第 5 章　Kubernetes

设想一下，如果在生产环境中要部署一个大型的 Web 应用（如网站或微信小程序后端），为了保证该 Web 应用性能的优异性，需要将其部署在一个分布式集群上。此时会有两种方案：其一，将该 Web 应用的每个服务分别部署在一台虚拟机上，然后用 OpenStack 管理和调度这些虚拟机，使这些服务能协同运行；其二，将该 Web 应用的每个服务打包成一个容器，然后将这些容器分别放在不同的虚拟机上。显然，第二种方案更好，因为容器的性能优于虚拟机，并且对容器的调度比对虚拟机的调度更加灵活。

对于用集群部署一个 Web 应用，大多数用户的需求是：有这个 Web 应用对应的一组容器镜像，要在一个给定的集群上把这个 Web 应用运行起来。此时要解决的关键问题是怎样对分布在集群上的一组容器进行调度和管理。这就需要用到 Kubernetes。

5.1　Kubernetes 概述

近年来，云计算的核心理念从以资源编排为主体向以应用编排为主体转变。第一代云计算平台的设计以虚拟机为核心配套设计，典型代表是 OpenStack。第二代云计算平台以更加敏捷、更细粒度的控制适应应用的快速迭代，实现应用和资源的统一编排，典型代表是 Kubernetes。第二代云计算平台的优势在于：由于容器不包含操作系统，因此，相较于将应用直接部署在虚拟机中，将应用部署在容器中能节省大量的计算和存储资源，对应用的调度和控制的效率也更高。故对容器的编排成为第二代云计算平台的关键技术。

5.1.1　容器编排工具

容器的编排技术目前主要有两种：其一是针对单机上容器编排的 Docker Compose；其二是针对集群上容器编排的 Kubernetes。Kubernetes 是一个可以实现跨主机管理容器化应用的系统，是容器化应用和服务生命周期管理平台，它的出现不仅解决了多容器之间数据传输与沟通的瓶颈，而且促进了容器技术的发展。

容器的出现和普及为开发者提供了良好的平台和媒介，使开发和运维工作变得简单与高效。随着企业业务和需求的增长，在大规模使用容器技术后，如何对这些运行的容器进行管理成为首要问题。在此情况下，容器编排工具应运而生，最具代表性的有以下 3 种。

1. Apache 公司的 Mesos

Mesos 是 Apache 公司推出的开源分布式资源管理框架，由美国加利福尼亚大学伯克利分校的 AMPLab（Algorithms，Machine and People Lab，算法、计算机和人实验室）开发。Mesos 早期通过了万台节点验证，2014 年之后又被广泛使用在 eBay、Twitter 等大型互联网公司的生产环境中。

2. Docker 公司的“三剑客”

容器诞生后，Docker 公司就意识到单一容器体系的弊端。为了能够有效地满足用户的

需求并消除集群中的瓶颈,Docker公司相继推出Machine、Compose、Swarm项目。

Machine项目用Go语言编写,可以实现Docker运行环境的安装与管理,实现在指定节点或平台上批量安装并启动Docker服务。

Compose项目用Python语言编写,可以实现基于Docker容器多应用服务的快速编排,其前身是开源项目Fig。Compose项目使用户可以通过单独的YAML文件批量创建自定义的容器,并通过API对集群中的Docker服务进行管理。

Swarm项目用Go语言编写,支持原生的Docker API和Docker网络插件,很容易实现跨主机集群部署。

3. Google公司的Kubernetes

Kubernetes来自希腊语,意为“舵手”。因为这个单词的K与s之间有8个字母,所以常简称为K8s。Kubernetes用Go语言开发,是Google公司发起并维护的开源容器集群管理系统,其底层基于Docker、rkt等容器技术,其前身是Google公司开发的Borg系统。Borg系统在Google公司内部已经应用了十几年,曾管理超过20亿个容器。经过多年的技术积累,Google公司将Borg系统完善后贡献给了开源社区,并将其重新命名为Kubernetes。

