Comware V9 容器技术白皮书

目 录

1 概述

1.1 产生背景

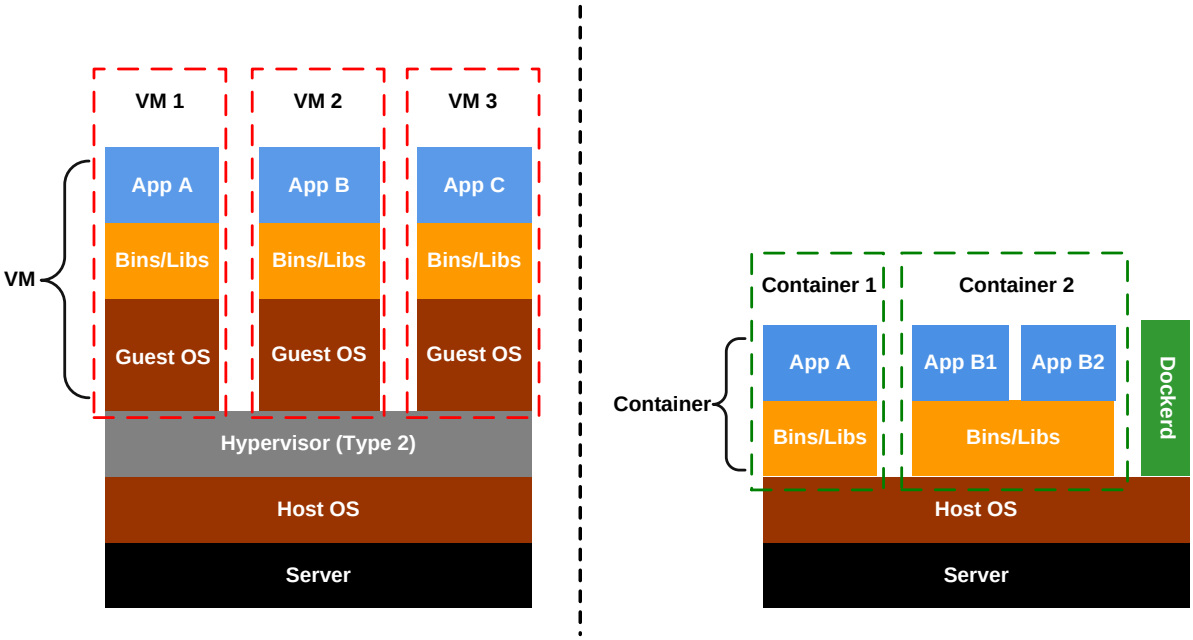
随着互联网的普及，为满足人们的生产和生活需求，各式各样地互联网业务层出不穷，各服务提供商的数据中心规模不断地扩大，业务对服务器的需求和要求也越来越高。对于单个服务提供商而言，大规模的服务器投入使用使得业务部署成本激增，网络维护难度加大；对于整个社会而言，各服务提供商重复投资，资源利用率低，资源浪费严重。服务器虚拟化能帮助用户快速、低成本的部署新业务，受到了市场的青睐。

服务器虚拟化有两大分支：一个是传统的 VM（Virtual Machine，虚拟机）技术；一个是容器技术。

如图 1 所示，传统的 VM 技术，如 KVM、Xen 等，通过虚拟化层去抽象底层物理资源，提供相互隔离的虚拟机环境，在虚拟机中要求安装和运行独立的客户机操作系统，客户机的变化不会影响到宿主机，可以屏蔽底层硬件的差异，向上给应用提供一致性的运行环境。但由于需要额外的虚拟化层，以及独立的客户操作系统，虚拟机不可避免的带来了对 CPU、磁盘、网络等各方面性能的损失。

容器技术是一种轻量级的虚拟化技术，相较于虚拟机的实现更加轻量化，它共享宿主机的内核实现，不需要独立的客户机操作系统，这样占用的磁盘更少、启动更快，也越来越受业界的欢迎。

图1 VM 和容器架构对比示意图

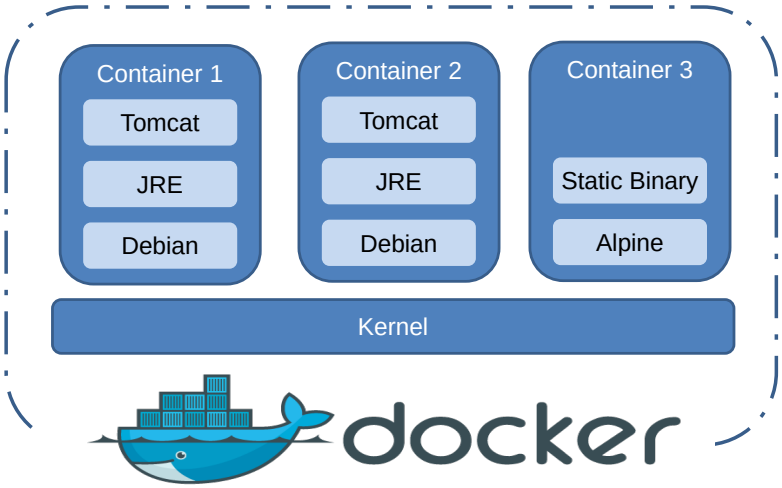


1.2 容器简介

容器技术是一种轻量级的虚拟化技术。一个容器是一个标准的软件单元，它可以对应一个应用（单独提供一种服务），也可以对应一组应用（多个应用相互配合提供服务）。它包含软件运行所需的程序、环境变量、系统库、配置等数据和文件。一个宿主机上可以部署多个容器，各个容器共享宿主机内核，但容器间互相隔离，属于操作系统层面的虚拟化技术。

常见的容器管理应用主要有 Docker、LXC 等。Docker 通过 Linux 内核的 Namespace 和 Cgroup 技术对资源进行隔离和分组，采用镜像技术打包容器交付件，极大地简化了容器的部署和使用，使得容器技术迅速普及，应用越来越广泛和深入。

图2 Docker 容器示意图



Comware V9 集成了 Docker 容器管理功能。用户可以在 Comware V9 系统上运行容器化的第三方应用。容器化的第三方应用可独立开发和部署，能方便快捷地补充 Comware V9 系统提供的功能，同时第三方应用和 Comware V9 系统互相独立运行，第三方应用故障、重启不会影响 Comware V9 系统。

1.3 技术优点

容器运行在 Linux 操作系统之上，直接使用宿主机的硬件资源，利用 Linux 的 Namespace 和 Cgroup 等技术来实现容器间的隔离和资源分组控制。与传统的虚拟机（VM）技术相比，容器启动更快，占用资源更少，在宿主机同等配置条件下，容器可承载更多的应用。

Docker 以集装箱的方式，将应用软件以及应用软件依赖的组件打包成容器镜像，进行发布和管理。它使得容器可以在物理机、虚拟机、公有云、私有云等不同环境中部署，屏蔽了运行环境的差异，具有灵活度高、应用部署快的特点。

容器镜像有统一的打包和发布标准，具有很强的一致性和可复制性。容器镜像只需要打包一次、配置标准的运行环境，就可以在任何机器上运行容器，使得运维工作更加高效和可靠。

容器消除了开发、测试、生产环境的不一致性，使得各个角色各司其职，更关注于业务本身。

2 容器核心技术

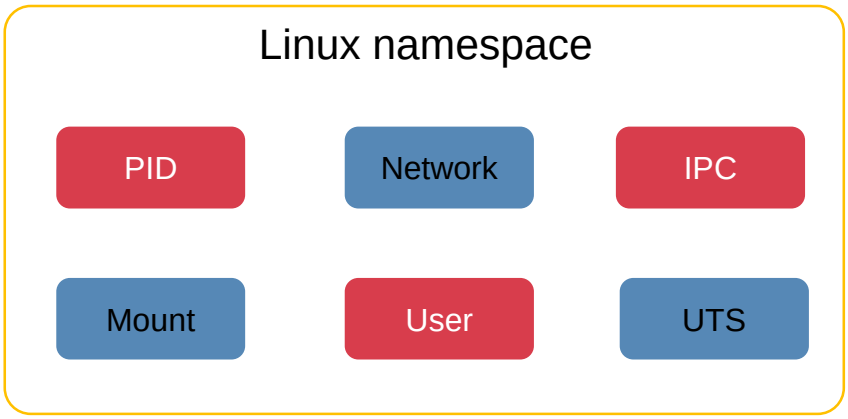
2.1 资源隔离（Namespace）和分组（Cgroup）

容器的底层核心技术是 Linux 的 Namespace 和 Cgroup 技术。

2.1.1 Namespace 介绍

Linux namespace 提供了一种内核级资源隔离的方法。不同类型的 Namespace 可以实现不同类型资源的隔离。缺省情况下，管理员创建容器的时候，Linux 内核会为该容器创建对应的 Namespace 实例。各容器使用自己的 Namespace 实例，在该 Namespace 实例中进行资源的创建和修改。各 Namespace 实例之间互不影响，从而实现资源的隔离。如果两个容器要共享某 Namespace，可以通过指定容器运行参数来实现。例如，执行“docker run --name tftpd --network container:comware tftpd”命令，可运行一个 TFTP server 容器，使得该容器和 Comware 共享命名空间。

