生产环境

生产质量的 Kubernetes 集群需要规划和准备。 如果你的 Kubernetes 集群是用来运行关键负载的，该集群必须被配置为弹性的（Resilient）。 本页面阐述你在安装生产就绪的集群或将现有集群升级为生产用途时可以遵循的步骤。 如果你已经熟悉生产环境安装，因此只关注一些链接，则可以跳到接下来节。

生产环境考量

通常，一个生产用 Kubernetes 集群环境与个人学习、开发或测试环境所使用的 Kubernetes 相比有更多的需求。生产环境可能需要被很多用户安全地访问，需要 提供一致的可用性，以及能够与需求变化相适配的资源。

在你决定在何处运行你的生产用 Kubernetes 环境（在本地或者在云端），以及 你希望承担或交由他人承担的管理工作量时，需要考察以下因素如何影响你对 Kubernetes 集群的需求：

- *可用性*：一个单机的 Kubernetes 学习环境 具有单点失效特点。创建高可用的集群则意味着需要考虑：
  - 将控制面与工作节点分开
  - 在多个节点上提供控制面组件的副本
  - 为针对集群的 API 服务器 的流量提供负载均衡
  - 随着负载的合理需要，提供足够的可用的（或者能够迅速变为可用的）工作节点

- *规模*：如果你预期你的生产用 Kubernetes 环境要承受固定量的请求， 你可能可以针对所需要的容量来一次性完成安装。 不过，如果你预期服务请求会随着时间增长，或者因为类似季节或者特殊事件的 原因而发生剧烈变化，你就需要规划如何处理请求上升时对控制面和工作节点 的压力，或者如何缩减集群规模以减少未使用资源的消耗。

- *安全性与访问管理*：在你自己的学习环境 Kubernetes 集群上，你拥有完全的管理员特权。 但是针对运行着重要工作负载的共享集群，用户账户不止一两个时，就需要更细粒度 的方案来确定谁或者哪些主体可以访问集群资源。 你可以使用基于角色的访问控制（RBAC） 和其他安全机制来确保用户和负载能够访问到所需要的资源，同时确保工作负载及集群 自身仍然是安全的。 你可以通过管理策略和 容器资源来 针对用户和工作负载所可访问的资源设置约束，

在自行构造 Kubernetes 生产环境之前，请考虑将这一任务的部分或者全部交给 云方案承包服务 提供商或者其他 Kubernetes 合作伙伴。 选项有：

- *无服务*：仅是在第三方设备上运行负载，完全不必管理集群本身。你需要为 CPU 用量、内存和磁盘请求等付费。

- *托管控制面*：让供应商决定集群控制面的规模和可用性，并负责打补丁和升级等操作。
- *托管工作节点*：配置一个节点池来满足你的需要，由供应商来确保节点始终可用， 并在需要的时候完成升级。
- *集成*：有一些供应商能够将 Kubernetes 与一些你可能需要的其他服务集成， 这类服务包括存储、容器镜像仓库、身份认证方法以及开发工具等。

无论你是自行构造一个生产用 Kubernetes 集群还是与合作伙伴一起协作，请审阅 下面章节以评估你的需求，因为这关系到你的集群的 *控制面*、*工作节点*、 *用户访问* 以及 *负载资源*。

生产用集群安装

在生产质量的 Kubernetes 集群中，控制面用不同的方式来管理集群和可以 分布到多个计算机上的服务。每个工作节点则代表的是一个可配置来运行 Kubernetes Pods 的实体。

生产用控制面

最简单的 Kubernetes 集群中，整个控制面和工作节点服务都运行在同一台机器上。 你可以通过添加工作节点来提升环境能力，正如 Kubernetes 组件示意图所示。 如果只需要集群在很短的一段时间内可用，或者可以在某些事物出现严重问题时直接丢弃， 这种配置可能符合你的需要。

如果你需要一个更为持久的、高可用的集群，那么你就需要考虑扩展控制面的方式。 根据设计，运行在一台机器上的单机控制面服务不是高可用的。 如果保持集群处于运行状态并且需要确保在出现问题时能够被修复这点很重要， 可以考虑以下步骤：

- *选择部署工具*：你可以使用类似 kubeadm、kops 和 kubespray 这类工具来部署控制面。 参阅使用部署工具安装 Kubernetes 以了解使用这类部署方法来完成生产就绪部署的技巧。 存在不同的容器运行时 可供你的部署采用。

- *管理证书*：控制面服务之间的安全通信是通过证书来完成的。证书是在部署期间 自动生成的，或者你也可以使用你自己的证书机构来生成它们。 参阅 PKI 证书和需求了解细节。

- *为 API 服务器配置负载均衡*：配置负载均衡器来将外部的 API 请求散布给运行在 不同节点上的 API 服务实例。参阅 创建外部负载均衡器 了解细节。

- *分离并备份 etcd 服务*：etcd 服务可以运行于其他控制面服务所在的机器上， 也可以运行在不同的机器上以获得更好的安全性和可用性。 因为 etcd 存储着集群的配置数据，应该经常性地对 etcd 数据库进行备份， 以确保在需要的时候你可以修复该数据库。与配置和使用 etcd 相关的细节可参阅 etcd FAQ。 更多的细节可参阅为 Kubernetes 运维 etcd 集群 和使用 kubeadm 配置高可用的 etcd 集群。

- *创建多控制面系统*：为了实现高可用性，控制面不应被限制在一台机器上。 如果控制面服务是使用某 init 服务（例如 systemd）来运行的，每个服务应该 至少运行在三台机器上。不过，将控制面作为服务运行在 Kubernetes Pods 中可以确保你所请求的个数的服务始终保持可用。 调度器应该是可容错的，但不是高可用的。 某些部署工具会安装 Raft 票选算法来对 Kubernetes 服务执行领导者选举。如果主节点消失，另一个服务会被选中并接手相应服务。

- *跨多个可用区*：如果保持你的集群一直可用这点非常重要，可以考虑创建一个跨 多个数据中心的集群；在云环境中，这些数据中心被视为可用区。 若干个可用区在一起可构成地理区域。 通过将集群分散到同一区域中的多个可用区内，即使某个可用区不可用，整个集群 能够继续工作的机会也大大增加。 更多的细节可参阅跨多个可用区运行。

- *管理演进中的特性*：如果你计划长时间保留你的集群，就需要执行一些维护其 健康和安全的任务。例如，如果你采用 kubeadm 安装的集群，则有一些可以帮助你完成 证书管理 和升级 kubeadm 集群 的指令。 参见管理集群了解一个 Kubernetes 管理任务的较长列表。

要了解运行控制面服务时可使用的选项，可参阅 kube-apiserver、 kube-controller-manager 和 kube-scheduler 组件参考页面。 如要了解高可用控制面的例子，可参阅 高可用拓扑结构选项、 使用 kubeadm 创建高可用集群 以及为 Kubernetes 运维 etcd 集群。 关于制定 etcd 备份计划，可参阅 对 etcd 集群执行备份。

生产用工作节点

生产质量的工作负载需要是弹性的；它们所依赖的其他组件（例如 CoreDNS）也需要是弹性的。无论你是自行管理控制面还是让云供应商来管理，你都需要考虑如何管理工作节点 （有时也简称为 *节点*）。

- *配置节点*：节点可以是物理机或者虚拟机。如果你希望自行创建和管理节点， 你可以安装一个受支持的操作系统，之后添加并运行合适的 节点服务。 考虑：

  - 在安装节点时要通过配置适当的内存、CPU 和磁盘速度、存储容量来满足 你的负载的需求。
  - 是否通用的计算机系统即足够，还是你有负载需要使用 GPU 处理器、Windows 节点 或者 VM 隔离。

- *验证节点*：参阅验证节点配置 以了解如何确保节点满足加入到 Kubernetes 集群的需求。

- *添加节点到集群中*：如果你自行管理你的集群，你可以通过安装配置你的机器， 之后或者手动加入集群，或者让它们自动注册到集群的 API 服务器。参阅 节点节，了解如何配置 Kubernetes 以便以这些方式来添加节点。

- *向集群中添加 Windows 节点*：Kubernetes 提供对 Windows 工作节点的支持； 这使得你可以运行实现于 Windows 容器内的工作负载。参阅 Kubernetes 中的 Windows 了解进一步的详细信息。

- *扩缩节点*：制定一个扩充集群容量的规划，你的集群最终会需要这一能力。 参阅大规模集群考察事项 以确定你所需要的节点数；这一规模是基于你要运行的 Pod 和容器个数来确定的。如果你自行管理集群节点，这可能意味着要购买和安装你自己的物理设备。

- *节点自动扩缩容*：大多数云供应商支持 集群自动扩缩器（Cluster Autoscaler） 以便替换不健康的节点、根据需求来增加或缩减节点个数。参阅 常见问题 了解自动扩缩器的工作方式，并参阅 Deployment 了解不同云供应商是如何实现集群自动扩缩器的。 对于本地集群，有一些虚拟化平台可以通过脚本来控制按需启动新节点。

- *安装节点健康检查*：对于重要的工作负载，你会希望确保节点以及在节点上 运行的 Pod 处于健康状态。通过使用 Node Problem Detector， 你可以确保你的节点是健康的。

生产级用户环境

在生产环境中，情况可能不再是你或者一小组人在访问集群，而是几十 上百人需要访问集群。在学习环境或者平台原型环境中，你可能具有一个 可以执行任何操作的管理账号。在生产环境中，你可需要对不同名字空间 具有不同访问权限级别的很多账号。

建立一个生产级别的集群意味着你需要决定如何有选择地允许其他用户访问集群。 具体而言，你需要选择验证尝试访问集群的人的身份标识（身份认证），并确定 他们是否被许可执行他们所请求的操作（鉴权）：

- *认证（Authentication）*： API 服务器可以使用客户端证书、持有者令牌、身份 认证代理或者 HTTP 基本认证机制来完成身份认证操作。 你可以选择你要使用的认证方法。通过使用插件，API 服务器可以充分利用你所在 组织的现有身份认证方法，例如 LDAP 或者 Kerberos。 关于认证 Kubernetes 用户身份的不同方法的描述，可参阅 身份认证。

- *鉴权（Authorization）*： 当你准备为一般用户执行权限判定时，你可能会需要 在 RBAC 和 ABAC 鉴权机制之间做出选择。参阅 鉴权概述，了解 对用户账户（以及访问你的集群的服务账户）执行鉴权的不同模式。
  - *基于角色的访问控制*（RBAC）： 让你通过为通过身份认证的用户授权特定的许可集合来控制集群访问。 访问许可可以针对某特定名字空间（Role）或者针对整个集群（ClusterRole）。 通过使用 RoleBinding 和 ClusterRoleBinding 对象，这些访问许可可以被 关联到特定的用户身上。
  - *基于属性的访问控制*（ABAC）： 让你能够基于集群中资源的属性来创建访问控制策略，基于对应的属性来决定 允许还是拒绝访问。策略文件的每一行都给出版本属性（apiVersion 和 kind） 以及一个规约属性的映射，用来匹配主体（用户或组）、资源属性、非资源属性 （/version 或 /apis）和只读属性。 参阅示例以了解细节。

作为在你的生产用 Kubernetes 集群中安装身份认证和鉴权机制的负责人， 要考虑的事情如下：

- *设置鉴权模式*：当 Kubernetes API 服务器 （kube-apiserver） 启动时，所支持的鉴权模式必须使用 `--authorization-mode` 标志配置。 例如， `kube-apiserver.yaml` （位于 `/etc/kubernetes/manifests` 下）中对应的 标志可以设置为 `Node,RBAC` 。这样就会针对已完成身份认证的请求执行 Node 和 RBAC 鉴权。

- *创建用户证书和角色绑定（RBAC）*： 如果你在使用 RBAC 鉴权，用户可以创建 由集群 CA 签名的 CertificateSigningRequest（CSR）。接下来你就可以将 Role 和 ClusterRole 绑定到每个用户身上。 参阅证书签名请求 了解细节。

- *创建组合属性的策略（ABAC）*： 如果你在使用 ABAC 鉴权，你可以设置属性组合 以构造策略对所选用户或用户组执行鉴权，判定他们是否可访问特定的资源 （例如 Pod）、名字空间或者 apiGroup。进一步的详细信息可参阅 示例。

- *考虑准入控制器*：针对指向 API 服务器的请求的其他鉴权形式还包括 Webhook 令牌认证。Webhook 和其他特殊的鉴权类型需要通过向 API 服务器添加 准入控制器 来启用。

为负载资源设置约束

生产环境负载的需求可能对 Kubernetes 的控制面内外造成压力。 在针对你的集群的负载执行配置时，要考虑以下条目：

- *设置名字空间限制*：为每个名字空间的内存和 CPU 设置配额。 参阅管理内存、CPU 和 API 资源 以了解细节。你也可以设置 层次化名字空间 来继承这类约束。
- *为 DNS 请求做准备*：如果你希望工作负载能够完成大规模扩展，你的 DNS 服务 也必须能够扩大规模。参阅 自动扩缩集群中 DNS 服务。
- *创建额外的服务账户*：用户账户决定用户可以在集群上执行的操作，服务账号则定义的 是在特定名字空间中 Pod 的访问权限。 默认情况下，Pod 使用所在名字空间中的 default 服务账号。 参阅管理服务账号 以了解如何创建新的服务账号。例如，你可能需要：
  - 为 Pod 添加 Secret，以便 Pod 能够从某特定的容器镜像仓库拉取镜像。 参阅为 Pod 配置服务账号 以获得示例。
  - 为服务账号设置 RBAC 访问许可。参阅 服务账号访问许可 了解细节。

接下来

- 决定你是想自行构造自己的生产用 Kubernetes 还是从某可用的 云服务外包厂商 或 Kubernetes 合作伙伴获得集群。
- 如果你决定自行构造集群，则需要规划如何处理 证书 并为类似 etcd 和 API 服务器 这些功能组件配置高可用能力。
- 选择使用 kubeadm、 kops 或 Kubespray 作为部署方法。
- 通过决定身份认证和 鉴权方法来配置用户管理。
- 通过配置资源限制、 DNS 自动扩缩 和服务账号 来为应用负载作准备。

1 - 容器运行时

你需要在集群内每个节点上安装一个 容器运行时 以使 Pod 可以运行在上面。本文概述了所涉及的内容并描述了与节点设置相关的任务。

本文列出了在 Linux 上结合 Kubernetes 使用的几种通用容器运行时的详细信息：

- containerd
- CRI-O
- Docker

提示：对于其他操作系统，请查阅特定于你所使用平台的相关文档。

Cgroup 驱动程序

控制组用来约束分配给进程的资源。

当某个 Linux 系统发行版使用 systemd 作为其初始化系统时，初始化进程会生成并使用一个 root 控制组（`cgroup`），并充当 cgroup 管理器。 Systemd 与 cgroup 集成紧密，并将为每个 systemd 单元分配一个 cgroup。 你也可以配置容器运行时和 kubelet 使用 `cgroupfs` 。 连同 systemd 一起使用 `cgroupfs` 意味着将有两个不同的 cgroup 管理器。

单个 cgroup 管理器将简化分配资源的视图，并且默认情况下将对可用资源和使用 中的资源具有更一致的视图。 当有两个管理器共存于一个系统中时，最终将对这些资源产生两种视图。 在此领域人们已经报告过一些案例，某些节点配置让 kubelet 和 docker 使用 `cgroupfs` ，而节点上运行的其余进程则使用 systemd; 这类节点在资源压力下 会变得不稳定。

更改设置，令容器运行时和 kubelet 使用 `systemd` 作为 cgroup 驱动，以此使系统更为稳定。 对于 Docker, 设置 `native.cgroupdriver=systemd` 选项。

注意：更改已加入集群的节点的 cgroup 驱动是一项敏感的操作。 如果 kubelet 已经使用某 cgroup 驱动的语义创建了 pod，更改运行时以使用 别的 cgroup 驱动，当为现有 Pods 重新创建 PodSandbox 时会产生错误。 重启 kubelet 也可能无法解决此类问题。 如果你有切实可行的自动化方案，使用其他已更新配置的节点来替换该节点， 或者使用自动化方案来重新安装。

Cgroup v2

Cgroup v2 是 cgroup Linux API 的下一个版本。与 cgroup v1 不同的是， Cgroup v2 只有一个层次结构，而不是每个控制器有一个不同的层次结构。

新版本对 cgroup v1 进行了多项改进，其中一些改进是：

- 更简洁、更易于使用的 API
- 可将安全子树委派给容器
- 更新的功能，如压力失速信息（Pressure Stall Information）

尽管内核支持混合配置，即其中一些控制器由 cgroup v1 管理，另一些由 cgroup v2 管理，Kubernetes 仅支持使用同一 cgroup 版本来管理所有控制器。

如果 systemd 默认不使用 cgroup v2，你可以通过在内核命令行中添加 `systemd.unified_cgroup_hierarchy=1` 来配置系统去使用它。

```
# dnf install -y grubby && \
  sudo grubby \
  --update-kernel=ALL \
  --args="systemd.unified_cgroup_hierarchy=1"
```

要应用配置，必须重新启动节点。

切换到 cgroup v2 时，用户体验不应有任何明显差异， 除非用户直接在节点上或在容器内访问 cgroup 文件系统。 为了使用它，CRI 运行时也必须支持 cgroup v2。

将 kubeadm 托管的集群迁移到 `systemd` 驱动

如果你想迁移到现有 kubeadm 托管集群中的 `systemd` cgroup 驱动程序， 遵循此迁移指南。

容器运行时

> **说明：** 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。

containerd

本节包含使用 containerd 作为 CRI 运行时的必要步骤。

使用以下命令在系统上安装 Containerd：

安装和配置的先决条件：

```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

# 设置必需的 sysctl 参数，这些参数在重新启动后仍然存在。
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

# 应用 sysctl 参数而无需重新启动
sudo sysctl --system
```

安装 containerd:

Linux | Windows (PowerShell)

1. 从官方Docker仓库安装 `containerd.io` 软件包。可以在 安装 Docker 引擎 中找到有关为各自的 Linux 发行版设置 Docker 存储库和安装 `containerd.io` 软件包的说明。
2. 配置 containerd：

```
sudo mkdir -p /etc/containerd
containerd config default | sudo tee /etc/containerd/config.toml
```

3. 重新启动 containerd:

```
sudo systemctl restart containerd
```

使用 `systemd` cgroup 驱动程序

结合 `runc` 使用 `systemd` cgroup 驱动，在 `/etc/containerd/config.toml` 中设置

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  ...
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    SystemdCgroup = true
```

如果您应用此更改，请确保再次重新启动 containerd：

```
sudo systemctl restart containerd
```

当使用 kubeadm 时，请手动配置 kubelet 的 cgroup 驱动.

CRI-O

本节包含安装 CRI-O 作为容器运行时的必要步骤。

使用以下命令在系统中安装 CRI-O：

> **说明：**
> CRI-O 的主要以及次要版本必须与 Kubernetes 的主要和次要版本相匹配。 更多信息请查阅 CRI-O 兼容性列表。

安装并配置前置环境：

```
# 创建 .conf 文件以在启动时加载模块
cat <<EOF | sudo tee /etc/modules-load.d/crio.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

# 配置 sysctl 参数，这些配置在重启之后仍然起作用
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF

sudo sysctl --system
```

Debian | Ubuntu | CentOS | openSUSE Tumbleweed | Fedora

在下列操作系统上安装 CRI-O, 使用下表中合适的值设置环境变量 `OS`：

操作系统	$OS
Debian Unstable	`Debian_Unstable`
Debian Testing	`Debian_Testing`

然后，将 `$VERSION` 设置为与你的 Kubernetes 相匹配的 CRI-O 版本。 例如，如果你要安装 CRI-O 1.20, 请设置 `VERSION=1.20` . 你也可以安装一个特定的发行版本。 例如要安装 1.20.0 版本，设置 `VERSION=1.20.0:1.20.0` .

然后执行

```
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.li
deb https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stab
```

```
EOF
cat <<EOF | sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cr
deb http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stabl
EOF

curl -L https://download.opensuse.org/repositories/devel:kubic:libcontainers:sta
curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/

sudo apt-get update
sudo apt-get install cri-o cri-o-runc
```

启动 CRI-O：

```
sudo systemctl daemon-reload
sudo systemctl enable crio --now
```

参阅CRI-O 安装指南 了解进一步的详细信息。

cgroup 驱动

默认情况下，CRI-O 使用 systemd cgroup 驱动程序。要切换到 `cgroupfs` 驱动程序，或者编辑 `/etc / crio / crio.conf` 或放置一个插件 在 `/etc/crio/crio.conf.d/02-cgroup-manager.conf` 中的配置，例如：

```
[crio.runtime]
conmon_cgroup = "pod"
cgroup_manager = "cgroupfs"
```

另请注意更改后的 `conmon_cgroup` ，将 CRI-O 与 `cgroupfs` 一起使用时， 必须将其设置为 `pod` 。通常有必要保持 kubelet 的 cgroup 驱动程序配置 （通常透过 kubeadm 完成）和 CRI-O 一致。

Docker

1. 在每个节点上，根据安装 Docker 引擎 为你的 Linux 发行版安装 Docker。 你可以在此文件中找到最新的经过验证的 Docker 版本 依赖关系。

2. 配置 Docker 守护程序，尤其是使用 systemd 来管理容器的 cgroup。

```
sudo mkdir /etc/docker
cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

> **说明：**
> 对于运行 Linux 内核版本 4.0 或更高版本，或使用 3.10.0-51 及更高版本的 RHEL 或 CentOS 的系统， `overlay2` 是首选的存储驱动程序。

3. 重新启动 Docker 并在启动时启用：

```
sudo systemctl enable docker
sudo systemctl daemon-reload
sudo systemctl restart docker
```

说明：
有关更多信息，请参阅

- 配置 Docker 守护程序
- 使用 systemd 控制 Docker

2 - Turnkey 云解决方案

本页列示 Kubernetes 认证解决方案供应商。 在每一个供应商分页，你可以学习如何安装和设置生产就绪的集群。

3 - 使用部署工具安装 Kubernetes

3.1 - 使用 kubeadm 引导集群

3.1.1 - 安装 kubeadm

本页面显示如何安装 `kubeadm` 工具箱。 有关在执行此安装过程后如何使用 kubeadm 创建集群的信息，请参见 使用 kubeadm 创建集群 页面。

准备开始

- 一台兼容的 Linux 主机。Kubernetes 项目为基于 Debian 和 Red Hat 的 Linux 发行版以及一些不提供包管理器的发行版提供通用的指令
- 每台机器 2 GB 或更多的 RAM （如果少于这个数字将会影响你应用的运行内存)
- 2 CPU 核或更多
- 集群中的所有机器的网络彼此均能相互连接(公网和内网都可以)
- 节点之中不可以有重复的主机名、MAC 地址或 product_uuid。请参见这里了解更多详细信息。
- 开启机器上的某些端口。请参见这里 了解更多详细信息。
- 禁用交换分区。为了保证 kubelet 正常工作，你 **必须** 禁用交换分区。

确保每个节点上 MAC 地址和 product_uuid 的唯一性

- 你可以使用命令 `ip link` 或 `ifconfig -a` 来获取网络接口的 MAC 地址
- 可以使用 `sudo cat /sys/class/dmi/id/product_uuid` 命令对 product_uuid 校验

