作业五

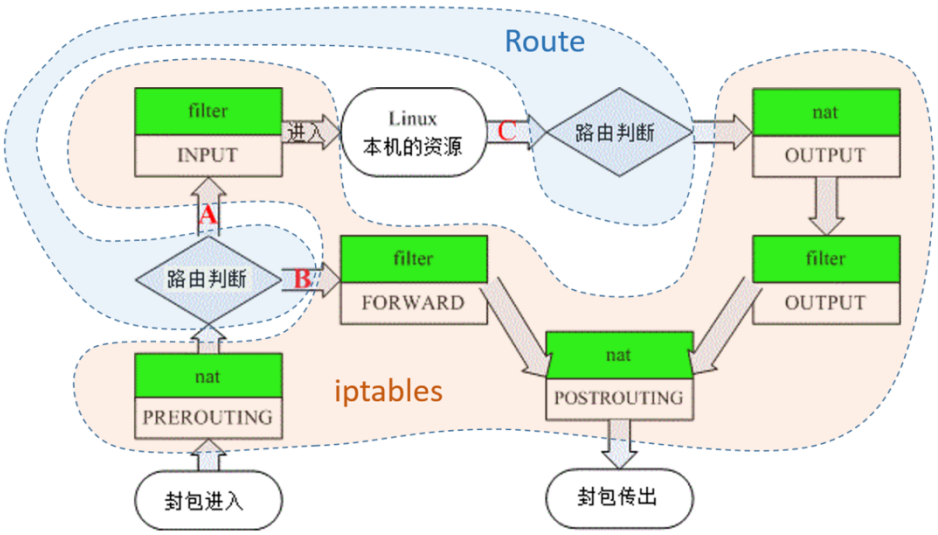
学号：1300013022

姓名：武守北

一、描述 Linux 内核如何对 IP 数据包进行处理

Linux 内核通过 Netfilter 进行 IP 数据包处理，Netfilter 是 Linux 操作系统核心层内部的一个数据包处理框架，可以在 Linux 内核中过滤、修改和封装数据包，Netfilter 在 IP 报文处理流程中插入 5 个挂载点来，可以在挂载点注册处理数据包的回调函数，当数据包进入 Linux 内核经过挂载点的时候，会执行回调函数来处理数据包。Netfilter 更准确地讲是 Linux 内核中，一个包过滤框架，默认地，它在这个框架上实现了包过滤、状态检测、网络地址转换和包标记等多种功能，因为它设计的开放性，任何有内核开发经验的开发人员，也可以很容易地利用它提供接口，在内核的数据链路层、网络层，实现自己的功能模块。Netfilter 为多种网络协议各提供了一套钩子函数，在内核态提供数据包过滤、内容修改等功能；netfilter 向用户态暴露了一些接口，用户态程序可以通过它们来控制内核的包路由规则。其中，iptables 就是一个为用户提供 IP 数据包管理功能的用户态程序。

一个数据包被处理的具体流程为：



1、具体流程：

netfilter 一共有五个位置提供 hook，每个 hook 对应 iptables 的一个 chain（即对每个 IP 包在这里的一系列处理流程），分别是：

