第 3 章

CHAPTER 3

企业级容器编排技术 Kubernetes

3.1 Kubernetes 介绍

随着微服务架构和容器虚拟化技术在企业环境中的深入应用，这种架构风格与技术路线不仅改变了已有应用程序的构建和部署方式，还对企业组织结构与技术关系产生了深远影响。在当前企业技术快速变化的时代，企业面临着前所未有的挑战和机遇，为了保持企业的竞争力，就要不断地优化其 IT 基础设施，以确保应用的高可用性、可扩展性和敏捷性。而 Kubernetes 作为谷歌开源的容器编排平台，逐渐成为行业标准，这也促使更多的企业拥抱 Kubernetes。

1. 企业为什么使用 Kubernetes

首先，Kubernetes 提供了一种高效的容器编排和管理机制，为企业应用部署带来了前所未有的高效性与便捷性。它通过自动化的方式，精准地处理应用容器的部署、扩展、负载均衡等一系列复杂任务，从而极大地减轻了企业在应用管理方面的负担。这就使企业可以更聚焦在应用的开发上，进一步提升开发效率，降低管理成本，为业务的快速发展奠定了坚实的基础。

其次，Kubernetes 在可伸缩性和容错性方面展现出了卓越的能力。它能够根据企业业务的变化，动态地调整资源分配，满足各种负载需求。更重要的是 Kubernetes 内置了多种容错机制，例如自动替换故障节点、数据备份与恢复等，确保系统在任何情况下都能保持高度的稳定性和高可用性。这种强大的容错能力，为企业业务的连续运行提供了有力保障。

此外，Kubernetes 还具备出色的跨平台兼容性。无论是公有云、私有云还是混合云环境，Kubernetes 都能提供一致性的管理和部署体验。这一特性为企业实现跨平台的无缝迁移和集成提供了极大的便利，有助于企业灵活应对市场变化，快速调整业务布局。

最后，Kubernetes 拥有非常庞大的社区支持和丰富的生态系统，这意味着企业在使用 Kubernetes 的过程中，可以轻松地获取来自全球各地的专业支持和技术资源。无论是遇到技术难题，还是希望优化、丰富和改进企业的 IT 基础架构，Kubernetes 社区都能提供强有力的支持。这种强大的社区力量，无疑为企业的发展注入了新的活力。

2. Kubernetes 与 Docker Swarm 的区别

Kubernetes 与 Docker Swarm 都是业界知名的容器编排工具,尽管都致力于提升容器化应用的部署与管理效率,但它们在多个维度上展现出了显著的差异,这些差异深刻地影响着用户的选型决策。

首先,从起源与社区支持的角度来看,Kubernetes 由科技巨头谷歌孕育并慷慨捐赠给云原生计算基金会(CNCF),这一背景赋予了它强大的技术底蕴与广泛的社区基础。相比之下,Docker Swarm 作为 Docker 公司自家的容器编排解决方案,虽然同样拥有一定的市场份额,但在社区活跃度和生态系统丰富度上,显然无法与 Kubernetes 相提并论。Kubernetes 的开源特性吸引了全球范围内的开发者、企业及解决方案提供商的积极参与,形成了一个庞大的知识库与技术支持网络,这对于解决复杂问题、快速迭代升级具有不可估量的价值。

其次,在功能特性与可扩展性方面,Kubernetes 展现出了无与伦比的优势。它不仅支持自动扩缩容、智能负载均衡、动态服务发现等核心功能,还提供了详尽的管理配置、存储编排及强大的安全机制,这些特性使其成为构建复杂微服务架构的理想选择。相比之下,Docker Swarm 虽然也具备基本的容器编排与管理能力,但在面对大型、分布式应用部署时,其功能集显得相对单薄,难以满足高度定制化、高可用性的需求。

在配置管理方面,两者虽都采用了声明式配置方式,但具体实现上却大相径庭。Docker Swarm 依赖于 docker-compose. yaml 文件来定义服务、网络及存储资源,这种方式直观且易于上手,适合小规模或单项目环境,而 Kubernetes 则通过一系列 YAML 文件来精确描述 Pod、服务、存储卷等资源的状态与关系,这种高度模块化的设计使系统更加灵活、易于维护,尤其适合大型、多组件应用的复杂场景。

最后,在兼容性与集成能力上,Kubernetes 凭借其设计之初就确立的开放性与中立性,能够轻松地与多种容器运行时集成,包括但不限于 Docker、Containerd 等,这为企业在容器技术选型上提供了极大的自由度与灵活性,而 Docker Swarm 则与 Docker 深度绑定,虽然这种紧密集成在某些场景下简化了部署流程,但也限制了其跨平台、跨技术的灵活性。

3. Kubernetes 的功能特点

Kubernetes 之所以成为事实上的行业标准,并被企业广泛使用是由其自身的架构特点、优良的生态环境、独特的功能特点等众多因素所决定的,其功能特点主要表现如下。

1) 自动化容器编排和调度

Kubernetes 可以自动管理容器的创建、销毁和调度,它会根据资源需求和约束条件将容器分配到合理的可用节点,以实现资源的高效利用和负载均衡。

2) 弹性伸缩

Kubernetes 可以根据应用程序的负载情况,自动地对应用进行水平扩展和收缩,以满足应用的实际需求。

3) IPv4/IPv6 双协议栈

Kubernetes 集群中的节点和 Pod 可以具有双栈(IPv4 和 IPv6)地址。这意味着节点和

Pod 可以同时具有 IPv4 和 IPv6 地址，并且可以使用这两种地址进行通信。

4）服务发现和负载均衡

Kubernetes 提供了内建的服务发现机制，可以自动地为应用程序创建服务，并为每个服务分配唯一的 DNS 名称，用于集群内部应用程序间的访问。

5）自我修复

Kubernetes 可以监控容器的健康状态，并在容器失败时自动进行恢复。它可以根据设置的规则，自动重启容器实例或将容器迁移到其他可用的节点上，以确保应用程序的持续可用性。

6）滚动升级和回滚

Kubernetes 可以对应用程序进行滚动升级，即根据设置的策略，逐步启动新版本的容器实例，并在新版本验证通过后，再逐步停止旧版本的容器实例。如果升级失败，Kubernetes 则可以自动回滚到之前的版本。

7）配置和存储管理

Kubernetes 可以为容器提供环境变量、配置文件和密钥等配置信息，并将其注入容器内部。它还可以为容器提供持久化存储卷，以保存应用程序的数据，例如 NFS、iSCSI、Ceph、Cinder 等网络存储系统。

8）多租户支持

Kubernetes 支持多租户架构，可以将集群划分为多个逻辑区域，每个逻辑区域可以由不同的团队或用户独立管理。

9）安全性和权限控制

Kubernetes 提供了多层次的安全性和权限控制机制，以保护容器和集群的安全。它可以为每个容器分配独立的用户和组身份，限制容器的权限和访问范围。它还可以为集群提供身份验证和授权机制，以控制用户对集群资源的访问权限。

10）多云和混合云支持

Kubernetes 可以在不同的云平台上运行，包括公有云、私有云和混合云环境。它可以通过提供统一的 API 和管理界面，简化跨云平台的应用程序部署和管理。

11）社区支持和生态系统

Kubernetes 拥有一个活跃的开源社区，提供了丰富的文档、教程和示例代码。它还有一个庞大的生态系统，包括第三方工具和服务，可以扩展和增强 Kubernetes 的功能。

4. Kubernetes 的主要应用场景

Kubernetes 最初是为了解决云原生应用程序的部署和管理问题而开发的，因此它天然具备强大的工具和工作机制用于自动化部署、管理和扩展云原生应用程序。它可以管理大规模集群中数千个甚至数十万个容器实例，并且能够根据集群的资源和负载情况，自动调度和平衡实例对资源的需求，以实现最优的资源利用和最佳的性能。同时还可以在达到预设的阈值时自动触发并完成容器实例的扩展或缩减，以满足应用服务的实际需求。

随着云计算技术在企业内的广泛应用，Kubernetes 支持在不同的平台上运行，它通过

提供统一的 API 和管理界面,满足企业在公有云、私有云和混合云的多场景下跨平台部署和管理应用。尤其是随着开发和运行维护(Development-Operations,DevOps)理念在企业内部的应用,对持续集成和持续交付的需求变得更为强烈,而 Kubernetes 则可以完美地与持续集成和持续交付工具进行集成,实现了代码获取、代码封装、镜像构建、应用发布等全过程的自动化管理,同时还可以根据预设的方案进行滚动升级和回滚,以实现持续交付和快速迭代的目标。

这些只是 Kubernetes 应用场景中的冰山一角,它还在诸多场景中发挥着重要作用,例如 Kubernetes 还可以通过在边缘服务节点上部署和管理容器实例来实现低延迟和高可用性的边缘计算和物联网应用需求等。相信随着容器化技术的进一步普及和发展,Kubernetes 的应用场景会更加丰富,这就需要不断地探索和实践。

3.1.1 Kubernetes 发展

Kubernetes 作为一个开源的容器编排平台,通常被简称为 K8s。它最早可以追溯到谷歌内部用于管理和编排容器化工作负载的 Borg 系统,该系统是谷歌在 2003 年开始开发的一个内部项目,主要是为数百万个容器化应用提供自动化的容器部署、资源调度、服务发现和健康检查等功能,其目的是简化谷歌内部应用程序开发和部署流程。

2014 年 Docker 容器技术正在兴起,在当时技术发展的趋势下,谷歌将其内部使用的容器编排系统 Borg 的经验、技术、理念等相关概念开放给更为广泛的用户和开发者社区,Kubernetes 项目正式诞生。同年 6 月 6 日 Kubernetes 项目第 1 次提交并推送至 GitHub,并迅速引起了业界的广泛关注和认可,它的发布标志着容器编排领域的一个重要里程碑,使容器化应用的部署、扩展和管理变得更加简单和高效,同时也为开发者和企业提供了更好的开发和运维体验。在接下来的一年时间内,一个由主要来自谷歌和红帽(RedHat)等公司的贡献者组成的小型社区的成员经过努力最终在 2015 年 7 月 21 日发布了 1.0 版本。与此同时谷歌宣布将 Kubernetes 捐赠给 Linux 基金会下的一个新成立的分支云原生基金会(Cloud Native Computing Foundation,CNCF)。

自此以后,随着社区的不断扩大,吸引了全球各地的开发者参与其中,这使 Kubernetes 逐渐成为行业事实上的标准,并且发布了一系列具有代表性、里程碑的版本。例如 2016 年 12 月发布的 Kubernetes 1.5 引入了运行时可插拔性,OpenAPI 也首次出现,为客户端能够发现扩展 API 铺平了道路。2018 年 12 月发布的 Kubernetes 1.13 版本中,容器存储接口、用于部署集群的 kubeadm 工具完全稳定可用,并且 CoreDNS 成为集群内部默认的 DNS 服务器。2020 年 12 月发布的 Kubernetes 1.20 版本中 Dockershim 被弃用。在随后的 2022 年 5 月发布的 Kubernetes 1.24 版本中彻底移除 Dockershim。2023 年 4 月发布的 Kubernetes 1.27 版本中将 k8s.gcr.io 重定向到 registry.k8s.io,默认启用安全计算模型(Secure Computing Mode)提高 Pod 容器的安全性等诸多功能,当前最新版本为 Kubernetes 1.31 于 2024 年 9 月发布。相信随着云计算技术的进一步发展和 Kubernetes 生态系统的不断发展壮大,Kubernetes 会以更加简化的用户体验、更强大的扩展性、更强大的生态系统和更全

面的安全性服务更多企业项目。

3.1.2　Kubernetes 架构与核心概念

Kubernetes 的架构设计旨在提供一个高效、可扩展和可靠的容器编排平台。首先，通过自动化容器操作让容器部署、扩展、管理和运维变得更智能，尽量减少在这一过程中人工干预的因素，提升开发效率和系统可靠性。其次，通过精细资源调度和分配提升集群资源利用率。同时，通过自动扩展和收缩以应对不同的工作负载需求，进一步提升应用的高可用性和稳定性。最后，内置服务发现和负载均衡机制，确保服务之间可以正常通信。一个典型的 Kubernetes 集群架构如图 3-1 所示。

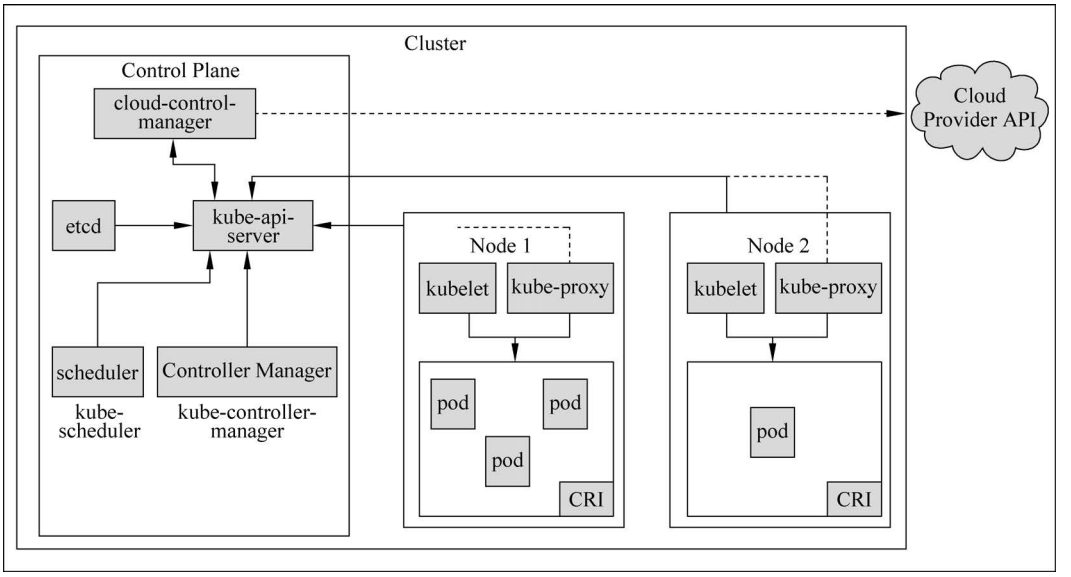

图 3-1　典型的 Kubernetes 集群架构

从架构图中可以清晰地看到一个典型的 Kubernetes 架构包含三部分，分别是控制平面(Control Plane)、节点(Node)、插件(例如网络插件等)，这些相关组件的有机协调，支撑了 Kubernetes 集群可以高效、稳定地工作。

1. 控制平面

控制平面是集群的全局决策者，例如资源调度、集群监测及集群事件的响应等。在默认情况下控制平面的各组件运行在同一台服务器节点上，控制平面包含的相关组件见表 3-1。

表 3-1　控制平面组件

组 件 名 称	功 能 说 明
kube-api-server	提供 Kubernetes API 的访问接口，用于处理所有的 API 请求，并将请求转发给其他相关组件处理，同时还负责验证和授权请求

续表

组件名称	功能说明
etcd	分布式键值存储系统,用于存储集群的配置数据和状态信息
kube-scheduler	根据相关的调度策略和约束条件,将容器组(Pod)调度至集群内合适的节点。例如,资源需求、软硬件及策略约束、亲和性和反亲和性等调度决策因素
kube-controller-manager	负责管理集群中的各种控制器,例如节点控制器,用于负责在节点出现故障时进行通知和响应等

2. 节点

Kubernetes集群中节点用于提供集群运行环境并负责维护和运行Pod,每个集群至少有一个工作节点来负责运行Pod。每个节点包含以下组件,并且这些组件会运行在每个节点上,相关组件见表3-2。

表3-2 节点组件

组件名称	功能说明
kubelet	负责监控和管理节点上的Pod及其容器正常运行,需要特别注意的是kubelet只管理由Kubernetes创建的容器
kube-proxy	提供网络代理和网络负载功能,它会在每个节点上维护一个网络规则集,用于确保Pod之间可以互相通信并且可以从集群内部或外部访问应用服务
容器运行时(Container Runtime)	负责运行容器的软件,例如Docker、Containerd、CRI-O等

3. 插件

在Kubernetes集群中,通过插件扩展了集群的功能,并且非常多的插件是属于集群级别的,典型的插件见表3-3。

表3-3 插件

插件名称	功能说明
DNS插件	用于提供集群范围内的DNS解析
仪表盘(Dashboard)	Kubernetes的Web界面,用于集群可视化管理,例如查看和监控集群状态、创建和管理Pod、服务等
网络插件	用于提供网络连接和通信,典型的网络插件有Calico、Fannel、Weave Net等
存储插件	提供数据持久化存储,典型的存储插件有Cinder、NFS、Ceph RBD等

4. 核心概念

Kubernetes的核心概念是构建在容器虚拟化应用程序基础之上的,主要包含以下概念。

1) 节点(Node)

通常是指Kubernetes集群中的工作节点,它可以是物理机或虚拟机,用于运行容器化应用程序。每个节点都有属于自己单独的计算资源、存储资源、网络资源和运行的容器环境,并且运行用于和管理节点进行通信的kubelet进程,该进程用于Pod生命周的管理。

2）管理节点(Master)

管理节点负责管理和调度集群中的所有资源，通常情况下用于部署控制平面组件。例如 API Server 提供了与集群通信的接口，Controller Manager 负责管理集群中的各种控制器，Scheduler 负责将 Pod 调度到合适的节点上运行，etcd 是一个分布式键值存储系统，用于存储集群的状态信息。

3）容器组(Pod)

Pod 是 Kubernetes 集群中最小的可部署单元，它是由一个或多个容器组成的，这些容器共享相同的网络和存储资源。首先 Pod 是通过名称空间进行隔离的，其次 Pod 有自己的运行状态，见表 3-4，掌握并熟悉 Pod 的状态至关重要，直接关系到后续相关内容的深入学习。

表 3-4　Pod 状态

状　　态	说　　明
挂起(Pending)	表示 Pod 已经被 Kubernetes 系统接管，但是有一个或多个容器既未创建，也未运行。该阶段包含等待 Pod 被调度的时间和镜像通过网络下载的时间
运行中(Running)	表示 Pod 已经被绑定到某个节点，Pod 中所有的容器都已经被创建。至少有一个容器仍在运行，或者正处于启动或重启状态
成功(Succeeded)	表示 Pod 中的所有容器已经成功终止，并且不会再重启
失败(Failed)	表示 Pod 中的所有容器都已终止，并且至少有一个容器是因为失败终止的，即容器以非 0 状态退出或者被系统终止
未知(Unknown)	因为某些原因无法获取 Pod 状态。这种情况通常是因为与 Pod 所在主机通信失败所造成的

注意：当 Pod 反复重启失败时，使用 kubectl 命令在获取 Pod 信息时 status 字段有可能会显示 CrashLoopBackOff。同样，当 Pod 在删除时，使用 kubectl 命令在获取 Pod 信息时 status 字段有可能会显示 Terminating。

在 Kubernetes 集群中定义 Pod 的 YAML 文件的示例代码如下：

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

4）容器状态

当集群中的 Pod 被创建后，Kubernetes 会自动跟踪 Pod 中每个容器的状态。一旦 Pod 被调度至合适的节点，节点上的 kubelet 进程就会通过容器运行时开始为 Pod 创建容器，此

时容器的状态见表 3-5。

表 3-5 容器状态

状 态	说 明
Waiting	表示容器正在运行它完全启动所需的操作,例如,从某个镜像仓库拉取容器镜像等
Running	表示容器正在运行,并且没有问题发生
Terminated	表示容器已经开始执行或者正常结束或者因为某些原因失败

5) 命名空间(Namespace)

命名空间用于对集群中的资源进行逻辑隔离,通过这种方式可以将集群划分为多个虚拟集群,每个集群又拥有专属的资源配额和访问控制策略。

6) 服务(Service)

服务是对一组 Pod 的抽象,它定义了访问 Pod 的策略。服务可以提供一个稳定的网络地址和端口,使其他应用程序可以通过该地址和端口访问 Pod,同时还具有负载均衡、服务发现和故障恢复等功能。在 Kubernetes 集群中服务类型见表 3-6。

表 3-6 服务类型

类 型	说 明
ClusterIP	用于将服务暴露在集群内部,它会为服务分配一个虚拟的 ClusterIP 地址,允许集群内的其他 Pod 中使用该地址来访问服务。 (1) 内部服务暴露:ClusterIP 允许在集群内部将服务暴露给其他 Pod,这些 Pod 可以使用该地址来访问服务,这样可以方便地在集群内部实现服务之间的通信。 (2) 负载均衡:ClusterIP 会自动在集群内部进行负载均衡,将请求分发到后端 Pod 上,这样可以确保服务的高可用性和性能。 (3) 自动服务发现:Kubernetes 会为每个 Service 创建一个 DNS 记录,该记录将 ClusterIP 与服务名称关联起来。这样其他 Pod 可以通过服务名称来解析 ClusterIP 地址,而无须手动配置 IP 地址。 (4) 无须暴露到外部:ClusterIP 只在集群内部可访问,不会暴露到集群外部,确保只有集群内部的 Pod 可以访问服务
NodePort	它允许将集群内部的服务暴露到集群外部的节点上。 (1) 外部访问:NodePort 允许外部用户通过节点的 IP 地址和指定的端口访问集群内部的服务。 (2) 端口映射:NodePort 会将指定的端口映射到服务的端口上,以便外部用户可以通过该端口访问服务,例如,如果将 NodePort 设置为 30 000,那么外部用户可以通过节点的 IP 地址和端口 30 000 访问服务。 (3) 负载均衡:NodePort 可以与 Kubernetes 的负载均衡器结合使用,以便将请求分发到集群中的多个节点上,从而实现了高可用性和负载均衡。 (4) 安全性:NodePort 可以通过网络策略来限制访问,例如,可以配置网络策略以允许或禁止特定的 IP 地址或 IP 范围访问 NodePort。 (5) 端口范围:NodePort 使用的端口范围是 30 000～32 767,在默认情况下,Kubernetes 会自动分配一个未使用的端口作为 NodePort

续表

类　型	说　明
LoadBalancer	(1) 自动负载均衡：LoadBalancer 会自动地将流量分发到集群中的多个 Pod 实例上，以实现负载均衡。它可以根据每个 Pod 的资源使用情况来动态地分配流量，确保每个 Pod 都能够平均地处理请求。 (2) 外部访问：LoadBalancer 可以将集群内部的服务暴露给外部网络，使外部用户可以通过公共网络访问集群中的服务。它会为服务分配一个公共 IP 地址，并将流量从该 IP 地址转发到服务的 Pod 实例上。 (3) 高可用性：LoadBalancer 会监控集群中的 Pod 实例，并自动检测故障或不可用的实例。当发现故障时，它会将流量重新分发到其他可用的实例上，以确保服务的高可用性。 (4) 灵活配置：LoadBalancer 可以根据需要进行灵活配置。管理员可以指定负载均衡算法、会话保持策略、健康检查参数等，以满足不同的需求。 (5) 与云平台集成：Kubernetes 的 LoadBalancer 可以与各种云平台进行集成，以利用云平台提供的负载均衡服务。它可以与云平台的负载均衡器进行通信，以便动态地调整负载均衡策略和配置

在 Kubernetes 集群中定义 service 的 YAML 文件的示例代码如下：

```
apiVersion: v1
kind: Service
metadata:
  name: my-clusterip
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
     port: 80
     targetPort: 8080
  type: ClusterIP
```

7) 副本数(ReplicaSet)与标签(Label)

副本数是用来设置 Pod 在 Kubernetes 中运行的数量，在 Pod 发生故障或被删除时，ReplicaSet 会自动创建新的 Pod 副本数来替代故障 Pod。标签是 Kubernetes 中对资源进行标记和分类的一种方式，采用键-值对的形式定义。在 Kubernetes 集群中定义副本数、定义标签的 YAML 文件的示例代码如下：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
```

```
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:v5
```

8) 部署(Deployment)

它定义了应用部署的策略,例如应用程序的副本数量、升级策略等参数,YAML 文件的示例代码如下:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

9) 存储卷(Volume)

存储卷是一种抽象的存储设备,通过将存储设备(例如磁盘、网络存储等)挂载到 Pod 中的容器内,实现数据的持久化存储。在 Kubernetes 集群中定义存储卷的 YAML 文件的示例代码如下:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
```

```
containers:
   - name: test
     image: busybox:1.28
     command: ['sh', '-c', 'echo "The app is running!" && tail -f /dev/null']
     volumeMounts:
       - name: config-vol
         mountPath: /etc/config
 volumes:
   - name: config-vol
     configMap:
     name: log-config
     items:
       - key: log_level
         path: log_level
```

10）端口(Port)

Kubernetes 集群在应用与管理过程中的常见端口类型见表 3-7。

表 3-7　端口类型

类　　型	功 能 说 明
节点端口(NodePort)	允许用户从集群外部访问 Kubernetes 服务，当用户创建一个 NodePort 类型的服务时，Kubernetes 会为该服务在每个节点上分配一个端口，并将流量转发到该端口上的服务，端口范围为 30 000～32 767。如果用户没有自定义服务器端口，则 Kubernetes 会在该范围内自动分配端口
目标端口(TargetPort)	Pod 中容器的监听端口，定义了容器中正在运行的应用程序所侦听的端口号。当流量到达 Port 时 Kubernetes 将其转发到 Pod 中的 TargetPort
服务器端口	服务监听端口，用于接受来自集群内部其他服务的流量，它是服务的入口，可以自动地将流量转发到后端 Pod 的 TargetPort
容器端口(ContainerPort)	容器中应用程序实际监听的端口，是容器内应用程序与容器外部之间的通信端口，也是 TargetPort 的一部分

3.1.3　Kubernetes 工作流程

Kubernetes 的工作流程是一个高度自动化和动态的过程，从应用的容器化开始，到配置、部署、运行和管理，形成了一套完整的闭环。它通过声明式 API、自动调度、自愈机制及良好的扩展性，为开发者提供了强大的支持，使容器化应用的管理变得轻松而高效。这一流程不仅提升了应用的可靠性，还大幅降低了运维成本，使 Kubernetes 成为现代云原生应用不可或缺的核心组件，其关键工作流程主要包含以下环节。

1. 应用程序容器虚拟化

在部署应用程序之前，首先需要将应用打包成容器镜像，这一过程的典型方案是基于 Dockerfile 文件构建镜像。镜像构建完成后可以将镜像推送至镜像仓库，以备应用部署时下载调用或者将镜像分享给其他使用者。

2. 定义应用配置

接下来,通过 YAML 文件定义应用部署所需的各类资源,例如设置 Pod 的副本数、更新策略、对外暴露的端口、数据持久化存储及对敏感数据的处理方式等。

3. 将配置提交至 Kubernetes 集群

一旦配置文件准备完成,就可以使用 kubectl 命令将这些配置提交至 Kubernetes 集群。由 Kubernetes 集群中的 API Server 将所有请求转发至其他核心组件,例如调度器等。

4. 任务调度

调度器会根据集群节点的资源情况来决定由哪些节点来运行相关 Pod,以确保其能够合理、高效地运行。

5. 容器启动

节点调度位置一旦确定,节点中的 kubelet 就会在该节点创建 Pod,这一过程涉及依赖镜像的下载,以及容器的启动。

6. 服务发现与负载均衡

当 Pod 启动成功后,Kubernetes 集群会自动为 Pod 创建和配置服务对象,同时集群还会通过负载均衡机制将网络流量均匀地分发到后端 Pod 中,以确保服务的可用性。

7. 状态监控与自愈能力

集群节点中的 kubelet 会定期地向 API Server 报告 Pod 的状态,一旦发现某个 Pod 失败或不健康,Kubernetes 集群控制器就会自动尝试重新创建 Pod,用于替换故障实例,确保应用状态的预期值不变。

8. 扩缩与更新

通过修改应用配置文件中的副本数或者使用 kubectl 命令均可方便地对运行的 Pod 进行扩缩,同时 Kubernetes 集群还支持多种更新策略,例如滚动更新和蓝绿发布,用于满足企业在不同场景下的业务需求。

9. 数据持久化存储

对于 Kubernetes 集群中的数据则可以采用持久卷(Persistent Volume,PV)和持久卷申领(Persistent Volume Claim,PVC)的方式实现,确保数据的完整性和可用性。

10. 监控与日志管理

为了保证集群的健康和高效,Kubernetes 集群通常会集成监控和日志管理系统,用于时时监控集群状态、应用性能和错误日志等。

3.1.4 Kubernetes 典型命令

在 Kubernetes 集群管理过程中命令行工具的运用是必备的技能,典型的命令行工具包括 kubeadm 和 kubectl。

kubeadm 命令用于 Kubernetes 集群初始化及节点管理，典型的 kubeadm 子命令见表 3-8。

表 3-8　kubeadm 子命令

命　令	功 能 说 明
kubeadm init	用于初始化新的 Kubernetes 控制平面节点
kubeadm join	将一个新的节点加入现有的 Kubernetes 集群中
kubeadm upgrade	用于升级 Kubernetes 集群的版本
kubeadm token	用于管理集群的加入令牌
kubeadm reset	用于重置 Kubernetes 集群
kubeadm certs	用于管理 Kubernetes 集群证书
kubeadm version	显示 kubeadm 的版本信息

集群的交互工具 kubectl 命令，它的主要功能是通过与集群的交互执行各种操作，例如应用部署、查看集群状态、管理集群资源等，典型的 kubectl 命令见表 3-9。

表 3-9　kubectl 命令

功　能	命　令	命 令 说 明
获取集群信息	kubectl cluster-info	显示集群的概要信息
	kubectl version	显示客户端和服务器端的版本信息
管理命名空间	kubectl create namespace < namespace >	创建新的命名空间
	kubectl get namespaces	列出集群内的所有命名空间
	kubectl delete namespace < namespace >	删除指定的命名空间
管理 Deployment	kubectl create deployment < name > --image=< image >	创建一个新的 Deployment
	kubectl get deployments	列出当前命名空间中的所有 Deployment
	kubectl scale deployment < name > --replicas=< number >	调整 Deployment 的副本数量
	kubectl delete deployment < name >	删除一个 Deployment
管理 Pod	kubectl get pods	列出当前命名空间中的所有 Pod
	kubectl describe pod < pod-name >	显示指定 Pod 的详细信息
	kubectl logs < pod-name >	查看指定 Pod 的日志
	kubectl exec < pod-name > --command	在 Pod 中执行命令
管理 Service	kubectl create service < type > < name > --tcp=< port >:< targetPort >	创建一个新的 Service
	kubectl get services	列出当前命名空间中的所有 Service
	kubectl delete service < name >	删除指定的 Service

续表

功　能	命　令	命令说明
管理 Ingress	kubectl create ingress < name > --rule = < host > = < service >:< port >	创建一个新的 Ingress 规则
	kubectl get ingresses	获取当前命名空间中的所有 Ingress 规则
	kubectl delete ingress < name >	删除指定的 Ingress 规则
其他常用命令	kubectl apply -f < file >	用于将配置应用于资源
	kubectl config view	查看当前的 kubeconfig 文件
	kubectl cp	用于在主机与 Pod 之间传输文件和目录
	kubectl drain	用于安全地驱离节点上的所有 Pod
	kubectl edit	用于编辑资源配置
	kubectl event	用于获取与资源相关的事件信息
	kubectl taint	用于管理节点上的污点

表中列出的只是 kubectl 的部分命令,更多命令可以通过执行 kubectl --help 命令来获取。

3.1.5　Kubernetes 部署实战：基于 Docker 环境

当前企业环境中采用 Docker 作为 Kubernetes 集群的容器运行时仍然是一个最常见的选择,主要原因是 Docker 在企业内使用广泛,并且它完全符合开放容器标准(Open Container Initiative,OCI)。在这种组合方案中,Docker 主要负责构建、运行和分发容器,而 Kubernetes 则负责调度多个容器在集群内的运行,并为它们提供服务发现、负载均衡、弹性伸缩等高级功能。下面将演示该方案的部署过程,具体如下。

1. 节点资源需求

在部署 Kubernetes 环境过程中,节点资源需要满足一定的条件才可以部署集群,官方推荐的节点资源需求见表 3-10。

表 3-10　节点资源需求

类　型	说　明
操作系统	Linux 操作系统
内存	最小 2GB
CPU	控制节点最少 2vCPU
网络	节点间网络通信正常,若需要基于互联网安装相关依赖包、镜像等,则需要保持访问互联网通畅
交换分区	关闭

续表

类　型	说　明
主机名	唯一
MAC 地址	唯一
product_uuid	唯一

当企业基于安全因素在集群内配置了防火墙，则需要开放以下相关端口，端口开放范围见表 3-11。

表 3-11　端口开放范围

类　型	协　议	数据流方向	端口范围	对应组件	服务范围
管理节点（控制平面）	TCP	入站	6433	Kubernetes API Server	所有组件
	TCP	入站	2379～2380	etcd 服务器客户端 API	kube-apiserver、etcd
	TCP	入站	10 250	kubelet API	自身、管理节点
	TCP	入站	10 257	kube-controller-manager	自身
	TCP	入站	10 259	kube-scheduler	自身
工作节点	TCP	入站	10 250	kubelet API	自身、管理节点
	TCP	入站	10 256	kube-proxy	自身、负载均衡器
	TCP	入站	30 000～32 767	NodePort	所有节点

注意：在演示环境中各节点防火墙处于关闭状态，企业实际生产环境应根据实际情况决定是否开启防火墙。

2. 节点环境准备

首先依据集群配置需求完成集群节点的创建，然后就可以对节点进行相关配置。节点配置的关键步骤和操作命令如下。

1）演示环境集群信息

演示环境采用 3 节点集群，即 1 个管理节点和两个工作节点，集群节点信息见表 3-12。

表 3-12　集群节点信息

主　机　名	IP 地址	说　明
node01	192.168.79.181	管理节点
node02	192.168.79.182	工作节点
node03	192.168.79.183	工作节点

