A Dissertation Submitted to Zhejiang University for the Degree of Master of Engineering

TITLE: Design and implementation of Docker network system based on macvlan

Author: Mingzhen Feng

Supervisor: Zhongdong Huang

Subject: Software Engineering

College: Computer Science and Technology

Submitted Date: 2016.1.10

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得浙江大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：冯明振　　签字日期：2016 年 3 月 4 日

学位论文版权使用授权书

本学位论文作者完全了解 浙江大学 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 浙江大学 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：冯明振　　导师签名：

签字日期：2016 年 3 月 4 日　　签字日期：2016 年 3 月 7 日

摘要

Docker 作为时下最流行的容器技术，已经在云计算领域掀起了一股 Docker 热潮。Docker 容器相比虚拟机有诸多优势，如超高的资源利用率、超快的起停速度。此外，方便的镜像构造与管理机制也是 Docker 容器的一大优势和特点。这些特性使得应用可以快速地构建、发布、升级、测试等，成为了开发人员以及运维人员手中的利器。同时，Docker 容器进一步模糊了 IaaS 和 PaaS 的界限，成为了各个云计算厂商的新宠。

但是，Docker 技术也并不完美。其网络方面就是一大短板。首先，单主机的网络模式就使其在规模上和扩展上存在很大的问题，跨主机通信是业界亟待解决的问题之一。此外，Docker 的网络只提供了四种简单的网络模式，缺少多租户网络隔离、网络 QoS、服务发现等功能，不能满足生产环境下复杂的使用场景。

本文针对以上问题，提出并实现了基于 macvlan 的 Docker 容器网络系统。该系统主要由 macvlan daemon 和 macvlan 功能模块区组成。macvlan 功能模块区包括 Flat、DHCP、VLAN 三大网络模块以及 QoS 和服务发现等功能。Flat 网络用来解决跨主机通信以及 NAT 性能损失的问题。DHCP 网络使容器支持从 DHCP 服务器获取网络配置。VLAN 网络为多租户的云环境提供了网络二层隔离。QoS 模块实现了容器带宽的限制。服务发现模块实现了 Docker 容器间的服务发现机制。macvlan daemon 则可以处理原生的 Docker Remote API，使得整个系统能够和 Docker daemon 无缝整合。

本文所实现的基于 macvlan 的 Docker 网络系统不仅弥补了 Docker 本身网络的各种功能缺陷，在数据传输效率上也比 Docker 的原生网络系统及基于 OVS 的 overlay 网络系统有提升。

关键词： Docker，网络虚拟化，macvlan

Abstract

Docker as the most popular container technology, has set off a wave of Docker boom in the field of cloud computing. Docker container has many advantages compared to the virtual machine, such as super high resource utilization, the speed of the start and stop. In addition, the convenient mirror structure and management mechanism is a major advantage of Docker container and features. These characteristics make the application can quickly build, release, upgrade, testing, etc., has become the developers and operators in the hands of the weapon. At the same time, the Docker container further blurring the boundaries between IaaS and PaaS, became the new favorites of each cloud computing vendors.

However, Docker technology is not perfect. Its network is a big short board. First of all, the network mode of single host makes it have very big problem on the scale and the expansion, and the multi-host communication is one of the problems to be solved urgently. In addition, the Docker network only provides four simple network model, the lack of multi tenant network isolation, network QoS, service discovery and other functions, can not meet the production environment, the use of complex scenes.

In view of the above problems, this paper proposes and implements a Docker container network system based on macvlan. The system is mainly composed of macvlan daemon and macvlan function module. macvlan function module area including Flat, DHCP, VLAN three major network modules as well as QoS and service discovery and other functions. The Flat network is used to solve the problem of the loss of performance of the multi-host communication and NAT. DHCP network enables the container to support the DHCP server. VLAN network provides two layers of isolation for multi tenant environment. QoS module to achieve the limit of the container bandwidth. Service discovery module implements the service discovery mechanism between Docker containers. macvlan daemon can handle the native Remote API Docker, so that the entire system can be seamlessly integrated with Docker daemon.

The Docker network system based macvlan realized in this paper not only makes up for the various functional defects of the Docker itself, but also has a greater improvement in the transmission efficiency than the native network of Docker. At the same time, compared to the general use of the industry based on vSwitch overlay network system, the system has advantages in software dependence, ease of use and efficiency.

Keywords: Docker,Network virtualization,macvlan

目录

图目录

表目录

第1章 绪论

1.1 研究背景和意义

在“互联网+”时代的大背景下，传统行业纷纷借助互联网的平台进行产业升级，这极大地方便了人们的日常生活，提高了办事效率。同时，这也带来了互联网业务的大规模增长。突发式的访问量增长基本成为互联网公司的常态，大家现在很多耳熟能详的应用都几乎在一夜间火爆起来。如一款叫足记的 APP[1]，从最初的不愠不火，到后来一个月就获得 100 万用户的快速增长，甚至一度出现服务器无法访问的情况。面对突然增长的访问量，传统的应对方式需要手动增加服务器、配置运行环境、部署应用等。然而这么做不仅繁琐、效率底下，而且服务器的利用率也会相当的低。如为了应付“双 11”和各种秒杀活动，增加的服务器，在活动结束后，被闲置起来，造成资源的浪费以及成本的提高。于是，人们迫切地需要应用能在资源层和应用层两方面都做到弹性扩展。而云计算正好满足了这样一种需求。

1.1.1 云计算背景

在传统的 IT 运营模式下，企业需要自己花费金钱、人力来构建自己的机房，搭建平台环境，并进行维护，在一些情况下，还需要额外增加机器来应对业务的突发增长。在这种背景下，云计算应运而生。云计算的产生，使得用户不必再自建基础设施，而是可以向云计算提供商购买，就如同购买水电一样方便。用户不但可以更加专注于自己的业务，而且可以按需获取服务，以应对突发增长的访问需求。云计算所涉及的关键技术主要有：虚拟化、分布式计算、并行计算、网络存储、海量数据管理等。对于云计算的定义，有很多种说法，其中比较全面、系统的是美国国家技术与标准局（NIST）给出的定义。云计算是对基于网络的、可配置的共享计算资源池能够方便的、随需访问的一种模式，这些可配置的共享资源计算池包括网络、服务器、存储、应用和服务，并且这些资源池以最小化的管

理或者通过与服务器提供商的交互可以快速地提供和释放，这样的云模式提升了可用性并且具有 5 个基本特征和 3 种交付模式以及 4 种部署模式[2]。如图 1.1 所示。

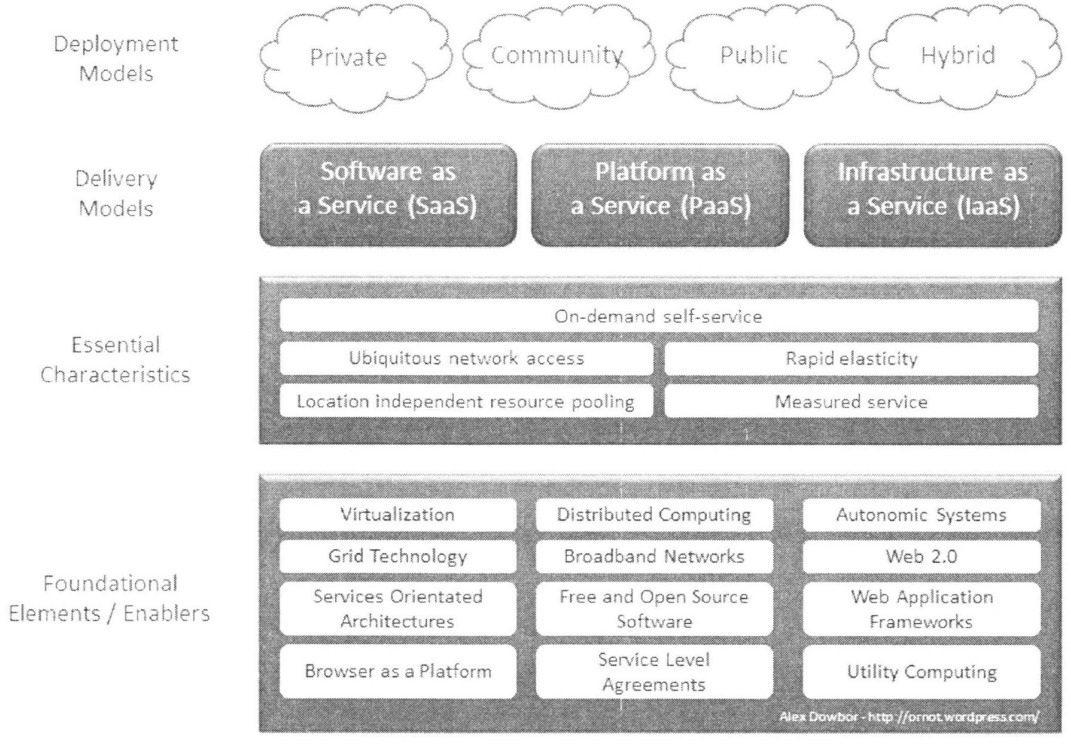

图 1.1 NIST 的云计算架构图

从定义中可以看到，云计算并不是一种新的技术，而是将现有的技术整合，提出的一种新的 IT 服务提供模式。云计算一般分为三个层面： 最下面是 IaaS(Infrastructure as a Service)、中间是 PaaS（Platform as a Service)、再上面是 SaaS（Software as a Service）[3],如图 1.2 所示。

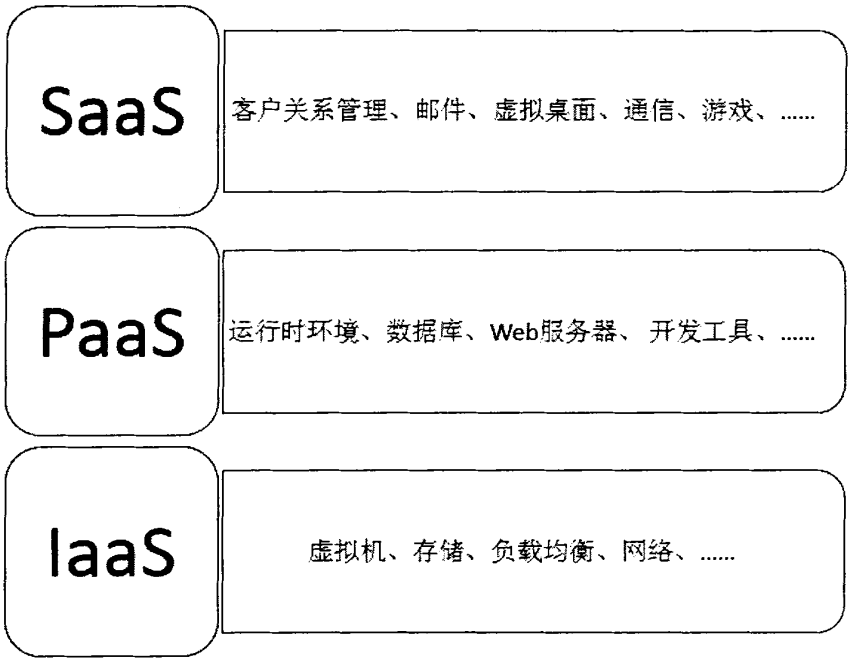

图 1.2 云计算的三层架构

（1） IaaS：基础设施即服务。为消费者提供计算机基础设施服务，包括CPU、内存、网络、磁盘等计算资源。最常见的IaaS服务即虚拟机服务，用户可以自定义虚拟机的规格和数目，可在其上安装任意操作系统及应用软件，并可以根据自己的需求申请网络带宽和磁盘空间，而不用关心这些基础设施的具体维护。相应的IaaS开源软件有OpenStack以及轻量级虚拟化的Docker容器等。

（2） PaaS：平台即服务。把应用运行平台作为一种服务的商业模式。PaaS层关心的是应用的运行环境，即语言环境和各种中间件服务，如JVM、tomcat、数据库等。目前，业界比较出名的PaaS平台有Cloud Foundry、Google App Engine等。

（3） SaaS：软件即服务。一般情况下，用户是在本地运行软件应用。而SaaS则将软件运行在云端，即远程服务器上，用户只需通过互联网获取运算及显示结果即可，而不用关心软件的安装维护及所需的硬件资源。

这就是SaaS所提供的服务模式。常见的应用有Google Map、Google Doc等。

1.1.2 Docker的产生与PaaS的发展

2013年初，Docker 0.1发布，立即在云计算领域引起了高度关注，社区异常活跃，成为年度十大开源软件之一，随后，业界巨头如微软、谷歌、IBM等纷纷表示与Docker建立合作，并在各自相应平台上支持Docker。Docker的发展势头相当迅猛。Docker是Docker.Inc公司开源的一个基于namespace和Cgroup技术之上的Container容器引擎，源代码托管在GitHub上，使用Go语言开发并遵从Apache2.0协议开源[4]。根据官方的定义，Docker是一个提供应用构建、运行、分发的开放平台，它为编程者、开发团队和运维工程师提供了一个他们所需要的共同的工具，来更好的发挥现代应用的分布式特性和网络特性。简单地讲，Docker就是一个应用程序运行容器，可以将应用以及所有依赖打包进一个标准的单元中，如同集装箱一样。Docker的功能类似虚拟机，但是要比虚拟机启停更快、更加节省计算机的资源。Docker具有以下几个特点和优点：

（1） Build[5]：Docker允许你从微服务来组合你的应用，而不用担心生产环境和开发环境的不一致。且独立于任何语言和平台。

（2） Ship[5]：Docker允许你设计应用开发、测试、发布的整个闭环，并且使用一个统一的用户接口管理它。

（3） Run[5]：Docker给你提供在任意平台上安全、可靠地部署可伸缩服务的能力。

（4） 快速构建环境：使用Docker可以使开发者快速构建自己的环境，能避免令人头疼的依赖冲突、环境不一致等等问题。

（5） 应用易迁移：将你的应用、依赖、配置一起打包进容器中，可确保你的应用在任何基础设施的任何环境上都工作地很好，就像在你本地运行一样。

（6） 动态更新、改变、扩展应用：对容器中的应用进行修改、更新、扩展都非常的方便，并且不会影响临近容器中的应用。容器启停很快，且占用的资

源相当小。

Docker 容器在云计算的三层架构中，既可以属于 IaaS，又可以属于 PaaS。说它属于 IaaS，是因为它是虚拟化技术的一种，可以提供类似虚拟机的功能。但是其设计理念更接近 PaaS，因为 Docker 容器以应用为中心，可以使用很少的资源将应用封装并隔离起来，一个容器对应一个应用，可以方便的调度、运行、迁移等。因此，Docker 等容器技术对 PaaS 产生了很大的影响，如传统的 PaaS 平台 Cloud Foundry 在其新一代的 DEA（Droplet Execution Agent）中加入了对 Docker 的支持。还有很多以 Docker 为基础做的微 PaaS 平台，如 Flynn 和 Deis。在这些新一代的 PaaS 中，应用都是以容器为载体。可见以 Docker 为主的容器类工具已经成为 PaaS 中应用调度、扩展、分配的基本单位[6]。

1.1.3 Docker 网络的现状

虚拟化技术是云计算的主要推动技术，而相较于服务器虚拟化及存储虚拟化的不断突破和成熟，网络虚拟化似乎有些跟不上节奏，成为目前云计算发展的一大瓶颈。Docker 作为云计算领域的一颗耀眼新星，彻底释放了轻量级虚拟化的威力，使得计算资源的利用率提升到了一个新的层次，大有取代虚拟机的趋势。同时，Docker 借助强大的镜像技术，让应用的分发、部署与管理变得异常便捷。那么，Docker 是否真的是银弹？能够一劳永逸，解决所有的问题。其网络功能是否可以满足各种场景的需求？但是，事实并非如此，Docker 目前的网络功能还是极其简陋的。

Docker 使用 Linux network namespace 作为网络实现的基础。network namespace 主要提供了关于网络资源的隔离，包括网络设备、IPv4 和 IPv6 协议栈、IP 路由表、防火墙等所有网络资源。使用了 network namespace，一个 Docker 容器就与宿主机的网络以及其他的 Docker 容器隔离了开来[7]。这样，当容器中的应用需要与外界通信时，就得依靠网络虚拟化。Docker 采用的是 Linux 网桥和 veth pair 的网络技术栈。Linux 网桥是一个虚拟的交换机设备，可以为虚拟机或容器提供网络访问[8]，功能和物理的二层交换机等同。veth pair 为虚拟网络

设备对，有两端，类似管道，如果数据从一端传入另一端也能接收到，反之亦然[9]。veth pair 既当作容器中的虚拟网卡存在，也作为容器连接其他容器和网桥设备的桥梁。在以上技术背景的基础上，Docker 实现了 4 种网络模式供用户选择[10]：

（1） host 模式：默认容器都会有一个独立的网络栈，但如果启动容器的时候使用 host 模式，那么这个容器将不会获得一个独立的 network namespace，而是和宿主机共用一个 network namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是共享宿主机的网络资源，如网卡、IP、端口等。但是，容器其他方面，如文件系统、进程列表等还是和宿主机隔离的。host 模式很好地解决了容器与外界通信的地址转换问题，可以直接使用宿主机的 IP 进行通信。但是也降低了隔离性，同时还会引起网络资源的竞争与冲突。