图3 Comware V9 系统支持的 Linux namespace



如图 3 所示，Comware V9 系统通过 Namespace 实现了如下资源的隔离：

- PID（Process ID，进程 ID）：通过 PID namespace 实现进程 ID 的隔离。PID namespace 使得不同容器上的进程可以使用相同的 ID，各容器上进程独立编号。容器内可以看到该容器的进程及其所有子进程，但是无法看到宿主机和其它容器的进程；在宿主机上可以看到所有容器的所有进程及其子进程。
- Network：通过 Network namespace 实现容器间网络资源的隔离，包括网络设备、ARP 表、路由表、防火墙规则、IP 地址等。
- IPC（Inter-Process Communication，进程内通信）：System IPC 通信机制、POSIX 消息队列。通过 IPC namespace 实现信号量/消息队列/共享内存等的隔离，使得不同 IPC namespace 下的进程无法通过 IPC 机制进行通信。
- Mount（文件系统的挂载点）：Mount namespace 通过隔离文件系统挂载点来实现不同容器之间文件系统的隔离，不同 Mount namespace 中的文件系统互相不影响。
- User：用户名和用户组名。User namespace 实现了容器间用户和用户组的隔离。
- UTS（UNIX Time-sharing System）：通过 UTS namespace 可实现主机名和域名的隔离，使得每个容器可以拥有独立的主机名和域名。

2.1.2 Cgroup 介绍

Linux Cgroup 提供了一种为应用（进程）组织和分发系统资源的机制，它通过 Cgroupfs 文件系统来控制应用对不同类型资源的使用。Cgroup 还支持对资源进行度量，并限制每个应用使用的资源数量，以避免单个应用过多消耗系统资源，而导致其它应用无法正常运行。

Cgroup 主要提供如下几大功能：

- 资源限制：限制容器对系统资源的使用，如 CPU 节点、CPU 时间、内存节点、内存大小、I/O 设备等。
- 资源统计：对系统资源的使用进行统计。
- 任务分组：将不同资源需求的进程进行分组，将相同资源需求的进程划分到同一分组，以便组内的进程共享一套资源，方便配置和管理。

在 Cgroup 中，每种资源由一个单独的子系统进行分组和控制，常用的子系统包括：

- cpu 子系统：为每个任务分组设置一个 CPU 权重值，操作系统根据 CPU 权重来限制各任务分组对 CPU 的使用。
- cpuset 子系统：对于多核 CPU，该子系统可以设置进程组只能在指定的核上运行。
- cpuacct 子系统：该子系统用于生成进程组内进程的 CPU 使用情况报告。
- memory 子系统：该子系统用于以页为单位限制进程对内存的使用，比如设置进程组的内存使用上限，同时可以生成内存使用情况报告。
- blkio 子系统：该子系统用于限制每个块设备的读写。
- device 子系统：通过该系统可以限制进程组对设备的访问，即允许或禁止对设备进行访问。
- freezer 子系统：该子系统可以使得进程组中的所有进程挂起。
- net-cls 子系统：该子系统提供对网络带宽的访问限制，比如限制进程的发送带宽和接收带宽。

例如，执行“docker run –it –name test --cpu-period=100000 --cpu-quota=20000 ubuntu /bin/bash”命令，通过 ubuntu 镜像创建 test 容器时，使用 cpu 子系统可以限制该容器最多可占用 20%的 CPU 资源。

2.2 容器镜像

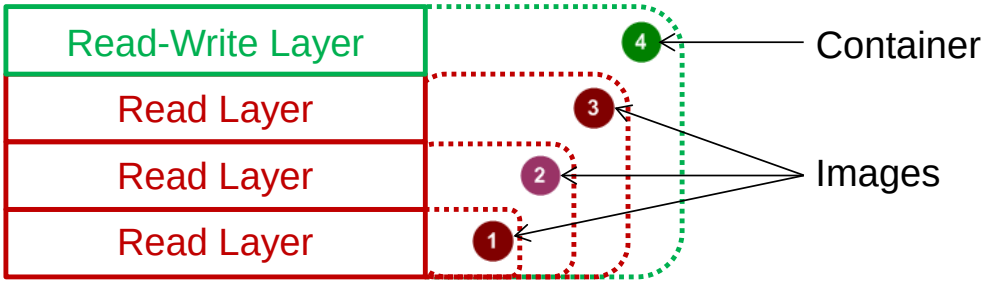
2.2.1 容器镜像简介

容器通过镜像来描述，镜像是 Docker 容器的核心技术。镜像是一种轻量级、可执行的独立软件包，不仅包含了容器运行时所需要的程序、库、资源、配置等文件，还包含了运行时需要的配置参数（如匿名卷、环境变量、用户等）。运行镜像，就能创建并运行容器。

Docker 采用分层文件系统（aufs/unionfs/overlay2）来组织和管理容器根文件系统。如图 4 所示，一个 Docker 镜像由多个只读的镜像层组成，运行的容器在 Docker 镜像上增加一层可读写的容器层。容器在当前运行过程中对文件的任何修改都只保存在容器层。所有镜像层和容器层联合在一起组成一个统一的文件系统，用户最终看到的是一个叠加之后的文件系统。

如果重启容器，容器层数据会自动删除，当前运行过程中对容器的操作不会影响到镜像本身。如果保存容器，将容器层固化成只读的镜像层，便可以将当前运行过程中的数据保存下来，容器重启后，还能继承重启前的配置。

图4 分层文件系统示意图



分层文件系统中的常用文件操作包括：

- 添加文件

  在容器中创建新文件时，新文件被添加到容器层中。

- 读取文件

  在容器中读取某个文件时，Docker 会从上往下依次在各层中查找该文件。

- 修改文件

  在容器中修改已存在的文件时，Docker 会从上往下依次在各镜像层中查找该文件，找到后将其复制到容器层，然后修改。

- 删除文件

  在容器中删除文件时，Docker 会从上往下依次在镜像层中查找该文件，找到后，会在容器层中标记该文件已删除，实际上并未真的删除镜像层的文件。

2.2.2 容器镜像的构建

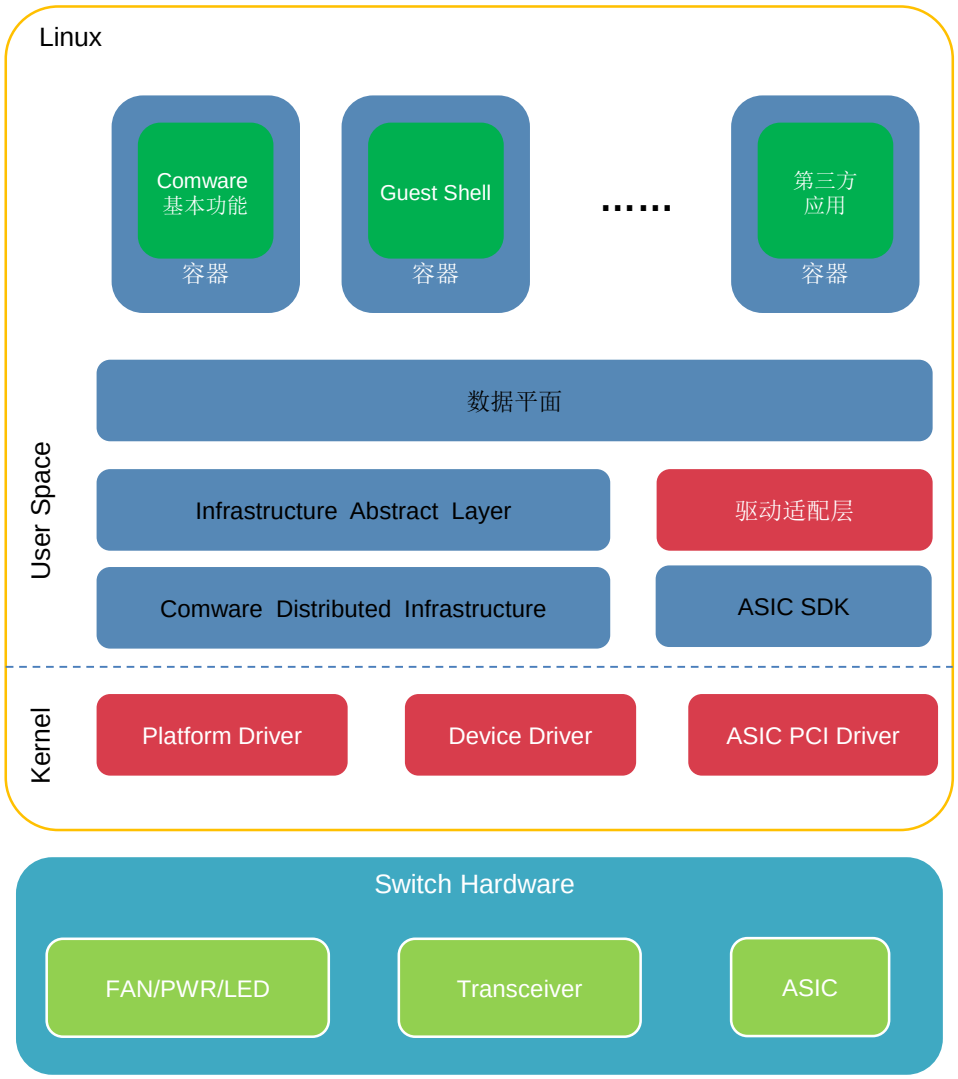
Dockefile 是构建 Docker 镜像的指令集合，里面包含了构建一个镜像所需的所有指令。通过 **`docker build`** 命令读取 Dockefile，按照 Dockefile 中的指令可以方便地编译、打包、生成镜像。

