Docker运行原理

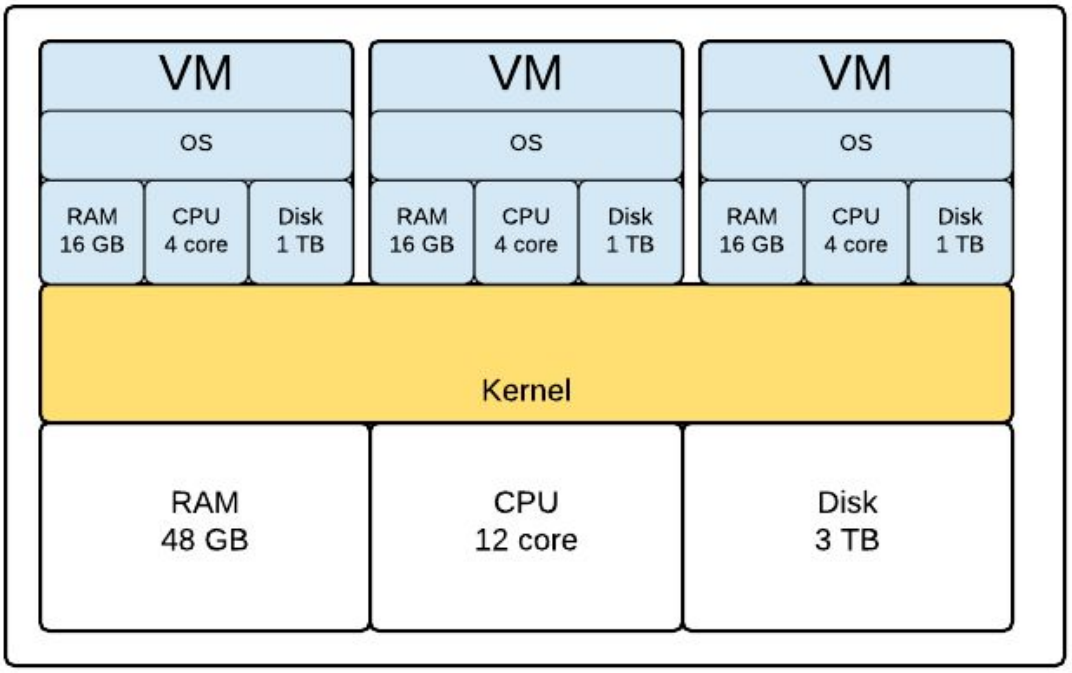

图 1 传统虚拟机原理图

如上图虽然虚拟机的隔离性是比较好的但是占用资源是比较多的，其次假如你在一个主机上创建了三个虚拟机。主机有 12 个 CPU，48 GB 内存和 3TB 的存储空间。每个虚拟机配置为有 4 个 CPU，16 GB 内存和 1TB 存储空间。到现在为止，一切都还好。主机有这个容量。但这里有个缺陷。所有分配给一个虚拟机的资源，无论是什么，都是专有的。每台机器都分配了 16 GB 的内存。但是，如果第一个虚拟机永不会使用超过 1GB 分配的内存，剩余的 15 GB 就会被浪费在那里。如果第三个虚拟机只使用分配的 1TB 存储空间中的 100GB，其余的 900GB 就成为浪费空间。

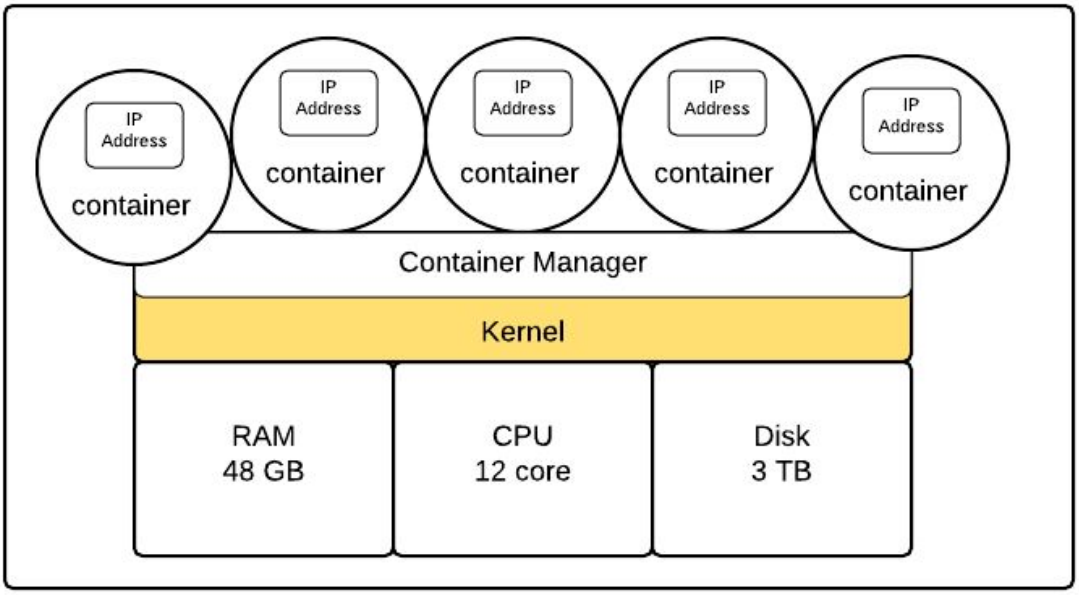

图 2 Linux容器原理图

概念上来说，容器是一个 Linux 进程，Linux 认为它只是一个运行中的进程。该进程只知道它被告知的东西。另外，在容器化方面，该容器进程也分配了它自己的 IP 地址。这点很重要，重要的事情讲三遍，这是第二遍。在容器化方面，容器进程有它自己的 IP 地址。一旦给予了一个 IP 地址，该进程就是宿主网络中可识别的资源。然后，你可以在容器管理器上运行命令，使容器 IP 映射到主机中能访问公网的 IP 地址。建立了该映射，无论出于什么意图和目的，容器就是网络上一个可访问的独立机器，从概念上类似于虚拟机。这是第三遍，容器是拥有不同 IP 地址从而使其成为网络上可识别的独立 Linux 进程

（参考https://linux.cn/article-6594-1.html）

要实现容器如上特性需要用到如下技术Namespace,Graphdriver,CGroup,Chroot,Iptable等.接下来将分别介绍这些技术.

1 Linux Namespace隔离

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

表 1 Linux Namespace隔离

a) 调用Namespace的API

Namespace的API包括clone()、setns()以及unshare()，还有/proc下的部分文件。为了确定隔离的到底是哪种namespace，在使用这些API时，通常需要指定以下六个常数的一个或多个，通过|（位或）操作来实现。你可能已经在上面的表格中注意到，这六个参数分别是CLONE_NEWIPC、CLONE_NEWNS、CLONE_NEWNET、CLONE_NEWPID、CLONE_NEWUSER和CLONE_NEWUTS。

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

参数child_func传入子进程运行的程序主函数。

参数child_stack传入子进程使用的栈空间

参数flags表示使用哪些CLONE_*标志位

参数args则可用于传入用户参数

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

参数fd表示我们要加入的namespace的文件描述符。上文已经提到，它是一个指向/proc/[pid]/ns目录的文件描述符，可以通过直接打开该目录下的链接或者打开一个挂载了该目录下链接的文件得到。

参数nstype让调用者可以去检查fd指向的namespace类型是否符合我们实际的要求。如果填0表示不检查。

b) UTS（UNIX Time-sharing System）namespace

UTS namespace提供了主机名和域名的隔离，这样每个容器就可以拥有了独立的主机名和域名，在网络上可以被视作一个独立的节点而非宿主机上的一个进程。下面我们通过代码来感受一下UTS隔离的效果，首先需要一个程序的骨架，如下所示。打开编辑器创建uts.c文件，输入如下代码。

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

```
#define STACK_SIZE (1024 * 1024)

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

int child_main(void* args) {
  printf("在子进程中!\n");
  execv(child_args[0], child_args);
  return 1;
}

int main() {
  printf("程序开始: \n");
  int child_pid = clone(child_main, child_stack + STACK_SIZE, SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

编译并运行上述代码，执行如下命令，效果如下。