PREROUTING：在 IP 包刚刚进入三层处理时（因此所有进入的包都会经过）；

INPUT：在 IP 包被确定是发给本机之后，在被本机更高层处理之前；

FORWARD：在 IP 包被确定需要继续转发并被路由后；

OUTPUT：在本机生成的 IP 包产生、路由之后；

POSTROUTING：在 IP 包完成所有处理，交由二层之前（因此所有要从网络设备上出去的包都会经过）；

2、回调函数返回值：

注册到挂载点的回调函数，可以获取完整的数据包，对数据包进行操作，回调函数需要一个返回值

NF_ACCEPT：继续正常的报文处理；

NF_DROP： 将报文丢弃；

NF_STOLEN：由钩子函数处理了该报文，不要再继续传送；

NF_QUEUE：将报文入队，通常交由用户程序处理；

NF_REPEAT：再次调用该钩子函数。

3、基本操作：

1、在 Linux 内核中利用 Netfilter 解析数据包，自适应多层 VLan 和 PPPoe 网络环境；

2、对特定 DNS 域名的请求数据包进行过滤

3、解析 IP 层数据头部

4、对特定端口的数据包进行过滤

5、解析 HTTP 请求和返回数据包，对特定 Host、URI、文件下载的数据包进行过滤

6、在以太网环境中，对数据包进行处理

7、在网桥环境中，对经过网桥的数据包进行解析

为了进一步方便用户使用，iptables 还提供更高层抽象的 table，每个 table 会在多个 chain 中安装处理规则，之间互相合作提供对应的功能。

上图中：

route

route 是根据路由表进行路由判断，得到下一节点的地址

iptables

iptables 是用来进行 IP 数据包 filter 和 NAT 的工具，由 filter 表和 NAT 表组成

filter 表

有三种内建链：

INPUT：处理输入的数据

OUTPUT：处理产生的要输出的数据

FORWARD：转发数据到本机的其他设备

NAT 表

有三种内建链：

OUTPUT：处理产生的要输出的数据包

PREROUTING：处理路由之前的数据包，转换 IP 数据包中的目标 IP 地址

POSTROUTING：处理经过路由之后的数据包，转换 IP 数据包中的源 IP 地址

二、在服务器上使用 iptables 分别实现如下功能并测试：

1、拒绝来自某一特定 IP 地址的访问；

2、拒绝来自某一特定 mac 地址的访问；

3、只开放本机的 http 服务，其余协议与端口均拒绝；

4、拒绝回应来自某一特定 IP 地址的 ping 命令。

语法规则：

```
[root@www ~]# iptables [-AI 链名] [-io 网络接口] [-p 协议] \
> [-s 来源IP/网域] [-d 目标IP/网域] -j [ACCEPT|DROP|REJECT|LOG]
选项与参数:
-AI 链名: 针对某的链进行规则的 "插入" 或 "累加"
    -A : 新增加一条规则，该规则增加在原本规则的最后面。例如原本已经有四条规则，
         使用 -A 就可以加上第五条规则。
    -I : 插入一条规则。如果没有指定此规则的顺序，默认是插入变成第一条规则。
         例如原本有四条规则，使用 -I 则该规则变成第一条，而原本四条变成 2~5 号
    链 : 有 INPUT, OUTPUT, FORWARD 等，此链名称又与 -io 有关。

-io 网络接口: 设定封包进出的接口规范
    -i : 封包所进入的那个网络接口，例如 eth0, lo 等接口。需与 INPUT 链配合;
    -o : 封包所传出的那个网络接口，需与 OUTPUT 链配合;

-p 协定: 设定此规则适用于哪种封包格式
    主要的封包格式有: tcp, udp, icmp 及 all 。

-s 来源 IP/网域: 设定此规则之封包的来源项目，可指定单纯的 IP 或包括网域，例如:
    IP  : 192.168.0.100
    网域: 192.168.0.0/24, 192.168.0.0/255.255.255.0 均可。
    若规范为『不许』时，则加上 ! 即可，例如:
    -s ! 192.168.100.0/24 表示不许 192.168.100.0/24 之封包来源;

-d 目标 IP/网域: 同 -s ，只不过这里指的是目标的 IP 或网域。

-j : 后面接动作，主要的动作有接受(ACCEPT)、丢弃(DROP)、拒绝(REJECT)及记录(LOG)
```

1、拒绝来自某一特定 IP 地址的访问：

```
iptables -A INPUT -s 123.107.165.198 -j REJECT
```

查看 INPUT 表：

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  123.107.165.198      0.0.0.0/0           reject-with icmp-port-
unreachable
```

成功；

2、拒绝来自某一特定 mac 地址的访问：

语法规则：

```
iptables -A INPUT -m mac --mac-source aa:bb:cc:dd:ee:ff \
> -j ACCEPT
选项与参数:
```

```
--mac-source ： 来源主机的 MAC
```

使用：

```
iptables -A INPUT -m mac --mac-source 02:00:0d:4c:00:03 -j REJECT
```

查看 INPUT 表：