（2） container 模式：container 模式与 host 模式类似，指定新创建的容器和已经存在的某个容器共享同一个 network namespace。都是共享 network namespace，区别就在于 host 模式与宿主机共享，而 container 模式与某个存在的容器共享。新创建的容器不会创建自己的网卡，也不配置 IP，而是与一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。在这种模式下，两个容器的进程可以通过 lo 回环网卡设备通信，增加了容器间通信的便利性和效率。container 模式的应用场景就在于可以将一个应用的多个组件放在不同的容器中，这些容器配成 container 模式的网络，这样它们就可以作为一个整体对外提供服务。同样，这种模式也降低了容器间的隔离性。

（3） none 模式：这种模式下，Docker 容器拥有自己的 network namespace，但是，并不为 Docker 容器进行任何网络配置。也就是说，这个 Docker 容器除了 network namespace 自带的 loopback 网卡外没有其他任何网卡、IP、路由等信息，需要用户为 Docker 容器添加网卡、配置 IP 等。这种模

式如果不进行特定的配置是无法正常使用的，但是优点也非常明显，它给了用户最大的自由度来自定义容器的网络环境。

（4） bridge 模式：Docker 默认的网络模式，也是最主要的网络模式。这种模式下，容器有自己的 network namespace，且所有宿主机上的 Docker 容器都通过 veth pair 连接到 Linux 网桥上。这种模式的网络拓扑图如图 1.3 所示。图中 docker0 网桥上的 veth 网卡设备相当于交换机上的端口，可以将多个容器或虚拟机连接在其上，docker0 网桥就为连在其上的容器转发数据帧，使得同一台宿主机上的 Docker 容器之间可以相互通信。同时，Docker daemon 会从 RFC1918 所定义的私有 IP 网段中，选择一个和宿主机不同的 IP 地址和子网分配给 docker0[111]，连接到 docker0 的容器就从这个子网中选择一个未占用的 IP 使用，docker0 的 IP 充当这些容器的默认网关。因此，Docker 容器的网段和主机不在一个平面上，Docker 容器通过 Linux 宿主机的 IP_Forward 功能与外界通信。

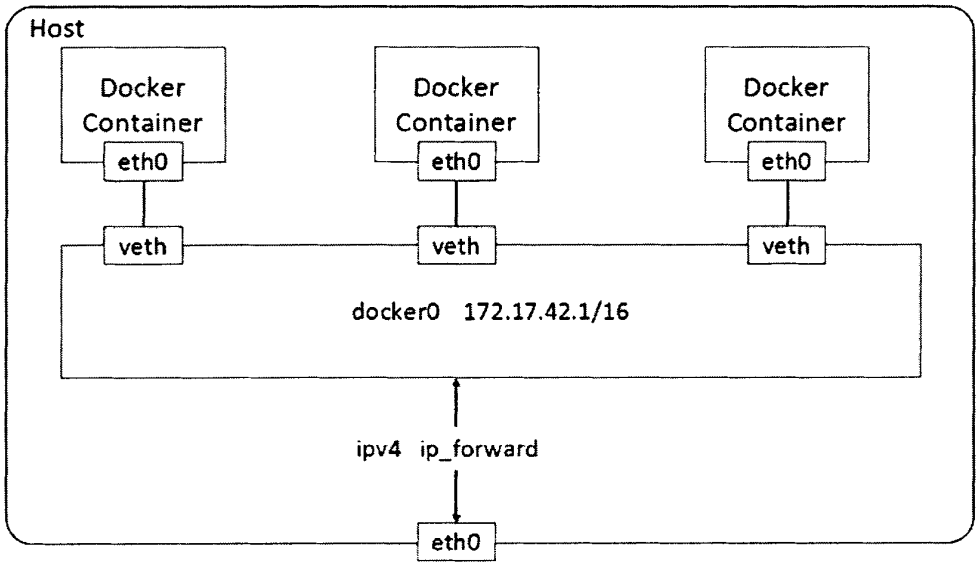

图 1.3 Docker 的 bridge 网络模式

1.1.4 Docker 网络增强的意义

从上一节中可以看到，Docker 的网络并不完美，Docker 目前的网络模型存在性能低下和不易扩展等问题，且不支持复杂的设置，具体体现在以下几点：

（1） 容器间不能跨主机进行通信。Docker 目前的网络模型属于单机型，每一台宿主机上的 Docker 容器都是一个独立的私有子网，子网内可以相互通信，不同子网间容器的通信则需要进行映射到主机的端口上进行。这种模型在单机下还可行，但用来构建 Docker 服务器集群甚至用来构建公有云时，这样的网络是远远满足不了需求的。且现在微服务架构流行，一个应用有多个服务组成，服务之间的通信也需要跨主机通信来保证[12]。

（2） 不支持 VLAN（virtual Local Area Network）隔离。VLAN 隔离在现在企业中已经广泛应用，其二层隔离的特性不仅带来了安全上的便利，也有效的隔离了广播域，提高了网络带宽的利用率。使用 Docker 来构建私有云和公有云的时候，多租户是必须考虑的一个问题，而 VLAN 隔离则是支持多租户的一个必要前提。在云环境中，将虚拟机或容器放在 VLAN 中，而不是直接暴露在互联网上，可以避免很多安全问题[13]。

（3） IP 地址管理不完善。Docker 容器只能从一个私有网段中随机获取一个 IP，不支持用户指定 IP 的方式，也不支持 DHCP，这就给用户使用上带来了很多限制。还有一个问题是，当同一个 Docker 容器重启后，容器的 IP 会改变，这有时候也会成为一个问题。

（4） 没有网络 QoS 支持。网络 QoS 在企业和公有云环境下是很重要的一个功能[14]。目前 Docker 并不支持，因此，当使用 Docker 来构建云服务时，QoS 是一个不得不解决的问题。

这些问题在容器像虚拟机一样在企业环境中大规模部署时，或使用容器部署分布式应用时，都会浮现出来，成为拦路虎。因此，解决以上问题成为 Docker 容器商用化的必经之路，只有把 Docker 网络问题真正解决好了，Docker 才能发挥出它最大的价值。

1.2 研究内容和目标

上一节提出了 Docker 网络存在的诸多不足，这些问题在不同的使用场景下有不同的表现形似，需要根据各自的需求，来提出最合理的解决方案。本文根据实验室实际使用过程中遇到的问题，提出以 Linux macvlan 虚拟化设备为基础，使用 IEEE 802.1Q 、DHCP、Traffic Control[15]等技术实现的一套 Docker 网络系统。主要按照以下几个目标进行设计实现：

（1） Flat 网络模块。Flat 网络使得 Docker 容器接入到宿主机所在的二层网络中，这样所有的容器和宿主机在同一个平面上，方便容器和宿主机的通信，也可以完成容器和容器的跨主机通信功能。同时，在一些容器和虚拟机混合使用的场景中，这样做也可以对容器和虚拟机进行统一的调度和分配。

（2） VLAN 网络模块。Flat 网络是将所有容器连成一片，即放在同一个二层中。而 VLAN 网络则相当于对容器进行分组，即进行 VLAN 的划分。VLAN 的划分可以按照业务类型、用户等进行划分，而不用关心容器在哪一台机器上，即 VLAN 的划分也是可以跨宿主机的。VLAN 模块满足了隔离和安全的需求。

（3） 完善的 IP 地址管理方案。Docker 本身的 IP 地址分配方案是单机的，且不支持 DHCP 服务。本系统实现了一套分布式的 IP 地址管理方案，并且支持本地 DHCP 服务器。并使用分布式键值存储系统 etcd 存储 Docker 容器和 IP 地址的对应关系，以解决 Docker 容器 IP 地址重启后不一致的问题，同时可以用在本地 DNS 系统中，作为服务发现使用。

（4） 网络 QoS 支持。使用 Traffic Control 技术实现了对容器的 QoS 支持。

（5） 支持容器间服务发现。Docker 容器使得微服务架构流行，而容器间的相互联系和协作需要有服务发现机制的支撑[16]。

（6） 方便集成。目前，基于 Docker 容器的资源调度平台有很多，Docker 容器不可能单独使用。因此，Docker 的网络系统要做到平台无关性，能够

与任何第三方调度平台集成使用。

1.3 文章组织结构

第一章主要介绍了本文的研究背景、研究内容和目标。并分析了 Docker 网络的现状及不足，提出了 Docker 网络增强的意义。

第二章为本文所用到的主要技术的概述，包括 Docker 的底层架构，所用到的 Linux 内核的知识，以及 macvlan 设备。同时分析了目前业界普遍使用的基于 Open vSwitch（OVS）的 overlay 网络方案。

第三章为需求分析与系统整体设计，根据实验室提出的需求，提出使用基于 macvlan 的解决方案，并对基于 macvlan 的网络系统做了一个整体的介绍。

第四章为macvlan功能模块区的具体设计和实现部分，包括Flat、DHCP、VLAN、QoS、容器服务发现等模块。

第五章描述了 macvlan daemon 的设计与实现，macvlan daemon 为自己实现的一个 HTTP 代理，可以处理原生的 Docker HTTP 请求，实现与 Docker daemon 的完美对接。

第六章对系统的主要功能做了一个简单展示并且做了性能测试，系统基本弥补了 Docker 网络现有的一些不足，并且在数据传输效率上也比 Docker 原生的和基于 OVS 的 overlay 系统有提高。

第七章为本文的总结与展望，主要说明了本文的工作以及还存在的不足及以后的改进方向。

1.4 本章小结

本章主要介绍了本文的研究背景、研究内容和目标。首先介绍了云计算并引出Docker容器，接着分析了Docker的优势以及在网络方面存在的不足。将Docker网络的改进作为本文研究的内容，并使用 macvlan 实现了一套 Docker 网络系统。最后介绍了本文的组织结构。

第2章 相关技术概述

2.1 Docker 核心原理解读

2.1.1 Docker 整体架构

Docker 使用了传统的 client-server 架构模式，总架构图如图 2.1 所示。用户通过 Docker client 与 Docker daemon 建立通信，并将请求发送给后者。而 Docker 的后端是松耦合结构，不同模块各司其职，并有机组合，完成用户的请求。

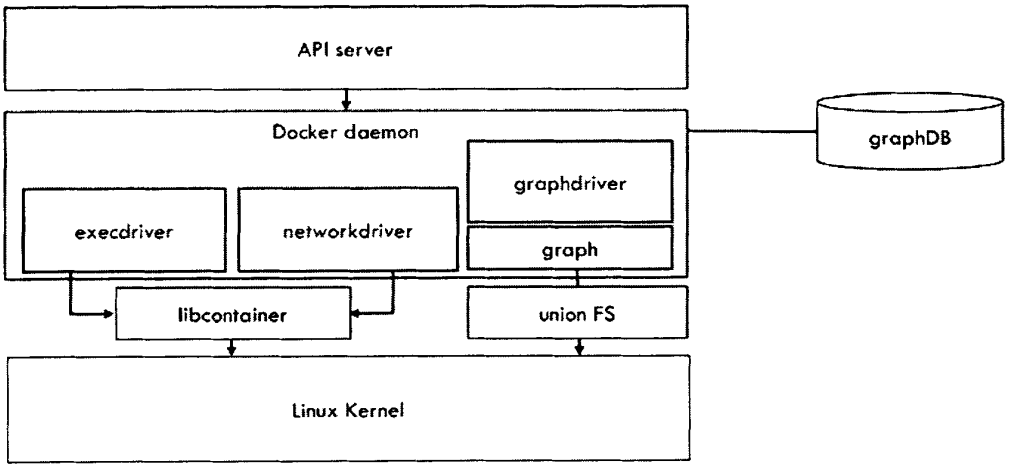

图 2.1 Docker 架构图[17]

Docker daemon 是 Docker 架构中的主要用户接口。首先，它提供了 Docker server 用于接受来自 Docker client 的请求，其后根据不同的请求分发给 daemon 的不同模块执行相应的工作。Docker 通过 driver 模块来实现对 Docker 容器执行环境的定制。当需要创建 Docker 容器时，可从 Docker registry 中下载镜像，并通过镜像管理驱动 graphdriver 将下载的镜像以 graph 的形式存储在本地；当需要为 Docker 容器创建网络环境时，则通过网络管理驱动 networkdriver 创建并配置 Docker 容器网络环境；当需要限制 Docker 容器运行资源或执行用户指令

等操作时，则通过 execdriver 来完成[18]。libcontainer 是一个独立的容器管理包，networkdriver 和 execdriver 都通过 libcontainer 来实现对容器的具体操作，包括利用 UTS、IPC、PID、Network、Mount、User 等 5 个 namespace 子系统实现容器之间的资源隔离和利用 cgroup 实现对容器的资源限制。当运行容器的命令执行完毕后，一个实际的容器就处于运行状态，该容器拥有独立的文件系统，安全且相互隔离的运行环境。

2.1.2 Docker 各组件介绍

Docker daemon

Docker daemon 是 Docker 最核心的后台进程，它负责响应来自客户端的请求，然后将这些请求翻译成系统调用完成容器操作。该进程会在后台启动了一个 API server，负责接收由 Docker client 发送的 API 请求；接收到的请求将通过 Docker daemon 内部的一个路由分发调度，再由具体的函数来执行请求。

Docker client

Docker client 是一个泛称，用来向指定的 Docker daemon 发起请求，执行相应的容器操作。它既可以是 docker 命令行，也可以是任何遵循了 Docker API 的远程客户端。

graph

graph 组件负责维护已下载的镜像信息及它们之间的关系，所以大部分 Docker 镜像相关的操作都会由 graph 组件来完成。graph 通过镜像“层”和每层的元数据来记录这些镜像的信息，用户发起的镜像操作最终都是转换成了 graph 对这些层和元数据的操作，但是正是由于这个原因，很多时候 Docker 操作都需要加载当前 Docker daemon 维护着的所有镜像信息，这时 graph 组件往往会成为性能瓶颈。

driver

前面提到，Docker daemon 是负责将用户请求转译成系统调用，进而创建、管理容器的核心进程。而在具体实现过程中，为了将这些系统调用抽象成为统一

的操作接口方便调用者使用，Docker 把这些操作分类成容器管理驱动、网络管理驱动、文件存储驱动 3 种，分别对应 execdriver、networkdriver 和 graphdriver。

（1） execdriver 是对 Linux 操作系统的 namespaces、cgroups、apparmor、SELinux 等容器运行所需的系统操作进行的一层二次封装，其本质作用类似于 LXC，但是功能要更全面。这也就是为什么 LXC 会作为 execdriver 的一种实现而存在。当然，execdriver 最主要的实现也是现在的默认实现是 Docker 官方编写的 libcontainer 库。

（2） networkdriver 是对容器网络环境操作所进行的封装。对于容器来说，网络设备的配置相对比较独立，并且应该允许用户进行更多的配置，所以在 Docker 中，这一部分是单独作为一个 driver 来设计和实现的。这些操作具体包括：这些操作具体包括创建容器通信所需的网络，容器的 network namespace，这个网络所需的虚拟网卡，分配通信所需的 IP，服务访问的端口和容器与宿主机之间的端口映射，设置 hosts、resolv.conf、iptables 等。

（3） graphdriver 是所有与容器镜像相关操作的最终执行者。graphdriver 会在 Docker 工作目录下维护一组与镜像层对应的目录，并记下容器和镜像之间关系等元数据。这样，用户对镜像的操作最终会被映射成对这些目录文件以及元数据的增删改查，从而屏蔽掉不同文件存储实现对于上层调用者的影响。目前 Docker 已经支持的文件存储实现包括 aufs、btrfs、devicemapper、overlay、vfs。

2.1.3 namespace 资源隔离

Docker 之类的容器技术都是使用了 Linux 的 namespace 和 cgroups 技术，namespace 技术实现资源隔离，就如同沙盒一样，可以将容器与宿主机和其他容器隔离开来。Linux 内核中提供了 6 种资源隔离技术，如表 2.1 所示。

表 2.1 六种 namespace 隔离技术

namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、信息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

下面对这 6 个 namespaces 进行简单介绍：

（1） UTS namespace ：UTS（UNIX Time-sharing System）namespace 提供系统标识的隔离，即主机名和域名的隔离。这些标识可以通过 sethostname() 和 setdomainname()等系统调用设置。在 Docker 容器环境中，就是通过 UTS namespace 给容器以不同的主机名及域名，使得不同的容器可以在网络中被当作一个独立的节点而区分开来。同时，初始化及配置脚本也可以根据这些标识来决定执行哪部分功能。

（2） PID naemsapce：PID（process ID）namespace 提供进程 ID 号的隔离，即在不同的 PID namespace 中，可以有相同 ID 号的进程。在 Docker 容器中，PID namespace 可以使每一个容器有其自己的 init 进程（PID 为 1，每一个新的 PID namespace 从 1 开始编号）。Init 进程为所有进程的父进程，同时可以用来管理系统的初始化工作以及回收孤儿进程。同时，PID namespace 也可以使得容器在不同的主机间迁移，而容器的进程 ID 可以不改变。PID namespace 可以嵌套，即可以组织成树状的机构，最上层的为 root namespace（宿主机的 PID namespace）。一个进程在当前 namespace 和祖先 namespace 中都有一个进程号，其只能看到同一个 PID namespace 中的进程及该进程所嵌套的子 PID namespace 中的进程。

（3） IPC namespace：IPC（Inter-Process Communication，IPC）namespace

提供进程间通信资源的隔离，及 System V IPC 对象以及 POSIX 消息队列。这些 IPC 机制的共同特点是 IPC 对象靠机制作用区分而不是靠文件系统路径。每一个 IPC namespace 有其自己的 System V IPC 标识符集合和 POSIX 消息队列文件系统。IPC 对象的创建对同一 IPC namespace 下的所有进程可见，对其他 IPC namespace 下的进程不可见。同时，/proc/sys/fs/mqueue、/proc/sys/kernel、/proc/sysvipc 在不同的 IPC namespace 下不同。当一个 IPC namespace 结束时，其 IPC 对象也自动销毁。