```
root@local:~# gcc -Wall uts.c -o uts && ./uts
程序开始:
在子进程中!
root@local:~# exit
exit
已退出
root@local:~#
```

接下来加入UTS隔离

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

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

#define STACK_SIZE (1024 * 1024)

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

int child_main(void* args) {
  printf("在子进程中!\n");
  sethostname("NewNamespace", 12);
  execv(child_args[0], child_args);
  return 1;
}

int main() {
  printf("程序开始: \n");

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
    CLONE_NEWUTS | SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

编译并运行上述代码，执行如下命令，效果如下。

```
root@local:~# gcc -Wall namespace.c -o main.o && ./main.o
程序开始:
在子进程中!
root@NewNamespace:~# exit
exit
已退出
root@local:~#  <- 回到原来的hostname
```

总结：也许有读者试着不加CLONE_NEWUTS参数运行上述代码，发现主机名也变了，输

入exit以后主机名也会变回来，似乎没什么区别。实际上不加CLONE_NEWUTS参数进行隔离而使用sethostname已经把宿主机的主机名改掉了。你看到exit退出后还原只是因bash只在刚登录的时候读取一次UTS，当你重新登陆或者使用uname命令进行查看时，就会发现产生了变化。Docker中，每个镜像基本都以自己所提供的服务命名了自己hostname而没有对宿主机产生任何影响，用的就是这个原理。

c) IPC（Interprocess Communication）namespace

容器中进程间通信采用的方法包括常见的信号量、消息队列和共享内存。然而与虚拟机不同的是，容器内部进程间通信对宿主机来说，实际上是具有相同PID namespace中的进程间通信，因此需要一个唯一的标识符来进行区别。申请IPC资源就申请了这样一个全局唯一的32位ID，所以IPC namespace中实际上包含了系统IPC标识符以及实现POSIX消息队列的文件系统。在同一个IPC namespace下的进程彼此可见，而与其他的IPC namespace下的进程则互相不可见。

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

#define STACK_SIZE (1024 * 1024)

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

int child_main(void* args) {
  printf("在子进程中!\n");
  sethostname("NewNamespace", 12);
  execv(child_args[0], child_args);
  return 1;
}

int main() {
  printf("程序开始: \n");
```

```
  int child_pid = clone(child_main, child_stack+STACK_SIZE,
    CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

我们首先在shell中使用ipcmk -Q命令创建一个message queue。

```
root@local:~# ipcmk -Q
Message queue id: 32769
```

通过ipcs -q可以查看到已经开启的message queue，序号为32769。

```
root@local:~# ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x4cf5e29f 32769      root       644        0            0
```

然后我们可以编译运行加入了IPC namespace隔离的ipc.c，在新建的子进程中调用的shell中执行ipcs -q查看message queue。

```
root@local:~# gcc -Wall ipc.c -o ipc && ./ipc
程序开始:
在子进程中!
root@NewNamespace:~# ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
root@NewNamespace:~# exit
exit
已退出
```

上面的结果显示中可以发现，已经找不到原先声明的message queue，实现了IPC的隔离。目前使用IPC namespace机制的系统不多，其中比较有名的有PostgreSQL。Docker本身通过socket或tcp进行通信。

d) PID namespace

PID namespace隔离非常实用，它对进程PID重新标号，即两个不同namespace下的进程可以有同一个PID。每个PID namespace都有自己的计数程序。内核为所有的PID namespace维护了一个树状结构，最顶层的是系统初始时创建的，我们称之为root namespace。他创建的新PID namespace就称之为child namespace（树的子节点），而原先的PID namespace就是新创建的PID namespace的parent namespace（树的父节点）。通过这种方式，不同的PID namespaces会形成一个等级体系。所属的父节点可以看到子节点中的进程，并可以通过信号等方式对子节点中的进程产生影响。反过来，子节点不能看到父节点PID namespace中的任何内容。

1）每个PID namespace中的第一个进程“PID 1“，都会像传统Linux中的init进程一样拥有特权，起特殊作用。

2）一个namespace中的进程，不可能通过kill或ptrace影响父节点或者兄弟节点中的进程，因为其他节点的PID在这个namespace中没有任何意义。

3）如果你在新的PID namespace中重新挂载/proc文件系统，会发现其下只显示同属一个PID namespace中的其他进程。

4）在root namespace中可以看到所有的进程，并且递归包含所有子节点中的进程。

到这里，可能你已经联想到一种在外部监控Docker中运行程序的方法了，就是监Docker Daemon所在的PID namespace下的所有进程即其子进程，再进行删选即可。下面我们通过运行代码来感受一下PID namespace的隔离效果。修改上文的代码，加入PID namespace的标识位，并把程序命名为pid.c。

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

#define STACK_SIZE (1024 * 1024)

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

```
  NULL
};

int child_main(void* args) {
  printf("在子进程中!\n");
  sethostname("NewNamespace", 12);
  execv(child_args[0], child_args);
  return 1;
}

int main() {
  printf("程序开始: \n");

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
   CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

编译运行可以看到如下结果。

```
root@local:~# gcc -Wall pid.c -o pid && ./pid
程序开始:
在子进程中!
root@NewNamespace:~# echo $$
1                    <<--注意此处看到shell的PID变成了1
root@NewNamespace:~# exit
exit
已退出
```

打印$$可以看到shell的PID，退出后如果再次执行可以看到效果如下。

```
root@local:~# echo $$
17542
```

已经回到了正常状态。可能有的读者在子进程的shell中执行了ps aux/top之类的命令，发现还是可以看到所有父进程的PID，那是因为我们还没有对文件系统进行隔离，ps/top之类的命令调用的是真实系统下的/proc文件内容，看到的自然是所有的进程。

e) Mount namespace

Mount namespace通过隔离文件系统挂载点对隔离文件系统提供支持，它是历史上第一个Linux namespace，所以它的标识位比较特殊，就是CLONE_NEWNS。隔离后，不同mount namespace中的文件结构发生变化也互不影响。你可以通过/proc/[pid]/mounts查看到所有挂载在当前namespace中的文件系统，还可以通过/proc/[pid]/mountstats看到mount namespace中文件设备的统计信息，包括挂载文件的名字、文件系统类型、挂载位置等。

进程在创建 mount namespace 时，会把当前的文件结构复制给新的namespace。新namespace中的所有mount操作都只影响自身的文件系统，而对外界不会产生任何影响。这样做非常严格地实现了隔离，但是某些情况可能并不适用。比如父节点namespace中的进程挂载了一张CD-ROM，这时子节点namespace拷贝的目录结构就无法自动挂载上这张CD-ROM，因为这种操作会影响到父节点的文件系统。

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

#define STACK_SIZE (1024 * 1024)

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

int child_main(void* args) {
  printf("在子进程中!\n");
  sethostname("NewNamespace", 12);
  execv(child_args[0], child_args);
  return 1;
}
```

```
int main() {
  printf("程序开始: \n");

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
   CLONE_NEWNS |CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWUTS |
SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

f) Network namespace

Network namespace主要提供了关于网络资源的隔离，包括网络设备、IPv4和IPv6协议栈、IP路由表、防火墙、/proc/net目录、/sys/class/net目录、端口（socket）等等。一个物理的网络设备最多存在在一个network namespace中，你可以通过创建veth pair（虚拟网络设备对：有两端，类似管道，如果数据从一端传入另一端也能接收到，反之亦然）在不同的network namespace间创建通道，以此达到通信的目的。

当我们说到network namespace时，其实我们指的未必是真正的网络隔离，而是把网络独立出来，给外部用户一种透明的感觉，仿佛跟另外一个网络实体在进行通信。为了达到这个目的，容器的经典做法就是创建一个veth pair，一端放置在新的namespace中，通常命名为eth0，一端放在原先的namespace中连接物理网络设备，再通过网桥把别的设备连接进来或者进行路由转发，以此网络实现通信的目的。

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

#define STACK_SIZE (1024 * 1024)

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

int child_main(void* args) {
  printf("在子进程中!\n");
  sethostname("NewNamespace", 12);
  execv(child_args[0], child_args);
  return 1;
}
```

```
int main() {
  printf("程序开始: \n");
  int child_pid = clone(child_main, child_stack+STACK_SIZE,
   CLONE_NEWNET|CLONE_NEWNS |CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD,
NULL);
  waitpid(child_pid, NULL, 0);
  printf("已退出\n");
  return 0;
}
```

编译运行可以看到如下结果。

