masq-agent:
outbound traffic should be subject to MASQUERADE (this match must come
after cluster-local CIDR matches) */ ADDRTYPE match dst-type !LOCAL
```

默认情况下，本地链路范围 (169.254.0.0/16) 也由 ip-masq agent 处理，该代理设置适当的 iptables 规则。要使 ip-masq-agent 忽略本地链路，可以在配置映射中将 *masqLinkLocal* 设置为 true。

```
nonMasqueradeCIDRs:
```

```
- 10.0.0.0/8
```

```
resyncInterval: 60s
```

```
masqLinkLocal: true
```

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 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

Kubernetes 云管理控制器

FEATURE STATE: Kubernetes v1.11 [beta]

由于云驱动的开发和发布的步调与 Kubernetes 项目不同，将服务提供商专用代码抽象到 [cloud-controller-manager](#) 二进制中有助于云服务厂商在 Kubernetes 核心代码之外独立进行开发。

cloud-controller-manager 可以被链接到任何满足 [cloudprovider.Interface](#) 约束的云服务提供商。为了兼容旧版本，Kubernetes 核心项目中提供的 [cloud-controller-manager](#) 使用和 kube-controller-manager 相同的云服务类库。已经在 Kubernetes 核心项目中支持的云服务提供商预计将来通过使用 in-tree 的 cloud-controller-manager 过渡为非 Kubernetes 核心代码。

管理

需求

每个云服务都有一套各自的需求用于系统平台的集成，这不应与运行 kube-controller-manager 的需求有太大差异。作为经验法则，你需要：

- 云服务认证/授权：你的云服务可能需要使用令牌或者 IAM 规则以允许对其 API 的访问
- kubernetes 认证/授权：cloud-controller-manager 可能需要 RBAC 规则以访问 kubernetes apiserver
- 高可用：类似于 kube-controller-manager，你可能希望通过主节点选举（默认开启）配置一个高可用的云管理控制器。

运行云管理控制器

你需要对集群配置做适当的修改以成功地运行云管理控制器：

- 一定不要为 kube-apiserver 和 kube-controller-manager 指定 --cloud-provider 标志。这将保证它们不会运行任何云服务专用循环逻辑，这将会由云管理控制器运行。未来这个标记将被废弃并去除。
- kubelet 必须使用 --cloud-provider=external 运行。这是为了保证让 kubelet 知道在执行任何任务前，它必须被云管理控制器初始化。

请记住，设置群集使用云管理控制器将用多种方式更改群集行为：

- 指定了 --cloud-provider=external 的 kubelet 将被添加一个 node.cloudprovider.kubernetes.io/uninitialized 的污点，导致其在初始化过程中不可调度（NoSchedule）。这将标记该节点在能够正常调度前，需要外部的控制器进行二次初始化。请注意，如果云管理控制器不可用，集群中的新节点会一直处于不可调度的状态。这个污点很重要，因为调度器可能需要关于节点的云服务特定的信息，比如他们的区域或类型（高端 CPU、GPU 支持、内存较大、临时实例等）。
- 集群中节点的云服务信息将不再能够从本地元数据中获取，取而代之的是所有获取节点信息的 API 调用都将通过云管理控制器。这意味着你可以通过限制到 kubelet 云服务 API 的访问来提升安全性。在更大的集群中你可能需要考虑云管理控制器是否会遇到速率限制，因为它现在负责集群中几乎所有到云服务的 API 调用。

云管理控制器可以实现：

- 节点控制器 - 负责使用云服务 API 更新 kubernetes 节点并删除在云服务上已经删除的 kubernetes 节点。
- 服务控制器 - 负责在云服务上为类型为 LoadBalancer 的 service 提供负载均衡器。
- 路由控制器 - 负责在云服务上配置网络路由。
- 如果你使用的是 out-of-tree 提供商，请按需实现其余任意特性。

示例

如果当前 Kubernetes 内核支持你使用的云服务，并且想要采用云管理控制器，请参见 [kubernetes 内核中的云管理控制器](#)。

对于不在 Kubernetes 核心代码库中的云管理控制器，你可以在云服务厂商或 SIG 领导者的源中找到对应的项目。

- [DigitalOcean](#)
- [keepalived](#)
- [Oracle Cloud Infrastructure](#)
- [Rancher](#)

对于已经存在于 Kubernetes 内核中的提供商，你可以在集群中将 in-tree 云管理控制器作为守护进程运行。请使用如下指南：

[admin/cloud/ccm-example.yaml](#)



```
# This is an example of how to setup cloud-controller-manger as a Daemonset in
your cluster.
# It assumes that your masters can run pods and has the role node-
role.kubernetes.io/master
# Note that this Daemonset will not work straight out of the box for your cloud,
this is
# meant to be a guideline.

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
    metadata:
      labels:
        k8s-app: cloud-controller-manager
    spec:
      serviceAccountName: cloud-controller-manager
      containers:
        - name: cloud-controller-manager
          # for in-tree providers we use k8s.gcr.io/cloud-controller-manager
          # this can be replaced with any other image for out-of-tree providers
          image: k8s.gcr.io/cloud-controller-manager:v1.8.0
          command:
            - /usr/local/bin/cloud-controller-manager
            - --cloud-provider=[YOUR_CLOUD_PROVIDER] # Add your own cloud provider here!
              - --leader-elect=true
              - --use-service-account-credentials
              # these flags will vary for every cloud provider
              - --allocate-node-cidrs=true
              - --configure-cloud-routes=true
              - --cluster-cidr=172.17.0.0/16
      tolerations:
        # this is required so CCM can bootstrap itself
        - key: node.cloudprovider.kubernetes.io/uninitialized
          value: "true"
          effect: NoSchedule
        # this is to have the daemonset runnable on master nodes
        # the taint may vary depending on your cluster setup
        - key: node-role.kubernetes.io/master
          value: "true"
          effect: NoSchedule
        # this is to restrict CCM to only run on master nodes
        # the node selector may vary depending on your cluster setup
      nodeSelector:
        node-role.kubernetes.io/master: ""
```

限制

运行云管理控制器会有一些可能的限制。虽然以后的版本将处理这些限制，但是知道这些生产负载的限制很重要。

对 Volume 的支持

云管理控制器未实现 kube-controller-manager 中的任何 volume 控制器，因为和 volume 的集成还需要与 kubelet 协作。由于我们引入了 CSI (容器存储接口，container storage interface) 并对弹性 volume 插件添加了更强大的支持，云管理控制器将添加必要的支持，以使云服务同 volume 更好的集成。请在 [这里](#) 了解更多关于 out-of-tree CSI volume 插件的信息。

可扩展性

在以前为云服务提供商提供的架构中，我们依赖 kubelet 的本地元数据服务来获取关于它本身的节点信息。通过这个新的架构，现在我们完全依赖云管理控制器来获取所有节点的信息。对于非常大的集群，你需要考虑可能的瓶颈，例如资源需求和 API 速率限制。

鸡和蛋的问题

云管理控制器的目标是将云服务特性的开发从 Kubernetes 核心项目中解耦。不幸的是，Kubernetes 项目的许多方面都假设云服务提供商的特性同项目紧密结合。因此，这种新架构的采用可能导致某些场景下，当一个请求需要从云服务提供商获取信息时，在该请求没有完成的情况下云管理控制器不能返回那些信息。

Kubelet 中的 TLS 引导特性是一个很好的例子。目前，TLS 引导认为 kubelet 有能力从云提供商（或本地元数据服务）获取所有的地址类型（私有、公用等），但在被初始化之前，云管理控制器不能设置节点地址类型，而这需要 kubelet 拥有 TLS 证书以和 API 服务器通信。

随着整个动议的演进，将来的发行版中将作出改变来解决这些问题。

接下来

要构建和开发你自己的云管理控制器，请阅读 [开发云管理控制器](#) 文档。

反馈

此页是否对您有帮助？

是 否

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

最后修改 October 30, 2020 at 6:00 PM PST: [Update running-cloud-controller.md \(09d0779ed\)](#)

为 Kubernetes 运行 etcd 集群

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

您的 Kubernetes 集群的 etcd 数据库通常需要有个备份计划。

要了解 etcd 更深层次的信息，请参考 [etcd 文档](#)。

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.

先决条件

- 运行的 etcd 集群个数成员为奇数。
- etcd 是一个 leader-based 分布式系统。确保主节点定期向所有从节点发送心跳，以保持集群稳定。
- 确保不发生资源不足。

集群的性能和稳定性对网络和磁盘 IO 非常敏感。任何资源匮乏都会导致心跳超时，从而导致集群的不稳定。不稳定的情况表明没有选出任何主节点。在这种情况下，集群不能对其当前状态进行任何更改，这意味着不能调度新的 pod。

- 保持稳定的 etcd 集群对 Kubernetes 集群的稳定性至关重要。因此，请在专用机器或隔离环境上运行 etcd 集群，以满足[所需资源需求](#)。
- 在生产中运行的 etcd 的最低推荐版本是 3.2.10+。

资源要求

使用有限的资源运行 etcd 只适合测试目的。为了在生产中部署，需要先进的硬件配置。在生产中部署 etcd 之前，请查看[所需资源参考文档](#)。

启动 etcd 集群

本节介绍如何启动单节点和多节点 etcd 集群。

单节点 etcd 集群

只为测试目的使用单节点 etcd 集群。

1. 运行以下命令：

```
./etcd --listen-client-urls=http://$PRIVATE_IP:2379 --advertise-client-urls=http://$PRIVATE_IP:2379
```

2. 使用参数 `--etcd-servers=$PRIVATE_IP:2379` 启动 Kubernetes API 服务器。

使用您 etcd 客户端 IP 替换 PRIVATE_IP。

多节点 etcd 集群

为了耐用性和高可用性，在生产中将以多节点集群的方式运行 etcd，并且定期备份。建议在生产中使用五个成员的集群。有关该内容的更多信息，请参阅[常见问题文档](#)。

可以通过静态成员信息或动态发现的方式配置 etcd 集群。有关集群的详细信息，请参阅[etcd 集群文档](#)。

例如，考虑运行以下客户端 URL 的五个成员的 etcd 集群：`http://$IP1:2379`，`http://$IP2:2379`，`http://$IP3:2379`，`http://$IP4:2379` 和 `http://$IP5:2379`。要启动 Kubernetes API 服务器：

1. 运行以下命令：

```
./etcd --listen-client-urls=http://$IP1:2379, http://$IP2:2379, http://$IP3:2379, http://$IP4:2379, http://$IP5:2379 --advertise-client-urls=http://$IP1:2379, http://$IP2:2379, http://$IP3:2379, http://$IP4:2379, http://$IP5:2379
```

2. 使用参数 `--etcd-servers=$IP1:2379, $IP2:2379, $IP3:2379, $IP4:2379, $IP5:2379` 启动 Kubernetes API 服务器。

使用您 etcd 客户端 IP 地址替换 IP。

使用负载均衡的多节点 etcd 集群

要运行负载均衡的 etcd 集群：

1. 建立一个 etcd 集群。
2. 在 etcd 集群前面配置负载均衡器。例如，让负载均衡器的地址为 `$LB`。
3. 使用参数 `--etcd-servers=$LB:2379` 启动 Kubernetes API 服务器。

安全的 etcd 集群

对 etcd 的访问相当于集群中的 root 权限，因此理想情况下只有 API 服务器才能访问它。考虑到数据的敏感性，建议只向需要访问 etcd 集群的节点授予权限。

想要确保 etcd 的安全，可以设置防火墙规则或使用 etcd 提供的安全特性，这些安全特性依赖于 x509 公钥基础设施（PKI）。首先，通过生成密钥和证书来建立安全的通信通道。例如，使用密钥对 peer.key 和 peer.cert 来保护 etcd 成员之间的通信，而 client.cert 和 client.key 用于保护 etcd 与其客户端之间的通信。请参阅 etcd 项目提供的[示例脚本](#)，以生成用于客户端身份验证的密钥对和 CA 文件。

安全通信

若要使用安全对等通信对 etcd 进行配置，请指定参数 --peer-key-file=peer.key 和 --peer-cert-file=peer.cert，并使用 https 作为 URL 模式。

类似地，要使用安全客户端通信对 etcd 进行配置，请指定参数 --key-file=k8sclient.key 和 --cert-file=k8sclient.cert，并使用 https 作为 URL 模式。

限制 etcd 集群的访问

配置安全通信后，将 etcd 集群的访问限制在 Kubernetes API 服务器上。使用 TLS 身份验证来完成此任务。

例如，考虑由 CA etcd.ca 信任的密钥对 k8sclient.key 和 k8sclient.cert。当 etcd 配置为 --client-cert-auth 和 TLS 时，它使用系统 CA 或由 --trusted-ca-file 参数传入的 CA 验证来自客户端的证书。指定参数 --client-cert-auth=true 和 --trusted-ca-file=etcd.ca 将限制对具有证书 k8sclient.cert 的客户端的访问。

一旦正确配置了 etcd，只有具有有效证书的客户端才能访问它。要让 Kubernetes API 服务器访问，可以使用参数 --etcd-certfile=k8sclient.cert,--etcd-keyfile=k8sclient.key 和 --etcd-cafile=ca.cert 配置它。

说明： Kubernetes 目前不支持 etcd 身份验证。想要了解更多信息，请参阅相关的问题[支持 etcd v2 的基本认证](#)。

替换失败的 etcd 成员

etcd 集群通过容忍少数成员故障实现高可用性。但是，要改善集群的整体健康状况，请立即替换失败的成员。当多个成员失败时，逐个替换它们。替换失败成员需要两个步骤：删除失败成员和添加新成员。

虽然 etcd 在内部保留唯一的成员 ID，但建议为每个成员使用唯一的名称，以避免人为错误。例如，考虑一个三成员的 etcd 集群。让 URL 为：member1=http://10.0.0.1，member2=http://10.0.0.2 和 member3=http://10.0.0.3。当 member1 失败时，将其替换为 member4=http://10.0.0.4。

1. 获取失败的 member1 的成员 ID：

```
etcdctl --endpoints=http://10.0.0.2,http://10.0.0.3 member list
```

显示以下信息：

```
8211f1d0f64f3269, started, member1, http://10.0.0.1:2380, http://  
10.0.0.1:2379  
91bc3c398fb3c146, started, member2, http://10.0.0.2:2380, http://  
10.0.0.2:2379  
fd422379fda50e48, started, member3, http://10.0.0.3:2380, http://  
10.0.0.3:2379
```

2. 移除失败的成员

```
etcdctl member remove 8211f1d0f64f3269
```

显示以下信息：

```
Removed member 8211f1d0f64f3269 from cluster
```

3. 增加新成员：

```
./etcdctl member add member4 --peer-urls=http://10.0.0.4:2380
```

显示以下信息：

```
Member 2be1eb8f84b7f63e added to cluster ef37ad9dc622a7c4
```

4. 在 IP 为 10.0.0.4 的机器上启动新增加的成员：

```
export ETCD_NAME="member4"  
export ETCD_INITIAL_CLUSTER="member2=http://  
10.0.0.2:2380,member3=http://10.0.0.3:2380,member4=http://10.0.0.4:2380"  
export ETCD_INITIAL_CLUSTER_STATE=existing  
etcd [flags]
```

5. 做以下事情之一：

1. 更新其 `--etcd-servers` 参数，使 Kubernetes 知道配置进行了更改，然后重新启动 Kubernetes API 服务器。
2. 如果在 deployment 中使用了负载均衡，更新负载均衡配置。

有关集群重新配置的详细信息，请参阅 [etcd 重构文档](#)。

备份 etcd 集群

所有 Kubernetes 对象都存储在 etcd 上。定期备份 etcd 集群数据对于在灾难场景（例如丢失所有主节点）下恢复 Kubernetes 集群非常重要。快照文件包含所有 Kubernetes 状态和关键信息。为了保证敏感的 Kubernetes 数据的安全，可以对快照文件进行加密。

备份 etcd 集群可以通过两种方式完成：etcd 内置快照和卷快照。

内置快照

etcd 支持内置快照，因此备份 etcd 集群很容易。快照可以从使用 etcdctl snapshot save 命令的活动成员中获取，也可以通过从 etcd [数据目录复制](#) member/snap/db 文件，该 etcd 数据目录目前没有被 etcd 进程使用。获取快照通常不会影响成员的性能。

下面是一个示例，用于获取 \$ENDPOINT 所提供的键空间的快照到文件 snapshotdb：

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshotdb
# exit 0

# verify the snapshot
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshotdb
+-----+-----+-----+-----+
| HASH | REVISION | TOTAL KEYS | TOTAL SIZE |
+-----+-----+-----+-----+
| fe01cf57 |    10 |      7 | 2.1 MB   |
+-----+-----+-----+-----+
```

卷快照

如果 etcd 运行在支持备份的存储卷（如 Amazon Elastic Block 存储）上，则可以通过获取存储卷的快照来备份 etcd 数据。

扩展 etcd 集群

通过交换性能，扩展 etcd 集群可以提高可用性。缩放不会提高集群性能和能力。一般情况下不要扩大或缩小 etcd 集群的集合。不要为 etcd 集群配置任何自动缩放组。强烈建议始终在任何官方支持的规模上运行生产 Kubernetes 集群时使用静态的五成员 etcd 集群。

合理的扩展是在需要更高可靠性的情况下，将三成员集群升级为五成员集群。请参阅 [etcd 重新配置文档](#) 以了解如何将成员添加到现有集群中的信息。

恢复 etcd 集群

etcd 支持从 [major.minor](#) 或其他不同 patch 版本的 etcd 进程中获取的快照进行恢复。还原操作用于恢复失败的集群的数据。

在启动还原操作之前，必须有一个快照文件。它可以是来自以前备份操作的快照文件，也可以是来自剩余[数据目录](#)的快照文件。有关从快照文件还原集群的详细信息和示例，请参阅 [etcd 灾难恢复文档](#)。

如果还原的集群的访问 URL 与前一个集群不同，则必须相应地重新配置 Kubernetes API 服务器。在本例中，使用参数 `--etcd-servers=$NEW_ETCD_CLUSTER` 而不是参数 `--etcd-servers=$OLD_ETCD_CLUSTER` 重新启动 Kubernetes API 服务器。用相应的 IP 地址替换 `$NEW_ETCD_CLUSTER` 和 `$OLD_ETCD_CLUSTER`。如果在 etcd 集群前面使用负载平衡，则可能需要更新负载均衡器。

如果大多数 etcd 成员永久失败，则认为 etcd 集群失败。在这种情况下，Kubernetes 不能对其当前状态进行任何更改。虽然已调度的 pod 可能继续运行，但新的 pod 无法调度。在这种情况下，恢复 etcd 集群并可能需要重新配置 Kubernetes API 服务器以修复问题。

说明：

如果集群中正在运行任何 API 服务器，则不应尝试还原 etcd 的实例。相反，请按照以下步骤还原 etcd：

- 停止所有 kube-apiserver 实例
- 在所有 etcd 实例中恢复状态
- 重启所有 kube-apiserver 实例

我们还建议重启所有组件（例如 kube-scheduler、kube-controller-manager、kubelet），以确保它们不会依赖一些过时的数据。请注意，实际中还原会花费一些时间。在还原过程中，关键组件将丢失领导锁并自行重启。

说明：

为系统守护进程预留计算资源

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

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

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](#)

您的 Kubernetes 服务器版本必须不低于版本 1.8. 要获知版本信息，请输入 kubectl version.

您的 kubernetes 服务器版本必须至少是 1.17 版本，才能使用 kubelet 命令行选项 --reserved-cpus 设置 [显式预留 CPU 列表](#)。

节点可分配

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

可分配的节点暴露为 API 中 v1.Node 对象的一部分，也是 CLI 中 kubectl describe node 的一部分。

在 kubelet 中，可以为两类系统守护进程预留资源。

启用 QoS 和 Pod 级别的 cgroups

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

配置 cgroup 驱动

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

支持的参数值如下：

- cgroupfs 是默认的驱动，在主机上直接操作 cgroup 文件系统以对 cgroup 沙箱进行管理。

- `systemd` 是可选的驱动，使用 init 系统支持的资源的瞬时切片管理 cgroup 沙箱。

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

Kube 预留值

- **Kubelet 标志:** `--kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet 标志:** `--kube-reserved-cgroup=`

`kube-reserved` 用来给诸如 `kubelet`、容器运行时、节点问题监测器等 kubernetes 系统守护进程记述其资源预留值。该配置并非用来给以 Pod 形式运行的系统守护进程保留资源。`kube-reserved` 通常是节点上 pod 密度的函数。

除了 `cpu`，内存和 `ephemeral-storage` 之外，`pid` 可用来指定为 kubernetes 系统守护进程预留指定数量的进程 ID。

要选择性地对系统守护进程上执行 `kube-reserved` 保护，需要把 `kubelet` 的 `--kube-reserved-cgroup` 标志的值设置为 `kube` 守护进程的父控制组。

推荐将 kubernetes 系统守护进程放置于顶级控制组之下（例如 `systemd` 机器上的 `runtime.slice`）。理想情况下每个系统守护进程都应该在其自己的子控制组中运行。请参考[这篇文档](#)，进一步了解关于推荐控制组层次结构的细节。

请注意，如果 `--kube-reserved-cgroup` 不存在，`Kubelet` 将 **不会** 创建它。如果指定了一个无效的 cgroup，`Kubelet` 将会失败。

系统预留值

- **Kubelet 标志:** `--system-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet 标志:** `--system-reserved-cgroup=`

`system-reserved` 用于为诸如 `sshd`、`udev` 等系统守护进程记述其资源预留值。`system-reserved` 也应该为 `kernel` 预留内存，因为目前 `kernel` 使用的内存并不记在 Kubernetes 的 Pod 上。同时还推荐为用户登录会话预留资源（`systemd` 体系中的 `user.slice`）。

除了 `cpu`，内存和 `ephemeral-storage` 之外，`pid` 可用来指定为 kubernetes 系统守护进程预留指定数量的进程 ID。

要想为系统守护进程上可选地实施 `system-reserved` 约束，请指定 `kubelet` 的 `--system-reserved-cgroup` 标志值为 OS 系统守护进程的父级控制组。

推荐将 OS 系统守护进程放在一个顶级控制组之下（例如 `systemd` 机器上的 `system.slice`）。

请注意，如果 `--system-reserved-cgroup` 不存在，Kubelet 不会 创建它。如果指定了无效的 cgroup，Kubelet 将会失败。

显式保留的 CPU 列表

FEATURE STATE: Kubernetes v1.17 [stable]

- **Kubelet 标志:** `--reserved-cpus=0-3`

`reserved-cpus` 旨在为操作系统守护程序和 kubernetes 系统守护程序定义一个显式 CPU 集合。`reserved-cpus` 适用于不打算针对 cpuset 资源为操作系统守护程序和 kubernetes 系统守护程序定义独立的顶级 cgroups 的系统。如果 Kubelet 没有 指定参数 `--system-reserved-cgroup` 和 `--kube-reserved-cgroup`，则 `reserved-cpus` 提供的显式 cpuset 将优先于 `--kube-reserved` 和 `--system-reserved` 选项定义的 cpuset。

此选项是专门为电信/NFV 用例设计的，在这些用例中不受控制的中断或计时器可能会影响其工作负载性能。你可以使用此选项为系统或 kubernetes 守护程序以及中断或计时器显式定义 cpuset，这样系统上的其余 CPU 可以专门用于工作负载，因不受控制的中断或计时器的影响得以降低。要将系统守护程序、kubernetes 守护程序和中断或计时器移动到此选项定义的显式 cpuset 上，应使用 Kubernetes 之外的其他机制。例如：在 Centos 系统中，可以使用 tuned 工具集来执行此操作。

驱逐阈值

- **Kubelet 标志:** `--eviction-hard=[memory.available<500Mi]`

节点级别的内存压力将导致系统内存不足，这将影响到整个节点及其上运行的所有 Pod。节点可以暂时离线直到内存已经回收为止。为了防止（或减少可能性）系统内存不足，kubelet 提供了 [资源不足](#) 管理。驱逐操作只支持 memory 和 ephemeral-storage。通过 `--eviction-hard` 标志预留一些内存后，当节点上的可用内存降至保留值以下时，kubelet 将尝试驱逐 Pod。如果节点上不存在系统守护进程，Pod 将不能使用超过 `capacity-eviction-hard` 所 指定的资源量。因此，为驱逐而预留的资源对 Pod 是不可用的。

实施节点可分配约束

- **Kubelet 标志:** `--enforce-node-allocatable=pods[,][system-reserved][,][kube-reserved]`

调度器将 Allocatable 视为 Pod 可用的 capacity (资源容量)。

kubelet 默认对 Pod 执行 Allocatable 约束。无论何时，如果所有 Pod 的总用量超过了 Allocatable，驱逐 Pod 的措施将被执行。有关驱逐策略的更多细节可以在 [这里](#) 找到。可通过设置 kubelet `--enforce-node-allocatable` 标志值为 pods 控制这个措施。

可选地，通过在同一标志中同时指定 `kube-reserved` 和 `system-reserved` 值，可以使 kubelet 强制实施 `kube-reserved` 和 `system-reserved` 约束。请注意，要想执行 `kube-reserved` 或者 `system-reserved` 约束，需要对应设置 `--kube-reserved-cgroup` 或者 `--system-reserved-cgroup`。

一般原则

系统守护进程一般会被按照类似 Guaranteed Pod 一样对待。 系统守护进程可以在与其对应的控制组中出现突发资源用量，这一行为要作为 kubernetes 部署的一部分进行管理。 例如，kubelet 应该有它自己的控制组并和容器运行时共享 Kube-reserved 资源。 不过，如果执行了 kube-reserved 约束，则 kubelet 不可出现突发负载并用光 节点的所有可用资源。

在执行 system-reserved 预留策略时请加倍小心，因为它可能导致节点上的 关键系统服务出现 CPU 资源短缺、因为内存不足而被终止或者无法在节点上创建进程。 建议只有当用户详尽地描述了他们的节点以得出精确的估计值，并且对该组中进程因内存不足而被杀死时，有足够的信心将其恢复时，才可以强制执行 system-reserved 策略。

- 作为起步，可以先针对 pods 上执行 Allocatable 约束。
- 一旦用于追踪系统守护进程的监控和告警的机制到位，可尝试基于用量估计的方式 执行 kube-reserved 策略。
- 随着时间推进，在绝对必要的时候可以执行 system-reserved 策略。

随着时间推进和越来越多特性被加入，kube 系统守护进程对资源的需求可能也会增加。 以后 kubernetes 项目将尝试减少对节点系统守护进程的利用，但目前这件事的优先级并不是最高。 所以，将来的发布版本中 Allocatable 容量是有可能降低的。

示例场景

这是一个用于说明节点可分配 (Node Allocatable) 计算方式的示例：

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

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

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

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 07, 2020 at 10:11 AM PST: [Update reserve-compute-resources.md \(953e4ecf8\)](#)

为节点发布扩展资源

本文展示了如何为节点指定扩展资源（Extended Resource）。扩展资源允许集群管理员发布节点级别的资源，这些资源在不进行发布的情况下无法被 Kubernetes 感知。

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.

获取你的节点名称

```
kubectl get nodes
```

选择一个节点用于此练习。

在你的一个节点上发布一种新的扩展资源

为在一个节点上发布一种新的扩展资源，需要发送一个 HTTP PATCH 请求到 Kubernetes API server。例如：假设你的一个节点上带有四个 dongle 资源。下面是一个 PATCH 请求的示例，该请求为你的节点发布四个 dongle 资源。

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080
```

```
[
  {
    "op": "add",
    "path": "/status/capacity/example.com~1dongle",
    "value": "4"
  }
]
```

注意：Kubernetes 不需要了解 dongle 资源的含义和用途。前面的 PATCH 请求仅仅告诉 Kubernetes 你的节点拥有四个你称之为 dongle 的东西。

启动一个代理（ proxy ），以便你可以很容易地向 Kubernetes API server 发送请求：

```
kubectl proxy
```

在另一个命令窗口中，发送 HTTP PATCH 请求。用你的节点名称替换 <your-node-name>：

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/example.com~1dongle", "value": "4"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

说明：在前面的请求中，~1 为 patch 路径中 "/" 符号的编码。JSON-Patch 中的操作路径值被解析为 JSON 指针。更多细节，请查看 [IETF RFC 6901](#) 的第 3 节。

输出显示该节点的 dongle 资源容量（ capacity ）为 4：

```
"capacity": {
  "cpu": "2",
  "memory": "2049008Ki",
  "example.com/dongle": "4",
```

描述你的节点：

```
kubectl describe node <your-node-name>
```

输出再次展示了 dongle 资源：

```
Capacity:
cpu: 2
memory: 2049008Ki
example.com/dongle: 4
```

现在，应用开发者可以创建请求一定数量 dongle 资源的 Pod 了。参见[将扩展资源分配给容器](#)。

讨论

扩展资源类似于内存和 CPU 资源。例如，正如一个节点具有一定数量的内存和 CPU 资源，它们被节点上运行的所有组件共享，该节点也可以具有一定数量的 dongle 资源，这些资源同样被节点上运行的所有组件共享。此外，正如应用开发者可以创建请求一定数量的内存和 CPU 资源的 Pod，他们也可以创建请求一定数量 dongle 资源的 Pod。

扩展资源对 Kubernetes 是不透明的。Kubernetes 不知道扩展资源含义相关的任何信息。Kubernetes 只了解一个节点拥有一定数量的扩展资源。扩展资源必须以整形数量进行发布。例如，一个节点可以发布 4 个 `dongle` 资源，但是不能发布 4.5 个。

存储示例

假设一个节点拥有一种特殊类型的磁盘存储，其容量为 800 GiB。你可以为该特殊存储创建一个名称，如 `example.com/special-storage`。然后你就可以按照一定规格的块（如 100 GiB）对其进行发布。在这种情况下，你的节点将会通知它拥有八个 `example.com/special-storage` 类型的资源。

Capacity:

...

example.com/special-storage: 8

如果你想要允许针对特殊存储任意（数量）的请求，你可以按照 1 字节大小的块来发布特殊存储。在这种情况下，你将会发布 800Gi 数量的 `example.com/special-storage` 类型的资源。

Capacity:

...

example.com/special-storage: 800Gi

然后，容器就能够请求任意数量（多达 800Gi）字节的特殊存储。

Capacity:

...

example.com/special-storage: 800Gi

清理

这里是一个从节点移除 `dongle` 资源发布的 PATCH 请求。

PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1

Accept: application/json

Content-Type: application/json-patch+json

Host: k8s-master:8080

```
[  
  {  
    "op": "remove",  
    "path": "/status/capacity/example.com~1dongle",  
  }  
]
```

启动一个代理，以便你可以很容易地向 Kubernetes API 服务器发送请求：

kubectl proxy

在另一个命令窗口中，发送 HTTP PATCH 请求。用你的节点名称替换 <your-node-name>：

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "remove", "path": "/status/capacity/example.com~1dongle"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

验证 dongle 资源的发布已经被移除：

```
kubectl describe node <your-node-name> | grep dongle
```

(你应该看不到任何输出)

接下来

针对应用开发人员

- [将扩展资源分配给容器](#)

针对集群管理员

- [为名字空间配置最小和最大内存约束](#)
- [为名字空间配置最小和最大 CPU 约束](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 14, 2020 at 3:14 PM PST: [\[zh\] Tidy up and fix links in tasks section \(6/10\) \(2c8a55a6e\)](#)

使用 CoreDNS 进行服务发现

此页面介绍了 CoreDNS 升级过程以及如何安装 CoreDNS 而不是 kube-dns。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.9. 要获知版本信息，请输入 kubectl version.

关于 CoreDNS

[CoreDNS](#) 是一个灵活可扩展的 DNS 服务器，可以作为 Kubernetes 集群 DNS。与 Kubernetes 一样，CoreDNS 项目由 [CNCF](#) 托管。

通过在现有的集群中替换 kube-dns，可以在集群中使用 CoreDNS 代替 kube-dns 部署，或者使用 kubeadm 等工具来为你部署和升级集群。

安装 CoreDNS

有关手动部署或替换 kube-dns，请参阅 [CoreDNS GitHub 工程](#)。

迁移到 CoreDNS

使用 kubeadm 升级现有集群

在 Kubernetes 1.10 及更高版本中，当你使用 kubeadm 升级使用 kube-dns 的集群时，你还可以迁移到 CoreDNS。在本例中 kubeadm 将生成 CoreDNS 配置（"Corefile"）基于 kube-dns ConfigMap，保存联邦、存根域和上游名称服务器的配置。

如果你正在从 kube-dns 迁移到 CoreDNS，请确保在升级期间将 CoreDNS 特性门设置为 true。例如，v1.11.0 升级应该是这样的：

```
kubeadm upgrade apply v1.11.0 --feature-gates=CoreDNS=true
```

在 Kubernetes 版本 1.13 和更高版本中，CoreDNS 特性门已经删除，CoreDNS 在默认情况下使用。如果你想升级集群以使用 kube-dns，请遵循 [此处](#)。

在 1.11 之前的版本中，核心文件将被升级过程中创建的文件覆盖。**如果已对其进行自定义，则应保存现有的 ConfigMap。** 在新的 ConfigMap 启动并运行后，你可以重新应用自定义。

如果你在 Kubernetes 1.11 及更高版本中运行 CoreDNS，则在升级期间，将保留现有的 Corefile。

使用 kubeadm 安装 kube-dns 而不是 CoreDNS

说明：在 Kubernetes 1.11 中，CoreDNS 已经升级到通用可用性（GA），并默认安装。

警告：在 Kubernetes 1.18 中，用 kubeadm 来安装 kube-dns 这一做法已经被废弃，会在将来版本中移除。

若要在 1.13 之前版本上安装 kube-dns，请将 CoreDNS 特性门控设置为 false：

```
kubeadm init --feature-gates=CoreDNS=false
```

对于 1.13 版和更高版本，请遵循 [此处](#) 概述到指南。

升级 CoreDNS

从 v1.9 起，Kubernetes 提供了 CoreDNS。你可以在[此处](#) 查看 Kubernetes 随附的 CoreDNS 版本以及对 CoreDNS 所做的更改。

如果你只想升级 CoreDNS 或使用自己的自定义镜像，则可以手动升级 CoreDNS。参看[指南和演练](#) 文档了解如何平滑升级。

CoreDNS 调优

当资源利用方面有问题时，优化 CoreDNS 的配置可能是有用的。有关详细信息，请参阅[有关扩缩 CoreDNS 的文档](#)。

接下来

你可以通过修改 Corefile 来配置 [CoreDNS](#)，以支持比 kube-dns 更多的用例。请参考[CoreDNS 网站](#) 以了解更多信息。

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 August 14, 2020 at 3:14 PM PST: [\[zh\] Tidy up and fix links in tasks section \(6/10\) \(2c8a55a6e\)](#)

使用 KMS 驱动进行数据加密

本页展示了如何配置秘钥管理服务—— Key Management Service (KMS) 驱动和插件以启用 Secret 数据加密。

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.

- 需要 Kubernetes 1.10.0 或更新版本
- 需要 etcd v3 或更新版本

FEATURE STATE: Kubernetes v1.12 [beta]

KMS 加密驱动使用封套加密模型来加密 etcd 中的数据。 数据使用数据加密密钥 (DEK) 加密；每次加密都生成一个新的 DEK。 这些 DEK 经一个密钥加密密钥 (KEK) 加密后在一个远端的 KMS 中存储和管理。 KMS 驱动使用 gRPC 与一个特定的 KMS 插件通信。这个 KMS 插件作为一个 gRPC 服务器被部署在 Kubernetes 主服务器的同一个主机上，负责与远端 KMS 的通信。

配置 KMS 驱动

为了在 API 服务器上配置 KMS 驱动，在加密配置文件中的驱动数组中加入一个类型为 kms 的驱动，并设置下列属性：

- name: KMS 插件的显示名称。
- endpoint: gRPC 服务器 (KMS 插件) 的监听地址。该端点是一个 UNIX 域套接字。
- cachesize: 以明文缓存的数据加密密钥 (DEKs) 的数量。一旦被缓存，就可以直接使用 DEKs 而无需另外调用 KMS；而未被缓存的 DEKs 需要调用一次 KMS 才能解包。
- timeout: 在返回一个错误之前，kube-apiserver 等待 kms-plugin 响应的时间 (默认是 3 秒)。

参见[理解静态数据加密配置](#)

实现 KMS 插件

为实现一个 KMS 插件，你可以开发一个新的插件 gRPC 服务器或启用一个由你的云服务驱动提供的 KMS 插件。你可以将这个插件与远程 KMS 集成，并把它部署到 Kubernetes 的主服务器上。

启用由云服务驱动支持的 KMS

有关启用云服务驱动特定的 KMS 插件的说明，请咨询你的云服务驱动。

开发 KMS 插件 gRPC 服务器

你可以使用 Go 语言的存根文件开发 KMS 插件 gRPC 服务器。对于其他语言，你可以用 proto 文件创建可以用于开发 gRPC 服务器代码的存根文件。

- 使用 Go：使用存根文件 [service.pb.go](#) 中的函数和数据结构开发 gRPC 服务器代码。
- 使用 Go 以外的其他语言：用 protoc 编译器编译 proto 文件：[service.proto](#) 为指定语言生成存根文件。

然后使用存根文件中的函数和数据结构开发服务器代码。

注意：

- kms 插件版本：v1beta1

作为对过程调用 Version 的响应，兼容的 KMS 插件应把 v1beta1 作为 VersionResponse.version 返回

- 消息版本：v1beta1

所有来自 KMS 驱动的消息都把 version 字段设置为当前版本 v1beta1

- 协议：UNIX 域套接字 (unix)

gRPC 服务器应监听 UNIX 域套接字

将 KMS 插件与远程 KMS 整合

KMS 插件可以用任何受 KMS 支持的协议与远程 KMS 通信。所有的配置数据，包括 KMS 插件用于与远程 KMS 通信的认证凭据，都由 KMS 插件独立地存储和管理。KMS 插件可以用额外的元数据对密文进行编码，这些元数据是在把它发往 KMS 进行解密之前可能要用到的。

部署 KMS 插件

确保 KMS 插件与 Kubernetes 主服务器运行在同一主机上。

使用 KMS 驱动加密数据

为了加密数据：

1. 使用 kms 驱动的相应的属性创建一个新的加密配置文件：

```
kind: EncryptionConfiguration
apiVersion: apiserver.config.k8s.io/v1
resources:
  - resources:
    - secrets
  providers:
    - kms:
      name: myKmsPlugin
      endpoint: unix:///tmp/socketfile.sock
      cachesize: 100
      timeout: 3s
    - identity: {}
```

1. 设置 kube-apiserver 的 --encryption-provider-config 参数指向配置文件的位置。
2. 重启 API 服务器。

验证数据已经加密

写入 etcd 时数据被加密。重启 kube-apiserver 后，任何新建或更新的 Secret 在存储时应该已被加密。要验证这点，你可以用 etcdctl 命令行程序获取 Secret 内容。

1. 在默认的命名空间里创建一个名为 secret1 的 Secret：

```
kubectl create secret generic secret1 -n default --from-literal=mykey=mydata
```

1. 用 etcdctl 命令行，从 etcd 读取出 Secret：

```
ETCDCTL_API=3 etcdctl get /kubernetes.io/secrets/default/secret1 [...] | hexdump -C
```

其中 [...] 是用于连接 etcd 服务器的额外参数。

1. 验证保存的 Secret 是否是以 k8s:enc:kms:v1: 开头的，这表明 kms 驱动已经对结果数据加密。
1. 验证 Secret 在被 API 获取时已被正确解密：

```
kubectl describe secret secret1 -n default
```

结果应该是 mykey: mydata。

确保所有 Secret 都已被加密

因为 Secret 是在写入时被加密的，所以在更新 Secret 时也会加密该内容。

下列命令读取所有 Secret 并更新它们以便应用服务器端加密。如果因为写入冲突导致错误发生，请重试此命令。对较大的集群，你可能希望根据命名空间或脚本更新去细分 Secret 内容。

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

从本地加密驱动切换到 KMS 驱动

为了从本地加密驱动切换到 kms 驱动并重新加密所有 Secret 内容：

1. 在配置文件中加入 kms 驱动作为第一个条目，如下列样例所示

```
kind: EncryptionConfiguration
apiVersion: apiserver.config.k8s.io/v1
resources:
  - resources:
    - secrets
  providers:
    - kms:
        name: myKmsPlugin
        endpoint: unix:///tmp/socketfile.sock
        cachesize: 100
    - aescbc:
        keys:
          - name: key1
            secret: <BASE 64 ENCODED SECRET>
```

1. 重启所有 kube-apiserver 进程。
2. 运行下列命令使用 kms 驱动强制重新加密所有 Secret。

```
kubectl get secrets --all-namespaces -o json| kubectl replace -f -
```

禁用静态数据加密

要禁用静态数据加密：

1. 将 identity 驱动作为配置文件中的第一个条目：

```
kind: EncryptionConfiguration
apiVersion: apiserver.config.k8s.io/v1
resources:
  - resources:
    - secrets
  providers:
    - identity: {}
    - kms:
```

```
name: myKmsPlugin
endpoint: unix:///tmp/socketfile.sock
cachesize: 100
```

1. 重启所有 kube-apiserver 进程。
2. 运行下列命令强制重新加密所有 Secret。

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 14, 2020 at 4:16 PM PST: [\[zh\] Tune KMS provider task \(4de8e2964\)](#)

使用 Kubernetes API 访问集群

本页展示了如何使用 Kubernetes API 访问集群

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.

访问集群 API

使用 kubectl 进行首次访问

首次访问 Kubernetes API 时，请使用 Kubernetes 命令行工具 kubectl。

要访问集群，你需要知道集群位置并拥有访问它的凭证。通常，当你完成[入门指南](#)时，这会自动设置完成，或者由其他人设置好集群并将凭证和位置提供给你。

使用此命令检查 kubectl 已知的位置和凭证：

```
kubectl config view
```

许多[样例](#)提供了使用 kubectl 的介绍。完整文档请见 [kubectl 手册](#)。

直接访问 REST API

kubectl 处理对 API 服务器的定位和身份验证。如果你想通过 http 客户端（如 curl 或 wget，或浏览器）直接访问 REST API，你可以通过多种方式对 API 服务器进行定位和身份验证：

1. 以代理模式运行 kubectl（推荐）。推荐使用此方法，因为它用存储的 apiserver 位置并使用自签名证书验证 API 服务器的标识。使用这种方法无法进行中间人（MITM）攻击。
2. 另外，你可以直接为 HTTP 客户端提供位置和身份认证。这适用于被代理混淆的客户端代码。为防止中间人攻击，你需要将根证书导入浏览器。

使用 Go 或 Python 客户端库可以在代理模式下访问 kubectl。

使用 kubectl 代理

下列命令使 kubectl 运行在反向代理模式下。它处理 API 服务器的定位和身份认证。

像这样运行它：

```
kubectl proxy --port=8080 &
```

参见 [kubectl 代理](#) 获取更多细节。

然后你可以通过 curl，wget，或浏览器浏览 API，像这样：

```
curl http://localhost:8080/api/
```

输出类似如下：

```
{
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

不使用 kubectl 代理

通过将身份认证令牌直接传给 API 服务器，可以避免使用 kubectl 代理，像这样：

使用 grep/cut 方式：

```
# 查看所有的集群，因为你的 .kubeconfig 文件中可能包含多个上下文
kubectl config view -o jsonpath='{"Cluster name\tServer\n"}{range .clusters[*]}\n{.name}{`\t`}{.cluster.server}{`\n`}{end}'

# 从上述命令输出中选择你要与之交互的集群的名称
export CLUSTER_NAME="some_server_name"

# 指向引用该集群名称的 API 服务器
APISERVER=$(kubectl config view -o jsonpath=".clusters[?(@.name==\"$CLUSTER_NAME\")].cluster.server")

# 获得令牌
TOKEN=$(kubectl get secrets -o jsonpath=".items[?(@.metadata.annotations['kubernetes\\.io/service-account\\.name']=='default')].data.token" | base64 -d)

# 使用令牌玩转 API
curl -X GET $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

输出类似如下：

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

使用 jsonpath 方式：

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
TOKEN=$(kubectl get secret $(kubectl get serviceaccount default -o jsonpath='{.secrets[0].name}') -o jsonpath='{.data.token}' | base64 --decode)
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

```
{
  "kind": "APIVersions",
  "versions": [
```

```

    "v1"
],
"serverAddressByClientCIDRs": [
{
  "clientCIDR": "0.0.0.0/0",
  "serverAddress": "10.0.1.149:443"
}
]
}

```

上面例子使用了 `--insecure` 标志位。这使它易受到 MITM 攻击。当 `kubectl` 访问集群时，它使用存储的根证书和客户端证书访问服务器。（已安装在 `~/.kube` 目录下）。由于集群认证通常是自签名的，因此可能需要特殊设置才能让你的 http 客户端使用根证书。

在一些集群中，API 服务器不需要身份认证；它运行在本地，或由防火墙保护着。对此并没有一个标准。[配置对 API 的访问](#) 讲解了作为集群管理员可如何对此进行配置。

编程方式访问 API

Kubernetes 官方支持 [Go](#)、[Python](#)、[Java](#)、[dotnet](#)、[Javascript](#) 和 [Haskell](#) 语言的客户端库。还有一些其他客户端库由对应作者而非 Kubernetes 团队提供并维护。参考[客户端库](#)了解如何使用其他语言 来访问 API 以及如何执行身份认证。

Go 客户端

- 要获取库，运行下列命令：`go get k8s.io/client-go/<版本号>/kubernetes`，参见<https://github.com/kubernetes/client-go> 查看受支持的版本。
- 基于 client-go 客户端编写应用程序。

说明：注意 client-go 定义了自己的 API 对象，因此如果需要，请从 client-go 而不是主仓库导入 API 定义，例如 `import "k8s.io/client-go/kubernetes"` 是正确做法。

Go 客户端可以使用与 `kubectl` 命令行工具相同的 [kubeconfig 文件](#) 定位和验证 API 服务器。参见这个[例子](#)：

```

package main

import (
  "context"
  "fmt"
  "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/clientcmd"
)
func main() {

```

```
// uses the current context in kubeconfig
// path-to-kubeconfig -- for example, /root/.kube/config
config, _ := clientcmd.BuildConfigFromFlags("", "<path-to-kubeconfig>")
// creates the clientset
clientset, _ := kubernetes.NewForConfig(config)
// access the API to list pods
pods, _ := clientset.CoreV1().Pods("").List(context.TODO(), v1.ListOptions{})
fmt.Printf("There are %d pods in the cluster\n", len(pods.Items))
}
```

如果该应用程序部署为集群中的一个 Pod , 请参阅[下一节](#)。

Python 客户端

要使用 [Python 客户端](#) , 运行下列命令 : pip install kubernetes。 参见 [Python 客户端库主页](#) 了解更多安装选项。

Python 客户端可以使用与 kubectl 命令行工具相同的 [kubeconfig 文件](#) 定位和验证 API 服务器。参见这个 [例子](#) :

```
from kubernetes import client, config

config.load_kube_config()

v1=client.CoreV1Api()
print("Listing pods with their IPs:")
ret = v1.list_pod_for_all_namespaces(watch=False)
for i in ret.items:
    print("%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name)
)
```

Java 客户端

- 要安装 [Java 客户端](#) , 只需执行 :

```
# 克隆 Java 库
git clone --recursive https://github.com/kubernetes-client/java

# 安装项目文件、POM 等
cd java
mvn install
```

参阅<https://github.com/kubernetes-client/java/releases> 了解当前支持的版本。

Java 客户端可以使用 kubectl 命令行所使用的 [kubeconfig 文件](#) 以定位 API 服务器并向其认证身份。参看此[示例](#) :

```
package io.kubernetes.client.examples;

import io.kubernetes.client.ApiClient;
import io.kubernetes.client.ApiException;
import io.kubernetes.client.Configuration;
import io.kubernetes.client.apis.CoreV1Api;
import io.kubernetes.client.models.V1Pod;
import io.kubernetes.client.models.V1PodList;
import io.kubernetes.client.util.ClientBuilder;
import io.kubernetes.client.util.KubeConfig;
import java.io.FileReader;
import java.io.IOException;

/**
 * A simple example of how to use the Java API from an application outside a
kubernetes cluster
 *
 * <p>Easiest way to run this: mvn exec:java
 * -
Dexec.mainClass= "io.kubernetes.client.examples.KubeConfigFileClientExample"
*
*/
public class KubeConfigFileClientExample {
    public static void main(String[] args) throws IOException, ApiException {

        // file path to your KubeConfig
        String kubeConfigPath = "~/.kube/config";

        // loading the out-of-cluster config, a kubeconfig from file-system
        ApiClient client =
            ClientBuilder.kubeconfig(KubeConfig.loadKubeConfig(new FileReader(kubeConfigPath))).build();

        // set the global default api-client to the in-cluster one from above
        Configuration.setDefaultApiClient(client);

        // the CoreV1Api loads default api-client from global configuration.
        CoreV1Api api = new CoreV1Api();

        // invokes the CoreV1Api client
        V1PodList list = api.listPodForAllNamespaces(null, null, null, null, null, null, null, null,
null, null);
        System.out.println("Listing all pods: ");
        for (V1Pod item : list.getItems()) {
            System.out.println(item.getMetadata().getName());
        }
    }
}
```

```
}
```

.Net 客户端

要使用[.Net 客户端](#)，运行下面的命令：dotnet add package KubernetesClient --version 1.6.1。参见[.Net 客户端库页面](#)了解更多安装选项。关于可支持的版本，参见<https://github.com/kubernetes-client/csharp/releases>。

.Net 客户端可以使用与 kubectl CLI 相同的 [kubeconfig 文件](#) 来定位并验证 API 服务器。参见[样例](#)：

```
using System;
using k8s;

namespace simple
{
    internal class PodList
    {
        private static void Main(string[] args)
        {
            var config = KubernetesClientConfiguration.BuildDefaultConfig();
            IKubernetes client = new Kubernetes(config);
            Console.WriteLine("Starting Request!");

            var list = client.ListNamespacedPod("default");
            foreach (var item in list.Items)
            {
                Console.WriteLine(item.Metadata.Name);
            }
            if (list.Items.Count == 0)
            {
                Console.WriteLine("Empty!");
            }
        }
    }
}
```

JavaScript 客户端

要安装[JavaScript 客户端](#)，运行下面的命令：npm install @kubernetes/client-node。参见<https://github.com/kubernetes-client/javascript/releases>了解可支持的版本。

JavaScript 客户端可以使用 kubectl 命令行所使用的 [kubeconfig 文件](#) 以定位 API 服务器并向其认证身份。参见[此例](#)：

```
const k8s = require('@kubernetes/client-node');

const kc = new k8s.KubeConfig();
kc.loadFromDefault();

const k8sApi = kc.makeApiClient(k8s.CoreV1Api);

k8sApi.listNamespacedPod('default').then((res) => {
  console.log(res.body);
});
```

Haskell 客户端

参考 <https://github.com/kubernetes-client/haskell/releases> 了解支持的版本。

[Haskell 客户端](#) 可以使用 kubectl 命令行所使用的 [kubeconfig 文件](#) 以定位 API 服务器并向其认证身份。参见[此例](#)：

```
exampleWithKubeConfig :: IO ()
exampleWithKubeConfig = do
  oidcCache <- atomically $ newTVar $ Map.fromList []
  (mgr, kcfg) <- mkKubeClientConfig oidcCache $ KubeConfigFile "/path/to/
kubeconfig"
  dispatchMime
    mgr
    kcfg
    (CoreV1.listPodForAllNamespaces (Accept MimeJSON))
  >>= print
```

从 Pod 中访问 API

从 Pod 内部访问 API 时，定位 API 服务器和向服务器认证身份的操作 与上面描述的外部客户场景不同。

从 Pod 使用 Kubernetes API 的最简单的方法就是使用官方的 [客户端库](#)。这些库可以自动发现 API 服务器并进行身份验证。

使用官方客户端库

从一个 Pod 内部连接到 Kubernetes API 的推荐方式为：

- 对于 Go 语言客户端，使用官方的 [Go 客户端库](#)。函数 rest.InClusterConfig() 自动处理 API 主机发现和身份认证。参见[这里的一个例子](#)。
- 对于 Python 客户端，使用官方的 [Python 客户端库](#)。函数 config.load_incluster_config() 自动处理 API 主机的发现和身份认证。参见[这里的一个例子](#)。
- 还有一些其他可用的客户端库，请参阅[客户端库](#)页面。

在以上场景中，客户端库都使用 Pod 的服务账号凭据来与 API 服务器安全地通信。

直接访问 REST API

在运行在 Pod 中时，可以通过 default 命名空间中的名为 kubernetes 的服务访问 Kubernetes API 服务器。也就是说，Pod 可以使用 kubernetes.default.svc 主机名来查询 API 服务器。官方客户端库自动完成这个工作。

向 API 服务器进行身份认证的推荐做法是使用 [服务账号](#)凭据。默认情况下，每个 Pod 与一个服务账号关联，该服务账户的凭证（令牌）放置在此 Pod 中每个容器的文件系统树中的 /var/run/secrets/kubernetes.io/serviceaccount/token 处。

如果由证书包可用，则凭证包被放入每个容器的文件系统树中的 /var/run/secrets/kubernetes.io/serviceaccount/ca.crt 处，且将被用于验证 API 服务器的服务证书。

最后，用于命名空间域 API 操作的默认命名空间放置在每个容器中的 /var/run/secrets/kubernetes.io/serviceaccount/namespace 文件中。

使用 kubectl proxy

如果你希望不实用官方客户端库就完成 API 查询，可以将 kubectl proxy 作为 [command](#) 在 Pod 启动一个边车（Sidecar）容器。这样，kubectl proxy 自动完成对 API 的身份认证，并将其暴露到 Pod 的 localhost 接口，从而 Pod 中的其他容器可以直接使用 API。

不使用代理

通过将认证令牌直接发送到 API 服务器，也可以避免运行 kubectl proxy 命令。内部的证书机制能够为链接提供保护。

```
# 指向内部 API 服务器的主机名
APISERVER=https://kubernetes.default.svc

# 服务账号令牌的路径
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# 读取 Pod 的名字空间
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# 读取服务账号的持有者令牌
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# 引用内部整数机构 ( CA )
CACERT=${SERVICEACCOUNT}/ca.crt

# 使用令牌访问 API
```

```
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISEVER}/api
```

输出类似于：

```
{  
  "kind": "APIVersions",  
  "versions": [  
    "v1"  
,  
    "serverAddressByClientCIDRs": [  
      {  
        "clientCIDR": "0.0.0.0/0",  
        "serverAddress": "10.0.1.149:443"  
      }  
    ]  
}
```

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 18, 2020 at 8:38 PM PST: [Update access-cluster-api.md \(3f4388829\)](#)

保护集群安全

本文档涉及与保护集群免受意外或恶意访问有关的主题，并对总体安全性提出建议。

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.

控制对 Kubernetes API 的访问

因为 Kubernetes 是完全通过 API 驱动的，所以，控制和限制谁可以通过 API 访问集群，以及允许这些访问者执行什么样的 API 动作，就成为了安全控制的第一道防线。

为所有 API 交互使用传输层安全（TLS）

Kubernetes 期望集群中所有的 API 通信在默认情况下都使用 TLS 加密，大多数安装方法也允许创建所需的证书并且分发到集群组件中。请注意，某些组件和安装方法可能使用 HTTP 来访问本地端口，管理员应该熟悉每个组件的设置，以识别潜在的不安全的流量。

API 认证

安装集群时，选择一个 API 服务器的身份验证机制，去使用与之匹配的公共访问模式。例如，小型的单用户集群可能希望使用简单的证书或静态承载令牌方法。更大的集群则可能希望整合现有的、OIDC、LDAP 等允许用户分组的服务器。

所有 API 客户端都必须经过身份验证，即使它是基础设施的一部分，比如节点、代理、调度程序和卷插件。这些客户端通常使用 [服务帐户](#) 或 X509 客户端证书，并在集群启动时自动创建或是作为集群安装的一部分进行设置。

如果你希望获取更多信息，请参考[认证参考文档](#)。

API 授权

一旦通过身份认证，每个 API 的调用都将通过鉴权检查。Kubernetes 集成[基于角色的访问控制（RBAC）](#)组件，将传入的用户或组与一组绑定到角色的权限匹配。这些权限将动作（get, create, delete）和资源（pod, service, node）在命名空间或者集群范围内结合起来，根据客户可能希望执行的操作，提供了一组提供合理的违约责任分离的外包角色。建议你将[节点](#)和[RBAC](#)一起作为授权者，再与[NodeRestriction](#)准入插件结合使用。

与身份验证一样，简单而广泛的角色可能适合于较小的集群，但是随着更多的用户与集群交互，可能需要将团队划分成有更多角色限制的单独的命名空间。

就鉴权而言，理解怎么样更新一个对象可能导致在其它地方的发生什么样的行为是非常重要的。例如，用户可能不能直接创建 Pod，但允许他们通过创建一个 Deployment 来创建这些 Pod，这将让他们间接创建这些 Pod。同样地，从 API 删除一个节点将导致调度到这些节点上的 Pod 被中止，并在其他节点上重新创建。原生的角色设计代表了灵活性和常见用例之间的平衡，但有限制的角色应该仔细审查，以防止意外升级。如果外包角色不满足你的需求，则可以为用例指定特定的角色。

如果你希望获取更多信息，请参阅[鉴权参考](#)。

控制对 Kubelet 的访问

Kubelet 公开 HTTPS 端点，这些端点授予节点和容器强大的控制权。默认情况下，Kubelet 允许对此 API 进行未经身份验证的访问。

生产级别的集群应启用 Kubelet 身份验证和授权。

如果你希望获取更多信息，请参考 [Kubelet 身份验证/授权参考](#)。

控制运行时负载或用户的能力

Kubernetes 中的授权故意设置为了高层级，它侧重于对资源的粗粒度行为。更强大的控制是以通过用例限制这些对象如何作用于集群、自身和其他资源上的策略存在的。

限制集群上的资源使用

[资源配置](#) 限制了授予命名空间的资源的数量或容量。这通常用于限制命名空间可以分配的 CPU、内存或持久磁盘的数量，但也可以控制每个命名空间中有多少个 Pod、服务或卷的存在。

[限制范围](#) 限制上述某些资源的最大值或者最小值，以防止用户使用类似内存这样的通用保留资源时请求不合理的过高或过低的值，或者在没有指定的情况下提供默认限制。

控制容器运行的特权

Pod 定义包含了一个[安全上下文](#)，用于描述允许它请求访问某个节点上的特定 Linux 用户（如 root）、获得特权或访问主机网络、以及允许它在主机节点上不受约束地运行的其它控件。[Pod 安全策略](#) 可以限制哪些用户或服务帐户可以提供危险的安全上下文设置。例如，Pod 的安全策略可以限制卷挂载，尤其是 hostpath，这些都是 Pod 应该控制的一些方面。

一般来说，大多数应用程序需要限制对主机资源的访问，他们可以在不能访问主机信息的情况下成功以根进程（UID 0）运行。但是，考虑到与 root 用户相关的特权，在编写应用程序容器时，你应该使用非 root 用户运行。类似地，希望阻止客户端应用程序逃避其容器的管理员，应该使用限制性的 pod 安全策略。

限制网络访问

基于命名空间的[网络策略](#) 允许应用程序作者限制其它命名空间中的哪些 Pod 可以访问它们命名空间内的 Pod 和端口。现在已经有许多支持网络策略的 [Kubernetes 网络供应商](#)。

对于可以控制用户的应用程序是否在集群之外可见的许多集群，配额和限制范围也可用于控制用户是否可以请求节点端口或负载均衡服务。

在插件或者环境基础上控制网络规则可以增加额外的保护措施，比如节点防火墙、物理分离群集节点以防止串扰、或者高级的网络策略。

限制云 metadata API 访问

云平台（AWS, Azure, GCE 等）经常讲 metadata 本地服务暴露给实例。默认情况下，这些 API 可由运行在实例上的 Pod 访问，并且可以包含该云节点的凭据或配置数据（如 kubelet 凭据）。这些凭据可以用于在集群内升级或在同一账户下升级到其他云服务。

在云平台上运行 Kubernetes 时，限制对实例凭据的权限，使用 [网络策略](#) 限制对 metadata API 的 pod 访问，并避免使用配置数据来传递机密。

控制 Pod 可以访问哪些节点

默认情况下，对哪些节点可以运行 pod 没有任何限制。Kubernetes 给最终用户提供了一组丰富的策略用于控制 pod 放在节点上的位置，以及基于污点的 Pod 放置和驱逐。对于许多集群，可以约定由作者采用或者强制通过工具使用这些策略来分离工作负载。

对于管理员，Beta 阶段的准入插件 PodNodeSelector 可用于强制命名空间中的 Pod 使用默认或需要使用特定的节点选择器。如果最终用户无法改变命名空间，这可以强烈地限制所有的 pod 在特定工作负载的位置。

保护集群组件免受破坏

本节描述保护集群免受破坏的一些常见模式。

限制访问 etcd

对于 API 来说，拥有 etcd 后端的写访问权限，相当于获得了整个集群的 root 权限，并且可以使用写访问权限来相当快速地升级。从 API 服务器访问它们的 etcd 服务器，管理员应该使用广受信任的凭证，如通过 TLS 客户端证书的相互认证。通常，我们建议将 etcd 服务器隔离到只有 API 服务器可以访问的防火墙后面。

注意：允许集群中其它组件拥有读或写全空间的权限去访问 etcd 实例，相当于授予群集管理员访问的权限。对于非主控组件，强烈推荐使用单独的 etcd 实例，或者使用 etcd 的访问控制列表 去限制只能读或者写空间的一个子集。

开启审计日志

[审计日志](#) 是 Beta 特性，负责记录 API 操作以便在发生破坏时进行事后分析。建议启用审计日志，并将审计文件归档到安全服务器上。

限制使用 alpha 和 beta 特性

Kubernetes 的 alpha 和 beta 特性还在努力开发中，可能存在导致安全漏洞的缺陷或错误。要始终评估 alpha 和 beta 特性可能为你的安全态势带来的风险。当你怀疑存在风险时，可以禁用那些不需要使用的特性。

频繁回收基础设施证书

一个 Secret 或凭据的寿命越短，攻击者就越难使用该凭据。在证书上设置短生命周期并实现自动回收，是控制安全的一个好方法。因此，使用身份验证提供程序时，应该要求可以控制发布令牌的可用时间，并尽可能使用短寿命。如果在外部集成中使用服务帐户令牌，则应该频繁地回收这些令牌。例如，一旦引导阶段完成，就应该撤销用于设置节点的引导令牌，或者取消它的授权。

在启用第三方集成之前，请先审查它们

许多集成到 Kubernetes 的第三方都可以改变你集群的安全配置。启用集成时，在授予访问权限之前，你应该始终检查扩展所请求的权限。例如，许多安全集成可以请求访问来查看集群上的所有 Secret，从而有效地使该组件成为集群管理。当有疑问时，如果可能的话，将集成限制在单个命名空间中运行。

如果组件创建的 Pod 能够在命名空间中做一些类似 kube-system 命名空间中的事情，那么它也可能是出乎意料的强大。因为这些 Pod 可以访问服务账户的 Secret，或者，如果这些服务帐户被授予访问许可的 [Pod 安全策略](#) 的权限，它们能以高权限运行。

对 Secret 进行静态加密

一般情况下，etcd 数据库包含了通过 Kubernetes API 可以访问到的所有信息，并且可以授予攻击者对集群状态的可见性。始终使用经过良好审查的备份和加密解决方案来加密备份，并考虑在可能的情况下使用全磁盘加密。

Kubernetes 1.7 包含了[静态数据加密](#)，它是一个 alpha 特性，会加密 etcd 里面的 Secret 资源，以防止某一方通过查看 etcd 的备份文件查看到这些 Secret 的内容。虽然目前这还只是实验性的功能，但是在备份没有加密或者攻击者获取到 etcd 的读访问权限的时候，它能提供额外的防御层级。

接收安全更新和报告漏洞的警报

加入 [kubernetes-announce](#) 组，能够获取有关安全公告的邮件。有关如何报告漏洞的更多信息，请参见 [安全报告](#) 页面。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 15, 2020 at 11:30 AM PST: [\[lzh\] Tidy up and fix links in tasks section \(7/10\) \(30423d108\)](#)

关键插件 Pod 的调度保证

除了在主机上运行的 Kubernetes 核心组件（如 api-server、scheduler、controller-manager）之外，还有许多插件，由于各种原因，必须在常规集群节点（而不是 Kubernetes 主节点）上运行。其中一些插件对于功能完备的群集至关重要，例如 Heapster、DNS 和 UI。如果关键插件被逐出（手动或作为升级等其他操作的副作用）或者变成挂起状态，群集可能会停止正常工作。关键插件进入挂起状态的例子有：集群利用率过高；被逐出的关键插件 Pod 释放了空间，但该空间被之前悬决的 Pod 占用；由于其它原因导致节点上可用资源的总量发生变化。

标记关键 Pod

要将 pod 标记为关键性 (critical) , pod 必须在 kube-system 命名空间中运行 (可通过参数配置)。同时 , 需要将 priorityClassName 设置为 system-cluster-critical 或 system-node-critical , 后者是整个群集的最高级别。或者 , 也可以为 Pod 添加名为 scheduler.alpha.kubernetes.io/critical-pod、值为空字符串的注解。不过 , 这一注解从 1.13 版本开始不再推荐使用 , 并将在 1.14 中删除。

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 June 01, 2020 at 9:23 AM PST: [add zh pages \(4b35d4d40\)](#)

升级集群

本页概述升级 Kubernetes 集群的步骤。

升级集群的方式取决于你最初部署它的方式、以及后续更改它的方式。

从高层规划的角度看 , 要执行的步骤是 :

- 升级[控制平面](#)
- 升级集群中的节点
- 升级 [kubectl](#) 之类的客户端
- 根据新 Kubernetes 版本带来的 API 变化 , 调整清单文件和其他资源

准备开始

你必须有一个集群。本页内容涉及从 Kubernetes 1.19 升级到 Kubernetes 1.20。如果你的集群未运行 Kubernetes 1.19 , 那请参考目标 Kubernetes 版本的文档。

升级方法

kubeadm

如果你的集群是使用 kubeadm 安装工具部署而来 , 那么升级群集的详细信息 , 请参阅 [升级 kubeadm 集群](#)。

升级集群之后 , 要记得 [安装最新版本的 kubectl](#)。

手动部署

注意：这些步骤不考虑第三方扩展，例如网络和存储插件。

你应该跟随下面操作顺序，手动更新控制平面：

- etcd (所有实例)
- kube-apiserver (所有控制平面的宿主机)
- kube-controller-manager
- kube-scheduler
- cloud controller manager, 在你用到时

现在，你应该 [安装最新版本的 kubectl](#)。

对于群集中的每个节点，[排空](#) 节点，然后，或者用一个运行了 1.20 kubelet 的新节点替换它；或者升级此节点的 kubelet，并使节点恢复服务。

其他部署方式

参阅你的集群部署工具对应的文档，了解用于维护的推荐设置步骤。

升级后的任务

切换群集的存储 API 版本

对象序列化到 etcd，是为了提供集群中活动 Kubernetes 资源的内部表示法，这些对象都使用特定版本的 API 编写。

当底层的 API 更改时，这些对象可能需要用新 API 重写。如果不能做到这一点，会导致再也不能用 Kubernetes API 服务器解码、使用该对象。

对于每个受影响的对象，用最新支持的 API 获取它，然后再用最新支持的 API 写回来。

更新清单

升级到新版本 Kubernetes 就可以提供新的 API。

你可以使用 kubectl convert 命令在不同 API 版本之间转换清单。例如：

```
kubectl convert -f pod.yaml --output-version v1
```

kubectl 替换了 pod.yaml 的内容，在新的清单文件中，kind 被设置为 Pod（未变），但 apiVersion 则被修订了。

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 22, 2020 at 10:55 PM PST: [\[zh\] Translate task cluster-upgrade \(edff1c853\)](#)

名字空间演练

Kubernetes [名字空间](#) 有助于不同的项目、团队或客户去共享 Kubernetes 集群。

名字空间通过以下方式实现这点：

1. 为[名字](#)设置作用域.
2. 为集群中的部分资源关联鉴权和策略的机制。

使用多个名字空间是可选的。

此示例演示了如何使用 Kubernetes 名字空间细分群集。

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.

环境准备

此示例作如下假设：

1. 你已拥有一个[配置好的 Kubernetes 集群](#)。
 2. 你已对 Kubernetes 的 [Pods](#)、[Services](#) 和 [Deployments](#) 有基本理解。
1. 理解默认名字空间

默认情况下，Kubernetes 集群会在配置集群时实例化一个默认名字空间，用以存放集群所使用的默认 Pod、Service 和 Deployment 集合。

假设你有一个新的集群，你可以通过执行以下操作来检查可用的名字空间：

```
kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	13m

创建新的名字空间

在本练习中，我们将创建两个额外的 Kubernetes 名字空间来保存我们的内容。

我们假设一个场景，某组织正在使用共享的 Kubernetes 集群来支持开发和生产：

开发团队希望在集群中维护一个空间，以便他们可以查看用于构建和运行其应用程序的 Pod、Service 和 Deployment 列表。在这个空间里，Kubernetes 资源被自由地加入或移除，对谁能够或不能修改资源的限制被放宽，以实现敏捷开发。

运维团队希望在集群中维护一个空间，以便他们可以强制实施一些严格的规程，对谁可以或谁不可以操作运行生产站点的 Pod、Service 和 Deployment 集合进行控制。

该组织可以遵循的一种模式是将 Kubernetes 集群划分为两个名字空间：development 和 production。

让我们创建两个新的名字空间来保存我们的工作。

文件 [namespace-dev.json](#) 描述了 development 名字空间：

[admin/namespace-dev.json](#)



```
{  
  "apiVersion": "v1",  
  "kind": "Namespace",  
  "metadata": {  
    "name": "development",  
    "labels": {  
      "name": "development"  
    }  
  }  
}
```

使用 kubectl 创建 development 名字空间。

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

将下列的内容保存到文件 [namespace-prod.json](#) 中，这些内容是对 production 名字空间的描述：

[admin/namespace-prod.json](#)



```
{  
  "apiVersion": "v1",  
  "kind": "Namespace",  
  "metadata": {  
    "name": "production",  
    "labels": {  
      "name": "production"  
    }  
  }  
}
```

```
"name": "production",
"labels": {
  "name": "production"
}
}
```

让我们使用 kubectl 创建 production 名字空间。

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.json
```

为了确保一切正常，我们列出集群中的所有名字空间。

```
kubectl get namespaces --show-labels
```

NAME	STATUS	AGE	LABELS
default	Active	32m	<none>
development	Active	29s	name=development
production	Active	23s	name=production

在每个名字空间中创建 pod

Kubernetes 名字空间为集群中的 Pod、Service 和 Deployment 提供了作用域。

与一个名字空间交互的用户不会看到另一个名字空间中的内容。

为了演示这一点，让我们在 development 名字空间中启动一个简单的 Deployment 和 Pod。

我们首先检查一下当前的上下文：

```
kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: REDACTED
  server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
contexts:
- context:
  cluster: lithe-cocoa-92103_kubernetes
  user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
current-context: lithe-cocoa-92103_kubernetes
kind: Config
preferences: {}
users:
```

```
- name: lithe-cocoa-92103_kubernetes
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b
- name: lithe-cocoa-92103_kubernetes-basic-auth
  user:
    password: h5M0FtUUIfIBSdI7
    username: admin
```

kubectl config current-context

lithe-cocoa-92103_kubernetes

下一步是为 kubectl 客户端定义一个上下文，以便在每个名字空间中工作。“cluster”和“user”字段的值将从当前上下文中复制。

```
kubectl config set-context dev --namespace=development \
--cluster=lithe-cocoa-92103_kubernetes \
--user=lithe-cocoa-92103_kubernetes
```

```
kubectl config set-context prod --namespace=production \
--cluster=lithe-cocoa-92103_kubernetes \
--user=lithe-cocoa-92103_kubernetes
```

默认情况下，上述命令会添加两个上下文到 .kube/config 文件中。你现在可以查看上下文并根据你希望使用的名字空间并在这两个新的请求上下文之间切换。

查看新的上下文：

kubectl config view

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority-data: REDACTED
  server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
contexts:
- context:
  cluster: lithe-cocoa-92103_kubernetes
  user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
- context:
  cluster: lithe-cocoa-92103_kubernetes
  namespace: development
  user: lithe-cocoa-92103_kubernetes
  name: dev
```

```
- context:  
  cluster: lithe-cocoa-92103_kubernetes  
  namespace: production  
  user: lithe-cocoa-92103_kubernetes  
  name: prod  
current-context: lithe-cocoa-92103_kubernetes  
kind: Config  
preferences: {}  
users:  
- name: lithe-cocoa-92103_kubernetes  
  user:  
    client-certificate-data: REDACTED  
    client-key-data: REDACTED  
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b  
- name: lithe-cocoa-92103_kubernetes-basic-auth  
  user:  
    password: h5M0FtUUIfIBSdI7  
    username: admin
```

让我们切换到 development 名字空间进行操作。

```
kubectl config use-context dev
```

你可以使用下列命令验证当前上下文：

```
kubectl config current-context
```

```
dev
```

此时，我们从命令行向 Kubernetes 集群发出的所有请求都限定在 development 名字空间中。

让我们创建一些内容。

[admin/snowflake-deployment.yaml](#)



```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  labels:  
    app: snowflake  
    name: snowflake  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: snowflake
```

```
template:  
  metadata:  
    labels:  
      app: snowflake  
  spec:  
    containers:  
      - image: k8s.gcr.io/serve_hostname  
        imagePullPolicy: Always  
        name: snowflake
```

应用清单文件来创建 Deployment。

我们刚刚创建了一个副本大小为 2 的 Deployment，该 Deployment 运行名为 snowflake 的 Pod，其中包含一个仅提供主机名服务的基本容器。

```
kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
snowflake	2	2	2	2	2m

```
kubectl get pods -l app=snowflake
```

NAME	READY	STATUS	RESTARTS	AGE
snowflake-3968820950-9dgr8	1/1	Running	0	2m
snowflake-3968820950-vgc4n	1/1	Running	0	2m

这很棒，开发人员可以做他们想要的事情，而不必担心影响 production 名字空间中的内容。

让我们切换到 production 名字空间，展示一个名字空间中的资源如何对另一个名字空间不可见。

```
kubectl config use-context prod
```

production 名字空间应该是空的，下列命令应该返回的内容为空。

```
kubectl get deployment  
kubectl get pods
```

生产环境需要以放牛的方式运维，让我们创建一些名为 cattle 的 Pod。

```
kubectl create deployment cattle --image=k8s.gcr.io/serve_hostname --  
replicas=5  
kubectl get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
cattle	5	5	5	5	10s

```
kubectl get pods -l run=cattle
```

NAME	READY	STATUS	RESTARTS	AGE
cattle-2263376956-41xy6	1/1	Running	0	34s
cattle-2263376956-kw466	1/1	Running	0	34s
cattle-2263376956-n4v97	1/1	Running	0	34s
cattle-2263376956-p5p3i	1/1	Running	0	34s
cattle-2263376956-sxpth	1/1	Running	0	34s

此时，应该很清楚的展示了用户在一个名字空间中创建的资源对另一个名字空间是不可见的。

随着 Kubernetes 中的策略支持的发展，我们将扩展此场景，以展示如何为每个名字空间提供不同的授权规则。

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 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

启用 EndpointSlices

本页提供启用 Kubernetes EndpointSlice 的总览。

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.

介绍

EndpointSlice（端点切片）为 Kubernetes Endpoints 提供了可伸缩和可扩展的替代方案。它们建立在 Endpoints 提供的功能基础之上，并以可伸缩的方式进行扩展。

当 Service 具有大量 (>100) 网络端点时，它们将被分成多个较小的 EndpointSlice 资源，而不是单个大型 Endpoints 资源。

启用 EndpointSlice

FEATURE STATE: Kubernetes v1.17 [beta]

说明：

EndpointSlice 资源旨在解决较早资源：Endpoints 中的缺点。一些 Kubernetes 组件和第三方应用程序继续使用并依赖 Endpoints。既然情况如此，应该将 EndpointSlices 视为集群中 Endpoints 的补充，而不是彻底替代。

Kubernetes 中的 EndpointSlice 功能包含若干不同组件。它们中的大部分都是默认被启用的：

- *EndpointSlice API*：EndpointSlice 隶属于 discovery.k8s.io/v1beta1 API。此 API 处于 Beta 阶段，从 Kubernetes 1.17 开始默认被启用。下面列举的所有组件都依赖于此 API 被启用。
- *EndpointSlice 控制器*：此 [控制器](#) 为 Service 维护 EndpointSlice 及其引用的 Pods。此控制器通过 EndpointSlice 特性门控控制。自从 Kubernetes 1.18 起，该特性门控默认被启用。
- *EndpointSliceMirroring 控制器*：此 [控制器](#) 将自定义的 Endpoints 映射为 EndpointSlice。控制器受 EndpointSlice 特性门控控制。该特性门控自 1.19 开始被默认启用。
- *kube-proxy*：当 [kube-proxy](#) 被配置为使用 EndpointSlice 时，它会支持更大数量的 Service 端点。此功能在 Linux 上受 EndpointSliceProxying 特性门控控制；在 Windows 上受 WindowsEndpointSliceProxying 特性门控控制。在 Linux 上，从 Kubernetes 1.19 版本起自动启用。目前尚未在 Windows 节点上默认启用。要在 Windows 节点上配置 kube-proxy 使用 EndpointSlice，你需要为 kube-proxy 启用 WindowsEndpointSliceProxying [特性门控](#)。

API 字段

EndpointSlice API 中的某些字段有对应的特性门控控制。

- `EndpointSlicenodeName` 特性门控控制对 `nodeName` 字段的访问。这是默认情况下禁用的 Alpha 功能。
- `EndpointSliceTerminating` 特性门控控制对 `serving` 和 `terminating` 状况字段的访问。这是默认情况下禁用的 Alpha 功能。

使用 EndpointSlice

在集群中完全启用 EndpointSlice 的情况下，你应该看到对应于每个 Endpoints 资源的 EndpointSlice 资源。除了支持现有的 Endpoints 功能外，EndpointSlices 将允许集群中网络端点更好的可伸缩性和可扩展性。

接下来

- 参阅 [EndpointSlices](#)
- 参阅 [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 December 11, 2020 at 9:05 AM PST: [Update enabling-endpointslices.md \(59e11f5ca\)](#)

启用/禁用 Kubernetes API

本页展示怎么用集群的 [控制平面](#). 启用/禁用 API 版本。

通过 API 服务器的命令行参数 `--runtime-config=api/<version>`，可以开启/关闭某个指定的 API 版本。此参数的值是一个逗号分隔的 API 版本列表。此列表中，后面的值可以覆盖前面的值。

命令行参数 `runtime-config` 支持两个特殊的值 (keys) :

- `api/all` : 指所有已知的 API
- `api/legacy` : 指过时的 API。过时的 API 就是明确地 [弃用](#) 的 API。

例如：为了停用除去 v1 版本之外的全部其他 API 版本，就用参数 `--runtime-config=api/all=false,api/v1=true` 启动 `kube-apiserver`。

接下来

阅读[完整的文档](#), 以了解 `kube-apiserver` 组件。

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 22, 2020 at 2:11 PM PST: [\[zh\] Translate task enable-disable-api make change according to tengqqm's comment \(0720b1d6d\)](#)

在 Kubernetes 集群中使用 NodeLocal DNSCache

本页概述了 Kubernetes 中的 NodeLocal DNSCache 功能。

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.

引言

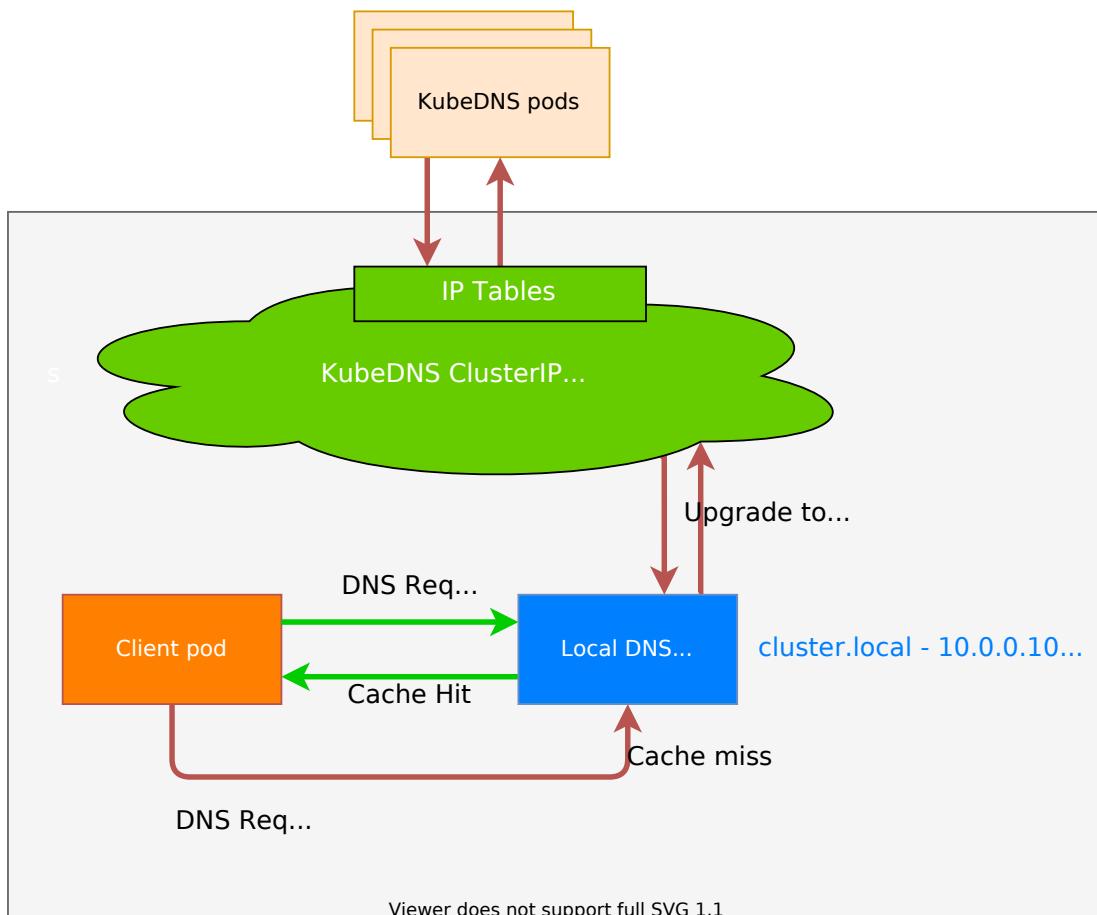
NodeLocal DNSCache 通过在集群节点上作为 DaemonSet 运行 dns 缓存代理来提高集群 DNS 性能。在当今的体系结构中，处于 ClusterFirst DNS 模式的 Pod 可以连接到 kube-dns serviceIP 进行 DNS 查询。通过 kube-proxy 添加的 iptables 规则将其转换为 kube-dns/CoreDNS 端点。借助这种新架构，Pods 将可以访问在同一节点上运行的 dns 缓存代理，从而避免了 iptables DNAT 规则和连接跟踪。本地缓存代理将查询 kube-dns 服务以获取集群主机名的缓存缺失（默认为 cluster.local 后缀）。

动机

- 使用当前的 DNS 体系结构，如果没有本地 kube-dns/CoreDNS 实例，则具有最高 DNS QPS 的 Pod 可能必须延伸到另一个节点。在这种脚本下，拥有本地缓存将有助于改善延迟。
- 跳过 iptables DNAT 和连接跟踪将有助于减少 [conntrack 竞争](#) 并避免 UDP DNS 条目填满 conntrack 表。
- 从本地缓存代理到 kube-dns 服务的连接可以升级到 TCP。TCP conntrack 条目将在连接关闭时被删除，相反 UDP 条目必须超时(默认 `nf_conntrack_udp_time_out` 是 30 秒)
- 将 DNS 查询从 UDP 升级到 TCP 将减少归因于丢弃的 UDP 数据包和 DNS 超时的尾部等待时间，通常长达 30 秒 (3 次重试 + 10 秒超时)。
- 在节点级别对 dns 请求的度量和可见性。
- 可以重新启用负缓存，从而减少对 kube-dns 服务的查询数量。

架构图

启用 NodeLocal DNSCache 之后，这是 DNS 查询所遵循的路径：



NodeLocal DNSCache 流

此图显示了 NodeLocal DNSCache 如何处理 DNS 查询。

配置

可以使用以下命令启用此功能：

```
KUBE_ENABLE_NODELOCAL_DNS=true go run hack/e2e.go -v --up
```

这适用于在 GCE 上创建 e2e 集群。 在所有其他环境上，以下步骤将设置 NodeLocal DNSCache：

- 可以使用 `kubectl create -f` 命令应用类似于[这个](#)的 Yaml。
- 需要修改 `kubelet` 的 `--cluster-dns` 标志以使用 NodeLocal DNSCache 正在侦听的 `LOCAL_DNS` IP (默认为 169.254.20.10)

启用后，`node-local-dns` Pods 将在每个集群节点上的 `kube-system` 名称空间中运行。此 Pod 在缓存模式下运行 [CoreDNS](#)，因此每个节点都可以使用不同插件公开的所有 CoreDNS 指标。

功能可用性

可以在任何 K8s 版本中使用上面指定的 yaml 应用该插件。功能支持如下所述：

k8s 版本	功能支持
1.15	Beta(默认情况下未启用)
1.13	Alpha(默认情况下未启用)

反馈

此页是否对您有帮助？

是 否

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

最后修改 June 01, 2020 at 9:23 AM PST: [add zh pages \(4b35d4d40\)](#)

在 Kubernetes 集群中使用 sysctl

FEATURE STATE: Kubernetes v1.12 [beta]

本文档介绍如何通过 [sysctl](#) 接口在 Kubernetes 集群中配置和使用内核参数。

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.

获取 Sysctl 的参数列表

在 Linux 中，管理员可以通过 sysctl 接口修改内核运行时的参数。在 /proc/sys/ 虚拟文件系统下存放许多内核参数。这些参数涉及了多个内核子系统，如：

- 内核子系统（通常前缀为：kernel.）
- 网络子系统（通常前缀为：net.）
- 虚拟内存子系统（通常前缀为：vm.）
- MDADM 子系统（通常前缀为：dev.）
- 更多子系统请参见[内核文档](#)。

若要获取完整的参数列表，请执行以下命令

```
sudo sysctl -a
```

启用非安全的 Sysctl 参数

sysctl 参数分为 安全 和 非安全的。安全 sysctl 参数除了需要设置恰当的命名空间外，在同一 node 上的不同 Pod 之间也必须是 相互隔离的。这意味着在 Pod 上设置 安全 sysctl 参数

- 必须不能影响到节点上的其他 Pod
- 必须不能损害节点的健康
- 必须不允许使用超出 Pod 的资源限制的 CPU 或内存资源。

至今为止，大多数 有命名空间的 sysctl 参数不一定被认为是 安全 的。以下几种 sysctl 参数是 安全 的：

- kernel.shm_rmid_forced
- net.ipv4.ip_local_port_range
- net.ipv4.tcp_syncookies
- net.ipv4.ping_group_range（从 Kubernetes 1.18 开始）

说明：示例中的 net.ipv4.tcp_syncookies 在Linux 内核 4.4 或更低的版本中是无命名空间的。

在未来的 Kubernetes 版本中，若 kubelet 支持更好的隔离机制，则上述列表中将会列出更多 安全 的 sysctl 参数。

所有 安全的 sysctl 参数都默认启用。

所有 非安全的 sysctl 参数都默认禁用，且必须由集群管理员在每个节点上手动开启。那些设置了不安全 sysctl 参数的 Pod 仍会被调度，但无法正常启动。

参考上述警告，集群管理员只有在一些非常特殊的情况下（如：高可用或实时应用调整），才可以启用特定的 非安全的 sysctl 参数。如需启用 非安全的 sysctl 参数，请你在每个节点上分别设置 kubelet 命令行参数，例如：

```
kubelet --allowed-unsafe-sysctls \
'kernel.msg*,net.core.somaxconn' ...
```

如果你使用 [Minikube](#)，可以通过 extra-config 参数来配置：

```
minikube start --extra-config="kubelet.allowed-unsafe-
sysctls=kernel.msg*,net.core.somaxconn" ...
```

只有 有命名空间的 sysctl 参数可以通过该方式启用。

设置 Pod 的 Sysctl 参数

目前，在 Linux 内核中，有许多的 sysctl 参数都是 有命名空间的。这就意味着可以为节点上的每个 Pod 分别去设置它们的 sysctl 参数。在 Kubernetes 中，只有那些有命名空间的 sysctl 参数可以通过 Pod 的 securityContext 对其进行配置。

以下列出有命名空间的 sysctl 参数，在未来的 Linux 内核版本中，此列表可能会发生变化。

- kernel.shm*,
- kernel.msg*,
- kernel.sem,
- fs.mqueue.*,
- net.* (内核中可以在容器命名空间里被更改的网络配置项相关参数)。然而也有一些特例（例如，net.netfilter.nf_conntrack_max 和 net.netfilter.nf_conntrack_expect_max 可以在容器命名空间里被更改，但它们是非命名空间的）。

没有命名空间的 sysctl 参数称为 节点级别的 sysctl 参数。如果需要对其进行设置，则必须在每个节点的操作系统上手动地去配置它们，或者通过在 DaemonSet 中运行特权模式容器来配置。

可使用 Pod 的 securityContext 来配置有命名空间的 sysctl 参数，securityContext 应用于同一个 Pod 中的所有容器。

此示例中，使用 Pod SecurityContext 来对一个安全的 sysctl 参数 kernel.shm_rmid_forced 以及两个非安全的 sysctl 参数 net.core.somaxconn 和 kernel.msgmax 进行设置。在 Pod 规约中对 安全的 和 非安全的 sysctl 参数不做区分。

警告：为了避免破坏操作系统的稳定性，请你在了解变更后果之后再修改 sysctl 参数。

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
      - name: kernel.shm_rmid_forced
        value: "0"
      - name: net.core.somaxconn
        value: "1024"
      - name: kernel.msgmax
        value: "65536"
...

```

警告：由于 [非安全的 sysctl](#) 参数其本身具有不稳定性，在使用 [非安全的 sysctl](#) 参数时可能会导致一些严重问题，如容器的错误行为、机器资源不足或节点被完全破坏，用户需自行承担风险。

最佳实践方案是将集群中具有特殊 sysctl 设置的节点视为 [有污点的](#)，并且只调度需要使用到特殊 sysctl 设置的 Pod 到这些节点上。建议使用 Kubernetes 的 [污点和容忍度特性](#) 来实现它。

设置了 [非安全的 sysctl](#) 参数的 Pod 在禁用了这两种 [非安全的 sysctl](#) 参数配置的节点上启动都会失败。与 [节点级别的 sysctl](#) 一样，建议开启 [污点和容忍度特性](#) 或 [为节点配置污点](#) 以便将 Pod 调度到正确的节点之上。

PodSecurityPolicy

你可以通过在 PodSecurityPolicy 的 `forbiddenSysctls` 和/或 `allowedUnsafeSysctls` 字段中，指定 sysctl 或填写 sysctl 匹配模式来进一步为 Pod 设置 sysctl 参数。sysctl 参数匹配模式以 * 字符结尾，如 `kernel.*`。单独的 * 字符匹配所有 sysctl 参数。

所有 [安全的 sysctl](#) 参数都默认启用。

`forbiddenSysctls` 和 `allowedUnsafeSysctls` 的值都是字符串列表类型，可以添加 sysctl 参数名称，也可以添加 sysctl 参数匹配模式（以*结尾）。只填写 * 则匹配所有的 sysctl 参数。

`forbiddenSysctls` 字段用于禁用特定的 sysctl 参数。你可以在列表中禁用安全和非安全的 sysctl 参数的组合。要禁用所有的 sysctl 参数，请设置为 *。

如果要在 `allowedUnsafeSysctls` 字段中指定一个非安全的 sysctl 参数，并且它在 `forbiddenSysctls` 字段中未被禁用，则可以在 Pod 中通过 PodSecurityPolicy 启用该 sysctl 参数。若要在 PodSecurityPolicy 中开启所有非安全的 sysctl 参数，请设 `allowedUnsafeSysctls` 字段值为 *。

`allowedUnsafeSysctls` 与 `forbiddenSysctls` 两字段的配置不能重叠，否则这就意味着存在某个 `sysctl` 参数既被启用又被禁用。

警告：如果你通过 `PodSecurityPolicy` 中的 `allowedUnsafeSysctls` 字段将非安全的 `sysctl` 参数列入白名单，但该 `sysctl` 参数未通过 `kubelet` 命令行参数 `--allowed-unsafe-sysctls` 在节点上将其列入白名单，则设置了这个 `sysctl` 参数的 Pod 将会启动失败。

以下示例设置启用了以 `kernel.msg` 为前缀的非安全的 `sysctl` 参数，同时禁用了 `sysctl` 参数 `kernel.shm_rmid_forced`。

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
    - kernel.msg*
  forbiddenSysctls:
    - kernel.shm_rmid_forced
  ...
```

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 22, 2020 at 11:34 AM PST: [Sync with the English version.](#) (`8febf98cb`)

在运行中的集群上重新配置节点的 `kubelet`

FEATURE STATE: Kubernetes v1.11 [beta]

[动态 `kubelet` 配置](#) 允许你在一个运行的 Kubernetes 集群上通过部署 ConfigMap 并配置每个节点来使用它来更改每个 `kubelet` 的配置，。

警告：所有 `kubelet` 配置参数都可以动态更改，但这对某些参数来说是不安全的。在决定动态更改参数之前，你需要深刻理解这种变化将如何影响你的集群的行为。在把一组变更推广到集群范围之前，需要在较小规模的节点集合上仔细地测试这些配置变化。与特定字段配置相关的建议可以在源码中 `KubeletConfiguration` [类型文档](#)中找到。

准备开始

你需要一个 Kubernetes 集群。你需要 v1.11 或更高版本的 kubectl，并以配置好与集群通信。要获知版本信息，请输入 `kubectl version`。你的集群 API 服务器版本（如 v1.12）不能比你所用的 kubectl 的版本差不止一个小版本号。例如，如果你的集群在运行 v1.16，那么你可以使用 v1.15、v1.16、v1.17 的 kubectl，所有其他的组合都是 [不支持的](#)。

某些例子中使用了命令行工具 [jq](#)。你并不一定需要 jq 才能完成这些任务，因为总是有一些手工替代的方式。

针对你所重新配置的每个节点，你必须设置 kubelet 的参数 `-dynamic-config-dir`，使之指向一个可写的目录。

重配置 集群中运行节点上的 kubelet

基本工作流程概述

在运行中的集群中配置 kubelet 的基本工作流程如下：

1. 编写一个 YAML 或 JSON 的配置文件包含 kubelet 的配置。
2. 将此文件包装在 ConfigMap 中并将其保存到 Kubernetes 控制平面。
3. 更新 kubelet 的相应节点对象以使用此 ConfigMap。

每个 kubelet 都会在其各自的节点对象上监测（Watch）配置引用。当引用更改时，kubelet 将下载新配置，更新本地引用以引用该文件，然后退出。要想使该功能正常地工作，你必须运行操作系统级别的服务管理器（如 systemd），在 kubelet 退出时将其重启。kubelet 重新启动时，将开始使用新配置。

这个新配置完全地覆盖 `--config` 所提供的配置，并被命令行标志覆盖。新配置中未指定的值将收到适合配置版本的默认值（e.g. `kubelet.config.k8s.io/v1beta1`），除非被命令行标志覆盖。

节点 kubelet 配置状态可通过 `node.spec.status.config` 获取。一旦你已经改变了一个节点去使用新的 ConfigMap，就可以观察此状态以确认该节点正在使用的预期配置。

本文用命令 `kubectl edit` 描述节点的编辑，还有一些其他的方式去修改节点的规约，包括更利于脚本化的工作流程的 `kubectl patch`。

本文仅仅讲述在单节点上使用每个 ConfigMap。请注意对于多个节点使用相同的 ConfigMap 也是合法的。

警告：通过就地更新 ConfigMap 来更改配置是 可能的。尽管如此，这样做会导致所有配置为使用该 ConfigMap 的 kubelet 被同时更新。更安全的做法是按惯例将 ConfigMap 视为不可变更的，借助于 kubectl 的 `--append-hash` 选项逐步把更新推广到 `node.spec.configSource`。

节点鉴权器的自动 RBAC 规则

以前，你需要手动创建 RBAC 规则以允许节点访问其分配的 ConfigMap。节点鉴权器现在能够自动配置这些规则。

生成包含当前配置的文件

动态 kubelet 配置特性允许你为整个配置对象提供一个重载配置，而不是靠单个字段的叠加。这是一个更简单的模型，可以更轻松地跟踪配置值的来源，更便于调试问题。然而，相应的代价是你必须首先了解现有配置，以确保你只更改你打算修改的字段。

组件 kubelet 从其配置文件中加载配置数据，不过你可以通过设置命令行标志 来重载文件中的一些配置。这意味着，如果你仅知道配置文件的内容，而你不知道 命令行重载了哪些配置，你就无法知道 kubelet 的运行时配置是什么。

因为你需要知道运行时所使用的配置才能重载之，你可以从 kubelet 取回其运行时配置。你可以通过访问 kubelet 的 configz 末端来生成包含节点当前配置的配置文件；这一操作可以通过 kubectl proxy 来完成。下一节解释如何完成这一操作。

注意：组件 kubelet 上的 configz 末端是用来协助调试的，并非 kubelet 稳定行为的一部分。请不要在产品环境下依赖此末端的行为，也不要在自动化工具中使用此末端。

关于如何使用配置文件来配置 kubelet 行为的更多信息可参见 [通过配置文件设置 kubelet 参数](#) 文档。

生成配置文件

说明：下面的任务步骤中使用了 jq 命令以方便处理 JSON 数据。为了完成这里讲述的任务，你需要安装 jq。如果你更希望手动提取 kubeletconfig 子对象，也可以对这里的对应步骤做一些调整。

1. 选择要重新配置的节点。在本例中，此节点的名称为 NODE_NAME。

2. 使用以下命令在后台启动 kubectl 代理：

```
kubectl proxy --port=8001 &
```

1. 运行以下命令从 configz 端点中下载并解压配置。这个命令很长，因此在复制粘贴时要小心。**如果你使用 zsh**，请注意常见的 zsh 配置要添加反斜杠转义 URL 中变量名称周围的大括号。例如：在粘贴时，\${NODE_NAME} 将被重写为 \\$\{NODE_NAME\}。你必须在运行命令之前删除反斜杠，否则命令将失败。

```
NODE_NAME="the-name-of-the-node-you-are-reconfiguring"; curl -sSL "http://localhost:8001/api/v1/nodes/${NODE_NAME}/proxy/configz" | jq '.kubeletConfig.config.kubeletConfig.k8s.io.v1beta1' > kubelet_configz_${NODE_NAME}
```

说明：你需要手动将 kind 和 apiVersion 添加到下载对象中，因为它们不是由 configz 末端 返回的。

修改配置文件

使用文本编辑器，改变上述操作生成的文件中一个参数。例如，你或许会修改 QPS 参数 eventRecordQPS。

把配置文件推送到控制平面

用以下命令把编辑后的配置文件推送到控制平面：

```
kubectl -n kube-system create configmap my-node-config \
--from-file=kubelet=kubelet_configz_${NODE_NAME} \
--append-hash -o yaml
```

下面是合法响应的一个例子：

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-09-14T20:23:33Z
  name: my-node-config-gkt4c2m4b2
  namespace: kube-system
  resourceVersion: "119980"
  selfLink: /api/v1/namespaces/kube-system/configmaps/my-node-config-
gkt4c2m4b2
  uid: 946d785e-998a-11e7-a8dd-42010a800006
data:
  kubelet: |
    {...}
```

你会在 kube-system 命名空间中创建 ConfigMap，因为 kubelet 是 Kubernetes 的系统组件。

--append-hash 选项给 ConfigMap 内容附加了一个简短校验和。这对于先编辑后推送的工作流程很方便，因为它自动并确定地为新 ConfigMap 生成新的名称。在以下示例中，包含生成的哈希字符串的对象名被称为 CONFIG_MAP_NAME。

配置节点使用新的配置

```
kubectl edit node ${NODE_NAME}
```

在你的文本编辑器中，在 spec 下增添以下 YAML：

```
configSource:
  configMap:
    name: CONFIG_MAP_NAME
```

```
namespace: kube-system
kubeletConfigKey: kubelet
```

你必须同时指定 name、 namespace 和 kubeletConfigKey 这三个属性。 kubeletConfigKey 这个参数通知 kubelet ConfigMap 中的哪个键下面包含所要的配置。

观察节点开始使用新配置

用 kubectl get node \${NODE_NAME} -o yaml 命令读取节点并检查 node.status.config 内容。 状态部分报告了对应 active (使用中的) 配置、 assigned (被赋予的) 配置和 lastKnownGood (最近已知可用的) 配置的配置源。

- active 是 kubelet 当前运行时所使用的版本。
- assigned 参数是 kubelet 基于 node.spec.configSource 所解析出来的最新版本。
- lastKnownGood 参数是 kubelet 的回退版本；如果在 node.spec.configSource 中包含了无效的配置值，kubelet 可以回退到这个版本。

如果用本地配置部署节点，使其设置成默认值，这个 lastKnownGood 配置可能不存在。 在 kubelet 配置好后，将更新 lastKnownGood 为一个有效的 assigned 配置。 决定如何确定某配置成为 lastKnownGood 配置的细节并不在 API 保障范畴，不过目前实现中采用了 10 分钟的宽限期。

你可以使用以下命令（使用 jq ）过滤出配置状态：

```
kubectl get no ${NODE_NAME} -o json | jq '.status.config'
```

以下是一个响应示例：

```
{
  "active": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  },
  "assigned": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  }
},
```

```
"lastKnownGood": {
  "configMap": {
    "kubeletConfigKey": "kubelet",
    "name": "my-node-config-9mbkccg2cc",
    "namespace": "kube-system",
    "resourceVersion": "1326",
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
  }
}
```

如果你没有安装 jq , 你可以查看整个响应对象 , 查找其中的 node.status.config 部分。

如果发生错误 , kubelet 会在 node.status.config.error 中显示出错误信息的结构体。可能的错误列在[了解节点配置错误信息](#)节。 你可以在 kubelet 日志中搜索相同的文本以获取更多详细信息和有关错误的上下文。

做出更多的改变

按照下面的工作流程做出更多的改变并再次推送它们。 你每次推送一个 ConfigMap 的新内容时 , kubectl 的 --append-hash 选项都会给 ConfigMap 创建一个新的名称。最安全的上线策略是首先创建一个新的 ConfigMap , 然后更新节点以使用新的 ConfigMap。

重置节点以使用其本地默认配置

要重置节点 , 使其使用节点创建时使用的配置 , 可以用 kubectl edit node \${NODE_NAME} 命令编辑节点 , 并删除 node.spec.configSource 字段。

观察节点正在使用本地默认配置

在删除此字段后 , node.status.config 最终变成空 , 所有配置源都已重置为 nil。 这表示本地默认配置成为了 assigned、active 和 lastKnownGood 配置 , 并且没有报告错误。

kubectl patch 示例

你可以使用几种不同的机制来更改节点的 configSource。

本例使用kubectl patch :

```
kubectl patch node ${NODE_NAME} -p "{\"spec\":{\"configSource\":
{\\"configMap\\":{\\\"name\\\":\"${CONFIG_MAP_NAME}\\\",\\\"namespace\\\":\"kube-
system\\\",\\\"kubeletConfigKey\\\":\\\"kubelet\\\"}}}}"
```

了解 Kubelet 如何为配置生成检查点

当为节点赋予新配置时，kubelet 会下载并解压配置负载为本地磁盘上的一组文件。 kubelet 还记录一些元数据，用以在本地跟踪已赋予的和最近已知良好的配置源，以便 kubelet 在重新启动时知道使用哪个配置，即使 API 服务器变为不可用。 在为配置信息和相关元数据生成检查点之后，如果检测到已赋予的配置发生改变，则 kubelet 退出。 当 kubelet 被 OS 级服务管理器（例如 systemd）重新启动时，它会读取新的元数据并使用新配置。

当记录的元数据已被完全解析时，意味着它包含选择一个指定的配置版本所需的所有信息 -- 通常是 UID 和 ResourceVersion。 这与 node.spec.configSource 形成对比，后者通过幂等的 namespace/name 声明来标识目标 ConfigMap；kubelet 尝试使用此 ConfigMap 的最新版本。

当你在调试节点上问题时，可以检查 kubelet 的配置元数据和检查点。 kubelet 的检查点目录结构是：

```
- --dynamic-config-dir (用于管理动态配置的根目录)
|-- meta
| - assigned (编码后的 kubeletconfig/v1beta1.SerializedNodeConfigSource 对象，对应赋予的配置)
|   | - last-known-good (编码后的 kubeletconfig/v1beta1.SerializedNodeConfigSource 对象，对应最近已知可用配置)
| - checkpoints
|   | - uid1 (用 uid1 来标识的对象版本目录)
|     | - resourceVersion1 (uid1 对象 resourceVersion1 版本下所有解压文件的目录)
|     | - ...
|   | - ...
| - ...
```

了解节点配置错误信息

下表描述了使用动态 kubelet 配置时可能发生的错误消息。 你可以在 kubelet 日志中搜索相同的文本来获取有关错误的其他详细信息和上下文。

错误信息	可能的原因
failed to load config, see Kubelet log for details	kubelet 可能无法解析下载配置的有效负载，或者当尝试从磁盘中加载有效负载时，遇到文件系统错误。
failed to validate config, see Kubelet log for details	有效负载中的配置，与命令行标志所产生的覆盖配置以及特行门控的组合、配置文件本身、远程负载被 kubelet 判定为无效。
invalid NodeConfigSource, exactly one subfield must be non-nil, but all were nil	由于 API 服务器负责对 node.spec.configSource 执行验证，检查其中是否包含至少一个非空子字段，这个消息可能意味着 kubelet 比 API 服务器版本低，因而无法识别更新的源类型。

错误信息	可能的原因
failed to sync: failed to download config, see Kubelet log for details	kubelet 无法下载配置数据。可能是 node.spec.configSource 无法解析为具体的 API 对象，或者网络错误破坏了下载。处于此错误状态时，kubelet 将重新尝试下载。
failed to sync: internal failure, see Kubelet log for details	kubelet 遇到了一些内部问题，因此无法更新其配置。例如：发生文件系统错误或无法从内部缓存中读取对象。
internal failure, see Kubelet log for details	在对配置进行同步的循环之外操作配置时，kubelet 遇到了一些内部问题。

接下来

- 关于如何通过配置文件来配置 kubelet 的更多细节信息，可参阅 [使用配置文件设置 kubelet 参数](#)。
- 阅读 API 文档中 [NodeConfigSource](#) 说明

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 14, 2020 at 11:02 PM PST: [\[zh\] Rework the reconfig kubelet task \(b7f82b11b\)](#)

声明网络策略

本文可以帮助你开始使用 Kubernetes 的 [NetworkPolicy API](#) 声明网络策略去管理 Pod 之间的通信

注意：本部分链接到提供 Kubernetes 所需功能的第三方项目。

Kubernetes 项目作者不负责这些项目。此页面遵循[CNCF 网站指南](#)，按字母顺序列出项目。要将项目添加到此列表中，请在提交更改之前阅读[内容指南](#)。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.8. 要获知版本信息，请输入 `kubectl version`.

你首先需要有一个支持网络策略的 Kubernetes 集群。已经有许多支持 NetworkPolicy 的网络提供商，包括：

- [Calico](#)
- [Cilium](#)
- [Kube-router](#)
- [Romana](#)
- [Weave 网络](#)

创建一个nginx Deployment 并且通过服务将其暴露

为了查看 Kubernetes 网络策略是怎样工作的，可以从创建一个nginx Deployment 并且通过服务将其暴露开始

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

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

将此 Deployment 以名为 nginx 的 Service 暴露出来：

```
kubectl expose deployment nginx --port=80
```

```
service/nginx exposed
```

上述命令创建了一个带有一个 nginx 的 Deployment，并将之通过名为 nginx 的 Service 暴露出来。名为 nginx 的 Pod 和 Deployment 都位于 default 名字空间内。

```
kubectl get svc,pod
```

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/kubernetes	10.100.0.1	<none>	443/TCP	46m
svc/nginx	10.100.0.16	<none>	80/TCP	33s

NAME	READY	STATUS	RESTARTS	AGE
po/nginx-701339712-e0qfq	1/1	Running	0	35s

通过从 Pod 访问服务对其进行测试

你应该可以从其它的 Pod 访问这个新的 nginx 服务。要从 default 命名空间中的其它 Pod 来访问该服务。可以启动一个 busybox 容器：

```
kubectl run busybox --rm -ti --image=busybox /bin/sh
```

在你的 Shell 中，运行下面的命令：

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

限制 nginx 服务的访问

如果想限制对 nginx 服务的访问，只让那些拥有标签 access: true 的 Pod 访问它，那么可以创建一个如下所示的 NetworkPolicy 对象：

[service/networking/nginx-policy.yaml](#)



```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

NetworkPolicy 对象的名称必须是一个合法的 [DNS 子域名](#).