Kubernetes系统支持用户通过模板定义服务配置。用户提交配置信息后,系统会自动完成对应用容器的创建、部署、发布、伸缩、更新等操作。Kubernetes发布以来吸引了Red Hat、CentOS等知名互联网公司与容器爱好者的关注,是目前容器集群管理系统中优秀的开源项目之一。

2017年Google公司的搜索热度报告中显示,Kubernetes搜索热度已经远远超过Mesos和Docker Swarm,这也标志着Kubernetes在容器编排市场逐渐占有主导地位。

5.1.2 Kubernetes解决的问题

运行在大规模集群中的各种任务之间存在着千丝万缕的关系。如何处理这些关系是作业编排和管理系统的难点。这种关系在各种技术场景中随处可见,例如,Web应用与数据库之间的访问关系,负载均衡器和后端服务之间的代理关系,门户应用与授权组件之间的调用关系。同属于一个服务单位的不同功能之间也存在这样的关系。例如,Web应用与日志搜集组件之间的文件交换关系。

在容器普及前,传统虚拟化环境对这种关系的处理方法都是粗粒度的。很多并不相关的应用被部署在同一台虚拟机中,也许是因为这些应用之间偶尔会互相发起几个HTTP请求。更常见的是,把应用部署在虚拟机中之后,还需要手动维护协作处理日志搜集、灾难恢复、数据备份等辅助工作的守护进程。

容器技术在功能单位的划分上有着独一无二的细粒度优势。使用容器技术可以将那些原先挤在同一个虚拟机里的应用、组件、守护进程分别生成镜像,然后运行在专属的容器中。进程互不干涉,各自拥有资源配额,可以被调度到整个集群里的任何一台计算机上。这正是PaaS系统最理想的工作状态,也是微服务思想得以落地的先决条件。Kubernetes也因此被认为是云原生开发的事实标准。

云原生是指开发软件产品时就将软件设计为适合部署在云上的分布式组件模式,是基于分布式部署和统一运管的分布式云,以容器、微服务、DevOps等技术为基础建立的一套

软件产品体系。

1. Kubernetes 对容器的管理

为了解决容器间需要紧密协作的难题，Kubernetes 系统中使用了 Pod（容器集）这种抽象的概念来管理各种资源。当需要一次性启动多个应用实例时，可以通过系统中的多实例管理器 Deployment 实现，当需要通过一个固定的 IP 地址和端口以负载均衡的方式访问 Pod 时，可以通过 Service 实现。

Kubernetes 项目对容器间的访问进行了分类。

（1）在服务器上运行的应用服务频繁进行交互访问和信息交换。在常规环境下，这些应用往往会被直接部署在同一台计算机上，通过本地主机（local host）通信并在本地磁盘目录中交换文件。在 Kubernetes 项目中，这些运行的容器被划分到同一个 Pod 内，并共享 Namespace 和同一组数据卷，从而达到高效率交换信息的目的。

（2）还有另外一些常见的需求，例如 Web 应用对数据库的访问。在生产环境中它们不会被部署在同一台计算机上，这样，即使 Web 应用所在的服务器宕机，数据库也不会受影响。容器的 IP 地址等信息不是固定的，为了使 Web 应用可以快速找到数据库容器的 Pod，Kubernetes 项目提供了一种名为 Service 的服务。Service 的主要作用是作为 Pod 的代理入口（Portal），代替 Pod 对外暴露一个固定的网络地址。这样，运行 Web 应用的 Pod 就只需要关心数据库容器的 Pod 提供的 Service 信息。

Kubernetes 系统不仅可以实现跨集群调度、水平扩展、监控、备份、灾难恢复，还可以消除大型互联网集群中多任务处理的瓶颈。Kubernetes 遵循微服务架构理论，将整个系统划分为多个功能各异的组件。各组件结构清晰，部署简单，可以非常方便地运行于系统环境之中。利用容器的扩容机制，系统将容器归类，形成 Pod，用于帮助用户调度工作负载，并为这些容器提供联网和存储服务。