```
root@local:~# gcc -Wall pid.c -o pid && ./pid
程序开始:
在子进程中!
root@NewNamespace:~# ifconfig

root@NewNamespace:~#      <<--注意这里没有任何输出
```

g) User namespace

User namespace主要隔离了安全相关的标识符（identifiers）和属性（attributes），包括用户ID、用户组ID、root目录、key（指密钥）以及特殊权限。说得通俗一点，一个普通用户的进程通过clone()创建的新进程在新user namespace中可以拥有不同的用户和用户组。这意味着一个进程在容器外属于一个没有特权的普通用户，但是他创建的容器进程却属于拥有所有权限的超级用户，这个技术为容器提供了极大的自由。

User namespace是目前的六个namespace中最后一个支持的，并且直到Linux内核3.8版本的时候还未完全实现（还有部分文件系统不支持）。因为user namespace实际上并不算完全成熟，很多发行版担心安全问题，在编译内核的时候并未开启USER_NS。实际上目前Docker也还不支持user namespace，但是预留了相应接口，相信在不久后就会支持这一特性。所以在进行接下来的代码实验时，请确保你系统的Linux内核版本高于3.8并且内核编译时开启了USER_NS（如果你不会选择，可以使用Ubuntu14.04）。

```
sudo nsenter --target $pid --mount --pid --uts --net --ipc
```

2 Graph Driver

Docker 提供的 graphdriver 主要有 btrfs,aufs,overlayfs(3.18 以上 kernel 支持), zfs, devicemapper。

1. btrfs 文件系统本身支持快照功能，可以实现我们的需求，但是 btrfs 本身貌似目前还没有 productionready
2. aufs 文件系统是叠合文件系统，专门为了解决这种问题而生，可惜因为代码质量问题，一直被 linus 拒之门外，这样在很多发行版里都不会默认安装，用户使用需要自己编译对非 kernel 维护人员来说也比较麻烦。
3. overlayfs 文件系统也是叠合文件系统，但是需要 3.18 以上 kernel 版本。

结论是如果没有自己的 kernel 团队，那么目前唯一的选择似乎就是 devicemapper 了。下面将主要介绍Aufs

1）aufs

$ *sudo mount -t aufs -o br=./tmp=rw:./dira=ro:./dirb=ro:./dirc=ro -o udba=none none ./mnt*

`-t aufs` 指定文件系统类型为`aufs`

`-o` 后面是挂载选项，指定我们要挂载哪些目录

`none` 说明我们挂载的不是设备文件，因为这里我们是直接挂载目录的

`udba`有三种级别：`none`、`reval`、`inotify`，对性能的影响依次增加，当然安全性也有所增强。

`None`：这种检测是最快的，但可能导致错误的数据，例如在原始目录修改文件，在`aufs`中读取，不完全保证正确

`reval`：`aufs`会访问重新原始目录，如果文件有更新，在会反映在`aufs`中

`Notify`： 会在所有原始目录中的所有目录上注册`notify`事件，这会严重的影响性能，不建议使用。

如上命令将会将`dira`，`dirb`，`dirc` `mount`到`mnt`目录，在`mnt`目录下将可以看到这个目录下到所有内容，但是在`mnt`下所有添加到文件将全部保存在`dira`目录。

3 CGroup介绍

1）相关概念

任务（task）。在 cgroups 中，任务就是系统的一个进程；

控制族群（control group）。控制族群就是一组按照某种标准划分的进程。Cgroups 中的资源控制都是以控制族群为单位实现。一个进程可以加入到某个控制族群，也从一个进程组迁移到另一个控制族群。一个进程组的进程可以使用 cgroups 以控制族群为单位分配的资源，同时受到 cgroups 以控制族群为单位设定的限制；

层级（hierarchy）。控制族群可以组织成 hierarchical 的形式，既一颗控制族群树。控制族群树上的子节点控制族群是父节点控制族群的孩子，继承父控制族群的特定的属性；

子系统（subsystem）。一个子系统就是一个资源控制器，比如 cpu 子系统就是控制 cpu 时间分配的一个控制器。子系统必须附加（attach）到一个层级上才能起作用，一个子系统附加到某个层级以后，这个层级上的所有控制族群都受到这个子系统的控制。

2）子系统介绍

（a）blkio --为块设备设定输入/输出限制，比如物理设备（磁盘，固态硬盘，USB 等）。

（b）cpu -- 使用调度程序提供对CPU的cgroup任务访问。

（c）cpuacct --自动生成cgroup中任务所使用的CPU资源报告。

（d）cpuset --为cgroup中的任务分配独立CPU（在多核系统）和内存节点。

（e）devices --可允许或者拒绝中的任务对设备的访问。

（f）freezer --挂起或者恢复cgroup中的任务。

（g）memory --设定 cgroup 中任务使用的内存限制，并自动生成任务使用的内存资源报告。

（h）net_cls--使用等级识别符（classid）标记网络数据包，可允许 Linux 流量控制程序（tc）识别从具体 cgroup 中生成的数据包。

3）测试

查看cgroup所挂载目录

```
# mount | grep cgroup
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup
(rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib/systemd/systemd-cgroups-agent,name=systemd)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
```

CPU测试

```
# cd /sys/fs/cgroup/cpu
# mkdir cputest
# cd cputest
# echo $$ >> cgroup.procs
```

```
# echo ‘100000’ > cpu.cfs_period_us
# echo ‘50000’ > cpu.cfs_quota_us          # 50%
# cat ~/test.sh
#/bin/bash
c=1
while true
do
   c=$(($c + 1))
done

# chmod +x ~/test.sh
# ~/test.sh &
[1] 2584
# top
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2584	root	20	0	23012	2408	1836	R	49.3	0.1	0:34.73	bash

4 Chroot

chroot，即 change root directory (更改 root 目录)。在 linux 系统中，系统默认的目录结构都是以 `/`，即是以根 (root) 开始的。而在使用 chroot 之后，系统的目录结构将以指定的位置作为 `/` 位置。

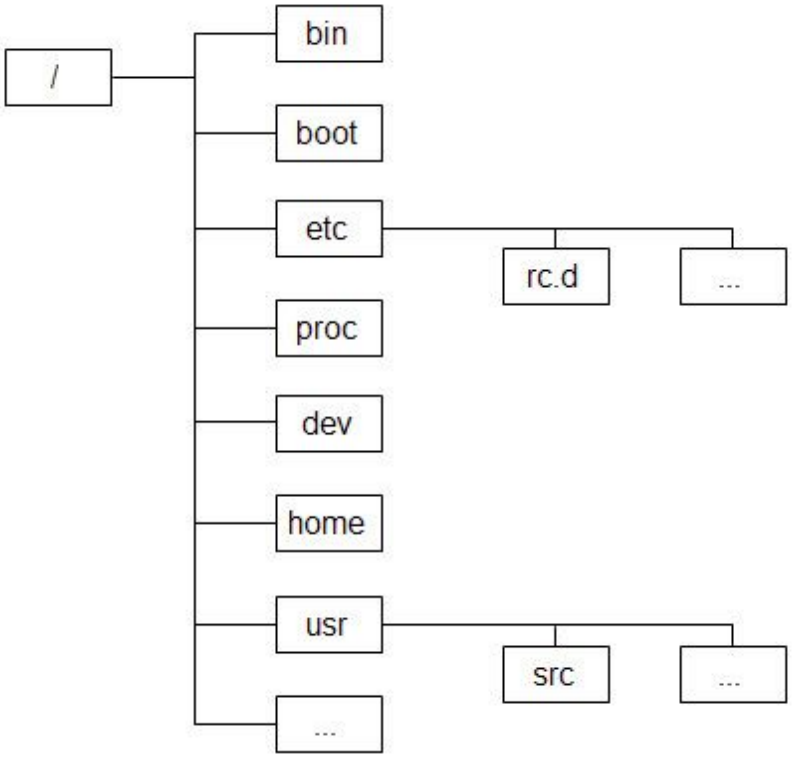

图 8.3 Linux系统的目录结构

```
$ mkdir myroot
$ mkdir myroot/bin myroot/lib myroot/lib64
```

$ which bash

/bin/bash

$ cp /bin/bash myroot/bin

$ ldd /bin/bash

```
        linux-vdso.so.1 =>  (0x00007ffdd555c000)
        libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f466aa48000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f466a844000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f466a47a000)
        /lib64/ld-linux-x86-64.so.2 (0x000055cac1b9f000)
```