说明： NetworkPolicy 中包含选择策略所适用的 Pods 集合的 podSelector。你可以看到上面的策略选择的是带有标签 app=nginx 的 Pods。此标签是被自动添加到 nginx Deployment 中的 Pod 上的。如果 podSelector 为空，则意味着选择的是名字空间中的所有 Pods。

为服务指定策略

使用 kubectl 根据上面的 nginx-policy.yaml 文件创建一个 NetworkPolicy：

```
kubectl apply -f https://k8s.io/examples/service/networking/nginx-policy.yaml
networkpolicy.networking.k8s.io/access-nginx created
```

测试没有定义访问标签时访问服务

如果你尝试从没有设定正确标签的 Pod 中去访问 nginx 服务，请求将会超时：

```
kubectl run busybox --rm -ti --image=busybox -- /bin/sh
```

在 Shell 中运行命令：

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
wget: download timed out
```

定义访问标签后再次测试

创建一个拥有正确标签的 Pod，你将看到请求是被允许的：

```
kubectl run busybox --rm -ti --labels="access=true" --image=busybox -- /bin/sh
```

在 Shell 中运行命令：

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

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 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

安全地清空一个节点

本页展示了如何在确保 PodDisruptionBudget 的前提下，安全地清空一个[节点](#)。

准备开始

您的 Kubernetes 服务器版本必须不低于版本 1.5. 要获知版本信息，请输入 `kubectl version`.

此任务假定你已经满足了以下先决条件：

- 使用的 Kubernetes 版本 ≥ 1.5 。
- 以下两项，具备其一：
 1. 在节点清空期间，不要求应用程序具有高可用性
 2. 你已经了解了 [PodDisruptionBudget 的概念](#)，并为需要它的应用程序[配置了 PodDisruptionBudget](#)。

(可选) 配置干扰预算

为了确保你的负载在维护期间仍然可用，你可以配置一个 [PodDisruptionBudget](#)。如果可用性对于正在清空的该节点上运行或可能在该节点上运行的任何应用程序很重要，首先 [配置一个 PodDisruptionBudgets](#) 并继续遵循本指南。

使用 kubectl drain 从服务中删除一个节点

在对节点执行维护（例如内核升级、硬件维护等）之前，可以使用 kubectl drain 从节点安全地逐出所有 Pods。安全的驱逐过程允许 Pod 的容器 [体面地终止](#)，并确保满足指定的 PodDisruptionBudgets。

说明：默认情况下，kubectl drain 将忽略节点上不能杀死的特定系统 Pod；有关更多细节，请参阅 [kubectl drain 文档](#)。

kubectl drain 的成功返回，表明所有的 Pods（除了上一段中描述的被排除的那些），已经被安全地逐出（考虑到期望的终止宽限期和你定义的 PodDisruptionBudget）。然后就可以安全地关闭节点，比如关闭物理机器的电源，如果它运行在云平台上，则删除它的虚拟机。

首先，确定想要清空的节点的名称。可以用以下命令列出集群中的所有节点：

```
kubectl get nodes
```

接下来，告诉 Kubernetes 清空节点：

```
kubectl drain <node name>
```

一旦它返回（没有报错），你就可以下电此节点（或者等价地，如果在云平台上，删除支持该节点的虚拟机）。如果要在维护操作期间将节点留在集群中，则需要运行：

```
kubectl uncordon <node name>
```

然后告诉 Kubernetes，它可以继续在此节点上调度新的 Pods。

并行清空多个节点

kubectl drain 命令一次只能发送给一个节点。但是，你可以在不同的终端或后台为不同的节点并行地运行多个 kubectl drain 命令。同时运行的多个 drain 命令仍然遵循你指定的 PodDisruptionBudget。

例如，如果你有一个三副本的 StatefulSet，并设置了一个 PodDisruptionBudget，指定 minAvailable: 2。如果所有的三个 Pod 均就绪，并且你并行地发出多个 drain 命令，那么 kubectl drain 只会从 StatefulSet 中逐出一个 Pod，因为 Kubernetes 会遵守 PodDisruptionBudget 并确保在任何时候只有一个 Pod 不可用（最多不可用 Pod 个数的计算方法：replicas - minAvailable）。任何会导致就绪副本数量低于指定预算的清空操作都将被阻止。

驱逐 API

如果你不喜欢使用 [kubectl drain](#) (比如避免调用外部命令，或者更细化地控制 pod 驱逐过程)，你也可以用驱逐 API 通过编程的方式达到驱逐的效果。

首先应该熟悉使用 [Kubernetes 语言客户端](#)。

Pod 的 Eviction 子资源可以看作是一种策略控制的 DELETE 操作，作用于 Pod 本身。要尝试驱逐（更准确地说，尝试 创建一个 Eviction），需要用 POST 发出所尝试的操作。这里有一个例子：

```
{  
  "apiVersion": "policy/v1beta1",  
  "kind": "Eviction",  
  "metadata": {  
    "name": "quux",  
    "namespace": "default"  
  }  
}
```

你可以使用 curl 尝试驱逐：

```
curl -v -H 'Content-type: application/json' http://127.0.0.1:8080/api/v1/namespaces/default/pods/quux/eviction -d @eviction.json
```

API 可以通过以下三种方式之一进行响应：

- 如果驱逐被授权，那么 Pod 将被删掉，并且你会收到 200 OK，就像你向 Pod 的 URL 发送了 DELETE 请求一样。
- 如果按照预算中规定，目前的情况不允许的驱逐，你会收到 429 Too Many Requests。这通常用于对一些请求进行通用速率限制，但这里我们的意思是：此请求 现在 不允许，但以后可能会允许。目前，调用者不会得到任何 Retry-After 的提示，但在将来的版本中可能会得到。
- 如果有一些错误的配置，比如多个预算指向同一个 Pod，你将得到 500 Internal Server Error。

对于一个给定的驱逐请求，有两种情况：

- 没有匹配这个 Pod 的预算。这种情况，服务器总是返回 200 OK。
- 至少匹配一个预算。在这种情况下，上述三种回答中的任何一种都可能适用。

驱逐阻塞

在某些情况下，应用程序可能会到达一个中断状态，除了 429 或 500 之外，它将永远不会返回任何内容。例如 ReplicaSet 创建的替换 Pod 没有变成就绪状态，或者被驱逐的最后一个 Pod 有很长的终止宽限期，就会发生这种情况。

在这种情况下，有两种可能的解决方案：

- 中止或暂停自动操作。调查应用程序卡住的原因，并重新启动自动化。
- 经过适当的长时间等待后，从集群中删除 Pod 而不是使用驱逐 API。

Kubernetes 并没有具体说明在这种情况下应该采取什么行为，这应该由应用程序所有者和集群所有者紧密沟通，并达成对行动一致意见。

接下来

- 执行[配置 PDB](#)中的各个步骤，保护你的应用
- 进一步了解[节点维护](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 17, 2020 at 11:24 PM PST: [Update safely-drain-node.md \(88dae3183\)](#)

开发云控制器管理器

FEATURE STATE: Kubernetes v1.11 [beta]

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

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

背景

由于云驱动的开发和发布与 Kubernetes 项目本身步调不同，将特定于云环境的代码抽象到 cloud-controller-manager 二进制组件有助于云厂商独立于 Kubernetes 核心代码推进其驱动开发。

Kubernetes 项目提供 cloud-controller-manager 的框架代码，其中包含 Go 语言的接口，便于你（或者你的云驱动提供者）接驳你自己的实现。这意味着每个云驱动可以通过从 Kubernetes 核心代码导入软件包来实现一个 cloud-controller-manager；每个云驱动会通过调用 cloudprovider.RegisterCloudProvider 接口来注册其自身实现代码，从而更新记录可用云驱动的全局变量。

开发

Out of Tree

要为你的云环境构建一个 out-of-tree 云控制器管理器：

1. 使用满足 [cloudprovider.Interface](#) 的实现创建一个 Go 语言包。
2. 使用来自 Kubernetes 核心代码库的 [cloud-controller-manager 中的 main.go](#) 作为 main.go 的模板。如上所述，唯一的区别应该是将导入的云包。
3. 在 main.go 中导入你的云包，确保你的包有一个 init 块来运行 [cloudprovider.RegisterCloudProvider](#)。

很多云驱动都将其控制器管理器代码以开源代码的形式公开。如果你在开发一个新的 cloud-controller-manager，你可以选择某个 out-of-tree 云控制器管理器作为出发点。

In Tree

对于 in-tree 驱动，你可以将 in-tree 云控制器管理器作为群集中的 [Daemonset](#) 来运行。有关详细信息，请参阅[云控制器管理器管理](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 05, 2021 at 1:24 AM PST: [\[zh\] Cloud Controller Manager: fix a broken link \(f0e7fc716\)](#)

开启服务拓扑

本页面提供了在 Kubernetes 中启用服务拓扑的概述。

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.

介绍

服务拓扑 (*Service Topology*) 使服务能够根据集群中的 Node 拓扑来路由流量。比如，服务可以指定将流量优先路由到与客户端位于同一节点或者同一可用区域的端点上。

先决条件

需要下面列的先决条件，才能启用拓扑感知的服务路由：

- Kubernetes 1.17 或更新版本
- [Kube-proxy](#) 以 iptables 或者 IPVS 模式运行
- 启用[端点切片](#)

启用服务拓扑

FEATURE STATE: Kubernetes v1.17 [alpha]

要启用服务拓扑功能，需要为所有 Kubernetes 组件启用 ServiceTopology 和 EndpointSlice 特性门控：

```
--feature-gates="ServiceTopology=true,EndpointSlice=true"
```

接下来

- 阅读[服务拓扑](#)概念
- 阅读[端点切片](#)
- 阅读[通过服务来连接应用](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 June 30, 2020 at 5:24 PM PST: [translate into chinese \(268492a36\)](#)

控制节点上的 CPU 管理策略

FEATURE STATE: Kubernetes v1.12 [beta]

按照设计，Kubernetes 对 pod 执行相关的很多方面进行了抽象，使得用户不必关心。然而，为了正常运行，有些工作负载要求在延迟和/或性能方面有更强的保证。为此，kubelet 提供方法来实现更复杂的负载放置策略，同时保持抽象，避免显式的放置指令。

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.

CPU 管理策略

默认情况下，kubelet 使用 [CFS 配额](#) 来执行 Pod 的 CPU 约束。当节点上运行了很多 CPU 密集的 Pod 时，工作负载可能会迁移到不同的 CPU 核，这取决于调度时 Pod 是否被扼制，以及哪些 CPU 核是可用的。许多工作负载对这种迁移不敏感，因此无需任何干预即可正常工作。

然而，有些工作负载的性能明显地受到 CPU 缓存亲和性以及调度延迟的影响。对此，kubelet 提供了可选的 CPU 管理策略，来确定节点上的一些分配偏好。

配置

CPU 管理策略通过 kubelet 参数 --cpu-manager-policy 来指定。支持两种策略：

- none: 默认策略，表示现有的调度行为。
- static: 允许为节点上具有某些资源特征的 pod 赋予增强的 CPU 亲和性和独占性。

CPU 管理器定期通过 CRI 写入资源更新，以保证内存中 CPU 分配与 cgroupfs 一致。同步频率通过新增的 Kubelet 配置参数 --cpu-manager-reconcile-period 来设置。如果不指定，默认与 --node-status-update-frequency 的周期相同。

none 策略

none 策略显式地启用现有的默认 CPU 亲和方案，不提供操作系统调度器默认行为之外的亲和性策略。通过 CFS 配额来实现 [Guaranteed pods](#) 的 CPU 使用限制。

static 策略

static 策略针对具有整数型 CPU requests 的 Guaranteed Pod，它允许该类 Pod 中的容器访问节点上的独占 CPU 资源。这种独占性是使用 [cpuset cgroup 控制器](#) 来实现的。

说明：诸如容器运行时和 kubelet 本身的系统服务可以继续在这些独占 CPU 上运行。独占性仅针对其他 Pod。

说明：CPU 管理器不支持运行时下线和上线 CPUs。此外，如果节点上的在线 CPUs 集合发生变化，则必须驱逐节点上的 Pod，并通过删除 kubelet 根目录中的状态文件 cpu_manager_state 来手动重置 CPU 管理器。

该策略管理一个共享 CPU 资源池，最初，该资源池包含节点上所有的 CPU 资源。可用的独占性 CPU 资源数量等于节点的 CPU 总量减去通过 `--kube-reserved` 或 `--system-reserved` 参数保留的 CPU。从1.17版本开始，CPU保留列表可以通过 kublet 的 '`--reserved-cpus`' 参数显式地设置。通过 '`--reserved-cpus`' 指定的显式CPU列表优先于使用 '`--kube-reserved`' 和 '`--system-reserved`' 参数指定的保留CPU。通过这些参数预留的 CPU 是以整数方式，按物理内核 ID 升序从初始共享池获取的。共享池是 BestEffort 和 Burstable pod 运行的 CPU 集合。Guaranteed pod 中的容器，如果声明了非整数值的 CPU requests，也将运行在共享池的 CPU 上。只有 Guaranteed pod 中，指定了整数型 CPU requests 的容器，才会被分配独占 CPU 资源。

说明：当启用 static 策略时，要求使用 `--kube-reserved` 和/或 `--system-reserved` 或 `--reserved-cpus` 来保证预留的 CPU 值大于零。这是因为零预留 CPU 值可能使得共享池变空。

当 Guaranteed Pod 调度到节点上时，如果其容器符合静态分配要求，相应的 CPU 会被从共享池中移除，并放置到容器的 cpuset 中。因为这些容器所使用的 CPU 受到调度域本身的限制，所以不需要使用 CFS 配额来进行 CPU 的绑定。换言之，容器 cpuset 中的 CPU 数量与 Pod 规约中指定的整数型 CPU limit 相等。这种静态分配增强了 CPU 亲和性，减少了 CPU 密集的工作负载在节流时引起的上下文切换。

考虑以下 Pod 规格的容器：

```
spec:  
  containers:  
    - name: nginx  
      image: nginx
```

该 Pod 属于 BestEffort QoS 类型，因为其未指定 requests 或 limits 值。所以该容器运行在共享 CPU 池中。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:  
        limits:  
          memory: "200Mi"  
        requests:  
          memory: "100Mi"
```

该 Pod 属于 Burstable QoS 类型，因为其资源 requests 不等于 limits，且未指定 cpu 数量。所以该容器运行在共享 CPU 池中。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:
```

```
limits:  
  memory: "200Mi"  
  cpu: "2"  
requests:  
  memory: "100Mi"  
  cpu: "1"
```

该 pod 属于 Burstable QoS 类型，因为其资源 requests 不等于 limits。所以该容器运行在共享 CPU 池中。

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

该 Pod 属于 Guaranteed QoS 类型，因为其 requests 值与 limits 相等。同时，容器对 CPU 资源的限制值是一个大于或等于 1 的整数值。所以，该 nginx 容器被赋予 2 个独占 CPU。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:  
        limits:  
          memory: "200Mi"  
          cpu: "1.5"  
        requests:  
          memory: "200Mi"  
          cpu: "1.5"
```

该 Pod 属于 Guaranteed QoS 类型，因为其 requests 值与 limits 相等。但是容器对 CPU 资源的限制值是一个小数。所以该容器运行在共享 CPU 池中。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:  
        limits:
```

```
memory: "200Mi"
cpu: "2"
```

该 Pod 属于 Guaranteed QoS 类型，因其指定了 limits 值，同时当未显式指定时， requests 值被设置为与 limits 值相等。同时，容器对 CPU 资源的限制值是一个大于或等于 1 的整数值。所以，该 nginx 容器被赋予 2 个独占 CPU。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 15, 2020 at 11:30 AM PST: [\[zh\] Tidy up and fix links in tasks section \(7/10\) \(30423d108\)](#)

控制节点上的拓扑管理策略

FEATURE STATE: Kubernetes v1.18 [beta]

越来越多的系统利用 CPU 和硬件加速器的组合来支持对延迟要求较高的任务和高吞吐量的并行计算。这类负载包括电信、科学计算、机器学习、金融服务和数据分析等。此类混合系统即用于构造这些高性能环境。

为了获得最佳性能，需要进行与 CPU 隔离、内存和设备局部性有关的优化。但是，在 Kubernetes 中，这些优化由各自独立的组件集合来处理。

拓扑管理器 (*Topology Manager*) 是一个 kubelet 的一部分，旨在协调负责这些优化的一组组件。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.18. 要获知版本信息，请输入 kubectl version.

拓扑管理器如何工作

在引入拓扑管理器之前，Kubernetes 中的 CPU 和设备管理器相互独立地做出资源分配决策。这可能会导致在多处理系统上出现并非期望的资源分配；由于这些与期望相左的分配，对性能或延迟敏感的应用将受到影响。这里的不符合期望意指，例如，CPU 和设备是从不同的 NUMA 节点分配的，因此会导致额外的延迟。

拓扑管理器是一个 Kubelet 组件，扮演信息源的角色，以便其他 Kubelet 组件可以做出与拓扑结构相对应的资源分配决定。

拓扑管理器为组件提供了一个称为 *建议供应者* (*Hint Providers*) 的接口，以发送和接收拓扑信息。拓扑管理器具有一组节点级策略，具体说明如下。

拓扑管理器从 *建议提供者* 接收拓扑信息，作为表示可用的 NUMA 节点和首选分配指示的位掩码。拓扑管理器策略对所提供的建议执行一组操作，并根据策略对提示进行约减以得到最优解；如果存储了与预期不符的建议，则该建议的优选字段将被设置为 `false`。在当前策略中，首选的是最窄的优先掩码。所选建议将被存储为拓扑管理器的一部分。取决于所配置的策略，所选建议可用来决定节点接受或拒绝 Pod。之后，建议会被存储在拓扑管理器中，供 *建议提供者* 进行资源分配决策时使用。

启用拓扑管理器功能特性

对拓扑管理器的支持要求启用 TopologyManager 特性门控。从 Kubernetes 1.18 版本开始，这一特性默认是启用的。

拓扑管理器策略

拓扑管理器目前：

- 对所有 QoS 类的 Pod 执行对齐操作
- 针对建议提供者所提供的拓扑建议，对请求的资源进行对齐

如果满足这些条件，则拓扑管理器将对齐请求的资源。

说明：为了将 Pod 规约中的 CPU 资源与其他请求资源对齐，CPU 管理器需要被启用并且节点上应配置了适当的 CPU 管理器策略。参看[控制 CPU 管理策略](#)。

拓扑管理器支持四种分配策略。你可以通过 Kubelet 标志 `--topology-manager-policy` 设置策略。所支持的策略有四种：

- `none` (默认)
- `best-effort`
- `restricted`
- `single-numa-node`

none 策略

这是默认策略，不执行任何拓扑对齐。

best-effort 策略

对于 Guaranteed 类的 Pod 中的每个容器，具有 best-effort 拓扑管理策略的 kubelet 将调用每个建议提供者以确定资源可用性。使用此信息，拓扑管理器存储该容器的首选 NUMA 节点亲和性。如果亲和性不是首选，则拓扑管理器将存储该亲和性，并且无论如何都将 pod 接纳到该节点。

之后 建议提供者 可以在进行资源分配决策时使用这个信息。

restricted 策略

对于 Guaranteed 类 Pod 中的每个容器，配置了 restricted 拓扑管理策略的 kubelet 调用每个建议提供者以确定其资源可用性。。使用此信息，拓扑管理器存储该容器的首选 NUMA 节点亲和性。如果亲和性不是首选，则拓扑管理器将从节点中拒绝此 Pod 。这将导致 Pod 处于 Terminated 状态，且 Pod 无法被节点接纳。

一旦 Pod 处于 Terminated 状态，Kubernetes 调度器将不会尝试重新调度该 Pod。建议使用 ReplicaSet 或者 Deployment 来重新部署 Pod。还可以通过实现外部控制环，以启动对具有 Topology Affinity 错误的 Pod 的重新部署。

如果 Pod 被允许运行在某节点，则 建议提供者 可以在做出资源分配决定时使用此信息。

single-numa-node 策略

对于 Guaranteed 类 Pod 中的每个容器，配置了 single-numa-node 拓扑管理策略的 kubelet 调用每个建议提供者以确定其资源可用性。使用此信息，拓扑管理器确定单 NUMA 节点亲和性是否可能。如果是这样，则拓扑管理器将存储此信息，然后 建议提供者 可以在做出资源分配决定时使用此信息。如果不可能，则拓扑管理器将拒绝 Pod 运行于该节点。这将导致 Pod 处于 Terminated 状态，且 Pod 无法被节点接受。

一旦 Pod 处于 Terminated 状态，Kubernetes 调度器将不会尝试重新调度该 Pod。建议使用 ReplicaSet 或者 Deployment 来重新部署 Pod。还可以通过实现外部控制环，以触发具有 Topology Affinity 错误的 Pod 的重新部署。

Pod 与拓扑管理器策略的交互

考虑以下 pod 规范中的容器：

```
spec:  
  containers:  
    - name: nginx  
      image: nginx
```

该 Pod 以 BestEffort QoS 类运行，因为没有指定资源 requests 或 limits。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx
```

```
resources:  
  limits:  
    memory: "200Mi"  
  requests:  
    memory: "100Mi"
```

由于 requests 数少于 limits，因此该 Pod 以 Burstable QoS 类运行。

如果选择的策略是 none 以外的任何其他策略，拓扑管理器都会评估这些 Pod 的规范。拓扑管理器会咨询建议提供者，获得拓扑建议。若策略为 static，则 CPU 管理器策略会返回默认的拓扑建议，因为这些 Pod 并没有显式地请求 CPU 资源。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:  
        limits:  
          memory: "200Mi"  
          cpu: "2"  
          example.com/device: "1"  
        requests:  
          memory: "200Mi"  
          cpu: "2"  
          example.com/device: "1"
```

此 Pod 以 Guaranteed QoS 类运行，因为其 requests 值等于 limits 值。

```
spec:  
  containers:  
    - name: nginx  
      image: nginx  
      resources:  
        limits:  
          example.com/deviceA: "1"  
          example.com/deviceB: "1"  
        requests:  
          example.com/deviceA: "1"  
          example.com/deviceB: "1"
```

因为未指定 CPU 和内存请求，所以 Pod 以 BestEffort QoS 类运行。

拓扑管理器将考虑以上两个 Pod。拓扑管理器将咨询建议提供者即 CPU 和设备管理器，以获取 Pod 的拓扑提示。对于 Guaranteed 类的 CPU 请求数为整数的 Pod，static CPU 管理器策略将返回与 CPU 请求有关的提示，而设备管理器将返回有关所请求设备的提示。

对于 Guaranteed 类的 CPU 请求可共享的 Pod , static CPU 管理器策略将返回默认的拓扑提示 , 因为没有排他性的 CPU 请求 ; 而设备管理器则针对所请求的设备返回有关提示。

在上述两种 Guaranteed Pod 的情况下 , none CPU 管理器策略会返回默认的拓扑提示。

对于 BestEffort Pod , 由于没有 CPU 请求 , static CPU 管理器策略将发送默认提示 , 而设备管理器将为每个请求的设备发送提示。

基于此信息 , 拓扑管理器将为 Pod 计算最佳提示并存储该信息 , 并且供 提示提供程序在进行资源分配时使用。

已知的局限性

1. 拓扑管理器所能处理的最大 NUMA 节点个数是 8。若 NUMA 节点数超过 8 , 枚举可能的 NUMA 亲和性并为之生成提示时会发生状态爆炸。
2. 调度器不支持拓扑功能 , 因此可能会由于拓扑管理器的原因而在节点上进行调度 , 然后在该节点上调度失败。
3. 设备管理器和 CPU 管理器时能够采纳拓扑管理器 HintProvider 接口的唯一两个组件。这意味着 NUMA 对齐只能针对 CPU 管理器和设备管理器所管理的资源实现。内存和大页面在拓扑管理器决定 NUMA 对齐时都还不会被考虑在内。

反馈

此页是否对您有帮助 ?

是 否

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

最后修改 August 15, 2020 at 8:24 PM PST: [\[zh\] Resync TopologyManager task \(0f97bf3c6\)](#)

搭建高可用的 Kubernetes Masters

FEATURE STATE: Kubernetes 1.5 [alpha]

你可以在谷歌计算引擎 (GCE) 的 kubeup 或 kube-down 脚本中复制 Kubernetes Master。本文描述了如何使用 kube-up/down 脚本来管理高可用 (HA) 的 Master , 以及如何使用 GCE 实现高可用控制节点。

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.

启动一个兼容高可用的集群

要创建一个新的兼容高可用的集群，你必须在 kubeup 脚本中设置以下标志：

- MULTIZONE=true - 为了防止从不同于服务器的默认区域的区域中删除 kubelets 副本。如果你希望在不同的区域运行副本，那么这一项是必需并且推荐的。
- ENABLE_ETCD_QUORUM_READ=true - 确保从所有 API 服务器读取数据时将返回最新的数据。如果为 true，读操作将被定向到主 etcd 副本。可以选择将这个值设置为 true，那么读取将更可靠，但也会更慢。

你还可以指定一个 GCE 区域，在这里创建第一个主节点副本。设置以下标志：

- KUBE_GCE_ZONE=zone - 将运行第一个主节点副本的区域。

下面的命令演示在 GCE europe-west1-b 区域中设置一个兼容高可用的集群：

```
MULTIZONE=true KUBE_GCE_ZONE=europe-west1-b ENABLE_ETCD_QUORUM_READS=true ./cluster/kube-up.sh
```

注意，上面的命令创建一个只有单一主节点的集群；但是，你可以使用后续命令将新的主节点副本添加到集群中。

增加一个新的主节点副本

在创建了兼容高可用的集群之后，可以向其中添加主节点副本。你可以使用带有如下标记的 kubeup 脚本添加主节点副本：

- KUBE_REPLICATE_EXISTING_MASTER=true - 创建一个已经存在的主节点的副本。
- KUBE_GCE_ZONE=zone - 主节点副本将运行的区域。必须与其他副本位于同一区域。

你无需设置 MULTIZONE 或 ENABLE_ETCD_QUORUM_READS 标志，因为他们可以从兼容高可用的集群中继承。

使用下面的命令可以复制现有兼容高可用的集群上的 Master：

```
KUBE_GCE_ZONE=europe-west1-c KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

删除主节点副本

你可以使用一个 `kube-down` 脚本从高可用集群中删除一个主节点副本，并可以使用以下标记：

- `KUBE_DELETE_NODES=false` - 限制删除 kubelets。
- `KUBE_GCE_ZONE=zone` - 将移除主节点副本的区域。
- `KUBE_REPLICA_NAME=replica_name` - (可选) 要删除的主节点副本的名称。
如果为空：将删除给定区域中的所有副本。

使用下面的命令可以从一个现有的高可用集群中删除一个 Master 副本：

```
KUBE_DELETE_NODES=false KUBE_GCE_ZONE=europe-west1-c ./cluster/kube-down.sh
```

处理主节点副本失败

如果高可用集群中的一个主节点副本失败，最佳实践是从集群中删除副本，并在相同的区域中添加一个新副本。下面的命令演示了这个过程：

1. 删除失败的副本：

```
KUBE_DELETE_NODES=false KUBE_GCE_ZONE=replica_zone KUBE_REPLICA_NAME=replica_name ./cluster/kube-down.sh
```

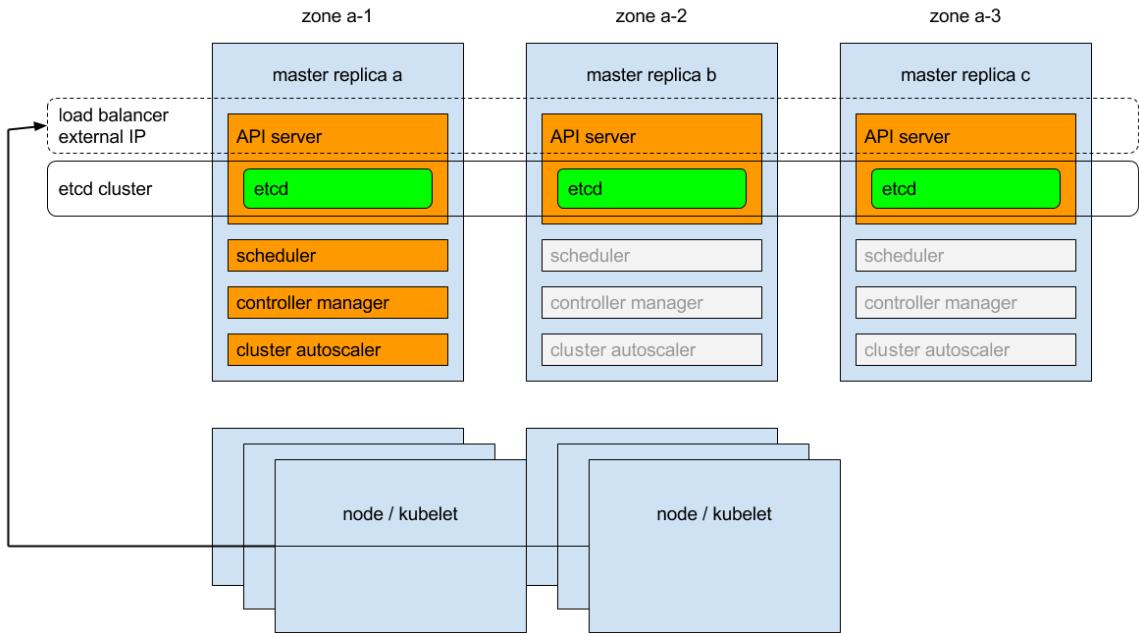
1. 在原有位置增加一个新副本：

```
KUBE_GCE_ZONE=replica-zone KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

高可用集群复制主节点的最佳实践

- 尝试将主节点副本放置在不同的区域。在某区域故障时，放置在该区域内的所有主机都将失败。为了在区域故障中幸免，请同样将工作节点放置在多区域中（详情请见[多区域](#)）。
- 不要使用具有两个主节点副本的集群。在双副本集群上达成一致需要在更改持久状态时两个副本都处于运行状态。因此，两个副本都是需要的，任一副本的失败都会将集群带入多数失败状态。因此，就高可用而言，双副本集群不如单个副本集群。
- 添加主节点副本时，集群状态（etcd）会被复制到一个新实例。如果集群很大，可能需要很长时间才能复制它的状态。这个操作可以通过迁移 etcd 数据存储来加速，详情参见[这里](#)（我们正在考虑在未来添加对迁移 etcd 数据存储的支持）。

实现说明



概述

每个主节点副本将以以下模式运行以下组件：

- etcd 实例：所有实例将会以共识方式组建集群；
- API 服务器：每个服务器将与本地 etcd 通信——集群中的所有 API 服务器都可用；
- 控制器、调度器和集群自动扩缩器：将使用租约机制——每个集群中只有一个实例是可用的；
- 插件管理器：每个管理器将独立工作，试图保持插件同步。

此外，在 API 服务器前面将有一个负载均衡器，用于将外部和内部通信路由到他们。

负载均衡

启动第二个主节点副本时，将创建一个包含两个副本的负载均衡器，并将第一个副本的 IP 地址提升为负载均衡器的 IP 地址。类似地，在删除倒数第二个主节点副本之后，将删除负载均衡器，并将其 IP 地址分配给最后一个剩余的副本。请注意，创建和删除负载均衡器是复杂的操作，可能需要一些时间（约20分钟）来同步。

主节点服务 & kubelets

Kubernetes 并不试图在其服务中保持 apiserver 的列表为最新，相反，它将将所有访问请求指向外部 IP：

- 在拥有一个主节点的集群中，IP 指向单一的主节点，
- 在拥有多个主节点的集群中，IP 指向主节点前面的负载均衡器。

类似地，kubelets 将使用外部 IP 与主节点通信。

主节点证书

Kubernetes 为每个副本的外部公共 IP 和本地 IP 生成主节点 TLS 证书。副本的临时公共 IP 没有证书；要通过其临时公共 IP 访问副本，必须跳过 TLS 检查。

etcd 集群

为了允许 etcd 组建集群，需开放 etcd 实例之间通信所需的端口（用于集群内部通信）。为了使这种部署安全，etcd 实例之间的通信使用 SSL 进行鉴权。

拓展阅读

[自动化高可用集群部署 - 设计文档](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

改变默认 StorageClass

本文展示了如何改变默认的 Storage Class，它用于为没有特殊需求的 PersistentVolumeClaims 配置 volumes。

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.

为什么要改变默认存储类？

取决于安装模式，你的 Kubernetes 集群可能和一个被标记为默认的已有 StorageClass 一起部署。这个默认的 StorageClass 以后将被用于动态的为没有特定存储类需求的 PersistentVolumeClaims 配置存储。更多细节请查看 [PersistentVolumeClaim 文档](#)。

预先安装的默认 StorageClass 可能不能很好的适应你期望的工作负载；例如，它配置的存储可能太过昂贵。如果是这样的话，你可以改变默认 StorageClass，或者完全禁用它以防止动态配置存储。

简单的删除默认 StorageClass 可能行不通，因为它可能会被你集群中的扩展管理器自动重建。请查阅你的安装文档中关于扩展管理器的细节，以及如何禁用单个扩展。

改变默认 StorageClass

1. 列出你的集群中的 StorageClasses：

```
kubectl get storageclass
```

输出类似这样：

NAME	PROVISIONER	AGE
standard (default)	kubernetes.io/gce-pd	1d
gold	kubernetes.io/gce-pd	1d

默认 StorageClass 以 (default) 标记。

1. 标记默认 StorageClass 非默认：

默认 StorageClass 的注解 storageclass.kubernetes.io/is-default-class 设置为 true。注解的其它任意值或者缺省值将被解释为 false。

要标记一个 StorageClass 为非默认的，你需要改变它的值为 false：

```
kubectl patch storageclass standard -p '{"metadata": {"annotations": {"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

这里的 standard 是你选择的 StorageClass 的名字。

1. 标记一个 StorageClass 为默认的：

和前面的步骤类似，你需要添加/设置注解 `storageclass.kubernetes.io/is-default-class=true`。

```
kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

请注意，最多只能有一个 StorageClass 能够被标记为默认。如果它们中有两个或多个被标记为默认，Kubernetes 将忽略这个注解，也就是它将表现为没有默认 StorageClass。

1. 验证你选用的 StorageClass 为默认的：

```
kubectl get storageclass
```

输出类似这样：

NAME	PROVISIONER	AGE
standard	kubernetes.io/gce-pd	1d
gold (default)	kubernetes.io/gce-pd	1d

接下来

- 进一步了解 [PersistentVolumes](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

更改 PersistentVolume 的回收策略

本文展示了如何更改 Kubernetes PersistentVolume 的回收策略。

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.

为什么要更改 PersistentVolume 的回收策略

PersistentVolumes 可以有多种回收策略，包括 "Retain"、"Recycle" 和 "Delete"。对于动态配置的 PersistentVolumes 来说，默认回收策略为 "Delete"。这表示当用户删除对应的 PersistentVolumeClaim 时，动态配置的 volume 将被自动删除。如果 volume 包含重要数据时，这种自动行为可能是不合适的。那种情况下，更适合使用 "Retain" 策略。使用 "Retain" 时，如果用户删除 PersistentVolumeClaim，对应的 PersistentVolume 不会被删除。相反，它将变为 Released 状态，表示所有的数据可以被手动恢复。

更改 PersistentVolume 的回收策略

1. 列出你集群中的 PersistentVolumes

```
kubectl get pv
```

输出类似于这样：

NAME	RECLAIMPOLICY	CAPACITY	ACCESSMODES	STATUS	CLAIM	REASON	AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	Delete	4Gi	RWO	Bound	default/claim1	10s	
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	Delete	4Gi	RWO	Bound	default/claim2	6s	
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	Delete	4Gi	RWO	Bound	default/claim3	3s	

这个列表同样包含了绑定到每个卷的 claims 名称，以便更容易的识别动态配置的卷。

1. 选择你的 PersistentVolumes 中的一个并更改它的回收策略：

```
kubectl patch pv <your-pv-name> -p '{"spec": {"persistentVolumeReclaimPolicy": "Retain"}}'
```

这里的 <your-pv-name> 是你选择的 PersistentVolume 的名字。

说明：

在 Windows 系统上，你必须对包含空格的 JSONPath 模板加双引号（而不是像上面一样为 Bash 环境使用的单引号）。这也意味着你必须使用单引号或者转义的双引号 来处理模板中的字面值。例如：

```
kubectl patch pv <your-pv-name> -p "{\"spec\":{\"persistentVolumeReclaimPolicy\": \"Retain\"}}"
```

1. 验证你选择的 PersistentVolume 拥有正确的策略：

```
kubectl get pv
```

输出类似于这样：

NAME	RECLAIMPOLICY	STATUS	CLAIM	CAPACITY	ACCESSMODES	REASON	AGE
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	Delete	Bound	default/claim1	4Gi	RWO	40s	
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	Delete	Bound	default/claim2	4Gi	RWO	36s	
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	Retain	Bound	default/claim3	4Gi	RWO	33s	

在前面的输出中，你可以看到绑定到申领 default/claim3 的卷的回收策略为 Retain。当用户删除申领 default/claim3 时，它不会被自动删除。

接下来

- 进一步了解 [PersistentVolumes](#)
- 进一步了解 [PersistentVolumeClaims](#)

参考

- [PersistentVolume](#)
- [PersistentVolumeClaim](#)
- 参阅 [PersistentVolumeSpec](#) 的 persistentVolumeReclaimPolicy 字段

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

自动扩缩集群 DNS 服务

本页展示了如何在集群中启用和配置 DNS 服务的自动扩缩功能。

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.

- 本指南假设你的节点使用 AMD64 或 Intel 64 CPU 架构
- 确保已启用 [DNS 功能](#)本身。
- 建议使用 Kubernetes 1.4.0 或更高版本。

确定是否 DNS 水平 水平自动扩缩特性已经启用

在 kube-system 命名空间中列出集群中的 [Deployments](#) :

```
kubectl get deployment --namespace=kube-system
```

输出类似如下这样：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
...					
dns-autoscaler	1	1	1	1	...
...					

如果在输出中看到 "dns-autoscaler"，说明 DNS 水平自动扩缩已经启用，可以跳到 [调优自动扩缩参数](#)。

获取 DNS Deployment 的名称

列出集群内 kube-system 名字空间中的 DNS Deployment :

```
kubectl get deployment -l k8s-app=kube-dns --namespace=kube-system
```

输出类似如下这样：

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
...				
coredns	2/2	2	2	...
...				

如果看不到 DNS 服务的 Deployment，你也可以通过名字来查找：

```
kubectl get deployment --namespace=kube-system
```

并在输出中寻找名称为 coredns 或 kube-dns 的 Deployment。

你的扩缩目标为：

Deployment/<your-deployment-name>

其中 <your-deployment-name> 是 DNS Deployment 的名称。例如，如果你的 DNS Deployment 名称是 coredns，则你的扩展目标是 Deployment/coredns。

说明：CoreDNS 是 Kubernetes 的默认 DNS 服务。CoreDNS 设置标签 k8s-app=kube-dns，以便能够在原来使用 kube-dns 的集群中工作。

启用 DNS 水平自动扩缩

在本节，我们创建一个 Deployment。Deployment 中的 Pod 运行一个基于 cluster-proportional-autoscaler-amd64 镜像的容器。

创建文件 dns-horizontal-autoscaler.yaml，内容如下所示：

[admin/dns/dns-horizontal-autoscaler.yaml](#)


```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: dns-autoscaler
spec:
  selector:
    matchLabels:
      k8s-app: dns-autoscaler
  template:
    metadata:
      labels:
        k8s-app: dns-autoscaler
    spec:
      containers:
        - name: autoscaler
          image: k8s.gcr.io/cluster-proportional-autoscaler-amd64:1.6.0
          resources:
            requests:
              cpu: 20m
              memory: 10Mi
          command:
            - /cluster-proportional-autoscaler
```

```
- --namespace=kube-system
- --configmap=dns-autoscaler
- --target=<SCALE_TARGET>
# When cluster is using large nodes(with more cores), "coresPerReplica" should dominate.
# If using small nodes, "nodesPerReplica" should dominate.
- --default-params={"linear":{"coresPerReplica":256,"nodesPerReplica":16,"min":1}}
- --logtostderr=true
- --v=2
```

在文件中，将 <SCALE_TARGET> 替换成扩缩目标。

进入到包含配置文件的目录中，输入如下命令创建 Deployment：

```
kubectl apply -f dns-horizontal-autoscaler.yaml
```

一个成功的命令输出是：

```
deployment.apps/dns-autoscaler created
```

DNS 水平自动扩缩已经在启用了。

调优自动扩缩参数

验证 dns-autoscaler [ConfigMap](#) 是否存在：

```
kubectl get configmap --namespace=kube-system
```

输出类似于：

NAME	DATA	AGE
...
dns-autoscaler	1	...
...

修改该 ConfigMap 中的数据：

```
kubectl edit configmap dns-autoscaler --namespace=kube-system
```

找到如下这行内容：

```
linear: '{"coresPerReplica":256,"min":1,"nodesPerReplica":16}'
```

根据需要修改对应的字段。"min" 字段表明 DNS 后端的最小数量。实际后端的数量通过使用如下公式来计算：

```
replicas = max( ceil( cores * 1/coresPerReplica ), ceil( nodes * 1/nodesPerReplica ) )
```

注意 coresPerReplica 和 nodesPerReplica 的值都是整数。

背后的思想是，当一个集群使用具有很多核心的节点时，由 coresPerReplica 来控制。
当一个集群使用具有较少核心的节点时，由 nodesPerReplica 来控制。

其它的扩缩模式也是支持的，详情查看 [cluster-proportional-autoscaler](#)。

禁用 DNS 水平自动扩缩

有几个可供调优的 DNS 水平自动扩缩选项。具体使用哪个选项因环境而异。

选项 1：缩容 dns-autoscaler Deployment 至 0 个副本

该选项适用于所有场景。运行如下命令：

```
kubectl scale deployment --replicas=0 dns-autoscaler --namespace=kube-system
```

输出如下所示：

```
deployment.apps/dns-autoscaler scaled
```

验证当前副本数为 0：

```
kubectl get rs --namespace=kube-system
```

输出内容中，在 DESIRED 和 CURRENT 列显示为 0：

NAME	DESIRED	CURRENT	READY	AGE
...				
dns-autoscaler-6b59789fc8	0	0	0	...
...				

选项 2：删除 dns-autoscaler Deployment

如果 dns-autoscaler 为你所控制，也就说没有人会去重新创建它，可以选择此选项：

```
kubectl delete deployment dns-autoscaler --namespace=kube-system
```

输出内容如下所示：

```
deployment.apps "dns-autoscaler" deleted
```

选项 3：从主控节点删除 dns-autoscaler 清单文件

如果 dns-autoscaler 在[插件管理器](#)的控制之下，并且具有操作 master 节点的写权限，可以使用此选项。

登录到主控节点，删除对应的清单文件。 dns-autoscaler 对应的路径一般为：

```
/etc/kubernetes/addons/dns-horizontal-autoscaler/dns-horizontal-autoscaler.yaml
```

当清单文件被删除后，插件管理器将删除 dns-autoscaler Deployment。

理解 DNS 水平自动扩缩工作原理

- cluster-proportional-autoscaler 应用独立于 DNS 服务部署。
- autoscaler Pod 运行一个客户端，它通过轮询 Kubernetes API 服务器获取集群中节点和核心的数量。
- 系统会基于当前可调度的节点个数、核心数以及所给的扩缩参数，计算期望的副本数并应用到 DNS 后端。
- 扩缩参数和数据点会基于一个 ConfigMap 来提供给 autoscaler，它会在每次轮询时刷新它的参数表，以与最近期望的扩缩参数保持一致。
- 扩缩参数是可以被修改的，而且不需要重建或重启 autoscaler Pod。
- autoscaler 提供了一个控制器接口来支持两种控制模式：*linear* 和 *ladder*。

接下来

- 阅读[为关键插件 Pod 提供的调度保障](#)
- 进一步了解[cluster-proportional-autoscaler 实现](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

自定义 DNS 服务

本页说明如何配置 DNS [Pod\(s\)](#)，以及定制集群中 DNS 解析过程。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.12. 要获知版本信息，请输入 kubectl version.

你的集群必须运行 CoreDNS 插件。 文档[迁移到 CoreDNS](#) 解释了如何使用 kubeadm 从 kube-dns 迁移到 CoreDNS。

您的 Kubernetes 服务器版本必须不低于版本 v1.12. 要获知版本信息，请输入 kubectl version.

介绍

DNS 是使用[集群插件](#) 管理器自动启动的内置的 Kubernetes 服务。

从 Kubernetes v1.12 开始，CoreDNS 是推荐的 DNS 服务器，取代了 kube-dns。如果 你的集群原来使用 kube-dns，你可能部署的仍然是 kube-dns 而不是 CoreDNS。

说明：CoreDNS 和 kube-dns 的 Service 都在其 metadata.name 字段使用名字 kube-dns。这是为了能够与依靠传统 kube-dns 服务名称来解析集群内部地址的工作负载具有更好的互操作性。使用 kube-dns 作为服务名称可以抽离共有名称之后运行的是哪个 DNS 提供程序这一实现细节。

如果你在使用 Deployment 运行 CoreDNS，则该 Deployment 通常会向外暴露为一个具有 静态 IP 地址 Kubernetes 服务。kubelet 使用 --cluster-dns=<DNS 服务 IP> 标志将 DNS 解析器的信息传递给每个容器。

DNS 名称也需要域名。你可在 kubelet 中使用 --cluster-domain=<默认本地域名> 标志配置本地域名。

DNS 服务器支持正向查找（A 和 AAAA 记录）、端口发现（SRV 记录）、反向 IP 地址发现（PTR 记录）等。更多信息，请参见[Pod 和服务的 DNS](#)。

如果 Pod 的 dnsPolicy 设置为 "default"，则它将从 Pod 运行所在节点继承名称解析配置。Pod 的 DNS 解析行为应该与节点相同。但请参阅[已知问题](#)。

如果你不想这样做，或者想要为 Pod 使用其他 DNS 配置，则可以 使用 kubelet 的 --resolv-conf 标志。将此标志设置为 "" 可以避免 Pod 继承 DNS。将其设置为有别于 /etc/resolv.conf 的有效文件路径可以设定 DNS 继承不同的配置。

CoreDNS

CoreDNS 是通用的权威 DNS 服务器，可以用作集群 DNS，符合 [DNS 规范](#)。

CoreDNS ConfigMap 选项

CoreDNS 是模块化且可插拔的 DNS 服务器，每个插件都为 CoreDNS 添加了新功能。可以通过维护 [Corefile](#)，即 CoreDNS 配置文件，来定制其行为。集群管理员可以修改 CoreDNS Corefile 的 ConfigMap，以更改服务发现的工作方式。

在 Kubernetes 中，CoreDNS 安装时使用如下默认 Corefile 配置。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
      errors
      health {
        lameduck 5s
      }
      ready
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
        ttl 30
      }
      prometheus :9153
      forward ./etc/resolv.conf
      cache 30
      loop
      reload
      loadbalance
    }
```

Corefile 配置包括以下 CoreDNS 插件：

- [errors](#)：错误记录到标准输出。
- [health](#)：在 `http://localhost:8080/health` 处提供 CoreDNS 的健康报告。
- [ready](#)：在端口 8181 上提供的一个 HTTP 末端，当所有能够表达自身就绪的插件都已就绪时，在此末端返回 200 OK。
- [kubernetes](#)：CoreDNS 将基于 Kubernetes 的服务和 Pod 的 IP 答复 DNS 查询。你可以在 CoreDNS 网站阅读[更多细节](#)。你可以使用 `ttl` 来定制响应的 TTL。默认值是 5 秒钟。TTL 的最小值可以是 0 秒钟，最大值为 3600 秒。将 TTL 设置为 0 可以禁止对 DNS 记录进行缓存。

`pods insecure` 选项是为了与 `kube-dns` 向后兼容。你可以使用 `pods verified` 选项，该选项使得仅在相同名称空间中存在具有匹配 IP 的 Pod 时才返回 A 记录。如果你不使用 Pod 记录，则可以使用 `pods disabled` 选项。

- [prometheus](#)：CoreDNS 的度量指标值以 [Prometheus](#) 格式在 `http://localhost:9153/metrics` 上提供。
- [forward](#)：不在 Kubernetes 集群域内的任何查询都将转发到 预定义的解析器 (`/etc/resolv.conf`)。
- [cache](#)：启用前端缓存。
- [loop](#)：检测到简单的转发环，如果发现死循环，则中止 CoreDNS 进程。
- [reload](#)：允许自动重新加载已更改的 Corefile。编辑 ConfigMap 配置后，请等待两分钟，以使更改生效。
- [loadbalance](#)：这是一个轮转式 DNS 负载均衡器，它在应答中随机分配 A、AAAA 和 MX 记录的顺序。

你可以通过修改 ConfigMap 来更改默认的 CoreDNS 行为。

使用 CoreDNS 配置存根域和上游域名服务器

CoreDNS 能够使用 [forward 插件](#)配置存根域和上游域名服务器。

示例

如果集群操作员在 10.150.0.1 处运行了 [Consul](#) 域服务器，且所有 Consul 名称都带有后缀 `.consul.local`。要在 CoreDNS 中对其进行配置，集群管理员可以在 CoreDNS 的 ConfigMap 中创建加入以下字段。

```
consul.local:53 {  
    errors  
    cache 30  
    forward . 10.150.0.1  
}
```

要显式强制所有非集群 DNS 查找通过特定的域名服务器（位于 172.16.0.1），可将 `forward` 指向该域名服务器，而不是 `/etc/resolv.conf`。

```
forward . 172.16.0.1
```

最终的包含默认的 Corefile 配置的 ConfigMap 如下所示：

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: coredns  
  namespace: kube-system  
data:  
  Corefile: |  
    .:53 {  
      errors
```

```
health
kubernetes cluster.local in-addr.arpa ip6.arpa {
    pods insecure
    fallthrough in-addr.arpa ip6.arpa
}
prometheus :9153
forward . 172.16.0.1
cache 30
loop
reload
loadbalance
}
consul.local:53 {
    errors
    cache 30
    forward . 10.150.0.1
}
```

工具 kubeadm 支持将 kube-dns ConfigMap 自动转换为 CoreDNS ConfigMap。

说明：尽管 kube-dns 接受 FQDN (例如：ns.foo.com) 作为存根域和名字服务器，CoreDNS 不支持此功能。转换期间，CoreDNS 配置中将忽略所有的 FQDN 域名服务器。

CoreDNS 配置等同于 kube-dns

CoreDNS 不仅仅提供 kube-dns 的功能。为 kube-dns 创建的 ConfigMap 支持 StubDomains 和 upstreamNameservers 转换为 CoreDNS 中的 forward 插件。同样，kube-dns 中的 Federations 插件会转换为 CoreDNS 中的 federation 插件。

示例

用于 kubedns 的此示例 ConfigMap 描述了 federations、stubdomains and upstreamnameservers：

```
apiVersion: v1
data:
  federations: |
    {"foo" : "foo.feddomain.com"}
  stubDomains: |
    {"abc.com" : ["1.2.3.4"], "my.cluster.local" : ["2.3.4.5"]}
  upstreamNameservers: |
    ["8.8.8.8", "8.8.4.4"]
kind: ConfigMap
```

CoreDNS 中的等效配置将创建一个 Corefile :

- 针对 federations:

```
federation cluster.local {  
    foo foo.feddomain.com  
}
```

- 针对 stubDomains:

```
abc.com:53 {  
    errors  
    cache 30  
    proxy . 1.2.3.4  
}  
my.cluster.local:53 {  
    errors  
    cache 30  
    proxy . 2.3.4.5  
}
```

带有默认插件的完整 Corefile :

```
.:53 {  
    errors  
    health  
    kubernetes cluster.local in-addr.arpa ip6.arpa {  
        pods insecure  
        fallthrough in-addr.arpa ip6.arpa  
    }  
    federation cluster.local {  
        foo foo.feddomain.com  
    }  
    prometheus :9153  
    forward . 8.8.8.8 8.8.4.4  
    cache 30  
}  
abc.com:53 {  
    errors  
    cache 30  
    forward . 1.2.3.4  
}  
my.cluster.local:53 {  
    errors  
    cache 30  
    forward . 2.3.4.5  
}
```

迁移到 CoreDNS

要从 kube-dns 迁移到 CoreDNS，[此博客](#) 提供了帮助用户将 kube-dns 替换为 CoreDNS。 集群管理员还可以使用[部署脚本](#) 进行迁移。

接下来

- 阅读[调试 DNS 解析](#)

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 22, 2020 at 7:32 PM PST: [fix-24683 \(6ed67daf6\)](#)

访问集群上运行的服务

本文展示了如何连接 Kubernetes 集群上运行的服务。

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.

访问集群上运行的服务

在 Kubernetes 里，[节点](#)、[Pod](#) 和 [服务](#) 都有自己的 IP。许多情况下，集群上的节点 IP、Pod IP 和某些服务 IP 是路由不可达的，所以不能从集群之外访问它们，例如从你自己的台式机。

连接方式

你有多种可选方式从集群外连接节点、Pod 和服务：

- 通过公网 IP 访问服务
 - 使用类型为 NodePort 或 LoadBalancer 的服务，可以从外部访问它们。请查阅[服务](#)和[kubectl expose](#) 文档。
 - 取决于你的集群环境，你可以仅把服务暴露在你的企业网络环境中，也可以将其暴露在因特网上。需要考虑暴露的服务是否安全，它是否有自己的用户认证？
 - 将 Pod 放置于服务背后。如果要访问一个副本集合中特定的 Pod，例如用于调试目的，请给 Pod 指定一个独特的标签并创建一个新服务选择该标签。
 - 大部分情况下，都不需要应用开发者通过节点 IP 直接访问节点。
- 通过 Proxy 动词访问服务、节点或者 Pod
 - 在访问远程服务之前，利用 API 服务器执行身份认证和鉴权。如果你的服务不够安全，无法暴露到因特网中，或者需要访问节点 IP 上的端口，又或者出于调试目的，可使用这种方式。
 - 代理可能给某些应用带来麻烦
 - 此方式仅适用于 HTTP/HTTPS
 - 进一步的描述在[这里](#)
 - 从集群中的 node 或者 pod 访问。
- 从集群中的一个节点或 Pod 访问
 - 运行一个 Pod，然后使用[kubectl exec](#) 连接到它的 Shell。从那个 Shell 连接其他的节点、Pod 和服务
 - 某些集群可能允许你 SSH 到集群中的节点。你可能可以从那儿访问集群服务。这是一个非标准的方式，可能在一些集群上能工作，但在另一些上却不能。浏览器和其他工具可能已经安装也可能没有安装。集群 DNS 可能不会正常工作。

发现内置服务

典型情况下，kube-system 名字空间中会启动集群的几个服务。使用 `kubectl cluster-info` 命令获取这些服务的列表：

```
kubectl cluster-info
```

输出类似于：

```
Kubernetes master is running at https://104.197.5.247
elasticsearch-logging is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-
system/services/kibana-logging/proxy
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/
services/kube-dns/proxy
grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/
services/monitoring-grafana/proxy
```

```
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/monitoring-heapster/proxy
```

这一输出显示了用 proxy 动词访问每个服务时可用的 URL。例如，此集群（使用 Elasticsearch）启用了集群层面的日志。如果提供合适的凭据，可以通过 https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/ 访问，或通过一个 kubectl proxy 来访问： http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/。

说明：请参阅[使用Kubernetes API访问集群](#)了解如何传递凭据或如何使用 kubectl proxy。

手动构建 API 服务器代理 URLs

如前所述，你可以使用 kubectl cluster-info 命令取得服务的代理 URL。为了创建包含服务末端、后缀和参数的代理 URLs，你可以简单地在服务的代理 URL 中添加：`http://kubernetes_master_address/api/v1/namespaces/namespace_name/services/service_name[:port_name]/proxy`

如果还没有为你的端口指定名称，你可以不用在 URL 中指定 `port_name`。

示例

- 如要访问 Elasticsearch 服务末端 `_search?q=user:kimchy`，你可以使用：

```
http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy
```

- 如要访问 Elasticsearch 集群健康信息 `_cluster/health?pretty=true`，你会使用：

```
https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_cluster/health?pretty=true`
```

健康信息与下面的例子类似：

```
{
  "cluster_name": "kubernetes_logging",
  "status": "yellow",
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 5,
  "active_shards": 5,
  "relocating_shards": 0,
  "initializing_shards": 0,
```

```
        "unassigned_shards" : 5  
    }
```

- 要访问 `https` Elasticsearch 服务健康信息 `_cluster/health?pretty=true`，你会使用：

```
https://104.197.5.247/api/v1/namespaces/kube-system/services/  
https:elasticsearch-logging/proxy/_cluster/health?pretty=true
```

通过 Web 浏览器访问集群中运行的服务

你或许能够将 API 服务器代理的 URL 放入浏览器的地址栏，然而：

- Web 服务器通常不能传递令牌，所以你可能需要使用基本（密码）认证。 API 服务器可以配置为接受基本认证，但你的集群可能并没有这样配置。
- 某些 Web 应用可能无法工作，特别是那些使用客户端 Javascript 构造 URL 的应用，所构造的 URL 可能并不支持代理路径前缀。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

调试 DNS 问题

这篇文章提供了一些关于 DNS 问题诊断的方法。

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](#)

你的集群必须使用了 CoreDNS 插件 或者其前身，`kube-dns`。

您的 Kubernetes 服务器版本必须不低于版本 v1.6. 要获知版本信息，请输入 `kubectl version`.

创建一个简单的 Pod 作为测试环境

[admin/dns/dnsutils.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dnsutils
  namespace: default
spec:
  containers:
  - name: dnsutils
    image: gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
    command:
      - sleep
      - "3600"
  imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

使用上面的清单来创建一个 Pod :

```
kubectl apply -f https://k8s.io/examples/admin/dns/dnsutils.yaml
```

```
pod/dnsutils created
```

验证其状态 :

```
kubectl get pods dnsutils
```

NAME	READY	STATUS	RESTARTS	AGE
dnsutils	1/1	Running	0	<some-time>

一旦 Pod 处于运行状态 , 你就可以在该环境里执行 nslookup 。如果你看到类似下列的内容 , 则表示 DNS 是正常运行的。

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10
```

```
Name: kubernetes.default
Address 1: 10.0.0.1
```

如果 nslookup 命令执行失败 , 请检查下列内容 :

先检查本地的 DNS 配置

查看 resolv.conf 文件的内容（阅读[从节点继承 DNS 配置](#)和后文的[已知问题](#)，获取更多信息）

```
kubectl exec -ti dnsutils -- cat /etc/resolv.conf
```

验证 search 和 nameserver 的配置是否与下面的内容类似（注意 search 根据不同的云提供商可能会有所不同）：

```
search default.svc.cluster.local svc.cluster.local cluster.local google.internal
c.gce_project_id.internal
nameserver 10.0.0.10
options ndots:5
```

下列错误表示 CoreDNS（或 kube-dns）插件或者相关服务出现了问题：

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

输出为：

```
Server: 10.0.0.10
Address 1: 10.0.0.10

nslookup: can't resolve 'kubernetes.default'
```

或者

```
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

nslookup: can't resolve 'kubernetes.default'
```

检查 DNS Pod 是否运行

使用 kubectl get pods 命令来验证 DNS Pod 是否运行。

```
kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
```

NAME	READY	STATUS	RESTARTS	AGE
... coredns-7b96bf9f76-5hsxb coredns-7b96bf9f76-mvmmt	1/1 1/1	Running Running	0 0	1h 1h
...				

说明：对于 CoreDNS 和 kube-dns 部署而言，标签 k8s-app 的值都应该是 kube-dns。

如果你发现没有 CoreDNS Pod 在运行，或者该 Pod 的状态是 failed 或者 completed，那可能这个 DNS 插件在您当前的环境里并没有成功部署，你将需要手动去部署它。

检查 DNS Pod 里的错误

使用 kubectl logs 命令来查看 DNS 容器的日志信息。

```
kubectl logs --namespace=kube-system -l k8s-app=kube-dns
```

下列是一个正常运行的 CoreDNS 日志信息：

```
.:53
2018/08/15 14:37:17 [INFO] CoreDNS-1.2.2
2018/08/15 14:37:17 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.2
linux/amd64, go1.10.3, 2e322f6
2018/08/15 14:37:17 [INFO] plugin/reload: Running configuration MD5 =
24e6c59e83ce706f07bcc82c31b1ea1c
```

查看是否日志中有一些可疑的或者意外的消息。

检查是否启用了 DNS 服务

使用 kubectl get service 命令来检查 DNS 服务是否已经启用。

```
kubectl get svc --namespace=kube-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
...					
kube-dns	ClusterIP	10.0.0.10	<none>	53/UDP,53/TCP	1h
...					

说明：不管是 CoreDNS 还是 kube-dns，这个服务的名字都会是 kube-dns。

如果你已经创建了 DNS 服务，或者该服务应该是默认自动创建的但是它并没有出现，请阅读[调试服务](#) 来获取更多信息。

DNS 的端点公开了吗？

你可以使用 kubectl get endpoints 命令来验证 DNS 的端点是否公开了。

```
kubectl get ep kube-dns --namespace=kube-system
```

NAME	ENDPOINTS	AGE
kube-dns	10.180.3.17:53,10.180.3.17:53	1h

如果你没看到对应的端点，请阅读[调试服务的端点部分](#)。

若需要了解更多的 Kubernetes DNS 例子，请在 Kubernetes GitHub 仓库里查看 [cluster-dns 示例](#)。

DNS 查询有被接收或者执行吗？

你可以通过给 CoreDNS 的配置文件（也叫 Corefile）添加 log 插件来检查查询是否被正确接收。CoreDNS 的 Corefile 被保存在一个叫 coredns 的 ConfigMap 里，使用下列命令来编辑它：

```
kubectl -n kube-system edit configmap coredns
```

然后按下面的例子给 Corefile 添加 log。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
      log
      errors
      health
      kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
      }
      prometheus :9153
      forward . /etc/resolv.conf
      cache 30
      loop
      reload
      loadbalance
    }
```

保存这些更改后，你可能会需要等待一到两分钟让 Kubernetes 把这些更改应用到 CoreDNS 的 Pod 里。

接下来，发起一些查询并依照前文所述查看日志信息，如果 CoreDNS 的 Pod 接收到这些查询，你将可以在日志信息里看到它们。

下面是日志信息里的查询例子：

```
.:53
2018/08/15 14:37:15 [INFO] CoreDNS-1.2.0
2018/08/15 14:37:15 [INFO] linux/amd64, go1.10.3, 2e322f6
```

```
CoreDNS-1.2.0
linux/amd64, go1.10.3, 2e322f6
2018/09/07 15:29:04 [INFO] plugin/reload: Running configuration MD5 =
162475cdf272d8aa601e6fe67a6ad42f
2018/09/07 15:29:04 [INFO] Reloading complete
172.17.0.18:41675 - [07/Sep/2018:15:29:11 +0000] 59925 "A IN
kubernetes.default.svc.cluster.local. udp 54 false 512" NOERROR qr,aa,rd,ra 106
0.000066649s
```

已知问题

有些 Linux 发行版本（比如 Ubuntu）默认使用一个本地的 DNS 解析器（systemd-resolved）。systemd-resolved 会用一个存根文件（Stub File）来覆盖 /etc/resolv.conf 内容，从而可能在上游服务器中解析域名产生转发环（forwarding loop）。这个问题可以通过手动指定 kubelet 的 --resolv-conf 标志为正确的 resolv.conf（如果是 systemd-resolved，则这个文件路径为 /run/systemd/resolve/resolv.conf）来解决。kubeadm 会自动检测 systemd-resolved 并对应的更改 kubelet 的命令行标志。

Kubernetes 的安装并不会默认配置节点的 resolv.conf 文件来使用集群的 DNS 服务，因为这个配置对于不同的发行版本是不一样的。这个问题应该迟早会被解决的。

Linux 的 libc 限制 nameserver 只能有三个记录。不仅如此，对于 glibc-2.17-222 之前的版本（[参见此 Issue 了解新版本的更新](#)），search 的记录不能超过 6 个（[详情请查阅这个 2005 年的 bug](#)）。Kubernetes 需要占用一个 nameserver 记录和三个 search 记录。这意味着如果一个本地的安装已经使用了三个 nameserver 或者使用了超过三个 search 记录，而你的 glibc 版本也在有问题的版本列表中，那么有些配置很可能会丢失。为了绕过 DNS nameserver 个数限制，节点可以运行 dnsmasq，以提供更多的 nameserver 记录。你也可以使用 kubelet 的 --resolv-conf 标志来解决这个问题。要想修复 DNS search 记录个数限制问题，可以考虑升级你的 Linux 发行版本，或者升级 glibc 到一个不再受此困扰的版本。

如果你使用 Alpine 3.3 或更早版本作为你的基础镜像，DNS 可能会由于 Alpine 中一个已知的问题导致无法正常工作。请查看[这里](#)获取更多信息。

接下来

- 参阅[自动扩缩集群中的 DNS 服务](#).
- 阅读[服务和 Pod 的 DNS](#)

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 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

通过名字空间共享集群

本页展示如何查看、使用和删除[名字空间](#)。 本页同时展示如何使用 Kubernetes 名字空间来划分集群。

准备开始

- 你已拥有一个[配置好的 Kubernetes 集群](#).
- 你已对 Kubernetes 的 [Pods](#), [Services](#), 和 [Deployments](#) 有基本理解。

查看名字空间

1. 列出集群中现有的名字空间：

```
kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	11d
kube-system	Active	11d
kube-public	Active	11d

初始状态下，Kubernetes 具有三个名字空间：

- default 无名字空间对象的默认名字空间
- kube-system 由 Kubernetes 系统创建的对象的名字空间
- kube-public 自动创建且被所有用户可读的名字空间（包括未经身份认证的）。此名字空间通常在某些资源在整个集群中可见且可公开读取时被集群使用。此名字空间的公共方面只是一个约定，而不是一个必要条件。

你还可以通过下列命令获取特定名字空间的摘要：

```
kubectl get namespaces <name>
```

或用下面的命令获取详细信息：

```
kubectl describe namespaces <name>
```

Name:	default
Labels:	<none>
Annotations:	<none>
Status:	Active

```
No resource quota.
```

Resource Limits

Type	Resource	Min	Max	Default
Container	cpu	-	-	100m

请注意，这些详情同时显示了资源配额（如果存在）以及资源限制区间。

资源配置跟踪并聚合 *Namespace* 中资源的使用情况，并允许集群运营者定义 *Namespace* 可能消耗的 *Hard* 资源使用限制。

限制区间定义了单个实体在一个 *Namespace* 中可使用的最小/最大资源量约束。

参阅 [准入控制: 限制区间](#)

名字空间可以处于下列两个阶段中的一个：

- Active 名字空间正在被使用中
- Terminating 名字空间正在被删除，且不能被用于新对象。

参见[设计文档](#) 查看更多细节。

创建名字空间

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

1. 新建一个名为 `my-namespace.yaml` 的 YAML 文件，并写入下列内容：

```
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

然后运行：

```
kubectl create -f ./my-namespace.yaml
```

1. 或者，你可以使用下面的命令创建名字空间：

```
kubectl create namespace <insert-namespace-name-here>
```

请注意，名字空间的名称必须是一个合法的 [DNS 标签](#)。

可选字段 `finalizers` 允许观察者们在名字空间被删除时清除资源。记住如果指定了一个不存在的终结器，名字空间仍会被创建，但如果用户试图删除它，它将陷入 `Terminating` 状态。

更多有关 `finalizers` 的信息请查阅 [设计文档](#) 中名字空间部分。

删除名字空间

删除名字空间使用命令：

```
kubectl delete namespaces <insert-some-namespace-name>
```

警告： 这会删除名字空间下的 **所有内容**！

删除是异步的，所以有一段时间你会看到名字空间处于 Terminating 状态。

使用 Kubernetes 名字空间细分你的集群

1. 理解 default 名字空间

默认情况下，Kubernetes 集群会在配置集群时实例化一个 default 名字空间，用以存放集群所使用的默认 Pods、Services 和 Deployments 集合。

假设你有一个新的集群，你可以通过执行以下操作来内省可用的名字空间

```
kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	13m

1. 创建新的名字空间

在本练习中，我们将创建两个额外的 Kubernetes 名字空间来保存我们的内容。

在某组织使用共享的 Kubernetes 集群进行开发和生产的场景中：

开发团队希望在集群中维护一个空间，以便他们可以查看用于构建和运行其应用程序的 Pods、Services 和 Deployments 列表。在这个空间里，Kubernetes 资源被自由地加入或移除，对谁能够或不能修改资源的限制被放宽，以实现敏捷开发。

运维团队希望在集群中维护一个空间，以便他们可以强制实施一些严格的规程，对谁可以或不可以操作运行生产站点的 Pods、Services 和 Deployments 集合进行控制。

该组织可以遵循的一种模式是将 Kubernetes 集群划分为两个名字空间：development 和 production。

让我们创建两个新的名字空间来保存我们的工作。

使用 kubectl 创建 development 名字空间。

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

让我们使用 kubectl 创建 production 名字空间。

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.json
```

为了确保一切正常，列出集群中的所有名字空间。

```
kubectl get namespaces --show-labels
```

NAME	STATUS	AGE	LABELS
default	Active	32m	<none>
development	Active	29s	name=development
production	Active	23s	name=production

1. 在每个名字空间中创建 pod

Kubernetes 名字空间为集群中的 Pods、Services 和 Deployments 提供了作用域。

与一个名字空间交互的用户不会看到另一个名字空间中的内容。

为了演示这一点，让我们在 development 名字空间中启动一个简单的 Deployment 和 Pod。

```
kubectl create deployment snowflake --image=k8s.gcr.io/serve_hostname -n=development  
kubectl scale deployment snowflake --replicas=2 -n=development
```

我们刚刚创建了一个副本个数为 2 的 Deployment，运行名为 snowflake 的 Pod，其中包含一个仅负责提供主机名的基本容器。

```
kubectl get deployment -n=development
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
snowflake	2/2	2	2	2m

```
kubectl get pods -l app=snowflake -n=development
```

NAME	READY	STATUS	RESTARTS	AGE
snowflake-3968820950-9dgr8	1/1	Running	0	2m
snowflake-3968820950-vgc4n	1/1	Running	0	2m

看起来还不错，开发人员能够做他们想做的事，而且他们不必担心会影响到 production 名字空间下面的内容。

让我们切换到 production 名字空间，展示一下一个名字空间中的资源是如何对另一个名字空间隐藏的。

名字空间 production 应该是空的，下面的命令应该不会返回任何东西。

```
kubectl get deployment -n=production  
kubectl get pods -n=production
```

生产环境下一般以养牛的方式运行负载，所以让我们创建一些 Cattle (牛) Pod。

```
kubectl create deployment cattle --image=k8s.gcr.io/serve_hostname -n=production
```

```
kubectl scale deployment cattle --replicas=5 -n=production
```

```
kubectl get deployment -n=production
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
cattle	5/5	5	5	10s

```
kubectl get pods -l app=cattle -n=production
```

NAME	READY	STATUS	RESTARTS	AGE
cattle-2263376956-41xy6	1/1	Running	0	34s
cattle-2263376956-kw466	1/1	Running	0	34s
cattle-2263376956-n4v97	1/1	Running	0	34s
cattle-2263376956-p5p3i	1/1	Running	0	34s
cattle-2263376956-sxpth	1/1	Running	0	34s

此时，应该很清楚的展示了用户在一个名字空间中创建的资源对另一个名字空间是隐藏的。

随着 Kubernetes 中的策略支持的发展，我们将扩展此场景，以展示如何为每个名字空间提供不同的授权规则。

理解使用名字空间的动机

单个集群应该能满足多个用户及用户组的需求（以下称为“用户社区”）。

Kubernetes 名字空间帮助不同的项目、团队或客户去共享 Kubernetes 集群。

名字空间通过以下方式实现这点：

1. 为**名字**设置作用域。
2. 为集群中的部分资源关联鉴权和策略的机制。

使用多个名字空间是可选的。

每个用户社区都希望能够与其他社区隔离开展工作。

每个用户社区都有自己的：

1. 资源（pods、服务、副本控制器等等）
2. 策略（谁能或不能在他们的社区里执行操作）
3. 约束（该社区允许多少配额，等等）

集群运营者可以为每个唯一用户社区创建名字空间。

名字空间为下列内容提供唯一的作用域：

1. 命名资源（避免基本的命名冲突）

2. 将管理权限委派给可信用户
3. 限制社区资源消耗的能力

用例包括：

1. 作为集群运营者，我希望能在单个集群上支持多个用户社区。
2. 作为集群运营者，我希望将集群分区的权限委派给这些社区中的受信任用户。
3. 作为集群运营者，我希望能限定每个用户社区可使用的资源量，以限制对使用同一集群的其他用户社区的影响。
4. 作为群集用户，我希望与我的用户社区相关的资源进行交互，而与其他用户社区在该集群上执行的操作无关。

理解名字空间和 DNS

当你创建[服务](#)时，Kubernetes 会创建相应的 [DNS 条目](#)。此条目的格式为 <服务名称>.<名字空间名称>.svc.cluster.local。这意味着如果容器只使用 <服务名称>，它将解析为名字空间本地的服务。这对于在多个名字空间（如开发、暂存和生产）中使用相同的配置非常有用。如果要跨名字空间访问，则需要使用完全限定的域名（FQDN）。

接下来

- 进一步了解[设置名字空间偏好](#)
- 进一步了解[设置请求的名字空间](#)
- 参阅[名字空间的设计文档](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 15, 2020 at 12:13 AM PST: [\[zh\] Resync namespace task \(82b25694e\)](#)

通过配置文件设置 Kubelet 参数

FEATURE STATE: Kubernetes v1.10 [beta]

通过保存在硬盘的配置文件设置 kubelet 的部分配置参数，这可以作为命令行参数的替代。此功能在 v1.10 中为 beta 版。

建议通过配置文件的方式提供参数，因为这样可以简化节点部署和配置管理。

准备开始

- 需要安装 1.10 或更高版本的 kubelet 可执行文件，才能使用此 beta 功能。

创建配置文件

KubeletConfiguration 结构体定义了可以通过文件配置的 Kubelet 配置子集，该结构体在 [这里 \(v1beta1\)](#) 可以找到。

配置文件必须是这个结构体中参数的 JSON 或 YAML 表现形式。确保 kubelet 可以读取该文件。

下面是一个 Kubelet 配置文件示例：

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
evictionHard:
  memory.available: "200Mi"
```

在这个示例中，当可用内存低于 200Mi 时，kubelet 将会开始驱逐 Pods。没有声明的其余配置项都将使用默认值，除非使用命令行参数来重载。命令行中的参数将会覆盖配置文件中的对应值。

作为一个小技巧，你可以从活动节点生成配置文件，相关方法请查看 [重新配置活动集群节点的 kubelet](#)。

启动通过配置文件配置的 Kubelet 进程

启动 Kubelet 需要将 --config 参数设置为 Kubelet 配置文件的路径。Kubelet 将从此文件加载其配置。

请注意，命令行参数与配置文件有相同的值时，就会覆盖配置文件中的该值。这有助于确保命令行 API 的向后兼容性。

请注意，kubelet 配置文件中的相对文件路径是相对于 kubelet 配置文件的位置解析的，而命令行参数中的相对路径是相对于 kubelet 的当前工作目录解析的。

请注意，命令行参数和 Kubelet 配置文件的某些默认值不同。如果设置了 --config，并且没有通过命令行指定值，则 KubeletConfiguration 版本的默认值生效。在上面的例子中，version 是 kubelet.config.k8s.io/v1beta1。

与动态 Kubelet 配置的关系

如果你正在使用[动态 kubelet 配置](#)特性，那么自动回滚机制将认为通过 --config 提供的配置与覆盖这些值的任何参数的组合是 "最后已知正常 (last known good)" 的配置。

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 August 13, 2020 at 5:49 PM PST: [\[zh\] Tidy up and fix links in tasks section \(5/10\) \(68abcb963\)](#)

配置 API 对象配额

本文讨论如何为 API 对象配置配额，包括 PersistentVolumeClaim 和 Service。配额限制了可以在命名空间中创建的特定类型对象的数量。你可以在 [ResourceQuota](#) 对象中指定配额。

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.

创建命名空间

创建一个命名空间以便本例中创建的资源和集群中的其余部分相隔离。

```
kubectl create namespace quota-object-example
```

创建 ResourceQuota

下面是一个 ResourceQuota 对象的配置文件：

[admin/resource/quota-objects.yaml](#)



```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-quota-demo
```

```
spec:  
  hard:  
    persistentvolumeclaims: "1"  
    services.loadbalancers: "2"  
    services.nodeports: "0"
```

创建 ResourceQuota :

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects.yaml --  
namespace=quota-object-example
```

查看 ResourceQuota 的详细信息 :

```
kubectl get resourcequota object-quota-demo --namespace=quota-object-  
example --output=yaml
```

输出结果表明在 quota-object-example 命名空间中，至多只能有一个 PersistentVolumeClaim，最多两个 LoadBalancer 类型的服务，不能有 NodePort 类型的服务。

```
status:  
  hard:  
    persistentvolumeclaims: "1"  
    services.loadbalancers: "2"  
    services.nodeports: "0"  
  used:  
    persistentvolumeclaims: "0"  
    services.loadbalancers: "0"  
    services.nodeports: "0"
```

创建 PersistentVolumeClaim

下面是一个 PersistentVolumeClaim 对象的配置文件：

[admin/resource/quota-objects-pvc.yaml](#)


```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: pvc-quota-demo  
spec:  
  storageClassName: manual  
  accessModes:  
    - ReadWriteOnce  
  resources:
```

```
requests:  
storage: 3Gi
```

创建 PersistentVolumeClaim :

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects-pvc.yaml  
--namespace=quota-object-example
```

确认已创建完 PersistentVolumeClaim :

```
kubectl get persistentvolumeclaims --namespace=quota-object-example
```

输出信息表明 PersistentVolumeClaim 存在并且处于 Pending 状态 :

NAME	STATUS
pvc-quota-demo	Pending

尝试创建第二个 PersistentVolumeClaim

下面是第二个 PersistentVolumeClaim 的配置文件 :

[admin/resource/quota-objects-pvc-2.yaml](#)


```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: pvc-quota-demo-2  
spec:  
  storageClassName: manual  
  accessModes:  
    - ReadWriteOnce  
  resources:  
    requests:  
      storage: 4Gi
```

尝试创建第二个 PersistentVolumeClaim :

```
kubectl create -f https://k8s.io/examples/admin/resource/quota-objects-  
pvc-2.yaml --namespace=quota-object-example
```

输出信息表明第二个 PersistentVolumeClaim 没有创建成功 , 因为这会超出命名空间的配额。

```
persistentvolumeclaims "pvc-quota-demo-2" is forbidden:  
exceeded quota: object-quota-demo, requested: persistentvolumeclaims=1,  
used: persistentvolumeclaims=1, limited: persistentvolumeclaims=1
```

说明

下面这些字符串可被用来标识那些能被配额限制的 API 资源：

String	API Object
"pods"	Pod
"services"	Service
"replicationcontrollers"	ReplicationController
"resourcequotas"	ResourceQuota
"secrets"	Secret
"configmaps"	ConfigMap
"persistentvolumeclaims"	PersistentVolumeClaim
"services.nodeports"	Service of type NodePort
"services.loadbalancers"	Service of type LoadBalancer

清理

删除你的命名空间：

```
kubectl delete namespace quota-object-example
```

接下来

集群管理员参考

- [为命名空间配置默认的内存请求和限制](#)
- [为命名空间配置默认的 CPU 请求和限制](#)
- [为命名空间配置内存的最小和最大限制](#)
- [为命名空间配置 CPU 的最小和最大限制](#)
- [为命名空间配置 CPU 和内存配额](#)
- [为命名空间配置 Pod 配额](#)

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)
- [为 Pod 配置服务质量](#)

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](#).

配置资源不足时的处理方式

本页介绍如何使用 kubelet 配置资源不足时的处理方式。

当可用计算资源较少时，kubelet 需要保证节点稳定性。这在处理如内存和硬盘之类的不可压缩资源时尤为重要。如果任意一种资源耗尽，节点将会变得不稳定。

驱逐信号

kubelet 支持按照以下表格中描述的信号触发驱逐决定。每个信号的值在 description 列描述，基于 kubelet 摘要 API。

驱逐信号	描述
memory.available	memory.available := node.status.capacity[memory] - node.stats.memory.workingSet
nodefs.available	nodefs.available := node.stats.fs.available
nodefs.inodesFree	nodefs.inodesFree := node.stats.fs.inodesFree
imagefs.available	imagefs.available := node.stats.runtime.imagefs.available
imagefs.inodesFree	imagefs.inodesFree := node.stats.runtime.imagefs.inodesFree

上面的每个信号都支持字面值或百分比的值。基于百分比的值的计算与每个信号对应的总容量相关。

memory.available 的值从 cgroupfs 获取，而不是通过类似 free -m 的工具。这很重要，因为 free -m 不能在容器中工作，并且如果用户使用了 [节点可分配资源](#) 特性，资源不足的判定将同时在本地 cgroup 层次结构的终端用户 Pod 部分和根节点做出。这个[脚本](#) 复现了与 kubelet 计算 memory.available 相同的步骤。kubelet 将 inactive_file（意即活动 LRU 列表上基于文件后端的内存字节数）从计算中排除，因为它假设内存在出现压力时将被回收。

kubelet 只支持两种文件系统分区。

1. nodefs 文件系统，kubelet 将其用于卷和守护程序日志等。
2. imagefs 文件系统，容器运行时用于保存镜像和容器可写层。

imagefs 可选。kubelet 使用 cAdvisor 自动发现这些文件系统。kubelet 不关心其它文件系统。当前不支持配置任何其它类型。例如，在专用 filesystem 中存储卷和日志是不可以的。

在将来的发布中，kubelet 将废除当前存在的 [垃圾回收](#) 机制，这种机制目前支持将驱逐操作作为对磁盘压力的响应。

驱逐阈值

kubelet支持指定驱逐阈值，用于触发 kubelet 回收资源。

每个阈值形式如下：

[eviction-signal][operator][quantity]

- 合法的 eviction-signal 标志如上所示。
- operator 是所需的关系运算符，例如 <。
- quantity 是驱逐阈值值标志，例如 1Gi。合法的标志必须匹配 Kubernetes 使用的数量表示。驱逐阈值也可以使用 % 标记表示百分比。

举例说明，如果一个节点有 10Gi 内存，希望在可用内存下降到 1Gi 以下时引起驱逐操作，则驱逐阈值可以使用下面任意一种方式指定（但不是两者同时）。

- memory.available<10%
- memory.available<1Gi

软驱逐阈值

软驱逐阈值使用一对由驱逐阈值和管理员必须指定的宽限期组成的配置对。在超过宽限期前，kubelet不会采取任何动作回收和驱逐信号关联的资源。如果没有提供宽限期，kubel et启动时将报错。

此外，如果达到了软驱逐阈值，操作员可以指定从节点驱逐 pod 时，在宽限期内允许结束的 pod 的最大数量。如果指定了 pod.Spec.TerminationGracePeriodSeconds 值，kubelet 将使用它和宽限期二者中较小的一个。如果没有指定，kubelet将立即终止 pod，而不会优雅结束它们。

软驱逐阈值的配置支持下列标记：

- eviction-soft 描述了驱逐阈值的集合（例如 memory.available<1.5Gi），如果在宽限期之外满足条件将触发 pod 驱逐。
- eviction-soft-grace-period 描述了驱逐宽限期的集合（例如 memory.available=1m30s），对应于在驱逐 pod 前软驱逐阈值应该被控制的时长。
- eviction-max-pod-grace-period 描述了当满足软驱逐阈值并终止 pod 时允许的最大宽限期值（秒数）。

硬驱逐阈值

硬驱逐阈值没有宽限期，一旦察觉，kubelet将立即采取行动回收关联的短缺资源。如果满足硬驱逐阈值，kubelet将立即结束 pod 而不是优雅终止。

硬驱逐阈值的配置支持下列标记：

- eviction-hard 描述了驱逐阈值的集合（例如 memory.available<1Gi），如果满足条件将触发 pod 驱逐。

kubelet 有如下所示的默认硬驱逐阈值：

- memory.available < 100Mi
- nodefs.available < 10%
- nodefs.inodesFree < 5%
- imagefs.available < 15%

驱逐监控时间间隔

kubelet 根据其配置的整理时间间隔计算驱逐阈值。

- housekeeping-interval 是容器管理时间间隔。

节点状态

kubelet 会将一个或多个驱逐信号映射到对应的节点状态。

如果满足硬驱逐阈值，或者满足独立于其关联宽限期的软驱逐阈值时，kubelet 将报告节点处于压力下的状态。

下列节点状态根据相应的驱逐信号定义。

节点状态	驱逐信号	描述
MemoryPressure	memory.available	节点上可用内存量达到逐出阈值
DiskPressure	nodefs.available, nodefs.inodesFree, imagefs.available, 或 imagefs.inodesFree	节点或者节点的根文件系统或镜像文件系统上可用磁盘空间和 i 节点个数达到逐出阈值

kubelet 将以 --node-status-update-frequency 指定的频率连续报告节点状态更新，其默认值为 10s。

节点状态振荡

如果节点在软驱逐阈值的上下振荡，但没有超过关联的宽限期时，将引起对应节点的状态持续在 true 和 false 间跳变，并导致不好的调度结果。

为了防止这种振荡，可以定义下面的标志，用于控制 kubelet 从压力状态中退出之前必须等待的时间。

- `eviction-pressure-transition-period` 是 kubelet 从压力状态中退出之前必须等待的时长。

kubelet 将确保在设定的时间段内没有发现和指定压力条件相对应的驱逐阈值被满足时，才会将状态变回 `false`。

回收节点层级资源

如果满足驱逐阈值并超过了宽限期，kubelet 将启动回收压力资源的过程，直到它发现低于设定阈值的信号为止。

kubelet 将尝试在驱逐终端用户 pod 前回收节点层级资源。发现磁盘压力时，如果节点针对容器运行时配置有独占的 `imagefs`，kubelet 回收节点层级资源的方式将会不同。

使用 `imagefs`

如果 `nodefs` 文件系统满足驱逐阈值，kubelet 通过驱逐 pod 及其容器来释放磁盘空间。

如果 `imagefs` 文件系统满足驱逐阈值，kubelet 通过删除所有未使用的镜像来释放磁盘空间。

未使用 `imagefs`

如果 `nodefs` 满足驱逐阈值，kubelet 将以下面的顺序释放磁盘空间：

1. 删除停止运行的 pod/container
2. 删除全部没有使用的镜像

驱逐最终用户的 pod

如果 kubelet 在节点上无法回收足够的资源，kubelet 将开始驱逐 pod。

kubelet 首先根据他们对短缺资源的使用是否超过请求来排除 pod 的驱逐行为，然后通过[优先级](#)，然后通过相对于 pod 的调度请求消耗急需的计算资源。

kubelet 按以下顺序对要驱逐的 pod 排名：

- BestEffort 或 Burstable，其对短缺资源的使用超过了其请求，此类 pod 按优先级排序，然后使用高于请求。
- Guaranteed pod 和 Burstable pod，其使用率低于请求，最后被驱逐。Guaranteed Pod 只有为所有的容器指定了要求和限制并且它们相等时才能得到保证。由于另一个 Pod 的资源消耗，这些 Pod 保证永远不会被驱逐。如果系统守护进程（例如 kubelet、docker、和 journald）消耗的资源多于通过 system-reserved 或 kube-reserved 分配保留的资源，并且该节点只有 Guaranteed 或 Burstable Pod 使用少于剩余的请求，然后节点必须选择驱逐这样的 Pod 以保持节点的稳定性并限制意外消耗对其他 pod 的影响。在这种情况下，它将首先驱逐优先级最低的 pod。

必要时，kubelet会在遇到 DiskPressure 时逐个驱逐 Pod 来回收磁盘空间。如果 kubelet 响应 inode 短缺，它会首先驱逐服务质量最低的 Pod 来回收 inodes。如果 kubelet 响应缺少可用磁盘，它会将 Pod 排在服务质量范围内，该服务会消耗大量的磁盘并首先结束这些磁盘。

使用 imagefs

如果是 nodefs 触发驱逐，kubelet 将按 nodefs 用量 - 本地卷 + pod 的所有容器日志的总和对其进行排序。

如果是 imagefs 触发驱逐，kubelet 将按 pod 所有可写层的用量对其进行排序。

未使用 imagefs

如果是 nodefs 触发驱逐，kubelet 会根据磁盘的总使用情况对 pod 进行排序 - 本地卷 + 所有容器的日志及其可写层。

最小驱逐回收

在某些场景，驱逐 pod 会导致回收少量资源。这将导致 kubelet 反复碰到驱逐阈值。除此之外，对如 disk 这类资源的驱逐时比较耗时的。

为了减少这类问题，kubelet 可以为每个资源配置一个 minimum-reclaim。当 kubelet 发现资源压力时，kubelet 将尝试至少回收驱逐阈值之下 minimum-reclaim 数量的资源。

例如使用下面的配置：

```
--eviction-
hard=memory.available<500Mi,nodefs.available<1Gi,imagefs.available<100Gi
--eviction-minimum-
reclaim="memory.available=0Mi,nodefs.available=500Mi,imagefs.available=2Gi"
```

如果 memory.available 驱逐阈值被触发，kubelet 将保证 memory.available 至少为 500Mi。对于 nodefs.available，kubelet 将保证 nodefs.available 至少为 1.5Gi。对于 imagefs.available，kubelet 将保证 imagefs.available 至少为 102Gi，直到不再有相关资源报告压力为止。

所有资源的默认 eviction-minimum-reclaim 值为 0。

调度器

当资源处于压力之下时，节点将报告状态。调度器将那种状态视为一种信号，阻止更多 pod 调度到这个节点上。

节点状态	调度器行为
MemoryPressure	新的 BestEffort Pod 不会被调度到该节点
DiskPressure	没有新的 Pod 会被调度到该节点

节点 OOM 行为

如果节点在 kubelet 回收内存之前经历了系统 OOM (内存不足) 事件 , 它将基于 [oom-killer](#) 做出响应。

kubelet 基于 pod 的 service 质量为每个容器设置一个 oom_score_adj 值。

Service 质量	oom_score_adj
Guaranteed	-998
BestEffort	1000
Burstable	$\min(\max(2, 1000 - (1000 * \text{memoryRequestBytes}) / \text{machineMemoryCapacityBytes}), 999)$

如果 kubelet 在节点经历系统 OOM 之前无法回收内存 , oom_killer 将基于它在节点上使用的内存百分比算出一个 oom_score , 并加上 oom_score_adj 得到容器的有效 oom_score , 然后结束得分最高的容器。

预期的行为应该是拥有最低服务质量并消耗和调度请求相关内存量最多的容器第一个被结束 , 以回收内存。

和 pod 驱逐不同 , 如果一个 Pod 的容器是被 OOM 结束的 , 基于其 RestartPolicy , 它可能会被 kubelet 重新启动。

最佳实践

以下部分描述了资源外处理的最佳实践。

可调度资源和驱逐策略

考虑以下场景 :

- 节点内存容量 : 10Gi
- 操作员希望为系统守护进程保留 10% 内存容量 (内核、kubelet 等) 。
- 操作员希望在内存用量达到 95% 时驱逐 pod , 以减少对系统的冲击并防止系统 OOM 的发生。

为了促成这个场景 , kubelet 将像下面这样启动 :

```
--eviction-hard=memory.available<500Mi  
--system-reserved=memory=1.5Gi
```

这个配置的暗示是理解系统保留应该包含被驱逐阈值覆盖的内存数量。

要达到这个容量 , 要么某些 pod 使用了超过它们请求的资源 , 要么系统使用的内存超过 $1.5Gi - 500Mi = 1Gi$ 。

这个配置将保证在 pod 使用量都不超过它们配置的请求值时，如果可能立即引起内存压力并触发驱逐时，调度器不会将 pod 放到这个节点上。

DaemonSet

我们永远都不希望 kubelet 驱逐一个从 DaemonSet 派生的 Pod，因为这个 Pod 将立即被重建并调度回相同的节点。

目前，kubelet没有办法区分一个 Pod 是由 DaemonSet 还是其他对象创建。如果/当这个信息可用时，kubelet 可能会预先将这些 pod 从提供给驱逐策略的候选集合中过滤掉。

总之，强烈推荐 DaemonSet 不要创建 BestEffort 的 Pod，防止其被识别为驱逐的候选 Pod。相反，理想情况下 DaemonSet 应该启动 Guaranteed 的 pod。

现有的回收磁盘特性标签已被弃用

kubelet 已经按需求清空了磁盘空间以保证节点稳定性。

当磁盘驱逐成熟时，下面的 kubelet 标志将被标记为废弃的，以简化支持驱逐的配置。

现有标签	新标签
--image-gc-high-threshold	--eviction-hard or eviction-soft
--image-gc-low-threshold	--eviction-minimum-reclaim
--maximum-dead-containers	deprecated
--maximum-dead-containers-per-container	deprecated
--minimum-container-ttl-duration	deprecated
--low-diskspace-threshold-mb	--eviction-hard or eviction-soft
--outofdisk-transition-frequency	--eviction-pressure-transition-period

已知问题

以下部分描述了与资源外处理有关的已知问题。

kubelet 可能无法立即发现内存压力

kubelet当前通过以固定的时间间隔轮询 cAdvisor 来收集内存使用数据。如果内存使用在那个时间窗口内迅速增长，kubelet可能不能足够快的发现 MemoryPressure，OOMKiller将不会被调用。我们准备在将来的发行版本中通过集成 memcg 通知 API 来减小这种延迟。当超过阈值时，内核将立即告诉我们。

如果您想处理可察觉的超量使用而不要求极端精准，可以设置驱逐阈值为大约 75% 容量作为这个问题的变通手段。这将增强这个特性的能力，防止系统 OOM，并提升负载卸载能力，以再次平衡集群状态。

kubelet 可能会驱逐超过需求数量的 pod

由于状态采集的时间差，驱逐操作可能驱逐比所需的更多的 pod。将来可通过添加从根容器获取所需状态的能力 <https://github.com/google/cadvisor/issues/1247> 来减缓这种状况。

反馈

此页是否对您有帮助？

是 否

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

最后修改 October 12, 2020 at 11:09 AM PST: [\[zh\] Sync changes to script location \(aa742890b\)](#)

限制存储消耗

此示例演示了一种限制名字空间中存储使用量的简便方法。

演示中用到了以下资源：[ResourceQuota](#)，[LimitRange](#) 和 [PersistentVolumeClaim](#)。

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.

场景：限制存储消耗

集群管理员代表用户群操作集群，管理员希望控制单个名称空间可以消耗多少存储空间以控制成本。

管理员想要限制：

1. 名字空间中持久卷申领 (persistent volume claims) 的数量
2. 每个申领 (claim) 可以请求的存储量
3. 名字空间可以具有的累计存储量

使用 LimitRange 限制存储请求

将 LimitRange 添加到名字空间会为存储请求大小强制设置最小值和最大值。 存储是通过 PersistentVolumeClaim 来发起请求的。 执行限制范围控制的准入控制器会拒绝任何高于或低于管理员所设阈值的 PVC。

在此示例中，请求 10Gi 存储的 PVC 将被拒绝，因为它超过了最大 2Gi。

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

当底层存储提供程序需要某些最小值时，将会用到所设置最小存储请求值。 例如，AWS EBS volumes 的最低要求为 1Gi。

使用 StorageQuota 限制 PVC 数目和累计存储容量

管理员可以限制某个名字空间中的 PVCs 个数以及这些 PVCs 的累计容量。 新 PVCs 请求如果超过任一上限值将被拒绝。

在此示例中，名字空间中的第 6 个 PVC 将被拒绝，因为它超过了最大计数 5。 或者，当与上面的 2Gi 最大容量限制结合在一起时，意味着 5Gi 的最大配额不能支持 3 个都是 2Gi 的 PVC。 后者实际上是向名字空间请求 6Gi 容量，而该命令空间已经设置上限为 5Gi。

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storagequota
spec:
  hard:
    persistentvolumeclaims: "5"
    requests.storage: "5Gi"
```

小结

限制范围对象可以用来设置可请求的存储量上限，而资源配置对象则可以通过申领计数和累计存储容量有效地限制名字空间耗用的存储量。 这两种机制使得集群管理员能够规划其集群存储预算而不会发生任一项目超量分配的风险。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 07, 2021 at 3:10 AM PST: [block yaml. \(4675779b4\)](#)

静态加密 Secret 数据

本文展示如何启用和配置静态 Secret 数据的加密

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.
- 需要 etcd v3 或者更高版本

配置并确定是否已启用静态数据加密

kube-apiserver 的参数 --experimental-encryption-provider-config 控制 API 数据在 etcd 中的加密方式。下面提供一个配置示例。

理解静态数据加密

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - identity: {}
    - aesgcm:
      keys:
        - name: key1
          secret: c2VjcmV0IGIzIHNIY3VyZQ==
        - name: key2
```

```

secret: dGhpcyBpcyBwYXNzd29yZA==
- aescbc:
  keys:
  - name: key1
    secret: c2VjcmV0IGIzIHNIY3VyZQ==
  - name: key2
    secret: dGhpcyBpcyBwYXNzd29yZA==
- secretbox:
  keys:
  - name: key1
    secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=

```

每个 resources 数组项目是一个单独的完整的配置。 resources.resources 字段是要加密的 Kubernetes 资源名称 (resource 或 resource.group) 的数组。 providers 数组是可能的加密 provider 的有序列表。 每个条目只能指定一个 provider 类型 (可以是 identity 或 aescbc , 但不能在同一个项目中同时指定) 。

列表中的第一个 provider 用于加密进入存储的资源。 当从存储器读取资源时 , 与存储的数据匹配的所有 provider 将按顺序尝试解密数据。 如果由于格式或密钥不匹配而导致没有 provider 能够读取存储的数据 , 则会返回一个错误 , 以防止客户端访问该资源。

注意 : 重要 : 如果通过加密配置无法读取资源 (因为密钥已更改) , 唯一的方法是直接从底层 etcd 中删除该密钥。 任何尝试读取资源的调用将会失败 , 直到它被删除或提供有效的解密密钥。

Providers:

名称	加密类型	强度	速度	密钥长度	其它事项
identity	无	N/A	N/A	N/A	不加密写入的资源。当设置为第一个 provider 时 , 资源将在新值写入时被解密。
aescbc	填充 PKCS#7 的 AES-CBC	最强	快	32字节	建议使用的加密项 , 但可能比 secretbox 稍微慢一些。
secretbox	XSalsa20 和 Poly1305	强	更快	32字节	较新的标准 , 在需要高度评审的环境中可能不被接受。
aesgcm	带有随机数的 AES-GCM	必须每 200k 写入一次	最快	16, 24 或者 32字节	建议不要使用 , 除非实施了自动密钥循环方案。

名称	加密类型	强度	速度	密钥长度	其它事项
kms	使用信封加密方案：数据使用带有 PKCS#7 填充的 AES-CBC 通过数据加密密钥（DEK）加密，DEK 根据 Key Management Service（KMS）中的配置通过密钥加密密钥（Key Encryption Keys，KEK）加密	最强	快	32字节	建议使用第三方工具进行密钥管理。为每个加密生成新的 DEK，并由用户控制 KEK 轮换来简化密钥轮换。 配置 KMS 提供程序

每个 provider 都支持多个密钥 - 在解密时会按顺序使用密钥，如果是第一个 provider，则第一个密钥用于加密。

在 EncryptionConfig 中保存原始的加密密钥与不加密相比只会略微地提升安全级别。请使用 kms 驱动以获得更强的安全性。 默认情况下，identity 驱动被用来对 etcd 中的 Secret 提供保护，而这个驱动不提供加密能力。EncryptionConfiguration 的引入是为了能够使用本地管理的密钥来在本地加密 Secret 数据。

使用本地管理的密钥来加密 Secret 能够保护数据免受 etcd 破坏的影响，不过无法针对主机被侵入提供防护。这是因为加密的密钥保存在主机上的 EncryptionConfig YAML 文件中，有经验的入侵者 仍能访问该文件并从中提取出加密密钥。

封套加密（Envelope Encryption）引入了对独立密钥的依赖，而这个密钥并不保存在 Kubernetes 中。在这种情况下下，入侵者需要攻破 etcd、kube-apiserver 和第三方的 KMS 驱动才能获得明文数据，因而这种方案提供了比本地保存加密密钥更高的安全级别。

加密你的数据

创建一个新的加密配置文件：

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - aescbc:
      keys:
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
        - identity: {}
```

遵循如下步骤来创建一个新的 secret：

1. 生成一个 32 字节的随机密钥并进行 base64 编码。如果你在 Linux 或 Mac OS X 上，请运行以下命令：

```
head -c 32 /dev/urandom | base64
```

1. 将这个值放入到 secret 字段中。
2. 设置 kube-apiserver 的 --experimental-encryption-provider-config 参数，将其指向配置文件所在位置。
3. 重启你的 API server。

注意：你的配置文件包含可以解密 etcd 内容的密钥，因此你必须正确限制主控节点的访问权限，以便只有能运行 kube-apiserver 的用户才能读取它。

验证数据已被加密

数据在写入 etcd 时会被加密。重新启动你的 kube-apiserver 后，任何新创建或更新的密码在存储时都应该被加密。如果想要检查，你可以使用 etcdctl 命令行程序来检索你的加密内容。

1. 创建一个新的 secret，名称为 secret1，命名空间为 default：

```
kubectl create secret generic secret1 -n default --from-literal=mykey=mydata
```

1. 使用 etcdctl 命令行，从 etcd 中读取 secret：

```
ETCDCTL_API=3 etcdctl get /registry/secrets/default/secret1 [...] | hexdump -C
```

这里的 [...] 是用来连接 etcd 服务的额外参数。

1. 验证存储的密钥前缀是否为 k8s:enc:aescbc:v1:，这表明 aescbc provider 已加密结果数据。
2. 通过 API 检索，验证 secret 是否被正确解密：

```
kubectl describe secret secret1 -n default
```

其输出应该是 mykey: bXlkYXRh，mydata 数据是被加密过的，请参阅 [解密 Secret](#) 了解如何完全解码 Secret 内容。

确保所有 Secret 都被加密

由于 Secret 是在写入时被加密，因此对 Secret 执行更新也会加密该内容。

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

上面的命令读取所有 Secret，然后使用服务端加密来更新其内容。

说明：如果由于冲突写入而发生错误，请重试该命令。对于较大的集群，你可能希望通过命名空间或更新脚本来对 Secret 进行划分。

轮换解密密钥

在不发生停机的情况下更改 Secret 需要多步操作，特别是在有多个 kube-apiserver 进程正在运行的 高可用环境中。

1. 生成一个新密钥并将其添加为所有服务器上当前提供程序的第二个密钥条目
2. 重新启动所有 kube-apiserver 进程以确保每台服务器都可以使用新密钥进行解密
3. 将新密钥设置为 keys 数组中的第一个条目，以便在配置中使用其进行加密
4. 重新启动所有 kube-apiserver 进程以确保每个服务器现在都使用新密钥进行加密
5. 运行 `kubectl get secrets --all-namespaces -o json | kubectl replace -f -` 以用新密钥加密所有现有的秘密
6. 在使用新密钥备份 etcd 后，从配置中删除旧的解密密钥并更新所有密钥

如果只有一个 kube-apiserver，第 2 步可能可以忽略。

解密所有数据

要禁用 rest 加密，请将 identity provider 作为配置中的第一个条目：

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - identity: {}
    - aescbc:
      keys:
        - name: key1
        secret: <BASE 64 ENCODED SECRET>
```

并重新启动所有 kube-apiserver 进程。然后运行：

```
kubectl get secrets -all-namespaces -o json | kubectl replace -f -`
```

以强制解密所有 secret。

反馈

此页是否对您有帮助？

是 否

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

最后修改 October 05, 2020 at 5:18 PM PST: [Update encrypt-data.md \(0997416af\)](#)

配置 Pods 和容器

对 Pod 和容器执行常见的配置任务。

[为容器和 Pod 分配内存资源](#)

[为 Windows Pod 和容器配置 GMSA](#)

[为 Windows 的 Pod 和容器配置 RunAsUserName](#)

[为容器和 Pods 分配 CPU 资源](#)

[配置 Pod 的服务质量](#)

[为容器分派扩展资源](#)

[配置 Pod 以使用卷进行存储](#)

[配置 Pod 以使用 PersistentVolume 作为存储](#)

[配置 Pod 使用投射卷作存储](#)

[为 Pod 或容器配置安全性上下文](#)

[为 Pod 配置服务账户](#)

[从私有仓库拉取镜像](#)

[配置存活、就绪和启动探测器](#)

[将 Pod 分配给节点](#)

[用节点亲和性把 Pods 分配到节点](#)

[配置 Pod 初始化](#)

[为容器的生命周期事件设置处理函数](#)

[配置 Pod 使用 ConfigMap](#)

[在 Pod 中的容器之间共享进程命名空间](#)

[创建静态 Pod](#)

[将 Docker Compose 文件转换为 Kubernetes 资源](#)

为容器和 Pod 分配内存资源

此页面展示如何将内存 请求 (request) 和内存 限制 (limit) 分配给一个容器。 我们保障容器拥有它请求数量的内存，但不允许使用超过限制数量的内存。

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.

你集群中的每个节点必须拥有至少 300 MiB 的内存。

该页面上的一些步骤要求你在集群中运行 [metrics-server](#) 服务。 如果你已经有在运行中的 metrics-server，则可以跳过这些步骤。

如果你运行的是 Minikube，可以运行下面的命令启用 metrics-server：

```
minikube addons enable metrics-server
```

要查看 metrics-server 或资源指标 API (metrics.k8s.io) 是否已经运行，请运行以下命令：

```
kubectl get apiservices
```

如果资源指标 API 可用，则输出结果将包含对 metrics.k8s.io 的引用信息。

```
NAME
v1beta1.metrics.k8s.io
```

创建命名空间

创建一个命名空间，以便将本练习中创建的资源与集群的其余部分隔离。

```
kubectl create namespace mem-example
```

指定内存请求和限制

要为容器指定内存请求，请在容器资源清单中包含 resources : requests 字段。 同理，要指定内存限制，请包含 resources : limits。

在本练习中，你将创建一个拥有一个容器的 Pod。 容器将会请求 100 MiB 内存，并且内存会被限制在 200 MiB 以内。 这是 Pod 的配置文件：

[pods/resource/memory-request-limit.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-ctr
      image: polinux/stress
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
      command: ["stress"]
      args: [--vm, "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

配置文件的 args 部分提供了容器启动时的参数。"--vm-bytes", "150M" 参数告知容器尝试分配 150 MiB 内存。

开始创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit.yaml --namespace=mem-example
```

验证 Pod 中的容器是否已运行：

```
kubectl get pod memory-demo --namespace=mem-example
```

查看 Pod 相关的详细信息：

```
kubectl get pod memory-demo --output=yaml --namespace=mem-example
```

输出结果显示：该 Pod 中容器的内存请求为 100 MiB，内存限制为 200 MiB。

```
...
resources:
  limits:
    memory: 200Mi
  requests:
    memory: 100Mi
...
```

运行 kubectl top 命令，获取该 Pod 的指标数据：

```
kubectl top pod memory-demo --namespace=mem-example
```

输出结果显示：Pod 正在使用的内存大约为 162,900,000 字节，约为 150 MiB。这大于 Pod 请求的 100 MiB，但在 Pod 限制的 200 MiB 之内。

NAME	CPU(cores)	MEMORY(bytes)
memory-demo	<something>	162856960

删除 Pod：

```
kubectl delete pod memory-demo --namespace=mem-example
```

超过容器限制的内存

当节点拥有足够的可用内存时，容器可以使用其请求的内存。但是，容器不允许使用超过其限制的内存。如果容器分配的内存超过其限制，该容器会成为被终止的候选容器。如果容器继续消耗超出其限制的内存，则终止容器。如果终止的容器可以被重启，则 kubelet 会重新启动它，就像其他任何类型的运行时失败一样。

在本练习中，你将创建一个 Pod，尝试分配超出其限制的内存。这是一个 Pod 的配置文件，其拥有一个容器，该容器的内存请求为 50 MiB，内存限制为 100 MiB：

[pods/resource/memory-request-limit-2.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-2-ctr
      image: polinux/stress
      resources:
        requests:
          memory: "50Mi"
        limits:
          memory: "100Mi"
      command: ["stress"]
      args: [--vm, "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

在配置文件的 args 部分中，你可以看到容器会尝试分配 250 MiB 内存，这远高于 100 MiB 的限制。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit-2.yaml --namespace=mem-example
```

查看 Pod 相关的详细信息：

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

此时，容器可能正在运行或被杀死。重复前面的命令，直到容器被杀掉：

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-2	0/1	OOMKilled	1	24s

获取容器更详细的状态信息：

```
kubectl get pod memory-demo-2 --output=yaml --namespace=mem-example
```

输出结果显示：由于内存溢出（OOM），容器已被杀掉：

```
lastState:  
  terminated:  
    containerID: docker://  
65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f  
    exitCode: 137  
    finishedAt: 2017-06-20T20:52:19Z  
    reason: OOMKilled  
    startedAt: null
```

本练习中的容器可以被重启，所以 kubelet 会重启它。多次运行下面的命令，可以看到容器在反复的被杀死和重启：

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

输出结果显示：容器被杀掉、重启、再杀掉、再重启……：

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-2	0/1	OOMKilled	1	37s

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-2	1/1	Running	2	40s

查看关于该 Pod 历史的详细信息：

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

输出结果显示：该容器反复的在启动和失败：

```
... Normal Created  Created container with id  
66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511  
... Warning BackOff  Back-off restarting failed container
```

查看关于集群节点的详细信息：

```
kubectl describe nodes
```

输出结果包含了一条练习中的容器由于内存溢出而被杀掉的记录：

```
Warning OOMKilling Memory cgroup out of memory: Kill process 4481 (stress)  
score 1994 or sacrifice child
```

删除 Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

超过整个节点容量的内存

内存请求和限制是与容器关联的，但将 Pod 视为具有内存请求和限制，也是很有用的。Pod 的内存请求是 Pod 中所有容器的内存请求之和。同理，Pod 的内存限制是 Pod 中所有容器的内存限制之和。

Pod 的调度基于请求。只有当节点拥有足够满足 Pod 内存请求的内存时，才会将 Pod 调度至节点上运行。

在本练习中，你将创建一个 Pod，其内存请求超过了你集群中的任意一个节点所拥有的内存。这是该 Pod 的配置文件，其拥有一个请求 1000 GiB 内存的容器，这应该超过了你集群中任何节点的容量。

[pods/resource/memory-request-limit-3.yaml](#)


```
apiVersion: v1  
kind: Pod  
metadata:  
  name: memory-demo-3  
  namespace: mem-example  
spec:  
  containers:  
    - name: memory-demo-3-ctr  
      image: polinux/stress  
      resources:  
        limits:  
          memory: "1000Gi"  
        requests:  
          memory: "1000Gi"  
        command: ["stress"]  
        args: [--vm, "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit-3.yaml --namespace=mem-example
```

查看 Pod 状态：

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

输出结果显示：Pod 处于 PENDING 状态。这意味着，该 Pod 没有被调度至任何节点上运行，并且它会无限期的保持该状态：

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-3	0/1	Pending	0	25s

查看关于 Pod 的详细信息，包括事件：

```
kubectl describe pod memory-demo-3 --namespace=mem-example
```

输出结果显示：由于节点内存不足，该容器无法被调度：

Events:

... Reason	Message
------------	---------

... FailedScheduling	No nodes are available that match all of the following predicates:: Insufficient memory (3).
----------------------	--

内存单位

内存资源的基本单位是字节 (byte)。你可以使用这些后缀之一，将内存表示为 纯整数或定点整数：E、P、T、G、M、K、Ei、Pi、Ti、Gi、Mi、Ki。例如，下面是一些近似相同的值：

128974848, 129e6, 129M , 123Mi

删除 Pod：

```
kubectl delete pod memory-demo-3 --namespace=mem-example
```

如果你没有指定内存限制

如果你没有为一个容器指定内存限制，则自动遵循以下情况之一：

- 容器可无限制地使用内存。容器可以使用其所在节点所有的可用内存，进而可能导致该节点调用 OOM Killer。此外，如果发生 OOM Kill，没有资源限制的容器将被杀掉的可行性更大。
- 运行的容器所在命名空间有默认的内存限制，那么该容器会被自动分配默认限制。集群管理员可用使用 [LimitRange](#) 来指定默认的内存限制。

内存请求和限制的目的

通过为集群中运行的容器配置内存请求和限制，你可以有效利用集群节点上可用的内存资源。通过将 Pod 的内存请求保持在较低水平，你可以更好地安排 Pod 调度。通过让内存限制大于内存请求，你可以完成两件事：

- Pod 可以进行一些突发活动，从而更好的利用可用内存。
- Pod 在突发活动期间，可使用的内存被限制为合理的数量。

清理

删除命名空间。下面的命令会删除你根据这个任务创建的所有 Pod：

```
kubectl delete namespace mem-example
```

接下来

应用开发者扩展阅读

- [为容器和 Pod 分配 CPU 资源](#)
- [配置 Pod 的服务质量](#)

集群管理员扩展阅读

- [为命名空间配置默认的内存请求和限制](#)
- [为命名空间配置默认的 CPU 请求和限制](#)
- [配置命名空间的最小和最大内存约束](#)
- [配置命名空间的最小和最大 CPU 约束](#)
- [为命名空间配置内存和 CPU 配额](#)
- [配置命名空间下 Pod 总数](#)
- [配置 API 对象配额](#)

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 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

为 Windows Pod 和容器配置 GMSA

FEATURE STATE: Kubernetes v1.18 [stable]

本页展示如何为将运行在 Windows 节点上的 Pod 和容器配置 [组管理的服务账号 \(Group Managed Service Accounts , GMSA \)](#)。组管理的服务账号是活动目录 (Active Directory) 的一种特殊类型，提供自动化的 密码管理、简化的服务主体名称 (Service Principal Name , SPN) 管理以及跨多个 服务器将管理操作委派给其他管理员等能力。

在 Kubernetes 环境中，GMSA 凭据规约配置为 Kubernetes 集群范围的自定义资源 (Custom Resources) 形式。Windows Pod 以及各 Pod 中的每个容器可以配置为 使用 GMSA 来完成基于域 (Domain) 的操作 (例如，Kerberos 身份认证)，以便 与 其他 Windows 服务相交互。自 Kubernetes 1.16 版本起，Docker 运行时为 Windows 负载支持 GMSA。

准备开始

你需要一个 Kubernetes 集群，以及 kubectl 命令行工具，且工具必须已配置 为能够与 你的集群通信。集群预期包含 Windows 工作节点。本节讨论需要为每个集群执行一次 的初始操作。

安装 GMSACredentialSpec CRD

你需要在集群上配置一个用于 GMSA 凭据规约资源的 [CustomResourceDefinition](#)(CRD)，以便定义类型为 GMSACredentialSpec 的自定义资源。首先下载 GMSA CRD [YAML](#) 并将其保存为 gmsa-crd.yaml。接下来执行 kubectl apply -f gmsa-crd.yaml 安装 CRD。

安装 Webhook 来验证 GMSA 用户

你需要为 Kubernetes 集群配置两个 Webhook，在 Pod 或容器级别填充和检查 GMSA 凭据规约引用。

1. 一个修改模式 (Mutating) 的 Webhook，将对 GMSA 的引用 (在 Pod 规约中 体现为名字) 展开为完整凭据规约的 JSON 形式，并保存回 Pod 规约中。
2. 一个验证模式 (Validating) 的 Webhook，确保对 GMSA 的所有引用都是已经 授权 给 Pod 的服务账号使用的。

安装以上 Webhook 及其相关联的对象需要执行以下步骤：

1. 创建一个证书密钥对 (用于允许 Webhook 容器与集群通信)
2. 安装一个包含如上证书的 Secret
3. 创建一个包含核心 Webhook 逻辑的 Deployment
4. 创建引用该 Deployment 的 Validating Webhook 和 Mutating Webhook 配置

你可以使用[这个脚本](#) 来部署和配置上述 GMSA Webhook 及相关联的对象。你还可以在运行脚本时设置 --dry-run=server 选项以便审查脚本将会对集群做出的变更。

脚本所使用的[YAML 模板](#) 也可用于手动部署 Webhook 及相关联的对象，不过需要对其中的参数作适当替换。

在活动目录中配置 GMSA 和 Windows 节点

在配置 Kubernetes 中的 Pod 以使用 GMSA 之前，需要按 [Windows GMSA 文档](#) 中描述的那样先在活动目录中准备好期望的 GMSA。Windows 工作节点（作为 Kubernetes 集群的一部分）需要被配置到活动目录中，以便 访问与期望的 GSMA 相关联的秘密凭据数据。这一操作的描述位于 [Windows GMSA 文档](#) 中。

创建 GMSA 凭据规约资源

当（如前所述）安装了 GMSACredentialSpec CRD 之后，你就可以配置包含 GMSA 凭据 规约的自定义资源了。GMSA 凭据规约中并不包含秘密或敏感数据。其中包含的信息主要用于容器运行时，便于后者向 Windows 描述容器所期望的 GMSA。GMSA 凭据规约可以使用 [PowerShell 脚本](#) 以 YAML 格式生成。

下面是手动以 JSON 格式生成 GMSA 凭据规约并对其进行 YAML 转换的步骤：

1. 导入 CredentialSpec 模块: ipmo CredentialSpec.psm1
2. 使用 New-CredentialSpec 来创建一个 JSON 格式的凭据规约。要创建名为 WebApp1 的 GMSA 凭据规约，调用 New-CredentialSpec -Name WebApp1 -AccountName WebApp1 -Domain \$(Get-ADDomain -CurrentLocalComputer)。
3. 使用 Get-CredentialSpec 来显示 JSON 文件的路径。
4. 将凭据规约从 JSON 格式转换为 YAML 格式，并添加必要的头部字段 apiVersion、kind、metadata 和 credspec，使其成为一个可以在 Kubernetes 中配置的 GMSACredentialSpec 自定义资源。

下面的 YAML 配置描述的是一个名为 gmsa-WebApp1 的 GMSA 凭据规约：

```
apiVersion: windows.k8s.io/v1alpha1
kind: GMSACredentialSpec
metadata:
  name: gmsa-WebApp1 # 这是随意起的一个名字，将用作引用
  credspec:
    ActiveDirectoryConfig:
      GroupManagedServiceAccounts:
        - Name: WebApp1 # GMSA 账号的用户名
          Scope: CONTOSO # NETBIOS 域名
        - Name: WebApp1 # GMSA 账号的用户名
          Scope: contoso.com # DNS 域名
```

CmsPlugins:

- ActiveDirectory

DomainJoinConfig:

DnsName: contoso.com # DNS 域名

DnsTreeName: contoso.com # DNS 域名根

Guid: 244818ae-87ac-4fcd-92ec-e79e5252348a # GUID

MachineAccountName: WebApp1 # GMSA 账号的用户名

NetBiosName: CONTOSO # NETBIOS 域名

Sid: S-1-5-21-2126449477-2524075714-3094792973 # GMSA 的 SID

上面的凭据规约资源可以保存为 gmsa-Webapp1-credspec.yaml，之后使用 kubectl apply -f gmsa-Webapp1-credspec.yaml 应用到集群上。

配置集群角色以启用对特定 GMSA 凭据规约的 RBAC

你需要为每个 GMSA 凭据规约资源定义集群角色。该集群角色授权某主体（通常是一个服务账号）对特定的 GMSA 资源执行 use 动作。下面的示例显示的是一个集群角色，对前文创建的凭据规约 gmsa-WebApp1 执行鉴权。将此文件保存为 gmsa-webapp1-role.yaml 并执行 kubectl apply -f gmsa-webapp1-role.yaml。

创建集群角色读取凭据规约

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: webapp1-role

rules:

- apiGroups: ["windows.k8s.io"]

resources: ["gmsacredentialspecs"]

verbs: ["use"]

resourceNames: ["gmsa-WebApp1"]

将角色指派给要使用特定 GMSA 凭据规约的服务账号

你需要将某个服务账号（Pod 配置所对应的那个）绑定到前文创建的集群角色上。这一绑定操作实际上授予该服务账号使用所指定的 GMSA 凭据规约资源的访问权限。下面显示的是一个绑定到集群角色 webapp1-role 上的 default 服务账号，使之能够使用前面所创建的 gmsa-WebApp1 凭据规约资源。

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: allow-default-svc-account-read-on-gmsa-WebApp1

namespace: default

subjects:

- kind: ServiceAccount

name: default

```
namespace: default
roleRef:
  kind: ClusterRole
  name: webapp1-role
  apiGroup: rbac.authorization.k8s.io
```

在 Pod 规约中配置 GMSA 凭据规约引用

Pod 规约字段 `securityContext.windowsOptions.gmsaCredentialSpecName` 可用来设置对指定 GMSA 凭据规约自定义资源的引用。设置此引用将会配置 Pod 中的所有容器使用所给的 GMSA。下面是一个 Pod 规约示例，其中包含了对 gmsa-WebApp1 凭据规约的引用：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      securityContext:
        windowsOptions:
          gmsaCredentialSpecName: gmsa-webapp1
      containers:
        - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
          imagePullPolicy: Always
          name: iis
      nodeSelector:
        kubernetes.io/os: windows
```

Pod 中的各个容器也可以使用对应容器的 `securityContext.windowsOptions.gmsaCredentialSpecName` 字段来设置期望使用的 GMSA 凭据规约。例如：

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  labels:
    run: with-creds
  name: with-creds
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: with-creds
  template:
    metadata:
      labels:
        run: with-creds
    spec:
      containers:
        - image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
          imagePullPolicy: Always
          name: iis
          securityContext:
            windowsOptions:
              gmsaCredentialSpecName: gmsa-Webapp1
          nodeSelector:
            kubernetes.io/os: windows
```

当 Pod 规约中填充了 GMSA 相关字段（如上所述），在集群中应用 Pod 规约时会依次发生以下事件：

1. Mutating Webhook 解析对 GMSA 凭据规约资源的引用，并将其全部展开，得到 GMSA 凭据规约的实际内容。
2. Validating Webhook 确保与 Pod 相关联的服务账号有权在所给的 GMSA 凭据规约上执行 use 动作。
3. 容器运行时为每个 Windows 容器配置所指定的 GMSA 凭据规约，这样容器就可以以活动目录中该 GMSA 所代表的身份来执行操作，使用该身份来访问域中的服务。

故障排查

如果在你的环境中配置 GMSA 时遇到了困难，你可以采取若干步骤来排查可能的故障。

首先，确保凭据规约已经被传递到 Pod。要实现这点，你需要先通过 exec 进入到你的 Pod 之一，检查 `nltest.exe /parentdomain` 命令的输出。在下面的例子中，Pod 未能正确地获得凭据规约：

```
kubectl exec -it iis-auth-7776966999-n5nzb powershell.exe
```

Windows PowerShell

Copyright (C) Microsoft Corporation. All rights reserved.