```
Chain INPUT (policy ACCEPT)
num  target     prot opt source               destination
1    REJECT     all  --  anywhere             anywhere             MAC
02:00:0D:4C:00:03 reject-with icmp-port-unreachable
```

成功；

3、只开放本机的 http 服务，其余协议与端口均拒绝；

只打开 tcp 协议的 80 端口：

```
# 允许 http 服务
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
# 关闭其他服务
iptables -P INPUT DROP
```

查看 INPUT 表：

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            0.0.0.0/0           tcp dpt:80
```

成功；

4、拒绝回应来自某一特定 IP 地址的 ping 命令。

语法规则：

```
iptables -A INPUT [-p icmp] [--icmp-type 类型] -j ACCEPT
```

实现：

```
iptables -A INPUT -p icmp --icmp-type 8 -s 172.16.1.33 -j REJECT
```

此时 1003 无法 ping 1002，反之 1002 可以 ping 1003。

三、解释 Linux 网络设备工作原理，至少介绍 bridge, vlan, veth 工作过程

Bridge：

Bridge（桥）是 Linux 上用来做 TCP / IP 二层协议交换的设备，与现实世界中的交换机功能相似。Bridge 设备实例可以和 Linux 上其他网络设备实例连接，既 attach 一个从设备，类似于在现实世界中的交换机和一个用户终端之间连接一根网线。当有数据到达时，Bridge 会根据报文中的 MAC 信息进行广播、转发、丢弃处理。

bridge 还隐式地包含一个和本机连接的端口，在本机上就显示为对应的 bridge 网络设备，例如 bridge0。在三层上，每一个 attach 到 bridge 的设备都要把 IP 地址“交给”bridge 管理。若 attach 在上面的设备依然拥有地址，网络包就无法进入 kernel 并传给 bridge 了。此时需要 bridge 来响应 ARP 请求，而不是 attach 在上面的设备。

Vlan：

VLAN 又称虚拟网络，是一个被广泛使用的概念，有些应用程序把自己的内部网络也称为 VLAN。此处主要说的是在物理世界中存在的，需要协议支持的 VLAN。它的种类很多，按照协议原理一般分为：MACVLAN、802.1.q VLAN、802.1.qbg VLAN、802.1.qbh VLAN。其中出现较早，应用广泛并且比较成熟的是 802.1.q VLAN，其基本原理是在二层协议里插入额外的 VLAN 协议数据（称为 802.1.q VLAN Tag），同时保持和传统二层设备的兼容性。Linux 里的 VLAN 设备是对 802.1.q 协议的一种内部软件实现，模拟现实世界中的 802.1.q 交换机。

vlan 设备的母设备（例如 eth0）下面由多个子设备，每个对应一个 VLAN ID，就像把 VLAN 交换机上的端口按照 PVID 成组，每一组对应 vlan 设备的

一个子设备。在母设备 eth0 收到 Ethernet 数据帧时，它开始检查 VLAN tag。若没有 VLAN tag，则直接从 eth0 传递给 kernel 三层。若含有 VLAN tag（例如 3），则传递给 eth0.3 接收。从子设备（例如 eth0.3）要发送数据时，实际上会从 eth0 发出，但发出之前会打上 VLAN tag 3。总的来说，就是给 Linux Ethernet 网络设备增加了收发带 VLAN tag 的 Ethernet 帧的能力，使其能够配合上层带 VLAN 功能的交换机、路由器使用。