Dockerfile 中可以基于其他已构建好的镜像，并在此基础上叠加新的交付件，以及修改容器的执行入口，这样可以方便的在原有镜像基础上，依据新的业务需求去扩展镜像，达到复用的效果，并减少重复工作。

3 ComwareV9 容器技术实现

3.1 Comware V9容器化架构

图5 Comware V9 容器化架构



如图 5 所示，Comware V9 采用容器化架构，为 Comware 以及第三方应用提供独立的运行环境：

- 系统配置管理以及核心业务运行在 Comware 容器中。
- Guest Shell 容器内嵌了 CentOS 7 系统，用户可在 Guest Shell 容器内安装基于 CentOS 7 系统的应用，来扩展 Comware 的功能。
- 支持第三方容器。用户使用简单的命令行，可以在 H3C 设备上安装第三方容器，通过容器化方式来便捷地扩展 Comware 的功能。在 Comware V7 上部署第三方软件时，需要在 Comware V7 的交叉编译环境下编译第三方软件的源代码，编译通过后才能在 Comware 上运行第三方软件。与 Comware V7 的集成方式相比，Comware V9 通过容器来部署第三方应用的集成方式更加灵活方便，使得 Comware 的开放性显著提升。

- Comware 容器和用户创建的容器独立运行，互不影响。它们均运行在 Linux 内核之上，直接使用物理设备的硬件资源，利用 Linux namespace 和 Cgroup 等技术来实现容器间的隔离和资源分配控制。
- Comware V9 封装了 Docker 的容器管理功能，并可根据产品需求对第三方应用可使用的容器资源最大值进行定制，保障系统转发平面和控制平面对资源的使用。Comware V9 封装了 Docker、kubelet 的命令界面，确保 Docker、kubelet 命令也受 Comware 用户认证体系及 RBAC 的管理，只有授权用户才可以使用。
- Comware V9 支持 Kubernetes 技术，可作为一个工作节点接受 Kubernetes Master 的调度和管理，实现第三方应用的大规模、集群化部署。

3.2 Comware V9支持的容器

Comware V9 支持三种容器：

- Comware 容器：Comware 容器中封装了 Comware 系统，提供交换、路由等基本功能。
- Guest Shell 容器：H3C 提供的官方容器，它内嵌了 CentOS 7 系统。用户可在该容器中安装基于 CentOS 7 系统的第三方应用。
- 第三方应用容器：第三方开发并发布的、基于 Docker 的容器，用于扩展 Comware 系统的功能。

3.2.1 Comware 容器

把 Comware 系统交付件和基础运行环境打包成容器镜像，并以容器的方式运行 Comware 系统，这个容器就是 Comware 容器。

设备上电启动时，系统会自动创建和运行 Comware 容器。Comware 容器是系统的管理容器，负责管理设备上所有的硬件资源。用户通过 Console 登录的是 Comware 容器的命令行界面。通过 Comware 容器的命令行，用户可以创建、运行、停止、销毁 Guest Shell 容器和第三方应用容器。用户被禁止通过 **docker** 相关命令行来操作 Comware 容器。

3.2.2 Guest Shell 容器

为便于用户在 Comware 系统中安装和运行第三方软件，且不影响 Comware 容器的运行，H3C 提供了另一种官方容器——Guest Shell 容器。Guest Shell 容器内嵌了 CentOS 7 系统，通过 Guest Shell 容器，用户可在 H3C Comware V9 设备上安装基于 CentOS 7 系统开发的应用。例如，用户可在 Guest Shell 容器内安装网络服务器软件、数据库和网页的开发工具，如 Apache、Sendmail、VSFtp、SSH、MySQL、PHP 和 JSP 等，来部署邮件和 Web 等服务。用户还可以在 Guest Shell 容器内程序语言与开发工具，如 gcc、cc、C++、Tcl/Tk、PIP（Python 的包管理工具）等，来实现对 Comware 应用进行二次开发的需求。

Guest Shell 容器支持 CentOS 7 官方 Docker 镜像版本所支持的所有命令。

Guest Shell 容器和宿主机、Comware 容器、第三方应用容器完全隔离，用户在 Guest Shell 容器中安装、操作应用不会影响其它容器的运行。

Guest Shell 容器与直接使用 CentOS 7 的镜像文件运行 CentOS 7 容器比较，还具有以下优势：

- Guest Shell 容器由 H3C 提供镜像文件，在设备上安装该镜像文件后，即能运行 Guest Shell 容器。

- Comware 系统对 Guest Shell 容器的命令行进行了封装。登录 Comware 系统后，用户执行简单的 **guestshell** 命令行，即可便捷地对 Guest Shell 容器进行管理和维护。

3.2.3 第三方应用容器

有些应用是以容器的形式发布的。为了支持这类应用，Comware 系统支持在其上部署基于 Docker 的第三方应用容器。用户可在设备上创建和使用基于 Docker 容器的应用，用于实现信息采集、设备状态监控等用户自定义功能。例如，每台设备运行一个资源采集的 Agent 容器，用来实时采集设备的 CPU、内存、流量等信息，并上报至监控设备统一管理。

第三方应用容器和 Comware 容器、Guest Shell 容器之间互相隔离。容器化应用技术将第三方应用的影响范围控制在容器内。

3.3 容器启动过程

Comware V9 系统中容器的启动过程如下：

(1) 设备加电后会启动基础系统的服务管理进程；

(2) 启动 Docker 服务；

(3) Docker 服务启动完成后，触发构建 Comware 容器镜像、创建 Comware 容器，建立 Comware 根文件系统中所需的目录映射。

(4) 基础系统启动完成后，通过自动调用 **docker** 命令启动 Comware 容器，并将控制权交给 Comware 容器，由 Comware 的服务管理进程继续进行容器内的初始化工作；

(5) 等待 Comware 容器启动完成后，自动登录到 Comware 容器中的命令行界面，最终把控制权交给用户。

(6) 用户登录 Comware 系统，使用 **guestshell start** 命令启动 Guest Shell 容器，使用 **docker** 命令启动第三方应用容器。

针对 Docker 服务，Comware V9 创建了两个 Docker 实例：

- Comware Docker 实例：用于管理 Comware 容器。
- 用户 Docker 实例：用于管理 Guest Shell 容器和第三方应用容器。

这两个 Docker 实例将 Comware 容器、Guest Shell 容器和第三方应用容器从管理上进行了分离，并通过指定通信通道将 Docker 指令分发到不同的实例。用户从命令行界面下发的 Docker 指令只会发送到用户 Docker 实例，从而避免用户误操作 Comware 容器，例如误删 Comware 容器。

提示

如果需要确保容器对外提供服务的 IP 地址不变，在启动容器时需要使用--ip 或--ip6 参数指定容器的 IP 地址。否则，容器重启后需要自行向外界重新宣告服务地址，才能保证服务被发现。因为，在未指定--ip 或--ip6 参数的情况下，容器每次启动时，系统分配给容器的 IP 地址可能不一样。

3.4 容器保存和恢复

由于 Docker 采用分层文件系统，容器运行过程中对容器运行参数的修改只保存在临时缓存（容器分层文件系统中的容器层）中。如果重启容器，容器会使用镜像文件启动，恢复到容器对外发布时镜像文件中定义的状态。

为便于容器继承重启前的状态继续运行，Comware V9 提供容器保存功能，该功能集成在 **save** 命令中。执行 **save** 命令且选择保存容器文件后，设备会自动将当前运行的所有容器的所有运行数据打包压缩到持久化存储介质中，文件名形如 flash:/third-party/dockerfs.cpio.gz。容器重启时，会将该文件从存储介质中解压到/var/lib/docker，然后再启动容器。容器启动完成后，会恢复到容器文件中记录的状态。