2）检查节点的 Shell 类型

由于演示环境采用的是 Ubuntu 22.04 系统，该系统默认的 Shell 类型为 dash 而非 bash，为了后续运行 bash 脚本，因此建议将其 Shell 类型修改为 bash。

首先查看系统的 Shell 类型，命令如下：

```
ll /bin/sh
```

命令执行的结果如图 3-2 所示。

```
user01@node01:~$ ll /bin/sh
lrwxrwxrwx 1 root root 4 Mar 23  2022 /bin/sh -> dash*
```

图 3-2　查看 Shell 类型

然后在确认本地 Shell 类型为 dash 后，执行命令以便将 Shell 类型修改为 bash，命令如下：

```
sudo dpkg - reconfigure dash
```

命令执行后在提示信息 Use dash as the default system shell (/bin/sh)? [yes/no]后输入 no，然后按 Enter 键即可完成修改，记得再次确认是否修改成功，命令执行的过程如图 3-3 所示。

```
user01@node01:~$ sudo dpkg-reconfigure dash
[sudo] password for user01:
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm
 line 78.)
debconf: falling back to frontend: Readline
Configuring dash
----------------

The system shell is the default command interpreter for shell scripts.

Using dash as the system shell will improve the system's overall performance. It does not alter the shell presented to interactive users.

Use dash as the default system shell (/bin/sh)? [yes/no] no

Removing 'diversion of /bin/sh to /bin/sh.distrib by dash'
Adding 'diversion of /bin/sh to /bin/sh.distrib by bash'
Removing 'diversion of /usr/share/man/man1/sh.1.gz to /usr/share/man/man1/sh.distrib.1.gz by dash'
Adding 'diversion of /usr/share/man/man1/sh.1.gz to /usr/share/man/man1/sh.distrib.1.gz by bash'
user01@node01:~$ ll /bin/sh
lrwxrwxrwx 1 root root 4 Oct  8 06:44 /bin/sh -> bash*
```

图 3-3　将 Shell 类型修改为 bash

3）配置主机名、主机 IP

Kubernetes 集群要求主机名和 IP 地址唯一，以节点 node01 为例演示配置过程。首先配置主机名，命令如下：

```
#配置主机名
sudo hostnamectl  -- static set - hostname node01
#当前连接立即生效
bash
```

然后配置主机 IP，在这一过程中配置 IP 的方式有很多种，推荐使用编辑配置文件的方式设置 IP 地址。Ubuntu 系统中网络配置文件路径为/etc/netplan/00-installer-config.yaml，节点 node01 的网络配置文件的示例代码如下：

```
#This is the network config written by 'subiquity'
network:
  ethernets:
    ens33:
      dhcp4: false
      addresses:
```

```
        - 192.168.79.181/24
      routes:
        - to: default
          via: 192.168.79.2
      nameservers:
        addresses: [114.114.114.114]
    ens34:
      dhcp4: false
      addresses:
      - 192.168.172.181/24
  version: 2
```

当配置文件编辑成功后，保存并退出编辑器，然后执行命令让配置生效，命令如下：

```
sudo netplan apply
```

4）校时与名称解析

Kubernetes 集群内各节点要保持时间一致，在企业生产环境内建议采用计划任务的方式进行校时。对于时间服务器的选择既可以是互联网上的时间服务器，也可以是企业内部的时间服务器。

首先使用 ntpdate 命令进行校时，命令如下：

```
sudo ntpdate time.windows.com
```

然后配置计划任务，命令如下：

```
sudo crontab -e
```

命令执行后会进入编辑界面，例如，设置每天凌晨 1 点校时，代码如下：

```
0 1 * * * /sbin/ntpdate time.windows.com >> /var/log/contab.log 2>&1
```

代码编辑完成后保存并退出，重新重启 cron 服务，命令如下：

```
sudo systemctl restart cron
```

注意：在默认情况下 Ubuntu 系统中计划任务是没有日志文件输出的，需要自定义日志文件的路径。

最后，在集群节点配置文件/etc/hosts 内配置主机名解析，代码如下：

```
192.168.79.181 node01
192.168.79.182 node02
192.168.79.183 node03
```

5）关闭交换分区

交换分区的关闭方法有两种，分别是临时关闭和永久关闭。在实际应用过程中往往将这两种方法结合使用，先使用命令方式临时关闭交换分区，然后在配置文件内修改参数永久

关闭交换分区,但需要特别注意的是如果是永久性关闭,则需要重启操作系统才能生效,因此采用命令和配置文件组合的方式可以有效地减少服务器节点的维护时间,并且在服务器进行到下一个维护周期时再加载配置并使其永久生效。

首先,使用命令暂时关闭交换分区,命令如下:

```
sudo swapoff -a
```

然后,编辑/etc/fstab 配置文件内的 swap 选项,代码如下:

```
/dev/disk/by-uuid/56b62c41-58f7-45a5-beaf-0b9cea1b91fe none swap sw,noauto 0 0
```

配置文件编辑完成后,保存并退出即可,此时交换分区处于关闭状态,除非使用命令再次开启交换分区,否则一直会处于关闭状态。

6) 开启流量转发

集群各节点均要开启流量转发等相关操作,首先编辑/etc/modules-load.d/k8s.conf 配置,命令如下:

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

然后加载配置文件/etc/modules-load.d/k8s.conf 中定义的参数,命令如下:

```
sudo modprobe overlay
sudo modprobe br_netfilter
```

接着,在配置文件/etc/sysctl.d/k8s.conf 内设置内核参数,命令如下:

```
cat << EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
```

最后,应用内核参数,命令如下:

```
sudo sysctl --system
```

7) 节点支持 IP 虚拟服务器(IP Virtual Server,IPVS)

Kubernetes 集群节点需要 IPVS 的相关功能,首先为各节点安装相关依赖包,命令如下:

```
sudo apt install ipset ipvsadm
```

然后编辑配置文件/etc/modules-load.d/ipvs.modules,代码如下:

```
#!/bin/bash
modprobe -- ip_vs
```

```
modprobe -- ip_vs_rr
modprobe -- ip_vs_wrr
modprobe -- ip_vs_sh
modprobe -- nf_conntrack
```

接着修改配置文件/etc/modules-load.d/ipvs.modules的文件操作权限并加载相关配置，命令如下：

```
sudo chmod 755 /etc/modules-load.d/ipvs.modules && \
sudo bash /etc/modules-load.d/ipvs.modules && \
sudo lsmod | grep -e ip_vs -e nf_conntrack
```

命令执行的状态如图3-4所示。

```
user01@node01:~$ sudo chmod 755 /etc/modules-load.d/ipvs.modules && \
sudo bash /etc/modules-load.d/ipvs.modules && \
sudo lsmod | grep -e ip_vs -e nf_conntrack
ip_vs_sh               16384  0
ip_vs_wrr              16384  0
ip_vs_rr               16384  0
ip_vs                 176128  6 ip_vs_rr,ip_vs_sh,ip_vs_wrr
nf_conntrack_netlink   49152  0
nfnetlink              20480  4 nft_compat,nf_conntrack_netlink,nf_tables
nf_conntrack          172032  7 xt_conntrack,nf_nat,openvswitch,nf_conntrack_netlink,nf_conncount,xt_MASQUERADE,ip_vs
nf_defrag_ipv6         24576  3 nf_conntrack,openvswitch,ip_vs
nf_defrag_ipv4         16384  1 nf_conntrack
libcrc32c              16384  8 nf_conntrack,nf_nat,openvswitch,btrfs,nf_tables,xfs,raid456,ip_vs
```

图3-4　加载IPVS相关配置参数

8）部署Docker Engine环境

相信经过前面的学习和练习，Docker Engine环境的部署应该轻松完成。需要注意的是如果在安装Docker Engine相关依赖包时速度慢或者出现网络访问超时的情况，则可以将官方的软件源地址修改为国内的清华源、中科大源等，下面展示如何使用清华源进行Docker Engine的环境部署。

首先安装依赖包，命令如下：

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
```

其次，信任Docker的GPG公钥并添加仓库，命令如下：

```
#设置权限
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | gpg --dearmor -o /etc/apt/
keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
#修改软件源地址
echo \
  "deb [arch= $(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://mirrors.tuna.tsinghua.edu.cn/docker-ce/linux/ubuntu \
  "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
  tee /etc/apt/sources.list.d/docker.list > /dev/null
```

接着，安装Docker Engine相关依赖包，命令如下：

```
sudo apt - get update
sudo apt - get install docker - ce docker - ce - cli containerd. io docker - buildx - plugin docker -
compose - plugin
```

最后,也是非常关键的一步,启动、检查并设置 Docker 服务自启动,命令如下:

```
sudo systemctl enable docker
sudo systemctl start docker
sudo systemctl status docker
sudo docker version
```

9) 部署 cri-dockered

首先在各节点下载 cri-dockered 最新稳定版,命令如下:

```
wget https://github. com/Mirantis/cri - dockerd/releases/download/v0. 3. 15/cri - dockerd_0. 3.
15. 3 - 0. Ubuntu - jammy_amd64. deb
```

其次安装 cri-dockered,命令如下:

```
sudo dpkg - i cri - dockerd_0. 3. 15. 3 - 0. Ubuntu - jammy_amd64. deb
```

由于 cri-docker 服务运行时需要用到二进制工具包 conntrack,因此需要手动安装该工具包,命令如下:

```
sudo apt - get install conntrack
```

然后启动 cri-docker 服务,并设置服务自启动完成 cri-docker 服务的安装与配置,命令如下:

```
sudo systemctl start cri - docker
sudo systemctl enable cri - docker
sudo systemctl status cri - docker
```

接着修改 cri-docker 服务的配置文件/usr/lib/systemd/system/cri-docker. service,将 sandbox 镜像源修改为国内源,代码如下:

```
[Service]
Type = notify
# ExecStart = /usr/bin/cri - dockerd -- container - runtime - endpoint fd://
ExecStart = /usr/bin/cri - dockerd -- pod - infra - container - image = registry. cn - hangzhou.
aliyuncs. com/google_containers/pause:3. 10 -- container - runtime - endpoint fd://
ExecReload = /bin/kill - s HUP $ MAINPID
TimeoutSec = 0
RestartSec = 2
Restart = always
```

最后重新加载 cri-docker 服务配置,并重启服务,命令如下:

```
sudo systemctl daemon - reload
sudo systemctl restart cri - docker
sudo systemctl status cri - docker
```

10）安装 Kubernetes 相关组件

集群各节点均需安装 Kubernetes 相关组件，例如 kubeadm、kubelet、kubectl 等，命令如下：

```
#更新索引、安装依赖包
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gpg -y
#下载 Kubernetes 软件包仓库的公共签名密钥
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -
o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
#添加 Kubernetes 仓库
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/
core:/stable:/v1.31/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
#再次更新索引，安装 kubelet、kubeadm 和 kubectl，并锁定其版本
sudo apt-get update
sudo apt-get install kubelet kubeadm kubectl socat
sudo apt-mark hold kubelet kubeadm kubectl
```

注意：Kubernetes 相关软件包安装完成后，kubelet 服务此时会每隔几秒重启一次，因为它在等待 kubeadm 的指令，直到集群初始化完成，集群加入节点。

11）配置 cgroup

将 kubelet 服务的 cgroup 驱动修改为 systemd，使其与容器运行时所使用的 cgroup 驱动保持一致，需要在配置文件/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf 内添加相关配置，代码如下：

```
#编辑配置文件/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf
Environment="KUBELET_CGROUP_ARGS=--cgroup-driver=systemd"
```

配置文件修改完成后需要重启 kubelet 服务，命令如下：

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

3. Kubernetes 集群部署

1）初始化集群

节点的相关配置完成后就可以开始着手初始化 Kubernetes 集群了，首先登录节点 node01 获取初始化集群的默认配置文件，命令如下：

```
sudo kubeadm config print init-defaults > kubeadm-init.yaml
```

其次编辑配置文件 kubeadm-init.yaml，修改相关配置，代码如下：

```
apiVersion: kubeadm.k8s.io/v1beta4
BootstrapTokens:
- groups:
  - system:Bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
```

```
    ttl: 24h0m0s
    usages:
    - signing
    - authentication
  kind: InitConfiguration
  localAPIEndpoint:
    #设置管理节点地址
    advertiseAddress: 192.168.79.181
    bindPort: 6443
  nodeRegistration:
    #设置 criSocket 的路径
    criSocket: unix://var/run/cri-dockerd.sock
    imagePullPolicy: IfNotPresent
    imagePullSerial: true
    #设置节点名称
    name: node01
    taints: null
  timeouts:
    controlPlaneComponentHealthCheck: 4m0s
    discovery: 5m0s
    etcdAPICall: 2m0s
    kubeletHealthCheck: 4m0s
    kubernetesAPICall: 1m0s
    tlsBootstrap: 5m0s
    upgradeManifests: 5m0s
  ---
  apiServer: {}
  apiVersion: kubeadm.k8s.io/v1beta4
  caCertificateValidityPeriod: 87600h0m0s
  certificateValidityPeriod: 8760h0m0s
  certificatesDir: /etc/kubernetes/pki
  clusterName: kubernetes
  controllerManager: {}
  dns: {}
  encryptionAlgorithm: RSA-2048
  etcd:
    local:
      dataDir: /var/lib/etcd
  #修改为国内镜像源
  imageRepository: registry.cn-hangzhou.aliyuncs.com/google_containers
  kind: ClusterConfiguration
  kubernetesVersion: 1.31.0
  networking:
    dnsDomain: cluster.local
    serviceSubnet: 10.96.0.0/12
    #设置 Pod 的子网范围
    podSubnet: 10.244.0.0/16
  proxy: {}
  scheduler: {}
```

然后基于修改后的 kubeadm-init.yaml 文件测试国内镜像源是否可用，命令如下：

```
sudo kubeadm config images list --config kubeadm-init.yaml
```

如果命令执行后如图 3-5 所示，则说明国内镜像源可用。

```
user01@node01:~$ sudo kubeadm config images list --config kubeadm-init.yaml
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-apiserver:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-controller-manager:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-scheduler:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-proxy:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:v1.11.3
registry.cn-hangzhou.aliyuncs.com/google_containers/pause:3.10
registry.cn-hangzhou.aliyuncs.com/google_containers/etcd:3.5.15-0
```

图 3-5　Kubernetes 国内镜像源

最后在 node01 节点执行命令开始初始化集群，命令如下：

```
#提升权限
sudo -s
#基于 kubeadm-init.yaml 文件初始化集群
kubeadm init --config kubeadm-init.yaml
```

初始化命令执行的过程及信息输出如下：

```
user01@node01:~ $ sudo -s
root@node01:/home/user01#kubeadm init --config kubeadm-init.yaml
[init] Using Kubernetes version: v1.31.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local node01] and IPs [10.96.0.1 192.
168.79.181]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost node01] and IPs [192.168.
79.181 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost node01] and IPs [192.168.79.
181 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "super-admin.conf" kubeconfig file
```

```
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/
kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests"
[kubelet-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can
take up to 4m0s
[kubelet-check] The kubelet is healthy after 502.251153ms
[api-check] Waiting for a healthy API server. This can take up to 4m0s
[api-check] The API server is healthy after 11.002503536s
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-
system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node node01 as control-plane by adding the labels: [node-
role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-
balancers]
[mark-control-plane] Marking the node node01 as control-plane by adding the taints [node-
role.kubernetes.io/control-plane:NoSchedule]
[Bootstrap-token] Using token: abcdef.0123456789abcdef
[Bootstrap-token] Configuring Bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order
for nodes to get long term certificate credentials
[Bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[Bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[Bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet
client certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

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
```

```
Alternatively, if you are the root user, you can run:

  export KUBECONFIG = /etc/kubernetes/admin.conf

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

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.79.181:6443 --token abcdef.0123456789abcdef \
      --discovery-token-ca-cert-hash sha256:
a68453d7701e825c480fb84776d0b04dfbe8f2f552088e465421f04cfcdd1414
```

当显示以上初始化成功的信息后，可以根据信息提示执行相关操作，命令如下：

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

2）部署网络插件 Calico

Calico 是目前开源的最成熟、使用最广泛的纯三层网络架构之一，它摆脱了如 flannel 类型插件要求集群节点必须在同一个二层网络的限制，极大地扩展了网络规模和边界。Calico 的工作原理是通过修改集群主机节点上的 iptables 和路由表规则，实现容器间数据通信和访问控制，并通过 etcd 协调节点配置信息，尤其是在基于边界网关协议（Border Gateway Protocol，BGP）下，它能够适应大型网络规模，其架构如图 3-6 所示。

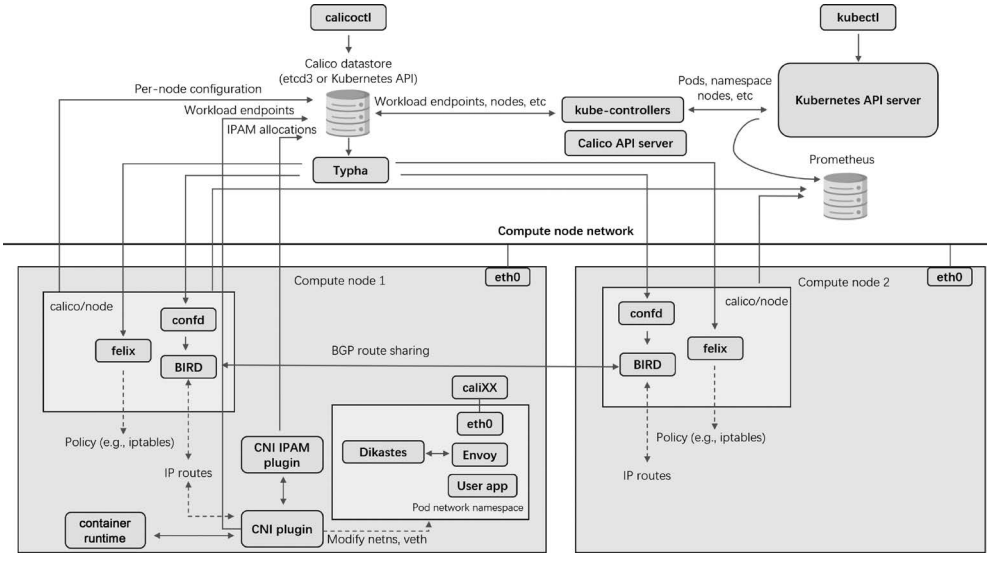

图 3-6　Calico 架构图

架构图内所涉及的 Calico 网络插件的主要组件及其功能，见表 3-13。

表 3-13 Calico 网络插件的主要组件及其功能

组件名称	功能说明
Calico API server	允许用户使用 kubectl 命令管理 Calico 资源
Felix	负责路由配置和 ACL 规则的配置及下发，它运行在所有节点上
BIRD	从 Felix 获取路由，并使用 BGP 协议对各节点的容器网络进行路由交换
confd	监控 BGP 配置和全局默认配置的变化，例如 AS 号、日志记录级别等信息
Dikastes	强制执行 Istio 服务网络策略，作为 Istio Envoy 的 sidecar 代理在集群上运行
CNI plugin	提供 Kubernetes 集群的网络服务
Datastore plugin	通过减少每个节点对数据存储的影响来增加规模
IPAM plugin	基于 Calico 的 IP 资源池分配集群内 Pod IP 地址
kube-controllers	监控 Kubernetes API 并根据集群状态执行相关操作
Typha	通过减少每个节点对数据存储的影响来增加规模，并作为守护进程在数据存储和 Felix 实例之间运行
calicoctl	命令行管理工具，用于创建、读取、更新和删除 Calico 对象

集群内部署 Calico 网络插件时，首先需要下载 Calico 网络插件的配置文件，命令如下：

```
#下载 tigera-operator.yaml 文件
curl -O https://raw.githubusercontent.com/projectcalico/calico/v3.28.2/manifests/tigera-operator.yaml
#下载 custom-resources.yaml 文件
curl -O https://raw.githubusercontent.com/projectcalico/calico/v3.28.2/manifests/custom-resources.yaml
```

然后编辑 custom-resources.yaml 文件，代码如下：

```
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  #Configures Calico networking.
  calicoNetwork:
    #定义 IP 地址池
    ipPools:
    - name: default-ipv4-ippool
      blockSize: 26 #定义子网的大小
      cidr: 10.244.0.0/16 #指定 IP 地址池的 CIDR 范围
      encapsulation: VXLANCrossSubnet #定义跨子网封装模式
      natOutgoing: Enabled #启用网络地址转换(NAT)
      nodeSelector: all() #选择所有节点

---

#This section configures the Calico API server.
```

```
# For more information, see: https://docs.tigera.io/calico/latest/reference/installation/
api#operator.tigera.io/v1.APIServer
apiVersion: operator.tigera.io/v1
kind: APIServer
metadata:
  name: default
spec: {}
```

配置修改完成后保存并退出。

注意：custom-resources.yaml 文件定义的子网范围 10.244.0.0/16 要与初始化集群 kubeadm-init.yaml 文件定义的 Pod 子网范围保持一致。

最后，在 node01 节点部署 Calico 网络插件，命令如下：

```
sudo kubectl create -f tigera-operator.yaml
sudo kubectl create -f custom-resources.yaml
```

命令执行过程的信息输出如下：

```
user01@node01:~$ sudo kubectl create -f tigera-operator.yaml
[sudo] password for user01:
namespace/tigera-operator created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpfilters.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.
org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/apiservers.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/imagesets.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/installations.operator.tigera.io created
customresourcedefinition.apiextensions.k8s.io/tigerastatuses.operator.tigera.io created
serviceaccount/tigera-operator created
clusterrole.rbac.authorization.k8s.io/tigera-operator created
clusterrolebinding.rbac.authorization.k8s.io/tigera-operator created
```

```
deployment.apps/tigera-operator created
user01@node01:~$ sudo kubectl create -f custom-resources.yaml
installation.operator.tigera.io/default created
apiserver.operator.tigera.io/default created
```

此时,可以在 node01 节点查看 Calico 插件的运行情况,命令如下:

```
sudo kubectl get pods -A
```

如果命令执行后的输出信息如图 3-7 所示,则表明 Calico 插件运行成功。

```
user01@node01:~$ sudo kubectl get pods -A
NAMESPACE          NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver   calico-apiserver-d5966b84d-ffxqj           1/1     Running   0          57s
calico-apiserver   calico-apiserver-d5966b84d-mrhtq           1/1     Running   0          57s
calico-system      calico-kube-controllers-5885b45f59-ptxm6   1/1     Running   0          2m51s
calico-system      calico-node-tw6q4                          1/1     Running   0          2m52s
calico-system      calico-typha-77bf8d5c9d-wxblw              1/1     Running   0          2m52s
calico-system      csi-node-driver-kmshg                      2/2     Running   0          2m51s
kube-system        coredns-fcd6c9c4-7sf9r                     1/1     Running   0          44m
kube-system        coredns-fcd6c9c4-k6g8v                     1/1     Running   0          44m
kube-system        etcd-node01                                1/1     Running   0          44m
kube-system        kube-apiserver-node01                      1/1     Running   0          44m
kube-system        kube-controller-manager-node01             1/1     Running   0          44m
kube-system        kube-proxy-z55hz                           1/1     Running   0          44m
kube-system        kube-scheduler-node01                      1/1     Running   0          44m
tigera-operator    tigera-operator-89c775547-f97rf            1/1     Running   0          3m9s
```

图 3-7　管理节点 Calico 插件运行成功

3) 工作节点加入集群

首先复制初始化集群时生成的命令提示并分别在 node02、node03 节点上运行,命令如下:

```
#提升权限
sudo -s
#加入节点
kubeadm join 192.168.79.181:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:a68453d7701e825c480fb84776d0b04dfbe8f2f552088
e465421f04cfcdd1414 \
--cri-socket unix://var/run/cri-dockerd.sock
```

命令执行的过程如图 3-8 所示。

```
root@node02:/home/user01# kubeadm join 192.168.79.181:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:a68453d7701e825c480fb84776d0b04dfbe8f2f552088e465421f04cfcdd1414 \
--cri-socket unix:///var/run/cri-dockerd.sock
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can take up to 4m0s
[kubelet-check] The kubelet is healthy after 1.002541869s
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

图 3-8　节点加入集群

然后在管理节点 node01 上查看节点状态，命令如下：

```
sudo kubectl get nodes
```

当看到集群中节点的状态为 Ready 时，表明集群节点工作正常，如图 3-9 所示。

```
user01@node01:~$ sudo kubectl get nodes
NAME     STATUS   ROLES           AGE     VERSION
node01   Ready    control-plane   57m     v1.31.1
node02   Ready    <none>          2m47s   v1.31.1
node03   Ready    <none>          2m19s   v1.31.1
```

图 3-9　集群节点状态

如果此时再去查看 Calico 插件的 Pod 状态，就会发现 Calico 网络插件会运行在集群中的各个节点，如图 3-10 所示。

```
user01@node01:~$ sudo kubectl get pods -A
NAMESPACE          NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver   calico-apiserver-d5966b84d-ffxqj           1/1     Running   0          23m
calico-apiserver   calico-apiserver-d5966b84d-mrhtq           1/1     Running   0          23m
calico-system      calico-kube-controllers-5885b45f59-ptxm6   1/1     Running   0          25m
calico-system      calico-node-rfc62                          1/1     Running   0          12m
calico-system      calico-node-rnrp9                          1/1     Running   0          11m
calico-system      calico-node-tw6q4                          1/1     Running   0          25m
calico-system      calico-typha-77bf8d5c9d-2t67n              1/1     Running   0          11m
calico-system      calico-typha-77bf8d5c9d-wxblw              1/1     Running   0          25m
calico-system      csi-node-driver-kmshg                      2/2     Running   0          25m
calico-system      csi-node-driver-rmmlv                      2/2     Running   0          12m
calico-system      csi-node-driver-v5q2m                      2/2     Running   0          11m
```

图 3-10　集群中 Calico 插件的运行状态

注意：如果发现某个节点网络连接有问题，则可以首先查看 Calico 网络插件的 Pod 运行状态。

4. 部署仪表盘（Dashboard）

Dashboard 是 Kubernetes 一个基于 Web 的图形化管理界面，用户可以通过这个简单、直观的管理界面轻松地管理和监控 Kubernetes 集群中的资源和应用程序。例如，既可以通过 Dashboard 查看和搜索集群中的各种资源，包括节点、命名空间、Pod、服务、副本集等。还可以通过它创建、编辑和删除资源，例如创建和删除 Pod、服务和副本集等。同时还支持应用程序的扩容、滚动更新和回滚、集群资源和应用资源的监控、用户认证和授权等众多功能。

1）部署 Dashboard

登录管理节点 node01 下载官方提供的 Dashboard 配置文件，命令如下：

```
sudo curl -o dashboard-v2.7.yaml \
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
```

文件下载完成后，只需修改配置文件 dashboard-v2.7.yaml 内服务对外开放的端口，定义服务器端口信息的代码如下：

```
---

kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
      #将对外暴露的端口定义为 31000/tcp
      nodePort: 31000
  selector:
    k8s-app: kubernetes-dashboard
  #将类型指定为 NodePort
  type: NodePort

---
```

配置文件 dashboard-v2.7.yaml 修改完成后保存并退出编辑器,然后执行命令部署 Dashboard,命令如下:

```
sudo kubectl apply -f dashboard-v2.7.yaml
```

命令执行后,可以查看 Dashboard 对应的 Pod 状态、服务状态等,命令如下:

```
#查看 Pod 状态
sudo kubectl get pods -n kubernetes-dashboard
#或者
sudo kubectl get pods -n kubernetes-dashboard -o wide
#查看服务状态
sudo kubectl get svc -n kubernetes-dashboard
#或者
sudo kubectl get svc -n kubernetes-dashboard -o wide
```

Dashboard 运行成功的标识如图 3-11 所示。

```
user01@node01:~$ sudo kubectl get pods -n kubernetes-dashboard
NAME                                         READY   STATUS    RESTARTS   AGE
dashboard-metrics-scraper-6b96ff7878-cn6qh   1/1     Running   0          25m
kubernetes-dashboard-8696f5f494-44ll8        1/1     Running   0          25m
user01@node01:~$ sudo kubectl get svc -n kubernetes-dashboard
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
dashboard-metrics-scraper   ClusterIP   10.98.202.219    <none>        8000/TCP        25m
kubernetes-dashboard        NodePort    10.109.250.249   <none>        443:31000/TCP   25m
user01@node01:~$
user01@node01:~$ sudo kubectl get pods -n kubernetes-dashboard -o wide
NAME                                         READY   STATUS    RESTARTS   AGE   IP               NODE     NOMINATED NODE   READINESS GATES
dashboard-metrics-scraper-6b96ff7878-cn6qh   1/1     Running   0          25m   10.244.186.195   node03   <none>           <none>
kubernetes-dashboard-8696f5f494-44ll8        1/1     Running   0          25m   10.244.186.196   node03   <none>           <none>
user01@node01:~$ sudo kubectl get svc -n kubernetes-dashboard -o wide
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE   SELECTOR
dashboard-metrics-scraper   ClusterIP   10.98.202.219    <none>        8000/TCP        25m   k8s-app=dashboard-metrics-scraper
kubernetes-dashboard        NodePort    10.109.250.249   <none>        443:31000/TCP   25m   k8s-app=kubernetes-dashboard
```

图 3-11 Dashboard 的 Pod 与 Service 状态

此时如果需要访问部署的 Kubernetes Dashboard 控制台，则只需通过浏览器方式访问集群任意节点 IP 的 31000/TCP 端口。

注意：在执行完 Dashboard 部署命令后，如果通过命令查看 Pod 时发现其状态为 ImagePullBackOff，则表明镜像下载失败，在这种情况下需要检查网络环境。

2）登录 Dashboard

登录 Dashboard 可以通过两种方式实现，分别是 Token 或者 kubeconfig。演示环境采用配置 Token 的方式访问 Dashboard，相关步骤如下。

首先配置访问 Dashboard 管理平台的权限文件 dashboard-admin-user. yaml，代码如下：

```
#创建管理账号 admin-user,域名空间为 kubernetes-dashboard
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
---
#创建 ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
---
#创建 admin-user 的 Bearer Token
apiVersion: v1
kind: Secret
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/service-account.name: "admin-user"
type: kubernetes.io/service-account-token
```

然后在集群内发布，命令如下：

```
sudo kubectl apply -f dashboard-admin-user.yaml
```

接着获取 admin-user 访问 Dashboard 管理平台的 Token 值，命令如下：

```
sudo kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath={".data.token"} |
base64 -d
```

命令执行后输出的信息如下：

```
user01@node01:~$ sudo kubectl get secret admin-user -n kubernetes-dashboard -o jsonpath=
{".data.token"} | base64 -d
eyJhbGciOiJSUzI1NiIsImtpZCI6IlAxMERIb28tS3Zwd3pIbmN3OFkxcjFBQmY1a2Nod21PNkM2VVJjelJ4TnMifQ.eyJpc
3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3
BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VjcmV0Lm5
hbWUiOiJhZG1pbi11c2VyIiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZXJ2aWNlLWFjY291bnQubmFt
ZSI6ImFkbWluLXVzZXIiLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiO
iI4MjlmYTJhYy03MzExLTQ3YTMtYjhiMS0xOTgwNmNkNTNhZjgiLCJzdWIiOiJzeXN0ZW06c2VydmljZWFjY291bn
Q6a3ViZXJuZXRlcy1kYXNoYm9hcmQ6YWRtaW4tdXNlciJ9.aW_MXLsTD3kHrd3o-ry_G13SshFiSz2EY33M7pu_
wRsYvNgMUs6APkZmqz4oWes4hs7jZ_Hqo6fKWOJR5Y2SJvCv7qIghHBXtYMELO5AOpD_l-uNUnFbwunQA4aCoAi1
Ggx9A0dZTo-XfFDO6HcxM0_5S8w0NB_Wf0oStmsh8ubmD5Q4E7Ci49kDgmIZm4XLU2c_eRXcQKdqnDjAzC481eg
77SGXJ7gZgC2U0vRDLX6cnkLqYGw_Bp6vHzK4EPiek_zG7HpR3b4NmJQdl3IekVu_M_Oo5HtG2IbrmwruKG2mFPX
W70tlsZ7jvEo_tPJ0wgYuTM6cwRYfu2RRynN9gQ
```

最后，浏览器访问集群内任意节点 IP 的 31000 端口，输入获取的 Token 值，如图 3-12 所示。

图 3-12 Dashboard 登录界面

在输入获取的 Token 值后，单击“登录”按钮即可完成登录。在登录后的界面内单击“工作负载”选项，系统会显示当前集群的工作负载情况，如图 3-13 所示。

Dashboard 的更多功能等待着你慢慢探索。

5. Kubernetes 集群功能验证

集群功能验证的方式有很多种，常见的方式是基于图形化管理工具(例如 Dashboard)、基于命令行模式和 YAML 文件管理应用。对于使用者来讲图形化界面下操作更直观，使用命令行模式时则需要对命令的熟练程度有一定要求。而使用 YAML 文件的方式是最常见的管理方式，但是该方式需要对 Kubernetes 集群中各种资源的参数非常熟悉，此方式的主要优点是后期维护方便。