```
PS C:\> nltest.exe /parentdomain
```

```
Getting parent domain failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

```
PS C:\>
```

如果 Pod 未能正确获得凭据规约，则下一步就要检查与域之间的通信。首先，从 Pod 内部快速执行一个 nslookup 操作，找到域根。

这一操作会告诉我们三件事情：

1. Pod 能否访问域控制器 (DC)
2. DC 能否访问 Pod
3. DNS 是否正常工作

如果 DNS 和通信测试通过，接下来你需要检查是否 Pod 已经与域之间建立了 安全通信通道。要执行这一检查，你需要再次通过 exec 进入到你的 Pod 中并执行 nltest.exe /query 命令。

```
PS C:\> nltest.exe /query
```

```
I_NetLogonControl failed: Status = 1722 0x6ba RPC_S_SERVER_UNAVAILABLE
```

这一输出告诉我们，由于某些原因，Pod 无法使用凭据规约中的账号登录到域。你可以通过运行 nltest.exe /sc_reset:domain.example 命令尝试修复安全通道。

```
PS C:\> nltest /sc_reset:domain.example
```

```
Flags: 30 HAS_IP HAS_TIMESERV
```

```
Trusted DC Name \\dc10.domain.example
```

```
Trusted DC Connection Status Status = 0 0x0 NERR_Success
```

```
The command completed successfully
```

```
PS C:\>
```

如果上述命令修复了错误，你就可以通过向你的 Pod 规约添加生命周期回调来将此操作自动化。如果上述命令未能奏效，你就需要再次检查凭据规约，以确保其数据时正确的而且是完整的。

```
image: registry.domain.example/iis-auth:1809v1
```

```
lifecycle:
```

```
  postStart:
```

```
    exec:
```

```
      command: ["powershell.exe", "-command", "do { Restart-Service -Name netlogon } while ( $($Result = (nltest.exe /query); if ($Result -like '*0x0 NERR_Success') {return $true} else {return $false}) -eq $false)"]
```

```
  imagePullPolicy: IfNotPresent
```

如果你向你的 Pod 规约中添加如上所示的 lifecycle 节，则 Pod 会自动执行所列举的命令来重启 netlogon 服务，直到 nltest.exe /query 命令返回时没有错误信息。

GMSA 的局限

在使用 [Windows 版本的 ContainerD 运行时](#) 时，通过 GMSA 域身份标识访问受限制的网络共享资源时会出错。容器会收到身份标识且 nltest.exe /query 调用也能正常工作。当需要访问网络共享资源时，建议使用 [Docker EE 运行时](#)。Windows Server 团队正在 Windows 内核中解决这一问题，并在将来发布解决此问题的补丁。你可以在 [Microsoft Windows Containers 问题跟踪列表](#) 中查找这类更新。

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 22, 2020 at 11:26 AM PST: [\[zh\] Translate docs/tasks/configure-pod-container/configure-gmsa.md \(4abcae949\)](#)

为 Windows 的 Pod 和容器配置 RunAsUserName

FEATURE STATE: Kubernetes v1.18 [stable]

本页展示如何为运行为在 Windows 节点上运行的 Pod 和容器配置 RunAsUserName。大致相当于 Linux 上的 runAsUser，允许在容器中以与默认值不同的用户名运行应用。

准备开始

你必须有一个 Kubernetes 集群，并且 kubectl 必须能和集群通信。集群应该要有 Windows 工作节点，将在其中调度运行 Windows 工作负载的 pod 和容器。

为 Pod 设置 Username

要指定运行 Pod 容器时所使用的用户名，请在 Pod 声明中包含 securityContext ([PodSecurityContext](#)) 字段，并在其内部包含 windowsOptions ([WindowsSecurityContextOptions](#)) 字段的 runAsUserName 字段。

你为 Pod 指定的 Windows SecurityContext 选项适用于该 Pod 中（包括 init 容器）的所有容器。

这儿有一个已经设置了 runAsUserName 字段的 Windows Pod 的配置文件：

[windows/run-as-username-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
    - name: run-as-username-demo
      image: mcr.microsoft.com/windows/servercore:ltsc2019
      command: ["ping", "-t", "localhost"]
  nodeSelector:
    kubernetes.io/os: windows
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-pod.yaml
```

验证 Pod 容器是否在运行：

```
kubectl get pod run-as-username-pod-demo
```

获取该容器的 shell：

```
kubectl exec -it run-as-username-pod-demo -- powershell
```

检查运行 shell 的用户的用户名是否正确：

```
echo $env:USERNAME
```

输出结果应该是这样：

```
ContainerUser
```

为容器设置 Username

要指定运行容器时所使用的用户名，请在容器清单中包含 securityContext（[SecurityContext](#)）字段，并在其内部包含 windowsOptions（[WindowsSecurityContextOptions](#)）字段的 runAsUserName 字段。

你为容器指定的 Windows SecurityContext 选项仅适用于该容器，并且它会覆盖 Pod 级别设置。

这里有一个 Pod 的配置文件，其中只有一个容器，并且在 Pod 级别和容器级别都设置了 runAsUserName：

[windows/run-as-username-container.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command: ["ping", "-t", "localhost"]
    securityContext:
      windowsOptions:
        runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/windows/run-as-username-
container.yaml
```

验证 Pod 容器是否在运行：

```
kubectl get pod run-as-username-container-demo
```

获取该容器的 shell：

```
kubectl exec -it run-as-username-container-demo -- powershell
```

检查运行 shell 的用户的用户名是否正确（应该是容器级别设置的那个）：

```
echo $env:USERNAME
```

输出结果应该是这样：

```
ContainerAdministrator
```

Windows Username 的局限性

想要使用此功能，在 runAsUserName 字段中设置的值必须是有效的用户名。它必须是 DOMAIN\USER 这种格式，其中 DOMAIN\ 是可选的。Windows 用户名不区分大小写。此外，关于 DOMAIN 和 USER 还有一些限制：

- runAsUserName 字段不能为空，并且不能包含控制字符（ASCII 值：0x00-0x1F、0x7F）
- DOMAIN 必须是 NetBios 名称或 DNS 名称，每种名称都有各自的局限性：
 - NetBios 名称：最多 15 个字符，不能以 .(点) 开头，并且不能包含以下字符：\ / : * ? < > |
 - DNS 名称：最多 255 个字符，只能包含字母、数字、点和中划线，并且不能以 .(点) 或 -(中划线) 开头和结尾。
- USER 最多不超过 20 个字符，不能 只 包含点或空格，并且不能包含以下字符：
"/ \ [] : ; | = , + * ? < > @

runAsUserName 字段接受的值的一些示例：ContainerAdministrator、ContainerUser、NT AUTHORITY\NETWORK SERVICE、NT AUTHORITY\LOCAL SERVICE。

关于这些限制的更多信息，可以查看[这里](#)和[这里](#)。

接下来

- [Kubernetes 中调度 Windows 容器的指南](#)
- [使用组托管服务帐户 \(GMSA\) 管理工作负载身份](#)
- [Windows 下 pod 和容器的 GMSA 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 December 12, 2020 at 11:45 PM PST: [Update configure-runasusername.md \(5526f4abd\)](#)

为容器和 Pods 分配 CPU 资源

本页面展示如何为容器设置 CPU *request* (请求) 和 CPU *limit* (限制)。容器使用的 CPU 不能超过所配置的限制。如果系统有空闲的 CPU 时间，则可以保证给容器分配其所请求数量的 CPU 资源。

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.

集群中的每个节点必须至少有 1 个 CPU 可用才能运行本任务中的示例。

本页的一些步骤要求你在集群中运行 [metrics-server](#) 服务。如果你的集群中已经有正在运行的 metrics-server 服务，可以跳过这些步骤。

如果你正在运行 [Minikube](#)，请运行以下命令启用 metrics-server：

```
minikube addons enable metrics-server
```

查看 metrics-server（或者其他资源度量 API metrics.k8s.io 服务提供者）是否正在运行，请键入以下命令：

```
kubectl get apiservices
```

如果资源指标 API 可用，则会输出将包含一个对 metrics.k8s.io 的引用。

```
NAME
v1beta1.metrics.k8s.io
```

创建一个名字空间

创建一个[名字空间](#)，以便将 本练习中创建的资源与集群的其余部分资源隔离。

```
kubectl create namespace cpu-example
```

指定 CPU 请求和 CPU 限制

要为容器指定 CPU 请求，请在容器资源清单中包含 resources: requests 字段。要指定 CPU 限制，请包含 resources:limits。

在本练习中，你将创建一个具有一个容器的 Pod。容器将会请求 0.5 个 CPU，而且最多限制使用 1 个 CPU。这是 Pod 的配置文件：

[pods/resource/cpu-request-limit.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
```

```
namespace: cpu-example
spec:
  containers:
    - name: cpu-demo-ctr
      image: vish/stress
      resources:
        limits:
          cpu: "1"
        requests:
          cpu: "0.5"
      args:
        - --cpus
        - "2"
```

配置文件的 args 部分提供了容器启动时的参数。 --cpus "2" 参数告诉容器尝试使用 2 个 CPU。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-limit.yaml --namespace=cpu-example
```

验证所创建的 Pod 处于 Running 状态

```
kubectl get pod cpu-demo --namespace=cpu-example
```

查看显示关于 Pod 的详细信息：

```
kubectl get pod cpu-demo --output=yaml --namespace=cpu-example
```

输出显示 Pod 中的一个容器的 CPU 请求为 500 milli CPU，并且 CPU 限制为 1 个 CPU。

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 500m
```

使用 kubectl top 命令来获取该 Pod 的度量值数据：

```
kubectl top pod cpu-demo --namespace=cpu-example
```

此示例输出显示 Pod 使用的是 974 milliCPU，即仅略低于 Pod 配置中指定的 1 个 CPU 的限制。

NAME	CPU(cores)	MEMORY(bytes)
cpu-demo	974m	<something>

回想一下，通过设置 `-cpu "2"`，你将容器配置为尝试使用 2 个 CPU，但是容器只被允许使用大约 1 个 CPU。容器的 CPU 用量受到限制，因为该容器正尝试使用超出其限制的 CPU 资源。

说明：CPU 使用率低于 1.0 的另一种可能的解释是，节点可能没有足够的 CPU 资源可用。回想一下，此练习的先决条件需要你的节点至少具有 1 个 CPU 可用。如果你的容器在只有 1 个 CPU 的节点上运行，则容器无论为容器指定的 CPU 限制如何，都不能使用超过 1 个 CPU。

CPU 单位

CPU 资源以 *CPU* 单位度量。Kubernetes 中的一个 CPU 等同于：

- 1 个 AWS vCPU
- 1 个 GCP核心
- 1 个 Azure vCore
- 裸机上具有超线程能力的英特尔处理器上的 1 个超线程

小数值是可以使用的。一个请求 0.5 CPU 的容器保证会获得请求 1 个 CPU 的容器的 CPU 的一半。你可以使用后缀 `m` 表示毫。例如 100m CPU、100 milliCPU 和 0.1 CPU 都相同。精度不能超过 1m。

CPU 请求只能使用绝对数量，而不是相对数量。0.1 在单核、双核或 48 核计算机上的 CPU 数量值是一样的。

删除 Pod：

```
kubectl delete pod cpu-demo --namespace(cpu-example)
```

设置超过节点能力的 CPU 请求

CPU 请求和限制与都与容器相关，但是我们可以考虑一下 Pod 具有对应的 CPU 请求和限制这样的场景。Pod 对 CPU 用量的请求等于 Pod 中所有容器的请求数量之和。同样，Pod 的 CPU 资源限制等于 Pod 中所有容器 CPU 资源限制数之和。

Pod 调度是基于资源请求值来进行的。仅在某节点具有足够的 CPU 资源来满足 Pod CPU 请求时，Pod 将会在对应节点上运行：

在本练习中，你将创建一个 Pod，该 Pod 的 CPU 请求对于集群中任何节点的容量而言都会过大。下面是 Pod 的配置文件，其中有一个容器。容器请求 100 个 CPU，这可能会超出集群中任何节点的容量。

[pods/resource/cpu-request-limit-2.yaml](#)
□

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
```

```
namespace: cpu-example
spec:
  containers:
    - name: cpu-demo-ctr-2
      image: vish/stress
      resources:
        limits:
          cpu: "100"
        requests:
          cpu: "100"
      args:
        - --cpus
        - "2"
```

创建 Pod :

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-limit-2.yaml
--namespace=cpu-example
```

查看该 Pod 的状态 :

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

输出显示 Pod 状态为 Pending。也就是说，Pod 未被调度到任何节点上运行，并且 Pod 将无限期地处于 Pending 状态：

NAME	READY	STATUS	RESTARTS	AGE
cpu-demo-2	0/1	Pending	0	7m

查看有关 Pod 的详细信息，包含事件：

```
kubectl describe pod cpu-demo-2 --namespace=cpu-example
```

输出显示由于节点上的 CPU 资源不足，无法调度容器：

Events:	
Reason	Message
-----	-----
FailedScheduling	No nodes are available that match all of the following predicates:: Insufficient cpu (3).

删除你的 Pod :

```
kubectl delete pod cpu-demo-2 --namespace=cpu-example
```

如果不指定 CPU 限制

如果你没有为容器指定 CPU 限制，则会发生以下情况之一：

- 容器在可以使用的 CPU 资源上没有上限。因而可以使用所在节点上所有的可用 CPU 资源。
- 容器在具有默认 CPU 限制的名字空间中运行，系统会自动为容器设置默认限制。集群管理员可以使用 [LimitRange](#) 指定 CPU 限制的默认值。

如果你设置了 CPU 限制但未设置 CPU 请求

如果你为容器指定了 CPU 限制值但未为其设置 CPU 请求，Kubernetes 会自动为其设置与 CPU 限制相同的 CPU 请求值。类似的，如果容器设置了内存限制值但未设置内存请求值，Kubernetes 也会为其设置与内存限制值相同的内存请求。

CPU 请求和限制的初衷

通过配置你的集群中运行的容器的 CPU 请求和限制，你可以有效利用集群上可用的 CPU 资源。通过将 Pod CPU 请求保持在较低水平，可以使 Pod 更有机会被调度。通过使 CPU 限制大于 CPU 请求，你可以完成两件事：

- Pod 可能会有突发性的活动，它可以利用碰巧可用的 CPU 资源。
- Pod 在突发负载期间可以使用的 CPU 资源数量仍被限制为合理的数量。

清理

删除名称空间：

```
kubectl delete namespace cpu-example
```

接下来

针对应用开发者

- [将内存资源分配给容器和 Pod](#)
- [配置 Pod 服务质量](#)

针对集群管理员

- [配置名称空间的默认内存请求和限制](#)
- [为名字空间配置默认 CPU 请求和限制](#)
- [为名字空间配置最小和最大内存限制](#)
- [为名字空间配置最小和最大 CPU 约束](#)
- [为名字空间配置内存和 CPU 配额](#)
- [为名字空间配置 Pod 配额](#)
- [配置 API 对象的配额](#)

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 12, 2020 at 9:33 PM PST: [Update assign-cpu-resource.md \(af59908f3\)](#)

配置 Pod 的服务质量

本页介绍怎样配置 Pod 让其获得特定的服务质量 (QoS) 类。Kubernetes 使用 QoS 类来决定 Pod 的调度和驱逐策略。

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.

QoS 类

Kubernetes 创建 Pod 时就给它指定了下列一种 QoS 类：

- Guaranteed
- Burstable
- BestEffort

创建命名空间

创建一个命名空间，以便将本练习所创建的资源与集群的其余资源相隔离。

```
kubectl create namespace qos-example
```

创建一个 QoS 类为 Guaranteed 的 Pod

对于 QoS 类为 Guaranteed 的 Pod :

- Pod 中的每个容器，包含初始化容器，必须指定内存请求和内存限制，并且两者要相等。
- Pod 中的每个容器，包含初始化容器，必须指定 CPU 请求和 CPU 限制，并且两者要相等。

下面是包含一个容器的 Pod 配置文件。容器设置了内存请求和内存限制，值都是 200 MiB。容器设置了 CPU 请求和 CPU 限制，值都是 700 milliCPU：

[pods/qos/qos-pod.yaml](#)



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

创建 Pod :

```
kubectl create -f https://k8s.io/examples/pods/qos/qos-pod.yaml --
namespace=qos-example
```

查看 Pod 详情：

```
kubectl get pod qos-demo --namespace=qos-example --output=yaml
```

结果表明 Kubernetes 为 Pod 配置的 QoS 类为 Guaranteed。结果也确认了 Pod 容器设置了与内存限制匹配的内存请求，设置了与 CPU 限制匹配的 CPU 请求。

```
spec:
  containers:
  ...
  resources:
```

```
limits:  
  cpu: 700m  
  memory: 200Mi  
requests:  
  cpu: 700m  
  memory: 200Mi  
...  
status:  
  qosClass: Guaranteed
```

说明：如果容器指定了自己的内存限制，但没有指定内存请求，Kubernetes 会自动为它指定与内存限制匹配的内存请求。同样，如果容器指定了自己的 CPU 限制，但没有指定 CPU 请求，Kubernetes 会自动为它指定与 CPU 限制匹配的 CPU 请求。

删除 Pod：

```
kubectl delete pod qos-demo --namespace=qos-example
```

创建一个 QoS 类为 **Burstable** 的 Pod

如果满足下面条件，将会指定 Pod 的 QoS 类为 **Burstable**：

- Pod 不符合 **Guaranteed** QoS 类的标准。
- Pod 中至少一个容器具有内存或 CPU 请求。

下面是包含一个容器的 Pod 配置文件。容器设置了内存限制 200 MiB 和内存请求 100 MiB。

[pods/qos/qos-pod-2.yaml](#)



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

创建 Pod：

```
kubectl create -f https://k8s.io/examples/pods/qos/qos-pod-2.yaml --namespace=qos-example
```

查看 Pod 详情：

```
kubectl get pod qos-demo-2 --namespace=qos-example --output=yaml
```

结果表明 Kubernetes 为 Pod 配置的 QoS 类为 Burstable。

```
spec:  
  containers:  
    - image: nginx  
      imagePullPolicy: Always  
      name: qos-demo-2-ctr  
    resources:  
      limits:  
        memory: 200Mi  
      requests:  
        memory: 100Mi  
...  
status:  
  qosClass: Burstable
```

删除 Pod：

```
kubectl delete pod qos-demo-2 --namespace=qos-example
```

创建一个 QoS 类为 BestEffort 的 Pod

对于 QoS 类为 BestEffort 的 Pod，Pod 中的容器必须没有设置内存和 CPU 限制或请求。

下面是包含一个容器的 Pod 配置文件。容器没有设置内存和 CPU 限制或请求。

[pods/qos/qos-pod-3.yaml](#)



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

创建 Pod：

```
kubectl create -f https://k8s.io/examples/pods/qos/qos-pod-3.yaml --namespace=qos-example
```

查看 Pod 详情：

```
kubectl get pod qos-demo-3 --namespace=qos-example --output=yaml
```

结果表明 Kubernetes 为 Pod 配置的 QoS 类为 BestEffort。

```
spec:  
  containers:  
    ...  
    resources: {}  
    ...  
status:  
  qosClass: BestEffort
```

删除 Pod：

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

创建包含两个容器的 Pod

下面是包含两个容器的 Pod 配置文件。一个容器指定了内存请求 200 MiB。另外一个容器没有指定任何请求和限制。

[pods/qos/qos-pod-4.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  name: qos-demo-4  
  namespace: qos-example  
spec:  
  containers:  
    - name: qos-demo-4-ctr-1  
      image: nginx  
      resources:  
        requests:  
          memory: "200Mi"  
    - name: qos-demo-4-ctr-2  
      image: redis
```

注意此 Pod 满足 Burstable QoS 类的标准。也就是说它不满足 Guaranteed QoS 类标准，因为它的一个容器设有内存请求。

创建 Pod :

```
kubectl create -f https://k8s.io/examples/pods/qos/qos-pod-4.yaml --namespace=qos-example
```

查看 Pod 详情：

```
kubectl get pod qos-demo-4 --namespace=qos-example --output=yaml
```

结果表明 Kubernetes 为 Pod 配置的 QoS 类为 Burstable：

```
spec:  
  containers:  
    ...  
    name: qos-demo-4-ctr-1  
    resources:  
      requests:  
        memory: 200Mi  
    ...  
    name: qos-demo-4-ctr-2  
    resources: {}  
    ...  
status:  
  qosClass: Burstable
```

删除 Pod :

```
kubectl delete pod qos-demo-4 --namespace=qos-example
```

环境清理

删除命名空间：

```
kubectl delete namespace qos-example
```

接下来

应用开发者参考

- [为 Pod 和容器分配内存资源](#)
- [为 Pod 和容器分配 CPU 资源](#)

集群管理员参考

- [为命名空间配置默认的内存请求和限制](#)
- [为命名空间配置默认的 CPU 请求和限制](#)

[为命名空间配置最小和最大内存限制](#)

- [为命名空间配置最小和最大 CPU 限制](#)
- [为命名空间配置内存和 CPU 配额](#)
- [为命名空间配置 Pod 配额](#)
- [为 API 对象配置配额](#)
- [控制节点上的拓扑管理策略](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 13, 2020 at 10:18 AM PST: [\[zh\] Sync changes on quality-service-pod from English site \(b2aa481d3\)](#)

为容器分派扩展资源

FEATURE STATE: Kubernetes v1.20 [stable]

本文介绍如何为容器指定扩展资源。

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.

在你开始此练习前，请先练习 [为节点广播扩展资源](#)。在那个练习中将配置你的一个节点来广播 dongle 资源。

给 Pod 分派扩展资源

要请求扩展资源，需要在你的容器清单中包括 resources:requests 字段。扩展资源可以使用任何完全限定名称，只是不能使用 *.kubernetes.io/。有效的扩展资源名的格式为 e

xample.com/foo，其中 example.com 应被替换为 你的组织的域名，而 foo 则是描述性的资源名称。

下面是包含一个容器的 Pod 配置文件：

[pods/resource/extended-resource-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo
spec:
  containers:
    - name: extended-resource-demo-ctr
      image: nginx
      resources:
        requests:
          example.com/dongle: 3
        limits:
          example.com/dongle: 3
```

在配置文件中，你可以看到容器请求了 3 个 dongles。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/extended-resource-pod.yaml
```

检查 Pod 是否运行正常：

```
kubectl get pod extended-resource-demo
```

描述 Pod：

```
kubectl describe pod extended-resource-demo
```

输出结果显示 dongle 请求如下：

```
Limits:
  example.com/dongle: 3
Requests:
  example.com/dongle: 3
```

尝试创建第二个 Pod

下面是包含一个容器的 Pod 配置文件，容器请求了 2 个 dongles。

[pods/resource/extended-resource-pod-2.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo-2
spec:
  containers:
  - name: extended-resource-demo-2-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 2
      limits:
        example.com/dongle: 2
```

Kubernetes 将不能满足 2 个 dongles 的请求，因为第一个 Pod 已经使用了 4 个可用 dongles 中的 3 个。

尝试创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/resource/extended-resource-
pod-2.yaml
```

描述 Pod：

```
kubectl describe pod extended-resource-demo-2
```

输出结果表明 Pod 不能被调度，因为没有一个节点上存在两个可用的 dongles。

Conditions:

Type	Status
------	--------

PodScheduled	False
--------------	-------

...

Events:

...

... Warning FailedScheduling pod (extended-resource-demo-2) failed to fit in
any node

fit failure summary on nodes : Insufficient example.com/dongle (1)

查看 Pod 的状态：

```
kubectl get pod extended-resource-demo-2
```

输出结果表明 Pod 虽然被创建了，但没有被调度到节点上正常运行。Pod 的状态为 Pending：

NAME	READY	STATUS	RESTARTS	AGE
extended-resource-demo-2	0/1	Pending	0	6m

清理

删除本练习中创建的 Pod：

```
kubectl delete pod extended-resource-demo  
kubectl delete pod extended-resource-demo-2
```

接下来

应用开发者参考

- [为容器和 Pod 分配内存资源](#)
- [为容器和 Pod 分配 CPU 资源](#)

集群管理员参考

- [为节点广播扩展资源](#)

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 August 13, 2020 at 5:49 PM PST: [\[zh\] Tidy up and fix links in tasks section \(5/10\) \(68abcb963\)](#)

配置 Pod 以使用卷进行存储

此页面展示了如何配置 Pod 以使用卷进行存储。

只要容器存在，容器的文件系统就会存在，因此当一个容器终止并重新启动，对该容器的文件系统改动将丢失。对于独立于容器的持久化存储，你可以使用[卷](#)。这对于有状态应用程序尤为重要，例如键值存储（如 Redis）和数据库。

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.

为 Pod 配置卷

在本练习中，你将创建一个运行 Pod，该 Pod 仅运行一个容器并拥有一个类型为 `emptyDir` 的卷，在整个 Pod 生命周期中一直存在，即使 Pod 中的容器被终止和重启。以下是 Pod 的配置：

[pods/storage/redis.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
```

1. 创建 Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/redis.yaml
```

1. 验证 Pod 中的容器是否正在运行，然后留意 Pod 的更改：

```
kubectl get pod redis --watch
```

输出如下：

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s

1. 在另一个终端，用 shell 连接正在运行的容器：

```
kubectl exec -it redis -- /bin/bash
```

1. 在你的 Shell 中，切换到 /data/redis 目录下，然后创建一个文件：

```
root@redis:/data# cd /data/redis/  
root@redis:/data/redis# echo Hello > test-file
```

1. 在你的 Shell 中，列出正在运行的进程：

```
root@redis:/data/redis# apt-get update  
root@redis:/data/redis# apt-get install procps  
root@redis:/data/redis# ps aux
```

输出类似于：

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME
COMMAND									
redis	1	0.1	0.1	33308	3828	?	Ssl	00:46	0:00 redis-server *:6379
root	12	0.0	0.0	20228	3020	?	Ss	00:47	0:00 /bin/bash
root	15	0.0	0.0	17500	2072	?	R+	00:48	0:00 ps aux

1. 在你的 Shell 中，结束 Redis 进程：

```
root@redis:/data/redis# kill <pid>
```

其中 <pid> 是 Redis 进程的 ID (PID)。

1. 在你原先终端中，留意 Redis Pod 的更改。最终你将会看到和下面类似的输出：

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s
redis	0/1	Completed	0	6m
redis	1/1	Running	1	6m

此时，容器已经终止并重新启动。这是因为 Redis Pod 的 [restartPolicy](#) 为 Always。

1. 用 Shell 进入重新启动的容器中：

```
kubectl exec -it redis -- /bin/bash
```

1. 在你的 Shell 中，进入到 /data/redis 目录下，并确认 test-file 文件是否仍然存在。

```
root@redis:/data/redis# cd /data/redis/  
root@redis:/data/redis# ls  
test-file
```

1. 删除为此练习所创建的 Pod：

```
kubectl delete pod redis
```

接下来

- 参阅 [Volume](#)。
- 参阅 [Pod](#)。
- 除了 emptyDir 提供的本地磁盘存储外，Kubernetes 还支持许多不同的网络附加存储解决方案，包括 GCE 上的 PD 和 EC2 上的 EBS，它们是关键数据的首选，并将处理节点上的一些细节，例如安装和卸载设备。了解更多详情请参阅[卷](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 12, 2020 at 11:25 AM PST: [\[zh\] Tidy up and fix links in tasks section \(4/10\) \(a3a745838\)](#)

配置 Pod 以使用 PersistentVolume 作为存储

本文介绍如何配置 Pod 使用 [PersistentVolumeClaim](#) 作为存储。以下是该过程的总结：

1. 你作为集群管理员创建由物理存储支持的 PersistentVolume。你不会将卷与任何 Pod 关联。
2. 你现在以开发人员或者集群用户的角色创建一个 PersistentVolumeClaim，它将自动绑定到合适的 PersistentVolume。
3. 你创建一个使用 PersistentVolumeClaim 作为存储的 Pod。

准备开始

- 你需要一个包含单个节点的 Kubernetes 集群，并且必须配置 kubectl 命令行工具以便与集群交互。如果还没有单节点集群，可以使用 [Minikube](#) 创建一个。 .
- 熟悉[持久卷](#)中的材料。

在你的节点上创建一个 index.html 文件

打开集群中节点的一个 Shell。如何打开 Shell 取决于集群的设置。例如，如果你正在使用 Minikube，那么可以通过输入 minikube ssh 来打开节点的 Shell。

在 Shell 中，创建一个 /mnt/data 目录：

```
# 这里假定你的节点使用 "sudo" 来以超级用户角色执行命令  
sudo mkdir /mnt/data
```

在 /mnt/data 目录中创建一个 index.html 文件：

```
# 这里再次假定你的节点使用 "sudo" 来以超级用户角色执行命令  
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.html"
```

说明：如果你的节点使用某工具而不是 sudo 来完成超级用户访问，你可以将上述命令 中的 sudo 替换为该工具的名称。

测试 index.html 文件确实存在：

```
cat /mnt/data/index.html
```

输出应该是：

```
Hello from Kubernetes storage
```

现在你可以关闭节点的 Shell 了。

创建 PersistentVolume

在本练习中，你将创建一个 *hostPath* 类型的 PersistentVolume。Kubernetes 支持用于在单节点集群上开发和测试的 hostPath 类型的 PersistentVolume。hostPath 类型的 PersistentVolume 使用节点上的文件或目录来模拟网络附加存储。

在生产集群中，你不会使用 hostPath。集群管理员会提供网络存储资源，比如 Google Compute Engine 持久盘卷、NFS 共享卷或 Amazon Elastic Block Store 卷。集群管理员还可以使用 [StorageClasses](#) 来设置[动态提供存储](#)。

下面是 hostPath PersistentVolume 的配置文件：

[pods/storage/pv-volume.yaml](#)



```
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: task-pv-volume  
  labels:  
    type: local  
spec:  
  storageClassName: manual  
  capacity:  
    storage: 10Gi  
  accessModes:  
    - ReadWriteOnce
```

```
hostPath:  
  path: "/mnt/data"
```

创建 PersistentVolume :

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

查看 PersistentVolume 的信息 :

```
kubectl get pv task-pv-volume
```

输出结果显示该 PersistentVolume 的状态 (STATUS) 为 Available。 这意味着它还没有被绑定给 PersistentVolumeClaim。

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS
CLAIM	STORAGECLASS	REASON	AGE	
task-pv-volume	10Gi	RWO	Retain	Available
manual		4s		

创建 PersistentVolumeClaim

下一步是创建一个 PersistentVolumeClaim。 Pod 使用 PersistentVolumeClaim 来请求物理存储。 在本练习中，你将创建一个 PersistentVolumeClaim，它请求至少 3 GB 容量的卷，该卷至少可以为一个节点提供读写访问。

下面是 PersistentVolumeClaim 的配置文件：

[pods/storage/pv-claim.yaml](#)



```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: task-pv-claim  
spec:  
  storageClassName: manual  
  accessModes:  
    - ReadWriteOnce  
resources:  
  requests:  
    storage: 3Gi
```

创建 PersistentVolumeClaim :

```
kubectl create -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

创建 PersistentVolumeClaim 之后，Kubernetes 控制平面将查找满足申领要求的 PersistentVolume。 如果控制平面找到具有相同 StorageClass 的适当的

PersistentVolume，则将 PersistentVolumeClaim 绑定到该 PersistentVolume 上。

再次查看 PersistentVolume 信息：

```
kubectl get pv task-pv-volume
```

现在输出的 STATUS 为 Bound。

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	STATUS	
CLAIM	STORAGECLASS	REASON	AGE		
task-pv-volume	10Gi	RWO	Retain	Bound	default/task-pv-claim manual 2m

查看 PersistentVolumeClaim：

```
kubectl get pvc task-pv-claim
```

输出结果表明该 PersistentVolumeClaim 绑定了你的 PersistentVolume task-pv-volume。

NAME	STATUS	VOLUME	CAPACITY	ACCESSMODES	
STORAGECLASS	AGE				
task-pv-claim	Bound	task-pv-volume	10Gi	RWO	manual 30s

创建 Pod

下一步是创建一个 Pod，该 Pod 使用你的 PersistentVolumeClaim 作为存储卷。

下面是 Pod 的 配置文件：

[pods/storage/pv-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
```

```
  name: "http-server"
  volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: task-pv-storage
```

注意 Pod 的配置文件指定了 PersistentVolumeClaim，但没有指定 PersistentVolume。对 Pod 而言，PersistentVolumeClaim 就是一个存储卷。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

检查 Pod 中的容器是否运行正常：

```
kubectl get pod task-pv-pod
```

打开一个 Shell 访问 Pod 中的容器：

```
kubectl exec -it task-pv-pod -- /bin/bash
```

在 Shell 中，验证 nginx 是否正在从 hostPath 卷提供 index.html 文件：

```
# 一定要在上一步 "kubectl exec" 所返回的 Shell 中执行下面三个命令
root@task-pv-pod:/# apt-get update
root@task-pv-pod:/# apt-get install curl
root@task-pv-pod:/# curl localhost
```

输出结果是你之前写到 hostPath 卷中的 index.html 文件中的内容：

```
Hello from Kubernetes storage
```

如果你看到此消息，则证明你已经成功地配置了 Pod 使用 PersistentVolumeClaim 的存储。

清理

删除 Pod、PersistentVolumeClaim 和 PersistentVolume 对象：

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

如果你还没有连接到集群中节点的 Shell，可以按之前所做操作，打开一个新的 Shell。

在节点的 Shell 上，删除你所创建的目录和文件：

```
# 这里假定你使用 "sudo" 来以超级用户的角色执行命令
sudo rm /mnt/data/index.html
sudo rmdir /mnt/data
```

你现在可以关闭连接到节点的 Shell。

访问控制

使用组 ID (GID) 配置的存储仅允许 Pod 使用相同的 GID 进行写入。 GID 不匹配或缺失将会导致无权访问错误。为了减少与用户的协调，管理员可以对 PersistentVolume 添加 GID 注解。这样 GID 就能自动添加到使用 PersistentVolume 的任何 Pod 中。

使用 `pv.beta.kubernetes.io/gid` 注解的方法如下所示：

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

当 Pod 使用带有 GID 注解的 PersistentVolume 时，注解的 GID 会被应用于 Pod 中的所有容器，应用的方法与 Pod 的安全上下文中指定的 GID 相同。每个 GID，无论是来自 PersistentVolume 注解还是来自 Pod 规约，都会被应用于每个容器中运行的第一个进程。

说明：当 Pod 使用 PersistentVolume 时，与 PersistentVolume 关联的 GID 不会在 Pod 资源本身的对象上出现。

接下来

- 进一步了解 [PersistentVolumes](#)
- 阅读[持久存储设计文档](#)

参考

- [PersistentVolume](#)
- [PersistentVolumeSpec](#)
- [PersistentVolumeClaim](#)
- [PersistentVolumeClaimSpec](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 13, 2020 at 12:05 AM PST: [Update configure-persistent-volume-storage.md \(4675237b5\)](#)

配置 Pod 使用投射卷作存储

本文介绍怎样通过[projected](#) 卷将现有的多个卷资源挂载到相同的目录。当前，secret、configMap、downwardAPI 和 serviceAccountToken 卷可以被投射。

说明： serviceAccountToken 不是一种卷类型

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.

为 Pod 配置 projected 卷

本练习中，您将从本地文件来创建包含有用户名和密码的 Secret。然后创建运行一个容器的 Pod，该 Pod 使用[projected](#) 卷将 Secret 挂载到相同的路径下。

下面是 Pod 的配置文件：

[pods/storage/projected.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
    - name: test-projected-volume
      image: busybox
      args:
        - sleep
        - "86400"
  volumeMounts:
```

```
- name: all-in-one
  mountPath: "/projected-volume"
  readOnly: true
volumes:
- name: all-in-one
  projected:
    sources:
      - secret:
          name: user
      - secret:
          name: pass
```

1. 创建 Secret:

```
# 创建包含用户名和密码的文件:
echo -n "admin" > ./username.txt
echo -n "1f2d1e2e67df" > ./password.txt-->

# 将上述文件引用到 Secret :
kubectl create secret generic user --from-file=./username.txt
kubectl create secret generic pass --from-file=./password.txt
```

2. 创建 Pod :

```
kubectl create -f https://k8s.io/examples/pods/storage/projected.yaml
```

3. 确认 Pod 中的容器运行正常，然后监视 Pod 的变化：

```
kubectl get --watch pod test-projected-volume
```

输出结果和下面类似：

NAME	READY	STATUS	RESTARTS	AGE
test-projected-volume	1/1	Running	0	14s

4. 在另外一个终端中，打开容器的 shell：

```
kubectl exec -it test-projected-volume -- /bin/sh
```

5. 在 shell 中，确认 projected-volume 目录包含你的投射源：

```
ls /projected-volume/
```

清理

删除 Pod 和 Secret:

```
kubectl delete pod test-projected-volume
kubectl delete secret user pass
```

接下来

- 进一步了解[projected 卷](#)。
- 阅读[一体卷设计文档](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 13, 2020 at 12:20 AM PST: [Update configure-projected-volume-storage.md \(7eda46408\)](#)

为 Pod 或容器配置安全性上下文

安全上下文 (Security Context) 定义 Pod 或 Container 的特权与访问控制设置。 安全上下文包括但不限于：

- 自主访问控制 (Discretionary Access Control) : 基于 [用户 ID \(UID \) 和组 ID \(GID \)](#). 来判定对对象 (例如文件) 的访问权限。
- [安全性增强的 Linux \(SELinux \)](#) : 为对象赋予安全性标签。
- 以特权模式或者非特权模式运行。
- [Linux 权能](#): 为进程赋予 root 用户的部分特权而非全部特权。
- [AppArmor](#) : 使用程序框架来限制个别程序的权能。
- [Seccomp](#) : 过滤进程的系统调用。
- AllowPrivilegeEscalation : 控制进程是否可以获得超出其父进程的特权。 此布尔值直接控制是否为容器进程设置 [no_new_privs](#) 标志。 当容器以特权模式运行或者具有 CAP_SYS_ADMIN 权能时，AllowPrivilegeEscalation 总是为 true。
- readOnlyRootFilesystem : 以只读方式加载容器的根文件系统。

以上条目不是安全上下文设置的完整列表 -- 请参阅 [SecurityContext](#) 了解其完整列表。

关于在 Linux 系统中的安全机制的更多信息，可参阅 [Linux 内核安全性能概述](#)。

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`.

为 Pod 设置安全性上下文

要为 Pod 设置安全性设置，可在 Pod 规约中包含 `securityContext` 字段。`securityContext` 字段值是一个 [PodSecurityContext](#) 对象。你为 Pod 所设置的安全性配置会应用到 Pod 中所有 Container 上。下面是一个 Pod 的配置文件，该 Pod 定义了 `securityContext` 和一个 `emptyDir` 卷：

[pods/security/security-context.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
  securityContext:
    allowPrivilegeEscalation: false
```

在配置文件中，`runAsUser` 字段指定 Pod 中的所有容器内的进程都使用用户 ID 1000 来运行。`runAsGroup` 字段指定所有容器中的进程都以主组 ID 3000 来运行。如果忽略此字段，则容器的主组 ID 将是 root (0)。当 `runAsGroup` 被设置时，所有创建的文件也会划归用户 1000 和组 3000。由于 `fsGroup` 被设置，容器中所有进程也会是附组 ID 2000 的一部分。卷 `/data/demo` 及在该卷中创建的任何文件的属主都会是组 ID 2000。

创建该 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context.yaml
```

检查 Pod 的容器处于运行状态：

```
kubectl get pod security-context-demo
```

开启一个 Shell 进入到运行中的容器：

```
kubectl exec -it security-context-demo -- sh
```

在你的 Shell 中，列举运行中的进程：

```
ps
```

输出显示进程以用户 1000 运行，即 runAsUser 所设置的值：

PID	USER	TIME	COMMAND
1	1000	0:00	sleep 1h
6	1000	0:00	sh
...			

在你的 Shell 中，进入 /data 目录列举其内容：

```
cd /data  
ls -l
```

输出显示 /data/demo 目录的组 ID 为 2000，即 fsGroup 的设置值：

```
drwxrwsrwx 2 root 2000 4096 Jun 6 20:08 demo
```

在你的 Shell 中，进入到 /data/demo 目录下创建一个文件：

```
cd demo  
echo hello > testfile
```

列举 /data/demo 目录下的文件：

```
ls -l
```

输出显示 testfile 的组 ID 为 2000，也就是 fsGroup 所设置的值：

```
-rw-r--r-- 1 1000 2000 6 Jun 6 20:08 testfile
```

运行下面的命令：

```
id
```

输出为：

```
uid=1000 gid=3000 groups=2000
```

你会看到 gid 值为 3000，也就是 runAsGroup 字段的值。如果 runAsGroup 被忽略，则 gid 会取值 0 (root)，而进程就能够与 root 用户组所拥有以及要求 root 用户组访问权限的文件交互。

退出你的 Shell :

```
exit
```

为 Pod 配置卷访问权限和属主变更策略

FEATURE STATE: Kubernetes v1.20 [beta]

默认情况下，Kubernetes 在挂载一个卷时，会递归地更改每个卷中的内容的属主和访问权限，使之与 Pod 的 securityContext 中指定的 fsGroup 匹配。对于较大的数据卷，检查和变更属主与访问权限可能会花费很长时间，降低 Pod 启动速度。你可以在 securityContext 中使用 fsGroupChangePolicy 字段来控制 Kubernetes 检查和管理卷属主和访问权限的方式。

fsGroupChangePolicy - fsGroupChangePolicy 定义在卷被暴露给 Pod 内部之前对其内容的属主和访问许可进行变更的行为。此字段仅适用于那些支持使用 fsGroup 来控制属主与访问权限的卷类型。此字段的取值可以是：

- OnRootMismatch：只有根目录的属主与访问权限与卷所期望的权限不一致时，才改变其中内容的属主和访问权限。这一设置有助于缩短更改卷的属主与访问权限所需要的时间。
- Always：在挂载卷时总是更改卷中内容的属主和访问权限。

例如：

```
securityContext:  
  runAsUser: 1000  
  runAsGroup: 3000  
  fsGroup: 2000  
  fsGroupChangePolicy: "OnRootMismatch"
```

这是一个 Alpha 阶段的功能特性。要使用此特性，需要在 kube-apiserver、kube-controller-manager 和 kubelet 上启用 ConfigurableFSGroupPolicy [特性门控](#)。

说明：此字段对于[secret](#)、[configMap](#) 和 [emptydir](#) 这类临时性存储无效。

为 Container 设置安全性上下文

若要为 Container 设置安全性配置，可以在 Container 清单中包含 securityContext 字段。securityContext 字段的取值是一个 [SecurityContext](#) 对象。你为 Container 设置的安全性配置仅适用于该容器本身，并且所指定的设置 在与 Pod 层面设置的内容发生重叠时，会重载后者。Container 层面的设置不会影响 到 Pod 的卷。

下面是一个 Pod 的配置文件，其中包含一个 Container。Pod 和 Container 都有 securityContext 字段：

[pods/security/security-context-2.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
    - name: sec-ctx-demo-2
      image: gcr.io/google-samples/node-hello:1.0
      securityContext:
        runAsUser: 2000
      allowPrivilegeEscalation: false
```

创建该 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-2.yaml
```

验证 Pod 中的容器处于运行状态：

```
kubectl get pod security-context-demo-2
```

启动一个 Shell 进入到运行中的容器内：

```
kubectl exec -it security-context-demo-2 -- sh
```

在你的 Shell 中，列举运行中的进程：

```
ps aux
```

输出显示进程以用户 2000 账号运行。该值是在 Container 的 runAsUser 中设置的。该设置值重载了 Pod 层面所设置的值 1000。

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
2000	1	0.0	0.0	4336	764	?	Ss	20:36	0:00	/bin/sh -c node server.js
2000	8	0.1	0.5	772124	22604	?	SI	20:36	0:00	node server.js
...										

退出你的 Shell：

```
exit
```

为 Container 设置权能

使用 [Linux 权能](#)，你可以 赋予进程 root 用户所拥有的某些特权，但不必赋予其全部特权。要为 Container 添加或移除 Linux 权能，可以在 Container 清单的 securityContext 节包含 capabilities 字段。

首先，查看不包含 capabilities 字段时候会发什么。下面是一个配置文件，其中没有添加或移除容器的权能：

[pods/security/security-context-3.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
    - name: sec-ctx-3
      image: gcr.io/google-samples/node-hello:1.0
```

创建该 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-3.yaml
```

验证 Pod 的容器处于运行状态：

```
kubectl get pod security-context-demo-3
```

启动一个 Shell 进入到运行中的容器：

```
kubectl exec -it security-context-demo-3 -- sh
```

在你的 Shell 中，列举运行中的进程：

```
ps aux
```

输出显示容器中进程 ID (PIDs)：

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.0 0.0 4336 796 ? Ss 18:17 0:00 /bin/sh -c node server.js
root 5 0.1 0.5 772124 22700 ? Sl 18:17 0:00 node server.js
```

在你的 Shell 中，查看进程 1 的状态：

```
cd /proc/1
cat status
```

输出显示进程的权能位图：

```
...
CapPrm: 00000000a80425fb
```

```
CapEff: 00000000a80425fb
```

```
...
```

记下进程权能位图，之后退出你的 Shell：

```
exit
```

接下来运行一个与前例中容器相同的容器，只是这个容器有一些额外的权能设置。

下面是一个 Pod 的配置，其中运行一个容器。配置为容器添加 CAP_NET_ADMIN 和 CAP_SYS_TIME 权能：

[pods/security/security-context-4.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/security/security-context-4.yaml
```

启动一个 Shell，进入到运行中的容器：

```
kubectl exec -it security-context-demo-4 -- sh
```

在你的 Shell 中，查看进程 1 的权能：

```
cd /proc/1
cat status
```

输出显示的是进程的权能位图：

```
...
CapPrm: 00000000aa0435fb
CapEff: 00000000aa0435fb
...
```

比较两个容器的权能位图：

```
00000000a80425fb  
00000000aa0435fb
```

在第一个容器的权能位图中，位 12 和 25 是没有设置的。在第二个容器中，位 12 和 25 是设置了的。位 12 是 CAP_NET_ADMIN 而位 25 则是 CAP_SYS_TIME。参见 [capability.h](#) 了解权能常数的定义。

说明：Linux 权能常数定义的形式为 CAP_XXX。但是你在 Container 清单中列举权能时，要将权能名称中的 CAP_ 部分去掉。例如，要添加 CAP_SYS_TIME，可在权能列表中添加 SYS_TIME。

为容器设置 Seccomp 样板

若要为容器设置 Seccomp 样板（Profile），可在你的 Pod 或 Container 清单的 securityContext 节中包含 seccompProfile 字段。该字段是一个 [SeccompProfile](#) 对象，包含 type 和 localhostProfile 属性。type 的合法选项包括 RuntimeDefault、Unconfined 和 Localhost。localhostProfile 只能在 type: Localhost 配置下才需要设置。该字段标明节点上预先配置的样板的路径，路径是相对于 kubelet 所配置的 Seccomp 样板路径（使用 --root-dir 配置）而言的。

下面是一个例子，设置容器使用节点上容器运行时的默认样板作为 Seccomp 样板：

```
...  
securityContext:  
  seccompProfile:  
    type: RuntimeDefault
```

下面是另一个例子，将 Seccomp 的样板设置为位于 <kubelet-根目录>/seccomp/my-profiles/profile-allow.json 的一个预先配置的文件。

```
...  
securityContext:  
  seccompProfile:  
    type: Localhost  
    localhostProfile: my-profiles/profile-allow.json
```

为 Container 赋予 SELinux 标签

若要给 Container 设置 SELinux 标签，可以在 Pod 或 Container 清单的 securityContext 节包含 seLinuxOptions 字段。seLinuxOptions 字段的取值是一个 [SELinuxOptions](#) 对象。下面是一个应用 SELinux 标签的例子：

```
...  
securityContext:  
  seLinuxOptions:  
    level: "s0:c123,c456"
```

说明：要指定 SELinux，需要在宿主操作系统中装载 SELinux 安全性模块。

讨论

Pod 的安全上下文适用于 Pod 中的容器，也适用于 Pod 所挂载的卷（如果有的话）。尤其是，`fsGroup` 和 `seLinuxOptions` 按下面的方式应用到挂载卷上：

- `fsGroup`：支持属主管理的卷会被修改，将其属主变更为 `fsGroup` 所指定的 GID，并且对该 GID 可写。进一步的细节可参阅 [属主变更设计文档](#)。
- `seLinuxOptions`：支持 SELinux 标签的卷会被重新打标签，以便可被 `seLinuxOptions` 下所设置的标签访问。通常你只需要设置 `level` 部分。该部分设置的是赋予 Pod 中所有容器及卷的 [多类别安全性 \(Multi-Category Security , MCS\)](#) 标签。

警告：在为 Pod 设置 MCS 标签之后，所有带有相同标签的 Pod 可以访问该卷。如果你需要跨 Pod 的保护，你必须为每个 Pod 赋予独特的 MCS 标签。

清理

删除之前创建的所有 Pod：

```
kubectl delete pod security-context-demo  
kubectl delete pod security-context-demo-2  
kubectl delete pod security-context-demo-3  
kubectl delete pod security-context-demo-4
```

接下来

- [PodSecurityContext API 定义](#)
- [SecurityContext API 定义](#)
- [使用最新的安全性增强来调优 Docker](#)
- [安全性上下文的设计文档](#)
- [属主管理的设计文档](#)
- [Pod 安全策略](#)
- [AllowPrivilegeEscalation 的设计文档](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 13, 2020 at 12:40 AM PST: [Update security-context.md \(605bae56c\)](#)

为 Pod 配置服务账户

服务账户为 Pod 中运行的进程提供了一个标识。

说明：本文是服务账户的用户使用介绍，描述服务账号在集群中如何起作用。你的集群管理员可能已经对你的集群做了定制，因此导致本文中所讲述的内容并不适用。

当你（自然人）访问集群时（例如，使用 kubectl），API 服务器将你的身份验证为特定的用户帐户（当前这通常是 admin，除非你的集群管理员已经定制了你的集群配置）。Pod 内的容器中的进程也可以与 api 服务器接触。当它们进行身份验证时，它们被验证为特定的服务帐户（例如，default）。

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.

使用默认的服务账户访问 API 服务器

当你创建 Pod 时，如果没有指定服务账户，Pod 会被指定给命名空间中的 default 服务账户。如果你查看 Pod 的原始 JSON 或 YAML（例如：kubectl get pods/podname -o yaml），你可以看到 spec.serviceAccountName 字段已经被自动设置了。

你可以使用自动挂载给 Pod 的服务账户凭据访问 API，[访问集群](#) 中有相关描述。服务账户的 API 许可取决于你所使用的 [鉴权插件和策略](#)。

在 1.6 以上版本中，你可以通过在服务账户上设置 automountServiceAccountToken: false 来实现不给服务账号自动挂载 API 凭据：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

在 1.6 以上版本中，你也可以选择不给特定 Pod 自动挂载 API 凭据：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
...
```

如果 Pod 和服务账户都指定了 automountServiceAccountToken 值，则 Pod 的 spec 优先于服务帐户。

使用多个服务账户

每个命名空间都有一个名为 default 的服务账户资源。你可以用下面的命令查询这个服务账户以及命名空间中的其他 ServiceAccount 资源：

```
kubectl get serviceAccounts
NAME      SECRETS   AGE
default   1          1d
```

你可以像这样来创建额外的 ServiceAccount 对象：

```
kubectl create -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
serviceaccount/build-robot created
```

如果你查询服务帐户对象的完整信息，如下所示：

```
kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

那么你就能看到系统已经自动创建了一个令牌并且被服务账户所引用。

你可以使用授权插件来 [设置服务账户的访问许可](#)。

要使用非默认的服务账户，只需简单的将 Pod 的 spec.serviceAccountName 字段设置为你想用的服务账户名称。

Pod 被创建时服务账户必须存在，否则会被拒绝。

你不能更新已经创建好的 Pod 的服务账户。

你可以清除服务账户，如下所示：

```
kubectl delete serviceaccount/build-robot
```

手动创建服务账户 API 令牌

假设我们有一个上面提到的名为 "build-robot" 的服务账户，然后我们手动创建一个新的 Secret。

```
kubectl create -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
secret/build-robot-secret created
```

现在，你可以确认新构建的 Secret 中填充了 "build-robot" 服务帐户的 API 令牌。

令牌控制器将清理不存在的服务帐户的所有令牌。

```
kubectl describe secrets/build-robot-secret
Name:      build-robot-secret
Namespace:  default
Labels:     <none>
Annotations: kubernetes.io/service-account.name=build-robot
             kubernetes.io/service-account.uid=da68f9c6-9d26-11e7-
             b84e-002dc52800da
Type:      kubernetes.io/service-account-token
```

Data

```
=====
ca.crt:      1338 bytes
namespace:   7 bytes
token:      ...
```

说明：这里省略了 token 的内容。

为服务账户添加 ImagePullSecrets

创建 ImagePullSecret

- 创建一个 ImagePullSecret，如同[为 Pod 设置 ImagePullSecret](#)所述。

```
kubectl create secret docker-registry myregistrykey --docker-server=DUMMY_SERVER \
--docker-username=DUMMY_USERNAME --docker-password=DUMMY_PASSWORD \
--docker-email=DUMMY_DOCKER_EMAIL
```

- 确认创建成功：

```
kubectl get secrets myregistrykey
```

输出类似于：

NAME	TYPE	DATA	AGE
myregistrykey	kubernetes.io/.dockerconfigjson	1	1d

将镜像拉取 Secret 添加到服务账号

接着修改命名空间的 default 服务帐户，以将该 Secret 用作 imagePullSecret。

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}
```

你也可以使用 kubectl edit，或者如下所示手动编辑 YAML 清单：

```
kubectl get serviceaccounts default -o yaml > ./sa.yaml
```

sa.yaml 文件的内容类似于：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
```

使用你常用的编辑器（例如 vi），打开 sa.yaml 文件，删除带有键名 resourceVersion 的行，添加带有 imagePullSecrets: 的行，最后保存文件。

所得到的 sa.yaml 文件类似于：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
```

最后，用新的更新的 sa.yaml 文件替换服务账号。

```
kubectl replace serviceaccount default -f ./sa.yaml
```

验证镜像拉取 Secret 已经被添加到 Pod 规约

现在，在当前命名空间中创建的每个使用默认服务账号的新 Pod，新 Pod 都会自动设置其 .spec.imagePullSecrets 字段：

```
kubectl run nginx --image=nginx --restart=Never
kubectl get pod nginx -o=jsonpath='{.spec.imagePullSecrets[0].name}{"\n"}'
```

输出为：

```
myregistrykey
```

服务帐户令牌卷投射

FEATURE STATE: Kubernetes v1.12 [beta]

说明：

ServiceAccountTokenVolumeProjection 在 1.12 版本中是 **beta** 阶段，可以通过向 API 服务器传递以下所有参数来启用它：

- --service-account-issuer
- --service-account-signing-key-file
- --service-account-api-audiences

kubelet 还可以将服务帐户令牌投影到 Pod 中。你可以指定令牌的所需属性，例如受众和有效持续时间。这些属性在默认服务帐户令牌上无法配置。当删除 Pod 或 ServiceAccount 时，服务帐户令牌也将对 API 无效。

使用名为 [ServiceAccountToken](#) 的 ProjectedVolume 类型在 PodSpec 上配置此功能。要向 Pod 提供具有 "vault" 用户以及两个小时有效期的令牌，可以在 PodSpec 中配置以下内容：



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
    volumeMounts:
      - mountPath: /var/run/secrets/tokens
        name: vault-token
  serviceAccountName: build-robot
  volumes:
    - name: vault-token
  projected:
    sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

创建 Pod :

```
kubectl create -f https://k8s.io/examples/pods/pod-projected-svc-token.yaml
```

kubelet 组件会替 Pod 请求令牌并将其保存起来，通过将令牌存储到一个可配置的 路径使之在 Pod 内可用，并在令牌快要到期的时候刷新它。 kubelet 会在令牌存在期达到其 TTL 的 80% 的时候或者令牌生命期超过 24 小时的时候主动轮换它。

应用程序负责在令牌被轮换时重新加载其内容。对于大多数使用场景而言，周期性地（例如，每隔 5 分钟）重新加载就足够了。

发现服务账号分发者

FEATURE STATE: Kubernetes v1.18 [alpha]

通过启用 ServiceAccountIssuerDiscovery 特性门控，并按[前文所述](#)启用服务账号令牌投射，可以启用发现服务账号分发者（Service Account Issuer Discovery）这一功能特性。

说明：

分发者的 URL 必须遵从 [OIDC 发现规范](#)。这意味着 URL 必须使用 https 模式，并且必须在 {service-account-issuer}/.well-known/openid-configuration 路径提供 OpenID 提供者（Provider）配置。

如果 URL 没有遵从这一规范，ServiceAccountIssuerDiscovery 末端就不会被注册，即使该特性已经被启用。

发现服务账号分发者这一功能使得用户能够用联邦的方式结合使用 Kubernetes 集群（Identity Provider，标识提供者）与外部系统（relying parties，依赖方）所分发的服务账号令牌。

当此功能被启用时，Kubernetes API 服务器会在 /.well-known/openid-configuration 提供一个 OpenID 提供者配置文档，并在 /openid/v1/jwks 处提供与之关联的 JSON Web Key Set (JWKS)。这里的 OpenID 提供者配置有时候也被称作发现文档（Discovery Document）。

特性被启用时，集群也会配置名为 system:service-account-issuer-discovery 的默认 RBAC ClusterRole，但默认情况下不提供角色绑定对象。举例而言，管理员可以根据其安全性需要以及期望集成的外部系统选择是否将该角色绑定到 system:authenticated 或 system:unauthenticated。

说明：对 /.well-known/openid-configuration 和 /openid/v1/jwks 路径请求的响应被设计为与 OIDC 兼容，但不是完全与其一致。返回的文档仅包含对 Kubernetes 服务账号令牌进行验证所必须的参数。

JWKS 响应包含依赖方可以用来验证 Kubernetes 服务账号令牌的公钥数据。依赖方先会查询 OpenID 提供者配置，之后使用返回响应中的 jwks_uri 来查找 JWKS。

在很多场合，Kubernetes API 服务器都不会暴露在公网上，不过对于缓存并向外提供 API 服务器响应数据的公开末端而言，用户或者服务提供商可以选择将其暴露在公网上。在这种环境中，可能会重载 OpenID 提供者配置中的 jwks_uri，使之指向公网上可用的末端地址，而不是 API 服务器的地址。这时需要向 API 服务器传递 --service-account-jwks-uri 参数。与分发者 URL 类似，此 JWKS URI 也需要使用 https 模式。

接下来

另请参见：

- [服务账号的集群管理员指南](#)
- [服务账号签署密钥检索 KEP](#)
- [OIDC 发现规范](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 07, 2020 at 1:59 PM PST: [Update configure-service-account.md \(8e8257ab0\)](#)

从私有仓库拉取镜像

本文介绍如何使用 Secret 从私有的 Docker 镜像仓库或代码仓库拉取镜像来创建 Pod。

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.

你需要 [Docker ID](#) 和密码来进行本练习。

登录 Docker 镜像仓库

在个人电脑上，要想拉取私有镜像必须在镜像仓库上进行身份验证。

```
docker login
```

当出现提示时，输入 Docker 用户名和密码。

登录过程会创建或更新保存有授权令牌的 config.json 文件。

查看 config.json 文件：

```
cat ~/.docker/config.json
```

输出结果包含类似于以下内容的部分：

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "auth": "c3R...zE2"
    }
  }
}
```

说明：如果使用 Docker 凭证仓库，则不会看到 auth 条目，看到的将是以仓库名称作为值的 credsStore 条目。

在集群中创建保存授权令牌的 Secret

Kubernetes 集群使用 docker-registry 类型的 Secret 来通过容器仓库的身份验证，进而提取私有映像。

创建 Secret，命名为 regcred：

```
kubectl create secret docker-registry regcred \
--docker-server=<你的镜像仓库服务器> \
--docker-username=<你的用户名> \
--docker-password=<你的密码> \
--docker-email=<你的邮箱地址>
```

在这里：

- <your-registry-server> 是你的私有 Docker 仓库全限定域名 (FQDN)。 (参考 <https://index.docker.io/v1/> 中关于 DockerHub 的部分)
- <your-name> 是你的 Docker 用户名。
- <your-pword> 是你的 Docker 密码。
- <your-email> 是你的 Docker 邮箱。

这样你就成功地将集群中的 Docker 凭据设置为名为 regcred 的 Secret。

检查 Secret regcred

要了解你创建的 regcred Secret 的内容，可以用 YAML 格式进行查看：

```
kubectl get secret regcred --output=yaml
```

输出和下面类似：

```
apiVersion: v1
data:
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUI6RTIifX0=
kind: Secret
metadata:
  ...
  name: regcred
  ...
type: kubernetes.io/dockerconfigjson
```

.dockerconfigjson 字段的值是 Docker 凭据的 base64 表示。

要了解 dockerconfigjson 字段中的内容，请将 Secret 数据转换为可读格式：

```
kubectl get secret regcred --output="jsonpath={.data.\.dockerconfigjson}" |  
base64 --decode
```

输出和下面类似：

```
{"auths":{"yourprivateregistry.com":{"username":"janedoe","password":"xxxxxxxxxx  
x","email":"jdoe@example.com","auth":"c3R...zE2"}}}
```

要了解 auth 字段中的内容，请将 base64 编码过的数据转换为可读格式：

```
echo "c3R...zE2" | base64 --decode
```

输出结果中，用户名和密码用 : 链接，类似下面这样：

```
janedoe:xxxxxxxxxxxx
```

注意，Secret 数据包含与本地 `~/.docker/config.json` 文件类似的授权令牌。

这样你就已经成功地将 Docker 凭据设置为集群中的名为 regcred 的 Secret。

创建一个使用你的 Secret 的 Pod

下面是一个 Pod 配置文件，它需要访问 regcred 中的 Docker 凭据：

[pods/private-reg-pod.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  name: private-reg  
spec:  
  containers:  
    - name: private-reg-container  
      image: <your-private-image>  
  imagePullSecrets:  
    - name: regcred
```

下载上述文件：

```
wget -O my-private-reg-pod.yaml https://k8s.io/examples/pods/private-reg-  
pod.yaml
```

在 `my-private-reg-pod.yaml` 文件中，使用私有仓库的镜像路径替换 `<your-private-image>`，例如：

```
janedoe/jdoe-private:v1
```

要从私有仓库拉取镜像，Kubernetes 需要凭证。 配置文件中的 `imagePullSecrets` 字段表明 Kubernetes 应该通过名为 `regcred` 的 Secret 获取凭证。

创建使用了你的 Secret 的 Pod，并检查它是否正常运行：

```
kubectl apply -f my-private-reg-pod.yaml  
kubectl get pod private-reg
```

接下来

- 进一步了解 [Secret](#)
- 进一步了解 [使用私有仓库](#)
- 参考 [kubectl create secret docker-registry](#)
- 参考 [Secret](#)
- 参考 [PodSpec](#) 中的 `imagePullSecrets` 字段

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 12, 2020 at 11:25 AM PST: [\[zh\] Tidy up and fix links in tasks section \(4/10\) \(a3a745838\)](#)

配置存活、就绪和启动探测器

这篇文章介绍如何给容器配置存活、就绪和启动探测器。

[kubelet](#) 使用存活探测器来知道什么时候要重启容器。例如，存活探测器可以捕捉到死锁（应用程序在运行，但是无法继续执行后面的步骤）。这样的情况下重启容器有助于让应用程序在有问题的情况下更可用。

[kubelet](#) 使用就绪探测器可以知道容器什么时候准备好了并可以开始接受请求流量，当一个 Pod 内的所有容器都准备好了，才能把这个 Pod 看作就绪了。这种信号的一个用途就是控制哪个 Pod 作为 Service 的后端。在 Pod 还没有准备好的时候，会从 Service 的负载均衡器中被剔除的。

[kubelet](#) 使用启动探测器可以知道应用程序容器什么时候启动了。如果配置了这类探测器，就可以控制容器在启动成功后再进行存活性和就绪检查，确保这些存活、就绪探测器不会影响应用程序的启动。这可以用于对慢启动容器进行存活性检测，避免它们在启动运行之前就被杀掉。

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.

定义存活命令

许多长时间运行的应用程序最终会过渡到断开的状态，除非重新启动，否则无法恢复。Kubernetes 提供了存活探测器来发现并补救这种情况。

在这篇练习中，你会创建一个 Pod，其中运行一个基于 k8s.gcr.io/busybox 镜像的容器。下面是这个 Pod 的配置文件。

[pods/probe/exec-liveness.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -C
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
  livenessProbe:
    exec:
      command:
      - cat
      - /tmp/healthy
    initialDelaySeconds: 5
    periodSeconds: 5
```

在这个配置文件中，可以看到 Pod 中只有一个容器。periodSeconds 字段指定了 kubelet 应该每 5 秒执行一次存活探测。initialDelaySeconds 字段告诉 kubelet 在执行第一次探测前应该等待 5 秒。kubelet 在容器内执行命令 cat /tmp/healthy 来进行探测。如果命令执行成功并且返回值为 0，kubelet 就会认为这个容器是健康存活的。如果这个命令返回非 0 值，kubelet 会杀死这个容器并重新启动它。

当容器启动时，执行如下的命令：

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

这个容器生命的前 30 秒，/tmp/healthy 文件是存在的。所以在这最开始的 30 秒内，执行命令 cat /tmp/healthy 会返回成功代码。30 秒之后，执行命令 cat /tmp/healthy 就会返回失败代码。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

在 30 秒内，查看 Pod 的事件：

```
kubectl describe pod liveness-exec
```

输出结果表明还没有存活探测器失败：

FirstSeen	LastSeen	Count	From	SubobjectPath	Type
Reason	Message				
24s	24s	1	{default-scheduler}		Normal Scheduled
					Successfully assigned liveness-exec to worker0
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Pulling pulling image "k8s.gcr.io/busybox"
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Pulled Successfully pulled image "k8s.gcr.io/busybox"
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Created Created container with docker id 86849c15382e; Security: [seccomp=unconfined]
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Started Started container with docker id 86849c15382e

35 秒之后，再来看 Pod 的事件：

```
kubectl describe pod liveness-exec
```

在输出结果的最下面，有信息显示存活探测器失败了，这个容器被杀死并且被重建了。

FirstSeen	LastSeen	Count	From	SubobjectPath	Type
Reason	Message				
37s	37s	1	{default-scheduler}		Normal Scheduled
					Successfully assigned liveness-exec to worker0
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Pulling pulling image "k8s.gcr.io/busybox"
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal
					Pulled Successfully pulled image "k8s.gcr.io/busybox"
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal

```
Created    Created container with docker id 86849c15382e; Security:  
[seccomp=unconfined]  
36s    36s    1 {kubelet worker0} spec.containers{liveness} Normal  
Started    Started container with docker id 86849c15382e  
2s    2s    1 {kubelet worker0} spec.containers{liveness} Warning  
Unhealthy  Liveness probe failed: cat: can't open '/tmp/healthy': No such file or  
directory
```

再等另外 30 秒，查看这个容器被重启了：

```
kubectl get pod liveness-exec
```

输出结果显示 RESTARTS 的值增加了 1。

NAME	READY	STATUS	RESTARTS	AGE
liveness-exec	1/1	Running	1	1m

定义一个存活态 HTTP 请求接口

另外一种类型的存活探测方式是使用 HTTP GET 请求。下面是一个 Pod 的配置文件，其中运行一个基于 k8s.gcr.io/liveness 镜像的容器。

[pods/probe/http-liveness.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  labels:  
    test: liveness  
  name: liveness-http  
spec:  
  containers:  
    - name: liveness  
      image: k8s.gcr.io/liveness  
      args:  
        - /server  
      livenessProbe:  
        httpGet:  
          path: /healthz  
          port: 8080  
          httpHeaders:  
            - name: Custom-Header  
              value: Awesome  
        initialDelaySeconds: 3  
        periodSeconds: 3
```

在这个配置文件中，可以看到 Pod 也只有一个容器。 periodSeconds 字段指定了 kubelet 每隔 3 秒执行一次存活探测。 initialDelaySeconds 字段告诉 kubelet 在执行第一次探测前应该等待 3 秒。 kubelet 会向容器内运行的服务（服务会监听 8080 端口）发送一个 HTTP GET 请求来执行探测。如果服务器上 /healthz 路径下的处理程序返回成功代码，则 kubelet 认为容器是健康存活的。如果处理程序返回失败代码，则 kubelet 会杀死这个容器并且重新启动它。

任何大于或等于 200 并且小于 400 的返回代码标示成功，其它返回代码都标示失败。

可以在这里看服务的源码 [server.go](#)。

容器存活的最开始 10 秒中，/healthz 处理程序返回一个 200 的状态码。之后处理程序返回 500 的状态码。

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

kubelet 在容器启动之后 3 秒开始执行健康检测。所以前几次健康检查都是成功的。但是 10 秒之后，健康检查会失败，并且 kubelet 会杀死容器再重新启动容器。

创建一个 Pod 来测试 HTTP 的存活检测：

```
kubectl apply -f https://k8s.io/examples/pods/probe/http-liveness.yaml
```

10 秒之后，通过看 Pod 事件来检测存活探测器已经失败了并且容器被重新启动了。

```
kubectl describe pod liveness-http
```

在 1.13（包括 1.13 版本）之前的版本中，如果在 Pod 运行的节点上设置了环境变量 http_proxy（或者 HTTP_PROXY），HTTP 的存活探测会使用这个代理。在 1.13 之后的版本中，设置本地的 HTTP 代理环境变量不会影响 HTTP 的存活探测。

定义 TCP 的存活探测

第三种类型的存活探测是使用 TCP 套接字。通过配置，kubelet 会尝试在指定端口和容器建立套接字链接。如果能建立连接，这个容器就被看作是健康的，如果不能则这个容器就被看作是有问题的。

[pods/probe/tcp-liveness-readiness.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
    - name: goproxy
      image: k8s.gcr.io/goproxy:0.1
      ports:
        - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20
```

如你所见，TCP 检测的配置和 HTTP 检测非常相似。下面这个例子同时使用就绪和存活探测器。kubelet 会在容器启动 5 秒后发送第一个就绪探测。这会尝试连接 goproxy 容器的 8080 端口。如果探测成功，这个 Pod 会被标记为就绪状态，kubelet 将继续每隔 10 秒运行一次检测。

除了就绪探测，这个配置包括了一个存活探测。kubelet 会在容器启动 15 秒后进行第一次存活探测。就像就绪探测一样，会尝试连接 goproxy 容器的 8080 端口。如果存活探测失败，这个容器会被重新启动。

```
kubectl apply -f https://k8s.io/examples/pods/probe/tcp-liveness-readiness.yaml
```

15 秒之后，通过看 Pod 事件来检测存活探测器：

```
kubectl describe pod goproxy
```

使用命名端口

对于 HTTP 或者 TCP 存活检测可以使用命名的 [ContainerPort](#)。

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080
```

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: liveness-port
```

使用启动探测器保护慢启动容器

有时候，会有一些现有的应用程序在启动时需要较多的初始化时间。要不影响对引起探测死锁的快速响应，这种情况下，设置存活探测参数是要技巧的。技巧就是使用一个命令来设置启动探测，针对HTTP或者TCP检测，可以通过设置 failureThreshold * periodSeconds 参数来保证有足够长的时间应对糟糕情况下的启动时间。

所以，前面的例子就变成了：

```
ports:  
- name: liveness-port  
  containerPort: 8080  
  hostPort: 8080
```

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: liveness-port  
  failureThreshold: 1  
  periodSeconds: 10
```

```
startupProbe:  
  httpGet:  
    path: /healthz  
    port: liveness-port  
  failureThreshold: 30  
  periodSeconds: 10
```

幸亏有启动探测，应用程序将会有最多 5 分钟($30 * 10 = 300s$) 的时间来完成它的启动。一旦启动探测成功一次，存活探测任务就会接管对容器的探测，对容器死锁可以快速响应。如果启动探测一直没有成功，容器会在 300 秒后被杀死，并且根据 restartPolicy 来设置 Pod 状态。

定义就绪探测器

有时候，应用程序会暂时性的不能提供通信服务。例如，应用程序在启动时可能需要加载很大的数据或配置文件，或是启动后要依赖等待外部服务。在这种情况下，既不想杀死应用程序，也不想给它发送请求。Kubernetes 提供了就绪探测器来发现并缓解这些情况。容器所在 Pod 上报还未就绪的信息，并且不接受通过 Kubernetes Service 的流量。

说明：就绪探测器在容器的整个生命周期中保持运行状态。

就绪探测器的配置和存活探测器的配置相似。 唯一区别就是要使用 `readinessProbe` 字段，而不是 `livenessProbe` 字段。

`readinessProbe:`

`exec:`

`command:`

- `cat`
- `/tmp/healthy`

`initialDelaySeconds: 5`

`periodSeconds: 5`

HTTP 和 TCP 的就绪探测器配置也和存活探测器的配置一样的。

就绪和存活探测可以在同一个容器上并行使用。 两者都用可以确保流量不会发给还没有准备好的容器，并且容器会在它们失败的时候被重新启动。

配置探测器

[Probe](#) 有很多配置字段，可以使用这些字段精确的控制存活和就绪检测的行为：

- `initialDelaySeconds`：容器启动后要等待多少秒后存活和就绪探测器才被初始化， 默认是 0 秒，最小值是 0。
- `periodSeconds`：执行探测的时间间隔（单位是秒）。默认是 10 秒。最小值是 1。
- `timeoutSeconds`：探测的超时后等待多少秒。默认值是 1 秒。最小值是 1。
- `successThreshold`：探测器在失败后，被视为成功的最小连续成功数。默认值是 1。 存活和启动探测的这个值必须是 1。最小值是 1。
- `failureThreshold`：当探测失败时，Kubernetes 的重试次数。存活探测情况下的放弃就意味着重新启动容器。 就绪探测情况下的放弃 Pod 会被打上未就绪的标签。默认值是 3。最小值是 1。

[HTTP Probes](#) 可以在 `httpGet` 上配置额外的字段：

- `host`：连接使用的主机名，默认是 Pod 的 IP。也可以在 HTTP 头中设置 "Host" 来代替。
- `scheme`：用于设置连接主机的方式（HTTP 还是 HTTPS）。默认是 HTTP。
- `path`：访问 HTTP 服务的路径。
- `httpHeaders`：请求中自定义的 HTTP 头。HTTP 头字段允许重复。
- `port`：访问容器的端口号或者端口名。如果数字必须在 1 ~ 65535 之间。

对于 HTTP 探测，`kubelet` 发送一个 HTTP 请求到指定的路径和端口来执行检测。除非 `httpGet` 中的 `host` 字段设置了，否则 `kubelet` 默认是给 Pod 的 IP 地址发送探测。如果 `scheme` 字段设置为了 HTTPS，`kubelet` 会跳过证书验证发送 HTTPS 请求。大多数情况下，不需要设置 `host` 字段。这里有个需要设置 `host` 字段的场景，假设容器监听 127.0.0.1，并且 Pod 的 `hostNetwork` 字段设置为了 true。那么 `httpGet` 中的 `host` 字段应该设置为 127.0.0.1。可能更常见的原因是如果 Pod 依赖虚拟主机，你不应该设置 `host` 字段，而是应该在 `httpHeaders` 中设置 Host。

对于一次 TCP 探测，kubelet 在节点上（不是在 Pod 里面）建立探测连接，这意味着你不能在 host 参数上配置服务名称，因为 kubelet 不能解析服务名称。

接下来

- 进一步了解[容器探针](#)。

参考

- [Pod](#)
- [Container](#)
- [Probe](#)

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 27, 2020 at 4:37 PM PST: [Sync tasks/configure-pod-container/configure-liveness-readiness-startup-probes.md \(a121748b8\)](#)

将 Pod 分配给节点

此页面显示如何将 Kubernetes Pod 分配给 Kubernetes 集群中的特定节点。

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.

给节点添加标签

1. 列出集群中的节点

```
kubectl get nodes
```

输出类似如下：

NAME	STATUS	AGE	VERSION
worker0	Ready	1d	v1.6.0+ffff5156
worker1	Ready	1d	v1.6.0+ffff5156
worker2	Ready	1d	v1.6.0+ffff5156

1. 选择其中一个节点，为它添加标签：

```
kubectl label nodes <your-node-name> disktype=ssd
```

<your-node-name> 是你选择的节点的名称。

1. 验证你选择的节点是否有 disktype=ssd 标签：

```
kubectl get nodes --show-labels
```

输出类似如下：

NAME	STATUS	AGE	VERSION	LABELS
worker0	Ready	1d	v1.6.0+ffff5156	...,disktype=ssd,kubernetes.io/hostname=worker0
worker1	Ready	1d	v1.6.0+ffff5156	...,kubernetes.io/hostname=worker1
worker2	Ready	1d	v1.6.0+ffff5156	...,kubernetes.io/hostname=worker2

在前面的输出中，你可以看到 worker0 节点有 disktype=ssd 标签。

创建一个调度到你选择的节点的 pod

此 Pod 配置文件描述了一个拥有节点选择器 disktype: ssd 的 Pod。这表明该 Pod 将被调度到有 disktype=ssd 标签的节点。

[pods/pod-nginx.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

```
nodeSelector:
```

```
disktype: ssd
```

1. 使用该配置文件去创建一个 pod , 该 pod 将被调度到你选择的节点上 :

```
kubectl create -f https://k8s.io/examples/pods/pod-nginx.yaml
```

1. 验证 pod 是不是运行在你选择的节点上 :

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

输出类似如下 :

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

接下来

进一步了解[标签和选择器](#)

反馈

此页是否对您有帮助 ?

是 否

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

最后修改 August 13, 2020 at 5:49 PM PST: [\[zh\] Tidy up and fix links in tasks section \(5/10\) \(68abcb963\)](#)

用节点亲和性把 Pods 分配到节点

本页展示在 Kubernetes 集群中，如何使用节点亲和性把 Kubernetes Pod 分配到特定节点。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.10. 要获知版本信息，请输入 kubectl version.

给节点添加标签

1. 列出集群中的节点及其标签：

```
kubectl get nodes --show-labels
```

输出类似于此：

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker0
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker1
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker2

2. 选择一个节点，给它添加一个标签：

```
kubectl label nodes <your-node-name> disktype=ssd
```

其中 <your-node-name> 是你所选节点的名称。

3. 验证你所选节点具有 disktype=ssd 标签：

```
kubectl get nodes --show-labels
```

输出类似于此：

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,disktype=ssd,kubernetes.io/hostname=worker0
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker1
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker2

在前面的输出中，可以看到 worker0 节点有一个 disktype=ssd 标签。

依据强制的节点亲和性调度 Pod

下面清单描述了一个 Pod，它有一个节点亲和性配置 requiredDuringSchedulingIgnoredDuringExecution，disktype=ssd。这意味着 pod 只会被调度到具有 disktype=ssd 标签的节点上。

[pods/pod-nginx-required-affinity.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
```

1. 执行 (Apply) 此清单来创建一个调度到所选节点上的 Pod :

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-required-affinity.yaml
```

2. 验证 pod 已经在所选节点上运行 :

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

输出类似于此 :

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

使用首选的节点亲和性调度 Pod

本清单描述了一个Pod，它有一个节点亲和性设置 preferredDuringSchedulingIgnoredDuringExecution , disktype: ssd。这意味着 pod 将首选具有 disktype=ssd 标签的节点。

[pods/pod-nginx-preferred-affinity.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
```

```
affinity:  
  nodeAffinity:  
    preferredDuringSchedulingIgnoredDuringExecution:  
      - weight: 1  
        preference:  
          matchExpressions:  
            - key: disktype  
              operator: In  
              values:  
                - ssd  
  containers:  
    - name: nginx  
      image: nginx  
      imagePullPolicy: IfNotPresent
```

1. 执行此清单创建一个会调度到所选节点上的 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx-preferred-affinity.yaml
```

2. 验证 pod 是否在所选节点上运行：

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

输出类似于此：

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

接下来

进一步了解 [节点亲和性](#).

反馈

此页是否对您有帮助？

是 否

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

最后修改 September 24, 2020 at 11:50 AM PST: [\[zh\] add page: /zh/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/ fix 24086 make change according to tengqm's comment \(5586c71b6\)](#)

配置 Pod 初始化

本文介绍在应用容器运行前，怎样利用 Init 容器初始化 Pod。

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.

创建一个包含 Init 容器的 Pod

本例中你将创建一个包含一个应用容器和一个 Init 容器的 Pod。Init 容器在应用容器启动前运行完成。

下面是 Pod 的配置文件：

[pods/init-containers.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: workdir
          mountPath: /usr/share/nginx/html
    # These containers are run during pod initialization
  initContainers:
    - name: install
      image: busybox
      command:
        - wget
        - "-O"
        - "/work-dir/index.html"
```

```
- http://kubernetes.io
volumeMounts:
- name: workdir
  mountPath: "/work-dir"
dnsPolicy: Default
volumes:
- name: workdir
  emptyDir: {}
```

配置文件中，你可以看到应用容器和 Init 容器共享了一个卷。

Init 容器将共享卷挂载到了 /work-dir 目录，应用容器将共享卷挂载到了 /usr/share/nginx/html 目录。Init 容器执行完下面的命令就终止：

```
wget -O /work-dir/index.html http://info.cern.ch
```

请注意 Init 容器在 nginx 服务器的根目录写入 index.html。

创建 Pod：

```
kubectl create -f https://k8s.io/examples/pods/init-containers.yaml
```

检查 nginx 容器运行正常：

```
kubectl get pod init-demo
```

结果表明 nginx 容器运行正常：

NAME	READY	STATUS	RESTARTS	AGE
init-demo	1/1	Running	0	1m

通过 shell 进入 init-demo Pod 中的 nginx 容器：

```
kubectl exec -it init-demo -- /bin/bash
```

在 shell 中，发送个 GET 请求到 nginx 服务器：

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

结果表明 nginx 正在为 Init 容器编写的 web 页面服务：

```
<html><head></head><body><header>
<title>http://info.cern.ch</title>
</header>

<h1>http://info.cern.ch - home of the first website</h1>
...
...
```

```
<li><a href="http://info.cern.ch/hypertext/WWW/TheProject.html">Browse the first website</a></li>
```

```
...
```

接下来

- 进一步了解[同一 Pod 中的容器间的通信](#)。
- 进一步了解[Init 容器](#)。
- 进一步了解[卷](#)。
- 进一步了解[Init 容器排错](#)。

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 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

为容器的生命周期事件设置处理函数

这个页面将演示如何为容器的生命周期事件挂接处理函数。Kubernetes 支持 postStart 和 preStop 事件。当一个容器启动后，Kubernetes 将立即发送 postStart 事件；在容器被终结之前，Kubernetes 将发送一个 preStop 事件。容器可以为每个事件指定一个处理程序。

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.

定义 postStart 和 preStop 处理函数

在本练习中，你将创建一个包含一个容器的 Pod，该容器为 postStart 和 preStop 事件提供对应的处理函数。

下面是对应 Pod 的配置文件：



```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler > /usr/share/message"]
      preStop:
        exec:
          command: ["/bin/sh", "-c", "nginx -s quit; while killall -0 nginx; do sleep 1; done"]
```

在上述配置文件中，你可以看到 postStart 命令在容器的 /usr/share 目录下写入文件 message。命令 preStop 负责优雅地终止 nginx 服务。当因为失效而导致容器终止时，这一处理方式很有用。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/lifecycle-events.yaml
```

验证 Pod 中的容器已经运行：

```
kubectl get pod lifecycle-demo
```

使用 shell 连接到你的 Pod 里的容器：

```
kubectl exec -it lifecycle-demo -- /bin/bash
```

在 shell 中，验证 postStart 处理函数创建了 message 文件：

```
root@lifecycle-demo:/# cat /usr/share/message
```

命令行输出的是 postStart 处理函数所写入的文本

```
Hello from the postStart handler
```

讨论

Kubernetes 在容器创建后立即发送 postStart 事件。然而，postStart 处理函数的调用不保证早于容器的入口点 (entrypoint) 的执行。postStart 处理函数与容器的代码是异步执行的，但 Kubernetes 的容器管理逻辑会一直阻塞等待 postStart 处理函数执行完毕。只有 postStart 处理函数执行完毕，容器的状态才会变成 RUNNING。

Kubernetes 在容器结束前立即发送 preStop 事件。除非 Pod 宽限期超时，Kubernetes 的容器管理逻辑会一直阻塞等待 preStop 处理函数执行完毕。更多的相关细节，可以参阅 [Pods 的结束](#)。

说明：Kubernetes 只有在 Pod 结束 (*Terminated*) 的时候才会发送 preStop 事件，这意味着在 Pod 完成 (*Completed*) 时 preStop 的事件处理逻辑不会被触发。这个限制在 [issue #55087](#) 中被追踪。

接下来

- 进一步了解[容器生命周期回调](#)。
- 进一步了解[Pod 的生命周期](#)。

参考

- [Lifecycle](#)
- [Container](#)
- 参阅 [PodSpec](#) 中关于terminationGracePeriodSeconds 的部分

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 11, 2020 at 10:17 AM PST: [\[zh\]Sync "State that no more than one handler is allowed per lifecycle event #25547"](#) (adc4dd4f3)

配置 Pod 使用 ConfigMap

ConfigMap 允许你将配置文件与镜像文件分离，以使容器化的应用程序具有可移植性。本页提供了一系列使用示例，这些示例演示了如何创建 ConfigMap 以及配置 Pod 使用存储在 ConfigMap 中的数据。

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.

创建 ConfigMap

你可以使用 kubectl create configmap 或者在 kustomization.yaml 中的 ConfigMap 生成器来创建 ConfigMap。注意，kubectl 从 1.14 版本开始支持 kustomization.yaml。

使用 kubectl create configmap 创建 ConfigMap

你可以使用 kubectl create configmap 命令基于 [目录、文件](#) 或者[字面值](#)来创建 ConfigMap：

```
kubectl create configmap <map-name> <data-source>
```

其中，<map-name> 是要设置的 ConfigMap 名称，<data-source> 是要从中提取数据的目录、文件或者字面值。ConfigMap 对象的名称必须是合法的 [DNS 子域名](#)。

在你基于文件来创建 ConfigMap 时，<data-source> 中的键名默认取自 文件的基本名，而对应的值则默认为文件的内容。

你可以使用[kubectl describe](#) 或者 [kubectl get](#) 获取有关 ConfigMap 的信息。

基于目录创建 ConfigMap

你可以使用 kubectl create configmap 基于同一目录中的多个文件创建 ConfigMap。当你基于目录来创建 ConfigMap 时，kubectl 识别目录下基本名可以作为合法键名的文件，并将这些文件打包到新的 ConfigMap 中。普通文件之外的所有目录项都会被忽略（例如，子目录、符号链接、设备、管道等等）。

例如：

```
# 创建本地目录
mkdir -p configure-pod-container/configmap/

# 将实例文件下载到 `configure-pod-container/configmap/` 目录
wget https://kubernetes.io/examples/configmap/game.properties -O configure-
pod-container/configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O configure-pod-
container/configmap/ui.properties

# 创建 configmap
```

```
kubectl create configmap game-config --from-file=configure-pod-container/configmap/
```

以上命令将 configure-pod-container/configmap 目录下的所有文件，也就是 game.properties 和 ui.properties 打包到 game-config ConfigMap 中。你可以使用下面的命令显示 ConfigMap 的详细信息：

```
kubectl describe configmaps game-config
```

输出类似以下内容：

```
Name:      game-config
Namespace: default
Labels:    <none>
Annotations: <none>

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
-----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

configure-pod-container/configmap/ 目录中的 game.properties 和 ui.properties 文件出现在 ConfigMap 的 data 部分。

```
kubectl get configmaps game-config -o yaml
```

输出类似以下内容：

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
```

```
selfLink: /api/v1/namespaces/default/configmaps/game-config
uid: b4952dc3-d670-11e5-8cd0-68f728db1985
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
```

基于文件创建 ConfigMap

你可以使用 `kubectl create configmap` 基于单个文件或多个文件创建 ConfigMap。

例如：

```
kubectl create configmap game-config-2 --from-file=configure-pod-container/
configmap/game.properties
```

将产生以下 ConfigMap:

```
kubectl describe configmaps game-config-2
```

输出类似以下内容:

```
Name:      game-config-2
Namespace: default
Labels:    <none>
Annotations: <none>

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
```

```
secret.code.allowed=true  
secret.code.lives=30
```

你可以多次使用 `--from-file` 参数，从多个数据源创建 ConfigMap。

```
kubectl create configmap game-config-2 --from-file=configure-pod-container/  
configmap/game.properties --from-file=configure-pod-container/configmap/  
ui.properties
```

描述上面创建的 game-config-2 configmap

```
kubectl describe configmaps game-config-2
```

输出类似以下内容：

```
Name:      game-config-2  
Namespace: default  
Labels:    <none>  
Annotations: <none>  
  
Data  
=====
```

game.properties:

```
----  
enemies=aliens  
lives=3  
enemies.cheat=true  
enemies.cheat.level=noGoodRotten  
secret.code.passphrase=UUDDLRLRBABAS  
secret.code.allowed=true  
secret.code.lives=30  
ui.properties:  
----  
color.good=purple  
color.bad=yellow  
allow.textmode=true  
how.nice.to.look=fairlyNice
```

使用 `--from-env-file` 选项从环境文件创建 ConfigMap，例如：

Env 文件包含环境变量列表。 其中适用以下语法规则：

- Env 文件中的每一行必须为 `VAR=VAL` 格式。
- 以 `#` 开头的行（即注释）将被忽略。
- 空行将被忽略。
- 引号不会被特殊处理（即它们将成为 ConfigMap 值的一部分）。

将示例文件下载到 `configure-pod-container/configmap/` 目录

```
 wget https://kubernetes.io/examples/configmap/game-env-file.properties -O configure-pod-container/configmap/game-env-file.properties
```

env 文件 game-env-file.properties 如下所示：

```
 cat configure-pod-container/configmap/game-env-file.properties
```

```
 enemies=aliens  
 lives=3  
 allowed="true"
```

```
 kubectl create configmap game-config-env-file \  
   --from-env-file=configure-pod-container/configmap/game-env-  
   file.properties
```

将产生以下 ConfigMap：

```
 kubectl get configmap game-config-env-file -o yaml
```

输出类似以下内容：

```
 apiVersion: v1  
 kind: ConfigMap  
 metadata:  
   creationTimestamp: 2017-12-27T18:36:28Z  
   name: game-config-env-file  
   namespace: default  
   resourceVersion: "809965"  
   selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file  
   uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8  
 data:  
   allowed: '"true"'  
   enemies: aliens  
   lives: "3"
```

注意：当多次使用 --from-env-file 来从多个数据源创建 ConfigMap 时，仅仅最后一个 env 文件有效。

下面是一个多次使用 --from-env-file 参数的示例：

```
# 将样本文件下载到 `configure-pod-container/configmap/` 目录  
 wget https://k8s.io/examples/configmap/ui-env-file.properties -O configure-pod-  
 container/configmap/ui-env-file.properties  
  
 # 创建 configmap  
 kubectl create configmap config-multi-env-files \  
   --from-env-file=configure-pod-container/configmap/game-env-
```

```
file.properties \
--from-env-file=configure-pod-container/configmap/ui-env-file.properties
```

将产生以下 ConfigMap:

```
kubectl get configmap config-multi-env-files -o yaml
```

输出类似以下内容:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  selfLink: /api/v1/namespaces/default/configmaps/config-multi-env-files
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
data:
  color: purple
  how: fairlyNice
  textmode: "true"
```

定义从文件创建 ConfigMap 时要使用的键

在使用 --from-file 参数时，你可以定义在 ConfigMap 的 data 部分出现键名，而不是按默认行为使用文件名：

```
kubectl create configmap game-config-3 --from-file=<my-key-name>=<path-to-file>
```

<my-key-name> 是你要在 ConfigMap 中使用的键名，<path-to-file> 是你想要键表示数据源文件的位置。

例如：

```
kubectl create configmap game-config-3 --from-file=game-special-key=configure-pod-container/configmap/game.properties
```

将产生以下 ConfigMap:

```
kubectl get configmaps game-config-3 -o yaml
```

输出类似以下内容：

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
```

```
name: game-config-3
namespace: default
resourceVersion: "530"
selfLink: /api/v1/namespaces/default/configmaps/game-config-3
uid: 05f8da22-d671-11e5-8cd0-68f728db1985
data:
  game-special-key: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
```

根据字面值创建 ConfigMap

你可以将 kubectl create configmap 与 --from-literal 参数一起使用，从命令行定义文字值：

```
kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm
```

你可以传入多个键值对。命令行中提供的每对键值在 ConfigMap 的 data 部分中均表示为单独的条目。

```
kubectl get configmaps special-config -o yaml
```

输出类似以下内容：

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
data:
  special.how: very
  special.type: charm
```

基于生成器创建 ConfigMap

自 1.14 开始，kubectl 开始支持 kustomization.yaml。你还可以基于生成器创建 ConfigMap，然后将其应用于 API 服务器上创建对象。生成器应在目录内的 kustomization.yaml 中指定。

基于文件生成 ConfigMap

例如，要从 configure-pod-container/configmap/kubectl/game.properties 文件生成一个 ConfigMap：

```
# 创建包含 ConfigMapGenerator 的 kustomization.yaml 文件
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-4
  files:
  - configure-pod-container/configmap/kubectl/game.properties
EOF
```

使用 kustomization 目录创建 ConfigMap 对象：

```
kubectl apply -k .
```

```
configmap/game-config-4-m9dm2f92bt created
```

你可以检查 ConfigMap 是这样创建的：

```
kubectl get configmap
```

NAME	DATA	AGE
game-config-4-m9dm2f92bt	1	37s

```
kubectl describe configmaps/game-config-4-m9dm2f92bt
```

```
Name: game-config-4-m9dm2f92bt
```

```
Namespace: default
```

```
Labels: <none>
```

```
Annotations: kubectl.kubernetes.io/last-applied-configuration:
```

```
 {"apiVersion": "v1", "data": "
```

```
{"game.properties": "enemies=aliens\\nlives=3\\nenemies.cheat=true\\nenemies.cheat.level=noGoodRotten\\nsecret.code.p..."}
```

```
Data
```

```
====
```

```
game.properties:
```

```
----
```

```
enemies=aliens
```

```
lives=3
```

```
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
Events: <none>
```

请注意，生成的 ConfigMap 名称具有通过对内容进行散列而附加的后缀，这样可以确保每次修改内容时都会生成新的 ConfigMap。

定义从文件生成 ConfigMap 时要使用的键

在 ConfigMap 生成器，你可以定义一个非文件名的键名。例如，从 configure-pod-container/configmap/game.properties 文件生成 ConfigMap，但使用 game-special-key 作为键名：

```
# 创建包含 ConfigMapGenerator 的 kustomization.yaml 文件
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-5
  files:
    - game-special-key=configure-pod-container/configmap/kubectl/
game.properties
EOF
```

使用 Kustomization 目录创建 ConfigMap 对象。

```
kubectl apply -k .
```

```
configmap/game-config-5-m67dt67794 created
```

从字面值生成 ConfigMap

要基于字符串 special.type=charm 和 special.how=very 生成 ConfigMap，可以在 kusotmization.yaml 中配置 ConfigMap 生成器：

```
# 创建带有 ConfigMapGenerator 的 kustomization.yaml 文件
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: special-config-2
  literals:
    - special.how=very
    - special.type=charm
EOF
```

应用 Kustomization 目录创建 ConfigMap 对象。

```
kubectl apply -k .
```

```
configmap/special-config-2-c92b5mmcf2 created
```

使用 ConfigMap 数据定义容器环境变量

使用单个 ConfigMap 中的数据定义容器环境变量

- 在 ConfigMap 中将环境变量定义为键值对:

```
kubectl create configmap special-config --from-literal=special.how=very
```

- 将 ConfigMap 中定义的 special.how 值分配给 Pod 规范中的 SPECIAL_LEVEL_KEY 环境变量。

[pods/pod-single-configmap-env-variable.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        # Define the environment variable
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              # The ConfigMap containing the value you want to assign to
              # SPECIAL_LEVEL_KEY
              name: special-config
              # Specify the key associated with the value
              key: special.how
  restartPolicy: Never
```

创建 Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-single-
configmap-env-variable.yaml
```

现在，Pod 的输出包含环境变量 SPECIAL_LEVEL_KEY=very。

使用来自多个 ConfigMap 的数据定义容器环境变量

- 与前面的示例一样，首先创建 ConfigMap。

[configmap/configmaps.yaml](#)



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

创建 ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/
configmaps.yaml
```

- 在 Pod 规范中定义环境变量。

[pods/pod-multiple-configmap-env-variable.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
```

```
name: env-config
key: log_level
restartPolicy: Never
```

创建 Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-multiple-
configmap-env-variable.yaml
```

现在，Pod 的输出包含环境变量 SPECIAL_LEVEL_KEY=very 和 LOG_LEVEL=INFO。

将 ConfigMap 中的所有键值对配置为容器环境变量

说明：Kubernetes v1.6 和更高版本支持此功能。

- 创建一个包含多个键值对的 ConfigMap。

[configmap/configmap-multikeys.yaml](#)



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

创建 ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmap-
multikeys.yaml
```

- 使用 envFrom 将所有 ConfigMap 的数据定义为容器环境变量，ConfigMap 中的键成为 Pod 中的环境变量名称。

[pods/pod-configmap-envFrom.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
```

```
image: k8s.gcr.io/busybox
command: [ "/bin/sh", "-c", "env" ]
envFrom:
- configMapRef:
  name: special-config
restartPolicy: Never
```

创建 Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-envFrom.yaml
```

现在，Pod 的输出包含环境变量 SPECIAL_LEVEL=very 和 SPECIAL_TYPE=charm。

在 Pod 命令中使用 ConfigMap 定义的环境变量

你可以使用 `$(VAR_NAME)` Kubernetes 替换语法在容器的 command 和 args 部分中使用 ConfigMap 定义的环境变量。

例如，以下 Pod 规范

[pods/pod-configmap-env-var-valueFrom.yaml](#)


```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_LEVEL
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_TYPE
  restartPolicy: Never
```

通过运行下面命令创建 Pod :

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-env-var-valueFrom.yaml
```

在 test-container 容器中产生以下输出:

```
very charm
```

将 ConfigMap 数据添加到一个卷中

如基于文件创建 [ConfigMap](#) 中所述，当你使用 --from-file 创建 ConfigMap 时，文件名成为存储在 ConfigMap 的 data 部分中的键，文件内容成为键对应的值。

本节中的示例引用了一个名为 special-config 的 ConfigMap，如下所示：

[configmap/configmap-multikeys.yaml](#)



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

创建 ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmap-multikeys.yaml
```

使用存储在 ConfigMap 中的数据填充数据卷

在 Pod 规约的 volumes 部分下添加 ConfigMap 名称。这会将 ConfigMap 数据添加到指定为 volumeMounts.mountPath 的目录（在本例中为 /etc/config）。command 部分引用存储在 ConfigMap 中的 special.level。

[pods/pod-configmap-volume.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
```

```
containers:  
  - name: test-container  
    image: k8s.gcr.io/busybox  
    command: [ "/bin/sh", "-c", "ls /etc/config/" ]  
volumeMounts:  
  - name: config-volume  
    mountPath: /etc/config  
volumes:  
  - name: config-volume  
configMap:  
    # Provide the name of the ConfigMap containing the files you want  
    # to add to the container  
    name: special-config  
restartPolicy: Never
```

创建 Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-  
volume.yaml
```

Pod 运行时，命令 ls /etc/config/ 产生下面的输出：

```
SPECIAL_LEVEL  
SPECIAL_TYPE
```

注意：如果在 /etc/config/ 目录中有一些文件，它们将被删除。

说明：文本数据会使用 UTF-8 字符编码的形式展现为文件。如果使用其他字符编码，可以使用 binaryData。

将 ConfigMap 数据添加到数据卷中的特定路径

使用 path 字段为特定的 ConfigMap 项目指定预期的文件路径。在这里，SPECIAL_LEVEL 将挂载在 config-volume 数据卷中 /etc/config/keys 目录下。

[pods/pod-configmap-volume-specific-key.yaml](#)

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: dapi-test-pod  
spec:  
  containers:  
    - name: test-container  
      image: k8s.gcr.io/busybox  
      command: [ "/bin/sh", "-c", "cat /etc/config/keys" ]
```

```
volumeMounts:  
- name: config-volume  
  mountPath: /etc/config  
volumes:  
- name: config-volume  
configMap:  
  name: special-config  
  items:  
  - key: SPECIAL_LEVEL  
    path: keys  
restartPolicy: Never
```

创建Pod：

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume-specific-key.yaml
```

当 pod 运行时，命令 cat /etc/config/keys 产生以下输出：

```
very
```

注意：如前，/etc/config/ 目录中所有先前的文件都将被删除。

映射键以指定路径和文件权限

你可以通过指定键名到特定目录的投射关系，也可以逐个文件地设定访问权限。[Secret 用户指南](#) 中对这一语法提供了解释。

挂载的 ConfigMap 将自动更新

更新已经在数据卷中使用的 ConfigMap 时，已映射的键最终也会被更新。kubelet 在每次定期同步时都会检查已挂载的 ConfigMap 是否是最新的。但是，它使用其本地的基于 TTL 的缓存来获取 ConfigMap 的当前值。因此，从更新 ConfigMap 到将新键映射到 Pod 的总延迟可能与 kubelet 同步周期 + ConfigMap 在 kubelet 中缓存的 TTL 一样长。

说明：使用 ConfigMap 作为 [subPath](#) 的数据卷将不会收到 ConfigMap 更新。

了解 ConfigMap 和 Pod

ConfigMap API 资源将配置数据存储为键值对。数据可以在 Pod 中使用，也可以提供系统组件（如控制器）的配置。ConfigMap 与 [Secret](#) 类似，但是提供了一种使用不包含敏感信息的字符串的方法。用户和系统组件都可以在 ConfigMap 中存储配置数据。

说明：ConfigMap 应该引用属性文件，而不是替换它们。可以将 ConfigMap 理解为类似于 Linux /etc 目录及其内容的东西。例如，如果你

从 ConfigMap 创建 [Kubernetes 卷](#)，则 ConfigMap 中的每个数据项都由该数据卷中的单个文件表示。

ConfigMap 的 data 字段包含配置数据。如下例所示，它可以简单（如用 --from-literal 的单个属性定义）或复杂（如用 --from-file 的配置文件或 JSON blob 定义）。

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  # 使用 --from-literal 定义的简单属性
  example.property.1: hello
  example.property.2: world
  # 使用 --from-file 定义复杂属性的例子
  example.property.file: |-  
    property.1=value-1  
    property.2=value-2  
    property.3=value-3
```

限制

- 在 Pod 规范中引用之前，必须先创建一个 ConfigMap（除非将 ConfigMap 标记为“可选”）。如果引用的 ConfigMap 不存在，则 Pod 将不会启动。同样，引用 ConfigMap 中不存在的键也会阻止 Pod 启动。
- 如果你使用 envFrom 基于 ConfigMap 定义环境变量，那么无效的键将被忽略。可以启动 Pod，但无效名称将记录在事件日志中（InvalidVariableNames）。日志消息列出了每个跳过的键。例如：

```
kubectl get events
```

输出与此类似：

LASTSEEN	FIRSTSEEN	COUNT	NAME	KIND	SUBOBJECT	TYPE
REASON		SOURCE			MESSAGE	
0s	0s	1	dapi-test-pod	Pod	Warning	
InvalidEnvironmentVariableNames {kubelet, 127.0.0.1} Keys [1badkey, 2alsobad] from the EnvFrom configMap default/myconfig were skipped since they are considered invalid environment variable names.						

- ConfigMap 位于特定的[名字空间](#)中。每个 ConfigMap 只能被同一名字空间中的 Pod 引用。
- 你不能将 ConfigMap 用于 [静态 Pod](#)，因为 Kubernetes 不支持这种用法。

接下来

- 浏览[使用 ConfigMap 配置 Redis 真实实例](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 08, 2020 at 2:49 PM PST: [Update configure-pod-configmap.md \(f6b254cf1\)](#)

在 Pod 中的容器之间共享进程命名空间

FEATURE STATE: Kubernetes v1.17 [stable]

此页面展示如何为 pod 配置进程命名空间共享。当启用进程命名空间共享时，容器中的进程对该 pod 中的所有其他容器都是可见的。

您可以使用此功能来配置协作容器，比如日志处理 sidecar 容器，或者对那些不包含诸如 shell 等调试实用工具的镜像进行故障排查。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.10. 要获知版本信息，请输入 kubectl version.

配置 Pod

进程命名空间共享使用 v1.PodSpec 中的 ShareProcessNamespace 字段启用。例如：

[pods/share-process-namespace.yaml](#)


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

```
metadata:  
  name: nginx  
spec:  
  shareProcessNamespace: true  
  containers:  
    - name: nginx  
      image: nginx  
    - name: shell  
      image: busybox  
  securityContext:  
    capabilities:  
      add:  
        - SYS_PTRACE  
  stdin: true  
  tty: true
```

1. 在集群中创建 nginx pod :

```
kubectl apply -f https://k8s.io/examples/pods/share-process-  
namespace.yaml
```

2. 获取容器 shell , 执行 ps :

```
kubectl attach -it nginx -c shell
```

如果没有看到命令提示符 , 请按 enter 回车键。

```
/ # ps ax  
PID  USER  TIME  COMMAND  
 1 root   0:00 /pause  
 8 root   0:00 nginx: master process nginx -g daemon off;  
14 101   0:00 nginx: worker process  
15 root   0:00 sh  
21 root   0:00 ps ax
```

您可以在其他容器中对进程发出信号。例如 , 发送 SIGHUP 到 nginx 以重启工作进程。
这需要 SYS_PTRACE 功能。

```
/ # kill -HUP 8  
/ # ps ax  
PID  USER  TIME  COMMAND  
 1 root   0:00 /pause  
 8 root   0:00 nginx: master process nginx -g daemon off;  
15 root   0:00 sh  
22 101   0:00 nginx: worker process  
23 root   0:00 ps ax
```

甚至可以使用 /proc/\$pid/root 链接访问另一个容器镜像。

```
/ # head /proc/8/root/etc/nginx/nginx.conf

user nginx;
worker_processes 1;

error_log /var/log/nginx/error.log warn;
pid      /var/run/nginx.pid;

events {
    worker_connections 1024;
```

理解进程命名空间共享

Pod 共享许多资源，因此它们共享进程命名空间是很有意义的。不过，有些容器镜像可能希望与其他容器隔离，因此了解这些差异很重要：

1. **容器进程不再具有 PID 1。** 在没有 PID 1 的情况下，一些容器镜像拒绝启动（例如，使用 systemd 的容器），或者拒绝执行 kill -HUP 1 之类的命令来通知容器进程。在具有共享进程命名空间的 pod 中，kill -HUP 1 将通知 pod 沙箱（在上面的例子中是 /pause）。
2. **进程对 pod 中的其他容器可见。** 这包括 /proc 中可见的所有信息，例如作为参数或环境变量传递的密码。这些仅受常规 Unix 权限的保护。
3. **容器文件系统通过 /proc/\$pid/root 链接对 pod 中的其他容器可见。** 这使调试更加容易，但也意味着文件系统安全性只受文件系统权限的保护。

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 \(4b35d4d40\)](#)

创建静态 Pod

静态 Pod 在指定的节点上由 kubelet 守护进程直接管理，不需要 [API 服务器](#) 监管。与由控制面管理的 Pod（例如，[Deployment](#)）不同；kubelet 监视每个静态 Pod（在它崩溃之后重新启动）。

静态 Pod 永远都会绑定到一个指定节点上的 [Kubelet](#)。

kubelet 会尝试通过 Kubernetes API 服务器为每个静态 Pod 自动创建一个 [镜像 Pod](#)。这意味着节点上运行的静态 Pod 对 API 服务来说是可见的，但是不能通过 API 服务器来控制。

说明：如果你在运行一个 Kubernetes 集群，并且在每个节点上都运行一个静态 Pod，就可能需要考虑使用 [DaemonSet](#) 替代这种方式。

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.

本文假定你在使用 [Docker](#) 来运行 Pod，并且你的节点是运行着 Fedora 操作系统。其它发行版或者 Kubernetes 部署版本上操作方式可能不一样。

创建静态 Pod

可以通过[文件系统上的配置文件](#) 或者 [web 网络上的配置文件](#) 来配置静态 Pod。

文件系统上的静态 Pod 声明文件

声明文件是标准的 Pod 定义文件，以 JSON 或者 YAML 格式存储在指定目录。路径设置在 [Kubelet 配置文件](#) 的 staticPodPath: <目录> 字段，kubelet 会定期的扫描这个文件夹下的 YAML/JSON 文件来创建/删除静态 Pod。注意 kubelet 扫描目录的时候会忽略以点开头的文件。

例如：下面是如何以静态 Pod 的方式启动一个简单 web 服务：

1. 选择一个要运行静态 Pod 的节点。在这个例子中选择 my-node1。

```
ssh my-node1
```

1. 选择一个目录，比如在 /etc/kubelet.d 目录来保存 web 服务 Pod 的定义文件，/etc/kubelet.d/static-web.yaml：

```
# 在 kubelet 运行的节点上执行以下命令
mkdir /etc/kubelet.d/
cat <<EOF >/etc/kubelet.d/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
```

```
role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
EOF
```

1. 配置这个节点上的 kubelet , 使用这个参数执行 --pod-manifest-path=/etc/kubelet.d/。在 Fedora 上编辑 /etc/kubernetes/kubelet 以包含下行 :

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --pod-manifest-path=/etc/kubelet.d/"
```

或者在 [Kubelet配置文件](#) 中添加 staticPodPath: <目录>字段。

1. 重启 kubelet。Fedora 上使用下面的命令 :

```
# 在 kubelet 运行的节点上执行以下命令
systemctl restart kubelet
```

Web 网上的静态 Pod 声明文件

Kubelet 根据 --manifest-url=<URL> 参数的配置定期的下载指定文件，并且转换成 JSON/YAML 格式的 Pod 定义文件。与[文件系统上的清单文件](#)使用方式类似，kubelet 调度获取清单文件。如果静态 Pod 的清单文件有改变，kubelet 会应用这些改变。

按照下面的方式来 :

1. 创建一个 YAML 文件，并保存在 web 服务上，为 kubelet 生成一个 URL。

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
```

```
containerPort: 80  
protocol: TCP
```

1. 通过在选择的节点上使用 `--manifest-url=<manifest-url>` 配置运行 kubelet。在 Fedora 添加下面这行到 `/etc/kubernetes/kubelet` :

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-domain=kube.local --  
manifest-url=<manifest-url>"
```

1. 重启 kubelet。在 Fedora 上运行如下命令 :

```
# 在 kubelet 运行的节点上执行以下命令  
systemctl restart kubelet
```

观察静态 pod 的行为

当 kubelet 启动时，会自动启动所有定义的静态 Pod。当定义了一个静态 Pod 并重新启动 kubelet 时，新的静态 Pod 就应该已经在运行了。

可以在节点上运行下面的命令来查看正在运行的容器（包括静态 Pod）：

```
# 在 kubelet 运行的节点上执行以下命令  
docker ps
```

输出可能会像这样：

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
f6d05272b57e	nginx:latest	"nginx"	8 minutes ago	Up 8 minutes	
k8s_web.6f802af4_static-web-fk-					
node1_default_67e24ed9466ba55986d120c867395f3c_378e5f3c					

可以在 API 服务上看到镜像 Pod：

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	2m

说明：要确保 kubelet 在 API 服务上有创建镜像 Pod 的权限。如果没有，创建请求会被 API 服务拒绝。可以看[Pod安全策略](#)。

静态 Pod 上的[标签](#)被传到镜像 Pod。你可以通过[选择算符](#)使用这些标签。

如果你用 kubectl 从 API 服务上删除镜像 Pod，kubelet 不会 移除静态 Pod：

```
kubectl delete pod static-web-my-node1
```

```
pod "static-web-my-node1" deleted
```

可以看到 Pod 还在运行：

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	12s

回到 kubelet 运行的节点上，可以手工停止 Docker 容器。可以看到过了一段时间后 kubelet 会发现容器停止了并且会自动重启 Pod：

```
# 在 kubelet 运行的节点上执行以下命令
```

```
# 把 ID 换为你的容器的 ID
```

```
docker stop f6d05272b57e  
sleep 20  
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	...
5b920cbaf8b1	nginx:latest	"nginx -g 'daemon off;'	2 seconds ago	...

动态增加和删除静态 pod

运行中的 kubelet 会定期扫描配置的目录(比如例子中的 /etc/kubelet.d 目录)中的变化，并且根据文件中出现/消失的 Pod 来添加/删除 Pod。

```
# 前提是你在用主机文件系统上的静态 Pod 配置文件
```

```
# 在 kubelet 运行的节点上执行以下命令
```

```
mv /etc/kubelet.d/static-web.yaml /tmp  
sleep 20
```

```
docker ps
```

```
# 可以看到没有 nginx 容器在运行
```

```
mv /tmp/static-web.yaml /etc/kubelet.d/  
sleep 20  
docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	...
e7a62e3427f1	nginx:latest	"nginx -g 'daemon off;'"	27 seconds ago	...

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 19, 2020 at 12:20 AM PST: [Update static-pod.md](#)
[\(832201373\)](#)

将 Docker Compose 文件转换为 Kubernetes 资源

Kompose 是什么？它是个转换工具，可将 compose（即 Docker Compose）所组装的所有内容 转换成容器编排器（Kubernetes 或 OpenShift）可识别的形式。

更多信息请参考 Kompose 官网 <http://kompose.io>。

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.

安装 Kompose

我们有很多种方式安装 Kompose。首选方式是从最新的 GitHub 发布页面下载二进制文件。

GitHub 发布版本

Kompose 通过 GitHub 发布版本，发布周期为三星期。你可以在 [GitHub 发布页面](#) 上看到所有当前版本。

Linux

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/  
kompose-linux-amd64 -o kompose
```

macOS

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/  
kompose-darwin-amd64 -o kompose
```

Windows

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/  
kompose-windows-amd64.exe -o kompose.exe
```

```
chmod +x kompose  
sudo mv ./kompose /usr/local/bin/kompose
```

或者，你可以下载 [tarball](#)。

Go

用 go get 命令从主分支拉取最新的开发变更的方法安装 Kompose。

```
go get -u github.com/kubernetes/kompose
```

CentOS

Kompose 位于 [EPEL](#) CentOS 代码仓库。 如果你还没有安装启用 [EPEL](#) 代码仓库，请运行命令 sudo yum install epel-release。

如果你的系统中已经启用了 [EPEL](#)， 你就可以像安装其他软件包一样安装 Kompose。

```
sudo yum -y install kompose
```

Fedora

Kompose 位于 Fedora 24、25 和 26 的代码仓库。你可以像安装其他软件包一样安装 Kompose。

```
sudo dnf -y install kompose
```

macOS

在 macOS 上你可以通过 [Homebrew](#) 安装 Kompose 的最新版本：

```
brew install kompose
```

使用 Kompose

再需几步，我们就把你从 Docker Compose 带到 Kubernetes。 你只需要一个现有的 docker-compose.yml 文件。

1. 进入 docker-compose.yml 文件所在的目录。如果没有，请使用下面这个进行测试。

```
version: "2"

services:

  redis-master:
    image: k8s.gcr.io/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v3
```

```
ports:
  - "6379"
environment:
  - GET_HOSTS_FROM=dns

frontend
  image: gcr.io/google-samples/gb-frontend:v4
  ports:
    - "80:80"
  environment:
    - GET_HOSTS_FROM=dns
  labels:
    kompose.service.type: LoadBalancer
```

2. 运行 kompose up 命令直接部署到 Kubernetes , 或者跳到下一步 , 生成 kubectl 使用的文件。

```
$ kompose up
We are going to create Kubernetes Deployments, Services and
PersistentVolumeClaims for your Dockerized application.
If you need different kind of resources, use the 'kompose convert' and 'kube
ctl create -f' commands instead.

INFO Successfully created Service: redis
INFO Successfully created Service: web
INFO Successfully created Deployment: redis
INFO Successfully created Deployment: web

Your application has been deployed to Kubernetes. You can run 'kubectl get
deployment,svc,pods,pvc' for details.
```

3. 要将 docker-compose.yml 转换为 kubectl 可用的文件 , 请运行 kompose convert 命令进行转换 , 然后运行 kubectl create -f <output file> 进行创建。

```
kompose convert

INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

```
kubectl create -f frontend-service.yaml,redis-master-service.yaml,redis-
slave-service.yaml,frontend-deployment.yaml,redis-master-
deployment.yaml,redis-slave-deployment.yaml
```

```
service/frontend created
service/redis-master created
service/redis-slave created
deployment.apps/frontend created
deployment.apps/redis-master created
deployment.apps/redis-slave created
```

你部署的应用在 Kubernetes 中运行起来了。

4. 访问你的应用

如果你在开发过程中使用 minikube , 请执行 :

```
minikube service frontend
```

否则 , 我们要查看一下你的服务使用了什么 IP !

```
kubectl describe svc frontend
```

```
Name:           frontend
Namespace:      default
Labels:         service=frontend
Selector:       service=frontend
Type:          LoadBalancer
IP:            10.0.0.183
LoadBalancer Ingress: 192.0.2.89
Port:          80    80/TCP
NodePort:       80    31144/TCP
Endpoints:     172.17.0.4:80
Session Affinity: None
No events.
```

如果你使用的是云提供商 , 你的 IP 将在 LoadBalancer Ingress 字段给出。

```
curl http://192.0.2.89
```

用户指南

- CLI

- [kompose convert](#)
- [kompose up](#)
- [kompose down](#)

- 文档

- [构建和推送 Docker 镜像](#)
- [其他转换方式](#)
- [标签](#)

- [重启](#)
- [Docker Compose 版本](#)

Kompose 支持两种驱动：OpenShift 和 Kubernetes。你可以通过全局选项 --provider 选择驱动方式。如果没有指定，会将 Kubernetes 作为默认驱动。

kompose convert

Kompose 支持将 V1、V2 和 V3 版本的 Docker Compose 文件转换为 Kubernetes 和 OpenShift 资源对象。

Kubernetes

```
kompose --file docker-voting.yml convert
```

```
WARN Unsupported key networks - ignoring
WARN Unsupported key build - ignoring
INFO Kubernetes file "worker-svc.yaml" created
INFO Kubernetes file "db-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "result-svc.yaml" created
INFO Kubernetes file "vote-svc.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
INFO Kubernetes file "result-deployment.yaml" created
INFO Kubernetes file "vote-deployment.yaml" created
INFO Kubernetes file "worker-deployment.yaml" created
INFO Kubernetes file "db-deployment.yaml" created
```

```
ls
```

```
db-deployment.yaml docker-compose.yml docker-gitlab.yml redis-
deployment.yaml result-deployment.yaml vote-deployment.yaml worker-
deployment.yaml
db-svc.yaml docker-voting.yml redis-svc.yaml result-svc.yaml
vote-svc.yaml worker-svc.yaml
```

你也可以同时提供多个 docker-compose 文件进行转换：

```
kompose -f docker-compose.yml -f docker-guestbook.yml convert
```

```
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "mlbparks-service.yaml" created
INFO Kubernetes file "mongodb-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "mlbparks-deployment.yaml" created
```

```
INFO Kubernetes file "mongodb-deployment.yaml" created
INFO Kubernetes file "mongodb-claim0-persistentvolumeclaim.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created
```

ls

```
mlbparks-deployment.yaml  mongodb-service.yaml          redis-slave-
service.jsonmlbparks-service.yaml
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  redis-
master-service.yaml
frontend-service.yaml     mongodb-deployment.yaml      redis-slave-
deployment.yaml
redis-master-deployment.yaml
```

当提供多个 docker-compose 文件时，配置将会合并。任何通用的配置都将被后续文件覆盖。

OpenShift

```
kompose --provider openshift --file docker-voting.yml convert
```

```
WARN [worker] Service cannot be created because of missing port.
INFO OpenShift file "vote-service.yaml" created
INFO OpenShift file "db-service.yaml" created
INFO OpenShift file "redis-service.yaml" created
INFO OpenShift file "result-service.yaml" created
INFO OpenShift file "vote-deploymentconfig.yaml" created
INFO OpenShift file "vote-imagestream.yaml" created
INFO OpenShift file "worker-deploymentconfig.yaml" created
INFO OpenShift file "worker-imagestream.yaml" created
INFO OpenShift file "db-deploymentconfig.yaml" created
INFO OpenShift file "db-imagestream.yaml" created
INFO OpenShift file "redis-deploymentconfig.yaml" created
INFO OpenShift file "redis-imagestream.yaml" created
INFO OpenShift file "result-deploymentconfig.yaml" created
INFO OpenShift file "result-imagestream.yaml" created
```

kompose 还支持为服务中的构建指令创建 buildconfig。默认情况下，它使用当前 git 分支的 remote 仓库作为源仓库，使用当前分支作为构建的源分支。你可以分别使用 --build-repo 和 --build-branch 选项指定不同的源仓库和分支。

```
kompose --provider openshift --file buildconfig/docker-compose.yml convert
```

```
WARN [foo] Service cannot be created because of missing port.
INFO OpenShift Buildconfig using git@github.com:rtnpro/kompose.git::master
as source.
INFO OpenShift file "foo-deploymentconfig.yaml" created
```

```
INFO OpenShift file "foo-imagestream.yaml" created  
INFO OpenShift file "foo-buildconfig.yaml" created
```

说明：如果使用 `oc create -f` 手动推送 Openshift 工件，则需要确保在构建配置工件之前推送 imagestream 工件，以解决 Openshift 的这个问题：
<https://github.com/openshift/origin/issues/4518>。

kompose up

Kompose 支持通过 `kompose up` 直接将你的“复合的（composed）”应用程序部署到 Kubernetes 或 OpenShift。

Kubernetes

```
kompose --file ./examples/docker-guestbook.yml up
```

We are going to create Kubernetes deployments and services for your Dockerized application.

If you need different kind of resources, use the '`kompose convert`' and '`kubectl create -f`' commands instead.