$ cp /lib/x86_64-linux-gnu/libtinfo.so.5 myroot/lib

$ cp /lib/x86_64-linux-gnu/libdl.so.2 myroot/lib

$ cp /lib/x86_64-linux-gnu/libc.so.6 myroot/lib

$ cp /lib64/ld-linux-x86-64.so.2 myroot/lib64

$ sudo chroot myroot/ /bin/bash

```
bash-4.3# pwd
/
bash-4.3# ls
bash: ls: command not found
```

5 实现我们自己的docker

step1: 写一个自己namespace隔离的bash（由于为了更好的理解网络network namespace隔离通过ip netns add来实现）

$ cat mybash.c

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <sched.h>
#include<stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#define STACK_SIZE (1024 * 1024)

extern int errno ;

static char child_stack[STACK_SIZE];
char**  child_args;
char* machinename = "testmachine";

int child_main(void* args) {
  sethostname(machinename, strlen(machinename));

  execv(child_args[0], child_args);
  return 1;
}
```

```
int main(int argc ,char * argv[]) {
  child_args = (char* *)malloc((argc+2) * sizeof(char*));
  child_args[0]  = "/bin/bash";
  child_args[1] = NULL;

  if (argc == 2) {
     machinename = argv[1];
  } else if (argc > 2) {
     machinename = argv[1];
     for (int i = 2; i < argc ; i++) {
         child_args[i-2] = argv[i];
     }
     child_args[argc-2] = NULL;
  }

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
   CLONE_NEWNS |CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWUTS | SIGCHLD, NULL);
  waitpid(child_pid, NULL, 0);
  return 0;
}
```

$ *gcc -Wall mybash.c -o mybash*

step2: 创建docker相关网络

$ sudo brctl addbr mydocker #创建mydocker网桥对应docker的docker0
$ sudo ip link set mydocker up #默认设备没有开启需要手动打开
$ sudo ip address add 10.10.14.1 dev mydocker # 设置网桥地址
$ sudo ip netns add ns1 # 创建ns1的网络namespace
$ sudo ip link add type veth # 创建veth设备对，类似现实生活中的网线
$ sudo ip link set veth0 netns ns1 #将veth设备对的veth0放入ns1网络namespace中
$ sudo ip netns exec ns1 ip address add 10.10.14.2 dev veth0 #设置ns1中veth0的ip地址
$ sudo ip netns exec ns1 ip link set dev veth0 name eth0 # 将ns1中的veth0命名为eth0
$ sudo ip netns exec ns1 ip link set dev eth0 up # 开启ns1中的eth0
$ sudo ip netns exec ns1 ip link set lo up # 开启ns1中的lo环回设备
$ sudo ip link set dev veth1 name machine1 # 将系统网络namespace中的veth1命名为machine1
$ sudo ip link set dev machine1 up # 开启系统网络namespace中的machine1设备
$ sudo brctl addif mydocker machine1 # 将machine1插入mydocker网桥
$ ip ad

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000
    link/ether 08:00:27:c3:a0:78 brd ff:ff:ff:ff:ff:ff
    inet 10.19.131.153/22 brd 10.19.131.255 scope global dynamic enp0s3
       valid_lft 170228sec preferred_lft 170228sec
    inet6 fe80::e569:fff3:4b83:84e2/64 scope link
       valid_lft forever preferred_lft forever
```

```
3: mydocker: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
default qlen 1000
    link/ether ca:dc:39:4b:c3:a1 brd ff:ff:ff:ff:ff:ff
    inet 10.11.0.1/32 scope global mydocker
       valid_lft forever preferred_lft forever
    inet6 fe80::1091:60ff:fec9:2e04/64 scope link
       valid_lft forever preferred_lft forever
5: machine1@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
mydocker state UP group default qlen 1000
    link/ether ca:dc:39:4b:c3:a1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::c8dc:39ff:fe4b:c3a1/64 scope link
       valid_lft forever preferred_lft forever
$ sudo ip netns exec ns1 ip ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
default qlen 1000
    link/ether 7e:fc:0b:fd:07:89 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.11.0.2/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::7cfc:bff:fefd:789/64 scope link
       valid_lft forever preferred_lft forever
```

step3: 创建相关目录及文件

```
$ mkdir image
$ mkdir image/bin image/lib image/lib64 image/proc
$ cp mybash image/bin/
$ cp busybox image/bin/     #在网上下的静态编译的无需依赖其它so
$ which bash
/bin/bash
$ cp /bin/bash image/bin
$ ldd /bin/bash
        linux-vdso.so.1 =>  (0x00007ffe514c8000)
        libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007ff1e0fdc000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007ff1e0dd8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff1e0a0e000)
        /lib64/ld-linux-x86-64.so.2 (0x000055bd6621d000)
$ cp  /lib/x86_64-linux-gnu/libtinfo.so.5 image/lib
$ cp /lib/x86_64-linux-gnu/libdl.so.2 image/lib
$ cp  /lib/x86_64-linux-gnu/libc.so.6 image/lib
$ cp /lib64/ld-linux-x86-64.so.2 image/lib64
$ tree image
image/
├── bin
│   ├── bash
```

```
│   ├── busybox
│   └── mybash
├── lib
│   ├── libc.so.6
│   ├── libdl.so.2
│   └── libtinfo.so.5
└── lib64
    └── ld-linux-x86-64.so.2

3 directories, 7 files
```

$ mkdir tmp mnt
$ sudo mount -v -t aufs -obr=./tmp:./image -o udba=none none ./mnt

step4: 启动mydocker
$ sudo route add -net 10.10.14.0 netmask 255.255.255.0 dev mydocker
$ sudo route add -net 10.10.15.0 netmask 255.255.255.0 gw 192.168.122.183
$ sudo sysctl -w net.ipv4.ip_forward=1
$ sudo iptables -t nat -A POSTROUTING -s 10.10.0.0/16 -d 10.10.0.0/16 -j RETURN
$ sudo iptables -t nat -A POSTROUTING -s 10.10.0.0/16 ! -d 224.0.0.0/4 -j MASQUERADE
$ sudo ip netns exec ns1 chroot ./mnt /bin/mybash
bash-4. 3# busybox mount -t proc proc /proc
bash-4. 3# busybox route add -net 10.10.14.0 netmask 255.255.255.0 dev eth0
bash-4. 3# busybox route add default gw 10.10.14.1

6 Iptables 详解

iptables的前身叫ipfirewall （内核1.x时代）,这是一个作者从freeBSD上移植过来的，能够工作在内核当中的，对数据包进行检测的一款简易访问控制工具。但是ipfirewall工作功能极其有限(它需要将所有的规则都放进内核当中，这样规则才能够运行起来，而放进内核，这个做法一般是极其困难的)。当内核发展到2.x系列的时候，软件更名为ipchains，它可以定义多条规则，将他们串起来，共同发挥作用，而现在，它叫做iptables，可以将规则组成一个列表，实现绝对详细的访问控制功能。

他们都是工作在用户空间中，定义规则的工具，本身并不算是防火墙。它们定义的规则，可以让在内核空间当中的netfilter来读取，并且实现让防火墙工作。而放入内核的地方必须要是特定的位置，必须是tcp/ip的协议栈经过的地方。而这个tcp/ip协议栈必须经过的地方，可以实现读取规则的地方就叫做 netfilter.(网络过滤器)

6.1 Iptable里的表和链

1.filter表：*filter*表是专门过滤包 的，内建三个链，可以毫无问题地对包进行**DROP**、**LOG**、**ACCEPT**和**REJECT**等操作。*FORWARD* 链过滤所有不是本地产生的并且目的地不是本地（所谓本地就是防火墙了）的包，而 *INPUT*恰恰针对那些目的地是本地的包。*OUTPUT* 是用来过滤所有本地生成的包的。