2. Kubernetes 的主要功能

近几年容器技术得到广泛应用，使用 Kubernetes 系统管理容器的企业也在不断增加，Kubernetes 系统的主要功能如下：

（1）自动发布和回滚。Kubernetes 通过持久化存储保存应用发布时的相关配置信息，从而在部署过程中发生问题时能够执行回滚操作。

（2）自动化装箱。Kubernetes 按照应用对资源的要求自动部署容器，从而提高了资源的利用率，节省了资源。

（3）水平扩容。Kubernetes 根据应用在运行过程中对 CPU、内存的使用情况，通过简单的命令即可对应用进行扩容和缩容。

（4）配置管理。Kubernetes 对集群和应用的配置信息进行持久化存储，可以在不重新构建镜像的情况下更新应用的配置信息。

（5）自愈能力。Kubernetes 实现了容器的高可用性。当节点上运行的容器失败后，Kubernetes 会对容器进行重启。即使节点出现宕机，Kubernetes 也会对容器进行重新部署和重新调度，容器能够正常运行后才会对外提供服务。

（6）服务发现和负载均衡。Kubernetes 内置了服务发现机制和负载均衡功能，不需要使用额外的服务。

（7）存储编排。Kubernetes 利用持久卷和持久卷声明完成存储系统的自动挂载，同时

支持多种存储系统(如本地存储系统、云存储系统和网络存储系统等)。

5.1.3 云原生与微服务架构

云原生是指软件产品设计之初就将软件设计为适合部署在云上的分布式组件模式,是基于分布式部署和统一运管的分布式云,以容器、微服务、DevOps(开发运维一体化)等技术为基础建立的一套软件产品体系。

早期互联网应用系统基本上采用单体架构模式进行设计。所谓单体架构就是把所有的业务模块编写在同一个项目中进行集中管理。

微服务架构是一种分布式系统的架构模式,它将一个大型的应用程序划分成一系列小型服务,并通过轻量级通信协议将这些服务连接起来。每个服务都有自己独立的业务功能和数据存储,可以使用不同的编程语言、开发框架和技术实现。这种架构模式强调了服务之间的松耦合和自治性,以支持快速迭代和部署并提高开发和运维效率。微服务架构的核心是服务拆分,这种拆分通常基于业务逻辑、可扩展性、可靠性和性能等因素。

在微服务架构中,每个服务都是一个独立的进程,它们之间通过 HTTP/RESTful、消息队列、RPC 等通信方式进行交互。通常情况下,每个服务都有自己独立的数据库或数据存储,可以使用不同的技术栈实现。

每个服务都是独立可部署的,可以单独进行开发、测试、部署和扩展。此外,微服务架构还强调了自动化运维和基础设施的可编程性,支持自动化部署、测试、监控和故障处理等流程。

微服务架构带来了以下好处:

(1) 高可扩展性。微服务架构支持水平扩展,可以根据需要增加或减少服务实例,以满足应用程序的负载变化。不同的服务也可以采用不同的扩展方式和策略,以适应不同的负载特征。

(2) 高可靠性。微服务架构强调了服务的自治性和容错性,每个服务都可以独立地处理请求和错误,避免了单点故障和服务之间的相互影响。此外,微服务架构也支持故障转移和负载均衡,以提高系统的可靠性和稳定性。

(3) 高灵活性。微服务架构允许开发人员使用不同的编程语言、开发框架和技术实现服务,以满足不同的业务需求和技术需求。每个服务都是独立的,可以根据需要进行修改、扩展或替换,以满足业务的变化和技术的演进。

(4) 高可维护性。微服务架构通过服务的自治性和清晰的接口规范,使得服务之间的依赖关系更加清晰和简单,易于理解和维护。每个服务也可以独立进行测试、部署和更新,避免了传统单体架构中复杂的代码耦合和版本管理问题。微服务系统将整个应用按功能分成一个个子服务系统,每个子服务系统都可以由不同的技术团队进行开发和维护,这样就大幅提升了系统开发和维护的效率。