提示

如果希望在设备重启时自动运行容器，则需要在创建容器时指定--restart=always 参数，指定容器重启策略为 always。

3.5 容器网络实现

Comware 可以为第三方容器中的应用提供网络支持。运行在设备上的第三方应用，可以与 Comware 中的应用通信，也可以通过 Comware 与外界通信。第三方容器中的应用与 Comware 中的应用之间的通信流量称为东西向流量，第三方容器中的应用通过 Comware 与外界通信的流量称为南北向流量。

Docker 使用命名空间来实现容器间的隔离。其中，网络命名空间用于隔离网络资源，例如 IP 地址、IP 路由表、/proc/net 目录、端口号等。用户在设备上创建容器时，可以选择是否和 Comware 共享网络命名空间。是否共享网络命名空间会影响容器和外界的通信方式以及网络参数的配置。


说明

- Guest Shell 容器需要使用 Guest Shell 镜像来创建。使用 **guestshell start** 命令创建的 Guest Shell 容器和 Comware 容器不共享网络命名空间。如果使用 **docker run** 命令创建 Guest Shell 容器，管理员可以选择和 Comware 容器共享网络命名空间或者和不共享网络命名空间。
- Guest Shell 容器与目的节点的通信过程同第三方容器与目的节点的通信过程，下面以第三方容器为例，说明容器的网络实现。

3.5.1 第三方容器与 Comware 共享网络命名空间

1. 原理介绍

共享网络命名空间时，第三方容器中的应用类似于 Comware 的应用，使用 Comware 的网络参数，包括接口、IP 地址、路由表、端口号等，与目的节点通信。

第三方应用与目的节点通信时，分为以下两种情况：

- 如果目的节点和 Comware 位于同一网段，则管理员不需要为第三方应用配置其他网络参数，目的节点使用 Comware 的 IP 地址即可与第三方应用通信。Comware 根据报文协议号或目的端口号将报文发送给对应的应用处理，如果 Comware 中的应用和第三方应用侦听的协议号或目的端口号相同，则优先发送给 Comware 中的应用处理。

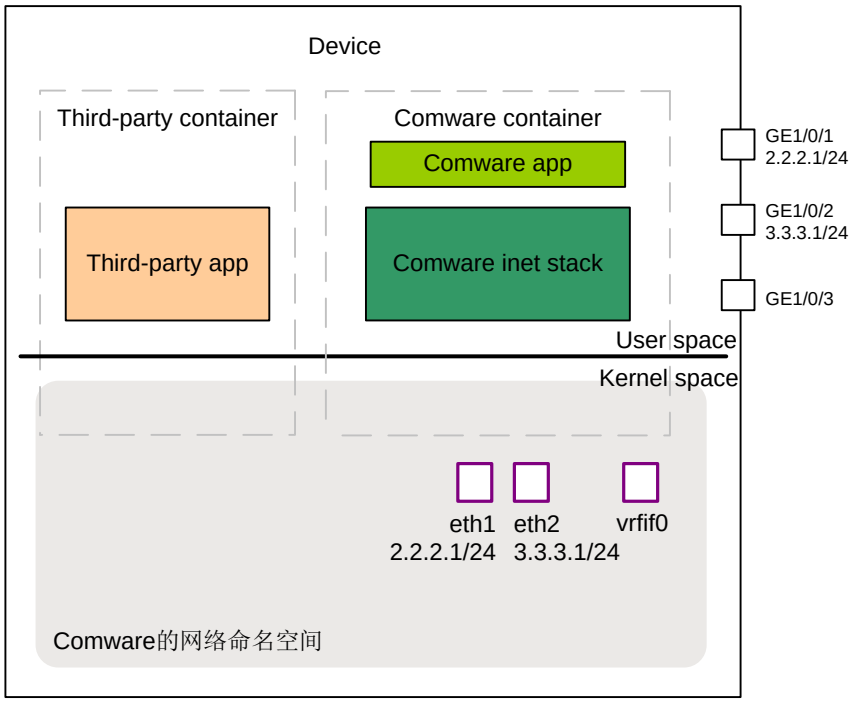
  管理员为 Comware 三层接口配置 IP 地址后，Comware 会在 Linux 内核创建对应的 eth 接口来实现 Linux 内核和 Comware 的通信，并将该 eth 接口的 IP 地址设置为该三层接口的 IP 地址，触发内核生成该 IP 地址所在网段的直连路由。Linux 内核转发报文时会查找这些路由，并且通过 eth 接口将报文发送给 Comware；Comware 也会通过 eth 接口将报文发送给第三方容器中的应用。
- 如果目的节点和 Comware 位于不同网段，则需要为第三方应用配置源 IP 地址，目的节点与这个源 IP 地址之间必须路由可达。第三方应用使用这个源地址与目的节点进行跨网段通信。

  Comware 会在内核创建一个虚接口 vrfif0，用来做容器的缺省出接口，并下发本容器内的缺省路由，缺省路由的出接口为 vrfif0，缺省路由的源地址为 Comware 中某三层接口的地址（缺省为 LoopBack0 接口的地址，可通过命令行修改）。内核转发模块会使用该缺省路由，并且通过 vrfif0 口将报文发送给 Comware，Comware 再发送给目的节点；目的节点将报文发送给 Comware，Comware 通过 vrfif0 口将报文发送给第三方容器中的应用。

以图 6 中的 Third-party container 为例，为 GE1/0/1 配置 IP 地址 2.2.2.1/24，给 GE1/0/2 配置 IP 地址 3.3.3.1/24，给 Loopback 0 接口配置 IP 地址 10.10.3.3/24 之后，在 Third-party container 上查看 Third-party container 的路由表，可以看到如下路由。

- 一条缺省路由，规则为发往非直连网段的报文，源地址为 10.10.3.3，需要将报文通过 vrfif0 交给 Comware 处理。
- 一条对应 Loopback 0 接口的直连路由，规则为发往 10.10.0.0/24 网段的报文，出接口为 Loop0。
- 一条对应 GE1/0/1 接口的直连路由，规则为发往 2.2.2.0/24 网段的报文，需要将报文通过 eth1 交给 Comware 处理。
- 一条对应 GE1/0/2 接口的直连路由，规则为发往 3.3.3.0/24 网段的报文，需要将报文通过 eth2 交给 Comware 处理。
- 一条本机路由，规则为发往 127.0.0.0/8 网段的报文，需要将报文通过 InLoop0 交给本机处理。

图6 第三方容器与 Comware 共享网络命名空间时网络互通原理意图



2. 东西向流量

第三方容器中的应用与 Comware 中的应用之间的通信流量称为东西向流量。

如图 7 所示，第三方容器中的应用通过内部接口将报文发送给 Comware 应用的过程如下：

(1) 第三方容器将报文发送给内核协议栈（Inet stack）。

(2) 内核协议栈将报文发送给内核转发模块（Kernel forwarding）处理。

(3) 由于目的 IP 为本机地址，内核转发模块判断需要将报文转发给 Comware 应用。内核转发模块根据 Comware 下发的转发规则，将报文发送给内核协议栈或者通过 ethX 接口发送给 Comware 协议栈处理。

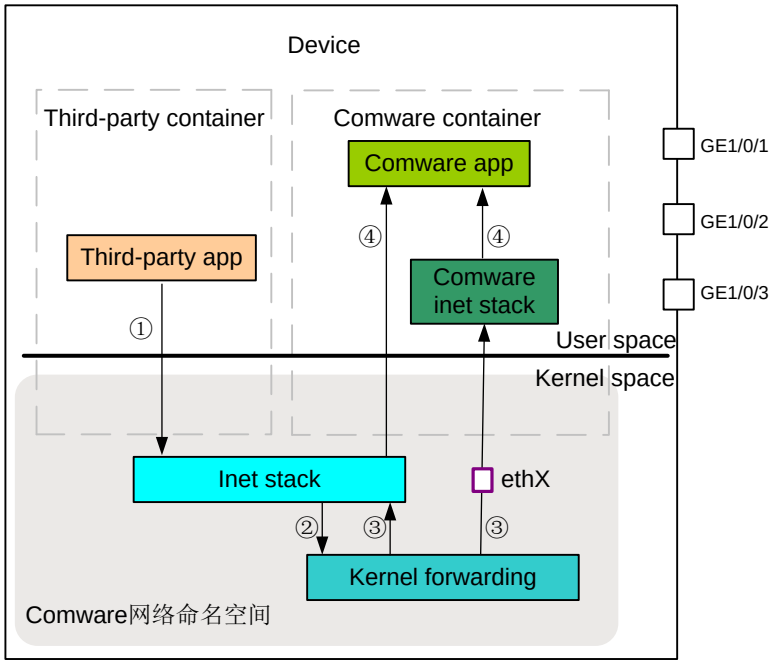
(4) 根据应用层端口号，内核协议栈或 Comware 协议栈将报文发送给 Comware 应用。