1) 基于图形化 Dashboard 管理应用

在 Dashboard 管理平台，单击页面右上角的加号(+)标识符创建新资源，在弹出的页面内选择“从表单创建”，如图 3-14 所示。

图 3-13　Dashboard 展示集群工作负载情况

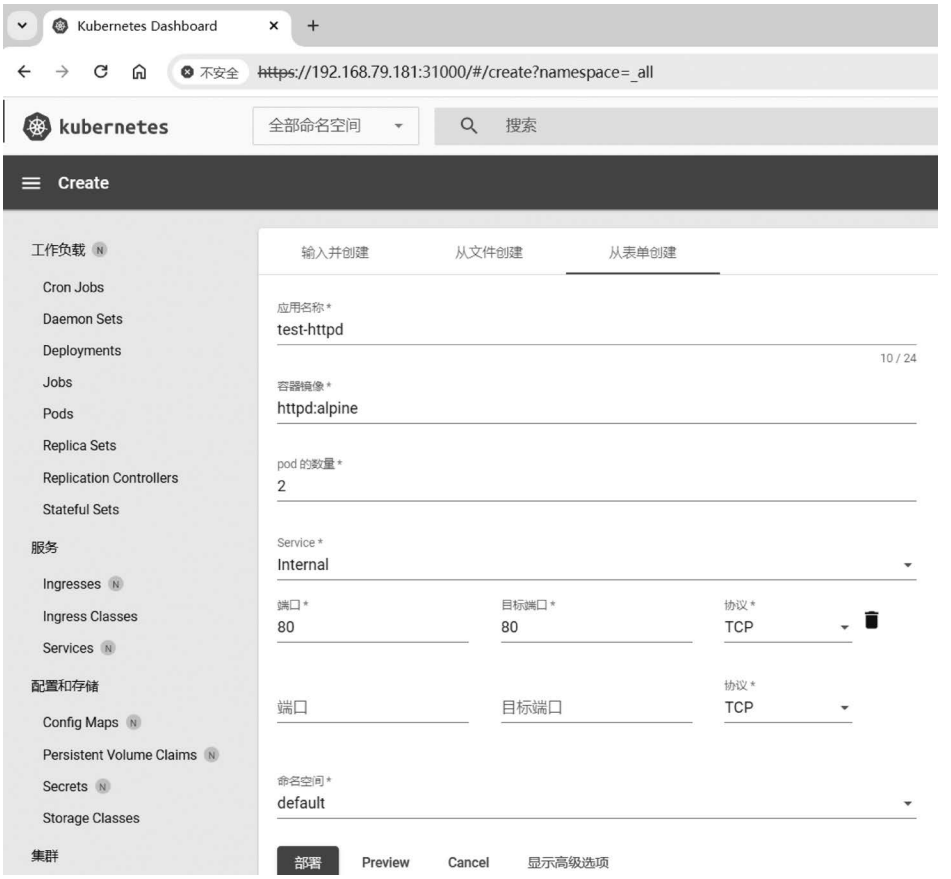

图 3-14　基于 Dashboard 创建资源

注意：图 3-14 内的选项说明如下。

应用名称：test-httpd

容器镜像：httpd:alpine

Pod 数量：2

Service：Internal(表示内网)

命名空间：default

内容填写完成后，单击“部署”按钮后，部署成功的页面如图 3-15 所示。

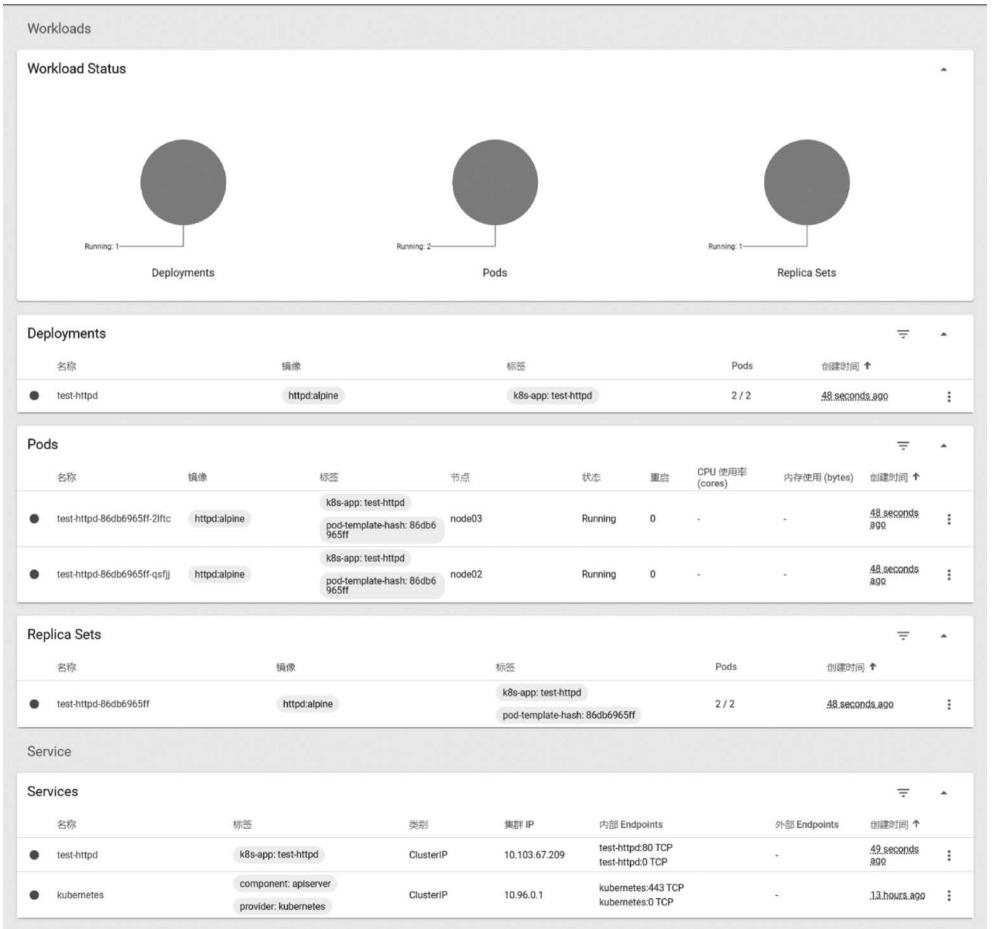

图 3-15　应用部署状态信息

如果需要对资源进行操作，则只需单击所要管理的资源，然后在菜单栏选择操作类型。以刚发布的 test-httpd 示例为例，单击“工作负载”菜单下的 Deployments 选项，在对应的 test-httpd 名称后单击操作选项(3 个竖点)即可选择操作类型，如图 3-16 所示。

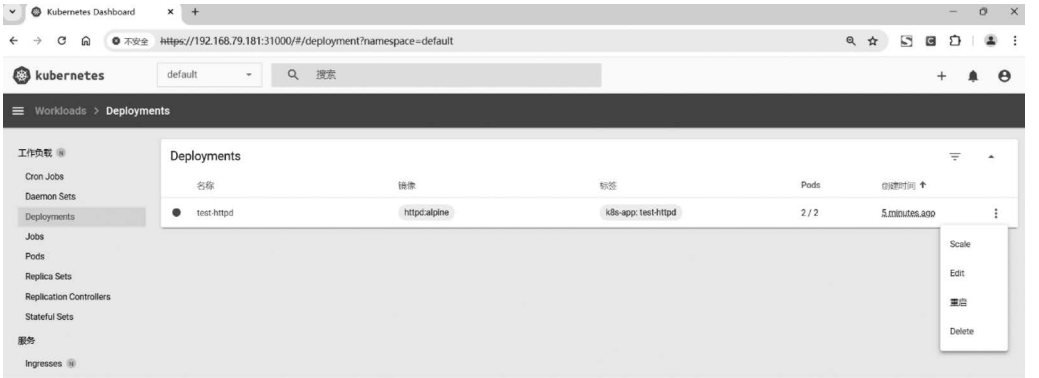

图 3-16　管理 Deployments

以同样的操作方式，单击 Services 选项，可以对服务资源进行编辑等操作，从而实现了 Kubernetes 集群中各种资源的图形化管理，使资源管理变得更简单。

2）基于命令行模式管理应用

在 Kubernetes 实际的运行管理过程中，使用命令行模式是非常普遍及高效的管理方式，例如，创建一个名称为 test-nginx 的 Deployment，镜像为 nginx:alpine，Pod 副本数为 2，命名空间为默认的 default，创建的命令如下：

```
sudo kubectl create deployment test-nginx \
--image=nginx:alpine \
--replicas=2 \
-n default
```

然后就可以创建对应的服务，例如将 NodePort 的端口设置为 30080/TCP，命令如下：

```
sudo kubectl create service nodeport test-nginx --tcp=80:80 --node-port=30080
```

最后，查看创建的 Pod 和 Service 状态，命令如下：

```
#查看 Pod 状态
sudo kubectl get pods -n default
#或者
sudo kubectl get pods
#查 Service 状态
sudo kubectl get svc -n default
#或者
sudo kubectl get svc
```

命令执行的过程如图 3-17 所示。

使用浏览器访问集群任意节点 IP 的 30080/TCP 端口测试其可用性，端口可用的标识如图 3-18 所示。

如果此时需要将 test-nginx 的 Pod 副本数由原来的 2 扩展为 3，则命令如下：

```
user01@node01:~$ sudo kubectl get pods -n default
NAME                          READY   STATUS    RESTARTS   AGE
test-httpd-86db6965ff-7sm86   1/1     Running   0          100m
test-httpd-86db6965ff-rk2p8   1/1     Running   0          100m
test-nginx-7d884887ff-n2bsh   1/1     Running   0          30m
test-nginx-7d884887ff-vz4cf   1/1     Running   0          30m
user01@node01:~$ sudo kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        24h
test-httpd   ClusterIP   10.101.212.115   <none>        80/TCP         100m
test-nginx   NodePort    10.97.135.217    <none>        80:30080/TCP   25s
```

图 3-17　应用 test-nginx 的 Pod 和 Service 状态

图 3-18　测试端口 30080/TCP 可用

```
#查看当前 test-nginx 的副本数
sudo kubectl get deployment test-nginx
#副本数由现在的 2 扩展到 3
sudo kubectl scale --current-replicas=2 --replicas=3 deployment/test-nginx
#确认当前 test-nginx 的副本数
sudo kubectl get deployment test-nginx
```

命令执行的过程如图 3-19 所示。

```
user01@node01:~$ sudo kubectl get deployment test-nginx
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
test-nginx   2/2     2            2           50m
user01@node01:~$ sudo kubectl scale --current-replicas=2 --replicas=3 deployment/test-nginx
deployment.apps/test-nginx scaled
user01@node01:~$ sudo kubectl get deployment test-nginx
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
test-nginx   3/3     3            3           50m
```

图 3-19　Pod 扩展

3）基于 YAML 文件

首先编写测试文件 test-apache.yaml，代码如下：

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-apache
  labels:
    app: test-apache
spec:
```

```
  replicas: 2
  selector:
    matchLabels:
      app: test-apache
  template:
    metadata:
      labels:
        app: test-apache
    spec:
      containers:
      - name: test-apache
        image: httpd:alpine
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: test-apache
  labels:
    app: test-apache
spec:
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30002
  selector:
    app: test-apache
  type: NodePort
```

然后在管理节点上发布对应的服务，命令如下：

```
#发布应用
sudo kubectl apply -f test-apache.yaml
#查看 Pod、Service 信息
sudo kubectl get deployment test-apache -o wide
sudo kubectl get service test-apache -o wide
```

命令执行后的输出信息如图 3-20 所示。

```
user01@node01:~$ sudo kubectl apply -f test-apache.yaml
deployment.apps/test-apache created
service/test-apache created
user01@node01:~$ sudo kubectl get deployment test-apache -o wide
NAME          READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS    IMAGES          SELECTOR
test-apache   2/2     2            2           4m27s   test-apache   httpd:alpine    app=test-apache
user01@node01:~$  sudo kubectl get service test-apache -o wide
NAME          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE     SELECTOR
test-apache   NodePort   10.99.237.187   <none>        80:30002/TCP   4m40s   app=test-apache
```

图 3-20　部署应用 test-apache

由于应用的发布基于 YAML 文件,因此后续的维护非常简单,例如,如果需要扩展 Pod 的副本数,则只需在对应的 replicas 字段内修改所需值,再执行 kubectl apply 命令应用配置文件使变更生效即可实现 Pod 扩缩。

注意:关于应用发布 YAML 文件编写的方法和技巧会在后续章节内详细讲解。

通过上述简单的功能验证,基本可以确认构建的 Kubernetes 集群是可用的。在企业实际应用场景中会有更多的验证内容,例如数据的持久化存储、负载均衡等,这些内容会在后续的章节中详细讲解。

3.1.6 Kubernetes 部署实战:基于 Containerd 环境

Containerd 是一个开源的容器运行引擎,最初是 Docker 项目的一部分,在 2017 年时 Docker 将拆分后的 Containerd 项目捐赠给云原生计算基金会(CNCF)。自此 Containerd 获得了更多的社区支持和代码贡献,使其得到了快速发展。

1. 弃用 dockershim

dockershim 是一个与 Docker 运行时接口进行交互的组件,它是 Kubernetes 与 Docker 容器运行时通信的桥梁,负责容器生命周期的管理和资源调度等相关操作。在 Kubernetes 集群中由于 dockershim 的存在,增加了 kubelet 自身管理的复杂性,因此从 Kubernetes 1.24 开始正式弃用 dockershim,默认的容器引擎也从 Docker 变更为 Containerd。

2. Containerd 与 Docker 的差异

首先,Docker 是一个完整的容器虚拟化平台,它提供了全栈解决方案,包括构建、打包、分发和运行容器等功能,而 Containerd 则是一个专注于底层容器生命周期管理的轻量级容器管理工具,它旨在为容器运行时提供核心功能,例如镜像下载与推送、容器创建、容器启动、容器停止等容器生命周期管理工作。

其次,由于 Docker 提供了内置的网络和存储解决方案,因此在容器之间通信和数据传输更轻松,而 Containerd 则不包含这些功能,它的网络通信是通过集成 CNI 插件实现的。

最后,Docker 采用客户端与服务器端架构,是通过 Docker 守护进程(dockerd)和 RESTful API 来管理和创建容器的,而 Containerd 则是作为守护进程直接负责管理容器的生命周期,例如容器创建、启动、停止等相关操作,Containerd 架构如图 3-21 所示。

3. Containerd 的优势

Containerd 被企业广泛地应用于大规模生产环境中是由其自身的优势所决定的,主要表现在以下方面。

首先,Containerd 的轻量级设计架构与代码优化,使 Containerd 不仅能以守护进程的形式存在还使其专注于容器生命周期的管理,还可以用最小的资源消耗高效运行。这对于企业大规模生产环境来讲尤为重要,因为它可以显著地提升系统的响应速度和处理能力。

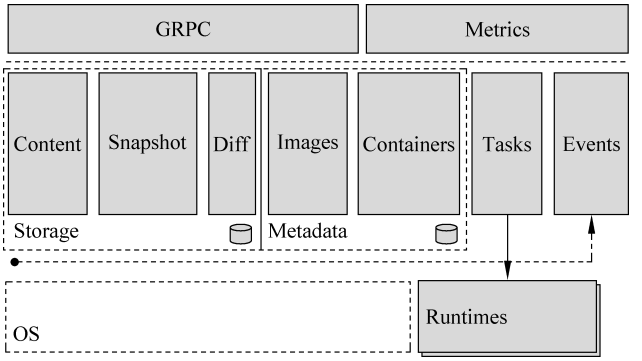

图 3-21　Containerd 架构

其次，支持插件机制，允许使用者根据自身需求扩展其核心功能。例如，可以通过集成不同的网络和存储方案来满足特定场景下的网络通信与数据持久化需求，使企业能够灵活地变化技术路线和业务需求。同时还支持 OCI 标准，使它与其他符合 OCI 标准的容器工具和平台兼容。

最后，Containerd 拥有丰富的生态系统，通过集成众多的工具和插件来增强它的功能。同时，它与 Kubernetes 的紧密集成对应用容器化提供了强有力的支撑，有利于容器虚拟化技术的进一步发展。

4. Containerd 典型管理命令

Containerd 常用的管理命令行工具有两种，分别是 ctr 和 crictl，但是它们之间有一定的差异。ctr 命令提供了更底层的容器操作，是针对容器运行时的，例如 Containerd，而 crictl 命令则提供了更高层次的容器管理功能，是针对 CRI 的。ctr 与 crictl 命令的功能特点见表 3-14。

表 3-14　ctr 与 crictl 命令的功能特点

ctr 命令	crictl 命令
(1) ctr 是 Containerd 的官方命令行工具，用于与 Containerd 进行交互和管理容器 (2) ctr 支持多种命令，包括创建、启动、停止、删除和查看容器等 (3) ctr 可以直接与 Containerd 通信，通过 gRPC 接口进行操作 (4) ctr 提供了更底层的容器管理功能，可以直接操作 containerd 的 API	(1) crictl 是 CRI(容器运行时接口)的官方命令行工具，用于与 CRI 兼容的容器运行时进行交互和管理容器 (2) crictl 支持多种命令，包括创建、启动、停止、删除和查看容器等 (3) crictl 通过 CRI 接口与容器运行时进行通信，可以与不同的容器运行时(例如 Docker、Containerd、rkt 等)进行交互 (4) crictl 提供了更高层的容器管理功能，对于使用 CRI 的容器运行时来讲更加方便

1) ctr 典型命令

在 Containerd 的管理过程中如果使用 ctr 命令，则典型的命令见表 3-15。

表 3-15 ctr 典型命令

类 型	命 令	说 明
镜像管理	ctr images list	列出本地镜像
	ctr images pull	下载镜像
	ctr images delete	删除镜像(参数 delete、del、remove、rm 的功能相同,均可使用)
	ctr images export	导出镜像
	ctr images import	导入镜像
容器管理	ctr run	运行容器
	ctr containers create	创建容器
	ctr containers delete	删除容器(参数 delete、del、remove、rm 的功能相同,均可使用)
	ctr containers info	获取容器信息
	ctr containers list	列出容器
	ctr containers label	设置和清除容器标签
任务管理	ctr task create	创建新任务
	ctr task delete	删除任务
	ctr task list	列出所有任务
	ctr task update	更新任务的配置
	ctr task exec	在任务中执行一个命令
	ctr task kill	终止任务
	ctr task pause	暂停任务
	ctr task resume	恢复任务
	ctr task logs	查看任务的日志
	ctr task inspect	查看任务的详细信息

2) crictl 典型命令

在 Containerd 的管理过程中如果使用 crictl 命令,则典型的命令见表 3-16。

表 3-16 crictl 典型命令

类 型	命 令	说 明
镜像管理	crictl images	列出镜像(参数 images、image、img 的功能相同,均可使用)
	crictl pull	下载镜像
	crictl rmi	删除镜像
容器管理	crictl create	创建一个新的容器
	crictl exec	在运行的容器中执行命令
	crictl inspect	查看容器的详细信息
	crictl logs	获取容器的日志
	crictl start	启动已经创建的容器
	crictl stop	停止容器
	crictl ps	列出容器
	crictl update	更新运行中的容器

续表

类　型	命　令	说　明
Pod 管理	crictl pods	列出所有的 Pod
	crictl stopp	停止运行中的 Pod
	crictl rmp	删除 Pod
	crictl statsp	列出 Pod 资源的使用情况

5. 节点环境准备

在部署基于 Containerd 环境的 Kubernetes 集群时，必要的节点基础配置是必不可少的，例如，配置主机名、IP 地址、时间同步、关闭交换分区、开启流量转发和部署 IPVS 相关组件等，相关环节的配置方法可以参看 3.1.5 节内容（3.1.5 Kubernetes 部署实战：基于 Docker 环境）。演示环境节点服务器信息见表 3-17。

表 3-17　节点服务器信息

主　机　名	IP 地址	说　明
node01	192.168.79.191	管理节点
node02	192.168.79.192	管理节点
node03	192.168.79.193	工作节点
node04	192.168.79.194	工作节点
负载均衡器	192.168.79.200	虚拟 IP 地址（VIP）

1）部署 Containerd

集群所有节点均需部署 Containerd，命令如下：

```
#更新索引、安装依赖组件
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg -y
#添加 Docker 官方 GPG key
sudo install -m 0755 -d /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/
keyrings/docker.gpg

sudo chmod a+r /etc/apt/keyrings/docker.gpg
#配置 Docker 官方源
echo \
"deb [arch=" $(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu \
"$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
#更新索引
sudo apt-get update
#安装 Containerd
sudo apt-get install containerd.io -y
```

Containerd 部署完成后需要将 sandbox 镜像仓库地址修改为国内镜像源地址，首先生

成 Containerd 的默认配置文件,命令如下:

```
#先提升权限
sudo -s
#生成默认的配置文件
containerd config default > /etc/containerd/config.toml
```

然后编辑配置文件/etc/containerd/config.toml 修改 sandbox_image 字段和 SystemdCgroup 字段对应的值,代码如下:

```
sandbox_image = "registry.cn-hangzhou.aliyuncs.com/google_containers/pause:3.10"
SystemdCgroup = true
```

接着,配置/etc/crictl.yaml 配置文件,命令如下:

```
cat > /etc/crictl.yaml << EOF
runtime-endpoint: unix://run/containerd/containerd.sock
image-endpoint: unix://run/containerd/containerd.sock
timeout: 10
debug: false
EOF
```

最后重启 Containerd 服务、设置服务自动并查看服务状态,命令如下:

```
sudo systemctl enable containerd
sudo systemctl start containerd
sudo systemctl status containerd
sudo containerd -v
```

命令执行后可以看到 containerd 服务运行正常,并且使用 containerd -v 命令可以显示当前版本,表示 Containerd 部署成功。

2) 安装 Kubernetes 相关组件

集群各节点均需安装 Kubernetes 相关组件,例如 kubeadm、kubelet、kubectl 等,命令如下:

```
#更新索引、安装依赖包
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gpg -y
#下载 Kubernetes 软件包仓库的公共签名密钥
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -
o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
#添加 Kubernetes 仓库
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/
core:/stable:/v1.31/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
#再次更新索引,安装 kubelet、kubeadm 和 kubectl,并锁定其版本
sudo apt-get update
sudo apt-get install kubelet kubeadm kubectl socat
sudo apt-mark hold kubelet kubeadm kubectl
```

6. Kubernetes 集群部署

Kubernetes 集群节点配置完成后,即可开始初始化集群。

首先，集群初始化之前需要将 kubelet 服务的 cgroup 驱动修改为 systemd，在配置文件/usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf 内添加以下参数，代码如下：

```
Environment = "KUBELET_CGROUP_ARGS = --cgroup-driver = systemd"
```

配置文件修改完成后保存，退出并重启 kubelet 服务，命令如下：

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

其次，获取初始化配置文件 kubeadm-init.yaml，命令如下：

```
sudo kubeadm config print init-defaults > kubeadm-init.yml
```

配置文件生成后编辑 kubeadm-init.yml 文件，代码如下：

```
apiVersion: kubeadm.k8s.io/v1beta4
BootstrapTokens:
- groups:
  - system:Bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
  ttl: 24h0m0s
  usages:
  - signing
  - authentication
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 192.168.79.191
  bindPort: 6443
nodeRegistration:
  criSocket: unix:///var/run/containerd/containerd.sock
  imagePullPolicy: IfNotPresent
  imagePullSerial: true
  name: node01
  taints: null
timeouts:
  controlPlaneComponentHealthCheck: 4m0s
  discovery: 5m0s
  etcdAPICall: 2m0s
  kubeletHealthCheck: 4m0s
  kubernetesAPICall: 1m0s
  tlsBootstrap: 5m0s
  upgradeManifests: 5m0s
---
apiServer: {}
apiVersion: kubeadm.k8s.io/v1beta4
caCertificateValidityPeriod: 87600h0m0s
certificateValidityPeriod: 8760h0m0s
certificatesDir: /etc/kubernetes/pki
```

```
clusterName: kubernetes
controllerManager: {}
dns: {}
encryptionAlgorithm: RSA-2048
etcd:
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.cn-hangzhou.aliyuncs.com/google_containers
kind: ClusterConfiguration
kubernetesVersion: 1.31.0
networking:
  dnsDomain: cluster.local
  serviceSubnet: 10.96.0.0/12
  podSubnet: 10.244.0.0/16
proxy: {}
scheduler: {}
```

基于修改后的 kubeadm-init.yaml 文件测试国内镜像源是否可用,命令如下:

```
sudo kubeadm config images list --config kubeadm-init.yml
```

如果命令执行后如图 3-22 所示,则说明国内镜像源可用。

```
user01@node01:~$ sudo kubeadm config images list --config kubeadm-init.yml
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-apiserver:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-controller-manager:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-scheduler:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/kube-proxy:v1.31.0
registry.cn-hangzhou.aliyuncs.com/google_containers/coredns:v1.11.3
registry.cn-hangzhou.aliyuncs.com/google_containers/pause:3.10
registry.cn-hangzhou.aliyuncs.com/google_containers/etcd:3.5.15-0
```

图 3-22 测试 Kubernetes 国内镜像源

然后初始化 Kubernetes 集群管理节点,命令如下:

```
#提升权限
sudo -s
#基于配置文件 kubeadm-init.yaml 初始化集群
kubeadm init --config kubeadm-init.yml
```

如果命令执行后输出的信息如下,则表示初始化成功:

```
user01@node01:~ $ sudo -s
[sudo] password for user01:
root@node01:/home/user01#kubeadm init --config kubeadm-init.yml
[init] Using Kubernetes version: v1.31.0
[preflight] Running pre-flight checks
        [WARNING FileExisting-socat]: socat not found in system path
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
```

```
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local node01] and IPs [10.96.0.1 192.
168.79.191]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost node01] and IPs [192.168.
79.191 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost node01] and IPs [192.168.79.
191 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "super-admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/
kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests"
[kubelet-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can
take up to 4m0s
[kubelet-check] The kubelet is healthy after 1.002354069s
[api-check] Waiting for a healthy API server. This can take up to 4m0s
[api-check] The API server is healthy after 7.00210864s
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-
system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node node01 as control-plane by adding the labels: [node-
role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-
balancers]
[mark-control-plane] Marking the node node01 as control-plane by adding the taints [node-
role.kubernetes.io/control-plane:NoSchedule]
```

```
[Bootstrap-token] Using token: abcdef.0123456789abcdef
[Bootstrap-token] Configuring Bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order
for nodes to get long term certificate credentials
[Bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[Bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[Bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet
client certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

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

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.79.191:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:8697688156116169d0b8509f64063d1596114
e7d870a3bc78e6c8febfbff7ff0
```

按照初始化后的信息提示操作即可,命令如下:

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

命令执行后,安装 Calico 网络插件具体操作过程可参看 3.1.5 节的步骤。需要特别注意的是 Calico 网络插件配置文件 custom-resources.yaml 内定义的 Pod 子网范围必须与集群初始化配置文件 Kubeadm-init.yml 内的 Pod 子网范围保持一致,演示环境定义的范围为 podSubnet: 10.244.0.0/16,部署命令如下:

```
kubectl create -f tigera-operator.yaml
kubectl create -f custom-resources.yaml
```

当使用命令 kubectl get pods -A 获取目前管理节点的 Pods 状态全部为 Running 时，表示管理节点的相关初始化工作完成，可以添加集群的其他节点，状态信息如图 3-23 所示。

```
user01@node01:~$ sudo kubectl get pods -A
NAMESPACE          NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver   calico-apiserver-6864b6879b-47s45          1/1     Running   0          61s
calico-apiserver   calico-apiserver-6864b6879b-5xfsn          1/1     Running   0          61s
calico-system      calico-kube-controllers-665db99547-t7lbm   1/1     Running   0          3m5s
calico-system      calico-node-pzh7d                          1/1     Running   0          3m5s
calico-system      calico-typha-858d7bb7c-pxtrp               1/1     Running   0          3m5s
calico-system      csi-node-driver-rwhqg                      2/2     Running   0          3m5s
kube-system        coredns-fcd6c9c4-5vmpf                     1/1     Running   0          32m
kube-system        coredns-fcd6c9c4-9dvjp                     1/1     Running   0          32m
kube-system        etcd-node01                                1/1     Running   0          32m
kube-system        kube-apiserver-node01                      1/1     Running   0          32m
kube-system        kube-controller-manager-node01             1/1     Running   0          32m
kube-system        kube-proxy-q65vg                           1/1     Running   0          32m
kube-system        kube-scheduler-node01                      1/1     Running   0          32m
tigera-operator    tigera-operator-89c775547-4p52m            1/1     Running   0          3m12s
```

图 3-23　Calico 插件状态

最后复制初始化后生成的节点加入指令，分别在 node03、node04 上执行，命令如下：

```
#提升权限
sudo -s
#加入集群
kubeadm join 192.168.79.191:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:8697688156116169d0b8509f64063d1596114e7d870a
3bc78e6c8febfbff7ff0 \
--cri-socket=unix://var/run/containerd/containerd.sock
```

节点加入集群后，在管理节点执行 kubectl get nodes 命令获取节点状态，当所有节点为 Ready 时表示集群节点工作正常，如图 3-24 所示。

```
user01@node01:~$ sudo kubectl get pods -A
[sudo] password for user01:
NAMESPACE          NAME                                       READY   STATUS    RESTARTS      AGE
calico-apiserver   calico-apiserver-6864b6879b-47s45          1/1     Running   0             124m
calico-apiserver   calico-apiserver-6864b6879b-5xfsn          1/1     Running   0             124m
calico-system      calico-kube-controllers-665db99547-t7lbm   1/1     Running   0             126m
calico-system      calico-node-n5lc7                          1/1     Running   0             59m
calico-system      calico-node-pzh7d                          1/1     Running   0             126m
calico-system      calico-node-v8nw6                          1/1     Running   0             33m
calico-system      calico-typha-858d7bb7c-2tc6s               1/1     Running   0             32m
calico-system      calico-typha-858d7bb7c-pxtrp               1/1     Running   0             126m
calico-system      csi-node-driver-pcs4t                      2/2     Running   0             113m
calico-system      csi-node-driver-r4g79                      2/2     Running   0             113m
calico-system      csi-node-driver-rwhqg                      2/2     Running   0             126m
kube-system        coredns-fcd6c9c4-5vmpf                     1/1     Running   0             155m
kube-system        coredns-fcd6c9c4-9dvjp                     1/1     Running   0             155m
kube-system        etcd-node01                                1/1     Running   0             156m
kube-system        kube-apiserver-node01                      1/1     Running   0             156m
kube-system        kube-controller-manager-node01             0/1     Running   2 (17s ago)   156m
kube-system        kube-proxy-q65vg                           1/1     Running   0             155m
kube-system        kube-proxy-r9zt6                           1/1     Running   0             113m
kube-system        kube-proxy-vqvk4                           1/1     Running   0             113m
kube-system        kube-scheduler-node01                      1/1     Running   2 (16s ago)   156m
tigera-operator    tigera-operator-89c775547-4p52m            1/1     Running   2 (17s ago)   126m
user01@node01:~$ sudo kubectl get nodes
NAME     STATUS   ROLES           AGE    VERSION
node01   Ready    control-plane   156m   v1.31.1
node03   Ready    <none>          114m   v1.31.1
node04   Ready    <none>          114m   v1.31.1
```

图 3-24　集群 Pod 和节点状态

7. Kubernetes 集群功能验证

集群部署完成后,可以通过发布应用的方式对基本的功能进行验证。编写典型的 Web 服务部署文件 myapp.yaml,代码如下:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: myapp
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: nginx:alpine
        name: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: myapp
  name: myapp
spec:
  ports:
  - name: 80-80
    nodePort: 30003
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: myapp
  type: NodePort
```

在管理节点 node01 上执行部署操作,命令如下:

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

命令执行后,查看对应的 Pods 状态、服务状态,命令如下:

```
#获取 Pods 状态
sudo kubectl get pods
#获取服务状态
sudo kubectl get svc myapp
```

应用成功部署的标识如图 3-25 所示。

```
user01@node01:~$ sudo kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
myapp-6bf5f486bd-2chl4   1/1     Running   0          6m25s
myapp-6bf5f486bd-brf96   1/1     Running   0          6m25s
user01@node01:~$ sudo kubectl get svc myapp
NAME    TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
myapp   NodePort   10.111.194.56   <none>        80:30003/TCP   6m37s
```

图 3-25　myapp 部署成功

如果此时使用浏览器访问集群任意节点的 30003/tcp 端口，则可以看到 Nginx 服务运行的默认页面。同时还可以在命令行模式下管理应用，如修改服务副本数、修改服务类型、修改服务暴露的端口等相关操作。

8. 配置 Kubernetes 集群的高可用架构

在默认情况下 Kubernetes 的集群为单管理节点(控制平面节点)，这种模式虽然可以满足开发和测试等小规模的业务应用场景，但是在生产环境中却面临着重大的风险。主要原因是单节点故障风险，一旦集群中的管理节点出现宕机等故障，整个集群的控制平面就失效了，无法实施对集群的有效管理，严重时可能会导致业务中断，从而影响业务的连续性，因此在企业生产环境中 Kubernetes 集群均采用高可用架构方案来提高集群的容错能力。典型的高可用架构方案有两种，分别是部署多个管理节点(控制平面节点)和使用外部 etcd 集群。

1) 高可用架构方案-部署多个管理节点(控制平面节点)

该方案的特点是通过在集群中部署多个管理节点(控制平面节点)和使用负载均衡器来分发请求，从而实现管理节点的高可用。同时，etcd 节点与管理节点共存，其架构如图 3-26 所示。