相对于单体架构,使用微服务架构开发的系统具有很大的优势。因此,国内外许多大型企业已经逐步将其系统开发转向微服务架构模式。

尽管微服务架构带来了很多好处,但也存在一些不容忽视的缺点:

(1) 复杂性增加。微服务架构的设计和实现都比较复杂,需要开发团队具备更高的技术水平和经验。此外,微服务架构涉及的服务数量和部署节点也会增加系统的复杂性,需要更多的管理和监控。

（2）带来了分布式系统的问题。微服务架构将系统拆分成多个服务，各个服务运行在不同的节点上，因此需要解决分布式系统所面临的问题，如网络通信、服务发现、负载均衡、容错和事务一致性等。

（3）测试和部署成本增加。由于微服务架构中服务数量的增加，测试和部署的成本也会相应增加。需要考虑如何对每个服务进行单独的测试和部署，并确保服务之间的兼容性和正确性。

（4）数据管理困难。在微服务架构中，每个服务都拥有自己的数据存储，因此需要考虑如何管理和同步数据。此外，由于每个服务都可以独立地进行数据库选择和设计，因此需要保证数据的一致性和可靠性。

（5）给服务治理和监控带来了挑战。由于微服务架构中服务的数量和复杂性都比较高，因此需要实现完善的服务治理和监控机制，确保服务的可用性和性能。

需要注意的是，微服务架构的缺点并不是不可克服的，而是需要在设计和实现过程中考虑到这些因素，并采取相应的措施加以解决。为了实现微服务架构，需要使用一些微服务框架和平台帮助开发人员构建、管理和部署微服务应用。

在常见的微服务框架和平台中，SpringCloud 是极为流行的微服务解决方案之一，它提供了多种组件以支持微服务架构的各种需求，例如服务发现、负载均衡、熔断器、API 网关等。SpringCloud 可以与 SpringBoot 框架配合使用，通过使用 SpringBoot Starter 插件简化配置，帮助开发人员快速构建微服务应用。相比之下，Kubernetes 是一种容器编排平台，可以用于自动化部署、扩展和管理容器化的微服务应用，提供了负载均衡、容器编排、服务发现等功能。Kubernetes 具有高度可扩展性和可定制性，适用于需要高度灵活性和可控制性的场景，可以支持大规模微服务应用。云原生微服务体系升级应用架构，使应用天然具有更好的可观测性、可控制性和可容错性等特性。

比较而言，SpringCloud 和 Kubernetes 的优缺点如下：

（1）技术栈。SpringCloud 使用 Spring 框架和 Netflix OSS 等技术栈，Kubernetes 使用 Docker 和 etcd 等技术栈。SpringCloud 更加适合 Java 开发人员和 Spring 框架的用户，而 Kubernetes 则更加适合云原生和容器化应用的开发人员。

（2）服务发现和注册。SpringCloud 使用 Consul、Eureka 等组件实现服务发现和注册，Kubernetes 使用 etcd 实现。Kubernetes 的服务发现更加可靠和稳定，支持多数据中心和跨云平台部署。

（3）配置管理。SpringCloud 使用 Config Server 实现集中式的配置管理，Kubernetes 使用 ConfigMap 和 Secret 实现。Kubernetes 的配置管理更加灵活和可扩展，支持多种数据源和配置格式。

（4）服务网关。SpringCloud 使用 Zuul、Spring Cloud Gateway 等组件实现 API 网关和负载均衡，Kubernetes 使用 Ingress 实现。Kubernetes 的 Ingress 功能还比较初级，不支持复杂的路由和负载均衡策略。

（5）缩放和扩展。Kubernetes 提供了强大的容器编排和自动扩缩容功能，可以方便地管理大规模的容器化应用程序。SpringCloud 的缩放和扩展需要手动实现，比较麻烦且容易出错。