反方向，对于 Comware 应用发送给第三方容器中应用的报文过程与此类似，不再赘述。

说明

同一设备内不同第三方容器应用之间的通信和第三方容器应用与 Comware 应用之间的通信流程类似，直接通过内核协议栈和内核转发模块互通，此处不再赘述。

图7 共享网络命名空间场景下，第三方应用向 Comware 应用发送报文流程图



3. 南北向流量

如图 8 所示，第三方容器中应用发送报文给目的节点的过程为：

(1) 第三方应用将报文发送给 Linux 内核协议栈（Inet stack）处理。

(2) Linux 内核协议栈将报文转发给内核转发模块（Kernel forwarding）处理。

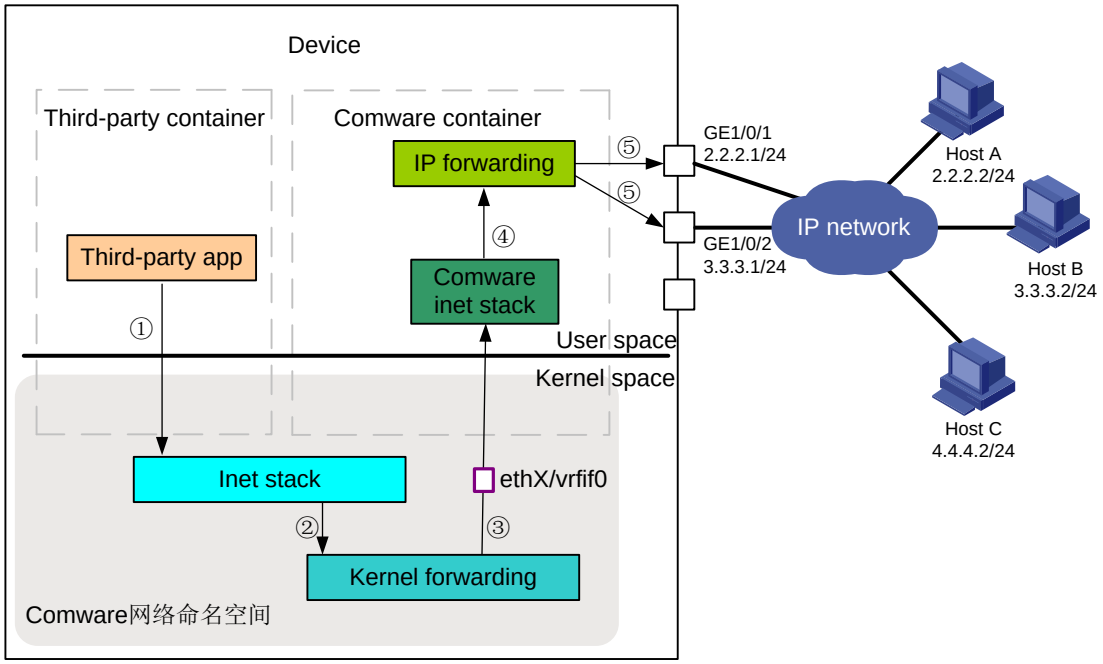
(3) 内核转发模块匹配路由，通过 ethX 接口或者 vrfif0 接口将报文发送给 Comware 协议栈。

(4) Comware 协议栈将报文发送给 Comware 转发模块。

(5) Comware 转发模块进行查表转发，将报文从物理接口发送出去。

反方向，对于外界目的节点发送给第三方容器中应用的报文，也是通过 Comware 转发、Comware 协议栈、内核转发模块，最终通过内核协议栈将报文发送给第三方应用。

图8 共享网络命名空间场景下，第三方应用向目的节点发送报文流程图



3.5.2 第三方容器与 Comware 不共享网络命名空间

1. 原理介绍

如果希望第三方容器使用独立于 Comware 的 IP 地址与其它容器/设备通信，或者不希望第三方应用看到 Comware 的地址和接口，在创建第三方容器时，可以选择第三方容器不与 Comware 共享网络命名空间。

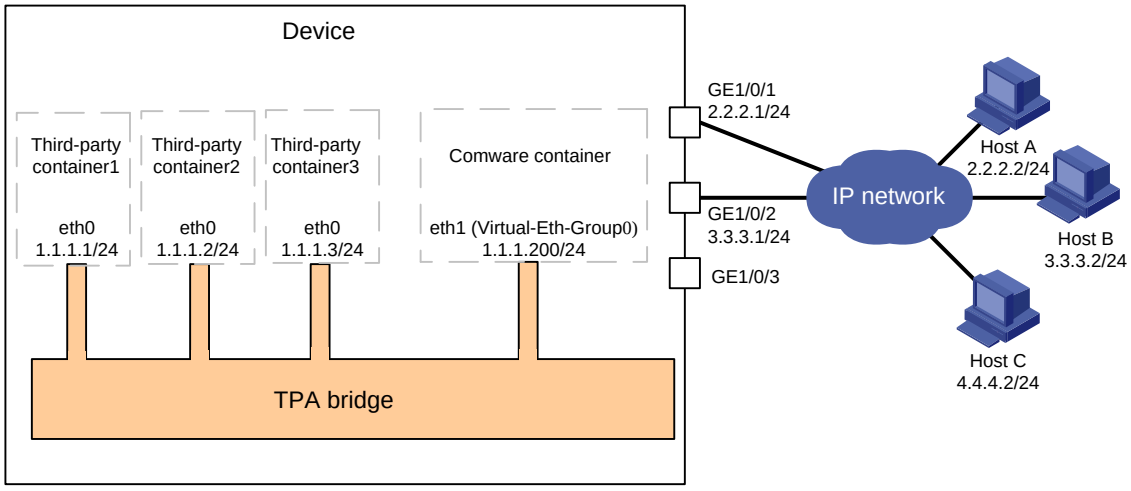
不共享网络命名空间时，Comware 使用 Virtual-Eth-Group0 接口为第三方容器及其应用提供网络支持。用户创建 Virtual-Eth-Group0 接口时，Comware 会同时创建一个 TPA（Third-party Application，第三方应用） bridge，不同容器中的第三方应用通过 TPA bridge 互相通信以及和外界通信。

如图 9 所示，用户在设备上创建 Virtual-Eth-Group0 接口，并为该接口配置 IP 地址 1.1.1.200/24，系统会自动创建一个 TPA bridge 和虚拟网口 eth1。当用户指定 TPA bridge 创建 Docker 容器并选择和 Comware 不共享网络命名空间时，系统会为 Docker 容器创建一个虚拟接口 eth0，并为该接口分配 Virtual-Eth-Group0 接口所在网段中空闲的 IP 地址（例如图中的 1.1.1.1/24、1.1.1.2/24 和 1.1.1.3/24），并将容器的缺省网关设置为 Virtual-Eth-Group0 接口的 IP 地址。

以图 9 中的 Third-party container3 为例，在 Third-party container3 上查看 Third-party container3 的路由表，可以看到两条路由。

- 一条缺省路由，规则为发往非 1.1.1.0/24 网段的报文，下一跳为 1.1.1.200，需要将报文通过 eth0 交给 Comware 处理。
- 一条直连路由，规则为发往 1.1.1.0/24 网段的报文，出接口为 eth0。