一般来讲，硬件设备会拥有唯一的地址，但是有些虚拟机的地址可能会重复。 Kubernetes 使用这些值来唯一确定集群中的节点。 如果这些值在每个节点上不唯一，可能会导致安装 失败。

检查网络适配器

如果你有一个以上的网络适配器，同时你的 Kubernetes 组件通过默认路由不可达，我们建议你预先添加 IP 路由规则，这样 Kubernetes 集群就可以通过对应的适配器完成连接。

允许 iptables 检查桥接流量

确保 `br_netfilter` 模块被加载。这一操作可以通过运行 `lsmod | grep br_netfilter` 来完成。若要显式加载该模块，可执行 `sudo modprobe br_netfilter` 。

为了让你的 Linux 节点上的 iptables 能够正确地查看桥接流量，你需要确保在你的 `sysctl` 配置中将 `net.bridge.bridge-nf-call-iptables` 设置为 1。例如：

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
br_netfilter
EOF

cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sudo sysctl --system
```

更多的相关细节可查看网络插件需求页面。

检查所需端口

启用这些必要的端口后才能使 Kubernetes 的各组件相互通信。可以使用 telnet 来检查端口是否启用，例如：

```
telnet 127.0.0.1 6443
```

你使用的 Pod 网络插件 (详见后续章节) 也可能需要开启某些特定端口。由于各个 Pod 网络插件的功能都有所不同， 请参阅他们各自文档中对端口的要求。

安装 runtime

为了在 Pod 中运行容器，Kubernetes 使用 容器运行时（Container Runtime）。

Linux 节点 | 其它操作系统

默认情况下，Kubernetes 使用 容器运行时接口（Container Runtime Interface，CRI） 来与你所选择的容器运行时交互。

如果你不指定运行时，则 kubeadm 会自动尝试检测到系统上已经安装的运行时， 方法是扫描一组众所周知的 Unix 域套接字。 下面的表格列举了一些容器运行时及其对应的套接字路径：

运行时	域套接字
Docker	/var/run/dockershim.sock
containerd	/run/containerd/containerd.sock
CRI-O	/var/run/crio/crio.sock

如果同时检测到 Docker 和 containerd，则优先选择 Docker。 这是必然的，因为 Docker 18.09 附带了 containerd 并且两者都是可以检测到的， 即使你仅安装了 Docker。 如果检测到其他两个或多个运行时，kubeadm 输出错误信息并退出。
kubelet 通过内置的 `dockershim` CRI 实现与 Docker 集成。

参阅容器运行时 以了解更多信息。

安装 kubeadm、kubelet 和 kubectl

你需要在每台机器上安装以下的软件包：

- `kubeadm`：用来初始化集群的指令。
- `kubelet`：在集群中的每个节点上用来启动 Pod 和容器等。
- `kubectl`：用来与集群通信的命令行工具。

kubeadm **不能** 帮你安装或者管理 `kubelet` 或 `kubectl`，所以你需要 确保它们与通过 kubeadm 安装的控制平面的版本相匹配。 如果不这样做，则存在发生版本偏差的风险，可能会导致一些预料之外的错误和问题。 然而，控制平面与 kubelet 间的相差一个次要版本不一致是支持的，但 kubelet 的版本不可以超过 API 服务器的版本。 例如，1.7.0 版本的 kubelet 可以完全兼容 1.8.0 版本的 API 服务器，反之则不可以。

有关安装 `kubectl` 的信息，请参阅安装和设置 kubectl文档。

> **警告：**
> 这些指南不包括系统升级时使用的所有 Kubernetes 程序包。这是因为 kubeadm 和 Kubernetes 有特殊的升级注意事项。

关于版本偏差的更多信息，请参阅以下文档：

- Kubernetes 版本与版本间的偏差策略
- Kubeadm 特定的版本偏差策略

基于 Debian 的发行版 | 基于 Red Hat 的发行版 | 无包管理器的情况

1. 更新 `apt` 包索引并安装使用 Kubernetes `apt` 仓库所需要的包：

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
```

2. 下载 Google Cloud 公开签名秘钥：

```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://pac
```

3. 添加 Kubernetes `apt` 仓库：

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https:
```

4. 更新 `apt` 包索引，安装 kubelet、kubeadm 和 kubectl，并锁定其版本：

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

kubelet 现在每隔几秒就会重启，因为它陷入了一个等待 kubeadm 指令的死循环。

配置 cgroup 驱动程序

容器运行时和 kubelet 都具有名字为 "cgroup driver" 的属性，该属性对于在 Linux 机器上管理 CGroups 而言非常重要。

警告：
你需要确保容器运行时和 kubelet 所使用的是相同的 cgroup 驱动，否则 kubelet 进程会失败。

相关细节可参见配置 cgroup 驱动。

故障排查

如果你在使用 kubeadm 时遇到困难，请参阅我们的 故障排查文档。

接下来

- 使用 kubeadm 创建集群

3.1.2 - 对 kubeadm 进行故障排查

与任何程序一样，你可能会在安装或者运行 kubeadm 时遇到错误。 本文列举了一些常见的故障场景，并提供可帮助你理解和解决这些问题的步骤。

如果你的问题未在下面列出，请执行以下步骤：

- 如果你认为问题是 kubeadm 的错误：
  - 转到 github.com/kubernetes/kubeadm 并搜索存在的问题。
  - 如果没有问题，请 打开 并遵循问题模板。
- 如果你对 kubeadm 的工作方式有疑问，可以在 Slack 上的 #kubeadm 频道提问， 或者在 StackOverflow 上提问。 请加入相关标签，例如 `#kubernetes` 和 `#kubeadm` ，这样其他人可以帮助你。

在安装过程中没有找到 `ebtables` 或者其他类似的可执行文件

如果在运行 `kubeadm init` 命令时，遇到以下的警告

```
[preflight] WARNING: ebtables not found in system path
[preflight] WARNING: ethtool not found in system path
```

那么或许在你的节点上缺失 `ebtables` 、 `ethtool` 或者类似的可执行文件。 你可以使用以下命令安装它们：

- 对于 Ubuntu/Debian 用户，运行 `apt install ebtables ethtool` 命令。
- 对于 CentOS/Fedora 用户，运行 `yum install ebtables ethtool` 命令。

在安装过程中，kubeadm 一直等待控制平面就绪

如果你注意到 `kubeadm init` 在打印以下行后挂起：

```
[apiclient] Created API client, waiting for the control plane to become ready
```

这可能是由许多问题引起的。最常见的是：

- 网络连接问题。在继续之前，请检查你的计算机是否具有全部联通的网络连接。
- 容器运行时的 cgroup 驱动不同于 kubelet 使用的 cgroup 驱动。要了解如何正确配置 cgroup 驱动， 请参阅配置 cgroup 驱动。
- 控制平面上的 Docker 容器持续进入崩溃状态或（因其他原因）挂起。你可以运行 `docker ps` 命令来检查以及 `docker logs` 命令来检视每个容器的运行日志。 对于其他容器运行时，请参阅使用 crictl 对 Kubernetes 节点进行调试。

当删除托管容器时 kubeadm 阻塞

如果 Docker 停止并且不删除 Kubernetes 所管理的所有容器，可能发生以下情况：

```
sudo kubeadm reset
```

```
[preflight] Running pre-flight checks
[reset] Stopping the kubelet service
[reset] Unmounting mounted directories in "/var/lib/kubelet"
[reset] Removing kubernetes-managed containers
(block)
```

一个可行的解决方案是重新启动 Docker 服务，然后重新运行 `kubeadm reset`：

```
sudo systemctl restart docker.service
sudo kubeadm reset
```

检查 docker 的日志也可能有用：

```
journalctl -ul docker
```

Pods 处于 `RunContainerError`、`CrashLoopBackOff` 或者 `Error` 状态

在 `kubeadm init` 命令运行后，系统中不应该有 pods 处于这类状态。

- 在 `kubeadm init` 命令执行完后，如果有 pods 处于这些状态之一，请在 kubeadm 仓库提起一个 issue。 `coredns` (或者 `kube-dns`) 应该处于 `Pending` 状态， 直到你部署了网络插件为止。
- 如果在部署完网络插件之后，有 Pods 处于 `RunContainerError` 、 `CrashLoopBackOff` 或 `Error` 状态之一，并且 `coredns` （或者 `kube-dns`）仍处于 `Pending` 状态， 那很可能是你安装的网络插件由于某种原因无法工作。你或许需要授予它更多的 RBAC 特权或使用较新的版本。请在 Pod Network 提供商的问题跟踪器中提交问题， 然后在此处分类问题。
- 如果你安装的 Docker 版本早于 1.12.1，请在使用 `systemd` 来启动 `dockerd` 和重启 `docker` 时， 删除 `MountFlags=slave` 选项。 你可以在 `/usr/lib/systemd/system/docker.service` 中看到 MountFlags。 MountFlags 可能会干扰 Kubernetes 挂载的卷， 并使 Pods 处于 `CrashLoopBackOff` 状态。 当 Kubernetes 不能找到 `var/run/secrets/kubernetes.io/serviceaccount` 文件时会发生错误。

`coredns` 停滞在 `Pending` 状态

这一行为是 **预期之中** 的，因为系统就是这么设计的。 kubeadm 的网络供应商是中立的，因此管理员应该选择 安装 pod 的网络插件。 你必须完成 Pod 的网络配置，然后才能完全部署 CoreDNS。在网络被配置好之前，DNS 组件会一直处于 `Pending` 状态。

`HostPort` 服务无法工作

此 `HostPort` 和 `HostIP` 功能是否可用取决于你的 Pod 网络配置。请联系 Pod 网络插件的作者，以确认 `HostPort` 和 `HostIP` 功能是否可用。

已验证 Calico、Canal 和 Flannel CNI 驱动程序支持 HostPort。

有关更多信息，请参考 CNI portmap 文档.

如果你的网络提供商不支持 portmap CNI 插件，你或许需要使用 NodePort 服务的功能 或者使用 `HostNetwork=true` 。

无法通过其服务 IP 访问 Pod

- 许多网络附加组件尚未启用 hairpin 模式 该模式允许 Pod 通过其服务 IP 进行访问。这是与 CNI 有关的问题。 请与网络附加组件提供商联系，以获取他们所提供的 hairpin 模式的最新状态。

- 如果你正在使用 VirtualBox (直接使用或者通过 Vagrant 使用)，你需要 确保 `hostname -i` 返回一个可路由的 IP 地址。默认情况下，第一个接口连接不能路由的仅主机网络。 解决方法是修改 `/etc/hosts` ，请参考示例 Vagrantfile。

TLS 证书错误

以下错误指出证书可能不匹配。

```
# kubectl get pods
Unable to connect to the server: x509: certificate signed by unknown authority (possibly
```

- 验证 `$HOME/.kube/config` 文件是否包含有效证书，并 在必要时重新生成证书。在 kubeconfig 文件中的证书是 base64 编码的。 该 `base64 -d` 命令可以用来解码证书，`openssl x509 -text -noout` 命令 可以用于查看证书信息。

- 使用如下方法取消设置 `KUBECONFIG` 环境变量的值：

```
unset KUBECONFIG
```

  或者将其设置为默认的 `KUBECONFIG` 位置：

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

- 另一个方法是覆盖 `kubeconfig` 的现有用户 "管理员"：

```
mv  $HOME/.kube $HOME/.kube.bak
mkdir $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Kubelet 客户端证书轮换失败

默认情况下，kubeadm 使用 `/etc/kubernetes/kubelet.conf` 中指定的 `/var/lib/kubelet/pki/kubelet-client-current.pem` 符号链接 来配置 kubelet 自动轮换客户端证书。如果此轮换过程失败，你可能会在 kube-apiserver 日志中看到 诸如 `x509: certificate has expired or is not yet valid` 之类的错误。要解决此问题，你必须执行以下步骤：

1. 从故障节点备份和删除 `/etc/kubernetes/kubelet.conf` 和 `/var/lib/kubelet/pki/kubelet-client*` 。
2. 在集群中具有 `/etc/kubernetes/pki/ca.key` 的、正常工作的控制平面节点上 执行 `kubeadm kubeconfig user --org system:nodes --client-name system:node:$NODE > kubelet.conf` 。`$NODE` 必须设置为集群中现有故障节点的名称。 手动修改生成的 `kubelet.conf` 以调整集群名称和服务器端点， 或传递 `kubeconfig user --config` （此命令接受 `InitConfiguration` ）。 如果你的集群没有 `ca.key` ，你必须在外部对 `kubelet.conf` 中的嵌入式证书进行签名。
3. 将得到的 `kubelet.conf` 文件复制到故障节点上，作为 `/etc/kubernetes/kubelet.conf` 。
4. 在故障节点上重启 kubelet（ `systemctl restart kubelet` ），等待 `/var/lib/kubelet/pki/kubelet-client-current.pem` 重新创建。

5. 手动编辑 `kubelet.conf` 指向轮换的 kubelet 客户端证书，方法是将 `client-certificate-data` 和 `client-key-data` 替换为：

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

6. 重新启动 kubelet。
7. 确保节点状况变为 `Ready` 。

在 Vagrant 中使用 flannel 作为 pod 网络时的默认 NIC

以下错误可能表明 Pod 网络中出现问题：

```
Error from server (NotFound): the server could not find the requested resource
```

- 如果你正在 Vagrant 中使用 flannel 作为 pod 网络，则必须指定 flannel 的默认接口名称。

  Vagrant 通常为所有 VM 分配两个接口。第一个为所有主机分配了 IP 地址 `10.0.2.15` ，用于获得 NATed 的外部流量。

  这可能会导致 flannel 出现问题，它默认为主机上的第一个接口。这导致所有主机认为它们具有 相同的公共 IP 地址。为防止这种情况，传递 `--iface eth1` 标志给 flannel 以便选择第二个接口。

容器使用的非公共 IP

在某些情况下 `kubectl logs` 和 `kubectl run` 命令或许会返回以下错误，即便除此之外集群一切功能正常：

```
Error from server: Get https://10.19.0.41:10250/containerLogs/default/mysql-ddc65b868-gl
```

- 这或许是由于 Kubernetes 使用的 IP 无法与看似相同的子网上的其他 IP 进行通信的缘故， 可能是由机器提供商的政策所导致的。
- Digital Ocean 既分配一个共有 IP 给 `eth0` ，也分配一个私有 IP 在内部用作其浮动 IP 功能的锚点， 然而 `kubelet` 将选择后者作为节点的 `InternalIP` 而不是公共 IP

  使用 `ip addr show` 命令代替 `ifconfig` 命令去检查这种情况，因为 `ifconfig` 命令 不会显示有问题的别名 IP 地址。或者指定的 Digital Ocean 的 API 端口允许从 droplet 中 查询 anchor IP：

  ```
  curl http://169.254.169.254/metadata/v1/interfaces/public/0/anchor_ipv4/address
  ```

  解决方法是通知 `kubelet` 使用哪个 `--node-ip` 。当使用 Digital Ocean 时，可以是公网IP（分配给 `eth0` 的）， 或者是私网IP（分配给 `eth1` 的）。私网 IP 是可选的。 kubadm `NodeRegistrationOptions` 结构 的 `KubeletExtraArgs` 部分被用来处理这种情况。

  然后重启 `kubelet` ：

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

coredns pods 有 CrashLoopBackOff 或者 Error 状态

如果有些节点运行的是旧版本的 Docker，同时启用了 SELinux，你或许会遇到 `coredns` pods 无法启动的情况。 要解决此问题，你可以尝试以下选项之一：

- 升级到 Docker 的较新版本。
- 禁用 SELinux.
- 修改 `coredns` 部署以设置 `allowPrivilegeEscalation` 为 `true`：

```
kubectl -n kube-system get deployment coredns -o yaml | \
  sed 's/allowPrivilegeEscalation: false/allowPrivilegeEscalation: true/g' | \
  kubectl apply -f -
```

CoreDNS 处于 `CrashLoopBackOff` 时的另一个原因是当 Kubernetes 中部署的 CoreDNS Pod 检测到环路时。有许多解决方法 可以避免在每次 CoreDNS 监测到循环并退出时，Kubernetes 尝试重启 CoreDNS Pod 的情况。

> **警告：** 禁用 SELinux 或设置 `allowPrivilegeEscalation` 为 `true` 可能会损害集群的安全性。

etcd pods 持续重启

如果你遇到以下错误：

```
rpc error: code = 2 desc = oci runtime error: exec failed: container_linux.go:247: starti
```

如果你使用 Docker 1.13.1.84 运行 CentOS 7 就会出现这种问题。 此版本的 Docker 会阻止 kubelet 在 etcd 容器中执行。

为解决此问题，请选择以下选项之一：

- 回滚到早期版本的 Docker，例如 1.13.1-75

```
yum downgrade docker-1.13.1-75.git8633870.el7.centos.x86_64 docker-client-1.13.1-75
```

- 安装较新的推荐版本之一，例如 18.06:

```
sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-
yum install docker-ce-18.06.1.ce-3.el7.x86_64
```

无法将以逗号分隔的值列表传递给 `--component-extra-args` 标志内的参数

`kubeadm init` 标志例如 `--component-extra-args` 允许你将自定义参数传递给像 kube-apiserver 这样的控制平面组件。然而，由于解析（`mapStringString`）的基础类型值，此机制将受到限制。

如果你决定传递一个支持多个逗号分隔值（例如 `--apiserver-extra-args "enable-admission-plugins=LimitRanger,NamespaceExists"`）参数，将出现 `flag: malformed pair, expect string=string` 错误。发生这种问题是因为参数列表 `--apiserver-extra-args` 预期的是 `key=value` 形式，而这里的 `NamespacesExists` 被误认为是缺少取值的键名。

一种解决方法是尝试分离 `key=value` 对，像这样：`--apiserver-extra-args "enable-admission-plugins=LimitRanger,enable-admission-plugins=NamespaceExists"` 但这将导致键 `enable-admission-plugins` 仅有值 `NamespaceExists`。

已知的解决方法是使用 kubeadm 配置文件。

在节点被云控制管理器初始化之前，kube-proxy 就被调度了

在云环境场景中，可能出现在云控制管理器完成节点地址初始化之前，kube-proxy 就被调度到新节点了。这会导致 kube-proxy 无法正确获取节点的 IP 地址，并对管理负载平衡器的代理功能产生连锁反应。

在 kube-proxy Pod 中可以看到以下错误：

```
server.go:610] Failed to retrieve node IP: host IP unknown; known addresses: []
proxier.go:340] invalid nodeIP, initializing kube-proxy with 127.0.0.1 as nodeIP
```

一种已知的解决方案是修补 kube-proxy DaemonSet，以允许在控制平面节点上调度它，而不管它们的条件如何，将其与其他节点保持隔离，直到它们的初始保护条件消除：

```
kubectl -n kube-system patch ds kube-proxy -p='{ "spec": { "template": { "spec": { "tole
```

此问题的跟踪在这里。

节点上的 `/usr` 被以只读方式挂载

在类似 Fedora CoreOS 或者 Flatcar Container Linux 这类 Linux 发行版本中，目录 `/usr` 是以只读文件系统的形式挂载的。在支持 FlexVolume时，类似 kubelet 和 kube-controller-manager 这类 Kubernetes 组件使用默认路径 `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/`，而 FlexVolume 的目录 *必须是可写入的*，该功能特性才能正常工作。（**注意**：FlexVolume 在 Kubernetes v1.23 版本中已被弃用）

为了解决这个问题，你可以使用 kubeadm 的配置文件 来配置 FlexVolume 的目录。

在（使用 `kubeadm init` 创建的）主控制节点上，使用 `-config` 参数传入如下文件：

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  kubeletExtraArgs:
    volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    flex-volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
```

在加入到集群中的节点上，使用下面的文件：

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
nodeRegistration:
  kubeletExtraArgs:
    volume-plugin-dir: "/opt/libexec/kubernetes/kubelet-plugins/volume/exec/"
```

或者，你要可以更改 `/etc/fstab` 使得 `/usr` 目录能够以可写入的方式挂载，不过 请注意这样做本质上是在更改 Linux 发行版的某种设计原则。

`kubeadm upgrade plan` 输出错误信息 `context deadline exceeded`

在使用 `kubeadm` 来升级某运行外部 etcd 的 Kubernetes 集群时可能显示这一错误信息。 这并不是一个非常严重的一个缺陷，之所以出现此错误信息，原因是老的 kubeadm 版本会对外部 etcd 集群执行版本检查。你可以继续执行 `kubeadm upgrade apply ...` 。

这一问题已经在 1.19 版本中得到修复。

`kubeadm reset` 会卸载 `/var/lib/kubelet`

如果已经挂载了 `/var/lib/kubelet` 目录，执行 `kubeadm reset` 操作的时候 会将其卸载。

要解决这一问题，可以在执行了 `kubeadm reset` 操作之后重新挂载 `/var/lib/kubelet` 目录。

这是一个在 1.15 中引入的故障，已经在 1.20 版本中修复。

无法在 kubeadm 集群中安全地使用 metrics-server

在 kubeadm 集群中可以通过为 metrics-server 设置 `--kubelet-insecure-tls` 来以不安全的形式使用该服务。 建议不要在生产环境集群中这样使用。

如果你需要在 metrics-server 和 kubelet 之间使用 TLS，会有一个问题， kubeadm 为 kubelet 部署的是自签名的服务证书。这可能会导致 metrics-server 端报告下面的错误信息：

```
x509: certificate signed by unknown authority
x509: certificate is valid for IP-foo not IP-bar
```

参见为 kubelet 启用签名的服务证书 以进一步了解如何在 kubeadm 集群中配置 kubelet 使用正确签名了的服务证书。

另请参阅How to run the metrics-server securely。

3.1.3 - 使用 kubeadm 创建集群

使用 `kubeadm`，你能创建一个符合最佳实践的最小化 Kubernetes 集群。事实上，你可以使用 `kubeadm` 配置一个通过 Kubernetes 一致性测试 的集群。 `kubeadm` 还支持其他集群生命周期功能， 例如 启动引导令牌 和集群升级。

kubeadm 工具很棒，如果你需要：

- 一个尝试 Kubernetes 的简单方法。
- 一个现有用户可以自动设置集群并测试其应用程序的途径。
- 其他具有更大范围的生态系统和/或安装工具中的构建模块。

你可以在各种机器上安装和使用 `kubeadm`： 笔记本电脑， 一组云服务器，Raspberry Pi 等。无论是部署到云还是本地， 你都可以将 `kubeadm` 集成到预配置系统中，例如 Ansible 或 Terraform。

准备开始

要遵循本指南，你需要：

- 一台或多台运行兼容 deb/rpm 的 Linux 操作系统的计算机；例如：Ubuntu 或 CentOS。
- 每台机器 2 GB 以上的内存，内存不足时应用会受限制。
- 用作控制平面节点的计算机上至少有2个 CPU。
- 集群中所有计算机之间具有完全的网络连接。你可以使用公共网络或专用网络。

你还需要使用可以在新集群中部署特定 Kubernetes 版本对应的 `kubeadm`。

Kubernetes 版本及版本倾斜支持策略 适用于 `kubeadm` 以及整个 Kubernetes。 查阅该策略以了解支持哪些版本的 Kubernetes 和 `kubeadm`。 该页面是为 Kubernetes v1.23 编写的。