综上所述，SpringCloud 和 Kubernetes 两种微服务框架具有不同的优点和适用场景，开

发人员可以根据具体需求选择合适的框架和平台构建、管理和部署微服务应用。

5.1.4 Kubernetes 的体系结构

Kubernetes 集群主要由控制节点(用 Master 表示,部署高可用的容器需要两个以上控制节点)和多个工作节点(用 Node 表示)组成,两种节点上分别运行不同的组件以维持集群高效、稳定的运转,另外还需要集群状态存储系统(etcd 组件)提供数据存储服务。Kubernetes 集群的组成如图 5-1 所示。

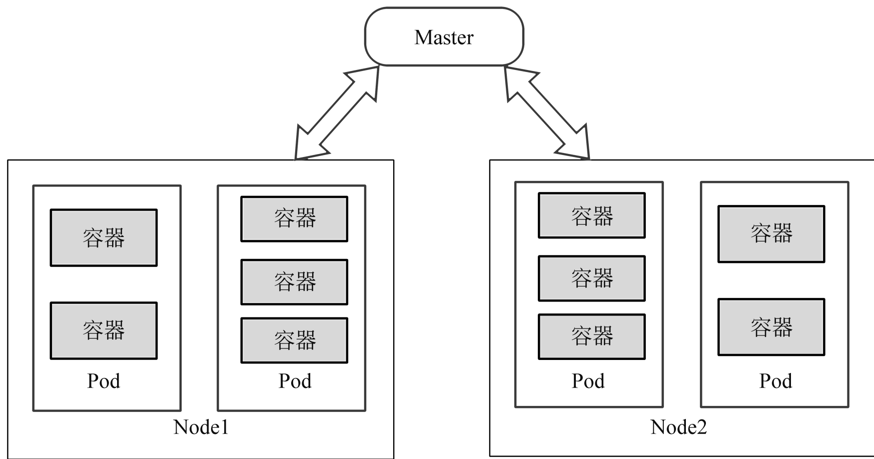

图 5-1 Kubernetes 集群的组成

在 Kubernetes 的系统架构中,控制节点上主要运行着 API Server、Controller Manager 和 Scheduler 组件,而每个工作节点上主要运行着 Kubelet、Kube-Proxy 和 Docker 引擎,如图 5-2 所示。除此之外,完整的集群服务还依赖一些附加的组件,如 KubeDNS、Heapster、Ingress Controller 等。

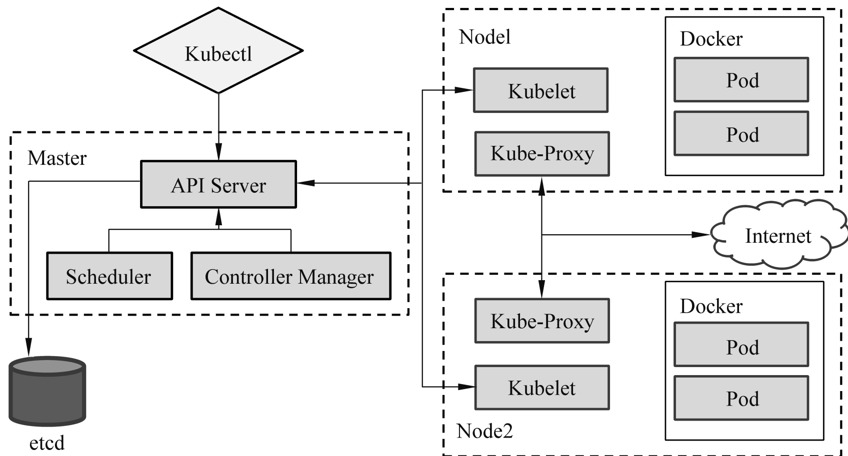

图 5-2 Kubernetes 的体系结构

1. 控制节点

控制节点是整个集群的网络中枢,主要负责组件或者服务进程的管理和控制,例如,追

踪其他服务器健康状态，保持各组件之间的通信，为用户或者服务提供 API。

控制节点中的组件可以在集群中的任何计算机上运行。但是，为简单起见，设置时通常会在一台计算机上部署和启动所有主组件，并且不在控制节点上运行用户容器。在控制节点上部署的组件包括以下 3 种。