```
INFO Successfully created service: redis-master  
INFO Successfully created service: redis-slave  
INFO Successfully created service: frontend  
INFO Successfully created deployment: redis-master  
INFO Successfully created deployment: redis-slave  
INFO Successfully created deployment: frontend
```

Your application has been deployed to Kubernetes. You can run '`kubectl get deployment,svc,pods`' for details.

```
kubectl get deployment,svc,pods
```

NAME	DESIRED	CURRENT	UP-TO-DATE
AVAILABLE AGE			
deployment.extensions/frontend	1	1	1
4m			
deployment.extensions/redis-master	1	1	1
4m			
deployment.extensions/redis-slave	1	1	1
4m			

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/frontend	ClusterIP	10.0.174.12	<none>	80/TCP	4m
service/kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	13d
service/redis-master	ClusterIP	10.0.202.43	<none>	6379/TCP	

service/redis-slave	ClusterIP	10.0.1.85	<none>	6379/TCP	4m
NAME READY STATUS RESTARTS AGE					
pod/frontend-2768218532-cs5t5	1/1	Running	0	4m	
pod/redis-master-1432129712-63jn8	1/1	Running	0	4m	
pod/redis-slave-2504961300-nve7b	1/1	Running	0	4m	

注意：

- 你必须有一个运行正常的 Kubernetes 集群，该集群具有预先配置的 kubectl 上下文。
- 此操作仅生成 Deployment 和 Service 对象并将其部署到 Kubernetes。如果需要部署其他不同类型的资源，请使用 kompose convert 和 kubectl create -f 命令。

OpenShift

```
kompose --file ./examples/docker-guestbook.yml --provider openshift up
```

We are going to create OpenShift DeploymentConfigs and Services for your Dockerized application.

If you need different kind of resources, use the 'kompose convert' and 'oc create -f' commands instead.

```
INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created service: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created ImageStream: redis-slave
INFO Successfully created deployment: frontend
INFO Successfully created ImageStream: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created ImageStream: redis-master
```

Your application has been deployed to OpenShift. You can run 'oc get dc,svc,is' for details.

```
oc get dc,svc,is
```

NAME BY	REVISION	DESIRED	CURRENT	TRIGGERED
dc/frontend	0 config,image(frontend:v4)	1	0	
dc/redis-master	0 master:e2e)	1	0	config,image(redis-
dc/redis-slave	0 slave:v1)	1	0	config,image(redis-

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/frontend	172.30.46.64	<none>	80/TCP	8s
svc/redis-master	172.30.144.56	<none>	6379/TCP	8s
svc/redis-slave	172.30.75.245	<none>	6379/TCP	8s
NAME	DOCKER REPO	TAGS	UPDATED	
is/frontend	172.30.12.200:5000/fff/frontend			
is/redis-master	172.30.12.200:5000/fff/redis-master			
is/redis-slave	172.30.12.200:5000/fff/redis-slave	v1		

注意：

- 你必须有一个运行正常的 OpenShift 集群，该集群具有预先配置的 oc 上下文 (oc login)。

kompose down

你一旦将"复合(composed)" 应用部署到 Kubernetes , kompose down 命令将能帮你通过删除 Deployment 和 Service 对象来删除应用。 如果需要删除其他资源，请使用 'kubectl' 命令。

```
kompose --file docker-guestbook.yml down
```

```
INFO Successfully deleted service: redis-master
INFO Successfully deleted deployment: redis-master
INFO Successfully deleted service: redis-slave
INFO Successfully deleted deployment: redis-slave
INFO Successfully deleted service: frontend
INFO Successfully deleted deployment: frontend
```

注意：

- 你必须有一个运行正常的 Kubernetes 集群，该集群具有预先配置的 kubectl 上下文。

构建和推送 Docker 镜像

Kompose 支持构建和推送 Docker 镜像。如果 Docker Compose 文件中使用了 build 关键字，你的镜像将会：

- 使用文档中指定的 image 键自动构建 Docker 镜像
- 使用本地凭据推送到正确的 Docker 仓库

使用 [Docker Compose 文件示例](#)

```
version: "2"
```

```
services
  foo:
```

```
build: "./build"
image: docker.io/foo/bar
```

使用带有 build 键的 kompose up 命令：

```
kompose up
```

```
INFO Build key detected. Attempting to build and push image 'docker.io/foo/bar'
INFO Building image 'docker.io/foo/bar' from directory 'build'
INFO Image 'docker.io/foo/bar' from directory 'build' built successfully
INFO Pushing image 'foo/bar:latest' to registry 'docker.io'
INFO Attempting authentication credentials 'https://index.docker.io/v1/'
INFO Successfully pushed image 'foo/bar:latest' to registry 'docker.io'
INFO We are going to create Kubernetes Deployments, Services and
PersistentVolumeClaims for your Dockerized application. If you need different
kind of resources, use the 'kompose convert' and 'kubectl create -f' commands
instead.
```

```
INFO Deploying application in "default" namespace
INFO Successfully created Service: foo
INFO Successfully created Deployment: foo
```

```
Your application has been deployed to Kubernetes. You can run 'kubectl get
deployment,svc,pods,pvc' for details.
```

要想禁用该功能，或者使用 BuildConfig 中的版本（在 OpenShift 中），可以通过传递 --build (local|build-config|none) 参数来实现。

```
# Disable building/pushing Docker images
kompose up --build none
```

```
# Generate Build Config artifacts for OpenShift
kompose up --provider openshift --build build-config
```

其他转换方式

默认的 kompose 转换会生成 yaml 格式的 Kubernetes [Deployment](#) 和 [Service](#) 对象。你可以选择通过 -j 参数生成 json 格式的对象。你也可以替换生成 [Replication Controllers](#) 对象、[Daemon Sets](#) 或 [Helm](#) charts。

```
kompose convert -j
```

```
INFO Kubernetes file "redis-svc.json" created
INFO Kubernetes file "web-svc.json" created
INFO Kubernetes file "redis-deployment.json" created
INFO Kubernetes file "web-deployment.json" created
```

*-deployment.json 文件中包含 Deployment 对象。

```
kompose convert --replication-controller
```

```
INFO Kubernetes file "redis-svc.yaml" created  
INFO Kubernetes file "web-svc.yaml" created  
INFO Kubernetes file "redis-replicationcontroller.yaml" created  
INFO Kubernetes file "web-replicationcontroller.yaml" created
```

*-replicationcontroller.yaml 文件包含 Replication Controller 对象。如果你想指定副本数（默认为 1），可以使用 --replicas 参数： kompose convert --replication-controller --replicas 3

```
kompose convert --daemon-set
```

```
INFO Kubernetes file "redis-svc.yaml" created  
INFO Kubernetes file "web-svc.yaml" created  
INFO Kubernetes file "redis-daemonset.yaml" created  
INFO Kubernetes file "web-daemonset.yaml" created
```

*-daemonset.yaml 文件包含 Daemon Set 对象。

如果你想生成 [Helm](#) 可用的 Chart，只需简单的执行下面的命令：

```
kompose convert -c
```

```
INFO Kubernetes file "web-svc.yaml" created  
INFO Kubernetes file "redis-svc.yaml" created  
INFO Kubernetes file "web-deployment.yaml" created  
INFO Kubernetes file "redis-deployment.yaml" created  
chart created in "./docker-compose/"
```

```
tree docker-compose/
```

```
docker-compose  
├── Chart.yaml  
├── README.md  
└── templates  
    ├── redis-deployment.yaml  
    ├── redis-svc.yaml  
    ├── web-deployment.yaml  
    └── web-svc.yaml
```

这个 Chart 结构旨在为构建 Helm Chart 提供框架。

标签

kompose 支持 docker-compose.yml 文件中用于 Kompose 的标签，以便在转换时明确定义 Service 的行为。

- kompose.service.type 定义要创建的 Service 类型。例如：

```
version: "2"
services:
  nginx:
    image: nginx
    dockerfile: foobar
    build: ./foobar
    cap_add:
      - ALL
    container_name: foobar
    labels:
      kompose.service.type: nodeport
```

- kompose.service.expose 定义是否允许从集群外部访问 Service。如果该值被设置为 "true"，提供程序将自动设置端点，对于任何其他值，该值将被设置为主机名。如果在 Service 中定义了多个端口，则选择第一个端口作为公开端口。
 - 对于 Kubernetes 驱动程序，创建了一个 Ingress 资源，并且假定已经配置了相应的 Ingress 控制器。
 - 对于 OpenShift 驱动程序，创建一个 route。

例如：

```
version: "2"
services:
  web:
    image: tuna/docker-counter23
    ports:
      - "5000:5000"
    links:
      - redis
    labels:
      kompose.service.expose: "counter.example.com"
  redis:
    image: redis:3.0
    ports:
      - "6379"
```

当前支持的选项有：

键	值
kompose.service.type	nodeport / clusterip / loadbalancer

键	值
kompose.service.expose	true / hostname

说明： kompose.service.type 标签应该只用ports来定义，否则 kompose 会失败。

重启

如果你想创建没有控制器的普通 Pod，可以使用 docker-compose 的 restart 结构来定义它。请参考下表了解 restart 的不同参数。

docker-compose restart	创建的对象	Pod restartPolicy
" "	控制器对象	Always
always	控制器对象	Always
on-failure	Pod	OnFailure
no	Pod	Never

说明： 控制器对象可以是 deployment 或 replicationcontroller 等。

例如，pival Service 将在这里变成 Pod。这个容器的计算值为 pi。

```
version: '2'

services:
  pival
    image: perl
    command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    restart: "on-failure"
```

关于 Deployment Config 的提醒

如果 Docker Compose 文件中为服务声明了卷，Deployment (Kubernetes) 或 DeploymentConfig (OpenShift) 的策略会从 "RollingUpdate" (默认) 变为 "Recreate"。这样做的目的是为了避免服务的多个实例同时访问卷。

如果 Docker Compose 文件中的服务名包含 _ (例如 web_service)，那么将被替换为 -，服务也相应的会重命名(例如 web-service)。Kompose 这样做的原因是 "Kubernetes" 不允许对象名称中包含 _。

请注意，更改服务名称可能会破坏一些 docker-compose 文件。

Docker Compose 版本

Kompose 支持的 Docker Compose 版本包括：1、2 和 3。有限支持 2.1 和 3.2 版本，因为它们还在实验阶段。

所有三个版本的兼容性列表请查看我们的 [转换文档](#)， 文档中列出了所有不兼容的 Docker Compose 关键字。

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 19, 2020 at 12:39 AM PST: [Update translate-compose-kubernetes.md \(8852b8cb8\)](#)

管理 Kubernetes 对象

用声明式和命令式范型与 Kubernetes API 交互。

[使用配置文件对 Kubernetes 对象进行声明式管理](#)

[使用 Kustomize 对 Kubernetes 对象进行声明式管理](#)

[使用指令式命令管理 Kubernetes 对象](#)

[使用配置文件对 Kubernetes 对象进行命令式管理](#)

[使用 kubectl patch 更新 API 对象](#)

使用 kubectl patch 更新 Kubernetes API 对象。做一个策略性的合并 patch 或 JSON 合并 patch。

使用配置文件对 Kubernetes 对象进行声明式管理

你可以通过在一个目录中存储多个对象配置文件、并使用 kubectl apply 来递归地创建和更新对象来创建、更新和删除 Kubernetes 对象。这种方法会保留对现有对象已作出的修改，而不会将这些更改写回到对象配置文件中。kubectl diff 也会给你呈现 apply 将作出的变更的预览。

准备开始

安装 [kubectl](#)。

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.

权衡取舍

kubectl 工具能够支持三种对象管理方式：

- 指令式命令
- 指令式对象配置
- 声明式对象配置

关于每种对象管理的优缺点的讨论，可参见 [Kubernetes 对象管理](#)。

概览

声明式对象管理需要用户对 Kubernetes 对象定义和配置有比较深刻的理解。如果你还没有这方面的知识储备，请先阅读下面的文档：

- [使用指令式命令管理 Kubernetes 对象](#)
- [使用配置文件对 Kubernetes 对象进行指令式管理](#)

以下是本文档中使用的术语的定义：

- **对象配置文件/配置文件**：一个定义 Kubernetes 对象的配置的文件。本主题展示如何将配置文件传递给 kubectl apply。配置文件通常存储于类似 Git 这种源码控制系统中。
- **现时对象配置/现时配置**：由 Kubernetes 集群所观测到的对象的现时配置值。这些配置保存在 Kubernetes 集群存储（通常是 etcd）中。
- **声明式配置写者/声明式写者**：负责更新现时对象的人或者软件组件。本主题中的声明式写者负责改变对象配置文件并执行 kubectl apply 命令以写入变更。

如何创建对象

使用 kubectl apply 来创建指定目录中配置文件所定义的所有对象，除非对应对象已经存在：

```
kubectl apply -f <目录>/
```

此操作会在每个对象上设置 kubectl.kubernetes.io/last-applied-configuration: '...' 注解。注解值中包含了用来创建对象的配置文件的内容。

说明：添加 -R 标志可以递归地处理目录。

下面是一个对象配置文件示例：

[application/simple_deployment.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

执行 kubectl diff 可以打印出将被创建的对象：

```
kubectl diff -f https://k8s.io/examples/application/simple_deployment.yaml
```

说明：

diff 使用[服务器端试运行 \(Server-side Dry-run \)](#) 功能特性；而该功能特性需要在 kube-apiserver 上启用。

由于 diff 操作会使用试运行模式执行服务器端 apply 请求，因此需要为 用户 配置 PATCH、CREATE 和 UPDATE 操作权限。参阅[试运行授权](#) 了解详情。

使用 kubectl apply 来创建对象：

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

使用 kubectl get 打印其现时配置：

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

输出显示注解 kubectl.kubernetes.io/last-applied-configuration 被写入到 现时配置中，并且其内容与配置文件相同：

```

kind: Deployment
metadata:
annotations:
# ...
# This is the json representation of simple_deployment.yaml
# It was written by kubectl apply when the object was created
kubectl.kubernetes.io/last-applied-configuration: |
{"apiVersion":"apps/v1","kind":"Deployment",
 "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
 "spec":{"minReadySeconds":5,"selector":{"matchLabels":
 {"app":nginx}}, "template":{"metadata":{"labels":{"app":nginx}},
 "spec":{"containers":[{"image":nginx:1.14.2,"name":nginx,
 "ports":[{"containerPort":80}]}]}}}
#
spec:
# ...
minReadySeconds: 5
selector:
matchLabels:
# ...
app: nginx
template:
metadata:
# ...
labels:
app: nginx
spec:
containers:
- image: nginx:1.14.2
# ...
name: nginx
ports:
- containerPort: 80
# ...
# ...
# ...
# ...

```

如何更新对象

你也可以使用 `kubectl apply` 来更新某个目录中定义的所有对象，即使那些对象已经存在。这一操作会隐含以下行为：

1. 在现时配置中设置配置文件中出现的字段；
2. 在现时配置中清除配置文件中已删除的字段。

```
kubectl diff -f <目录>/  
kubectl apply -f <目录>/
```

说明： 使用 -R 标志递归处理目录。

下面是一个配置文件示例：

[application/simple_deployment.yaml](#)



```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
spec:  
  selector:  
    matchLabels:  
      app: nginx  
  minReadySeconds: 5  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
      - name: nginx  
        image: nginx:1.14.2  
      ports:  
      - containerPort: 80
```

使用 kubectl apply 来创建对象：

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

说明： 出于演示的目的，上面的命令引用的是单个文件而不是整个目录。

使用 kubectl get 打印现时配置：

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o  
yaml
```

输出显示，注解 kubectl.kubernetes.io/last-applied-configuration 被写入到 现时配置中，并且其取值与配置文件内容相同。

```
kind: Deployment  
metadata:  
  annotations:  
    # ...
```

```

# 此为 simple_deployment.yaml 的 JSON 表示
# 在对象创建时由 kubectl apply 命令写入
kubectl.kubernetes.io/last-applied-configuration: |
  {"apiVersion": "apps/v1", "kind": "Deployment",
   "metadata": {"annotations": {}}, "name": "nginx-deployment", "namespace": "default"},
   "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.14.2", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}
  # ...
spec:
  # ...
minReadySeconds: 5
selector:
  matchLabels:
    # ...
    app: nginx
template:
  metadata:
    # ...
    labels:
      app: nginx
spec:
  containers:
    - image: nginx:1.14.2
      # ...
      name: nginx
  ports:
    - containerPort: 80
      # ...
      # ...
      # ...
# ...

```

通过 kubectl scale 命令直接更新现时配置中的 replicas 字段。这一命令没有使用 kubectl apply :

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

使用 kubectl get 来打印现时配置：

```
kubectl get deployment nginx-deployment -o yaml
```

输出显示，replicas 字段已经被设置为 2，而 last-applied-configuration 注解中并不包含 replicas 字段。

```

apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # 注意注解中并不包含 replicas
    # 这是因为更新并不是通过 kubectl apply 来执行的
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion":"apps/v1","kind":"Deployment",
     "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
     "spec":{"minReadySeconds":5,"selector":{"matchLabels":
     {"app":nginx}},"template":{"metadata":{"labels":{"app":nginx}}},
     "spec":{"containers":[{"image":nginx:1.14.2,"name":nginx,
     "ports":[{"containerPort":80}]}]}}}
    # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.14.2
        # ...
        name: nginx
      ports:
        - containerPort: 80
        # ...

```

现在更新 simple_deployment.yaml 配置文件，将镜像文件从 nginx:1.14.2 更改为 nginx:1.16.1，同时删除minReadySeconds 字段：

[application/update_deployment.yaml](#)



```

apiVersion: apps/v1
kind: Deployment

```

```
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1 # update the image
      ports:
        - containerPort: 80
```

应用对配置文件所作更改：

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

使用 kubectl get 打印现时配置：

```
kubectl get -f https://k8s.io/examples/application/update_deployment.yaml -o yaml
```

输出显示现时配置中发生了以下更改：

- 字段 replicas 保留了 kubectl scale 命令所设置的值：2；之所以该字段被保留是因为配置文件中并没有设置 replicas。
- 字段 image 的内容已经从 nginx:1.14.2 更改为 nginx:1.16.1。
- 注解 last-applied-configuration 内容被更改为新的镜像名称。
- 字段 minReadySeconds 被移除。
- 注解 last-applied-configuration 中不再包含 minReadySeconds 字段。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # 注解中包含更新后的镜像 nginx 1.16.1
    # 但是其中并不包含更改后的 replicas 值 2
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion":"apps/v1","kind":"Deployment",
     "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
```

```
  "spec": {"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.16.1", "name": "nginx", "ports": [{"containerPort": 80}]}]}}, "# ..."}, "spec": {"replicas": 2, "# 由 `kubectl scale` 设置，被 `kubectl apply` 命令忽略", "# minReadySeconds 被 `kubectl apply` 清除", "# ..."}, "selector": {"matchLabels": {"# ..."}, "app": "nginx"}, "template": {"metadata": {"# ..."}, "labels": {"app": "nginx"}, "spec": {"containers": [{"image": "nginx:1.16.1", "# 由 `kubectl apply` 设置", "# ..."}, {"name": "nginx"}, {"ports": [{"containerPort": 80, "# ..."}, {"# ..."}, {"# ..."}]}, "# ..."]}}}, "# ..."
```

警告：将 `kubectl apply` 与指令式对象配置命令 `kubectl create` 或 `kubectl replace` 混合使用是不受支持的。这是因为 `create` 和 `replace` 命令都不会保留 `kubectl apply` 用来计算更新内容所使用的 `kubectl.kubernetes.io/last-applied-configuration` 注解值。

如何删除对象

有两种方法来删除 `kubectl apply` 管理的对象。

建议操作：`kubectl delete -f <文件名>`

使用指令式命令来手动删除对象是建议的方法，因为这种方法更为明确地给出了要删除的内容是什么，且不容易造成用户不小心删除了其他对象的情况。

```
kubectl delete -f <文件名>
```

替代方式 : kubectl apply -f <目录名称/> --prune -l your=label

只有在充分理解此命令背后含义的情况下才建议这样操作。

警告 : kubectl apply --prune 命令本身仍处于 Alpha 状态，在后续发布版本中可能会引入一些向后不兼容的变化。

警告 : 在使用此命令时必须小心，这样才不会无意中删除不想删除的对象。

作为 kubectl delete 操作的替代方式，你可以在目录中对象配置文件被删除之后，使用 kubectl apply 来辨识要删除的对象。带 --prune 标志的 apply 命令会首先查询 API 服务器，获得与某组标签相匹配的对象列表，之后将返回的现时对象配置与目录中的对象配置文件相比较。如果某对象在查询中被匹配到，但在目录中没有文件与其相对应，并且其中还包含 last-applied-configuration 注解，则该对象会被删除。

```
kubectl apply -f <directory/> --prune -l <labels>
```

警告 : 带剪裁 (prune) 行为的 apply 操作应在包含对象配置文件的目录的根目录运行。如果在其子目录中运行，可能导致对象被不小心删除。因为某些对象可能与 -l <标签> 的标签选择算符匹配，但其配置文件不在当前子目录下。

如何查看对象

你可以使用 kubectl get 并指定 -o yaml 选项来查看现时对象的配置：

```
kubectl get -f <文件名 | URL> -o yaml
```

apply 操作是如何计算配置差异并合并变更的？

注意 : patch 是一种更新操作，其作用域为对象的一些特定字段而不是整个对象。这使得你可以更新对象的特定字段集合而不必先要读回对象。

kubectl apply 更新对象的现时配置，它是通过向 API 服务器发送一个 patch 请求来执行更新动作的。所提交的补丁中定义了对现时对象配置中特定字段的更新。kubectl apply 命令会使用当前的配置文件、现时配置以及现时配置中保存的 last-applied-configuration 注解内容来计算补丁更新内容。

合并补丁计算

kubectl apply 命令将配置文件的内容写入到 kubectl.kubernetes.io/last-applied-configuration 注解中。这些内容用来识别配置文件中已经移除的、因而也需要从现时配置中删除的字段。用来计算要删除或设置哪些字段的步骤如下：

1. 计算要删除的字段，即在 last-applied-configuration 中存在但在配置文件中不再存在的字段。
2. 计算要添加或设置的字段，即在配置文件中存在但其取值与现时配置不同的字段。

下面是一个例子。假定此文件是某 Deployment 对象的配置文件：

[application/update_deployment.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.16.1 # update the image
        ports:
          - containerPort: 80
```

同时假定同一 Deployment 对象的现时配置如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
       "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
       "spec":{"minReadySeconds":5,"selector":{"matchLabels":
       {"app":nginx}}, "template":{"metadata":{"labels":{"app":"nginx"}},
       "spec":{"containers":[{"image":"nginx:1.14.2","name":"nginx",
       "ports":[{"containerPort":80}]}]}}}
    # ...
spec:
  replicas: 2
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
```

```

# ...
app: nginx
template:
  metadata:
    # ...
  labels:
    app: nginx
spec:
  containers:
    - image: nginx:1.14.2
      # ...
      name: nginx
      ports:
        - containerPort: 80
      # ...

```

下面是 kubectl apply 将执行的合并计算：

1. 通过读取 last-applied-configuration 并将其与配置文件中的值相比较，计算要删除的字段。对于本地对象配置文件中显式设置为空的字段，清除其在现时配置中的设置，无论这些字段是否出现在 last-applied-configuration 中。在此例中，minReadySeconds 出现在 last-applied-configuration 注解中，但并不存在于配置文件中。**动作**：从现时配置中删除 minReadySeconds 字段。
2. 通过读取配置文件中的值并将其与现时配置相比较，计算要设置的字段。在这个例子中，配置文件中的 image 值与现时配置中的 image 不匹配。**动作**：设置现时配置中的 image 值。
3. 设置 last-applied-configuration 注解的内容，使之与配置文件匹配。
4. 将第 1、2、3 步骤得出的结果合并，构成向 API 服务器发送的补丁请求内容。

下面是此合并操作之后形成的现时配置：

```

apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # 注解中包含更新后的 image , nginx 1.11.9,
    # 但不包含更新后的 replicas
  kubectl.kubernetes.io/last-applied-configuration: |
    {"apiVersion":"apps/v1","kind":"Deployment",
     "metadata":{"annotations":{},"name":"nginx-
deployment","namespace":"default"},
     "spec":{"selector":{"matchLabels":{"app":nginx}}, "template":{"metadata":{},
       "labels":{"app":nginx}},
     "spec":{"containers":[{"image":nginx:1.16.1,"name":nginx},
       "ports":[{"containerPort":80}]]}}}
    # ...

```

```
spec:  
  selector:  
    matchLabels:  
      # ...  
      app: nginx  
  replicas: 2  
  # minReadySeconds 此字段被清除  
  # ...  
  template:  
    metadata:  
      # ...  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - image: nginx:1.16.1  
          # ...  
          name: nginx  
      ports:  
        - containerPort: 80  
          # ...  
          # ...  
          # ...  
        # ...  
      # ...
```

不同类型字段的合并方式

配置文件中的特定字段与现时配置合并时，合并方式取决于字段类型。字段类型有几种：

- **基本类型**：字段类型为 string、integer 或 boolean 之一。例如：image 和 replicas 字段都是基本类型字段。

动作：替换。

- **map**：也称作 *object*。类型为 map 或包含子域的复杂结构。例如，labels、annotations、spec 和 metadata 都是 map。

动作：合并元素或子字段。

- **list**：包含元素列表的字段，其中每个元素可以是基本类型或 map。例如，containers、ports 和 args 都是 list。

动作：不一定。

当 kubectl apply 更新某个 map 或 list 字段时，它通常不会替换整个字段，而是会更新其中的各个子元素。例如，当合并 Deployment 的 spec 时，kubectl 并不会将其整个替换掉。相反，实际操作会是对 replicas 这类 spec 的子字段来执行比较和更新。

合并对基本类型字段的更新

基本类型字段会被替换或清除。

说明：- 表示的是"不适用"，因为指定数值未被使用。

字段在对象配置文件中	字段在现时对象配置中	字段在 last-applied-configuration 中	动作
是	是	-	将配置文件中值设置到现时配置上。
是	否	-	将配置文件中值设置到现时配置上。
否	-	是	从现时配置中移除。
否	-	否	什么也不做。保持现时值。

合并对 map 字段的变更

用来表示映射的字段在合并时会逐个子字段或元素地比较：

说明：- 表示的是"不适用"，因为指定数值未被使用。

键存在于对象配置文件中	键存在于现时对象配置中	键存在于 last-applied-configuration 中	动作
是	是	-	比较子域取值。
是	否	-	将现时配置设置为本地配置值。
否	-	是	从现时配置中删除键。
否	-	否	什么也不做，保留现时值。

合并 list 类型字段的变更

对 list 类型字段的变更合并会使用以下三种策略之一：

- 如果 list 所有元素都是基本类型则替换整个 list。
- 如果 list 中元素是复合结构则逐个元素执行合并操作。
- 合并基本类型元素构成的 list。

策略的选择是基于各个字段做出的。

如果 list 中元素都是基本类型则替换整个 list

将整个 list 视为一个基本类型字段。或者整个替换或者整个删除。此操作会保持 list 中元素顺序不变

示例：使用 `kubectl apply` 来更新 Pod 中 Container 的 `args` 字段。此操作会将现时配置中的 `args` 值设为配置文件中的值。所有之前添加到现时配置中的 `args` 元素都会丢失。配置文件中的 `args` 元素的顺序在被添加到现时配置中时保持不变。

```
# last-applied-configuration 值
args: ["a", "b"]
```

```
# 配置文件值
args: ["a", "c"]
```

```
# 现时配置
args: ["a", "b", "d"]
```

```
# 合并结果
args: ["a", "c"]
```

解释：合并操作将配置文件中的值当做新的 list 值。

如果 list 中元素为复合类型则逐个执行合并

此操作将 list 视为 map，并将每个元素中的特定字段当做其主键。逐个元素地执行添加、删除或更新操作。结果顺序无法得到保证。

此合并策略会使用每个字段上的一个名为 `patchMergeKey` 的特殊标签。Kubernetes 源代码中为每个字段定义了 `patchMergeKey`：[types.go](#) 当合并由 map 组成的 list 时，给定元素中被设置为 `patchMergeKey` 的字段会被当做该元素的 map 键值来使用。

例如：使用 `kubectl apply` 来更新 Pod 规约中的 `containers` 字段。此操作会将 `containers` 列表视作一个映射来执行合并，每个元素的主键为 `name`。

```
# last-applied-configuration 值
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-a # 键 nginx-helper-a 会被删除
  image: helper:1.3
- name: nginx-helper-b # 键 nginx-helper-b 会被保留
  image: helper:1.3
```

```
# 配置文件值
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-b
  image: helper:1.3
- name: nginx-helper-c # 键 nginx-helper-c 会被添加
```

```

image: helper:1.3

# 现时配置
containers:
- name: nginx
  image: nginx:1.16
- name: nginx-helper-a
  image: helper:1.3
- name: nginx-helper-b
  image: helper:1.3
  args: ["run"]      # 字段会被保留
- name: nginx-helper-d # 键 nginx-helper-d 会被保留
  image: helper:1.3

# 合并结果
containers:
- name: nginx
  image: nginx:1.16
  # 元素 nginx-helper-a 被删除
- name: nginx-helper-b
  image: helper:1.3
  args: ["run"]      # 字段被保留
- name: nginx-helper-c # 新增元素
  image: helper:1.3
- name: nginx-helper-d # 此元素被忽略 ( 保留 )
  image: helper:1.3

```

解释：

- 名为 "nginx-helper-a" 的容器被删除，因为配置文件中不存在同名的容器。
- 名为 "nginx-helper-b" 的容器的现时配置中的 args 被保留。 kubectl apply 能够辨识出现时配置中的容器 "nginx-helper-b" 与配置文件中的容器 "nginx-helper-b" 相同，即使它们的字段值有些不同（配置文件中未给定 args 值）。这是因为 patchMergeKey 字段（ name ）的值在两个版本中都一样。
- 名为 "nginx-helper-c" 的容器是新增的，因为在配置文件中的这个容器尚不存在于现时配置中。
- 名为 "nginx-helper-d" 的容器被保留下，因为在 last-applied-configuration 中没有与之同名的元素。

合并基本类型元素 list

在 Kubernetes 1.5 中，尚不支持对由基本类型元素构成的 list 进行合并。

说明：选择上述哪种策略是由源码中给定字段的 patchStrategy 标记来控制的：[types.go](#) 如果 list 类型字段未设置 patchStrategy，则整个 list 会被替换掉。

默认字段值

API 服务器会在对象创建时其中某些字段未设置的情况下在现时配置中为其设置默认值。

下面是一个 Deployment 的配置文件。文件未设置 strategy :

[application/simple_deployment.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

使用 kubectl apply 创建对象 :

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

使用 kubectl get 打印现时配置 :

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

输出显示 API 在现时配置中为某些字段设置了默认值。这些字段在配置文件中并未设置。

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
```

```

app: nginx
minReadySeconds: 5
replicas: 1      # API 服务器所设默认值
strategy:
  rollingUpdate:    # API 服务器基于 strategy.type 所设默认值
    maxSurge: 1
    maxUnavailable: 1
  type: RollingUpdate # API 服务器所设默认值
template:
  metadata:
    creationTimestamp: null
    labels:
      app: nginx
spec:
  containers:
    - image: nginx:1.14.2
      imagePullPolicy: IfNotPresent # API 服务器所设默认值
      name: nginx
      ports:
        - containerPort: 80
          protocol: TCP      # API 服务器所设默认值
      resources: {}      # API 服务器所设默认值
      terminationMessagePath: /dev/termination-log # API 服务器所设默认值
      dnsPolicy: ClusterFirst # API 服务器所设默认值
      restartPolicy: Always # API 服务器所设默认值
      securityContext: {} # API 服务器所设默认值
      terminationGracePeriodSeconds: 30 # API 服务器所设默认值
# ...

```

在补丁请求中，已经设置了默认值的字段不会被重新设回其默认值，除非 在补丁请求中显式地要求清除。对于默认值取决于其他字段的某些字段而言，这可能会引发一些意想不到的行为。当所依赖的其他字段后来发生改变时，基于它们所设置的默认值只能在显式执行清除操作时才会被更新。

为此，建议在配置文件中为服务器设置默认值的字段显式提供定义，即使所 给的定义与服务器端默认值设定相同。这样可以使得辨识无法被服务器重新 基于默认值来设置的冲突字段变得容易。

示例：

```

# last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
spec:

```

```
containers:
- name: nginx
image: nginx:1.14.2
ports:
- containerPort: 80

# 配置文件
spec:
strategy:
  type: Recreate # 更新的值
template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
- name: nginx
image: nginx:1.14.2
ports:
- containerPort: 80

# 现时配置
spec:
strategy:
  type: RollingUpdate # 默认设置的值
  rollingUpdate: # 基于 type 设置的默认值
    maxSurge : 1
    maxUnavailable: 1
template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
- name: nginx
image: nginx:1.14.2
ports:
- containerPort: 80

# 合并后的结果 - 出错 !
spec:
strategy:
  type: Recreate # 更新的值 : 与 rollingUpdate 不兼容
  rollingUpdate: # 默认设置的值 : 与 "type: Recreate" 冲突
    maxSurge : 1
    maxUnavailable: 1
```

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

解释：

1. 用户创建 Deployment，未设置 strategy.type。
2. 服务器为 strategy.type 设置默认值 RollingUpdate，并为 strategy.rollingUpdate 设置默认值。
3. 用户改变 strategy.type 为 Recreate。字段 strategy.rollingUpdate 仍会取其默认设置值，尽管服务器期望该字段被清除。如果 strategy.rollingUpdate 值最初于配置文件中定义，则它们需要被清除。这一点就更明确一些。
4. apply 操作失败，因为 strategy.rollingUpdate 未被清除。strategy.rollingUpdate 在 strategy.type 为 Recreate 不可被设定。

建议：以下字段应该在对象配置文件中显式定义：

- 如 Deployment、StatefulSet、Job、DaemonSet、ReplicaSet 和 ReplicationController 这类负载的选择算符和 PodTemplate 标签
- Deployment 的上线策略

如何清除服务器端按默认值设置的字段或者被其他写者设置的字段

没有出现在配置文件中的字段可以通过将其值设置为 null 并应用配置文件来清除。对于由服务器按默认值设置的字段，清除操作会触发重新为字段设置新的默认值。

如何将字段的属主在配置文件和直接指令式写者之间切换

更改某个对象字段时，应该采用下面的方法：

- 使用 kubectl apply。
- 直接写入到现时配置，但不更改配置文件本身，例如使用 kubectl scale。

将属主从直接指令式写者更改为配置文件

将字段添加到配置文件。针对该字段，不再直接执行对现时配置的修改。修改均通过 kubectl apply 来执行。

将属主从配置文件改为直接指令式写者

在 Kubernetes 1.5 中，将字段的属主从配置文件切换到某指令式写者需要手动执行以下步骤：

- 从配置文件中删除该字段；
- 将字段从现时对象的 kubectl.kubernetes.io/last-applied-configuration 注解中删除。

更改管理方法

Kubernetes 对象在同一时刻应该只用一种方法来管理。从一种方法切换到另一种方法是可能的，但这一切换是一个手动过程。

说明：在声明式管理方法中使用指令式命令来删除对象是可以的。

从指令式命令管理切换到声明式对象配置

从指令式命令管理切换到声明式对象配置管理的切换包含以下几个手动步骤：

1. 将现时对象导出到本地配置文件：

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

2. 手动移除配置文件中的 status 字段。

说明：这一步骤是可选的，因为 kubectl apply 并不会更新 status 字段，即便 配置文件中包含 status 字段。

3. 设置对象上的 kubectl.kubernetes.io/last-applied-configuration 注解：

```
kubectl replace --save-config -f <kind>_<name>.yaml
```

4. 更改过程，使用 kubectl apply 专门管理对象。

从指令式对象配置切换到声明式对象配置

1. 在对象上设置 kubectl.kubernetes.io/last-applied-configuration 注解：

```
kubectl replace -save-config -f <kind>_<name>.yaml
```

2. 自此排他性地使用 kubectl apply 来管理对象。

定义控制器选择算符和 PodTemplate 标签

警告：强烈不建议更改控制器上的选择算符。

建议的方法是定义一个不可变更的 PodTemplate 标签，仅用于控制器选择算符且 不包含其他语义性的含义。

示例：

```
selector:  
  matchLabels:  
    controller-selector: "apps/v1/deployment/nginx"  
template:  
  metadata:  
    labels:  
      controller-selector: "apps/v1/deployment/nginx"
```

接下来

- [使用指令式命令管理 Kubernetes 对象](#)
- [使用配置文件对 Kubernetes 对象执行指令式管理](#)
- [Kubectl 命令参考](#)
- [Kubernetes API 参考](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 14, 2020 at 2:50 PM PST: [Update declarative-config.md \(4547d62ed\)](#)

使用 Kustomize 对 Kubernetes 对象进行声明式管理

[Kustomize](#) 是一个独立的工具，用来通过 [kustomization 文件](#) 定制 Kubernetes 对象。

从 1.14 版本开始，kubectl 也开始支持使用 kustomization 文件来管理 Kubernetes 对象。要查看包含 kustomization 文件的目录中的资源，执行下面的命令：

```
kubectl kustomize <kustomization_directory>
```

要应用这些资源，使用参数 --kustomize 或 -k 标志来执行 kubectl apply：

```
kubectl apply -k <kustomization_directory>
```

准备开始

安装 [kubectl](#).

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`.

Kustomize 概述

Kustomize 是一个用来定制 Kubernetes 配置的工具。它提供以下功能特性来管理 应用配置文件：

- 从其他来源生成资源
- 为资源设置贯穿性 (Cross-Cutting) 字段
- 组织和定制资源集合

生成资源

ConfigMap 和 Secret 包含其他 Kubernetes 对象（如 Pod）所需要的配置或敏感数据。 ConfigMap 或 Secret 中数据的来源往往是集群外部，例如某个 `.properties` 文件或者 SSH 密钥文件。 Kustomize 提供 `secretGenerator` 和 `configMapGenerator`，可以基于文件或字面 值来生成 Secret 和 ConfigMap。

configMapGenerator

要基于文件来生成 ConfigMap，可以在 configMapGenerator 的 `files` 列表中添加表项。 下面是一个根据 `.properties` 文件中的数据条目来生成 ConfigMap 的示例：

```
# 生成一个 application.properties 文件
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
  - application.properties
EOF
```

所生成的 ConfigMap 可以使用下面的命令来检查：

```
kubectl kustomize ./
```

所生成的 ConfigMap 为：

```
apiVersion: v1
data:
  application.properties: |
    FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-8mbdf7882g
```

ConfigMap 也可基于字面的键值偶对来生成。要基于键值偶对来生成 ConfigMap，在 configMapGenerator 的 literals 列表中添加表项。下面是一个例子，展示如何使用键值偶对中的数据条目来生成 ConfigMap 对象：

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-2
  literals:
  - FOO=Bar
EOF
```

可以用下面的命令检查所生成的 ConfigMap：

```
kubectl kustomize ./
```

所生成的 ConfigMap 为：

```
apiVersion: v1
data:
  FOO: Bar
kind: ConfigMap
metadata:
  name: example-configmap-2-g2hdhfc6tk
```

secretGenerator

你可以基于文件或者键值偶对来生成 Secret。要使用文件内容来生成 Secret，在 secretGenerator 下面的 files 列表中添加表项。下面是一个根据文件中数据来生成 Secret 对象的示例：

```
# 创建一个 password.txt 文件
cat <<EOF >./password.txt
username=admin
password=secret
EOF
```

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-1
  files:
  - password.txt
EOF
```

所生成的 Secret 如下：

```
apiVersion: v1
data:
  password.txt: dXNlcm5hbWU9YWRtaW4KcGFzc3dvcmQ9c2VjcmV0Cg==
kind: Secret
metadata:
  name: example-secret-1-t2kt65hgtb
type: Opaque
```

要基于键值偶对字面值生成 Secret，先要在 secretGenerator 的 literals 列表中添加表项。下面是基于键值偶对中数据条目来生成 Secret 的示例：

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-2
  literals:
  - username=admin
  - password=secret
EOF
```

所生成的 Secret 如下：

```
apiVersion: v1
data:
  password: c2VjcmV0
  username: YWRtaW4=
kind: Secret
metadata:
  name: example-secret-2-t52t6g96d8
type: Opaque
```

generatorOptions

所生成的 ConfigMap 和 Secret 都会包含内容哈希值后缀。这是为了确保内容发生变化时，所生成的是新的 ConfigMap 或 Secret。要禁止自动添加后缀的行为，用户可以使用 generatorOptions。除此以外，为生成的 ConfigMap 和 Secret 指定贯穿性选项也是可以的。

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
```

```
- name: example-configmap-3
literals:
- FOO=Bar
generatorOptions:
 disableNameSuffixHash: true
labels:
 type: generated
annotations:
 note: generated
EOF
```

运行 kubectl kustomize ./ 来查看所生成的 ConfigMap :

```
apiVersion: v1
data:
 FOO: Bar
kind: ConfigMap
metadata:
 annotations:
 note: generated
labels:
 type: generated
name: example-configmap-3
```

设置贯穿性字段

在项目中为所有 Kubernetes 对象设置贯穿性字段是一种常见操作。 贯穿性字段的一些使用场景如下：

- 为所有资源设置相同的名字空间
- 为所有对象添加相同的前缀或后缀
- 为对象添加相同的标签集合
- 为对象添加相同的注解集合

下面是一个例子：

```
# 创建一个 deployment.yaml
cat <<EOF >./deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
labels:
 app: nginx
spec:
 selector:
 matchLabels:
```

```
    app: nginx
template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
EOF
```

```
cat <<EOF >./kustomization.yaml
namespace: my-namespace
namePrefix: dev-
nameSuffix: "-001"
commonLabels:
  app: bingo
commonAnnotations:
  oncallPager: 800-555-1212
resources:
- deployment.yaml
EOF
```

执行 kubectl kustomize ./ 查看这些字段都被设置到 Deployment 资源上：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    oncallPager: 800-555-1212
  labels:
    app: bingo
  name: dev-nginx-deployment-001
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: bingo
  template:
    metadata:
      annotations:
        oncallPager: 800-555-1212
      labels:
        app: bingo
    spec:
      containers:
```

```
- image: nginx
  name: nginx
```

组织和定制资源

一种常见的做法是在项目中构造资源集合并将其放到同一个文件或目录中管理。Kustomize 提供基于不同文件来组织资源并向其应用补丁或者其他定制的能力。

组织

Kustomize 支持组合不同的资源。kustomization.yaml 文件的 resources 字段 定义配置中要包含的资源列表。你可以将 resources 列表中的路径设置为资源配置文件的路径。下面是由 Deployment 和 Service 构成的 NGINX 应用的示例：

```
# 创建 deployment.yaml 文件
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
        ports:
          - containerPort: 80
EOF
```

```
# 创建 service.yaml 文件
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
```

```
ports:
- port: 80
  protocol: TCP
selector:
  run: my-nginx
EOF

# 创建 kustomization.yaml 来组织以上两个资源
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

kubectl kustomize ./ 所得到的资源中既包含 Deployment 也包含 Service 对象。

定制

补丁文件 (Patches) 可以用来对资源执行不同的定制。 Kustomize 通过 patchesStrategicMerge 和 patchesJson6902 支持不同的打补丁机制。 patchesStrategicMerge 的内容是一个文件路径的列表，其中每个文件都应可解析为 [策略性合并补丁 \(Strategic Merge Patch \)](#)。 补丁文件中的名称必须与已经加载的资源的名称匹配。 建议构造规模较小的、仅做一件事情的补丁。 例如，构造一个补丁来增加 Deployment 的副本个数；构造另外一个补丁来设置内存限制。

```
# 创建 deployment.yaml 文件
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
      ports:
        - containerPort: 80
EOF
```

```
EOF
```

```
# 生成一个补丁 increase_replicas.yaml
cat <<EOF > increase_replicas.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
EOF
```

```
# 生成另一个补丁 set_memory.yaml
cat <<EOF > set_memory.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
        - name: my-nginx
          resources:
            limits:
              memory: 512Mi
EOF
```

```
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
patchesStrategicMerge:
- increase_replicas.yaml
- set_memory.yaml
EOF
```

执行 kubectl kustomize ./ 来查看 Deployment :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
run: my-nginx
template:
  metadata:
    labels:
      run: my-nginx
spec:
  containers:
    - image: nginx
      name: my-nginx
      ports:
        - containerPort: 80
      resources:
        limits:
          memory: 512Mi
```

并非所有资源或者字段都支持策略性合并补丁。为了支持对任何资源的任何字段进行修改，Kustomize 提供通过 patchesJson6902 来应用 [JSON 补丁](#) 的能力。为了给 JSON 补丁找到正确的资源，需要在 kustomization.yaml 文件中指定资源的组（group）、版本（version）、类别（kind）和名称（name）。例如，为某 Deployment 对象增加副本个数的操作也可以通过 patchesJson6902 来完成：

```
# 创建一个 deployment.yaml 文件
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF
```

```
# 创建一个 JSON 补丁文件
cat <<EOF > patch.yaml
```

```
- op: replace
  path: /spec/replicas
  value: 3
EOF

# 创建一个 kustomization.yaml
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml

patchesJson6902:
- target:
  group: apps
  version: v1
  kind: Deployment
  name: my-nginx
  path: patch.yaml
EOF
```

执行 `kubectl kustomize ./` 以查看 `replicas` 字段被更新：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: nginx
          name: my-nginx
        ports:
          - containerPort: 80
```

除了补丁之外，Kustomize 还提供定制容器镜像或者将其他对象的字段值注入到容器中的能力，并且不需要创建补丁。例如，你可以通过在 `kustomization.yaml` 文件的 `images` 字段设置新的镜像来更改容器中使用的镜像。

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF
```

```
cat <<EOF >/kustomization.yaml
resources:
- deployment.yaml
images:
- name: nginx
  newName: my.image.registry/nginx
  newTag: 1.4.0
EOF
```

执行 kubectl kustomize ./ 以查看所使用的镜像已被更新：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
```

```
- image: my.image.registry/nginx:1.4.0
  name: my-nginx
  ports:
    - containerPort: 80
```

有些时候，Pod 中运行的应用可能需要使用来自其他对象的配置值。例如，某 Deployment 对象的 Pod 需要从环境变量或命令行参数中读取 Service 的名称。由于在 kustomization.yaml 文件中添加 namePrefix 或 nameSuffix 时 Service 名称可能发生变化，建议不要在命令参数中硬编码 Service 名称。对于这种使用场景，Kustomize 可以通过 vars 将 Service 名称注入到容器中。

```
# 创建一个 deployment.yaml 文件
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          command: ["start", "--host", "$(MY_SERVICE_NAME)"]
EOF
```

```
# 创建一个 service.yaml 文件
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
```

```
run: my-nginx
EOF

cat <<EOF >./kustomization.yaml
namePrefix: dev-
nameSuffix: "-001"

resources:
- deployment.yaml
- service.yaml

vars:
- name: MY_SERVICE_NAME
  objref:
    kind: Service
    name: my-nginx
    apiVersion: v1
EOF
```

执行 kubectl kustomize ./ 以查看注入到容器中的 Service 名称是 dev-my-nginx-001

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dev-my-nginx-001
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - command:
            - start
            - --host
            - dev-my-nginx-001
          image: nginx
          name: my-nginx
```

基准 (Bases) 与覆盖 (Overlays)

Kustomize 中有 **基准 (bases)** 和 **覆盖 (overlays)** 的概念区分。 **基准** 是包含 kustomization.yaml 文件的一个目录，其中包含一组资源及其相关的定制。 基准可以是本地目录或者来自远程仓库的目录，只要其中存在 kustomization.yaml 文件即可。 **覆盖** 也是一个目录，其中包含将其他 kustomization 目录当做 bases 来引用的 kustomization.yaml 文件。 **基准**不了解覆盖的存在，且可被多个覆盖所使用。 覆盖则可以有多个基准，且可针对所有基准中的资源执行组织操作，还可以在其上执行定制。

```
# 创建一个包含基准的目录
mkdir base
# 创建 base/deployment.yaml
cat <<EOF > base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
EOF
```

```
# 创建 base/service.yaml 文件
cat <<EOF > base/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
```

```
EOF
```

```
# 创建 base/kustomization.yaml
cat <<EOF > base/kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF
```

此基准可在多个覆盖中使用。你可以在不同的覆盖中添加不同的 namePrefix 或 其他贯穿性字段。下面是两个使用同一基准的覆盖：

```
mkdir dev
cat <<EOF > dev/kustomization.yaml
bases:
- ../base
namePrefix: dev-
EOF
```

```
mkdir prod
cat <<EOF > prod/kustomization.yaml
bases:
- ../base
namePrefix: prod-
EOF
```

如何使用 Kustomize 来应用、查看和删除对象

在 kubectl 命令中使用 --kustomize 或 -k 参数来识别被 kustomization.yaml 所管理的资源。注意 -k 要指向一个 kustomization 目录。例如：

```
kubectl apply -k <kustomization 目录>/
```

假定使用下面的 kustomization.yaml，

```
# 创建 deployment.yaml 文件
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
```

```
metadata:  
  labels:  
    run: my-nginx  
spec:  
  containers:  
    - name: my-nginx  
      image: nginx  
      ports:  
        - containerPort: 80  
EOF
```

```
# 创建 kustomization.yaml  
cat <<EOF >./kustomization.yaml  
namePrefix: dev-  
commonLabels:  
  app: my-nginx  
resources:  
- deployment.yaml  
EOF
```

执行下面的命令来应用 Deployment 对象 dev-my-nginx：

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

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

运行下面的命令之一来查看 Deployment 对象 dev-my-nginx：

```
kubectl get -k ./
```

```
kubectl describe -k ./
```

执行下面的命令来比较 Deployment 对象 dev-my-nginx 与清单被应用之后 集群将处于的状态：

```
kubectl diff -k ./
```

执行下面的命令删除 Deployment 对象 dev-my-nginx：

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

```
deployment.apps "dev-my-nginx" deleted
```

Kustomize 功能特性列表

字段	类型	解释
namespace	string	为所有资源添加名字空间

字段	类型	解释
namePrefix	string	此字段的值将被添加到所有资源名称前面
nameSuffix	string	此字段的值将被添加到所有资源名称后面
commonLabels	map[string]string	要添加到所有资源和选择算符的标签
commonAnnotations	map[string]string	要添加到所有资源的注解
resources	[]string	列表中的每个条目都必须能够解析为现有的资源配置文件
configmapGenerator	[] ConfigMapArgs	列表中的每个条目都会生成一个 ConfigMap
secretGenerator	[] SecretArgs	列表中的每个条目都会生成一个 Secret
generatorOptions	GeneratorOptions	更改所有 ConfigMap 和 Secret 生成器的行为
bases	[]string	列表中每个条目都应能解析为一个包含 kustomization.yaml 文件的目录
patchesStrategicMerge	[]string	列表中每个条目都能解析为某 Kubernetes 对象的策略性合并补丁
patchesJson6902	[] Json6902	列表中每个条目都能解析为一个 Kubernetes 对象和一个 JSON 补丁
vars	[] Var	每个条目用来从某资源的字段来析取文字
images	[] Image	每个条目都用来更改镜像的名称、标记与/或摘要，不必生成补丁
configurations	[]string	列表中每个条目都应能解析为一个包含 Kustomize 转换器配置 的文件
crds	[]string	列表中每个条目都应能够解析为 Kubernetes 类别的 OpenAPI 定义文件

接下来

- [Kustomize](#)
- [Kubectl Book](#)
- [Kubectl 命令参考](#)
- [Kubernetes API 参考](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 11, 2021 at 9:31 AM PST: [Fix typo in kustomization example \(9845d4e7a\)](#)

使用指令式命令管理 Kubernetes 对象

使用构建在 kubectl 命令行工具中的指令式命令可以直接快速创建、更新和删除 Kubernetes 对象。本文档解释这些命令的组织方式以及如何使用它们来管理现时对象。

准备开始

安装[kubectl](#)。

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`.

权衡取舍

kubectl 工具能够支持三种对象管理方式：

- 指令式命令
- 指令式对象配置
- 声明式对象配置

关于每种对象管理的优缺点的讨论，可参见 [Kubernetes 对象管理](#)。

如何创建对象

kubectl 工具支持动词驱动的命令，用来创建一些最常见的对象类别。命令的名称设计使得不熟悉 Kubernetes 对象类型的用户也能做出判断。

- `run`：创建一个新的 Pod 来运行一个容器。
- `expose`：创建一个新的 Service 对象为若干 Pod 提供流量负载均衡。
- `autoscale`：创建一个新的 Autoscaler 对象来自动对某控制器（如 Deployment）执行水平扩缩。

kubectl 命令也支持一些对象类型驱动的创建命令。这些命令可以支持更多的对象类别，并且在其动机上体现得更为明显，不过要求 用户了解它们所要创建的对象的类别。

- `create <对象类别> [<子类别>] <实例名称>`

某些对象类别拥有自己的子类别，可以在 create 命令中设置。例如，Service 对象有 ClusterIP、LoadBalancer 和 NodePort 三种子类别。下面是一个创建 NodePort 子类别的 Service 的示例：

```
kubectl create service nodeport <服务名称>
```

在前述示例中，create service nodeport 命令也称作 create service 命令的子命令。可以使用 -h 标志找到一个子命令所支持的参数和标志。

```
kubectl create service nodeport -h
```

如何更新对象

kubectl 命令也支持一些动词驱动的命令，用来执行一些常见的更新操作。这些命令的设计是为了让一些不了解 Kubernetes 对象的用户也能执行更新操作，但不需要了解哪些字段必须设置：

- scale：对某控制器进行水平扩缩以便通过更新控制器的副本个数来添加或删除 Pod。
- annotate：为对象添加或删除注解。
- label：为对象添加或删除标签。

kubectl 命令也支持由对象的某一方面来驱动的更新命令。设置对象的这一方面可能对不同类别的对象意味着不同的字段：

- set <字段>：设置对象的某一方面。

说明：在 Kubernetes 1.5 版本中，并非所有动词驱动的命令都有对应的方面驱动的命令。

kubectl 工具支持以下额外的方式用来直接更新现时对象，不过这些操作要求 用户对 Kubernetes 对象的模式定义有很好的了解：

- edit：通过在编辑器中打开现时对象的配置，直接编辑其原始配置。
- patch：通过使用补丁字符串（Patch String）直接更改某现时对象的特定字段。关于补丁字符串的更详细信息，参见 [API 约定](#) 的 patch 节。

如何删除对象

你可以使用 delete 命令从集群中删除一个对象：

- delete <类别>/<名称>

你可以使用 kubectl delete 来执行指令式命令或者指令式对象配置。不同之处在于 传递给命令的参数。要将 kubectl delete 作为指令式命令使用，将要删除的对象作为 参数传递给它。下面是一个删除名为 nginx 的 Deployment 对象的命令：

```
kubectl delete deployment/nginx
```

如何查看对象

用来打印对象信息的命令有好几个：

- get：打印匹配到的对象的基本信息。使用 get -h 可以查看选项列表。
- describe：打印匹配到的对象的详细信息的汇集版本。
- logs：打印 Pod 中运行的容器的 stdout 和 stderr 输出。

使用 set 命令在创建对象之前修改对象

有些对象字段在 create 命令中没有对应的标志。在这些场景中，你可以使用 set 和 create 命令的组合来在对象创建之前设置字段值。这是通过将 create 命令的输出用管道方式传递给 set 命令来实现的，最后执行 create 命令来创建对象。下面是一个例子：

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run=client | kubectl set selector --local -f - 'environment=qa' -o yaml | kubectl create -f -
```

1. 命令 kubectl create service -o yaml --dry-run=client 创建 Service 的配置，但将其以 YAML 格式在标准输出上打印而不是发送给 API 服务器。
2. 命令 kubectl set selector --local -f - -o yaml 从标准输入读入配置，并将更新后的配置以 YAML 格式输出到标准输出。
3. 命令 kubectl create -f - 使用标准输入上获得的配置创建对象。

在创建之前使用 --edit 更改对象

你可以用 kubectl create --edit 来在对象被创建之前执行任意的变更。下面是一个例子：

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run=client > /tmp/srv.yaml  
kubectl create --edit -f /tmp/srv.yaml
```

1. 命令 kubectl create service 创建 Service 的配置并将其保存到 /tmp/srv.yaml 文件。
2. 命令 kubectl create --edit 在创建 Service 对象打开其配置文件进行编辑。

接下来

- [使用指令式对象配置管理 Kubernetes 对象](#)
- [使用声明式对象配置管理 Kubernetes 对象](#)
- [Kubectl 命令参考](#)
- [Kubernetes API 参考](#)

反馈

此页是否对您有帮助？

是否

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

最后修改 December 14, 2020 at 3:08 PM PST: [Update imperative-command.md \(4ab6bfbc2\)](#)

使用配置文件对 Kubernetes 对象进行命令式管理

可以使用 kubectl 命令行工具以及用 YAML 或 JSON 编写的对象配置文件来创建、更新和删除 Kubernetes 对象。本文档说明了如何使用配置文件定义和管理对象。

准备开始

安装 [kubectl](#)。

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`.

权衡

kubectl 工具支持三种对象管理：

- 命令式命令
- 命令式对象配置
- 声明式对象配置

参看 [Kubernetes 对象管理](#) 中关于每种对象管理的优缺点的讨论。

如何创建对象

你可以使用 `kubectl create -f` 从配置文件创建一个对象。请参考 [kubernetes API 参考](#) 有关详细信息。

- `kubectl create -f <filename|url>`

如何更新对象

警告： 使用 replace 命令更新对象会删除所有未在配置文件中指定的规范的某些部分。不应将其规范由集群部分管理的对象使用，比如类型为 LoadBalancer 的服务，其中 externalIPs 字段独立于配置文件进行管理。必须将独立管理的字段复制到配置文件中，以防止 replace 删除它们。

你可以使用 kubectl replace -f 根据配置文件更新活动对象。

- `kubectl replace -f <filename|url>`

如何删除对象

你可以使用 kubectl delete -f 删除配置文件中描述的对象。

- `kubectl delete -f <filename|url>`

说明：

如果配置文件在 metadata 节中设置了 generateName 字段而非 name 字段，你无法使用 `kubectl delete -f <filename|url>` 来删除该对象。你必须使用其他标志才能删除对象。例如：

```
kubectl delete <type> <name>
kubectl delete <type> -l <label>
```

如何查看对象

你可以使用 kubectl get -f 查看有关配置文件中描述的对象的信息。

- `kubectl get -f <filename|url> -o yaml`

`-o yaml` 标志指定打印完整的对象配置。使用 `kubectl get -h` 查看选项列表。

局限性

当完全定义每个对象的配置并将其记录在其配置文件中时，create、replace 和 delete 命令会很好的工作。但是，当更新一个活动对象，并且更新没有合并到其配置文件中时，下一次执行 replace 时，更新将丢失。如果控制器，例如 HorizontalPodAutoscaler，直接对活动对象进行更新，则会发生这种情况。这有一个例子：

1. 从配置文件创建一个对象。
2. 另一个源通过更改某些字段来更新对象。
3. 从配置文件中替换对象。在步骤2中所做的其他源的更改将丢失。

如果需要支持同一对象的多个编写器，则可以使用 kubectl apply 来管理该对象。

从 URL 创建和编辑对象而不保存配置

假设你具有对象配置文件的 URL。 你可以在创建对象之前使用 `kubectl create --edit` 对配置进行更改。 这对于指向可以由读者修改的配置文件的教程和任务特别有用。

```
kubectl create -f <url> --edit
```

从命令式命令迁移到命令式对象配置

从命令式命令迁移到命令式对象配置涉及几个手动步骤。

1. 将活动对象导出到本地对象配置文件：

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

1. 从对象配置文件中手动删除状态字段。
1. 对于后续的对象管理，只能使用 `replace`。

```
kubectl replace -f <kind>_<name>.yaml
```

定义控制器选择器和 PodTemplate 标签

警告：不建议在控制器上更新选择器。

推荐的方法是定义单个不变的 PodTemplate 标签，该标签仅由控制器选择器使用，而没有其他语义。

标签示例：

```
selector:  
  matchLabels:  
    controller-selector: "apps/v1/deployment/nginx"  
template:  
  metadata:  
    labels:  
      controller-selector: "apps/v1/deployment/nginx"
```

接下来

- [使用命令式命令管理 Kubernetes 对象](#)
- [使用对象配置管理 Kubernetes 对象 \(声明式\)](#)
- [Kubectl 命令参考](#)
- [Kubernetes API 参考](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 23, 2020 at 1:10 PM PST: [\[zh\] Sync changes from English site \(12\) \(81bc52053\)](#)

使用 kubectl patch 更新 API 对象

使用 kubectl patch 更新 Kubernetes API 对象。做一个策略性的合并 patch 或 JSON 合并 patch。

这个任务展示如何使用 kubectl patch 就地更新 API 对象。这个任务中的练习演示了一个策略性合并 patch 和一个 JSON 合并 patch。

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.

使用策略合并 patch 更新 Deployment

下面是具有两个副本的 Deployment 的配置文件。每个副本是一个 Pod，有一个容器：

[application/deployment-patch.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: patch-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
```

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: patch-demo-ctr
        image: nginx
    tolerations:
      - effect: NoSchedule
        key: dedicated
        value: test-team
```

创建 Deployment :

```
kubectl create -f https://k8s.io/examples/application/deployment-patch.yaml
```

查看与 Deployment 相关的 Pod :

```
kubectl get pods
```

输出显示 Deployment 有两个 Pod。1/1 表示每个 Pod 有一个容器:

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-28633765-670qr	1/1	Running	0	23s
patch-demo-28633765-j5qs3	1/1	Running	0	23s

把运行的 Pod 的名字记下来。稍后，你将看到这些 Pod 被终止并被新的 Pod 替换。

此时，每个 Pod 都有一个运行 nginx 镜像的容器。现在假设你希望每个 Pod 有两个容器：一个运行 nginx，另一个运行 redis。

创建一个名为 patch-file-containers.yaml 的文件。内容如下:

```
spec:
  template:
    spec:
      containers:
        - name: patch-demo-ctr-2
          image: redis
```

修补你的 Deployment :

```
kubectl patch deployment patch-demo --patch "$(cat patch-file-containers.yaml)"
```

查看修补后的 Deployment :

```
kubectl get deployment patch-demo --output yaml
```

输出显示 Deployment 中的 PodSpec 有两个容器:

```
containers:
- image: redis
  imagePullPolicy: Always
  name: patch-demo-ctr-2
...
- image: nginx
  imagePullPolicy: Always
  name: patch-demo-ctr
...

```

查看与 patch Deployment 相关的 Pod:

```
kubectl get pods
```

输出显示正在运行的 Pod 与以前运行的 Pod 有不同的名称。Deployment 终止了旧的 Pod，并创建了两个符合更新的部署规范的新 Pod。2/2 表示每个 Pod 有两个容器：

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-1081991389-2wrn5	2/2	Running	0	1m
patch-demo-1081991389-jmg7b	2/2	Running	0	1m

仔细查看其中一个 patch-demo Pod:

```
kubectl get pod <your-pod-name> --output yaml
```

输出显示 Pod 有两个容器：一个运行 nginx，一个运行 redis：

```
containers:
- image: redis
...
- image: nginx
...
```

策略性合并类的 patch 的说明

你在前面的练习中所做的 patch 称为策略性合并 patch (Strategic Merge Patch)。请注意，patch 没有替换 containers 列表。相反，它向列表中添加了一个新 Container。换句话说，patch 中的列表与现有列表合并。当你在列表中使用策略性合并 patch 时，并不总是这样。在某些情况下，列表是替换的，而不是合并的。

对于策略性合并 patch，列表可以根据其 patch 策略进行替换或合并。patch 策略由 Kubernetes 源代码中字段标记中的 patchStrategy 键的值指定。例如，PodSpec 结构体的 Containers 字段的 patchStrategy 为 merge：

```
type PodSpec struct {
...
    Containers []Container `json:"containers" patchStrategy:"merge"`
    patchMergeKey:"name" ...
}
```

你还可以在 [OpenAPI spec](#) 规范中看到 patch 策略：

```
"io.k8s.api.core.v1.PodSpec": {  
    ...  
    "containers": {  
        "description": "List of containers belonging to the pod. ...  
    },  
    "x-kubernetes-patch-merge-key": "name",  
    "x-kubernetes-patch-strategy": "merge"  
},
```

你可以在 [Kubernetes API 文档](#) 中看到 patch 策略。

创建一个名为 patch-file-tolerations.yaml 的文件。内容如下：

```
spec:  
template:  
  spec:  
    tolerations:  
      - effect: NoSchedule  
        key: disktype  
        value: ssd
```

对 Deployment 执行 patch 操作：

```
kubectl patch deployment patch-demo --patch "$(cat patch-file-containers.yaml)"
```

查看修补后的 Deployment：

```
kubectl get deployment patch-demo --output yaml
```

输出结果显示 Deployment 中的 PodSpec 只有一个容忍度设置：

```
containers:  
- image: redis  
  imagePullPolicy: Always  
  name: patch-demo-ctr-2  
...  
- image: nginx  
  imagePullPolicy: Always  
  name: patch-demo-ctr  
...
```

```
tolerations:  
  - effect: NoSchedule
```

```
key: disktype  
value: ssd
```

请注意，PodSpec 中的 tolerations 列表被替换，而不是合并。这是因为 PodSpec 的 tolerations 的字段标签中没有 patchStrategy 键。所以策略合并 patch 操作使用默认的 patch 策略，也就是 replace。

```
type PodSpec struct {  
    ...  
    Tolerations []Toleration `json:"tolerations,omitempty" protobuf:"bytes,  
22,opt,name=tolerations"`
```

使用 JSON 合并 patch 更新 Deployment

策略性合并 patch 不同于 [JSON 合并 patch](#)。使用 JSON 合并 patch，如果你想更新列表，你必须指定整个新列表。新的列表完全取代现有的列表。

kubectl patch 命令有一个 type 参数，你可以将其设置为以下值之一：

参数值	合并类型
json	JSON Patch, RFC 6902
merge	JSON Merge Patch, RFC 7386
strategic	策略性合并 patch

有关 JSON patch 和 JSON 合并 patch 的比较，查看 [JSON patch 和 JSON 合并 patch](#)。

type 参数的默认值是 strategic。在前面的练习中，我们做了一个策略性的合并 patch。

下一步，在相同的 Deployment 上执行 JSON 合并 patch。创建一个名为 patch-file-2 的文件。内容如下：

```
spec:  
  template:  
    spec:  
      containers:  
        - name: patch-demo-ctr-3  
          image: gcr.io/google-samples/node-hello:1.0
```

在 patch 命令中，将 type 设置为 merge：

```
kubectl patch deployment patch-demo --type merge --patch "$(cat patch-file-2.yaml)"
```

查看修补后的 Deployment：

```
kubectl get deployment patch-demo --output yaml
```

patch 中指定的containers列表只有一个 Container。 输出显示你所给出的 Contaier 列表替换了现有的 containers 列表。

```
spec:  
  containers:  
    - image: gcr.io/google-samples/node-hello:1.0  
      ...  
      name: patch-demo-ctr-3
```

列表中运行的 Pod :

```
kubectl get pods
```

在输出中，你可以看到已经终止了现有的 Pod，并创建了新的 Pod。1/1 表示每个新 Pod 只运行一个容器。

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-1307768864-69308	1/1	Running	0	1m
patch-demo-1307768864-c86dc	1/1	Running	0	1m

使用带 retainKeys 策略的策略合并 patch 更新 Deployment

[application/deployment-retainkeys.yaml](#)


```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: retainkeys-demo  
spec:  
  selector:  
    matchLabels:  
      app: nginx  
  strategy:  
    rollingUpdate:  
      maxSurge: 30%  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - name: retainkeys-demo-ctr  
          image: nginx
```

创建 Deployment :

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

这时，Deployment 被创建，并使用 RollingUpdate 策略。

创建一个名为 patch-file-no-retainkeys.yaml 的文件，内容如下：

```
spec:  
  strategy:  
    type: Recreate
```

修补你的 Deployment：

- [Bash](#)
- [PowerShell](#)

```
kubectl patch deployment retainkeys-demo --patch "$(cat patch-file-no-retainkeys.yaml)"
```

```
kubectl patch deployment retainkeys-demo --patch $(Get-Content patch-file-no-retainkeys.yaml -Raw)
```

在输出中，你可以看到，当 spec.strategy.rollingUpdate 已经拥有取值定义时，将其 type 设置为 Recreate 是不可能的。

```
The Deployment "retainkeys-demo" is invalid: spec.strategy.rollingUpdate:  
Forbidden: may not be specified when strategy `type` is 'Recreate'
```

更新 type 取值的同时移除 spec.strategy.rollingUpdate 现有值的方法是 为策略性合并操作设置 retainKeys 策略：

创建另一个名为 patch-file-retainkeys.yaml 的文件，内容如下：

```
spec:  
  strategy:  
    $retainKeys:  
      - type  
    type: Recreate
```

使用此 patch，我们表达了希望只保留 strategy 对象的 type 键。这样，在 patch 操作期间 rollingUpdate 会被删除。

使用新的 patch 重新修补 Deployment：

- [Bash](#)
- [PowerShell](#)

```
kubectl patch deployment retainkeys-demo --patch "$(cat patch-file-retainkeys.yaml)"
```

```
kubectl patch deployment retainkeys-demo --patch $(Get-Content patch-file-retainkeys.yaml -Raw)
```

检查 Deployment 的内容：

```
kubectl get deployment retainkeys-demo --output yaml
```

输出显示 Deployment 中的 strategy 对象不再包含 rollingUpdate 键：

```
spec:  
  strategy:  
    type: Recreate  
  template:
```

关于使用 retainKeys 策略的策略合并 patch 操作的说明

在前文练习中所执行的称作 带 retainKeys 策略的策略合并 patch (Strategic Merge Patch with retainKeys Strategy)。这种方法引入了一种新的 \$retainKey 指令，具有如下策略：

- 其中包含一个字符串列表；
- 所有需要被保留的字段必须在 \$retainKeys 列表中给出；
- 对于已有的字段，会和对象上对应的内容合并；
- 在修补操作期间，未找到的字段都会被清除；
- 列表 \$retainKeys 中的所有字段必须 patch 操作所给字段的超集，或者与之完全一致。

策略 retainKeys 并不能对所有对象都起作用。它仅对那些 Kubernetes 源码中 patchStrategy 字段标志值包含 retainKeys 的字段有用。例如 DeploymentSpec 结构的 Strategy 字段就包含了 patchStrategy 为 retainKeys 的标志。

```
type DeploymentSpec struct {  
  ...  
  // +patchStrategy=retainKeys  
  Strategy DeploymentStrategy `json:"strategy,omitempty"  
  patchStrategy:"retainKeys" ...`
```

你也可以查看 [OpenAPI 规范](#) 中的 retainKeys 策略：

```
"io.k8s.api.apps.v1.DeploymentSpec": {  
  ...  
  "strategy": {  
    "$ref": "#/definitions/io.k8s.api.apps.v1.DeploymentStrategy",  
    "description": "The deployment strategy to use to replace existing pods with
```

```
new ones.",
  "x-kubernetes-patch-strategy": "retainKeys"
},
```

而且你也可以在 [Kubernetes API 文档](#) 中看到 retainKey 策略。

kubectl patch 命令的其他形式

kubectl patch 命令使用 YAML 或 JSON。它可以接受以文件形式提供的补丁，也可以接受直接在命令行中给出的补丁。

创建一个文件名称是 patch-file.json 内容如下：

```
{
  "spec": {
    "template": {
      "spec": {
        "containers": [
          {
            "name": "patch-demo-ctr-2",
            "image": "redis"
          }
        ]
      }
    }
  }
}
```

以下命令是等价的：

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"
kubectl patch deployment patch-demo --patch 'spec:\n template:\n   spec:\n     containers:\n       - name: patch-demo-ctr-2\n         image: redis'
```

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.json)"
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spec": {"containers": [{"name": "patch-demo-ctr-2", "image": "redis"}]}}}}'
```

总结

在本练习中，你使用 kubectl patch 更改了 Deployment 对象的当前配置。你没有更改最初用于创建 Deployment 对象的配置文件。用于更新 API 对象的其他命令包括 [kubectl annotate](#)，[kubectl edit](#)，[kubectl replace](#)，[kubectl scale](#)，和 [kubectl apply](#)。

接下来

- [Kubernetes 对象管理](#)
- [使用指令式命令管理 Kubernetes 对象](#)
- [使用配置文件执行 Kubernetes 对象的指令式管理](#)
- [使用配置文件对 Kubernetes 对象进行声明式管理](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 October 27, 2020 at 7:21 PM PST: [Fix format error \(40ed84540\)](#)

管理 Secrets

使用 Secrets 管理机密配置数据。

[使用 kubectl 管理 Secret](#)

使用 kubectl 命令行创建 Secret 对象。

[使用配置文件管理 Secret](#)

使用资源配置文件创建 Secret 对象。

[使用 Kustomize 管理 Secret](#)

使用 kustomization.yaml 文件创建 Secret 对象。

使用 kubectl 管理 Secret

使用 kubectl 命令行创建 Secret 对象。

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](#)

创建 Secret

一个 Secret 可以包含 Pod 访问数据库所需的用户凭证。例如，由用户名和密码组成的数据连接字符串。你可以在本地计算机上，将用户名存储在文件 `./username.txt` 中，将密码存储在文件 `./password.txt` 中。

```
echo -n 'admin' > ./username.txt  
echo -n '1f2d1e2e67df' > ./password.txt
```

上面两个命令中的 `-n` 标志确保生成的文件在文本末尾不包含额外的换行符。这一点很重要，因为当 `kubectl` 读取文件并将内容编码为 base64 字符串时，多余的换行符也会被编码。

`kubectl create secret` 命令将这些文件打包成一个 Secret 并在 API 服务器上创建对象。

```
kubectl create secret generic db-user-pass \  
--from-file=./username.txt \  
--from-file=./password.txt
```

输出类似于：

```
secret/db-user-pass created
```

默认密钥名称是文件名。你可以选择使用 `--from-file=[key=]source` 来设置密钥名称。例如：

```
kubectl create secret generic db-user-pass \  
--from-file=username=./username.txt \  
--from-file=password=./password.txt
```

你无需转义文件 (`--from-file`) 中的密码的特殊字符。

你还可以使用 `--from-literal=<key>=<value>` 标签提供 Secret 数据。可以多次使用此标签，提供多个键值对。请注意，特殊字符（例如：`$`，`\`，`*`，`=` 和 `!`）由你的 [shell](#) 解释执行，而且需要转义。在大多数 shell 中，转义密码最简便的方法是用单引号括起来。比如，如果你的密码是 `S!B*d$zDsb=`，可以像下面一样执行命令：

```
kubectl create secret generic dev-db-secret \  
--from-literal=username=devuser \  
--from-literal=password='S!B\*d$zDsb='
```

验证 Secret

你可以检查 secret 是否已创建：

```
kubectl get secrets
```

输出类似于：

NAME	TYPE	DATA	AGE
db-user-pass	Opaque	2	51s

你可以查看 Secret 的描述：

```
kubectl describe secrets/db-user-pass
```

输出类似于：

```
Name: db-user-pass
Namespace: default
Labels: <none>
Annotations: <none>
```

```
Type: Opaque
```

```
Data
```

```
=====
```

```
password: 12 bytes
username: 5 bytes
```

kubectl get 和 kubectl describe 命令默认不显示 Secret 的内容。这是为了防止 Secret 被意外暴露给旁观者或存储在终端日志中。

解码 Secret

要查看我们刚刚创建的 Secret 的内容，可以运行以下命令：

```
kubectl get secret db-user-pass -o jsonpath='{.data}'
```

输出类似于：

```
{"password.txt":"MWYyZDFIMmU2N2Rm","username.txt":"YWRtaW4="}
```

现在你可以解码 password.txt 的数据：

```
echo 'MWYyZDFIMmU2N2Rm' | base64 --decode
```

输出类似于：

```
1f2d1e2e67df
```

清理

删除刚刚创建的 Secret：

```
kubectl delete secret db-user-pass
```

接下来

- 进一步阅读 [Secret 概念](#)
- 了解如何[使用配置文件管理 Secret](#)
- 了解如何[使用 kustomize 管理 Secret](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 14, 2020 at 5:38 PM PST: [Update managing-secret-using-kubectl.md \(7089e5a54\)](#)

使用配置文件管理 Secret

使用资源配置文件创建 Secret 对象。

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](#)

创建配置文件

你可以先用 JSON 或 YAML 格式在文件中创建 Secret，然后创建该对象。 [Secret](#) 资源包含2个键值对： data 和 stringData。 data 字段用来存储 base64 编码的任意数据。 提供 stringData 字段是为了方便，它允许 Secret 使用未编码的字符串。 data 和 stringData 的键必须由字母、数字、 - , _ 或 . 组成。

例如，要使用 Secret 的 data 字段存储两个字符串，请将字符串转换为 base64 ，如下所示：

```
echo -n 'admin' | base64
```

输出类似于：

```
YWRtaW4=
```

```
echo -n '1f2d1e2e67df' | base64
```

输出类似于：

```
MWYyZDFIMmU2N2Rm
```

编写一个 Secret 配置文件，如下所示：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFIMmU2N2Rm
```

注意，Secret 对象的名称必须是有效的 [DNS 子域名](#).

说明：

Secret 数据的 JSON 和 YAML 序列化结果是以 base64 编码的。换行符在这些字符串中无效，必须省略。在 Darwin/macOS 上使用 base64 工具时，用户不应该使用 -b 选项分割长行。相反地，Linux 用户 应该 在 base64 命令中添加 -w 0 选项，或者在 -w 选项不可用的情况下，输入 base64 | tr -d '\n'.

对于某些场景，你可能希望使用 stringData 字段。这字段可以将一个非 base64 编码的字符串直接放入 Secret 中，当创建或更新该 Secret 时，此字段将被编码。

上述用例的实际场景可能是这样：当你部署应用时，使用 Secret 存储配置文件，你希望在部署过程中，填入部分内容到该配置文件。

例如，如果你的应用程序使用以下配置文件：

```
apiUrl: "https://my.api.com/api/v1"
username: "<user>"
password: "<password>"
```

你可以使用以下定义将其存储在 Secret 中：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: |
```

```
apiUrl: "https://my.api.com/api/v1"  
username: <user>  
password: <password>
```

创建 Secret 对象

现在使用 [kubectl apply](#) 创建 Secret：

```
kubectl apply -f ./secret.yaml
```

输出类似于：

```
secret/mysecret created
```

检查 Secret

`stringData` 字段是只写的。获取 Secret 时，此字段永远不会输出。例如，如果你运行以下命令：

```
kubectl get secret mysecret -o yaml
```

输出类似于：

```
apiVersion: v1  
kind: Secret  
metadata:  
  creationTimestamp: 2018-11-15T20:40:59Z  
  name: mysecret  
  namespace: default  
  resourceVersion: "7225"  
  uid: c280ad2e-e916-11e8-98f2-025000000001  
type: Opaque  
data:  
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6Iht7dXNlcm5hbWV9fQpwYXNzd29yZDoge3twYXNzd29yZH19
```

命令 `kubectl get` 和 `kubectl describe` 默认不显示 Secret 的内容。这是为了防止 Secret 意外地暴露给旁观者或者保存在终端日志中。检查编码数据的实际内容，请参考[解码 secret](#)。

如果在 `data` 和 `stringData` 中都指定了一个字段，比如 `username`，字段值来自 `stringData`。例如，下面的 Secret 定义：

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mysecret  
type: Opaque
```

```
data:  
  username: YWRtaW4=  
stringData:  
  username: administrator
```

结果有以下 Secret :

```
apiVersion: v1  
kind: Secret  
metadata:  
  creationTimestamp: 2018-11-15T20:46:46Z  
  name: mysecret  
  namespace: default  
  resourceVersion: "7579"  
  uid: 91460ecb-e917-11e8-98f2-025000000001  
type: Opaque  
data:  
  username: YWRtaW5pc3RyYXRvcg==
```

其中 YWRtaW5pc3RyYXRvcg== 解码成 administrator。

清理

删除你刚才创建的 Secret :

```
kubectl delete secret db-user-pass
```

接下来

- 进一步阅读 [Secret 概念](#)
- 了解如何[使用 kubectl 命令管理 Secret](#)
- 了解如何[使用 kustomize 管理 Secret](#)

反馈

此页是否对您有帮助 ?

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

使用 Kustomize 管理 Secret

使用 kustomization.yaml 文件创建 Secret 对象。

从 kubernetes v1.14 开始，kubectl 支持[使用 Kustomize 管理对象](#)。Kustomize 提供了资源生成器 (Generators) 来创建 Secret 和 ConfigMap。Kustomize 生成器应该在某个目录的 kustomization.yaml 文件中指定。生成 Secret 后，你可以使用 kubectl apply 在 API 服务器上创建该 Secret。

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](#)

创建 Kustomization 文件

你可以在 kustomization.yaml 中定义 secretGenerator，并在定义中引用其他现成的文件，生成 Secret。例如：下面的 kustomization 文件引用了 ./username.txt 和 ./password.txt 文件：

```
secretGenerator:  
- name: db-user-pass  
  files:  
    - username.txt  
    - password.txt
```

你也可以在 kustomization.yaml 文件中指定一些字面量定义 secretGenerator。例如：下面的 kustomization.yaml 文件中包含了 username 和 password 两个字面量：

```
secretGenerator:  
- name: db-user-pass  
  literals:  
    - username=admin  
    - password=1f2d1e2e67df
```

注意，上面两种情况，你都不需要使用 base64 编码。

创建 Secret

使用 kubectl apply 命令应用包含 kustomization.yaml 文件的目录创建 Secret。

```
kubectl apply -k .
```

输出类似于：

```
secret/db-user-pass-96mffmfh4k created
```

请注意，生成 Secret 时，Secret 的名称最终是由 name 字段和数据的哈希值拼接而成。这将保证每次修改数据时生成一个新的 Secret。

检查创建的 Secret

你可以检查刚才创建的 Secret：

```
kubectl get secrets
```

输出类似于：

NAME	TYPE	DATA	AGE
db-user-pass-96mffmfh4k	Opaque	2	51s

你可以看到 Secret 的描述：

```
kubectl describe secrets/db-user-pass-96mffmfh4k
```

输出类似于：

```
Name: db-user-pass
Namespace: default
Labels: <none>
Annotations: <none>
```

```
Type: Opaque
```

```
Data
```

```
====
```

```
password.txt: 12 bytes
username.txt: 5 bytes
```

kubectl get 和 kubectl describe 命令默认不显示 Secret 的内容。这是为了防止 Secret 被意外暴露给旁观者或存储在终端日志中。检查编码后的实际内容，请参考[解码 secret](#)。-->

清理

删除你刚才创建的 Secret：

```
kubectl delete secret db-user-pass-96mffmfh4k
```

接下来

- 进一步阅读 [Secret 概念](#)
- 了解如何[使用 kubectl 命令管理 Secret](#)
- 了解如何[使用配置文件管理 Secret](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 14, 2020 at 5:51 PM PST: [Update managing-secret-using-kustomize.md \(8959cc699\)](#)

给应用注入数据

给你的工作负载 Pod 指定配置和其他数据。

[为容器设置启动时要执行的命令和参数](#)

[为容器设置环境变量](#)

[定义相互依赖的环境变量](#)

[通过环境变量将 Pod 信息呈现给容器](#)

[通过文件将 Pod 信息呈现给容器](#)

[使用 Secret 安全地分发凭证](#)

为容器设置启动时要执行的命令和参数

本页将展示如何为 [Pod](#) 中容器设置启动时要执行的命令及其参数。

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`.

创建 Pod 时设置命令及参数

创建 Pod 时，可以为其下的容器设置启动时要执行的命令及其参数。如果要设置命令，就填写在配置文件的 `command` 字段下，如果要设置命令的参数，就填写在配置文件的 `args` 字段下。一旦 Pod 创建完成，该命令及其参数就无法再进行更改了。

如果在配置文件中设置了容器启动时要执行的命令及其参数，那么容器镜像中自带的命令与参数将被覆盖而不再执行。如果配置文件中只是设置了参数，却没有设置其对应的命令，那么容器镜像中自带的命令会使用该新参数作为其执行时的参数。

说明：在有些容器运行时中，`command` 字段对应 `entrypoint`，请参阅下面的 [说明事项](#)。

本示例中，将创建一个只包含单个容器的 Pod。在 Pod 配置文件中设置了一个命令与两个参数：

[pods/commands.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
    - name: command-demo-container
      image: debian
      command: ["printenv"]
      args: ["HOSTNAME", "KUBERNETES_PORT"]
  restartPolicy: OnFailure
```

1. 基于 YAML 文件创建一个 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/commands.yaml
```

1. 获取正在运行的 Pods：

```
kubectl get pods
```

查询结果显示在 `command-demo` 这个 Pod 下运行的容器已经启动完成。

1. 如果要获取容器启动时执行命令的输出结果，可以通过 Pod 的日志进行查看：

```
kubectl logs command-demo
```

日志中显示了 HOSTNAME 与 KUBERNETES_PORT 这两个环境变量的值：

```
command-demo
tcp://10.3.240.1:443
```

使用环境变量来设置参数

在上面的示例中，我们直接将一串字符作为命令的参数。除此之外，我们还可以将环境变量作为命令的参数。

```
env:
- name: MESSAGE
  value: "hello world"
command: ["/bin/echo"]
args: ["$(MESSAGE)"]
```

这意味着你可以将那些用来设置环境变量的方法应用于设置命令的参数，其中包括了 [ConfigMaps](#) 与 [Secrets](#)。

说明： 环境变量需要加上括号，类似于 "\$(VAR)"。这是在 command 或 args 字段使用变量的格式要求。

在 Shell 来执行命令

有时候，你需要在 Shell 脚本中运行命令。例如，你要执行的命令可能由多个命令组合而成，或者它就是一个 Shell 脚本。这时，就可以通过如下方式在 Shell 中执行命令：

```
command: ["/bin/sh"]
args: ["-c", "while true; do echo hello; sleep 10;done"]
```

说明事项

下表给出了 Docker 与 Kubernetes 中对应的字段名称。

描述	Docker 字段名称	Kubernetes 字段名称
容器执行的命令	Entrypoint	command
传给命令的参数	Cmd	args

如果要覆盖默认的 Entrypoint 与 Cmd，需要遵循如下规则：

- 如果在容器配置中没有设置 command 或者 args，那么将使用 Docker 镜像自带的命令及其参数。
- 如果在容器配置中只设置了 command 但是没有设置 args，那么容器启动时只会执行该命令，Docker 镜像中自带的命令及其参数会被忽略。
- 如果在容器配置中只设置了 args，那么 Docker 镜像中自带的命令会使用该新参数作为其执行时的参数。

- 如果在容器配置中同时设置了 command 与 args , 那么 Docker 镜像中自带的命令及其参数会被忽略。容器启动时只会执行配置中设置的命令，并使用配置中设置的参数作为命令的参数。

下面是一些例子：

镜像 Entrypoint	镜像 Cmd	容器 command	容器 args	命令执行
[/ep-1]	[foo bar]	<not set>	<not set>	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	<not set>	[ep-2]
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	[/ep-2]	[zoo boo]	[ep-2 zoo boo]

接下来

- 进一步了解[配置 Pod 和容器](#)
- 进一步了解[在容器中运行命令](#)
- 参阅 [Container API 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 August 16, 2020 at 8:50 PM PST: [\[zh\] Tidy up and fix links in tasks section \(10/10\) \(114a75d78\)](#)

为容器设置环境变量

本页将展示如何为 kubernetes Pod 下的容器设置环境变量。

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](#)

为容器设置一个环境变量

创建 Pod 时，可以为其下的容器设置环境变量。通过配置文件的 env 或者 envFrom 字段来设置环境变量。

本示例中，将创建一个只包含单个容器的 Pod。Pod 的配置文件中设置环境变量的名称为 DEMO_GREETING，其值为 "Hello from the environment"。下面是 Pod 的配置文件内容：

[pods/inject/envvars.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envvars
spec:
  containers:
    - name: envar-demo-container
      image: gcr.io/google-samples/node-hello:1.0
      env:
        - name: DEMO_GREETING
          value: "Hello from the environment"
        - name: DEMO_FAREWELL
          value: "Such a sweet sorrow"
```

1. 基于 YAML 文件创建一个 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/inject/envvars.yaml
```

1. 获取一下当前正在运行的 Pods 信息：

```
kubectl get pods -l purpose=demonstrate-envvars
```

查询结果应为：

NAME	READY	STATUS	RESTARTS	AGE
envar-demo	1/1	Running	0	9s

1. 列出 Pod 容器的环境变量：

```
kubectl exec envar-demo -- printenv
```

打印结果应为：

```
NODE_VERSION=4.4.2
EXAMPLE_SERVICE_PORT_8080_TCP_ADDR=10.3.245.237
```

```
HOSTNAME=envar-demo
...
DEMO_GREETING=Hello from the environment
DEMO_FAREWELL=Such a sweet sorrow
```

说明：通过 env 或 envFrom 字段设置的环境变量将覆盖容器镜像中指定的所有环境变量。

说明：环境变量之间可能出现互相依赖或者循环引用的情况，使用之前需注意引用顺序

在配置中使用环境变量

您在 Pod 的配置中定义的环境变量可以在配置的其他地方使用，例如可用在为 Pod 的容器设置的命令和参数中。在下面的示例配置中，环境变量 GREETING，HONORIFIC 和 NAME 分别设置为 Warm greetings to，The Most Honorable 和 Kubernetes。然后这些环境变量在传递给容器 env-print-demo 的 CLI 参数中使用。

```
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
    - name: env-print-demo
      image: bash
      env:
        - name: GREETING
          value: "Warm greetings to"
        - name: HONORIFIC
          value: "The Most Honorable"
        - name: NAME
          value: "Kubernetes"
      command: ["echo"]
      args: ["$(GREETING) $(HONORIFIC) $(NAME)"]
```

创建后，命令 echo Warm greetings to The Most Honorable Kubernetes 将在容器中运行。

接下来

- 进一步了解[环境变量](#)
- 进一步了解[通过环境变量来使用 Secret](#)
- 关于[EnvVarSource](#) 资源的信息。

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 October 30, 2020 at 5:15 PM PST: [Update define-environment-variable-container.md \(379b6907f\)](#)

定义相互依赖的环境变量

本页展示了如何为 Kubernetes Pod 中的容器定义相互依赖的环境变量。

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](#)

为容器定义相互依赖的环境变量

当创建一个 Pod 时，你可以为运行在 Pod 中的容器设置相互依赖的环境变量。设置相互依赖的环境变量，你就可以在配置清单文件的 env 的 value 中使用 \$(VAR_NAME)。

在本练习中，你会创建一个单容器的 Pod。此 Pod 的配置文件定义了一个已定义常用用法的相互依赖的环境变量。下面是 Pod 的配置清单：

[pods/inject/dependent-envvars.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dependent-envvars-demo
spec:
  containers:
    - name: dependent-envvars-demo
      args:
        - while true; do echo -en '\n'; printf
          UNCHANGED_REFERENCE=$UNCHANGED_REFERENCE'\n'; printf
```

```
SERVICE_ADDRESS=$SERVICE_ADDRESS'\n';printf  
ESCAPED_REFERENCE=$ESCAPED_REFERENCE'\n'; sleep 30; done;  
command:  
- sh  
- -C  
image: busybox  
env:  
- name: SERVICE_PORT  
value: "80"  
- name: SERVICE_IP  
value: "172.17.0.1"  
- name: UNCHANGED_REFERENCE  
value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"  
- name: PROTOCOL  
value: "https"  
- name: SERVICE_ADDRESS  
value: "$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"  
- name: ESCAPED_REFERENCE  
value: "$$(PROTOCOL)://$(SERVICE_IP):$(SERVICE_PORT)"
```

1. 依据清单创建 Pod :

```
kubectl apply -f https://k8s.io/examples/pods/inject/dependent-envvars.yaml  
pod/dependent-envvars-demo created
```

2. 列出运行的 Pod :

```
kubectl get pods dependent-envvars-demo  
NAME READY STATUS RESTARTS AGE  
dependent-envvars-demo 1/1 Running 0 9s
```

3. 检查 Pod 中运行容器的日志 :

```
kubectl logs pod/dependent-envvars-demo
```

```
UNCHANGED_REFERENCE=$(PROTOCOL)://172.17.0.1:80  
SERVICE_ADDRESS=https://172.17.0.1:80  
ESCAPED_REFERENCE=$(PROTOCOL)://172.17.0.1:80
```

如上所示，你已经定义了 SERVICE_ADDRESS 的正确依赖引用， UNCHANGED_REFERENCE 的错误依赖引用，并跳过了 ESCAPED_REFERENCE 的依赖引用。

如果环境变量被引用时已事先定义，则引用可以正确解析，比如 SERVICE_ADDRESS 的例子。

当环境变量未定义或仅包含部分变量时，未定义的变量会被当做普通字符串对待，比如 UNCHANGED_REFERENCE 的例子。注意，解析不正确的环境变量通常不会阻止容器启动。

`$(VAR_NAME)` 这样的语法可以用两个 \$ 转义，既：`$$`(`VAR_NAME`)。无论引用的变量是否定义，转义的引用永远不会展开。这一点可以从上面 ESCAPED_REFERENCE 的例子得到印证。

接下来

- 进一步了解[环境变量](#)。
- 参阅 [EnvVarSource](#).

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 27, 2020 at 3:48 PM PST: [24157, add page:zh-task-define-interdependent-environment-variables make chage according to tengqm's comment \(764f5ccf9\)](#)

通过环境变量将 Pod 信息呈现给容器

此页面展示 Pod 如何使用环境变量把自己的信息呈现给 Pod 中运行的容器。环境变量可以呈现 Pod 的字段和容器字段。

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`.

Downward API

有两种方式可以将 Pod 和 Container 字段呈现给运行中的容器：

- 环境变量

- 卷文件

这两种呈现 Pod 和 Container 字段的方式统称为 *Downward API*。

用 Pod 字段作为环境变量的值

在这个练习中，你将创建一个包含一个容器的 Pod。这是该 Pod 的配置文件：

[pods/inject/dapi-envvars-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envvars-fieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "sh", "-c" ]
      args:
        - while true; do
            echo -en '\n';
            printenv MY_NODE_NAME MY_POD_NAME MY_POD_NAMESPACE;
            printenv MY_POD_IP MY_POD_SERVICE_ACCOUNT;
            sleep 10;
        done;
  env:
    - name: MY_NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MY_POD_SERVICE_ACCOUNT
      valueFrom:
```

```
fieldRef:  
  fieldPath: spec.serviceAccountName  
restartPolicy: Never
```

这个配置文件中，你可以看到五个环境变量。env 字段是一个 [EnvVars](#) 对象的数组。数组中第一个元素指定 MY_NODE_NAME 这个环境变量从 Pod 的 spec.nodeName 字段获取变量值。同样，其它环境变量也是从 Pod 的字段获取它们的变量值。

说明：本示例中的字段是 Pod 字段，不是 Pod 中 Container 的字段。

创建Pod：

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envvars-pod.yaml
```

验证 Pod 中的容器运行正常：

```
kubectl get pods
```

查看容器日志：

```
kubectl logs dapi-envvars-fieldref
```

输出信息显示了所选择的环境变量的值：

```
minikube  
dapi-envvars-fieldref  
default  
172.17.0.4  
default
```

要了解为什么这些值在日志中，请查看配置文件中的 command 和 args 字段。当容器启动时，它将五个环境变量的值写入 stdout。每十秒重复执行一次。

接下来，通过打开一个 Shell 进入 Pod 中运行的容器：

```
kubectl exec -it dapi-envvars-fieldref -- sh
```

在 Shell 中，查看环境变量：

```
/# printenv
```

输出信息显示环境变量已经设置为 Pod 字段的值。

```
MY_POD_SERVICE_ACCOUNT=default  
...  
MY_POD_NAMESPACE=default  
MY_POD_IP=172.17.0.4  
...  
MY_NODE_NAME=minikube
```

```
...
MY_POD_NAME=dapi-envars-fieldref
```

用 Container 字段作为环境变量的值

前面的练习中，你将 Pod 字段作为环境变量的值。接下来这个练习中，你将用 Container 字段作为环境变量的值。这里是包含一个容器的 Pod 的配置文件：

[pods/inject/dapi-envars-container.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox:1.24
      command: [ "sh", "-c" ]
      args:
        - while true; do
            echo -en '\n';
            printenv MY_CPU_REQUEST MY_CPU_LIMIT;
            printenv MY_MEM_REQUEST MY_MEM_LIMIT;
            sleep 10;
        done;
  resources:
    requests:
      memory: "32Mi"
      cpu: "125m"
    limits:
      memory: "64Mi"
      cpu: "250m"
  env:
    - name: MY_CPU_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: requests.cpu
    - name: MY_CPU_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test-container
          resource: limits.cpu
    - name: MY_MEM_REQUEST
```

```
valueFrom:  
  resourceFieldRef:  
    containerName: test-container  
    resource: requests.memory  
  - name: MY_MEM_LIMIT  
    valueFrom:  
      resourceFieldRef:  
        containerName: test-container  
        resource: limits.memory  
restartPolicy: Never
```

这个配置文件中，你可以看到四个环境变量。env 字段是一个 [EnvVars](#) 对象的数组。数组中第一个元素指定 MY_CPU_REQUEST 这个环境变量从 Container 的 requests.cpu 字段获取变量值。同样，其它环境变量也是从 Container 的字段获取它们的变量值。

说明：本例中使用的是 Container 的字段而不是 Pod 的字段。

创建Pod：

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envvars-container.yaml
```

验证 Pod 中的容器运行正常：

```
kubectl get pods
```

查看容器日志：

```
kubectl logs dapi-envvars-resourcefieldref
```

输出信息显示了所选择的环境变量的值：

```
1  
1  
33554432  
67108864
```

接下来

- [给容器定义环境变量](#)
- [PodSpec](#)
- [Container](#)
- [EnvVar](#)
- [EnvVarSource](#)
- [ObjectFieldSelector](#)
- [ResourceFieldSelector](#)

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 14, 2020 at 6:35 PM PST: [Update environment-variable-expose-pod-information.md \(69249f045\)](#)

通过文件将 Pod 信息呈现给容器

此页面描述 Pod 如何使用 DownwardAPIVolumeFile 把自己的信息呈现给 Pod 中运行的容器。DownwardAPIVolumeFile 可以呈现 Pod 的字段和容器字段。

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.

Downward API

有两种方式可以将 Pod 和 Container 字段呈现给运行中的容器：

- [环境变量](#)
- 卷文件

这两种呈现 Pod 和 Container 字段的方式都称为 *Downward API*.

存储 Pod 字段

在这个练习中，你将创建一个包含一个容器的 Pod。Pod 的配置文件如下：

[pods/inject/dapi-volume.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example
  labels:
    zone: us-est-coast
    cluster: test-cluster1
```

```

rack: rack-22
annotations:
  build: two
  builder: john-doe
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox
      command: ["sh", "-c"]
      args:
        - while true; do
            if [[ -e /etc/podinfo/labels ]]; then
              echo -en '\n\n'; cat /etc/podinfo/labels; fi;
            if [[ -e /etc/podinfo/annotations ]]; then
              echo -en '\n\n'; cat /etc/podinfo/annotations; fi;
            sleep 5;
        done;
  volumeMounts:
    - name: podinfo
      mountPath: /etc/podinfo
volumes:
  - name: podinfo
downwardAPI:
  items:
    - path: "labels"
      fieldRef:
        fieldPath: metadata.labels
    - path: "annotations"
      fieldRef:
        fieldPath: metadata.annotations

```

在配置文件中，你可以看到 Pod 有一个 downwardAPI 类型的卷，并且挂载到容器中的 /etc/podinfo 目录。

查看 downwardAPI 下面的 items 数组。每个数组元素都是一个 [DownwardAPIVolumeFile](#) 对象。第一个元素指示 Pod 的 metadata.labels 字段的值保存在名为 labels 的文件中。第二个元素指示 Pod 的 annotations 字段的值保存在名为 annotations 的文件中。

说明：本示例中的字段是Pod字段，不是Pod中容器的字段。

创建 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume.yaml
```

验证Pod中的容器运行正常：

```
kubectl get pods
```

查看容器的日志：

```
kubectl logs kubernetes-downwardapi-volume-example
```

输出显示 labels 和 annotations 文件的内容：

```
cluster="test-cluster1"  
rack="rack-22"  
zone="us-est-coast"  
  
build="two"  
builder="john-doe"
```

进入 Pod 中运行的容器，打开一个 Shell：

```
kubectl exec -it kubernetes-downwardapi-volume-example -- sh
```

在该 Shell 中，查看 labels 文件：

```
/# cat /etc/podinfo/labels
```

输出显示 Pod 的所有标签都已写入 labels 文件。

```
cluster="test-cluster1"  
rack="rack-22"  
zone="us-est-coast"
```

同样，查看 annotations 文件：

```
/# cat /etc/podinfo/annotations
```

查看 /etc/podinfo 目录下的文件：

```
/# ls -laR /etc/podinfo
```

在输出中可以看到，labels 和 annotations 文件都在一个临时子目录中。在这个例子，..2982_06_02_21_47_53.299460680。在 /etc/podinfo 目录中，..data 是一个指向临时子目录的符号链接。/etc/podinfo 目录中，labels 和 annotations 也是符号链接。

```
drwxr-xr-x ... Feb 6 21:47 ..2982_06_02_21_47_53.299460680  
lwxrwxrwx ... Feb 6 21:47 ..data -> ..2982_06_02_21_47_53.299460680  
lwxrwxrwx ... Feb 6 21:47 annotations -> ..data/annotations  
lwxrwxrwx ... Feb 6 21:47 labels -> ..data/labels
```

```
/etc/podinfo/..2982_06_02_21_47_53.299460680:  
total 8  
-rw-r--r-- ... Feb 6 21:47 annotations  
-rw-r--r-- ... Feb 6 21:47 labels
```

用符号链接可实现元数据的动态原子性刷新；更新将写入一个新的临时目录，然后通过使用[rename\(2\)](#) 完成 ..data 符号链接的原子性更新。

说明：如果容器以 [subPath](#) 卷挂载方式来使用 Downward API，则该容器无法收到更新事件。

退出 Shell：

```
/# exit
```

存储容器字段

前面的练习中，你将 Pod 字段保存到 DownwardAPIVolumeFile 中。接下来这个练习，你将存储 Container 字段。这里是包含一个容器的 Pod 的配置文件：

[pods/inject/dapi-volume-resources.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox:1.24
      command: ["sh", "-c"]
      args:
        - while true; do
            echo -en '\n';
            if [[ -e /etc/podinfo/cpu_limit ]]; then
              echo -en '\n'; cat /etc/podinfo/cpu_limit; fi;
            if [[ -e /etc/podinfo/cpu_request ]]; then
              echo -en '\n'; cat /etc/podinfo/cpu_request; fi;
            if [[ -e /etc/podinfo/mem_limit ]]; then
              echo -en '\n'; cat /etc/podinfo/mem_limit; fi;
            if [[ -e /etc/podinfo/mem_request ]]; then
              echo -en '\n'; cat /etc/podinfo/mem_request; fi;
            sleep 5;
        done;
  resources:
    requests:
      memory: "32Mi"
      cpu: "125m"
    limits:
      memory: "64Mi"
      cpu: "250m"
```

```
volumeMounts:
  - name: podinfo
    mountPath: /etc/podinfo
volumes:
  - name: podinfo
    downwardAPI:
      items:
        - path: "cpu_limit"
          resourceFieldRef:
            containerName: client-container
            resource: limits.cpu
            divisor: 1m
        - path: "cpu_request"
          resourceFieldRef:
            containerName: client-container
            resource: requests.cpu
            divisor: 1m
        - path: "mem_limit"
          resourceFieldRef:
            containerName: client-container
            resource: limits.memory
            divisor: 1Mi
        - path: "mem_request"
          resourceFieldRef:
            containerName: client-container
            resource: requests.memory
            divisor: 1Mi
```

在这个配置文件中，你可以看到 Pod 有一个 downwardAPI 类型的卷，并且挂载到容器的 /etc/podinfo 目录。

查看 downwardAPI 下面的 items 数组。每个数组元素都是一个 DownwardAPIVolumeFile。

第一个元素指定名为 client-container 的容器中 limits.cpu 字段的值应保存在名为 cpu_limit 的文件中。

创建Pod：

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume-resources.yaml
```

打开一个 Shell，进入 Pod 中运行的容器：

```
kubectl exec -it kubernetes-downwardapi-volume-example-2 -- sh
```

在 Shell 中，查看 cpu_limit 文件：

```
# cat /etc/podinfo/cpu_limit
```

你可以使用同样的命令查看 `cpu_request`、`mem_limit` 和 `mem_request` 文件。

Downward API 的能力

下面这些信息可以通过环境变量和 downwardAPI 卷提供给容器：

- 能通过 `fieldRef` 获得的：
 - `metadata.name` - Pod 名称
 - `metadata.namespace` - Pod 名字空间
 - `metadata.uid` - Pod 的 UID
 - `metadata.labels['<KEY>']` - Pod 标签 `<KEY>` 的值 (例如, `metadata.labels['mylabel']`)
 - `metadata.annotations['<KEY>']` - Pod 的注解 `<KEY>` 的值 (例如, `metadata.annotations['myannotation']`)
- 能通过 `resourceFieldRef` 获得的：
 - 容器的 CPU 约束值
 - 容器的 CPU 请求值
 - 容器的内存约束值
 - 容器的内存请求值
 - 容器的巨页限制值 (前提是启用了 DownwardAPIHugePages 特性门控)
 - 容器的巨页请求值 (前提是启用了 DownwardAPIHugePages 特性门控)
 - 容器的临时存储约束值
 - 容器的临时存储请求值

此外，以下信息可通过 downwardAPI 卷从 `fieldRef` 获得：

- `metadata.labels` - Pod 的所有标签，以 `label-key="escaped-label-value"` 格式显示，每行显示一个标签
- `metadata.annotations` - Pod 的所有注解，以 `annotation-key="escaped-annotation-value"` 格式显示，每行显示一个标签

以下信息可通过环境变量获得：

- `status.podIP` - 节点 IP
- `spec.serviceAccountName` - Pod 服务帐号名称, 版本要求 v1.4.0-alpha.3
- `spec.nodeName` - 节点名称, 版本要求 v1.4.0-alpha.3
- `status.hostIP` - 节点 IP, 版本要求 v1.7.0-alpha.1

说明：如果容器未指定 CPU 和内存限制，则 Downward API 默认将节点可分配值视为容器的 CPU 和内存限制。

投射键名到指定路径并且指定文件权限

你可以将键名投射到指定路径并且指定每个文件的访问权限。更多信息，请参阅 [Secrets](#)。

Downward API的动机

对于容器来说，有时候拥有自己的信息是很有用的，可避免与 Kubernetes 过度耦合。Downward API 使得容器使用自己或者集群的信息，而不必通过 Kubernetes 客户端或 API 服务器来获得。

一个例子是有一个现有的应用假定要用一个非常熟悉的环境变量来保存一个唯一标识。一种可能是给应用增加处理层，但这样是冗余和易出错的，而且它违反了低耦合的目标。更好的选择是使用 Pod 名称作为标识，把 Pod 名称注入这个环境变量中。

接下来

- [PodSpec](#)
- [Volume](#)
- [DownwardAPIVolumeSource](#)
- [DownwardAPIVolumeFile](#)
- [ResourceFieldSelector](#)

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 09, 2021 at 1:20 PM PST: [\[zh\] fix volume mount path \(d7d0131e1\)](#)

使用 Secret 安全地分发凭证

本文展示如何安全地将敏感数据（如密码和加密密钥）注入到 Pods 中。

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](#)

将 secret 数据转换为 base-64 形式

假设用户想要有两条 Secret 数据：用户名 my-app 和密码 39528\$vdg7Jb。首先使用 [Base64 编码](#) 将用户名和密码转化为 base-64 形式。下面是一个使用常用的 base64 程序的示例：

```
echo -n 'my-app' | base64  
echo -n '39528$vdg7Jb' | base64
```

结果显示 base-64 形式的用户名为 bXktYXBw，base-64 形式的密码为 Mzk1Mjgkd mRnN0pi。

注意： 使用你的操作系统所能信任的本地工具以降低使用外部工具的风险。

创建 Secret

这里是一个配置文件，可以用来创建存有用户名和密码的 Secret：

[pods/inject/secret.yaml](#)



```
apiVersion: v1  
kind: Secret  
metadata:  
  name: test-secret  
data:  
  username: bXktYXBw  
  password: Mzk1MjgkdmRnN0pi
```

1. 创建 Secret：

```
kubectl apply -f https://k8s.io/examples/pods/inject/secret.yaml
```

2. 查看 Secret 相关信息：

```
kubectl get secret test-secret
```

输出：

NAME	TYPE	DATA	AGE
test-secret	Opaque	2	1m

3. 查看 Secret 相关的更多详细信息：

```
kubectl describe secret test-secret
```

输出：

```
Name: test-secret
Namespace: default
Labels: <none>
Annotations: <none>

Type: Opaque

Data
=====
password: 13 bytes
username: 7 bytes
```

直接用 kubectl 创建 Secret

如果你希望略过 Base64 编码的步骤，你也可以使用 kubectl create secret 命令直接创建 Secret。例如：

```
kubectl create secret generic test-secret --from-literal='username=my-app' --
from-literal='password=39528$vdg7Jb'
```

这是一种更为方便的方法。前面展示的详细分解步骤有助于了解究竟发生了什么事情。

创建一个可以通过卷访问 secret 数据的 Pod

这里是一个可以用来创建 pod 的配置文件：

[pods/inject/secret-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
  volumeMounts:
    # name must match the volume name below
    - name: secret-volume
      mountPath: /etc/secret-volume
  # The secret data is exposed to Containers in the Pod through a Volume.
  volumes:
    - name: secret-volume
```

```
secret
  secretName: test-secret
```

1. 创建 Pod :

```
kubectl create -f secret-pod.yaml
```

2. 确认 Pod 正在运行 :

```
kubectl get pod secret-test-pod
```

输出 :

NAME	READY	STATUS	RESTARTS	AGE
secret-test-pod	1/1	Running	0	42m

3. 获取一个 shell 进入 Pod 中运行的容器 :

```
kubectl exec -it secret-test-pod -- /bin/bash
```

4. Secret 数据通过挂载在 /etc/secret-volume 目录下的卷暴露在容器中。

在 shell 中 , 列举 /etc/secret-volume 目录下的文件 :

```
ls /etc/secret-volume
```

输出包含两个文件 , 每个对应一个 Secret 数据条目 :

```
password username
```

5. 在 Shell 中 , 显示 username 和 password 文件的内容 :

```
# 在容器中 Shell 运行下面命令
echo "$(cat /etc/secret-volume/username)"
echo "$(cat /etc/secret-volume/password)"
```

输出为用户名和密码 :

```
my-app
39528$vdg7Jb
```

使用 Secret 数据定义容器变量

使用来自 Secret 中的数据定义容器变量

- 定义环境变量为 Secret 中的键值偶对 :

```
kubectl create secret generic backend-user --from-literal=backend-username='backend-admin'
```

- 在 Pod 规约中，将 Secret 中定义的值 backend-username 赋给 SECRET_USERNAME 环境变量

[pods/inject/pod-single-secret-env-variable.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: env-single-secret
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: backend-user
              key: backend-username
```

- 创建 Pod：

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-single-secret-env-variable.yaml
```

- 在 Shell 中，显示容器环境变量 SECRET_USERNAME 的内容：

```
kubectl exec -i -t env-single-secret -- /bin/sh -c 'echo $SECRET_USERNAME'
```

输出为：

```
backend-admin
```

使用来自多个 Secret 的数据定义环境变量

- 和前面的例子一样，先创建 Secret：

```
kubectl create secret generic backend-user --from-literal=backend-username='backend-admin'
```

```
kubectl create secret generic db-user --from-literal=db-username='db-admin'
```

- 在 Pod 规约中定义环境变量：

[pods/inject/pod-multiple-secret-env-variable.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: envvars-multiple-secrets
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: BACKEND_USERNAME
          valueFrom:
            secretKeyRef:
              name: backend-user
              key: backend-username
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-user
              key: db-username
```

- 创建 Pod :

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-multiple-secret-env-variable.yaml
```

- 在你的 Shell 中，显示容器环境变量的内容：

```
kubectl exec -i -t envvars-multiple-secrets -- /bin/sh -c 'env | grep _USERNAME'
```

输出：

```
DB_USERNAME=db-admin
BACKEND_USERNAME=backend-admin
```

将 Secret 中的所有键值偶对定义为环境变量

说明：此功能在 Kubernetes 1.6 版本之后可用。

- 创建包含多个键值偶对的 Secret：

```
kubectl create secret generic test-secret --from-literal=username='my-app'  
--from-literal=password='39528$vdg7Jb'
```

- 使用 envFrom 来将 Secret 中的所有数据定义为环境变量。 Secret 中的键名成为容器中的环境变量名：

[pods/inject/pod-secret-envFrom.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  name: envfrom-secret  
spec:  
  containers:  
    - name: envvars-test-container  
      image: nginx  
      envFrom:  
        - secretRef:  
            name: test-secret
```

- 创建 Pod：

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-secret-  
envFrom.yaml
```

- 在 Shell 中，显示环境变量 username 和 password 的内容：

```
kubectl exec -i -t envfrom-secret -- /bin/sh -c 'echo "username:  
$username\npassword: $password\n"'
```

输出为：

```
username: my-app  
password: 39528$vdg7Jb
```

参考

- [Secret](#)
- [Volume](#)
- [Pod](#)

接下来

- 进一步了解 [Secret](#)。
- 了解 [Volumes](#)。

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 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

运行应用

运行和管理无状态和有状态的应用程序。

[运行一个单实例有状态应用](#)

[运行一个有状态的应用程序](#)

[删除 StatefulSet](#)

[强制删除 StatefulSet 类型的 Pods](#)

[Pod 水平自动扩缩](#)

[Horizontal Pod Autoscaler 演练](#)

[为应用程序设置干扰预算 \(Disruption Budget \)](#)

[使用Deployment运行一个无状态应用](#)

[扩缩 StatefulSet](#)

运行一个单实例有状态应用

本文介绍在 Kubernetes 中如何使用 PersistentVolume 和 Deployment 运行一个单实例有状态应用。该应用是 MySQL.

教程目标

- 在你的环境中创建一个引用磁盘的 PersistentVolume
- 创建一个 MySQL Deployment.
- 在集群内以一个已知的 DNS 名称将 MySQL 暴露给其他 Pod

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.
- 您需要有一个带有默认[StorageClass](#)的动态持续卷供应程序，或者自己[静态的提供持久卷](#)来满足这里使用的[持久卷请求](#)。

部署 MySQL

你可以通过创建一个 Kubernetes Deployment 并使用 PersistentVolumeClaim 将其连接到 某已有的 PV 卷来运行一个有状态的应用。例如，这里的 YAML 描述的是一个运行 MySQL 的 Deployment，其中引用了 PVC 申领。文件为 /var/lib/mysql 定义了加载卷，并创建了一个 PVC 申领，寻找一个 20G 大小的卷。该申领可以通过现有的满足需求的卷来满足，也可以通过动态供应卷的机制来满足。

注意：在配置的 YAML 文件中定义密码的做法是不安全的。具体安全解决方案请参考 [Kubernetes Secrets](#).

