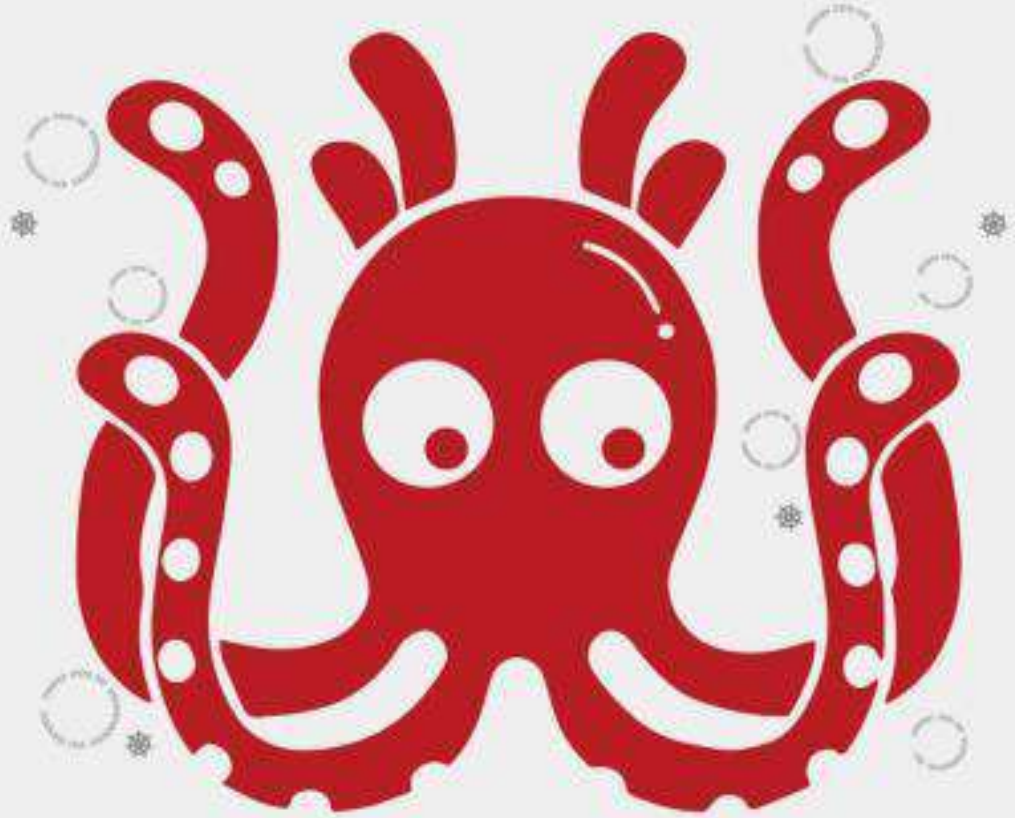
Broadview®
www.broadview.com.cn



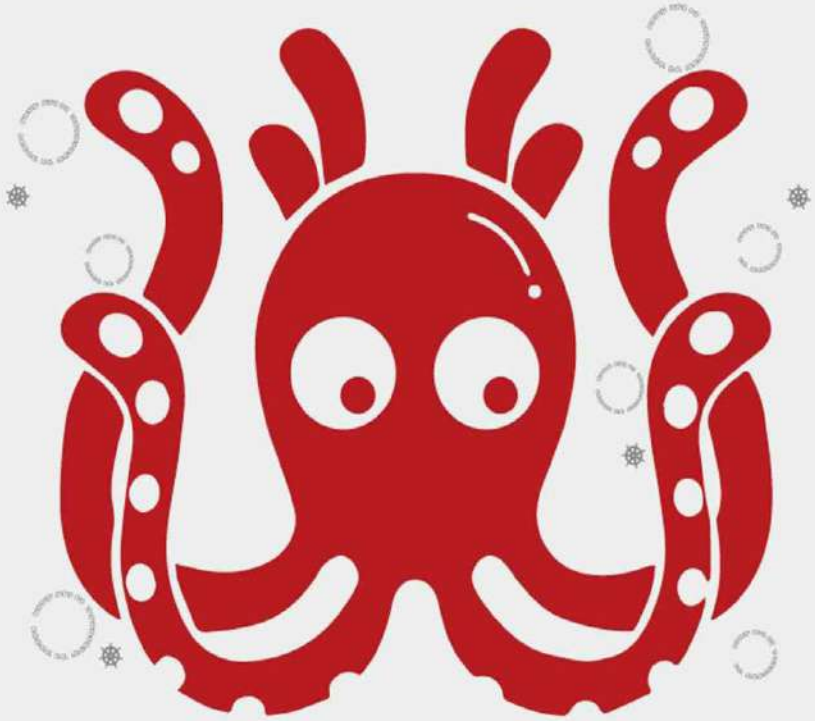
Kubernetes 网络权威指南

基础、原理与实践

杜军◎著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn



Kubernetes 网络权威指南

基础、原理与实践

杜军◎著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

杜　军

浙大SEL实验室硕士，曾任华为云架构师、咨询组专家，国内最早的容器技术布道师。开源社区资深贡献者与维护者（GitHub: m1093782566），CNCF TOC Contributor（全球1/70），Kubernetes和Istio双科maintainer，对云计算技术演进与内在驱动力有深刻见解，主要研究方向为容器、微服务、DevOps、边缘计算。主导了Kubernetes多个核心特性的设计与开发，如Kube-proxy IPVS模式、CNI带宽控制、拓扑感知的服务路由等。著有《Docker容器与容器云》《云原生分布式存储基石：etcd深入解析》等书籍，是KubeCon、CloudNativeCon、LinuxCon、Open Source Summit等峰会演讲嘉宾，做过华为云原生技术直播和Kubernetes CKA考试培训。致力于构建生产级可用的容器服务平台，同时积极把技术回馈开源社区。目前，负责一家上市互联网公司的云原生落地和企业上云工作。

Kubernetes 网络权威指南

基础、原理与实践

杜军◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书是容器与Kubernetes网络的基础和进阶书籍，旨在让更多人了解和学习云原生时代的底层网络模型与实现机制，指导企业在落地云原生时的网络方案选型。

全书共6章，第1章Linux网络虚拟化将支撑容器网络的内核技术娓娓道来，第2章简单介绍了Docker网络模型，第3章介绍Kubernetes网络原理与实践，第4章剖析了Kubernetes网络实现机制，第5章详解了业界主流的Kubernetes网络插件生态，第6章重点解析了Istio网络流量管控的背后机制。

本书适合作为高等院校计算机相关专业云计算课程的参考资料，也适合云计算从业者，特别是希望对云原生网络技术有较深入了解并希望将其应用到日常工作中的所有读者阅读。

图书在版编目（**CIP**）数据

Kubernetes网络权威指南：基础、原理与实践/杜军著.—北京：电子工业出版社，2019.10

ISBN 978-7-121-37339-8

I.①K… II.①杜… III.①Linux操作系统-程序设计-指南 IV.①TP316.85-62

中国版本图书馆CIP数据核字（2019）第193119号

责任编辑：郑柳洁

印　　刷：

装　　订：

出版发行：电子工业出版社

北京市海淀区万寿路173信箱　　邮编：100036

开　　本：787×980　1/16　印张：21.75　字数：455千字

版　　次：2019年10月第1版

印　　次：2019年10月第1次印刷

定　　价：89.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

自序

缘起

这些年来，我观察到每次Linux世界的重大技术创新都发源于内核，经过一层层面向用户的抽象和封装，演化出应用层的森罗万象。正所谓万变不离其宗！

我自认为是个“old　school”（老派）的人，坚信维持这个世界运转的、最本质的那部分东西是不会轻易改变的。尽管新技术“乱花渐欲迷人眼”，但经历过时间锤炼的实用技术和工具总是历久弥新的。一个很好的例子便是传统的网络虚拟化和BGP，它们就是在容器这个新瓶子里焕发第二春的。因此，当有时髦的新技术出现时，我的第一反应是这些新技术底层是怎么实现的，对那些看起来酷炫的功能反倒没有兴趣。正如OpenStack兴起的那些年，当大家都在谈论nova、neutron这类调度虚拟机和网络的组件时，我默默地翻读了内核虚拟机（KVM）的源码。至今，我对那种奇妙的感觉记忆犹新，恰如一个发烧级摄影爱好者走进暗房，欣喜若狂地亲自手洗一张可触摸的胶片影像。当我读懂了KVM时，再回过头来看OpenStack便有了一种“会当凌绝顶，一览众山小”的豁然贯通之感。

工作之余让心静下来，细细品味，认真思考技术的本质——相信这是所有有激情、有梦想的工程师的共同追求。虽然在软件版本快速迭代的高压面前，这种良好的追求有时也会变成一种奢求，但是我坚信工程师不能只当一个使用者，而一定要理解当前正在使用的技术的底层实现机制。因此，在我的《云原生分布式存储基石：etcd深入解析》一书中，开篇只字未提全书主角etcd，而耗费将近80页的笔墨，从分布式系统的基本理论一直讲到一致性协议Raft。尽管内容看似与这个高速发展、追求快节奏的社会“格格不入”，但我仍希望能够通过出版技术书这种本身就慢节奏且带仪式感的行为沉淀自己的思考。如果能够跟有缘的读者碰撞出思想的火花，则将是我人生的一大幸事！

我为什么写这本书

云计算的世界里，计算最基础，存储最重要，网络最复杂。在Kubernetes已经成为云原生代名词的今天，市面上介绍Kubernetes的书籍已经很多，然而限于篇幅或术业有专攻等诸多主客观因素，不少书籍对Kubernetes网络部分的讲解只是蜻蜓点水，甚至有些还存在专业性的错误。我经常在一些学习Kubernetes的论坛和群里看到有用户抱怨：传统网络架构都还没搞明白，又要理解容器网络。容器网络领域不但存在大量的术语，而

且理解具体的方案需要不少前置知识，这无形中增加了学习的难度。计算机网络是我在大学里最喜欢，也是最擅长的一门课程。在我看来，计算机网络趣味性强，而且对逻辑性和动手能力要求较高。看到整个云原生网络领域正发生着激动人心的技术变革，意义不亚于上一次SDN兴起带来的冲击，我感觉这对传统网络工程师来说会是一次自我升级转型的契机。于是，我萌生了专门为Docker、Kubernetes的用户，以及传统网络工程师撰写一本云原生网络书籍的想法，破除他们学习过程中“不识庐山真面目，只缘身在此山中”的无力感。

关于本书

虽然书名是《Kubernetes网络权威指南：基础、原理与实践》，但全书内容并不局限于Kubernetes。我对本书的定位是云原生领域的网络权威指南，企业落地方案的选型参考。按照我“old school”的思路，本书特别注重提供理解容器网络所必需的基础知识，会由浅入深地从架构、使用、实现原理等多方面展开，试图为读者呈现整个云原生网络的知识体系。

全书的脉络是：以Linux网络虚拟化基础作为“暖场嘉宾”，以Docker原生的容器网络“承前启后”，随后是主角Kubernetes网络“粉墨登场”，在各类CNI插件“沙场点兵”过后，以代表容器下半场的服务网格Istio“谢幕”。

王安石在登上飞来峰后曾吟下“不畏浮云遮望眼，自缘身在最高层”这样的千古佳句。希望本书能够成为云计算2.0时代的弄潮儿们叩开网络大门的敲门砖，在解决各类场景下错综复杂的问题时能够做到“口中有粮、心中不慌”。不论是定位疑难杂症，还是技术选型，抑或是定制化开发都能轻松驾驭！

“人生不止眼前的苟且，还有诗和远方的田野”，愿更多的同路人加入。

杜军

2019年8月于厦门鼓浪屿

【读者服务】

扫码回复：37339

·获取免费增值资源

·获取精选书单推荐

·加入读者交流群，与更多读者互动

目　录

第1章　夯实基础：Linux网络虚拟化

1.1　网络虚拟化基石：network namespace

在本书的开篇就介绍network namespace是因为它足够重要，毫不夸张地说，它是整个Linux网络虚拟化技术的基石。因此，我们将花较多篇幅介绍。

顾名思义，Linux的namespace（名字空间）的作用就是“隔离内核资源”。在Linux的世界里，文件系统挂载点、主机名、POSIX进程间通信消息队列、进程PID数字空间、IP地址、user ID数字空间等全局系统资源被namespace分割，装到一个个抽象的独立空间里。而隔离上述系统资源的namespace分别是Mount namespace、UTS namespace、IPC namespace、PID namespace、network namespace和user namespace。对进程来说，要想使用namespace里面的资源，首先要“进入”（具体操作方法，下文会介绍）到这个namespace，而且还无法跨namespace访问资源。Linux的namespace给里面的进程造成了两个错觉：

（1）它是系统里唯一的进程。

（2）它独享系统的所有资源。

默认情况下，Linux进程处在和宿主机相同的namespace，即初始的根namespace里，默认享有全局系统资源。

Linux内核自2.4.19版本接纳第一个namespace：Mount namespace（用于隔离文件系统挂载点）起，到3.8版本的user namespace（用于隔离用户权限），总共实现了上文提到的6种不同类型的namespace。尽管Linux的namespace隔离技术很早便存在于内核中，而且它就是为Linux的容器技术而设计的，但它一直鲜为人知。直到Docker引领的容器技术革命爆发，它才进入普罗大众的视线——Docker容器作为一项轻量级的虚拟化技术，它的隔离能力来自Linux内核的namespace技术。

说到network namespace，它在Linux内核2.6版本引入，作用是隔离Linux系统的设备，以及IP地址、端口、路由表、防火墙规则等网络资源。因此，每个网络namespace里都有自己的网络设备（如IP地址、路由表、端口范围、/proc/net目录等）。从网络的角度看，network namespace使得容器非常有用，一个直观的例子就是：由于每个容器都有自己的（虚拟）网络设备，并且容器里的进程可以放心地绑定在端口上而不必担心冲突，这就使得在一个主机上同时运行多个监听80端口的Web服务器变为可能，如图1-1所示。

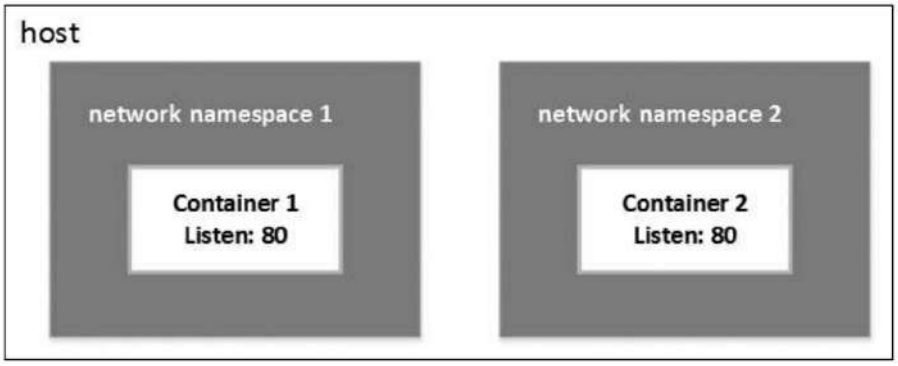


图1-1　network namespace示意图

1.1.1　初识network namespace

和其他namespace一样，network　namespace可以通过系统调用来创建，我们可以调用Linux的clone（）（其实是UNIX系统调用fork（）的延伸）API创建一个通用的namespace，然后传入CLONE_NEWNET参数表面创建一个network　namespace。高阶读者可以参考下文的C代码创建一个network namespace。与其他namespace需要读者自己写C语言代码调用系统API才能创建不同，network　namespace的增删改查功能已经集成到Linux的ip工具的netns子命令中，因此大大降低了初学者的体验门槛。下面先介绍几条简单的网络namespace管理的命令。

创建一个名为netns1的network namespace可以使用以下命令：

```
# ip netns add netns1
```

当ip命令创建了一个network　namespace时，系统会在/var/run/netns路径下面生成一个挂载点。挂载点的作用一方面是方便对namespace的管理，另一方面是使namespace即使没有进程运行也能继续存在。

一个network　namespace被创建出来后，可以使用ip　netns　exec命令进入，做一些网络查询/配置的工作。

```
# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

如上所示，就是进入netns1这个network　namespace查询网卡信息的命令。目前，我们没有任何配置，因此只有一块系统默认的本地回环设备lo。

想查看系统中有哪些network namespace，可以使用以下命令：

```
# ip netns list
nstns1
```

想删除network namespace，可以通过以下命令实现：

```
# ip netns delete netns1
```

注意，上面这条命令实际上并没有删除netns1这个network namespace，它只是移除了这个network namespace对应的挂载点（下文会解释）。只要里面还有进程运行着，network namespace便会一直存在。

1.1.2 配置network namespace

当namespace里面的进程涉及网络通信时，namespace里面的（虚拟）网络设备就必不可少了。通过上文的阅读我们已经知道，一个全新的network namespace会附带创建一个本地回环地址。除此之外，没有任何其他的网络设备。而且，细心的读者应该已经发现，network namespace自带的lo设备状态还是DOWN的，因此，当尝试访问本地回环地址时，网络也是不通的。下面的小测试就说明了这一点。

```
## 进入netns1这个network namespace, ping 127.0.0.1
# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
```

在我们的例子中，如果想访问本地回环地址，首先需要进入netns1这个network namespace，把设备状态设置成UP。

```
# ip netns exec netns1 ip link set dev lo up
```

然后，尝试ping 127.0.0.1，发现能够ping通。

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.012 ms
```

但是，仅有一个本地回环设备是没法与外界通信的。如果我们想与外界（比如主机上的网卡）进行通信，就需要在namespace里再创建一对虚拟的以太网卡，即所谓的veth pair。顾名思义，veth pair总是成对出现且相互连

接，它就像Linux的双向管道（pipe），报文从veth pair一端进去就会由另一端收到。关于veth pair更详细的介绍，参见1.2节，本节不再赘述。

下面的命令将创建一对虚拟以太网卡，然后把veth pair的一端放到netns1 network namespace。

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns netns1
```

如上所示，我们创建了veth0和veth1这么一对虚拟以太网卡。在默认情况下，它们都在主机的根network namespce中，将其中一块虚拟网卡veth1通过ip link set命令移动到netns1 network namespace。那么，veth0和veth1之间能直接通信吗？还不能，因为这两块网卡刚创建出来还都是DOWN状态，需要手动把状态设置成UP。这个步骤的操作和上文对lo网卡的操作类似，只是多了一步绑定IP地址，如下所示：

```
# ip netns exec netns1 ifconfig veth1 10.1.1.1/24 up
# ifconfig veth0 10.1.1.2/24 up
```

上面两条命令首先进入netns1这个network namespace，为veth1绑定IP地址10.1.1.1/24，并把网卡的状态设置成UP，而仍在主机根network namespace中的网卡veth0被我们绑定了IP地址10.1.1.2/24。这样一来，我们就可以ping通veth pair的任意一头了。例如，在主机上ping 10.1.1.1（netns1 network namespace里的网卡），如下所示：

```
# ping 10.1.1.1
PING 10.1.1.1 (10.1.1.1) 56(84) bytes of data.
64 bytes from 10.1.1.1: icmp_seq=1 ttl=64 time=0.022 ms
...
```

同理，我们可以进入netns1 network namespace去ping主机上的虚拟网卡，如下所示：

```
# ip netns exec netns1 ping 10.1.1.2
PING 10.1.1.2 (10.1.1.2) 56(84) bytes of data.
64 bytes from 10.1.1.2: icmp_seq=1 ttl=64 time=0.014 ms
...
```

另外，不同network namespace之间的路由表和防火墙规则等也是隔离

的，因此我们刚刚创建的netns1 network namespace没法和主机共享路由表和防火墙，这一点通过下面的测试就能说明。

```
# ip netns exec netns1 route
# ip netns exec netns1 iptables -L
```

如上所示，我们进入netns1 network namespace，分别输入route和iptables-L命令，期望查询路由表和iptables规则，却发现空空如也。这意味着从netns1 network namespace发包到因特网也是徒劳的，因为网络还不通！不信读者可以自行尝试。想连接因特网，有若干解决方法。例如，可以在主机的根network namespace创建一个Linux网桥并绑定veth pair的一端到网桥上；也可以通过适当的NAT（网络地址转换）规则并辅以Linux的IP转发功能（配置net.ipv4.ip_forward=1）。关于Linux网桥和NAT，下文会有详细介绍，这里不再赘述。

需要注意的是，用户可以随意将虚拟网络设备分配到自定义的network namespace里，而连接真实硬件的物理设备则只能放在系统的根network namesapce中。并且，任何一个网络设备最多只能存在于一个network namespace中。

进程可以通过Linux系统调用clone（）、unshare（）和setns进入network namespace，下面会有代码示例。非root进程被分配到network namespace后只能访问和配置已经存在于该network namespace的设备。当然，root进程可以在network namespace里创建新的网络设备。除此之外，network namespace里的root进程还能把本network namespace的虚拟网络设备分配到其他network namespace——这个操作路径可以从主机的根network namespace到用户自定义network namespace，反之亦可。请看下面这条命令：

```
# ip netns exec netns1 ip link set veth1 netns 1
```

该怎么理解上面这条看似有点复杂的命令呢？分解成两部分：

（1）ip netns exec netns1进入netns1 network namespace。

（2）ip link set veth1 netns 1把netns1 network namespace下的veth1网卡挪到PID为1的进程（即init进程）所在的network namespace。

通常，init进程都在主机的根network namespace下运行，因此上面这条命令其实就是把veth1从netns1 network namespace移动到系统根network namespace。有两种途径索引network namespace：名字（例如netns1）或者属于该namespace的进程PID，上文中用的就是后者。

对namespace的root用户而言，他们都可以把其namespace里的虚拟网络设备移动到其他network namespace，甚至包括主机根network namespace！这就带来了潜在的安全风险。如果用户希望屏蔽这一行为，则需要结合PID namespace和Mount namespace的隔离特性做到network namespace之间的完全不可达，感兴趣的读者可以自行查阅相关资料。

1.1.3 network namespace API的使用

前文已经提到network namespace作为Linux六大namespace之一，其API涉及三个Linux系统调用：clone、unshare和setns，以及一些系统/proc目录下的文件。本节我们将通过几个C语言程序的例子介绍network namespace API的使用方法。clone（）、unshare（）和setns（）系统调用会使用CLONE_NEW*常量来区别要操作的namespace类型。CLONE_NEW*常量一共有6个：CLONE_NEWIPC、CLONE_NEWNS、CLONE_NEWNET、CLONE_NEWPID、CLONE_NEW USER和CLONE_NEWUTS，分别代表6个不同的namespace类型。细心的读者通过名字应该能观察出来，CLONE_NEWNS指代的是network namespace。

1.创建**namespace**的黑科技：**clone**系统调用

用户可以使用clone（）系统调用创建一个namespace。但当知道clone（）系统调用是用来创建一个新的进程时，请不要感到惊讶，让我们看下面的例子。

```
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);
```

clone（）的调用方式如上所示，它其实就是我们熟悉的UNIX/Linux系统调用fork（）的延伸，我们可以通过其flags参数（标志位）控制特定的功能。clone（）总共有二十多种CLONE_*标志位用来控制clone（克隆）进程时的行为。例如，是否与父进程共享虚拟内存、打开的文件描述符、信号处理等。

只要在clone（）设置了其中一个标志位CLONE_NEW，系统就会创建一个新的对应类型的namespace及一个新的进程，并且会把这个进程放到这个新创建的namespace中。通过|（位或）操作，我们可以实现clone（）同时指定多个CLONE_NEW标志位。看到这里，读者是否已经恍然大悟了？原来clone（）创建namespace是这个道理啊。

再来解释clone（）的几个参数的含义，从左到右分别是：

·函数指针child_func，指定一个由新进程执行的函数。当这个函数返回

时，子进程终止。该函数返回一个整数，表示子进程的退出代码；

·指针child_stack传入子进程使用的栈空间，也就是把用户态堆栈指针赋给子进程的esp寄存器。调用clone（）的进程应该总是为子进程分配新的堆栈；

·int类型的flags参数表示CLONE_*标志位（可以多个）；

·args表示用户的自定义参数。

最后提一下权限/安全问题，大部分Linux namespace（user namespace除外）的创建都需要系统特权（capability），不一定是完整的root权限，但需要拥有CAP_SYS_ADMIN权限集来执行必要的系统调用。

注：Linux的特权是将root的权限划分为各个小部分，使得一个进程只需要被授予刚刚好的权限来执行特定的任务。如果这些特权足够小且选择得恰到好处，那么即使一个特权进程受损（比如缓冲区溢出），它所造成的危害也会受限于它所拥有的特权。例如，CAP_KILL允许进程向任意的进程发送信号，而CAP_SYS_TIME允许进程设置系统的时钟。

2.维持**namespace**存在：**/proc/PID/ns**目录的奥秘

每个Linux进程都拥有一个属于自己的/proc/PID/ns，这个目录下的每个文件都代表一个类型的namespace。

在Linux内核3.8版本以前，/proc/PID/ns目录下的文件都是硬链接（hard link），而且只有ipc、net和uts这三个文件。从Linux内核3.8版本开始，每个文件都是一个特殊的符号链接文件，如上文提到的那样，这些文件提供了操作进程关联namespace的一种方式。先来看一眼这些符号链接文件都长什么样。

```
$ ls -l /proc/$$/ns          # 注：$$是shell进程的PID
total 0
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 ipc -> ipc:[4026531839]
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 mnt -> mnt:[4026531840]
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 net -> net:[4026531956] # 代表network namespace
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 pid -> pid:[4026531836]
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 user -> user:[4026531837]
lrwxrwxrwx. 1 dj dj 0 Jan  4 04:12 uts -> uts:[4026531838]
```

我们看到的符号链接的其中一个用途是确定某两个进程是否属于同一个namespace。如果两个进程在同一个namespace中，那么这两个进程/proc/PID/ns目录下对应符号链接文件的inode数字（即上文例子中［］内的数字，例如4026531839，也可以通过stat（）系统调用获取返回结构体的st_ino字段）会是一样的。

除此之外，/proc/PID/ns目录下的文件还有一个作用——当我们打开这些文件时，只要文件描述符保持open状态，对应的namespace就会一直存在，哪怕这个namespace里的所有进程都终止运行了。听起来有点拗口，有什么意义呢？之前版本的Linux内核，要想保持namespace存在，需要在namespace里放一个进程（当然，不一定是运行中的），这种做法在一些场景下有些笨重（虽然Kubernetes就是这么做的）。因此，Linux内核提供的黑科技允许：只要打开文件描述符，不需要进程存在也能保持namespace存在！怎么操作？请看下面的命令：

```
# touch /my/net # 新建一个文件
# mount --bind /proc/$$/ns/net /my/net
```

如上所示，把/proc/PID/ns目录下的文件挂载起来就能起到打开文件描述符的作用，而且这个network namespace会一直存在，直到/proc/self/ns/net被卸载。

3.往namespace里添加进程：setns系统调用

我们已经使用一些黑科技使得namespace即使没有进程在其中也能保持开放。接下来，我们就要往这个namespace里“扔”进程，Linux系统调用setns（）就是用来做这个工作的，其主要功能就是把一个进程加入一个已经存在的namespace中。setns（）的定义如下所示：

```
int setns(int fd, int nstype);
```

其中：

·参数fd表示进程待加入的namespace对应的文件描述符。我们已经知道，它是一个指向/proc/PID/ns目录下符号链接的文件描述符；

·int类型参数nstype的作用是让调用者检查第一个参数fd指向的namespace类型是否符合我们的实际要求，0表示不检查。

还记得前文我们用的ip netns exec子命令吗？我们用这条子命令轻松进入一个network namespace，然后执行一些操作。我们通过setns（）和execve（）系列函数就能组合出一个更加通用的小工具：进入一个指定namespace，然后在里面执行一条shell命令。

注：execve（）系列函数可以用来执行用户自定义的命令，一个常用的方法是调用/bin/sh运行一个shell。

一个简单的C语言样例如下所示：

```
fd = open(argv[1], O_RDONLY);   // 获取namespace的文件描述符
setns(fd, 0);                   // 加入namespace
execvp(argv[2], &argv[2]);      // 执行用户自定义程序
```

让我们编译上面的代码段，假设编译后的二进制文件是enterns，那么执行以下命令：

```
# ./enterns /my/net /bin/sh
```

/my/net是上文我们新建的network namespace符号链接的挂载文件。以上命令执行完后，我们就进入一个新的network namespace，并且可以在里面执行shell命令。这种方式广泛应用于Docker和Kubernetes中。

注：在Linux内核3.8版本之前，setns（）还不能用于加入Mount namespace、PID namespace和user namespace，但从3.8版本以后，支持所有类型的namespace。

4.帮助进程逃离**namespace**：**unshare**系统调用

与namespace相关的最后一个系统调用是unshare（），用于帮助进程“逃离”namespace。unshare（）系统调用的函数声明如下：

```
int unshare(int flags);
```

细心的读者想必已经发现，unshare（）也有一个int类型的flags参数，这个flags的用法和clone（）的flags是否一样呢？答案是肯定的。

unshare（）系统调用的工作机制是：先通过指定的flags（即CLONE_NEW*bit位的组合）创建相应的namespace，再把这个进程挪到这些新创建的namespace中，于是也就完成了进程从原先namespace的撤离。unshare（）提供的功能其实很像clone（），区别在于unshare（）作用在一个已存在的进程上，而clone（）会创建一个新的进程。

那么，unshare（）系统调用具体有什么应用场景呢？大部分Linux发行版自带的unshare命令就是基于unshare（）系统调用的，它的作用就是在当前shell所在的namespace外执行一条命令。unshare命令的用法如下所示：

```
# unshare [options] program [arguments]
```

Linux会为需要执行的命令（在上面的例子中，即program）启动一个新进程，然后在另外一个namespace中执行操作，这样就可以起到执行结果和原（父）进程隔离的效果。

5.一个完整的例子

一个使用Linux network namespace相关系统调用的程序样例如下。由于代码比较简单，而且几个关键点都做了注释，笔者就不逐行解释了。

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <stdio.h>
#include <sched.h>
```

```
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

#define STACK_SIZE (1024 * 1024)

// sync primitive
int checkpoint[2];

static char child_stack[STACK_SIZE];
char* const child_args[] = {
  "/bin/bash",
  NULL
};

int child_main(void* arg)
{
  char c;

  // init sync primitive
  close(checkpoint[1]);

  // setup hostname
  printf(" - [%5d] World !\n", getpid());
  sethostname("In Namespace", 12);

  // remount "/proc" to get accurate "top" && "ps" output
  mount("proc", "/proc", "proc", 0, NULL);

  // wait for network setup in parent
  read(checkpoint[0], &c, 1);

  // setup network
  system("ip link set lo up");
  system("ip link set veth1 up");
  system("ip addr add 169.254.1.2/30 dev veth1");
```

```
  execv(child_args[0], child_args);
  printf("Ooops\n");
  return 1;
}

int main()
{
  // init sync primitive
  pipe(checkpoint);

  printf(" - [%5d] Hello ?\n", getpid());

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
      CLONE_NEWUTS | CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET |
SIGCHLD, NULL);

  // further init: create a veth pair
  char* cmd;
  asprintf(&cmd, "ip link set veth1 netns %d", child_pid);
  system("ip link add veth0 type veth peer name veth1");
  system(cmd);
  system("ip link set veth0 up");
  system("ip addr add 169.254.1.1/30 dev veth0");
  free(cmd);

  // signal "done"
  close(checkpoint[1]);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```

以上程序的大意是，在一个新的network namespace中初始化一个网络设备veth1，绑定IP地址169.254.1.1。当我们使用如下命令编译并运行以上程序时，会得到如下输出：

```
# gcc -Wall test.c -o ns && ./ns
 - [ 1831] Hello ?
 - [    1] World !
```

```
root@In Namespace#
# nc -l 1234
# 预计会收到Hi字符串
```

意思是父、子进程分别在两个不同的namespace中，其中父进程的PID是1831，子进程的PID是1（新建namespace中的第一个进程）。请注意上文输出的最后一行光标的停留位置，root@In Namespace～意味着我们已经进入新创建的namespace中。

进入这个新的namespace中，使用nc-l 1234命令监听0.0.0.0:1234。然后打开另一个终端，输入以下命令：

```
# nc 169.254.1.2 1234
Hi # 发送一个Hi字符串
```

意思是向新建network namespace的169.254.1.2:1234发送一个网络报文Hi。切回原先的终端，会发现顺利地收到了一个Hi字符串。

1.1.4 小结

我们知道通过Linux的network namespace技术可以自定义一个独立的网络栈，简单到只有loopback设备，复杂到具备系统完整的网络能力，这就使得network namespace成为Linux网络虚拟化技术的基石——不论是虚拟机还是容器时代。network namespace的另一个隔离功能在于，系统管理员一旦禁用namespace中的网络设备，即使里面的进程拿到了一些系统特权，也无法和外界通信。最后，网络对安全较为敏感，即使network namespace能够提供网络资源隔离的机制，用户还是会结合其他类型的namespace一起使用，以提供更好的安全隔离能力。

关于namespace的介绍就告一段落，如果想探究更多网络虚拟化的奥秘，就要理解更多Linux内核提供的veth、VLAN、VXLAN、Macvlan等知识，我们在后面的章节会一一介绍。

1.2 千呼万唤始出来：veth pair

介绍namespace时，我们多次提到veth pair。本节将详细介绍这对“孪生兄弟”。

部署过Docker或Kubernetes的读者肯定有这样的经历：在主机上输入ifconfig或ip addr命令查询网卡信息的时候，总会出来一大堆vethxxxx之类的网卡名，这些是Docker/Kubernetes为容器而创建的虚拟网卡。

veth是虚拟以太网卡（Virtual Ethernet）的缩写。veth设备总是成对的，因此我们称之为veth pair。veth pair一端发送的数据会在另外一端接收，非常像Linux的双向管道。根据这一特性，veth pair常被用于跨network namespace之间的通信，即分别将veth pair的两端放在不同的namespace里，如图1-2所示。

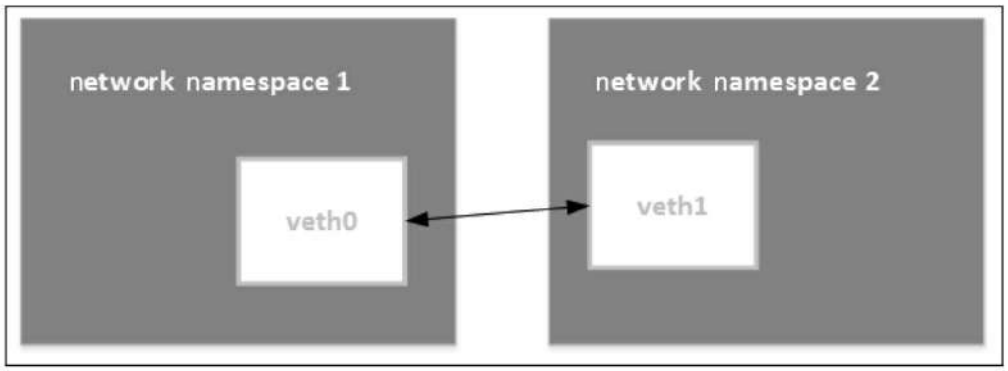


图1-2 veth pair示意图

前文已经提到，仅有veth pair设备，容器是无法访问外部网络的。为什么呢？因为从容器发出的数据包，实际上是直接进了veth pair设备的协议栈。如果容器需要访问网络，则需要使用网桥等技术将veth pair设备接收的数据包通过某种方式转发出去。

注：虽然我们在容器化的场景里经常看到veth网卡，但veth pair在传统的虚拟化技术（比如KVM等）中的应用也非常广泛。

下面，我们将演示veth pair的创建和使用：

```
## 创建veth pair，名字分别是veth0和veth1
# ip link add veth0 type veth peer name veth1
```

创建的veth pair在主机上表现为两块网卡，我们可以通过ip link命令查看：

```
# ip link list
20: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether c6:bb:c0:d0:54:71 brd ff:ff:ff:ff:ff:ff
21: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/ether da:a1:36:d1:3b:36 brd ff:ff:ff:ff:ff:ff
```

如上所示，新创建的veth pair设备的默认mtu是1500，设备初始状态是DOWN。我们同样可以使用ip link命令将这两块网卡的状态设置为UP。

```
# ip link set dev veth0 up
# ip link set dev veth1 up
```

veth pair设备同样可以配置IP地址，命令如下：

```
# ifconfig veth0 10.20.30.40/24
```

可以将veth pair设备放到namespace中。让我们来温习命令的使用：

```
# ip link set veth1 netns newns
```

veth pair设备的原理较简单，就是向veth pair设备的一端输入数据，数据通过内核协议栈后从veth pair的另一端出来。veth pair的基本工作原理如图1-3所示。

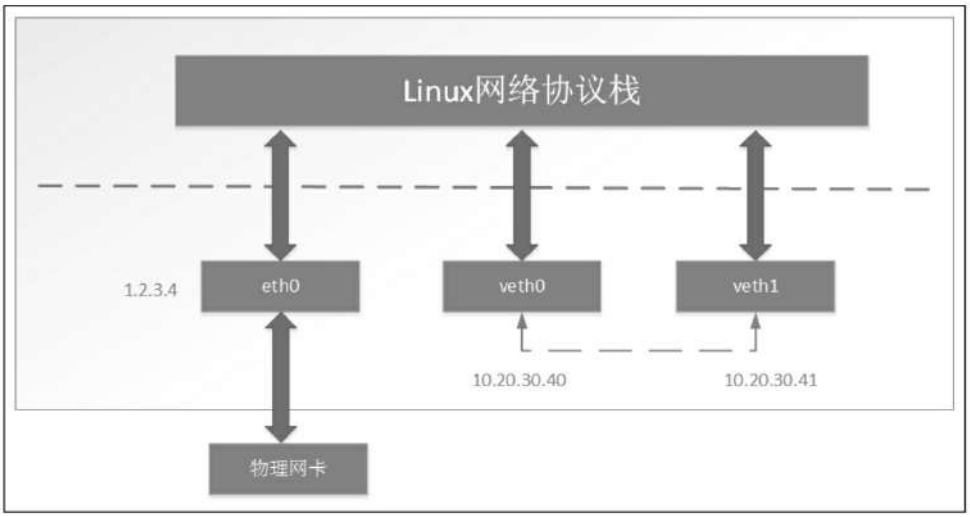


图1-3 veth pair的基本工作原理图

1.2.1 veth pair内核实现

veth pair内核实现的源代码如下：

```
static netdev_tx_t veth_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct veth_priv *priv = netdev_priv(dev);
    struct net_device *rcv;
    int length = skb->len;
    rcu_read_lock();
    rcv = rcu_dereference(priv->peer);
```

```
    if (unlikely(!rcv)) {
        kfree_skb(skb);
        goto drop;
    }
    // 转发给peer
    if (likely(dev_forward_skb(rcv, skb) == NET_RX_SUCCESS)) {
        struct pcpu_vstats *stats = this_cpu_ptr(dev->vstats);
        u64_stats_update_begin(&stats->syncp);
        stats->bytes += length;
        stats->packets++;
        u64_stats_update_end(&stats->syncp);
    } else {
drop:
        atomic64_inc(&priv->dropped);
    }
    rcu_read_unlock();
    return NETDEV_TX_OK;
}
```

上述代码块想表达的意思是：在veth pair设备上，任意一端（RX）接收的数据都会在另一端（TX）发送出去，veth pair在转发过程中不会篡改数据包的内容。

1.2.2 容器与host veth pair的关系

我们要学习的经典容器组网模型就是veth pair+bridge的模式。容器中的eth0实际上和外面host上的某个veth是成对的（pair）关系，那么，有没有办法知道host上的vethxxx和哪个container eth0是成对的关系呢？

方法**1**

首先，在目标容器里查看：

```
# cat /sys/class/net/eth0/iflink
5
```

然后，在主机上遍历/sys/claas/net下面的全部目录，查看子目录ifindex的值和容器里查出来的iflink值相当的veth名字，这样就找到了容器和主机的veth pair关系。例如，下面的例子中主机上veth63a89a3的ifindex刚好是5，意味着是目标容器veth pair的另一端。

```
# cat /sys/class/net/veth63a89a3/ifindex
5
```

方法**2**

在目标容器里执行以下命令：

```
# ip link show eth0
116: eth0@if117: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:0a:00:04:08 brd ff:ff:ff:ff:ff:ff
```

从上面的命令输出可以看到116:eth0@if117，其中116是eth0接口的index，117是和它成对的veth的index。

当host执行下面的命令时，可以看到对应117的veth网卡是哪一个，这样就得到了容器和veth pair的关系。

```
# ip link show | grep 117
117: veth145042b@if116: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue master docker0 state UP
```

方法**3**

可以通过ethtool-S命令列出veth pair对端的网卡index，例如：

```
# ethtool -S eth0
NIC statistics:
     peer_ifindex: 6
```

而主机上index为6的网卡为：

```
# ip addr
6: vethf24874aae57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether a2:ff:0a:99:57:d2 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::a0ff:aff:fe99:57d2/64 scope link
       valid_lft forever preferred_lft forever
```

1.2.3 小结

更多关于容器的介绍请参考后面的章节，这里仅仅是抛砖引玉，让读者对容器与veth pair的关系有个大概的了解。

那么，veth pair的实际用途是什么？1.3节将介绍Linux bridge，当Linux bridge和veth设备相结合时，就能组建一个最简单的容器网络。

1.3 连接你我他：Linux bridge

两个network namespace可以通过veth pair连接，但要做到两个以上network namespace相互连接，veth pair就显得捉襟见肘了。这就轮到本节的主角Linux bridge出场了。

我们在计算机网络课本上学的网桥正如其字面含义所描述的，有“牵线搭桥”之意，用于连接两个不同的局域网，是网线的延伸。网桥是二层网络设备，两个端口分别有一条独立的交换信道，不共享一条背板总线，可隔离冲突域。网桥比集线器（hub）性能更好，集线器上各端口都是共享同一条背板总线的。后来，网桥被具有更多端口、可隔离冲突域的交换机（switch）所取代。

顾名思义，Linux bridge就是Linux系统中的网桥，但是Linux bridge的行为更像是一台虚拟的网络交换机，任意的真实物理设备（例如eth0）和虚拟设备（例如，前面讲到的veth pair和后面即将介绍的tap设备）都可以连接到Linux bridge上。需要注意的是，Linux bridge不能跨机连接网络设备。

Linux bridge与Linux上其他网络设备的区别在于，普通的网络设备只有两端，从一端进来的数据会从另一端出去。例如，物理网卡从外面网络中收到的数据会转发给内核协议栈，而从协议栈过来的数据会转发到外面的物理网络中。Linux bridge则有多个端口，数据可以从任何端口进来，进来之后从哪个口出去取决于目的MAC地址，原理和物理交换机差不多。

1.3.1 Linux bridge初体验

我们先用iproute2软件包里的ip命令创建一个bridge：

```
# ip link add name br0 type bridge
# ip link set br0 up
```

除了ip命令，我们还可以使用bridge-utils软件包里的brctl工具管理网桥，例如新建一个网桥：

```
# brctl addbr br0
```

刚创建一个bridge时，它是一个独立的网络设备，只有一个端口连着协

议栈，其他端口什么都没连接，这样的bridge其实没有任何实际功能，如图1-4所示。

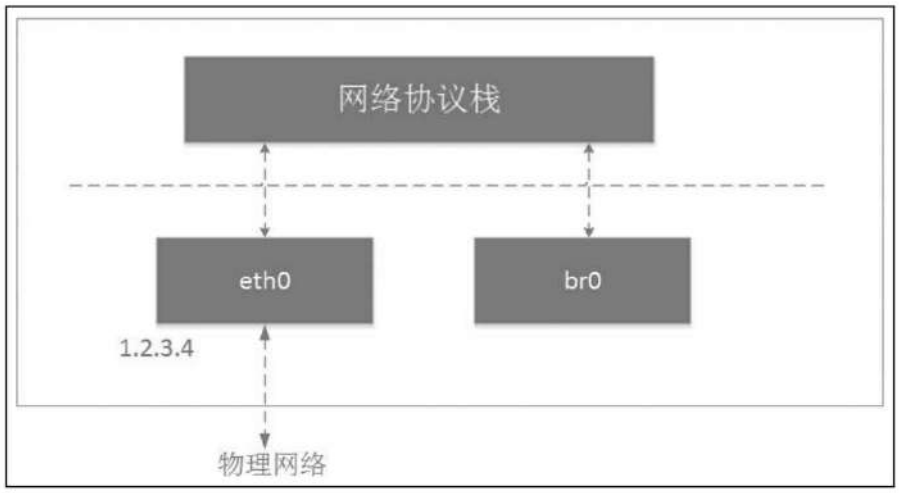


图1-4　独立的bridge设备示意图

假设eth0是我们的物理网卡，IP地址是1.2.3.4，并且假设实验室环境网关地址是1.2.3.1（下文会用到）。

为了充分发挥Linux bridge的作用，我们特将它和前文介绍的veth pair配合起来使用。我们将创建一对veth设备，并配置IP地址：

```
# ip link add veth0 type veth peer name veth1
# ip addr add 1.2.3.101/24 dev veth0
# ip addr add 1.2.3.102/24 dev veth1
# ip link set veth0 up
# ip link set veth1 up
```

然后，通过下面的命令将veth0连接到br0上：

```
# ip link set dev veth0 master br0
```

同样，可以使用brctl命令添加一个设备到网桥上：

```
# brctl addif br0 veth0
```

成功对接后，可以通过bridge　link（bridge也是iproute2的组成部分）命令查看当前网桥上都有哪些网络设备：

```
# bridge link
9: veth0 state UP : <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br0 state
forwarding priority 32 cost 2
```

也可以使用brctl命令显示当前存在的网桥及其所连接的网络端口，这个命令的输出和bridge link的输出有所区别，命令如下所示：

```
# brctl show
bridge name     bridge id           STP enabled     interfaces
br0             8000.02423c5eccd8   no              veth0
```

执行完以上命令后，连接veth pair的bridge设备的网络拓扑如图1-5所示。

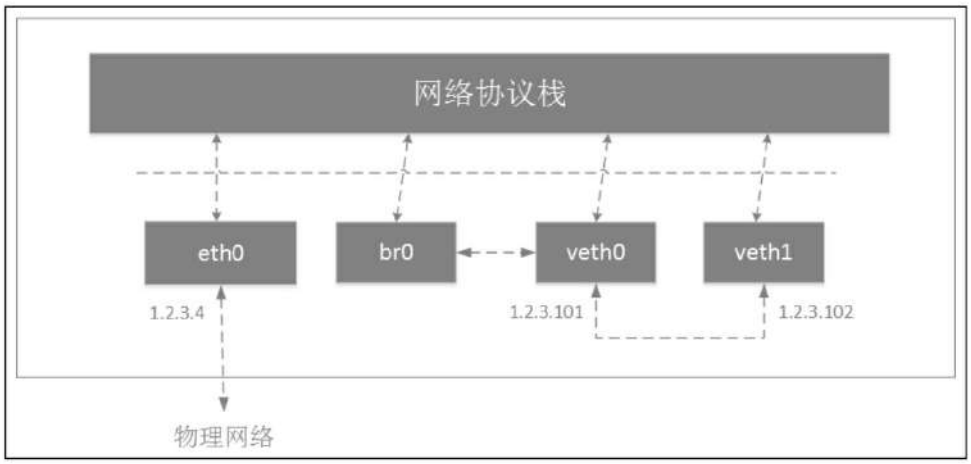


图1-5　连接veth pair的bridge设备的网络拓扑

br0和veth0相连之后发生了如下变化：

·br0和veth0之间连接起来了，并且是双向的通道；

·协议栈和veth0之间变成了单通道，协议栈能发数据给veth0，但veth0从外面收到的数据不会转发给协议栈；

·br0的MAC地址变成了veth0的MAC地址。

这就好比Linux bridge在veth0和协议栈之间做了一次拦截，在veth0上面做了点小动作，将veth0本来要转发给协议栈的数据拦截，全部转发给bridge。同时，bridge也可以向veth0发数据。

让我们做个小实验来验证以上观点。

首先，从veth0 ping veth1：

```
# ping -c 1 -I veth0 1.2.3.102
PING 1.2.3.102 (1.2.3.102) from 1.2.3.101 veth0: 56(84) bytes of data.
^C

--- 1.2.3.102 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

如上所示，veth0 ping veth1失败。为什么veth0加入bridge之后，就ping不通对端的veth1了呢？1.2.3.102原本应该是能ping通的，让我们通过抓包深入分析。先抓veth1网卡上的报文：

```
# tcpdump -n -i veth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
21:43:48.353509 ARP, Request who-has 1.2.3.102 tell 1.2.3.101, length 28
21:43:48.353518 ARP, Reply 1.2.3.102 is-at 26:58:a2:57:37:e9, length 28
```

如上所示，由于veth0的ARP缓存里没有veth1的MAC地址，所以ping之前先发ARP请求。veth1抓取的报文显示，veth1收到了ARP请求，并且返回了应答。

再抓veh0网卡上的报文：

```
# tcpdump -n -i veth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on veth0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:44:09.775392 ARP, Request who-has 1.2.3.102 tell 1.2.3.101, length 28
21:44:09.775400 ARP, Reply 1.2.3.102 is-at 26:58:a2:57:37:e9, length 28
```

如上所示，veth0上的数据包都发出去了，而且也收到了响应。

再看br0上的数据包，发现只有应答，如下所示：

```
# tcpdump -n -i br0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on br0, link-type EN10MB (Ethernet), capture size 262144 bytes
21:45:48.225459 ARP, Reply 192.168.3.102 is-at 26:58:a2:57:37:e9, length 28
```

通过分析以下报文可以看出，包的去和回的流程都没有问题，问题就出

在veth0收到应答包后没有给协议栈，而是给了br0，于是协议栈得不到veth1的MAC地址，导致通信失败。

1.3.2 把IP让给Linux bridge

通过上面的分析可以看出，给veth0配置IP没有意义，因为就算协议栈传数据包给veth0，回程报文也回不来。这里我们就把veth0的IP地址“让给”Linux bridge：

```
# ip addr del 1.2.3.101/24 dev veth0
# ip addr add 1.2.3.101/24 dev br0
```

以上命令将原本分配给veth0的IP地址配置到br0上。于是，绑定IP地址的bridge设备的网络拓扑如图1-6所示。

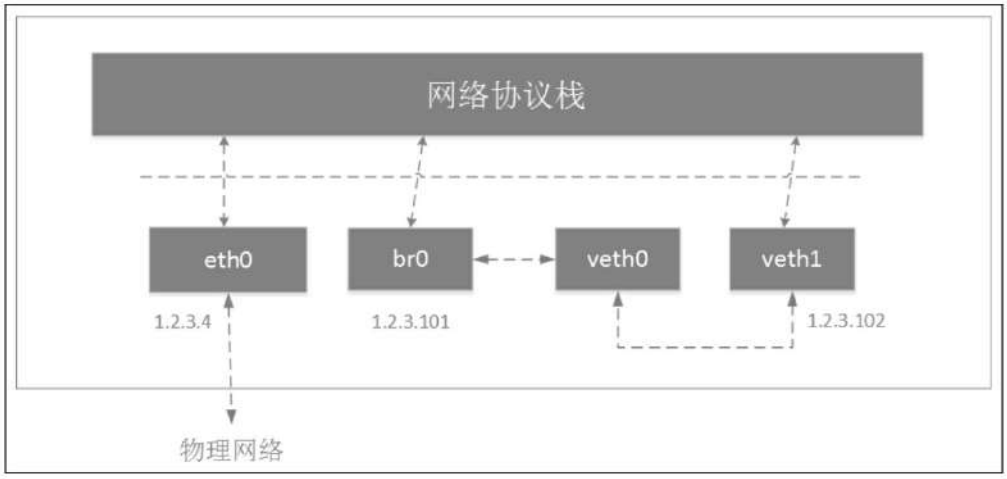


图1-6　绑定IP地址的bridge设备的网络拓扑

图1-6将协议栈和veth0之间的联系去掉了，veth0相当于一根网线。实际上，veth0和协议栈之间是有联系的，但由于veth0没有配置IP，所以协议栈在路由的时候不会将数据包发给veth0。就算强制要求数据包通过veth0发送出去，由于veth0从另一端收到的数据包只会给br0，协议栈还是没法收到相应的ARP应答包，同样会导致通信失败。

这时，再通过br0 ping veth1，结果成功收到了ICMP的回程报文：

```
# ping -c 1 -I br0 1.2.3.102
PING 1.2.3.102 (1.2.3.102) from 1.2.3.101 br0: 56(84) bytes of data.
64 bytes from 192.168.3.102: icmp_seq=1 ttl=64 time=0.121 ms

--- 1.2.3.102 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

但ping网关还是失败：

```
# ping -c 1 -I br0 1.2.3.1
PING 1.2.3.1 (1.2.3.1) from 1.2.3.101 br0: 56(84) bytes of data.
^C

--- 1.2.3.1 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

因为这个br0上只有192.168.3.101和192.168.3.102这两个网络设备，不知道1.2.3.1在哪儿。

1.3.3 将物理网卡添加到Linux bridge

下面，我们演示如何将主机上的物理网卡eth0添加到Linux bridge：

```
# ip link set dev eth0 master br0

# bridge link
2: eth0 state UP : <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br0 state
forwarding priority 32 cost 4
6: veth0 state UP : <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br0 state
forwarding priority 32 cost 2
```

Linux bridge不会区分接入进来的到底是物理设备还是虚拟设备，对它来说没有区别。因此，eth0加入br0后，落得和上面veth0一样的“下场”，从外面网络收到的数据包将无条件地转发给br0，自己变成了一根网线。

这时，通过eth0 ping网关失败。因为br0通过eth0这根网线连上了外面的物理交换机，所以连在br0上的设备都能ping通网关，这里连上的设备就是veth1和br0自己，veth1是通过eth0这根网线连上去的，而br0有一块自带的网

卡。

通过br0 ping网关成功：

```
# ping -c 1 -I br0 1.2.3.1
PING 1.2.3.1 (1.2.3.1) from 1.2.3.101 br0: 56(84) bytes of data.
64 bytes from 1.2.3.1: icmp_seq=1 ttl=64 time=27.5 ms

--- 1.2.3.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

通过veth1 ping网关成功：

```
# ping -c 1 -I veth1 1.2.3.1
PING 1.2.3.1 (1.2.3.1) from 1.2.3.102 veth1: 56(84) bytes of data.
64 bytes from 192.168.3.1: icmp_seq=1 ttl=64 time=68.8 ms

--- 1.2.3.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

通过eth0 ping网关失败：

```
# ping -c 1 -I eth0 1.2.3.1
PING 1.2.3.1 (1.2.3.1) from 1.2.3.4 eth0: 56(84) bytes of data.
^C

--- 1.2.3.1 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms
```

因为eth0的功能已经和网线差不多，所以在eth0上配置IP没有意义，还会影响协议栈的路由选择。例如，如果ping的时候不指定网卡，则协议栈有可能优先选择eth0，导致ping不通。因此，需要将eth0上的IP去掉。在以上测试过程中，由于eth0上有IP，在访问1.2.3.0/24网段时，会优先选择eth0。可以通过查看主机路由表来验证我们的判断：

```
# sudo route -v
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         1.2.3.1         0.0.0.0         UG    0      0        0 eth0
link-local      *               255.255.0.0     U     1000   0        0 eth0
1.2.3.0         *               255.255.255.0   U     0      0        0 eth0
1.2.3.0         *               255.255.255.0   U     0      0        0 veth1
1.2.3.0         *               255.255.255.0   U     0      0        0 br0
```

eth0接入了br0，因此它收到的数据包都会转发给br0，于是协议栈收不到ARP应答包，导致ping失败。

让我们将eth0上的IP删除：

```
# ip addr del 192.168.3.21/24 dev eth0
```

这时，再从eth0 ping一次网关，成功收到ICMP响应报文：

```
# ping -c 1 -I eth0 1.2.3.1
PING 1.2.3.1 (1.2.3.1) 56(84) bytes of data.
64 bytes from 1.2.3.1: icmp_seq=1 ttl=64 time=3.91 ms

--- 1.2.3.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

当我们删除eth0的IP后，路由表里就没有它了，于是数据包会从veth1出去。可以通过查看主机路由表来验证我们的判断。

```
# route -v
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
1.2.3.0         *               255.255.255.0   U     0      0        0 veth1
1.2.3.0         *               255.255.255.0   U     0      0        0 br0
```

通过观察以上路由表信息可以看出：原来的默认路由进过eth0，eth0的IP被删除后，默认路由不见了，想要连接1.2.3.0/24以外的网段，需要手动将默认网关加回来。

添加默认网关：

```
# sudo ip route add default via 192.168.3.1
```

再ping外网，成功返回ICMP报文：

```
# ping -c 1 www.baidu.com
PING baidu.com (111.13.101.208) 56(84) bytes of data.
64 bytes from 111.13.101.208: icmp_seq=1 ttl=51 time=30.6 ms

--- www.baidu.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

经过上面一系列的操作，将物理网卡添加到bridge设备的网络拓扑如图1-7所示。

注：要完成以上所有实验步骤，需要打开eth0网卡的混杂模式（下文会详细介绍Linux bridge的混杂模式），不然veth1的网络会不通。当eth0不在混杂模式时，只会接收目的MAC地址是自己的报文，丢掉目的MAC地址是veth1的数据包。

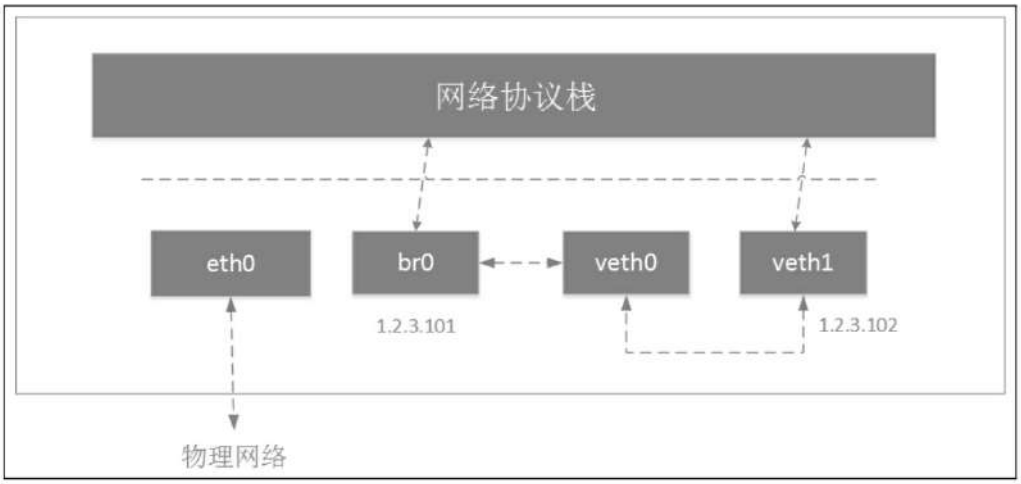


图1-7　将物理网卡添加到bridge设备的网络拓扑

1.3.4　Linux bridge在网络虚拟化中的应用

以上例子是为了阐述Linux bridge的底层机制而设计的，下面将通过Linux bridge的两种常见的部署方式说明其在现代网络虚拟化技术中的地位。

1.虚拟机

虚拟机通过tun/tap或者其他类似的虚拟网络设备，将虚拟机内的网卡同br0连接起来，这样就达到和真实交换机一样的效果，虚拟机发出去的数据包先到达br0，然后由br0交给eth0发送出去，数据包都不需要经过host机器

的协议栈，效率高，如图1-8所示。如果有多个虚拟机，那么这些虚拟机通过tun/tap设备连接到网桥。tun/tap设备的详细介绍将在1.4节展开。

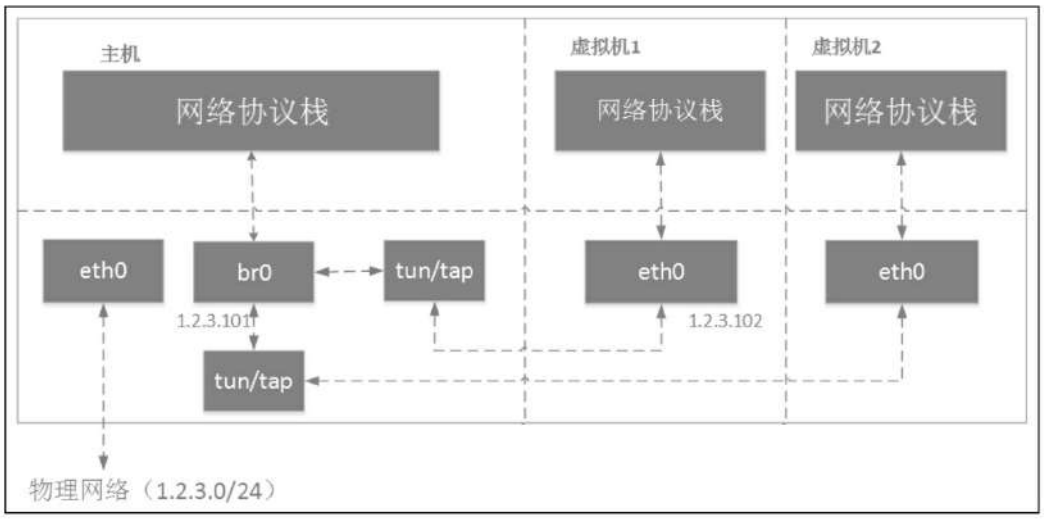


图1-8　Linux bridge在虚拟机中的应用

2.容器

容器运行在自己单独的network namespace里，因此都有自己单独的协议栈。Linux　bridge在容器场景的组网和上面的虚拟机场景差不多，但也存在一些区别。例如，容器使用的是veth　pair设备，而虚拟机使用的是tun/tap设备。在虚拟机场景下，我们给主机物理网卡eth0分配了IP地址；而在容器场景下，我们一般不会对宿主机eth0进行配置。在虚拟机场景下，虚拟器一般会和主机在同一个网段；而在容器场景下，容器和物理网络不在同一个网段内。Linux bridge在容器中的应用如图1-9所示。

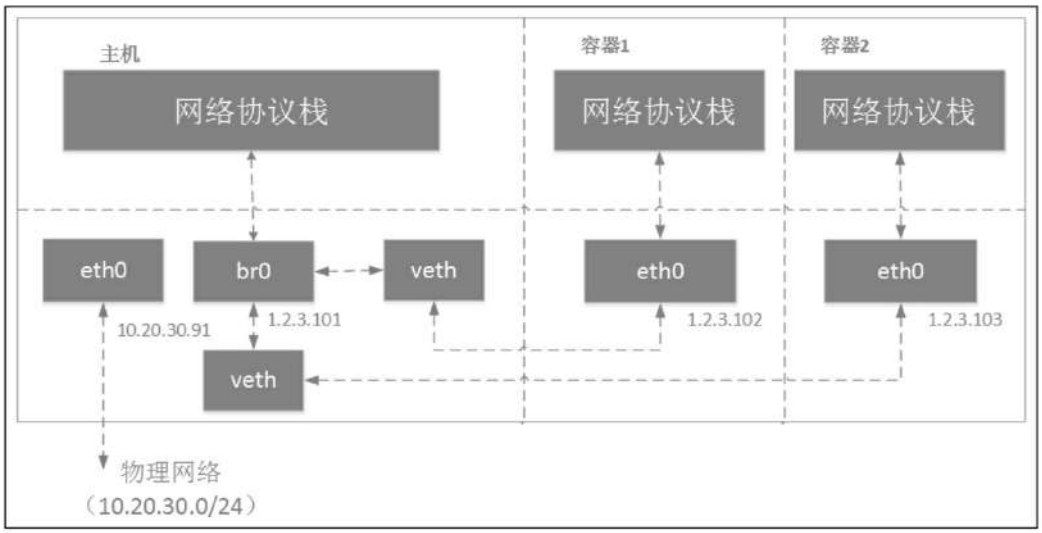


图1-9　Linux bridge在容器中的应用

在容器中配置其网关地址为br0，在我们的例子中即1.2.3.101（容器网络网段是1.2.3.0/24）。因此，从容器发出去的数据包先到达br0，然后交给

host机器的协议栈。由于目的IP是外网IP，且host机器开启了IP forward功能，数据包会通过eth0发送出去。因为容器所分配的网段一般都不在物理网络网段内（在我们的例子中，物理网络网段是10.20.30.0/24），所以一般发出去之前会先做NAT转换（NAT转换需要自己配置，可以使用iptables，1.5节会介绍iptables）。

1.3.5 网络接口的混杂模式

前文提到过网桥的混杂模式，网络接口的混杂模式在Kubernetes网络也有应用，本节将重点讨论网络接口的混杂模式。

混杂模式（Promiscuous mode），简称Promisc mode，俗称“监听模式”。混杂模式通常被网络管理员用来诊断网络问题，但也会被无认证的、想偷听网络通信的人利用。根据维基百科的定义，混杂模式是指一个网卡会把它接收的所有网络流量都交给CPU，而不是只把它想转交的部分交给CPU。在IEEE 802定的网络规范中，每个网络帧都有一个目的MAC地址。在非混杂模式下，网卡只会接收目的MAC地址是它自己的单播帧，以及多播及广播帧；在混杂模式下，网卡会接收经过它的所有帧！

我们可以使用ifconfig或者netstat-i命令查看一个网卡是否开启了混杂模式。

·ifconfig eth0，查看eth0的配置，包括混杂模式。当输出包含PROMISC时，表明该网络接口处于混杂模式。

```
# ifconfig eth0
  eth0    Link encap:Ethernet  HWaddr a0:36:9f:97:b1:68
          inet6 addr: fe80::a236:9fff:fe97:b168/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
```

启用网卡的混杂模式，可以使用下面这条命令：

```
# ifconfig eth0 promisc
```

使网卡退出混杂模式，可以使用下面这条命令：

```
# ifconfig eth0 -promisc
```

将网络设备加入Linux bridge后，会自动进入混杂模式。可以通过下面的小实验来说明：

```
# brctl addif br0 veth0

# dmesg | grep promiscuous
[298681.199680] device veth0 entered promiscuous mode
```

如上所示，veth设备加入Linux bridge后，可以通过查看内核日志看到veth0自动进入混杂模式，而且无法退出，直到将veth0从Linux bridge中移除。

即使手动将网卡设置为非混杂模式，实际上还是没有退出混杂模式，一边操作ifconfig veth0-promisc，一边观察内核日志（内核并不会真正处理）便可看出。有兴趣的读者可以自行验证，这里不再赘述。

网络设备离开Linux bridge后，会自动退出混杂模式，如下所示：

```
# brctl delif br0 veth0

# dmesg | grep promiscuous
[498665.638647] device veth0 left promiscuous mode
```

1.4 给用户态一个机会：tun/tap设备

我们在1.3节讲解Linux bridge时就向读者介绍过tun/tap设备，并强调tun/tap设备在虚拟机的组网过程中起到作用。但促使我们用一节的篇幅介绍它的另一个原因是：tun/tap设备是理解flannel的基础，而flannel是一个重要的Kubernetes网络插件。

tun/tap设备到底是什么？从Linux文件系统的角度看，它是用户可以用文件句柄操作的字符设备；从网络虚拟化角度看，它是虚拟网卡，一端连着网络协议栈，另一端连着用户态程序。

如果把veth pair称为设备孪生，那么tun/tap就像是一对表兄弟。虽然很多情况下我们都是连带提到它们，但它们还是有些区别的。tun表示虚拟的是点对点设备，tap表示虚拟的是以太网设备，这两种设备针对网络包实施不同的封装。

tun/tap设备有什么作用呢？tun/tap设备可以将TCP/IP协议栈处理好的网络包发送给任何一个使用tun/tap驱动的进程，由进程重新处理后发到物理链路中。tun/tap设备就像是埋在用户程序空间的一个钩子，我们可以很方便地将对网络包的处理程序挂在这个钩子上，OpenVPN、Vtun、flannel都是基于它实现隧道包封装的。

1.4.1 tun/tap设备的工作原理

我们先简单介绍物理设备上的数据是如何通过Linux网络栈送达用户态程序的，tun/tap设备的基本原理如图1-10所示。

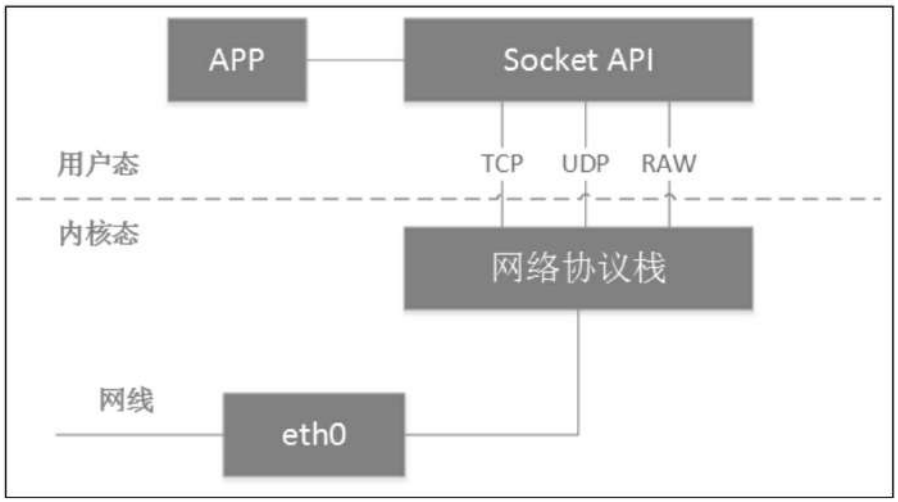


图1-10 tun/tap设备的基本原理

图1-10是一个经典的、通过Socket调用实现用户态和内核态数据交互的过程。物理网卡从网线接收数据后送达网络协议栈，而进程通过Socket创建特殊套接字，从网络协议栈读取数据。

从网络协议栈的角度看，tun/tap设备这类虚拟网卡与物理网卡并无区别。只是对tun/tap设备而言，它与物理网卡的不同表现在它的数据源不是物理链路，而是来自用户态！这也是tun/tap设备的最大价值所在。提前“剧透”：flannel的UDP模式的技术要点就是tun/tap设备。

tun/tap设备其实就是利用Linux的设备文件实现内核态和用户态的数据交互，而访问设备文件则会调用设备驱动相应的例程，要知道设备驱动也是内核态和用户态的一个接口。tun设备的工作模式如图1-11所示。

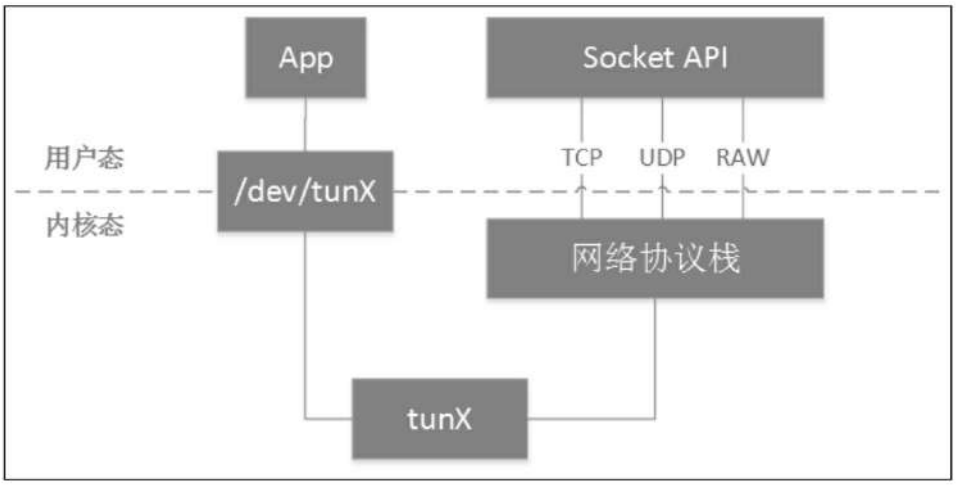


图1-11　tun设备的工作模式

普通的物理网卡通过网线收发数据包，而tun设备通过一个设备文件（/dev/tunX）收发数据包。所有对这个文件的写操作会通过tun设备转换成一个数据包传送给内核网络协议栈。当内核发送一个包给tun设备时，用户态的进程通过读取这个文件可以拿到包的内容。当然，用户态的程序也可以通过写这个文件向tun设备发送数据。

tap设备与tun设备的工作原理完全相同，区别在于：

·tun设备的/dev/tunX文件收发的是IP包，因此只能工作在L3，无法与物理网卡做桥接，但可以通过三层交换（例如ip_forward）与物理网卡连通；

·tap设备的/dev/tapX文件收发的是链路层数据包，可以与物理网卡做桥接。

1.4.2　利用tun设备部署一个VPN

tun设备的tun是英文隧道（tunnel）的缩写，言下之意，tun设备似乎与

隧道网络存在一丝联系。tun/tap设备的用处是将协议栈中的部分数据包转发给用户空间的应用程序，给用户空间的程序一个处理数据包的机会。常见的tun/tap设备使用场景有数据压缩、加密等，最常见的是VPN，包括tunnel及应用层的IPSec等。我们将使用tun设备搭建一个基于UDP的VPN，网络拓扑如图1-12所示。

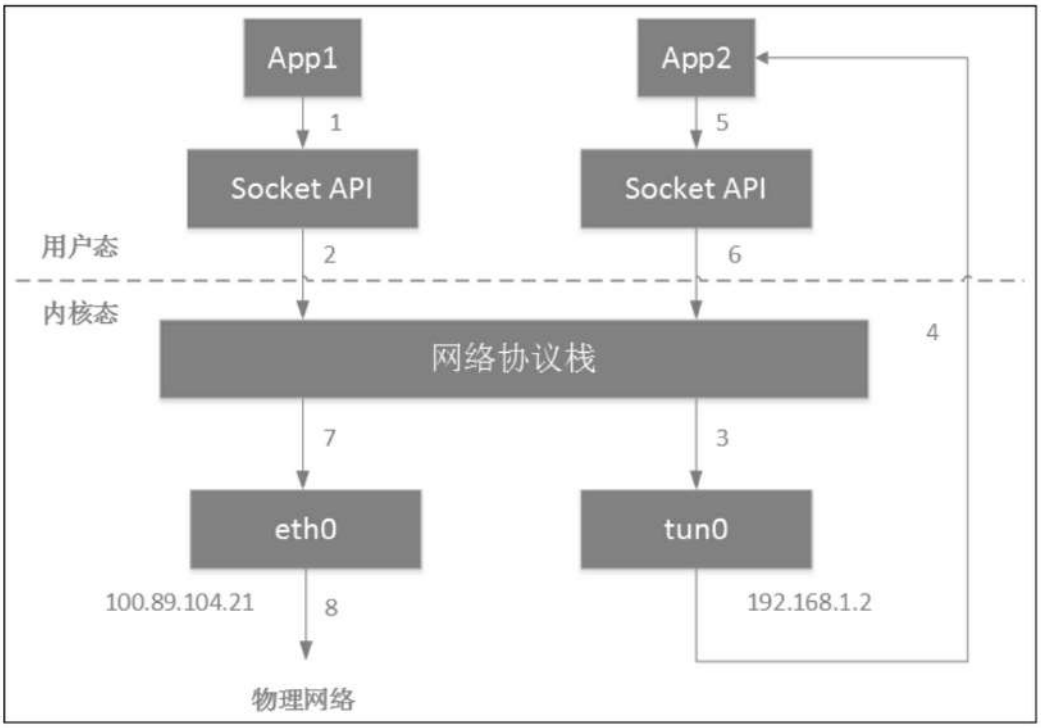


图1-12　使用tun设备搭建一个基于UDP的VPN

如上所示，数据包的流程包括：

（1）App1是一个普通的程序，通过Socket　API发送了一个数据包，假设这个数据包的目的IP地址是192.168.1.3（和tun0在同一个网段）。

（2）程序A的数据包到达网络协议栈后，协议栈根据数据包的目的IP地址匹配到这个数据包应该由tun0网口出去，于是将数据包发送给tun0网卡。

（3）tun0网卡收到数据包之后，发现网卡的另一端被App2打开了（这也是tun/tap设备的特点，一端连着协议栈，另一端连着用户态程序），于是将数据包发送给App2。

（4）App2收到数据包之后，通过报文封装（将原来的数据包封装在新的数据报文中，假设新报文的原地址是eth0的地址，目的地址是和eth0在同一个网段的VPN对端IP地址，例如100.89.104.22）构造出一个新的数据包。App2通过同样的Socket API将数据包发送给协议栈。

（5）协议栈根据本地路由，发现这个数据包应该通过eth0发送出去，于是将数据包交给eth0，最后eth0通过物理网络将数据包发送给VPN的对

端。

综上所述，发到192.168.1.0/24网络的数据首先通过监听在tun0设备上的App2进行封包，利用eth0这块物理网卡发到远端网络的物理网卡上，从而实现VPN。

不难看出，VPN网络的报文真正从物理网卡出去要经过网络协议栈两次，因此会有一定的性能损耗。另外，经过用户态程序的处理，数据包可能已经加密，包头进行了封装，所以第二次通过网络栈内核看到的是截然不同的网络包。这个过程和我们后面要讨论的flannel容器组网方案有异曲同工之处，flannel网络的本质就是一个隧道网络，后面我们会做更深入的介绍。

1.4.3 tun设备编程

我们将用一个简单的C语言程序示范tun设备的具体使用方法。这个程序在收到tun设备的数据包之后，打印出收到了多少字节的数据包。程序代码如下所示：

```
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <linux/if_tun.h>
#include<stdlib.h>
#include<stdio.h>

int tun_alloc(int flags)
{
    struct ifreq ifr;
    int fd, err;
    char *clonedev = "/dev/net/tun";

    if ((fd = open(clonedev, O_RDWR)) < 0) {
        return fd;
    }

    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = flags;

    if ((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0) {
        close(fd);
        return err;
```

```
    }

    printf("Open tun/tap device: %s for reading...\n", ifr.ifr_name);

    return fd;
}

int main()
{
    int tun_fd, nread;
    char buffer[1500];

    /* Flags: IFF_TUN   - TUN device (no Ethernet headers)
     *        IFF_TAP   - TAP device
     *        IFF_NO_PI - Do not provide packet information
     */
    tun_fd = tun_alloc(IFF_TUN | IFF_NO_PI);

    if (tun_fd < 0) {
        perror("Allocating interface");
        exit(1);
    }
    while (1) {
        nread = read(tun_fd, buffer, sizeof(buffer));
        if (nread < 0) {
            perror("Reading from interface");
            close(tun_fd);
            exit(1);
        }
        printf("Read %d bytes from tun/tap device\n", nread);
    }
    return 0;
}
```

假设我们把以上C语言代码保存在tun.c文件中，然后编译成tun二进制文件：

```
# gcc tun.c -o tun
```

tun程序启动后会一直阻塞并等待接收数据包：

```
# ./tun
Open tun/tap device tun1 for reading...
```

虽然tun程序目前还没有任何输出，但系统已经自动创建了一块新的tun设备。打开另一个shell terminal，通过ip命令查看网卡，输出如下所示：

```
#ip addr
12: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default
qlen 500
    link/none
```

这块tun0网卡没有被分配IP地址，初始状态也是DOWN。从tun0网卡的POINTOPOINT输出，也可以验证上文提到的“tun表示虚拟点对点设备”。接下来，将给它分配IP地址192.128.1.2，并设置状态为UP：

```
# ip addr add 192.128.1.2/24 dev tun0
# ip link set tun0 up
```

有意思的是，当我们做好上面一系列配置后，tun0网卡上已经收到3个48字节的报文：

```
# ./tun
Open tun/tap device: tun0 for reading...
Read 48 bytes from tun/tap device
Read 48 bytes from tun/tap device
Read 48 bytes from tun/tap device
```

再来给tun0发送4个ping包看看会发生什么：

```
# ping -c 4 192.128.1.2
^C
```

尽管ping包没有返回，但当我们切回shell terminal会发现tun0网卡上又收到了6个84字节的报文，如下所示：

```
# ./tun
Open tun/tap device: tun0 for reading...
Read 48 bytes from tun/tap device
Read 48 bytes from tun/tap device
```

```
Read 48 bytes from tun/tap device
Read 84 bytes from tun/tap device
Read 84 bytes from tun/tap device
Read 84 bytes from tun/tap device
```

对上述现象的解释是，ping包发送给了tun0网卡。因为我们的程序在收到数据包后不做任何处理也没有返回报文，所以我们看不到ping的回程报文。我们可以在发送ping包的同时通过tcpdump抓包看到发出去的4个icmp echo请求包，如下所示：

```
# tcpdump -i tun1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tun1, link-type RAW (Raw IP), capture size 262144 bytes
16:49:15.463101 IP 100.106.89.164 > 192.128.1.2: ICMP echo request, id 24028, seq 1,
length 64
16:49:15.480361 IP 100.106.89.164 > 192.128.1.2: ICMP echo request, id 24028, seq 2,
length 64
16:49:15.481246 IP 100.106.89.164 > 192.128.1.2: ICMP echo request, id 24028, seq 3,
length 64
16:49:16.496153 IP 100.106.89.164 > 192.128.1.2: ICMP echo request, id 24028, seq 4,
length 64
```

这也说明数据包正确地发送给了程序tun。

1.5 iptables

iptables在Docker和Kubernetes网络中应用甚广。例如，Docker容器和宿主机的端口映射、Kubernetes Service的默认模式、CNI的portmap插件、Kubernetes网络策略等都是通过iptables实现的，因此本节补充一些介绍iptables的基本知识，这对后面理解这些概念大有裨益。由于iptables涉及的知识点太广，面面俱到的话足以单独写成一本书（有不少专门讲解iptables的英文书籍），例如，*Linux Iptables Pocket Reference*，感兴趣的读者可以自行翻阅。本节只介绍在后面章节会用到的iptables的基本工作机制和相关用法。

1.5.1 祖师爷netfilter

iptables的底层实现是netfilter。netfilter是Linux内核2.4版引入的一个子系统，最初由Linux内核防火墙和网络的维护者Rusty Russell提出。它作为一个通用的、抽象的框架，提供一整套hook函数的管理机制，使得数据包过滤、包处理（设置标志位、修改TTL等）、地址伪装、网络地址转换、透明代理、访问控制、基于协议类型的连接跟踪，甚至带宽限速等功能成为可能。netfilter的架构就是在整个网络流程的若干位置放置一些钩子，并在每个钩子上挂载一些处理函数进行处理。

IP层的5个钩子点的位置，对应iptables就是5条内置链，分别是PREROUTING、POSTROUTING、INPUT、OUTPUT和FORWARD。netfilter原理图如图1-13所示。

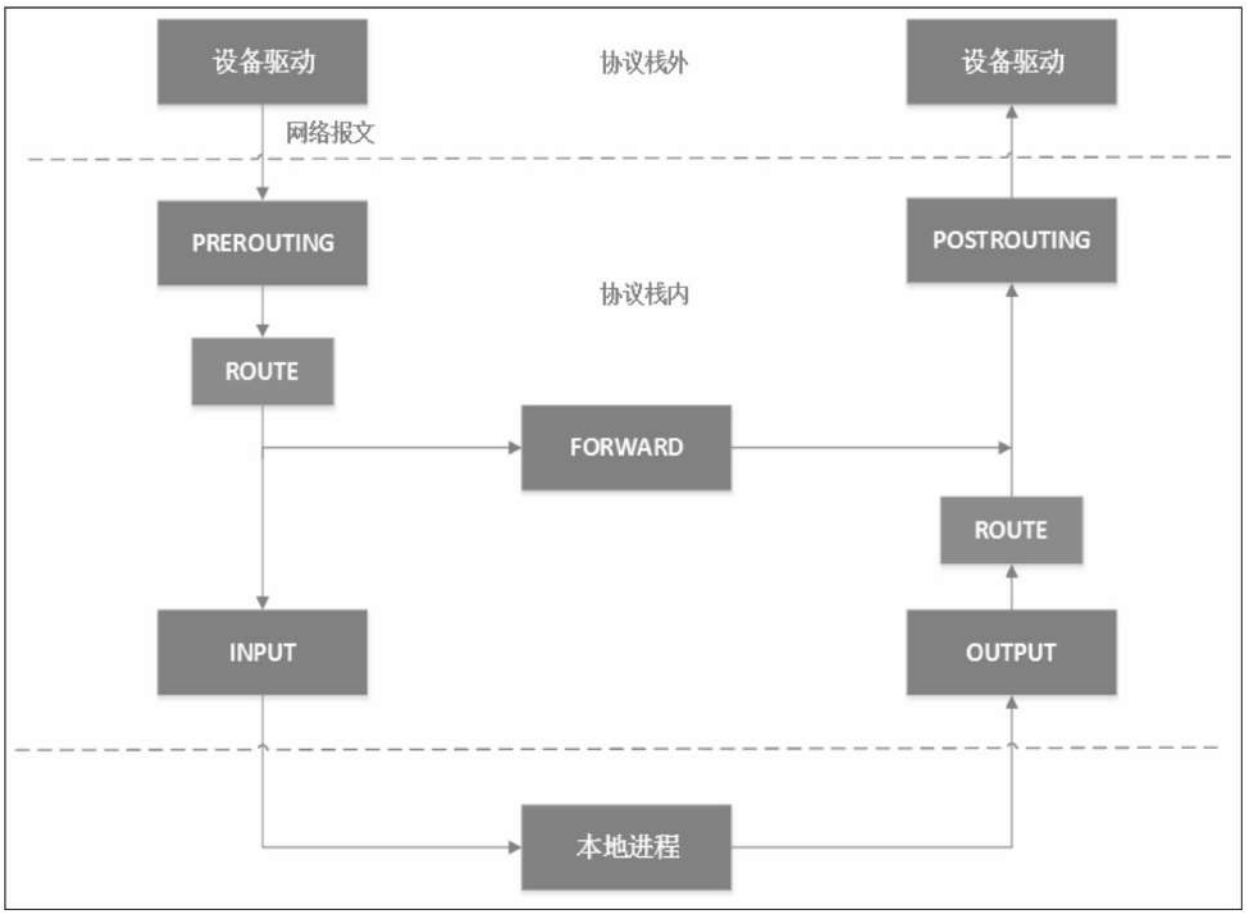


图1-13　netfilter原理图

当网卡上收到一个包送达协议栈时，最先经过的netfilter钩子是PREROUTING，如果确实有用户埋了这个钩子函数，那么内核将在这里对数据包进行目的地址转换（DNAT）。不管在PREROUTING有没有做过DNAT，内核都会通过查本地路由表决定这个数据包是发送给本地进程还是发送给其他机器。如果是发送给其他机器（或其他network namespace），就相当于把本地当作路由器，就会经过netfilter的FORWARD钩子，用户可以在此处设置包过滤钩子函数，例如iptables的reject函数。所有马上要发到协议栈外的包都会经过POSTROUTING钩子，用户可以在这里埋下源地址转换（SNAT）或源地址伪装（Masquerade，简称Masq）的钩子函数。如果经过上面的路由决策，内核决定把包发给本地进程，就会经过INPUT钩子。本地进程收到数据包后，回程报文会先经过OUTPUT钩子，然后经过一次路由决策（例如，决定从机器的哪块网卡出去，下一跳地址是多少等），最后出协议栈的网络包同样会经过POSTROUTING钩子。

netfilter是Linux内核网络模块的一个经典框架，毫不夸张地说，整个Linux系统的网络安全大厦都构建在netfilter之上，如图1-14所示。

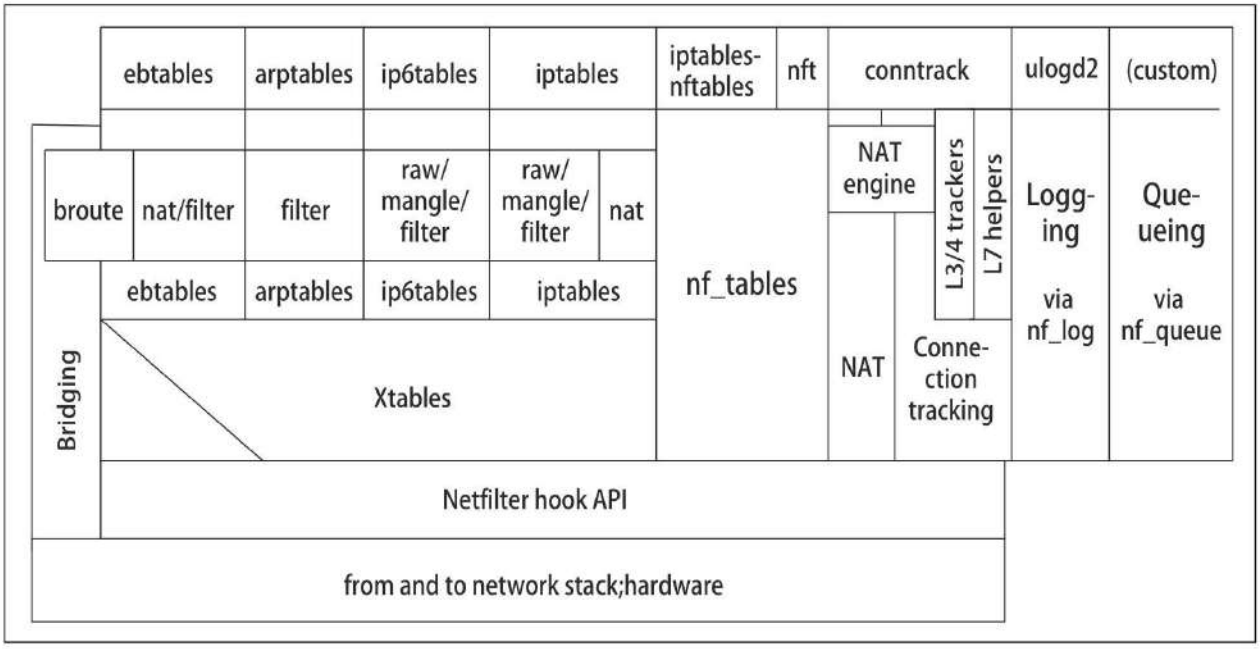


图1-14　netfilter在Linux网络中的地位

我们可以发现，构建在netfilter钩子之上的网络安全策略和连接跟踪的用户态程序就有ebtables、arptables、（IPv6版本的）ip6tables、iptables、iptables-nftables（iptables的改进版本）、conntrack（连接跟踪）等。Kubernetes网络之间用到的工具就有ebtables、iptables/ip6tables和conntrack，其中iptables是核心。

1.5.2　iptables的三板斧：table、chain和rule

iptables是用户空间的一个程序，通过netlink和内核的netfilter框架打交道，负责往钩子上配置回调函数。一般情况下用于构建Linux内核防火墙，特殊情况下也做服务负载均衡（这是Kubernetes的特色操作，我们将在后面章节专门分析）。iptables的工作原理如图1-15所示。

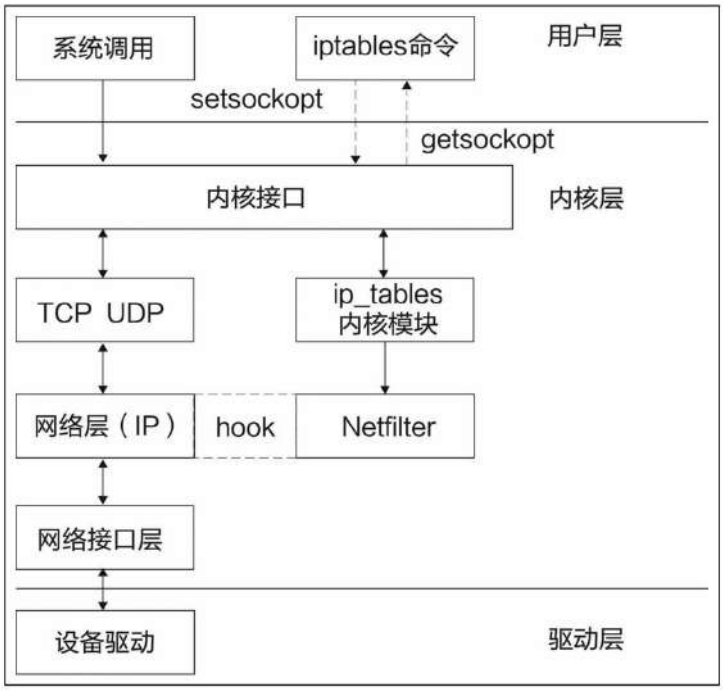


图1-15　iptables的工作原理

我们常说的iptables 5X5，即5张表（table）和5条链（chain）。5条链即iptables的5条内置链，对应上文介绍的netfilter的5个钩子。这5条链分别是：

·**INPUT**链：一般用于处理输入本地进程的数据包；

·**OUTPUT**链：一般用于处理本地进程的输出数据包；

·**FORWARD**链：一般用于处理转发到其他机器/network namespace的数据包；

·**PREROUTING**链：可以在此处进行DNAT；

·**POSTROUTING**链：可以在此处进行SNAT。

除了系统预定义的5条iptables链，用户还可以在表中定义自己的链，我们将通过例子详细说明。

5张表如下所示。

·**filter**表：用于控制到达某条链上的数据包是继续放行、直接丢弃（drop）或拒绝（reject）；

·**nat**表：用于修改数据包的源和目的地址；

·**mangle**表：用于修改数据包的IP头信息；

·**raw**表：iptables是有状态的，即iptables对数据包有连接追踪（connection tracking）机制，而raw是用来去除这种追踪机制的；