（4） Mount namespace：Mount namespace 提供文件系统挂载点集合的隔离，意味着不同 mount namespace 下的进程看到的文件系统结构不同。/proc/[pid]/mounts 文件可以看到当前进程所在 mount namespace 下的文件系统挂载点。/proc/[pid]/mountstats 文件列出了当前进程所在 mount namespace 下的挂载点具体信息（统计信息、配置信息等）。挂载点的信息可以通过 mount()、umount()系统调用设置，在某一 mount namespace 下调用这些系统调用不影响其他 mount namespace 下的挂载点。

（5） Network namespace：Network namespace 提供系统网络资源的隔离，包括网络设备、IPv4 和 Ipv6 协议栈、IP 路由表、防火墙、/proc/net 和 /sys/class/net 目录、端口等。一个物理网络设备只能存在于一个 network namespace 中。两个 Network namespace 之间可以通过 veth pair 设备直接连接起来，也可以接到一个网桥上进行通信。

（6） User namespaces：User namespace 提供与系统安全相关的标识和属性的隔离，如用户 IDs、组 IDs、根目录、keys、capabilities 等。一个进程在一个 user namespace 的内外可以有不同的用户 ID 和组 ID。如一个进程在一个 user namespace 中具有 root 权限（用户 ID 为 0），而在该 user namespace 外，它不具有特权。若在容器技术中使用了 user namespace，那么一个普通用户创建的普通进程可以在容器 user namespace 中具有特权，这样可以方便用户管理自己的容器，提高系统的安全性。

2.2 macvlan 设备介绍

有时可能需要一块物理网卡绑定多个 IP 以及多个 MAC 地址，虽然绑定多个 IP 容易，但是这些 IP 会共享物理网卡的 MAC 地址，可能无法满足设计需求，所以有了 macvlan 设备。macvlan 是 Linux 内核实现的一个虚拟网络设备，macvlan 使得一块物理网卡可以虚拟出多块虚拟网卡，每一个虚拟网卡都有自己的 MAC 地址，当物理网卡收到数据时，会根据 MAC 地址将数据发送到相应的网卡上，这里我们将物理网卡称为父网卡，虚拟出来的网卡叫子网卡。正因为这个特性，macvlan 设备多用在虚拟机或容器的网络虚拟化中，如之前的 LXC 容器就支持配置 macvlan 网络，就是虚拟出 macvlan 设备，再将这个虚拟网卡放到容器的 network namespace 中。如图 2.2 所示。

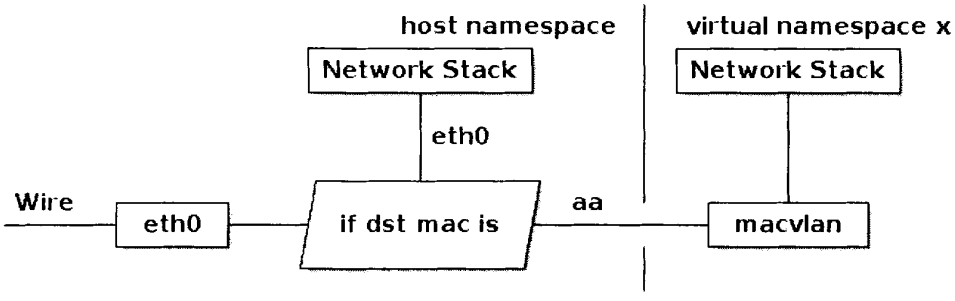

图 2.2 macvlan 使用示例图[19]

2.2.1 macvlan 的 4 种模式

macvlan 设备中有 4 中模式供用户选择，其功能和作用如下：

（1） Bridge 模式：这种模式类似于 Linux Bridge。一个网卡虚拟出来的所有 macvlan 设备（bridge 模式）之间可以直接交换数据，就如同连在同一个交换机上一样，如图 2.3 所示。而且，这种模式数据交换效率要比 Linux Bridge 高，因为它知道它可以接受的所有 MAC 地址，不需要像网桥那样去学习 MAC 地址，也不需要 STP（Spanning Tree Protocol）机制。这种

模式可以用来将容器或虚拟机连接到主机网络中，这样容器和主机就处在同一个二层网络中。

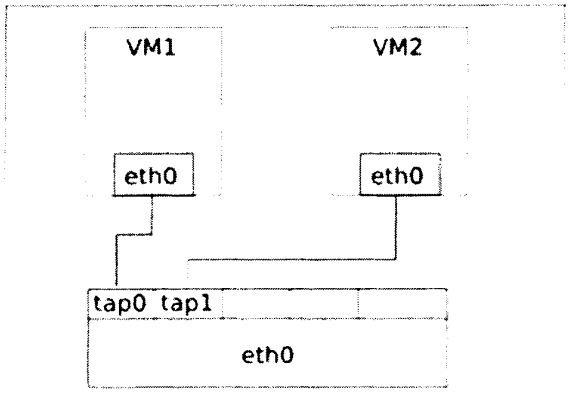

图 2.3 macvlan 的 bridge 模式[21]

（2） VEPA[20]（Virtual Ethernet Port Aggregator）模式：当父网卡从处于这种模式下的子网卡接收到数据时，都会将数据发往外界的上游交换机，尽管目的地址就是同一个父网卡的另一块子网卡。这样，容器内部的流量不再由本地交换机处理，而是发往物理网卡外部，由外部的交换机处理后再发回来，这么做可以用传统的交换机监控、处理虚拟机内部之间的通信，减少物理机的 CPU 消耗。但同时，一般交换机不允许一个物理端口接受数据帧，又同时从同一个端口发出该数据帧，这需要 Hairpin 技术支持。所以这种模式对物理交换机有一定的要求，同时，容器之间的流量通过物理交换机传输，既浪费了网络带宽又增加了数据延迟。该模式如图 2.4 所示。

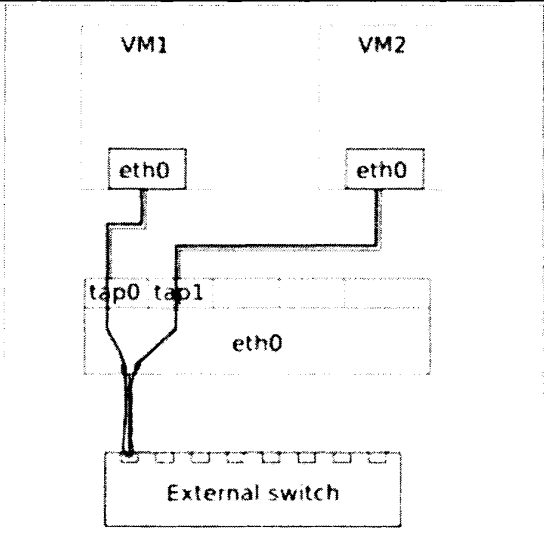

图 2.4 macvlan 的 VEPA 模式[21]

（3） Private 模式： 这种模式是一种隔离模式，处在这种模式下的子设备不能与其他子设备之间通信。

（4） Passthru 模式：一块网卡只允许有一个 macvlan 子设备。

2.3 现有相关 Docker 网络解决方案

Docker 网络问题存在有一段时间了，业界已经存在了一些相关的解决方案，主要有以下几个：

（1） pipework[22]：一个简单小巧而功能强大的网络配置工具，可以根据自己的需求去配置 Docker 容器的网络。缺点是不自动化、配置不持久、没有与 Docker daemon 集成。

（2） socketplane[23]：基于 OVS[24]的 overlay 解决方案，可以解决多主机间容器直接通信的问题。功能比较全，但是 bug 较多，不够稳定，且性能较差。

（3） flannel[25]：overlay 网络的解决方案。可以很好地与 kubernetes 结合，使得 kubernetes 可以部署在除 GCE（Google Compute Engine）以外的其他

IaaS 平台上。缺点是没有 VLAN 划分，不支持多租户。

（4） libnetwork[26]：Docker 官方提供的一个可插拔的网络后端方案。Libnetwork 旨在将 Docker 的网络功能从 Docker 核心代码中分离出去，形成一个单独的库。Libnetwork 通过提供插件的形式为 Docker 提供网络功能，准备实现一套可插拔的 API。使得用户可以根据自己的需求实现自己的 driver，driver 可以用来给容器提供网络功能。但由于 libnetwrok 的 API 尚在开发之中，功能和接口都不够完善，还不能满足我们使用的需求，因此本文叙述的方案并未按照 libnetwork 的 API 来进行开发，但是后续开发中可能会与 libnetwork 实现对接。

可以看到，Docker 网络目前的解决方案虽然很多，但没有哪一种方法可以一劳永逸。但对于 Docker 的网络问题，业界普遍采用基于 OVS 的 overlay 方案，下节会对此方案的利弊进行一个总结。

2.3.1 基于 OVS 的 overlay 网络方案总结

Open vSwitch 是一个开源的虚拟交换机，相比于 Linux Bridge，Open vSwitch 支持 VLAN、QoS 等功能，同时还提供对 OpenFlow 协议的支持，可以很好地与 SDN（Software Defined Network）体系融合[27]。因此，Open vSwitch 可以解决很多 Docker 环境中的复杂网络问题，socketplane 即是此种方案的代表。

2.3.1.1 overlay 技术模型

overlay 网络其实就是隧道技术，即将一种网络协议包装在另一种协议中传输的技术。如果有两个使用 IPv6 的站点之间需要通信，而它们之间的网络使用 IPv4 协议，这时就需要将 IPv6 的数据包装在 IPv4 数据包中进行传输。隧道被广泛用于连接因使用不同网络而被隔离的主机和网络，使用隧道技术搭建的网络就是所谓的 overlay 网络[28]。它能有效地覆盖在基础网络之上，该模型可以很好地解决跨网络 Docker 容器实现二层通信的需求。

当前主要的 overlay 技术有 VXLAN（Virtual Extensible LAN）和 NVGRE（Network Virtualization using Generic Routing Encapsulation）。VXLAN 是将以太

网报文封装在 UDP 传输层上的一种隧道转发模式，它采用 24 位比特标识二层网络分段，称为 VNI（VXLAN Network Identifier），类似于 VLAN ID 的作用[29]。NVGRE 同 VXLAN 类似，它使用 GRE 的方法来打通二层与三层之间的通路，采用 24 位比特的 GRE key 来作为网络标识（TNI）[30]。

2.3.1.2 GRE 实现 Docker 容器跨主机通信

GRE 协议可以用来封装任何其他网络层的协议，目前比较普遍的方法是结合 Open vSwitch 使用。如图 3.2 所示，容器 con1 可以和 con2 直接通信，这里 GRE 协议封装的是二层帧，解决了容器间跨主机通信的问题。

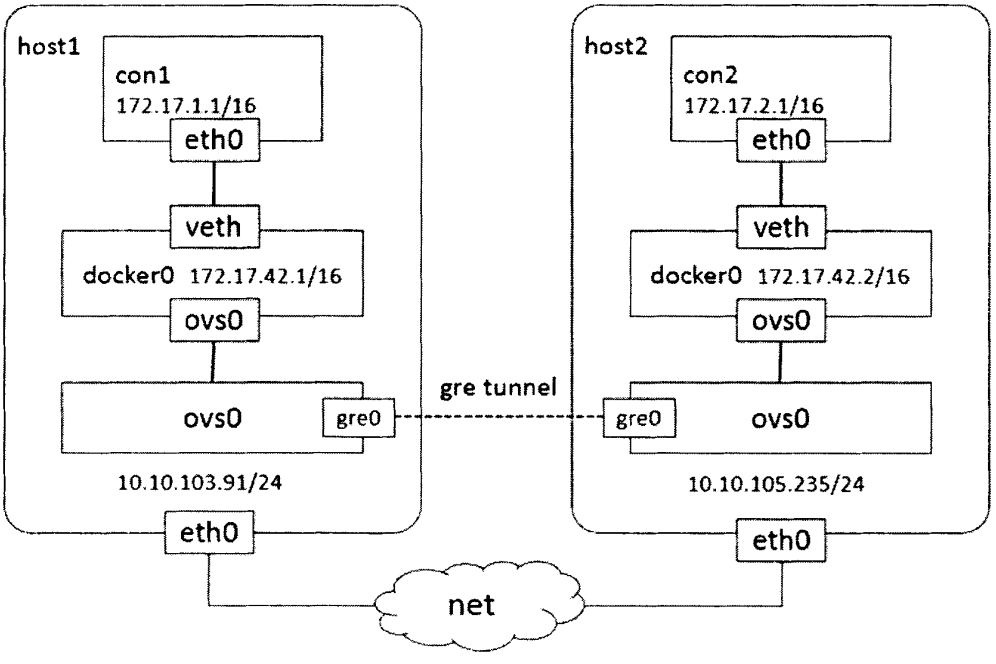

图 2.5 GRE 实现 Docker 容器跨主机通信

2.3.1.3 基于 OVS 的 overlay 网络方案的不足

基于 OVS 强大的功能，可以很方便的构建一套解决方案。但是，这种方案并不完美。当花费了大量时间学习研究了 OVS 后，发现 OVS 也并非银弹。使用 OVS 主要有以下几个问题：

（1） 首先，OVS 虽然很强大，但是毕竟是一个第三方依赖。同时，其本身也比较重，会使得问题变得更加复杂，使用 Docker 的开发人员，未必喜欢如此复杂的工具。而且，在某些常用环境下 OVS 会崩溃。

（2） 性能问题。OVS 本身的 internal 接口可以作为容器的网卡来连接两个容器，其性能相比 veth 设备和 Linux Bridge 都要好。但是涉及到多主机和隧道，情况就不一样了。隧道要对数据包进行封装和解封装，这些开销就会带来性能损失。这也是 overlay 方案被人诟病的一点。

（3） Debug 困难。overlay 的灵活是构建在宿主机之间的隧道上的，带来的一个弊端就是如果出了问题，我们不能够定位是物理链路还是隧道本身的问题。

基于以上几点，本文使用的基于 macvlan 的方案则更加简单有效。

2.4 本章小结

本章首先介绍了 Docker 的整体架构、各个功能组件的工作方式以及所使用的内核 namespace 隔离。接着讲解了 macvlan 虚拟设备的工作方式和使用场景，以及其 4 种不同的模式。最后分析了业界普遍采用的基于 OVS 的 overlay 网络所存在的一些问题。本文所介绍的系统主要是通过使用 macvlan 设备来改进 Docker 容器的网络功能，因此，本章所介绍的技术为本系统的基础。

第3章 基于 macvlan 网络系统需求分析与设计

3.1 需求分析

在第一章中，已经分析过 Docker 网络的现状以及不足。对于一个简单的基本的应用而言，Docker 的网络模型还算不错。然而伴随着云计算和微服务的普及，这是远远不够的，完全不能用在情况复杂的生产环境中，也不满足用其来构建云平台的要求。

Docker 的网络问题是极其复杂的，包括表面的和深层次的[31]。它会涉及到非常多的项目，小到本地开发环境，大到类似 kubernetes 的项目[31]。而且，不同的用户有不同的需求。面对 Docker 的网络问题，没有银弹可言。因此，要根据自己的需求实现适合自己的解决方案。

3.1.1 功能性需求分析

浙大 SEL 实验室对 Docker 有较长时间的研究和实践经验，从最初设想使用 Docker 代替虚拟机来部署 CloudFoundry，到使用 Docker 来构建自己的 PaaS 平台。因此，对 Docker 的网络问题也有较深的体会和理解。对此，实验室根据实际使用情况，对 Docker 网络有以下功能需求：

（1） 将容器配置到主机网络。容器本身是连接在一个虚拟网桥上的，使用由 Docker daemon 分配的一个私有网段。宿主机网络上的机器不能直接访问容器网络，容器需要做 NAT 映射后，才能访问。NAT 映射存在效率低且宿主机端口有限等各种问题。在我们的使用场景中，需要将 Docker 容器加入到宿主机网络的需求，这样既方便原有的宿主机和容器之间的通信，也可以将 Docker 容器作为一个新加入网络的虚拟机或宿主机对待，方便管理。

（2） Docker 容器间跨主机访问。Docker 网络的单主机模式是其一大缺陷。跨主机通信基本是每一个网络增强方案都必须的。

（3） 持久化的网络状态和更灵活的IP地址分配。Docker容器启动时IP是随机分配的，并且同一个容器重启后IP地址会改变。我们需要的不仅仅是IP随机分配，而是可以由用户指定并且可以从本地DHCP服务器上获取IP。同时，要保证容器的IP地址在重启后不变。

（4） VLAN安全隔离。多租户下的安全隔离是一个非常重要的需求，要将不同用户、不同业务下的Docker容器从网络二层上进行隔离。

（5） 支持服务发现机制。微服务随着Docker的流行而流行起来，服务之间需要了解对方是否已经准备就绪并且需要知道其在网络中的位置。服务发现机制在Docker的使用场景中已是一个非常普遍的需求。

（6） 网络QoS。Docker已经为容器的资源限制做了很多工作，但是在网络带宽方面却没有进行限制，这就可能带来一些问题。同一台宿主机上的Docker容器共享宿主机的资源，当某些容器占用了主机的大量带宽时，会影响到其他容器所提供的服务。因此，必须对容器进行网络QoS管理。