[application/mysql/mysql-deployment.yaml](#)


```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
```

```
type: Recreate
template:
  metadata:
    labels:
      app: mysql
  spec:
    containers:
      - image: mysql:5.6
        name: mysql
        env:
          # Use secret in real usage
          - name: MYSQL_ROOT_PASSWORD
            value: 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
```

[application/mysql/mysql-pv.yaml](#)



```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
```

```
spec:  
  storageClassName: manual  
  accessModes:  
    - ReadWriteOnce  
resources:  
  requests:  
    storage: 20Gi
```

1. 部署 YAML 文件中定义的 PV 和 PVC :

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

2. 部署 YAML 文件中定义的 Deployment :

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

3. 展示 Deployment 相关信息:

```
kubectl describe deployment mysql
```

```
Name:           mysql  
Namespace:      default  
CreationTimestamp: Tue, 01 Nov 2016 11:18:45 -0700  
Labels:         app=mysql  
Annotations:    deployment.kubernetes.io/revision=1  
Selector:       app=mysql  
Replicas:      1 desired | 1 updated | 1 total | 0 available | 1 unavailable  
StrategyType:   Recreate  
MinReadySeconds: 0  
Pod Template:  
  Labels:     app=mysql  
  Containers:  
    mysql:  
      Image:    mysql:5.6  
      Port:     3306/TCP  
      Environment:  
        MYSQL_ROOT_PASSWORD: password  
      Mounts:  
        /var/lib/mysql from mysql-persistent-storage (rw)  
  Volumes:  
    mysql-persistent-storage:  
      Type:     PersistentVolumeClaim (a reference to a PersistentVolumeClaim  
      in the same namespace)  
      ClaimName: mysql-pv-claim  
      ReadOnly:  false  
      Conditions:
```

Type	Status	Reason					
Available	False	MinimumReplicasUnavailable					
Progressing	True	ReplicaSetUpdated					
OldReplicaSets:	<none>						
NewReplicaSet:	mysql-63082529	(1/1 replicas created)					
Events:							
Type	Reason	Message	FirstSeen	LastSeen	Count	From	SubobjectPath
Normal	ScalingReplicaSet	Scaled up replica set mysql-63082529 to 1	33s	33s	1	{deployment-controller }	

4. 列举出 Deployment 创建的 pods:

```
kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-63082529-2z3ki	1/1	Running	0	3m

5. 查看 PersistentVolumeClaim :

```
kubectl describe pvc mysql-pv-claim
```

Name:	mysql-pv-claim
Namespace:	default
StorageClass:	
Status:	Bound
Volume:	mysql-pv-volume
Labels:	<none>
Annotations:	pv.kubernetes.io/bind-completed=yes pv.kubernetes.io/bound-by-controller=yes
Capacity:	20Gi
Access Modes:	RWO
Events:	<none>

访问 MySQL 实例

前面 YAML 文件中创建了一个允许集群内其他 Pod 访问的数据库服务。该服务中选项 clusterIP: None 让服务 DNS 名称直接解析为 Pod 的 IP 地址。当在一个服务下只有一个 Pod 并且不打算增加 Pod 的数量这是最好的。

运行 MySQL 客户端以连接到服务器:

```
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword
```

此命令在集群内创建一个新的 Pod 并运行 MySQL 客户端，并通过 Service 连接到服务器。如果连接成功，你就知道有状态的 MySQL 数据库正处于运行状态。

```
Waiting for pod default/mysql-client-274442439-zyp6i to be running, status is Pending, pod ready: false  
If you don't see a command prompt, try pressing enter.
```

```
mysql>
```

更新

Deployment 中镜像或其他部分同往常一样可以通过 kubectl apply 命令更新。以下是特定于有状态应用的一些注意事项：

- 不要对应用进行规模扩缩。这里的设置仅适用于单实例应用。下层的 PersistentVolume 仅只能挂载到一个 Pod 上。对于集群级有状态应用，请参考 [StatefulSet 文档](#)。
- 在 Deployment 的 YAML 文件中使用 strategy: type: Recreate。该选项指示 Kubernetes 不 使用滚动升级。滚动升级无法工作，因为这里一次不能 运行多个 Pod。在使用更新的配置文件创建新的 Pod 前，Recreate 策略将 保证先停止第一个 Pod。

删除 Deployment

通过名称删除部署的对象：

```
kubectl delete deployment,svc mysql  
kubectl delete pvc mysql-pv-claim  
kubectl delete pv mysql-pv-volume
```

如果通过手动的方式供应 PersistentVolume，那么也需要手动删除它以释放下层资源。如果是用动态供应方式创建的 PersistentVolume，在删除 PersistentVolumeClaim 后 PersistentVolume 将被自动删除。一些存储服务（比如 EBS 和 PD）也会在 PersistentVolume 被删除时自动回收下层资源。

接下来

- 欲进一步了解 Deployment 对象，请参考 [Deployment 对象](#)
- 进一步了解[部署应用](#)
- 参阅 [kubectl run 文档](#)
- 参阅[卷和持久卷](#)

反馈

此页是否对您有帮助？

是否

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

最后修改 August 11, 2020 at 4:35 PM PST: [\[zh\] Rework stateful application tasks \(e9c038050\)](#)

运行一个有状态的应用程序

本页展示如何使用 [StatefulSet](#) 控制器运行一个有状态的应用程序。此例是一主多从、异步复制的 MySQL 集群。

说明：这不是生产环境下配置。 尤其注意，MySQL 设置都使用的是不安全的默认值，这是因为我们想把重点放在 Kubernetes 中运行有状态应用程序的一般模式上。

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.

您需要有一个带有默认[StorageClass](#)的动态持续卷供应程序，或者自己[静态的提供持久卷](#)来满足这里使用的[持久卷请求](#)。

- 本教程假定你熟悉 [PersistentVolumes](#) 与 [StatefulSet](#), 以及其他核心概念，例如 [Pod](#)、[服务](#) 与 [ConfigMap](#).
- 熟悉 MySQL 会有所帮助，但是本教程旨在介绍对其他系统应该有用的常规模式。

教程目标

- 使用 StatefulSet 控制器部署多副本 MySQL 拓扑架构。
- 发送 MySQL 客户端请求
- 观察对宕机的抵抗力
- 扩缩 StatefulSet 的规模

部署 MySQL

MySQL 示例部署包含一个 ConfigMap、两个 Service 与一个 StatefulSet。

ConfigMap

使用以下的 YAML 配置文件创建 ConfigMap：

[application/mysql/mysql-configmap.yaml](#)


```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

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

这个 ConfigMap 提供 my.cnf 覆盖设置，使你可以独立控制 MySQL 主服务器和从服务器的配置。在这里，你希望主服务器能够将复制日志提供给从服务器，并且希望从服务器拒绝任何不是通过 复制进行的写操作。

ConfigMap 本身没有什么特别之处，因而也不会出现不同部分应用于不同的 Pod 的情况。每个 Pod 都会在初始化时基于 StatefulSet 控制器提供的信息决定要查看的部分。

服务

使用以下 YAML 配置文件创建服务：

[application/mysql/mysql-services.yaml](#)


```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
```

```

  app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the master: mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql

```

kubectl apply -f https://k8s.io/examples/application/mysql/mysql-services.yaml

这个无头服务给 StatefulSet 控制器为集合中每个 Pod 创建的 DNS 条目提供了一个宿主。因为服务名为 mysql，所以可以通过在同一 Kubernetes 集群和名字中的任何其他 Pod 内解析 <Pod 名称>.mysql 来访问 Pod。

客户端服务称为 mysql-read，是一种常规服务，具有其自己的集群 IP。该集群 IP 在报告就绪的所有MySQL Pod 之间分配连接。可能的端点集合包括 MySQL 主节点和所有从节点。

请注意，只有读查询才能使用负载平衡的客户端服务。因为只有一个 MySQL 主服务器，所以客户端应直接连接到 MySQL 主服务器 Pod（通过其在无头服务中的 DNS 条目）以执行写入操作。

StatefulSet

最后，使用以下 YAML 配置文件创建 StatefulSet：

[application/mysql/mysql-statefulset.yaml](#)

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
        - name: init-mysql
          image: mysql:5.7
          command:
            - bash
            - "-c"
            - |
              set -ex
              # Generate mysql server-id from pod ordinal index.
              [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
              ordinal=${BASH_REMATCH[1]}
              echo [mysqld] > /mnt/conf.d/server-id.cnf
              # Add an offset to avoid reserved server-id=0 value.
              echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
              # Copy appropriate conf.d files from config-map to emptyDir.
              if [[ $ordinal -eq 0 ]]; then
                cp /mnt/config-map/master.cnf /mnt/conf.d/
              else
                cp /mnt/config-map/slave.cnf /mnt/conf.d/
              fi
      volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
          mountPath: /mnt/config-map
        - name: clone-mysql
          image: gcr.io/google-samples/xtrabackup:1.0
          command:
            - bash
            - "-c"
            - |
```

```

set -ex
# Skip the clone if data already exists.
[[ -d /var/lib/mysql/mysql ]] && exit 0
# Skip the clone on master (ordinal index 0).
[[ `hostname` =~ -([0-9]+)$ ]] || exit 1
ordinal=${BASH_REMATCH[1]}
[[ $ordinal -eq 0 ]] && exit 0
# Clone data from previous peer.
ncat --recv-only mysql-$((ordinal-1)).mysql 3307 | xbstream -x -C /var/lib/
mysql
# Prepare the backup.
xtrabackup --prepare --target-dir=/var/lib/mysql
volumeMounts:
- name: data
  mountPath: /var/lib/mysql
  subPath: mysql
- name: conf
  mountPath: /etc/mysql/conf.d
containers:
- name: mysql
  image: mysql:5.7
  env:
  - name: MYSQL_ALLOW_EMPTY_PASSWORD
    value: "1"
  ports:
  - name: mysql
    containerPort: 3306
  volumeMounts:
  - name: data
    mountPath: /var/lib/mysql
    subPath: mysql
  - name: conf
    mountPath: /etc/mysql/conf.d
  resources:
    requests:
      cpu: 500m
      memory: 1Gi
  livenessProbe:
    exec:
      command: ["mysqladmin", "ping"]
  initialDelaySeconds: 30
  periodSeconds: 10
  timeoutSeconds: 5
  readinessProbe:
    exec:
      # Check we can execute queries over TCP (skip-networking is off).

```

```

command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
initialDelaySeconds: 5
periodSeconds: 2
timeoutSeconds: 1
- name: xtrabackup
image: gcr.io/google-samples/xtrabackup:1.0
ports:
- name: xtrabackup
  containerPort: 3307
command:
- bash
- "-c"
- |
  set -ex
  cd /var/lib/mysql

# Determine binlog position of cloned data, if any.
if [[ -f xtrabackup_slave_info && "x$(<xtrabackup_slave_info)" != "x" ]]; then
  # XtraBackup already generated a partial "CHANGE MASTER TO" query
  # because we're cloning from an existing slave. (Need to remove the
tailoring semicolon!)
  cat xtrabackup_slave_info | sed -E 's/;$/g' > change_master_to.sql.in
  # Ignore xtrabackup_binlog_info in this case (it's useless).
  rm -f xtrabackup_slave_info xtrabackup_binlog_info
elif [[ -f xtrabackup_binlog_info ]]; then
  # We're cloning directly from master. Parse binlog position.
  [[ `cat xtrabackup_binlog_info` =~ ^(.*)[[[:space:]]+(.*)$ ]] || exit 1
  rm -f xtrabackup_binlog_info xtrabackup_slave_info
  echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}'\ \
        MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
fi

# Check if we need to complete a clone by starting replication.
if [[ -f change_master_to.sql.in ]]; then
  echo "Waiting for mysqld to be ready (accepting connections)"
  until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

  echo "Initializing replication from clone position"
  mysql -h 127.0.0.1 \
    -e "$(cat change_master_to.sql.in), \
          MASTER_HOST='mysql-0.mysql', \
          MASTER_USER='root', \
          MASTER_PASSWORD='', \
          MASTER_CONNECT_RETRY=10; \
          START SLAVE;" || exit 1
# In case of container restart, attempt this at-most-once.

```

```

mv change_master_to.sql.in change_master_to.sql.orig
fi

# Start a server to send backups when requested by peers.
exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
"xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 --
user=root"

volumeMounts:
- name: data
  mountPath: /var/lib/mysql
  subPath: mysql
- name: conf
  mountPath: /etc/mysql/conf.d

resources:
  requests:
    cpu: 100m
    memory: 100Mi

volumes:
- name: conf
  emptyDir: {}
- name: config-map
  configMap:
    name: mysql

volumeClaimTemplates:
- metadata:
    name: data
  spec:
    accessModes: ["ReadWriteOnce"]
  resources:
    requests:
      storage: 10Gi

```

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

你可以通过运行以下命令查看启动进度：

```
kubectl get pods -l app=mysql --watch
```

一段时间后，你应该看到所有 3 个 Pod 进入 Running 状态：

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	2m
mysql-1	2/2	Running	0	1m
mysql-2	2/2	Running	0	1m

输入 **Ctrl+C** 结束 watch 操作。如果你看不到任何进度，确保已启用[前提条件](#) 中提到的动态 PersistentVolume 预配器。

此清单使用多种技术来管理作为 StatefulSet 的一部分的有状态 Pod。下一节重点介绍其中的一些技巧，以解释 StatefulSet 创建 Pod 时发生的状况。

了解有状态的 Pod 初始化

StatefulSet 控制器按序数索引顺序地每次启动一个 Pod。它一直等到每个 Pod 报告就绪才再启动下一个 Pod。

此外，控制器为每个 Pod 分配一个唯一、稳定的名称，形如 <statefulset 名称>-<序数索引>，其结果是 Pods 名为 mysql-0、mysql-1 和 mysql-2。

上述 StatefulSet 清单中的 Pod 模板利用这些属性来执行 MySQL 副本的有序启动。

生成配置

在启动 Pod 规约中的任何容器之前，Pod 首先按顺序运行所有的 [Init 容器](#)。

第一个名为 init-mysql 的 Init 容器根据序号索引生成特殊的 MySQL 配置文件。

该脚本通过从 Pod 名称的末尾提取索引来确定自己的序号索引，而 Pod 名称由 hostname 命令返回。然后将序数（带有数字偏移量以避免保留值）保存到 MySQL conf.d 目录中的文件 server-id.cnf。这一操作将 StatefulSet 所提供的唯一、稳定的标识转换为 MySQL 服务器的 ID，而这些 ID 也是需要唯一性、稳定性保证的。

通过将内容复制到 conf.d 中，init-mysql 容器中的脚本也可以应用 ConfigMap 中的 master.cnf 或 slave.cnf。由于示例部署结构由单个 MySQL 主节点和任意数量的从节点组成，因此脚本仅将序数 0 指定为主节点，而将其他所有节点指定为从节点。

与 StatefulSet 控制器的 [部署顺序保证](#) 相结合，可以确保 MySQL 主服务器在创建从服务器之前已准备就绪，以便它们可以开始复制。

克隆现有数据

通常，当新 Pod 作为从节点加入集合时，必须假定 MySQL 主节点可能已经有数据。还必须假设复制日志可能不会一直追溯到时间的开始。

这些保守的假设是允许正在运行的 StatefulSet 随时间扩大和缩小而不是固定在其初始大小的关键。

第二个名为 clone-mysql 的 Init 容器，第一次在带有空 PersistentVolume 的从属 Pod 上启动时，会在从属 Pod 上执行克隆操作。这意味着它将从另一个运行中的 Pod 复制所有现有数据，使此其本地状态足够一致，从而可以开始从主服务器复制。

MySQL 本身不提供执行此操作的机制，因此本示例使用了一种流行的开源工具 Percona XtraBackup。在克隆期间，源 MySQL 服务器性能可能会受到影响。为了最大程度地减少对 MySQL 主节点的影响，该脚本指示每个 Pod 从序号较低的 Pod 中克隆。可以这样做的原因是 StatefulSet 控制器始终确保在启动 Pod N + 1 之前 Pod N 已准备就绪。

开始复制

Init 容器成功完成后，应用容器将运行。 MySQL Pod 由运行实际 mysqld 服务的 mysql 容器和充当 [辅助工具](#) 的 xtrabackup 容器组成。

xtrabackup sidecar 容器查看克隆的数据文件，并确定是否有必要在从服务器上初始化 MySQL 复制。如果是这样，它将等待 mysqld 准备就绪，然后使用从 XtraBackup 克隆文件中提取的复制参数执行 CHANGE MASTER TO 和 START SLAVE 命令。

一旦从服务器开始复制后，它会记住其 MySQL 主服务器，并且如果服务器重新启动或连接中断也会自动重新连接。另外，因为从服务器会以其稳定的 DNS 名称查找主服务器（mysql-0.mysql），即使由于重新调度而获得新的 Pod IP，他们也会自动找到主服务器。

最后，开始复制后，xtrabackup 容器监听来自其他 Pod 的连接，处理其数据克隆请求。如果 StatefulSet 扩大规模，或者下一个 Pod 失去其 PersistentVolumeClaim 并需要重新克隆，则此服务器将无限期保持运行。

发送客户端请求

你可以通过运行带有 mysql:5.7 镜像的临时容器并运行 mysql 客户端二进制文件，将测试查询发送到 MySQL 主服务器（主机名 mysql-0.mysql）。

```
kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never -- \
  mysql -h mysql-0.mysql <<EOF
CREATE DATABASE test;
CREATE TABLE test.messages (message VARCHAR(250));
INSERT INTO test.messages VALUES ('hello');
EOF
```

使用主机名 mysql-read 将测试查询发送到任何报告为就绪的服务器：

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never -- \
  mysql -h mysql-read -e "SELECT * FROM test.messages"
```

你应该获得如下输出：

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready:
false
+-----+
| message |
+-----+
| hello   |
+-----+
pod "mysql-client" deleted
```

为了演示 mysql-read 服务在服务器之间分配连接，你可以在循环中运行 SELECT @@server_id：

```
kubectl run mysql-client-loop --image=mysql:5.7 -i -t --rm --restart=Never --\n  bash -ic "while sleep 1; do mysql -h mysql-read -e 'SELECT\n    @@server_id,NOW(); done"
```

你应该看到报告的 @@server_id 发生随机变化，因为每次尝试连接时都可能选择了不同的端点：

```
+-----+-----+\n| @@server_id | NOW() |\n+-----+-----+\n| 100 | 2006-01-02 15:04:05 |\n+-----+-----+\n+-----+-----+\n| @@server_id | NOW() |\n+-----+-----+\n| 102 | 2006-01-02 15:04:06 |\n+-----+-----+\n+-----+-----+\n| @@server_id | NOW() |\n+-----+-----+\n| 101 | 2006-01-02 15:04:07 |\n+-----+-----+
```

要停止循环时可以按 **Ctrl+C**，但是让它在另一个窗口中运行非常有用，这样你就可以看到以下步骤的效果。

模拟 Pod 和 Node 的宕机时间

为了证明从从节点缓存而不是单个服务器读取数据的可用性提高，请在使 Pod 退出 Ready 状态时，保持上述 SELECT @@server_id 循环一直运行。

破坏就绪态探测

mysql 容器的 [就绪态探测](#) 运行命令 `mysql -h 127.0.0.1 -e 'SELECT 1'`，以确保服务器已启动并能够执行查询。

迫使就绪态探测失败的一种方法就是中止该命令：

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql /usr/bin/mysql.off
```

此命令会进入 Pod mysql-2 的实际容器文件系统，重命名 mysql 命令，导致就绪态探测无法找到它。几秒钟后，Pod 会报告其中一个容器未就绪。你可以通过运行以下命令进行检查：

```
kubectl get pod mysql-2
```

在 READY 列中查找 1/2：

NAME	READY	STATUS	RESTARTS	AGE
mysql-2	1/2	Running	0	3m

此时，你应该会看到 `SELECT @@server_id` 循环继续运行，尽管它不再报告 102。回想一下，`init-mysql` 脚本将 `server-id` 定义为 `100 + $ordinal`，因此服务器 ID 102 对应于 Pod `mysql-2`。

现在修复 Pod，几秒钟后它应该重新出现在循环输出中：

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql.off /usr/bin/mysql
```

删除 Pods

如果删除了 Pod，则 StatefulSet 还会重新创建 Pod，类似于 ReplicaSet 对无状态 Pod 所做的操作。

```
kubectl delete pod mysql-2
```

StatefulSet 控制器注意到不再存在 `mysql-2` Pod，于是创建一个具有相同名称并链接到相同 PersistentVolumeClaim 的新 Pod。你应该看到服务器 ID 102 从循环输出中消失了一段时间，然后又自行出现。

腾空节点

如果你的 Kubernetes 集群具有多个节点，则可以通过发出以下 [drain](#) 命令来模拟节点停机（就好像节点在被升级）。

首先确定 MySQL Pod 之一在哪个节点上：

```
kubectl get pod mysql-2 -o wide
```

节点名称应显示在最后一列中：

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mysql-2	2/2	Running	0	15m	10.244.5.27	kubernetes-node-9l2t

然后通过运行以下命令腾空节点，该命令将其保护起来，以使新的 Pod 不能调度到该节点，然后逐出所有现有的 Pod。将 <节点名称> 替换为在上一步中找到的节点名称。

这可能会影响节点上的其他应用程序，因此最好 **仅在测试集群中执行此操作**。

```
kubectl drain <节点名称> --force --delete-local-data --ignore-daemonsets
```

现在，你可以看到 Pod 被重新调度到其他节点上：

```
kubectl get pod mysql-2 -o wide --watch
```

它看起来应该像这样：

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
mysql-2	2/2	Terminating	0	15m	10.244.1.56	kubernetes-

```
node-9l2t
[...]
mysql-2 0/2 Pending 0 0s <none> kubernetes-node-fjlm
mysql-2 0/2 Init:0/2 0 0s <none> kubernetes-node-fjlm
mysql-2 0/2 Init:1/2 0 20s 10.244.5.32 kubernetes-node-fjlm
mysql-2 0/2 PodInitializing 0 21s 10.244.5.32 kubernetes-node-fjlm
mysql-2 1/2 Running 0 22s 10.244.5.32 kubernetes-node-fjlm
mysql-2 2/2 Running 0 30s 10.244.5.32 kubernetes-node-fjlm
```

再次，你应该看到服务器 ID 102 从 SELECT @@server_id 循环输出中消失一段时间，然后自行出现。

现在去掉节点保护（Uncordon），使其恢复为正常模式：

```
kubectl uncordon <节点名称>
```

扩展从节点数量

使用 MySQL 复制，你可以通过添加从节点来扩展读取查询的能力。 使用 StatefulSet，你可以使用单个命令执行此操作：

```
kubectl scale statefulset mysql --replicas=5
```

查看新的 Pod 的运行情况：

```
kubectl get pods -l app=mysql --watch
```

一旦 Pod 启动，你应该看到服务器 IDs 103 和 104 开始出现在 SELECT @@server_id 循环输出中。

你还可以验证这些新服务器在存在之前已添加了数据：

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never -- \
  mysql -h mysql-3.mysql -e "SELECT * FROM test.messages"
```

```
Waiting for pod default/mysql-client to be running, status is Pending, pod ready:
false
+-----+
| message |
+-----+
| hello   |
+-----+
pod "mysql-client" deleted
```

向下缩容操作也是很平滑的：

```
kubectl scale statefulset mysql --replicas=3
```

但是请注意，按比例扩大会自动创建新的 PersistentVolumeClaims，而按比例缩小不会自动删除这些 PVC。这使你可以选择保留那些初始化的 PVC，以更快地进行缩放，或者在删除它们之前提取数据。

你可以通过运行以下命令查看此信息：

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

这表明，尽管将 StatefulSet 缩小为3，所有5个 PVC 仍然存在：

NAME	STATUS	VOLUME	CAPACITY
ACCESSMODES	AGE		
data-mysql-0	Bound	pvc-8acbf5dc-b103-11e6-93fa-42010a800002	10Gi
	RWO	20m	
data-mysql-1	Bound	pvc-8ad39820-b103-11e6-93fa-42010a800002	10Gi
10Gi	RWO	20m	
data-mysql-2	Bound	pvc-8ad69a6d-b103-11e6-93fa-42010a800002	10Gi
10Gi	RWO	20m	
data-mysql-3	Bound	pvc-50043c45-b1c5-11e6-93fa-42010a800002	10Gi
RWO	2m		
data-mysql-4	Bound	pvc-500a9957-b1c5-11e6-93fa-42010a800002	10Gi
10Gi	RWO	2m	

如果你不打算重复使用多余的 PVC，则可以删除它们：

```
kubectl delete pvc data-mysql-3  
kubectl delete pvc data-mysql-4
```

清理现场

1. 通过在终端上按 **Ctrl+C** 取消 SELECT @@server_id 循环，或从另一个终端运行以下命令：

```
kubectl delete pod mysql-client-loop --now
```

1. 删除 StatefulSet。这也会开始终止 Pod。

```
kubectl delete statefulset mysql
```

1. 验证 Pod 消失。他们可能需要一些时间才能完成终止。

```
kubectl get pods -l app=mysql
```

当上述命令返回如下内容时，你就知道 Pod 已终止：

```
No resources found.
```

1. 删除 ConfigMap、Services 和 PersistentVolumeClaims。

```
kubectl delete configmap,service,pvc -l app=mysql
```

1. 如果你手动供应 PersistentVolume，则还需要手动删除它们，并释放下层资源。如果你使用了动态预配器，当得知你删除 PersistentVolumeClaims 时，它将自动删除 PersistentVolumes。一些动态预配器（例如用于 EBS 和 PD 的预配器）也会在删除 PersistentVolumes 时释放下层资源。

接下来

- 进一步了解[为 StatefulSet 扩缩容](#).
- 进一步了解[调试 StatefulSet](#).
- 进一步了解[删除 StatefulSet](#).
- 进一步了解[强制删除 StatefulSet Pods](#).
- 在[Helm Charts 仓库](#)中查找其他有状态的应用程序示例。

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 23, 2020 at 1:10 PM PST: [\[zh\] Sync changes from English site \(12\) \(81bc52053\)](#)

删除 StatefulSet

本任务展示如何删除 StatefulSet。

准备开始

- 本任务假设在你的集群上已经运行了由 StatefulSet 创建的应用。

删除 StatefulSet

你可以像删除 Kubernetes 中的其他资源一样删除 StatefulSet：使用 kubectl delete 命令，并按文件或者名字指定 StatefulSet。

```
kubectl delete -f <file.yaml>
```

```
kubectl delete statefulsets <statefulset 名称>
```

删除 StatefulSet 之后，你可能需要单独删除关联的无头服务。

```
kubectl delete service <服务名称>
```

通过 kubectl 删除 StatefulSet 会将其缩容为 0，因此删除属于它的所有 Pod。如果你只想删除 StatefulSet 而不删除 Pod，使用 --cascade=false。

```
kubectl delete -f <file.yaml> --cascade=false
```

通过将 --cascade=false 传递给 kubectl delete，在删除 StatefulSet 对象之后，StatefulSet 管理的 Pod 会被保留下。如果 Pod 具有标签 app=myapp，则可以按照如下方式删除它们：

```
kubectl delete pods -l app=myapp
```

持久卷

删除 StatefulSet 管理的 Pod 并不会删除关联的卷。这是为了确保你有机会在删除卷之前从卷中复制数据。在 Pod 离开[终止状态](#)后删除 PVC 可能会触发删除背后的 PV 持久卷，具体取决于存储类和回收策略。永远不要假定在 PVC 删除后仍然能够访问卷。

说明：删除 PVC 时要谨慎，因为这可能会导致数据丢失。

完全删除 StatefulSet

要简单地删除 StatefulSet 中的所有内容，包括关联的 pods，你可能需要运行一系列类似于以下内容的命令：

```
grace=$(kubectl get pods <stateful-set-pod> --template '{{.spec.terminationGracePeriodSeconds}}')  
kubectl delete statefulset -l app=myapp  
sleep $grace  
kubectl delete pvc -l app=myapp
```

在上面的例子中，Pod 的标签为 app=myapp；适当地替换你自己的标签。

强制删除 StatefulSet 的 Pod

如果你发现 StatefulSet 的某些 Pod 长时间处于 'Terminating' 或者 'Unknown' 状态，则可能需要手动干预以强制从 API 服务器中删除这些 Pod。这是一项有点危险的任务。详细信息请阅读[删除 StatefulSet 类型的 Pods](#)。

接下来

进一步了解[强制删除 StatefulSet 的 Pods](#)。

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 14, 2020 at 11:36 PM PST: [Update delete-stateful-set.md \(be12af716\)](#)

强制删除 StatefulSet 类型的 Pods

本文介绍了如何删除 [StatefulSet](#) 管理的 Pods，并且解释了这样操作时需要记住的注意事项。

准备开始

- 这是一项相当高级的任务，并且可能会违反 StatefulSet 固有的某些属性。
- 继续任务之前，请熟悉下面列举的注意事项。

StatefulSet 注意事项

在 StatefulSet 的正常操作中，**永远不需要** 强制删除 StatefulSet 管理的 Pod。[StatefulSet 控制器](#) 负责创建、扩缩和删除 StatefulSet 管理的 Pods。它尝试确保指定数量的从序数 0 到 N-1 的 Pod 处于活跃状态并准备就绪。StatefulSet 确保在任何时候，集群中最多只有一个具有给定标识的 Pod。这就是所谓的由 StatefulSet 提供的 *最多一个 (At Most One) *的语义。

应谨慎进行手动强制删除操作，因为它可能会违反 StatefulSet 固有的至多一个的语义。StatefulSets 可用于运行分布式和集群级的应用，这些应用需要稳定的网络标识和可靠的存储。这些应用通常配置为具有固定标识固定数量的成员集合。具有相同身份的多个成员可能是灾难性的，并且可能导致数据丢失（例如：票选系统中的脑裂场景）。

删除 Pods

你可以使用下面的命令执行体面地删除 Pod:

```
kubectl delete pods <pod>
```

为了让上面操作能够体面地终止 Pod，Pod **一定不能** 设置 pod.Spec.TerminationGracePeriodSeconds 为 0。将 pod.Spec.TerminationGracePeriodSeconds 设置为 0s 的做法是不安全的，强烈建议 StatefulSet 类型的 Pod 不要使用。体面删除是安全的，并且会在 kubelet 从 API 服务器中删除资源名称之前确保 [体面地结束 pod](#)。

Kubernetes (1.5 版本或者更新版本) 不会因为一个节点无法访问而删除 Pod。在无法访问的节点上运行的 Pod 在 [超时](#) 后会进入 'Terminating' 或者 'Unknown' 状态。当用户尝试体面地删除无法访问的节点上的 Pod 时 Pod 也可能会进入这些状态。从 API 服务器上删除处于这些状态 Pod 的仅有可行方法如下：

- 删除 Node 对象（要么你来删除，要么[节点控制器](#)来删除）
- 无响应节点上的 kubelet 开始响应，杀死 Pod 并从 API 服务器上移除 Pod 对象

- 用户强制删除 pod

推荐使用第一种或者第二种方法。如果确认节点已经不可用了（比如，永久断开网络、断电等），则应删除 Node 对象。如果节点遇到网裂问题，请尝试解决该问题或者等待其解决。当网裂愈合时，kubelet 将完成 Pod 的删除并从 API 服务器上释放其名字。

通常，Pod 一旦不在节点上运行，或者管理员删除了节点，系统就会完成其删除动作。你也可以通过强制删除 Pod 来绕过这一机制。

强制删除

强制删除**不会**等待来自 kubelet 对 Pod 已终止的确认消息。无论强制删除是否成功杀死了 Pod，它都会立即从 API 服务器中释放该名字。这将让 StatefulSet 控制器创建一个具有相同标识的替身 Pod；因而可能导致正在运行 Pod 的重复，并且如果所述 Pod 仍然可以与 StatefulSet 的成员通信，则将违反 StatefulSet 所要保证的 最多一个的语义。

当你强制删除 StatefulSet 类型的 Pod 时，你要确保有问题的 Pod 不会再和 StatefulSet 管理的其他 Pod 通信并且可以安全地释放其名字以便创建替代 Pod。

如果要使用 kubectl 1.5 以上版本强制删除 Pod，请执行下面命令：

```
kubectl delete pods <pod> --grace-period=0 --force
```

如果你使用 kubectl 的 1.4 以下版本，则应省略 --force 选项：

```
kubectl delete pods <pod> --grace-period=0
```

如果在这些命令后 Pod 仍处于 Unknown 状态，请使用以下命令从集群中删除 Pod：

```
kubectl patch pod <pod> -p '{"metadata":{"finalizers":null}}'
```

请始终谨慎地执行强制删除 StatefulSet 类型的 pods，并完全了解所涉及的风险。

接下来

进一步了解[调试 StatefulSet](#)。

反馈

此页是否对您有帮助？

是 否

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

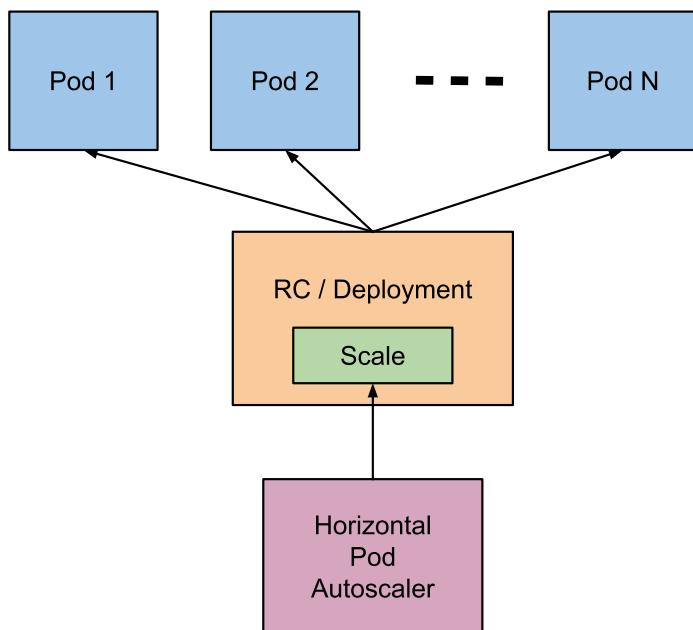
最后修改 August 11, 2020 at 4:05 PM PST: [\[zh\] Tidy up and fix links in tasks section \(3/10\) \(1d3f4449a\)](#)

Pod 水平自动扩缩

Pod 水平自动扩缩 (Horizontal Pod Autoscaler) 可以基于 CPU 利用率自动扩缩 ReplicationController、Deployment、ReplicaSet 和 StatefulSet 中的 Pod 数量。除了 CPU 利用率，也可以基于其他应用程序提供的[自定义度量指标](#)来执行自动扩缩。Pod 自动扩缩不适用于无法扩缩的对象，比如 DaemonSet。

Pod 水平自动扩缩特性由 Kubernetes API 资源和控制器实现。资源决定了控制器的行为。控制器会周期性的调整副本控制器或 Deployment 中的副本数量，以使得 Pod 的平均 CPU 利用率与用户所设定的目标值匹配。

Pod 水平自动扩缩工作机制



Pod 水平自动扩缩器的实现是一个控制回路，由控制器管理器的 `--horizontal-pod-autoscaler-sync-period` 参数指定周期（默认值为 15 秒）。

每个周期内，控制器管理器根据每个 HorizontalPodAutoscaler 定义中指定的指标查询资源利用率。控制器管理器可以从资源度量指标 API (按 Pod 统计的资源用量) 和自定义度量指标 API (其他指标) 获取度量值。

- 对于按 Pod 统计的资源指标 (如 CPU)，控制器从资源指标 API 中获取每一个 HorizontalPodAutoscaler 指定的 Pod 的度量值，如果设置了目标使用率，控制器获取每个 Pod 中的容器资源使用情况，并计算资源使用率。如果设置了

`target` 值，将直接使用原始数据（不再计算百分比）。接下来，控制器根据平均的资源使用率或原始值计算出扩缩的比例，进而计算出目标副本数。

需要注意的是，如果 Pod 某些容器不支持资源采集，那么控制器将不会使用该 Pod 的 CPU 使用率。下面的[算法细节](#)章节将会介绍详细的算法。

- 如果 Pod 使用自定义指示，控制器机制与资源指标类似，区别在于自定义指标只使用原始值，而不是使用率。
- 如果 Pod 使用对象指标和外部指标（每个指标描述一个对象信息）。这个指标将直接根据目标设定值相比较，并生成一个上面提到的扩缩比例。在 autoscaling/v2beta2 版本 API 中，这个指标也可以根据 Pod 数量平分后再计算。

通常情况下，控制器将从一系列的聚合 API (metrics.k8s.io、custom.metrics.k8s.io 和 external.metrics.k8s.io) 中获取度量值。metrics.k8s.io API 通常由 Metrics 服务器（需要额外启动）提供。可以从 [metrics-server](#) 获取更多信息。另外，控制器也可以直接从 Heapster 获取指标。

说明：

FEATURE STATE: Kubernetes 1.11 [deprecated]

自 Kubernetes 1.11 起，从 Heapster 获取指标特性已废弃。

关于指标 API 更多信息，请参考[度量值指标 API 的支持](#)。

自动扩缩控制器使用 scale 子资源访问相应可支持扩缩的控制器（如副本控制器、Deployment 和 ReplicaSet）。scale 是一个可以动态设定副本数量和检查当前状态的接口。关于 scale 子资源的更多信息，请参考[这里](#)。

算法细节

从最基本的角度来看，Pod 水平自动扩缩控制器根据当前指标和期望指标来计算扩缩比例。

期望副本数 = $\text{ceil}[\text{当前副本数} * (\text{当前指标} / \text{期望指标})]$

例如，当前度量值为 200m，目标设定值为 100m，那么由于 $200.0/100.0 == 2.0$ ，副本数量将会翻倍。如果当前指标为 50m，副本数量将会减半，因为 $50.0/100.0 == 0.5$ 。如果计算出的扩缩比例接近 1.0（根据--horizontal-pod-autoscaler-tolerance 参数全局配置的容忍值，默认为 0.1），将会放弃本次扩缩。

如果 HorizontalPodAutoscaler 指定的是 targetAverageValue 或 targetAverageUtilization，那么将会把指定 Pod 度量值的平均值做为 currentMetricValue。然而，在检查容忍度和决定最终扩缩值前，我们仍然会把那些无法获取指标的 Pod 统计进去。

所有被标记了删除时间戳（Pod 正在关闭过程中）的 Pod 和失败的 Pod 都会被忽略。

如果某个 Pod 缺失度量值，它将被搁置，只在最终确定扩缩数量时再考虑。

当使用 CPU 指标来扩缩时，任何还未就绪（例如还在初始化）状态的 Pod 或最近的指标度量值采集于就绪状态前的 Pod，该 Pod 也会被搁置。

由于受技术限制，Pod 水平扩缩控制器无法准确的知道 Pod 什么时候就绪，也就无法决定是否暂时搁置该 Pod。--horizontal-pod-autoscaler-initial-readiness-delay 参数（默认为 30s）用于设置 Pod 准备时间，在此时间内的 Pod 统统被认为未就绪。--horizontal-pod-autoscaler-cpu-initialization-period 参数（默认为 5 分钟）用于设置 Pod 的初始化时间，在此时间内的 Pod，CPU 资源度量值将不会被采纳。

在排除掉被搁置的 Pod 后，扩缩比例就会根据 currentMetricValue / desiredMetricValue 计算出来。

如果缺失任何的度量值，我们会更保守地重新计算平均值，在需要缩小时假设这些 Pod 消耗了目标值的 100%，在需要放大时假设这些 Pod 消耗了 0% 目标值。这可以在一定程度上抑制扩缩的幅度。

此外，如果存在任何尚未就绪的 Pod，我们可以在不考虑遗漏指标或尚未就绪的 Pod 的情况下进行扩缩，我们保守地假设尚未就绪的 Pod 消耗了期望指标的 0%，从而进一步降低了扩缩的幅度。

在扩缩方向（缩小或放大）确定后，我们会把未就绪的 Pod 和缺少指标的 Pod 考虑进来再次计算使用率。如果新的比率与扩缩方向相反，或者在容忍范围内，则跳过扩缩。否则，我们使用新的扩缩比例。

注意，平均利用率的原始值会通过 HorizontalPodAutoscaler 的状态体现（即使使用了新的使用率，也不考虑未就绪 Pod 和缺少指标的 Pod）。

如果创建 HorizontalPodAutoscaler 时指定了多个指标，那么会按照每个指标分别计算扩缩副本数，取最大值进行扩缩。如果任何一个指标无法顺利地计算出扩缩副本数（比如，通过 API 获取指标时出错），并且可获取的指标建议缩容，那么本次扩缩会被跳过。这表示，如果一个或多个指标给出的 desiredReplicas 值大于当前值，HPA 仍然能实现扩容。

最后，在 HPA 控制器执行扩缩操作之前，会记录扩缩建议信息。控制器会在操作时间窗口中考虑所有的建议信息，并从中选择得分最高的建议。这个值可通过 kube-controller-manager 服务的启动参数 --horizontal-pod-autoscaler-downscale-stabilization 进行配置，默认值为 5 分钟。这个配置可以让系统更为平滑地进行缩容操作，从而消除短时间内指标值快速波动产生的影响。

API 对象

HorizontalPodAutoscaler 是 Kubernetes autoscaling API 组的资源。在当前稳定版本（autoscaling/v1）中只支持基于 CPU 指标的扩缩。

API 的 beta 版本（autoscaling/v2beta2）引入了基于内存和自定义指标的扩缩。在 autoscaling/v2beta2 版本中新引入的字段在 autoscaling/v1 版本中以注解的形式得以保留。

创建 HorizontalPodAutoscaler 对象时，需要确保所给的名称是一个合法的 [DNS 子域名](#)。有关 API 对象的更多信息，请查阅 [HorizontalPodAutoscaler 对象设计文档](#)。

kubectl 对 Horizontal Pod Autoscaler 的支持

与其他 API 资源类似，kubectl 以标准方式支持 HPA。我们可以通过 kubectl create 命令创建一个 HPA 对象，通过 kubectl get hpa 命令来获取所有 HPA 对象，通过 kubectl describe hpa 命令来查看 HPA 对象的详细信息。最后，可以使用 kubectl delete hpa 命令删除对象。

此外，还有个简便的命令 kubectl autoscale 来创建 HPA 对象。例如，命令 kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80 将会为名为 *foo* 的 ReplicationSet 创建一个 HPA 对象，目标 CPU 使用率为 80%，副本数量配置为 2 到 5 之间。

滚动升级时扩缩

目前在 Kubernetes 中，可以针对 ReplicationController 或 Deployment 执行滚动更新，它们会为你管理底层副本数。Pod 水平扩缩只支持后一种：HPA 会被绑定到 Deployment 对象，HPA 设置副本数量时，Deployment 会设置底层副本数。

通过直接操控副本控制器执行滚动升级时，HPA 不能工作，也就是说你不能将 HPA 绑定到某个 RC 再执行滚动升级。HPA 不能工作的原因是它无法绑定到滚动更新时所新创建的副本控制器。

冷却/延迟支持

当使用 Horizontal Pod Autoscaler 管理一组副本扩缩时，有可能因为指标动态的变化造成副本数量频繁的变化，有时这被称为 *抖动* (*Thrashing*)。

从 v1.6 版本起，集群操作员可以调节某些 kube-controller-manager 的全局参数来缓解这个问题。

从 v1.12 开始，算法调整后，扩容操作时的延迟就不必设置了。

- `--horizontal-pod-autoscaler-downscale-stabilization`: kube-controller-manager 的这个参数表示缩容冷却时间。即自从上次缩容执行结束后，多久可以再次执行缩容，默认时间是 5 分钟(5m0s)。

说明：当调整这些参数时，集群操作员需要明白其可能的影响。如果延迟（冷却）时间设置的太长，Horizontal Pod Autoscaler 可能会不能很好的改变负载。如果延迟（冷却）时间设置的太短，那么副本数量有可能跟以前一样出现抖动。

多指标支持

Kubernetes 1.6 开始支持基于多个度量值进行扩缩。你可以使用 `autoscaling/v2beta2` API 来为 Horizontal Pod Autoscaler 指定多个指标。Horizontal Pod Autoscaler 会根据每个指标计算，并生成一个扩缩建议。幅度最大的扩缩建议会被采纳。

自定义指标支持

说明：在 Kubernetes 1.2 增加了支持基于使用特殊注解表达的、特定于具体应用的扩缩能力，此能力处于 Alpha 阶段。从 Kubernetes 1.6 起，由于新的 autoscaling API 的引入，这些 annotation 就被废弃了。虽然收集自定义指标的旧方法仍然可用，Horizontal Pod Autoscaler 调度器将不会再使用这些度量值。同时，Horizontal Pod Autoscaler 也不再使用之前用于指定用户自定义指标的注解。

自 Kubernetes 1.6 起，Horizontal Pod Autoscaler 支持使用自定义指标。你可以使用 autoscaling/v2beta2 API 为 Horizontal Pod Autoscaler 指定用户自定义指标。Kubernetes 会通过用户自定义指标 API 来获取相应的指标。

关于指标 API 的要求，请参阅[对 Metrics API 的支持](#)。

对 Metrics API 的支持

默认情况下，HorizontalPodAutoscaler 控制器会从一系列的 API 中检索度量值。集群管理员需要确保下述条件，以保证 HPA 控制器能够访问这些 API：

- 启用了[API 聚合层](#)
- 相应的 API 已注册：
 - 对于资源指标，将使用 metrics.k8s.io API，一般由[metrics-server](#) 提供。它可以做为集群插件启动。
 - 对于自定义指标，将使用 custom.metrics.k8s.io API。它由其他度量指标方案厂商的“适配器（Adapter）”API 服务器提供。确认你的指标流水线，或者查看[已知方案列表](#)。如果你想自己编写，请从[boilerplate](#)开始。
 - 对于外部指标，将使用 external.metrics.k8s.io API。可能由上面的自定义指标适配器提供。
- --horizontal-pod-autoscaler-use-rest-clients 参数设置为 true 或者不设置。如果设置为 false，则会切换到基于 Heapster 的自动扩缩，这个特性已经被弃用了。

关于指标来源以及其区别的更多信息，请参阅相关的设计文档，[the HPA V2](#)、[custom.metrics.k8s.io](#) 和 [external.metrics.k8s.io](#)。

关于如何使用它们的示例，请参考[使用自定义指标的教程](#)和[使用外部指标的教程](#)。

支持可配置的扩缩

从[v1.18](#) 开始，v2beta2 API 允许通过 HPA 的 behavior 字段配置扩缩行为。在 behavior 字段中的 scaleUp 和 scaleDown 分别指定扩容和缩容行为。可以两个方向指定一个稳定窗口，以防止扩缩目标中副本数量的波动。类似地，指定扩缩策略可以控制扩缩时副本数的变化率。

扩缩策略

在 spec 字段的 behavior 部分可以指定一个或多个扩缩策略。当指定多个策略时，默认选择允许更改最多的策略。下面的例子展示了缩容时的行为：

```
behavior:  
  scaleDown:  
    policies:  
      - type: Pods  
        value: 4  
        periodSeconds: 60  
      - type: Percent  
        value: 10  
        periodSeconds: 60
```

当 Pod 数量超过 40 个时，第二个策略将用于缩容。例如，如果有 80 个副本，并且目标必须缩小到 10 个副本，那么在第一步中将减少 8 个副本。在下一轮迭代中，当副本的数量为 72 时，10% 的 Pod 数为 7.2，但是这个数字向上取整为 8。在 autoscaler 控制器的每个循环中，将根据当前副本的数量重新计算要更改的 Pod 数量。当副本数量低于 40 时，应用第一个策略（Pods），一次减少 4 个副本。

periodSeconds 表示策略的时间长度必须保证有效。第一个策略允许在一分钟内最多缩小 4 个副本。第二个策略最多允许在一分钟内缩小当前副本的 10%。

可以指定扩缩方向的 selectPolicy 字段来更改策略选择。通过设置 Min 的值，它将选择副本数变化最小的策略。将该值设置为 Disabled 将完全禁用该方向的缩放。

稳定窗口

当用于扩缩的指标持续抖动时，使用稳定窗口来限制副本数上下振动。自动扩缩算法使用稳定窗口来考虑过去计算的期望状态，以防止扩缩。在下面的例子中，稳定化窗口被指定为 scaleDown。

```
scaleDown:  
  stabilizationWindowSeconds: 300
```

当指标显示目标应该缩容时，自动扩缩算法查看之前计算的期望状态，并使用指定时间间隔内的最大值。在上面的例子中，过去 5 分钟的所有期望状态都会被考虑。

默认行为

要使用自定义扩缩，不必指定所有字段。只有需要自定义的字段才需要指定。这些自定义值与默认值合并。默认值与 HPA 算法中的现有行为匹配。

```
behavior:  
  scaleDown:  
    stabilizationWindowSeconds: 300  
    policies:  
      - type: Percent
```

```
value: 100
periodSeconds: 15
scaleUp:
  stabilizationWindowSeconds: 0
  policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
  selectPolicy: Max
```

用于缩小稳定窗口的时间为 300 秒(或是 --horizontal-pod-autoscaler-downscale-stabilization 参数设定值)。只有一种缩容的策略，允许 100% 删除当前运行的副本，这意味着扩缩目标可以缩小到允许的最小副本数。对于扩容，没有稳定窗口。当指标显示目标应该扩容时，目标会立即扩容。这里有两种策略，每 15 秒添加 4 个 Pod 或 100% 当前运行的副本数，直到 HPA 达到稳定状态。

示例：更改缩容稳定窗口

将下面的 behavior 配置添加到 HPA 中，可提供一个 1 分钟的自定义缩容稳定窗口：

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 60
```

示例：限制缩容速率

将下面的 behavior 配置添加到 HPA 中，可限制 Pod 被 HPA 删除速率为每分钟 10%：

```
behavior:
  scaleDown:
    policies:
      - type: Percent
        value: 10
        periodSeconds: 60
```

为了确保每分钟删除的 Pod 数不超过 5 个，可以添加第二个缩容策略，大小固定为 5，并将 selectPolicy 设置为最小值。将 selectPolicy 设置为 Min 意味着 autoscaler 会选择影响 Pod 数量最小的策略：

```
behavior:
  scaleDown:
    policies:
      - type: Percent
        value: 10
```

```
periodSeconds: 60
- type: Pods
  value: 5
  periodSeconds: 60
selectPolicy: Min
```

示例：禁用缩容

selectPolicy 的值 Disabled 会关闭对给定方向的缩容。因此使用以下策略，将会阻止缩容：

```
behavior:
scaleDown:
  selectPolicy: Disabled
```

隐式维护状态禁用

你可以在不必更改 HPA 配置的情况下隐式地为某个目标禁用 HPA。如果此目标的期望副本个数被设置为 0，而 HPA 的最小副本个数大于 0，则 HPA 会停止调整目标（并将其自身的 ScalingActive 状况设置为 false），直到你通过手动调整目标的期望副本个数或 HPA 的最小副本个数来重新激活。

接下来

- 设计文档：[Horizontal Pod Autoscaling](#)
- kubectl autoscale 命令：[kubectl autoscale](#).
- 使用示例：[Horizontal Pod Autoscaler](#).

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 03, 2020 at 12:52 PM PST: [Update content/zh/docs/tasks/run-application/horizontal-pod-autoscale.md \(5f1a44e25\)](#)

Horizontal Pod Autoscaler 演练

Horizontal Pod Autoscaler 可以根据 CPU 利用率自动扩缩 ReplicationController、Deployment、ReplicaSet 或 StatefulSet 中的 Pod 数量（也可以基于其他应用程序提供的度量指标，目前这一功能处于 beta 版本）。

本文将引领你了解如何为 php-apache 服务器配置和使用 Horizontal Pod Autoscaler。与 Horizontal Pod Autoscaler 相关的更多信息请参阅 [Horizontal Pod Autoscaler 用户指南](#)。

准备开始

本文示例需要一个运行中的 Kubernetes 集群以及 kubectl，版本为 1.2 或更高。[Metrics 服务器](#) 需要被部署到集群中，以便通过 [Metrics API](#) 提供度量数据。Horizontal Pod Autoscaler 根据此 API 来获取度量数据。要了解如何部署 metrics-server，请参考 [metrics-server 文档](#)。

如果需要为 Horizontal Pod Autoscaler 指定多种资源度量指标，你的 Kubernetes 集群以及 kubectl 至少需要达到 1.6 版本。此外，如果要使用自定义度量指标，你的 Kubernetes 集群还必须能够与提供这些自定义指标的 API 服务器通信。最后，如果要使用与 Kubernetes 对象无关的度量指标，则 Kubernetes 集群版本至少需要达到 1.10 版本，同样，需要保证集群能够与提供这些外部指标的 API 服务器通信。更多详细信息，请参阅 [Horizontal Pod Autoscaler 用户指南](#)。

运行 php-apache 服务器并暴露服务

为了演示 Horizontal Pod Autoscaler，我们将使用一个基于 php-apache 镜像的定制 Docker 镜像。Dockerfile 内容如下：

```
FROM php:5-apache
COPY index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

该文件定义了一个 index.php 页面来执行一些 CPU 密集型计算：

```
<?php
$x = 0.0001;
for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
}
echo "OK!";
?>
```

首先，我们使用下面的配置启动一个 Deployment 来运行这个镜像并暴露一个服务：

[application/php-apache.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
```

```
matchLabels:  
  run: php-apache  
replicas: 1  
template:  
  metadata:  
    labels:  
      run: php-apache  
  spec:  
    containers:  
      - name: php-apache  
        image: k8s.gcr.io/hpa-example  
    ports:  
      - containerPort: 80  
    resources:  
      limits:  
        cpu: 500m  
      requests:  
        cpu: 200m
```

```
apiVersion: v1  
kind: Service  
metadata:  
  name: php-apache  
  labels:  
    run: php-apache  
spec:  
  ports:  
  - port: 80  
  selector:  
    run: php-apache
```

运行下面的命令：

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

```
deployment.apps/php-apache created  
service/php-apache created
```

创建 Horizontal Pod Autoscaler

现在，php-apache 服务器已经运行，我们将通过 [kubectl autoscale](#) 命令创建 Horizontal Pod Autoscaler。以下命令将创建一个 Horizontal Pod Autoscaler 用于控制我们上一步骤中创建的 Deployment，使 Pod 的副本数量维持在 1 到 10 之间。

大致来说，HPA 将（通过 Deployment）增加或者减少 Pod 副本的数量以保持所有 Pod 的平均 CPU 利用率在 50% 左右（由于每个 Pod 请求 200 毫核的 CPU，这意味着平均 CPU 用量为 100 毫核）。算法的详情请参阅[相关文档](#)。

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10  
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

我们可以通过以下命令查看 Autoscaler 的状态：

```
kubectl get hpa
```

NAME	REFERENCE	TARGET	MINPODS	MAXPODS
REPLICAS	AGE			
php-apache	Deployment/php-apache/scale	0% / 50%	1	10
	18s			

请注意当前的 CPU 利用率是 0%，这是由于我们尚未发送任何请求到服务器（CURRENT 列显示了相应 Deployment 所控制的所有 Pod 的平均 CPU 利用率）。

增加负载

现在，我们将看到 Autoscaler 如何对增加负载作出反应。我们将启动一个容器，并通过一个循环向 php-apache 服务器发送无限的查询请求（请在另一个终端中运行以下命令）：

```
kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /  
bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
```

一分钟时间左右之后，通过以下命令，我们可以看到 CPU 负载升高了：

```
kubectl get hpa
```

NAME	REFERENCE	TARGET	MINPODS	MAXPODS
REPLICAS	AGE			
php-apache	Deployment/php-apache/scale	305% / 50%	1	10
	3m			

这时，由于请求增多，CPU 利用率已经升至请求值的 305%。可以看到，Deployment 的副本数量已经增长到了 7：

```
kubectl get deployment php-apache
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	7/7	7	7	19m

说明：有时最终副本的数量可能需要几分钟才能稳定下来。由于环境的差异，不同环境中最终的副本数量可能与本示例中的数量不同。

停止负载

我们将通过停止负载来结束我们的示例。

在我们创建 busybox 容器的终端中，输入 $<\text{Ctrl}> + \text{C}$ 来终止负载的产生。

然后我们可以再次检查负载状态（等待几分钟时间）：

```
kubectl get hpa
```

NAME	REFERENCE	TARGET	MINPODS	MAXPODS
REPLICAS	AGE			
php-apache	Deployment/php-apache/scale	0% / 50%	1	10
	11m			1

```
kubectl get deployment php-apache
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	1/1	1	1	27m

这时，CPU 利用率已经降到 0，所以 HPA 将自动缩减副本数量至 1。

说明： 自动扩缩完成副本数量的改变可能需要几分钟的时间。

基于多项度量指标和自定义度量指标自动扩缩

利用 autoscaling/v2beta2 API 版本，你可以在自动扩缩 php-apache 这个 Deployment 时使用其他度量指标。

首先，将 HorizontalPodAutoscaler 的 YAML 文件改为 autoscaling/v2beta2 格式：

```
kubectl get hpa.v2beta2.autoscaling -o yaml > /tmp/hpa-v2.yaml
```

在编辑器中打开 /tmp/hpa-v2.yaml：

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
```

```
- type: Resource
  resource:
    name: cpu
    target
      type: Utilization
      averageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
    - type: Resource
      resource:
        name: cpu
        current:
          averageUtilization: 0
          averageValue: 0
```

需要注意的是，targetCPUUtilizationPercentage 字段已经被名为 metrics 的数组所取代。CPU 利用率这个度量指标是一个 *resource metric* (资源度量指标)，因为它表示容器上指定资源的百分比。除 CPU 外，你还可以指定其他资源度量指标。默认情况下，目前唯一支持的其他资源度量指标为内存。只要 metrics.k8s.io API 存在，这些资源度量指标就是可用的，并且他们不会在不同的 Kubernetes 集群中改变名称。

你还可以指定资源度量指标使用绝对数值，而不是百分比，你需要将 target.type 从 Utilization 替换成 AverageValue，同时设置 target.averageValue 而非 target.averageUtilization 的值。

还有两种其他类型的度量指标，他们被认为是 *custom metrics* (自定义度量指标)：即 Pod 度量指标和 Object 度量指标。这些度量指标可能具有特定于集群的名称，并且需要更高级的集群监控设置。

第一种可选的度量指标类型是 Pod 度量指标。这些指标从某一方面描述了 Pod，在不同 Pod 之间进行平均，并通过与一个目标值比对来确定副本的数量。它们的工作方式与资源度量指标非常相像，只是它们仅支持 target 类型为 AverageValue。

pod 度量指标通过如下代码块定义：

```
type: Pods
pods:
  metric:
    name: packets-per-second
    target:
      type: AverageValue
      averageValue: 1k
```

第二种可选的度量指标类型是对象 (Object) 度量指标。这些度量指标用于描述 在相同名字空间中的别的对象，而非 Pods。请注意这些度量指标不一定来自某对象，它们仅用于描述这些对象。对象度量指标支持的 target 类型包括 Value 和 AverageValue。如果是 Value 类型，target 值将直接与 API 返回的度量指标比较，而对于 AverageValue 类型，API 返回的度量值将按照 Pod 数量拆分，然后再与 target 值比较。下面的 YAML 文件展示了一个表示 requests-per-second 的度量指标。

```
type: Object
object:
  metric:
    name: requests-per-second
describedObject:
  apiVersion: networking.k8s.io/v1
  kind: Ingress
  name: main-route
target:
  type: Value
  value: 2k
```

如果你指定了多个上述类型的度量指标，HorizontalPodAutoscaler 将会依次考量各个指标。HorizontalPodAutoscaler 将会计算每一个指标所提议的副本数量，然后最终选择一个最高值。

比如，如果你的监控系统能够提供网络流量数据，你可以通过 kubectl edit 命令 将上述 Horizontal Pod Autoscaler 的定义更改为：

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageUtilization
          averageUtilization: 50
    - type: Pods
      pods:
        metric:
```

```
  name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
- type: Object
  object:
    metric:
      name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1beta1
      kind: Ingress
      name: main-route
    target:
      kind: Value
      value: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
- type: Resource
  resource:
    name: cpu
  current:
    averageUtilization: 0
    averageValue: 0
- type: Object
  object:
    metric:
      name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1beta1
      kind: Ingress
      name: main-route
    current:
      value: 10k
```

这样，你的 HorizontalPodAutoscaler 将会尝试确保每个 Pod 的 CPU 利用率在 50% 以内，每秒能够服务 1000 个数据包请求，并确保所有在 Ingress 后的 Pod 每秒能够服务的请求总数达到 10000 个。

基于更特别的度量值来扩缩

许多度量流水线允许你通过名称或附加的 标签 来描述度量指标。对于所有非资源类型度量指标（Pod、Object 和后面将介绍的 External），可以额外指定一个标签选择算

符。例如，如果你希望收集包含 verb 标签的 http_requests 度量指标，可以按如下所示设置度量指标块，使得扩缩操作仅针对 GET 请求执行：

```
type: Object
object:
  metric:
    name: `http_requests`
    selector: `verb=GET`
```

这个选择算符使用与 Kubernetes 标签选择算符相同的语法。如果名称和标签选择算符匹配到多个系列，监测管道会决定如何将多个系列合并成单个值。选择算符是可以累加的，它不会选择目标以外的对象（类型为 Pods 的目标 Pods 或者类型为 Object 的目标对象）。

基于与 Kubernetes 对象无关的度量指标执行扩缩

运行在 Kubernetes 上的应用程序可能需要基于与 Kubernetes 集群中的任何对象没有明显关系的度量指标进行自动扩缩，例如那些描述与任何 Kubernetes 名字空间中的服务都无直接关联的度量指标。在 Kubernetes 1.10 及之后版本中，你可以使用外部度量指标（external metrics）。

使用外部度量指标时，需要了解你所使用的监控系统，相关的设置与使用自定义指标时类似。外部度量指标使得你可以使用你的监控系统的任何指标来自动扩缩你的集群。你只需要在 metric 块中提供 name 和 selector，同时将类型由 Object 改为 External。如果 metricSelector 匹配到多个度量指标，HorizontalPodAutoscaler 将会把它们加和。外部度量指标同时支持 Value 和 AverageValue 类型，这与 Object 类型的度量指标相同。

例如，如果你的应用程序处理来自主机上消息队列的任务，为了让每 30 个任务有 1 个工作者实例，你可以将下面的内容添加到 HorizontalPodAutoscaler 的配置中。

```
- type: External
  external:
    metric:
      name: queue_messages_ready
      selector: "queue=worker_tasks"
    target:
      type: AverageValue
      averageValue: 30
```

如果可能，还是推荐定制度量指标而不是外部度量指标，因为这便于让系统管理员加固定制度量指标 API。而外部度量指标 API 可以允许访问所有的度量指标。当暴露这些服务时，系统管理员需要仔细考虑这个问题。

附录：Horizontal Pod Autoscaler 状态条件

使用 autoscaling/v2beta2 格式的 HorizontalPodAutoscaler 时，你将可以看到 Kubernetes 为 HorizontalPodAutoscaler 设置的状态条件（Status

Conditions)。这些状态条件可以显示当前 HorizontalPodAutoscaler 是否能够执行扩缩以及是否受到一定的限制。

status.conditions 字段展示了这些状态条件。可以通过 kubectl describe hpa 命令查看当前影响 HorizontalPodAutoscaler 的各种状态条件信息：

```
kubectl describe hpa cm-test
```

```
Name: cm-test
Namespace: prom
Labels: <none>
Annotations: <none>
CreationTimestamp: Fri, 16 Jun 2017 18:09:22 +0000
Reference: ReplicationController/cm-test
Metrics: ( current / target )
  "http_requests" on pods: 66m / 500m
Min replicas: 1
Max replicas: 4
ReplicationController pods: 1 current / 1 desired
Conditions:
  Type Status Reason Message
  ---- ---- -
  AbleToScale True ReadyForNewScale the last scale time was
sufficiently old as to warrant a new scale
  ScalingActive True ValidMetricFound the HPA was able to
successfully calculate a replica count from pods metric http_requests
  ScalingLimited False DesiredWithinRange the desired replica count is
within the acceptable range
Events:
```

对于上面展示的这个 HorizontalPodAutoscaler，我们可以看出有若干状态条件处于健康状态。首先，AbleToScale 表明 HPA 是否可以获取和更新扩缩信息，以及是否存在阻止扩缩的各种回退条件。其次，ScalingActive 表明 HPA 是否被启用（即目标的副本数量不为零）以及是否能够完成扩缩计算。当这一状态为 False 时，通常表明获取度量指标存在问题。最后一个条件 ScalingLimited 表明所需扩缩的值被 HorizontalPodAutoscaler 所定义的最大或者最小值所限制（即已经达到最大或者最小扩缩值）。这通常表明你可能需要调整 HorizontalPodAutoscaler 所定义的最大或者最小副本数量的限制了。

附录：量纲

HorizontalPodAutoscaler 和 度量指标 API 中的所有的度量指标使用 Kubernetes 中称为 [量纲 \(Quantity\)](#) 的特殊整数表示。例如，数量 10500m 用十进制表示为 10.5。如果可能的话，度量指标 API 将返回没有后缀的整数，否则返回以千分单位的数量。这意味着你可能会看到你的度量指标在 1 和 1500m（也就是在十进制记数法中的 1 和 1.5）之间波动。

附录：其他可能的情况

以声明式方式创建 Autoscaler

除了使用 `kubectl autoscale` 命令，也可以文件创建 `HorizontalPodAutoscaler`：

[application/hpa/php-apache.yaml](#)



```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

使用如下命令创建 autoscaler：

```
kubectl create -f https://k8s.io/examples/application/hpa/php-apache.yaml
```

```
horizontalpodautoscaler.autoscaling/php-apache created
```

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

为应用程序设置干扰预算 (Disruption Budget)

FEATURE STATE: Kubernetes v1.5 [beta]

本文展示如何限制应用程序的并发干扰数量，在允许集群管理员管理集群节点的同时保证高可用。

准备开始

- 你是 Kubernetes 集群中某应用的所有者，该应用有高可用要求。
- 你应了解如何部署[无状态应用](#) 和/或[有状态应用](#)。
- 你应当已经阅读过关于[Pod 干扰](#) 的文档。
- 用户应当与集群所有者或服务提供者确认其遵从 Pod 干扰预算 (Pod Disruption Budgets) 的规则。

用 PodDisruptionBudget 来保护应用

1. 确定想要使用 PodDisruptionBudget (PDB) 来保护的应用。
2. 考虑应用对干扰的反应。
3. 以 YAML 文件形式定义 PDB 。
4. 通过 YAML 文件创建 PDB 对象。

确定要保护的应用

用户想要保护通过内置的 Kubernetes 控制器指定的应用，这是最常见的使用场景：

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

在这种情况下，在控制器的 `.spec.selector` 字段中做记录，并在 PDB 的 `.spec.selector` 字段中加入同样的选择算符。

从 1.15 版本开始，PDB 支持启用[scale 子资源](#) 的自定义控制器。

用户也可以用 PDB 来保护不受上述控制器控制的 Pod，或任意的 Pod 集合，但是正如[任意控制器和选择算符](#)中描述的，这里存在一些限制。

考虑应用对干扰的反应

确定在自发干扰时，多少实例可以在短时间内同时关闭。

- 无状态的前端：
 - 关注：不能降低服务能力 10% 以上。
 - 解决方案：例如，使用 PDB，指定其 `minAvailable` 值为 90%。
- 单实例有状态应用：
 - 关注：不要在不通知的情况下终止该应用。
 - 可能的解决方案 1：不使用 PDB，并忍受偶尔的停机。
 - 可能的解决方案 2：设置 `maxUnavailable=0` 的 PDB。意为（Kubernetes 范畴之外的）集群操作人员需要在终止应用前与用户协

商，协商后准备停机，然后删除 PDB 表示准备接受干扰，后续再重新创建。

- 多实例有状态应用，如 Consul、ZooKeeper 或 etcd：
 - 关注：不要将实例数量减少至低于仲裁规模，否则将出现写入失败。
 - 可能的解决方案 1：设置 maxUnavailable 值为 1（适用于不同规模的应用）。
 - 可能的解决方案 2：设置 minAvailable 值为仲裁规模（例如规模为 5 时设置为 3）。（允许同时出现更多的干扰）。
- 可重新启动的批处理任务：
 - 关注：自发干扰的情况下，需要确保任务完成。
 - 可能的解决方案：不创建 PDB。任务控制器会创建一个替换 Pod。

指定百分比时的舍入逻辑

minAvailable 或 maxUnavailable 的值可以表示为整数或百分比。

- 指定整数值时，它表示 Pod 个数。例如，如果将 minAvailable 设置为 10，那么即使在干扰期间，也必须始终有 10 个 Pod 可用。
- 通过将值设置为百分比的字符串表示形式（例如 "50%"）来指定百分比时，它表示占总 Pod 数的百分比。例如，如果将 "minUnavailable" 设置为 "50%"，则干扰期间只允许 50% 的 Pod 不可用。

如果将值指定为百分比，则可能无法映射到确切数量的 Pod。例如，如果你有 7 个 Pod，并且你将 minAvailable 设置为 "50%"，具体是 3 个 Pod 或 4 个 Pod 必须可用并非显而易见。Kubernetes 采用向上取整到最接近的整数的办法，因此在这种情况下，必须有 4 个 Pod。你可以检查控制此行为的 [代码](#)。

指定 PodDisruptionBudget

一个 PodDisruptionBudget 有 3 个字段：

- 标签选择符 .spec.selector 用于指定其所作用的 Pod 集合，该字段为必需字段。
- .spec.minAvailable 表示驱逐后仍须保证可用的 Pod 数量。即使因此影响到 Pod 驱逐（即该条件在和 Pod 驱逐发生冲突时优先保证）。minAvailable 值可以是绝对值，也可以是百分比。
- .spec.maxUnavailable（Kubernetes 1.7 及更高的版本中可用）表示驱逐后允许不可用的 Pod 的最大数量。其值可以是绝对值或是百分比。

说明：对于 1.8 及更早的版本：当你用 kubectl 命令行工具创建 PodDisruptionBudget 对象时，如果既未指定 minAvailable 也未指定 maxUnavailable，则 minAvailable 字段有一个默认值 1。

用户在同一个 PodDisruptionBudget 中只能指定 maxUnavailable 和 minAvailable 中的一个。maxUnavailable 只能够用于控制存在相应控制器的 Pod 的驱逐（即不受控制器控制的 Pod 不在 maxUnavailable 控制范围内）。在下面的示例中，“所需副本”指的是相应控制器的 scale，控制器对 PodDisruptionBudget 所选择的 Pod 进行管理。

示例 1：设置 minAvailable 值为 5 的情况下，驱逐时需保证 PodDisruptionBudget 的 selector 选中的 Pod 中 5 个或 5 个以上处于健康状态。

示例 2：设置 minAvailable 值为 30% 的情况下，驱逐时需保证 Pod 所需副本的至少 30% 处于健康状态。

示例 3：设置 maxUnavailable 值为 5 的情况下，驱逐时需保证所需副本中最多 5 个处于不可用状态。

示例 4：设置 maxUnavailable 值为 30% 的情况下，驱逐时需保证所需副本中最多 30% 处于不可用状态。

在典型用法中，干扰预算会被用于一个控制器管理的一组 Pod 中 —— 例如：一个 ReplicaSet 或 StatefulSet 中的 Pod。

说明： 干扰预算并不能真正保证指定数量/百分比的 Pod 一直处于运行状态。

例如：当 Pod 集合的 规模处于预算指定的最小值时，承载集合中某个 Pod 的节点发生了故障，这样就导致集合中可用 Pod 的 数量低于预算指定值。预算只能能够针对自发的驱逐提供保护，而不能针对所有 Pod 不可用的诱因。

设置 maxUnavailable 值为 0% (或 0) 或设置 minAvailable 值为 100% (或等于副本数) 可能会阻塞节点，导致资源耗尽。按照 PodDisruptionBudget 的语义，这是允许的。

用户可以在下面看到 pod 干扰预算定义的示例，它们与带有 app: zookeeper 标签的 pod 相匹配：

使用 minAvailable 的PDB 示例：

[policy/zookeeper-pod-disruption-budget-minavailable.yaml](#)
□

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

使用 maxUnavailable 的 PDB 示例 (Kubernetes 1.7 或更高的版本)：

[policy/zookeeper-pod-disruption-budget-maxunavailable.yaml](#)
□

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
```

```
maxUnavailable: 1
selector:
  matchLabels:
    app: zookeeper
```

例如，如果上述 zk-pdb 选择的是一个规格为 3 的 StatefulSet 对应的 Pod，那么上面两种规范的含义完全相同。推荐使用 maxUnavailable，因为它自动响应控制器副本数量的变化。

创建 PDB 对象

你可以通过类似 kubectl apply -f mypdb.yaml 的命令来创建 PDB。

PDB 对象无法更新，必须删除后重新创建。

检查 PDB 的状态

使用 kubectl 来确认 PDB 被创建。

假设用户的名字空间下没有匹配 app: zookeeper 的 Pod，用户会看到类似下面的信息：

```
kubectl get poddisruptionbudgets
```

NAME	MIN AVAILABLE	MAX UNAVAILABLE	ALLOWED DISRUPTIONS	AGE
zk-pdb	2	N/A	0	7s

假设有匹配的 Pod (比如说 3 个)，那么用户会看到类似下面的信息：

```
kubectl get poddisruptionbudgets
```

NAME	MIN AVAILABLE	MAX UNAVAILABLE	ALLOWED DISRUPTIONS	AGE
zk-pdb	2	N/A	1	7s

ALLOWED-DISRUPTIONS 值非 0 意味着干扰控制器已经感知到相应的 Pod，对匹配的 Pod 进行统计，并更新了 PDB 的状态。

用户可以通过以下命令获取更多 PDB 状态相关信息：

```
kubectl get poddisruptionbudgets zk-pdb -o yaml
```

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  annotations: {}
  creationTimestamp: "2020-03-04T04:22:56Z"
  generation: 1
  name: zk-pdb
```

```
...  
status:  
  currentHealthy: 3  
  desiredHealthy: 2  
  disruptionsAllowed: 1  
  expectedPods: 3  
  observedGeneration: 1
```

任意控制器和选择算符

如果你只使用与内置的应用控制器（ Deployment、ReplicationController、ReplicaSet 和 StatefulSet ）对应的 PDB，也就是 PDB 的选择算符与 控制器的选择算符相匹配，那么可以跳过这一节。

你可以使用这样的 PDB：它对应的 Pod 可能由其他类型的控制器控制，可能由 "operator" 控制，也可能为"裸的（不受控制器控制）" Pod，但该类 PDB 存在以下限制：

- 只能够使用 .spec.minAvailable，而不能够使用 .spec.maxUnavailable。
- 只能够使用整数作为 .spec.minAvailable 的值，而不能使用百分比。

你可以令选择算符选择一个内置控制器所控制 Pod 的子集或父集。然而，当名字空间下存在多个 PDB 时，用户必须小心，保证 PDB 的选择算符之间不重叠。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 17, 2020 at 11:51 PM PST: [Update configure-pdb.md \(948b1e91a\)](#)

使用Deployment运行一个无状态应用

本文介绍通过Kubernetes Deployment对象如何去运行一个应用.

教程目标

- 创建一个 nginx Deployment.
- 使用 kubectl 列举关于 Deployment 的信息.
- 更新 Deployment.

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.

创建并了解一个 nginx Deployment

你可以通过创建一个 Kubernetes Deployment 对象来运行一个应用，且你可以在一个 YAML 文件中描述 Deployment。例如，下面这个 YAML 文件描述了一个运行 nginx: 1.14.2 Docker 镜像的 Deployment：

[application/deployment.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
        ports:
          - containerPort: 80
```

1. 通过 YAML 文件创建一个 Deployment：

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

1. 显示 Deployment 相关信息：

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

输出类似于这样：

```
Name: nginx-deployment
Namespace: default
CreationTimestamp: Tue, 30 Aug 2016 18:11:37 -0700
Labels: app=nginx
Annotations: deployment.kubernetes.io/revision=1
Selector: app=nginx
Replicas: 2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Pod Template:
  Labels: app=nginx
  Containers:
    nginx:
      Image: nginx:1.7.9
      Port: 80/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Conditions:
    Type Status Reason
    ----
    Available True MinimumReplicasAvailable
    Progressing True NewReplicaSetAvailable
    OldReplicaSets: <none>
    NewReplicaSet: nginx-deployment-1771418926 (2/2 replicas created)
    No events.
```

1. 列出 Deployment 创建的 Pods :

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

输出类似于这样：

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1771418926-7o5ns	1/1	Running	0	16h
nginx-deployment-1771418926-r18az	1/1	Running	0	16h

1. 展示某一个 Pod 信息：

```
kubectl describe pod <pod-name>
```

这里的 <pod-name> 是某一 Pod 的名称。

更新 Deployment

你可以通过更新一个新的 YAML 文件来更新 Deployment。下面的 YAML 文件指定该 Deployment 镜像更新为 nginx 1.16.1。

[application/deployment-update.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1 # Update the version of nginx from 1.14.2 to 1.16.1
          ports:
            - containerPort: 80
```

1. 应用新的 YAML：

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

1. 查看该 Deployment 以新的名称创建 Pods 同时删除旧的 Pods：

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

通过增加副本数来扩缩应用

你可以通过应用新的 YAML 文件来增加 Deployment 中 Pods 的数量。下面的 YAML 文件将 replicas 设置为 4，指定该 Deployment 应有 4 个 Pods：

[application/deployment-scale.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:  
  name: nginx-deployment  
spec:  
  selector:  
    matchLabels:  
      app: nginx  
  replicas: 4 # Update the replicas from 2 to 4  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
      - name: nginx  
        image: nginx:1.14.2  
      ports:  
      - containerPort: 80
```

1. 应用新的 YAML 文件：

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

1. 验证 Deployment 有 4 个 Pods：

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

输出的结果类似于：

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-148880595-4zdqq	1/1	Running	0	25s
nginx-deployment-148880595-6zgi1	1/1	Running	0	25s
nginx-deployment-148880595-fxcez	1/1	Running	0	2m
nginx-deployment-148880595-rwovn	1/1	Running	0	2m

删除 Deployment

通过名称删除 Deployment：

```
kubectl delete deployment nginx-deployment
```

ReplicationControllers -- 旧的方式

创建一个多副本应用首选方法是使用 Deployment，Deployment 内部使用 ReplicaSet。在 Deployment 和 ReplicaSet 被引入到 Kubernetes 之前，多副本应用通过 [ReplicationController](#) 来配置。

接下来

- 进一步了解 [Deployment 对象](#)。

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 14, 2020 at 11:20 PM PST: [Update run-stateless-application-deployment.md \(e21ba0e7f\)](#)

扩缩 StatefulSet

本文介绍如何扩缩StatefulSet。StatefulSet 的扩缩指的是增加或者减少副本个数。

准备开始

- StatefulSets 仅适用于 Kubernetes 1.5 及以上版本。
- 不是所有 Stateful 应用都能很好地执行扩缩操作。如果你不是很确定是否要扩缩你的 StatefulSet，可先参阅 [StatefulSet 概念](#) 或者 [StatefulSet 教程](#)。
- 仅当你确定你的有状态应用的集群是完全健康的，才可执行扩缩操作.

扩缩 StatefulSet

使用 kubectl 扩缩 StatefulSet

首先，找到你要扩缩的 StatefulSet。

```
kubectl get statefulsets <statefulset 名称>
```

更改 StatefulSet 中副本个数：

```
kubectl scale statefulsets <statefulset 名称> --replicas=<新的副本数>
```

对 StatefulSet 执行就地更新

另外，你可以[就地更新](#) StatefulSet。

如果你的 StatefulSet 最初通过 kubectl apply 或 kubectl create --save-config 创建，你可以更新 StatefulSet 清单中的 .spec.replicas，然后执行命令 kubectl apply:

```
kubectl apply -f <更新后的 statefulset 文件>
```

否则，可以使用 kubectl edit 编辑副本字段：

```
kubectl edit statefulsets <statefulset 名称>
```

或者使用 kubectl patch：

```
kubectl patch statefulsets <statefulset 名称> -p '{"spec":{"replicas":<new-replicas>}}'
```

故障排查

缩容操作无法正常工作

当 StatefulSet 所管理的任何 Pod 不健康时，你不能对该 StatefulSet 执行缩容操作。仅当 StatefulSet 的所有 Pod 都处于运行状态和 Ready 状况后才可缩容。

如果 spec.replicas 大于 1，Kubernetes 无法判定 Pod 不健康的原因。Pod 不健康可能是由于永久性故障造成也可能是瞬态故障。瞬态故障可能是节点升级或维护而引起的节点重启造成的。

如果该 Pod 不健康是由于永久性故障导致，则在不纠正该故障的情况下进行缩容可能会导致 StatefulSet 进入一种状态，其成员 Pod 数量低于应正常运行的副本数。这种状态也许会导致 StatefulSet 不可用。

如果由于瞬态故障而导致 Pod 不健康并且 Pod 可能再次变为可用，那么瞬态错误可能会干扰你对 StatefulSet 的扩容/缩容操作。一些分布式数据库在同时有节点加入和离开时会遇到问题。在这些情况下，最好是在应用级别进行分析扩缩操作的状态，并且只有在确保 Stateful 应用的集群是完全健康时才执行扩缩操作。

接下来

- 进一步了解[删除 StatefulSet](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 10, 2020 at 11:40 AM PST: [\[zh\] Tidy up and fix links in tasks section \(2/10\) \(128182188\)](#)

运行 Jobs

使用并行处理运行 Jobs。

[使用 CronJob 运行自动化任务](#)

[使用展开的方式进行并行处理](#)

[使用工作队列进行粗粒度并行处理](#)

[使用工作队列进行精细的并行处理](#)

使用 CronJob 运行自动化任务

你可以利用 [CronJobs](#) 执行基于时间调度的任务。这些自动化任务和 Linux 或者 Unix 系统的 [Cron](#) 任务类似。

CronJobs 在创建周期性以及重复性的任务时很有帮助，例如执行备份操作或者发送邮件。CronJobs 也可以在特定时间调度单个任务，例如你想调度低活跃周期的任务。

CronJobs 有一些限制和特点。例如，在特定状况下，同一个 CronJob 可以创建多个任务。因此，任务应该是幂等的。查看更多限制，请参考 [CronJobs](#)。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.8. 要获知版本信息，请输入 kubectl version。

创建 CronJob

CronJob 需要一个配置文件。本例中 CronJob 的.spec 配置文件每分钟打印出当前时间和一个问好信息：



[application/job/cronjob.yaml](#)

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
```

```
name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              imagePullPolicy: IfNotPresent
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

想要运行示例的 CronJob，可以下载示例文件并执行命令：

```
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
```

```
cronjob.batch/hello created
```

创建好 CronJob 后，使用下面的命令来获取其状态：

```
kubectl get cronjob hello
```

输出类似于：

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	50s	75s

就像你从命令返回结果看到的那样，CronJob 还没有调度或执行任何任务。大约需要一分钟任务才能创建好。

```
kubectl get jobs --watch
```

NAME	COMPLETIONS	DURATION	AGE
hello-4111706356	0/1	0s	
hello-4111706356	0/1	0s	0s
hello-4111706356	1/1	5s	5s

现在你已经看到了一个运行中的任务被 "hello" CronJob 调度。你可以停止监视这个任务，然后再次查看 CronJob 就能看到它调度任务：

```
kubectl get cronjob hello
```

输出类似于：

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	50s	75s

你应该能看到 "hello" CronJob 在 LAST-SCHEDULE 声明的时间点成功的调度了一次任务。有 0 个活跃的任务意味着任务执行完毕或者执行失败。

现在，找到最后一次调度任务创建的 Pod 并查看一个 Pod 的标准输出。请注意任务名称和 Pod 名称是不同的。

说明： Job 名称和 Pod 名称不同。

```
# 在你的系统上将 "hello-4111706356" 替换为 Job 名称
$pods=$(kubectl get pods --selector=job-name=hello-4111706356 --output=json
path=.items..metadata.name)
```

查看 Pod 日志：

```
kubectl logs $pods
```

```
Fri Feb 22 11:02:09 UTC 2019
Hello from the Kubernetes cluster
```

删除 CronJob

当你不再需要 CronJob 时，可以用 `kubectl delete cronjob <cronjob name>` 删掉它：

```
kubectl delete cronjob hello
```

删除 CronJob 会清除它创建的所有任务和 Pod，并阻止它创建额外的任务。你可以查阅[垃圾收集](#)。

编写 CronJob 声明信息

像 Kubernetes 的其他配置一样，CronJob 需要 `apiVersion`、`kind`、和 `metadata` 域。配置文件的一般信息，请参考[部署应用](#) 和[使用 kubectl 管理资源](#).

CronJob 配置也需要包括[.spec](#).

说明： 对 CronJob 的所有改动，特别是它的 `.spec`，只会影响将来的运行实例。

时间安排

`.spec.schedule` 是 `.spec` 需要的域。它使用了[Cron](#) 格式串，例如 `0 * * * *` or `@hourly`，做为它的任务被创建和执行的调度时间。

该格式也包含了扩展的 vixie cron 步长值。[FreeBSD 手册](#)中解释如下：

步长可被用于范围组合。范围后面带有 /<数字> 可以声明范围内的步幅数值。例如，0-23/2 可被用在小时域来声明命令在其他数值的小时数执行（V7 标准中对应的方法是0,2,4,6,8,10,12,14,16,18,20,22）。步长也可以放在通配符后面，因此如果你想表达 "每两小时"，就用 */2。

说明：调度中的问号 (?) 和星号 * 含义相同，表示给定域的任何可用值。

任务模版

.spec.jobTemplate 是任务的模版，它是必须的。它和 [Job](#) 的语法完全一样，除了它是嵌套的没有 apiVersion 和 kind。编写任务的 .spec，请参考 [编写 Job 的Spec](#)。

开始的最后期限

.spec.startingDeadlineSeconds 域是可选的。它表示任务如果由于某种原因错过了调度时间，开始该任务的截止时间的秒数。过了截止时间，CronJob 就不会开始任务。不满足这种最后期限的任务会被统计为失败任务。如果该域没有声明，那任务就没有最后期限。

CronJob 控制器会统计错过了多少次调度。如果错过了100次以上的调度，CronJob 就不再调度了。当没有设置 .spec.startingDeadlineSeconds 时，CronJob 控制器统计从 status.lastScheduleTime 到当前的调度错过次数。例如一个 CronJob 期望每分钟执行一次，status.lastScheduleTime 是 5:00am，但现在是 7:00am。那意味着 120 次调度被错过了，所以 CronJob 将不再被调度。如果设置了 .spec.startingDeadlineSeconds 域(非空)，CronJob 控制器统计从 .spec.startingDeadlineSeconds 到当前时间错过了多少次任务。例如设置了 200，它会统计过去 200 秒内错过了多少次调度。在那种情况下，如果过去 200 秒内错过了超过 100 次的调度，CronJob 就不再调度。

并发性规则

.spec.concurrencyPolicy 也是可选的。它声明了 CronJob 创建的任务执行时发生重叠如何处理。spec 仅能声明下列规则中的一种：

- Allow (默认)：CronJob 允许并发任务执行。
- Forbid：CronJob 不允许并发任务执行；如果新任务的执行时间到了而老任务没有执行完，CronJob 会忽略新任务的执行。
- Replace：如果新任务的执行时间到了而老任务没有执行完，CronJob 会用新任务替换当前正在运行的任务。

请注意，并发性规则仅适用于相同 CronJob 创建的任务。如果有多个 CronJob，它们相应的任务总是允许并发执行的。

挂起

.spec.suspend 域也是可选的。如果设置为 true，后续发生的执行都会挂起。这个设置对已经开始的执行不起作用。默认是关闭的。

注意：在调度时间内挂起的执行都会被统计为错过的任务。当 `.spec.suspended` 从 `true` 改为 `false` 时，且没有 [开始的最后期限](#)，错过的任务会被立即调度。

任务历史限制

`.spec.successfulJobsHistoryLimit` 和 `.spec.failedJobsHistoryLimit` 是可选的。这两个域声明了有多少执行完成和失败的任务会被保留。默认设置为3和1。限制设置为0代表相应类型的任务完成后不会保留。

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 15, 2020 at 5:58 PM PST: [Update automated-tasks-with-cron-jobs.md \(8f1ddcca2\)](#)

使用展开的方式进行并行处理

本任务展示基于一个公共的模板运行多个[Jobs](#)。你可以用这种方法来并行执行批处理任务。

在本任务示例中，只有三个工作条目：`apple`、`banana` 和 `cherry`。示例任务处理每个条目时仅仅是打印一个字符串之后结束。参考[在真实负载中使用 Job](#)了解更适用于真实使用场景的模式。

准备开始

你应先熟悉基本的、非并行的 [Job](#) 的用法。

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](#)

任务中的基本模板示例要求安装命令行工具 `sed`。要使用较高级的模板示例，你需要安装 [Python](#)，并且要安装 `Jinja2` 模板库。

一旦 Python 已经安装好，你可以运行下面的命令安装 `Jinja2`：

```
pip install --user jinja2
```

基于模板创建 Job

首先，将以下作业模板下载到名为 job-tmpl.yaml 的文件中。

[application/job/job-tmpl.yaml](#)



```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
          image: busybox
          command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never
```

使用 curl 下载 job-tmpl.yaml

```
curl -L -s -O https://k8s.io/examples/application/job/job-tmpl.yaml
```

你所下载的文件不是一个合法的 Kubernetes 清单。这里的模板只是 Job 对象的 yaml 表示，其中包含一些占位符，在使用它之前需要被填充。\$ITEM 语法对 Kubernetes 没有意义。

基于模板创建清单

下面的 Shell 代码片段使用 sed 将字符串 \$ITEM 替换为循环变量，并将结果写入到一个名为 jobs 的临时目录。

```
# 展开模板文件到多个文件中，每个文件对应一个要处理的条目
mkdir ./jobs
for i in apple banana cherry
do
  cat job-tmpl.yaml | sed "s/\$ITEM/\$i/" > ./jobs/job-$i.yaml
done
```

检查上述脚本的输出：

```
ls jobs/
```

输出类似于：

```
job-apple.yaml  
job-banana.yaml  
job-cherry.yaml
```

你可以使用任何一种模板语言（例如：Jinja2、ERB），或者编写一个程序来生成 Job 清单。

基于清单创建 Job

接下来用一个 kubectl 命令创建所有的 Job：

```
kubectl create -f ./jobs
```

输出类似于：

```
job.batch/process-item-apple created  
job.batch/process-item-banana created  
job.batch/process-item-cherry created
```

现在检查 Job：

```
kubectl get jobs -l jobgroup=jobexample
```

输出类似于：

NAME	COMPLETIONS	DURATION	AGE
process-item-apple	1/1	14s	22s
process-item-banana	1/1	12s	21s
process-item-cherry	1/1	12s	20s

使用 kubectl 的 -l 选项可以仅选择属于当前 Job 组的对象（系统中可能存在其他不相关的 Job）。

你可以使用相同的 [标签选择算符](#) 来过滤 Pods：

```
kubectl get pods -l jobgroup=jobexample
```

输出类似于：

NAME	READY	STATUS	RESTARTS	AGE
process-item-apple-kixwv	0/1	Completed	0	4m
process-item-banana-wrsf7	0/1	Completed	0	4m
process-item-cherry-dnfu9	0/1	Completed	0	4m

我们可以用下面的命令查看所有 Job 的输出：

```
kubectl logs -f -l jobgroup=jobexample
```

输出类似于：

```
Processing item apple
Processing item banana
Processing item cherry
```

清理

```
# 删除所创建的 Job
# 集群会自动清理 Job 对应的 Pod
kubectl delete job -l jobgroup=jobexample
```

使用高级模板参数

在[第一个例子](#)中，模板的每个示例都有一个参数 而该参数也用在 Job 名称中。不过，对象 [名称](#) 被限制只能使用某些字符。

这里的略微复杂的例子使用 [Jinja 模板语言](#) 来生成清单，并基于清单来生成对象，每个 Job 都有多个参数。

在本任务中，你将会使用一个一行的 Python 脚本，将模板转换为一组清单文件。

首先，复制下面的 Job 对象模板到一个名为 job.yaml.jinja2 的文件。

```
{%- set params = [{"name": "apple", "url": "http://dbpedia.org/resource/Apple", },
                  {"name": "banana", "url": "http://dbpedia.org/resource/Banana", },
                  {"name": "cherry", "url": "http://dbpedia.org/resource/Cherry" }]
%}
{%- for p in params %}
{%- set name = p["name"] %}
{%- set url = p["url"] %}

---
apiVersion: batch/v1
kind: Job
metadata:
  name: jobexample-{{ name }}
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
```

```
image: busybox
command: ["sh", "-c", "echo Processing URL {{ url }} && sleep 5"]
restartPolicy: Never
{%- endfor %}
```

上面的模板使用 python 字典列表（第 1-4 行）定义每个作业对象的参数。然后使用 for 循环为每组参数（剩余行）生成一个作业 yaml 对象。我们利用了多个 YAML 文档（这里的 Kubernetes 清单）可以用 --- 分隔符连接的事实。我们可以将输出直接传递给 kubectl 来创建对象。

接下来我们用单行的 Python 程序将模板展开。

```
alias render_template='python -c "from jinja2 import Template; import sys;
print(Template(sys.stdin.read()).render());"'
```

使用 render_template 将参数和模板转换成一个 YAML 文件，其中包含 Kubernetes 资源清单：

```
# 此命令需要之前定义的别名
cat job.yaml.jinja2 | render_template > jobs.yaml
```

你可以查看 jobs.yaml 以验证 render_template 脚本是否正常工作。

当你对输出结果比较满意时，可以用管道将其输出发送给 kubectl，如下所示：

```
cat job.yaml.jinja2 | render_template | kubectl apply -f -
```

Kubernetes 接收清单文件并执行你所创建的 Job。

清理

```
# 删除所创建的 Job
# 集群会自动清理 Job 对应的 Pod
kubectl delete job -l jobgroup=jobexample
```

在真实负载中使用 Job

在真实的负载中，每个 Job 都会执行一些重要的计算，例如渲染电影的一帧，或者处理数据库中的若干行。这时，\$ITEM 参数将指定帧号或行范围。

在此任务中，你运行一个命令通过取回 Pod 的日志来收集其输出。在真实应用场景中，Job 的每个 Pod 都会在结束之前将其输出写入到某持久性存储中。你可以为每个 Job 指定 PersistentVolume 卷，或者使用其他外部存储服务。例如，如果你在渲染视频帧，你可能会使用 HTTP 协议将渲染完的帧数据用 'PUT' 请求发送到某 URL，每个帧使用不同的 URL。

Job 和 Pod 上的标签

你创建了 Job 之后，Kubernetes 自动为 Job 的 Pod 添加 [标签](#)，以便能够将一个 Job 的 Pod 与另一个 Job 的 Pod 区分开来。

在本例中，每个 Job 及其 Pod 模板有一个标签: jobgroup=jobexample。

Kubernetes 自身对标签名 jobgroup 没有什么要求。为创建自同一模板的所有 Job 使用同一标签使得我们可以方便地同时操作组中的所有作业。在[第一个例子](#)中，你使用模板来创建了若干 Job。模板确保每个 Pod 都能够获得相同的标签，这样你可以用一条命令检查这些模板化 Job 所生成的全部 Pod。

说明： 标签键 jobgroup 没什么特殊的，也不是保留字。你可以选择你自己的标签方案。如果愿意，有一些[建议的标签](#)可供使用。

替代方案

如果你有计划创建大量 Job 对象，你可能会发现：

- 即使使用标签，管理这么多 Job 对象也很麻烦。
- 如果你一次性创建很多 Job，很可能会给 Kubernetes 控制面带来很大压力。一种替代方案是，Kubernetes API 可能对请求施加速率限制，通过 429 返回状态值临时拒绝你的请求。
- 你可能会受到 Job 相关的[资源配置](#)限制：如果你在一个批量请求中触发了太多的任务，API 服务器会永久性地拒绝你的某些请求。

还有一些其他[作业模式](#)可供选择，这些模式都能用来处理大量任务而又不会创建过多的 Job 对象。

你也可以考虑编写自己的[控制器](#)来自动管理 Job 对象。

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 15, 2020 at 5:43 PM PST: [Update parallel-processing-expansion.md \(aa7ac408d\)](#)

使用工作队列进行粗粒度并行处理

本例中，我们会运行包含多个并行工作进程的 Kubernetes Job。

本例中，每个 Pod 一旦被创建，会立即从任务队列中取走一个工作单元并完成它，然后将工作单元从队列中删除后再退出。

下面是本次示例的主要步骤：

1. **启动一个消息队列服务** 本例中，我们使用 RabbitMQ，你也可以用其他的消息队列服务。在实际工作环境中，你可以创建一次消息队列服务然后在多个任务中重复使用。
2. **创建一个队列，放上消息数据** 每个消息表示一个要执行的任务。本例中，每个消息是一个整数值。我们将基于这个整数值执行很长的计算操作。
3. **启动一个在队列中执行这些任务的 Job。** 该 Job 启动多个 Pod。每个 Pod 从消息队列中取走一个任务，处理它，然后重复执行，直到队列的队尾。

准备开始

要熟悉 Job 基本用法（非并行的），请参考 [Job](#)。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.8. 要获知版本信息，请输入 kubectl version.

启动消息队列服务

本例使用了 RabbitMQ，使用其他 AMQP 类型的消息服务应该比较容易。

在实际工作中，在集群中一次性部署某个消息队列服务，之后在很多 Job 中复用，包括需要长期运行的服务。

按下面的方法启动 RabbitMQ：

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-service.yaml
```

```
service "rabbitmq-service" created
```

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-controller.yaml
```

```
replicationcontroller "rabbitmq-controller" created
```

我们仅用到 [celery-rabbitmq](#) 示例 中描述的部分功能。

测试消息队列服务

现在，我们可以试着访问消息队列。我们将会创建一个临时的可交互的 Pod，在它上面安装一些工具，然后用队列做实验。

首先创建一个临时的可交互的 Pod：

```
# 创建一个临时的可交互的 Pod  
kubectl run -i --tty temp --image ubuntu:14.04
```

```
Waiting for pod default/temp-loe07 to be running, status is Pending, pod ready:  
false
```

```
... [ previous line repeats several times .. hit return when it stops ] ...
```

请注意你的 Pod 名称和命令提示符将会不同。

接下来安装 amqp-tools，这样我们就能用消息队列了。

```
# 安装一些工具  
root@temp-loe07:/# apt-get update  
.... [ lots of output ] ....  
root@temp-loe07:/# apt-get install -y curl ca-certificates amqp-tools python  
dnsutils  
.... [ lots of output ] ....
```

后续，我们将制作一个包含这些包的 Docker 镜像。

接着，我们将要验证我们发现 RabbitMQ 服务：

```
# 请注意 rabbitmq-service 有Kubernetes 提供的 DNS 名称，
```

```
root@temp-loe07:/# nslookup rabbitmq-service  
Server: 10.0.0.10  
Address: 10.0.0.10#53
```

```
Name: rabbitmq-service.default.svc.cluster.local  
Address: 10.0.147.152
```

```
# 你的 IP 地址会不同
```

如果 Kube-DNS 没有正确安装，上一步可能会出错。你也可以在环境变量中找到服务 IP。

```
# env | grep RABBIT | grep HOST  
RABBITMQ_SERVICE_SERVICE_HOST=10.0.147.152
```

```
# 你的 IP 地址会有所不同
```

接着我们将要确认可以创建队列，并能发布消息和消费消息。

```
# 下一行 , rabbitmq-service 是访问 rabbitmq-service 的主机名。 5672是 rabbitmq  
的标准端口。
```

```
root@temp-loe07:/# export BROKER_URL=amqp://guest:guest@rabbitmq-  
service:5672
```

```
# 如果上一步中你不能解析 "rabbitmq-service" , 可以用下面的命令替换 :
```

```
# root@temp-loe07:/# BROKER_URL=amqp://  
guest:guest@$RABBITMQ_SERVICE_SERVICE_HOST:5672
```

```
# 现在创建队列 :
```

```
root@temp-loe07:/# /usr/bin/amqp-declare-queue --url=$BROKER_URL -q foo -  
d foo
```

```
# 向它推送一条消息:
```

```
root@temp-loe07:/# /usr/bin/amqp-publish --url=$BROKER_URL -r foo -p -b  
Hello
```

```
# 然后取回它.
```

```
root@temp-loe07:/# /usr/bin/amqp-consume --url=$BROKER_URL -q foo -c 1  
cat && echo  
Hello  
root@temp-loe07:/#
```

最后一个命令中 , amqp-consume 工具从队列中取走了一个消息 , 并把该消息传递给了随机命令的标准输出。在这种情况下 , cat 只会打印它从标准输入或得的内容 , echo 只会添加回车符以便示例可读。

为队列增加任务

现在让我们给队列增加一些任务。在我们的示例中 , 任务是多个待打印的字符串。

实践中 , 消息的内容可以是 :

- 待处理的文件名
- 程序额外的参数
- 数据库表的关键字范围
- 模拟任务的配置参数
- 待渲染的场景的帧序列号

本例中 , 如果有大量的数据需要被 Job 的所有 Pod 读取 , 典型的做法是把它们放在一个共享文件系统中 , 如NFS , 并以只读的方式挂载到所有 Pod , 或者 Pod 中的程序从类似 HDFS 的集群文件系统中读取。

例如，我们创建队列并使用 amqp 命令行工具向队列中填充消息。实践中，你可以写个程序来利用 amqp 客户端库来填充这些队列。

```
/usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1 -d job1  
for f in apple banana cherry date fig grape lemon melon  
do  
  /usr/bin/amqp-publish --url=$BROKER_URL -r job1 -p -b $f  
done
```

这样，我们给队列中填充了8个消息。

创建镜像

现在我们可以创建一个做为 Job 来运行的镜像。

我们将用 amqp-consume 来从队列中读取消息并实际运行我们的程序。这里给出一个非常简单的示例程序：

[application/job/rabbitmq/worker.py](#)



```
#!/usr/bin/env python  
  
# Just prints standard out and sleeps for 10 seconds.  
import sys  
import time  
print("Processing " + sys.stdin.readlines()[0])  
time.sleep(10)
```

现在，编译镜像。如果你在用源代码树，那么切换到目录 examples/job/work-queue-1 。否则的话，创建一个临时目录，切换到这个目录。下载 [Dockerfile](#)，和 [worker.py](#)。无论哪种情况，都可以用下面的命令编译镜像

```
docker build -t job-wq-1 .
```

对于 [Docker Hub](#)，给你的应用镜像打上标签， 标签为你的用户名，然后用下面的命令推送到 Hub。用你的 Hub 用户名替换 <username>。

```
docker tag job-wq-1 <username>/job-wq-1  
docker push <username>/job-wq-1
```

如果你在用[谷歌容器仓库](#)，用你的项目 ID 作为标签打到你的应用镜像上，然后推送到 GCR。用你的项目 ID 替换 <project>。

```
docker tag job-wq-1 gcr.io/<project>/job-wq-1  
gcloud docker -- push gcr.io/<project>/job-wq-1
```

定义 Job

这里给出一个 Job 定义 yaml 文件。你需要拷贝一份并编辑镜像以匹配你使用的名称，保存为 ./job.yaml。

[application/job/rabbitmq/job.yaml](#)



```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  completions: 8
  parallelism: 2
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
        - name: c
          image: gcr.io/<project>/job-wq-1
          env:
            - name: BROKER_URL
              value: amqp://guest:guest@rabbitmq-service:5672
            - name: QUEUE
              value: job1
  restartPolicy: OnFailure
```

本例中，每个 Pod 使用队列中的一个消息然后退出。这样，Job 的完成计数就代表了完成的工作项的数量。本例中我们设置 .spec.completions: 8，因为我们放了8项内容在队列中。

运行 Job

现在我们运行 Job：

```
kubectl create -f ./job.yaml
```

稍等片刻，然后检查 Job。

```
kubectl describe jobs/job-wq-1
```

```
Name:      job-wq-1
Namespace:  default
Selector:   controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
Labels:    controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
```

Annotations: <none>
Parallelism: 2
Completions: 8
Start Time: Wed, 06 Sep 2017 16:42:02 +0800
Pods Statuses: 0 Running / 8 Succeeded / 0 Failed
Pod Template:
Labels: controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
job-name=job-wq-1
Containers:
C:
Image: gcr.io/causal-jigsaw-637/job-wq-1
Port:
Environment:
BROKER_URL: amqp://guest:guest@rabbitmq-service:5672
QUEUE: job1
Mounts: <none>
Volumes: <none>
Events:

FirstSeen	LastSeen	Count	From	SubobjectPath	Type	Reason
27s	27s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-hcobb						
27s	27s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-weytj						
27s	27s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-qaam5						
27s	27s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-b67sr						
26s	26s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-xe5hj						
15s	15s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-w2zqe						
14s	14s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-d6ppa						
14s	14s	1	{job }		Normal	SuccessfulCreate
pod: job-wq-1-p17e0						

我们所有的 Pod 都成功了。耶！

替代方案

本文所讲述的处理方法的好处是你不需要修改你的 "worker" 程序使其知道工作队列的存在。

本文所描述的方法需要你运行一个消息队列服务。如果不方便运行消息队列服务，你也许会考虑另外一种 [任务模式](#)。

本文所述的方法为每个工作项创建了一个 Pod。如果你的工作项仅需数秒钟，为每个工作项创建 Pod 会增加很多的常规消耗。可以考虑另外的方案请参考[示例](#)，这种方案可以实现每个 Pod 执行多个工作项。

示例中，我们使用 amqp-consume 从消息队列读取消息并执行我们真正的程序。这样的好处是你不需要修改你的程序使其知道队列的存在。要了解怎样使用客户端库和工作队列通信，请参考 [不同的示例](#)。

友情提醒

如果设置的完成数量小于队列中的消息数量，会导致一部分消息项不会被执行。

如果设置的完成数量大于队列中的消息数量，当队列中所有的消息都处理完成后，Job 也会显示为未完成。Job 将创建 Pod 并阻塞等待消息输入。

当发生下面两种情况时，即使队列中所有的消息都处理完了，Job 也不会显示为完成状态：

- 在 amqp-consume 命令拿到消息和容器成功退出之间的时段内，执行杀死容器操作；
- 在 kubelet 向 api-server 传回 Pod 成功运行之前，发生节点崩溃。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 16, 2020 at 4:31 PM PST: [\[zh\] Tidy up and fix links in tasks section \(9/10\) \(dabb6d689\)](#)

使用工作队列进行精细的并行处理

在这个例子中，我们会运行一个Kubernetes Job，其中的 Pod 会运行多个并行工作进程。

在这个例子中，当每个pod被创建时，它会从一个任务队列中获取一个工作单元，处理它，然后重复，直到到达队列的尾部。

下面是这个示例的步骤概述：

1. **启动存储服务用于保存工作队列。** 在这个例子中，我们使用 Redis 来存储工作项。在上一个例子中，我们使用了 RabbitMQ。在这个例子中，由于 AMQP 不

能为客户端提供一个良好的方法来检测一个有限长度的工作队列是否为空，我们使用了 Redis 和一个自定义的工作队列客户端库。在实践中，你可能会设置一个类似于 Redis 的存储库，并将其同时用于多项任务或其他事务的工作队列。

1. **创建一个队列，然后向其中填充消息。** 每个消息表示一个将要被处理的工作任务。在这个例子中，消息只是一个我们将用于进行长度计算的整数。
1. **启动一个 Job 对队列中的任务进行处理。** 这个 Job 启动了若干个 Pod。每个 Pod 从消息队列中取出一个工作任务，处理它，然后重复，直到到达队列的尾部。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.8. 要获知版本信息，请输入 kubectl version.

熟悉基本的、非并行的 [Job](#)。

启动 Redis

对于这个例子，为了简单起见，我们将启动一个单实例的 Redis。了解如何部署一个可伸缩、高可用的 Redis 例子，请查看 [Redis 示例](#)

你可以直接下载如下文件：

- [redis-pod.yaml](#)
- [redis-service.yaml](#)
- [Dockerfile](#)
- [job.yaml](#)
- [rediswq.py](#)
- [worker.py](#)

使用任务填充队列

现在，让我们往队列里添加一些“任务”。在这个例子中，我们的任务只是一些将被打印出来的字符串。

启动一个临时的可交互的 pod 用于运行 Redis 命令行界面。

```
kubectl run -i --tty temp --image redis --command "/bin/sh"
```

```
Waiting for pod default/redis2-c7h78 to be running, status is Pending, pod
ready: false
Hit enter for command prompt
```

现在按回车键，启动 redis 命令行界面，然后创建一个存在若干个工作项的列表。

```
# redis-cli -h redis
redis:6379> rpush job2 "apple"
(integer) 1
redis:6379> rpush job2 "banana"
(integer) 2
redis:6379> rpush job2 "cherry"
(integer) 3
redis:6379> rpush job2 "date"
(integer) 4
redis:6379> rpush job2 "fig"
(integer) 5
redis:6379> rpush job2 "grape"
(integer) 6
redis:6379> rpush job2 "lemon"
(integer) 7
redis:6379> rpush job2 "melon"
(integer) 8
redis:6379> rpush job2 "orange"
(integer) 9
redis:6379> lrange job2 0 -1
1) "apple"
2) "banana"
3) "cherry"
4) "date"
5) "fig"
6) "grape"
7) "lemon"
8) "melon"
9) "orange"
```

因此，这个键为 job2 的列表就是我们的工作队列。

注意：如果你还没有正确地配置 Kube DNS，你可能需要将上面的第一步改为 redis-cli -h \$REDIS_SERVICE_HOST。

创建镜像

现在我们已经准备好创建一个我们要运行的镜像

我们会使用一个带有 redis 客户端的 python 工作程序从消息队列中读出消息。

这里提供了一个简单的 Redis 工作队列客户端库，叫 rediswq.py ([下载](#))。

Job 中每个 Pod 内的 "工作程序" 使用工作队列客户端库获取工作。如下：

[application/job/redis/worker.py](#)



```
#!/usr/bin/env python

import time
import rediswq

host="redis"
# Uncomment next two lines if you do not have Kube-DNS working.
# import os
# host = os.getenv("REDIS_SERVICE_HOST")

q = rediswq.RedisWQ(name="job2", host="redis")
print("Worker with sessionID: " + q.sessionID())
print("Initial queue state: empty=" + str(q.empty()))
while not q.empty():
    item = q.lease(lease_secs=10, block=True, timeout=2)
    if item is not None:
        itemstr = item.decode("utf-8")
        print("Working on " + itemstr)
        time.sleep(10) # Put your actual work here instead of sleep.
        q.complete(item)
    else:
        print("Waiting for work")
print("Queue empty, exiting")
```

你也可以下载 [worker.py](#)、[rediswq.py](#) 和 [Dockerfile](#)。然后构建镜像：

```
docker build -t job-wq-2 .
```

Push 镜像

对于 [Docker Hub](#)，请先用你的用户名给镜像打上标签，然后使用下面的命令 push 你的镜像到仓库。请将 <username> 替换为自己的用户名。

```
docker tag job-wq-2 <username>/job-wq-2
docker push <username>/job-wq-2
```

你需要将镜像 push 到一个公共仓库或者 [配置集群访问你的私有仓库](#)。

如果你使用的是 [Google Container Registry](#)，请先用你的 project ID 给你的镜像打上标签，然后 push 到 GCR。请将 <project> 替换为自己的 project ID

```
docker tag job-wq-2 gcr.io/<project>/job-wq-2  
gcloud docker -- push gcr.io/<project>/job-wq-2
```

定义一个 Job

这是 job 定义：

[application/job/redis/job.yaml](#)



```
apiVersion: batch/v1  
kind: Job  
metadata:  
  name: job-wq-2  
spec:  
  parallelism: 2  
  template:  
    metadata:  
      name: job-wq-2  
    spec:  
      containers:  
      - name: c  
        image: gcr.io/myproject/job-wq-2  
      restartPolicy: OnFailure
```

请确保将 job 模板中的 gcr.io/myproject 更改为你自己的路径。

在这个例子中，每个 pod 处理了队列中的多个项目，直到队列中没有项目时便退出。因为是由工作程序自行检测工作队列是否为空，并且 Job 控制器不知道工作队列的存在，所以依赖于工作程序在完成工作时发出信号。工作程序以成功退出的形式发出信号表示工作队列已经为空。所以，只要有任意一个工作程序成功退出，控制器就知道工作已经完成了，所有的 Pod 将很快会退出。因此，我们将 Job 的完成计数 (Completion Count) 设置为 1。尽管如此，Job 控制器还是会等待其它 Pod 完成。

运行 Job

现在运行这个 Job：

```
kubectl apply -f ./job.yaml
```

稍等片刻，然后检查这个 Job。

```
kubectl describe jobs/job-wq-2
```

```
Name:      job-wq-2  
Namespace: default  
Selector:   controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
```

```
Labels: controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
       job-name=job-wq-2
Annotations: <none>
Parallelism: 2
Completions: <unset>
Start Time: Mon, 11 Jan 2016 17:07:59 -0800
Pods Statuses: 1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels: controller-uid=b1c7e4e3-92e1-11e7-b85e-fa163ee3c11f
          job-name=job-wq-2
  Containers:
    C:
      Image: gcr.io/exampleproject/job-wq-2
      Port:
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Events:
    FirstSeen  LastSeen  Count  From           SubobjectPath  Type
Reason        Message
-----  -----
33s          33s      1      {job-controller }          Normal        SuccessfulCreate
Created pod: job-wq-2-lglf8
```

查看日志：

```
kubectl logs pods/job-wq-2-7r7b2
```

```
Worker with sessionID: bbd72d0a-9e5c-4dd6-abf6-416cc267991f
Initial queue state: empty=False
Working on banana
Working on date
Working on lemon
```

你可以看到，其中的一个 pod 处理了若干个工作单元。

替代方案

如果你不方便运行一个队列服务或者修改你的容器用于运行一个工作队列，你可以考虑其它的 [Job 模式](#)。

如果你有连续的后台处理业务，那么可以考虑使用 replicationController 来运行你的后台业务，和运行一个类似 <https://github.com/resque/resque> 的后台处理库。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 22, 2020 at 7:20 PM PST: [Remove execute permission of markdown files, not only glossary term pages but all \(5661c9095\)](#)

访问集群中的应用程序

配置负载平衡、端口转发或设置防火墙或 DNS 配置，以访问集群中的应用程序。

[Web 界面 \(Dashboard\)](#)

[访问集群](#)

[使用端口转发来访问集群中的应用](#)

[使用服务来访问集群中的应用](#)

[使用 Service 把前端连接到后端](#)

[创建外部负载均衡器](#)

[列出集群中所有运行容器的镜像](#)

[在 Minikube 环境中使用 NGINX Ingress 控制器配置 Ingress](#)

[为集群配置 DNS](#)

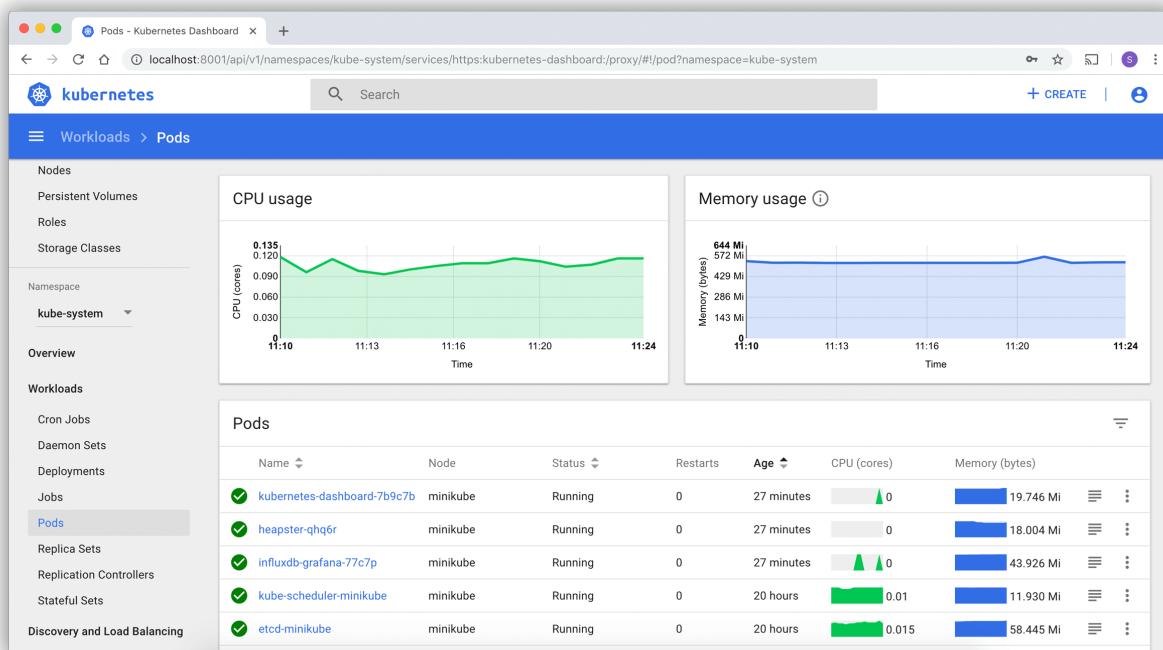
[同 Pod 内的容器使用共享卷通信](#)

[配置对多集群的访问](#)

Web 界面 (Dashboard)

Dashboard 是基于网页的 Kubernetes 用户界面。你可以使用 Dashboard 将容器应用部署到 Kubernetes 集群中，也可以对容器应用排错，还能管理集群资源。你可以使用 Dashboard 获取运行在集群中的应用的概览信息，也可以创建或者修改 Kubernetes 资源（如 Deployment, Job, DaemonSet 等等）。例如，你可以对 Deployment 实现弹性伸缩、发起滚动升级、重启 Pod 或者使用向导创建新的应用。

Dashboard 同时展示了 Kubernetes 集群中的资源状态信息和所有报错信息。



部署 Dashboard UI

默认情况下不会部署 Dashboard。可以通过以下命令部署：

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml
```

访问 Dashboard UI

为了保护你的集群数据，默认情况下，Dashboard 会使用最少的 RBAC 配置进行部署。当前，Dashboard 仅支持使用 Bearer 令牌登录。要为此样本演示创建令牌，你可以按照 [创建示例用户](#) 上的指南进行操作。

警告：在教程中创建的样本用户将具有管理特权，并且仅用于教育目的。

命令行代理

你可以使用 kubectl 命令行工具访问 Dashboard，命令如下：

```
kubectl proxy
```

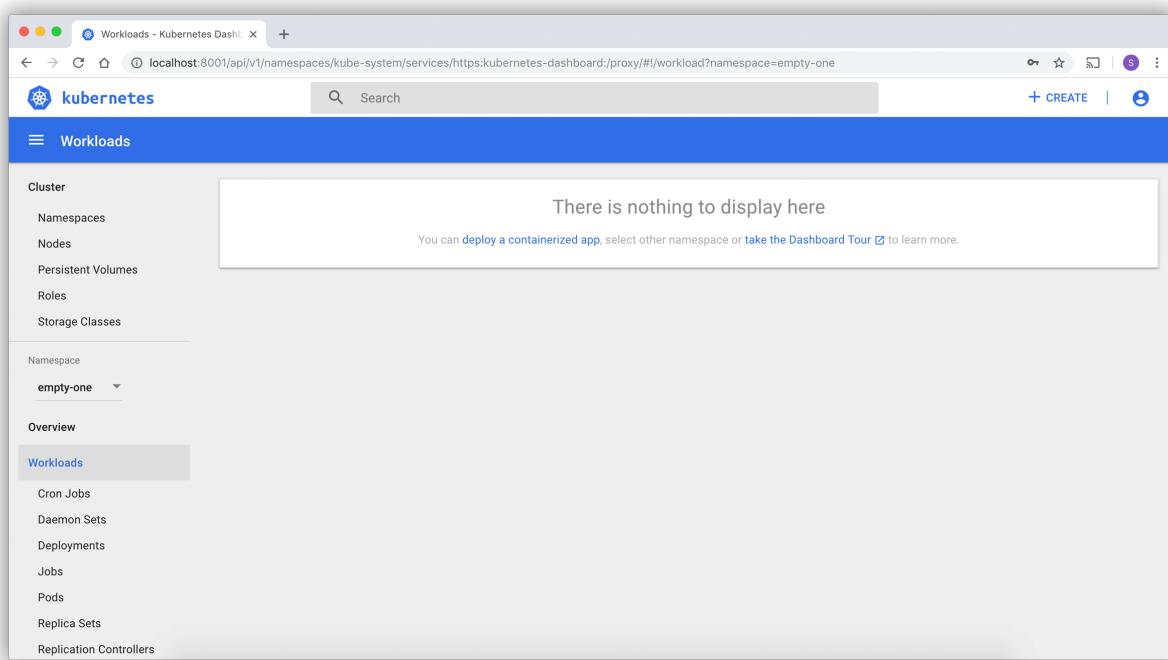
kubectl 会使得 Dashboard 可以通过 <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/> 访问。

UI 只能 通过执行这条命令的机器进行访问。更多选项参见 kubectl proxy --help。

说明：Kubeconfig 身份验证方法不支持外部身份提供程序或基于 x509 证书的身份验证。

欢迎界面

当访问空集群的 Dashboard 时，你会看到欢迎界面。页面包含一个指向此文档的链接，以及一个用于部署第一个应用程序的按钮。此外，你可以看到在默认情况下有哪些默认系统应用运行在 kube-system 名字空间中，比如 Dashboard 自己。



部署容器化应用

通过一个简单的部署向导，你可以使用 Dashboard 将容器化应用作为一个 Deployment 和可选的 Service 进行创建和部署。可以手工指定应用的详细配置，或者上传一个包含应用配置的 YAML 或 JSON 文件。

点击任何页面右上角的 **CREATE** 按钮以开始。

指定应用的详细配置

部署向导需要你提供以下信息：

- **应用名称**（必填）：应用的名称。内容为应用名称的 [标签](#) 会被添加到任何将被部署的 Deployment 和 Service。

在选定的 Kubernetes [名字空间](#) 中，应用名称必须唯一。必须由小写字母开头，以数字或者小写字母结尾，并且只含有小写字母、数字和中划线（-）。小于等于 24 个字符。开头和结尾的空格会被忽略。

- **容器镜像**（必填）：公共镜像仓库上的 Docker [容器镜像](#) 或者私有镜像仓库（通常是 Google Container Registry 或者 Docker Hub）的 URL。容器镜像参数说明必须以冒号结尾。

- **Pod 的数量**（必填）：你希望应用程序部署的 Pod 的数量。值必须为正整数。

系统会创建一个 [Deployment](#) 以保证集群中运行期望的 Pod 数量。

- **服务**（可选）：对于部分应用（比如前端），你可能想对外暴露一个 [Service](#)，这个 Service 可能用的是集群之外的公网 IP 地址（外部 Service）。

说明：对于外部服务，你可能需要开放一个或多个端口才行。

其它只能对集群内部可见的 Service 称为内部 Service。

不管哪种 Service 类型，如果你选择创建一个 Service，而且容器在一个端口上开启了监听（入向的），那么你需要定义两个端口。创建的 Service 会把（入向的）端口映射到容器可见的目标端口。该 Service 会把流量路由到你部署的 Pod。支持的协议有 TCP 和 UDP。这个 Service 的内部 DNS 解析名就是之前你定义的应用名称的值。

如果需要，你可以打开 **Advanced Options** 部分，这里你可以定义更多设置：

- **描述**：这里你输入的文本会作为一个 [注解](#) 添加到 Deployment，并显示在应用的详细信息中。
- **标签**：应用默认使用的 [标签](#) 是应用名称和版本。你可以为 Deployment、Service（如果有）定义额外的标签，比如 release（版本）、environment（环境）、tier（层级）、partition（分区）和 release track（版本跟踪）。

例子：

```
release=1.0
tier=frontend
environment=prod
track=stable
```

- **名字空间**：Kubernetes 支持多个虚拟集群依附于同一个物理集群。这些虚拟集群被称为 [名字空间](#)，可以让你将资源划分为逻辑命名的组。

Dashboard 通过下拉菜单提供所有可用的名字空间，并允许你创建新的名字空间。名字空间的名称最长可以包含 63 个字母或数字和中横线（-），但是不能包含大写字母。

名字空间的名称不能只包含数字。如果名字被设置成一个数字，比如 10，pod 就

在名字空间创建成功的情况下，默认会使用新创建的名字空间。如果创建失败，那么第一个名字空间会被选中。

- **镜像拉取 Secret**：如果要使用私有的 Docker 容器镜像，需要拉取 [Secret](#) 凭证。

Dashboard 通过下拉菜单提供所有可用的 Secret，并允许你创建新的 Secret。Secret 名称必须遵循 DNS 域名语法，比如 new.image-pull.secret。Secret 的内容必须是 base64 编码的，并且在一个 [.dockercfg](#) 文件中声明。Secret 名称最大可以包含 253 个字符。

在镜像拉取 Secret 创建成功的情况下，默认会使用新创建的 Secret。如果创建失败，则不会使用任何 Secret。

- **CPU 需求（核数）和内存需求（MiB）**：你可以为容器定义最小的 [资源限制](#)。默认情况下，Pod 没有 CPU 和内存限制。
- **运行命令和运行命令参数**：默认情况下，你的容器会运行 Docker 镜像的默认 [入口命令](#)。你可以使用 command 选项覆盖默认值。
- **以特权模式运行**：这个设置决定了在 [特权容器](#) 中运行的进程是否像主机中使用 root 运行的进程一样。特权容器可以使用诸如操纵网络堆栈和访问设备的功能。
- **环境变量**：Kubernetes 通过 [环境变量](#) 暴露 Service。你可以构建环境变量，或者将环境变量的值作为参数传递给你的命令。它们可以被应用用于查找 Service。值可以通过 \$(VAR_NAME) 语法关联其他变量。

上传 YAML 或者 JSON 文件

Kubernetes 支持声明式配置。所有的配置都存储在遵循 Kubernetes [API](#) 规范的 YAML 或者 JSON 配置文件中。

作为一种替代在部署向导中指定应用详情的方式，你可以在 YAML 或者 JSON 文件中定义应用，并且使用 Dashboard 上传文件：

使用 Dashboard

以下各节描述了 Kubernetes Dashboard UI 视图；包括它们提供的内容，以及怎么使用它们。

导航

当在集群中定义 Kubernetes 对象时，Dashboard 会在初始视图中显示它们。默认情况下只会显示 默认 名字空间中的对象，可以通过更改导航栏菜单中的名字空间筛选器进行改变。

Dashboard 展示大部分 Kubernetes 对象，并将它们分组放在几个菜单类别中。

管理概述

集群和名字空间管理的视图，Dashboard 会列出节点、名字空间和持久卷，并且有它们的详细视图。节点列表视图包含从所有节点聚合的 CPU 和内存使用的度量值。详细信息视图显示了一个节点的度量值，它的规格、状态、分配的资源、事件和这个节点上运行的 Pod。

负载

显示选中的名字空间中所有运行的应用。视图按照负载类型（如 Deployment、ReplicaSet、StatefulSet 等）罗列应用，并且每种负载都可以单独查看。列表总结了关于负载的可执行信息，比如一个 ReplicaSet 的准备状态的 Pod 数量，或者目前一个 Pod 的内存使用量。

工作负载的详情视图展示了对象的状态、详细信息和相互关系。例如，ReplicaSet 所控制的 Pod，或者 Deployment 关联的新 ReplicaSet 和 Pod 水平扩展控制器。

服务

展示允许暴露给外网服务和允许集群内部发现的 Kubernetes 资源。因此，Service 和 Ingress 视图展示他们关联的 Pod、给集群连接使用的内部端点和给外部用户使用的外部端点。

存储

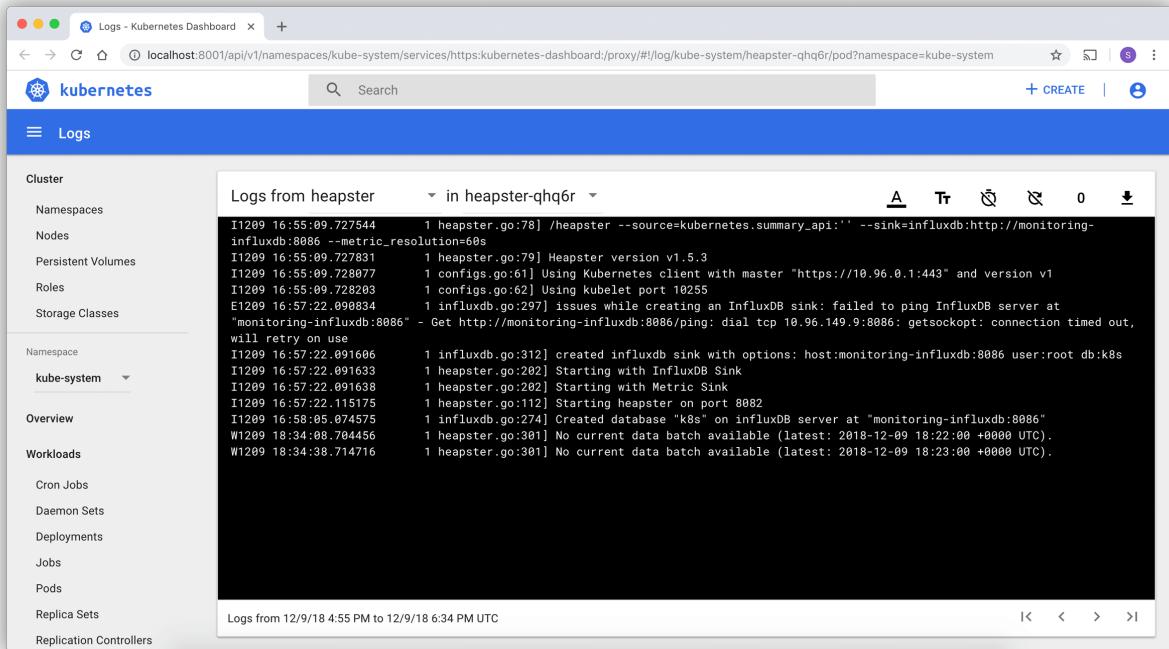
存储视图展示持久卷申领（PVC）资源，这些资源被应用程序用来存储数据。

ConfigMap 和 Secret

展示的所有 Kubernetes 资源是在集群中运行的应用程序的实时配置。通过这个视图可以编辑和管理配置对象，并显示那些默认隐藏的 secret。

日志查看器

Pod 列表和详细信息页面可以链接到 Dashboard 内置的日志查看器。查看器可以钻取属于同一个 Pod 的不同容器的日志。



接下来

更多信息，参见 [Kubernetes Dashboard 项目页面](#).