·**security**表：最不常用的表（通常，我们说iptables只有4张表，security表是新加入的特性），用于在数据包上应用SELinux。

这5张表的优先级从高到低是：raw、mangle、nat、filter、security。需要注意的是，iptables不支持用户自定义表。

不是每个链上都能挂表，iptables表与链的对应关系如表1-1所示。

表1-1　iptables表与链的对应关系

	PREROUTING	POSTROUTING	FORWARD	INPUT	OUTPUT
raw	Y	N	N	N	Y
mangle	Y	Y	Y	Y	Y
nat(SNAT)	N	Y	N	Y	N
nat(DNAT)	Y	N	N	N	Y
filter	N	N	Y	Y	Y
security	N	N	Y	Y	Y

所以，如果我们扩充图1-13，给它加上iptables的5张表，那么一个网络包经过iptables的处理路径如图1-16所示。

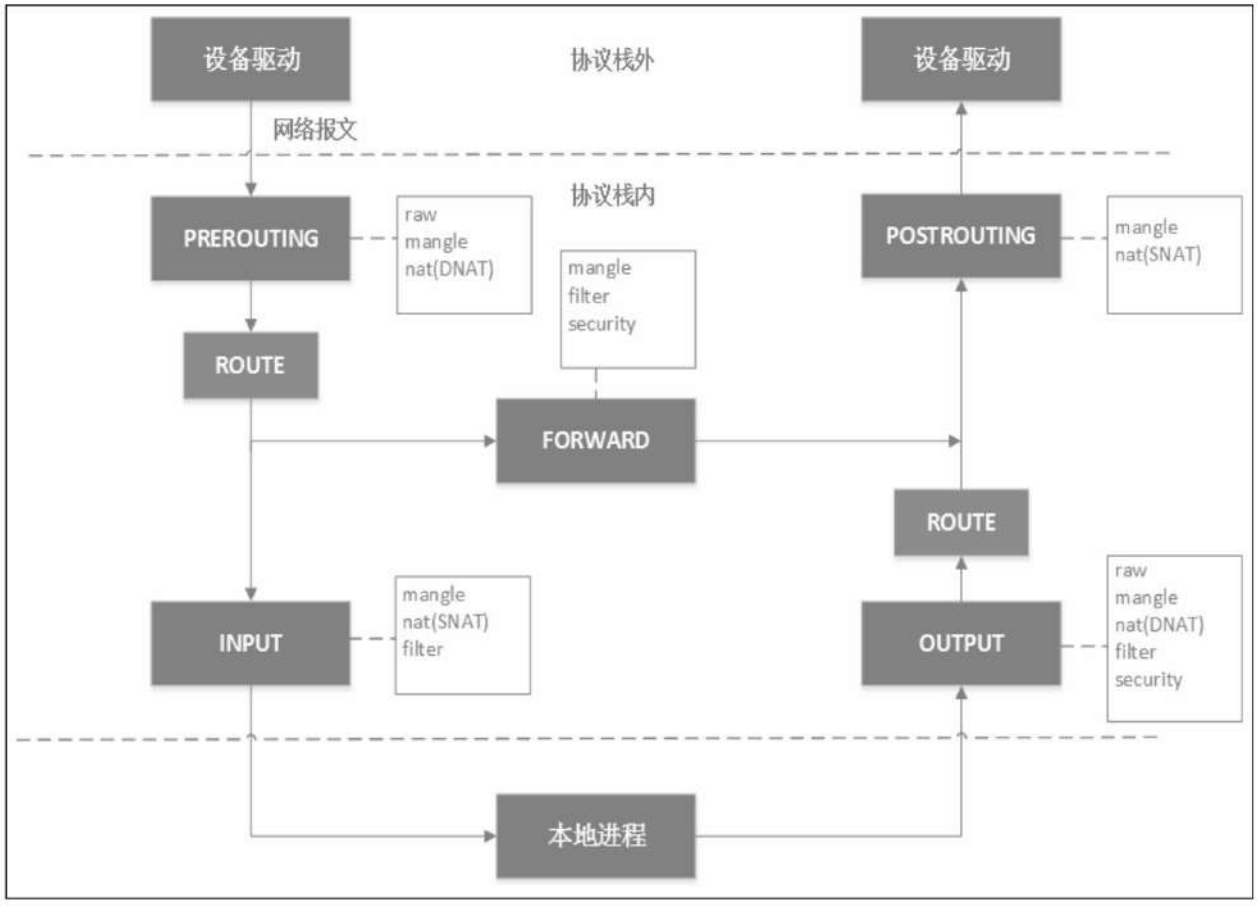


图1-16　一个网络包经过iptables的处理路径

可能有读者会问，iptables的链（chain）的概念还比较好理解，就是对应netfilter的5处钩子，而iptables的表（table）的具体作用是什么呢？而且上

文说的“不是每个链上都能挂表”这句话的表述似乎也很奇怪。

任何概念都不是凭空想象出来的，它都是有实际用途的。其实，iptables的表是来分类管理iptables规则（rule）的，系统所有的iptables规则都被划分到不同的表集合中。上文也提到了，filter表中会有过滤数据包的规则，nat表中会有做地址转换的规则。因此，iptables的规则就是挂在netfilter钩子上的函数，用来修改数据包的内容或过滤数据包，iptables的表就是所有规则的5个逻辑集合！下面就让我们见识iptables的规则吧。

iptables的规则是用户真正要书写的规则。一般情况下，一条iptables规则包含两部分信息：匹配条件和动作。匹配条件很好理解，即匹配数据包被这条iptables规则“捕获”的条件，例如协议类型、源IP、目的IP、源端口、目的端口、连接状态等。每条iptables规则允许多个匹配条件任意组合，从而实现多条件的匹配，多条件之间是逻辑与（&&）关系。

那么，数据包匹配后该怎么办，常见的动作有下面几个：

·DROP：直接将数据包丢弃，不再进行后续的处理。应用场景是不让某个数据源意识到你的系统的存在，可以用来模拟宕机；

·REJECT：给客户端返回一个connection refused或destination unreachable报文。应用场景是不让某个数据源访问你的系统，善意地告诉他：我这里没有你要的服务内容；

·QUEUE：将数据包放入用户空间的队列，供用户空间的程序处理；

·RETURN：跳出当前链，该链里后续的规则不再执行；

·ACCEPT：同意数据包通过，继续执行后续的规则；

·JUMP：跳转到其他用户自定义的链继续执行。

值得一提的是，用户自定义链中的规则和系统预定义的5条链里的规则没有区别。由于自定义的链没有与netfilter里的钩子进行绑定，所以它不会自动触发，只能从其他链的规则中跳转过来，这也是JUMP动作存在的意义。

在初步认识了iptables的表、链和规则三个最重要的概念后，我们介绍iptables命令的常见用法。

1.5.3 iptables的常规武器

在见识了iptables的“三板斧”之后，相信读者已经对iptables的工作机制了然于胸，接下来将以iptables在Kubernetes网络中的使用为背景，简单介绍iptables的常见用法。本节的目的是为后面理解Kubernetes网络，尤其是

Service部分做铺垫。

1.查看所有**iptables**规则

Kubernetes一个节点上的iptables规则输出如下。

```
# iptables -L -n

Chain INPUT (policy ACCEPT)
target     prot opt source               destination
KUBE-FIREWALL  all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     udp  --  0.0.0.0/0            0.0.0.0/0           udp dpt:60129

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
KUBE-FORWARD  all  --  0.0.0.0/0            0.0.0.0/0            /* kubernetes
forwarding rules */
REJECT     all  --  0.0.0.0/0            0.0.0.0/0           reject-with icmp-port-
unreachable

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     udp  --  0.0.0.0/0            0.0.0.0/0           udp dpt:68

Chain KUBE-FIREWALL (1 references)
target     prot opt source               destination
DROP       all  --  0.0.0.0/0            0.0.0.0/0           /* kubernetes firewall
for dropping marked packets */ mark match 0x8000/0x8000

Chain KUBE-FORWARD (1 references)
target     prot opt source               destination
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0           /* kubernetes
forwarding rules */ mark match 0x4000/0x4000
```

严格地说，输出是iptables的filter表的所有规则。使用iptables命令，必须指定针对哪个表进行操作，默认是filter表。因此，如果想输出系统中nat表的所有iptables规则，可以使用如下命令：

```
# iptables -t nat -L -n
```

使用-n选项将以数字形式列出信息，即将域名解析成IP地址。想输出更详细的信息，例如，经过某条rule处理的数据包字节、进出网口等信息，可以使用-v选项。

从上面的输出结果可以看出，filter表上挂了5条链，分别是INPUT、FORWARD和OUTPUT这三条系统内置链，以及KUBE-FIREWALL和KUBE-FORWARD这两条用户自定义链。iptables的内置链都有默认规则，例如在我们的例子中，INPUT、FORWARD和OUTPUT的默认规则是ACCEPT，即全部放行。用户自己定义的链后面都有一个引用计数，在我们的例子中KUBE-FIREWALL和KUBE-FORWARD都有一个引用，它们分别在INPUT和FORWARD中被引用。iptables的每条链下面的规则处理顺序是从上到下逐条遍历的，除非中途碰到DROP，REJECT，RETURN这些内置动作。如果iptables规则前面是自定义链，则意味着这条规则的动作是JUMP，即跳到这条自定义链遍历其下的所有规则，然后跳回来遍历原来那条链后面的规则。以上过程如图1-17所示。

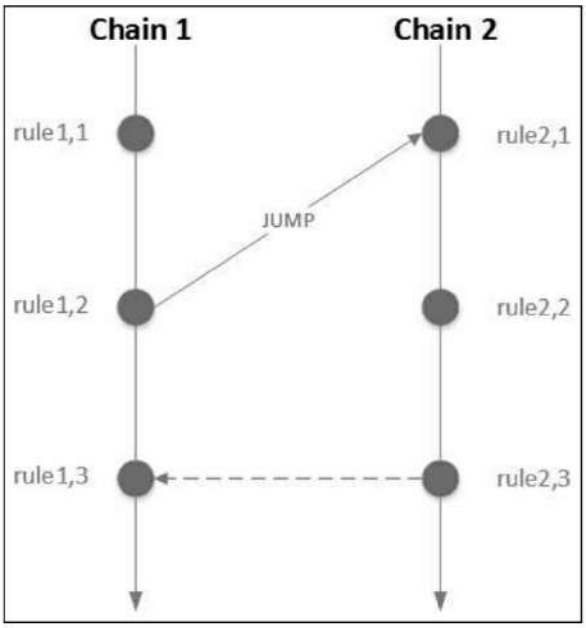


图1-17　iptables遍历规则

在我们的例子中，iptables规则的遍历顺序应该是rule　1，1→rule　1，2→rule 2，1→rule 2，2→rule 2，3→rule 1，3。

2.配置内置链的默认策略

当然，我们可以自己配置内置链的默认策略，决定是放行还是丢弃，如下所示：

```
## 默认的不让进
# iptables --policy INPUT DROP
## 默认的不允许转发
# iptables --policy FORWARD DROP
```

```
## 默认的可以出去
# iptables --policy OUTPUT ACCEPT
```

3.配置防火墙规则策略

防火墙策略一般分为通和不通两种。如果默认策略是“全通”，例如上文的policy ACCEPT，就要定义一些策略来封堵；反之，如果默认策略是“全不通”，例如上文的policy DROP，就要定义一些策略来解封。如果是做访问控制列表（ACL），即俗称的白名单，则用解封策略更常用。我们将用几个实际的例子来说明。

1）配置允许SSH连接

```
# iptables -A INPUT -s 10.20.30.40/24 -p tcp --dport 22 -j ACCEPT
```

简单分析这条命令，-A的意思是以追加（Append）的方式增加这条规则。-A INPUT表示这条规则挂在INPUT链上。-s 10.20.30.40/24表示允许源（source）地址是10.20.30.40/24这个网段的连接。-p tcp表示允许TCP（protocol）包通过。--dport 22的意思是允许访问的目的端口（destination port）为22，即SSH端口。-j ACCEPT表示接受这样的连接。综上所述，这条iptables规则的意思是允许源地址是10.20.30.40/24这个网段的包发到本地TCP 22端口。除了按追加的方式添加规则，还可以使用iptables-I［chain］［number］将规则插入（Insert）链的指定位置。如果不指定number，则插到链的第一条处。

2）阻止来自某个IP/网段的所有连接

如果要阻止10.10.10.10上所有的包，则可以使用以下命令：

```
# iptables -A INPUT -s 10.10.10.10 -j DROP
```

也可以使用-j REJECT，这样就会发一个连接拒绝的回程报文，客户端收到后立刻结束。不像-j DROP那样不返回任何响应，客户端只能一直等待直到请求超时。如果要屏蔽一个网段，例如10.10.10.0/24，则可以使用-s 10.10.10.0/24或10.10.10.0/255.255.255.0。如果要“闭关锁国”，即屏蔽所有的

外来包，则可以使用-s 0.0.0.0/0。

3）封锁端口

要阻止从本地1234端口建立对外连接，可以使用以下命令：

```
iptables -A OUTPUT -p tcp --dport 1234 -j DROP
```

注意，我们要在OUTPUT链上挂规则是因为我们的需求是屏蔽本地进程对外的连接。如果我们要阻止外部连接访问本地1234端口，就需要在INPUT链上挂规则，命令如下：

```
iptables -A INPUT -p tcp --dport 1234 -j DROP
```

感到困惑的读者可以复习关于netfilter的内容。

4）端口转发

有时，我们要把服务器的某个端口流量转发给另一个端口。例如，我们对外声称Web服务在80端口上运行，但由于种种原因80端口被人占了，实际的Web服务监听在8080端口上。为了让外部客户端能无感知地依旧访问80端口，可以使用以下命令实现端口转发：

```
iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080
```

以上命令中-p tcp--dport 80-j REDIRECT--to-port 8080的意思是发到TCP 80端口的流量转发（REDIRECT）给8080端口，iptables并不会修改任何IP地址或者协议。需要注意的是，我们修改了目的端口，因此该规则应该发生在nat表上的PREROUTING链上。至于-i eth0则表示匹配eth0网卡上接收到的包。

5）禁用PING

大部分的公有云默认都是屏蔽ICMP的，即禁止ping报文。具体是如何实现的呢？请看下面这条命令：

```
iptables -A INPUT -p icmp -j DROP
```

以上命令的-p icmp-j DROP即表示匹配ICMP报文，然后丢弃。

6）删除规则

最暴力地清除当前所有的规则命令（请慎用）：

```
# iptables -F
```

要清空特定的表可以使用-t参数进行指定，例如清除nat表所有的规则如下：

```
# iptables -t nat -F
```

删除规则的最直接的需求是解封某条防火墙策略。因此，笔者建议使用iptables的-D参数：

```
# iptables -D INPUT -s 10.10.10.10 -j DROP
```

-D表示从链中删除一条或多条指定的规则，后面跟的就是要删除的规则。

当某条链上的规则被全部清除变成空链后，可以使用-X参数删除：

```
# iptables -X FOO
```

以上命令删除FOO这条用户自定义的空链，但需要注意的是系统内置链无法删除。

7）自定义链

说了这么多自定义链，具体该怎么创建它呢？请看以下命令：

```
# iptables -N BAR
```

如上所示，我们在filter表（因为未指定表，所以可以使用-t参数指定）创建了一条用户自定义的链BAR。

4.DNAT

DNAT根据指定条件修改数据包的目标IP地址和目标端口。DNAT的原理和我们上文讨论的端口转发原理差不多，差别是端口转发不修改IP地址。使用iptables做目的地址转换的一个典型例子如下：

```
#iptables -t nat -A PREROUTING -d 1.2.3.4 -p tcp --dport 80 -j DNAT --to-destination
10.20.30.40:8080
```

-j DNAT表示目的地址转换，-d 1.2.3.4-p tcp--dport 80表示匹配的包条件是访问目的地址和端口为1.2.3.4:80的TCP包，--to-destination表示将该包的目的地址和端口修改成10.20.30.40:8080。同样，DNAT不修改协议。如果要匹配网卡，可以用-i eth0指定收到包的网卡（i是input的缩写）。需要注意的是，DNAT只发生在nat表的PREROUTING链，这也是我们要指定收到包的网卡而不是发出包的网卡的原因。

需要注意的是，当涉及转发的目的IP地址是外机时，需要确保启用ip forward功能，即把Linux当交换机用，命令如下：

```
echo 1 >/proc/sys/net/ipv4/ip_forward
```

5.SNAT/网络地址欺骗

神秘的网络地址欺骗其实是SNAT的一种。SNAT根据指定条件修改数据包的源IP地址，即DNAT的逆操作。与DNAT的限制类似，SNAT策略只能发生在nat表的POSTROUTING链。具体命令如下：

```
# iptables -t nat -A POSTROUTING -s 192.168.1.2 -j SNAT --to-source 10.172.16.1
```

-j SNAT表示源地址转换，-s 192.168.1.12表示匹配的包源地址是192.168.1.12，--to-source表示将该包的源地址修改成10.172.16.1。与DNAT类似，我们也可以匹配发包的网卡，例如-o eth0（o是output的缩写）。

至于网络地址伪装，其实就是一种特殊的源地址转换，报文从哪个网卡出就用该网卡上的IP地址替换该报文的源地址，具体用哪个IP地址由内核决定。下面这条规则的意思是：源地址是10.8.0.0/16的报文都做一次Masq。

```
# iptables-t nat -A POSTROUTING -s 10.8.0.0/16 -j MASQUERADE
```

与SNAT类似，如果要控制被替换的源地址，则可以指定匹配从哪块网卡发出报文。例如：-o eth0选项指定报文从eth0出去并使用eth0的IP地址做源地址伪装。

6.保存与恢复

上述方法对iptables规则做出的改变是临时的，重启机器后就会丢失。如果想永久保存这些更改，则需要运行以下命令：

```
# iptables-save
```

iptables-save在保存系统所有iptables规则的同时，也会在标准输出打印这些规则，如有需要可以重定向到一个文件，例如：

```
# iptables-save > iptables.bak
```

后续，我们可以使用iptables-restore命令还原iptables-save命令备份的iptables配置，原理就是逐条执行文件里的iptables规则，如下所示：

```
# iptables-restore < iptables.bak
```

1.6 初识Linux隧道：ipip

前文介绍的tun设备也叫作点对点设备，之所以叫这个名字，是因为tun经常被用来做隧道通信（tunnel）。

我们可以通过命令ip tunnel help查看IP隧道的相关操作。Linux原生支持下列5种L3隧道：

·ipip：即IPv4 in IPv4，在IPv4报文的基础上封装一个IPv4报文；

·GRE：即通用路由封装（Generic Routing Encapsulation），定义了在任意一种网络层协议上封装其他任意一种网络层协议的机制，适用于IPv4和IPv6；

·sit：和ipip类似，不同的是sit用IPv4报文封装IPv6报文，即IPv6 over IPv4；

·ISATAP：即站内自动隧道寻址协议（Intra-Site Automatic Tunnel Addressing Protocol），与sit类似，也用于IPv6的隧道封装；

·VTI：即虚拟隧道接口（Virtual Tunnel Interface），是思科提出的一种IPSec隧道技术。下面我们以ipip为例，介绍Linux隧道通信的基本原理。

注：Linux L3隧道底层实现原理都基于tun设备，因此我们可以将本节看作tun设备的高级应用篇。

1.6.1 测试ipip隧道

要使用ipip隧道，首先需要内核模块ipip.ko的支持。

通过lsmod|grep ipip查看内核是否加载，若没有则用modprobe ipip加载，正常加载应该显示：

```
# modprobe ipip
# lsmod | grep ipip
ipip                   13465  0
tunnel4                13252  1 ipip
ip_tunnel              25163  1 ipip
```

加载ipip内核模块后，就可以创建隧道了。方法是先创建一个tun设备，然后将该tun设备绑定为一个ipip隧道。ipip隧道网络拓扑如图1-18所示。

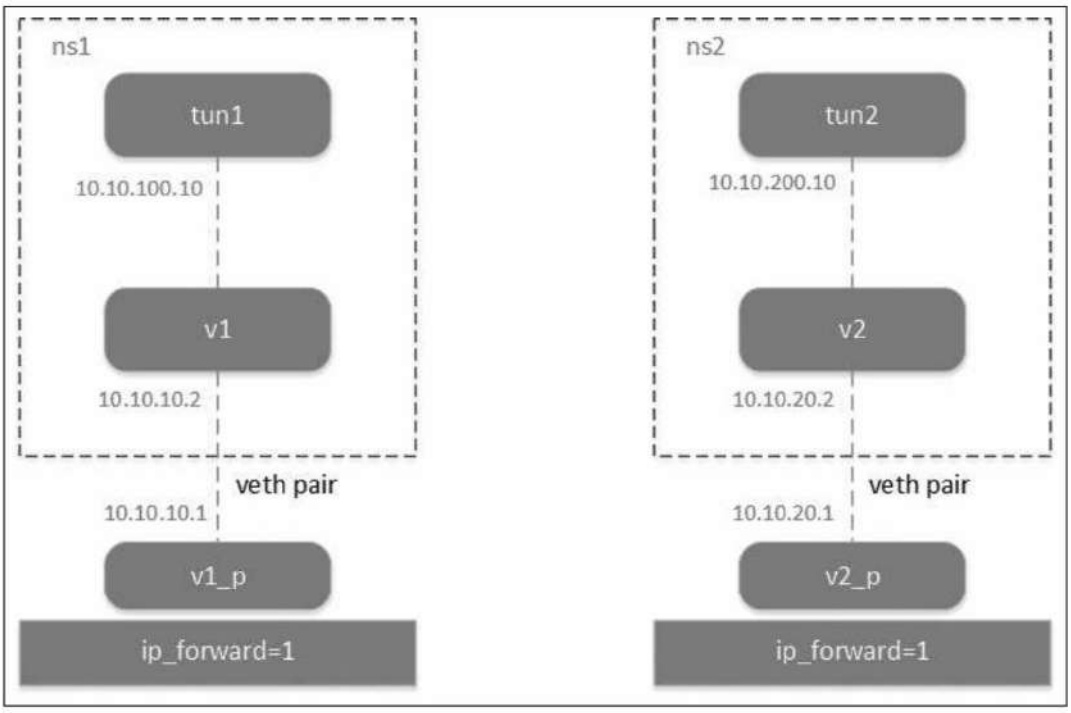


图1-18　ipip隧道网络拓扑

创建两个network namespace：

```
# ip netns add ns1
# ip netns add ns2
```

创建两对veth pair，令其一端挂在某个namespace下：

```
# ip link add v1 type veth peer name v1_p
# ip link add v2 type veth peer name v2_p

# ip link set v1 netns ns1
# ip link set v2 netns ns2
```

分别给两对veth-pair端点配上IP并启用：

```
# ip addr add 10.10.10.1/24 dev v1_p
# ip link set v1_p up
# ip addr add 10.10.20.1/24 dev v2_p
# ip link set v2_p up

# ip netns exec ns1 ip addr add 10.10.10.2/24 dev v1
# ip netns exec ns1 ip link set v1 up
# ip netns exec ns2 ip addr add 10.10.20.2/24 dev v2
# ip netns exec ns2 ip link set v2 up
```

验证一下：v1 ping v2，结果为不通。

检查ip_forward的值：

```
# cat /proc/sys/net/ipv4/ip_forward
0
```

我们已经知道Linux本身就是一台路由器，Linux提供开关/proc/sys/net/ipv4/ip_forward来操作路由功能，默认这个开关是关的，打开只需：

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

这种打开方式只是临时的，若想持久，可以修改配置文件/etc/sysctl.conf，添加或修改项net.ipv4.ip_forward为：

```
net.ipv4.ip_forward = 1
```

在我们的例子中，即使打开了ip_forward选项，v1到v2依然不通。

查看ns1的路由表：

```
# ip netns exec ns1 route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.10.10.0      0.0.0.0         255.255.255.0   U     0      0        0 v1
```

只有一条直连路由，没有通往10.10.20.0/24网段的路由，因此手动配置

一条路由，如下所示：

```
# ip netns exec ns1 route add -net 10.10.20.0 netmask 255.255.255.0 gw 10.10.10.1
```

再查看路由表：

```
# ip netns exec ns1 route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.10.10.0      0.0.0.0         255.255.255.0   U     0      0        0 v1
10.10.20.0      10.10.10.1      255.255.255.0   UG    0      0        0 v1
```

同理，也给ns2配上通往10.10.10.0/24网段的路由。

再ping一次，发现通了！

```
# ip netns exec ns1 ping 10.10.20.2
PING 10.10.20.2 (10.10.20.2) 56(84) bytes of data.
64 bytes from 10.10.20.2: icmp_seq=1 ttl=63 time=0.071 ms
64 bytes from 10.10.20.2: icmp_seq=2 ttl=63 time=0.070 ms
^C
--- 10.10.20.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.070/0.070/0.071/0.008 ms
```

保证v1和v2能够通信后，再创建tun设备，并设置为ipip隧道。

在ns1上创建tun1和ipip tunnel：

```
# ip netns exec ns1 ip tunnel add tun1 mode ipip remote 10.10.20.2 local 10.10.10.2
# ip netns exec ns1 ip link set tun1 up
# ip netns exec ns1 ip addr add 10.10.100.10 peer 10.10.200.10 dev tun1
```

上面的命令是在ns1上创建tun设备tun1，并设置隧道模式为ipip，然后设置隧道端点，用remote和local表示，这是隧道外层IP。对应的还有隧道内层IP，用ip addr xx peer xx配置。从tun1发到tun2的原始IP报文和经过隧道封装后的IP报文如图1-19所示。

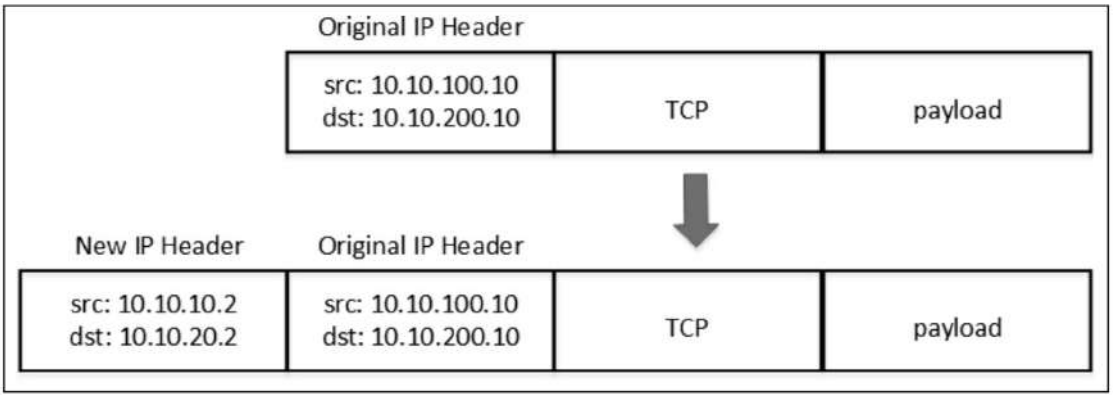


图1-19　IP报文封装示意图

同理，我们也在ns2上创建tun2和ipip tunnel。

```
# ip netns exec ns2 ip tunnel add tun2 mode ipip remote 10.10.10.2 local 10.10.20.2
# ip netns exec ns2 ip link set tun2 up
# ip netns exec ns2 ip addr add 10.10.200.10 peer 10.10.100.10 dev tun2
```

完成上述配置，两个tun设备端点就可以互通了，代码如下：

```
# ip netns exec ns1 ping 10.10.200.10 -c 4
PING 10.10.200.10 (10.10.200.10) 56(84) bytes of data.
64 bytes from 10.10.200.10: icmp_seq=1 ttl=64 time=0.090 ms
64 bytes from 10.10.200.10: icmp_seq=2 ttl=64 time=0.148 ms
64 bytes from 10.10.200.10: icmp_seq=3 ttl=64 time=0.112 ms
64 bytes from 10.10.200.10: icmp_seq=4 ttl=64 time=0.110 ms

--- 10.10.200.10 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.090/0.115/0.148/0.020 ms
```

1.6.2　ipip隧道测试结果复盘

我们试着分析上面的实验过程：

（1）ping命令构建一个ICMP请求，ICMP报文封装在IP报文中，源和目的IP地址分别是10.10.100.10和10.10.200.10。

（2）由于tun1和tun2不在同一网段，所以要查看路由表。通过ip　tunnel命令建立ipip隧道后，会自动生成一条路由，如下所示：

```
# ip netns exec ns1 route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.10.10.0      0.0.0.0         255.255.255.0   U     0      0        0 v1
10.10.20.0      10.10.10.1      255.255.255.0   UG    0      0        0 v1
10.10.200.10    0.0.0.0         255.255.255.255 UH    0      0        0 tun1
```

以上路由表信息表明：去往目的地10.10.200.10的报文直接从tun1出去了。

（3）由于配置了隧道端点，数据包出了tun1，直接到达v1。根据ipip隧道的配置，会封装上一层新的IP头，源和目的IP地址分别为10.10.10.2和10.10.20.2。

（4）由于v1和v2同样不在一个网段，查看路由表，发现去往10.10.20.0网段的报文从v1网卡出，去往10.10.10.1网关，即veth pair在主机上的另一端v1_p。

（5）Linux打开了ip_forward，它相当于一台路由器，10.10.10.0和10.10.20.0是两条直连路由，所以直接查路由表转发，从这台主机转到另一台主机的v2_p上。

（6）根据veth pair的设备特性，数据包到达另一台主机的v2_p上，会直接从ns2的v2出来。内核解封装数据包，发现内层IP报文的目的IP地址是10.10.200.10，这正是自己配置的ipip隧道的tun2地址，于是将报文交给tun2设备。至此，tun1的ping请求包成功到达tun2。

（7）由于ICMP报文的传输特性，有去必有回，所以ns2上会构造ICMP响应报文，并根据以上相同步骤封装和解封装数据包，直至到达tun1，整个ping过程完成。

以上便是ipip隧道的通信过程，感兴趣的读者可以自行抓包验证。读者会发现有两层IP报文头，外层使用ipip协议构成隧道的端点，内层是正常的通信报文，封装了ICMP报文作为payload。

1.6.3 小结

现在的Linux内核原生支持5种隧道协议，它们的底层实现都采用tun设备。我们熟知的各种VPN软件，其底层实现都离不开这5种隧道协议。其他隧道实现方式与ipip隧道的大同小异。

1.7 Linux隧道网络的代表：VXLAN

VXLAN（Virtual eXtensible LAN，虚拟可扩展的局域网），是一种虚拟化隧道通信技术。它是一种overlay（覆盖网络）技术，通过三层的网络搭建虚拟的二层网络。RFC7348中是这样介绍VXLAN的：

A framework for overlaying virtualized layer 2 networks over lay 3 networks.

简单来讲，VXLAN是在底层物理网络（underlay）之上使用隧道技术，依托UDP层构建的overlay的逻辑网络，使逻辑网络与物理网络解耦，实现灵活的组网需求。它不仅能适配虚拟机环境，还能用于容器环境。由此可见，VXLAN这类隧道网络的一个特点是对原有的网络架构影响小，不需要对原网络做任何改动，就可在原网络的基础上架设一层新的网络。

不同于其他隧道协议，VXLAN是一个一对多的网络，并不仅是一对一的隧道协议。一个VXLAN设备能通过像网桥一样的学习方式学习到其他对端的IP地址，也可以直接配置静态转发表。

一个典型的数据中心VXLAN网络拓扑图如图1-20所示。

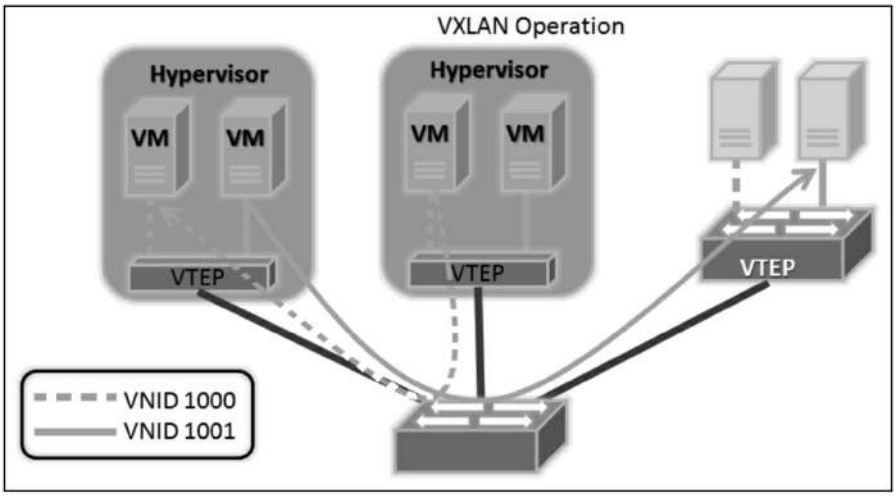


图1-20　VXLAN网络拓扑图

图1-20中的VM指的是虚拟机，Hypervisor指的是节点的虚拟机管理器。VXLAN不仅能用在基于虚拟机的虚拟化系统中，还被广泛应用于容器集群。

1.7.1 为什么需要VXLAN

相信不少读者知道VLAN，VLAN技术的缺陷是VLAN Header预留的长

度只有12 bit，故最多只能支持2的12次方（4096）子网的划分，无法满足云计算场景下主机数量日益增长的需求。VXLAN能突破VLAN的最多4096个子网的数量限制，以满足大规模云计算数据中心的需求。具体来说，VXLAN技术解决以下几个问题：

·当前VXLAN的报文Header内有24 bit，可以支持2的24次方个子网，并通过VNI（Virtual Network Identifier）区分不同的子网，相当于VLAN中的LAN ID；

·多租网络隔离。越来越多的数据中心（尤其是公有云服务）需要提供多租户的功能，不同用户之间需要独立地分配IP和MAC地址。如何保证这个功能的扩展性和正确性是待解决的问题；

·云计算业务对业务灵活性要求很高，虚拟机可能会大规模迁移，并保证网络一直可用，也就是大二层的概念。解决这个问题同时保证二层的广播域不会过分扩大，这也是云计算网络的要求。

传统的二层或三层的网络在应对这些挑战时显得捉襟见肘，虽然很多改进型的技术能够增加二层的范围，但是要做到对网络改动尽量小的同时保证灵活性非常困难。

1.7.2 VXLAN协议原理简介

VXLAN隧道网络方案相比改造传统的二层或三层网络，对原有的网络架构影响小。隧道网络不需要原来的网络做任何改动，可直接在原来的网络基础上架设一层新的网络。

图1-21所示为VXLAN的工作模型，它创建在原来的IP网络（三层）上，只要是三层可达（能够通过IP互相通信）的网络就能部署VXLAN。在VXLAN网络的每个端点都有一个VTEP设备，负责VXLAN协议报文的封包和解包，也就是在虚拟报文上封装VTEP通信的报文头部。物理网络上可以创建多个VXLAN网络，可以将这些VXLAN网络看作一个隧道，不同节点上的虚拟机/容器能够通过隧道直连。通过VNI标识不同的VXLAN网络，使得不同的VXLAN可以相互隔离。

VXLAN的几个重要概念如下：

·VTEP（VXLAN Tunnel Endpoints）：VXLAN网络的边缘设备，用来进行VXLAN报文的处理（封包和解包）。VTEP可以是网络设备（例如交换机），也可以是一台机器（例如虚拟化集群中的宿主机）；

·VNI（VXLAN Network Identifier）：VNI是每个VXLAN的标识，是个24位整数，因此最大值是2^{24}=16777216。如果一个VNI对应一个租户，那么

理论上VXLAN可以支撑千万级别的租户。

·tunnel：隧道是一个逻辑上的概念，在VXLAN模型中并没有具体的物理实体相对应。隧道可以看作一种虚拟通道，VXLAN通信双方（图中的虚拟机）都认为自己在直接通信，并不知道底层网络的存在。从整体看，每个VXLAN网络像是为通信的虚拟机搭建了一个单独的通信通道，也就是隧道。

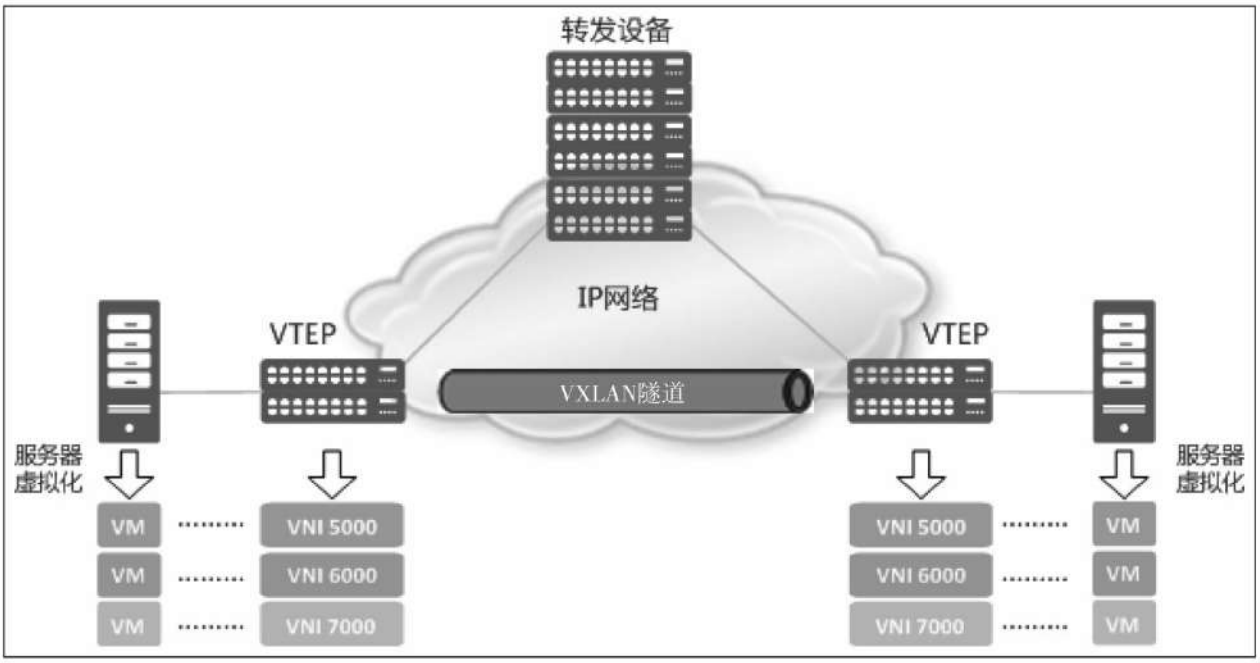


图1-21　VXLAN的工作模型

前文提到，VXLAN其实是在三层网络上构建出来的一个二层网络的隧道。VNI相同的机器逻辑上处于同一个二层网络中。VXLAN封包格式如图1-22所示。

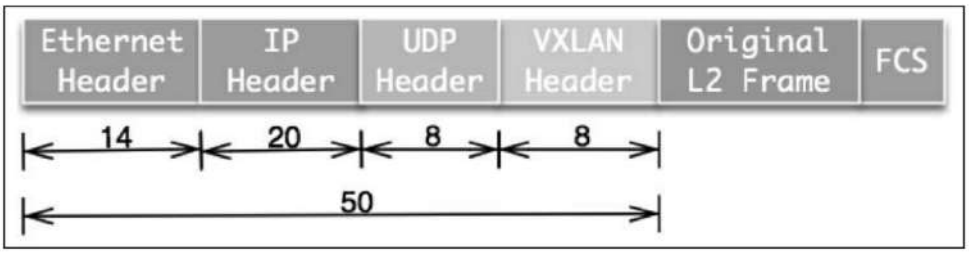


图1-22　VXLAN封包格式

VXLAN的报文就是MAC in UDP，即在三层网络的基础上构建一个虚拟的二层网络。为什么这么说呢？VXLAN的封包格式显示原来的二层以太网帧（包含MAC头部、IP头部和传输层头部的报文），被放在VXLAN包头里进行封装，再套到标准的UDP头部（UDP头部、IP头部和MAC头部），用来在底层网络上传输报文。

可以看出，VXLAN报文比原始报文多出了50个字节，包括8个字节的

VXLAN协议相关的部分，8个字节的UDP头部，20个字节的IP头部和14个字节的MAC头部。这降低了网络链路传输有效数据的比例，特别是对小包。

需要注意的是，UDP目的端口是接收方VTEP设备使用的端口，IANA（Internet Assigned Numbers Authority，互联网号码分配局）分配了4789作为VXLAN的目的UDP端口。

VXLAN的配置管理使用iproute2包，这个工具是和VXLAN一起合入内核的，我们常用的ip命令就是iproute2的客户端工具。VXLAN要求Linux内核版本在3.7以上，最好为3.9以上，所以在一些旧版本的Linux上无法使用基于VXLAN的封包技术。

1.7.3 VXLAN组网必要信息

总的来说，VXLAN报文的转发过程就是：原始报文经过VTEP，被Linux内核添加上VXLAN包头及外层的UDP头部，再发送出去，对端VTEP接收到VXLAN报文后拆除外层UDP头部，并根据VXLAN头部的VNI把原始报文发送到目的服务器。

以上过程看似并不复杂，但前提是通信双方已经知道所有通信信息。第一次通信之前有以下问题待解决：

·哪些VTEP需要加到一个相同的VNI组？

·发送方如何知道对方的MAC地址？

·如何知道目的服务器在哪个节点上？

第一个问题最好回答，VTEP通常由网络管理员进行配置。另外两个问题可以归结为同一个问题：VXLAN网络的通信双方如何感知彼此并选择正确的路径传输报文？要回答这两个问题，还得回到VXLAN协议报文上，看看一个完整的VXLAN报文需要哪些信息。

·内层报文：通信双方的IP地址已经明确，需要VXLAN填充的是对方的MAC地址，VXLAN需要一个机制来实现ARP的功能；

·VXLAN头部：只需要知道VNI。它一般是直接配置在VTEP上的，即要么是提前规划的，要么是根据内部报文自动生成的；

·UDP头部：最重要的是源地址和目的地址的端口，源地址端口是由系统生成并管理的，目的端口一般固定为IANA分配的4789端口；

·IP头部：IP头部关心的是对端VTEP的IP地址，源地址可以用很简单的方式确定，目的地址是虚拟机所在地址宿主机VTEP的IP地址，需要由某种方式来确定；

·MAC头部：确定了VTEP的IP地址，MAC地址可以通过经典的ARP方式获取，毕竟VTEP在同一个三层网络内。

总结一下，一个VXLAN报文需要确定两个地址信息：内层报文（对应目的虚拟机/容器）的MAC地址和外层报文（对应目的虚拟机/容器所在宿主机上的VTEP）IP地址。如果VNI也是动态感知的，那么VXLAN一共需要知道三个信息：内部MAC、VTEP IP和VNI。

一般有两种方式获得以上VXLAN网络的必要信息：多播和控制中心。多播的概念是同一个VXLAN网络的VTEP加入同一个多播网络。如果需要知道以上信息，就在组内发送多播来查询。控制中心的概念是在某个集中式的地方保存所有虚拟机的上述信息，自动告知VTEP它需要的信息即可。

1.7.4 VXLAN基本配置命令

VXLAN接口的基本管理如下。

1.创建**VXLAN**接口

```
#ip link add VXLAN0 type VXLAN id 42 group 239.1.1.1 dev eth0 dstport 4789
```

这条命令会创建一个叫作VXLAN0的新接口，它使用在eth0上的多播组239.1.1.1通信。初始化时没有转发表。目的端口号是IANA规定的4789。在VXLAN中，一般将VXLAN接口（在我们的例子中即vxlan0）叫作VTEP（VXLAN tunnel endpoint），VXLAN子网的报文需要从VTEP出去。

多播组主要通过ARP洪泛来学习MAC地址，即在VXLAN子网内广播ARP请求，然后对应节点进行响应。但多播不是VXLAN所必需的，下文会进行解释。

如果网络不复杂，则可以认为某一节点上所有子网IP的MAC和节点上的VTEP的MAC一致，直接用VTEP MAC封装报文。至于VTEP的MAC地址，可以用bridge命令手工配置。

2.删除**VXLAN**接口

与普通网络接口一样，ip link delete命令就能删除一个VXLAN接口：

```
# ip link delete VXLAN0
```

3.查看**VXLAN**接口信息

常规命令ip-d link show可以查看VXLAN网卡信息：

```
# ip -d link show VXLAN0
```

4.VXLAN转发表

可以使用bridge命令创建、删除或查看VXLAN接口的转发表。

创建一条转发表项：

```
# bridge fdb add to 00:17:42:8a:b4:05 dst 192.19.0.2 dev VXLAN0
```

00:17:42:8a:b4:05即对端VTEP的MAC地址，192.19.0.2即对端VTEP的IP地址。

删除一条转发表项：

```
# bridge fdb delete 00:17:42:8a:b4:05 dev VXLAN0
```

查看VXLAN接口的转发表：

```
# bridge fdb show dev VXLAN0
```

注：网络设备都以MAC地址唯一地标识自己，而交换机要实现设备之间的通信就必须知道自己的哪个端口连接着哪台设备，因此就需要一张MAC地址与端口号一一对应的表，以便在交换机内部实现二层数据转发。这张二层转发表就是FDB表。它主要由MAC地址、VLAN号、端口号和一些标志域等信息组成。如果收到数据帧的目的MAC地址不在FDB地址表中，那么该数据将被发送给除源端口外，该数据包所属VLAN中的其他所有端口。把数据发给其他所有端口的行为称为洪泛。

那么，FDB表是怎么形成的呢？很简单，交换机会在收到数据帧时，提取数据帧中的源MAC、VLAN和接收数据帧的端口，组成FDB表的条目。当下次看到相同VLAN时，相同MAC地址的报文就直接从记录的端口“丢”出去。

1.7.5 VXLAN网络实践

了解了VXLAN必要的背景知识后，我们将通过几个例子说明如何搭建基于VXLAN的overlay网络，顺便阐述上文提到的多播和控制中心这两种方式的原理。

1.点对点的**VXLAN**

让我们先从最简单的点对点的VXLAN网络说起。点对点VXLAN即两台机器构成一个VXLAN网络，每台机器上有一个VTEP，VTEP之间通过它们

的IP地址进行通信。点对点VXLAN网络拓扑如图1-23所示。

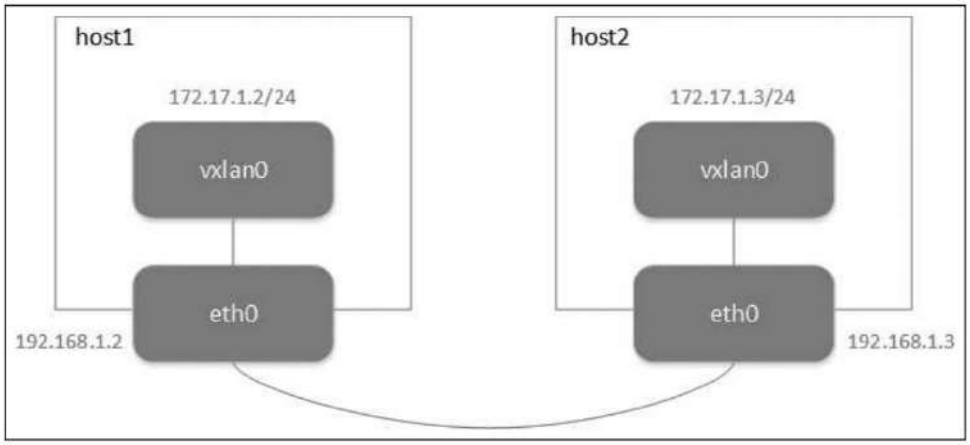


图1-23　点对点VXLAN网络拓扑图

首先，使用ip link命令创建VXLAN接口：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    remote 192.168.1.3 \
    local 192.168.1.2 \
    dev eth0
```

上面这条命令创建了一个名为vxlan0，类型为vxlan的网络接口，一些重要的参数如下：

·id 42：指定VNI的值，有效值在1到2^{24}之间；

·dstport：VTEP通信的端口，IANA分配的端口是4789。如果不指定，Linux默认使用8472；

·remote 192.168.1.3：对端VTEP的地址；

·local 192.168.1.2：当前节点VTEP要使用的IP地址，即当前节点隧道口的IP地址；

·dev eth0：当前节点用于VTEP通信的网卡设备，用来获取VTEP IP地址。注意，这个参数和local参数含义相同，读者在使用过程中二选一即可。

执行完之后，系统就会创建一个名字为vxlan0的网卡，可以用ip-d link命令查看它的详细信息：

```
# ip -d link show dev vxlan0
...
4: vxlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1000
```

```
    link/ether ba:d4:0a:f8:41:7c brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 42 remote 192.168.1.3 dev eth0 srcport 0 0 dstport 4789 ageing 300
addrgenmode eui64
```

接下来，为刚创建的VXLAN网卡配置IP地址并启用它：

```
# ip addr add 172.17.1.2/24 dev vxlan0
# ip link set vxlan0 up
```

执行成功后会发现路由表项多了下面的内容，所有目的地址是172.17.1.0/24网段的包要通过vxlan0转发：

```
# ip route
172.17.1.0/24 dev vxlan0  proto kernel  scope link  src 172.17.1.2
```

同时，vxlan0的FDB表项中的内容如下：

```
# bridge fdb
00:00:00:00:00:00 dev vxlan0 dst 192.168.1.3 via eth0 self permanent
```

这个表项的意思是，默认的VTEP对端地址为192.168.1.3。换句话说，原始报文经过vxlan0后会被内核添加上VXLAN头部，而外部UDP头的目的IP地址会被冠上192.168.1.3。

在另外一台机器上（192.168.1.3）也进行相同的配置，要保证VNI也是42，dstport也是4789，并修改VTEP的local和remote IP地址到相应的值。测试两个VTEP的连通性，如下所示：

```
# ping -c 3 172.17.1.3
PING 172.17.1.3 (172.17.1.3) 56(84) bytes of data.
64 bytes from 172.17.1.3: icmp_seq=1 ttl=64 time=1.84 ms
64 bytes from 172.17.1.3: icmp_seq=2 ttl=64 time=0.462 ms
64 bytes from 172.17.1.3: icmp_seq=3 ttl=64 time=0.427 ms

--- 172.17.1.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.427/0.911/1.844/0.659 ms
```

通过点对点组网模型实践了VXLAN网络的基本要点，下面介绍多节点怎么组成VXLAN网络进行通信。

2.多播模式的**VXLAN**

要组成同一个VXLAN网络，VTEP必须能感知到彼此的存在。多播组本来的功能就是把网络中的某些节点组成一个虚拟的组，所以VXLAN最初想到用多播来实现是很自然的事情。

注：如果VXLAN要使用多播模式，那么底层的网络结构需要支持多播的功能。

这个实验和前面一个非常相似，只不过主机之间不是点对点的连接，而是通过多播组成一个虚拟的整体。多播模式的VXLAN网络拓扑如图1-24所示。

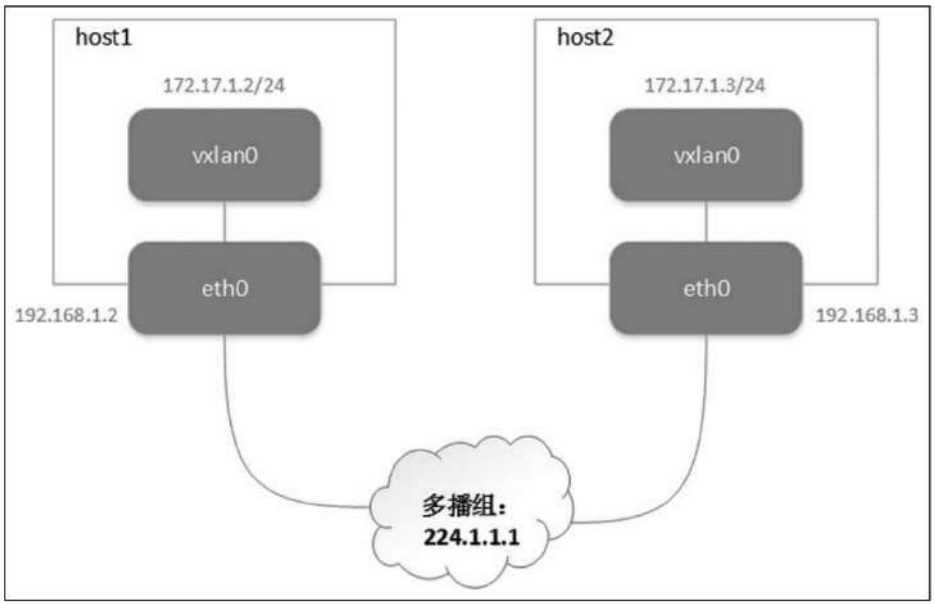


图1-24　多播模式的VXLAN网络拓扑

虽然加入同一个多播组听起来挺复杂，但实际上使用比较简单，和点对点模型相比就多了group参数，命令如下：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    local 192.168.1.2 \
    group 117.1.1.1 \
    dev eth0
```

同样，为刚创建的VXLAN网卡配置IP地址并启用它：

```
# ip addr add 172.17.1.2/24 dev vxlan0
# ip link set vxlan0 up
```

这里最重要的参数是group 224.1.1.1，它表示将VTEP加入一个多播组，多播地址是224.1.1.1。

注：多播地址让源设备能够将分组发送给一组设备。属于多播组的设备将被分配一个多播组IP地址，多播地址范围为224.0.0.0～239.255.255.255。

运行上面的命令之后，一样添加了以下路由信息：

```
# ip route
172.17.1.0/24 dev vxlan0  proto kernel  scope link  src 172.17.1.2
```

不同的是FDB表项的内容：

```
# bridge fdb
00:00:00:00:00:00 dev vxlan0 dst 239.1.1.1 via eth0 self permanent
```

dst字段的值变成了多播地址224.1.1.1，而不是之前对方的VTEP地址，意思是原始报文经过vxlan0后被内核添加上VXLAN头部，其外部UDP头的目的IP地址会被冠上多播地址224.1.1.1。

同理，对所有需要通信的节点进行上述配置，可以验证它们能否通过172.17.1.0/24网络互相通信。

配置完成之后，VTEP通过IGMP加入同一个多播组224.1.1.1。

我们来分析多播模式下VXLAN通信的全过程：

（1）主机1的vxlan0发送ping报文到主机2的172.17.1.3地址，内核发现源地址和目的IP地址在同一个局域网内。需要知道对方的MAC地址，而本地又没有缓存，故先发送一个ARP查询报文。

（2）ARP报文源MAC地址为主机1上vxlan0的MAC地址，目的MAC地址为255.255.255.255（广播地址），并根据配置添加VXLAN头部VNI=42。

（3）不知道对端VTEP在哪台主机上但又配置了多播组，根据配置，VTEP会往多播地址239.1.1.1发送多播报文。

（4）多播组中所有的主机都会收到这个报文，内核发现是VXLAN报文，根据VNI发送给对应的VTEP。

（5）主机2的VTEP去掉VXLAN头部，取出真正的ARP请求报文。同时，VTEP会记录源MAC地址和IP地址信息到FDB表中，这便是一次学习过程。如果发现ARP不是发送给自己的，则直接丢弃；如果是发送给自己的，则生成ARP应答报文。

（6）应答报文目的MAC地址是发送方VTEP的MAC地址，不需要多播。对端VTEP已经通过源报文学习到了VTEP所在主机的MAC地址，故会直接单播发送给目的VTEP。

（7）应答报文通过底层网络直接返回发送方主机，发送方主机根据VNI把报文转发给VTEP，VTEP解包取出ARP应答报文，添加到VTEP缓存中，并根据报文学习到目的VTEP所在的主机地址，添加到FDB表中。

（8）VTEP双方（隧道网络双方）已经通过一次ARP报文知道了建立ICMP通信需要的所有信息，因此后续的ICMP报文都是在这条逻辑隧道中单播进行的。

总结以上过程：一个VXLAN网络的ping报文要经历ARP寻址+ICMP响应两个过程。当然，VTEP设备学习到对方ARP地址后就可以免去ARP寻址的过程。

3.VXLAN+桥接网络

尽管上面的方法能够通过多播实现自动化的overlay网络构建，但是通信的双方只有一个VTEP。在实际生产中，每台主机上都有几十台甚至上百台虚拟机或者容器需要通信，因此需要找到一种方法将这些通信实体组织起来，再经过隧道口VTEP转发出去。

通过前面的介绍我们知道，Linux网桥可以连接多块虚拟网卡，因此可以使用网桥把多个虚拟机或者容器放到同一个VXLAN网络中，VXLAN+网桥的网络拓扑如图1-25所示。

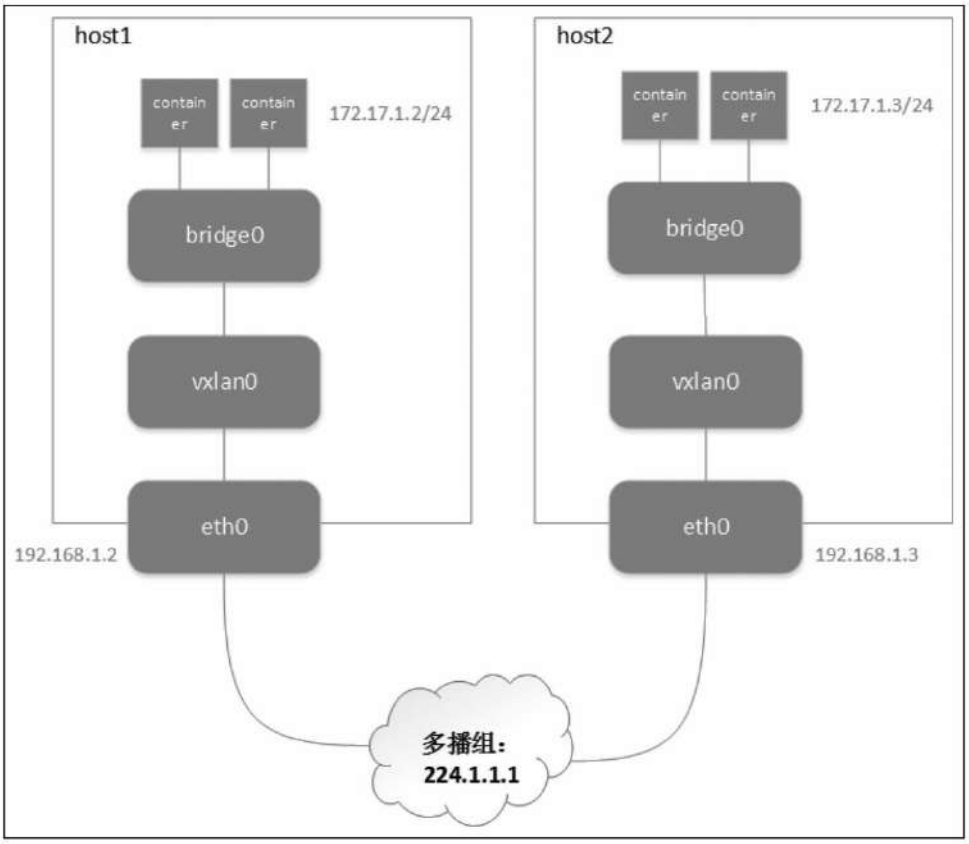


图1-25　VXLAN+网桥的网络拓扑

和上面的多播模式相比，这里只是多了一块网桥，连接同一个主机上的不同容器的veth pair。我们还没有介绍如何创建容器，因此用前面介绍的network namespace代替容器，其实原理都是一样的。我们将创建一个network namespace，并通过一对veth pair将namespace中的eth0网卡连接到网桥。同时，VXLAN网卡也连接到网桥。

首先，创建VXLAN网卡，使用的是多播模式：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    local 192.168.1.2 \
    group 223.1.1.1 \
    dev eth0
```

然后创建网桥bridge0，把VXLAN网卡vxlan0绑定到上面，并启动它们：

```
# ip link add bridge0 type bridge
# ip link set vxlan0 master bridge
# ip link set vxlan0 up
# ip link set bridge0 up
```

下面创建network namespace和一对veth pair，并把veth pair的其中一端绑定到网桥，然后把另一端放到network namespace并绑定IP地址172.17.1.2：

```
# ip netns add container1

# ip link add veth0 type veth peer name veth1
# ip link set dev veth0 master br0
# ip link set dev veth0 up

#ip link set dev veth0 netns container1
#ip netns exec container1 ip link set lo up

#ip netns exec container1 ip link set veth0 name eth0
#ip netns exec container1 ip addr add 172.17.1.2/24 dev eth0
#ip netns exec container1 ip link set eth0 up
```

用同样的方法在另一台主机上配置VXLAN网络，绑定172.17.1.3到另外一个network namespace中的eth0。

从172.17.1.2 ping 172.17.1.3发现整个通信过程和前面的实验类似，只不过容器发出的ARP报文会先经过网桥，再转发给vxlan0；然后在vxlan0处由Linux内核添加VXLAN头部；最后通过多播的方式查询通信对端的MAC地址。

逻辑上，VXLAN网络下不同主机上的network namespace中的网卡被连接到了同一个网桥上。

1.7.6 分布式控制中心

因为并不是所有的网络设备都支持多播，再加上多播方式带来的报文浪费，在实际生产中VXLAN的多播模式很少被采用。

我们来回顾为什么要引入多播。从多播的流程可以看出，其实隧道网络发送报文最关键的就是要知道对方虚拟机/容器的MAC地址及所在主机的VTEP IP地址。对overlay网络来说，它的网段范围是分布在多个主机上的，

因此传统ARP报文的广播无法直接使用。要想做到overlay网络的广播，必须把报文发送到所有VTEP在的节点，这才使用了多播。如果能够事先知道MAC地址和VTEP IP信息，直接告诉发送方VTEP，就不需要多播了。

在虚拟机和容器的场景中，当虚拟机或者容器还没有进行网络通信时，我们就可以知道它的IP和MAC（可能是用某种方式获取，也可能是事先控制这两个地址），分布式控制中心保存了这些信息。除此之外，控制中心还保存了每个VXLAN网络的VTEP，以及这些VTEP的地址。有了这些信息，VTEP就能在发送报文时直接查询并添加头部，不需要多播去满网络地“问”。

在分布式控制中心，一般情况下，这种架构在每个VTEP所在的节点都运行一个agent，它会和控制中心通信，获取隧道通信需要的信息并以某种方式告诉VTEP。

下面我们用一个手动配置VTEP组和FDB表的例子结束本节的讨论。

1.7.7 自维护VTEP组

如果有一种方法能够不通过多播，把overlay的广播报文发送给所有的VTEP主机，则也能完成相同的功能。当然，在维护VTEP网络组之前，必须提前知道哪些VTEP要组成一个网络，以及这些VTEP在哪些主机上。在我们创建VXLAN网卡时，不使用remote或group参数就能达成以上目的，例如：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
```

```
    dev eth0
```

这个VTEP创建时没有指定多播地址，当第一个ARP请求报文发送时，它也不知道要发送给谁，但是我们可以手动添加默认的FDB表项，例如：

```
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.101
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.102
```

如果不知道对方VTEP的地址，内核就会选择默认的FDB表项将网络包发送到192.168.8.101和192.168.8.102，相当于手动维护了一个VTEP的多播组。

在所有节点的VTEP上更新对应的FDB表项，就能实现overlay网络的连通。整个通信流程和多播模式相同，唯一的区别是，VTEP第一次通信时会给所有组内成员发送单播报文，当然也只有一个VTEP会做出应答。

使用一些自动化工具，定时更新FDB表项，就能动态地维护VTEP的拓扑结构。

这个方案解决了在某些underlay网络中不能使用多播的问题，但并没有解决多播的另外一个问题：每次查找MAC地址要发送大量的无用报文，如果VTEP组节点数量很大，那么每次查询都发送N个报文，其中只有一个报文真正有用。

手动维护**FDB**表项

如果提前知道目的容器的MAC地址和它所在主机的IP地址，则可以通过更新FDB表项来减少广播的报文数量。

这次我们在创建VXLAN设备时添加了nolearning参数，这个参数告诉VTEP不要通过收到的报文学习FDB表项的内容，因为我们自己会进行维护：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    dev enp0s8 \
    nolearning
```

然后，添加FDB表项，告诉VTEP容器/虚拟机MAC地址与对应主机IP地址的映射关系：

```
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.101
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.102
# bridge fdb append 52:5e:55:58:9a:ab dev vxlan0 dst 192.168.8.101
# bridge fdb append d6:d9:cd:0a:a4:28 dev vxlan0 dst 192.168.8.102
```

如果知道了对方的MAC地址，则VTEP搜索FDB表项就知道应该发送到哪个对应的VTEP了。需要注意的是，这个情况还是需要默认的表项（那些全零的表项），在不知道容器IP和MAC的对应关系时，通过默认方式发送ARP报文查询对方的MAC地址。

和上一个方法相比，这个方法并没有任何效率上的提升，只是把自动学习FDB表项换成了手动维护（当然，实际情况一般是由自动化程序来维

护），第一次发送ARP请求还是会往VTEP组发送大量单播报文。

这个方法给了我们很重要的提示：如果事先知道VXLAN网络的信息，则VTEP需要的信息都是可以自动维护的，不需要学习。

手动维护**ARP**表项

除了FDB表项，ARP表项也是可以维护的。如果能通过某个方式知道容器的IP和MAC地址的对应关系，只要更新到每个节点，就能实现网络的连通。

这里有个问题，我们需要维护的是每个容器里的ARP表项，因为最终通信的双方是容器。到每个容器里（所有的network namespace）更新对应的ARP表，是一件工作量很大的事情，而且容器的创建和删除还是动态的。Linux提供了一个解决方案，使VTEP可以作为ARP代理，回复ARP请求。也就是说，只要VTEP知道对应的IP地址和MAC地址的映射关系，在接收到容器发来的ARP请求时就可以直接做出应答。这样的话，我们只需要更新VTEP上的ARP表项就行了。

下面这条命令和上面的相比多了proxy参数，这个参数告诉VTEP承担ARP代理的功能，即如果收到ARP请求并且知道结果就直接做出应答。

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    dev enp0s8 \
    nolearning \
    proxy
```

当然，我们还是要手动更新FDB表项来构建VTEP组：

```
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.101
# bridge fdb append 00:00:00:00:00:00 dev vxlan0 dst 192.168.8.102
# bridge fdb append 52:5e:55:58:9a:ab dev vxlan0 dst 192.168.8.101
# bridge fdb append d6:d9:cd:0a:a4:28 dev vxlan0 dst 192.168.8.102
```

还需要为VTEP添加ARP表项。所有要通信容器的IP地址和MAC地址的映射关系都要加进去：

```
# ip neigh add 10.20.1.3 lladdr d6:d9:cd:0a:a4:28 dev vxlan0
# ip neigh add 10.20.1.4 lladdr 52:5e:55:58:9a:ab dev vxlan0
```

在要通信的所有节点配置完之后，容器就能互相ping通。当容器要访问彼此，并且第一次发送ARP请求时，这个请求并不会发给所有的VTEP，而是由当前VTEP做出应答，大大减少了网络上的报文。

借助自动化的工具做到实时的表项（FDB和ARP）更新，这种方法能高效实现overlay网络的通信。

动态更新**FDB**和**ARP**表项

尽管前一种方法通过动态更新FDB和ARP表项避免了多余的网络报文，但还有一个问题：为了让所有的容器能够正常通信，必须提前添加所有容器到ARP和FDB表项中。因为并不是所有的容器都会互相通信，所以添加的有些表项（尤其是ARP表项）是用不到的。

Linux提供了另外一种方法，使内核能够动态地通知节点要和哪个容器通信，应用程序可以订阅这些事件。如果内核发现需要的ARP或者FDB表项不存在，则会发送事件给订阅的应用程序，这样应用程序可以从内核拿到这些信息更新表项，做到更精确的控制。

要收到L2（FDB）miss，必须满足几个条件：

·目的MAC地址未知，即没有对应的FDB表项；

·FDB中没有全零的表项，即默认规则；

·目的MAC地址不是多播或者广播地址。

要实现这种功能，创建VTEP的时候需要加上额外的参数：

```
# ip link add vxlan0 type vxlan \
    id 42 \
    dstport 4789 \
    dev enp0s8 \
    nolearning \
    proxy \
    l2miss \
    l3miss
```

这次多了两个参数l2miss和l3miss：

·l2miss：如果设备找不到MAC地址需要的VTEP地址，就发送通知事件；

·l3miss：如果设备找不到需要IP对应的MAC地址，就发送通知事件。

ip monitor命令能监听某个网卡的事件：

```
# ip monitor all dev vxlan0
```

如果从本节点的容器ping另外一个节点的容器，就先发送L3 miss，下面就是一个L3 miss的通知事件：

```
# ip monitor all dev vxlan0
[nsid current]miss 10.20.1.3  STALE
```

l3miss说明了VTEP不知道它对应的MAC地址，因此要手动添加ARP记录，如下所示：

```
# ip neigh replace 10.20.1.3 \
    lladdr b2:ee:aa:42:8b:0b \
    dev vxlan0 \
    nud reachable
```

上面这条命令设置的nud reachable参数的意思是，这条记录有一个超时时间，系统发现它无效一段时间后会自动删除，无须用户手动删除。ARP记录被删除后，当需要通信时，内核会再次发送L3 miss通知事件。nud是Neighbour Unreachability Detection的缩写，根据需要这个参数也可以设置成其他值，比如permanent，表示这个记录永远不会过时，系统不会检查它是否正确，也不会删除它。

这时还是不能正常通信，内核出现了L2 miss的通知事件：

```
# ip monitor all dev vxlan0
[nsid current]miss lladdr b2:ee:aa:42:8b:0b STALE
```

类似地，这个事件的含义是不知道这个容器的MAC地址在哪个节点上，所以要手动添加FDB记录：

```
# bridge fdb add b2:ee:aa:42:8b:0b dst 192.168.8.101 dev vxlan0
```

在通信的另一台机器上执行响应的操作，就会发现两者能ping通。

1.7.8 小结

VXLAN协议给虚拟网络带来了灵活性和扩展性，让云计算网络能够像计算、存储资源那样按需扩展，并灵活分布。与此同时，VXLAN也带来了额外的复杂度和性能开销。性能开销主要包含两个方面：

·每个VXLAN报文都有额外的50字节开销，加上VXLAN字段，开销要到54字节。这对于小报文的传输是非常昂贵的操作。试想，如果某个报文应用数据才几个字节，那么原来的网络头部加上VXLAN报文头部就有100字节的控制信息；

·每个VXLAN报文的封包和解包操作都是必需的，如果用软件来实现这些步骤，则额外的计算量是不可忽略的影响因素。

多播实现很简单，不需要中心化的控制。不是所有的网络都支持多播，需要事先规划多播组和VNI的对应关系，在overlay网络数量比较多时也会很麻烦，多播会导致大量的无用报文出现在网络中。很多云计算的网络通过自动化的方式发现VTEP和MAC地址等信息，避免多播。

参考资料：https://cizixs.com/2017/09/28/linux-vxlan/.

1.8 物理网卡的分身术：Macvlan

最早版本的Docker是不支持容器网络与宿主机网络直接互通的。从Docker 1.12版本开始，为了解决容器的跨机通信问题引入了overlay和Macvlan网络，其中Macvlan支持容器之间使用宿主机所在网段资源。在介绍Kubernetes网络插件的章节会详细讨论overlay网络，本节将重点介绍Macvlan。除了在容器中有着广泛应用，Macvlan/Macvtap设备也大量应用在虚拟机场景。

1.8.1 Macvlan五大工作模式解析

通常，我们在自定义Docker与外部网络通信的网络时会用到NAT，还有Linux bridge、Open vSwitch、Macvlan几种选择，相比之下，Macvlan拥有更好的性能。

在Macvlan出现之前，我们可以通过网卡别名（例如eth0:1）的方式为一块以太网卡添加多个IP地址，却不能为其添加多个MAC地址。原因是以太网卡是以MAC地址为唯一识别的，而网卡别名并没有改变这些网卡的MAC地址。

Macvlan接口可以看作是物理以太网接口的虚拟子接口。Macvlan允许用户在主机的一个网络接口上配置多个虚拟的网络接口，每个Macvlan接口都有自己的区别于父接口的MAC地址，并且可以像普通网络接口一样分配IP地址。因此，使用Macvlan技术带来的效果是一块物理网卡上可以绑定多个IP地址，每个IP地址都有自己的MAC地址。

用Macvlan虚拟出来的虚拟网卡，在逻辑上和物理网卡是对等的，应用程序可以像使用物理网卡的IP地址那样使用Macvlan设备的IP地址。Macvlan所连接的物理接口通常称为“下部设备”或“上部设备”，方便起见，且与下文要介绍的IPvlan保持一致，称物理接口为父接口，Macvlan设备为子接口。有时，我们也形象地称Macvlan“寄生”在宿主机的物理网卡上。

Macvlan的主要用途是网络虚拟化（包括容器和虚拟机）。另外，有一些比较特殊的场景，例如，keepalived使用虚拟MAC地址。需要注意的是，使用Macvlan的虚拟机或者容器网络与主机在同一个网段，即同一个广播域中。

Macvlan支持5种模式，分别是bridge、VEPA、Private、Passthru和Source模式。

1.bridge模式

该模式类似Linux bridge，是Macvlan最常用的模式，比较适合共享同一个父接口的Macvlan网卡进行直接通信的场景。在bridge模式下，拥有相同父接口的两块Macvlan虚拟网卡可以直接通信，不需要把流量通过父接口发送到外部网络，广播帧将会被洪泛到连接在“网桥”上的所有其他子接口和物理接口。网桥带双引号是因为实际上并没有网桥实体的产生，而是指在这些网卡之间数据流可以实现直接转发，这有点类似于Linux网桥。但Macvlan的bridge模式和Linux网桥不是一回事，它不需要学习MAC地址，也不需要生成树协议（STP），因此性能要优于Linux网桥。Macvlan bridge模式如图1-26所示。

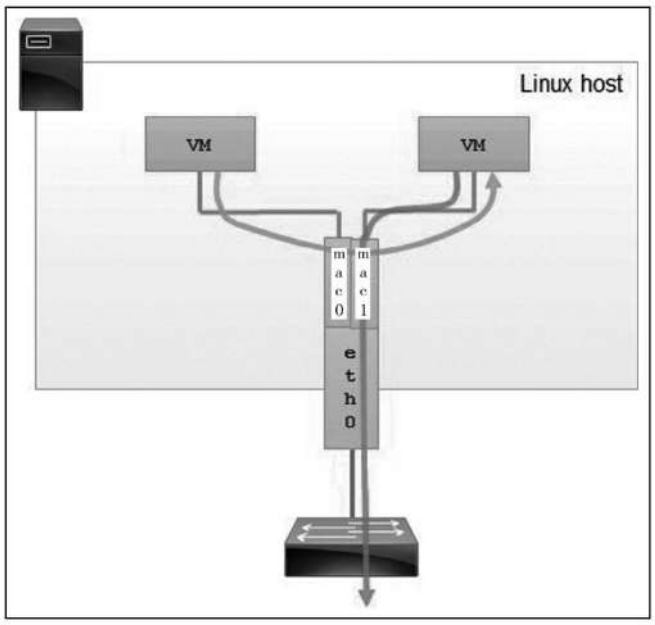


图1-26 Macvlan bridge模式

bridge模式的缺点是如果父接口故障，所有Macvlan子接口会跟着故障，子接口之间也将无法进行通信。

2.VEPA模式

VEPA（Virtual Ethernet Port Aggregator，虚拟以太网端口聚合）是默认模式。所有从Macvlan接口发出的流量，不管目的地址是什么，全部“一股脑”地发送给父接口——即使流量的目的地是共享同一个父接口的其他Macvlan接口。在二层网络下，由于生成树协议的原因，两个Macvlan接口之间的通信会被阻塞，这时就需要接入的外部交换机支持hairpin，把源和目的地址都是本地Macvlan接口地址的流量，发给相应的接口。在VEPA模式下，从父接口收到的广播包会洪泛给所有的子接口。Macvlan VEPA模式如图1-27所示。

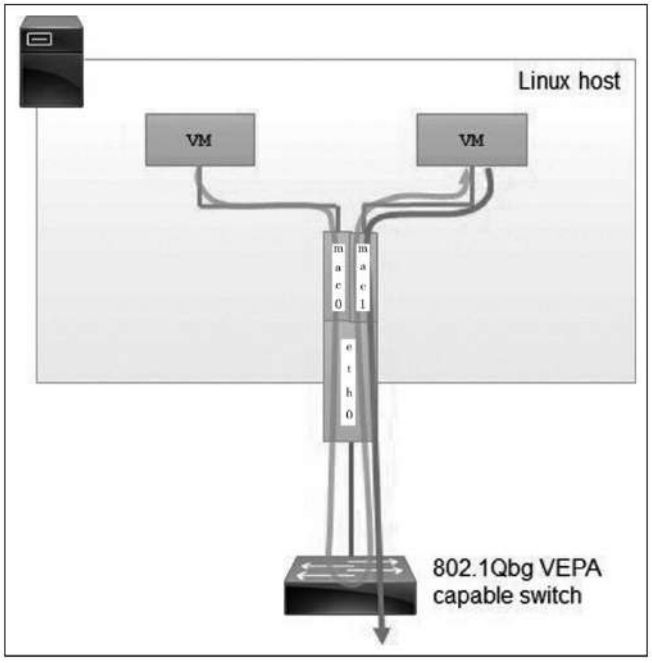


图1-27　Macvlan VEPA模式

目前，大多数交换机都不支持hairpin模式，但Linux可以通过一种hairpin模式的网桥，让VEPA模式下的Macvlan接口能够直接通信（前文已经提到，我们可以把Linux网桥看作二层交换机）。接下来，配置Linux网桥某个端口的hairpin模式：

```
# brctl hairpin br0 eth0 on
```

以上命令的作用是配置Linux网桥br0，使得从eth0收到包后再从eth0发送出去。

以上brctl haripin子命令原型是：

```
# irpin <bridge> <port> {on|off}
```

或者使用iproute2直接设置网卡的hairpin模式：

```
# ip link set dev eth0 hairpin on
```

也可以通过写sysfs目录下的设备文件设置网桥某个端口的hairpin模式。下面的例子设置了网桥br0的eth1端口的hairpin：

```
# echo 1 >/sys/class/net/br0/brif/eth1/hairpin_mode
```

配置了hairpin后，源地址和目的地址都是本地Macvlan接口地址的流

量，会被Linux网桥发回给相应的接口。

如果想在物理交换机层面对虚拟机或容器之间的访问流量进行优化设定，VEPA模式是一种比较好的选择。

3.Private模式

Private模式类似于VEPA模式，但又增强了VEPA模式的隔离能力，其完全阻止共享同一父接口的Macvlan虚拟网卡之间的通信。即使配置了hairpin，让从父接口发出的流量返回宿主机，相应的通信流量依然被丢弃。Macvlan Priviate模式如图1-28所示。

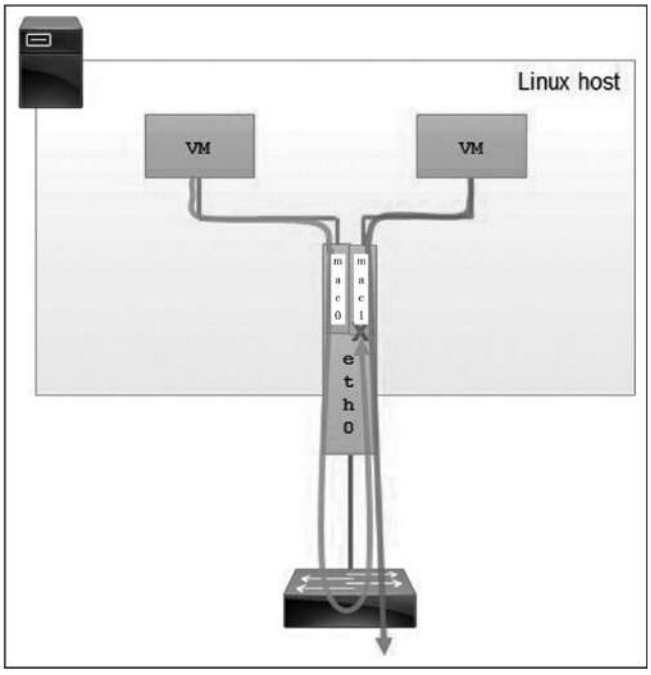


图1-28　Macvlan Private模式

Private的具体实现方式是丢弃广播/多播数据，这就意味着以太网地址解析ARP将无法工作。除非手工探测MAC地址，否则通信将无法在同一宿主机下的多个Macvlan网卡间进行。

如果需要Macvlan的隔离功能，那么Private模式会非常有用。

4.Passthru模式

Passthru模式翻译过来就是直通模式。在这种模式下，每个父接口只能和一个Macvlan网卡捆绑，并且Macvlan网卡继承父接口的MAC地址。Macvlan Passthru模式如图1-29所示。

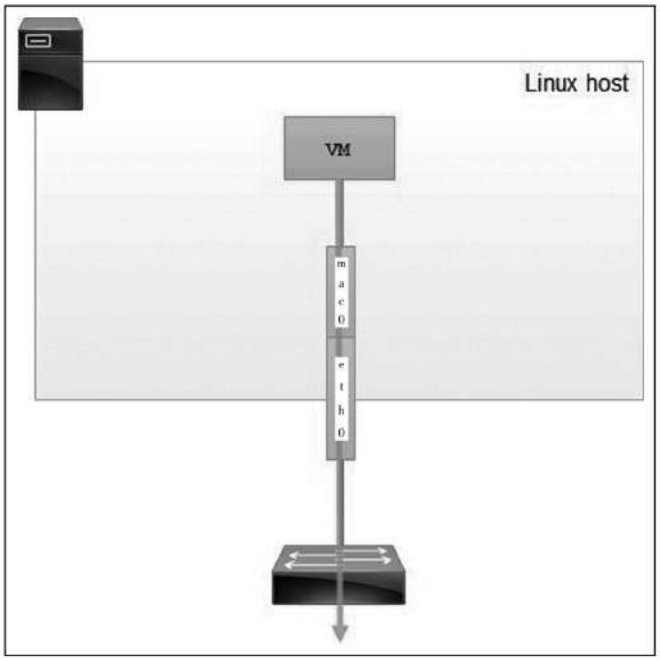


图1-29　Macvlan Passthru模式

在VEPA和Passthru模式下，两个Macvlan网卡之间的通信会经过父接口两次——第一次是发出的时候，第二次是返回的时候。除了限制了Macvlan接口之间的通信性能，还影响物理接口的宽带。

5.Source模式

在这种模式下，寄生在物理设备上，Macvlan设备只接收指定的源Mac地址的数据包，其他数据包一概丢弃。

1.8.2　测试使用Macvlan设备

在宿主机上创建Macvlan设备：

```
# ip link add eth0.1 link eth0 type macvlan mode bridge
```

查看该Macvlan网卡的详细信息：

```
# ip -d link show eth0.1
```

```
27: macvlan2@eth2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode
DEFAULT
    link/ether 26:8a:3c:07:7d:f4 brd ff:ff:ff:ff:ff:ff
    macvlan mode bridge
```

这时，Macvlan网卡eth0.1的设备状态为DOWN，启用它：

```
# ip link set eth0.1 up
```

一般情况下，Macvlan设备的MAC地址是Linux系统自动分配的，用户也可以自定义。使用以下命令就可在新建Macvlan网卡的同时指定其MAC地址：

```
# ip link add eth0.1 link eth0 address 56:61:4f:7c:77:db type macvlan mode vepa
```

如果要删除Macvlan网卡，则只需简单调用以下命令：

```
# ip link del eth0.1
```

1.8.3 Macvlan的跨机通信

我们的实验环境包含两个节点，分别是A节点，IP地址为192.168.1.2；B节点，IP地址为192.168.1.3。直接使用Docker容器进行下面的实验。如果读者对Docker还不熟悉，则可以阅读第2章的内容提前获取必要的背景知识。

先在A节点上创建一个不带网络初始化的Docker容器：

```
# docker run -d --net="none" --name=test1 busybox
```

同时，获取新创建容器对应的PID：

```
# docker inspect --format="" test1
20845
```

然后创建Macvlan设备：

```
# ip link add eth0.1 link eth0 type macvlan mode bridge
```

再将Macvlan设备eth0.1放入容器的网络namespace中：

```
# ip link set netns 2084 eth0.1
```

进入网络namespace中配置Macvlan网卡：

```
# nsenter --target=2084 --net
# ip link set eth0.1 up
# ifconfig
eth0.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::6ce2:9cff:fee3:15c6  prefixlen 64  scopeid 0x20<link>
        ether 6e:e2:9c:e3:15:c6  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 468 (468.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

# ip addr add 192.168.1.12/24 dev eth0.1
# ip route add default via 192.168.1.254 dev eth0.1
```

如上所示，Macvlan设备启用后会自动分配MAC地址而没有IP（v4）地址。我们为其分配IP地址并设置网关。

现在，我们可以从主机B上访问容器test1，如下所示：

```
# ping 192.168.1.12
PING 192.168.1.12 (192.168.1.12) from 192.168.1.3 eth0: 56(84) bytes of data.
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.121 ms
^C
```

从主机A上ping容器test1，发现不通：

```
# ping 192.168.1.12
PING 192.168.1.12 (192.168.1.12) 56(84) bytes of data.
From 192.168.1.2 icmp_seq=2 Destination Host Unreachable
```

这是什么原因呢？在Macvlan虚拟网络世界中，物理网卡（父接口）相当于一个交换机，对于进出其子Macvlan网卡的数据包，物理网卡只转发数据包而不处理数据包，于是也就造成了使用本机Macvlan网卡的IP无法和物理网卡的IP通信。

总结，Macvlan只为虚拟机或容器提供访问外部物理网络的连接。

1.8.4 Macvlan与overlay对比

Macvlan和overlay都是Docker原生提供的网络方案，但是Macvlan和overlay网络的工作方式有所不同。

首先，Macvlan和overlay的网络作用范围不一样。overlay是全局作用范围类型的网络，而Macvlan的作用范围只是本地。举例来说，全局类型的网络可以作用于一个Docker集群，而本地类型的网络只作用于一个Docker节点。

其次，每个宿主机创建的Macvlan网络都是独立的，一台机器上创建的Macvlan网络并不影响另一台机器上的网络。但这并不意味着两台Macvlan的主机不能通信，当同时满足以下两个条件时，可以实现Macvlan网络的跨主机通信：

（1）通信两端的主机在网卡配置混杂模式。

（2）两台主机上的Macvlan子网IP段没有重叠。

1.8.5 小结

Macvlan是将虚拟机或容器通过二层连接到物理网络的一个不错的方案，但它也有一些局限性，例如：

·因为每个虚拟网卡都要有自己的MAC地址，所以Macvlan需要大量的MAC地址，而Linux主机连接的交换机可能会限制一个物理端口的MAC地址数量上限，而且许多物理网卡的MAC地址数量也有限制，超过这个限制就会影响到系统的性能；

·IEEE 802.11标准（即无线网络）不喜欢同一个客户端上有多个MAC地址，这意味着你的Macvlan子接口没法在无线网卡上通信。

我们可以通过复杂的办法突破以上这些限制，但还有一种更简单的办法。那就是使用IPvlan。

1.9 Macvlan的救护员：IPvlan

前文我们详细讨论了Macvlan，而且总结了Macvlan的一些局限性，也提到了IPvlan可以解决Macvlan的一些限制，因此本节将介绍IPvlan。

Macvlan和IPvlan虚拟网络模型提供的功能看起来差不多，那么，什么时候需要用到IPvlan呢？要回答这个问题，先来看看Macvlan先天存在的不足：

·无法支持大量的MAC地址；

·无法工作在无线网络环境中。

1.9.1 IPvlan简介

与Macvlan类似，IPvlan也是从一个主机接口虚拟出多个虚拟网络接口。区别在于IPvlan所有的虚拟接口都有相同的MAC地址，而IP地址却各不相同。因为所有的IPvlan虚拟接口共享MAC地址，所以特别需要注意DHCP使用的场景。DHCP分配IP地址的时候一般会用MAC地址作为机器的标识。因此，在使用Macvlan的情况下，客户端动态获取IP的时候需要配置唯一的Client ID，并且DHCP服务器也要使用该字段作为机器标识，而不是使用MAC地址。

Linux内核3.19版本才开始支持IPvlan，Docker从4.2版本起能够稳定支持IPvlan。

IPvlan有两种不同的模式，分别是L2和L3。一个父接口只能选择其中一种模式，依附于它的所有子虚拟接口都运行在该模式下。

1.L2模式

IPvlan L2模式和Macvlan bridge模式的工作原理很相似，父接口作为交换机转发子接口的数据。同一个网络的子接口可以通过父接口转发数据，如果想发送到其他网络，则报文会通过父接口的路由转发出去。

2.L3模式

L3模式下，IPvlan有点像路由器的功能。如图1-30所示，IPvlan在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相ping通对方，因为IPvlan会在中间做报文的转发工作。

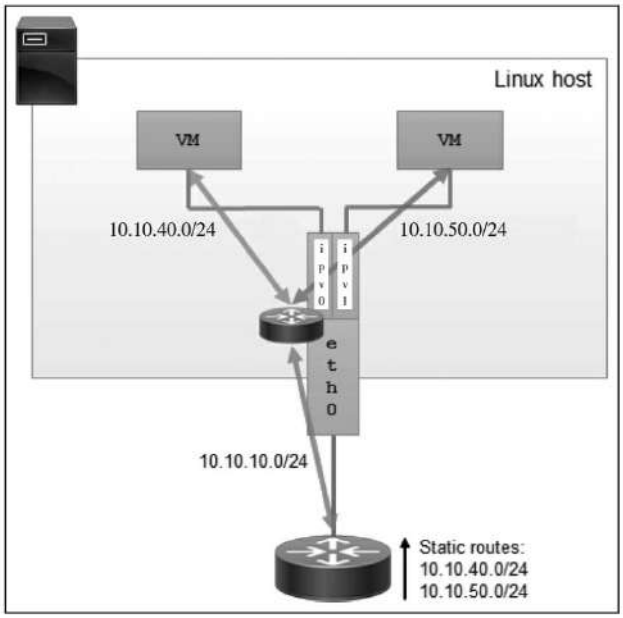


图1-30　IPvlan L3模式

L3模式下的虚拟接口不会接收多播或者广播的报文，原因是所有的网络都会发送给父接口，所有的ARP过程或者其他多播报文都是在底层的父接口完成的。需要注意的是，外部网络默认情况下是不知道IPvlan虚拟出来的网络的，如果不在外部路由器上配置好对应的路由规则，那么IPvlan的网络是不能被外部直接访问的。

1.9.2　测试IPvlan

先创建两个测试用的network namespace：

```
# ip netns add net1
# ip netns add net2
```

然后创建IPvlan的虚拟网卡接口，因为L2和Macvlan功能相同，所以这里测试L3模式。创建IPvlan虚拟接口的命令和Macvlan格式相同：

```
# ip link add ipv1 link eth0 type ipvlan mode l3
# ip link add ipv2 link eth0 type ipvlan mode l3
```

把IPvlan接口放到前面创建好的network namespace中：

```
# ip link set ipv1 netns net1
# ip link set ipv2 netns net2
# ip netns exec net1 ip link set ipv1 up
# ip netns exec net2 ip link set ipv2 up
```

给两个虚拟网卡接口配置不同网络IP地址，并配置好路由项：

```
# ip netns exec net1 ip addr add 10.0.1.10/24 dev ipv1
# ip netns exec net2 ip addr add 192.168.1.10/24 dev ipv2
# ip netns exec net1 ip route add default dev ipv1
# ip netns exec net2 ip route add default dev ipv2
```

最后测试两个网络的连通性：

```
# ip netns exec net1 ping -c 3 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=0.053 ms
64 bytes from 192.168.1.10: icmp_seq=2 ttl=64 time=0.035 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=64 time=0.036 ms

--- 192.168.1.10 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.035/0.041/0.053/0.009 ms
```

1.9.3 Docker IPvlan网络

从Docker 1.13版本开始加入了对IPvlan的支持，测试过程如下所示。

读者可以先阅读第2章，获得关于Docker的背景知识。

首先，创建IPvlan的网络，这需要在创建Docker网络时把网络驱动（-d）设置成ipvlan，同时设置IPvlan的工作模式为L3：

```
# docker network create -d ipvlan \
    --subnet=192.168.30.0/24 \
    -o parent=eth0 \
    -o ipvlan_mode=l3 ipvlan30
```

然后，启动两个容器，发现在同一个IPvlan网络的两个容器可以互相

ping通：

```
# docker run --net=ipvlan30 -it --name ivlan_test3 --rm alpine /bin/sh
# docker run --net=ipvlan30 -it --name ivlan_test4 --rm alpine /bin/sh
```

再创建另外一个IPvlan网络，和前面的网络不在同一个广播域：

```
# docker network  create  -d ipvlan \
    --subnet=192.168.110.0/24 \
    -o parent=eth0 \
    -o ipvlan_mode=l3 ipvlan110
```

最后，在新建的网络中运行容器，发现可以ping通前面网络中的容器：

```
# docker run --net=ipvlan30 -it --name ivlan_test3 --rm alpine /bin/sh
```