`kubeadm` 工具的整体功能状态为一般可用性（GA）。一些子功能仍在积极开发中。 随着工具的发展，创建集群的实现可能会略有变化，但总体实现应相当稳定。

> **说明：** 根据定义，在 `kubeadm alpha` 下的所有命令均在 alpha 级别上受支持。

目标

- 安装单个控制平面的 Kubernetes 集群
- 在集群上安装 Pod 网络，以便你的 Pod 可以相互连通

操作指南

在你的主机上安装 kubeadm

查看 "安装 kubeadm"。

> **说明：**
> 如果你已经安装了kubeadm，执行 `apt-get update && apt-get upgrade` 或 `yum update` 以获取 kubeadm 的最新版本。
>
> 升级时，kubelet 每隔几秒钟重新启动一次， 在 crashloop 状态中等待 kubeadm 发布指令。crashloop 状态是正常现象。 初始化控制平面后，kubelet 将正常运行。

准备所需的容器镜像

这个步骤是可选的，只适用于你希望 `kubeadm init` 和 `kubeadm join` 不去下载存放在 `k8s.gcr.io` 上的默认的容器镜像的情况。

当你在离线的节点上创建一个集群的时候，Kubeadm 有一些命令可以帮助你预拉取所需的镜像。阅读离线运行 kubeadm 获取更多的详情。

Kubeadm 允许你给所需要的镜像指定一个自定义的镜像仓库。 阅读使用自定义镜像 获取更多的详情。

初始化控制平面节点

控制平面节点是运行控制平面组件的机器， 包括 etcd （集群数据库） 和 API Server （命令行工具 kubectl 与之通信）。

1. （推荐）如果计划将单个控制平面 kubeadm 集群升级成高可用， 你应该指定 `--control-plane-endpoint` 为所有控制平面节点设置共享端点。 端点可以是负载均衡器的 DNS 名称或 IP 地址。
2. 选择一个 Pod 网络插件，并验证是否需要为 `kubeadm init` 传递参数。 根据你选择的第三方网络插件，你可能需要设置 `--pod-network-cidr` 的值。 请参阅 安装Pod网络附加组件。
3. （可选）从版本1.14开始， `kubeadm` 尝试使用一系列众所周知的域套接字路径来检测 Linux 上的容器运行时。 要使用不同的容器运行时， 或者如果在预配置的节点上安装了多个容器，请为 `kubeadm init` 指定 `--cri-socket` 参数。 请参阅安装运行时。
4. （可选）除非另有说明，否则 `kubeadm` 使用与默认网关关联的网络接口来设置此控制平面节点 API server 的广播地址。 要使用其他网络接口，请为 `kubeadm init` 设置 `--apiserver-advertise-address=<ip-address>` 参数。 要部署使用 IPv6 地址的 Kubernetes 集群， 必须指定一个 IPv6 地址，例如 `--apiserver-advertise-address=fd00::101`

要初始化控制平面节点，请运行：

```
kubeadm init <args>
```

关于 apiserver-advertise-address 和 ControlPlaneEndpoint 的注意事项

`--apiserver-advertise-address` 可用于为控制平面节点的 API server 设置广播地址， `--control-plane-endpoint` 可用于为所有控制平面节点设置共享端点。

`--control-plane-endpoint` 允许 IP 地址和可以映射到 IP 地址的 DNS 名称。 请与你的网络管理员联系，以评估有关此类映射的可能解决方案。

这是一个示例映射：

```
192.168.0.102 cluster-endpoint
```

其中 `192.168.0.102` 是此节点的 IP 地址， `cluster-endpoint` 是映射到该 IP 的自定义 DNS 名称。这将允许你将 `--control-plane-endpoint=cluster-endpoint` 传递给 `kubeadm init` ，并将相同的 DNS 名称传递给 `kubeadm join` 。 稍后你可以修改 `cluster-endpoint` 以指向高可用性方案中的负载均衡器的地址。

kubeadm 不支持将没有 `--control-plane-endpoint` 参数的单个控制平面集群转换为高可用性集群。

更多信息

有关 `kubeadm init` 参数的更多信息，请参见 kubeadm 参考指南。

要使用配置文件配置 `kubeadm init` 命令，请参见带配置文件使用 kubeadm init。

要自定义控制平面组件，包括可选的对控制平面组件和 etcd 服务器的活动探针提供 IPv6 支持，请参阅自定义参数。

要再次运行 `kubeadm init` ，你必须首先卸载集群。

如果将具有不同架构的节点加入集群， 请确保已部署的 DaemonSet 对这种体系结构具有容器镜像支持。

`kubeadm init` 首先运行一系列预检查以确保机器 准备运行 Kubernetes。这些预检查会显示警告并在错误时退出。然后 `kubeadm init` 下载并安装集群控制平面组件。这可能会需要几分钟。 完成之后你应该看到：

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a Pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  /docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join <control-plane-host>:<control-plane-port> --token <token> --discovery-toke
```

要使非 root 用户可以运行 kubectl，请运行以下命令， 它们也是 `kubeadm init` 输出的一部分：

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

或者，如果你是 `root` 用户，则可以运行：

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

警告：
kubeadm 对 `admin.conf` 中的证书进行签名时，将其配置为 `Subject: O = system:masters, CN = kubernetes-admin`。 `system:masters` 是一个例外的、超级用户组，可以绕过鉴权层（例如 RBAC）。 不要将 `admin.conf` 文件与任何人共享，应该使用 `kubeadm kubeconfig user` 命令为其他用户生成 kubeconfig 文件，完成对他们的定制授权。

记录 `kubeadm init` 输出的 `kubeadm join` 命令。 你需要此命令将节点加入集群。

令牌用于控制平面节点和加入节点之间的相互身份验证。 这里包含的令牌是密钥。确保它的安全，因为拥有此令牌的任何人都可以将经过身份验证的节点添加到你的集群中。 可以使用 `kubeadm token` 命令列出，创建和删除这些令牌。 请参阅 kubeadm 参考指南。

安装 Pod 网络附加组件

注意：
本节包含有关网络设置和部署顺序的重要信息。 在继续之前，请仔细阅读所有建议。

你必须部署一个基于 Pod 网络插件的 容器网络接口 (CNI)，以便你的 Pod 可以相互通信。 在安装网络之前，集群 DNS (CoreDNS) 将不会启动。

- 注意你的 Pod 网络不得与任何主机网络重叠： 如果有重叠，你很可能会遇到问题。 （如果你发现网络插件的首选 Pod 网络与某些主机网络之间存在冲突， 则应考虑使用一个合适的 CIDR 块来代替， 然后在执行 `kubeadm init` 时使用 `--pod-network-cidr` 参数并在你的网络插件的 YAML 中替换它）。

- 默认情况下， `kubeadm` 将集群设置为使用和强制使用 RBAC（基于角色的访问控制）。确保你的 Pod 网络插件支持 RBAC，以及用于部署它的 manifests 也是如此。
- 如果要为集群使用 IPv6（双协议栈或仅单协议栈 IPv6 网络）， 请确保你的 Pod 网络插件支持 IPv6。 IPv6 支持已在 CNI v0.6.0 版本中添加。

说明： kubeadm 应该是与 CNI 无关的，对 CNI 驱动进行验证目前不在我们的端到端测试范畴之内。 如果你发现与 CNI 插件相关的问题，应在其各自的问题跟踪器中记录而不是在 kubeadm 或 kubernetes 问题跟踪器中记录。

一些外部项目为 Kubernetes 提供使用 CNI 的 Pod 网络，其中一些还支持网络策略。

请参阅实现 Kubernetes 网络模型 的附加组件列表。

你可以使用以下命令在控制平面节点或具有 kubeconfig 凭据的节点上安装 Pod 网络附加组件：

```
kubectl apply -f <add-on.yaml>
```

每个集群只能安装一个 Pod 网络。

安装 Pod 网络后，您可以通过在 `kubectl get pods --all-namespaces` 输出中检查 CoreDNS Pod 是否 `Running` 来确认其是否正常运行。 一旦 CoreDNS Pod 启用并运行，你就可以继续加入节点。

如果您的网络无法正常工作或 CoreDNS 不在“运行中”状态，请查看 `kubeadm` 的 故障排除指南。

控制平面节点隔离

默认情况下，出于安全原因，你的集群不会在控制平面节点上调度 Pod。 如果你希望能够在控制平面节点上调度 Pod， 例如用于开发的单机 Kubernetes 集群，请运行：

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

输出看起来像：

```
node "test-01" untainted
taint "node-role.kubernetes.io/master:" not found
taint "node-role.kubernetes.io/master:" not found
```

这将从任何拥有 `node-role.kubernetes.io/master` taint 标记的节点中移除该标记， 包括控制平面节点，这意味着调度程序将能够在任何地方调度 Pods。

加入节点

节点是你的工作负载（容器和 Pod 等）运行的地方。要将新节点添加到集群，请对每台计算机执行以下操作：

- SSH 到机器
- 成为 root （例如 `sudo su -` ）
- 运行 `kubeadm init` 输出的命令。例如：

```
kubeadm join --token <token> <control-plane-host>:<control-plane-port> --discovery-token
```

如果没有令牌，可以通过在控制平面节点上运行以下命令来获取令牌：

```
kubeadm token list
```

输出类似于以下内容：

```
TOKEN                     TTL  EXPIRES              USAGES           DESCRIPTION
8ewj1p.9r9hcjoqgajrj4gi   23h  2018-06-12T02:51:28Z authentication,  The default bootstrap
                                                   signing          token generated by
                                                                    'kubeadm init'.
```

默认情况下，令牌会在24小时后过期。如果要在当前令牌过期后将节点加入集群， 则可以通过在控制平面节点上运行以下命令来创建新令牌：

```
kubeadm token create
```

输出类似于以下内容：

```
5didvk.d09sbcov8ph2amjw
```

如果你没有 `--discovery-token-ca-cert-hash` 的值，则可以通过在控制平面节点上执行以下命令链来获取它：

```
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/
   openssl dgst -sha256 -hex | sed 's/^.* //'
```

输出类似于以下内容：

```
8cb2de97839780a412b93877f8507ad6c94f73add17d5d7058e91741c9d5ec78
```

说明： 要为 `<control-plane-host>:<control-plane-port>` 指定 IPv6 元组，必须将 IPv6 地址括在方括号中，例如：`[fd00::101]:2073`

输出应类似于：

```
[preflight] Running pre-flight checks

... (log output of join workflow) ...

Node join complete:
* Certificate signing request sent to control-plane and response
  received.
* Kubelet informed of new secure connection details.

Run 'kubectl get nodes' on control-plane to see this machine join.
```

几秒钟后，当你在控制平面节点上执行 `kubectl get nodes` ，你会注意到该节点出现在输出中。

说明： 由于集群节点通常是按顺序初始化的，CoreDNS Pods 很可能都运行在第一个控制面节点上。 为了提供更高的可用性，请在加入至少一个新节点后 使用 `kubectl -n kube-system rollout restart deployment coredns` 命令，重新平衡 CoreDNS Pods。

（可选）从控制平面节点以外的计算机控制集群

为了使 kubectl 在其他计算机（例如笔记本电脑）上与你的集群通信， 你需要将管理员 kubeconfig 文件从控制平面节点复制到工作站，如下所示：

```
scp root@<control-plane-host>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf get nodes
```

说明：
上面的示例假定为 root 用户启用了SSH访问。如果不是这种情况， 你可以使用 `scp` 将 admin.conf 文件复制给其他允许访问的用户。

admin.conf 文件为用户提供了对集群的超级用户特权。 该文件应谨慎使用。对于普通用户，建议生成一个你为其授予特权的唯一证书。 你可以使用 `kubeadm alpha kubeconfig user --client-name <CN>` 命令执行此操作。 该命令会将 KubeConfig 文件打印到 STDOUT，你应该将其保存到文件并分发给用户。 之后，使用 `kubectl create (cluster)rolebinding` 授予特权。

（可选）将API服务器代理到本地主机

如果要从集群外部连接到 API 服务器，则可以使用 `kubectl proxy`：

```
scp root@<control-plane-host>:/etc/kubernetes/admin.conf .
kubectl --kubeconfig ./admin.conf proxy
```

你现在可以在本地访问API服务器 http://localhost:8001/api/v1

清理

如果你在集群中使用了一次性服务器进行测试，则可以关闭这些服务器，而无需进一步清理。你可以使用 `kubectl config delete-cluster` 删除对集群的本地引用。

但是，如果要更干净地取消配置群集， 则应首先清空节点并确保该节点为空， 然后取消配置该节点。

删除节点

使用适当的凭证与控制平面节点通信，运行：

```
kubectl drain <node name> --delete-emptydir-data --force --ignore-daemonsets
```

在删除节点之前，请重置 `kubeadm` 安装的状态：

```
kubeadm reset
```

重置过程不会重置或清除 iptables 规则或 IPVS 表。如果你希望重置 iptables，则必须手动进行：

```
iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
```

如果要重置 IPVS 表，则必须运行以下命令：

```
ipvsadm -C
```

现在删除节点：

```
kubectl delete node <node name>
```

如果你想重新开始，只需运行 `kubeadm init` 或 `kubeadm join` 并加上适当的参数。

清理控制平面

你可以在控制平面主机上使用 `kubeadm reset` 来触发尽力而为的清理。

有关此子命令及其选项的更多信息，请参见 `kubeadm reset` 参考文档。

下一步

- 使用 Sonobuoy 验证集群是否正常运行。
- 有关使用 kubeadm 升级集群的详细信息，请参阅升级 kubeadm 集群。
- 在 kubeadm 参考文档中了解有关高级 `kubeadm` 用法的信息。
- 了解有关 Kubernetes 概念和 `kubectl` 的更多信息。
- 有关 Pod 网络附加组件的更多列表，请参见集群网络页面。
- 请参阅附加组件列表以探索其他附加组件， 包括用于 Kubernetes 集群的日志记录，监视，网络策略，可视化和控制的工具。
- 配置集群如何处理集群事件的日志以及 在 Pods 中运行的应用程序。 有关所涉及内容的概述，请参见日志架构。

反馈

- 有关 bugs, 访问 kubeadm GitHub issue tracker
- 有关支持, 访问 #kubeadm Slack 频道
- General SIG 集群生命周期开发 Slack 频道: #sig-cluster-lifecycle
- SIG 集群生命周期 SIG information
- SIG 集群生命周期邮件列表: kubernetes-sig-cluster-lifecycle

版本倾斜政策

版本 v1.23 的kubeadm 工具可以使用版本 v1.23 或 v1.22 的控制平面部署集群。kubeadm v1.23 还可以升级现有的 kubeadm 创建的 v1.22 版本的集群。

由于我们不能预见未来，kubeadm CLI v1.23 可能会或可能无法部署 v1.24 集群。

这些资源提供了有关 kubelet 与控制平面以及其他 Kubernetes 组件之间受支持的版本倾斜的更多信息：

- Kubernetes 版本和版本偏斜政策
- Kubeadm-specific 安装指南

局限性

集群弹性

此处创建的集群具有单个控制平面节点，运行单个 etcd 数据库。 这意味着如果控制平面节点发生故障，你的集群可能会丢失数据并且可能需要从头开始重新创建。

解决方法:

- 定期备份 etcd。 kubeadm 配置的 etcd 数据目录位于控制平面节点上的 `/var/lib/etcd` 中。
- 使用多个控制平面节点。你可以阅读 可选的高可用性拓扑 选择集群拓扑提供的 高可用性.

平台兼容性

kubeadm deb/rpm 软件包和二进制文件是为 amd64，arm (32-bit)，arm64，ppc64le 和 s390x 构建的遵循多平台提案。

从 v1.12 开始还支持用于控制平面和附加组件的多平台容器镜像。

只有一些网络提供商为所有平台提供解决方案。请查阅上方的网络提供商清单或每个提供商的文档以确定提供商是否支持你选择的平台。

故障排除

如果你在使用 kubeadm 时遇到困难，请查阅我们的故障排除文档。

3.1.4 - 使用 kubeadm API 定制组件

本页面介绍了如何自定义 kubeadm 部署的组件。 你可以使用 `ClusterConfiguration` 结构中定义的参数，或者在每个节点上应用补丁来定制控制平面组件。 你可以使用 `KubeletConfiguration` 和 `KubeProxyConfiguration` 结构分别定制 kubelet 和 kube-proxy 组件。

所有这些选项都可以通过 kubeadm 配置 API 实现。 有关配置中的每个字段的详细信息，你可以导航到我们的 API 参考页面 。

> **说明：**
> kubeadm 目前不支持对 CoreDNS 部署进行定制。 你必须手动更新 `kube-system/coredns` ConfigMap 并在更新后重新创建 CoreDNS Pods。 或者，你可以跳过默认的 CoreDNS 部署并部署你自己的 CoreDNS 变种。 有关更多详细信息，请参阅在 kubeadm 中使用 init phases.

FEATURE STATE: `Kubernetes 1.12 [stable]`

使用 `ClusterConfiguration` 中的标志自定义控制平面

kubeadm `ClusterConfiguration` 对象为用户提供了一种方法， 用以覆盖传递给控制平面组件（如 APIServer、ControllerManager、Scheduler 和 Etcd）的默认参数。 各组件配置使用如下字段定义：

- `apiServer`
- `controllerManager`
- `scheduler`
- `etcd`

这些结构包含一个通用的 `extraArgs` 字段，该字段由 `key: value` 组成。 要覆盖控制平面组件的参数：

1. 将适当的字段 `extraArgs` 添加到配置中。
2. 向字段 `extraArgs` 添加要覆盖的参数值。
3. 用 `--config <YOUR CONFIG YAML>` 运行 `kubeadm init` 。

> **说明：**
> 你可以通过运行 `kubeadm config print init-defaults` 并将输出保存到你所选的文件中， 以默认值形式生成 `ClusterConfiguration` 对象。

> **说明：**
> `ClusterConfiguration` 对象目前在 kubeadm 集群中是全局的。 这意味着你添加的任何标志都将应用于同一组件在不同节点上的所有实例。 要在不同节点上为每个组件应用单独的配置，您可以使用补丁。

> **说明：**
> 当前不支持重复的参数（keys）或多次传递相同的参数 `--foo` 。 要解决此问题，你必须使用补丁。

APIServer 参数

有关详细信息，请参阅 kube-apiserver 参考文档。

使用示例：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
apiServer:
  extraArgs:
    anonymous-auth: "false"
    enable-admission-plugins: AlwaysPullImages,DefaultStorageClass
    audit-log-path: /home/johndoe/audit.log
```

ControllerManager 参数

有关详细信息，请参阅 kube-controller-manager 参考文档。

使用示例：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
controllerManager:
  extraArgs:
    cluster-signing-key-file: /home/johndoe/keys/ca.key
    deployment-controller-sync-period: "50"
```

Scheduler 参数

有关详细信息，请参阅 kube-scheduler 参考文档。

使用示例：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: v1.16.0
scheduler:
  extraArgs:
    config: /etc/kubernetes/scheduler-config.yaml
  extraVolumes:
    - name: schedulerconfig
      hostPath: /home/johndoe/schedconfig.yaml
      mountPath: /etc/kubernetes/scheduler-config.yaml
      readOnly: true
      pathType: "File"
```

Etcd 参数

有关详细信息，请参阅 etcd 服务文档.

使用示例：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
etcd:
  local:
    extraArgs:
      election-timeout: 1000
```

使用补丁定制控制平面

FEATURE STATE: `Kubernetes v1.22 [beta]`

Kubeadm 允许将包含补丁文件的目录传递给各个节点上的 `InitConfiguration` 和 `JoinConfiguration`。 这些补丁可被用作控制平面组件清单写入磁盘之前的最后一个自定义步骤。

可以使用 `--config <你的 YAML 格式控制文件>` 将配置文件传递给 `kubeadm init`：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
patches:
  directory: /home/user/somedir
```

> **说明：**
> 对于 `kubeadm init`，你可以传递一个包含 `ClusterConfiguration` 和 `InitConfiguration` 的文件，以 `---` 分隔。