2 nat表：的主要用处是网络地址转换，即Network Address Translation，缩写为NAT。做过NAT操作的数据包的地址就被改变了，当然这种改变是根据我们的规则进行 的。属于一个流的包只会经过这个表一次。如果第一个包被允许做NAT或Masqueraded，那么余下的包都会自 动地被做相同的操作。也就是说，余下的包不会再通过这个表，一个一个的被NAT，而是自动地完成。这就 是我们为什么不应该在这个表中做任何过滤的主要原因，对这一点，后面会有更加详细的讨论。PREROUTING 链的作用是在包刚刚到达防火墙时改变它的目的地址，如果需要的话。OUTPUT链改变本地产生的包的目的地 址。POSTROUTING链在包就要离开防火墙之前改变其源地址。
3.mangle表：这个表主要用来mangle数据包。我们可以改变不同的包及包 头的内容，比如**TTL**，**TOS**或**MARK**。 注意**MARK**并没有真正地改动数据包，它只是在内核空间为包设了一个标记。防火墙 内的其他的规则或程序（如tc）可以使用这种标记对包进行过滤或高级路由。这个表有五个内建的链： *PREROUTING*，*POSTROUTING*， *OUTPUT*，*INPUT*和*FORWARD*。*PREROUTING*在包进入防火墙之后、路由判断之前改变 包，*POSTROUTING*是在所有路由判断之后。 *OUTPUT*在确定包的目的之前更改数据包。*INPUT*在包被路由到本地 之后，但在用户空间的程序看到它之前改变包。*FORWARD*在最初的路由判 断之后、最后一次更改包的目的之前mangle包。注意，mangle表不能做任何NAT，它只是改变数据包的**TTL**，**TOS**或**MARK**，而不是其源目地 址。NAT是在nat表中操作的。

五个钩子函数（hook functions）,也叫五个规则链。这是NetFilter规定的五个规则链，任何一个数据包，只要经过本机，必将经过这五个链中的其中一个链。

1.PREROUTING (路由前)
2.INPUT (数据包流入口)
3.FORWARD (转发管卡)
4.OUTPUT(数据包出口)
5.POSTROUTING（路由后）

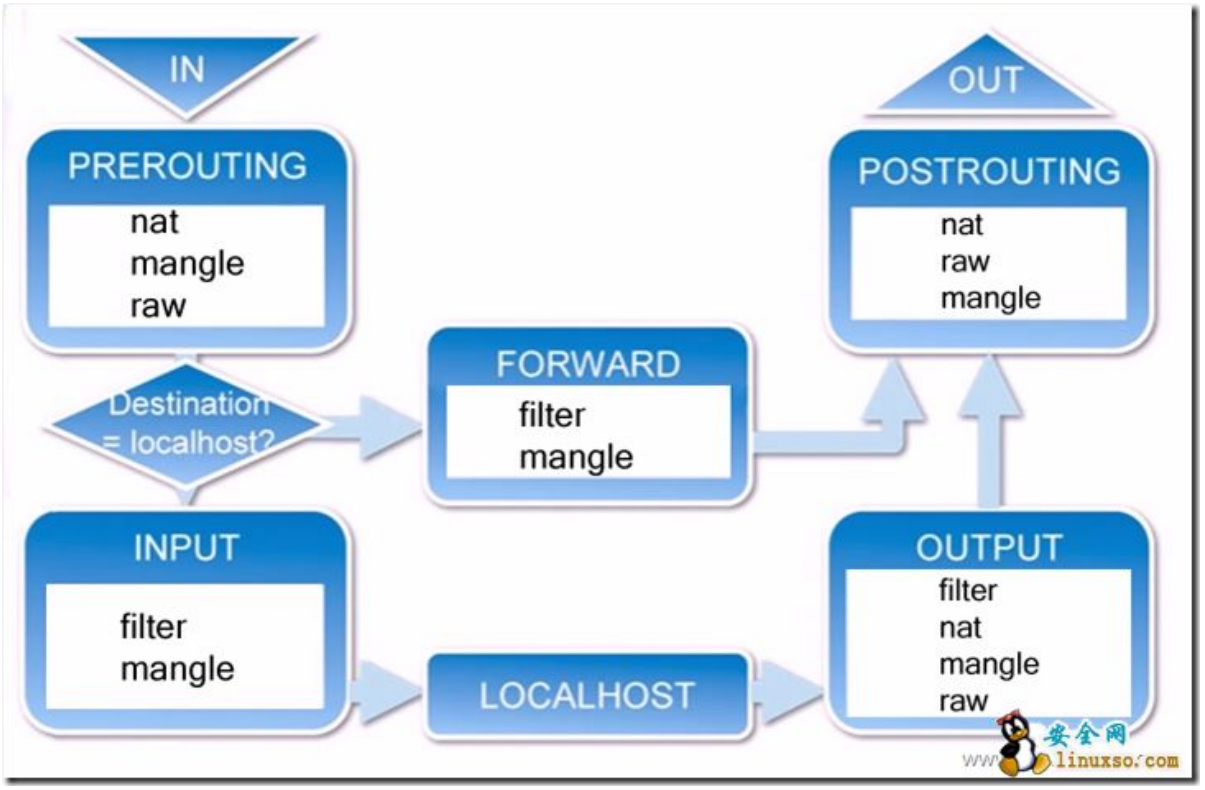

图 4 Iptable数据处理流程图1

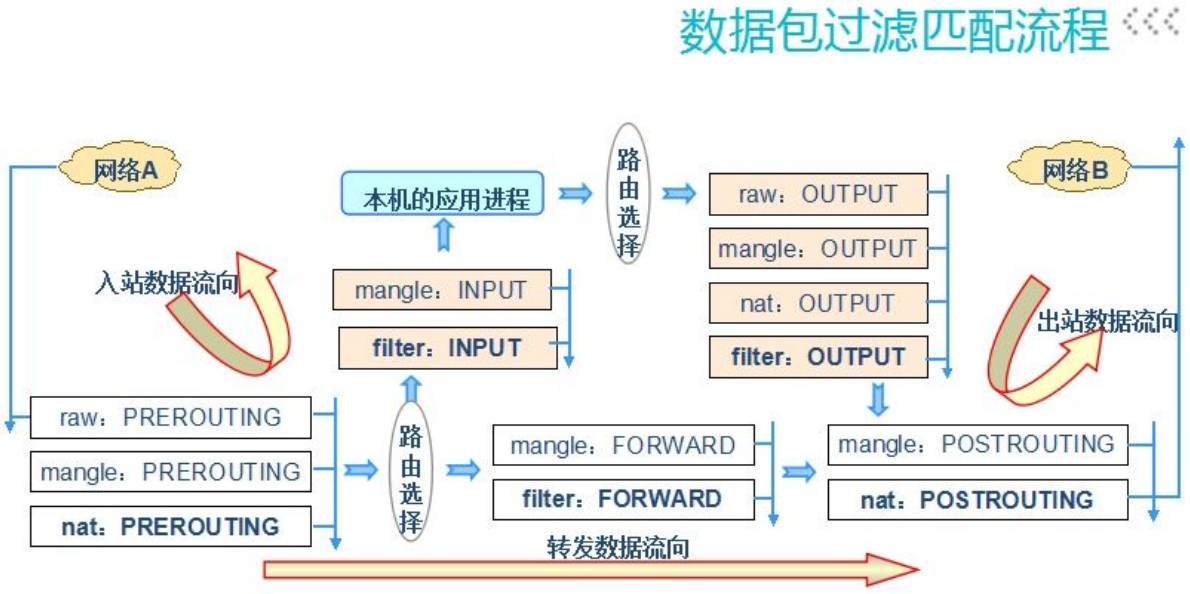

图 5 Iptable数据处理流程图2

6.2 Iptable命令

iptables定义规则的方式比较复杂:
格式：iptables [-t table] COMMAND chain MATCH -j ACTION
-t table ：如filter nat mangle
COMMAND：定义如何对规则进行管理

-A：追加，在当前链的最后新增一个规则
-I num：插入，把当前规则插入为第几条。
-I 3 :插入为第三条
-R num：Replays替换/修改第几条规则
格式：iptables -R 3 ……
-D num：删除，明确指定删除第几条规则
-N ：创建一条新链
-F：清空链里的规则
-X：清空所有创建的链

chain：指定你接下来的规则到底是在哪个链上操作的，当定义策略的时候，是可以省略的
MATCH:指定匹配标准我把它们归为五类。第一类是generic matches（通用的匹配），适用于所有的规则；第二类是TCP matches，顾名思 义，这只能用于TCP包；第三类是UDP matches， 当然它只能用在UDP包上了；第四类是ICMP matches，针对ICMP包的；第五类比较特殊，针对的是状态（state），所有 者（owner）和访问的频率限制（limit）等，它们已经被分到更多的小类当中，尽管它们并不是完全不同 的。我希望这是一种大家都容易理解的分类。
-j ACTION :指定如何进行处理
常用的ACTION：
DROP：悄悄丢弃
一般我们多用DROP来隐藏我们的身份，以及隐藏我们的链表
REJECT：明示拒绝
ACCEPT：接受
custom_chain：转向一个自定义的链
DNAT
SNAT
MASQUERADE：源地址伪装
REDIRECT：重定向：主要用于实现端口重定向
MARK：打防火墙标记的
RETURN：返回
在自定义链执行完毕后使用返回，来返回原规则链。