1.9.4 小结

我们将IPvlan称为Macvlan的“救护员”是因为IPvlan除了能够完美解决以上问题，还允许用户基于IPvlan搭建比较复杂的网络拓扑，不再基于Macvlan的简单的二层网络，而是能够与BGP（Boader Gateway Protocol，边界网关协议）等协议扩展我们的网络边界。

第2章　饮水思源：Docker网络模型简介

2.1　主角登场：Linux容器

软件开发最麻烦的事情之一，就是环境配置。举例来说，安装一个Python应用，除了需要该应用的二进制文件，系统中还必须要有Python的解析器、该应用程序的外部依赖及配置环境变量等，而且还要考虑操作系统的因素（例如跨平台兼容性等）。我们将软件运行时依赖的环境统称为运行时（runtime）。既然环境配置如此麻烦，一个直观的感受就是软件能否带环境安装？

虚拟机的快照（snapshot）就是带环境安装的一种解决方案。虽然用户可以通过虚拟机还原软件的原始环境，但每个虚拟机都包含完整的操作系统，因此有资源占用多、启动慢等缺点。

鉴于虚拟机的这些缺点，Linux发展出了另一种虚拟化技术：容器。

2.1.1　容器是什么

容器不是模拟一个完整的操作系统，而是对进程进行隔离，隔离用到的技术就是1.1节介绍的Linux namespace。对容器里的进程来说，它接触到的各种资源看似是独享的，但在底层其实是隔离的。容器是进程级别的隔离技术，因此相比虚拟机有启动快、占用资源少、体积小等优点。

目前最流行的Linux容器非Docker莫属，它是对Linux底层容器技术的一种封装，提供简单易用的容器使用接口。Docker将应用程序与该程序的依赖打包在同一个文件里，即所谓的Docker image。运行Docker image就会生成一个Docker容器。程序在这个虚拟容器里运行就像是在物理机上或虚拟机上运行一样。

2.1.2　容器与虚拟机对比

传统的虚拟机需要模拟整台机器，包括硬件（因此，虚拟机方案需要硬件的支持，例如VT-X），每台虚拟机都需要有自己的操作系统。虚拟机一

旦被开启，预分配给它的资源将全部被占用。每台虚拟机包括应用程序、必要的依赖库，以及一个完整的用户操作系统。

容器和宿主机共享操作系统，而且可以实现资源的动态分配。容器包含应用程序和所依赖的软件包，并且不同容器之间共享内核，这与虚拟机相比大大节省了额外的资源占用。在宿主机操作系统中，不同容器在用户空间以隔离的方式运行着各自的进程。容器与虚拟机对比如图2-1所示。

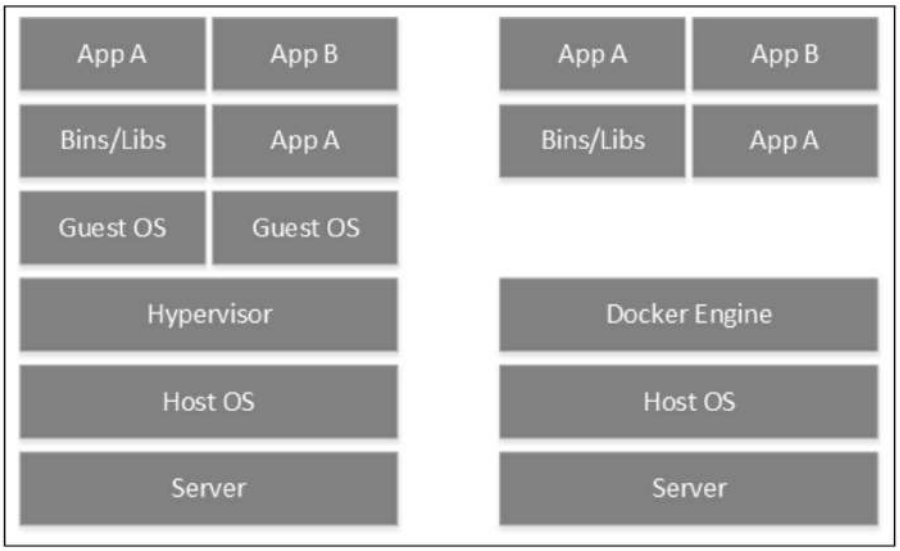


图2-1　容器与虚拟机对比

从图2-1可以看出，虚拟机和容器最大的区别在于没有Guest　OS（客户虚拟机）这一层。虚拟机和容器都是在硬件和操作系统以上的，虚拟机有Hypervisor层，Hypervisor即虚拟化管理软件，是整个虚拟机的核心所在。它为虚拟机提供了虚拟的运行平台，管理虚拟机的操作系统运行。每台虚拟机都有自己的操作系统及系统库。

容器没有Hypervisor这一层，也没有Hypervisor带来性能的损耗，每个容器和宿主机共享硬件资源及操作系统。细心的读者可能已经看出来了，Docker容器有Docker　Engine这一层。其实Docker　Engine远远比Hypervisor轻量，它只负责对Linux内核namespace　API的封装和调用，真正的内核虚拟化技术是由Linux提供的。容器的轻量带来的一个好处是资源占用远小于虚拟机。同样的硬件环境，可以运行容器的数量远大于虚拟机，这对提供系统资源利用率非常有用。

每台虚拟机都有一个完整的Guest OS，Guest OS能为应用提供一个更加隔离和安全的环境，不会因为应用程序的漏洞给宿主机造成任何威胁。

2.1.3　小结

从虚拟化层面来看，传统虚拟化技术是对硬件资源的虚拟，容器技术则

是对进程的虚拟，从而提供更轻量级的虚拟化，实现进程和资源的隔离。从架构来看，容器比虚拟机少了Hypervisor层和Guest OS层，使用Docker Engine进行资源分配调度并调用Linux内核namespace API进行隔离，所有应用共用主机操作系统。因此在体量上，Docker较虚拟机更轻量级，在性能上优于虚拟化，接近裸机性能。

2.2 打开万花筒：Docker的四大网络模式

从网络的角度看容器，就是network namespace+容器的组网方案。利用network namespace，可以为Docker容器创建隔离的网络环境。容器具有完全独立的网络栈，与宿主机隔离。用户也可以让Docker容器共享主机或者其他容器的network namespace。network namespace的相关技术点已经在第1章详细地介绍过，本节不再赘述。

容器的网络方案可以分为三大部分：

·单机的容器间通信；

·跨主机的容器间通信；

·容器与主机间通信。

本节将介绍单机下Docker的4种网络通信模式。

我们在使用docker run命令创建Docker容器时，可以使用--network选项指定容器的网络模式。Docker有以下4种网络模式：

·bridge模式，通过--network=bridge指定；

·host模式，通过--network=host指定；

·container模式，通过--network=container:NAME_or_ID指定，即joiner容器；

·none模式，通过--network=none指定。

在安装完Docker之后，Docker Daemon会在宿主机上自动创建三个网络，分别是bridge网络、host网络和none网络，可以使用docker network ls命令查看，如下所示：

```
# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b9dfa4e41c0b        bridge              bridge              local
a07e27bc3ca3        host                host                local
b94a368407c5        none                null                local
```

2.2.1 bridge模式

Docker在安装时会创建一个名为docker0的Linux网桥。bridge模式是Docker默认的网络模式，在不指定--network的情况下，Docker会为每一个容器分配network namespace、设置IP等，并将Docker容器连接到docker0网桥上。严谨的表述是，创建的容器的veth pair中的一端桥接到docker0上。docker0网桥是普通的Linux网桥，而非OVS网桥，因此我们通过brctl命令查看该网桥信息：

```
# brctl show
bridge name  bridge id      STP enabled  interfaces
docker0   8000.02423c5eccd8 no
```

注意，现在的docker0网桥上还没有任何对接的网卡。我们创建一个容器后再来看docker0有什么变化。

```
# docker run -d nginx

# brctl show
bridge name bridge id  STP enabled interfaces
docker0    8000.02423c5eccd8 no veth160074d
```

如上所示，当我们成功创建了一个容器后，docker0上就挂接了一块新的网卡，这块网卡就是新建容器时创建的，并放在主机network namespace的veth pair一端。

在默认情况下，docker0的IP地址均为172.17.0.1（除非在Docker Daemon启动时自行配置），而接到docker0上的Docker容器的IP地址范围是172.17.0.0/24。连接在docker0上的所有容器的默认网关均为docker0，即访问非本机容器网段要经过docker0网关转发，而同主机上的容器（同网段）之间通过广播通信。下面的容器内的路由表信息证实了这一点：

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
```

不仅如此，发到主机上要访问Docker容器的报文默认也要经过docker0进行广播，请看下面主机上的路由表信息：

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
```

bridge模式为Docker容器创建独立的网络栈，保证容器内的进程使用独立的网络环境，使容器和容器、容器和宿主机之间能实现网络隔离。

2.2.2 host模式

连接到host网络的容器共享Docker host的网络栈，容器的网络配置与host完全一样。host模式下容器将不会获得独立的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，而是使用宿主机的IP和端口。

我们先查看主机的网络：

```
# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 100.106.89.164  netmask 255.255.254.0  broadcast 100.106.89.255
        inet6 fe80::2a6e:d4ff:fe89:1e92  prefixlen 64  scopeid 0x20<link>
        ether 28:6e:d4:89:1e:92  txqueuelen 1000  (Ethernet)
        RX packets 39335399  bytes 18896959864 (17.5 GiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1932208  bytes 1343486708 (1.2 GiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 7641540  bytes 483642612 (461.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7641540  bytes 483642612 (461.2 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

然后创建host网络的容器，再查看容器的网络信息：

```
# docker run busybox --network=host ifconfig
```

host模式下的容器可以看到宿主机的所有网卡信息，甚至可以直接使用宿主机IP地址和主机名与外界通信，无须额外进行NAT，也无须通过Linux bridge进行转发或者进行数据包的封装。Docker的host模式网络原理如图2-2所示，其中虚线框表示是容器和主机共享network namespace。

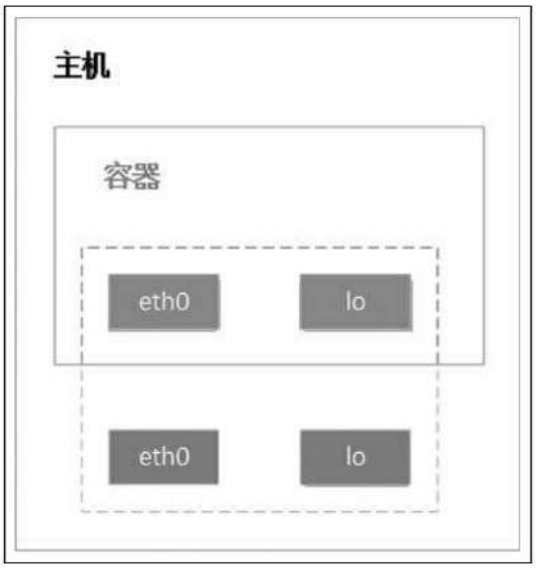


图2-2　Docker的host模式网络原理

当然，host模式有利有弊，优点是没有性能损耗且配置方便，缺点也很明显，例如：

·容器没有隔离、独立的网络栈：容器因与宿主机共用网络栈而争抢网络资源，并且容器崩溃也可能使主机崩溃，导致网络的隔离性不好；

·端口资源冲突：宿主机上已经使用的端口就不能再用了。

2.2.3　container模式

创建容器时使用--network=container:NAME_or_ID模式，在创建新的容器时指定容器的网络和一个已经存在的容器共享一个network namespace，但是并不为Docker容器进行任何网络配置，这个Docker容器没有网卡、IP、路由等信息，需要手动为Docker容器添加网卡、配置IP等。

需要注意的是，container模式指定新创建的容器和已经存在的任意一个容器共享一个network namespace，但不能和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的IP，而是和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。两个容器的进程可以通过lo网卡设备通信。

Kubernetes的Pod网络采用的就是Docker的container模式网络，我们将在后面的章节详细介绍。

2.2.4 none模式

none模式下的容器只有lo回环网络，没有其他网卡。none模式网络可以在容器创建时通过--network=none指定。这种类型的网络没有办法联网，属于完全封闭的网络。唯一的用途是客户有充分的自由度做后续的配置。

这种模式下的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容器没有网卡、IP、路由等信息，需要我们自己为Docker容器添加网卡、配置IP等。

2.3 最常用的Docker网络技巧

Docker容器既可以让外部访问，也可以访问外网，还可以容器间相互访问。Docker本身提供了很多对网络的配置命令。本节将简单介绍Docker中的网络配置、网络端口映射、容器互联、DNS配置等实战知识。bridge模式是Docker容器的默认组网模式，因此没有特别说明的情况下下文使用的都是bridge网络模式。

2.3.1 查看容器IP

查看一个正在运行的容器被分配了什么IP地址，这个需求既简单又常见。一般有两种方式：从外面看和从里面看。从外面看即使用docker inspect命令。例如：

```
# docker inspect -f "{{ .NetworkSettings.IPAddress }}" <containerNameOrId>
```

其实也等价于：

```
# docker inspect <containerNameOrId> | grep '"IPAddress"' | head -n 1
```

一般情况下，默认Docker分配的IP地址范围是172.17.0.0/16，那么第一个IP地址172.17.0.1是分配给docker0的，第一个容器将会被分到172.17.0.2，第二个容器是172.17.0.3，依此类推。

从里面看就是指通过docker exec或docker attach进入容器，通过ip或ifconfig命令查看，这里省略演示。

2.3.2 端口映射

在使用docker run的时候可以使用-P或者-p命令进行容器和主机之间的端口映射。使用-P（大写）不需要指定任何映射关系，默认情况下，Docker会随机将一个49000 49900的端口映射到内部容器开放的网络端口。使用-p（小写）则需要指定主机的端口应该映射到容器的哪个端口。使用格式如下所示：

```
# docker run -p 1234:80 -d nginx
```

注：-p的格式是hostport:containerport。

通过docker ps查看：

```
# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
ef84a127db3c        nginx               "nginx -g 'daemon of…"   2 minutes ago
 Up 2 minutes        0.0.0.0:1234->80/tcp   jolly_noethe
```

Docker容器端口映射原理都是在本地的iptable的nat表中添加相应的规则，将访问本机IP地址:hostport的网包进行一次DNAT，转换成容器IP:containerport。可以通过以下命令查看iptables得到验证：

```
# iptables -t nat -nL
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type
 LOCAL

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type
 LOCAL

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:1234 to
:172.17.0.2:80
```

如上所示，DNAT发生在DOCKER这条iptables链，它有两处引用，分别是PREROUTING链和OUTPUT链，意味着从外面发到本机和本地进程访问本机（由iptables匹配规则ADDRTYPE match dst-type LOCAL指定）的1234端口的包目的地址都会被修改成172.17.0.2:80。

除了使用docker ps命令给出容器的端口映射关系，还可以使用docker port命令查看容器的端口在主机上的映射，形如：

```
# docker port <container> <port number>
```

2.3.3 访问外网

怎么从容器内访问外网呢？一般情况下需要两个因素：ip_forward和SNAT/MASQUERADE。在默认情况下，容器可以访问外部网络的连接，因为容器的默认网络接口为docker0网桥上的接口，即主机上的本地接口。其原理是通过Linux系统的转发功能实现的（把主机当交换机）。如果发现容器内访问不了外网，则需要确认系统的ip_forward是否已打开。

```
# sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

或者检查docker daemon启动的时候--ip-forward参数是不是被设置成false了，如果是的话，则需要设置--ip-forward=true重新启动Docker，Docker会打开主机的ip forward。

至于SNAT/MASQUERADE，Docker会自动在iptables的POSTROUTING链上创建形如下面的规则：

```
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       0.0.0.0/0
```

即从容器网段出来访问外网的包，都要做一次MASQUERADE，即出去的包都用主机的IP地址替换源地址。

2.3.4 DNS和主机名

容器中的DNS和主机名一般通过三个系统配置文件维护，分别是/etc/resolv.conf、/etc/hosts和/etc/hostname，其中：

·/etc/resolv/conf在创建容器的时候，默认与本地主机/etc/resolv.conf保持一致；

·/etc/hosts中则记载了容器自身的一些地址和名称；

·/etc/hostname中记录容器的主机名。

用户如果直接在容器内修改这三个文件会立即生效，但容器重启后修改又会失效。如果想统一、持久化配置所有容器的DNS，通过修改主机Docker Daemon的配置文件（一般是/etc/docker/daemon.json）的方式指定除主机/etc/resolv.conf的其他DNS信息：

```
{
    "dns" : [
        "114.114.114.114",
        "8.8.8.8"
    ]
}
```

同时，也可以在docker run时使用--dns=address参数来指定。

至于配置Docker容器的主机名，则可以在运行docker run创建容器时使用参数-h hostname或者--hostname hostname配置。

2.3.5 自定义网络

上文提到Docker默认会创建一个docker0网桥，其实这个网桥和Docker容器使用的网段都是可以自定义的。

当启动Docker Daemon时，可以对docker0网桥进行配置，例如：

·--bip=CIDR，即配置接到这个网桥上的Docker容器的IP地址网段，例如192.168.1.0/24；

·--mtu=BYTES，配置docker0网桥的MTU（最大传输单元）。

如果已经使用默认配置启动Docker Daemon，又希望有个自定义的容器网段却不想重启Docker Daemon，那么可以通过再创建一个自定义的网络达成目的。简而言之，一个主机上Docker支持多个网络。

Docker的命令行工具支持直接操作网络，就像操作容器一样。用户创建一个网络可以使用下面的命令：

```
# docker network create -d bridge --subnet 172.25.0.0/16 mynet
3df48fd0c8dd57a7da52c8568b9b1b6af0f6b306f0494f02054ec2bc7ee3e17a
```

以上命令创建了一个名为mynet的网络，网络ID是3df48fd0c8dd...。该网络使用的是bridge模式，网段是172.25.0.0/16。不难猜测，该网络的第一个IP地址172.25.0.1应该是分配给网桥了。那么，这个bridge网络的网桥叫什么名

字呢？可以通过ip addr命令查看：

```
# ip addr
19: br-3df48fd0c8dd: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
 DOWN group default
    link/ether 02:42:6e:d4:95:d2 brd ff:ff:ff:ff:ff:ff
    inet 172.25.0.1/16 scope global br-3df48fd0c8dd
       valid_lft forever preferred_lft forever
```

如上所示，新创建bridge网络的网桥是br-3df48fd0c8dd（后面这串字符其实是网络ID的前半部分），配置的也是默认的1500 MTU。该网桥的IP地址果然是172.25.0.1。

想查看某个网络的详细信息，可以使用inspect子命令，就像inspect容器一样。命令如下：

```
# docker inspect 3df48fd0c8dd
[
    {
        "Name": "mynet",
        "Id": "3df48fd0c8dd57a7da52c8568b9b1b6af0f6b306f0494f02054ec2bc7ee3e17a",
        "Created": "2019-03-16T20:07:24.930922415+08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.25.0.0/16"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {},
```

```
        "Options": {},
        "Labels": {}
    }
]
```

也可以加上network前缀：

```
# docker network inspect 3df48fd0c8dd
```

查看主机上有多少个docker network，可以使用以下命令：

```
# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b9dfa4e41c0b        bridge              bridge              local
a07e27bc3ca3        host                host                local
3df48fd0c8dd        mynet               bridge              local
b94a368407c5        none                null                local
```

DRIVE字段即使用的Docker容器网络模式。可能有读者会好奇，明明我只创建了一个自定义网络3df48fd0c8dd，如果再加上一个docker0所在的网络，那么上面列出的另外两个网络是由谁、在什么时候创建的呢？原来，默认情况下，Docker Daemon会预创建3个网络，分别使用对应的网络模式作为网络名称，即上面列出的bridge、host和null（对应none模式）。我们在用docker run命令启动容器时使用的--net或--network原来是在指定使用哪个网络！

如果希望删除某个网络，则只需使用network rm子命令，例如：

```
# docker network rm 3df48fd0c8dd
```

连接一个容器到网络中，用法如下：

```
# docker network connect mynet containername
```

将容器和网络断开，用法如下：

```
# docker networdk disconnect
```

2.3.6 发布服务

这里先简单介绍Docker service的概念。首先，Docker service是Docker的一个子命令，如下所示：

```
\section{docker service --help}

Usage:  docker service COMMAND

Manage services

Commands:
  create      Create a new service
  inspect     Display detailed information on one or more services
  ls          List services
  ps          List the tasks of a service
  rm          Remove one or more services
  scale       Scale one or multiple replicated services
  update      Update a service
```

Docker的service子命令是对Docker容器的一层封装，有点类似于Kubernetes的Service概念。一个结合network和service的例子如下：

```
# docker network create -d brige foo
# docker service publish my-service.foo
# docker service attach <container-id> my-service.foo
```

以上三条命令，先创建一个Docker的网络，然后在这个网络里发布一个服务，最后将这个服务和容器绑定（也就是把这个容器加入刚创建的网络中），这样就达到了把容器当作服务发布的目的。

用下面这条命令可以一次性实现以上三个步骤的效果：

```
# docker run -itd --publish-service db.foo.brige busybox
```

2.3.7 docker link：两两互联

首先声明，docker link是一个遗留的特性，在新版本的Docker中，一般

不推荐使用。考虑到link也算是Docker发明的一个比较有意思的功能，对后面理解Kubernetes Service也有一定帮助，本节将对其做简单介绍，重点阐述其实现机制。

简单地说，docker link就是把两个容器连起来，相互通信。link的使用方式如下：

```
# docker run -d nginx --link=<container name or ID>:<alias>
```

上面这条命令alias是源容器在link下的别名。link方式最大的便利在于容器可以使用容器名进行通信，而不需要依赖IP地址。不过，link方式仅解决了单机容器间点对点的互联问题。

下面将用一个例子演示docker link的具体用法。首先，创建一个名为abc的容器：

```
# docker run -d --name abc nginx
```

然后，创建link到容器abc的容器efg，如下所示：

```
# docker run -d --name efg --link abc:source nginx
```

上面的命令创建并启动名为efg的容器，并把该容器和名为abc的容器链接起来。而--link abc:source的意思是将启动的abc作为源容器，source是该容器在link下的别名。换句话说，站在efg容器的角度看，abc和source都指向同一个容器，并且可以作为该容器的主机名。efg容器用这两个名字中的任意一个都可以与那个容器通信。我们进入efg这个容器ping容器abc和source，并查看结果：

```
# docker exec -it efg /bin/sh

# ping abc
PING hub (172.17.0.2) 56(84) bytes of data.
64 bytes from hub (172.17.0.2): icmp_seq=1 ttl=64 time=0.184 ms
64 bytes from hub (172.17.0.2): icmp_seq=2 ttl=64 time=0.133 ms
64 bytes from hub (172.17.0.2): icmp_seq=3 ttl=64 time=0.216 ms

# ping source
PING hub (172.17.0.2) 56(84) bytes of data.
64 bytes from hub (172.17.0.2): icmp_seq=1 ttl=64 time=0.194 ms
64 bytes from hub (172.17.0.2): icmp_seq=2 ttl=64 time=0.218 ms
64 bytes from hub (172.17.0.2): icmp_seq=3 ttl=64 time=0.128 ms
```

可见，abc和source两个主机名都指向IP地址172.17.0.2。打开efg容器的host文件一探究竟：

```
# docker exec -it efg /bin/sh
# cat /etc/hosts
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
172.17.0.2  source 96f07804df65 abc
```

```
172.17.0.3  dfe3767h9990
```

我们可以发现接收容器（efg）的host文件里把源容器的ID（96f07804df65）、容器名（abc）和别名（source）都映射到172.17.0.2 IP地址。如果重启了源容器，则接收容器的/etc/hosts文件会自动更新源容器的新IP。

2.4 容器网络的第一个标准：CNM

围绕Docker生态，目前有两种主流的网络接口方案，即Docker主导的Container Network Model（CNM）和Kubernetes社区主推的Container Network Interface（CNI），其中CNM相对CNI较早提出。下文将重点介绍Docker的CNM模型，CNI的介绍将放在后面的章节。

2.4.1 CNM标准

CNM模型如图2-3所示。

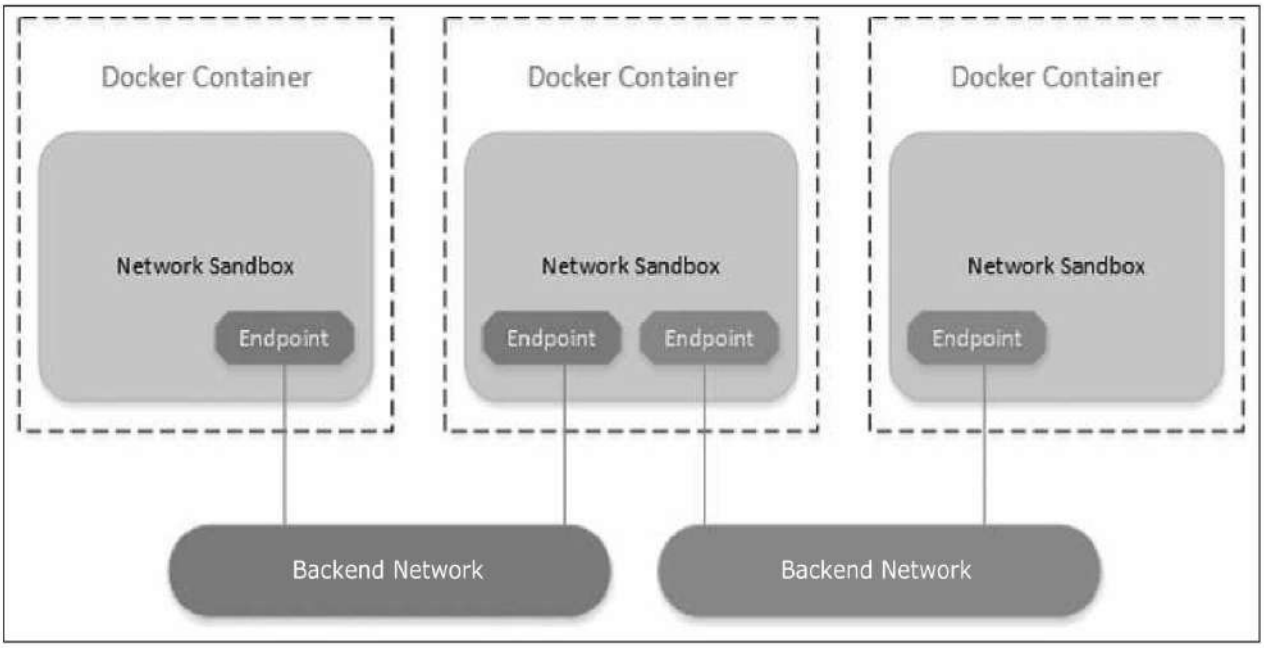


图2-3 CNM模型

从上图不难发现，CNM有3个主要概念：

·**Network Sandbox**：容器网络栈，包括网卡、路由表、DNS配置等。对应的技术实现有network namespace、FreeBSD Jail（类似于namespace的隔离技术）等。一个Sandbox可能包含来自多个网络（Network）的多个Endpoint；

·**Endpoint**：Endpoint作为Sandbox接入Network的介质，是Network Sandbox和Backend Network的中间桥梁。对应的技术实现有veth pair设备、tap/tun设备、OVS内部端口等；

·**Backend Network**：一组可以直接相互通信的Endpoint集合。对应的技

术实现包括Linux bridge、VLAN等。如读者所见，一个Backend Network可以包含多个Endpoint。除此之外，CNM还需要依赖另外两个关键的对象来完成Docker的网络管理功能，它们分别是：

·**Network Controller**：对外提供分配及管理网络的APIs，Docker Libnetwork支持多个网络驱动，Network Controller允许绑定特定的驱动到指定的网络；

·**Driver**：网络驱动对用户而言是不直接交互的，它通过插件式的接入方式，提供最终网络功能的实现。Driver（包括IPAM）负责一个Network的管理，包括资源分配和回收。

有了这些关键的概念和对象，配合Docker的生命周期，通过API就能完成管理容器网络的功能。

最后，CNM还支持标签（label）。label是以key-value对定义的元数据，用户可以通过定义label这样的元数据自定义Libnetwork和驱动的行为。

2.4.2 体验CNM接口

CNM的接口较多，这里就不一一展开解释了，主要介绍几个常见的Docker daemon和CNM接口交互的过程。

1.Create Network

这一系列调用发生在使用docker network create的过程中。

（1）IpamDriver.RequestPool：创建subnetpool用于分配IP。

（2）IpamDriver.RequestAddress：为gateway获取IP。

（3）NetworkDriver.CreateNetwork：创建网络和子网。

2.Create Container

这一系列调用发生在使用docker run创建一个容器的过程中。当然，也可以通过docker network connect触发。

（1）IpamDriver.RequestAddress：为容器获取IP。

（2）NetworkDriver.CreateEndpoint：为容器创建网络接口。

（3）NetworkDriver.Join：为容器和外部网络驱动绑定。

（4）NetworkDriver.ProgramExternalConnectivity：使容器与外部进行网络连接。

（5）NetworkDriver.EndpointOperInfo：返回容器网络信息。

3.Delete Container

这一系列调用发生在使用docker delete删除一个容器的过程中。当然，也可以通过docker network disconnect触发。

（1）NetworkDriver.RevokeExternalConnectivity：撤销容器与外部网络的连接。

（2）NetworkDriver.Leave：容器与外部网络驱动解绑。

（3）NetworkDriver.DeleteEndpoint：删除容器内的网络设备。

（4）IpamDriver.ReleaseAddress：删除port并释放IP。

4.Delete Network

这一系列调用发生在使用docker network delete的过程中。（1）NetworkDriver.DeleteNetwork：删除Network。

（2）IpamDriver.ReleaseAddress：释放gateway的IP。

（3）IpamDriver.ReleasePool：删除subnetpool。

除了命令行，Docker还提供了CNM的remote plugin，即RESTful API。remote plugin监听一个指定的端口，Docker Daemon直接通过这个端口与remote plugin进行交互。这里不再赘述，感兴趣的读者可以自行查阅相关资料。

2.4.3 Libnetwork

上文多次提到Libnetwork，接下来我们就介绍CNM的原生实现——Libnetwork。Libnetwork是Docker团队将Docker的网络功能从Docker核心代码中分离出去，用Go语言实现的一个独立库。Libnetwork通过插件的形式为Docker容器提供网络功能，用户可以根据自己的需求实现自己的网络驱动，以便提供不同的网络功能。

从架构上看，Libnetwork为Docker Daemon和网络驱动提供了接口。每次用户通过命令行或API提交的与网络相关的指令，都会先在Docker Daemon预处理，再根据驱动类型调用Libnetwork模块的相应实现。因此，Libnetwork最主要的工作是联结底层驱动。

Libnetwork的网络控制器（Network Controller）负责将网络驱动和一个Docker网络进行对接。每个网络驱动负责为对接的网络提供服务，例如IPAM（IP地址管理）等。网络驱动可以按提供方被划分为原生驱动或远程驱动（第三方插件），原生驱动包括none、bridge、overlay和Macvlan。驱动也可以按照适用范围被划分为本地的和跨主机的。

Docker Daemon和Libnetwork的调用过程如图2-4所示。

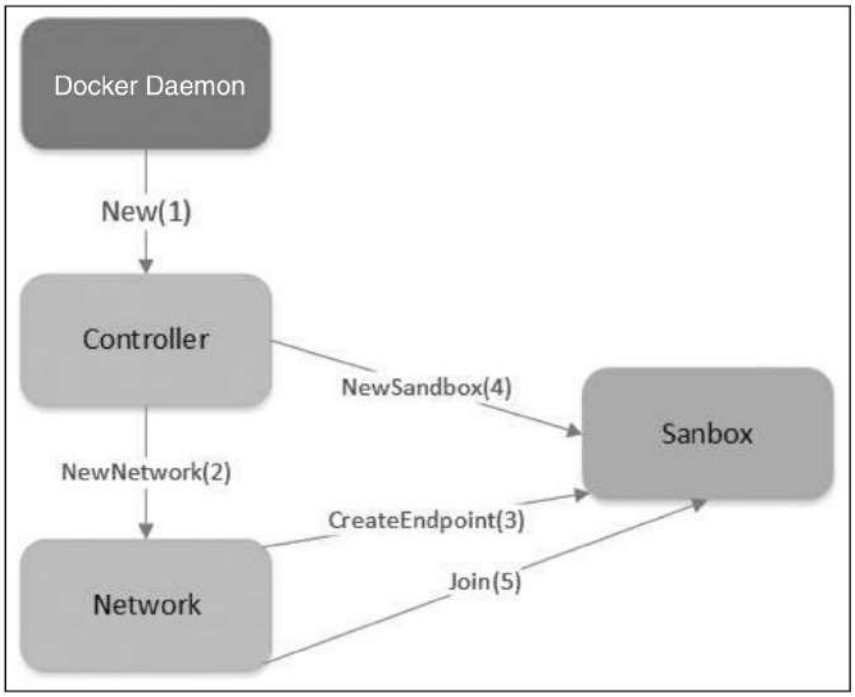


图2-4　Docker Daemon和Libnetwork的调用过程

无论使用哪种网络类型，Docker Daemon与Libnetwok的交互流程都是一样的，它很好地对底层（驱动）实现进行了抽象。如果需要实现自己的驱动，则开发者必须关注Libnetwork。

（1）Daemon创建网络控制器（Controller）实例，入参包括Docker层面的网络全局参数（genericOption），例如默认的bridge模式、kv store等。

（2）网络控制器创建容器网络，入参包括网络名、类型和对应驱动参数。从这里开始，会调用至真正的驱动实现。

（3）创建Endpoint，进行IP分配，准备网络设备接口。

（4）创建Sandbox，使用容器对应的网络命令空间，包含该容器的网络环境。

（5）已有的Endpoint加入Sandbox，完成容器与网络设备的对接。

结合CNM，Libnetwork所要达成的效果应该是：用户可以创建一个或多个网络（一个网络就是一个网桥或者一个VLAN），一个容器可以加入一个或多个网络。同一个网络中的容器可以通信，不同网络中的容器隔离。笔者认为这才是将网络从Docker分离出来的真正意义，即在创建容器之前，可以先创建网络（即创建容器与创建网络是分开的），然后决定让容器加入哪个网络。一个例子就是：

首先，创建两个overlay网络net1和net2，如下所示：

```
# docker network create -d overlay net1
```

然后，运行一个容器net1c1，加入net1网络，如下所示：

```
docker run -d --name net1c1 --net net1 nginx
```

最后，Libnetwork的架构如图2-5所示。

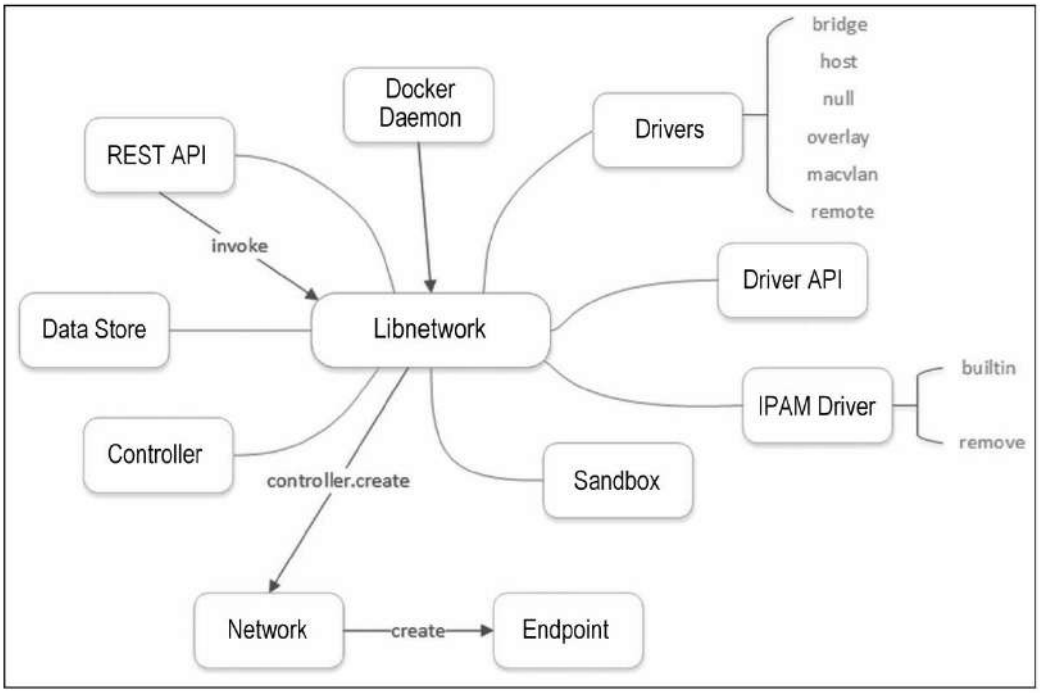


图2-5　Libnetwork的架构

2.4.4　Libnetwork扩展

前文提到，Libnetwork支持remote的网络驱动。Remote Driver提供了自定义网络的可能。远程驱动作为服务端，Libnetwork作为客户端，两者通过一系列带JSON格式的HTTP POST请求进行交互。这种形式对网络实现的可维护性、可扩展性有很大好处。对千差万别的网络实施方案而言，Docker算是解放了开发者的双手，让他们把更多精力放在自己擅长的容器方面，这一点和Kubernetes放手CNI plugin的思路非常像。远端的网络驱动，只要实现了以下接口，就能支持Docker容器网络的生命周期管理。

```
Plugin.Activate
Plugin.Deactivate
NetworkDriver.GetCapabilities
NetworkDriver.CreateNetwork
```

```
NetworkDriver.DeleteNetwork
NetworkDriver.CreateEndpoint
NetworkDriver.DeleteEndpoint
NetworkDriver.EndpointOperInfo
NetworkDriver.Join
NetworkDriver.Leave
```

CNM总计10个接口，这么看，似乎CNI要实现的接口更少一点。

接下来，以思科的Contiv项目为例，演示Libnetwork扩展方法。

Contiv是思科主导开发的以远程插件的形式，基于OVS提供Docker容器网络的SDN能力，功能上支持VLAN、VXLAN和QoS。要想使用Contiv，首先需要在/var/run/docker/plugins/目录下注册socket文件，每次处理Libnetwork的RPC请求时，通过ovsdbserver的管理接口执行修改ovs流表的操作，如图2-6所示。

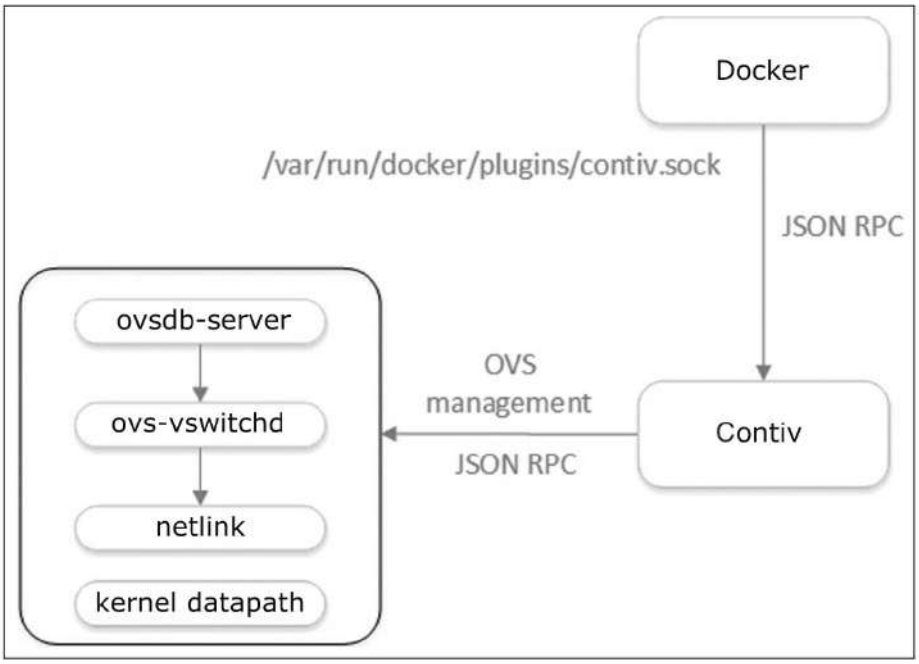


图2-6　Docker集成Contiv插件的原理

2.4.5　小结

不管是CNM还是CNI，其最终目标都是以一致的编程接口，抽象网络实现。世界上有许多广泛的网络解决方案，它们适配不同的应用场景。然而，容器生态圈存在两个网络标准，对开发人员和运维人员而言绝对不是一个好消息，因为简单、自动化的网络配置是我们的最终诉求。笔者认为，对于已经失掉容器编排标准话语权的Docker而言，CNM也是“鸡肋”，虽然曾经被

思科Contiv、Kuryr、Open Virtual Networking（OVN）、Project Calico、VMware、Weave等公司和项目采纳过，但其实更多只是“兼容”。未来的主流要看CNI！再加上2019年2月CNCF（Cloud Native Computing Foundation，云原生基金会）推出的CNF，将彻底统一云原生的网络技术标准。若读者已经采用CNM，也不必过分担心，因为CNI和CNM并非是完全不可调和的两个模型，二者是可以进行转化的。我们将在后面的章节详解CNI及如何和CNM进行转换。

2.5 天生不易：容器组网的挑战

在了解了Docker容器网络的基本概念和Docker网络子命令的常见用法后，本节将介绍Docker网络面临的挑战。

容器化应用与传统企业应用相比，拥有截然不同的特性。在过去，企业IT部门的目标在于提供不会出错的弹性系统，但随着Docker的兴起，虚拟机被更轻量级的容器方式颠覆，从而大幅降低了开发、运维、测试和部署、维护的成本，当然也带来了很多在虚拟机里没有的问题。

俗称“三驾马车”的Docker Swarm、Mesos和Kubernetes，为大规模的容器集群提供了资源调度、服务发现、扩容缩容等功能，但并未提供网络基础设施！

然而，真正要支撑大规模的容器集群，网络才是最基础的一环。因此，现在企业IT部门的首要任务变成了提供敏捷和弹性的、高度灵活的网络基础设施。网络工程师则更多地思考如何让可编程网络、软件定义网络（SDN）及网络功能虚拟化（NFV）等技术在新型的基于容器的IT基础设施中起作用。

一句话概括，容器网络的挑战：“以隔离的方式部署容器，在提供隔离自己容器内数据所需功能的同时，保持有效的连接性”，提炼两个关键词便是“隔离”和“连接”。虽然容器可以复用宿主机的基础网络“完美”解决“连接”问题，但网络隔离等问题决定了我们必须要为容器专门抽象出新的网络模型。

2.5.1 容器网络挑战综述

原生Docker容器网络的组网模型比较简单，主要是单主机模式，它基于以下几个简单的假定：

·它充分利用与容器相连接的本地Linux网桥；

·每个宿主机都有集群看得见的公共IP地址；

·每个容器都有集群看不见的专有的IP地址；

·通过NAT将容器的专有IP地址绑定到宿主机公共IP地址上；

·iptable用于容器与用户基础网络之间的网络隔离；

·负载均衡系统将服务映射至一组容器IP地址和端口。

相较于虚拟机，容器的诸多特点和以上Docker对网络模型的假定迫使我们必须忘掉一些在传统网络领域的经验，例如SDN。

容器网络最大的挑战在跨容器通信层面，具体有以下几点：

·虚拟机拥有较强的隔离机制，虚拟网卡与硬件网卡在使用上没有什么区别。由于容器基于网络隔离能力较弱的network namespace，容器网络的设计首要考虑隔离性和安全性；

·出于安全考虑，很多情况下容器会被部署在虚拟机内部，这种嵌套部署对应一个新的组网模型；

·原生Docker容器不解决跨主机通信问题，而大规模的容器部署势必涉及不同主机上的网络通信；

·容器相较虚拟机生命周期更短，重启更加频繁，重启后IP地址将发生变化，需要进行高效的网络地址管理，因此静态的IP地址分配或DHCP（耗费数秒才能生效）将不起作用；

·在创建虚拟机的时候，IP地址的分配是一种静态方式。但是在容器上面，IP地址的分配通常是动态的。在创建容器时，通常我们不知道它的IP是什么，只有真正运行后才能知道；

·容器相较虚拟机发生迁移的频率更高，且原生的Docker容器不支持热迁移，迁移后的容器面临IP地址漂移，而且经常出现不在同一网段的情况，对应的网络策略的刷新要及时跟上；

·容器的部署数量远大于虚拟机，如果为每个容器分配一个主机网络的IP地址，则可能会导致不够分配；

·大规模跨机部署带来多主机间的ARP洪泛，造成大量的资源浪费；

·大部分的容器网络方案都会使用iptables和NAT，NAT的存在会造成通信两端看不到彼此的真实IP地址，并且iptable和NAT的结合使用限制了可扩展性和性能，这让使用容器的主要优势之一荡然无存；

·大规模部署使用veth设备会影响网络性能，最严重时甚至会比裸机降低50%；

·过多的MAC地址。当MAC地址超过限制时，主机中的物理网络接口卡虽然可以切换到混杂模式，但会导致性能下降，而且顶级机架（ToR）交换机可支持的MAC地址数量也有上限。

针对以上挑战，有一些针对性的解决方案。例如，可以使用Macvlan和IPvlan替代veth，以降低处理veth网络的性能开销。当使用IPvlan驱动时，IPvlan工作在L3，而L3网络不会受到洪泛影响，因此只有主机的物理网卡MAC地址可见，容器MAC地址不会暴露在网络中，从而解决数据中心可见

MAC地址过多给ToR交换机带来的影响。

上文提到容器网络对NAT的依赖及NAT自身的局限性，而避免NAT的基本思想是将容器和虚拟机、主机同等对待。我们可以直接将容器连接到主机的网络接口来实现这一点，即容器与主机共享本地局域网（LAN）的资源，通过DHCP服务器或静态地址分配的方式给容器分配LAN的IP地址。这样一来，所有L4的容器网络端口完全暴露，虽然这种直接的暴露比管理映射端口更好，但保证安全状态需要网络策略的加持。直接连接到物理接口，即把容器的veth/Macvlan网卡与连通互联网的物理网卡桥接。不过，这种方法需要修改物理接口，即移除IP地址并将其分配到桥接接口上。

消除NAT的第二种方法是把主机变成全面的路由器，甚至是使用BGP的路由器。主机会将前缀路由到主机中的容器，每个容器会使用全球唯一的IP地址。尽管在IPv4地址空间已经耗尽的今天，给每个容器提供一个完全可路由的IPv4地址有些不太实际，但IPv6将使这种主机作为路由器的容器组网技术变得可能，事实上很多公有云都已经支持IPv6。

以上这些都只是针对某个具体问题的，而非整体的端到端的方案，例如IP地址管理、跨主机通信、消除NAT、安全策略等。Docker最初并没有提供必要的此类能力，在开始的很长一段时间内，只支持使用Linux bridge+iptables的单机网络部署方式，这种方式下容器的可见性只存在于主机内部，这严重地限制了容器集群的规模及可用性！

2.5.2 Docker的解决方案

手动配置Docker网络是一件很麻烦的事情，尽管有pipework这样的shell脚本工具，但是脚本的自动化程度相对较低，用来运维大规模的Docker网络还是太初级了。虽然Docker社区很早就意识到了这些问题，但是Docker的跨主机通信问题始终没有得到很好的解决。

Docker目前拥有两种网络解决方案，一种是原生方式，即Docker官方支持且无须配置的开箱即用。归功于收购SocketPlane，Docker才拥有了自己的一套跨主机容器连接方案。如果用户需要更多的网络功能，例如可编程、网络政策、负载均衡等，那么可以选择自定义网络插件的方式。Docker公司将其形容为“内置电池，支持更换”（batteries included，but swappable）。

在意识到可扩展、自动化的网络对大规模容器部署的重要性后，Docker公司于2015年3月收购了初创企业SocketPlane，后者的主要业务就是将软件定义网络功能以原生方式添加到Docker中。同年6月，Docker将SocketPlane技术整合至其集群管理项目Swarm中，基于Linux桥接与VXLAN解决容器的跨主机通信问题。

与此同时，Docker将网络管理从Docker Daemon中独立出来形成Libnetwork，该项目从1.9版本开始提供多种网络驱动，并支持跨主机的网络部署等功能，也允许第三方网络管理工具以插件方式替代Docker中内置的网络功能，接口标准便是CNM。

尽管Libnetwork提供了基本的跨机通信功能，但内置的网络功能相对于专业的网络插件显得比较简单。最重要的是，Docker的容器解决方案Libnetwork是CNM标准的实现。

2.5.3 第三方容器网络插件

随着Docker的兴起，一些专业的容器网络插件，例如flannel、Weave、Calico等也开始与各个容器编排平台进行集成，OpenStack社区也成立了专门的子项目Kuryr提供Neutron与容器的对接。

那么，一个容器网络插件应该提供什么功能呢？用户又该如何评估解决方案与实际使用场景的契合度呢？我们不妨把以上这些问题分成三个小问题，而第三方容器网络插件的目标无非就是解决以下三个问题。

1.你会在基于容器的基础设施上运行哪种类型的应用程序？

这直接影响到网络基础设施的蓝图及如何构建网络基础设施。你的应用程序需要复杂的网络拓扑结构及高级网络服务吗？它们是多租户模式，还是简单的“扁平网络”？

面向容器的虚拟网络解决方案让用户和网络管理员都可以定义并控制各自的网络要求。解决方案还必须提供跨多个物理主机的微分段（micro-segmentation）和隔离所需要的能力。

2.性能和可扩展性方面的要求是什么？

你在思考这个问题时，要考虑基础设施上应用程序的要求。应当考虑这样的解决方案：在一种完全分布式的架构中提供隔离和网络功能，为应用程序的发展和扩展铺平道路。随着部署的云越来越庞大，网络解决方案应该跨多个物理主机向外扩展，还应该在云编排框架里紧密地整合起来。

3.你需要将容器与虚拟机和裸机工作负载互联起来吗？

大多数应用程序需要用容器支持混合工作负载，所以要寻求同时支持两者的解决方案。一致的抽象模型（网络、子网、路由器、接口和浮动IP地址）和一套用于配置和自动化的一致的API，是完成这项工作的方法。

用户希望将面向任何工作负载的网络模型与功能强大的网络抽象结合起来，简化容器与容器、容器与主机的连接，并且保证持续增加功能的扩展性。

如果对相对成熟的第三方容器解决方案进行分类，则大致可以分为隧道方案和路由方案。关于业界通用的容器组网方案，我们将在后面的章节详细讨论。

2.5.4 小结

尽管虚拟机和容器网络是两个截然不同的网络模型，却存在同时支持容器和虚拟机的网络解决方案的实际需求，这大大增加了容器组网的技术难度。

2.6 如何做好技术选型：容器组网方案沙场点兵

当前主流的容器网络虚拟化技术有Linux bridge、Macvlan、IPvlan、Open vSwitch、flannel、Weave、Calico等。而容器网络最基础的一环是为容器分配IP地址，主流的方案有本地存储+固定网段的host-local，DHCP，分布式存储+IPAM的flannel和SDN的Weave等。

任何一个主流的容器组网方案无非就是网络虚拟机+IP地址分配，即Linux bridge、Macvlan、IPvlan、Open vSwitch、flannel、Weave、Calico等虚拟化技术和host-local、DHCP、flannel、Weave任取两样的排列组合。以Macvlan+DHCP为例，这种方案一般通过以下步骤初始化一个容器网络：

（1）创建Macvlan设备。

（2）DHCP请求一个IP地址。

（3）为Macvlan设备配置IP地址。

而路由+IPAM初始化容器网络的步骤为：

（1）向IPAM请求一个IP地址。

（2）在主机和/或网络设备上创建veth设备和路由规则。

（3）为veth设备配置IP地址。

正如前文提到的，目前业界主流的容器解决方案大致可以分为“隧道方案”和“路由方案”，而这些方案其实都可以归结为容器网络的虚拟化方案。

2.6.1 隧道方案

隧道网络也称为overlay网络，有时也被直译为覆盖网络。其实隧道方案不是容器领域新发明的概念，它在IaaS网络中应用得也比较多。overlay网络最大的优点是适用于几乎所有网络基础架构，它唯一的要求是主机之间的IP连接。但overlay网络的问题是随着节点规模的增长，复杂度也会随之增加，而且用到了封包，因此出了网络问题定位起来比较麻烦。

典型的基于overlay的网络插件有：

·Weave：源自Weaveworks，包括Weave Net、Weave Scope和Weave Flux。Weave Net是一种用于构建和部署Docker容器的网络工具；

·Open vSwitch（OVS）：基于VXLAN和GRE协议，但是性能方面损失比较严重；

·flannel：源自CoreOS，支持自研的UDP封包及Linux内核的VXLAN协议。

flannel的想法很好：每个主机负责一个网段，在这个网段里分配一个IP地址。访问另外一台主机时，通过网桥到达主机上的IP地址，这边会有一个设备，程序会把你发的包读出来，去判断你的目的地址是什么，归哪台机器管。flannel的UDP封包协议会在原始的IP包外面加一个UDP包，发到目的地址。收到之后，会把前面的UDP包扔掉，留下来的就是目标容器地址。这个方法有几个问题，第一个问题是要做封包的操作。第二个问题是每个主机上的容器是固定的，容器的不同主机之间的迁移必然带来IP的变化。使用UDP封包的一个比较大的问题是性能较差，我们会在后面的章节专门说明。

Weave和flannel用到的封包技术特点类似，不过使用的是VXLAN。另外，Weave的思路是共享IP而非绑定。在传输层先找到目的地址，然后把包发到对端，节点之间互相通过协议共享信息。

使用UDP进行封包的时候性能损失在50%以上，使用VXLAN也会有20%～30%的损耗。

2.6.2 路由方案

回过头来想一下，我们为什么要封包？其实它是改包，主要解决的问题是同一个问题，即在容器网络里，主机间不知道对方的目的地址，没有办法把IP包投递到正确的地方。传统的三层网络是用路由来互相访问的，不需要封包。至于路由规则怎么维护？传统的网络解决方案是利用BGP部署一个分布式的路由集群，如图2-7所示。

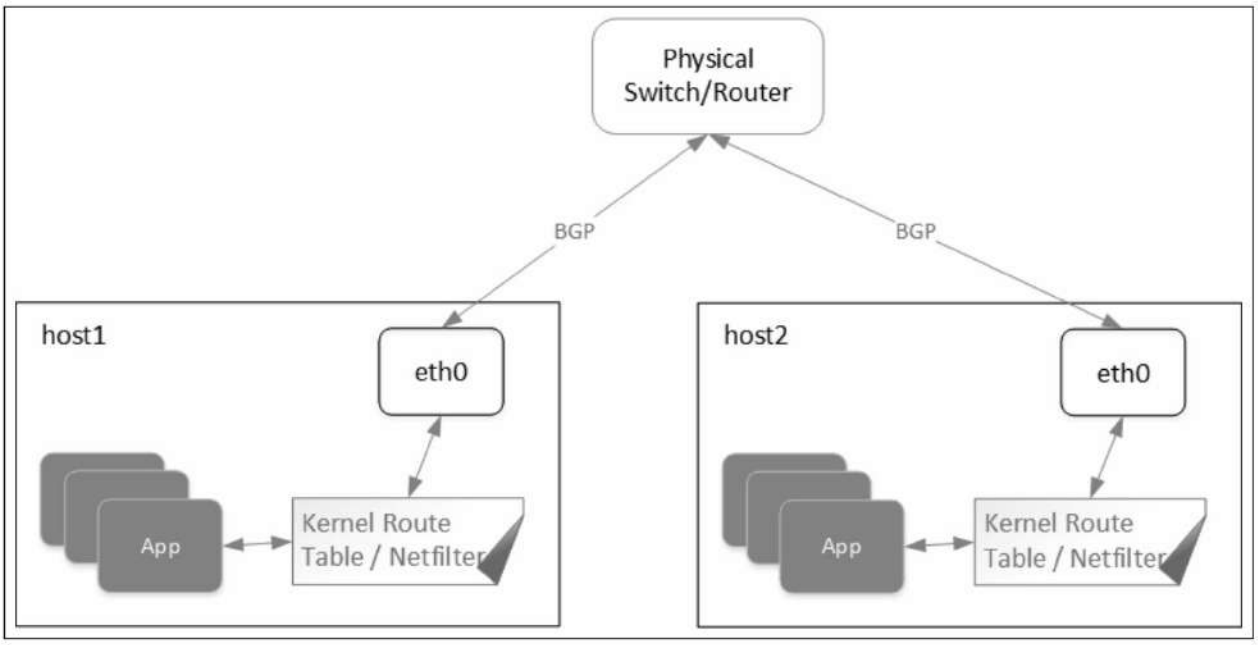


图2-7　传统BGP分布式路由集群方案

通过路由来实现，比较典型的网络插件有：

·Calico：源自Tigera，基于BGP的路由方案，支持很细致的ACL控制，对混合云亲和度比较高；

·Macvlan：从逻辑和Kernel层来看，是隔离性和性能最优的方案，基于二层隔离，需要二层路由器支持，大多数云服务商不支持，因此混合云上比较难以实现；

·Metaswitch：容器内部配一个路由指向自己宿主机的地址，这是一个纯三层的网络不存在封包，因此性能接近原生网络。

路由方案的另一个优点是出了问题也很容易排查。路由方案往往需要用户了解底层网络基础结构，因此使用和运维门槛较高。

在容器里怎么实现基于路由的网络呢？以Calico为例，Calico是一个纯三层网络方案。不同主机上的每个容器内部都配一个路由，指向自己所在的IP地址；每台服务器变成路由器，配置自己的路由规则，通过网卡直接到达目标容器，整个过程没有封包。

那么，路由交换是不是很难呢？用传统的BGP技术就可以实现。这个协议在大规模应用下是一个很好的场景，而且BGP有一个自治域的概念。在这个场景下会有一个问题，路由之间的信息交换实际上基于TCP，每两个之间都有一个TCP连接，规模大了维护这些连接的开销会非常高，具体解决方法我们会在后面的章节介绍。

Calico的设计灵感源自通过将整个互联网的可扩展IP网络原则压缩到数据中心级别。Calico在每一个计算节点，利用Linux Kernel实现高效的vRouter来负责数据转发，而每个vRouter通过BGP把自己节点上的工作负载

的路由信息向整个Calico网络传播。小规模部署可以直接互联，大规模下可通过指定的BGP Route Reflector完成。保证最终所有的容器之间的数据流量都是通过IP路由的方式完成互联的。

Calico网络拓扑图如图2-8所示。

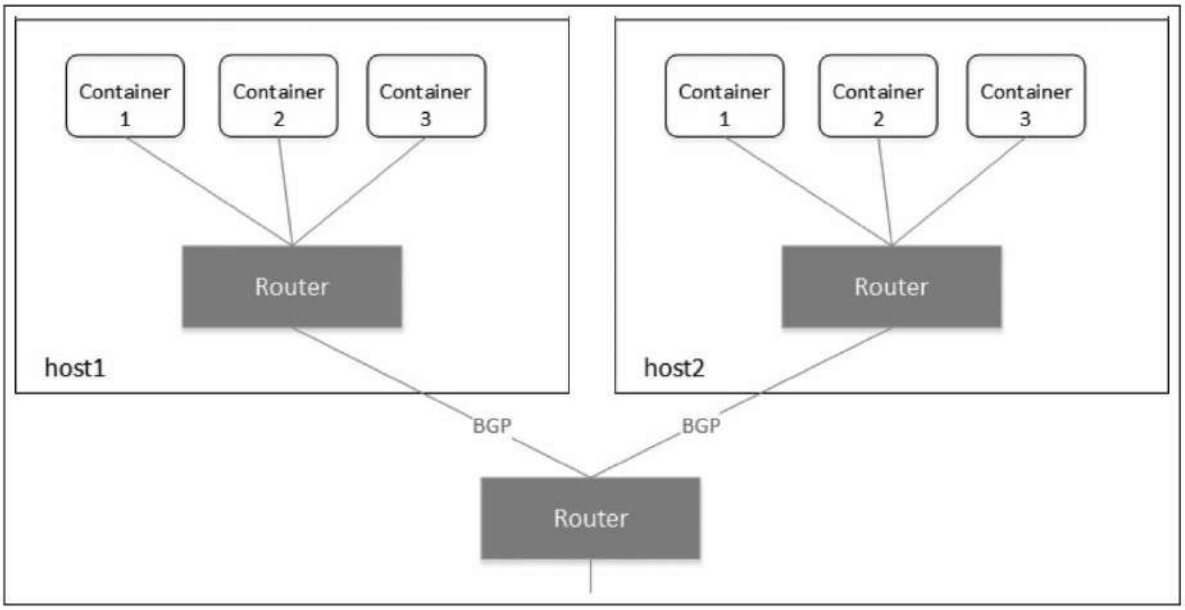


图2-8　Calico网络拓扑图

2.6.3　容器网络组网类型

关于容器网络对overlay和underlay的分类，常见的非主机网络（host network）的容器组网类型有L2 overlay、L3 overlay、L2 underlay和L3 underlay。主机网络前面章节中已有介绍，下面将分别介绍L2 overlay、L3 overlay、L2 underlay和L3 underlay。

1.overlay网络

overlay网络是在传统网络上虚拟出一个虚拟网络，承载的底层网络不再需要做任何适配。在容器的世界里，物理网络只承载主机网络通信，虚拟网络只承载容器网络通信。overlay网络的任何协议都要求在发送方对报文进行包头封装，接收方剥离包头。

L2 overlay

传统的二层网络的范围有限，L2 overlay网络是构建在底层物理网络上的L2网络，相较于传统的L2网络，L2 overlay网络是个“大二层”的概念，其中“大”的含义是可以跨越多个数据中心（即容器可以跨L3 underlay进行L2通信），而“二层”指的是通信双方在同一个逻辑的网段内，例如172.17.1.2/16和172.17.2.3/16。

VXLAN就是L2 overlay网络的典型实现，其通过在UDP包中封装原始L2报文，实现了容器的跨主机通信。

L2 overlay网络容器可在任意宿主机间迁移而不改变其IP地址的特性，使得构建在大二层overlay网络上的容器在动态迁移时具有很高的灵活性。

L3 overlay

L3 overlay组网类似L2 overlay，但会在节点上增加一个网关。每个节点上的容器都在同一个子网内，可以直接进行二层通信。跨节点的容器间通信只能走L3，都会经过网关转发，性能相比于L2 overlay较弱。牺牲的性能获得了更高的灵活性，跨节点通信的容器可以存在于不同的网段中，例如192.168.1.0/24和172.17.16.0/24。

flannel的UDP模式采用的就是L3 overlay模型。

L3 overlay网络容器在主机间迁移时可能需要改变其IP地址。

2.underlay网络

underlay网络一般理解为底层网络，传统的网络组网就是underlay类型，区别于上文提到的overlay网络。

L2 underlay

L2 underlay网络就是链路层（L2）互通的底层网络。IPvlan L2模式和Macvlan属于L2 underlay类型的网络。

L3 underlay

在L3 underlay组网中，可以选择IPvlan的L3模式，该模式下IPvlan有点像路由器的功能，它在各个虚拟网络和主机网络之间进行不同网络报文的路由转发工作。只要父接口相同，即使虚拟机/容器不在同一个网络，也可以互相ping通对方，因为IPvlan会在中间做报文的转发工作。

IPvlan的L3模式，flannel的host-gw模式和Calico的BGP组网方式都是L3 underlay类型的网络。

2.6.4 关于容器网络标准接口

笔者认为Docker 1.9以后讨论容器网络方案，不仅要看实现方式，还要看网络模型的“站队”，例如，用Docker原生的CNM，还是CoreOS，或是CNI呢？毕竟，云原生时代，标准为王！

1.CNM阵营

支持CNM标准的网络插件有：

·Docker Swarm overlay；

·Macvlan&IP network drivers；

·Calico；

·Contiv。

Docker Libnetwork的优势就是Docker原生与Docker容器生命周期结合紧密，缺点是与Docker耦合度过高。

2.CNI阵营

支持CNI标准的网络插件：

·Weave；

·Macvlan；

·flannel；

·Calico；

·Contiv；

·Mesos CNI。

CNI的优势是兼容其他容器技术（例如rkt）及上层编排系统（Kubernetes&Mesos），而且社区活跃势头迅猛，再加上Kubernetes主推，迅速成为容器网络的事实标准。

2.6.5 小结

（1）flannel仅仅作为单租户的容器互联方案是很不错的，但需要额外的组件实现更高级的功能，例如网络隔离、服务发现和负载均衡等。

（2）Weave自带DNS，一定程度上能解决服务发现的问题，但因隔离功能有限，作为多租户的联通方案还稍欠缺。

（3）Calico和Weave都使用了路由协议作为控制面，而自主路由学习在大规模网络端点下的表现其实是未经验证的，以笔者的网络工作经验看，大规模端点的拓扑计算和收敛往往需要一定的时间和计算资源。有意思的是，Calico在CNM和CNI两大阵营都扮演着比较重要的角色，既有着不俗的性能表现，又提供了良好的隔离性和ACL，但其基于三层转发的设计对物理网络架构可能会有一定的侵入性。

参考资料：https://www.bookstack.cn/read/sdn-handbook/container-kubernetes.md.

第3章　标准的胜利：Kubernetes网络原理与实践

3.1　容器基础设施的代言人：Kubernetes

云时代，企业将不再耗巨资投资自己的IT系统，而是直接使用无限制的按需付费的云服务，这无疑将显著降低IT基础设施的开销。作为程序员，笔者时常为自己有机会投身于这个波澜壮阔的技术变革事业感到莫名的激动。全面云化推进过程中最引人注目的便是“云原生”（Cloud　Native）的概念。CNCF对云原生的最原始的定义是：

> 云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。这些技术能够构建容错性好、易于管理、便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统做出频繁和可预测的重大变更。

简单地说，就是充分利用云计算的优势构建（build）和运行（run）软件，而一切技术都是围绕容器发展的。“Build　once，run　everywhere”，Docker公司提出的这个口号现在喊起来已经朗朗上口了。

有时我们会思考，当容器技术方兴未艾的时候，是什么促使了云原生技术如雨后春笋般在企业间萌发？诚然，“一次构建，到处运行”——容器良好的可移植性、敏捷性和Docker革新的镜像打包方式相较云计算最初的IaaS虚拟机形态，更容易成为公有云全新的基础设施和交付手段。但Docker毕竟只是一个运行时，当我们需要一个编排框架作为系统核心来串联开发、测试、发布、运行、升级等整个软件生命周期时，它就略显单薄。

3.1.1　Kubernetes简介

Kubernetes，简称k8s（k和s中间有8个英文字符），是谷歌照着鼎鼎大名的Borg系统开源的容器编排、调度和管理平台。如今的Kubernetes，既是GitHub上的明星项目，也获得了公有云厂商（包括AWS）的青睐。

关心容器生态圈的朋友，想必都已经接受了Kubernetes作为容器编排甚至是云原生标准的事实。

如图3-1所示，Kubernetes的logo是个船舵形象，寓意成为运输容器集装箱货轮的舵手。船舵的七个辐条来源于项目原先的名称，“Seven of Nine”。

Borg是“星际迷航”电影中的一个宇宙种族，Seven of Nine则是该种族的一名女性角色。

图3-1　Kubernetes的logo

Kubernetes通过将容器分类组成Pod。Pod是Kubernetes中特有的一个概念，它是容器分组的一层抽象，也是Kubernetes最小的部署单元。Kubernetes基于Pod提供工作负载的概念及网络、存储等能力。

和其他技术一样，大量的专有名词有可能成为入门的障碍。下面解释一些通用的术语，希望能够帮助读者理解Kubernetes。

·Master（主节点）：控制Kubernetes工作节点的机器，运行着Kubernetes的管理面，也是创建Kubernetes工作负载的地方；

·Node（工作节点）：这些机器在Kubernetes主节点的控制下将工作负载以容器的方式运行起来；

·Pod：由一个或多个容器构成的集合，作为一个整体被部署到一个节点上。同一个Pod中的容器共享网络协议栈、进程间通信（IPC）、主机名、存储其他资源。Kubernetes的Pod概念使得用户能够方便地对一组功能相似的容器的网络、存储、迁移和生命周期等进行管理。Pod运行一个或多个容器，节点运行零个或多个Pod；

·Replication Controller/Replication Set：控制Pod在集群上运行的副本数量；

·Service（服务）：将服务访问信息与具体的Pod解耦。Kubernetes服务代理负责自动将服务请求分发到正确的后端Pod处；

·Kubelet：守护进程，运行在每个工作节点上，保证该节点上容器的正常运行；

·kubectl：Kubernetes的命令行工具。

从底层实现技术的角度看，Pod内不同容器之间共享存储和一些namespace。具体来说，Pod中的容器可以共享的资源有：

·PID命名空间：Pod中的不同应用程序可以看到其他应用程序的进程ID；

·网络命名空间：Pod中的多个容器能够访问同一个IP和端口范围；

·IPC命名空间：Pod中的多个容器能够使用SystemV IPC或POSIX消息队列进行通信；

·UTS命名空间：Pod中的多个容器共享一个主机名；

·Volumes（共享存储卷）：Pod中的各个容器可以访问在Pod级别定义的存储卷。

需要注意的是，在Kubernetes1.8版本之前默认支持Pod PID namespace共享，在之后的版本中默认关闭了PID namespace共享，具体原因我们会在介绍Kubernetes的pause容器时详细解释。

本书重点讨论Kubernetes网络，因此不花大量篇幅介绍Kubernetes的基本概念，感兴趣的读者可以自行查阅其他相关资料。

关于Kubernetes为什么要发明Pod概念，在后面的章节中会专门解释。

3.1.2 Kubernetes能做什么

真实的生产环境应用会包含多个容器，而这些容器很可能会跨越多个服务器部署。Kubernetes提供了容器大规模部署、编排与管理的能力。Kubernetes提供的工作负载抽象能够让用户非常方便地构建多容器的应用服务，并且支持弹性伸缩、滚动升级和健康检查等功能。

一个初步的Linux容器应用程序把容器视作高效、快速的虚拟机。一旦把它部署到生产环境或者扩展为多个应用，随着这些容器的累积，运行环境中容器的数量就会急剧增加，复杂度也随之增大。因此，Kubernetes通常需要与镜像仓库、网络、存储、安全、监控告警等其他服务集成才能提供综合性的容器基础设施，如图3-2所示。

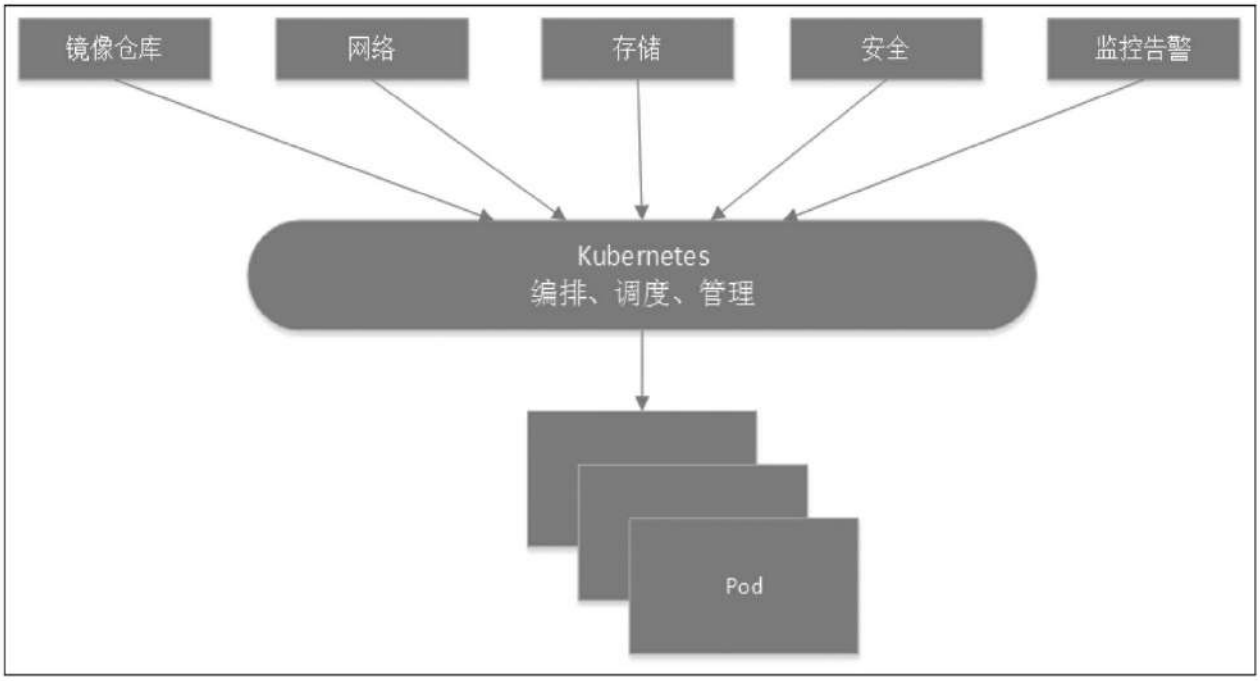


图3-2　Kubernetes及其周边服务

有了Kubernetes，用户可以：

·跨主机编排容器；

·更充分地利用硬件资源最大化地满足企业应用的需求；

·控制与自动化应用的部署与升级；

·为有状态的应用程序挂载和添加存储器；

·线上扩展或裁剪容器化应用程序与它们的资源；

·声明式的容器管理，保证所部署的应用按照我们部署的方式运作；

·通过自动布局、自动重启、自动复制、自动伸缩实现应用的状态检查与自我修复。

然而，Kubernetes依赖其他项目提供完整的编排服务，要正确实施的Kubernetes可以集成云原生领域的其他开源项目，让用户可以管理自己的容器基础设施。一些参考的项目如下所示：

·仓库：Harbor、Docker Registry等；

·网络：OpenvSwitch、CNI、Calico等；

·监控：Prometheus、Kibana、watchdog、Elastic等；

·安全：LDAP、SELinux、OAUTH、Spiffe等；

·存储：Rook、Torus等。

3.1.3 如何用Kubernetes

我们将Kubernetes视为云计算时代的操作系统，它管理着集群中的节点和在之上运行的容器。Kubernetes的Master节点从管理员处接收命令，再把指令转交给其管理的工作节点，然后在该节点上分配资源并指派Pod来完成任务请求。

所以从基础设施的角度看，管理容器的方式发生了一点小小的变化。对容器的控制在更高的层次进行，提供了更佳的控制方式，无须用户微观管理每个单独的容器或者节点。必要的工作则集中在如何指派Kubernetes主节点、定义节点、Pod和更高层次的工作负载（Workload）等问题上，如图3-3所示。

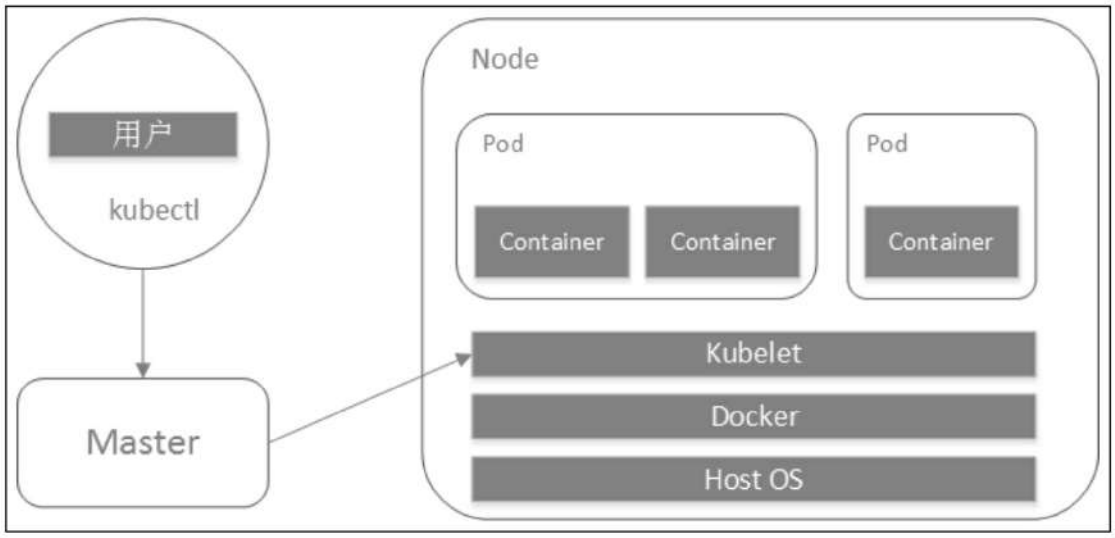


图3-3　Kubernetes工作原理

3.1.4 Docker在Kubernetes中的角色

最后，介绍Docker在Kubernetes中扮演的角色。通常情况下，Docker依然执行它原本的任务，即管理容器和镜像。当Kubernetes把Pod调度到节点上，节点上的Kubelet会指示Docker启动特定的容器。接着，Kubelet会通过cgroup持续地收集容器的信息，然后提交到Kubernetes的管理面。Docker如往常一样拉取容器镜像、启动或停止容器。

值得一提的是，随着CRI标准的提出和成熟，Kubernetes正在逐渐弱化Docker的作用。诚然，Docker对容器技术的迅速普及做出了非常重要的贡献，随着Docker中集成越来越多容器网络和编排层的功能，Docker不再是一个纯粹的容器引擎，而且稳定性也受到一定影响。Kubernetes作为编排层完全可以替用户屏蔽底层容器实现技术，提供一个更简单和通用的容器接口层，让containerd、cri-o、katacontainer等更轻量或更安全的容器技术实现

CRI接口并集成Kubernetes。

3.2 终于等到你：Kubernetes网络

结束了3.1节的铺垫，我们终于要开始对全书的核心内容——Kubernetes网络的介绍了！

Kubernetes网络包括网络模型、CNI、Service、Ingress、DNS等。

在Kubernetes的网络模型中，每台服务器上的容器有自己独立的IP段，各个服务器之间的容器可以根据目标容器的IP地址进行访问，如图3-4所示。

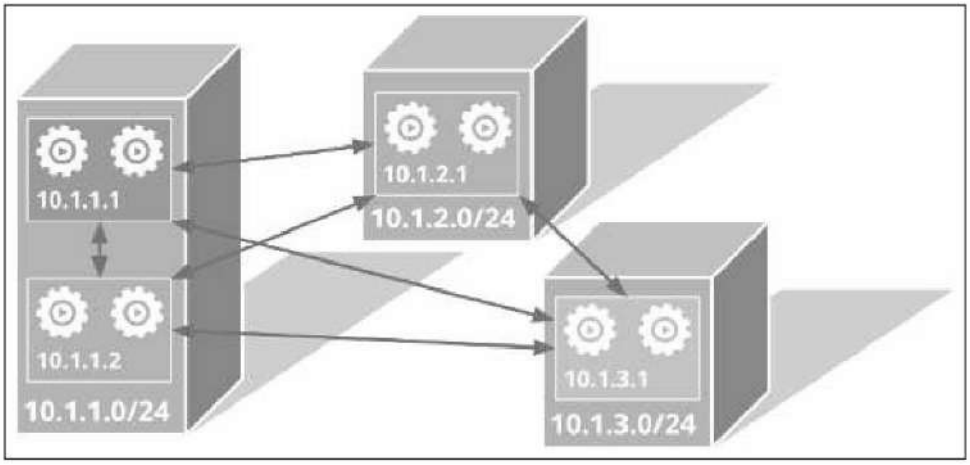


图3-4　Kubernetes网络模型概览

为了实现这一目标，重点解决以下两点：

·各台服务器上的容器IP段不能重叠，所以需要有某种IP段分配机制，为各台服务器分配独立的IP段；

·从某个Pod发出的流量到达其所在服务器时，服务器网络层应当具备根据目标IP地址，将流量转发到该IP所属IP段对应的目标服务器的能力。

总结起来，实现Kubernetes的容器网络重点需要关注两方面：IP地址分配和路由。

3.2.1 Kubernetes网络基础

在开始对Kubernetes网络进行探讨之前，我们先从Pod网络入手，简单介绍Kubernetes网络的基本概念和框架。

1.IP地址分配

Kubernetes使用各种IP范围为节点、Pod和服务分配IP地址。

·系统会从集群的VPC网络为每个节点分配一个IP地址。该节点IP用于提供从控制组件（如Kube-proxy和Kubelet）到Kubernetes Master的连接；

·系统会为每个Pod分配一个地址块内的IP地址。用户可以选择在创建集群时通过--pod-cidr指定此范围；

·系统会从集群的VPC网络为每项服务分配一个IP地址（称为ClusterIP）。大部分情况下，该VPC与节点IP地址不在同一个网段，而且用户可以选择在创建集群时自定义VPC网络。

2.Pod出站流量

Kubernetes处理Pod的出站流量的方式主要分为以下三种：

Pod到Pod

在Kubernetes集群中，每个Pod都有自己的IP地址，运行在Pod内的应用都可以使用标准的端口号，不用重新映射到不同的随机端口号。所有的Pod之间都可以保持三层网络的连通性，比如可以相互ping对方，相互发送TCP/UDP/SCTP数据包。CNI就是用来实现这些网络功能的标准接口。

Pod到Service

Pod的生命周期很短暂，但客户需要的是可靠的服务，因此Kubernetes引入了新的资源对象Service，其实它就是Pod前面的4层负载均衡器。Service总共有4种类型，其中最常用的类型是ClusterIP，这种类型的Service会自动分配一个仅集群内部可以访问的虚拟IP。

Kubernetes通过Kube-proxy组件实现这些功能，每台计算节点上都运行一个Kubeproxy进程，通过复杂的iptables/IPVS规则在Pod和Service之间进行各种过滤和NAT。

Pod到集群外

从Pod内部到集群外部的流量，Kubernetes会通过SNAT来处理。SNAT做的工作就是将数据包的源从Pod内部的IP:Port替换为宿主机的IP:Port。当数据包返回时，再将目的地址从宿主机的IP:Port替换为Pod内部的IP:Port，然后发送给Pod。当然，中间的整个过程对Pod来说是完全透明的，它们对地址转换不会有任何感知。

以上涉及的概念我们在后面的章节会进行详细讲解，本节只是抛砖引玉，不做深入分析。

3.2.2 Kubernetes网络架构综述

谈到Kubernetes的网络模型，就不能不提它著名的“单Pod单IP”模型，即

每个Pod都有一个独立的IP，Pod内所有容器共享network namespace（同一个网络协议栈和IP）。

“单Pod单IP”网络模型为我们勾勒了一个Kubernetes扁平网络的蓝图，在这个网络世界里：容器是一等公民，容器之间直接通信，不需要额外的NAT，因此不存在源地址被伪装的情况；Node与容器网络直连，同样不需要额外的NAT。扁平化网络的优点在于：没有NAT带来的性能损耗，而且可追溯源地址，为后面的网络策略做铺垫，降低网络排错的难度等。

总体而言，集群内访问Pod，会经过Service；集群外访问Pod，经过的是Ingress。Service和Ingress是Kubernetes专门为服务发现而抽象出来的相关概念，后面会做详细讨论。

与CRI之于Kubernetes的runtime类似，Kubernetes使用CNI作为Pod网络配置的标准接口。需要注意的是，CNI并不支持Docker网络，也就是说，docker0网桥会被大部分CNI插件“视而不见”。

当然也有例外，Weave就是一个会处理docker0的CNI插件，具体分析请看后面章节的内容。

Kubernetes网络总体架构如图3-5所示。

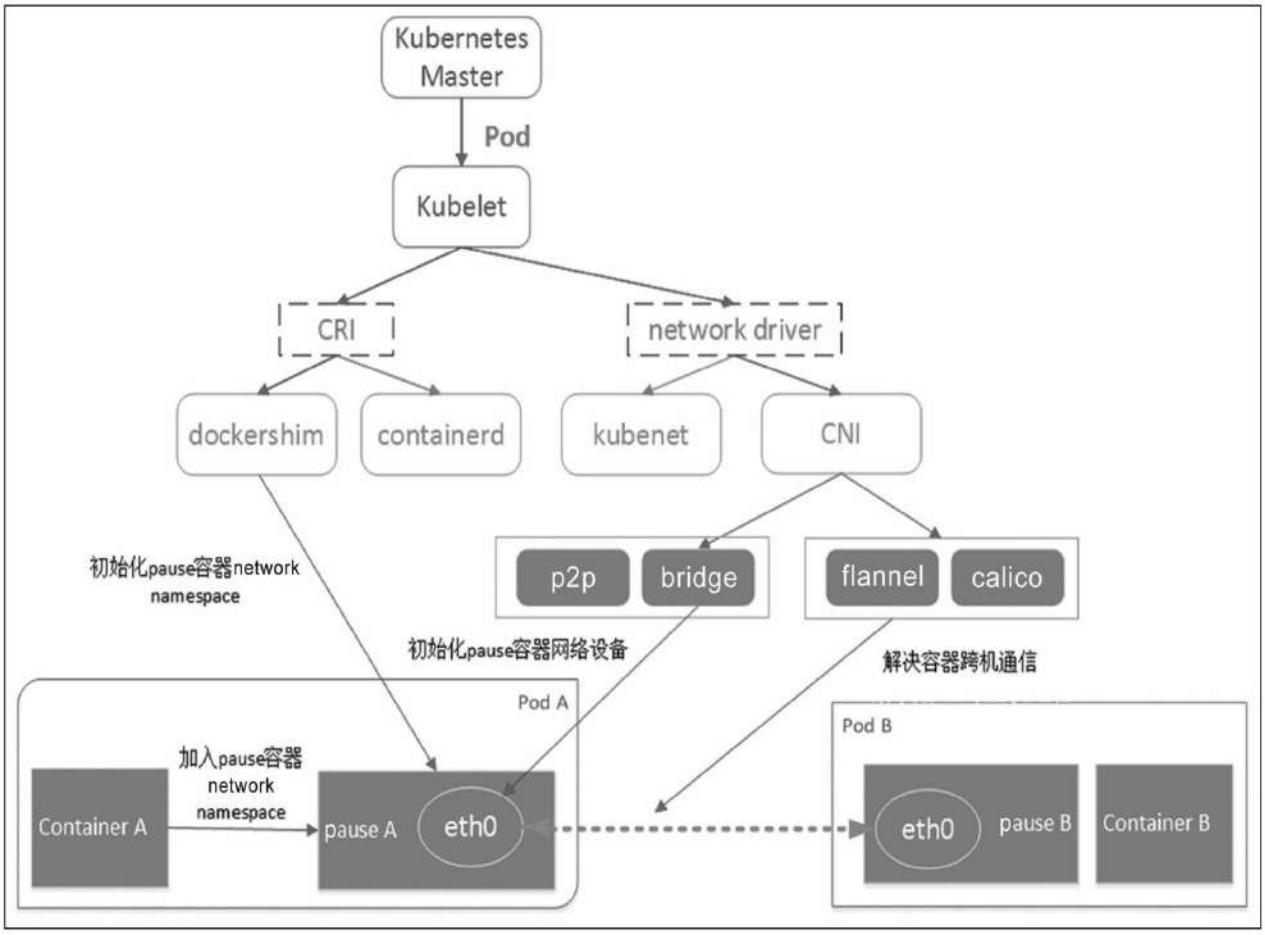


图3-5　Kubernetes网络总体架构

图3-5描绘了当用户在Kubernetes里创建了一个Pod后，CRI和CNI协同创建Pod所属容器，并为它们初始化网络协议栈的全过程。具体过程如下：

（1）当用户在Kubernetes的Master里创建了一个Pod后，Kubelet观察到新Pod的创建，于是首先调用CRI（后面的runtime实现，比如dockershim、containerd等）创建Pod内的若干个容器。

（2）在这些容器里，第一个被创建的pause容器是比较特殊的，这是Kubernetes系统“赠送”的容器，也称pause容器。里面运行着一个功能十分简单的C程序，具体逻辑是一启动就把自己永远阻塞在那里。一个永远阻塞而且没有实际业务逻辑的pause容器到底有什么用呢？用处很大。我们知道容器的隔离功能利用的是Linux内核的namespace机制，而只要是一个进程，不管这个进程是否处于运行状态（挂起亦可），它都能“占”用着一个namespace。因此，每个Pod内的第一个系统容器pause的作用就是占用一个Linux的network namespace。

（3）Pod内其他用户容器通过加入这个network namespace的方式共享同一个network namespace。用户容器和pause容器之间的关系有点类似于寄居蟹和海螺。因此，Container runtime创建Pod内的用户容器时，调用的都是同一个命令：docker run--net=none。意思是只创建一个network namespace，不初始化网络协议栈。如果这个时候通过nsenter方式进入容器，会看到里面只有一个本地回环设备lo。

（4）容器的eth0是怎么创建出来的呢？答案是CNI。CNI主要负责容器的网络设备初始化工作。Kubelet目前支持两个网络驱动，分别是Kubenet和CNI。Kubenet是一个历史产物，即将废弃，因此本节不过多介绍。CNI有多个实现，官方自带的插件就有p2p、bridge等，这些插件负责初始化pause容器的网络设备，也就是给pause容器内的eth0分配IP等，到时候，Pod内其他容器就使用这个IP与其他容器或节点进行通信。Kubernetes主机内容器的默认组网方案是bridge。flannel、Calico这些第三方插件解决容器之间的跨机通信问题，典型的跨机通信解决方案有bridge和overlay等。

3.2.3 Kubernetes主机内组网模型

Kubernetes经典的主机内组网模型是veth pair+bridge的方式。

前文提到，当Kubernetes调度Pod在某个节点上运行时，它会在该节点的Linux内核中为Pod创建network namespace，供Pod内所有运行的容器使用。从容器的角度看，Pod是具有一个网络接口的物理机器，Pod中的所有容器都会看到此网络接口。因此，每个容器通过localhost就能访问同一个Pod内的其他容器。

Kubernetes使用veth　pair将容器与主机的网络协议栈连接起来，从而使数据包可以进出Pod。容器放在主机根network namespace中veth pair的一端连接到Linux网桥，可让同一节点上的各Pod之间相互通信，如图3-6所示。

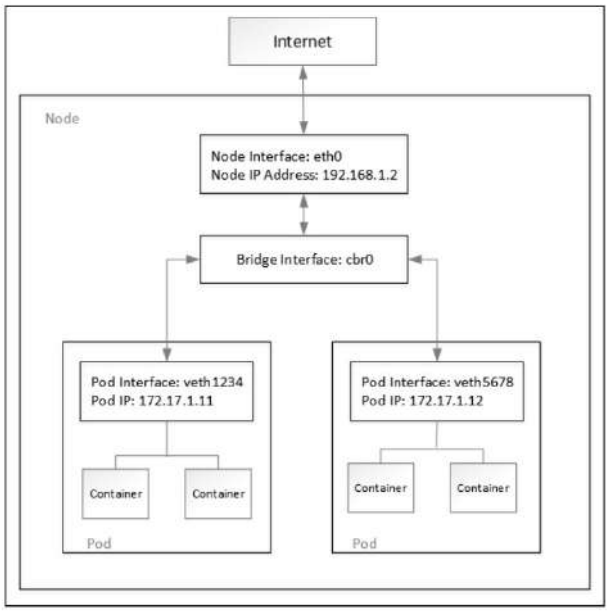


图3-6　Kubernetes bridge网络模型

如果Kubernetes集群发生节点升级、修改Pod声明式配置、更新容器镜像或节点不可用，那么Kubernetes就会删除并重新创建Pod。在大部分情况下，Pod创建会导致容器IP发生变化。也有一些CNI插件提供Pod固定IP的解决方案，例如Weave、Calico等。

3.2.4　Kubernetes跨节点组网模型

前文提到，Kubernetes典型的跨机通信解决方案有bridge、overlay等，下面我们将简单介绍这两种方案的基本思路。

Kubernetes的bridge跨机通信网络模型如图3-7所示。

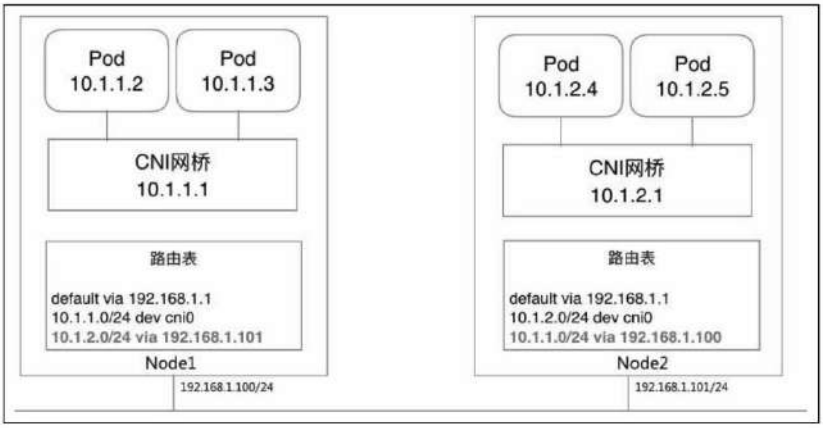


图3-7　Kubernetes的bridge跨机通信网络模型

如图3-7所示，Node1上Pod的网段是10.1.1.0/24，接的Linux网桥是10.1.1.1，Node2上Pod的网段是10.1.2.0/24，接的Linux网桥是10.1.2.1，接在同一个网桥上的Pod通过局域网广播通信。我们发现，Node1上的路由表的第二条是：

```
10.1.1.0/24 dev cni0
```