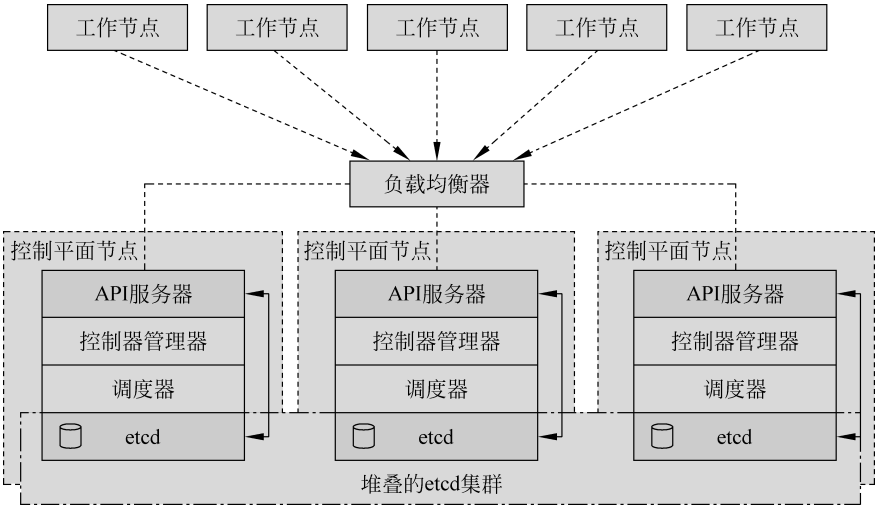

图 3-26　部署多个控制平面节点架构

架构中清晰地列出了每个管理节点(控制平面节点)上均运行 API 服务器、控制器管理器和调度器实例,其中 API 服务器使用负载均衡器暴露给工作节点。同时每个管理节点(控制平面节点)创建一个本地 etcd 成员,这个 etcd 成员只与该节点的 API 服务器、控制器管理器和调度器实例进行通信。通常情况下在该模式下 HA(高可用)集群至少运行 3 个堆叠的管理节点(控制平面节点)。当主管理节点出现故障时,备份管理节点会通过选举算法推举一个新的主节点接管集群,同时会重新分配集群的工作负载。这一过程通常需要一定的时间来完成,在这个期间有可能会出现短暂的服务中断现象,一旦新的管理节点选举完成,集群将完全恢复正常状态。

2) 高可用架构方案-使用外部 etcd 集群

该方案的特点是使用外部 etcd 集群的方式来实现 Kubernetes 集群的高可用。在 Kubernetes 集群中 etcd 集群是用于存储集群元数据和状态信息的,方案是通过配置多个 etcd 节点和使用数据复制机制来实现数据的一致性和可用性,其架构如图 3-27 所示。

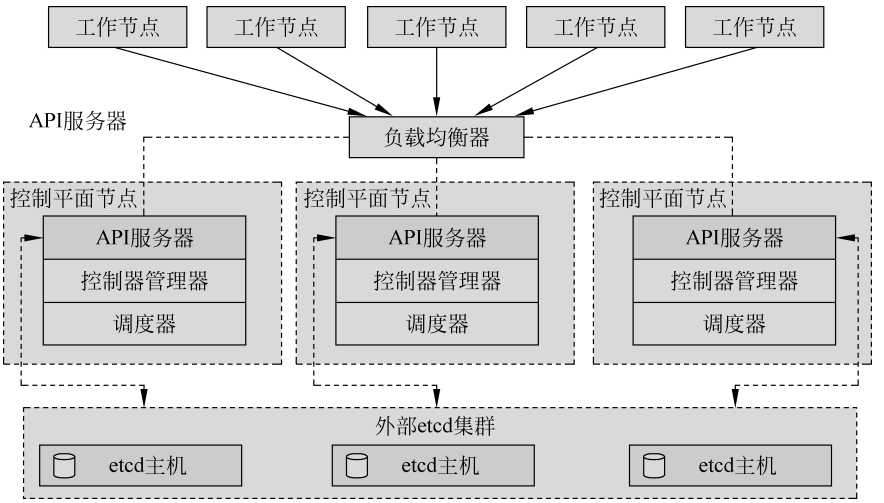

图 3-27 使用外部 etcd 集群

在该方案中外部 etcd 集群中的每个控制平面节点都会运行 API 服务器、控制器管理器和调度器,其中 API 服务器使用负载均衡器暴露给工作节点。需要特别关注的是,etcd 成员在不同的主机上运行,每个 etcd 主机与每个管理节点(控制平面节点)的 API 服务器进行通信。通常情况下在该模式下的高可用集群至少需要 3 个管理节点(控制平面节点)和 3 个冗余 etcd 节点。

3) 高可用架构实现方案

Kubernetes 的高可用架构的核心之一是负载均衡器,它既可以是硬件负载均衡器也可以是软件负载均衡器。在企业生产环境中,负载均衡器的选择是由众多因素决定的,例如项目的实际需求、性能要求、易维护性、性价比等。通常情况下硬件负载均衡器具有专用的处理器和高性能的网络接口,用于处理大量的并发和请求,由于是专业的设备,因此稳定性和

可靠性更高。同时还可以提供其他相关服务，例如流量管理、网络攻击防护等功能，常见的设备有 f5 等设备，而软件负载均衡器则是通过软件方式实现网络流量的负载均衡，与硬件负载均衡器相比具有更高的性价比，它可以根据实际需求轻松地完成扩容或缩减，更适合于动态和弹性架构。针对软件负载均衡器的官方解决方案有两种，分别是 HAProxy 与 Keepalived 组合应用方案和 kube-vip 方案。

注意：在 HAProxy 与 Keepalived 组合应用方案内，Keepalived 官方建议的最优版本为 2.0.20 和 2.2.4。

(1) HAProxy 与 Keepalived 组合。

在 Kubernetes 高可用方案中 HAProxy 是一个开源的负载均衡器和反向代理软件，它的主要功能是将大量并发和请求按照预设的负载均衡算法分配给多个后端的 Kubernetes API 服务器响应，从而提升系统的稳健性和请求处理能力。同时 HAProxy 还具备监控后端服务器健康状态的能力，并且会自动剔除不健康的实例。由于 HAProxy 还支持 SSL/TLS 加密，因此可以在负载均衡层处理加密，进而减轻后端服务器的压力。

在 Kubernetes 高可用方案中 Keepalived 是基于虚拟路由冗余协议(Virtual Router Redundancy Protocol，VRRP)的开源软件，它的主要功能是管理一个由多个 HAProxy 实例共享的虚拟 IP 地址(Virtual IP Address，VIP)，在主节点发生故障时虚拟 IP 地址会自动漂移至备份节点，从而实现冗余和故障转移。

通过二者的组合完美地实现了一个简洁而又高效的 Kubernetes 高可用方案，确保了集群的高可用性和高可靠性。

(2) 使用 kube-vip 实现。

kube-vip 作为 HAProxy 与 Keepalived 组合等传统负载均衡的替代方案，它可以在一个服务中实现虚拟 IP 地址的管理和负载均衡功能。kube-vip 可以在网络模型中的第 2 层(使用 ARP 和 leaderElection 方式)或者第 3 层(使用 BGP 方式)实现，需要特别注意的是此时 kube-vip 将作为管理节点上的静态 Pod 运行。

注意：kube-vip 需要访问 Kubernetes API 服务器，尤其是在集群初始化期间，即使用 kubeadm init 命令初始化集群阶段。

kube-vip 的工作模式包括 ARP 模式、BGP 模式、Routing Table 模式和 WireGuard 模式，掌握不同模式的功能特点及应用场景对后续 Kubernetes 相关技术的应用至关重要。

首先是 ARP 模式，在该模式下 kube-vip 利用地址解析协议(Address Resolution Protocol，ARP)来发布和更新 VIP 地址。一旦 kube-vip 在集群节点上成功运行，该节点就会主动向集群所在网络发送 ARP 广播，告知网络内的所有节点自己所持有的 VIP 地址，这就使其他节点能够通过该 VIP 与 Kubernetes 集群中的服务进行通信，但是由于 ARP 协议本身并不完美，具有一定的缺陷，例如缓存攻击、广播风暴等，因此它更适合应用于相对简单

的网络,尤其是没有复杂路由需求的场景。

其次是 BGP 模式,在该模式下 kube-vip 先使用边界网关协议发布 VIP,然后 VIP 将被通告给网络中的其他 BGP 路由器,从而允许通过 BGP 管理流量的路由选择。这种模式更适合大规模的 Kubernetes 集群或者多租户场景及与数据中心或公有云环境的深度集成。

再次是 Routing Table 模式,在该模式下 kube-vip 通过直接操作节点路由表的方式来管理 VIP,例如通过添加、删除或修改路由条目来实现流量的精准控制。该模式更适合对流量需要精细化控制的场景。

最后是 WireGuard 模式,在该模式下利用 WireGuard 实现安全的点对点加密连接,可以有效地提升数据安全,该模式主要用于多云架构场景。

4) 部署高可用环境

在构建 Kubernetes 高可用环境时容器运行时采用 Containerd,并且集群初始化所需相关环境、软件包等已经配置且部署完成。在此基础上演示典型的高可用方案 HAProxy 与 Keepalived 组合应用方案。

注意:由于 Kubernetes 集群初始化前的相关环境配置、软件部署等操作已展示,因此在高可用部署环节就省略此内容。如果有需要,则可以参看 3.1.6 节的相关内容。

首先在集群节点 node01、node02、node03 上安装 HAProxy 和 Keepalived 软件包,命令如下:

```
sudo apt install haproxy keepalived
```

软件部署完成后,修改集群 node01、node02、node03 节点上的 Keepalived 配置文件 /etc/keepalived/keepalived.conf,其中 node01 为主节点,node02、node03 为备份节点。node01 节点的 keepalived.conf 配置文件中的代码如下:

```
! /etc/keepalived/keepalived.conf
! Configuration File for keepalived

#定义全局参数
global_defs {
   script_user root                                #将脚本执行的用户指定为 root
   enable_script_security                          #启动脚本安全性检查
   router_id LVS_DEVEL                             #设置路由器标识
}
vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"  #指定检测脚本的路径
  interval 3                                      #将检测频率设定为 3s
  weight -3                                       #当检测失败时,降低该实例的权重值
  fall 10                                         #连续 10 次失败后,认定脚本故障
  rise 2                                          #连续两次成功后,认定脚本已恢复健康
}
```

```
vrrp_instance VI_1 {
    state MASTER            #实例状态,MASTER 表示主服务器,BACKUP 表示备份服务器
    interface ens33         #绑定到特定的网络接口
    virtual_router_id 51    #设置虚拟路由器的 ID,在同组 VRRP 内,所有有实例的虚拟路由 ID 必
                            #须相同
    priority 100            #定义优先级,数值越大优先级越高
    authentication {
        auth_type PASS      #认证类型为密码
        auth_pass 1111      #设置认证密码,在同组 VRRP 内的所有实例保持一致
    }
    virtual_ipaddress {
        192.168.79.200/24   #设置 VIP 地址
    }
    track_script {
        check_apiserver     #监控 check_apiserver 脚本
    }
}
```

node02 节点的 keepalived.conf 配置文件中的代码如下：

```
! /etc/keepalived/keepalived.conf
! Configuration File for keepalived
global_defs {
    script_user root
    enable_script_security
    router_id LVS_DEVEL
}
vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"
  interval 3
  weight -3
  fall 10
  rise 2
}

vrrp_instance VI_1 {
    state BACKUP                    #备份服务器
    interface ens33
    virtual_router_id 51
    priority 99
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.79.200/24
    }
    track_script {
        check_apiserver
    }
}
```

node03 节点的 keepalived.conf 配置文件中的代码如下：

```
! /etc/keepalived/keepalived.conf
! Configuration File for keepalived
global_defs {
    script_user root
    enable_script_security
    router_id LVS_DEVEL
}
vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"
  interval 3
  weight -3
  fall 10
  rise 2
}

vrrp_instance VI_1 {
    state BACKUP
    interface ens33
    virtual_router_id 51
    priority 98
    authentication {
        auth_type PASS
        auth_pass 1111
    }
    virtual_ipaddress {
        192.168.79.200/24
    }
    track_script {
        check_apiserver
    }
}
```

检测脚本/etc/keepalived/check_apiserver.sh 的代码如下：

```
#!/bin/sh

errorExit() {
    echo "*** $*" 1>&2
    exit 1
}

curl -sfk --max-time 2 https://localhost:6433/healthz -o /dev/null || errorExit "Error
GET https://localhost:6433/healthz"
```

检测脚本编辑完成后需要配置权限，命令如下：

```
sudo chmod 755 /etc/keepalived/check_apiserver.sh
```

然后将检测脚本/etc/keepalived/check_apiserver.sh 分发至其他高可用管理节点的相

同目录即可。

注意：检测脚本/etc/keepalived/check_apiserver.sh 在集群的 3 个高可用管理节点上的代码一致。

接着配置高可用管理节点上的 HAProxy 配置文件，代码如下：

```
#/etc/haproxy/haproxy.cfg
#---------------------------------------------------------------------
#Global settings
#---------------------------------------------------------------------
#全局设置
global
    #定义日志
    log stdout format raw local0
    #以守护进程模式运行 HAProxy
    daemon
#---------------------------------------------------------------------
#common defaults that all the 'listen' and 'backend' sections will
#use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    http            #HTTP 模式
    log                     global          #使用全局日志模式
    option                  httplog         #启用 httplog
    option                  dontlognull     #不记录空请求的日志
    option http-server-close                #在 HTTP 连接结束时关闭连接
    option forwardfor       except 127.0.0.0/8 #转发客户端 IP,排除 127.0.0.0/8 地址范围
    option                  redispatch      #当服务器不可用时重新调度请求
    retries                 1               #请求失败时重试次数
    timeout http-request    10s             #HTTP 请求超时时间
    timeout queue           20s             #队列等待超时时间
    timeout connect         5s              #连接超时时间
    timeout client          35s             #客户端超时时间
    timeout server          35s             #服务器超时时间
    timeout http-keep-alive 10s             #HTTP 长连接超时时间
    timeout check           10s             #健康检查超时时间

#---------------------------------------------------------------------
#apiserver frontend which proxys to the control plane nodes
#---------------------------------------------------------------------
frontend apiserver
    bind *:6433                             #监听所有 IP 地址的 6433 端口
    mode tcp                                #模式为 TCP
    option tcplog                           #启用 TCP 日志
    default_backend apiserverbackend        #默认后端为 apiserverbackend

#---------------------------------------------------------------------
```

```
#round robin balancing for apiserver
#---------------------------------------------------------------------
#配置负载均衡
backend apiserverbackend
    option httpchk                              #启用 HTTP 健康检查

    http-check connect ssl                      #使用 SSL 连接进行 HTTP 健康检查
    http-check send meth GET uri /healthz       #将 GET 请求发送到/healthz 进行健康检查
    http-check expect status 200

    mode tcp
    balance             roundrobin              #负载均衡模式为轮询方式

    #定义后端服务器配置(包含主机名、IP 地址、端口及健康检查设置)
    server node01 192.168.79.191:6433 check verify none
    server node02 192.168.79.192:6433 check verify none
    server node03 192.168.79.193:6433 check verify none
```

当 Keepalived 和 HAProxy 配置完成后,启动服务并设置服务自启动,命令如下:

```
sudo systemctl enable haproxy --now
sudo systemctl enable keepalived --now
```

注意:HAProxy 支持图形化管理界面,但需要在/etc/haproxy/haproxy.cfg 配置文件中添加相关配置,例如添加以下代码即可实现浏览器方式登录 HAProxy。

```
listen admin_stats
        stats   enable
        #配置访问端口
        bind      *:8000
        #配置浏览器访问的模式
        mode      http
        stats   refresh 30s
        stats   uri   /haproxy-status
        stats   realm haproxy_admin
        #用户 admin 的密码设置为 admin@
        stats   auth admin:admin@
        stats   hide-version
        stats   admin if TRUE
```

一旦 haproxy 和 keepalived 服务成功启动,就可查看 node01 节点的 IP 信息,如果 VIP 地址被成功绑定,则表明高可用负载均衡器部署成功,如图 3-28 所示。

由于后续需要使用 root 权限在管理节点之间进行数据文件传输,因此需要配置 SSH

```
user01@node01:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:2a:6d:21 brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 192.168.79.191/24 brd 192.168.79.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet 192.168.79.200/32 scope global ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe2a:6d21/64 scope link
       valid_lft forever preferred_lft forever
3: ens34: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:2a:6d:2b brd ff:ff:ff:ff:ff:ff
    altname enp2s2
    inet 192.168.172.191/24 brd 192.168.172.255 scope global ens34
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe2a:6d2b/64 scope link
       valid_lft forever preferred_lft forever
```

图 3-28　VIP 地址绑定成功

服务允许 root 用户登录，具体配置如下：

```
#在节点 node01 执行以下操作
#设置 root 登录密码
sudo passwd root
#编辑/etc/ssh/sshd_config 配置文件，设置内容如下
PermitRootLogin yes
#配置文件/etc/ssh/sshd_config 编辑完成后保存并退出编辑器，然后重启 sshd 服务
sudo systemctl restart sshd
```

注意：Ubuntu 22.04 系统内 SSH 服务默认为禁止使用 root 用户登录。

然后使用 kubeadm config 命令生成集群初始化文件 kubeadm-init-ha.yml，命令如下：

```
sudo kubeadm config print init-defaults --component-configs KubeletConfiguration > kubeadm-init-ha.yml
```

初始化配置文件 kubeadm-init-ha.yml 生成后修改其相关参数，代码如下：

```
apiVersion: kubeadm.k8s.io/v1beta4
BootstrapTokens:
- groups:
  - system:Bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
  ttl: 24h0m0s
  usages:
  - signing
  - authentication
kind: InitConfiguration
localAPIEndpoint:
```

```
  advertiseAddress: 192.168.79.191
  bindPort: 6443
nodeRegistration:
  criSocket: unix://var/run/containerd/containerd.sock
  imagePullPolicy: IfNotPresent
  imagePullSerial: true
  name: node01
  taints: null
timeouts:
  controlPlaneComponentHealthCheck: 4m0s
  discovery: 5m0s
  etcdAPICall: 2m0s
  kubeletHealthCheck: 4m0s
  kubernetesAPICall: 1m0s
  tlsBootstrap: 5m0s
  upgradeManifests: 5m0s
---
apiServer: {}
apiVersion: kubeadm.k8s.io/v1beta4
caCertificateValidityPeriod: 87600h0m0s
certificateValidityPeriod: 8760h0m0s
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controllerManager: {}
dns: {}
encryptionAlgorithm: RSA-2048
etcd:
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.cn-hangzhou.aliyuncs.com/google_containers
controlPlaneEndpoint: 192.168.79.200:6443
kind: ClusterConfiguration
kubernetesVersion: 1.31.0
networking:
  dnsDomain: cluster.local
  serviceSubnet: 10.96.0.0/12
  podSubnet: 10.244.0.0/16
proxy: {}
scheduler: {}
---
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt
```

```
authorization:
  mode: Webhook
  webhook:
    cacheAuthorizedTTL: 0s
    cacheUnauthorizedTTL: 0s
cgroupDriver: systemd
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
containerRuntimeEndpoint: ""
cpuManagerReconcilePeriod: 0s
evictionPressureTransitionPeriod: 0s
fileCheckFrequency: 0s
healthzBindAddress: 127.0.0.1
healthzPort: 10248
httpCheckFrequency: 0s
imageMaximumGCAge: 0s
imageMinimumGCAge: 0s
kind: KubeletConfiguration
logging:
  flushFrequency: 0
  options:
    json:
      infoBufferSize: "0"
    text:
      infoBufferSize: "0"
  verbosity: 0
memorySwap: {}
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
shutdownGracePeriod: 0s
shutdownGracePeriodCriticalPods: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

配置文件修改完成后，基于该配置文件初始化 Kubernetes 集群，命令如下：

```
#提升权限
sudo -s
#初始化集群
kubeadm init --upload-certs --config kubeadm-init-ha.yml
```

初始化命令执行后如果有以下信息输出，则表示初始化成功。

```
user01@node01:~$ sudo -s
[sudo] password for user01:
```

```
root@node01:/home/user01# kubeadm init --upload-certs --config kubeadm-init-ha.yml
[init] Using Kubernetes version: v1.31.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local node01] and IPs [10.96.0.1 192.
168.79.191 192.168.79.200]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost node01] and IPs [192.168.
79.191 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost node01] and IPs [192.168.79.
191 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "super-admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/
kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests"
[kubelet-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can
take up to 4m0s
[kubelet-check] The kubelet is healthy after 1.507239244s
[api-check] Waiting for a healthy API server. This can take up to 4m0s
[api-check] The API server is healthy after 8.005761295s
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-
system" Namespace
```

```
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Storing the certificates in Secret "kubeadm-certs" in the "kube-
system" Namespace
[upload-certs] Using certificate key:
5373cc483f3c4d521f873847b8797291aae86582ff3d4ff0c4af55305dc5d4af
[mark-control-plane] Marking the node node01 as control-plane by adding the labels: [node
-role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-
balancers]
[mark-control-plane] Marking the node node01 as control-plane by adding the taints [node-
role.kubernetes.io/control-plane:NoSchedule]
[Bootstrap-token] Using token: abcdef.0123456789abcdef
[Bootstrap-token] Configuring Bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order
for nodes to get long term certificate credentials
[Bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[Bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[Bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet
client certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

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

You can now join any number of the control-plane node running the following command on each as
root:

  kubeadm join 192.168.79.200:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:0b59e90ce9c178ae8ccd31d588898d65c363
6d0408c8b40f28d986388675d3c6 \
```

```
        -- control - plane  -- certificate - key 5373cc483f3c4d521f873847b8797291aae86582ff
3d4ff0c4af55305dc5d4af

Please note that the certificate - key gives access to cluster sensitive data, keep it secret!
As a safeguard, uploaded - certs will be deleted in two hours; If necessary, you can use
"kubeadm init phase upload - certs  -- upload - certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.79.200:6443  -- token abcdef.0123456789abcdef \
        -- discovery - token - ca - cert - hash sha256:0b59e90ce9c178ae8ccd31d588898d65c363
6d0408c8b40f28d986388675d3c6
```

然后按照提示,先部署网络插件,再添加节点,命令如下:

```
#在初始节点 node01 上执行下面的命令
mkdir  - p  $ HOME/.kube
sudo cp  - i /etc/kubernetes/admin.conf  $ HOME/.kube/config
sudo chown  $ (id  - u):$ (id  - g)  $ HOME/.kube/config
#在初始节点 node01 上部署 Calico 插件
sudo kubectl create  - f tigera - operator.yaml
sudo kubectl create  - f custom - resources.yaml
```

此时可以将 node01 节点中的 etcd 相关数据复制至 node02、node03 节点,命令如下:

```
#分别在 node02、node03 节点创建相关目录
cd /home/user01/ && sudo mkdir  - p /etc/kubernetes/pki/etcd && mkdir  - p ~/.kube/
#分别在 node02、node03 节点执行以下命令
sudo scp node01:/etc/kubernetes/pki/ca *  /etc/kubernetes/pki/
sudo scp node01:/etc/kubernetes/pki/sa *  /etc/kubernetes/pki/
sudo scp node01:/etc/kubernetes/pki/front - proxy - ca *  /etc/kubernetes/pki/
sudo scp node01:/etc/kubernetes/pki/etcd/ca *  /etc/kubernetes/pki/etcd/
```

当 Calico 插件相关组件处于 Running 状态时,可以分别将 node02、node03 节点加入管理节点,命令如下:

```
#分别在 node02 和 node03 节点上执行
#提升权限
sudo  - s
#加入管理节点
kubeadm join 192.168.79.200:6443  -- token abcdef.0123456789abcdef \
 -- discovery - token - ca - cert - hash sha256:0b59e90ce9c178ae8ccd31d588898d65c3636d0408c8
b40f28d986388675d3c6 \
 -- control - plane  -- certificate - key 5373cc483f3c4d521f873847b8797291aae86582ff3d4ff0c
4af55305dc5d4af
```

将 node04 节点作为工作节点加入集群,命令如下:

```
#提升权限
sudo  - s
#加入集群
```

```
kubeadm join 192.168.79.200:6443 -- token abcdef.0123456789abcdef \
-- discovery - token - ca - cert - hash sha256:0b59e90ce9c178ae8ccd31d588898d65c3636d0408c8
b40f28d986388675d3c6
```

最终 Kubernetes 高可用集群的节点状态如图 3-29 所示。

```
root@node01:/home/user01# kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
node01    Ready    control-plane   38m   v1.31.1
node02    Ready    control-plane   14m   v1.31.1
node03    Ready    control-plane   13m   v1.31.1
node04    Ready    <none>          12m   v1.31.1
```

图 3-29　高可用集群节点状态

如果此时查看集群中 Kubernetes 关键组件就会发现它们均有 3 个副本，并且 3 个副本分布在 3 个不同的节点上，如图 3-30 所示。

```
root@node01:/home/user01# kubectl get pods -n kube-system
NAME                              READY   STATUS    RESTARTS      AGE
coredns-fcd6c9c4-r8chj            1/1     Running   0             102m
coredns-fcd6c9c4-r9fkg            1/1     Running   0             102m
etcd-node01                       1/1     Running   0             102m
etcd-node02                       1/1     Running   0             78m
etcd-node03                       1/1     Running   0             77m
kube-apiserver-node01             1/1     Running   0             102m
kube-apiserver-node02             1/1     Running   0             78m
kube-apiserver-node03             1/1     Running   0             77m
kube-controller-manager-node01    1/1     Running   1 (38m ago)   102m
kube-controller-manager-node02    1/1     Running   1 (30m ago)   78m
kube-controller-manager-node03    1/1     Running   0             77m
kube-proxy-662p7                  1/1     Running   0             102m
kube-proxy-7gh2l                  1/1     Running   0             77m
kube-proxy-d28nf                  1/1     Running   0             76m
kube-proxy-ltmqx                  1/1     Running   0             78m
kube-scheduler-node01             1/1     Running   1 (38m ago)   102m
kube-scheduler-node02             1/1     Running   1 (30m ago)   78m
kube-scheduler-node03             1/1     Running   0             77m
```

图 3-30　Kubernetes 管理节点组件副本

最后可以通过关闭 node01 节点来模拟该节点故障，查看集群的 VIP 是否会漂移，登录 node02 节点，查看 IP 地址信息的命令如下：

```
ip addr
```

命令执行的结果如图 3-31 所示。

从图 3-31 的 IP 地址信息中可以发现，当节点 node01 故障时 VIP 地址会按照提前定义的节点优先级自动漂移至 node02，在该节点查看集群状态，命令如下：

```
#查看集群节点状态
sudo kubectl get nodes
#查看集群中 Kubernetes 关键组件
sudo kubectl get pods - n kube - system - o wide
```

命令执行的过程如图 3-32 所示。

```
user01@node02:~$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:20:2b:c3 brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 192.168.79.192/24 brd 192.168.79.255 scope global ens33
       valid_lft forever preferred_lft forever
    inet 192.168.79.200/24 scope global secondary ens33
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe20:2bc3/64 scope link
       valid_lft forever preferred_lft forever
```

图 3-31　VIP 漂移至 node02 节点

```
user01@node02:~$ sudo kubectl get nodes
NAME     STATUS     ROLES           AGE    VERSION
node01   NotReady   control-plane   150m   v1.31.1
node02   Ready      control-plane   126m   v1.31.1
node03   Ready      control-plane   125m   v1.31.1
node04   Ready      <none>          124m   v1.31.1
user01@node02:~$
user01@node02:~$ sudo kubectl get pods -n kube-system -o wide
NAME                              READY   STATUS        RESTARTS      AGE     IP               NODE     NOMINATED NODE   READINESS GATES
coredns-fcd6c9c4-7br6s            1/1     Running       0             5m55s   10.244.186.195   node03   <none>           <none>
coredns-fcd6c9c4-8c8ws            1/1     Running       0             5m55s   10.244.248.195   node04   <none>           <none>
coredns-fcd6c9c4-r8chj            1/1     Terminating   0             150m    10.244.196.130   node01   <none>           <none>
coredns-fcd6c9c4-r9fkg            1/1     Terminating   0             150m    10.244.196.129   node01   <none>           <none>
etcd-node01                       1/1     Running       0             150m    192.168.79.191   node01   <none>           <none>
etcd-node02                       1/1     Running       0             126m    192.168.79.192   node02   <none>           <none>
etcd-node03                       1/1     Running       0             125m    192.168.79.193   node03   <none>           <none>
kube-apiserver-node01             1/1     Running       0             150m    192.168.79.191   node01   <none>           <none>
kube-apiserver-node02             1/1     Running       0             126m    192.168.79.192   node02   <none>           <none>
kube-apiserver-node03             1/1     Running       0             125m    192.168.79.193   node03   <none>           <none>
kube-controller-manager-node01    1/1     Running       1 (86m ago)   150m    192.168.79.191   node01   <none>           <none>
kube-controller-manager-node02    1/1     Running       2 (15m ago)   126m    192.168.79.192   node02   <none>           <none>
kube-controller-manager-node03    1/1     Running       0             125m    192.168.79.193   node03   <none>           <none>
kube-proxy-662p7                  1/1     Running       0             150m    192.168.79.191   node01   <none>           <none>
kube-proxy-7gh2l                  1/1     Running       0             125m    192.168.79.193   node03   <none>           <none>
kube-proxy-d28nf                  1/1     Running       0             124m    192.168.79.194   node04   <none>           <none>
kube-proxy-ltmqx                  1/1     Running       0             126m    192.168.79.192   node02   <none>           <none>
kube-scheduler-node01             1/1     Running       2 (15m ago)   150m    192.168.79.191   node01   <none>           <none>
kube-scheduler-node02             1/1     Running       1 (78m ago)   126m    192.168.79.192   node02   <none>           <none>
kube-scheduler-node03             1/1     Running       0             125m    192.168.79.193   node03   <none>           <none>
```

图 3-32　节点 node02 接管管理权

从命令执行后的结果可以发现 node02 节点已完全接管控制权，并且原来运行在 node01 节点上的 Kubernetes 组件会在集群的其他管理节点上创建并运行，始终保持集群的可用性。当然也可以基于该集群发布相关测试服务。

至此，官方提供的 Kubernetes 集群高可用典型方案 HAProxy 与 Keepalived 组合应用演示完成，接下来演示基于 kube-vip 的高可用方案。

注意：由于 Kubernetes 集群初始化前的相关环境配置、软件部署等操作已展示，因此在高可用部署环节省略此内容。如果有需要，则可以参看 3.1.6 节的相关内容。

以下操作在 node01 节点上执行。

首先，部署 kube-vip 前需要配置相关环境变量，命令如下：

```
#提升权限
sudo -s
#设置 VIP 地址
export VIP=192.168.79.200
#指定 VIP 绑定的网卡
export INTERFACE=ens33
#获取 kube-vip 最新版本号
KVVERSION=$(curl -sL https://api.github.com/repos/kube-vip/kube-vip/releases | jq -r
".[0].name")
#配置容器运行时
alias kube-vip="ctr run --rm --net-host ghcr.io/kube-vip/kube-vip:$KVVERSION vip /
kube-vip"
#生成部署清单
kube-vip manifest pod \
--interface $INTERFACE \
--vip $VIP \
--controlplane \
--arp \
--leaderElection | tee /etc/kubernetes/manifests/kube-vip.yaml
```

命令执行后，生成的 kube-vip.yaml 文件代码如下：

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-vip
  namespace: kube-system
spec:
  containers:
  - args:
    - manager
   env:
    - name: vip_arp
      value: "true"
    - name: port
      value: "6443"
    - name: vip_nodename
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: vip_interface
      value: ens33
    - name: dns_mode
      value: first
    - name: cp_enable
      value: "true"
    - name: cp_namespace
      value: kube-system
    - name: vip_leaderelection
```

```
        value: "true"
      - name: vip_leasename
        value: plndr-cp-lock
      - name: vip_leaseduration
        value: "5"
      - name: vip_renewdeadline
        value: "3"
      - name: vip_retryperiod
        value: "1"
      - name: vip_address
        value: 192.168.79.200
      - name: prometheus_server
        value: :2112
    image: ghcr.io/kube-vip/kube-vip:v0.8.4
    imagePullPolicy: IfNotPresent
    name: kube-vip
    resources: {}
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        - NET_RAW
    volumeMounts:
    - mountPath: /etc/kubernetes/admin.conf
      name: kubeconfig
  hostAliases:
  - hostnames:
    - kubernetes
    ip: 127.0.0.1
  hostNetwork: true
  volumes:
  - hostPath:
      path: /etc/kubernetes/admin.conf
    name: kubeconfig
status: {}
```

将生成的/etc/kubernetes/manifests/kube-vip.yaml文件复制至集群其他高可用节点node02、node03的相同目录下,命令如下:

```
#复制至 node02 节点
scp /etc/kubernetes/manifests/kube-vip.yaml root@node02:/etc/kubernetes/manifests/
#复制至 node03 节点
scp /etc/kubernetes/manifests/kube-vip.yaml root@node03:/etc/kubernetes/manifests/
```

注意:

(1) Ubuntu 22 系统 SSH 服务默认禁止 root 登录,需要将/etc/ssh/sshd_config 配置文件内参数 PermitRootLogin 的值修改为 yes,修改完成后重启 sshd 服务即可。