3.1.2 非功能性需求分析

除了功能性需求之外，还有以下几个重要的非功能性需求需要满足：

（1） 网络功能可插拔。根据libnetwork的设计理念，可插拔的设计方案是比较合理的。可插拔意味着可以很方便的与Docker daemon相结合，同时又不会对Docker daemon原来的代码和功能有什么影响。

（2） 易于与容器调度平台集成。当大规模使用Docker容器时，容器集群调度是必要的，如现在较常用的kubernetes、mesos和swarm。但是，无论使用哪种调度工具，都要非常方便的能与所实现的网络方案对接。达到只要在使用Docker client的地方，不需要什么配置，就能无障碍的使用所实现的方案，做到真正的开箱即用。

（3） 尽可能的去除第三方依赖。依赖问题的管理通常令人头疼，过多的依赖常带来各种冲突和安装上的麻烦。Docker官方也在尽可能的去除依赖第三方库，如使用自己开发的项目libcontainer来摆脱对LXC的依赖。因此，

所实现的网络系统，也应尽可能的减少第三方依赖。

（4） 广泛的平台适用性。能够适应各种常见的 Linux 发行版，如 Ubuntu、Debian、RedHat、CentOS 等。做到在各个系统上无差别的使用。

（5） 高效的网络传输效率。云环境中，容器数量数以千万，容器之间需要紧密合作。因此，高效的网络传输至关重要。

（6） 易于维护。有健全的日志记录系统，能方便快速的定位到引起错误的地方。

3.2 基于 macvlan 网络系统的整体设计

基于以上需求，并在比较了基于 Open vSwitch 的相关方案后，本文选择使用 macvlan 设备进行 Docker 网络的设计与实现。macvlan 以及 VLAN 设备本身 Linux 内核支持，不需要额外的依赖。且 macvlan 设备使用方便效率也很高，已经足够满足系统的使用需求。

3.2.1 系统整体架构

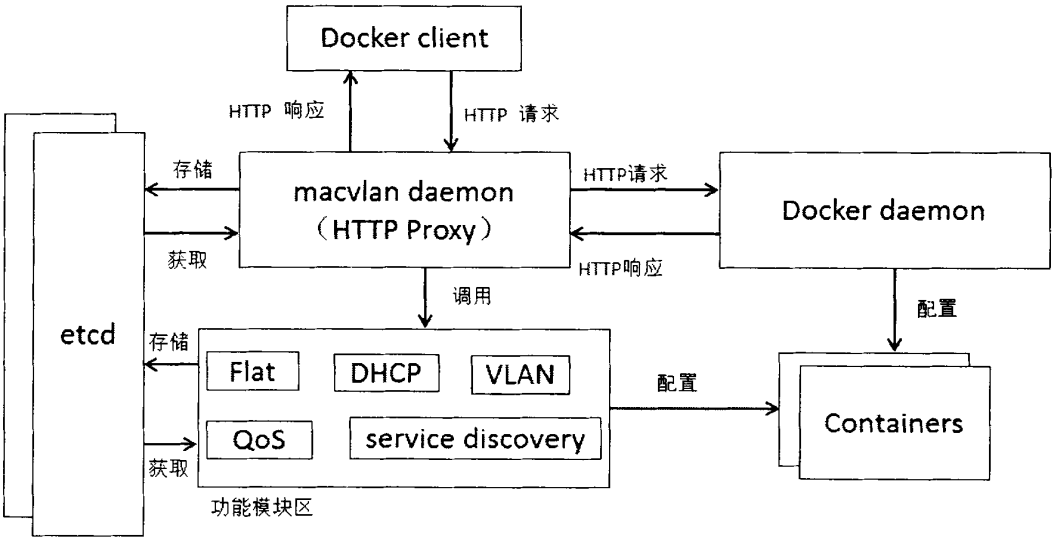

图 3.1 系统整体架构图

为了方便系统的调试、维护和升级，系统遵从模块化的设计原理，其整体架构图如图3.1所示。系统主要由两部分构成。上半部分的HTTP请求链以macvlan daemon为中心，还包括Docker client、Docker daemon，为c/s架构，主要用以响应Docker用户所发送的HTTP请求。下半部分以macvlan功能模块区为中心，用以为容器配置特定功能的网络，其中的每一个模块都既可以独立作为一个命令使用，又可以结合Docker client，被macvlan daemon调用，在HTTP请求链中完成容器的配置。

3.2.2 系统各模块简介

从图3.1中可以看到，本系统主要由以下几个模块构成：

（1） Docker client/daemon：原本Docker架构中的组件，本系统未做任何修改。

（2） macvlan daemon：一个伪装成Docker daemon的后台进程。在原本的Docker架构中，Docker client向Docker daemon发送HTTP请求，Docker daemon调用相应的后台模块来完成请求。而在本系统中，Docker client的请求先发到macvlan daemon上，再由macvlan daemon发往Docker daemon。macvlan daemon在此作为一个HTTP代理存在，根据收到的HTTP请求和回复，调用macvlan功能模块配置相应的容器网络。macvlan daemon可以监听一个tcp端口或者Unix域套接字文件。

（3） Flat模块：Flat模块即用来将容器连接到本地主机网络中的。这么做一来可以方便本地主机和容器之间的通信；二来可以实现容器跨主机通信；三来可以方便容器和宿主机的统一管理。

（4） DHCP模块：DHCP模块使得容器可以使用本地DHCP服务。通过从本地DHCP获取网络配置，方便容器和本地网络的对接，也方便容器网络的管理。

（5） VLAN模块：使得不同业务、租户的容器实现二层隔离。通过Linux内核本身的VLAN设备实现VLAN隔离。在每一个VLAN内部，类似于

一个 Flat 网络，同一个租户内的容器也可以实现跨主机通信。

（6） QoS 模块：完成容器网络流量、带宽控制，保证服务质量。

（7） service discovery 模块：服务发现模块，使得服务之间可以相互知道对方在网络中的位置。

（8） etcd 集群：系统的数据存储中心，主要为容器保存网络状态信息。

3.2.3 系统工作流

上节介绍了系统的整体架构和各个模块的功能，下面从一个 Flat 网络容器的创建出发，通过时序图来描述系统各个组件之间的交互以及工作流。

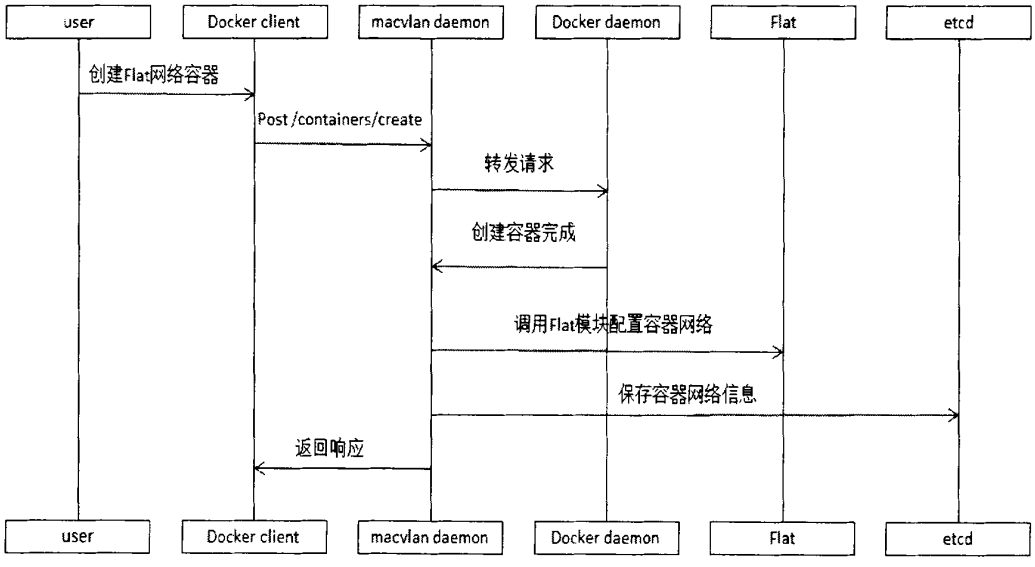

图 3.2 创建 Flat 网络容器的工作流

图 3.2 为创建一个 Flat 网络容器的工作流，其工作流程如下：

（1） 用户通过 Docker client 发送创建容器的 HTTP 请求给 macvlan daemon。

（2） macvlan daemon 接受请求后转发给 Docker daemon，Docker daemon 做真正的创建容器工作。

（3） 容器创建完成后，macvlan daemon 根据请求中关于网络的参数信息，去调用相应地模块完成容器网络配置工作，如本例中调用 Flat 网络模块完成容器网络的配置。

（4） 配置完成之后，macvlan daemon 将容器的网络配置信息保存到 etcd 集群中。

（5） 最后 macvlan daemon 将结果反馈给 Docker client。

以上即为创建 Flat 网络容器的工作流，为容器配置其他网络（DHCP、VLAN 等）流程与其类似，只需 macvlan daemon 调用相关模块完成配置即可。

3.3 本章小结

本章首先根据实验室的具体使用场景提出系统需求分析。接着根据需求分析提出了基于 macvlan 的 Docker 容器网络系统。最后就 macvlan 方案的整体架构做了分析和设计。

第4章 macvlan 功能模块区的设计与实现

macvlan 功能模块区主要用来为容器配置具体的网络功能，主要包括 Flat、DHCP、VLAN 三大网络模块以及网络 QoS 和容器服务发现等功能。

4.1 Flat 网络模块的设计与实现

4.1.1 Flat 网络模块设计

Flat 网络就是将容器的网络和宿主机的网络放在同一个二层网络中，其最主要的目的是方便容器和宿主机的通信以及容器与容器之间的跨主机通信，避免原 Docker 系统 NAT 方式所带来的性能损失。

4.1.1.1 规避 NAT

在 Docker 的 bridge 模式中，Docker daemon 创建了一个 Linux Bridge（默认为 docker0）和一些 veth pair 设备。veth pair 的一端作为网卡放在容器的 network namespace 中，另一端作为端口连接在 docker0 网桥上。docker0 网桥上连接的容器属于一个独立的私有二层网络，其外出的流量需要首先路由到宿主机的网卡上，然后再外出。并且，在宿主机上设有 MASQUERADE 的 IPtables 规则，此规则的作用是，当容器所发出的数据包从宿主机网卡出去时，会将其源 IP 地址改为宿主机网卡的 IP 地址，也就是 SNAT。这样，容器网络对外是不可见的，当外界想要访问容器服务时，容器必须首先在主机上做好映射，即将自己的服务端口映射到主机的端口上，当主机相应端口收到访问数据时，再修改目的 IP 地址和端口，将数据转发到相应容器上，这就是 DNAT。NAT 的方式一来使得外界访问服务变得不方便；二来会占用主机端口，而端口资源是有限的，最终可能会造成冲突；最重要的一点，是其会修改 3 层数据包的头部，不仅会带来流量统计和分析的不便，更会造成数据传输性能的严重损失。因此，我们需要用其他技术来规避 NAT。

第一种方法是使用路由。通过将主机变为一个路由器，使得容器中的数据包

不再进行NAT，而是使用路由的方式与外界进行数据交换。但是，这个方案实际实施起来却并不容易，需要对每一个宿主机上容器的网段做好规划，同时在相应宿主机上添加大量的路由表。而在多个宿主机和多租户情况下，用户其实并不关心容器会启动在哪一台宿主机上，同一个租户的同一个网段的容器可能散落在不同的宿主机上，这种情况下，配置路由规则将会是一场恶梦。

第二种方法便是本文提到的Flat网络。即将容器直接连接到宿主机网络上，它们从本地的DHCP服务器获取IP地址或者使用静态分配的地址，所有4层网络端口完全暴露。这种方法也有一些缺点，例如会造成本地终端数量及MAC地址激增，而ToR交换机可支持的MAC地址数量不同，可能会造成洪范所有帧，造成本地网络吞吐能力的下降。但是其简单快速的配置及高效率的连接主机和容器是选择此方案的重要原因。

4.1.1.2 网桥桥接的Flat网络拓扑

如果想要使Docker容器和容器主机处于同一个二层网络中，最简单的方法就是把两台机器连在同一个交换机上，或者连在不同的但相互级联的交换机上。在虚拟场景下，虚拟网桥可以将容器连在一个二层网络中，那么只要再将主机的网卡桥接到虚拟网桥中，就能将容器和主机的网络连接起来。构建完拓扑结构后，只需再给Docker容器分配一个本地局域网IP即可。

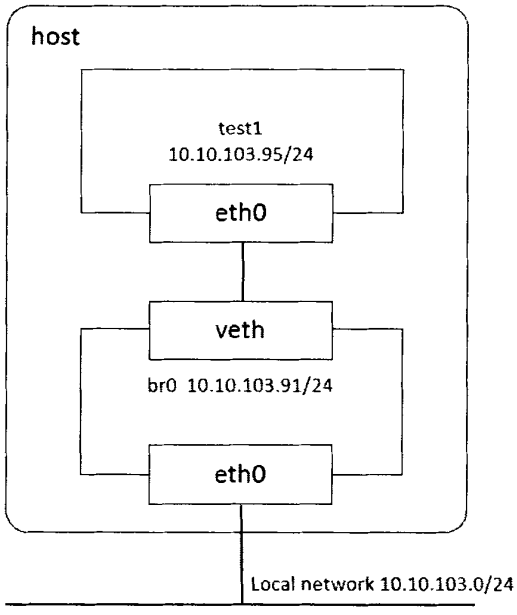

图 4.1 网桥实现的 Flat 网络拓扑图

图 4.1 即为一个使用 Linux Bridge 将容器桥接到本地网络的一个例子，test1 为 Docker 容器，br0 为 Linux Bridge，eth0 为物理机网卡。当将 eth0 桥接到 br0 上后，需要注意的一点是，eth0 由三层设备变为一个二层设备（交换机的一个端口），因此，其原本的 IP 地址需要配置到 br0 网桥上。

Linux Bridge 桥接网络虽然可以实现 Flat 网络，但是性能上不如 macvlan。因为容器中产生的网络包需要经过两个 veth 设备、一个网桥和一个物理网卡，经过的设备越多，需要的时间就越久。而 macvlan 方案，只需要经过两个设备。

4.1.1.3 macvlan 的 Flat 网络拓扑

macvlan 设备有 4 种工作方式，其中 bridge 模式类似于 Linux Bridge，同一个物理设备虚拟出来的 macvlan 设备如同连接在同一个网桥上。不仅如此，由于是同一个物理设备虚拟出来的，这个物理网卡也如同和这些虚拟设备连接在一起一样，这就有点像网桥桥接物理网卡的方式了。基于此种特性，本系统舍弃掉 Docker 网络原本的 docker0 网桥以及 veth pair 设备，直接使用 macvlan 虚拟设备代替两

者，即可实现 Flat 网络。其拓扑图如图 4.2 所示：

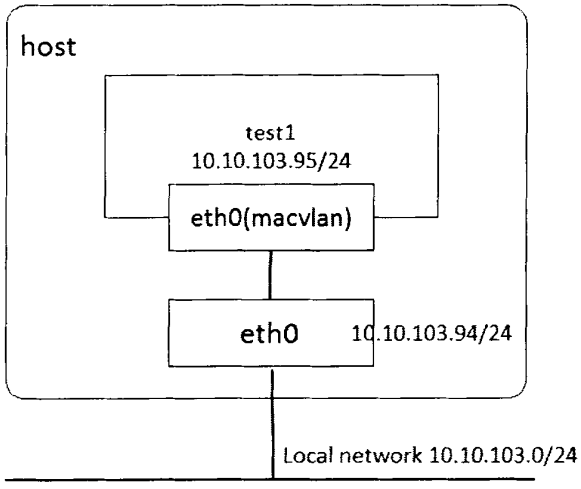

图 4.2 macvlan 实现的 Flat 网络拓扑图

容器 test1 的网卡 eth0 不再是 veth 设备，而是主机网卡虚拟出来的 macvlan 设备。这时，主机网卡工作在混杂模式，即会接受和自己 MAC 地址不匹配的二层帧。还有一点需要说明的是，容器通过 macvlan 连在了主机的网络中，可以和网络中的其他主机和容器通信，但是容器却不能直接和宿主机通信，需要通过另外一个 macvlan 设备作为媒介才行，这是和网桥桥接方式不一样的地方。

4.1.2 Flat 网络模块的具体实现

Flat 模块可以给一个具体的 Docker 容器配置 Flat 网络，其网络 IP 地址的获取是由用户静态指定的，并且可以保存其状态，在容器重启后自动获取之前配置的 IP 信息。Flat 模块功能比较独立，既可以作为一个命令行工具独立配置一个已经启动的容器，又可以嵌入到 Docker client、Docker daemon 的 HTTP 请求链中发挥作用。

4.1.2.1 Flat 网络模块参数说明及解析

Flat 模块给具体容器配置网络时，需要用户传入相应的配置参数才能完成。Flat 模块的配置参数如表 4.1 所示。

表 4.1 Flat 命令参数表

参数项	类型	参数说明
container-name	String	要配置的 Docker 容器名称或 ID
host-interface	String	macvlan 虚拟设备所依附的宿主机网卡
mode	String	macvlan 的工作模式，默认为 bridge
IP	String	容器所分配的 IP 地址，CIDR 格式
MTU	Int	macvlan 虚拟设备的 MTU，默认继承自父网卡
gateway	String	容器网络的网关