图9 第三方容器与 Comware 不共享网络命名空间时网络互通原理图



2. 东西向流量

同一设备上，不同容器中的第三方应用处于相同网段，可以通过 TPA bridge 和 eth0 直接互通。

第三方容器中应用通过内部接口将报文发送给 Comware 应用，如图 10 所示，流程如下：

(1) 第三方容器内的应用将报文发送给 Linux 内核协议栈。

(2) 内核协议栈将报文发送给内核转发模块处理。

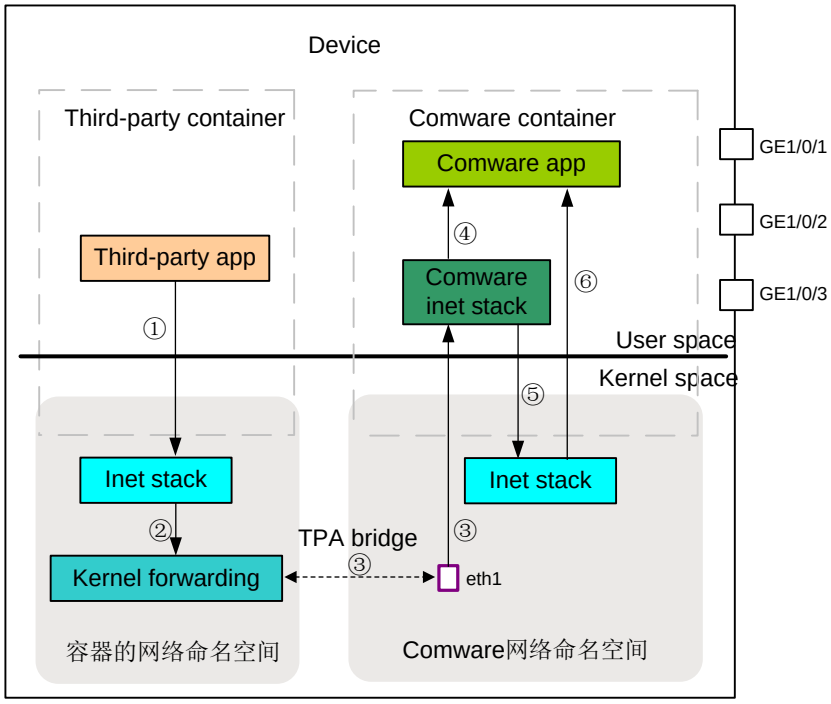
(3) 内核转发模块经过 eth0—>TPA bridge—>eth1，将报文从第三方容器的网络命名空间发送给 Comware 容器，Comware 容器匹配内核路由，将报文通过 vrfif0 接口发送给 Comware 协议栈。

(4) Comware 协议栈如果支持处理该应用的报文，则报文发送给 Comware 应用；Comware 协议栈如果不支持处理该应用的报文，则报文发送给 Linux 内核协议栈。

(5) 内核协议栈将报文发送给 Comware 应用。

反方向的，对于 Comware 应用发送给第三方容器中应用的报文处理流程与此类似，不再赘述。

图10 不共享网络空间场景下，第三方应用向 Comware 应用发送报文流程图



3. 南北向流量

南北向流量需要经过 Comware 从物理接口发给目的节点。目的节点需要使用第三方容器的 IP 地址和第三方容器中的应用通信。

如图 11 所示，第三方容器中应用发送报文给目的节点的过程为：

(1) 第三方应用将报文发送给 Linux 内核协议栈处理。

(2) Linux 内核协议栈将报文转发给内核转发模块处理。

(3) 内核转发模块 eth0—>TPA bridge—>eth1，将报文从第三方容器的网络命名空间发送给 Comware 协议栈。

(4) Comware 协议栈将报文发送给 Comware 转发模块。

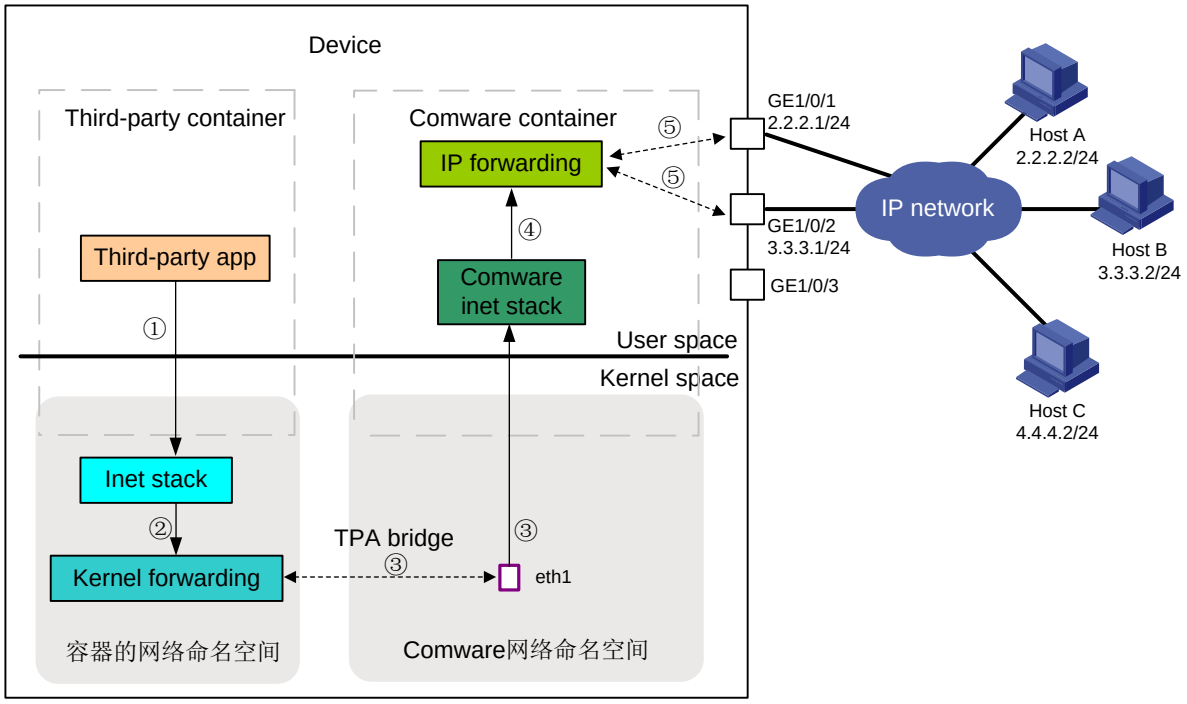
提示

管理员可以根据实际需求选择是否在 Comware 发送报文前对第三方应用的流量进行 NAT（Network Address Translation，网络地址转换）处理。如果可以直接对外宣告第三方容器的 IP 地址，则不需要进行 NAT 处理；否则，可以通过 NAT 将第三方容器的 IP 地址转换成 Comware 的地址与外界通信，对外界隐藏第三方容器的存在。

(5) Comware 转发模块进行查表转发，将报文从物理接口发送出去。

反方向，外界目的节点发送给第三方容器中应用的报文处理流程与此类似，不再赘述。

图11 不共享网络空间场景下，第三方应用向目的节点发送报文流程图



3.5.3 为宿主机上的应用提供网络支持（与 Comware 不共享网络空间）

1. 原理介绍

宿主机和 Comware 不共享网络空间。宿主机上的接口由 Comware 来驱动和管理，直接运行在宿主机上的应用需要通过 Comware 的接口与外界通信。例如，Docker 和 Kubelet 虽然是 Comware 中集成的功能，但是它们直接运行在宿主机上，Docker、Kubelet 应用与 Comware 不共享网络空间。当 Docker 应用从 Docker hub 上拉取镜像或者向 Docker hub 上传镜像时，需要 Comware 为 Docker 提供网络支持。当 Kubelet 要接受 Kubernetes Master 的调度和管理时，也需要 Comware 为其提供网络支持。

如图 12 所示，为了实现 Comware 为宿主机上的应用提供网络支持，设备启动时，会自动执行以下操作：

(1) 在宿主机的 Linux 内核创建缺省网桥 Docker0 bridge，用于连接宿主机的网络命名空间和 Comware 的网络命名空间。

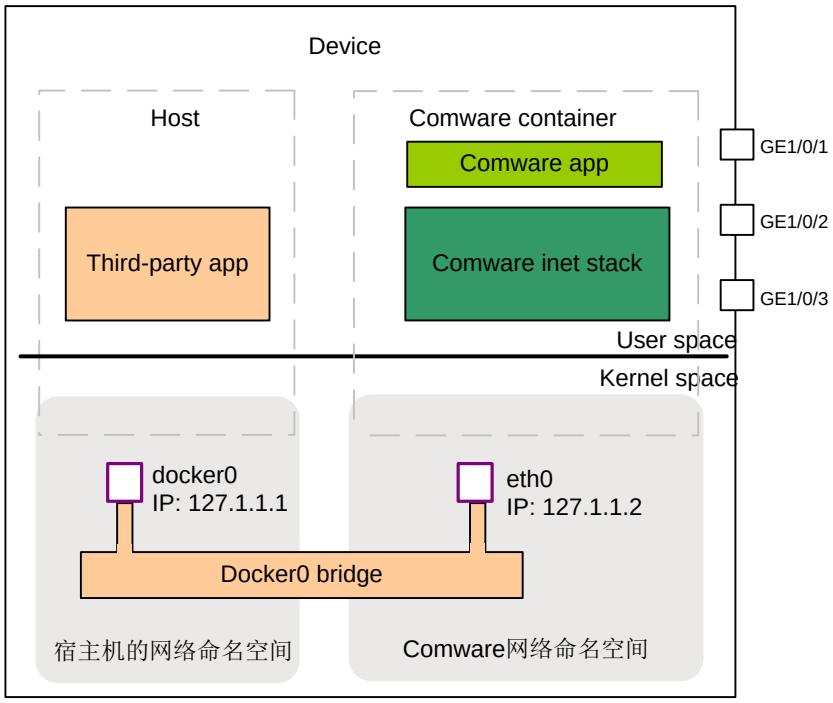
(2) 在 Docker0 bridge 宿主机的网络命名空间侧，创建 docker0 接口并设置其 IP 地址为 127.1.1.1。