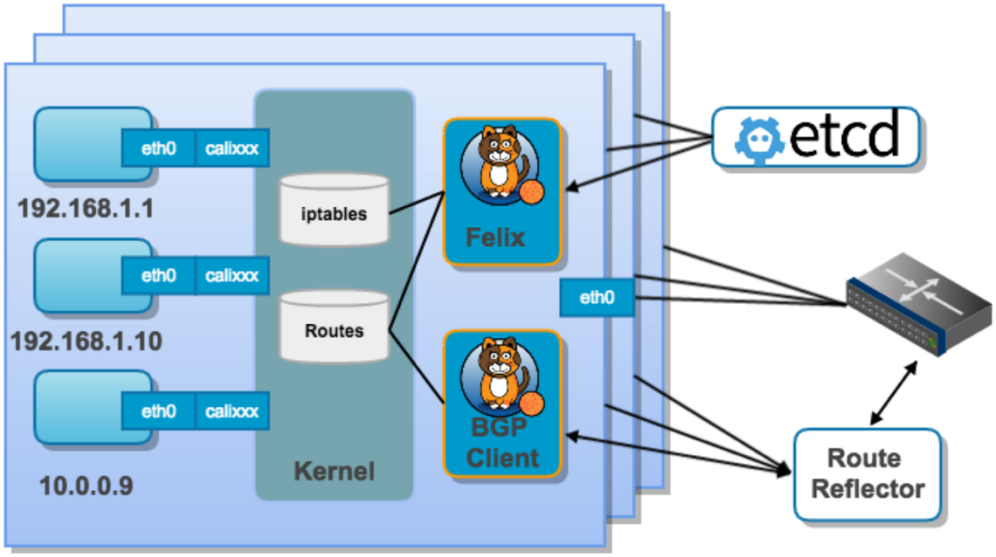
Veth：

VETH 即 virtual Ethernet，是一种 Linux 专门为容器技术设计的网络通信机制。它的作用是把从一个 network namespace 发出的数据包转发到另一个 namespace。VETH 设备总是成对出现的，一个是 container 之中，另一个在 container 之外，即在真实机器上能看到的。它能够反转通讯数据的方向，需要发送的数据会被转换成需要收到的数据重新送入内核网络层进行处理，从而间接的完成数据的注入。

四、说明在 calico 容器网络中，一个数据包从源容器发送到目标容器接收的具体过程：

Calico 架构：

calico 是纯三层的 SDN 实现，它基于 BPG 协议和 Linux 自身的路由转发机制，不依赖特殊硬件，容器通信也不依赖 iptables NAT 或 Tunnel 等技术。能够方便的部署在物理服务器、虚拟机（如 OpenStack）或者容器环境下。同时 calico 自带的基于 iptables 的 ACL 管理组件非常灵活，能够满足比较复杂的安全隔离需求。



如图，Calico 组件包含:

Felix：Calico agent，跑在每台需要运行 workload 的节点上，主要负责配置路由及 ACLs 等信息来确保 endpoint 的连通状态；

etcd：分布式键值存储，主要负责网络元数据一致性，确保 Calico 网络状态的准确性；

BGP：Client(BIRD)，主要负责把 Felix 写入 kernel 的路由信息分发到当前 Calico 网络，确保 workload 间的通信的有效性；

BGP Route Reflector(BIRD)，大规模部署时使用，摒弃所有节点互联的 mesh 模式，通过一个或者多个 BGP Route Reflector 来完成集中式的路由分发；

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 实现了一个高效的 vRouter 来负责数据转发而每个 vRouter 通过 BGP 协议负责把自己上运行的 workload 的路由信息像整个 Calico 网络内传播 – 小规模部署可以直接互联，大规模下可通过指定的 BGP route reflector 来完成。

这样保证最终所有的 workload 之间的数据流量都是通过 IP 包的方式完

成互联的。

当容器创建时，calico 为容器生成 veth pair，一端作为容器网卡加入到容器的网络命名空间，并设置 IP 和掩码，一端直接暴露在宿主机上，并通过设置路由规则，将容器 IP 暴露到宿主机的通信路由上。于此同时，calico 为每个主机分配了一段子网作为容器可分配的 IP 范围，这样就可以根据子网的 CIDR 为每个主机生成比较固定的路由规则。