(2) 在撰写本书实验时由于 Kubernetes 1.31 对安全规则进行了调整,因此需要对生成

的清单文件/etc/kubernetes/manifests/kube-vip.yaml进行修改，否则初始化第1个管理节点会报错。大家可以等待kube-vip新版本的更新。

修改清单文件/etc/kubernetes/manifests/kube-vip.yaml内镜像拉取的方式，命令如下：

```
sed -i "s#imagePullPolicy: Always#imagePullPolicy: IfNotPresent#g" /etc/kubernetes/manifests/kube-vip.yaml
```

修改清单文件/etc/kubernetes/manifests/kube-vip.yaml内权限认证，命令如下：

```
sed -i 's#path: /etc/kubernetes/admin.conf#path: /etc/kubernetes/super-admin.conf#' /etc/kubernetes/manifests/kube-vip.yaml
```

kube-vip清单修改完成后，编辑Kubernetes集群初始化配置文件kubeadm-init-ha.yml，代码如下：

```
apiVersion: kubeadm.k8s.io/v1beta4
BootstrapTokens:
- groups:
  - system:Bootstrappers:kubeadm:default-node-token
  token: abcdef.0123456789abcdef
  ttl: 24h0m0s
  usages:
  - signing
  - authentication
kind: InitConfiguration
localAPIEndpoint:
  advertiseAddress: 192.168.79.191
  bindPort: 6443
nodeRegistration:
  criSocket: unix://var/run/containerd/containerd.sock
  imagePullPolicy: IfNotPresent
  imagePullSerial: true
  name: node01
  taints: null
timeouts:
  controlPlaneComponentHealthCheck: 4m0s
  discovery: 5m0s
  etcdAPICall: 2m0s
  kubeletHealthCheck: 4m0s
  kubernetesAPICall: 1m0s
  tlsBootstrap: 5m0s
  upgradeManifests: 5m0s
---
apiServer: {}
apiVersion: kubeadm.k8s.io/v1beta4
caCertificateValidityPeriod: 87600h0m0s
certificateValidityPeriod: 8760h0m0s
```

```
certificatesDir: /etc/kubernetes/pki
clusterName: kubernetes
controllerManager: {}
dns: {}
encryptionAlgorithm: RSA-2048
etcd:
  local:
    dataDir: /var/lib/etcd
imageRepository: registry.cn-hangzhou.aliyuncs.com/google_containers
controlPlaneEndpoint: 192.168.79.200:6443
kind: ClusterConfiguration
kubernetesVersion: 1.31.0
networking:
  dnsDomain: cluster.local
  serviceSubnet: 10.96.0.0/12
  podSubnet: 10.244.0.0/16
proxy: {}
scheduler: {}
---
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt
authorization:
  mode: Webhook
  webhook:
    cacheAuthorizedTTL: 0s
    cacheUnauthorizedTTL: 0s
cgroupDriver: systemd
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
containerRuntimeEndpoint: ""
cpuManagerReconcilePeriod: 0s
evictionPressureTransitionPeriod: 0s
fileCheckFrequency: 0s
healthzBindAddress: 127.0.0.1
healthzPort: 10248
httpCheckFrequency: 0s
imageMaximumGCAge: 0s
imageMinimumGCAge: 0s
kind: KubeletConfiguration
logging:
  flushFrequency: 0
```

```
 options:
    json:
      infoBufferSize: "0"
    text:
      infoBufferSize: "0"
  verbosity: 0
memorySwap: {}
nodeStatusReportFrequency: 0s
nodeStatusUpdateFrequency: 0s
rotateCertificates: true
runtimeRequestTimeout: 0s
shutdownGracePeriod: 0s
shutdownGracePeriodCriticalPods: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
```

执行集群初始化操作，命令如下：

```
#提升权限
sudo -s
#初始化集群
kubeadm init --upload-certs --config kubeadm-init-ha.yml
```

初始化命令执行后输出的信息如下：

```
root@node01:/home/user01#kubeadm init --upload-certs --config kubeadm-init-ha.yml
[init] Using Kubernetes version: v1.31.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local node01] and IPs [10.96.0.1 192.
168.79.191 192.168.79.200]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost node01] and IPs [192.168.
79.191 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost node01] and IPs [192.168.79.
191 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
```

```
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "super-admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/
kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests"
[kubelet-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can
take up to 4m0s
[kubelet-check] The kubelet is healthy after 1.001354904s
[api-check] Waiting for a healthy API server. This can take up to 4m0s
[api-check] The API server is healthy after 5.902690692s
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-
system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Storing the certificates in Secret "kubeadm-certs" in the "kube-
system" Namespace
[upload-certs] Using certificate key:
1e086e93270f296156b84e9090481a569d6d71ae8b01e803b34a48985e9ae1ff
[mark-control-plane] Marking the node node01 as control-plane by adding the labels: [node-
role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-
balancers]
[mark-control-plane] Marking the node node01 as control-plane by adding the taints [node-
role.kubernetes.io/control-plane:NoSchedule]
[Bootstrap-token] Using token: abcdef.0123456789abcdef
[Bootstrap-token] Configuring Bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[Bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order
for nodes to get long term certificate credentials
[Bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[Bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[Bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet
client certificate and key
```

```
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

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

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

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

You can now join any number of the control-plane node running the following command on each as
root:

  kubeadm join 192.168.79.200:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:d59648d259effadf27345c7abeb4f8584d239
cd306b0decfc08decfa9a960cf7 \
        --control-plane --certificate-key 1e086e93270f296156b84e9090481a569d6d71ae8b
01e803b34a48985e9ae1ff

Please note that the certificate-key gives access to cluster sensitive data, keep it secret!
As a safeguard, uploaded-certs will be deleted in two hours; If necessary, you can use
"kubeadm init phase upload-certs --upload-certs" to reload certs afterward.

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.79.200:6443 --token abcdef.0123456789abcdef \
        --discovery-token-ca-cert-hash sha256:d59648d259effadf27345c7abeb4f8584d239
cd306b0decfc08decfa9a960cf7
```

按照提示完成目录创建、文件复制、权限设置及网络插件部署等相关操作，命令如下：

```
#创建目录、复制文件、权限设置
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
#部署 Calico 网络插件
kubectl create -f tigera-operator.yaml
kubectl create -f custom-resources.yaml
```

最后就可以增加新的高可用管理节点、工作节点至集群，命令如下：

```
#增加高可用管理节点
kubeadm join 192.168.79.200:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:d59648d259effadf27345c7abeb4f8584d239cd306b0d
ecfc08decfa9a960cf7 \
--control-plane --certificate-key 1e086e93270f296156b84e9090481a569d6d71ae8b01e803b3
4a48985e9ae1ff
#增加工作节点
kubeadm join 192.168.79.200:6443 --token abcdef.0123456789abcdef \
--discovery-token-ca-cert-hash sha256:d59648d259effadf27345c7abeb4f8584d239cd306b0d
ecfc08decfa9a960cf7
```

注意：一定要按照命令执行后的提示进行操作，否则容易出现集群节点故障或集群工作异常的情况。

确认集群中的 Kubernetes 组件、Calico 组件及 kube-vip 组件全部处于 Running 状态，如果集群节点处于 Ready 状态，则表明集群运行成功，命令如下：

```
#获取集群中的 Kubernetes 组件、kube-vip 组件的状态
sudo kubectl get pods -n kube-system
#获取集群中的 Calico 组件的状态
sudo kubectl get pods -A
#获取集群中节点的状态
Sudo kubectl get nodes
```

命令执行的过程如图 3-33～图 3-35 所示。

```
user01@node01:~$ sudo kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS      AGE
coredns-fcd6c9c4-9pl2n              1/1     Running   0             107m
coredns-fcd6c9c4-g76qs              1/1     Running   0             107m
etcd-node01                         1/1     Running   1             107m
etcd-node02                         1/1     Running   0             91m
etcd-node03                         1/1     Running   0             87m
kube-apiserver-node01               1/1     Running   1             107m
kube-apiserver-node02               1/1     Running   0             91m
kube-apiserver-node03               1/1     Running   0             87m
kube-controller-manager-node01      1/1     Running   2 (63m ago)   107m
kube-controller-manager-node02      1/1     Running   0             91m
kube-controller-manager-node03      1/1     Running   1 (27m ago)   87m
kube-proxy-c456l                    1/1     Running   0             107m
kube-proxy-m7cwt                    1/1     Running   0             91m
kube-proxy-pnf52                    1/1     Running   0             84m
kube-proxy-zqx2c                    1/1     Running   0             87m
kube-scheduler-node01               1/1     Running   2 (63m ago)   107m
kube-scheduler-node02               1/1     Running   1 (27m ago)   91m
kube-scheduler-node03               1/1     Running   1 (59m ago)   87m
kube-vip-node01                     1/1     Running   2 (27m ago)   107m
kube-vip-node02                     1/1     Running   0             70m
kube-vip-node03                     1/1     Running   1 (59m ago)   87m
```

图 3-33　Kubernetes 组件与 kube-vip 组件的运行状态

至此，基于 kube-vip 的 Kubernetes 集群部署完成，接下来就可以进行集群高可用性测

```
user01@node01:~$ sudo kubectl get pods -A --field-selector metadata.namespace!=kube-system
NAMESPACE          NAME                                       READY   STATUS    RESTARTS      AGE
calico-apiserver   calico-apiserver-658b56bf77-5hj58          1/1     Running   0             89m
calico-apiserver   calico-apiserver-658b56bf77-qr48r          1/1     Running   0             89m
calico-system      calico-kube-controllers-99ffb7cfb-zvgr8    1/1     Running   0             96m
calico-system      calico-node-gqsw9                          1/1     Running   0             81m
calico-system      calico-node-jjnlm                          1/1     Running   0             85m
calico-system      calico-node-pdmg4                          1/1     Running   0             96m
calico-system      calico-node-xkrls                          1/1     Running   0             78m
calico-system      calico-typha-898db5497-tq895               1/1     Running   0             96m
calico-system      calico-typha-898db5497-zbllx               1/1     Running   0             81m
calico-system      csi-node-driver-6j897                      2/2     Running   0             78m
calico-system      csi-node-driver-g9xqn                      2/2     Running   0             81m
calico-system      csi-node-driver-hhr4t                      2/2     Running   0             85m
calico-system      csi-node-driver-mfq9r                      2/2     Running   0             96m
tigera-operator    tigera-operator-89c775547-6g8rx            1/1     Running   3 (21m ago)   96m
```

图 3-34　Calico 网络组件的运行状态

```
user01@node01:~$ sudo kubectl get nodes
NAME     STATUS   ROLES           AGE    VERSION
node01   Ready    control-plane   107m   v1.31.1
node02   Ready    control-plane   92m    v1.31.1
node03   Ready    control-plane   87m    v1.31.1
node04   Ready    <none>          84m    v1.31.1
```

图 3-35　集群节点状态

试了。常见的方式是通过模拟管理节点故障来测试集群是否具有高可用性。

3.2　基于 Kubernetes 的应用管理

随着企业在应用开发过程中越来越多地采用云原生架构，用于满足产品的快速迭代、灵活扩展和高可用性的需求，基于 Kubernetes 的应用管理日益显示出其重要性。Kubernetes 应用容器管理的核心是通过容器编排来实现容器化应用的自动化管理，确保在分布式环境下应用的高可用性和弹性伸缩性。

1. 应用管理的核心思想

在 Kubernetes 应用管理过程中，其核心思想是将应用程序与其运行环境进行解耦，通过抽象层实现应用程序的可移植性、可扩展性和高可用性。首先，Kubernetes 的设计理念是尽可能地对容器进行自动化部署、扩展和管理，使开发者和运维人员能够更专注于应用开发和架构优化。其次，Kubernetes 采用声明式配置，用户只需通过配置文件定义期望的应用状态，系统便会自动调整实际状态，易实现预期效果。同时，Kubernetes 具有强大的自我修复能力，例如，当某个 Pod 失败或被终止时，Kubernetes 会自动重新调度并创建新的 Pod 实例，以确保应用的高可用性和服务的持久性。这样就极大地简化了运维管理的复杂性，使应用能够快速恢复。Kubernetes 还可以根据应用对资源的需求量在集群中自动选择最合适的节点来运行应用实例，例如对于计算密集型的应用会优先调度至 CPU 性能强的节点。最后，Kubernetes 会根据集群中业务的实际负载情况自动地对业务进行扩容或缩减，已达到对资源的合理利用。

2. 应用管理的原理与实现方法

在应用管理过程中,Kubernetes 是通过将资源归属到不同的命名空间中来实现逻辑上的资源隔离。首先通过 YAML 文件或 JSON 文件定义 Kubernetes 资源(例如 Pod、Deployment、ConfigMap 等)来明确应用的期望状态,然后使用 kubectl apply 命令部署这些已定义的资源,其中 Deployment 资源是用来管理应用的生命周期的,包括版本控制、滚动更新和自愈能力等。

在企业实践中 Kubernetes 包管理工具 Helm 的应用也非常广泛,它通过 chart 提供一种标准化的方式来描述 Kubernetes 资源,例如 Deployment、Service、PersistenvolumeClaim 等,使应用的部署变得可复制并易于管理和共享。同时还具有版本控制功能,可以非常方便地对版本进行升级和在必要时对版本进行回滚。尤其是由于 Helm 类似于软件包管理器,它可以处理复杂的应用依赖关系,用于确保所有相关的 Kubernetes 应用资源能够正确地进行部署与更新,进一步地提升了部署的准确性和可靠性。

对于应用管理的实现方法不仅只有上述方案,企业中典型的实现方式还有很多,例如利用 CI/CD 工具(例如 Jenkins 等)可以实现应用的构建、测试和部署的自动化,以及使用持久卷(PV)和持久卷申领(PVC)来实现应用数据的持久化存储等众多方式。

3.2.1 Kubernetes 集群应用生命周期管理

Kubernetes 集群中应用生命周期的管理涉及众多阶段和环节,下面将以企业真实场景下的应用生命周期管理为案例进行展开,在这一过程中将涉及应用打包准备阶段、应用部署阶段、应用相关管理阶段、应用被删除阶段等所涉及的各个环节。

1. 应用镜像构建

它属于应用准备阶段,通常情况下使用 Dockerfile 文件构建应用的容器镜像,并将经过验证测试后的镜像推送至镜像仓库,其中构建镜像所需的源代码可以来源于企业本地代码仓库或公共代码仓库,例如 GitHub 等。

2. 应用创建

应用镜像准备完成后就可以通过 YAML 文件或 JSON 文件创建和自定义应用所需资源及应用的期望状态,例如应用副本数、服务端口、数据持久化存储等。当然也可以使用 Helm 定义应用资源相关参数,以及设置期望状态等。

注意:关于 Helm 的使用方法会在后续章节进行详细讲解。

---

3. 应用部署

应用部署的方式有多种,常见的方式是使用 kubectl create 命令、基于 YAML 文件或者使用 Helm 部署应用等,其中基于 YAML 文件和使用 Helm 方式在企业生产环境应用最普遍,最主要的原因是后期,可以非常方便和快捷地对应用进行维护。

4. 应用运行与监控

当应用被成功部署后，Kubernetes 会监控集群中应用的相关状态，一旦某个 Pod 运行失败就会立即创建并生成新的 Pod，使之始终与定义的应用期望值保持一致。在这一环节中，企业往往会采用第三方监控工具对集群中的各种资源进行多维度监控，例如 Prometheus、Grafana、ELK 等。

5. 应用资源扩缩容

集群中应用对资源的需求是一个动态的过程，这就要求应用副本应该被控制在一个合理的范围内。过多的资源配置会造成资源浪费，过少的资源配置又会影响应用的响应效率。通常情况下的应对方案分别是手动扩缩容和自动扩缩容，手动扩缩容则是通过命令 kubectl scales 实现的，而自动扩缩容方式中最常见的是 Pod 自动伸缩（Horizontal Pod Autoscaler，HPA），例如定义一个 HPA 对象 php-apache，让它监控 php-apache 的 CPU 使用率。将平均 CPU 利用率的阈值设置为 50%，当高于 50%时增加 Pod 数量，反之则减少 Pod 数量，同时将 Pod 数量的范围设置在 1～10，实现的代码如下：

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler                    #指定资源类型
metadata:
  name: php-apache                               #定义 HPA 名称
spec:
  #制定扩缩的目标资源
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1                                 #定义最小副本数
  maxReplicas: 10                                #定义最大副本数
  #定义用于扩缩的指标
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

6. 应用更新与回滚

在企业生产环境中应用更新的典型方式是滚动更新，例如通过修改 Deployment 配置中的镜像版本来实现应用的滚动更新，并且在更新失败时可以使用命令 kubectl rollout undo 快速地恢复到上一个版本。

7. 应用故障处理

Kubernetes 具有自愈能力，即它会自动检查并重启失败的 Pod，用于保持服务的高可用

性。同时还可以使用命令 kubectl get event 查看事件日志,识别和分析故障原因。如果集群集成了第三方监控告警工具,则可以通过集成工具分析故障原因。

8. 应用删除和清理

当应用被废弃时即代表应用生命终结,可以使用 kubectl delete 删除不再使用的资源,如果相关应用的数据进行了持久化存储,则需确认存储的数据确实可以清除时,才手动删除相关无效数据。

3.2.2 编写 YAML 文件的技巧介绍

Kubernetes 在企业生产活动中,使用 YAML 文件描述系统中各组件的期望值及定义和配置资源是最主要的方式,因此掌握 YAML 文件的编写技巧与方法是至关重要的,它涵盖了 YAML 文件的基本结构、最佳实践和常见问题分析方法等。

1. YAML 文件的基本结构

首先来看在 Kubernetes 中的一个典型 Pod 文件示例,代码如下:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.27.2
    ports:
     - containerPort: 80
```

从典型示例中可以发现,YMAL 文件通常由以下几部分组成。

1) apiVersion

用于指定 Kubernetes API 的版本,例如 apps/v1、v1 等。需要特别注意的是不同资源版本中资源的使用方式和参数配置存在一定的差异,需要查阅 Kubernetes 对应文档。

2) kind

用于指定创建的对象类别,例如 Pod、Deployment、Service 等。

3) metadata

它包含了资源对象的元数据,例如 name(必需的字段)、namespace(可选字段,用于指定命名空间,默认为 default)、labels(标签字段)等。

4) spec

用于指定对象的期望状态,包含资源对象的规格、配置信息等,例如在示例中将容器的名称设置为 nginx,容器启动的基础镜像为 nginx:1.27.2,同时还将容器端口指定为 80/tcp 等。

注意：不同的 Kubernetes 资源对象，spec 内的字段值具有的自身特性各不相同。

2. 编写 YAML 文件的技巧

在 Kubernetes 应用实践中编写高效的 YAML 文件时需要遵循一定的规范。

1）明确资源类型与 API 版本

在编写 YAML 文件时需要明确创建的资源类型和对应的 API 版本，其目的是确保编写后的 YAML 文件与 Kubernetes 集群版本相兼容，例如，Pod 资源通常使用的 apiVersion 为 v1，而 Deployment 资源通常使用的 apiVersion 为 apps/v1。

2）合理使用标签和选择器

在编写 YAML 过程中需要合理地使用标签（labels）和选择器（selectors）。标签是 Kubernetes 中用于标识和选择资源的关键字段，通过为资源添加合适的标签，可以非常方便地对资源进行筛选和管理。同时，选择器也常用于指定哪些资源应该被选中操作，例如，可以使用标签选择器来指定 Deployment 应该管理哪些 Pod，代码如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.27.2
        ports:
        - containerPort: 80
```

在上述示例代码中，通过 Deployment 创建了 3 个 nginx 应用副本。

3）定义明确的资源配置信息

这部分定义的内容在 spec 内，需要详细的资源配置信息，同时也是 YAML 内最复杂的部分，也是最能体现资源特性的地方。如果是定义 Pod 资源，则需要指定容器列表、容器启动时所需的镜像、端口映射、环境变量、资源限额等配置信息。如果是定义 Deployment 资源，则需要指定副本数（如果未指定该字段，则默认副本数为 1）、滚动更新策略、Pod 模板

等。如果是定义 Service 资源,则需要指定服务类型(ClusterIP、NodePort、LoadBalancer)、端口映射、选择器等。

4) 使用模板

在实际工作中,为了提升 YAML 文件的复用率往往会通过定义通用模板的方式来实现,例如,首先定义一个通用的 Deployment 模板,然后可以根据不同的应用需求修改模板中的部分字段,例如镜像、副本数、端口等,通过这种自定义通用模板的方式可以实现快速生成多个 Deployment 资源。如果使用的是 Helm 包管理工具,则可以通过 Helm chart 中包含的一组 YAML 模板文件和配置文件来管理应用程序的配置与部署。

注意:在编写 YAML 模板时,应该考虑不同环境和场景的需求,并定义相应的变量和条件语句,这有助于保持 YAML 文件的灵活性和可扩展性。

---

5) 资源规划

在 YAML 文件编写的过程中,对资源的合理规划可以有效地提升代码的执行效率和复用率。例如,为了后期维护更便利,通常情况下会将相同的资源放置在同一个 YAML 文件中(使用---分隔),同时还可以使用不同的命名空间来隔离不同的业务环境。例如,可以使用 ConfigMap 和 Secret 将配置数据与敏感数据从应用程序代码中分离出来,并作为独立的资源进行管理,这有助于保持应用程序的灵活性和可移植性。再如,通过 HPA 来根据 CPU 或内存的使用情况自动调整 Pod 的数量,以此来提升资源利用率,通过 Liveness 和 Readiness 探针来检测 Pod 的健康状况等。

注意:在编写 YAML 文件时,可以使用 ConfigMap 和 Secret 来存储配置数据,并在 Pod 的 spec.containers.envFrom 字段中引用它们。这有助于简化 Pod 的配置和管理,并减少数据的泄露风险。

---

6) 可维护性与持续性

在编写 YAML 文件时需要考虑未来可能会升级和变更,因此建议使用版本控制来管理 YAML 文件。同时为了提升代码的复用率,除了在编写代码时应对一些关键值尽量采用变量的形式外,合理的代码注释也是非常重要的方式。例如可以使用 Helm 工具来管理 Kubernetes 应用程序的配置与部署,以便在升级时能够轻松地更新 YAML 文件。

7) 调试与验证

在将编写好的 YAML 文件部署到 Kubernetes 集群之前,需要使用 kubectl 命令行工具进行验证和测试。例如,使用 kubectl explain 命令查看资源的详细字段和说明,以确保 YAML 文件中使用的字段是有效的。使用 kubectl apply --dry-run=client -f <filename> 命令模拟部署过程并检查潜在的问题。该命令会解析 YAML 文件并生成相应的 Kubernetes 对象,但不会实际创建它们。此时,通过检查输出信息,可以发现可能的错误或警告。最后,可以使用 kubectl apply -f <filename>命令将 YAML 文件部署到 Kubernetes 集群中,并观

察资源的创建和状态变化。如果出现问题，则可以使用 kubectl describe 命令查看资源的详细信息以便排除故障。

8）常见问题与解决方法

当基于 YAML 文件部署失败时，常见的问题与解决方法如下：

（1）YAML 文件解析错误。

检查缩进是否一致，确保没有混用空格和制表符，尤其是不能有多余的空格或不符合 YAML 规范的字符。

（2）资源创建失败。

资源创建失败的原因有很多种，例如资源配额不足、网络异常等情况。最常见的原因是镜像下载失败，主要是由于网络原因造成的。如果需要进一步分析故障原因，则可以使用 kubectl describe 来查看事件和错误信息。

（3）更新配置不生效。

最常见的原因是服务未加载新配置，处理方法是使用命令 kubectl rollout restart < deployment >重新启动 Pod 副本。

9）使用社区插件

人们常讲“工欲善其事，必先利其器”，高效高质量地编写 YAML 代码同样也不能缺少相关工具插件。例如开发者常用的 VS Code 开发工具，运行该工具并在扩展内搜索 Kubernetes 关键字，就会有大量的 Kubernetes 插件可供选择和使用。从中安装适合自己的插件即可，例如提供语法高亮显示、提供模板等众多功能插件，合理有效地利用这些插件可以有效地提升 YAML 文件的编写效率和质量，同时还可以积极参加 Kubernete 社区，从中了解并学习最新的开发动态、开发经验等。

3.2.3　应用发布实战

“纸上得来终觉浅，绝知此事要躬行”，快速高效地编写 YAML 需要大量的实践练习，下面将以企业环境下典型的编写方式进行演示，涵盖了编程工具、相关插件的部署与应用、代码测试、应用发布等环节。

1. 演示环境集群信息

演示环境集群节点信息见表 3-18，集群中容器运行时采用的是 Containerd。

表 3-18　演示环境集群节点信息

名　　称	IP 地址	说　　明
node01	192.168.79.191	管理节点
node03	192.168.79.193	工作节点
node04	192.168.79.194	工作节点

注意：在使用 Kubernetes 集群进行应用发布前，需要确认集群运行正常。例如查看集

群节点状态、Kubernetes 组件运行状态、网络插件运行状态等相关信息。当前在演示环境中的 Kubernetes 版本为 1.31。

2. 基于 Kubernetes 官方模板

随着 Kubernetes 版本的不断迭代更新,有部分参数会被弃用,因此在编写应用的 YAML 文件时,建议一定要查看 Kubernetes 官方提供的相关资源定义模板。

通过前面的学习知道 Pod 是 Kubernetes 中最小的运行单元,通常是由 Deployment 或者 Job 等这类工作负载资源来创建的。首先来看官方提供的 nginx-deployment.yaml 示例,代码如下:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

示例中将 Pod 的副本数定义为 3,容器名称为 nginx,镜像为 nginx:1.14.2,容器端口为 80/TCP。基于上述 Deployment 示例,如果此时需要部署 Tomcat,使用的镜像为 tomcat:11,副本数为 2,则 tomcat-deployment.yaml 文件中的代码如下:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tomcat-deployment
  labels:
    app: tomcat
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: tomcat
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
      - name: tomcat
        image: tomcat:11
        ports:
        - containerPort: 8080
```

注意：Tomcat 服务默认的侦听端口为 8080/TCP。

在集群管理节点部署 Tomcat 应用，命令如下：

```
sudo kubectl apply -f tomcat-deployment.yaml
```

部署完成后可以查看其状态，命令如下：

```
#获取 Deployment 状态信息
sudo kubectl get deployment
#获取 Pods 状态信息
sudo kubectl get pods --show-labels
```

命令执行的过程及部署成功的标识如图 3-36 所示。

```
user01@node01:~$ sudo kubectl get deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
tomcat-deployment   1/1     1            1           25s
user01@node01:~$ sudo kubectl get pods --show-labels
NAME                                 READY   STATUS    RESTARTS   AGE   LABELS
tomcat-deployment-66c9cddcd4-6m4zd   1/1     Running   0          37s   app=tomcat,pod-template-hash=66c9cddcd4
```

图 3-36　Deployment 和 Pods 状态信息

此时集群外部是无法访问已部署的 Tomcat 应用的，需要将外部请求路由至 Pods 的 8080/TCP 端口，即通过创建服务来实现。创建服务文件 tomcat-service.yaml，将 NodePort 的端口设置为 30080/TCP，代码如下：

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
  labels:
    app: tomcat
spec:
  selector:
    app: tomcat
  ports:
    - protocol: TCP
```

```
    port: 8080
    targetPort: 8080
    nodePort: 30080
  type: NodePort
```

部署并查看服务,命令如下:

```
#部署服务
sudo kubectl apply -f tomcat-service.yaml
#查看服务状态
sudo kubectl get svc
```

命令执行的过程如图 3-37 所示。

```
user01@node01:~$ sudo kubectl apply -f tomcat-service.yaml
service/tomcat-service created
user01@node01:~$ sudo kubectl get svc
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP          9d
tomcat-service   NodePort    10.97.153.206   <none>        8080:30080/TCP   16s
```

图 3-37 Tomcat 服务部署成功

如果此时通过浏览器的方式访问集群任意节点的 30080/TCP 端口,则会提示 404 错误,如图 3-38 所示。

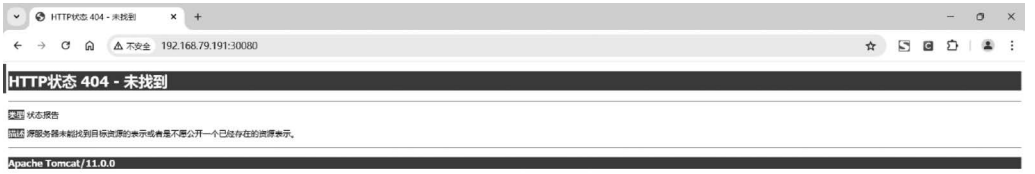

图 3-38 404 错误提示

出现错误的原因是,基于 Tomcat 11 镜像启动的容器默认/usr/local/tomcat/webapps 目录内是空的,Tomcat 默认页面相关数据存储在/usr/local/tomcat/webapps/webapps.dist 目录下。解决问题的方法是将目录 webapps.dist 内的数据复制至 webapps 目录,或者先删除 webapps 目录再将 webapps.dist 目录重命名为 webapps,命令如下:

```
#获取 Tomcat 的 Pod 名称
sudo kubectl get pods
#登录 Pod 查看当前目录下的文件信息
sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls -l'
#查看 webapps 目录内的数据
sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls webapps'
#删除原始的 webapps 目录,并将 webapps.dist 目录重命名为 webapps
sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'rm -rf webapps;mv
webapps.dist webapps'
#再次查看 webapps 目录内的数据
sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls webapps'
```

命令执行的过程如图 3-39 所示。

```
user01@node01:~$ sudo kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
tomcat-deployment-66c9cddcd4-6m4zd   1/1     Running   0          8m59s
user01@node01:~$ sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls -l'
total 132
drwxr-xr-x 2 root root  4096 Oct 19 02:57 bin
-rw-r--r-- 1 root root 21039 Oct  3 17:00 BUILDING.txt
drwxr-xr-x 1 root root    22 Oct 21 15:55 conf
-rw-r--r-- 1 root root  6166 Oct  3 17:00 CONTRIBUTING.md
drwxr-xr-x 2 root root  4096 Oct 19 02:57 lib
-rw-r--r-- 1 root root 60517 Oct  3 17:00 LICENSE
drwxrwxrwt 1 root root    80 Oct 21 15:55 logs
drwxr-xr-x 2 root root   158 Oct 19 02:57 native-jni-lib
-rw-r--r-- 1 root root  2333 Oct  3 17:00 NOTICE
-rw-r--r-- 1 root root  3291 Oct  3 17:00 README.md
-rw-r--r-- 1 root root  6469 Oct  3 17:00 RELEASE-NOTES
-rw-r--r-- 1 root root 16109 Oct  3 17:00 RUNNING.txt
drwxrwxrwt 2 root root    30 Oct 19 02:57 temp
drwxr-xr-x 2 root root     6 Oct 19 02:57 webapps
drwxr-xr-x 7 root root    81 Oct  3 17:00 webapps.dist
drwxrwxrwt 2 root root     6 Oct  3 17:00 work
user01@node01:~$ sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls webapps'
user01@node01:~$ sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'rm -rf webapps;mv webapps.dist webapps'
user01@node01:~$ sudo kubectl exec tomcat-deployment-66c9cddcd4-6m4zd -- /bin/sh -c 'ls webapps'
docs
examples
host-manager
manager
ROOT
```

图 3-39　解决 404 故障的过程

上述操作完成后刷新浏览器即可看到 Tomcat 的初始页面，如图 3-40 所示。

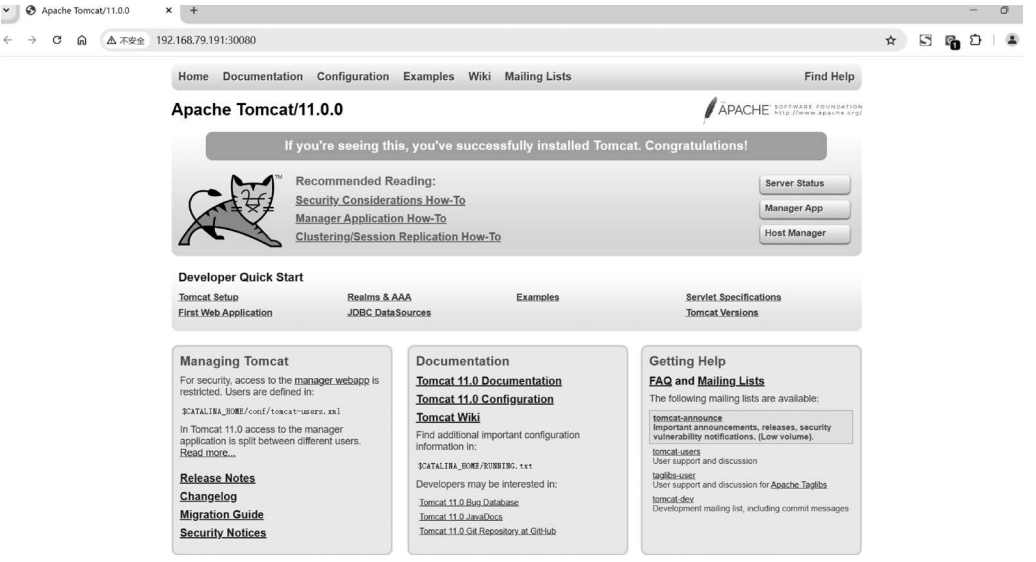

图 3-40　Tomcat 初始页面

注意：在后续的学习与工作过程中，建议养成经常查看官方文档的习惯。因为随着 Kubernetes 版本的更新迭代，以及技术的不断更新，对于已有字段参数的支持也会发生变化甚至弃用。