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 5:27 PM PST: [Remove "Configure Your Cloud Provider's Firewalls" for zh \(893570b79\)](#)

访问集群

本文阐述多种与集群交互的方法。

- [使用 kubectl 完成集群的第一次访问](#)
- [直接访问 REST API](#)
 - [使用 kubectl proxy](#)
 - [不使用 kubectl proxy](#)
- [以编程方式访问 API](#)
 - [Go 客户端](#)
 - [Python 客户端](#)

- [其它语言](#)
- [从 Pod 中访问 API](#)
- [访问集群中正在运行的服务](#)
 - [连接的方法](#)
 - [发现内建服务](#)
 - [使用 web 浏览器访问运行在集群上的服务](#)
- [请求重定向](#)
- [多种代理](#)

使用 kubectl 完成集群的第一次访问

当你第一次访问 Kubernetes API 的时候，我们建议你使用 Kubernetes CLI，kubectl。

访问集群时，你需要知道集群的地址并且拥有访问的凭证。通常，这些在你通过[启动安装](#)安装集群时都是自动安装好的，或者其他人安装时也应该提供了凭证和集群地址。

通过以下命令检查 kubectl 是否知道集群地址及凭证：

```
kubectl config view
```

有许多[例子](#)介绍了如何使用 kubectl，可以在[kubectl手册](#)中找到更完整的文档。

直接访问 REST API

Kubectl 处理 apiserver 的定位和身份验证。如果要使用 curl 或 wget 等 http 客户端或浏览器直接访问 REST API，可以通过多种方式查找和验证：

- 以代理模式运行 kubectl。
 - 推荐此方式。
 - 使用已存储的 apiserver 地址。
 - 使用自签名的证书来验证 apiserver 的身份。杜绝 MITM 攻击。
 - 对 apiserver 进行身份验证。
 - 未来可能会实现智能化的客户端负载均衡和故障恢复。
- 直接向 http 客户端提供位置和凭据。
 - 可选的方案。
 - 适用于代理可能引起混淆的某些客户端类型。
 - 需要引入根证书到你的浏览器以防止 MITM 攻击。

使用 kubectl proxy

以下命令以反向代理的模式运行 kubectl。它处理 apiserver 的定位和验证。像这样运行：

```
kubectl proxy --port=8080 &
```

参阅[kubectl proxy](#) 获取更多详细信息。

然后，你可以使用 curl、wget 或浏览器访问 API，如果是 IPv6 则用 [::1] 替换 localhost，如下所示：

```
curl http://localhost:8080/api/
```

```
{  
  "kind": "APIVersions",  
  "versions": [  
    "v1"  
,  
    "serverAddressByClientCIDRs": [  
      {  
        "clientCIDR": "0.0.0.0/0",  
        "serverAddress": "10.0.1.149:443"  
      }  
    ]  
  ]  
}
```

不使用 kubectl proxy

在 Kubernetes 1.3 或更高版本中，kubectl config view 不再显示 token。使用 kubectl describe secret ... 来获取默认服务帐户的 token，如下所示：

grep/cut 方法实现：

```
APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d "\n")  
TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -d ' ') | grep -E '^token' | cut -f2 -d ':' | tr -d '\n')  
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

```
{  
  "kind": "APIVersions",  
  "versions": [  
    "v1"  
,  
    "serverAddressByClientCIDRs": [  
      {  
        "clientCIDR": "0.0.0.0/0",  
        "serverAddress": "10.0.1.149:443"  
      }  
    ]  
  ]  
}
```

jsonpath 方法实现：

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')  
TOKEN=$(kubectl get secret $(kubectl get serviceaccount default -o jsonpath='{.s
```

```
crets[0].name}') -o jsonpath='{.data.token}' | base64 --decode )  
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure  
  
{  
  "kind": "APIVersions",  
  "versions": [  
    "v1"  
  ],  
  "serverAddressByClientCIDRs": [  
    {  
      "clientCIDR": "0.0.0.0/0",  
      "serverAddress": "10.0.1.149:443"  
    }  
  ]  
}
```

上面的例子使用了 `--insecure` 参数，这使得它很容易受到 MITM 攻击。当 `kubectl` 访问集群时，它使用存储的根证书和客户端证书来访问服务器（它们安装在 `~/.kube` 目录中）。由于集群证书通常是自签名的，因此可能需要特殊配置才能让你的 http 客户端使用根证书。

在一些集群中，apiserver 不需要身份验证；它可能只服务于 `localhost`，或者被防火墙保护，这个没有一定的标准。[配置对 API 的访问](#) 描述了集群管理员如何进行配置。此类方法可能与未来的高可用性支持相冲突。

以编程方式访问 API

Kubernetes 官方提供对 [Go](#) 和 [Python](#) 的客户端库支持。

Go 客户端

- 想要获得这个库，请运行命令：`go get k8s.io/client-go/<version number>/kubernetes`。参阅 <https://github.com/kubernetes/client-go> 来查看目前支持哪些版本。
- 基于这个 `client-go` 客户端库编写应用程序。请注意，`client-go` 定义了自己的 API 对象，因此如果需要，请从 `client-go` 而不是从主存储库导入 API 定义，例如，`import "k8s.io/client-go/1.4/pkg/api/v1"` 才是对的。

Go 客户端可以像 `kubectl` CLI 一样使用相同的 [kubeconfig](#) 文件 来定位和验证 apiserver。可参阅 [示例](#)。

如果应用程序以 Pod 的形式部署在集群中，那么请参阅 [下一章](#)。

Python 客户端

如果想要使用 [Python 客户端](#)，请运行命令：`pip install kubernetes`。参阅 [Python Client Library page](#) 以获得更详细的安装参数。

Python 客户端可以像 kubectl CLI 一样使用相同的 [kubeconfig 文件](#) 来定位和验证 apiserver，可参阅 [示例](#)。

其它语言

目前有多个[客户端库](#)为其它语言提供访问 API 的方法。参阅其它库的相关文档以获取他们是如何验证的。

从 Pod 中访问 API

当你从 Pod 中访问 API 时，定位和验证 apiserver 会有些许不同。

在 Pod 中定位 apiserver 的推荐方式是通过 kubernetes.default.svc 这个 DNS 名称，该名称将会解析为服务 IP，然后服务 IP 将会路由到 apiserver。

向 apiserver 进行身份验证的推荐方法是使用 [服务帐户](#)凭据。通过 kube-system，Pod 与服务帐户相关联，并且该服务帐户的凭证（token）被放置在该 Pod 中每个容器的文件系统中，位于 /var/run/secrets/kubernetes.io/serviceaccount/token。

如果可用，则将证书放入每个容器的文件系统中的 /var/run/secrets/kubernetes.io/serviceaccount/ca.crt，并且应该用于验证 apiserver 的服务证书。

最后，名字空间作用域的 API 操作所使用的 default 名字空间将被放置在每个容器的 /var/run/secrets/kubernetes.io/serviceaccount/namespace 文件中。

在 Pod 中，建议连接 API 的方法是：

- 在 Pod 的边车容器中运行 kubectl proxy，或者以后台进程的形式运行。这将把 Kubernetes API 代理到当前 Pod 的 localhost 接口，所以 Pod 中的所有容器中的进程都能访问它。
- 使用 Go 客户端库，并使用 rest.InClusterConfig() 和 kubernetes.NewForConfig() 函数创建一个客户端。他们处理 apiserver 的定位和身份验证。[示例](#)

在每种情况下，Pod 的凭证都是为了与 apiserver 安全地通信。

访问集群中正在运行的服务

上一节介绍了如何连接 Kubernetes API 服务。本节介绍如何连接到 Kubernetes 集群上运行的其他服务。在 Kubernetes 中，[节点](#)、[pods](#) 和 [服务](#)都有自己的 IP。在许多情况下，集群上的节点 IP、Pod IP 和某些服务 IP 将无法路由，因此无法从集群外部的计算机（例如桌面计算机）访问它们。

连接的方法

有多种方式可以从集群外部连接节点、Pod 和服务：

- 通过公共 IP 访问服务。
 - 类型为 NodePort 或 LoadBalancer 的服务，集群外部可以访问。请参阅 [服务](#) 和 [kubectl expose](#) 文档。
 - 取决于你的集群环境，该服务可能仅暴露给你的公司网络，或者也可能暴露给整个互联网。请考虑公开该服务是否安全。它是否进行自己的身份验证？
 - 在服务后端放置 Pod。要从一组副本中访问一个特定的 Pod，例如进行调试，请在 Pod 上设置一个唯一的标签，然后创建一个选择此标签的新服务。
 - 在大多数情况下，应用程序开发人员不应该通过其 nodeIP 直接访问节点。
- 使用 proxy 动词访问服务、节点或者 Pod。
 - 在访问远程服务之前进行 apiserver 身份验证和授权。如果服务不能够安全地暴露到互联网，或者服务不能获得节点 IP 端口的访问权限，或者是为了调试，那么请使用此选项。
 - 代理可能会给一些 web 应用带来问题。
 - 只适用于 HTTP/HTTPS。
 - 更多详细信息在[这里](#)。
- 从集群中的节点或者 Pod 中访问。
 - 运行一个 Pod，然后使用 [kubectl exec](#) 来连接 Pod 里的 Shell。然后从 Shell 中连接其它的节点、Pod 和服务。
 - 有些集群可能允许你通过 SSH 连接到节点，从那你可以访问集群的服务。这是一个非正式的方式，可能可以运行在个别的集群上。浏览器和其它一些工具可能没有被安装。集群的 DNS 可能无法使用。

发现内建服务

通常来说，集群中会有 kube-system 创建的一些运行的服务。

通过 kubectl cluster-info 命令获得这些服务列表：

```
kubectl cluster-info
```

```
Kubernetes master is running at https://104.197.5.247
elasticsearch-logging is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-
system/services/kibana-logging/proxy
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/
services/kube-dns/proxy
grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/
services/monitoring-grafana/proxy
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/
services/monitoring-heapster/proxy
```

这展示了访问每个服务的 proxy-verb URL。例如，如果集群启动了集群级别的日志（使用 Elasticsearch），并且传递合适的凭证，那么可以通过 `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` 进行访问。日志也能通过 kubectl 代理获取，例如：`http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/。`（参阅[使用 Kubernetes API 访问集群](#)了解如何传递凭据，或者使用 kubectl proxy）

手动构建 apiserver 代理 URL

如上所述，你可以使用 kubectl cluster-info 命令来获得服务的代理 URL。要创建包含服务端点、后缀和参数的代理 URL，只需添加到服务的代理 URL：`http://kubernetes_master_address/api/v1/namespaces/namespace_name/services/service_name[:port_name]/proxy`

如果尚未为端口指定名称，则不必在 URL 中指定 `port_name`。

默认情况下，API server 使用 HTTP 代理你的服务。要使用 HTTPS，请在服务名称前加上 https： `http://kubernetes_master_address/api/v1/namespaces/namespace_name/services/https:service_name:[port_name]/proxy`

URL 名称段支持的格式为：

- <service_name> - 使用 http 代理到默认或未命名的端口
- <service_name>:<port_name> - 使用 http 代理到指定的端口
- https:<service_name>: - 使用 https 代理到默认或未命名的端口（注意后面的冒号）
- https:<service_name>:<port_name> - 使用 https 代理到指定的端口

示例

- 要访问 Elasticsearch 服务端点 `_search?q=user:kimchy`，你需要使用：`http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_search?q=user:kimchy`
- 要访问 Elasticsearch 集群健康信息 `_cluster/health?pretty=true`，你需要使用：`https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_cluster/health?pretty=true`

```
{  
  "cluster_name": "kubernetes_logging",  
  "status": "yellow",  
  "timed_out": false,  
  "number_of_nodes": 1,  
  "number_of_data_nodes": 1,  
  "active_primary_shards": 5,  
  "active_shards": 5,  
  "relocating_shards": 0,  
  "initializing_shards": 0,
```

```
        "unassigned_shards" : 5  
    }
```

使用 web 浏览器访问运行在集群上的服务

你可以在浏览器地址栏中输入 apiserver 代理 URL。但是：

- Web 浏览器通常不能传递令牌，因此你可能需要使用基本（密码）身份验证。
Apiserver 可以配置为接受基本身份验证，但你的集群可能未进行配置。
- 某些 Web 应用程序可能无法运行，尤其是那些使用客户端 javascript 以不知道代理路径前缀的方式构建 URL 的应用程序。

请求重定向

重定向功能已弃用并被删除。请改用代理（见下文）。

多种代理

使用 Kubernetes 时可能会遇到几种不同的代理：

1. [kubectl](#) 代理：

- 在用户的桌面或 Pod 中运行
- 代理从本地主机地址到 Kubernetes apiserver
- 客户端到代理将使用 HTTP
- 代理到 apiserver 使用 HTTPS
- 定位 apiserver
- 添加身份验证头部

1. [apiserver](#) 代理：

- 内置于 apiserver 中
- 将集群外部的用户连接到集群 IP，否则这些 IP 可能无法访问
- 运行在 apiserver 进程中
- 客户端代理使用 HTTPS（也可配置为 http）
- 代理将根据可用的信息决定使用 HTTP 或者 HTTPS 代理到目标
- 可用于访问节点、Pod 或服务
- 在访问服务时进行负载平衡

1. [kube proxy](#)：

- 运行在每个节点上
- 代理 UDP 和 TCP
- 不能代理 HTTP
- 提供负载均衡
- 只能用来访问服务

1. 位于 apiserver 之前的 Proxy/Load-balancer：

- 存在和实现因集群而异（例如 nginx）

- 位于所有客户和一个或多个 apiserver 之间
- 如果有多个 apiserver，则充当负载均衡器

1. 外部服务上的云负载均衡器：

- 由一些云提供商提供（例如 AWS ELB，Google Cloud Load Balancer）
- 当 Kubernetes 服务类型为 LoadBalancer 时自动创建
- 只使用 UDP/TCP
- 具体实现因云提供商而异。

除了前两种类型之外，Kubernetes 用户通常不需要担心任何其他问题。 集群管理员通常会确保后者的正确配置。

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 December 15, 2020 at 11:12 AM PST: [Update access-cluster.md \(04986e710\)](#)

使用端口转发来访问集群中的应用

本文展示如何使用 kubectl port-forward 连接到在 Kubernetes 集群中运行的 Redis 服务。这种类型的连接对数据库调试很有用。

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.
- 安装 [redis-cli](#).

创建 Redis deployment 和服务

1. 创建一个 Redis deployment :

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

查看输出是否成功，以验证是否成功创建 deployment：

```
deployment.apps/redis-master created
```

查看 pod 状态，检查其是否准备就绪：

```
kubectl get pods
```

输出显示创建的 pod：

NAME	READY	STATUS	RESTARTS	AGE
redis-master-765d459796-258hz	1/1	Running	0	50s

查看 deployment 状态：

```
kubectl get deployment
```

输出显示创建的 deployment：

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
redis-master	1	1	1	1	55s

查看 replicaset 状态：

```
kubectl get rs
```

输出显示创建的 replicaset：

NAME	DESIRED	CURRENT	READY	AGE
redis-master-765d459796	1	1	1	1m

1. 创建一个 Redis 服务：

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

查看输出是否成功，以验证是否成功创建 service：

```
service/redis-master created
```

检查 service 是否创建：

```
kubectl get svc | grep redis
```

输出显示创建的 service：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
redis-master	ClusterIP	10.0.0.213	<none>	6379/TCP	27s

1. 验证 Redis 服务是否运行在 pod 中并且监听 6379 端口：

```
kubectl get pods redis-master-765d459796-258hz \
--template='{{(index .spec.containers 0).ports 0}.containerPort}}
{{"\n"}}'
```

输出应该显示端口：

```
6379
```

转发一个本地端口到 pod 端口

- 从 Kubernetes v1.10 开始，kubectl port-forward 允许使用资源名称（例如 pod 名称）来选择匹配的 pod 来进行端口转发。

```
kubectl port-forward redis-master-765d459796-258hz 7000:6379
```

这相当于

```
kubectl port-forward pods/redis-master-765d459796-258hz 7000:6379
```

或者

```
kubectl port-forward deployment/redis-master 7000:6379
```

或者

```
kubectl port-forward rs/redis-master 7000:6379
```

或者

```
kubectl port-forward svc/redis-master 7000:redis
```

以上所有命令都应该有效。输出应该类似于：

```
I0710 14:43:38.274550 3655 portforward.go:225] Forwarding from
127.0.0.1:7000 -> 6379
I0710 14:43:38.274797 3655 portforward.go:225] Forwarding from [::1]:
7000 -> 6379
```

- 启动 Redis 命令行接口：

```
redis-cli -p 7000
```

- 在 Redis 命令行提示符下，输入 ping 命令：

```
127.0.0.1:7000>ping
```

成功的 ping 请求应该返回 PONG。

讨论

与本地 7000 端口建立的连接将转发到运行 Redis 服务器的 pod 的 6379 端口。通过此连接，您可以使用本地工作站来调试在 pod 中运行的数据库。

警告：由于已知的限制，目前的端口转发仅适用于 TCP 协议。在 [issue 47862](#) 中正在跟踪对 UDP 协议的支持。

接下来

进一步了解 [kubectl port-forward](#)。

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 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

使用服务来访问集群中的应用

本文展示如何创建一个 Kubernetes 服务对象，能让外部客户端访问在集群中运行的应用。该服务为一个应用的两个运行实例提供负载均衡。

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.

教程目标

- 运行 Hello World 应用的两个实例。
- 创建一个服务对象来暴露 node port.
- 使用服务对象来访问正在运行的应用。

为运行在两个 pod 中的应用创建一个服务

这是应用程序部署的配置文件：

[service/access/hello-application.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
      labels:
        run: load-balancer-example
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
              protocol: TCP
```

1. 在你的集群中运行一个 Hello World 应用： 使用上面的文件创建应用程序 Deployment：

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

上面的命令创建一个 [Deployment](#) 对象 和一个关联的 [ReplicaSet](#) 对象。 这个 ReplicaSet 有两个 [Pod](#)，每个 Pod 都运行着 Hello World 应用。

1. 展示 Deployment 的信息：

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

1. 展示你的 ReplicaSet 对象信息：

```
kubectl get replicsets
kubectl describe replicsets
```

1. 创建一个服务对象来暴露 Deployment：

```
kubectl expose deployment hello-world --type=NodePort --name=example-service
```

1. 展示 Service 信息：

```
kubectl describe services example-service
```

输出类似于：

```
Name:           example-service
Namespace:      default
Labels:         run=load-balancer-example
Annotations:    <none>
Selector:       run=load-balancer-example
Type:           NodePort
IP:             10.32.0.16
Port:           <unset> 8080/TCP
TargetPort:     8080/TCP
NodePort:       <unset> 31496/TCP
Endpoints:     10.200.1.4:8080,10.200.2.5:8080
Session Affinity: None
Events:         <none>
```

注意服务中的 NodePort 值。例如在上面的输出中，NodePort 是 31496。

1. 列出运行 Hello World 应用的 Pod：

```
kubectl get pods --selector="run=load-balancer-example" --output=wide
```

输出类似于：

NAME	READY	STATUS	... IP	NODE
hello-world-2895499144-bsbk5	1/1	Running	...	10.200.1.4 worker1
hello-world-2895499144-m1pwt	1/1	Running	...	10.200.2.5 worker2

1. 获取运行 Hello World 的 pod 的其中一个节点的公共 IP 地址。如何获得此地址取决于你设置集群的方式。例如，如果你使用的是 Minikube，则可以通过运行 kubectl cluster-info 来查看节点地址。如果你使用的是 Google Compute Engine 实例，则可以使用 gcloud compute instances list 命令查看节点的公共地址。
2. 在你选择的节点上，创建一个防火墙规则以开放节点端口上的 TCP 流量。例如，如果你的服务的 NodePort 值为 31568，请创建一个防火墙规则以允许 31568 端口上的 TCP 流量。不同的云提供商提供了不同方法来配置防火墙规则。
3. 使用节点地址和 node port 来访问 Hello World 应用：

```
curl http://<public-node-ip>:<node-port>
```

这里的 <public-node-ip> 是你节点的公共 IP 地址，<node-port> 是你服务的 NodePort 值。对于请求成功的响应是一个 hello 消息：

```
Hello Kubernetes!
```

使用服务配置文件

作为 kubectl expose 的替代方法，你可以使用 [服务配置文件](#) 来创建服务。

清理现场

想要删除服务，输入以下命令：

```
kubectl delete services example-service
```

想要删除运行 Hello World 应用的 Deployment、ReplicaSet 和 Pod，输入以下命令：

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

接下来

- 进一步了解[通过服务连接应用](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 16, 2020 at 11:05 AM PST: [\[zh\] Tidy up and fix links in tasks section \(8/10\) \(c2aae6890\)](#)

使用 Service 把前端连接到后端

本任务会描述如何创建前端微服务和后端微服务。后端微服务是一个 hello 欢迎程序。前端和后端的连接是通过 Kubernetes [服务](#) 完成的。

教程目标

- 使用部署对象 (Deployment object) 创建并运行一个微服务
- 从后端将流量路由到前端
- 使用服务对象把前端应用连接到后端应用

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.

本任务使用 [外部负载均衡服务](#)，所以需要对应的可支持此功能的环境。如果你的环境不能支持，你可以使用 [NodePort](#) 类型的服务代替。

使用部署对象（Deployment）创建后端

后端是一个简单的 hello 欢迎微服务应用。这是后端应用的 Deployment 配置文件：

[service/access/hello.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  selector:
    matchLabels:
      app: hello
      tier: backend
      track: stable
  replicas: 7
  template:
    metadata:
      labels:
        app: hello
        tier: backend
        track: stable
    spec:
      containers:
        - name: hello
          image: "gcr.io/google-samples/hello-go-gke:1.0"
          ports:
            - name: http
              containerPort: 80
```

创建后端 Deployment：

```
kubectl apply -f https://k8s.io/examples/service/access/hello.yaml
```

查看后端的 Deployment 信息：

```
kubectl describe deployment hello
```

输出类似于：

```
Name:           hello
Namespace:      default
CreationTimestamp: Mon, 24 Oct 2016 14:21:02 -0700
Labels:          app=hello
                  tier=backend
                  track=stable
Annotations:    deployment.kubernetes.io/revision=1
Selector:        app=hello,tier=backend,track=stable
Replicas:        7 desired | 7 updated | 7 total | 7 available | 0 unavailable
StrategyType:   RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Pod Template:
  Labels:  app=hello
          tier=backend
          track=stable
Containers:
  hello:
    Image:      "gcr.io/google-samples/hello-go-gke:1.0"
    Port:       80/TCP
    Environment: <none>
    Mounts:     <none>
    Volumes:    <none>
Conditions:
  Type  Status  Reason
  ----  -----  -----
  Available  True  MinimumReplicasAvailable
  Progressing  True  NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   hello-3621623197 (7/7 replicas created)
Events:
  ...
```

创建后端服务对象

前端连接到后端的关键是 Service (服务) 。 Service 创建一个固定 IP 和 DNS 解析名入口，使得后端微服务可达。 Service 使用 [选择算符](#) 来寻找目标 Pod。

首先，浏览 Service 的配置文件：

[service/access/hello-service.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: hello
spec:
  selector:
    app: hello
    tier: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: http
```

配置文件中，你可以看到 Service 将流量路由到包含 app: hello 和 tier: backend 标签的 Pod。

创建 hello Service：

```
kubectl apply -f https://k8s.io/examples/service/access/hello-service.yaml
```

此时，你已经有了一个在运行的后端 Deployment，你也有了一个 Service 用于路由网络流量。

创建前端应用

既然你已经有了后端应用，你可以创建一个前端应用连接到后端。前端应用通过 DNS 名连接到后端的工作 Pods。DNS 名是 "hello"，也就是 Service 配置文件中 name 字段的值。

前端 Deployment 中的 Pods 运行一个 nginx 镜像，这个已经配置好镜像去寻找后端的 hello Service。只是 nginx 的配置文件：

[service/access/frontend.conf](#)



```
upstream hello {
  server hello;
}

server {
  listen 80;

  location / {
    proxy_pass http://hello;
```

```
    }  
}
```

与后端类似，前端用包含一个 Deployment 和一个 Service。Service 的配置文件包含了 type: LoadBalancer，也就是说，Service 会使用你的云服务商的默认负载均衡设备。

[service/access/frontend.yaml](#)



```
apiVersion: v1  
kind: Service  
metadata:  
  name: frontend  
spec:  
  selector:  
    app: hello  
    tier: frontend  
  ports:  
  - protocol: "TCP"  
    port: 80  
    targetPort: 80  
  type: LoadBalancer  
---  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: frontend  
spec:  
  selector:  
    matchLabels:  
      app: hello  
      tier: frontend  
      track: stable  
  replicas: 1  
  template:  
    metadata:  
      labels:  
        app: hello  
        tier: frontend  
        track: stable  
    spec:  
      containers:  
      - name: nginx  
        image: "gcr.io/google-samples/hello-frontend:1.0"  
      lifecycle:
```

```
preStop:  
  exec:  
    command: ["/usr/sbin/nginx", "-s", "quit"]
```

创建前端 Deployment 和 Service :

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

通过输出确认两个资源都已经被创建 :

```
deployment.apps/frontend created  
service/frontend created
```

说明 : 这个 nginx 配置文件是被打包在 [容器镜像](#) 里的。更好的方法是使用 [ConfigMap](#) , 这样的话你可以更轻易地更改配置。

与前端 Service 交互

一旦你创建了 LoadBalancer 类型的 Service , 你可以使用这条命令查看外部 IP :

```
kubectl get service frontend
```

外部 IP 字段的生成可能需要一些时间。如果是这种情况 , 外部 IP 会显示为 <pending> 。

```
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE  
frontend  10.51.252.116  <pending>        80/TCP      10s
```

当外部 IP 地址被分配可用时 , 配置会更新 , 在 EXTERNAL-IP 头部下显示新的 IP :

```
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE  
frontend  10.51.252.116  XXX.XXX.XXX.XXX  80/TCP      1m
```

这一新的 IP 地址就可以用来从集群外与 frontend 服务交互了。

通过前端发送流量

前端和后端已经完成连接了。你可以使用 curl 命令通过你的前端 Service 的外部 IP 访问服务端点。

```
curl http://<EXTERNAL-IP>
```

后端生成的消息输出如下 :

```
{"message":"Hello"}
```

清理现场

要删除服务 , 输入下面的命令 :

```
kubectl delete services frontend hello
```

要删除在前端和后端应用中运行的 Deployment、ReplicaSet 和 Pod，输入下面的命令：

```
kubectl delete deployment frontend hello
```

接下来

- 进一步了解[Service](#)
- 进一步了解[ConfigMap](#)

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 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

创建外部负载均衡器

本文展示如何创建一个外部负载均衡器。

说明：此功能仅适用于支持外部负载均衡器的云提供商或环境。

创建服务时，你可以选择自动创建云网络负载均衡器。这提供了一个外部可访问的 IP 地址，可将流量分配到集群节点上的正确端口上（假设集群在支持的环境中运行，并配置了正确的云负载平衡器提供商包）。

有关如何配置和使用 Ingress 资源为服务提供外部可访问的 URL、负载均衡流量、终止 SSL 等功能，请查看 [Ingress](#) 文档。

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`.

配置文件

要创建外部负载均衡器，请将以下内容添加到 [服务配置文件](#)：

```
type: LoadBalancer
```

你的配置文件可能会如下所示：

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
  - port: 8765
    targetPort: 9376
  type: LoadBalancer
```

使用 kubectl

你也可以使用 kubectl expose 命令及其 --type=LoadBalancer 参数创建服务：

```
kubectl expose rc example --port=8765 --target-port=9376 \
--name=example-service --type=LoadBalancer
```

此命令通过使用与引用资源（在上面的示例的情况下，名为 example 的 replication controller）相同的选择器来创建一个新的服务。

更多信息（包括更多的可选参数），请参阅 [kubectl expose 指南](#)。

找到你的 IP 地址

你可以通过 kubectl 获取服务信息，找到为你的服务创建的 IP 地址：

```
kubectl describe services example-service
```

这将获得如下输出：

Name:	example-service
Namespace:	default
Labels:	<none>
Annotations:	<none>
Selector:	app=example
Type:	LoadBalancer
IP:	10.67.252.103

```
LoadBalancer Ingress: 192.0.2.89
Port: <unnamed> 80/TCP
NodePort: <unnamed> 32445/TCP
Endpoints: 10.64.0.4:80,10.64.1.5:80,10.64.2.4:80
Session Affinity: None
Events: <none>
```

IP 地址列在 LoadBalancer Ingress 旁边。

说明：

如果你在 Minikube 上运行服务，你可以通过以下命令找到分配的 IP 地址和端口：

```
minikube service example-service --url
```

保留客户端源 IP

由于此功能的实现，目标容器中看到的源 IP 将 不是客户端的原始源 IP。要启用保留客户端 IP，可以在服务的 spec 中配置以下字段（支持 GCE/Google Kubernetes Engine 环境）：

- `service.spec.externalTrafficPolicy` - 表示此服务是否希望将外部流量路由到节点本地或集群范围的端点。有两个可用选项：Cluster（默认）和 Local。Cluster 隐藏了客户端源 IP，可能导致第二跳到另一个节点，但具有良好的整体负载分布。Local 保留客户端源 IP 并避免 LoadBalancer 和 NodePort 类型服务的第二跳，但存在潜在的不均衡流量传播风险。
- `service.spec.healthCheckNodePort` - 指定服务的 healthcheck nodePort（数字端口号）。如果未指定 `healthCheckNodePort`，服务控制器从集群的 NodePort 范围内分配一个端口。你可以通过设置 API 服务器的命令行选项 `--service-node-port-range` 来配置上述范围。它将会使用用户指定的 `healthCheckNodePort` 值（如果被客户端指定）。仅当 `type` 设置为 `LoadBalancer` 并且 `externalTrafficPolicy` 设置为 `Local` 时才生效。

可以通过在服务的配置文件中将 `externalTrafficPolicy` 设置为 `Local` 来激活此功能。

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
```

`externalTrafficPolicy: Local`

`type: LoadBalancer`

回收负载均衡器

在通常情况下，应在删除 LoadBalancer 类型服务后立即清除云提供商中的相关负载均衡器资源。但是，众所周知，在删除关联的服务后，云资源被孤立的情况很多。引入了针对服务负载均衡器的终结器保护，以防止这种情况发生。通过使用终结器，在删除相关的负载均衡器资源之前，也不会删除服务资源。

具体来说，如果服务具有 type LoadBalancer，则服务控制器将附加一个名为 `service.kubernetes.io/load-balancer-cleanup` 的终结器。仅在清除负载均衡器资源后才能删除终结器。即使在诸如服务控制器崩溃之类的极端情况下，这也可以防止负载均衡器资源悬空。

外部负载均衡器提供商

请务必注意，此功能的数据路径由 Kubernetes 集群外部的负载均衡器提供。

当服务 type 设置为 LoadBalancer 时，Kubernetes 向集群中的 Pod 提供的功能等同于 type 等于 ClusterIP，并通过使用 Kubernetes pod 的条目对负载均衡器（从外部到 Kubernetes）进行编程来扩展它。Kubernetes 服务控制器自动创建外部负载均衡器、健康检查（如果需要）、防火墙规则（如果需要），并获取云提供商分配的外部 IP 并将其填充到服务对象中。

保留源 IP 时的注意事项和限制

GCE/AWS 负载均衡器不为其目标池提供权重。对于旧的 LB `kube-proxy` 规则来说，这不是一个问题，它可以在所有端点之间正确平衡。

使用新功能，外部流量不会在 pod 之间平均负载，而是在节点级别平均负载（因为 GCE/AWS 和其他外部 LB 实现无法指定每个节点的权重，因此它们的平衡跨所有目标节点，并忽略每个节点上的 Pod 数量）。

但是，我们可以声明，对于 `NumServicePods << NumNodes` 或 `NumServicePods >> NumNodes` 时，即使没有权重，也会看到接近相等的分布。

一旦外部负载均衡器提供权重，就可以将此功能添加到 LB 编程路径中。未来工作：1.4 版本不提供权重支持，但可能会在将来版本中添加

内部 Pod 到 Pod 的流量应该与 ClusterIP 服务类似，所有 Pod 的概率相同。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 16, 2020 at 11:05 AM PST: [\[zh\] Tidy up and fix links in tasks section \(8/10\) \(c2aae6890\)](#)

列出集群中所有运行容器的镜像

本文展示如何使用 kubectl 来列出集群中所有运行 Pod 的容器的镜像

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.

在本练习中，你将使用 kubectl 来获取集群中运行的所有 Pod，并格式化输出来提取每个 Pod 中的容器列表。

列出所有命名空间下的所有容器

- 使用 kubectl get pods --all-namespaces 获取所有命名空间下的所有 Pod
- 使用 -o jsonpath={..image} 来格式化输出，以仅包含容器镜像名称。这将以递归方式从返回的 json 中解析出 image 字段。
 - 参阅 [jsonpath 说明](#) 获得更多关于如何使用 jsonpath 的信息。
- 使用标准化工具来格式化输出：tr, sort, uniq
 - 使用 tr 以用换行符替换空格
 - 使用 sort 来对结果进行排序
 - 使用 uniq 来聚合镜像计数

```
kubectl get pods --all-namespaces -o jsonpath="{..image}" |  
tr -s '[:space:]' '\n' |  
sort |  
uniq -c
```

上面的命令将递归获取所有返回项目的名为 image 的字段。

作为替代方案，可以使用 Pod 的镜像字段的绝对路径。这确保即使字段名称重复的情况下也能检索到正确的字段，例如，特定项目中的许多字段都称为 name：

```
kubectl get pods --all-namespaces -o jsonpath=".items[*].spec.containers[*].image"
```

jsonpath 解释如下：

- `.items[*]`: 对于每个返回的值
- `.spec`: 获取 spec
- `.containers[*]`: 对于每个容器
- `.image`: 获取镜像

说明：按名字获取单个 Pod 时，例如 `kubectl get pod nginx`，路径的 `.items[*]` 部分应该省略，因为返回的是一个 Pod 而不是一个项目列表。

列出 Pod 中的容器

可以使用 range 操作进一步控制格式化，以单独操作每个元素。

```
kubectl get pods --all-namespaces -o=jsonpath='{range .items[*]}{"\n"}{.metadata.name}":\t{range .spec.containers[*]}{.image}{", "}{end}{end}' | sort
```

列出以标签过滤后的 Pod 的所有容器

要获取匹配特定标签的 Pod，请使用 `-l` 参数。以下匹配仅与标签 `app=nginx` 相符的 Pod。

```
kubectl get pods --all-namespaces -o=jsonpath=".image" -l app=nginx
```

列出以命名空间过滤后的 Pod 的所有容器

要获取匹配特定命名空间的 Pod，请使用 `namespace` 参数。以下仅匹配 `kube-system` 命名空间下的 Pod。

```
kubectl get pods --namespace kube-system -o jsonpath=".image"
```

使用 go-template 替换 jsonpath 来获取容器

作为 jsonpath 的替代，Kubectl 支持使用 [go-templates](#) 来格式化输出：

```
kubectl get pods --all-namespaces -o go-template --template="{{range .items}}{{range .spec.containers}}{{.image}} {{end}}{{end}}"
```

接下来

参考

- [Jsonpath 参考指南](#)
- [Go template 参考指南](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 16, 2020 at 11:05 AM PST: [\[zh\] Tidy up and fix links in tasks section \(8/10\) \(c2aae6890\)](#)

在 Minikube 环境中使用 NGINX Ingress 控制器配置 Ingress

[Ingress](#)是一种 API 对象，其中定义了一些规则使得集群中的 服务可以从集群外访问。[Ingress 控制器](#) 负责满足 Ingress 中所设置的规则。

本节为你展示如何配置一个简单的 Ingress，根据 HTTP URI 将服务请求路由到 服务 web 或 web2。

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.

创建一个 Minikube 集群

1. 点击 Launch Terminal

最后修改 November 20, 2020 at 4:42 PM PST: [\[zh\] Sync English site changes \(9\) \(51949a940\)](#)

为集群配置 DNS

Kubernetes 提供 DNS 集群插件，大多数支持的环境默认情况下都会启用。在 Kubernetes 1.11 及其以后版本中，推荐使用 CoreDNS，kubeadm 默认会安装 CoreDNS。

要了解关于如何为 Kubernetes 集群配置 CoreDNS 的更多信息，参阅 [定制 DNS 服务](#)。关于如何利用 kube-dns 配置 kubernetes DNS 的演示例子，参阅 [Kubernetes DNS 插件示例](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 07, 2020 at 10:44 PM PST: [Update configure-dns-cluster.md \(12d2845dc\)](#)

同 Pod 内的容器使用共享卷通信

本文旨在说明如何让一个 Pod 内的两个容器使用一个卷 (Volume) 进行通信。参阅如何让两个进程跨容器通过 [共享进程名字空间](#)。

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.

创建一个包含两个容器的 Pod

在这个练习中，你会创建一个包含两个容器的 Pod。两个容器共享一个卷用于他们之间的通信。Pod 的配置文件如下：

[pods/two-container-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
```

```
volumes:
- name: shared-data
  emptyDir: {}

containers:

- name: nginx-container
  image: nginx
  volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html

- name: debian-container
  image: debian
  volumeMounts:
    - name: shared-data
      mountPath: /pod-data
  command: ["/bin/sh"]
  args: ["-c", "echo Hello from the debian container > /pod-data/index.html"]
```

在配置文件中，你可以看到 Pod 有一个共享卷，名为 shared-data。

配置文件中的第一个容器运行了一个 nginx 服务器。共享卷的挂载路径是 /usr/share/nginx/html。第二个容器是基于 debian 镜像的，有一个 /pod-data 的挂载路径。第二个容器运行了下面的命令然后终止。

```
echo Hello from the debian container > /pod-data/index.html
```

注意，第二个容器在 nginx 服务器的根目录下写了 index.html 文件。

创建一个包含两个容器的 Pod：

```
kubectl apply -f https://k8s.io/examples/pods/two-container-pod.yaml
```

查看 Pod 和容器的信息：

```
kubectl get pod two-containers --output=yaml
```

这是输出的一部分：

```
apiVersion: v1
kind: Pod
metadata:
  ...
name: two-containers
namespace: default
```

```
...
spec:
...
  containerStatuses:
    - containerID: docker://c1d8abd1 ...
      image: debian
      ...
      lastState:
        terminated:
          ...
          name: debian-container
          ...
    - containerID: docker://96c1ff2c5bb ...
      image: nginx
      ...
      name: nginx-container
      ...
      state:
        running:
          ...

```

你可以看到 debian 容器已经被终止了，而 nginx 服务器依然在运行。

进入 nginx 容器的 shell：

```
kubectl exec -it two-containers -c nginx-container -- /bin/bash
```

在 shell 中，确认 nginx 还在运行。

```
root@two-containers:/# ps aux
```

输出类似于这样：

```
USER      PID ... STAT START   TIME COMMAND
root      1 ... Ss  21:12 0:00 nginx: master process nginx -g daemon off;
```

回忆一下，debian 容器在 nginx 的根目录下创建了 index.html 文件。使用 curl 向 nginx 服务器发送一个 GET 请求：

```
root@two-containers:/# curl localhost
```

输出表示 nginx 提供了 debian 容器写的页面：

```
Hello from the debian container
```

讨论

Pod 能有多个容器的主要原因是支持辅助应用（ helper applications ），以协助主应用（ primary application ）。辅助应用的典型例子是数据抽取，数据推送和代理。辅助应用和主应用经常需要相互通信。就如这个练习所示，通信通常是通过共享文件系统完成的，或者，也通过回环网络接口 localhost 完成。举个网络接口的例子， web 服务器带有一个协助程序用于拉取 Git 仓库的更新。

在本练习中的卷为 Pod 生命周期中的容器相互通信提供了一种方法。如果 Pod 被删除或者重建了，任何共享卷中的数据都会丢失。

接下来

- 进一步了解[复合容器的模式](#)
- 学习[模块化架构中的复合容器](#)
- 参见[配置 Pod 使用卷来存储数据](#)
- 参考[在 Pod 中的容器之间共享进程命名空间](#)
- 参考[Volume](#)
- 参考[Pod](#)

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 15, 2020 at 3:12 PM PST: [Update communicate-containers-same-pod-shared-volume.md \(68e1ee3be\)](#)

配置对多集群的访问

本文展示如何使用配置文件来配置对多个集群的访问。在将集群、用户和上下文定义在一个或多个配置文件中之后，用户可以使用 `kubectl config use-context` 命令快速地在集群之间进行切换。

说明：用于配置集群访问的文件有时被称为 *kubeconfig* 文件。这是一种引用配置文件的通用方式，并不意味着存在一个名为 *kubeconfig* 的文件。

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](#)

要检查 [kubectl](#) 是否安装，执行 `kubectl version --client` 命令。 `kubectl` 的版本应该与集群的 API 服务器 [使用同一次版本号](#)。

定义集群、用户和上下文

假设用户有两个集群，一个用于正式开发工作，一个用于其它临时用途（ scratch ）。在 development 集群中，前端开发者在名为 frontend 的名字空间下工作，存储开发者在名为 storage 的名字空间下工作。在 scratch 集群中，开发人员可能在默认名字空间下工作，也可能视情况创建附加的名字空间。访问开发集群需要通过证书进行认证。访问其它临时用途的集群需要通过用户名和密码进行认证。

创建名为 config-exercise 的目录。在 config-exercise 目录中，创建名为 config -demo 的文件，其内容为：

```
apiVersion: v1
kind: Config
preferences: {}

clusters:
- cluster:
  name: development
- cluster:
  name: scratch

users:
- name: developer
- name: experimenter

contexts:
- context:
  name: dev-frontend
- context:
  name: dev-storage
- context:
  name: exp-scratch
```

配置文件描述了集群、用户名和上下文。 config-demo 文件中含有描述两个集群、两个用户和三个上下文的框架。

进入 config-exercise 目录。 输入以下命令，将群集详细信息添加到配置文件中：

```
kubectl config --kubeconfig=config-demo set-cluster development --  
server=https://1.2.3.4 --certificate-authority=fake-ca-file  
kubectl config --kubeconfig=config-demo set-cluster scratch --server=https  
://5.6.7.8 --insecure-skip-tls-verify
```

将用户详细信息添加到配置文件中：

```
kubectl config --kubeconfig=config-demo set-credentials developer --  
client-certificate=fake-cert-file --client-key=fake-key-seefile  
kubectl config --kubeconfig=config-demo set-credentials experimenter --  
username=exp --password=some-password
```

注意：

- 要删除用户，可以运行 `kubectl --kubeconfig=config-demo config unset users.<name>`
- 要删除集群，可以运行 `kubectl --kubeconfig=config-demo config unset clusters.<name>`
- 要删除上下文，可以运行 `kubectl --kubeconfig=config-demo config unset contexts.<name>`

将上下文详细信息添加到配置文件中：

```
kubectl config --kubeconfig=config-demo set-context dev-frontend --  
cluster=development --namespace=frontend --user=developer  
kubectl config --kubeconfig=config-demo set-context dev-storage --  
cluster=development --namespace=storage --user=developer  
kubectl config --kubeconfig=config-demo set-context exp-scratch --  
cluster=scratch --namespace=default --user=experimenter
```

打开 config-demo 文件查看添加的详细信息。 也可以使用 config view 命令进行查看：

```
kubectl config --kubeconfig=config-demo view
```

输出展示了两个集群、两个用户和三个上下文：

```
apiVersion: v1  
clusters:  
- cluster:  
  certificate-authority: fake-ca-file  
  server: https://1.2.3.4  
  name: development  
- cluster:
```

```
insecure-skip-tls-verify: true
server: https://5.6.7.8
name: scratch
contexts:
- context
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
- context
  cluster: development
  namespace: storage
  user: developer
  name: dev-storage
- context
  cluster: scratch
  namespace: default
  user: experimenter
  name: exp-scratch
current-context: ""
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    password: some-password
    username: exp
```

其中的 fake-ca-file、fake-cert-file 和 fake-key-file 是证书文件路径名的占位符。你需要更改这些值，使之对应你的环境中证书文件的实际路径名。

有时你可能希望在这里使用 BASE64 编码的数据而不是一个个独立的证书文件。如果是这样，你需要在键名上添加 -data 后缀。例如，certificate-authority-data、client-certificate-data 和 client-key-data。

每个上下文包含三部分（集群、用户和名字空间），例如，dev-frontend 上下文表明：使用 developer 用户的凭证来访问 development 集群的 frontend 名字空间。

设置当前上下文：

```
kubectl config --kubeconfig=config-demo use-context dev-frontend
```

现在当输入 kubectl 命令时，相应动作会应用于 dev-frontend 上下文中所列的集群和名字空间，同时，命令会使用 dev-frontend 上下文中所列用户的凭证。

使用 --minify 参数，来查看与当前上下文相关联的配置信息。

```
kubectl config --kubeconfig=config-demo view --minify
```

输出结果展示了 dev-frontend 上下文相关的配置信息：

```
apiVersion: v1
clusters:
- cluster:
  certificate-authority: fake-ca-file
  server: https://1.2.3.4
  name: development
contexts:
- context:
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```

现在假设用户希望在其它临时用途集群中工作一段时间。

将当前上下文更改为 exp-scratch：

```
kubectl config --kubeconfig=config-demo use-context exp-scratch
```

现在你发出的所有 kubectl 命令都将应用于 scratch 集群的默认名字空间。同时，命令会使用 exp-scratch 上下文中所列用户的凭证。

查看更新后的当前上下文 exp-scratch 相关的配置：

```
kubectl config --kubeconfig=config-demo view --minify
```

最后，假设用户希望在 development 集群中的 storage 名字空间下工作一段时间。

将当前上下文更改为 dev-storage：

```
kubectl config --kubeconfig=config-demo use-context dev-storage
```

查看更新后的当前上下文 dev-storage 相关的配置：

```
kubectl config --kubeconfig=config-demo view --minify
```

创建第二个配置文件

在 config-exercise 目录中，创建名为 config-demo-2 的文件，其中包含以下内容：

```
apiVersion: v1
kind: Config
preferences: {}

contexts:
- context
  cluster: development
  namespace: ramp
  user: developer
  name: dev-ramp-up
```

上述配置文件定义了一个新的上下文，名为 dev-ramp-up。

设置 KUBECONFIG 环境变量

查看是否有名为 KUBECONFIG 的环境变量。如有，保存 KUBECONFIG 环境变量当前的值，以便稍后恢复。例如：

Linux

```
export KUBECONFIG_SAVED=$KUBECONFIG
```

Windows PowerShell

```
$Env:KUBECONFIG_SAVED=$Env:KUBECONFIG
```

KUBECONFIG 环境变量是配置文件路径的列表，该列表在 Linux 和 Mac 中以冒号分隔，在 Windows 中以分号分隔。如果有 KUBECONFIG 环境变量，请熟悉列表中的配置文件。

临时添加两条路径到 KUBECONFIG 环境变量中。例如：

Linux

```
export KUBECONFIG=$KUBECONFIG:config-demo:config-demo-2
```

Windows PowerShell

```
$Env:KUBECONFIG=("config-demo;config-demo-2")
```

在 config-exercise 目录中输入以下命令：

```
kubectl config view
```

输出展示了 KUBECONFIG 环境变量中所列举的所有文件合并后的信息。特别地，注意合并信息中包含来自 config-demo-2 文件的 dev-ramp-up 上下文和来自 config-demo 文件的三个上下文：

```
contexts:
- context
  cluster: development
  namespace: frontend
  user: developer
  name: dev-frontend
- context
  cluster: development
  namespace: ramp
  user: developer
  name: dev-ramp-up
- context
  cluster: development
  namespace: storage
  user: developer
  name: dev-storage
- context
  cluster: scratch
  namespace: default
  user: experimenter
  name: exp-scratch
```

关于 kubeconfig 文件如何合并的更多信息，请参考 [使用 kubeconfig 文件组织集群访问](#)

探索 \$HOME/.kube 目录

如果用户已经拥有一个集群，可以使用 kubectl 与集群进行交互，那么很可能在 \$HOME/.kube 目录下有一个名为 config 的文件。

进入 \$HOME/.kube 目录，看看那里有什么文件。通常会有一个名为 config 的文件，目录中可能还有其他配置文件。请简单地熟悉这些文件的内容。

将 \$HOME/.kube/config 追加到 KUBECONFIG 环境变量中

如果有 \$HOME/.kube/config 文件，并且还未列在 KUBECONFIG 环境变量中，那么现在将它追加到 KUBECONFIG 环境变量中。例如：

Linux

```
export KUBECONFIG=$KUBECONFIG:$HOME/.kube/config
```

Windows Powershell

```
$Env:KUBECONFIG="$Env:KUBECONFIG;$HOME\.kube\config"
```

在配置练习目录中输入以下命令，查看当前 KUBECONFIG 环境变量中列举的所有文件合并后的配置信息：

```
kubectl config view
```

清理

将 KUBECONFIG 环境变量还原为原始值。例如：

Linux

```
export KUBECONFIG=$KUBECONFIG_SAVED
```

Windows PowerShell

```
$Env:KUBECONFIG=$Env:KUBECONFIG_SAVED
```

接下来

- [使用 kubeconfig 文件组织集群访问](#)
- [kubectl config](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

监控、日志和排错

设置监视和日志记录以对集群进行故障排除或调试容器化应用。

[StackDriver 中的事件](#)

[使用 crictl 对 Kubernetes 节点进行调试](#)

[使用 ElasticSearch 和 Kibana 进行日志管理](#)

[使用 Stackdriver 生成日志](#)

[在本地开发和调试服务](#)

[审计](#)

[应用故障排查](#)

[应用自测与调试](#)

[故障诊断](#)

[确定 Pod 失败的原因](#)

[节点健康监测](#)

[获取正在运行容器的 Shell](#)

[调试 Init 容器](#)

[调试 Pods 和 ReplicationControllers](#)

[调试 Service](#)

[调试StatefulSet](#)

[调试运行中的 Pod](#)

[资源指标管道](#)

[资源监控工具](#)

[集群故障排查](#)

StackDriver 中的事件

Kubernetes 事件是一种对象，它为用户提供了洞察集群内发生的事情的能力，例如调度程序做出了什么决定，或者为什么某些 Pod 被逐出节点。你可以在[应用程序自检和调试](#)中阅读有关使用事件调试应用程序的更多信息。

因为事件是 API 对象，所以它们存储在主控节点上的 API 服务器中。为了避免主节点磁盘空间被填满，将强制执行保留策略：事件在最后一次发生的一小时后将会被删除。为了提供更长的历史记录和聚合能力，应该安装第三方解决方案来捕获事件。

本文描述了一个将 Kubernetes 事件导出为 Stackdriver Logging 的解决方案，在这里可以对它们进行处理和分析。

说明：不能保证集群中发生的所有事件都将导出到 Stackdriver。事件不能导出的一种可能情况是事件导出器没有运行（例如，在重新启动或升级期间）。在大多数情况下，可以将事件用于设置 [metrics](#) 和 [alerts](#) 等目的，但你应该注意其潜在的不准确性。

部署

Google Kubernetes Engine

在 Google Kubernetes Engine 中，如果启用了云日志，那么事件导出器默认部署在主节点运行版本为 1.7 及更高版本的集群中。为了防止干扰你的工作负载，事件导出器没有设置资源，并且处于尽力而为的 QoS 类型中，这意味着它将在资源匮乏的情况下第一个被杀死。如果要导出事件，请确保有足够的资源给事件导出器 Pod 使用。这可能会因为工作负载的不同而有所不同，但平均而言，需要大约 100MB 的内存和 100m 的 CPU。

部署到现有集群

使用下面的命令将事件导出器部署到你的集群：

```
kubectl create -f https://k8s.io/examples/debug/event-exporter.yaml
```

由于事件导出器访问 Kubernetes API，因此它需要权限才能访问。以下的部署配置为使用 RBAC 授权。它设置服务帐户和集群角色绑定，以允许事件导出器读取事件。为了确保事件导出器 Pod 不会从节点中退出，你可以另外设置资源请求。如前所述，100MB 内存和 100m CPU 应该就足够了。

[debug/event-exporter.yaml](#)



```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: event-exporter-sa
```

```
namespace: default
labels:
  app: event-exporter
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: event-exporter-rb
  labels:
    app: event-exporter
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- kind: ServiceAccount
  name: event-exporter-sa
  namespace: default
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: event-exporter-v0.2.3
  namespace: default
  labels:
    app: event-exporter
spec:
  selector:
    matchLabels:
      app: event-exporter
  replicas: 1
  template:
    metadata:
      labels:
        app: event-exporter
    spec:
      serviceAccountName: event-exporter-sa
      containers:
        - name: event-exporter
          image: k8s.gcr.io/event-exporter:v0.2.3
          command:
            - '/event-exporter'
      terminationGracePeriodSeconds: 30
```

用户指南

事件在 Stackdriver Logging 中被导出到 GKE Cluster 资源。 你可以通过从可用资源的下拉菜单中选择适当的选项来找到它们：

The screenshot shows the Stackdriver Logging interface. At the top, there is a search bar labeled "Filter by label or text search". Below it are three dropdown menus: "GKE Cluster" (set to "GKE Cluster"), "All logs" (set to "All logs"), and "Any log level" (set to "Any log level"). A list of resources is displayed on the left, with "GKE Cluster" checked. The right side shows the logs for the selected resource. One log entry is highlighted: "19:38:50.000 Started container". A yellow banner at the bottom right states "No newer entries found matching current filter".

你可以使用 Stackdriver Logging 的 [过滤机制](#) 基于事件对象字段进行过滤。 例如，下面的查询将显示调度程序中有关 Deployment nginx-deployment 中的 Pod 的事件：

```
resource.type="gke_cluster"
jsonPayload.kind="Event"
jsonPayload.source.component="default-scheduler"
jsonPayload.involvedObject.name:"nginx-deployment"
```

```
1 resource.type="gke_cluster"
2 jsonPayload.kind="Event"
3 jsonPayload.source.component="default-scheduler"
4 jsonPayload.involvedObject.name:"nginx-deployment"
```

Submit Filter Jump to date ▾

2017-06-21 CEST

↓

▶ i 19:38:41.000 Successfully assigned nginx-deployment-3088474477-9bbgf

▶ i 19:38:41.000 Successfully assigned nginx-deployment-3088474477-w3hzb

↑

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 10, 2020 at 11:40 AM PST: [\[zh\] Tidy up and fix links in tasks section \(2/10\) \(128182188\)](#)

使用 crictl 对 Kubernetes 节点进行调试

FEATURE STATE: Kubernetes v1.11 [stable]

crictl 是 CRI 兼容的容器运行时命令行接口。你可以使用它来检查和调试 Kubernetes 节点上的容器运行时和应用程序。crictl 和它的源代码在 [cri-tools](#) 代码库。

准备开始

crictl 需要带有 CRI 运行时的 Linux 操作系统。

安装 crictl

你可以从 cri-tools [发布页面](#) 下载一个压缩的 crictl 归档文件，用于几种不同的架构。下载与你的 kubernetes 版本相对应的版本。提取它并将其移动到系统路径上的某个位置，例如 /usr/local/bin/。

一般用法

cricl 命令有几个子命令和运行时参数。有关详细信息，请使用 crictl help 或 crictl <subcommand> help 获取帮助信息。

cricl 默认连接到 unix:///var/run/dockershim.sock。对于其他的运行时，你可以用多种不同的方法设置端点：

- 通过设置参数 --runtime-endpoint 和 --image-endpoint
- 通过设置环境变量 CONTAINER_RUNTIME_ENDPOINT 和 IMAGE_SERVICE_ENDPOINT
- 通过在配置文件中设置端点 --config=/etc/crictl.yaml

你还可以在连接到服务器并启用或禁用调试时指定超时值，方法是在配置文件中指定 timeout 或 debug 值，或者使用 --timeout 和 --debug 命令行参数。

要查看或编辑当前配置，请查看或编辑 /etc/crictl.yaml 的内容。

```
cat /etc/crictl.yaml
```

```
runtime-endpoint: unix:///var/run/dockershim.sock
image-endpoint: unix:///var/run/dockershim.sock
timeout: 10
debug: true
```

crictl 命令示例

警告：

如果使用 crictl 在正在运行的 Kubernetes 集群上创建 Pod 沙盒或容器，kubelet 最终将删除它们。crictl 不是一个通用的工作流工具，而是一个对调试有用的工具。

打印 Pod 清单

打印所有 Pod 的清单：

```
crictl pods
```

POD ID NAMESPACE	CREATED ATTEMPT	STATE	NAME
926f1b5a1d33a sh-84d7dcf559-4r2gq	About a minute ago default	Ready 0	
4dccb216c4adb wv2gp	About a minute ago default	Ready 0	nginx-65899c769f-
a86316e96fa89 gblk4	17 hours ago kube-system	Ready 0	kube-proxy-

```
919630b8f81f1 17 hours ago Ready nvidia-device-plugin-zgbbv kube-system 0
```

根据名称打印 Pod 清单：

```
crlctl pods --name nginx-65899c769f-wv2gp
```

POD ID	CREATED	STATE	NAME
NAMESPACE	ATTEMPT		
4dcc216c4adb default	2 minutes ago 0	Ready	nginx-65899c769f-wv2gp

根据标签打印 Pod 清单：

```
crlctl pods --label run=nginx
```

POD ID	CREATED	STATE	NAME
NAMESPACE	ATTEMPT		
4dcc216c4adb default	2 minutes ago 0	Ready	nginx-65899c769f-wv2gp

打印镜像清单

打印所有镜像清单：

```
crlctl images
```

IMAGE	TAG	IMAGE ID	SIZE
busybox	latest	8c811b4aec35f	1.15MB
k8s-gcrio.azureedge.net/hyperkube-amd64	v1.10.3		
e179bbfe5d238	665MB		
k8s-gcrio.azureedge.net/pause-amd64	3.1	da86e6ba6ca19	
742kB			
nginx	latest	cd5239a0906a6	109MB

根据仓库打印镜像清单：

```
crlctl images nginx
```

IMAGE	TAG	IMAGE ID	SIZE
nginx	latest	cd5239a0906a6	109MB

只打印镜像 ID：

```
crlctl images -q
```

```
sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed93  
6a472a  
sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b29
```

```
70c5  
sha256:da86e6ba6ca197bf6bc5e9d900feb906b133eaa4750e6bed647b0fbe  
50ed43e  
sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57  
553569
```

打印容器清单

打印所有容器清单：

```
crlctl ps -a
```

CONTAINER ID	IMAGE	CREATED	STATE	NAME	ATTEMPT
1f73f2d81bf98	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f	41046e0f37d47		7 minutes ago	Running
sh	9c5951df22c78		1		
nginx	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f	41046e0f37d47		8 minutes ago	Exited
sh	87d3992f84f74		0		
nginx	nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c	55d8ac139d5f		8 minutes ago	Running
nginx	1941fb4da154f	k8s-gcrio.azureedge.net/hyperkube-			
amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd7					
14b30a714260a	18 hours ago		Running		kube-proxy
					0

打印正在运行的容器清单：

```
crlctl ps
```

CONTAINER ID	IMAGE	CREATED	STATE	NAME	ATTEMPT
1f73f2d81bf98	busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f	41046e0f37d47		6 minutes ago	Running
sh	87d3992f84f74		1		
nginx	nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c	55d8ac139d5f		7 minutes ago	Running
nginx	1941fb4da154f	k8s-gcrio.azureedge.net/hyperkube-			

```
amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd7  
14b30a714260a 17 hours ago      Running      kube-proxy      0
```

在正在运行的容器上执行命令

```
cricl exec -i -t 1f73f2d81bf98 ls
```

```
bin dev etc home proc root sys tmp usr var
```

获取容器日志

获取容器的所有日志：

```
cricl logs 87d3992f84f74
```

```
10.240.0.96 -- [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 "-"  
"curl/7.47.0" "-"  
10.240.0.96 -- [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 "-"  
"curl/7.47.0" "-"  
10.240.0.96 -- [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-"  
"curl/7.47.0" "-"
```

获取最近的 N 行日志：

```
cricl logs --tail=1 87d3992f84f74
```

```
10.240.0.96 -- [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-"  
"curl/7.47.0" "-"
```

运行 Pod 沙盒

用 crictl 运行 Pod 沙盒对容器运行时排错很有帮助。在运行的 Kubernetes 集群中，沙盒会随机地被 kubelet 停止和删除。

1. 编写下面的 JSON 文件：

```
{  
  "metadata": {  
    "name": "nginx-sandbox",  
    "namespace": "default",  
    "attempt": 1,  
    "uid": "hdishd83djaidwnduwk28bcsb"  
  },  
  "logDirectory": "/tmp",  
  "linux": {}  
}
```

2. 使用 crictl runp 命令应用 JSON 文件并运行沙盒。

```
crictl runp pod-config.json
```

返回了沙盒的 ID。

创建容器

用 crictl 创建容器对容器运行时排错很有帮助。 在运行的 Kubernetes 集群中，沙盒会随机的被 kubelet 停止和删除。

1. 拉取 busybox 镜像

```
crictl pull busybox
Image is up to date for
busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416
dea4f41046e0f37d47
```

2. 创建 Pod 和容器的配置：

Pod 配置：

```
{
  "metadata": {
    "name": "nginx-sandbox",
    "namespace": "default",
    "attempt": 1,
    "uid": "hdishd83djaidwnduwk28bcsb"
  },
  "log_directory": "/tmp",
  "linux": {}
}
```

容器配置：

```
{
  "metadata": {
    "name": "busybox"
  },
  "image": {
    "image": "busybox"
  },
  "command": [
    "top"
  ],
  "log_path": "busybox.log",
  "linux": {
```

```
}
```

3. 创建容器，传递先前创建的 Pod 的 ID、容器配置文件和 Pod 配置文件。返回容器的 ID。

```
cricctl create  
f84dd361f8dc51518ed291fbadd6db537b0496536c1d2d6c05ff943ce8c9  
a54f container-config.json pod-config.json
```

4. 查询所有容器并确认新创建的容器状态为 Created。

```
cricctl ps -a
```

CONTAINER ID	IMAGE	CREATED	STATE
NAME	ATTEMPT		
3e025dd50a72d	busybox	32 seconds ago	
Created	busybox	0	

启动容器

要启动容器，要将容器 ID 传给 cricctl start：

```
cricctl start  
3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60  
3e025dd50a72d956c4f14881fbb5b1080c9275674e95fb67f965f6478a957d60
```

确认容器的状态为 Running。

```
cricctl ps
```

CONTAINER ID	IMAGE	CREATED	STATE
NAME	ATTEMPT		
3e025dd50a72d	busybox	About a minute ago	Running
busybox	0		

更多信息请参考 [kubernetes-sigs/cri-tools](#)。

反馈

此页是否对您有帮助？

是 否

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

使用 ElasticSearch 和 Kibana 进行日志管理

在 Google Compute Engine (GCE) 平台上，默认的日志管理支持目标是 [Stackdriver Logging](#)，在[使用 Stackdriver Logging 管理日志](#) 中详细描述了这一点。

本文介绍了如何设置一个集群，将日志导入 [Elasticsearch](#)，并使用 [Kibana](#) 查看日志，作为在 GCE 上 运行应用时使用 Stackdriver Logging 管理日志的替代方案。

说明：你不能在 Google Kubernetes Engine 平台运行的 Kubernetes 集群上自动部署 Elasticsearch 和 Kibana。你必须手动部署它们。

要使用 Elasticsearch 和 Kibana 处理集群日志，你应该在使用 kube-up.sh 脚本创建集群时设置下面所示的环境变量：

```
KUBE_LOGGING_DESTINATION=elasticsearch
```

你还应该确保设置了 KUBE_ENABLE_NODE_LOGGING=true（这是 GCE 平台的默认设置）。

现在，当你创建集群时，将有一条消息将指示每个节点上运行的 fluentd 日志收集守护进程以 ElasticSearch 为日志输出目标：

```
cluster/kube-up.sh
```

```
...
Project: kubernetes-satnam
Zone: us-central1-b
... calling kube-up
Project: kubernetes-satnam
Zone: us-central1-b
+++ Staging server tars to Google Storage: gs://kubernetes-staging-e6d0e81793/devel
+++ kubernetes-server-linux-amd64.tar.gz uploaded (sha1 = 6987c098277871b6d69623141276924ab687f89d)
+++ kubernetes-salt.tar.gz uploaded (sha1 = bdfc83ed6b60fa9e3bff9004b542cf643464cd0)
Looking for already existing resources
Starting master and configuring firewalls
Created [https://www.googleapis.com/compute/v1/projects/kubernetes-satnam/zones/us-central1-b/disks/kubernetes-master-pd].
```

NAME	ZONE	SIZE_GB	TYPE	STATUS
kubernetes-master-pd	us-central1-b	20	pd-ssd	READY
Created [https://www.googleapis.com/compute/v1/projects/kubernetes-satnam/regions/us-central1/addresses/kubernetes-master-ip].				
+++ Logging using Fluentd to elasticsearch				

每个节点的 Fluentd Pod、Elasticsearch Pod 和 Kibana Pod 都应该在集群启动后不久运行在 kube-system 名字空间中。

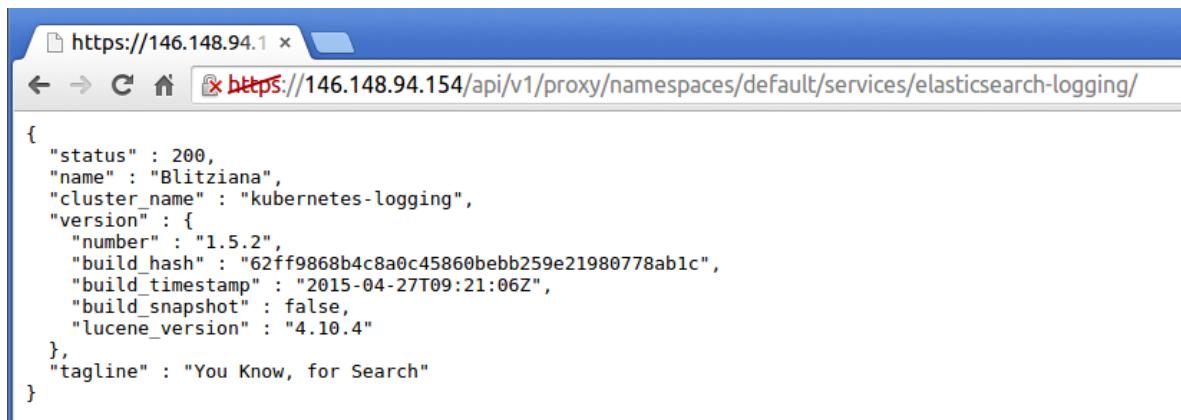
```
kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
elasticsearch-logging-v1-78nog	1/1	Running	0	2h
elasticsearch-logging-v1-nj2nb	1/1	Running	0	2h
fluentd-elasticsearch-kubernetes-node-5oq0	1/1	Running	0	2h
fluentd-elasticsearch-kubernetes-node-6896	1/1	Running	0	2h
fluentd-elasticsearch-kubernetes-node-l1ds	1/1	Running	0	2h
fluentd-elasticsearch-kubernetes-node-lz9j	1/1	Running	0	2h
kibana-logging-v1-bhpo8	1/1	Running	0	2h
kube-dns-v3-7r1l9	3/3	Running	0	2h
monitoring-heapster-v4-yl332	1/1	Running	1	2h
monitoring-influx-grafana-v1-o79xf	2/2	Running	0	2h

fluentd-elasticsearch Pod 从每个节点收集日志并将其发送到 elasticsearch-logging Pod，该 Pod 是名为 elasticsearch-logging 的 [服务](#)的一部分。这些 ElasticSearch pod 存储日志，并通过 REST API 将其公开。kibana-logging pod 提供了一个用于读取 ElasticSearch 中存储的日志的 Web UI，它是名为 kibana-logging 的服务的一部分。

Elasticsearch 和 Kibana 服务都位于 kube-system 名字空间中，并且没有通过可公开访问的 IP 地址直接暴露。要访问它们，请参照 [访问集群中运行的服务](#) 的说明进行操作。

如果你想在浏览器中访问 elasticsearch-logging 服务，你将看到类似下面的状态页面：



```

https://146.148.94.154/api/v1/proxy/namespaces/default/services/elasticsearch-logging/_status/_source

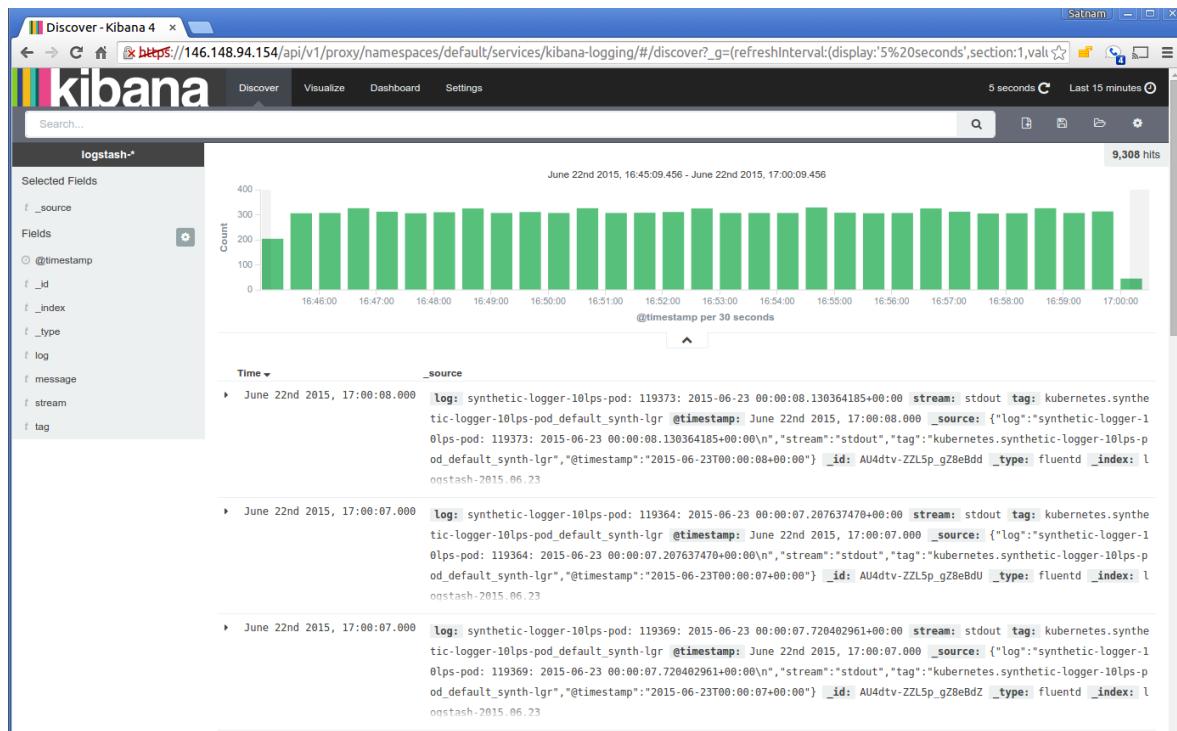
{
  "status" : 200,
  "name" : "Blitziana",
  "cluster_name" : "kubernetes-logging",
  "version" : {
    "number" : "1.5.2",
    "build_hash" : "62ff9868b4c8a0c45860bebb259e21980778ab1c",
    "build_timestamp" : "2015-04-27T09:21:06Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}

```

现在你可以直接在浏览器中输入 Elasticsearch 查询，如果你愿意的话。请参考[Elasticsearch 的文档](#)以了解这样做的更多细节。

或者，你可以使用 Kibana 查看集群的日志（再次使用[访问集群中运行的服务的说明](#)）。第一次访问 Kibana URL 时，将显示一个页面，要求你配置所接收日志的视图。选择时间序列值的选项，然后选择 @timestamp。在下面的页面中选择 Discover 选项卡，然后你应该能够看到所摄取的日志。你可以将刷新间隔设置为 5 秒，以便定期刷新日志。

以下是从 Kibana 查看器中摄取日志的典型视图：



接下来

Kibana 为浏览你的日志提供了各种强大的选项！有关如何深入研究它的一些想法，请查看[Kibana 的文档](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

使用 Stackdriver 生成日志

在阅读这篇文档之前，强烈建议你先熟悉一下 [Kubernetes 日志概况](#)

说明：默认情况下，Stackdriver 日志机制仅收集容器的标准输出和标准错误流。如果要收集你的应用程序写入一个文件（例如）的任何日志，请参见 Kubernetes 日志概述中的 [sidecar 方式](#)

部署

为了接收日志，你必须将 Stackdriver 日志代理部署到集群中的每个节点。此代理是一个已配置的 fluentd，其配置存在一个 ConfigMap 中，且实例使用 Kubernetes 的 DaemonSet 进行管理。ConfigMap 和 DaemonSet 的实际部署，取决于你的集群设置。

部署到一个新的集群

Google Kubernetes Engine

对于部署在 Google Kubernetes Engine 上的集群，Stackdriver 是默认的日志解决方案。Stackdriver 日志机制会默认部署到你的新集群上，除非你明确地不选择。

其他平台

为了将 Stackdriver 日志机制部署到你正在使用 kube-up.sh 创建的新集群上，执行如下操作：

1. 设置环境变量 KUBE_LOGGING_DESTINATION 为 gcp。
2. **如果不是跑在 GCE 上**，在 KUBE_NODE_LABELS 变量中包含 beta.kubernetes.io/fluentd-ds-ready=true。

集群启动后，每个节点都应该运行 Stackdriver 日志代理。DaemonSet 和 ConfigMap 作为附加组件进行配置。如果你不是使用 kube-up.sh，可以考虑不使用预先配置的日志方案启动集群，然后部署 Stackdriver 日志代理到正在运行的集群。

警告：除了 Google Kubernetes Engine，Stackdriver 日志守护进程在其他的平台有已知的问题。请自行承担风险。

部署到一个已知集群

1. 在每个节点上打标签（如果尚未存在）

Stackdriver 日志代理部署使用节点标签来确定应该将其分配到给哪些节点。引入这些标签是为了区分 Kubernetes 1.6 或更高版本的节点。如果集群是在配置了 Stackdriver 日志机制的情况下创建的，并且节点的版本为 1.5.X 或更低版本，则它将使用 fluentd 用作静态容器。节点最多只能有一

个 fluentd 实例，因此只能将标签打在未分配过 fluentd pod 的节点上。你可以通过运行 kubectl describe 来确保你的节点被正确标记，如下所示：

```
kubectl describe node $NODE_NAME
```

输出应类似于如下内容：

```
Name:      NODE_NAME
Role:
Labels:    beta.kubernetes.io/fluentd-ds-ready=true
...
```

确保输出内容包含 beta.kubernetes.io/fluentd-ds-ready=true 标签。如果不存在，则可以使用 kubectl label 命令添加，如下所示：

```
kubectl label node $NODE_NAME beta.kubernetes.io/fluentd-ds-ready=true
```

说明：如果节点发生故障并且必须重新创建，则必须将标签重新打在重新创建了的节点。为了让此操作更便捷，你可以在节点启动脚本中使用 Kubelet 的命令行参数给节点添加标签。

1. 通过运行以下命令，部署一个带有日志代理配置的 ConfigMap：

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-configmap.yaml
```

该命令在 default 命名空间中创建 ConfigMap。你可以在创建 ConfigMap 对象之前手动下载文件并进行更改。

1. 通过运行以下命令，部署日志代理的 DaemonSet：

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-ds.yaml
```

你也可以在使用前下载和编辑此文件。

验证日志代理部署

部署 Stackdriver DaemonSet 之后，你可以通过运行以下命令来查看日志代理的部署状态：

```
kubectl get ds --all-namespaces
```

如果你的集群中有 3 个节点，则输出应类似于如下：

NAMESPACE	NAME	DESIRED	CURRENT	READY	NODE-SELECTOR	AGE
...						
default	fluentd-gcp-v2.0	3	3	3	beta.kubernetes.io/	

```
fluentd-ds-ready=true 5m
```

```
...
```

要了解使用 Stackdriver 进行日志记录的工作方式，请考虑以下具有日志生成的 pod 定义 [counter-pod.yaml](#)：

[debug/counter-pod.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
      image: busybox
      args: [/bin/sh, -c,
              'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

这个 pod 定义里有一个容器，该容器运行一个 bash 脚本，脚本每秒写一次计数器的值和日期时间，并无限期地运行。让我们在默认命名空间中创建此 pod。

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

你可以观察到正在运行的 pod：

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
counter	1/1	Running	0	5m

在短时间内，你可以观察到 "pending" 的 pod 的状态，因为 kubelet 必须先下载容器镜像。当 pod 状态变为 Running 时，你可以使用 kubectl logs 命令查看此 counter pod 的输出。

```
kubectl logs counter
```

```
0: Mon Jan 1 00:00:00 UTC 2001
1: Mon Jan 1 00:00:01 UTC 2001
2: Mon Jan 1 00:00:02 UTC 2001
...
...
```

正如日志概览所述，此命令从容器日志文件中获取日志项。如果该容器被 Kubernetes 杀死然后重新启动，你仍然可以访问前一个容器的日志。但是，如果将 Pod 从节点中驱逐，则日志文件会丢失。让我们通过删除当前运行的 counter 容器来演示这一点：

```
kubectl delete pod counter
```

```
pod "counter" deleted
```

然后重建它：

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

```
pod/counter created
```

一段时间后，你可以再次从 counter pod 访问日志：

```
kubectl logs counter
```

```
0: Mon Jan 1 00:01:00 UTC 2001  
1: Mon Jan 1 00:01:01 UTC 2001  
2: Mon Jan 1 00:01:02 UTC 2001  
...
```

如预期的那样，日志中仅出现最近的日志记录。但是，对于实际应用程序，你可能希望能够访问所有容器的日志，特别是出于调试的目的。这就是先前启用的 Stackdriver 日志机制可以提供帮助的地方。

查看日志

Stackdriver 日志代理为每个日志项关联元数据，供你在后续的查询中只选择感兴趣的消息：例如，来自某个特定 Pod 的消息。

元数据最重要的部分是资源类型和日志名称。容器日志的资源类型为 `container`，在用户界面中名为 GKE Containers（即使 Kubernetes 集群不在 Google Kubernetes Engine 上）。日志名称是容器的名称，因此，如果你有一个包含两个容器的 pod，在 spec 中名称定义为 `container_1` 和 `container_2`，则它们的日志的名称分别为 `container_1` 和 `container_2`。

系统组件的资源类型为 `compute`，在接口中名为 GCE VM Instance。系统组件的日志名称是固定的。对于 Google Kubernetes Engine 节点，系统组件中的每个日志项都具有以下日志名称之一：

- docker
- kubelet
- kube-proxy

你可以在[Stackdriver 专用页面](#)上了解有关查看日志的更多信息。

查看日志的一种可能方法是使用 [Google Cloud SDK](#) 中的 [gcloud logging](#) 命令行接口。它使用 Stackdriver 日志机制的 [过滤语法](#)查询特定日志。例如，你可以运行以下命令：

```
gcloud beta logging read 'logName="projects/$YOUR_PROJECT_ID/logs/count"' --format json | jq '.[].textPayload'
```

```
...
"2: Mon Jan 1 00:01:02 UTC 2001\n"
"1: Mon Jan 1 00:01:01 UTC 2001\n"
"0: Mon Jan 1 00:01:00 UTC 2001\n"
...
"2: Mon Jan 1 00:00:02 UTC 2001\n"
"1: Mon Jan 1 00:00:01 UTC 2001\n"
"0: Mon Jan 1 00:00:00 UTC 2001\n"
```

如你所见，尽管 kubelet 已经删除了第一个容器的日志，日志中仍会包含 counter 容器第一次和第二次运行时输出的消息。

导出日志

你可以将日志导出到 [Google Cloud Storage](#) 或 [BigQuery](#) 进行进一步的分析。Stackdriver 日志机制提供了接收器（Sink）的概念，你可以在其中指定日志项的存放地。可在 Stackdriver [导出日志页面](#) 上获得更多信息。

配置 Stackdriver 日志代理

有时默认的 Stackdriver 日志机制安装可能无法满足你的需求，例如：

- 你可能需要添加更多资源，因为默认的行为表现无法满足你的需求。
- 你可能需要引入额外的解析机制以便从日志消息中提取更多元数据，例如严重性或源代码引用。
- 你可能想要将日志不仅仅发送到 Stackdriver 或仅将部分日志发送到 Stackdriver。

在这种情况下，你需要更改 DaemonSet 和 ConfigMap 的参数。

先决条件

如果使用的是 GKE，并且集群中启用了 Stackdriver 日志机制，则无法更改其配置，因为它是由 GKE 管理和支持的。但是，你可以禁用默认集成的日志机制并部署自己的。

说明：你将需要自己支持和维护新部署的配置了：更新映像和配置、调整资源等等。

若要禁用默认的日志记录集成，请使用以下命令：

```
gcloud beta container clusters update --logging-service=none CLUSTER
```

你可以在[部署部分](#)中找到有关如何将 Stackdriver 日志代理安装到正在运行的集群中的说明。

更改 DaemonSet 参数

当集群中有 Stackdriver 日志机制的 DaemonSet 时，你只需修改其 spec 中的 template 字段，daemonset 控制器将为你更新 Pod。例如，假设你按照上面的描述已经安装了 Stackdriver 日志机制。现在，你想更改内存限制，来给 fluentd 提供的更多内存，从而安全地处理更多日志。

获取集群中运行的 DaemonSet 的 spec：

```
kubectl get ds fluentd-gcp-v2.0 --namespace kube-system -o yaml > fluentd-gcp-ds.yaml
```

然后在 spec 文件中编辑资源需求，并使用以下命令更新 apiserver 中的 DaemonSet 对象：

```
kubectl replace -f fluentd-gcp-ds.yaml
```

一段时间后，Stackdriver 日志代理的 pod 将使用新配置重新启动。

更改 fluentd 参数

Fluentd 的配置存在 ConfigMap 对象中。它实际上是一组合并在一起的配置文件。你可以在[官方网站](#)上了解 fluentd 的配置。

假设你要向配置添加新的解析逻辑，以便 fluentd 可以理解默认的 Python 日志记录格式。一个合适的 fluentd 过滤器类似如下：

```
<filter reform.**>
type parser
format /^(<severity>\w):(<logger_name>\w):(<log>.*)/
reserve_data true
suppress_parse_error_log true
key_name log
</filter>
```

现在，你需要将其放入配置中，并使 Stackdriver 日志代理感知它。通过运行以下命令，获取集群中当前版本的 Stackdriver 日志机制的 ConfigMap：

```
kubectl get cm fluentd-gcp-config --namespace kube-system -o yaml > fluentd-gcp-configmap.yaml
```

然后在 containers.input.conf 键的值中，在 source 部分之后插入一个新的过滤器。

说明：顺序很重要。

在 apiserver 中更新 ConfigMap 比更新 DaemonSet 更复杂。最好考虑 Config Map 是不可变的。如果是这样，要更新配置，你应该使用新名称创建 ConfigMap，然后使用[上面的指南](#)将 DaemonSet 更改为指向它。

添加 fluentd 插件

Fluentd 用 Ruby 编写，并允许使用 [plugins](#) 扩展其功能。如果要使用默认的 Stackdriver 日志机制容器镜像中未包含的插件，则必须构建自定义镜像。假设你要为来自特定容器添加 Kafka 信息接收器，以进行其他处理。你可以复用默认的 [容器镜像源](#)，并仅添加少量更改：

- 将 Makefile 更改为指向你的容器仓库，例如 PREFIX=gcr.io/<your-project-id>。
- 将你的依赖项添加到 Gemfile 中，例如 gem 'fluent-plugin-kafka'。

然后在该目录运行 make build push。在更新 DaemonSet 以使用新镜像后，你就可以使用在 fluentd 配置中安装的插件了。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

在本地开发和调试服务

Kubernetes 应用程序通常由多个独立的服务组成，每个服务都在自己的容器中运行。在远端的 Kubernetes 集群上开发和调试这些服务可能很麻烦，需要 [在运行的容器上打开 Shell](#)，然后在远端 Shell 中运行你所需的工具。

telepresence 是一种工具，用于在本地轻松开发和调试服务，同时将服务代理到远程 Kubernetes 集群。使用 telepresence 可以为本地服务使用自定义工具（如调试器和 IDE），并提供对 Configmap、Secret 和远程集群上运行的服务的完全访问。

本文档描述如何在本地使用 telepresence 开发和调试远程集群上运行的服务。

准备开始

- Kubernetes 集群安装完毕
- 配置好 kubectl 与集群交互
- [Telepresence](#) 安装完毕

打开终端，不带参数运行 telepresence，以打开 telepresence Shell。这个 Shell 在本地运行，使你可以完全访问本地文件系统。

telepresence Shell 的使用方式多种多样。例如，在你的笔记本电脑上写一个 Shell 脚本，然后直接在 Shell 中实时运行它。你也可以在远端 Shell 上执行此操作，但这样可能无法使用首选的代码编辑器，并且在容器终止时脚本将被删除。

开发和调试现有的服务

在 Kubernetes 上开发应用程序时，通常对单个服务进行编程或调试。服务可能需要访问其他服务以进行测试和调试。一种选择是使用连续部署流水线，但即使最快的部署流水线也会在程序或调试周期中引入延迟。

使用 `--swap-deployment` 选项将现有部署与 Telepresence 代理交换。交换允许你在本地运行服务并能够连接到远端的 Kubernetes 集群。远端集群中的服务现在就可以访问本地运行的实例。

要运行 telepresence 并带有 `--swap-deployment` 选项，请输入：

```
telepresence --swap-deployment $DEPLOYMENT_NAME
```

这里的 `$DEPLOYMENT_NAME` 是你现有的部署名称。

运行此命令将生成 Shell。在该 Shell 中，启动你的服务。然后，你就可以在本地对源代码进行编辑、保存并能看到更改立即生效。你还可以在调试器或任何其他本地开发工具中运行服务。

接下来

如果你对实践教程感兴趣，请查看[本教程](#)，其中介绍了在 Google Kubernetes Engine 上本地开发 Guestbook 应用程序。

Telepresence 有[多种代理选项](#)，以满足你的各种情况。

要了解更多信息，请访问 [Telepresence 网站](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 10, 2020 at 11:40 AM PST: [\[zh\] Tidy up and fix links in tasks section \(2/10\) \(128182188\)](#)

审计

FEATURE STATE: Kubernetes v1.20 [beta]

Kubernetes 审计功能提供了与安全相关的按时间顺序排列的记录集，记录每个用户、管理员或系统其他组件影响系统的活动顺序。它能帮助集群管理员处理以下问题：

- 发生了什么？
- 什么时候发生的？
- 谁触发的？
- 活动发生在哪个（些）对象上？
- 在哪观察到的？
- 它从哪触发的？
- 活动的后续处理行为是什么？

审计记录最初产生于 [kube-apiserver](#) 内部。每个请求在不同执行阶段都会生成审计事件；这些审计事件会根据特定策略被预处理并写入后端。策略确定要记录的内容和用来存储记录的后端。当前的后端支持日志文件和 webhook。

每个请求都可以用相关的 "stage" 记录。已知的 stage 有：

- RequestReceived - 事件的 stage 将在审计处理器接收到请求后，并且在委托给其余处理器之前生成。
- ResponseStarted - 在响应消息的头部发送后，但是响应消息体发送前。这个阶段仅为长时间运行的请求生成（例如 watch）。
- ResponseComplete - 当响应消息体完成并且没有更多数据需要传输的时候。
- Panic - 当 panic 发生时生成。

说明： 审计日志记录功能会增加 API server 的内存消耗，因为需要为每个请求存储审计所需的某些上下文。此外，内存消耗取决于审计日志记录的配置。

审计策略

审计政策定义了关于应记录哪些事件以及应包含哪些数据的规则。审计策略对象结构定义在 [audit.k8s.io API 组](#) 处理事件时，将按顺序与规则列表进行比较。第一个匹配规则设置事件的 "审计级别"。已知的审计级别有：

- None - 符合这条规则的日志将不会记录。
- Metadata - 记录请求的元数据（请求的用户、时间戳、资源、动词等等），但是不记录请求或者响应的消息体。
- Request - 记录事件的元数据和请求的消息体，但是不记录响应的消息体。这不适用于非资源类型的请求。
- RequestResponse - 记录事件的元数据，请求和响应的消息体。这不适用于非资源类型的请求。

你可以使用 `--audit-policy-file` 标志将包含策略的文件传递给 `kube-apiserver`。如果不设置该标志，则不记录事件。注意 `rules` 字段 **必须** 在审计策略文件中提供。没有 (0) 规则的策略将被视为非法配置。

以下是一个审计策略文件的示例：

[audit/audit-policy.yaml](#)



```
apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in RequestReceived stage.
omitStages:
- "RequestReceived"
rules:
# Log pod changes at RequestResponse level
- level: RequestResponse
  resources:
  - group: ""
    # Resource "pods" doesn't match requests to any subresource of pods,
    # which is consistent with the RBAC policy.
    resources: ["pods"]
# Log "pods/log", "pods/status" at Metadata level
- level: Metadata
  resources:
  - group: ""
    resources: ["pods/log", "pods/status"]

# Don't log requests to a configmap called "controller-leader"
- level: None
  resources:
  - group: ""
    resources: ["configmaps"]
  resourceNames: ["controller-leader"]

# Don't log watch requests by the "system:kube-proxy" on endpoints or
services
- level: None
  users: ["system:kube-proxy"]
  verbs: ["watch"]
  resources:
  - group: "" # core API group
    resources: ["endpoints", "services"]

# Don't log authenticated requests to certain non-resource URL paths.
- level: None
```

```

userGroups: ["system:authenticated"]
nonResourceURLs:
- "/api*" # Wildcard matching.
- "/version"

# Log the request body of configmap changes in kube-system.
- level: Request
resources:
- group: "" # core API group
resources: ["configmaps"]
# This rule only applies to resources in the "kube-system" namespace.
# The empty string "" can be used to select non-namespaced resources.
namespaces: ["kube-system"]

# Log configmap and secret changes in all other namespaces at the Metadata level.
- level: Metadata
resources:
- group: "" # core API group
resources: ["secrets", "configmaps"]

# Log all other resources in core and extensions at the Request level.
- level: Request
resources:
- group: "" # core API group
- group: "extensions" # Version of group should NOT be included.

# A catch-all rule to log all other requests at the Metadata level.
- level: Metadata
# Long-running requests like watches that fall under this rule will not generate an audit event in RequestReceived.
omitStages:
- "RequestReceived"

```

你可以使用最低限度的审计策略文件在 Metadata 级别记录所有请求：

```

# 在 Metadata 级别为所有请求生成日志
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
- level: Metadata

```

管理员构建自己的审计配置文件时，可参考 [configure-helper.sh](#) 脚本，该脚本生成审计策略文件。你可以直接在脚本中看到审计策略的绝大部分内容。[]。

审计后端

审计后端实现将审计事件导出到外部存储。Kube-apiserver 默认提供两个后端：

- Log 后端，将事件写入到磁盘
- Webhook 后端，将事件发送到外部 API

在这两种情况下，审计事件结构均由 audit.k8s.io API 组中的 API 定义。当前版本的 API 是 [v1](#).

说明：

在 patch 请求的情况下，请求的消息体需要是一个 JSON 串指定 patch 操作，而不是一个完整的 Kubernetes API 对象 JSON 串。例如，以下的示例是一个合法的 patch 请求消息体，该请求对应 /apis/batch/v1/namespaces/some-namespace/jobs/some-job-name。

```
[  
  {  
    "op": "replace",  
    "path": "/spec/parallelism",  
    "value": 0  
  },  
  {  
    "op": "remove",  
    "path": "/spec/template/spec/containers/0/  
terminationMessagePolicy"  
  }  
]
```

Log 后端

Log 后端将审计事件写入 JSON 格式的文件。你可以使用以下 kube-apiserver 标志配置 Log 审计后端：

- --audit-log-path 指定用来写入审计事件的日志文件路径。不指定此标志会禁用日志后端。- 意味着标准化
- --audit-log-maxage 定义了保留旧审计日志文件的最大天数
- --audit-log-maxbackup 定义了要保留的审计日志文件的最大数量
- --audit-log-maxsize 定义审计日志文件的最大大小（兆字节）

如果 kube-apiserver 被配置为运行在 Pod 中，请记得将包含策略文件和日志文件的位置用 hostPath 挂载到 Pod 中。例如，

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml  
--audit-log-path=/var/log/audit.log
```

接下来挂载数据卷：

```
volumeMounts:
- mountPath: /etc/kubernetes/audit-policy.yaml
  name: audit
  readOnly: true
- mountPath: /var/log/audit.log
  name: audit-log
  readOnly: false
```

下面是 hostPath 卷本身。

```
- name: audit
hostPath:
  path: /etc/kubernetes/audit-policy.yaml
  type: File

- name: audit-log
hostPath:
  path: /var/log/audit.log
  type: FileOrCreate
```

Webhook 后端

Webhook 后端将审计事件发送到远程 API，该远程 API 应该暴露与 kube-apiserver 相同的 API。你可以使用如下 kube-apiserver 标志来配置 webhook 审计后端：

- --audit-webhook-config-file webhook 配置文件的路径。Webhook 配置文件实际上是一个 [kubeconfig 文件](#)
- --audit-webhook-initial-backoff 指定在第一次失败后重发请求等待的时间。随后的请求将以指数退避重试。

webhook 配置文件使用 kubeconfig 格式指定服务的远程地址和用于连接它的凭据。

批处理

log 和 webhook 后端都支持批处理。以 webhook 为例，以下是可用参数列表。要获取 log 后端的同样参数，请在参数名称中将 webhook 替换为 log。默认情况下，在 webhook 中启用批处理，在 log 中禁用批处理。同样，默认情况下，在 webhook 中启用带宽限制，在 log 中禁用带宽限制。

- --audit-webhook-mode 定义缓存策略，可选值如下：
 - batch - 以批处理缓存事件和异步的过程。这是默认值。
 - blocking - 在 API 服务器处理每个单独事件时，阻塞其响应。
 - blocking-strict - 与 blocking 相同，不过当审计日志在 RequestReceived 阶段失败时，整个 API 服务请求会失效。

以下参数仅用于 batch 模式。

- --audit-webhook-batch-buffer-size 定义 batch 之前要缓存的事件数。如果传入事件的速率溢出缓存区，则会丢弃事件。
- --audit-webhook-batch-max-size 定义一个 batch 中的最大事件数。
- --audit-webhook-batch-max-wait 无条件 batch 队列中的事件前等待的最大事件。
- --audit-webhook-batch-throttle-qps 每秒生成的最大批次数。
- --audit-webhook-batch-throttle-burst 在达到允许的 QPS 前，同一时刻允许存在的最大 batch 生成数。

参数调整

需要设置参数以适应 apiserver 上的负载。

例如，如果 kube-apiserver 每秒收到 100 个请求，并且每个请求仅在 Response Started 和 ResponseComplete 阶段进行审计，则应该考虑每秒生成约 200 个审计事件。假设批处理中最多有 100 个事件，则应将限制级别设置为至少 2 个 QPS。假设后端最多需要 5 秒钟来写入事件，你应该设置缓冲区大小以容纳最多 5 秒的事件，即 10 个 batch，即 1000 个事件。

但是，在大多数情况下，默认参数应该足够了，你不必手动设置它们。你可以查看 kube-apiserver 公开的以下 Prometheus 指标，并在日志中监控审计子系统的状态。

- apiserver_audit_event_total 包含所有暴露的审计事件数量的指标。
- apiserver_audit_error_total 在暴露时由于发生错误而被丢弃的事件的数量。

多 API 服务器的配置

如果你通过[聚合层](#)对 Kubernetes API 进行扩展，那么你也可以为聚合的 API 服务器设置审计日志。想要这么做，你需要以上述的格式给聚合的 API 服务器配置参数，并且配置日志管道以采用审计日志。不同的 API 服务器可以配置不同的审计配置和策略。

日志收集器示例

使用 fluentd 从日志文件中选择并且分发审计日志

[Fluentd](#) 是一个开源的数据采集器，可以从统一的日志层中采集。在以下示例中，我们将使用 fluentd 来按照命名空间划分审计事件。

说明： fluent-plugin-forest 和 fluent-plugin-rewrite-tag-filter fluentd 的插件。你可以在 [fluentd plugin-management](#) 了解安装插件相关的细节。

1. 在 kube-apiserver 节点上安装 [fluentd](#)、[fluent-plugin-forest](#) 和 [fluent-plugin-rewrite-tag-filter](#)。

为 fluentd 创建一个配置文件

2.