当容器需要跨主机通信时，主要经过下面的简单步骤：

1、容器流量通过 veth pair 到达宿主机的网络命名空间上。

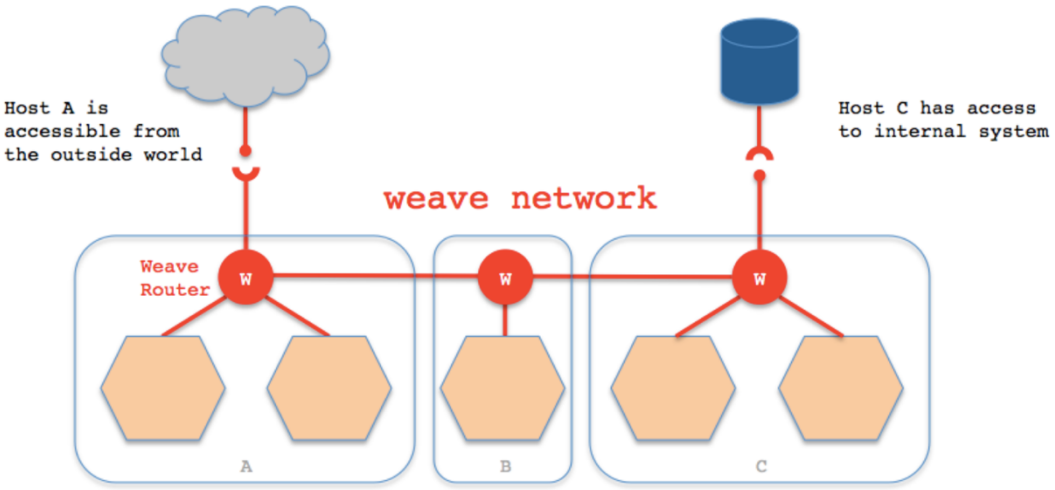
2、根据容器要访问的 IP 所在的子网 CIDR 和主机上的路由规则，找到下一跳要到达的宿主机 IP。

3、流量到达下一跳的宿主机后，根据当前宿主机上的路由规则，直接到达对端容器的 veth pair 插在宿主机的一端，最终进入容器。

五、调研除 calico 以外的一种容器网络方案，比较其与 calico 的优缺点：

1、weave 架构：

weave 通过在 docker 集群的每个主机上启动虚拟的路由器，将主机作为路由器，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。其主机网络拓扑参见下图：



Hosts A, B & C running the containers shown in previous figure

在每一个部署 Docker 的主机（可能是物理机也可能是虚拟机）上都部署有一个 W（即 weave router，它本身也可以以一个容器的形式部署）。weave 网络是由这些 weave routers 组成的对等端点（peer）构成，并且可以通过 weave 命令行来定制网络拓扑。

每个部署了 weave router 的主机之间都会建立 TCP 和 UDP 两个连接，保证 weave router 之间控制面流量和数据面流量的通过。控制面由 weave routers 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。控制面的通信可以被配置为加密通信。而数据面由 weave routers 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。

对每一个 weave 网络中的容器，weave 都会创建一个网桥，并且在网桥和每个容器之间创建一个 veth pair，一端作为容器网卡加入到容器的网络命名空间中，并为容器网卡配置 ip 和相应的掩码，一端连接在网桥上，最终通过宿主机上 weave router 将流量转发到对端主机上。其基本过程如下：

容器流量通过 veth pair 到达宿主机上 weave router 网桥上。

weave router 在混杂模式下使用 pcap 在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以及宿

主机和本地容器之间的流量。捕获的包通过 UDP 转发到所其他主机的 weave router 端。

在接收端，weave router 通过 pcap 将包注入到网桥上的接口，通过网桥的上的 veth pair，将流量分发到容器的网卡上。

2、Calico 与 Weave 比较

Calico 优势:

基于 iptable/linux kernel 包转发效率高，损耗低。

容器间网络三层隔离。

网络拓扑直观易懂，平行式扩展，可扩展性强。

Weave 优势:

支持主机间通信加密。

支持跨主机多子网通信

Calico 劣势:

Calico 仅支持 TCP, UDP, ICMP andICMPv6 协议。

Calico 没有加密数据路径。 用不可信网络上的 Calico 建立覆盖网络是不安全的。

没有 IP 重叠支持。

Weave 劣势:

网络封装是一种传输开销，对网络性能会有影响，不适用于对网络性能要求高的生产场景。

六、编写一个 mesos framework，使用 calico 容器网络自动搭建一个 docker 容器集群（docker 容器数量不少于三个），并在其中一个容器中部署 jupyter notebook。运行后外部主机可以通过访问宿主机 ip+8888 端口使用 jupyter notebook 对 docker 集群进行管理。

在三台服务器上完成以下操作：

1、搭建 etcd 服务器：

```
sudo apt install etcd
sudo service etcd stop
```

2、配置集群：

```
etcd --name node0 --initial-advertise-peer-urls http://192.168.0.111:2380 \
--listen-peer-urls http://192.168.0.111 \
--listen-client-urls http://192.168.0.111:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://192.168.0.111:2379 \
--initial-cluster-token etcd-cluster-hw5 \
--initial-cluster node0=http://192.168.0.111:2380,node1=http://192.168.0.111:2380,node2=http://
192.168.0.111:2380 \
--initial-cluster-state new
```

3、重启 docker：

```
sudo service docker stop
sudo dockerd --cluster-store etcd:// 192.168.0.111&
```

4、安装 calico：

```
wget -O /usr/local/bin/calicoctl
https://github.com/projectcalico/calicoctl/releases/download/v1.1.3/calicoctl
chmod +x /usr/local/bin/calicoctl
calicoctl node run --ip 192.168.0.111 --name node0
```

查看状态：

```
sudo    calicoctl node status
Calico process is running.
```

```
IPv4 BGP status
+--------------+-------------------+-------+----------+-------------+
| PEER ADDRESS |         PEER TYPE         | STATE |    SINCE      |        INFO         |
+--------------+-------------------+-------+----------+-------------+
| 192.168.0.111| node-to-node mesh | up        | 16:34:13 | Established |
| 192.168.0.110| node-to-node mesh | up        | 16:34:13 | Established |
+--------------+-------------------+-------+----------+-------------+

IPv6 BGP status
No IPv6 peers found.
```

5、反向代理：

```
sudo apt install -y npm nodejs-legacy
sudo npm install -g configurable-http-proxy

sudo nohup configurable-http-proxy --default-target=http:// 192.168.0.111:2379 --ip=192.168.0.111 --port=8888 >> /dev/null 2>&1 &
```

6、创建 docker 镜像：

```
FROM ubuntu

RUN apt-get update
RUN apt-get install -y ssh
RUN useradd -ms /bin/bash hw5
RUN adduser hw5 sudo
RUN echo 'hw5:hw5' | chpasswd
RUN mkdir /var/run/sshd
USER hw5
WORKDIR /home/hw5

CMD ["/usr/sbin/sshd", "-D"]
```

7、创建 jupyter 镜像

```
FROM ubuntu

RUN apt-get update
RUN apt-get install -y ssh
```

```
RUN apt update
RUN apt install -y --fix-missing apt-utils sudo python3-pip ssh

RUN pip3 install --upgrade pip
RUN pip3 install jupyter

RUN useradd -ms /bin/bash hw5
RUN adduser hw5 sudo
RUN echo 'hw5:hw5' | chpasswd
RUN mkdir /var/run/sshd
USER hw5
WORKDIR /home/hw5

CMD ["/usr/local/bin/jupyter", "notebook", "--NotebookApp.token=", "--ip=0.0.0.0", "--port=8888"]
```

8、创建 jupyter 镜像：

```
FROM ubuntu

RUN apt-get update
RUN apt-get install -y ssh

RUN apt update
RUN apt install -y --fix-missing apt-utils sudo python3-pip ssh

RUN pip3 install --upgrade pip
RUN pip3 install jupyter

RUN useradd -ms /bin/bash hw5
RUN adduser hw5 sudo
RUN echo 'hw5:hw5' | chpasswd
RUN mkdir /var/run/sshd
USER hw5
WORKDIR /home/hw5

CMD ["/usr/local/bin/jupyter", "notebook", "--NotebookApp.token=", "--ip=0.0.0.0", "--port=8888"]
```