3. 基于 kubectl 命令

在 Kubernetes 的应用与管理过程中,使用命令行模式是最常见、最高效的,其中最常见的应用场景是在部署应用时使用命令直接生成部署文件,例如,基于 httpd:alpine 镜像部署 3 个副本 myapp 应用,同时使用 NodePort 服务方式发布应用服务并将端口设置为 30081/TCP,实现的命令如下:

```
#使用命令生成 myapp-deployment.yaml 文件
sudo kubectl create deployment myapp \
--image=httpd:alpine \
--port=80 \
--replicas=3 \
--namespace=default \
--dry-run=client \
-o yaml > myapp-deployment.yaml
#使用命令生成 myapp-server.yaml 文件
sudo kubectl create service nodeport myapp \
--tcp=80:80 \
--node-port=30081 \
--dry-run=client \
-o yaml > myapp-service.yaml
```

注意:--dry-run 参数必须是 none、server 或者 client,默认值为 none。当参数是 client 时,仅打印将要发送的对象,而实际并不发送。当参数是 server 时,提交服务器端请求而不持久化资源。

生成的 myapp-deployment.yaml 文件中的代码如下:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: myapp
  name: myapp
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: myapp
```

```
    spec:
      containers:
      - image: httpd:alpine
        name: httpd
        ports:
        - containerPort: 80
        resources: {}
status: {}
```

生成的 myapp-service.yaml 文件中的代码如下：

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: myapp
  name: myapp
spec:
  ports:
  - name: 80-80
    nodePort: 30081
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: myapp
  type: NodePort
status:
  loadBalancer: {}
```

如果此时将生成的文件与 Kubernetes 官方提供的示例进行对比，就会发现核心字段都是一致的。只是使用命令生成的文件中多了 creationTimestamp 字段和 status 字段，其中 creationTimestamp 字段用于表示资源对象在 Kubernetes 集群中被创建的时间，这个时间戳是资源对象在被创建时自动生成的，主要用于审计、监控和故障排查等场景。status 字段是表示资源对象在 Kubernetes 集群中的当前状态，这种状态信息是 Kubernetes 系统资源对象的实际运行情况，例如资源的健康状态、资源是否就绪、是否有错误等。这两个字段都是由 Kubernetes 系统自动生成的，用户无法直接定义或修改其字段值，因此在编写 YAML 文件时均可省略。

接下来可以在集群中直接使用该文件部署应用，如果后续需要维护，则可直接修改该文件即可。当然为了使后续维护更方便可以直接将 myapp-deployment.yaml 和 myapp-server.yaml 文件合并为一个 myapp.yaml 文件，它们之间使用 3 个短横线进行隔离，并且在代码中移除 creationTimestamp 字段和 status 字段，最终合并后的代码如下：

```
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  labels:
    app: myapp
  name: myapp
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  strategy: {}
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: httpd:alpine
        name: httpd
        ports:
        - containerPort: 80
        resources: {}
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: myapp
  name: myapp
spec:
  ports:
  - name: 80-80
    nodePort: 30081
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: myapp
  type: NodePort
```

4. 基于 VS Code 编程工具

Visual Studio Code(简称 VS Code)是由微软开发的一款功能强大的免费代码编辑器，以其跨平台兼容性和出色的代码编辑能力、内置调试工具、广泛的编程语言支持、丰富的扩展插件等受到众多开发者的喜欢。同时还提供了智能的代码补全、语法高亮、代码折叠、多光标编辑等快捷功能，使开发者在代码编写、调试和管理代码等方面变得更加高效和流畅。VS Code 还支持与 Git 等版本控制系统集成及内置的终端模拟器，让开发者能够在不离开

编辑器的情况下完成代码的编写、版本控制和运行命令等一系列操作。此外，VS Code 还提供了丰富的扩展插件，使代码编写变得更加智能和高效。

1）下载、安装 VS Code

访问 VS Code 官方网站下载合适的版本并安装，当前 VS Code 支持的操作系统类型与版本如图 3-41 所示。

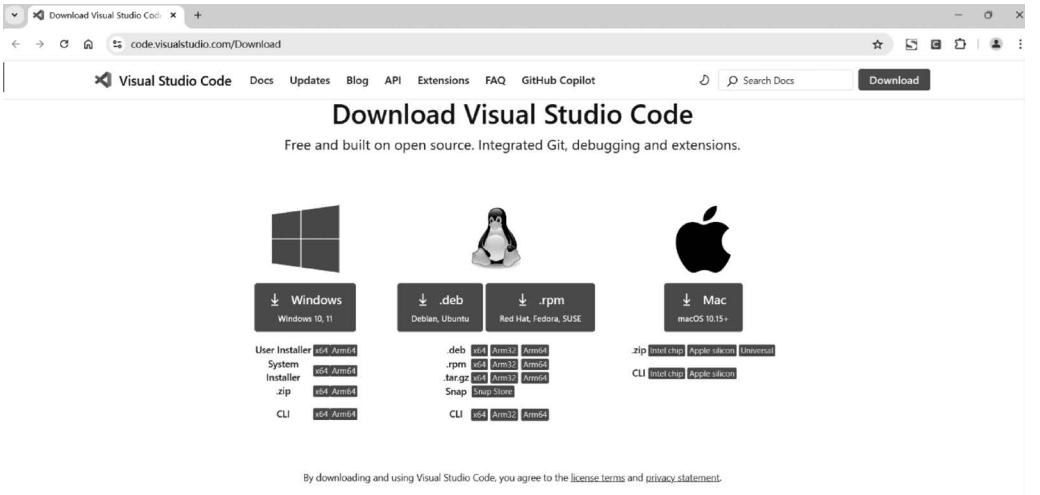

图 3-41　VS Code 支持的操作系统类型与版本

2）安装配置 Kubernetes 插件

丰富的扩展插件是 VS Code 的优势，在扩展栏内输入关键字 Kubernetes 后将显示所有包含 Kubernetes 关键字的插件，选择由微软发布的 Kubernetes 插件并安装，如图 3-42 所示。

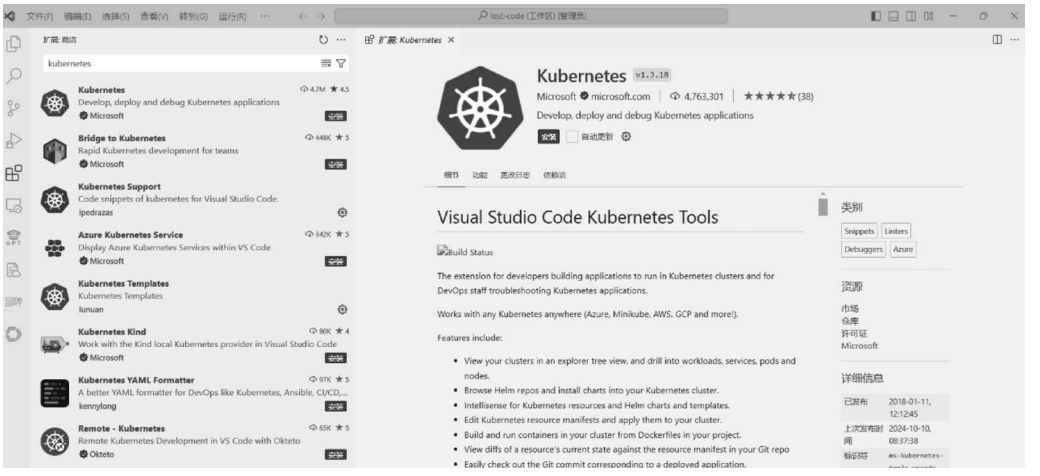

图 3-42　搜索 Kubernetes 插件

Kubernetes 插件安装完成后，单击 VS Code 左侧导航栏内的 Kubernetes 图标后会显示 CLUSTERS、HELM REPOS 和 CLOUDS 这 3 个选项，选择 CLUSTERS 选项的更多操作，选择 Set Kubeconfig，即可实现通过配置文件的方式添加需要管理的 Kubernetes 集群，如图 3-43 所示。

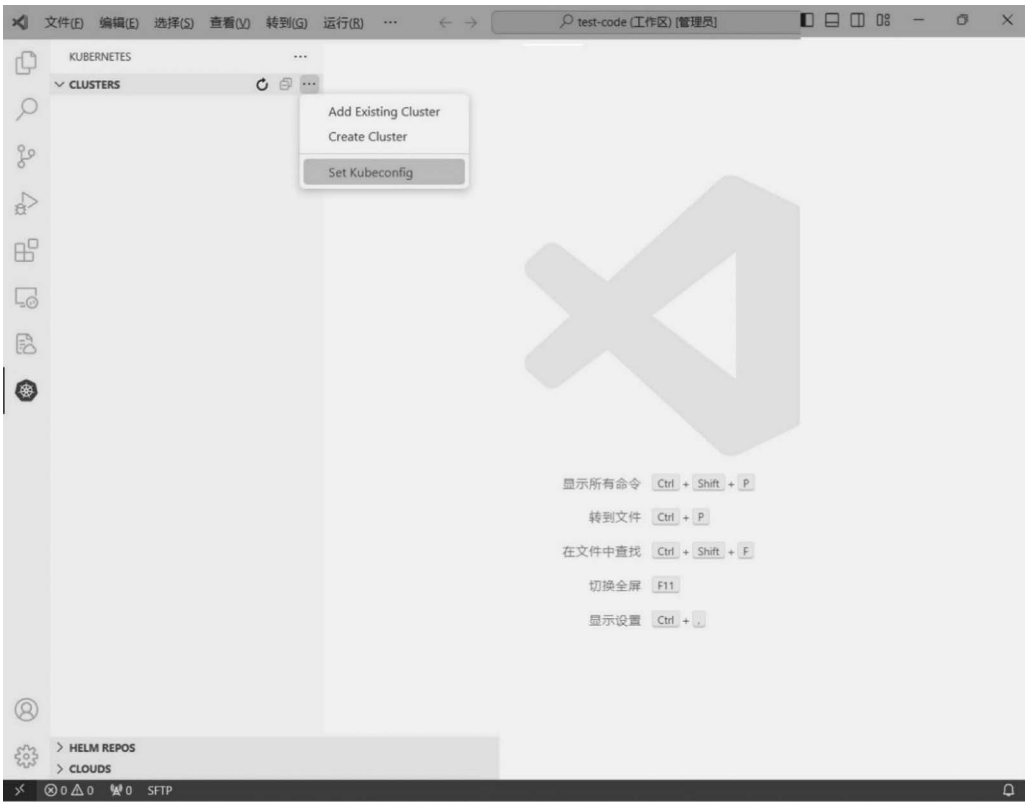

图 3-43　添加 Kubernetes 集群

注意：

(1) 插件 Kubernetes 安装后系统会提示安装相关依赖环境，选择安装即可，系统会自动安装，如图 3-44 所示。

(2) 通过 VS Code 编辑器中的 Kubernetes 插件管理集群时，插件所需的 Kubeconfig 文件需要从运行中的 Kubernetes 集群的管理节点内获取，配置文件路径为/root/.kube/config。

如果远程 Kubernetes 集群添加成功，就可以通过 VS Code 终端使用命令管理 Kubernetes 集群，如图 3-45 所示。

接下来，首先使用 VS Code 代码编辑器创建测试代码 vscode-deployment-nginx.yaml，然后在代码编辑页面输入 depl 关键字，编辑器会提示包含 Deployment 的提示信息，如图 3-46 所示。

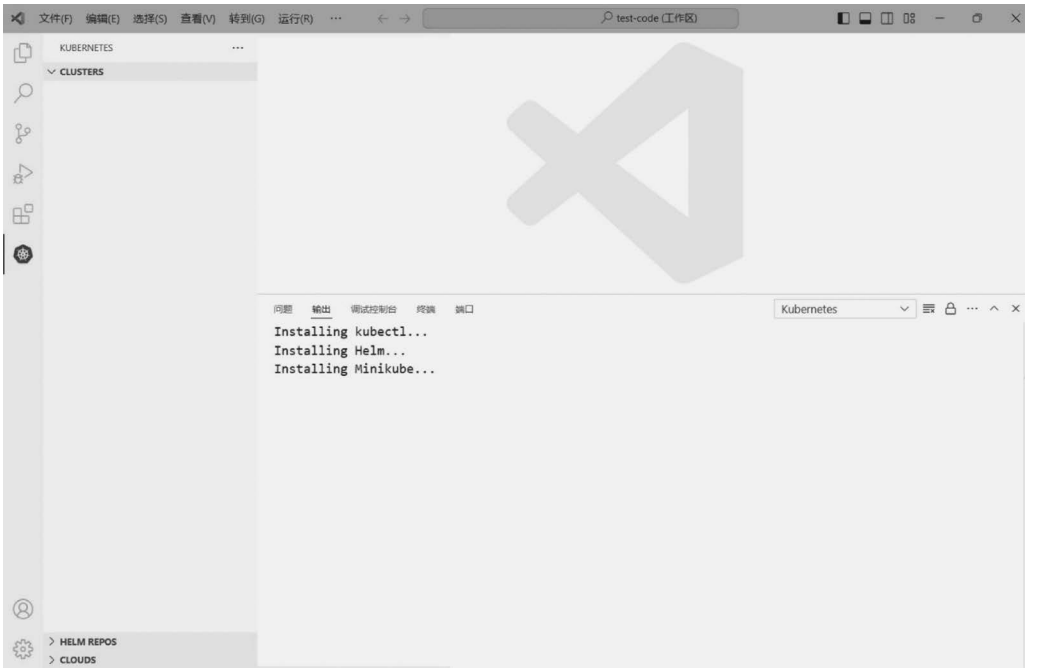

图 3-44　依赖环境安装进程显示

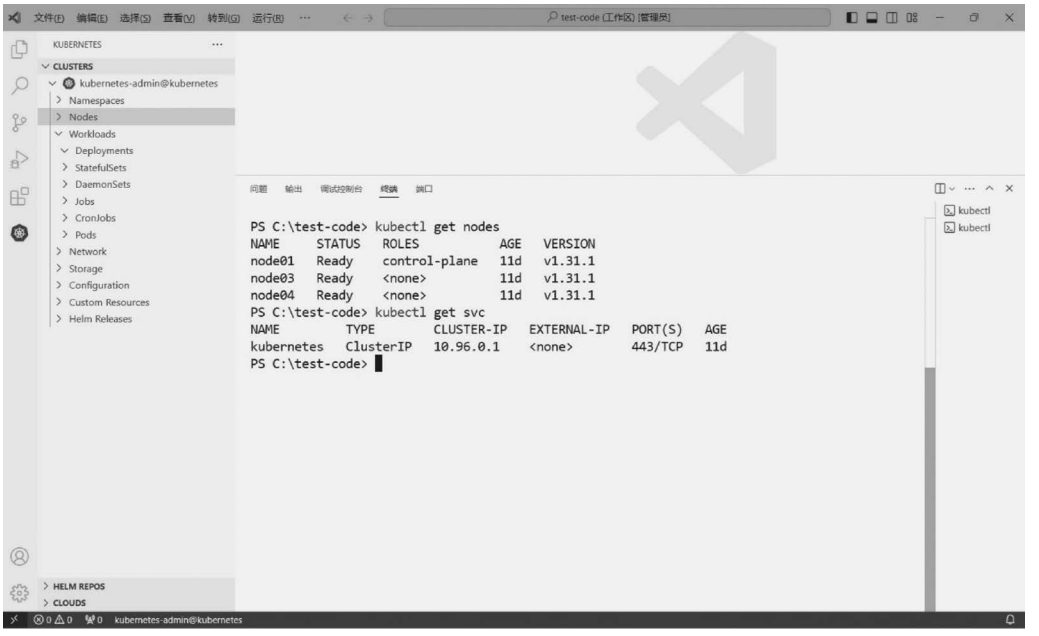

图 3-45　VS Code 终端远程管理 Kubernetes 集群

在提示信息内选择 Kubernetes Deployment 后，编辑器会自动生成 Deployment 示例代码，如图 3-47 所示。

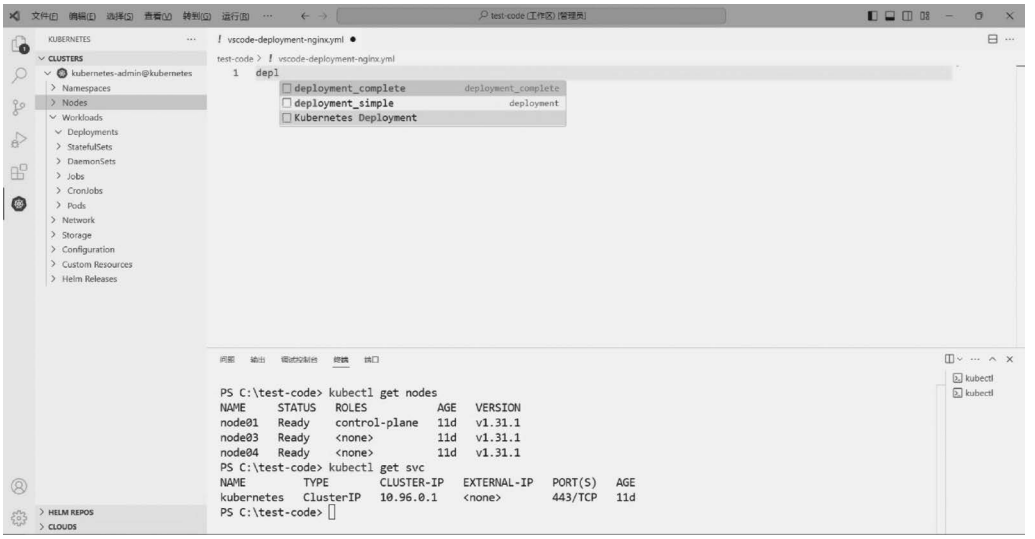

图 3-46 Deployment 代码提示

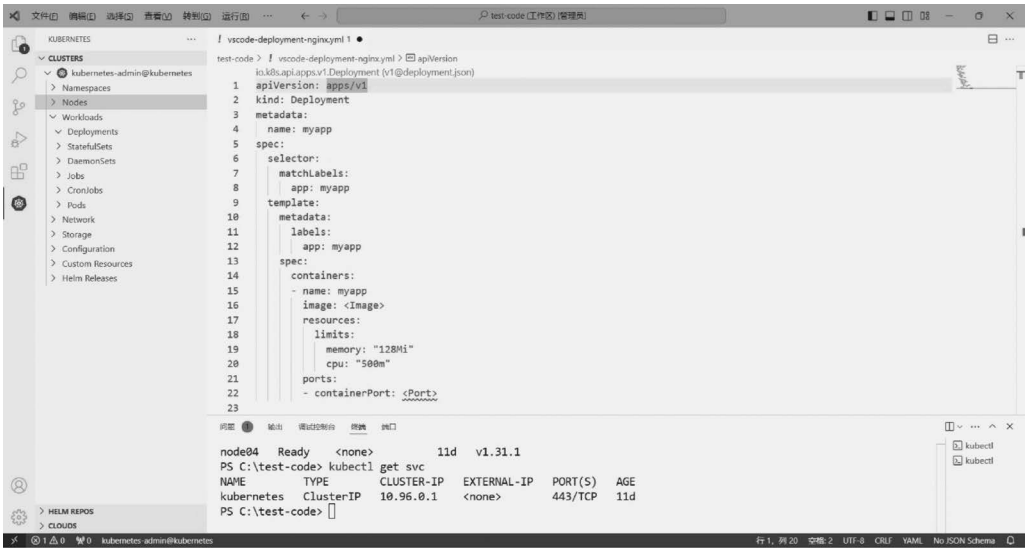

图 3-47 Deployment 代码示例

Kubernetes 插件生成的代码包含了 Deployment 的关键字段，在此基础上可以编辑适合自身需求的代码内容，对于开发者来讲很高效。例如，如果需要基于 nginx:alpine 镜像部署 3 个副本应用，则只需修改名称、端口和资源限制参数，代码如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-nginx
```

```
spec:
  selector:
    matchLabels:
      app: myapp-nginx
  template:
    metadata:
      labels:
        app: myapp-nginx
    spec:
      containers:
      - name: myapp-nginx
        image: nginx:alpine
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

同样的操作，在代码编辑器页面输入 service 关键字后，编辑器会提示包含 service 的提示信息，如图 3-48 所示。

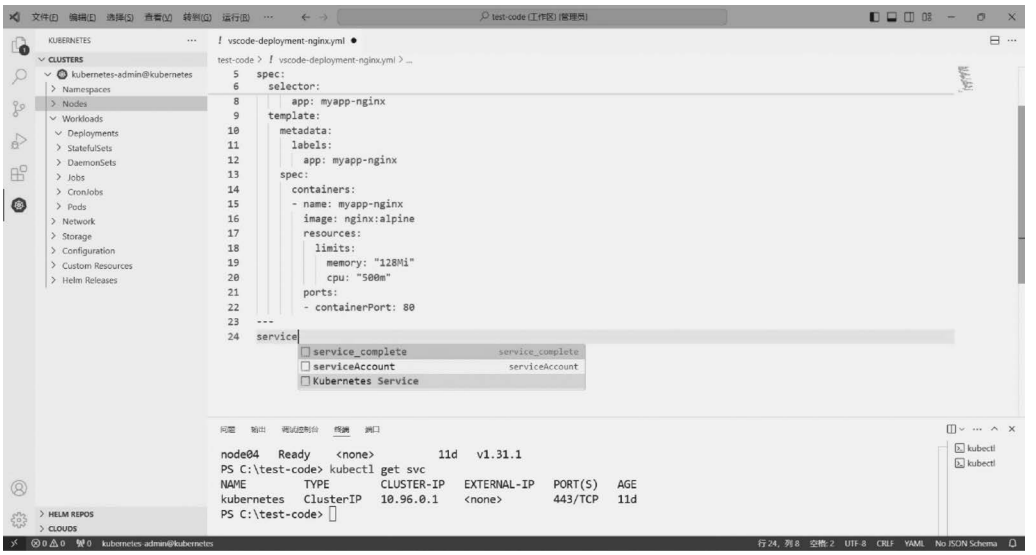

图 3-48　Service 代码提示

在提示信息内选择 Kubernetes Service 后，编辑器会自动生成 Service 示例代码，如图 3-49 所示。

该示例代码包含 Service 的关键字段，需要依据自身需求添加相关参数，例如，当需要允许集群之外的用户访问时，需要使用 NodePort 类型，并且可以自定义端口等，修改后的代码如下：

图 3-49 Service 示例代码

```
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-nginx
spec:
  selector:
    app: myapp-nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
  type: NodePort
```

在实际应用场景中,更多的是将 Deployment 和 Service 保存在一个文件内,最终整合后的 vscode-deployment-nginx.yaml 文件中的代码如下:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-nginx
spec:
  selector:
    matchLabels:
      app: myapp-nginx
  template:
    metadata:
      labels:
        app: myapp-nginx
```

```
    spec:
      containers:
      - name: myapp-nginx
        image: nginx:alpine
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: myapp-nginx
spec:
  selector:
    app: myapp-nginx
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
  type: NodePort
```

文件保存后，在 VS Code 终端使用 kubectl 命令直接将测试服务发布到集群进行测试，如图 3-50 所示。

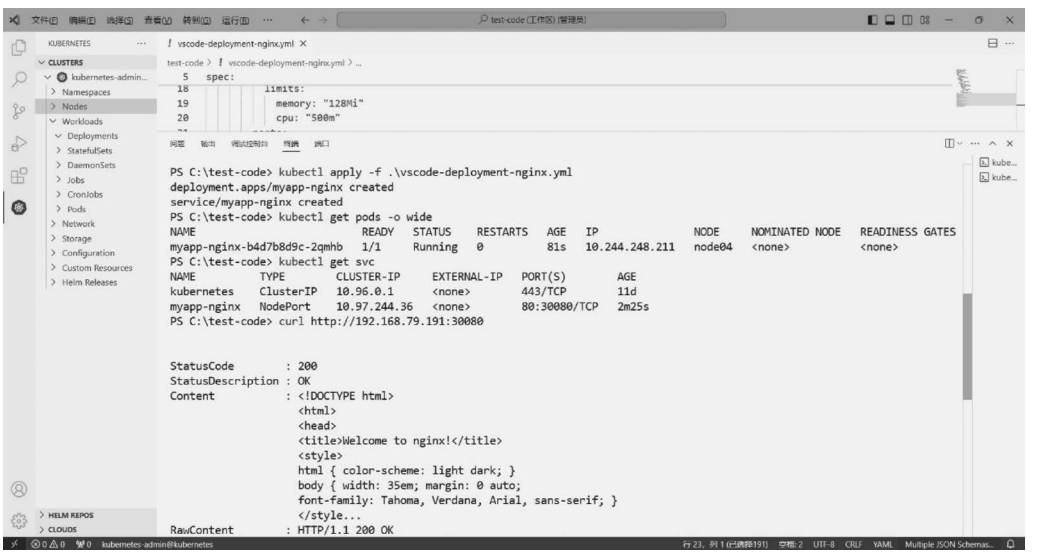

图 3-50　基于 VS Code 终端部署服务

通过这种方式实现了代码的高效编写和代码测试。

5. 基于 Helm

Helm 是 Kubernetes 的包管理工具,它设计的初衷是简化在 Kubernetes 集群中部署和管理应用程序的流程,降低管理难度。它类似于 Ubuntu 系统内的包管理命令 apt,能够快速地查找、下载、安装软件包。Helm 是通过 chart(是 Helm 的打包格式,它定义了 Kubernetes 应用程序所需要的所有资源)来简化复杂应用的部署过程,并提供版本控制、参数化配置、依赖管理等功能。它可以使开发者更加高效地管理和部署 Kubernetes 应用,提高开发效率和应用的稳定性。

注意:Helm 3 中移除了原有 Helm 2 的 Tiller 组件,改为直接从 Kubernetes API Server 获取信息,通过 chart 客户端处理并在 Kubernetes 中存储安装记录。

---

1) 为什么要使用 Helm

首先 Helm 简化了 Kubernetes 应用管理的流程,它通过 chart 简化应用的安装和管理,实现了复杂应用的快速部署。同时在企业内部可以将 Kubernetes 资源封装成可复用的 chart 并用于构建专属的 chart 库,实现了 chart 的共享和复用,进一步地降低了开发成本,并且提高了开发效率,其次 Helm 提供了应用的版本管理功能,可以非常方便地实现版本升级和回滚,提升了对应用生命周期的管理能力。与此同时,Helm 还支持使用应用配置文件的形式来管理应用,只需通过调整配置参数而无须修改 chart 便可实现应用在不同环境中快速部署。最后 Helm 拥有一个非常活跃的社区,在 Helm 仓库内有数量庞大的高质量 chart 供用户下载使用,进而加快企业应用程序的开发、部署和管理过程。

2) Helm 相关术语

(1) chart:chart 是 Helm 的打包格式,它定义了在 Kubernetes 中应用程序、工具或服务运行所需的所有 YAML 格式的资源定义文件,以及可能的服务定义、依赖关系和应用程序的配置。

(2) Repository:Repository 是用来存储和共享 chart 的地方,可以理解为软件仓库,只不过它是提供给 Kubernetes 所用。

(3) Release:Release 是运行在 Kubernetes 集群中的 chart 实例,一个 chart 通常可以在同一个集群中被多次安装,每次安装都会创建一个新的 Release,并且每个 Release 都有一个唯一的名称,同时还记录了部署时的状态等信息。

(4) config:config 包含了用于创建一个可发布对象所需的配置信息。

(5) Helm 客户端:Helm 客户端用于和 Kubernetes 集群交互、管理 chart 和 Release。例如,可以通过 Helm 客户端执行查询、安装、升级、卸载等操作。

(6) values.yaml 文件:它是 chart 中的一个配置文件,用于定义 chart 部署时的变量和配置信息。例如,可以在 values.yaml 文件中设置应用程序所需的镜像版本、资源配额等相关参数。

3）Helm 3 架构

当前 Helm 的最新版本是 Helm 3，它是 Helm 的第 3 个主要版本。它在 Helm 2 的基础上进行了重大更新，特别是移除了 Tiller 服务器组件，这使 Helm 3 的架构更加简洁，安全性也进一步得到提升。Helm 3 的核心架构组件如下。

（1）Helm 客户端：Helm 客户端是架构中的核心组件之一，它是一个命令行工具，用于和 Kubernetes 集群交互，主要用于管理本地 chart、管理仓库、管理发布及与 Helm 库建立接口等。通过客户端命令可以轻松地实现应用的安装、升级、回滚、移除等。典型的客户端命令有 helm install、helm upgrade、helm rollback 等。

（2）Helm 库：它提供与 Helm 操作相关的所有功能，并且 Helm 库独立封装了 Helm 操作逻辑，以便不同的客户端使用。同时 Helm 库通过与 Kubernetes API 服务器交互可以实现对 chart 进行管理，例如在 Kubernetes 集群中安装、升级、卸载 chart 等相关操作。

4）Helm 3 语法规范

Helm 3 的语法规范主要围绕 Helm Chart 的定义和使用，以下是 Helm 3 的典型基础语法规范。

（1）Chart.yaml 文件：Helm chart 是一个包含 Kubernetes 清单文件的文件夹，它有固定的目录结构。一个 chart 至少有一个包含 chart 元数据的 Chart.yaml 文件，该文件定义了名称、版本和其他描述信息。同时一个 chart 还可以包含一个 values.yaml 文件，它用于定义 chart 的默认配置值。Helm chart 的典型目录结构如下：

```
mychart/
  Chart.yaml
  values.yaml
  charts/
  templates/
```

其中，典型的 Chart.yaml 文件字段说明如下：

```
apiVersion:                    #chart API 版本(必选)
name:                          #chart 名称(必选)
version:                       #语义化 2 版本(必选)
kubeVersion:                   #兼容 Kubernetes 版本的语义化版本(可选)
description:                   #对项目的简单描述(可选)
type:                          #chart 类型(可选)
keywords:
  - #关于项目的一组关键字(可选)
home:                          #项目 home 页面的 URL(可选)
sources:
  - #项目源码的 URL 列表(可选)
dependencies:                  #chart 必要条件列表(可选)
  - name:                      #chart 名称(例如,nginx)
    version:                   #chart 版本(例如,"1.2.3")
    repository: #(可选)仓库 URL("https://example.com/charts")或别名("@repo-name")
    condition: #(可选)解析为布尔值的 YAML 路径,用于启用/禁用 chart(例如,subchart1.enabled)
```

```
    tags:                 #(可选)
      - #用于一次启用/禁用 一组 chart 的 tag
    import-values:        #(可选)
      - #ImportValue 将源值保存到导入父键的映射.每项可以是字符串或者一对子/父列表项
    alias: #(可选)chart 中使用的别名(当用户要多次添加相同的 chart 时非常有用)
maintainers:              #(可选)
  - name:                 #维护者名字(每个维护者都需要)
    email:                #维护者邮箱(每个维护者可选)
    url:                  #维护者 URL(每个维护者可选)
icon:                     #用作 icon 的 SVG 或 PNG 图片 URL(可选)
appVersion:               #包含的应用版本(可选),不需要语义化,建议使用引号
deprecated:               #不被推荐的 chart(可选,布尔值)
annotations:
  example:                #按名称输入的批注列表(可选)
```

(2) 模板：templates 目录用于存储模板文件，需要注意的是模板文件是使用 Go 语言编写的，可以引用 values.yaml 文件中的变量和 Helm 的全局作用域对象。常见的模板文件包括 Deployment、Service、ConfigMap、Secret 等 Kubernetes 资源定义文件。

注意：模板文件在命名时应该遵循以下原则：

① 模板文件名称具有唯一性，避免与其他文件混淆。

② 模板文件名称应该清晰，建议能够直观地反映文件的内容或用途，例如 bar-svc.yaml

③ 模板文件名称使用中画线(my-front-configmap.yaml)，不使用驼峰命名方式，避免使用特殊字符。

④ 模板文件后缀通常是.tpl 或者.yaml(直接作为 Kubernetes 资源清单文件)。

---

(3) values.yaml 文件：values.yaml 文件包含了 chart 的默认配置选项，在安装或升级 chart 时，可以指定一个 values.yaml 文件，用于覆盖 chart 的默认值。

5) Helm 3 常用命令

Helm 具有丰富的管理命令，常用的管理命令有包管理命令、仓库管理命令、插件管理命令等。

首先是典型的 Helm 包管理命令，见表 3-19。

表 3-19 典型的 Helm 包管理命令

命 令	功 能 说 明
helm create	创建新的 chart
helm install	将 chart 上传到 Kubernetes
helm list	列出已发布的 chart
helm pull	将 chart 下载到本地目录
helm rollback	回滚到指定版本
helm search	搜索 chart

续表

命　　令	功 能 说 明
helm show	显示 chart 内容
helm status	显示命名版本的状态
helm upgrade	升级版本
helm uninstall	卸载版本

其次是典型的 Helm 仓库管理命令，见表 3-20。

表 3-20　典型的 Helm 仓库管理命令

命　　令	功 能 说 明
helm repo add	添加 chart 仓库
helm repo list	列出 chart 仓库
helm repo remove	删除一个或多个仓库
helm repo update	从 chart 仓库中更新本地可用 chart 的信息

最后是典型的 Helm 插件管理命令，见表 3-21。

表 3-21　典型的 Helm 插件管理命令

命　　令	功 能 说 明
helm plugin install	安装一个或多个 Helm 插件
helm plugin list	列出已安装的 Helm 插件
helm plugin uninstall	卸载一个或多个 Helm 插件
helm plugin update	升级一个或多个 Helm 插件

6）部署 Helm 3

通常情况在 Helm 3 部署在 Kubernetes 的管理节点，部署命令如下：