你可以使用 `--config <你的 YAML 格式配置文件>` 将配置文件传递给 `kubeadm join`：

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
patches:
  directory: /home/user/somedir
```

补丁目录必须包含名为 `target[suffix][+patchtype].extension` 的文件。 例如， `kube-apiserver0+merge.yaml` 或只是 `etcd.json`。

- `target` 可以是 `kube-apiserver`、 `kube-controller-manager`、 `kube-scheduler` 和 `etcd` 之一。
- `patchtype` 可以是 `strategy`、 `merge` 或 `json` 之一，并且这些必须匹配 kubectl 支持 的补丁格式。 默认补丁类型是 `strategic` 的。
- `extension` 必须是 `json` 或 `yaml`。
- `suffix` 是一个可选字符串，可用于确定首先按字母数字应用哪些补丁。

> **说明：**
> 如果你使用 `kubeadm upgrade` 升级 kubeadm 节点，你必须再次提供相同的补丁，以便在升级后保留自定义配置。 为此，你可以使用 `--patches` 参数，该参数必须指向同一目录。 `kubeadm upgrade` 目前不支持用于相同目的的 API 结构配置。

自定义 kubelet

要自定义 kubelet，你可以在同一配置文件中的 `ClusterConfiguration` 或 `InitConfiguration` 之外添加一个 `KubeletConfiguration`，用 `---` 分隔。 然后可以将此文件传递给 `kubeadm init`。

> **说明：**
> kubeadm 将相同的 `KubeletConfiguration` 配置应用于集群中的所有节点。 要应用节点特定设置，你可以使用 `kubelet` 参数进行覆盖，方法是将它们传递到 `InitConfiguration` 和 `JoinConfiguration` 支持的 `nodeRegistration.kubeletExtraArgs` 字段中。一些 kubelet 参数已被弃用， 因此在使用这些参数之前，请在 kubelet 参考文档 中检查它们的状态。

更多详情，请参阅使用 kubeadm 配置集群中的每个 kubelet

自定义 kube-proxy

要自定义 kube-proxy，你可以在 `ClusterConfiguration` 或 `InitConfiguration` 之外添加一个由 `---` 分隔的 `KubeProxyConfiguration`， 传递给 `kubeadm init`。

可以导航到 API 参考页面 查看更多详情，

> **说明：**
> kubeadm 将 kube-proxy 部署为 DaemonSet， 这意味着 `KubeProxyConfiguration` 将应用于集群中的所有 kube-proxy 实例。

3.1.5 - 高可用拓扑选项

本页面介绍了配置高可用（HA） Kubernetes 集群拓扑的两个选项。

您可以设置 HA 集群：

- 使用堆叠（stacked）控制平面节点，其中 etcd 节点与控制平面节点共存
- 使用外部 etcd 节点，其中 etcd 在与控制平面不同的节点上运行

在设置 HA 集群之前，您应该仔细考虑每种拓扑的优缺点。

> **说明：** kubeadm 静态引导 etcd 集群。 阅读 etcd 集群指南以获得更多详细信息。

堆叠（Stacked） etcd 拓扑

堆叠（Stacked） HA 集群是一种这样的拓扑，其中 etcd 分布式数据存储集群堆叠在 kubeadm 管理的控制平面节点上，作为控制平面的一个组件运行。

每个控制平面节点运行 `kube-apiserver`， `kube-scheduler` 和 `kube-controller-manager` 实例。 `kube-apiserver` 使用负载均衡器暴露给工作节点。

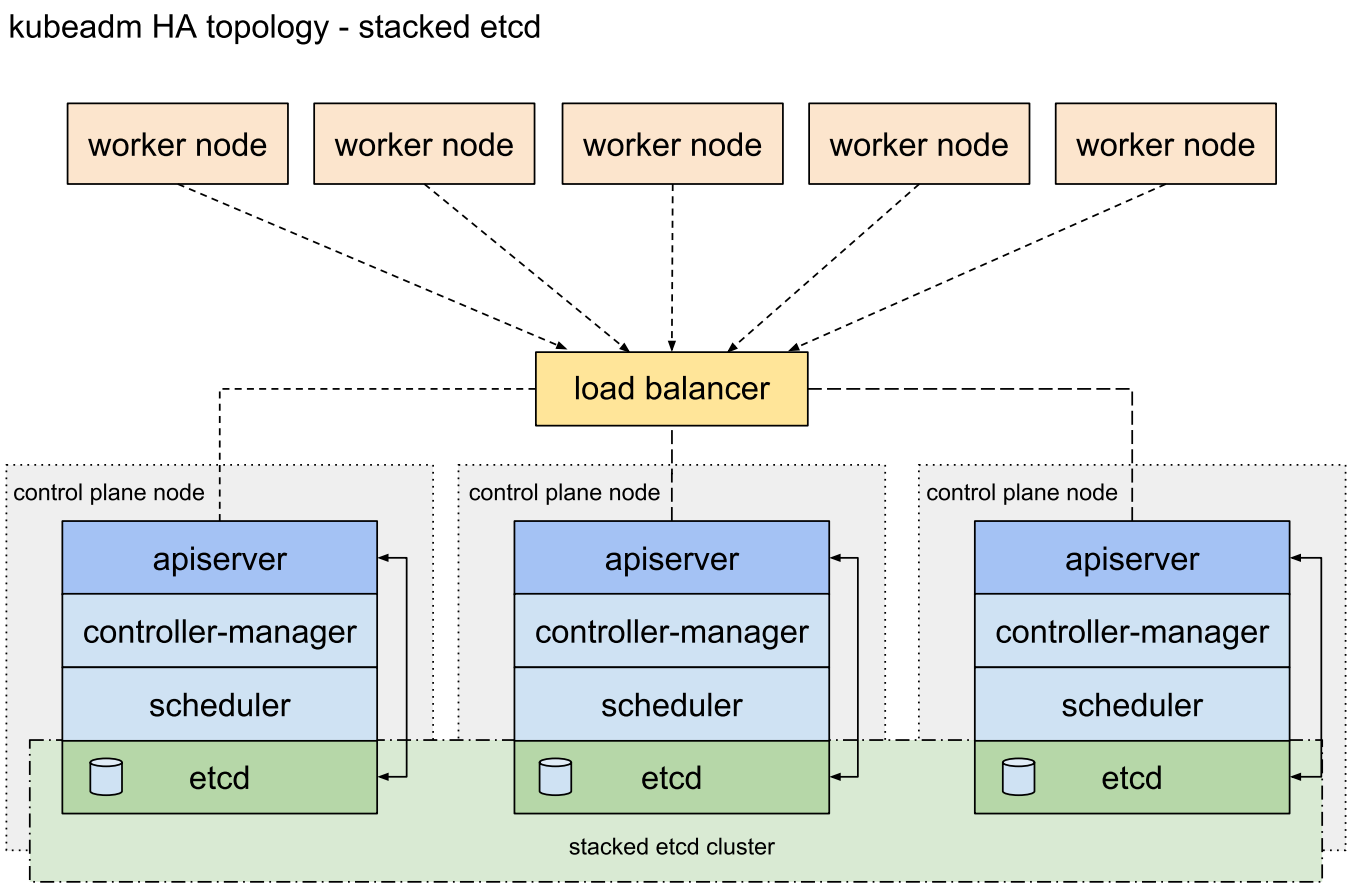
每个控制平面节点创建一个本地 etcd 成员（member），这个 etcd 成员只与该节点的 `kube-apiserver` 通信。这同样适用于本地 `kube-controller-manager` 和 `kube-scheduler` 实例。

这种拓扑将控制平面和 etcd 成员耦合在同一节点上。相对使用外部 etcd 集群，设置起来更简单，而且更易于副本管理。

然而，堆叠集群存在耦合失败的风险。如果一个节点发生故障，则 etcd 成员和控制平面实例都将丢失，并且冗余会受到影响。您可以通过添加更多控制平面节点来降低此风险。

因此，您应该为 HA 集群运行至少三个堆叠的控制平面节点。

这是 kubeadm 中的默认拓扑。当使用 `kubeadm init` 和 `kubeadm join --control-plane` 时，在控制平面节点上会自动创建本地 etcd 成员。



外部 etcd 拓扑

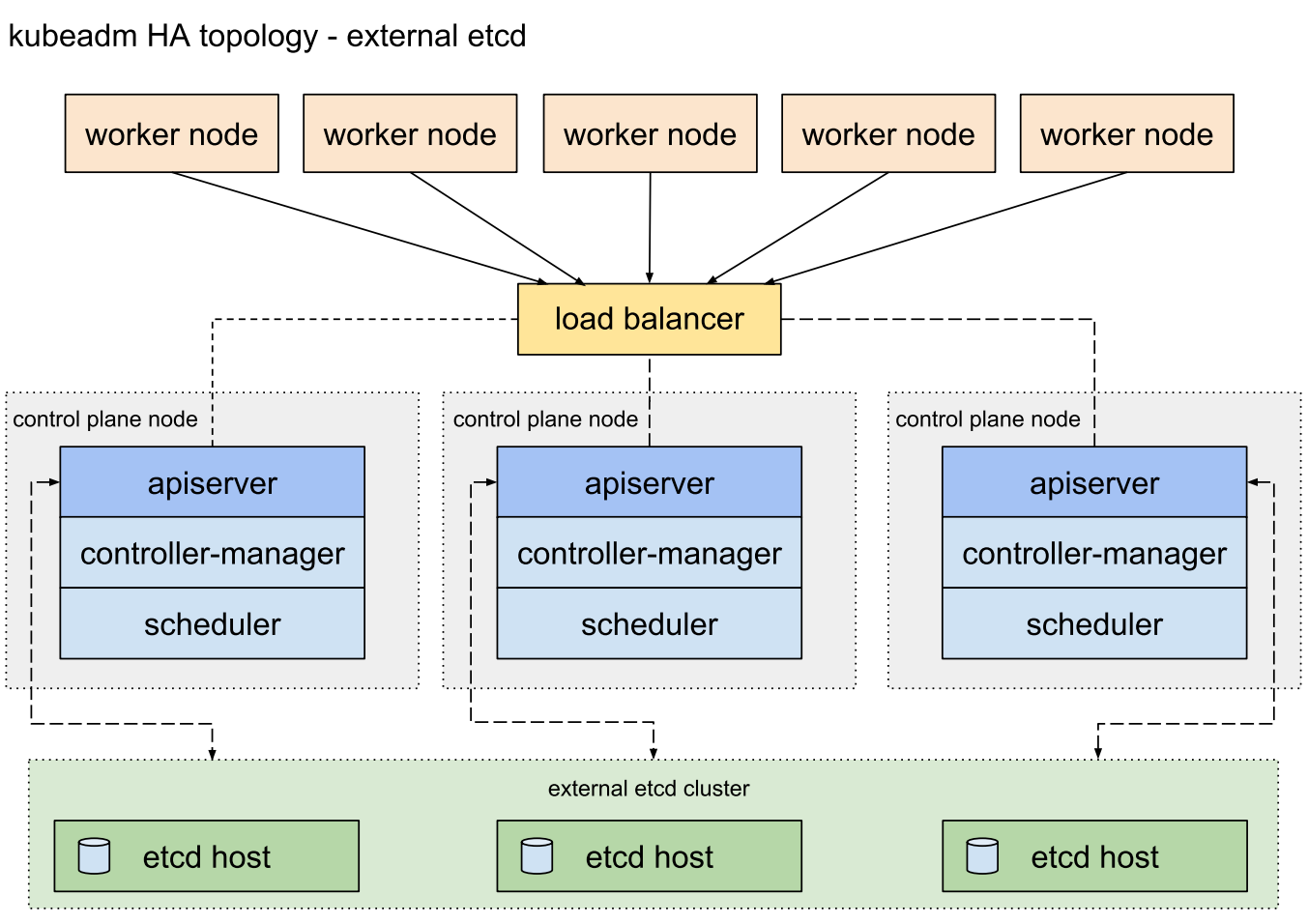
具有外部 etcd 的 HA 集群是一种这样的拓扑，其中 etcd 分布式数据存储集群在独立于控制平面节点的其他节点上运行。

就像堆叠的 etcd 拓扑一样，外部 etcd 拓扑中的每个控制平面节点都运行 `kube-apiserver`， `kube-scheduler` 和 `kube-controller-manager` 实例。同样， `kube-apiserver` 使用负载均衡器暴露给工作节点。但是，etcd 成员在不同的主机上运行，每个 etcd 主机与每个控制平面节点的 `kube-apiserver` 通信。

这种拓扑结构解耦了控制平面和 etcd 成员。因此，它提供了一种 HA 设置，其中失去控制平面实例或者 etcd 成员的影响较小，并且不会像堆叠的 HA 拓扑那样影响集群冗余。

但是，此拓扑需要两倍于堆叠 HA 拓扑的主机数量。

具有此拓扑的 HA 集群至少需要三个用于控制平面节点的主机和三个用于 etcd 节点的主机。



接下来

- 使用 kubeadm 设置高可用集群

3.1.6 - 利用 kubeadm 创建高可用集群

本文讲述了使用 kubeadm 设置一个高可用的 Kubernetes 集群的两种不同方式：

- 使用具有堆叠的控制平面节点。这种方法所需基础设施较少。etcd 成员和控制平面节点位于同一位置。
- 使用外部集群。这种方法所需基础设施较多。控制平面的节点和 etcd 成员是分开的。

在下一步之前，你应该仔细考虑哪种方法更好的满足你的应用程序和环境的需求。 高可用拓扑选项讲述了每种方法的优缺点。

如果你在安装 HA 集群时遇到问题，请在 kubeadm 问题跟踪里向我们提供反馈。

你也可以阅读升级文档

> **注意：** 这篇文档没有讲述在云提供商上运行集群的问题。在云环境中，此处记录的方法不适用于类型为 LoadBalancer 的服务对象，或者具有动态的 PersistentVolumes。

准备开始

根据集群控制平面所选择的拓扑结构不同，准备工作也有所差异：

堆叠（Stacked） etcd 拓扑 | 外部 etcd 拓扑

需要准备：

- 配置满足 kubeadm 的最低要求 的三台机器作为控制面节点。奇数台控制平面节点有利于机器故障或者网络分区时进行重新选主。
  - 机器已经安装好容器运行时，并正常运行
- 配置满足 kubeadm 的最低要求 的三台机器作为工作节点
  - 机器已经安装好容器运行时，并正常运行
- 在集群中，确保所有计算机之间存在全网络连接（公网或私网）
- 在所有机器上具有 sudo 权限
  - 可以使用其他工具；本教程以 `sudo` 举例
- 从某台设备通过 SSH 访问系统中所有节点的能力
- 所有机器上已经安装 `kubeadm` 和 `kubelet`

拓扑详情请参考堆叠（Stacked）etcd 拓扑。

容器镜像

每台主机需要能够从 Kubernetes 容器镜像仓库（ `k8s.gcr.io` ）读取和拉取镜像。 想要在无法拉取 Kubernetes 仓库镜像的机器上部署高可用集群也是可行的。通过其他的手段保证主机上已经有对应的容器镜像即可。

命令行

一旦集群创建成功，需要在 PC 上安装 kubectl 用于管理 Kubernetes。为了方便故障排查，也可以在每个控制平面节点上安装 `kubectl` 。

这两种方法的第一步

为 kube-apiserver 创建负载均衡器

> **说明：** 使用负载均衡器需要许多配置。你的集群搭建可能需要不同的配置。 下面的例子只是其中的一方面配置。

1. 创建一个名为 kube-apiserver 的负载均衡器解析 DNS。

   - 在云环境中，应该将控制平面节点放置在 TCP 转发负载平衡后面。 该负载均衡器将流量分配给目标列表中所有运行状况良好的控制平面节点。 API 服务器的健康检查是在 kube-apiserver 的监听端口（默认值 `:6443`） 上进行的一个 TCP 检查。
   - 不建议在云环境中直接使用 IP 地址。
   - 负载均衡器必须能够在 API 服务器端口上与所有控制平面节点通信。 它还必须允许其监听端口的入站流量。
   - 确保负载均衡器的地址始终匹配 kubeadm 的 `ControlPlaneEndpoint` 地址。
   - 阅读软件负载平衡选项指南 以获取更多详细信息。

2. 添加第一个控制平面节点到负载均衡器并测试连接：

```
nc -v LOAD_BALANCER_IP PORT
```

   由于 apiserver 尚未运行，预期会出现一个连接拒绝错误。 然而超时意味着负载均衡器不能和控制平面节点通信。 如果发生超时，请重新配置负载均衡器与控制平面节点进行通信。

3. 将其余控制平面节点添加到负载均衡器目标组。

使用堆控制平面和 etcd 节点

控制平面节点的第一步

1. 初始化控制平面：

```
sudo kubeadm init --control-plane-endpoint "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" -
```

   - 你可以使用 `--kubernetes-version` 标志来设置要使用的 Kubernetes 版本。 建议将 kubeadm、kebelet、kubectl 和 Kubernetes 的版本匹配。
   - 这个 `--control-plane-endpoint` 标志应该被设置成负载均衡器的地址或 DNS 和端口。
   - 这个 `--upload-certs` 标志用来将在所有控制平面实例之间的共享证书上传到集群。 如果正好相反，你更喜欢手动地通过控制平面节点或者使用自动化工具复制证书， 请删除此标志并参考如下部分证书分配手册。

> **说明：** 标志 `kubeadm init`、`--config` 和 `--certificate-key` 不能混合使用， 因此如果你要使用 kubeadm 配置，你必须在相应的配置结构 （位于 `InitConfiguration` 和 `JoinConfiguration: controlPlane`）添加 `certificateKey` 字段。

> **说明：** 一些 CNI 网络插件如 Calico 需要 CIDR 例如 `192.168.0.0/16` 和一些像 Weave 没有。参考 CNI 网络文档。 通过传递 `--pod-network-cidr` 标志添加 pod CIDR，或者你可以使用 kubeadm 配置文件，在 `ClusterConfiguration` 的 `networking` 对象下设置 `podSubnet` 字段。

   - 输出类似于：

```
...
You can now join any number of control-plane node by running the following
kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discover

Please note that the certificate-key gives access to cluster sensitive dat
```

```
As a safeguard, uploaded-certs will be deleted in two hours; If necessary,

Then you can join any number of worker nodes by running the following on e
  kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discov
```

- 将此输出复制到文本文件。 稍后你将需要它来将控制平面节点和工作节点加入集群。
- 当使用 `--upload-certs` 调用 `kubeadm init` 时，主控制平面的证书被加密并上传到 `kubeadm-certs` Secret 中。
- 要重新上传证书并生成新的解密密钥，请在已加入集群节点的控制平面上使用以下命令：

```
sudo kubeadm init phase upload-certs --upload-certs
```

- 你还可以在 `init` 期间指定自定义的 `--certificate-key`，以后可以由 `join` 使用。 要生成这样的密钥，可以使用以下命令：

```
kubeadm certs certificate-key
```

> **说明：** `kubeadm-certs` Secret 和解密密钥会在两个小时后失效。

> **注意：** 正如命令输出中所述，证书密钥可访问群集敏感数据。请妥善保管！

2. 应用你所选择的 CNI 插件： 请遵循以下指示 安装 CNI 驱动。如果适用，请确保配置与 kubeadm 配置文件中指定的 Pod CIDR 相对应。

> **说明：** 在进行下一步之前，必须选择并部署合适的网络插件。 否则集群不会正常运行。

3. 输入以下内容，并查看控制平面组件的 Pods 启动：

```
kubectl get pod -n kube-system -w
```

其余控制平面节点的步骤

> **说明：** 从 kubeadm 1.15 版本开始，你可以并行加入多个控制平面节点。 在此版本之前，你必须在第一个节点初始化后才能依序的增加新的控制平面节点。

对于每个其他控制平面节点，你应该：

1. 执行先前由第一个节点上的 `kubeadm init` 输出提供给你的 join 命令。 它看起来应该像这样：

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-to
```

- 这个 `--control-plane` 标志通知 `kubeadm join` 创建一个新的控制平面。

- `--certificate-key ...` 将导致从集群中的 `kubeadm-certs` Secret 下载控制平面证书并使用给定的密钥进行解密。

外部 etcd 节点

使用外部 etcd 节点设置集群类似于用于堆叠 etcd 的过程， 不同之处在于你应该首先设置 etcd，并在 kubeadm 配置文件中传递 etcd 信息。

设置 ectd 集群

1. 按照这些指示 去设置 etcd 集群。
2. 根据这里 的描述配置 SSH。
3. 将以下文件从集群中的任何 etcd 节点复制到第一个控制平面节点：

```
export CONTROL_PLANE="ubuntu@10.0.0.7"
scp /etc/kubernetes/pki/etcd/ca.crt "${CONTROL_PLANE}":
scp /etc/kubernetes/pki/apiserver-etcd-client.crt "${CONTROL_PLANE}":
scp /etc/kubernetes/pki/apiserver-etcd-client.key "${CONTROL_PLANE}":
```

- 用第一台控制平面机的 `user@host` 替换 `CONTROL_PLANE` 的值。

设置第一个控制平面节点

1. 用以下内容创建一个名为 `kubeadm-config.yaml` 的文件：

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
kubernetesVersion: stable
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" # change this (see bel
etcd:
  external:
    endpoints:
      - https://ETCD_0_IP:2379 # change ETCD_0_IP appropriately
      - https://ETCD_1_IP:2379 # change ETCD_1_IP appropriately
      - https://ETCD_2_IP:2379 # change ETCD_2_IP appropriately
    caFile: /etc/kubernetes/pki/etcd/ca.crt
    certFile: /etc/kubernetes/pki/apiserver-etcd-client.crt
    keyFile: /etc/kubernetes/pki/apiserver-etcd-client.key
```

> **说明：** 这里的堆叠（stacked）etcd 和外部 etcd 之前的区别在于设置外部 etcd 需要一个 `etcd` 的 `external` 对象下带有 etcd 端点的配置文件。 如果是内部 etcd，是自动管理的。

- 在你的集群中，将配置模板中的以下变量替换为适当值：
  - `LOAD_BALANCER_DNS`
  - `LOAD_BALANCER_PORT`
  - `ETCD_0_IP`
  - `ETCD_1_IP`
  - `ETCD_2_IP`

以下的步骤与设置内置 etcd 的集群是相似的：

1. 在节点上运行 `sudo kubeadm init --config kubeadm-config.yaml --upload-certs` 命令。
2. 记下输出的 join 命令，这些命令将在以后使用。

3. 应用你选择的 CNI 插件。

> **说明：** 在进行下一步之前，必须选择并部署合适的网络插件。 否则集群不会正常运行。

其他控制平面节点的步骤

步骤与设置内置 etcd 相同：

- 确保第一个控制平面节点已完全初始化。
- 使用保存到文本文件的 join 命令将每个控制平面节点连接在一起。 建议一次加入一个控制平面节点。
- 不要忘记默认情况下， `--certificate-key` 中的解密秘钥会在两个小时后过期。

列举控制平面之后的常见任务

安装工作节点

你可以使用之前存储的 `kubeadm init` 命令的输出将工作节点加入集群中：

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-ca
```

手动证书分发

如果你选择不将 `kubeadm init` 与 `--upload-certs` 命令一起使用， 则意味着你将必须手动将证书从主控制平面节点复制到 将要加入的控制平面节点上。

有许多方法可以实现这种操作。在下面的例子中我们使用 `ssh` 和 `scp`：

如果要在单独的一台计算机控制所有节点，则需要 SSH。

1. 在你的主设备上启用 ssh-agent，要求该设备能访问系统中的所有其他节点：

   ```
   eval $(ssh-agent)
   ```

2. 将 SSH 身份添加到会话中：

   ```
   ssh-add ~/.ssh/path_to_private_key
   ```

3. 检查节点间的 SSH 以确保连接是正常运行的

   - SSH 到任何节点时，请确保添加 `-A` 标志：

     ```
     ssh -A 10.0.0.7
     ```

   - 当在任何节点上使用 sudo 时，请确保保持环境变量设置，以便 SSH 转发能够正常工作：

     ```
     sudo -E -s
     ```

4. 在所有节点上配置 SSH 之后，你应该在运行过 `kubeadm init` 命令的第一个 控制平面节点上运行以下脚本。 该脚本会将证书从第一个控制平面节点复制到另一个控制平面节点：

   在以下示例中，用其他控制平面节点的 IP 地址替换 `CONTROL_PLANE_IPS` 。

```
USER=ubuntu # 可定制
CONTROL_PLANE_IPS="10.0.0.7 10.0.0.8"
for host in ${CONTROL_PLANE_IPS}; do
    scp /etc/kubernetes/pki/ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.key "${USER}"@$host:
    scp /etc/kubernetes/pki/sa.pub "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.crt "${USER}"@$host:
    scp /etc/kubernetes/pki/front-proxy-ca.key "${USER}"@$host:
    scp /etc/kubernetes/pki/etcd/ca.crt "${USER}"@$host:etcd-ca.crt
    scp /etc/kubernetes/pki/etcd/ca.key "${USER}"@$host:etcd-ca.key
done
```

   注意： 只需要复制上面列表中的证书。kubeadm 将负责生成其余证书以及加入控制平面实例所需的 SAN。 如果你错误地复制了所有证书，由于缺少所需的 SAN，创建其他节点可能会失败。

5. 然后，在每个即将加入集群的控制平面节点上，你必须先运行以下脚本，然后 再运行 `kubeadm join` 。 该脚本会将先前复制的证书从主目录移动到 `/etc/kubernetes/pki` ：

```
USER=ubuntu # 可定制
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
```

3.1.7 - 使用 kubeadm 创建一个高可用 etcd 集群

> **说明：**
> 在本指南中，当 kubeadm 用作为外部 etcd 节点管理工具，请注意 kubeadm 不计划支持此类节点的证书更换或升级。对于长期规划是使用 etcdadm 增强工具来管理这方面。

默认情况下，kubeadm 运行单成员的 etcd 集群，该集群由控制面节点上的 kubelet 以静态 Pod 的方式进行管理。由于 etcd 集群只包含一个成员且不能在任一成员不可用时保持运行，所以这不是一种高可用设置。本任务，将告诉你如何在使用 kubeadm 创建一个 kubernetes 集群时创建一个外部 etcd：有三个成员的高可用 etcd 集群。

准备开始

- 三个可以通过 2379 和 2380 端口相互通信的主机。本文档使用这些作为默认端口。不过，它们可以通过 kubeadm 的配置文件进行自定义。
- 每个主机必须 安装有 docker、kubelet 和 kubeadm。
- 一些可以用来在主机间复制文件的基础设施。例如 `ssh` 和 `scp` 就可以满足需求。