1）API Server

API Server 组件是整个集群的网关，作为 Kubernetes 系统的入口，其内部封装了核心对象的增、删、改、查操作，以 RESTful API 方式供外部用户和内部组件调用，就像是机场的“联络室”。

2）Scheduler

Scheduler（调度器）组件监视新创建且未分配工作节点的 Pod，并根据不同的需求将其分配到工作节点中。同时负责集群的资源调度、组件抽离。

3）Controller Manager

Controller Manager（控制器管理器）组件是所有资源对象的自动化控制中心，大多数对集群的操作都是由几个被称为控制器的进程执行的，这些进程被集成于 Kube-Controller-Manager 守护进程中。该组件实现的主要功能如下：

- 生命周期功能，例如 Namespace 创建以及事件、Pod、工作节点和级联垃圾的回收。
- API 业务逻辑功能，例如 ReplicaSet 执行的 Pod 扩展等。

2. 工作节点

工作节点（在早期的版本中也被称为 Minion）主要负责接收控制节点的工作指令并执行相应的任务。当某个工作节点宕机时，控制节点会将负载切换到其他的工作节点上。

工作节点上部署的组件包括以下 3 种。

1）Kubelet

Kubelet 组件主要负责管控容器，它会从 API Server 接收 Pod 的创建请求然后进行相关的启动和停止容器操作。同时，Kubelet 监控容器的运行状态并汇报给 API Server。

2）Kube-Proxy

Kube-Proxy 组件负责为 Pod 创建代理服务，从 API Server 获取所有的 Service 信息，并创建相关的代理服务，实现 Service 到 Pod 的请求路由和转发。Kube-Proxy 在 Kubernetes 层级的虚拟转发网络中扮演着重要的角色。

3）Docker 引擎

Docker 引擎负责 Docker 的镜像管理以及 Pod 和容器的运行，是 Kubernetes 真正的执行引擎。

3. etcd

etcd 是一个高可用的键-值存储系统，主要用于共享配置和服务发现。etcd 不仅可提供键-值存储，还可以提供监听（watch）机制。键-值发生改变时 etcd 会通知 API Server，并由其通过 Watch API 向客户端输出。实际上，etcd 提供了与 ZooKeeper 相似的功能，在分布式系统中，如何管理节点间的状态一直是一个难题，etcd 像是专门为集群环境的服务发现和注册而设计的，它提供了数据 TTL 失效、数据改变监视、多值、目录监听、分布式锁原子操作等功能，可以方便地跟踪并管理集群节点的状态。

Kubernetes 集群中所有的状态信息都存储于 etcd 数据库中。etcd 提供了高度一致的

分布式键-值存储功能,在集群中是独立的服务组件,可以实现集群发现、共享配置以及一致性保障(如数据库控制节点选择、分布式锁)等功能。在生产环境中,建议以集群的方式运行 etcd 并保证其可用性。

5.2 Pod

有些容器天生就需要紧密联系、一起工作。Pod 提供了比容器更高层次的抽象,将这些容器封装到一个部署单元中。Kubernetes 以 Pod 为最小单位进行调度、扩展、共享资源、管理生命周期。Pod 也是最小化运行容器化应用的资源对象。一个 Pod 由一个或多个容器组成,Pod 中的容器共享存储和网络资源,在同一台 Docker 主机上运行。一个 Pod 也代表着集群中运行的一个进程。如果 Pod 所在的工作节点宕机,会将这个工作节点上的所有 Pod 重新调度到其他工作节点上。