```
#获取密钥并将密钥导入 helm.gpg
sudo curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee /usr/share/keyrings/helm.gpg > /dev/null
#安装所需软件包等
sudo apt-get install apt-transport-https --yes
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/helm.gpg] https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/sources.list.d/helm-stable-debian.list
#更新索引并安装 Helm 3
sudo apt-get update
sudo apt-get install helm
#查看安装后的 Helm 版本信息
sudo helm version
```

如果可以查看版本信息，则标志着 Helm 3 环境部署完成，如图 3-51 所示。

```
user01@node01:~$ sudo helm version
version.BuildInfo{Version:"v3.16.2", GitCommit:"13654a52f7c70a143b1dd51416d633e1071faffb", GitTreeState:"clean", GoVersion:"go1.22.7"}
```

图 3-51　Helm 版本信息

Helm 3 部署完成后,需要安装 Helm Chart 库,命令如下:

```
#添加微软、阿里云 Chart 库
sudo helm repo add stable http://mirror.azure.cn/kubernetes/charts
sudo helm repo add aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
#更新 Chart 库
sudo helm repo update
```

命令执行的过程及 Chart 库添加成功的标识如图 3-52 所示。

```
user01@node01:~$ sudo helm repo add stable http://mirror.azure.cn/kubernetes/charts
"stable" has been added to your repositories
user01@node01:~$ sudo helm repo add aliyun https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
"aliyun" has been added to your repositories
user01@node01:~$ sudo helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "aliyun" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈Happy Helming!⎈
```

图 3-52　添加 Chart 库

此时就可以查询所需的应用 Chart 信息,例如,搜索常用的 ingress 应用,命令如下:

```
sudo helm search repo ingress
```

命令执行后可以显示仓库内 ingress 应用的 Chart 版本信息,如图 3-53 所示。

```
user01@node01:~$ sudo helm search repo ingress
NAME                                    CHART VERSION   APP VERSION     DESCRIPTION
aliyun/nginx-ingress                    0.9.5           0.10.2          An nginx Ingress controller that uses ConfigMap...
stable/gce-ingress                      1.2.2           1.4.0           DEPRECATED A GCE Ingress Controller
stable/ingressmonitorcontroller         1.0.50          1.0.47          DEPRECATED - IngressMonitorController chart tha...
stable/nginx-ingress                    1.41.3          v0.34.1         DEPRECATED! An nginx Ingress controller that us...
aliyun/external-dns                     0.4.9           0.4.8           Configure external DNS servers (AWS Route53, Go...
aliyun/lamp                             0.1.4                           Modular and transparent LAMP stack chart suppor...
aliyun/nginx-lego                       0.3.1                           Chart for nginx-ingress-controller and kube-lego
aliyun/traefik                          1.24.1          1.5.3           A Traefik based Kubernetes ingress controller w...
aliyun/voyager                          3.1.0           6.0.0-rc.0      Voyager by AppsCode - Secure Ingress Controller...
stable/contour                          0.2.2           v0.15.0         DEPRECATED Contour Ingress controller for Kuber...
stable/external-dns                     1.8.0           0.5.14          Configure external DNS servers (AWS Route53, Go...
stable/kong                             0.36.7          1.4             DEPRECATED The Cloud-Native Ingress and API-man...
stable/lamp                             1.1.6           7               DEPRECATED - Modular and transparent LAMP stack...
stable/nginx-lego                       0.3.1                           Chart for nginx-ingress-controller and kube-lego
stable/traefik                          1.87.7          1.7.26          DEPRECATED - A Traefik based Kubernetes ingress...
stable/voyager                          3.2.4           6.0.0           DEPRECATED Voyager by AppsCode - Secure Ingress...
```

图 3-53　ingress 应用的 Chart 版本信息

例如查看 stable/nginx-ingress 的 Chart 信息,命令如下:

```
sudo helm show chart stable/nginx-ingress
```

命令执行后详细显示了 stable/nginx-ingress 的相关信息,如图 3-54 所示。

7) 基于 Helm 3 部署 WordPress

首先搜索 WordPress 获取最新稳定版的 Chart 信息,命令如下:

```
#添加官方 Chart 库
sudo helm repo add bitnami https://charts.bitnami.com/bitnami
#更新 Chart 库
sudo helm repo update
#搜索 WordPress 的 Chart 信息
sudo helm search repo wordpress
```

```
user01@node01:~$ sudo helm show chart stable/nginx-ingress
apiVersion: v1
appVersion: v0.34.1
deprecated: true
description: DEPRECATED! An nginx Ingress controller that uses ConfigMap to store
  the nginx configuration.
home: https://github.com/kubernetes/ingress-nginx
icon: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c5/Nginx_logo.svg/500px-Nginx_logo.svg.png
keywords:
- ingress
- nginx
kubeVersion: '>=1.10.0-0'
name: nginx-ingress
sources:
- https://github.com/kubernetes/ingress-nginx
version: 1.41.3
```

图 3-54　nginx-ingress 应用的 Chart 信息

命令执行的过程如图 3-55 所示。

```
user01@node01:~$ helm repo add bitnami https://charts.bitnami.com/bitnami
^C
user01@node01:~$ sudo helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
user01@node01:~$ sudo helm repo list
NAME    URL
stable  http://mirror.azure.cn/kubernetes/charts
aliyun  https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts
bitnami https://charts.bitnami.com/bitnami
user01@node01:~$ sudo helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "aliyun" chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "bitnami" chart repository
Update Complete. ⎈Happy Helming!⎈
user01@node01:~$ sudo helm search repo wordpress
NAME                    CHART VERSION   APP VERSION     DESCRIPTION
aliyun/wordpress        0.8.8           4.9.4           Web publishing platform for building blogs and ...
bitnami/wordpress       23.1.24         6.6.2           WordPress is the world's most popular blogging ...
bitnami/wordpress-intel 2.1.31          6.1.1           DEPRECATED WordPress for Intel is the most popu...
stable/wordpress        9.0.3           5.3.2           DEPRECATED Web publishing platform for building...
```

图 3-55　获取 WordPress 信息

其次，获取 stable/wordpress 资源包，命令如下：

```
#获取 WordPress 资源包
sudo helm pull bitnami/wordpress --version 23.1.24
#解压 WordPress 资源包
tar zxvf wordpress-23.1.24.tgz
```

编辑解压后 wordpress 文件夹内的 values.yaml 文件，设置登录名、密码等相关信息，代码如下：

```
#将 WordPress 的服务类型设置为 NodePort 并将服务器端口自定义为 30080/TCP(在 values.yaml 文
#件的第 553～574 行)
service:
  #@param service.type WordPress service type
  #
  #type: LoadBalancer
```

```
  type: NodePort
  ## @param service.ports.http WordPress service HTTP port
  ## @param service.ports.https WordPress service HTTPS port
  ##
  ports:
    http: 80
    https: 443
  ## @param service.httpsTargetPort Target port for HTTPS
  ##
  httpsTargetPort: https
  ## Node ports to expose
  ## @param service.nodePorts.http Node port for HTTP
  ## @param service.nodePorts.https Node port for HTTPS
  ## NOTE: choose port between <30000-32767>
  ##
  nodePorts:
    http: "30080"
    https: "30443"
## 配置 WordPress 管理后台的用户名和密码
wordpressUsername: admin
wordpressPassword: "Admin123"
## 关闭 WordPress 数据持久化存储功能(在 values.yaml 文件的第 845 行)
persistence:
  ## @param persistence.enabled Enable persistence using Persistent Volume Claims
  ##
  enabled: false
## 关闭 MariaDB 数据持久化存储功能(在 values.yaml 文件的第 1258 行)
  primary:
    ## MariaDB Primary Persistence parameters
    ## ref: https://kubernetes.io/docs/concepts/storage/persistent-volumes/
    ## @param mariadb.primary.persistence.enabled Enable persistence on MariaDB using PVC(s)
    ## @param mariadb.primary.persistence.storageClass Persistent Volume storage class
    ## @param mariadb.primary.persistence.accessModes [array] Persistent Volume access modes
    ## @param mariadb.primary.persistence.size Persistent Volume size
    ##
    persistence:
      enabled: false
      storageClass: ""
      accessModes:
        - ReadWriteOnce
      size: 8Gi
```

注意：

(1) 只展示 values.yaml 文件内修改的参数。

(2) 在默认情况下 values.yaml 内开启了 MariaDB、WordPress 的数据持久化存储功

能，在演示环境内暂时关闭，只需将 enabled 字段的参数设置为 false。

（3）后续章节会详细地讲解 Kubernetes 集群中的数据持久化相关技术细节。

保存 values.yaml 配置文件并部署应用，命令如下：

```
sudo helm install myblog wordpress/ -f wordpress/values.yaml
```

命令执行的过程如图 3-56 所示。

```
user01@node01:~$ sudo helm install myblog wordpress/ -f wordpress/values.yaml
NAME: myblog
LAST DEPLOYED: Fri Oct 25 01:24:13 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: wordpress
CHART VERSION: 23.1.24
APP VERSION: 6.6.2

** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within your cluster:

    myblog-wordpress.default.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

   export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services myblog-wordpress)
   export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
   echo "WordPress URL: http://$NODE_IP:$NODE_PORT/"
   echo "WordPress Admin URL: http://$NODE_IP:$NODE_PORT/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

  echo Username: admin
  echo Password: $(kubectl get secret --namespace default myblog-wordpress -o jsonpath="{.data.wordpress-password}" | base64 -d)
```

图 3-56　部署 WordPress

命令执行后查看 Pods、svc 状态，命令如下：

```
#查看 Pods 状态
sudo kubectl get pods
#查看 svc 状态
sudo kubectl get svc
```

注意：初始化 Pods 的时间取决于集群可用资源情况、依赖镜像下载时间等。

当 WordPress 相关组件运行成功后，使用浏览器访问集群任意节点的 30080 端口即可访问已部署的 WordPress 站点，如图 3-57 所示。

如果需登录到 WordPress 后台，则可访问 http://192.168.79.191:30080/admin，如图 3-58 所示。

输入在 values.yaml 文件内定义的用户名和密码后单击 Login 按钮完成登录，如图 3-59 所示。

此时就可以通过管理后台管理专属的博客站点了。

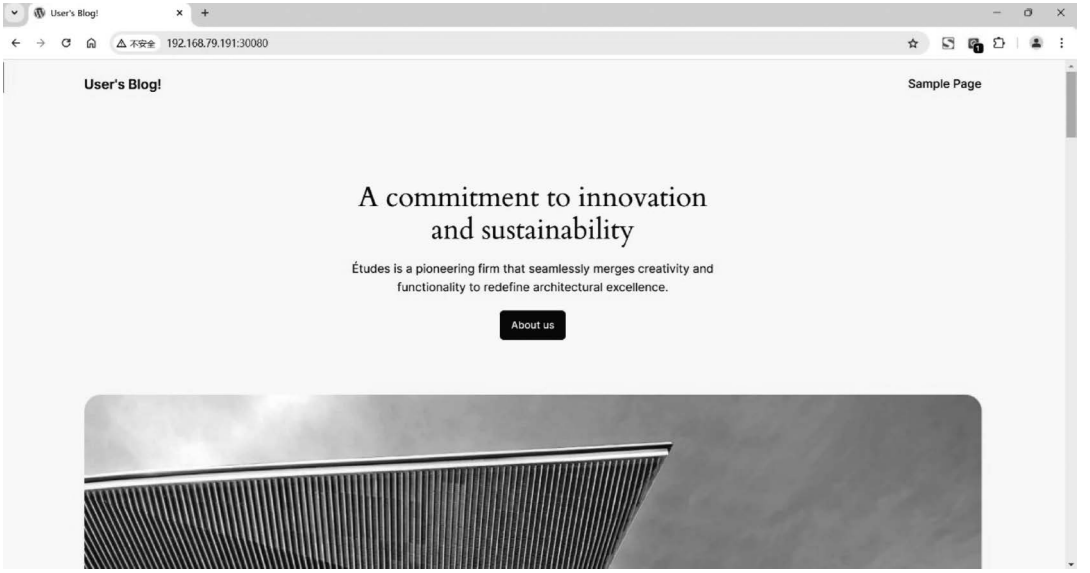

图 3-57　WordPress 初始页面

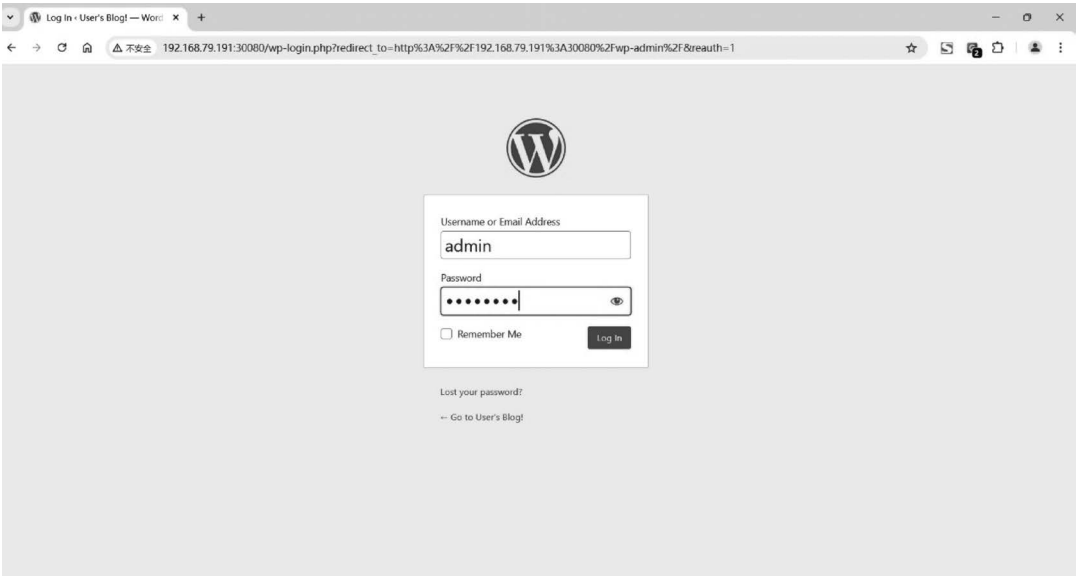

图 3-58　WordPress 后台登录页面

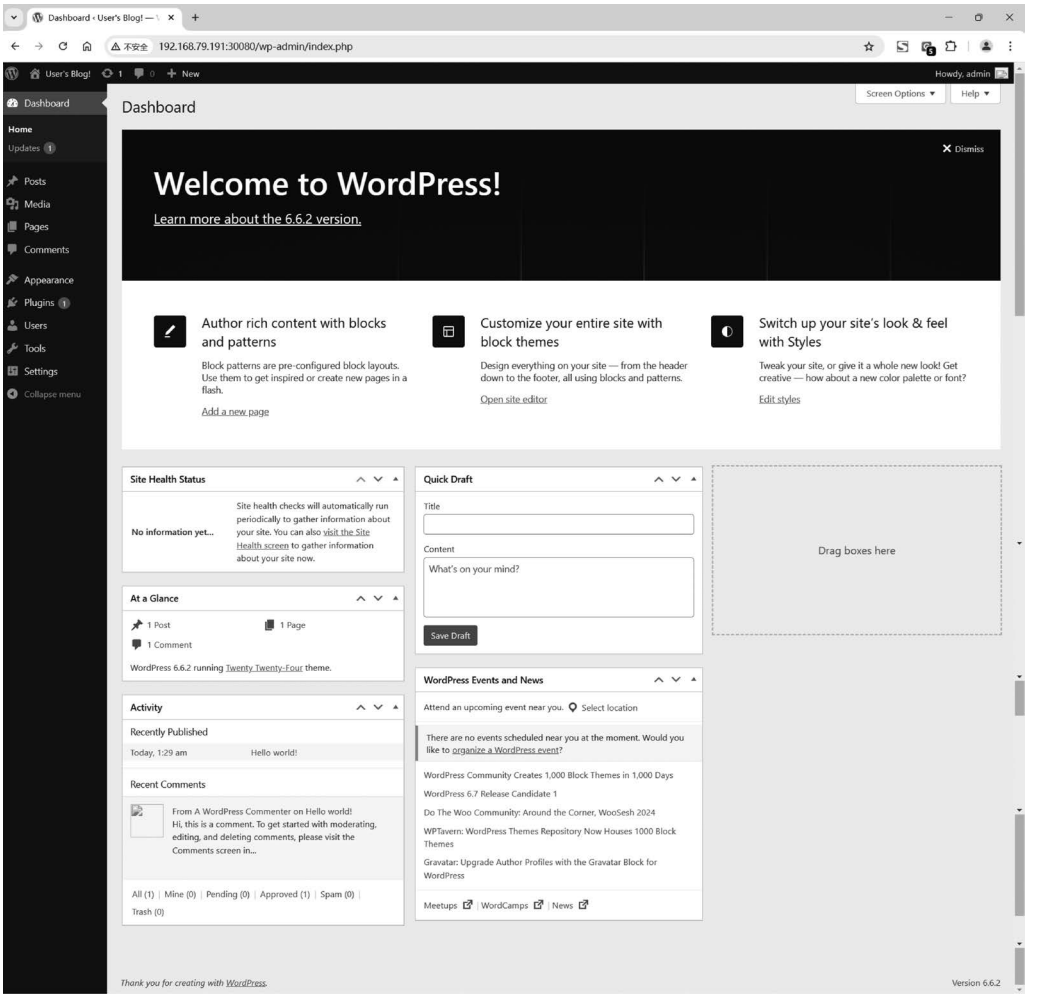

图 3-59　WordPress 管理后台

3.3　基于 Kubernetes 的数据持久化存储管理

在信息化、数字化高速发展的当下，数据存储管理的重要性日益突出。它不仅关乎企业的运营效率和核心竞争力，还会直接影响国家的信息安全和社会稳定。

1. 数据对国家的重要性

在全球化背景下，国与国之间的信息交流日益频繁，对数据的存储和管理是维护国家信息安全的重要手段。通过建立健全的数据存储管理制度，可以确保关键数据不被非法获取、篡改或泄露，从而维护国家的政治、经济、军事等各方面的安全。同时通过优化数据结构、提升数据存储效率，可以进一步地推动各行业、各部门的数字化转型，提升整个国家的数字化

水平。通过对这些数据进行分析和挖掘,国家还可及时发现和解决社会问题,提高社会治理的效率和效果。

2. 数据对企业的重要性

对于企业而言,数据存储管理是业务创新、决策和提高运营效率的关键。尤其是通过分析历史数据,企业可以了解市场趋势、客户需求和竞争态势,为业务创新提供有力支持。同时,数据存储管理还有助于企业实现数据驱动决策,通过数据分析和挖掘,企业可以更加精准地制定战略规划和业务计划,提高决策的科学性和准确性。

3. 集群中的数据为什么要持久化

Kubernetes 集群中的数据持久化存储是集群管理中的重要一环。通常情况下,在基于 Kubernetes 集群发布应用时,如果没有自定义数据存储方式,则默认情况下应用产生的数据直接存储在容器中,而容器是临时的,当容器被重新调度、重启或删除时,容器内的数据也会随之被删除。为了便于容器被重新调度或重建后能够继续访问已产生的数据,因此必须对数据进行持久化存储处理。同时,一旦数据做了持久化存储处理,那么在 Kubernetes 集群中多个 Pod 彼此之间共享数据就成为可能,还可以在系统发生宕机时提供数据恢复能力,从而确保了数据的完整性,以及服务的连贯性。

4. 数据生命周期管理

在企业生产活动中数据生命周期管理大致可以分为数据产生、数据存储、数据使用和数据销毁等阶段。首先数据产生是数据生命周期的起点,在 Kubernetes 集群中数据的产生既可能是 Pod 内的容器生成,也可能是外部数据的导入。无论以何种方式产生数据,都必须确保数据与当前应用需求规范相适配,即需要对数据进行校验,例如导入原始数据等。

其次是数据的存储,在 Kubernetes 集群集中需要考虑数据的持久化存储方式、存储的可靠性、可扩展性和安全性等多种因素。尤其是需要依据实际需求选择合适的存储解决方案,进一步确保数据的质量和存储效率,例如对于需要读写频繁和高并发访问的数据,建议选择高性能的固态硬盘存储;对于需要长期保存的存档数据则可以选择成本较低的机械硬盘存储等。

再次是数据的使用,它是数据生命周期中的关键阶段。在 Kubernetes 集群中,数据可能会被多个 Pod 或外部应用访问并使用,这就需要对数据有效地进行组织和管理,从而提高数据的使用效率,例如,建立数据索引和缓存机制,提高数据的访问速度;对数据合理地进行分区和分片,优化数据查询和更新操作。当然也可以通过建立数据访问的权限控制机制来提升数据的安全性和隐私性。

然后是数据更新和维护,它是数据生命周期的持续性阶段。在 Kubernetes 集群中,数据会随着时间的推移而不断变化,这就需要对数据及时地进行维护和更新,从而确保数据的准确性和一致性。同时还需要对数据进行定期清理、合并和归档,例如,定期删除无效或过期数据,保持数据的整洁性。

最后是数据的销毁,也是数据生命周期的终点。在 Kubernetes 集群中,对于确认不需

要的数据可以进行销毁，例如，过期的日志文件等。一方面可以起到保护数据安全性和隐私性的作用，另一方面可以提升存储的有效使用率。

3.3.1　数据持久化存储方案介绍

Kubernetes 集群中的数据持久化是采用卷存储的方式来实现的，而卷存储有很多种针对不同场景下的解决方案。在实际应用过程中，在选择合适的解决方案时需要考虑应用程序的需求、性能需求、可靠性要求及 Kubernetes 集群所处的平台环境等众多因素。在企业生产活动中典型的卷存储方案如下。

1. NFS(Network File System)卷持久化存储方案

在 Kubernetes 集群中，可以使用 NFS 作为持久化存储卷，将数据存储在 NFS 服务器上并将其挂载到 Pod 中，从而达到数据持久化存储和数据共享的目的，该方案的适用场景及方案优缺点见表 3-22。

表 3-22　NFS 卷持久化存储方案

适 用 场 景	优　　点	缺　　点
数据存储在 Kubernetes 集群之外或者使用类似文件系统的应用场景，例如多 Pod 通过共享存储卷的方式读写文件等	NFS 是一种非常成熟的网络文件系统，具有部署简单、维护成本低等优点。可以非常方便地在 Kubernetes 集群内实现数据共享	NFS 的性能相对低，不太适合高性能场景。同时 NFS 的可靠性和稳定性依赖于底层的网络架构和存储设备，一旦底层设备出现故障就有可能会导致数据丢失

2. hostPath 卷持久化存储方案

hostPath 是一种简单的持久化存储解决方案，它将数据存储在宿主机的本地文件系统中，并将其挂载至 Pod 中，从而实现数据的持久化存储，该方案的适用场景及方案优缺点见表 3-23。

表 3-23　hostPath 卷持久化存储方案

使 用 场 景	优　　点	缺　　点
使用 hostPath 方式可以快速、方便地将数据存储在宿主机节点，避免配置和管理外置存储的复杂过程，同时还能够提供较高的性能和较低的延迟，因此被广泛地应用于开发、测试或临时数据存储等场景，例如缓存数据、日志文件等	简单易用，在 Kubernetes 集群中只需在定义 Pod 时指定宿主机路径。同时由于是直接存储在宿主机内，所以可以提供较高的性能和较低的延迟	数据只存储在宿主机节点，缺乏数据冗余和高可用，并且无法实现数据在不同节点之间进行共享访问

3. cephfs 卷持久化存储方案

该方案后端存储采用的是分布式存储系统 ceph，它具有开源性、高可用性、可扩展性、数据高一致性及易管理等特点，被企业广泛地应用于大规模数据存储，该方案的适用场景及方案优缺点见表 3-24。

表 3-24 cephfs 卷持久化存储方案

适 用 场 景	优 点	缺 点
适用于大规模集群中的数据存储	具有高可靠性、可扩展性及动态调整存储资源的能力	配置复杂

注意：CephFS 卷插件在 Kubernetes 1.28 版本中正式被标记为已弃用，在 Kubernetes 1.31 版本中移除，但这并不意味着不可以使用 Ceph 作为后端存储，而是需要使用 CephFS CSI 第三方驱动来与 Ceph 集群进行数据通信。

4. emptyDir 卷持久化存储方案

emptyDir 是 Kubernetes 中最简单的存储方式，它是一个临时目录，只在 Pod 的生命周期内存在。通常情况下 emptyDir 卷用于在同一 Pod 的不同容器之间共享数据，需要特别注意的是初始状态卷是空的，当 Pod 被删除或重新调度时，emptyDir 卷内的数据会被删除。该方案的适用场景及方案优缺点见表 3-25。

表 3-25 emptyDir 卷持久化存储方案

适 用 场 景	优 点	缺 点
主要适用于缓存数据或者存储临时文件	访问数据速度快，并且由于 emptyDir 是 Kubernetes 的一种原生存储解决方案，因此在创建 Pod 时可以很容易地进行配置及使用	数据的临时性，由于 emptyDir 存储的生命周期是与 Pod 绑定的，所以当 Pod 重启、删除、迁移时都会导致数据丢失，其次 emptyDir 存储的路径是在宿主机本地节点上，因此不适合存储大量数据。最后是单点故障，如果 Pod 所在的宿主机节点发生故障，则该节点上采用 emptyDir 方式存储的数据就会丢失

5. fc(光纤通道)卷持久化存储方案

该方案是通过光纤通道的方式将存储卷挂载至 Pod，从而实现数据的持久化存储，该方案的适用场景及方案优缺点见表 3-26。

表 3-26 fc 卷持久化存储方案

适 用 场 景	优 点	缺 点
由于 fc(光纤通道)具有低延迟和高吞吐量等特性，因此适用于需要快速数据访问、数据存储等对访问、存储性能敏感的应用，例如大规模数据库、数据分析和机器学习等场景	fc 存储具有高性能、高可靠、可扩展及多路径冗余等优点	相对其他类型的存储，fc 存储方案成本高、对设备的依赖性强

6. iSCSI 卷持久化存储方案

iSCSI(Internet Small Computer System Interface)是一种通过网络连接的存储协议，可以将远程存储设备映射为本地磁盘。在 Kubernetes 集群中，可以使用 iSCSI(基于 IP 的

SCSI)作为持久化存储卷,将远程存储设备映射到 Pod 中。该方案的适用场景及方案优缺点见表 3-27。

表 3-27　iSCSI 卷持久化存储方案

使 用 场 景	优　　点	缺　　点
高可用性场景：iSCSI 存储可以通过配置多路径的方式提升持久化存储数据的高可用性。同时 iSCSI 存储还支持数据保护功能,例如快照、数据复制等,从而提升了数据的可靠性,因此还适用于对数据保护有要求的场景	iSCSI 存储可以使用标准以太网网络进行连接,无需额外的硬件设备,就可以在现有基础设施中部署和扩展。还可以通过千兆以太网或万兆以太网提升 iSCSI 存储的性能。同时,由于 iSCSI 是一种标准协议,因此可以与各种操作系统和存储产品进行互操作,跨平台的兼容性强	对网络的依赖性强及配置相对较为复杂

7. local 卷持久化存储方案

该方案中数据被挂载至本地存储设备,例如磁盘、分区或目录。适用于需要临时性存储或对性能要求较高的应用场景,需要注意的是 local 卷只能用于静态创建的卷持久化存储,不支持动态配置。该方案的适用场景及方案优缺点见表 3-28。

表 3-28　local 卷持久化存储方案

适 用 场 景	优　　点	缺　　点
需要高性能与低延迟的应用需求或者需要本地存储的应用等	具有高性能、低延迟和具有独立存储的优点	具有单点故障,由于数据直接存储在本地节点,所以存储容量、数据迁移等都会受到限制

3.3.2　持久卷介绍

在 Kubernetes 集群中,容器与 Pod 的生命周期是相同的。当 Pod 被删除时,容器内的数据也会随之被删除,然而,在企业实际生产活动中,需要持久化存储这些数据,以便在 Pod 被删除或者重新调度后仍然可以访问这些数据,实现的方法是使用持久卷的方式存储这些数据,这就涉及以下两个重要概念。

1. 持久卷

持久卷(Persistent Volume,PV)是 Kubernetes 集群中的存储资源,与普通的卷一样,持久卷也是通过插件来实现的。只是 PV 拥有独立于任何使用 PV 资源的 Pod 的生命周期,即当使用 PV 来存储 Pod 数据后,无论 Pod 是被删除、重启或重新调度,这些数据都不但不会丢失,并且 Pod 还可以继续访问之前的数据。

PV 的创建方式可以通过静态或动态的方式完成,它们之间的差异在于静态方式需要管理员预先在集群中创建 PV,使用者通过持久卷申领(Persistent Volume Claim,PVC)的方式来申请预设的 PV 资源,而动态方式则是由 Kubernetes 提供的存储插件根据 PVC 的需求动态地创建并绑定 PV 资源,在使用过程中 PV 的常见状态见表 3-29。

表 3-29 PV 的常见状态

状　态	说　明
可用(Available)	表示 PV 未被任何 PVC 绑定,等待被绑定
已绑定(Bound)	表示 PV 已经被绑定到 PVC
已释放(Released)	表示 PVC 被删除,但 PV 的资源尚未被集群回收
失败(Failed)	表示失败,可能是由于资源不足、配置错误等原因造成的

当前的 Kubernetes 1.31 版本支持的 PV 持久卷插件见表 3-30。

表 3-30 PV 持久卷插件

插件类型	说　明
csi	容器存储接口(CSI)
fc	Fibre Channel(FC)存储
hostPath	HostPath 卷(通常用于单节点测试使用,不适用于集群环境)
iscsi	iSCSI 存储
local	挂载节点本地存储
nfs	NFS 存储

注意:需要特别注意的是,随着 Kubernetes 版本的不断迭代更新和新技术的兴起,所支持的存储插件会有所变化,因此在使用前强烈建议查看官方文档。

以 PV 支持的 NFS 存储为例,当在集群中需要手动创建 PV 时,代码如下:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001
spec:
  capacity:
    #配置限额
    storage: 5Gi
  volumeMode: Filesystem
  #配置访问模式
  accessModes:
    - ReadWriteOnce
  #配置资源回收策略
  persistentVolumeReclaimPolicy: Retain
  storageClassName: network-nfs
  #配置 nfs 共享目录和服务器地址
  nfs:
    path: /nfs-share
    server: 192.168.79.191
```

代码中 storage 字段用于定义 PV 资源的容量,volumeMode 字段用于定义 PV 卷模式,当前 Kubernetes 集群支持两种卷模式,分别是文件系统(Filesystem)和块(Block),两者之

间的差异见表 3-31。

表 3-31　卷模式

类　　型	Filesystem	Block
使用方法	将存储卷作为文件系统挂载至 Pod 中的某个目录	将存储卷作为块设备暴露给 Pod
适用场景	适用于大多数需要持久化的场景，尤其是需要将数据存储在文件系统中的场景	适用于需要直接访问块设备的场景，例如数据库或高性能计算
工作原理	当 Pod 访问 Filesystem 模式的 PV 时，它会看到一个挂载到特定目录的文件系统，此时 Pod 可以在这个文件系统上创建、读取、写入和删除文件	当 Pod 访问 Block 模式的 PV 时，它会看到一个块设备。Pod 可以通过文件系统工具（例如 mkfs）在块设备上创建文件系统，或者直接通过原始块 I/O 访问数据
特点	易用且支持多种文件系统（例如 ext4、xfs 等）	高性能，通常情况下块设备比文件系统具有更高的 I/O 性能

代码中 accessModes 字段定义了访问指定 PV 的访问模式，在 Kubernetes 集群中访问 PV 的模式见表 3-32。

表 3-32　集群中访问 PV 的模式

访 问 模 式	说　　明	适 用 场 景
ReadWriteOnce(RWO)	PV 可以被单个节点以读写模式挂载	适用于同一节点上的多个 Pod 访问挂载卷
ReadOnlyMany(ROX)	PV 可以被多个节点以只读模式挂载	适用于以只读方式访问共享数据
ReadWriteMany(RWX)	PV 可以被多个节点以读写模式挂载	适用于同时从多个节点读写数据
ReadWriteOncePod(RWOP)	PV 可以被单个 Pod 以读写方式挂载	适用于在整个集群中只有一个 Pod 可以读取或写入该 PV

代码中字段 persistentVolumeReclaimPolicy 用于设置 PV 的资源回收策略，在 Kubernetes 集群中 PV 资源的回收策略见表 3-33。

表 3-33　PV 资源的回收策略

回 收 策 略	说　　明	适 用 场 景
Retain(保留)	当将 PV 的回收策略设置为 Retain 时，即使对应的 PVC 被删除，PV 也不会自动被删除，此时 PV 的状态进入 Released 状态，表示该 PV 已被释放但未被回收	保留 PV 中的数据
Delete(删除)	当将 PV 的回收策略设置为 Delete 时，当 PVC 被删除后，对应的 PV 也会被自动删除	不保留数据或者数据可以重新生成
Recycle(回收，已弃用)	Recycle 策略原本的意图是自动清理 PV 中的数据（类似于执行 rm -rf /thevolume/*），然后使 PV 变为 Available 状态，可以被新的 PVC 重新绑定，然而，由于 Recycle 策略存在问题和限制，它已被 Kubernetes 官方弃用	替代方案建议使用 Delete 策略或者手动删除数据