如：
1 设置chain策略
对于filter table，默认的chain策略为ACCEPT，我们可以通过以下命令修改chain的策略：
```
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT DROP
```

2 屏蔽指定ip
有时候我们发现某个ip不停的往服务器发包，这时我们可以使用以下命令，将指定ip发来的包丢弃：
```
BLOCK_THIS_IP="x.x.x.x"
iptables -A INPUT -i eth0 -p tcp -s "$BLOCK_THIS_IP" -j DROP
```

3 网口转发配置

对于用作防火墙或网关的服务器，一个网口连接到公网，其他网口的包转发到该网口实现内网向公网通信，假设eth0连接内网，eth1连接公网，配置规则如下：

```
iptables -A FORWARD -i eth0 -o eth1 -j ACCEPT
```

4 配置服务项

利用iptables，我们可以对日常用到的服务项进行安全管理，比如设定只能通过指定网段、由指定网口通过SSH连接本机：

```
iptables -A INPUT -i eth0 -p tcp -s 192.168.100.0/24 --dport 22 -m state --state NEW,ESTABLESHED -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 22 -m state --state ESTABLISHED -j ACCEPT
```

5 端口转发配置

```
iptables -t nat -A PREROUTING -p tcp -d 192.168.102.37 --dport 422 -j DNAT --to 192.168.102.37:22
```

以上命令将422端口的包转发到22端口，因而通过422端口也可进行SSH连接

6 DoS攻击防范

利用扩展模块limit，我们还可以配置iptables规则，实现DoS攻击防范：

```
iptables -A INPUT -p tcp --dport 80 -m limit --limit 25/minute --limit-burst 100 -j ACCEPT
```

--litmit 25/minute 指示每分钟限制最大连接数为25

--litmit-burst 100 指示当总连接数超过100时，启动 litmit/minute 限制

7 跳转到子链

```
iptables -N mytcpchain
iptables -A INPUT -p tcp -j mytcpchain
```

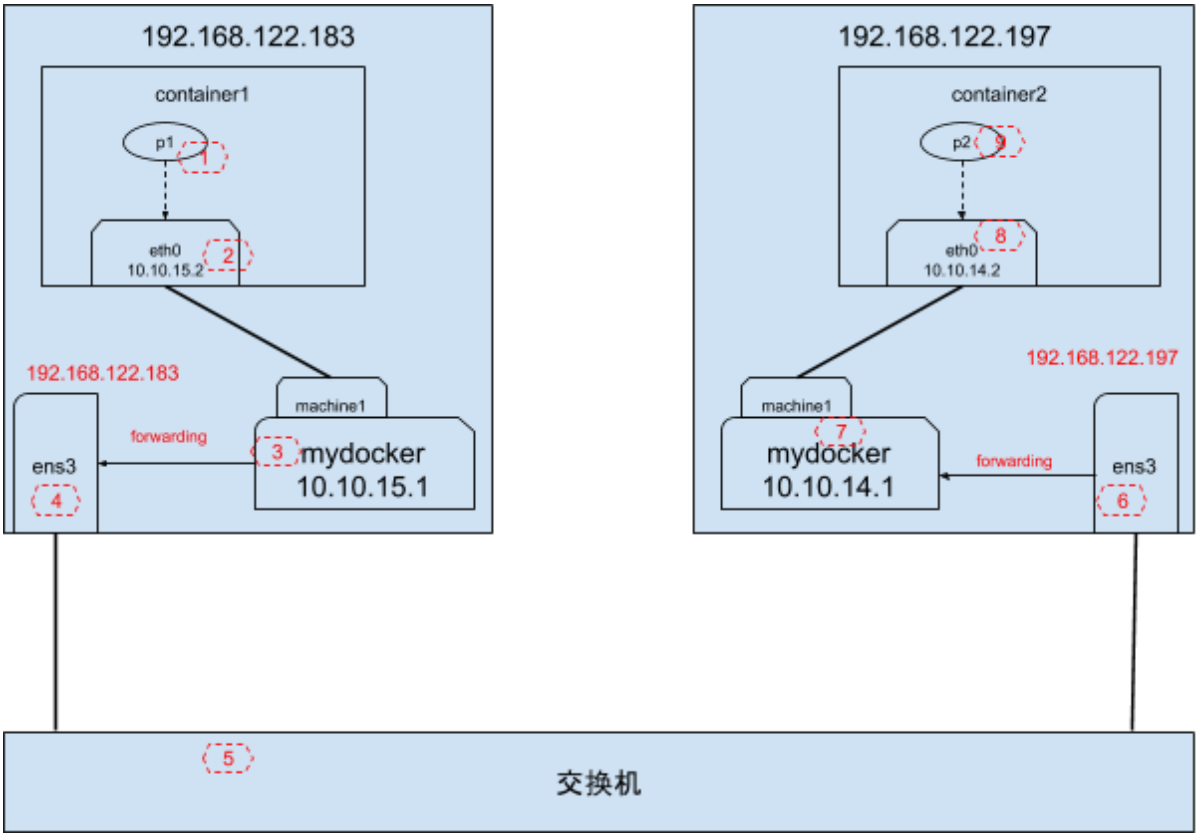

图6 p1进程向p2进程发送数据包的数据流

如上图为192.168.122.183上的一个进程p1向192.168.122.197上的进程p2发送数据包的整个流程(接下来将以ping为例子详细介绍整个包的流向).

1) 在container1里执行busybox ping 10.10.14.2 (container路由表如下):

```
bash-4.3# busybox route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.10.15.1      0.0.0.0         UG    0      0        0 eth0
10.10.15.0      0.0.0.0         255.255.255.0   U     0      0        0 eth0
bash-4.3#
```

从路由表可以看出到10.10.14.2的路由只有一条到10.10.15.1（183上mydocker的ip), 所以该ping的icmp包的ip源地址为10.10.15.2，ip目的地址为10.10.14.2，mac源地址为container1的mac，mac目的地址为mydocker的mac(mydocker是container1的默认网关) 如下为tcpdump抓包。

```
k8s@node2-183:~$ sudo ip netns exec ns1 tcpdump -i eth0 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
                                                    mydocker MAC
10:30:23.207903 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93 ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 0, length 64
10:30:23.209850 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 0, length 64
```

2) 由于container1的eth0网卡和183上的machine1为一对veth网卡(往其中一个写在另一个就可以读到和container1中eth0上dump的一样如下所示。

```
k8s@node2-183:~$ sudo tcpdump -i machine1 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on machine1, link-type EN10MB (Ethernet), capture size 262144 bytes
10:43:20.463764 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 777, length 64
10:43:20.464838 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 777, length 64
```

3) 由于machine1设备已经被加到mydocker网桥上且数据包的目的mac地址为mydocker的mac，所以数据包被mydocker接收如下所示。

```
[k8s@node2-183:~$ sudo tcpdump -i mydocker icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mydocker, link-type EN10MB (Ethernet), capture size 262144 bytes
10:51:09.646584 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 1246, length 64
10:51:09.647001 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 1246, length 64
```

虽然数据包的目的mac地址为mydocker自己但是目的ip(目的ip10.10.14.2自己的IP是10.10.15.1)不是自己所以它需要将数据包转发出去(默认系统内核是没有开启转发功能的因此数据包在此将会被丢弃，如要开启可以*sysctl -w net.ipv4.ip_forward=1*曾经在这里被卡了很久不知道为什么数据包到这里就丢了tcpdump在mydocker上抓不到数据）。转发就需要查路由表(如下为183上的路由表),目的地址10.10.14.0将走ens3网关为192.168.122.197

```
[k8s@node2-183:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.122.1   0.0.0.0         UG    100    0        0 ens3
10.10.14.0      192.168.122.197 255.255.255.0   UG    0      0        0 ens3
10.10.15.0      0.0.0.0         255.255.255.0   U     0      0        0 mydocker
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens3
192.168.122.0   0.0.0.0         255.255.255.0   U     100    0        0 ens3
```