传入参数后，需要做相关的校验工作。container-name、host-interface 等必须验证其是否在主机中存在。IP、gateway 必须验证其格式以及两者是否属于同一网段。mode 和 MTU 有默认值，可以不指定。但若指定 MTU，根据 RFC791 中的规定，需要大于 68。mode 在 Flat 网络中应是 bridge 模式，但也有特殊情况，如需要严格安全隔离时，也可以使用 private 模式。还有一个隐性的注意点是 Docker 的网络模式，由于我们需要给容器自定义网络，所以需要容器存在自己的 netwrok namespace。因此，若需要配置的容器属于 host 或 container 模式，则不对其进行 macvlan 配置。host 模式下，容器和主机网络本身就是一体，不用配置；而 container 模式下，容器依附于其他容器，并不能保证其他容器是否有独立的 network namespace。

4.1.2.2 配置 Flat 网络的具体过程

给 Docker 容器配置自定义网络的过程如图 4.3 所示，主要过程如下：

（1） 创建 macvlan 设备。macvlan 设备即用来替换原 Docker 网络中的 veth pair 设备，作为 Docker 容器中的网卡，与外界交换数据。macvlan 设备需要依附于一块父网卡，具有不同于父网卡的 MAC 地址，并自动连接到了父网卡所在的二层网络中。

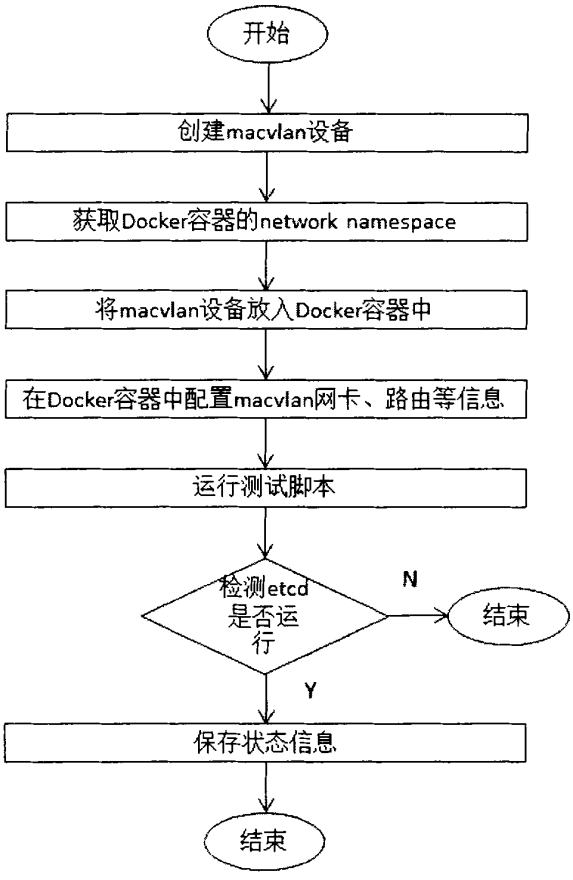

图 4.3 Flat 网络配置流程图

（2） 获取 Docker 容器的 network namespace。这是比较困难的一步，需要对 network namespace 有一个深入的了解。在 Linux 系统中，namespace 的 API 包括 clone()、setns()、unshare()。当使用 clone()系统调用创建新的进程时，可以通过传递 CLONE_NEWNET 参数创建新的 netwrok namespace，Docker 内部即使用此方法。因此，对于每一个进程来说，都有一个唯一个 network namespace 与之对应。如图 4.4 所示，每一个进程的/proc/[pid]/ns 下有 6 个 link 类型的文件，每一个文件指向一个类型的 namespace 编号，这些编号就唯一代表一个 namespace。这些 link 文件的另一个作用是，一旦这些文件被打开，只要打开的文件描述符（fd）存在，就算该 namespace

下的所有进程都结束，这个 namespace 也会一直存在。而通常一个 Docker 容器中只有一个进程，这样说来，Docker 的本质其实就是一个进程，获取了 Docker 内运行的进程，即可获得这些 link 文件。通过这些文件的打开文件描述符，即可获得所需要的 network namespace。

```
$ ls -l /proc/$$/ns          <<-- $$是shell中表示当前运行的进程ID号
total 0
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 net -> net:[4026531956]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 user->user:[4026531837]
lrwxrwxrwx. 1 mtk mtk 0 Jan  8 04:12 uts -> uts:[4026531838]
```

图 4.4 进程目录下的六种 namespace

（3） 将 macvlan 置入获取到的 Docker network namespace 中。一个网络设备只属于一个 network namespace，刚创建好的 macvlan 设备默认在主机的 network namespace 中，需要将其放入 Docker 容器的 network namespace 中。

（4） 进入 Docker network namespace 中进行 macvlan 网卡设置。将 macvlan 设备加入 Docker 容器中后，在宿主机上就看不到此设备了。需要进入 Docker 容器中，对其进行设置。设置的内容主要包括为设备重命名（如 eth0、eth1）、为设备添加 IP 地址、启动设备等。

（5） 配置 Docker 容器的默认路由。容器的网卡设备配置完成后，接着需要配置内核路由表，完成容器通信的最后一步。由于一个 netwrok namespace 中只能有一个默认路由，若容器本身处于 bridge 模式，则其已经配置了默认路由，默认网关即为 docker0 网桥的 IP，需要首先将此路由删掉，然后根据用户传进的 gateway 参数设置默认路由表。

（6） 运行测试脚本。以上配置完成之后，容器自动运行测试脚本，ping 宿主机网络的网关和 Internet 上的主机（如 www.baidu.com）,以确保配置完成，若出错，则输出相关的错误日志。

（7） 检测 etcd 运行状态，保存 IP 等状态信息。Docker 网络原本是无状态

的，即 IP 等信息在重启后会丢失。因为，在 Docker 容器关闭后，其中运行的进程结束，相应的 namespace 也丢失掉了，容器重启后的 namespace 和之前已经不同，所以相应的状态信息也会丢失。解决的办法是将状态信息另外存储，如本系统将 IP 等信息存储在了 etcd 集群中。在本系统中，etcd 集群默认是随着 macvlan daemon 的启动而启动的，因此，若单独使用 Flat 命令配置容器时，etcd 集群可能未启动。因此，最后一步会检测 etcd 是否运行，若运行则存储相关信息，否则不进行存储。所以 Flat 模块单独使用时，IP 状态的信息未必是保存的。

Flat 网络的配置过程和 Docker daemon 实际启动容器时配置网络的过程是相似的，只不过所使用的虚拟技术不一样。而本系统所采用的 macvlan 设备要比默认的 veth 设备加网桥的方式有更高的数据传输效率。

4.2 DHCP 网络模块的设计与实现

DHCP 模块使得容器可以从 DHCP 服务器上获取 IP 信息，其网络拓扑和 Flat 网络一样，只不过网络配置不再由用户指定，而是由本地 DHCP 服务器配置。

4.2.1 DHCP 网络模块参数说明及解析

除了网卡 IP 信息，用户还是需要指定如表 4.2 所示的配置参数。DHCP 模块也是可以作为一个独立命令单独使用的。

表 4.2 DHCP 命令参数表

参数项	类型	说明
host-interface	String	macvlan 虚拟设备所依附的宿主机网卡
container-name	String	要配置的 Docker 容器名称或 ID
mode	String	macvlan 的工作模式，默认为 bridge
MTU	Int	macvlan 虚拟设备的 MTU，默认继承自父网卡

续表 4.2

参数项	类型	说明
dhcp-client	String	指定 DHCP 客户端，可选
dhcp-option	String	DHCP 客户端命令选项，可选

DHCP 是 C/S 模型架构，首先，Docker 容器要连入的网络中必须要存在 DHCP 服务器。其次需要有 DHCP 客户端，此客户端不一定存在于要配置的 Docker 容器中，也可以是宿主机中的 DHCP 客户端。因为，从某方面来说，Docker 的本质其实是进程，那么只需在宿主机上运行 DHCP 请求时，将该进程的 netwrok namespace 设置为 Docker 容器所在的 network namespace 即可，可以使用 setns()系统调用实现。本系统中，用户可以通过 dhcp-client 命令行参数指定 DHCP 客户端，有 4 种选择，分别是 dhcpclient、udhcpc、dhcpcd 和 dhcp。前三种为正常的 DHCP 客户端，必须安装在要配置的容器或宿主机中。最后一种略微不同，是在当你主机中没有安装 DHCP 客户端时使用。它会使用 busybox 镜像（一个非常轻量级的 Docker 镜像）启动一个内置 udhcpc 客户端的 Docker 容器，并且此容器的网络模式为 container 模式，即 busybox 容器与要配置的容器共享 netwrok namespace，这样便可以解决 DHCP 客户端不存在的问题。

还有一个用户参数是 dhcp-option，即可以为所使用的 DHCP 客户端命令指定额外的选项，以完成更丰富的功能。如客户端指定为 dhcp，选项指定为-f。那么，系统将会创建 busybox 容器，并将-f 选项传入，告诉此客户端应在前台运行而不是后台。当没有-f 选项时，DHCP 客户端获得网络配置和一个租期后，DHCP 客户端到后台运行，而此时 busybox 中 PID 为 1 的进程退出，导致整个 busybox 容器关闭。这样，在被配置的容器获得网络配置后，DHCP 客户端已经关闭，那么当网络配置到期后，不能到 DHCP 服务器处续约，会导致 DHCP 服务器又将相同的网络参数配置给网络中的其他客户端，引起 IP 地址冲突。而加上此选项后，则不会有这种问题，但是在被配置的容器关闭后，需要记得手动关闭此 busybox 容器。

4.2.2 为容器配置 DHCP 网络的过程

给容器配置 DHCP 网络的过程和配置 Flat 网络的过程基本相同，有如下几步：

（1） 为容器创建 macvlan 虚拟设备。

（2） 获取容器的 network namespace，将 macvlan 设备置入容器中。

（3） 检查 DHCP 客户端，发起 DHCP 请求。

（4） 获得网络配置和租期，配置 macvlan 设备。

（5） 检查 etcd 集群状态，保存容器配置信息。

4.3 VLAN 网络模块的设计与实现

4.3.1 VLAN 网络模块的设计

在多租户的云环境下，多个用户的容器可能连在同一个虚拟交换机中，安全隔离显得至关重要，一般企业都选择采用 VLAN 技术在二层隔离网络。由于 Open vSwitch 本身提供对 VLAN 的有效支持，使用 OVS 可以方便的为容器网络配置多主机的 VLAN。因此，业界普遍采用 OVS 来实现 VLAN 隔离。而本系统所使用的 macvlan 在连接容器到主机网络中虽然相当的快捷和方便，但是在实现二层隔离方面确实“心有余而力不足”。对此，业界曾有过一个错误的方案是这样的，将容器都按 Flat 那样的网络连接起来，然后为不同租户或业务的机器配置上不同的网段，这样这些网段上的容器之间也是不能相通的。但是，这是伪的二层隔离，因为，不同网段之间还是可以收到各自的广播帧，并没有隔离广播域，这仅仅算是三层隔离。因此，仅凭 macvlan 设备是不够的，还需要 Linux 下的 VLAN 模块“帮忙”。

Linux 上的 VLAN 和 Open vSwitch 等交换机的 VLAN 略微不同，后者是先有一个大的 LAN，再划分不同的 VLAN。而 Linux 则正好相反，需要先创建 VLAN，再将相关的设备连接到该 VLAN 上。Linux 上创建 VLAN 也很简单，使用 ip 命令即可创建 VLAN 设备。和 macvlan 设备一样，VLAN 设备也需要有一个父网卡，数据就是从父网卡上发送出去的。除了指定父网卡外，还需指定 VLAN ID，表示

它可以承载相应的数据帧，并且在数据帧从该网卡发出时，打上相应的 tag，如图 4.5 所示。

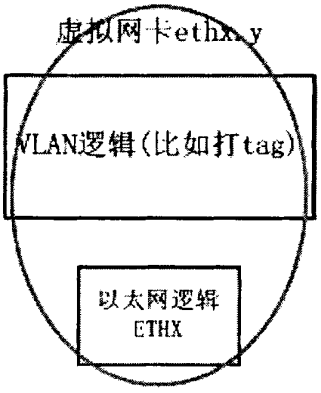

图 4.5 一块物理网卡虚拟出单 VLAN 设备[32]

若一块父网卡上虚拟出了多个 VLAN 设备，则它可以允许不同的 VLAN ID 数据帧通过，因此它就相当于 trunk 端口，如图 4.6 所示。

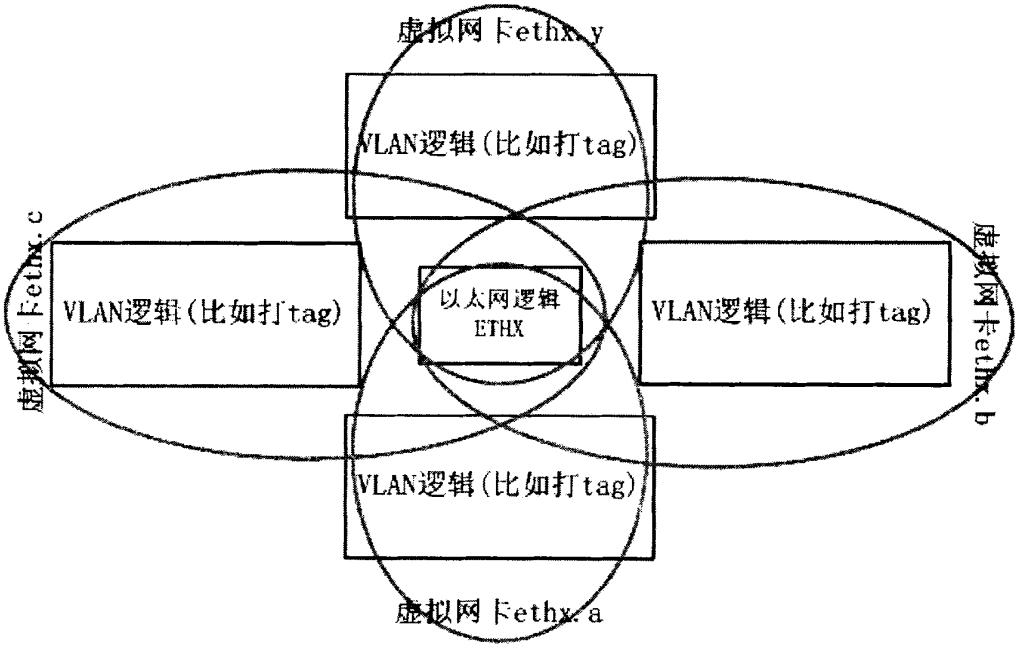

图 4.6 一块物理网卡虚拟出多 VLAN 设备[32]

有了划 VLAN 的方法，再结合 macvlan 就可以实现系统所需的目的。具体来

说，就是先创建 VLAN 虚拟设备，再在这个 VLAN 设备上虚拟出 macvlan 设备，则相当于将 macvlan 所连接的机器加入了相应的 VLAN，如图 4.7 所示。

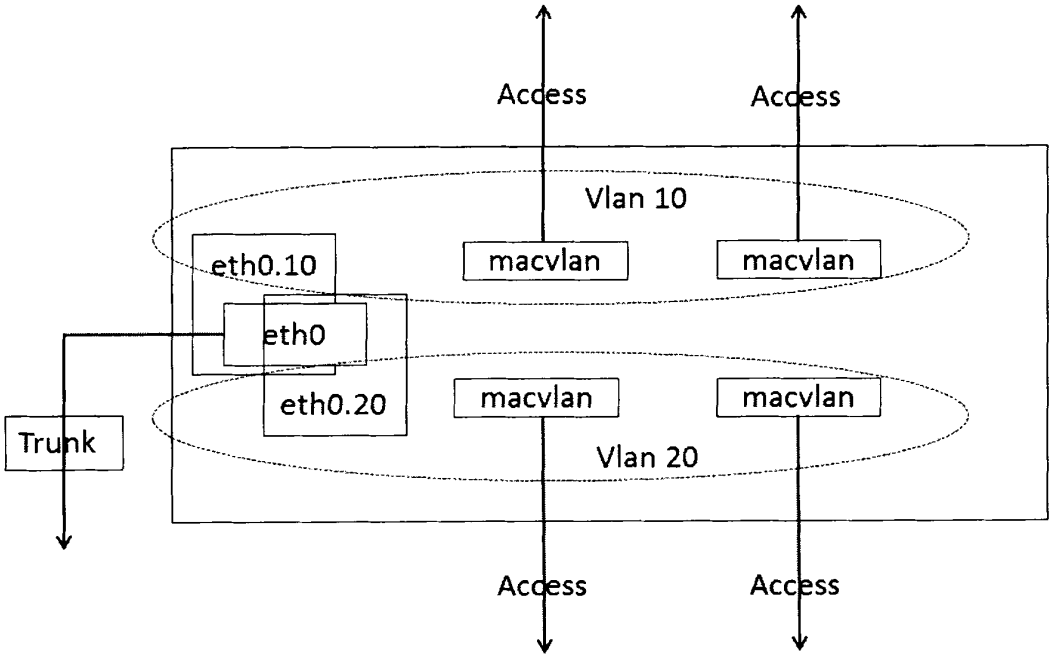

图 4.7 macvlan 实现的 VLAN 模型

VLAN 划分解决后，接下来还要考虑一个问题，就是 VLAN 如何与外界通信的问题。在 Docker 的 bridge 网络中，有 docker0 网桥充当容器的网关；在 macvlan 的 Flat 网络中，有本地网关可以使用。现在，VLAN 将容器和主机网络隔离成了不同的二层网络。容器想要出外网，必须有一个同网段的网关地址，使得流量路由出去。那么这个网关 IP 要如何设置？设置在什么地方？都是需要解决的问题。网关的设置首先要考虑网关 IP 的可达性，首先其必须和要配置的 VLAN 属于同一个 VLAN ID，其 IP 也应属于同一个网段，因此，我们可以再在同一个 VLAN 网段中虚拟出一个 macvlan 作为网关。其次，设置在哪里的话有两种方案。一种是分布式路由，即每一台宿主机上都有其上相关 VLAN 网段的网关，若主机直接连着外网，则每一台宿主机上的 Docker 容器直接通过本主机网络通向外网，每台宿主机只为自己机器上的容器负责。这么做，优点很明显，即网关没有单点故障，