2. 持久卷声明

持久卷声明(Persistent Volume Claim,PVC)是用户对存储资源的请求,需要注意的是PVC在声明时会消耗PV资源,同时PVC可以请求特定的存储空间和自定义访问模式,例如定义PVC访问基于NFS创建的pv001,代码如下:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc001
spec:
  #必须与pv001中的accessModes匹配
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
     #必须与pv001中的capacity匹配
     storage: 5Gi
  storageClassName: network-nfs
```

在企业环境中部分应用需要低延迟、高I/O,例如数据库系统等。在Kubernetes集群中持久卷类型往往采用块方式,典型的PV与PVC应用的代码如下:

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  #光纤通道参数配置
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
---
piVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
```

```
  resources:
    requests:
      storage: 10Gi
```

注意：代码中字段 targetWWNs 和 LUN 的说明如下。

(1) targetWWNs 是 World Wide Names 的缩写，它指的是 Fibre Channel 存储设备的目标全球唯一名称，用于唯一标识存储设备上的目标，客户端通过这些目标来访问存储设备。

(2) LUN(Logical Unit Number)是逻辑单元号的缩写，它是 Fibre Channel 存储设备上的一个逻辑卷或分区。每个 LUN 都有一个唯一的编号，用于在存储设备上标识不同的逻辑存储单元。

3.3.3　存储类(StorageClass)介绍

在 Kubernetes 集群中，StorageClass 提供了一种声明式的方法来描述存储的类型，这些不同的类型会被映射到不同的服务等级或备份策略。同时，StorageClass 还允许自定义不同类型的存储，让开发者选择自身需要的存储类型，而不用关注底层的存储实现细节，例如，开发者可以通过 PVC 指定所需的 StorageClass 后，Kubernetes 集群自动创建相应的 PV 并绑定到 PVC 上，从而简化了存储配置的过程，提升了管理效率。

每个 StorageClass 都会包含 provisioner、parameters 和 reclaimPolicy 字段，其中字段 provisioner 用于关联创建 PV 所使用的卷插件，例如 AzureFile、RBD 等。字段 parameters 用于配置存储的具体属性，例如类型、大小、IOPS 等。字段 reclaimPolicy 用于指定资源的回收策略，可以是 Delete 或者 Retain。需要注意的是，如果字段 reclaimPolicy 在 StorageClass 对象被创建时未指定，则它的默认值为 Delete。典型的 StorageClass 的示例代码如下：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: low-latency
  annotations:
    storageclass.kubernetes.io/is-default-class: "false"
#定义资源提供者
provisioner: csi-driver.example-vendor.example
#设置回收策略,默认值为 Delete
reclaimPolicy: Retain
#设置是否支持卷扩展
allowVolumeExpansion: true
#挂载选项
mountOptions:
  - discard
#绑定模式
volumeBindingMode: WaitForFirstConsumer
#设置参数
```

```
parameters:
  guaranteedReadWriteLatency: "true"
```

注意：

(1) 在使用挂载选项(mountOptions)时需要特别注意，如果卷插件不支持挂载选项，但是在代码中又定义了挂载选项，则创建 PV 会失败。同时还需要注意，在定义 StorageClass 内的挂载选项时，如果其中有一个挂载选项是无效的，则这个 PV 挂载操作就会失败。

(2) 字段 volumeBindingMode 用于控制何时将 PV 绑定到 PVC，例如，立即绑定(Immediate)、等待延迟绑定(WaitForFirstConsumer)，默认值为立即绑定。

3.3.4 数据持久化应用实战

在企业环境下 Kubernetes 集群内应用数据的重要性不言而喻，如何稳定、高效、可靠地存储数据是最关键的业务之一。接下来将以企业使用最广泛的网络存储 NFS 为示例来演示数据持久化中最关键的技术点，演示环境集群信息见表 3-34。

表 3-34 演示环境集群信息

主 机 名	IP 地址	说 明
node01	192.168.79.191	(1) 集群管理节点(控制平面节点) (2) 部署 NFS 服务器
node03	192.168.79.193	集群工作节点
node04	192.168.79.194	集群工作节点

注意：通常情况下在企业生产环境中 NFS 服务器使用单独的节点部署，不建议直接在集群节点内部署，其原因是避免了与集群中其他应用或服务争夺 CPU、内存、存储空间、网络等资源，同时还可以更好地对 NFS 服务器进行管理和优化，确保数据传输效率。

演示案例的详细需求说明见表 3-35。

表 3-35 需求说明

需 求	说 明
项目背景	Kubernetes 集群中应用数据持久化存储
项目要求	(1) 采用 Apache Http 作为 Web 服务前端 (2) 应用的副本数为 3 (3) 服务对外暴露的端口为 30080/TCP (4) 应用所属命名空间为 default (5) 需要将 Apache Http 服务/usr/local/apache2/htdocs/ 路径下的数据持久化存储
存储方案	采用 NFS

下面将通过持久卷和存储类的方式来详细展示在 Kubernetes 集群中数据持久化的关

键技术点和相关实现代码。

1. NFS 环境准备

登录 node01 节点部署 NFS 服务器，首先创建演示环境所需的数据存储目录并设置权限，命令如下：

```
sudo mkdir /{nfs-share01,nfs-share02}
sudo chmod 777 /nfs-share02
```

其次安装 NFS 服务器端相关依赖包，命令如下：

```
#更新索引
sudo apt update
#安装 NFS 服务器端
sudo apt install nfs-kernel-server nfs-common -y
```

安装完成后，在配置文件/etc/exports 内添加 NFS 服务器端的共享目录及访问权配置，代码如下：

```
/nfs-share01 *(rw,sync,no_root_squash,no_subtree_check)
/nfs-share02 *(rw,sync,no_root_squash,no_subtree_check)
```

注意：nfs server 服务器端配置文件/etc/exports 内的典型参数说明如下。

(1) /nfs-share01 表示共享目录。

(2) 星号(*)，表示任意地址。

(3) rw 表示共享目录具有读写权限。

(4) sync 表示数据同步写入磁盘与内存。

(5) no_subtree_check 表示不检查子目录权限。

保存并退出编辑器后重启 nfs server 服务，命令如下：

```
#重启 nfs server 服务
sudo systemctl restart nfs-kernel-server
#设置 nfs server 服务自启动
sudo systemctl enable nfs-kernel-server
#查看 nfs server 服务状态
sudo systemctl status nfs-kernel-server
```

接着在 NFS 服务器端验证服务的可用性，命令如下：

```
#列出共享目录
sudo exportfs -v
#或者
sudo showmount -e 192.168.79.191
```

命令执行的过程如图 3-60 所示。

此时的状态表示 NFS 服务器端环境运行成功，等待客户端挂载。接着集群中各节点需

```
user01@node01:~$ sudo systemctl status nfs-kernel-server
● nfs-server.service - NFS server and services
     Loaded: loaded (/lib/systemd/system/nfs-server.service; enabled; vendor preset: enabled)
    Drop-In: /run/systemd/generator/nfs-server.service.d
             └─order-with-mounts.conf
     Active: active (exited) since Wed 2024-10-30 02:19:06 UTC; 5min ago
    Process: 18047 ExecStartPre=/usr/sbin/exportfs -r (code=exited, status=0/SUCCESS)
    Process: 18048 ExecStart=/usr/sbin/rpc.nfsd (code=exited, status=0/SUCCESS)
   Main PID: 18048 (code=exited, status=0/SUCCESS)
        CPU: 14ms

Oct 30 02:19:06 node01 systemd[1]: Starting NFS server and services...
Oct 30 02:19:06 node01 systemd[1]: Finished NFS server and services.
user01@node01:~$ sudo exportfs -v
/nfs-share01    <world>(sync,wdelay,hide,no_subtree_check,sec=sys,rw,secure,no_root_squash,no_all_squash)
/nfs-share02    <world>(sync,wdelay,hide,no_subtree_check,sec=sys,rw,secure,no_root_squash,no_all_squash)
user01@node01:~$ sudo showmount -e 192.168.79.191
Export list for 192.168.79.191:
/nfs-share02 *
/nfs-share01 *
```

图 3-60　NFS 服务器端验证

要安装 NFS 服务的客户端,命令如下:

```
#更新索引
sudo apt update
#安装 NFS 服务的客户端
sudo apt install nfs-common -y
```

NFS 客户端部署完成后测试与服务器端的连接,命令如下:

```
sudo showmount -e 192.168.79.191
```

如果命令执行后的显示结果如图 3-61 所示,则表示客户端与服务器端通信正常,客户端可以挂载服务器端的共享目录。

```
user01@node04:~$ sudo showmount -e 192.168.79.191
Export list for 192.168.79.191:
/nfs-share02 *
/nfs-share01 *
```

图 3-61　客户端与服务器端连接测试

为了测试的直观性,在共享目录/nfs-share01 内创建测试页面 index.html,代码如下:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>nfs server 测试</title>
    </head>
    <body>
        <div style="text-align: center;">
            <h1>基于 NFS 持久化存储 -- 测试页面!</h1>
        </div>
    </body>
</html>
```

至此，NFS服务器端和客户端相关环境准备完成，接下来将会展示如何将 Kubernetes 内的应用数据持久化存储在 NFS 共享目录内。

2. 持久卷方案

首先在文件 myapp-pv.yaml 内将 PV 的名称定义为 nfs-pv01，容量为 2GB，卷模式为 Filesystem，访问模式为 ReadWriteMany，回收策略为保留数据，NFS 服务器地址为 192.168.79.131，共享目录为/nfs-share01，myapp-pv.yaml 文件中的代码如下：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv01
spec:
  capacity:
    storage: 2Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /nfs-share01
    server: 192.168.79.191
```

其次再定义持久卷声明 myapp-pvc.yaml，代码如下：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc01
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
```

然后部署 PV、PVC 并验证绑定是否成功，命令如下：

```
#部署 PV
sudo kubectl apply -f myapp-pv.yaml
#查看 PV 状态
sudo kubectl get pv
#部署 PVC
sudo kubectl apply -f myapp-pvc.yaml
#查看 PVC 状态
sudo kubectl get pvc
```

命令执行的过程及信息显示如图 3-62 所示。

```
user01@node01:~$ sudo kubectl apply -f myapp-pv.yaml
persistentvolume/nfs-pv01 created
user01@node01:~$ sudo kubectl get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
nfs-pv01   2Gi        RWX            Retain           Available                          <unset>                          11s
user01@node01:~$ sudo kubectl apply -f myapp-pvc.yaml
persistentvolumeclaim/nfs-pvc01 created
user01@node01:~$ sudo kubectl get pvc
NAME        STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
nfs-pvc01   Bound    nfs-pv01   2Gi        RWX                           <unset>                 9s
```

图 3-62　PVC 成功绑定 PV

此时，如果 PVC 的状态信息内显示绑定的卷为预定义的 nfs-pv01，则表示两者绑定成功。然后编写测试应用 myapp-nfs.yaml 并关联已绑定成功的 nfs-pvc01，代码如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: myapp-nfs
  name: myapp-nfs
  namespace: default
spec:
  #设置副本数
  replicas: 2
  selector:
    matchLabels:
      app: myapp-nfs
  template:
    metadata:
      labels:
        app: myapp-nfs
    spec:
      containers:
      - image: httpd:alpine
        name: httpd
        #设置卷挂载
        volumeMounts:
        - name: nfs-vol1
          mountPath: /usr/local/apache2/htdocs/
        ports:
        - containerPort: 80
      #设置卷关联已定义的 PVC
      volumes:
      - persistentVolumeClaim:
          claimName: nfs-pvc01
        name: nfs-vol1
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: myapp-nfs
  name: myapp-nfs
```

```
spec:
  ports:
  - name: 80-80
    nodePort: 30080
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: myapp-nfs
  type: NodePort
```

接着在 Kubernetes 集群中发布创建的测试服务并验证挂载的 NFS 存储是否可用，命令如下：

```
#部署测试服务
sudo kubectl apply -f myapp-nfs.yaml
#查看测试服务的 Pod、服务状态
sudo kubectl get pods
sudo kubectl get svc
```

命令执行的过程如图 3-63 所示。

```
user01@node01:~$ sudo kubectl apply -f myapp-nfs.yaml
deployment.apps/myapp-nfs created
service/myapp-nfs created
user01@node01:~$ sudo kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
myapp-nfs-5776646b48-2p8xt    1/1     Running   0          11s
myapp-nfs-5776646b48-m6zkt    1/1     Running   0          11s
user01@node01:~$ sudo kubectl get svc
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        18d
myapp-nfs    NodePort    10.101.159.24   <none>        80:30080/TCP   19s
```

图 3-63　部署测试服务 myapp-nfs

通过命令执行后输出的信息可以判断测试服务部署成功。一旦测试服务部署成功，就可以使用浏览器或命令行的方式访问集群内任意节点的 30080/TCP 来验证 NFS 共享存储的可用性，如图 3-64 所示。

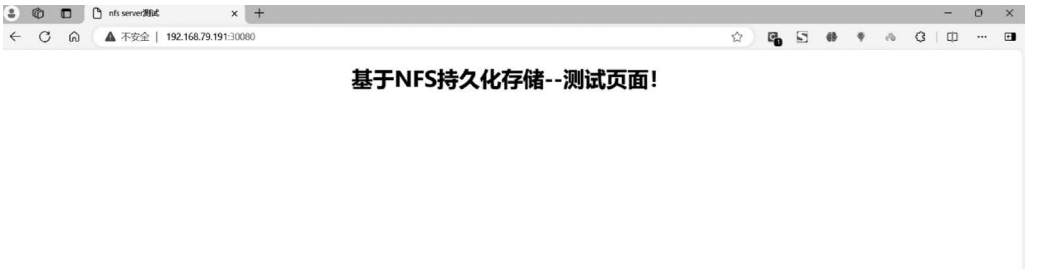

图 3-64　以浏览器方式访问测试页面

注意：在命令行模式下可以使用 lynx 命令进行高效测试，需要注意的是 lynx 是纯文本模式的网页浏览器，不支持音视频等多媒体资源，例如，lynx http://192.168.79.191:30080

最后测试数据是否被持久化存储，在这一测试过程中可以先在/nfs-share01 目录之外创建新的测试文件 index02.html，代码如下：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>nfs server 测试</title>
    </head>
    <body>
        <div style="text-align: center;">
            <h1>基于 NFS 持久化存储 -- 测试页面 V2!</h1>
        </div>
    </body>
</html>
```

接着使用命令获取 myapp-nfs 的 Pod 信息，命令如下：

```
#获取 Pod 信息
sudo kubectl get pods -o wide
#将当前节点内的 index02.html 复制至任意 Pod 内
sudo kubectl cp index02.html default/myapp-nfs-5776646b48-2p8xt:/usr/local/apache2/
htdocs/
```

命令执行的过程如图 3-65 所示。

```
user01@node01:~$ sudo kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP               NODE     NOMINATED NODE   READINESS GATES
myapp-nfs-5776646b48-2p8xt   1/1     Running   0          66m   10.244.186.212   node03   <none>           <none>
myapp-nfs-5776646b48-m6zkt   1/1     Running   0          66m   10.244.248.219   node04   <none>           <none>
user01@node01:~$ sudo kubectl cp index02.html default/myapp-nfs-5776646b48-2p8xt:/usr/local/apache2/htdocs/
```

图 3-65　将文件复制至 Pod

文件复制完成后可以使用浏览器访问 index02.html 测试页，以便验证测试页面的可用性。如果通过浏览器方式可以看到测试页面 index02.html 的内容，则表明测试页面 index02.html 可用，如图 3-66 所示。

图 3-66　测试页面 index02

此时查看 NFS 服务器共享目录/nfs-share01 内的文件，就会发现该文件夹内已经存储了新文件 index02.html，验证了集群内应用数据可以持久化存储在 NFS 存储服务器内，如图 3-67 所示。

```
user01@node01:~$ sudo kubectl cp index02.html default/myapp-nfs-5776646b48-2p8xt:/usr/local/apache2/htdocs/
user01@node01:~$
user01@node01:~$ sudo ls /nfs-share01
index.html  index02.html
```

图 3-67　数据实现持久化存储

如果此时删除测试应用 myapp-nfs，并同时删除与之关联的 PVC、PV 后，再查看 NFS 共享目录/nfs-share01 内的文件就会发现已经存储在该目录内的数据依然存在，并未随着应用、PVC、PV 的删除而被清除，实现了数据持久化存储。相关操作命令如下：

```
#删除应用
sudo kubectl delete -f myapp-nfs.yaml
#删除 PVC
sudo kubectl delete -f myapp-pvc.yaml
#删除 PV
sudo kubectl delete -f myapp-pv.yaml
#查看共享目录/nfs-share01 内的文件
ls /nfs-share01
#查看与测试应用相关资源是否被完全删除
sudo kubectl get pvc
sudo kubectl get pv
sudo kubectl get pods
sudo kubectl get svc
```

命令的执行过程及信息输出如图 3-68 所示。

```
user01@node01:~$ sudo kubectl delete -f myapp-nfs.yaml
deployment.apps "myapp-nfs" deleted
service "myapp-nfs" deleted
user01@node01:~$ sudo kubectl delete -f myapp-pvc.yaml
persistentvolumeclaim "nfs-pvc01" deleted
user01@node01:~$ sudo kubectl delete -f myapp-pv.yaml
persistentvolume "nfs-pv01" deleted
user01@node01:~$ sudo kubectl get pvc
No resources found in default namespace.
user01@node01:~$ sudo kubectl get pv
No resources found
user01@node01:~$ ls /nfs-share01/
index.html  index02.html
user01@node01:~$ sudo kubectl get pods
No resources found in default namespace.
user01@node01:~$ sudo kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   18d
```

图 3-68　验证数据持久化存储

通过 ls 命令查看共享目录的数据文件可以验证数据已经被持久化存储。如果需要在应用中重新启用原数据，则只需重新挂载 NFS 服务器端共享目录/nfs-share01 即可。

3. 存储类方案

由于采用静态的方式构建基于 nfs server 的持久化存储时,需要提前创建目录并定义 PV 的大小等信息以供 PVC 绑定。因此在实际生产环境中,还可以使用存储类(StorageClass)的方式动态地创建和管理存储资源,以便提升资源的利用率。由于 Kubernetes 没有内置的 NFS 驱动,因此需要使用外部驱动创建 StorageClass,目前 Kubernetes 官方提供了两种外部 NFS 驱动,分别是 NFS Ganesha 服务器和其外部驱动及 NFS subdir 外部驱动。以下将演示基于 NFS subdir 外部驱动的方式实现动态存储的具体步骤。

首先在 NFS 共享目录/nfs-share02 目录内创建测试文件 index.html,代码如下:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>nfs server 测试</title>
        </head>
        <body>
                <div style="text-align: center;">
                        <h1>StorageClass 功能测试页面!</h1>
                </div>
        </body>
</html>
```

其次,在管理节点 node01 上下载第三方驱动 NFS Subdir External Provisioner,命令如下:

```
sudo wget -O master.zip
https://codeload.github.com/kubernetes-sigs/nfs-subdir-external-provisioner/zip/refs/
heads/master
```

将已下载的 master.zip 文件解压,命令如下:

```
#解压文件
unzip master.zip
```

如果在执行该命令时提示"-bash: unzip: command not found",则只需安装 zip 软件包,命令如下:

```
sudo apt install zip -y
```

master.zip 文件解压后的目录为 nfs-subdir-external-provisioner-master,如图 3-69 所示。

```
user01@node01:~$ ll | grep master
-rw-r--r-- 1 root   root   9131631 Oct 31 00:15 master.zip
drwxrwxr-x 9 user01 user01    4096 Feb 16  2024 nfs-subdir-external-provisioner-master/
```

图 3-69 解压 NFS 第三方驱动文件

对已下载的第三方驱动的相关参数进行修改，命令如下：

```
#将 nfs-subdir-external-provisioner-master 目录内的 deploy 文件夹复制至当前目录
cp -rf nfs-subdir-external-provisioner-master/deploy .
#设置 NS 变量
NS=$(kubectl config get-contexts|grep -e "^\*" |awk '{print $5}')
#赋值变量 NS
NAMESPACE=${NS:-default}
#修改 deploy 文件夹中的配置文件 rbac.yaml,deployment.yaml 内的命名空间名称
sed -i'' "s/namespace:.*/namespace: $NAMESPACE/g" ./deploy/rbac.yaml ./deploy/deployment.yaml
#检查 rbac.yaml 文件内的 namespace 值是否被成功修改
cat ./deploy/rbac.yaml | grep default
#部署
sudo kubectl create -f deploy/rbac.yaml
```

上述命令的执行过程如图 3-70 所示。

```
user01@node01:~$ cp -rf nfs-subdir-external-provisioner-master/deploy .
user01@node01:~$ NS=$(kubectl config get-contexts|grep -e "^\*" |awk '{print $5}')
user01@node01:~$ NAMESPACE=${NS:-default}
user01@node01:~$ sed -i'' "s/namespace:.*/namespace: $NAMESPACE/g" ./deploy/rbac.yaml ./deploy/deployment.yaml
user01@node01:~$ cat ./deploy/rbac.yaml | grep default
  namespace: default
    namespace: default
  namespace: default
  namespace: default
    namespace: default
user01@node01:~$ sudo kubectl create -f deploy/rbac.yaml
[sudo] password for user01:
serviceaccount/nfs-client-provisioner created
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-runner created
clusterrolebinding.rbac.authorization.k8s.io/run-nfs-client-provisioner created
role.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
rolebinding.rbac.authorization.k8s.io/leader-locking-nfs-client-provisioner created
```

图 3-70　应用第三方驱动中的权限控制

然后需要编辑第三方驱动中的 deploy/deployment.yaml 文件，代码如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
  #replace with namespace where provisioner is deployed
  namespace: default
spec:
  replicas: 1
  #定义部署策略
  strategy:
    type: Recreate #采用重建策略,Recreate 表示更新时先销毁旧的 Pod,再重新创建新的 Pod
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
```

```
      metadata:
        labels:
          app: nfs-client-provisioner
      spec:
        serviceAccountName: nfs-client-provisioner
        containers:
          - name: nfs-client-provisioner
            image: docker.io/dyrnq/nfs-subdir-external-provisioner:v4.0.2
            volumeMounts:
              - name: nfs-client-root
                mountPath: /persistentvolumes
            #定义环境变量
            env:
              - name: PROVISIONER_NAME
                value: k8s-sigs.io/nfs-subdir-external-provisioner
              - name: NFS_SERVER #设置 NFS 服务器
                value: 192.168.79.191
              - name: NFS_PATH #设置共享目录
                value: /nfs-share02
        #定义卷
        volumes:
          - name: nfs-client-root
            #卷类型为 NFS
            nfs:
              server: 192.168.79.191
              path: /nfs-share02
```

同时还需要编辑第三方驱动中的 deploy/class.yaml 文件,代码如下:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-client
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner #or choose another name, must
match deployment's env PROVISIONER_NAME'
parameters:
  pathPattern: "${.PVC.namespace}/${.PVC.annotations.nfs.io/storage-path}"
  archiveOnDelete: "false"
```

接着就可以使用第三方驱动提供的测试文件对 StorageClass 的相关功能进行测试,命令如下:

```
#部署测试应用
sudo kubectl create -f deploy/deployment.yaml -f deploy/class.yaml
sudo kubectl create -f deploy/test-claim.yaml -f deploy/test-pod.yaml
#查看 Pod 状态
sudo kubectl get pods
#查看 PV 状态
sudo kubectl get pv
#查看 PVC 状态
```

```
sudo kubectl get pvc
#查看 NFS 服务器端共享目录/nfs-share02 内的数据信息
ls /nfs-share02/default/
```

命令执行的过程如图 3-71 所示。

```
user01@node01:~$ sudo kubectl create -f deploy/deployment.yaml -f deploy/class.yaml
deployment.apps/nfs-client-provisioner created
storageclass.storage.k8s.io/nfs-client created
user01@node01:~$ sudo kubectl create -f deploy/test-claim.yaml -f deploy/test-pod.yaml
persistentvolumeclaim/test-claim created
pod/test-pod created
user01@node01:~$ sudo kubectl get pods
NAME                                     READY   STATUS      RESTARTS   AGE
nfs-client-provisioner-7595b9d99b-7jsnm  1/1     Running     0          8m29s
test-pod                                 0/1     Completed   0          8m16s
user01@node01:~$ sudo kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS   VOLUMEATTRIBUT
ESCLASS   REASON   AGE
pvc-13925c55-46fc-457d-8e8f-4acac75ff12a   1Mi        RWX            Delete           Bound    default/test-claim   nfs-client     <unset>
                  8m23s
user01@node01:~$ sudo kubectl get pvc
NAME         STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
test-claim   Bound    pvc-13925c55-46fc-457d-8e8f-4acac75ff12a   1Mi        RWX            nfs-client     <unset>                 8m27s
user01@node01:~$ ls /nfs-share02/default/
SUCCESS
```

图 3-71　StorageClass 测试过程

基于上述命令及命令执行后输出的信息，可以看到在 NFS 服务器的共享目录/nfs-share02 内创建了测试文件夹 default 及该目录下的 SUCCESS 文件，验证了 StorageClass 在配置文件中的可写功能。

接着可以执行命令删除测试 Pod，命令如下：

```
#删除测试 Pod
sudo kubectl delete -f deploy/test-pod.yaml -f deploy/test-claim.yaml
#查看共享目录内的文件信息
ls /nfs-share02
```

命令执行的过程及信息提示如图 3-72 所示。

```
user01@node01:~$ sudo kubectl delete -f deploy/test-pod.yaml -f deploy/test-claim.yaml
pod "test-pod" deleted
persistentvolumeclaim "test-claim" deleted
user01@node01:~$ ls /nfs-share02/
index.html
```

图 3-72　删除测试 Pod

此时就会发现，测试 Pod 所创建的 default 测试文件被删除，验证了 StorageClass 在配置文件中的删除功能。

在此基础上，就可以部署专属应用并实现应用数据的持久化存储。例如，基于动态卷制备的方式实现演示示例，myapp-nfs-dynamic.yaml 文件中的代码如下：

```
#部署 nfs-client-provisioner
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nfs-client-provisioner
  labels:
    app: nfs-client-provisioner
```

```
  namespace: default
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nfs-client-provisioner
  template:
    metadata:
      labels:
        app: nfs-client-provisioner
    spec:
      serviceAccountName: nfs-client-provisioner
      containers:
        - name: nfs-client-provisioner
          image: docker.io/dyrnq/nfs-subdir-external-provisioner:v4.0.2
          volumeMounts:
            - name: nfs-client-root
              mountPath: /persistentvolumes
          env:
            - name: PROVISIONER_NAME
              value: k8s-sigs.io/nfs-subdir-external-provisioner
            - name: NFS_SERVER
              value: 192.168.79.191
            - name: NFS_PATH
              value: /nfs-share02
      volumes:
        - name: nfs-client-root
          nfs:
            server: 192.168.79.191
            path: /nfs-share02
---
#创建 StorageClass
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs-dynamic
provisioner: k8s-sigs.io/nfs-subdir-external-provisioner
---
#创建 PVC,基于 StorageClass 动态生成 PV
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim-nfs-dynamic
spec:
  accessModes:
    - ReadOnlyMany
  storageClassName: nfs-dynamic
```

```
  resources:
    requests:
      storage: 3Gi
---
#发布应用并挂载存储
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nfs-dynamic
  name: nfs-dynamic
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nfs-dynamic
  template:
    metadata:
      labels:
        app: nfs-dynamic
    spec:
      containers:
      - image: httpd:alpine
        name: httpd
        volumeMounts:
        - name: nfs-dynamic
          mountPath: /usr/local/apache2/htdocs/
        ports:
        - containerPort: 80
      volumes:
      - persistentVolumeClaim:
          claimName: claim-nfs-dynamic
        name: nfs-dynamic
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nfs-dynamic
  name: nfs-dynamic
spec:
  ports:
  - name: 80-80
    nodePort: 30080
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nfs-dynamic
  type: NodePort
```

示例代码编辑完成后,发布应用并验证数据持久化功能,命令如下:

```
#部署
sudo kubectl create -f myapp-nfs-dynamic.yaml
#查看应用的 Pod 状态
sudo kubectl get pods
#查看 PV、PVC 绑定状态
sudo kubectl get pv
sudo kubectl get pvc
#将/nfs-share02/index.html 复制至应用的任意 Pod 的指定目录/usr/local/apache2/htdocs/
sudo kubectl cp /nfs-share02/index.html default/nfs-dynamic-786577b688-429k8:/usr/
local/apache2/htdocs/
```

命令执行的过程如图 3-73 所示。

```
user01@node01:~$ sudo kubectl create -f myapp-nfs-dynamic.yaml
deployment.apps/nfs-client-provisioner created
storageclass.storage.k8s.io/nfs-dynamic created
persistentvolumeclaim/claim-nfs-dynamic created
deployment.apps/nfs-dynamic created
service/nfs-dynamic created
user01@node01:~$ sudo kubectl get pods
NAME                                     READY   STATUS    RESTARTS   AGE
nfs-client-provisioner-7595b9d99b-765zb   1/1     Running   0          44s
nfs-dynamic-786577b688-429k8              1/1     Running   0          44s
nfs-dynamic-786577b688-zkq7q              1/1     Running   0          44s
user01@node01:~$ sudo kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                       STORAGECLASS   VOLUMEA
TTRIBUTESCLASS   REASON   AGE
pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa   3Gi        ROX            Delete           Bound    default/claim-nfs-dynamic   nfs-dynamic    <unset>
                         30s
user01@node01:~$ sudo kubectl get pvc
NAME                STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
claim-nfs-dynamic   Bound    pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa   3Gi        ROX            nfs-dynamic    <unset>                 52s
user01@node01:~$ sudo kubectl cp /nfs-share02/index.html default/nfs-dynamic-786577b688-429k8:/usr/local/apache2/htdocs/
```

图 3-73 基于 StorageClass 部署应用

测试文件 index.html 一旦成功复制,就可以使用浏览器访问该测试页面验证该页面的可用性,如图 3-74 所示。

图 3-74 StorageClass 功能测试页面

如果此时查看 NFS 服务器端共享目录/nfs-share02 内的数据就会发现,在该目录下系统自动创建了用于存储应用数据的目录文件,如图 3-75 所示。

```
user01@node01:~$ ls /nfs-share02/
default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa  index.html
user01@node01:~$
user01@node01:~$
user01@node01:~$
user01@node01:~$ ls /nfs-share02/
default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa  index.html
user01@node01:~$ ls /nfs-share02/default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa/
index.html
```

图 3-75 自动创建存储数据的持久化目录

当应用被删除时，对应的 PVC 会被删除，但存储数据的 PV 一直存在，除非管理员手动删除。需要特别注意的是即便手动删除了 PV，存储在 NFS 服务上的数据依然存在，相关操作命令如下：

```
#删除测试应用
sudo kubectl delete -f myapp-nfs-dynamic.yaml
#查看 PVC、PV 状态
sudo kubectl get pvc
sudo kubectl get pv
#删除 pv
sudo kubectl delete pv pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa
#查看 NFS 服务器端共享目录/nfs-share02 目录内的数据
ls /nfs-share02/
ls /nfs-share02/default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa/
```

命令执行的过程及信息输出如图 3-76 所示。

```
user01@node01:~$ sudo kubectl delete -f myapp-nfs-dynamic.yaml
[sudo] password for user01:
deployment.apps "nfs-client-provisioner" deleted
storageclass.storage.k8s.io "nfs-dynamic" deleted
persistentvolumeclaim "claim-nfs-dynamic" deleted
deployment.apps "nfs-dynamic" deleted
service "nfs-dynamic" deleted
user01@node01:~$ sudo kubectl get pvc
No resources found in default namespace.
user01@node01:~$ sudo kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM                       STORAGECLASS   VOLUM
EATTRIBUTESCLASS   REASON   AGE
pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa   3Gi        ROX            Delete           Released   default/claim-nfs-dynamic   nfs-dynamic    <unse
t>                          84m
user01@node01:~$ ls /nfs-share02/
default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa  index.html
user01@node01:~$ sudo kubectl delete pv pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa
persistentvolume "pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa" deleted
user01@node01:~$ sudo kubectl get pv
No resources found
user01@node01:~$ ls /nfs-share02/
default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa  index.html
user01@node01:~$ ls /nfs-share02/default-claim-nfs-dynamic-pvc-b8eee63b-351f-4fcc-bb4e-be1365c487aa/
index.html
```

图 3-76　验证数据持久化存储

通过上述演示案例不难发现，使用动态卷制备(Dynamic Volume Provisioning)的方式实现数据卷的动态创建和挂载，可以提升部署效率、简化存储管理流程，因此该方式在企业环境中被广泛使用。

3.4　本章小结

本章从企业实际应用的角度出发并结合具体案例，全面而深入地探讨了容器编排技术 Kubernetes 的核心知识点。内容涵盖了 Kubernetes 的起源与发展、独特的架构设计、核心概念解析、详尽的工作流程描述，以及一系列典型的管理命令介绍。其中不仅深入地讲解了基于 Kubernetes 集群的部署实践，还详细地阐述了应用生命周期管理的全过程，并提供了 YAML 文件编写的实用技巧，帮助读者更好地掌握这一技术。尤为重要的是，本章还着重

介绍了Kubernetes集群的数据持久化功能,这是确保应用稳定运行和数据安全的关键所在。

尤其是通过一系列实际案例的学习,读者将能够更直观地理解Kubernetes的运作机制,进一步加深对容器编排技术的认识与掌握。这些案例不仅提升了理论知识的实用性,也为读者在实际工作中应用Kubernetes提供了宝贵的参考和借鉴。