意思是，所有目的地址是本机上Pod的网络包，都发到cni0这个Linux网桥，进而广播给Pod。

注意看第三条路由规则：

```
10.1.2.0/24 via 192.168.1.101
```

10.1.2.0/24是Node2上Pod的网段，192.168.1.101又恰好是Node2的IP。意思是，目的地址是10.1.2.0/24的网络包，发到Node2上。这时，我们观察Node2上面的第二条路由信息：

```
10.1.2.0/24 dev cni0
```

就会知道，这个包会被接着发给Node2上的Linux网桥cni0，再广播给目标Pod。回程报文同理（走一条逆向的路径）。因此，我们可以得出一个结论：bridge网络本身不解决容器的跨机通信问题，需要显式地书写主机路由表，映射目标容器网段和主机IP的关系，集群内如果有N个主机，需要N-1条路由表项。

至于overlay网络，它是构建在物理网络之上的一个虚拟网络，其中VXLAN是主流的overlay标准。VXLAN就是用UDP包头封装二层帧，即所谓的MAC in UDP。图3-8所示为典型的overlay网络的拓扑图。

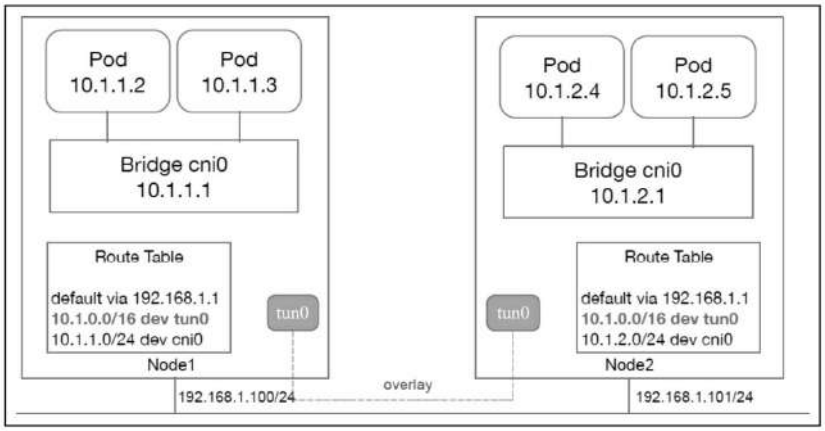


图3-8　典型的overlay网络的拓扑图

和bridge网络类似，Pod同样接在Linux网桥上，目的地址落在本机Pod网段的网络包同样发给Linux网桥cni0。不同的是，目的Pod在其他节点上的路由表规则，例如：

```
10.1.0.0/16 dev tun0
```

这次是直接发给本机的tun/tap设备tun0，而tun0就是overlay隧道网络的入口。我们注意到，集群内所有机器都只需要这么一条路由表，不需要像bridge网络那样，写N-1条路由表项。如何将网络包正确地传递到目标主机的隧道口另一端呢？以flannel的实现为例，它会借助一个分布式的数据库，记录目的容器IP与所在主机的IP的映射关系，而且每个节点上都会运行一个agent。例如，flanneld会监听在tun0上进行的封包和解包操作。例如，Node1上的容器发包给Node2上的容器，flanneld会在tun0处将一个目的地址是192.168.1.101:8472的UDP包头（校验和置成0）封装到这个包的外层，然后借着主机网络的东风顺利到达Node2。监听在Node2的tun0上的flanneld捕获这个特殊的UDP包（检验和为0），知道这是一个overlay的封包，于是解开UDP包头，将它发给本机的Linux网桥cni0，进而广播给目的容器。

bridge和overlay是Kubernetes最早采用的跨机通信方案，但随着集成Weave和Calico等越来越多的CNI插件，Kubernetes也支持虚拟路由等方式，在后面的章节中会详细介绍。

3.2.5 Pod的hosts文件

与宿主机一样，容器也有/etc/hosts文件，用来记录容器的hostname和IP地址的映射关系。通过向Pod的/etc/hosts文件中添加条目，可以在Pod级别覆盖对hostname的解析。

当一个Pod被创建后，默认情况下，hosts文件只包含IPv4和IPv6的样板内容，例如localhost和主机名称。除了默认的样板内容，我们可能有向Pod的hosts文件添加额外的条目的需求，例如将foo.local、bar.local解析为127.0.0.1，将foo.remote、bar.remote解析为10.1.2.3。怎么办呢？总不能让用户手动修改hosts文件吧？在Kubernetes 1.7版本以后，Kubernetes提供downward API，支持用户通过PodSpec的HostAliases字段添加这些自定义的条目，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: hostaliases-pod
spec:
  hostAliases:
  - ip: "127.0.0.1"
```

```
    hostnames:
    - "foo.local"
    - "bar.local"
  - ip: "10.1.2.3"
    hostnames:
    - "foo.remote"
    - "bar.remote"
  containers:
  - name: cat-hosts
    image: busybox
```

于是，Pod的hosts文件的内容类似如下：

```
# cat /etc/hosts
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
fe00::0 ip6-mcastprefix
fe00::1 ip6-allnodes
fe00::2 ip6-allrouters
10.200.0.4 hostaliases-pod
127.0.0.1   foo.local
127.0.0.1   bar.local
10.1.2.3    foo.remote
10.1.2.3    bar.remote
```

为什么我们不建议用户在Docker容器启动后手动修改Pod的/etc/hosts文件，而是建议通过使用HostAliases的方式进行修改呢？最主要的原因是该文件由Kubelet托管，用户修改该hosts文件的任何内容都会在容器重启或Pod重新调度后被Kubelet覆盖。

注：如果Pod启用了hostNetwork（即使用主机网络），那么将不能使用HostAliases特性，因为Kubelet只管理非hostNetwork类型Pod的hosts文件。

3.2.6 Pod的hostname

Docker使用UTS namespace进行主机名（hostname）隔离，而Kubernetes的Pod也继承了Docker的UTS namespace隔离技术，即Pod之间主机名相互隔离，但Pod内容器分享同一个主机名。

Docker主要有两种使用UTS namespace的用法：

·第一种是docker run--uts=""busybox。这种用法在创建容器的同时会新创建一个UTS namespace；

·第二种是docekr run--uts="host"busybox。这种用法在创建容器的同时会使用物理机的UTS namespace。

除此之外，Kubernetes在处理UTS namespace时也会考虑Pod的网络模式。

```
func modifyHostNetworkOptionForContainer(hostNetwork bool, sandboxID string, hc *
dockercontainer.HostConfig) {
    sandboxNSMode := fmt.Sprintf("container:%v", sandboxID)
    hc.NetworkMode = dockercontainer.NetworkMode(sandboxNSMode)
    hc.IpcMode = dockercontainer.IpcMode(sandboxNSMode)
    hc.UTSMode = ""

    if hostNetwork {
        hc.UTSMode = namespaceModeHost
    }
}
```

如上所示，从代码里我们可以分析出：如果Kubelet判断Pod使用宿主机网络（即host-Network），则会将UTS的mode设置为“host”，也就是使用物理机的UTS namespace。因此，如果这时容器修改主机名，则会影响宿主机的主机名。

如果容器想要修改主机名（通过hostname命令），则需要privileged权限。修改容器主机名后，容器重启或被Kubelet重建都会恢复成原来的主机名，主要原因是容器重启会导致创建新的UTS namespace。

一个Pod内如果有多个容器，修改任意一个容器的hostname都会影响其他容器，因为Pod共享UTS namespace。

以下是完整的实验过程：首先创建一个容器，指定容器名为busyboxtest：

```
# docker run -d busybox --name busyboxtest
6f57f6e459df5644a1db4bc9ad2206a0f99e40312de1892695f8a09d52faa9c1
```

然后进入该容器查看hostname：

```
# docker exec 6f5 hostname
busyboxtest
```

修改该容器的hostname为test123，如下所示：

```
# docker exec 6f5 hostname test123
# docker exec 6f5 hostname
test123
```

为了证明容器重启带来的UTS namespace的改变会导致hostname被覆盖，我们先找到该容器对应的PID，然后查看该容器的namespace文件链接：

```
# docker inspect 6f5 | grep Pid
"Pid": 15818,
# ll /proc/15818/ns/
total 0
lrwxrwxrwx 1 root root 0 Jan 10 16:16 ipc -> ipc:[4026535715]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 mnt -> mnt:[4026535789]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 net -> net:[4026535718]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 pid -> pid:[4026535791]
lrwxrwxrwx 1 root root 0 Jan 10 16:18 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 uts -> uts:[4026535790]
```

重启容器（容器ID保持不变，但PID发生了变化）：

```
# docker restart 6f5
6f5
# docker inspect 6f5 | grep Pid
"Pid": 17553,
# ll /proc/17553/ns/
total 0
lrwxrwxrwx 1 root root 0 Jan 10 16:19 ipc -> ipc:[4026535715]
lrwxrwxrwx 1 root root 0 Jan 10 16:19 mnt -> mnt:[4026535341]
lrwxrwxrwx 1 root root 0 Jan 10 16:19 net -> net:[4026535718]
lrwxrwxrwx 1 root root 0 Jan 10 16:19 pid -> pid:[4026535714]
lrwxrwxrwx 1 root root 0 Jan 10 16:19 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jan 10 16:19 uts -> uts:[4026535713]

# docker exec 6f5 hostname
busyboxtest
```

如上所示，可以看到容器重启后UTS的namespace发生了变化，而且对hostname的修改也被覆盖了。

3.3 Pod的核心：pause容器

当检查Kubernetes集群的节点时，在节点上执行命令docker ps，用户可能会注意到一些被称为pause的容器，例如：

```
# docker ps
CONTAINER ID IMAGE COMMAND ...
...
3b45e983c859 gcr.io/google_containers/pause-amd64:3.0 "/pause" ...
dbfc35b00062 gcr.io/google_containers/pause-amd64:3.0 "/pause" ...
c4e998ec4d5d gcr.io/google_containers/pause-amd64:3.0 "/pause" ...
508102acf1e7 gcr.io/google_containers/pause-amd64:3.0 "/pause" ...
```

Kubernetes中的pause容器可以说是网络模型的精髓，理解pause容器能够更好地理解Kubernetes Pod的设计初衷。

我们知道Kubernetes的Pod抽象基于Linux的namespace和cgroups，为容器提供了隔离的环境。从网络的角度看，同一个Pod中的不同容器犹如运行在同一个主机上，可以通过localhost进行通信。

为什么要发明Pod呢？直接使用Docker容器不好吗？Docker容器非常适合部署单个软件单元。但是当你想要一起运行多个软件时，尤其是在一个容器里管理多个进程时，这种模式会变得有点麻烦。Kubernetes非常不建议“富容器”这种方式，认为将这些应用程序部署在部分隔离并且部分共享资源的容器组中更为有用。为此，Kubernetes为这种使用场景提供了一个称为Pod的抽象。

原则上，任何人都可以配置Docker来控制容器组之间的共享级别——只需创建一个父容器，并创建与父容器共享资源的新容器，然后管理这些容器的生命周期。在Kubernetes中，pause容器被当作Pod中所有容器的“父容器”，并为每个业务容器提供以下功能：

·在Pod中，它作为共享Linux namespace（Network、UTS等）的基础；

·启用PID namespace共享，它为每个Pod提供1号进程，并收集Pod内的僵尸进程。

pause容器源码

在Kubernetes中，pause容器运行着一个非常简单的进程，它不执行任何功能，基本上是永远“睡觉”的，源代码在Kubernetes项目的build/pause/目录

中。它比较简单，完整的源代码如下所示：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define STRINGIFY(x) #x
#define VERSION_STRING(x) STRINGIFY(x)

#ifndef VERSION
#define VERSION HEAD
#endif

static void sigdown(int signo) {
  psignal(signo, "Shutting down, got signal");
  exit(0);
}

static void sigreap(int signo) {
  while (waitpid(-1, NULL, WNOHANG) > 0)
    ;
}

int main(int argc, char **argv) {
  int i;
  for (i = 1; i < argc; ++i) {
    if (!strcasecmp(argv[i], "-v")) {
      printf("pause.c %s\n", VERSION_STRING(VERSION));
      return 0;
    }
  }

  if (getpid() != 1)
    /* Not an error because pause sees use outside of infra containers. */
```

```
    fprintf(stderr, "Warning: pause should be the first process\n");

  if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 1;
  if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown}, NULL) < 0)
    return 2;
  if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
                                             .sa_flags = SA_NOCLDSTOP},
                                             NULL) < 0)
    return 3;

  for (;;)
    pause();
  fprintf(stderr, "Error: infinite loop terminated\n");
  return 42;
}
```

如上所示，这个pause容器运行一个非常简单的进程，它不执行任何功能，一启动就永远把自己阻塞住了（见pause（）系统调用）。正如你看到的，它当然不会只知道“睡觉”。它执行另一个重要的功能——即它扮演PID 1的角色，并在子进程成为“孤儿进程”的时候，通过调用wait（）收割这些僵尸子进程。这样就不用担心我们的Pod的PID namespace里会堆满僵尸进程了。这也是为什么Kubernetes不随便找个容器（例如Nginx）作为父容器，让用户容器加入的原因。

1.从namespace看pause容器

第1章介绍过，在Linux系统中运行新进程时，该进程从父进程继承了其namespace。在namespace中运行进程的方法是通过取消与父进程的共享namespace，从而创建一个新的namespace。以下是使用unshare工具在新的PID、UTS、IPC和Mount namespace中运行shell的示例。

```
# unshare --pid --uts --ipc --mount -f chroot rootfs /bin/sh
```

一旦进程运行，用户可以将其他进程添加到该进程的namespace中以形成一个Pod，Pod中的容器在其中共享namespace。读者可以使用第1章提到的setns系统调用将新进程添加到现有namespace，Docker也提供命令行功能自动完成此过程。下面来看一下如何使用pause容器和共享namespace创建Pod。

首先，我们使用Docker启动pause容器，以便可以将其他容器添加到Pod中，如下所示：

```
# docker run -d --name pause gcr.io/google_containers/pause-amd64:3.0
```

然后，我们在Pod中运行其他容器，分别是Nginx代理和ghost博客应用。

Nginx代理的后端配置成http://127.0.0.1:2368，也就是ghost进程监听的地址，如下所示：

```
# cat <<EOF >> nginx.conf
> error_log stderr;
> events { worker_connections  1024; }
> http {
>     access_log /dev/stdout combined;
>     server {
>         listen 80 default_server;
>         server_name example.com www.example.com;
>         location / {
>             proxy_pass http://127.0.0.1:2368;
>         }
>     }
> }
> EOF

# docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf -p 8080:80 --
net=container:pause --ipc=container:pause --pid=container:pause nginx
```

为ghost博客应用程序创建另一个容器，如下所示：

```
# docker run -d --name ghost --net = container: pause --ipc = container: pause --pid
= container: pause ghost
```

在这个例子中，我们将pause容器指定为要加入其namespace的容器。如果访问http://localhost:8080/，那么应该能够看到ghost通过Nginx代理运行，因为pause、Nginx和ghost容器之间共享network namespace，如图3-9所示。

通过Pod，Kubernetes屏蔽了以上所有复杂度。

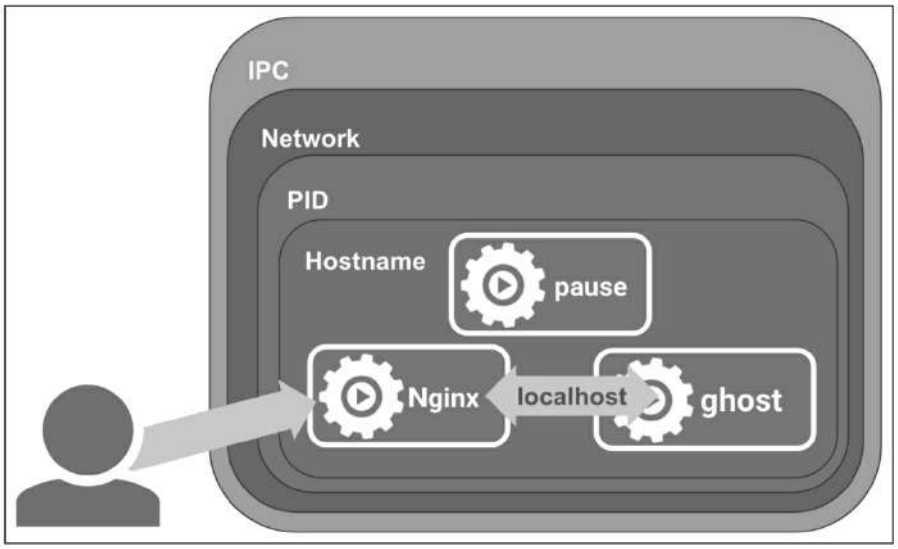


图3-9　Pod的底层实现原理

2.从PID看pause容器

在UNIX系统中，PID为1的进程是init进程，即所有进程的父进程。init进程比较特殊，它维护一张进程表并且不断地检查其他进程的状态。init进程的其中一个作用是当某个子进程由于父进程的错误退出而变成了“孤儿进程”，便会被init进程“收养”并在该进程退出时回收资源。

进程可以使用fork和exec两个系统调用启动其他进程。当启动了其他进程后，新进程的父进程就是调用fork系统调用的进程。fork用于启动正在运行的进程的另一个副本，而exec则用于启动不同的进程。每个进程在操作系统进程表中都有一个条目。这将记录有关进程的状态和退出代码。当子进程运行完成后，它的进程表条目仍然保留，直到父进程使用wait系统调用获得其退出代码后才会清理进程条目。这被称为“收割”僵尸进程，并且僵尸进程无法通过kill命令清除。

僵尸进程是已停止运行但进程表条目仍然存在的进程，父进程尚未通过wait系统调用进行检索。从技术层面来说，终止的每个进程都算是一个僵尸进程，尽管只是在很短的时间内发生的。当用户程序写得不好并且简单地省略wait系统调用，或者当父进程在子进程之前异常退出并且新的父进程没有调用wait检索子进程时，会出现较长时间的僵尸进程。系统中存在过多僵尸进程将占用大量操作系统进程表资源。

当进程的父进程在子进程完成前退出时，OS将子进程分配给init进程。init进程“收养”子进程并成为其父进程。这意味着当子进程退出时，新的父进程（init进程）必须调用wait获取其退出代码，否则其进程表项将一直保留，并且它也将成为一个僵尸进程。同时，init进程必须拥有“信号屏蔽”功能，不能处理某个信号逻辑，从而防止init进程被误杀。所以不是随随便便一个进程都能当init进程的。

容器使用PID namespace对PID进行隔离，因此每个容器中均可以有独立的init进程。当在主机上发送SIGKILL或者SIGSTOP（也就是docker kill或者docker stop命令）强制终止容器的运行时，其实就是在终止容器内的init进程。一旦init进程被销毁，同一PID namespace下的进程也随之被销毁。

在容器中，必须要有一个进程充当每个PID namespace的init进程，使用Docker的话，ENTRYPOINT进程是init进程。如果多个容器之间共享PID namespace，那么拥有PID namespace的那个进程须承担init进程的角色，其他容器则作为init进程的子进程添加到PID namespace中。

为了给读者一个直观的印象，下面给出一个例子来说明用户容器和pause容器的PID关系。

先启动一个pause容器：

```
# docker run -idt --name pause gcr.io/google_containers/pause-amd64:3.0
7f6e459df5644a1db4bc9ad2206a0f99e40312de1892695f8a09d52faa9c1073
```

再运行一个busybox容器，加入pause容器的namespace（Network、PID、IPC）中：

```
# docker run -idt --name busybox --net=container:pause --pid=container:pause --ipc=
container:pause busybox
ad3029c55476e431101473a34a71516949d1b7de3afe3d505b51d10c436b4b0f
```

上述这种加入pause容器的方式也是Kubernetes启动Pod的原理。

接下来，让我们进入busybox容器查看里面的进程，发现里面PID=1的进程是/pause：

```
# docker exec -it ad3029c55476 /bin/sh
/ # ps aux
PID   USER     TIME   COMMAND
    1 root       0:00 /pause
    5 root       0:00 sh
    9 root       0:00 /bin/sh
   13 root       0:00 ps aux
```

我们完全可以在父容器中运行Nginx，并将ghost添加到Nginx容器的PID命名空间：

```
# docker run -d --name nginx -v  pwd /nginx.conf:/etc/nginx/nginx.conf -p 8080:80
nginx
# docker run -d --name ghost --net=container:nginx --ipc=container:nginx --pid=
container:nginx ghost
```

在这种情况下，Nginx将承担PID 1的作用，并将ghost添加为Nginx的子进程。虽然这样貌似不错，但从技术角度看，Nginx需要负责ghost进程的所有子进程。例如，如果ghost在其子进程完成之前异常退出，那么这些子进程将被Nginx收养。但是，Nginx并不是设计用来作为一个init进程运行并收割僵尸进程的。这意味着将会有很多这种僵尸进程，并且这种情况将持续整个容器的生命周期。

最后总结一句，Pod的init进程，pause容器舍它其谁？

3.在**Kubernetes**中使用**PID namespace**共享/隔离

共享/隔离Pod内容器的PID namespace是一个见仁见智的问题。支持共享的人觉得方便了进程间通信，例如可以在容器中给另外一个容器内的进程发送信号，还不用担心僵尸进程回收问题。

在Kubernetes 1.8版本之前，默认是启用PID namespace共享的，除非将Kubelet的标志--docker-disable-shared-pid设置成true，来禁用PID namespace共享。然而在Kubernetes 1.8版本以后，情况刚好相反，在默认情况下，Kubelet标志--docker-disable-sharedpid设置成true，如果要开启，还要设置成false。下面就来看看Kubernetes提供的关于是否共享PID namespace的downward API。

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox
    securityContext:
      capabilities:
        add:
        - SYS_PTRACE
    stdin: true
    tty: true
```

如上所示，podSpec.shareProcessNamespace指示了是否启用PID namespace共享。

通过前文的讨论，我们知道Pod内容器共享PID namespace是很有意义的，那为什么还要开放这个禁止PID namesapce共享的开关呢？那是因为当应用程序不会产生其他进程，而且僵尸进程带来的问题可以忽略不计时，就用不到PID namespace的共享了。在有些场景下，用户希望Pod内容器能够与其他容器隔离PID namespace，例如下面两个场景：

（1）PID namespace共享时，由于pause容器成了PID=1，其他用户容器就没有PID 1了。但像systemd这类镜像要求获得PID 1，否则无法正常启动。有些容器通过kill-HUP 1命令重启进程，然而在由pause容器托管init进程的Pod里，上面这条命令只会给pause容器发信号。

（2）PID namespace共享Pod内不同容器的进程对其他容器是可见的。这包括/proc中可见的所有信息，例如，作为参数或环境变量传递的密码，这将带来一定的安全风险。

3.4 打通CNI与Kubernetes：Kubernetes网络驱动

Kubernetes支持两种网络驱动，分别是Kubenet和CNI，其中：

·CNI plugins：遵守appc/CNI规范，允许自由接入多个符合CNI标准的网络插件；

·Kubenet plugins：基于cbr0的一个单机容器网络方案，同时使用CNI的bridge和host-local插件实现一些功能。

3.4.1 即将完成历史使命：Kubenet

Kubenet是Kubernetes早期原生的网络驱动，提供非常简单和基本的单机容器网络能力。使用Kubelet必须运行--network-plugin=kubenet参数，以开启插件。

Kubenet本身不会实现更高级的功能，如跨节点网络连接或网络策略等，它通常与cloud provider一起使用，配置路由规则实现跨主机的通信能力。随着CNI的广泛使用，Kubenet正在被慢慢弃用。然而，Kubenet也用了一些CNI的功能，例如它基于CNI bridge插件创建名为cbr0的Linux网桥，为每个容器建立一对veth pair并连接到cbr0网桥上。每个Pod通过配置的网段分配到一个IP地址。

Kubenet在bridge插件的基础上拓展了很多功能，包括：

·使用CNI的host-local IP地址管理插件，为Pod分配IP地址，并定期释放已分配但未使用的IP地址；

·设置sysctl net.bridge.bridge-nf-call-iptables=1；

·为Pod的IP配置SNAT（MASQUERADE）规则，允许Pod访问因特网；

·开启网桥的hairpin和Promisc模式，允许Pod访问它自己所在的Service IP（即通过NAT后再访问Pod）；

·HostPort管理及设置端口映射；

·带宽控制，支持通过kubernetes.io/ingress-bandwidth和kubernetes.io/egress-bandwidth这两个annotations设置Pod网络带宽限制。

除此之外，使用Kubenet插件还需要注意以下几点：

·Kubenet使用了几个标准CNI插件，例如bridge、lo和host-local，其最低的版本为0.2.0。Kubenet默认会在/opt/cni/bin中搜索这些插件的二进制，用户也可以指定network-plugin-dir以支持额外的搜索路径。Kubenet还会去/etc/cni/net.d目录搜索CNI的配置文件，使用找到的第一个匹配的conf文件；

·Kubelet启动时可以添加--non-masquerade-cidr=<clusterCidr>参数，意思是向该IP段之外的IP地址发送的流量将进行一次SNAT；

·节点必须被分配到一个IP子网，通过Kubelet的--pod-cidr参数进行配置；

·--allcate-node-cidrs=true--cluster-cidr=<cidr>参数指定集群和节点的网段。

1.定制**MTU**

正确地配置MTU才能获得最佳的网络性能。网络插件通常会尝试推断出合理的MTU，但由于网络拓扑的复杂性，有时程序无法自动推算最优的MTU。例如，适合Docker网桥和IPSEC封装的MTU是不一样的。

Kubenet支持用户使用Kubelet的--network-plugin-mtu参数指定MTU。例如在AWS上，eth0　MTU通常为9001，于是用户可以指定--network-plugin-mtu=9001。如果使用IPSEC，则可以因为封装开销而减少MTU，例如--network-plugin-mtu=8873。

目前，只有Kubenet支持--network-plugin-mtu选项。

2.带宽控制

Kubenet调用Linux　tc实现了一个简单的ingress/egress的带宽控制器。当我们对Pod做了以下配置时：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  annotations:
    kubernetes.io/ingress-bandwidth: 1M
    kubernetes.io/egress-bandwidth: 1M
```

```
spec:
  containers:
  - name: test
    image: nginx:1.13
```

Kubenet就会自动为这个Pod分别将上传和下载的最高带宽限制为1Mb/s。在很长一段时间内，Pod带宽控制这个功能一直是Kubenet的专利，CNI插件一直不支持，而这也是Kubenet迟迟没有被CNI完全代替的重要原因之一。当集群不需要SDN时，例如单机部署，可能Kubenet是一个不错的选择。

3.4.2 网络生态第一步：CNI

CNI是容器网络的标准化，试图通过JSON描述一个容器网络配置。CNI的原理如图3-10所示。

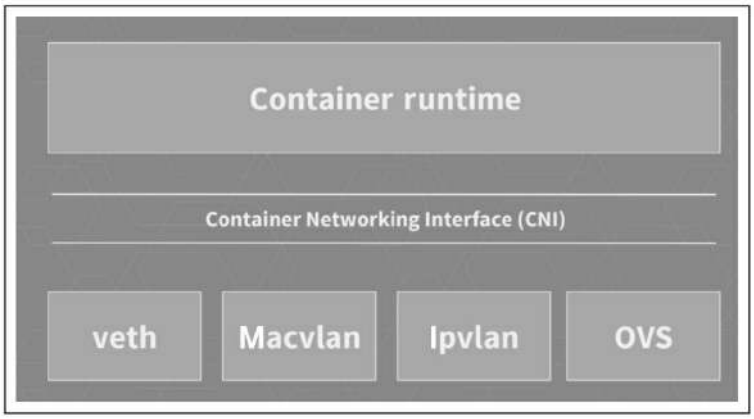


图3-10　CNI的原理

从图3-10中可以看出，CNI是Kubernetes与底层网络插件之间的一个抽象层，为Kubernetes屏蔽了底层网络实现的复杂度，同时解耦了Kubernetes的具体网络插件实现。

CNI主要有两类接口：分别是在创建容器时调用的配置网络接口：

```
AddNetwork(net *NetworkConfig, rt* RuntimeConf) (types.Result, error)
```

和删除容器时调用的清理网络接口：

```
DelNetwork(net *NetworkConfig, rt* RuntimeConf)
```

不论是配置网络接口还是清理网络接口，都有两个入参，分别是网络配置和runtime配置。网络配置很好理解，runtime配置则主要是容器运行时传入的网络namespace信息。符合CNI标准的默认/第三方网络插件如图3-11所示。

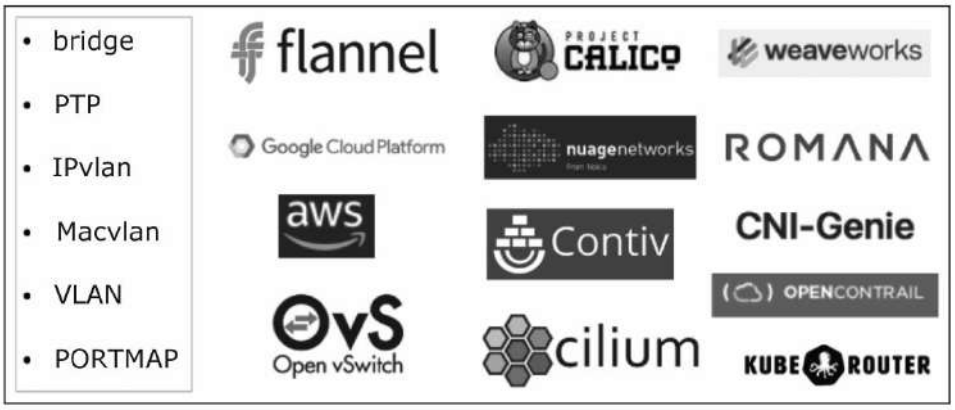


图3-11　符合CNI标准的默认/第三方网络插件

其中，左边是CNI自带插件（其实还有更多），右边是符合CNI标准的第三方插件。CNI-Genie是一个开源的多网络的容器解决方案，后面章节会进行详细介绍。

安装**CNI**

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
yum install -y kubernetes-cni
```

下面我们举几个CNI网络插件的例子。

host-local&bridge插件

```
# mkdir -p /etc/cni/net.d
# cat >/etc/cni/net.d/10-mynet.conf <<EOF
{
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
```

```
    "ipam": {
        "type": "host-local",
        "subnet": "10.10.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

上文的JSON文件是一个host-local+bridge插件组合的例子，在一个JSON文件中，我们定义了一个名为mynet的网络。它是一个bridge模型，而IP地址管理（ipam）使用的是host-local（在本地用一个文件记录已经分配的容器IP地址），且可供分配的容器网段是10.10.0.0/16。

至于在Kubernets中如何使用它们？Kubelet和CNI给出了两个默认的文件系统路径，/etc/cni/net.d用来存储CNI配置文件，/opt/cni/bin目录用来存放CNI插件的二进制文件。在我们这个例子中，至少要提前放好bridge和host-local两个插件的二进制，以及10-mynet.conf配置文件（叫什么名字随意，Kubelet只解析*.conf文件）。主流的网络插件都集成了bridge插件并提供了各自的ipam功能，因此在实际的Kubernets使用过程中，我们不需要操心上面的过程，也无须做额外配置。

在**Kubernetes**中使用**CNI**

与Kubenet类似，Kubelet要使用CNI网络驱动需要配置启动参数--network-plugin=cni。Kubelet从--cni-conf-dir（默认为/etc/cni/net.d）中读取文件，并使用该文件中的CNI配置配置每个Pod网络。如果目录中有多个CNI配置文件，则使用文件名字典序列中的第一个文件。CNI插件二进制文件所放置的目录通过Kubelet的--cni-bin-dir参数配置，默认为/opt/cni/bin。Kubernetes要求标准CNI插件的最低版本为0.2.0。

Kubernetes自己定义一套网络接口，这是面向Pod的API：

```
// Plugin is an interface to network plugins for the kubelet
type NetworkPlugin interface {
  // Init initializes the plugin.  This will be called exactly once
  // before any other methods are called.
  Init(host Host, hairpinMode componentconfig.HairpinMode, nonMasqueradeCIDR string)
error

  // Called on various events like:
```

```
  // NET_PLUGIN_EVENT_POD_CIDR_CHANGE
  Event(name string, details map[string]interface{})

  // Name returns the plugin's name. This will be used when searching
  // for a plugin by name, e.g.
  Name() string

  // Returns a set of NET_PLUGIN_CAPABILITY_*
  Capabilities() utilsets.Int

  // SetUpPod is the method called after the infra container of
  // the pod has been created but before the other containers of the
  // pod are launched.
  SetUpPod(namespace string, name string, podInfraContainerID kubecontainer.
ContainerID) error

  // TearDownPod is the method called before a pod's infra container will be deleted
  TearDownPod(namespace string, name string, podInfraContainerID kubecontainer.
ContainerID) error

  // Status is the method called to obtain the ipv4 or ipv6 addresses of the
container
  GetPodNetworkStatus(namespace string, name string, podInfraContainerID
kubecontainer.ContainerID) (*PodNetworkStatus, error)

  // NetworkStatus returns error if the network plugin is in error state
  Status() error
}
```

一个使用Kubernetes网络API的例子：

```
func (plugin *kubenetNetworkPlugin) addContainerToNetwork(config *libcni.
NetworkConfig, ifName, namespace, name string, id kubecontainer.ContainerID) (*
cnitypes.Result, error) {
  rt, err := plugin.buildCNIRuntimeConf(ifName, id)
  if err != nil {
    return nil, fmt.Errorf("Error building CNI config: %v", err)
  }
```

```
  glog.V(3).Infof("Adding %s/%s to '%s' with CNI '%s' plugin and runtime: %+v",
namespace, name, config.Network.Name, config.Network.Type, rt)
  res, err := plugin.cniConfig.AddNetwork(config, rt)
  if err != nil {
    return nil, fmt.Errorf("Error adding container to network: %v", err)
  }
  return res, nil
}
```

下面我们以Pod带宽控制为例，介绍Kubernetes使用CNI插件的流程。

CNI支持Pod的带宽控制是Kubernetes 1.11版本新开发的一个功能。当书写了以下Pod配置时：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
  annotations:
    kubernetes.io/ingress-bandwidth: 1M
    kubernetes.io/egress-bandwidth: 1M
spec:
  containers:
  - name: test
    image: nginx:1.13
```

Kubernetes会自动为这个Pod分别限制上传和下载的带宽为1Mb/s。注意，我们需要自己在/etc/cni/net.d目录下写一个配置文件，例如my-net.conf：

```
{
  "type": "bandwidth",
  "capabilities": {"bandwidth": true}
}
```

这个配置文件会告诉Kubelet去调用CNI的默认bandwidth插件，然后根据Pod annotation里带宽的ingress/egress值进行容器上行/下行带宽的限制。当然，CNI插件最后调用的还是Linux tc工具，主机上的tc配置如下所示：

```
# tc qdisc show
```

```
qdisc tbf 1: dev cali72f390115b5 root refcnt 2 rate 1Mbit burst 27917286b lat 1924.2s
qdisc ingress ffff: dev cali72f390115b5 parent ffff:fff1 ----------------
qdisc tbf 1: dev e544 root refcnt 2 rate 1Mbit burst 27917286b lat 1924.2s
```

Pod带宽控制的底层技术栈如图3-12所示。不难看出，用户通过Pod的annotations下发带宽限制数值，CNI的bandwidth插件调用Linux流量控制插件tc，在宿主机上应用tc配置。

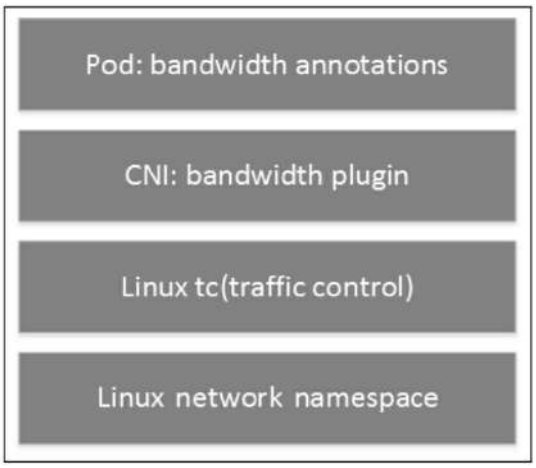


图3-12　Pod带宽控制的底层技术栈

Pod带宽控制的工作流如图3-13所示。

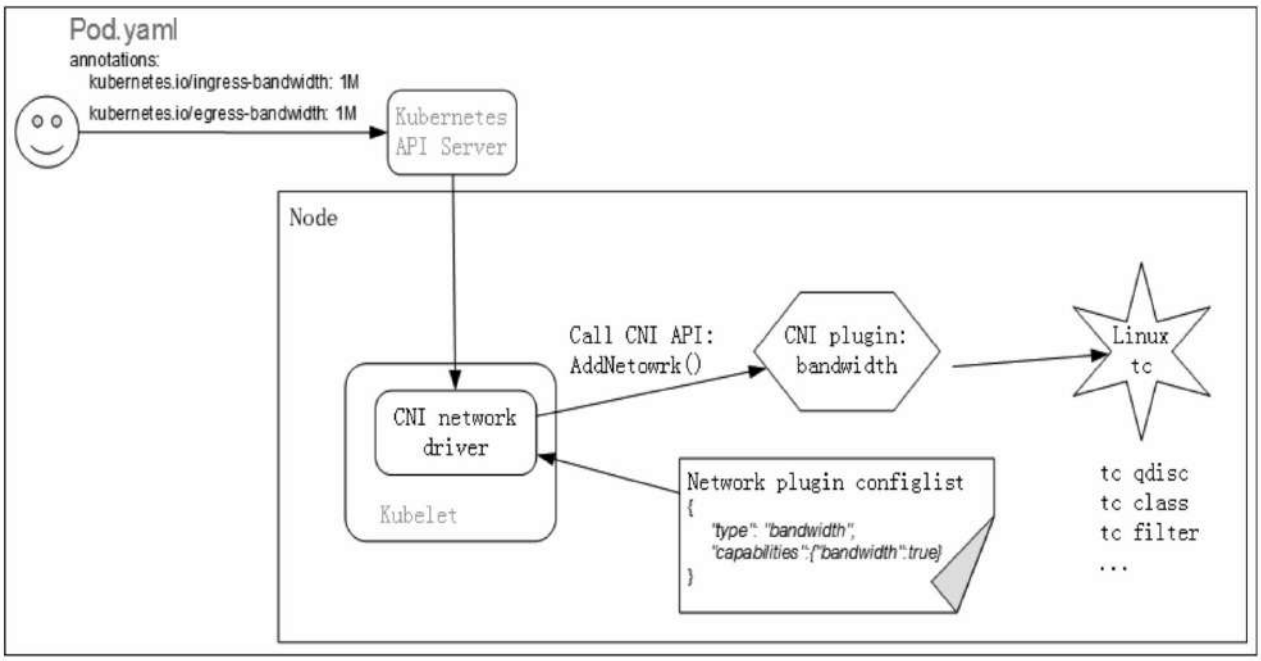


图3-13　Pod带宽控制的工作流

3.5 找到你并不容易：从集群内访问服务

Kubernetes里服务（Service）的概念即我们通常说的微服务（micro-service）。至于引入Service这个概念的初衷，可以从一个简单的例子说起。客户端访问容器应用，最简单的方式莫过于容器IP+端口，如图3-14所示。

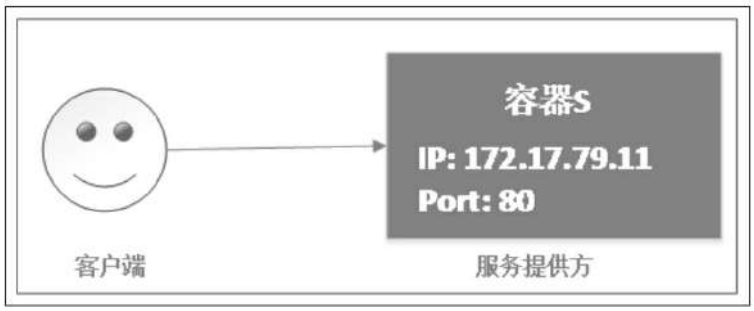


图3-14　客户端直接访问容器

但是，当有多个后端实例时：

·如何做到负载均衡？

·如何保持会话亲和性？

·容器迁移，IP发生变化如何访问？

·健康检查怎么做？

·怎么通过域名访问？

Kubernetes提供的解决方案是在客户端和后端Pod之间引入一个抽象层：Service。那么，什么是Kubernetes的Service呢？

在Kubernetes中，用户可以为任何Kubernetes资源分配成为Labels（标签）的任意键值。Kubernetes使用Labels将多个相关的Pod组合成一个逻辑单元，称为Service。Service具有稳定的IP地址（区别于容器不固定的IP地址）和端口，并会在一组匹配的后端Pod之间提供负载均衡，匹配的条件就是Service的Label Selector与Pod的Labels相匹配。

图3-15所示为两个独立服务，每个服务都由多个Pod组成。图中的每个Pod都带有app=demo标签，但这些Pod的其他标签又有所不同。服务“frontend”匹配所有带有app=demo和component=frontend的Pod，而服务“users”匹配所有带有app=demo和component=users的Pod。客户端Pod与任意服务选择器都不匹配，因此它不是任意一个服务的一部分。客户端Pod在同一集群内运行，因此它可与任意服务进行通信。

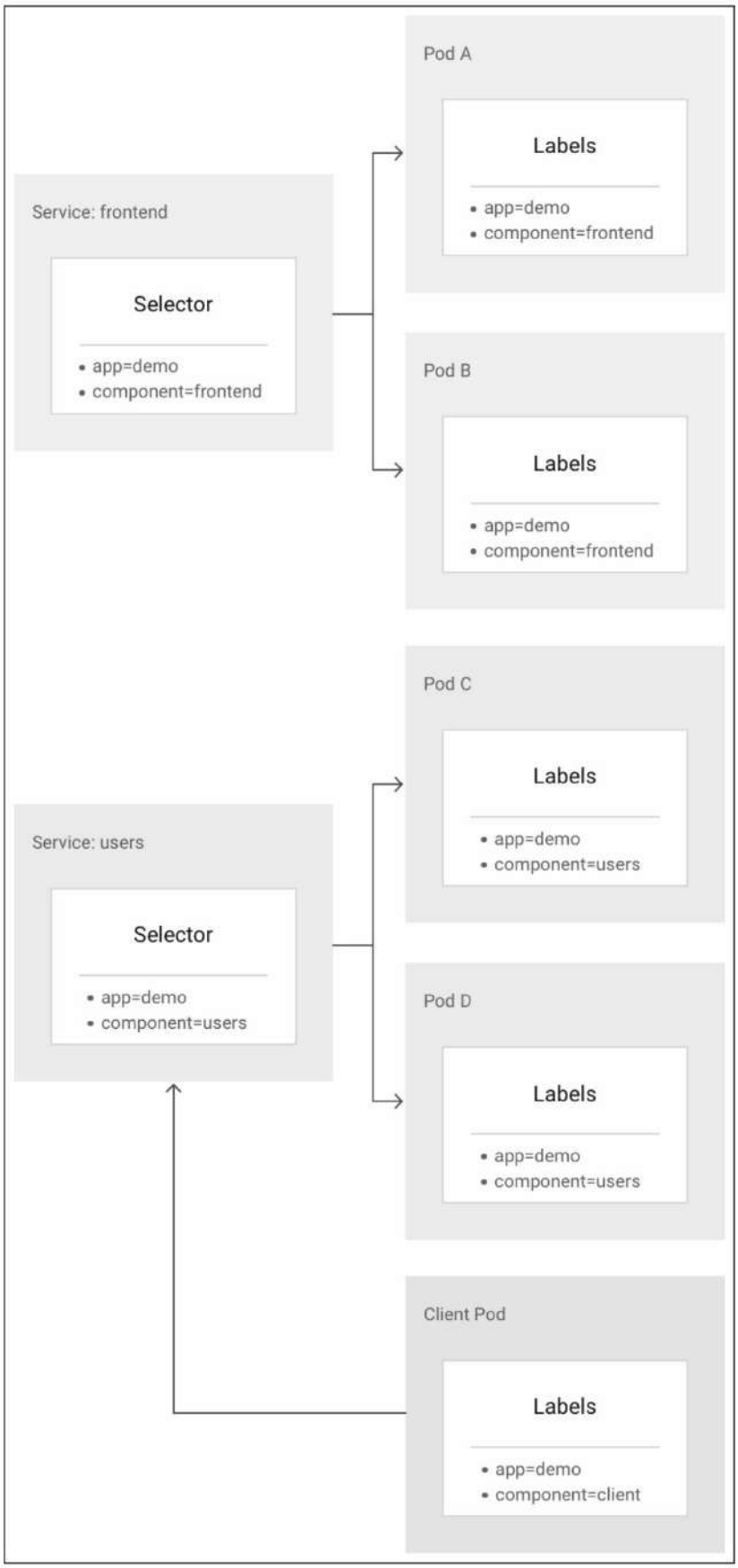


图3-15　两个独立服务

3.5.1　Kubernetes Service详解

简单说，Kubernetes的Service代表的是Kubernetes后端服务的入口，它主要包含服务的访问IP（虚IP）和端口，因此工作在L4。既然Service只存储服务入口信息，那如何关联后端Pod呢？前文已经提到Service通过Label Selector选择与之匹配的Pod。那么被Service选中的Pod，当它们运行且能对外提供服务后，Kubernetes的Endpoints Controller会生成一个新的Endpoints对象，记录Pod的IP和端口，这就解决了前文提到的后端实例健康检查问题。另外，Service的访问IP和Endpoints/Pod IP都会在Kubernetes的DNS服务器里存储域名和IP的映射关系，因此用户可以在集群内通过域名的方式访问Service和Pod。Service与Endpoints的关系如图3-16所示。

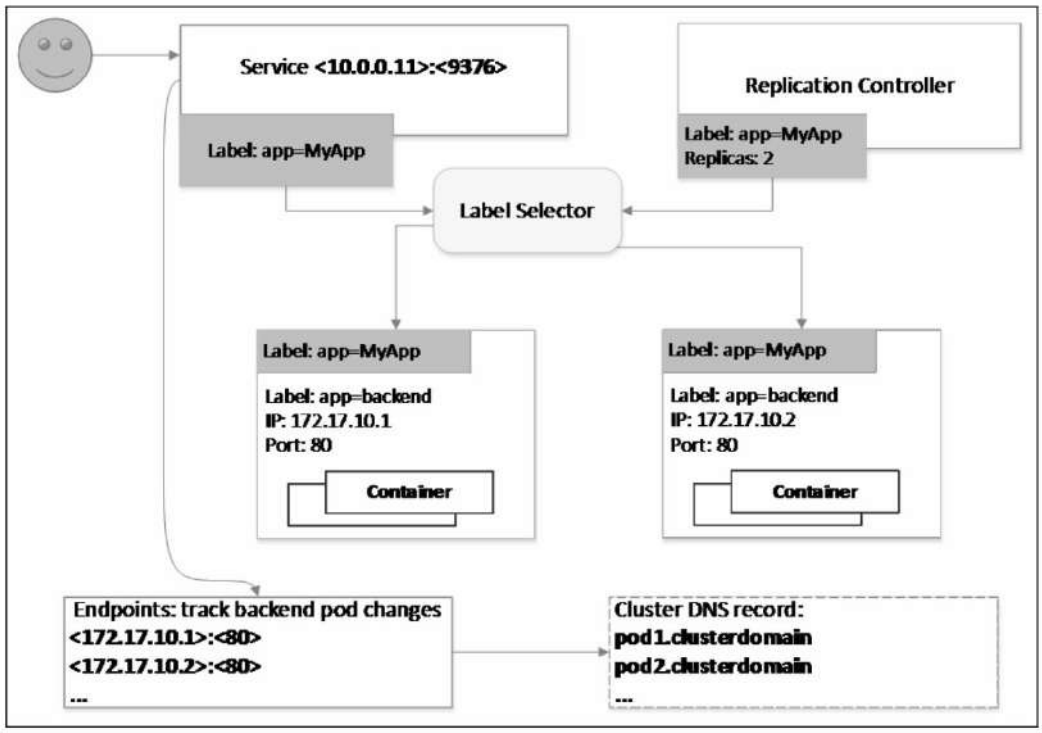


图3-16 Service与Endpoints的关系

Kubernetes会从集群的可用服务IP池中为每个新创建的服务分配一个稳定的集群内访问IP地址，称为Cluster IP。Kubernetes还会通过添加DNS条目为Cluster IP分配主机名。Cluster IP和主机名在集群内是独一无二的，并且在服务的整个生命周期内不会更改。只有将服务从集群中删除，Kubernetes才会释放Cluster IP和主机名。用户可以使用服务的Cluster IP或主机名访问正常运行的Pod。

用户不用担心服务出现单点故障问题，Kubernetes会尽可能均匀地将流量分布到在多个节点上运行的Pod，因此一个或若干个（但不是所有）节点的服务中断情况不会影响服务的整体可用性。

Kubernetes使用Kube-proxy组件管理各服务与之后端Pod的连接，该组件在每个节点上运行。Kube-proxy是一个基于出站流量的负载平衡控制器，它监控Kubernetes API Service并持续将服务IP（包括Cluster IP等）映射到运行

状况良好的Pod，落实到主机上就是iptables/IPVS等路由规则。访问服务的IP会被这些路由规则直接DNAT到Pod IP，然后走底层容器网络送到对应的Pod。

一个最简单的Kubernetes Service的定义如下所示：

```
kind: Service
apiVersion: v1
metadata:
  name: nginx-service
spec:
  clusterIP: 100.101.28.148
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
```

其中，spec.ClusterIP就是Service的（其中一个）访问IP，俗称虚IP（Virtual IP，即VIP）。如果用户不指定的话，那么Kubernetes Master会自动从一个配置范围内随机分配一个。

注意：服务分配的Cluster IP是一个虚拟IP，刚接触Kubernetes Service的人经常犯的错误是试图ping这个IP，然后发现它没有任何响应。实际上，这个虚拟IP只有和它的port一起使用才有作用，直接访问该IP或者想访问该IP的其他端口都是徒劳。

我们注意到，该Service的selector是app:nginx，即匹配那些被打上app=nginx标签的Pod。

spec.ports［］.port是Service的访问端口，而与之对应的spec.ports［］.targetPort是后端Pod的端口，Kubernetes会自动做一次映射（80->8080），具体实现机制后面会详细解释。Kubernetes Service能够支持TCP、UDP和SCTP三种协议，默认是TCP协议。

上文提到，当Service的后端Pod准备就绪后，Kubernetes会生成一个新的Endpoints对象，而且这个Endpoints对象和Service同名。一个Endpoints的定义如下所示：

```
apiVersion: v1
kind: Endpoints
metadata:
  name: nginx-service
subsets:
  \item addresses:
      - ip: 172.17.10.1
      - nodeName: 100-106-179-237.node
      targetRef:
        kind: Pod
        name: nginx-rc-c8tw2
  - addresses:
      - ip: 172.17.10.2
      - nodeName: 100-106-179-238.node
      targetRef:
        kind: Pod
        name: nginx-rc-x14tv
    ports:
      - name: http
        port: 8080
        protocol: TCP
```

用户可以通过以下命令得到这个Endpoints对象：

```
# kubectl get endpoints
```

其中，subsets［］.addresses［］.ip是后端Pod的IP，subsets［］.ports是后端Pod的端口，与Service的targetPort对应。

如果觉得通过yaml方式创建Service太麻烦，则kubectl还提供了expose子命令直接将deployment暴露成服务。

先创建一个两副本的deployment，如下所示：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: whoami
spec:
  selector:
```

```
    matchLabels:
      run: whoami
  replicas: 2
  template:
    metadata:
      labels:
        run: whoami
    spec:
      containers:
      - name: whoami
        image: cizixs/whoami:v0.5
        ports:
        - containerPort: 3000
```

然后为上面的deployment创建出来的两个Nginx副本暴露一个Service：

```
# kubectl expose deployment/whoami
service "whoami" exposed
```

这等价于使用kubectl create-f命令创建Service，对应如下的yaml文件：

```
apiVersion: v1
kind: Service
metadata:
  name: whoami
  labels:
    run: whoami
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 3000
  selector:
    run: whoami
```

查看Service的详情可以看到它的Endpoints列表，如下所示：

```
# kubectl get svc
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
whoami       10.10.10.28   <none>        8080/TCP   1d
```

```
# kubectl describe svc whoami
Name:                   whoami
Namespace:              default
Labels:                 run=whoami
Selector:               run=whoami
Type:                   ClusterIP
IP:                     10.10.10.28
Port:                   <unset> 8080/TCP
Endpoints:              172.17.32.6:3000,172.18.192.3:3000
Session Affinity:       None
No events.
```

测试发现，访问whoami服务会随机转发给后端Pod，如下所示：

```
# curl http://10.10.10.28:8080
viola from whoami-c0x6h
# curl http://10.10.10.28:8080
viola from whoami-8fpqp
```

访问服务Cluster IP:Port返回了不同后端Pod的响应。

在默认情况下，服务会随机转发到可用的后端。如果希望保持会话（同一个client永远都转发到相同的Pod），可以把service.spec.sessionAffinity设置为ClientIP，即基于客户端源IP的会话保持，而且默认会话保持时间是10800秒。这会起到什么样的效果呢？即在3小时内，同一个客户端访问同一服务的请求都会被转发给第一个Pod。例如，给上面的Service配置了sessionAffinity=ClientIP后，再发起请求的效果如下所示：

```
# curl http://10.10.10.28:8080
viola from whoami-c0x6h
# curl http://10.10.10.28:8080
viola from whoami-c0x6h
```

3.5.2 Service的三个port

先来看一个最简单的Service定义：

```
apiVersion: v1
kind: Service
```

```
metadata:
  labels:
    name: app1
  name: app1
  namespace: default
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30062
    selector:
    name: app1
```

Service的几个port的概念很容易混淆，它们分别是port、targetPort和NodePort。

port表示Service暴露的服务端口，也是客户端访问用的端口，例如Cluster IP:port是提供给集群内部客户访问Service的入口。需要注意的是，port不仅是Cluster IP上暴露的端口，还可以是external IP和Load Balancer IP。Service的port并不监听在节点IP上，即无法通过节点IP:port的方式访问Service。

NodePort是Kubernetes提供给集群外部访问Service入口的一种方式（另一种方式是Load Balancer），所以可以通过Node IP:nodePort的方式提供集群外访问Service的入口。需要注意的是，我们这里说的集群外指的是Pod网段外，例如Kubernetes节点或因特网。

targetPort很好理解，它是应用程序实际监听Pod内流量的端口，从port和NodePort上到来的数据，最终经过Kube-proxy流入后端Pod的targetPort进入容器。

在配置服务时，可以选择定义port和targetPort的值重新映射其监听端口，这也被称为Service的端口重映射。Kube-proxy通过在节点上iptables规则管理此端口的重新映射过程。

3.5.3 你的服务适合哪种发布形式

1.Cluster IP

Kubernetes Service有几种类型：Cluster IP、Load Balancer和NodePort。

其中，Cluster IP是默认类型，自动分配集群内部可以访问的虚IP——Cluster IP。我们随便创建一个Service，只要不做特别指定，都是Cluster IP类型。Cluster IP的主要作用是方便集群内Pod到Pod之间的调用。一个典型的Clustre IP类型的Service如下：

```
# kubectl describe service redis-sentinel
Name:              redis-sentinel
Namespace:         default
Labels:            name=sentinel,role=service
Selector:          redis-sentinel=true
Type:              ClusterIP
IP:                10.254.142.111
Port:              <unnamed>  26379/TCP
Endpoints:         <none>
Session Affinity:    None
No events.
```

Cluster IP主要在每个node节点使用iptables，将发向Cluster IP对应端口的数据转发到后端Pod中。针对iptables的更详细分析见后面章节。

2.Load Balancer

我们已经了解了Kubernetes如何使用Service为Pod内运行的应用提供稳定的IP地址。在默认情况下，Pod不会公开一个外部IP地址，而是由每个节点上的Kube-proxy管理所有流量。集群内的Pod之间可以自由通信，但集群外的连接无法访问服务。例如，集群外部的客户端无法通过Cluster IP访问Service。

Load Balancer（简称LB）类型的Service需要Cloud Provider的支持。Kubernetes原生支持的Cloud Provider有GCE和AWS，因此和不同云平台的网络方案耦合较大，而且只能在特定的云平台上使用，局限性也较大。除了“外用”，Load Balancer还可以“内服”，即如果要在集群内访问Load Balancer类型的Service，则Kube-proxy用iptables或ipvs实现云服务提供商Load Balancer（一般都是L7的）的部分功能：L4转发、安全组规则等。

说到安全组规则，在默认情况下，来自任何外部IP地址的流量都可以访问Load Balancer类型的服务。创建Service时，通过配置serviceSpec.loadBalancerSourceRanges字段，可以限制哪些IP地址范围可以访问集群内的服务。loadBalancerSourceRanges可以指定多个范围，并且支持随时更新。Kube-proxy会配置该节点的iptables规则，以拒绝与指定loadBalancerSourceRanges不匹配的所有流量。这样就不需要额外配置VPC的

防火墙规则了。

一个Load Balancer类型Service的定义如下所示：

```
kind: Service
apiVersion: v1
```

```
metadata:
  name: influxdb
spec:
  type: LoadBalancer
  loadBalancerSourceRanges:
  - 130.211.204.1/32
  - 130.211.204.2/32
  ports:
    - port: 8086
  selector:
    name: influxdb
```

查看这个服务的细节：

```
# kubectl get svc influxdb
NAME       CLUSTER-IP     EXTERNAL-IP     PORT(S)    AGE
influxdb   10.97.121.42   10.13.242.236   8086/TCP   39s
```

内部可以使用Cluster-IP加端口来访问该服务，在我们这个例子中就是19.97.121.42:8086。

外部可以使用EXTERNAL-IP加端口来访问该服务，这是一个云供应商提供的负载均衡器IP，在我们这个例子中就是10.13.242.236:8086。

Load Balancer类型Service的原理如图3-17所示。

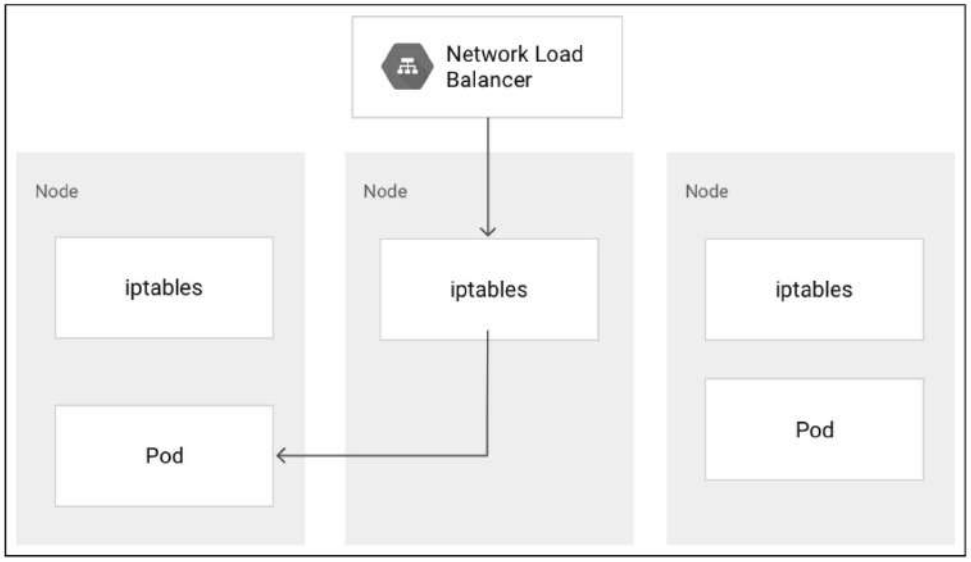


图3-17　Load Balancer类型Service的原理

3.NodePort

NodePort类似Service，被称为乞丐版的Load Balancer类型Service，这也暗示了Node-Port Service可以用于集群外部访问Service，而且成本低廉（无须一个外部Load Balancer）。

NodePort为Service在Kubernetes集群的每个节点上分配一个真实的端口，即NodePort。集群内/外部可基于集群内任何一个节点的IP:NodePort的形式访问Service。NodePort支持TCP、UDP、SCTP，默认端口范围是30000-32767，Kubernetes在创建NodePort类型Service对象时会随机选取一个。用户也可以在Service的spec.ports.nodePort中自己指定一个NodePort端口，就像指定Cluster IP那样。如果觉得默认端口范围不够用或者太大，可以修改API Server的--service-node-port-range的参数，修改默认NodePort的范围，例如--service-node-port-range=8000-9000。

一个典型的NodePort类型Service如下：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    name: ssh
    role: service
  name: ssh-service1
spec:
  ports:
    - port: 2222
      targetPort: 22
      nodePort: 30239
  type: NodePort
  selector:
    ssh-service: "true"
```

使用kubectl describe service可以看到，虽然Service的类型是NodePort，但是Kubernetes依然为其分配了一个Cluster IP，输出如下：

```
# kubectl describe service ssh-service1
Name:           ssh-service1
Namespace:      default
Labels:         name=ssh,role=service
Selector:       ssh-service=true
Type:           NodePort
```

```
IP:             10.254.132.107
Port:           <unnamed> 2222/TCP
NodePort:       <unnamed> 30239/TCP
Endpoints:      <none>
Session Affinity:    None
No events.
```

以上输出的IP字段即该Service的Cluster IP。

NodePort的实现机制是Kube-proxy会创建一个iptables规则，所有访问本地NodePort的网络包都会被直接转发至后端Port。NodePort会在主机上打开（但不监听）一个实际的端口，当NodePort类型服务创建并且被Kube-proxy感知后，可以通过以下命令验证某个端口（例如9000）是否打开：

```
# lsof -i:9000
```

某个NodePort被打开后，主机上其他进程将无法再使用该端口，除非Service被删除该端口才会被释放。

在一般情况下，不建议用户自己指定NodePort，而是应该让Kubernetes选择，否则维护的成本会很高。

NodePort是解决服务对外暴露的最简单方法，对于那些没有打通容器网络和主机网络的用户，NodePort成了他们从外部访问Service的首选。但NodePort的问题集中体现在性能和可对宿主机端口占用方面。一旦服务多起来，NodePort在每个节点上开启的端口会变得非常庞大且难以维护。

3.5.4 Kubernetes Service发现

Kubernetes Service创建好后，如何使用它，即如何进行服务发现呢？

Kubernetes提出了Service的概念，使得用户可以通过服务IP地址（不管是虚IP，还是Node IP，还是外部Load Balancer的IP）访问后端Pod提供的服务。但是在使用时有一个鸿沟（Gap），即客户端如何知道服务的访问IP？一个典型的例子是，假设有个应用要访问后端数据库（DB），后端DB已经通过Service对外暴露服务。我们只知道DB应用的名称和开放的端口，并不知道该服务的访问地址。

一个办法是使用Kubernetes提供的API或者kubectl查询Service的信息。但这是一个糟糕的做法，导致每个应用都要在启动时编写查询依赖服务的逻辑，让应用程序做运维操作。这无疑是重复劳动而且增加了应用的复杂度。这也导致应用耦合Kubernetes，使得应用程序无法单独部署和运行。

最早的时候，Kubernetes采用了Docker曾经使用过的方法——环境变量。Kubelet创建每个Pod时，会把系统当前所有服务的IP和端口信息都通过环境变量的方式注入容器。这样Pod中的应用可以通过读取环境变量获取所需服务的地址信息。

Kubelet为每个Pod注入所有Service的环境变量信息，形如：

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

不难发现，环境变量注入这种方式，对服务和环境变量的匹配关系有一定的规范，使用起来也相对简单。

但这种方式的缺点也很明显：

·容易环境变量洪泛，Docker启动参数过长会影响性能，甚至直接导致容器启动失败；

·Pod想要访问的任何Service必须在Pod自己被创建之前创建，否则这些环境变量就不会被注入。

更理想的方案是，应用能够直接使用服务的名字，不需要关心它实际的IP地址，中间的转换能够自动完成。名字和IP之间的转换即DNS，DNS的方式并没有以上两个限制。

在Kubernetes中使用域名服务，即假设Service（my-svc）在namespace（my-ns）中，暴露名为http的TCP端口，那么在Kubernetes的DNS服务器中会生成两种记录，分别是A记录：域名（my-svc.my-ns）到Cluster IP的映射和SRV记录，例如_http._tcp.my-svc.my-ns到一个http端口号的映射。我们会在Kube-dns一节做更详细的介绍。

3.5.5 特殊的无头Service

所谓的无头（headless）Service即没有selector的Service。Servcie抽象了该如何访问Kubernetes Pod，也能够抽象其他类型的backend，例如：

·希望在生产环境中使用外部的数据库集群，但在测试环境使用自己的数据库；

·希望服务指向另一个namespace中或其他集群中的服务；

·正在将工作负载转移到Kubernetes集群，以及运行在Kubernetes集群之外的backend。在任何场景中，都能够定义没有selector的Service：

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

这个Service没有selector，就不会创建相关的Endpoints对象。可以手动将Service映射到指定的Endpoints：

```
kind: Endpoints
apiVersion: v1
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 1.2.3.4
    ports:
      - port: 9376
```

注意：Endpoint　IP地址不能是loopback（127.0.0.0/8）、link-local（169.254.0.0/16）或linklocal多播（224.0.0.0/24）。

访问没有selector的Service，与有selector的Service的原理相同。请求将被路由到用户定义的Endpoint（该示例中为1.2.3.4:9376）。

ExternalName　Service是Service的特例，它没有selector，也没有定义任何的端口和Endpoint。相反，对于运行在集群外部的服务，它通过返回该外部服务的别名这种方式提供服务。

```
kind: Service
apiVersion: v1
metadata:
```

```
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

当查询主机my-service.prod.svc.CLUSTER时，集群的DNS服务将返回一个值为my.database.example.com的CNAME记录。访问这个服务的工作方式与其他的相同，唯一不同的是重定向发生在DNS层，而且不会进行代理或转发。如果后续决定要将数据库迁移到Kubernetes集群中，则可以启动对应的Pod，增加合适的Selector或Endpoint，修改Service的type。

3.5.6 怎么访问本地服务

当访问NodePort或Load Balancer类型Service的流量到底节点时，流量可能会被转发到其他节点上的Pod。这可能需要额外一跳的网络。如果要避免额外的跃点，则用户可以指定流量必须转到最初接收流量的节点上的Pod。

要指定流量必须转到同一节点上的Pod，可以将serviceSpec.externalTraffic Policy设置为Local（默认是Cluster）：

```
apiVersion: v1
kind: Service
metadata:
  name: my-lb-service
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  selector:
    app: demo
    component: users
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

将externalTrafficPolicy设置为Local时，负载平衡器仅将流量发送到具有属于服务的正常Pod所在的节点。每个节点上的Kube-proxy都健康运行，检

查服务器对外提供该节点上的Endpoint信息，以便系统确定哪些节点具有适当的Pod。

那么问题来了，为什么externalTrafficPolicy只支持NodePort和Load Balancer的Service，不支持Cluster IP呢？原因在于externalTrafficPolicy的设定是当流量到达确定的节点后，再由Kube-proxy在该节点上找Service的Endpoint。有些节点上存在Service Endpoint，有些则没有，再配合Kube-proxy的健康检查就能确定哪些节点上有符合要求的后端Pod。访问NodePort和Load Balancer都能指定节点，但Cluster IP无法指定节点，因此Service流量就永远出不了发起访问的客户端的那个节点，这也不是externalTrafficPolicy这个特性的设计初衷。

3.6 找到你并不容易：从集群外访问服务

在开始本节的讨论之前，先留一个开放性的问题给读者：如何从集群外访问Kubernetes Service？

可以使用NodePort类型的Service，但这种做法除了要求集群内Node有对外访问IP，还有一些已知的性能问题，具体请参考微信公众号“容器魔方”上的文章《记一次Docker/Kubernetes上无法解释的超时原因探寻之旅》。

那使用Load Balancer类型的Service呢？它要求在特定的云服务上运行Kubernetes，而且Service只提供L4负载均衡功能，一些高级的L7转发功能，例如基于HTTP header、cookie、URL的转发就做不了。

在Kubernets中，L7的转发功能、集群外访问Service，都是专门交给Ingress的。虽然前文提到NodePort和Load Balancer类型的Service在功能上也能起到对外暴露服务的作用，但是都存在各种限制，而Ingress没有上述两种Service限制。

Ingress可能是暴露服务的最强大方式，也是最复杂的。Kubernetes Ingress提供了负载平衡器的典型特性：HTTP路由、黏性会话、SSL终止、SSL直通、TCP和UDP负载平衡等。目前，并不是所有的Ingress Controller都实现了这些功能，需要查看具体的Ingress Controller文档。Ingress控制器有各种类型，包括Google Cloud Load Balancer、Nginx、Istio等。有些Ingress Controller可能还依赖各种插件，例如cert-manager，它可以为服务自动提供SSL证书。

除此之外，如果你想要使用同一个IP暴露多个服务，这些服务都使用相同的七层协议（如HTTP），那么Ingress可能正是你在寻觅的。

本节将重点介绍Kubernetes的Ingress概念。

3.6.1 Kubernetes Ingress

何谓Ingress？从字面意思解读，就是“入站流量”。Kubernetes的Ingress资源对象是指授权入站连接到达集群内服务的规则集合。看图3-18所示的这个例子便一目了然。

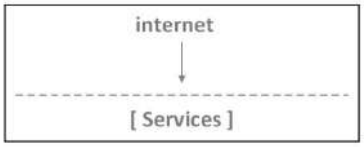


图3-18　没有Ingress时访问集群内服务

通常情况下，Service和Pod仅可在集群内部网络中通过IP地址访问。所有到达边界路由的流量或被丢弃或被转发到其他地方。图3-19所示为引入Ingress后访问集群内服务的状态。

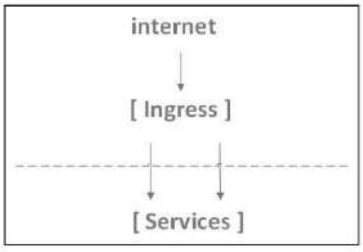


图3-19　引入Ingress后访问集群内服务的状态

Ingress的作用就是在边界路由处开个口子，放外部流量进来。因此，Ingress是建立在Service之上的L7访问入口，它支持通过URL的方式将Service暴露到k8s集群外；支持自定义Service的访问策略；提供按域名访问的虚拟主机功能；支持TLS通信。Ingress的作用如图3-20所示。

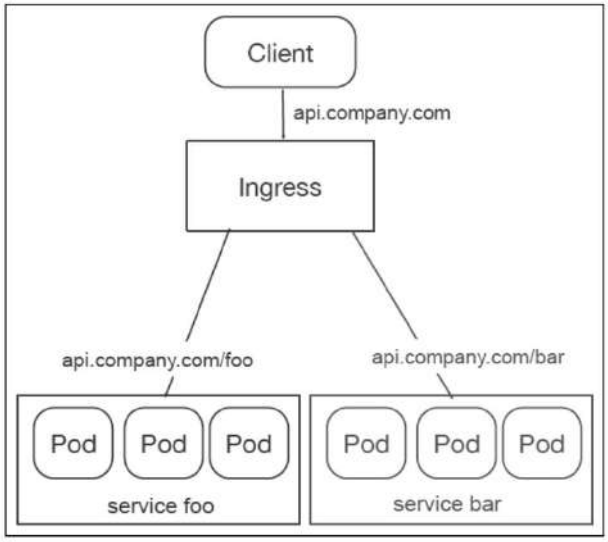


图3-20　Ingress的作用

在上面这个例子中，Ingress可以基于客户端请求的URL做流量分发，转发给不同的Service后端。

我们来看Ingress资源对象的API定义：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  tls:
  - secretName: testsecret
  backend:
    serviceName: testsvc
    servicePort: 80
```

把上面这个Ingress对象创建起来后，通过kubectl get我们可以看到：

```
# kubectl get ingress test-ingress
NAME           RULE     BACKEND      ADDRESS
test-ingress   -        testsvc:80   107.178.254.228
```

其中，ADDRESS即Ingress的访问入口地址，由Ingress Controller分配。一般由Ingress的底层实现Load Balancer的IP地址，例如Ingress、GCE LB、F5等；BACKEND是Ingress对接的后端Kubernetes Service IP+Port；RULE是自定义的访问策略，主要基于URL的转发策略，若为空，则访问ADDRESS的所有流量都转发给BACKEND。

下面给出一个Ingress的rules不为空的例子：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
```

```
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

这个例子和前一个的最明显区别是，rules定义了path分别为/foo和/bar的分发规则，分别转发给s1:80和s2:80。kubectl get的输出一目了然：

```
# kubectl get ingress test
NAME    RULE              BACKEND      ADDRESS
test    -
        foo.bar.com
        /foo              s1:80
        /bar              s2:80
```

需要注意的是，当底层Load Balancer准备就绪时，Ingress Controller把Load Balancer的IP填充到ADDRESS字段。而在我们的例子中，这个Load Balancer显然还未准备就绪。

Ingress是一个非常“极客”并需要DIY的产物，Kubernetes只负责提供一个API定义，具体的Ingress Controller需要用户自己实现！官方提供了Nginx和GCE的Ingress Controller示例供开发者参考。实现一个Ingress Controller的大致框架是：List/Watch Kubernetes的Service、Endpoints、Ingress对象，并根据这些信息刷新外部Load Balancer的规则和配置。

这还不算，如果想要通过域名访问Ingress，则需要用户自己配置域名和Ingress IP的映射关系，例如host文件、自己的DNS（不是Kube-dns）。下面章节会讲到，“高冷”的Kube-dns只负责集群内的域名解析，集群外的一概不管。

3.6.2 小结

Kubernetes的Ingress简单理解就是个规则定义，例如某个域名对应某个Service，即当某个域名的请求进来时，转发给某个Service。Ingress Controller负责实现这个规则，即Ingress Controller将其动态写入负载均衡器的配置中，从而实现服务的负载均衡。我们将在后面的章节详解Ingress Controller的实现机制。

3.7 你的名字：通过域名访问服务

众所周知，Kubernetes Master存储了所有Service的定义和更新。但是，要与后端Pod通信的客户端Pod在使用Service实现负载均衡时也需要知道这些请求会发送到何处。这些Pod可以将网络信息存储在容器环境变量中，但从长远来看这是不可行的。如果网络详细信息和一组后端Pod在将来发生更改，客户端Pod将无法与它们通信。前文提到过，Kubernetes DNS系统可以解决这个问题，下面我们将详细讨论Kubernetes的DNS。

在一个Kubernetes集群中，DNS服务是非必需的，因此无论是Kube-dns还是CoreDNS，通常以插件（add-on）的方式安装，作为运行在集群上的应用。如果涉及服务发现和域名访问，那么Kubernetes的DNS服务又是必不可少的。

3.7.1 DNS服务基本框架

具体到Kubernetes DNS服务的功能，它是用来解析Kubernetes集群内的Pod和Service域名的，而且一般情况下只供集群内的容器使用，不给外人使用。

有读者可能会好奇地问，Pod到底怎么使用Kubernetes的DNS服务呢？通常，Kubernetes的DNS应用部署好后，会对外暴露一个服务，集群内的容器通过访问该服务的Cluster IP+53端口获得域名解析服务，而这个Service的Cluster IP一般情况下都是固定的。

一般应用程序是无须感知DNS服务器的IP地址的，以Linux系统为例，容器内进程想要获得域名解析服务，只需把DNS Server写入/etc/resolv.conf文件。那么刷新/etc/resolv.conf配置这个动作是谁完成的呢？答案是Kubelet。

原来，当Kubernetes的DNS服务Cluster IP分配后，系统（一般是指安装程序）会给Kubelet配置--cluster-dns=<dns service ip>启动参数，DNS服务的IP地址将在用户容器启动时传递，并写入每个容器的/etc/resolv.conf文件。DNS服务IP即上文提到的DNS Service的Cluster IP，可以配置成--cluster-dns=10.0.0.1。

除此之外，Kubelet的--cluster_domain=<default-local-domain>参数支持配置集群域名后缀，默认是cluster.local。

打开Kubelet配置查看便一目了然：

```
# cat /etc/kubernetes/kubelet
KUBELET_ARGS="--cluster_dns=10.0.0.1 --cluster_domain=cluster.local"
```

以上流程是Kubernetes使用集群内DNS Server的基本框架，与具体的DNS Server后端无关，即使后面换成CoreDNS，基本用法也保持一致。

3.7.2 域名解析基本原理

Kubernetes DNS的命名方案也遵循可预测的模式，使各种服务的地址更容易被记住。服务不仅可以通过完全限定域名（FQDN）引用，还可以仅通过服务本身的name引用。目前，Kubernetes DNS加载项支持正向查找（A Record）、端口查找（SRV记录）、反向IP地址查找（PTR记录）及其他功能。

对于Service，Kubernetes DNS服务器会生成三类DNS记录，分别是A记录、SRV记录和CNAME记录。

1.A记录

A记录（A Record）是用于将域或子域指向某个IP地址的DNS记录的最基本类型。记录包括域名、解析它的IP地址和以秒为单位的TTL。TTL代表生存时间，是DNS记录上的一种到期日期。每个TTL都会告诉DNS服务器，它应该在其缓存中保留给定记录多长时间。

Kubernetes为“normal”和“headless”服务分配不同的A Record name。“headless”服务与“normal”服务的不同之处在于它们未分配Cluster IP且不执行负载均衡。

“normal”服务分配一个DNS A Record作为表单your-svc.your-namespace.svc.cluster.local的name（根域名可以在kubelet设置中更改）。此name解析为服务的集群IP。“headless”服务为表单your-svc.your-namespace.svc.cluster.local的name分配一个DNS A Record。与“normal”服务相反，此name解析的是，为服务选择的一组Pod IP。DNS不会自动将此设置解析为特定的IP，因此客户端应该负责集合中进行的负载均衡或循环选择。

A记录与普通Service和headless Service有区别。普通Service的A记录的映射关系是：

```
{service name}.{service namespace}.svc.{domain} -> Cluster IP
```

每个部分字段的含义是：

·service_name：Service名；

·namespace：Service所在namespace；

·domain：提供的域名后缀，是Kubelet通过--cluster-domain配置的，比如默认的cluster.local。

在Pod中可以通过域名｛service name｝.｛service namespace｝.svc.｛domain｝访问任何服务，也可以使用缩写｛service name｝.｛service namespace｝直接访问。如果Pod和Service在同一个namespace中，那么甚至可以直接使用｛service name｝访问。

headless Service的A记录的映射关系是：

```
{service name}.{service namespace}.svc.{domain} -> 后端Pod IP列表
```

含义跟前面的一致，本节不再赘述。

一旦启用了DNS，Pod将被分配一个DNS A记录，格式如下所示：

```
{pod-ip}.{pod namespace}.pod.{domain} -> Pod IP
```

其中pod-ip为Pod的IP地址用-符号隔开，例如Pod IP是1.2.3.4，上面的｛pod-ip｝即1-2-3-4，因此对应的域名就是1-2-3-4.default.pod.cluster.local。Pod的A记录其实没什么用，都知道Pod IP了，还用查DNS吗？所以这也是一个即将废弃的特性。

如果在Pod Spec指定hostname和subdomain，那么Kubernetes DNS会额外生成Pod的A记录：

```
{hostname}.{subdomain}.{pod namespace}.pod.cluster.local -> Pod IP
```

同样，后面那一串子域名pod.cluster.local是Kubelet配置的伪域名。

2.SRV记录

SRV记录是通过描述某些服务协议和地址促进服务发现的。SRV记录通常定义一个符号名称和作为域名一部分的传输协议（如TCP），并定义给定服务的优先级、权重、端口和目标。详细内容请参阅下面的示例：

```
_sip._tcp.example.com.   3600 IN    SRV 10      70     5060 srvrecord.example.com
_sip._tcp.example.com.   3600 IN    SRV 10      20     5060 srvrecord2.ex
```

在上面的示例中，_sip是服务的符号名称，_tcp是服务的使用传输协议。记录内容代表：两个记录都定义了10的优先级。另外，第一个记录的权重为70，第二个记录的权重为20。优先级和权重通常用于建议指定使用某些服务器。记录中的最后两个值定义了要连接的端口和主机名，以便与服务通信。

Kubernetes DNS的SRV记录是按照一个约定俗成的规定实现了对服务端口的查询：

```
_{port name}._{port protocol}.{service name}.{service namespace}.svc.cluster.local ->
 Service Port
```

SRV记录是为“normal”或“headless”服务的部分指定端口创建的。SRV记录采用_my-port-name._my-port-protocol.my-svc.my-namespace.svc.cluster.local的形式。对于常规服务，它被解析的端口号和域名是my-svc.my-namespace.svc.cluster.local。在“headless”服务的情况下，此name解析为多个answer，每个answer都支持服务。每个answer都包含auto-generated-name.my-svc.my-namespace.svc.cluster.local表单的Pod端口号和域名。

3.CNAME记录

CNAME记录用于将域或子域指向另一个主机名。为此，CNAME使用现有的A记录作为其值。相反，A记录会解析为指定的IP地址。此外，在Kubernetes中，CNAME记录可用于联合服务的跨集群服务发现。在整个场景中会有一个跨多个Kubernetes集群的公共服务。所有Pod都可以发现这项服务（无论这些Pod在哪个集群上）。这是一种跨集群服务发现方法。

3.7.3 DNS使用

Pod的默认主机名由Pod的metadata.name值定义。用户可以通过在可选的hostname字段中指定一个新值更改默认主机名。用户还可以在subdomain字段中自定义子域名。例如，在命名空间my-namespace中，将hostname设置为custom-host，将subdomain设置为custom-subdomain的Pod将具有完全限定的域名（FQDN）custom-host.customsubdomain.my-namespace.svc.cluster.local。

下面我们运行一个带nslookup命令的busybox容器，说明Kubernetes域名解析的使用。

大部分busybox都不带nslookup命令，因此这里给出一个带nslookup命令的busybox Pod示例yaml文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: busybox:1.28
```

```
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

在default namespace下把这个Pod创建好，就可以通过kubectl exec进入busybox容器进行以下试验了。

我们先来查询Kubernetes默认的API Server服务Kubernetes的IP地址：

```
# kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes.default
Address 1: 10.0.0.1
```

如上所示，Kube-dns的IP地址是10.0.0.1，而API Server的Cluster IP地址是10.0.0.10。

我们在default namespace下又创建了一个名为whoami的服务，并做以下域名解析动作：

```
/ # kubectl exec -ti busybox -- nslookup whoami
Server:    10.10.10.10
Address 1: 10.10.10.10

Name:      whoami
Address 1: 10.10.10.175

/ # kubectl exec -ti busybox -- nslookup whoami.default.svc
Server:    10.10.10.10
Address 1: 10.10.10.10

Name:      whoami.default.svc
Address 1: 10.10.10.175

/ # kubectl exec -ti busybox -- nslookup whoami.default.svc.transwarp.local
Server:    10.10.10.10
Address 1: 10.10.10.10
```

```
Name:      whoami.default.svc.transwarp.local
Address 1: 10.10.10.175
```

如上所示，发起域名解析请求的busybox Pod和whoami服务均在default namespace下，因此以下所有域名都能被正确解析并且返回相同的结果。

```
whoami
whoami.default.svc
whoami.default.svc.cluster.local
```

查看busybox Pod的DNS配置文件，可以看到如下DNS Server的地址及搜索的domain列表：

```
/ # cat /etc/resolv.conf
search default.pod.cluster.local default.svc.cluster.local svc.cluster.local cluster.
local
nameserver 10.0.0.1
options ndots:5
options ndots:5
```

其中，DNS Server的IP地址是10.0.0.1。options ndots:5的含义是当查询的域名字符串内的点字符数量超过ndots（5）值时，则认为是完整域名，直接解析，否则Linux系统会自动尝试用default.pod.cluster.local、default.svc.cluster.local或svc.cluster.local补齐域名后缀。例如，查询whoami会自动补齐成whoami.default.pod.cluster.local、whoami.default.svc.cluster.local和whoami.svc.cluster.local，查询过程中，任意一个记录匹配便返回，显然能返回DNS记录的匹配是whoami+default.svc.cluster.local。而查询whoami.default能返回DNS记录的匹配是whoami.default+svc.cluster.local。

最后，运行DNS Pod可能需要特权，即配置Kubelet的参数：--allow-privileged=true。

1.Kubernetes域名解析策略

Kubernetes域名解析策略对应到Pod配置中的dnsPolicy，有4种可选策略，分别是None、ClusterFirstWithHostNet、ClusterFirst和Default，其中：

·None：从Kubernetes 1.9版本起引入的一个新选项值。它允许Pod忽略Kubernetes环境中的DNS设置。应使用dnsConfigPod规范中的字段提供所有DNS设置；

·ClusterFirstWithHostNet：对于使用hostNetwork运行的Pod，用户应该明确设置其DNS策略为ClusterFirstWithHostNet；

·ClusterFirst：任何与配置的群集域后缀（例如cluster.local）不匹配的DNS查询（例如“www.kubernetes.io”）将转发到从宿主机上继承的上游域名服务器。集群管理员可以根据需要配置上游DNS服务器；

·Default：Pod从宿主机上继承名称解析配置。

None

一个配置了None类型的dnsPolicy的Pod如下：

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.2.3.4
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

以上Pod被创建后，test容器的/etc/resolv.conf内容如下所示：

```
nameserver 1.2.3.4
search ns1.svc.cluster.local my.dns.search.suffix
options ndots:2 edns0
```

ClusterFirstWithHostNet

使用ClusterFirstWithHostNet策略的一个Pod配置文件如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

如上所示，当Pod使用主机网络（hostNetwork:true）时，DNS策略需要设置成ClusterFirstWithHostNet。对于那些使用主机网路的Pod，它们是可以直接访问宿主机的/etc/resolv.conf文件的。因此，如果不加上dnsPolicy:ClusterFirstWithHostNet，则Pod将默认使用宿主机的DNS配置，这样会导致集群内容器无法通过域名访问Kubernetes的服务（除非在宿主机的/etc/resolv.conf文件配置了Kubernetes的DNS服务器）。

ClusterFirst

使用ClusterFirst策略的一个Pod配置文件如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
```

```
  dnsPolicy: ClusterFirst
```

ClusterFirst策略就是优先使用Kubernetes的DNS服务解析，失败后再使用外部级联的DNS服务解析，工作流程如图3-21所示。

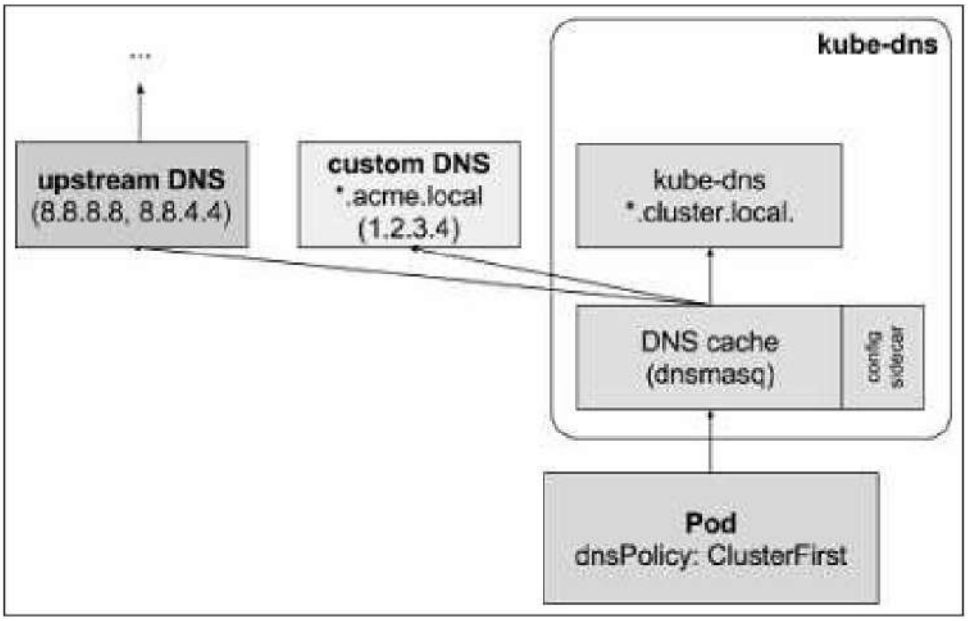


图3-21　Kubernetes DNS级联的工作流程

3.7.4　调试DNS

如果nslookup命令由于某种原因失败，则有几种调试和排除故障的方案。但是，应该如何得知DNS查找失败呢？若DNS失败，通常会得到如下响应：

```
# kubectl exec -ti busybox -- nslookup kubernetes.default
Server:    10.0.0.10
Address 1: 10.0.0.10
nslookup: can't resolve 'kubernetes.default'
```

如果出现此错误，则需要做的第一件事是检查DNS配置是否正确。

查看容器中的resolv.conf文件：

```
# kubectl exec test-deployment-84dc998fc5-772gj cat /etc/resolv.conf
```

验证是否正确设置了搜索路径和名称服务器，如下例所示：

```
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
```

```
options ndots:5
```

如果/etc/resolve.conf的所有条目都是正确的，则需要检查kube-dns/coredns插件是否已启用，或者检查kubedns/coredns Pod是否正在运行。

如果Pod正在运行，则全局DNS服务可能存在问题。

检查：

```
# kubectl get svc --namespace=kube-system
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
 AGE
kube-dns                ClusterIP   10.96.0.10       <none>        53/UDP,53
```

可能还需要检查后端DNS Endpoint是否准备好：

```
# kubectl get ep kube-dns --namespace=kube-system
NAME       ENDPOINTS                     AGE
kube-dns   172.17.0.5:53,172.17.0.5:53   133d
```

3.8 Kubernetes网络策略：为你的应用保驾护航

在默认情况下，Kubernetes底层网络是“全连通”的，即在同一集群内运行的所有Pod都可以自由通信。但是也支持用户根据实际需求以不同方式限制集群内Pod的连接。

例如，我们需要实现图3-22中的需求，即只允许访问default namespace的Label是app=web的Pod，default namespace的其他Pod都不允许外部访问。这个隔离需求在多租户的场景下十分普遍。Kubernetes的解决方案是Network Policy，即网络策略。

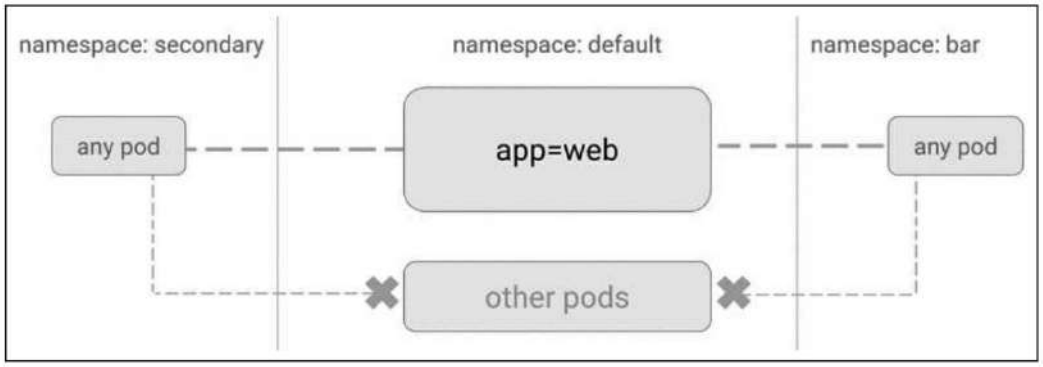


图3-22 一个需要网络策略的典型场景

网络策略就是基于Pod源IP（所以Kubernetes网络不能随随便便做SNAT）的访问控制列表，限制的是Pod之间的访问。通过定义网络策略，用户可以根据标签、IP范围和端口号的任意组合限制Pod的入站/出站流量。网络策略作为Pod网络隔离的一层抽象，用白名单实现了一个访问控制列表（ACL），从Label Selector、namespace selector、端口、CIDR这4个维度限制Pod的流量进出。

值得一提的是，Kubernetes的网络策略采用了比较严格的单向流控制，即假如允许服务A访问服务B，反过来服务B并不一定能访问服务A。这与Docker内置的Network命令实现的隔离有很大不同。

3.8.1 网络策略应用举例

让我们先来见识几个默认的网络策略，如图3-23所示。

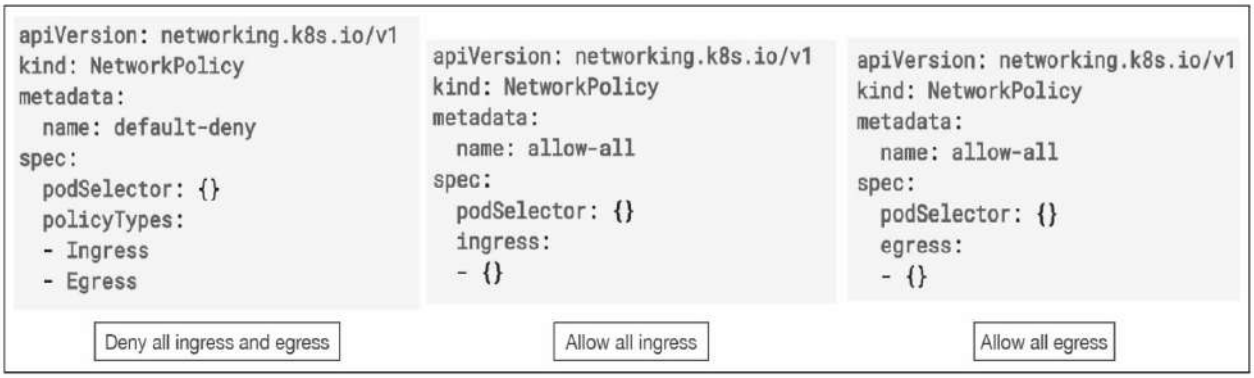


图3-23　几个Kubernetes默认的网络策略

注：｛｝代表允许所有，［］代表拒绝所有。

如果拒绝所有流量进入，例如图3-24所示的场景。

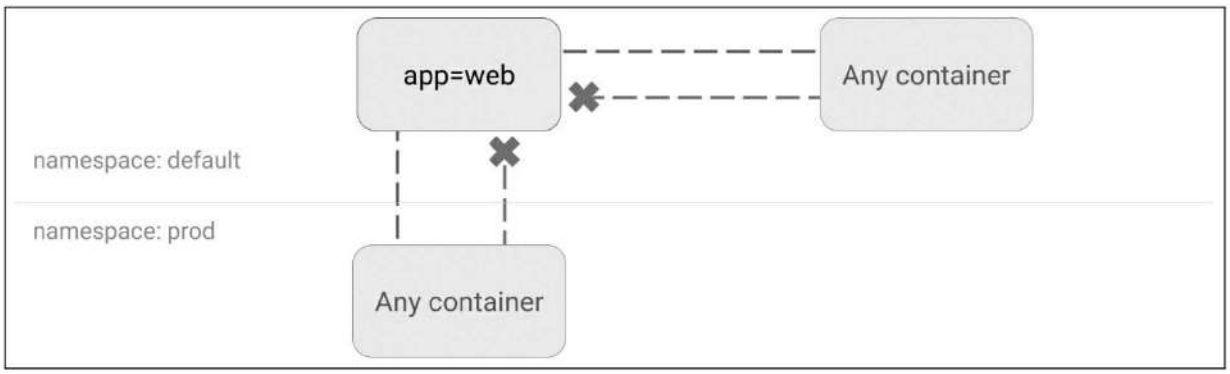


图3-24　拒绝所有流量进入的网络策略场景

那么Network Policy对象应该定义成：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

如果限制部分流量进入，例如图3-25所示的场景。

图3-25　限制部分流量进入的网络策略场景

那么，Network Policy对象应该定义成：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: bookstore
```