且流量分配均匀。缺点是每一个网关都占用一个 IP 地址，且不同机器上容器网关设置不统一，配置复杂。另一种方式是每一个 VLAN 只有一个单网关，所有宿主机上的容器外出流量都要经过这个网关。这种做法的缺点就是单点故障，且容器流量都通过网关节点外出，有可能造成网络拥塞。具体选择哪一种，需要根据实际情况和需求。由于我们实验室宿主机网络也是属于内网，需要通过网关访问互联网，即便使用分布式网关，最后流量还是会聚集到主机网络的物理网关上，且实际情况下，容器访问外网的需求也不是非常大，因此，本系统采用第二种方案，在宿主机网关的网卡上虚拟出各个 VLAN 网络的网关，最终的网络拓扑图如图 4.8 所示。

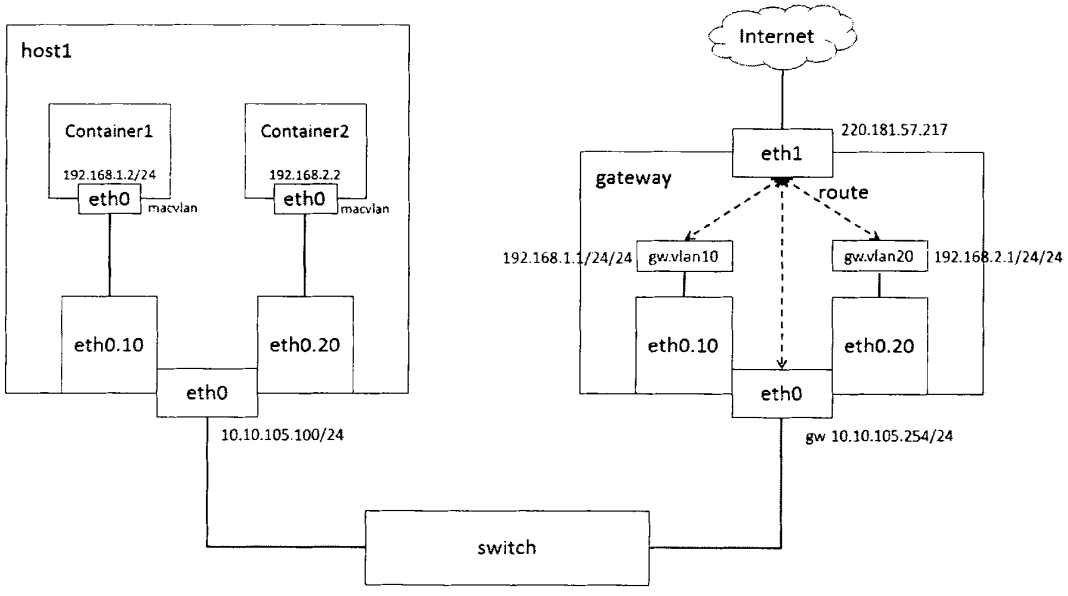

图 4.8 VLAN 网络下容器与外界通信拓扑图

4.3.2 VLAN 网络模块的具体实现

4.3.2.1 创建、删除 VLAN

VLAN 模式下，用户创建容器之前先要创建 VLAN。创建 VLAN 需要指定表 4.3 所示的参数选项。Subnet 为三层网络地址，由用户指定，由于所有子网的网关都设置在同一台机器上，为了避免 IP 冲突，不同 VLAN 的子网 IP 地址不能重

复，创建 VLAN 的时候会检查此选项。此外，VLAN 模块需要 etcd 集群保存 VLAN 状态信息，若没有检测到 etcd 集群，则会创建失败，这是和 Flat 和 DHCP 模块不同的地方。

表 4.3 创建 VLAN 参数表

参数项	类型	说明
Name	String	VLAN 网络的名字
Subnet	String	VLAN 网络的 IP 范围，CIDR 格式
Host-interface	String	VLAN 虚拟设备所依赖的主机网卡
etcd	String	etcd 服务访问点

参数校验完毕后，接着系统执行创建 VLAN 的过程，如图 4.9 所示。系统会随机从 0-4095 中随机选取一个未被占用的数字作为此 VLAN 的 VLAN ID。然后根据用户传入的 subnet 值构建 IPAllocator 对象，IPAllocator 对象即为本 VLAN 的 IP 地址管理、分配机构。IPAllocator 对象的内部实现可简单的抽象为两个集合，一个为未使用的 IP 集合，一个为已经使用过的 IP 集合。当在本 VLAN 网络中创建一个 Docker 容器时，IPAllocator 随机分配一个或由用户指定的一个未使用的 IP 给容器。当销毁容器时，IPAllocator 回收其 IP。接着系统会将 VLAN 相关信息保存在 etcd 集群的 vlan 目录下，key 为 VLAN 网络的名称，value 的格式为“vlanid，host-interface，subnet，IPAllocator”。创建 VLAN 的最后一步便是配置 VLAN 的网关，通常使用 subnet 参数中的第一个 IP 作为本 VLAN 网络的网关地址。先在宿主机网络的网关网卡上虚拟出相应 VLAN ID 的 VLAN 虚拟设备，然后再使用此虚拟设备创建 macvlan 设备，接着为此 macvlan 设备配置上 IP 地址即可。这里的 macvlan 设备无需加入任何容器的 network namespace 中，留在宿主机中即可，系统会根据其 IP 自动生成相应的内核路由表。删除 VLAN 网络的过程与创建 VLAN 的过程基本相反，包括删除网关设置、销毁 IPAllocator 对象、删除 etcd 中

的存储信息等。删除 VLAN 需要确保本 VLAN 网络中已经没有 Docker 容器，否则删除失败。

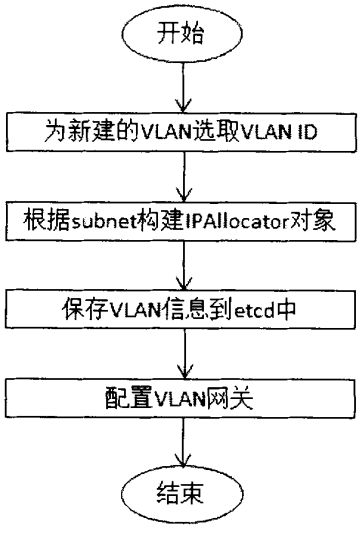

图 4.9 创建 VLAN 流程图

4.3.2.2 在 VLAN 网络中启动容器

当在一个 VLAN 网络中启动容器时，用户需要指定如表 4.4 所示的参数。其中 IP 参数可以由用户指定，也可以由 IPAllocator 自动分配。由用户指定的 IP 地址必须在 subnet 网络范围之内。

表 4.4 VLAN 网络下容器启动参数表

参数项	类型	说明
name	String	容器要加入的 VLAN 网络名称
container-name	String	容器的名称或 ID
IP	String	要配置容器的 IP，可以不指定

系统接收到相关参数后，接着就配置相关的容器网络，过程如下：

（1） 根据 VLAN 网络的名称获取 etcd 集群中 VLAN 网络的配置参数。

（2） 判断容器宿主机上是否存在相应 VLAN ID 的虚拟设备，若不存在，则使用主机网卡虚拟出 VLAN 设备，作为 macvlan 虚拟设备的父设备。

（3） 使用虚拟的 VLAN 设备创建 macvlan 设备，将 macvlan 设备放入目标容器的 network namespace 中。

（4） 向 IPAllocator 申请 IP 地址，IPAllocator 返回一个可用的 IP 并做出记录。

（5） 进入容器的 network namespace 中，对容器进行 IP 和路由的配置。

（6） 将容器的 IP 等状态信息保存到 etcd 集群中。

4.3.2.3 etcd 中容器的状态信息存储

无论使用 Flat、DHCP、VLAN 哪种网络，最后，系统都会将容器网络状态信息保存到 etcd 集群中。当容器关闭重启时，会首先从 etcd 集群中获取到该容器之前的配置信息，然后重新装填到容器中。这样避免了 Docker 容器关闭时，网络信息全部丢失的问题。如图 4.10 所示，容器的配置信息分别保存在/flat、/dhcp、/vland 目录下，VLAN 网络的信息保存在/vlan 目录下。

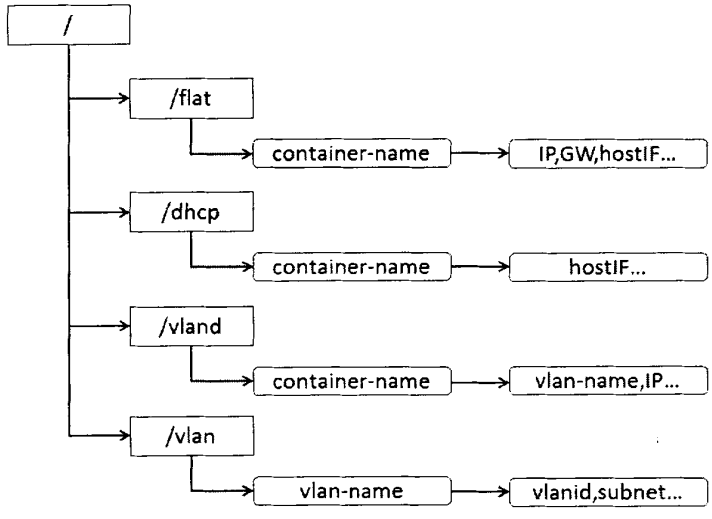

图 4.10 etcd 存储容器信息的目录树

4.4 QoS 模块的设计和实现

多个容器在同一台宿主机上会共享宿主机的带宽，为了不让容器间互相影响，本系统还支持为特定的容器提供带宽限制。为了提高系统的兼容性，本系统同时支持原 Docker 网络容器（Linux Bridge/veth pair 方式）和基于 macvlan 的 Docker 网络容器的 QoS 设置。其原理为：通过使用 Traffic Control 在容器网卡上建立无类别 qdisc（queueing discipline）对容器流量进行整形（排序、限速、丢包），以达到限速的目的。由于只是进行简单的速率限制，系统采用 TBF（Token Bucket Filter）无类算法实现限速。在原生 Docker 网络中，qdisc 建立在 veth pair 的任何一端都可以，而使用 macvlan 的 Docker 容器，则必须在该 macvlan 网卡上建立 qdisc。限制具体容器带宽时，需要指定如表 4.5 所示参数，以完成对容器的速率限制。

表 4.5 QoS 模块参数表

参数项	类型	参数说明
Container-name	String	容器名称
rate	Int	限制容器的速率，单位 kbit
brust	Int	桶尺寸，单位 byte
latency	Int	数据包最长等待时间，单位 ms

4.5 容器间服务发现模块的设计与实现

本系统除了可以配置容器的三种网络之外，还支持服务发现。一个 Docker 容器一般运行一个进程的原则，使得应用模块化、细分化的更加明显，形成现在流行的所谓微服务架构。多个容器之间相互依赖、相互协作共同提供一个强大的服务，这种场景越发普遍。基于此，服务发现就变的相当重要。服务发现是让这些相互依赖的服务，可以相互感知对方，并建立连接的过程。具体来说就是使得

容器在没有人为配置的情况下而发现所需服务的位置，并可以自行注册自身以便让其他组件发现自己。如图 4.11 即为服务发现的一个例子。应用 A 启动后，在服务发现机制下注册自己。而 B 需要与 A 通信时，通过查找机制找到 A 在网络中的位置，然后与 A 建立连接。一般，通信时只需指定服务的名字即可，而不用关心具体的 IP 地址和端口，这有点类似 DNS。并且，当服务地址改变时，服务发现机制也会自动更新其地址。

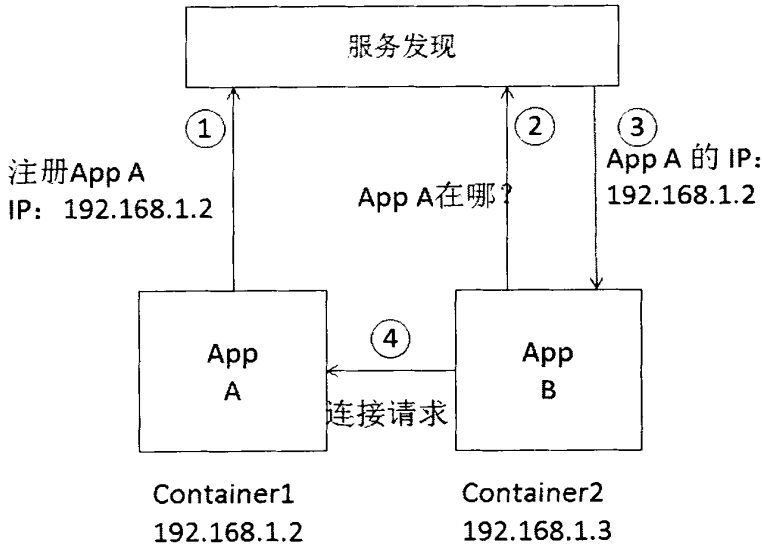

图 4.11 服务发现示意图[33]

可以看到，服务发现机制需要一个应用注册其地址的地方。在分布式环境中，一般是一个强一致性、高可用的存储目录，如 etcd。本系统中就是使用 etcd 来提供服务发现机制，etcd 提供了 HTTP 接口用来存储、更新和获取值，正好用来注册和发现服务，并且可以设置 key TTL（存活时间），定时保持服务心跳以监听应用的状态。本系统可以使用 publish 命令对需要使用服务发现的容器进行配置。原理就是当一个 Docker 容器启动时（一个服务上线），将其上应用所监听的 IP 与端口记录到 etcd 集群中，当容器关闭或应用失效后，将其从 etcd 中删除。

为了将应用注册到 etcd 集群中，有如下两种方法：

（1）　一种是应用能自己能建立连接到 etcd 中，并且创建键值来通知其它服

务它已启动，当应用关闭时，应用必须移除键值。但是，当应用崩溃时，就不容易及时删除 etcd 中的键值信息。这么做，也会使应用变得复杂。

（2） 另外一种就是将应用与注册进程分离。当容器启动时，会在容器中另外执行一个脚本，来进行应用的注册管理。

本系统采用第二种方式来实现应用的注册。当容器启动后，会创建一个名为“dockerReg”的进程。该进程的工作主要为：定期检查应用的运行状态，当应用运行正常时，会将应用信息创建或更新到 etcd 集群中，保存的目录为/service，key 为服务名称（一般为容器的名称），value 为容器中应用监听的 IP 地址及端口号；当应用没有响应或容器关闭时，该进程将对应的信息从 etcd 中删除。同时，dockerReg 进程可接收如表 4.6 所示参数（可选），当未指定 service-name 和 IP 时，dockerReg 可通过 Docker API 获取容器的名称和 IP，整个过程如图 4.12 所示。注册完成后，其他容器就可以通过查询 etcd 中/service 目录下的服务名称，找到相关服务在网络中的具体位置。

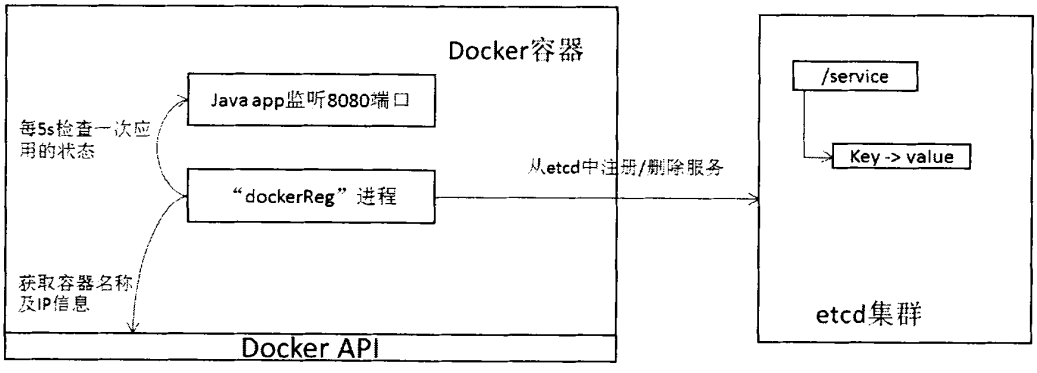

图 4.12 dockerReg 进程

表 4.6 dockerReg 接收参数表

参数项	类型	参数说明
service-name	String	服务名称

续表 4.6

参数项	类型	参数说明
IP	String	容器地址
Port	String	应用监听端口号

4.6 本章小结

本章介绍了 macvlan 功能模块区的设计和实现，主要包括 Flat、DHCP、VLAN 三种网络模型。Flat 和 DHCP 网络可以规避 NAT、为容器配置自定义 IP 以及解决跨主机通信的问题。VLAN 网络解决了多租户环境下二层网络隔离的问题。Docker VLAN 网络的实现业界通常采用 Open vSwitch，本系统使用 Linux 内核原生的 VLAN 模块加 macvlan 设备实现了 Docker 的 VLAN 网络。此外，还介绍了本系统的网络 QoS 模块和服务发现机制。