4) 数据包在mydocker经转发到网卡ens3, tcpdump抓包如下(因为要经过网关所以目的MAC被替换成网关197的MAC).

```
[k8s@node2-183:~$ sudo tcpdump -i ens3 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
11:04:50.948205 52:54:00:1e:45:e1 > 52:54:00:1e:45:ef, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 2067, length 64
11:04:50.948582 52:54:00:1e:45:ef > 52:54:00:1e:45:e1, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 2067, length 64
```

197 ens3 MAC

5) 由于183和197的ens3网卡多是接在同一个交换机上的所以从183上发出的数据包将被197的ens3网卡所接收(之所以是被197所接收是因为数据包的目的mac为197上ens3网卡的mac), 如下为197上的抓包数据。

```
[k8s@node1-197:~$ sudo tcpdump -i ens3 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
11:16:56.156416 52:54:00:1e:45:e1 > 52:54:00:1e:45:ef, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 2792, length 64
11:16:56.156572 52:54:00:1e:45:ef > 52:54:00:1e:45:e1, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 2792, length 64
```

197的ens3接到数据包之后发现目的ip不是自己所以也需要转发(和之前在183上的mydocker一样在197上也需要开启内核转发功能否则在这里数据也抓不到),根据197上的路由表如下10.10.14.0数据包将被转发到mydocker网桥。

```
[k8s@node1-197:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.122.1   0.0.0.0         UG    100    0        0 ens3
10.10.14.0      0.0.0.0         255.255.255.0   U     0      0        0 mydocker
10.10.15.0      192.168.122.183 255.255.255.0   UG    0      0        0 ens3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens3
192.168.122.0   0.0.0.0         255.255.255.0   U     100    0        0 ens3
```

6) 如上数据包将被转发到mydocker上抓包如下，但是数据包的目的mac被替换成container2 eth0的mac地址。

```
[k8s@node1-197:~$ sudo tcpdump -i mydocker icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mydocker, link-type EN10MB (Ethernet), capture size 262144 bytes
11:26:44.376132 fe:54:ef:48:ea:6d > 42:4e:00:a4:76:e3, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 3380, length 64
11:26:44.376265 42:4e:00:a4:76:e3 > fe:54:ef:48:ea:6d, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 3380, length 64
```

container2 eth0 MAC

7) 数据包转发到197 mydocker上后会被再次转发到所有插在mydocker上到设备上（这就是网桥的工作原理）。所以197 machine1将收到数据包如下抓包所示：

```
[k8s@node1-197:~$ sudo tcpdump -i machine1 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on machine1, link-type EN10MB (Ethernet), capture size 262144 bytes
11:37:20.630327 fe:54:ef:48:ea:6d > 42:4e:00:a4:76:e3, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 4016, length 64
11:37:20.630485 42:4e:00:a4:76:e3 > fe:54:ef:48:ea:6d, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 4016, length 64
```

8) 由于machine1和container2里eth0为一个veth设备对，所以数据包也会被container2 eth0所接收(因为数据包的目的mac就是自己的mac，插在mydocker上的其它container里的eth0将不会接收数据因为目的mac和自己不一样）。

```
k8s@node1-197:~$ sudo ip netns exec ns1 tcpdump -i eth0 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:41:44.728346 fe:54:ef:48:ea:6d > 42:4e:00:a4:76:e3, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 10.10.14.2: ICMP echo request, id 1792, seq 4280, length 64
11:41:44.728397 42:4e:00:a4:76:e3 > fe:54:ef:48:ea:6d, ethertype IPv4 (0x0800), length 98: 10.10.14.2 > 10.10.15.2: ICMP echo reply, id 1792, seq 4280, length 64
```

至此icmp数据包从183上的container1上已经到了197上到container2上，但这只是request数据包，但是从container2上返回的reply数据包也一样经过如上原理返回container1,这才是一个完整的ping数据流，其中任何一个地方有问题多将导致ping不通。如上就是k8s上不同node上pod间网络通信的原理（flannel 工作在host gateway模式下）。

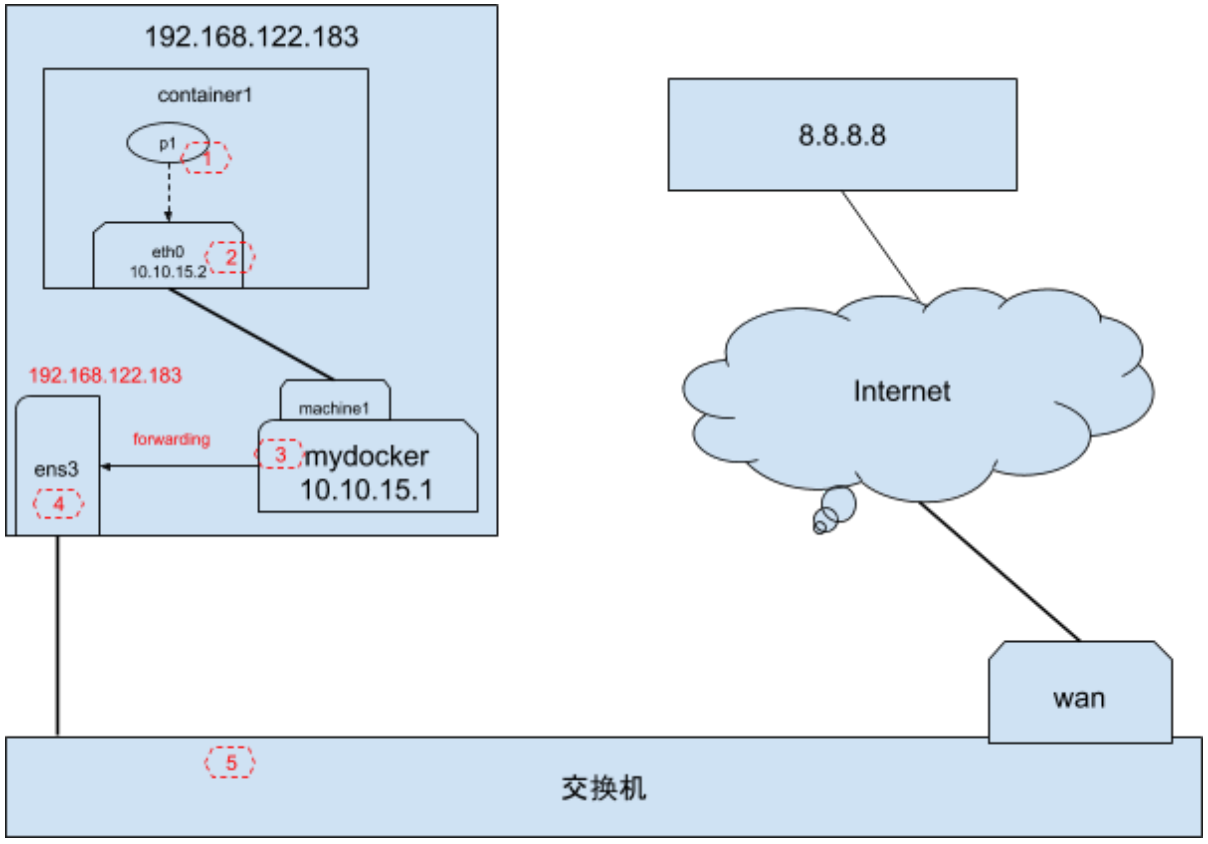

图7 p1进程向Internet上某台服务区发送数据包的数据流

如上图为192.168.122.183上的一个进程p1向8.8.8.8的进程发送数据包的整个流程(接下来将以ping为例子详细介绍整个包的流向).