```
$ cat <<EOF > /etc/fluentd/config
# fluentd 运行在 kube-apiserver 相同的主机上
<source>
  @type tail
  # kube-apiserver 审计日志路径
  path /var/log/audit
  pos_file /var/log/audit.pos
  format json
  time_key time
  time_format %Y-%m-%dT%H:%M:%S.%N%z
  tag audit
</source>

<filter audit>
  #https://github.com/fluent/fluent-plugin-rewrite-tag-filter/issues/13
  type record_transformer
  enable_ruby
  <record>
    namespace ${record["objectRef"].nil? ? "none":(record["objectRef"]
    ["namespace"].nil? ? "none":record["objectRef"]["namespace"])}
  </record>
</filter>

<match audit>
  # 根据上下文中的名字空间元素对审计进行路由
  @type rewrite_tag_filter
  rewriterule1 namespace ^(.+) ${tag}.$1
</match>

<filter audit.**>
  @type record_transformer
  remove_keys namespace
</filter>

<match audit.**>
  @type forest
  subtype file
  remove_prefix audit
  <template>
    time_slice_format %Y%m%d%H
    compress gz
    path /var/log/audit-${tag}.*.log
    format json
    include_time_key true
  </template>
</match>
```

```
</template>
</match>
```

1. 启动 fluentd

```
fluentd -c /etc/fluentd/config -vv
```

1. 给 kube-apiserver 配置以下参数并启动：

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-log-path=/var/log/kube-audit --audit-log-format=json
```

1. 在 /var/log/audit-* .log 文件中检查不同命名空间的审计事件

使用 logstash 采集并分发 webhook 后端的审计事件

[Logstash](#) 是一个开源的、服务器端的数据处理工具。在下面的示例中，我们将使用 logstash 采集 webhook 后端的审计事件，并且将来自不同用户的事件存入不同的文件。

1. 安装 [logstash](#)

2. 为 logstash 创建配置文件

```
cat <<EOF > /etc/logstash/config
input{
    http{
        #TODO, figure out a way to use kubeconfig file to authenticate to
        logstash
        #https://www.elastic.co/guide/en/logstash/current/plugins-
        inputs-http.html#plugins-inputs-http-ssl
        port=>8888
    }
}
filter{
    split{
        # Webhook 审计后端与 EventList 一起发送若干事件
        # 对事件进行分割
        field=>[items]
        # 我们只需要 event 子元素，去掉其他元素
        remove_field=>[headers, metadata, apiVersion, "@timestamp",
        kind, "@version", host]
    }
    mutate{
        rename => {items=>event}
    }
}
output{
```

```
file{
    # 来自不同用户的审计事件会被保存到不同文件中
    path=> "/var/log/kube-audit-%{[event][user][username]}/audit"
}
}
```

1. 启动 logstash

```
bin/logstash -f /etc/logstash/config --path.settings /etc/logstash/
```

1. 为 kube-apiserver webhook 审计后端创建一个 [kubeconfig 文件](#) :

```
cat <<EOF > /etc/kubernetes/audit-webhook-kubeconfig
apiVersion: v1
clusters:
- cluster:
  server: http://<ip_of_logstash>:8888
  name: logstash
contexts:
- context:
  cluster: logstash
  user: ""
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users: []
EOF
```

1. 为 kube-apiserver 配置以下参数并启动 :

```
--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-webhook-
config-file=/etc/kubernetes/audit-webhook-kubeconfig
```

1. 在 logstash 节点的 /var/log/kube-audit-*/* 目录中检查审计事件

请注意，除了文件输出插件外，logstash 还有其它多种输出可以让用户路由不同的数据。例如，用户可以将审计事件发送给支持全文搜索和分析的 elasticsearch 插件。

接下来

了解 [Mutating webhook 审计注解](#)。

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 20, 2020 at 3:18 PM PST: [\[zh\] sync tasks/debug-application-cluster/audit.md \(2dac08f68\)](#)

应用故障排查

本指南帮助用户调试那些部署到 Kubernetes 上后没有正常运行的应用。本指南并非指导用户如何调试集群。如果想调试集群的话，请参阅[这里](#)。

诊断问题

故障排查的第一步是先给问题分类。问题是什么？是关于 Pods、Replication Controller 还是 Service？

- [调试 Pods](#)
- [调试副本控制器](#)
- [调试服务](#)

调试 Pods

调试 Pod 的第一步是查看 Pod 信息。用如下命令查看 Pod 的当前状态和最近的事件：

```
kubectl describe pods ${POD_NAME}
```

查看一下 Pod 中的容器所处的状态。这些容器的状态都是 Running 吗？最近有没有重启过？

后面的调试都是要依靠 Pod 的状态的。

Pod 停滞在 Pending 状态

如果一个 Pod 停滞在 Pending 状态，表示 Pod 没有被调度到节点上。通常这是因为某种类型的资源不足导致无法调度。查看上面的 kubectl describe ... 命令的输出，其中应该显示了为什么没被调度的原因。常见原因如下：

- **资源不足**: 你可能耗尽了集群上所有的 CPU 或内存。此时，你需要删除 Pod、调整资源请求或者为集群添加节点。更多信息请参阅[计算资源文档](#)
- **使用了 hostPort**: 如果绑定 Pod 到 hostPort，那么能够运行该 Pod 的节点就有限了。多数情况下，hostPort 是非必要的，而应该采用 Service 对象来暴露 Pod。如果确实需要使用 hostPort，那么集群中节点的个数就是所能创建的 Pod 的数量上限。

Pod 停滞在 Waiting 状态

如果 Pod 停滞在 Waiting 状态，则表示 Pod 已经被调度到某工作节点，但是无法在该节点上运行。同样，`kubectl describe ...` 命令的输出可能很有用。Waiting 状态的最常见原因是拉取镜像失败。要检查的有三个方面：

- 确保镜像名字拼写正确
- 确保镜像已被推送到镜像仓库
- 用手动命令 `docker pull <镜像>` 试试看镜像是否可拉取

Pod 处于 Crashing 或别的不健康状态

一旦 Pod 被调度，就可以采用 [调试运行中的 Pod](#) 中的方法来进一步调试。

Pod 处于 Running 态但是没有正常工作

如果 Pod 行为不符合预期，很可能 Pod 描述（例如你本地机器上的 `mypod.yaml`）中有问题，并且该错误在创建 Pod 时被忽略掉，没有报错。通常，Pod 的定义中节区嵌套关系错误、字段名字拼错的情况都会引起对应内容被忽略掉。例如，如果你误将 `command` 写成 `commnd`，Pod 虽然可以创建，但它不会执行你期望它执行的命令行。

可以做的第一件事是删除你的 Pod，并尝试带有 `--validate` 选项重新创建。例如，运行 `kubectl apply --validate -f mypod.yaml`。如果 `command` 被误拼成 `com mnd`，你将会看到下面的错误信息：

```
I0805 10:43:25.129850 46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973 46757 schema.go:129] this may be a false alarm,
see https://github.com/kubernetes/kubernetes/issues/6842
pods/mypod
```

接下来就要检查的是 API 服务器上的 Pod 与你所期望创建的是否匹配（例如，你原本使用本机上的一个 YAML 文件来创建 Pod）。例如，运行 `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml`，之后手动比较 `mypod.yaml` 与从 API 服务器取回的 Pod 描述。从 API 服务器处获得的 YAML 通常包含一些创建 Pod 所用的 YAML 中不存在的行，这是正常的。不过，如果如果源文件中有些行在 API 服务器版本中不存在，则意味着 Pod 规约是有问题的。

调试副本控制器

副本控制器相对比较简单直接。它们要么能创建 Pod，要么不能。如果不能创建 Pod，请参阅[上述说明](#)调试 Pod。

你也可以使用 `kubectl describe rc ${CONTROLLER_NAME}` 命令来检视副本控制器相关的事件。

调试服务

服务支持在多个 Pod 间负载均衡。有一些常见的问题可以造成服务无法正常工作。以下说明将有助于调试服务的问题。

首先，验证服务是否有端点。对于每一个 Service 对象，API 服务器为其提供 对应的 endpoints 资源。

通过如下命令可以查看 endpoints 资源：

```
kubectl get endpoints ${SERVICE_NAME}
```

确保 Endpoints 与服务成员 Pod 个数一致。例如，如果你的 Service 用来运行 3 个副本的 nginx 容器，你应该会在服务的 Endpoints 中看到 3 个不同的 IP 地址。

服务缺少 Endpoints

如果没有 Endpoints，请尝试使用 Service 所使用的标签列出 Pod。假定你的服务包含如下标签选择算符：

```
...
spec:
- selector:
  name: nginx
  type: frontend
```

你可以使用如下命令列出与选择算符相匹配的 Pod，并验证这些 Pod 是否归属于创建的服务：

```
kubectl get pods --selector=name=nginx,type=frontend
```

如果 Pod 列表符合预期，但是 Endpoints 仍然为空，那么可能暴露的端口不正确。如果服务指定了 containerPort，但是所选中的 Pod 没有列出该端口，这些 Pod 不会被添加到 Endpoints 列表。

验证 Pod 的 containerPort 与服务的 targetPort 是否匹配。

网络流量未被转发

如果你可以连接到服务上，但是连接立即被断开了，并且在 Endpoints 列表中有末端表项，可能是代理无法连接到 Pod。

要检查的有以下三项：

- Pod 工作是否正常？看一下重启计数，并参阅[调试 Pod](#)；
- 是否可以直接连接到 Pod？获取 Pod 的 IP 地址，然后尝试直接连接到该 IP；
- 应用是否在配置的端口上进行服务？Kubernetes 不进行端口重映射，所以如果应用在 8080 端口上服务，那么 containerPort 字段就要设定为 8080。

接下来

如果上述方法都不能解决你的问题，请按照[调试服务文档](#)中的介绍，确保你的 Service 处于 Running 状态，有 Endpoints 被创建，Pod 真的在提供服务；DNS 服务已配置并正常工作，iptables 规则也已安装并且 kube-proxy 也没有异常行为。

你也可以访问[故障排查文档](#)来获取更多信息。

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 December 17, 2020 at 7:07 PM PST: [Update debug-application.md \(d2bf36047\)](#)

应用自测与调试

运行应用时，不可避免的需要定位问题。前面我们介绍了如何使用 kubectl get pods 来查询 pod 的简单信息。除此之外，还有一系列的方法来获取应用的更详细信息。

使用 kubectl describe pod 命令获取 Pod 详情

与之前的例子类似，我们使用一个 Deployment 来创建两个 Pod。

[application/nginx-with-request.yaml](#)



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
```

```
app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
      ports:
        - containerPort: 80
```

使用如下命令创建 Deployment：

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-
request.yaml
```

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

使用如下命令查看 Pod 状态：

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1006230814-6winp	1/1	Running	0	11s
nginx-deployment-1006230814-fmgu3	1/1	Running	0	11s

我们可以使用 kubectl describe pod 命令来查询每个 Pod 的更多信息，比如：

```
kubectl describe pod nginx-deployment-1006230814-6winp
```

```
Name:           nginx-deployment-1006230814-6winp
Namespace:      default
Node:           kubernetes-node-wul5/10.240.0.9
Start Time:     Thu, 24 Mar 2016 01:39:49 +0000
Labels:         app=nginx,pod-template-hash=1006230814
Annotations:    kubernetes.io/created-
by={"kind":"SerializedReference","apiVersion":"v1","reference":-
{"kind":"ReplicaSet","namespace":"default","name":"nginx-
deployment-1956810328","uid":"14e607e7-8ba1-11e7-b5cb-fa16" ...}
Status:         Running
IP:             10.244.0.6
Controllers:    ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID:  docker://
90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb1149
    Image:        nginx
```

Image ID: docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e5163707
Port: 80/TCP
QoS Tier:
 cpu: Guaranteed
 memory: Guaranteed
Limits:
 cpu: 500m
 memory: 128Mi
Requests:
 memory: 128Mi
 cpu: 500m
State: Running
Started: Thu, 24 Mar 2016 01:39:51 +0000
Ready: True
Restart Count: 0
Environment: <none>
Mounts:
 /var/run/secrets/kubernetes.io/serviceaccount from default-token-5kdvl
(ro)
Conditions:
Type Status
Initialized True
Ready True
PodScheduled True
Volumes:
default-token-4bcbi:
 Type: Secret (a volume populated by a Secret)
 SecretName: default-token-4bcbi
 Optional: false
QoS Class: Guaranteed
Node-Selectors: <none>
Tolerations: <none>
Events:
FirstSeen LastSeen Count From SubobjectPath Type Reason Message

54s 54s 1 {default-scheduler }
Normal Scheduled Successfully assigned nginx-deployment-1006230814-6winp to kubernetes-node-wul5
54s 54s 1 {kubelet kubernetes-node-wul5}
spec.containers{nginx} Normal Pulling pulling image "nginx"
53s 53s 1 {kubelet kubernetes-node-wul5}
spec.containers{nginx} Normal Pulled Successfully pulled

```
image "nginx"
  53s   53s     1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Created      Created container
with docker id 90315cc9f513
  53s   53s     1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Started     Started container
with docker id 90315cc9f513
```

这里可以看到容器和 Pod 的标签、资源需求等配置信息，还可以看到状态、就绪态、重启次数、事件等状态信息。

容器状态是 Waiting、Running 和 Terminated 之一。根据状态的不同，还有对应的额外的信息——在这里你可以看到，对于处于运行状态的容器，系统会告诉你容器的启动时间。

Ready 指示是否通过了最后一个就绪态探测。(在本例中，容器没有配置就绪态探测；如果没有配置就绪态探测，则假定容器已经就绪。)

Restart Count 告诉你容器已重启的次数；这些信息对于定位配置了“Always”重启策略的容器持续崩溃问题非常有用。

目前，唯一与 Pod 有关的状态是 Ready 状况，该状况表明 Pod 能够为请求提供服务，并且应该添加到相应服务的负载均衡池中。

最后，你还可以看到与 Pod 相关的近期事件。系统通过指示第一次和最后一次看到事件以及看到该事件的次数来压缩多个相同的事件。“From”标明记录事件的组件，“SubobjectPath”告诉你引用了哪个对象（例如 Pod 中的容器），“Reason”和“Message”告诉你发生了什么。

例子：调试 Pending 状态的 Pod

可以使用事件来调试的一个常见的场景是，你创建 Pod 无法被调度到任何节点。比如，Pod 请求的资源比较多，没有任何一个节点能够满足，或者它指定了一个标签，没有节点可匹配。假定我们创建之前的 Deployment 时指定副本数是 5（不再是 2），并且请求 600 毫核（不再是 500），对于一个 4 个节点的集群，若每个节点只有 1 个 CPU，这时至少有一个 Pod 不能被调度。（需要注意的是，其他集群插件 Pod，比如 fluentd、skydns 等等会在每个节点上运行，如果我们需求 1000 毫核，将不会有 Pod 会被调度。）

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-1006230814-6winp	1/1	Running	0	7m
nginx-deployment-1006230814-fmgu3	1/1	Running	0	7m
nginx-deployment-1370807587-6ekbw	1/1	Running	0	1m
nginx-deployment-1370807587-fg172	0/1	Pending	0	1m
nginx-deployment-1370807587-fz9sd	0/1	Pending	0	1m

为了查找 Pod nginx-deployment-1370807587-fz9sd 没有运行的原因，我们可以使用 kubectl describe pod 命令描述 Pod，查看其事件：

```
kubectl describe pod nginx-deployment-1370807587-fz9sd
```

```
Name:      nginx-deployment-1370807587-fz9sd
Namespace: default
Node:      /
Labels:    app=nginx,pod-template-hash=1370807587
Status:    Pending
IP:
Controllers: ReplicaSet/nginx-deployment-1370807587
Containers:
  nginx:
    Image:    nginx
    Port:     80/TCP
    QoS Tier:
      memory: Guaranteed
      cpu:    Guaranteed
    Limits:
      cpu: 1
      memory: 128Mi
    Requests:
      cpu: 1
      memory: 128Mi
    Environment Variables:
Volumes:
  default-token-4bcbi:
    Type:  Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
Events:
FirstSeen   LastSeen  Count  From                    SubobjectPath
Type        Reason            Message
-----  -----
1m          48s       7  {default-scheduler }
Warning     FailedScheduling  pod (nginx-deployment-1370807587-fz9sd) failed to fit in any node
               fit failure on node (kubernetes-node-6ta5): Node didn't have enough
               resource: CPU, requested: 1000, used: 1420, capacity: 2000
               fit failure on node (kubernetes-node-wul5): Node didn't have enough
               resource: CPU, requested: 1000, used: 1100, capacity: 2000
```

这里你可以看到由调度器记录的事件，它表明了 Pod 不能被调度的原因是 FailedScheduling（也可能是其他值）。其 message 部分表明没有任何节点拥有足够多的资源。

要纠正这种情况，可以使用 `kubectl scale` 更新 Deployment，以指定 4 个或更少的副本。（或者你可以让 Pod 继续保持这个状态，这是无害的。）

你在 `kubectl describe pod` 结尾处看到的事件都保存在 etcd 中，并提供关于集群中正在发生的事情的高级信息。如果需要列出所有事件，可使用命令：

```
kubectl get events
```

但是，需要注意的是，事件是区分名字空间的。如果你对某些名字空间域的对象（比如 `my-namespace` 名字下的 Pod）的事件感兴趣，你需要显式地在命令行中指定名字空间：

```
kubectl get events --namespace=my-namespace
```

查看所有 namespace 的事件，可使用 `--all-namespaces` 参数。

除了 `kubectl describe pod` 以外，另一种获取 Pod 额外信息（除了 `kubectl get pod`）的方法是给 `kubectl get pod` 增加 `-o yaml` 输出格式参数。该命令将以 YAML 格式为你提供比 `kubectl describe pod` 更多的信息——实际上是系统拥有的关于 Pod 的所有信息。在这里，你将看到注解（没有标签限制的键值元数据，由 Kubernetes 系统组件在内部使用）、重启策略、端口和卷等。

```
kubectl get pod nginx-deployment-1006230814-6winp -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: |
      {"kind": "SerializedReference", "apiVersion": "v1", "reference": {"kind": "ReplicaSet", "namespace": "default", "name": "nginx-deployment-1006230814", "uid": "4c84c175-f161-11e5-9a78-42010af00005", "apiVersion": "extensions", "resourceVersion": "133434"}}
  creationTimestamp: 2016-03-24T01:39:50Z
  generateName: nginx-deployment-1006230814-
  labels:
    app: nginx
    pod-template-hash: "1006230814"
  name: nginx-deployment-1006230814-6winp
  namespace: default
  resourceVersion: "133447"
  uid: 4c879808-f161-11e5-9a78-42010af00005
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
    ports:
```

```
- containerPort: 80
  protocol: TCP
resources:
  limits:
    cpu: 500m
    memory: 128Mi
  requests:
    cpu: 500m
    memory: 128Mi
terminationMessagePath: /dev/termination-log
volumeMounts:
- mountPath: /var/run/secrets/kubernetes.io/serviceaccount
  name: default-token-4bcbi
  readOnly: true
dnsPolicy: ClusterFirst
nodeName: kubernetes-node-wul5
restartPolicy: Always
securityContext: {}
serviceAccount: default
serviceAccountName: default
terminationGracePeriodSeconds: 30
volumes:
- name: default-token-4bcbi
  secret:
    secretName: default-token-4bcbi
status:
  conditions:
- lastProbeTime: null
  lastTransitionTime: 2016-03-24T01:39:51Z
  status: "True"
  type: Ready
  containerStatuses:
- containerID: docker://
  90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb1149
  image: nginx
  imageID: docker://
  6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e5163707
  lastState: {}
  name: nginx
  ready: true
  restartCount: 0
  state:
    running:
      startedAt: 2016-03-24T01:39:51Z
  hostIP: 10.240.0.9
  phase: Running
```

podIP: 10.244.0.6

startTime: 2016-03-24T01:39:49Z

示例：调试宕机或无法联系的节点

有时候，在调试时，查看节点的状态是很有用的——例如，因为你已经注意到节点上运行的 Pod 的奇怪行为，或者想了解为什么 Pod 不会调度到节点上。与 Pod 一样，你可以使用 kubectl describe node 和 kubectl get node -o yaml 来查询节点的详细信息。例如，如果某个节点宕机（与网络断开连接，或者 kubelet 挂掉无法重新启动等等），你将看到以下情况。请注意显示节点未就绪的事件，也请注意 Pod 不再运行（它们在5分钟未就绪状态后被驱逐）。

kubectl get nodes

NAME	STATUS	ROLES	AGE	VERSION
kubernetes-node-861h	NotReady	<none>	1h	v1.13.0
kubernetes-node-bols	Ready	<none>	1h	v1.13.0
kubernetes-node-st6x	Ready	<none>	1h	v1.13.0
kubernetes-node-unaj	Ready	<none>	1h	v1.13.0

kubectl describe node kubernetes-node-861h

Name:	kubernetes-node-861h			
Role				
Labels:	kubernetes.io/arch=amd64 kubernetes.io/os=linux kubernetes.io/hostname=kubernetes-node-861h			
Annotations:	node.alpha.kubernetes.io/ttl=0 volumes.kubernetes.io/controller-managed-attach-detach=true			
Taints:	<none>			
CreationTimestamp:	Mon, 04 Sep 2017 17:13:23 +0800			
Phase:				
Conditions:				
Type	Status	LastHeartbeatTime	Reason	Message
LastTransitionTime				
----	-----	-----	-----	-----
OutOfDisk	Unknown	Fri, 08 Sep 2017 16:04:28 +0800	Fri, 08 Sep 2017 16:20:58 +0800	NodeStatusUnknown Kubelet stopped posting node status.
MemoryPressure	Unknown	Fri, 08 Sep 2017 16:04:28 +0800	Fri, 08 Sep 2017 16:20:58 +0800	NodeStatusUnknown Kubelet stopped posting node status.
DiskPressure	Unknown	Fri, 08 Sep 2017 16:04:28 +0800	Fri, 08 Sep 2017 16:20:58 +0800	NodeStatusUnknown Kubelet stopped posting node status.
Ready	Unknown	Fri, 08 Sep 2017 16:04:28 +0800	Fri, 08 Sep 2017 16:20:58 +0800	NodeStatusUnknown Kubelet stopped

```

posting node status.

Addresses:      10.240.115.55,104.197.0.26
Capacity:
cpu:            2
hugePages:      0
memory:         4046788Ki
pods:           110
Allocatable:
cpu:            1500m
hugePages:      0
memory:         1479263Ki
pods:           110
System Info:
Machine ID:      8e025a21a4254e11b028584d9d8b12c4
System UUID:      349075D1-D169-4F25-9F2A-E886850C47E3
Boot ID:          5cd18b37-c5bd-4658-94e0-e436d3f110e0
Kernel Version:   4.4.0-31-generic
OS Image:         Debian GNU/Linux 8 (jessie)
Operating System: linux
Architecture:     amd64
Container Runtime Version: docker://1.12.5
Kubelet Version:   v1.6.9+a3d1dfa6f4335
Kube-Proxy Version: v1.6.9+a3d1dfa6f4335
ExternalID:        15233045891481496305
Non-terminated Pods: (9 in total)
  Namespace          Name             CPU Requests
  CPU Limits    Memory Requests Memory Limits
  -----        -----
  .....          .....
Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  -----
  900m (60%)   2200m (146%) 1009286400 (66%)  5681286400 (375%)
Events:          <none>

```

```
kubectl get node kubernetes-node-861h -o yaml
```

```

apiVersion: v1
kind: Node
metadata:
  creationTimestamp: 2015-07-10T21:32:29Z
  labels:
    kubernetes.io/hostname: kubernetes-node-861h
  name: kubernetes-node-861h

```

```
resourceVersion: "757"
selfLink: /api/v1/nodes/kubernetes-node-861h
uid: 2a69374e-274b-11e5-a234-42010af0d969
spec:
  externalID: "15233045891481496305"
  podCIDR: 10.244.0.0/24
  providerID: gce://striped-torus-760/us-central1-b/kubernetes-node-861h
status:
  addresses:
    - address: 10.240.115.55
      type: InternalIP
    - address: 104.197.0.26
      type: ExternalIP
  capacity:
    cpu: "1"
    memory: 3800808Ki
    pods: "100"
  conditions:
    - lastHeartbeatTime: 2015-07-10T21:34:32Z
      lastTransitionTime: 2015-07-10T21:35:15Z
      reason: Kubelet stopped posting node status.
      status: Unknown
      type: Ready
  nodeInfo:
    bootID: 4e316776-b40d-4f78-a4ea-ab0d73390897
    containerRuntimeVersion: docker://Unknown
    kernelVersion: 3.16.0-0.bpo.4-amd64
    kubeProxyVersion: v0.21.1-185-gffc5a86098dc01
    kubeletVersion: v0.21.1-185-gffc5a86098dc01
    machineID: ""
    osImage: Debian GNU/Linux 7 (wheezy)
    systemUUID: ABE5F6B4-D44B-108B-C46A-24CCE16C8B6E
```

接下来

了解更多的调试工具：

- [日志](#)
- [监控](#)
- [使用 exec 进入容器](#)
- [使用代理连接容器](#)
- [使用端口转发连接容器](#)
- [使用 crictl 检查节点](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 09, 2020 at 7:17 PM PST: [\[zh\] Tidy up and fix links in tasks section \(1/10\) \(15845b322\)](#)

故障诊断

有时候事情会出错。本指南旨在解决这些问题。它包含两个部分：

- [应用排错](#) - 针对部署代码到 Kubernetes 并想知道代码为什么不能正常运行的用户。
- [集群排错](#) - 针对集群管理员以及 Kubernetes 集群表现异常的用户。

你也应该查看所用[发行版本](#)的已知问题。

获取帮助

如果你的问题在上述指南中没有得到答案，你还有另外几种方式从 Kubernetes 团队获得帮助。

问题

本网站上的文档针对回答各类问题进行了结构化组织和分类。[概念](#)部分解释 Kubernetes 体系结构以及每个组件的工作方式，[安装](#)部分提供了安装的实用说明。[任务](#)部分展示了如何完成常用任务，[教程](#)部分则提供对现实世界、特定行业或端到端开发场景的更全面的演练。[参考](#)部分提供了详细的 [Kubernetes API](#) 文档和命令行 (CLI) 接口的文档，例如[kubectl](#)。

求救！我的问题还没有解决！我现在需要帮助！

Stack Overflow

社区中的其他人可能已经问过和你类似的问题，也可能能够帮助解决你的问题。Kubernetes 团队还会监视[带有 Kubernetes 标签的帖子](#)。如果现有的问题对你没有帮助，请[问一个新问题](#)！

Slack

Kubernetes 社区中有很多人在 #kubernetes-users 这一 Slack 频道聚集。Slack 需要注册；你可以[请求一份邀请](#)，并且注册是对所有人开放的。欢迎你随时来问任何问题。一旦注册了，就可以访问通过 Web 浏览器或者 Slack 专用的应用访问 [Slack 上的 Kubernetes 组织](#)。

一旦你完成了注册，就可以浏览各种感兴趣主题的频道列表（一直在增长）。例如，Kubernetes 新人可能还想加入 [#kubernetes-novice](#) 频道。又比如，开发人员应该加入 [#kubernetes-dev](#) 频道。

还有许多国家/地区语言频道。请随时加入这些频道以获得本地化支持和信息：

国家	频道
中国	#cn-users , #cn-events
芬兰	#fi-users
法国	#fr-users , #fr-events
德国	#de-users , #de-events
印度	#in-users , #in-events
意大利	#it-users , #it-events
日本	#jp-users , #jp-events
韩国	#kr-users
荷兰	#nl-users
挪威	#norw-users
波兰	#pl-users
俄罗斯	#ru-users
西班牙	#es-users
瑞典	#se-users
土耳其	#tr-users , #tr-events

论坛

欢迎你加入 Kubernetes 官方论坛 [discuss.kubernetes.io](#)。

Bugs 和功能请求

如果你发现一个看起来像 Bug 的问题，或者你想提出一个功能请求，请使用 [Github 问题跟踪系统](#)。

在提交问题之前，请搜索现有问题列表以查看是否其中已涵盖你的问题。

如果提交 Bug，请提供如何重现问题的详细信息，例如：

- Kubernetes 版本：kubectl version
- 云平台、OS 发行版、网络配置和 Docker 版本
- 重现问题的步骤

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 20, 2020 at 10:52 AM PST: [\[zh\] resync troubleshooting \(b554875de\)](#)

确定 Pod 失败的原因

本文介绍如何编写和读取容器的终止消息。

终止消息为容器提供了一种方法，可以将有关致命事件的信息写入某个位置，在该位置可以通过仪表板和监控软件等工具轻松检索和显示致命事件。在大多数情况下，您放入终止消息中的信息也应该写入 [常规 Kubernetes 日志](#)。

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.

读写终止消息

在本练习中，您将创建运行一个容器的 Pod。配置文件指定在容器启动时要运行的命令。

[debug/termination.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
    - name: termination-demo-container
      image: debian
```

```
command: ["/bin/sh"]
args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

1. 基于 YAML 配置文件创建 Pod :

```
kubectl create -f https://k8s.io/examples/debug/termination.yaml
```

YAML 文件中，在 cmd 和 args 字段，你可以看到容器休眠 10 秒然后将 "Sleep expired" 写入 /dev/termination-log 文件。容器写完 "Sleep expired" 消息后就终止了。

2. 显示 Pod 的信息：

```
kubectl get pod termination-demo
```

重复前面的命令直到 Pod 不再运行。

3. 显示 Pod 的详细信息：

```
kubectl get pod --output=yaml
```

输出结果包含 "Sleep expired" 消息：

```
apiVersion: v1
kind: Pod
...
lastState:
  terminated:
    containerID: ...
    exitCode: 0
    finishedAt: ...
    message: |
      Sleep expired
...
```

4. 使用 Go 模板过滤输出结果，使其只含有终止消息：

```
kubectl get pod termination-demo -o go-template="{{range .status.containerStatuses}}{{.lastState.terminated.message}}{{end}}"
```

定制终止消息

Kubernetes 从容器的 terminationMessagePath 字段中指定的终止消息文件中检索终止消息，默认值为 /dev/termination-log。通过定制这个字段，您可以告诉 Kubernetes 使用不同的文件。Kubernetes 使用指定文件中的内容在成功和失败时填充容器的状态消息。

在下例中，容器将终止消息写入 /tmp/my-log 给 Kubernetes 来接收：

```
apiVersion: v1
kind: Pod
metadata:
  name: msg-path-demo
spec:
  containers:
    - name: msg-path-demo-container
      image: debian
      terminationMessagePath: "/tmp/my-log"
```

此外，用户可以设置容器的 `terminationMessagePolicy` 字段，以便进一步自定义。此字段默认为 "File"，这意味着仅从终止消息文件中检索终止消息。通过将 `terminationMessagePolicy` 设置为 "FallbackToLogsOnError"，你就可以告诉 Kubernetes，在容器因错误退出时，如果终止消息文件为空，则使用容器日志输出的最后一块作为终止消息。日志输出限制为 2048 字节或 80 行，以较小者为准。

接下来

- 参考 [Container](#) 资源的 `terminationMessagePath` 字段。
- 了解 [接收日志](#)。
- 了解 [Go 模版](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 10, 2020 at 11:40 AM PST: [\[zh\] Tidy up and fix links in tasks section \(2/10\) \(128182188\)](#)

节点健康监测

节点问题探测器 是一个 [DaemonSet](#)，用来监控节点健康。它从各种守护进程收集节点问题，并以 [NodeCondition](#) 和 [Event](#) 的形式报告给 API 服务器。

它现在支持一些已知的内核问题检测，并将随着时间的推移，检测更多节点问题。

目前，Kubernetes 不会对节点问题探测器监测到的节点状态和事件采取任何操作。将来可能会引入一个补救系统来处理这些节点问题。

更多信息请参阅 [这里](#)。

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`.

局限性

- 节点问题检测器的内核问题检测现在只支持基于文件类型的内核日志。它不支持像 journald 这样的命令行日志工具。
- 节点问题检测器的内核问题检测对内核日志格式有一定要求，现在它只适用于 Ubuntu 和 Debian。不过将其扩展为[支持其它日志格式](#)也很容易。

在 GCE 集群中启用/禁用

节点问题检测器在 gce 集群中以 [集群插件的形式](#) 默认启用。

你可以在运行 `kube-up.sh` 之前，以设置环境变量 `KUBE_ENABLE_NODE_PROBLEM_DETECTOR` 的形式启用/禁用它。

在其它环境中使用

要在 GCE 之外的其他环境中启用节点问题检测器，你可以使用 `kubectl` 或插件 pod。

Kubectl

这是在 GCE 之外启动节点问题检测器的推荐方法。它的管理更加灵活，例如覆盖默认配置以使其适合你的环境或检测自定义节点问题。

- **步骤 1:** `node-problem-detector.yaml`:

[debug/node-problem-detector.yaml](#)



```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
```

```
kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: k8s.gcr.io/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
              cpu: "200m"
              memory: "100Mi"
            requests:
              cpu: "20m"
              memory: "20Mi"
          volumeMounts:
            - name: log
              mountPath: /log
              readOnly: true
          volumes:
            - name: log
              hostPath:
                path: /var/log/
```

请注意保证你的系统日志路径与你的 OS 发行版相对应。

- 步骤 2: 执行 kubectl 来启动节点问题检测器：

```
kubectl create -f https://k8s.io/examples/debug/node-problem-detector.yaml
```

插件 Pod

这适用于拥有自己的集群引导程序解决方案的用户，并且不需要覆盖默认配置。他们可以利用插件 Pod 进一步自动化部署。

只需创建 node-problem-detector.yaml，并将其放在主节点上的插件 pod 目录 /etc/kubernetes/addons/node-problem-detector 下。

覆盖配置文件

构建节点问题检测器的 docker 镜像时，会嵌入 [默认配置](#)。

不过，你可以像下面这样使用 [ConfigMap](#) 将其覆盖：

- **步骤 1:** 在 config/ 中更改配置文件。
- **步骤 2:** 使用 kubectl create configmap node-problem-detector-config --from-file=config/ 创建 node-problem-detector-config 。
- **步骤 3:** 更改 node-problem-detector.yaml 以使用 ConfigMap:
[debug/node-problem-detector-configmap.yaml](#)



```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: k8s.gcr.io/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
```

```

cpu: "200m"
memory: "100Mi"
requests:
  cpu: "20m"
  memory: "20Mi"
volumeMounts:
- name: log
  mountPath: /log
  readOnly: true
- name: config # Overwrite the config/ directory with ConfigMap
volume
  mountPath: /config
  readOnly: true
volumes:
- name: log
  hostPath:
    path: /var/log/
- name: config # Define ConfigMap volume
  configMap:
    name: node-problem-detector-config

```

- 步骤 4: 使用新的 yaml 文件重新创建节点问题检测器 :

```

kubectl delete -f https://k8s.io/examples/debug/node-problem-
detector.yaml # If you have a node-problem-detector running
kubectl create -f https://k8s.io/examples/debug/node-problem-detector-
configmap.yaml

```

请注意，此方法仅适用于通过 `kubectl` 启动的节点问题检测器。

由于插件管理器不支持ConfigMap，因此现在不支持对于作为集群插件运行的节点问题检测器的配置进行覆盖。

内核监视器

内核监视器 是节点问题检测器中的问题守护进程。它监视内核日志并按照预定义规则检测已知内核问题。

内核监视器根据 [config/kernel-monitor.json](#) 中的一组预定义规则列表匹配内核问题。规则列表是可扩展的，你始终可以通过覆盖配置来扩展它。

添加新的 NodeCondition

你可以使用新的状态描述来扩展 config/kernel-monitor.json 中的 conditions 字段以支持新的节点状态。

```
{  
  "type": "NodeConditionType",  
  "reason": "CamelCaseDefaultNodeConditionReason",  
  "message": "arbitrary default node condition message"  
}
```

检测新的问题

你可以使用新的规则描述来扩展 config/kernel-monitor.json 中的 rules 字段以检测新问题。

```
{  
  "type": "temporary/permanent",  
  "condition": "NodeConditionOfPermanentIssue",  
  "reason": "CamelCaseShortReason",  
  "message": "regexp matching the issue in the kernel log"  
}
```

更改日志路径

不同操作系统发行版的内核日志的可能不同。 config/kernel-monitor.json 中的 log 字段是容器内的日志路径。你始终可以修改配置使其与你的 OS 发行版匹配。

支持其它日志格式

内核监视器使用 [Translator] 插件将内核日志转换为内部数据结构。我们可以很容易为新的日志格式实现新的翻译器。

注意事项

我们建议在集群中运行节点问题检测器来监视节点运行状况。但是，你应该知道这将在每个节点上引入额外的资源开销。一般情况下没有影响，因为：

- 内核日志生成相对较慢。
- 节点问题检测器有资源限制。
- 即使在高负载下，资源使用也是可以接受的。 (参阅 [基准测试结果](#))

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 10, 2020 at 11:40 AM PST: [\[zh\] Tidy up and fix links in tasks section \(2/10\) \(128182188\)](#)

获取正在运行容器的 Shell

本文介绍怎样使用 kubectl exec 命令获取正在运行容器的 Shell。

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.

获取容器的 Shell

在本练习中，你将创建包含一个容器的 Pod。容器运行 nginx 镜像。下面是 Pod 的配置文件：

[application/shell-demo.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: shell-demo
spec:
  volumes:
    - name: shared-data
      emptyDir: {}
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: shared-data
          mountPath: /usr/share/nginx/html
  hostNetwork: true
  dnsPolicy: Default
```

创建 Pod：

```
kubectl create -f https://k8s.io/examples/application/shell-demo.yaml
```

检查容器是否运行正常：

```
kubectl get pod shell-demo
```

获取正在运行容器的 Shell：

```
kubectl exec -it shell-demo -- /bin/bash
```

说明：

双破折号 "--" 用于将要传递给命令的参数与 kubectl 的参数分开。

在 shell 中，打印根目录：

```
root@shell-demo:/# ls /
```

在 shell 中，实验其他命令。下面是一些示例：

```
root@shell-demo:/# ls /
root@shell-demo:/# cat /proc/mounts
root@shell-demo:/# cat /proc/1/maps
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install -y tcpdump
root@shell-demo:/# tcpdump
root@shell-demo:/# apt-get install -y lsof
root@shell-demo:/# lsof
root@shell-demo:/# apt-get install -y procps
root@shell-demo:/# ps aux
root@shell-demo:/# ps aux | grep nginx
```

编写 nginx 的 根页面

在看一下 Pod 的配置文件。该 Pod 有个 emptyDir 卷，容器将该卷挂载到了 /usr/share/nginx/html。

在 shell 中，在 /usr/share/nginx/html 目录创建一个 index.html 文件：

```
root@shell-demo:/# echo Hello shell demo > /usr/share/nginx/html/
index.html
```

在 shell 中，向 nginx 服务器发送 GET 请求：

```
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install curl
root@shell-demo:/# curl localhost
```

输出结果显示了你在 index.html 中写入的文本。

```
Hello shell demo
```

当用完 shell 后，输入 exit 退出。

在容器中运行单个命令

在普通的命令窗口（而不是 shell）中，打印环境运行容器中的变量：

```
kubectl exec shell-demo env
```

实验运行其他命令。下面是一些示例：

```
kubectl exec shell-demo ps aux  
kubectl exec shell-demo ls /  
kubectl exec shell-demo cat /proc/1/mounts
```

当 Pod 包含多个容器时打开 shell

如果 Pod 有多个容器，--container 或者 -c 可以在 kubectl exec 命令中指定容器。例如，您有个名为 my-pod 的容器，该 Pod 有两个容器分别为 main-app 和 helper-app。下面的命令将会打开一个 shell 访问 main-app 容器。

```
kubectl exec -it my-pod --container main-app -- /bin/bash
```

接下来

- [kubectl exec](#)

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 27, 2020 at 2:28 PM PST: [Fix format error \(15c6e3899\)](#)

调试 Init 容器

此页显示如何核查与 Init 容器执行相关的问题。下面的示例命令行将 Pod 称为 <pod-name>，而 Init 容器称为 <init-container-1> 和 <init-container-2>。

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.

- 你应该熟悉 [Init 容器](#)的基础知识。
- 你应该已经[配置好一个 Init 容器](#)。

检查 Init 容器的状态

显示你的 Pod 的状态：

```
kubectl get pod <pod-name>
```

例如，状态 Init:1/2 表明两个 Init 容器中的一个已经成功完成：

NAME	READY	STATUS	RESTARTS	AGE
<pod-name>	0/1	Init:1/2	0	7s

更多状态值及其含义请参考[理解 Pod 的状态](#)。

获取 Init 容器详情

查看 Init 容器运行的更多详情：

```
kubectl describe pod <pod-name>
```

例如，对于包含两个 Init 容器的 Pod 可能显示如下信息：

```
Init Containers:  
<init-container-1>:  
  Container ID: ...  
  ...  
  State: Terminated  
    Reason: Completed  
    Exit Code: 0  
    Started: ...  
    Finished: ...  
  Ready: True  
  Restart Count: 0  
  ...  
<init-container-2>:  
  Container ID: ...  
  ...  
  State: Waiting
```

```
Reason: CrashLoopBackOff
Last State: Terminated
Reason: Error
Exit Code: 1
Started: ...
Finished: ...
Ready: False
Restart Count: 3
...
```

你还可以通过编程方式读取 Pod Spec 上的 status.initContainerStatuses 字段，了解 Init 容器的状态：

```
kubectl get pod nginx --template '{{.status.initContainerStatuses}}'
```

此命令将返回与原始 JSON 中相同的信息。

通过 Init 容器访问日志

与 Pod 名称一起传递 Init 容器名称，以访问容器的日志。

```
kubectl logs <pod-name> -c <init-container-2>
```

运行 Shell 脚本的 Init 容器在执行 Shell 脚本时输出命令本身。例如，你可以在 Bash 中通过在脚本的开头运行 set -x 来实现。

理解 Pod 的状态

以 Init: 开头的 Pod 状态汇总了 Init 容器执行的状态。下表介绍调试 Init 容器时可能看到的一些状态值示例。

状态	含义
Init:N/M	Pod 包含 M 个 Init 容器，其中 N 个已经运行完成。
Init:Error	Init 容器已执行失败。
Init:CrashLoopBackOff	Init 容器执行总是失败。
Pending	Pod 还没有开始执行 Init 容器。
PodInitializing or Running	Pod 已经完成执行 Init 容器。

反馈

此页是否对您有帮助？

是 否

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

调试 Pods 和 ReplicationControllers

此页面展示如何调试 Pod 和 ReplicationController。

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.

- 你应该先熟悉 [Pods](#) 和 [Pod 生命周期](#) 的基础概念。

调试 Pod

调试一个 pod 的第一步是观察它。使用下面的命令检查 Pod 的当前状态和最近事件：

```
kubectl describe pods ${POD_NAME}
```

看看 Pod 中的容器的状态。它们都是 Running 吗？最近有重启吗？

根据 Pod 的状态继续调试。

我的 Pod 停滞在 Pending 状态

如果 Pod 被卡在 Pending 状态，就意味着它不能调度在某个节点上。一般来说，这是因为某种类型的资源不足而 导致无法调度。查看上面的命令 kubectl describe ... 的输出。调度器的消息中应该会包含无法调度 Pod 的原因。原因包括：

资源不足

你可能已经耗尽了集群中供应的 CPU 或内存。在这个情况下你可以尝试几件事情：

- 向集群中添加节点。
- [终止不需要的 Pod](#) 为 Pending 状态的 Pod 提供空间。

- 检查该 Pod 是否不大于你的节点。例如，如果全部节点具有 cpu:1 容量，那么具有 请求为 cpu: 1.1 的 Pod 永远不会被调度。

你可以使用 `kubectl get nodes -o <format>` 命令来检查节点容量。下面是一些能够提取必要信息的命令示例：

```
kubectl get nodes -o yaml | egrep '\sname:|cpu:|memory:'  
kubectl get nodes -o json | jq '.items[] | {name: .metadata.name,  
cap: .status.capacity}'
```

可以考虑配置[资源配置](#) 来限制可耗用的资源总量。如果与命名空间一起使用，它可以防止一个团队吞噬所有的资源。

使用hostPort

当你将一个 Pod 绑定到某 hostPort 时，这个 Pod 能被调度的位置数量有限。在大多数情况下，hostPort 是不必要的；尝试使用服务对象来暴露你的 Pod。如果你需要 hostPort，那么你可以调度的 Pod 数量不能超过集群的节点个数。

我的 Pod 一直在 Waiting

如果 Pod 一直停滞在 Waiting 状态，那么它已被调度在某个工作节点，但它不能在该机器上运行。再次，来自 `kubectl describe ...` 的内容应该是可以是很有用的。最常见的原因 Waiting 的 Pod 是无法拉取镜像。有三件事要检查：

- 确保你的镜像的名称正确。
- 你是否将镜像推送到存储库？
- 在你的机器上手动运行 `docker pull <image>`，看看是否可以拉取镜像。

我的 Pod 一直 Crashing 或者其他不健康状态

一旦 Pod 已经被调度，就可以依据[调试运行中的 Pod](#) 展开进一步的调试工作。

调试 Replication Controller

Replication Controller 相当简单。它们或者能或者不能创建 Pod。如果它们无法创建 Pod，请参考[上面的说明](#) 来调试你的 Pod。

你也可以使用 `kubectl describe rc ${CONTROLLER_NAME}` 来检查和副本控制器有关的事件。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

调试 Service

对于新安装的 Kubernetes，经常出现的问题是 Service 无法正常运行。你已经通过 Deployment (或其他工作负载控制器) 运行了 Pod，并创建 Service，但是当你尝试访问它时，没有任何响应。此文档有望对你有所帮助并找出问题所在。

在 Pod 中运行命令

对于这里的许多步骤，你可能希望知道运行在集群中的 Pod 看起来是什么样的。最简单的方法是运行一个交互式的 alpine Pod：

```
$ kubectl run -it --rm --restart=Never alpine --image=alpine sh
```

说明：如果没有看到命令提示符，请按回车。

如果你已经有了你想使用的正在运行的 Pod，则可以运行以下命令去进入：

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

设置

为了完成本次实践的任务，我们先运行几个 Pod。由于你可能正在调试自己的 Service，所以，你可以使用自己的信息进行替换，或者你也可以跟着教程并开始下面的步骤来获得第二个数据点。

```
kubectl create deployment hostnames --image=k8s.gcr.io/serve_hostname  
deployment.apps/hostnames created
```

kubectl 命令将打印创建或变更的资源的类型和名称，它们可以在后续命令中使用。让我们将这个 deployment 的副本数扩至 3。

```
kubectl scale deployment hostnames --replicas=3  
deployment.apps/hostnames scaled
```

请注意这与你使用以下 YAML 方式启动 Deployment 类似：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:
```

```
labels:  
  app: hostnames  
  name: hostnames  
spec:  
  selector:  
    matchLabels:  
      app: hostnames  
  replicas: 3  
  template:  
    metadata:  
      labels:  
        app: hostnames  
    spec:  
      containers:  
      - name: hostnames  
        image: k8s.gcr.io/serve_hostname
```

"app" 标签是 `kubectl create deployment` 根据 Deployment 名称自动设置的。

确认你的 Pods 是运行状态：

```
kubectl get pods -l app=hostnames
```

NAME	READY	STATUS	RESTARTS	AGE
hostnames-632524106-bbpiw	1/1	Running	0	2m
hostnames-632524106-ly40y	1/1	Running	0	2m
hostnames-632524106-tlaok	1/1	Running	0	2m

你还可以确认你的 Pod 是否正在提供服务。你可以获取 Pod IP 地址列表并直接对其进行测试。

```
kubectl get pods -l app=hostnames \  
-o go-template='{{range .items}}{{.status.podIP}}\n{{end}}
```

```
10.244.0.5  
10.244.0.6  
10.244.0.7
```

用于本教程的示例容器仅通过 HTTP 在端口 9376 上提供其自己的主机名，但是如果要调试自己的应用程序，则需要使用你的 Pod 正在侦听的端口号。

在 Pod 内运行：

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do  
  wget -qO- $ep  
done
```

输出类似这样：

```
hostnames-632524106-bbpiw  
hostnames-632524106-ly40y  
hostnames-632524106-tlaok
```

如果此时你没有收到期望的响应，则你的 Pod 状态可能不健康，或者可能没有在你认为正确的端口上进行监听。你可能会发现 kubectl logs 命令对于查看正在发生的事情很有用，或者你可能需要通过 kubectl exec 直接进入 Pod 中并从那里进行调试。

假设到目前为止一切都已按计划进行，那么你可以开始调查为何你的 Service 无法正常工作。

Service 是否存在？

细心的读者会注意到我们实际上尚未创建 Service -这是有意而为之。这一步有时会被遗忘，这是首先要检查的步骤。

那么，如果我尝试访问不存在的 Service 会怎样？假设你有另一个 Pod 通过名称匹配到 Service，你将得到类似结果：

```
wget -O- hostnames
```

```
Resolving hostnames (hostnames)... failed: Name or service not known.  
wget: unable to resolve host address 'hostnames'
```

首先要检查的是该 Service 是否真实存在：

```
kubectl get svc hostnames
```

```
No resources found.  
Error from server (NotFound): services "hostnames" not found
```

让我们创建 Service。和以前一样，在这次实践中 - 你可以在此处使用自己的 Service 的内容。

```
kubectl expose deployment hostnames --port=80 --target-port=9376
```

```
service/hostnames exposed
```

重新运行查询命令，确认没有问题：

```
kubectl get svc hostnames
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
hostnames	ClusterIP	10.0.1.175	<none>	80/TCP	5s

现在你知道了 Service 确实存在。

同前，此步骤效果与通过 YAML 方式启动 'Service' 一样：

```
apiVersion: v1
kind: Service
metadata:
  name: hostnames
spec:
  selector:
    app: hostnames
  ports:
    - name: default
      protocol: TCP
      port: 80
      targetPort: 9376
```

为了突出配置范围的完整性，你在此处创建的 Service 使用的端口号与 Pods 不同。对于许多真实的 Service，这些值可以是相同的。

Service 是否可通过 DNS 名字访问？

通常客户端通过 DNS 名称来匹配到 Service。

从相同命名空间下的 Pod 中运行以下命令：

```
nslookup hostnames
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name: hostnames
```

```
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

如果失败，那么你的 Pod 和 Service 可能位于不同的命名空间中，请尝试使用限定命名空间的名称（同样在 Pod 内运行）：

```
nslookup hostnames.default
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name: hostnames.default
```

```
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

如果成功，那么需要调整你的应用，使用跨命名空间的名称去访问它，或者在相同的命名空间中运行应用和 Service。如果仍然失败，请尝试一个完全限定的名称：

```
nslookup hostnames.default.svc.cluster.local
```

```
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name: hostnames.default.svc.cluster.local
```

```
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

注意这里的后缀："default.svc.cluster.local"。"default" 是我们正在操作的命名空间。"svc" 表示这是一个 Service。"cluster.local" 是你的集群域，在你自己的集群中可能会有所不同。

你也可以在集群中的节点上尝试此操作：

说明：10.0.0.10 是集群的 DNS 服务 IP，你的可能有所不同。

```
nslookup hostnames.default.svc.cluster.local 10.0.0.10
```

```
Server: 10.0.0.10
Address: 10.0.0.10#53
```

```
Name: hostnames.default.svc.cluster.local
Address: 10.0.1.175
```

如果你能够使用完全限定的名称查找，但不能使用相对名称，则需要检查你 Pod 中的 /etc/resolv.conf 文件是否正确。在 Pod 中运行以下命令：

```
cat /etc/resolv.conf
```

你应该可以看到类似这样的输出：

```
nameserver 10.0.0.10
search default.svc.cluster.local svc.cluster.local cluster.local example.com
options ndots:5
```

nameserver 行必须指示你的集群的 DNS Service，它是通过 --cluster-dns 标志传递到 kubelet 的。

search 行必须包含一个适当的后缀，以便查找 Service 名称。在本例中，它查找本地命名空间 (default.svc.cluster.local) 中的服务和所有命名空间 (svc.cluster.local) 中的服务，最后在集群 (cluster.local) 中查找服务的名称。根据你自己的安装情况，可能会有额外的记录（最多 6 条）。集群后缀是通过 --cluster-domain 标志传递给 kubelet 的。本文中，我们假定后缀是 "cluster.local"。你的集群配置可能不同，这种情况下，你应该在上面的所有命令中更改它。

options 行必须设置足够高的 ndots，以便 DNS 客户端库考虑搜索路径。在默认情况下，Kubernetes 将这个值设置为 5，这个值足够高，足以覆盖它生成的所有 DNS 名称。

是否存在 Service 能通过 DNS 名称访问？

如果上面的方式仍然失败，DNS 查找不到你需要的 Service，你可以后退一步，看看还有什么其它东西没有正常工作。Kubernetes 主 Service 应该一直是工作的。在 Pod 中运行如下命令：

```
nslookup kubernetes.default
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

```
Name: kubernetes.default
Address 1: 10.0.0.1 kubernetes.default.svc.cluster.local
```

如果失败，你可能需要转到本文的 [kube-proxy](#) 节，或者甚至回到文档的顶部重新开始，但不是调试你自己的 Service，而是调试 DNS Service。

Service 能够通过 IP 访问么？

假设你已经确认 DNS 工作正常，那么接下来要测试的是你的 Service 能否通过它的 IP 正常访问。从集群中的一个 Pod，尝试访问 Service 的 IP（从上面的 `kubectl get` 命令获取）。

```
for i in $(seq 1 3); do
    wget -qO- 10.0.1.175:80
done
```

输出应该类似这样：

```
hostnames-632524106-bbpiw
hostnames-632524106-ly40y
hostnames-632524106-tlaok
```

如果 Service 状态是正常的，你应该得到正确的响应。如果没有，有很多可能出错的地方，请继续阅读。

Service 的配置是否正确？

这听起来可能很愚蠢，但你应该两次甚至三次检查你的 Service 配置是否正确，并且与你的 Pod 匹配。查看你的 Service 配置并验证它：

```
kubectl get service hostnames -o json
```

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "hostnames",
    "namespace": "default",
    "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
    "resourceVersion": "347189",
    "creationTimestamp": "2015-07-07T15:24:29Z",
    "labels": {
      "app": "hostnames"
    }
  },
}
```

```

"spec": {
  "ports": [
    {
      "name": "default",
      "protocol": "TCP",
      "port": 80,
      "targetPort": 9376,
      "nodePort": 0
    }
  ],
  "selector": {
    "app": "hostnames"
  },
  "clusterIP": "10.0.1.175",
  "type": "ClusterIP",
  "sessionAffinity": "None"
},
"status": {
  "loadBalancer": {}
}
}

```

- 你想要访问的 Service 端口是否在 spec.ports[] 中列出？
- targetPort 对你的 Pod 来说正确吗（许多 Pod 使用与 Service 不同的端口）？
- 如果你想使用数值型端口，那么它的类型是一个数值（9376）还是字符串“9376”？
- 如果你想使用名称型端口，那么你的 Pod 是否暴露了一个同名端口？
- 端口的 protocol 和 Pod 的是否对应？

Service 有 Endpoints 吗？

如果你已经走到了这一步，你已经确认你的 Service 被正确定义，并能通过 DNS 解析。现在，让我们检查一下，你运行的 Pod 确实是被 Service 选中的。

早些时候，我们已经看到 Pod 是运行状态。我们可以再检查一下：

```
kubectl get pods -l app=hostnames
```

NAME	READY	STATUS	RESTARTS	AGE
hostnames-632524106-bbpiw	1/1	Running	0	1h
hostnames-632524106-ly40y	1/1	Running	0	1h
hostnames-632524106-tlaok	1/1	Running	0	1h

-l app=hostnames 参数是一个标签选择算符 - 和我们 Service 中定义的一样。

"AGE" 列表明这些 Pod 已经启动一个小时了，这意味着它们运行良好，而未崩溃。

"RESTARTS" 列表明 Pod 没有经常崩溃或重启。经常性崩溃可能导致间歇性连接问题。如果重启次数过大，通过[调试 pod](#) 了解相关技术。

在 Kubernetes 系统中有一个控制回路，它评估每个 Service 的选择算符，并将结果保存到 Endpoints 对象中。

```
kubectl get endpoints hostnames
```

NAME	ENDPOINTS
hostnames	10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376

这证实 Endpoints 控制器已经为你的 Service 找到了正确的 Pods。如果 ENDPOINTS 列的值为 <none>，则应检查 Service 的 spec.selector 字段，以及你实际想选择的 Pod 的 metadata.labels 的值。常见的错误是输入错误或其他错误，例如 Service 想选择 app=hostnames，但是 Deployment 指定的是 run=hostnames。在 1.18 之前的版本中 kubectl run 也可以被用来创建 Deployment。

Pod 正常工作吗？

至此，你知道你的 Service 已存在，并且已匹配到你的 Pod。在本实验的开始，你已经检查了 Pod 本身。让我们再次检查 Pod 是否确实在工作 - 你可以绕过 Service 机制并直接转到 Pod，如上面的 Endpoint 所示。

说明：这些命令使用的是 Pod 端口（9376），而不是 Service 端口（80）。

在 Pod 中运行：

```
for ep in 10.244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376; do  
    wget -qO- $ep  
done
```

输出应该类似这样：

```
hostnames-632524106-bbpiw  
hostnames-632524106-ly40y  
hostnames-632524106-tlaok
```

你希望 Endpoint 列表中的每个 Pod 都返回自己的主机名。如果情况并非如此（或你自己的 Pod 的正确行为是什么），你应调查发生了什么事情。

kube-proxy 正常工作吗？

如果你到达这里，则说明你的 Service 正在运行，拥有 Endpoints，Pod 真正在提供服务。此时，整个 Service 代理机制是可疑的。让我们一步一步地确认它没问题。

Service 的默认实现（在大多数集群上应用的）是 kube-proxy。这是一个在每个节点上运行的程序，负责配置用于提供 Service 抽象的机制之一。如果你的集群不使用 kube-proxy，则以下各节将不适用，你将必须检查你正在使用的 Service 的实现方式。

kube-proxy 正常运行吗？

确认 kube-proxy 正在节点上运行。在节点上直接运行，你将会得到类似以下的输出：

```
ps auxw | grep kube-proxy
```

```
root 4194 0.4 0.1 101864 17696 ? Sl Jul04 25:43 /usr/local/bin/kube-proxy --master=https://kubernetes-master --kubeconfig=/var/lib/kube-proxy/kubeconfig --v=2
```

下一步，确认它并没有出现明显的失败，比如连接主节点失败。要做到这一点，你必须查看日志。访问日志的方式取决于你节点的操作系统。在某些操作系统上日志是一个文件，如 /var/log/messages kube-proxy.log，而其他操作系统使用 journalctl 访问日志。你应该看到输出类似于：

```
I1027 22:14:53.995134 5063 server.go:200] Running in resource-only container "/kube-proxy"
I1027 22:14:53.998163 5063 server.go:247] Using iptables Proxier.
I1027 22:14:53.999055 5063 server.go:255] Tearing down userspace rules. Errors here are acceptable.
I1027 22:14:54.038140 5063 proxier.go:352] Setting endpoints for "kubesystem/kube-dns:dns-tcp" to [10.244.1.3:53]
I1027 22:14:54.038164 5063 proxier.go:352] Setting endpoints for "kubesystem/kube-dns:dns" to [10.244.1.3:53]
I1027 22:14:54.038209 5063 proxier.go:352] Setting endpoints for "default/kubernetes:https" to [10.240.0.2:443]
I1027 22:14:54.038238 5063 proxier.go:429] Not syncing iptables until Services and Endpoints have been received from master
I1027 22:14:54.040048 5063 proxier.go:294] Adding new service "default/kubernetes:https" at 10.0.0.1:443/TCP
I1027 22:14:54.040154 5063 proxier.go:294] Adding new service "kubesystem/kube-dns:dns" at 10.0.0.10:53/UDP
I1027 22:14:54.040223 5063 proxier.go:294] Adding new service "kubesystem/kube-dns:dns-tcp" at 10.0.0.10:53/TCP
```

如果你看到有关无法连接主节点的错误消息，则应再次检查节点配置和安装步骤。

kube-proxy 无法正确运行的可能原因之一是找不到所需的 conntrack 二进制文件。在一些 Linux 系统上，这也是可能发生的，这取决于你如何安装集群，例如，你是手动开始一步步安装 Kubernetes。如果是这样的话，你需要手动安装 conntrack 包（例如，在 Ubuntu 上使用 sudo apt install conntrack），然后重试。

Kube-proxy 可以以若干模式之一运行。在上述日志中，Using iptables Proxier 表示 kube-proxy 在 "iptables" 模式下运行。最常见的另一种模式是 "ipvs"。先前的 "userspace" 模式已经被这些所代替。

Iptables 模式

在 "iptables" 模式中，你应该可以在节点上看到如下输出：

```
iptables-save | grep hostnames
```

```
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/
hostnames:" -m tcp -j DNAT --to-destination 10.244.3.6:9376
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment
"default/hostnames:" -m tcp -j DNAT --to-destination 10.244.1.7:9376
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment
"default/hostnames:" -m tcp -j DNAT --to-destination 10.244.2.3:9376
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment
"default/hostnames: cluster IP" -m tcp --dport 80 -j KUBE-SVC-
NWV5X2332I4OT4T3
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/
hostnames:" -m statistic --mode random --probability 0.33332999982 -j
KUBE-SEP-WNBA2IHDGP2BOBGZ
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/
hostnames:" -m statistic --mode random --probability 0.500000000000 -j
KUBE-SEP-X3P2623AGDH6CDF3
-A KUBE-SVC-NWV5X2332I4OT4T3 -m comment --comment "default/
hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

对于每个 Service 的每个端口，应有 1 条 KUBE-SERVICES 规则、一个 KUBE-SVC-<hash> 链。对于每个 Pod 末端，在那个 KUBE-SVC-<hash> 链中应该有一些规则与之对应，还应该有一个 KUBE-SEP-<hash> 链与之对应，其中包含为数不多的几条规则。实际的规则数量可能会根据你实际的配置（包括 NodePort 和 LoadBalancer 服务）有所不同。

IPVS 模式

在 "ipvs" 模式中，你应该在节点下看到如下输出：

```
ipvsadm -Ln
```

```
Prot LocalAddress:Port Scheduler Flags
-> RemoteAddress:Port      Forward Weight ActiveConn InActConn
...
TCP 10.0.1.175:80 rr
-> 10.244.0.5:9376      Masq    1      0      0
-> 10.244.0.6:9376      Masq    1      0      0
-> 10.244.0.7:9376      Masq    1      0      0
...
```

对于每个 Service 的每个端口，还有 NodePort，External IP 和 LoadBalancer 类型服务的 IP，kube-proxy 将创建一个虚拟服务器。对于每个 Pod 末端，它将创建相应的真实服务器。在此示例中，服务主机名（10.0.1.175:80）拥有 3 个末端（10.244.0.5:9376、10.244.0.6:9376 和 10.244.0.7:9376）。

Userspace 模式

在极少数情况下，你可能会用到 "userspace" 模式。在你的节点上运行：

```
iptables-save | grep hostnames
```

```
-A KUBE-PORTALS-CONTAINER -d 10.0.1.175/32 -p tcp -m comment --
comment "default/hostnames:default" -m tcp --dport 80 -j REDIRECT --to-
ports 48577
-A KUBE-PORTALS-HOST -d 10.0.1.175/32 -p tcp -m comment --comment
"default/hostnames:default" -m tcp --dport 80 -j DNAT --to-destination
10.240.115.247:48577
```

对于 Service（本例中只有一个）的每个端口，应当有 2 条规则：一条 "KUBE-PORTALS-CONTAINER" 和一条 "KUBE-PORTALS-HOST" 规则。

几乎没有人应该再使用 "userspace" 模式，因此你在这里不会花更多的时间。

kube-proxy 是否在运行？

假设你确实遇到上述情况之一，请重试从节点上通过 IP 访问你的 Service：

```
curl 10.0.1.175:80
```

```
hostnames-632524106-bbpiw
```

如果失败，并且你正在使用用户空间代理，则可以尝试直接访问代理。如果你使用的是 iptables 代理，请跳过本节。

回顾上面的 iptables-save 输出，并提取 kube-proxy 为你的 Service 所使用的端口号。在上面的例子中，端口号是 "48577"。现在试着连接它：

```
curl localhost:48577
```

```
hostnames-632524106-tlaok
```

如果这步操作仍然失败，请查看 kube-proxy 日志中的特定行，如：

```
Setting endpoints for default/hostnames:default to [10.244.0.5:9376  
10.244.0.6:9376 10.244.0.7:9376]
```

如果你没有看到这些，请尝试将 -V 标志设置为 4 并重新启动 kube-proxy，然后再查看日志。

边缘案例：Pod 无法通过 Service IP 连接到它本身

这听起来似乎不太可能，但是确实可能发生，并且应该可行。

如果网络没有为“发夹模式（Hairpin）”流量生成正确配置，通常当 kube-proxy 以 iptables 模式运行，并且 Pod 与桥接网络连接时，就会发生这种情况。kubeblet 提供了 hairpin-mode [标志](#)。如果 Service 的末端尝试访问自己的 Service VIP，则该端点可以把流量负载均衡回到它们自身。hairpin-mode 标志必须被设置为 hairpin-veth 或者 promiscuous-bridge。

诊断此类问题的常见步骤如下：

- 确认 hairpin-mode 被设置为 hairpin-veth 或 promiscuous-bridge。你应该可以看到下面这样。本例中 hairpin-mode 被设置为 promiscuous-bridge

```
ps auxw | grep kubelet
```

```
root    3392  1.1  0.8 186804 65208 ?    Sl  00:51 11:11 /usr/local/  
bin/kubelet --enable-debugging-handlers=true --config=/etc/  
kubernetes/manifests --allow-privileged=True --v=4 --cluster-  
dns=10.0.0.10 --cluster-domain=cluster.local --configure-cbr0=true --  
cgroup-root=/ --system-cgroups=/system --hairpin-  
mode=promiscuous-bridge --runtime-cgroups=/docker-daemon --  
kubelet-cgroups=/kubelet --babysit-daemons=true --max-pods=110  
--serialize-image-pulls=false --outofdisk-transition-frequency=0
```

- 确认有效的 hairpin-mode。要做到这一点，你必须查看 kubelet 日志。访问日志取决于节点的操作系统。在一些操作系统上，它是一个文件，如 /var/log/kubelet.log，而其他操作系统则使用 journalctl 访问日志。请注意，由于兼容性，有效的 hairpin-mode 可能不匹配 --hairpin-mode 标志。在 kubelet.log 中检查是否有带有关键字 hairpin 的日志行。应该有日志行指示有效的 hairpin-mode，就像下面这样。

```
I0629 00:51:43.648698  3252 kubelet.go:380] Hairpin mode set to  
"promiscuous-bridge"
```

- 如果有效的发夹模式是 hairpin-veth，要保证 Kubelet 有操作节点上 /sys 的权限。如果一切正常，你将会看到如下输出：

```
for intf in /sys/devices/virtual/net/cbr0/brif/*; do cat $intf/
hairpin_mode; done
```

```
1
1
1
1
```

- 如果有效的发卡模式是 `promiscuous-bridge`, 要保证 Kubelet 有操作节点上 Linux 网桥的权限。如果 `cbr0` 桥正在被使用且被正确设置，你将会看到如下输出:

```
ifconfig cbr0 |grep PROMISC
```

```
UP BROADCAST RUNNING PROMISC MULTICAST MTU:1460 Metric:1
```

- 如果以上步骤都不能解决问题，请寻求帮助。

寻求帮助

如果你走到这一步，那么就真的是奇怪的事情发生了。你的 Service 正在运行，有 Endpoints 存在，你的 Pods 也确实在提供服务。你的 DNS 正常，iptables 规则已经安装，`kube-proxy` 看起来也正常。然而 Service 还是没有正常工作。这种情况下，请告诉我们，以便我们可以帮助调查！

通过 [Slack](#) 或者 [Forum](#) 或者 [GitHub](#) 联系我们。

接下来

访问[故障排查文档](#) 获取更多信息。

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 December 15, 2020 at 9:56 PM PST: [Update debug-service.md \(7de0e8f25\)](#)

调试StatefulSet

此任务展示如何调试 StatefulSet。

准备开始

- 你需要有一个 Kubernetes 集群，已配置好的 kubectl 命令行工具与你的集群进行通信。
- 你应该有一个运行中的 StatefulSet，以便用于调试。

调试 StatefulSet

StatefulSet 在创建 Pod 时为其设置了 `app=myapp` 标签，列出仅属于某 StatefulSet 的所有 Pod 时，可以使用以下命令：

```
kubectl get pods -l app=myapp
```

如果你发现列出的任何 Pod 长时间处于 Unknown 或 Terminating 状态，请参阅 [删除 StatefulSet Pods](#) 了解如何处理它们的说明。你可以参考[调试 Pods](#) 来调试 StatefulSet 中的各个 Pod。

接下来

- 进一步了解如何[调试 Init 容器](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 15, 2020 at 9:34 PM PST: [Update debug-stateful-set.md \(1f6a8b0d5\)](#)

调试运行中的 Pod

本页解释如何在节点上调试运行中（或崩溃）的 Pod。

准备开始

- 你的 [Pod](#) 应该已经被调度并正在运行中，如果你的 Pod 还没有运行，请参阅 [应用问题排查](#)。
- 对于一些高级调试步骤，你应该知道 Pod 具体运行在哪个节点上，在该节点上有权限去运行一些命令。你不需要任何访问权限就可以使用 kubectl 去运行一些标准调试步骤。

检查 Pod 的日志

首先，查看受到影响的容器的日志：

```
kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

如果你的容器之前崩溃过，你可以通过下面命令访问之前容器的崩溃日志：

```
kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

使用容器 exec 进行调试

如果 [容器镜像](#) 包含调试程序，比如从 Linux 和 Windows 操作系统基础镜像构建的镜像，你可以使用 kubectl exec 命令 在特定的容器中运行一些命令：

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}
```

说明： -c \${CONTAINER_NAME} 是可选择的。如果Pod中仅包含一个容器，就可以忽略它。

例如，要查看正在运行的 Cassandra pod中的日志，可以运行：

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

你可以在 kubectl exec 命令后面加上 -i 和 -t 来运行一个连接到你的终端的 Shell，比如：

```
kubectl exec -it cassandra -- sh
```

若要了解更多内容，可查看[获取正在运行容器的 Shell](#)。

使用临时调试容器来进行调试

FEATURE STATE: Kubernetes v1.18 [alpha]

当由于容器崩溃或容器镜像不包含调试程序（例如[无发行版镜像](#)等）而导致 kubectl exec 无法运行时，[临时容器](#)对于排除交互式故障很有用。从 'v1.18' 版本开始，'kubectl' 有一个可以创建用于调试的临时容器的 alpha 命令。

使用临时容器来调试的例子

说明： 本示例需要你的集群已经开启 EphemeralContainers [特性门控](#)，kubectl 版本为 v1.18 或者更高。

你可以使用 kubectl alpha debug 命令来给正在运行中的 Pod 增加一个临时容器。首先，像示例一样创建一个 pod：

```
kubectl run ephemeral-demo --image=k8s.gcr.io/pause:3.1 --restart=Never
```

说明：本节示例中使用 pause 容器镜像，因为它不包含任何用户级调试程序，但是这个方法适用于所有容器镜像。

如果你尝试使用 kubectl exec 来创建一个 shell，你将会看到一个错误，因为这个容器镜像中没有 shell。

```
kubectl exec -it ephemeral-demo -- sh
```

```
OCI runtime exec failed: exec failed: container_linux.go:346: starting
container process caused "exec: \"sh\": executable file not found in $PATH":
unknown
```

你可以改为使用 kubectl alpha debug 添加调试容器。如果你指定 -i 或者 --interactive 参数，kubectl 将自动挂接到临时容器的控制台。

```
kubectl alpha debug -it ephemeral-demo --image=busybox --target=ephemeral-demo
```

```
Defaulting debug container name to debugger-8xzrl.
If you don't see a command prompt, try pressing enter.
/ #
```

此命令添加一个新的 busybox 容器并将其挂接到该容器。--target 参数指定另一个容器的进程命名空间。这是必需的，因为 kubectl run 不能在它创建的 pod 中启用 [共享进程命名空间](#)。

说明：容器运行时必须支持--target参数。如果不支持，则临时容器可能不会启动，或者可能使用隔离的进程命名空间启动。

你可以使用 kubectl describe 查看新创建的临时容器的状态：

```
kubectl describe pod ephemeral-demo
```

```
...
Ephemeral Containers:
  debugger-8xzrl:
    Container ID: docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4028ffb2eb
    Image:      busybox
    Image ID:   docker-pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f07a21d1c08084
    Port:       <none>
    Host Port: <none>
    State:     Running
    Started:   Wed, 12 Feb 2020 14:25:42 +0100
    Ready:     False
```

```
Restart Count: 0
Environment:  <none>
Mounts:       <none>
...

```

使用 kubectl delete 来移除已经结束掉的 Pod :

```
kubectl delete pod ephemeral-demo
```

通过 Pod 副本调试

有些时候 Pod 的配置参数使得在某些情况下很难执行故障排查。例如，在容器镜像中不包含 shell 或者你的应用程序在启动时崩溃的情况下，就不能通过运行 kubectl exec 来排查容器故障。在这些情况下，你可以使用 kubectl debug 来创建 Pod 的副本，通过更改配置帮助调试。

在添加新的容器时创建 Pod 副本

当应用程序正在运行但其表现不符合预期时，你会希望在 Pod 中添加额外的调试工具，这时添加新容器是很有用的。

例如，应用的容器镜像是建立在 busybox 的基础上，但是你需要 busybox 中并不包含的调试工具。你可以使用 kubectl run 模拟这个场景：

```
kubectl run myapp --image=busybox --restart=Never -- sleep 1d
```

通过运行以下命令，建立 myapp 的一个名为 myapp-debug 的副本，新增了一个用于调试的 Ubuntu 容器，

```
kubectl debug myapp -it --image=ubuntu --share-processes --copy-to=myapp-debug
```

```
Defaulting debug container name to debugger-w7xmf.
```

```
If you don't see a command prompt, try pressing enter.
```

```
root@myapp-debug:/#
```

说明：

- 如果你没有使用 --container 指定新的容器名，kubectl debug 会自动生成的。
- 默认情况下，-i 标志使 kubectl debug 附加到新容器上。你可以通过指定 --attach=false 来防止这种情况。如果你的会话断开连接，你可以使用 kubectl attach 重新连接。
- share-processes 允许在此 Pod 中的其他容器中查看该容器的进程。参阅[在 Pod 中的容器之间共享进程命名空间](#) 获取更多信息。

不要忘了清理调试 Pod：

```
kubectl delete pod myapp myapp-debug
```

在改变 Pod 命令时创建 Pod 副本

有时更改容器的命令很有用，例如添加调试标志或因为应用崩溃。

为了模拟应用崩溃的场景，使用 `kubectl run` 命令创建一个立即退出的容器：

```
kubectl run --image=busybox myapp -- false
```

使用 `kubectl describe pod myapp` 命令，你可以看到容器崩溃了：

Containers:

myapp:

 Image: busybox

 ...

 Args:

 false

 State: Waiting

 Reason: CrashLoopBackOff

 Last State: Terminated

 Reason: Error

 Exit Code: 1

你可以使用 `kubectl debug` 命令创建该 Pod 的一个副本，在该副本中命令改变为交互式 shell：

```
kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh
```

If you don't see a command prompt, try pressing enter.

```
/ #
```

现在你有了一个可以执行类似检查文件系统路径或者手动运行容器命令的交互式 shell。

说明：

- 要更改指定容器的命令，你必须用 `--container` 命令指定容器的名字，否则 `kubectl debug` 将建立一个新的容器运行你指定的命令。
- 默认情况下，标志 `-i` 使 `kubectl debug` 附加到容器。你可通过指定 `--attach=false` 来防止这种情况。如果你的断开连接，可以使用 `kubectl attach` 重新连接。

不要忘了清理调试 Pod：

```
kubectl delete pod myapp myapp-debug
```

在更改容器镜像时创建 Pod 副本

在某些情况下，你可能想从正常生产容器镜像中 把行为异常的 Pod 改变为包含调试版本或者附加应用的镜像。

下面的例子，用 kubectl run 创建一个 Pod：

```
kubectl run myapp --image=busybox --restart=Never -- sleep 1d
```

现在可以使用 kubectl debug 创建一个副本 并改变容器镜像为 ubuntu：

```
kubectl debug myapp --copy-to=myapp-debug --set-image=*=ubuntu
```

--set-image 与 container_name=image 使用相同的 kubectl set image 语法。
*=ubuntu 表示把所有容器的镜像改为 ubuntu。

```
kubectl delete pod myapp myapp-debug
```

在节点上通过 shell 来调试

如果这些方法都不起作用，你可以找到运行 Pod 的主机并通过 SSH 进入该主机，但是如果使用 Kubernetes API 中的工具，则通常不需要这样做。因此，如果你发现自己需要使用 ssh 进入主机，请在 GitHub 上提交功能请求，以描述你的用例以及这些工具不足的原因。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 07, 2021 at 10:18 AM PST: [Fix pod name \(fc159daa1\)](#)

资源指标管道

资源使用指标，例如容器 CPU 和内存使用率，可通过 Metrics API 在 Kubernetes 中获得。这些指标可以直接被用户访问，比如使用 kubectl top 命令行，或者被集群中的控制器（例如 Horizontal Pod Autoscalers）使用来做决策。

Metrics API

通过 Metrics API，你可以获得指定节点或 Pod 当前使用的资源量。此 API 不存储指标值，因此想要获取某个指定节点 10 分钟前的 资源使用量是不可能的。

此 API 与其他 API 没有区别：

- 此 API 和其它 Kubernetes API 一起位于同一端点 (endpoint) 之下且可发现，路径为 /apis/metrics.k8s.io/

- 它具有相同的安全性、可扩展性和可靠性保证

Metrics API 在 k8s.io/metrics 仓库中定义。你可以在那里找到有关 Metrics API 的更多信息。

说明： Metrics API 需要在集群中部署 Metrics Server。否则它将不可用。

度量资源用量

CPU

CPU 用量按其一段时间内的平均值统计，单位为 [CPU 核](#)。此度量值通过在内核（包括 Linux 和 Windows）提供的累积 CPU 计数器乘以一个系数得到。kubel et 组件负责选择计算系数所使用的窗口大小。

内存

内存用量按工作集（Working Set）的大小字节数统计，其数值为收集度量值的那一刻的内存用量。如果一切都很理想化，“工作集”是任务在使用的内存总量，该内存是不可以在内存压力较大的情况下被释放的。不过，具体的工作集计算方式取决于宿主 OS，有很大不同，且通常都大量使用启发式规则来给出一个估计值。其中包含所有匿名内存使用（没有后台文件提供存储者），因为 Kubernetes 不支持交换分区。度量值通常包含一些高速缓存（有后台文件提供存储）内存，因为宿主操作系统并不是总能回收这些页面。

Metrics 服务器

[Metrics 服务器](#) 是集群范围资源用量数据的聚合器。默认情况下，在由 kube-up.sh 脚本创建的集群中会以 Deployment 的形式被部署。如果你使用其他 Kubernetes 安装方法，则可以使用提供的 [部署组件 components.yaml](#) 来部署。

Metric 服务器从每个节点上的 [kubelet](#) 公开的 Summary API 中采集指标信息。该 API 通过 [Kubernetes 聚合器](#) 注册到主 API 服务器上。

在[设计文档](#) 中可以了解到有关 Metrics 服务器的更多信息。

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 December 08, 2020 at 10:16 PM PST: [Fix broken link to metrics-server \(a67a087bf\)](#)

资源监控工具

要扩展应用程序并提供可靠的服务，你需要了解应用程序在部署时的行为。你可以通过检测容器检查 Kubernetes 集群中的应用程序性能，[Pods](#), [服务](#) 和整个集群的特征。Kubernetes 在每个级别上提供有关应用程序资源使用情况的详细信息。此信息使你可以评估应用程序的性能，以及在何处可以消除瓶颈以提高整体性能。

在 Kubernetes 中，应用程序监控不依赖单个监控解决方案。在新集群上，你可以使用[资源度量](#)或[完整度量](#)管道来收集监视统计信息。

资源度量管道

资源指标管道提供了一组与集群组件，例如[Horizontal Pod Autoscaler](#)控制器以及 kubectl top 实用程序相关的有限度量。这些指标是由轻量级的、短期、内存存储的[metrics-server](#)收集的，通过 metrics.k8s.io 公开。

度量服务器发现集群中的所有节点，并且查询每个节点的[kubelet](#)以获取 CPU 和内存使用情况。Kubelet 充当 Kubernetes 主节点与节点之间的桥梁，管理机器上运行的 Pod 和容器。kubelet 将每个 Pod 转换为其组成的容器，并在容器运行时通过容器运行时接口 获取各个容器使用情况统计信息。kubelet 从集成的 cAdvisor 获取此信息，以进行旧式 Docker 集成。然后，它通过 metrics-server Resource Metrics API 公开聚合的 pod 资源使用情况统计信息。该 API 在 kubelet 的经过身份验证和只读的端口上的 /metrics/resource/v1beta1 中提供。

完整度量管道

一个完整度量管道可以让你访问更丰富的度量。Kubernetes 还可以根据集群的当前状态，使用 Pod 水平自动伸缩器等机制，通过自动调用扩展或调整集群来响应这些度量。监控管道从 kubelet 获取度量值，然后通过适配器将它们公开给 Kubernetes，方法是实现 custom.metrics.k8s.io 或 external.metrics.k8s.io API。

[Prometheus](#) 是一个 CNCF 项目，可以原生监控 Kubernetes、节点和 Prometheus 本身。完整度量管道项目不属于 CNCF 的一部分，不在 Kubernetes 文档的范围之内。

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 23, 2020 at 1:10 PM PST: [\[zh\] Sync changes from English site \(12\) \(81bc52053\)](#)

集群故障排查

本篇文档是介绍集群故障排查的；我们假设对于你碰到的问题，你已经排除了是由应用程序造成的。对于应用的调试，请参阅 [应用故障排查指南](#)。你也可以访问[故障排查](#)来获取更多的信息。

列举集群节点

调试的第一步是查看所有的节点是否都已正确注册。

运行

```
kubectl get nodes
```

验证你所希望看见的所有节点都能够显示出来，并且都处于 Ready 状态。

为了了解你的集群的总体健康状况详情，你可以运行：

```
kubectl cluster-info dump
```

查看日志

到这里，挖掘出集群更深层的信息就需要登录到相关的机器上。下面是相关日志文件所在的位置。（注意，对于基于 systemd 的系统，你可能需要使用journalctl）。

主控节点

- /var/log/kube-apiserver.log - API 服务器, 提供API服务
- /var/log/kube-scheduler.log - 调度器, 负责产生调度决策
- /var/log/kube-controller-manager.log - 管理副本控制器的控制器

工作节点

- /var/log/kubelet.log - kubelet, 负责在节点运行容器
- /var/log/kube-proxy.log - kube-proxy, 负责服务的负载均衡

集群故障模式的一般性概述

下面是一个不完整的列表，列举了一些可能的出错场景，以及通过调整集群配置来解决相关问题的方法。

根本原因

- VM(s) 关机
- 集群之间，或者集群和用户之间网络分裂
- Kubernetes 软件本身崩溃
- 数据丢失或者持久化存储不可用（如：GCE PD 或 AWS EBS 卷）
- 操作错误，如：Kubernetes 或者应用程序配置错误

具体情况：

- API 服务器所在的 VM 关机或者 API 服务器崩溃
 - 结果
 - 不能停止、更新或者启动新的 Pod、服务或副本控制器
 - 现有的 Pod 和服务在不依赖 Kubernetes API 的情况下应该能继续正常工作
- API 服务器的后端存储丢失
 - 结果
 - API 服务器应该不能启动
 - kubelet 将不能访问 API 服务器，但是能够继续运行之前的 Pod 和提供相同的服务代理
 - 在 API 服务器重启之前，需要手动恢复或者重建 API 服务器的状态
- Kubernetes 服务组件（节点控制器、副本控制器管理器、调度器等）所在的 VM 关机或者崩溃
 - 当前，这些控制器是和 API 服务器在一起运行的，它们不可用的现象是与 API 服务器类似的
 - 将来，这些控制器也会复制为多份，并且可能不在运行于同一节点上
 - 它们没有自己的持久状态
- 单个节点（VM 或者物理机）关机
 - 结果
 - 此节点上的所有 Pod 都停止运行
- 网络分裂
 - 结果
 - 分区 A 认为分区 B 中所有的节点都已宕机；分区 B 认为 API 服务器宕机（假定主控节点所在的 VM 位于分区 A 内）。
- kubelet 软件故障
 - 结果
 - 崩溃的 kubelet 就不能在其所在的节点上启动新的 Pod
 - kubelet 可能删掉 Pod 或者不删
 - 节点被标识为非健康态
 - 副本控制器会在其它的节点上启动新的 Pod
- 集群操作错误
 - 结果
 - 丢失 Pod 或服务等等

- 丢失 API 服务器的后端存储
- 用户无法读取 API
- 等等

缓解措施：

- 措施：对于 IaaS 上的 VMs，使用 IaaS 的自动 VM 重启功能
 - 缓解：API 服务器 VM 关机或 API 服务器崩溃
 - 缓解：Kubernetes 服务组件所在的 VM 关机或崩溃
- 措施：对于运行 API 服务器和 etcd 的 VM，使用 IaaS 提供的可靠的存储（例如 GCE PD 或者 AWS EBS 卷）
 - 缓解：API 服务器后端存储的丢失
- 措施：使用高可用性的配置
 - 缓解：主控节点 VM 关机或者主控节点组件（调度器、API 服务器、控制器管理器）崩溃
 - 将容许一个或多个节点或组件同时出现故障
 - 缓解：API 服务器后端存储（例如 etcd 的数据目录）丢失
 - 假定你使用了高可用的 etcd 配置
- 措施：定期对 API 服务器的 PDs/EBS 卷执行快照操作
 - 缓解：API 服务器后端存储丢失
 - 缓解：一些操作错误的场景
 - 缓解：一些 Kubernetes 软件本身故障的场景
- 措施：在 Pod 的前面使用副本控制器或服务
 - 缓解：节点关机
 - 缓解：kubelet 软件故障
- 措施：应用（容器）设计成容许异常重启
 - 缓解：节点关机
 - 缓解：kubelet 软件故障

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 23, 2020 at 1:10 PM PST: [\[zh\] Sync changes from English site \(12\) \(81bc52053\)](#)

扩展 Kubernetes

了解针对工作环境需要来调整 Kubernetes 集群的进阶方法。

[使用自定义资源](#)

[配置聚合层](#)

[安装一个扩展的 API server](#)

[配置多个调度器](#)

[使用 HTTP 代理访问 Kubernetes API](#)

[设置 Konnectivity 服务](#)

使用自定义资源

[使用 CustomResourceDefinition 扩展 Kubernetes API](#)

[CustomResourceDefinition 的版本](#)

使用 CustomResourceDefinition 扩展 Kubernetes API

本页展示如何使用 [CustomResourceDefinition](#) 将 [定制资源 \(Custom Resource\)](#) 安装到 Kubernetes API 上。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 1.16. 要获知版本信息，请输入 kubectl version.

如果你在使用较老的、仍处于被支持范围的 Kubernetes 版本，请切换到该版本的文档查看对于的集群而言有用的建议。

创建 CustomResourceDefinition

当你创建新的 CustomResourceDefinition (CRD) 时，Kubernetes API 服务器会为你所指定的每一个版本生成一个 RESTful 的资源路径。CRD 可以是名字空间作用域的，也可以是集群作用域的，取决于 CRD 的 scope 字段设置。和其他现有的内置对象一样，删除一个名字空间时，该名字空间下的所有定制对象也会被删除。CustomResourceDefinition 本身是不受名字空间限制的，对所有名字空间可用。

例如，如果你将下面的 CustomResourceDefinition 保存到 `resourcedefinition.yaml` 文件：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # 名字必需与下面的 spec 字段匹配，并且格式为 '<名称的复数形式>.<组名>'
  name: crontabs.stable.example.com
spec:
  # 组名称，用于 REST API: /apis/<组>/<版本>
  group: stable.example.com
  # 列举此 CustomResourceDefinition 所支持的版本
  versions:
    - name: v1
      # 每个版本都可以通过 served 标志来独立启用或禁止
      served: true
      # 其中一个且只有一个版本必需被标记为存储版本
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  # 可以是 Namespaced 或 Cluster
  scope: Namespaced
  names:
```

```
# 名称的复数形式，用于 URL : /apis/<组>/<版本>/<名称的复数形式>
plural: crontabs
# 名称的单数形式，作为命令行使用时和显示时的别名
singular: crontab
# kind 通常是单数形式的驼峰编码 ( CamelCased ) 形式。你的资源清单会使用这一形式。
kind: CronTab
# shortNames 允许你在命令行使用较短的字符串来匹配资源
shortNames:
- ct
```

之后创建它：

```
kubectl apply -f resourcedefinition.yaml
```

这样一个新的受名字空间约束的 RESTful API 端点会被创建在：

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

此端点 URL 自此可以用来创建和管理定制对象。对象的 kind 将是来自你上面创建时 所用的 spec 中指定的 CronTab。

创建端点的操作可能需要几秒钟。你可以监测你的 CustomResourceDefinition 的 Established 状况变为 true，或者监测 API 服务器的发现信息等待你的资源出现在那里。

创建定制对象

在创建了 CustomResourceDefinition 对象之后，你可以创建定制对象 (Custom Objects)。定制对象可以包含定制字段。这些字段可以包含任意的 JSON 数据。在下面的例子中，在类别为 CronTab 的定制对象中，设置了 cronSpec 和 image 定制字段。类别 CronTab 来自你在上面所创建的 CRD 的规约。

如果你将下面的 YAML 保存到 my-crontab.yaml：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

并执行创建命令：

```
kubectl apply -f my-crontab.yaml
```

你就可以使用 kubectl 来管理你的 CronTab 对象了。例如：

```
kubectl get crontab
```

应该会输出如下列表：

NAME	AGE
my-new-cron-object	6s

使用 kubectl 时，资源名称是大小写不敏感的，而且你既可以使用 CRD 中所定义的单数形式或复数形式，也可以使用其短名称：

```
kubectl get ct -o yaml
```

你可以看到输出中包含了你创建定制对象时在 YAML 文件中指定的定制字段 cronSpec 和 image：

```
apiVersion: v1
kind: List
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    creationTimestamp: 2017-05-31T12:56:35Z
    generation: 1
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * /5'
    image: my-awesome-cron-image
  metadata:
    resourceVersion: ""
```

删除 CustomResourceDefinition

当你删除某 CustomResourceDefinition 时，服务器会卸载其 RESTful API 端点，并删除服务器上存储的所有定制对象。

```
kubectl delete -f resourcedefinition.yaml
kubectl get crontabs
```

```
Error from server (NotFound): Unable to list {"stable.example.com" "v1"
"crontabs"}: the server could not find the requested resource (get
crontabs.stable.example.com)
```

如果你在以后创建相同的 CustomResourceDefinition 时，该 CRD 会是一个空的结构。

设置结构化的模式

CustomResource 对象在定制字段中保存结构化的数据，这些字段和内置的字段 apiVersion、kind 和 metadata 等一起存储，不过内置的字段都会被 API 服务器隐式完成合法性检查。有了 [OpenAPI v3.0 检查](#) 能力之后，你可以设置一个模式（ Schema ），在创建和更新定制对象时，这一模式会被用来 对对象内容进行合法性检查。参阅下文了解这类模式的细节和局限性。

在 apiextensions.k8s.io/v1 版本中，CustomResourceDefinition 的这一结构化模式 定义是必需的。在 CustomResourceDefinition 的 beta 版本中，结构化模式定义是可选的。

结构化模式本身是一个 [OpenAPI v3.0 验证模式](#)，其中：

1. 为对象根（ root ）设置一个非空的 type 值（藉由 OpenAPI 中的 type ），对每个 object 节点的每个字段（藉由 OpenAPI 中的 properties 或 additionalProperties ）以及 array 节点的每个条目（藉由 OpenAPI 中的 items ）也要设置非空的 type 值，除非：
 - 节点包含属性 x-kubernetes-int-or-string: true
 - 节点包含属性 x-kubernetes-preserve-unknown-fields: true
2. 对于 object 的每个字段或 array 中的每个条目，如果其定义中包含 allOf、anyOf、oneOf 或 not，则模式也要指定这些逻辑组合之外的字段或条目（试比较例 1 和例 2）。
3. 在 allOf、anyOf、oneOf 或 not 上下文内不设置 description、type、default、additionalProperties 或者 nullable。此规则的例外是 x-kubernetes-int-or-string 的两种模式（见下文）。
4. 如果 metadata 被设置，则只允许对 metadata.name 和 metadata.generateName 设置约束。

非结构化的例 1：

```
allOf:  
- properties:  
  foo:  
    ...  
...
```

违反了第 2 条规则。下面的是正确的：

```
properties:  
  foo:  
    ...  
allOf:  
- properties:  
  foo:  
    ...  
...
```

非结构化的例 2：

```
allOf:  
- items:  
  properties:  
    foo:  
    ...  
  ...
```

违反了第 2 条规则。下面的是正确的：

```
items:  
  properties:  
    foo:  
    ...  
allOf:  
- items:  
  properties:  
    foo:  
  ...
```

非结构化的例 3：

```
properties:  
  foo:  
    pattern: "abc"  
  metadata:  
    type: object  
  properties:  
    name:  
      type: string  
      pattern: "^a"  
    finalizers:  
      type: array  
      items:  
        type: string  
        pattern: "my-finalizer"  
anyOf:  
- properties:  
  bar:  
    type: integer  
    minimum: 42  
  required: ["bar"]  
  description: "foo bar object"
```

不是一个结构化的模式，因为其中存在以下违例：

- 根节点缺失 type 设置（规则 1）
- foo 的 type 缺失（规则 1）
- anyOf 中的 bar 未在外部指定（规则 2）

- bar 的 type 位于 anyOf 中 (规则 3)
- anyOf 中设置了 description (规则 3)
- metadata.finalizers 不可以被限制 (规则 4)

作为对比，下面的 YAML 所对应的模式则是结构化的：

```

type: object
description: "foo bar object"
properties:
  foo:
    type: string
    pattern: "abc"
  bar:
    type: integer
  metadata:
    type: object
    properties:
      name:
        type: string
        pattern: "^\w+"
anyOf:
- properties:
  bar:
    minimum: 42
required: ["bar"]

```

如果违反了结构化模式规则，CustomResourceDefinition 的 NonStructural 状况中会包含报告信息。

字段剪裁

CustomResourceDefinition 在集群的持久性存储 [etcd](#) 中保存经过合法性检查的资源数据。就像原生的 Kubernetes 资源，例如 [ConfigMap](#)，如果你指定了 API 服务器所无法识别的字段，则该未知字段会在保存资源之前被剪裁 (*Pruned*) 掉 (删除)。

说明：

从 `apiextensions.k8s.io/v1beta1` 转换到 `apiextensions.k8s.io/v1` 的 CRD 可能没有结构化的模式定义，因此其 `spec.preserveUnknownFields` 可能为 `true`。

对于迁移而来的 CustomResourceDefinition，如果其 `spec.preserveUnknownFields` 被设置为 `true`，则 Kubernetes 不会执行剪裁操作，你可以存储任意数据。要实现最佳的兼容性，你应该更新定制资源以满足某 OpenAPI 模式定义，并且你应该将 CustomResourceDefinition 自身的 `spec.preserveUnknownFields` 设置为 `true`。

如果你将下面的 YAML 保存到 my-crontab.yaml 文件：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
  someRandomField: 42
```

并创建之：

```
kubectl create --validate=false -f my-crontab.yaml -o yaml
```

输出类似于：

```
apiVersion: stable.example.com/v1
kind: CronTab
metadata:
  creationTimestamp: 2017-05-31T12:56:35Z
  generation: 1
  name: my-new-cron-object
  namespace: default
  resourceVersion: "285"
  uid: 9423255b-4600-11e7-af6a-28d2447dc82b
spec:
  cronSpec: '* * * * */5'
  image: my-awesome-cron-image
```

注意其中的字段 someRandomField 已经被剪裁掉。

本例中通过 --validate=false 命令行选项 关闭了客户端的合法性检查以展示 API 服务器的行为，因为 [OpenAPI 合法性检查模式也会发布到](#) 客户端，kubectl 也会检查未知的字段并在对象被发送到 API 服务器之前就拒绝它们。

控制剪裁

默认情况下，定制资源的所有版本中的所有未规定的字段都会被剪裁掉。 通过在结构化的 OpenAPI v3 [检查模式定义](#) 中为特定字段的子树添加 x-kubernetes-preserve-unknown-fields: true 属性，可以选择不对其执行剪裁操作。 例如：

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
```

字段 json 可以保存任何 JSON 值，其中内容不会被剪裁掉。

你也可以部分地指定允许的 JSON 数据格式；例如：

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
    description: this is arbitrary JSON
```

通过这样设置，JSON 中只能设置 object 类型的值。

对于所指定的每个属性（或 additionalProperties），剪裁会再次被启用。

```
type: object
properties:
  json:
    x-kubernetes-preserve-unknown-fields: true
    type: object
    properties:
      spec:
        type: object
        properties:
          foo:
            type: string
          bar:
            type: string
```

对于上述定义，如果提供的数值如下：

```
json:
  spec:
    foo: abc
    bar: def
    something: x
  status:
    something: x
```

则该值会被剪裁为：

```
json:
  spec:
    foo: abc
    bar: def
  status:
    something: x
```

这意味着所指定的 spec 对象中的 something 字段被剪裁掉，而其外部的内容都被保留。

IntOrString

模式定义中标记了 `x-kubernetes-int-or-string: true` 的节点不受前述规则 1 约束，因此下面的定义是结构化的模式：

```
type: object
properties:
  foo:
    x-kubernetes-int-or-string: true
```

此外，所有这类节点也不再受规则 3 约束，也就是说，下面两种模式是被允许的（注意，仅限于这两种模式，不支持添加新字段的任何其他变种）：

```
x-kubernetes-int-or-string: true
anyOf:
- type: integer
- type: string
...
```

和

```
x-kubernetes-int-or-string: true
allOf:
- anyOf:
  - type: integer
  - type: string
- ... # zero or more
...
```

在以上两种规约中，整数值和字符串值都会被认为是合法的。

在[合法性检查模式定义的发布时](#)，`x-kubernetes-int-or-string: true` 会被展开为上述两种模式之一。

RawExtension

RawExtensions（就像在[k8s.io/apimachinery](#) 项目中 `runtime.RawExtension` 所定义的那样）可以保存完整的 Kubernetes 对象，也就是，其中会包含 `apiVersion` 和 `kind` 字段。

通过 `x-kubernetes-embedded-resource: true` 来设定这些嵌套对象的规约（无论是完全无限制还是部分指定都可以）是可能的。例如：

```
type: object
properties:
  foo:
    x-kubernetes-embedded-resource: true
    x-kubernetes-preserve-unknown-fields: true
```

这里，字段 foo 包含一个完整的对象，例如：

```
foo:  
  apiVersion: v1  
  kind: Pod  
  spec:  
    ...
```

由于字段上设置了 `x-kubernetes-preserve-unknown-fields: true`，其中的内容不会被剪裁。不过，在这个语境中，`x-kubernetes-preserve-unknown-fields: true` 的使用是可选的。

设置了 `x-kubernetes-embedded-resource: true` 之后，`apiVersion`、`kind` 和 `metadata` 都是隐式设定并隐式完成合法性验证。

提供 CRD 的多个版本

关于如何为你的 CustomResourceDefinition 提供多个版本的支持，以及如何将你的对象从一个版本迁移到另一个版本，详细信息可参阅 [定制资源定义的版本](#)。

高级主题

Finalizers

Finalizer 能够让控制器实现异步的删除前（Pre-delete）回调。定制对象和内置对象一样支持 Finalizer。

你可以像下面一样为定制对象添加 Finalizer：

```
apiVersion: "stable.example.com/v1"  
kind: CronTab  
metadata:  
  finalizers:  
    - stable.example.com/finalizer
```

自定义 Finalizer 的标识符包含一个域名、一个正向斜线和 finalizer 的名称。任何控制器都可以在任何对象的 finalizer 列表中添加新的 finalizer。

对带有 Finalizer 的对象的第一个删除请求会为其 `metadata.deletionTimestamp` 设置一个值，但不会真的删除对象。一旦此值被设置，finalizers 列表中的表项只能被移除。在列表中仍然包含 finalizer 时，无法强制删除对应的对象。

当 `metadata.deletionTimestamp` 字段被设置时，监视该对象的各个控制器会执行它们所能处理的 finalizer，并在完成处理之后将其从列表中移除。每个控制器负责将其 finalizer 从列表中删除。

`metadata.deletionGracePeriodSeconds` 的取值控制对更新的轮询周期。

一旦 finalizers 列表为空时，就意味着所有 finalizer 都被执行过， Kubernetes 会最终删除该资源，

合法性检查

定制资源是通过 [OpenAPI v3 模式定义](#) 来执行合法性检查的， 你可以通过使用[准入控制 Webhook](#) 来添加额外的合法性检查逻辑。

此外，对模式定义存在以下限制：

- 以下字段不可设置：
 - definitions
 - dependencies
 - deprecated
 - discriminator
 - id
 - patternProperties
 - readOnly
 - writeOnly
 - xml
 - \$ref
- 字段 uniqueItems 不可设置为 true
- 字段 additionalProperties 不可设置为 false
- 字段 additionalProperties 与 properties 互斥，不可同时使用

当[设置默认值特性](#)被启用时，可以设置字段 default。就 apiextensions.k8s.io/v1 组的 CustomResourceDefinitions，这一条件是满足的。设置默认值的功能特性从 1.17 开始正式发布。该特性在 1.16 版本中处于 Beta 状态，要求 CustomResourceDefaulting [特性门控](#) 被启用。对于大多数集群而言，Beta 状态的特性门控默认都是自动启用的。

关于对某些 CustomResourceDefinition 特性所必需的限制，可参见 [结构化的模式定义](#)小节。

模式定义是在 CustomResourceDefinition 中设置的。在下面的例子中，CustomResourceDefinition 对定制对象执行以下合法性检查：

- spec.cronSpec 必须是一个字符串，必须是正则表达式所描述的形式；
- spec.replicas 必须是一个整数，且其最小值为 1、最大值为 10。

将此 CustomResourceDefinition 保存到 resourcedefinition.yaml 文件中：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
```

```
- name: v1
  served: true
  storage: true
  schema:
    # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      type: object
      properties:
        spec:
          type: object
          properties:
            cronSpec:
              type: string
              pattern: '^(\d+|*)(\d+)?(\s+(\d+|*)(\d+)?){4}$'
            image:
              type: string
            replicas:
              type: integer
              minimum: 1
              maximum: 10
        scope: Namespaced
        names:
          plural: crontabs
          singular: crontab
          kind: CronTab
        shortNames:
        - ct
```

并创建 CustomResourceDefinition :

```
kubectl apply -f resourcedefinition.yaml
```

对于一个创建 CronTab 类别对象的定制对象的请求而言，如果其字段中包含非法值，则该请求会被拒绝。在下面的例子中，定制对象中包含带非法值的字段：

- spec.cronSpec 与正则表达式不匹配
- spec.replicas 数值大于 10。

如果你将下面的 YAML 保存到 my-crontab.yaml :

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * *"
```

```
image: my-awesome-cron-image
replicas: 15
```

并尝试创建定制对象：

```
kubectl apply -f my-crontab.yaml
```

你会看到下面的错误信息：

```
The CronTab "my-new-cron-object" is invalid: []: Invalid value:
map[string]interface {}{"apiVersion":"stable.example.com/v1",
"kind":"CronTab", "metadata":map[string]interface {}{"name":"my-new-cron-
object", "namespace":"default", "deletionTimestamp":interface {}(nil),
"deletionGracePeriodSeconds":(*int64)(nil),
"creationTimestamp":"2017-09-05T05:20:07Z", "uid":"e14d79e7-91f9-11e7-
a598-f0761cb232d1", "clusterName":""}, "spec":map[string]interface {}
{"cronSpec":"* * * * *", "image":"my-awesome-cron-image", "replicas":15}):
validation failure list:
spec.cronSpec in body should match '^(\d+|\*)(\d+)?(\s+(\d+|\*)(\d+)?)
{4}$'
spec.replicas in body should be less than or equal to 10
```

如果所有字段都包含合法值，则对象创建的请求会被接受。

将下面的 YAML 保存到 my-crontab.yaml 文件：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
  replicas: 5
```

并创建定制对象：

```
kubectl apply -f my-crontab.yaml
crontab "my-new-cron-object" created
```

设置默认值

说明：要使用设置默认值功能，你的 CustomResourceDefinition 必须使用 API 版本 apiextensions.k8s.io/v1。

设置默认值的功能允许在 [OpenAPI v3 合法性检查模式定义](#) 中设置默认值：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
```

```
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        # openAPIV3Schema 是用来检查定制对象的模式定义
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                  pattern: '^(\d+|\*)(\d+)?(\s+(\d+|\*)(\d+)?){4}$'
                  default: "5 0 * * *"
                image:
                  type: string
                replicas:
                  type: integer
                  minimum: 1
                  maximum: 10
                  default: 1
            scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  shortNames:
    - ct
```

使用此 CRD 定义时，cronSpec 和 replicas 都会被设置默认值：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  image: my-awesome-cron-image
```

会生成：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "5 0 * * *"
  image: my-awesome-cron-image
  replicas: 1
```

默认值设定的行为发生在定制对象上：

- 在向 API 服务器发送的请求中，基于请求版本的设定设置默认值；
- 在从 etcd 读取对象时，使用存储版本来设置默认值；
- 在 Mutating 准入控制插件执行非空的补丁操作时，基于准入 Webhook 对象 版本设置默认值。

从 etcd 中读取数据时所应用的默认值设置不会被写回到 etcd 中。需要通过 API 执行更新请求才能将这种方式设置的默认值写回到 etcd。

默认值一定会被剪裁（除了 metadata 字段的默认值设置），且必须通过所提供的模式定义的检查。

针对 x-kubernetes-embedded-resource: true 节点（或者包含 metadata 字段的结构的默认值）的 metadata 字段的默认值设置不会在 CustomResourceDefinition 创建时被剪裁，而是在处理请求的字段剪裁阶段被删除。

设置默认值和字段是否可为空 (Nullable)

1.20 版本新增: 对于未设置其 nullable 标志的字段或者将该标志设置为 false 的字段，其空值 (Null) 会在设置默认值之前被剪裁掉。如果对应字段 存在默认值，则默认值会被赋予该字段。当 nullable 被设置为 true 时，字段的空值会被保留，且不会在设置默认值时被覆盖。

例如，给定下面的 OpenAPI 模式定义：

```
type: object
properties:
  spec:
    type: object
    properties:
      foo:
        type: string
        nullable: false
        default: "default"
      bar:
        type: string
        nullable: true
```

```
baz  
type: string
```

像下面这样创建一个为 foo、bar 和 baz 设置空值的对象时：

```
spec:  
  foo: null  
  bar: null  
  baz: null
```

其结果会是这样：

```
spec:  
  foo: "default"  
  bar: null
```

其中的 foo 字段被剪裁掉并重新设置默认值，因为该字段是不可为空的。 bar 字段的 nullable: true 使得其能够保有其空值。 baz 字段则被完全剪裁掉，因为该字段是不可为空的，并且没有默认值设置。

以 OpenAPI v2 形式发布合法性检查模式

CustomResourceDefinition 的[结构化的、启用了剪裁的 OpenAPI v3 合法性检查模式](#)会在 Kubernetes API 服务器上作为[OpenAPI v2 规约](#)的一部分发布出来。

`kubectl` 命令行工具会基于所发布的模式定义来执行客户端的合法性检查（`kubectl create` 和 `kubectl apply`），为定制资源的模式定义 提供解释（`kubectl explain`）。 所发布的模式还可被用于其他目的，例如生成客户端或者生成文档。

OpenAPI v3 合法性检查模式定义会被转换为 OpenAPI v2 模式定义，并出现在[OpenAPI v2 规范](#)的 definitions 和 paths 字段中。

在转换过程中会发生以下修改，目的是保持与 1.13 版本以前的 `kubectl` 工具兼容。 这些修改可以避免 `kubectl` 过于严格，以至于拒绝它无法理解的 OpenAPI 模式定义。 转换过程不会更改 CRD 中定义的合法性检查模式定义，因此不会影响到 API 服务器中的[合法性检查](#)。

1. 以下字段会被移除，因为它们在 OpenAPI v2 中不支持（在将来版本中将使用 OpenAPI v3，因而不会有这些限制）
 - 字段 allOf、anyOf、oneOf 和 not 会被删除
2. 如果设置了 nullable: true，我们会丢弃 type、nullable、items 和 properties OpenAPI v2 无法表达 Nullable。为了避免 `kubectl` 拒绝正常的对象，这一转换是必要的。

额外的打印列

kubectl 工具依赖服务器端的输出格式化。你的集群的 API 服务器决定 kubectl get 命令要显示的列有哪些。你可以为 CustomResourceDefinition 定制这些要打印的列。下面的例子添加了 Spec、Replicas 和 Age 列：

将此 CustomResourceDefinition 保存到 resourcedefinition.yaml 文件：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  shortNames:
    - ct
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  additionalPrinterColumns:
    - name: Spec
      type: string
      description: The cron spec defining the interval a CronJob is run
      jsonPath: .spec.cronSpec
    - name: Replicas
      type: integer
      description: The number of jobs launched by the CronJob
```

```
jsonPath: .spec.replicas
- name: Age
  type: date
jsonPath: .metadata.creationTimestamp
```

创建 CustomResourceDefinition :

```
kubectl apply -f resourcedefinition.yaml
```

使用前文中的 my-crontab.yaml 创建一个实例。

启用服务器端打印输出 :

```
kubectl get crontab my-new-cron-object
```

注意输出中的 NAME、SPEC、REPLICAS 和 AGE 列 :

NAME	SPEC	REPLICAS	AGE
my-new-cron-object	*****	1	7s

说明： NAME 列是隐含的，不需要在 CustomResourceDefinition 中定义。

优先级

每个列都包含一个 priority (优先级) 字段。当前，优先级用来区分标准视图 (Standard View) 和宽视图 (Wide View) (使用 -o wide 标志) 中显示的列 :

- 优先级为 0 的列会在标准视图中显示。
- 优先级大于 0 的列只会在宽视图中显示。

类型

列的 type 字段可以是以下值之一 (比较 [OpenAPI v3 数据类型](#)) :

- integer - 非浮点数字
- number - 浮点数字
- string - 字符串
- boolean - true 或 false
- date - 显示为以自时间戳以来经过的时长

如果定制资源中的值与列中指定的类型不匹配，该值会被忽略。你可以通过定制资源的合法性检查来确保取值类型是正确的。

格式

列的 format 字段可以是以下值之一：

- int32
- int64
- float
- double
- byte
- date
- date-time
- password

列的 format 字段控制 kubectl 打印对应取值时采用的风格。

子资源

定制资源支持 /status 和 /scale 子资源。

通过在 CustomResourceDefinition 中定义 status 和 scale，可以有选择地启用这些子资源。

Status 子资源

当启用了 status 子资源时，对应定制资源的 /status 子资源会被暴露出来。

- status 和 spec 内容分别用定制资源内的 .status 和 .spec JSON 路径来表达；
- 对 /status 子资源的 PUT 请求要求使用定制资源对象作为其输入，但会忽略 status 之外的所有内容。
- 对 /status 子资源的 PUT 请求仅对定制资源的 status 内容进行合法性检查。
- 对定制资源的 PUT、POST、PATCH 请求会忽略 status 内容的改变。
- 对所有变更请求，除非改变是针对 .metadata 或 .status，.metadata.generation 的取值都会增加。
- 在 CRD OpenAPI 合法性检查模式定义的根节点，只允许存在以下结构：
 - description
 - example
 - exclusiveMaximum
 - exclusiveMinimum
 - externalDocs
 - format
 - items
 - maximum
 - maxItems
 - maxLength
 - minimum

- minItems
- minLength
- multipleOf
- pattern
- properties
- required
- title
- type
- uniqueItems

Scale 子资源

当启用了 scale 子资源时，定制资源的 /scale 子资源就被暴露出来。针对 /scale 所发送的对象是 autoscaling/v1.Scale。

为了启用 scale 子资源，CustomResourceDefinition 定义了以下字段：

- specReplicasPath 指定定制资源内与 scale.spec.replicas 对应的 JSON 路径。
 - 此字段为必需值。
 - 只可以使用 .spec 下的 JSON 路径，只可使用带句点的路径。
 - 如果定制资源的 specReplicasPath 下没有取值，则针对 /scale 子资源执行 GET 操作时会返回错误。
- statusReplicasPath 指定定制资源内与 scale.status.replicas 对应的 JSON 路径。
 - 此字段为必需值。
 - 只可以使用 .status 下的 JSON 路径，只可使用带句点的路径。
 - 如果定制资源的 statusReplicasPath 下没有取值，则针对 /scale 子资源的 副本个数状态值默认为 0。
- labelSelectorPath 指定定制资源内与 scale.status.selector 对应的 JSON 路径。
 - 此字段为可选值。
 - 此字段必须设置才能使用 HPA。
 - 只可以使用 .status 或 .spec 下的 JSON 路径，只可使用带句点的路径。
 - 如果定制资源的 labelSelectorPath 下没有取值，则针对 /scale 子资源的选择算符状态值默认为空字符串。
 - 此 JSON 路径所指向的字段必须是一个字符串字段（而不是复合的选择算符结构），其中包含标签选择算符串行化的字符串形式。

在下面的例子中，status 和 scale 子资源都被启用。

将此 CustomResourceDefinition 保存到 resourcedefinition.yaml 文件：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
            status:
              type: object
              properties:
                replicas:
                  type: integer
                labelSelector:
                  type: string
    # subresources 描述定制资源的子资源
  subresources:
    # status 启用 status 子资源
    status: {}
    # scale 启用 scale 子资源
    scale:
      # specReplicasPath 定义定制资源中对应 scale.spec.replicas 的 JSON 路径
      specReplicasPath: .spec.replicas
      # statusReplicasPath 定义定制资源中对应 scale.status.replicas 的 JSON 路径
      statusReplicasPath: .status.replicas
      # labelSelectorPath 定义定制资源中对应 scale.status.selector 的 JSON 路径
      labelSelectorPath: .status.labelSelector
  scope: Namespaced
```

```
names:  
  plural: crontabs  
  singular: crontab  
  kind: CronTab  
  shortNames:  
    - ct
```

之后创建此 CustomResourceDefinition：

```
kubectl apply -f resourcedefinition.yaml
```

CustomResourceDefinition 对象创建完毕之后，你可以创建定制对象，。

如果你将下面的 YAML 保存到 my-crontab.yaml 文件：

```
apiVersion: "stable.example.com/v1"  
kind: CronTab  
metadata:  
  name: my-new-cron-object  
spec:  
  cronSpec: "* * * * */5"  
  image: my-awesome-cron-image  
  replicas: 3
```

并创建定制对象：

```
kubectl apply -f my-crontab.yaml
```

那么会创建新的、命名空间作用域的 RESTful API 端点：

```
/apis/stable.example.com/v1/namespaces/*/crontabs/status
```

和

```
/apis/stable.example.com/v1/namespaces/*/crontabs/scale
```

定制资源可以使用 kubectl scale 命令来扩缩其规模。例如下面的命令将前面创建的定制资源的 .spec.replicas 设置为 5：

```
kubectl scale --replicas=5 crontabs/my-new-cron-object
```

```
crontabs "my-new-cron-object" scaled
```

```
kubectl get crontabs my-new-cron-object -o jsonpath='{.spec.replicas}'
```

```
5
```

你可以使用 [PodDisruptionBudget](#) 来保护启用了 scale 子资源的定制资源。

分类

FEATURE STATE: Kubernetes v1.10 [beta]

分类 (Categories) 是定制资源所归属的分组资源列表 (例如, all)。你可以使用 kubectl get <分类名称> 来列举属于某分类的所有资源。

下面的示例在 CustomResourceDefinition 中将 all 添加到分类列表中，并展示了如何使用 kubectl get all 来输出定制资源：

将下面的 CustomResourceDefinition 保存到 resourcedefinition.yaml 文件中：

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
  kind: CronTab
  shortNames:
    - ct
# categories 是定制资源所归属的分类资源列表
categories:
  - all
```

之后创建此 CRD：

```
kubectl apply -f resourcedefinition.yaml
```

创建了 CustomResourceDefinition 对象之后，你可以创建定制对象。

将下面的 YAML 保存到 my-crontab.yaml 中：

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

并创建定制对象：

```
kubectl apply -f my-crontab.yaml
```

你可以在使用 kubectl get 时指定分类：

```
kubectl get all
```

输出中会包含类别为 CronTab 的定制资源：

NAME	AGE
crontabs/my-new-cron-object	3s

接下来

- 阅读了解[定制资源](#)
- 参阅[CustomResourceDefinition](#)
- 参阅支持 CustomResourceDefinition 的[多个版本](#)

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 11, 2021 at 10:15 AM PST: [\[zh\] Resync tasks/extend-kubernetes/custom-resources/custom-resource-definitions.md \(b40f3649e\)](#)

CustomResourceDefinition 的版本

本页介绍如何添加版本信息到 [CustomResourceDefinitions](#)。目的是标明 CustomResourceDefinitions 的稳定级别或者服务于 API 升级。 API 升级时需要在不同 API 表示形式之间进行转换。 本页还描述如何将对象从一个版本升级到另一个版本。

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](#)

你应该对[定制资源](#)有一些初步了解。

您的 Kubernetes 服务器版本必须不低于版本 v1.16. 要获知版本信息，请输入 kubectl version.

概览

CustomResourceDefinition API 提供了用于引入和升级的工作流程到 CustomResourceDefinition 的新版本。

创建 CustomResourceDefinition 时，会在 CustomResourceDefinition spec.versions 列表设置适当的稳定级别和版本号。例如，v1beta1 表示第一个版本尚未稳定。所有定制资源对象将首先用这个版本保存。

创建 CustomResourceDefinition 后，客户端可以开始使用 v1beta1 API。

稍后可能需要添加新版本，例如 v1。

添加新版本：

1. 选择一种转化策略。由于定制资源对象需要能够两种版本都可用，这意味着它们有时会以与存储版本不同的版本来提供服务。为了能够做到这一点，有时必须在它们存储的版本和提供的版本之间进行转换。如果转换涉及模式变更，并且需要自定义逻辑，则应该使用 Webhook 来完成。如果没有模式变更，则可使用默认的 None 转换策略，为不同版本提供服务时只有 apiVersion 字段会被改变。
2. 如果使用转换 Webhook，请创建并部署转换 Webhook。更多详细信息请参见 [Webhook conversion](#)。
3. 更新 CustomResourceDefinition，将新版本设置为 served : true，加入到 spec.versions 列表。另外，还要设置 spec.conversion 字段为所选的转

换策略。如果使用转换 Webhook，请配置 spec.conversion.webhookClientConfig 来调用 Webhook。

添加新版本后，客户端可以逐步迁移到新版本。让某些客户使用旧版本的同时 支持其他人使用新版本是相当安全的。

将存储的对象迁移到新版本：

1. 请参阅[将现有对象升级到新的存储版本](#)节。

对于客户来说，在将对象升级到新的存储版本之前、期间和之后使用旧版本和新版本都是安全的。

删除旧版本：

1. 确保所有客户端都已完全迁移到新版本。可以查看 kube-apiserver 的日志以识别仍通过旧版本进行访问的所有客户端。
2. 在 spec.versions 列表中将旧版本的 served 设置为 false。如果仍有客户端意外地使用旧版本，他们可能开始会报告采用旧版本尝试访问定制资源的错误消息。如果发生这种情况，请将旧版本的 served : true 恢复，然后迁移余下的客户端 使用新版本，然后重复此步骤。
3. 确保已完成[将现有对象升级到新存储版本](#)的步骤。
 1. 在 CustomResourceDefinition 的 spec.versions 列表中，确认新版本的 stored 已被设置为 true。
 2. 确认旧版本不在 CustomResourceDefinition status.storedVersions 中。
4. 从 CustomResourceDefinition spec.versions 列表中删除旧版本。
5. 在转换 Webhooks 中放弃对旧版本的转换支持。

指定多个版本

CustomResourceDefinition API 的 versions 字段可用于支持你所开发的 定制资源的多个版本。版本可以具有不同的模式，并且转换 Webhooks 可以在多个版本之间转换定制资源。在适当的情况下，Webhook 转换应遵循 [Kubernetes API 约定](#)。尤其是，请查阅 [API 变更文档](#) 以了解一些有用的常见错误和建议。

说明：在 apiextensions.k8s.io/v1beta1 版本中曾经有一个 version 字段，名字不叫做 versions。该 version 字段已经被废弃，成为可选项。不过如果该字段不是空，则必须与 versions 字段中的第一个条目匹配。

下面的示例显示了两个版本的 CustomResourceDefinition。第一个例子中假设所有的版本使用相同的模式而它们之间没有转换。YAML 中的注释提供了更多背景信息。

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

apiVersion: apiextensions.k8s.io/v1

kind: CustomResourceDefinition

```
metadata:  
  # name 必须匹配后面 spec 中的字段，且使用格式 <plural>.<group>  
  name: crontabs.example.com  
spec:  
  # 组名，用于 REST API: /apis/<group>/<version>  
  group: example.com  
  # 此 CustomResourceDefinition 所支持的版本列表  
  versions:  
    - name: v1beta1  
      # 每个 version 可以通过 served 标志启用或禁止  
      served: true  
      # 有且只能有一个 version 必须被标记为存储版本  
      storage: true  
      # schema 是必需字段  
      schema:  
        openAPIV3Schema:  
          type: object  
          properties:  
            host:  
              type: string  
            port:  
              type: string  
    - name: v1  
      served: true  
      storage: false  
      schema:  
        openAPIV3Schema:  
          type: object  
          properties:  
            host:  
              type: string  
            port:  
              type: string  
  # conversion 节是 Kubernetes 1.13+ 版本引入的，其默认值为无转换，即  
  # strategy 子字段设置为 None。  
  conversion:  
    # None 转换假定所有版本采用相同的模式定义，仅仅将定制资源的 apiVersion  
    # 设置为合适的值。  
    strategy: None  
    # 可以是 Namespaced 或 Cluster  
    scope: Namespaced  
names:  
  # 名称的复数形式，用于 URL: /apis/<group>/<version>/<plural>  
  plural: crontabs  
  # 名称的单数形式，用于在命令行接口和显示时作为其别名  
  singular: crontab
```

```
# kind 通常是驼峰编码 ( CamelCased ) 的单数形式，用于资源清单中
kind: CronTab
# shortNames 允许你在命令行接口中使用更短的字符串来匹配你的资源
shortNames:
- ct

# 在 v1.16 中被弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
# name 必须匹配后面 spec 中的字段，且使用格式 <plural>.<group>
name: crontabs.example.com
spec:
# 组名，用于 REST API: /apis/<group>/<version>
group: example.com
# 此 CustomResourceDefinition 所支持的版本列表
versions:
- name: v1beta1
# 每个 version 可以通过 served 标志启用或禁止
served: true
# 有且只能有一个 version 必须被标记为存储版本
storage: true
- name: v1
  served: true
  storage: false
validation:
openAPIV3Schema:
  type: object
  properties:
    host:
      type: string
    port:
      type: string
# conversion 节是 Kubernetes 1.13+ 版本引入的，其默认值为无转换，即
# strategy 子字段设置为 None。
conversion:
# None 转换假定所有版本采用相同的模式定义，仅仅将定制资源的 apiVersion
# 设置为合适的值。
strategy: None
# 可以是 Namespaced 或 Cluster
scope: Namespaced
names:
# 名称的复数形式，用于 URL: /apis/<group>/<version>/<plural>
plural: crontabs
# 名称的单数形式，用于在命令行接口和显示时作为其别名
singular: crontab
```

```
# kind 通常是驼峰编码 ( CamelCased ) 的单数形式，用于资源清单中
kind: CronTab
# shortNames 允许你在命令行接口中使用更短的字符串来匹配你的资源
shortNames:
  - ct
```

你可以将 CustomResourceDefinition 存储在 YAML 文件中，然后使用 kubectl apply 来创建它。

```
kubectl apply -f my-versioned-crontab.yaml
```

在创建之后，API 服务器开始在 HTTP REST 端点上为每个已启用的版本提供服务。在上面的示例中，API 版本可以在 /apis/example.com/v1beta1 和 /apis/example.com/v1 处获得。

版本优先级

不考虑 CustomResourceDefinition 中版本被定义的顺序，kubectl 使用具有最高优先级的版本作为访问对象的默认版本。通过解析 name 字段确定优先级来决定版本号，稳定性（GA、Beta 或 Alpha）级别及该稳定性级别的序列。

用于对版本进行排序的算法在设计上与 Kubernetes 项目对 Kubernetes 版本进行排序的方式相同。版本以 v 开头跟一个数字，一个可选的 beta 或者 alpha 和一个可选的附加数字 作为版本信息。从广义上讲，版本字符串可能看起来像 v2 或者 v2beta1。使用以下算法对版本进行排序：

- 遵循 Kubernetes 版本模式的条目在不符合条件的条目之前进行排序。
- 对于遵循 Kubernetes 版本模式的条目，版本字符串的数字部分从最大到最小排序。
- 如果第一个数字后面有字符串 beta 或 alpha，它们首先按去掉 beta 或 alpha 之后的版本号排序（相当于 GA 版本），之后按 beta 先、alpha 后的顺序排序，
- 如果 beta 或 alpha 之后还有另一个数字，那么也会针对这些数字 从大到小排序。
- 不符合上述格式的字符串按字母顺序排序，数字部分不经过特殊处理。请注意，在下面的示例中，foo1 排在 foo10 之前。这与遵循 Kubernetes 版本模式的条目的数字部分排序不同。

如果查看以下版本排序列表，这些规则就容易懂了：

```
- v10
- v2
- v1
- v11beta2
- v10beta3
- v3beta1
- v12alpha1
- v11alpha2
```

```
- foo1
- foo10
```

对于[指定多个版本](#)中的示例，版本排序顺序为 v1，后跟着 v1beta1。这导致了 kubectl 命令使用 v1 作为默认版本，除非所提供的对象指定了版本。

版本废弃

FEATURE STATE: Kubernetes v1.19 [stable]

从 v1.19 开始，CustomResourceDefinition 可用来标明所定义的资源的特定版本 被废弃。当发起对已废弃的版本的 API 请求时，会在 API 响应中以 HTTP 头部的形式返回警告消息。如果需要，可以对资源的每个废弃版本定制该警告消息。

定制的警告消息应该标明废弃的 API 组、版本和类别（kind），并且应该标明 应该使用（如果有的话）哪个 API 组、版本和类别作为替代。

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
  name: crontabs.example.com
spec:
  group: example.com
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  scope: Namespaced
  versions:
    - name: v1alpha1
      served: true
        # 此属性标明此定制资源的 v1alpha1 版本已被弃用。
        # 发给此版本的 API 请求会在服务器响应中收到警告消息头。
      deprecated: true
        # 此属性设置用来覆盖返回给发送 v1alpha1 API 请求的客户端的默认警告信息。
      deprecationWarning: "example.com/v1alpha1 CronTab is deprecated; see
http://example.com/v1alpha1-v1 for instructions to migrate to
example.com/v1 CronTab"
      schema: ...
    - name: v1beta1
      served: true
        # 此属性标明该定制资源的 v1beta1 版本已被弃用。
        # 发给此版本的 API 请求会在服务器响应中收到警告消息头。
        # 针对此版本的请求所返回的是默认的警告消息。
      deprecated: true
```

```

schema: ...
- name: v1
  served: true
  storage: true
  schema: ...