第5章 macvlan daemon 的设计与实现

要实现可插拔的网络配置服务，需要有和 Docker daemon 灵活对接的方案。本系统中 macvlan daemon 即用来实现这一需求，在使用 Docker 的地方，只需修改 Docker client 中 Docker daemon 的 url 就可以无缝对接 macvlan daemon。使得 macvlan 的网络方案可以不局限于任何第三方的平台，可以随意的与 kubernetes、swarm、mesos 等容器资源调度工具结合使用。

5.1 macvlan daemon 架构设计

macvlan daemon 本质上是一个 HTTP 代理服务器，位于 Docker client 和 Docker daemon 之间，扮演中间人的角色[34]。Docker client 会向 macvlan daemon 发送请求报文，macvlan daemon 会像 docker daemon 一样处理请求和连接，然后将结果返回给 Docker client，在这个过程中，macvlan daemon 充当 HTTP 服务器的角色。同时，macvlan daemon 自己也要向 Docker deamon 发送请求，并接受 Docker daemon 的响应，这时，其又充当 HTTP 客户端。macvlan daemon 架构如图 5.1 所示。其主要包括 HTTP server、router、handler、HTTP client 四个模块。HTTP server 和 HTTP client 用来接收 HTTP 请求和重新构造 HTTP 请求。router 用来将请求路由到不同的 handler 上进行处理。handler 即 macvlan daemon 作为中间人的角色在一个请求响应中所做的具体工作，主要有 CreateContainer、StartContainer、DeleteContainer、JustForward 四个 handler，可以调用相应网络模块完成容器的配置工作。

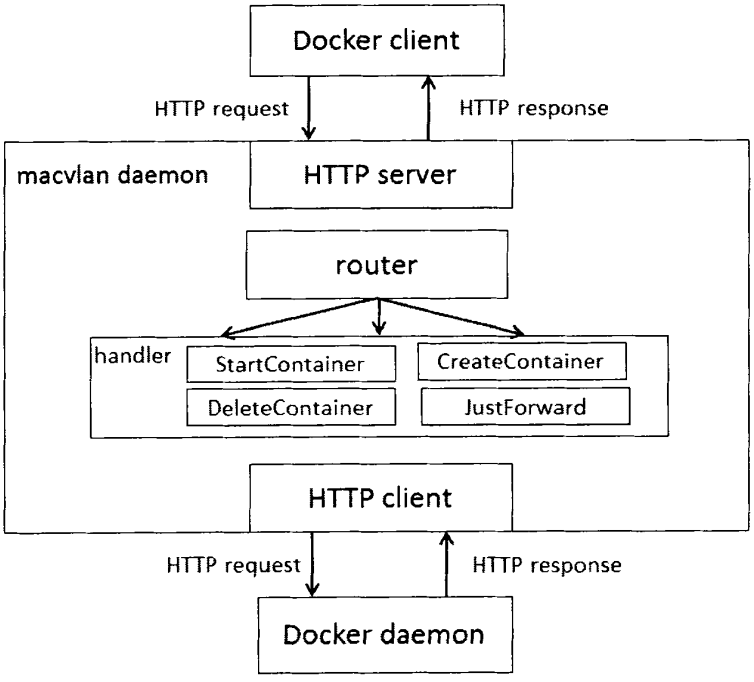

图 5.1 macvlan daemon 架构图

5.2 macvlan daemon 实现

5.2.1 macvlan daemon 启动参数与 Docker 环境变量

macvlan daemon 启动需要指定如表 5.1 所示的参数。endpoint 为 macvlan daemon 监听的地址，Docker client 即与此地址通信。docker-daemon 的值为真正处理 Docker 请求的服务地址。

表 5.1 macvlan daemon 启动参数表

启动参数	类型	说明
debug	Bool	是否开启 dubug 模式
endpoint	string	本地 socket 文件或所监听的 tcp 地址

续表 5.1

启动参数	类型	说明
etcd	string	etcd 服务器集群地址
docker-daemon	string	Docker daemon 的 url，socket 文件或 tcp 地址

做一个自己的 Docker HTTP 代理程序，需要了解 Docker client 和 Docker daemon 之间的通信规则，本系统使用 Docker Remote API v1.19 进行通信。要实现可插拔的模式，就不能对 Docker Remote API 做修改。因此，需要使用其他方式来自定义请求，实现对容器进行具体的网络配置。macvlan daemon 采用 Docker 环境变量的方式来携带所需的信息，Docker 的环境变量类似于 Linux 系统的环境变量，以键值对的形式存储在系统中。用户使用 docker run 运行容器或使用 dockerfile 创建镜像时，都可以使用类似 ENV 的指令指定环境变量。此外，环境变量还可通过创建容器的 HTTP 请求由用户传递给容器。Docker 的 HTTP 请求主体为 JSON 格式，环境变量的信息保存在“Env”字段中，如图 5.2 所示。“Env”为一个键值对的数组，可以用来保存多个环境变量。macvlan daemon 在创建容器时，就是根据此字段的值来调用相应的网络模块配置容器的。

```
"OpenStdin": false,
"StdinOnce": false,
"Env": [
        "FOO=bar",
        "BAZ=quux"
],
```

图 5.2 创建容器的 HTTP 请求

5.2.2 macvlan daemon 各个组件实现

HTTP server

macvlan daemon 可以监听两种类型的地址，一种为网络中的 TCP 套接字，用 IP：PORT 表示；一种为本地 Unix 域套接字文件，Unix 域套接字文件只能用

于同一主机之间的进程通信。HTTP server 由 GO 语言自带的 net/http 包实现，可以解析 HTTP 请求，并将 HTTP 响应发回给 Docker client。

router

macvlan daemon 接收到请求后，请求传递给 router，router 根据请求行中的 HTTP 方法和 url 判断将请求转发给哪一个 handler 处理，如图 5.3 所示。若请求行包含“POST /containers/create”，则为创建容器的请求，转发给 CreateContainer 函数处理。若请求行包含“POST /start”字样，则为启动容器请求，转发给 StartContainer 函数处理。若请求中的 HTTP 方法为 DELETE，则为删除容器的请求，交给 DeleteContainer 函数处理。其他请求则都由 JustForward 函数处理。

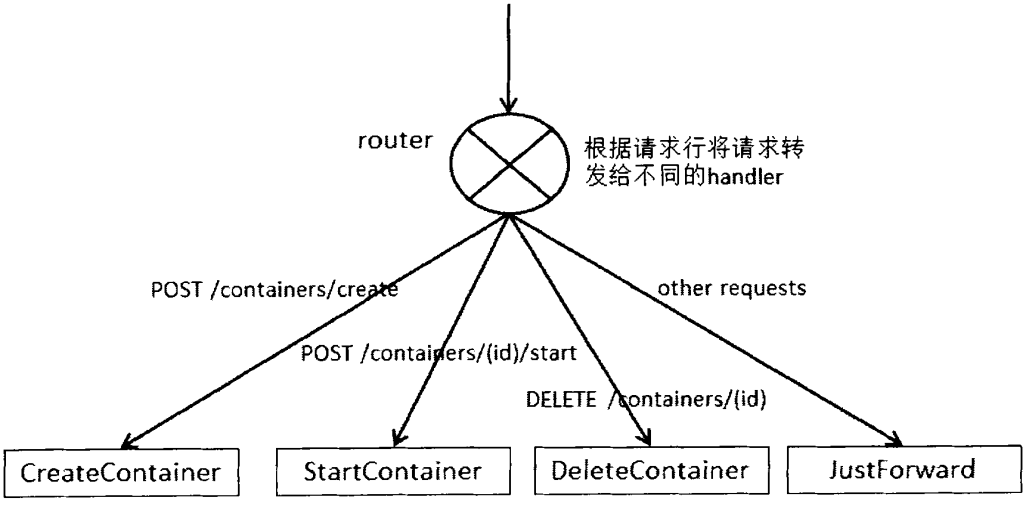

图 5.3 router 请求转发图

handler

macvaln daemon 中实现了 4 种 handler 来处理不同的请求。

（1） CreateContainer。这是最为重要的一个 handler。之前提到过，为 Docker 容器设置的环境变量会通过创建请求传递给 Docker daemon。而本系统需要通过环境变量传递一些容器的网络信息，macvlan daemon 通过分析创建容器的请求主体即可获得所需的信息。本系统规定的环境变量如表 5.2 所示。

表 5.2 Env 环境变量说明

环境变量	环境变量的意义
TYPE	该容器网络类型，三个可选 flat|dhcp|vlan
HOSTIF	macvlan 虚拟网卡的父网卡
IP	容器 IP 地址
GW	容器网关地址
VLANID	要加入的 VLAN 的名字
RATE	限制容器的速率，单位 kbit
BRUST	桶尺寸，单位 byte
LATENCY	数据包最长等待时间，单位 ms
DISCOVER	是否在 etcd 上注册服务 true|false
PORT	容器应用所监听的端口

这些环境变量不需要全部指定，根据该容器的网络类型来确定哪些环境变量是必须的。如若容器类型为 DHCP，则只需指定“HOSTIF”变量即可。从请求中获取环境变量之后，再依次获取容器的默认网络模式和容器名称。接着 CreateContainer 将请求发送给 Docker daemon，若容器创建成功，即返回的 HTTP 状态码为 201，则根据容器默认网络情况和 TYPE 的值将容器网络信息保存在 etcd 集群中，流程如图 5.4 所示。以上处理完成后，将 Docker daemon 返回的 HTTP 响应包发回给 Docker client。

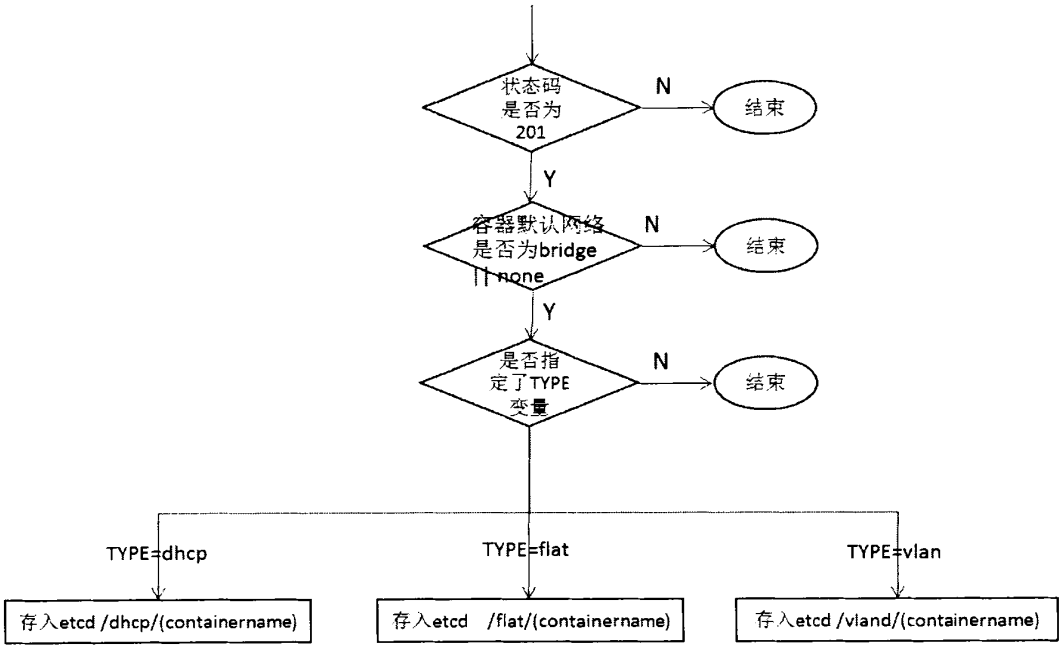

图 5.4 CreateContainer 执行流程图

（2） StartContainer。容器启动的请求是由 StartContainer 处理的。处理的流程为先启动容器，然后再调用相应的网络模块配置容器的 macvlan 网络。其流程图如图 5.5 所示。首先，StartContainer 从请求中获取要启动容器的名字，然后将请求复制一份发送给 Docker daemon。当容器启动成功并返回响应码为 204 时，接着判断 etcd 集群中，是否存在此容器的记录。主要从/dhcp、/flat、/vland 目录下查找相应的 Docker 名称，查找到之后，从中获取 value 的值，然后作为参数调用相关网络模块对容器进行配置。若不存在 etcd 集群中，则不做任何配置。

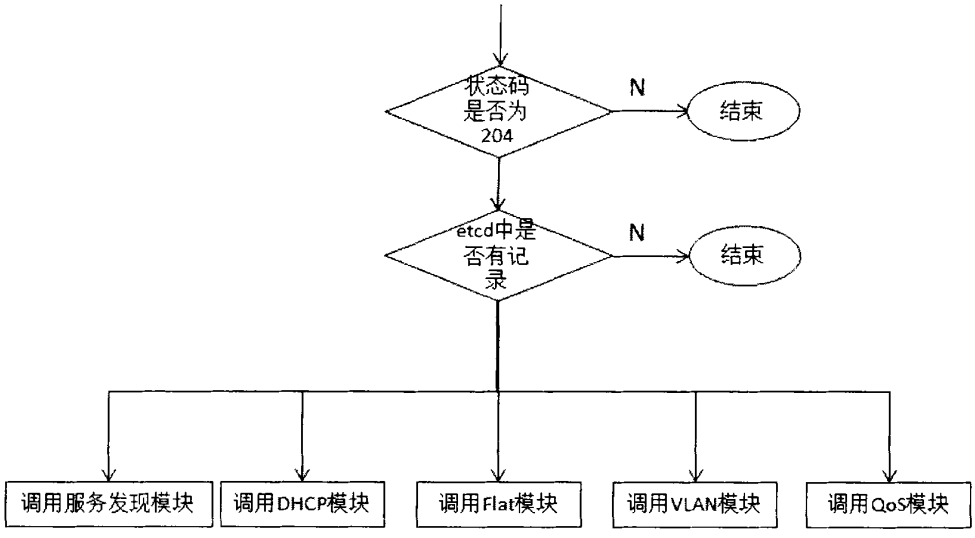

图 5.5 StartContainer 执行流程图

（3） DeleteContainer。DeleteContainer 处理删除容器的请求。删除容器的操作还是交由 Docker daemon 处理，而 DeleteContainer 主要删除容器在 etcd 集群中所存储的数据。首先从请求中获取要删除容器的名称或 ID，在 etcd 中查找是否存在该容器，若存在，则将其数据删除。然后，将请求复制一份转发给 Docker daemon 进行真正的删除容器操作。

（4） JustForward。除了以上的 HTTP 请求外，其他请求都是由此 handler 处理。从名字可知，此函数仅仅做转发处理，并不对请求响应以及容器做任何改变。

HTTP client

构造 HTTP 请求发送给 Docker deamon，HTTP 请求的各个部分从原始 HTTP 请求中复制过来，只修改少数部分或不做修改。

5.3 本章小结

本章主要介绍了 macvlan daemon 的实现。macvlan daemon 主要实现了一个 HTTP 代理服务器，将 Docker client 的 HTTP 请求转发到 Docker daemon，同时可

以根据 HTTP 请求中的信息调用相应的功能模块配置 Docker 容器。macvlan daemon 的存在使得本网络系统可以轻易地与原生 Docker 集成，在任何使用 Docker 的地方，都可以方便地使用本系统。

第6章 系统功能展示及对比性能测试

6.1 功能展示

本节就系统的主要功能模块——macvlan daemon、Flat、DHCP、VLAN、QoS、容器间服务发现做一个演示。操作系统版本：Ubuntu14.04，Docker 版本：Docker 1.7.0。

6.1.1 启动 macvlan daemon

本系统无需 OVS 等第三方软件依赖，编译完成后，可在命令行直接使用 macvlan 命令来启动 macvlan daemon，如下所示：

```
$ macvlan -d --etcd=0"http://127.0.0.1:2379"
INFO[0000] socketPath:[/run/docker/plugins/macvlan.sock],
INFO[0000] etcd:[http://127.0.0.1:2379],
INFO[0000] macvlan daemon initialized successfully.
```

6.1.2 使用 Flat 网络创建容器

本系统可以很方便地与 Docker daemon 集成，直接使用 docker run 命令就可以使用该系统的功能。下面使用 docker run 命令创建一个 Flat 网络的容器，主机的 IP 网段为 192.168.0.0/24，给容器指定的 IP 为 192.168.0.100，如下所示：

```
$ docker -H unix:///run/docker/plugins/macvlan.sock run -it
--name=ftest --net=none -e TYPE=Flat -e HOSTIF=eth0 -e
IP=192.168.0.100/24 -e GW=192.168.0.1 ubuntu
# root@f6d3b6a38414:/# ip a            //在容器中
7: eth1@if2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN group default
    link/ether 66:e5:93:6f:0a:ca brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.100/24 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::64e5:93ff:fe6f:aca/64 scope link
    valid_lft forever preferred_lft forever
# root@f6d3b6a38414:/# ping 192.168.0.1      //与网关通信
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=1.32 ms
…
```