1) 在container1里执行busybox ping 8.8.8.8 (container路由表如下):

```
bash-4.3# busybox route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.10.15.1      0.0.0.0         UG    0      0        0 eth0
10.10.15.0      0.0.0.0         255.255.255.0   U     0      0        0 eth0
bash-4.3#
```

从路由表可以看出到8.8.8.8 的路由只有一条到10.10.15.1（183上mydocker的ip), 所以该ping的icmp包的ip源地址为10.10.15.2，ip目的地址为8.8.8.8 mac源地址为container1的mac，mac目的地址为mydocker的mac(mydocker是container1的默认网关) 如下为tcpdump抓包。

```
k8s@node2-183:~$ sudo ip netns exec ns1 tcpdump -i eth0 icmp -n -e
[sudo] password for k8s:
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:01:33.500742 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 8.8.8.8: ICMP echo request, id 2304, seq 0, length 64
12:01:33.577551 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 8.8.8.8 > 10.10.15.2: ICMP echo reply, id 2304, seq 0, length 64
```

mydocker MAC

2) 由于container1的eth0网卡和183上的machine1为一对veth网卡(往其中一个写在另一个就可以读到和container1中eth0上dump的一样如下所示。

```
k8s@node2-183:~$ sudo tcpdump -i machine1 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on machine1, link-type EN10MB (Ethernet), capture size 262144 bytes
12:10:26.640234 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 8.8.8.8: ICMP echo request, id 2560, seq 527, length 64
12:10:26.716513 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 8.8.8.8 > 10.10.15.2: ICMP echo reply, id 2560, seq 527, length 64
```

3) 由于machine1设备已经被加到mydocker网桥上且数据包的目的mac地址为mydocker的mac，所以数据包被mydocker接收如下所示。

```
k8s@node2-183:~$ sudo tcpdump -i mydocker icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mydocker, link-type EN10MB (Ethernet), capture size 262144 bytes
12:12:45.717413 c2:f1:0d:f9:b8:49 > fa:49:41:17:2e:93, ethertype IPv4 (0x0800), length 98: 10.10.15.2 > 8.8.8.8: ICMP echo request, id 2560, seq 666, length 64
12:12:45.793733 fa:49:41:17:2e:93 > c2:f1:0d:f9:b8:49, ethertype IPv4 (0x0800), length 98: 8.8.8.8 > 10.10.15.2: ICMP echo reply, id 2560, seq 666, length 64
```

虽然数据包的目的mac地址为mydocker自己但是目的ip(目的ip 8.8.8.8自己的IP是10.10.15.1)不是自己所以它需要将数据包转发出去(默认系统内核是没有开启转发功能的因此数据包在此将会被丢弃，如要开启可以*sysctl -w net.ipv4.ip_forward=1*曾经在这里被卡了很久不知道为什么数据包到这里就丢了tcpdump在mydocker上抓不到数据）。转发就需要查路由表(如下为183上的路由表),目的地址0.0.0.0(默认路由)将走ens3网关为192.168.122.1

```
k8s@node2-183:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.122.1   0.0.0.0         UG    100    0        0 ens3
10.10.14.0      192.168.122.197 255.255.255.0   UG    0      0        0 ens3
10.10.15.0      0.0.0.0         255.255.255.0   U     0      0        0 mydocker
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens3
192.168.122.0   0.0.0.0         255.255.255.0   U     100    0        0 ens3
```

4) 数据包在mydocker经转发到网卡ens3, tcpdump抓包如下(因为要经过默认网关所以目的MAC被替换成网关192.168.122.1的MAC).

```
k8s@node2-183:~$ sudo tcpdump -i ens3 icmp -n -e
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens3, link-type EN10MB (Ethernet), capture size 262144 bytes
12:17:01.803761 52:54:00:1e:45:e1 > fe:54:00:1e:45:e1, ethertype IPv4 (0x0800), length 98: 192.168.122.183 > 8.8.8.8: ICMP echo request, id 2560, seq 922, length 64
12:17:01.881032 fe:54:00:1e:45:e1 > 52:54:00:1e:45:e1, ethertype IPv4 (0x0800), length 98: 8.8.8.8 > 192.168.122.183: ICMP echo reply, id 2560, seq 922, length 64
```

192.168.122.1 MAC

这里和之前ping 10.10.14.2不同的是源IP地址被替换成192.168.122.183（如果不替换8.8.8.8返回的数据包将无法返回（返回数据包到192.168.122.1时将不知道将数据包转给谁，当然数据包在进入Internet的时候还会再次被替换成一个真正的外网地址）。但是这里源IP地址是被谁替换的呢答案是之前介绍的Iptables(在docker环境下是由docker创建的），如下所示如果源地址为10.10.0.0将做SNAT，但是为了兼容之前介绍的ping内网的10.10.14.2时是不需要做SNAT的所以在这条Iptables还加了一条只要目的地址是10.10.0.0网段的将直接返回从而不会做SNAT。

```
k8s@node2-183:~$ sudo iptables-save -t nat | grep POSTROUTING
:POSTROUTING ACCEPT [60:5342]
-A POSTROUTING -s 10.10.0.0/16 -d 10.10.0.0/16 -j RETURN
-A POSTROUTING -s 10.10.0.0/16 ! -d 224.0.0.0/4 -j MASQUERADE
```

（注意虽然源IP虽然已经被替换，但是当数据包返回的时候目的ip会被还原成10.10.15.2, 在内核有之前做SNAT的记录）

到此数据包已经交给默认网关处理了（该网关有进入Internat的路由或者直接链接在Internet上否则也是到不了8.8.8.8的）也就不能继续抓包了（当然如果你能到网关上还可以继续抓包）。

7 模拟Kubernetes里的Service访问

这里假设Service IP为2.2.2.2 端口为88 对应到2个mydocker machine的10.10.63.3.12345和10.10.14.4.12345, 对外暴露的Node port为30001（如下标黑的规则可以不加）。

```
# 创建相关子链
$ sudo iptables -t nat -N Test-Service
$ sudo iptables -t nat -N Test-SVC-1
$ sudo iptables -t nat -N Test-SEP-1
$ sudo iptables -t nat -N Test-SEP-2
$ sudo iptables -t nat -N Test-Mark
$ sudo iptables -t nat -N Test-PostRouting
$ sudo iptables -t nat -N Test-Nodeport

$ sudo iptables -t nat -A Test-Mark -j MARK --set-mark 0x0001/0x0001
$ sudo iptables -t nat -A OUTPUT -j Test-Service
# 在Test-Service链中添加test-service1分支
$ sudo iptables -t nat -A Test-Service -d 2.2.2.2/32 -p tcp -m tcp -m comment --comment
“test-service1” -m tcp --dport 88 -j Test-SVC-1

# 在Test-Service链中添加nodeport跳转
$ sudo iptables -t nat -A Test-Service -m addrtype --dst-type LOCAL -j Test-Nodeport



# docker 10.10.63.2
$ sudo iptables -t nat -A Test-SEP-1 -s 10.10.63.2/32 -j Test-Mark
$ sudo iptables -t nat -A Test-SEP-1 -p tcp -m comment --comment "testservice1 ep1" -m tcp
-j DNAT --to-destination 10.10.63.2:12345

# docker 10.10.14.2
$ sudo iptables -t nat -A Test-SEP-2 -s 10.10.14.2/32 -j Test-Mark
$ sudo iptables -t nat -A Test-SEP-2 -p tcp -m comment --comment "testservice1 ep2" -m tcp
-j DNAT --to-destination 10.10.14.2:12345

# Test-SVC-1负载均衡
$ sudo iptables -t nat -A Test-SVC-1 -m comment --comment "test-service1" -m statistic
--mode random --probability 0.5 -j Test-SEP-1
$  sudo iptables -t nat -A Test-SVC-1 -m comment --comment "testservice1" -j Test-SEP-2
```

 # nodeport 链跳转规则
$ sudo iptables -t nat -A Test-Nodeport -p tcp -m tcp --dport 30001 -j Test-SVC-1
#如果不加下面这条那从外面范围该30001端口将不通,或者在容器里将无法访问service ip
$ sudo iptables -t nat -A PREROUTING -j Test-Service

 # PostRouting链MASQUERADE
$ sudo iptables -t nat -A POSTROUTING -j Test-PostRouting
$ sudo iptables -t nat -A Test-PostRouting -m mark --mark 0x0001/0x0001 -j MASQUERADE
$ sudo iptables -t nat -A POSTROUTING ! -s 10.10.0.0/16 -d 10.10.0.0/16 -j MASQUERADE

$ curl 2.2.2.2:88/hello

代码：
https://github.com/zhunzhun1988/mydocker