如果只允许特定namespace的Pod流量进入，例如图3-26所示的场景。

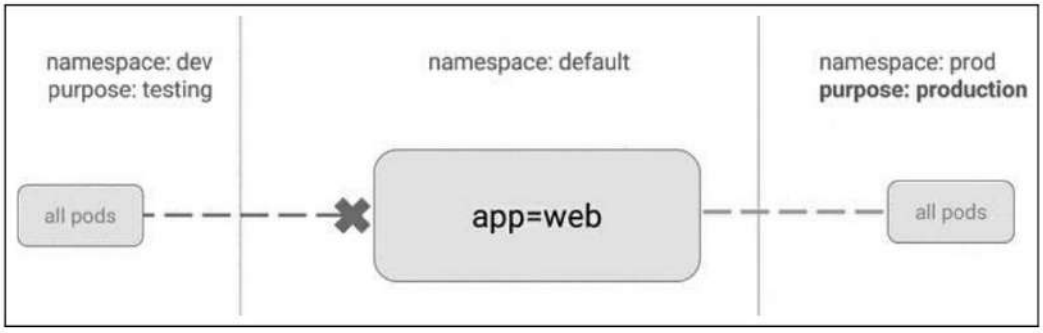


图3-26　只允许特定namespace的Pod流量进入的网络策略场景

那么Network Policy对象应该定义成：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-prod
spec:
  podSelector:
    matchLabels:
      app: web
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production
```

如果限制流量从指定端口进入，例如图3-27所示的场景。

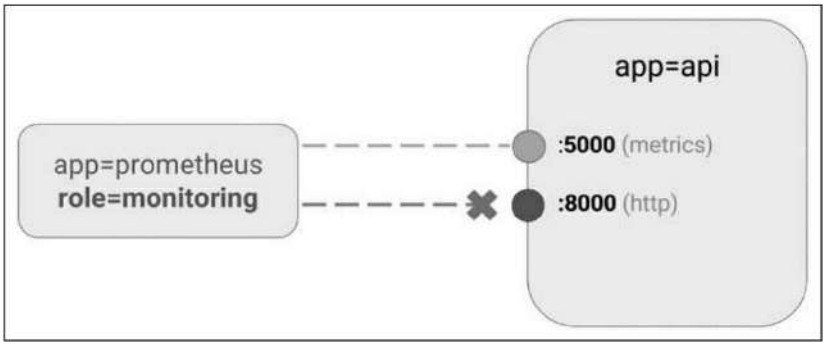


图3-27　限制流量从指定端口进入的网络策略场景

那么，Network Policy对象应该定义成：

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
```

```
metadata:
  name: api-allow-5000
spec:
  podSelector:
    matchLabels:
      app: apiserver
  ingress:
  - ports:
    - port: 5000
    from:
    - podSelector:
        matchLabels:
          role: monitoring
```

再举一个同时限定入站（Ingress）和出站（Egress）的例子：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
```

```
      matchLabels:
        role: frontend
  ports:
  - protocol: TCP
    port: 6379
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

该例子的效果如下：default namespace下Labels包含role=db的Pod，都会被隔绝，它们只能建立满足Network Policy的Ingress和Egress描述的连接。即：

（1）所有属于172.17.0.0/16网段的IP，除了172.17.1.0/24中的IP，其他的都可以与上述Pod的6379端口建立TCP连接。

（2）所有包含project=myproject Labels的namespace中的Pod可以与上述Pod的6379端口建立TCP连接。

（3）所有default namespace下的包含role=frontend Labels的Pod可以与上述Pod的6379端口建立TCP连接。

（4）允许上述Pod访问网段为10.0.0.0/24的目的IP的5978端口。

3.8.2 小结

最后，让我们梳理Kubernetes网络策略的知识点。

·Egress表示出站流量，即Pod作为客户端访问外部服务，Pod地址作为源地址。策略可以定义目的地址和目的端口，可以根据ports和to定义规则。ports字段用来指定目的端口和协议。to（目的地址）分为IP地址段、Pod selector和Kubernetes namespace selector；

·Ingress表示入站流量，Pod地址和服务作为服务端（目的地址），提供外部访问。与Egress类似，策略可以定义源地址和端口，可以根据ports和from定义规则。ports字段同样用来指定目的端口和协议。from（源地址）分为IP地址段、Pod selector和Kubernetes namespace selector；

·podSelector用于指定网络策略在哪些Pod上生效，用户可以配置单个Pod或者一组Pod。可以定义单方向。空podSelector选择命名空间中的Pod。podSelector定义Network Policy的限制范围，就是规则应用到哪个Pod上。如上所示，podSelector:｛｝留空就是定义对Kubernetes当前namespace下的所有Pod生效。没有定义白名单的话，默认就是Deny ALL（拒绝所有）；

·policyTypes：指定策略类型，包括Ingress和Egress，不指定默认就是Ingress。默认情况下，一个policyTypes的值一定会包含Ingress。

说到匹配规则，namespaceSelector和podSelector是彼此独立的两个单独的匹配规则。你可以使用ipBlock字段允许来自或者到达特定IP地址段范围的Ingress或Egress流量。

最后，介绍Kubernetes网络策略与一些安全组（例如Neutron的安全组）的区别。一般来说，安全组的规则记录在上层网关，而不是每一个节点上。而Kubernetes网络是应用到每个节点上的，实现Network Policy的agent需要在每个节点上都List/Watch Kubernetes的Network Policy、namespace、Pod等资源对象，因此在实现上可能会有较大的性能影响。Network Policy agent一般通过Kubernetes DaemonSet实现每个节点上都有一个部署。

3.9 前方高能：Kubernetes网络故障定位指南

网络可以说是Kubernetes部署和使用过程中最容易出问题的，通过前面的理论讲解我们已经掌握了Kubernetes网络的基本知识，本节将重点介绍Kubernetes网络故障定位的实战经验。

3.9.1 IP转发和桥接

Kubernetes网络利用Linux内核Netfilter模块设置低级别的集群IP负载均衡，除了iptables和IPVS（前面已经详细解析过，这里不再赘述），还需要用到两个关键的模块：IP转发（IP forward）和桥接。

1.IP转发

IP转发是一种内核态设置，允许将一个接口的流量转发到另一个接口，该配置是Linux内核将流量从容器路由到外部所必需的。有时，该项设置可能会被安全团队运行的定期安全扫描重置，或者没有配置为重启后生效。在这种情况下，就会出现网络访问失败的情况，例如，访问Pod服务连接超时：

```
* connect to 10.100.225.223 port 5000 failed: Connection timed out
* Failed to connect to 10.100.225.223 port 5000: Connection timed out
* Closing connection 0
curl: (7) Failed to connect to 10.100.225.223 port 5000: Connection timed out
```

Tcpdump可以显示发送了大量重复的SYN数据包，但没有收到ACK。

那么，该如何诊断呢？请看下面的诊断方法：

```
# 检查ipv4 forwarding是否开启
sysctl net.ipv4.ip_forward
# 0意味着未开启
net.ipv4.ip_forward = 0
```

修复也很简单，如下所示：

```
# this will turn things back on a live server
sysctl -w net.ipv4.ip_forward=1
# on Centos this will make the setting apply after reboot
echo net.ipv4.ip_forward=1 >> /etc/sysconf.d/10-ipv4-forwarding-on.conf

# 验证并生效
sysctl -p
```

2.桥接

Kubernetes通过bridge-netfilter配置使iptables规则应用在Linux网桥上。该配置对Linux内核进行宿主机和容器之间数据包的地址转换是必需的。否则，Pod进行外部服务网络请求时会出现目标主机不可达或者连接拒绝等错误（host unreachable或connection refused）。

那么，如何诊断呢？请看下面的命令：

```
# 检查bridge netfilter是否开启
sysctl net.bridge.bridge-nf-call-iptables

# 0表示未开启
net.bridge.bridge-nf-call-iptables = 0
```

如何修复呢？请看下面的命令：

```
# Note some distributions may have this compiled with kernel,
# check with cat /lib/modules/$(uname -r)/modules.builtin | grep netfilter
modprobe br_netfilter

# 开启这个iptables设置
sysctl -w net.bridge.bridge-nf-call-iptables=1
echo net.bridge.bridge-nf-call-iptables=1 >> /etc/sysconf.d/10-bridge-nf-call-
iptables.conf
sysctl -p
```

3.9.2 Pod CIDR冲突

Kubernetes有时会为容器和容器之间的通信建立一层特殊的overlay网络（取决于你使用什么样的网络插件）。使用隔离的Pod网络容器可以获得唯

一的IP并且可以避免集群上的端口冲突，而当Pod子网和主机网络出现冲突时就会出现问题。Pod和Pod之间通信会因为路由问题被中断：

```
# curl http://172.28.128.132:5000
curl: (7) Failed to connect to 172.28.128.132 port 5000: No route to host
```

那么，该如何诊断呢？首先，查看Pod分配的IP地址：

```
$ kubectl get pods -o wide
NAME                        READY     STATUS    RESTARTS   AGE       IP
NODE
netbox-2123814941-f7qfr     1/1       Running   4          21h       172.28.27.2
172.28.128.103
netbox-2123814941-ncp3q     1/1       Running   4          21h       172.28.21.3
172.28.128.102
testbox-2460950909-5wdr4    1/1       Running   3          21h       172.28.128.132
172.28.128.101
```

然后，将主机IP范围与API Server中指定的子网进行比较，如果出现了同网段的IP，则很大概率会出现冲突。

解决方法就是仔细检查你的网络设置，确保你正在使用的网络、VLAN或VPC之间不会有重叠。如果有冲突，则可以在CNI插件或Kubelet的pod-cidr参数中指定IP地址范围，避免冲突。

3.9.3 hairpin

hairpin的含义用一句话表述就是“自己访问自己”。例如，Pod有时无法通过Service IP访问自己，这就有可能是hairpin的配置问题了。通常，当Kube-proxy以iptables或IPVS模式运行，并且Pod与桥接网络连接时，就会发生这种情况。Kubelet的启动参数提供了一个--hairpin-mode的标志，支持的值有hairpin-veth和promiscuous-bridge。检查Kubelet的日志也能看到以下日志行，例如：

```
I0629 00:51:43.648698    3252 kubelet.go:380] Hairpin mode set to "promiscuous-bridge
"
```

用户需要检查Kubelet的--hairpin-mode是否被设置为一个合法的值。

--hairpin-mode被Kubelet设置成hairpin-veth并且生效后，底层其实是在修改宿主机操作系统/sys/devices/virtual/net目录下设备文件hairpin_mode的值，可以通过以下命令确认是否修改成功：

```
# for intf in /sys/devices/virtual/net/cbr0/brif/; do cat $intf/hairpin_mode; done
1
1
1
1
```

如果没有修改成功，则请确保Kubelet拥有宿主机/sys的读写权限。

如果有效的--hairpin-mode被设置成promiscuous-bridge，则请确保Kubelet拥有操作宿主机Linux网桥的权限，具体说就是要把网桥配置成混杂模式，例如：

```
# ifconfig cbr0 |grep PROMISC
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460  Metric:1
```

3.9.4 查看Pod IP地址

严格意义上讲，查看Pod IP地址只能说是一个需求，而非“问题”，是定位Kubernetes网络问题的基本手段。因此，我们将花费一点篇幅介绍如何在Kubernetes中查看Pod的IP地址。通俗地说，查看Kubernetes Pod IP有两个办法，分别是从外面看和进到里面看。从外面看又分为Kubernetes API和docker命令。进到里面看就是进到容器内使用ip或ifconfig等命令查看。

1.从外面看Pod IP地址之Kubernetes API

Kubernetes是知道集群中所有Pod的IP信息的，也提供相应的查询接口，因此kubectl get pod或者kubectl describe pod就能显示Pod的详细信息，自然也携带Pod的IP地址（在Pod的status.podIP字段），读者可以自行尝试，这里不再赘述。

如果运行在容器内的进程希望获取该容器的IP（同样是Pod的IP），环境变量是一个不错的选择，尤其是Kubernetes提供了相关的downward API。当在Pod的配置文件书写如下env字段时：

```
env:
- name: MY_POD_IP
   valueFrom:
     fieldRef:
       fieldPath: status.podIP
```

容器启动后MY_POD_IP这个环境变量的值来自Pod的status.podIP，容器内的应用只要读取该环境变量即可获取Pod的真实IP地址。

2.从外面看**Pod IP**地址之**docker**命令

假设容器的ID是abc，一般情况下我们可以通过以下命令查询容器的IP地址：

```
# docker inspect --format '{{ .NetworkSettings.IPAddress }}' abc
```

然而对于Pod中的容器，这一招并不管用，你会发现输出是一个空字符串。究其原因，既不是Kubernetes的Bug也不是用户配置错误，而是Kubernetes使用的CNI与Docker奉行的CNM标准割裂导致的。Kubernetes曾旗帜鲜明地宣布永不使用Libnetwork（具体原因见5.2节），而Libnetwork正是CNM的标准实现，docker inspect命令查询容器IP就是调用的Libnetwork。

那么是不是就意味着不能使用docker或者简单的ip命令查看由Kubernetes管理的容器IP了呢？非也，具体请看下面的方法。

3.进到容器里面看**Pod IP**地址

进到容器的docker命令有docker exec或docker attach，进到容器后再执行ip addr或ifconfig这类常规命令。同一个Pod内所有容器共享网络namespace，因此随便找一个有ip或者ifconfig命令的容器就能很容易地查询到Pod IP地址。如果Pod内所有容器都不自带这些命令呢？从1.1节的内容中我们知道，在我们这个场景下，进入容器网络namespace的作用等价于进入容器，而且还能使用宿主机的ip或者ifconfig命令。

假设Pod的pause容器ID是abc，首先获得该容器在宿主机上映射的PID，如下所示：

```
# docker inspect abc --format '{{ .State.Pid }}'
32466
```

如上所示，PID是32466。

然后调用nsenter命令（没有安装的需要自行安装），进入pause容器的网络namespace，如下所示：

```
# nsenter --target 32466 --net
# ip addr
```

正常情况下，上面的命令会输出pause容器的IP，也就是Pod的IP。

3.9.5 故障排查工具

下面是一些我们在排查上述问题时使用的非常有用的工具。

1.tcpdump

tcpdump是一个用来捕获网络流量的“利器”，不论是传统主机网络还是容器网络架构，它可以帮助我们定位一些常见的网络问题，下面是一个使用tcpdump进行流量捕获的简单例子。

```
$ tcpdump -i any host 172.28.21.3
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
20:15:59.903566 IP 172.28.128.132.60358 > 172.28.21.3.5000: Flags [S], seq
3042274422, win 28200, options [mss 1410,sackOK,TS val 10056152 ecr 0,nop,wscale 7],
length 0
20:15:59.903566 IP 172.28.128.132.60358 > 172.28.21.3.5000: Flags [S], seq
3042274422, win 28200, options [mss 1410,sackOK,TS val 10056152 ecr 0,nop,wscale 7],
length 0
20:15:59.905481 ARP, Request who-has 172.28.21.3 tell 10.244.27.0, length 28
20:16:00.907463 ARP, Request who-has 172.28.21.3 tell 10.244.27.0, length 28
20:16:01.909440 ARP, Request who-has 172.28.21.3 tell 10.244.27.0, length 28
20:16:02.911774 IP 172.28.128.132.60358 > 172.28.21.3.5000: Flags [S], seq
3042274422, win 28200, options [mss 1410,sackOK,TS val 10059160 ecr 0,nop,wscale 7],
length 0
```

```
20:16:02.911774 IP 172.28.128.132.60358 > 172.28.21.3.5000: Flags [S], seq
3042274422, win 28200, options [mss 1410,sackOK,TS val 10059160 ecr 0,nop,wscale 7],
length 0
```

2.容器镜像内置常用网络工具

在一个镜像中内置一些网络工具包，对我们的排查工作会非常有帮助，比如在下面的简单服务中，我们添加一些常用的网络工具包：iproute2 net-tools ethtool，Dockerfile如下所示：

```
FROM library/python:3.3
RUN apt-get update && apt-get -y install iproute2 net-tools ethtool nano
CMD ["/usr/bin/python", "-m", "SimpleHTTPServer", "5000"]
```

我们将上面Docker构建出来的镜像名字称为netbox，下面是一个使用netbox镜像部署Deployment的manifest文件：

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    run: netbox
    name: netbox
spec:
  replicas: 2
  selector:
    matchLabels:
      run: netbox
  template:
    metadata:
      labels:
        run: netbox
    spec:
      nodeSelector:
        type: other
      containers:
      - image: quay.io/gravitational/netbox:latest
        imagePullPolicy: Always
        name: netbox
```

```
      securityContext:
        runAsUser: 0
      terminationGracePeriodSeconds: 30
```

另外，就算只用busybox镜像，其实也有很多“讲究”。要知道，busybox作为基础镜像，很多版本是不自带例如nslookup这类网络调试工具的，而nslookup在调试Kube-dns的域名功能时又非常有用。幸运的是，busybox 1.28版自带nslookup，需要的读者可以部署一个做域名解析测试之用。一个使用busybox 1.28版的Pod配置文件如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: busybox:1.28
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

3.9.6 为什么不推荐使用SNAT

如果从外部主机无法直接访问容器，那么容器不可能和外部服务通信。如果一个容器请求外部的服务，由于容器IP是不可路由的，则远程服务器不知道应该把响应发到哪里。但事实上，只要每个主机对容器到外部的连接做一次SNAT或是Masquerade就能实现。

1.问题描述

我们的Docker主机能够和数据中心的其他机器通信，它们有可路由的IP。当一个容器尝试访问一个外部服务时，运行容器的主机将网络包中的容器IP用它本身的IP替换，即Masquerade（SNAT的一种）。对于外部服务，看起来像是和主机建立了连接。当响应返回到主机的时候，它进行一个逆转换（把网络包中的主机IP替换成容器IP）。对于容器，这个操作完全是透明的，它不知道发生了这样的一个转换。Kubernetes NodePort的实现默认就开启了Kube-proxy的--masq-all=true选项。

例如，一个Docker主机10.0.0.1上运行着一个名为container-1的容器，它的IP为172.16.1.8。容器内部的进程初始化一个访问10.0.0.99:80的连接。它

绑定本地容器端口32000。

（1）这个包离开容器到达Docker主机，源地址为172.16.1.8:32000。

（2）Docker主机将源地址由172.16.1.8:32000替换为10.0.0.1:32000，并把包转发给10.0.0.99:80。Linux用一个表格追踪转换过程，以便在包的响应中能够进行逆向转换。

（3）远程服务10.0.0.99:80处理这个请求并返回响应给主机。

（4）响应返回到主机的32000端口。Linux看到这是一个已经转换的连接的响应，便把目的地址从10.0.0.1:32000修改为172.16.1.8:32000，把包转发给容器。

这么做会带来一个问题，就是当并发访问NodePort时会导致丢包。除了NodePort，flannel的--ip-masq=true选项也会引起丢包。

SNAT导致Linux内核丢包的原因在于其conntrack的实现。SNAT代码在POSTROUTING链上被调用两次。首先通过修改源地址和/或端口修改包的结构，如果包在这个过程中没有丢失，则内核在conntrack表中记录这个转换。这意味着在SNAT端口分配和插入conntrack表之间有一个时延，如果有冲突的话可能最终导致插入失败及丢包。当在TCP连接上做SNAT的时候，NAT模块会做以下尝试：

（1）如果包的源地址是在目标NAT池中，且｛IP，端口，协议｝三元组是可用的，返回（包没有改变）。

（2）找到池中最少使用的IP，用它来替换包中的源IP。

（3）检查端口是否在允许的范围（默认1024-64512）内，并且检查带这个端口的三元组是否可用。如果可用则返回（源IP已经改变，端口未改变）。（注意：SNAT的端口范围不受内核参数net.ipv4.ip_local_port_range的影响。）

（4）端口不可用，内核通过调用nf_nat_l4proto_unique_tuple（）请求TCP层找到一个未使用的端口来做SNAT。

当主机上只运行着一个容器时，NAT模块最可能在第三步返回。容器内部进程使用到的本地端口会被保留并用以对外的连接。当在Docker主机上运行多个容器时，很可能一个连接的源端口已经被另一个容器的连接使用。在那种情况下，通过nf_nat_l4proto_unique_tuple（）调用来找到另一个可用的端口进行NAT操作。默认的端口分配做以下的事：

（1）从一个初始位置开始搜索并复制最近一次分配的端口。

（2）递增1。

（3）调用nf_nat_used_tuple（）检查端口是否已被使用。如果已被使

用，重复上一步。

（4）用刚分配的端口更新最近一次分配的端口并返回。

端口分配和把连接插入conntrack表之间有延时，因此当nf_nat_used_tuple（）被并行调用时，对同一个端口的nf_nat_used_tuple（）多次调用可能均回真——当端口分配以相同的初始值开始时，这种现象尤为明显。在我们的测试中，大多数端口分配冲突来自时间间隔在0～2us内初始化的连接。

netfilter也支持两种其他的算法来找到可用的端口：

·使用部分随机来选择端口搜索的初始位置。当SNAT规则带有flag NF_NAT_RANGE_PROTO_RANDOM时，这种模式被使用；

·完全随机来选择端口搜索的初始位置。带有flag NF_NAT_RANGE_PROTO_RANDOM_FULLY时使用。

NF_NAT_RANGE_PROTO_RANDOM降低了两个线程以同一个初始端口开始搜索的次数，但是仍然有很多的错误。只有使用NF_NAT_RANGE_PROTO_RANDOM_FULLY才能显著减少conntrack表插入错误的次数。在一台Docker上测试虚拟机，使用默认的masquerade规则，10到80个线程并发请求连接同一个主机有2%～4%的插入错误。当在内核强制使用完全随机时，丢包率降到了0。

2.解决方法

需要在masquerade规则中设置flag NF_NAT_RANGE_PROTO_RANDOM_FULLY。从1.6.2版本开始的，iptables工具已经支持配置--ramdom-fully这个flag。通过使用打了补丁的flannel和Kube-proxy，能够显著降低conntrack表的插入错误，使整个集群中的丢包错误的数目从每几秒一次下降到每几个小时一次。

需要注意的是，iptabels的--ramdom-fully选项只能缓解集群SNAT带来的这个问题，而并不能根治。因此，我并不推荐在生产环境使用NodePort。

参考资料：https://gravitational.com/blog/troubleshooting-kubernetes-networking.

第4章　刨根问底：Kubernetes网络实现机制

4.1　岂止iptables：Kubernetes Service官方实现细节探秘

下面我们介绍Kubernetes Service的工作原理，主要涉及的Kubernetes组件有Controller Manager（包括Service Controller和Endpoints Controller）和Kube-proxy，如图4-1所示。

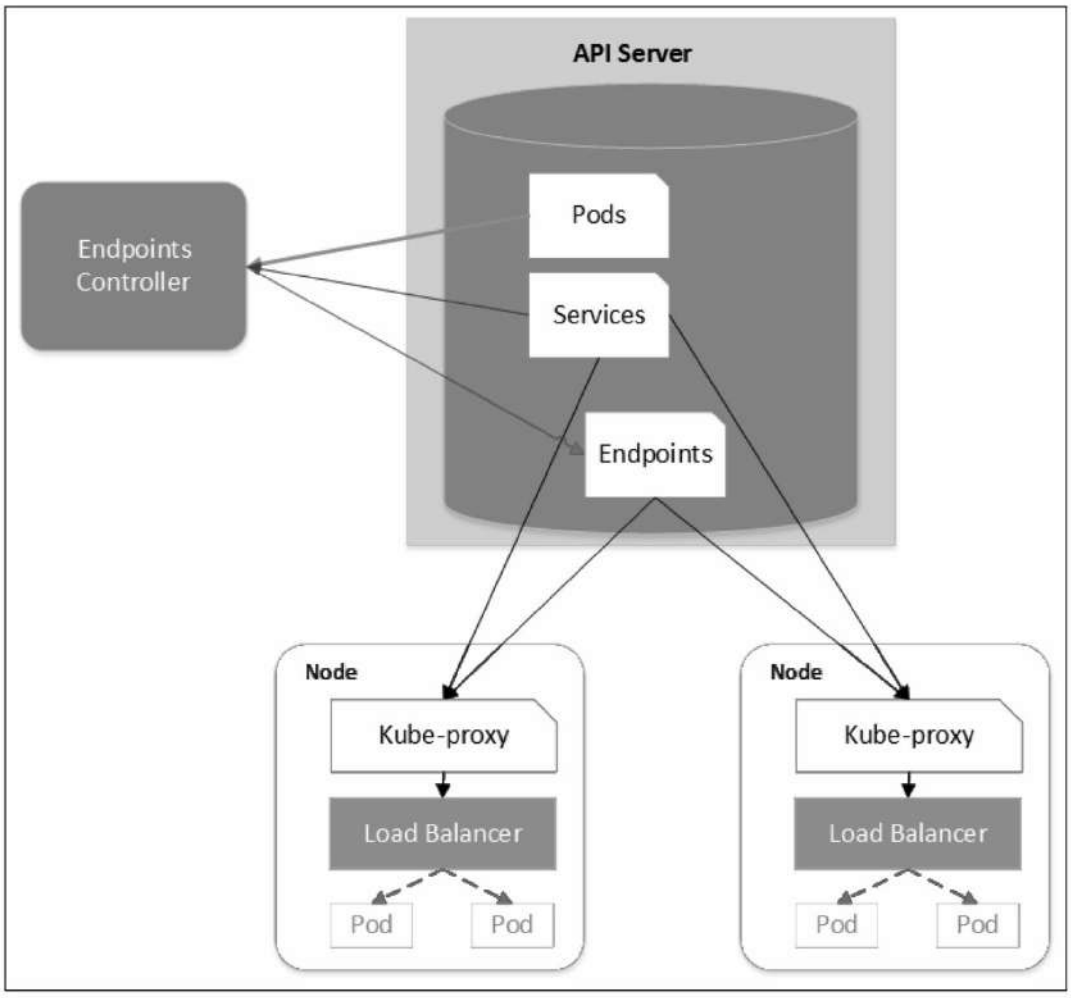


图4-1　Kubernetes Service实现原理

当用户创建Service和对应的后端Pod时，Endpoints Controller会监控Pod的状态变化，当Pod处于Running且准备就绪状态时，Endpoints Controller会生成Endpoints对象。

运行在每个节点上的Kube-proxy会监控Service和Endpoints的更新，并调用其Load Balancer模块在主机上刷新路由转发规则。每个Pod都可以通过Liveness Probe和Readiness probe探针解决健康检查问题，当有Pod处于非准备就绪状态时，Kube-proxy会删除对应的转发规则。需要注意的是，Kube-proxy的Load Balancer模块并不那么智能，如果转发的Pod不能正常提供服务，它不会重试或尝试连接其他Pod。

Kube-proxy的Load Balancer模块实现有userspace、iptables和IPVS三种，当前主流的实现方式是iptables和IPVS。随着iptables在大规模环境下暴露出了扩展性和性能问题，越来越多的厂商开始使用IPVS模式。

Kube-proxy的转发模式可以通过启动参数--proxy-mode进行配置，有userspace、iptables、ipvs等可选项。

4.1.1 userspace模式

Kube-proxy的userspace模式是通过Kube-proxy用户态程序实现Load Balancer的代理服务。userspace模式是Kube-proxy 1.0之前版本的默认模式。由于转发发生在用户态，效率自然不太高，而且容易丢包。

为什么还要花篇幅介绍本应被废弃的特性呢？因为iptables和IPVS模式都依赖Linux内核的能力，尤其是IPVS，而且iptables和IPVS都要求较高版本的内核和iptables版本。那些使用低版本内核的操作系统（例如SUSE 11）用不了iptables和IPVS模式，但又希望拥有基本的服务转发能力，这时userspace模式就派上用场了。userspace模式的Kube-proxy工作原理如图4-2所示。

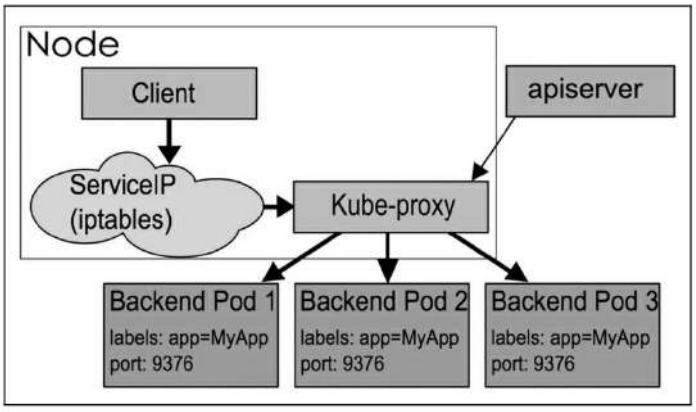


图4-2 userspace模式的Kube-proxy工作原理

不难看出在userspace模式下，访问服务的请求到达节点后首先会进入内核iptables，然后回到用户空间，由Kube-proxy完成后端Pod的选择，并建立一条到后端Pod的连接，完成代理转发工作。这样流量从用户空间进出内核

将带来不小的性能损耗。这也是Kubernetes 1.0及之前版本中对Kube-proxy质疑最大的一点，于是也就有了后来的iptables模式。

至于为什么userspace模式要建立iptables规则，原因是Kube-proxy进程只监听一个端口，而且这个端口并不是服务的访问端口也不是服务的NodePort，因此需要一层iptables把访问服务的连接重定向给Kube-proxy进程的一个临时端口。Kube-proxy在代理客户端请求时会开放一个临时端口，以便后端Pod的响应返回给Kube-proxy，然后Kube-proxy再返回给客户端。

下面将以一个Service为例，讲解userspace模式下的iptables规则。

```
# kubectl describe service ssh-service1
Name:           ssh-service1
Namespace:      default
Labels:         name=ssh,role=service
Selector:       ssh-service=true
Type:           NodePort
IP:             10.254.132.107
Port:           <unnamed>    2222/TCP
NodePort:       <unnamed>    30239/TCP
Endpoints:      <none>
Session Affinity:    None
No events.
```

如上所示，该Service的类型是NodePort，服务端口是2222，NodePort端口是30239，Cluster IP是10.254.132.107。

NodePort的工作原理与Cluster IP大致相同，是发送到Node上指定端口的数据，通过iptables重定向到Kube-proxy对应的端口上。然后由Kube-proxy把数据发送到其中一个Pod上。

```
# iptables -S -t nat
-A KUBE-NODEPORT-CONTAINER -p tcp -m comment --comment "default/ssh-service1:" -m tcp
 --dport 30239 -j REDIRECT --to-ports 36463
-A KUBE-NODEPORT-HOST -p tcp -m comment --comment "default/ssh-service1:" -m tcp --
dport 30239 -j DNAT --to-destination 10.0.0.5:36463
-A KUBE-PORTALS-CONTAINER -d 10.254.132.107/32 -p tcp -m comment --comment "default/
ssh-service1:" -m tcp --dport 2222 -j REDIRECT --to-ports 36463
```

```
-A KUBE-PORTALS-HOST -d 10.254.132.107/32 -p tcp -m comment --comment "default/ssh-
service1:" -m tcp --dport 2222 -j DNAT --to-destination 10.0.0.5:36463
...
```

可以看到Kube-proxy针对不同的服务类型各自创建了iptables入口链。例如，容器访问NodePort的入口是KUBE-NODEPORT-CONTAINER，节点上进程访问NodePort的入口是KUBE-NODEPORT-HOST；容器访问Cluster IP:Port的入口是KUBE-PORTALS-CONTAINER，节点上进程访问Cluster IP:Port的入口是KUBE-PORTALS-HOST。无一例外，访问这4个入口的流量都被DNAT/重定向到本机的36463端口上。36463端口实际上被Kube-proxy监听，流量进入Kube-proxy用户态程序后再转发给后端的Pod。

注：Kube-proxy所在节点的IP地址为10.0.0.5。

4.1.2 iptables模式

从Kubernetes 1.1版本开始，增加了iptables模式，在1.2版本中它正式替代userspace模式成为默认模式。我们在前面的章节介绍过iptables是用户空间应用程序，通过配置Netfilter规则表（Xtables）构建Linux内核防火墙。Kube-proxy iptables模式的原理如图4-3所示。

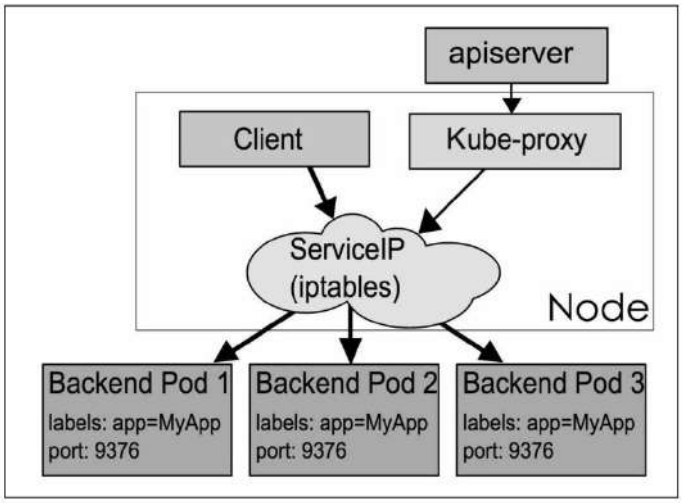


图4-3 Kube-proxy iptables模式的原理

iptables模式与userspace模式最大的区别在于，Kube-proxy利用iptables的DNAT模块，实现了Service入口地址到Pod实际地址的转换，免去了一次内核态到用户态的切换。

下面我们将以一个Service为例，分析Kube-proxy创建的iptables规则。

```
apiVersion: v1
kind: Service
```

```
metadata:
  labels:
    name: mysql
    role: service
  name: mysql-service
spec:
  ports:
    - port: 3306
      targetPort: 3306
      nodePort: 30964
  type: NodePort
  selector:
    mysql-service: "true"
```

如上所示，在本例中我们创建了一个名为mysql-service的服务。该服务的访问端口是3306，NodePort是30964，对应的后端Pod的端口也是3306。另外，虽然上面没有显示，但是该服务的Cluster IP是10.254.162.44。mysql-service服务有两个后端Pod，IP分别是192.168.125.129和192.168.125.131。

Kube-proxy为该服务创建的iptables如下所示：

```
# iptables -S -t nat
-A PREROUTING -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
-A OUTPUT -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
-A POSTROUTING -m comment --comment "kubernetes postrouting rules" -j KUBE-
POSTROUTING
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/mysql-service:" -m tcp --dport
 30964 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/mysql-service:" -m tcp --dport
 30964 -j KUBE-SVC-67RL4FN6JRUPOJYM
-A KUBE-SEP-ID6YWIT3F6WNZ47P -s 192.168.125.129/32 -m comment --comment "default/
mysql-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-ID6YWIT3F6WNZ47P -p tcp -m comment --comment "default/mysql-service:" -m
tcp -j DNAT --to-destination 192.168.125.129:3306
-A KUBE-SEP-IN2YML2VIFH5RO2T -s 192.168.125.131/32 -m comment --comment "default/
mysql-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-IN2YML2VIFH5RO2T -p tcp -m comment --comment "default/mysql-service:" -m
```

```
tcp -j DNAT --to-destination 192.168.125.131:3306
-A KUBE-SERVICES -d 10.254.162.44/32 -p tcp -m comment --comment "default/mysql-
service: cluster IP" -m tcp --dport 3306 -j KUBE-SVC-67RL4FN6JRUPOJYM
-A KUBE-SERVICES -m comment --comment "kubernetes service nodeports; NOTE: this must
be the last rule in this chain" -m addrtype --dst-type LOCAL -j KUBE-NODEPORTS
-A KUBE-SVC-67RL4FN6JRUPOJYM -m comment --comment "default/mysql-service:" -m
statistic --mode random --probability 0.50000000000 -j KUBE-SEP-ID6YWIT3F6WNZ47P
-A KUBE-SVC-67RL4FN6JRUPOJYM -m comment --comment "default/mysql-service:" -j KUBE-
SEP-IN2YML2VIFH5RO2T
...
```

下面来逐条分析。首先，如果通过节点的30964端口访问NodePort，则会进入以下链：

```
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/mysql-service:" -m tcp --dport
 30964 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/mysql-service:" -m tcp --dport
 30964 -j KUBE-SVC-67RL4FN6JRUPOJYM
```

Kube-proxy针对NodePort流量入口专门创建了KUBE-NODEPORTS链。

在我们这个例子中，KUBE-NODEPORTS链进一步跳转到KUBE-SVC-67RL4FN6JRUPOJYM链。

再看下面的iptables规则：

```
-A KUBE-SVC-67RL4FN6JRUPOJYM -m comment --comment "default/mysql-service:" -m
statistic --mode random --probability 0.50000000000 -j KUBE-SEP-ID6YWIT3F6WNZ47P
-A KUBE-SVC-67RL4FN6JRUPOJYM -m comment --comment "default/mysql-service:" -j KUBE-
SEP-IN2YML2VIFH5RO2T
```

这里利用了iptables的random模块，使连接有50%的概率进入KUBE-SEP-ID6YWI T3F6WNZ47P链，50%的概率进入KUBE-SEP-IN2YML2VIFH5RO2T链。因此，Kube-proxy的iptables模式采用随机数实现了服务的负载均衡。

KUBE-SEP-ID6YWIT3F6WNZ47P链的具体作用就是将请求通过DNAT发送到192.168.125.129的3306端口。

```
-A KUBE-SEP-ID6YWIT3F6WNZ47P -s 192.168.125.129/32 -m comment --comment "default/
mysql-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-ID6YWIT3F6WNZ47P -p tcp -m comment --comment "default/mysql-service:" -m
tcp -j DNAT --to-destination 192.168.125.129:3306
```

同理，KUBE-SEP-IN2YML2VIFH5RO2T的作用是通过DNAT发送到192.168.125.131的3306端口。

```
-A KUBE-SEP-IN2YML2VIFH5RO2T -s 192.168.125.131/32 -m comment --comment "default/
mysql-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-IN2YML2VIFH5RO2T -p tcp -m comment --comment "default/mysql-service:" -m
tcp -j DNAT --to-destination 192.168.125.131:3306
```

分析完NodePort的iptables规则后，接下来介绍Cluster IP的访问方式。Cluster IP的访问入口链是KUBE-SERVICES。直接访问Cluster IP（10.254.162.44）的3306端口会跳转到KUBE-SVC-67RL4FN6JRUPOJYM链，如下所示：

```
-A KUBE-SERVICES -d 10.254.162.44/32 -p tcp -m comment --comment "default/mysql-
service: cluster IP" -m tcp --dport 3306 -j KUBE-SVC-67RL4FN6JRUPOJYM
```

接下来的跳转方式同上文分析的NodePort方式类似，这里不再赘述。

综上所述，iptables模式最主要的链是KUBE-SERVICES、KUBE-SVC-*和KUBE-SEP-*。

·KUBE-SERVICES链是访问集群内服务的数据包入口点，它会根据匹配到的目标IP:port将数据包分发到相应的KUBE-SVC-*链；

·KUBE-SVC-*链相当于一个负载均衡器，它会将数据包平均分发到KUBE-SEP-*链。每个KUBE-SVC-*链后面的KUBE-SEP-*链都和Service的后端Pod数量一样；

·KUBE-SEP-*链通过DNAT将连接的目的地址和端口从Service的IP:port替换为后端Pod的IP:port，从而将流量转发到相应的Pod。

最后，我们用图4-4总结Kube-proxy iptables模式的工作流。图4-4演示了从客户端Pod到不同节点上的服务器Pod的流量路径。客户端通过172.16.12.100:80连接到服务。Kubernetes API Server会维护一个运行应用的后端Pod列表。每个节点上的Kube-proxy进程会根据Service和对应的Endpoints创建一系列的iptables规则，以将流量重定向到相应Pod（例如10.255.255.202:8080）。整个过程客户端Pod无须感知集群的拓扑或集群内Pod的任何IP信息。

iptables模式与userspace模式相比虽然在稳定性和性能上均有不小的提升，但因为iptable使用NAT完成转发，也存在不可忽视的性能损耗。另外，当集群中存在上万服务时，Node上的iptables rules会非常庞大，对管理是个不小的负担，性能还会大打折扣。

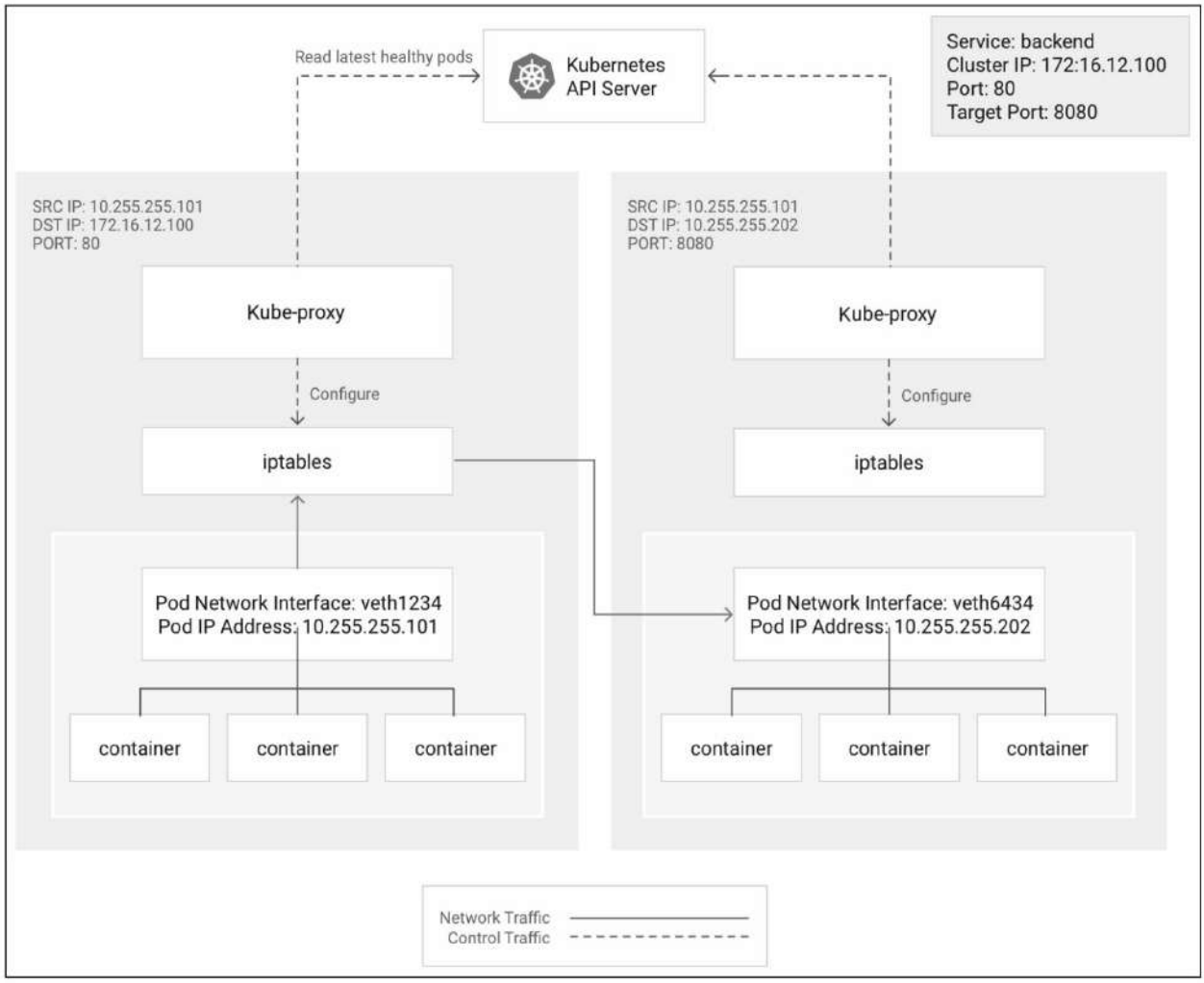


图4-4　Kube-proxy iptables模式的工作流

iptables的SNAT

既然利用iptables做了一次DNAT，为了保证回程报文能够顺利返回，需要做一次SNAT。Kube-proxy创建的对应iptables规则形如：

```
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/mysql-service:" -m tcp --dport
 30964 -j KUBE-MARK-MASQ
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT"
-m mark --mark 0x4000/0x4000 -j MASQUERADE
```

KUBE-MARK-MASQ链本质上使用了iptables的MARK命令。对KUBE-MARK-MASQ链中的所有规则设置了Kubernetes独有的MARK标记（0x4000/0x4000）。在KUBE-POSTROU TING链中，对节点上匹配MARK标记（0x4000/0x4000）的数据包在离开节点时进行一次SNAT，即MASQUERADE（用节点IP替换包的源IP）。

iptables做了一次DNAT后必须做SNAT的原理如图4-5所示，客户端

（C）发起对一个服务（S）的访问，假设目的地址是（C，VIP），那么客户端期待得到的回程报文的源地址是VIP，即回程报文的源和目的地址对应该是（VIP，C）。当网络报文经过网关（Linux内核的netfilter，包括iptables和IPVS director）进行一次DNAT后，报文的源和目的地址对被修改成了（C，S）。当报文送到服务端S后，服务端一看报文的源地址是C便直接把响应报文返回给C，即此时响应报文的源和目的地址对是（S，C）。这与客户端期待的报文源和目的地址对不匹配，客户端收到后会简单丢弃该报文。

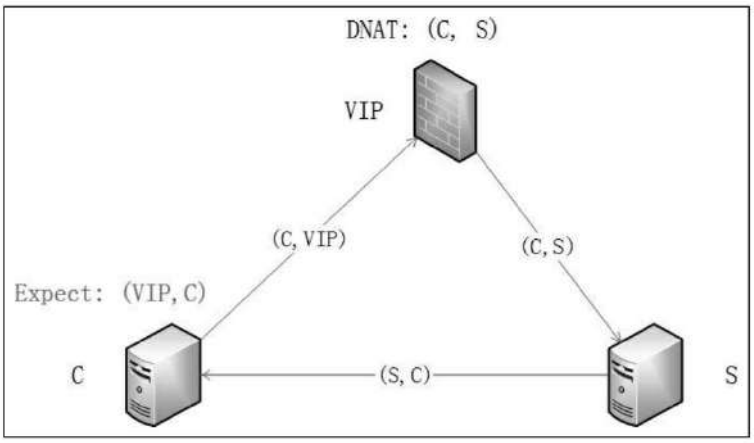


图4-5 iptables SNAT的原理

因此，当报文不直接送达后端服务器，而是访问虚IP，经过一次中间网关（不管是虚拟网关还是实际网关）时，都需要在网关处做一次SNAT把报文的源IP修改成网关IP地址，以便回程报文回到该网关。再让该网关把回程报文目的修改成客户端（C）的IP地址，源地址改成虚IP。

同理，IPVS也有类似的场景，即当IPVS访问VIP并做了一次DNAT后，必须要做一次SNAT才能让报文顺利返回，详见下文。

4.1.3 IPVS模式

IPVS是LVS的负载均衡模块，亦基于netfilter，但比iptables性能更高，具备更好的可扩展性。Kube-proxy的IPVS模式在Kubernetes 1.11版本达到稳定。

既然Kube-proxy已经有了iptables模式，为什么Kubernetes还选择IPVS呢？随着Kubernetes集群规模的增长，其资源的可扩展性变得越来越重要，特别是对那些运行大型工作负载的企业，其服务的可扩展性尤其重要。要知道，iptables难以扩展到支持成千上万的服务，它纯粹是为防火墙而设计的，并且底层路由表的实现是链表，对路由规则的增删改查操作都涉及遍历一次链表。

尽管Kubernetes在1.6版本中已经支持5000个节点，但使用iptables模式的

Kube-proxy实际上是将集群扩展到5000个节点的最大瓶颈。一个例子是，如果我们有1000个服务并且每个服务有10个后端Pod，将在每个工作节点上至少产生10000×N（N≥4）个iptable记录，这可能使内核非常繁忙地处理每次iptables规则的刷新。

另外，使用IPVS做集群内服务的负载均衡可以解决iptables带来的性能问题。IPVS专门用于负载均衡，并使用更高效的数据结构（散列表），允许几乎无限的规模扩张。

1.IPVS的工作原理

IPVS是Linux内核实现的四层负载均衡，是LVS负载均衡模块的实现。上文也提到，IPVS基于netfilter的散列表，相对于同样基于netfilter框架的iptables有更好的性能表现和可扩展性，具体见下文的实测对比数据。

IPVS支持TCP、UDP、SCTP、IPv4、IPv6等协议，也支持多种负载均衡策略，例如rr、wrr、lc、wlc、sh、dh、lblc等。IPVS通过persistent connection调度算法原生支持会话保持功能。LVS的工作原理如图4-6所示。

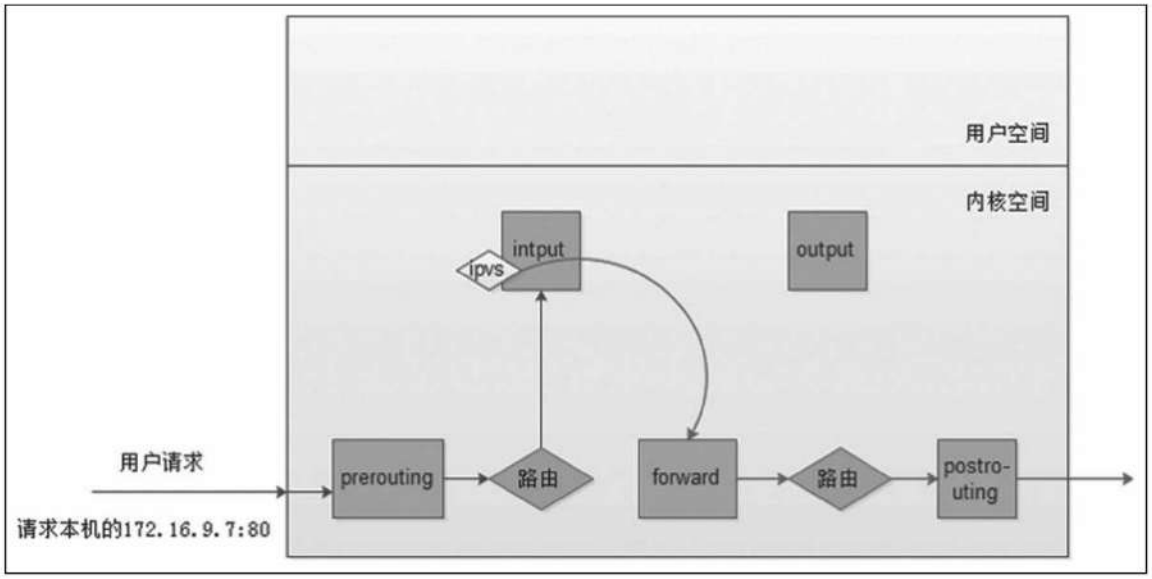


图4-6　LVS的工作原理

简单说，当外机的数据包首先经过netfilter的PREROUTING链，然后经过一次路由决策到达INPUT链，再做一次DNAT后经过FORWARD链离开本机网路协议栈。由于IPVS的DNAT发生在netfilter的INPUT链，因此如何让网路报文经过INPUT链在IPVS中就变得非常重要了。一般有两种解决方法，一种方法是把服务的虚IP写到本机的本地内核路由表中；另一种方法是在本机创建一个dummy网卡，然后把服务的虚IP绑定到该网卡上。Kubernetes使用的是第二种方法，详见下文。

IPVS支持三种负载均衡模式：Direct Routing（简称DR）、Tunneling（也称ipip模式）和NAT（也称Masq模式）。

注：虽然有些版本的IPVS，例如华为和阿里自己维护的分支支持fullNAT，即同时支持SNAT和DNAT，但是Linux内核原生版本的IPVS只做DNAT，不做SNAT。因此，在Kubernetes Service的某些场景下，我们仍需要iptables。

DR

IPVS的DR模式如图4-7所示。DR模式是应用最广泛的IPVS模式，它工作在L2，即通过MAC地址做LB，而非IP地址。在DR模式下，回程报文不会经过IPVS Director而是直接返回给客户端。因此，DR在带来高性能的同时，对网络也有一定的限制，即要求IPVS的Director和客户端在同一个局域网内。另外，比较遗憾的是，DR不支持端口映射，无法支撑Kubernetes Service的所有场景。

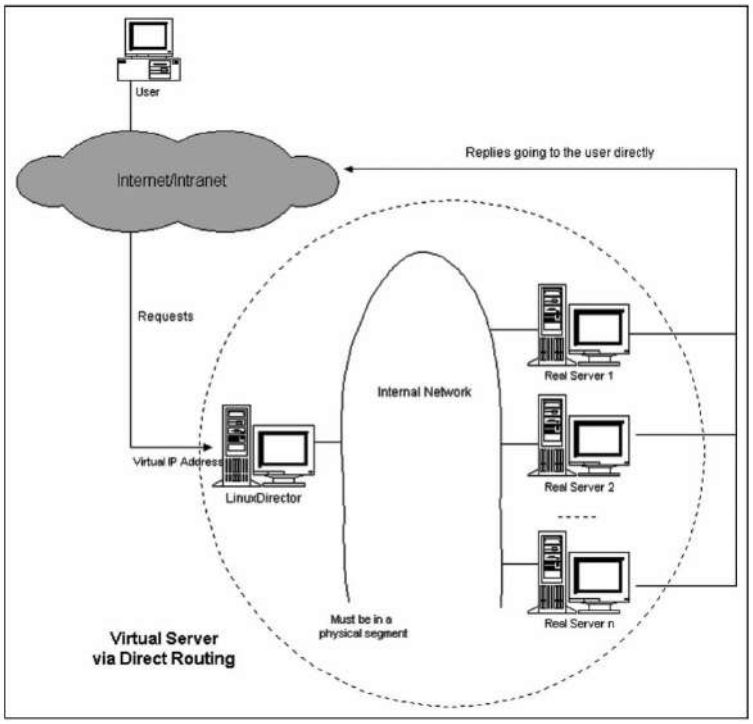


图4-7 IPVS的DR模式

Tunneling

IPVS的Tunneling模式就是用IP包封装IP包，因此也称ipip模式，如图4-8所示。Tunneling模式下的报文不经过IPVS Director，而是直接回复给客户端。Tunneling模式同样不支持端口映射，因此很难被用在Kubernetes的Service场景中。

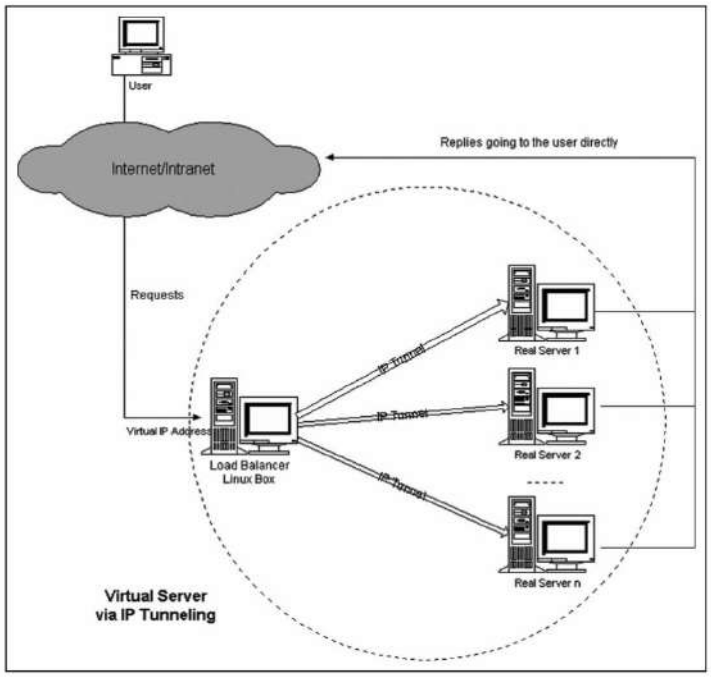


图4-8　IPVS的Tunneling模式

NAT

IPVS的NAT模式支持端口映射，回程报文需要经过IPVS Director，因此也称Masq（伪装）模式，如图4-9所示。Kubernetes在用IPVS实现Service时用的正是NAT模式。当使用NAT模式时，需要注意对报文进行一次SNAT，这也是Kubernetes使用IPVS实现Service的微秒（tricky）之处。

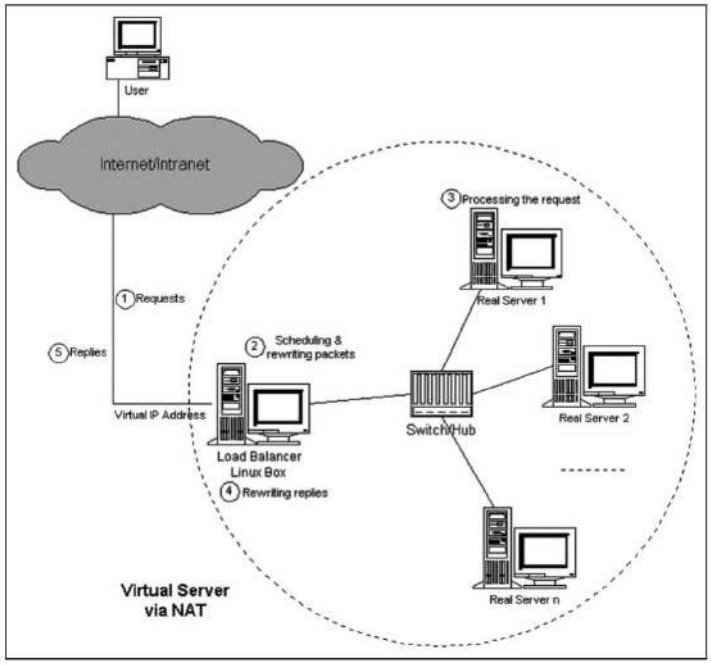


图4-9　IPVS的NAT模式

2.Kube-proxy IPVS模式参数

在运行基于IPVS的Kube-proxy时，需要注意以下参数：

·--proxy-mode：除了现有的userspace和iptables模式，IPVS模式通过--proxy-mode=ipvs进行配置；

·--ipvs-scheduler：用来指定IPVS负载均衡算法，如果不配置则默认使用roundrobin（rr）算法。支持配置的负载均衡算法有：

-rr：轮询

-lc：最小连接

-dh：目的地址哈希

-sh：源地址哈希

-sed：最短时延

未来，Kube-proxy可能实现在Service的annotations配置负载均衡策略，这个功能应该也只有IPVS模式才能支持。

·--cleanup-ipvs：类似于--cleanup-iptables参数。如果设置为true，则清除在IPVS模式下创建的IPVS规则；

·--ipvs-sync-period：表示Kube-proxy刷新IPVS规则的最大间隔时间，例如5秒、1分钟等，要求大于0；

·-ipvs-min-sync-period：表示Kube-proxy刷新IPVS规则的最小时间间隔，例如5秒、1分钟等，要求大于0；

·--ipvs-exclude-cidrs：用于在清除IPVS规则时告知Kube-proxy不要清理该参数配置的网段的IPVS规则。因为我们无法区分某条IPVS规则到底是Kube-proxy创建的，还是其他用户程序的，配置该参数是为了避免误删用户自己的IPVS规则。

目前，本地local-up脚本、GCE安装脚本和kubeadm都支持通过导出环境变量（KUBE_PROXY_MODE=ipvs）切换到IPVS模式。在运行IPVS模式的Kube-proxy前，请确保主机上已经加载了IPVS所需的所有内核模块，如下所示：

```
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_sh
nf_conntrack_ipv4
```

最后，如果你需要在Kubernetes 1.11之前使用IPVS模式的Kube-proxy，需要打开Kubernetes的特性开关，形如--feature-gates=SupportIPVSProxyMode=true。

3.IPVS模式实现原理

Kube-proxy IPVS模式的工作原理如图4-10所示。

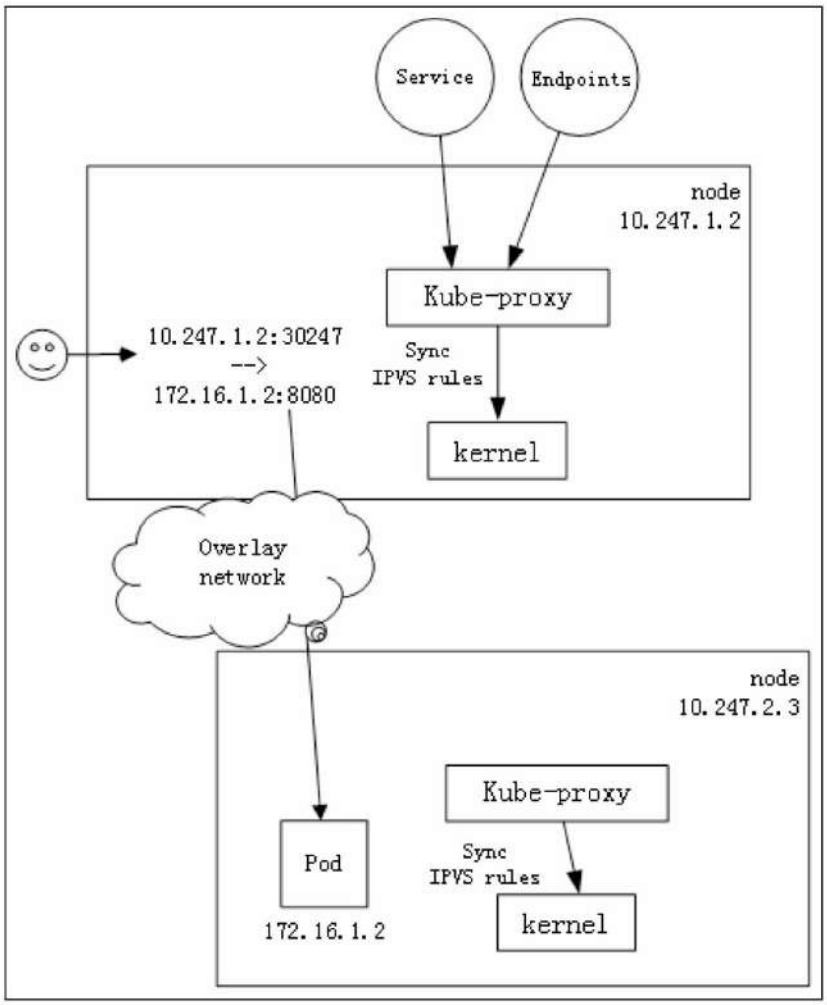


图4-10　Kube-proxy IPVS模式的工作原理

一旦创建一个Service和Endpoints，IPVS模式的Kube-proxy会做以下三件事情。

（1）确保一块dummy网卡（kube-ipvs0）存在。为什么要创建dummy网卡？因为IPVS的netfilter钩子挂载INPUT链，我们需要把Service的访问IP绑定在dummy网卡上让内核“觉得”虚IP就是本机IP，进而进入INPUT链。

（2）把Service的访问IP绑定在dummy网卡上。

（3）通过socket调用，创建IPVS的virtual server和real server，分别对应Kubernetes的Service和Endpoints。

下面我们用一个例子来说明：

```
# kubectl describe svc nginx-service
Name:           nginx-service
Type:           ClusterIP
IP:             10.102.128.4
Port:           http    3080/TCP
Endpoints:      10.244.0.235:8080,10.244.1.237:8080
Session Affinity:   None
...

# ip addr
73: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 1a:ce:f5:5f:c1:4d brd ff:ff:ff:ff:ff:ff
    inet 10.102.128.4/32 scope global kube-ipvs0
       valid_lft forever preferred_lft forever
...

# ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  10.102.128.4:3080 rr
  -> 10.244.0.235:8080            Masq    1      0          0
  -> 10.244.1.237:8080            Masq    1      0          0
...
```

需要注意的是，Kubernetes服务和IPVS virtual server的映射关系是“1：N”。例如，Load Balancer类型的Service有两个服务访问IP，分别是Cluster IP和Load Balancer IP，IPVS模式的Kube-proxy将创建两个IPVS virtual server，其中一个对应Cluster IP，另一个对应Load Balancer IP。Kubernetes的Endpoint（IP+端口）与IPVS real server之间的映射关系是“1：1”。

删除Kubernetes的Service将触发删除相应的IPVS virtual server、IPVS real server和绑定在dummy网卡上的IP地址。

IPVS虽然有三种代理模式NAT、ipip和DR，但只有NAT模式支持端口映射。因此，Kube-proxy的IPVS使用了NAT模式，为的就是支持端口映射。一个IPVS服务端口3080到Pod端口8080的映射的样例如下：

```
TCP  10.102.128.4:3080 rr
  -> 10.244.0.235:8080            Masq    1      0          0
  -> 10.244.1.237:8080            Masq    1      0
```

4.IPVS模式中的iptables和ipset

IPVS用于流量转发，它无法处理Kube-proxy中的其他问题，例如包过滤、SNAT等。具体来说，IPVS模式的Kube-proxy将在以下4种情况下依赖iptables：

·Kube-proxy配置启动参数masquerade-all=true，即集群中所有经过Kube-proxy的包都做一次SNAT；

·Kube-proxy启动参数指定集群IP地址范围；

·支持Load Balancer类型的服务，用于配置白名单；

·支持NodePort类型的服务，用于在包跨节点前配置MASQUERADE，类似于上文提到的iptables模式。

我们不想创建太多的iptables规则，因此使用ipset减少iptables规则，使得不管集群内有多少服务，IPVS模式下iptables规则的总数在5条以内。

4.1.4 iptables VS.IPVS

都说IPVS的性能优于iptables，表4-1为iptables和IPVS在刷新服务路由规则上的时延对比。

表4-1 iptables和IPVS在刷新服务路由规则上的时延对比

Service 数量	iptables 规则数量	增加 1 条 iptables 规则时延	增加 1 条 IPVS 规则时延
1	8	50 us	30 us
5000	40000	11 mins	50 us
20000	160000	5 hours	70 us

通过上表我们可以发现，IPVS刷新规则的时延明显要低iptables几个数量级。

从表4-2可以发现，相较于iptables，IPVS在端到端的吞吐率和平均时延方面均有不小的优化。值得一提的是，这是端到端的数据，包含了底层容器网络的RTT，还能有30%左右的性能提升。

表4-2 iptables和IPVS的吞吐率与平均时延对比

集群转发模式	并发	吞吐量	平均时延
iptables	500	23353/s	30.11 ms
IPVS	500	31094/s	30.06
iptables	1000	28492/s	125.22 ms
IPVS	1000	31361/s	30.16 ms

表4-3对比说明了Kube-proxy在IPVS和iptables模式下的资源消耗。

表4-3　Kube-proxy在IPVS和iptables模式下的资源消耗

Metrios	Service 数量	IPVS	iptables
内存消耗	1000	386MB	1.1GB
	5000	N/A	1.9GB
	10000	542MB	2.3GB
	15000	N/A	00M
	50000	1272MB	00M
CPU 使用率	1000	0%	N/A
	5000		50% 至 85%
	10000		50% 至 100%
	15000		N/A
	50000		N/A

不难看出，无论是资源消耗还是性能，IPVS模式均要优于iptables模式。

4.1.5　conntrack

在内核中，所有由netfilter框架实现的连接跟踪模块称作conntrack（connection tracking）。在DNAT的过程中，conntrack使用状态机启动并跟踪连接状态。为什么需要记录连接的状态呢？因为iptables/IPVS做了DNAT后需要记住数据包的目的地址被改成了什么，并且在返回数据包时将目的地址改回来。除此之外，iptables还可以依靠conntrack的状态（cstate）决定数据包的命运。其中最主要的4个conntrack状态如下：

（1）NEW：匹配连接的第一个包，这表示conntrack对该数据包的信息一无所知。通常发生在收到SYN数据包时。

（2）ESTABLISHED：匹配连接的响应包及后续的包，conntrack知道

该数据包属于一个已建立的连接。通常发生在TCP握手完成之后。

（3）RELATED：RELATED状态有点复杂，当一个连接与另一个ESTABLISHED状态的连接有关时，这个连接就被认为是RELATED。这意味着，一个连接要想成为RELATED，必须先有一条已经是ESTABLISHED状态的连接存在。这个ESTABLISHED状态连接再产生一个主连接之外的新连接，这个新连接就是RELATED状态。

（4）INVALID：匹配那些无法识别或没有任何状态的数据包，conntrack不知道如何处理它。该状态在分析Kubernetes故障的过程中起着重要的作用。

在Pod和Service之间的TCP数据流路径如图4-11所示。

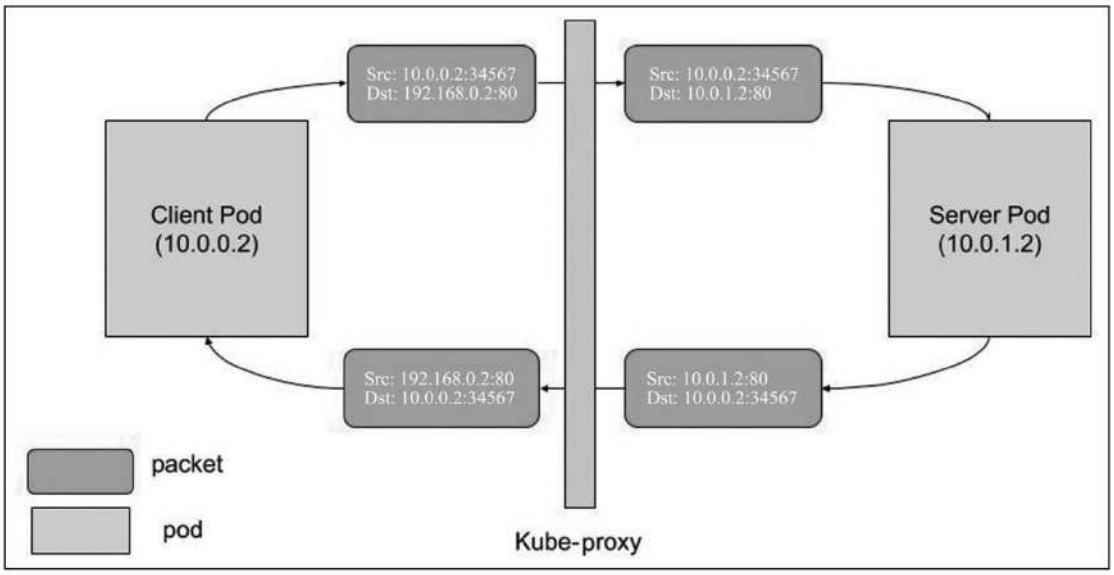


图4-11　Pod和Service之间的TCP数据流路径

TCP连接的生命周期分析如下：

·左边的客户端发送数据包到Service：192.168.0.2:80；

·数据包通过本地节点的iptables规则，目的地址被改为Pod的地址：10.0.1.2:80；

·提供服务的Pod处理完数据包后返回响应包给客户端：10.0.0.2；

·数据包到达客户端所在的节点后，被conntrack模块识别并将源地址改为：192.169.0.2:80；

·客户端接收到响应包。

4.1.6　小结

Kube-proxy实现的是分布式负载均衡，而非集中式负载均衡。何谓分布式负载均衡器呢？就是每个节点都充当一个负载均衡器，每个节点上都会被配置一模一样的转发规则。

上文提到，受制于iptables的实现，iptables模式的转发策略底层实现其实就是随机法，即将请求随机地分配到各个后端Pod（可能在不同节点上）。由概率统计理论得知，随着客户端调用服务端次数的增加，其实际效果越来越接近评价分配，也就是轮询（rr）的结果。缺点也比较明显，就是没有考虑机器的性能问题。根据木桶原理，Service的性能瓶颈会受性能最差的节点影响。那么，支持多种Load Balancer算法的IPVS模式呢？例如，lc（最小连接数）策略能否奏效？受制于Kube-proxy的分布式负载均衡架构，恐怕很难。同一个后端Pod可能有不同的Kube-proxy把请求转发给它，因此任何一个Kube-proxy都无法准确估计其后端Pod的连接数，故最小连接数这种转发策略无法派上用场。不过，可以尝试IPVS模式的sed（最短时延）转发策略。

4.2 Kubernetes极客们的日常：DIY一个Ingress Controller

要使用Kubernetes的Ingress访问集群内服务，需要准备三样东西：

·反向代理负载均衡器；

·Ingress Controller；

·Ingress API。

前文提到，Kubernetes只提供Ingress API的定义，Ingress API的具体实现还要依靠Ingress Controller。前文提到，Kube-proxy是一个基于出站流量的负载均衡控制器，而Ingress Controller则负责Kubernetes进站流量的反向代理负载均衡控制器。反向代理负载均衡（即常见的Nginx、HA proxy等）与Ingress Controller是一一对应的关系。常见的Ingress Controller有以下这些：

·Ingress Controller：Nginx Kubernetes官方维护的方案，安装Ingress Controller会自动安装Nginx；

·F5 BIG-IP Controller：F5所开发的Controller，它能够让管理员通过CLI或API，让Kubernetes与OpenShift管理F5 BIG-IP设备；

·Ingress Kong：著名的开源API Gateway方案所维护的Kubernetes Ingress Controller；

·Traefik：是一套开源的HTTP反向代理与负载均衡器，同时支持Ingress；

·Voyager：基于HA Proxy的Ingress Controller。

Ingress Controller的实现方案远不止上面这些，理论上用户可以基于任意一个Load Balancer实现对应的Ingress Controller。

4.2.1 Ingress Controller的通用框架

Ingress Controller实质上可以理解为监视器，Ingress Controller通过不断地跟Kubernetes API打交道，实时地感知后端Service、Pod等的变化，比如新增和减少Pod，Service增加与减少等；当得到这些变化信息后，Ingress Controller再结合下文的Ingress生成配置，然后更新反向代理负载均衡器，并刷新其配置，起到服务发现的作用。Ingress Controller的通用框架如图4-12所示。

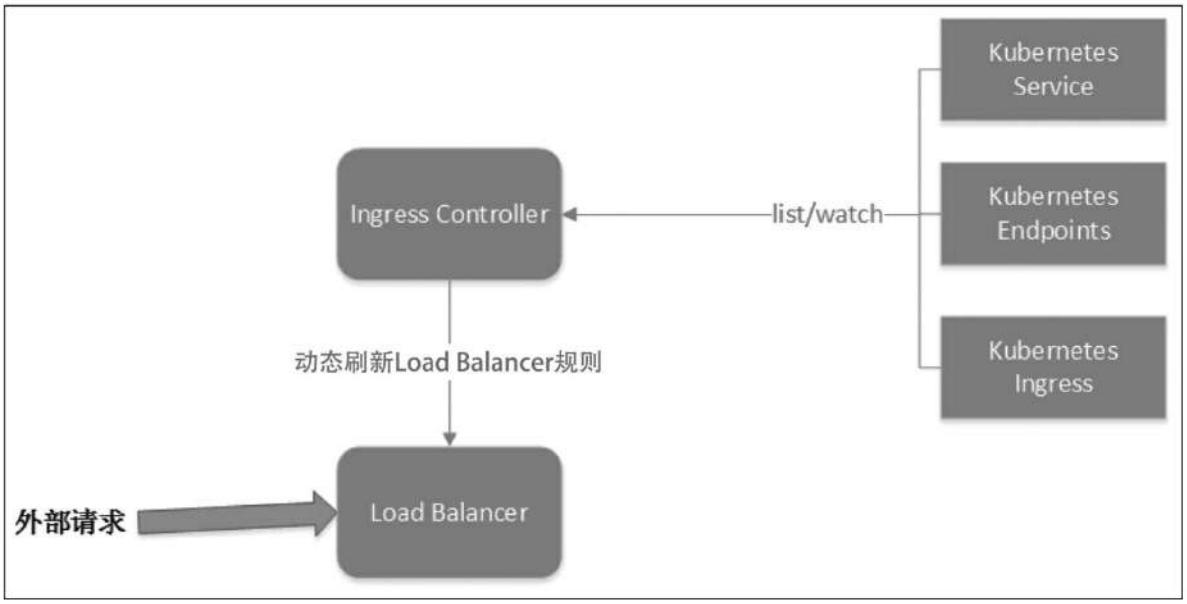


图4-12　Ingress Controller的通用框架

Ingress Controller将Ingress入口地址和后端Pod地址的映射关系（规则）实时刷新到Load Balancer的配置文件中，再让负载均衡器重载（reload）该规则，便可实现服务的负载均衡和自动发现。

4.2.2　Nginx Ingress Controller详解

对绝大多数刚刚接触Kubernetes的人来说，都比较熟悉Nginx Ingress Controller，一个对外暴露Service的7层反向代理。Nginx Ingress Controller通过Kubernetes的annotations配置，为Ingress提供丰富的个性化配置。下面让我们看下Nginx Ingress Controller是如何工作的。

部署一个Nginx Ingress Controller实例的命令非常简单，如下所示：

```
# kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/
deploy/mandatory.yaml
```

其实，上面创建的yaml文件mandatory.yaml包含了Namespace、Configmap、ServiceAccount、ClusterRole、Role、ClusterRoleBinding、RoleBinding、Deployment等多种对象，限于篇幅这里就不一一展开了。这里只展示ClusterRole的配置：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: nginx-ingress-controller
rules:
  - apiGroups:
      - ""
    resources:
      - services
      - endpoints
      - secrets
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
```

上述ClusterRole配置相当于授予了Nginx Ingress Controller允许访问（get、list、watch）Kubernetes的Service、Endpoints、Ingress和secrets（为Nginx Ingress Controller存放证书和密钥文件）的RBAC权限。

Nginx Ingress Controller的Deployment配置文件如下所示：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
```

```
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: ingress-nginx
      app.kubernetes.io/part-of: ingress-nginx
  template:
    metadata:
      labels:
        app.kubernetes.io/name: ingress-nginx
        app.kubernetes.io/part-of: ingress-nginx
      annotations:
        prometheus.io/port: "10254"
        prometheus.io/scrape: "true"
    spec:
      serviceAccountName: nginx-ingress-serviceaccount
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller
:0.23.0
          args:
            - /nginx-ingress-controller
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
            - --tcp-services-configmap=$(POD_NAMESPACE)/tcp-services
            - --udp-services-configmap=$(POD_NAMESPACE)/udp-services
            - --publish-service=$(POD_NAMESPACE)/ingress-nginx
            - --annotations-prefix=nginx.ingress.kubernetes.io
          securityContext:
            allowPrivilegeEscalation: true
            capabilities:
              drop:
                - ALL
              add:
                - NET_BIND_SERVICE
```

```
            # www-data -> 33
            runAsUser: 33
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          ports:
            - name: http
              containerPort: 80
            - name: https
              containerPort: 443
            ...
```

不难看出，Nginx Ingress Controller容器监听在80和443端口上。细心的读者应该已经发现了，Nginx Ingress Controller这个Deployment是在集群内部署的，还需要把它暴露给集群外。如果你的实验环境是裸机或者虚拟机，那么还需要额外对外暴露Nginx Ingress Controller。一个使用NodePort暴露服务的配置如下所示：

```
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
```

```
    - name: https
      port: 443
      targetPort: 443
      protocol: TCP
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
```

当然，你也可以使用其他方式，只要能从集群外访问到就行。读者应该已经知道了，Ingress的IP就是宿主机上的一个IP地址。Nginx Ingress Controller会选择宿主机的一个IP地址（一般是默认网关对应网卡的IP）。

1.测试**HTTP L7**负载均衡

部署了官方的Nginx Ingress Controller后，我们再部署一个Nginx的测试应用并暴露为服务，配置如下所示：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
```

```
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

然后，创建对应的一个Ingress对象，对外暴露之前创建的Nginx服务，配置文件如下所示：

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: nginx-ingress
spec:
  rules:
  - host: nginx.testdomain.com
    http:
      paths:
      - backend:
          serviceName: nginx
          servicePort: 80
```

找到Nginx Ingress Controller对应的容器后查看里面的Nginx配置文件，生成的对应配置段如下所示：

```
# kubectl -n ingress-nginx exec nginx-ingress-controller-6cdcfd8ff9-t5sxl -- cat /etc
/nginx/nginx.conf
    ## start server nginx.testdomain.com
    server {
        server_name nginx.testdomain.com ;

        listen 80;

        set $proxy_upstream_name "-";

        location / {

            set $namespace      "default";
            set $ingress_name   "nginx-ingress";

            set $service_name   "nginx";
            set $service_port   "80";
            set $location_path  "/";
            ........
    ## end server nginx.testdomain.com
...
```

再找一台客户机，设置hosts文件，把域名nginx.testdomain.com设置到和Nginx Ingress Controller所在主机同集群的任意一个节点IP上，打开浏览器访问nginx.testdoma in.com即可发现集群内的Nginx已经暴露在集群外。需要注意的是，Ingress API定义的后端虽然是Kubernetes的Service，但实际上Nginx Ingress Controller是直接负载到Service的Endpoints上的。

另外，低版本的Nginx Ingress Controller的配置参数是--default-backend-service=$（POD_NAMESPACE）/default-http-backend，该参数指定Nginx Ingress Controller的同namespace下，名为default-http-backend的Service作为默认访问时的页面。通常，那时会创建一个404页面，响应不存在请求的Pod和对应Service，我们称之为默认后端（default backend）。Nginx Ingress Controller启动的时候没找到这个默认后端会无法启动。新版本的Nginx Ingress Controller自带了404页面，因此老版本专门为404页面而生的Pod和Service也就不再是必需的了。

2.如何实现**L4**负载均衡

Nginx除了支持L7负载均衡，在新版本中还实现了L4的负载均衡。

Nginx Ingress Controller有以下两个启动参数：

```
--tcp-services-configmap=$(NAMESPACE)/tcp-services
--udp-services-configmap=$(NAMESPACE)/udp-services
```

Ingress API本身并没有关于L4负载均衡的相关定义，因此要想使用Nginx的L4代理功能（包括TCP和UDP），需要在Configmap中配置并传给Nginx Ingress Controller。上面例子中指定了存储转发信息的两个Configmap对象。

下面这个例子就是上文提到的ingress-nginx/tcp-services这个Configmap对象，意思是访问Ingress的3306端口，流量转发给default namespace中名为mysql的Service的3306端口。

```
kind: ConfigMap
apiVersion: v1
```

```
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  3306: "default/mysql:3306"
```

4.2.3 小结

问个开放性问题，Kubernetes为什么要发明Ingress这个概念呢？笔者认为，其中一个重要的原因便是服务动态发现和负载均衡。在微服务的开发模式下，外部网络要通过域名、UR路径、负载均衡等转发到后端私有网络中，微服务之所以称为微，是因为它是动态变化的，它会经常被增加、删除或更新。传统的反向代理对服务动态变化的支持不是很方便，也就是服务变更后，不是很容易马上改变配置和热加载。Nginx Ingress Controller的出现就是为了解决这个问题，它可以时刻监听服务注册或服务编排API，随时感知后端服务变化，自动重新更改配置并重新热加载，期间服务不会暂停或停止，这对用户来说是无感知的。

因为微服务架构及Kubernetes等编排工具最近几年才开始逐渐流行，所以一开始的反向代理服务器（例如Nginx和HA Proxy）并未提供对微服务的支持，才会出现Nginx Ingress Controller这种中间层做Kubernetes和负载均衡

器（例如Nginx）之间的适配器（adapter）。Nginx Ingress Controller的存在就是为了与Kubernetes交互，同时刷新Nginx配置，还能重载Nginx。而号称云原生边界路由的Traefik设计得更彻底，首先它是个反向代理，其次原生提供了对Kubernetes的支持，也就是说，Traefik本身就能跟Kubernetes打交道，感知Kubernetes集群服务的更新。Traefik是原生支持Kubernetes Ingress的，因此用户在使用Traefik时无须再开发一套Nginx Ingress Controller，受到了广大运维人员的好评。相比Nginx和HA Proxy这类老古董，Traefik设计思想比较先进，有点“Envoy+Istio”降维打击Nginx的意思。

4.3 沧海桑田：Kubernetes DNS架构演进之路

Kubernetes DNS服务目前有两个实现，分别是Kube-dns和CoreDNS。

4.3.1 Kube-dns的工作原理

Kube-dns架构历经两个较大的变化。Kubernetes 1.3之前使用etcd+kube2sky+SkyDNS的架构，Kubernetes 1.3之后使用kubedns+dnsmasq+exechealthz的架构，这两种架构都利用了SkyDNS的能力。SkyDNS支持正向查找（A记录）、服务查找（SRV记录）和反向IP地址查找（PTR记录）。下面我们将分别详解Kube-dns的这两种架构。

1.etcd+kube2sky+SkyDNS

etcd+kube2sky+SkyDNS属于第一个版本的Kube-dns架构，包含三个重要的组件，分别是：

·etcd：存储所有DNS查询所需的数据；

·kube2sky：观察Kubernetes API Server处Service和Endpoints的变化，然后同步状态到Kube-dns自己的etcd；

·SkyDNS：监听在53端口，根据etcd中的数据对外提供DNS查询服务。etcd+kube2sky+SkyDNS版本的Kube-dns架构如图4-13所示。

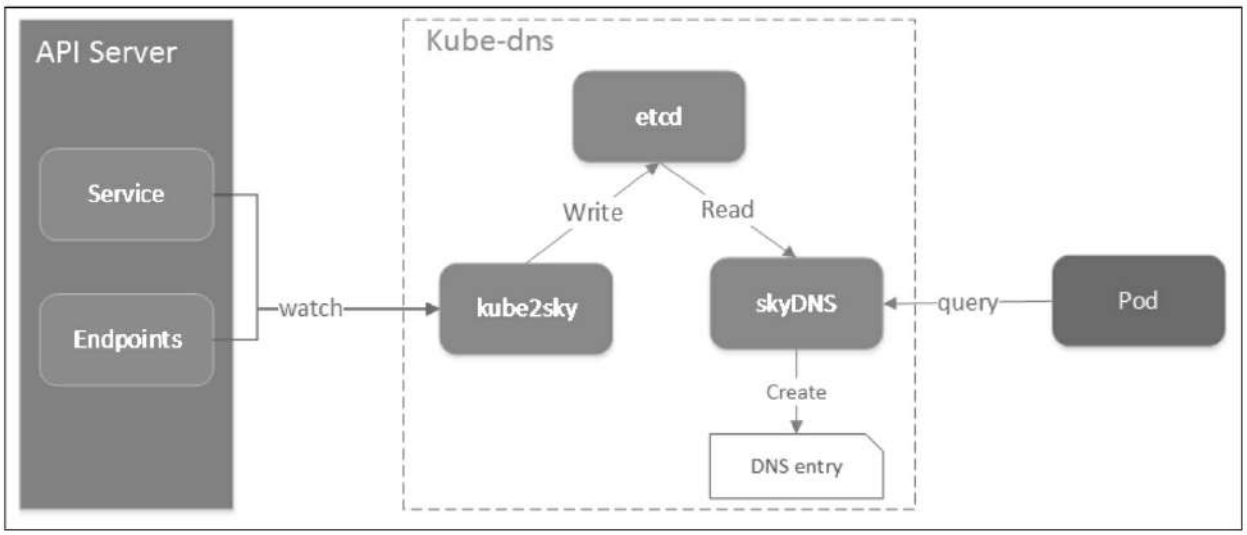


图4-13 etcd+kube2sky+SkyDNS版本的Kube-dns架构

SkyDNS配置etcd作为后端数据储存，当Kubernetes cluster中的DNS请求被SkyDNS接受时，SkyDNS从etcd中读取数据，然后封装数据并返回，完成DNS请求响应。Kube2Sky通过watch Service和Endpoints更新etcd中的数据。

假设etcd容器ID为0fb60dcfb8b4，下面让我们一窥Service的Cluster IP在etcd中的数据存储：

```
# docker exec -it 0fb60d etcdctl get /skydns/local/kube/svc/default/mysql-service
{"host":"10.254.162.44","priority":10,"weight":10,"ttl":30,"targetstrip":0}
```

如上所示，位于default namespace的服务mysql-service的Cluster IP为10.254.162.44。

SkyDNS的基本配置信息也存在etcd中，例如Kube-dns IP地址、域名后缀等。如下所示：

```
# docker exec -it 0fb60d etcdctl get /skydns/config
{"dns-addr":"10.254.10.2:53","ttl":30,"domain":"cluster.local"}
```

不难看出，SkyDNS是真正对外提供DNS查询服务的组件，kube2sky是Kubernetes到SkyDNS之间的“桥梁”，而etcd则存储Kubernetes只涉及域名解析部分的API对象。这个版本的Kube-dns选择直接部署SkyDNS进程来提供域名解析服务。

2.kubedns+dnsmasq+exechealthz

新演进的Kube-dns架构如图4-14所示。

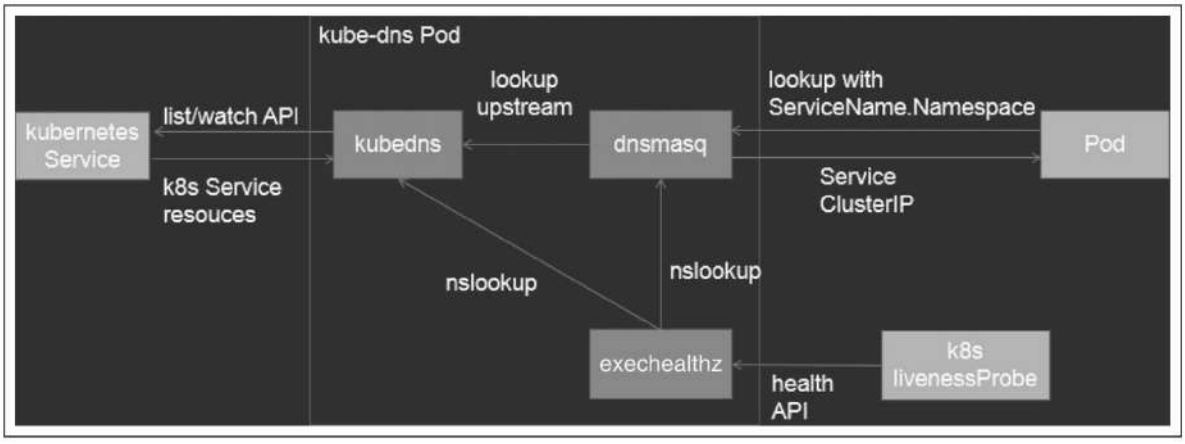


图4-14 新演进的Kube-dns架构

Kube-dns包含以下三个核心组件：

·kubedns：从Kubernetes API Server处观察Service和Endpoints的变化并调

用SkyDNS的golang库，在内存中维护DNS记录。kubedns作为dnsmasq的上游在dnsmasq cache未命中时提供DNS数据；

·dnsmasq：DNS配置工具，监听53端口，为集群提供DNS查询服务。dnsmasq提供DNS缓存，降低了kubedns的查询压力，提升了DNS域名解析的整体性能；

·exechealthz：健康检查，检查Kube-dns和dnsmasq的健康，对外提供/healthz HTTP接口以查询Kube-dns的健康状况。

这个版本的Kube-dns和之前版本的区别在于：

·从Kubernetes API Server那边观察（watch）到的Service和Endpoints对象没有存在etcd，而是缓放在内存中，既提高了查询性能，也省去了维护etcd存储的工作量；

·引入了dnsmasq容器，由它接受Kubernetes集群中的DNS请求，目的就是利用dnsmasq的cache模块，提高解析性能；

·没有直接部署SkyDNS，而是调用了SkyDNS的golang库，相当于之前的SkyDNS和Kube2Sky整合到了一个进程中；

·增加了健康检查功能。

运行过程中，dnsmasq在内存中预留一个缓冲区（默认是1GB），保存最近使用到的DNS查询记录。如果缓存中没有要查找的记录，dnsmasq会去kubeDNS中查询，同时把结果缓存起来。

需要注意的是，dnsmasq是一个C++写的小程序，有内存泄漏的“小毛病”。

综上所述，无论是哪个版本的架构，Kube-dns的本质就是一个Kubernetes API对象的监视器+SkyDNS。

4.3.2 上位的CoreDNS

CoreDNS作为CNCF中托管的一个域名发现的项目，原生集成Kubernetes，它的目标是成为云原生的DNS服务器和服务发现的参考解决方案。所以，CoreDNS走的也是Traefik的路子，降维打击SkyDNS。

从Kubernetes 1.12开始，CoreDNS就成了Kubernetes的默认DNS服务器，但kubeadm默认安装CoreDNS的时间要更早。在Kuberentes 1.9版本中，使用kubeadm方式安装的集群可以通过以下命令直接安装CoreDNS。

```
# kubeadm init --feature-gates=CoreDNS=true
```

下面，我们将详细解释CoreDNS的架构设计和基本用法。

从功能角度看，CoreDNS更像是一个通用的DNS方案，通过插件模式极大地扩展自身功能，从而适用于不同的场景。正如CoreDNS官方博客描述的那样：

> CoreDNS is powered by plugins.