Kubernetes 中其他大多数组件都是围绕着 Pod 进行支撑和功能扩展的,例如,用于管理 Pod 运行的 StatefulSet 和 Deployment 等控制器对象,用于暴露 Pod 应用的 Service 和 Ingress 对象,为 Pod 提供存储的 Persistent Volume 存储资源对象,等等。

5.2.1 Pod 的使用方式

Pod 的使用有如下两种方式:

(1) 一个 Pod 中运行一个容器。这是最常见的使用方式。在这种使用方式中,Pod 相当于单个容器的封装,因为 Kubernetes 只能管理 Pod,而不能直接管理容器。这种方式通常用于微服务。

(2) 一个 Pod 中同时运行多个容器。一个 Pod 中也可以同时封装几个需要紧密耦合、互相协作的容器,它们共享资源。这些在同一个 Pod 中的容器可以互相协作,成为一个服务单位,例如一个容器共享文件,另一个容器更新这些文件。Pod 将这些容器的存储资源作为一个实体管理。

一个 Pod 中的容器必须运行于同一节点上。现代容器技术建议一个容器只运行一个进程,进程终止时,容器生命周期也就结束了。若想在容器内运行多个进程,需要有一个类似于 Linux 操作系统 init 进程的管控类进程,以树状结构完成多进程的生命周期管理。运行于各自容器内的进程无法直接完成网络通信,这是由于容器间的隔离机制导致的, Kubernetes 中的 Pod 资源抽象正是为了解决此类问题而提出的。Pod 对象是一组容器的集合,这些容器共享网络、UTS 及 IPC 命名空间,因此具有相同的域名、主机名和网络接口,并可通过 IPC 直接通信。

当 Pod 数量过多时,副本控制器(Replication Controller,RC)会终止多余的 Pod。当 Pod 数量太少时,副本控制器将会启动新的 Pod。

5.2.2 Pod 的资源共享

每个 Pod 都有一个特殊的被称为基础容器的 Pause 容器。Pause 容器对应的镜像属于 Kubernetes 平台的一部分。除了 Pause 容器,每个 Pod 还包含一个或者多个紧密相关的用户容器。Pod 中的 Pause 容器和用户容器如图 5-3 所示。

Pause 容器是 Pod 资源中针对各容器提供网络命名空间等共享机制的底层容器，Pause 容器（也可称为父容器）就是为了管理 Pod 中容器间的共享操作，这个父容器要能够准确地知道如何创建共享运行环境的容器，还要能够管理这些容器的生命周期。

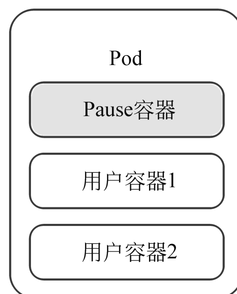

图 5-3　Pod 中的 Pause 容器和用户容器

为了实现这个父容器的构想，在 Kubernetes 中，用 Pause 容器作为一个 Pod 中所有容器的父容器。Pause 容器有两个核心的功能：一是提供整个 Pod 的 Linux 命名空间的基础；二是用来启用 PID 命名空间，它在每个 Pod 中都作为 PID 为 1 的进程（init 进程），并回收僵尸进程。

Pause 容器管理整个容器组，也负责管理容器的生命周期，健康检查和生存探针实际上就是为了监测容器的生命周期。因为在一个容器组作为一个单元的情况下，难以对整个容器组简单地进行判断及行动。为此引入与业务无关的 Pause 容器作为 Pod 的根容器，以它的状态代表整个容器组的状态，这样就可以解决该问题。

Pause 容器还为容器提供网络和资源共享，Pod 里的多个业务容器共享 Pause 容器的 IP 地址，共享 Pause 容器挂载的数据卷，这样既简化了容器之间的通信问题，也解决了容器之间的文件共享问题。