建立集群

一般来说，是在一个节点上生成所有证书并且只分发这些*必要*的文件到其它节点上。

> **说明：**
> kubeadm 包含生成下述证书所需的所有必要的密码学工具；在这个例子中，不需要其他加密工具。

1. 将 kubelet 配置为 etcd 的服务管理器。

   > **说明：** 你必须在要运行 etcd 的所有主机上执行此操作。

   由于 etcd 是首先创建的，因此你必须通过创建具有更高优先级的新文件来覆盖 kubeadm 提供的 kubelet 单元文件。

```
cat << EOF > /etc/systemd/system/kubelet.service.d/20-etcd-service-manager.conf
[Service]
ExecStart=
# 将下面的 "systemd" 替换为你的容器运行时所使用的 cgroup 驱动。
# kubelet 的默认值为 "cgroupfs"。
ExecStart=/usr/bin/kubelet --address=127.0.0.1 --pod-manifest-path=/etc/kubernetes/
Restart=always
EOF

systemctl daemon-reload
systemctl restart kubelet
```

检查 kubelet 的状态以确保其处于运行状态：

```
systemctl status kubelet
```

2. 为 kubeadm 创建配置文件。

使用以下脚本为每个将要运行 etcd 成员的主机生成一个 kubeadm 配置文件。

```
# 使用 IP 或可解析的主机名替换 HOST0、HOST1 和 HOST2
export HOST0=10.0.0.6
export HOST1=10.0.0.7
export HOST2=10.0.0.8

# 创建临时目录来存储将被分发到其它主机上的文件
mkdir -p /tmp/${HOST0}/ /tmp/${HOST1}/ /tmp/${HOST2}/

ETCDHOSTS=(${HOST0} ${HOST1} ${HOST2})
NAMES=("infra0" "infra1" "infra2")

for i in "${!ETCDHOSTS[@]}"; do
HOST=${ETCDHOSTS[$i]}
NAME=${NAMES[$i]}
cat << EOF > /tmp/${HOST}/kubeadmcfg.yaml
apiVersion: "kubeadm.k8s.io/v1beta3"
kind: ClusterConfiguration
etcd:
    local:
        serverCertSANs:
        - "${HOST}"
        peerCertSANs:
        - "${HOST}"
        extraArgs:
            initial-cluster: infra0=https://${ETCDHOSTS[0]}:2380,infra1=https://${E
            initial-cluster-state: new
            name: ${NAME}
            listen-peer-urls: https://${HOST}:2380
            listen-client-urls: https://${HOST}:2379
            advertise-client-urls: https://${HOST}:2379
            initial-advertise-peer-urls: https://${HOST}:2380
EOF
done
```

3. 生成证书颁发机构

如果你已经拥有 CA，那么唯一的操作是复制 CA 的 `crt` 和 `key` 文件到 `etc/kubernetes/pki/etcd/ca.crt` 和 `/etc/kubernetes/pki/etcd/ca.key`。 复制完这些文件后继续下一步，“为每个成员创建证书”。

如果你还没有 CA，则在 `$HOST0` （你为 kubeadm 生成配置文件的位置）上运行此命令。

```
kubeadm init phase certs etcd-ca
```

这一操作创建如下两个文件

- `/etc/kubernetes/pki/etcd/ca.crt`
- `/etc/kubernetes/pki/etcd/ca.key`

4. 为每个成员创建证书

```
kubeadm init phase certs etcd-server --config=/tmp/${HOST2}/kubeadmcfg.yaml
kubeadm init phase certs etcd-peer --config=/tmp/${HOST2}/kubeadmcfg.yaml
kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST2}/kubeadm
kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST2}/kubeadmcf
cp -R /etc/kubernetes/pki /tmp/${HOST2}/
# 清理不可重复使用的证书
find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete
```

```
kubeadm init phase certs etcd-server --config=/tmp/${HOST1}/kubeadmcfg.yaml
kubeadm init phase certs etcd-peer --config=/tmp/${HOST1}/kubeadmcfg.yaml
kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST1}/kubeadmc
kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST1}/kubeadmcfg
cp -R /etc/kubernetes/pki /tmp/${HOST1}/
find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete

kubeadm init phase certs etcd-server --config=/tmp/${HOST0}/kubeadmcfg.yaml
kubeadm init phase certs etcd-peer --config=/tmp/${HOST0}/kubeadmcfg.yaml
kubeadm init phase certs etcd-healthcheck-client --config=/tmp/${HOST0}/kubeadmc
kubeadm init phase certs apiserver-etcd-client --config=/tmp/${HOST0}/kubeadmcfg
# 不需要移动 certs 因为它们是给 HOST0 使用的

# 清理不应从此主机复制的证书
find /tmp/${HOST2} -name ca.key -type f -delete
find /tmp/${HOST1} -name ca.key -type f -delete
```

5. 复制证书和 kubeadm 配置

   证书已生成，现在必须将它们移动到对应的主机。

```
USER=ubuntu
HOST=${HOST1}
scp -r /tmp/${HOST}/* ${USER}@${HOST}:
ssh ${USER}@${HOST}
USER@HOST $ sudo -Es
root@HOST $ chown -R root:root pki
root@HOST $ mv pki /etc/kubernetes/
```

6. 确保已经所有预期的文件都存在

   `$HOST0` 所需文件的完整列表如下：

```
/tmp/${HOST0}
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── ca.key
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

在 `$HOST1` 上：

```
$HOME
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

在 `$HOST2` 上：

```
$HOME
└── kubeadmcfg.yaml
---
/etc/kubernetes/pki
├── apiserver-etcd-client.crt
├── apiserver-etcd-client.key
└── etcd
    ├── ca.crt
    ├── healthcheck-client.crt
    ├── healthcheck-client.key
    ├── peer.crt
    ├── peer.key
    ├── server.crt
    └── server.key
```

7. 创建静态 Pod 清单

   既然证书和配置已经就绪，是时候去创建清单了。 在每台主机上运行 `kubeadm` 命令来生成 etcd 使用的静态清单。

```
root@HOST0 $ kubeadm init phase etcd local --config=/tmp/${HOST0}/kubeadmcfg.yaml
root@HOST1 $ kubeadm init phase etcd local --config=/tmp/${HOST1}/kubeadmcfg.yaml
root@HOST2 $ kubeadm init phase etcd local --config=/tmp/${HOST2}/kubeadmcfg.yaml
```

8. 可选：检查群集运行状况

```
docker run --rm -it \
--net host \
-v /etc/kubernetes:/etc/kubernetes k8s.gcr.io/etcd:${ETCD_TAG} etcdctl \
--cert /etc/kubernetes/pki/etcd/peer.crt \
--key /etc/kubernetes/pki/etcd/peer.key \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--endpoints https://${HOST0}:2379 endpoint health --cluster
...
https://[HOST0 IP]:2379 is healthy: successfully committed proposal: took = 16.2833
https://[HOST1 IP]:2379 is healthy: successfully committed proposal: took = 19.4440
https://[HOST2 IP]:2379 is healthy: successfully committed proposal: took = 35.9264
```

   - 将 `${ETCD_TAG}` 设置为你的 etcd 镜像的版本标签，例如 `3.4.3-0` 。 要查看 kubeadm 使用的 etcd 镜像和标签，请执行 `kubeadm config images list --kubernetes-version ${K8S_VERSION}` ， 例如，其中的 `${K8S_VERSION}` 可以是 `v1.17.0` 。
   - 将 `${HOST0}` 设置为要测试的主机的 IP 地址。

接下来

一旦拥有了一个正常工作的 3 成员的 etcd 集群，你就可以基于 使用 kubeadm 外部 etcd 的方法，继续部署一个高可用的控制平面。

3.1.8 - 使用 kubeadm 配置集群中的每个 kubelet

FEATURE STATE: `Kubernetes 1.11 [stable]`

kubeadm CLI 工具的生命周期与 kubelet 解耦；kubelet 是一个守护程序，在 Kubernetes 集群中的每个节点上运行。 当 Kubernetes 初始化或升级时，kubeadm CLI 工具由用户执行，而 kubelet 始终在后台运行。

由于kubelet是守护程序，因此需要通过某种初始化系统或服务管理器进行维护。 当使用 DEB 或 RPM 安装 kubelet 时，配置系统去管理 kubelet。 你可以改用其他服务管理器，但需要手动地配置。

集群中涉及的所有 kubelet 的一些配置细节都必须相同， 而其他配置方面则需要基于每个 kubelet 进行设置，以适应给定机器的不同特性（例如操作系统、存储和网络）。 你可以手动地管理 kubelet 的配置，但是 kubeadm 现在提供一种 `KubeletConfiguration` API 类型 用于集中管理 kubelet 的配置。

Kubelet 配置模式

以下各节讲述了通过使用 kubeadm 简化 kubelet 配置模式，而不是在每个节点上手动地管理 kubelet 配置。

将集群级配置传播到每个 kubelet 中

你可以通过使用 `kubeadm init` 和 `kubeadm join` 命令为 kubelet 提供默认值。 有趣的示例包括使用其他 CRI 运行时或通过服务器设置不同的默认子网。

如果你想使用子网 `10.96.0.0/12` 作为services的默认网段，你可以给 kubeadm 传递 `--service-cidr` 参数：

```
kubeadm init --service-cidr 10.96.0.0/12
```

现在，可以从该子网分配服务的虚拟 IP。 你还需要通过 kubelet 使用 `--cluster-dns` 标志设置 DNS 地址。 在集群中的每个管理器和节点上的 kubelet 的设置需要相同。 kubelet 提供了一个版本化的结构化 API 对象，该对象可以配置 kubelet 中的大多数参数，并将此配置推送到集群中正在运行的每个 kubelet 上。 此对象被称为 `KubeletConfiguration` 。 `KubeletConfiguration` 允许用户指定标志，例如用骆峰值代表集群的 DNS IP 地址，如下所示：

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
clusterDNS:
- 10.96.0.10
```

有关 `KubeletConfiguration` 的更多详细信息，亲参阅本节。

提供指定实例的详细配置信息

由于硬件、操作系统、网络或者其他主机特定参数的差异。某些主机需要特定的 kubelet 配置。 以下列表提供了一些示例。

- 由 kubelet 配置标志 `--resolv-conf` 指定的 DNS 解析文件的路径在操作系统之间可能有所不同， 它取决于你是否使用 `systemd-resolved` 。 如果此路径错误，则在其 kubelet 配置错误的节点上 DNS 解析也将失败。
- 除非你使用云驱动，否则默认情况下 Node API 对象的 `.metadata.name` 会被设置为计算机的主机名。 如果你需要指定一个与机器的主机名不同的节点名称，你可以使用 `--hostname-override` 标志覆盖默认值。

- 当前，kubelet 无法自动检测 CRI 运行时使用的 cgroup 驱动程序， 但是值 `--cgroup-driver` 必须与 CRI 运行时使用的 cgroup 驱动程序匹配，以确保 kubelet 的健康运行状况。
- 取决于你的集群所使用的 CRI 运行时，你可能需要为 kubelet 指定不同的标志。 例如，当使用 Docker 时，你需要指定如 `--network-plugin=cni` 这类标志；但是如果你使用的是外部运行时， 则需要指定 `--container-runtime=remote` 并使用 `--container-runtime-endpoint=<path>` 指定 CRI 端点。

你可以在服务管理器（例如 systemd）中设定某个 kubelet 的配置来指定这些参数。

使用 kubeadm 配置 kubelet

如果自定义的 `KubeletConfiguration` API 对象使用像 `kubeadm ... --config some-config-file.yaml` 这样的配置文件进行传递，则可以配置 kubeadm 启动的 kubelet。

通过调用 `kubeadm config print init-defaults --component-configs KubeletConfiguration` ， 你可以看到此结构中的所有默认值。

也可以阅读 KubeletConfiguration 参考 来获取有关各个字段的更多信息。

当使用 `kubeadm init`时的工作流程

当调用 `kubeadm init` 时，kubelet 配置被编组到磁盘上的 `/var/lib/kubelet/config.yaml` 中， 并且上传到集群中的 ConfigMap。 ConfigMap 名为 `kubelet-config-1.X` ，其中 `X` 是你正在初始化的 kubernetes 版本的次版本。 在集群中所有 kubelet 的基准集群范围内配置，将 kubelet 配置文件写入 `/etc/kubernetes/kubelet.conf` 中。 此配置文件指向允许 kubelet 与 API 服务器通信的客户端证书。 这解决了将集群级配置传播到每个 kubelet 的需求。

该文档 提供特定实例的配置详细信息 是第二种解决模式， kubeadm 将环境文件写入 `/var/lib/kubelet/kubeadm-flags.env` ，其中包含了一个标志列表， 当 kubelet 启动时，该标志列表会传递给 kubelet 标志在文件中的显示方式如下：

```
KUBELET_KUBEADM_ARGS="--flag1=value1 --flag2=value2 ..."
```

除了启动 kubelet 时使用该标志外，该文件还包含动态参数，例如 cgroup 驱动程序以及是否使用其他 CRI 运行时 socket （ `--cri-socket` ）。

将这两个文件编组到磁盘后，如果使用 systemd，则 kubeadm 尝试运行以下两个命令：

```
systemctl daemon-reload && systemctl restart kubelet
```

如果重新加载和重新启动成功，则正常的 `kubeadm init` 工作流程将继续。

当使用 `kubeadm join`时的工作流程

当运行 `kubeadm join` 时，kubeadm 使用 Bootstrap Token 证书执行 TLS 引导，该引导会获取一份证书， 该证书需要下载 `kubelet-config-1.X` ConfigMap 并把它写入 `/var/lib/kubelet/config.yaml` 中。 动态环境文件的生成方式恰好与 `kubeadm init` 完全相同。

接下来， `kubeadm` 运行以下两个命令将新配置加载到 kubelet 中：

```
systemctl daemon-reload && systemctl restart kubelet
```

在 kubelet 加载新配置后，kubeadm 将写入 `/etc/kubernetes/bootstrap-kubelet.conf` KubeConfig 文件中， 该文件包含 CA 证书和引导程序令牌。 kubelet 使用这些证书执行 TLS 引导程序并获取唯一的凭据，该凭据被存储在 `/etc/kubernetes/kubelet.conf` 中。

当 `/etc/kubernetes/kubelet.conf` 文件被写入后，kubelet 就完成了 TLS 引导过程。 Kubeadm 在完成 TLS 引导过程后将删除 `/etc/kubernetes/bootstrap-kubelet.conf` 文件。

kubelet 的 systemd 文件

`kubeadm` 中附带了有关系统如何运行 kubelet 的 systemd 配置文件。 请注意 kubeadm CLI 命令不会修改此文件。

通过 `kubeadm` DEB 或者 RPM 包 安装的配置文件被写入 `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` 并由系统使用。 它对原来的 RPM 版本 `kubelet.service` 或者 DEB 版本 `kubelet.service` 作了增强：

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kub
--kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# 这是 "kubeadm init" 和 "kubeadm join" 运行时生成的文件，动态地填充 KUBELET_KUBEADM_ARGS 变
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# 这是一个文件，用户在不得已下可以将其用作替代 kubelet args。
# 用户最好使用 .NodeRegistration.KubeletExtraArgs 对象在配置文件中替代。
# KUBELET_EXTRA_ARGS 应该从此文件中获取。
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM
```

该文件为 kubelet 指定由 kubeadm 管理的所有文件的默认位置。

- 用于 TLS 引导程序的 KubeConfig 文件为 `/etc/kubernetes/bootstrap-kubelet.conf`， 但仅当 `/etc/kubernetes/kubelet.conf` 不存在时才能使用。
- 具有唯一 kubelet 标识的 KubeConfig 文件为 `/etc/kubernetes/kubelet.conf` 。
- 包含 kubelet 的组件配置的文件为 `/var/lib/kubelet/config.yaml` 。
- 包含的动态环境的文件 `KUBELET_KUBEADM_ARGS` 是来源于 `/var/lib/kubelet/kubeadm-flags.env` 。
- 包含用户指定标志替代的文件 `KUBELET_EXTRA_ARGS` 是来源于 `/etc/default/kubelet` （对于 DEB），或者 `/etc/sysconfig/kubelet` （对于 RPM）。 `KUBELET_EXTRA_ARGS` 在标志链中排在最后，并且在设置冲突时具有最高优先级。

Kubernetes 可执行文件和软件包内容

Kubernetes 版本对应的 DEB 和 RPM 软件包是：

Package name	Description
`kubeadm`	给 kubelet 安装 `/usr/bin/kubeadm` CLI 工具和 kubelet 的 systemd 文件。
`kubelet`	安装 kubelet 可执行文件到 `/usr/bin` 路径，安装 CNI 可执行文件到 `/opt/cni/bin` 路径。
`kubectl`	安装 `/usr/bin/kubectl` 可执行文件。
`cri-tools`	从 cri-tools git 仓库中安装 `/usr/bin/crictl` 可执行文件。

3.1.9 - 使用 kubeadm 支持双协议栈

FEATURE STATE: `Kubernetes v1.23 [stable]`

你的集群包含双协议栈组网支持， 这意味着集群网络允许你在两种地址族间任选其一。在集群中，控制面可以为同一个 Pod 或者 Service 同时赋予 IPv4 和 IPv6 地址。

准备开始

你需要已经遵从安装 kubeadm 中所给的步骤安装了 kubeadm 工具。

针对你要作为节点使用的每台服务器， 确保其允许 IPv6 转发。在 Linux 节点上，你可以通过以 root 用户在每台服务器上运行 `sysctl -w net.ipv6.conf.all.forwarding=1` 来完成设置。

你需要一个可以使用的 IPv4 和 IPv6 地址范围。集群操作人员通常为 IPv4 使用 私有地址范围。对于 IPv6，集群操作人员通常会基于分配给该操作人员的地址范围， 从 `2000::/3` 中选择一个全局的单播地址块。你不需要将集群的 IP 地址范围路由 到公众互联网。

> **说明:**
> 如果你在使用 `kubeadm upgrade` 命令升级现有的集群， `kubeadm` 不允许更改 Pod 的 IP 地址范围（“集群 CIDR”），也不允许更改集群的服务地址范围（“Service CIDR”）。

创建双协议栈集群

要使用 `kubeadm init` 创建一个双协议栈集群，你可以传递与下面的例子类似的命令行参数：

```
# 这里的地址范围仅作示例使用
kubeadm init --pod-network-cidr=10.244.0.0/16,2001:db8:42:0::/56 --service-cidr=10.96.0.(
```

为了更便于理解，参看下面的名为 `kubeadm-config.yaml` 的 kubeadm 配置文件， 该文件用于双协议栈控制面的主控制节点。

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
networking:
  podSubnet: 10.244.0.0/16,2001:db8:42:0::/56
  serviceSubnet: 10.96.0.0/16,2001:db8:42:1::/112
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: "10.100.0.1"
  bindPort: 6443
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.2,fd00:1:2:3::2
```

InitConfiguration 中的 `advertiseAddress` 给出 API 服务器将公告自身要监听的 IP 地址。 `advertiseAddress` 的取值与 `kubeadm init` 的标志 `--apiserver-advertise-address` 的取值相同。

运行 kubeadm 来实例化双协议栈控制面节点：

```
kubeadm init --config=kubeadm-config.yaml
```

kube-controller-manager 标志 `--node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6` 是使用默认值来设置的。参见配置 IPv4/IPv6 双协议栈。

> **说明：**
> 标志 `--apiserver-advertise-address` 不支持双协议栈。

向双协议栈集群添加节点

在添加节点之前，请确保该节点具有 IPv6 可路由的网络接口并且启用了 IPv6 转发。

下面的名为 `kubeadm-config.yaml` 的 kubeadm 配置文件 示例用于向集群中添加工作节点。

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a9881cf0d9fe0e"
    # 请更改上面的认证信息，使之与你的集群中实际使用的令牌和 CA 证书匹配
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.3,fd00:1:2:3::3
```

下面的名为 `kubeadm-config.yaml` 的 kubeadm 配置文件 示例用于向集群中添加另一个控制面节点。

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
controlPlane:
  localAPIEndpoint:
    advertiseAddress: "10.100.0.2"
    bindPort: 6443
discovery:
  bootstrapToken:
    apiServerEndpoint: 10.100.0.1:6443
    token: "clvldh.vjjwg16ucnhp94qr"
    caCertHashes:
    - "sha256:a4863cde706cfc580a439f842cc65d5ef112b7b2be31628513a9881cf0d9fe0e"
    # 请更改上面的认证信息，使之与你的集群中实际使用的令牌和 CA 证书匹配
nodeRegistration:
  kubeletExtraArgs:
    node-ip: 10.100.0.4,fd00:1:2:3::4
```

JoinConfiguration.controlPlane 中的 `advertiseAddress` 设定 API 服务器将公告自身要监听的 IP 地址。 `advertiseAddress` 的取值与 `kubeadm join` 的标志 `--apiserver-advertise-address` 的取值相同。

```
kubeadm join --config=kubeadm-config.yaml
```

创建单协议栈集群

> **说明：**
> 双协议栈支持并不意味着你需要使用双协议栈来寻址。 你可以部署一个启用了双协议栈联网特性的单协议栈集群。

为了更便于理解，参看下面的名为 `kubeadm-config.yaml` 的 kubeadm 配置文件示例， 该文件用于单协议栈控制面节点。

```yaml
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
networking:
  podSubnet: 10.244.0.0/16
  serviceSubnet: 10.96.0.0/16
```

接下来

- 验证 IPv4/IPv6 双协议栈联网
- 阅读双协议栈集群网络
- 进一步了解 kubeadm 配置格式

3.2 - 使用 Kops 安装 Kubernetes

本篇快速入门介绍了如何在 AWS 上轻松安装 Kubernetes 集群。 本篇使用了一个名为 kops 的工具。

kops 是一个自动化的制备系统：

- 全自动安装流程
- 使用 DNS 识别集群
- 自我修复：一切都在自动扩缩组中运行
- 支持多种操作系统（如 Debian、Ubuntu 16.04、CentOS、RHEL、Amazon Linux 和 CoreOS） - 参考 images.md
- 支持高可用 - 参考 high_availability.md
- 可以直接提供或者生成 terraform 清单 - 参考 terraform.md

准备开始

- 你必须安装 kubectl。
- 你必须安装安装 kops 到 64 位的（AMD64 和 Intel 64）设备架构上。
- 你必须拥有一个 AWS 账户， 生成 IAM 秘钥 并配置 该秘钥。IAM 用户需要足够的权限许可。

创建集群

(1/5) 安装 kops

安装

从下载页面下载 kops （从源代码构建也很方便）：

macOS Linux

使用下面的命令下载最新发布版本：

```
curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.
```

要下载特定版本，使用特定的 kops 版本替换下面命令中的部分：

```
$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_na
```

例如，要下载 kops v1.20.0，输入：

```
curl -LO https://github.com/kubernetes/kops/releases/download/v1.20.0/kops-darwin-am
```

令 kops 二进制文件可执行：