CoreDNS有以下3个特点。

（1）插件化（**Plugins**）。基于Caddy服务器框架，CoreDNS实现了一个插件链的架构，将大量应用端的逻辑抽象成插件的形式（例如，Kubernetes的DNS服务发现、Prometheus监控等）暴露给使用者。CoreDNS以预配置的方式将不同的插件串成一条链，按序执行插件链上的逻辑。在编译层面，用户选择需要的插件编译到最终的可执行文件中，使得运行效率更高。CoreDNS采用Go语音编写，所以从代码层面来看，每个插件其实都只实现了CoreDNS定义的接口的组件而已。第三方开发者只要按照CoreDNS Plugin API编写自定义插件，就可以很方便地集成到CoreDNS中。

（2）配置简单化。引入表达力更强的DSL，即Corefile形式的配置文件（也是基于Caddy框架开发的）。

（3）一体化的解决方案。区别于Kube-dns“三合一”的架构，CoreDNS编译出来就是一个单独的可执行文件，内置了缓存、后端存储管理和健康检查等功能，无须第三方组件辅助实现其他功能，从而使部署更方便，内存管理更安全。

1.Corefile知多少

Corefile是CoreDNS的配置文件（源于Caddy框架的配置文件Caddyfile），它定义了：

·DNS server以什么协议监听在哪个端口（可以同时定义多个server监听不同端口）；

·DNS负责哪个zone的权威（authoritative）DNS解析；

·DNS server将加载哪些插件。

通常，一个典型的Corefile格式如下：

```
ZONE:[PORT] {
    [PLUGIN] ...
}
```

·ZONE：定义DNS server负责的zone，PORT是可选项，默认为53；

·PLUGIN：定义DNS server要加载的插件，每个插件可以有多个参数。

例如：

```
. {
    chaos CoreDNS-001
}
```

上述配置文件表达的是：DNS server负责根域.的解析，其中插件是chaos且没有参数。

1）定义**DNS server**

一个最简单的DNS server配置文件如下：

```
.{}
```

即DNS server监听53端口并且不使用任何插件。如果此时定义其他DNS server，需要保证监听端口不冲突。如果是在原来DNS server的基础上增加zone，则要保证zone之间不冲突。例如：

```
.    {}
.:54 {}
```

如上所示，另一个DNS server监听在54端口并负责根域.的解析。

又如：

```
example.org {
    whoami
}
org {
    whoami
}
```

这是同一个DNS server但是负责不同zone的解析，而且有不同的插件链。

2）定义**Reverse Zone**

跟其他DNS服务器类似，Corefile也可以定义Reverse Zone：

```
0.0.10.in-addr.arpa {
    whoami
}
```

或者简化版本：

```
10.0.0.0/24 {
    whoami
}
```

3）使用不同的通信协议

CoreDNS除了支持DNS协议，也支持TLS和gRPC，即DNS-over-TLS和DNS-overgRPC模式。例如：

```
tls://example.org:1443 {
#...
}
```

2.插件工作模式

当CoreDNS启动后，它将根据配置文件启动不同的DNS server，每个DNS server都拥有自己的插件链。当新来一个DNS请求时，它将依次经历以下3步逻辑：

（1）如果当前请求的DNS server有多个zone，则将采用贪心原则选择最匹配的zone。

（2）一旦找到匹配的DNS server，按照plugin.cfg定义的顺序遍历执行插件链上的插件。

（3）每个插件将判断当前请求是否应该处理，将有以下几种可能的情况：

·请求被当前插件处理。插件将生成对应的响应并返回客户端，此时请求结束，下一个插件将不会被调用，如whoami插件；

·请求不被当前插件处理。直接调用下一个插件。如果最后一个插件执行错误，服务器返回SERVFAIL响应；

·请求被当前插件以**Fallthrough**的形式处理。如果请求在该插件处理过程中有可能跳转至下一个插件，该过程称为fallthrough，并以关键字fallthrough决定是否允许此项操作。例如，host插件尝试用/etc/hosts查询域

名，如果没有查询到则会调用下一个插件；

·请求在处理过程携带**hint**。请求被插件处理，并在其响应中添加了某些提示信息（hint），继续交由下一个插件处理。这些额外的信息将组成对客户端的最终响应，例如metric插件。

3.CoreDNS请求处理工作流

下面将以一个实际的Corefile为例，详解CoreDNS处理DNS请求的工作流。Corefile如下所示：

```
coredns.io:5300 {
    file /etc/coredns/zones/coredns.io.db
}

example.io:53 {
    errors
    log
    file /etc/coredns/zones/example.io.db
}

example.net:53 {
    file /etc/coredns/zones/example.net.db
}

.:53 {
    errors
```

```
    log
    health
    rewrite name foo.example.com foo.default.svc.cluster.local
}
```

通过配置文件不难看出，我们定义了两个DNS server（尽管有4个配置块），分别监听5300和53端口。将以上Corefile翻译成处理逻辑图，如图4-15所示。

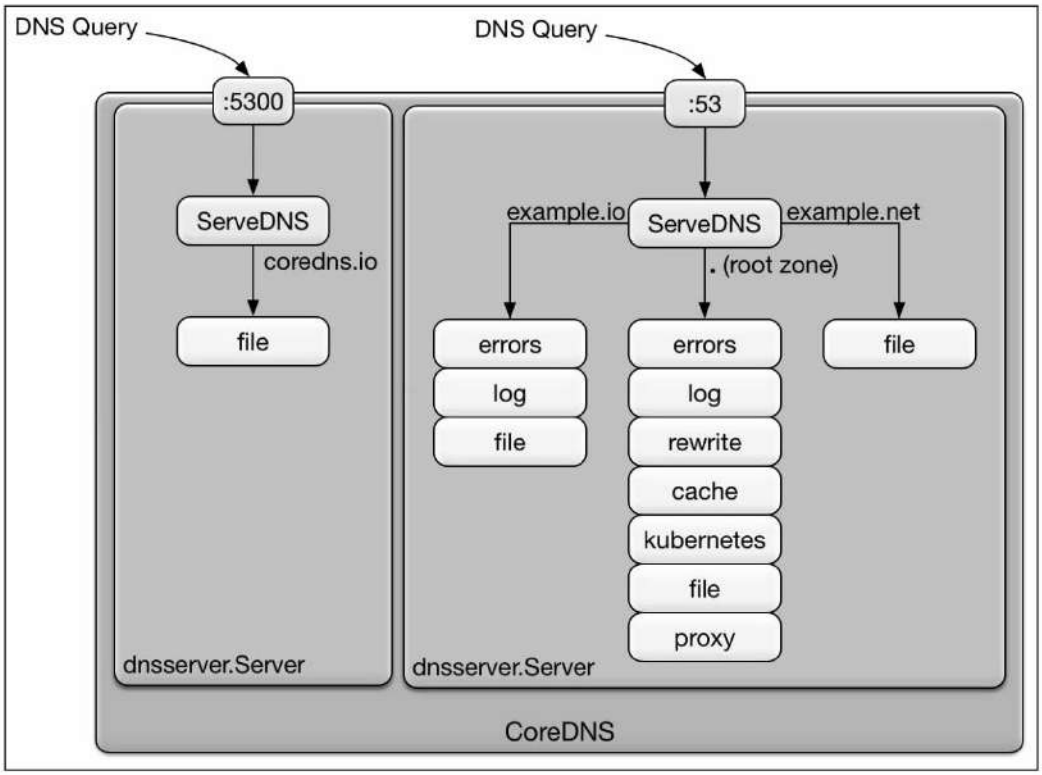


图4-15　CoreDNS请求处理流程

每个进入某个DNS server的请求将按照plugin.cfg的定义顺序执行其已经加载的插件。

需要注意的是，尽管在.:53配置了health插件，但是它并未在上面的逻辑图中出现，原因是该插件并未参与请求相关的逻辑（即并没有在插件链上），只是修改了DNS server的配置。通常，我们可以将插件分为两种：

·Normal插件：参与请求相关的逻辑，且插入插件链中；

·其他插件：不参与请求相关的逻辑，也不出现在插件链中，只是用于修改DNS server的配置，例如health、tls等插件。

4.性能测试下面介绍CoreDNS的域名解析性能和资源消耗情况，如表4-4所示。

表4-4　CoreDNS的域名解析性能和资源消耗情况

Services (with 1% change per minute)	Max QPS	Latency (Median)	CoreDNS memory (at max QPS)	CoreDNS CPU (at max QPS)
1,000	18,000	0.1 ms	38 MB	95%
5,000	16,000	0.1 ms	73 MB	93%
10,000	10,000	0.1 ms	115 MB	78%

值得一提的是，以上性能测试数据是在不带cache的情况下取得的，一般情况下要高于Kube-dns，具体的CoreDNS与Kube-dns的性能对比数据请看下文。

4.3.3 Kube-dns VS.CoreDNS

虽然Kube-dns血统纯正，而且早早地进入Kubernetes的“后宫”，也早有“名分”，但近来CoreDNS却独得Kubernetes Network工作小组核心成员“圣宠”，它不仅早早地进入CNCF，就连其中一位创始人也被挖到谷歌的Kubernetes核心团队担任资深工程师（senior staff engineer）。

与Kube-dns的三进程架构不同，CoreDNS就一个进程，运维起来更加简单。而且采用Go语言编写，内存安全，高性能。值得称道的是，CoreDNS采用的是“插件链”架构，每个插件挂载一个DNS功能，保证了功能的灵活、易扩展。尽管资历不深，却“集万千宠爱于一身”，自然是有绝技的。

下面我们从性能和功能两个维度全面对比Kube-dns和CoreDNS。

1.性能对比

下面从内存、CPU、QPS和时延这4个维度对比Kube-dns和CoreDNS的性能和资源消耗。

内存和**CPU**消耗

CoreDNS和Kube-dns都维护集群中所有服务和端点的本地缓存。因此，随着服务和端点数量的增加，每个DNS Pod的内存要求也会增加。在默认设置下，CoreDNS应该比Kube-dns使用更少的内存，这是由于部署Kube-dns需要使用三个容器，而CoreDNS只要一个容器便足够。

图4-16对比了随着集群内服务和Pod数量的增加，CoreDNS和Kube-dns消耗的内存。

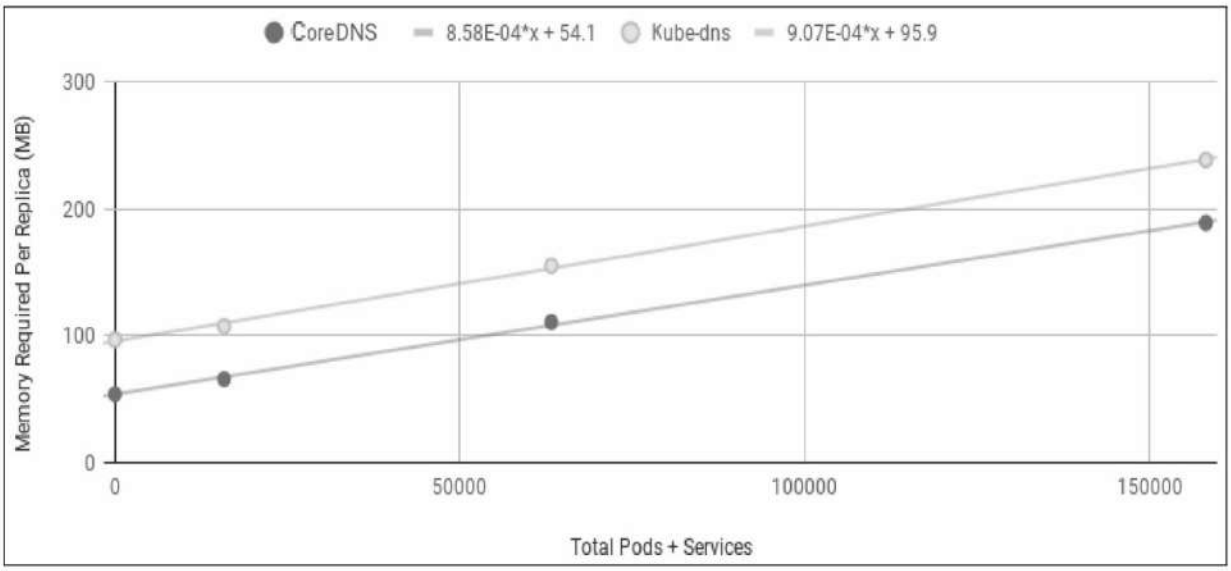


图4-16　CoreDNS和Kube-dns内存消耗对比

当Kube-dns和CoreDNS分别达到最大QPS负载时，观测CPU的使用量分别是58Mi（Kube-dns）和5Mi（CoreDNS）。

注：Mi是CPU核心数的度量单位，代表千分之一核，因此58Mi即0.058核，5Mi即0.005核。

QPS和时延

从表4-5所示的数据中可以看出：

·Kube-dns对外部域名解析（例如kubernetes.default.svc.cluster.local）的表现比CoreDNS要好10%左右。这可能是因为dnsmasq比CoreDNS的内置缓存做了更多的性能优化；

·CoreDNS对外部域名解析的性能提高了约3倍。部分原因可能是Kube-dns没有在缓存中存储DNS查询的负面响应（例如访问一个不存在的域名）。然而，即使Kube-dns在缓存中存储了负面响应也没有明显的区别，数据如表4-6所示。

表4-5　Kube-dns和CoreDNS的QPS和时延对比

DNS Server	Query Type	QPS	Avg Latency (ms)
CoreDNS	external	6733	12.02
CoreDNS	internal	33669	2.608
Kube-dns	external	2227	41.585
Kube-dns	internal	36648	2.639

表4-6　Kube-dns在缓存中存储了负面响应时响应内/外部域名请求的QPS与时延

DNS Server	Query Type	QPS	Avg Latency (ms)
Kube-dns + neg-cache	external	2552	36.665
Kube-dns + neg-cache	internal	28971	3.385

2.功能对比

为什么建议使用CoreDNS呢？Kubernetes官方已经将CoreDNS“扶正”，成为了默认模式，除了性能好，还有什么其他优势吗？Core DNS修复了Kube-dns的一些“令人讨厌”的的问题：

·dns#55-Allow custom DNS entries for Kube-dns（允许Kube-dns自定义DNS记录）；

·dns#116-Missing‘A’records for headless service with pods sharing hostname（当后端Pod共享主机名时，headless Service丢失A记录）；

·dns#131-ExternalName not using stubDomains settings（stubDomains配置对External-Name Service不生效）；

·dns#167-Enable round robin A/AAAA records（启用Kube-dns A/AAAA记录的轮询）；

·dns#190-Kube-dns cannot run as non-root user（只有root用户才能运行Kube-dns）；

·dns#232-Use pod’s name instead of pod’s hostname in DNS SRV records（在DNS SRV记录中使用Pod名而不是Pod主机名）。

同时，还有一些吸引人的特性：

·Zone transfers-list all records，or copy records to another server（CoreDNS允许将全部或部分的DNS记录复制到另一个CoreDNS服务器）；

·Namespace and label filtering-expose a limited set of services（基于namespace和labels过滤一部分Service记录）；

·Adjustable TTL-adjust up/down default service record TTL（动态调整DNS记录的TTL）；

·Negative Caching-By default caches negative responses（默认存储DNS查询的负面响应）；

其中，原生支持基于namespace和labels隔离及过滤Service和Pod的DNS记录这一特性，在多租户场景下格外有用。

4.3.4 小结

无论是Kube-dns还是CoreDNS，基本原理都是利用watch Kubernetes的Service和Pod，生成DNS记录，然后通过重新配置Kubelet的DNS选项让新启动的Pod使用Kube-dns或CoreDNS提供的Kubernetes集群内域名解析服务。

4.4 你的安全我负责：使用Calico提供Kubernetes网络策略

与Kubernetes Ingress API类似，Kubernetes只提供了Network Policy的API定义，不负责具体实现。实现Network Policy的控制器称之为Policy Controller。Network Policy是Kubernetes对Pod的网络隔离手段，对应到宿主机上是一系列iptables规则。这些iptables规则就是由Policy Controller配置的。Policy Controller实现Kubernetes网络策略的一般架构如图4-17所示。

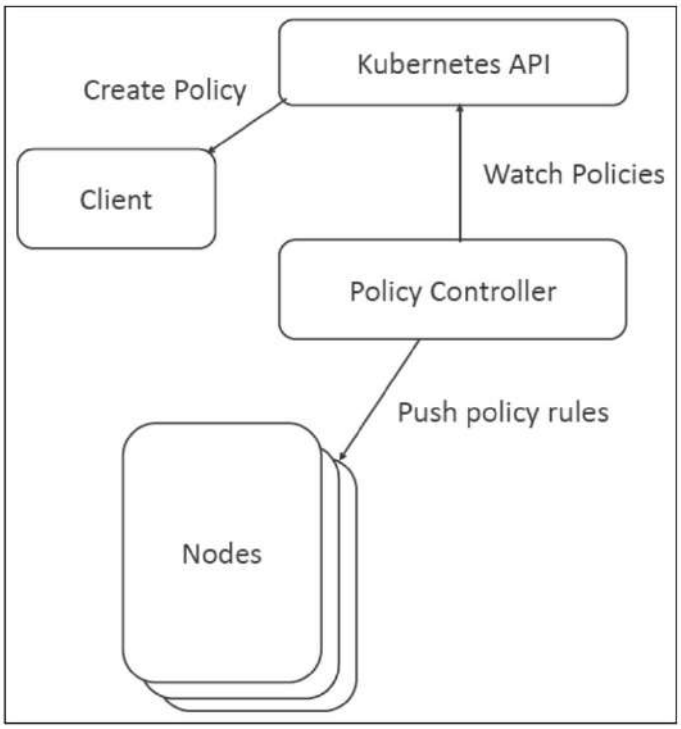


图4-17 Policy Controller实现Kubernetes网络策略的一般架构

通常，Policy Controller是由Kubernetes网络插件提供的。支持Network Policy的网络插件有Calico、Cilium、Weave Net、Kube-router、Romana等。需要注意的是，flannel不在这个名单中。

虽然Calico也有提供容器网络的能力，但下面我们将主要展示Calico是如何提供对Kubernetes网络策略的支持的。

4.4.1 部署一个带Calico的Kubernetes集群

使用以下命令，GCE的用户就能一键部署一个使用Calico提供网络策略的Kubernetes集群：

```
export NETWORK_POLICY_PROVIDER=calico
export KUBE_NODE_OS_DISTRIBUTION=debian
curl -sS https://get.k8s.io | bash
```

当然，你也可以先部署一个Kubernetes集群再安装Calico，下面我们以kubeadm为例进行讲解。

（1）先初始化一个Kubernetes的Master节点：

```
# kubeadm init --pod-network-cidr=192.168.0.0/16
```

其中，--pod-network-cidr=192.168.0.0/16指的是整个集群Pod的IP地址范围（192.168.0.0/16），因此要确保这个网段还没被占用。

（2）为kubectl配置好kubeconfig，使得我们可以通过kubectl访问Kubernetes API Server：

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

（3）安装Calico网络插件。我们使用Kubernetes的add-on机制（即把Calico部署在Kubernetes上）部署Calico的各个组件。

```
# kubectl apply -f \
https://docs.projectcalico.org/v3.6/getting-started/kubernetes/installation/hosted/
kubernetes-datastore/calico-networking/1.7/calico.yaml
```

calico.yaml文件包含了太多运行Calico集群所需的Kubernetes API资源对象，我们无法一一罗列，只列举calico-config这个ConfigMap：

```
# This ConfigMap is used to configure a self-hosted Calico installation.
kind: ConfigMap
apiVersion: v1
metadata:
  name: calico-config
  namespace: kube-system
data:
```

```
# Typha is disabled.
typha_service_name: "none"
# Configure the Calico backend to use.
calico_backend: "bird"

# Configure the MTU to use
veth_mtu: "1440"

# The CNI network configuration to install on each node.  The special
# values in this config will be automatically populated.
cni_network_config: |-
  {
    "name": "k8s-pod-network",
    "cniVersion": "0.3.0",
    "plugins": [
      {
        "type": "calico",
        "log_level": "info",
        "datastore_type": "kubernetes",
        "nodename": "__KUBERNETES_NODE_NAME__",
        "mtu": __CNI_MTU__,
        "ipam": {
            "type": "calico-ipam"
        },
        "policy": {
            "type": "k8s"
        },
        "kubernetes": {
            "kubeconfig": "__KUBECONFIG_FILEPATH__"
        }
      },
      {
        "type": "portmap",
        "snat": true,
        "capabilities": {"portMappings": true}
      }
    ]
```

```
    }
```

当Calico进程的Pod运行起来后，以上Configmap会挂载到Pod成为Calico的配置文件。

这时，如果查看Calico Pod的状态，会发现这些Pod都处于Pending状态，原因是当前还没有合适的Node来运行这些Pod。为了便于实验，我们去掉Master节点不能运行用户Pod的限制，让Calico的Pod也能调度到Master节点上运行，如下所示：

```
# kubectl taint nodes --all node-role.kubernetes.io/master-
node/<k8s-master> untainted
```

在默认情况下，由kubeadm安装的Kubernetes集群的Master节点是被打上了taint（污点）的，该taint的作用是使Master节点无法注册成为Kubernetes的Node。这么做是Kubernetes为了避免管理面与用户程序部署在同一个节点上。上述命令就是去掉Master节点的taint。

Calico运行起来后，通过查看运行的Pod会发现已经有两个Pod处于Running状态了：

```
kubectl get pods --namespace=kube-system
NAMESPACE    NAME                                       READY  STATUS   RESTARTS  AGE
kube-system  calico-kube-controllers-6ff88bf6d4-tgtzb   1/1    Running  0         2
m45s
kube-system  calico-node-24h85                          2/2    Running  0         2
m43s
```

Calico Pod通常被放在kube-system namespace，主要包含两种组件。

（1）calico-kube-controllers。整个集群环境只有一个以calico-kube-controllers开头命名的Pod，用于从Kubernetes API Server中读取Network Policy等信息，并对Calico进行相应的配置。

（2）calico-node。在集群的每个节点上都会运行一个以calico-node开头命名的Pod，通过配置iptables实现节点上Pod的出/入（ingress/egress）网络策略。calico-node通常以Kubernetes DaemonSet方式部署运行。

calico-kube-controllers包含以下几个组件：

（1）policy controller：监视Kubneters的NetworkPolicy对象并编程Calico策略。policy controller会把Kubernetes的NetworkPolicy对象同步到Calico的数据库中，因此需要有对Kubernetes API的读权限，以监听NetworkPolicy的增

删改查事件。用户在Kubernetes集群中配置了Pod网络策略后，policy controller就会通知各个节点上的calico-node服务，在主机上刷新相应的iptables规则，完成对Pod间网络的隔离。

（2）namespace controller：监视Kubernetes的Namespace对象并刷新Calico配置文件，它会把Kubernetes的namespace label变化同步到Calico的数据库中。

（3）serviceaccount controller：监视Kubernetes的ServiceAccount对象和刷新Calico配置文件，它会把Kubernetes的ServiceAccount变化同步到Calico的数据库中。

（4）workloadendpoint controller：监视Kubernetes Pod标签的更改并刷新Calico工作负载中的Endpoints配置，它会把Kubernetes的Pod label变化同步到Calico的数据库中。

（5）node controller：监视Kubernetes Node的删除动作并从Calico数据库中删除相应的数据。

总的来说，calico-kube-controllers需要有对Kubernetes NetworkPolicy、Namespace、ServiceAccount、Pod和Node等API的读权限，以便监听这些资源对象的增删改查事件。

Calico与Kubernetes交互的架构图如图4-18所示。

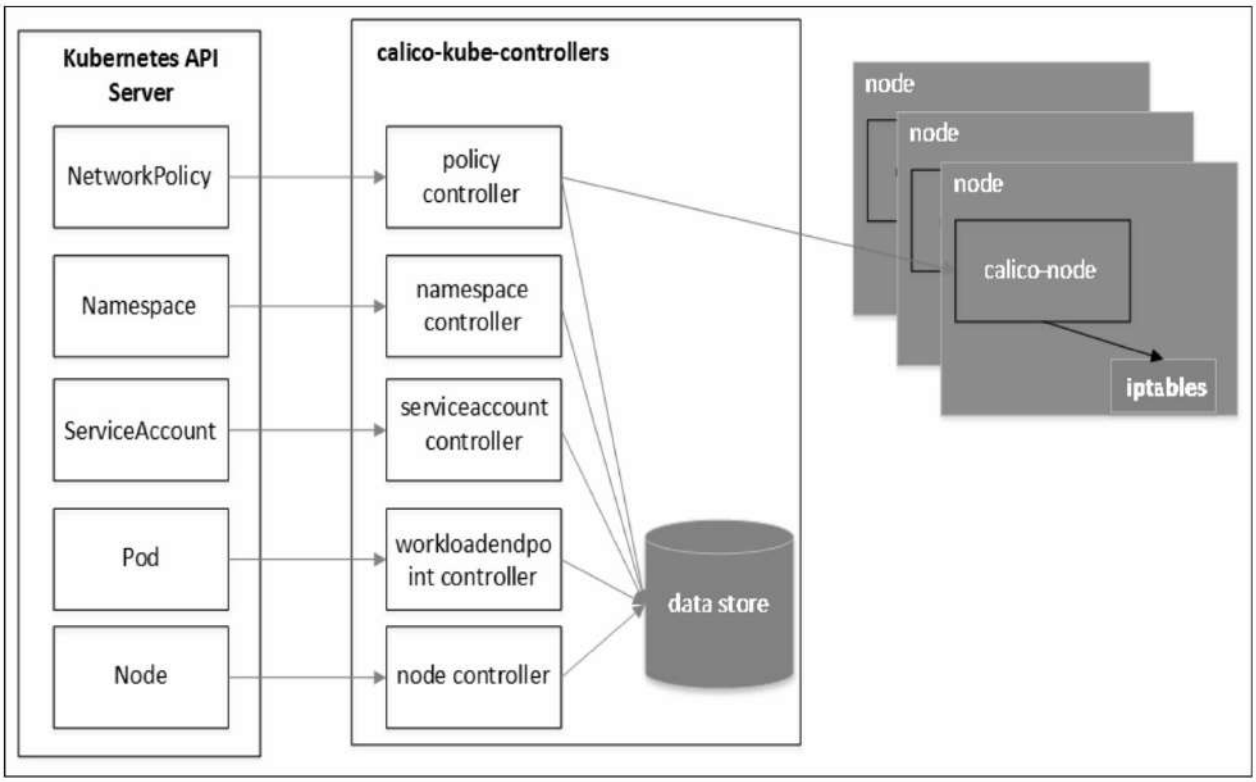


图4-18 Calico与Kubernetes交互的架构图

关于calico-kube-controllers，需要注意如下几点：

（1）除了node controller，其他功能的控制器均是默认启用的。如果你直接使用Calico提供的manifest部署calico-kube-controllers容器，则仍然通过配置ENABLED_CONTROLLERS选项打开node controller控制器。启用node controller控制器，还需要在calico-node DaemonSet的manifest文件中添加一个环境变量CALICO_K8S_NODE_REF，取值为nodeName。

（2）以上所有的控制器都是仅在使用etcd作为Calico后端数据存储时才有效。

（3）为了使calico-kube-controllers不受容器网络访问控制策略的阻止，calico-kube-controllers容器必须使用主机网络，即在Pod的manifest配置hostNetwork:true。

calico-node要用到的主要参数如下。

（1）CALICO_IPV4POOL_CIDR：Calico IPAM的IP地址池，Pod的IP地址将从该池中进行分配。

（2）CALICO_IPV4POOL_IPIP：是否启用ipip模式。启用ipip模式时，Calico将在Node上创建一个名为tunl0的虚拟隧道。使用ipip模式时，设置CALICO_IPV4POOL_IPIP=always；不使用ipip模式时，设置CALICO_IPV4POOL_IPIP=off，此时将使用BGP模式。

（3）FELIX_IPV6SUPPORT：是否启用IPv6。

（4）FELIX_LOGSEVERITYSCREEN：日志级别。

calico-node在正常运行之后，会根据CNI规范默认在/etc/cni/net.d/目录下生成如下文件和目录，并在/opt/cni/bin目录下安装二进制文件calico和calico-ipam，供Kubelet调用。

·10-calico.conf：符合CNI规范的容器网络配置文件，其中type=calico表示该插件的二进制文件名为calico；

·calico-kubeconfig：Calico连接Kubernetes API Server所需的kubeconfig文件；

·calico-tls目录：存放以TLS方式连接etcd的相关文件。

后面章节会更详细地解析Calico底层的实现原理，这里不再赘述。

4.4.2 测试Calico网络策略

集群部署完成之后，我们将结合Calico实践Kubernetes的网络策略。

首先，创建一个测试用的namespace：

```
# kubectl create ns policy-demo
```

然后，在上面的namespace里创建测试用的Pod并暴露为服务：

```
# kubectl run --namespace=policy-demo nginx --replicas=2 --image=nginx
# kubectl expose --namespace=policy-demo deployment nginx --port=80
```

这时，创建一个客户端Pod测试上面的Nginx服务的连通性：

```
# kubectl run --namespace=policy-demo access --rm -ti --image busybox /bin/sh
# wget -q --timeout=5 nginx -O -
```

正常情况下，你会看到Nginx返回的结果。这也说明了默认情况下，Kubernetes的Pod网络是不受限的。

1.启用网络隔离

下面的网络策略将阻止对policy-demo这个namespace下所有Pod的访问。

```
kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: default-deny
  namespace: policy-demo
spec:
  podSelector:
    matchLabels: {}
EOF
```

当我们再次测试网络连通性的时候，会发现无法访问Nginx服务，如下所示：

```
# kubectl run --namespace=policy-demo access --rm -ti --image busybox /bin/sh
# wget -q --timeout=5 nginx -O -
wget: download timed out
```

2.通过网络策略设置白名单

下面，我们将通过Kubernetes网络策略重新允许对Nginx服务的访问。

```
kubectl create -f - <<EOF
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: access-nginx
  namespace: policy-demo
spec:
  podSelector:
    matchLabels:
      run: nginx
  ingress:
    - from:
      - podSelector:
          matchLabels:
            run: access
EOF
```

以上网络策略允许access　Pod访问Nginx服务，并非其他任意来源都可以。为什么呢？通过分析以上NetworkPolicy对象会发现：

·该网络策略约束Label是run:nginx的Pod，即我们Nginx服务对应的后端Pod（kubectl run创建Deployment时自动添加的labels）；

·ingrss.from.podSelector.matchLabels字段的run:access表示允许Label为run:access的Pod访问被限制的Pod。除此之外，其他Pod都无法访问。

让我们来验证上面的分析：

```
# kubectl run --namespace=policy-demo access --rm -ti --image busybox /bin/sh
# wget -q --timeout=5 nginx -O -
```

测试发现返回了Nginx的响应页面。

如果从没有run:access这个Label的Pod发起对Nginx服务的访问，就会出现连接超时的情况，如下所示：

```
# kubectl run --namespace=policy-demo cant-access --rm -ti --image busybox /bin/sh
# wget -q --timeout=5 nginx -O -
wget: download timed out
```

很明显，cant-access这个Pod没有被打上run:access Label，因此访问不通。

第5章　百花齐放：Kubernetes网络插件生态

5.1　从入门到放弃：Docker原生网络的不足

Docker自己的网络方案比较简单，就是每个宿主机上会跑一个非常纯粹的Linux　bridge，这个bridge可以认为是一个二层的交换机，但它的能力有限，只能做一些简单的学习和转发。出网桥的流量会经过iptables，经过NAT，最后通过路由转发在宿主之间进行通信。

当真正用Docker原生的网络模型部署一个比较复杂的业务时，会遇到诸如：容器重启之后IP就变了；每台宿主机会分配固定的网段，因此同一个容器迁到不同宿主机时，除了IP发生变化，网段也会变化，随之而来的网络策略都需要调整等问题。另外，NAT的存在会造成两端在通信时看到对方的地址是不真实的，而且NAT本身也有性能损耗。这些问题都对Docker自身网络方案的应用造成了障碍。

在详细介绍Kubernetes的CNI插件之前，我们可以先对网络中会见到的相关术语做一个整体的了解。不论是阅读本书，还是今后接触其他和CNI有关的内容，了解一些常见术语总是非常有用的。

一些最常见的术语包括：

·第**2**层网络：OSI（Open Systems Interconnections，开放系统互连）网络模型的“数据链路”层。第2层网络会处理网络上两个相邻节点之间的帧传递。第2层网络的一个典型示例是以太网。

·第**3**层网络：OSI网络模型的“网络”层。第3层网络的主要关注点，是在第2层连接之上的主机之间路由数据包。IPv4、IPv6和ICMP是第3层网络协议的示例。

·**VXLAN**：即虚拟可扩展的LAN。首先，VXLAN用于通过在UDP数据包中封装第2层以太网帧帮助实现大型云部署。VXLAN虚拟化与VLAN类似，但提供更大的灵活性和功能（VLAN仅限于4096个网络ID）。VXLAN是一种overlay协议，可在现有网络之上运行。

·**overlay**网络：是建立在现有网络之上的虚拟逻辑网络。overlay网络通常用于在现有网络之上提供有用的抽象，并分离和保护不同的逻辑网络。

·封装：是指在附加层中封装网络数据包以提供其他上下文和信息的过

程。在overlay网络中，封装被用于从虚拟网络转换到底层地址空间，从而能路由到不同的位置（数据包可以被解封装，并继续到其目的地）。

·网状网络：是指每个节点连接到许多其他节点以协作路由，并实现更大连接的网络。网状网络（mesh network）允许通过多个路径进行路由，从而提供更可靠的网络。网状网格的缺点是每个附加节点都会增加大量开销。

·**BGP**：代表“边界网关协议”，用于管理边缘路由器之间数据包的路由方式。BGP通过考虑可用路径、路由规则和特定网络策略等因素，将数据包从一个网络发送到另一个网络。BGP有时被用作容器网络的路由机制，但不会用在overlay网络中。

5.2 CNI标准的胜出：从此江湖没有CNM

CNI即容器网络接口（Container Network Interface）。Kubernetes采用CNI而非CNM（容器网络模型），这背后有很长的一段故事，核心的原因就是CNI对开发者的约束更少，更开放，不依赖于Docker工具，而CNM对Docker有非常强的依赖，无法作为通用的容器网络标准。

在CNI标准中，网络插件是独立的可执行文件，被上层的容器管理平台调用。网络插件只有两件事情要做：把容器加入网络或把容器从网络中删除。调用插件的配置通过两种方式传递：环境变量和标准输入。

我们经常戏称CNI很简单，只需要：

·1个配置文件，配置文件描述插件的版本、名称、描述等基本信息；

·1个可执行文件，可执行文件就是CNI插件本身会在容器需要建立网络和需要销毁容器时被调用；

·读取6个环境变量，获得需要执行的操作、目标网络Namespace、容器的网卡必要信息；

·接受1个命令行参数，同样用于获得需要执行的操作、目标网络Namespace、容器的网卡必要信息；

·实现2个操作（ADD/DEL）。

图5-1说明了Kubernetes使用CNI网络插件的工作流程：

·Kubernetes调用CRI创建pause容器，生成对应的network namespace；

·调用网络driver（因为配置的是CNI，所以会调用CNI的相关代码）；

·CNI driver根据配置调用具体的CNI插件；

·CNI插件给pause容器配置正确的网络，Pod中的其他容器都是用pause容器的网络栈。

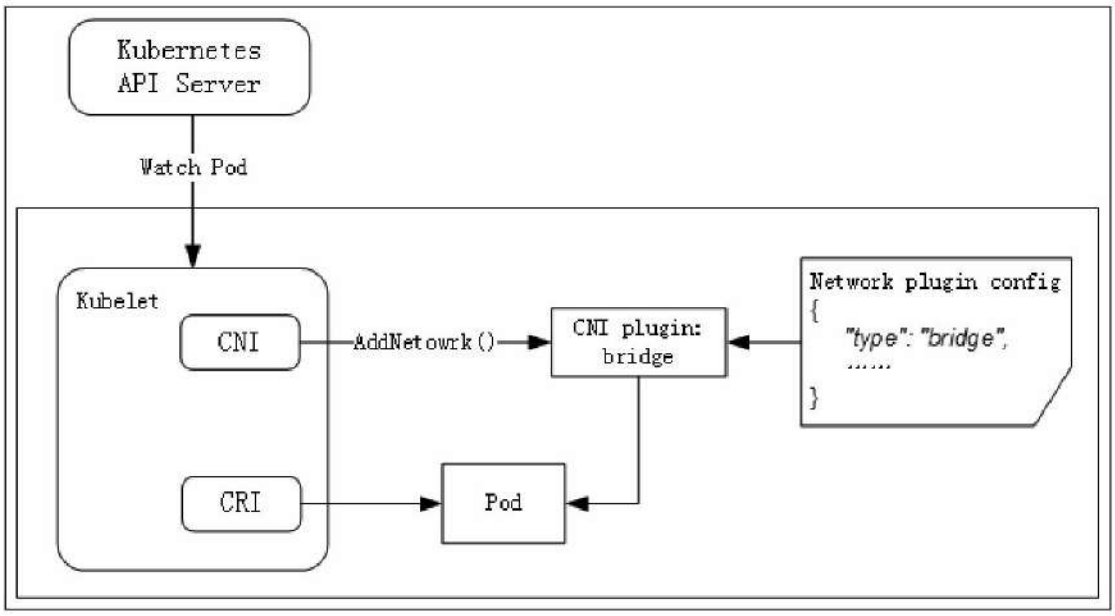


图5-1　Kubernetes使用CNI网络插件的工作流程

CNI的初衷是创建一个框架，用于在配置或销毁容器时动态配置适当的网络配置和资源。CNI规范概括了用于配制网络的插件接口，这个接口可以让容器运行时与插件进行协调。CNI插件负责为每个容器分配IP地址，为容器接口配置和管理IP地址，以及多主机连接相关的功能。容器运行时（runtime）会调用网络插件，从而在容器启动时分配IP地址并配置网络，并在删除容器时再次调用它以清理这些资源。

容器运行时决定了容器应该加入哪个网络及它需要调用哪个插件。然后，插件会将网络接口添加到容器网络命名空间中（例如，作为一个veth pair的一端）。接着，它会在主机上进行相关配置（例如，将veth的其他部分连接到网桥上）。最后，CNI会通过调用单独的IPAM（IP地址管理）插件分配IP地址并设置路由。

在Kubernetes中，Kubelet可以在适当的时间调用它找到的插件，为通过Kubelet启动的Pod进行自动的网络配置。

5.2.1　CNI与CNM的转换

CNI和CNM并非完全不可调和的两个模型。例如，Calico项目就是直接支持两种接口模型的。

从模型的角度看，CNI中的container应与CNM中的sandbox概念一致，CNI中的network与CNM中的network一致。在CNI中，CNM中的Endpoint被隐含在了ADD/DELETE的操作中。CNI接口更简洁，把更多的工作托管给了容器的管理者和网络的管理者。换句话说，CNI的ADD/DELETE接口其实只是实现了docker network connect和docker network disconnect两个命令。

Kubernetes/contrib项目提供了一种从CNI向CNM转化的过程。其中的原理很简单，就是直接通过shell脚本执行docker network connect和docker network disconnect命令，实现从CNI到CNM的转化，感兴趣的读者可以自行查看，这里不再赘述。

5.2.2 CNI的工作原理

毕竟CNI只是一个接口标准，看了上面几个例子后可能有读者会好奇，CNI到底是如何工作的，即如何与网络插件集成的？

下面我们将基于CNI自带的docker-run.sh和exec-plugins.sh详细说明CNI与网络插件集成的原理。

```
#!/usr/bin/env bash

# Run a docker container with network namespace set up by the
# CNI plugins.

# Example usage: ./docker-run.sh --rm busybox /sbin/ifconfig

contid=$(docker run -d --net=none busybox:latest /bin/sleep 10000000)
pid=$(docker inspect -f '{{ .State.Pid }}' $contid)
netnspath=/proc/$pid/ns/net

./exec-plugins.sh add $contid $netnspath

function cleanup() {
    ./exec-plugins.sh del $contid $netnspath
    docker rm -f $contid >/dev/null
}
trap cleanup EXIT

docker run --net=container:$contid $@
```

exec-plugins.sh如下所示：

```
#!/usr/bin/env bash

if [[ ${DEBUG} -gt 0 ]]; then set -x; fi

NETCONFPATH=${NETCONFPATH-/etc/cni/net.d}

function exec_plugins() {
    i=0
    contid=$2
    netns=$3
    export CNI_COMMAND=$(echo $1 | tr '[:lower:]' '[:upper:]')
    export PATH=$CNI_PATH:$PATH
    export CNI_CONTAINERID=$contid
    export CNI_NETNS=$netns

    for netconf in $(echo $NETCONFPATH/*.conf | sort); do
        name=$(jq -r '.name' <$netconf)
        plugin=$(jq -r '.type' <$netconf)
        export CNI_IFNAME=$(printf eth%d $i)

        res=$($plugin <$netconf)
        if [ $? -ne 0 ]; then
            errmsg=$(echo $res | jq -r '.msg')
            if [ -z "$errmsg" ]; then
                errmsg=$res
            fi

            echo "${name} : error executing $CNI_COMMAND: $errmsg"
            exit 1
        elif [[ ${DEBUG} -gt 0 ]]; then
            echo ${res} | jq -r .
        fi

        let "i=i+1"
    done
}
```

```
if [ $# -ne 3 ]; then
    echo "Usage: $0 add|del CONTAINER-ID NETNS-PATH"
    echo "  Adds or deletes the container specified by NETNS-PATH to the networks"
    echo "  specified in \$NETCONFPATH directory"
    exit 1
fi

exec_plugins $1 $2 $3
```

简单说就是检查CNI配置文件目录（$NETCONFPATH），读取配置（*.conf文件），然后把配置输入给配置的type字段的插件。输出如下所示：

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.22.0.5  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::4ce9:51ff:fee1:2f97  prefixlen 64  scopeid 0x20<link>
        ether 4e:e9:51:e1:2f:97  txqueuelen 0  (Ethernet)
        RX packets 1  bytes 90 (90.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1  bytes 90 (90.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

其中：

```
contid=456a387326754f9
netnspath=/var/run/netns/456a387326754f9
```

等价于：

```
# export CNI_COMMAND=ADD
```

```
# export CNI_NETNS=/var/run/netns/456a387326754f9
# export CNI_CONTAINERID=456a387326754f9
# export CNI_IFNAME=eth0
# $CNI_PATH/bridge </etc/cni/net.d/10-mynet.conf
```

如果要删除以上操作创建的容器的网络，只需执行：

```
# CNI_PATH=$CNI_PATH ./exec-plugins.sh del 456a387326754f9 /var/run/netns/456
a387326754f9
```

等价于：

```
# export CNI_COMMAND=DEL
# export CNI_NETNS=/var/run/netns/456a387326754f9
# export CNI_CONTAINERID=456a387326754f9
# export CNI_IFNAME=eth0

# $CNI_PATH/bridge </etc/cni/net.d/10-mynet.conf
```

以上过程对所有插件都有效，除了loopback、bridge、ptp这些CNI的默认插件，还支持flannel和Calico等第三方插件。

下面就以flannel为例介绍CNI是如何与第三方插件集成的。首先，安装好flannel（具体步骤后面的章节会给出），启动flannel后查看flannel的配置，如下所示：

```
# cat /run/flannel/subnet.env*
FLANNEL_NETWORK=192.168.0.0/16
FLANNEL_SUBNET=192.168.22.129/26
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

由此可见，通过flannel创建的容器网段应该在192.168.0.0/16。在由CNI创建容器网络之前，先配置如何使用flannel网络插件。

```
# cat >/etc/cni/net.d/10-mynet.conf <<EOF
{
  "name": "mynet",
  "type": "flannel"
}
EOF
```

如果是其他网络插件，则更改type字段再加上对应的配置即可。

```
# CNI_PATH=$CNI_PATH ./docker-run.sh ifconfig

contid=6e3b681a422f5174
netnspath=/var/run/netns/6e3b681a422f5174

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
inet 192.168.22.130  netmask 255.255.255.192  broadcast 0.0.0.0
inet6 fe80::d810:64ff:fe2c:831d  prefixlen 64  scopeid 0x20<link>
ether da:10:64:2c:83:1d  txqueuelen 0  (Ethernet)
RX packets 2  bytes 180 (180.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 1  bytes 90 (90.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
inet 127.0.0.1  netmask 255.0.0.0
inet6 ::1  prefixlen 128  scopeid 0x10<host>
loop  txqueuelen 0  (Local Loopback)
RX packets 0  bytes 0 (0.0 B)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 0  bytes 0 (0.0 B)
TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

删除网络也和上面的用法一样：

```
delete network
# CNI_PATH=$CNI_PATH ./exec-plugins.sh del 6e3b681a422f5174 /var/run/netns/6
e3b681a422f5174
```

由此可见，CNI的最大价值在于提供了一致的容器网络操作界面，不论

是什么网络插件都使用一致的API，提高了网络配置的自动化程度和一致性的体验。

5.2.3 为什么Kubernetes不使用Libnetwork

Libnetwork是2015年5月1日由Docker发布的容器网络管理项目，实现了Docker定义的CNM，目标是为应用程序提供一致的编程接口及网络抽象。CNM历史较CNI悠久，而且Libnetwork也是使用Go语言编写的。

在Kubernetes 1.0版本之前就已经具有基本的网络插件体系了。与此同时，Docker也将Libnetwork和CNM纳入网络系统中使用。既然Docker已经发布并支持网络插件Libnetwork，而且大部分供应商肯定会支持Docker的网络插件，为什么后来发布的Kubernetes没有使用Docker的Libnetwork管理容器网络呢？

Tim Hockin，Kubernetes的创始人之一，也是Kubernetes SIG Network的Leader，曾旗帜鲜明地指出：“Kubernetes永远不会使用Docker的Libnetwork！”。这其中存在一些技术方面的原因。下面我们单纯从技术的角度出发，探讨Kubernetes为什么不使用Libnetwork。

> 注：以下很多观点出自Tim Hockin在Kubernetes blog发表的博文*why kubernetes does not use libnetwork*。

深入讨论之前，我们要知道：Kubernetes是一个支持多容器的运行环境，而Docker只是其中一个容器而已。每一个运行环境都会配置网络环境，所以当人们问“Kubernetes会支持CNM吗？”时，他们真正的意思是“Kubernetes是否在Docker运行时下支持CNM？”。当然，我们希望同一个网络插件支持所有的运行环境，但这并不是一个绝对目标。

Kubernetes确实没有在Docker中采用CNM/Libnetwork。事实上，早期的Kubernetes开发者一直在研究能不能用CoreOS提出的APP Container（appc）标准中的CNI替代它。为什么呢？这里有一些技术和非技术上的原因。

在Docker的网络驱动设计中，预先做了一些兼容的基本假设，这给Kubernetes带来了不少的问题。例如，在Docker中有一个“本地”驱动和“全局”驱动的概念。本地驱动（例如bridge）是以一台机器为中心，不能实现跨机器节点的协作。全局驱动依赖libkv（一个抽象的键值对存数对象）实现跨机器节点的协作（例如overlay）。libkv是一个非常底层（语义层面的）的键值对存储接口。为了能让Docker中类似overlay的全局驱动在Kubernetes集群运行，我们还需要集群系统管理员运行consul、etcd或者ZooKeeper，或者不得不在Kubernetes中提供自己对于libkv的实现。

自己实现libkv听起来更有吸引力。然而，libkv的接口太过于底层，而

且它的架构模式也是Docker内部的量身定制版，我们要么直接暴露底层的key-value存储，要么提供keyvalue语义接口（在一个key-value系统上实现结构存储的API）。就性能、可扩展性和安全性而言，以上两种方案都不太合适。我们使用Docker网络的目标是简化操作，如果那样做，很明显会使整个系统变得更复杂。

对于那些想运行Docker的全局驱动并有能力配置Docker的用户来说，Docker的网络应该是“可行的”。对Kubernetes来说，我们不希望介入或影响Docker的配置步骤，并且不论Kubernetes这个项目今后如何发展，这一点应该不会改变。Docker的全局驱动对用户和开发者来说增加了多余的负担，并且我们不会用它作为默认的网络选项，这也意味着使用Docker插件没什么价值。

Docker的网络模型在设计上还做了很多和Kubernetes不兼容的假设。在Docker 1.8和1.9版本中，它在实现“Discovery”时有一个根本性的设计缺陷，导致容器中的/etc/hosts文件被随意改写甚至破坏（详见docker issue#17190），而且用户还不能轻易关闭“服务发现”这个功能。对Kubernetes来说，把命名寻址绑定在容器层面，并不是一个正确的设计——Kubernetes已经自己定义了一套Service命名、寻址、绑定的概念和规则，并且也有了自己的DNS架构和服务（构建在SkyDNS上）。所以，捆绑一个DNS服务器这样的方案并不能满足Kubernetes的需求，而且它还有可能无法被关闭。

除了“本地/全局”驱动这样的区分，Docker还定义了“进程内”和“进程外”（“远程”）插件。虽然我们可以研究是否可以绕过Libnework本身（这样就能避开之前所述的那些问题）直接使用“远程”插件，但不幸的是，这意味着我们将失去使用Docker的那些“进程内”插件的可能，特别是“bridge”和“overlay”这样的全局插件。这又令使用Libnetwork失去了意义。

值得一提的是，CNI和Kubernetes在设计哲学上非常一致。它远比CNM简单，不需要守护进程，而且是跨平台的。跨平台意味着同一个网络配置可以在多个运行环境下使用（例如Docker、rkt和Hyper）。这也非常符合UNIX的设计哲学：做好一件事。

另外，包装CNI模块来实现一个自定义的模块是非常简单的，通过一个简单的shell脚本就可以完成。相反，CNM就复杂多了。因此，Kubernetes认为CNI更适合快速开发和迭代。早期的实验尝试证明，Kubernetes可以利用CNI插件替代几乎所有Kubelet中硬编码的网络逻辑。

Kubernetes曾经也尝试为Docker实现了一个“bridge”CNM驱动来使用CNI的驱动。结果表明这使问题更复杂。首先，CNM和CNI的模型非常不同，没有一个方法能把它们很好的兼容。其次，还是存在之前提到的“全局”和“本地”及key-value的问题。假设这是个本地驱动，那么我们仍旧要从Kubernetes

中得到相应的逻辑网络的信息。

最让Kubernetes这些外部平台头疼的是，Docker网络驱动的设计不具备良好的开放性和可扩展性。例如，Docker网络驱动使用了Docker内部分配的一个ID，而不是一个通用的网络名字来指向一个容器所属的网络。这样的设计导致一个被定义在外部系统中（例如Kubernetes）的网络很难被Docker的网络驱动理解和识别。

Kubernetes和其他网络提供商甚至将这些问题和其他相关问题都汇报给了Docker的开发人员。尽管这些问题给非Docker的第三方系统带来了很多困扰，它们通常被以“设计就是如此”为理由关闭，例如libnetwork issue#139、libnetwork issue#486、libnetwork issue#514、libnetwork issue#865、docker issue#18864等。通过这些举动，Kubernetes感觉到Docker对于一些建议的态度不够开放，因为这些建议可能会分散其主要精力，或者降低其对项目的控制。

考虑到Kubernetes和Docker项目的独立性等种种原因，促使Kubernetes选择CNI作为网络模型。这将带来诸如：docker inspect命令显示不了Pod的IP地址，直接被Docker启动的容器可能无法和被Kubernetes启动的容器通信等问题。

但Kubernetes必须做一个权衡：选择CNI，使Kubernetes网络配置更简单、灵活并且不需要额外的配置（例如，配置Docker使用Kubernetes或其他网络插件的网桥）。

5.3 Kubernetes网络插件鼻祖flannel

flannel可以为容器提供跨节点网络服务，其模型为集群内所有容器使用一个网络，然后在每个主机上从该网络中划分一个子网。flannel为主机上的容器创建网络时，从子网中划分一个IP给容器。根据Kubernetes的模型，为每个Pod提供一个IP，flannel的模型正好与之契合。难得的是，flannel安装方便且简单易用。

flannel几乎是最早的跨节点容器通信解决方案，如果仔细观察会发现其他网络插件都有flannel的身影。在笔者看来，其他的方案都可以看作flannel的某种改进版！因此，我们不妨先以flannel为例，介绍这些网络插件是如何解决容器网络问题的。说到容器跨节点访问，主要有以下几方面的技术挑战：

容器**IP**地址的重复问题。由于Docker等容器工具只是利用Linux内核的network namespace实现了网络隔离，各个节点上的容器IP地址是在所属节点上自动分配的，从全局来看，这种局部地址就像是不同小区里的门牌号，一旦拿到一个更大的范围上看，就可能是重复的。为了解决这个问题，flannel设计了一种全局的网络地址分配机制，即使用etcd存储网段和节点之间的关系，然后flannel配置各个节点上的Docker（或其他容器工具），只在分配到当前节点的网段里选择容器IP地址。这样就确保了IP地址分配的全局唯一性。

容器**IP**地址路由问题。是不是地址不重复网络就可以联通了呢？这里还有一个问题，因为通常虚拟网络的IP和MAC地址在物理网络上是不认识的，所以数据包即使被发送到网络中，也会因为无法进行路由而被丢掉。虽然地址唯一了，但是依然无法实现真正的网络通信。flannel早期用得比较多的一种方式是overlay网络，其实就是个隧道网络。后来，flannel也开发了另外几种处理方法，对应于flannel的几种网络模式。

在overlay网络下，所有被发送到网络中的数据包会被添加上额外的包头封装。这些包头里通常包含了主机本身的IP地址，因为只有主机的IP地址是原本就可以在网络里路由传播的。根据不同的封包方式，flannel提供了UDP和VXLAN两种传输方法。UDP封包使用了flannel自定义的一种包头协议，数据是在Linux的用户态进行封包和解包的，因此当数据进入主机后，需要经历两次内核态到用户态的转换。VXLAN封包采用的是内置在Linux内核里的标准协议，因此虽然它的封包结构比UDP模式复杂，但所有的数据装、解包过程均在内核中完成，实际的传输速度要比UDP模式快许多。比较不幸的是，在flannel开始流行时，大概2014年，主流的Linux系统还是Ubuntu 14.04

或者CentOS 6.x，这些发行版的内核比较低，没有包含VXLAN的内核模块。因此，多数人在开始接触flannel时，都只能使用它的UDP模式，使flannel一不小心落得了一个“速度慢”的名声。overlay是第一种解决容器网络地址路由的方法。

路由是第二种解决容器网络地址路由的方法，在flannel中就是Host-Gateway模式。从上面的讨论我们得知，容器网络无法进行路由是因为宿主机之间没有路由信息，但flannel是知道这个信息的，因此一个直观的想法是能不能把这个信息告诉网络上的节点呢？在Host-Gateway模式下，flannel通过在各个节点上运行的agent将容器网络的路由信息刷到主机的路由表上，这样一来，所有的主机就都有整个容器网络的路由数据了。Host-Gateway的方式没有引入overlay额外封包和解包操作，完全是普通的网络路由机制，通信效率与裸机直连相差无几。事实上，flannel的Gateway模式的性能甚至要比Calico好。然而，由于flannel只能修改各个主机的路由表，一旦主机之间隔了其他路由设备，比如三层路由器，这个包就会在路由设备上被丢掉。这样一来，Host-Gateway的模式就只能用于二层直接可达的网络，由于广播风暴的问题，这种网络通常是比较小规模的。近年来，也出现了一些专门的设备能够构建出大规模的二层网络（这就是我们经常听到的“大二层”网络）。

5.3.1 flannel简介

在Kubernetes的网络模型中，假设了每个物理节点应该具备一段“属于同一个内网IP段内”的“专用的子网IP”。例如：

```
节点A: 10.0.1.0/24
节点B: 10.0.2.0/24
节点C: 10.0.3.0/24
```

在默认的Docker配置中，每个节点上的Docker服务会分别负责所在节点容器的IP分配。这样导致的一个问题是，不同节点上的容器可能获得相同的IP地址。

flannel最早由CoreOS开发，它是容器编排系统中最成熟的网络插件示例之一。随着CNI概念的兴起，flannel也是最早实现CNI标准的网络插件（CNI标准也是由CoreOS提出的）。flannel的功能非常简单明确，解决的就是上文我们提到的容器跨节点访问的两个问题。

flannel的设计目的是为集群中的所有节点重新规划IP地址的使用规则，从而使得集群中的不同节点主机创建的容器都具有全集群“唯一”且“可路由的IP地址”，并让属于不同节点上的容器能够直接通过内网IP通信。那么节

点是如何知道哪些IP可用，哪些不可用，即其他节点已经使用了哪些网段的呢？flannel用到了etcd的分布式协同功能。

flannel的架构如图5-2所示。

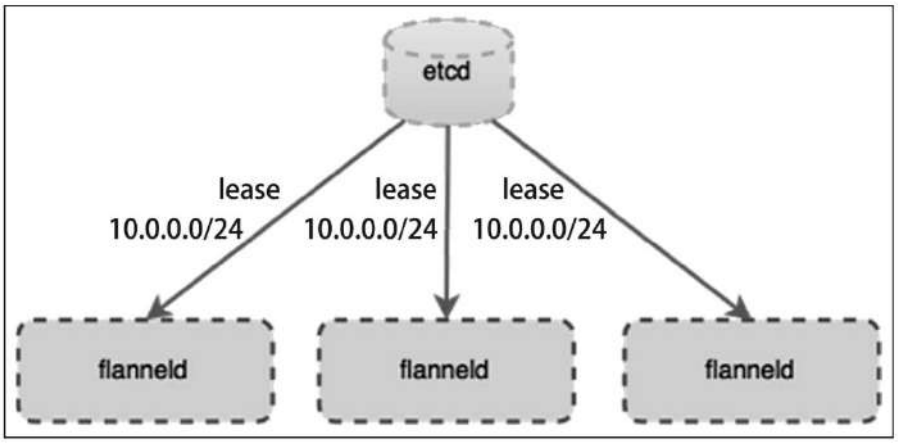


图5-2　flannel的架构

flannel在架构上分为管理面和数据面。管理面主要包含一个etcd，用于协调各个节点上容器分配的网段，数据面即在每个节点上运行一个flanneld进程。与其他网络方案不同的是，flannel采用的是no　server架构，即不存在所谓的控制节点，简化了flannel的部署与运维。

集群内所有flannel节点共享一个大的容器地址段（在我们的例子中就是10.0.0.0　16），flanneld一启动便会观察etcd，从etcd得知其他节点上的容器已占用的网段信息，然后向etcd申请该节点可用的IP地址段（在大网段中划分一个，例如10.0.2.0/24），并把该网段和主机IP地址等信息都记录在etcd中。

flannel通过etcd分配了每个节点可用的IP地址段后，修改了Docker的启动参数，例如--bip=172.17.18.1/24限制了所在节点容器获得的IP范围，以确保每个节点上的Docker会使用不同的IP地址段。需要注意的是，这个IP范围是由flannel自动分配的，由flannel通过保存在etcd服务中的记录确保它们不会重复，无须用户手动干预。

flannel的底层实现实质上是一种overlay网络（除了Host-Gateway模式），即把某一协议的数据包封装在另一种网络协议中进行路由转发。

flannel（v0.7+）目前已经支持的底层实现有：

·UDP；

·VXLAN；

·Alloc；

·Host-Gateway；

·AWS VPC；

·GCE路由。

其中，性能最好的应该是Host-Gateway。AWS VPC和GCE路由都需要L2网络的支持，如果没有接入云服务，通常维护成本也比较高。Alloc只为本机创建子网，多个主机上的子网之间不能直接通信。

接下来，我们将详细分析flannel的各种数据转发的实现机制。

最后，flannel在封包的时候是怎么知道目的容器所在主机的IP地址的呢？flanneld会观察etcd的数据，因此在其他节点向etcd更新网段和主机IP信息时，etcd就感知到了，在向其他主机上的容器转发网络包时，用对方容器所在主机的IP进行封包，然后将数据发往对应主机上的flanneld，再交由其转发给目的容器。

5.3.2 flannel安装配置

分别从flannel和etcd的GitHub项目地址获取release包后，先启动etcd，待etcd启动后就可以配置flannel了。

flannel的配置信息全部存在etcd里。假设某个flannel集群的容器网段是172.17.0.0/16，那么，可以通过下面这条命令往etcd里写入配置信息，指定flannel的可用IP地址段：

```
# etcdctl set /coreos.com/network/config '{ "Network": "172.17.0.0/16" }'
```

然后，在每个节点分别启动守护进程flanneld：

```
flanneld &
```

需要注意的是，flanneld要先于Docker启动。flanneld启动时主要做了以下几个动作：

·从etcd中获取network（大网）的配置信息；

·划分subnet（子网），并在etcd中进行注册；

·将子网信息记录到flannel维护的/run/flannel/subnet.env文件中；

·将subnet.env转写成一个Docker的环境变量文件/run/flannel/docker。

一个subnet.env文件可以长成下面这样：

```
# cat /run/flannel/subnet.env
Flannel_NETWORK=10.0.0.0/16
Flannel_SUBNET=10.0.34.1/24
Flannel_MTU=1472
Flannel_IPMASQ=false
```

对应的/run/flannel/docker文件可以如下所示：

```
DOCKER_OPT_BIP="--bip=10.0.34.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=true"
DOCKER_OPT_MTU="--mtu=1472"
DOCKER_NETWORK_OPTIONS="--bip=10.0.34.1/24 --ip-masq=true --mtu=1472"
```

接下来，启动Docker daemon，使用/run/flannel/docker中的变量作为启动参数，例如：

```
# /usr/bin/docker-current daemon --exec-opt native.cgroupdriver=systemd --selinux-
enabled --log-driver=journald --bip=10.0.100.1/24 --ip-masq=true --mtu=1472
```

因为配置了bip，所以创建出来的容器会使用该网段的IP，即容器其实还是使用Docker的bridge网络模式。

如果你的环境中是先启动Docker后启动flannel，那么在准备好以上配置后还需要重启一次Docker daemon让配置生效。

到此，整个flannel集群也就正常运行了。现在在两个节点分别启动一个Docker容器，它们之间已经通过IP地址直接相互通信了。

最后，前面反复提到过flannel有一个保存在etcd的路由表，用户可以通过etcdctl命令在etcd中找到这些路由记录，如下所示：

```
# etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/172.17.18.0-24
/coreos.com/network/subnets/172.17.19.0-24
/coreos.com/network/subnets/172.17.20.0-24
```

以上就是整个flannel集群的子网信息。

```
# etcdctl get /coreos.com/network/subnets/172.17.18.0-24
{"PublicIP":"192.168.14.97"}
# etcdctl get /coreos.com/network/subnets/172.17.19.0-24
```

```
{"PublicIP":"192.168.14.98"}
# etcdctl get /coreos.com/network/subnets/172.17.20.0-24
{"PublicIP":"192.168.14.100"}
```

以上就是每个节点容器子网与节点公共IP地址信息的映射关系。

使用**Kubernetes**安装**flannel**

由于flannel没有Master和slave之分，每个节点上都安装一个agent，即flanneld。我们可以使用Kubernetes的DaemonSet部署flannel以达到每个节点部署一个flanneld实例的目的。

我们在每一个flannel的Pod中都运行了一个flanneld进程，且flanneld的配置文件以ConfigMap的形式挂载到容器内的/etc/kube-flannel/目录，供flanneld使用，配置如下所示：

```
# kubectl get configmap -n kube-system -o yaml kube-flannel-cfg
apiVersion: v1
data:
cni-conf.json: |
{
  "name": "cbr0",
  "type": "flannel",
  "delegate": {
    "isDefaultGateway": true
  }
}
net-conf.json: |
{
  "Network": "10.244.0.0/16",
  "Backend": {
    "Type": "udp"
  }
}
kind: ConfigMap
metadata:
  labels:
    app: flannel
    tier: node
```

```
  name: kube-flannel-cfg
  namespace: kube-system
...
```

从以上配置文件中我们发现了一个关键字段，即Backend.Type字段。在我们这个例子中，Backend.Type的值是udp，即flannel将采用UDP的封包模式。在5.3.3节中，我们将详细介绍flannel的各种backend。

5.3.3 flannel backend详解

flannel通过在每一个节点上启动一个叫flanneld的进程，负责每一个节点上的子网划分，并将相关的配置信息（如各个节点的子网网段、外部IP等）保存到etcd中，而具体的网络包转发交给具体的backend实现。

flanneld可以在启动时通过配置文件指定不同的backend进行网络通信，目前比较成熟的backend有UDP、VXLAN和Host Gateway三种，也已经有诸如AWS、GCE这些还在实验阶段的backend。目前，VXLAN是官方最推崇的一种backend实现方式；Host Gateway一般用于对网络性能要求比较高的场景，但需要基础网络架构的支持；UDP则用于测试及一些比较老的不支持VXLAN的Linux内核。

这里只深入研究最成熟也最通用的三种backend的网络通信实现原理，即：

·UDP；

·VXLAN；

·Host Gateway。

1.UDP

UDP模式相对容易理解，故先采用UDP这种backend模式进行举例说明，然后延伸到其他模式。采用UDP模式时需要在flanneld的配置文件中指定Backend.Type为UDP，可以通过直接修改flanneld的ConfigMap的方式实现，配置过程类似于上文的例子，这里不再赘述。

当采用UDP模式时，flanneld进程在启动时会通过打开/dev/net/tun的方式生成一个tun设备。我们在第1章已经详细介绍过tun设备，tun设备可以简单理解为Linux中提供的一种内核网络与用户空间（应用程序）通信的机制，即应用可以通过直接读写tun设备的方式收发RAW IP包。

flanneld进程启动后，通过ip addr命令可以发现节点中已经多了一个叫flannel0的网络接口：

```
# ip addr
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen
1000
    link/ether 08:00:27:b7:7e:f3 brd ff:ff:ff:ff:ff:ff
    inet 10.8.65.66/24 brd 10.8.65.255 scope global dynamic enp0s3
       valid_lft 67134sec preferred_lft 67134sec
    inet6 fe80::a00:27ff:feb7:7ef3/64 scope link
       valid_lft forever preferred_lft forever
...
5: flannel0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1472 qdisc pfifo_fast
state UNKNOWN qlen 500
link/none
inet 10.244.0.0/16 scope global flannel0
   valid_lft forever preferred_lft forever
inet6 fe80::969a:a8eb:e4da:308b/64 scope link flags 800
   valid_lft forever preferred_lft forever
```

通过ip-d link show flannel0可以看到这是一个tun设备：

```
# ip -d link show flannel0
5: flannel0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1472 qdisc pfifo_fast
state UNKNOWN mode DEFAULT qlen 500
link/none  promiscuity 0
tun
```

细心的读者可能已经发现了，flannel0网络接口上的MTU是1472，相比宿主机网络接口eth0少了28个字节，为什么呢？请看下文分析。

通过netstat-ulnp命令可以看到此时flanneld进程监听在8285端口：

```
# netstat -ulnp | grep flanneld
udp        0      0 172.16.130.140:8285     0.0.0.0:*
```

flannel UDP模式本机通信实践。现在，有3个容器分别是A、B和C，这3个容器的IP地址如表5-1所示。

表5-1　3个容器的IP地址

容器	IP
A	4.0.100.3
B	4.0.100.5
C	4.0.32.3

接下来，我们来看宿主机host上的路由信息，如下所示：

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.8.65.1       0.0.0.0         UG    100    0        0 eth0
4.0.0.0         0.0.0.0         255.255.0.0     U     0      0        0 flannel0
4.0.100.0       0.0.0.0         255.255.255.0   U     0      0        0 docker0
10.8.64.10      10.8.65.1       255.255.255.255 UGH   100    0        0 eth0
10.8.65.0       0.0.0.0         255.255.255.0   U     100    0        0 eth0
```

当容器A发送到同一个subnet的容器B时，因为二者处于同一个子网，所以容器A和B位于同一个宿主机host上，而容器A和B也均桥接在docker0上。

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02422e5ecd90       no              veth2d1c8
                                                        veth91606
```

借助网桥docker0，即可实现同一个主机上容器A和B的直接通信。

注：较早版本的flannel是直接复用Docker创建的docker0网桥，后面版本的flannel将使用CNI创建自己的cni0网桥。不管是docker0还是cni0，本质都是Linux网桥，功能也一样。

那么，位于不同宿主机的容器A和C如何通信呢？这时就要用到上文提到的flannel0网卡。

flannel UDP模式跨主机通信实践。容器跨节点通信实现流程：假设在节点A上有容器A（10.244.1.96），在节点B上有容器B（10.244.2.194）。此时，容器A向容器发送一个ICMP请求报文（ping），我们来逐步分析ICMP报文从容器A到达容器B的整个过程，如图5-3所示。

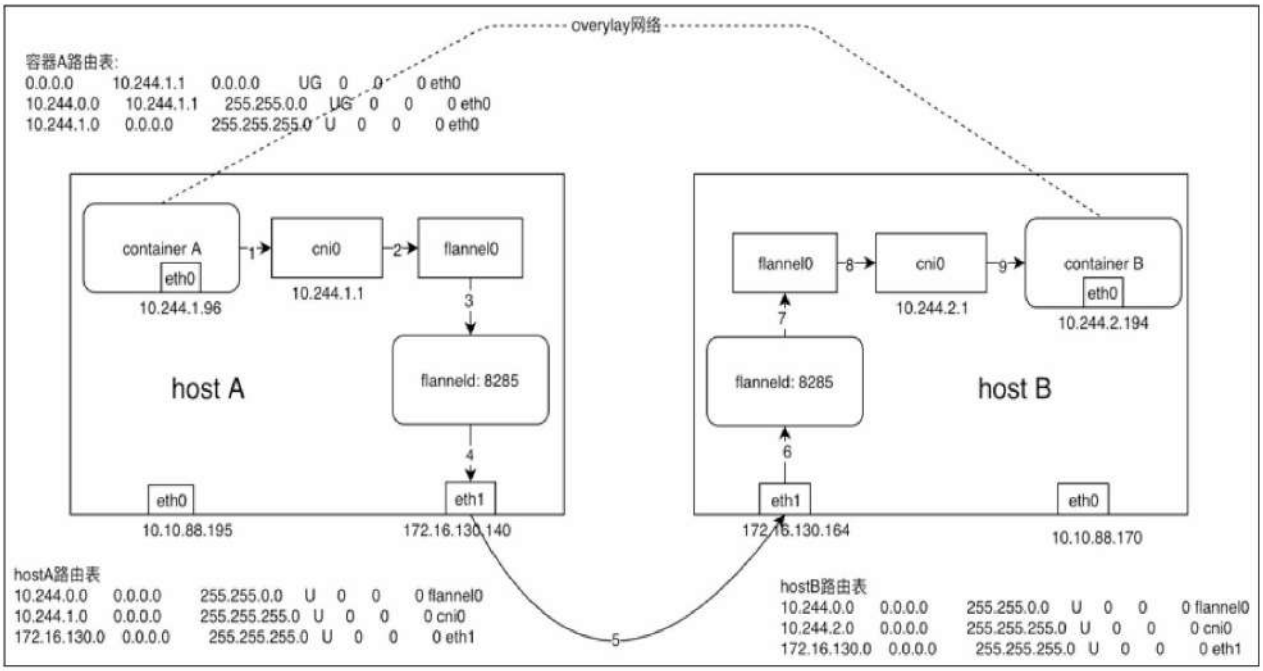


图5-3　flannel UDP模式数据路径

（1）容器A发出ICMP请求报文，通过IP封装后的形式为10.244.1.96（源）→10.244.2.194（目的）。此时，通过容器A内的路由表匹配到应该将IP包发送到网关10.244.1.1（cni0网桥）。

此时的ICMP报文以太网帧格式如图5-4所示。

图5-4　ICMP报文以太网帧格式

（2）到达cni0的IP包目的地IP　10.244.2.194，匹配到节点A上第一条路由规则（10.244.0.0），内核通过查本机路由表知道应该将RAW　IP包发送给flannel0接口。

（3）flannel0为tun设备，发送给flannel0接口的RAW　IP包（无MAC信息）将被flanneld进程接收，flanneld进程接收RAW　IP包后在原有的基础上进行UDP封包，UDP封包的形式为172.16.130.140:｛系统管理的随机端口｝→172.16.130.164:8285。

注：172.16.130.164是10.244.2.194这个目的容器所在宿主机的IP地址，flanneld通过查询etcd很容易得到，8285是flanneld监听端口。

（4）flanneld将封装好的UDP报文经eth0发出，从这里可以看出网络包在通过eth0发出前先是加上了UDP头（8个字节），再加上IP头（20个字节）进行封装，这也是flannel0的MTU要比eth0的MTU小28个字节的原因，即防止封包后的以太网帧超过eth0的MTU，而在经过eth0时被丢弃。

此时，完整的封包后的ICMP以太网帧格式如图5-5所示。

图5-5　封包后的ICMP以太网帧格式

（5）网络包经过主机网络从节点A到达节点B。

（6）主机B收到UDP报文后，Linux内核通过UDP端口号8285将包交给正在监听的flanneld。

（7）运行在host　B中的flanneld将UDP包解封包后得到RAW　IP包：10.244.1.96→10.244.2.194。

（8）解封包后的RAW IP包匹配到主机B上的路由规则（10.244.2.0），内核通过查本机路由表知道应该将RAW IP包发送到cni0网桥。

此时，完整的解封包后的以太网帧格式如图5-6所示。

图5-6　解封包后的以太网帧格式

（9）cni0网桥将IP包转发给连接在该网桥上的容器B，就像上文提到的本机容器通信一样，由docker0转发到目标容器，至此整个流程结束。回程报文将按上面的数据流原路返回。

纵观整个过程，flanneld在其中主要起到的作用是：

·UDP封包解包；

·节点上的路由表的动态更新。

flannel的UDP封装是指原始数据由起始节点的flanneld进行UDP封装，经过主机网络投递到目的节点后就被另一端的flanneld还原成了原始的数据包，两边的Docker daemon都感觉不到这个过程的存在。flannel根据etcd的数据刷新本节点路由表，通过路由表信息在寻址时找到应该投递的目标节点。

从图5-3中的虚线部分就可以看到容器A和容器B虽然在物理网络上并没有直接相连，但在逻辑上就好像是处于同一个三层网络中，这种基于底层物理网络设备通过flannel等软件定义网络技术构建的上层网络称之为overlay网络。

以上过程的简图如图5-7所示。

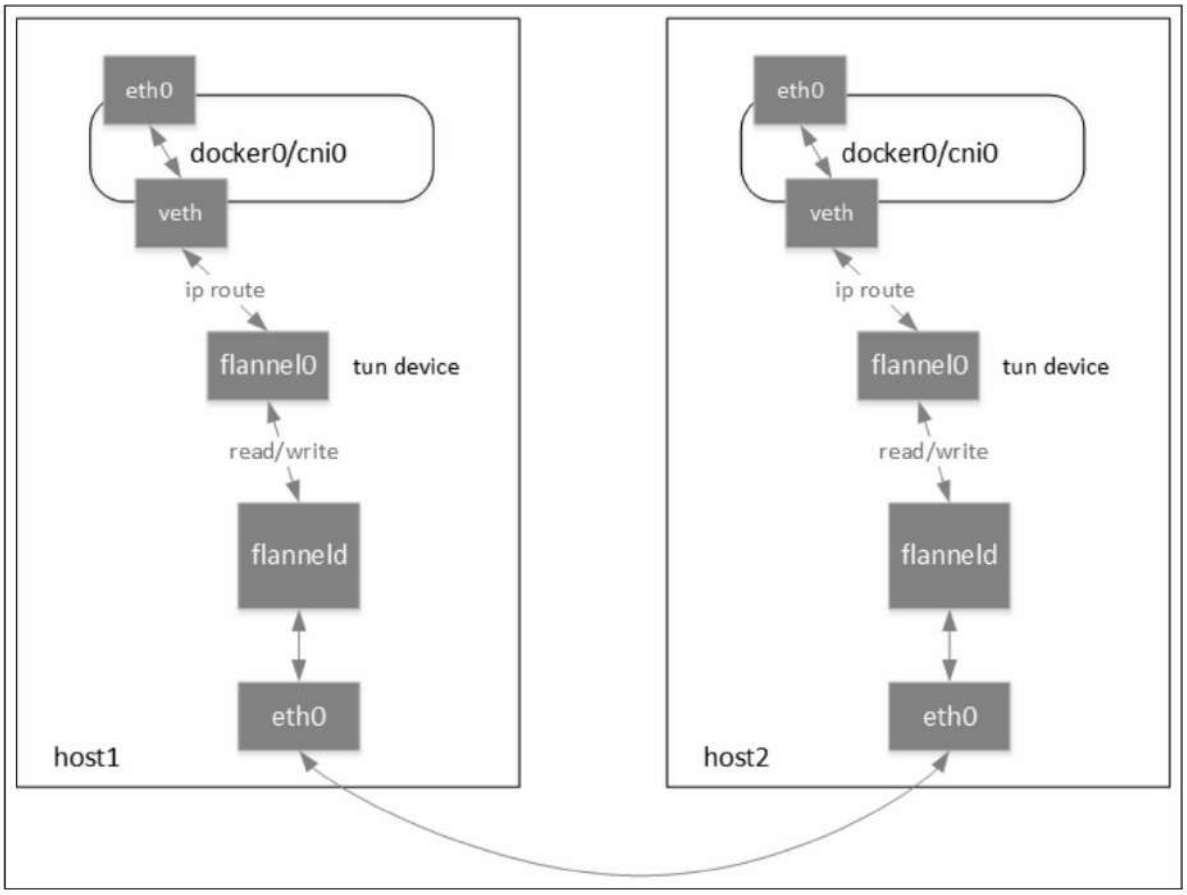


图5-7　flannel UDP模式数据流简图

通过UDP这种backend实现的网络传输过程有没有问题呢？最明显的问题便是网络数据包先通过tun设备从内核中复制到用户态，再由用户态的应用复制到内核，仅一次网络传输就进行了两次用户态和内核态的切换，显然效率是不高的。

那么，有没有比UDP模式更高效的办法呢？能否把封包/解包这些事情都交给Linux内核去做，而不是让flanneld代劳呢？事实上，Linux内核本身也提供了比较成熟的网络封包/解包（隧道传输）实现方案VXLAN，下面我们将分析内核的VXLAN与flanneld的UDP处理网络包时的区别。

2.VXLAN

在开始对flannel VXLAN模式的讨论之前，建议先复习第1章关于VXLAN的介绍。

flannel对VXLAN的使用比较简单，因为目前Kubernetes只支持单网络，故在三层网络上只有1个VXLAN网络。因此，你会发现flannel会在集群的节点上新创建一个名为flannel.1（命名规则为flannel.［VNI］，VNI默认为1）的VXLAN网卡。VTEP的MAC地址不是通过组播学习的，而是通过从API Server处watch Node发现的。

反观flannel UDP模式下创建的flannel0是个tun设备。

可以通过ip-d link查看VTEP设备flannel.1的配置信息。从以下输出可以看到，flannel.1配置的local IP为172.17.130.244（容器网段），flanneld为VXLAN外部UDP包目的端口配置的是Linux VXLAN默认端口8472，而不是IANA分配的4789端口。

```
# ip -d link show flannel.1
...
5: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN
mode DEFAULT
link/ether a2:5e:b0:43:09:a7 brd ff:ff:ff:ff:ff:ff promiscuity 0
VXLAN id 1 local 172.17.130.244 dev eth1 srcport 0 0 dstport 8472 nolearning ageing
300 addrgenmode eui64
```

可能读者会有疑惑，在UDP模式下由flanneld进程进行网络包的封包和解包工作，而在VXLAN模式下解封包的事情交由内核处理，那么此时flanneld的作用是什么呢？带着这个疑问我们先来简单介绍flannel的VXLAN模式是如何工作的。

flanneld启动时先确保VTEP设备（默认为flannel.1）已存在，若已经创建则跳过，并将VTEP设备的相关信息上报到etcd中，当在flannel网络中有新的节点加入集群并向etcd注册时，各个节点上的flanneld从etcd得知通知，并依次执行以下流程：

（1）在节点中创建一条该节点所属网段的路由表，主要是能让Pod中的流量路由到flannel.1接口。

通过route-n可以查看到节点上相关的路由信息：

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.17.0.0      0.0.0.0         255.255.255.0   U     0      0        0 cni0
172.17.1.0      172.17.1.0      255.255.255.0   UG    0      0        0 flannel.1
172.17.2.0      172.17.2.0      255.255.255.0   UG    0      0        0 flannel.1
```

（2）在节点中添加一条该节点的IP及VTEP设备的静态ARP缓存。

可以通过arp-n命令查看当前节点中已经缓存的其他节点上容器的ARP信息。

```
# arp -n
172.17.1.3 dev flannel.1 lladdr 42:7f:69:c7:cd:37 STALE
172.17.2.4 dev flannel.1 lladdr 7a:2c:d0:7f:48:3f STALE
```

通过bridge命令查看节点上的VXLAN转发表（FDB entry），可以看到已经有两条转发表，其中MAC为对端容器的MAC，IP为对端VTEP的对外IP。

```
# bridge fdb show dev flannel.1
42:7f:69:c7:cd:37 dst 192.168.130.164 self permanent
7a:2c:d0:7f:48:3f dst 192.168.130.140 self permanent
```

flannel VTEP的对外IP地址可通过flanneld的启动参数-iface=eth0指定，若不指定则按默认网关查找网络接口对应的IP。

VXLAN模式配置。跟UDP Backend类似，使用flannel的VXLAN模式只需要将flannel配置文件的Backend.Type字段修改为VXLAN。当使用Kubernetes部署flannel时，一个VXLAN模式的ConfigMap如下：

```
# kubectl get configmap -o yaml -n kube-system kube-flannel-cfg
apiVersion: v1
data:
cni-conf.json: |
{
```

```
  "name": "cbr0",
  "type": "flannel",
  "delegate": {
    "isDefaultGateway": true
  }
}
net-conf.json: |
{
  "Network": "10.244.0.0/16",
  "Backend": {
    "Type": "VXLAN"
  }
}
kind: ConfigMap
metadata:
  labels:
  app: flannel
  tier: node
  name: kube-flannel-cfg
  namespace: kube-system
...
```

配置生效后，查看VXLAN的端口是8472：

```
# netstat -ulnp | grep 8472
udp        0      0 0.0.0.0:8472            0.0.0.0:*                -
```

需要注意的是，上面输出的最后一栏显示的不是进程的ID和名称，而是一个破折号“-”，这说明8472这个UDP端口不是由用户态的进程监听的，而是flannel的VXLAN模式工作在内核态下。

VXLAN模式数据路径。在VXLAN模式下，flannel集群容器跨节点通信的数据流程如图5-8所示。

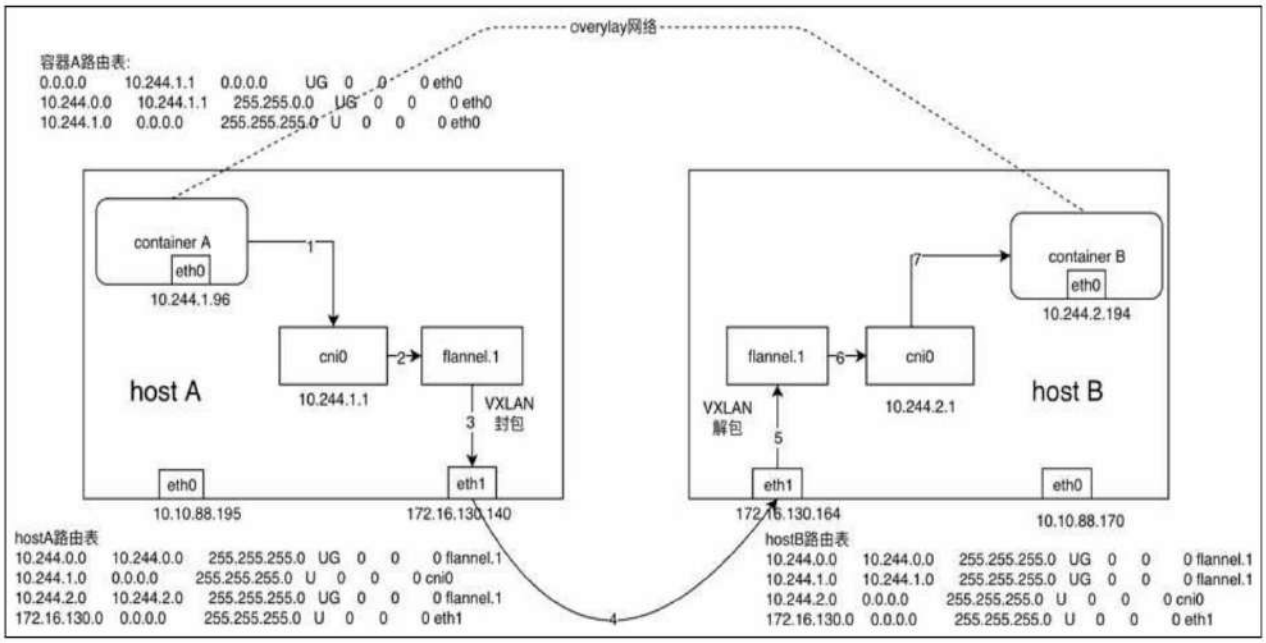


图5-8　flannel VXLAN模式下的网络数据流

（1）同UDP Backend模式，容器A中的IP包通过容器A内的路由表被发送到cni0。

（2）到达cni0中的IP包通过匹配host A中的路由表发现通往10.244.2.194的IP包应该交给flannel.1接口。

（3）flannel.1作为一个VTEP设备，收到报文后将按照VTEP的配置进行封包。首先，通过etcd得知10.244.2.194属于节点B，并得到节点B的IP。然后，通过节点A中的转发表得到节点B对应的VTEP的MAC，根据flannel.1设备创建时的设置参数（VNI、local IP、Port）进行VXLAN封包。

（4）通过host A跟host B之间的网络连接，VXLAN包到达host B的eth1接口。

（5）通过端口8472，VXLAN包被转发给VTEP设备flannel.1进行解包。

（6）解封装后的IP包匹配host B中的路由表（10.244.2.0），内核将IP包转发给cni0。

（7）cni0将IP包转发给连接在cni0上的容器B。

是不是觉得相比UDP模式，VXLAN模式仅从步骤上就少了很多？VXLAN模式相比UDP模式高效也就不足为奇了。

在VXLAN模式下，数据是由内核转发的，flannel不转发数据，仅动态设置ARP和FDB表项。

flannel VXLAN模式的实现

flannel VXLAN模式的实现历经了三个版本的迭代。

（1）flannel的第一个版本，l3miss学习，是通过查找ARP表MAC完成的。l2miss学习，通过获取VTEP上的对外IP地址实现。

（2）flannel的第二个版本，移除了l3miss学习。当远端主机上线时，直接添加对应的ARP表项即可，不用查找学习。

（3）最新版的flannel移除了l2miss学习和l3miss学习，它的工作模式如下：

·创建VXLAN设备，不再监听任何l2miss和l3miss事件消息；

·为远端的子网创建路由；

·为远端主机创建静态ARP表项；

·创建FDB转发表项，包含VTEP MAC和远端flannel的对外IP。

下面介绍flannel的最新版对VXLAN的实现，如图5-9所示。

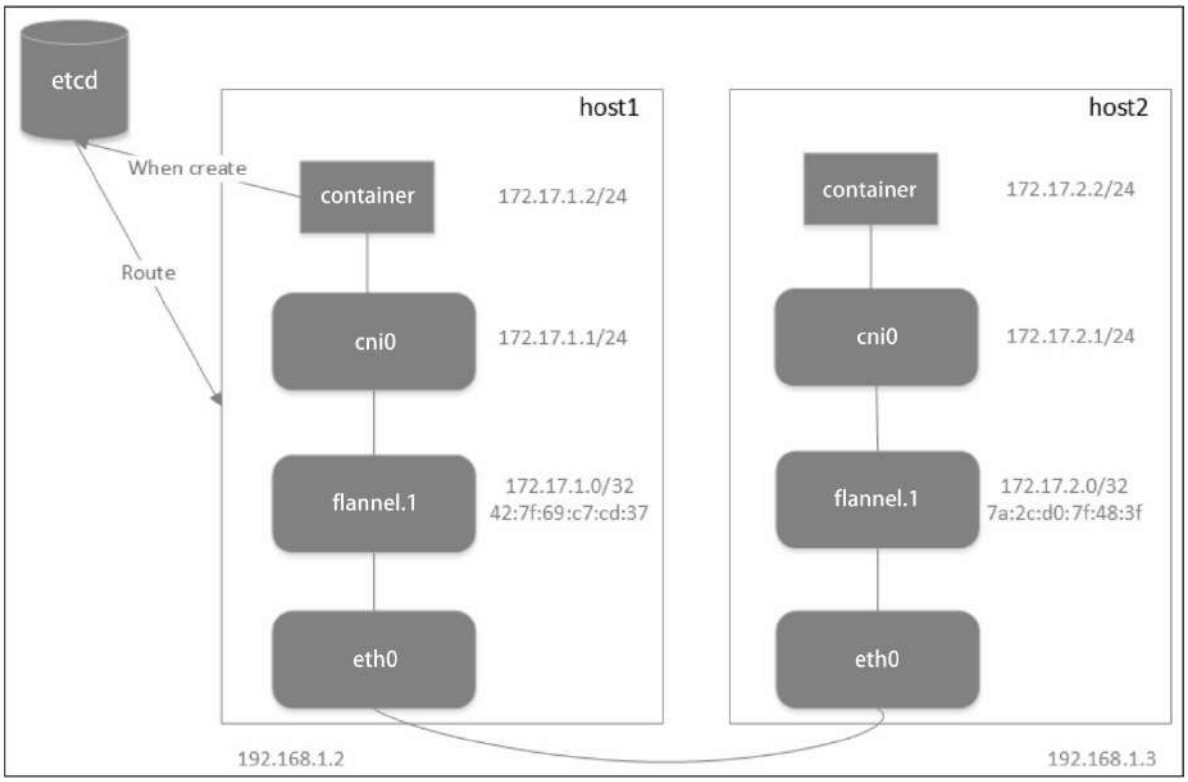


图5-9　flannel的最新版对VXLAN的实现原理图

最新版的flannel完全去掉了l2miss&l3miss方式，改成主动给目的子网添加远端主机路由的方式。同时，为VTEP和网桥各自分配三层IP地址。当数据包到达目的主机后，在内部进行三层转发，这样的好处是主机不需要配置所有的容器二层MAC地址。从一个二层寻址转换成三层寻址，路由数目与主机数（而不是容器）线性相关。官方声称做到了同一个VXLAN子网下每个主机对应一个路由表项，一个ARP表项和一个FDB表项。下面我们将给出在主机1上生成的这几个表项。

一个路由表项：