(3) 在 Docker0 bridge Comware 网络命名空间侧，创建 eth0 接口并设置其 IP 地址为 127.1.1.2。

(4) 在宿主机的网络空间生成两条路由。

- 一条缺省路由，规则为发往除 Comware 之外目的节点的报文，下一跳为 127.1.1.2，需要将报文通过 eth0 交给 Comware 处理。
- 一条直连路由，规则为发往 Comware 应用的报文，出接口为 docker0。

图12 Comware 为宿主机提供网络支持原理图



2. 东西向流量

如图 13 所示，宿主机中的应用通过内部接口将报文发送给 Comware 应用的过程如下：

(1) 宿主机内的应用将报文发送给 Linux 内核协议栈。

(2) 内核协议栈将报文发送给内核转发模块。

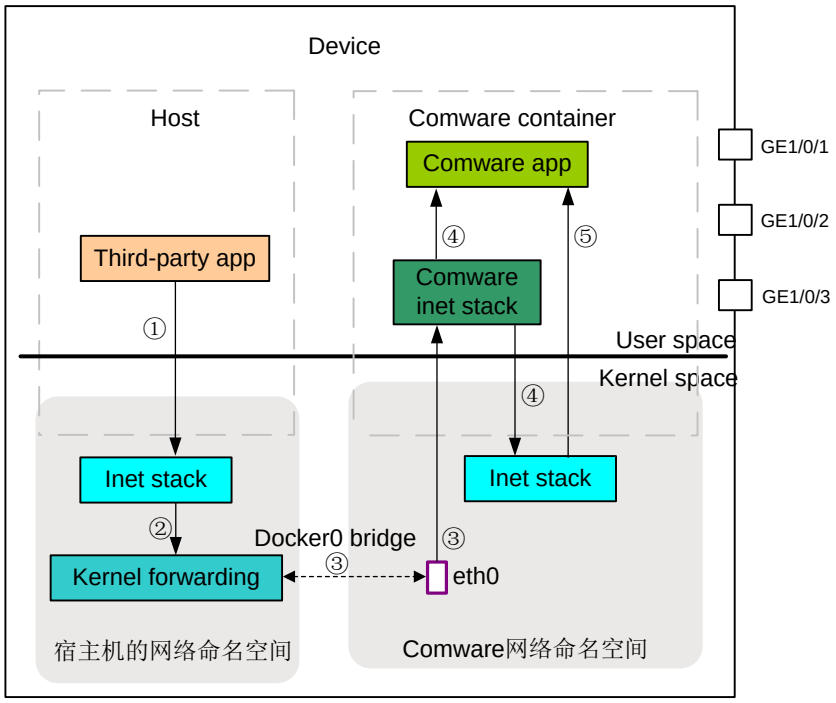
(3) 内核转发模块经过 docker0—>Docker0 bridge—>eth0，将报文从宿主机的网络命名空间发送给 Comware 协议栈。

(4) Comware 协议栈如果支持处理该应用的报文，则报文发送给 Comware 应用；Comware 协议栈如果不支持处理该应用的报文，则报文发送给 Linux 内核协议栈。

(5) 内核协议栈将报文发送给 Comware 应用。

反方向的，对于 Comware 应用发送给第三方容器中应用的报文处理流程与此类似，不再赘述。

图13 宿主机向 Comware 应用发送报文流程图



3. 南北向流量

南北向流量需要经过 Comware 从物理接口发给目的节点。目的节点需要使用 Comware Loopback0 接口的 IP 地址和宿主机通信。

如图 14 所示，宿主机中应用发送报文给目的节点的过程如下：

(1) 宿主机中应用将报文发送给 Linux 内核协议栈处理。

(2) Linux 内核协议栈将报文转发给内核转发模块处理。

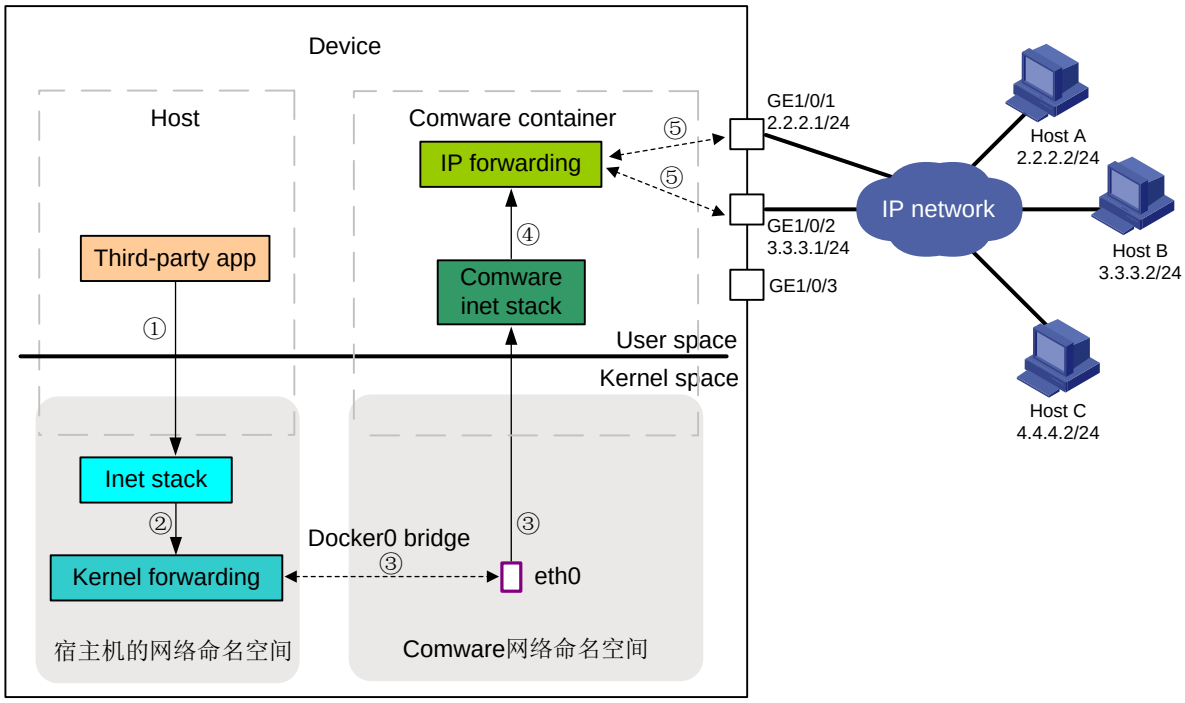
(3) 内核转发模块对报文进行 NAT 转换，将报文的源地址转换为 Comware 的 Loopback0 接口的 IP 地址，经过 docker0—>Docker0 bridge—>eth0，将报文从宿主机的网络命名空间发送给 Comware 协议栈。