```
chmod +x kops-darwin-amd64
```

将 kops 二进制文件移到你的 PATH 下：

```
sudo mv kops-darwin-amd64 /usr/local/bin/kops
```

你也可以使用 Homebrew 安装 kops：

```
brew update && brew install kops
```

(2/5) 为你的集群创建一个 route53 域名

kops 在集群内部和外部都使用 DNS 进行发现操作，这样你可以从客户端访问 kubernetes API 服务器。

kops 对集群名称有明显的要求：它应该是有效的 DNS 名称。这样一来，你就不会再使集群混乱，可以与同事明确共享集群，并且无需依赖记住 IP 地址即可访问群集。

你可以，或许应该使用子域名来划分集群。作为示例，我们将使用域名 `useast1.dev.example.com` 。这样，API 服务器端点域名将为 `api.useast1.dev.example.com` 。

Route53 托管区域可以服务子域名。你的托管区域可能是 `useast1.dev.example.com` ，还有 `dev.example.com` 甚至 `example.com` 。 kops 可以与以上任何一种配合使用，因此通常你出于组织原因选择不同的托管区域。 例如，允许你在 `dev.example.com` 下创建记录，但不能在 `example.com` 下创建记录。

假设你使用 `dev.example.com` 作为托管区域。你可以使用 正常流程 或者使用诸如 `aws route53 create-hosted-zone --name dev.example.com --caller-reference 1` 之类的命令来创建该托管区域。

然后，你必须在父域名中设置你的 DNS 记录，以便该域名中的记录可以被解析。 在这里，你将在 `example.com` 中为 `dev` 创建 DNS 记录。 如果它是根域名，则可以在域名注册机构配置 DNS 记录。 例如，你需要在购买 `example.com` 的地方配置 `example.com` 。

检查你的 route53 域已经被正确设置（这是导致问题的最常见原因！）。 如果你安装了 dig 工具，则可以通过运行以下步骤再次检查集群是否配置正确：

```
dig DNS dev.example.com
```

你应该看到 Route53 分配了你的托管区域的 4 条 DNS 记录。

(3/5) 创建一个 S3 存储桶来存储集群状态

kops 使你即使在安装后也可以管理集群。为此，它必须跟踪已创建的集群及其配置、所使用的密钥等。 此信息存储在 S3 存储桶中。S3 权限用于控制对存储桶的访问。

多个集群可以使用同一 S3 存储桶，并且你可以在管理同一集群的同事之间共享一个 S3 存储桶 - 这比传递 kubecfg 文件容易得多。 但是有权访问 S3 存储桶的任何人都将拥有对所有集群的管理访问权限， 因此你不想在运营团队之外共享它。

因此，通常每个运维团队都有一个 S3 存储桶（而且名称通常对应于上面托管区域的名称！）

在我们的示例中，我们选择 `dev.example.com` 作为托管区域，因此我们选择 `clusters.dev.example.com` 作为 S3 存储桶名称。

- 导出 `AWS_PROFILE` 文件（如果你需要选择一个配置文件用来使 AWS CLI 正常工作）
- 使用 `aws s3 mb s3://clusters.dev.example.com` 创建 S3 存储桶
- 你可以进行 `export KOPS_STATE_STORE=s3://clusters.dev.example.com` 操作， 然后 kops 将默认使用此位置。 我们建议将其放入你的 bash profile 文件或类似文件中。

(4/5) 建立你的集群配置

运行 `kops create cluster` 以创建你的集群配置：

```
kops create cluster --zones=us-east-1c useast1.dev.example.com
```

kops 将为你的集群创建配置。请注意，它_仅_创建配置，实际上并没有创建云资源 - 你将在下一步中使用 `kops update cluster` 进行配置。 这使你有机会查看配置或进行更改。

它打印出可用于进一步探索的命令：

- 使用以下命令列出集群： `kops get cluster`
- 使用以下命令编辑该集群： `kops edit cluster useast1.dev.example.com`

- 使用以下命令编辑你的节点实例组： `kops edit ig --name = useast1.dev.example.com nodes`
- 使用以下命令编辑你的主实例组： `kops edit ig --name = useast1.dev.example.com master-us-east-1c`

如果这是你第一次使用 kops，请花几分钟尝试一下！ 实例组是一组实例，将被注册为 kubernetes 节点。 在 AWS 上，这是通过 auto-scaling-groups 实现的。你可以有多个实例组。 例如，如果你想要的是混合实例和按需实例的节点，或者 GPU 和非 GPU 实例。

(5/5) 在 AWS 中创建集群

运行 "kops update cluster" 以在 AWS 中创建集群：

```
kops update cluster useast1.dev.example.com --yes
```

这需要几秒钟的时间才能运行，但实际上集群可能需要几分钟才能准备就绪。 每当更改集群配置时，都会使用 `kops update cluster` 工具。 它将对配置进行的更改应用于你的集群 - 根据需要重新配置 AWS 或者 kubernetes。

例如，在你运行 `kops edit ig nodes` 之后，然后运行 `kops update cluster --yes` 应用你的配置，有时你还必须运行 `kops rolling-update cluster` 立即回滚更新配置。

如果没有 `--yes` 参数， `kops update cluster` 操作将向你显示其操作的预览效果。这对于生产集群很方便！

探索其他附加组件

请参阅附加组件列表探索其他附加组件， 包括用于 Kubernetes 集群的日志记录、监视、网络策略、可视化和控制的工具。

清理

- 删除集群： `kops delete cluster useast1.dev.example.com --yes`

接下来

- 了解有关 Kubernetes 的概念 和 `kubectl` 有关的更多信息。
- 了解 `kops` 高级用法。
- 请参阅 `kops` 文档 获取教程、 最佳做法和高级配置选项。

3.3 - 使用 Kubespray 安装 Kubernetes

此快速入门有助于使用 Kubespray 安装在 GCE、Azure、OpenStack、AWS、vSphere、Packet（裸机）、Oracle Cloud Infrastructure（实验性）或 Baremetal 上托管的 Kubernetes 集群。

Kubespray 是一个由 Ansible playbooks、 清单（inventory）、 制备工具和通用 OS/Kubernetes 集群配置管理任务的领域知识组成的。 Kubespray 提供：

- 高可用性集群
- 可组合属性
- 支持大多数流行的 Linux 发行版
  - Ubuntu 16.04、18.04、20.04
  - CentOS / RHEL / Oracle Linux 7、8
  - Debian Buster、Jessie、Stretch、Wheezy
  - Fedora 31、32
  - Fedora CoreOS
  - openSUSE Leap 15
  - Kinvolk 的 Flatcar Container Linux
- 持续集成测试

要选择最适合你的用例的工具，请阅读 kubeadm 和 kops 之间的 这份比较。 。

创建集群

（1/5）满足下层设施要求

按以下要求来配置服务器：

- 在将运行 Ansible 命令的计算机上安装 Ansible v2.9 和 python-netaddr
- **运行 Ansible Playbook 需要 Jinja 2.11（或更高版本）**
- 目标服务器必须有权访问 Internet 才能拉取 Docker 镜像。否则， 需要其他配置（请参见离线环境）
- 目标服务器配置为允许 IPv4 转发
- **你的 SSH 密钥必须复制**到部署集群的所有服务器中
- **防火墙不是由 kubespray 管理的**。你需要根据需求设置适当的规则策略。为了避免部署过程中出现问题，可以禁用防火墙。
- 如果从非 root 用户帐户运行 kubespray，则应在目标服务器中配置正确的特权升级方法 并指定 `ansible_become` 标志或命令参数 `--become` 或 `-b`

Kubespray 提供以下实用程序来帮助你设置环境：

- 为以下云驱动提供的 Terraform 脚本：
- AWS
- OpenStack
- Packet

（2/5）编写清单文件

设置服务器后，请创建一个 Ansible 的清单文件。 你可以手动执行此操作，也可以通过动态清单脚本执行此操作。有关更多信息，请参阅 “建立你自己的清单”。

（3/5）规划集群部署

Kubespray 能够自定义部署的许多方面：

- 选择部署模式： kubeadm 或非 kubeadm
- CNI（网络）插件
- DNS 配置
- 控制平面的选择：本机/可执行文件或容器化

- 组件版本
- Calico 路由反射器
- 组件运行时选项
  - Docker
  - containerd
  - CRI-O
- 证书生成方式

可以修改变量文件 以进行 Kubespray 定制。 如果你刚刚开始使用 Kubespray，请考虑使用 Kubespray 默认设置来部署你的集群 并探索 Kubernetes 。

（4/5）部署集群

接下来，部署你的集群：

使用 ansible-playbook 进行集群部署。