```
172.17.2.0/24 via 172.17.1.0 dev flannel.1 onlink
```

一个ARP表项：

```
172.17.2.0 dev flannel.1 lladdr 7a:2c:d0:7f:48:3f PERMANENT
```

一个FDB表项：

```
7a:2c:d0:7f:48:3f dev flannel.1 dst 192.168.1.3 self permanent
```

最后，让我们用图5-10总结VXLAN模式下flannel的工作原理。

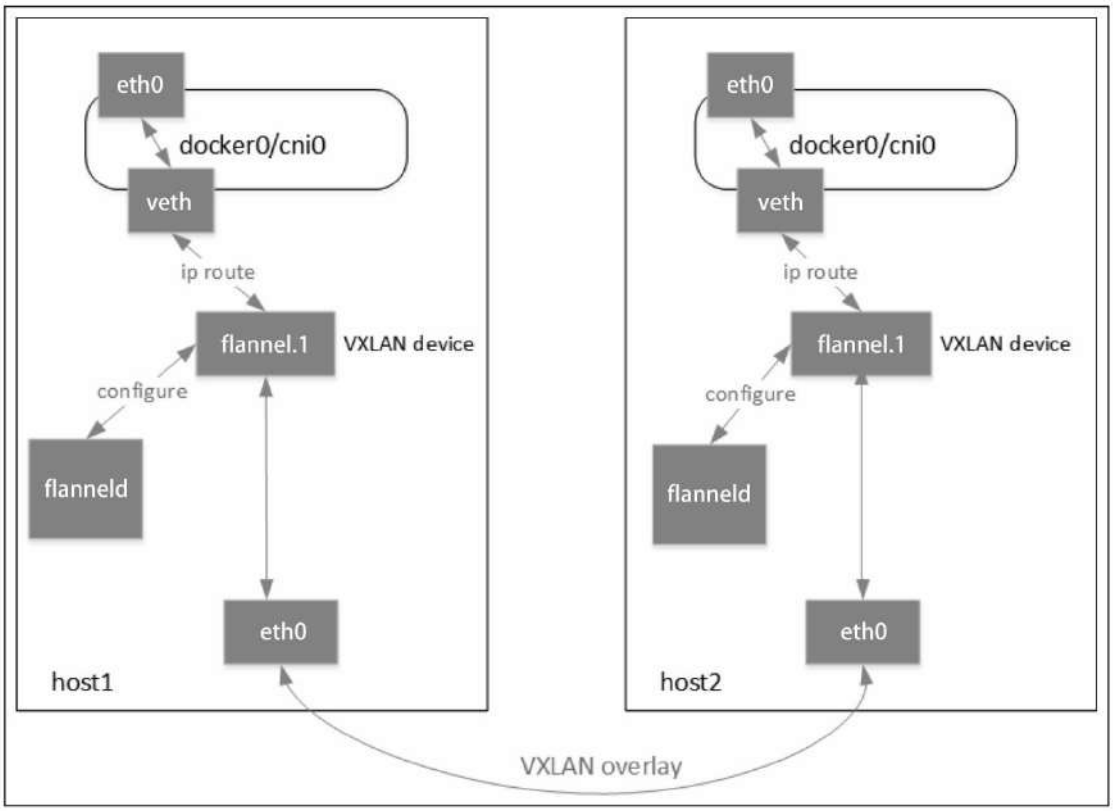


图5-10　VXLAN模式下flannel的工作原理

3.Host Gateway

Host Gateway简称host-gw，从名字中就可以想到这种方式是通过把主机当作网关实现跨节点网络通信的。那么，具体如何实现跨节点通信呢？与UDP和VXLAN模式类似，要使用host-gw模式，需要将flannel的Backend中的.Type参数设置成“host-gw”。这里不再赘述，只介绍网络通信的实现原理。

使用host-gw Backend的flannel网络的网络包传输过程如图5-11所示。

（1）同UDP、VXLAN模式一致，通过容器A的路由表IP包到达cni0。

（2）到达cni0的IP包匹配到host A中的路由规则（10.244.2.0），并且网关为172.16.130.164，即host B，所以内核将IP包发送给host B（172.16.130.164）。

（3）IP包通过物理网络到达host B的eth1。

（4）到达host B eth1的IP包匹配到host B中的路由表（10.244.2.0），IP包被转发给cni0。

（5）cni0将IP包转发给连接在cni0上的容器B。

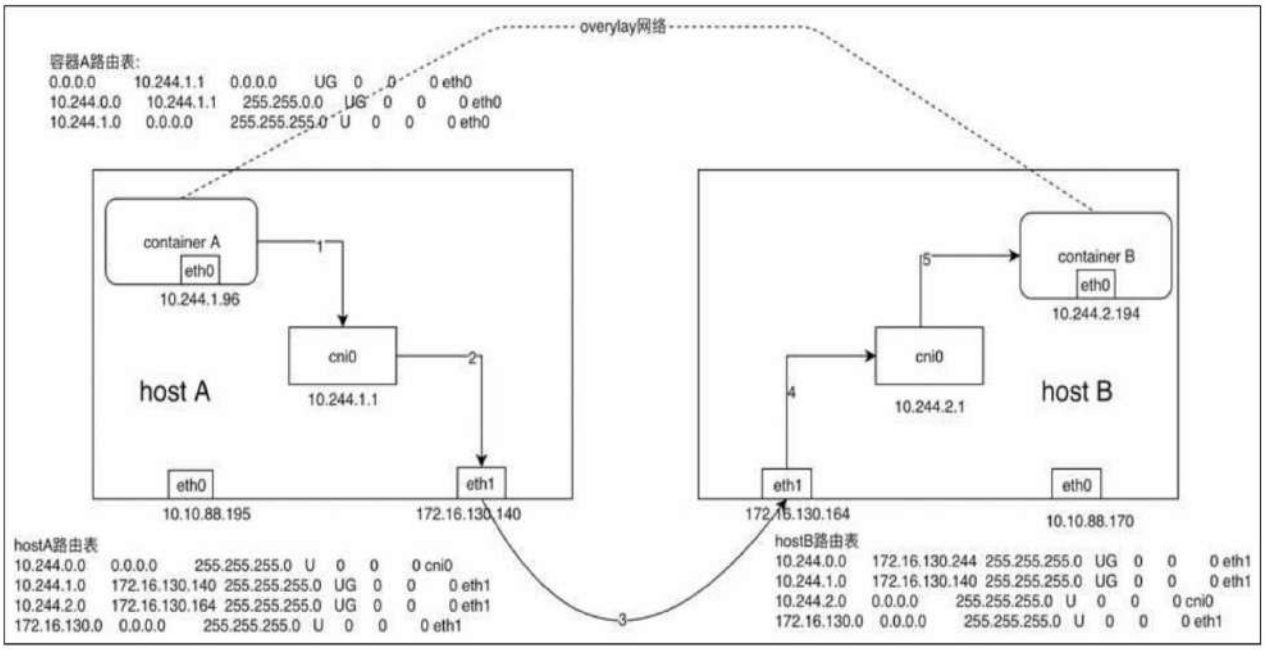


图5-11　flannel host-gw模式

host-gw模式下，各个节点之间的跨节点网络通信要通过节点上的路由表实现，因此必须要通信双方所在的宿主机能够直接路由。这就要求flannel host-gw模式下集群中所有的节点必须处于同一个网络内，这个限制使得host-gw模式无法适用于集群规模较大且需要对节点进行网段划分的场景。host-gw的另外一个限制则是随着集群中节点规模的增大，flanneld维护主机上成千上万条路由表的动态更新也是一个不小的压力，因此在路由方式下，路由表规则的数量是限制网络规模的一个重要因素，我们在Calico时也会讨论这个话题。

采用host-gw模式后，flanneld的唯一作用就是负责主机上路由表的动态刷新。

5.3.4　flannel与etcd

flannel需要使用etcd保存网络元数据。可能有读者会关心flannel在etcd中保存的数据是什么，保存在哪个key中了？在flannel的版本演进过程中，etcd的存储路径发生过多次变化，因此在etcd数据库中全文搜flannel的关键字是最靠谱的一个方式，如下所示：

```
# etcdctl get "" --prefix --keys-only | grep -Ev "^$" | grep "flannel"
```

除了flannel，Calico（5.4节会介绍）和Canal在etcd中的配置数据的key都可以用上面的方法查询，感兴趣的读者可以自行尝试，这里不再赘述。

5.3.5 小结

flannel配置L3 overlay网络，它会创建一个大型内部网络，跨越集群中每个节点。在此overlay网络中，每个节点都有一个子网，用于在内部分配IP地址。在配置Pod时，每个节点上的Docker桥接口都会为每个新容器分配一个地址。同一主机中的Pod可以使用Docker桥接进行通信，而不同主机上的Pod会使用flanneld将其流量封装在UDP数据包中，以便路由到适当的目标。flannel有几种不同类型的后端可用于封装和路由，历经了UDP、VXLAN和Host-Gateway等技术的演进，效率也逐步提升。这一切都离不开Linux的设计哲学：Less is More。

总的来说，flannel是大多数用户的不错选择。从管理角度来看，它提供了一个简单的网络模型，用户只需要一些基础知识，就可以设置适合大多数用例的环境。与其他方案相比，flannel相对容易安装和配置，许多常见的Kubernetes集群部署工具和许多Kubernetes发行版都可以默认安装flannel。在学习Kubernetes的初期，使用flannel是一个稳妥且明智的选择，直到你开始需要一些它无法提供的东西。

5.4 全能大三层网络插件：Calico

Calico性能不错，而且支持Kubernetes网络策略，用的人也多。如果你的环境非常需要支持网络策略，而且你对性能和其他功能也有要求，那么Calico会是一个理想的选择。

Calico作为容器网络方案和我们前面介绍的那些方案最大的不同是它没有采用overlay网络做报文的转发，而是提供了纯3层的网络模型。三层通信模型表示每个容器都通过IP直接通信，中间通过路由转发找到对方。在这个过程中，容器所在的节点类似于传统的路由器，提供了路由查找的功能。要想路由能够正常工作，每个容器所在的主机节点扮演了虚拟路由器（vRouter）的功能，而且这些vRouter必须有某种方法，能够知道整个集群的路由信息。

Calico是一个基于BGP的纯三层的数据中心网络方案（也支持overlay网络），并且与Kubernetes、OpenStack、AWS、GCE等IaaS和容器平台有良好的集成。

Calico的设计比较新颖，之前提到flannel的host-gw模式之所以不能跨二层网络，是因为它只能修改主机的路由，Calico把改路由表的做法换成了标准的BGP路由协议。相当于在每个节点上模拟出一个额外的路由器，由于采用的是标准协议，Calico模拟路由器的路由表信息可以被传播到网络的其他路由设备中，这样就实现了在三层网络上的高速跨节点网络。

> 注：现实中的网络并不总是支持BGP路由，因此Calico也设计了一种ipip模式，使用overlay的方式传输数据。ipip的包头非常小，而且是内置在内核中的，因此它的速度理论上要比VXLAN快，但是安全性更差。

5.4.1 Calico简介

虽然flannel被公认为是最简单的选择，但Calico以其性能、灵活性闻名于Kubernetes生态系统。Calico的功能更全面，正如Calico的slogan提到的：

> 为容器和虚拟机工作负载提供一个安全的网络连接。

Calico以其丰富的网络功能著称，不仅提供容器的网络解决方案，还可以用在虚拟机网络上。除了网络连接，网络策略是Calico最受追捧的功能之一。使用Calico的策略语言，可以实现对容器、虚拟机工作负载和裸机主机各节点之间网络通信进行细粒度和动态的安全规则控制。Calico基于iptables实现了Kubernetes的网络策略，通过在各个节点上应用ACL（访问控制列表）提供工作负载的多租户隔离、安全组及其他可达性限制等功能。此外，

Calico还可以与服务网格Istio集成，以便在服务网格层和网络基础架构层中解释和实施集群内工作负载的网络策略。

在一个Kubernetes集群中，用户可以通过manifest文件快速部署Calico。如果对Calico的网络策略功能感兴趣，则同样可以向集群应用manifest启用这些功能。

Calico还支持容器漂移。因为Calico配置的三层网络使用BGP路由协议在主机之间路由数据包。BGP路由机制可以本地引导数据包，这意味着无须overlay网络中额外的封包解包操作。由于免去了额外包头（大部分情况下依赖宿主机IP地址）封装数据包，容器在不同主机之间迁移没有网络的限制。

虽然VXLAN等技术进行封装也是一个不错的解决方案，但该过程处理数据包的方式通常难以追踪。在Calico网络出现问题时，用户和网络管理员可以使用常规的方法进行故障定位。

Calico在每一个计算节点利用Linux内核的一些能力实现了一个高效的vRouter负责数据转发，而每个vRouter通过BGP把自己运行的工作负载的路由信息向整个Calico网络传播。小规模部署可以直接互联，大规模下可以通过指定的BGP Route Reflector（下文会专门介绍）完成。最终保证所有的工作负载之间的数据流量都是通过IP路由的方式完成互联的。Calico节点组网可以直接利用数据中心的网络结构（无论是L2还是L3），不需要额外的NAT或隧道。

1.名词解释

在深入解析Calico原理前，让我们先来熟悉Calico的几个重要名词：

·Endpoint：接入Calico网络中的网卡（IP）；

·AS：网络自治系统，通过BGP与其他AS网络交换路由信息；

·iBGP：AS内部的BGP Speaker，与同一个AS内部的iBGP、eBGP交换路由信息；

·eBGP：AS边界的BGP Speaker，与同一个AS内部的iBGP、其他AS的eBGP交换路由信息；

·workloadEndpoint：虚拟机和容器端点，一般指它们的IP地址；

·hostEndpoint：宿主机端点，一般指它们的IP地址。

2.Calioc架构解析

Calico的基本工作原理如图5-12所示。

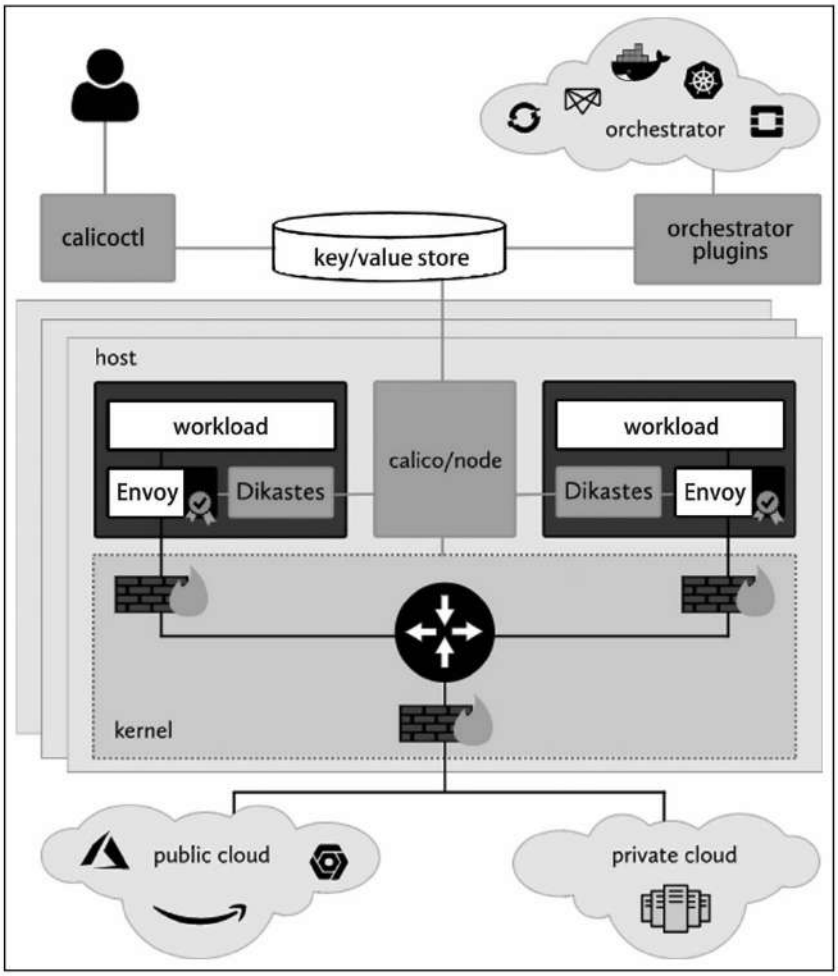


图5-12　Calico的基本工作原理图

·calicoctl：Calico的命令行工具，允许从命令行界面配置实现高级策略和网络；

·orchestrator　plugins：提供与各种流行的云计算编排工具的紧密集成和同步支持；

·key/value　store：存储Calico的策略配置和网络状态信息，目前主要使用etcdv3或Kubernetes API Server；

·calico/node：在每个主机上运行的agent，从key/value存储中读取相关的策略和网络配置信息，并在Linux内核中实现；

·Dikastes/Envoy：可选的Kubernetes　sidecar，可以通过相互TLS身份验证保护工作负载之间的通信安全，并动态配置应用层控制策略。

下面我们将对Calico的几个重要组件进行简要的分析。

Orchestrator Plugin

每个主要的云编排平台都有单独的Calico网络插件（例如Kubernetes、Docker　Swarm等）。这些插件的目的是将Calico无缝集成到编排工具中，就像它们在管理编排工具中内置的网络工具一样管理Calico网络。除此之外，

一个好的Orchestrator Plugin还应该允许用户直接调用编排层的API配置Calico网络，例如Kubernetes的NetworkPolicy API。

Orchestrator Plugin一般负责以下工作：

·API转换。每个编排工具都不可避免地拥有自己的一套用于管理网络的API接口规范，Orchestrator Plugin的作用就是将这些API转换为Calico的数据模型，然后将其存储在Calico的数据存储区中；

·如有需要，orchestrator plugin将从Calico网络向编排器提供管理命令的反馈信息，包括提供有关Felix存活的信息，以及如果网络配置失败，就将某些Endpoint标记为失败。

etcd

etcd是一个分布式键值存储数据库，专注于实现数据存储的一致性。Calico使用etcd提供不同节点上组件之间的数据协同。

在一个Calico集群中，etcd负责：

·数据存储。etcd以分布式、一致和容错的方式存储Calico网络的数据（一般至少由3个etcd节点组成一个集群）。确保Calico网络始终处于已知良好状态，同时允许运行etcd的个别机器节点失败或无法访问。Calico网络数据的这种分布式存储提高了Calico组件从数据库读取的能力；

·组件之间协同交互。我们通过让Calico其他组件watch etcd键值空间中的某些key，确保它们能够及时响应集群状态的更新。

注：Calico可以使用Kubernetes API datastore作为数据库，这还是一个beta版本的功能。因为该API不支持Calico IPAM功能。

如果我们把Calico打开，则Calico内部的实现机制如图5-13所示。

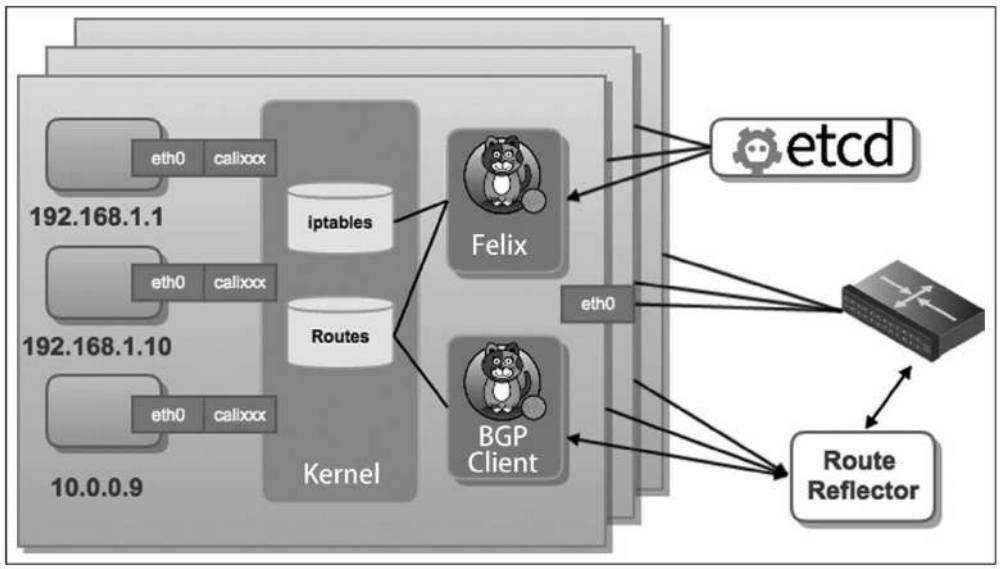


图5-13 Calico内部的实现机制图

Felix

Felix是一个守护程序，作为agent运行在托管容器或虚拟机的Calico节点上。Felix负责刷新主机路由和ACL规则等，以便为该主机上的Endpoint正常运行提供所需的网络连接和管理。进出容器、虚拟机和物理主机的所有流量都会遍历Calico，利用Linux内核原生的路由和iptables生成的规则。

Felix一般负责以下工作：

·管理网络接口，Felix将有关网络接口的一些信息编程到内核中，使内核能够正确处理该Endpoint发出的流量。Felix将确保主机正确响应来自每个工作负载的ARP请求，并将其管理的网卡启用IP Forward；

·编写路由，Felix负责将到其主机上Endpoint的路由编写到Linux内核FIB（转发信息库）中。这可以确保那些发往目标主机的Endpoint的数据包被正确地转发；

·编写ACL，Felix还负责将ACL编程到Linux内核中，即iptables规则。这些ACL用于确保只在Endpoints之间发送有效的网络流量，并确保Endpoint无法绕过Calico的安全措施；

·报告状态，Felix负责提供有关网络健康状况的数据。例如，它将报告配置其主机时发生的错误和问题。该数据会被写入etcd，并对网络中的其他组件可见。

BGP Client

Calico在每个运行Felix服务的节点上都部署一个BGP Client（BGP客户端）。BGP客户端的作用是读取Felix编写到内核中的路由信息，由BGP客户端对这些路由信息进行分发。具体来说，当Felix将路由插入Linux内核FIB时，BGP客户端将接收它们，并将它们分发到集群中的其他工作节点。

BGP Route Reflector（BIRD）

简单的BGP可能成为较大规模部署的性能瓶颈，因为它要求每个BGP客户端连接到网状拓扑中的每一个其他BGP客户端。随着集群规模的增大，一些设备的路由表甚至会被撑满。

因此，在较大规模的部署中，Calico建议使用BGP Route Reflector（路由器反射器）。互联网中通常使用BGP Route Reflector充当BGP客户端连接的中心点，从而避免与互联网中的每个BGP客户端进行通信。Calico使用BGP Route Reflector是为了减少给定一个BGP客户端与集群其他BGP客户端的连接。用户也可以同时部署多个BGP Route Reflector服务实现高可用。Route Reflector仅仅是协助管理BGP网络，并没有工作负载的数据包经过它们，如图5-14所示。

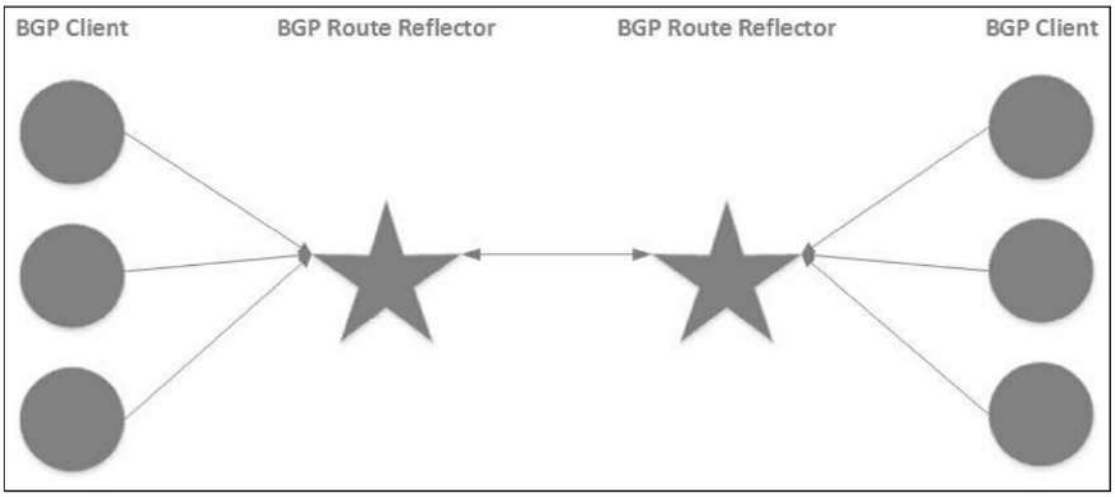


图5-14　BGP Route Reflector的工作原理

在Calico中，最常见的BGP组件是BIRD，配置为Route Reflector运行，而非标准BGP客户端。BIRD是布拉格查理大学数学与物理学院的一个学校项目，项目全名是BIRD Internet Routing Daemon。BIRD旨在开发一个全功能的动态IP路由守护进程，主要针对（但不限于）Linux、FreeBSD和其他类UNIX系统。更多关于BIRD的详细介绍请查看官网资料，这里不再展开介绍。

BGP Route Reflector主要负责集中式的路由信息分发，当Calico BGP客户端将路由通告到Route Reflector时，Route Reflector会将这些路由通告给集群中的其他节点。

5.4.2　Calico的隧道模式

Calico可以创建并管理一个3层平面网络，为每个工作负载分配一个完全可路由的IP地址。工作负载可以在没有IP封装或NAT的情况下进行通信，以实现裸机性能，简化故障排除和提供更好的互操作性。我们称这种网络管理模式为vRtouer模式。vRouter模式直接使用物理机作为虚拟路由器，不再创建额外的隧道。然而在需要使用overlay网络的环境中，Calico也提供了IP-in-IP（简称ipip）的隧道技术。

和其他overlay模式一样，ipip是在各节点之间“架起”一个隧道，通过隧道两端节点上的容器网络连接，实现机制简单说就是用IP包头封装原始IP报文。启用ipip模式时，Calico将在各个节点上创建一个名为tunl0的虚拟网络接口，如图5-15所示。

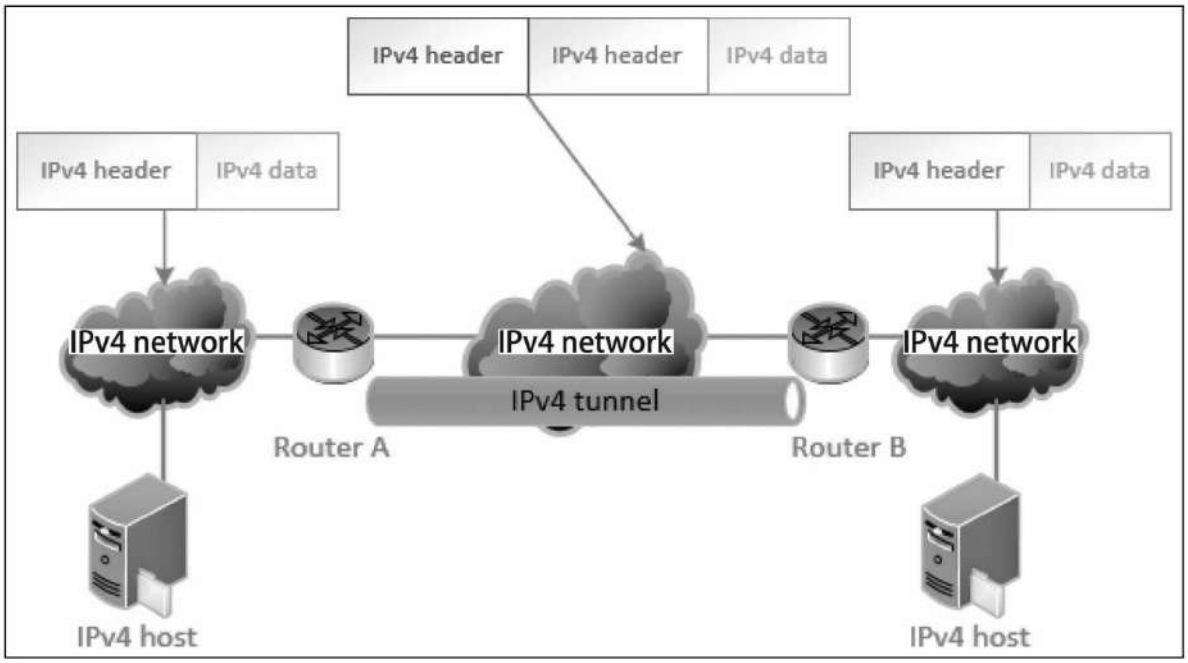


图5-15　Calico ipip模式原理图

我们以flannel为例，已经对隧道网络做了非常详细的介绍，下文将重点介绍Calico的vRouter实现机制。

5.4.3　安装Calico

通过Kubernetes的add-on机制安装Calico非常方便，用户可以通过向Kubernetes应用一个manifest文件部署Calico，用下面一条命令即可：

```
# kubectl apply -f http://docs.projectcalico.org/v2.1/getting-started/kubernetes/
installation/hosted/kubeadm/1.6/calico.yaml
```

如果要使用Calico的可选网络策略，也可以通过Kubernetes创建相应的Network　Policy对象。更多关于Kubernetes和Calico安装及使用Calico网络策略的步骤请参考前面章节，这里不再赘述。

前面的章节中提到的Calico不仅支持CNI还支持CNM模型，即实现了Docker　libnetwork的网络驱动。因此，在Docker中使用Calico作为网络驱动的命令如下所示：

```
# docker network create --driver calico --ipam-driver calico-ipam net1
```

上面的命令创建了一个以Calico为网络驱动的Docker网络，使用的是calico-ipam模块分配容器IP地址。

安装好Calico后，在后续的Pod创建过程中，Kubelet将通过CNI接口调用Calico进行Pod网络的配置，包括IP地址、路由规则、iptables规则等。

注：如果设置CALICO_IPV4POOL_IPIP="off"，即不使用ipip模式，则Calico将不会创建tunl0网络接口，路由规则直接使用物理机网卡作为路由器进行转发。

5.4.4 Calico报文路径

下面我们来分析同一个网络、不同节点的容器是怎么通信的，借此一窥Calico的实现原理。以容器1（container1）ping容器2（container2）为例，先进入容器1中查看它的网络配置和路由表：

```
# docker exec -it container1 sh
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: cali0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff
    inet 192.168.18.64/32 scope global cali0
       valid_lft forever preferred_lft forever
```

可以看到容器1的IP地址为192.168.18.64/32，需要注意的是，它的MAC地址为ee:ee:ee:ee:ee:ee，很明显是个固定的特殊地址。事实上，Calico为所有容器分配的容器MAC地址都一样，这么做是因为Calico只关心三层的IP地址，根本不关心二层MAC地址。

要ping的目的地址是192.168.196.129/32，因为两者不在同一个网络中，所以容器1会查自己的路由表获取下一跳的地址：

```
/ # ip route
default via 169.254.1.1 dev cali0
169.254.1.1 dev cali0
```

容器的路由表非常有趣，和一般服务器创建的规则不同，所有的报文都会经过cali0发送到下一跳169.254.1.1（这是预留的本地IP网段）。Calico为了简化网络配置将容器里的路由规则都配置成一样的，不需要动态更新。知道下一跳之后，容器会查询下一跳168.254.1.1的MAC地址，这个ARP请求发到哪里了呢？cali0是veth pair的一端，其另一端是主机上以caliXXXX命名的

网卡，因此可以通过ethtool-S cali0列出对端的网卡index。

```
# ethtool -S cali0
NIC statistics:
     peer_ifindex: 6
```

而主机上index为6的网卡为：

```
# ip addr
6: calif24874aae57: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether a2:ff:0a:99:57:d2 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::a0ff:aff:fe99:57d2/64 scope link
       valid_lft forever preferred_lft forever
```

报文会发送到这块网卡，这块网卡被分配了一个随机的MAC地址，但是没有IP地址，接收到想要169.254.1.1　MAC地址的ARP请求报文，它会怎么做呢？它的做法是接收ARP请求后直接进行了应答，应答报文中MAC地址是自己的MAC地址。容器的后续报文IP地址还是目的容器，但是MAC地址就变成了主机上该网卡的地址。也就是说，所有的报文都会发给主机，主机根据IP地址再进行转发。

主机上这块网卡不管ARP请求的内容，直接用自己的MAC地址作为应答的行为被称为“ARP proxy”，可以通过以下内核参数检查：

```
# cat /proc/sys/net/ipv4/conf/calif24874aae57/proxy_arp
1
```

总的来说，可以认为Calico把主机作为容器的默认网关使用，所有的报文发到主机，主机根据路由表进行转发。和经典的网络架构不同的是，Calico并没有给默认网关配置一个IP地址，而是通过“ARP proxy”和修改容器路由表的机制实现。

主机上的calico-xxxx网卡接收到报文之后，所有的报文会根据路由表转发：

```
# ip route
169.254.0.0/16 dev eth0  scope link  metric 1002
192.168.18.64 dev calif24874aae57  scope link
blackhole 192.168.18.64/26  proto bird
192.168.18.65 dev cali4e5ed993aed  scope link
192.168.196.128/26 via 172.17.8.101 dev eth0  proto bird
```

由于我们ping报文的目的地址是192.168.196.129，也就是说会匹配路由表的最后一个表项，把172.17.8.101作为下一跳地址，并通过主机的eth0网卡发出。

注：在发送到另一台主机之前，报文还会经过Calico配置的iptables规则。如果被iptables规则拦住，包就会被内核丢弃。

报文到达容器所在的主机172.17.8.101，我们再来介绍主机上的路由表信息：

```
# ip route
169.254.0.0/16 dev eth0 scope link  metric 1003
192.168.18.64/26 via 172.17.8.100 dev eth0  proto bird
192.168.196.128 dev cali4907e793262  scope link
blackhole 192.168.196.128/26  proto bird
192.168.196.129 dev cali69b2b8c106c  scope link
```

同样地，这个报文会匹配最后一个路由规则，这个规则匹配的是一个IP地址，而不是网段。也就是说，主机上的每个容器都会有一个对应的路由表项。报文被发送到cali69b2b8 c106c这个veth pair，然后从另一端发送给目标容器。目标容器接收到报文之后，回复ICMP报文，应答报文原路返回。

以上过程如图5-16所示。

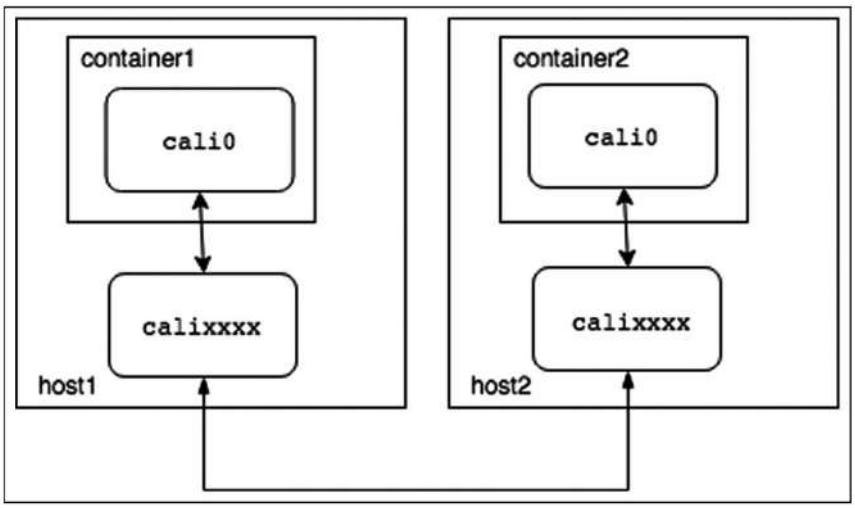


图5-16　Calico报文路径简图

5.4.5　Calico使用指南

在了解了Calico的内部实现机制后，我们将介绍Calico常见的用法。

1.Calico CNI插件配置说明

Calico　CNI插件是按照标准的CNI配置规范进行配置的。一个最小化的配置文件如下：

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "ipam": {
        "type": "calico-ipam"
    }
}
```

如果节点上的calico-node容器注册了节点主机名以外的NODENAME，则此节点上的CNI插件必须配置使用相同的NODENAME，例如：

```
{
    "name": "any_name",
    "nodename": "<NODENAME>",
    "type": "calico",
    "ipam": {
        "type": "calico-ipam"
    }
}
```

一些常见的配置项如下所示：

·datastore_type、Datastore　type、default:etcdv3，设置为Kubernetes时表示直接使用Kubernetes API存取数据库服务；

·etcd location，在使用etcd作为后端数据库服务时，以下配置项有效：

-etcd_endpoints

-etcd_key_file

-etcd_cert_file

-etcd_ca_cert_file

·log_level，可选值为WARNING，INFO，DEBUG，默认值是WARNING，日志打印到stderr；

·ipam，IP地址管理工具，值为一个JSON字典，可以包含以下子配置项：

-“type”:“calico-ipam”

-“assign_ipv4”:“true”

-“assign_ipv6”:“true”

-“ipv4_pools”:［“10.0.0.0/24”，“20.0.0.0/16”，“default-ipv4-ippool”］

-“ipv6_pools”:［“2001:db8::1/120”，“default-ipv6-ippool”］

“container_settings”:｛“allow_ip_forwarding”:true｝，默认值为false，该选项允许在容器命名空间内配置设置。

将Calico CNI插件与Kubernetes一起使用时，插件必须能够访问Kubernetes API服务器，才能找到分配给Kubernetes Pod的标签，一般都通过在Calico CNI配置文件中指定kubeconfig文件解决，例如：

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "kubernetes": {
        "kubeconfig": "/path/to/kubeconfig"
    },
    "ipam": {
        "type": "calico-ipam"
    }
}
```

或者通过Kubernetes API Server，例如：

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "kubernetes": {
        "k8s_api_root": "http://127.0.0.1:8080"
    },
    "ipam": {
        "type": "calico-ipam"
    }
}
```

如果要使用Kubernetes NetworkPolicy功能，则必须在网络配置中设置策略类型。Calico目前只支持一个策略类型，即Kubernetes。

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "policy": {
      "type": "k8s"
    },
    "kubernetes": {
        "kubeconfig": "/path/to/kubeconfig"
    },
    "ipam": {
        "type": "calico-ipam"
    }
}
```

当使用type:k8s时，Calico CNI插件需要对所有命名空间中的Pods资源具备读访问权限。

2.为Pod分配固定IP

很多传统的应用程序，在上容器云时都会有使用固定IP地址的需求。虽然这种要求并不符合Kubernetes对网络的基本假定，但Calico IPAM却支持为Pod分配固定IP。Kubernetes有两个相关的annotations可以使用。

cni.projectcalico.org/ipAddrs

指定一个要分配给Pod的IPv4和/或IPv6的地址列表。请求的IP地址将从Calico IPAM分配，并且必须存在于已配置的IP池中。例如：

```
annotations: "cni.projectcalico.org/ipAddrs": "[\"192.168.0.1\"]"
```

cni.projectcalico.org/ipAddrsNoIpam

指定一个要分配给Pod的IPv4和/或IPv6地址列表，绕过IPAM。任何IP冲突和路由配置都必须由人工或其他系统处理。Calico仅处理那些属于Calico IP池中的IP地址，将其路由分发到Pod。如果分配的IP地址不在Calico IP池中，则必须确保通过其他机制正确地处理该IP地址的路由。例如：

```
annotations: "cni.projectcalico.org/ipAddrsNoIpam": "[\"10.0.0.1\"]"
```

默认情况下，Calico禁用了ipAddrsNoIpam功能。用户若要使用该功能，还需要在CNI网络配置的feature_control字段中启用：

```
{
      "name": "any_name",
      "cniVersion": "0.1.0",
      "type": "calico",
      "ipam": {
          "type": "calico-ipam"
      },
     "feature_control": {
         "ip_addrs_no_ipam": true
     }
}
```

注：此功能允许通过IP欺骗绕过网络策略。用户应确保适当的准入控制，以防止用户使用不当的IP地址。

需要注意的是：

·ipAddrs和ipAddrNoIpam不能同时使用；

·只能使用这些批注指定一个IPv4/IPv6地址或一个IPv4和一个IPv6地址；

·当ipAddrs或ipAddrsNoIpam与ipv4pools或ipv6pools一起使用时，ipAddrs/ipAddrsNoIpam有更高的优先级。

3.IPAM

使用CNI的host-local IPAM插件时，subnet字段允许使用一个特殊的值“usePodCidr”。告诉插件从Kubernetes API获取Node.podCIDR字段，以确定自己要使用的子网。Calico不使用网段范围的网关字段，因此不需要该字段。如果提供了，则忽略该字段。

注意：usePodCidr只能用作子网字段的值，不能在rangeStart或rangeEnd中使用，因此如果子网设置为usePodCidr，则这些值无用。

Calico支持host-local IPAM插件的routes字段，如下所示：

·如果没有routes字段，Calico将在Pod中安装默认的0.0.0/0和/或::/0的路由规则（取决于Pod是否具有IPv4和/或IPv6地址）；

·如果存在routes字段，则Calico将仅将routes字段中的路由规则添加到Pod中。Calico在Pod中实现了点对点链接，因此不需要gw字段，如果存在则忽略它。Calico安装的所有路由都会将Calico的link-local IP作为下一跳；

·node_name字段，指定用于查询“usePodCidr”的节点的nodename，默认值为当前节点的hostname。

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "kubernetes": {
        "kubeconfig": "/path/to/kubeconfig",
        "node_name": "node-name-in-k8s"
    },
    "ipam": {
        "type": "host-local",
        "ranges": [
            [
                { "subnet": "usePodCidr" }
            ],
            [
                { "subnet": "2001:db8::/96" }
            ]
        ],
        "routes": [
            { "dst": "0.0.0.0/0" },
```

```
            { "dst": "2001:db8::/96" }
        ]
    }
}
```

4.基于每个命名空间或每个**Pod**指定专属**IP**池

Calico IPAM支持为每个命名空间或者每个Pod指定专用的IP池资源，这一功能是通过应用Kubernetes annotations实现的。

cni.projectcalico.org/ipv4pools

已配置的IPv4 pool列表，可从中选择Pod的地址。

```
annotations: “cni.projectcalico.org/ipv4pools”: “[\” default-ipv4-ippool\” ]”
```

cni.projectcalico.org/ipv6pools

已配置的IPv6 pool列表，可从中选择Pod的地址。

```
annotations: “cni.projectcalico.org/ipv6pools”: “[\” 2001:db8::1/120\” ]”
```

如果提供上面这样的配置，则这些指定的IP池将覆盖CNI基础配置中指定的任何IP池资源。

Calico CNI插件支持为每个namespace配置annotations信息。如果namespace和Pod都有此配置，则优先使用Pod的配置。

5.4.6 为什么Calico网络选择BGP

我们都知道Calico使用了BGP，那么为什么Calico要选择BGP呢？为什么选择BGP而不是一个IGP协议（如OSPF或者IS-IS）？要弄清楚原因，我们需要先明确目前BGP和IGP是如何在一个大规模网络中工作的。

任何网络，尤其是大型网络，都需要处理两个不同的路由问题：

（1）发现网络中路由器之间的拓扑结构。

（2）发现网络中正在工作的节点，以及可到达该网络的外部连接。

图5-17可以帮助我们理解以上这两个问题。

网络中的端点（Endpoint）没有运行路由协议的能力，而路由器

（Router）可以。同一个网络内的两个网络之间通过路由器连接。

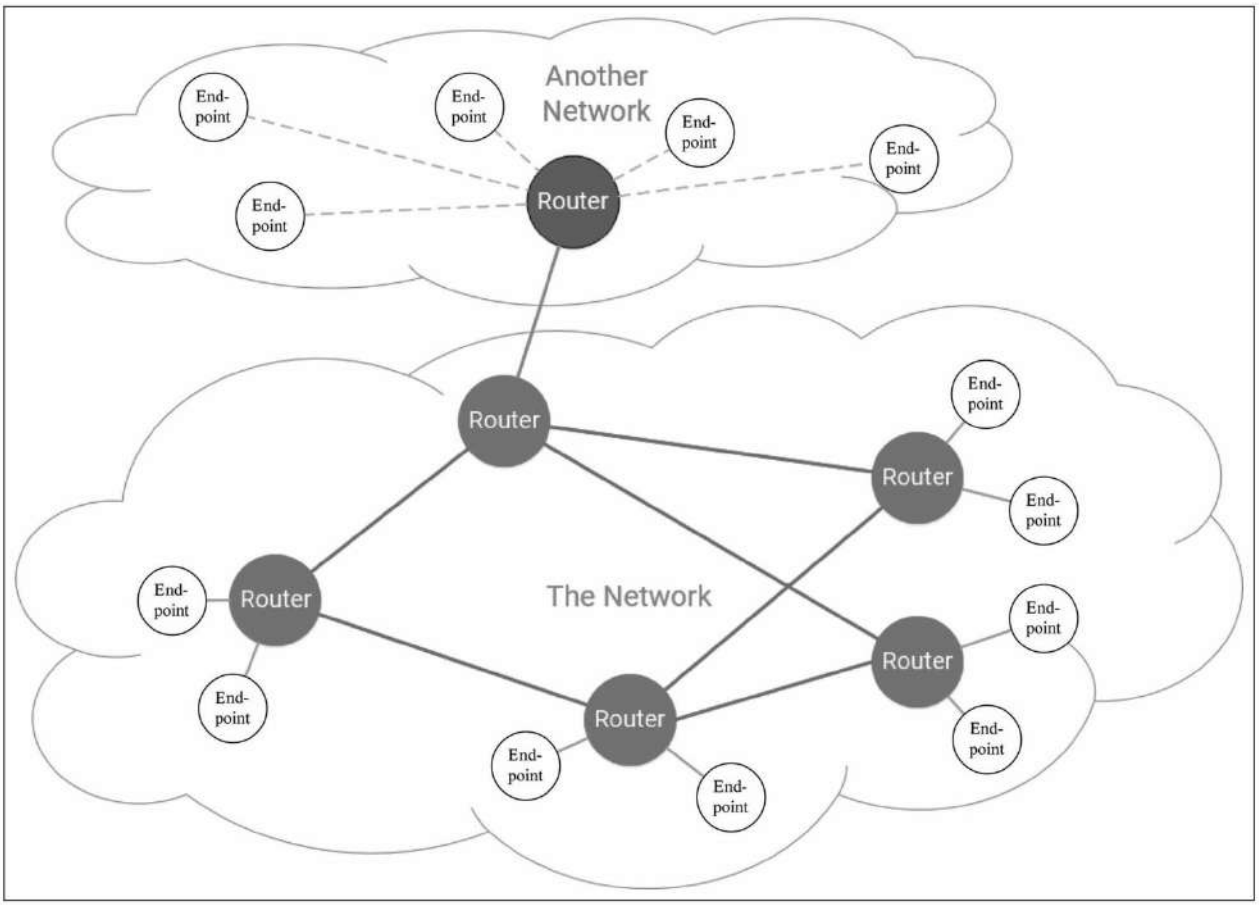


图5-17　大型网络的路由问题

IGP需要执行大量复杂计算，才能让每台设备在同一时刻都能得到对所处网络拓扑的相同认知。这其实就限制了IGP所能运行的规模。在一个IGP的单一视图内应该只包含几十个（极端情况下可能是小几百个）路由器，以满足这些规模和性能的需求。虽然存在一些可以突破规模限制的技术［比如在OSPF中使用area（区域）或者在IS-IS中使用Level（层），两者都可以认为是将网络切割成若干个单一视图］，但这些技术也带来了其他架构上的限制（超出本书范畴这里不再展开）。

IGP也被限制在它们所能通告的最大Endpoints数量上，这个数量浮动范围比较大，但是上限也就在几千到1万多。当IGP中的路由数量超过5000或6000条时，许多大型网络的管理人员都会感到紧张。

那么BGP呢？截至2015年1月，在公共互联网上宣告的路由超过526000条，一些网络中甚至拥有超过100万的节点。为了解决大规模网络中的路由可扩展性问题，BGP被开发出来。BGP可以在一个网络中扩容到几百台路由器的规模，而如果使用BGP Route Reflection这一数量更是可以到达数万台。如果需要的话，BGP可以宣告数百万条路由信息，并通过非常灵活的策略加以管理。

因此，我们就能理解为什么Calico使用BGP宣告网络端点的路由了。一言以蔽之，就是提高路由规则的可扩展性以满足大规模组网的需求。在Calico设计的网络中，可能会出现几万台路由器，以及潜在的几百万条路由信息或者Endpoints，这种数量级是与IGP的设计不匹配的，但是BGP可以，特别是当我们使用BGP Route Reflection扩展路由器数量的时候。

Calico的核心设计思想是：使用通用的互联网工具和技术实现大规模网络互连结构。产生这种设计思想的主要原因是互联网行业在运营真正大型网络上已经积累了几十年的经验，使用的工具和技术（例如BGP）也已经过长时间的磨炼，如果把这些工作全部丢进垃圾桶，重新造轮子就不够明智了。对于那些数据终端规模已经接近互联网规模的公有云环境，一个云上可用区就可以轻松承载数千到数万个服务器，并运行着几万甚至小几十万个虚拟机（在Calico中称为Endpoints）。如果这些虚拟机中再运行容器，则Endpoints的数量可能还会增加一个或两个数量级。因此，从规模角度，我们应该尽量复用互联网运营商的成功经验。

总结Calico网络使用BGP的原因有：

（1）BGP是一种简单的路由协议。

（2）拥有当前行业的最佳实践。

（3）唯一能够支撑Calico网络规模的协议。

5.4.7 小结

Calico是一个比较有意思的虚拟网络解决方案，完全利用路由规则实现动态组网，通过BGP通告路由。由于Calico利用宿主机协议栈的三层确保容器之间跨主机的连通性，报文的流向完全通过路由规则控制，没有overlay，没有NAT，直接经过宿主机协议栈处理，因此我们称Calico是一个纯三层的组网方案，转发效率也较高。

Calico的核心设计思想就是Router，它把每个操作系统的协议栈看成一个路由器，然后把所有的容器看成连在这个路由器上的网络终端，在路由器之间运行标准的BGP，并让节点自己学习这个网络拓扑该如何转发。然而，当网络端点数量足够大时，自我学习和发现拓扑的收敛过程非常耗费资源和时间。

Calico的缺点是网络规模会受到BGP网络规模的限制。Calico路由的数目与容器数目相同，极易超过三层交换、路由器或节点的处理能力，从而限制了整个网络的扩张，因此Calico网络的瓶颈在于路由信息的容量。Calico会在每个节点上设置大量的iptables规则和路由规则，这将带来极大的运维和故障排障难度。Calico的原理决定了它不可能支持VPC，容器只能从

Calico设置的网段中获取IP。Calico目前的实现没有流量控制的功能，会出现少数容器抢占节点多数带宽的情况。当然，我们可以结合CNI的bandwidth插件实现流量整形。

Calico的应用场景主要在IDC内部，Calico官方推荐将其部署在大二层网络上，这样所有路由器之间是互通的。所谓大二层就是没有任何三层的网关，所有的机器、宿主机、物理机在二层是可达的。大二层主要的问题是弹性伸缩的问题。频繁开关机的时候，容器启停虽然不影响交换机，但容易产生广播风暴。事实上，在很多有经验的网络工程师眼里，大二层存在单一故障问题，也就是说，任何一个都会有一定的硬件风险让整个大二层瘫痪。因此，实际场景经常会把集群划分成多个网段，对外是三层网络的结构。

如图5-18所示，从架构上看，Calico在每个节点上会运行两个主要的程序，一个是Felix，另一个是BIRD。Felix会监听etcd的事件并负责配置节点上容器的网络协议栈和主机上的iptables规则及路由表项。BIRD会从内核里获取IP的路由发生了变化的信息，并告知Route Reflector。Route Reflector是一个路由程序，它会通过标准BGP的路由协议扩散到其他宿主机上，让集群的其他容器都知道这个IP。

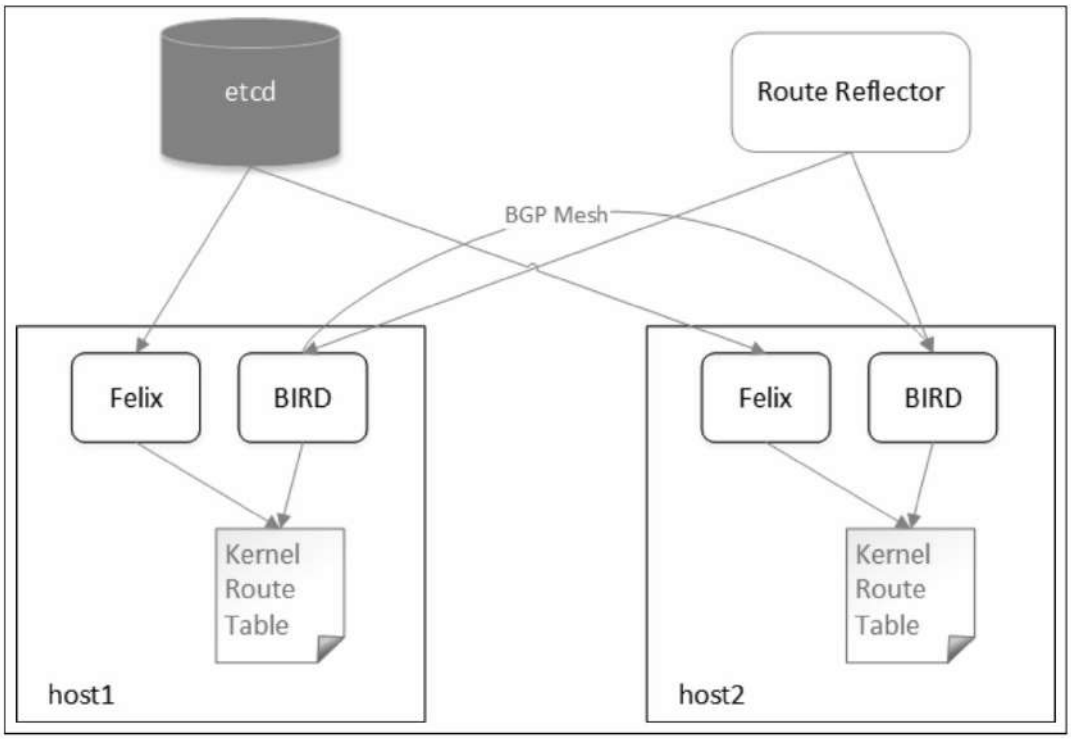


图5-18　Calico网络拓扑

参考资料：https://www.lijiaocn.com/%E9%A1%B9%E7%9B%AE/2017/04/11/calico-usage.html.

5.5 Weave：支持数据加密的网络插件

Weave是CNCF的官方子项目，是其第一个也是目前唯一一个容器网络插件项目。用一个词评价Weave，就是“功能齐全”，有网络通信、有安全策略、有域名服务、有加密通信还有监控。

Weave的工作模式与flannel相似，它最早只提供了UDP（称为sleeve模式）的网络模式，后来又加上了fastpath方式（基于VXLAN和OVS），不过Weave消除了flannel中用来存储网络地址的额外组件etcd，自己集成了高可用的数据存储功能。

5.5.1 Weave简介

Weave是由Weaveworks提供的一种符合Kubernetes CNI定义的网络插件，它提供的模式和我们目前为止讨论的所有网络方案都不同。Weave是一个多主机容器网络方案，能够创建一个虚拟网络，用于连接部署在多台主机上的Docker容器，这样容器就像被接入了同一个网络交换机。Weave支持将容器应用暴露成服务供外部访问，客户端访问者无须关心后端提供服务的容器运行在哪里。另外，Weave支持加密通信，允许用户从一个不受信任的网络连接到Weave主机。

Weave的控制平面是个去中心化的架构，由于没有了中心控制节点，Weave网络中的节点通过Gossip协议进行数据同步。这种方式省去了集中式的存储系统，能够在一定程度上降低部署的复杂性。

Weave网络中的每个主机上都会安装wRouter，这些wRouter之间建立全网络的TCP连接，通过这个连接进行心跳握手和拓扑信息交换（Gossip协议的需要），以维护可用网络环境的最新视图。当流量通过路由器时，它们会了解哪些对等体与哪些MAC地址相关联，从而允许它们以更少的跳数、更智能地路由后续流量。wRouter建立的连接可以通过配置进行加密。Weave的网络拓扑如图5-19所示。

对那些寻求网络功能丰富，同时不增加大量复杂性的人来说，Weave是一个很好的选择。它设置起来相对容易，提供了许多内置和自动配置的功能，并且可以在其他解决方案出现故障的场景下提供智能路由。虽然网状（mesh）的网络拓扑结构会限制网络的可扩展性，但是对大多数用户来说，这不是一个大问题。此外，Weave也提供收费的技术支持，可以为企业用户提供故障排除等技术服务。因此，我们一般也认为Weave是商业化程度较高的容器网络产品。

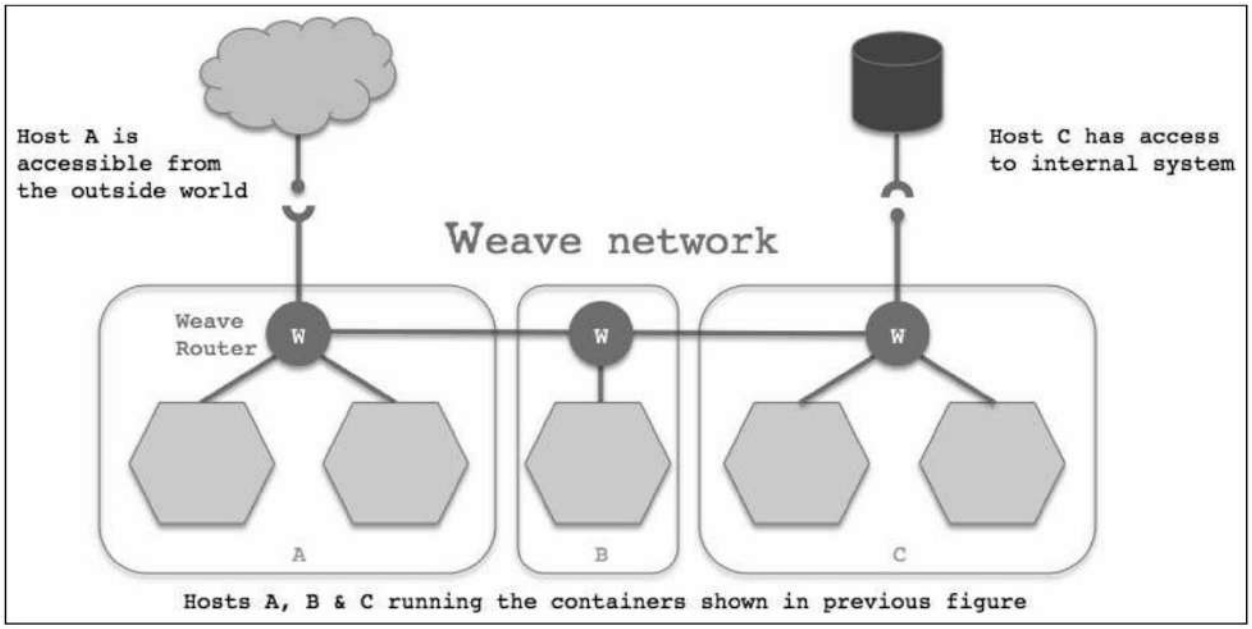


图5-19　Weave的网络拓扑

通过Kubernetes安装Weave比较简单，只需一条命令，如下所示：

```
# kubectl apply -f https://git.io/weave-kube
```

5.5.2　Weave实现原理

数据平面上，Weave通过封包的实现了L2　　overlay。封包支持两种模式，一种是运行在用户态的Sleeve模式，另一种是运行在内核态的fastpath（快速数据路径）模式。Sleeve模式通过pcap设备在Linux网桥上截获数据包并由wRouter完成UDP封包。Sleeve模式支持加密流量，但参考flannel我们知道，这种方式性能损失明显。fastpath模式通过VXLAN协议进行封包，wRouter不直接参与转发，而是通过下发odp流表的方式控制转发。fastpath模式依靠内核的本机Open vSwitch数据路径模块（ODP），将数据包转发到目的容器，数据包无须多次在用户态和内核态拷贝。Fastpath模式下，wRouter的职责是更新Open　vSwitch配置，以确保Linux内核掌握路由数据包的准确信息。fastpath模式相对Sleeve模式可以明显地提升性能，但是不支持加密等高级功能。当需要将流量发送到位于不同节点上的Pod时，Weave路由组件会自动决定是通过Fastpath模式发送，还是回退到Sleeve模式转发的方法（例如，当低版本的内核不支持VXLAN时，或者在缺少Fastpath模式必要的路由信息或连接的情况下）。

与Calico一样，Weave也为Kubernetes集群提供网络策略功能。一个其他网络方案都没有，但Weave独有的功能是对整个网络的简单加密，虽然这会增加相当多的网络开销。在Sleeve模式下，Weave可以使用NaCl加密所有流

量，而对于Fastpath模式，因为它需要加密内核中的VXLAN流量，所以Weave会使用IPsec ESP进行加密。

Weave网络主要通过路由服务和网桥转发。Weave会在主机上创建一个网桥，每一个容器通过veth pair连接到该网桥，容器的IP地址可以由用户或者Weave的IPADM分配。同时，网桥上有个wRouter容器与之连接，该router会通过连接在网桥上的接口抓取网络包。Weave网桥工作在Promisc模式，更多关于混杂模式的介绍详见第1章：

每个部署了wRouter的主机都需要打开防火墙的678端口（包括TCP和UDP），以保证wRouter之间控制面流量和数据面流量的通过。控制面由wRouter之间建立的TCP连接构成，通过它进行握手和拓扑关系信息的交换通信。这个通信可以被配置为加密通信。而数据面由wRouter之间建立的UDP连接构成，这些连接大部分都会加密。这些连接都是全双工的，并且可以穿越防火墙。

5.5.3 Weave安装

以下安装步骤需要在集群所有节点都执行一遍。

Weave不需要集中式的key-value存储，所以安装和运行都很简单。直接把Weave二进制文件下载到系统中就可以了：

```
# curl -L git.io/weave -o /usr/local/bin/weave
# chmod a+x /usr/local/bin/weave
```

运行weave launch启动相关服务（先下载Weave镜像，提前安装好Docker）：

```
# weave launch
Unable to find image 'weaveworks/weaveexec:latest' locally
Trying to pull repository docker.io/weaveworks/weaveexec ...
latest: Pulling from docker.io/weaveworks/weaveexec
79650cf9cc01: Pull complete
.......
```

此时，会发现有两个网桥，一个是Docker默认生成的，另一个是Weave生成的：

```
# brctl show
bridge name bridge id           STP enabled   interfaces
docker0     8000.0242376456d7   no
weave       8000.32298bba31f1   no            vethwe-bridge
```

查看运行的容器，发现Weave路由容器已经自动运行，如下所示：

```
# docker ps
CONTAINER ID        IMAGE                       COMMAND                  CREATED
      STATUS              PORTS               NAMES
c5aacecbe40e        weaveworks/weave:latest   "/home/weave/weaver -"   6 minutes ago
      Up 6 minutes                          weave
```

关闭Weave：

```
# weave stop

# 或者直接关闭并移除weave容器
# docker stop weave
# docker rm weave
```

关闭Weave（weave stop）后，发现Weave网桥信息还在：

```
# brctl show
bridge name bridge id           STP enabled interfaces
docker0     8000.0242b0c9bf2d   no
weave       8000.22a85b2682a1   no          vethwe-bridge

# ifconfig
weave: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1376
        inet 192.168.100.1  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::20a8:5bff:fe26:82a1  prefixlen 64  scopeid 0x20<link>
        ether 22:a8:5b:26:82:a1  txqueuelen 0  (Ethernet)
        RX packets 57  bytes 3248 (3.1 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 1460 (1.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

删除Weave网桥：

```
# ip link set dev weave down
# brctl del br weave

# brctl show
bridge name bridge id           STP enabled interfaces
docker0     8000.0242b0c9bf2d   no
```

5.5.4 Weave网络通信模型

Weave通过在集群的每个节点上启动虚拟路由器wRouter，形成互联互通的网络拓扑，在此基础上，实现容器的跨主机通信。因此，Weave网络本质上是由这些wRouter组成的网状拓扑。

wRouter起到路由器的作用，先在路由器之间建立一个全打通的TCP连接网，然后在这张TCP的连接网里运行路由协议，形成一个控制平面。Weave的控制平面如图5-20所示。

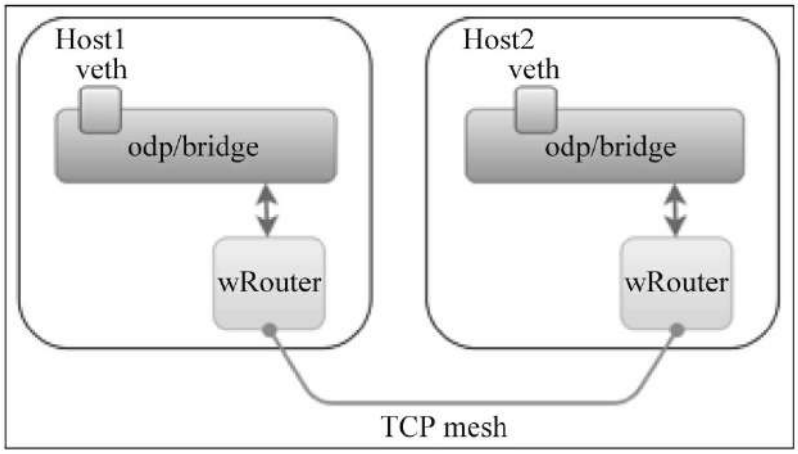


图5-20　Weave控制平面

从图5-20中可以看出，Weave的控制平面和Calico非常类似。

Weave数据平面是overlay的，这一点和flannel一致，如图5-21所示。

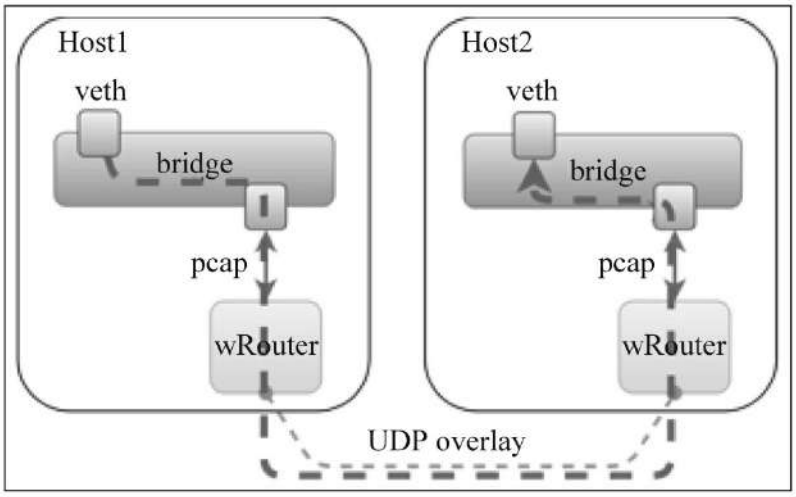


图5-21　Weave数据平面

因此，Weave被认为是吸取了flannel和Calico这两个方案的设计思想的一个网络插件。

在网络隔离方面，Weave的方案相对比较粗糙，只是子网级的隔离，如图5-22所示。

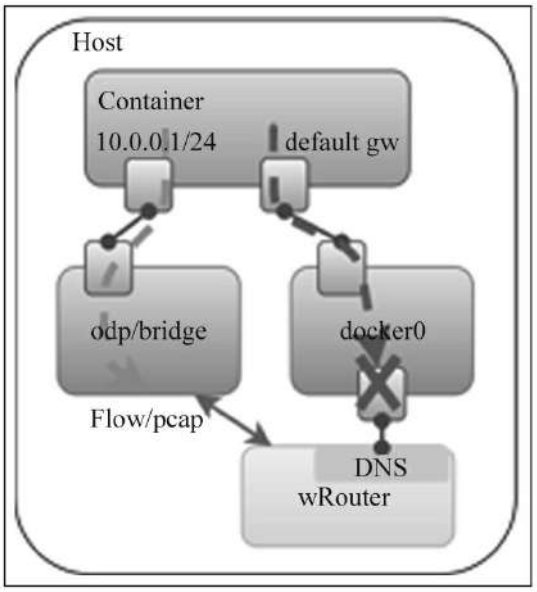


图5-22　Weave网络隔离

图5-22示意的是，假如有两个容器都处在10.0.1/24网段，那么它会在所有的容器里加一条路由，声明该网段的流量都会从左边的网桥出去，但是所有非此网段的流量会经过docker0。docker0和其他网络是不通的，因此就达到了隔离网段的效果。

Weave的服务发现如图5-23所示，Weave会为每个容器创建两块网卡，其中一块网卡绑定在Weave自己的网桥上，另一块绑定在原生Docker的网桥上。Weave会有一个个DNS的服务器监听在这两个网桥上，实际上这个DNS内嵌在wRouter中。当容器发起DNS查询时，这些请求都会被路由到wRouter

中的DNS服务器。Weave的负载均衡是基于DNS的轮询机制。

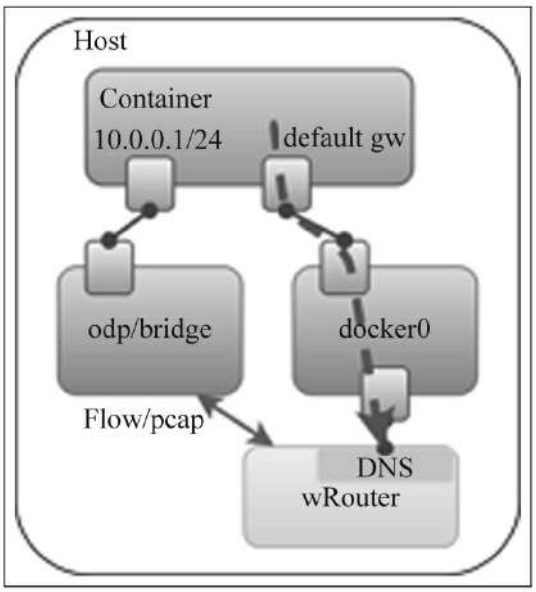


图5-23　Weave的服务发现与负载均衡机制

下面以Sleeve模式为例，介绍Weave跨主机通信的网络报文路径，如图5-24所示。

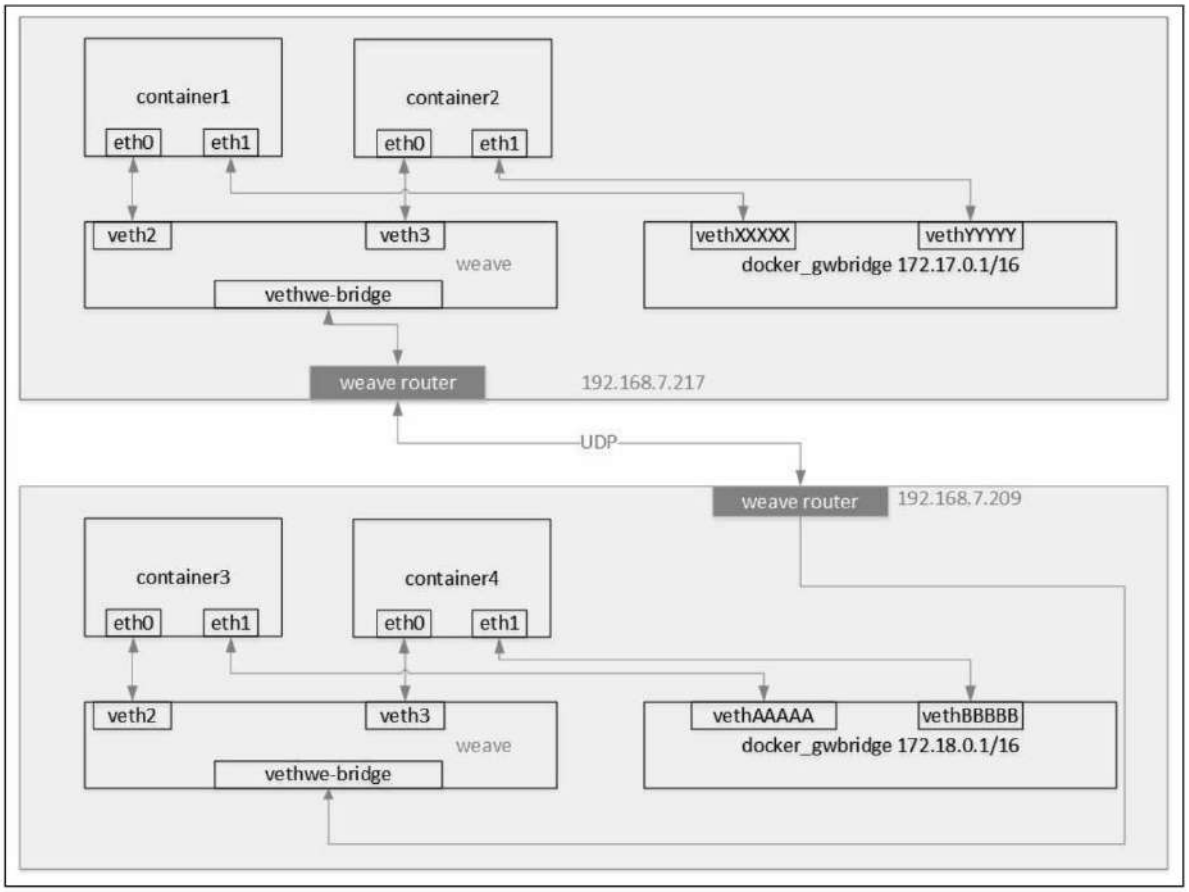


图5-24　Weave跨主机通信

从图5-24可以看出，Sleeve模式下，wRouter承担了将流量转发到对端主

机上的工作，其基本流程如下所示：

（1）容器流量通过veth pair到达宿主机的Weave网桥上，Weave网桥工作在混杂模式。

（2）wRouter通过pcap设备在Weave网桥上截获网络数据包，并排除由内核直接通过网桥转发的数据流量，例如本子网内部、本地容器之间的数据及宿主机和本地容器之间的流量。捕获的包通过UDP转发给对端主机的wRouter。

（3）在接收端，wRouter通过pcap设备将包注入网桥上的接口，并通过网桥上的veth pair将流量分发到容器的网卡上。

5.5.5 Weave的应用示例

先准备两台测试用机器：

```
node-1    103.10.86.238
node-2    103.10.86.239
```

两台机上均安装Docker及Weave，并启动wRouter（安装及启动操作见上文）。最好关闭两台机器的防火墙，如果打开防火墙，需要开放6783端口。我们准备分别在node-1和node-2上创建测试容器。

1.指定容器IP

一般情况下，在Weave节点上有两种方式启动一个应用容器。第一种方式是直接使用weave run命令：

```
# weave run 192.168.0.2/24 -itd docker.io/centos /bin/bash
```

不过，Weave在2.0版本之后就没有weave run命令了，所以推荐使用下面的第二种方式。

第二种方式是先使用docker run启动容器，然后使用weave attach命令给容器绑定IP地址。因此，在node-1机器上启动第一个容器my-test1，容器IP绑定为192.168.0.2。

```
# docker run -itd --name=my-test1 docker.io/centos /bin/bash
06d70049141048798519bfa1292ed81068fc28f1e142a51d22afd8f3fc6d0239
# weave attach 192.168.0.2/24 my-test1
192.168.0.2
```

绑IP时使用容器名称或容器id都可以，上面这个例子中，给my-test1容器绑定的IP为192.168.0.2。

```
# docker exec -ti my-test1 /bin/bash -c ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:acff:fe11:2  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 5559  bytes 11893401 (11.3 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5287  bytes 410268 (400.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ethwe: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1376
        inet 192.168.0.2  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::88b0:ddff:fea2:58c5  prefixlen 64  scopeid 0x20<link>
        ether 8a:b0:dd:a2:58:c5  txqueuelen 0  (Ethernet)
        RX packets 97  bytes 7234 (7.0 KiB)
```

```
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 66  bytes 4316 (4.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
...
```

weave attach命令的反操作weave detach命令表示删除IP绑定。例如：

```
# weave detach 192.168.0.2/24 my-test1
```

在docker run启动容器时，可以添加--net=none参数，表示容器启动后不使用默认的虚拟网卡docker0自动分配的IP，而是使用Weave分配的IP。当然，也可以选择不添加这个参数直接启动容器。这样，容器启动后就会有两个网卡：一个是docker0自动分配的IP，适用于同主机内的容器间通信，即同主机的容器使用docker0分配的IP可以相互通信；另一个就是Weave网桥绑

定的IP。因此，上面的例子中，容器eth0网卡来自Docker分配，ethwe网卡来自Weave分配。

下面测试添加只绑定Weave网卡的场景：

```
# docker run -itd --name=my-test2 docker.io/centos --net=none /bin/bash
8f2ecc2449a0be1f1be2825cb211f275f9adb2109249ab0ff1ced6bbb92dd733
# weave attach 192.168.0.3/24 my-test2

# docker exec -ti my-test2 /bin/bash
# ifconfig
ethwe: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1376
        inet 192.168.0.3  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::3064:8fff:fe3c:909a  prefixlen 64  scopeid 0x20<link>
        ether 32:64:8f:3c:90:9a  txqueuelen 0  (Ethernet)
        RX packets 63  bytes 4734 (4.6 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 34  bytes 2580 (2.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 21  bytes 2352 (2.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
```

```
        TX packets 21  bytes 2352 (2.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
...
```

发现eth0网卡没了。

2.容器互联

默认情况下，上面在node-1和node-2两台宿主机上创建的两个容器间都是相互ping不通的。需要使用weave connect命令在两台Weave的路由器之间建立连接。

```
# weave connect 103.10.86.239
```

weave connect连接的是对方宿主机的IP。weave forget ip则是逆操作，表示断开这个连接。

这时再执行测试就会发现，位于两台不同主机上的相同子网段内的容器之间可以相互ping通，命令如下所示：

```
# docker exec -ti my-test1 /bin/bash
# ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=3.27 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=0.657 ms
.....
```

```
# docker exec -ti my-test2 /bin/bash
# ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.453 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=0.320 ms
.....
```

如果再在node-1上启动容器my-test3，绑定ip为192.168.0.8；在node-2上启动容器my-test4，绑定ip为192.168.0.10，会发现这4个在同一个子网内的容器都可以相互ping通。

接着，启动与上面不在同一个子网内的容器。分别在节点node-1上启动容器my-test4，绑定ip为192.168.10.10；在节点node-2上启动容器my-test5，绑定ip为192.168.10.20。测试网络连通性时会发现在跨主机情况下，相同子网内的容器是可以相互通信的；但处于不同子网的两个容器是不能互联的，尽管这两个容器在同一个主机下也是不能通信的！这样的好处是使用不同子网进行容器的网络隔离。

注意一个细节，在使用Weave时如果使用Docker的原生网络，在容器内部是可以访问宿主机及外部网络的。也就是说，在启动容器的时候，使用了虚拟网卡docker0分配IP。在这种情况下，登录容器后是可以ping通宿主机IP，并且可以对外联网的！

这时，在宿主机上可以ping通docker0网桥的IP，但是ping不通主机上的Weave网桥的IP。在我们的例子中，默认在node-1和node-2宿主机上是ping不通my-test1容器的Weave网桥IP的。这时，可以使用weave expose命令给Weave网桥添加IP，以实现容器与宿主机网络的连通，命令如下所示：

```
# 在node-1和node-2两台机器上添加Weave网桥的ip
# weave expose 192.168.0.1/24
```

注意，这里的192.168.0.1/24是上面my-test1、my-test2、my-test3、my-test4容器的Weave网桥的网关地址。

weave hide 192.168.0.1/24是weave expose的逆操作，表示删除之前的配置。

然后在两台宿主机上ping上面同网段内（192.168.0.0/24）的容器（例如192.168.0.3），发现都可以ping通。

给另一网段（192.168.10.0/24）的容器的Weave网桥添加ip（可以在宿主机上对不同网段的容器的Weave网桥添加ip），在node-1和node-2上分别执行：

```
# weave expose 192.168.10.1/24
```

在node-1和node-2上ping 192.168.10.20都能ping通。

总结，如果不使用Docker的原生网络，即在容器启动的时候，添加--net=none，那么在容器启动后，就不会使用Docker分配IP。这种情况下，登录容器后发现不能访问宿主机及外部网络，而在宿主机上也不能ping通容器IP。这时，添加对应容器网段的Weave网桥IP，就可以实现容器与宿主机网络的连通。但是，此时在容器内部依然不能访问外部网络。所以，可以同时使用Docker的原生网络和Weave网络实现容器互联及容器访问外网和端口映射。使用外部网络及端口映射时就使用docker0网桥，需要容器互联时就使用Weave网桥。每个容器分配两个网卡。

3.查看容器路由信息

使用weave ps可以查看Weave为每个容器设置的路由信息，即容器MAC地址和IP地址的映射关系，例如：

```
# weave ps
weave:expose 06:9d:3b:91:3d:f3 192.168.0.1/24 192.168.10.1/24
```

```
2896b6cad7af 56:46:8c:14:42:e6 192.168.10.10/24
c9aa381c1203 4a:0d:16:4d:bb:c2 192.168.0.8/24
00efd39d3a7d 8a:b0:dd:a2:58:c5 192.168.0.2/24
```

如上所示，weave ps输出的第一栏是容器ID，第二栏是容器MAC地址，第三栏是容器IP地址。

weave ps的输出在每个节点上各不相同。

4.Weave的其他特性

应用隔离。不同子网容器之间默认是隔离的，即便它们位于同一台物理机上也相互不通（使用-icc=false关闭容器互通）；不同物理机之间的容器默认也是隔离的。

安全性。可以通过weave launch-password wEaVe设置一个密码，用于weave peers之间的加密通信。

服务发现。Weave网络通过在每个节点上启动一个“微型的DNS”服务实现服务发现。你只需要给你的容器起个名字就可以使用服务发现，还可以在多个同名的容器上提供负载均衡的功能。每个容器都可以通过域名与另外的容器通信，也可以直接通信，无须使用NAT，也不需要使用端口映射。

无须额外的集群存储。所有其他的容器网络插件，包括flannel、Calico和Docker自带的overlay驱动，在你真正能使用它们之前，都需要安装额外的集群存储（一个类似于etcd、Consul或者ZooKeeper的分布式数据库）。除了维护和管理的成本，集群节点强依赖于与集群存储的连接，如果断开了与其的连接，即使很短暂你也无法再配置停止容器的网络。Weave网络是与Docker网络插件捆绑在一起的，这意味着你可以马上就使用它，而且可以在网络连接出现问题时依旧启动和停止容器。

性能。Weave overlay通过pcap捕获包这种方式完成封装和解封装这个过程，这个过程中需要将数据包从内核态复制到用户态，然后按照自定义的格式完成封装和解封装，因此性能不高。

Weave网络的Fast Datapath模式自动在两个节点之间选择最快的路径，提供接近本地网络的吞吐量和延迟，而且不需要你的干预。

组播支持。Weave网络完全支持组播地址和路径。数据可以被发送给一个组播地址，数据的副本可以被自动地广播。

总结Weave的优劣势如下：

·支持主机间通信加密；

·支持容器动态加入或者移除网络；

·支持跨主机多子网通信。

5.WeaveScope网络监控

WeaveScope可以对Docker运行网络进行监控，安装方法如下：

```
# curl -L git.io/scope -o /usr/local/bin/scope
# chmod a+x /usr/local/bin/scope
# scope
```

然后，浏览器访问按上面的scope启动后的提示信息，例如http://localhost:4040。

对每一个参与scope监视的节点也需要安装scope软件。

5.5.6 小结

Weave控制面在实现上与Calico类似，数据面在1.2版本前使用userspace实现，即通过UDP封装实现L2 overlay。Weave在1.2版本后，结合了Linux内核的Open vSwitch datapata（odp）和VXLAN，在网络性能上有较大的提升。由于odp和VXLAN与内核相关模块结合较为紧密，因此在实际使用过程中可能会遇到一些和内核相关的问题。

参考资料：

［1］https://blog.csdn.net/cloudman6/article/details/77856911.

［2］https://www.bladewan.com/2017/12/15/docker_weave/.

5.6 Cilium：为微服务网络连接安全而生

什么是Cilium？Cilium是“纤毛”的意思，它十分细小又无处不在。Cilium官方的定位是：

API-aware Networking and Security plugin，helping Linux Secure Microservices.

Cilium是一个具备API感知的网络和安全的开源软件，该软件用于透明地保护使用Linux容器管理平台（如Docker和Kubernetes）部署的应用程序服务之间的网络连接。Cilium以API为粒度的安全功能如图5-25所示。

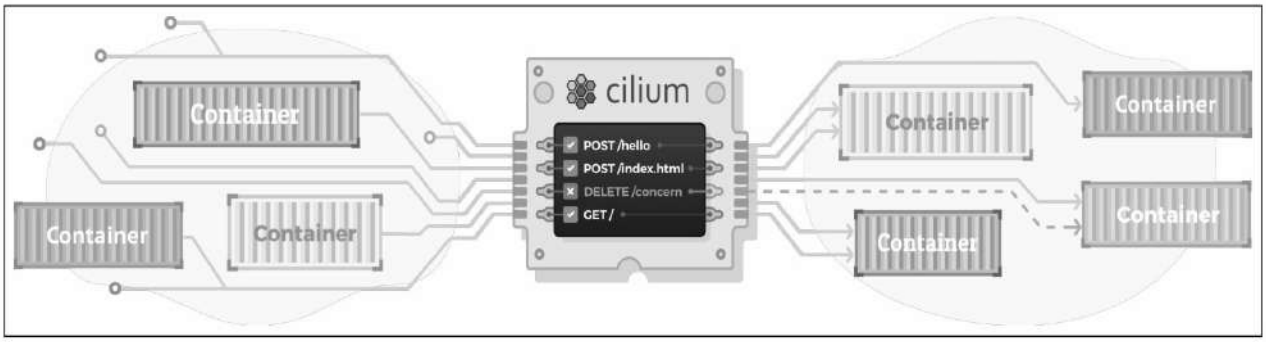


图5-25　Cilium以API为粒度的安全功能

下面我们将重点介绍Cilium独特的基于API感知网络的安全过滤特性。虽然Cilium也提供容器的网络连通方案，但没有特别之处，因此将简单带过。

5.6.1 为什么使用Cilium

在回答为什么使用Cilium这个问题之前，我们先来探讨为什么从单机时代便广泛应用的iptables在微服务时代显得有些力不从心？

1.iptables在微服务时代的限制

作为通用操作系统的一个组件，iptables专注于为Linux管理员提供系统安全性管理的“瑞士军刀”，即基于静态环境的IP和端口配置网络转发、过滤等规则。然而，随着iptables在大规模、高度动态的微服务环境中投入使用，原始设计目标与现代基础设施需求之间的不匹配愈发明显。前文在对Kube-proxy转发模式讨论时就提到了基于iptables服务负载均衡的严重性能瓶颈。除了性能和规模，由于容器的创建和销毁非常频繁，基于IP做身份关联的故障排除和安全审计等功能很难实现。

现代数据中心应用程序的开发已经转向面向服务的体系结构（SOA），

即我们常说的微服务。基于微服务的应用程序被拆分为一个个独立的小型服务，这些服务使用HTTP、gRPC和Kafka等轻量级协议，通过API相互通信。但是，现有的Linux网络安全机制（例如iptables）仅在网络和传输层（即IP地址和端口）上运行，并且缺乏对微服务层的可见性（visibility）。

微服务架构下的应用程序，尤其是通过容器部署的是高度动态变化的。在向高度动态的微服务架构的转变过程中，确实给微服务之间的连接安全性提出了挑战和机遇。具体表现为：传统的Linux网络安全方法（例如iptables）过滤IP地址和TCP/UDP端口，但容器高度不稳定的生命周期导致这些方法难以与应用程序并排扩展。因为负载均衡和访问控制列表要不断更新，系统可能要维护成千上万条规则，这给运维带来了较大负担。出于更精细的安全考虑，协议端口不能再用于区分应用流量，因为同一端口可能承载跨服务的各种消息。另一个挑战是提供准确的可见性，因为传统系统使用IP地址作为主要识别工具，而IP在微服务架构中的寿命可能才几秒。

归根结底，通用的基于IP/端口的防火墙方式在微服务架构中的网络和安全面临一系列限制。因此我们不禁发问：如果在微服务时代从头开始设计内核Linux网络和安全方法会是什么样子？

2.BPF：让**Linux**感知微服务

幸运的是，我们拥有BPF（Berkeley Packet Filter）。一句话总结BPF就是：BPF是Linux内核中的一个高性能沙箱虚拟机，它将内核变成了可编程的。

作为一种Linux内核的黑科技，BPF可以在不影响安全性或性能的情况下扩展Linux内核，提供了内核的数据包过滤机制。跟netfilter和tc等一样，BPF是一个框架，用于在内核中的各个挂钩点运行自定义逻辑，这其中就包括Linux网络协议栈中的多处挂载点。这也是那些基于BPF实现的profiling和tracing（例如tcpdump）工具的工作原理。

BPF给用户提供两种SOCKET选项：SO_ATTACH_FILTER和SO_ATTACH_BPF，允许用户在sokcet上添加自定义的filter，只有满足该filter指定条件的数据包才会上发到用户空间。SO_ATTACH_FILTER插入的是cBPF（classic Berkeley Packet Filter，即经典BPF，我们说的BPF值就是cBPF）代码，SO_ATTACH_BPF插入的是eBPF（extended Berkeley Packet Filter，即扩展BPF）代码。

从Linux 3.15开始，eBPF被引入内核，eBPF扩充了cBPF的功能，丰富了指令集但保留了对cBPF的兼容。例如，tcpdump还是用的cBPF，但cBPF字节码被加载到内核后会被内核自动转换为eBPF字节码。

注：若不特殊说明，本书不区分cBPF和eBPF，统一用BPF指代。

BPF的工作原理是在内核提供了一个虚拟机，用户态将过滤规则以虚拟

机指令的形式传递到内核，由内核根据这些指令来过滤网络数据包。BPF逻辑被编写为简单的“BPF程序”，它们在运行前先要通过验证，以确保它们在任何情况下都不会导致运行它的内核崩溃。验证之后，这些程序被一个JIT（just in time）编译器编译成CPU相关的代码（例如X86）在内核态运行，这意味着它们以与编译到内核中的代码相同的速度运行。最重要的是，任何网络报文都没法绕过BPF在内核态的限制。

要理解BPF的作用，首先要意识到Linux内核本质上是事件驱动的！写数据到磁盘，读写socket，请求定时器等，这些过程都是系统调用，都是事件驱动的。世界上最大的单体应用，有着1000万行代码量的Linux无时无刻不在处理各种事件。

BPF给我们提供了在事件发生时运行指定的BPF程序的能力。例如，我们可以在以下事件发生时运行我们的BPF程序：

·应用发起read/write/connect等系统调用；

·TCP发生重传；

·网络包达到网卡。

BPF在过去几年中发展迅速，Netflix、Facebook和Google等在Linux上做了大量投资的公司，都在积极探索使用BPF作为内核的可扩展性机制，把Linux打造成一个可感知微服务的操作系统。为什么这么说呢？BPF能够使Linux内核感知到API层。当内核能够理解两个应用程序通信过程中调用了哪些API时，它便能够为API调用提供安全保障，并以此构建一个基于身份认证的机制。因此，不同于以前简单的IP+Port过滤网络包的方式，有了BPF加持的Linux内核可以理解什么是一个微服务，微服务的标签有哪些，这个微服务的安全性是怎么样的。

3.Cilium：把BPF带到Kubernetes

首先，Cilium是一个CNI插件，它提供网络连通性、服务负载均衡等功能，但主打的功能还是安全。例如，Cilium实现了Kubernetes的网络策略API，还提供了基于身份认证的微服务安全机制。

从一开始，Cilium就是为大规模、高度动态的容器环境而设计的。Cilium在3/4层运行，除了提供传统的网络和安全服务，还有一些L7的负载均衡和流量过滤功能。区别于传统系统中的IP地址识别，Cilium原生地了解服务/容器/Pod标识，并解析HTTP、gRPC和Kafka等协议，提供比传统防火墙更简单、更强大的可见性和安全性。

BPF的高效灵活是Cilium的基础。Cilium支持在各种集成点（例如，网络IO、应用程序套接字和跟踪点）中将BPF字节码动态插入Linux内核，为工作负载（包括进程和容器）提供透明的网络连接保护、负载均衡、安全和

可观测性支持。最关键的是，BPF的使用使得Cilium能够以高度可扩展的方式实现以上功能，尤其能够应对大规模的微服务场景。

Cilium基于BPF，但为用户隐藏了BPF的复杂性，提供了与通用编排框架（例如Kubernetes等）Mesos的集成。Cilium的工作原理如图5-26所示。

BPF的强大功能可实现高效的内核数据转发，为常见的微服务用例提供巨大的性能优势，例如Cilium就可以作为Kubernetes服务负载均衡（iptables或IPVS）和Istio本地代理（Envoy）的可选项。

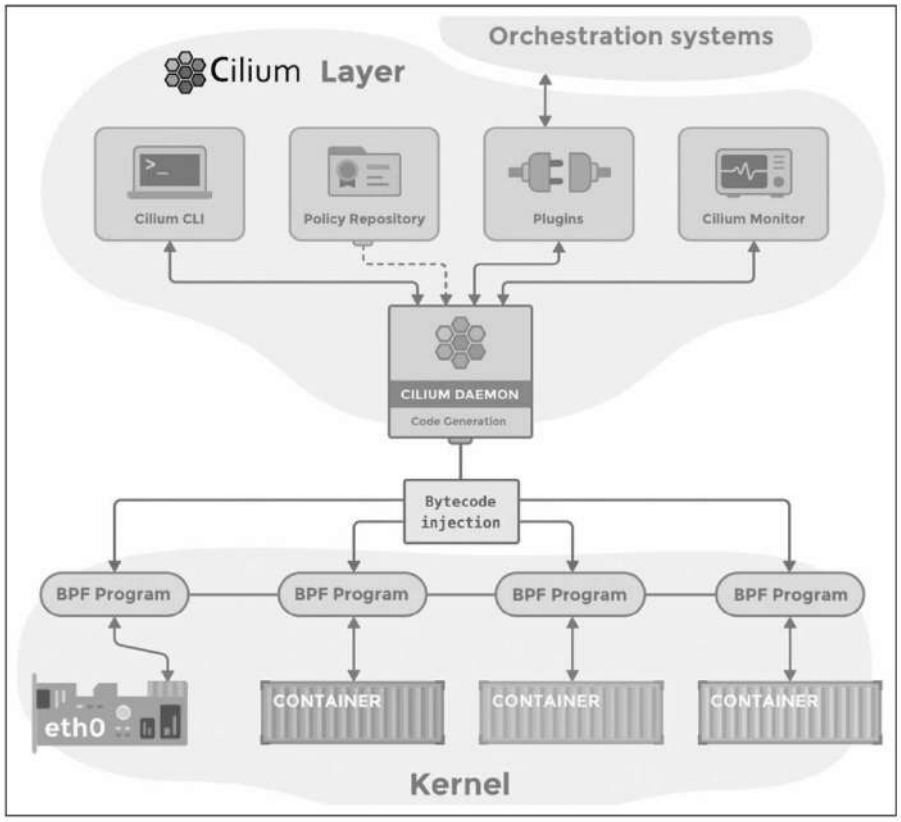


图5-26　Cilium的工作原理

4.Cilium功能一览

具体来说，Cilium实现了以下功能。

1）容器的网络连接

Cilium的网络模型较简单，即一个三层网络空间为所有服务端点提供链接，并通过策略层实现安全控制。Cilium支持以下跨节点网络模型。

·overlay：基于封装的虚拟网络产生所有主机。目前，已支持基于VXLAN和Geneve等封包协议；

·直接路由：使用Linux主机内置或云提供商的路由表，通过底层网络路由应用程序容器的IP地址。

从架构上看，Cilium将安全与网络寻址进行解耦，极大简化了网络模

型，也提高了扩展性，降低了排错难度。

2）基于策略的网络安全

Cilium同时提供基于数据包和API的网络安全与认证，为传统部署和微服务架构提供安全的网络连接。Cilium的网络策略分为以下几大类。

·基于身份：在每个包内，Cilium将负载和身份信息打包在一起（而不是依靠源IP地址），提供高可扩展安全性；

·基于IP/CIDR：如果基于身份的方式不适用，那么可以采用基于IP/CIDR安全的方式控制安全访问。Cilium建议在安全策略中尽量采用抽象方式，避免写入具体IP地址。例如，使用Kubernetes的Service（通过label selector选择后端Pod）名；

·基于API：HTTP/REST、gRPC和Kafka通过暴露IP和端口对外提供服务，其安全机制明显不足。基于API的网络策略允许使用更细粒度的安全限制，例如REST方法等。下文会详细展开介绍Cilium的安全机制，这里不再赘述。

3）分布式可扩展负载均衡

BPF提供高性能L3/L4的服务转发与负载均衡，转发策略有rr、wrr、源hash等。基于散列表实现的BPF提供O（1）时间复杂度的路由性能（这一点与IPVS很像），也就是说，Cilium可以替换Kube-proxy实现Kubernetes Service机制。这样所有Kubernetes集群IP服务会自动在BPF中得到高效地实现，而且性能不会随着服务数量的增加而下降。

4）可视化

与网络策略类似，Cilium也同时在网络包和API调用两个层面实现了可视化。所有可视化信息，不仅是IP地址和端口号，还包括丰富的工作流元数据，例如容器和Pod标签、服务名等。这样一来，Cilium就能提供基于标签、安全身份和事件类型的过滤和可视化。

Cilium的可视化基于BPF高性能循环缓冲区（perf ring buffer），可以追踪每秒百万级的应用事件。除此之外，还利用BPF可编程性的高效通道允许数据可视化同时降低额外负担。

最后，Cilium的所有可视化都对外提供API，可以嵌入现有系统中。

5）监控

Cilium周期性地监控集群连接状态，包括节点之间延迟、节点失效和底层网络问题等，并且可以将Cilium整合到Prometheus监控系统中。

总的来说，Cilium使用BPF作为底层引擎，创建了一个精确优化的网络堆栈，用于在Kubernetes等平台上运行API驱动的微服务。下文将重点讨论

使用Cilium带来的两个主要好处：

·不只简单关注数据包、IP地址和端口，而是将服务标识和API协议（例如HTTP、gRPC和Kafka）视为平台中的一等公民；

·针对在微服务环境中越来越常见的规模、动态和部署模式（例如服务网格代理）优化Linux网络转发、可见性和过滤。

5.6.2 以API为中心的微服务安全

微服务架构下大多数通信（例如HTTP）只使用少量端口，并且服务之间通信的实际“意图”只能通过理解服务之间的远端过程调用（RPC）来确定。

典型的微服务端点暴露RPC调用的典型场景有：

·RESTful基于HTTP的服务经常为许多不同的资源类型公开POST/PUT/GET/DELETE端点，并且每种资源类型都由URL前缀表示；

·Kafka通常会有许多不同的主题，并允许将每个主题的生产和消费等操作发送给不同的客户；

·Cassandra、Elasticsearch、MongoDB、Redis、MySQL和Postgres等数据存储提供对不同表/索引的读写访问。

因此，基于端口的可见性和安全性对单个RPC的调用做策略是盲目的，其只能带来两种操作：完全暴露两个不同服务之间的所有RPC，或根本不暴露任何RPC。

前文提到，Cilium基于身份认证提供了微服务API层面的安全机制，而不是传统的基于IP和端口的。换句话说，Cilium允许用户给服务指定一个身份（identity），实现基于服务的标签来定义安全策略。

使用Cilium，微服务的身份从容器管理平台处提取并嵌入每个网络请求中（例如id=app1）。与IP地址不同，这种身份信息在服务的整个生命周期内也是一致的，并且在服务的多个副本之间是一致的。如图5-27所示，Cilium提供了API感知的可见性和安全性，可以理解HTTP方法/URL、gRPC服务方法、Kafka主题等，并实现单个RPC细粒度上的可见性和安全性。例如，只允许部分REST API调用，只允许访问Kafka集群，或只能生产或消费特定主题的消息等。

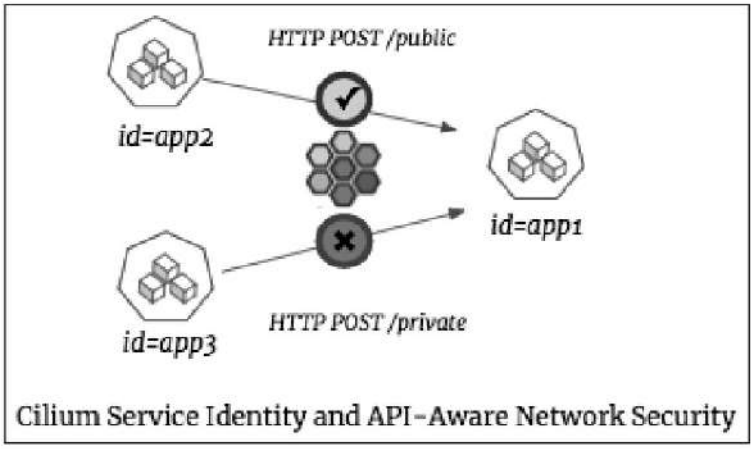


图5-27　Cilium提供了API感知的可见性和安全性

下面是一个使用Cilium　Microscope查看微服务app1的所有HTTP请求并记录安全策略是允许还是拒绝请求的示例：