9、在一台主机上使用 scheldule.py，可以完成对于 jupyter notebook 的访问。

```
#!/usr/bin/env python2.7
from __future__ import print_function

import sys
import uuid
```

```
import time
import socket
import signal
import getpass
from threading import Thread
from os.path import abspath, join, dirname

from pymesos import MesosSchedulerDriver, Scheduler, encode_data, decode_data
from addict import Dict

TASK_CPU = 1
TASK_MEM = 32



class MinimalScheduler(Scheduler):

    def __init__(self):
        self.count = 0;

    def resourceOffers(self, driver, offers):
        filters = {'refuse_seconds': 5}
        for offer in offers:
            if self.count>4:
                break
            cpus = self.getResource(offer.resources, 'cpus')
            mem = self.getResource(offer.resources, 'mem')
            if cpus < TASK_CPU or mem < TASK_MEM:
                continue


            #设置 DockerInfo 与 Command
            if self.count==0:
              ip = Dict();
              ip.key = 'ip'
              ip.value = '192.168.0.100'
              NetworkInfo = Dict();
              NetworkInfo.name = 'calico_net'
              DockerInfo = Dict()
              DockerInfo.image = 'docker-jupyter'
              DockerInfo.network = 'USER'
              DockerInfo.parameters = [ip]
              ContainerInfo = Dict()
              ContainerInfo.type = 'DOCKER'
              ContainerInfo.docker = DockerInfo
```

```
                    ContainerInfo.network_infos = [NetworkInfo]
                    CommandInfo = Dict()
                    CommandInfo.shell = False
                    CommandInfo.value = 'jupyter'
                    CommandInfo.arguments = ['notebook', '--ip=0.0.0.0', '--NotebookApp.token=zzw', '--
port=8888']
                else:
                    ip = Dict()
                    ip.key = 'ip'
                    ip.value = '192.168.0.10' + str(self.count)
                    NetworkInfo = Dict()
                    NetworkInfo.name = 'calico_net'
                    DockerInfo = Dict()
                    DockerInfo.image = 'docker-ssh'
                    DockerInfo.network = 'USER'
                    DockerInfo.parameters = [ip]
                    ContainerInfo = Dict()
                    ContainerInfo.type = 'DOCKER'
                    ContainerInfo.docker = DockerInfo
                    ContainerInfo.network_infos = [NetworkInfo]
                    CommandInfo = Dict()
                    CommandInfo.shell = False
                    CommandInfo.value = '/usr/sbin/sshd'
                    CommandInfo.arguments = ['-D']
                task = Dict()
                task_id = 'task_'+str(self.count);
                task.task_id.value = task_id
                task.agent_id.value = offer.agent_id.value
                task.name = 'hw5'

                task.container = ContainerInfo;
                task.command = CommandInfo;
                task.resources = [
                        dict(name='cpus', type='SCALAR', scalar={'value': TASK_CPU}),
                        dict(name='mem', type='SCALAR', scalar={'value': TASK_MEM}),
                ]

                self.count = self.count +1;
                driver.launchTasks(offer.id, [task], filters)

    def getResource(self, res, name):
            for r in res:
                if r.name == name:
                    return r.scalar.value
```

```
            return 0.0

        def statusUpdate(self, driver, update):
            logging.debug('Status update TID %s %s',
                          update.task_id.value,
                          update.state)



def main(master):
        framework = Dict()
        framework.user = getpass.getuser()
        framework.name = "mynginx"
        framework.hostname = socket.gethostname()

        driver = MesosSchedulerDriver(
            MinimalScheduler(),
            framework,
            master,
            use_addict=True,
        )

        def signal_handler(signal, frame):
            driver.stop()

        def run_driver_thread():
            driver.run()

        driver_thread = Thread(target=run_driver_thread, args=())
        driver_thread.start()

        print('Scheduler running, Ctrl+C to quit.')
        signal.signal(signal.SIGINT, signal_handler)

        while driver_thread.is_alive():
            time.sleep(1)



if __name__ == '__main__':
        import logging
        logging.basicConfig(level=logging.DEBUG)
        if len(sys.argv) != 2:
            print("Usage: {} <mesos_master>".format(sys.argv[0]))
            sys.exit(1)
```

```
    else:
        main(sys.argv[1])
```