# 在 v1.16 中弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.example.com
spec:
  group: example.com
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
  scope: Namespaced
  validation: ...
  versions:
    - name: v1alpha1
      served: true
      # 此属性标明此定制资源的 v1alpha1 版本已被弃用。
      # 发给此版本的 API 请求会在服务器响应中收到警告消息头。
      deprecated: true
      # 此属性设置用来覆盖返回给发送 v1alpha1 API 请求的客户端的默认警告信息。
      deprecationWarning: "example.com/v1alpha1 CronTab is deprecated; see
http://example.com/v1alpha1-v1 for instructions to migrate to
example.com/v1 CronTab"
    - name: v1beta1
      served: true
      # 此属性标明该定制资源的 v1beta1 版本已被弃用。
      # 发给此版本的 API 请求会在服务器响应中收到警告消息头。
      # 针对此版本的请求所返回的是默认的警告消息。
      deprecated: true
    - name: v1
      served: true
      storage: true

```

Webhook 转换

FEATURE STATE: Kubernetes v1.16 [stable]

说明： Webhook 转换在 Kubernetes 1.13 版本引入，在 Kubernetes 1.15 中成为 Beta 功能。要使用此功能，应启用 Custo

`mResourceWebhookConversion` 特性。在大多数集群上，这类 Beta 特性应该是自动启用的。请参阅[特性门控](#) 文档以获得更多信息。

上面的例子在版本之间有一个 `None` 转换，它只在转换时设置 `apiVersion` 字段而不改变对象的其余部分。API 服务器还支持在需要转换时调用外部服务的 webhook 转换。例如：

- 定制资源的请求版本与其存储版本不同。
- 使用某版本创建了 Watch 请求，但所更改对象以另一版本存储。
- 定制资源的 PUT 请求所针对版本与存储版本不同。

为了涵盖所有这些情况并优化 API 服务器所作的转换，转换请求可以包含多个对象，以便减少外部调用。Webhook 应该独立执行各个转换。

编写一个转换 Webhook 服务器

请参考[定制资源转换 Webhook 服务器](#) 的实现；该实现在 Kubernetes e2e 测试中得到验证。Webhook 处理由 API 服务器发送的 `ConversionReview` 请求，并在 `ConversionResponse` 中封装发回转换结果。请注意，请求包含需要独立转换的定制资源列表，这些对象在被转换之后不能改变其在列表中的顺序。该示例服务器的组织方式使其可以复用于其他转换。大多数常见代码都位于[framework 文件](#)中，只留下[一个函数](#) 用于实现不同的转换。

说明：转换 Webhook 服务器示例中将 `ClientAuth` 字段设置为 [空](#)，默认为 `NoClientCert`。这意味着 webhook 服务器没有验证客户端（也就是 API 服务器）的身份。如果你需要双向 TLS 或者其他方式来验证客户端，请参阅如何[验证 API 服务](#)。

被允许的变更

转换 Webhook 不可以更改被转换对象的 `metadata` 中除 `labels` 和 `annotations` 之外的任何属性。尝试更改 `name`、`UID` 和 `namespace` 时都会导致引起转换的请求失败。所有其他变更只是被忽略而已。

部署转换 Webhook 服务

用于部署转换 webhook 的文档与[准入 Webhook 服务示例](#)相同。这里的假设是转换 Webhook 服务器被部署为 `default` 名字空间中名为 `example-conversion-webhook-server` 的服务，并在路径 `/crdconvert` 上处理请求。

说明：当 Webhook 服务器作为一个服务被部署到 Kubernetes 集群中时，它必须通过端口 443 公开其服务（服务器本身可以使用任意端口，但是服务对象应该将它映射到端口 443）。如果为服务器使用不同的端口，则 API 服务器和 Webhook 服务器之间的通信可能会失败。

配置 CustomResourceDefinition 以使用转换 Webhook

通过修改 spec 中的 conversion 部分，可以扩展 None 转换示例来 使用转换 Webhook。

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name 必须匹配后面 spec 中的字段，且使用格式 <plural>.<group>
  name: crontabs.example.com
spec:
  # 组名，用于 REST API: /apis/<group>/<version>
  group: example.com
  # 此 CustomResourceDefinition 所支持的版本列表
  versions:
    - name: v1beta1
      # 每个 version 可以通过 served 标志启用或禁止
      served: true
      # 有且只能有一个 version 必须被标记为存储版本
      storage: true
      # 当不存在顶级模式定义时，每个版本（version）可以定义其自身的模式
      schema:
        openAPIV3Schema:
          type: object
          properties:
            hostPort:
              type: string
      - name: v1
        served: true
        storage: false
        schema:
          openAPIV3Schema:
            type: object
            properties:
              host:
                type: string
              port:
                type: string
  conversion:
    # Webhook strategy 告诉 API 服务器调用外部 Webhook 来完成定制资源
    # 之间的转换
    strategy: Webhook
    # 当 strategy 为 "Webhook" 时，webhook 属性是必需的
```

```
# 该属性配置将被 API 服务器调用的 Webhook 端点
webhook:
  # conversionReviewVersions 标明 Webhook 所能理解或偏好使用的
  # ConversionReview 对象版本。
  # API 服务器所能理解的列表中的第一个版本会被发送到 Webhook
  # Webhook 必须按所接收到的版本响应一个 ConversionReview 对象
  conversionReviewVersions: ["v1", "v1beta1"]
  clientConfig:
    service:
      namespace: default
      name: example-conversion-webhook-server
      path: /crdconvert
      caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
  # 可以是 Namespaced 或 Cluster
  scope: Namespaced
  names:
    # 名称的复数形式，用于 URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # 名称的单数形式，用于在命令行接口和显示时作为其别名
    singular: crontab
    # kind 通常是驼峰编码 ( CamelCased ) 的单数形式，用于资源清单中
    kind: CronTab
    # shortNames 允许你在命令行接口中使用更短的字符串来匹配你的资源
    shortNames:
      - ct
```

```
# 在 v1.16 中被弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name 必须匹配后面 spec 中的字段，且使用格式 <plural>.<group>
  name: crontabs.example.com
spec:
  # 组名，用于 REST API: /apis/<group>/<version>
  group: example.com
  # 裁剪掉下面的 OpenAPI 模式中未曾定义的对象字段
  preserveUnknownFields: false
  # 此 CustomResourceDefinition 所支持的版本列表
  versions:
    - name: v1beta1
      # 每个 version 可以通过 served 标志启用或禁止
      served: true
      # 有且只能有一个 version 必须被标记为存储版本
      storage: true
      # 当不存在顶级模式定义时，每个版本 ( version ) 可以定义其自身的模式
      schema:
```

```
openAPIV3Schema:
  type: object
  properties:
    hostPort:
      type: string
    - name: v1
      served: true
      storage: false
    schema:
      openAPIV3Schema:
        type: object
        properties:
          host:
            type: string
          port:
            type: string
conversion:
  # Webhook strategy 告诉 API 服务器调用外部 Webhook 来完成定制资源
  strategy: Webhook
  # 当 strategy 为 "Webhook" 时， webhookClientConfig 属性是必需的
  # 该属性配置将被 API 服务器调用的 Webhook 端点
  webhookClientConfig:
    service:
      namespace: default
      name: example-conversion-webhook-server
      path: /crdconvert
      caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
  # 可以是 Namespaced 或 Cluster
  scope: Namespaced
names:
  # 名称的复数形式，用于 URL: /apis/<group>/<version>/<plural>
  plural: crontabs
  # 名称的单数形式，用于在命令行接口和显示时作为其别名
  singular: crontab
  # kind 通常是驼峰编码 ( CamelCased ) 的单数形式，用于资源清单中
  kind: CronTab
  # shortNames 允许你在命令行接口中使用更短的字符串来匹配你的资源
  shortNames:
    - ct
```

你可以将 CustomResourceDefinition 保存在 YAML 文件中，然后使用 kubectl apply 来应用它。

```
kubectl apply -f my-versioned-crontab-with-conversion.yaml
```

在应用新更改之前，请确保转换服务器已启动并正在运行。

调用 Webhook

API 服务器一旦确定请求应发送到转换 Webhook，它需要知道如何调用 Webhook。这是在 webhookClientConfig 中指定的 Webhook 配置。

转换 Webhook 可以通过 URL 或服务引用来调用，并且可以选择包含自定义 CA 包，以用于验证 TLS 连接。

URL

url 以标准 URL 形式给出 Webhook 的位置 (scheme://host:port/path)。 host 不应引用集群中运行的服务，而应通过指定 service 字段来提供服务引用。在某些 API 服务器中，host 可以通过外部 DNS 进行解析（即 kube-apiserver 无法解析集群内 DNS，那样会违反分层规则）。host 也可以是 IP 地址。

请注意，除非你非常小心地在所有运行着可能调用 Webhook 的 API 服务器的主机上运行此 Webhook，否则将 localhost 或 127.0.0.1 用作 host 是风险很大的。这样的安装很可能是不可移植的，即很难在新集群中启用。

HTTP 协议必须为 https；URL 必须以 https:// 开头。

尝试使用用户名或基本身份验证（例如，使用 user:password@）是不允许的。URL 片段 (#...) 和查询参数 (?...) 也是不允许的。

下面是为调用 URL 来执行转换 Webhook 的示例，其中期望使用系统信任根来验证 TLS 证书，因此未指定 caBundle：

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
...
spec:
  ...
    conversion:
      strategy: Webhook
      webhook:
        clientConfig:
          url: "https://my-webhook.example.com:9443/my-webhook-path"
  ...
# 在 v1.16 中已弃用以推荐使用 apiextensions.k8s.io/v1
```

在 v1.16 中已弃用以推荐使用 apiextensions.k8s.io/v1

apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

...

spec:

...

```
conversion:  
  strategy: Webhook  
  webhookClientConfig:  
    url: "https://my-webhook.example.com:9443/my-webhook-path"  
...  
...
```

服务引用

webhookClientConfig 内部的 service 段是对转换 Webhook 服务的引用。如果 Webhook 在集群中运行，则应使用 service 而不是 url。服务的名字空间和名称是必需的。端口是可选的，默认为 443。路径是可选的，默认为/。

下面配置中，服务配置为在端口 1234、子路径 /my-path 上被调用。例子中针对 ServerName my-service-name.my-service-namespace.svc，使用自定义 CA 包验证 TLS 连接。

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
```

```
kind: CustomResourceDefinition
```

```
...
```

```
spec:
```

```
...
```

```
conversion:
```

```
  strategy: Webhook
```

```
  webhook:
```

```
    clientConfig:
```

```
      service:
```

```
        namespace: my-service-namespace
```

```
        name: my-service-name
```

```
        path: /my-path
```

```
        port: 1234
```

```
      caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
```

```
...
```

```
# v1.16 中被弃用以推荐使用 apiextensions.k8s.io/v1
```

```
apiVersion: apiextensions.k8s.io/v1beta1
```

```
kind: CustomResourceDefinition
```

```
...
```

```
spec:
```

```
...
```

```
conversion:
```

```
  strategy: Webhook
```

```
  webhookClientConfig:
```

```
    service:
```

```
      namespace: my-service-namespace
```

```
name: my-service-name
path: /my-path
port: 1234
caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
...
...
```

Webhook 请求和响应

请求

向 Webhooks 发起请求的动词是 POST，请求的 Content-Type 为 application/json。请求的主题为 JSON 序列化形式的 apiextensions.k8s.io API 组的 ConversionReview API 对象。

Webhooks 可以在其 CustomResourceDefinition 中使用 conversionReviewVersions 字段 设置它们接受的 ConversionReview 对象的版本：

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
...
spec:
  ...
    conversion:
      strategy: Webhook
      webhook
      conversionReviewVersions: ["v1", "v1beta1"]
  ...
...
```

创建 apiextensions.k8s.io/v1 版本的自定义资源定义时，conversionReviewVersions 是必填字段。Webhooks 要求支持至少一个 ConversionReview 当前和以前的 API 服务器可以理解的版本。

```
# v1.16 已弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
...
spec:
  ...
    conversion:
      strategy: Webhook
      conversionReviewVersions: ["v1", "v1beta1"]
  ...
...
```

创建 apiextensions.k8s.io/v1beta1 定制资源定义时若未指定 conversionReviewVersions，则默认值为 v1beta1。

API 服务器将 conversionReviewVersions 列表中他们所支持的第一个 ConversionReview 资源版本发送给 Webhook。如果列表中的版本都不被 API 服务器支持，则无法创建自定义资源定义。如果某 API 服务器遇到之前创建的转换 Webhook 配置，并且该配置不支持 API 服务器知道如何发送的任何 ConversionReview 版本，调用 Webhook 的尝试会失败。

下面的示例显示了包含在 ConversionReview 对象中的数据，该请求意在将 CronTab 对象转换为 example.com/v1：

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: ConversionReview
request:
  # 用来唯一标识此转换调用的随机 UID
  uid: 705ab4f5-6393-11e8-b7cc-42010a800002

  # 对象要转换到的目标 API 组和版本
  desiredAPIVersion: example.com/v1

  # 要转换的对象列表
  # 其中可能包含一个或多个对象，版本可能相同也可能不同
  objects:
    - kind: CronTab
      apiVersion: example.com/v1beta1
      metadata:
        creationTimestamp: "2019-09-04T14:03:02Z"
        name: local-crontab
        namespace: default
        resourceVersion: "143"
        uid: "3415a7fc-162b-4300-b5da-fd6083580d66"
      hostPort: "localhost:1234"
    - kind: CronTab
      apiVersion: example.com/v1beta1
      metadata:
        creationTimestamp: "2019-09-03T13:02:01Z"
        name: remote-crontab
        resourceVersion: "12893",
        uid: "359a83ec-b575-460d-b553-d859cedde8a0"
      hostPort: example.com:2345

# v1.16 中已废弃以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
```

```

kind: ConversionReview
request:
  # 用来唯一标识此转换调用的随机 UID
  uid: 705ab4f5-6393-11e8-b7cc-42010a800002

  # 对象要转换到的目标 API 组和版本
  desiredAPIVersion: example.com/v1

  # 要转换的对象列表
  # 其中可能包含一个或多个对象，版本可能相同也可能不同
objects:
  - kind: CronTab
    apiVersion: example.com/v1beta1
    metadata:
      creationTimestamp: "2019-09-04T14:03:02Z"
      name: local-crontab
      namespace: default
      resourceVersion: "143"
      uid: "3415a7fc-162b-4300-b5da-fd6083580d66"
    hostPort: "localhost:1234"
  - kind: CronTab
    apiVersion: example.com/v1beta1
    metadata:
      creationTimestamp: "2019-09-03T13:02:01Z"
      name: remote-crontab
      resourceVersion: "12893",
      uid: "359a83ec-b575-460d-b553-d859cedde8a0"
    hostPort: example.com:2345

```

响应

Webhooks 响应包含 200 HTTP 状态代码、Content-Type: application/json，在主体中包含 JSON 序列化形式的数据，在 response 节中给出 ConversionReview 对象（与发送的版本相同）。

如果转换成功，则 Webhook 应该返回包含以下字段的 response 节：

- uid，从发送到 webhook 的 request.uid 复制而来
- result，设置为 {"status": "Success"}
- convertedObjects，包含来自 request.objects 的所有对象，均已转换为 request.desiredVersion

Webhook 的最简单成功响应示例：

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: ConversionReview
response:
  # 必须与 <request.uid> 匹配
  uid: "705ab4f5-6393-11e8-b7cc-42010a800002"
  result:
    status: Success
    # 这里的对象必须与 request.objects 中的对象顺序相同并且其 apiVersion
    # 被设置为 <request.desiredAPIVersion>。
    # kind、metadata.uid、metadata.name 和 metadata.namespace 等字段都不
    可
    # 被 Webhook 修改。
    # Webhook 可以更改 metadata.labels 和 metadata.annotations 字段值
    # Webhook 对 metadata 中其他字段的更改都会被忽略
  convertedObjects:
    - kind: CronTab
      apiVersion: example.com/v1
      metadata:
        creationTimestamp: "2019-09-04T14:03:02Z"
        name: local-crontab
        namespace: default
        resourceVersion: "143",
        uid: "3415a7fc-162b-4300-b5da-fd6083580d66"
      host: localhost
      port: "1234"
    - kind: CronTab
      apiVersion: example.com/v1
      metadata:
        creationTimestamp: "2019-09-03T13:02:01Z",
        name: remote-crontab
        resourceVersion: "12893",
        uid: "359a83ec-b575-460d-b553-d859cedde8a0"
      host: example.com
      port: "2345"
```

```
# v1.16 中已弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: ConversionReview
response:
  # 必须与 <request.uid> 匹配
  uid: "705ab4f5-6393-11e8-b7cc-42010a800002"
  result:
    status: Failed
    # 这里的对象必须与 request.objects 中的对象顺序相同并且其 apiVersion
    # 被设置为 <request.desiredAPIVersion>。
    # kind、metadata.uid、metadata.name 和 metadata.namespace 等字段都不
```

可

```

# 被 Webhook 修改。
# Webhook 可以更改 metadata.labels 和 metadata.annotations 字段值
# Webhook 对 metadata 中其他字段的更改都会被忽略
convertedObjects:
  - kind: CronTab
    apiVersion: example.com/v1
    metadata:
      creationTimestamp: "2019-09-04T14:03:02Z"
      name: local-crontab
      namespace: default
      resourceVersion: "143",
      uid: "3415a7fc-162b-4300-b5da-fd6083580d66"
    host: localhost
    port: "1234"
  - kind: CronTab
    apiVersion: example.com/v1
    metadata:
      creationTimestamp: "2019-09-03T13:02:01Z",
      name: remote-crontab
      resourceVersion: "12893",
      uid: "359a83ec-b575-460d-b553-d859cedde8a0"
    host: example.com
    port: "2345"

```

如果转换失败，则 Webhook 应该返回包含以下字段的 response 节：

*uid，从发送到 Webhook 的 request.uid 复制而来 *result，设置为 {"status": "Failed"}

警告：

转换失败会破坏对定制资源的读写访问，包括更新或删除资源的能力。
转换失败应尽可能避免，并且不可用于实施合法性检查约束（应改用验证模式或 Webhook 准入插件）。

来自 Webhook 的响应示例，指示转换请求失败，并带有可选消息：

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```

apiVersion: apiextensions.k8s.io/v1
kind: ConversionReview
response:
  uid: <value from request.uid>
  result: {
    status: Failed
    message: hostPort could not be parsed into a separate host and port

```

```
# v1.16 中弃用以推荐使用 apiextensions.k8s.io/v1
apiVersion: apiextensions.k8s.io/v1beta1
kind: ConversionReview
response:
  uid: <value from request.uid>
  result:
    status: Failed
  message: hostPort could not be parsed into a separate host and port
```

编写、读取和更新版本化的 CustomResourceDefinition 对象

写入对象时，将使用写入时指定的存储版本来存储。如果存储版本发生变化，现有对象永远不会被自动转换。然而，新创建或被更新的对象将以新的存储版本写入。对象写入的版本不再被支持是有可能的。

当读取对象时，作为路径的一部分，你需要指定版本。如果所指定的版本与对象的持久版本不同，Kubernetes 会按所请求的版本将对象返回，但是在满足服务请求时，被持久化的对象既不会在磁盘上更改，也不会以任何方式进行转换（除了 api Version 字符串被更改之外）。你可以以当前提供的任何版本来请求对象。

如果你更新一个现有对象，它将以当前的存储版本被重写。这是可以将对象从一个版本改到另一个版本的唯一办法。

为了说明这一点，请考虑以下假设的一系列事件：

1. 存储版本是 v1beta1。你创建一个对象。该对象以版本 v1beta1 存储。
2. 你将为 CustomResourceDefinition 添加版本 v1，并将其指定为存储版本。
3. 你使用版本 v1beta1 来读取你的对象，然后你再次用版本 v1 读取对象。除了 apiVersion 字段之外，返回的两个对象是完全相同的。
4. 你创建一个新对象。对象以版本 v1 保存在存储中。你现在有两个对象，其中一个是 v1beta1，另一个是 v1。
5. 你更新第一个对象。该对象现在以版本 v1 保存，因为 v1 是当前的存储版本。

以前的存储版本

API 服务器在状态字段 storedVersions 中记录曾被标记为存储版本的每个版本。对象可能以任何曾被指定为存储版本的版本保存。存储中不会出现从未成为存储版本的版本的对象。

将现有对象升级到新的存储版本

弃用版本并删除其支持时，请设计存储升级过程。

选项 1：使用存储版本迁移程序 (Storage Version Migrator)

1. 运行[存储版本迁移程序](#)
2. 从 CustomResourceDefinition 的 status.storedVersions 字段中去掉老的版本。

选项 2：手动将现有对象升级到新的存储版本

以下是从 v1beta1 升级到 v1 的示例过程。

1. 在 CustomResourceDefinition 文件中将 v1 设置为存储版本，并使用 kubectl 应用它。 storedVersions 现在是 v1beta1, v1。
2. 编写升级过程以列出所有现有对象并使用相同内容将其写回存储。 这会强制后端使用当前存储版本（即 v1）写入对象。
3. 通过从 storedVersions 字段中删除 v1beta1 来更新 CustomResourceDefinition 的 Status。

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 23, 2020 at 4:19 PM PST: [\[zh\] Resync tasks/.../custom-resource-definition-versioning.md \(54ab3daff\)](#)

配置聚合层

配置[聚合层](#) 可以允许 Kubernetes apiserver 使用其它 API 扩展，这些 API 不是核心 Kubernetes API 的一部分。

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.

说明：要使聚合层在你的环境中正常工作以支持代理服务器和扩展 apiserver 之间的相互 TLS 身份验证，需要满足一些设置要求。

Kubernetes 和 kube-apiserver 具有多个 CA，因此请确保代理是由聚合层 CA 签名的，而不是由主 CA 签名的。

注意：对不同的客户端类型重复使用相同的 CA 会对群集的功能产生负面影响。有关更多信息，请参见 [CA 重用和冲突](#)。

身份认证流程

与自定义资源定义 (CRD) 不同，除标准的 Kubernetes apiserver 外，Aggregation API 还涉及另一个服务器：扩展 apiserver。Kubernetes apiserver 将需要与你的扩展 apiserver 通信，并且你的扩展 apiserver 也需要与 Kubernetes apiserver 通信。为了确保此通信的安全，Kubernetes apiserver 使用 x509 证书向扩展 apiserver 认证。

本节介绍身份认证和鉴权流程的工作方式以及如何配置它们。

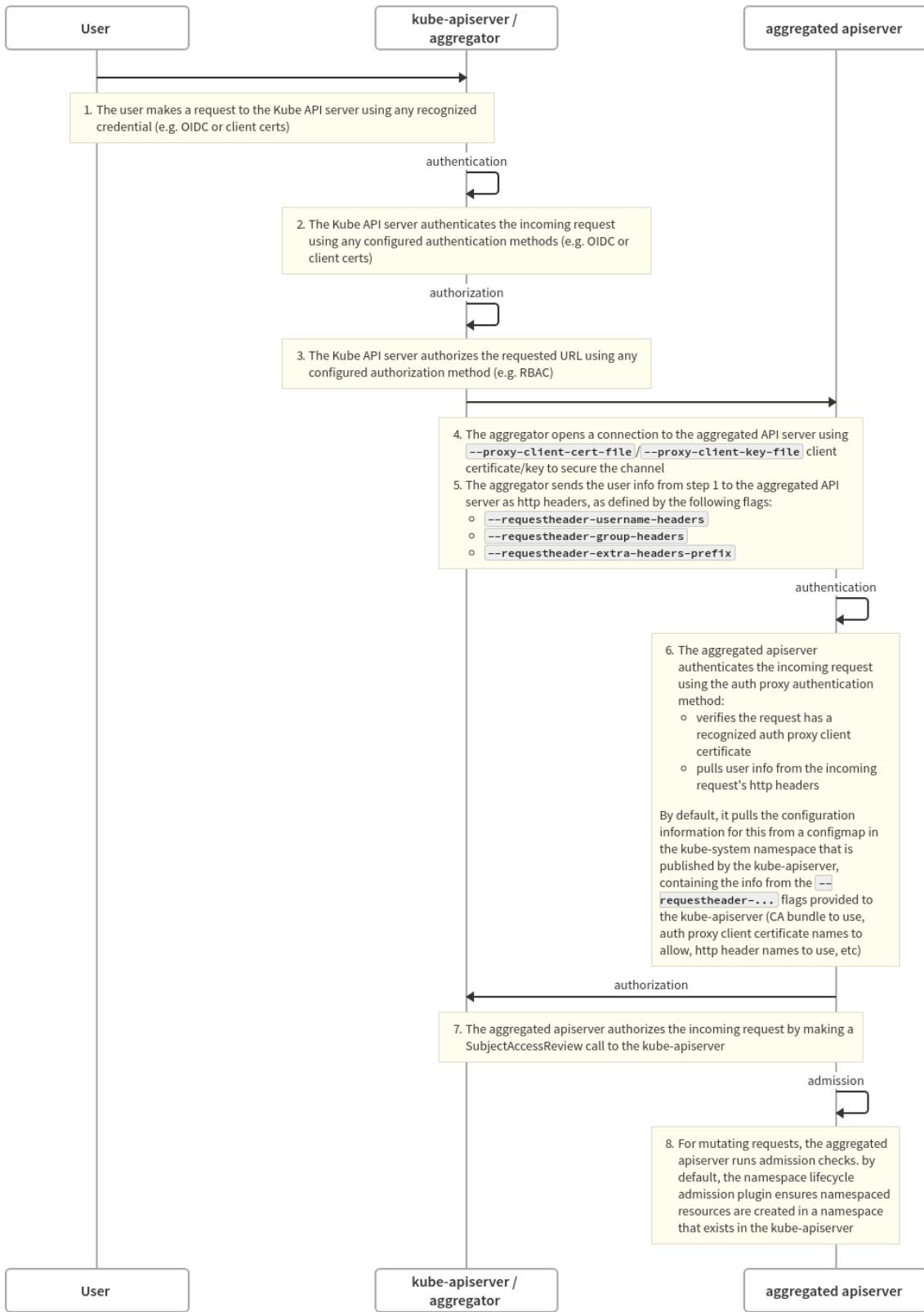
大致流程如下：

1. Kubernetes apiserver：对发出请求的用户身份认证，并对请求的 API 路径执行鉴权。
2. Kubernetes apiserver：将请求转发到扩展 apiserver
3. 扩展 apiserver：认证来自 Kubernetes apiserver 的请求
4. 扩展 apiserver：对来自原始用户的请求鉴权
5. 扩展 apiserver：执行

本节的其余部分详细描述了这些步骤。

该流程可以在下图中看到。

Welcome to swimlanes.io



POWERED BY [swimlanes.io](#)

以上泳道的来源可以在本文档的源码中找到。

Kubernetes Apiserver 认证和授权

由扩展 apiserver 服务的对 API 路径的请求以与所有 API 请求相同的方式开始：与 Kubernetes apiserver 的通信。该路径已通过扩展 apiserver 在 Kubernetes apiserver 中注册。

用户与 Kubernetes apiserver 通信，请求访问路径。Kubernetes apiserver 使用它的标准认证和授权配置来对用户认证，以及对特定路径的鉴权。

有关对 Kubernetes 集群认证的概述，请参见 [对集群认证](#)。有关对 Kubernetes 集群资源的访问鉴权的概述，请参见 [鉴权概述](#)。

到目前为止，所有内容都是标准的 Kubernetes API 请求，认证与鉴权。

Kubernetes apiserver 现在准备将请求发送到扩展 apiserver。

Kubernetes Apiserver 代理请求

Kubernetes apiserver 现在将请求发送或代理到注册以处理该请求的扩展 apiserver。为此，它需要了解几件事：

1. Kubernetes apiserver 应该如何向扩展 apiserver 认证，以通知扩展 apiserver 通过网络发出的请求来自有效的 Kubernetes apiserver？
2. Kubernetes apiserver 应该如何通知扩展 apiserver 原始请求 已通过认证的用户名和组？

为提供这两条信息，你必须使用若干标志来配置 Kubernetes apiserver。

Kubernetes Apiserver 客户端认证

Kubernetes apiserver 通过 TLS 连接到扩展 apiserver，并使用客户端证书认证。你必须在启动时使用提供的标志向 Kubernetes apiserver 提供以下内容：

- 通过 `--proxy-client-key-file` 指定私钥文件
- 通过 `--proxy-client-cert-file` 签名的客户端证书文件
- 通过 `--requestheader-client-ca-file` 签署客户端证书文件的 CA 证书
- 通过 `--requestheader-allowed-names` 在签署的客户证书中有效的公用名 (CN)

Kubernetes apiserver 将使用由 `--proxy-client-*file` 指示的文件来验证扩展 apiserver。为了使合规的扩展 apiserver 能够将该请求视为有效，必须满足以下条件：

1. 连接必须使用由 CA 签署的客户端证书，该证书的证书位于 `--requestheader-client-ca-file` 中。
2. 连接必须使用客户端证书，该客户端证书的 CN 是 `--requestheader-allowed-names` 中列出的证书之一。

说明：你可以将此选项设置为空白，即为`--requestheader-allowed-names`。这将向扩展 apiserver 指示任何 CN 是可接受的。

使用这些选项启动时，Kubernetes apiserver 将：

1. 使用它们向扩展 apiserver 认证。
2. 在 `kube-system` 命名空间中 创建一个名为 `extension-apiserver-authentication` 的 ConfigMap，它将在其中放置 CA 证书和允许的 CN。反过来，扩展 apiserver 可以检索这些内容以验证请求。

请注意，Kubernetes apiserver 使用相同的客户端证书对所有扩展 apiserver 认证。它不会为每个扩展 apiserver 创建一个客户端证书，而是创建一个证书作为 Kubernetes apiserver 认证。所有扩展 apiserver 请求都重复使用相同的请求。

原始请求用户名和组

当 Kubernetes apiserver 将请求代理到扩展 apiserver 时，它将向扩展 apiserver 通知原始请求已成功通过其验证的用户名和组。它在其代理请求的 HTTP 头部中提供这些。你必须将要使用的标头名称告知 Kubernetes apiserver。

- 通过`--requestheader-username-headers` 标明用来保存用户名的头部
- 通过`--requestheader-group-headers` 标明用来保存 group 的头部
- 通过`--requestheader-extra-headers-prefix` 标明用来保存拓展信息前缀的头部

这些头部名称也放置在 `extension-apiserver-authentication` ConfigMap 中，因此扩展 apiserver 可以检索和使用它们。

扩展 Apiserver 认证

扩展 apiserver 在收到来自 Kubernetes apiserver 的代理请求后，必须验证该请求确实来自有效的身份验证代理，该认证代理由 Kubernetes apiserver 履行。扩展 apiserver 通过以下方式对其认证：

1. 如上所述，从 `kube-system` 中的 configmap 中检索以下内容：
 - 客户端 CA 证书
 - 允许名称 (CN) 列表
 - 用户名，组和其他信息的头部
2. 使用以下证书检查 TLS 连接是否已通过认证：
 - 由其证书与检索到的 CA 证书匹配的 CA 签名。
 - 在允许的 CN 列表中有一个 CN，除非列表为空，在这种情况下允许所有 CN。
 - 从适当的头部中提取用户名和组

如果以上均通过，则该请求是来自合法认证代理（在本例中为 Kubernetes apiserver）的有效代理请求。

请注意，扩展 apiserver 实现负责提供上述内容。默认情况下，许多扩展 apiserver 实现利用 k8s.io/apiserver/ 软件包来做到这一点。也有一些实现可能支持使用命令行选项来覆盖这些配置。

为了具有检索 configmap 的权限，扩展 apiserver 需要适当的角色。在 kube-system 名字空间中有一个默认角色 extension-apiserver-authentication-reader 可用于设置。

扩展 Apiserver 对请求鉴权

扩展 apiserver 现在可以验证从标头检索的 user/group 是否有权执行给定请求。通过向 Kubernetes apiserver 发送标准 [SubjectAccessReview](#) 请求来实现。

为了使扩展 apiserver 本身被鉴权可以向 Kubernetes apiserver 提交 SubjectAccessReview 请求，它需要正确的权限。Kubernetes 包含一个具有相应权限的名为 system:auth-delegator 的默认 ClusterRole，可以将其授予扩展 apiserver 的服务帐户。

扩展 Apiserver 执行

如果 SubjectAccessReview 通过，则扩展 apiserver 执行请求。

启用 Kubernetes Apiserver 标志

通过以下 kube-apiserver 标志启用聚合层。你的服务提供商可能已经为你完成了这些工作：

```
--requestheader-client-ca-file=<path to aggregator CA cert>
--requestheader-allowed-names=front-proxy-client
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
--proxy-client-cert-file=<path to aggregator proxy cert>
--proxy-client-key-file=<path to aggregator proxy key>
```

CA-重用和冲突

Kubernetes apiserver 有两个客户端 CA 选项：

- --client-ca-file
- --requestheader-client-ca-file

这些功能中的每个功能都是独立的；如果使用不正确，可能彼此冲突。

- --client-ca-file：当请求到达 Kubernetes apiserver 时，如果启用了此选项，则 Kubernetes apiserver 会检查请求的证书。如果它是由 --client-ca-file 引用的文件中的 CA 证书之一签名的，并且用户是公用名 CN= 的值，

而组是组织O= 的取值，则该请求被视为合法请求。请参阅[关于 TLS 身份验证的文档](#)。

- --requestheader-client-ca-file：当请求到达 Kubernetes apiserver 时，如果启用此选项，则 Kubernetes apiserver 会检查请求的证书。如果它是由文件引用中的 --requestheader-client-ca-file 所签署的 CA 证书之一签名的，则该请求将被视为潜在的合法请求。然后，Kubernetes apiserver 检查通用名称 CN= 是否是 --requestheader-allowed-names 提供的列表中的名称之一。如果名称允许，则请求被批准；如果不是，则请求被拒绝。

如果同时提供了 --client-ca-file 和 --requestheader-client-ca-file，则首先检查 --requestheader-client-ca-file CA，然后再检查 --client-ca-file。通常，这些选项中的每一个都使用不同的 CA（根 CA 或中间 CA）。常规客户端请求与 --client-ca-file 相匹配，而聚合请求要与 --requestheader-client-ca-file 相匹配。但是，如果两者都使用同一个 CA，则通常会通过 --client-ca-file 传递的客户端请求将失败，因为 CA 将与 --requestheader-client-ca-file 中的 CA 匹配，但是通用名称 CN= 将不匹配 --requestheader-allowed-names 中可接受的通用名称之一。这可能导致你的 kubelet 和其他控制平面组件以及最终用户无法向 Kubernetes apiserver 认证。

因此，请对用于控制平面组件和最终用户鉴权的 --client-ca-file 选项和用于聚合 apiserver 鉴权的 --requestheader-client-ca-file 选项使用不同的 CA 证书。

警告：除非你了解风险和保护 CA 用法的机制，否则 不要重用在不同上下文中使用的 CA。

如果你未在运行 API 服务器的主机上运行 kube-proxy，则必须确保使用以下 kubectl 标志启用系统：

```
--enable-aggregator-routing=true
```

注册 APIService 对象

你可以动态配置将哪些客户端请求代理到扩展 apiserver。以下是注册示例：

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: <注释对象名称>
spec:
  group: <扩展 Apiserver 的 API 组名>
  version: <扩展 Apiserver 的 API 版本>
  groupPriorityMinimum: <APIService 对应组的优先级, 参考 API 文档>
  versionPriority: <版本在组中的优先排序, 参考 API 文档>
  service:
    namespace: <拓展 Apiserver 服务的名字空间>
    name: <拓展 Apiserver 服务的名称>
    caBundle: <PEM 编码的 CA 证书, 用于对 Webhook 服务器的证书签名>
```

APIService 对象的名称必须是合法的 [路径片段名称](#)。

调用扩展 apiserver

一旦 Kubernetes apiserver 确定应将请求发送到扩展 apiserver，它需要知道如何调用它。

service 部分是对扩展 apiserver 的服务的引用。服务的名字空间和名字是必需的。端口是可选的，默認為 443。路径配置是可选的，默認為 /。

下面是为可在端口 1234 上调用的扩展 apiserver 的配置示例 服务位于子路径 /my-path 下，并针对 ServerName my-service-name.my-service-namespace.svc 使用自定义的 CA 包来验证 TLS 连接 使用自定义 CA 绑定包的 my-service-name.my-service-namespace.svc。

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
...
spec:
  ...
    service:
      namespace: my-service-namespace
      name: my-service-name
      port: 1234
      caBundle: "Ci0tLS0tQk...<base64-encoded PEM bundle>...tLS0K"
  ...
...
```

接下来

- 使用聚合层[安装扩展 API 服务器](#)。
- 有关高级概述，请参阅[使用聚合层扩展 Kubernetes API](#)。
- 了解如何[使用自定义资源扩展 Kubernetes API](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

安装一个扩展的 API server

安装扩展的 API 服务器来使用聚合层以让 Kubernetes API 服务器使用其它 API 进行扩展，这些 API 不是核心 Kubernetes API 的一部分。

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.

- 你必须[配置聚合层](#) 并且启用 API 服务器的相关参数。

安装一个扩展的 API 服务器来使用聚合层

以下步骤描述如何 在一个高层次 设置一个扩展的 apiserver。无论你使用的是 YAML 配置还是使用 API，这些步骤都适用。目前我们正在尝试区分出两者的区别。有关使用 YAML 配置的具体示例，你可以在 Kubernetes 库中查看 [sample-apiserver](#)。

或者，你可以使用现有的第三方解决方案，例如 [apiserver-builder](#)，它将生成框架并自动执行以下所有步骤。

1. 确保启用了 APIService API (检查 `--runtime-config`)。默认应该是启用的，除非被特意关闭了。
2. 你可能需要制定一个 RBAC 规则，以允许你添加 APIService 对象，或让你的集群管理员创建一个。（由于 API 扩展会影响整个集群，因此不建议在实时集群中对 API 扩展进行测试/开发/调试）
3. 创建 Kubernetes 命名空间，扩展的 api-service 将运行在该命名空间中。
4. 创建（或获取）用来签署服务器证书的 CA 证书，扩展 api-server 中将使用该证书做 HTTPS 连接。
5. 为 api-server 创建一个服务端的证书（或秘钥）以使用 HTTPS。这个证书应该由上述的 CA 签署。同时应该还要有一个 Kube DNS 名称的 CN，这是从 Kubernetes 服务派生而来的，格式为 `<service name>.<service name namespace>.svc`。
6. 使用命名空间中的证书（或秘钥）创建一个 Kubernetes secret。
7. 为扩展 api-server 创建一个 Kubernetes Deployment，并确保以卷的方式挂载了 Secret。它应该包含对扩展 api-server 镜像的引用。Deployment 也应该在同一个命名空间中。
 1. 确保你的扩展 apiserver 从该卷中加载了那些证书，并在 HTTPS 握手过程中使用它们。
 2. 在你的命令空间中创建一个 Kubernetes 服务账号。
 3. 为资源允许的操作创建 Kubernetes 集群角色。

4. 用你命令空间中的服务账号创建一个 Kubernetes 集群角色绑定，绑定到你刚创建的角色上。
 5. 用你命令空间中的服务账号创建一个 Kubernetes 集群角色绑定，绑定到 system:auth-delegator 集群角色，以将 auth 决策委派给 Kubernetes 核心 API 服务器。
 6. 以你命令空间中的服务账号创建一个 Kubernetes 集群角色绑定，绑定到 extension-apiserver-authentication-reader 角色。这将让你的扩展 api-server 能够访问 extension-apiserver-authentication configmap。
1. 创建一个 Kubernetes apiservice。上述的 CA 证书应该使用 base64 编码，剥离新行并用作 apiservice 中的 spec.caBundle。该资源不应放到任何名字空间。如果使用了 [kube-aggregator API](#)，那么只需要传入 PEM 编码的 CA 绑定，因为 base 64 编码已经完成了。
 2. 使用 kubectl 来获得你的资源。它应该返回 "找不到资源"。此消息表示一切正常，但你目前还没有创建该资源类型的对象。

接下来

- 如果你尚未配置，请[配置聚合层](#) 并启用 apiserver 的相关参数。
- 高级概述，请参阅[使用聚合层扩展 Kubernetes API](#)。
- 了解如何[使用 Custom Resource Definition 扩展 Kubernetes API](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

配置多个调度器

Kubernetes 自带了一个默认调度器，其详细描述请查阅 [这里](#)。如果默认调度器不适合你的需求，你可以实现自己的调度器。不仅如此，你甚至可以和默认调度器一起同时运行多个调度器，并告诉 Kubernetes 为每个 Pod 使用哪个调度器。让我们通过一个例子讲述如何在 Kubernetes 中运行多个调度器。

关于实现调度器的具体细节描述超出了本文范围。请参考 kube-scheduler 的实现，规范示例代码位于 [pkg/scheduler](#)。

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.

打包调度器

将调度器可执行文件打包到容器镜像中。出于示例目的，我们就使用默认调度器（kube-scheduler）作为我们的第二个调度器。克隆 [Github 上 Kubernetes 源代码](#)，并编译构建源代码。

```
git clone https://github.com/kubernetes/kubernetes.git  
cd kubernetes  
make
```

创建一个包含 kube-scheduler 二进制文件的容器镜像。用于构建镜像的 Dockerfile 内容如下：

```
FROM busybox  
ADD ./_output/dockerized/bin/linux/amd64/kube-scheduler /usr/local/bin/  
kube-scheduler
```

将文件保存为 Dockerfile，构建镜像并将其推送到镜像仓库。此示例将镜像推送到 [Google 容器镜像仓库 \(GCR \)](#)。有关详细信息，请阅读 GCR [文档](#)。

```
docker build -t gcr.io/my-gcp-project/my-kube-scheduler:1.0 .  
gcloud docker -- push gcr.io/my-gcp-project/my-kube-scheduler:1.0
```

为调度器定义 Kubernetes Deployment

现在我们将调度器放在容器镜像中，我们可以为它创建一个 Pod 配置，并在我们的 Kubernetes 集群中运行它。但是与其在集群中直接创建一个 Pod，不如使用 [Deployment](#)。Deployment 管理一个 [ReplicaSet](#)，ReplicaSet 再管理 Pod，从而使调度器能够免受一些故障的影响。以下是 Deployment 配置，将其保存为 my-scheduler.yaml：

[admin/sched/my-scheduler.yaml](#)



```
apiVersion: v1  
kind: ServiceAccount  
metadata:  
  name: my-scheduler  
  namespace: kube-system  
---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
    name: my-scheduler
    namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
```

```
spec:  
  serviceAccountName: my-scheduler  
  containers:  
    - command:  
        - /usr/local/bin/kube-scheduler  
        - --address=0.0.0.0  
        - --leader-elect=false  
        - --scheduler-name=my-scheduler  
      image: gcr.io/my-gcp-project/my-kube-scheduler:1.0  
      livenessProbe:  
        httpGet:  
          path: /healthz  
          port: 10251  
        initialDelaySeconds: 15  
      name: kube-second-scheduler  
      readinessProbe:  
        httpGet:  
          path: /healthz  
          port: 10251  
      resources:  
        requests:  
          cpu: '0.1'  
      securityContext:  
        privileged: false  
      volumeMounts: []  
    hostNetwork: false  
    hostPID: false  
    volumes: []
```

这里需要注意的是，在容器规约中配置的调度器启动命令参数（`--scheduler-name`）所指定的 调度器名称应该是唯一的。这个名称应该与 Pod 上的可选参数 `spec.schedulerName` 的值相匹配，也就是说调度器名称的匹配关系决定了 Pods 的调度任务由哪个调度器负责。

还要注意，我们创建了一个专用服务账号 `my-scheduler` 并将集群角色 `system:kube-scheduler` 绑定到它，以便它可以获得与 `kube-scheduler` 相同的权限。

请参阅 [kube-scheduler 文档](#)以获取其他命令行参数的详细说明。

在集群中运行第二个调度器

为了在 Kubernetes 集群中运行我们的第二个调度器，只需在 Kubernetes 集群中创建上面配置中指定的 Deployment：

```
kubectl create -f my-scheduler.yaml
```

验证调度器 Pod 正在运行：

```
kubectl get pods --namespace=kube-system
```

输出类似于：

NAME	READY	STATUS	RESTARTS	AGE
...				
my-scheduler-lnf4s-4744f	1/1	Running	0	2m
...				

此列表中，除了默认的 kube-scheduler Pod 之外，你应该还能看到处于 "Running" 状态的 my-scheduler Pod。

启用领导者选举

要在启用了 leader 选举的情况下运行多调度器，你必须执行以下操作：

首先，更新上述 Deployment YAML (my-scheduler.yaml) 文件中的以下字段：

- --leader-elect=true
- --lock-object-namespace=lock-object-namespace
- --lock-object-name=lock-object-name

如果在集群上启用了 RBAC，则必须更新 system : kube-scheduler 集群角色。将调度器名称添加到应用于端点资源的规则的 resourceNames，如以下示例所示：

```
kubectl edit clusterrole system:kube-scheduler
```

[admin/sched/clusterrole.yaml](#)



```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-scheduler
rules:
- apiGroups:
  - coordination.k8s.io
resources:
- leases
verbs:
- create
- apiGroups:
```

```
- coordination.k8s.io
resourceNames:
- kube-scheduler
- my-scheduler
resources:
- leases
verbs:
- get
- update
- apiGroups:
- ""

resourceNames:
- kube-scheduler
- my-scheduler
resources:
- endpoints
verbs:
- delete
- get
- patch
- update
```

为 Pod 指定调度器

现在我们的第二个调度器正在运行，让我们创建一些 Pod，并指定它们由默认调度器或我们刚部署的 调度器进行调度。为了使用特定的调度器调度给定的 Pod，我们在那个 Pod 的规约中指定调度器的名称。让我们看看三个例子。

- Pod spec 没有任何调度器名称

[admin/sched/pod1.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: no-annotation
labels:
  name: multischeduler-example
spec:
  containers:
    - name: pod-with-no-annotation-container
      image: k8s.gcr.io/pause:2.0
```

如果未提供调度器名称，则会使用 default-scheduler 自动调度 pod。

将此文件另存为 pod1.yaml，并将其提交给 Kubernetes 集群。

```
kubectl create -f pod1.yaml
```

- Pod spec 设置为 default-scheduler

[admin/sched/pod2.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-default-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: default-scheduler
  containers:
    - name: pod-with-default-annotation-container
      image: k8s.gcr.io/pause:2.0
```

通过将调度器名称作为 spec.schedulerName 参数的值来指定调度器。在这种情况下，我们提供默认调度器的名称，即 default-scheduler。

将此文件另存为 pod2.yaml，并将其提交给 Kubernetes 集群。

```
kubectl create -f pod2.yaml
```

- Pod spec 设置为 my-scheduler

[admin/sched/pod3.yaml](#)



```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
    - name: pod-with-second-annotation-container
      image: k8s.gcr.io/pause:2.0
```

在这种情况下，我们指定此 pod 使用我们部署的 my-scheduler 来调度。请注意，spec.schedulerName 参数的值应该与 Deployment 中配置的提供给 scheduler 命令的参数名称匹配。

将此文件另存为 pod3.yaml，并将其提交给 Kubernetes 集群。

```
kubectl create -f pod3.yaml
```

确认所有三个 pod 都在运行。

```
kubectl get pods
```

验证是否使用所需的调度器调度了 pod

为了更容易地完成这些示例，我们没有验证 Pod 实际上是使用所需的调度程序调度的。 我们可以通过更改 Pod 的顺序和上面的部署配置提交来验证这一点。 如果我们在提交调度器部署配置之前将所有 Pod 配置提交给 Kubernetes 集群，我们将看到注解了 annotation-second-scheduler 的 Pod 始终处于 "Pending" 状态，而其他两个 Pod 被调度。 一旦我们提交调度器部署配置并且我们的新调度器开始运行，注解了 annotation-second-scheduler 的 pod 就能被调度。

或者，可以查看事件日志中的 "Scheduled" 条目，以验证是否由所需的调度器调度了 Pod。

```
kubectl get events
```

反馈

此页是否对您有帮助？

是 否

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

最后修改 August 13, 2020 at 5:49 PM PST: [\[zh\] Tidy up and fix links in tasks section \(5/10\) \(68abcb963\)](#)

使用 HTTP 代理访问 Kubernetes API

本文说明如何使用 HTTP 代理访问 Kubernetes API。

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`。
- 如果您的集群中还没有任何应用，使用如下命令启动一个 Hello World 应用：

```
kubectl create deployment node-hello --image=gcr.io/google-samples/node-hello:1.0 --port=8080
```

使用 `kubectl` 启动代理服务器

使用如下命令启动 Kubernetes API 服务器的代理：

```
kubectl proxy --port=8080
```

探究 Kubernetes API

当代理服务器在运行时，你可以通过 `curl`、`wget` 或者浏览器访问 API。

获取 API 版本：

```
curl http://localhost:8080/api/
```

输出应该类似这样：

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.2.15:8443"
    }
  ]
}
```

获取 Pod 列表：

```
curl http://localhost:8080/api/v1/namespaces/default/pods
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "33074"
  },
  "items": [

```

```
{  
  "metadata": {  
    "name": "kubernetes-bootcamp-2321272333-ix8pt",  
    "generateName": "kubernetes-bootcamp-2321272333-",  
    "namespace": "default",  
    "uid": "ba21457c-6b1d-11e6-85f7-1ef9f1dab92b",  
    "resourceVersion": "33003",  
    "creationTimestamp": "2016-08-25T23:43:30Z",  
    "labels": {  
      "pod-template-hash": "2321272333",  
      "run": "kubernetes-bootcamp"  
    },  
    ...  
  }  
}
```

接下来

想了解更多信息，请参阅 [kubectl 代理](#)。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 07, 2020 at 3:40 PM PST: [Update http-proxy-access-api.md \(99467aea3\)](#)

设置 Konnectivity 服务

Konnectivity 服务为控制平面提供集群通信的 TCP 级别代理。

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](#)

配置 Konnectivity 服务

接下来的步骤需要出口配置，比如：

[admin/konnectivity/egress-selector-configuration.yaml](#)



```
apiVersion: apiserver.k8s.io/v1beta1
kind: EgressSelectorConfiguration
egressSelections:
  # Since we want to control the egress traffic to the cluster, we use the
  # "cluster" as the name. Other supported values are "etcd", and "master".
  - name: cluster
    connection:
      # This controls the protocol between the API Server and the Konnectivity
      # server. Supported values are "GRPC" and "HTTPConnect". There is no
      # end user visible difference between the two modes. You need to set the
      # Konnectivity server to work in the same mode.
    proxyProtocol: GRPC
    transport:
      # This controls what transport the API Server uses to communicate with
      # the
      # Konnectivity server. UDS is recommended if the Konnectivity server
      # locates on the same machine as the API Server. You need to configure
      # the
      # Konnectivity server to listen on the same UDS socket.
      # The other supported transport is "tcp". You will need to set up TLS
      # config to secure the TCP transport.
    uds:
      udsName: /etc/srv/kubernetes/konnectivity-server/konnectivity-
server.socket
```

你需要配置 API 服务器来使用 Konnectivity 服务，并将网络流量定向到集群节点：

1. 确保 ServiceAccountTokenVolumeProjection 特性门控 被启用。你可以通过为 kube-apiserver 提供以下标志启用 [服务账号令牌卷保护](#)：

```
--service-account-issuer=api
--service-account-signing-key-file=/etc/kubernetes/pki/sa.key
--api-audiences=system:konnectivity-server
```

[admin/konnectivity/egress-selector-configuration.yaml](#)



```
apiVersion: apiserver.k8s.io/v1beta1
kind: EgressSelectorConfiguration
```

```

egressSelections:
# Since we want to control the egress traffic to the cluster, we use the
# "cluster" as the name. Other supported values are "etcd", and "master".
- name: cluster
connection:
  # This controls the protocol between the API Server and the Konnectivity
  # server. Supported values are "GRPC" and "HTTPConnect". There is no
  # end user visible difference between the two modes. You need to set the
  # Konnectivity server to work in the same mode.
proxyProtocol: GRPC
transport:
  # This controls what transport the API Server uses to communicate with
  # the
    # Konnectivity server. UDS is recommended if the Konnectivity server
    # locates on the same machine as the API Server. You need to configure
    # the
      # Konnectivity server to listen on the same UDS socket.
      # The other supported transport is "tcp". You will need to set up TLS
      # config to secure the TCP transport.
uds:
  udsName: /etc/srv/kubernetes/konnectivity-server/konnectivity-
server.socket

```

2. 创建一个出口配置文件比如 admin/konnectivity/egress-selector-configuration.yaml。 3. 将 API 服务器的 --egress-selector-config-file 参数设置为你的 API 服务器的 离站流量配置文件路径。

为 konnectivity-server 生成或者取得证书和 kubeconfig 文件。例如，你可以使用 OpenSSL 命令行工具，基于存放在某控制面主机上 /etc/kubernetes/pki/ca.crt 文件中的集群 CA 证书来发放一个 X.509 证书，

```

openssl req -subj "/CN=system:konnectivity-server" -new -newkey rsa:2048
-nodes -out konnectivity.csr -keyout konnectivity.key -out konnectivity.csr
openssl x509 -req -in konnectivity.csr -CA /etc/kubernetes/pki/ca.crt -
CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out konnectivity.crt -
days 375 -sha256
$ SERVER=$(kubectl config view -o jsonpath='{.clusters..server}')
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set-
credentials system:konnectivity-server --client-certificate konnectivity.crt --
client-key konnectivity.key --embed-certs=true
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set-
cluster kubernetes --server "$SERVER" --certificate-authority /etc/
kubernetes/pki/ca.crt --embed-certs=true
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config set-
context system:konnectivity-server@kubernetes --cluster kubernetes --user
system:konnectivity-server

```

```
kubectl --kubeconfig /etc/kubernetes/konnectivity-server.conf config use-context system:konnectivity-server@kubernetes  
rm -f konnectivity.crt konnectivity.key konnectivity.csr
```

接下来，你需要部署 Konnectivity 服务器和代理。 [kubernetes-sigs/apiserver-network-proxy](#) 是一个参考实现。

在控制面节点上部署 Konnectivity 服务。下面提供的 konnectivity-server.yaml 配置清单假定在你的集群中 Kubernetes 组件都是部署为[静态 Pod](#) 的。如果不是，你可以将 Konnectivity 服务部署为 DaemonSet。

[admin/konnectivity/konnectivity-server.yaml](#)



```
apiVersion: v1  
kind: Pod  
metadata:  
  name: konnectivity-server  
  namespace: kube-system  
spec:  
  priorityClassName: system-cluster-critical  
  hostNetwork: true  
  containers:  
    - name: konnectivity-server-container  
      image: us.gcr.io/k8s-artifacts-prod/kas-network-proxy/proxy-server:v0.0.8  
      command: ["/proxy-server"]  
      args: [  
        "--log-file=/var/log/konnectivity-server.log",  
        "--logtostderr=false",  
        "--log-file-max-size=0",  
        # This needs to be consistent with the value set in  
        egressSelectorConfiguration.  
        "--uds-name=/etc/srv/kubernetes/konnectivity-server/konnectivity-  
server.socket",  
        # The following two lines assume the Konnectivity server is  
        # deployed on the same machine as the apiserver, and the certs and  
        # key of the API Server are at the specified location.  
        "--cluster-cert=/etc/srv/kubernetes/pki/apiserver.crt",  
        "--cluster-key=/etc/srv/kubernetes/pki/apiserver.key",  
        # This needs to be consistent with the value set in  
        egressSelectorConfiguration.  
        "--mode=grpc",  
        "--server-port=0",  
        "--agent-port=8132",  
        "--admin-port=8133",  
        "--agent-namespace=kube-system",  
        "--agent-service-account=konnectivity-agent",
```

```

    "--kubeconfig=/etc/srv/kubernetes/konnectivity-server/kubeconfig",
    "--authentication-audience=system:konnectivity-server"
]
livenessProbe:
  httpGet:
    scheme: HTTP
    host: 127.0.0.1
    port: 8133
    path: /healthz
  initialDelaySeconds: 30
  timeoutSeconds: 60
ports:
- name: agentport
  containerPort: 8132
  hostPort: 8132
- name: adminport
  containerPort: 8133
  hostPort: 8133
volumeMounts:
- name: varlogkonnectivityserver
  mountPath: /var/log/konnectivity-server.log
  readOnly: false
- name: pki
  mountPath: /etc/srv/kubernetes/pki
  readOnly: true
- name: konnectivity-uds
  mountPath: /etc/srv/kubernetes/konnectivity-server
  readOnly: false
volumes:
- name: varlogkonnectivityserver
  hostPath:
    path: /var/log/konnectivity-server.log
    type: FileOrCreate
- name: pki
  hostPath:
    path: /etc/srv/kubernetes/pki
- name: konnectivity-uds
  hostPath:
    path: /etc/srv/kubernetes/konnectivity-server
    type: DirectoryOrCreate

```

在你的集群中部署 Konnectivity 代理：

[admin/konnectivity/konnectivity-agent.yaml](#)



```
apiVersion: apps/v1
# Alternatively, you can deploy the agents as Deployments. It is not
necessar
# to have an agent on each node.
kind: DaemonSet
metadata:
  labels:
    addonmanager.kubernetes.io/mode: Reconcile
    k8s-app: konnectivity-agent
  namespace: kube-system
  name: konnectivity-agent
spec:
  selector:
    matchLabels:
      k8s-app: konnectivity-agent
  template:
    metadata:
      labels:
        k8s-app: konnectivity-agent
    spec:
      priorityClassName: system-cluster-critical
      tolerations:
        - key: "CriticalAddonsOnly"
          operator: "Exists"
      containers:
        - image: us.gcr.io/k8s-artifacts-prod/kas-network-proxy/proxy-
agent:v0.0.8
          name: konnectivity-agent
          command: ["/proxy-agent"]
          args: [
            "--logtostderr=true",
            "--ca-cert=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt",
            # Since the konnectivity server runs with hostNetwork=true,
            # this is the IP address of the master machine.
            "--proxy-server-host=35.225.206.7",
            "--proxy-server-port=8132",
            "--service-account-token-path=/var/run/secrets/tokens/
konnectivity-agent-token"
          ]
      volumeMounts:
        - mountPath: /var/run/secrets/tokens
          name: konnectivity-agent-token
      livenessProbe:
        httpGet:
          port: 8093
          path: /healthz
```

```
initialDelaySeconds: 15
timeoutSeconds: 15
serviceAccountName: konnectivity-agent
volumes:
- name: konnectivity-agent-token
projected:
sources:
- serviceAccountToken:
  path: konnectivity-agent-token
  audience: system:konnectivity-server
```

最后，如果你的集群启用了 RBAC，请创建相关的 RBAC 规则：

[admin/konnectivity/konnectivity-rbac.yaml](#)


```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:konnectivity-server
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: system:konnectivity-server
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: konnectivity-agent
  namespace: kube-system
  labels:
    kubernetes.io/cluster-service: "true"
    addonmanager.kubernetes.io/mode: Reconcile
```

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 20, 2020 at 2:08 PM PST: [\[zh\] sync tasks/extend-kubernetes/setup-konnectivity.md \(7ba789171\)](#)

TLS

了解如何使用传输层安全性（ TLS ）保护集群中的流量。

[为 kubelet 配置证书轮换](#)

[手动轮换 CA 证书](#)

[管理集群中的 TLS 认证](#)

为 kubelet 配置证书轮换

本文展示如何在 kubelet 中启用并配置证书轮换。

FEATURE STATE: Kubernetes v1.19 [stable]

准备开始

- 要求 Kubernetes 1.8.0 或更高的版本

概述

Kubelet 使用证书进行 Kubernetes API 的认证。 默认情况下，这些证书的签发期限为一年，所以不需要太频繁地进行更新。

Kubernetes 1.8 版本中包含 beta 特性 [kubelet 证书轮换](#)，在当前证书即将过期时，将自动生成新的秘钥，并从 Kubernetes API 申请新的证书。一旦新的证书可用，它将被用于与 Kubernetes API 间的连接认证。

启用客户端证书轮换

kubelet 进程接收 --rotate-certificates 参数，该参数决定 kubelet 在当前使用的证书即将到期时，是否会自动申请新的证书。

kube-controller-manager 进程接收 --cluster-signing-duration 参数（在 1.19 版本之前为 --experimental-cluster-signing-duration ），用来 控制签发证书的有效期限。

理解证书轮换配置

当 kubelet 启动时，如被配置为自举（使用`--bootstrap-kubeconfig`参数），kubelet 会使用其初始证书连接到 Kubernetes API，并发送证书签名的请求。可以通过以下方式查看证书签名请求的状态：

```
kubectl get csr
```

最初，来自节点上 kubelet 的证书签名请求处于 Pending 状态。如果证书签名请求满足特定条件，控制器管理器会自动批准，此时请求会处于 Approved 状态。接下来，控制器管理器会签署证书，证书的有效期限由 `--cluster-signing-duration` 参数指定，签署的证书会被附加到证书签名请求中。

Kubelet 会从 Kubernetes API 取回签署的证书，并将其写入磁盘，存储位置通过 `--cert-dir` 参数指定。然后 kubelet 会使用新的证书连接到 Kubernetes API。

当签署的证书即将到期时，kubelet 会使用 Kubernetes API，发起新的证书签名请求。同样地，控制器管理器会自动批准证书请求，并将签署的证书附加到证书签名请求中。Kubelet 会从 Kubernetes API 取回签署的证书，并将其写入磁盘。然后它会更新与 Kubernetes API 的连接，使用新的证书重新连接到 Kubernetes API。

反馈

此页是否对您有帮助？

是 否

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

最后修改 November 23, 2020 at 10:24 AM PST: [\[zh\] Sync changes from English site \(11\) \(aca3e081f\)](#)

手动轮换 CA 证书

本页展示如何手动轮换证书机构（CA）证书。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 v1.13. 要获知版本信息，请输入 kubectl version.

- 要了解 Kubernetes 中用户认证的更多信息，参阅 [认证](#)；
- 要了解与 CA 证书最佳实践有关的更多信息，参阅[单根 CA](#)。

手动轮换 CA 证书

注意：

确保备份你的证书目录、配置文件以及其他必要文件。

这里的方法假定 Kubernetes 的控制面通过运行多个 API 服务器以高可用配置模式运行。另一假定是 API 服务器可体面地终止，因而客户端可以彻底地与一个 API 服务器断开连接并连接到另一个 API 服务 器。

如果集群中只有一个 API 服务器，则在 API 服务器重启期间会经历服 务中断期。

1. 将新的 CA 证书和私钥（例如：ca.crt、ca.key、front-proxy-ca.crt 和 front -proxy-client.key）分发到所有控制面节点，放在其 Kubernetes 证书目录 下。
1. 更新 [kube-controller-manager](#) 的 --root-ca-file 标志，使之同时包含老的 和新的 CA，之后重启组件。

自此刻起，所创建的所有服务账号都会获得同时包含老的 CA 和新的 CA 的 Secret。

说明： kube-controller-manager 标志 --client-ca-file 和 -- cluster-signing-cert-file 所引用的文件 不能是 CA 证书包。如果 这些标志和 --root-ca-file 指向同一个 ca.crt 包文件（包含老的和 新的 CA 证书），你将会收到出错信息。要解决这个问题，可以 将新的 CA 证书复制到单独的文件中，并将 --client-ca-file 和 -- cluster-signing-cert-file 标志指向该副本。一旦 ca.crt 不再是证 书包文件，就可以恢复有问题的标志指向 ca.crt 并删除该副本。

1. 更新所有服务账号令牌，使之同时包含老的和新的 CA 证书。

如果在 API 服务器使用新的 CA 之前启动了新的 Pod，这些 Pod 也会获得 此更新并且同时信任老的和新的 CA 证书。

```
base64_encoded_ca="$(base64 <path to file containing both old and new CAs>)"
```

```
for namespace in $(kubectl get ns --no-headers | awk '{print $1}'); do
    for token in $(kubectl get secrets --namespace "$namespace" --
field-selector type=kubernetes.io/service-account-token -o name); do
```

```
kubectl get $token --namespace "$namespace" -o yaml | \
/bin/sed "s/\(ca.crt\).*\$/\1 ${base64_encoded_ca}" | \
kubectl apply -f -
done
done
```

1. 重启所有使用集群内配置的 Pods (例如 : kube-proxy、coredns 等) , 以便这些 Pod 能够使用来自 *ServiceAccount Secret* 中的、已更新的证书机构数据。
 - 确保 coredns、kube-proxy 和其他使用集群内配置的 Pod 都正按预期方式工作。
2. 将老的和新的 CA 都追加到 kube-apiserver 配置的 --client-ca-file 和 --kubelet-certificate-authority 标志所指的文件。
3. 将老的和新的 CA 都追加到 kube-scheduler 配置的 --client-ca-file 标志所指的文件。
1. 通过替换 client-certificate-data 和 client-key-data 中的内容 , 更新用户账号的证书。

有关为独立用户账号创建证书的更多信息 , 可参阅 [为用户帐号配置证书](#)。

另外 , 还要更新 kubeconfig 文件中的 certificate-authority-data 节 , 使之包含 Base64 编码的老的和新的证书机构数据。

1. 遵循下列步骤执行滚动更新

1. 重新启动所有其他 [被聚合的 API 服务器](#) 或者 Webhook 处理程序 , 使之信任新的 CA 证书。
2. 在所有节点上更新 kubelet 配置中的 clientCAFile 所指文件以及 kubelet.conf 中的 certificate-authority-data 并重启 kubelet 以同时使用老的和新的 CA 证书。

如果你的 kubelet 并未使用客户端证书轮换 , 则在所有节点上更新 kubelet.conf 中 client-certificate-data 和 client-key-data 以及 kubelet 客户端证书文件 (通常位于 /var/lib/kubelet/pki 目录下)

1. 使用用新的 CA 签名的证书 (apiserver.crt、apiserver-kubelet-client.crt 和 front-proxy-client.crt) 来重启 API 服务器。你可以使用现有的私钥 , 也可以使用新的私钥。如果你改变了私钥 , 则要将更新的私钥也放到 Kubernetes 证书目录下。

由于 Pod 既信任老的 CA 也信任新的 CA , Pod 中的客户端会经历短暂的连接断开状态 , 之后再连接到使用新的 CA 所签名的证书的新的 API 服务器。

- 重启调度器以使用新的 CA 证书。

- 确保控制面组件的日志中没有 TLS 相关的错误信息。

说明：要使用 openssl 命令为集群生成新的证书和私钥，可参阅 [证书 \(openssl\)](#)。你也可以使用[cfssl](#)。

1. 为 Daemonset 和 Deployment 添加注解，从而触发较安全的滚动更新，替换 Pod。

示例：

```
for namespace in $(kubectl get namespace -o jsonpath='{.items[*].metadata.name}'); do
    for name in $(kubectl get deployments -n $namespace -o jsonpath='{.items[*].metadata.name}'); do
        kubectl patch deployment -n ${namespace} ${name} -p '{"spec":{"template":{"metadata":{"annotations":{"ca-rotation": "1"}}}}';
    done
    for name in $(kubectl get daemonset -n $namespace -o jsonpath='{.items[*].metadata.name}'); do
        kubectl patch daemonset -n ${namespace} ${name} -p '{"spec":{"template":{"metadata":{"annotations":{"ca-rotation": "1"}}}}';
    done
done
```

说明：要限制应用可能受到的并发干扰数量，可以参阅 [配置 Pod 干扰预算](#)。

1. 如果你的集群使用启动引导令牌来添加节点，则需要更新 kube-public 名字空间下的 ConfigMap cluster-info，使之包含新的 CA 证书。

```
base64_encoded_ca="$(base64 /etc/kubernetes/pki/ca.crt)"

kubectl get cm/cluster-info --namespace kube-public -o yaml | \
/bin/sed "s/\(certificate-authority-data\).*/\1 ${base64_encoded_ca}" \
| \
kubectl apply -f -
```

1. 验证集群的功能正常

1. 验证控制面组件的日志，以及 kubelet 和 kube-proxy 的日志，确保其中没有抛出 TLS 错误，参阅 [查看日志](#)。

- 验证被聚合的 API 服务器的日志，以及所有使用集群内配置的 Pod 的
- 日志。

1. 完成集群功能的检查之后：

- 更新所有的服务账号令牌，使之仅包含新的 CA 证书。
 - 使用集群内 kubeconfig 的 Pod 最终也需要被重启，以获得新的服务账号 Secret 数据，进而不再信任老的 CA 证书。
- 从 kubeconfig 文件和 --client-ca-file 以及 --root-ca-file 标志所指向的文件中去除老的 CA 数据，之后重启控制面组件。
- 重启 kubelet，移除 clientCAFile 标志所指向的文件以及 kubelet kubeconfig 文件中的老的 CA 数据。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 06, 2020 at 11:00 PM PST: [zh-trans Update manual-rotation-of-ca-certificates.md \(653d4b69b\)](#)

管理集群中的 TLS 认证

Kubernetes 提供 certificates.k8s.io API，可让你配置由你控制的证书颁发机构（CA）签名的 TLS 证书。你的工作负载可以使用这些 CA 和证书来建立信任。

certificates.k8s.io API 使用的协议类似于 [ACME 草案](#)。

说明：

使用 certificates.k8s.io API 创建的证书由指定 CA 颁发。将集群配置为使用集群根目录 CA 可以达到这个目的，但是你永远不要依赖这一假定。不要以为这些证书将针对群根目录 CA 进行验证。

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`.

集群中的 TLS 信任

信任 Pod 中运行的应用程序所提供的 CA 通常需要一些额外的应用程序配置。你需要将 CA 证书包添加到 TLS 客户端或服务器信任的 CA 证书列表中。例如，你可以使用 Golang TLS 配置通过解析证书链并将解析的证书添加到 [tls.Config](#) 结构中的 `RootCAs` 字段中。

你可以用你的应用能够访问到的 [ConfigMap](#) 的形式来发布 CA 证书。

请求证书

以下部分演示如何为通过 DNS 访问的 Kubernetes 服务创建 TLS 证书。

说明：本教程使用 CFSSL : Cloudflare's PKI 和 TLS 工具包 [点击此处](#) 了解更多信息。

下载并安装 CFSSL

本例中使用的 cfssl 工具可以在 github.com/cloudflare/cfssl/releases 下载。

创建证书签名请求

通过运行以下命令生成私钥和证书签名请求（或 CSR）：

```
cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "192.0.2.24",
    "10.0.34.2"
  ],
  "CN": "system:node:my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {
      "O": "system:nodes"
    }
  ]
}
```

```
}
```

EOF

其中 192.0.2.24 是服务的集群 IP , my-svc.my-namespace.svc.cluster.local 是服务的 DNS 名称 , 10.0.34.2 是 Pod 的 IP , 而 my-pod.my-namespace.pod.cluster.local 是 Pod 的 DNS 名称。 你能看到以下的输出 :

```
2017/03/21 06:48:17 [INFO] generate received request
2017/03/21 06:48:17 [INFO] received CSR
2017/03/21 06:48:17 [INFO] generating key: ecdsa-256
2017/03/21 06:48:17 [INFO] encoded CSR
```

此命令生成两个文件 ; 它生成包含 PEM 编码 [pkcs#10](#) 证书请求的 server.csr , 以及 PEM 编码密钥的 server-key.pem , 用于待生成的证书。

创建证书签名请求对象发送到 Kubernetes API

使用以下命令创建 CSR YAML 文件 , 并发送到 API 服务器 :

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  request: $(cat server.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kubelet-serving
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

请注意 , 在步骤 1 中创建的 server.csr 文件是 base64 编码并存储在 .spec.request 字段中的。我们还要求提供 "digital signature (数字签名) " , "密钥加密 (key encipherment) " 和 "服务器身份验证 (server auth) " 密钥用途 , 由 kubernetes.io/kubelet-serving 签名程序签名的证书。 你也可以要求使用特定的 signerName。更多信息可参阅 [支持的签署者名称](#)。

在 API server 中可以看到这些 CSR 处于 Pending 状态。执行下面的命令你将可以看到 :

```
kubectl describe csr my-svc.my-namespace
```

Name:	my-svc.my-namespace
Labels:	<none>
Annotations:	<none>

```
CreationTimestamp: Tue, 21 Mar 2017 07:03:51 -0700
Requesting User: yourname@example.com
Status: Pending
Subject:
  Common Name: my-svc.my-namespace.svc.cluster.local
  Serial Number:
Subject Alternative Names:
  DNS Names: my-svc.my-namespace.svc.cluster.local
  IP Addresses: 192.0.2.24
    10.0.34.2
Events: <none>
```

批准证书签名请求

批准证书签名请求是通过自动批准过程完成的，或由集群管理员一次性完成。有关这方面涉及的更多信息，请参见下文。

下载证书并使用它

CSR 被签署并获得批准后，你应该看到以下内容：

```
kubectl get csr
```

NAME	AGE	REQUESTOR	CONDITION
my-svc.my-namespace	10m	yourname@example.com	
		Approved,Issued	

你可以通过运行以下命令下载颁发的证书并将其保存到 server.crt 文件中：

```
kubectl get csr my-svc.my-namespace -o jsonpath='{.status.certificate}' \
| base64 --decode > server.crt
```

现在你可以将 server.crt 和 server-key.pem 作为键值对来启动 HTTPS 服务器。

批准证书签名请求

Kubernetes 管理员（具有适当权限）可以使用 kubectl certificate approve 和 kubectl certificate deny 命令手动批准（或拒绝）证书签名请求。但是，如果你打算大量使用此 API，则可以考虑编写自动化的证书控制器。

无论上述机器或人使用 kubectl，批准者的作用是验证 CSR 满足如下两个要求：

1. CSR 的 subject 控制用于签署 CSR 的私钥。这解决了伪装成授权主体的第三方的威胁。在上述示例中，此步骤将验证该 Pod 控制了用于生成 CSR 的私钥。

2. CSR 的 subject 被授权在请求的上下文中执行。这点用于处理不期望的主体被加入集群的威胁。在上述示例中，此步骤将是验证该 Pod 是否被允许加入到所请求的服务中。

当且仅当满足这两个要求时，审批者应该批准 CSR，否则拒绝 CSR。

关于批准权限的警告

批准 CSR 的能力决定了群集中的信任关系。这也包括 Kubernetes API 所信任的人。批准 CSR 的能力不能过于广泛和轻率。在给予本许可之前，应充分了解上一节中提到的挑战和发布特定证书的后果。

给集群管理员的一个建议

本教程假设已经为 certificates API 配置了签名者。Kubernetes 控制器管理器提供了一个签名者的默认实现。要启用它，请为控制器管理器设置 `--cluster-signing-cert-file` 和 `--cluster-signing-key-file` 参数，使之取值为你的证书机构的密钥对的路径。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 19, 2021 at 1:18 AM PST: [\[zh\] upgrade cfssl installations \(009fd0f66\)](#)

管理集群守护进程

执行 DaemonSet 管理的常见任务，例如执行滚动更新。

[对 DaemonSet 执行滚动更新](#)

[对 DaemonSet 执行回滚](#)

对 DaemonSet 执行滚动更新

本文介绍了如何对 DaemonSet 执行滚动更新。

准备开始

- Kubernetes 1.6 或者更高版本中才支持 DaemonSet 滚动更新功能。

DaemonSet 更新策略

DaemonSet 有两种更新策略：

- OnDelete: 使用 OnDelete 更新策略时，在更新 DaemonSet 模板后，只有当你手动删除老的 DaemonSet pods 之后，新的 DaemonSet Pod 才会被自动创建。跟 Kubernetes 1.6 以前的版本类似。
- RollingUpdate: 这是默认的更新策略。使用 RollingUpdate 更新策略时，在更新 DaemonSet 模板后，老的 DaemonSet pods 将被终止，并且将以受控方式自动创建新的 DaemonSet pods。更新期间，最多只能有一个 DaemonSet 的一个 Pod 运行于每个节点上。

执行滚动更新

要启用 DaemonSet 的滚动更新功能，必须设置 `.spec.updateStrategy.type` 为 `RollingUpdate`。

你可能想设置 `.spec.updateStrategy.rollingUpdate.maxUnavailable` (默认为 1) 和 `.spec.minReadySeconds` (默认为 0)。

创建带有 RollingUpdate 更新策略的 DaemonSet

下面的 YAML 包含一个 DaemonSet，其更新策略为 'RollingUpdate'：

[controllers/fluentd-daemonset.yaml](#)



```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
```

```
template:
  metadata:
    labels:
      name: fluentd-elasticsearch
  spec:
    tolerations:
      # this toleration is to have the daemonset runnable on master nodes
      # remove it if your masters can't run pods
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
    containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        volumeMounts:
          - name: varlog
            mountPath: /var/log
          - name: varlibdockercontainers
            mountPath: /var/lib/docker/containers
            readOnly: true
    terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
```

检查了 DaemonSet 清单中更新策略的设置之后，创建 DaemonSet：

```
kubectl create -f https://k8s.io/examples/controllers/fluentd-
daemonset.yaml
```

另一种方式是如果你希望使用 kubectl apply 来更新 DaemonSet 的话，也可以使用 kubectl apply 来创建 DaemonSet：

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml
```

检查 DaemonSet 的滚动更新策略

首先，检查 DaemonSet 的更新策略，确保已经将其设置为 RollingUpdate：

```
kubectl get ds/fluentd-elasticsearch -o go-template='{{.spec.updateStrategy.
type}}{{"\n"}}' -n kube-system
```

如果还没在系统中创建 DaemonSet，请使用以下命令检查 DaemonSet 的清单：

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml --dry-run=client -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
```

两个命令的输出都应该为：

RollingUpdate

如果输出不是 RollingUpdate，请返回并相应地修改 DaemonSet 对象或者清单。

更新 DaemonSet 模板

对 RollingUpdate DaemonSet 的 .spec.template 的任何更新都将触发滚动更新。这可以通过几个不同的 kubectl 命令来完成。

[controllers/fluentd-daemonset-update.yaml](#)


```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
        # this toleration is to have the daemonset runnable on master nodes
        # remove it if your masters can't run pods
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
          resources:
```

```
limits:
  memory: 200Mi
  requests:
    cpu: 100m
    memory: 200Mi
  volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers
```

声明式命令

如果你使用 [配置文件](#) 来更新 DaemonSet，请使用 kubectl apply:

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset-update.yaml
```

指令式命令

如果你使用 [指令式命令](#) 来更新 DaemonSets，请使用 kubectl edit :

```
kubectl edit ds/fluentd-elasticsearch -n kube-system
```

只更新容器镜像

如果你只需要更新 DaemonSet 模板里的容器镜像，比如，.spec.template.spec.containers[*].image, 请使用 kubectl set image:

```
kubectl set image ds/fluentd-elasticsearch fluentd-elasticsearch=quay.io/fluentd_elasticsearch/fluentd:v2.6.0 -n kube-system
```

监视滚动更新状态

最后，观察 DaemonSet 最新滚动更新的进度：

```
kubectl rollout status ds/fluentd-elasticsearch -n kube-system
```

当滚动更新完成时，输出结果如下：

```
daemonset "fluentd-elasticsearch" successfully rolled out
```

故障排查

DaemonSet 滚动更新卡住

有时，DaemonSet 滚动更新可能卡住，以下是一些可能的原因：

一些节点可用资源耗尽

DaemonSet 滚动更新可能会卡住，其 Pod 至少在某个节点上无法调度运行。当节点上[可用资源耗尽](#)时，这是可能的。

发生这种情况时，通过对 kubectl get nodes 和下面命令行的输出作比较，找出没有调度部署 DaemonSet Pods 的节点：

```
kubectl get pods -l name=fluentd-elasticsearch -o wide -n kube-system
```

一旦找到这些节点，从节点上删除一些非 DaemonSet Pod，为新的 DaemonSet Pod 腾出空间。

说明：当所删除的 Pod 不受任何控制器管理，也不是多副本的 Pod 时，上述操作将导致服务中断。同时，上述操作也不会考虑 [PodDisruptionBudget](#) 所施加的约束。

不完整的滚动更新

如果最近的 DaemonSet 模板更新被破坏了，比如，容器处于崩溃循环状态或者容器镜像不存在（通常由于拼写错误），就会发生 DaemonSet 滚动更新中断。

要解决此问题，只需再次更新 DaemonSet 模板即可。以前不健康的滚动更新不会阻止新的滚动更新。

时钟偏差

如果在 DaemonSet 中指定了 `.spec.minReadySeconds`，主控节点和工作节点之间的时钟偏差会使 DaemonSet 无法检测到正确的滚动更新进度。

清理

从名字空间中删除 DaemonSet：

```
kubectl delete ds fluentd-elasticsearch -n kube-system
```

接下来

- 查看[任务](#)：在 DaemonSet 上执行回滚
- 查看[概念](#)：创建 DaemonSet 以收养现有 DaemonSet Pod

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 16, 2020 at 2:34 PM PST: [t # This is a combination of 3 commits. \(ba84d93eb\)](#)

对 DaemonSet 执行回滚

本文展示了如何对 [DaemonSet](#) 执行回滚。

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](#)

您的 Kubernetes 服务器版本必须不低于版本 1.7. 要获知版本信息，请输入 kubectl version.

你应该已经了解如何[为 DaemonSet 执行滚动更新](#)。

对 DaemonSet 执行回滚

步骤 1：找到想要 DaemonSet 回滚到的历史修订版本 (revision)

如果只想回滚到最后一个版本，可以跳过这一步。

列出 DaemonSet 的所有版本：

```
kubectl rollout history daemonset <daemonset-name>
```

此命令返回 DaemonSet 版本列表：

```
daemonsets "<daemonset-name>"  
REVISION      CHANGE-CAUSE  
1            ...  
2            ...  
...          ...
```

- 在创建时，DaemonSet 的变化原因从 kubernetes.io/change-cause 注解（annotation）复制到其修订版本中。用户可以在 kubectl 命令中设置 --record=true，将执行的命令记录在变化原因注解中。

执行以下命令，来查看指定版本的详细信息：

```
kubectl rollout history daemonset <daemonset-name> --revision=1
```

该命令返回相应修订版本的详细信息：

```
daemonsets "<daemonset-name>" with revision #1  
Pod Template:  
Labels:    foo=bar  
Containers:  
app:  
Image:    ...  
Port:     ...  
Environment: ...  
Mounts:   ...  
Volumes:  ...
```

步骤 2：回滚到指定版本

```
# 在 --to-revision 中指定你从步骤 1 中获取的修订版本  
kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>
```

如果成功，命令会返回：

```
daemonset "<daemonset-name>" rolled back
```

说明：如果 --to-revision 参数未指定，将选中最近的版本。

步骤 3：监视 DaemonSet 回滚进度

kubectl rollout undo daemonset 向服务器表明启动 DaemonSet 回滚。真正的回滚是在集群的 [控制面](#) 异步完成的。

执行以下命令，来监视 DaemonSet 回滚进度：

```
kubectl rollout status ds/<daemonset-name>
```

回滚完成时，输出形如：

```
daemonset "<daemonset-name>" successfully rolled out
```

理解 DaemonSet 修订版本

在前面的 `kubectl rollout history` 步骤中，你获得了一个修订版本列表，每个修订版本都存储在名为 ControllerRevision 的资源中。

要查看每个修订版本中保存的内容，可以找到 DaemonSet 修订版本的原生资源：

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-selector-value>
```

该命令返回 ControllerRevisions 列表：

NAME	CONTROLLER	REVISION	AGE
<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>		
1	1h		
<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>		
2	1h		

每个 ControllerRevision 中存储了相应 DaemonSet 版本的注解和模板。

`kubectl rollout undo` 选择特定的 ControllerRevision，并用 ControllerRevision 中存储的模板代替 DaemonSet 的模板。`kubectl rollout undo` 相当于通过其他命令（如 `kubectl edit` 或 `kubectl apply`）将 DaemonSet 模板更新至先前的版本。

说明：注意 DaemonSet 修订版本只会正向变化。也就是说，回滚完成后，所回滚到的 ControllerRevision 版本号（`.revision` 字段）会增加。例如，如果用户在系统中有版本 1 和版本 2，并从版本 2 回滚到版本 1，带有 `.revision: 1` 的 ControllerRevision 将变为 `.revision: 3`。

故障排查

- 参阅 [DaemonSet 滚动升级故障排除](#)。

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 August 16, 2020 at 8:50 PM PST: [\[zh\] Tidy up and fix links in tasks section \(10/10\) \(114a75d78\)](#)

安装服务目录

安装服务目录扩展 API。

[使用 Helm 安装 Service Catalog](#)

[使用 SC 安装服务目录](#)

使用 Helm 安装 Service Catalog

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

服务目录可以检索、供应、和绑定由 [服务代理人 \(Service Brokers\)](#) 提供的外部 [托管服务 \(Managed Services\)](#)，而无需知道那些服务具体是怎样创建和托管的。

使用 [Helm](#) 在 Kubernetes 集群上安装 Service Catalog。要获取有关此过程的最新信息，请浏览 [kubernetes-incubator/service-catalog](#) 仓库。

准备开始

- 理解[服务目录](#)的关键概念。
- Service Catalog 需要 Kubernetes 集群版本在 1.7 或更高版本。
- 你必须启用 Kubernetes 集群的 DNS 功能。
 - 如果使用基于云的 Kubernetes 集群或 [Minikube](#)，则可能已经启用了集群 DNS。
 - 如果你正在使用 `hack/local-up-cluster.sh`，请确保设置了 `KUBE_ENABLE_CLUSTER_DNS` 环境变量，然后运行安装脚本。
- [安装和设置 v1.7 或更高版本的 kubectl](#)，确保将其配置为连接到 Kubernetes 集群。
- 安装 v2.7.0 或更高版本的 [Helm](#)。
 - 遵照 [Helm 安装说明](#)。
 - 如果已经安装了适当版本的 Helm，请执行 `helm init` 来安装 Helm 的服务器端组件 Tiller。

添加 service-catalog Helm 仓库

安装 Helm 后，通过执行以下命令将 `service-catalog` Helm 存储库添加到本地计算机：

```
helm repo add svc-cat https://svc-catalog-charts.storage.googleapis.com
```

通过执行以下命令进行检查，以确保安装成功：

```
helm search service-catalog
```

如果安装成功，该命令应输出以下内容：

NAME	VERSION	DESCRIPTION
svc-cat/catalog	0.0.1	service-catalog API server and controller-manag...

启用 RBAC

你的 Kubernetes 集群必须启用 RBAC，这需要你的 Tiller Pod 具有 cluster-admin 访问权限。

如果你使用的是 Minikube，请使用以下参数运行 minikube start 命令：

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

如果你使用 hack/local-up-cluster.sh，请使用以下值设置 AUTHORIZATION_MODE 环境变量：

```
AUTHORIZATION_MODE=Node,RBAC hack/local-up-cluster.sh -O
```

默认情况下，helm init 将 Tiller Pod 安装到 kube-system 命名空间，Tiller 配置为使用 default 服务帐户。

说明：如果在运行 helm init 时使用了 --tiller-namespace 或 --service-account 参数，则需要调整以下命令中的 --serviceaccount 参数以引用相应的名字空间和服务账号名称。

配置 Tiller 以获得 cluster-admin 访问权限：

```
kubectl create clusterrolebinding tiller-cluster-admin \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:default
```

在 Kubernetes 集群中安装 Service Catalog

使用以下命令从 Helm 存储库的根目录安装 Service Catalog：

- [Helm version 3](#)
- [Helm version 2](#)

```
helm install catalog svc-cat/catalog --namespace catalog
```

```
helm install svc-cat/catalog --name catalog --namespace catalog
```

接下来

- 查看[示例服务代理](#)。
- 探索 [kubernetes-incubator/service-catalog](#) 项目。

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 26, 2020 at 8:20 PM PST: [Update link in page install-service-catalog-using-helm.md \(428fe6f90\)](#)

使用 SC 安装服务目录

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

服务目录可以检索、供应、和绑定由 [服务代理人（Service Brokers）](#) 提供的外部 [托管服务（Managed Services）](#)，而无需知道那些服务具体是怎样创建和托管的。

使用[服务目录安装程序](#)工具可以轻松地在 Kubernetes 集群上安装或卸载服务目录。这个 CLI 工具以 sc 命令形式被安装在您的本地环境中。

准备开始

- 了解[服务目录](#)的主要概念。
- 安装[Go 1.6+](#)以及设置 GOPATH。
- 安装生成 SSL 工件所需的[cfssl](#)工具。
- 服务目录需要 Kubernetes 1.7+ 版本。
- [安装和设置 kubectl](#)，以便将其配置为连接到 Kubernetes v1.7+ 集群。
- 要安装服务目录，kubectl 用户必须绑定到 cluster-admin 角色。为了确保这是正确的，请运行以下命令：

```
kubectl create clusterrolebinding cluster-admin-binding --clusterrole=cluster-admin --user=<user-name>
```

在本地环境中安装 sc

使用 go get 命令安装 sc CLI 工具：

```
go get github.com/GoogleCloudPlatform/k8s-service-catalog/installer/cmd/  
sc
```

执行上述命令后，sc 应被安装在 GOPATH/bin 目录中了。

在 Kubernetes 集群中安装服务目录

首先，检查是否已经安装了所有依赖项。运行：

```
sc check
```

如检查通过，应输出：

```
Dependency check passed. You are good to go.
```

接下来，运行安装命令并指定要用于备份的 storageclass：

```
sc install --etcd-backup-storageclass "standard"
```

卸载服务目录

如果您想使用 sc 工具从 Kubernetes 集群卸载服务目录，请运行：

```
sc uninstall
```

接下来

- 查看[服务代理示例](#)。
- 探索 [kubernetes-sigs/service-catalog](#) 项目。

反馈

此页是否对您有帮助？

是 否

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

最后修改 December 04, 2020 at 4:05 PM PST: [\[zh\] Fix links in zh localization \(3\) \(b9e8fb699\)](#)

网络

了解如何为你的集群配置网络。

[验证 IPv4/IPv6 双协议栈](#)

验证 IPv4/IPv6 双协议栈

本文分享了如何验证 IPv4/IPv6 双协议栈的 Kubernetes 集群。

准备开始

- 提供程序对双协议栈网络的支持（云供应商或其他方式必须能够为 Kubernetes 节点 提供可路由的 IPv4/IPv6 网络接口）
- 一个能够支持双协议栈的 [网络插件](#)，（如 kubenet 或 Calico）。
- [启用双协议栈](#) 集群

验证寻址

验证节点寻址

每个双协议栈节点应分配一个 IPv4 块和一个 IPv6 块。通过运行以下命令来验证是否配置了 IPv4/IPv6 Pod 地址范围。将示例节点名称替换为集群中的有效双协议栈节点。在此示例中，节点的名称为 k8s-linuxpool1-34450317-0：

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .spec.podCIDRs}}{{printf "%s\n" .}}{{end}}'
```

```
10.244.1.0/24  
a00:100::/24
```

应该分配一个 IPv4 块和一个 IPv6 块。

验证节点是否检测到 IPv4 和 IPv6 接口（用集群中的有效节点替换节点名称。在此示例中，节点名称为 k8s-linuxpool1-34450317-0）：

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .status.addresses}}{{printf "%s: %s \n" .type .address}}{{end}}'
```

```
Hostname: k8s-linuxpool1-34450317-0  
InternalIP: 10.240.0.5  
InternalIP: 2001:1234:5678:9abc::5
```

验证 Pod 寻址

验证 Pod 已分配了 IPv4 和 IPv6 地址。（用集群中的有效 Pod 替换 Pod 名称。在此示例中，Pod 名称为 pod01）

```
kubectl get pods pod01 -o go-template --template='{{range .status.podIPs}}{{printf "%s \n" .ip}}{{end}}'
```

```
10.244.1.4
```

```
a00:100::4
```

你也可以通过 `status.podIPs` 使用 Downward API 验证 Pod IP。以下代码段演示了如何通过容器内称为 `MY_POD_IPS` 的环境变量公开 Pod 的 IP 地址。

env:

```
- name: MY_POD_IPS
  valueFrom:
    fieldRef:
      fieldPath: status.podIPs
```

使用以下命令打印出容器内部 `MY_POD_IPS` 环境变量的值。该值是一个逗号分隔的列表，与 Pod 的 IPv4 和 IPv6 地址相对应。

```
kubectl exec -it pod01 -- grep MY_POD_IPS
```

```
MY_POD_IPS=10.244.1.4,a00:100::4
```

Pod 的 IP 地址也将被写入容器内的 `/etc/hosts` 文件中。在双栈 Pod 上执行 `cat /etc/hosts` 命令操作。从输出结果中，你可以验证 Pod 的 IPv4 和 IPv6 地址。

```
kubectl exec -it pod01 -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.244.1.4 pod01
a00:100::4 pod01
```

验证服务

创建以下未显式定义 `.spec.ipFamilyPolicy` 的 Service。Kubernetes 将从首个配置的 `service-cluster-ip-range` 给 Service 分配集群 IP，并将 `.spec.ipFamilyPolicy` 设置为 `SingleStack`。

[service/networking/dual-stack-default-svc.yaml](#)



```
apiVersion: v1
```

```
kind: Service
```

```
metadata:
```

```
  name: my-service
```

```
labels:  
  app: MyApp  
spec:  
  selector:  
    app: MyApp  
  ports:  
    - protocol: TCP  
      port: 80
```

使用 kubectl 查看 Service 的 YAML 定义。

```
kubectl get svc my-service -o yaml
```

该 Service 通过在 kube-controller-manager 的 --service-cluster-ip-range 标志设置的第一个配置范围，将 .spec.ipFamilyPolicy 设置为 SingleStack，将 .spec.clusterIP 设置为 IPv4 地址。

```
apiVersion: v1  
kind: Service  
metadata:  
  name: my-service  
  namespace: default  
spec:  
  clusterIP: 10.0.217.164  
  clusterIPs:  
    - 10.0.217.164  
  ipFamilies:  
    - IPv4  
  ipFamilyPolicy: SingleStack  
  ports:  
    - port: 80  
      protocol: TCP  
      targetPort: 9376  
  selector:  
    app: MyApp  
  sessionAffinity: None  
  type: ClusterIP  
status:  
  loadBalancer: {}
```

创建以下显示定义 .spec.ipFamilies 数组中的第一个元素为 IPv6 的 Service。Kubernetes 将 service-cluster-ip-range 配置的 IPv6 地址范围给 Service 分配集群 IP，并将 .spec.ipFamilyPolicy 设置为 SingleStack。

[service/networking/dual-stack-ipv6-svc.yaml](#)



```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ipFamily: IPv6
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

使用 kubectl 查看 Service 的 YAML 定义。

```
kubectl get svc my-service -o yaml
```

该 Service 通过在 kube-controller-manager 的 --service-cluster-ip-range 标志设置的 IPv6 地址范围，将 .spec.ipFamilyPolicy 设置为 SingleStack，将 .spec.clusterIP 设置为 IPv6 地址。

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: MyApp
  name: my-service
spec:
  clusterIP: fd00::5118
  clusterIPs:
    - fd00::5118
  ipFamilies:
    - IPv6
  ipFamilyPolicy: SingleStack
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: MyApp
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

创建以下显式定义 .spec.ipFamilyPolicy 为 PreferDualStack 的 Service。Kubernetes 将分配 IPv4 和 IPv6 地址（因为该集群启用了双栈），并根据 .spe

c.ipFamilies 数组中第一个元素的地址族，从 .spec.ClusterIPs 列表中选择 .spec.ClusterIP。

[service/networking/dual-stack-preferred-svc.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  ipFamilyPolicy: PreferDualStack
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
```

说明：

kubectl get svc 命令将仅在 CLUSTER-IP 字段中显示主 IP。

```
kubectl get svc -l app=MyApp
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
my-service	ClusterIP	fe80:20d::d06b	<none>	80/TCP 9s

使用 kubectl describe 验证服务是否从 IPv4 和 IPv6 地址块中获取了集群 IP。然后你就可以通过 IP 和端口，验证对服务的访问。

```
kubectl describe svc -l app=MyApp
```

```
Name:           my-service
Namespace:      default
Labels:          app=MyApp
Annotations:    <none>
Selector:        app=MyApp
Type:           ClusterIP
IP Family Policy: PreferDualStack
IP Families:    IPv4,IPv6
IP:             10.0.216.242
IPs:            10.0.216.242,fd00::af55
Port:           <unset> 80/TCP
TargetPort:     9376/TCP
```

```
Endpoints: <none>
Session Affinity: None
Events: <none>
```

创建双协议栈负载均衡服务

如果云提供商支持配置启用 IPv6 的外部负载均衡器，则将 ipFamily 字段设置为 IPv6 并将 type 字段设置为 LoadBalancer 的方式创建以下服务：

[service/networking/dual-stack-ipv6-lb-svc.yaml](#)


```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  ipFamily: IPv6
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

验证服务是否从 IPv6 地址块中接收到 CLUSTER-IP 地址以及 EXTERNAL-IP。然后，你可以通过 IP 和端口验证对服务的访问。

```
kubectl get svc -l app=MyApp
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE		
my-service	ClusterIP	fe80:20d::d06b	2001:db8:f100:4002::9d37:c0d7
	80:31868/TCP	30s	

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](#)。

用插件扩展 kubectl

通过创建和安装 kubectl 插件扩展 kubectl。

本指南演示了如何为 [kubectl](#) 安装和编写扩展。通过将核心 kubectl 命令看作与 Kubernetes 集群交互的基本构建块，集群管理员可以将插件视为一种利用这些构建块创建更复杂行为的方法。插件用新的子命令扩展了 kubectl，允许新的和自定义的特性不包括在 kubectl 的主要发行版中。

准备开始

你需要安装一个可用的 kubectl 可执行文件。

安装 kubectl 插件

插件只不过是一个独立的可执行文件，名称以 kubectl- 开头。要安装插件，只需将此可执行文件移动到 PATH 中的任何位置。

你也可以使用 [Krew](#) 来发现和安装开源的 kubectl 插件。Krew 是一个由 Kubernetes SIG CLI 社区维护的插件管理器。

注意： Krew [插件索引](#) 所维护的 kubectl 插件并未经过安全性审查。
你要了解安装和运行第三方插件的安全风险，因为它们本质上时是一些
在你的机器上 运行的程序。

发现插件

kubectl 提供一个命令 kubectl plugin list，用于搜索路径查找有效的插件可执行文件。执行此命令将遍历路径中的所有文件。任何以 kubectl- 开头的可执行文件都将在命令的输出中以它们在路径中出现的顺序显示。任何以 kubectl- 开头的文件如果不可执行，都将包含一个警告。对于任何相同的有效插件文件，都将包含一个警告。

你可以使用 [Krew](#) 从社区策划的[插件索引](#) 中发现和安装 kubectl 插件。

限制

目前无法创建覆盖现有 kubectl 命令的插件。例如，创建一个插件 kubectl-version 将导致该插件永远不会被执行，因为现有的 kubectl version 命令总是优先于它执行。由于这个限制，也不可能使用插件将新的子命令添加到现有的 kubectl 命令中。例如，通过将插件命名为 kubectl-create-foo 来添加子命令 kubectl create foo 将导致该插件被忽略。

对于任何试图这样做的有效插件 kubectl plugin list 的输出中将显示警告。

编写 kubectl 插件

你可以用任何编程语言或脚本编写插件，允许你编写命令行命令。

不需要安装插件或预加载，插件可执行程序从 kubectl 二进制文件接收继承的环境，插件根据其名称确定它希望实现的命令路径。例如，一个插件想要提供一个新的命令 kubectl foo，它将被简单地命名为 kubectl-foo，并且位于用户 PATH 的某个位置。

示例插件

```
#!/bin/bash

# 可选的参数处理
if [[ "$1" == "version" ]]
then
    echo "1.0.0"
    exit 0
fi

# 可选的参数处理
if [[ "$1" == "config" ]]
then
    echo $KUBECONFIG
    exit 0
fi

echo "I am a plugin named kubectl-foo"
```

使用插件

要使用上面的插件，只需使其可执行：

```
sudo chmod +x ./kubectl-foo
```

并将它放在你的 PATH 中的任何地方：

```
sudo mv ./kubectl-foo /usr/local/bin
```

你现在可以调用你的插件作为 kubectl 命令：

```
kubectl foo
```

```
I am a plugin named kubectl-foo
```

所有参数和标记按原样传递给可执行文件：

```
kubectl foo version
```

```
1.0.0
```

所有环境变量也按原样传递给可执行文件：

```
export KUBECONFIG=~/.kube/config
```

```
kubectl foo config
```

```
/home/<user>/.kube/config
```

```
KUBECONFIG=/etc/kube/config kubectl foo config
```

```
/etc/kube/config
```

此外，传递给插件的第一个参数总是调用它的位置的绝对路径（在上面的例子中，\$0 将等于 /usr/local/bin/kubectl-foo）。

命名插件

如上面的例子所示，插件根据文件名确定要实现的命令路径，插件所针对的命令路径中的每个子命令都由破折号（-）分隔。例如，当用户调用命令 kubectl foo bar baz 时，希望调用该命令的插件的文件名为 kubectl-foo-bar-baz。

参数和标记处理

说明：

插件机制不会为插件进程创建任何定制的、特定于插件的值或环境变量。

较老的插件机制会提供环境变量（例如 KUBECTL_PLUGINS_CURRENT_NAMESPACE）；这种机制已被废弃。

kubectl 插件必须解析并检查传递给它们的所有参数。参阅[使用命令行运行时包](#)了解针对插件开发人员的 Go 库的细节。

这里是一些用户调用你的插件的时候提供额外标志和参数的场景。这些场景时基于上述案例中的 kubectl-foo-bar-baz 插件的。

如果你运行 kubectl foo bar baz arg1 --flag=value arg2，kubectl 的插件机制将首先尝试找到最长可能名称的插件，在本例中是 kubectl-foo-bar-baz-arg1。当没有找到这个插件时，kubectl 就会将最后一个以破折号分隔的值视为参数（在本例中为 arg1），并尝试找到下一个最长的名称 kubectl-foo-bar-baz。在找到具有此名称的插件后，它将调用该插件，并在其名称之后将所有参数和标志传递给插件进程。

示例：

```
# 创建一个插件
echo -e '#!/bin/bash\n\n echo "My first command-line argument was $1"' >
kubectl-foo-bar-baz
sudo chmod +x ./kubectl-foo-bar-baz

# 将插件放到 PATH 下完成"安装"
sudo mv ./kubectl-foo-bar-baz /usr/local/bin

# 确保 kubectl 能够识别我们的插件
kubectl plugin list
```

The following kubectl-compatible plugins are available:

```
/usr/local/bin/kubectl-foo-bar-baz
```

```
# 测试通过 "kubectl" 命令来调用我们的插件时可行的
# 即使我们给插件传递一些额外的参数或标志
kubectl foo bar baz arg1 --meaningless-flag=true
```

```
My first command-line argument was arg1
```

正如你所看到的，我们的插件是基于用户指定的 kubectl 命令找到的，所有额外的参数和标记都是按原样传递给插件可执行文件的。

带有破折号和下划线的名称

虽然 kubectl 插件机制在插件文件名中使用破折号（-）分隔插件处理的子命令序列，但是仍然可以通过在文件名中使用下划线（_）来创建命令行中包含破折号的插件命令。

例子：

```
# 创建文件名中包含下划线的插件
echo -e '#!/bin/bash\n\n echo "I am a plugin with a dash in my name"' > ./
kubectl-foo_bar
sudo chmod +x ./kubectl-foo_bar

# 将插件放到 PATH 下
sudo mv ./kubectl-foo_bar /usr/local/bin

# 现在可以通过 kubectl 来调用插件
kubectl foo-bar
```

```
I am a plugin with a dash in my name
```

请注意，在插件文件名中引入下划线并不会阻止我们使用 kubectl foo_bar 之类的命令。可以使用破折号（-）或下划线（_）调用上面示例中的命令：

```
# 我们的插件也可以用破折号来调用  
kubectl foo-bar
```

I am a plugin with a dash in my name

```
# 你也可以使用下划线来调用我们的定制命令  
kubectl foo_bar
```

I am a plugin with a dash in my name

命名冲突和弊端

可以在 PATH 的不同位置提供多个文件名相同的插件，例如，给定一个 PATH 为：PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins，在 /usr/local/bin/plugins 和 /usr/local/bin/moreplugins 中可以存在一个插件 kubectl-foo 的副本，这样 kubectl plugin list 命令的输出就是：

```
PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins kubectl plugin list
```

The following kubectl-compatible plugins are available:

```
/usr/local/bin/plugins/kubectl-foo  
/usr/local/bin/moreplugins/kubectl-foo  
  - warning: /usr/local/bin/moreplugins/kubectl-foo is overshadowed by a  
similarly named plugin: /usr/local/bin/plugins/kubectl-foo  
  
error: one plugin warning was found
```

在上面的场景中 /usr/local/bin/moreplugins/kubectl-foo 下的警告告诉我们这个插件永远不会被执行。相反，首先出现在我们路径中的可执行文件 /usr/local/bin/plugins/kubectl-foo 总是首先被 kubectl 插件机制找到并执行。

解决这个问题的一种方法是你确保你希望与 kubectl 一起使用的插件的位置总是在你的路径中首先出现。例如，如果我们总是想使用 /usr/local/bin/moreplugins/kubectl foo，那么在调用 kubectl 命令 kubectl foo 时，我们只需将路径的值更改为 PATH=/usr/local/bin/moreplugins:/usr/local/bin/plugins。

调用最长的可执行文件名

对于插件文件名而言还有另一种弊端，给定用户路径中的两个插件 kubectl-foo-bar 和 kubectl-foo-bar-baz kubectl 插件机制总是为给定的用户命令选择尽可能长的插件名称。下面的一些例子进一步的说明了这一点：

```
# 对于给定的 kubectl 命令，最长可能文件名的插件是被优先选择的  
kubectl foo bar baz
```

Plugin kubectl-foo-bar-baz is executed

```
kubectl foo bar
```

```
Plugin kubectl-foo-bar is executed
```

```
kubectl foo bar baz buz
```

```
Plugin kubectl-foo-bar-baz is executed, with "buz" as its first argument
```

```
kubectl foo bar buz
```

```
Plugin kubectl-foo-bar is executed, with "buz" as its first argument
```

这种设计选择确保插件子命令可以跨多个文件实现，如果需要，这些子命令可以嵌套在“父”插件命令下：

```
ls ./plugin_command_tree
```

```
kubectl-parent
```

```
kubectl-parent-subcommand
```

```
kubectl-parent-subcommand-subsubcommand
```

检查插件警告

你可以使用前面提到的 `kubectl plugin list` 命令来确保你的插件可以被 `kubectl` 看到，并且验证没有警告防止它被称为 `kubectl` 命令。

```
kubectl plugin list
```

```
The following kubectl-compatible plugins are available:
```

```
test/fixtures/pkg/kubectl/plugins/kubectl-foo
```

```
/usr/local/bin/kubectl-foo
```

```
  - warning: /usr/local/bin/kubectl-foo is overshadowed by a similarly
    named plugin: test/fixtures/pkg/kubectl/plugins/kubectl-foo
    plugins/kubectl-invalid
```

```
  - warning: plugins/kubectl-invalid identified as a kubectl plugin, but it is
    not executable
```

```
error: 2 plugin warnings were found
```

使用命令行运行时包

如果你在编写 `kubectl` 插件，而且你选择使用 Go 语言，你可以利用 [cli-runtime](#) 工具库。

这些库提供了一些辅助函数，用来解析和更新用户的 [kubeconfig](#) 文件，向 API 服务器发起 REST 风格的请求，或者将参数绑定到某配置上，抑或将其打印输出。

关于 CLI Runtime 仓库所提供的工具的使用实例，可参考 [CLI 插件示例](#) 项目。

分发 kubectl 插件

如果你开发了一个插件给别人使用，你应该考虑如何为其封装打包、如何分发软件以及将来的更新到用户。

Krew

[Krew](#) 提供了一种对插件进行打包和分发的跨平台方式。基于这种方式，你会在所有的目标平台（Linux、Windows、macOS 等）使用同一种打包形式，包括为用户提供更新。Krew 也维护一个[插件索引（plugin index）](#)以便其他人能够发现你的插件并安装之。

原生的与特定平台的包管理

另一种方式是，你可以使用传统的包管理器（例如 Linux 上的 apt 或 yum，Windows 上的 Chocolatey、macOs 上的 Homebrew）。只要能够将新的可执行文件放到用户的 PATH 路径上某处，这种包管理器就符合需要。作为一个插件作者，如果你选择这种方式来分发，你就需要自己来管理和更新你的 kubectl 插件的分发包，包括所有平台和所有发行版本。

源代码

你也可以发布你的源代码，例如，发布为某个 Git 仓库。如果你选择这条路线，希望使用该插件的用户必须取回代码、配置一个构造环境（如果需要编译的话）并部署该插件。如果你也提供编译后的软件包，或者使用 Krew，那就会大大简化安装过程了。

接下来

- 查看 CLI 插件库示例，查看用 Go 编写的插件的[详细示例](#)
- 如有任何问题，请随时联系 [SIG CLI](#)
- 了解 [Krew](#)，一个 kubectl 插件管理器。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 18, 2021 at 5:01 PM PST: [\[zh\] Fix nit in kubectl-plugins.md \(5a7e944bf\)](#)

管理巨页 (HugePages)

将大页配置和管理为集群中的可调度资源。

FEATURE STATE: Kubernetes v1.20 [stable]

Kubernetes 支持在 Pod 应用中使用预先分配的巨页。本文描述了用户如何使用巨页，以及当前的限制。

准备开始

- 为了使节点能够上报巨页容量，Kubernetes 节点必须预先分配巨页。每个节点只能预先分配一种特定规格的巨页。

节点会自动发现全部巨页资源，并作为可供调度的资源进行上报。

API

用户可以通过在容器级别的资源需求中使用资源名称 `hugepages-<size>` 来使用巨页，其中的 size 是特定节点上支持的以整数值表示的最小二进制单位。例如，如果一个节点支持 2048KiB 和 1048576KiB 页面大小，它将公开可调度的资源 `hugepages-2Mi` 和 `hugepages-1Gi`。与 CPU 或内存不同，巨页不支持过量使用 (`overcommit`)。注意，在请求巨页资源时，还必须请求内存或 CPU 资源。

同一 Pod 的 spec 中可能会消耗不同尺寸的巨页。在这种情况下，它必须对所有挂载卷使用 `medium: HugePages-<hugepagesize>` 标识。

```
apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
spec:
  containers:
    - name: example
      image: fedora:latest
      command:
        - sleep
        - inf
      volumeMounts:
        - mountPath: /hugepages-2Mi
          name: hugepage-2mi
        - mountPath: /hugepages-1Gi
          name: hugepage-1gi
      resources:
        limits:
          hugepages-2Mi: 100Mi
```

```

hugepages-1Gi: 2Gi
memory: 100Mi
requests:
  memory: 100Mi
volumes:
- name: hugepage-2mi
  emptyDir:
    medium: HugePages-2Mi
- name: hugepage-1gi
  emptyDir:
    medium: HugePages-1Gi

```

Pod 只有在请求同一大小的巨页时才使用 medium : HugePages。

```

apiVersion: v1
kind: Pod
metadata:
  name: huge-pages-example
spec:
  containers:
- name: example
  image: fedora:latest
  command:
    - sleep
    - inf
  volumeMounts:
- mountPath: /hugepages
  name: hugepage
resources:
  limits:
    hugepages-2Mi: 100Mi
    memory: 100Mi
  requests:
    memory: 100Mi
volumes:
- name: hugepage
  emptyDir:
    medium: HugePages

```

- 巨页的资源请求值必须等于其限制值。该条件在指定了资源限制，而没有指定请求的情况下默认成立。
- 巨页是被隔离在 pod 作用域的，因此每个容器在 spec 中都对 cgroup 沙盒有自己的限制。
- 巨页可用于 EmptyDir 卷，不过 EmptyDir 卷所使用的巨页数量不能够超出 Pod 请求的巨页数量。
- 通过带有 SHM_HUGETLB 的 shmget() 使用巨页的应用，必须运行在一个与 proc/sys/vm/hugetlb_shm_group 匹配的补充组下。

- 通过 ResourceQuota 资源，可以使用 hugepages-<size> 标记控制每个命名空间下的巨页使用量，类似于使用 cpu 或 memory 来控制其他计算资源。
- 多种尺寸的巨页的支持需要特性门控配置。它可以通过 HugePageStorageMediumSize 特性门控在 [kubelet](#) 和 [kube-apiserver](#) 中开启（--feature-gates=HugePageStorageMediumSize=false）。

反馈

此页是否对您有帮助？

是 否

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

最后修改 January 04, 2021 at 1:14 AM PST: [\[zh\] fix feature gate for hugepage \(5f5a8a172\)](#)

调度 GPUs

配置和调度 GPU 成一类资源以供集群中节点使用。

FEATURE STATE: Kubernetes v1.10 [beta]

Kubernetes 支持对节点上的 AMD 和 NVIDIA GPU (图形处理单元) 进行管理，目前处于**实验状态**。

本页介绍用户如何在不同的 Kubernetes 版本中使用 GPU，以及当前存在的一些限制。

使用设备插件

Kubernetes 实现了[设备插件 \(Device Plugins \)](#) 以允许 Pod 访问类似 GPU 这类特殊的硬件功能特性。

作为集群管理员，你要在节点上安装来自对应硬件厂商的 GPU 驱动程序，并运行来自 GPU 厂商的对应的设备插件。

- [AMD](#)
- [NVIDIA](#)

当以上条件满足时，Kubernetes 将暴露 `amd.com/gpu` 或 `nvidia.com/gpu` 为可调度的资源。

你可以通过请求 `<vendor>.com/gpu` 资源来使用 GPU 设备，就像你为 CPU 和内存所做的那样。不过，使用 GPU 时，在如何指定资源需求这个方面还是有一些限制的：

- GPUs 只能设置在 `limits` 部分，这意味着：
 - 你可以指定 GPU 的 `limits` 而不指定其 `requests`，Kubernetes 将使用限制值作为默认的请求值；
 - 你可以同时指定 `limits` 和 `requests`，不过这两个值必须相等。
 - 你不可以仅指定 `requests` 而不指定 `limits`。
- 容器（以及 Pod）之间是不共享 GPU 的。GPU 也不可以过量分配（Overcommitting）。
- 每个容器可以请求一个或者多个 GPU，但是用小数值来请求部分 GPU 是不允许的。

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/
      nvidia-cuda/Dockerfile
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1 # requesting 1 GPU
```

部署 AMD GPU 设备插件

[官方的 AMD GPU 设备插件](#) 有以下要求：

- Kubernetes 节点必须预先安装 AMD GPU 的 Linux 驱动。

如果你的集群已经启动并且满足上述要求的话，可以这样部署 AMD 设备插件：

```
kubectl create -f https://raw.githubusercontent.com/RadeonOpenCompute/
k8s-device-plugin/r1.10/k8s-ds-amdgpu-dp.yaml
```

你可以到 [RadeonOpenCompute/k8s-device-plugin](#) 项目报告有关此设备插件的问题。

部署 NVIDIA GPU 设备插件

对于 NVIDIA GPUs，目前存在两种设备插件的实现：

官方的 NVIDIA GPU 设备插件

官方的 NVIDIA GPU 设备插件 有以下要求:

- Kubernetes 的节点必须预先安装了 NVIDIA 驱动
- Kubernetes 的节点必须预先安装 [nvidia-docker 2.0](#)
- Docker 的默认运行时必须设置为 nvidia-container-runtime , 而不是 runc
- NVIDIA 驱动版本 ~ = 384.81

如果你的集群已经启动并且满足上述要求的话 , 可以这样部署 NVIDIA 设备插件 :

```
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/1.0.0-beta4/nvidia-device-plugin.yaml
```

请到 [NVIDIA/k8s-device-plugin](#) 项目报告有关此设备插件的问题。

GCE 中使用的 NVIDIA GPU 设备插件

[GCE 使用的 NVIDIA GPU 设备插件](#) 并不要求使用 nvidia-docker , 并且对于任何实现了 Kubernetes CRI 的容器运行时 , 都应该能够使用。这一实现已经在 [Container-Optimized OS](#) 上进行了测试 , 并且在 1.9 版本之后会有对于 Ubuntu 的实验性代码。

你可以使用下面的命令来安装 NVIDIA 驱动以及设备插件 :

```
# 在 COntainer-Optimized OS 上安装 NVIDIA 驱动:
```

```
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPlatform/container-engine-accelerators/stable/daemonset.yaml
```

```
# 在 Ubuntu 上安装 NVIDIA 驱动 (实验性质):
```

```
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPlatform/container-engine-accelerators/stable/nvidia-driver-installer/ubuntu/daemonset.yaml
```

```
# 安装设备插件:
```

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.12/cluster/addons/device-plugins/nvidia-gpu/daemonset.yaml
```

请到 [GoogleCloudPlatform/container-engine-accelerators](#) 报告有关此设备插件以及安装方法的问题。

关于如何在 GKE 上使用 NVIDIA GPUs , Google 也提供自己的[指令](#)。

集群内存在不同类型的 GPU

如果集群内部的不同节点上有不同类型的 NVIDIA GPU , 那么你可以使用 [节点标签和节点选择器](#) 来将 pod 调度到合适的节点上。

例如：

```
# 为你的节点加上它们所拥有的加速器类型的标签
kubectl label nodes <node-with-k80> accelerator=nvidia-tesla-k80
kubectl label nodes <node-with-p100> accelerator=nvidia-tesla-p100
```

自动节点标签

如果你在使用 AMD GPUs，你可以部署 [Node Labeler](#)，它是一个 [控制器](#)，会自动给节点打上 GPU 属性标签。目前支持的属性：

- 设备 ID (-device-id)
- VRAM 大小 (-vram)
- SIMD 数量(-simd-count)
- 计算单位数量(-cu-count)
- 固件和特性版本 (-firmware)
- GPU 系列，两个字母的首字母缩写(-family)
 - SI - Southern Islands
 - CI - Sea Islands
 - KV - Kaveri
 - VI - Volcanic Islands
 - CZ - Carrizo
 - AI - Arctic Islands
 - RV - Raven

示例：

```
kubectl describe node cluster-node-23
```

```
Name:           cluster-node-23
Roles:          <none>
Labels:         beta.amd.com/gpu.cu-count.64=1
                beta.amd.com/gpu.device-id.6860=1
                beta.amd.com/gpu.family.AI=1
                beta.amd.com/gpusimd-count.256=1
                beta.amd.com/gpu.vram.16G=1
                beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/hostname=cluster-node-23
Annotations:    kubeadm.alpha.kubernetes.io/cri-socket: /var/run/
dockershim.sock
                node.alpha.kubernetes.io/ttl: 0
.....
```

使用了 Node Labeler 的时候，你可以在 Pod 的规约中指定 GPU 的类型：

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

```
name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/
      # nvidia-cuda/Dockerfile
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100 # or nvidia-tesla-k80 etc.
```

这能够保证 Pod 能够被调度到你所指定类型的 GPU 的节点上去。

反馈

此页是否对您有帮助？

是 否

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

最后修改 September 21, 2020 at 1:12 PM PST: [\[zh\] translate /zh/docs/tasks/configmap-secret/managing-secret-using-kustomize.md \(#24016\) \(314820acd\)](#)