```
Cilium-microscope --to-selector id=app1 --type l7
([k8s:id=app2]) => ([k8s:id=app1]) http GET http://app1-service/public Forwarded
([k8s:id=app3]) => ([k8s:id=app1]) http GET http://app1-service/private Denied
...
```

使用Cilium，用户无须操心有多少容器副本正在实现服务app1，这些容器运行在哪个主机上，或者容器分配的IP地址等问题。

5.6.3　BPF优化的数据平面性能

Cilium使用的BPF技术可以在数据转发的许多方面产生一些惊人的好处，但本节我们只关注以下两个方面。

1.Kubernetes服务负载平衡

前文做了iptables进行服务转发的性能测试，发现其已成为Kubernetes服务转发的瓶颈，并指出部署5000个服务时吞吐量降低了约30%，而部署10000个服务时的吞吐量降低了80%。测试还发现，刷新5000个服务的iptables规则花费了整整11分钟。

IPVS相对iptables可以极大提升性能，但IPVS最大的问题是过于通用且很难扩展新特性，还依赖iptables和ipset做包过滤和SNAT等。

Cilium使用的BPF技术底层也是使用的散列表数据结构，因此查找和更新服务路由规则的时间复杂度都是O（1），这意味着即使有10000+服务，更新转发规则的时延也是微秒级，这个数据和IPVS很像。Cilium也实现了Kubernetes　Service的全部功能，因此Cilium可以作为Kube-proxy的一个替代

方案。

2.服务网格加速

微服务的新兴部署模式是使用本地代理（作为每个Pod的sidecar代理或每个Linux主机的一个代理运行）在一组微服务之间实现服务网格（Service Mesh），典型实现是Istio+Envoy。

更多关于Service Mesh和sidecar的介绍请见第6章。

Service Mesh中，两个服务并不是直接和彼此通信，而是通过各自的sidecar。服务出来的请求经过协议栈和iptables规则进入sidecar的监听socket，这个TCP连接到这里就是终点了。sidecar会将请求收进来，检查HTTP头和其他信息，做一些它自己的处理后再将请求转发给真正的业务服务器。

必须指出的是，以上这个过程并不高效。微服务间通信加上这一层sidecar会有10倍的性能损耗，然而这并不是sidecar（例如Envoy）本身的性能造成的，而是sidecar代理的工作方式造成的。

Istio目前使用iptables在数据包级别执行此重定向。但是在数据包级别进行此重定向意味着每个数据字节都会通过整个TCP/IP堆栈，Linux会执行TCP拥塞控制，将二层数据帧切分成IP数据包，在某些情况下，甚至会通过虚拟以太网设备传递数据。缓冲存储器、CPU周期、上下文切换和数据包延迟方面的成本非常高。然而在同一主机上，将数据从一个Linux套接字复制到另一个Linux套接字可以做到非常高效。因此，一个直观的想法是如果服务和sidecar运行在同一台宿主机上，那么我们可以直接在两个socket之间复制数据，这将带来极大的性能提升（3～4倍）。这也是Cilium和BPF使Linux内核可感知微服务的一个例子。

幸运的是，Linux内核对BPF进行了一项称为“Sockmap”的改进，Cilium通过它能够在套接字层安全地进行数据的重定向。iptables、loopback、Cilium的转发效率对比如图5-28所示。

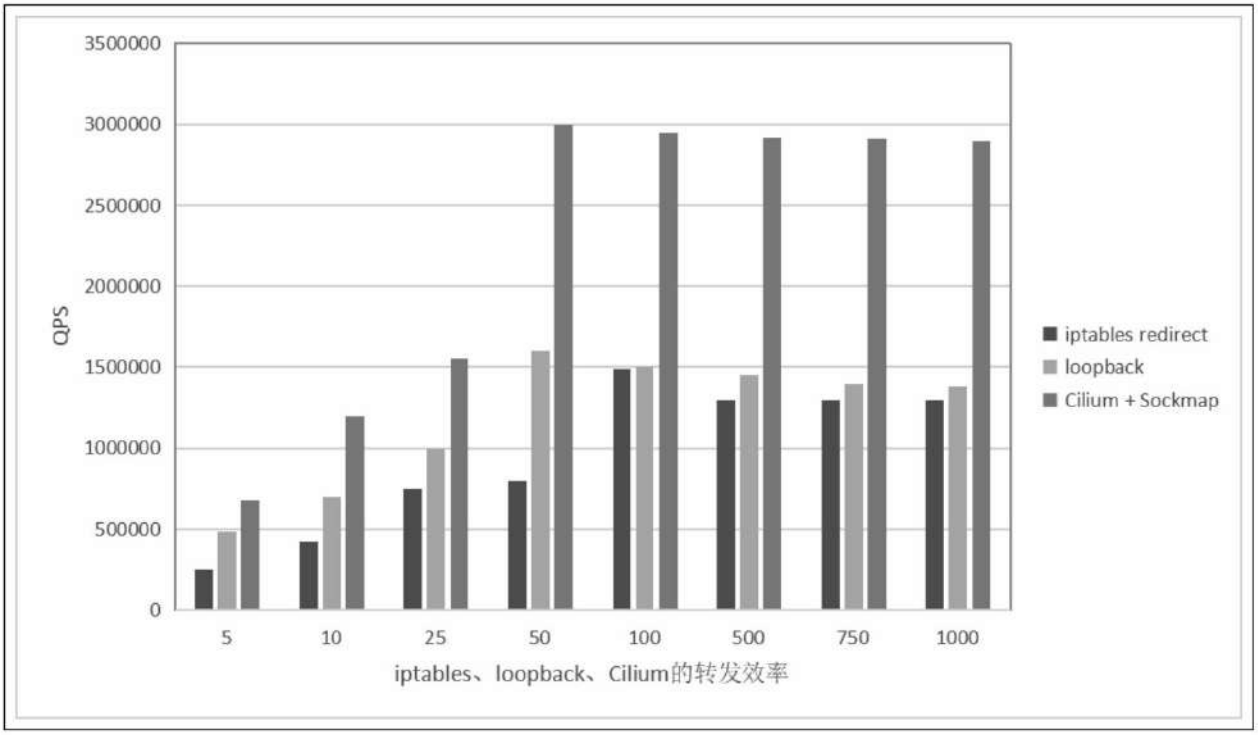


图5-28　iptables、loopback、Cilium的转发效率对比

不难发现，使用Sockmap优化后的BPF吞吐量约为iptables重定向的2倍，甚至比loopback（直接访问本地回环地址）还要高不少。

简而言之，如果采用Service Mesh这种架构，那么使用Cilium+Sockmap应该是减少CPU/内存使用和降低延迟的一种简单方法。

5.6.4 试用Cilium：网络策略

Cilium简单易用，在Kubernetes中安装Cilium只需一条kubectl create命令，如下所示：

```
# curl -sLO https://releases.cilium.io/v1.0.0/examples/kubernetes/cilium.yaml
# kubectl create -f cilium.yaml
```

前文提到，Cilium除了实现Kubernetes的NetworkPolicy API，还定义了一套自己的网络安全策略。下面举几个简单的例子。

1.TCP策略

Cilium提供的基于TCP的网络策略的一个例子如下：

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L3-L4 policy to restrict deathstar access to empire ships only"
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

2.基于网段的策略

Cilium提供的基于网段的网络策略的一个例子如下：

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "cidr-rule"
spec:
  endpointSelector:
    matchLabels:
      app: myService
  egress:
  - toCIDR:
    - 20.1.1.1/32
  - toCIDRSet:
    - cidr: 10.0.0.0/8
      except:
      - 10.96.0.0/12
```

3.HTTP策略

Cilium提供的基于HTTP的网络策略的一个例子如下：

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
description: "L7 policy to restrict access to specific HTTP call"
metadata:
  name: "rule1"
spec:
  endpointSelector:
    matchLabels:
      org: empire
      class: deathstar
  ingress:
  - fromEndpoints:
    - matchLabels:
        org: empire
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
      rules:
        http:
        - method: "POST"
          path: "/v1/request-landing"
```

5.6.5 小结

在云原生带来的微服务浪潮下，尽管几乎所有关于如何设计和运行应用程序的内容都在变化，但像iptables（基于内核的netfilter）这样的内核功能仍然是Kubernetes、Mesos、Docker等现代微服务环境中网络路由、包过滤、安全性和记录网络数据的最常用工具。

然而，在高度动态和复杂的微服务世界中，仅仅通过IP地址和端口的传统镜头来思考网络和安全性会导致实现效率非常低，只能实现过程可见性和过滤，并且通常非常复杂且难以排查。由于BPF是Linux内部强大的新内核可扩展性机制，使我们有机会在微服务时代重新思考Linux网络和安全堆栈

并解决这些问题。

Cilium通过将安全性与寻址分离，不仅可以在高度动态的环境中应用安全策略，还提供了除传统的L3/L4网络隔离外的应用层安全限制。Cilium将不再基于传统的“IP+Port”的方式做网络策略，而是基于“身份”，这将带来可见性和安全性，使用起来也更简单，而且功能上更加强大（支持对单个RPC调用的细粒度控制）。

参考资料：http://Cilium.readthedocs.io/en/stable/bpf/.

5.7 Kubernetes多网络的先行者：CNI-Genie

Kubernetes采用的CNI标准，让Kubernetes生态系统中的网络解决方案百花齐放。更多样的选择，意味着大多数用户将能够找到适合其当前需求和部署环境的CNI插件，同时还可以在环境发生变化时找到新的解决方案。

不同企业之间的运营要求差异很大，因此拥有一系列具有不同复杂程度和功能丰富性的成熟解决方案，有助于Kubernetes在满足不同用户独特需求的前提下，仍然能够提供一致的用户体验。很多情况下，这一套解决方案不是某个单一的网络插件能够提供的，例如flannel虽然易用，但不支持网络策略。因此一个直观的想法是能不能在同一个容器集群中集成多个网络插件，博采众长？

下面介绍一个由华为开源的多网络插件：CNI-Genie。CNI-Genie是一个集成引导插件，本身无具体功能，由引用的插件完成网络功能，支持flannel、Calico、Weave Net、Canal、Romana等CNI插件，还支持SR-IOV、DPDK等。值得一提的是，CNI-Genie本身也是一个符合CNI标准的容器网络插件。

CNI-Genie的工作原理如图5-29所示。

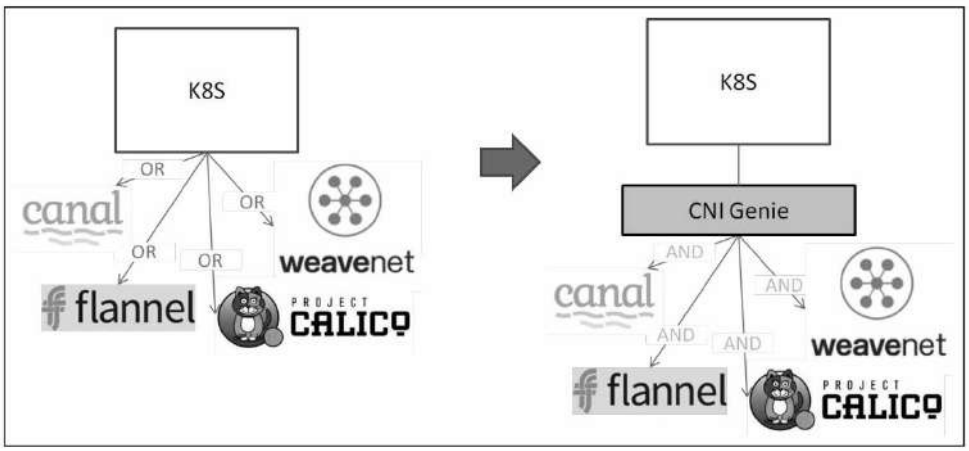


图5-29 CNI-Genie的工作原理

从图中不难发现，CNI-Genie本质上就是Kubernetes和底层多个CNI插件之间的适配器（adapter）。

5.7.1 为什么需要CNI-Genie

在传统的基于CNI的网络中，Kubernetes允许使用单个CNI插件来满足容器网络的所有需求。Kubelet以--network-plugin=cni--cni-conf-dir=/etc/cni/net.d启动。CNI-Genie允许运行时多个CNI插件共存，而且用户可以根据实时的工作负载在不同的（物理或逻辑）网络之间切换使用网络插件。CNI-Genie还支持使用Kubernetes　CRD（自定义资源定义）自定义逻辑网络和网络策略。而Kubernetes Pod对多个网络平面的选择通过其annotations定义。

在特定场景中，用户可能必须使用多个CNI共同实现单个网络可能无法提供的功能。CNI-Genie使用户能够在同一个集群中运行多个CNI，并有助于为每个容器创建多个接口。事实上，多网络平面是CNI-Genie的一个重要功能。需要注意的是，使用CNI-Genie所集成的网络插件才是容器多网络平面和多网卡（IP地址）的真正提供者。CNI-Genie在Kubernetes上的工作流如图5-30所示。

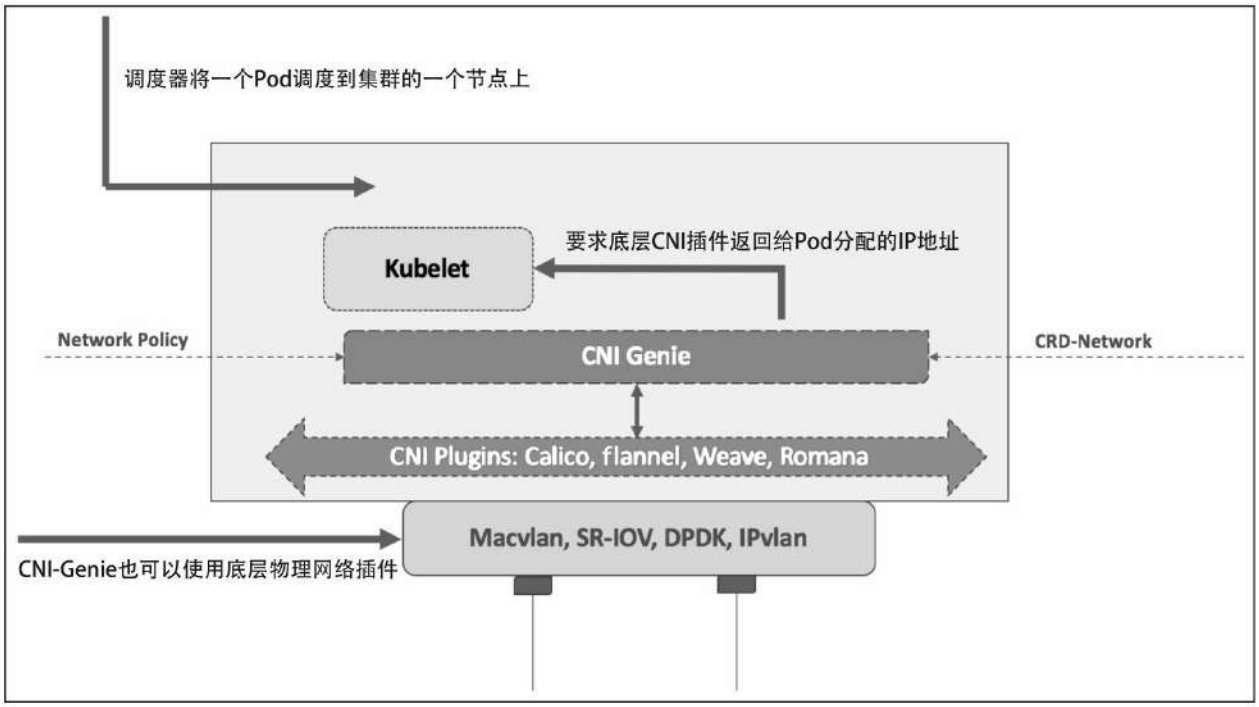


图5-30　CNI-Genie在Kubernetes上的工作流

与其他需要与Kubernetes交互的CNI插件一样，CNI-Genie的配置文件需要指定kubeconfig的位置，以便与Kubernetes API Server通信。

CNI-Genie的配置文件如下所示：

```
{
    "name": "k8s-pod-network",
    "type": "genie",
    "log_level": "info",
    "datastore_type": "kubernetes",
    "hostname": "host1",
    "policy": {
        "type": "k8s",
        "k8s_auth_token": "__SERVICEACCOUNT_TOKEN__"
    },
    "kubernetes": {
        "k8s_api_root": "https://10.96.0.1:443",
        "kubeconfig": "/etc/cni/net.d/genie-kubeconfig"
        },
    "default_plugin": "calico,flannel",
    "cniVersion": "0.3.0"
}
```

需要注意的是，这里CNI-Genie的配置文件是00-**.conf，而其他插件的配置文件是10-**.conf，因此按优先级默认先加载的插件是CNI-Genie。

5.7.2 CNI-Genie功能速递

下面简单介绍CNI-Genie的几个常用功能。

自定义网络插件

CNI-Genie通过Kubernetes Pod的annotations指定其要使用的底层CNI插件，示例如下：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mytest
  namespace: mynamespace
  labels:
    name: mytest
spec:
  replicas: 2
  template:
    metadata:
      name: mytest
      namespace: mynamespace
      labels:
        name: mytest
      annotations:
        cni: "calico"
    spec:
      containers:
        - name: mytest
          image: busybox:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
```

这里annotations添加了cni字段表示该Deployment对应的Pod使用的网络插件，在我们这个例子中是Calico。

5.7.3 容器多IP

此功能允许用户创建具有多个网络接口（除了本地回环地址）的容器，并关联来自多个CNI插件的IP地址。此功能可以满足多租户架构和路由器/防火墙应用程序等需要多个网卡的场景。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: mytest
  namespace: mynamespace
  labels:
    name: mytest
spec:
  replicas: 2
  template:
    metadata:
      name: mytest
      namespace: mynamespace
      labels:
        name: mytest
      annotations:
        cni: "calico,weave"
        multi-ip-preferences: |
          {
            "multi_entry": 0,
            "ips": {
              "": {
                "ip": "",
                "interface": ""
              }
            }
          }
    spec:
      containers:
        - name: mytest
          image: busybox:latest
          imagePullPolicy: IfNotPresent
```

```
          ports:
            - containerPort: 80
```

如上所示，我们给Deployment对应的Pod指定了两个网络接口，分别由Calico和flannel初始化并分配IP地址。

第6章　Kubernetes网络下半场：Istio

6.1　微服务架构的大地震：sidecar模式

在Kubernetes逐渐普及的时代，Service　Mesh技术已完全取代了使用软件库实现网络运维的方式。严格来说，Service　Mesh并不在Kubernetes的核心范围之内。但是，在Kubernetes的帮助下，应用上云后，还面临着服务治理的难题。现在，大多数云原生的应用都是微服务架构，微服务的注册。服务之间的相互调用关系，服务异常后的熔断、降级，调用链的跟踪、分析等一系列现实问题摆在各机构面前。Service　Mesh就是解决这类微服务发现和治理问题的一个概念。

在我看来，Service　Mesh之于微服务架构就像TCP之于Web应用。我们大部分人写Web应用，关心的是RESTful、HTTP等上层协议，很少操心网络报文超时重传、分割组装、内容校验等底层细节。正是因为有了Service Mesh，企业只需在云原生和微服务架构的实践道路上根据自己的业务做适当的微服务拆分，无须过多关注底层服务发现和治理的复杂度。

而Istio的出现，使得有些“学院派”的Service　Mesh概念真正得到了落地。Istio提供了真正可供操作、非侵入式的方案，相对于Spring　Cloud、Dubbo这些SDK方式让人有种耳目一新的感觉。

在传统的MVC三层Web应用程序架构下，服务之间的通信并不复杂，在应用程序内部自己管理即可，但是在现今的复杂的大型网站情况下，单体应用被分解为众多的微服务，服务之间的依赖和通信十分复杂，出现了Twitter开发的Finagle，Netflix开发的Hystrix和Google开发的Stubby这样的“胖客户端”库，这些就是早期的Service Mesh，但是它们都适用于特定的环境和特定的开发语言，并不能作为平台级的支持。

随着Cloud Native概念的流行，容器和Kubernetes的组合增强了应用的横向扩容能力，用户可以轻松地编排出依赖关系复杂的应用程序。同时，开发者无须过分关心应用程序的监控、扩展性、服务发现和分布式追踪这些烦琐的事情，只需专注于程序开发。

说到Isito，不得不提Envoy。Envoy对Service　Mesh最大的贡献就是定义了xDS，Envoy虽然本质上是一个proxy，但是它的配置协议被众多开源软件支持，如Istio、Linkerd、AWS App Mesh、SOFA Mesh等。

下文我们将深入探讨Service Mesh及其实现。

6.1.1 你真的需要Service Mesh吗

云原生时代，微服务化似乎就是一个真命题，唯一值得探讨的就是服务分割的力度。但是，软件架构师需要问自己的一个问题是，我真的需要服务治理吗？或者我真的需要借助Service Mesh治理服务流量吗？

正如Nginx在其官方博客上发表的一篇名叫“Do I Need a Service Mesh？”的文章中提到的：“随着应用程序复杂性的增加，服务网格将成为实现服务到服务的能力的现实选择。”Service Mesh与应用复杂度的关系如图6-1所示。

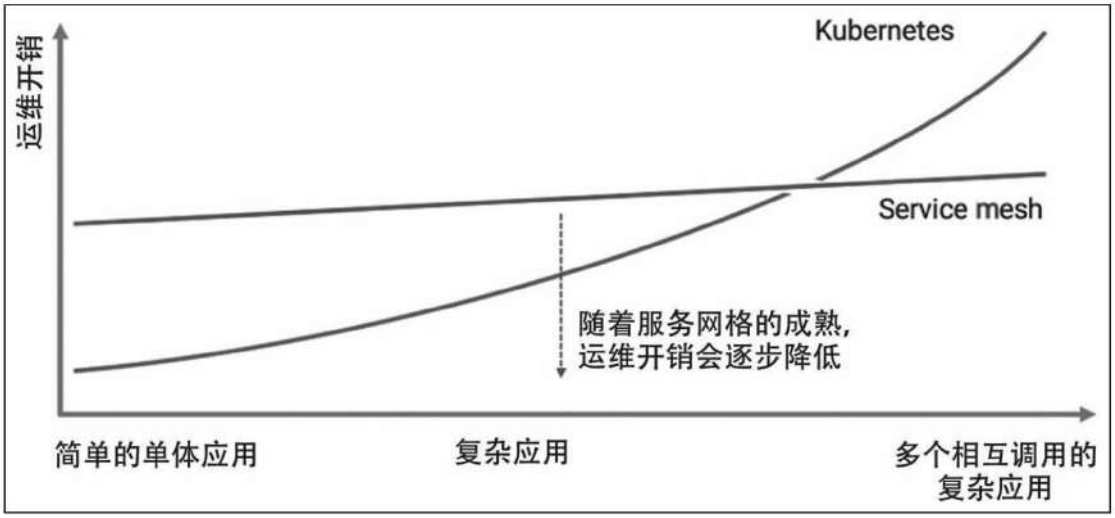


图6-1 Service Mesh与应用复杂度的关系

随着我们的微服务越来越细分，我们所要管理的服务正在成倍增长，Kubernetes提供了丰富的功能，使得我们可以快速地部署和调度这些服务。同时，以我们熟悉的方式来实现那些复杂的功能，但是当临界点到来时，可能就是我们真正要考虑使用Service Mesh的时候了。

是的，只有在服务数量和服务间调用的复杂度上升到一定程度后，Service Mesh才会真正派上用场。

6.1.2 sidecar模式

所谓sidecar模式，翻译过来就是边车模式，是一种分布式和微服务架构的设计模式，目的是实现了控制和逻辑的分离与解耦。因为其非常类似于生活中的边三轮摩托车而得名。边车加装在摩托车旁起拓展功能，比如行驶更稳定，可以拉更多的人和货物，坐在边车上的人可以给驾驶员指路。

软件设计中的sidecar模式通过给应用服务加装一个“边车”达到控制和逻辑分离的目的。该设计模式通过给应用程序加上一个“边车”的方式拓展应用

程序现有的功能，例如日志记录、监控、流量控制、服务注册、服务发现、服务限流、服务熔断等在业务服务中不需要实现的控制面功能，可以交给“边车”，业务服务只需要专注于实现业务逻辑即可。

边车模式出现得很早，实现的方式也多种多样。这个模式随着Service Mesh的逐渐成熟进入人们的视野。

在Azure Architecture Center的云设计模式中是这么介绍边车模式的：

> Deploy components of an application into a separate process or container to provide isolation and encapsulation.

sidecar模式一般有两种实现方式：

·通过SDK的形式，在开发时引入该软件包依赖，使其与业务服务集成起来。这种方法可以与应用密切集成，提高资源利用率并且提高应用性能，但也对代码有侵入，受到编程语言和软件开发人员水平的限制；

·agent形式。服务所有的通信都是通过这个agent代理的，这个agent同服务一起部署，和服务一起有着相同的生命周期创建。这种方式对应用服务没有侵入性，不受编程语言和开发人员水平的限制，做到了控制与逻辑分开部署。但是会增加应用延迟，并且管理和部署的复杂度会增加。

6.1.3 Service Mesh与sidecar

Service Mesh作为sidecar运行时，对应用程序来说是透明的，所有应用程序间的流量都会通过sidecar，然后由sidecar转发给应用程序。换句话说，由于sidecar劫持了流量，所以对应用程序流量的控制都可以在sidecar中实现。sidecar模式下的Service Mesh的基本原理如图6-2所示。

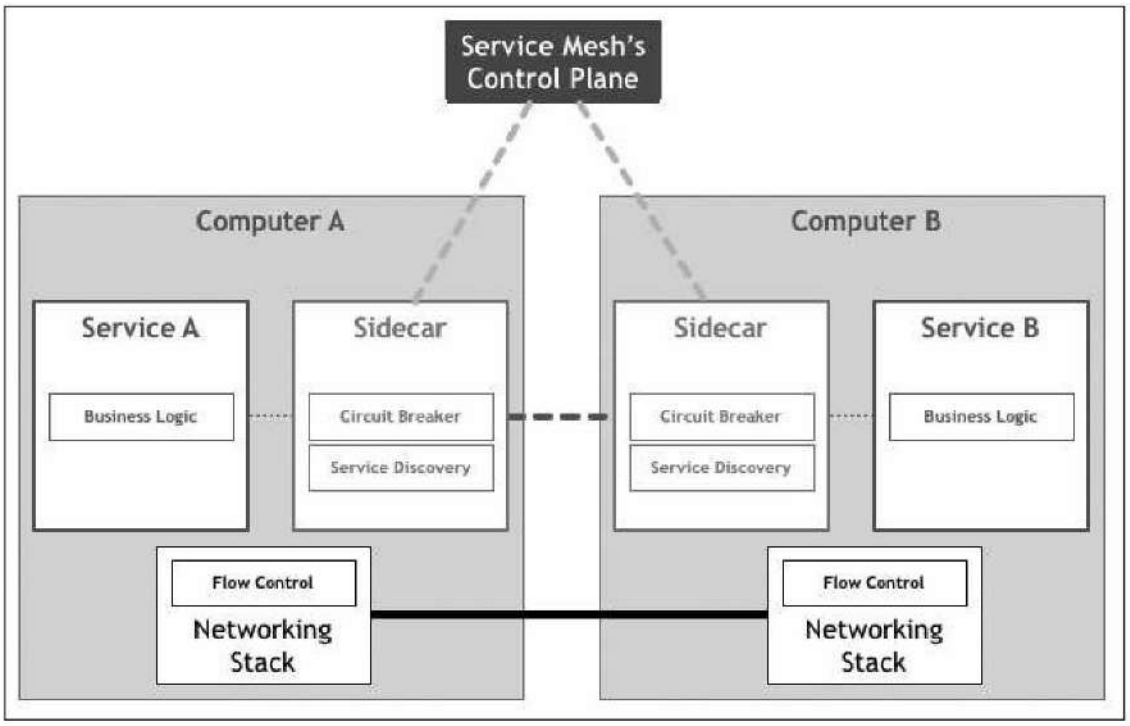


图6-2　Sidecar模式下的Service Mesh的基本原理

sidecar模式在概念上比较简单，我们为什么要在Service　　Mesh中使用sidecar模式呢？

·解决服务之间调用越来越复杂的问题。随着分布式架构越来越复杂和微服务越拆越细，我们越来越迫切地希望有一个统一的控制面来管理我们的微服务，帮助我们维护和管理所有微服务，这时传统开发层面上的控制就远远不够了，而边sidecar模式可以很好地解决这个问题；

·控制与逻辑分离的问题。sidecar模式基于将控制与逻辑分离和解耦的思想。通俗地讲就是，让专业的人做专业的事，业务代码只需要关心其复杂的业务逻辑，其他的事情sidecar会帮其处理。例如，日志记录、监控、流量控制、服务注册、服务发现、服务限流、服务熔断、鉴权、访问控制和服务调用可视化等功能本质上讲和业务服务的关系并不大，完全可以交给sidecar。

这就是Service　　Mesh诞生的契机，它是CNCF主推的新一代微服务架构。William Morgan在*What's a service mesh*？*And why do I need one*文章中指出Service Mesh有以下几个特点：

·应用程序间通信的中间层；

·轻量级网络代理；

·应用程序无感知；

·解耦应用程序的重试/超时、监控、追踪和服务发现。

Service　　Mesh将底层那些难以控制的网络通信统一管理，诸如流量管

控、丢包重试、访问控制等。而上层的应用层协议只须关心业务逻辑。Service Mesh是一个用于处理服务间通信的基础设施层，它负责为构建复杂的云原生应用传递可靠的网络请求。

6.1.4 Kubernetes Service VS.Service Mesh

下面，让我们对比Kubernetes Service与Service Mesh中的服务访问关系。

Kubernetes服务访问原理如图6-3所示。

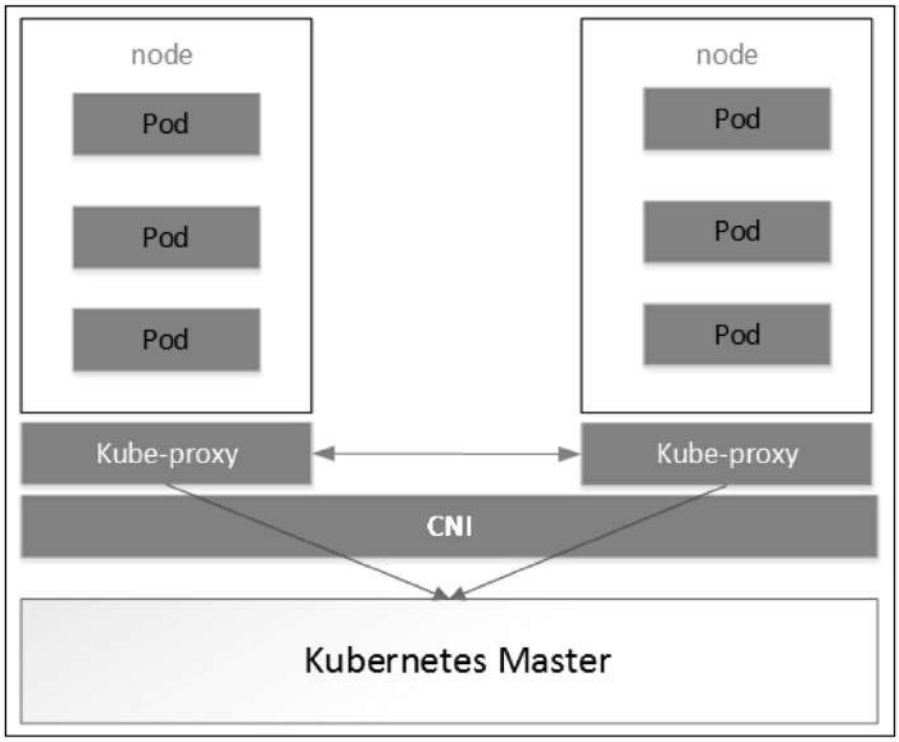


图6-3 Kubernetes服务访问原理

正如我们在前文讨论的，Kubernetes集群的每个节点都部署了一个Kube-proxy组件，该组件会与Kubernetes API Server通信，获取集群中的Service信息，然后设置iptables/ipvs规则，直接将对某个Service的请求发送到对应的后端Pod上。

Kube-proxy实现了流量在Kubernetes Service的负载均衡，但是没法对流量做细粒度的控制，例如灰度发布和蓝绿发布（按照百分比划分流量到不同的应用版本）等。Kubernetes社区提供的蓝绿发布案例其实是针对Deployment的，但不支持Service。

Istio Service Mesh的原理如图6-4所示。

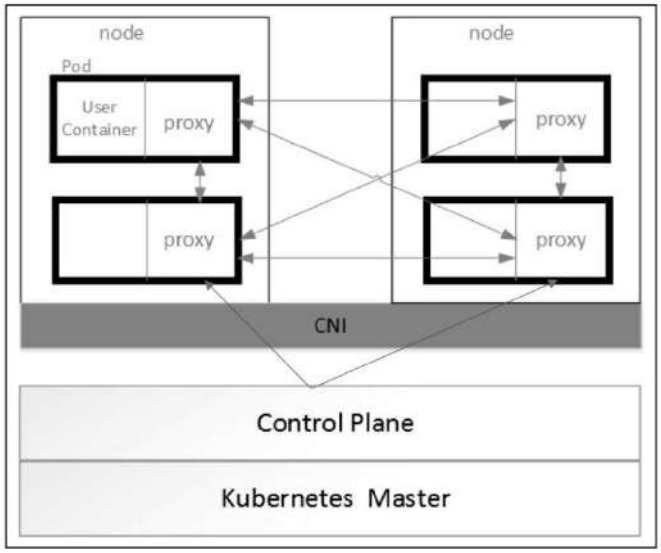


图6-4　Istio Service Mesh的原理

Istio Service Mesh把Kubernetes看作服务注册机构，通过控制平面生成数据平面的配置，数据平面的透明代理以sidecar容器的方式部署在每个应用服务的Pod中。之所以说是透明代理，是因为应用程序容器完全无感知代理的存在。区别在于Kube-proxy拦截的是进出Kubernetes节点的流量，而Istio sidecar拦截的是进出该Pod的流量。

6.1.5　Service Mesh典型实现之Linkerd

Linkerd是一个用于云原生应用的开源Service Mesh实现，也是CNCF的子项目。

Linkerd的出现是为了解决像Twitter、Google这类超大规模生产系统的复杂性问题。Linkerd不是通过控制服务之间的通信机制来解决这个问题的，而是通过在服务实例之上添加一个抽象层来解决的。Linkerd的架构如图6-5所示。

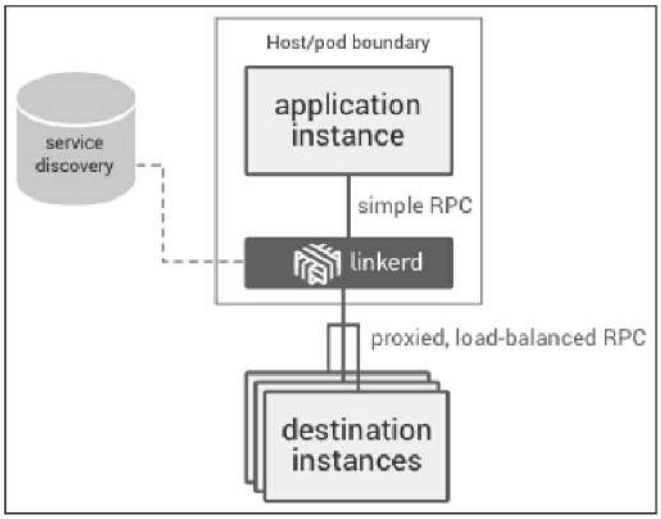


图6-5　Linkerd的架构

Linkerd负责跨服务通信中最困难、最易出错的部分，包括延迟感知、负载均衡、连接池、TLS、仪表盘、请求路由等，这些都会影响应用程序的伸缩性、性能和弹性。

运行**Linkerd**

Linkerd作为独立代理运行，无须特定的语言和库支持。应用程序通常会在已知位置运行Linkerd实例，然后通过这些实例代理服务调用——即不是直接连接到目标服务，服务连接到它们对应的Linkerd实例，并将它们视为目标服务。

在该层上，Linkerd应用路由规则，与现有服务发现机制通信，对目标实例做负载均衡。与此同时，调整通信并报告指标。

通过延迟调用Linkerd的机制，应用程序代码与以下内容解耦：

·生产拓扑；

·服务发现机制；

·负载均衡和连接管理逻辑。

应用程序也将从一致的全局流量控制系统中受益。这对多语言应用程序尤为重要，因为通过库来实现这种一致性是非常困难的。

Linkerd实例可以作为sidecar运行。Linkerd实例是无状态和独立的，因此它们可以轻松适应现有的部署拓扑。它们可以与各种配置的应用程序代码一起部署，并且基本不需要协调它们。

Linkerd的工作原理如下：

（1）Linkerd将服务请求路由到目的地址，根据请求的参数判断是到生产环境、测试环境还是staging环境中的服务（服务可能同时部署在这三个环境中），是路由到本地环境还是公有云环境？所有这些路由信息既可以动态配置，也可以是全局配置，甚至也支持为某些服务单独配置。

（2）当Linkerd确认了目的地址后，将流量发送到相应服务发现端点，在Kubernetes中就是Service，然后Service会将服务转发给后端的实例。

（3）Linkerd根据观测到最近请求的延迟时间，选择所有应用程序的实例中响应最快的实例。

（4）Linkerd将请求发送给该实例，同时记录响应类型和延迟数据。

（5）如果该实例挂了，则Linkerd将把请求发送到其他实例上重试。

（6）如果该实例持续返回错误，则Linkerd会将该实例从负载均衡池中

移除，稍后再周期性地重试。

（7）如果请求的截止时间已过，则Linkerd主动使该请求失败，而不是再次尝试添加负载。

（8）Linkerd以metric和分布式追踪的形式捕获上述行为的各个方面，这些追踪信息将被发送到集中metric系统。

6.2 Istio：引领新一代微服务架构潮流

在GitHub上，Istio项目的活跃率非常高，俨然成了容器生态区继Kubernetes后最大的项目。目前，Istio项目贡献排名最高的分别是Google、IBM、Casico和国内的华为，同时也对应于对项目的控制力度。

6.2.1 Istio简介

什么是Istio？官方给出的定义是：

An open platform to connect，secure，control and control services.

即一个提供微服务连接的、安全的、流量控制的和可观察性的开放平台。

Istio分为两个平面：数据平面和控制平面。

数据平面由一组sidecar的代理（Envoy）组成。这些代理调解和控制微服务之间的所有网络通信，并且与控制平面的Mixer通信，接受调度策略。

控制平面通过管理和配置Envoy管理流量。此外，控制平面配置Mixers来实施路由策略并收集检测到的监控数据。

Istio的整体架构如图6-6所示。

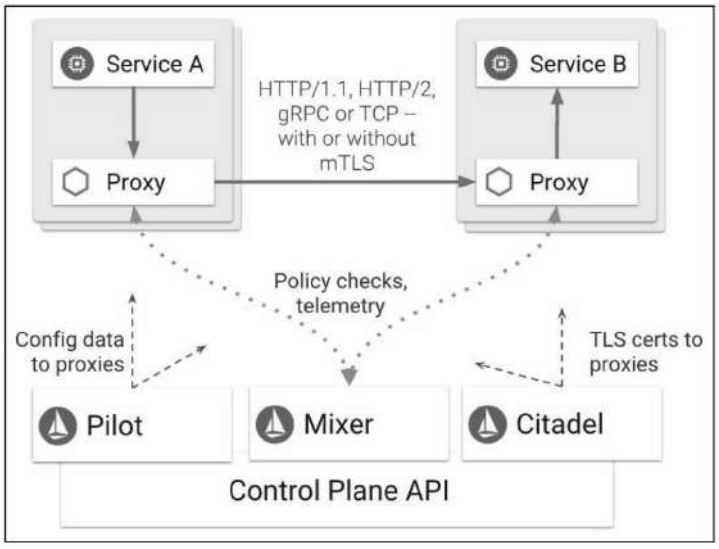


图6-6　Istio的整体架构

从图6-6可以看出，Istio的组件主要包括Envoy、Pilot、Mixer和Citadel。

Envoy

Envoy是Lyft开源的一个用C++开发的高性能代理，用于管理Service Mesh中所有服务的所有入站和出站流量。Envoy从功能上看是一个类似于Nginx的七层代理，但更加贴近Service Mesh的使用场景。Istio利用了Envoy的许多内置功能，例如：

·动态服务发现；

·负载均衡；

·TLS终止；

·HTTP/2和gRPC代理；

·断路器；

·健康检查；

·流量分割；

·故障注入；

·监控指标。

与Linderd类似，Istio在每个用户创建的Pod中自动注入一个sidecar，即Envoy。

Pilot

Pilot用于配置Envoy，提供服务发现、流量管理（例如A/B测试、金丝雀部署等）、异常控制（例如超时、重试、熔断）等功能。

Mixer

Mixer是一个独立于平台的组件，负责在整个Service Mesh中执行访问控制和使用策略，并通过Envoy代理和其他服务收集监控到的数据。

Citadel

Citadel通过内置身份和凭证管理，提供服务和用户的身份验证。我们可以使用Citadel升级Service Mesh中的未加密流量。

6.2.2 Istio安装

在安装Istio核心组件之前，需要安装一个“服务注册器”，这个“服务注册器”既可以是Kubernetes，也可以是Nomad & Consul。下面笔者以Kubernetes为例，讲解如何在Kubernetes集群中安装Istio控制平面。

Istio提供多种安装路径，具体取决于你环境中的Kubernetes平台。但不论平台如何，基本流程都是相同的，即：

（1）确认Istio对Pod和服务的要求。

（2）安装Kubernetes。

（3）在Kubernetes上安装Istio。

Istio对Pod和服务的要求。要成为服务网格的一部分，Kubernetes集群中的Pod和服务必须满足以下几个要求：

·需要对端口进行正确命名：服务端口必须进行命名。端口名称只允许是<协议>［-<后缀>-］模式，其中<协议>部分的可选择范围包括grpc、http、http2、https、mongo、redis、tcp、tls及udp，Istio可以通过对这些协议的支持提供路由能力。例如name:http2-foo和name:http都是有效的端口名，但name:http2foo就是无效的。如果没有对端口进行命名，或者命名没有使用指定前缀，那么这一端口的流量就会被视为普通TCP流量；

·**Pod**端口：Pod必须包含每个容器将监听的明确端口列表。在每个端口的容器规范中使用containerPort。任何未列出的端口都将绕过Istio Proxy；

·关联服务：Pod不论是否公开端口，都必须关联到至少一个Kubernetes服务上，如果一个Pod属于多个服务，这些服务不能在同一端口上使用不同协议，例如HTTP和TCP；

·**Deployment**应带有**App**及**version**标签：在使用Kubernetes Deployment进行Pod部署的时候，建议显式地为Deployment加上app及version标签。每个Deployment都应该有一个有意义的app标签和一个用于标识Deployment版本的version标签。app标签在分布式追踪的过程中会被用来加入上下文信息。Istio还会用app和version标签向遥测指标数据加入上下文信息；

·**Application UID**：不要使用ID（UID）值为1337的用户运行应用；

·**NET_ADMIN**功能：如果你的Kubernetes群集中实施了Pod安全策略，除非使用Istio

CNI插件，否则你的Pod必须开启NET_ADMIN权限。

1.安装**Kubernetes**

安装Kubernetes的方式有很多，我们在前面的章节中介绍过使用kubeadm安装Kubernetes集群。用户可以自由选择安装方式，这里不再赘述。

2.在**Kubernetes**上安装**Istio**

下面我们将介绍无须安装Helm，只使用基本的kubectl命令部署Istio的步骤。

1）下载**Istio**安装包

（1）进入Istio release页面，下载对应目标操作系统的安装文件。在macOS或者Linux系统中，还可以运行下面的命令，进行下载和自动解压缩：

```
# curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.1.5 sh -
```

（2）进入Istio包目录。例如，假设这个包是istio-1.1.5：

```
# cd istio-1.1.5
```

安装目录中包含：

·在install/目录中包含了Kubernetes安装所需的.yaml文件；

·samples/目录中是示例应用；

·istioctl客户端文件保存在bin/目录中。istioctl的功能是手工进行Envoy Sidecar的注入；

·istio.VERSION配置文件。

（3）把istioctl客户端加入PATH环境变量，如果是macOS或者Linux，则可以这样实现：

```
# export PATH=$PWD/bin:$PATH
```

2）安装Istio

在安装好Kubernetes后，Istio的安装无非就是一系列的CRD，Istio会被安装到自己的istio-system命名空间，并且能够对所有其他命名空间的服务进行管理。使用kubectl apply安装Istio的自定义资源定义（CRD），如下所示：

```
# for i in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl apply -f $i;
 done
```

如果使用mutual TLS的宽容模式，则所有服务会同时允许明文和双向TLS的流量。在没有明确配置客户端进行双向TLS通信的情况下，客户端会发送明文流量。

这种方式的适用场景：

·已有应用的集群；

·注入了Istio sidecar的服务有和非Istio Kubernetes服务通信的需要；

·需要进行liveness probe和readiness probes的应用；

·Headless服务；

·StatefulSet。

运行下面的命令即可完成这一模式的安装：

```
# kubectl apply -f install/kubernetes/istio-demo.yaml
```

如果使用严格模式的mutual TLS方案，会在所有的客户端和服务器之间使用双向TLS。这种方式只适合所有工作负载都受Istio管理的Kubernetes集群。所有新部署的工作负载都会注入Istio sidecar。

运行下面的命令可以安装这种方案：

```
# kubectl apply -f install/kubernetes/istio-demo-auth.yaml
```

3.确认部署结果

如何确认Istio已经安装完成了呢？

（1）确认下列Kubernetes服务已经部署并都具有各自的CLUSTER-IP。由于输出结果较多，这里就不列出来了。如果你的集群在一个没有外部负载均衡器支持的环境中运行（例如Minikube），那么istio-ingressgateway的EXTERNAL-IP会是<pending>。要访问这个网关，只能通过服务的NodePort或者使用端口转发进行访问。

（2）确认必要的Kubernetes Pod都已经创建并且其STATUS的值是Running。由于输出结果较多，这里就不一一列出了。

4.部署应用

安装好Istio后，就可以部署用户自己的应用了。初学者可以从Istio的发布包中找一个示例应用（例如官方的Bookinfo例子）进行部署。

在使用kubectl apply进行应用部署的时候，如果目标命名空间已经打上了标签istio-injection=enabled，Istio sidecar injector会自动把Envoy容器注入你的应用Pod中：

```
# kubectl label namespace <namespace> istio-injection=enabled
# kubectl create -n <namespace> -f <your-app-spec>.yaml
```

如果目标命名空间中没有打上istio-injection标签，则可以使用istioctl kubeinject命令，在部署之前手工把Envoy容器注入应用Pod中：

```
# istioctl kube-inject -f <your-app-spec>.yaml | kubectl apply -f -
```

6.2.3 Istio路由规则的实现

在Istio中，与路由相关的有4个概念：Virtual Services、Destination Rules、ServiceEntry和Gateways。

Virtual Services的作用是定义了针对Istio中一个微服务的请求的路由规则。Virtual Services既可以将请求路由到一个应用的不同版本，也可以将请求路由到完全不同的应用。在下面的示例配置中，发给微服务的请求将被路由到productpage，端口号为9080：

```
route:
    - destination:
        host: productpage
        port:
          number: 9080
```

在下面的示例配置中，我们定义了熔断策略：

```
spec:
  host: productpage
  subsets:
  - labels:
      version: v1
    name: v1
  trafficPolicy:
    connectionPool:
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
      tcp:
        maxConnections: 1
    tls:
```

ServiceEntry用于将Istio外部的服务注册到Istio的内部服务注册表，以便Istio内部的服务可以访问这些外部的服务，如Istio外部的Web API。

在下面的示例配置中，定义了Istio外部的mongocluster与Istio内部的访问规则：

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: external-svc-mongocluster
```

```
spec:
  hosts:
  - mymongodb.somedomain # not used
  addresses:
  - 192.192.192.192/24 # VIPs
  ports:
  - number: 27018
    name: mongodb
    protocol: MONGO
  location: MESH_INTERNAL
  resolution: STATIC
  endpoints:
  - address: 2.2.2.2
  - address: 3.3.3.3
```

Gateway定义了Istio边缘的负载均衡器。所谓边缘，就是Istio的入口和出口。这个负载均衡器用于接收传入或传出Istio的HTTP/TCP连接。在Istio中会有ingressgateway和egressgateway，前者负责入口流量，后者负责出口流量。

在下面的示例配置中，定义了Istio的入口流量。

```
spec:
  selector:
    istio: ingressgateway
  servers:
  - hosts:
    - '*'
    port:
      name: http
      number: 80
      protocol: HTTP
```

参考资料：

［1］https://istio.io/zh/docs/setup/kubernetes/additional-setup/sidecar-injection/.

［2］https://istio.io/zh/blog/2019/data-plane-setup/.

6.3 一切尽在不言中：Istio sidecar透明注入

网格中的每个Pod都必伴随着一个Istio的sidecar一同运行。下文中将会介绍两种把sidecar注入Pod的方法：使用istioctl客户端工具进行注入，或者使用Istio sidecar injector自动完成注入过程，并且深入sidecar内部解析其工作原理。

手工注入过程会修改控制器（例如Deployment）的配置。这种注入方法会修改Pod template，把sidecar注入目标控制器生成的所有Pod。要加入、更新或者移除sidecar，就需要修改整个控制器。

自动注入过程会在Pod的生成过程中进行注入，这种方法不会更改控制器的配置。手工删除Pod或者使用滚动更新都可以选择性地对sidecar进行更新。

手工或自动注入都会从istio-system命名空间的istio-sidecar-injector及istio ConfigMap中获取配置信息。手工注入方式还可以选择从本地文件中读取配置。

6.3.1 Init容器

在介绍Istio如何将Envoy代理注入用户Pod之前，我们需要先了解什么是Init容器。

Init容器是一种专用容器，它在应用程序容器启动之前运行，用来包含一些应用镜像中不存在的实用工具或安装脚本。一个Pod中可以指定多个Init容器，如果指定了多个，那么Init容器将会按顺序依次运行。只有当前面的Init容器运行成功后，才可以运行下一个Init容器。Init容器运行完成以后就会自动终止，当所有的Init容器运行完成后，Kubernetes才运行应用容器。

在Pod启动过程中，Init容器会按顺序在网络和数据卷初始化之后启动。如果Init容器退出失败，则它会根据Pod的restartPolicy指定的策略进行重试。在Init容器没有全部成功退出之前，Pod将不会变成Running状态而是一直处在Initializing状态。

虽然Pod内所有容器都共享网络namespace，但Init容器和Pod内其他应用容器的文件系统是隔离的。例如，当赋予Init容器访问Kubernetes Secret的权限时，其他应用容器可能没有这个权限。

6.3.2 sidecar注入示例

关于bookinfo应用的详细YAML配置请参考官网的bookinfo.yaml示例：

```
apiVersion: extensions/v1beta1
kind: Deployment
```

```
metadata:
  name: productpage-v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: productpage
        version: v1
    spec:
      containers:
      - name: productpage
        image: istio/examples-bookinfo-productpage-v1:1.8.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
```

再查看productpag容器的Dockerfile：

```
FROM python:2.7-slim

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY productpage.py /opt/microservices/
COPY templates /opt/microservices/templates
COPY requirements.txt /opt/microservices/
EXPOSE 9080
WORKDIR /opt/microservices
CMD python productpage.py 9080
```

我们看到Dockerfile中没有配置ENTRYPOINT，所以CMD的配置python productpage.py 9080将作为默认的ENTRYPOINT。记住这一点，再看注入sidecar之后的配置：

```
# istioctl kube-inject -f yaml/istio-bookinfo/bookinfo.yaml
```

让我们看下productpage这个Deployment配置文件发生了哪些变化：

```
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
  creationTimestamp: null
  name: productpage-v1
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        sidecar.istio.io/status: '{"version":"
fde14299e2ae804b95be08e0f2d171d466f47983391c00519bbf01392d9ad6bb","initContainers":["
istio-init"],"containers":["istio-proxy"],"volumes":["istio-envoy","istio-certs"],"
imagePullSecrets":null}'
      creationTimestamp: null
      labels:
        app: productpage
        version: v1
    spec:
      containers:
      - image: istio/examples-bookinfo-productpage-v1:1.8.0
        name: productpage
        ports:
        - containerPort: 9080
        ...
      initContainers:
      - args:
        - -p
        - "15001"
        - -u
        - "1337"
        - -m
        - REDIRECT
        - -i
        - '*'
        - -x
        - ""
        - -b
```

```
      - 9080,
      - -d
      - ""
      image: jimmysong/istio-release-proxy_init:1.0.0
      imagePullPolicy: IfNotPresent
      name: istio-init
      securityContext:
        capabilities:
          add:
          - NET_ADMIN
        privileged: true
    volumes:
    - emptyDir:
        medium: Memory
      name: istio-envoy
    - name: istio-certs
      secret:
        optional: true
        secretName: istio.default
```

我们看到，Service的配置没有变化，所有的变化都在Deployment里，Istio给应用Pod注入的配置主要包括：

·Init容器istio-init：用于给sidecar容器（即Envoy代理）做初始化，设置iptables端口转发；

·Envoy sidecar容器istio-proxy：运行Envoy代理。接下来将详细解析Init容器。

1.istio-init容器解析

Istio在Pod中注入的Init容器名为istio-init，我们在Istio注入完成后的YAML文件中看到了该容器的启动参数，如下所示：

```
-p 15001 -u 1337 -m REDIRECT -i '*' -x "" -b 9080 -d ""
```

再检查该容器的Dockerfile，看看ENTRYPOINT是什么：

```
FROM ubuntu:xenial
RUN apt-get update && apt-get install -y \
```

```
    iproute2 \
    iptables \
 && rm -rf /var/lib/apt/lists/*

ADD istio-iptables.sh /usr/local/bin/
ENTRYPOINT ["/usr/local/bin/istio-iptables.sh"]
```

我们看到istio-init容器的入口是/usr/local/bin/istio-iptables.sh脚本，再按图索骥看看这个脚本里到底写的什么，该脚本即Istio源码仓库的tools/deb/istioiptables.sh，一共300多行。该脚本的用法如下：

```
# istio-iptables.sh -p PORT -u UID -g GID [-m mode] [-b ports] [-d ports] [-i CIDR]
[-x CIDR] [-h]
```

其中：

-p：指定重定向所有TCP流量的Envoy端口，默认Envoy的数据端口为15001；

-u：指定未应用重定向的用户的UID。通常这是代理容器的UID，默认为1337；

-g：指定未应用重定向的用户的GID；

-m：指定入站连接重定向到Envoy的模式，包括REDIRECT和TPROXY两种；

-b：逗号分隔的入站端口列表，其流量将重定向到Envoy（可选）。使用通配符“*”表示重定向所有端口，为空时表示禁用所有入站重定向；

-d：指定要从重定向到Envoy中排除（可选）的入站端口列表，以逗号格式分隔。使用通配符“*”表示重定向所有入站流量；

-i：指定重定向到Envoy（可选）的IP地址范围，以逗号分隔的CIDR格式列表。使用通配符“*”表示重定向所有出站流量。空列表将禁用所有出站重定向；

-x：指定将从重定向中排除的IP地址范围，以逗号分隔的CIDR格式列表。使用通配符“*”表示重定向所有出站流量。

通过查看istio-iptables.sh脚本，发现以上传入的参数都会重新组装成iptables命令的参数。

istio-iptables.sh完整的启动命令如下：

```
# /usr/local/bin/istio-iptables.sh -p 15001 -u 1337 -m REDIRECT -i '*' -x "" -b 9080
-d ""
```

该容器存在的意义就是让Envoy代理可以拦截所有的进出Pod的流量，即将入站流量重定向到sidecar，再拦截应用容器的出站流量，经过sidecar处理后再出站。

上面这条启动命令的作用是：

·将应用容器的所有流量都转发到Envoy的15001端口；

·使用istio-proxy用户身份运行，UID为1337，即Envoy所处的用户空间，这也是istioproxy容器默认使用的用户，见YAML配置中的runAsUser字段；

·使用默认的REDIRECT模式重定向流量；

·将所有出站流量都重定向到Envoy代理；

·将所有访问9080端口（即应用容器productpage的端口）的流量重定向到Envoy代理。

Init容器初始化完毕后就会自动终止，因此我们无法直接登录Init容器查看iptables信息，但是Init容器的初始化结果会保留到应用容器和sidecar容器中。

如果确实需要查看Istio Pod中的iptables规则，我们需要登录sidecar容器查看。因为kubectl无法使用特权模式远程操作Docker容器，所以我们需要登录Pod所在的主机，使用docker exec命令登录容器查看。

查看iptables配置，列出NAT表的所有规则。Init容器启动时，在向istio-iptables.sh传递的参数中指定Envoy的入站流量为REDIRECT，因此在iptables中将只有NAT表的规格配置。如果选择TPROXY，还会有mangle表配置。iptables命令的详细用法请参考第1章的内容。

Init容器启动时，命令行参数中指定了REDIRECT模式，因此只创建了nat表规则。下面列出的nat表规则，其中链的名字中包含ISTIO前缀的是由Init容器注入的，规则匹配是根据下面显示的顺序执行的，其中会有多次跳转：

```
$ iptables -t nat -L -v
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2   120 ISTIO_INBOUND  tcp  --  any    any     anywhere             anywhere

Chain INPUT (policy ACCEPT 2 packets, 120 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 41146 packets, 3845K bytes)
 pkts bytes target     prot opt in     out     source               destination
   93  5580 ISTIO_OUTPUT  tcp  --  any    any     anywhere             anywhere

Chain POSTROUTING (policy ACCEPT 41199 packets, 3848K bytes)
 pkts bytes target     prot opt in     out     source               destination
```

```
Chain ISTIO_INBOUND (1 references)
 pkts bytes target     prot opt in     out     source               destination
    2   120 ISTIO_IN_REDIRECT  tcp  --  any    any     anywhere             anywhere
           tcp dpt:9080

Chain ISTIO_IN_REDIRECT (1 references)
 pkts bytes target     prot opt in     out     source               destination
    2   120 REDIRECT   tcp  --  any    any     anywhere             anywhere
    redir ports 15001

Chain ISTIO_OUTPUT (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 ISTIO_REDIRECT  all  --  any    lo      anywhere             !localhost
   40  2400 RETURN     all  --  any    any     anywhere             anywhere
    owner UID match istio-proxy
    0     0 RETURN     all  --  any    any     anywhere             anywhere
    owner GID match istio-proxy
    0     0 RETURN     all  --  any    any     anywhere             localhost
   53  3180 ISTIO_REDIRECT  all  --  any    any     anywhere             anywhere

Chain ISTIO_REDIRECT (2 references)
 pkts bytes target     prot opt in     out     source               destination
   53  3180 REDIRECT   tcp  --  any    any     anywhere             anywhere
    redir ports 15001
```

其中：

·PREROUTING链用于DNAT，将所有入站TCP流量跳转到ISTIO_INBOUND链上；

·INPUT链处理输入数据包，目前没有任何规则；

·OUTPUT链将所有出站数据包跳转到ISTIO_OUTPUT链上；

·POSTROUTING链，所有数据包流出网卡时都要先进入POSTROUTING链，内核根据数据包目的地判断是否需要转发出去，我们看到此处未做任何处理；

·ISTIO_INBOUND链将所有目的地为9080端口的入站流量重定向到ISTIO_IN_REDIRECT链上；

·ISTIO_IN_REDIRECT链将所有的入站流量跳转到本地的15001端口，

至此成功地拦截了流量并转发给Envoy；

·ISTIO_OUTPUT链选择需要重定向到Envoy（即本地）的出站流量，所有非localhost的流量全部转发到ISTIO_REDIRECT。为了避免流量在该Pod中无限循环，所有到istio-proxy用户空间的流量都返回到它的调用点中的下一条规则，本例中即OUTPUT链，因为跳出ISTIO_OUTPUT规则之后就进入下一条POSTROUTING链。如果目的地非localhost，就跳转到ISTIO_REDIRECT；如果流量是来自istio-proxy用户空间的，就跳出该链，返回它的调用链继续执行下一条规则（OUPT的下一条规则，无须对流量进行处理）；所有的非istio-proxy用户空间的目的地是localhost的流量就跳转到ISTIO_REDIRECT；

·ISTIO_REDIRECT链将所有流量重定向到Envoy（即本地）的15001端口；

其实在最后这条规则前，还可以增加IP地址过滤，让某些IP地址段不通过Envoy代理。

6.3.3 手工注入sidecar

使用集群内置配置将sidecar注入Deployment中：

```
# istioctl kube-inject -f samples/sleep/sleep.yaml | kubectl apply -f -
```

此外，还可以使用本地的配置信息进行注入。

istioctl kube-inject操作不具备幂等性，因此istioctl kube-inject的输出内容是无法再次进行注入的。要对手工注入的工作负载进行更新，建议保留原本的未经注入的yaml文件，这样数据平面的sidecar就可以被更新了。

```
# kubectl -n istio-system get configmap istio-sidecar-injector -o=jsonpath='{.data.
config}' > inject-config.yaml
# kubectl -n istio-system get configmap istio -o=jsonpath='{.data.mesh}' > mesh-
config.yaml
```

对输入文件运行kube-inject，并进行部署：

```
# istioctl kube-inject \
    --injectConfigFile inject-config.yaml \
    --meshConfigFile mesh-config.yaml \
    --filename samples/sleep/sleep.yaml \
    --output sleep-injected.yaml
# kubectl apply -f sleep-injected.yaml
```

检查注入Deployment中的sidecar：

```
# kubectl get deployment sleep -o wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE       CONTAINERS
IMAGES                              SELECTOR
sleep     1         1         1            1           2h        sleep,istio-proxy
tutum/curl,unknown/proxy:unknown    app=sleep
```

6.3.4 自动注入sidecar

使用Kubernetes的mutating webhook admission controller，可以进行sidecar的自动注入，不过这个功能要求使用Kubernetes 1.9以后的版本。使用这一功能之前首先要检查Kubernetes API Server是否配置了admission-control参数，并且这个参数的值中需要包含MutatingAdmissionWebhook及ValidatingAdmissionWebhook两项，并且按照正确的顺序加载，这样才能启用admissionregistration API：

```
# kubectl api-versions | grep admissionregistration
admissionregistration.k8s.io/v1alpha1
admissionregistration.k8s.io/v1beta1
```

需要注意的是，跟手工注入不同，自动注入过程是发生在Pod级别的，因此不会看到Deployment本身发生什么变化。但是可以使用kubectl describe观察单独的Pod，在其中能看到注入sidecar的相关信息。

1.验证**sidecar**注入

部署sleep应用，检查是不是只产生了一个容器。

```
# kubectl apply -f samples/sleep/sleep.yaml

# kubectl get deployment -o wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE       CONTAINERS   IMAGES
      SELECTOR
sleep     1         1         1            1           12m       sleep        tutum/
curl   app=sleep
```

```
# kubectl get pod
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-7hpnk   1/1       Running       0          4
```

给default命名空间设置标签istio-injection=enabled：

```
# kubectl label namespace default istio-injection=enabled

# kubectl get namespace -L istio-injection
NAME           STATUS    AGE       ISTIO-INJECTION
default        Active    1h        enabled
istio-system   Active    1h
kube-public    Active    1h
kube-system    Active    1h
```

这样就会在Pod创建时触发sidecar的注入过程。删除运行的Pod，会产生一个新的Pod，新的Pod会被注入sidecar。原有的Pod只有一个容器，而被注入sidecar的Pod会有两个容器：

```
# kubectl delete pod sleep-776b7bcdcd-7hpnk

# kubectl get pod
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-7hpnk   1/1       Terminating   0          1m
sleep-776b7bcdcd-bhn9m   2/2       Running       0          7s
```

查看被注入的Pod的细节。不难发现多出了一个istio-proxy容器及其对应的存储卷。注意，用正确的Pod名称来执行下面的命令：

```
# kubectl describe pod sleep-776b7bcdcd-bhn9m
```

禁用default命名空间的自动注入功能，然后检查新建Pod是不是不带sidecar容器：

```
# kubectl label namespace default istio-injection-
# kubectl delete pod sleep-776b7bcdcd-bhn9m

# kubectl get pod
NAME                     READY     STATUS        RESTARTS   AGE
sleep-776b7bcdcd-bhn9m   2/2       Terminating   0          2m
sleep-776b7bcdcd-gmvnr   1/1       Running       0          2s
```

6.3.5 从应用容器到sidecar代理的通信

现在我们知道了sidecar容器和初始化容器被注入到应用中的过程，sidecar代理是如何截获进出容器的流量的呢？我们前面提到过，这是通过对Pod命名空间中iptable规则的设置来完成的，这个设置过程由istio-init容器控制。现在可以看看命名空间中到底更新了些什么。

进入前面我们部署的应用Pod的命名空间，看看配置完成的iptables。既可以使用nsenter，也可以用特权模式进入容器，获得同样的信息。如果无法访问节点，则可以用exec进入sidecar来执行iptables指令。

```
# docker inspect b8de099d3510 --format '{{ .State.Pid }}'
4125
```

从上面的命令查到容器对应的PID是4125。因此，通过下面的命令查看该进程namespace里面的iptables规则，如下所示：

```
# nsenter -t 4215 -n iptables -t nat -S

-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N ISTIO_INBOUND
-N ISTIO_IN_REDIRECT
-N ISTIO_OUTPUT
-N ISTIO_REDIRECT
-A PREROUTING -p tcp -j ISTIO_INBOUND
-A OUTPUT -p tcp -j ISTIO_OUTPUT
-A ISTIO_INBOUND -p tcp -m tcp --dport 80 -j ISTIO_IN_REDIRECT
-A ISTIO_IN_REDIRECT -p tcp -j REDIRECT --to-ports 15001
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -j ISTIO_REDIRECT
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -d 127.0.0.1/32 -j RETURN
-A ISTIO_OUTPUT -j ISTIO_REDIRECT
-A ISTIO_REDIRECT -p tcp -j REDIRECT --to-ports 15001
```

上面展示的内容中，可以清楚地看到，所有从80端口（red-demo应用监听的端口）进入的流量，被REDIRECTED到了端口15001。这是istio-proxy（Envoy代理）监听的端口。外发流量也是如此。

至此，我们已经介绍了Istio将sidecar注入Pod，以及Istio将流量路由到代理服务器的全过程。

6.4 不再为iptables脚本所困：Istio CNI插件

Istio当前默认使用特权init容器istio-init将访问用户容器的流量转发到Envoy。istioinit的主要作用是运行脚本配置容器内的iptables规则。Istio CNI插件的主要设计目标是消除这个特权init container，使用Kubernetes CNI机制实现相同功能的替代方案，因此它是Kubernetes CNI的一个具体实现。

前面的章节已经提到Kubernetes CNI插件是一条链，在创建和销毁Pod的时候会调用链上所有的插件来安装和卸载容器的网络。Istio CNI Plugin相当于在创建销毁Pod的这些钩子处针对Istio的Pod做网络配置，即配置iptables规则让访问该Pod的网络流量转发给Envoy。

当然，要让Kubernetes使用Istio CNI插件需要Kubelet启动时配置参数--networkplugin=cni，复制Istio CNI插件二进制程序到CNI的bin目录，即Kubelet启动参数--cni-bin-dir指定的路径，默认是/opt/cni/bin。

Istio CNI插件工作原理

在每个节点上运行一个名为istio-cni-node的Daemonset，用于安装Istio CNI插件。启用Istio CNI插件后，sidecar的自动注入或手动注入（istioctl kube-inject）将不再注入init容器（istio-init）。

istio-cni-node的工作流程如下所示：

·使用istio-cni-node自己的ServiceAccount信息为CNI插件生成kubeconfig，让插件能与Kubernetes API Server通信，其中ServiceAccount信息会被自动挂载到/var/run/secrets/kubernetes.io/serviceaccount；

·生成CNI插件的配置，并将其插到CNI配置插件链的末尾，其中CNI的配置文件路径是Kubelet启动参数--cni-conf-dir所指定的目录，默认是/etc/cni/net.d；

·watch--cni-conf-dir目录下的CNI配置，用来检测是否有配置更新并在内存中重载；

·watch存储istio-cni-node自身的配置ConfigMap对象，检测到有修改就重新执行CNI配置生并与下发流程。

6.5 除了微服务，Istio还能做更多

在前面的章节中，我们探讨了微服务新的设计模式：sidecar模式及Istio创新的架构在微服务中扮演的重要角色。在笔者看来，Istio的能量绝不仅仅体现在微服务领域。它将在现代网络基础设施的路由和安全领域扮演非常重要的角色，甚至极有可能颠覆现有的网络基础设施，成为未来混合云的基石。

Istio在混合云中的应用

为什么需要混合云这种形态的“云”？混合云有多种组成形式，形式一是公有云和私有云的融合，形式二是多种公有云的互补。

先来看公有云和私有云融合这种形式的混合云。笔者曾供职于国内一线公有云团队，虽然笔者相信最终所有的私有云都会被公有云收编，但我们必须意识到用户在公有云的数据隐私保护等方面仍有所保留，一些经济效益好的企业（例如金融等），在现阶段更喜欢自建专有云平台。而混合云可以认为是对当前这种公有云和私有云共存现状的妥协，做到公有云和私有云的优势互补。例如，用户可以同时享受云的优势（灵活性、可扩展性、成本降低）和本地的好处（安全性、低延迟、硬件复用）。如果用户是首次把业务迁移到云端，采用混合云步骤可以按照用户自己的节奏，以最适合业务的方式进行。此外，私有云的一些业务在峰值时可以弹性扩容到公有云上。

而采用多共有云的混合云意味着应用可以跨多个公共云平台运行。使用多个云提供商可以帮助用户避免厂商锁定，并为实现用户目标选择最佳的云服务。

笔者的经验是，无论应用程序是运行在容器中，还是在虚拟机中，采用混合服务网格可以简化混合云应用程序管理，网络流量治理，并且实现安全性和可靠性。下面我们将讨论如何使用Istio来支持混合服务网格。

1.多集群**Istio**之一个控制平面

使用Istio管理多个Kubernetes集群（代表多个云）的一种方法是先部署一个Kubernetes集群，然后配置该集群连接到远程的Istio控制平面（同样运行在Kubernetes中）。需要注意的是，这两个集群中的Kubernetes Pod需要相互通信。这种多集群的架构常用的场景有：

·用户逐步将在测试集群中验证过的新功能过渡到生产集群；

·准备处理故障转移的备用集群；

·跨地域或可用区的冗余集群。

图6-7演示的是跨越两个不同的可用区（例如us-central和us-east，但底层网络仍然可达）部署多集群Istio。我们先在一个Kubernetes集群上安装Istio控制平面，在另一个Kubernetes集群上安装Istio的远程组件（包括sidecar代理注入器）。在这两个集群中，我们可以跨Kubernetes集群部署应用程序。

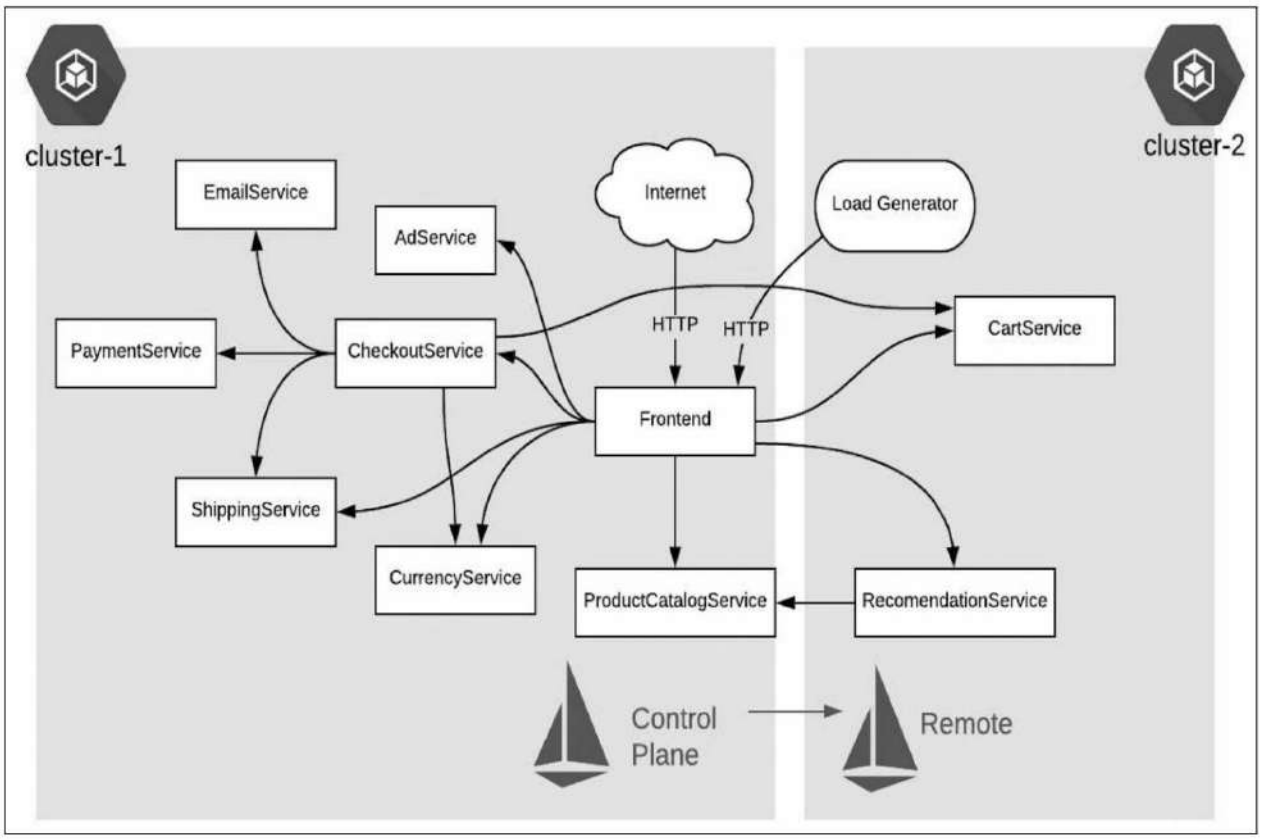


图6-7　多集群Istio之一个控制平面

这种单一控制平面的部署无须改变任何有关微服务之间相互通信的信息。例如，只要两个集群底层网络打通，前端仍然可以使用本地Kubernetes DNS域名（cartservice:port）访问CartService。

这是最基本的多集群Istio示例，下面进一步演示稍微复杂一点的部署方式。

2.多集群**Istio**之两个控制平面

假设在本地和云中或跨云平台运行应用程序，为了使Istio跨越这些不同的环境，两个集群内的Pod必须能够跨越网络边界。

图6-8演示了使用两个Istio控制平面，每个集群各一个，形成一个双头逻辑服务网格。两个集群间的流量交互是通过Istio的Ingress Gateway，而不是使用sidecar代理直接相互通信。Istio Ingress Gateway也是一个Envoy代理，但它专门用于进出集群Istio网格的流量。

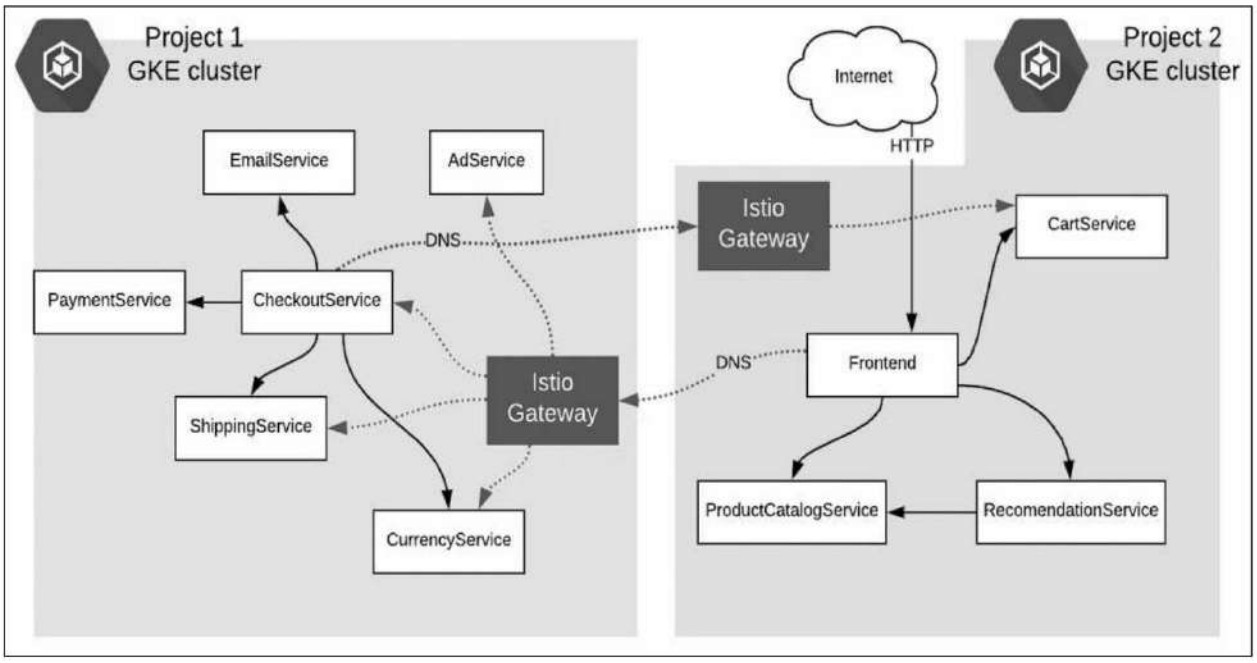


图6-8　多集群Istio之两个控制平面

在两个控制平面拓扑中，Istio安装辅助DNS服务器（CoreDNS），该服务器解析本地集群的外部服务的域名。对于那些外部服务，流量在Istio Ingress网关之间透传，然后转移到相关服务。

为了使跨两个集群运行的微服务能够互相通信，我们还通过Istio ServiceEntry完成此操作。Istio ServiceEntry用于手动添加不在当前服务网格内的服务，例如，我们将集群2的前端服务条目通过Istio ServiceEntry添加到集群1中，这样一来集群1就知道集群2中运行的服务。

与第一个演示不同，这种双控制平面的Istio设置不需要集群之间的扁平网络，只需要把Istio网关暴露在Internet上即可。这意味着两个集群之间的Pod允许有重叠的网断，而且每个集群内的服务可以在各自的网络环境中安全运行。

3.将虚拟机纳入**Istio**

鉴于很多遗留系统仍运行在虚拟机中，因此我们得想办法使用Istio把虚拟机和容器一起管理，让虚拟机应用也能享受Istio服务网格带来的福利。

图6-9演示了GKE（Google的容器服务）上运行Istio时集成Google GCE（Google的虚拟机服务）的实例。我们像以前一样部署相同的应用程序。但这一次，一个服务（ProductCatalog）被部署在Kubernetes集群之外的外部虚拟机上运行。

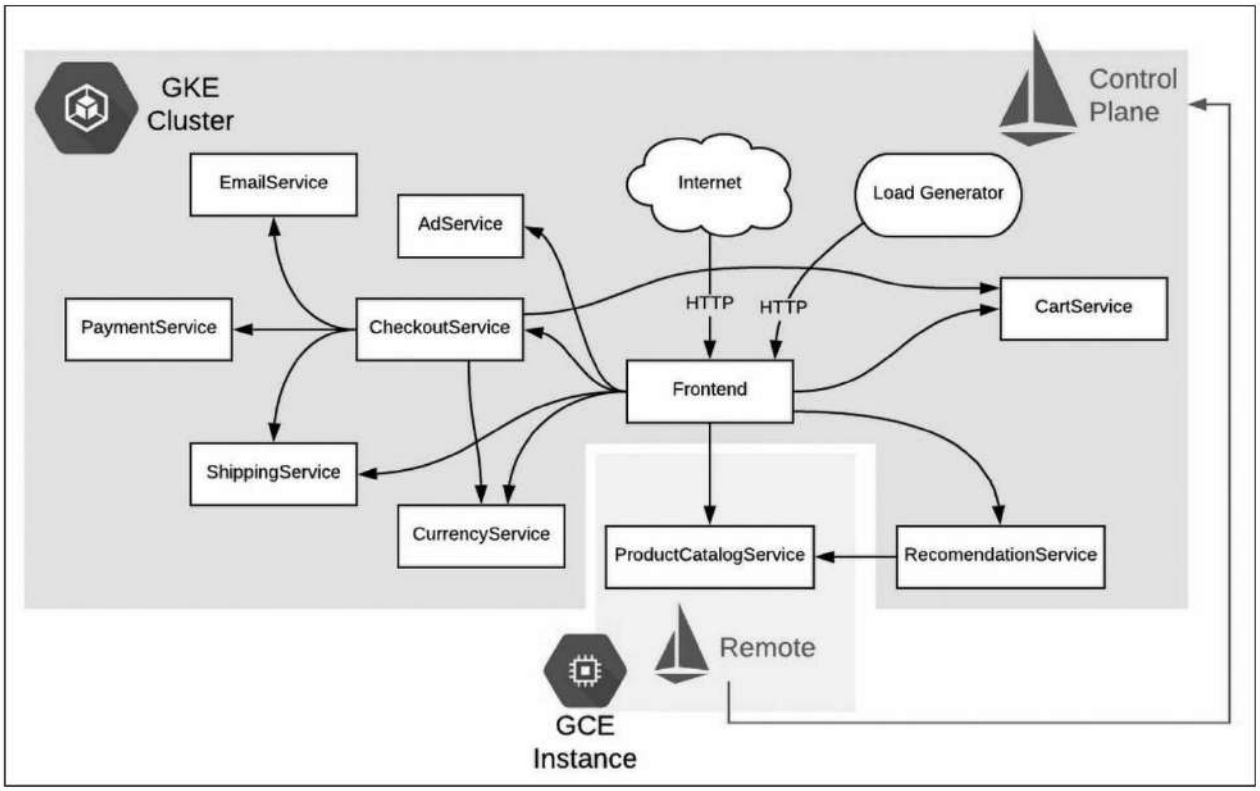


图6-9　GKE上运行I8tio时集成Google GCE的实例

该GCE虚拟机运行一组最小的Istio组件集，以便能够与中心的Istio控制平面通信。然后，我们将Istio　ServiceEntry对象部署到GKE集群上，该集群在逻辑上将外部Product-CatalogService添加到Istio网格中。

通过这种Istio配置模型，运行在虚拟机中的ProductCatalogService逻辑像是运行在Kubernetes集群内部一样。用户可以为ProductCatalogService添加Istio策略和规则，也可以为虚拟机的所有入站流量启用双向TLS等Istio服务治理的功能。

需要注意的是，虽然图6-9使用GCE进行演示，但用户可以在物理机上或使用本地虚拟机运行相同的示例。通过这种方式，用户可以将Istio的现代云原生原则带到在任何地方运行的虚拟机或物理机中。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

网络是Kubernetes技术体系中最复杂的一环，也体现了Kubernetes独特的设计理念。本书作者杜军曾是华为云原生团队核心成员，多个Kubernetes社区网络特性的贡献者与维护者。本书内容涵盖Kubernetes基础容器网络，4/7层服务发现、路由与注册管理，以及与服务网格的配合等方面内容，是不可多得的Kubernetes网络专著，推荐大家阅读。

华为云容器服务总经理，**方璞**

在Kubernetes技术体系飞速发展的过程中，其在弹性、可用性及可维护性方面日趋成熟，同时在敏捷性、提升研发效率和降低系统复杂度方面也表现卓越。敏捷开发、基础设施服务化、DevOps、ContainerOps深度融合，这就让普通开发者也可以快速地进入软件的构建中，可以让企业更多地将精力聚焦在应用层逻辑开发中。杜军是云原生技术的拥护者，Kubernetes社区核心成员，Kubernetes核心组件、网络服务的主要代码贡献者和维护者之一，也是一位优秀的技术分享者。本书对Kubernetes生态中颇为复杂的网络体系，从底层基础原理、概念、理论、关键技术点，到容器网络标准、模型、选型及实践，进行了深入浅出的论述，值得各位技术从业者仔细阅读和思考。

神州优车技术总监，**黄强元**

本书内容贯穿了整个虚拟网络演化历程，不仅探讨了容器网络和Kubernetes网络模型，更从底层原理到生产实践、再到源码解析和故障排查，层层递进，全方位、全视角地展现了整个云原生网络知识体系。容器、etcd、Kubernetes及istio是云原生技术生态的基石，本书作者杜军任Kubernetes和Istio项目的maintainer，为这些技术的发展做出了突出贡献。他也是多部畅销云原生技术书籍的作者，是业界认可的云原生技术专家。相信本书能给广大读者带来全新的知识体验，提供一场云原生技术全貌的饕餮盛宴。

北京虚云科技总经理，易宝支付前资深架构师，**李大伟**

云原生容器技术经过多年的进化和积累，已经被业界广泛认可，基于容器的技术和方案百花齐放，而网络也成为容器技术里最难攻克的一个领域。杜军对容器和虚拟化网络有深入的探索和实战经验，本书从虚拟化网络模型、协议、服务网格、DNS、网络策略、网络插件生态等方面进行了系统且深入的讲解，是虚拟化网络技术领域技术爱好者的一把利器。

西雅图华为云计算研究所资深架构师，**谢海滨**

自从Kubernetes被公认为构建平台的平台，无论是上游开源社区还是终端用户都将越来越多的注意力集中到编排层之上的技术创新。从Istio到Knative，再到OpenKruise，令人兴奋的技术创新层出不穷。而本书主题却有些“反流行”，用三百多页篇幅，从network namespace一直讲到Docker、Kubernetes网络模型和实现，从底层基础原理讲起，帮助开源技术的爱好者和使用者厘清当下主流容器网络标准及技术背后的机理，为技术选型和落地提供切实有用的帮助和指导，是对业界当前重上层创新的趋势的一个很好的补充。

浙江大学计算机系博士 谐云首席科学家，**丁轶群**

责任编辑：郑柳洁
封面设计：吴海燕

上架建议：计算机>容器

ISBN 978-7-121-37339-8

定价：89.00元