6.1.3 使用 DHCP 网络创建容器

同样的，可以使用 docker run 创建 DHCP 网络的容器，主机 IP 网段为 192.168.0.0/24，容器 IP 由 DHCP 服务器分配，如下所示：

```
$ docker -H unix:///run/docker/plugins/macvlan.sock run -it
--name=dtest -e TYPE=dhcp -e HOSTIF=eth0 ubuntu /bin/bash
root@dd952efd7bb4:/# ip a                    //在容器中
11: eth1@if2: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN group default
    link/ether 9e:3c:98:99:85:5e brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.127/24 brd 192.168.0.255 scope global eth1
       valid_lft forever preferred_lft forever
root@dd952efd7bb4:/# ping 192.168.0.1     //与网关通信
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=0.935 ms
…
```

6.1.4 使用 VLAN 网络创建容器

使用 VLAN 网络之前，首先要创建 VLAN，创建 VLAN 命令如下：

```
$ macvlan create-vlan --name=test1 --subnet=10.10.10.0/24
--host-interface=eth0
$ macvlan create-vlan --name=test2 --subnet=10.10.20.0/24
--host-interface=eth0
```

以上分别创建了名为 test1 和 test2 的 VLAN 网络，下面分别在这两个网络上各起一个容器 c1、c2，如下所示：

```
$  docker -H unix:///run/docker/plugins/macvlan.sock run -itd
--name=c1 -e TYPE=vlan -e VLAN=test1 ubuntu /bin/bash
$  docker -H unix:///run/docker/plugins/macvlan.sock run -itd
--name=c2 -e TYPE=vlan -e VLAN=test2 ubuntu /bin/bash
// c1 和 c2 相互隔离，如下
root@7f083f054825:/# ifconfig eth1          //在容器 c2 中
eth1      Link encap:Ethernet  HWaddr 9e:3c:98:99:85:5e
          inet addr:10.10.20.2  Bcast:10.10.20.255
          inet6 addr: fe80::9c3c:98ff:fe99:855e/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:555 errors:0 dropped:0 overruns:0 frame:0
          TX packets:32 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:37537 (37.5 KB)  TX bytes:3190 (3.1 KB)
root@7f083f054825:/# ping 10.10.10.3        //ping c1
PING 10.10.10.3 (10.10.10.3) 56(84) bytes of data.
From 10.10.10.2 icmp_seq=1 Destination Host Unreachable
```

可以看到，容器 c1 和 c2 在不同的 VLAN 网络中，它们之间是隔离的。

6.1.5 容器网络 QoS 测试

启动一个 Flat 网络的容器，将该容器速率限制为 500kbit/s，并使用 iperf 工具测试带宽，如下所示：

```
$ docker -H unix:///run/docker/plugins/macvlan.sock run -it
--name=ftest --net=none -e TYPE=Flat -e HOSTIF=eth0 -e
IP=192.168.0.100/24 -e GW=192.168.0.1 -e RATE=500 ubuntu
root@1ab4d6bf03ab:/# iperf -c 172.17.1.2   //在容器中测试带宽
------------------------------------------------------------
Client connecting to 172.17.1.2, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[ 3] local 172.17.1.1 port 60482 connected with 172.17.1.2 port 5001
[ ID] Interval Transfer Bandwidth
[ 3] 0.0-10.0 sec 625 Bytes 500 kbits/sec
```

6.1.6 跨主机通信测试及容器间服务发现

以上实验是在单主机环境下做的测试，下面使用两台 Ubuntu14.04 主机（host-A 和 host-B）做跨主机通信的测试以及容器的服务发现，容器网络使用之前创建的 test1 VLAN 网络。

在 host-A 上，启动一个 web 应用（监听 8000 端口），并使用 publish 命令进行服务发现的注册。

```
$  docker -H unix:///run/docker/plugins/macvlan.sock run -it
--name=webapp -e TYPE=vlan -e VLAN=test1 fmzhen/webapp python app.py
root@21578ff721a9:/# ip add show eth0
34: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ec:41:35:bf brd ff:ff:ff:ff:ff:ff
    inet 10.10.10.3/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:ecff:fe41:35bf/64 scope link
       valid_lft forever preferred_lft forever
$ macvlan publish --port=8000 webapp
```

在 host-B 上，启动一个数据库应用（监听 3306 端口），并进行服务发现的注册。

```
$  docker -H unix:///run/docker/plugins/macvlan.sock run -itd
--name=mydb -e TYPE=vlan -e VLAN=test1 fmzhen/db
$ macvlan publish --port=3306 mydb
root@d217828eb876:/# curl http://webapp          //在容器中访问 webapp
Hello World!
```

webapp 和 mydb 两个容器分别处在不同的宿主机上，但在同一个网络中，并且在 etcd 集群中注册了各自的信息。从 mydb 中可以通过名字访问 webapp，说明两容器间可以跨主机进行通信，并且服务发现机制也发挥了作用。

6.2 对比性能测试

6.2.1 macvlan 设备和 veth pair 设备数据传输性能对比

Docker 原生的容器间通信是通过 Linux Bridge 和 veth pair 实现的，本系统使用 macvlan 设备替换了它们，其通信性能会有怎样的变化？本节使用 iperf 工具对两种方式进行测试。iperf 是一个网络性能测试工具，工作模式为 c/s 模式，支持多线程[35]。在实验中，测试脚本在同一台宿主机上创建两个容器，分别使用 Linux Bridge/veth pair（原 Docker 网络）和 macvlan（Flat 网络）连接它们，一个容器运行 iperf 服务器，另一个容器运行 iperf 客户端，分别创建 1、2、4、8、16 个线程测试网络带宽。测试结果如表 6.1 所示。

表 6.1 veth 和 macvlan 性能测试表

连接设备及类型	线程数	10s 传输数据量（GBytes）	带宽（Gbits/s）
Bridge/veth	1	52.1	44.7
Bridge/veth	2	86.2	74.1
Bridge/veth	4	74.8	64.2
Bridge/veth	8	80.1	68.8
Bridge/veth	16	65.0	55.7
macvlan	1	66.9	57.5
macvlan	2	92.1	79.1

续表 6.1

连接设备及类型	线程数	10s 传输数据量（GBytes）	带宽（Gbits/s）
macvlan	4	83.3	71.5
macvlan	8	75.0	64.4
macvlan	16	77.6	66.5

可以看到 macvlan 在同一台机器上连接 Docker 容器的效率比 Linux Bridge/veth pair 设备高很多，带宽提升有 10%~20%左右。

6.2.2 容器跨主机数据传输性能各系统方案对比

Docker 原生网络基本不支持容器跨主机通信，若用端口映射的方法将容器端口暴露在主机上，则可以通过访问主机的端口访问容器，这勉强算一种跨主机通信，但是需要 SNAT（容器流量出宿主机时）和 DNAT（到达目的主机时）双层转换，效率肯定有损失，而且不是真正的跨主机通信。业界普遍使用的基于 Open vSwitch 的 overlay 网络虽然可以实现真正的跨宿主机通信，但是需要使用隧道技术对数据包进行封装和解封装，效率也会有损失。而基于 macvlan 的 Flat 网络，则既实现了真正的跨主机通信，又没有对数据包进行额外的操作。

在实验中，测试脚本在两台宿主机上创建了两个容器，分别用作 iperf 客户端与服务器。然后用以上三种方式连接它们（overlay 的网络拓扑参照图 2.5），并对这三种跨主机通信方式进行了性能测试，测试结果如表 6.2 所示：

表 6.2 跨主机通信性能测试表

连接方式	线程数	10s 数据传输量（MBytes）	带宽（Mbits/s）
NAT	1	113.0	94.6
NAT	2	114.0	95.4
NAT	4	116.0	96.3
NAT	8	119.0	96.6

续表 6.2

连接方式	线程数	10s 数据传输量（MBytes）	带宽（Mbits/s）
NAT	16	125.0	97.8
OVS/GRE	1	110.3	91.5
OVS/GRE	2	112.5	93.3
OVS/GRE	4	114.0	94.6
OVS/GRE	8	117.9	97.8
OVS/GRE	16	120.6	100.0
macvlan	1	116.0	96.3
macvlan	2	118.0	96.6
macvlan	4	124.0	100
macvlan	8	125.0	101
macvlan	16	126.0	102

从上表可以看出，Docker 容器跨主机通信比同主机上的效率要低很多，因为跨主机要受物理交换机性能的影响，而同主机则主要由主机的性能和虚拟设备的实现方法决定。上述三种跨主机方案中，可以看到基于 macvlan 的方案最优，其次是 NAT 的方法，而 Open vSwitch/GRE 的方案则性能损失最多，这也是本系统舍弃 Open vSwitch 而使用 macvlan 的重要原因。

6.3 本章小结

本章首先对 macvlan 网络系统的主要功能模块做了展示，各个模块都能够满足需求。同时，可以看到，本系统的运行不需要依赖 OVS 等第三方软件，且非常容易与 Docker 集成。然后，对本系统进行了性能测试，本系统在数据传输性能上比 Docker 本身的网络以及基于 OVS 的 overlay 网络都要好。

第7章 总结与展望

7.1 工作总结

本文针对 Docker 容器目前在网络方面所存在的不足，提出了使用 macvlan 设备来改进 Docker 的网络功能，并实现了基于 macvlan 的 Docker 网络系统。该系统针对不同的使用场景，设计实现了 Flat、DHCP、VLAN 三种网络模式，并为了方便与 Docker daemon 整合，实现了 macvlan daemon，可以完美处理 Docker 原生的 HTTP 请求。

Flat 网络有效地解决了 Docker 容器跨主机通信的问题，同时相比于业界使用的基于 OVS 的 overlay 网络方案，Flat 网络在性能上有较好的提升。DHCP 网络解决了容器从 DHCP 服务器获取 IP 地址的需求。VLAN 网络主要解决的是多租户环境下的安全隔离问题，通过在 Flat 网络的基础上，对数据打上不同 Tag 来区别不同的租户，以达到隔离的目的。同时，这样的 VLAN 网络也继承了 Flat 网络的跨主机通信特点。在企业应用中，二层隔离的需求还是比较普遍的。本系统没有采用普遍的 Open vSwitch 方案，而是使用了 Linux 内核中的 VLAN 模块，这样可以减少繁琐的外部依赖。

除了上面三种自定义容器网络的方式外。本系统还实现了特定容器流量限制、服务发现等特性。QoS 的实现是通过使用 Linux 的 TC 模块在 macvlan 设备上建立队列规则完成的。而服务发现则依赖于键值存储系统 etcd 为容器提供的注册、查询功能。同时，etcd 集群还可以保存容器的网络配置信息，使得容器在重启后能够获得之前的配置信息。

最后实现的 macvlan daemon 作为 Docker client 和 Docker daemon 的中间人，可以接受并处理 Docker client 的 HTTP 请求，并根据请求的内容调用上面介绍的几种网络模式自定义 Docker 的网络。其本质是一个 HTTP 代理，在 Docker client 和 Docker daemon 之间转发请求和响应，并从中执行一些额外的操作。

总体上，该系统很好的弥补了目前 Docker 网络存在的一些缺陷，满足了更高

的使用需求。在数据传输性能上，也比 Docker 原生网络以及现有的基于 OVS 的 overlay 网络方案有提升。同时，系统还具有无软件包依赖、轻量级、易于集成等优点。

7.2 未来工作展望

尽管基于 macvlan 的网络系统已经基本满足使用需求，但仍有不足需要改进。在与 Docker 官方对接方面，Docker 目前开发的 libnetwork 库提供了 Driver 的 API，用以实现第三方的网络库，目前由于 libnetwork 不够完善，本系统没有实现其 API，在后面的开发中可能会对接其 API。在跨主机通信方面，本系统只支持同一个网段的跨主机通信，对于跨不同网段甚至跨数据中心的需求，还不能满足。在网络 QoS 方面，本系统只支持对特定容器进行简单的流量限制，还有待加入更多、更细致的规则，丰富 QoS 的功能。此外，在扩展性和对 SDN 的支持方面，本系统还有改进空间。

参考文献

[1] 数人科技. 集群管理与公司管理对互联网公司一样重要[EB/OL]. http://blog.dataman-inc.com/ji-qun-guan-li-zhong-yao/

[2] 雷万云. 云计算: 技术、平台及应用案例[M]. 清华大学出版社, 2011

[3] Mell P, Grance T. The NIST definition of cloud computing[J], 2011

[4] 肖德时. Docker 核心技术预览[EB/OL]. http://blog.dataman-inc.com/untitled-2/

[5] Docker[EB/OL]. https://www.docker.com/

[6] Pahl C. Containerization and the PaaS cloud[J]. IEEE Cloud Computing, 2015 (3): 24-31

[7] Marmol V, Jnagal R, Hockin T. Networking in Containers and Container Clusters[J], 2015. 2

[8] Emmerich P, Raumer D, Wohlfart F, et al. Performance characteristics of virtual switching[C]. Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on. IEEE, 2014: 120-125

[9] Lantz B, Heller B, McKeown N. A network in a laptop: rapid prototyping for software-defined networks[C]. Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM, 2010: 19

[10] Turnbull J. The Docker Book[J], 2014. 3

[11] Docker Advanced networking[EB/OL]. https://docs.docker.com/v1.6/articles/networking/

[12] Namiot D, Sneps-Sneppe M. On micro-services architecture[J]. International Journal of Open Information Technologies, 2014, 2(9): 24-27

[13] Amies A，Wu C F，Wang G C，Criveti M. Networking on the cloud[J]. IBM developerWorks, 2012. 6

[14] Calheiros R N, Ranjan R, Buyya R. Virtual machine provisioning based on analytical performance and QoS in cloud computing environments[C].

Parallel Processing (ICPP), 2011 International Conference on. IEEE, 2011: 295-304

[15] Delchev I. Linux Traffic Control[C]. Networks and Distributed Systems Seminar, International University Bremen, Spring, 2006

[16] Guttman E. Service location protocol: Automatic discovery of IP network services[J]. Internet Computing, IEEE, 1999, 3(4): 71-80

[17] 浙江大学 SEL 实验室. Docker 容器与容器云[M]. 人民邮电出版社, 2015

[18] 孙宏亮. Docker 源码分析（一）: Docker 架构[EB/OL]. http://www.infoq.com/cn/articles/docker-source-code-analysis-part1/

[19] 网络虚拟化技术（二）: TUN/TAP MACVLAN MACVTAP[EB/OL]. https://blog.kghost.info/2013/03/27/linux-network-tun/

[20] Kashyap V, Bergman A, Berger S, et al. Automating Virtual Machine Network Profiles[C]. Linux Symposium, 2010: 147

[21] Seravo. Virtualized bridged networking with MacVTap[EB/OL]. https://seravo.fi/2012/virtualized-bridged-networking-with-macvtap

[22] Github. Jpetazzo/pipework[EB/OL]. https://github.com/jpetazzo/pipework

[23] Github. Socketplane/socketplane[EB/OL]. https://github.com/socketplane/socketplane

[24] Open vSwitch[EB/OL]. http://openvswitch.org/

[25] Github. Coreos/flannel[EB/OL]. https://github.com/coreos/flannel

[26] Github. Docker/libnetwork[EB/OL]. https://github.com/docker/libnetwork

[27] Pfaff B, Pettit J, Koponen T, et al. The design and implementation of Open vSwitch[C]. 12th USENIX Symposium on Networked Systems Design and Implementation, 2015

[28] Andrew S. Tanenbaum, David J. Wetherall. 计算机网络（第 5 版）[M].清华大学出版社, 2012

[29] Mahalingam M, Dutt D, Duda K, et al. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks[R], 2014

[30] Garg P, Wang Y S. NVGRE: Network Virtualization using Generic Routing

Encapsulation[J], 2014

[31] Dockone. Docker 网络管理的未来[EB/OL]. http://dockone.io/article/50

[32] Bomb250. Linux 实现的 IEEE 802.1Q VLAN[EB/OL]. http://blog.csdn.net/dog250/article/details/7354590

[33] Justin Ellingwood. The Docker Ecosystem: An Introduction to Common Components[EB/OL]. https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components

[34] David Gourley, Brian Totty, Marjorie Sayer, Sailu Reddy, Anshu Aggarwal. HTTP 权威指南[M]. 人民邮电出版社, 2012

[35] Open Cloud Blog. Switching Performance – Connecting Linux network namespaces[EB/OL]. http://www.opencloudblog.com/?p=96

攻读硕士学位期间主要的研究成果

致谢

历时两个多月，毕业论文终于将近完成。这也意味着研究生生涯即将结束。回顾整个过程，我经历过很多困难，也曾经迷茫、彷徨过，但在导师、同学、家人、朋友的帮助和关心下，最终能够克服困难，顺利地修完学分、做完项目、完成毕业论文的编写。在此，要对他们致以最诚挚的谢意。

首先感谢黄忠东老师。作为我的研究生导师，黄老师从我刚入学时课程、方向的选择上，到最后毕业论文的完成，都给予了最专业的知道和帮助。整个研究生期间，每当学业上碰到难题和困惑时，黄老师总能耐心地教导，并给出建设性的意见。黄老师渊博的学识、科学严谨的治学态度都深深地影响了我。除此之外，在生活上，黄老师也给予了莫大的帮助。

感谢实验室的丁轶群老师、张磊学长、苌程老师。在研究上期间，他们帮助我快速上手了实验室的项目，并在一些具体的技术问题上，能帮忙分析和解决。毕业设计的实验方面，也给了很大的支持。同时也非常感谢实验室的孙健波、张建霞、房伟利、刘敏献、周宇哲等同学在学习和生活上的帮助。

最后感谢我的父母，感谢你们一直以来对我的培养和支持。在此，祝你们身体健康，开心幸福。

2016 年 1 月 10 日