(4) Comware 协议栈将报文发送给 Comware 转发模块。

(5) Comware 转发模块进行查表转发，最终将报文从物理接口发送出去。

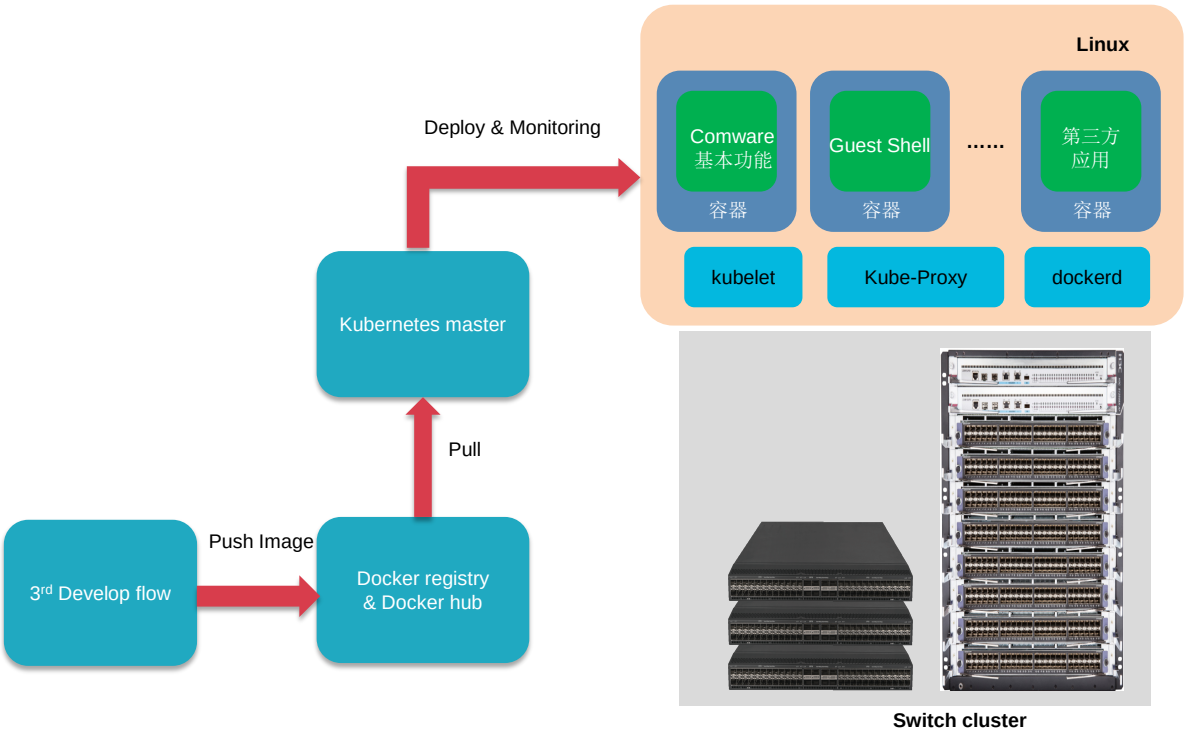
反方向，外界目的节点发送给第三方容器中应用的报文处理流程与此类似，不再赘述。

图14 宿主机向目的节点发送报文流程图



3.6 通过K8S管理容器

图15 Comware V9 支持 K8S 管理



如图 15所示，Comware V9 系统内部集成了 Kubelet，使得设备可作为 K8S 集群中的一个工作节点，接受 Kubernetes Master 的调度和管理，从而实现通过 K8S 对设备上运行的第三方容器进行大规模、集群化部署和集中式管理。对于管理员来说，在 Comware V9 设备上部署和管理容器，与在服务器上部署和管理容器的方法基本相同，不影响管理员的配置习惯。

3.7 容器使用限制

如果安装的第三方应用和 Comware 自带的应用冲突，设备会优先使用 Comware 应用。例如，用户开启了 Comware 中自带的 FTP server 应用，又在 Docker 容器中启用了 FTP server 应用。如果这两个应用使用的 IP 地址和端口号也相同，FTP client 通过 FTP 协议访问设备时，设备会使用 Comware 自带的 FTP server 响应请求。只有 Comware 自带的 FTP server 被关闭，才会使用 Docker 容器中的 FTP server 应用来响应请求。

不允许对 Comware 容器进行容器的相关操作（包括 Docker 命令行及 Kubelet 下发的指令）。

对于无专用硬盘或 Flash 的 Comware 设备，无法保存容器的运行状态，建议使用无状态的容器。

为了防止第三方容器过度消耗资源，而对 Comware 容器等设备主要组件产生不利影响，设备出厂时对每个第三方容器可使用的 CPU、内存资源进行了限额。在使用 **`docker run`** 命令启动第三方容器，并通过**`--cpuset-cpus`**、**`--cpuset-shared`**、**`--memory`** 参数指定可使用的 CPU、内存资源超出设备出厂限额时，以设备出厂限额为准。

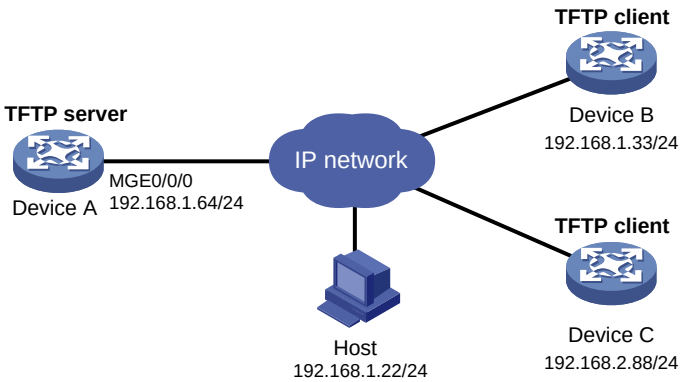
4 典型组网应用

4.1 共享网络命名空间场景下容器化部署第三方应用

Comware 系统支持 TFTP client 功能，但不支持作为 TFTP server。使用容器功能，无需修改 Comware 系统软件，甚至无需获取 Comware 系统编程接口，直接在 Device A 上安装 TFTP server 容器，即可在 Device A 上部署 TFTP server，使得设备能对外提供 TFTP server 服务。

如果 IP 地址紧张，TFTP server 可以和设备共用 IP 地址 192.168.1.64。这样，Device A、Device B 和 Device C 均能通过地址 192.168.1.64 访问 TFTP server，将各自的配置文件上传到 Device A。

图16 共享网络命名空间场景下容器化部署第三方应用组网图

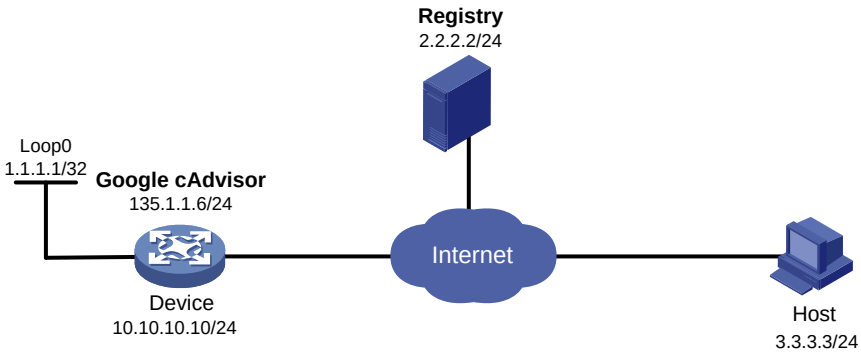


4.2 不共享网络命名空间且不进行NAT场景下，容器化部署第三方应用

管理员需要设备提供一款管理软件，用于监控当前设备的资源使用情况（包括 CPU、内存、网络、文件系统等）。如果基于 Comware 系统做增量开发，需要假以时日，并投入相当的人力物力。使用容器功能，可以立即从公共镜像仓库拉取 cAdvisor 镜像文件，在设备上以此镜像文件运行 cAdvisor 容器，即可实现监控需求。

如果管理员只想给部分低级别的用户开放 cAdvisor 访问权限。为了不暴露设备的 IP 地址（10.10.10.10），cAdvisor 使用固定公网 IP 地址 135.1.1.6，管理员在创建容器时，定制成不共享网络命名空间且不进行 NAT 地址转换。

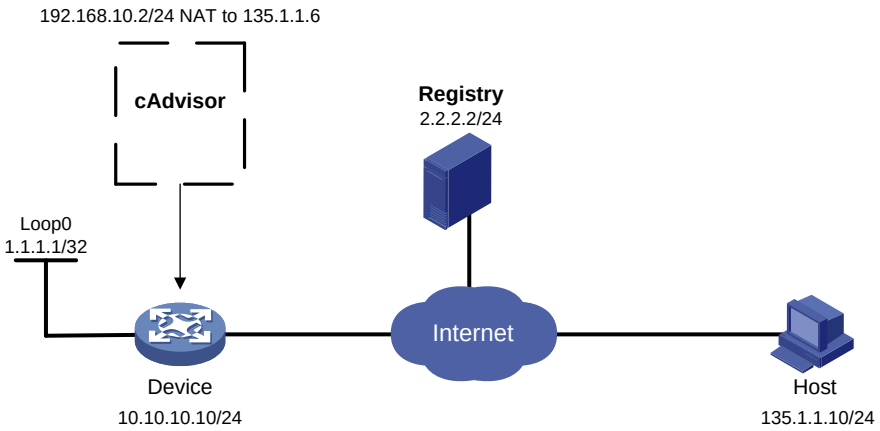
图17 不共享网络命名空间且不进行 NAT 场景下，容器化部署第三方应用组网图



4.3 不共享网络命名空间且进行NAT场景下，容器化部署第三方应用

从镜像仓库（IP 地址为 2.2.2.2）拉取镜像文件，在设备上安装第三方应用，用于监控当前设备的资源使用情况（包括 CPU、内存、网络、文件系统等）。设备的 IP 地址为 10.10.10.10，cAdvisor 使用私网地址 192.168.10.2/24。通过定制成不共享网络命名空间且配合部署 NAT 地址转换功能，可实现在 Host 上可以通过固定 IP 地址 135.1.1.6 访问 cAdvisor，并能从 cAdvisor 提供的 Web 页面中监控设备资源的使用情况。

图18 不共享网络命名空间且进行 NAT 场景下，容器化部署第三方应用组网图



5 参考文献

- OCI 官网：https://opencontainers.org/
- OCI 运行时规范：https://github.com/opencontainers/runtime-spec
- OCI 镜像规范：https://github.com/opencontainers/image-spec
- Kubernetes 官网文档：https://kubernetes.io/docs
- Docker 官网文档：https://docs.docker.com