5.2.3　Pod 的存储共享机制

Pod 可以指定多个共享的数据卷。Pod 中的所有容器都可以访问共享的数据卷。数据卷也可以用来持久化 Pod 中的存储资源，以防容器重启后文件丢失。

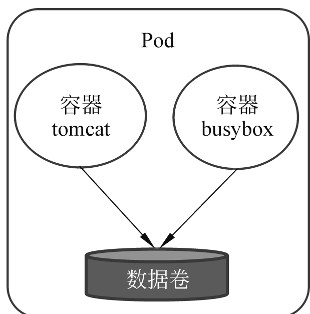

图 5-4　Pod 的存储共享机制

在同一个 Pod 中的多个容器能够共享 Pod 级别的数据卷。数据卷可以被定义为各种类型，多个容器各自进行挂载操作，将一个数据卷挂载为容器内部需要的目录，如图 5-4 所示。

在下面的例子中，Pod 内包含两个容器：tomcat 和 busybox，在 Pod 级别设置数据卷名为 app-logs，用于 tomcat 容器向其中写日志，busybox 容器从中读取日志文件。

Kubernetes 中所有的资源对象都可以用 YAML 文件定义和描述。使用如下命令创建一个描述 Pod 的 YAML 文件（kind 标签为 Pod 表示该文件是描述 Pod 的）：

```
kubectl create -f pod-volume-applogs.yaml
```

pod-volume-applogs.yaml 的内容如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - name: tomcat
```

```
      image: tomcat
      ports:
      - containerPort: 8080
      volumeMounts:
      - name: app-logs
        mountPath: /usr/local/tomcat/logs
    - name: busybox
      image: busybox
      command: ["sh", "-c", "tail -f /logs/catalina*.log"]
      volumeMounts:
      - name: app-logs
        mountPath: /logs
    volumes:
    - name: app-logs
  emptyDir: {}
```

上述代码设置数据卷名为 app-logs,类型为 emptyDir(也可以设置为其他类型),挂载到 tomcat 容器内的/usr/local/tomcat/logs 目录,同时挂载到 busybox 容器内的/logs 目录。tomcat 容器启动后就可以向/usr/local/tomcat/logs 目录写文件,busybox 容器就可以读取文件了。

5.2.4 Pod 的网络共享机制

Pod 中的所有容器使用同一个网络命名空间,即相同的 IP 地址和端口的空间。它们可以直接用 localhost 通信。同样,这些容器可以共享存储,当 Kubernetes 挂载数据卷到 Pod 上时,本质上是将数据卷挂载到 Pod 中的每一个容器上。每个 Pod 都会被分配一个唯一的 IP 地址。Pod 中的所有容器共享网络命名空间,包括 IP 地址和端口。Pod 中的容器与外界通信时,必须分配共享网络资源(例如使用宿主机的端口映射)。

在 Kubernetes 中,Pod 的网络通信可分为以下 3 种情况:

(1) Pod 中的容器之间的通信。在同一个 Pod 中的容器(Pod 中的容器是不会跨宿主机的)共享同一个网络命名空间,相当于它们在同一台计算机上,可以用 localhost 地址访问彼此的端口。

(2) 同一个工作节点上的 Pod 之间的通信。每个 Pod 都有一个真实的全局 IP 地址,同一个工作节点上的不同 Pod 之间可以直接用对方 Pod 的 IP 地址进行通信。两个 Pod 都通过虚拟网卡连接到同一个 Docker0 网桥,网段相同,所以它们之间可以直接通信。

(3) 不同工作节点上的 Pod 之间的通信。Pod 地址与 Docker0 在同一网段,Docker0 网段与宿主机网卡是两个不同的网段,且不同工作节点之间的通信只能通过宿主机的物理网卡进行。

要想实现不同工作节点上的 Pod 之间的通信,就必须想办法通过主机的物理网卡 IP 地址进行寻址和通信。因此要满足两个条件:一是 Pod 的 IP 地址不能冲突;二是将 Pod 的 IP 地址和所在的工作节点的 IP 地址关联起来,通过这个关联让不同工作节点上的 Pod 之间直接通过内网 IP 地址通信。