```
ansible-playbook -i your/inventory/inventory.ini cluster.yml -b -v \
  --private-key=~/.ssh/private_key
```

大型部署（超过 100 个节点）可能需要 特定的调整， 以获得最佳效果。

（5/5）验证部署

Kubespray 提供了一种使用 Netchecker 验证 Pod 间连接和 DNS 解析的方法。 Netchecker 确保 netchecker-agents Pods 可以解析 DNS 请求， 并在默认命名空间内对每个请求执行 ping 操作。 这些 Pod 模仿其他工作负载类似的行为，并用作集群运行状况指示器。

集群操作

Kubespray 提供了其他 Playbooks 来管理集群： *scale* 和 *upgrade*。

扩展集群

你可以通过运行 scale playbook 向集群中添加工作节点。有关更多信息， 请参见 “添加节点”。 你可以通过运行 remove-node playbook 来从集群中删除工作节点。有关更多信息， 请参见 “删除节点”。

升级集群

你可以通过运行 upgrade-cluster Playbook 来升级集群。有关更多信息，请参见 “升级”。

清理

你可以通过 reset Playbook 重置节点并清除所有与 Kubespray 一起安装的组件。

> **注意：** 运行 reset playbook 时，请确保不要意外地将生产集群作为目标！

反馈

- Slack 频道：#kubespray （你可以在此处获得邀请）
- GitHub 问题

接下来

查看有关 Kubespray 的 路线图 的计划工作。

4 - Windows Kubernetes

4.1 - Kubernetes 对 Windows 的支持

在很多组织中，其服务和应用的很大比例是 Windows 应用。 Windows 容器提供了一种对进程和包依赖关系 进行封装的现代方式，这使得用户更容易采用 DevOps 实践，令 Windows 应用同样遵从云原生模式。 Kubernetes 已经成为事实上的标准容器编排器，Kubernetes 1.14 发行版本中包含了将 Windows 容器调度到 Kubernetes 集群中 Windows 节点上的生产级支持，从而使得巨大 的 Windows 应用生态圈能够充分利用 Kubernetes 的能力。 对于同时投入基于 Windows 应用和 Linux 应用的组织而言，他们不必寻找不同的编排系统 来管理其工作负载，其跨部署的运维效率得以大幅提升，而不必关心所用操作系统。

kubernetes 中的 Windows 容器

若要在 Kubernetes 中启用对 Windows 容器的编排，可以在现有的 Linux 集群中 包含 Windows 节点。在 Kubernetes 上调度 Pods 中的 Windows 容器与调用基于 Linux 的容器类似。

为了运行 Windows 容器，你的 Kubernetes 集群必须包含多个操作系统，控制面 节点运行 Linux，工作节点则可以根据负载需要运行 Windows 或 Linux。 Windows Server 2019 是唯一被支持的 Windows 操作系统，在 Windows 上启用 Kubernetes 节点 支持（包括 kubelet, 容器运行时、 以及 kube-proxy）。关于 Windows 发行版渠道的详细讨论，可参见 Microsoft 文档。

> **说明：** Kubernetes 控制面，包括主控组件， 继续在 Linux 上运行。 目前没有支持完全是 Windows 节点的 Kubernetes 集群的计划。

> **说明：** 在本文中，当我们讨论 Windows 容器时，我们所指的是具有进程隔离能力的 Windows 容器。具有 Hyper-V 隔离能力 的 Windows 容器计划在将来发行版本中推出。

支持的功能与局限性

支持的功能

Windows 操作系统版本支持

参考下面的表格，了解 Kubernetes 中支持的 Windows 操作系统。 同一个异构的 Kubernetes 集群中可以同时包含 Windows 和 Linux 工作节点。 Windows 容器仅能调度到 Windows 节点，Linux 容器则只能调度到 Linux 节点。

Kubernetes 版本	Windows Server LTSC 版本	Windows Server SAC 版本
Kubernetes v1.20	Windows Server 2019	Windows Server ver 1909, Windows Server ver 2004
Kubernetes v1.21	Windows Server 2019	Windows Server ver 2004, Windows Server ver 20H2
Kubernetes v1.22	Windows Server 2019	Windows Server ver 2004, Windows Server ver 20H2

关于不同的 Windows Server 版本的服务渠道，包括其支持模式等相关信息可以在 Windows Server servicing channels 找到。

我们并不指望所有 Windows 客户都为其应用频繁地更新操作系统。 对应用的更新是向集群中引入新代码的根本原因。 对于想要更新运行于 Kubernetes 之上的容器中操作系统的客户，我们会在添加对新 操作系统版本的支持时提供指南和分步的操作指令。 该指南会包含与集群节点一起来升级用

户应用的建议升级步骤。 Windows 节点遵从 Kubernetes 版本偏差策略（节点到控制面的 版本控制），与 Linux 节点的现行策略相同。

Windows Server 主机操作系统会受 Windows Server 授权策略控制。Windows 容器镜像则遵从 Windows 容器的补充授权条款 约定。

带进程隔离的 Windows 容器受一些严格的兼容性规则约束， 其中宿主 OS 版本必须与容器基准镜像的 OS 版本相同。 一旦我们在 Kubernetes 中支持带 Hyper-V 隔离的 Windows 容器， 这一约束和兼容性规则也会发生改变。

Pause 镜像

Kubernetes 维护着一个多体系结构镜像，其中包括对 Windows 的支持。 对于 Kubernetes v1.22，推荐的 pause 镜像是 `k8s.gcr.io/pause:3.5` 。 源代码可在 GitHub 上找到。

Microsoft 维护了一个支持 Linux 和 Windows amd64 的多体系结构镜像：`mcr.microsoft.com/oss/kubernetes/pause:3.5` 。 此镜像与 Kubernetes 维护的镜像是从同一来源构建，但所有 Windows 二进制文件 均由 Microsoft 签名。 当生产环境需要被签名的二进制文件时，建议使用 Microsoft 维护的镜像。

计算

从 API 和 kubectl 的角度，Windows 容器的表现在很大程度上与基于 Linux 的容器 是相同的。不过也有一些与关键功能相关的差别值得注意，这些差别列举于 局限性小节中。

关键性的 Kubernetes 元素在 Windows 下与其在 Linux 下工作方式相同。我们在本节中 讨论一些关键性的负载支撑组件及其在 Windows 中的映射。

- Pods

  Pod 是 Kubernetes 中最基本的构造模块，是 Kubernetes 对象模型中你可以创建或部署的 最小、最简单元。你不可以在同一 Pod 中部署 Windows 和 Linux 容器。 Pod 中的所有容器都会被调度到同一节点（Node），而每个节点代表的是一种特定的平台 和体系结构。Windows 容器支持 Pod 的以下能力、属性和事件：

  - 在带进程隔离和卷共享支持的 Pod 中运行一个或多个容器
  - Pod 状态字段
  - 就绪态（Readiness）和活跃性（Liveness）探针
  - postStart 和 preStop 容器生命周期事件
  - ConfigMap、Secrets：用作环境变量或卷
  - emptyDir 卷
  - 从宿主系统挂载命名管道
  - 资源限制

- 控制器（Controllers）

  Kubernetes 控制器处理 Pod 的期望状态。Windows 容器支持以下负载控制器：

  - ReplicaSet
  - ReplicationController
  - Deployment
  - StatefulSet
  - DaemonSet
  - Job
  - CronJob

- 服务（Services）

  Kubernetes Service 是一种抽象对象，用来定义 Pod 的一个逻辑集合及用来访问这些 Pod 的策略。Service 有时也称作微服务（Micro-service）。你可以使用服务来实现 跨操作系统的连接。在 Windows 系统中，服务可以使用下面的类型、属性和能力：

  - Service 环境变量
  - NodePort
  - ClusterIP

    - LoadBalancer
    - ExternalName
    - 无头（Headless）服务

Pods、控制器和服务是在 Kubernetes 上管理 Windows 负载的关键元素。 不过，在一个动态的云原生环境中，这些元素本身还不足以用来正确管理 Windows 负载的生命周期。我们为此添加了如下功能特性：

- Pod 和容器的度量（Metrics）
- 对水平 Pod 自动扩展的支持
- 对 kubectl exec 命令的支持
- 资源配额
- 调度器抢占

容器运行时

Docker EE

FEATURE STATE: `Kubernetes v1.14 [stable]`

Docker EE-basic 19.03+ 是建议所有 Windows Server 版本采用的容器运行时。 该容器运行时能够与 kubelet 中的 dockershim 代码协同工作。

CRI-ContainerD

FEATURE STATE: `Kubernetes v1.20 [stable]`

ContainerD 1.4.0+ 也可作为 Windows Kubernetes 节点上的容器运行时。

持久性存储

使用 Kubernetes 卷，对数据持久性和 Pod 卷 共享有需求的复杂应用也可以部署到 Kubernetes 上。 管理与特定存储后端或协议相关的持久卷时，相关的操作包括：对卷的配备（Provisioning）、 去配（De-provisioning）和调整大小，将卷挂接到 Kubernetes 节点或从节点上解除挂接， 将卷挂载到需要持久数据的 Pod 中的某容器或从容器上卸载。 负责实现为特定存储后端或协议实现卷管理动作的代码以 Kubernetes 卷 插件的形式发布。 Windows 支持以下大类的 Kubernetes 卷插件：

树内卷插件

与树内卷插件（In-Tree Volume Plugin）相关的代码都作为核心 Kubernetes 代码基 的一部分发布。树内卷插件的部署不需要安装额外的脚本，也不需要额外部署独立的 容器化插件组件。这些插件可以处理：对应存储后端上存储卷的配备、去配和尺寸更改， 将卷挂接到 Kubernetes 或从其上解挂，以及将卷挂载到 Pod 中各个容器上或从其上 卸载。以下树内插件支持 Windows 节点：

- awsElasticBlockStore
- azureDisk
- azureFile
- gcePersistentDisk
- vsphereVolume

FlexVolume 插件

与 FlexVolume 插件相关的代码是作为 树外（Out-of-tree）脚本或可执行文件来发布的，因此需要在宿主系统上直接部署。 FlexVolume 插件处理将卷挂接到 Kubernetes 节点或从其上解挂、将卷挂载到 Pod 中 各个容器上或从其上卸载等操作。对于与 FlexVolume 插件相关联的持久卷的配备和去配操作，可以通过外部的配置程序来处理。这类配置程序通常与 FlexVolume 插件 相分离。下面的 FlexVolume 插件 可以以 PowerShell 脚本的形式部署到宿主系统上，支持 Windows 节点：

- SMB
- iSCSI

CSI 插件

FEATURE STATE: `Kubernetes v1.22 [stable]`

与 CSI 插件相关联的代码作为 树外脚本和可执行文件来发布且通常发布为容器镜像形式，并使用 DaemonSet 和 StatefulSet 这类标准的 Kubernetes 构造体来部署。 CSI 插件处理 Kubernetes 中的很多卷管理操作：对卷的配备、去配和调整大小， 将卷挂接到 Kubernetes 节点或从节点上解除挂接，将卷挂载到需要持久数据的 Pod 中的某容器或从容器上卸载，使用快照和克隆来备份或恢复持久数据。

来支持；csi-proxy 是一个社区管理的、独立的可执行文件，需要预安装在每个 Windows 节点之上。请参考你要部署的 CSI 插件的部署指南以进一步了解其细节。

CSI 插件与执行本地存储操作的 CSI 节点插件通信。 在 Windows 节点上，CSI 节点插件通常调用处理本地存储操作的 csi-proxy 公开的 API, csi-proxy 由社区管理。

有关安装的更多详细信息，请参阅你要部署的 Windows CSI 插件的环境部署指南。 你也可以参考以下安装步骤 。

联网

Windows 容器的联网是通过 CNI 插件 来暴露出来的。Windows 容器的联网行为与虚拟机的联网行为类似。 每个容器有一块虚拟的网络适配器（vNIC）连接到 Hyper-V 的虚拟交换机（vSwitch）。宿主的联网服务（Host Networking Service，HNS）和宿主计算服务（Host Compute Service，HCS）协同工作，创建容器并将容器的虚拟网卡连接到网络上。 HCS 负责管理容器，HNS 则负责管理网络资源，例如：

- 虚拟网络（包括创建 vSwitch）
- 端点（Endpoint）/ vNIC
- 名字空间（Namespace）
- 策略（报文封装、负载均衡规则、访问控制列表、网络地址转译规则等等）

支持的服务规约类型如下：

- NodePort
- ClusterIP
- LoadBalancer
- ExternalName

网络模式

Windows 支持五种不同的网络驱动/模式：二层桥接（L2bridge）、二层隧道（L2tunnel）、 覆盖网络（Overlay）、透明网络（Transparent）和网络地址转译（NAT）。 在一个包含 Windows 和 Linux 工作节点的异构集群中，你需要选择一种对 Windows 和 Linux 兼容的联网方案。下面是 Windows 上支持的一些树外插件及何时使用某种 CNI 插件的建议：

网络驱动	描述	容器报文更改	网络插件	网络插件特点

网络驱动	描述	容器报文更改	网络插件	网络插件特点
L2bridge	容器挂接到外部 vSwitch 上。容器挂接到下层网络之上，但由于容器的 MAC 地址在入站和出站时被重写，物理网络不需要这些地址。	MAC 地址被重写为宿主系统的 MAC 地址，IP 地址也可能依据 HNS OutboundNAT 策略重写为宿主的 IP 地址。	win-bridge、Azure-CNI、Flannel 宿主网关（host-gateway）使用 win-bridge	win-bridge 使用二层桥接（L2bridge）网络模式，将容器连接到下层宿主系统上， 从而提供最佳性能。需要用户定义的路由（User-Defined Routes，UDR）才能 实现节点间的连接。
L2Tunnel	这是二层桥接的一种特殊情形，但仅被用于 Azure 上。所有报文都被发送到虚拟化环境中的宿主机上并根据 SDN 策略进行处理。	MAC 地址被改写，IP 地址在下层网络上可见。	Azure-CNI	Azure-CNI 使得容器能够与 Azure vNET 集成，并允许容器利用 [Azure 虚拟网络](https://azure.microsoft.com/en-us/services/virtual-network/) 所提供的功能特性集合。例如，可以安全地连接到 Azure 服务上或者使用 Azure NSG。 你可以参考 [azure-cni](https://docs.microsoft.com/en-us/azure/aks/concepts-network#azure-cni-advanced-networking) 所提供的一些示例。
覆盖网络（Kubernetes 中为 Windows 提供的覆盖网络支持处于 *alpha* 阶段)	每个容器会获得一个连接到外部 vSwitch 的虚拟网卡（vNIC）。 每个覆盖网络都有自己的、通过定制 IP 前缀来定义的 IP 子网。 覆盖网络驱动使用 VxLAN 封装。	封装于外层包头内。	Win-overlay、Flannel VXLAN（使用 win-overlay）	当（比如出于安全原因）期望虚拟容器网络与下层宿主网络隔离时，应该使用 win-overlay。如果你的数据中心可用 IP 地址受限， 覆盖网络允许你在不同的网络中复用 IP 地址（每个覆盖网络有不同的 VNID 标签）。 这一选项要求在 Windows Server 2009 上安装 [KB4489899](https://support.microsoft.com/help/4489899) 补丁。

网络驱动	描述	容器报文更改	网络插件	网络插件特点
透明网络（[ovn-kubernetes](https://github.com/openvswitch/ovn-kubernetes) 的特殊用例）	需要一个外部 vSwitch。容器挂接到某外部 vSwitch 上，该 vSwitch 通过逻辑网络（逻辑交换机和路由器）允许 Pod 间通信。	报文或者通过 [GENEVE](https://datatracker.ietf.org/doc/draft-gross-geneve/) 来封装，或者通过 [STT](https://datatracker.ietf.org/doc/draft-davie-stt/) 隧道来封装，以便能够到达不在同一宿主系统上的每个 Pod。 报文通过 OVN 网络控制器所提供的隧道元数据信息来判定是转发还是丢弃。 北-南向通信通过 NAT 网络地址转译来实现。	ovn-kubernetes	[通过 Ansible 来部署](https://github.com/openvswitch/ovn-kubernetes/tree/master/contrib)。 所发布的 ACL 可以通过 Kubernetes 策略来应用实施。支持 IPAM 。 负载均衡能力不依赖 kube-proxy。 网络地址转译（NAT）也不需要 iptables 或 netsh。
NAT（**未在 Kubernetes 中使用**）	容器获得一个连接到某内部 vSwitch 的 vNIC 接口。DNS/DHCP 服务通过名为 [WinNAT](https://blogs.technet.microsoft.com/virtualization/2016/05/25/windows-nat-winnat-capabilities-and-limitations/) 的内部组件来提供。	MAC 地址和 IP 地址都被重写为宿主系统的 MAC 地址和 IP 地址。	nat	列在此表中仅出于完整性考虑

如前所述，Flannel CNI meta 插件 在 Windows 上也是 被支持 的，方法是通过 VXLAN 网络后端（**alpha 阶段**：委托给 win-overlay）和 主机-网关（host-gateway）网络后端 （稳定版本；委托给 win-bridge 实现）。 此插件支持将操作委托给所引用的 CNI 插件（win-overlay、win-bridge）之一， 从而能够与 Windows 上的 Flannel 守护进程（Flanneld）一同工作，自动为节点 分配子网租期，创建 HNS 网络。 该插件读入其自身的配置文件（cni.conf），并将其与 FlannelD 所生成的 subnet.env 文件中的环境变量整合，之后将其操作委托给所引用的 CNI 插件之一以完成网络发现，并将包含节点所被分配的子网信息的正确配置发送给 IPAM 插件（例如 host-local）。

对于节点、Pod 和服务对象，可针对 TCP/UDP 流量支持以下网络数据流：

- Pod -> Pod （IP 寻址）
- Pod -> Pod （名字寻址）
- Pod -> 服务（集群 IP）
- Pod -> 服务（部分限定域名，仅适用于名称中不包含“.”的情形）
- Pod -> 服务（全限定域名）
- Pod -> 集群外部（IP 寻址）
- Pod -> 集群外部（DNS 寻址）
- 节点 -> Pod
- Pod -> 节点

IP 地址管理（IPAM）

Windows 上支持以下 IPAM 选项：

- host-local
- HNS IPAM (Inbox 平台 IPAM，未指定 IPAM 时的默认设置)
- Azure-vnet-ipam（仅适用于 azure-cni ）

负载均衡与服务

在 Windows 系统上，你可以使用以下配置来设定服务和负载均衡行为：

功能特性	描述	所支持的 Kubernetes 版本	所支持的 Windows OS 版本	如何启用
会话亲和性	确保来自特定客户的连接每次都被交给同一 Pod。	v1.20+	[Windows Server vNext Insider Preview Build 19551] (https://blogs.windows.com/windowsexperience/2020/01/28/announcing-windows-server-vnext-insider-preview-build-19551/) 或更高版本	将 `service.spec.sessionAffinitys` 设置为 "ClientIP"
直接服务器返回（DSR）	这是一种负载均衡模式，IP 地址的修正和负载均衡地址转译（LBNAT） 直接在容器的 vSwitch 端口上处理；服务流量到达时，其源端 IP 地址 设置为来源 Pod 的 IP。	v1.20+	Windows Server 2019	为 kube-proxy 设置标志：`--feature-gates="WinDSR=true" --enable-dsr=true`
保留目标地址	对服务流量略过 DNAT 步骤，这样就可以在到达后端 Pod 的报文中保留目标服务的 虚拟 IP 地址。还要禁止节点之间的转发。	v1.20+	Windows Server 1903 或更高版本	在服务注解中设置 `"preserve-destination": "true"` 并启用 kube-proxy 中的 DSR 标志。
IPv4/IPv6 双栈网络	在集群内外同时支持原生的 IPv4-到-IPv4 和 IPv6-到-IPv6 通信。	v1.19+	Windows Server 2004 或更高版本	参见 [IPv4/IPv6 双栈网络] (#ipv4ipv6-dual-stack)

功能特性	描述	所支持的 Kubernetes 版本	所支持的 Windows OS 版本	如何启用
保留客户端 IP	确保入站流量的源 IP 地址被保留。同样要禁止节点之间的转发。	v1.20+	Windows Server 2019 或更高版本	将 `service.spec.externalTrafficPolicy` 设置为 "Local"，并在 kube-proxy 上启用 DSR。

IPv4/IPv6 双栈支持

你可以通过使用 `IPv6DualStack` 特性门控 来为 `l2bridge` 网络启用 IPv4/IPv6 双栈联网支持。 进一步的细节可参见 启用 IPv4/IPv6 双协议栈。

对 Windows 而言，在 Kubernetes 中使用 IPv6 需要 Windows Server 2004 （内核版本 10.0.19041.610）或更高版本。

目前 Windows 上的覆盖网络（VXLAN）还不支持双协议栈联网。

局限性

在 Kubernetes 架构和节点阵列中仅支持将 Windows 作为工作节点使用。 这意味着 Kubernetes 集群必须总是包含 Linux 主控节点，零个或者多个 Linux 工作节点以及零个或者多个 Windows 工作节点。

资源处理

Linux 上使用 Linux 控制组（CGroups）作为 Pod 的边界，以实现资源控制。 容器都创建于这一边界之内，从而实现网络、进程和文件系统的隔离。 控制组 CGroups API 可用来收集 CPU、I/O 和内存的统计信息。 与此相比，Windows 为每个容器创建一个带有系统名字空间过滤设置的 Job 对象， 以容纳容器中的所有进程并提供其与宿主系统间的逻辑隔离。 没有现成的名字空间过滤设置是无法运行 Windows 容器的。 这也意味着，系统特权无法在宿主环境中评估，因而 Windows 上也就不存在特权容器。 归咎于独立存在的安全账号管理器（Security Account Manager，SAM），容器也不能 获得宿主系统上的任何身份标识。

资源预留

内存预留

Windows 不像 Linux 一样有一个内存耗尽（Out-of-memory）进程杀手（Process Killer）机制。 Windows 总是将用户态的内存分配视为虚拟请求，页面文件（Pagefile） 是必需的。这一差异的直接结果是 Windows 不会像 Linux 那样出现内存耗尽的状况， 系统会将进程内存页面写入磁盘而不会因内存耗尽而终止进程。 当内存被过量使用且所有物理内存都被用光时，系统的换页行为会导致性能下降。

使用 kubelet 参数 `--kubelet-reserve` 与/或 `-system-reserve` 可以统计 节点上的内存用量（各容器之外），进而可能将内存用量限制在一个合理的范围，。 这样做会减少节点可分配内存（NodeAllocatable）。

在你部署工作负载时，对容器使用资源限制（必须仅设置 limits 或者让 limits 等于 requests 值）。 这也会从 NodeAllocatable 中耗掉部分内存量，从而避免在节点 负荷已满时调度器继续向节点添加 Pods。

避免过量分配的最佳实践是为 kubelet 配置至少 2 GB 的系统预留内存，以供 Windows、Docker 和 Kubernetes 进程使用。

CPU 预留

为了统计 Windows、Docker 和其他 Kubernetes 宿主进程的 CPU 用量，建议 预留一定比例的 CPU，以便对事件作出相应。此值需要根据 Windows 节点上 CPU 核的个数来调整，要确定此百分比值，用户需要为其所有节点确定 Pod 密度的上线，并监控系统服务的 CPU 用量，从而选择一个符合其负载需求的值。

使用 kubelet 参数 `--kubelet-reserve` 与/或 `-system-reserve` 可以统计 节点上的 CPU 用量（各容器之外），进而可能将 CPU 用量限制在一个合理的范围，。 这样做会减少节点可分配 CPU（NodeAllocatable）。

功能特性限制

- 终止宽限期（Termination Grace Period）：未实现
- 单文件映射：将用 CRI-ContainerD 来实现
- 终止消息（Termination message）：将用 CRI-ContainerD 来实现
- 特权容器：Windows 容器当前不支持
- 巨页（Huge Pages）：Windows 容器当前不支持
- 现有的节点问题探测器（Node Problem Detector）仅适用于 Linux，且要求使用特权容器。一般而言，我们不设想此探测器能用于 Windows 节点，因为 Windows 不支持特权容器。
- 并非支持共享名字空间的所有功能特性（参见 API 节以了解详细信息）

与 Linux 相比参数行为的差别

以下 kubelet 参数的行为在 Windows 节点上有些不同，描述如下：

- `--kubelet-reserve`、 `--system-reserve` 和 `--eviction-hard` 标志 会更新节点可分配资源量
- 未实现通过使用 `--enforce-node-allocable` 来完成的 Pod 驱逐
- 未实现通过使用 `--eviction-hard` 和 `--eviction-soft` 来完成的 Pod 驱逐
- `MemoryPressure` 状况未实现
- `kubelet` 不会采取措施来执行基于 OOM 的驱逐动作
- Windows 节点上运行的 kubelet 没有内存约束。 `--kubelet-reserve` 和 `--system-reserve` 不会为 kubelet 或宿主系统上运行 的进程设限。这意味着 kubelet 或宿主系统上的进程可能导致内存资源紧张， 而这一情况既不受节点可分配量影响，也不会被调度器感知。
- 在 Windows 节点上存在一个额外的参数用来设置 kubelet 进程的优先级，称作 `--windows-priorityclass`。此参数允许 kubelet 进程获得与 Windows 宿主上 其他进程相比更多的 CPU 时间片。 关于可用参数值及其含义的进一步信息可参考 Windows Priority Classes。 为了让 kubelet 总能够获得足够的 CPU 周期，建议将此参数设置为 `ABOVE_NORMAL_PRIORITY_CLASS` 或更高。

存储

Windows 上包含一个分层的文件系统来挂载容器的分层，并会基于 NTFS 来创建一个 拷贝文件系统。容器中的所有文件路径都仅在该容器的上下文内完成解析。

- Docker 卷挂载仅可针对容器中的目录进行，不可针对独立的文件。 这一限制不适用于 CRI-containerD。
- 卷挂载无法将文件或目录投射回宿主文件系统。
- 不支持只读文件系统，因为 Windows 注册表和 SAM 数据库总是需要写访问权限。 不过，Windows 支持只读的卷。
- 不支持卷的用户掩码和访问许可，因为宿主与容器之间并不共享 SAM，二者之间不存在 映射关系。所有访问许可都是在容器上下文中解析的。

因此，Windows 节点上不支持以下存储功能特性：

- 卷的子路径挂载；只能在 Windows 容器上挂载整个卷。
- 为 Secret 执行子路径挂载；
- 宿主挂载投射；
- 默认访问模式 defaultMode（因为该特性依赖 UID/GID）；
- 只读的根文件系统；映射的卷仍然支持 `readOnly`；

- 块设备映射；
- 将内存作为存储介质；
- 类似 UUID/GUID、每用户不同的 Linux 文件系统访问许可等文件系统特性；
- 基于 NFS 的存储和卷支持；
- 扩充已挂载卷（resizefs）。

联网

Windows 容器联网与 Linux 联网有着非常重要的差别。 Microsoft documentation for Windows Container Networking 中包含额外的细节和背景信息。

Windows 宿主联网服务和虚拟交换机实现了名字空间隔离，可以根据需要为 Pod 或容器 创建虚拟的网络接口（NICs）。不过，很多类似 DNS、路由、度量值之类的配置数据都 保存在 Windows 注册表数据库中而不是像 Linux 一样保存在 `/etc/...` 文件中。 Windows 为容器提供的注册表与宿主系统的注册表是分离的，因此类似于将 /etc/resolv.conf 文件从宿主系统映射到容器中的做法不会产生与 Linux 系统相同的效果。 这些信息必须在容器内部使用 Windows API 来配置。 因此，CNI 实现需要调用 HNS，而不是依赖文件映射来将网络细节传递到 Pod 或容器中。

Windows 节点不支持以下联网功能：

- Windows Pod 不能使用宿主网络模式；
- 从节点本地访问 NodePort 会失败（但从其他节点或外部客户端可访问）
- Windows Server 的未来版本中会支持从节点访问服务的 VIP；
- 每个服务最多支持 64 个后端 Pod 或独立的目标 IP 地址；
- kube-proxy 的覆盖网络支持是 Beta 特性。此外，它要求在 Windows Server 2019 上安装 KB4482887 补丁；
- 非 DSR（保留目标地址）模式下的本地流量策略；
- 连接到覆盖网络的 Windows 容器不支持使用 IPv6 协议栈通信。 要使得这一网络驱动支持 IPv6 地址需要在 Windows 平台上开展大量的工作， 还需要在 Kubernetes 侧修改 kubelet、kube-proxy 以及 CNI 插件。

- 通过 win-overlay、win-bridge 和 Azure-CNI 插件使用 ICMP 协议向集群外通信。 尤其是，Windows 数据面 （VFP） 不支持转换 ICMP 报文。这意味着：
  - 指向同一网络内目标地址的 ICMP 报文（例如 Pod 之间的 ping 通信）是可以工作的，没有局限性；
  - TCP/UDP 报文可以正常工作，没有局限性；
  - 指向远程网络的 ICMP 报文（例如，从 Pod 中 ping 外部互联网的通信）无法被转换，因此也无法被路由回到其源点；
  - 由于 TCP/UDP 包仍可被转换，用户可以将 `ping <目标>` 操作替换为 `curl <目标>` 以便能够调试与外部世界的网络连接。

Kubernetes v1.15 中添加了以下功能特性：

- `kubectl port-forward`

CNI 插件

- Windows 参考网络插件 win-bridge 和 win-overlay 当前未实现 CNI spec v0.4.0， 原因是缺少检查（CHECK）用的实现。

- Windows 上的 Flannel VXLAN CNI 有以下局限性：

  1. 其设计上不支持从节点到 Pod 的连接。 只有在 Flannel v0.12.0 或更高版本后才有可能访问本地 Pods。
  2. 我们被限制只能使用 VNI 4096 和 UDP 端口 4789。 VNI 的限制正在被解决，会在将来的版本中消失（开源的 Flannel 更改）。 参见官方的 Flannel VXLAN 后端文档以了解关于这些参数的详细信息。

DNS

- 不支持 DNS 的 ClusterFirstWithHostNet 配置。Windows 将所有包含 “.” 的名字 视为全限定域名（FQDN），因而不会对其执行部分限定域名（PQDN）解析。

- 在 Linux 上，你可以有一个 DNS 后缀列表供解析部分限定域名时使用。 在 Windows 上，我们只有一个 DNS 后缀，即与该 Pod 名字空间相关联的 DNS 后缀（例如 `mydns.svc.cluster.local` ）。 Windows 可以解析全限定域名、或者恰好可用该后缀来解析的服务名称。 例如，在 default 名字空间中生成的 Pod 会获得 DNS 后缀 `default.svc.cluster.local` 。在 Windows Pod 中，你可以解析 `kubernetes.default.svc.cluster.local` 和 `kubernetes` ，但无法解析二者 之间的形式，如 `kubernetes.default` 或 `kubernetes.default.svc` 。

- 在 Windows 上，可以使用的 DNS 解析程序有很多。由于这些解析程序彼此之间 会有轻微的行为差别，建议使用 `Resolve-DNSName` 工具来完成名字查询解析。

IPv6

Windows 上的 Kubernetes 不支持单协议栈的“只用 IPv6”联网选项。 不过，系统支持在 IPv4/IPv6 双协议栈的 Pod 和节点上运行单协议家族的服务。 更多细节可参阅 IPv4/IPv6 双协议栈联网一节。

会话亲和性

不支持使用 `service.spec.sessionAffinityConfig.clientIP.timeoutSeconds` 来为 Windows 服务设置最大会话粘滞时间。

安全性

Secret 以明文形式写入节点的卷中（而不是像 Linux 那样写入内存或 tmpfs 中）。 这意味着客户必须做以下两件事：

1. 使用文件访问控制列表来保护 Secret 文件所在的位置
2. 使用 BitLocker 来执行卷层面的加密

用户可以为 Windows Pods 或 Container 设置 `RunAsUserName` 以便以非节点默认用户来执行容器中的进程。这大致等价于设置 `RunAsUser` 。

不支持特定于 Linux 的 Pod 安全上下文特权，例如 SELinux、AppArmor、Seccomp、 权能字（POSIX 权能字）等等。

此外，如前所述，Windows 不支持特权容器。

API

对 Windows 而言，大多数 Kubernetes API 的工作方式没有变化。 一些不易察觉的差别通常体现在 OS 和容器运行时上的不同。 在某些场合，负载 API （如 Pod 或 Container）的某些属性在设计时假定其 在 Linux 上实现，因此会无法在 Windows 上运行。

在较高层面，不同的 OS 概念有：

- 身份标识 - Linux 使用证书类型来表示用户 ID（UID）和组 ID（GID）。用户和组名 没有特定标准，它们是 `/etc/groups` 或 `/etc/passwd` 中的别名表项，会映射回 UID+GID。Windows 使用一个更大的二进制安全标识符（SID），保存在 Windows 安全访问管理器（Security Access Manager，SAM）数据库中。此数据库并不在宿主系统 与容器间，或者任意两个容器之间共享。

- 文件许可 - Windows 使用基于 SID 的访问控制列表，而不是基于 UID+GID 的访问权限位掩码。

- 文件路径 - Windows 上的习惯是使用 `\` 而非 `/` 。Go 语言的 IO 库同时接受这两种文件路径分隔符。不过，当你在指定要在容器内解析的路径或命令行时， 可能需要使用 `\` 。

- 信号（Signal） - Windows 交互式应用以不同方式来处理终止事件，并可实现以下方式之一或组合：

    - UI 线程处理包含 `WM_CLOSE` 在内的良定的消息
    - 控制台应用使用控制处理程序来处理 Ctrl-C 或 Ctrl-Break
    - 服务会注册服务控制处理程序，接受 `SERVICE_CONTROL_STOP` 控制代码

退出代码遵从相同的习惯，0 表示成功，非 0 值表示失败。 特定的错误代码在 Windows 和 Linux 上可能会不同。不过，从 Kubernetes 组件 （kubelet、kube-proxy）所返回的退出代码是没有变化的。

- `v1.Container.ResourceRequirements.limits.cpu` 和 `v1.Container.ResourceRequirements.limits.memory` - Windows 不对 CPU 分配设置硬性的限制。与之相反，Windows 使用一个份额（share）系统。 基于毫核（millicores）的现有字段值会被缩放为相对的份额值，供 Windows 调度器使用。 参见 kuberuntime/helpers_windows.go 和 Microsoft 文档中关于资源控制的部分。
  - Windows 容器运行时中没有实现巨页支持，因此相关特性不可用。 巨页支持需要判定用户的特权 而这一特性无法在容器级别配置。
- `v1.Container.ResourceRequirements.requests.cpu` 和 `v1.Container.ResourceRequirements.requests.memory` - 请求 值会从节点可分配资源中扣除，从而可用来避免节点上的资源过量分配。 但是，它们无法用来在一个已经过量分配的节点上提供资源保障。 如果操作员希望彻底避免过量分配，作为最佳实践，他们就需要为所有容器设置资源请求值。
- `v1.Container.SecurityContext.allowPrivilegeEscalation` - 在 Windows 上无法实现，对应的权能无一可在 Windows 上生效。
- `v1.Container.SecurityContext.Capabilities` - Windows 上未实现 POSIX 权能机制
- `v1.Container.SecurityContext.privileged` - Windows 不支持特权容器
- `v1.Container.SecurityContext.procMount` - Windows 不包含 `/proc` 文件系统
- `v1.Container.SecurityContext.readOnlyRootFilesystem` - 在 Windows 上无法实现， 要在容器内使用注册表或运行系统进程就必需写访问权限。
- `v1.Container.SecurityContext.runAsGroup` - 在 Windows 上无法实现，没有 GID 支持
- `v1.Container.SecurityContext.runAsNonRoot` - Windows 上没有 root 用户。 与之最接近的等价用户是 `ContainerAdministrator`，而该身份标识在节点上并不存在。
- `v1.Container.SecurityContext.runAsUser` - 在 Windows 上无法实现， 因为没有作为整数支持的 GID。
- `v1.Container.SecurityContext.seLinuxOptions` - 在 Windows 上无法实现， 因为没有 SELinux
- `V1.Container.terminationMessagePath` - 因为 Windows 不支持单个文件的映射，这一功能 在 Windows 上也受限。默认值 `/dev/termination-log` 在 Windows 上也无法使用因为 对应路径在 Windows 上不存在。

V1.Pod

- `v1.Pod.hostIPC`、 `v1.Pod.hostPID` - Windows 不支持共享宿主系统的名字空间
- `v1.Pod.hostNetwork` - Windows 操作系统不支持共享宿主网络
- `v1.Pod.dnsPolicy` - 不支持 `ClusterFirstWithHostNet`，因为 Windows 不支持宿主网络
- `v1.Pod.podSecurityContext` - 参见下面的 `v1.PodSecurityContext`
- `v1.Pod.shareProcessNamespace` - 此为 Beta 特性且依赖于 Windows 上未实现 的 Linux 名字空间。 Windows 无法共享进程名字空间或者容器的根文件系统。只能共享网络。
- `v1.Pod.terminationGracePeriodSeconds` - 这一特性未在 Windows 版本的 Docker 中完全实现。 参见问题报告。 目前实现的行为是向 `ENTRYPOINT` 进程发送 `CTRL_SHUTDOWN_EVENT` 事件，之后 Windows 默认 等待 5 秒钟，并最终使用正常的 Windows 关机行为关闭所有进程。这里的 5 秒钟默认值实际上保存在 容器内 的 Windows 注册表中，因此可以在构造容器时重载。
- `v1.Pod.volumeDevices` - 此为 Beta 特性且未在 Windows 上实现。Windows 无法挂接 原生的块设备到 Pod 中。
- `v1.Pod.volumes` - `emptyDir`、 `secret`、 `configMap` 和 `hostPath` 都可正常工作且在 TestGrid 中测试。
  - `v1.emptyDir.volumeSource` - Windows 上节点的默认介质是磁盘。 不支持将内存作为介质，因为 Windows 不支持内置的 RAM 磁盘。

- `v1.VolumeMount.mountPropagation` - Windows 上不支持挂载传播。

V1.PodSecurityContext

PodSecurityContext 的所有选项在 Windows 上都无法工作。这些选项列在下面仅供参考。

- `v1.PodSecurityContext.seLinuxOptions` - Windows 上无 SELinux
- `v1.PodSecurityContext.runAsUser` - 提供 UID；Windows 不支持
- `v1.PodSecurityContext.runAsGroup` - 提供 GID；Windows 不支持
- `v1.PodSecurityContext.runAsNonRoot` - Windows 上没有 root 用户 最接近的等价账号是 `ContainerAdministrator`，而该身份标识在节点上不存在
- `v1.PodSecurityContext.supplementalGroups` - 提供 GID；Windows 不支持
- `v1.PodSecurityContext.sysctls` - 这些是 Linux sysctl 接口的一部分；Windows 上 没有等价机制。

操作系统版本限制

Windows 有着严格的兼容性规则，宿主 OS 的版本必须与容器基准镜像 OS 的版本匹配。 目前仅支持容器操作系统为 Windows Server 2019 的 Windows 容器。 对于容器的 Hyper-V 隔离、允许一定程度上的 Windows 容器镜像版本向后兼容性等等， 都是将来版本计划的一部分。

获取帮助和故障排查

对你的 Kubernetes 集群进行排查的主要帮助信息来源应该是 这份文档。 该文档中包含了一些额外的、特定于 Windows 系统的故障排查帮助信息。 Kubernetes 中日志是故障排查的一个重要元素。确保你在尝试从其他贡献者那里获得 故障排查帮助时提供日志信息。你可以按照 SIG-Windows 贡献指南和收集日志 所给的指令来操作。

- 我怎样知道 `start.ps1` 是否已成功完成?

  你应该能看到节点上运行的 kubelet、kube-proxy 和（如果你选择 Flannel 作为联网方案）flanneld 宿主代理进程，它们的运行日志显示在不同的 PowerShell 窗口中。此外，你的 Windows 节点应该在你的 Kubernetes 集群 列举为 "Ready" 节点。

- 我可以将 Kubernetes 节点进程配置为服务运行在后台么?

  kubelet 和 kube-proxy 都已经被配置为以本地 Windows 服务运行， 并且在出现失效事件（例如进程意外结束）时通过自动重启服务来提供一定的弹性。 你有两种办法将这些节点组件配置为服务。

  - 以本地 Windows 服务的形式

    Kubelet 和 kube-proxy 可以用 `sc.exe` 以本地 Windows 服务的形式运行：

```
# 用两个单独的命令为 kubelet 和 kube-proxy 创建服务
sc.exe create <组件名称> binPath="<可执行文件路径> -service <其它参数>"

# 请注意如果参数中包含空格，必须使用转义
sc.exe create kubelet binPath= "C:\kubelet.exe --service --hostname-override '

# 启动服务
Start-Service kubelet
Start-Service kube-proxy

# 停止服务
Stop-Service kubelet (-Force)
Stop-Service kube-proxy (-Force)

# 查询服务状态
Get-Service kubelet
Get-Service kube-proxy
```

```
```

- 使用 nssm.exe

  你也总是可以使用替代的服务管理器，例如nssm.exe，来为你在后台运行 这些进程（ `flanneld` 、 `kubelet` 和 `kube-proxy` ）。你可以使用这一 示例脚本， 利用 `nssm.exe` 将 `kubelet` 、 `kube-proxy` 和 `flanneld.exe` 注册为要在后台运行的 Windows 服务。

```
register-svc.ps1 -NetworkMode <网络模式> -ManagementIP <Windows 节点 IP> -Clust
```

这里的参数解释如下：

- `NetworkMode` ： 网络模式 l2bridge（flannel host-gw，也是默认值）或 overlay（flannel vxlan）选做网络方案
- `ManagementIP` ： 分配给 Windows 节点的 IP 地址。你可以使用 ipconfig 得到此值
- `ClusterCIDR` ： 集群子网范围（默认值为 10.244.0.0/16）
- `KubeDnsServiceIP` ： Kubernetes DNS 服务 IP（默认值为 10.96.0.10）
- `LogDir` ： kubelet 和 kube-proxy 的日志会被重定向到这一目录中的对应输出文件， 默认值为 `C:\k` 。

若以上所引用的脚本不适合，你可以使用下面的例子手动配置 `nssm.exe` 。

注册 flanneld.exe：

```
nssm install flanneld C:\flannel\flanneld.exe
nssm set flanneld AppParameters --kubeconfig-file=c:\k\config --iface=<Managem
nssm set flanneld AppEnvironmentExtra NODE_NAME=<hostname>
nssm set flanneld AppDirectory C:\flannel
nssm start flanneld
```

注册 kubelet.exe：

```
nssm install kubelet C:\k\kubelet.exe
nssm set kubelet AppParameters --hostname-override=<hostname> --v=6 --pod-infr
nssm set kubelet AppDirectory C:\k
nssm start kubelet
```

注册 kube-proxy.exe（二层网桥模式和主机网关模式）

```
nssm install kube-proxy C:\k\kube-proxy.exe
nssm set kube-proxy AppDirectory c:\k
nssm set kube-proxy AppParameters --v=4 --proxy-mode=kernelspace --hostname-ov
nssm.exe set kube-proxy AppEnvironmentExtra KUBE_NETWORK=cbr0
nssm set kube-proxy DependOnService kubelet
nssm start kube-proxy
```

注册 kube-proxy.exe（覆盖网络模式或 VxLAN 模式）

```
nssm install kube-proxy C:\k\kube-proxy.exe
nssm set kube-proxy AppDirectory c:\k
nssm set kube-proxy AppParameters --v=4 --proxy-mode=kernelspace --feature-gat
```

```
nssm set kube-proxy DependOnService kubelet
nssm start kube-proxy
```

作为初始的故障排查操作，你可以使用在 nssm.exe 中使用下面的标志 以便将标准输出和标准错误输出重定向到一个输出文件：

```
nssm set <服务名称> AppStdout C:\k\mysvc.log
nssm set <服务名称> AppStderr C:\k\mysvc.log
```

要了解更多的细节，可参见官方的 nssm 用法文档。

- 我的 Windows Pods 无发连接网络

  如果你在使用虚拟机，请确保 VM 网络适配器均已开启 MAC 侦听（Spoofing）。

- 我的 Windows Pods 无法 ping 外部资源

  Windows Pods 目前没有为 ICMP 协议提供出站规则。不过 TCP/UDP 是支持的。 尝试与集群外资源连接时，可以将 `ping <IP>` 命令替换为对应的 `curl <IP>` 命令。

  如果你还遇到问题，很可能你在 cni.conf 中的网络配置值得额外的注意。你总是可以编辑这一静态文件。 配置的更新会应用到所有新创建的 Kubernetes 资源上。

  Kubernetes 网络的需求之一（参见 Kubernetes 网络模型） 是集群内部无需网络地址转译（NAT）即可实现通信。 为了符合这一要求，对所有我们不希望出站时发生 NAT 的通信都存在一个 ExceptionList。 然而这也意味着你需要将你要查询的外部 IP 从 ExceptionList 中移除。 只有这时，从你的 Windows Pod 发起的网络请求才会被正确地通过 SNAT 转换以接收到来自外部世界的响应。 就此而言，你在 `cni.conf` 中的 `ExceptionList` 应该看起来像这样：

  ```
  "ExceptionList": [
      "10.244.0.0/16",  # 集群子网
      "10.96.0.0/12",   # 服务子网
      "10.127.130.0/24" # 管理（主机）子网
  ]
  ```

- 我的 Windows 节点无法访问 NodePort 服务

  从节点自身发起的本地 NodePort 请求会失败。这是一个已知的局限。 NodePort 服务的访问从其他节点或者外部客户端都可正常进行。

- 容器的 vNICs 和 HNS 端点被删除了

  这一问题可能因为 `hostname-override` 参数未能传递给 kube-proxy 而导致。解决这一问题时，用户需要按如下方式将主机名传递给 kube-proxy：

  ```
  C:\k\kube-proxy.exe --hostname-override=$(hostname)
  ```

- 使用 Flannel 时，我的节点在重新加入集群时遇到问题

  无论何时，当一个之前被删除的节点被重新添加到集群时，flannelD 都会将为节点分配 一个新的 Pod 子网。 用户需要将将下面路径中的老的 Pod 子网配置文件删除：

  ```
  Remove-Item C:\k\SourceVip.json
  Remove-Item C:\k\SourceVipRequest.json
  ```

- 在启动了 `start.ps1` 之后，flanneld 一直停滞在 "Waiting for the Network to be created" 状态

关于这一问题有很多的报告； 最可能的一种原因是关于何时设置 Flannel 网络的管理 IP 的时间问题。 一种解决办法是重新启动 `start.ps1` 或者按如下方式手动重启之：

```
[Environment]::SetEnvironmentVariable("NODE_NAME", "<Windows 工作节点主机名>")
C:\flannel\flanneld.exe --kubeconfig-file=c:\k\config --iface=<Windows 工作节点 IP>
```

- 我的 Windows Pods 无法启动，因为缺少 `/run/flannel/subnet.env` 文件

  这表明 Flannel 网络未能正确启动。你可以尝试重启 flanneld.exe 或者将文件手动地 从 Kubernetes 主控节点的 `/run/flannel/subnet.env` 路径复制到 Windows 工作 节点的 `C:\run\flannel\subnet.env` 路径，并将 `FLANNEL_SUBNET` 行改为一个 不同的数值。例如，如果期望节点子网为 `10.244.4.1/24`：

  ```
  FLANNEL_NETWORK=10.244.0.0/16
  FLANNEL_SUBNET=10.244.4.1/24
  FLANNEL_MTU=1500
  FLANNEL_IPMASQ=true
  ```

- 我的 Windows 节点无法使用服务 IP 访问我的服务

  这是 Windows 上当前网络协议栈的一个已知的限制。 Windows Pods 能够访问服务 IP。

- 启动 kubelet 时找不到网络适配器

  Windows 网络堆栈需要一个虚拟的适配器，这样 Kubernetes 网络才能工作。 如果下面的命令（在管理员 Shell 中）没有任何返回结果，证明虚拟网络创建 （kubelet 正常工作的必要前提之一）失败了：

  ```
  Get-HnsNetwork | ? Name -ieq "cbr0"
  Get-NetAdapter | ? Name -Like "vEthernet (Ethernet*"
  ```

  当宿主系统的网络适配器名称不是 "Ethernet" 时，通常值得更改 `start.ps1` 脚本中的 InterfaceName 参数来重试。否则可以查验 `start-kubelet.ps1` 的输出，看看是否在虚拟网络创建 过程中报告了其他错误。

- 我的 Pods 停滞在 "Container Creating" 状态或者反复重启

  检查你的 pause 镜像是与你的 OS 版本兼容的。 这里的指令 假定你的 OS 和容器版本都是 1803。如果你安装的是更新版本的 Windows，比如说 某个 Insider 构造版本，你需要相应地调整要使用的镜像。 请参照 Microsoft 的 Docker 仓库 了解镜像。不管怎样，pause 镜像的 Dockerfile 和示例服务都期望镜像的标签 为 `:1atest` 。

- DNS 解析无法正常工作

  参阅 Windows 上 DNS 相关的局限 节。

- `kubectl port-forward` 失败，错误信息为 "unable to do port forwarding: wincat not found"

  此功能是在 Kubernetes v1.15 中实现的，pause 基础设施容器 `mcr.microsoft.com/oss/kubernetes/pause:3.4.1` 中包含了 wincat.exe。 请确保你使用的是这些版本或者更新版本。 如果你想要自行构造你自己的 pause 基础设施容器，要确保其中包含了 wincat

  Windows 的端口转发支持需要在 pause 基础设施容器 中提供 wincat.exe。 确保你使用的是与你的 Windows 操作系统版本兼容的受支持镜像。 如果你想构建自己的 pause 基础架构容器，请确保包含 wincat.。

- 我的 Kubernetes 安装失败，因为我的 Windows Server 节点在防火墙后面

  如果你处于防火墙之后，那么必须定义如下 PowerShell 环境变量：

```
[Environment]::SetEnvironmentVariable("HTTP_PROXY", "http://proxy.example.com:80/",
[Environment]::SetEnvironmentVariable("HTTPS_PROXY", "http://proxy.example.com:443/
```

- `pause` 容器是什么?

  在一个 Kubernetes Pod 中，一个基础设施容器，或称 "pause" 容器，会被首先创建出来，用以托管容器端点。属于同一 Pod 的容器，包括基础设施容器和工作容器，会共享相同的 网络名字空间和端点（相同的 IP 和端口空间）。我们需要 pause 容器来工作容器崩溃或 重启的状况，以确保不会丢失任何网络配置。

  请参阅 pause 镜像 部分以查找 pause 镜像的推荐版本。

进一步探究

如果以上步骤未能解决你遇到的问题，你可以通过以下方式获得在 Kubernetes 中的 Windows 节点上运行 Windows 容器的帮助:

- StackOverflow Windows Server Container 主题
- Kubernetes 官方论坛 discuss.kubernetes.io
- Kubernetes Slack #SIG-Windows 频道

报告问题和功能需求

如果你遇到看起来像是软件缺陷的问题，或者你想要提起某种功能需求，请使用 GitHub 问题跟踪系统。 你可以在 GitHub 上发起 Issue 并将其指派给 SIG-Windows。你应该首先搜索 Issue 列表，看看是否 该 Issue 以前曾经被报告过，以评论形式将你在该 Issue 上的体验追加进去，并附上 额外的日志信息。SIG-Windows Slack 频道也是一个获得初步支持的好渠道，可以在 生成新的 Ticket 之前对一些想法进行故障分析。

在登记软件缺陷时，请给出如何重现该问题的详细信息，例如:

- Kubernetes 版本：kubectl 版本
- 环境细节：云平台、OS 版本、网络选型和配置情况以及 Docker 版本
- 重现该问题的详细步骤
- 相关的日志
- 通过为该 Issue 添加 `/sig windows` 评论为其添加 `sig/windows` 标签， 进而引起 SIG-Windows 成员的注意。

接下来

在我们的未来蓝图中包含很多功能特性（要实现）。下面是一个浓缩的简要列表，不过我们 鼓励你查看我们的 roadmap 项目并 通过贡献的方式 帮助我们把 Windows 支持做得更好。

Hyper-V 隔离

要满足 Kubernetes 中 Windows 容器的如下用例，需要利用 Hyper-V 隔离:

- 在 Pod 之间实施基于监管程序（Hypervisor）的隔离，以增强安全性
- 出于向后兼容需要，允许添加运行新 Windows Server 版本的节点时不必 重新创建容器
- 为 Pod 设置特定的 CPU/NUMA 配置
- 实施内存隔离与预留

使用 kubeadm 和 Cluster API 来部署

kubeadm 已经成为用户部署 Kubernetes 集群的事实标准。 kubeadm 对 Windows 节点的支持目前还在开发过程中，不过你可以阅读相关的 指南。 我们也在投入资源到 Cluster API，以确保 Windows 节点被正确配置。

4.2 - Kubernetes 中 Windows 容器的调度指南

Windows 应用程序构成了许多组织中运行的服务和应用程序的很大一部分。 本指南将引导您完成在 Kubernetes 中配置和部署 Windows 容器的步骤。

目标

- 配置一个示例 deployment 以在 Windows 节点上运行 Windows 容器
- （可选）使用组托管服务帐户（GMSA）为您的 Pod 配置 Active Directory 身份

在你开始之前

- 创建一个 Kubernetes 集群，其中包括一个控制平面和 运行 Windows 服务器的工作节点
- 重要的是要注意，对于 Linux 和 Windows 容器，在 Kubernetes 上创建和部署服务和工作负载的行为几乎相同。 与集群接口的 kubectl 命令相同。 提供以下部分中的示例只是为了快速启动 Windows 容器的使用体验。

入门：部署 Windows 容器

要在 Kubernetes 上部署 Windows 容器，您必须首先创建一个示例应用程序。 下面的示例 YAML 文件创建了一个简单的 Web 服务器应用程序。 创建一个名为 `win-webserver.yaml` 的服务规约，其内容如下：

```yaml
apiVersion: v1
kind: Service
metadata:
  name: win-webserver
  labels:
    app: win-webserver
spec:
  ports:
    # the port that this service should serve on
    - port: 80
      targetPort: 80
  selector:
    app: win-webserver
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: win-webserver
  name: win-webserver
spec:
  replicas: 2
  selector:
    matchLabels:
      app: win-webserver
  template:
    metadata:
      labels:
        app: win-webserver
      name: win-webserver
    spec:
     containers:
      - name: windowswebserver
        image: mcr.microsoft.com/windows/servercore:ltsc2019
        command:
        - powershell.exe
```

```
    - -command
    - "<#code used from https://gist.github.com/19WAS85/5424431#> ; $$listener = New
  nodeSelector:
   kubernetes.io/os: windows
```

说明： 端口映射也是支持的，但为简单起见，在此示例中容器端口 80 直接暴露给服务。

1. 检查所有节点是否健康：

```
kubectl get nodes
```

2. 部署服务并观察 pod 更新：

```
kubectl apply -f win-webserver.yaml
kubectl get pods -o wide -w
```

正确部署服务后，两个 Pod 都标记为“Ready”。要退出 watch 命令，请按 Ctrl + C。

3. 检查部署是否成功。验证：

   - Windows 节点上每个 Pod 有两个容器，使用 `docker ps`
   - Linux 控制平面节点列出两个 Pod，使用 `kubectl get pods`
   - 跨网络的节点到 Pod 通信，从 Linux 控制平面节点 `curl` 您的 pod IPs 的端口80，以检查 Web 服务器响应
   - Pod 到 Pod 的通信，使用 docker exec 或 kubectl exec 在 Pod 之间 （以及跨主机，如果你有多个 Windows 节点）进行 ping 操作
   - 服务到 Pod 的通信，从 Linux 控制平面节点和各个 Pod 中 `curl` 虚拟服务 IP （在 `kubectl get services` 下可见）
   - 服务发现，使用 Kubernetes `curl` 服务名称 默认 DNS 后缀
   - 入站连接，从 Linux 控制平面节点或集群外部的计算机 `curl` NodePort
   - 出站连接，使用 kubectl exec 从 Pod 内部 curl 外部 IP

说明： 由于当前平台对 Windows 网络堆栈的限制，Windows 容器主机无法访问在其上调度的服务的 IP。只有 Windows pods 才能访问服务 IP。

可观测性

抓取来自工作负载的日志

日志是可观测性的重要一环；使用日志用户可以获得对负载运行状况的洞察， 因而日志是故障排查的一个重要手法。 因为 Windows 容器中的 Windows 容器和负载与 Linux 容器的行为不同， 用户很难收集日志，因此运行状态的可见性很受限。 例如，Windows 工作负载通常被配置为将日志输出到 Windows 事件跟踪 （Event Tracing for Windows，ETW），或者将日志条目推送到应用的事件日志中。 LogMonitor 是 Microsoft 提供的一个开源工具，是监视 Windows 容器中所配置的日志源 的推荐方式。 LogMonitor 支持监视时间日志、ETW 提供者模块以及自定义的应用日志， 并使用管道的方式将其输出到标准输出（stdout），以便 `kubectl logs <pod>` 这类命令能够读取这些数据。

请遵照 LogMonitor GitHub 页面上的指令，将其可执行文件和配置文件复制到 你的所有容器中，并为其添加必要的入口点（Entrypoint），以便 LogMonitor 能够将你的日志输出推送到标准输出（stdout）。

使用可配置的容器用户名

从 Kubernetes v1.16 开始，可以为 Windows 容器配置与其镜像默认值不同的用户名 来运行其入口点和进程。 此能力的实现方式和 Linux 容器有些不同。 在此处 可了解更多信息。

使用组托管服务帐户管理工作负载身份

从 Kubernetes v1.14 开始，可以将 Windows 容器工作负载配置为使用组托管服务帐户（GMSA）。 组托管服务帐户是 Active Directory 帐户的一种特定类型，它提供自动密码管理， 简化的服务主体名称（SPN）管理以及将管理委派给跨多台服务器的其他管理员的功能。 配置了 GMSA 的容器可以访问外部 Active Directory 域资源，同时携带通过 GMSA 配置的身份。 在此处了解有关为 Windows 容器配置和使用 GMSA 的更多信息。

污点和容忍度

目前，用户需要将 Linux 和 Windows 工作负载运行在各自特定的操作系统的节点上， 因而需要结合使用污点和节点选择算符。 这可能仅给 Windows 用户造成不便。 推荐的方法概述如下，其主要目标之一是该方法不应破坏与现有 Linux 工作负载的兼容性。

确保特定操作系统的工作负载落在适当的容器主机上

用户可以使用污点和容忍度确保 Windows 容器可以调度在适当的主机上。目前所有 Kubernetes 节点都具有以下默认标签：

- kubernetes.io/os = [windows|linux]
- kubernetes.io/arch = [amd64|arm64|...]

如果 Pod 规范未指定诸如 `"kubernetes.io/os": windows` 之类的 nodeSelector，则该 Pod 可能会被调度到任何主机（Windows 或 Linux）上。 这是有问题的，因为 Windows 容器只能在 Windows 上运行，而 Linux 容器只能在 Linux 上运行。 最佳实践是使用 nodeSelector。

但是，我们了解到，在许多情况下，用户都有既存的大量的 Linux 容器部署，以及一个现成的配置生态系统， 例如社区 Helm charts，以及程序化 Pod 生成案例，例如 Operators。 在这些情况下，您可能会不愿意更改配置添加 nodeSelector。替代方法是使用污点。 由于 kubelet 可以在注册期间设置污点，因此可以轻松修改它，使其仅在 Windows 上运行时自动添加污点。

例如： `--register-with-taints='os=windows:NoSchedule'`

向所有 Windows 节点添加污点后，Kubernetes 将不会在它们上调度任何负载（包括现有的 Linux Pod）。 为了使某 Windows Pod 调度到 Windows 节点上，该 Pod 需要 nodeSelector 和合适的匹配的容忍度设置来选择 Windows，

```yaml
nodeSelector:
  kubernetes.io/os: windows
  node.kubernetes.io/windows-build: '10.0.17763'
tolerations:
  - key: "os"
    operator: "Equal"
    value: "windows"
    effect: "NoSchedule"
```

处理同一集群中的多个 Windows 版本

每个 Pod 使用的 Windows Server 版本必须与该节点的 Windows Server 版本相匹配。 如果要在同一集群中使用多个 Windows Server 版本，则应该设置其他节点标签和 nodeSelector。

Kubernetes 1.17 自动添加了一个新标签 `node.kubernetes.io/windows-build` 来简化此操作。 如果您运行的是旧版本，则建议手动将此标签添加到 Windows 节点。

此标签反映了需要兼容的 Windows 主要、次要和内部版本号。以下是当前每个 Windows Server 版本使用的值。

产品名称	内部编号
Windows Server 2019	10.0.17763
Windows Server version 1809	10.0.17763
Windows Server version 1903	10.0.18362

使用 RuntimeClass 简化

RuntimeClass 可用于 简化使用污点和容忍度的过程。 集群管理员可以创建 `RuntimeClass` 对象，用于封装这些污点和容忍度。

1. 将此文件保存到 `runtimeClasses.yml` 文件。 它包括适用于 Windows 操作系统、体系结构和版本的 `nodeSelector` 。

```yaml
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: windows-2019
handler: 'docker'
scheduling:
  nodeSelector:
    kubernetes.io/os: 'windows'
    kubernetes.io/arch: 'amd64'
    node.kubernetes.io/windows-build: '10.0.17763'
  tolerations:
  - effect: NoSchedule
    key: os
    operator: Equal
    value: "windows"
```

2. 集群管理员执行 `kubectl create -f runtimeClasses.yml` 操作
3. 根据需要向 Pod 规约中添加 `runtimeClassName: windows-2019`

例如：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2019
  labels:
    app: iis-2019
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2019
      labels:
        app: iis-2019
    spec:
      runtimeClassName: windows-2019
      containers:
      - name: iis
        image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
        resources:
          limits:
            cpu: 1
            memory: 800Mi
          requests:
            cpu: .1
            memory: 300Mi
        ports:
          - containerPort: 80
```

```
 selector:
    matchLabels:
      app: iis-2019
---
apiVersion: v1
kind: Service
metadata:
  name: iis
spec:
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 80
  selector:
    app: iis-2019
```